Procedia Computer Science

Volume 29, 2014, Pages 1113–1122

ICCS 2014. 14th International Conference on Computational Science

Application-Level Performance Optimization: A Computer Vision Case Study on STHORM

Vítor Schwambach[1,2], Sébastien Cleyet-Merle[2], Alain Issard[2], and Stéphane Mancini[1]

[1] Univ. Grenoble Alpes, TIMA, F-38031 Grenoble, France
CNRS, TIMA, F-38031 Grenoble, France
stephane.mancini@imag.fr
[2] STMicroelectronics, Grenoble, France
vitor.schwambach@st.com, sebastien.cleyet-merle@st.com, alain.issard@st.com

Abstract
Computer vision applications constitute one of the key drivers for embedded many-core architectures. In order to exploit the full potential of such systems, a balance between computation and communication is critical, but many computer vision algorithms present a highly data-dependent behavior that complexifies this task. To enable application performance optimization, the development environment must provide the developer with tools for fast and precise application-level performance analysis. We describe the process to port and optimize a face detection application onto the STHORM many-core accelerator using the STHORM OpenCL SDK. We identify the main factors that limit performance and discern the contributions arising from: the application itself, the OpenCL programming model, and the STHORM OpenCL SDK. Finally, we show how these issues can be addressed in the future to enable developers to further improve application performance.

Keywords: many-core, embedded vision, parallelism, performance, optimization.

1 Introduction

For many years, the performance improvements stated by Moore's Law[11] have been achieved through a combination of device, architecture and compiler advances. In the race to scale single-core processor performance, architects have crashed into the so-called *power wall*[2]. Profiting from transistor count increases, the shift to more energy efficient multi- and many-core designs aims to continue the proportional scaling of performance within a fixed power envelope. This trend is also observed on embedded processors for battery powered mobile devices – where low power consumption is key – and ITRS predicts that the core count for multiprocessor systems will continue to increase in the near future by 1.4x per year[6].

doi:10.1016/j.procs.2014.05.100

Applications drive the development of new devices[6], and one of the drivers for embedded many-core architectures are computer vision applications[17]. Embedded vision algorithms require vast amounts of computational power and possess a high degree of available parallelism that many-core architectures can exploit to achieve high performance.

In order to achieve high processor utilization, a balance between communication and computation is necessary[4]. As the core count increases, the necessary memory bandwidth is higher and even highly compute-bound algorithms can become memory-bound on many-core architectures. Embedded vision algorithms are no exception, as they process video streams in real-time and require high memory bandwidth.

A further challenge to achieve a good parallel efficiency is the load balancing. Some computer vision applications present very data-dependent behavior which can negatively impact their performance on many-core devices. In [15], the author evaluates the performance of a face detection application written in OpenCL on a GPGPU system. He shows how parallel performance on the GPU is impacted by the data-dependent behavior and that some classification steps present a higher performance when executed on the host processor.

Although GPU architectures have a high number of processing elements, their compute units are SIMD (single instruction, multiple data), what leads to branch divergence penalties in data-dependent algorithms. Many-core architectures such as Kalray's Multi-Purpose Processor Array[5] and STMicrolectronic's STHORM[12] are composed of MIMD (multiple instruction, multiple data) processor clusters that better cope with data-dependent algorithms[10].

In this work, we port a face detection application onto OpenCL for the STHORM many-core accelerator, and follow a methodology for application performance optimization with the STHORM OpenCL SDK. Our goal is to determine the best configuration in the context of the algorithm-architecture co-design of a new embedded system. We identify the key limiting factors for application-level performance optimization, and break them down into three categories: those inherent to the application behavior, those due to the OpenCL programming model, and those arising from the STHORM environment. We then propose modifications to the OpenCL runtime and STHORM tools to minimize these issues.

In Section 2 we present an overview of the STHORM many-core accelerator architecture and the OpenCL programming model. In Section 3 we detail the porting and performance optimization of the face detection application onto the STHORM platform, and present the results in Section 4. In Section 5 we list the limitations for application-level performance optimization and discuss how these can be overcome. Finally, in Section 6 we conclude the paper and show how we plan to address the aforementioned limitations.

