A Ultra-Low-Energy Convolution Engine for Fast Brain-Inspired Vision in Multicore Clusters

Francesco Conti* and Luca Benini*†
*Department of Electrical, Electronic and Information Engineering, University of Bologna, Italy
†Integrated Systems Laboratory, ETH Zurich, Switzerland
f.conti@unibo.it, lbenini@iis.ee.ethz.ch

***Abstract*— State-of-art brain-inspired computer vision algorithms such as Convolutional Neural Networks (CNNs) are reaching accuracy and performance rivaling that of humans; however, the gap in terms of energy consumption is still many degrees of magnitude wide. Many-core architectures using shared-memory clusters of power-optimized RISC processors have been proposed as a possible solution to help close this gap. In this work, we propose to augment these clusters with Hardware Convolution Engines (HWCEs): ultra-low energy coprocessors for accelerating convolutions, the main building block of many brain-inspired computer vision algorithms. Our synthesis results in ST 28nm FD-SOI technology show that the HWCE is capable of performing a convolution in the lowest-energy state spending as little as 35 pJ/pixel on average, with an optimum case of 6.5 pJ/pixel. Furthermore, we show that augmenting a cluster with a HWCE can lead to an average boost of 40x or more in energy efficiency in convolutional workloads.**

I. Introduction

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 *ConvNets* [1]) that are state-of-art in many accuracy benchmarks [2][3][4][5][6].

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-art BICV algorithms: wearable computers, wireless sensor networks, visual impairment aids are just some examples. For instance, let us imagine we want a Google Glass-like device to last a full day on its battery (2.1Wh) while processing in real-time a HD video stream with a CNN. If we assume an equivalent workload of 100 convolutions per frame at 30 fps (big, but not unrealistic), we would need to spend a maximum of 14pJ for each input pixel, while keeping an overall power budget of 87mW and delivering 600 GOPS of performance; that is, we would need an energy efficiency of ~7 GOPS/mW to stay real-time. Even in a less extreme setting, current architectures are not up to the job of delivering this level of efficiency.

Much research has recently focused on many-core architectures using many simple power-optimized RISC cores [7][8] to build up a high-performance, low-power platform. However, relying on pure SW is often suboptimal in the context of BICV because performing convolutions in software is inherently wasteful in terms of energy. 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 using a specialized convolution architecture, to provide a level of performance and energy efficiency unreachable by pure SW. To this end, we present 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 [8], a ultra-low power multicore platform featuring optimized OpenRISC cores designed for the power-efficient STMicroelectronics 28nm FD-SOI technology.

In this work, we show that by using the HWCE it is possible to spend as little as 6.5 pJ/output pixel in optimal cases, 40x better than what achievable by SW and improving more than 6x with respect to the state-of-art in convolution acceleration in a multi-core cluster [9]; moreover, we provide results for the implementation of a full Convolutional Neural Network using the HWCE.

II. Related work

Application-specific architectures for CV are a very active area of research. To improve energy efficiency, many platforms rely on dataflow engines, often implemented on FPGAs or CGRAs. Examples of this approach include Vortex [10][11] for biologically-inspired vision acceleration, and NeuFlow [12][13] and nn-X [14], which focus on acceleration of ConvNets. Differently from ours, they have loosely coupled accelerators that use a dataflow computational model and rely on explicit data copy between blocks.

Another common approach is to augment a RISC processor with a SIMD-like extension. Qadeer et al. [9] present a Convolution Engine (CE) implemented as an extension to a Tensilica processor and used like a SIMD extension. The CE also provides some flexibility: it features full ALUs and it can run not only convolutions, but also other operations with the same “map & reduce” structure. Commercial platforms following a similar trend include Movidius Myriad [15] and TI AccelerationPAC [16], that extend RISC processors using VLIW coprocessors. Clemons et al. [17] propose EVA (Efficient Vision Architecture), an asymmetric multicore featuring 1 out-of-order coordinating core and many supporting low-power cores, all augmented with

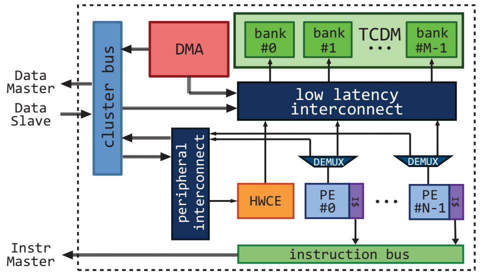

Figure 1. HWCE-augmented shared-memory PULP cluster.

