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

Abstract

Specialized accelerators have become prevalent in many mobile computing platforms for their ability to perform certain tasks, such as image or sound recognition, at a lower power cost than a generalized CPU or GPU. In this paper, we focus on using Cellular Neural Networks as a specialized accelerator. The Cellular Neural Network (CNN) is a neural computing paradigm that is well suited for image processing applications. CNNs have already been implemented as commercial products. However, they were originally developed only to handle relatively small image sizes. In this paper, we propose SP-CNN, which provides scalability to the CNN paradigm. We present our SP-CNN architecture, as well as a set of multiplexing algorithms. With these additions, SP-CNN can handle any image size and various multiplexing schemes. We demonstrate the proposed multiplexing algorithms over a set of six image processing benchmarks, as well as present a performance and power analysis of SP-CNN .

1. Introduction

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 SOC and Bay Trail SOC [34], more than 25% of the chip area is used for video/image-related processors. Excluding memory/storage, the majority of the chip area is dedicated to special accelerators for image/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 getting high attention as special accelerators. Qualcomm's NPU systems or IBM's TrueNorth are two examples. These processors promise a high energy efficiency. Similar to these neural network processors, which are based on artificial neural networks (ANN), cellular neural network processors (CNN) [5, 3] have been studied widely for image processing applications. The CNN was introduced by Chua and Yang [5, 3] and is a type of neural network that consists of a homogenous 2D array of cells, in which each cell communicates with only a fixed set of neighbor cells. The connections between cells are local, unlike ANN, thus eliminating the need for long distance interconnects. Since the performance and power of interconnects has scaled poorly over time, the CNN is well suited for hardware implementation in future technology nodes for image processing applications. The other benefit of CNN over other neural network processors is very few parameters need to be trained (CNN has only 1 layer and all cell connections are limited to neighborhood cells, while other neural network processors have variable number of layers and cell connections are programmable.) Hence, we argue that while developing an algorithm for CNN is hard, once an algorithm is developed, it is very robust. Detailed comparisons are the beyond of the scope of our papers.

Surprisingly, CNN has a longer history of being implemented as a hardware solution, even for commercial chips, than ANNs as shown in Table 1. CNN is an still active research areas for image processing applications. For example, Ramirez-Quintana et al. surveyed artificial neural network algorithms that are used for image processing from the last 10 years and showed that CNN and self organizing map have the largest number of papers [24].

There are two challenges in the CNN research, which are not well addressed yet. First, developing new algorithms for the CNN is not a trivial task. Unlike other neural network processors, learning algorithms for CNN have not been well developed. However, there are already a number of algorithms and applications have been developed for CNN. For example, the cellular wave computing library [28] by Roska's research group provided more than 100 processing algorithms or segmentation algorithms baed on more complex PDE solver equations [32]. Hence, if these algorithms can be easily used like as libraries on a special accelerator, the difficulties of developing new algorithms might not be exposed to programmers.

The second challenge is the scalability of CNN hardware and algorithms. So far, all CNN algorithms are developed under the assumption that the number of cells is equal to the image size. Unfortunately, designing a CNN processor to handle large image sizes is not a trivial task. Using simple naive multiplexing algorithms in [11] or simple stencil operations will quickly saturate memory bandwidth.

Hence, to overcome these two problems, in this paper, we propose a scalable CNN architecture that can act as a *accelerator*, which can be easily integrated with other general purpose processors. Furthermore, we purpose both software and hardware solutions to provide *scalability*. The proposed software algorithms use small CNN arrays to handle large input images and the hardware architecture provides the necessary programming features to overcome the scalability problem. The proposed architecture borrows many architecture design decisions from GPUs even though CNN is a *non-von neumann architecture* to be easily integrated with a host processor. The proposed solution adopts a stencil computation approach to achieve CNN scalability, but the algorithms detail how we use the robustness of CNN applications to reduce memory bandwidth requirements significantly.

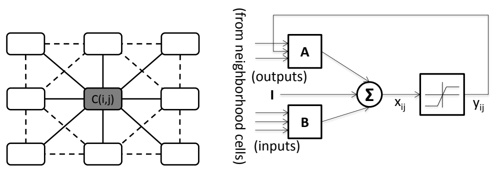

Figure 1: left: CNN cell connections and neighbor cells of the cell at C(i,j). right: Operation of CNN cell

Our proposed solution consists of two components: (1) algorithms to partition input images into multiple images that will use a smaller CNN-based processor, and (2) an architecture that provides a programmable interface that supports these scalable CNN algorithms.

Our contributions are as follows:

1. We propose a programmable CNN architecture, called SP-CNN, as an accelerator to be connected with a host processor. SP-CNN has multiple CNN array processors, a scheduler and a prefetcher.
2. We propose an algorithm and an architecture that can handle any input image size without causing memory contentions. This is the first work to demonstrate that the CNN paradigm can perform real time image processing on modern image sizes.

2. Background on CNN

2.1. CNN Operation

The Cellular Neural Network (CNN) is a type of neural network that is composed of a homogeneous 2D-array of cells. The array can be 2-,3- or n-dimensional, but most CNNs are implemented as a 2D-array. The CNN was introduced using analog processing cells with continuous signal values, where cells are only connected with a finite radius of neighbor cells [5, 3]. Figure 1 shows a 3x3 CNN array with a radius of one. The solid lines represent the connections between cell C(i,j) with its neighbor cells, while the dashed lines represent the connections other cells have with their neighbors.

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

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

Equation (1) is the digital state equation and Equation (2) is the digital output equation [35]. The variable x is the cell's *internal state*, y is the cell *output*, and u is the input to the cell. ij refers to a grid point associated with a cell in the 2D array and $C(k,l)\in N_r(i,j)$ represents the grid points within a radius of r of the cell ij (e.g. the cell's neighborhood). The scaling factors for the output-related signals are called the *A template*, which works as a feedback operator. The scaling factors for the input-related signals are the *B template*, and act as a control operator. In every cell there is a constant bias I, which is called the *I template* or threshold. These templates are typically constant[1] and are used to specify the interactions between a cell and its neighbor. The group of template values used to solve a given problem is known as the *CNN GENE*[2]. The output Equation (2) is a nonlinear function, which is typically a piecewise linear function.

During a discrete CNN time unit, each cell in the CNN array computes the solutions to equations (1) and (2) in parallel. These values represent the respective state and output in the next CNN time unit. Typically, a CNN application is run by having the CNN array process through these time units until the state has converged to a steady-state value.

2.2. Survey of CNN chips

Table 1 shows some of the commercial and academic CNN chips that have been developed. The largest size that has been implemented is 176x144 on QCIF [21]. Because the operation of CNN cells is fairly simple, these chips show fairly high clock frequencies or low time constant, with very low power consumption (less than 10μW per cell even with previous generations of technologies).

3. Programmable Multiplexing Algorithms

3.1. Multiplexing CNN

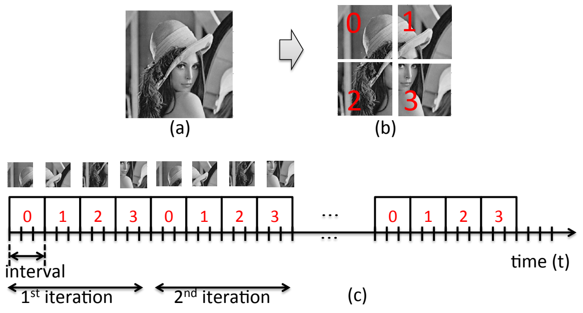

Figure 2: Multiplexing Example. (a) original image (b) partitioned images (c) illustration of partitioning operations and interval, iteration definitions.