2 STHORM

STHORM[12] is a many-core processor designed by STMicroelectronics to handle compute intensive embedded applications. It is derived from Platform2012[3], a joint effort between STMicroelectronics and CEA (French Alternative Energies and Atomic Energy Comission), and can be used either as a stand-alone processor or as an accelerator coupled to a host processor.

2.1 Architecture

STHORM has a scalable architecture, organized as clusters of processing elements(PEs), configurable from 1 to 4 clusters with up to 16 processing elements each. Figure 1 shows a high-level block diagram of the STHORM architecture.

The clusters are interconnected via a network-on-chip (NoC) and have integrated dynamic

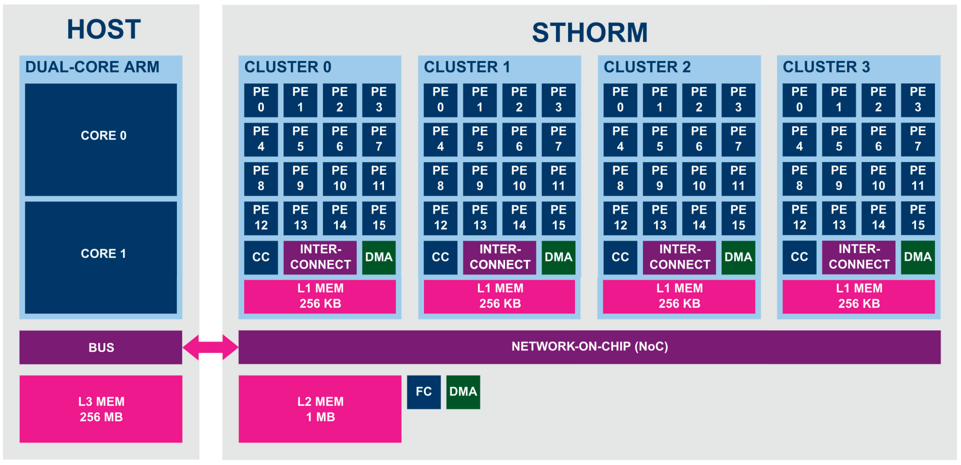

Figure 1: STHORM Architecture Block Diagram

frequency and voltage scaling (DFVS) capabilities that can be controlled on a per-cluster granularity. The cluster's processing elements are dual-issue STxP70 processors – an in-order 32-bit RISC processor – with a floating point unit. One additional STxP70 processor acts as a dedicated cluster controller.

Internally, each cluster counts with 256KB of shared memory accessible by all processors in the cluster. In order to reduce the probability of conflict, the memory is organized in 32 banks with address interleaving. The logarithmic interconnect, with a mesh-of-trees (MoT)[14] topology, provides concurrent single-cycle memory access. In case of conflict – when two or more processors access a same bank simultaneously –, a single request is serviced per cycle, while the remaining requests remain pending. Although processors block until their request is serviced, a round-robin mechanism ensures fair access to the contended resources.

2.2 OpenCL Programming Model

STHORM supports three parallel programming models at different abstraction levels[13], among which OpenCL 1.1[7]. It has been originally developed for heterogeneous GP-GPU applications, but it is increasingly being used for programming embedded multi- and many-core processors as well. It is based on the concept that the application runs on a host processor and offloads computation kernels to a many-core compute device. The compute device is composed of a number of compute units, each counting with numerous processing elements.

The kernel's workload is partitioned in work-groups composed of a number of work-items. The OpenCL runtime schedules the execution of the work-groups and work-items on the compute units and their processing elements. Several synchronization mechanisms are available, such as barriers, locks and atomic operations.

