Alma Mater Studiorum – Università di Bologna

DOTTORATO DI RICERCA IN

Ingegneria Elettronica, delle Telecomunicazioni e Tecnologie dell'Informazione

Ciclo XVIII

Settore Concorsuale di afferenza:
Area 09 (Ingegneria Industriale e dell'Informazione) – 09/E3 Elettronica

Settore Scientifico disciplinare:
Area 09 (Ingegneria Industriale e dell'Informazione) – ING-INF/01 Elettronica

HETEROGENEOUS ARCHITECTURES FOR PARALLEL ACCELERATION

Presentata da: Dott. Francesco Conti

Coordinatore Dottorato

Prof. Alessandro Vanelli Coralli

Relatore

Prof. Luca Benini

Esame finale anno 2016

Heterogeneous Architectures for Parallel Acceleration

Francesco Conti

Department of Electrical, Electronic and Information Engineering
University of Bologna

A thesis submitted for the degree of
Doctor of Philosophy in
Electronics, Telecommunications and Information Technologies Engineering

2015

There are more things in heaven and earth, Horatio,
than are dreamt of in your philosophy.

W. Shakespeare, *Hamlet*

Abstract

To enable a new generation of digital computing applications, the greatest challenge is to provide a better level of energy efficiency *(intended as the performance that a system can provide within a certain power budget) without giving up a systems's* flexibility*. This constraint applies to digital system across all scales, starting from ultra-low power implanted devices up to datacenters for high-performance computing and for the "cloud". In this thesis, we show that* architectural heterogeneity *is the key to provide this efficiency and to respond to many of the challenges of tomorrow's computer architecture - and at the same time we show methodologies to introduce it with little or no loss in terms of flexibility. In particular, we show that heterogeneity can be employed to tackle the "walls" that impede further development of new computing applications: the utilization wall, i.e. the impossibility to keep all transistors on in deeply integrated chips, and the "data deluge", i.e. the amount of data to be processed that is scaling up much faster than the computing performance and efficiency. We introduce a methodology to improve heterogeneous design exploration of tightly coupled clusters; moreover we propose a fractal heterogeneity architecture that is a parallel accelerator for low-power sensor nodes, and is itself internally heterogeneous thanks to an heterogeneous coprocessor for brain-inspired computing. This platform, which is silicon-proven, can lead to more than* $100\times$ *improvement in terms of energy efficiency with respect to typical computing nodes used within the same domain, enabling the application of complex algorithms, vastly more performance-hungry than the current state-of-the-art in the ULP computing domain.*

Contents

List of Figures

Acknowledgements

Apparently writing a PhD thesis is hard, but writing acknowledgements is MUCH harder!

I want to thank all of my friends and current and former colleagues in Prof. Benini's group in Bologna and in Zurich. In particular, I want to thank (in no particular order) Alessandro Capotondi, Francesco Paci, Giuseppe Tagliavini, Andrea Bartolini, Francesco Beneventi, Daniele Cesarini, Marco Balboni, Daniele Bortolotti, Christian Pinto, Igor Loi, Erfan Azarkhish, Thomas Bridi, Simone Benatti, Filippo Casamassima, Francesco Fraternali, Pietro Mercati, Elisabetta Farella, Bojan Milosevic and Manuele Rusci who are or have been my colleagues at the Energy-Efficient Embedded Systems laboratory in Bologna - working with you was really lots of fun (and of creative craziness)! I also want to thank Angela Cavazzini for her invaluable help in the most difficult of research tasks - bureaucracy! Finally, I'd like to thank a not so talkative, but certainly invaluable "member" of our research group - our beloved NeuroLab coffee machine! My PhD certainly wouldn't have been the same without you.

From the group in the Integrated Systems Laboratory in ETH Zurich, with whom I have passed five months during my PhD and am now a stable collaborator, I want to thank everyone, but in particular Daniele Palossi, Antonio Pullini, Germain Haugou, Michael Gautschi, Pirmin Vogel, Michael Schaffner, Lukas Cavigelli, and Frank Gurkaynak for the many interesting discussions on ultra-low power parallel platforms, exciting chips and other stuff - especially in front of a good glass of beer.

A very special "thank you" goes in particular to the two post-doctoral researchers who most closely followed my PhD work, Andrea Marongiu and Davide Rossi. Much of what I have learned during my PhD work is due to you - it's been a pleasure to work with you guys as a student, and it will be a pleasure to continue to do so in the future. I want to specifically thank Andrea for forging my scientific methodology (and writing abilities) especially at the very beginning of my PhD, when I was still a bit

lost, and for his incredible patience and precision in following us students; and Davide for his precious technical and scientific lessons, and the many coffee-break discussions on how to save any small bit of energy in the PULP project of which he is a leading member.

I also want to thank Prof. David Bol from KU Leuven and Prof. Andreas Burg from EPFL for their illuminating reviews to the first draft of this thesis, that hopefully helped to make it considerably better.

Last but not least, of course, I would like to thank Prof. Luca Benini for his continuous support, feedback, sometimes criticism for my work; for his contagious creative and technical excitement; and also for the many conversations that helped me find new angles into what I was doing to make it better and more consistent. Thanks for our exciting work!

Francesco

Ringraziamenti

Credevate che i ringraziamenti fossero finiti? Ho scritto in inglese unicamente i miei ringraziamenti per i colleghi, ma altrettanto critico per questo dottorato è stato l'aiuto e l'amore di amici e parenti.

Per questo motivo voglio ringraziare i miei amici / fratelli Lorenzo, Davide, Riccardo, Luca, Marco senza cui la vita sarebbe noiosa e che sono per me fratelli di cuore se non di sangue da tantissimi anni. Ringrazio tutti i miei parenti vicini e lontani, e in particolare i miei zii Franca e Franco, i miei cugini Alessio e Chiara con i piccoli Diana e Sebastian, e mia nonna Lia che mi hanno dato tanto e tanto continuano a donarmi ogni giorno. Ringrazio i miei parenti (non ancora) acquisiti, e in particolare Orietta e Mario perchè mi hanno sempre fatto sentire a casa.

Ringrazio i miei genitori Claudia e Marco, supporto e sfogo di ogni mia frustrazione ma anche ricettori di tutte le grandi soddisfazioni. Non esistono parole per esprimere il ringraziamento che vi devo, per il vostro amore e il vostro aiuto.

Infine, ringrazio l'amore della mia vita, Francesca, per essermi stata accanto ogni giorno di questi tre anni faticosi per entrambi ma anche belli e pieni di soddisfazioni. Grazie!

Francesco

Chapter 1

Introduction

1.1 Homogeneous and heterogeneous many-cores

In the last decades computing has become truly pervasive in a way never before imagined even in the most daring predictions. What is particularly mind-shattering about this phenomenon is that the utter dominance of digital computing has arrived at the same time at many scales: from the enormous data centers that constitute the backbone of the "cloud", to the smallest body-implanted devices, passing through a variety of applications in industrial plants, vehicles, everyday objects such as watches and glasses, not to mention "explicit" computers such as PCs, tablets and smartphones.

Yet, even if the scale of these computers encompasses orders of magnitude in terms of size, performance and energy consumption, all of them are based on relatively similar principles, starting from the physics of Silicon devices up along the technology stack. A corollary of the fundamental similarity of these systems in everything except for scale is that problems, solutions and general trends are shared between systems that seem totally dissimilar: for example, the irresistible trends to integrate more functionality on single chips [9] and to provide higher flexibility substituting dedicated hardware with software [10].

In the past, these trends have been fundamentally driven by Moore's Law [11] and Dennard scaling [12]: the number of transistors increased exponentially with each technology generation, while the overall power density was kept constant by downscaling the operating voltage and delay was reduced due to the smaller capacitances to be driven. While Moore's law has continued working up to now, in the past 15

years it has become increasingly clear that it is not possible to further scale down the operating voltage. This is due to the fact that the leakage current grows exponentially as the threshold voltage is reduced, effectively constituting a so-called *power wall* that impedes further voltage downscaling [13]. At the end of the "single-core era", complex hardware was necessary to support the increasingly sophisticate techniques necessary to exploit instruction level parallelism (ILP). More and more transistors had to be invested into this kind of hardware (e.g. out-of-order execution, speculation, etc.) with diminishing performance returns, resulting in an overall decrease of energy efficiency that was no more counteracted by the availability of more transistors within the same power budget, as was the case in the Dennard regime.

The solution to this issue was an industry-wide shift towards *parallel computing*. Traditional single-core computers are not designed to be *scalable* from an architectural point of view; there is no conceptually easy way to take a core and increase its performance by 100× while spending no more than 100× the original power apart from scaling its clock frequency. Conversely, parallel architectures are architecturally scalable by construction, the main tweakable parameter being the number of cores. This fact is resumed into Pollack's rule of thumb [14], that states that the performance of a single core processor scales with the square root of its complexity, which is a strong push towards architectures with many small and efficient cores. In fact, each core or "processing element" in a parallel computer can be much simpler and smaller than the most complex ILP-oriented cores, performing more operations per Joule even if it is much slower. Several recently proposed platforms take this concept to its extreme, working at very low clock speed to exploit the maximum efficiency of CMOS logic when it is operated near the voltage threshold [15], and rely on thread-level parallelism (TLP) or data-level parallelism (DLP) to reach an adequate performance target [16].

The scalability of multi-core architectures has of course its own limits: first, some applications have limited amounts of available TLP and DLP [17]; second, the communication infrastructure typically poses a hard limit to either the number of cores or to how flexibly they are able to share data - and thus to how much of the available parallelism can be exploited. In practice, these observations limit the number of cores that can cooperate in a tightly coupled fashion through a shared local cache or scratchpad, requiring the usage of a more scalable medium such as a network-on-chip [18] to scale the number of processors above 8-16 [6]. We loosely distinguish between

multi-core architectures, that are typically organized around a single memory shared at level 1 (L1) or level 2 (L2) and feature a relatively small number of cores, and *many-cores*, that are composed of *clusters* of cores interconnected with a network-on-chip [5]. Many-cores can have from several tens up to thousands of cores, thanks to the high degree of scalability of the network-on-chip. Each cluster can be a single core or a group of cores sharing memory with a more flexible but less scalable scheme.

There are several limits for which these *homogeneous many-cores*, based on simple replication of identical processing elements, could not grant the level of energy efficiency that is needed for several applications. The end of Dennard's scaling poses an intrinsic upper limit to the number of cores that can be operated simultaneously: as the power density increases with each technology node, keeping all transistors powered on (and at a sufficiently low temperature) is rapidly becoming an intractable issue for relatively high-performance platforms - a problem known as the *utilization wall* [19][20]. To cope with the utilization wall, it is necessary to either power on only a fraction of the chip at a time (*dark silicon*) or operate it at a much slower operating point (*dim silicon*). At the same time, the amount of computing performance that is required by very low power sensors and systems is growing very rapidly due to the quantity of raw data that is necessary to drive modern "smart" applications based on data mining and machine learning; much more rapidly, in fact, than the performance that is available in typical systems used in sensor nodes, that are based on low-power microcontrollers [21][22] - or than the capability to send this information on the network for storage on a remote server [23]. We could call this limitation a *data wall* that is difficult to solve by simple replication of the cores on existing microcontrollers, that are mainly optimized for intrinsically sequential control tasks. Finally, the replication of processors does not address at all the inherent energy efficiency limitations of Von Neumann's computational model, that are related to the necessity to repeatedly fetch instructions in a fetch-decode-execute loop.

A possible common solution to many of these limitations is that of abandoning the premise that the replicated cores are identical, optimizing each core or set of cores for a particular task. This flavor of parallel computing is often called *heterogeneous computing* and is already a popular solution in many fields, especially in the form of Graphic Processing Unit (GPU) computing. Heterogeneous processing elements may be cores with the same Instruction Set Architecture (ISA) but different microarchitecture, cores with different ISAs, or even accelerators with no ISA at all or a simple

programming interface based on microcode. The heterogeneity of computing tasks is often well reflected on a heterogeneous computing fabric. Heterogeneous computing can be effective in addressing the utilization wall [20][1]: while the performance in a homogeneous many-core is, in a first-order analysis, directly proportional to the number of active cores (and thus to the dissipated power), in a heterogeneous many-core the relationship depends on the specialization of each core to a particular application, giving the system designer an additional knob to keep thermal constraints in check. Moreover, specialized cores are typically designed to yield much more performance for each consumed watt than general-purpose ones (for a subset of application domains), and they can exploit much finer grain data parallelism. Heterogeneous many-cores that employ both standard cores and accelerators are therefore highly suited to cope with the data deluge coming from sensors; more efficient computation directly on sensors can greatly reduce the dimensionality of the data (by increasing its level of abstraction), helping to solve the "*data wall*".

In general, the effectiveness of heterogeneous computing is that it can help to unlock yet another level of energy efficiency that can be spent to get more performance with the same power, or to get the same performance with less power, or a combination of both. Heterogeneous platforms with software cores and hardware accelerators achieve this target by unlocking the energy efficiency potential that is kept unaccessible by the burden of the Von Neumann fetch-decode-execute loop. Conversely, heterogeneous platforms composed of relatively complex ILP-optimized multi-cores and programmable accelerators do the same at another level, unlocking the potential of extracting data parallelism while keeping the flexibility of the more standard cores. In fact, these two levels of heterogeneity can be coupled in what could be called *fractal heterogeneity*: like a Russian matrioska, a standard multi-core can be accelerated with a many-core platform that is itself accelerated internally with specialized units to perform some tasks.

1.2 Taxonomy of heterogeneous parallel architectures

As previously explained, heterogeneous architectures based on HW accelerators work in an asymmetric fashion: they feature a processor (or a group of processors) acting as *host* running non-accelerable code, and *accelerators* used to achieve a greater level

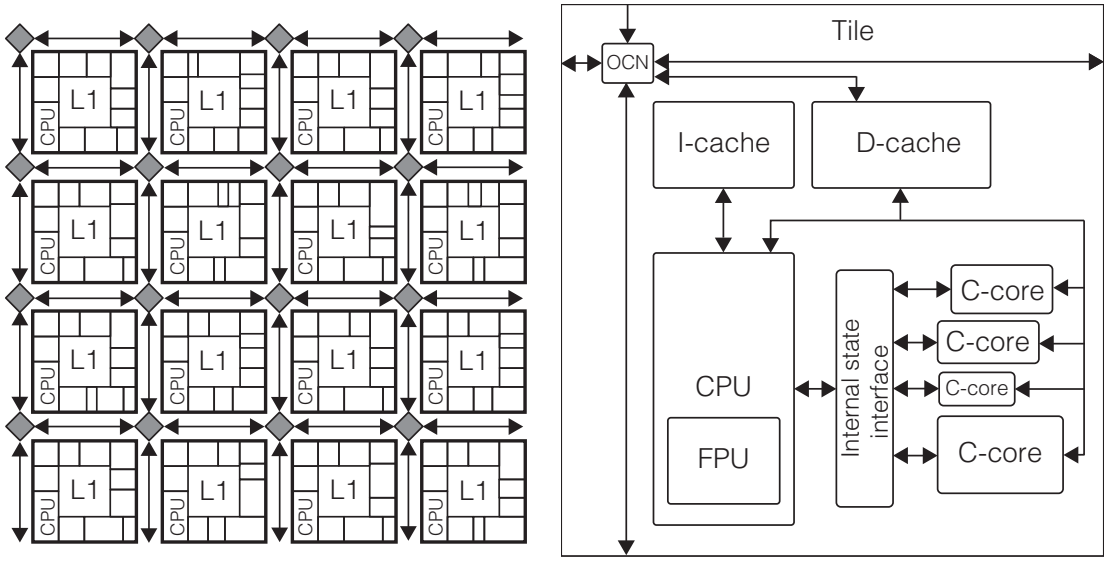
in some metric (e.g. energy-efficiency). This kind of heterogeneity can be present at many scales in what we call *fractal heterogeneity*: for example, He-P2012 [24][25] – the platform we present in Chapter 2 – is itself meant to be used as an accelerator to an ARM host processor running mostly sequential code. We can group acceleration paradigms regardless of the level at which they work in this fractal hierarchy, by considering two main axes: how tight is the integration between host and accelerator, and how is data exchanged between them.

A first approach is to insert accelerators directly inside the processor pipeline, and share data through the register file. This is the case of ASIPs with specialized functional units, such as the ones provided by Tensilica (now Cadence) [26] and Synopsys Processor Designer [27]. Other examples include specialized architectures such as Movidius Myriad [28], Silicon Hive [29] and TI AccelerationPAC [30] that rely on Very-Long Instruction Word (VLIW) cores augmented with special-purpose units. Clemons et al. [31] propose EVA (Efficient Vision Architecture), an asymmetric multicore featuring 1 out-of-order coordinating core and many supporting low-power cores, all augmented with custom Single Instruction, Multiple Data-stream (SIMD) accelerators for common operations such as dot products. This acceleration approach can provide significant performance and energy-efficiency gains in very fine-grain workloads, but is inefficient for coarser-grain functionality, its main limitation being the fact that it is usually limited to communicate through the register file interface. Moreover, as it is implemented inside a processor's pipeline, it is still subject to the Von Neumann *fetch-decode-execute* bottleneck, which can seriously limit its potential efficiency. Efficacy of this acceleration approach also relies on very deep knowledge of the algorithm and the architecture on the programmer's part (therefore limiting the number of proficient users), or on very efficient compiler technology that is often not available.

The opposite extreme of the spectrum is to use very coarse-grained accelerators and rely mostly on message-passing channels for host-accelerator communication, adopting a dataflow model of computation. This approach has been leveraged for software-defined radio applications (see for example Ramacher et al. [32], CEA MAGALI [33][34] and StepNP [35]), in high-performance computing (for example by Maxeler [36]) and in the embedded vision domain (for example Vortex [37][38] for biologically-inspired vision acceleration). Loose coupling of processors and accelerators through message-passing channels is extremely scalable and power-efficient, as accelerators

completely bypass the Von Neumann bottleneck, working only when they are fed with input. Many high-level synthesis tools [39] partially relieve the designer of some of the complexities of hardware design; however, this approach requires extensive rewriting of the code following a dataflow model that is completely different with the dominant SW imperative paradigm [40], thus requiring extensive specialization which greatly reduces their practicality. Moreover, not all algorithms are amenable to be implemented in a dataflow fashion without loss of generality or efficiency, and fine-grain acceleration is typically not possible due to the overhead of message-passing communication. Much recent research on loosely-coupled accelerators has concentrated on targeting FPGA devices, often trying to hide the under-the-hood model from the developer by exposing a high-level programming model such as OpenCL [41][42].

In between these two extremes, we find many architectures that employ architectures clearly separated from the host processor pipeline, sharing data through one of the levels of the memory hierarchy. At high level, this is what is done by APUs [43] and many-core accelerators such as P2012 [5], which share DRAM with a x86 or ARM host processor optimized for sequential execution. At a smaller scale, special-function units in Nvidia GPUs [44] and coprocessors in Plurality HyperCore [45] are also examples of this trend. The *GreenDroid* architecture [1], shown in Figure 1.1, is composed



(a) Overall GreenDroid tiled architecture.

(b) A single GreenDroid tile featuring several c-cores.

Figure 1.1: *GreenDroid* architecture (from Goulding-Hotta et al. [1]).

of a set of tiles connected through a point-to-point mesh network-on-chip. Each tile contains a single energy efficient general purpose processor able to boot Android, 32 kB of L1 data cache and a different array of 8 to 15 *conservation cores* or c-cores that are tightly coupled to the processor through the L1 data cache. The c-cores are generated via high-level synthesis of hot regions of code in Android libraries and in the Dalvik virtual machine. In the Android-based workload that was considered by Goulding-Hotta et al. [1] 95% was spent executing c-core accelerated code, with an 18× improvement on energy per instruction on average on the accelerated fraction.

Another similar architecture is EXOCHI, which was introduced by Wang et al. [46], features hardware accelerators modeled as coarse-grain MIMD functional units, collaborating with IA32 CPU cores by sharing of the same virtual memory space. Cong et al. [47] also tackle the utilization wall by developing a heterogeneous multi-core architecture with shared-memory accelerators; their HW IPs communicate by means of shared L2 caches, accessible through NoC nodes. The model we propose in Chapter 2, that is also based on previous work in our group (see e.g. Burgio et al. [48], Dehyadegari et al. [49][50], Conti et al. [51]) is similar in that we consider hardware processing elements (HWPEs) sharing memory at L1 with a tightly coupled cluster of cores.

Platform	**Offload latency** *cycles*	**Memory latency** *cycles*	**Memory bandwidth** *B/cycle*	**Memory size** *B*
Dataflow (Maxeler)	1000	10-100	64 (per link)	$> 10^{10}$
GPGPU (Nvidia)	1000	10-100	64	10^9
FPGA SoC (Zynq)	1000	10-100	8	10^9
He-P2012 (many-core)	1000	10-100	8	10^9
He-P2012 (HWPEs)	10-50	2-3	4 (per port)	10^5
ASIP	1	1	4 (per port)	10^3

Table 1.1: Orders of magnitude of offload latency and memory parameters for several heterogeneous paradigms.

Table 1.1 specifies orders of magnitude for offload latency and memory access bandwidth, latency and size for several heterogeneous platforms. Orders of magnitude of memory performance metrics for Nvidia GPGPUs and the Zynq FPGA SoC are derived from publicly available information from Nvidia and Xilinx; metrics for the Maxeler dataflow engine are derived from [36].

1.3 Homogeneous and heterogeneous parallel platforms for low power

Heterogeneous platforms and systems are particularly useful in all use cases where there is need for high performance under a very tight power envelope. In this scenario, the energy efficiency boost provided by a heterogeneous platform can be a key enabler to implement complex functionality with a limited system power. We believe that it is at the ultra-low power scale, where energy efficiency matters most, that heterogeneity can be most effective.

Currently, the state-of-the-art in low power and ultra-low power computation is represented by single-core microcontrollers that can easily target power budgets of 50 mW and below. An example of such a platform is the STMicroelectronics STM32L47, based on a ARM Cortex-M4 [52]. State-of-the-art ULP microcontrollers that can work with less than 10 mW include the SiliconLabs *EFM32* [53], Texas Instruments *MSP430* [54] families of microcontroller units (MCUs), and *Ambiq Apollo* [55]. With respect to traditional microcontrollers operating at relatively high voltage (e.g. 1.2 V), efficiency can be improved by operating near the threshold voltage, as is demonstrated for example in Ickes et al. [56], *SleepWalker* [57] and *Bellevue* [58], which also exploits SIMD parallelism to further improve performance. Similarly to the case of general-purpose processors one decade ago, power considerations do not allow for a significant improvement of the performance of these platforms. Both situations are related to the end of Dennard scaling: while then the main issue was the *thermal wall*, in the case of these microcontrollers the scaling limitation is mostly due to the objective power constraints of the final applications.

However, many burgeoning applications such as vision-enabled sensor nodes would need far more powerful platforms within a power budget comparable to that of the lowest power MCUs currently on the market. For this reason, heterogeneous architectures already exist at this scale, although the coupling mechanisms are usually less sophisticated than those used in higher power & performance state-of-the-art platforms. For example, in many cases fixed-function HW blocks or ASICs are used to augment a low-power microcontroller, with little flexibility in the application [59][60]. Computing capabilities such as simple filtering can also be integrated directly within the vision sensor [61], forming an unusual kind of heterogeneous system.

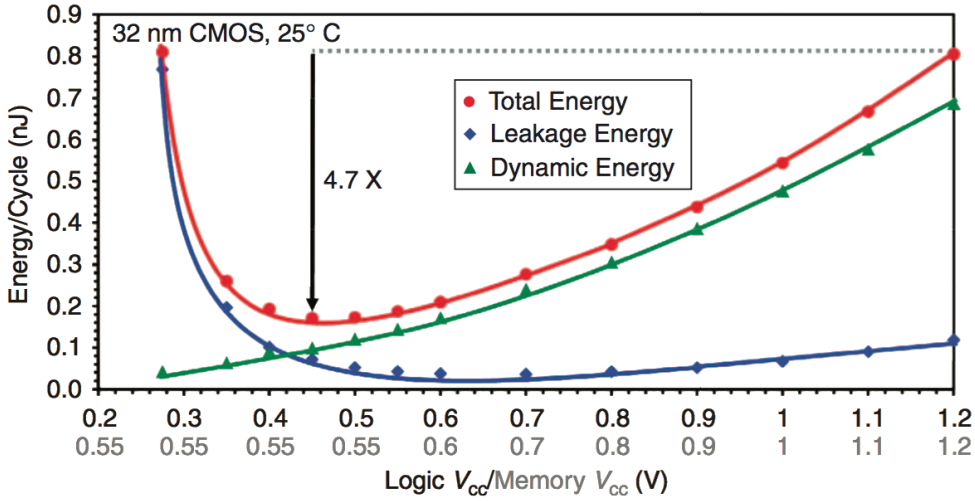


Figure 1.2: Peak energy efficiency at near-threshold voltage in a Pentium-class IA-32 microprocessor (from Jain et al. [2]).

To provide the same boost in efficiency while also keeping a higher level of flexibility, a possible solution is to be inspired again by Pollack's rule [14] and use many small and slow cores working at ultra low voltage as an accelerator, as advocated for example by Pinckney et al. [62]. To illustrate this principle, Figure 1.2 shows the results reported by Jain et al. [2] regarding a Pentium-class in-order core in 32nm CMOS logic that was demonstrated to be operational across a voltage range between 0.28 V and 1.2 V, correspondent to a variation in frequency between 3 MHz and 915 MHz and in power between 2 mW and 715 mW. At the minimum energy point (@0.45 V) the core consumes 50× less than at the nominal 1.2 V operating point while being only 15× slower, underlying the advantage of this approach.

The same approach is followed by *Centip3de* [3]. Centip3de consists of a large scale 3D-integrated fabric of clusters of Cortex M3 cores. As shown in Figure 1.3, there are 64 cores organized in 16 clusters of 4 cores each. Each cluster shares instruction and data caches residing on a separate die bonded face-to-face; these memories operate at a higher voltage and at 4× the frequency of the cores, with the double advantage of keeping the illusion of single-cycle access from the core's point of view despite the sharing and bypassing the necessity of 8T-SRAM cells that can be driven at low voltage. In this way the frequency of the cores can be scaled much further down than what is typically possible with standard 6T-SRAM [63].

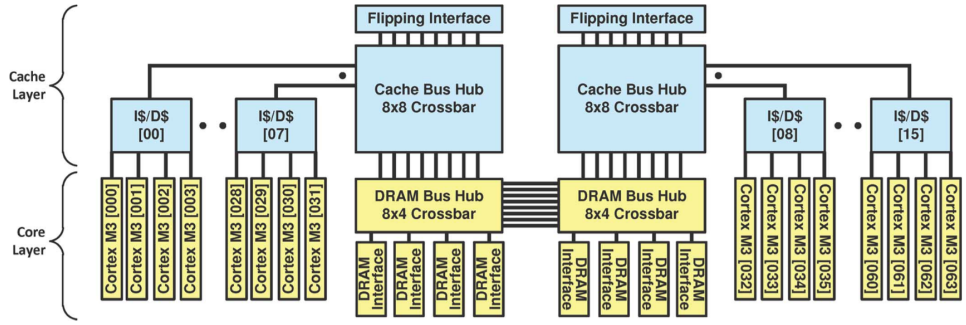


Figure 1.3: Centip3De near-threshold parallel architecture (from Fick et al. [3]).

The *PULP* architecture [64][65][66][67][68], that will be presented in a comprehensive way in Chapter 3, combines this approach with the heterogeneous accelerator model to enable a new class of applications with relatively high performance needs on systems with a very low power budget. pulp introduces the advantages of heterogeneity to the sub-10 mW design space, by providing a general-purpose computing device that is designed to deliver high performance/watt for parallel workloads while leaving sequential control-related tasks to a normal microcontroller unit. The current embodiment of this general model is a 28nm FD-SOI programmable ULP parallel platform [67] that can be coupled to a low-power off-the-shelf MCU such as a STM32 via a low-power SPI connection that is used both for controlling the accelerator and for data exchange. Moreover, we argue this kind of platform introduces significant potential in terms of fractal heterogeneity - by combining software flexibility, efficiency given by the technology and enabled by the architecture, and heterogeneous acceleration, it would be possible to extract much of the potential efficiency of many applications. Chapter 4 will describe such an architecture, targeted at the acceleration of convolutional neural networks.

1.4 Balancing flexibility and efficiency: brain-inspired computing and heterogeneity

As explained in the previous sections, it is often difficult to balance flexibility with energy efficiency in the context of heterogeneous systems, especially when efficiency is extracted by means of fixed-function hardware blocks that are intrinsically designed

to perform a single function. However, a class of computational models that can be used to enable a wide-ranging set of functionalities while still requiring only a small set of primitive functions does exist: that of *brain-inspired computing*. Brain-inspired models are very different one from the other, but at the fundamental level they share the common nature of being suitable for a very large set of seemingly unrelated tasks due to their capability to *learn* via an offline training procedure, via online learning or a combination of both. In fact, some neural models have been proven to be capable of universal approximated computation [69][70] and are thus capable, at least in theory, of being trained to perform *any* task. These characteristic make them attractive for integration in a heterogeneous parallel platform as an alternative to traditional Von Neumann architectures[1]: the trade-off between efficiency and flexibility would be substituted by another one between efficiency and accuracy. The fact that this second trade-off between accuracy and energy efficiency is a fundamental one is well known and is at the heart of the so-called "approximate computing" field [72][73], in the context of which neural-network based approximators have been proposed a number of times [74][75]. Flipping the approximate computing paradigm the other way around, brain-inspired trained models are particularly effective to "approximate" tasks of too high complexity to be represented by an actual closed form function or to be efficiently solvable by simple algorithms. This is the case of most computer vision algorithms, such as for example classification of images [76] and scene parsing [77].

Internally, the class of brain-inspired models has extreme variance in terms of the accuracy of representation of the inner mechanisms of the brain. They range from very high level models of the primary visual cortex such as HMAX [78] and Convolutional Neural Networks [79]) to accurate large-scale spiking models to study the internal mechanisms of the brain such as Izhikevich's model [80], passing through "middle ground" such as spiking-based liquid state machines [81][82].

A particularly interesting class of algorithms from this point of view is that of Convolutional Neural Networks (*CNN*s or *ConvNet*s) that are state-of-the-art in many accuracy benchmarks, particularly in the computer vision field [76][83][84][85][86]. As CNNs are based on accumulation of convolutions, they are well suited to being im-

[1] An interesting trivia note is that the original Von Neumann architecture, as proposed in the "First Draft of a Report on the EDVAC" [71], uses a good amount of brain-inspired concepts in justifying its architectural proposal, drawing explicit comparisons between what we would now call "logic gates" and neurons.

plemented in hardware using well-known techniques, which makes them attractive for integration in heterogeneous platforms and systems to enable implementation of a task in a different point of the accuracy-efficiency point mentioned before. Research on specific hardware for CNNs is rich and includes FPGA platforms such as NeuFlow [87][88] and nn-X [89], ASICs such as ShiDianNao [90][91] and Origami [92], and SIMD processor extensions such as the Convolution Engine presented in Qadeer et al. [93]. In Chapter 4 we present the HWCE [94], an energy efficient tightly-coupled coprocessor for acceleration of convolutional workloads in a ultra-low power heterogeneous platform.

Several architectures making use of spiking neural networks (*SNN*s) for recognition and vision have been recently proposed [95][96]. IBM TrueNorth [97][95] targets vision applications using event-based spiking neural networks, and provides first class energy results, being able to perform real-time multi-object recognition with a 72mW power budget. With respect to the amount of outstanding results available for more abstract models such as CNNs, state-of-the-art accuracy out of SNNs has not been fully demonstrated yet; however, recent results such as those of Diehl et al. [98] have shown that it is possible to reuse the training infrastructure for non-spiking deep neural networks also for SNNs, enabling their usage in a much bigger class of cases.

1.5 Contributions and claims

The great challenge of heterogeneous architectures is to find the most effective point in the trade-off between the full flexibility of executing all computation in software on a general-purpose machine and the full efficiency of executing it in custom hardware, specialized for one and only one task. In the remainder of this thesis, we will explore both extremes in the trade-off between these two defining characteristics of heterogeneous architectures, concentrating on the space of energy-efficient multicores. We will first describe a platform with embedded fixed-function accelerators, in which design-time flexibility comes out of the usage of high-level synthesis and a certain degree of run-time flexibility can also be reached, by using simple and standard software primitives for the accelerator control. However, this run-time flexibility is of course still limited by the initial choice of accelerators. Then, we will switch to the opposite side of the trade-off describing a fully programmable accelerator for

microcontrollers employing a set of Reduced Instruction Set Computer (RISC) cores to accelerate data-parallel algorithms. This platform can reach a very high level of energy efficiency thanks to its power-optimized architecture and the extended usage of several low-power features of the STMicroelectronics FD-SOI 28nm technology; this notwithstanding, its efficiency is still fundamentally limited by the inefficiency of the Von Neumann *fetch-decode-execute* bottleneck. Finally, we will propose a compromise solution that can help achieve the same high level of efficiency of traditional accelerators while also delivering a very significant level of application-level flexibility, by leveraging the features of brain-inspired computational models such as neural networks.

