SP-CNN: A Scalable and Programmable CNN-Based Accelerator

In this article, the authors use a cellular neural network (CNN)—a neural computing paradigm that is well suited for image processing applications—as a specialized accelerator for mobile computing. They propose SP-CNN, an architecture and a multiplexing algorithm that provides scalability to CNN architectures. They demonstrate the proposed multiplexing algorithms over six image processing benchmarks and present a performance analysis of SP-CNN.

Dilan Manatunga
Hyesoon Kim
Saibal Mukhopadhyay
Georgia Institute of Technology

With the rapid growth of mobile computing, the need to design energy-efficient and special-purpose accelerators is high. For example, in Intel's Atom Lexington and Bay Trail systems on chip,[1] more than 25 percent of the chip area is used for video or image-related processors. Excluding memory or storage, the majority of the chip area is dedicated to special accelerators for image and graphics and security-related engines. Furthermore, considering the growing need for processing sensor data like camera input, the demand for special-purpose accelerators will continue to increase.

Recently, brain-inspired computing systems, especially neural-network-based systems, are receiving much attention as special accelerators. Qualcomm's Neural Processing Unit (NPU) systems and IBM's TrueNorth are two examples that promise high energy efficiency. Similar to these neural-network processors, which are based on artificial neural networks, are cellular neural network (CNN) processors.[2] CNNs have been studied widely for image-processing applications and are another option for neural-network-based accelerators.

One challenge of current CNN research is the scalability of CNN hardware and algorithms. So far, all CNN algorithms have been developed under the assumption that the number of cells is equal to the image size. Unfortunately, current CNN designs have relatively small array sizes, such as 80×60,[3] and have not scaled to the multiple megapixel images present today. Furthermore, handling large images is not a trivial task. Using simple naive multiplexing algorithms or simple stencil operations will quickly saturate memory bandwidth.

In this article, we propose a mechanism to operate CNN algorithms on large images while still using relatively small arrays. Our contributions are twofold. First, we propose a

Published by the IEEE Computer Society

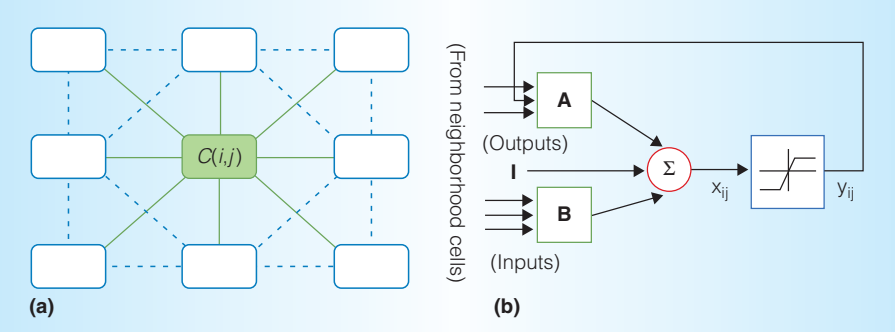

Figure 1. CNN cell connection. (a) Neighbor cells of the cell at $C(i, j)$. (b) Operation of one CNN cell. The figure shows the local connections for an example 3 × 3 network with radius of 1.

programmable CNN architecture, called SP-CNN, which is an accelerator to be connected with a host processor. SP-CNN has multiple CNN array processors and a scheduler. Second, we propose an algorithm to be used on our proposed architecture that can handle any input image size without causing heavy memory contention. We also demonstrate that the CNN paradigm can perform real-time image processing on modern image sizes.

CNN background

The CNN, which was introduced by Chua and Yang,[2] is a type of neural network that comprises a homogenous array of cells, in which each cell communicates with only a fixed set of neighbor cells. The array can be *n*-dimensional, but is typically 2-dimensional for most implementations. The connections between cells are local, which eliminates the need for long-distance interconnects. Specifically, a cell is connected to cells within a fixed radius of its position. These cells are part of what is called the cell's neighborhood. Figure 1 shows the local connections for an example 3 × 3 network with radius of 1.

