CELLULAR NEURAL NETWORK VIRTUAL MACHINE FOR GRAPHICS HARDWARE WITH APPLICATIONS IN IMAGE PROCESSING

A Thesis presented to the Faculty of the Graduate School at the University of Missouri

---

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

---

by

RYANNE DOLAN

Dr. Guilherme DeSouza, Thesis Supervisor

MAY 2009

The undersigned, appointed by the dean of the Graduate School, have examined the thesis entitled

CELLULAR NEURAL NETWORK VIRTUAL MACHINE FOR GRAPHICS HARDWARE WITH APPLICTIONS IN IMAGE PROCESSING

presented by Ryanne Dolan, a candidate for the degree of Master of Science, and hereby certify that, in their opinion, it is worthy of acceptance.

____

Professor Guilherme DeSouza

____

Professor Tony Han

____

Professor Youssef Saab

ACKNOWLEDGEMENTS

I would like to thank Dr. DeSouza for advising me during my research and studies. I would also like to thank Yuanqiang "Evan" Dong for inspiring me to explore GPUs in the first place.

TABLE OF CONTENTS

List of Figures

List of Tables

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 computation using cellular neural networks. The abstraction offers a simplified view of massively parallel computation which remains universal and reasonably efficient. An image processing library with visualization software has been developed using the abstraction to showcase the flexibility and power of cellular computation on GPUs. A simple virtual machine and language is presented to manipulate images using the library for single-core, multi-core, and GPU backends.

Part I

INTRODUCTION

1 Introduction to Research Topic

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 [26, 15] 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 [6, 14, 21, 16, 31]. 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? Can CNN simulation and visualization tools make it easier to design image processing algorithms in terms of CNNs compared to more traditional programming paradigms?

Simple GPU-based CNN simulations have been demonstrated that run much faster than CPU-based CNN simulations [13, 11]. The thesis supported here is that this improvement can translate to faster and easier image processing algorithms compared to traditional CPU-based algorithms.

1.1 Research Goals and Intended Contributions

This research addresses the difficulty of developing software for GPUs and other massively parallel architectures by abstracting the hardware as arrays of configurable CNN cells. The software described here enables programmers to implement efficient visual and cellular computations using the CNN paradigm, while leveraging the parallelism of GPU hardware. Introducing an efficient CNN-based abstraction of GPU computation should encourage the study of massively parallel computation using tools and methods of dynamic systems theory and abstract automata theory which have shown great utility in CNN research [6, 8]. This abstraction makes GPU programming approachable from a mathematical perspective such that engineers and scientists can utilize GPU parallelism without understanding the underlying hardware.

A key advantage of designing algorithms in terms of CNNs is that changes in the underlying hardware do not affect the high-level CNN algorithm. In other words, once a CNN-based algorithm is designed, the same algorithm can be simulated on GPUs, distributed over a computer cluster, implemented directly in specialized digital or analog hardware ("vision chips"), or implemented in any other combined solution of hardware and software [10, 14, 15, 26, 29].

Many image processing algorithms are natural to implement in terms of spatially invariant local rules. Thus, CNNs are a convenient platform for image processing. However, specialized CNN processors can be cost-prohibitive and cumbersome research platforms, and CPU-based CNN simulation is very slow in comparison. To make CNN-based image processing applications practical, an alternative platform is needed. This research supports the idea that GPUs are a viable option for both CNN research and applciation.

A software CNN simulator is required for developing and exploring CNN algorithms and applications [16]. In particular, a fast CNN simulator is required for CNN algorithm discovery using evolutionay computation [12]. This CNN simulation and visualization software will facilitate the design and evaluation of new CNN applications, especially those related to image processing.

In summary, the goals of this research are:

- to demonstrate the CNN as a universal abstraction of parallel computation on various platforms,
- to exploit commodity GPUs to improve image processing performance, and
- to provide CNN simulation and visualization tools for CNN research and application.

Part II

BACKGROUND INFORMATION

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

Table 1: CNNs, ANNs compared

2 Cellular Neural Networks

Cellular neural networks (CNNs) are similar to 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 1). Instead of the usual feed-forward, multi-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.

2.1 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 $i, j \in [1..m]$. 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. As we will see, boundary cells must have different properties compared to inner cells because they are connected to fewer neighboring cells.

Considering only inner cells for now, each cell is the center of a neighborhood N_{ij}

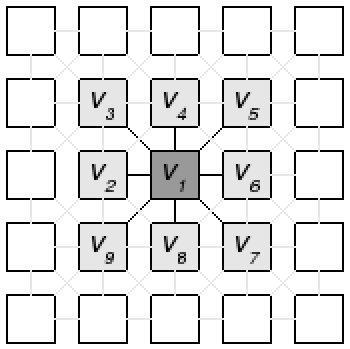

Figure 1: CNN neighborhood

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

2.2 The CNN Cell

Each cell v_k is composed of the following elements:

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 given by $x_k(0)$

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 Section 2.3. 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 determining 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 [8]; 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.

2.3 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 FEEDFORWARD TEMPLATE because it gates the constant input u_k, which in known initially. See Figure 2 to understand this designation.

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.

