Proceedings of International Joint Conference on Neural Networks, Atlanta, Georgia, USA, June 14-19, 2009

GPU-Based Simulation of Cellular Neural Networks for Image Processing

Ryanne Dolan and Guilherme DeSouza
Vision-Guided and Intelligent Robotics Laboratory
University of Missouri, Columbia, MO USA

Abstract—**The inherent massive parallelism of cellular neural networks makes them an ideal computational platform for kernel-based algorithms and image processing. General-purpose GPUs provide similar massive parallelism, but it can be difficult to design algorithms to make optimal use of the hardware. The presented research includes a GPU abstraction based on cellular neural networks. The abstraction offers a simplified view of massively parallel computation which remains reasonably efficient. An image processing library with visualization software has been developed to showcase the flexibility and power of cellular computation on GPUs. Benchmarks of the library indicate that commodity GPUs can be used to significantly accelerate CNN research and offer a viable alternative to CPU-based image processing algorithms.**

I. Introduction

Cellular neural networks (CNNs) are an attractive platform for parallel image processing due to their ability to perform per-pixel operations in parallel. The research presented here aims to target commodity graphics processing units (GPUs) for efficient simulation and visualization of CNNs. GPUs are readily available and provide a massively parallel platform ideally suited to the simulation of CNNs. Simulating CNNs on commodity GPU hardware allows for the straightforward application of existing CNN image processing algorithms without special CNN hardware. Additionally, CNN visualization software provides a convenient platform for further research on CNNs [13], [9] and similar networks, including artificial neural networks and continuous cellular automata.

It is difficult to structure algorithms to take advantage of massively parallel processors and GPUs. Cellular automata, neural networks, and CNNs are abstract computing machines which make use of networks of processing elements following simple rules. Some of these systems can be implemented efficiently in hardware, but it can be difficult to translate their parallelism into an efficient software implementation for simulation on commodity hardware.

CNNs have been shown to be especially adept at image processing tasks. The increasing popularity of dedicated GPU hardware prompts the following questions: Can we make use of commodity GPU hardware to simulate CNNs for image processing? Can a GPU-based cellular image processing library outperform CPU-based image processing implementations like OpenCV?

Simple GPU-based CNN simulations have been demonstrated that run much faster than CPU-based CNN simulations [7], [6]. The research presented here examines whether this improvement translates to faster image processing algorithms compared to traditional CPU-based algorithms.

	CNNs	**ANNs**
topology	uniform 2D grid	usually feed-forward
processing element	dynamic equations	nonlinear weighted sum
common uses	image processing	classification, control

Table I
CNNs, ANNs compared

II. Background Information

A. Cellular Neural Networks

Cellular neural networks (CNNs) are similar to well-known artificial neural networks (ANNs) in that they are composed of many distributed processing elements called “cells”, which are connected in a network; however, there are several important differences between CNNs and ANNs (see Table I). Instead of the usual feed-forward, layered architecture seen in many types of neural networks, CNNs were designed to operate in a two-dimensional grid where each processing element (cell) is connected to neighboring cells in the grid. The cells comprising a CNN communicate by sending signals to neighboring cells in a manner similar to ANNs, but the signals are processed by each cell in a unique way. Specifically, CNN cells maintain a state which evolves through time due to differential (or difference) equations dependent on the cell's inputs and feedback.

B. CNN Topology

CNNs are composed of many cells arranged in a grid, M. To simplify discussion, we will assume these grids are always square with dimensions $m \times m$ for m^2 cells. Each cell in the grid is denoted v_{ij} for $i, j \in M$. Thus each cell is labeled from v_{11} to v_{mm}. We define two types of cell depending on their location in the grid: *inner* cells and *boundary* cells. Boundary cells occur near the edges of the grid; inner cells occur everywhere else. Boundary cells necessarily have different properties than inner cells because they are connected to fewer neighboring cells.