Overall, our contributions add up to show that heterogeneous computing is an effective solution to many of the challenges that computer architecture is facing after the end of Dennard's scaling and at the twilight of Moore's law, as will be detailed in the final Chapter 5 of this thesis. In absence of a technological breakthrough, in a few years' time the effort to keep up with the growing computational needs of humanity will be almost entirely up to computer architects, who will have to find innovative solutions that bypass the limitations we have described in Section 1.1. Essentially, this boils down to increasing *computational energy efficiency* (in terms of operations per unit energy) and *computational density* (in terms of operations per unit area) by pure architecture changes. We claim that despite the power and attractiveness of some computational models such as approximate computing and brain-inspired computing, general-purpose computing as was refined during the ~ 70 years since the invention of the first digital electronic computers [71] is here to stay. The reason is simply that general-purpose computers are able to do everything, even if they might be suboptimal at it - and the level of precision on a given task is defined at the algorithmic and not at the architectural level, leaving full freedom to developers to explore any kind of solution. Moreover, and perhaps most importantly, success of a paradigm in computer architecture (as in any engineering discipline) depends not only on general effectiveness, but on *cost-effectiveness*. With the cost of integrated circuit design and fabrication becoming higher and higher with each process node [99], economy of scale considerations encourage designs that can be reused for many tasks against ones that are specific to a single one - more so in advanced technology nodes.

The necessary increase of efficiency and density will therefore arrive not by substituting general-purpose computers, but rather by introducing together with them

heterogeneous elements, ranging from different microarchitectures using the same ISA up to elements heterogeneous to the Von Neumann model itself. These heterogeneous cores will progressively be tightly integrated with "traditional" cores, as they are more effective and efficient at particular tasks (be they parallel computing in a GPU or a many-core platform, perception in a neural computing unit, or cryptography in a specialized engine) without fundamentally impairing the system flexibility.

Chapter 2

He-P2012: exploring heterogeneity in tightly-coupled clusters

As explained in Chapter 1, adding heterogeneity to a computing platform adds both advantages and challenges. In this Chapter, we provide two main contributions: first, we define an architectural paradigm for the integration of heterogeneous fixed-function cores (i.e. accelerators) inside a homogeneous tightly-coupled cluster of processors; second, we provide a methodology for the exploration of the heterogeneous design space starting from software running on the original homogeneous cluster.

2.1 Overview

One of the main challenges for the design of future embedded Systems-on-Chip is the *utilization wall* [20]: due to the end of Dennardian scaling, the fraction of the chip that can be kept on at a given power budget decreases with every new technology node. Most of the area in future SoCs will therefore be "dark silicon"[19], i.e. composed of transistors that are only seldom powered on; designers must face the challenge to provide more computing performance by exploiting this dark silicon in an effective and sensible way. The utilization wall limits the amount of parallelism that can be exploited by adding more and more general-purpose processors, because it is impossible to keep them simultaneously on. At the same time, though, it also frees chip area for other purposes.

Proposed techniques to make effective use of dark silicon include *dim* and *specialized* silicon [20], i.e. respectively silicon that is used seldom/at a lower frequency and

silicon that performs specialized functions. By inserting both general and special-purpose processors in a single computing fabric we can effectively counteract the loss of potential performance due to the utilization wall, since special-purpose processors are typically not in use simultaneously nor are they always on [100]. However, heterogeneous architectures increase the size and complexity of the design space along several axes: granularity of the heterogeneous processors, coupling with software cores, communication interfaces, etc. Simulation flows and tools aimed at simplifying design space exploration are thus highly beneficial.

Evolution towards heterogeneity in multi- and many-core SoCs stimulates an evolution also in traditional HW/SW codesign techniques [101][102]. In this chapter, we describe our technique to augment the STMicroelectronics P2012 cluster with tightly-coupled shared-L1 memory *hardware processing elements* (HWPEs). HWPEs can be used to increase performance and/or energy efficiency of the P2012 homogeneous clusters. Shared-L1 HWPEs work in-place on data shared by the SW cores instead than on a copy, therefore avoiding consistency issues typical of many acceleration schemes and allowing cooperation between SW threads and HW jobs. To support this architectural heterogeneity paradigm, we developed a novel tool flow and design methodology that allows to generate HWPEs semi-automatically starting from normal C code for homogeneous P2012, using commercial high-level and RTL synthesis tools. Our methodology supports a significant fraction of the C syntax and respects SW conventions for pointer manipulation and data layout, therefore allowing shared-memory acceleration for non-trivial accelerated code. Generated HWPE models are integrated in the P2012 platform simulator; this facilitates exploration of various acceleration schemes, early performance assessment and generation of HW using commercial HLS tools.

This architectural approach is promising since it overcomes the scalability issues of ASIP-like architectures while not requiring extensive changes to the application code, and making it easier to integrate accelerator programming in standard shared memory programming models; it makes efficient use of dark silicon since accelerators are triggered only in correspondence of hot regions of code, and then switched off. There are several major differences between GreenDroid [1] (and similar architectures) and our proposal. First, in our case the shared memory is an L1 software-managed scratchpad, and not a cache. This implies a completely different usage of the two basic computation elements (i.e. the cluster in our case, the tile in the case of GreenDroid),

as in our case communication is explicitly managed via DMA transfers. This adds more complexity to the management of data transfers, but allows to extract a much higher degree of performance by hiding data transfer overhead with techniques such as double buffering. A second critical difference is that in the case of GreenDroid the cluster contains a single software core and a sea of c-cores, while in our case we have multiple cores per cluster to maintain the flexibility of the homogeneous P2012 architecture. Finally, contrarily to previous work in this category of accelerators, we propose a full-fledged flow that allows fast design exploration of heterogeneous clusters, with semi-automatic generation of HWPEs and estimation of power and area consumption based on RTL synthesis results.

2.2 Hardware architecture

2.2.1 P2012 homogeneous cluster architecture

Platform 2012 [5] is an effort led by STMicroelectronics to build up a fabric of *tightly-coupled homogeneous clusters*. In the tightly-coupled cluster paradigm (exemplified by many-core accelerators such as P2012 and Kalray MPPA [103]), each cluster is composed by a relatively small number of simple in-order RISC cores (*Processing Elements* or PEs) that communicate through a fast low-latency interconnection to a shared L1 or L2 data memory. Clusters are then connected through a high-bandwidth scalable medium such as a network-on-chip.

Figure 2.1 shows the overall architecture of Platform 2012, with 4 clusters and one fabric controller communicating through an asynchronous network-on-chip.

In P2012, each cluster, as shown in Figure 2.2, contains up to 16 STxP70-v4 cores (PEs) sharing an L1 scratchpad of 256 KB, called tightly-coupled data memory (*TCDM*). Each PE has a 16 KB instruction cache; there is no data cache. The *logarithmic interconnection* between the PEs and the TCDM was designed to allow single-cycle access to the memory banks in case there is no contention (see Rahimi et al.[6]). To minimize access contention, word-level interleaving and a banking factor of 2 were used. The cluster also has a similarly designed *peripheral interconnection* that is used for communication between PEs and peripherals (e.g. DMAs), memory outside of the cluster, and a Cluster Controller containing another STxP70 core.

The P2012 fabric is not meant to be used as a stand-alone computing device, but

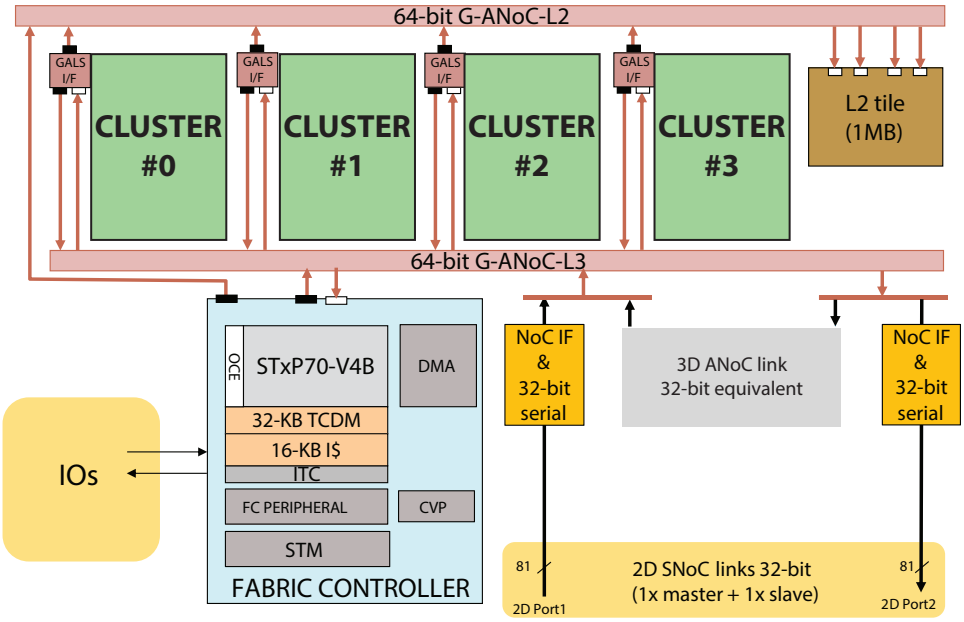


Figure 2.1: P2012 platform overview, with 4 clusters and a fabric controller. Adapted from Marongiu et al. [4].

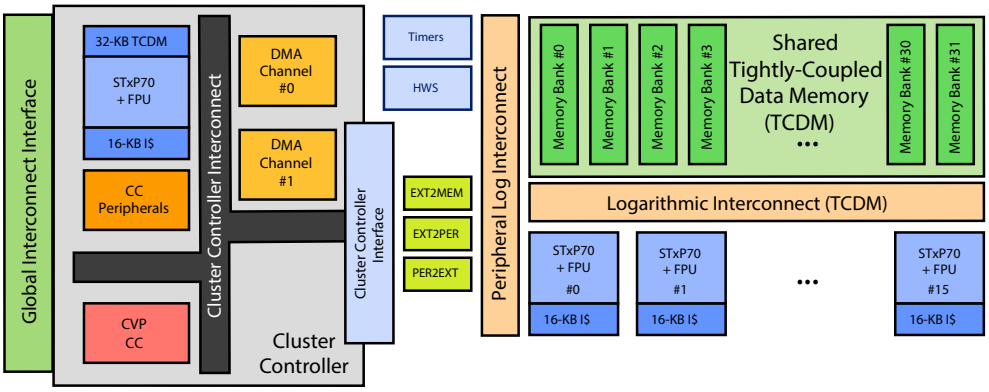


Figure 2.2: P2012 cluster architecture. Adapted from Benini et al. [5].

as a programmable accelerator connected to a ARM host processor. Sequential code is mostly executed on the ARM, while highly parallel workloads are offloaded to the fabric. In its first embodiment, the STHORM board, the P2012 fabric is connected to a Zynq device as a host running a full operating system (Linux or Android), while the fabric runs only a light environment to support the OpenCL and OpenMP programming models.

2.2.2 He-P2012: Heterogeneous P2012

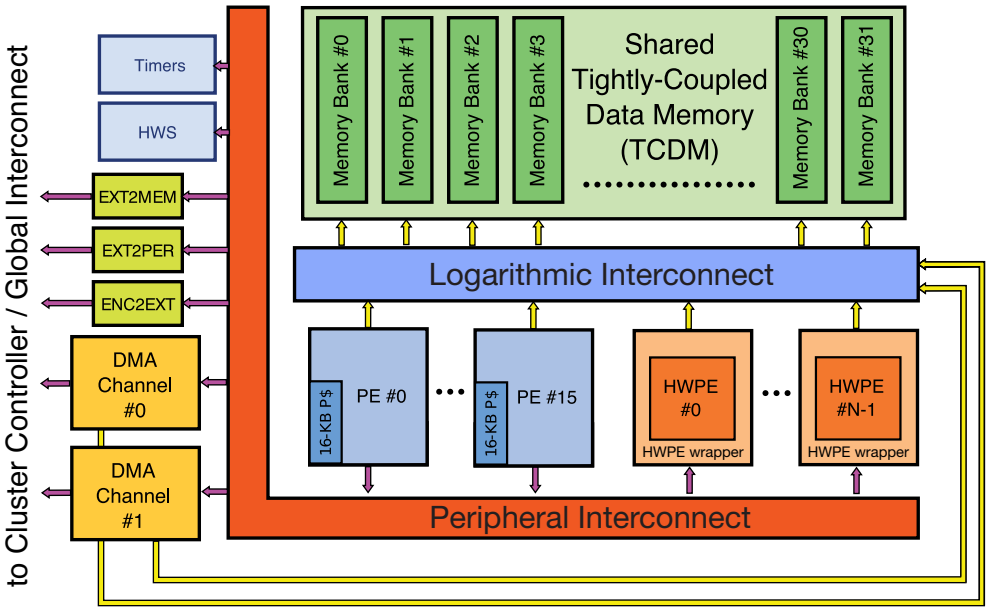


Figure 2.3: He-P2012 cluster: a P2012 cluster extended for heterogeneous computing.

We propose here *He-P2012*, a heterogeneous extension to the P2012 architecture. In He-P2012, the tightly-coupled clusters are augmented with a set of HW accelerators that communicate with PEs via the same shared L1 data memory (a scratchpad) used by the PEs themselves. These tightly-coupled shared memory HW Processing Elements or *HWPE*s [49] address the shortcomings of more traditional copy-based accelerator models, such as the necessity to transfer and maintain coherent multiple copies of data between distinct processor and accelerator memory spaces. In addition, they diminish the semantic gap between executing a task in SW or HW: from a programming perspective, dispatching a HW task on a HW IP is not different from calling a SW function to perform the same task. In fact, the programming model we

propose in section 2.3.1 exposes a synchronous HW interface call that maintains the same interface and semantics of the accelerated SW function, although asynchronous calls may also be used to enhance performance or energy savings if that is needed.

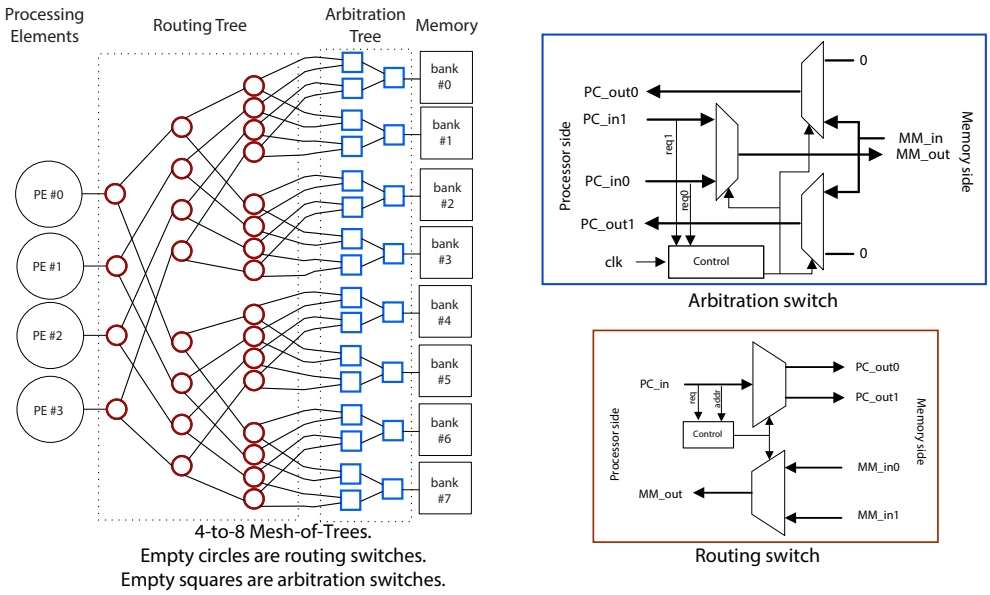


Figure 2.4: Logarithmic interconnect with 4 masters and 8 memory banks (adapted from Rahimi et al. [6]).

Each HWPE is treated as an additional master with one or more ports on the cluster logarithmic interconnection, exactly as if it was an additional core in the cluster. Figure 2.4, which was adapted from Rahimi et al. [6], shows the logarithmic interconnect in a simple configuration with 4 masters and 8 slaves (memory banks). The logarithmic interconnection manages synchronization on memory accesses via a simple round-robin scheme to avoid starvation [6]: access to a contended memory location is granted to a single master, while the other masters are stalled for one cycle. Contention on accesses from HWPEs and software PEs (or between those from different HWPEs) is treated in the same way as in the homogeneous cluster. The performance loss due to contention depends on the computation-to-communication ratio of PEs and HWPEs and on the architectural parameters of the shared memory, such as its banking factor. In Figure 2.5 we plot the results of a synthetic benchmark showing the execution time overhead of a 4-master port HWPE running in a cluster with 16 PEs, while varying the probability of a memory access for both each PE and each HWPE port. The maximum overhead is ≈70% when all 16 PEs have

$P_{\text{mem op PE}} = 90\%$ and each HWPE port has $P_{\text{mem op HWCE}} = 90\%$; this can be considered a worst-case scenario where all 16 PEs are essentially being used for data movement only. In a more realistic high-contention scenario, with $P_{\text{mem op PE}} = 50\%$ and $P_{\text{mem op HWCE}} = 80\%$, overhead is reduced to ≈30%. This is consistent with our empirical observations on the tests shown in Section 2.4. In many cases, software computation-to-communication ratio is much higher and as a consequence the overhead introduced by memory contention is less than 15%. [1] All results reported in Section 2.4 include modeling of memory contention.

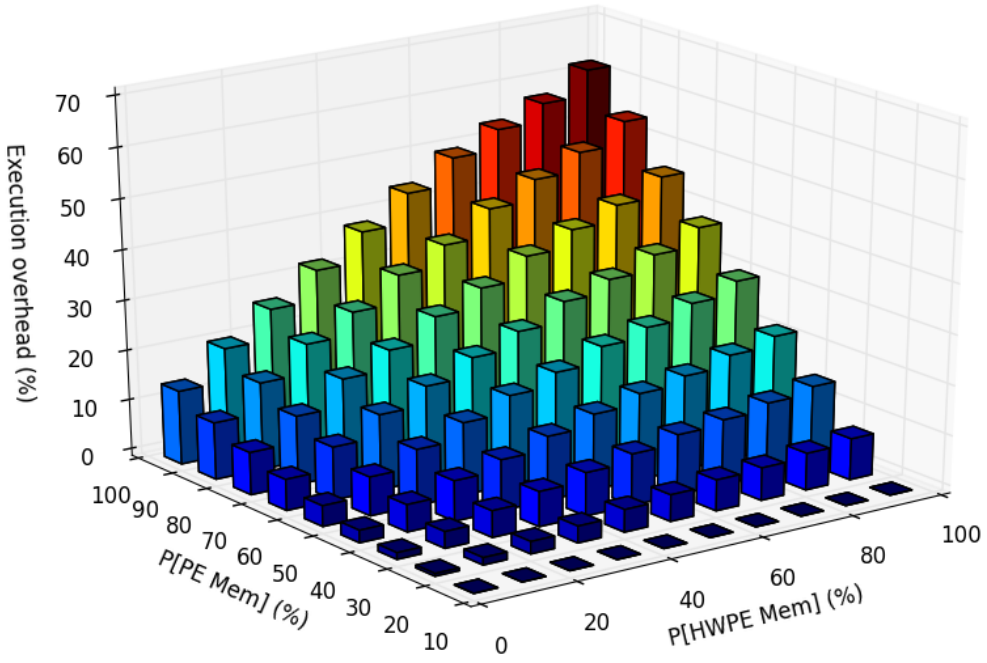


Figure 2.5: Overhead due to memory contention on a synthetic benchmark.

HWPEs are also connected as targets to the peripheral interconnection for control. Figure 2.3 shows a diagram of the He-P2012 cluster extended with HWPEs for heterogeneous computing. HWPEs are designed as two separate modules:

1. the *HW IP* or accelerator proper is a datapath that implements the accelerated function. It can be designed using high-level synthesis from a C function.

2. the *wrapper* provides the HW IP with the capability of accessing shared memory

[1] A more detailed discussion regarding the architectural tradeoffs of shared-memory HWPEs versus private-memory ones can be found in Dehyadegari et al. [50].

through a set of ports on the logarithmic interconnect and of accepting jobs from the SW processing elements into a job queue.

The HW IP can be developed directly from the original software function by means of standard methodologies (e.g., HLS tools [39][104]) with no additional programmer effort. High-level synthesis produces an IP that can read and write to an external memory using a custom protocol based on address and handshake. It is possible to build HW IPs with more than one port for enhanced memory bandwidth. An important limitation of current HLS tools is that they do not support IPs dynamically addressing data in the external memory: all transactions happen in a private memory space, where data is placed statically (i.e. at design time). This limitation is overcome by connecting the HW IP to the HWPE wrapper.

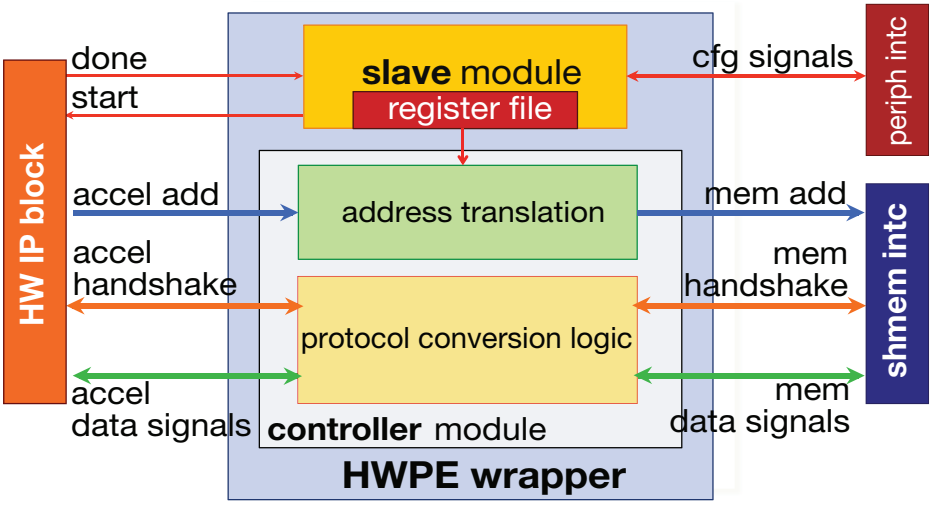


Figure 2.6: Simplified architecture of the HWPE wrapper.

The wrapper, shown in Figure 2.6, was designed to take care of three tasks:

- translating the memory accesses from the custom handshake protocol produced in the HLS tool to the one used in the shared-memory interconnection;
- augmenting the memory transactions so that they support data placed dynamically (i.e. at run time) in the shared-memory;
- providing a control interface through which the SW PEs can offload jobs to the HWPE.

The wrapper is divided in a *controller* submodule that translates HW IP transactions into shared memory ones and a *slave* module hosting a register file to provide the control interface.

To offload a new job to the HWPE, a SW PE must i) take a lock on the HWPE (so that no other PE can concurrently offload a different job) by accessing a special register in the register file, ii) write the base addresses of I/O data in the HWPE register file and iii) trigger the execution (and release the lock) by accessing another special register. The controller then uses the addresses in its register file to remap memory accesses from the HW IP to the right region in the shared memory using the address translation block. In the current implementation, the register file in the wrapper can host a queue of up to 4 jobs that can be offloaded to the HWPE also when the HW IP is busy, to minimize the offload overhead; as soon as one of the jobs is completed, the next one is started.

The HLS tool produces a cycle-accurate SystemC model for simulation and a Verilog model that can be used for both RTL simulation and synthesis; these can be used in conjunction to a SystemC or SystemVerilog implementation of the wrapper, respectively.

2.3 Exploration flow

The final goal of the tool-flow presented here is to enable fast design space exploration of heterogeneous architectures. To achieve this goal we need an exploration tool that is easy-to-use, and that leverages a clear, well-defined methodology. *GEPOP* (Generic Posix Platform), the simulation infrastructure provided with the official P2012 SDK, provides a convenient starting point to achieve this goal. Clearly, it only models the original homogeneous P2012 clusters, so we extended it to support accurate simulation of HWPEs, wrapped with the discussed tightly-coupled shared memory methodology. Models of all HWPEs are encapsulated in a single SystemC model that runs as a separate process. A GEPOP plugin manages communication between this model and the rest of the simulation platform, by means of UNIX pipes.

As our target is to explore the costs and gains related to the tightly-coupled shared memory acceleration model on practical applications, in these experiments we generate separate simulation platforms for each application. Based on the results of this

exploration, it would be possible to define which subset of accelerators explored are definitely useful for a whole class of applications. We simplify simulator generation by automating essentially all the steps required to create an executable simulation. The user is exclusively responsible for:

1. writing the main application source code, using one of the programming models supported in homogeneous P2012 (i.e., OpenCL or OpenMP);
2. defining the HW/SW interface in a custom language (SIDL, see Section 2.3.1);
3. extracting the functions to be accelerated into separate C files.

The tool flow uses this information to automatically generate all of the underlying glue code that is required to execute the application on our heterogeneous platform.

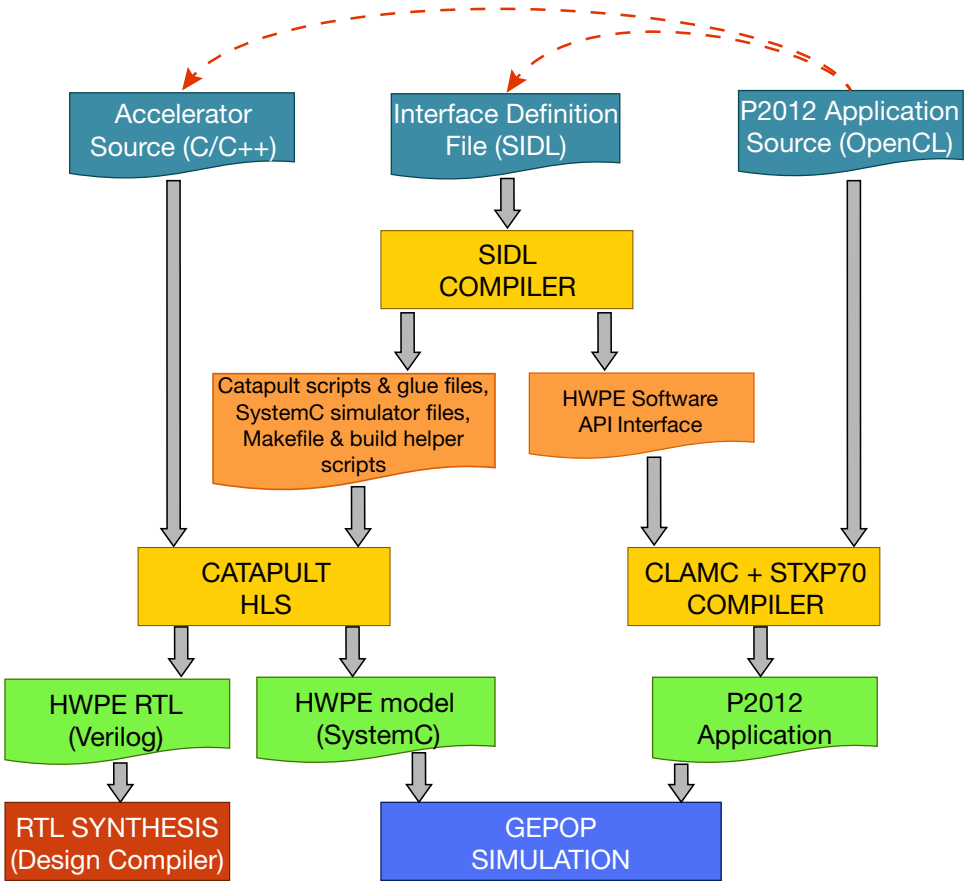


Figure 2.7: He-P2012 flow for cycle-accurate HWPE simulation.

HLS and RTL synthesis scripts, SystemC wrappers for the HWPEs and SW API calls to be used in the application code are all generated from the interface, which is described in a custom language called *SIDL* (Software/hardware Interface Definition Language). Figure 2.7 shows how the SIDL compiler takes the HW/SW interface specification, and produces information for both the HW and the SW sides of the flow. The compiler produces several TCL scripts to drive the HLS and RTL synthesis tools (Calypto Catapult and Synopsys Design Compiler, respectively). Moreover, it also generates a set of C++ classes that are used to support the shared-memory interface between HW and SW in the Catapult HLS tool, as described in Section 2.3.2, and the top-level SystemC wrapper that encapsulates all HWPEs. Finally, it also generates the C API (application programming interface) that can be used by the OpenCL or OpenMP kernels in the GEPOP simulation to call for HW-accelerated execution of a function.

It is worth mentioning that our flow also supports a less accurate simulation mode, in which the HWPEs are modeled at a purely functional abstraction level, with accelerated portions of code executed in zero time.. This can be useful at early design stages to quickly get a feeling of the behavior of different SW/HW partitioning schemes.

2.3.1 Software/Hardware Interface

The tools model the interaction between software and HWPEs as a client/server communication, similarly to what is done in DSOC [105]; in our case, the client is the SW running on the He-P2012 processing elements and the server are the available HWPEs. The client/server interface is expressed in *SIDL* (Software/Hardware Interface Description Language), a subset of C++.

A SIDL source file is composed by a *preamble*, in which structured data types are declared using C++ `struct` syntax, and a series of *interfaces*, which are declared as the virtual methods of a pure virtual class (e.g., class `hwpe`). Since the HWPEs share data exclusively through the shared TCDM, inputs and outputs of the HW IP are all expressed by means of pointers to simple or structured data types, as when passing arguments by reference to a C function. The following is the SIDL code describing the interface for two functions called `foo` and `bar`:

```
// library interface function specification (SIDL)
class hwpe {
```

```
    #pragma sidl unroll   core/inner_loop 4
    #pragma sidl pipeline core/main_loop  2
    virtual void foo(int *in, int *out) = 0;

    #pragma sidl unroll   core/main_loop
    virtual void bar(int *in, int *out) = 0;
};
```

The SIDL source may also contain a small set of `pragma`s that are used to direct the HLS tools in a relatively easy way. Table 2.1 lists the supported `pragma`s.

`#pragma sidl`	**Description**
`nb_hwpe N`	duplicates the HWPE `N` times
`nb_master N`	uses `N` shared-mem ports
`unroll LOOP N`	unrolls `LOOP` `N` times
`pipeline LOOP N`	pipelines `LOOP` with init. interval `N`
`directive DIR`	passes `DIR` directive to Catapult

Table 2.1: SIDL Pragmas.

Following the classical client/server IDL approach, we developed a compiler to process the SIDL interface definition and generate all downstream files as shown in Figure 2.7 from a set of templates. The SIDL compiler is based on open-source tools (`flex`, `bison` and `libtemplate`) and easily extensible.

On the client (i.e. SW) side, the result of the SIDL compilation is a set of APIs for a light-weight runtime environment for the HWPE that can be used instead of the native low-level HAL that drives the HWPE wrapper module. These APIs are expressed in pure C, and they are therefore compatible with all the programming paradigms available for the P2012 platform, such as the OpenMP and OpenCL programming models. The tools provide both a synchronous interface call, in which the client blocks until the results are available, as well as an asynchronous one that returns immediately. In the latter case, the SIDL compiler generates also APIs to wait for the end of the HWPE computation or to check if it has ended.

2.3.2 High-Level Synthesis

Due to the shared-memory nature of our architecture, argument marshaling is not needed to collect the client arguments, and it is not necessary to copy data from the

client to the server (i.e., from the SW-accessible memory to the HWPE). Rather, data is left in-place, and only pointers are exchanged between the client and the hardware accelerator. This allows a very low-overhead client/server call.

As a high-level synthesis tool, we decided to use Calypto Catapult System-Level Synthesis as it supports out-of-the box most of the C and C++ syntax, as well as SystemC. However, to make better use of the shared-memory nature of He-P2012, we also need to support the conventions used by the SW compiler for data layout: transforming by software the client data to a representation usable by the hardware is not desirable - the overheads of data reformatting would defeat the advantages of our shared-memory architecture. Manually modifying the HLS code to explicitly deal with the layout used by the software compiler is also undesirable, as it would require discarding all higher-level data structures. It would also lower the general abstraction level, which is highly undesirable as this would defeat part of the purpose of our flow.

Therefore, we introduced a C++ template-based technique that statically maps simple and structured data types used in the source code of the HW IP to a lower-level, word-based representation that respects the conventions set by the SW compiler. This way, normal data access syntax in the HW IP source is maintained, while correct data addressing in the shared-memory is also guaranteed.

At the lowest level, all inputs/outputs are represented by an array of `sc_uint<36>` (i.e., 36-bit SystemC unsigned integers) called *backing store*. The highest four bits of the backing store represent a byte enable signal, whereas the lowest 32 bits are the actual data word. The 36-bit backing store is used to abstract the actual shared-memory concept from the HLS tool, which is not meant to support such a scenario but can easily handle non-standard word sizes such as 36 bits. To represent the address of a particular word in the backing store, a `hwpe_addr` class is used. Note that, as the HLS tool does not support dynamically positioning data in the shared-memory, this class merely represents an offset with respect to the base of the input/output array pointed by the backing store.

