# Fully Embedding Fast Convolutional Networks on Pixel Processor Arrays

Laurie Bose[1], Piotr Dudek[2], Jianing Chen[2], Stephen J. Carey[2], and Walterio W. Mayol-Cuevas[1]

[1] University of Bristol, Bristol, United Kingdom
{lb7943,Walterio.Mayol-Cuevas}@bristol.ac.uk
[2] University of Manchester, Manchester, United Kingdom
{pdudek,jianing.chen,stephen.carey}@manchester.ac.uk

**Abstract.** We present a novel method of CNN inference for pixel processor array (PPA) vision sensors, designed to take advantage of their massive parallelism and analog compute capabilities. PPA sensors consist of an array of processing elements (PEs), with each PE capable of light capture, data storage and computation, allowing various computer vision processes to be executed directly upon the sensor device. The key idea behind our approach is storing network weights "in-pixel" within the PEs of the PPA sensor itself to allow various computations, such as multiple different image convolutions, to be carried out in parallel. Our approach can perform convolutional layers, max pooling, ReLu, and a final fully connected layer entirely upon the PPA sensor, while leaving no untapped computational resources. This is in contrast to previous works that only use a sensor-level processing to sequentially compute image convolutions, and must transfer data to an external digital processor to complete the computation. We demonstrate our approach on the SCAMP-5 vision system, performing inference in a MNIST digit classification network at over 3000 frames per second and over 93% classification accuracy. This is the first work demonstrating CNN inference conducted entirely upon a PPA vision sensor, requiring no external processing.

**Keywords:** Low-level Vision, PPA, CNN, vision sensor, edge computing

## 1 Introduction

Recently, there has been much interest in developing hardware architectures for acceleration of deep learning algorithms. In particular, as Convolutional Neural Networks (CNNs) have become a staple of computer vision applications, there have been many approaches to implementing these efficiently in hardware [9],[15],[1],[7],[13],[17]. Some of the most challenging application scenarios involve "edge computing" or "on-device computing", where computations are carried out as close to sensors as possible, to achieve low power operation and minimise bandwidth of downstream communications. Ultimately, the sensing and processing can be integrated in a single device. One approach to such integration is through distribution of photosensors of the image sensor within a

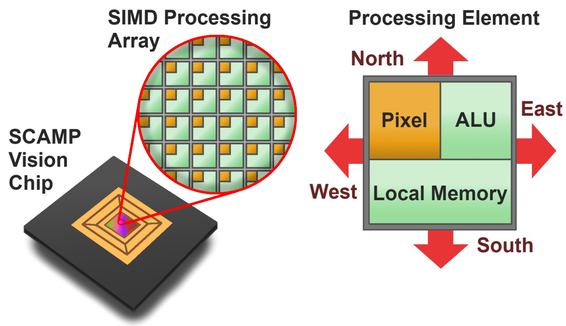


**Fig. 1.** A Pixel Processor Array device performs computations on the image sensor chip, using a SIMD processor array, with each pixel containing arithmetic logic unit (ALU), local memory circuits, and nearest-neighbour communication links.

massively-parallel fine-grain SIMD cellular processor array [4],[12],[14], an approach we term Pixel Processor Array (PPA). The PPA concept is illustrated in Figure 1.

In areas of computer vision and robotics applications, PPA sensors may potentially offer a wealth of benefits over standard camera sensors that are primarily developed with the human viewer in mind, and designed to capture entire high fidelity images for later inspection. The complete image capture, read-out, analog-digital conversion and transfer process in standard sensors introduces a significant time and energy bottleneck in computer vision pipelines, and typically results in low temporal resolution visual information (e.g. typical video-rate of 30 frames per second) that is highly prone to motion blur. A PPA sensor circumvents this scenario by instead performing visual computation directly at the point of light capture, extracting the desired information on-sensor, before transferring it over to a host processor. In many situations this can result in a vast decrease in data bandwidth between the sensor and the external hardware, allowing the system to conduct visual processing at much higher frame-rates, well beyond the capabilities of more standard sensors, while maintaining a low power consumption [4],[2].

One application of such PPA sensors is that of neural network inference in which captured visual information is immediately fed through a neural network being executed wholly or partially on-sensor, with the PPA's output then being compressed to simply neuron activations, ideally of the network's final layer. Such an application of future PPA sensors may offer real world network inference at speeds well beyond standard visual pipelines, however implementation on current PPA hardware is a highly challenging area of investigation.

As an emerging area of research, there exist only a small number of prior works in this area, as discussed in Section 3. These approaches [3],[16],[10] suffer from a number of limitations, such as having to perform image convolutions sequentially, requiring certain computation to be performed on external hardware, and only utilizing a small area of the entire processor array. The work presented in this paper aims to address these issues. Our main contribution is a new approach for structuring the execution of CNN network inference on PPA architectures. The key idea behind our approach is the concept of embedding network weights into the "pixels" of the PPA's processor array. This is done by storing weights within the processing elements (PEs) of the array, rather than weights being contained in the instructions transmitted to the processor array during inference as in previous works. This embedding of weights allows different parts of the processor array to perform different computations, upon different local data, simultaneously. As such, our approach can perform many different image convolutions, upon multiple images, spread across the PPA array in parallel, and efficiently perform a final fully connected layer entirely on-sensor. This computation can be structured to make use of the entire processor array at all times, improving the utilisation of available computational resources. To the best of our knowledge this is the first work to present such an approach, and the first to demonstrate multiple convolutional layers, a fully connected layer, and complete network inference upon a PPA. We demonstrate inference of both 2 and 3 layer networks upon the SCAMP-5 PPA performing digit recognition, able to achieve classifications at over 3000 frames per second and over 93% accuracy.