Four data address spaces exist: global, constant, local, and private. The STHORM OpenCL runtime allocates global data buffers on the L3 host memory, while constant data are placed on the L2 memory. Local and private data are placed in the L1, the cluster's shared memory. The OpenCL API specifies asynchronous work-group copy functions to transfer data between such buffers, where a DMA transfer is launched for the entire work-group. STHORM's OpenCL implementation adds the concept of a work-item copy, where individual work-items launch DMA transfers autonomously.

3 Face Detection Performance Optimization

In this case study, we start from a sequential face detection application and show how to derive a parallel implementation. Then experiments are done in order to optimize the application's performance on the STHORM architecture.

3.1 Application Description

The sequential face detection algorithm used as a starting point for the study is based on the Viola and Jones approach[16], where the detector consists of a classifier cascade of Haar-like features, trained using AdaBoost. The classifier is evaluated at regular intervals using a scanning window technique over an image pyramid to achieve scale invariance. At each image pyramid level, the detector builds an integral image whose purpose is to accelerate feature computation.

The classifier cascade is organised as a series of stages. Each stage has a set of features from which a response is computed and tested against lower and upper bounds. If the stage response falls within bounds, the classifier proceeds to evaluate the next stage, otherwise, the detection is aborted and the window is rejected. If all stages in the classifier cascade are successful, the window is accepted and a positive detection result is reported. Neighboring detections are merged based on a distance metric and are assigned a score based on the number of single detections merged. A final filtering step discards detections with a low score.

3.2 Methodology

It is known from Amdahl's law that the speed-up of a parallel application is limited by the execution time of its sequential portion[1]. As such, we seek to parallelize the portions of the application which contribute the most to the execution time.

The first step is to identify the application's hotspots, which is achieved by profiling the reference sequential application with an STxP70 cycle-approximate simulator and ranking the functions in descending order of their cumulative execution time. The top ranked functions are our initial candidates for parallelization.

These functions are refactored into OpenCL kernels with clear inputs and outputs. Then, we need to dimension the parallel workload so as to overlap computation and communication, and optimise the load balance. Some key parallel implementation decisions must be made at this point, such as:

- the parallel granularity, e.g.: image frame, line, window, or pixel;
- the workload distribution strategy: static or dynamic workload distribution;
- the working data placement: global, local or private spaces;
- the data transfer strategy: individualistic or collaborative.

The STHORM OpenCL simulator and runtime can also be parameterized with different configurations in terms of the number of physical clusters and processing elements of the target platform, as well as the OpenCL kernel's local and global work-group dimensions.

This performance optimization flow is iterative, with simulations done to estimate the impact of different design choices in the execution time. Its goal is to determine the best architectural parameters and algorithm parallelization strategy in order to meet the functional and non-functional requirements of the application. Therefore, the simulator needs to be fast enough to allow iterative design space exploration, and precise enough to allow comparison of different alternatives and to ensure that the final implementation will meet the application requirements.

3.3 Hotspot Analysis

In order to identify the hotspots in the face detection application, we have profiled the reference sequential implementation with a cycle-approximate STxP70v4 simulator. The analysis is based on a worst-case QVGA image with 24 faces, designated herein as the test image.

Table 1 reports the profiling results for the test image, grouped by algorithmic phase and ranked according to their cumulative execution cycles. The three hotspots identified are: the classifier cascade, the integral image generation, and the scaler. These three phases together account for $\sim 95\%$ of the execution time, and were thus selected as candidates for parallelization.

Table 1: Profiling results for the sequential face detection application on the worst-case image (24 faces) in our testing database. The results are obtained on a cycle-approximate STxP70v4 simulator in a dual-issue configuration and for a clock frequency of 500 MHz.

Application Phase	Cycles	Time (ms)	% of Total
Detection Cascade	61,879,829	123.8	56.8%
Integral Image	27,159,728	54.3	24.9%
Scaler	14,627,913	29.3	13.4%
Other	5,196,975	10.4	4.8%
Total	108,864,445	217.7	100.0%

3.4 Parallel Implementation in OpenCL

3.4.1 Partitioning into OpenCL Kernels

Once the parallelization candidates have been selected, we refactor them into OpenCL kernels. Each kernel execution processes a single image pyramid iteration, with multiscale detection requiring successive executions of the kernels.