To map atomic (i.e. non-structured) types to the underlying backing store, a template C++ `hwpe_atomic<T>` class is used, where `T` is the C atomic type (e.g. `int16_t`). From the moment of its construction, a `hwpe_atomic` instance features a pointer to a underlying backing store and an offset value. Assignment and access operators on `hwpe_atomic` classes are overloaded so that, depending on the byte width of the high-level type `T`, the correct byte enables are activated (in case of a store operation) or

the correct part of the data word is retrieved (in case of a load). All standard integer types are supported: signed and unsigned 8-bit, 16-bit, 32-bit and 64-bit integers. The following code snippet shows some parts of the `hwpe_atomic` class.

```
class hwpe_atomic<T> {
  sc_uint<36> *backing_store;
  int byte_offset;
 public:
  const T operator=(hwpe_atomic &atomic) {
    return operator=(atomic.operator T());
  }
  operator T() {
    int bs_offset = byte_offset / 4;
    int byte_idx = (byte_offset % 4)/sizeof(T);
    // for brevity, only sizeof(T) = 4 case is shown
    return (T) backing_store[bs_offset].range(31,0);
  }
  hwpe_ptr_atomic<T> operator&() {
    return hwpe_ptr_atomic<T>(backing_store, byte_offset);
  }
};
```

To provide the shared-memory interface, a *smart pointer* `hwpe_ptr_atomic<T>` template class is also created corresponding to each `hwpe_atomic` class, to represent a pointer to the atomic type. Similarly to the `hwpe_atomic` class, `hwpe_ptr_atomic` features a pointer to the underlying backing store and an offset. Pointer and array access operators (`*` and `[]`) are overloaded to return a `hwpe_atomic` instance; conversely, the `&` operator of `hwpe_atomic` is overloaded to return a `hwpe_ptr_atomic` instance.

For each structured type present on the interface of the HWPE, two template C++ classes are generated by the SIDL compiler. Let us suppose that the higher-level data structure is the following `structured_t`:

```
typedef struct {
  uint32_t foo;
  int16_t bar;
  uint16_t who;
} struct_t;
```

To represent it, the SIDL compiler generates a C++ class that includes `hwpe_atomic` types for each of the members of the higher-level struct:

```
class hwpe_struct_t {
```

```
  sc_uint<36> *backing_store;
  int byte_offset;
 public:
  hwpe_atomic<uint32_t> foo;
  hwpe_atomic<int16_t> bar;
  hwpe_atomic<uint16_t> who;
};
```

This class also overloads the field access operator (`.`) to allow access to its fields. Finally, a smart pointer template class `hwpe_ptr_struct<T>` allows pointer access to the structured data type by overloading pointer, array access and field access operators (`*`, `[]` and `->` respectively). For convenience, the SIDL compiler defines new types for all pointers used in the interface of the HWPE:

```
typedef hwpe_ptr_atomic<uint32_t>      hwpe_ptr_uint32_t;
typedef hwpe_ptr_struct<hwpe_struct_t> hwpe_ptr_struct_t;
```

In this way, most features typically required from accelerated hardware are supported: simple data structures, random access to the shared memory, data-dependent control flow, inlined function calls. The main limitation, which is imposed by the fact that Catapult does not support dynamic addressing, is that it is not possible to dynamically reference data: double pointers are not supported, and therefore all structures must be purely composed of atomic data types. Calling inlined functions is supported as a means of code organization; it is not possible to call functions to be executed in software from the accelerator, nor is it possible to use function pointers.

From the perspective of the designer of the HW IP source code, only minimal changes are strictly needed to the HW IP source code to achieve a functional accelerator, though naturally to achieve better performance it is sometimes necessary to perform optimizations. The following code snippet shows an example of the transformation of a simple function from the regular C pointer syntax to the Catapult smart pointer one.

```
// Function with regular C pointer
void add_one(int *the_value) {
   *the_value = *the_value + 1;
}
// Function with Catapult smart pointer
void add_one(hwpe_ptr_int the_value) {
   *the_value = *the_value + 1;
}
```

In this example, `the_value` is pointing to data in the shared memory space. In the Catapult implementation the `hwpe_ptr_int` type must be used in the function declaration, as in the second case. The `hwpe_ptr_int` type generated by the SIDL compiler can be used like an integer pointer, i.e., it seems to the HW IP developer that this type was defined like `typedef int *hwpe_ptr_int`. This modification must be carried out in all pointer types referring to data placed in the shared memory.

To perform the high-level synthesis of a HW IP the developer must therefore only *i*) define the SW/HW interface, *ii*) adapt the HW IP source code from the original SW version so that it uses smart pointers and *iii*) optionally optimize the HW IP for better performance. The Catapult backend is called automatically from a lightweight `make`-based wrapper that is used to automate the flow. HWPEs generated with this methodology show only a minor area and power increase with respect to those generated directly inside Catapult. For example, the size of the CSC HWPE (see Section 2.4) increases by 6.3% with respect to the one that would be obtained directly, while power increases by 3%. We believe this increase is well worth the gain in generality and the speedup in design time; moreover, as this flow is meant to be the first exploration step in the definition of a particular He-P2012 version, it is possible to use it only as a methodology for the fast assessment of the best acceleration strategy, switching then to a manually tweaked HWPE. One must note, however, that data types different from 32-bit integers in the shared memory cannot be directly supported in Catapult without manually replicating much of what is automatically generated by our flow.

2.3.3 Power/Area Estimation and Simulation

The high-level synthesis tool produces two outputs: a synthesizable Verilog model and a cycle-accurate SystemC model of the HW IP. After high-level synthesis, RTL synthesis of the HWPE (including both the HW IP and the wrapper) can be performed using Synopsys Design Compiler. The SIDL compiler generates all the scripts necessary to fully automatize the process. Estimations of area and power consumption can be extracted from the Design Compiler synthesis.

To evaluate the execution time, the SystemC models of all HW IPs are encapsulated in a single executable together with a model of all the wrappers. The SystemC model and the power estimation are loaded by a GEPOP plugin to estimate the energy

consumed by the platform in the heterogeneous configuration. We used the following simple energy model:

$$
\begin{aligned}
E_{\text{tot}} &= E_{\text{uncore}} + E_{\text{PEs}} + E_{\text{HWPEs}} \\
E_{\text{uncore}} &= P_{\text{uncore}} \cdot t_{\text{exec}} \\
E_{\text{PEs}} &= P_{\text{PE}} \sum_{\text{PE}_i} t_{\text{on,PE}_i} \\
E_{\text{HWPEs}} &= \sum_{\text{HWPE}_j} P_{\text{HWPE}_j} \cdot t_{\text{on,HWPE}_j}
\end{aligned}
$$

The total energy spent in the cluster is due to three components: the energy spent in uncore devices (i.e. everything in the cluster that is not a PE or a HWPE, e.g. DMAs), which are supposed to be on during the whole execution time; the energy spent in PEs; the energy spent in HWPEs. Energy spent in memory access is collapsed inside the "uncore" category and was measured in a worst-case scenario, using results provided by STMicroelectronics. We assume PEs and HWPEs to be clock-gated and consume no leakage power when idle. This simplistic power model has the goal to facilitate design space exploration and to compare different architectural solutions, not to provide an accurate estimation of the full-system power. We argue that the model is sufficient for this purpose, as (due to the complexity of the cluster) it would be necessary to resynthesize the cluster from scratch down to the bottom of the backend flow, with full synthesis of the clock tree, placement and routing to provide really accurate results in terms of power.

2.4 Results

In this section we explore various versions of the He-P2012 cluster on a set of benchmarks that were originally developed for the homogeneous P2012. Note that the focus here is not on optimizing the HW blocks, which we simply derived from C code with minimal modifications, but on showing the effects of augmenting the P2012 architecture with heterogeneity by adding tightly-coupled accelerators.

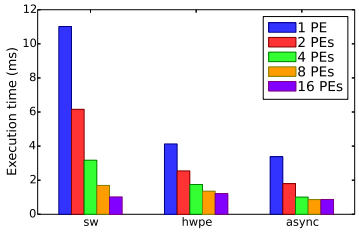
For the purpose of this work, we concentrated our analysis of performance and energy to a single P2012 cluster clocked at 400 MHz, where we sweep the number of SW and HW processing elements present in the cluster. In fact, communication

between clusters and with the host is dealt with in the same way in both homogeneous and heterogeneous clusters and is an orthogonal issue with respect to our focus.

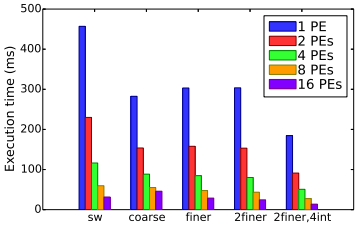
For our experiments we consider six applications that were originally developed for the homogeneous P2012 cluster: Viola-Jones Face Detection using Haar features [106], FAST Circular Corner Detection [107][108], Removed Object Detection based on normalized cross-correlation (NCC) [109], a parallel version of Color Tracking from the OpenCV library, a Convolutional Neural Network (CNN) forward-propagation benchmark [76] [110], and a Mahalanobis Distance (10-dimensional) kernel. Face Detection is parallelized with OpenCL, while the other benchmarks use the OpenMP implementation described in Marongiu et al. [4]. Four of the applications we chose to focus on (Viola-Jones, FAST, ROD, and Color Tracking) are from the computer vision field, while the two remaining ones (CNN, Mahalanobis Distance) are more generally from the machine learning field, though vision is one of their main uses. We chose to focus on applications from this field for several reasons: first, they are typically very demanding from the computational point of view and feature high computation-to-communication ratios, making them well suited for HW acceleration. Second, these kernels are often used in pipelines to build complete applications; this makes them an ideal target for our heterogeneity paradigm, that is highly focused on cooperation between software and hardware cores where we could run different stages of a pipeline. Finally, they are among the main applications targeted by the homogeneous platform we built upon (P2012), which allowed us to directly compare the homogeneous and heterogeneous platforms. This set of applications differs significantly in size, execution time and structure, with the objective of demonstrating our flow on a wide range of applications.

Color Tracking (**CT**) is composed of three kernels: color scale conversion (`CSC`), color based threshold (`TH`) and moment based center of gravity computation (`MOM`). The most computationally intensive kernel is CSC, which we chose for HW acceleration. The application is structured as a software pipeline made up of a DMA-in stage, a `CSC` stage and a `TH`+`MOM`+DMA-out stage. It is possible to use both synchronous and asynchronous calls for the accelerated `CSC` kernel; in the latter case, the accelerated `CSC` execution can be almost completely hidden behind the `TH`+`MOM` stage.

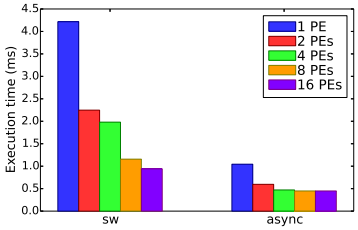
Viola-Jones object recognition (**VJ**) is the most complex of our set of benchmarks; it includes two computation-intensive kernels: `cascade`, whose execution time might vary on a scale of 3 orders of magnitude, depending on input data; `integral image`,



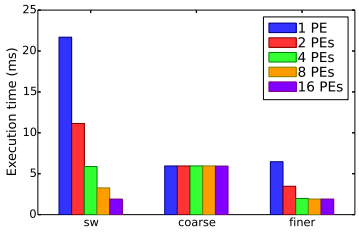
(a) Color Tracking



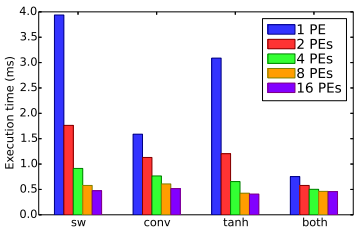
(b) Viola-Jones Face Detection



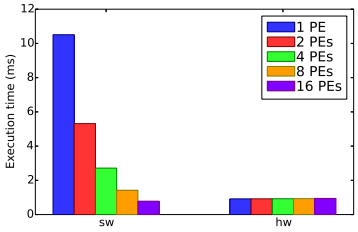
(c) FAST Corner Detection



(d) Removed Object Detection

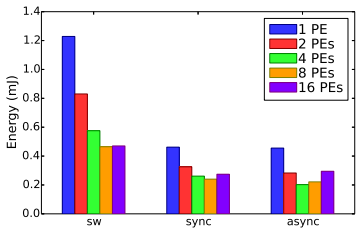


(e) Convolutional Neural Network

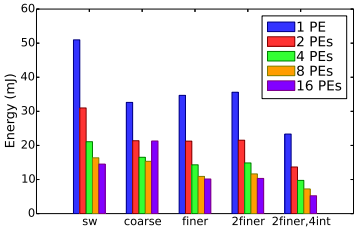


(f) Mahalanobis Distance

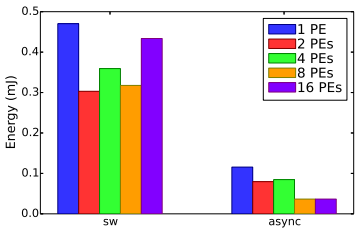
Figure 2.8: Execution time.



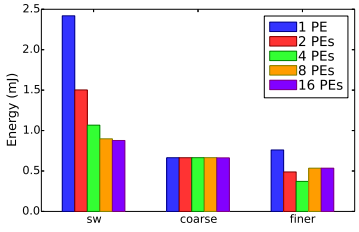
(a) Color Tracking



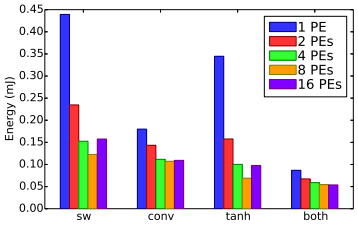
(b) Viola-Jones Face Detection



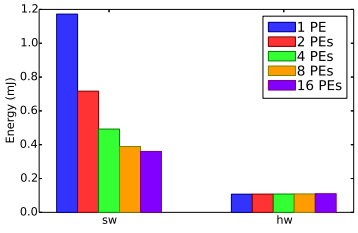
(c) FAST Corner Detection



(d) Removed Object Detection

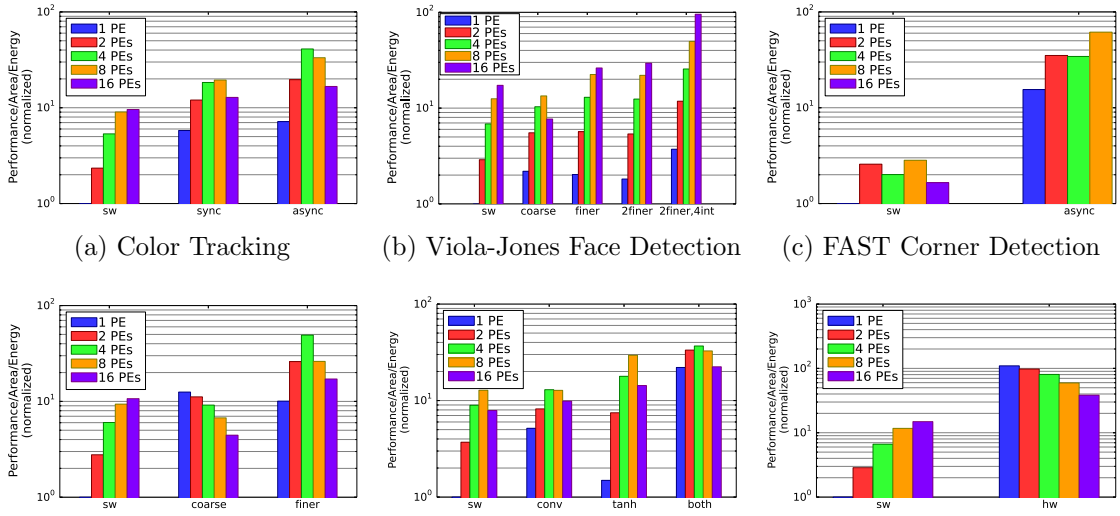


(e) Convolutional Neural Network



(f) Mahalanobis Distance

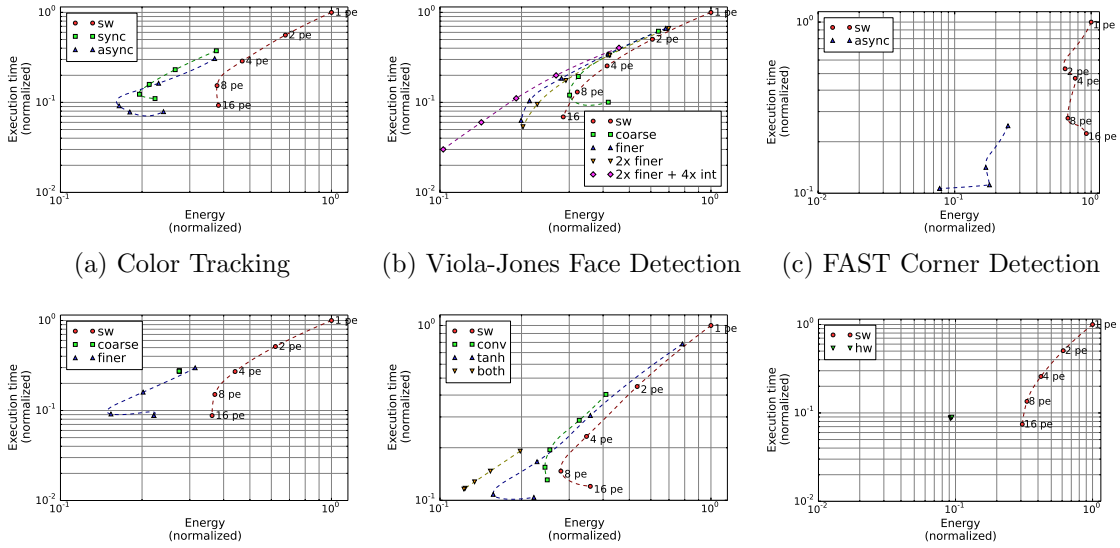
Figure 2.9: Energy.



(a) Color Tracking (b) Viola-Jones Face Detection (c) FAST Corner Detection

(d) Removed Object Detection (e) Convolutional Neural Network (f) Mahalanobis Distance

Figure 2.10: Performance/Area/Energy.



(a) Color Tracking (b) Viola-Jones Face Detection (c) FAST Corner Detection

(d) Removed Object Detection (e) Convolutional Neural Network (f) Mahalanobis Distance

Figure 2.11: Energy vs Execution Time.

whose execution time is not strongly dependent on input. For this benchmark, we developed three HWPEs.

The `cascade` kernel is composed by a series of 13 stages, each of which is more expensive than the previous one. Only matches have to go through the full cascade; non-matching windows usually get out of the cascade at an earlier stage. In our experiments, up to 95% of the windows got out of the cascade in the first two stages. Therefore, we developed two distinct strategies to accelerate the `cascade` kernel: the first strategy consists in accelerating the whole cascade in HW (*coarse* HWPE); the second in accelerate only stages from the third on (*finer* HWPE), i.e. only in the least common cases.

FAST circular detection (**FAST**) is composed of two kernels, `detection` and `score`. To accelerate the application, we merged them in a single HWPE, that is called asynchronously.

Removed Object Detection (**ROD**) spends most of its execution time in the normalized cross-correlation kernel (`NCC`). We consider two approaches for acceleration: a *coarse* HWPE that accelerates all the iterations, and a *fine* HWPE that accelerates only one iteration. In the first case, only one PE is actually used; in the second case the code containing the offload call to a HWPE can be parallelized among all the threads.

Convolutional Neural Network (**CNN**) is composed of a series of convolutional, max-pooling and linear layers. Convolutional layers are responsible for the great majority of execution time and energy consumption; we therefore concentrated on accelerating their two main kernels: convolutions (*conv* HWPE) and hyperbolic tangents (*tanh* HWPE), both used in an asynchronous fashion. We simulated configurations with one of the two kind of HWPEs, or both.

Mahalanobis Distance (**MD**) is composed by a single 10-dimensional distance kernel, which was fully HW-accelerated in the heterogeneous configuration.

We collect results for three different metrics: execution time, energy, performance/area/energy. Performance is measured as inverse execution time (as it is seen by the controlling PEs), while energy is estimated using the model described in Section 2.3.3. We measure execution time To obtain area and power values for the HWPEs, we used Calypto Catapult University Version 2011.a62 for high-level synthesis and we synthesized its RTL output together with a SystemVerilog wrapper using Synopsys Design Compiler G-2012.06. We used the 28 nm bulk STMicroelectronics technology

libraries as a target, with a clock frequency of 400 MHz. Table 2.2 reports power and area estimations for all HWPEs used in these benchmarks.

HWPE	**Power** (mW)	**Area** (kgates)
CT	5.82	16.74
VJ *coarse*	12.58	44.98
VJ *finer*	7.86	27.20
VJ *int*	4.88	19.79
FAST	4.55	13.88
ROD *coarse*	4.98	17.74
ROD *fine*	5.04	17.52
CNN *conv*	7.99	26.67
CNN *tanh*	2.43	7.10
MD	5.59	19.06

Table 2.2: Design Compiler synthesis results.

To better understand how much acceleration is achievable with HWPEs, we introduce a new metric called *accelerator efficacy* ξ whose goal is to compare actual accelerators versus a perfect accelerator by Amdahl's law, i.e. one that reduces the accelerable fraction of our application to 0 time. Acceleration efficacy is defined as the following:

$$\xi_{\text{hwpe}} = \frac{T_{\text{sw}} - T_{\text{Amdahl}}}{T_{\text{sw}} - T_{\text{sw+hwpe}}}$$

Figure 2.12 shows the accelerator efficacy ξ for the accelerators we considered in our study. We can reach an average efficacy of $\sim 80\%$, with some lower-efficacy accelerators such as VJ stopping at $\sim 60\%$ and others, especially those exploiting asynchronicity between SW and HWPE execution, reaching up to 98%.

In Figures 2.8, 2.9, 2.10 we show results for the execution of our six benchmarks in terms of execution time, energy spent in the cluster and normalized performance/area/energy respectively. From these results we can drive some interesting observations on the class of accelerators that can be generated from our flow. First, as shown by the *VJ* benchmark, acceleration of coarse code blocks is best controlled by a small number of PEs. Long-running HWPEs are clearly more likely to generate high HWPE contention in cases where the code parallelized among a large number of PEs does not contain much computation to be done in SW (besides the code to offload computation). HWPE contention can be seen in most benchmarks; when the number of PEs is high the execution time does not scale down any more, thus limiting

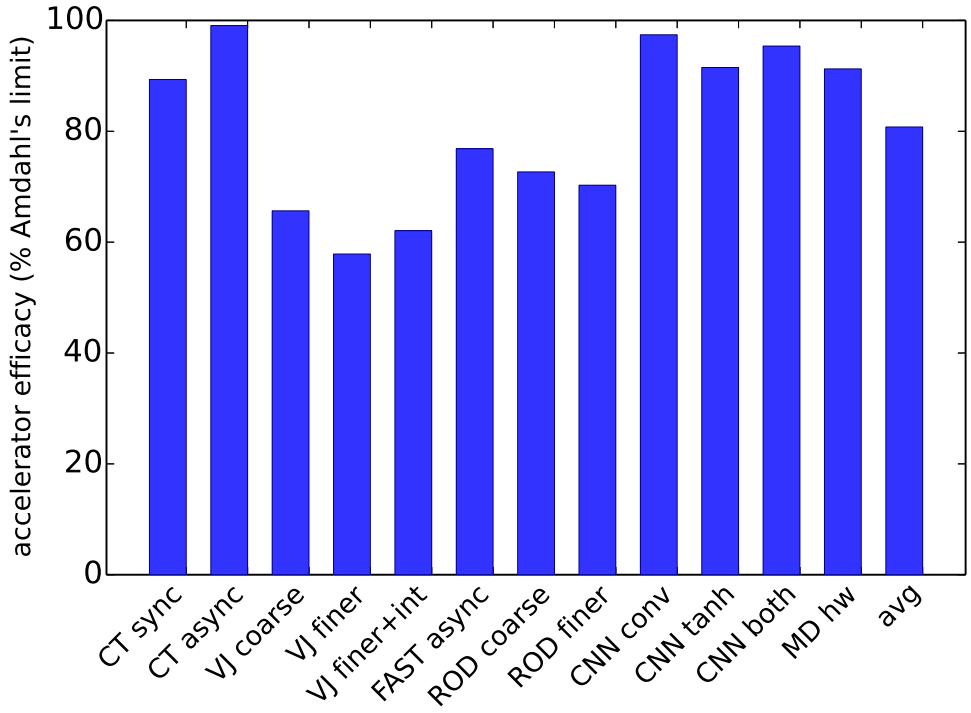


Figure 2.12: Accelerator efficacy with 1 PE + HWPEs in % of Amdahl's limit.

performance/area/energy, which shows a peak at 4 or 8 cores in most tests.

A second effect is that when the number of PEs is relatively low, area and power consumption are mostly reduced to the invariant contribution of other components of the cluster (DMAs and external interfaces). Therefore, adding PEs or HWPEs to the cluster improves performance without significantly worsening area and power, whereas when the number of PEs is higher the reverse may be true, as they scale approximately with the number of PEs and HWPEs. The combination of the effect of HWPE contention and of the linear scaling of power and area with number of processing elements can be seen in Figure 2.10, where the performance/area/energy in most tests reaches its maximum with 4 or 8 PEs.

Third, applications structured as pipelines, such as *CT*, benefit the most from this acceleration model because it is possible to exploit asynchronous HWPE calls and to hide most of the accelerated execution behind the rest of the application.

In the single-PE *CT* case, asynchronous execution allowed us to almost fully exploit `CSC` acceleration, obtaining a speedup of 123×, that is 98% of the maximum permitted by Amdahl's law. The same benchmark shows an increase of almost an order of

magnitude in performance/area/energy; the *FAST* benchmark, which is also relatively efficient in term of Amdahl's limit, goes even farther achieving a 21× increase in that metric in the measurement with 4 PEs.

Figure 2.11 helps understanding the energy-delay tradeoffs implied by the choice of a particular homogeneous or heterogeneous architecture, by reporting normalized execution time versus energy spent in each benchmark while we sweep the number of SW PEs from 1 to 16. Many plots exhibit near-ideal performance scaling in SW; however, it is clear that in most benchmarks that only a certain amount of SW parallelism can be extracted in an energy-efficient way, as efficiency drops considerably when using more than a certain number of PEs (between 4 and 8, depending on the benchmark). This is due to the fact that performance scaling is never completely ideal and usually saturates at a certain threshold, while power scaling is always linear; this is clearly shown in the plots, where most curves bend rightwards after 4/8 PEs.

In the *CT* and *VJ* benchmarks and in the *finer* configuration of the *ROD* HWPE jobs are offloaded concurrently by all PEs, which causes two visible effects: the first is HWPE contention by PEs, which is particularly visible in the *coarse* configuration of *VJ*. The second, which is shown in both *CT* and *VJ*, is the "constructive interference" of SW and HW co-working, which leads to significant overall speedups even when we are using the most SW parallelism available (up to 6× overall in *VJ*) or to complete acceleration (i.e. reaching Amdahl's limit) in the case of *CT*. We also see that in *ROD finer* there is a hard limit to performance (both in SW and HW) given by memory bandwidth: bringing data in/out of the cluster via DMA transfer becomes the bottleneck.

The *ROD coarse* configuration and the *CNN* and *MD* benchmarks show instead what happens when HWPEs are called by a single PE. In *ROD coarse* and *MD*, we have no scaling at all as the whole application is accelerated. In *CNN* it is possible to employ SW cores concurrently with HWPE execution, which is started by a single PE. It is interesting to note that though convolution is responsible for more than 80% of the total execution time, it is difficult to achieve a top-grade accelerator using our HLS flow in this case, because the efficient way of expressing convolutions in SW is fundamentally different from the top-efficiency way to design them in HW [8]. For this reason, the single accelerator we use in the *conv* configuration is not able to compete with 16 PEs performance-wise. Conversely, accelerating the hyperbolic tangent in the *tanh* configuration is very easy and effective using our flow, even if the hyperbolic

tangent is not as critical a kernel as the convolution is. Accelerating both kernels leads to the best results in terms of energy efficiency, but makes it impossible to exploit SW scaling to reach even better performance and energy; the convolution accelerator acts as the bottleneck in the 16 PE *both* configuration.

Finally, the *FAST* benchmarks shows what happens to a very unbalanced application. We used a synthetic image (a checkerboard) as input to the benchmark; this tends to exacerbate the imbalance as, depending on the total number of SW threads and the subsequent chunking of the image, some cores will consistently have no corners and some others many corners to find. The plot clearly shows that neither SW-only nor HW-augmented scaling are ideal; however, we also see that using an accelerator in this case is helpful as it reduces the difference between threads with many corners and threads with none, reducing the imbalance. This results in slightly improved scaling and significant gains in terms of both energy and performance.

2.5 Conclusions

The novel methodology and tools we developed, oriented at heterogeneity exploration on the tightly-coupled shared-memory He-P2012 platform, allow fast and easy exploration of a number of hardware acceleration alternatives. We have shown that with our heterogeneous approach it is possible to obtain a significant advantage with respect to pure software in terms of performance, energy consumption and performance/area/energy. This was achieved without major rewriting of any benchmark, since we adopted the viewpoint of augmenting the multicore cluster of P2012 with heterogeneity rather than completely changing its working paradigm.

Design space exploration and early assessment of the benefits of one acceleration solution over the others are critical to help overcome the utilization wall by means of architectural heterogeneity. This chapter demonstrates a technique to apply HW acceleration to a wide range of SW kernels and exemplifies it in the case of a state-of-art many-core platform, P2012. Although the flow we have shown is specific to P2012, we argue that the underlying heterogeneity approach and the proposed methodology are applicable to the entire class of clustered many-cores (e.g. Kalray MPPA [103]). Chapter 4 will show how it is possible to use the architectural paradigm of tightly-coupled shared memory acceleration we proposed in this chapter in a com-

pletely different scenario, i.e. integration of a manually designed heterogeneous core specialized in the acceleration of CNNs.

Chapter 3

PULP: a programmable accelerator for MCUs

Chapter 2 introduced an architectural paradigm to augment tightly-coupled clusters of processors with heterogeneity, and provided a methodology for the exploration of the resulting expanded design space. Let us now take a step back and consider whether a tightly-coupled cluster can itself be considered as a programmable accelerator for another system, constituting a "higher level" heterogeneous system with respect to the one explored in Chapter 2. PULP, the platform that we will introduce, is internally homogeneous but is meant to be used in such a scenario, i.e. as an accelerator for microcontroller units. We will focus our attention in particular on applications with very tight power and performance constraints such as embedded vision in smart nodes for Wireless Sensor Networks (WSNs) and the Internet-of-Things (IoT), and insect-size unmanned aerial vehicles (UAVs), that provide a challenging scenario especially for an internally homogeneous cluster that must rely on software efficiency.

3.1 Overview

With the introduction of cheap and powerful embedded computing devices such as Qualcomm Snapdragon 810 [111] and Nvidia Tegra K1 [112], the computer vision field has started to shift from theory and PC-based prototypes towards embedded applications such as smart cameras, self-driving cars and semi-autonomous robots. However, all current vision devices depend on the availability of a relatively abundant source of energy such as a mobile phone battery, which prevents integration of significant vision

capabilities in devices that must run on very limited power and energy budgets, such as micro- or nano-UAVs that have a limited payload to host a battery or wireless sensor nodes that run on harvested power or must live years on a single charge [113]. These devices typically employ low power and ultra-low power microcontroller units (MCUs) that cannot cope with the heavy workloads of CV algorithms, even for very small images.

The ideal computing platform for this kind of heavily energy-constrained applications would be a low power, yet flexible fabric that is able to provide significant performance when needed and remain in a very low-consumption state when not. In particular, smart cameras, micro-UAVs and other similarly constrained applications that are designed to work with input from low-power imagers and performing vision-related algorithms need an exceptional degree of performance and energy scalability to cope both with the limited energy budget and with the frame-rate requirements of vision applications. At the same time, a computing fabric answering to these needs should also provide a very high level of programmability with an easy-to-use model, to keep on track with the fast-moving CV field.

In this Chapter we introduce the *PULP* platform, i.e. Parallel processing Ultra-Low Power platform, a project trying to answer some of these needs. The PULP platform project is a collaborative effort of several academic and industrial institutions[1], whose goal is to design an ultra-low power achieving high levels of energy efficiency by combining near-threshold computing and parallel computing and by exposing low-power features of the technology up the technological stack, at the architecture and software levels. For the purposes of this chapter, we will concentrate on the version of the architecture implemented on the PULPv2 chip, exploiting the capabilities of STMicroelectronics Ultra-Thin Body and BOX Fully Depleted Silicon-on-Insulator (UTBB FD-SOI) technology [114][115] that, in contrast with deep submicron bulk technologies, allows to exploit an extended body bias range to modulate the performance/energy trade-off at different operating points.

To achieve high performance when needed, PULP features a cluster of simple, yet complete, OpenRISC [116] cores that can be used to exploit both coarse- and fine-grain data level parallelism or task level parallelism. At the same time, operating points (voltage, frequency, body biasing) can be controlled at a fine granularity and

[1]Including the University of Bologna, ETH Zurich, STMicroelectronics, EPFL Lausanne, Politecnico di Milano and others.

high speed to achieve high energy efficiency when the performance constraints are more relaxed or when the power budget is tighter. Current PULP chips are not intended as a standalone computing platform, but rather as a component of a low-power heterogeneous system that works in unison with a microcontroller and a set of sensors, targeting mainly the internet-of-things[2]. In particular, it is meant as a general-purpose computing device that is designed to deliver high performance/watt for parallel workloads.