2.4 CNN Genes

The template matrices A and B, together with the bias term z, are uniform for every inner cell in a CNN. These parameters alone (ignoring the choice of boundary condition) 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:

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

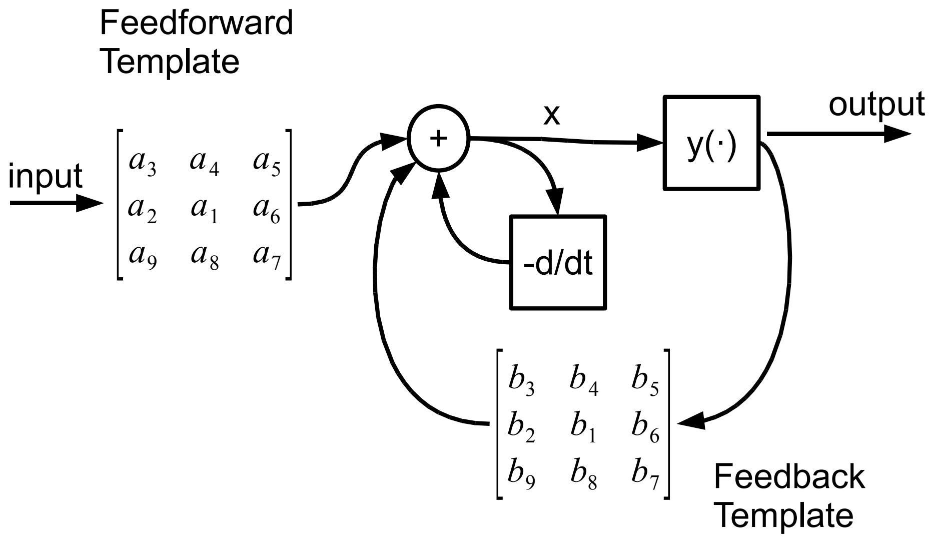
Feedforward Template
input
$\begin{bmatrix} a_3 & a_4 & a_5 \\ a_2 & a_1 & a_6 \\ a_9 & a_8 & a_7 \end{bmatrix}$
+
x
y(·)
output
-d/dt
$\begin{bmatrix} b_3 & b_4 & b_5 \\ b_2 & b_1 & b_6 \\ b_9 & b_8 & b_7 \end{bmatrix}$
Feedback Template

Figure 2: CNN cell system diagram

This vector is usually called a CNN *gene*, owing to the fact that evolutionary algorithms can be employed to find these parameters [8, 18, 22, 32, 12].

3 CNN Algorithms

CNNs can be viewed as a type of generalized cellular automata, and as such they have intrinsic computing power not too dissimilar from Turing complete cellular automata like Conway's Game of Life [8]. This section discusses the use of CNNs for *cellular computation.*

The CNN was designed as a feasible architecture for single-chip, massively-parallel supercomputers, and their cellular, locally-connected topology reflects this effort. Several attempts have been made to fabricate physical CNNs on microchips, many of which have proven very useful for image processing tasks (see Section 3.3).

Computing with these CNN "vision chips" is very different from using traditional processors [10, 14]. Instead of specifying sequences of instructions and loops, CNN algorithms are specified by a CNN gene–that is, by the parameters A, B, and z. More complex algorithms can be constructed using multiple CNNs with different genes, operating in turn like instructions of a processor [25]. For now, though, we will consider algorithms designed for one vision chip using only one gene.

To compute with a CNN, the input data must be cast in the form of two large matrices U and $X(0)$, corresponding to the inputs u_{ij} and initial conditions $x_{ij}(0)$ (respectively) for each cell v_{ij}. These two matrices are called the *input image* and *initial state image* for reasons addressed in Section 3.3. A vision chip operates by loading these images as initial conditions and evolving each cell's state according to (1) and (2). After a set time interval (or after the states converge to steady-state), the final states of each cell are returned as an *output image.* Depending on the gene specifying the behavior of the CNN, the output image will be some transform of the input and initial state images.

A useful CNN algorithm will in this way produce a meaningful output image which reflects some computation of the input and initial state images. For example, consider the task of adding two large matrices. If these two matrices are taken as the input image and initial state image, respectively, a proper CNN algorithm would produce an output image corresponding to the sum of the original two matrices. Admittedly, it is less than straightforward to design a CNN gene for such a task, but evolutionary computation can be used to discover the appropriate CNN genes.

Oftentimes a computation does not require both the input image and initial state image, so one is taken as all zeros. In this case, the CNN algorithm has only one input image and one output image, and the computation can be considered a complicated nonlinear transformation of the input image. The local connections and nonlinear feedback in CNNs make it possible to perform many types of computations with them.

3.1 Universality

When several CNN algorithms are used in succession or in parallel to manipulate a set of images, each CNN algorithm can be considered a unique instruction of a CNN-based supercomputer. One such model, the CNN Universal Machine (CNN-UM), has been proven to be Turing-complete [25, 8, 7]. The proof is fairly simple: a set of CNN "instructions" working together has been used to simulate Conway's Game of Life, which is well-known to be Turing-complete [27]. Thus it is theoretically possible to implement *any* computation or algorithm in terms of a finite set of CNNs working together to manipulate a common pool of data (images).

The CNN Universal Machine makes use of extra memory for image storage and retrieval, and requires a high-level control unit to supervise the loading, initializing, etc of the CNN instructions in sequence. Each instruction is encoded as a gene plus the input and output operands.

3.2 Software Simulations

It is obviously possible to simulate a CNN (or indeed an entire CNN Universal Machine) in software on traditional processors. However, the analog and massively parallel nature of CNNs makes them a less-than-perfect fit for typical processors, which are strictly digital, serial devices. Therefore, CNN simulators typically operate in small time increments, updating all cells in succession at time t and then again at $t+1$. To approximate the continuous evolution of the state variables using these discrete time steps, a numerical integration method such as the Euler method is required when simulating CNNs on any discrete system, including GPUs and CPUs. This results in a mere approximation of the analog CNN behavior, though with a small enough time-step, such an approximation should be sufficient for most CNN computations.