978-3-9815370-4-8/DATE15/©2015 EDAA

custom SIMD-like accelerators for common operations such as dot products. These approaches directly extend processors and still rely on its instruction fetch & decode units; by contrast, our work completely bypasses the Von Neumann *fetch-decode-execute* bottleneck, which invariably brings at least one order of magnitude of energy loss.

DianNao [18] is an accelerator targeted specifically at deep neural networks that can execute linear, convolutional and pooling layers through a structure similar to our *engine* (Section III-B). However, whereas DianNao is meant to be used almost standalone, with a very lightweight processor used only for control flow and dedicated scratchpads for input and output, our HWCE was designed to be integrated with a shared-memory cluster of fully capable RISC processors. This leads to entirely different choices especially for what concerns storing and moving data.

Similarly to what we do with the HWCE, Cong et al. [19][20], Venkatesh et al.[21][22] and Goulding-Hotta et al. [23] propose to extend multicore clusters with shared-memory special-purpose cores sharing memory (at L2 in the two former cases, at L1 in the latter). Our previous work in Conti et al. [24] extended the P2012 parallel platform [25] with shared-L1 accelerators to achieve higher performance and energy efficiency in CV workloads. However, these accelerators are generated by high-level synthesis and they do not currently reach the same level of efficiency of a manually tuned core such as the HWCE.

Neuromorphic platforms go much deeper in brain-inspiration than CNN-based approaches. IBM TrueNorth [26][27] 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. However, state-of-art precision-recall performance of spiking NNs has not been shown yet. Camunas-Mesa et al. [28] propose an accelerator for spiking convolutional neural networks. The main limit of this approach is that event-based imaging sensors are not nearly as mature as frame-based ones, and accuracy results using this model are still behind the state-of-art.

In Section IV-C we list performance and energy efficiency of many of these approaches, contrasting them with our results.

III. Architecture

A. Shared-memory cluster

The target shared-memory cluster is that of the PULP [7][8] architecture, featuring 4 Processing Elements (*PE*s) with a highly power optimized microarchitecture based on OpenRISC 32-bit ISA, each one with a private instruction cache (*I$*) with all refill ports converging on a common instruction bus.

The PEs do not have private data caches, avoiding memory coherency overhead and increasing area efficiency, while they all share a 16kB multi-banked tightly coupled data memory (*TCDM*) acting as a L1 shared scratchpad. 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 [29]. As an additional master in the low-latency interconnect, the heterogeneous cluster includes a Hardware Convolution Engine (*HWCE*), whose internal architecture is detailed in Section III-B.

A lightweight, ultra-low-programming-latency, multi-channel DMA enables fast and flexible communication with other clusters, the L2 memory and external peripherals [30]. The DMA uses minimal request buffering and features a direct connection to the TCDM to remove 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.

B. 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.

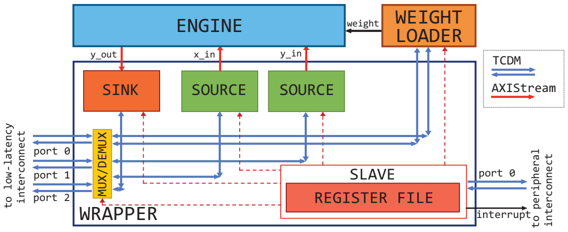

Figure 2. Architecture of the HW Convolution Engine.

The HWCE is algorithmically optimized to minimize memory bandwidth requirements to perform convolutions. It is structured in three main blocks: the *engine* proper (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).

1) Convolution strategy: In many applications, the output image is the sum of multiple different convolutions over the input; for this reason, we chose a *convolution-accumulation* step as the basic kernel to accelerate:

$$y_{\text{out}}(i,j) = y_{\text{in}}(i,j) + \sum_{u_i=-P}^{+P} \sum_{u_j=-P}^{+P} W_K(u_i,u_j) x_{\text{in}}(i-u_i,j-u_j) \quad (1)$$

where $K = 2P + 1$ is the size of the filter; the HWCE can be configured at design time to support in an optimal way 3x3, 5x5, 7x7, 9x9, or 11x11 convolutions, though all convolutions can be implemented in all accelerators (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 KxK convolution, the engine must use all pixels in the neighborhood of the output pixel, as shown in Equation 1. 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 shift register, as proposed in Bosi et al. [31]. The strategy is shown in Figure 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 KxK 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