Each cell in the array operates on the basis of a state and output equation. The original CNN design was based on analog circuits, so the cell's operation was specified by continuous time state and output equations. Most CNNs typically use the standard state and output equations. For our work, we focused on digital CNN implementations, so we chose to use the digital equivalent of the standard state and output equations (see Equations 1 and 2, respectively). In these equations, x represents the state, y represents the output, and A and B represent the weights for connections from the neighborhood cells. The CNN gene controls a CNN application and specifies the A and B scaling factors, the state equation threshold, I, the initial cell state, and boundary conditions. In the traditional CNN operation, as shown by the pseudocode in Algorithm 1 (see Figure 2), each cell iteratively computes its new state and output i values in parallel until the cells reach a steady state.

$$x_{ij}(t+1) = \Sigma_{C(k,l)\in N_r(i,j)} A(i,j;k,l) y_{kl}(t) + \Sigma_{C(k,l)\in N_r(i,j)} B(i,j;k,l) u_{kl}(t) + I \quad (1)$$

$$y_{ij}(t+1) = \frac{1}{2}\left[|x_{ij}(t)+1| - |x_{ij}(t)-1|\right] \quad (2)$$

Many applications and algorithms have been developed for CNNs. Because CNNs naturally have a 2D array representation, much of this work has involved image processing applications.[2,4] For example, Roska's research group developed a cellular wave computing library containing hundreds of CNN applications.[5] (For more information on other approaches, see the "Related Work in Cellular Neural Networks" sidebar.)

For most-developed CNN applications, there is an implicit assumption that the size of the input matches the size of the CNN array. For image processing, this would mean that the size of the CNN array would equal the size of the image. However, most hardware implementations have small sizes,

Related Work in Cellular Neural Networks

Several researchers have studied applying time multiplexing to the CNN.[1,2] However, these multiplexing mechanisms are similar to ideal multiplexing, and do not scale because of the communication overhead. On the contrary, SP-CNN provides a programmable interface to support such scalability.

Another direction to support large images with smaller-dimension CNN arrays is to use software algorithms. For example, the visual attention engine chip processes large images by first preprocessing images with high-level algorithms such as object recognitions. These types of mechanism can be used in addition to the scalability features provided by SP-CNN.

Researchers have also implemented CNN algorithms on GPUs.[3,4] Using the large parallel computation resources provided by GPUs, these implementations show a large speedup over CPUs as well as support for arbitrary image sizes. However, these implementations still tend to be many factors slower than a hardware CNN array.

Finally, although CNNs are not as computationally strong as other neural networks, such as spiking neural networks or convolutional neural networks, they have an advantage in terms of many prior hardware implementations and many software algorithms.[4] Furthermore, one of our research avenues has been investigating how to modify and generalize the SP-CNN design so that it can be used to model certain artificial neural networks, perceptrons, and even convolutional neural networks.

References

1. S. Lee et al., "24-GOPS 4.5-mm2 Digital Cellular Neural Network for Rapid Visual Attention in an Object-Recognition SoC," *IEEE Trans. Neural Networks*, vol. 22, no. 1, 2011, pp. 64–73.
2. L. Wang, J. de Gyvez, and E. Sanchez-Sinencio, "Time Multiplexed Color Image Processing based on a CNN with Cell-State Outputs," *IEEE Trans. Very Large Scale Integration Systems*, vol. 6, no. 2, 1998, pp. 314–322.
3. R. Dolan and G. DeSouza, "GPU-Based Simulation of Cellular Neural Networks for Image Processing," *Proc. Int'l Joint Conf. Neural Networks*, 2009, pp. 730–735.
4. S. Potluri et al., "CNN Based High Performance Computing For Real Time Image Processing on GPU," *Proc. Joint 3rd Int'l Workshop Nonlinear Dynamics and Synchronization and 16th Int'l Symp. Theoretical Electrical Eng.*, 2011, pp. 1–7.


```
while change do
        change = parallel-for-i,j of Eq. (1)
        parallel-for-i,j of Eq. (2)
        t = vt +=1
end while
```

Figure 2. Pseudocode of Algorithm 1, the traditional CNN algorithm. The algorithm shows that operation of the whole array is basically the parallel operation of each cell.

ranging from 3 × 3 to 80 × 60, and are therefore significantly smaller than the multiple-megapixel images seen today.[3]

Programmable multiplexing algorithm

To enable the efficient operation of CNN applications on large images, we propose a multiplexing algorithm based on how stencil computation is performed on GPUs. In GPU stencil computation, the input image is spatially partitioned, and each GPU computing unit performs the stencil operation on an image partition. If we applied this approach naively, we would run each image partition on a CNN array using the traditional CNN algorithm (Figure 2).