Since the CNN operation can be represented as a stencil computation (convolution), the basis of our SP-CNN mechanism is similar to how stencil computations are done in GPU systems. Specifically, the image is partitioned based on some block size (in our case this the size of the CNN array), and the compute units operate on each block. Figure 2 shows how the original image (a) is divided into multiple partitions (b), and each partition (sub-image) is processed by an available CNN array. Finally, if we wanted to emulate the ideal CNN case, we would process each partition on the CNN array for a single CNN time unit dt, and then repeat the process until the entire image has converged. Figure 2 (c) shows a timescale example

[1]In general, A and B are not space invariant. That is why they are noted as $A(i,j;k,l)$ and $B(i,j;k,l)$ in Equation (1). However, typical CNN templates have space invariant A and B values.

name	tech./@supply	image size	performance	power, area	note
VAE [11] (2008)	130 nm	80x60	200MHz, 22GOPS , 120 PES	84mW, 4.5 mm^2	support ideal time multiplexing
g_mOTA [30] (2008)	180 nm	-	-	7.0μW	Hspice simulation
Flak+ [8] (2006)	180 nm	4x4	4 ns time constant	155μm^2/cell, 9.8μW per PE	Boolean operations
SCAMP [7] (2005)	0.6(analog 3.3V, digital 5V) μm CMOS	21x21	1.1GIPS (2.5MHz)	40mW, 98.6μm x 98.6μm	SIMD current-mode analog matrix processor (vision chip)
Gabor Filter [1] (2004)	250 nm2.5V	32x64	434μs	3mW	
ACE16k [25, 13](2002)	350 nm3.3V CMOS	128 x 128	330GOPS,	100GOP/Joule	mixed-signal digital chip
QCIF [21] (1999)	250 nm CMOS@(2.5 or 3.3V)	176x144		73 transistors per PE 3000 cells/mm^2	
Kingset+ [10] (1995)	2.4μm	4x4	10μ cell time, time constant 10μs	380cells/cm^2 40μA/cell	Analog CNN implementation using CMOS

Table 1: A summary of example CNN chips

of this operation, where the time the partition operates on the CNN array is called the interval, and the operation of every partition on the CNN array is known as an iteration. Convergence is then defined as when there is no change in the cell state values for every partition from one iteration to the next.

Now, let us consider processing a 1024x1024 image by emulating a 1024x1024 CNN array using only a 128x128 CNN array. Since the local memory of the CNN is proportional to the number of cells in the array, we can typically only keep one or two partitions worth of data in the CNN's local memory. This means to process each partition, we need to read the partition data from global memory and then write the computed results back to global memory. However, the issue is that we only process a partition for one CNN time unit. As seen in Equation (1), only a small amount of calculation is necessary for a single CNN time unit, approximately 18 multiply-adds each cell when running a 3x3 CNN template. To load the partition though, we need to read about 2 * 128 * 128 = 32 KB from memory (one byte for state and one byte for input). Therefore, since the computation is very small compared to the memory transfer cost, the ideal CNN emulation is heavily memory-bandwidth bound.

To solve this memory-bandwidth issue, we realized that certain CNN applications are robust enough to allow a partition to process for multiple *dt* units while still converging to the correct solution. For example, we saw that with the HoleFilling application, we could actually operate each partition on a 128x128 for 128 CNN time units during an iteration and still converge to the correct solution. This means that the memory bandwidth bound ideal CNN emulation, becomes a compute bound approximation of the ideal CNN operation.

Alg. 1 and 2 show the pseudocode of the original CNN and our multiplexing CNN algorithms. Essentially, with SP-CNN, the CNN is operated on each partition for *interval* time units, and then iterates this process until the entire state has converged to a steady state. We will discuss the mechanism behind $loadStateAndInputToCNN()$ in Section 3.3.

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

Algorithm 1: Traditional CNN Algorithm

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

Algorithm 2: SP-CNN Algorithm (Multiplexing CNN)

3.2. Multiplexing Example and Issues

In this section, we evaluate some other time-multiplexing mechanism and explain the issues that can occur. These issues motivated the development of a programmable interface for the multiplexing module. The first issue is that what we term as naive time-multiplexing does not produce the correct outputs. In naive time-multiplexing, we apply each partition on the CNN array until the partition converges to a steady state value. This can be viewed as applying the SP-CNN mechanism for just one iteration with an infinite interval value. As one might expect, for certain types of applications, this process does not produce the correct solution.

To illustrate the problem concretely, we use the Hole-Filling algorithm [15]. The Hole-Filling algorithm fills the interior of all closed contours in an image, and this algorithm is typically used for character recognition. The Hole-Filling algorithm requires *global information* to know whether a pixel is within a hole, i.e. a pixel cannot determine its final value based simply on its neighbors. The basic insight of identifying holes is that the algorithm starts to search holes from one corner and propagates the empty information until it hits connected com-

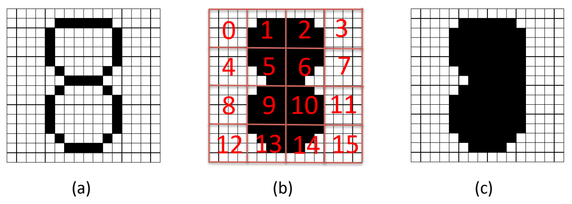

Figure 3: Example of the Hole-Filling with Naive Multiplexing (a) input image (b) the correct output image (c) naive multiplexing outcome

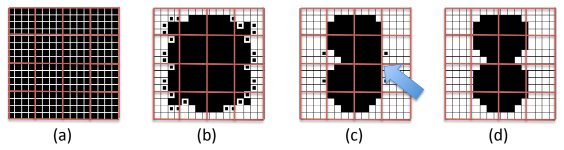

Figure 4: Example of Hole-Filling using SP-CNN algorithm 2. (a) initial x(0) (b) output values after 1st iteration (c) output values after 2nd iteration (d) final output

ponents. [2] Since the Hole-Filling algorithm requires global information, it is difficult to arrive at the correct solution by just processing the image for one iteration.

Figure 3 shows an example of multiplexing the Hole-Filling CNN, where the input image size is 16x16 but the CNN array 4x4. Therefore, to perform the multiplexing, the input image is partitioned into 16 4x4 partitions, which are processed sequentially. Figure 3 (a) shows the input image, while (b) shows the correct output and the partitioning. Figure 3 (c) shows the output image of operating partitions 0-15 until each partition converges. Most partitions find the right answer but partitions 6 and 10 fill the wrong pixels. This is because none of the new pixel values are shared to partitions 6 and 10 during the processing of one iteration. Specifically, for partitions 6 and 10 to see the information of how pixels from neighboring partitions have changed, it must repeat processing with the new values other partitions have calculated.

Figure 4 shows the progress of $y(t)$ values for the Hole-Filling algorithm with our proposed SP-CNN multiplexing algorithm. The key of the proposed algorithm is that image is processed for multiple iterations, and new cell information is shared between the partitions at the end of each iteration. Furthermore, unlike the naive case, each partition is only processed for an interval instead of till convergence. Since global information is now passed at every iteration, partitions 6 and 10 will have the information necessary to compute their correct outputs. Therefore, with algorithm 2, SP-CNN eventually saturated to the correct output.