## 2 SCAMP-5 Overview

The PPA used for this work is the SCAMP-5 vision sensor [4],[6] consisting of a $256 \times 256$ array of processing elements (PEs), each containing processor circuitry allowing visual data to be stored, and manipulated directly at the point of light capture. The chip architecture, as shown in Figure 2, has been described in [4]. Briefly, each PE contains 13 digital registers (1-Bit) and 7 analog memory registers. Various operations can be performed between the memory registers of a PE, such as addition and subtraction of analog registers, and standard Boolean logic operations between digital registers. PEs can also exchange data with their neighbours. The array operates as an SIMD computer. The operations on local memory registers are performed across all PEs of the $256 \times 256$ array in parallel, using a single instruction. Each PE also contains an Execution Flag register allowing it to ignore received operations and allowing for conditional execution.

The operations performed by the PE array are dictated by a central controller, built upon ARM Cortex M0 processor. This controller executes its own program, primarily for sending instructions to the SCAMP-5 PPA to perform the sequence of operations that will result in some desired computation being performed upon the array.

The near-sensor processing approach of this architecture is very efficient. The SCAMP-5 chip performs up to 535 GOPS/W (Giga Operations Per Second per watt). Note that this device is manufactured using two decades old 180nm CMOS silicon technology [4]. Very significant gains can clearly be made on future devices in terms of increasing computing power and decreasing power consumption.

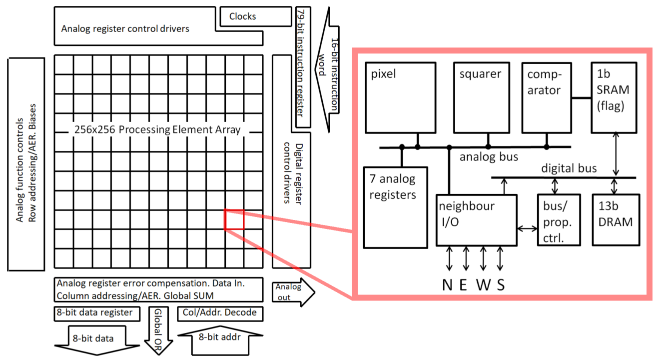


**Fig. 2.** SCAMP-5 Architecture [4]. Left: The PPA chip contains a 256x256 SIMD processor array and associated control, readout and interface peripheral circuits. Right: The processing element shown contains the photosensor, seven analog local registers, supporting arithmetic operations of addition, negation and division, neighbour communications with 4 nearest neighbour, 1 bit activity flag, and 13 bits of digital memory supporting logic operations.

## 3 Related Work

While previous works exist regarding CNN inference on PPAs [3],[16],[10] typically these methods perform various parts of the network computation in serial, rely on external hardware for additional computation, and only make use of a small area of the PPA's processor array leaving a great amount of processing power untapped. For example, these approaches are demonstrated upon MNIST/digit classification task on SCAMP-5 in which they load a single small MNIST digit ($28 \times 28$) into the center of the the $256 \times 256$ SIMD processor array. These approaches then sequentially compute image convolutions upon this central digit, effectively leaving well over 90% of the processor array unused. These convolution results are passed to the ARM controller connected to the SCAMP-5 PPA, which is used to perform one or more fully connected layers. Therefore, while demonstrating the concept of on-sensor CNN, a significant portion of the neural network computation in these approaches is actually conducted upon the ARM controller in a standard C++ program rather than by making use of the PPA's processing power.

By comparison, the approach proposed in this paper performs complete inference computation, including the fully-connected layer, upon the PPA device, potentially utilizing 100% of the processing array, and efficiently performing convolutional layers by computing many different image convolutions in parallel.

The proposed approach requires all network weights to be stored upon the processing elements (PEs) of the PPA itself. Due to the limited memory (13 bits, 7 analog values) of each PE on current generation PPA hardware, we are restricted to low-bit quantised weights and a limited number of layers. However it should be noted that many tasks have been successfully demonstrated on such

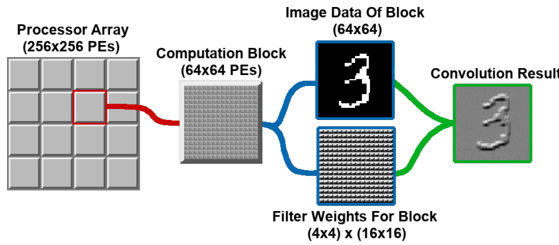


**Fig. 3.** Computational layout of the processing array for a convolutional layer. The array is split into computation blocks, each containing a set of filter weights (duplicated many times within a block) and the image to which the filter will be applied. A single SIMD routine can then be executed to apply each block's filter to its image data in parallel.

low-bit weight networks [19][11][18], and it is likely that next generation PPA hardware will see a significant boost in memory per PE.