Since OpenCL allows no local or private data-persistence, an important factor to consider is the the bandwidth required to stream data in and out of the cluster's shared memory for each kernel. In order to exploit data locality, we merged the integral image generation into the classifier kernel, allowing us to keep the integral image in the cluster's local memory at all times, and thus reduce the bandwidth to the external memory.

3.4.2 Scaler Kernel

The scaler kernel implements a bilinear scaler and, as such, produces an output pixel by interpolating four input pixels. An outer loop processes each line of the output image, whereas an inner loop processes each pixel of a line. As the computation is well balanced and not data-dependent, a simple static workload allocation scheme of lines to work-items is used. In this scheme, a work-item is assigned a number of consecutive lines to process based on its global ID and the number of global work-items. This scheme also advantages scalability, since it seamlessly partitions the load across processing elements of all clusters.

On each outer loop iteration, a processing element will: fetch lines of the input image from global memory; produce the output line; write-back the line of the output image to global memory. In order to hide the data transfer latency, we implement double buffering on both input and output transfers, which essentially results in a software pipeline with three stages – fetch, process and write-back.

3.4.3 Classifier Kernel

The classifier kernel encompasses the integral image generation and classifier cascade execution. We have experimented with different data transfer strategies, which are detailed in the sequence.

Integral image generation. Similarly to the scaler phase, the integral image generation is not data-dependent and rather well-balanced. Many methods to compute the integral image exist, but a two-pass approach can reduce the number of operations[9] and is amenable to parallel implementation[18]. Thus, we implement the two-pass approach, where the first pass consists in an horizontal scan that accumulates the elements in each line, while the second pass consists in a vertical scan that accumulates the elements in each column. On the parallel version, a static allocation of lines and columns to processing elements is used for the first and second pass, respectively, with barriers after each pass to ensure correct synchronization.

Classifier cascade. In this phase the classifier is applied to integral image windows. Since this computation is very data-dependent and presents a highly variable execution time for different windows, a dynamic workload allocation scheme is used where work-items increment an atomic counter to determine the next window to process. In case of a positive detection result, the work-item adds an entry with the coordinates of the window to a list residing on the global memory. A global atomic counter determines the next available position on the list. Scalability is achieved by partitioning the input image in horizontal image stripes allocated to different work-groups.

3.4.4 Data Management

The classifier cascade constant data placement is also critical. The STHORM OpenCL runtime places cascade data in STHORM's L2 memory by default. Since it has higher latency and the cascades are frequently accessed, cascade data is explicitly copied to the local memory at the start of a work-group execution. This is true for both the collaborative and individualistic data transfer approaches for the Classifier Kernel, as detailed below.

Collaborative approach. In the collaborative approach, work-groups load entire horizontal image stripes via a work-group copy call, with double buffering on input to hide the latency of loading new stripes. Allocation of image stripes to work-groups is static. The integral of the entire image stripe is then computed in parallel by local work-items. A barrier call synchronizes work-items prior to starting the classifier phase, with dynamic allocation of windows in an image stripe to work-items. A second barrier call ensures all work-items are finished executing the classifier cascade on the current stripe prior to moving to the next stripe.

Individualistic approach. In the individualistic approach, each work-item fetches and processes an image window autonomously. Work-items obtain the index of the next window to process dynamically and load the window into local memory via a work-item copy call. Once the transfer is complete, the work-item generates the integral image for the window on a private buffer. Then the classifier evaluates the window and reports any successful detection. Barriers are not needed in this case, since work-items are completely autonomous. With the dependency among work-items removed, load balancing can be improved at the expense of increased data transfer and computation.