3.3. Information Propagation

3.3.1. Boundary Conditions Similar to other stencil operations, cells in x=0 or y=0 do not have all their neighbor cells.

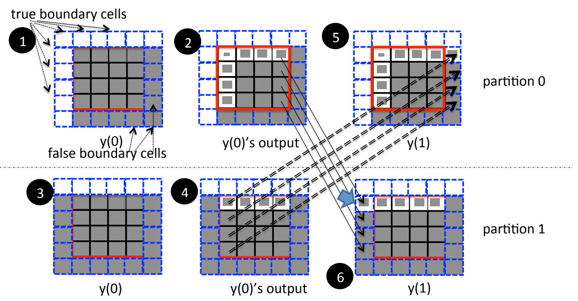

Figure 5: Boundary condition propagation. Top row: partition 0. Bottom-row: partition 1 (Depending on the value, the filled box size is varied.) y(0): iteration 0, y(1): iteration 1

Typically, these boundary conditions can be simply 0 values, values of the boundary cells, or values from the the other side of the image (x=N or y=N where N is the size of the image). [3] In the case of SP-CNN's partition CNN operation, SP-CNN uses the apron approach used in GPU stencil computation [22]. Specifically, we set the boundary conditions of the partition as the boundary values from neighboring partitions. This boundary condition method is also how cell information is shared partitions. Another way to view this technique is as an emulation of ideal time-multiplexing, except where the boundary values during partition processing change at the end of every iteration.

Figure 5 illustrates how information propagates through boundary conditions. The first row shows $y(t)$ values (state variable) in partition 0. With partitioned images, we have *true boundary cells* and *false boundary cells*. True boundary cells are the boundary cells from the ideal CNN case (non-multiplexing version), and false boundary cells are new boundaries that are created because of partitioning the image. True boundary cells have values specified by the original algorithm, but the false boundary cells are now the values the neighbor cells in the neighboring partitions arrived at during the previous iteration.

Figure 5 shows how boundary information is copied for partitions 0 and 1. These partitions are from Figure 3(b). To indicate the hole information propagation, we filled part of pixels to indicate values ranging from [0-1]. A fully filled gray box means 1 and an empty box means 0. The top row shows the $y(t)$ value of partition 0 and bottom row shows that of partition 1. After the first iteration is over, the boundary information is passed between partition 0 and 1 because they are neighboring partitions. Through these boundary conditions, the global information can be passed between partitions.

Referencing algorithm 2, the $loadStateAndInput()$ function is responsible for ensuring that the boundary values to the CNN array are set as pixel values from neighboring partitions. Once partition processing is finished, we save the new state values to a new location using the $saveToNextStateFromCNN()$ function. The reason we must save to a new location is be-

[2] The details of Hole-Filling algorithm can be found in [15].

[3] Boundary conditions are typically specified by the CNN gene.

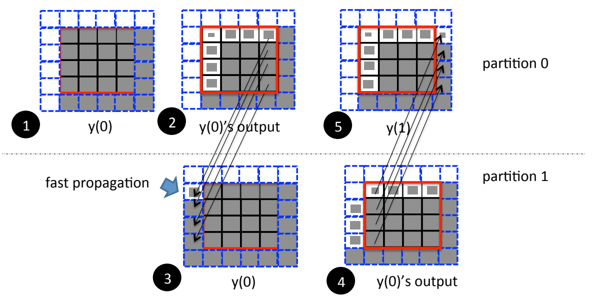

Figure 6: Fast-Propagation method

cause the original state values may still be used as boundary conditions for another partition. Finally, once one iteration has passed, the *swapStateAndNextStatePointers*() function is called, so that the next iteration has uses the information propagated through the cells.

3.3.2. Fast Propagation According to algorithm 2, the boundary values are passed at the next virtual time because x(t) and y(t) are updated after one iteration, as shown in Figure 5. However, instead of waiting to get the updated y(t+1) value, since y(t+1) value may already be available from partition 0 when we begin operating on partition 1, we could pass that information at time t. This can allow certain applications to converge faster. Figure 6 shows this example. We call this boundary condition update method *fast-propagation* and the original method as called *slow-propagation*. While fast-propagation can reduce the convergence time, it can cause other performance bottlenecks, which we will discuss in a later section. Furthermore, with fast-propagation, partition 1 can only start after partition 0's interval is finished, while in slow-propagation, partition 1 can start before partition 0 is finished. This difference may not seem large when only one partition is processed at a time by the CNN architecture, but when multiple partitions can be handled in parallel (discussed in a later section), there is a trade-off between fast and slow propagation.

3.3.3. Different traversal order When using fast propagation, depending on the order of processing partitions, the global information is propagated differently. Examples of traversal orders could be row-major, column-major, spiral, zigzag etc. With certain applications, a different traversal order can help reduce convergence time. For example, the hole-filing application can converge faster with a spiral traversal when using fast propagation.

3.4. Early-Finish: Variable Interval Length

After some preliminary analysis with SP-CNN, we learned that certain partitions would converge to a steady state before the interval was over. This does not necessarily mean the entire image has converged, but it does mean that for the rest of the interval, the CNN array would not perform any useful computation. In order to avoid this, we introduced the concept of $Early-Finish$ to our algorithm. With $Early-Finish$, a partition stops processing on the CNN array when it has either hit the interval limit or when it has converged to a steady state. Our evaluations show that $Early-Finish$ performs more than 2 times better than fixed interval, so all of our SP-CNN evaluations employ $Early-Finish$.

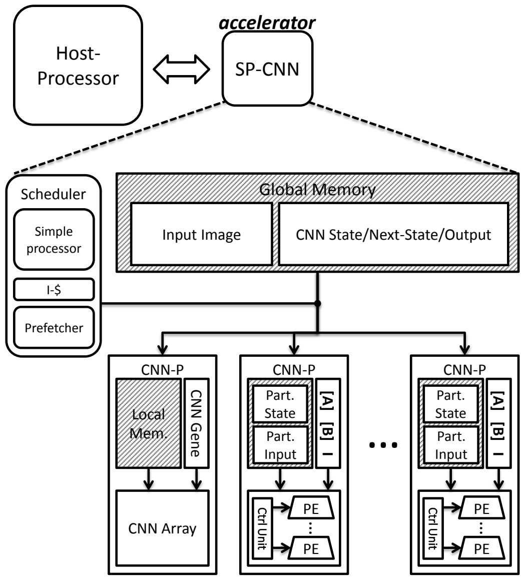

Figure 7: Overview of SP-CNN

3.5. SP-CNN Programming Interface

Similar to CNN-UM [27] [4], SP-CNN is programmable. CNN-UM can program the basic CNN parameters, like *CNN Gene*, that consist of A, B, and I and the radius values, as well as the initial states, and true boundary conditions. However, in SP-CNN, in addition to these basic parameters, it can program the partitioning size, interval period, the iteration limit, traversal order, and boundary condition propagation method to support different partitioning and multiplexing algorithms.

4. Architecture of SP-CNN

There are several microarchitecture design options that can be explored to design a scalable and programmable CNN processor (e.g. memory hierarchy, the CNN array hierarchy, etc). For many decisions, we take NVIDIA's G80 architecture [19] as an example and focus more on demonstrating the necessary features to support scalability and the programmable interface.

4.1. Design Overview

Figure 7 shows an overview of the proposed architecture, SP-CNN, which is connected to a host processor (e.g., a CPU processor) as an accelerator. SP-CNN has global memory, a scheduler, and several `CNN-Ps`. A CNN-P is a CNN processor that has local memory, the CNN Gene, and a CNN array, where the array is composed of processing elements (PE).