This naive multiplexing approach leads to errors for many CNN applications, because cells on the partition boundary do not operate correctly since they never see updated values for the neighboring cells in different partitions. Figure 3 shows an example of the incorrect outcomes of naive multiplexing for the hole-filling CNN application.

The easy solution to this issue would be to employ *ideal multiplexing*, in which we run the partition on the CNN array for only 1 CNN time unit, which is equivalent to one state or output computation. We do this for every partition and then repeat the process. This method will always produce the correct output. Figure 4a illustrates the general process for ideal multiplexing.

However, ideal multiplexing comes with a cost of high memory contention. Specifically, processing the partition on the CNN array for 1 CNN time unit takes relatively little time. On the other hand, transferring the

image partition, even one as small as 128 × 128, takes a comparatively much larger time. This leads to heavy memory contention from the continuous transfer of image partitions to and from the CNN computing units.

SP-CNN multiplexing algorithm

The key insight to the SP-CNN multiplexing algorithm is that many CNN applications are robust to small errors with input values and/or computation. Specifically, we realized that although it is important for the boundary cells to see updated values from neighboring cells in different partitions, for many CNN applications, the correct output could still be achieved even if the partition boundary cells did not have the correct neighbor values for a specified interval of time.

Algorithm 2 (see Figure 5) illustrates the pseudocode for the SP-CNN multiplexing algorithm. In essence, we take the ideal multiplexing approach, but instead of running one state or output computation, we run multiple cycles of the computations. We term this period the *interval.* At the end of each interval, SP-CNN saves the computed state output in a new array. Because we are saving this state value to a new array, the neighboring partitions will not yet see the newly computed values. Once every partition has been processed for one interval, the period which we call an *iteration,* we then repeat the process, except we now use the array with the newly calculated state values. This means that boundary cells will now see the updated values for their neighboring cells in neighboring partitions. These iterations are repeated until we converge to a steady state. Again, Figure 4 illustrates this process. Ideal multiplexing then becomes a special case of SP-CNN multiplexing, wherein the interval time equals 1 state or output computation.

After some preliminary analysis, we saw that certain partitions would converge to a steady state before their interval was over. This does not necessarily mean the entire image had converged, but it does indicate that no further useful computation would occur for the rest of the interval. To avoid this, we introduced the concept of Early-Finish. With Early-Finish, a partition stops

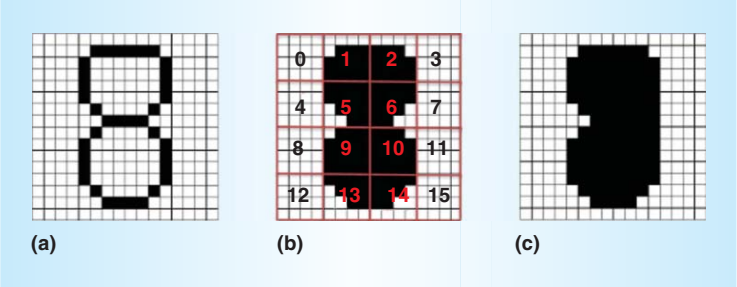

Figure 3. Example of hole-filling with naive multiplexing. (a) Input. (b) Correct output. (c) Naive multiplexing.