Each inner cell is the center of a neighborhood N_{ij} of $n \times n$ cells. By this definition, n must be odd and is usually $n = 3$. By convention, each cell in a given neighborhood is assigned an index k from $1..n^2$, with $k = 1$ denoting the center cell, as shown in Figure 1. Thus any given center cell $v_{ij} = v_1$ belongs

978-1-4244-3553-1/09/$25.00 ©2009 IEEE

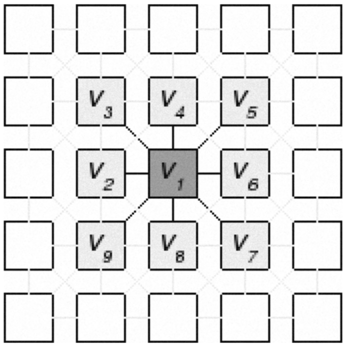

Figure 1. CNN neighborhood

to the neighborhood $N_{ij} = N = \{v_1, v_2, ..v_{n^2}\}$, where we have dropped the i, j indexes for cleaner notation.

C. The CNN Cell

Each cell v_k is composed of the following variables:

input u_k a constant scalar parameter, independent of the cell dynamics

state $x_k(t)$ scalar variable which evolves over time with initial condition $x_k(0)$

output $y_k(x_k(t))$ scalar function of $x_k(t)$

Additionally, each cell in the network is influenced by a scalar *bias* parameter z, which is uniform throughout the network. The input, bias, and initial condition $x_k(0)$ are all independent of the cell dynamics and are specified *a priori*. Using this convention, the STATE EQUATION for center cell v_1 can be described as follows:

$$\dot{x}_1 = -x_1 + \sum_{k \in N} a_k y_k + \sum_{k \in N} b_k u_k + z \tag{1}$$

with coefficients a_k and b_k as described in II-D. The OUTPUT EQUATION for v_k is defined as:

$$y_k(x_k) = \frac{1}{2}\left(|x_k + 1| - |x_k - 1|\right) \tag{2}$$

These equations alone are sufficient for finding the time evolution of $x_k(t)$ and $y_k(x_k(t))$ for each cell in the grid, given initial conditions $x_k(0)$ and parameters u_k and z.

Boundary cells must be treated separately, in a manner usually called the *boundary condition* of a given CNN. Several types of boundary conditions exist, and the choice of boundary condition may affect the behavior of the network as a whole [4]; however, for the remainder of our discussion we will assume a static boundary condition in which each boundary cell performs no processing and maintains a constant state. Specifically, for boundary cells v_b, we will define $x_b \equiv 0$, $y_b \equiv 0$ everywhere.

D. CNN Templates

The coefficients a_k and b_k from (1) form vectors $\vec{a}$ and $\vec{b}$ of length n^2. Each coefficient corresponds to a neighboring cell v_k. By arranging the coefficients according to the shape of the neighborhood N (which is square), we get matrices A and B, called CNN *templates*. For example, if $n = 3$ the neighborhood is a 3×3 square, yielding templates as follows:

$$A = \begin{bmatrix} a_3 & a_4 & a_5 \\ a_2 & a_1 & a_6 \\ a_9 & a_8 & a_7 \end{bmatrix} \tag{3}$$

$$B = \begin{bmatrix} b_3 & b_4 & b_5 \\ b_2 & b_1 & b_6 \\ b_9 & b_8 & b_7 \end{bmatrix} \tag{4}$$

The template A is called the FEEDBACK TEMPLATE because it gates the feedback from the neighborhood's previous states. Similarly, B is called the FEED-FORWARD TEMPLATE because it gates the constant input u_k, which is known initially. The templates A and B are similar to *weights* in the nomenclature of ANNs in that they gate the signals sent between connected cells. A larger a_k or b_k signifies a "stronger" connection between cells v_1 and v_k. Unlike ANNs though, the templates A and B associated with any inner cell are uniform across all cells in the network. In other words, all inner cells use the same templates A and B regardless of their location in the network. This can be contrasted with ANNs, in which a weight vector must be found for each neuron in the network.