## 4 Parallel Convolutional Layer Computation

In this section we describe our approach for the computation of convolutional layers upon the PPA. The weights of all the various convolutional filters are stored upon the processing array simultaneously, within the registers of the PEs. This enables different convolutional filters to be applied to different areas of the PE array in parallel. This can allow us to perform all the computation required for a convolutional layer in parallel.

For example, in the case of SCAMP-5, up to 64 MNIST digits can be spread across the $256 \times 256$ PE array. This allows for 64 different convolutions to be performed simultaneously at no additional time or power cost. In the case of digit classification this can be used to compute 64 different convolutions on the same digit duplicated 64 times in parallel.

### 4.1 Computational Layout on PE Array

Our convolutional layer approach effectively divides the PE array into multiple rectangular "computation" blocks of processing elements. The PEs of each computation block contain both the weights of a specific convolutional filter and image data to which the filter should be applied as shown in Figure 3. A sequence of SIMD operations can then be formulated to simultaneously apply each computational block's filter to its stored image data, performing all computation required for a convolutional layer. Examples of such computation are illustrated in Figure 4 for MNIST digits.

Note this approach is flexible in that each computational block may contain different image data and vary in size, however, for convolutional layer computation we use identical square blocks. For digit recognition we demonstrate convolutional layers of both 64 and 16 convolutions, using computational blocks of

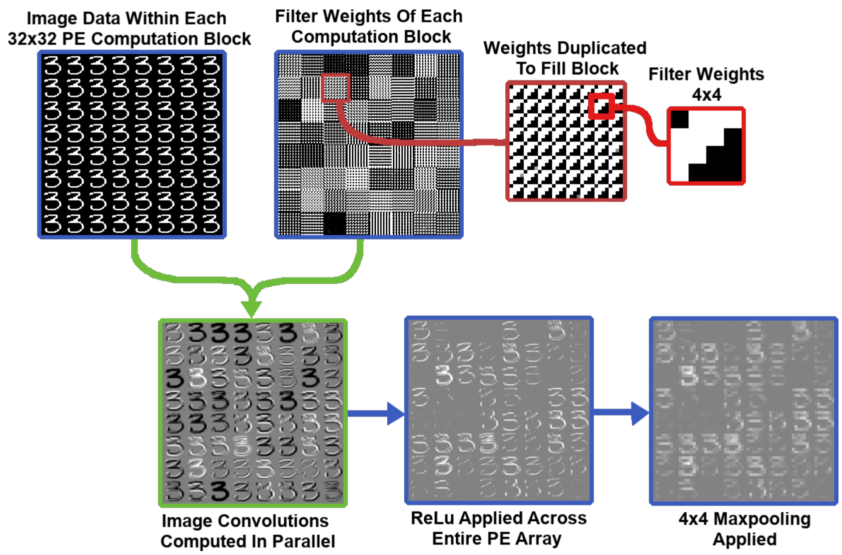


**Fig. 4.** Examples of Convolutional layer computation, illustrating how multiple convolutions are computed in parallel by splitting the PPA array into distinct computation blocks. In this case the Scamp5's processing array is split into 64 computational blocks of $32 \times 32$ PEs each. Upon computing image convolutions ReLu and Maxpooling can also be applied across the array at very little computational cost.

size 32 and 64 respectively. In both cases, the $28 \times 28$ MNIST digits are rescaled to fill these computation blocks.

### 4.2 In-Pixel Filter Weights

Each computation block stores within it the weights of a specific filter. When the SIMD routine for a convolutional layer is sent to the processor array, each block will use these weights to compute a convolution upon its stored image data. Directly storing filter weights upon the processor array at the locations where they are to be applied is what allows our approach to perform multiple filters simultaneously.

There are many possible layouts for storing a set of filter weights within a computational block of PEs. However, it is generally not possible for each PE to store a complete copy of its block's filter weights due to the limited local memory resources available on current generation PPA devices. The solution is to spread the storage of a computational block's filter weights across multiple PEs. This means each PE no longer has immediate access to every filter weight, however, weights can be copied over from other nearby PEs of the same computational block during convolution computation. To minimize the time transferring filter weights between PEs, its important to use a layout in which each PE is located in close proximity to other PEs storing the weights it will require during computation. This prompted a ”checker board” style layout, where multiple copies of convolutional filter weights are stored within each computational block to ensure each PE is located within a reasonable distance from each filter weight. This concept is illustrated for $4 \times 4$ filters in the right of Figure 4. Future PPA devices, with greater resources per PE, should allow each PE to store its own dedicated copy of any filter weights, significantly speeding up the convolution computation.

In our demonstrated networks each PE in a block stores a single binary filter weight, with the weight values of $+1$ and $-1$ naturally corresponding to image addition and subtraction operations. There are many schemes that could be used to store and apply higher bit-count weights but for now we leave this to future work.