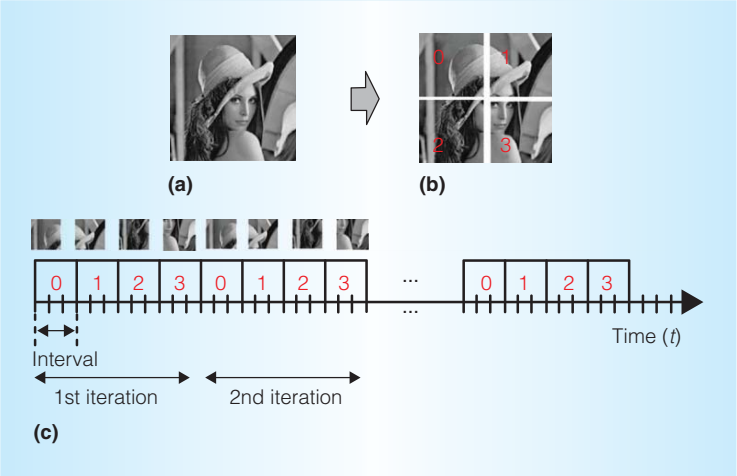

Figure 4. Example of CNN multiplexing. (a) Original image. (b) Partitioned images. (c) Illustration of multiplexing operation. SP-CNN operates one subimage at a time during one interval and it iteratively operates the entire image.

```
while change do
        change = false
        for p in partitions do
          loadStateAndInputToCNN(p)
          for n = 0; n < interval; n++, t++ do
            change l= parallel-for-i,j on p with Eq. (1)
            parallel-for-i,j on p of Eq. (2)
          end for
          saveToNextStateFromCNN(p)
        end for
        swapStateAndNextStatePointers(p)
        iter += 1, vt += interval
end while
```

Figure 5. Pseudocode of Algorithm 2, the SP-CNN multiplexing algorithm. The operations are applied to all of the CNN cells.

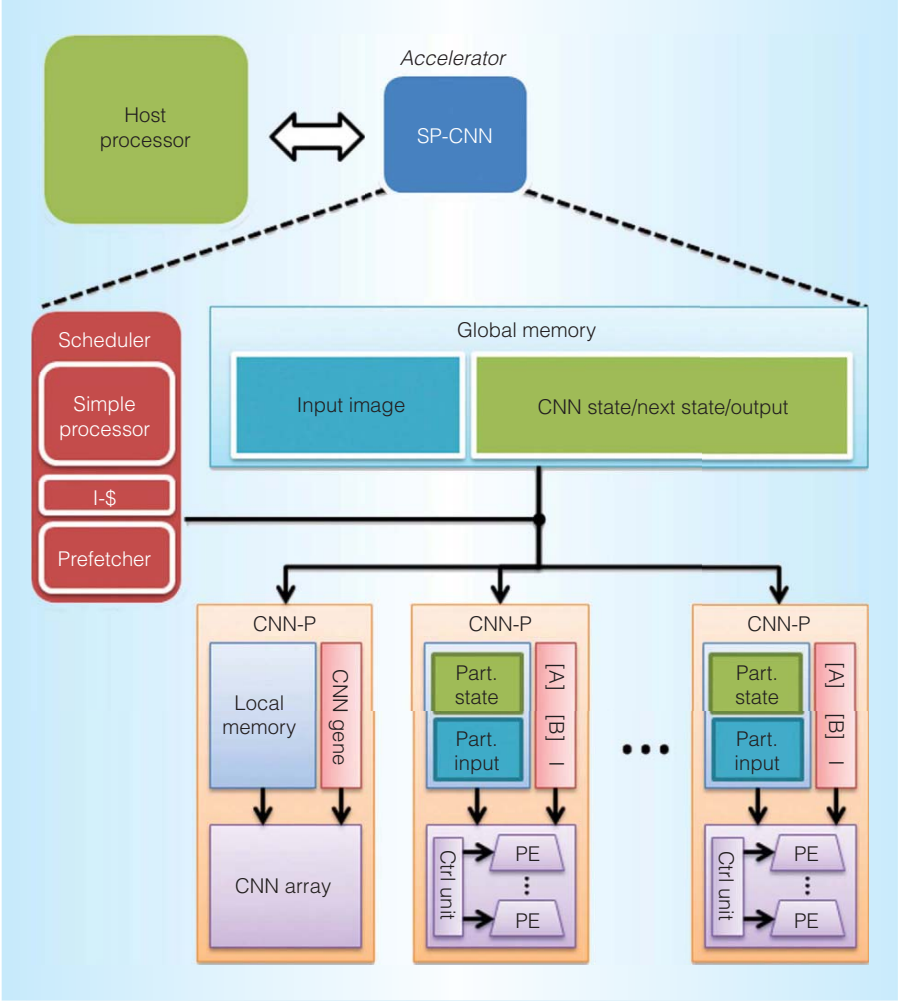

Figure 6. Example SP-CNN architecture. The SP-CNN architecture works as an accelerator to the host processor. The SP-CNN architecture design was influenced by GPGPU accelerators.

processing on the CNN array when it has either hit the interval or converged to a steady state.