[4] A programmable CNN chip is typically referred as a CNN UM (Universal Machine)

The scheduler handles CNN programs and also generates most of the control signals for the CNN-Ps and DRAM memory (i.e., global memory) requests. A CNN-P communicates with the scheduler and global memory, but does not communicate directly with other CNN-Ps.

4.2. Processing Elements

Each CNN-P has several processing elements (PEs) that are responsible for computing the state and output equations for each cell. Each PE has a set of processing units (PU) and registers. Processing Units have multipliers and other special transfer functions for output equations. The number of multipliers in one PU is a design option. One multiplier can be used to compute all neighbor cells, or several multipliers could be used to compute in parallel. If we consider the case of $r = 1$, 18 multiplications are required. Since $A(i, j; k, l) * y_{kl}(t)$ and $B(i, j; k, l) * u_{kl}(t)$ are convolution operations, and for a 3x3 matrix we would need $9\times$ 2 multiplications.

4.3. Registers and Interconnection

Registers store input ($u_{ij}(t)$), output ($y_{ij}(t)$), and state values ($x_{ij}(t)$) in Equation (1). In a $r = 1$ system, each PU takes nine-neighborhood values. Within a PE, the register can be directly connected as an input to a multiplier or an MUX if multipliers are shared. $A(i, j; k, l)$ and $B(i, j; k, l)$ are constant during a program execution time. The registers that store these constant values are called *constant registers*. These constant registers are set at the beginning of the program by the scheduler processor. Space-variant and temporal-variant CNN paradigms can be supported by programmable registers. The next output values ($y_{ij}(t + 1)$) are stored in output register after passing the non-linear filter unit. The output registers are shifted to state register values at the next virtual time.

4.4. Multiple CNN-Ps

In Section 3, we explained the concept of multiplexing with one CNN-P. With multiple CNN-Ps, multiple partitions can be executed concurrently. Multiple CNN-Ps provide flexibility in terms of software and hardware designs, and can also help hide memory latency. However, too many CNN-Ps can increase memory bandwidth contention, leading to less than linear scaling. This impact is evaluated in Section 6.5. Furthermore, a design tradeoff must be made between having multiple small CNN-Ps versus few large CNN-Ps. When we have a smaller partitions, there will be more false boundary cells, which can increase convergence time. However, few CNN-Ps reduce the amount of parallelism we can extract as well as reducing the CNN-Ps ability to hide memory latency.

4.4.1. Fast-propagation Drawback In section 3.3.1, we discussed the benefit of fast-propagation. However, we have not defined a standard for fast-propagation with multiple CNN units. In this case, since a subset of partitions are always operating in parallel, some partitions will receive updated values while others have none. Therefore, in our simulations, we only observe the benefit of fast-propagation for one CNN-P.

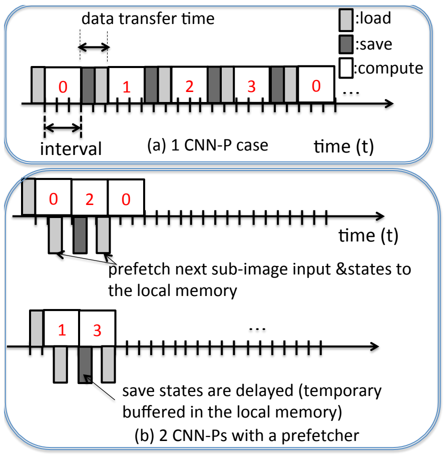
data transfer time
:load
:save
:compute
0
1
2
3
0
...
interval
(a) 1 CNN-P case
time (t)
0
2
0
time (t)
prefetch next sub-image input &states to the local memory
1
3
...
save states are delayed (temporary buffered in the local memory)
(b) 2 CNN-Ps with a prefetcher

Figure 8: CNN timing chart example: With a prefetcher and 2 CNN-Ps, data transfer time is completely overlapped with computation period.

4.5. Scheduler Processor

The scheduler processor is a simple in-order processor that fetches instructions and sets up the CNN GENE in CNN-P. The main job of the scheduler processor is as a task scheduler to the CNN-Ps and a prefetcher from global memory to the local memory in each CNN-P. Each CNN-P also communicates with the scheduler processor at the end of each interval period. The scheduler generates memory requests to load/store input& state information from/to global memory. These memory requests are fairly simple, and similar to a repeat move operation in x86. The scheduler processor can have a prefetcher to hide memory latency as shown in Figure 8(b).

4.6. Local Memory

Each CNN-P processes one partition at a time. The number of registers should be sufficiently large to keep all the input/state/output values for one partition, so in theory it is possible not to have any additional storage. In that case, as shown in Figure 8(a), the CNN-P is idle while the states are copied from/back to the global memory. However, to optimize performance, SP-CNN adds local storage to each CNN-P. Local storage stores input/state information (Part. State and Part. Input in Figure 7) for the next partition to be processed by the CNN-P, and these local storage is utilized with a prefetcher (or having multiple CNN-Ps). The local storage can be used as double buffering similar to graphics frame buffers.

4.7. SP-CNN vs. GPUs

Many of the design decisions of SP-CNN are inspired by the G80 architecture, so they share many similarities. Table 2 compares SP-CNN and GPUs for both hardware and programming aspects. GPUs are a Harvard architecture but SP-CNN is somewhat similar to systolic array processors.

	SP-CNN	GPUs (G80)
Hardware	CNN-P	SM
	Processing elements	SP
	Processing units	Execution units
	Registers	Register
	Const. registers	Const. cache
	Local storage	Shared memory
	Global memory	Global memory
Software	execution of one sub-image	CUDA block
	one cell operation	CUDA thread
	template (CNN GENE)	CUDA programs

Table 2: Comparisons between SP-CNN and GPUs

5. Evaluation Methods

5.1. Benchmarks

Table 3 describes our evaluated benchmarks, which are commonly used in CNN-based image processing applications. Two of the applications (Corner Detection and Edge Detection) do not require global information to be propagated while the rest do. All applications use 3x3 templates. We use 10 test input images, with each image stored at a resolution of 1024x1024 and 2048x2048. We chose these dimensions since the number of pixels for each dimension roughly corresponds to the number of pixels in 720p and 2048x1536 (i.e., modern high-resolution mobil devices) respectively.

5.2. Methods

In our evaluation, we developed functional simulators for four different CNN architectures, which are described in Table 4. Our default SP-CNN parameters are 1 CNN-P unit, row-major partition order, slow-propagation, interval length=128, and a CNN-P size of 128x128. We also developed a timing simulator that connects our functional SP-CNN simulator to the DRAMSim2 memory simulator [26]. These timing parameters are based on the VAE architecture, which is the latest digital based CNN chip [11]. Each PE has an ALU that has a multiplier, adder and shifter. The time to process 1 CNN (t) is proportionally dependent on the number of multipliers in one PE. Similar to the VAE architecture, SP-CNN also has 1 multipler shared for all 18 multiplications. To reduce power, the VAE architecture chooses to share a PE amongst many CNN cells. In our design, we chose to use a value of 32 (a single PE is responsible for computing the values of 32 cells). The CNN chip is clocked at 200 MHz.