### 4.3 Parallel ReLU and Max Pooling

After performing a convolutional layer the PE array will hold multiple convolution images such as those shown in Figure 4. We then turn these images into activation data by applying the ReLU activation function. The SCAMP-5 hardware has the function to flag all PEs whose stored values in a certain analog register are positive or negative. This allows us to simply flag all PEs whose convolution result is negative and input a value of zero into these flagged registers, generating ReLu activations.

We then perform a $4 \times 4$ max pooling routine by first making a copy of the activation values image. This copied image is then shifted horizontally right, with each PE then containing both its original activation value and a value from this shifted data. In parallel, every PE then compares these two activation values, replacing the stored activation data with the shifted data whenever it is greater in value. This routine of shifting, comparing activations and replacing with the higher value is repeated horizontally right three times,and three times vertically down. This results in every PE holding the highest activation value in the $4 \times 4$ square of which it is in the top left corner. The pixels holding the correct max-pooled values for each $4 \times 4$ grid space are then copied back into each PE of their $4 \times 4$ block.

### 4.4 Further Convolutional Layers

After performing an initial convolutional layer, either a final fully connected layer, or an additional convolutional layer is performed. This section describes one possible method to compute such an additional convolution layer, where each feature map is constructed from those of the previous layer as standard. Note that this approach could in future be used to add multiple additional convolutional layers, however this is difficult to achieve within the limited memory resources of current SCAMP-5 hardware.

In brief, the feature maps of a previous convolution layer (consisting of max-pooled activation data) are shrunk and duplicated to fill the processor array. Each duplicate of a feature map is then used in computing a feature map in

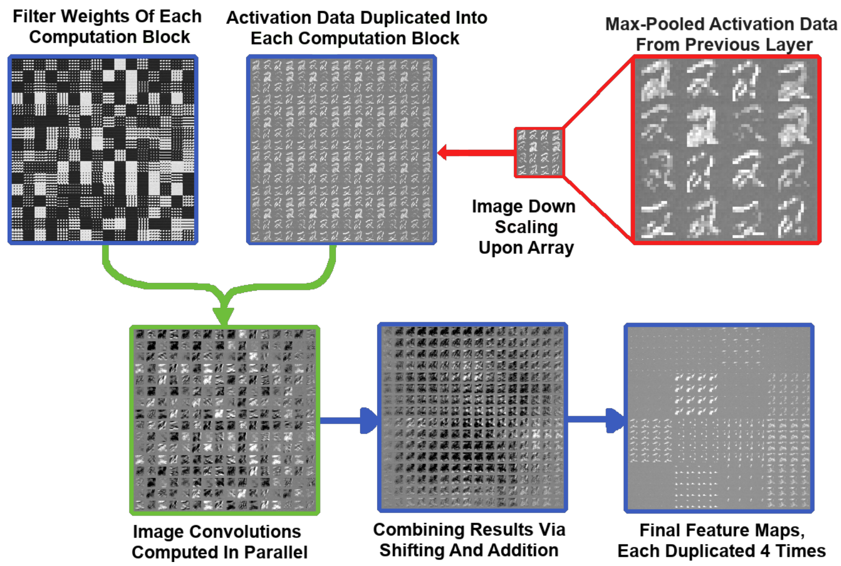


**Fig. 5.** Computation of feature maps from an additional convolutional Layer. Max-pooled activation data from the previous layer (top right) is shrunk and duplicated across the processor array. Image convolutions for the new layer are them performed upon each duplicated image, in the same manner as network's initial convolution layer. The resulting convolutional images (bottom left) are then combined accordingly and ReLu is applied forming feature maps of this new layer (bottom right). In this example 16 feature maps from the initial layer are duplicated and 256 convolutions computed, before being recombined to form 16 feature maps.

the new convolutional layer. An example of this is shown in Figure 5, where 256 convolutions are computed in parallel upon the 16 feature maps (each duplicated 16 times) of the previous layer. These convolution results are then added together accordingly to form the 16 feature maps of the new convolutional layer.

Many of the concepts introduced previously are re-used for this computation. The in-pixel storage of filter weights and computational layout is identical to the initial convolutional layer, with the processor array again being split into computation blocks each storing its own set of weights and image data as shown in Figure 5. The same SIMD routine used to perform the initial convolutional layer can simply be executed again for computing this additional layer, helping to reduce program size.

The resulting convolution results are then repeatedly shifted and added together, iteratively accumulating feature maps of this new convolutional layer. These feature maps can then be duplicated across the array, correctly positioning their activation data to be aligned with the weights of any following fully connected layer, so that parallel multiplication between activations and fully connected weights can be performed as described in Section 5.