3.3 CNN-Based Image Processing

Image processing (see Section 4) 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. The hardware "vision chips" mentioned earlier were designed for this purpose.

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 of the image processing algorithms that have been implemented for CNNs include contrast enhancement, edge detection, text extraction, skeletonization, and smoothing [21, 31]. Global image operations performed by CNNs include compression, quantization, and adaptation to illumination conditions [9, 19].

3.4 Generalized Cellular Automata

A single-layer CNN is useful for many image processing tasks but is not universal unless the output function is made more complex. Replacing eq. (2) with a more complex or more specific output function can make a single-layer CNN universal or at least more useful. The universal binary neuron (UBN) and multi-valued neuron (MVN), for example, were designed to make single-layer CNNs universal in the binary domain [2, 1]. Generalizing CNNs to arbitrary output functions yields the *generalized cellular automata* (GCAs), which is a superset of both CNNs and cellular automata.

Since some cellular automata (such as Conway's Game of Life) are universal, by extension GCAs are also universal. In fact, the added power and generality of GCAs compared to CNNs means that some image processing tasks which would otherwise require several layers of CNNs can be done with a single-layer GCA.

Of course, the added generality of GCAs makes them more complicated to understand, implement, and study; therefore, it typically makes sense to restrict ourselves to CNNs only, unless confronted with a particular problem that warrants a more complicated output function.

3.5 The CNN Universal Machine

The CNN Universal Machine (CNN-UM) as introduced in [25] is a processor architecture that makes use of an analog, programmable vision chip (hardware CNN) and a store of user-defined instructions which specify the genes for CNNs. Instead of registers, RAM, or a stack, the processor has an image store. Assembly-like programs can be written that transform images from the store using CNN operations or built-in logic operations.

CNN-UM programs consist of a collection of user-defined instructions (CNN genes) and a sequence of statements of the following form:

```
output_img = instruction (input_img, init_img)
```

If 'instruction' is a user-defined CNN, the system will simulate the CNN until it has settled. (Logic operations are handled with a separate arithmetic logic unit.) The respective images in the image store are used for the input image, initial state image, and output image.

The CNN-UM has been suggested as a supercomputer architecture due to its massively parallel computational power. Multi-stage CNN algorithms can be implemented using appropriate sequences of CNN-UM instructions.

4 Image Processing

Image processing is a form of signal processing with two-dimensional arrays (images) of *picture elements* (pixels) as input. Usually, image processing involves performing one of the following operations on images:

1. feature extraction: finding features such as edges, corners, color ranges, textures, etc within an image
2. image enhancement: improving the perceived quality of an image
3. image segmentation: separating regions of an image based on differences in color, texture, or other features

4.1 Edge Detection and Linear Filters

In image processing, a *linear filter* is a mapping from an input image to an output image where the output image contains only some of the information contained in the input image (thus "filter"), and the mapping at each pixel involves only a linear combination of neighboring pixels (thus "linear"). Linear filters are described with an *operator* matrix such as:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

which means that each output pixel $out_{i,j}$ is a linear combination of the corresponding input pixel $in_{i,j}$ and its neighbors:

$$out_{i,j} = (-1)in_{i-1,j-1} + (-2)in_{i-1,j} + (-1)in_{i-1,j+1} + ... + (1)in_{i+1,j+1}$$

Linear filters apply an operator to each pixel in an image, making them *spatially invariant.* In other words, the operation is the same regardless of the pixels position

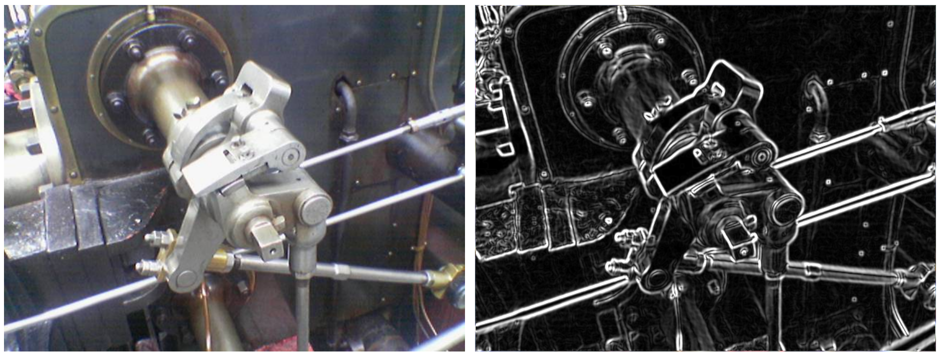

Figure 3: Sobel edge detector output (left: original input; right: output)

(i, j).

One application of linear filters in image processing is *edge detection*–finding pixels which lie on a boundary between regions of an image. Edge detection usually involves finding sharp color gradients since these typically correspond with transitions between regions, and linear filters are well-suited to this task. For example, the Sobel edge detector (see Figure 3) uses a combination of the following Sobel operators, both linear filters:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 & 1 \\ 2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

As stated in Section 3.3, CNNs are often applied to image processing, especially linear filters and other spatially invariant operations. Indeed, CNNs are nonlinear transformations of an input image and include linear filters as a subset. One straightforward application of CNNs, therefore, is edge detection.

4.2 OpenCV Library

One of the most popular programming libraries for image processing (as well as computer vision, etc) is the Open Computer Vision Library (OpenCV), originally developed by Intel [4, 3]. OpenCV is popular for C, C++, and Python applications because of its highly optimized C implementation. OpenCV is specially optimized for Intel x86 processors, but is very fast on any architecture. As such, OpenCV is often used as the *de facto* standard for image processing benchmarks. In this work, OpenCV is used as the basis for comparison with CPU-based image processing algorithms for these reasons.