SP-CNN architecture

The SP-CNN architecture (see Figure 6) was influenced by Nvidia's G80 architecture (www.nvidia.com/page/geforce_8800.html). SP-CNN serves as an accelerator to a host processor. The SP-CNN accelerator has a global memory, a scheduler, and possibly several CNN-Ps. A CNN-P is a CNN computing unit that has local memory, the CNN gene, and a CNN computing array that comprises a set of processing elements (PEs).

We mentioned that the SP-CNN accelerator can have one or more CNN-Ps. The benefit of having multiple CNN-Ps is that multiple partitions can be executed concurrently. Furthermore, multiple CNN-Ps can better hide the memory latency. However, too many CNN-Ps can increase memory bandwidth contention, which leads to less-than-linear scaling in performance. Furthermore, a design tradeoff must be made between having many small CNN arrays versus a few large arrays.

Evaluation methods

We evaluated the following benchmarks, which are commonly used in CNN-based

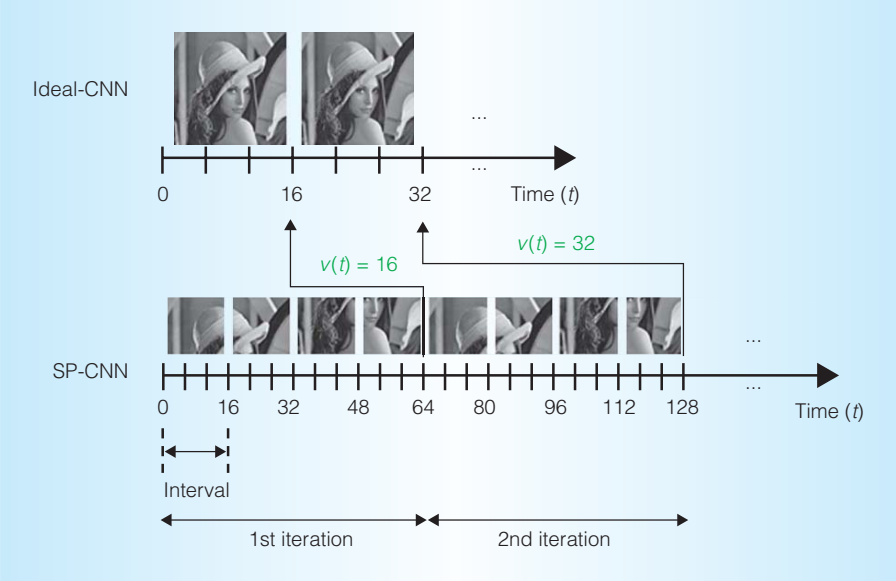

Figure 7. Virtual time example. The virtual time is the same "*t*" as in the original CNN equations, and the physical time is the virtual time multiplied by the number of the multiplexing factor.

image processing applications: Corner Detection, Edge Detection, Connected Component, Hole Filling, Rotation Detector, and Shadow Creator. We used 10 test input images, with each image stored at a resolution of 1,024 × 1,024 or 2,048 × 2,048. We chose these dimensions because the number of pixels for each dimension roughly corresponds to the number of pixels in 720p and 2,048 × 1,536, respectively.

We developed a functional simulator of our SP-CNN architecture. The default SP-CNN parameters are one CNN-P unit with a CNN array size of 128 × 128 and an interval time of 128 CNN time units. The functional simulator returns the convergence time of an application in CNN time units, wherein a single unit corresponds to one state or output computation. Furthermore, transfers of partitions to and from memory are assumed to occur instantaneously.

To observe the effects of memory transfers, we also developed a timing simulator that used DRAMSim2 as a memory simulator,[6] with CNN-P timing parameters based on the visual attention engine (VAE) architecture.[3] We used a 2-Gbyte global DDR3 memory specified by the DRAMSim2 configuration file *DDR3_micron_16M_8B_x8_sg15.ini*.

Directly comparing the total time between the ideal CNN (in which the CNN array size equals the image size) and SP-CNN is not very informative because the SP-CNN's time will be at least scaled by the image to CNN array size ratio. Instead, we introduce the concept of *virtual time,* which represents how many CNN time units a partition has actually processed on the CNN computing unit. Figure 7 illustrates the concept of physical time; t represents total time and $v(t)$ represents virtual time.