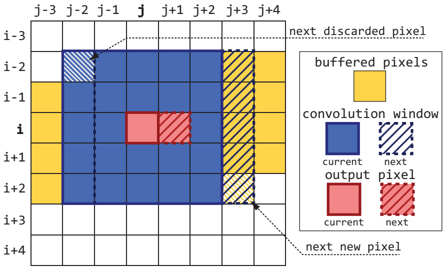

Figure 3. Strategy for 2D 5x5 convolution with linear data streams [31].

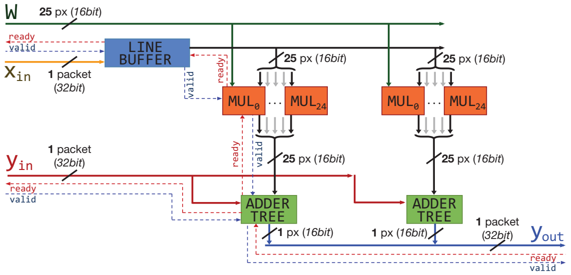

Figure 4. Internal structure of the *engine* in the 16-bit configuration.

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 3 minimizes interference with PEs working in parallel by further reducing banking contention.

Algorithms such as convolutional neural networks often accumulate convolutions of the type of Equation 1 over multiple input feature maps. To directly support this behavior, the HWCE can also be programmed to repeat convolution on multiple input blocks, accumulating on a single output image:

$$y_{\text{out}}(i,j) = y_{\text{in}}(i,j) + \sum_{p=1}^{N_{\text{blocks}}} \left(\left(W_K * x_{\text{in}}^{(p)} \right)(i,j) \right) \quad (2)$$

2) Engine: The convolution engine is the core of the HWCE, which performs the actual computation. The HWCE engine receives x_{in} and y_{in} as streams using the AXI4-Stream protocol and produces a y_{out} stream following Equations 1 or 2. 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 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 2 pixels/cycle. As most CV applications do not need full 32-bit precision for convolutions, we will refer to the 16-bit mode for the remainder of this work except when noted otherwise.

The datapath of the HWCE engine (Figure 4) is composed of three sets of blocks: the linebuffer, the multipliers and the adder trees. These blocks communicate by a simple streaming protocol using an handshake with a *ready* and a *valid* signal; a transaction is valid when both are asserted.

The *linebuffer* contains the shift register storing input pixels as explained in Section III-B1. 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 size of the linebuffer is a design-time parameter that must have the form $d_{\text{LB}} + K - 1$, where d_{LB} is a power of 2. By setting a runtime parameter d_{hwce} in the job offload phase, the HWCE can be configured to bypass part of the linebuffer to support other row widths. The d_{hwce} parameter must have the form $2^N + K - 1$; other widths need software support as explained in Section III-C.

The output windows of the linebuffer 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.

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 linebuffer in the preload phase, all blocks afterwards) and ensures that the x_{in} and y_{in} streams are always in sync to ensure correctness.

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 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. 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 11x11 convolution computed with a 5x5 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.

4) Wrapper: The HWCE wrapper is responsible of 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.

Both the *source* and *sink* modules contain a FIFO buffer of eight 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 to generate addresses for a 3D strided data pattern:

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

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

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 AXI sink and sources). When a job is selected for execution, the weight loader

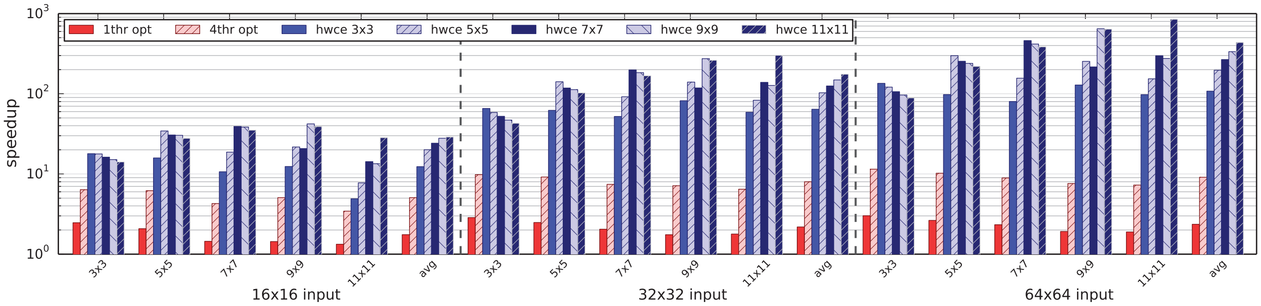

Figure 5. Log-scale speedup over naive single-thread convolution on 16x16, 32x32 and 64x64 input images.