Benchmarks	Abbr.	global/local
Corner Detection [4]	Corner-Detect	local
Edge Detection [31]	Edge-Detect	local
Connected Component [16]	ConnComp	global
Hole Filling [15]	Hole-Fill	global
Rotation Detector [14]	Rot-Detect	global
Shadow Creator [17]	Shadow	global

Table 3: Evaluated Benchmarks

Name	Mechanism
Ideal	CNN array size is equal to the image size
Naive-No-share	Boundary conditions when processing each partition is determined by the CNN gene's true boundary conditions.
Naive-Share	Boundary conditions when processing each partition is determined by the neighboring pixels or true boundary conditions.
SP-CNN	Proposed architecture

Table 4: Experimental CNN paradigms

For the memory configuration, we limited each CNN-P unit to issuing one memory request per cycle and 64 MSHR entries, a value based on the estimated number of MSHR entries in a Fermi GPU [18]. A CNN-P unit also has local memory to store the partition's state and input. The size of local memory is proportional to the number of cells in the CNN array (2 * number of CNN cells). For the case of 128x128, this translates to a local memory of size 32 KB. The local memory is viewed as a scratchpad memory that is managed by the scheduler. Finally, we use DDR3 memory specified by the DRAMSim2 config file $DDR3\_micron\_16M\_8B\_x8\_sg15.ini$, which is an 8 bank system with timings of 10-10-10-24. The size of global memory was set at 2GB. This value was chosen since it is similar to common GPU memory sizes, as well as mobile memory sizes.

5.3. Virtual Time vs. Total Time

Directly comparing the total time between the Ideal CNN and SP-CNN is not very informative since SP-CNN's time will generally be scaled by the image to CNN array size ratio. Instead, we introduce the measurement of virtual time. Virtual time represents how much time a partition has actually processed on the CNN. For the Ideal CNN, this is equivalent to the actual time that has passed. For SP-CNN, if $|P|$ is the number of partitions, then after one iteration, even though $interval \times |P|$ total time units has passed, only $interval$ virtual time has passed since each partition was processed for $interval$ units. For algorithms 1 and 2, t and vt represent the total time passed and virtual time passed.

Due to the Early-Finish optimization, algorithm 2's computation of virtual time cannot be used. Instead, virtual convergence time is calculated according to equation 3. Essentially, the idea is that for any given iteration, the amount of virtual time that has passed is equivalent to the partition that took the longest to converge, or the partition that computed for the entire interval without reaching convergence.

$$virt\_conv\_time = \sum_{1}^{N} min(interval, max(\forall_{p \in Partitions} convTime(p,i))) \quad (3)$$

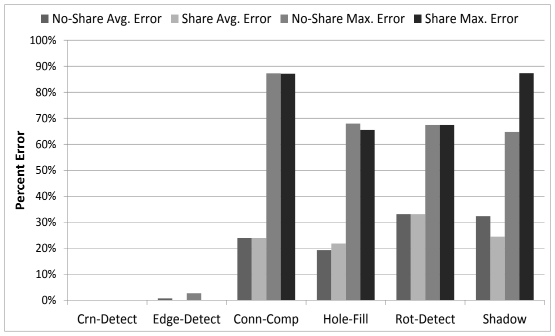

Figure 9: Average Error of Naive Multiplexing Mechanisms

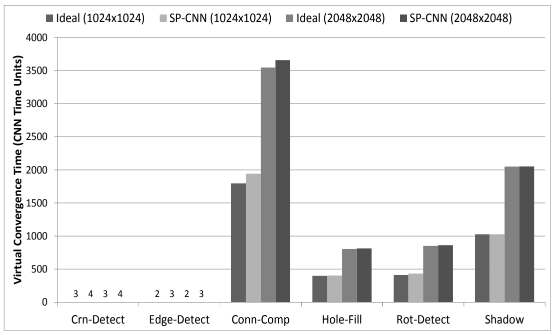

Figure 10: Avg. Virtual Conv. Time of Ideal CNN and SP-CNN

6. Results

6.1. Correctness

In Section 3.2, we showed one example on how naive multiplexing can lead to an incorrect output. Here, we present the results of applying the two naive mechanisms across our benchmark applications. For each application, we measure error as the total percentage of pixels different from the output of the test image applied on the naive multiplexing mechanism compared to the output on an ideal CNN (a CNN whose array is the same dimension as the input). Figure 9 shows the average and max error of applying either the Naive-No-Share and Naive-Share mechanisms on the ten 1024x1024 test images. Both mechanisms show an average error near or exceeding 10% for the 4 global benchmarks. Furthermore, the max error for these global benchmarks ranged from 40% to as high as 90%. Clearly, a naive multiplexing mechanism is ill-suited for global CNN applications.

For the local benchmarks of Edge Detection and Corner Detection, the error values are significantly lower. In the case of Naive-Share, with local type genes, the mechanism will provide an output equivalent to the ideal case for any test image suite since it essentially emulates the convolution operations that occur with local-type templates.

Finally, the SP-CNN mechanism always converged to the correct output for our test benchmarks and images, so we chose to exclude SP-CNN from Figure 9.

6.2. Virtual Convergence Time

Figure 10 illustrates the average virtual convergence time for the Ideal-CNN and our SP-CNN mechanism. As seen in the figure, the SP-CNN is competitive with the Ideal CNN for all our benchmarks. Since the Corner Detection and Edge Detection benchmarks are local applications, their convergence times are very small and are shown as data values.

6.3. Interval Period Effects

As stated before, one of the important contributions of this work is showing that certain CNN applications are robust enough to use interval values larger than 1 during the multiplexing execution. Therefore, we investigated the effect different interval times had on our set of benchmark applications. Choosing an effective interval period is important in limiting the convergence time of applications. Too small of an interval may lower virtual convergence time, but increases the overhead of data transfers. On the other hand, too large of an interval time can increase the virtual convergence time, and therefore also affect the total convergence time.

Figure 11 shows the average virtual convergence results of SP-CNN as the interval time increases, while Figure 12 shows the average total convergence time results when varying interval time. We do not include the results for local applications since their convergence times will not change with interval time. Finally, note that while the results shown are only for the 1024x1024 test images, the same patterns were evident the 2048x2048 test images.

When increasing the interval time, the virtual convergence time tends to increase. This is best seen in Connected Component, but it does occur to a small extent for the other global applications. The reason that growth is small for certain cases and large for others is most likely due to the Early-Finish optimization. From equation 3, note how the interval time only contributes to the virtual convergence time as long as there is some partition whose convergence time would have taken larger than the interval time. Therefore, as the interval time grows larger, it begins to contribute less and less to the virtual convergence time. Furthermore, at some point, the interval time will be large enough where the virtual convergence time remains the same even if the interval time continues to grow. For Connected Component, we can surmise that the partition convergence times tend to be larger than the interval