4 Results

4.1 Setup

The experiments take the form of simulation runs with the simulator in the STHORM OpenCL SDK version 2013.2. The simulator models a platform with an ARM processor as OpenCL host and the STHORM many-core accelerator as an OpenCL device. The Posix-XP70 configuration of the simulator is used, which is functional for the host, and cycle-approximate for the device. No cycle-approximate simulator for the host is available in the SDK.

A STHORM prototype board is used for comparison. It counts with an ARM host and a STHORM device fabricated in ST's 28nm process. The L3 memory is connected via a bridge with a bandwidth of 400 MB/s, while the L2 and L1 memories are integrated into STHORM. In both cases, STHORM is setup as 4 clusters of 16 processing elements running at 500MHz.

4.2 Performance Measurements

Table 2 lists the kernel time measurements for the worst-case QVGA image (24 faces) in our testing database, for both the collaborative and individualistic approaches. Simulator results are compared to those of the prototype board. The *kernel processing time* reflects effective computation time and local memory accesses, *kernel prolog and epilogue* accounts for overheads in launching and terminating kernels, while the *time spent in runtime* encompasses the asynchronous data transfer time, as well as the time spent waiting for events and on barriers.

These results show that the collaborative version is negatively impacted by the synchronization barriers, which take roughly a third of the *total time in kernels*. The individualistic version provides better overall performance at both simulator and board, mainly due to the reduced synchronization overhead. The highest source of inefficiency according to the simulator results is the kernel prolog and epilogue.

The simulator results indicate that the collaborative approach has smaller *kernel processing time* (6.9 ms) than the individualistic approach (7.8 ms). However, the prototype results show an inversion, with the collaborative approach presenting a higher *processing time* (44.9 ms) than the individualistic approach (12.4 ms). Furthermore, while the time spent *waiting for events* is close to 1% on the simulator, it can amount to nearly half of the *total kernel time* on the prototype board. Thus, although the STHORM simulator used is cycle-approximate, a large mismatch between the simulator and the prototype board results has been observed.

Table 2: Execution time for the face detection application on STHORM simulator and prototype, for 4 clusters of 16 processing elements at 500 MHz. % are relative to the total time.

	Simulator				Prototype			
Data Transfer Strategy	Collaborative		Individualistic		Collaborative		Individualistic	
Kernel Processing Time	6.9 ms	21.5%	7.8 ms	34.7%	44.9 ms	26.3%	12.4 ms	13.9%
Kernel Prolog & Epilog	13.6 ms	42.2%	14.1 ms	62.4%	47.9 ms	28.0%	32.4 ms	36.1%
Time Spent in Runtime	11.7 ms	36.3%	0.7 ms	2.9%	78.0 ms	45.6%	44.8 ms	50.0%
– Asynchronous Copies	0.3 ms	0.9%	0.3 ms	1.2%	0.0 ms	0.0%	1.6 ms	1.8%
– Waiting for Events	0.4 ms	1.1%	0.3 ms	1.2%	30.2 ms	17.7%	43.2 ms	48.2%
– Waiting on Barriers	11.1 ms	34.3%	0.1 ms	0.5%	48.3 ms	28.3%	0.0 ms	0.0%
Total Time in Kernels	32.2 ms	100.0%	22.7 ms	100.0%	170.9 ms	100.0%	89.7 ms	100.0%

4.3 Detailed Analysis

The results in Table 2 show that on the simulator the highest contributor to the total time is the time spent in the *kernel prolog and epilogue*. Figure 2 shows a partial trace visualization of the collaborative approach execution, from which it can be seen that the *kernel prolog and epilogue* accounts not only for the time to launch and terminate kernels, but to any interstices between work-group executions where the cluster is idle. These typically arise due to inter-work-group load imbalance, as the data-dependent behavior causes some work-groups to finish earlier than others, or due to the interaction with the host processor.

The *kernel processing time* results on the prototype board are higher than on the simulator. The reason is that the STHORM simulator does not accurately model memory access times, which, except for DMA transfers, are accounted for in the *kernel processing time*. The simulator does not model memory conflicts. This, together with the higher latency and limited bandwidth to the global memory on the prototype board, leads to a high mismatch between simulator and board. Moreover, as synchronization barriers require all processors to reach the barrier call to proceed, the increased processing time will cause processors on the critical path to take longer to reach the barriers, and thus lead to increased time *waiting on barriers*.