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 AXI4-Stream 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).

C. 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 I shows its main functions.

HAL function	explanation
ACQUIRE_BUSYWAIT	polls on the job lock register in the HWCE register file until it gets a new job id
SETUP_PARAMS	writes convolution parameters in HWCE regfile
TRIGGER	ends the offload by writing in the HWCE trigger register; also releases the job lock
WAIT_JOB	polls on the running job id register in the HWCE until it is equal to the given id

Table I. HWCE Hardware Abstraction Layer.

The HW support of the HWCE is limited to convolutions on images whose width is of the type $d_{\text{in}} = 2^n + K - 1$ ($d_{\text{out}} = 2^n$), where K is the kernel size. To support general convolutions, we observed that since any integer number is representable in base 2, any d_{out} can be split into a set of power-of-2 d_{out}'s. Equivalently, any d_{in} can be split into a set of (partially overlapping) d_{in}'s supported by the HWCE. Therefore, it is possible to split a convolution on any image in a set of offloads using a simple greedy algorithm: always offload a convolution on the biggest set of columns possible.

IV. Results

A. HWCE synthesis results

We synthesized the HWCE in STMicroelectronics 28nm UTB FD-SOI technology using Synopsys Design Compiler. We backannotated switching activity using MentorGraphics ModelSim for RTL simulation and estimated dynamic and leakage power consumption at five operating points(reported in Table II) 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 [32]. Execution time is derived from full-platform RTL simulations of the HWCE-augmented shared-memory cluster.

Table III reports the relative area of the 16-bit HWCE with respect to an OpenRISC core (I$ excluded). The linebuffer parameter d_{LB} is sweeped 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 linebuffer increases linearly with both d_{LB} and K, while the number of multipliers is $2K^2$.

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

Table II. Operating points for the heterogeneous cluster.

Area		Filter size				
		3x3	*5x5*	*7x7*	*9x9*	*11x11*
d_{LB}	*32*	1.52	2.61	4.14	6.28	8.63
	64	1.68	2.92	4.62	6.82	9.42
	128	2.00	3.55	5.56	8.06	11.00

Table III. Relative area of 16-bit HWCE vs OpenRISC.

The design is pipelined in such a way that it is never a frequency bottleneck for the rest of the cluster; in fact, while the nominal frequency for the cluster is that shown in Table II, the HWCE is always synthesizable at a higher frequency.

B. Convolve-accumulate performance

To assess speedup over SW convolution and estimate the energy-efficiency boost, we developed a microbenchmark focusing on a single 3x3, 5x5, 7x7, 9x9 or 11x11 convolution-accumulation (as defined in Equation 1) over a 16x16, 32x32 or 64x64 image. We developed two SW implementations of convolution-accumulation for each filter size: the `naive` implementation directly follows the definition in Equation 1 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 III-C.

Figure 5 shows the speedup over a single-thread naive implementation with the optimized SW and using HWCEs of different sizes to implement 3x3, 5x5, 7x7, 9x9 and 11x11 filters. All HWCEs have the linebuffer 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 16x16 to 64x64, we ovserve 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 NxN convolution is using a HWCE configured with a NxN 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 11x11 convolution on a 64x64 image can be performed with a 3x3 HWCE with 16 distinct offloads,

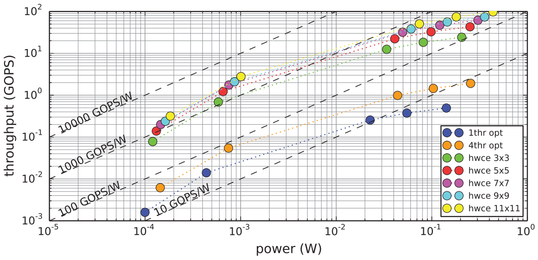

Figure 6. Energy efficiency (average throughput in equivalent GOPS vs total platform power consumption).

and a performance loss of 8x with respect to the 11x11 HWCE (but is still 14x faster than SW). On average, the speedup on the biggest image ranges between 40x and 160x compared to the the fastest single-thread SW implementation (10x-40x over the multi-thread one); on the smallest image it is reduced to 7x-18x (2x-6x over the multi-thread one).