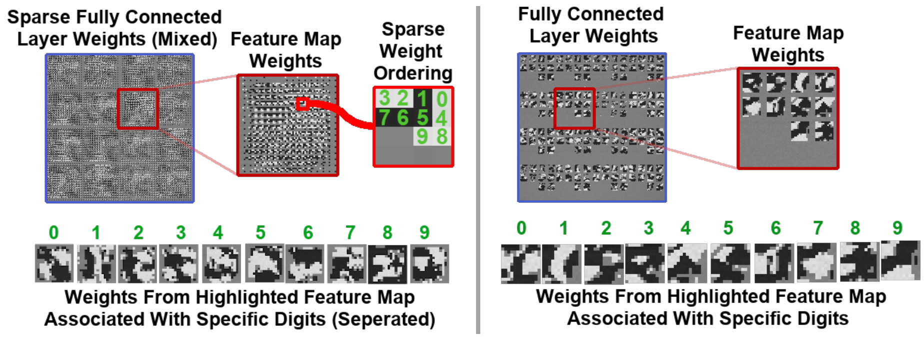


**Fig. 6.** Two examples of ternary fully connected weights stored upon the PPA's processor array in analog memory, both connected to 16 feature maps from a previous convolution layer. Left: weights for connecting to $(4 \times 4)$ max-pooled activation data stored in a sparse checkerboard like layout, with weights for the different digits mixed/interweaved with one another. Right: weights for connecting to duplicated feature maps.

### 4.5 Feature Map Shrinking and Duplication

The process of shrinking the max-pooled activation data upon the PPA leverages the image transformation methods first introduced in [2] for image scaling. However, conducting such scaling operations using analog memory registers results in the build-up of systematic errors and noise [4], from analog data having to be repeatedly copied from one PE to the next. This would corrupt the activation data beyond use. To avoid this issue we instead convert the analog activation data to a 3-bit digital representation, with each PE's stored analog value being split across 3 digital registers (within the same PE). This creates 3 binary images, one for each bit, which then can then all be scaled and duplicated across the array. Afterwards this digital data can be recombined to once again form a single gray-scale analog image, but devoid of corruption.

## 5 Parallel Fully Connected Layer Computation

Following on from a convolutional layers, we perform computation of a final ternary weight fully connected layer upon the PPA, again storing weights directly in the PEs of the processor array. The activations of the previous convolution layer are duplicated as shown in Figure 7, either by max-pooling (which creates blocks of duplicated values) or by duplicating the feature maps multiple times across the array. By correctly arranging the layout of the fully connected weights, each weight's PE can then directly receive the activation data associated with that weight. This layout varies as illustrated in Figure 7 depending on whether the previous layer produces max-pooled data or duplicated feature maps. All duplicated activations from the previous layer can then be multiplied by their associated fully connected weights simultaneously in parallel, using the native

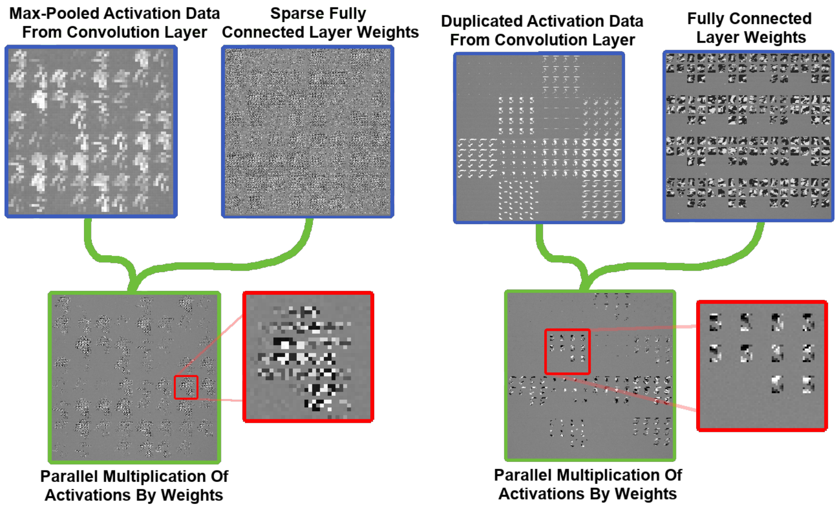


**Fig. 7.** Layout and application of fully connected layer weights, accepting as input either max-pooled activation data (Left) or duplicated feature maps (right). The arrangement of weights is different in each case to enable the correct parallel in-line multiplication between the weights and activation data. After this multiplication, all resulting data associated with a specific fully connected neuron can be summed in parallel by flagging the appropriate PEs.

analog image addition (for weights of value 1) and subtraction (weights of value -1) operations of SCAMP-5 (or no operation for weights of value 0). Examples of this process are shown in Figure 7.

The limited memory of the processing elements on SCAMP-5 restricts the programmer to the use of ternary weights for the final fully connected layer, stored within the analog registers of the PEs to save digital resources. Note that the content of analog registers decays over time, drifting away from the stored value [4]. However, with quantized ternary values being stored one can "Refresh" the register's content at set intervals to prevent such decay.

### 5.1 Activation Value Summation

After the multiplication step each PE contains a synaptic contribution to the activation of one of the final neurons in the fully connected layer. SCAMP-5 has the capability of performing a global summation of many analog values distributed across the PE array in parallel, which can be used to effectively add all synaptic contributions, however this summation introduces significant noise.