The time spent *waiting for events* is the figure with the highest mismatch between the simulator and the prototype. When launching a DMA transfer via a non-blocking asynchronous copy, an event handle is returned by the runtime. Processors can perform other operations asynchronously and then do a wait call on the event handle, which returns only when the transfer is complete. Thus, the time spent *waiting for events* in our experiments actually corresponds to the time waiting for non-blocking DMA transfers to complete. The high mismatch indicates that the simulator does not precisely model the DMA transfer times found on the prototype. No parameters are available in the STHORM SDK to compensate for this mismatch.

The total time lost due to load imbalance cannot be precisely estimated from the figures provided, since they do not discriminate among different contributing factors. Nonetheless, a large portion of the *kernel prolog and epilogue* is relative to inter-work-group imbalance, as shown in Figure 2, and the time lost due to inter- and intra-work-group load imbalance could amount to up to 70% of the *total kernel time* for the collaborative approach on the simulator.

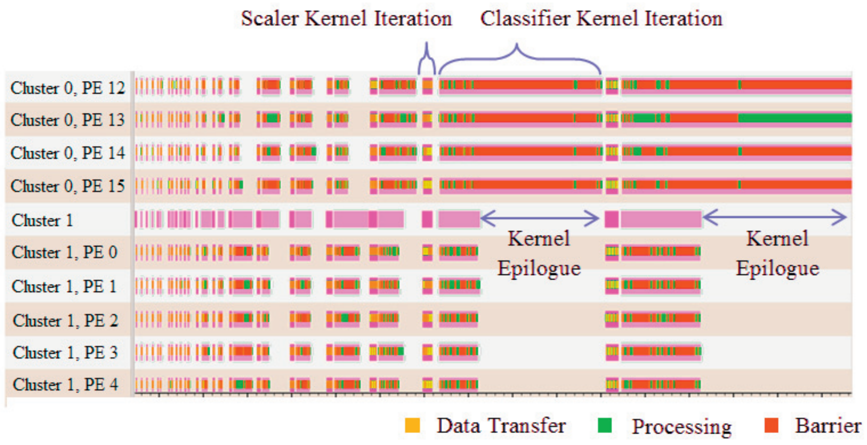

Figure 2: Portion of a trace for the face detection collaborative approach on STHORM. It shows the kernel execution traces of multiple image pyramid iterations for a single image frame.

The individualistic approach virtually eliminates intra-work-group imbalance, but still presents high inter-work-group imbalance. Even though the time spent waiting for data transfers using the individualistic approach on the board is higher, it still yields better performance than the collaborative approach, as the latter incurs higher memory conflict penalties and presents worse load balance.

5 Discussion

The face detection application presents a data-dependent behavior, which leads to load imbalance. Different parallelization strategies could provide better load balance and could be evaluated even before having a parallel implementation using a high-level model built from the sequential application traces[4]. Nonetheless, fine performance tuning can only be done with a precise simulator or prototype implementation.

The OpenCL programming model has limited data placement options, which hinders optimal data placement. Custom vendor extensions are needed to enable fine tuning of the data placement. Control over the scheduling is also limited, leading to processing gaps due to inter-work-group load imbalance. A more dynamic kernel enqueueing mechanism as proposed in OpenCL 2.0[8] could provide better load balancing.

As the STHORM SDK provides no cycle-approximate simulator for the host, it is not possible to estimate application-level performance. On the device side, although using a cycle-approximate simulator of the STHORM processing elements, a large mismatch has been observed between the simulator and the prototype board results. The causes of such mismatches are mostly due to the memory subsystem modeling. The inclusion of memory latency and bandwidth parameters in the SDK, the modeling of local memory conflicts on the simulator and the usage of a cycle-approximate host simulator should reduce the mismatch. This will enable application-level profiling and optimisation, so that the we can find the best algorithm-architecture co-design trade-offs as early as possible in the flow.