5 Graphics Hardware and Multiprocessors

GPUs have been readily available to consumers since the 90's, and practically every notebook and desktop computer has some form of graphics acceleration today. With the rapidly increasing demands of video games and other graphics-intensive applications, GPUs have evolved from highly specialized auxiliary components to general-purpose computing platforms.

5.1 CUDA GPUs

NVIDIA's CUDA-capable GPUs are significantly different from previous graphics cards because of their ability to execute general-purpose (though not arbitrary) C code. They provide more than a programmable graphics pipeline; many general-purpose multiprocessors with a fast hierarchical shared memory model make them a truly flexible computing platform.

CUDA GPUs are not stand-alone processors, but complex co-processors controlled by the CPU. The CPU sends data and bytecode instructions (called PTX) to the GPU for processing (see Figure 4). The bytecode and data are stored in the GPU's internal memory, called the *device memory*. During processing, the GPU has access only to the internal GPU device memory–it cannot access RAM or any other devices. When a job is complete, the CPU must copy the resulting (modified) device memory back to RAM. Alternatively, the device memory can be displayed directly on a monitor[1] when the GPU is used for graphics rendering.

Once code and data are loaded onto the GPU, the hardware processes without intervention from the CPU; the CPU process blocks until the GPU is finished. Several GPUs can be controlled in this way using several CPU threads, one to monitor each GPU. Onboard the GPU, many *multiprocessors* execute the bytecode simultaneously

[1]Some GPUs (such as the NVIDIA Tesla series) are designed strictly for general-purpose computing and do not interface to monitors.

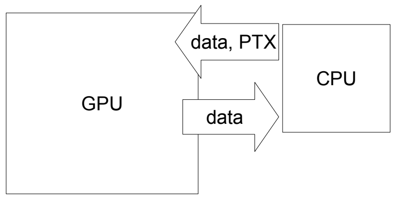

Figure 4: GPU and CPU relationship

using device memory as input and output. In most cases, the code and data is exactly identical for each multiprocessor's *lightweight threads,* except for the *thread index* which is unique to each thread. Each lightweight thread can decide to behave differently based on its thread index.

Most CUDA GPUs on the market have around 64 multiprocessors, each of which can switch between about four register banks, allowing for cost-free context switches. In this way, the number of lightweight threads optimally executed by a GPU is many times more than the number of multiprocessors. For example, a typical NVIDIA Tesla series GPU has 128 multiprocessors and can sometimes run about 3000 threads optimally, depending on the code and data organization.

The multiprocessors are arranged in a two-dimensional *grid* (see Figure 5). Typical grids contain a 4x4 array of multiprocessors, with each block comprising a 2x2 arrangement of processor cores. Each multiprocessor shares a high-speed *parallel data cache* (PDC) designed to support simultaneous reads and writes. Each multiprocessor also shares code and a single entry point (i.e. a function) called a *kernel function.* A GPU with 16 multiprocessor can support 16 different kernel functions simultaneously, and a multiprocessor with 4 processor cores can support around 16 lightweight threads of the same kernel. A hardware *thread execution manager* onboard the GPU provides scheduling and queuing services to the entire GPU. In this way, a GPU can support hundreds or thousands of threads simultaneously without help from the CPU

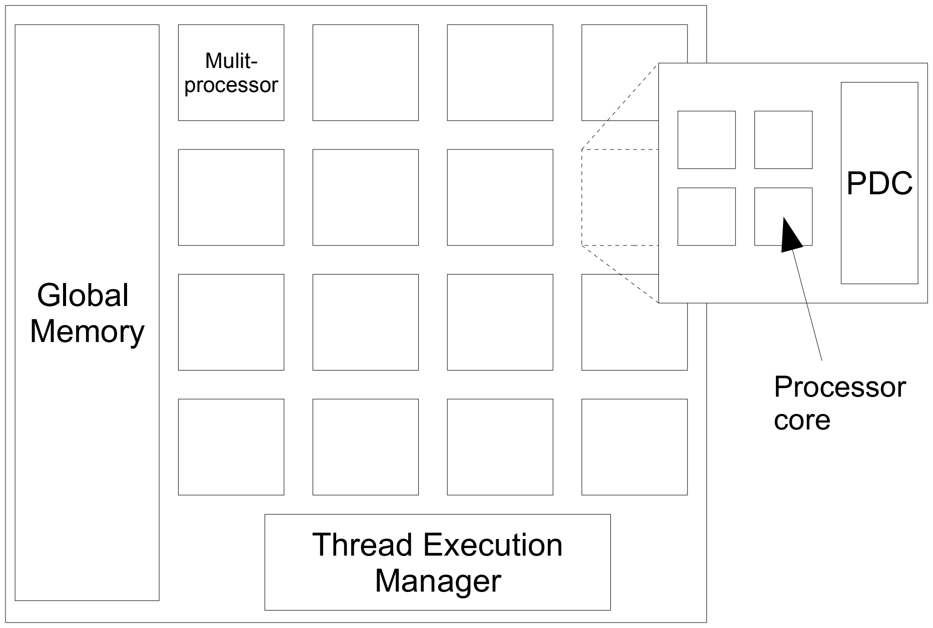

Figure 5: CUDA GPU architecture

or operating system.

CUDA GPUs can be found alongside graphics-only GPUs at major computer retailers at around \$100, making them an affordable platform for high-performance computing. Additionally, NVIDIA currently (as of 2009) sells multi-GPU systems with 960 multiprocessors for around \$8000. These systems provide 4 teraflops of potential computing power in a relatively affordable package.