For two layer networks this does not pose a major issue and analog summation can be simply be performed a number times and averaged. However for three layer networks, with more discriminative second layer features, analog summation proves too noisy for accurate classification.

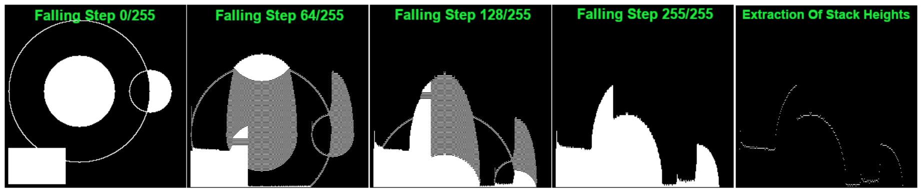


**Fig. 8.** Example of our proposed method for rapid binary image summation being iteratively performed upon a test image. With each iteration the "1s" of the image are made to fall forming stacks at the bottom of the image. All but the tops of these stacks are then eliminated allowing the heights of each to be read directly.

For such three layer networks we instead turned to using the PPA's digital computation, creating a new method to rapidly count the "set" white pixels ("1s") in a binary image. This method, as visualised in Figure 8, functions by essentially stacking pixels together on one side of the array. This process can be efficiently implemented upon the PPA's parallel architecture, performing 255 iterations of simultaneously shifting and stacking pixels. A simple shift copy and XOR can be used to eliminate all but the top pixels of each of these stacks. The image coordinates of these remaining pixels (up to 256) can then be read directly (using an address-event readout scheme of SCAMP-5) to give the heights of each stack, which when added together give the total number of white pixels in the original image. This entire process takes $260\mu s$ to complete, and while slower than the analog global summation, it provides a perfectly accurate summation result. This method is employed in the fully connected layer of our demonstrated three layer network, converting the analog activations into multi-bit representations, which can then be summed.

## 6 Results

### 6.1 MNIST Network Training

We trained networks with a mixture of binary and ternary weights (for convolutional filters and fully connected weights respectively) using an approach similar as [8],whereby real-valued weights are stochastically quantized during every forward pass. The errors obtained from the forward pass are then used to update the real-valued weights using the standard error back propagation algorithm, resulting in these real values converging towards binary/ternary ones over the course of training.

### 6.2 Inference on SCAMP-5 Hardware

We evaluated our inference approach using both two and three layer networks, trained on MNIST classification. The two layer networks used $32 \times 32$ input images, and consisted of one convolutional layer (64 feature maps, $4 \times 4$ filters),

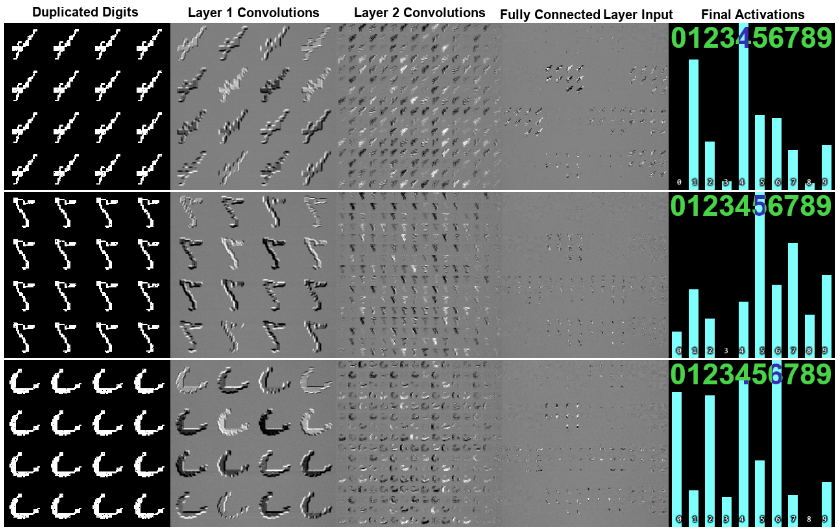


**Fig. 9.** Example classifications of some ambiguous digits via inference of a three layer network upon SCAMP-5. The convolutions computed from both layers are shown, along with the activations multiplied by the fully connected layer weights, and the final neuron activations for digits 0-9.

max pooling ($4 \times 4$), and a final fully connected layer. Three layer networks used up scaled $64 \times 64$ input images, a first convolutional layer (16 feature maps), max pooling ($4 \times 4$), a second convolutional layer (also 16 feature maps), and a final fully connected layer. Some sample classifications of such networks are shown in Figures 9 and 10. Training is performed on a standard PC, using the 60,000 samples dataset. The trained weights were then loaded into the PEs of the SCAMP-5's processor array as described in previous sections, and evaluation of inference was performed by directly loading the test set images (one at a time) onto the PE array. With each image the SCAMP-5 then executed the SIMD routines to compute the network layers and output a final classification. Table 1 shows the computation times of the various processes used during inference.