We put the PULP platform to test using several vision benchmarks, which were implemented in pure C code using the OpenMP programming model to express parallelism. Two benchmarks are targeted at the smart surveillance use case. The first is absolute difference motion estimation, a well known highly parallel algorithm that can be used to detect intruders in a camera stream, and is also a component of successful video compression algorithms[117]. The second benchmark is based on Convolutional Neural Networks (*CNN*s or *ConvNet*s) [79], a model that is state-of-art in many current CV benchmarks and has shown promising accuracy results in new classification, detection, and full-scene understanding tasks. CNN-based algorithms are typically computationally demanding and require a good level of performance to work at acceptable frame rates. Finally, to demonstrate the micro-UAV use case for a device such as PULP, we provide a benchmark based on Lucas-Kanade optical flow [118] that can be used as input for self-stabilization and hovering in an aerial vehicle.

The first application we chose to evaluate PULP is smart visual surveillance, with the motion estimation and CNN benchmarks. Motion estimation is a well-known algorithm that is part of video standards such as MPEG [117], with known hardware (e.g. Hsieh et al. [119]) and software (e.g. Brockmeyer[120]) implementations. Conversely, Convolutional Neural Networks (CNNs or ConvNets), originally proposed by LeCun et al. [79], have been object of many recent developments that were rekindled by the discovery of efficient ways to train them [76]. ConvNets have been used to obtain state-of-art accuracy results on scene labeling, video classification and object detection and interest in their applications has been shown by companies such as Google [85][84], Microsoft [86] and Facebook [121].

Future applications for scalable ultra-low power and energy computing devices are beginning to emerge in many fields, such as that of micro-UAVs and of smart and

[2]For an elaborate discussion on the reasons for this choice, see Section 3.4.

ubiquitous surveillance. State-of-art work on autonomous UAVs focuses on relatively big UAVs that are driven by full desktop-class processors and GPUs [122] [123]; to achieve full autonomy in micro-UAVs with much more limited batteries and payload a breakthrough in computing efficiency is needed. Wood et al. [124] quantify the total power budget for this kind of vehicle as 100 mW, of which only 5 mW can be dedicated to sensing and computation. In a similar fashion, smart wireless cameras acting as WSN node need to perform relatively complex activities in a reduced amount of time, while keeping the energy consumption at a minimum [125].

3.2 Architecture

3.2.1 PULP SoC overview

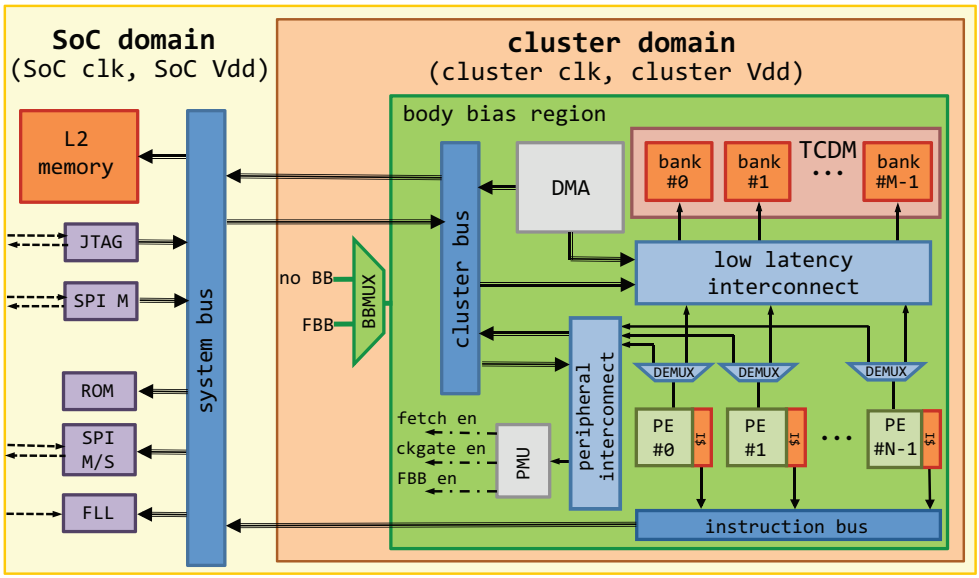


Figure 3.1: PULP architecture.

PULP (Parallel processing Ultra Low Power platform) is a scalable, clustered multi-core computing platform able to operate on a large range of operating voltages, achieving in this way a high level of energy efficiency over a wide range of application workloads. Figure 3.1 shows the main building blocks of a single-cluster PULP SoC. The SoC features an L2 memory (sized in the 32 kB to 256 kB range) accessible through a system bus, plus I/O peripherals that provide flexibility to the whole platform.

The set of peripherals integrated in the PULP platform includes two SPI (Serial Peripheral Interface) interfaces (one master and one slave), GPIOs, a bootup ROM and a JTAG interface suitable for testing purposes. Both SPI interfaces can be configured in *single* mode or *quad* mode depending on the required bandwidth. This provides the necessary flexibility to be able to interface PULP with a large set of off-chip components, such as non-volatile memories, voltage regulators and digital cameras. Moreover, the SPI slave can be configured as a master, and a set of enable signals placed on both SPI interfaces allow the SoC to interface to up to 4 slave peripherals. To connect to more complex devices such as sensors providing on an analog interface, as well as to more complex communication devices (e.g. wireless radios), PULP relies on the host that is also responsible for overall control. Section includes a thorough discussion of this computation model and the reasons behind it.

Normally, PULP behaves as a multi-core accelerator of a standard *host* processor (e.g. an ARM Cortex M low-power microcontroller). In this configuration the host microcontroller is responsible for loading the application and processing data on the PULP L2 through the SPI slave interface on PULP, and initiate and synchronize the computation through dedicated memory mapped signals (e.g. fetch enable) and GPIOs. It is also possible to configure the PULP SoC to work in a "stand-alone" mode where it detects the presence of a flash memory on its SPI master interface and boots from it.

3.2.2 Cluster architecture

The cluster architecture features a parametric number of Processing Elements (*PE*s) consisting of a highly power optimized microarchitecture based on OpenRISC 32-bit ISA [116], each one with a private instruction cache (*I$*). The refill ports of all instruction caches converge on a common cluster instruction initiator port through a cluster instruction bus. The OpenRISC cores were optimized to achieve a high IPC on a wide variety of benchmarks, including control-intensive code [126]. Energy efficiency is boosted by using a simple in-order pipeline to reduce register and clocking overhead, while the datapath was area-optimized to reduce leakage. Further, extensive architectural clock gating was employed to reduce spurious dynamic power.

The PEs do not have private data caches, avoiding memory coherency overhead and increasing area efficiency, while they all share an L1 multi-banked tightly coupled

data memory (*TCDM*) acting as a shared data scratchpad memory. The TCDM has a number of ports equal to the number of memory banks providing concurrent access to different memory locations. Intra-cluster communication is based on a high bandwidth *low-latency interconnect*, implementing a word-level interleaving scheme to reduce access contention [6].

A lightweight multi-channel DMA enables fast and flexible communication with the L2 memory and external peripherals [127]. The DMA uses minimal request buffering and features a direct connection to the TCDM, to eliminate the need for internal buffering, which is very expensive in terms of power. A peripheral interconnect provides access to all the cluster peripherals and to all the resources external to the cluster.

3.2.3 Power management

In order to provide the best energy efficiency across a wide range of workloads, the cluster can work at its own voltage and frequency. To enable fine grained tuning of the SoC frequency, an FLL (Frequency-Locked Loop [128]) is included as a peripheral at SoC level. Moreover, a set of clock dividers (one for the SoC + one for each cluster) allow to further divide the clock generated by the FLL. To reduce the dynamic power consumption in idle mode, each processor can be separately disabled and clock-gated through a set of registers mapped on the peripheral interconnect. In this way, depending on the required workload, each cluster is able to work with an arbitrary number of processing elements, while the others consume zero dynamic power.

The STMicroelectronics 28nm FD-SOI technology enables control of the voltage threshold with body biasing; by using forward body-biasing (FBB) it is possible to lower the threshold improving speed by paying a cost in terms of leakage current, while on the other hand reverse body biasing (RBB) allows to increase the threshold and lower the leakage, with reduced performance[3]. To dynamically select the back-bias voltage of the cluster, a body bias multiplexer (*BBMUX*) is included, enabling ultra-fast transitions between the normal operating mode and the boost mode when temporary peaks of computation are required by the applications. To reduce the latency of the transitions between different operating modes, and making them

[3]Depending on the flavor of the technology (*regular well* or *flip well*) the amount of body biasing that can be performed varies. In particular, flip well LVT transistors, used in the version of PULP described in this chapter, allow for more FBB than RBB

transparent to the software, a power management unit (PMU) was added to generate the control signals of the processors fetch enables, clock gating units, and BBMUX.

3.3 Benchmarking PULP

This section examines the implementation results of the PULP platform on the reference configuration described in Section 3.2, providing the area of the platform, the estimation of the energy efficiency at the different operating points, and a comparison with other state of the art multi-core platforms for embedded computing. It also reports the results of our evaluation of performance and energy in the motion estimation, ConvNet and optical flow benchmarks.

3.3.1 Implementation results

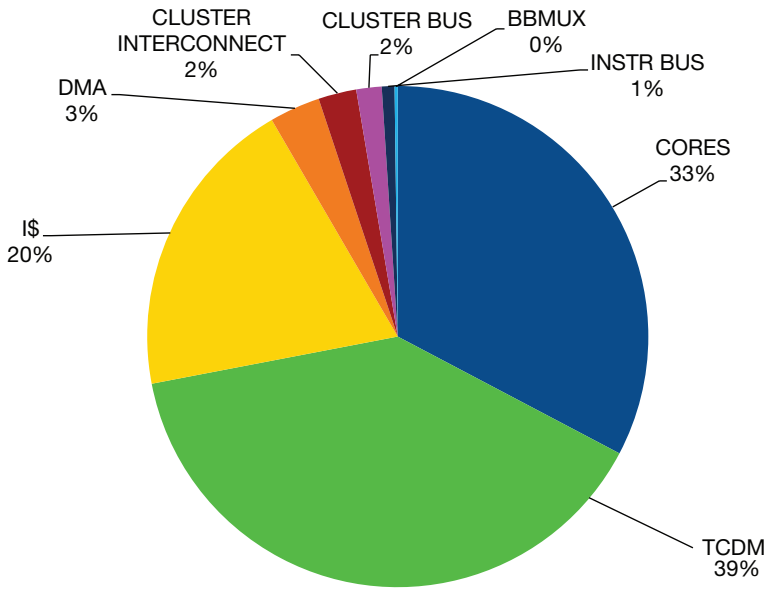


Figure 3.2: PULP cluster area breakdown.

In the context of this chapter we consider a single cluster PULP implementation operating in stand-alone mode. Thus, we assume the SoC connected to an external flash memory which contains the application code, a video surveillance camera periodically feeding the L2 of the SoC with a new frame, and a programmable DC/DC converter configured by the cores to switch between the idle, *search* and *follow* mode

described in Section 3.3.4. The L2 memory consists of 128 kB of SRAM to fit both the program code and one 320×240 frame, that are offloaded via SPI from the host.. The cluster consists of 8 cores featuring 1 kB of I$ each, while the TCDM is composed of 16 banks of 2 kB each, leading to an overall TCDM size of 32 kB. These architectural parameters were chosen to fit the constraints of the benchmarks described in Section 3.3, and should be sufficiently flexible for a broad variety of vision tasks. Both the TCDM banks and the processor's I$ are implemented using standard cell memory (SCM) cuts of 4 kbits each. While SRAMs may achieve a higher density than SCMs (by a factor of $\sim 3\times$), SCMs are able to work at the same voltage ranges as the rest of the logic, with the key benefit of providing much smaller energy/access ($\sim 4\times$)[129].

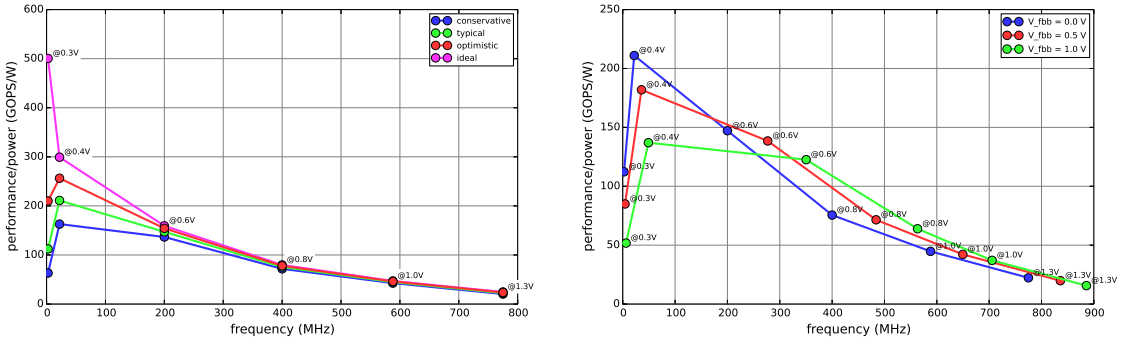
Our results refer to a post place & route implementation of the proposed SoC in STMicroelectronics 28nm UTBB FD-SOI technology, using low voltage threshold (LVT) transistors. Thus, they include the overheads (i.e. timing, area, power) caused by the clock tree implementation, accurate parasitic models extraction, cell sizing for setup fixing and delay buffers for hold fixing (neglecting these would cause significant underestimations in the clock tree dynamic power). The SoC was synthesized with Synopsys Design Compiler, the place & route was performed using Cadence SoC Encounter, and the signoff was performed using Synopsys StarRC for parasitic extraction and Synopsys PrimeTime for timing and power analysis with backannotated switching activities.

V_{DD} [V]	f_{max}[MHz] $V_{FBB} = 0$V	f_{max}[MHz] $V_{BB} = 0.5$V	f_{max}[MHz] $V_{BB} = 1$V
0.3	**2.5**	4.45	6.31
0.4	**22**	35.9	**49.1**
0.6	**200**	**277**	**350**
0.8	400	**484**	**563**
1.0	588	**650**	**705**
1.3	**775**	**836**	885

Table 3.1: Supply voltage and peak frequencies for the reference PULP cluster. Bold values indicate reference operating points.

We tested our platform with power supplies ranging from 0.3V to 1.3V and forward body biasing ranging from 0 to 1V in the typical corner case at the temperature of 25°C[4]. Table 3.1 shows the peak frequency that the PULP cluster can reach at

[4]In FBB n-wells and p-wells are biased by a similar amount, lowering the threshold voltage of both NMOS and PMOS transistors.



(a) GOPS/W while scaling the leakage contribution to power.

(b) GOPS/W with 0V, 0.5V and 1.0V FBB.

Figure 3.3: PULP energy efficiency in GOPS/W.

each operating point, considering the power spent in the cluster. Being the cluster composed of 8 cores, the theoretical performance of the platform can scale between 20 MOPS @0.3V, no FBB to 7 GOPS @ 1.3V, 1.0V FBB. An additional amount of scalability is given by the possibility to enable/disable cores at a fine grain to lower the dynamic power consumption.

Figure 3.2 shows the area breakdown of the cluster, where the overall cluster area in the considered configuration is 1.2 mm^2. It is possible to note that the TCDM and the cores I$ occupy ∼59% of the overall cluster area, mainly due to the SCM based implementation. However, this is fully compensated by the improvement in terms of dynamic power consumption of the memories, which are responsible for the ∼15% of the overall cluster dynamic power, with an improvement of ∼ 4× with respect to a previous implementation of the same architecture [126].

3.3.2 Energy efficiency analysis

This section provides an evaluation of the energy efficiency of the proposed PULP implementation at the different operating points that can be exploited on the platform, focusing on the cluster power domain. To cope with the leakage power variation in the 28nm UTBB FD-SOI, cell libraries are characterized very conservatively; early silicon measurements on PULP prototypes showed that there is more than a 2× guardband on power models. For this reason, we first evaluated the energy efficiency of the platform in four scenarios, accounting for various levels of pessimism for leakage: *conservative*,

where the leakage power is directly extracted from the standard cell libraries; *typical*, with leakage scaled down by 2×; *optimistic*, where it is scaled down by 5×; and *ideal* with no leakage. This experiment allowed us to quantify the impact of the leakage power model guardband over our energy efficiency estimation.

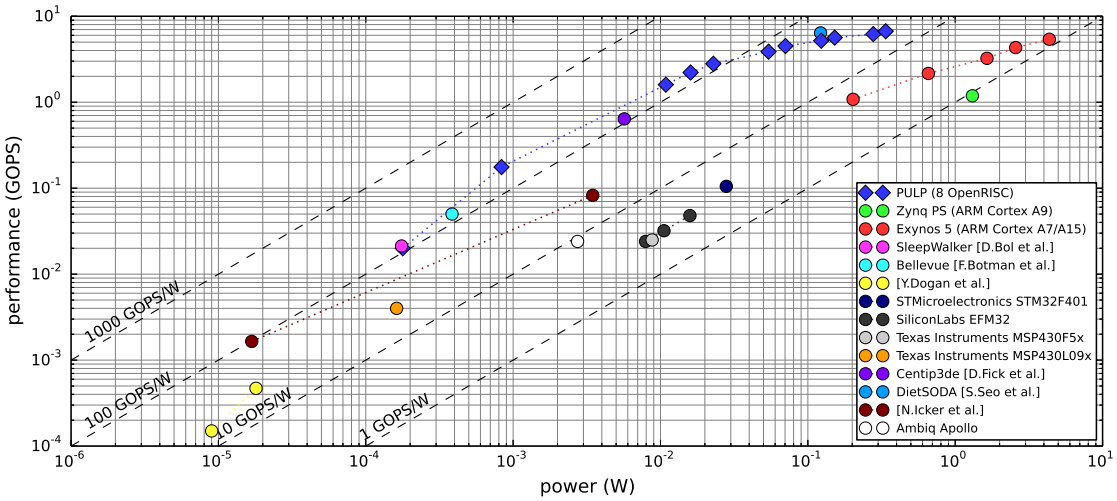
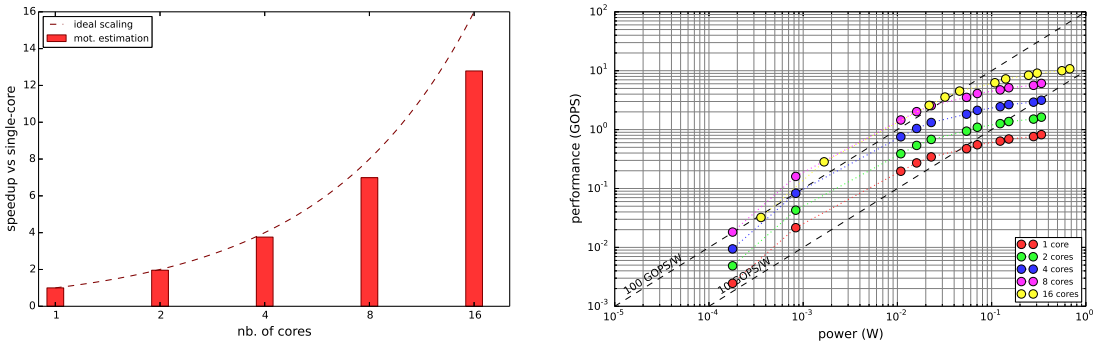


Figure 3.4: Energy efficiency comparison with several platforms.

Figure 3.3a shows the results of this exploration; the platform is working at the maximum operating frequency achievable at each given supply voltage. The peak energy efficiency points in the four scenarios are 172 GOPS/W, 211 GOPS/W, 262 GOPS/W, and 500 GOPS/W respectively. The best energy efficiency point is around 0.4V in all the scenarios except for the ideal. In all but the ideal scenario, the impact of leakage power is huge in the 0.3V to 0.4V operating range, when the supply voltage V_{DD} is close to V_{th} (0.28V for this technology), due to the relatively slow operating frequency (2.5 MHz to 50 MHz) that causes the static contribution of leakage to be dominant. On the other hand, when working with V_{DD} larger than 0.6V, the combined effect of increased dynamic power density (which scales as V_{DD}^2), and higher operating frequency causes the impact of leakage to be smaller. Here we only consider the typical scenario with a twofold leakage reduction as our reference for further power estimations and comparisons; measurements on a previous batch of fabricated PULP prototypes suggest that this is the most realistic value.

Figure 3.3b shows what happens when forward body biasing (FBB) is introduced. By applying FBB, it is possible to dynamically modulate the V_{th} of transistors to

improve the frequency without changing the supply, with only a slight increase of dynamic power in the high-V_{DD} range. On the other hand, FBB introduces an overhead in leakage power, quantifiable as a 7× increase when V_{BB} is 1V [114][115]. For these reasons, FBB is an effective knob to increase the energy efficiency by up to 1.5× for workloads larger than 1.6 GOPS (200 MHz). For example, the target workload of 3.2 GOPS (400 MHz) can be achieved @0.8V with 0V FBB or @0.6V with 1V FBB, resulting in a 1.5× improvement in energy efficiency. These results also justify the need for adaptive body biasing, by showing that the cost of FBB in terms of leakage power is significant: at the 0.6V operating point, it is possible to boost performance by up to 2× using body biasing, but leakage also gets roughly 2× worse. As a consequence, FBB is more effectively used by switching between a *boost* mode and normal active mode than by being used constantly, as this second option would enormously increase the idle power consumption.



(a) Parallel speedup over sequential execution (1 tile).

(b) Energy efficiency (1 tile).

Figure 3.5: Motion estimation benchmark results.

To further provide insight into the scaling capabilities of the PULP platform, in Figure 3.4 we investigate energy efficiency in terms of peak GOPS per watt. We compare the reference PULP platform with several other commercial and academic platforms: the Processing System of the Xilinx Zynq platform (i.e. a dual core ARM Cortex A9), a Samsung Exynos 5 (i.e. a ARM big.LITTLE quad-core A7 + quad-core A15), and many of the ULP platforms referenced in Section 1.3. PULP, providing up to 211 GOPS/W, is competitive with microcontrollers specialized for low-power (Bellevue, SleepWalker) and more performant parallel ULP platforms (Centip3de,

DietSoda), and is much more efficient than mobile solutions such as the Exynos 5 due to the simpler, optimized architecture of the OpenRISC cores and to the fine-grain knobs for power management provided by the FD-SOI technology and enabled by the PULP architecture. It must also be noted that both Centip3de and DietSoda do not support a programming model, whereas PULP has been designed for compatibility with standards such as OpenCL and OpenMP, to ease the exploitation of potential performance in applications.

3.3.3 Motion estimation benchmark

As a first test for the PULP cluster, we wrote an absolute difference motion estimation [117] benchmark composed of several simple kernels: background subtraction, absolute value, binarization, erosion, dilation and a Sobel filter. The aim of the proposed algorithm is to detect the presence of external objects on a video transmitted by a camera framing a fixed background. For each video frame the first stage performs the absolute difference between the current and the background image. The resulting maximum value is extracted and used to calculate the threshold for binarization. The binarized image is then processed by three spatial operators. Erosion and dilatation implement the opening kernel which denoises the binarized image, while edge detection is implemented through a bidimensional Sobel convolution filter to create the external object boundary. If an external object is detected, the final kernel returns the highlighting of that object on the original frame.

The motion estimation benchmark runs on an input 8-bit grayscale 176×120 QCIF image produced by a low-power camera and loaded on the PULP L2 memory along with a prerecorded background. The program code occupies 12508 bytes in the L2 memory. Since the full image cannot fit in the TCDM, we divided the input image in slices or *tiles* of 44×20 pixels that are loaded into the TCDM and processed separately. Each tile occupies 1320 bytes in the TCDM, with a total TCDM occupation of 10560 bytes (four buffers used for present tile output computation, plus four used for double buffering).

Figure 3.5a shows the speedup of parallel versus sequential execution. This kernel is relatively simple and linear and completely parallelizable; as a consequence, its performance scales nicely up to 16 cores. The slight gap between the theoretical and simulated performance is mainly caused by the calculation of the maximum pixel value

after the binarization stage, that cannot be completely parallelized over the available cores. Even so, as that is the only sequential part of the benchmark, speedup and energy efficiency are almost ideal. Figure 3.5b shows performance (in terms of GOPS) against power to evaluate the energy efficiency of the motion estimation benchmark. The plot shows that efficiency peaks at 192 GOPS/W at the 0.4V operating point, reaching 90% of the theoretical limit. One interesting observation is that in active mode the energetic cost of the cluster infrastructure (e.g. memories, interconnections, DMA, etc.) is dominant with respect to the cost of the simple cores that are used in the cluster. The cost of the cores begins to dominate between 4 and 8 cores, which indicates that the "sweet spot" for tightly-coupled clusters with this kind of cores is in this point, and justifies our choice of an 8-core cluster.

3.3.4 ConvNet benchmarks

Convolution-accumulation optimization

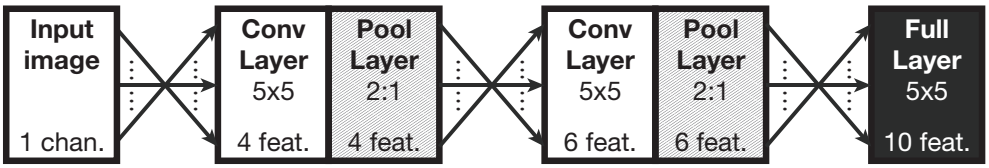


Figure 3.6: Reference convolutional network.

A CNN is composed by a deep sequence of convolutional or fully-connected linear layers intermixed with pooling ones to perform a transformation on *feature maps* produced by the previous layer. Weights in convolutional and linear layers are trained by backpropagation but are used thereafter in a strictly feedforward fashion; due to their data parallel nature they are a natural candidate for acceleration in a parallel platform such as PULP. Convolutional layers in CNNs compute output feature maps of a layer as sums of convolutions over input feature maps; therefore, we chose to use a *convolution-accumulation* step as our basic kernel: $y(i,j) := y(i,j) + \Big(W * x\Big)(i,j)$. where x is the input image, W is the convolution kernel and y is the output image.

We used 16-bit fixed point numbers for inputs, kernels and outputs. We implemented three versions of convolution-accumulation: `naive` directly implements it as four nested loops (two on the output pixels and two for the convolution kernel W); `1-unrolled` uses manual loop unrolling on the innermost loop; `2-unrolled` uses loop

Implementation	3×3	5×5	7×7	9×9	11×11
`naive`, single thread	0.26	0.32	0.34	0.35	0.36
`1-unrolled`, single thread	0.52	0.62	0.65	**0.69**	**0.88**
`2-unrolled`, single thread	**0.80**	**0.83**	**0.76**	0.26	0.18
`naive`, 8 threads	0.26	0.31	0.34	0.35	0.36
`1-unrolled`, 8 threads	0.49	0.60	0.65	**0.69**	**0.85**
`2-unrolled`, 8 threads	**0.71**	**0.77**	**0.74**	0.27	0.18

Table 3.2: Convolution-Accumulation: average efficiency/core.

unrolling on the two innermost loops. We benchmarked these convolutions with a single thread or 8 parallel threads[5].

Table 3.2 shows the efficiency/core for the various convolution-accumulation implementations on a 32×32 input image, computed as the ratio between useful (i.e. computation) cycles and the total number of cycles spent in the outermost loop. For smaller convolution kernels, unrolling both inner loops provides a much better efficiency; however, for kernels bigger than 7×7, efficiency is reduced by I$ misses due to the size of the unrolled loop. As a consequence, the tighter `1-unrolled` convolution-accumulation step is more convenient for bigger kernels. Results are similar in the multi-threaded case, as data contention on the TCDM causes on average only a small amount of efficiency decrease.

Use case: CNN for visual surveillance

On top of these optimized convolutions, we developed a network based on the one proposed by LeCun et al. [79] for MNIST classification, which is shown in Figure 3.6. This network has 2220 parameters and a footprint of 11408 bytes for data and 4400 bytes for weights on the L1 TCDM; Table 3.3a summarizes them. The program code uses 16768 bytes on the L2 memory. As shown in Conti et al. [110], a network of this kind can be trained for complex object detection tasks by running it on a window sliding over the input frame.

We use this CNN for visual surveillance. The platform spends most of the time in a low-power *search* mode looking for suspicious objects (as this task requires only a relatively low frame rate), and it switches to a high-performance *follow* mode to keep

[5]We used the `or1k-elf-gcc` compiler (build `4.9.0 20140308`), with the following flags: `-O2 -nostdlib -mhard-mul -msoft-div`.

layer	params			memory (*bytes*)	
	#feat	*W*	*feat*	*weights*	*data*
input	1	-	32×32	0	2048
conv 0	4	5×5	28×28	200	6272
pool 1	4	-	14×14	0	1568
conv 2	6	5×5	10×10	1200	1200
pool 3	6	-	5×5	0	300
full 4	10	5×5	1×1	3000	20

(a) *small* CNN.

layer	params			memory (*bytes*)	
	#feat	*W*	*feat*	*weights*	*data*
input	1	-	32×32	0	2048
conv 0	8	5×5	28×28	400	12544
pool 1	8	-	14×14	0	3136
conv 2	12	5×5	10×10	4800	2400
pool 3	12	-	5×5	0	600
full 4	10	5×5	1×1	6000	20

(b) *medium* CNN.

layer	params			memory (*bytes*)	
	#feat	*W*	*feat*	*weights*	*data*
input	1	-	32×32	0	2048
conv 0	16	5×5	28×28	800	25088
pool 1	16	-	14×14	0	6272
conv 2	24	5×5	10×10	19200	4800
pool 3	24	-	5×5	0	1200
full 4	10	5×5	1×1	12000	20

(c) *big* CNN.

layer	params			memory (*bytes*)	
	#feat	*W*	*feat*	*weights*	*data*
input	1	-	64×64	0	16384
conv 0	4	5×5	60×60	200	28800
pool 1	4	-	30×30	0	7200
conv 2	6	5×5	26×26	1200	8112
pool 3	6	-	13×13	0	2028
conv 4	10	5×5	9×9	3000	920

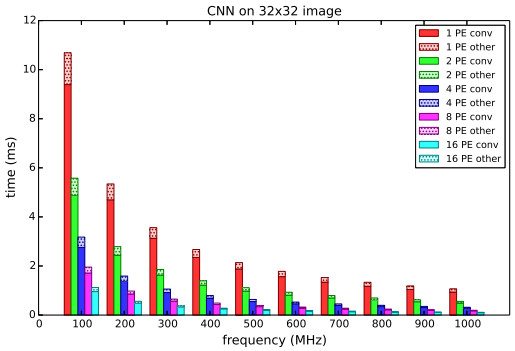
(d) *small* CNN on a 64×64 image.

Table 3.3: Parameters and memory usage of all CNN networks.

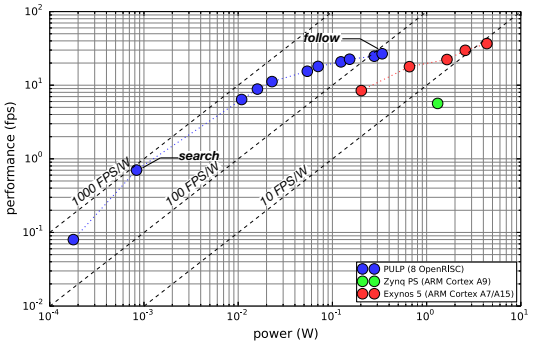
track of a previously detected object. Input frames are brought inside the PULP cluster by DMA transfer from the L2. This transfer is superimposed to the computation of deeper layers and has no impact on the final throughput.

Figure 3.7a shows the performance of the reference CNN when run on a 32×32 image patch, scaling the clock frequency of the cluster from 100 MHz to 1 GHz and the number of OpenRISC cores in the PULP cluster between 1, 2, 4, 8 or 16. As expected from a highly data-parallel algorithm such as ConvNets, execution time scales almost linearly with the number of cores. In our visual surveillance application, the ConvNet is run on a 32×32 window spanning a QVGA (320×240) image with a stride of 32 pixels. Each frame is spanned two times: one with no offset, the other with an offset of 16 pixels in both directions so that the chance of missed detections on the border of a window are reduced. PULP can be set to work at a very low frame rate (∼0.7 fps at the 0.4V operating point) in the *search* mode, and then switched to a frame rate as high as 27 fps (at the 1.3V operating point with 1V FBB) in the *follow* mode.

Figure 3.7b shows the energy efficiency of the ConvNet execution on a frame in terms of FPS/W; we ran the same ConvNet on the Xilinx Zynq PS and on a Samsung Exynos 5 for comparison, as this benchmark is beyond the typical performance capabilities of most ULP microcontroller architectures. Benchmark results substantially confirm



(a) Execution time in ms (32×32 patch).

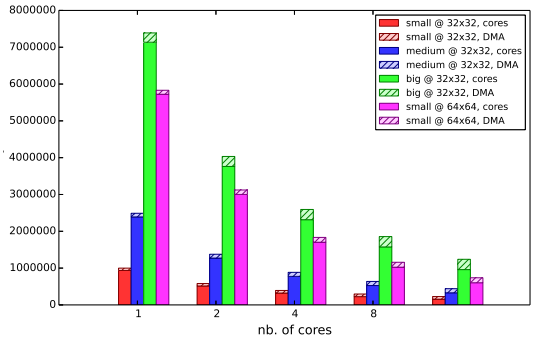


(b) Energy efficiency in FPS/W (QVGA frame).