In the experiment reported in Table IV, we repeated 10 times the greedy offload of a 5x5 convolution on a 32x32 image, using a 5x5 HWCE and we annotated the average execution time in clock cycles. We also report two ideal conditions: *i*) no memory stalls due to contention, and *ii*) single-shot offload instead of greedy offload (this requires modifying the HWCE to directly support a 32px wide input).

	idle	*first config*	*preload*	*run*	***total***
real	45	258	97	588	**730**
no stalls *efficiency*	- -	- -	89 91.75%	560 95.24%	**649** **88.90%**
no stalls, no offload *efficiency*	- -	- -	67 69.07%	448 76.19%	**515** **70.55%**

Table IV. HWCE OVERHEADS.

Each convolution is composed by three consecutive offloads to the HWCE (on a 16x28, 8x28, 4x28 output image respectively) and thanks to the availability of a queue of jobs, the configuration overhead is payed only on the first. Though the multiple offloads are slower than the single one due to recomputation, the greedy offload algorithm is still convenient as it supports generic image sizes with no increase in HW size and complexity.

C. Convolve-accumulate power & energy

Energy-wise, our results show significant savings compared to SW solutions for convolutional workloads. In Figure 6, we show average energy efficiency of the SW convolutions and HWCEs benchmarked in Figure 5 in the case if a 64x64 input image, at the operating points of Table II. 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 6 is the total of the shared-memory cluster, keeping three cores off when the HWCE is on. Peak efficiency is 2.75 GOPS/mW considering equivalent RISC operations, reached with the 11x11 HWCE at the 0.4V operating point. By comparison, the peak efficiency of the SW implementation is 72 GOPS/W (∼38x 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 most performant configuration, total power is always kept under 1 W.

To contrast SW convolution on OpenRISC and the HWCE convolution, in Figure 7 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 ∼40x 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 64x64 image.

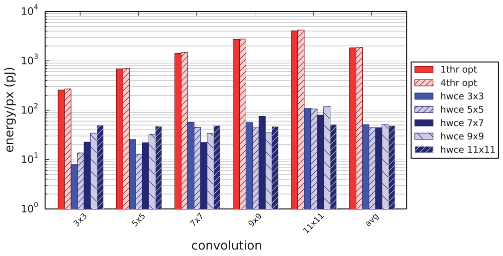

Figure 7. Energy spent per output pixel at 22 MHz (0.4V operating point).

		technology	*performance*	*energy eff.*
NeuFlow [12]		IBM 45nm	160 GMAC/s	230 GOPS/W
nn-X [14]		Xilinx Zynq	240 GOPS	25 GOPS/W
Qadeer et al. [9]		IBM 45nm	387 GOPS	889 GOPS/W
DianNao [18]		TSMC 65nm	452 GOPS	931 GOPS/W
Camunas-Mesa et al. [28]		350nm	16.6 Mevt/s	185 Mevt/s/W
IBM TrueNorth [27]		Samsung 28nm	-	46 GSOPS/W
Our work[2]	***@0.4V*** ***@0.8V***	**ST FDSOI 28nm**	**2.78 GOPS** **74 GOPS**	**2750 GOPS/W** **412 GOPS/W**

Table V. COMPARISON WITH STATE OF ART.

Pointwise, the 3x3 HWCE reaches the peak efficiency of 6.5 pJ/px in the 3x3 convolution.

Table V contrasts our results with those of state-of-art related works listed in Section II, showing that this work improves state of art in terms of energy efficiency in its most efficient point.

D. ConvNet benchmark

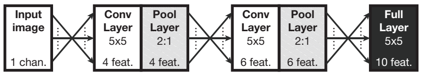

Figure 8. Convolutional network topology used for the ConvNet benchmark.

We used a complete CNN on our platform to further test the HWCE; its topology is shown in Figure 8. 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 32x32 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 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 3.86x and 3.41x for the single- and 4-thread implementations respectively. Figure 9 also shows that the combined action of SW parallelism and HW acceleration achieves an overall 12x 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 bigger CNNs, the impact of the HWCE is also likely to be higher.