The template matrices A and B, together with the bias term z, specify the behavior of the CNN for a given set of initial conditions and inputs. We can organize these parameters into a single vector as follows:

$$G = \langle z, a_1, a_2, .., a_k, b_1, b_2, .., b_k \rangle \tag{5}$$

This vector is usually called a CNN *gene*, owing to the fact that evolutionary algorithms can be employed to discover these parameters.

E. CNN-Based Image Processing

Image processing with CNNs is possible when the state and input of each cell is interpreted as a pixel in an image. It is easy to imagine the input and initial state images to have visual information (as is usually the case with images) such as colors and shapes. The output image might then be some different form of visual information, such as an edge map or distance transform. In this case, we can consider a CNN as an image processor. Hardware "vision chips" have been designed for this purpose [5], [8].

The ability to transform images in complex, nonlinear ways makes CNNs ideal for many image processing and computer vision tasks. In particular, spatially invariant image filters are well-suited to CNN implementations because of the CNN's ability to apply a nonlinear function to every pixel in an image simultaneously. However, spatially sensitive image processing with CNNs is also possible due to the ability of signals to propagate throughout the CNN. In other words, CNNs are able to perform both local and global image operations. Some image processing algorithms that have been implemented for CNNs include contrast enhancement, edge detection, text extraction, skeletonization, and smoothing [12], [15].

III. CNN Simulation

A. Baseline Single Processor Implementation

CNNs were designed to be implemented with custom hardware (vision chips); however, in order to experiment with CNN-based algorithms, an easy way to test them is obviously required. Hardware CNN implementations can be prohibitive in cost and availability. Usually, a software simulator is used to prototype and discover new CNN algorithms (via genetic algorithms, for example).

CNNs have been implemented on many platforms, including the PC [13], [9], cluster architectures [14], custom hardware [5], [8], and GPUs. The current research aims to provide a common software interface to some of these platforms, allowing the same CNN algorithms to run with the same code on single-core, multi-core, and graphics processors.

Largely as a basis for comparison, a serial CNN simulator was developed for execution on a single processor. This baseline implementation is derived directly from (1) and (2) using the Euler method of numerical integration. At each timestep, all cells are updated in succession according to Algorithm 1. The algorithm is run for a specified number of iterations, allowing the cell states to settle into steady-state values.

Notice that an obvious optimization has been made in the algorithm. All feedforward terms are calculated once at the start of the algorithm as an extra initialization step. This significantly reduces the run-time of the computation, since it eliminates nearly half of the multiplies. The optimization is possible because the input image and z are both constant for the duration of the simulation. This fact allows us to rewrite (1) as follows:

$$\dot{x}_1 = x_1 + \sum_{k \in N} a_k y_k + c_1 \qquad (6)$$

with

$$c_1 = \sum_{k \in N} b_k u_k + z \qquad (7)$$

The c_1 term for each cell is calculated at the start of the algorithm, generating a *feed-forward image*.

The code has been implemented in C using the core data structures provided by the Open Computer Vision Library from Intel (OpenCV) [2]. The code is written as an extension to the OpenCV library, since it requires no other dependencies and forms the basis of an OpenCV-like CNN image processing library. From OpenCV, the CNN simulator has inherited the ability to operate on integer, floating point, and double precision values with the same small codebase. Additionally, the simulator can operate on four color channels in a single pass, allowing for manipulation of color images. Processing a four-channel image with the library is functionally equivalent to processing four separate images with four identical CNNs.

The CNN simulator provides parameters for template size n, time step ΔT, and end-time (settling time) T, accommodating a large number of standard CNN algorithms. The code could be further generalized by adding support for additional output equations $y_k(x_k)$, such as those corresponding to the "universal binary neuron" and "multi-valued binary neuron" [1].