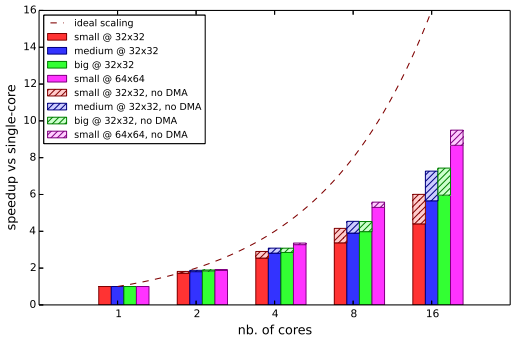
Figure 3.7: Surveillance ConvNet benchmark results.

the theoretical values shown in Figure 3.4. The energy/execution time tradeoff when switching between *search* and *follow* mode is also clearly shown: in *search* mode, PULP consumes 1.18 mJ per frame and lives at a power budget of 834 µW, whereas in *follow* mode energy consumption jumps at 12.6 mJ per frame.

Tiled CNNs



(a) Execution time in cycles.



(b) Parallel execution speedup.

Figure 3.8: Tiled CNN benchmark performance results.

To further explore the capabilities of the PULP platform in this scenario, we considered the case that the CNN or its input image cannot fit in the TCDM. In this case, it is necessary to tile the CNN similarly to what is described in Section 3.3.3; also in this case, double buffering can be employed to hide the latency of the L2/L1

memory transfer.

In the case of CNNs, tiling involves some amount of recomputation as the receptive field of each output convolutional tile is partially superimposed to that of the next output tile. We can tile the same ConvNet with two distinct approaches. With a "vertical" tiling approach, the full network is applied to each input tile until the last layer, then the output is transferred to the L2 memory and a new tile is loaded; "horizontal" tiling instead is applied by dividing input of a single layer in tiles and computing all output tiles before proceeding to the following layer. In this approach, intermediate results (i.e. the outputs of intermediate layers) have to be stored in buffers in the L2 memory.

We chose to concentrate on horizontal tiling for three reasons: first, vertical tiling involves a lot of recomputation as the smaller ConvNet tile has to be moved over the input image (similarly to what we did in Section 3.3.4, but with a stride of 1 pixel instead of 32). Second, the horizontal approach allows us to tile also in the input feature dimension, whereas in vertical tiling all input features are needed in the shared memory to compute the following layer. Third, although horizontal tiling involves frequent data transfers between L1 and L2, we will show in the following that the impact of these transfers scales nicely with the size of the input data and the amount of parallelism.

We extended the reference CNN[6] of Section 3.3.4 in the following way. The *medium* and *big* CNNs, whose parameters are reported in Tables 3.3b and 3.3c respectively, are similar to the *small* one but their intermediate layers have more features. The fourth CNN shares the same parameters as the *small* one, but runs on a bigger image patch of 64×64 pixels; its parameters and memory consumption are reported in Table 3.3d. In this network, the final linear layer is substituted by an equivalent convolutional layer using the same weights; the output is equivalent to the separate classification of all pixels (see for example Sermanet et al. [130]). In all benchmarks, we set the maximum dimension of the tiles to 4KB so that it is possible to fit two input tiles and two output tiles in the TCDM. The dimension of the program code is similar for all of these benchmarks (∼25KB loaded on the L2 memory), since we relied on the same ConvNet library extended with horizontal tiling support.

Figure 3.8a reports the execution time of all benchmarks in terms of cluster clock

[6]We will refer to this network as the *small* network from this point on.

cycles. The computational complexity of the CNN raises exponentially when we double the number of feature maps used in each layer or the pitch of the input image; we observe that the *big* CNN applied on a 32×32 image and the *small* one on a 64×64 image impose similar constraints both in terms of workload and of memory occupation. To better evaluate how performance scales in all benchmarks as we vary the number of cores, Figure 3.8b compares speedup versus single-core execution for all benchmarks. Figure 3.8b also reports the theoretical speedup if we neglected all impact of DMA transfers. The main limiting factor for speedup is given by Amdahl's law: due to the small dimension of the tiles, the parallel fraction of the code is not sufficient to yield quasilinear speedup. This is clearly visible in that the same ConvNet applied to a $4\times$ bigger input image yields much better results in terms of performance scaling. The size of the input image itself is mainly limited by the availability of L2 memory. The plot also shows that, due to the higher computation to communication ratio, the impact of data transfers on the speedup scales nicely with the size of the workload, i.e. the bigger the input image and/or the CNN is, the less limiting impact DMA transfers have over parallel execution speedup.

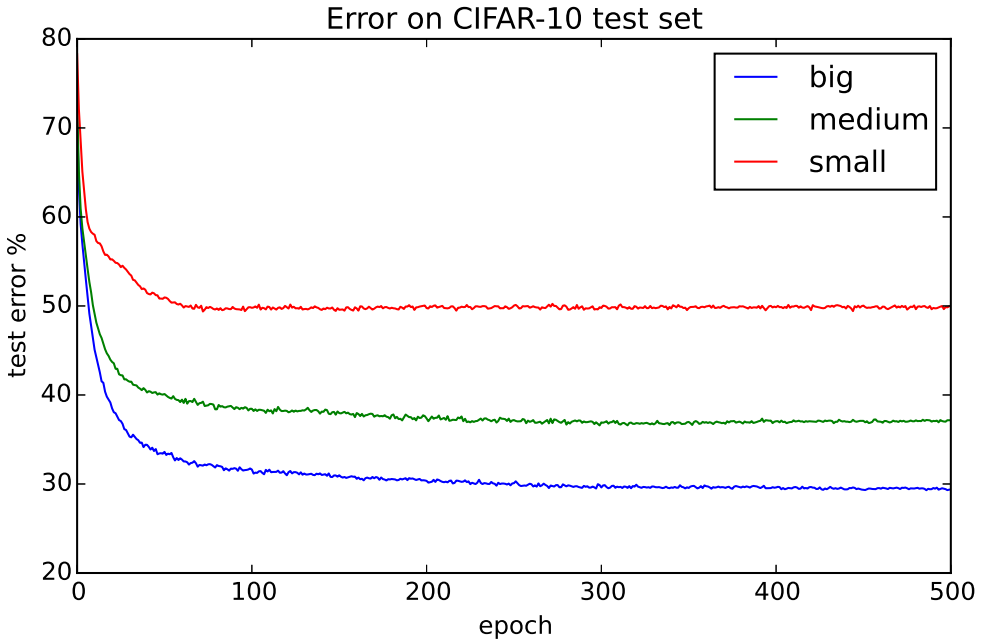


Figure 3.9: Test error of *small*, *medium* and *big* CNNs on the CIFAR-10 set over 500 training epochs.

To estimate how much the accuracy may vary between the *small*, *medium* and *big*

CNNs, we trained them to classify the CIFAR-10 dataset [131], a well known and freely available set of 60000 32×32 images labeled in 10 classes [7]. Figure 3.9 shows that the difference can be significant: after 500 epochs of training the final accuracy is 70.64% for the *big* CNN, which drops to *64.38%* for the *medium* one and to 50.05% for the *small* one. The difference is greatly reduced if we compare the CNNs for a one versus all classification task over the same dataset: the final accuracy in this case is 95.1%, 94.2% and 93.3% for the *big*, *medium* and *small* CNNs respectively.

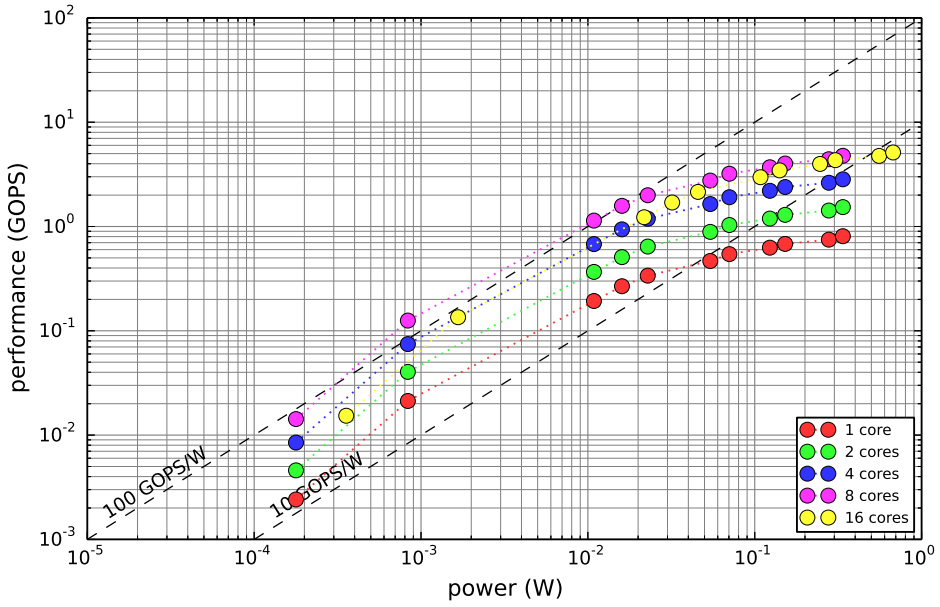
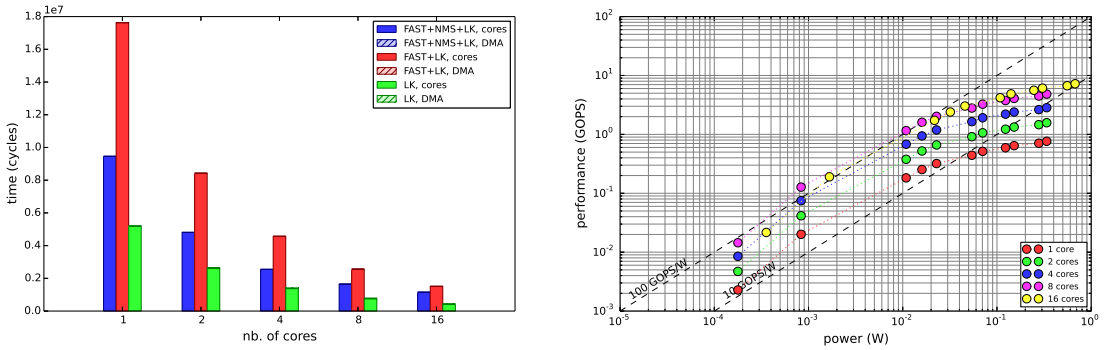


Figure 3.10: Energy efficiency for execution of the *small* CNN on a 64×64 image, while sweeping the number of cores.

In Figure 3.10, we plot the energy efficiency in terms of GOPS/W for the execution of the *big* ConvNet (results are practically identical for the other benchmarks). Compared with the peak theoretical value of 211 GOPS/W, we measured a peak of 150 GOPS/W in this benchmark, which correspond to an average IPC of 0.71 per core. By comparison, average single-core IPC in the inner convolutional loops is 0.96, and average single-core overall IPC is 0.87. The IPC reduction in the multi-core tests is mainly accounted for by contention on the shared TCDM and, to a lesser extent,

[7] As our CNNs work on grayscale images, the training and test samples where converted from RGB to grayscale. Training consisted in 500 epochs of mini-batch stochastic gradient descent with momentum $\mu = 0.9$ and starting learning rate $\lambda_0 = 0.01$ (dropping exponentially as $\lambda = \lambda_0 \cdot 0.995^{n_{\text{epoch}}}$), using 20% dropout [76] layers for better regularization.



(a) Execution time in cycles (all).

(b) Energy efficiency in GOPS/W (*FAST+NMS+LK*).

Figure 3.11: Optical flow benchmark results.

by contention on the I$ refill bus. Still, IPC in the inner-loops is as high as 0.90 per core when executing with 8 cores. The energy efficiency results mimic the peak ones presented in Figure 3.4, and peak efficiency (125 MOPS @ 834 µW) at the same operating point (0.4V without FBB) in the near-threshold region.

3.3.5 Optical flow benchmark

As a representative application for the usage of PULP as an accelerator for an autonomous nano-UAV, we developed an optical flow benchmark that is meant to be integrated in the drone control loop to make completely autonomous hovering and navigation possible. In this scenario, a low-resolution (e.g. 128×128 pixel) ultra-low-power imager such as a CentEye Stonyman [132] continuously feeds frames to PULP via the QSPI slave interface. On turn, PULP computes the optical flow and uses its QSPI master to report the flow vectors back to the microcontroller driving the vehicle, where they are used to estimate rotations and translations of the drone.

The benchmark is composed of three kernels: FAST corner detection [107][108], non-maximal suppression and Lucas-Kanade optical flow estimation [118]. Since Lucas-Kanade should be applied to strong corners to yield high-quality, it is generally not advisable to drop either the non-maximal suppression step or the whole corner detection. Nonetheless, since the users of the flow vectors (i.e. the aerial vehicle software developers) might want to trade off optical flow accuracy for performance and energy, we decided to explore also these non-optimal cases. Therefore, we present results

for three separate implementations: *FAST+NMS+LK* that feeds corners produced by the FAST algorithm in non-maximal suppression before computing optical flow; *FAST+LK* that uses all corners produced by FAST for the optical flow; *LK* that drops corner detection and computes optical flow on all pixels.

The input of the optical flow application are two 128×128 8-bit grayscale frames stored in the L2 memory by the QSPI slave module. To cope with the dimension of the input frames, we divided them in stripes of 128×16 pixels; we use double buffering to transfer the stripes from the L2 to the TCDM while we are computing the optical flow of the previous stripe.

Figure 3.11a reports the execution time in cycles for all versions of the benchmark, sweeping the number of cores in the PULP cluster from 1 to 16. The first observation is that the *FAST+LK* benchmark is the slowest; this is due to the fact that if non-maximal suppression is dropped, the Lucas-Kanade step has to be performed on a much higher number of corners, in the order of several hundreds. The *LK* benchmark drops FAST altogether and is therefore the fastest, even if it computes optical flow on the full 16384 pixels of the image. Conversely, the *FAST+NMS+LK* benchmarks spends most of its time in computing the best corners (in the order of some tens) in the picture and much less time in the actual optical flow, as it is computed only on those corners. In all cases, optical flow computation on the 128×128 input frames takes more than 1 million cycles when performed with 8 cores: intuitively, this means that the workload to perform this task at 60fps is bigger than 60 MOPS.

Figure 3.11b helps to understand whether this is a feasible target, and at what power budget, by plotting energy efficiency in terms of GOPS versus watt measured by profiling the optical flow *FAST+NMS+LK* benchmark. At the most efficient operating point (0.4V with no FBB, 834 µW of power consumption) the 8-core cluster achieves a performance of 127 MOPS, with an efficiency of 152 GOPS/W. The peak efficiency is similar to that of the CNN benchmark but this is due to a different mixture of effects from the result obtained in Section 3.3.4. First, the lower internal regularity of the FAST benchmark (which is responsible for the majority of the execution time) hits the inner-loop IPC with respect to the very regular and manually optimized convolutional kernels employed in the ConvNet. At the same time, however, it also significantly lowers data contention, leading to a similar overall efficiency result. At this operating point, optical flow would be feasible for a micro-UAV application as it would add less than a mW to the total vehicle power, which nicely fits in the 5 mW

budget for computing in Wood et al. [124]. The energy budget to compute a frame is 13.9 µJ; to make this measure concrete let us take for example the commercial Crazyflie Nano Quadcopter [133], that mounts a 240 mAh 3.3V battery, hosting approximately 2850 J of energy destined primarily to power DC motors. If we suppose a flight time of one hour, the battery consumption due to the PULP accelerator would amount to ∼3 J, i.e. 0.1% of the total battery, which is almost negligible with respect to the energy consumed by the vehicle actuators.

3.4 PULP as a programmable accelerator

Heterogeneous acceleration in the design space of low power sensor nodes, nano-UAVs and similar devices answers to a different set of requirements with respect to other domains. Although energy efficiency is extremely important, absolute power consumption is also a first-class citizen; sensor nodes in particular are severely constrained in terms of cost and power delivery, which is usually implemented with small batteries and/or harvesters [21]. As a target model for flexible heterogeneous acceleration within this domain, we consider an accelerator with three main characteristics:

1. *energy efficiency*: to a much larger degree with respect to what happens in larger scale heterogeneous computers, an ULP accelerator needs to be significantly more energy-efficient than its host to be effective - otherwise it will not be useful for acceleration under the low-power constraints typical of these systems;

2. *dynamic offload*: for the accelerator to be programmable (as opposed to simply reconfigurable), it must be possible to dynamically offload different applications from the host; this marks a strong difference to typical fixed functionality ASICs that are often used in a similar domain;

3. *programmability*: development of accelerator code must rely on a programming model that allows efficient exploitation of parallelism, while leveraging a lightweight runtime system with low execution overhead and memory footprint.

PULP satisfies these requirements: it is fully programmable with OpenMP and supports code and data offload via the SPI interfaces, while the previous sections have shown how it is possible to obtain significant performance and energy efficiency on the platform.

Within the work described in this chapter, we chose to treat PULP as an accelerator to a host MCU, avoiding usage of PULP standalone or the integration of both the MCU and PULP within a single-chip heterogeneous system; instead, we proposed to use an accelerator built in 28nm FD-SOI technology and an off-the-shelf MCU fabricated in a more "conservative" technology node as the host. This choice answers to cost considerations specific of the low-power domain. The PULP accelerator absolutely needs to be implemented in a deeply integrated technology to achieve the high energy-efficiency (as shown in Figure 3.4) that can be "spent" to provide a high level of speedup. Conversely, MCUs are typically fabricated with a different set of goals, such as very low mask cost and fast turn-around time due to the need to produce a great number of models differing by interfaces, memory size, etc. In particular, a significant fraction of many MCU chips is occupied by analog and mixed-signal IPs such as analog-to-digital converters that are difficult to scale down to deeply integrated technologies, and therefore are not easy to integrate on the accelerator.

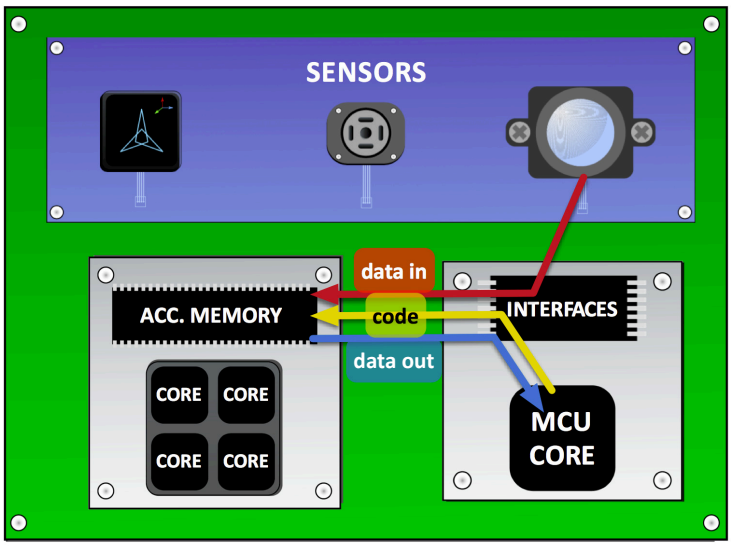


Figure 3.12: Low-power heterogeneous accelerator model, showing a simple offload procedure with code (yellow), input data (light blue) and output data transfers (orange).

Figure 3.12 shows an abstract model of the heterogeneous system we consider. For generality, we consider the case of a sensor connected to the host MCU memory and communication between the MCU and the PULP accelerator via DMA. We consider

this model as it provides a baseline for the offload mechanism; connecting the sensor directly to the accelerator would be optimal from the energy point of view, but it would be less cost effective unless the sensor uses a standard digital interface such as SPI, since it would require the addition of a custom interface to the accelerator.

In fact, to amortize the non-recurrent engineering costs of the higher density technology, accelerators such as PULP must be produced in high volume; therefore, the most sensible approach is to make them able to couple with the highest possible number of microcontrollers on the market. In PULP, this is achieved using an SPI interface, which is available on the overwhelming majority of MCU platforms and allows relatively easy and cheap integration of the host and accelerator in a system-on-board. It also has the significant advantage of being fully retrocompatible with the programming legacy, as it builds on existing MCU programming models without disrupting them.

From the energy perspective, it is of course possible to significantly improve compared to this baseline by using a low-power, high-throughput serial link such as, for example, that presented in Choi et al. [134]. Clearly, another variation on the model proposed in Figure 3.12 is to bring data from the sensor directly to the internal memory of the accelerator. This reduces the pressure on the coupling link in terms of throughput, while it can still be used for pipeline cooperation between accelerator and host. However, it also requires a dedicated (and more expensive) interface between the sensor and the accelerator.

To further understand the implications of the choice of a "slow" serial link as the main communication mechanism between the PULP accelerator and the host MCU, we measured the overhead of the offload procedure over a set of kernels from IoT applications (`matmul`, `strassen`, `cnn`, `hog`, `svm`) [7] [8]. We consider an offload following this procedure: *i*) kernel code is transferred from the host MCU to PULP; *ii*) the first input data frame is transferred from the MCU to PULP; *iii*) while PULP computes over the frame, a new input frame is loaded using a double buffering scheme; *iv*) similarly, double buffering is also used for output data to be transferred from PULP to the MCU.

Figure 3.13 shows the offload efficiency, measured as the ratio between the effec-

[8] In this experiment, the kernel code has an average size of 20.4 kB, the average input/output data payload is of 15.4 kB, and the average computation-to-communication ratio is kept relatively low at 0.31 ops/B.

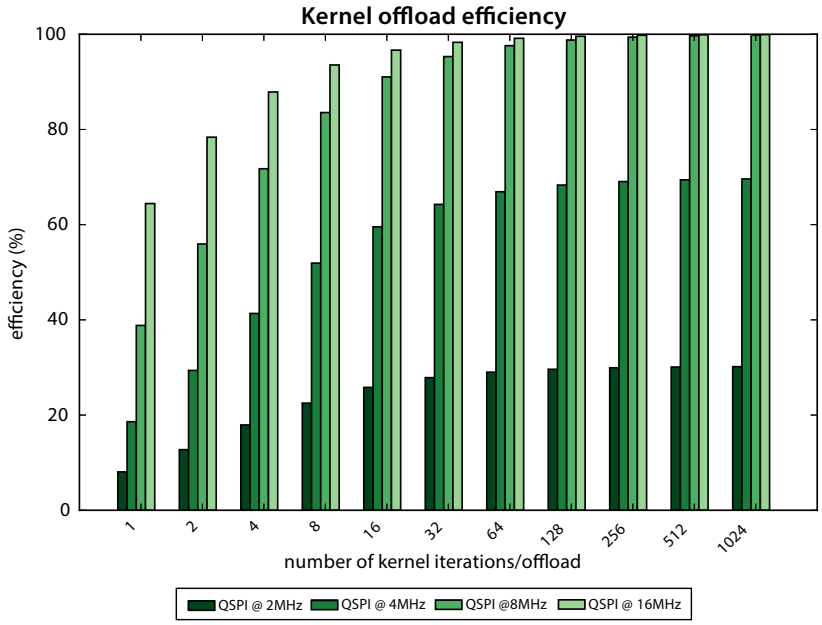


Figure 3.13: Offload efficiency; geometric mean over the `matmul`, `strassen`, `cnn`, `hog`, `svm` kernels described in Conti et al. [7].

tive execution time (including the offload overhead) and the ideal execution time (i.e. that with no offload overhead) while varying the number of kernel iterations per each offload. Transfer happens over quad-SPI at four possible frequencies. As expected, excessively low transfer speed can seriously hurt performance for kernels with very low computation-to-communication ratio; however, with relatively low speed requirements (e.g. 8 MHz), very good efficiency can be reached by using the simple double buffering scheme we propose and reducing the initial overhead of kernel code offload with repeated executions of the same kernel. In the case of a sensor node, repeating the same basic task over varying data is a reasonable assumption, therefore we can conclude that - even in the "worst-case" conditions of not having a dedicated sensor-to-accelerator interface - it is indeed possible to use an accelerator such as a PULP SoC as accelerator.

3.5 Conclusions

In this chapter we introduced the PULP (*Parallel processing Ultra-Low Power*) platform that features a cluster of tightly-coupled OpenRISC cores to achieve high energy

efficiency through parallelism. We have analyzed the platform, showing that its performance can be scaled by the dramatic factor of 354× and that it features a peak energy efficiency of 211 GOPS/W.

As a use case for PULP, we show a motion estimation algorithm for smart surveillance which almost fully exploits the available performance, with a peak energy efficiency of 192 GOPS/W, i.e. 90% of the theoretical peak. We also implemented a ConvNet-based algorithm for video surveillance, showing that it can be switched from a low-power state consuming just 1.18 mJ per frame with a rate of 0.7 fps to a high-performance state running at 27 fps and consuming 12.6 mJ per frame. Finally, we wrote a sample benchmark for applications in the nano-UAV field, where we use PULP to accelerate estimation of optical flow from frames produced by a ULP imager, with the objective of autonomous hovering and navigation; we show that it is possible to meet tight timing constraints (60fps frame rate) at the energy budget of 14 µJ per frame. These benchmarks showcase the high level of flexibility and programmability of the PULP platform, that does not come at the expense of energy efficiency: all of them were able to reach at least 70% of the peak efficiency overall, with much higher peaks in highly parallel regions such as in the motion detection and inner convolutional kernels.

Several PULP chips have been fabricated and tested at the time of this thesis using several technologies, including STMicroelectronics 28nm FD-SOI (both LVT and RVT[67]) and UMC 65nm bulk. A functional PULP chip featuring 4 OpenRISC cores, 64 kB of L2 memory and 24 kB of TCDM has been fabricated in 28nm STMicroelectronics FD-SOI technology in 2015 [135]. With respect to the work described in this Chapter, later revisions of the PULP architecture include a shared instruction cache for improved energy efficiency [136], a redesigned microarchitecture for the core [68], support for a shared floating-point unit [137], hardware accelerators [94] and other improvements.

This chapter has shown a different flavor of heterogeneity than what was shown in Chapter 2: a loose coupling with the host via SPI instead of a tight one using the TCDM; a fully programmable accelerator instead of a a fixed-function one. However, despite these substantial differences architectural heterogeneity is used in both cases to extract more energy efficiency out of the overall platform. This is done by exploiting some of the potential parallelism that is present in the application without paying the full cost of being completely general-purpose, although PULP is of course a much

more flexible kind of accelerator than the HWPEs presented in Chapter 2.

Moreover, as both He-P2012 and PULP are based on the same tightly-coupled cluster architectural paradigm, it is possible to reuse a similar acceleration methodology in the PULP scenario. In fact, Chapter 4 will deal with this idea, introducing a tightly-coupled shared memory accelerator inside the PULP cluster.

Chapter 4

Brain-inspired acceleration in an ultra-low energy budget

As was discussed in Chapter 2, one of the key issues when augmenting a platform with fixed-function accelerators is the actual choice of which tasks should be implemented in hardware; this choice can be made only at design time. Therefore, it would be particularly desirable to introduce a heterogeneous core that, by accelerating a specific fixed function, can be used to implement an entire class of tasks. This kind of functionality can be partially introduced relying on learning based models, which are in general loosely inspired by some of the mechanisms of the mammal neural cortex. Some of these models, such as convolutional neural networks, can be used to give approximate solutions to problems for which an algorithmic solution is known or, equivalently, to solve problems for which an algorithmic solution is not known (e.g. counting the number of people from an image [110]). At the same time, models such as CNNs are extremely relevant in the domains targeted by the PULP platform, e.g. in "smart" sensor nodes in wireless sensor networks, as they can be used to extract relevant semantic information (such as a class) out of low "information density" raw data such as images - which allows to greatly reduce the amount of data to be sent through wireless radio links that are relatively expensive (in terms of energy). Therefore, CNN models are an extremely interesting target for acceleration for both conceptual and pragmatic reasons. In this Chapter, we propose to augment a platform such as PULP with a core that is specialized in acceleration of convolutional neural networks.

4.1 Overview

Computer vision (CV) is a rapidly developing field; algorithms showing excellent results in terms of object detection, full scene parsing, image segmentation have been successfully proposed in the last years. A particularly interesting class of algorithms is that of brain-inspired CV (BICV), which is loosely inspired by the inner working of the mammal brain. This class includes algorithms such as HMAX and Convolutional Neural Networks (*CNN*s or *ConvNet*s [79]) that are state-of-the-art in many accuracy benchmarks [76][83][84][85][86].

Most of these algorithms rely on a large number of time- and energy-hungry 2D convolutions. While some BICV applications are not particularly performance- or power-constrained, and can thus rely on off-the-shelf HW for implementation, many more would greatly benefit from low-energy implementations of state-of-the-art BICV algorithms: wearable computers, wireless sensor networks, visual impairment aids are just some examples. As an example use case, let us consider a Google Glass-like device powered by a 2.1Wh (7560 J) battery that we want to last for 24 hours, while running the GoogLeNet convolutional network, which has achieved top-quality results in the ImageNet Large-Scale Visual Recognition Challenge 2014 and has been designed for running on computation- and memory-constrained hardware [138]. The network has a receptive field of 224×224 RGB pixels and requires a total of 1.5 billion multiply-accumulate (MAC) operations. Scaling this to a standard QVGA camera (320×320 pixels), the performance requirement is of 2.3 GMAC/s. Let us assume that our target is real-time classification at 15 fps. This requires a performance of 34.5 GMAC/s or a total of 2.98×10^{15} MAC operations per day, giving us a budget of 2.5 pJ per MAC operation. Equivalently, as we have an average power budget of 87 mW for computation, the overall efficiency must be of 397 GMAC/s/W or more, i.e. more than 1500 GOPS/W if we consider that each MAC corresponds to at least 4 RISC operations (load pixel, load weight, multiply, add). [1].

Much research has recently focused on many-core architectures using many simple power-optimized RISC cores to build up a high-performance, low-power platform.

[1] This is simply a lower bound, as in general we define the number of RISC operations required by a given function in an operational fashion: we run it on a single OpenRISC core and count the number of instructions executed. Since the OpenRISC core has a very simple 5-stage pipeline and a reduced instruction set (comparable to that of the original MIPS) this gives a relatively accurate measurement of the computational characteristics of the function.

The PULP platform [68][64][66], that was described in Chapter 3, is such an effort and demonstrates a high degree of energy-efficiency without sacrificing the flexibility of software. Relying on pure software can be a suboptimal choice in the context of brain-inspired CV that is mostly based on repetitions of a single basic kernel: convolution. First, convolution is heavily data-driven; the same basic *fetch weight - fetch pixel - multiply - accumulate* loop is repeated many times on changing inputs, and a lot of unnecessary energy is spent in fetching these same instructions over and over. Moreover, convolution is a reduction operation in which input data and weights are reused many times, too; redundant memory operations result in energy waste. Third, it also shows an excellent degree of data-level parallelism that could be used to counteract the former overheads, but is usually left untouched in ILP-oriented processors and cannot be fully exploited by parallel platforms unless they feature a very high core count. Finally, deep convolutional networks share a very reduced set of kernels, but can be used to implement a huge application domain by simply changing weights and network topology.

All these characteristics make a strong case for a mixed solution in which a set of flexible dim cores cooperate with a specialized convolution core that is kept dark most of the time, but can provide much better performance and energy efficiency when needed. To this end, this Chapter presents the *Hardware Convolution Engine* or HWCE, a coprocessor for 2D convolution optimized for enhanced speed and energy efficiency in BICV and other convolution-heavy applications. We integrated the HWCE in PULP [66], the ultra-low power multi-core platform described in 2.

In this Chapter, we show that by using the HWCE it is possible to spend as little as 6.5 pJ/output pixel in optimal cases, 40× better than what achievable by SW and improving more than 6× with respect to the state-of-the-art in convolution acceleration in a multi-core cluster [93]; moreover, we provide results for the implementation of a full Convolutional Neural Network using the HWCE. The main feature distinguishing the HWCE from the state-of-the-art is its integration within the PULP cluster, enabling to access ASIC-grade energy efficiency while still retaining a great amount of flexibility with respect e.g. to CNN topology.

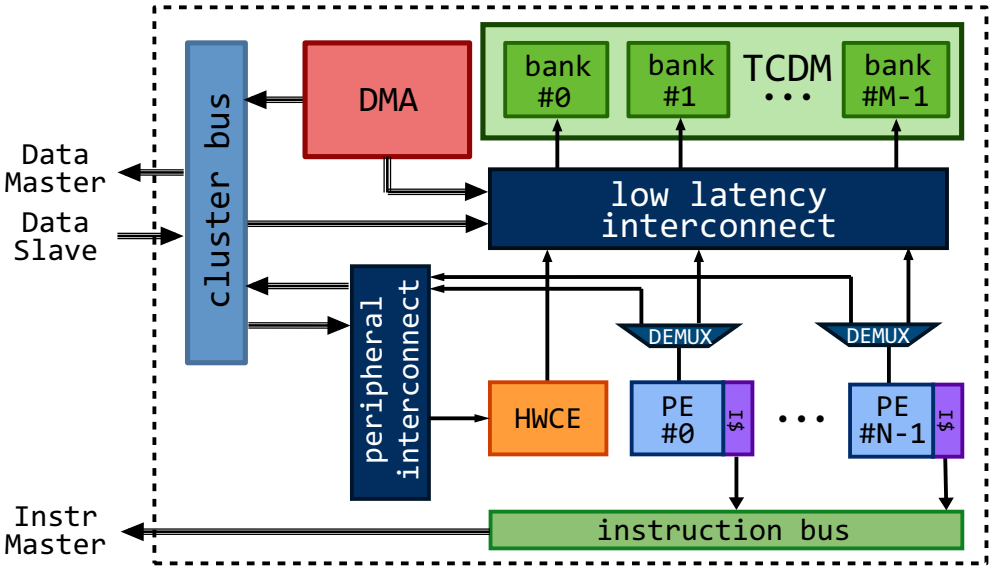


Figure 4.1: HWCE-augmented shared-memory PULP cluster.