For the SP-CNN case, even though 64 total time units have passed since the first iteration, each partition has computed for at most 16 time units. Again, after the second iteration, while the total time units is 128, only 32 units of virtual time have passed. For Algorithms 1 and 2, t and vt represent the total time passed and virtual time passed.

Results

Figure 8 compares the average virtual convergence time between the ideal CNN and our proposed SP-CNN mechanism. SP-CNN is competitive with the ideal CNN for all benchmarks for both the 1,024 × 1,024 and 2,048 × 2,048 images.

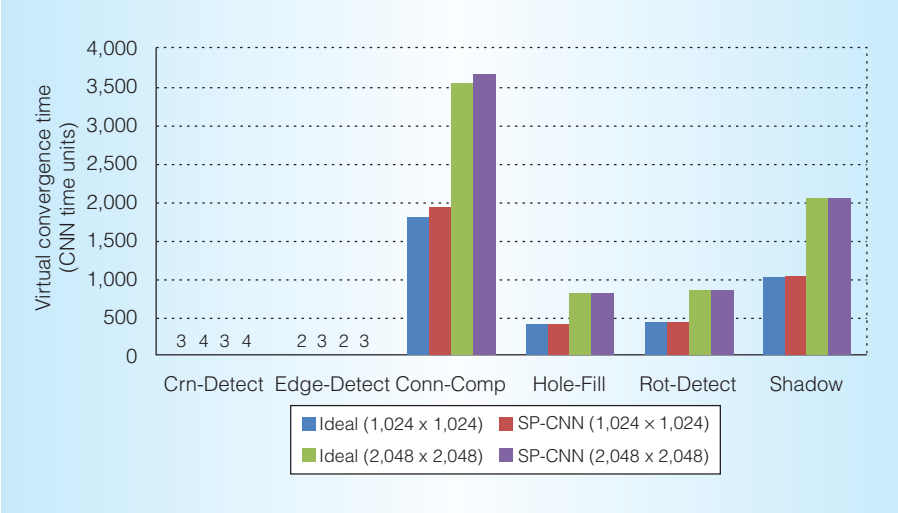

Figure 8. Average virtual convergence time of ideal CNN versus SP-CNN. The performance of SP-CNN is competitive with the ideal CNN for all benchmarks. (Crn-Detect: Corner Detection; Edge-Detect: Edge Detection; Conn-Comp: Connected Component; Hole-Fill: Hole Filling; Rot-Detect: Rotation Detector; Shadow: Shadow Creator.)

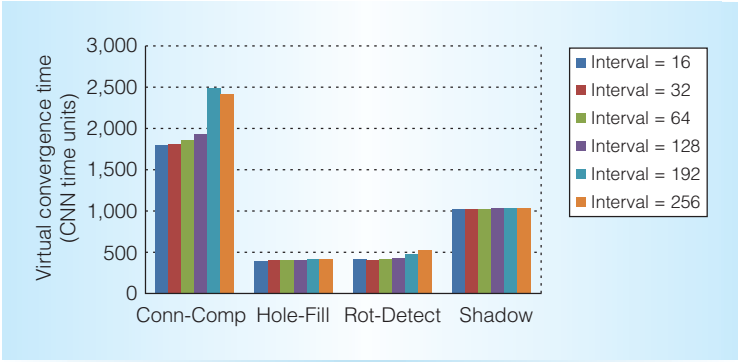

Figure 9. Effect of interval time on virtual convergence time. For Conn-Comp and Rot-Detect, as interval time increases, the convergence time also increases, whereas Shadow and Hole-Fill do not show that much difference.

One important contribution of this article is that it shows that certain CNN applications are robust enough to use interval values larger than 1 during multiplexing execution. Choosing an effective interval time is important in limiting an application's convergence time. Too small of an interval time could lower the virtual convergence time, but it also increases the overhead of data transfers. On the other hand, too large of an interval time can increase the virtual and total convergence times.

Figure 9 shows the average virtual convergence time results as interval time increases. We exclude Corner Detection and Edge Detection from our results because these two applications were local-type CNN applications and converge much sooner than the interval time limit. As expected, when increasing the interval time, the virtual convergence time tends to increase. This is best seen with Connected Component.