[2] Note that GOPS/W rating for our work considers the full platform power, with post-synthesis back-annotation, running a fully functional workload (as opposed to the theoretical peak in Qadeer et al., which is based on the values reported in [9]). Peak performance and efficiency of the HWCE-augmented cluster are ∼3x better than the ones reported in Table V.

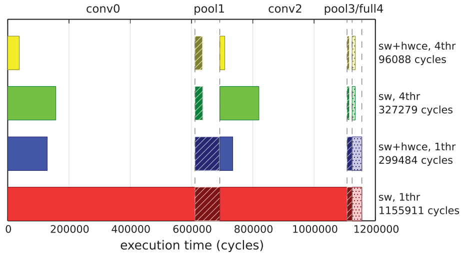

Figure 9. CNN benchmark execution time.

V. Conclusions

To support state-of-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. To this end, our main contributions are 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 results show that the HWCE-augmented PULP cluster reaches state-of-art convolution energy-efficiency: up to 2.76 GOPS/mW, spending 35 pJ/px on average in the most efficient configuration.

Coupling efficiency with flexibility is key to the success of specialized architectures [9]. 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-art accuracy [5][4][6]. Our future work will devote to support more BICV kernels, with the objective of delivering enough performance and energy efficiency to drive future embedded and wearable vision applications.

Acknowledgements

This work was funded by the EU ERC-Advanced Grant MULTITHERMAN (FP7-291125) and by the YINS RTD project (no. 20NA21 150939), evaluated by the Swiss NSF and funded by Nano-Tera.ch with Swiss Confederation financing. We also thank STMicroelectronics for granting access to the FDSOI 28nm technology libraries.

References

[1] 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.

[2] 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.), pp. 1097–1105, Curran Associates, Inc., 2012.

[3] 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*, pp. 580–587, 2014.

[4] 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.

[5] 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*, pp. 1725–1732, 2014.

[6] 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.

[7] M. Gautschi, D. Rossi, and L. Benini, "Customizing an open source processor to fit in an ultra-low power cluster with a shared L1 memory," in *Proceedings of the 24th edition of the Great Lakes Symposium on VLSI - GLSVLSI '14*, (New York, New York, USA), pp. 87–88, ACM Press, 2014.

[8] 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 *Proceedings of the 2014 IEEE International Workshop on Signal Processing Systems*, 2014.

[9] 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), p. 24, ACM Press, 2013.

[10] 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.

[11] 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*, pp. 813–818, IEEE, Jan. 2012.

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

[13] 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*, pp. 109–116, IEEE, June 2011.

[14] 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*.

[15] D. Moloney, "1TOPS/W software programmable media processor," in *HotChips HC23*, (Stanford), 2011.

[16] Z. Lin, J. Sankaran, and T. Flanagan, "Empowering automotive vision with TIs Vision AccelerationPac," *TI White Paper*, 2013.

[17] 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.

[18] 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 the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 269–284, ACM, 2014.

[19] 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.

[20] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "CHARM," in *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design - ISLPED '12*, (New York, New York, USA), p. 379, ACM Press, 2012.

[21] 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), p. 205, ACM Press, 2010.

[22] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, "QsCores," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-44 '11*, (New York, New York, USA), p. 163, ACM Press, 2011.

[23] 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, pp. 86–95, Mar. 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] 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)*, pp. 983–987, IEEE, Mar. 2012.

[26] 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 Neurosynaptic Cores," 2011.

[27] 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, pp. 668–673, Aug. 2014.

[28] 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, pp. 504–517, Feb. 2012.

[29] A. Rahimi, I. Loi, M. R. Kakoee, and L. Benini, "A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters," *2011 Design, Automation & Test in Europe*, pp. 1–6, Mar. 2011.

[30] 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), pp. 1–10, ACM Press, 2014.

[31] 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, pp. 299–308, Sept. 1999.

[32] A. Teman, D. Rossi, P. Menierzhagen, L. Benini, and A. Burg, "Controlled Placement of Standard Cell Memory Arrays for Improved Density and Low Power in 28nm FD-SOI," in *Proceedings of the 20th Asia and South Pacific Design Automation Conference (ASP-DAC 2015)*, 2015.