4.2 Architecture

4.2.1 Shared-memory cluster

The target shared-memory cluster is that of the PULP [68][66] architecture that was descrived in Chapter 3. Specifically, the architecture we consider features 4 OpenRISC cores sharing 16kB of TCDM acting as a L1 shared scratchpad and having each 1kB of private instruction cache. It also includes a lightweight multi-channel DMA for flexible and fast communication with L2 and external peripherals [127]. Following the tightly-coupled acceleration paradigm that was detailed in Chapter 2, the heterogeneous cluster includes a Hardware Convolution Engine (*HWCE*), whose internal architecture is detailed in Section 4.2.2. The HWCE shares data with the rest of the cluster through three ports on the low-latency interconnect, behaving essentially as an additional special-purpose core. At the same time, control of the HWCE is memory mapped through the peripheral interconnect much like other peripherals in the cluster. The main difference between the HWCE and the HWPEs featured in Chapter 2 is that those are generated semi-automatically from a set of C sources, while the HWCE has been optimized for much better energy efficiency, which is essential for achieving adequate performance within a tight power budget.

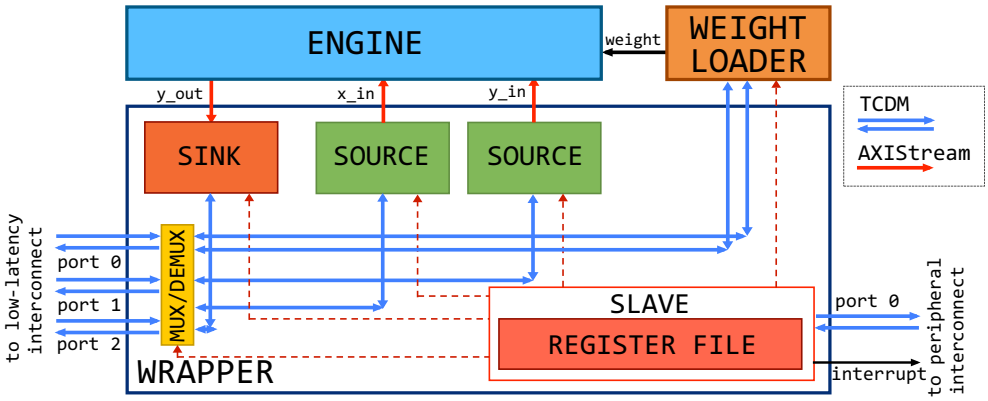


Figure 4.2: Architecture of the HW Convolution Engine.

4.2.2 Hardware Convolution Engine

The Hardware Convolution Engine (*HWCE*) is designed in a modular and parametric fashion, to ease IP reuse and provide sufficient flexibility for a great majority of CV applications relying on convolutional kernels. It is entirely described at the register-transfer level in the synthesizable subset of the SystemVerilog HDL.

The HWCE is algorithmically optimized to minimize memory bandwidth requirements to perform convolutions. It is structured in three main blocks: the proper *engine* (a datapath purely responsible of computation of output pixels), the *wrapper* (which provides data- and control-plane communication with the shared-memory cluster) and a simple *weight loader* (that loads and keeps weights used for convolution).

Convolution strategy

In convolutional neural networks, convolutional layers are well known to be the most expensive in terms of computation [92]. The general structure of the linear part of a convolutional layer is given in Equation 4.1. $K = 2P + 1$ is the size of the filter, while N_{if} and N_{of} are the number of input and output feature maps, respectively. x is the input feature map set, a 3-dimensional tensor of size $(N_{\text{if}}; h, w)$, where h and w are the height and width of the input frame, respectively. y is the output feature map set, a 3-dimensional tensor of size $\big(N_{\text{of}}; (h - K + 1), (w - K + 1)\big)$. W_K is the weight 4-dimensional tensor, of size $(N_{\text{of}}, N_{\text{if}}; K, K)$. b is the 2-dimensional bias of size $(N_{\text{of}}, N_{\text{if}})$. k_{if}, N_{of} are indices spanning the first dimension of the input feature map set and the first dimension of the output feature map set, respectively. i, j, u_i, u_j are

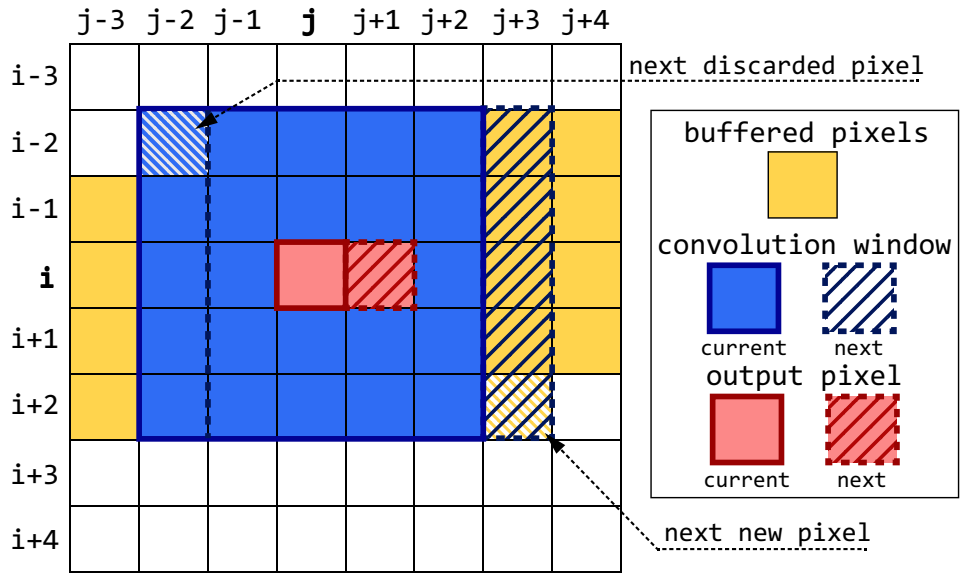


Figure 4.3: Strategy for 2D 5×5 convolution with linear data streams [8].

the row and column indices of the feature maps and of the filter, respectively.

$$y(k_{\text{of}},k_{\text{if}};\,i,j) = \sum_{k_{\text{of}}=0}^{N_{\text{of}}-1} \sum_{k_{\text{if}}=0}^{N_{\text{if}}-1} \left(b(k_{\text{of}},\,k_{\text{if}}) + \sum_{u_i=-P}^{P} \sum_{u_j=-P}^{P} W_K(k_{\text{of}},k_{\text{if}};\,u_i,u_j)\, x(i-u_i,j-u_j) \right). \quad (4.1)$$

The non-linear part of the convolutional layer applies a function such as rectification or hyperbolic tangent to the output y of the linear part. As the linear part is responsible for the overwhelming majority of the computation performed in a CNN, we will focus exclusively on it for the remainder of this Chapter.

The computation of Equation 4.1 can be visualized as the repetition for $N_{\text{if}} \times N_{\text{of}}$ times of a basic *convolution-accumulation* step while changing the W_K filters. Equation 4.2 shows this basic convolution-accumulation step:

$$y(k_{\text{of}};\,i,j) := y(k_{\text{of}};\,i,j) + \sum_{u_i=-P}^{+P} \sum_{u_j=-P}^{+P} W_K(k_{\text{of}},k_{\text{if}};\,u_i,u_j)\, x(i-u_i,j-u_j). \quad (4.2)$$

For convenience, we can split the y feature accumulator in two separate y_{in} and y_{out}

and rewrite Equation 4.2 as follows:

$$y_{\text{out}}(k_{\text{of}};\, i,j) := y_{\text{in}}(k_{\text{of}};\, i,j) + \sum_{u_i=-P}^{+P} \sum_{u_j=-P}^{+P} W_K(k_{\text{of}},k_{\text{if}};\, u_i,u_j)\, x_{\text{in}}(k_{\text{if}};\, i-u_i,j-u_j). \tag{4.3}$$

The specific task performed by the HWCE is the acceleration of this convolution-accumulation step as defined by Equation 4.3. x_{in} and y_{in} are treated as input streams and y_{out} is treated as an output stream.

The HWCE can be configured at design time to support in an optimal way 3×3, 5×5, 7×7, 9×9 or 11×11 weights, though all convolutions can be supported with all accelerators by software (with a performance and energy penalty). As customary for CV applications, borders are neglected; thus the y image is smaller than x by $(K-1)$ pixels in both directions. To compute a $K \times K$ convolution, the engine must use all pixels in the neighborhood of the output pixel, as shown in Equation 4.3. A naive strategy to compute convolution would be: *i*) keep $(K-1) \cdot K$ unchanged pixels from the last iteration; *ii*) load the K pixels of the next column; *iii*) compute the new output pixel; *iv*) discard the oldest K pixels and return to *i*). However, this strategy requires a memory bandwidth of K pixels/cycle in input (plus 1 for y_{in}) to sustain a throughput of 1 output pixel/cycle.

Instead, we adopted a strategy to extract windows from a linear data stream by storing $(K-1)$ lines of the image and K pixels of the following line in a conceptual "shift register", as proposed for example in Bosi et al. [8]. This is a well established technique in image processing, used also in other fields such as stereo vision [139] to reduce memory bandwidth at the expense of the necessity of a register file that can be accessed in a parallel fashion. The strategy is explained by Figure 4.3: in the first phase, the shift register is filled with input pixels. When this buffer is full, there is enough data for a full convolution window and the computation can begin. Each cycle, *i*) a new pixel is inserted in the shift register; *ii*) the top left pixel in the buffer is discarded; *iii*) the leftmost $K \times K$ window is used to compute the output pixel. In this way, the memory bandwidth requirement is only 1 pixel/cycle in input (plus 1 for y_{in}) to reach the peak throughput of 1 output pixel/cycle. This strategy is more scalable and suited for integration in a shared-memory cluster with respect to the naive one, as it minimizes the ports required by the HWCE on the shared-memory interconnect (typically one of the critical paths of a cluster). Thanks to the

word interleaving strategy adopted in the low-latency interconnect, the HWCE traffic nicely spreads between all banks instead of hitting always a single one; the reduced bandwidth requirements using the strategy shown in Figure 4.3 minimizes interference with PEs working in parallel by further reducing banking contention.

To provide more efficient support for convolution-accumulation over a set of input feature maps, the HWCE can be programmed to perform multiple iterations of the step of Equation 4.3. The outermost loop (i.e. that spanning the output feature maps set) is typically managed via software, for two reasons: first, we deemed the additional hardware to manage it too expensive for the very small gain (waking up an idle core every several thousand cycles is acceptable); second, a fully computed output feature map can be transferred to a higher level of the memory hierarchy via a DMA transfer, which anyways implies software intervention.

Engine

The convolution engine is the inner core of the HWCE, which performs the actual computation. The HWCE engine receives x_{in} and y_{in} as streams and produces a y_{out} stream following Equation4.3. The engine was designed to be as unaware as possible of details such as the layout of data in memory and the size of the image, as well as to be entirely streaming-oriented in that it continues to work as long as it is fed with new pixels. The streams are generated outside the engine, in the wrapper module described in Section 4.2.2. The streams follow the AXI4-Stream protocol interface, that is based on a simple valid-ready handshake: the valid bit comes from the stream source, the ready bit from the sink; a packet exchange is valid only when both are asserted in the same cycle.

The HWCE engine can be configured at design time to work with 32-bit or 16-bit pixel data; in the latter case, since the I/O streams are fixed to 32 bits, the engine works at a peak throughput of 2 pixels/cycle. CNNs are typically trained using single precision floating point, but as pixels and weights typically have low dynamic range fixed point numbers can also be used, with an often cited "golden-point" for precision of 12 bits [92][140]. In our case we chose to support standard sizes well understood by software to ease communication with the rest of the PULP cluster. We will refer to the 16-bit mode for the remainder of this Chapter except when noted otherwise.

The datapath of the HWCE engine (Figure 4.4) is composed of three sets of blocks:

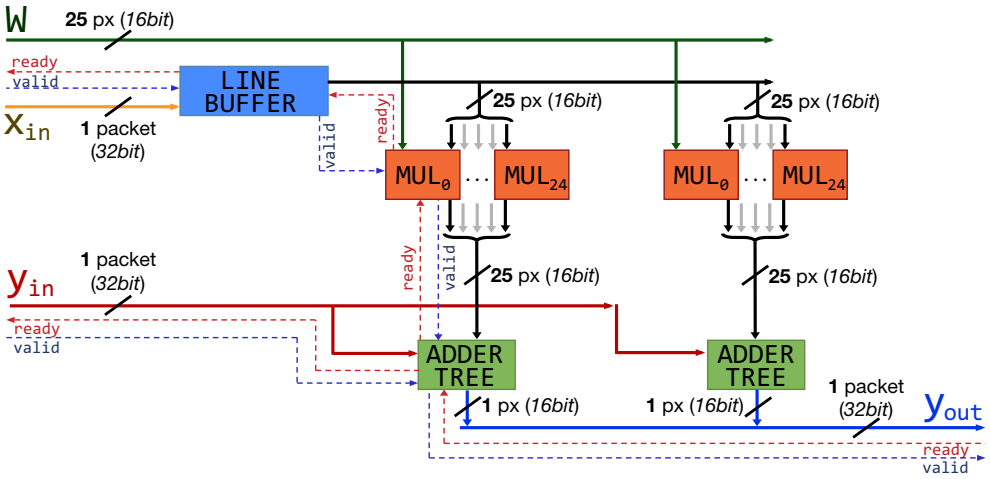


Figure 4.4: Internal structure of the *engine* in the 16-bit configuration.

the line buffer, the multipliers and the adder trees. These blocks communicate using an internal streaming protocol with the same handshake used in AXI4-Stream: a transaction is valid when both the valid and ready signals are asserted.

The *line buffer* implements the conceptual "shift register" storing input pixels as explained in Section 4.2.2. It takes the x_{in} stream as input and outputs two windows of K^2 pixels per cycle after the initial loading phase (one in case the HWCE is in the 32-bit configuration). The width of the line buffer is a design-time parameter that must have the form $d_{\text{LB}} + K - 1$, where d_{LB} is a power of 2. Feature maps with lines longer than $d_{\text{LB}} + K - 1$ are not supported in hardware, but are easily supported via software as described in Section 4.2.3.

To reduce the burden of a naive implementation of the line buffer based on a long shift register, as well as to make the line buffer flexible, the internal microarchitecture of this element relies on two hardware FIFO queues, a "big" one d_{LB} elements deep and $(K-1) \times 32$ bits wide (which implements the first $K-1$ lines of the conceptual shift register), and a "small" one $K/2+1$ elements deep and 32 bits wide (which implements the last incomplete line). This choice offers several distinct advantages:

- the amount of internal switching is greatly reduced, which reduces the cost of the line buffer in terms of dynamic power.
- the FIFO can be used to implement a "virtual" line buffer of any depth comprised between K and d_LB elements.

- the FIFO could be implemented as a standard-cell memory (SCM) cut (built with latches and clock gating cells), which allows to further reduce its energy cost [2].

The output windows of the line buffer are multiplied with the stored weights in the multipliers. The multiplied value is right-shifted by a runtime-configurable number of bits to support fixed-point multiplication. Finally, the products are summed up in two adder trees (one in the 32-bit configuration), adding also the current y_{in} pixels to produce the final y_{out} stream packet. The modules implementing this functionality, that naturally are the heart of the engine data path, are called *sum-of-products* or SoP units.

Control is performed by means of a small finite-state machine embedded in the engine, which is in charge of activating the various parts of the datapath (just the line buffer in the preload phase, all blocks afterwards) and ensures that the x_{in} and y_{in} streams are always in sync to ensure correctness.

It must be noted that a possible way to improve the performance of the HWCE with respect to this work is to compute multiple filters on the same x_{in} stream simultaneously. While this would certainly improve performance, we believe it would not similarly impact energy efficiency: first, while a small part of the HWCE could remain shared (namely, the register file and the line buffer), the SoP units that dominate the HWCE area and dynamic power would have to be replicated, as well as all the facilities related to the y_{in} and y_{out} streams. We expect the overall energy consumption per full convolutional layer of a HWCE with more SoPs to be only marginally better than the one presented here, unless the energy efficiency of SoPs is also increased [3].

Weight loader

As convolution operates repeatedly on a same relatively small set of weights, it is obviously convenient to load them inside the HWCE at the start of the computation. Two main possible design choices can be adopted: either loading them manually inside

[2] It must be noted that except from Section 4.4 the results reported in this Chapter refer to a FIFO implemented using normal flip-flops.

[3] This is in the assumption that the number of TCDM ports is limited to 4 (the HWCE presented here uses 3). If more than 4 ports are needed to feed the HWCE y_{in} and y_{out} streams, we expect the results to be comparable or even marginally worse than in a smaller HWCE, unless this increase is matched by a significant improvement in the energy efficiency of the SoPs.

memory-mapped registers during the offload phase, or using a lightweight offload in which just a pointer to the weights is set. The former strategy has two shortcomings with respect to the latter: first, the offload procedure is much longer, making the HWCE less convenient to use for small and medium-sized images. Second, as the HWCE is able to keep in queue more than 1 job (2 in the default configuration), the weight registers would have to be duplicated. For these reasons, we chose the latter option for the HWCE.

The weight loader module is composed of a finite-state machine to generate memory requests and a set of registers to store weights, implemented with SCMs. An additional task performed by the weight loader is modifying the weights in case a HWCE with a small filter size K is used to compute a bigger convolution (e.g., a 11×11 convolution computed with a 5×5 HWCE). In this case, the bigger convolution kernel is split in smaller kernels that may be partially superimposed a number of times (typically a power of 2); to cope with this, some of the weights must be shifted by the appropriate amount of bits since they will be applied multiple times.

Wrapper

The HWCE wrapper is responsible for the communication between the convolution engine and the shared-memory cluster, both in the data and in the control plane. The former task is performed in the *source* and *sink* submodules that convert the address-based protocol of the cluster into the internal stream-based communication or vice versa; the latter in the *slave* module, which features a memory-mapped register to store a queue of offloaded jobs. All the blocks in the HWCE wrapper are generic IPs that can be easily reused in the design of other HW Processing Elements. In fact, many of them - especially those related with the *slave* module - are shared with the HWPEs described in Chapter 2.

Both the *source* and *sink* modules contain a FIFO buffer of four elements (the amount sufficient to decouple the streams from possible memory contention stalls, as seen in early simulations) and an *address generator* block, plus some simple control logic. The address generator is composed by three modulo counters that can be used

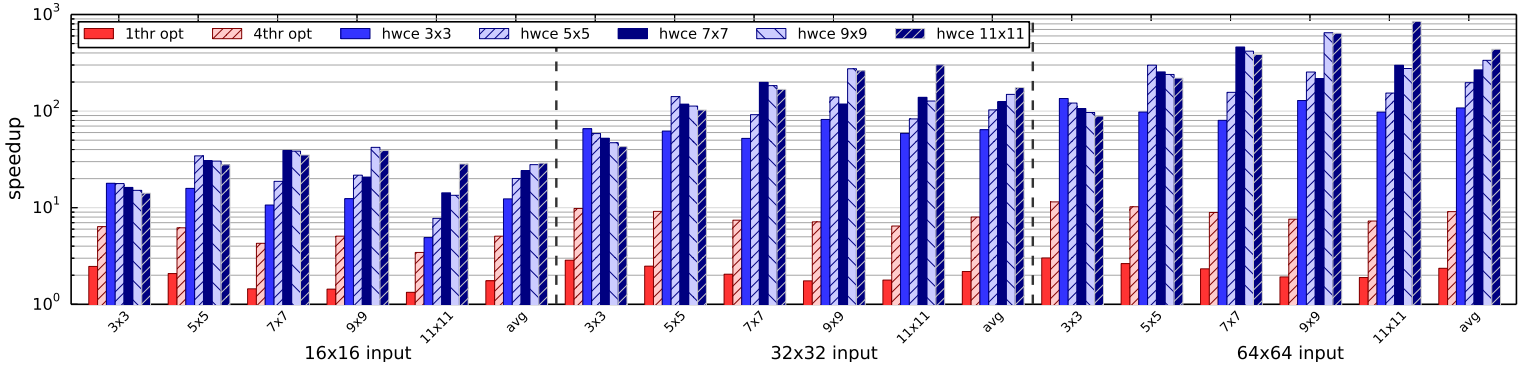


Figure 4.5: Log-scale speedup over naive single-thread convolution on 16×16, 32×32 and 64×64 input images.

to generate addresses for a 3D strided data pattern:

$$\begin{aligned} A(i) = A_{\text{base}} \\ &+ S_{\text{feat}} \cdot \left(i \,\mathbf{div}\, L_{\text{line}} \,\mathbf{div}\, L_{\text{feat}}\right) \\ &+ S_{\text{line}} \cdot \left(i \,\mathbf{div}\, L_{\text{line}} \,\mathbf{mod}\, L_{\text{feat}}\right) \\ &+ S_{\text{word}} \cdot \left(i \,\mathbf{mod}\, L_{\text{line}}\right) \end{aligned}$$

where $A(i)$ is the address of the i-th element, S are the strides and L the lengths of words, lines and feature maps (i.e. 2D blocks) in a given pattern. 3D strided data access is used to support accumulating multiple convolutions over a single output image, as in Equation 4.3.

Control of the HWCE is performed through a target port in its register file, that can be accessed by any core in the cluster via memory-mapped I/O. To offload a job to the HWCE, a core must first acquire a lock by reading a special location in the register file, which returns either the ID of the new job or an error code (if another core is already trying to offload a job or the job queue is full). Then, the necessary parameters (base pointers, strides and lengths for each stream + a pointer to the weights) must be set in the register file through simple writes. Finally, the job is triggered with a write to another special location, which also releases the lock.

The slave module takes care of executing the queue of jobs by dispatching control signals to the other modules in the HWCE (namely, the weight loader, the engine, and the stream sink and sources). When a job is selected for execution, the weight loader is activated to load the convolutional kernel weights inside its registers. After the end of the weight loading phase, the x_{in} source is activated to begin the *preload* phase, in which the line buffer is filled with the first rows of input data. When the buffer is full, the y_{in} source is also activated and the engine starts producing actual output pixels with a peak throughput of 1 streaming packet/cycle. Finally, when the sources and the sink have produced/consumed all the streams the slave module updates the running job ID and begins executing a new job (if present).

4.2.3 Programming model

To perform a convolution using the HWCE, the OpenRISC cores in the cluster must access the HWCE register file via normal memory-mapped load/store operations. To

ease the usage of the HWCE, we designed a simple hardware abstraction layer to perform the offload. Table 4.1 shows its main functions.

HAL function	**explanation**
ACQUIRE	polls on the job lock register in the HWCE register file until it gets a new job id
SETUP	writes convolution parameters in HWCE register file
TRIGGER	ends the offload by writing in the HWCE trigger register; also releases the job lock
WAIT	waits for an HWCE event, then checks if the running id is bigger than the given one

Table 4.1: HWCE Hardware Abstraction Layer.

Figure 4.6 formalizes the HWCE offload algorithm, which splits the input feature in vertical stripes of maximum width d_{LB} and minimum width of a single 32-bit word, i.e. 2 pixels for the 16-bit configuration. Convolutions on stripes with odd width can be supported by adding one vertical stripe of 1-pixel of zeros.

function HWCE_OFFLOAD(W_{ptr}, x_{ptr}, y_{ptr}, w, ...)
  $w_{\mathrm{int}} \leftarrow w$
  while $w_{\mathrm{int}} \geq 0$ **do**
    $w_{\mathrm{hwce}} \leftarrow \min\left(w_{\mathrm{int}}\rceil\,;\, d_{\mathrm{LB}}\right)$
    $w_{\mathrm{int}} \leftarrow w_{\mathrm{int}} - w_{\mathrm{hwce}}$
    $\mathrm{id}_{\mathrm{offload}} \leftarrow$ ACQUIRE()
    SETUP(W_{ptr}, x_{ptr}, y_{ptr}, w_{hwce}, ...)
    TRIGGER()
    $x_{\mathrm{ptr}} \leftarrow x_{\mathrm{ptr}} + w_{\mathrm{hwce}}$
    $y_{\mathrm{ptr}} \leftarrow y_{\mathrm{ptr}} + w_{\mathrm{hwce}}$
  end while
  return $\mathrm{id}_{\mathrm{offload}}$
end function

Figure 4.6: HWCE offload algorithm.

4.3 Results

4.3.1 HWCE synthesis results

We synthesized the HWCE in STMicroelectronics 28nm UTBB FD-SOI technology using Synopsys Design Compiler. We back-annotated switching activity using MentorGraphics ModelSim for RTL simulation and estimated dynamic and leakage power consumption at five operating points(reported in Table 4.2) at 25°C and with no body biasing. Power results also include dynamic power dissipation from the clock network as predicted by Design Compiler in topographical mode, and power consumption assuming the TCDM and I$ are implemented with SCM memories [141]. Execution time is derived from full-platform RTL simulations of the HWCE-augmented shared-memory cluster.

V_{DD} [V]	0.3	0.4	0.8	1.0	1.3
f_{max} [MHz]	2.5	22	400	588	775

Table 4.2: Operating points for the heterogeneous cluster.

Table 4.3 reports the relative area of the 16-bit HWCE described in this Chapter in terms of equivalent kilogates (kGE) [4]. The line buffer parameter d_{LB} is swept between 32 and 128 pixels and the filter size K between 3 and 11. We observe that these two parameters greatly affect area occupation; this is expected, as the total size of the line buffer increases linearly with both d_{LB} and K, while the number of multipliers is $2K^2$.

Area (kGE)		*Filter size*				
		3×3	5×5	7×7	9×9	11×11
d_{LB}	*32*	112	193	306	465	639
	64	124	216	342	505	697
	128	148	263	411	596	814

Table 4.3: Area of 16-bit HWCE in equivalent kilogates.

The design is pipelined in such a way that it is never a frequency bottleneck for the rest of the cluster by using Design Compiler retiming inside the datapath; in fact, while the nominal frequency for the cluster is that shown in Table 4.2, the HWCE is always synthesizable at a higher frequency.

[4] We define an equivalent gate as the smallest 2-input NAND cell available in our technology library.

4.3.2 Convolve-accumulate performance

To assess speedup over SW convolution and estimate the energy-efficiency boost, we developed a micro-benchmark focusing on a single 3×3, 5×5, 7×7, 9×9 or 11×11 convolution-accumulation (as defined in Equation 4.2) over a 16×16, 32×32 or 64×64 image. We developed two SW implementations of convolution-accumulation for each filter size: the `naive` implementation directly follows the definition in Equation 4.2 by using four nested loops (two on the output pixels and two for the convolution kernel W); the `optimized` implementation uses manual loop unrolling and explicit pointer arithmetic to achieve better performance. Both implementations run either on a single core or in four parallel threads running on the PEs of the shared-memory cluster. The HWCE implementation is based on the programming model discussed in Section 4.2.3.

Figure 4.5 shows the speedup over a single-thread naive implementation with the optimized SW and using HWCEs of different sizes to implement 3×3, 5×5, 7×7, 9×9 and 11×11 filters. All HWCEs have the line buffer parameter d_{LB} set to 32. Results include all overheads due to the offload procedure, including I$ misses and function stall overhead. By sweeping the size of the input image from 16×16 to 64×64, we observe a great increase in the acceleration's efficiency: this effect is due to the diminished impact of the phases during which no output is produced (offload and preload for the HWCE, branches for the SW optimizations) with respect to the rest of the run. Note that this behavior will not scale indefinitely, as eventually it is no longer possible to store bigger and bigger images in the shared memory, which becomes the system's bottleneck.

Unsurprisingly, the fastest way to compute a $N \times N$ convolution is using a HWCE configured with a $N \times N$ filter. Using a bigger HWCE requires an increased line buffer size, while a smaller HWCE requires more offloading, both resulting in performance losses. For example, a 11×11 convolution on a 64×64 image can be performed with a 3×3 HWCE with 16 distinct offloads, and a performance loss of $8\times$ with respect to the 11×11 HWCE (but is still $14\times$ faster than SW). On average, the speedup on the biggest image ranges between $40\times$ and $160\times$ compared to the the fastest single-thread SW implementation ($10\times$-$40\times$ over the multi-thread one); on the smallest image it is reduced to $7\times$-$18\times$ ($2\times$-$6\times$ over the multi-thread one).

To verify what is the overhead introduced by our shared memory paradigm, in

Table 4.4 we report average execution times (in clock cycles) of a 5×5 convolution-accumulation on a 32×32 image. For the test, we consider a 5×5 HWCE with small FIFOs of 4 elements in the sink and source units. We evaluate the runtime as divided in: *i*) a first configuration phase that is typically payed only at the first usage of the HWCE within a set of offloads, thanks to the job queue; *ii*) a startup/idle phase between the end of a job and the beginning of a new one; *iii*) the weight and line buffer preloading phase; *iv*) the actual convolution, i.e. the time during which the HWCE produces new output pixels. We compare three scenarios:

- an *ideal* scenario, in which the only time considered is that technically necessary for the engine data path to produce the 28×28 output pixels (i.e. infinite bandwidth towards the shared TCDM, no preloading phase);
- a *no contention* scenario, in which we consider the real interfaces of the HWCE towards the shared memory, but we disregard any performance degradation due to memory contention;
- a realistic *low contention* scenario, where the HWCE is used concurrently with control-oriented code run on a single OpenRISC core.
- a realistic *high contention* scenario, where the HWCE is used concurrently with a core executing control-oriented code and with a DMA transfer to enable a double buffering scheme (i.e. an additional traffic of 8 bytes per cycle towards the shared memory).

	first config	*startup/idle*	*preload*	*convolution*	***total (no config)***
ideal (infinite TCDM bandwidth)	-	-	-	392	**392**
no contention	258	2	67	448	**517**
efficiency vs ideal	-	-	-	88%	**76%**
low contention	258	2	86	509	**597**
efficiency vs no contention	100%	100%	78%	88%	**87%**
efficiency vs ideal	-	-	-	77%	**66%**
high contention	258	2	88	546	**636**
efficiency vs low contention	100%	100%	98%	93%	**94%**
efficiency vs no contention	100%	100%	77%	82%	**81%**
efficiency vs ideal	-	-	-	72%	**62%**

Table 4.4: HWCE overheads.

With respect to the infinite bandwidth scenario, the necessity to move data in and out of the accelerator causes a 25% efficiency loss, that could be reduced by increas-

ing the bandwidth towards the TCDM; this, however, would make the logarithmic interconnect more complex and increase its power consumption. The main source of overhead between the second and the third scenario is actually the self-contention between HWCE x_{in}, y_{in} and y_{out} streams, which is unavoidable in a flexible shared-memory platform such as the one we propose but can be effectively diminished by increasing the size of the sink and source FIFOs (e.g., increasing their size to 8 elements practically removes all self-contention in the low contention scenario). Some more overhead is visible between the low and high contention scenarios, introduced by the DMA traffic on the TCDM. Also in this case, the overhead can be reduced by increasing the size of the FIFOs. The memory contention overhead is essentially the cost of supporting the tightly-coupled shared memory acceleration paradigm; we deem this cost more than acceptable as in exchange for a performance loss of less than 30% for the "steady-state" convolution phase we gain total flexibility and the ability to freely intermix HWCE jobs and normal SW code, with no hidden copy overhead within the cluster.

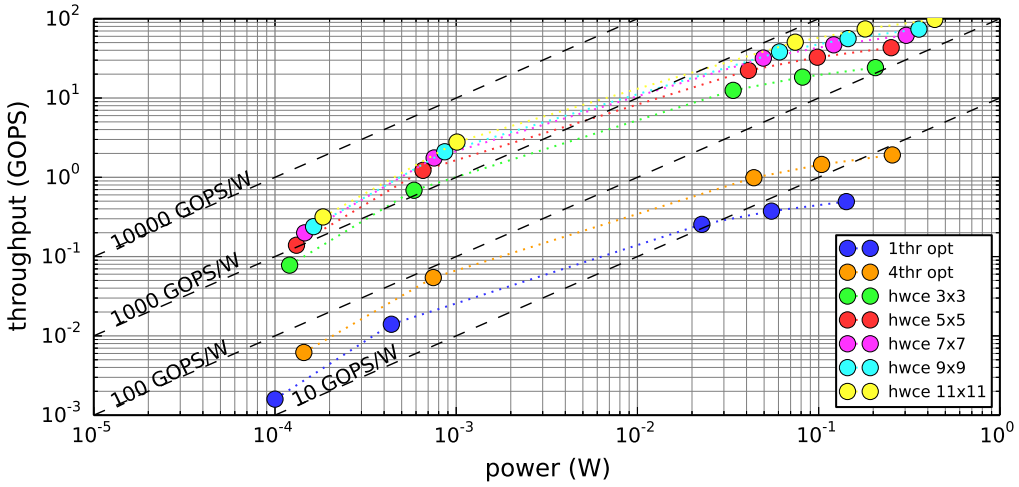


Figure 4.7: Energy efficiency (average throughput in equivalent GOPS vs total platform power consumption).