Algorithm 1 serial CNN simulator

```
-- initialize cell states
for each cell v[i,j] in M do
    X[i,j] = X0[i,j]
end
-- calculate feed-forward image BUz
for each cell v[i,j] in M do
    BUz[i,j] = z
    for each neighbor v[k]
       in N[i,j] do
        BUz[i,j] = BUz[i,j]+B[k]*U[k]
    end
end
-- perform numerical integration
for t = 0 to T by DeltaT do
    -- calculate cell outputs
    for each cell v[i,j] in M do
        Y[i,j] = 0.5*(abs(X[i,j]+1) -
           abs(X[i,j]-1))
    end
    -- calculate cell state deltas
    for each cell v[i,j] in M do
        Dx[i,j] = -X[i,j] + BUz[i,j]
        for each neighbor v[k]
           in N[i,j] do
            Dx[i,j] += A[k] * Y[k]
        end
    end
    -- update states (Euler method)
    for each cell v[i,j] in M do
        X[i,j] += DeltaT * Dx[i,j]
    end
end
```

It should be noted that the CNN simulator has been designed strictly for continuous-time CNNs. Other CNN derivatives and variations, such as the generalized cellular automata, are beyond the scope of this research; however, modification of the source for these purposes would be straightforward.

A few simple CNN algorithms were run using the baseline CNN simulator code to verify that the implementation is correct and working. One such algorithm was taken from [15] and performs edge detection on the input image (see Figure 2).

B. Multi-Core Implementation

A second implementation was made, based on the first, to parallelize the simulation on multiple processors, specifically multi-core processors. The implementation relies on shared memory and the POSIX standard thread model (pthreads). Interestingly, no mutual exclusions (mutexes), semaphores, or other synchronization methods were required, making for a very simple implementation.

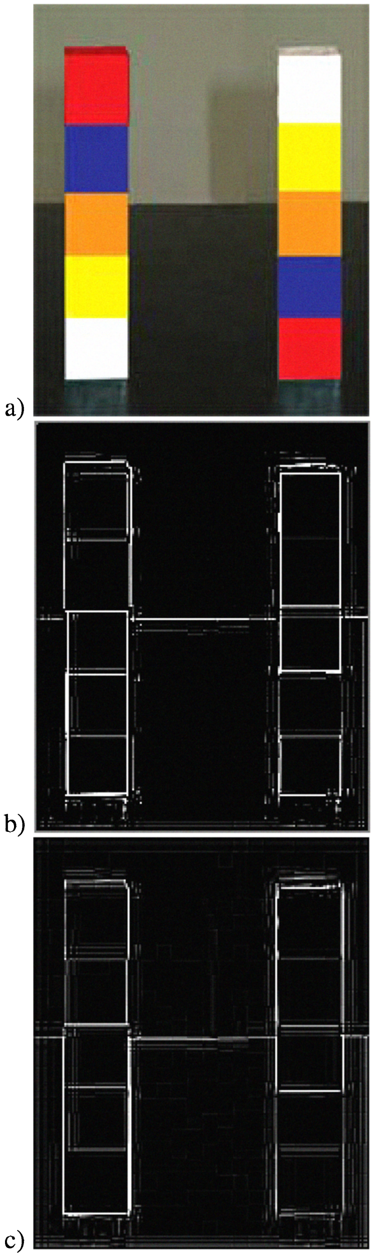

Figure 2. edge detector output

a) input image;
b) steady-state output of CNN inner edge detector described in [15];
c) output of OpenCV's Laplace transform for comparison

The multi-core implementation relies on the concept of a *kernel function*, a simple function executed in many parallel instances, each with different inputs. For example, kernel functions might be used to increment every integer in a list, saturate each pixel in an image (applying the sat function), or update each cell in a CNN.

The implementation uses four kernel functions:

- the feed-forward kernel, which computes the feed-forward image from U, B, and z according to (7),
- the feedback kernel, which computes $\dot{x}_1$ from $y_1(t)$, A, and the feed-forward image,
- the feedback integration kernel, which uses Euler integration to compute $x_1(t)$ from $x_1(0)$ and $\dot{x}_1$,
- and the output kernel, which computes $y_1(t)$ from $x_1(t)$.