5.2 CUDA API

To program a CUDA GPU, the CUDA programming platform provides a specialized C compiler which includes a few extensions to ANSI C. The NVIDIA CUDA C Compiler (NVCC) is actually a front-end to the GNU C Compiler (GCC) or a comparable C compiler. NVCC initially separates a CUDA C source file (*.cu) into two parts: host (CPU) code and device (GPU) code. The host code is compiled using GCC, and the

```
// element-wise addition of two matrices
__global__ void AddKernel (float *A, float *B, float *C) {

    int i = blockDim.y * blockIdx.y + threadIdx.y;
    int j = blockDim.x * blockIdx.x + threadIdx.x;
    C [i * cols + j] = A [i * cols + j] + B [i * cols + j];

}
```

Figure 6: example kernel function

GPU code is compiled by NVCC into PTX bytecode. The bytecode is stored within the host executable after linking such that a standard "a.out" executable is produced. When run, the host code sends the PTX bytecode to the GPU as necessary.

From a programming perspective, CUDA programs are written in C using three types of functions. *Host* functions are run on the GPU and include the usual "int main" entry point. *Global* functions are called from a host function but are executed on the GPU. *Device* functions are called from global functions, and are basically inlined into global functions before execution on the GPU.

To distinguish between threads running the same kernel, each thread has a unique block-thread index pair. These indexes are accessed via special registers named blockIdx and threadIdx within the kernel function definition. Additionally, a blockDim register is provided, which stores the size of each block. These variables have x and y components corresponding to the GPU's grid layout. Together, these variables can be used to calculate offsets within a data set such that each thread operates on non-overlapping regions of memory in a SIMD[2] fashion, as in Figure 6.

5.2.1 Limitations of the API

Host functions are written in ANSI C or C++, but global and device functions are restricted to a subset of C due to the limitations of the GPU hardware. The restrictions include:

[2]Single Instruction, Multiple Data

1. no recursion; global/device functions cannot call themselves
2. no function pointer dereferences will work within global/device functions
3. global and device functions can only call device functions, which are simply inlined

Restriction 3 means that global/device functions cannot use the C Standard Library, for example; however, GPU-optimized device functions are provided with the platform to replace the standard math routines.

5.2.2 Launching Global Functions

Host code initiates execution of global functions on the GPU via a kernel *launch*. A *launch configuration* includes the number of blocks to use on the GPU, the number of threads per block, and the parameters available to the lightweight threads. All threads created with a kernel launch use the same kernel function and are given the same parameters. The syntax for a kernel launch is as follows:

```
kernel_function_name <<< grid_size, block_size >>> (param_list, ...);
```

5.2.3 Address Space Separation

As mentioned in Section 5.1, GPUs with dedicated memory (including all CUDA GPUs) maintain their own memory and address space which is separate and distinct from the CPU's main memory (RAM). CUDA GPUs in particular have no access to RAM, and the host CPU has no direct access to the GPU's internal memory. This means that global/device functions cannot read or write to variables or memory stored in RAM, for example. Instead, host code must explicitly initiate transfers of data to and from the device using the built-in cudaMemcpy function. Memory space

```
// allocate memory on the device
cudaMalloc ((void **) &deviceArrayPtr, numBytes);
// copy data from host to device
cudaMemcpy (deviceArrayPtr, hostArrayPtr, numBytes,
    cudaMemcpyHostToDevice);
// launch kernel
kernelFunction <<<gsz, bsz>>> (deviceArrayPtr);
// copy data from device back to host
cudaMemcpy (hostArrayPtr, deviceArrayPtr, numBytes,
    cudaMemcpyDeviceToHost);
// free device memory
cudaFree (deviceArrayPtr);
```

Figure 7: example kernel configuration and launch

is reserved on the device using cudaMalloc and freed using cudaFree. This entire process is illustrated in Figure 7.

Part III

A CNN SIMULATOR

6 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 easier 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.

CNNs have been implemented on many platforms, including the PC [26, 15], cluster architectures [29], custom hardware [10, 14], and GPU [13]. 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 new serial CNN simulator was developed (by the author) for execution on a single processor. This baseline implementation is derived directly from eqs. (1) and (2) using the Euler method of numerical integration as described in Section 3.2. At each time-step, all cells are updated in succession according to Figure 8. The algorithm is run for a specified number of iterations, allowing the cell states to settle to 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 runtime of the computation, since it eliminates nearly half of the number of multiplications. 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 eq. (1) as follows:

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

```
-- initialize cell states
for each cell v[i,j] in M do
    X[i,j] = X0[i,j]
end
-- calculate feedforward 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 cell states (Euler method)
    for each cell v[i,j] in M do
        X[i,j] += DeltaT * Dx[i,j]
    end
end
```

Figure 8: serial CNN simulator pseudocode

with

$$c_1 = \sum_{k \in N} b_k u_k + z \tag{7}$$

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

The codeis implemented in C using the core data structures provided by OpenCV. 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 introduced in Section 14. 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 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. It should be noted that the CNN simulator has been designed strictly for continuous-time CNNs; however, discrete-time CNNs can be approximated by setting $\Delta T = 1$.

6.1 Verification and Output

A few simple CNN algorithms were run using the baseline CNN simulator code to verify that the implementation is correct and working. The first algorithm was taken from [31] and performs edge detection on the input image.