4.3.3 Convolve-accumulate power & energy

Energy-wise, our results show significant savings compared to SW solutions for convolutional workloads. In Figure 4.7, we show average energy efficiency of the SW convolutions and HWCEs benchmarked in Figure 4.5 in the case of a 64×64 input

image, at the operating points of Table 4.2. We consider throughput in terms of equivalent GOPS; this value is computed as the output pixel throughput multiplied by the minimum number of RISC operations needed to compute a pixel (i.e., $4W^2$). Power shown in Figure 4.7 is the total of the shared-memory cluster, keeping three cores off when the HWCE is on. Peak efficiency is 2.75 TOPS/W considering equivalent RISC operations, reached with the 11×11 HWCE at the 0.4V operating point. By comparison, the peak efficiency of the SW implementation is 72 GOPS/W ($\sim 38\times$ less). We also note that peak efficiency is reached near the voltage threshold (i.e. with V_{DD} at 0.4V or 0.3V) with a total power budget within 100 µW and 1 mW for the whole cluster. Even at the fastest configuration, total power is always kept under 1 W.

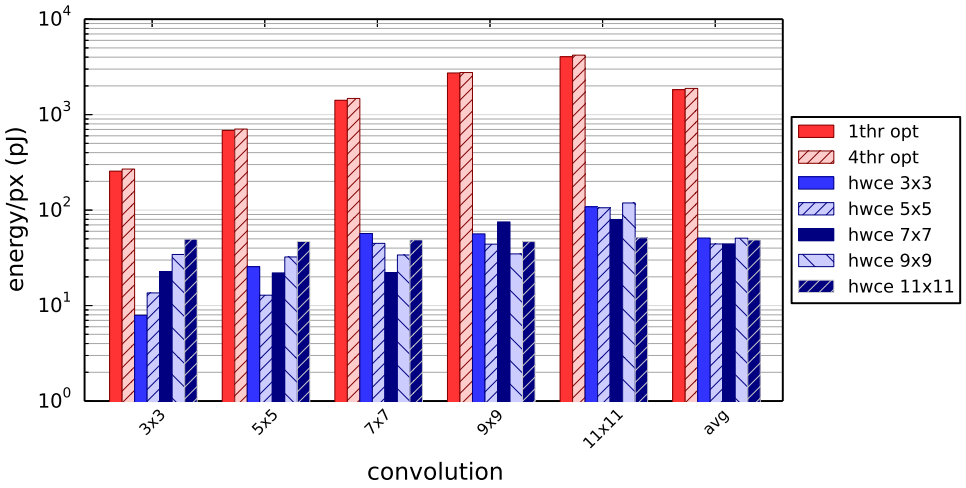


Figure 4.8: Energy spent per output pixel at 22 MHz (0.4V operating point).

To contrast SW convolution on OpenRISC and the HWCE convolution, in Figure 4.8 we plot the energy spent by the PE or HWCE to compute a single output pixel at 22 MHz (the most efficient operating point). On average all HWCEs are able to compute $\sim 40\times$ more pixels than PEs with the same energy; average consumption is 35 pJ/px instead of almost 2000 pJ/px. By comparison, a 2.1Wh battery for a wearable device would suffice for more than 50 billion convolutions on a 64×64 image. Point-wise, the 3×3 HWCE reaches the peak efficiency of 6.5 pJ/px in the 3×3 convolution.

Table 4.5 contrasts our results with those of several state-of-the-art platforms that were described in Section 1.4, showing that this work improves state of art in terms of

		technology	*maturity*	*performance*	*energy eff.*
NeuFlow [87]		IBM 45nm	post-layout	147 GMAC/s	245 GMAC/s/W
Qadeer et al. [93]		IBM 45nm	post-synthesis	205 GMAC/s	118 GMAC/s/W
ShiDianNao [91]		TSMC 65nm	post-layout	64 GMAC/s	200 GMAC/s/W
Camunas-Mesa et al. [142]		350nm	silicon	16.6 Mspike/s	185 Mspike/s/W
IBM TrueNorth [95]		Samsung 28nm	silicon	-	46 Gspike/s/W
Origami [92][140]	@0.8 V @1.2 V	UMC 65nm UMC 65nm	silicon silicon	28 GMAC/s 73 GMAC/s	402 GMAC/s/W 218 GMAC/s/W
PULP+HWCE	***@0.4V*** ***@1.0V***	**ST 28nm FD-SOI**	**post-layout**	**0.70 GMAC/s** **18.5 GMAC/s**	**688 GMAC/s/W** **103 GMAC/s/W**
PULP+HWCE	***@0.65V*** ***@1.05V***	**UMC 65nm**	**silicon**	**2.41 GMAC/s** **10.69 GMAC/s**	**261 GMAC/s/W** **104 GMAC/s/W**

Table 4.5: Comparison with other state-of-the-art CNN accelerators and brain-inspired platforms.

energy efficiency in its most efficient point. We also anticipate results for the HWCE implementation in UMC 65nm technology, that are further described in Section 4.4. For better consistency, all results are reported in terms of MAC operations as the definition of equivalent RISC ops may vary from source to source.

4.3.4 ConvNet benchmark

We used a complete CNN on our platform to further test the HWCE; its topology is the *small* one that was described in Chapter 3 as shown in Figure 3.6. Convolutional layers use a piecewise linear approximation of the hyperbolic tangent (executed in SW) for activation, while pooling ones use max-pooling. The CNN is fed with a 32×32 grayscale input image, and is sized to fully fit in the 16kB of cluster shared memory so that DMA data transfers from L2 can be completely hidden with double buffering. This is consistent with many of the CNNs listed in literature, which are used to classify a small window of an image at a time.

Figure 4.9 reports execution time in cycles for a full-SW CNN or a HWCE-accelerated one (both using either 1 or 4 threads for SW sections); we split this time between the various layers composing the CNN. CNNs are dominated by convolutional layers, and thus convolutions constitute the main bottleneck as processors employ respectively 74.6% and 72.6% of their time performing them. The overall speedup given by augmenting the cluster with a HWCE is 3.86× and 3.41× for the single- and 4-thread implementations respectively. Figure 4.9 also shows that the com-

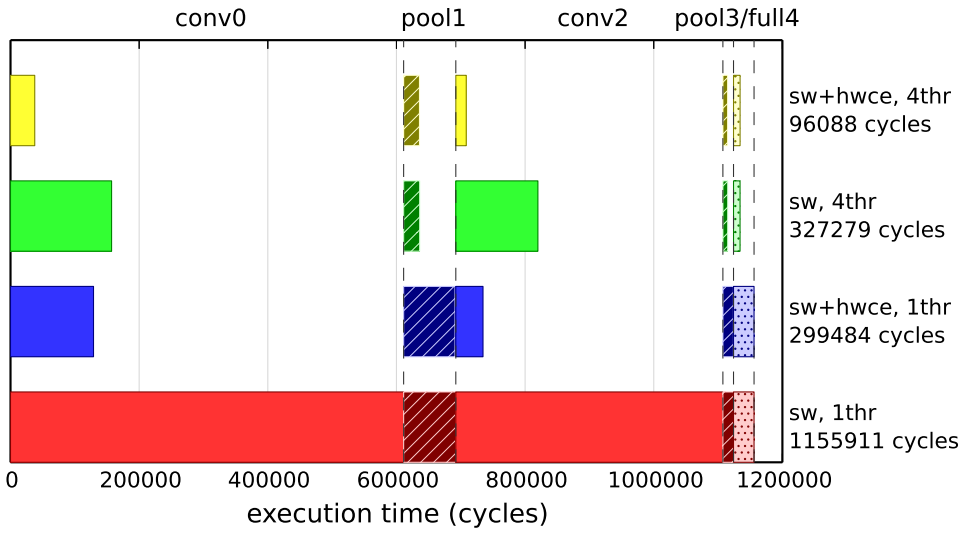


Figure 4.9: CNN benchmark execution time.

bined action of SW parallelism and HW acceleration achieves an overall 12× speedup over sequential execution; HWCEs essentially hit the wall of Amdahl's law, as the overall speedup is respectively 98.2% and 93.2% of the Amdahl limit. Note that, as convolutions would be more dominant in CNNs with more complex topology [140], the impact of the HWCE is also likely to be higher.

4.4 The Mia Wallace SoC

In April 2015, a PULP SoC codenamed "Mia Wallace" was taped out using the UMC 65nm technology. It contains an updated version of the cluster with respect to what was described in this Chapter and in Chapter 3; there are four OpenRISC cores redesigned from scratch (called OR10N) [68], a new shared instruction cache [136], several improvements to the DMA controller; moreover it has a relatively large L2 memory of 256 kB that can be used to perform significant computation. The chip contains a version of the HWCE essentially identical to the one described here, with a 5×5 HWCE and a 32-words line buffer. The HWCE is configured to work on two 16-bit pixels per cycle. The main microarchitectural difference between the HWCE implemented in Mia Wallace and the version described in the rest of this Chapter is that the register file and the line buffer are implemented using SCM cuts, to save area and dynamic power with respect to normal flip-flops. Figure 4.10 shows

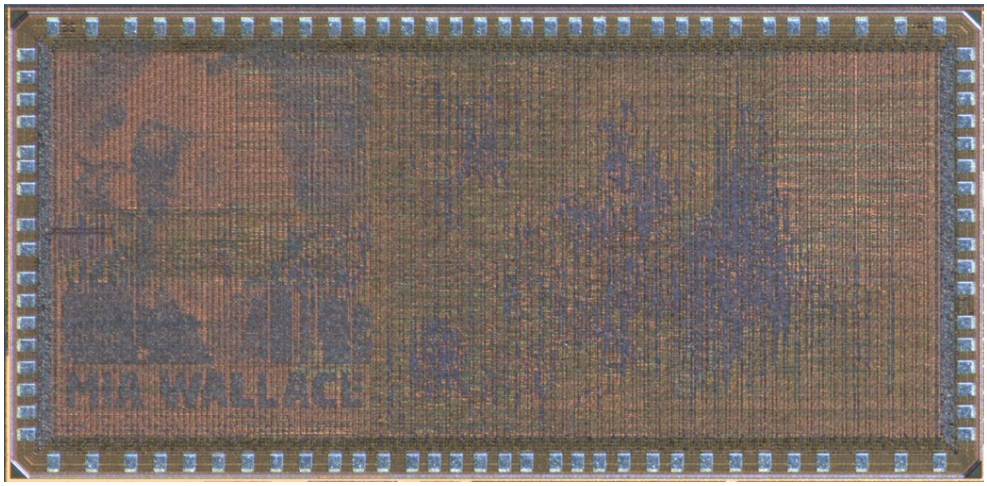


Figure 4.10: Mia Wallace die microphotograph.

a microphotograph of the Mia Wallace die, which measures 3.95 mm×1.88 mm.

To check the correct functionality of the HWCE and verify its peak energy efficiency, we used a simple test where CNN convolutional layers are computed multiple times on a 32×32 image stored in the TCDM. This test tracks accurately a scenario where the computation-to-communication ratio is very high and using a technique such as the *horizontal tiling* strategy (described in Section 3.3.4 and in Conti et al. [66]) the total execution time is dominated by computation time, with transfer time non-critical; this is often the case for CNNs, especially for the first layers that greatly expand the dimensionality of the data set (e.g. GoogLeNet [138])[5].

In Figures 4.11 and 4.12 we plot respectively frequency and power measured on the tester for the Mia Wallace chip while running the test previously described; we sweep the supply voltage V_{DD} between 0.65 V and 1.2 V and the body-biasing voltage V_{BB} between −0.4 V and 0.4 V. Figure 4.13 combines the two measurements to estimate the energy efficiency in terms of GMAC/s/W while using the HWCE at nominal throughput. The efficiency peak is 263 GMAC/s/W and is reached at minimum V_{DD} (0.65 V) while applying a small amount of reverse body biasing (−100 mV). In this operating point, the PULP cluster consumes 7.5 mW of power while working at 54 MHz.

[5]Note that this is true **only** if one does considers an efficient scheme for data movement such as the tiling strategies described in 3.3.4, or similar ones such as that of Peemen et al. [143]. Without using one of these strategies (e.g. in purely streaming accelerators such as nn-X [89]) it is necessary to rely on external memory as storage for intermediate results, which impacts performance in an extremely severe way with respect to ideal peak throughput [144].

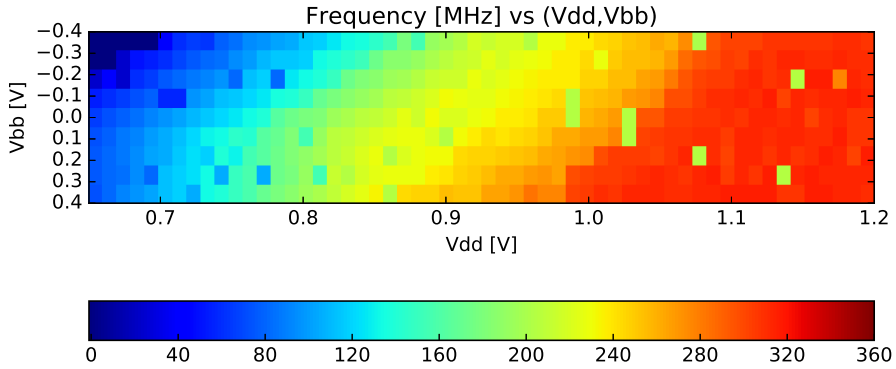


Figure 4.11: Frequency sweep on the HWCE test for the Mia Wallace chip while varying the supply voltage V_{DD} and the body-biasing V_{BB}.

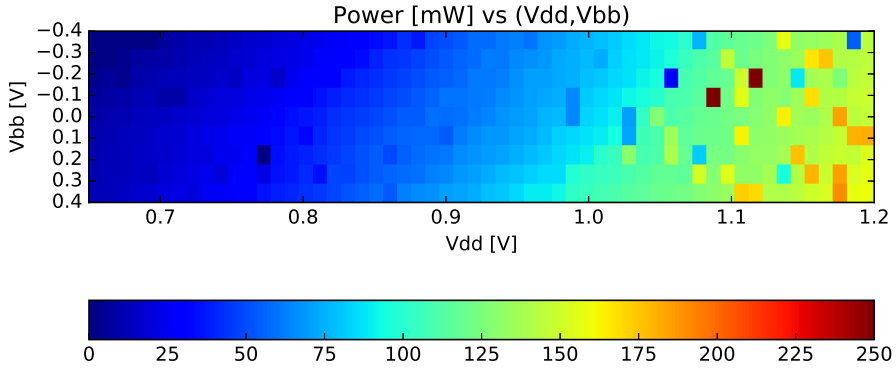


Figure 4.12: Power sweep on the HWCE test for the Mia Wallace chip while varying the supply voltage V_{DD} and the body-biasing V_{BB}.

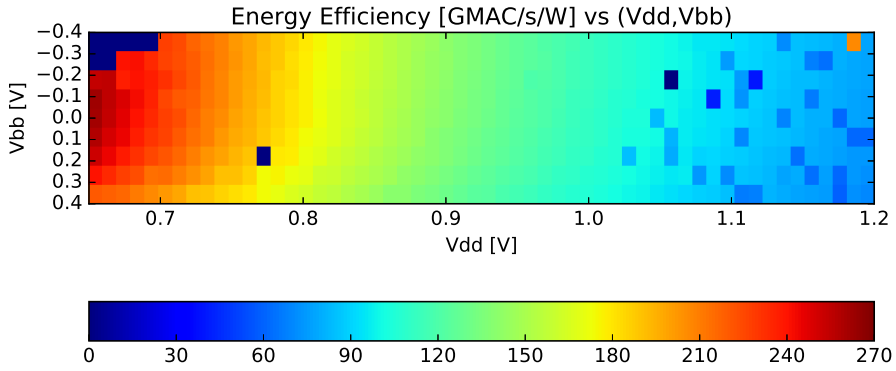


Figure 4.13: Energy efficiency sweep on the HWCE test for the Mia Wallace chip while varying the supply voltage V_{DD} and the body-biasing V_{BB}.

With respect to the GoogLeNet task mentioned in Section 4.1, the Mia Wallace SoC would be able to achieve 1 fps within a power envelope of less than 10 mW, using the efficient *horizontal tiling* strategy discussed in Section 3.3.4[145].

4.4.1 Scaling to ST 28nm FD-SOI

To compare this number with the pre-silicon ones reported for the initial HWCE architecture in ST 28nm FD-SOI technology, we can scale power using the simple model proposed by Cavigelli and Benini [140], i.e.

$$P_{\text{st28}} = P_{\text{umc65}} \frac{65\,\text{nm}}{28\,\text{nm}} \left(\frac{V_{\text{DD,st28}}}{V_{\text{DD,umc65}}} \right)^2 \tag{4.4}$$

We consider two operating points for both technologies, defined by their operating frequency[6]: a *high efficiency* and a *high performance* point, as defined by Table 4.6.

[6]For ST 28nm we estimate the probable operating points by interpolating results from post-layout static timing analysis, using a slow-slow corner.

In both technologies we don't consider using any body biasing.

		UMC 65nm	ST 28nm FD-SOI
High efficiency	*Frequency*	65 MHz	65 MHz
	V_{DD}	0.65 V	0.55 V
	Throughput	2.4 GMAC/s	2.4 GMAC/s
	Power	**9.2 mW**	**2.8 mW**
	Efficiency	**261 GMAC/s/W**	**859 GMAC/s/W**
High efficiency	*Frequency*	300 MHz	300 MHz
	V_{DD}	1.05 V	0.75 V
	Throughput	10.7 GMAC/s	10.7 GMAC/s
	Power	**103.2 mW**	**22.7 mW**
	Efficiency	**104 GMAC/s/W**	**471 GMAC/s/W**

Table 4.6: Power and Energy efficiency projection.

To compare the results with the ones presented in the other Sections of this Chapter (e.g. in Table 4.5), it is sufficient to consider a factor of ~ 4 between the number of MAC operations and that of equivalent RISC operations; therefore, our silicon results scaled to 28 nm appear to be slightly better than what estimated by power analysis and reported in Section 4.3.3. This is most likely due to two separate effects: on one hand, timing and power analysis use guardbands that might be on the pessimistic side, especially for relatively new processes such as the ST 28nm FD-SOI; on the other hand, we expect the scaling to be optimistic as Equation 4.4 disregards the impact of leakage that is higher in ST 28nm, due to the deeper integration. Moreover, the usage of SCMs (i.e. latches) in place of flip-flops in the line buffer and in the HWCE register file likely contribute to the reduction of the overall power in the UMC 65nm HWCE used in the Mia Wallace SoC.

4.4.2 Flexibility of the platform: Huffman decoding use case

As extensively argued during the course of this thesis, the main advantage of our approach to heterogeneity versus state-of-the-art ASICs is that it provides a much higher level of flexibility, while achieving competitive levels of energy efficiency (see Table 4.5). Use cases for this approach are literally infinite, but to show a practical example we report here a simple experiment considering that we may want to store the CNN weights in a compressed format and decompress them on-the-flight while

running the network. For the sake of this experiment, we will consider a simple compression scheme based on Huffman coding [146].

Let us consider the CNN network proposed by Cavigelli et al. [147] for scene parsing, which requires 8.1 GMAC operations per frame (considering the multi-scale version) and ∼1.24 MB of parameters for a 320 × 240 input frame. Using optimal Huffman compression on the network weights, we could reduce their size by a factor of 3.1×. Specifically, we computed a single Huffman code table for all weights in the network; then we divided it in blocks encoded separately. Since the code table is shared between all blocks, the preamble of each encoded block contains only the expected number of output bytes. To decode the Huffman-encoded weights, we ported the freely available `libhuffman` library [148] to PULP, optimized it, and adapted it so that multiple blocks can be decoded in parallel using the four cores available in the Mia Wallace chip[7].

Our version of `libhuffman` was able to decode Huffman encoded weights at a rate of ∼ 3.9 bits/cycle, i.e. enough to decode all weights in the network in 8.9 ms, considering the high-performance operating point of Table 4.6 (in UMC 65nm technology). In the same operating point, the CNN would require ∼ 758 ms; given the difference of two orders of magnitude between the execution, the two operations can be easily pipelined (decompressing the next weights while the current set is being used), hiding the decompression overhead almost completely[8].

Moreover, as the compression scheme is entirely implemented in software, the Mia Wallace SoC could support a plethora of other similar compression schemes (e.g. Adaptive Huffman) without any architectural change. At the same time, the presence of the heterogeneous architecture improves the energy efficiency on the bulk of the CNN computation by two orders of magnitude, as shown in the previous Sections of this Chapter. This provides a significant advantage with respect to fixed function platforms as it allows to adapt to situations with diverse requirements but at the same time can reach a level of efficiency comparable to a fully custom solution.

[7] We used the PULP OpenMP runtime, obtaining a speedup of ∼ 3.5× with respect to the original `libhuffman` port, which uses a single block and cannot be easily parallelized due to the sequential nature of bit-by-bit Huffman decoding by parsing the Huffman code tree.

[8] As Huffman decoding in this specific case is far from dominating the execution time, one could also think of using only one/two cores to execute it and use the others for other tasks e.g. DMA control. Our paradigm ensures a high degree of freedom in how to use the software cores and how to coordinate them with the bulk of the computation executed on the HWCE.

4.5 Conclusions

To support state-of-the-art BICV algorithms on highly battery-constrained devices such as WSN nodes and wearable computers, we need to extract as much performance as possible from every pJ of energy. In this Chapter, we provided a contribution towards this goal with the design of an energy-optimized streaming based engine for convolution computation and of a *wrapper* for efficient data/control integration in a shared-L1 multicore cluster. Our power analysis results show that the HWCE-augmented PULP cluster reaches state-of-the-art convolution energy-efficiency: up to 2.75 TOPS/W, spending 35 pJ/px on average in the most efficient configuration. We also validated our results on real silicon, showing results that are competitive in terms of energy efficiency with the state-of-the-art Origami ASIC [92][140], while proposing a platform that is significantly more flexible thanks to the internal architecture of the HWCE and obviously to the availability of the four PULP software cores.

Coupling efficiency with flexibility is key to the success of specialized architectures [93]. We argue that even if the HWCE is accelerating a single kernel, it nevertheless brings about a good level of application flexibility due to its applicability in deep convolutional networks, which are able to perform a huge variety of different tasks with state-of-the-art accuracy [85][84][86]. Therefore, it is a significant step towards the goal of the development of a heterogeneous parallel platform or system where "traditional" cores cooperate with accelerators or coprocessors that are able to implement a computational model different from the traditional Von Neumann machine, but still capable of universal or quasi-universal computation. In the case of the HWCE, the main limitation from this point of view is related to the nature of the underlying CNN model that has been shown to be capable of many tasks with excellent results in the computer vision and signal processing fields, but is not capable of fully universal computation. However, our work has the main purpose of showing a methodology and arguing that it can be used to yield excellent results; we argue that applying a similar methodology to other kinds of brain-inspired or even more exotic computational models is possible and likely effective.

Chapter 5

Conclusions

In Chapter 1, we surveyed many of the challenges and opportunities that are currently encountered by computing. These can be briefly resumed as follows in "four challenges":

1. the end of Dennard's scaling [13] and the twilight of Moore's law make it increasingly difficult to scale performance upwards by "simple" technological scaling, i.e. by relying on the increasing number of transistors in each technological node with no architectural innovation. Even if Moore's law can be prolonged for several more decades [149], the thermal constraint given by the utilization wall will likely require extensive advancements in computer architecture to make effective and energy-efficient use of the available transistors.

2. the insurgence of ubiquitous connected sensing devices (known as the "Internet-of-Things" or IoT) means that the amount of raw data produced is growing faster than our ability to transfer and use it, a phenomenon known as the *data deluge* [23]. Previously confined to relatively niche domains such as high-energy physics [150], the necessity to cope with huge amounts of data has been shifting towards mainstream computing. To reduce the quantity of traffic (and the consequent wasted energy), there is a strong push to move computation inside the sensors themselves, so that only higher level, more information-dense data is transferred [151][57]. This, however, requires significant work towards energy-efficient near-sensor computation, as sensor nodes are typically heavily power- and energy-constrained.

3. in part as a consequence of the availability of previously unconceivable amounts

of data, and in part of the availability of ever-more-powerful computing devices, *machine learning* techniques are becoming a mainstream algorithmic device to solve problems that were previously considered unsolvable [152][153]. While these algorithms are ideal candidates to solve the problem described in the previous point by being integrated in smart sensors, their computational burden is high (and growing) not only for the training phase, but also for simple inference. This will require a breakthrough in architectural energy efficiency.

4. the fabrication cost of chips in deeply integrated technologies increases with every process node, making it increasingly less convenient to design fully specialized computing devices such as ASICs in advanced nodes [99]; this is the strongest reason for which flexibility and the ability to execute general-purpose software will remain relevant in the future.

The results presented in this PhD thesis make it clear that *architectural heterogeneity* is a powerful technique to improve energy efficiency of computing systems, therefore targeting the "four challenges" listed above. We have shown three different (though linked) examples of heterogeneity in Chapters 2, 3 and 4, and in all cases we can conclude that coupling a more general-purpose platform with a more specialized one yields an improvement in efficiency that can vary from $\sim 10\times$ up to almost $1000\times$ due to specialization. We have contributed techniques that significantly lower the bar to access the "premium efficiency" granted by heterogeneity: a technique for systematic exploration of the heterogeneous design space (Chapter 2); a fully (and relatively easily) programmable system for heterogeneous acceleration in the low-power space (Chapter 3); and a heterogeneous cluster for CNN acceleration (Chapter 4). Regarding this last point, to the best of our knowledge this thesis contributes the first example of tight coupling between general-purpose cores and a brain-inspired accelerator for CNNs in the ultra-low power domain; this heterogeneous platform is silicon-proven and has been tested to reach up to 263 GMAC/s/W in 65nm technology, and potentially up to more than 800 GMAC/s/W in a more deeply integrated 28nm FD-SOI process.

Beyond these immediate technical results, as argued several times, the fundamental trade-off in heterogeneous platforms is that of *flexibility* versus *efficiency*, both of which are necessary to solve those challenges. In Chapter 2 we proposed a methodology that helps to simplify the exploration of the heterogeneous design space, therefore

alleviating the difficulty of the choice between the many possible trade-off points. Instead, the architectures we propose in Chapters 3 and 4 explore two different ways **not to** trade off efficiency with flexibility. In one case, we do this using a parallel programmable platform (PULP) and an offload model that reminds techniques that have been demonstrated to be effective in successful e.g. in the GPU computing domain, albeit applied to a totally different low-power scenario. In the other case, we propose an HW accelerator that uses the same kind of underlying methodology proposed in 2 to be tightly-coupled with a multi-core cluster, but is carefully optimized for CNN applications, essentially shifting flexibility from the architecture (that is now specialized) to the algorithm it accelerates. In both cases, we show that the addition of *flexible heterogeneity* is still extremely beneficial for energy efficiency, therefore targeting points 1,2,3 of the "four challenges" above, but also meet point 4. This last point might be regarded as the most important one since (as argued in Chapter 1) any engineering solution must be not only generally effective, but most importantly cost-effective. We believe the contribution of this thesis to be significant with respect to pushing this flexible heterogeneity paradigm into computing architecture in all spaces, and in particular in that of low-power smart sensor nodes.

5.1 Future research directions

We believe there is still a very significant body of research that could be performed building upon the results presented in this thesis.

First, in Chapter 4 we merely scratch the surface of "substituting" the flexibility versus efficiency trade-off with a more convenient accuracy versus efficiency one. For example, recent research has argued that deep neural networks are themselves approximable with several techniques [75][154][155] that could be applied to derive more energy-efficient hardware with a bounded loss in accuracy. An entire research field centering on this accuracy versus efficiency field is flourishing [74][73][156]. As a related research direction, it is clear that beyond CNNs and deep artificial neural networks in general, other brain-inspired models such as those based on spiking neural networks are becoming more and more relevant [97][95] and could open up additional design points in heterogeneous systems.

Moreover, we believe that while the general target for the heterogeneity approaches

we have shown in this thesis has been towards low-power and energy efficiency embedded systems such as autonomous WSN nodes, they respond to energy efficiency issues that are common up to much more powerful systems such as those for cloud servers [157] and HPC. In particular, the design challenges of HPC systems are also clearly related to energy efficiency via power and cooling costs, making the idea of using near-threshold many-core accelerators using replicated PULP clusters attractive.

In general, as argued in the very first lines of Chapter 1, in the computing architecture field problems and solutions tend to arise in a similar fashion at scales that differ by orders of magnitude in terms of performance, power consumption and mere physical size (ranging from implanted nano-devices to data centers). We believe that this is fundamentally true also for the ideas presented here, as they respond to challenges that are general and not specific to one or another domain in computing.

Chapter 6

Publications

2013

- **Francesco Conti**, Andrea Marongiu and Luca Benini. *Synthesis-friendly techniques for tightly-coupled integration of hardware accelerators into shared-memory multi-core clusters*. 2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'13).

2014

- Paolo Burgio, Giuseppe Tagliavini, **Francesco Conti**, Andrea Marongiu, Luca Benini. *Tightly-coupled hardware support to dynamic parallelism acceleration in embedded shared memory clusters*. 2014 Conference on Design, Automation & Test in Europe (DATE'14).

- **Francesco Conti**, Chuck Pilkington, Andrea Marongiu and Luca Benini. *HeP2012: Architectural Heterogeneity Exploration on a Scalable Many-Core Platform*. Best Paper Award at 2014 IEEE 25th International Conference on Application-specific Systems, Architectures and Processors (ASAP'14).

- **Francesco Conti**, Antonio Pullini and Luca Benini. *Brain-inspired Classroom Occupancy Monitoring on a Low-Power Mobile Platform*. Best Paper Award at the 10th Embedded Vision Workshop, Colocated with the 2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPRW'14).

- **Francesco Conti**, Davide Rossi, Antonio Pullini, Igor Loi and Luca Benini. *Energy-Efficient Vision on the PULP Platform for Ultra-Low Power Parallel Computing*. 2014 IEEE International Workshop on Signal Processing Systems (SiPS'14).

- Davide Rossi, Igor Loi, **Francesco Conti**, Giuseppe Tagliavini, Antonio Pullini and Andrea Marongiu. *Energy efficient parallel computing on the PULP platform with support for OpenMP*. 2014 IEEE 28th Convention of Electrical & Electronics Engineers in Israel (IEEEI'14).

- Paolo Meloni, Giuseppe Tuveri, Luigi Raffo, Igor Loi, and **Francesco Conti**. *Online process transformation for polyhedral process networks in shared-memory MPSoCs*. 2014 3rd Meditarranean Conference on Embedded Computing (MECO'14).

- Paolo Meloni, Giuseppe Tuveri, Luigi Raffo, Igor Loi, and **Francesco Conti**. *A Stream Buffer Mechanism for Pervasive Splitting Transformations on Polyhedral Process Networks*. Proceedings of 2014 International Workshop on Manycore Embedded Systems (MES'14).

2015

- **Francesco Conti** and Luca Benini. *A Ultra-Low-Energy Convolution Engine for Fast Brain-Inspired Vision in Multicore Clusters*. 2015 Conference on Design, Automation & Test in Europe (DATE'15).

- Davide Rossi, **Francesco Conti**, Andrea Marongiu, Antonio Pullini, Igor Loi, Michael Gautschi, Giuseppe Tagliavini, Philippe Flatresse and Luca Benini. *PULP: A Parallel Ultra-Low-Power Platform for Next Generation IoT Applications*. HOTCHIPS 2015.

- **Francesco Conti**, Andrea Marongiu, Chuck Pilkington and Luca Benini. *HeP2012: Performance and Energy Exploration of Architecturally Heterogeneous Many-Cores*. Journal of Signal Processing Systems, DOI 10.1007/s11265-015-1056-7

- **Francesco Conti**, Davide Rossi, Antonio Pullini, Igor Loi and Luca Benini. *PULP: A Ultra-Low Power Parallel Accelerator for Energy-Efficient and Flexible Embedded Vision.* Journal of Signal Processing Systems, DOI 10.1007/s11265-015-1070-9

2016

- **Francesco Conti**, Daniele Palossi, Andrea Marongiu, Davide Rossi and Luca Benini. *Enabling the Heterogeneous Accelerator Model on Ultra-Low Power Microcontroller Platforms.* 2016 Conference on Design, Automation & Test in Europe (DATE'16).

- Paolo Meloni, Gianfranco Deriu, **Francesco Conti**, Igor Loi, Luca Benini and Luigi Raffo. *Curbing the Roofline: a Scalable and Flexible Architecture for CNNs on FPGA.* Accepted in 2nd Workshop on Design of Low Power Embedded Systems (LP-EMS'16).

- Antonio Pullini, **Francesco Conti**, Davide Rossi, Igor Loi, Michael Gautschi and Luca Benini. *A Heterogeneous Multi-Core System-on-Chip for Energy Efficient Brain Inspired Vision.* Accepted in Late Breaking News session of the 2016 IEEE International Symposium on Circuits and Systems (ISCAS'16).