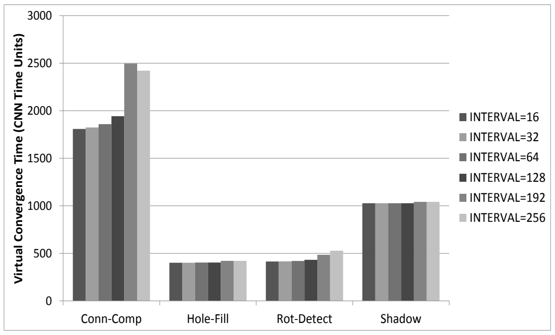

Figure 11: Effect of Interval Time on Virtual Convergence Time

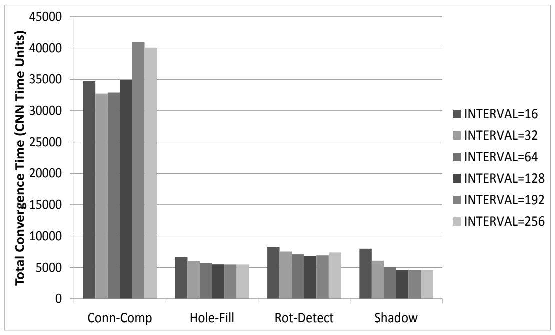

Figure 12: Effect of Interval Time on Total Convergence Time

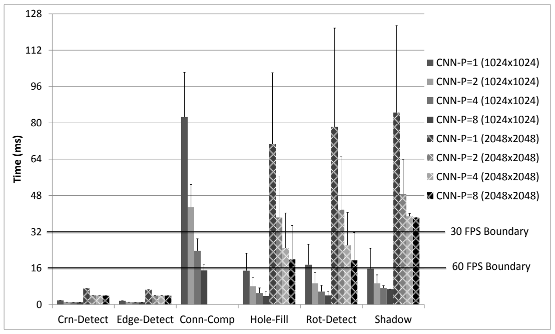

Figure 13: Average Execution Time of SP-CNN (Bar represent max execution time seen)

time, while for the other applications the partition convergence times was smaller than the larger interval times.

Figure 12 shows the effect of interval time on Total Convergence Time. In the cases of Connected Component and Rotation Detection, the growth in virtual convergence time also led to growth in the total convergence time. However, for the Hole Filling and Shadow Creator benchmarks, as well as for the smaller intervals in the Connected Component and Rotation Detector benchmarks, the total convergence time actually decreases when the interval time increases. While it may seem counter-intuitive for the total convergence time to decrease when the interval time/virtual convergence time is increasing, this effect is explained by the interaction between the Early-Finish optimization and the iterative process of SP-CNN.

Consider two interval times x and y, where $y = 2 \times x$. If the virtual convergence times for the two intervals are close or equal, then we can infer that two iterations at x is equivalent to one iteration at y. Now, consider the case where a partition has converged to the final output state earlier than other partitions in the image. Generally, for the remaining processing iterations, the partition remains converged so it only occupies the CNN-P for one CNN time unit. For the rest of the execution though, SP-CNN at x will perform twice as many iterations as y, so the converged partition will occupy the CNN-P twice as much at x than y. Since total convergence time is the sum of the total time each partition spends on a CNN-P, the total convergence time for x ends up larger than the time at interval time y. This means that in applications where partitions converge to the final output at different rates, the total convergence time will be smaller at larger interval times as long as the growth rate of the virtual convergence time is small. Furthermore, actual times will be even larger as the running of these already converged partitions contribute to a high memory cost with relatively little computational benefit.

Table 5 shows how the fast-propagation mechanism (Section 4.4.1) can reduce the total convergence times for applications such as Connected Component, Hole Filling, and Rotation Detector, showing speedups from 11% to 40%. For the local applications we see no speedups since no information is propagated through the cells. The Shadow Creator application also shows no speedup, because of the nature of the application. Instead, Shadow Creator is affected by traversal order (Section 3.3.3, since in this application most information propagates from right to left. In that case, using reverse-row-major order provides about a 13% speedup for the 1024x1024 images and a 30% speedup for the 2048x2048 images.

	Conn-Comp	Hole-Fill	Rot-Detect	Shadow
1024x1024	1.39	1.13	1.16	1.0
2048x2048	1.32	1.11	1.21	1.0

Table 5: Speedup of Fast Propagation over Slow Propagation

6.4. Timing Analysis with a DDR3 Memory System

So far, we have presented all our results in CNN time units. In this section, we provide a performance results by evaluating our mechanism on our timing simulator. Figure 13 shows the timing results for SP-CNN. With the 1024x1024 images, SP-CNN can perform below the 60 FPS boundary of for most applications other than Connected Component and the CNN-P=1 case of Rotation Detector. Note, that these are average times, and the bars show how some of the test images would cause the applications to pass the boundaries. In the case of 2048x2048 images, we are able to meet the 30 FPS boundary for most applications when the number of CNN-Ps is equal to 4 or 8. The exception is Shadow Creator, in which we cannot meet the 30 FPS boundary regardless of the number of CNN-Ps. However, note that when we run Shadow Creator, we are not using the Fast Propagation method that was shown to provide benefit over the base scheme.

6.5. Multiple CNN-P Units and Memory Effects

Since SP-CNN allows partitions to be processed in parallel, one might expect linear scaling in respect to the number of CNN-Ps. However, from Figure 14, we can see that none of the applications achieve linear scaling. The reason SP-CNN does not scale is because memory contention becomes a dominating factor, especially as the number of CNN-P grows to larger numbers. This can be seen in Figure 15, which illustrates the percentage of time a partition execution spends communicating to memory. As seen, for the benchmarks that show sub-linear scaling, memory communication accounts

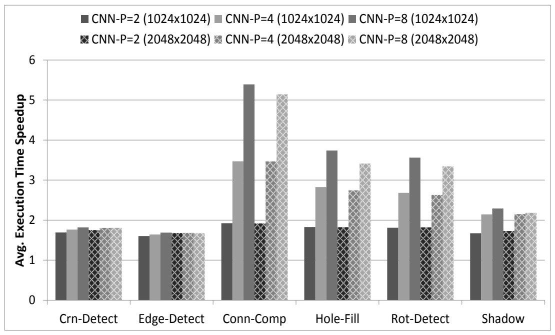

Figure 14: Execution Time Speedup with Multiple CNN-P Units

for over 50% of partition execution, and rapidly grows as the number of units increase. On the other hand, our most compute intensive benchmark, Connected Component, shows that the percentage of memory communication stays relatively constant as the number of units increase. This application is also the closest to showing linear scaling in the speedup results.

6.6. Prefetching

With SP-CNN, the scheduler can prefetch the input and state data for the next partition while the current partition is executing. Figure 16 shows the execution speedup of using prefetching versus no-prefetching on *N* CNN-Ps for the 1024x1024 images. Prefetching does not appear to provide as much benefit as multiple CNN-P units, with the best speedup being less than 1.8x. Furthermore, prefetching becomes much less effective as the number of CNN-Ps increases, with some cases even showing a slowdown. Again, this is due to memory contention from prefetching CNN-Ps and executing CNN-Ps. Prefetching also tended to perform better on local applications versus global. Intuitively, this is expected since local applications have very little computation.