The output of the CNN is compared with the cvLaplace routine from OpenCV in Figure 9. The cvLaplace routine was chosen for comparison because of its fast implemenetation and because the algorithm's output is very similar in appearance to the CNN output, especially compared to the Sobel edge detector (cvSobel) or Canny

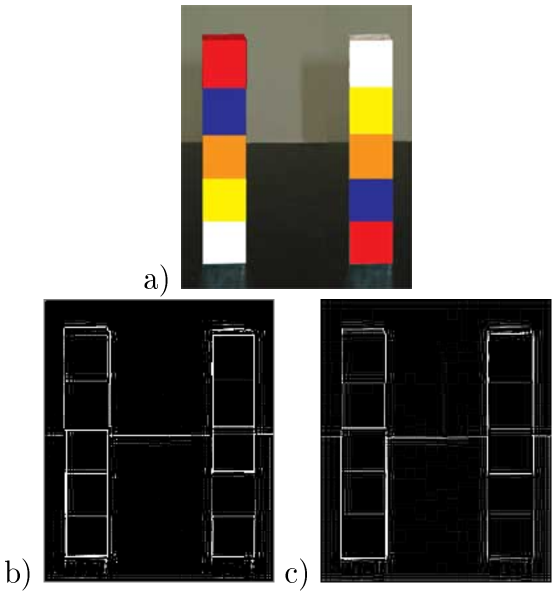

Figure 9: edge detector output
a) input image; b) steady-state output of CNN inner edge detector described in [31]; c) output of OpenCV's Laplace transform for comparison

edge detector (cvCanny). Both the CNN and cvLaplace edge detectors approximate gradients in all color channels (red, green, blue) and output white pixels where the combined gradient is large in magnitude. The test image has subtle blocking artifacts from the JPEG image compression algorithm [28]. Notice that both algorithms mistake these artifacts as edges, although the CNN edge detector seems to be more robust to this noise.

7 Multi-Core Implementation

A second implementation was made, based on the first, to parallelize the simulation on symmetric multiple processors, specifically multi-core processors. The number of threads used by the code can be configured at compile-time and should correspond to the number of cores available on the system. 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 and fast implementation.

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 feedforward kernel, which computes the feedforward image from U, B, and z according to (7),
- the feedback kernel, which computes $\dot{x}_1$ from $y_1(t)$, A, and the feedforward 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 $x_1(t)$.

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" listed in Figure 10.

```
function launch_kernel (kernel_function, inputs)

    local threads = {}    -- array to store thread handles
    local outputs = {}    -- array to store results
    -- launch threads
    for i = 1, N_CORES do

        threads [i] = pthread_create (kernel_function, inputs [i])
    end
    -- wait for all threads to terminate
    for i = 1, N_CORES do
        outputs [i] = pthread_join (threads [i])
    end
    return outputs
end
```

Figure 10: kernel launch function

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.

8 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 [5]. Additionally, the fast shared memory available on CUDA GPUs enables neighboring cells to communicate as required.

CNNs have been implemented using GPUs in the recent past using *shaders* to modify the GPU's rendering pipeline [13]. 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, and in fact very little code is shared between the CPU and CUDA implementations, unfortunately. 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 runtime 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 the custom implementation from Figure 10. 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.

To prevent the need for synchronization and to eliminate blocking between iterations of the simulation, a double-buffer is used by the GPU implementation. All data is read from one buffer (image) and written to a second. The order is switched in the next iteration without moving the buffers. This allows the output of one iteration to become the input of the next iteration without redundant moves.

```
void hook (CvCNN *cnn, CvCNNState *cur) {

    cvShowImage ("state", cur->X);
    cvShowImage ("output", cur->Y);
    cvWaitKey (2);

}
```

Figure 11: CNN visualization hook with highgui

9 Visualization

To support visualization of the CNN simulation, all backends support registering a callback function or event *hook* which is called between each iteration of the simulation. The hook is passed data structures containing the CNN gene, input image, current output image, and current state. The user can provide hooks which display or export this information as needed.

The OpenCV library includes routines for implementing simple graphical user interfaces. Together these routines are called “highgui”. Highgui contains simple functions which are well-suited to displaying the input, output, and state images, etc. Figure 11 shows an example hook function which uses highgui for visualization, and Figure 12 shows a screen capture during simulation with a similar visualization hook.