Glossary

6T-SRAM 6-transistor Static Random-Access Memory. 19

8T-SRAM 8-transistor Static Random-Access Memory. 19

ASIC Application-Specific Integrated Circuit. 18, 22, 72, 80, 102, 104, 106

BBMUX Body Bias Multiplexer. 56, 57

BICV Brain-Inspired Computer Vision. 79, 80, 104

BOX Buried Oxide. 52, 114

CNN Convolutional Neural Network. 7, 21, 22, 50, 78–80, 83, 85, 97–99, 102–104, 106, 107

CV Computer Vision. 52, 53, 79, 80, 82, 84

DLP Data Level Parallelism. 12

DMA Direct Memory Access. 27, 41, 42, 47, 48, 56, 65, 68, 73, 81, 85, 94, 95, 97, 98, 103

FBB Forward Body Biasing. 56, 58–61, 65, 70, 71

FD-SOI Fully Depleted Silicon-on-Insulator. 23, 52, 56, 58, 59, 62, 73, 76, 92, 101, 102, 106

FIFO First In, First Out (queue). 86–88, 94, 95

FLL Frequency-Locked Loop. 56

FPGA Field Programmable Gate Array. 16, 17, 22

GOPS Billions of Operations Per Second. 6, 59–61, 63, 69–71, 76

GPIO General Purpose Input/Output. 55

GPU Graphics Processing Unit. 13, 16, 17, 24, 54, 107

HLS High-Level Synthesis. 26, 32, 33, 35–37, 48

HPC High-Performance Computing. 108

HWCE HardWare Convolution Engine. 7, 22, 80–82, 84–88, 90–104

HWPE HardWare Processing Element. 6, 26, 27, 29–41, 45–48, 77, 81, 88

I/O Input/Output. 54, 85, 90

ILP Instruction Level Parallelism. 12, 14, 80

IoT Internet-of-Things. 51, 105

ISA Instruction Set Architecture. 13, 24, 55

L1 Level 1 memory or cache. 13, 17, 26, 27, 29, 55, 64, 66, 67

L2 Level 2 memory or cache. 13, 54–58, 62, 64–68, 71, 76, 98

LVT Low Voltage Threshold. 58

MAC Multiply-Accumulate. 79, 97, 99

MCU Micro-Controller Unit. 18, 20, 52, 73, 74

NMOS n-type Metal-Oxide-Semiconductor transistor. 58

PE Processing Element. 6, 27, 29–33, 41, 45–49, 53, 55

PMOS p-type Metal-Oxide-Semiconductor transistor. 58

PULP Parallel Ultra-Low Power Platform. 4–7, 20, 51–63, 65, 66, 70–78, 80, 81, 85, 98, 99, 103, 104, 107, 108

RBB Reverse Body Biasing. 56

RISC Reduced Instruction Set Computer. 23, 55, 62, 65, 75, 97

SCM Standard Cell Memory. 58, 59, 87, 88, 92, 98, 102

SIMD Single Instruction, Multiple Data-stream. 15, 18, 22

SoC System-on-Chip. 4, 5, 25, 26, 54–58, 75, 98, 101–103

SoP Sum-of-Products. 87

SPI Serial Protocol Interface. 55, 58, 70–72, 74, 76

SRAM Static Random-Access Memory. 58

TCDM Tightly-Coupled Data Memory. 27, 35, 56, 58, 59, 62, 64, 66, 67, 69, 71, 76, 81, 87, 92, 94, 95, 99

TLP Thread Level Parallelism. 12

UAV Unmanned Aerial Vehicle. 51–54, 70–72, 76

ULP Ultra-Low Power. 3, 61, 65, 72, 76

UTBB Ultra-Thin Body and BOX. 52, 58, 59, 92

VLIW Very-Long Instruction Word. 15

WSN Wireless Sensor Node. 51, 54, 108

Bibliography

[1] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P.-C. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor, "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future," *IEEE Micro*, vol. 31, no. 2, pp. 86–95, Mar. 2011.

[2] S. Jain, S. Khare, S. Yada, V. Ambili, P. Salihundam, S. Ramani, S. Muthukumar, M. Srinivasan, A. Kumar, S. Gb, R. Ramanarayanan, V. Erraguntla, J. Howard, S. Vangal, S. Dighe, G. Ruhl, P. Aseron, H. Wilson, N. Borkar, V. De, and S. Borkar, "A 280mV-to-1.2V wide-operating-range IA-32 processor in 32nm CMOS," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, Feb. 2012, pp. 66–68.

[3] D. Fick, R. G. Dreslinski, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, N. Liu, M. Wieckowski, G. Chen, T. Mudge, D. Blaauw, and D. Sylvester, "Centip3De: A Cluster-Based NTC Architecture With 64 ARM Cortex-M3 Cores in 3D Stacked 130 nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 1, pp. 104–117, Jan. 2013.

[4] A. Marongiu, A. Capotondi, G. Tagliavini, and L. Benini, "Improving the programmability of STHORM-based heterogeneous systems with offload-enabled OpenMP," in *Proceedings of the First International Workshop on Many-core Embedded Systems - MES '13*. New York, New York, USA: ACM Press, 2013, pp. 1–8.

[5] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Mar. 2012, pp. 983–987.

[6] A. Rahimi, I. Loi, M. R. Kakoee, and L. Benini, "A Fully-Synthesizable Single-Cycle Interconnection Network for Shared-L1 Processor Clusters," in *2011 Design, Automation & Test in Europe*. IEEE, Mar. 2011, pp. 1–6.

[7] F. Conti, D. Palossi, A. Marongiu, D. Rossi, and L. Benini, "Enabling the Heterogeneous Accelerator Model on Ultra-Low Power Microcontroller Platforms," in *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '16. San Jose, CA, USA: EDA Consortium, 2016.

[8] B. Bosi, G. Bois, and Y. Savaria, "Reconfigurable pipelined 2-D convolvers for fast digital signal processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 3, pp. 299–308, Sep. 1999.

[9] "ISSCC 2015 Trends," http://isscc.org/doc/2015/isscc2015_trends.pdf.

[10] D. Luebke, "GPU Architecture: Implications & Trends," *ser. SIGGRAPH*, vol. 2008, 2008.

[11] G. E. Moore, "Cramming more components onto integrated circuits," *Readings in computer architecture*, p. 56, 2000.

[12] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, no. 5, pp. 256–268, 1974.

[13] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein, "Scaling, power, and the future of CMOS," in *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*, Dec. 2005, pp. 7 pp.–15.

[14] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, p. 67, May 2011.

[15] R. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253–266, Feb. 2010.

[16] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu, "Runnemede: An architecture for Ubiquitous High-Performance Computing," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013)*, Feb. 2013, pp. 198–209.

[17] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, no. 7, pp. 33–38, 2008.

[18] L. Benini and G. De Micheli, "Networks on chips: a new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[19] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *Proceeding of the 38th annual international symposium on Computer architecture - ISCA '11*, p. 365, 2011.

[20] M. B. Taylor, "Is dark silicon useful?" in *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*. New York, New York, USA: ACM Press, 2012, p. 1131.

[21] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges," *Ad Hoc Networks*, vol. 10, no. 7, pp. 1497–1516, Sep. 2012.

[22] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, Sep. 2013.

[23] R. G. Baraniuk, "More Is Less: Signal Processing and the Data Deluge," *Science*, vol. 331, no. 6018, pp. 717–719, Feb. 2011.

[24] F. Conti, C. Pilkington, A. Marongiu, and L. Benini, "He-P2012 : Architectural Heterogeneity Exploration on a Scalable Many-Core Platform," in *Proceedings of 25th IEEE Conference on Application-Specific Architectures and Processors*, 2014.

[25] F. Conti, A. Marongiu, C. Pilkington, and L. Benini, "He-P2012: Performance and Energy Exploration of Architecturally Heterogeneous Many-Cores," *Journal of Signal Processing Systems*, pp. 1–16, 2015.

[26] Tensilica, "Xtensa Architecture and Performance White Paper," no. October, 2005.

[27] Synopsys, "Processor Designer Datasheet."

[28] Movidius, *Innovation in Mobile Computational Imaging.*

[29] Silicon Hive, *Silicon Hive website.*

[30] Z. Lin, J. Sankaran, and T. Flanagan, "Empowering automotive vision with TI's Vision AccelerationPac," *TI White Paper*, 2013.

[31] J. Clemons, A. Pellegrini, S. Savarese, and T. Austin, "EVA : An Efficient Vision Architecture for Mobile Systems," in *Proceedings of 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2013.

[32] U. Ramacher, "Software-Defined Radio Prospects for Multistandard Mobile Phones," *Computer*, vol. 40, no. 10, pp. 62–69, Oct. 2007.

[33] F. Clermidy, R. Lemaire, X. Popon, D. Ktenas, and Y. Thonnart, "An Open and Reconfigurable Platform for 4G Telecommunication: Concepts and Application," *2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pp. 449–456, Aug. 2009.

[34] C. Jalier, D. Lattard, A. A. Jerraya, G. Sassatelli, P. Benoit, and L. Torres, "Heterogeneous vs homogeneous MPSoC approaches for a Mobile LTE modem," *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pp. 184–189, Mar. 2010.

[35] P. Paulin, C. Pilkington, and E. Bensoudane, "StepNP: a system-level exploration platform for network processors," *IEEE Design & Test of Computers*, vol. 19, no. 6, pp. 17–26, Nov. 2002.

[36] O. Pell, J. Bower, R. Dimond, O. Mencer, and M. J. Flynn, "Finite-Difference Wave Propagation Modeling on Special-Purpose Dataflow Machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 5, pp. 906–915, May 2013.

[37] 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," *IPSJ Transactions on System LSI Design Methodology*, vol. 5, pp. 71–95, 2012.

[38] J. Sabarad, S. Kestur, D. Dantara, V. Narayanan, and D. Khosla, "A reconfigurable accelerator for neuromorphic object recognition," in *17th Asia and South Pacific Design Automation Conference*. IEEE, Jan. 2012, pp. 813–818.

[39] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing Modular Hardware Accelerators in C with ROCCC 2.0," *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 127–134, 2010.

[40] J. Backus, "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978.

[41] G. Falcao, V. Silva, L. Sousa, and J. Andrade, "Portable LDPC decoding on multicores using OpenCL," *Signal Processing Magazine, IEEE*, vol. 29, no. 4, pp. 81–109, 2012.

[42] K. Shagrithaya, K. Kepa, and P. Athanas, "Enabling Development of OpenCL Applications on FPGA platforms," pp. 26–30, 2013.

[43] N. A. Brookwood, "AMD Fusion Family of APUs : Enabling a Superior , Immersive PC Experience," Ph.D. dissertation.

[44] D. Patterson, "The top 10 innovations in the new NVIDIA Fermi architecture, and the top 3 next challenges," *NVIDIA Whitepaper*, vol. 47, 2009.

[45] Plurality, "The HyperCore Architecture White Paper," 2010.

[46] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang, "EXOCHI," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation - PLDI '07*. New York, New York, USA: ACM Press, 2007, p. 156.

[47] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for accelerator-rich CMPs," *Proceedings of the 49th Annual Design Automation Conference - DAC 2012*, p. 843, 2012.

[48] P. Burgio, A. Marongiu, D. Heller, C. Chavet, P. Coussy, and L. Benini, "OpenMP-based Synergistic Parallelization and HW Acceleration for On-Chip Shared-Memory Clusters," *15th Euromicro Conference on Digital System Design: Architectures, Methods & Tools, Turkey (2012)*, pp. 751–758, Sep. 2012.

[49] M. Dehyadegari, A. Marongiu, M. R. Kakoee, L. Benini, S. Mohammadi, and N. Yazdani, "A tightly-coupled multi-core cluster with shared-memory HW accelerators," *2012 International Conference on Embedded Computer Systems (SAMOS)*, pp. 96–103, Jul. 2012.

[50] M. Dehyadegari, A. Marongiu, M. Kakoee, S. Mohammadi, N. Yazdani, and L. Benini, "Architecture Support for Tightly-Coupled Multi-Core Clusters with Shared-Memory HW Accelerators," *IEEE Transactions on Computers*, vol. PP, no. 99, pp. 1–1, 2014.

[51] F. Conti, A. Marongiu, and L. Benini, "Synthesis-friendly techniques for tightly-coupled integration of hardware accelerators into shared-memory multi-core clusters," in *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. IEEE, Sep. 2013, pp. 1–10.

[52] "STMicroelectronics STM32L476xx Datasheet."

[53] "SiliconLabs EFM32G210 Datasheet."

[54] "MSP430FR59xx Mixed-Signal Microcontrollers (Rev. E)," http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf.

[55] "Ambiq Apollo Data Brief."

[56] N. Ickes, Y. Sinangil, F. Pappalardo, E. Guidetti, and A. P. Chandrakasan, "A 10 pJ/cycle ultra-low-voltage 32-bit microprocessor system-on-chip," in *2011 Proceedings of the ESSCIRC (ESSCIRC)*. IEEE, Sep. 2011, pp. 159–162.

[57] D. Bol, J. De Vos, C. Hocquet, F. Botman, F. Durvaux, S. Boyd, D. Flandre, and J.-D. Legat, "SleepWalker: A 25-MHz 0.4-V Sub-mm2 7-uW/MHz Microcontroller in 65-nm LP/GP CMOS for Low-Carbon Wireless Sensor Nodes," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 1, pp. 20–32, Jan. 2013.

[58] F. Botman, J. D. Vos, S. Bernard, F. Stas, J.-D. Legat, and D. Bol, "Bellevue : a 50MHz Variable-Width SIMD 32bit Microcontroller at 0 . 37V for Processing-Intensive Wireless Sensor Nodes," in *Proceedings of 2014 IEEE Symposium on Circuits and Systems*, 2014, pp. 1207–1210.

[59] D. Jeon, Y. Kim, I. Lee, Z. Zhang, D. Blaauw, and D. Sylvester, "A low-power VGA full-frame feature extraction processor," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, May 2013, pp. 2726–2730.

[60] J. Oh, S. Lee, and H.-J. Yoo, "1.2-mW Online Learning Mixed-Mode Intelligent Inference Engine for Low-Power Real-Time Object Recognition Processor," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 5, pp. 921–933, May 2013.

[61] S. J. Carey, A. Lopich, D. R. W. Barr, Bin Wang, and P. Dudek, "A 100,000 fps vision sensor with embedded 535GOPS/W 256x256 SIMD processor array." Kyoto: IEEE, Jun. 2013, pp. C182–C183.

[62] N. Pinckney, D. Blaauw, and D. Sylvester, "Low-Power Near-Threshold Design: Techniques to Improve Energy Efficiency Energy-efficient near-threshold design has been proposed to increase energy efficiency across a wid," *IEEE Solid-State Circuits Magazine*, vol. 7, no. 2, pp. 49–57, Spring 2015.

[63] L. Chang, R. K. Montoye, Y. Nakamura, K. A. Batson, R. J. Eickemeyer, R. H. Dennard, W. Haensch, and D. Jamsek, "An 8T-SRAM for Variability Tolerance and Low-Voltage Operation in High-Performance Caches," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 4, pp. 956–963, Apr. 2008.

[64] 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)*, Oct. 2014, pp. 1–6.

[65] D. Rossi, I. Loi, F. Conti, G. Tagliavini, A. Pullini, and A. Marongiu, "Energy efficient parallel computing on the PULP platform with support for OpenMP," in *2014 IEEE 28th Convention of Electrical Electronics Engineers in Israel (IEEEI)*, Dec. 2014, pp. 1–5.

[66] F. Conti, D. Rossi, A. Pullini, I. Loi, and L. Benini, "PULP: A Ultra-Low Power Parallel Accelerator for Energy-Efficient and Flexible Embedded Vision," *Journal of Signal Processing Systems, to appear.*, 2015.

[67] D. Rossi, A. Pullini, M. Gautschi, I. Loi, F. K. Gurkaynak, P. Flatresse, and L. Benini, "A -1.8V to 0.9V Body Bias, 60 GOPS/W 4-Core Cluster in Low-Power 28nm UTBB FD-SOI Technology," in *Proceedings of the 2015 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference.*

[68] M. Gautschi, M. Scandale, A. Traber, A. Pullini, A. Di Federico, M. Beretta, G. Agosta, and L. Benini, "Tailoring Instruction-Set Extensions for an Ultra-Low Power Tightly-Coupled Cluster of OpenRISC Cores," in *Proceedings of VLSI-SOC 2015*, 2015.

[69] E. J. Hartman, J. D. Keeler, and J. M. Kowalski, "Layered neural networks with Gaussian hidden units as universal approximations," *Neural computation*, vol. 2, no. 2, pp. 210–215, 1990.

[70] J. Park and I. W. Sandberg, "Universal approximation using radial-basis-function networks," *Neural computation*, vol. 3, no. 2, pp. 246–257, 1991.

[71] J. Von Neumann, "First Draft of a Report on the EDVAC," *Contract W670ORD4926, US Army Ordnance Department and University of Pennsilvanya*, 1945.

[72] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI '11.* New York, New York, USA: ACM Press, 2011, p. 164.

[73] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '12*, p. 301, 2012.

[74] ——, "Neural Acceleration for General-Purpose Approximate Programs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, Dec. 2012, pp. 449–460.

[75] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "AxNN: Energy-efficient Neuromorphic Systems Using Approximate Computing," in *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, ser. ISLPED '14. New York, NY, USA: ACM, 2014, pp. 27–32.

[76] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[77] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Scene Parsing with Multiscale Feature Learning, Purity Trees, and Optimal Covers," p. 9, Feb. 2012.

[78] M. Riesenhuber and T. Poggio, "Hierarchical models of object recognition in cortex." *Nature neuroscience*, vol. 2, no. 11, pp. 1019–25, Nov. 1999.

[79] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[80] E. M. Izhikevich and G. M. Edelman, "Large-scale model of mammalian thalamocortical systems," *Proceedings of the national academy of sciences*, vol. 105, no. 9, pp. 3593–3598, 2008.

[81] W. Maass, T. Natschläger, and H. Markram, "Real-time computing without stable states: a new framework for neural computation based on perturbations." *Neural computation*, vol. 14, no. 11, pp. 2531–60, Nov. 2002.

[82] B. Schrauwen, D. Verstraeten, and J. Van Campenhout, "An overview of reservoir computing: theory, applications and implementations," in *Proceedings of the 15th European Symposium on Artificial Neural Networks*, 2007, pp. 471–482.

[83] R. Girshick, J. Donahue, T. Darrell, J. Malik, and U. C. Berkeley, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of 2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 580–587.

[84] A. Toshev and C. Szegedy, "DeepPose : Human Pose Estimation via Deep Neural Networks," in *Proceedings of 2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014.

[85] A. Karpathy and T. Leung, "Large-scale Video Classification with Convolutional Neural Networks," in *Proceedings of 2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732.

[86] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams, Y. Gong, and A. Acero, "Recent advances in deep learning for speech research at Microsoft," *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 8604–8608, May 2013.

[87] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pp. 257–260, May 2010.

[88] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "NeuFlow: A runtime reconfigurable dataflow processor for vision," in *CVPR 2011 Workshops*. IEEE, Jun. 2011, pp. 109–116.

[89] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks," in *CVPR 2014 Workshops*, 2014.

[90] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of ASPLOS 2014*. ACM Press, 2014, pp. 269–284.

[91] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting Vision Processing Closer to the Sensor," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 92–104.

[92] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A Convolutional Network Accelerator," in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, ser. GLSVLSI '15. New York, NY, USA: ACM, 2015, pp. 199–204.

[93] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing," in *Proceedings of the 40th Annual International Symposium on Computer Architecture - ISCA '13*. New York, New York, USA: ACM Press, 2013, p. 24.

[94] F. Conti and L. Benini, "A Ultra-low-energy Convolution Engine for Fast Brain-inspired Vision in Multicore Clusters," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 683–688.

[95] P. a. Merolla, J. V. Arthur, R. Alvarez-Icaza, a. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, Aug. 2014.

[96] P. Knag, J. K. Kim, T. Chen, and Z. Zhang, "A Sparse Coding Neural Network ASIC With On-Chip Learning for Feature Extraction and Encoding," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 4, pp. 1070–1079, Apr. 2015.

[97] A. S. Cassidy, P. Merolla, J. V. Arthur, S. K. Esser, B. Jackson, R. Alvarez-Icaza, P. Datta, J. Sawada, T. M. Wong, V. Feldman, A. Amir, D. B.-d. Rubin, F. Akopyan, E. McQuinn, W. P. Risk, and D. S. Modha, "Cognitive computing building block: A versatile and efficient digital neuron model for neurosy-

naptic cores," in *The 2013 International Joint Conference on Neural Networks (IJCNN)*. IEEE, Aug. 2013, pp. 1–10.

[98] P. U. Diehl, D. Neil, J. Binas, M. Cook, S.-C. Liu, and M. Pfeiffer, "Fast-Classifying, High-Accuracy Spiking Deep Networks Through Weight and Threshold Balancing."

[99] "Is the cost reduction associated with IC scaling over? — EE Times," http://www.eetimes.com/author.asp?section_id=36&doc_id=1286363.

[100] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems - ASPLOS '10*. New York, New York, USA: ACM Press, 2010, p. 205.

[101] G. D. Micheli, *Readings in Hardware/software Co-design*. Morgan Kaufmann, 2002.

[102] P. Ienne and R. Leupers, *Customizable embedded processors: design technologies and applications*. Academic Press, 2006.

[103] Kalray, *Kalray MPPA 256*.

[104] C. Pilato, F. Ferrandi, and D. Sciuto, "A design methodology to implement memory accesses in high-level synthesis," *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2011 Proceedings of the 9th International Conference*, pp. 49–58, 2011.

[105] P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagne, and G. Nicolescu, "Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 7, pp. 667–680, Jul. 2006.

[106] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," *Computer Vision and Pattern Recognition, 2001. CVPR 2001*, vol. 1, pp. I–511–I–518, 2001.

[107] E. Rosten and T. Drummond, "Fusing points and lines for high performance tracking," *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, pp. 1508–1515 Vol. 2, 2005.

[108] ——, "Machine learning for high-speed corner detection," *Computer Vision–ECCV 2006*, pp. 1–14, 2006.

[109] M. Magno, F. Tombari, D. Brunelli, L. Di Stefano, and L. Benini, "Multimodal Abandoned/Removed Object Detection for Low Power Video Surveillance Systems," *2009 Sixth IEEE International Conference on Advanced Video and Signal Based Surveillance*, pp. 188–193, Sep. 2009.

[110] F. Conti, A. Pullini, and L. Benini, "Brain-inspired Classroom Occupancy Monitoring on a Low-Power Mobile Platform," in *CVPR 2014 Workshops*, 2014.

[111] "Qualcomm Snapdragon 810 Processor Product Brief."

[112] "NVIDIA Tegra 4 Family GPU Architecture - Tegra-X1-whitepaper-v1.0.pdf."

[113] K. Y. Ma, P. Chirarattananon, S. B. Fuller, and R. J. Wood, "Controlled Flight of a Biologically Inspired, Insect-Scale Robot," *Science*, vol. 340, no. 6132, pp. 603–607, Mar. 2013.

[114] D. Jacquet, G. Cesana, P. Flatresse, F. Arnaud, P. Menut, F. Hasbani, T. Di Gilio, C. Lecocq, T. Roy, A. Chhabra, C. Grover, O. Minez, J. Uginet, G. Durieu, F. Nyer, C. Adobati, R. Wilson, and D. Casalotto, "2.6GHz ultra-wide voltage range energy efficient dual A9 in 28nm UTBB FD-SOI," in *2013 Symposium on VLSI Technology (VLSIT)*, Jun. 2013, pp. C44–C45.

[115] D. Jacquet, F. Hasbani, P. Flatresse, R. Wilson, F. Arnaud, G. Cesana, T. Di Gilio, C. Lecocq, T. Roy, A. Chhabra, C. Grover, O. Minez, J. Uginet, G. Durieu, C. Adobati, D. Casalotto, F. Nyer, P. Menut, A. Cathelin, I. Vongsavady, and P. Magarshack, "A 3 GHz Dual Core Processor ARM Cortex-A9 in 28 nm UTBB FD-SOI CMOS With Ultra-Wide Voltage Range and Energy Efficiency Optimization," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 4, pp. 812–826, Apr. 2014.

[116] *OpenRISC 1000 Architecture Manual*, 2012.

[117] W. B. Pennebaker, J. L. Mitchell, C. Fogg, and D. LeGall, *MPEG Digital Video Compression Standard*. Chapman & Hall, 1997.

[118] B. D. Lucas and T. Kanade, "An Iterative Image Registration Technique with an Application to Stereo Vision," in *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'81. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1981, pp. 674–679.

[119] C.-H. Hsieh and T.-P. Lin, "VLSI architecture for block-matching motion estimation algorithm," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 2, no. 2, pp. 169–175, Jun. 1992.

[120] E. Brockmeyer, L. Nachtergaele, F. Catthoor, J. Bormans, and H. de Man, "Low power memory storage and transfer organization for the MPEG-4 full pel motion estimation on a multimedia processor," *IEEE Transactions on Multimedia*, vol. 1, no. 2, pp. 202–216, Jun. 1999.

[121] N. Zhang, M. Paluri, M. A. Ranzato, T. Darrell, L. Bourdev, and U. C. Berkeley, "PANDA : Pose Aligned Networks for Deep Attribute Modeling," in *Proceedings of 2014 IEEE Conference on Computer Vision and Pattern Recognition*, vol. 2, 2014.

[122] D. Scaramuzza, M. Achtelik, L. Doitsidis, F. Friedrich, E. Kosmatopoulos, A. Martinelli, M. Achtelik, M. Chli, S. Chatzichristofis, L. Kneip, D. Gurdan, L. Heng, G. H. Lee, S. Lynen, M. Pollefeys, A. Renzaglia, R. Siegwart, J. Stumpf, P. Tanskanen, C. Troiani, S. Weiss, and L. Meier, "Vision-Controlled Micro Flying Robots: From System Design to Autonomous Navigation and Mapping in GPS-Denied Environments," *IEEE Robotics Automation Magazine*, vol. 21, no. 3, pp. 26–40, Sep. 2014.

[123] L. Heng, D. Honegger, G. H. Lee, L. Meier, P. Tanskanen, F. Fraundorfer, and M. Pollefeys, "Autonomous Visual Mapping and Exploration With a Micro Aerial Vehicle," *Journal of Field Robotics*, vol. 31, no. 4, pp. 654–675, 2014.

[124] R. J. Wood, B. Finio, M. Karpelson, K. Ma, N. O. Pérez-Arancibia, P. S. Sreetharan, H. Tanaka, and J. P. Whitney, "Progress on 'pico' air vehicles,"

The International Journal of Robotics Research, vol. 31, no. 11, pp. 1292–1302, Sep. 2012.

[125] P. Chen, K. Hong, N. Naikal, S. S. Sastry, D. Tygar, P. Yan, A. Y. Yang, L.-C. Chang, L. Lin, S. Wang, E. Lobatón, S. Oh, and P. Ahammad, "A Low-bandwidth Camera Sensor Platform with Applications in Smart Camera Networks," *ACM Trans. Sen. Netw.*, vol. 9, no. 2, pp. 21:1–21:23, Apr. 2013.

[126] M. Gautschi, M. Muehlberghuber, A. Traber, S. Stucki, M. Baer, R. Andri, L. Benini, B. Muheim, and H. Kaeslin, "SIR10US: A tightly coupled elliptic-curve cryptography co-processor for the OpenRISC," in *2014 IEEE 25th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Jun. 2014, pp. 25–29.

[127] D. Rossi, I. Loi, G. Haugou, and L. Benini, "Ultra-Low-Latency Lightweight DMA for Tightly Coupled Multi-Core Clusters," in *Proceedings of the 11th ACM Conference on Computing Frontiers - CF '14*. New York, New York, USA: ACM Press, 2014, pp. 1–10.

[128] I. Miro-Panades, E. Beigné, Y. Thonnart, L. Alacoque, P. Vivet, S. Lesecq, D. Puschini, A. Molnos, F. Thabet, B. Tain, K. B. Chehida, S. Engels, R. Wilson, and D. Fuin, "A Fine-Grain Variation-Aware Dynamic Vdd-Hopping AVFS Architecture on a 32 nm GALS MPSoC," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 7, pp. 1475–1486, Jul. 2014.

[129] P. Meinerzhagen, S. M. Y. Sherazi, A. Burg, and J. N. Rodrigues, "Benchmarking of Standard-Cell Based Memories in the Sub-Vt Domain in 65-nm CMOS Technology," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 1, no. 2, pp. 173–182, Jun. 2011.

[130] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks," *arXiv:1312.6229 [cs]*, Dec. 2013.

[131] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," Apr. 2009.

[132] "Centeye Stonyman / Haskbill silicon documentation," 2013.

[133] "The Crazyflie Nano Quadcopter — Bitcraze."

[134] Choi, W. S., Shu, G., Talegaonkar, M., Liu, Y., Wei, D., Benini, Luca, and Hanumolu, P. K., "A 0.45-to-0.7V 1-to-6Gb/S 0.29-to-0.58pJ/b source-synchronous transceiver using automatic phase calibration in 65nm CMOS," in *Solid- State Circuits Conference - (ISSCC), 2015 IEEE International*, 22-26 Feb. 2015, pp. 1–3.

[135] D. Rossi, A. Pullini, I. Loi, M. Gautschi, F. K. Gurkaynak, A. Teman, J. Constantin,, A. Burg, I. M. Panades, E. Beigné, F. Clermidy, F. Abouzeid, P. Flatresse, and L. Benini, "193 MOPS/mW 162 MOPS, 0.32V to 1.15V Voltage Range Multi-Core Accelerator for Energy-Efficient Parallel and Sequential Digital Processing," 2016.

[136] I. Loi, D. Rossi, G. Haugou, M. Gautschi, and L. Benini, "Exploring Multi-banked shared-L1 Program Cache on Ultra-low Power, Tightly Coupled Processor Clusters," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, ser. CF '15. New York, NY, USA: ACM, 2015, pp. 64:1–64:8.

[137] M. Gautschi, M. Schaffner, L. Benini, and others, "A 65nm CMOS 6.4-to-29.2 pJ/FLOP@ 0.8 V shared logarithmic floating point unit for acceleration of nonlinear function kernels in a tightly coupled processor cluster," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2016, pp. 82–83.

[138] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," *arXiv:1409.4842 [cs]*, Sep. 2014.

[139] M. Kuhn, S. Moser, O. Isler, F. K. Gurkaynak, A. Burg, N. Felber, H. Kaeslin, and W. Fichtner, "Efficient ASIC implementation of a real-time depth mapping stereo vision system," in *2003 IEEE 46th Midwest Symposium on Circuits and Systems*, vol. 3, Dec. 2003, pp. 1478–1481 Vol. 3.

[140] L. Cavigelli and L. Benini, "Origami: A 803 GOp/s/W Convolutional Network Accelerator," *arXiv:1512.04295 [cs]*, Dec. 2015.

[141] A. Teman, D. Rossi, P. Meinerzhagen, L. Benini, and A. Burg, "Controlled placement of standard cell memory arrays for high density and low power in 28nm FD-SOI," in *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, Jan. 2015, pp. 81–86.

[142] L. Camunas-Mesa, C. Zamarreno-Ramos, A. Linares-Barranco, A. J. Acosta-Jimenez, T. Serrano-Gotarredona, and B. Linares-Barranco, "An Event-Driven Multi-Kernel Convolution Processor Module for Event-Driven Vision Sensors," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 2, pp. 504–517, Feb. 2012.

[143] M. Peemen, A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for Convolutional Neural Networks," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, Oct. 2013, pp. 13–19.

[144] P. Meloni, G. Deriu, F. Conti, I. Loi, L. Benini, and L. Raffo, "Curbing the Roofline: a Scalable and Flexible Architecture for CNNs on FPGA," in *Proceedings of 2016 Workshop on Low Power Embedded Systems Design, to appear*, 2016.

[145] A. Pullini, F. Conti, D. Rossi, I. Loi, M. Gautschi, and L. Benini, "A Heterogeneous Multi-Core System-On-Chip for Energy Efficient Brain Inspired Vision," in *Proceedings of 2016 IEEE International Symposium on Circuits and Systems, Late Breaking News, to appear*, 2016.

[146] D. A. Huffman and others, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[147] L. Cavigelli, M. Magno, and L. Benini, "Accelerating Real-time Embedded Scene Labeling with Convolutional Networks," in *Proceedings of the 52Nd Annual Design Automation Conference*, ser. DAC '15. New York, NY, USA: ACM, 2015, pp. 108:1–108:6.

[148] "GitHub libhuffman web page," https://github.com/drichardson/huffman.

[149] "ISSCC 2016 Trends," http://isscc.org/doc/2016/ISSCC2016_TechTrends.pdf.

[150] J. Shiers, "The Worldwide LHC Computing Grid (worldwide LCG)," *Computer Physics Communications*, vol. 177, no. 1–2, pp. 219–223, Jul. 2007.

[151] "More than Moore - ITRS White Paper," http://www.itrs2.net/uploads/4/9/7/7/49775221/irc-itrs-mtm-v2_3.pdf.

[152] G. Anthes, "Deep learning comes of age," *Communications of the ACM*, vol. 56, no. 6, p. 13, Jun. 2013.

[153] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015.

[154] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural Networks with Few Multiplications," *arXiv:1510.03009 [cs]*, Oct. 2015.

[155] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with binary weights during propagations," *arXiv:1511.00363 [cs]*, Nov. 2015.

[156] G. Tagliavini, D. Rossi, A. Marongiu, and L. Benini, "Synergistic Architecture and Programming Model Support for Approximate Micropower Computing," in *VLSI (ISVLSI), 2015 IEEE Computer Society Annual Symposium on*. IEEE, 2015, pp. 280–285.

[157] A. Pahlevan, J. Picorel, A. Pourhabibi Zarandi, D. Rossi, M. Zapater, A. Bartolini, P. G. Del Valle, D. Atienza, L. Benini, and B. Falsafi, "Towards Near-Threshold Server Processors," in *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '16, 2016.