6.7. Comparisons with Other Architectures

Table 6 lists the different architectures we compared SP-CNN against. In terms of applications, we created architecture-specific implementation for most of the benchmarks. For two benchmarks, Hole-Filling and Rotation Detector, our architec-

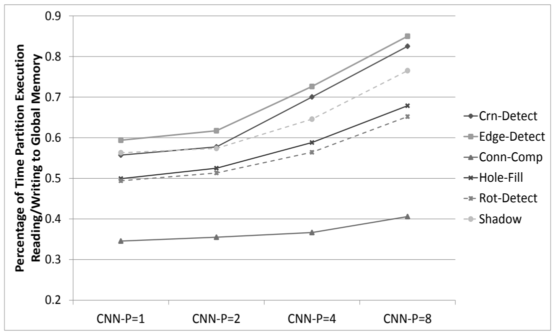

Figure 15: Avg. Percentage of Time Partition Execution Spends in Memory Communication

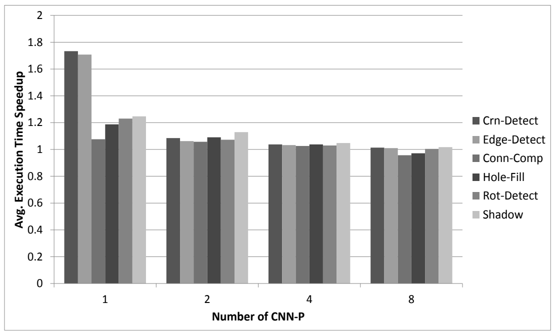

Figure 16: Execution Time Speedup with Prefetching

Name	Model	Power	Freq.	Tech.
CPU	Intel(R) Core (TM) i5-3550	77W (TDP)	3.3GHz	22nm
GPU	NVIDIA K1 mobile [20]	1.5W	900MHz	28nm
SP-CNN		0.73mW per PE , 35.6 μW per node	200MHz	45nm

Table 6: Other architectures and power numbers

ture specific implementations use an algorithm similar to the one employed the CNN mechanism. In our timing comparison on the 1024x1024 test images, seen in Figure 17, we also added the Ideal multiplexing (Ideal-Mul) mechanism (shown as the patterned bars). Ideal multiplexing is the same as SP-CNN where the interval time equals 1, and this is similar to the VAE architecture [11], which was designed for 130nm.

From Figure 17, we see that the Ideal-Mul mechanism hits our graph limit for the global applications. As expected, Ideal-Mul would take anywhere from 200 to 400 ms in executing these applications due to the memory transfer overhead. For the local applications, since the computation time is already very short for SP-CNN, the Ideal-Mul mechanism performs only slightly slower.

In comparison to the CPU and GPU architectures, the GPU architecture tends to perform the best for some applications. When the number of CNN-P equals 4, SP-CNN performs better than the GPU. For Connected Component and Shadow Creator, the CPU tends to perform best (though the GPU performs best for 2048x2048 images). As these are the two longest performing algorithms for SP-CNN, it was expected that the CPU or GPU would perform better. The GPU implementation only lost out when it came to the Hole-Filling and Rotation-Detector implementations. Figure 18 shows the energy usage for the different architectures with the 1024x1024 test images. As expected, while the CPU may perform quickly, the power consumption overweighs the time benefits. The GPU again shows the best results due to quick execution as well as its low power consumption design. Note that, the GPU uses the 28nm technology and it is a commercial product, while SP-CNN uses only the 45 nm technology[5] and the PE cell designs are

[5]45nm is the best technology that we have an access to.

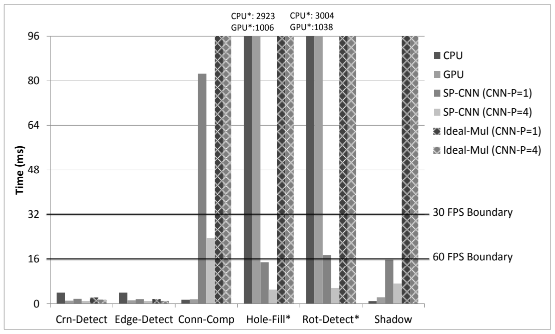

Figure 17: Timing Comparison for Various Architectures

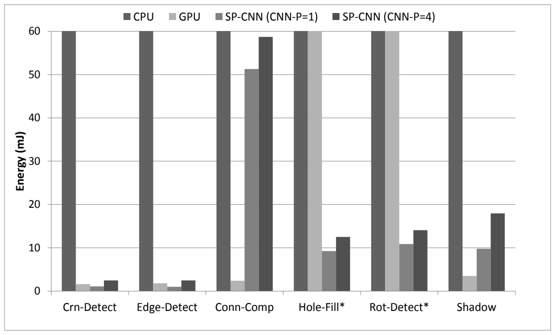

Figure 18: Energy Comparison for Various Architectures

not fully optimized. We used SPICE to perform power and timing analysis of SP-CNN. For the local applications, SP-CNN does have a better energy profile than the GPU when the number of CNN-Ps equals 1. For the global applications, execution time becomes the main factor in energy, where the GPU performs better on applications where our mechanism had a large execution time, and SP-CNN performing better when the GPU had a large execution time. In summary, SP-CNN shows fairly comparable energy consumption behavior with GPUs even with the penalty of an old generation of technology except for Connected Component. If SP-CNN can have a better circuit technology and optimized cell designs, such as for register files, we project that SP-CNN would consume much less energy than GPUs.

6.8. Limitations

SP-CNN depends on the CNN application being robust enough to handle the multiplexing scheme. As shown in previous sections, the choice of SP-CNN parameters such as interval and ordering can have different effects on the virtual and total convergence times. Currently, the values for these parameters must be determined by the application programmer, though future work may find that these parameters can be identified through simulations and profiling. Furthermore, while all of our benchmarks converged to the correct output under the various parameters, there are applications that never converge or converge to an incorrect solution when the interval time is greater than 1. Again, this is an evaluation that the application designer must make. For future work, we would like to develop mechanisms to identify applications suitable for SP-CNN, as well as to identify optimal parameters under different conditions.

7. Related Work

7.1. Multiplexing CNN

Applying time-multiplexing to the CNN has been studied by several researchers [9, 33, 11]. Their multiplexing mechanisms are similar to ideal multiplexing. However, as discussed previously, because of the communication overhead, ideal multiplexing is not scalable. On the contrary, SP-CNN provides the scalability and a programmable interface to support various multiplexing algorithm.

7.2. Handling Large Images

Another direction to support large images with smaller dimension of CNN cells is to utilize software algorithms. For example, the VAE chip processes large images by first preprocessing images with high-level algorithms such as object recognitions [12]. For local benchmarks such as Edge detection, a small dimension of CNN array can also handle large images. However, the SP-CNN can process any size image, without having multiple-levels of software algorithms.

7.3. Implementing CNN algorithms on GPUs

Several researchers have implemented CNN algorithms on GPUs [6, 23, 29]. Using GPUs SIMD feature, the CNN algorithms can get a great speedup over CPUs and it can support arbitrary image sizes. However, implementing CNN algorithms on GPUs is extremely slow because of CNN emulation.

8. Conclusions

The importance of designing specialized accelerators for various sensors has been growing. As one of the possible programmable image processing architectures, we focus on the traditional CNN paradigm due to its advantages in power and applications. In this paper, we proposed a scalable and programmable CNN, SP-CNN, to overcome the scalability problem in the traditional CNN paradigms. SP-CNN can handle any resolution image with a fixed dimension of CNN array. Our results show that the proposed multiplexing algorithms can successfully scale robust CNN applications while using smaller hardware resource. We also propose a microrarchitecture for SP-CNN and provide performance and power analysis. The performance results show that naive or ideal multiplexing algorithms have high errors and high performance degradations, but SP-CNN provides real-time images. We also compared SP-CNN with latest commercial GPUs and CPU. Even with a older generation of circuit technology, SP-CNN shows comparable energy numbers with commercial products and even much better energy efficiency and performance benefits on two benchmarks.

References

[1] T. Choi, B. Shi, and K. Boahen, "An on-off orientation selective address event representation image transceiver chip," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 51, no. 2, pp. 342–353, 2004.

[2] L. Chua, *Cnn: A Paradigm for Complexity*, 1998.

[3] L. Chua and L. Yang, "Cellular neural networks: applications," *Circuits and Systems, IEEE Transactions on*, vol. 35, no. 10, pp. 1273–1290, 1988.