6 Conclusion

Computer vision applications have gained widespread adoption in recent years and are pushing the current architectures. Although they present a high potential for parallelization, vision algorithms often present highly variable data-dependent execution times which lead to parallel load imbalance. The latter negatively affects the parallel efficiency, and is an important source of inefficiency in the system. Current data-parallel programming models, such as OpenCL, are not able to efficiently schedule work to fill-in the gaps generated by this imbalance, leaving it up to the programmer to do some gymnastics in order to refactor the algorithm and minimize the imbalance. Programming model support for more dynamic workload allocation is needed.

To extract the full potential of a parallel platform it is necessary to strike a good balance between computation and communication. The current generation of STHORM many-core simulation tools do not accurately model internal memory conflicts, external communication latency and throughput, as well as the interaction with the host processor. It is therefore not possible to use the simulators for application-level performance estimation and tuning. Physical prototypes are the only option, which, due to the higher setup effort and to constraints of the prototype itself, limit design space exploration. Moreover, when designing a new embedded system, a fully working prototype often comes too late in the development flow, when major architectural decisions have already been taken. Thus, a simulation environment that allows for precise application-level performance assessment and optimization early in the development flow is invaluable, and the only way to achieve this is if the simulation platforms are able to provide precise timing figures for the host, accelerator and memory subsystems.

References

[1] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 1–4, 1967.

[2] K. Asanovic, J. Wawrzynek, D. Wessel, K. Yelick, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, and K. Sen. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56, Oct. 2009.

[3] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 983–987. EDA Consortium, 2012.

[4] K. Czechowski, C. Battaglino, C. McClanahan, A. Chandramowlishwaran, and R. Vuduc. Balance principles for algorithm-architecture co-design. *USENIX Wkshp. Hot Topics in Parallelism (HotPar)*, pages 1–5, 2011.

[5] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel. A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor. *2013 International Conference on Computational Science. Procedia Computer Science.*, 18(0):1654–1663, 2013.

[6] ITRS. International Technology Roadmap for Semiconductors. `http://www.itrs.net`, 2011.

[7] Khronos OpenCL Working Group. The OpenCL Specification. *Version 1.1, Document Revision: 44*, 2010.

[8] Khronos OpenCL Working Group. The OpenCL Specification. *Version 2.0, Document Revision: 19*, 2013.

[9] B. Kisacanin. Integral Image Optimizations for Embedded Vision Applications. *Image Analysis and Interpretation, 2008. SSIAI 2008. IEEE Southwest Symposium on*, (1):181–184, 2008.

[10] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit. Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1137–1142. ACM, 2012.

[11] G. E. Moore. Cramming more components onto integrated circuits. 38(8), 1965.

[12] J. Mottin, M. Cartron, and G. Urlini. The STHORM Platform. In *Smart Multicore Embedded Systems*, pages 35–43. Springer, 2014.

[13] P. Paulin. Programming challenges & solutions for multi-processor SoCs: an industrial perspective. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 262–267. IEEE, 2011.

[14] A. Rahimi, I. Loi, M. R. Kakoee, and L. Benini. A Fully-Synthesizable Single-Cycle Interconnection Network for Shared-L1 Processor Clusters. In *Design, Automation & Test in Europe Conference & Exhibition*, 2011.

[15] B. Stefanizzi. The Programmer's Guide to a Universe of Possibility. In *AMD Fusion Developper Summit*, 2012.

[16] P. Viola and M. Jones. Robust real-time face detection. *International journal of computer vision*, 57(2):137–154, 2004.

[17] W. Wolf, A. A. Jerraya, and G. Martin. Multiprocessor System-on-Chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, 2008.

[18] N. Zhang. Working towards efficient parallel computing of integral images on multi-core processors. *2010 2nd International Conference on Computer Engineering and Technology*, 2:30–34, Apr. 2010.