The total computation time of two layer networks was 272 microseconds corresponding to the processing speed of 3676 classifications per second (excluding the time to load a testing set image to the array). In a real-life scenario, digit images are not loaded to the array but captured via the image sensing capabilities of the chip, as demonstrated in [3]. The inference accuracy of tested two layer networks varied around 92%-94% classification accuracy with different networks, a reduction from accuracy levels around 95% obtained in training on PC. It is worth noting that due to the nature of the analog computing used during inference, which exhibits noise and systematic errors varying from one device to another [4], such a drop in accuracy is not unexpected.

Three layer networks had a total computation time of 4.46 milliseconds (giving 224 classifications per second), the majority of which was from the shrinking

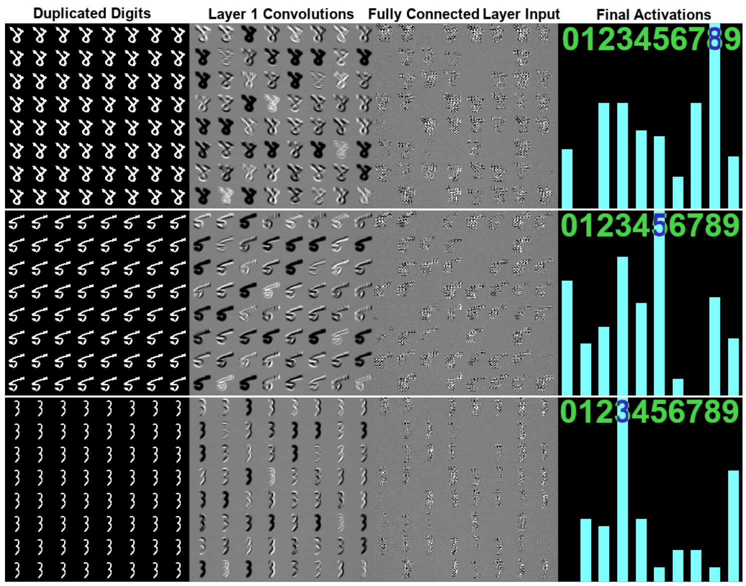


**Fig. 10.** Examples of digit classification via inference of a two layer network on SCAMP-5. showing both the 64 convolutions computed, parallel multiplication of fully connected weights and the activation values of the final fully connected layer's neurons.

of activation data between convolution layers, and the merging of convolutions to form feature maps. The methods and SIMD routines for these components are not as optimized as those for layer computation, and this is something to improve in future works. The classification accuracy obtained for three layer network inference was also in the range of 92%-94%, but with a more significant drop from the accuracy of 97% obtained in training.

It may be possible to reduce these discrepancies between training and inference accuracy in future work by investigating analog errors [5] and modelling them within the training process, using a hardware-in-the-loop training approach performing forward pass directly on SCAMP hardware, or by shifting certain operations to use digital computation upon the PPA. That said, the PPAs massively parallel analog computing still results in high performance and efficiency. During inference vision sensor itself consumes 1.25 W, with the rest of the current camera system contributing another 750 mW, when operating at the maximum throughput of 3676 classifications per second. It can be extrapolated, that for applications where frame rates in the range of 30 fps are acceptable, the operating power of the system executing a two-layer network model would be in the range of 10-20 mW.

Our approach also compares favorably to existing CNN works using the SCAMP-5 PPA. Against the digital CNN implementation of [3], our presented 2-layer network offers similar accuracy while being capable of frame-rates 10

| Component | Two Layer Network | Three Layer Network |
|---|---|---|
| Digit Duplication | $28\mu s$ | $28\mu s$ |
| Convolutional Layer/s | $160\mu s$ | $320\mu s$ (160 × 2) |
| ReLu | <$1\mu s$ | <$1\mu s$ |
| Max Pooling | $25\mu s$ | $25\mu s$ |
| Feature Map Shrink and Duplicate | - | $2095\mu s$ |
| Feature Map Creation | - | $1055\mu s$ |
| Fully Connected Layer | $59\mu s$ | $901\mu s$ |
| **Total** | $272\mu s$ (3676 fps) | $4464\mu s$ (224 fps) |

**Table 1.** Computation times of various network components.

times higher ($\tilde{2}00$ vs 3000+ FPS). Against the works of [10],[16] our approach offers similar speed and accuracy, however a key distinction is that these works make heavy use of traditional computation on the ARM Cortex controller connected to the array. Specifically these works compute 2 fully connected layers on the ARM controller, with then only a single convolutional layer of 3 convolutions computed upon the PPA array. This is in contrast to this work in which every layer is computed directly upon the PPA array, with our weights "in-pixel" approach allowing the entire array to be utilized. As such, we argue that this approach is far better suited to make use of any future PPA devices, as it can immediately make full use of any hardware improvements such as greater memory per PE or increased array size.

## 7 Conclusions

We have presented a novel approach for conducting CNN inference upon PPA hardware, exploiting analog computations, and storing the weights of the network directly within the array's processor elements rather than in the program running upon the array's controller chip. Unlike previous works, our approach can perform multiple convolution layers, and a final fully connected layer entirely upon the PE array of the device, with the only information read-out being the activations of the final neuron layer. Thus we demonstrate a complete visual CNN on-chip solution, from light sensing, to classification results.