[4] ——, "Cellular neural networks: applications," *Circuits and Systems, IEEE Transactions on*, vol. 35, no. 10, pp. 1273–1290, 1988.

[5] ——, "Cellular neural networks: theory," *Circuits and Systems, IEEE Transactions on*, vol. 35, no. 10, pp. 1257–1272, 1988.

[6] R. Dolan and G. DeSouza, "Gpu-based simulation of cellular neural networks for image processing," in *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, June 2009, pp. 730–735.

[7] P. Dudek, D. Barr, A. Lopich, and S. Carey, "Demonstration of real-time image processing on the scamp-3 vision system," in *Cellular Neural Networks and Their Applications, 2006. CNNA '06. 10th International Workshop on*, 2006, pp. 1–1.

[8] J. Flak, M. Laiho, A. Paasio, and K. Halonen, "Dense CMOS implementation of a binary-programmable cellular neural network: Research articles," *Int. J. Circuit Theory Appl.*, vol. 34, no. 4, pp. 429–443, Jul. 2006. [Online]. Available: http://dx.doi.org/10.1002/cta.v34:4

[9] G. Han, J. de Gyvez, and E. Sanchez-Sinencio, "Optimal manufacturable cnn array size for time multiplexing schemes," in *Cellular Neural Networks and their Applications, 1996. CNNA-96. Proceedings., 1996 Fourth IEEE International Workshop on*, 1996, pp. 387–392.

[10] P. Kinget and M. Steyaert, "A programmable analog cellular neural network CMOS chip for high speed image processing," *Solid-State Circuits, IEEE Journal of*, vol. 30, no. 3, pp. 235–243, 1995.

[11] S. Lee, M. Kim, K. Kim, J.-Y. Kim, and H.-J. Yoo, "24-GOPS 4.5- mm^2 digital cellular neural network for rapid visual attention in an object-recognition soc," *Neural Networks, IEEE Transactions on*, vol. 22, no. 1, pp. 64–73, 2011.

[12] S. Lee, J. Oh, J. Park, J. Kwon, M. Kim, and H.-J. Yoo, "A 345 mw heterogeneous many-core processor with an intelligent inference engine for robust object recognition," *J. Solid-State Circuits*, vol. 46, no. 1, pp. 42–51, 2011.

[13] G. Linan, R. Dominguez-Castro, S. Espejo, and A. Rodriguez-Vazquez, "ACE16K: an advanced focal-plane analog programmable array processor," in *Solid-State Circuits Conference, 2001. ESSCIRC 2001. Proceedings of the 27th European*, 2001, pp. 201–204.

[14] R. Matei, 2009-10-01. [Online]. Available: http://www.intechopen.com/books/export/citation/BibTex/advanced-technologies/new-model-and-applications-of-cellular-neural-networks-in-image-processing

[15] T. Matsumoto, L. Chua, and R. Furukawa, "Cnn cloning template: hole-filler," *Circuits and Systems, IEEE Transactions on*, vol. 37, no. 5, pp. 635–638, 1990.

[16] T. Matsumoto, L. Chua, and H. Suzuki, "CNN cloning template: connected component detector," *Circuits and Systems, IEEE Transactions on*, vol. 37, no. 5, pp. 633–635, 1990.

[17] ——, "CNN cloning template: shadow detector," *Circuits and Systems, IEEE Transactions on*, vol. 37, no. 8, pp. 1070–1073, 1990.

[18] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal, "A detailed gpu cache model based on reuse distance theory," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb 2014, pp. 37–48.

[19] NVIDIA, "Geforce 8800 graphics processors," http://www.nvidia.com/page/geforce_8800.html.

[20] "Tegra K1 Processor Specifications," http://www.nvidia.com/object/tegra-k1-processor.html, NVIDIA, 2014.

[21] A. Paasio, A. Kananen, K. Halonen, and V. Porra, "A QCIF resolution binary I/O CNN-UM chip," *J. VLSI Signal Process. Syst.*, vol. 23, no. 2/3, pp. 281–290, Nov. 1999. [Online]. Available: http://dx.doi.org.prx.library.gatech.edu/10.1023/A:1008188900876

[22] V. Podlozhnyuk, http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf, NVIDIA, 2007.

[23] S. Potluri, A. Fasih, L. Vutukuru, F. Al Machot, and K. Kyamakya, "Cnn based high performance computing for real time image processing on gpu," in *Nonlinear Dynamics and Synchronization (INDS) 16th Int'l Symposium on Theoretical Electrical Engineering (ISTET), 2011 Joint 3rd Int'l Workshop on*, July 2011, pp. 1–7.

[24] J. A. Ramirez-Quintana, M. I. Chacon-Murguia, and J. F. Chacon-Hinojos, "Artificial neural image processing applications: A survey," *Engineering Letters*, vol. 20, no. 1, pp. 68–80, 2012.

[25] A. Rodriguez-Vazquez, G. Linan-Cembrano, L. Carranza, E. Roca-Moreno, R. Carmona-Galan, F. Jimenez-Garrido, R. Dominguez-Castro, and S. Meana, "ACE16k: the third generation of mixed-signal SIMD-CNN ACE chips toward VSoCs," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 51, no. 5, pp. 851–863, 2004.

[26] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16 –19, jan.-june 2011.

[27] T. Roska and L. Chua, "The CNN universal machine: an analogic array computer," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 40, no. 3, pp. 163–173, 1993.

[28] C. Sensory and W. C. Laboratory, "Software library for cellular wave computing engines," http://cnn-technology.itk.ppke.hu/Template_library_v3.1.pdf, the Computer and Automation Research Inst., Hungarian Academy of Sciences and the Jedlik Laboratories of the Pázmány P. Catholic University.

[29] B. G. Soós, A. Rák, J. Veres, and G. Cserey, "Gpu boosted cnn simulator library for graphical flow-based programmability," *EURASIP J. Adv. Signal Process*, vol. 2009, pp. 8:1–8:11, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1155/2009/930619

[30] H. Tanaka, K. Tanno, H. Tamura, and K. Murao, "Design of CNN cell with low-power variable-gm ota and its application," in *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, 2008, pp. 392–395.

[31] R. S. Vaddi, L. N. P. Boggavarapu, H. D. Vankayalapati, and K. R. Anne, "Cellular neural network based pre-processing for localization of non standard licence plate," in *Electronics Computer Technology (ICECT), 2011 3rd International Conference on*, vol. 1, 2011, pp. 407–411.

[32] D. L. Vilariño, V. M. Brea, D. Cabello, and J. M. Pardo, "Discrete-time cnn for image segmentation by active contours," *Pattern Recogn. Lett.*, vol. 19, no. 8, pp. 721–734, Jun. 1998. [Online]. Available: http://dx.doi.org/10.1016/S0167-8655(98)00050-6

[33] L. Wang, J. de Gyvez, and E. Sanchez-Sinencio, "Time multiplexed color image processing based on a cnn with cell-state outputs," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 6, no. 2, pp. 314–322, 1998.

[34] WCCFTECH, "Intel aims for mobile space with the 32nm atom z2420 lexington soc and 22nm bay trail soc," http://wccftech.com/intel-aims-mobile-space-32nm-atom-z2420/, Intel.

[35] A. Zarandy, P. Keresztes, T. Roska, and P. Szolgay, "An emulated digital architecture implementing the cnn universal machine," in *Cellular Neural Networks and Their Applications Proceedings, 1998 Fifth IEEE International Workshop on*, 1998, pp. 249–252.