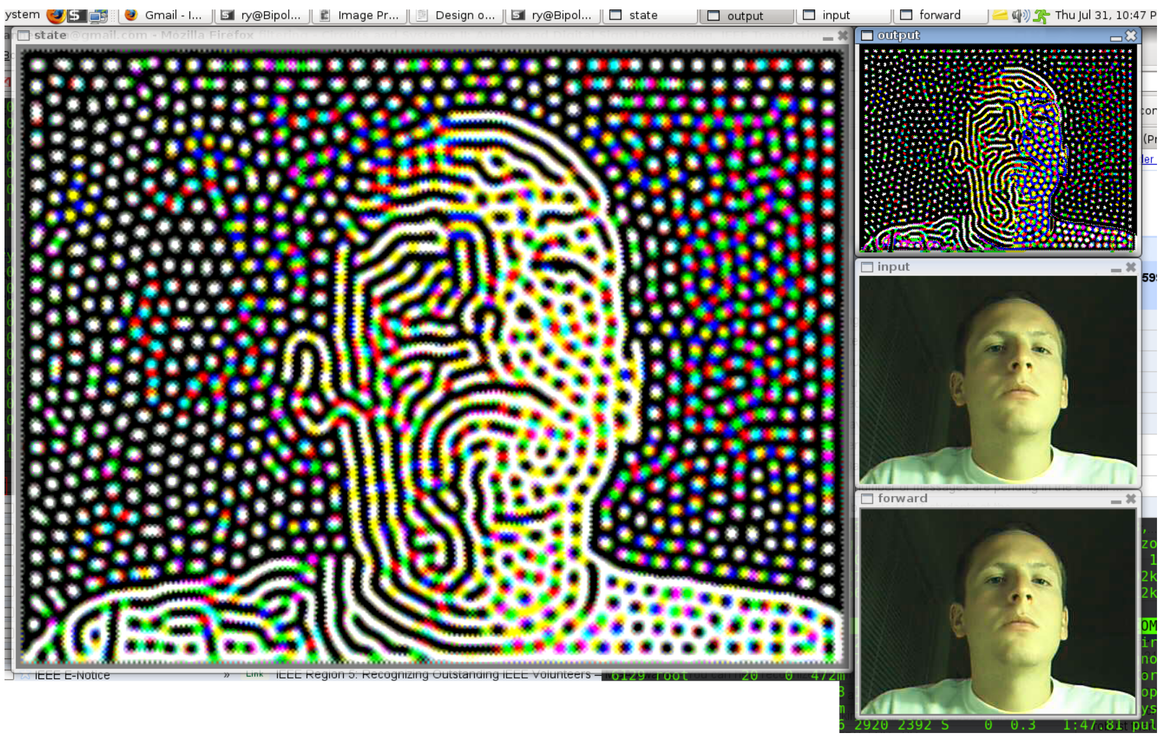

Figure 12: visualization screen capture

	multi-core CPU	**CUDA GPU**
1000x800px	7,500 cells/sec	170,000 cells/sec
500x400px	12,500 cells/sec	1,600,000 cells/sec

Table 2: GPU throughput

10 Comparative Analysis

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.

cvLaplace[3]	**CPU CNN**	**dual-core CNN**	**CUDA GPU**
0.10 s	3 mins	2 mins	0.11 s

Table 3: run-time comparison of 5x5 kernel edge detection algorithms

Table 2 shows that the GPU implementation can achieve approximately 100X speed-up compared to the multi-core implementation for some image sizes. Images approximately 195x195 in size were used to compare the three implementations in Table 3. While relatively modest in size, these images illuminate the shortcomings of CPU-based CNN simulation and image processing. The results in Table 3 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 [17, 5, 20].

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. Further 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 [11]. This is likely due to the fact that global memory is never written to by more than one GPU thread; each cell is computed in a separate thread and each thread writes to one global memory location. Additionally, the double-buffer approach insures that threads do not need to block or wait for neighboring computations. These two simple optimizations have a profound effect on the performance of the GPU without requiring manual manipulation of the parallel data caches within each block (as used in [11]). Instead, no memory conflicts are possible, and the parallel data caches are much less useful.

Part IV

VIRTUAL CNN-UM AND LANGUAGE

11 Overview

The CNN-UM architecture described in Section 3.5 assumes an analog hardware implementation. This is a less-than-desirable programming environment for the following reasons:

1. CNN-UM hardware is difficult to acquire, especially compared to CPUs and GPUs.

2. Analog circuitry is subject to noise and interference problems that limit the performance potential, especially compared to state-of-the-art CPUs and GPUs.

3. CNN-UM software is potentially difficult to develop, as no development tools, compilers, debuggers etc are believed to exist

For these reasons, a CNN-UM "virtual machine" has been implemented. While this virtual CNN-UM is only loosely based on the CNN-UM proposed in [25], we show in Section 13 that the virtual CNN-UM is universal and loses no generality.

Our virtual CNN-UM is programmed with assembly-like programs which operate on a dynamic memory of named images. Each instruction takes four parameters: the input image, the initial state image, the output image, and the simulation time (a unit-less parameter).

To program the CNN-UM, we can simplify the language introduced in [25]. An example program in our language is given in Figure 13. The zero (0) notation on line 12 is used to specify that an all-zero image is to be loaded for the initial state or input, allowing for single-input-single-output transformations.

To simulate a CNN-UM, the CNN simulator described in Part III is used in place of analog circuitry. This allows for CPU, multi-core, and GPU backends. A CNN-UM interpreter is provided which compiles and executes the assembly-like language.

```
-- define instructions:
def inst1 { } < 1 1 0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 >
def inst2 { } < 0 0 1 1 1 1 1 0 0 0 0.5 0 0 0 0 0 0 0 0 >
-- load input images
INPUT file1.jpg R1
INPUT file2.jpg R2
-- perform computations
inst1 R1 R2 R3 10     -- R3 = inst1(R1, R2) with T = 10
inst2 R3 R3 R4 10     -- R4 = inst2(R3, R3)
inst2 R2 0 R2 10      -- R2 = inst2(R2, 0)
-- save output images
OUTPUT file3.jpg R2
OUTPUT file4.jpg R3
```

Figure 13: CNN-UM example program

11.1 CNN-UM Language Specification

Our language is very simplistic, as implied above. A simple JUMP instruction is added to allow looping, and INPUT/OUTPUT instructions are provided to read/write images to/from file:

```
<statement> := <definition> | <instruction> | <jump>
      | <input-file> | <output-file> | <include-file>
<definition> := def <inst-name> <gene>
<instruction> := <inst-name> <opt-reg> <opt-reg> <reg-name> <float>
<jump> := 'jump' <integer> <integer>
<input-file> := 'INPUT' <reg-name> <file-name>
<output-file> := 'OUTPUT' <reg-name> <file-name>
<include-file> := 'include' <file-path>
<opt-reg> := <reg-name> | '0' | '1' | '-1'
<output-fxn> := '{' <minivm-expr> '}'
<gene> := '<' <gene-list> '>'
<gene-list> := <float> | <float> <gene-list>
<file-name> := <file-path> | '$'<integer>
```

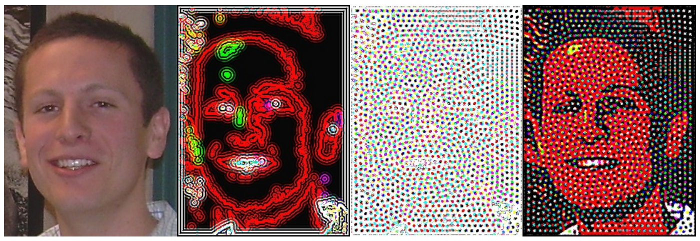

Figure 14: CNN-UM image processing output

A small expression language called MiniVM (see Section 12) is embedded within the CNN-UM language to allow for arbitrary output functions. MiniVM expressions are placed within curly brackets '{}' within an instruction definition.