Algorithm 2 kernel launch function

```
function launch_kernel (
       kernel_function, inputs
    )
    -- array to store thread handles
    local threads = {}
    -- array to store results
    local outputs = {}
    -- launch threads
    for i = 1, N_CORES do
        threads [i] = pthread_create (
           kernel_function, inputs [i]
        )
    end
    -- wait for threads to terminate
    for i = 1, N_CORES do
        outputs [i] = pthread_join (
           threads [i]
        )
    end
    return outputs
end
```

Each instance of a kernel operates on one cell at a time. Conceptually, there is one kernel instance for each cell, and when a kernel function is "launched", each kernel instance executes in parallel on its respective cell. In this way all cells are updated simultaneously using the same kernel function. The kernels are launched by a generic "kernel launch function" described by Algorithm 2.

In practice, multi-core processors have too few cores to execute all kernel instances in parallel, so instead of spawning one thread per kernel instance, the implementation spawns one thread per core and divides kernel instances among these threads. Each thread is responsible for executing its set of kernel instances in series. Since each core has its own thread, the processor is saturated without introducing unnecessary context switches.

C. CUDA Implementation

Lastly, a GPU-based CNN simulator was implemented using the CUDA platform from NVIDIA. CUDA's cellular architecture and data-parallel programming model is perfectly suited to simulation of CNNs, since each cell can be processed by a dedicated thread. CUDA can process thousands of threads efficiently, and therefore can processes many of the cells in a large CNN simultaneously [3]. Additionally, the fast shared memory available on CUDA GPUs enables neighboring cells to communicate as required.

cvLaplace	CPU CNN	dual-core CNN	CUDA GPU
0.10 s	7 mins	5 mins	0.11 s

Table II
RUN-TIME COMPARISON OF 5X5 KERNEL EDGE DETECTION ALGORITHMS

CNNs have been implemented using GPUs in the recent past using *shaders* to modify the GPU's rendering pipeline [7]. This is a significantly less convenient approach, requiring the programmer to formulate the algorithm in terms of pixels, textures, vertexes, and other graphics primitives. CUDA offers a much more flexible platform, which allows for a CNN implementation which follows directly from the multi-core version discussed above.

Currently, CUDA GPUs only support single-precision floating point operations. This limitation severely restricts the CUDA implementation of the simulator compared to the single- and multi-core implementations, which operate on several different data types. This discrepancy has necessitated that the CUDA implementation be developed separately from the single- and multi-core versions, though the basic algorithm remains the same. Also, the CUDA implementation does not use OpenCV's data structures since they are not supported by the hardware. The CUDA implementation can still be used alongside OpenCV, but only single-precision floating point arrays can be passed to the GPU; therefore, any OpenCV data structures must be converted accordingly. As a side effect of this departure from OpenCV, the CUDA implementation only operates on one channel at a time. Color images must be sliced and processed one channel at a time. It is hoped that the improved run-time of the CUDA implementation compensates for these shortcomings.

The GPU implementation uses kernels that are functionally equivalent to the multi-core kernels described in the previous section. The CUDA library includes kernel launching functions that replace Algorithm 2. Otherwise, the multi-core and GPU implementations are very similar in structure. This is a testament to how easy it can be to move from a multi-core program to a GPU-based program, especially compared to the prior use of shaders.

IV. RESULTS

None of the three implementations describe above are particularly optimized; in most cases, simplicity and readability of the code were emphasized rather than speed of execution. In this regard, it is somewhat superficial to compare the run-times of these implementations with other CNN simulators; however, it is instructive to compare between the three implementations, since we hope to see that GPUs have helped significantly without changing the CNN algorithm.

Images approximately 195x195 in size were used to compare the three implementations. While relatively modest in size, these images illuminate the shortcomings of CPU-based CNN simulation and image processing. The results in Table II illustrate two significant points: first, that CNN simulation on CPUs is indeed problematic, even at modest image sizes; and second, that GPUs enable CNN simulation at a pace comparable to even the simplest image processing algorithms. Thus, the library presented here gives CNN-based image processing research a chance to catch up with traditional CPU-based research. Since the CUDA implementation has a run-time on the same order of magnitude as the equivalent CPU-based algorithm, we can rightly assume that a more optimized CUDA implementation (and a faster GPU) could potentially outpace the CPU algorithm. Indeed, current research suggests that GPUs will get faster in the coming years while CPUs have largely reached their maximum speed potential, so a tie between CPU and GPU for a particular algorithm today might very well mean a win for the GPU in a year or so [10], [3], [11].