Figure 10 shows the timing results with SP-CNN. With 1,024 × 1,024 images, SP-CNN can perform below the 60 frames-per-second boundary for most applications. In the case of 2,048 × 2,048 images, SP-CNN meets the 30 frames-per-second boundary for most applications when the number of CNN-Ps is equal to 4 or 8.

Figure 11 illustrates the effects of scaling the number of CNN-Ps. None of the applications show linear scaling when we increase the number of CNN-Ps. SP-CNN does not scale because memory contention becomes a dominating factor. This can be seen in Figure 12, which illustrates the percentage of time a partition execution spends communicating to memory. As seen in the benchmarks that show sublinear scaling, memory communication accounts for more than 50 percent of

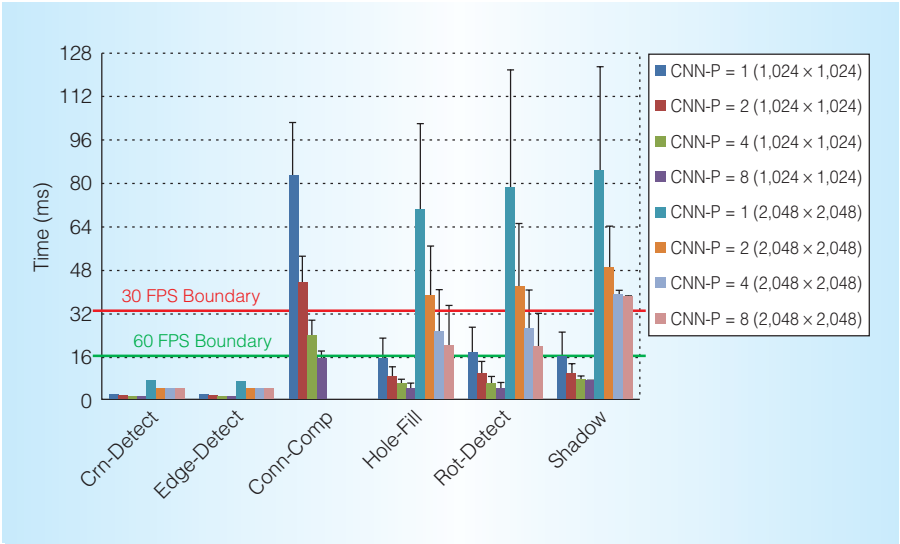

Figure 10. Average execution time of SP-CNN. (The bar represents maximum execution time observed). When the number of CNN-P is equal to 4 or 8, SP-CNN can perform below the 60 framers-per-second boundary.

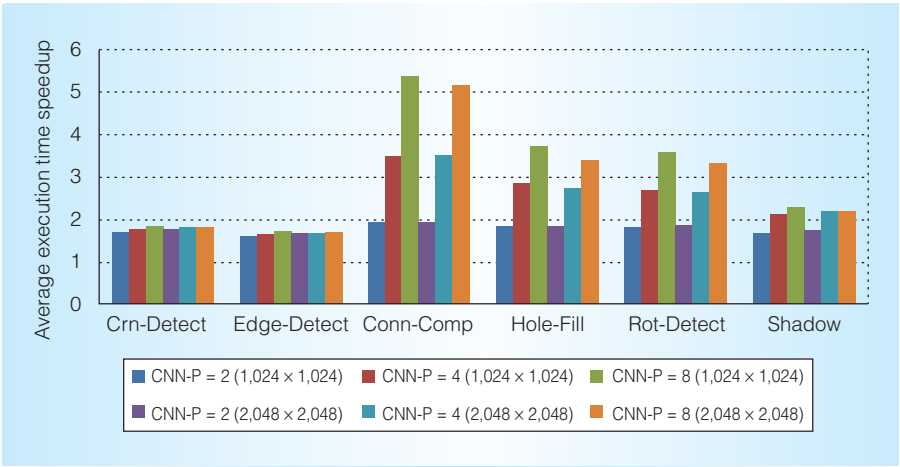

Figure 11. Speedup using multiple CNN-Ps. The performance sublinearly increases as we add more CNN-P units.

partition execution and rapidly grows as the number of units increases.