Our experiments considered only small networks using binary filters and ternary fully connected weights. However our approach can be used for deeper more complex networks as PPA hardware improves, and even these smaller networks have found practical applications in edge computing devices. Our inference of digit classification networks on the SCAMP-5 PPA sensor performs at over 93% classification accuracy and at speeds exceeding 3,000 image classifications per second, for the first time executing a complete classification network entirely upon the "focal-plane" of an image sensor. We expect that new PPA hardware utilizing more recent silicon technologies will provide substantial gains in performance and efficiency which our approach is well positioned to make use of. We hope this work also motivates better architectures designed for visual perception.

## References


1. Aimar, A., Mostafa, H., Calabrese, E., Rios-Navarro, A., Tapiador-Morales, R., Lungu, I.A., Milde, M.B., Corradi, F., Linares-Barranco, A., Liu, S.C., et al.: Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps. IEEE transactions on neural networks and learning systems (99), 1–13 (2018)
2. Bose, L., Chen, J., Carey, S.J., Dudek, P., Mayol-Cuevas, W.: Visual odometry for pixel processor arrays. In: Proceedings of the IEEE International Conference on Computer Vision. pp. 4604–4612 (2017)
3. Bose, L., Chen, J., Carey, S.J., Dudek, P., Mayol-Cuevas, W.: A camera that CNNs: Towards embedded neural networks on pixel processor arrays. arXiv preprint arXiv:1909.05647 (ICCV 2019 Accepted Submission) (2019)
4. Carey, S.J., Lopich, A., Barr, D.R., Wang, B., Dudek, P.: A 100,000 fps vision sensor with embedded 535GOPS/W 256× 256 SIMD processor array. In: 2013 Symposium on VLSI Circuits. pp. C182–C183. IEEE (2013)
5. Carey, S.J., Zarándy, Á., Dudek, P.: Characterization of processing errors on analog fully-programmable cellular sensor-processor arrays. In: 2014 IEEE International Symposium on Circuits and Systems (ISCAS). pp. 1580–1583. IEEE (2014)
6. Chen, J., Carey, S.J., Dudek, P.: Scamp5d vision system and development framework. In: Proceedings of the 12th International Conference on Distributed Smart Cameras. p. 23. ACM (2018)
7. Chen, Y.H., Emer, J., Sze, V.: Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In: ACM SIGARCH Computer Architecture News. vol. 44, pp. 367–379. IEEE Press (2016)
8. Courbariaux, M., Bengio, Y., David, J.P.: Binaryconnect: Training deep neural networks with binary weights during propagations. In: Advances in neural information processing systems. pp. 3123–3131 (2015)
9. Du, Z., Fasthuber, R., Chen, T., Ienne, P., Li, L., Luo, T., Feng, X., Chen, Y., Temam, O.: Shidiannao: Shifting vision processing closer to the sensor. In: ACM SIGARCH Computer Architecture News. vol. 43, pp. 92–104. ACM (2015)
10. Guillard, B.: Optimising convolutional neural networks for super fast inference on focal-plane sensor-processor arrays. Master's thesis, Imperial College London (2019)
11. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Quantized neural networks: Training neural networks with low precision weights and activations. The Journal of Machine Learning Research **18**(1), 6869–6898 (2017)
12. Komuro, T., Kagami, S., Ishikawa, M.: A dynamically reconfigurable SIMD processor for a vision chip. IEEE journal of solid-state circuits **39**(1), 265–268 (2004)
13. Liang, S., Yin, S., Liu, S., Luk, W., Wei, S.: Fp-bnn: Binarized neural network on fpga. Neurocomputing **275**, 1072–1086 (2018)
14. Rodriguez-Vazquez, A., Fernández-Berni, J., Leñero-Bardallo, J.A., Vornicu, I., Carmona-Galán, R.: CMOS vision sensors: embedding computer vision at imaging front-ends. IEEE Circuits and Systems Magazine **18**(2), 90–107 (2018)
15. Sim, J., Park, J.S., Kim, M., Bae, D., Choi, Y., Kim, L.S.: A 1.42 TOPS/W deep convolutional neural network recognition processor for intelligent IoE systems. In: 2016 IEEE International Solid-State Circuits Conference (ISSCC). pp. 264–265. IEEE (2016)
16. Wong, M.: Analog vision - neural network inference acceleration using analog SIMD computation in the focal plane. MSc Dissertation, Imperial College London (2018)

17. Zhao, R., Song, W., Zhang, W., Xing, T., Lin, J.H., Srivastava, M., Gupta, R., Zhang, Z.: Accelerating binarized convolutional neural networks with software-programmable fpgas. pp. 15–24 (02 2017)
18. Zhou, A., Yao, A., Guo, Y., Xu, L., Chen, Y.: Incremental network quantization: Towards lossless cnns with low-precision weights. arXiv preprint arXiv:1702.03044 (2017)
19. Zhu, C., Han, S., Mao, H., Dally, W.J.: Trained ternary quantization. arXiv preprint arXiv:1612.01064 (2016)