The implementations demonstrated here focused on simplicity and strove for a similar structure on all three platforms. Despite this, the GPU-based implementation is surprisingly fast. Optimization of the GPU-based CNN simulator is required, but GPU optimization is rarely straightforward with the current technology. Incidentally, the implementation presented here exhibits similar run-times as another GPU-based CNN simulator, which claims to be optimized for the hardware [6]. Clearly, further research is required to find effective optimization techniques applicable to CNN simulation.

V. CONCLUSION

The GPU-based CNN simulation library presented here offers a significant performance gain over CPU-based simulators. More importantly, the foregoing discussion indicates that GPU-based CNN simulation has immense potential for image processing, especially as GPU throughput increases in the months and years to come. Further optimizations need to be made for this CNN image processing library to be an attractive alternative to highly optimized CPU-based image processing libraries like OpenCV; however, the initial results presented here are encouraging.

REFERENCES

[1] I.N. Aizenberg. *Multi-Valued and Universal Binary Neurons: Theory, Learning, and Applications.* Kluwer Academic Publishers, 2000.

[2] G. Bradski. OpenCV: Examples of use and new applications in stereo, recognition and tracking. In *Proc. Intern. Conf. on Vision Interface (VI 2002).*

[3] I. Buck and GPU NVIDIA. GPU Computing: Programming a Massively Parallel Processor. *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pages 17–17, 2007.

[4] R. Dogaru. *Universality and emergent computation in cellular neural networks.* World Scientific River Edge, NJ, 2003.

[5] A. Dupret, J.O. Klein, and A. Nshare. A programmable vision chip for cnn based algorithms. *Cellular Neural Networks and Their Applications, 2000. (CNNA 2000). Proceedings of the 2000 6th IEEE International Workshop on*, pages 207–212, 2000.

[6] A. Fernandez, R. San Martin, E. Farguell, and G.E. Pazienza. Cellular neural networks simulation on a parallel graphics processing unit. *Cellular Neural Networks and Their Applications, 2008. CNNA 2008. 11th International Workshop on*, pages 208–212, July 2008.

[7] T.Y. Ho, P.M. Lam, and C.S. Leung. Parallelization of cellular neural networks on GPU. *Pattern Recognition*, 2008.

[8] P. Kinget and MSJ Steyaert. A programmable analog cellular neural network CMOS chip for highspeed image processing. *Solid-State Circuits, IEEE Journal of*, 30(3):235–243, 1995.

[9] C.C. Lee and JP de Gyvez. Single-layer CNN simulator. *Circuits and Systems, 1994. ISCAS'94., 1994 IEEE International Symposium on*, 6.

[10] D. Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, pages 836–838, May 2008.

[11] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. 2008.

[12] T. Nishio and Y. Nishio. Image processing using periodic pattern formation in cellular neural networks. *Circuit Theory and Design, 2005. Proceedings of the 2005 European Conference on*, 3:III/85–III/88 vol. 3, Aug.-2 Sept. 2005.

[13] JE Varrientos and E. Sanchez-Sinencio. CELLSIM: a cellular neural network simulator for the personal computer. *Circuits and Systems, 1992., Proceedings of the 35th Midwest Symposium on*, pages 1384–1387, 1992.

[14] T. Weishaupl and E. Schikuta. Parallelization of cellular neural networks for image processing on cluster architectures. *Parallel Processing Workshops, 2003. Proceedings. 2003 International Conference on*, pages 191–196, 2003.

[15] T. Yang. *Cellular Neural Networks and Image Processing*. Nova Science Publishers, 2002.