SP-CNN depends on the CNN application being robust enough to handle the multiplexing mechanism. The choice of SP-CNN parameters, such as interval and ordering, can have different effects on the virtual and convergence times. Currently, the application programmer must determine these parameters' values, but in future work, these parameters may be able to be identified through simulations and profiling. Furthermore, although all of our benchmarks converged to the correct output under the various parameters, some applications never converge, or converge to an incorrect solution when the interval time is greater than 1.

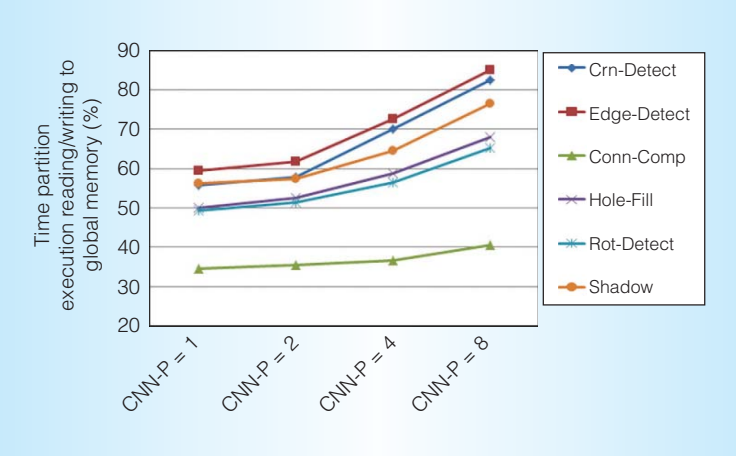

Figure 12. Average percentage of time partition execution spends in memory transfers. As we increase the number of CNN-P units, more time is spent for memory transfers.

Again, this is an evaluation that the application designer must make. For future work, we would like to develop a mechanism to identify applications suitable for SP-CNN, beyond image processing applications.

MICRO

References

1. H. Mujtaba, "Intel Aims for Mobile Space with the 32nm Atom z2420 Lexington SoC and 22nm Bay Trail SoC," *WCCF Tech*, 2012; http://wccftech.com/intel-aims-mobile-space-32nm-atom-z2420.
2. L.O. Chua and L. Yang, "Cellular Neural Networks: Theory," *IEEE Trans. Circuits and Systems*, vol. 35, no. 10, 1988, pp. 1257–1273.
3. S. Lee et al., "24-GOPS 4.5-mm2 Digital Cellular Neural Network for Rapid Visual Attention in an Object-Recognition SoC," *IEEE Trans. Neural Networks*, vol. 22, no. 1, 2011, pp. 64–73.
4. L.O. Chua and L. Yang, "Cellular Neural Networks: Applications," *IEEE Trans. Circuits and Systems*, vol. 35, no. 10, 1988, pp. 1273–1290.
5. K. Karacs et al., *Software Library for Cellular Wave Computing Engines*, Cellular Sensory Wave Computers Lab., 2010; http://cnn-technology.itk.ppke.hu/Template_library_v3.1.pdf.
6. P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *Computer Architecture Letters*, vol. 10, no. 1, 2011, pp. 16–19.

Dilan Manatunga is a PhD student in the School of Computer Science at the Georgia Institute of Technology. His research focuses on optimizations for mobile operating systems and architectures. Manatunga has a BS in computer science from the Georgia Institute of Technology. He is a member of IEEE and the ACM. Contact him at dmanatunga@gatech.edu.

Hyesoon Kim is an associate professor in the School of Computer Science at the Georgia Institute of Technology. Her research interests include high-performance, energy-efficient heterogeneous architectures; interaction between programmers, compilers, and microarchitectures; and developing tools to help parallel programming. Kim has a PhD in electrical and computer engineering from the University of Texas at Austin. She is a member of IEEE and the ACM. Contact her at hyesoon@cc.gatech.edu.

Saibal Mukhopadhyay is an associate professor in the School of Electrical and Computer Engineering at the Georgia Institute of Technology. His research interests include neuromorphic computing, low-power digital and mixed-signal systems, voltage regulation, and power and thermal management. Mukhopadhyay has a PhD in electrical and computer engineering from Purdue University. He is a member of IEEE. Contact him at saibal.mukhopadhyay@ece.gatech.edu.