11.2 CNN-UM Interpreter

A program `cnnum` has been written to parser the above language and emulate a CNN-UM accordingly. The program takes arguments as follows:

```
cnnum [options]
   options:
      --backend=<target>          <target> one of 'cpu', 'mc', or 'cuda'
      -b <target>
      --noviz                     don't show real-time visualization
```

The backend can thus be specified to use the single-core, multi-core, or CUDA CNN simulator. The program reads the program from standard input (stdin) and displays the input, initial state, feedforward, state, and output images of the current instruction in real-time (unless –noviz is specified). In this way a user can see the evolution of the state as a given instruction is simulated. The INPUT and OUTPUT instructions are used to read and write images to/from registers. Several file formats are supported (inherited from OpenCV), including PNG, JPEG, and BMP.

Programs written for the CNN-UM are interpreted and thus are considered scripts. By convention we take the extension *.cdo (CNN do-file) to indicate a script for this

Figure 15: visualization screen capture

purpose and call them *CDO scripts*.

12 MiniVM Expression Language for Output Functions

In order to provide support for arbitrary output functions within the CNN-UM simulator, a simple stack-oriented, post-fix expression language is embedded in the interpreter. The language resembles a subset of the FALSE programming language [30]. Dubbed MiniVM by the author, both the language and embedded interpreter were designed to be as simple as possible so that the interpreter could be implemented on CUDA GPUs.

MiniVM evaluates an expression in post-fix notation using a small stack and constant global variables indicated as \$0, \$1, \$2, etc. Number literals are terminated by a semi-colon. MiniVM expressions look like the following:

```
$01;+|$01;-|-0.5;*
```

When used as a CNN output function, the above MiniVM expression is the standard "Chua" output function (2). The vertical bar operator applies the absolute value function to the top of the stack.

The CNN-UM runtime provides the MiniVM embedded interpreter with constant global variables corresponding to the state of a cell's neighbors. For example, \$1 refers to x_1 and \$a refers to x_{10}. The variable \$0 is initialized with the usual parameter to the CNN output function, x_0, so that the MiniVM expression can implement the output function $y(x_0)$.

12.1 MiniVM on CUDA

The MiniVM language is admittedly cryptic, but readability was not the first design goal. Instead, the language was designed such that the interpreter could be embedded

```
for (; *inst != '\0'; ++inst) {

    switch (*inst) {
    case '+' : *(pos - 1) += *pos; --pos; break;
    case '-' : *(pos - 1) -= *pos; --pos; break;
    ...
    }

}
```

Figure 16: MiniVM interpreter implementation

within a CUDA kernel function. To this end, the interpreter takes up very little code memory and does not require function pointers or recursion.

To use MiniVM on a GPU, first the minivm_compile host function compiles a MiniVM expression into a compact bytecode representation. The bytecode is then passed as a parameter to a kernel launch. Within the kernel function, the minivm_eval device function evaluates the expression and returns the result. The minivm_eval device function is designed to run on the GPU despite the device's limitations (see Section 5.2.1). The main structure of the minivm_eval function is shown in Figure 16.

13 Computational Power of the CNN-UM Virtual Machine and Language

After generalizing the CNN to arbitrary output functions, the generalized cellular automaton contains the well-known cellular automata as a special case. Since some cellular automata (such as the Game of Life) are Turing-complete, we expect that CNNs with arbitrary output functions are also Turing-complete. Additionally, it is possible to prove that multi-stage CNNs (specifically, the CNN-UM implemented here) are universal in the Turing sense, with or without arbitrary output functions. This proof is outlined in Section 13.2.

Notice that in proving that the CNN-UM is Turing-complete, we establish the CNN-UM as a universal model of cellular computation and image processing. This means that any computable image processing task (indeed, any computation at all) can be performed by a CNN-UM algorithm.

13.1 Universality of CNN-UM

An intuitive proof of the universality of the CNN-UM is put forth in [7], in which the Game of Life is simulated with a multi-stage CNN. The Game of Life is often used as a basis for proving universality of models of cellular computations for the following reasons:

1. the proof of universality for the Game of Life is very complicated, and it is assumed that similar proofs for other models of computation will be at least as difficult,

2. the simplicity of the Game of Life makes it an easy target for equivalence proofs, and

3. many models of cellular computation include the Game of Life as a special case.

The elegance of the Game of Life lies in it simple definition and universality; however, this same elegance is what makes its proof of universality extremely complicated.

13.2 CNN-UM Playing the Game of Life

The Game of Life is defined in terms of four local rules acting on a grid of binary cells. Each cell can be "alive" or "dead". The rules to play the Game of Life for one generation are as follows:

1. if a cell has less than 2 living neighbors, it is dead in the next generation,
2. if a cell has more than three living neighbors, it is dead in the next generation,
3. if a living cell has two or three neighbors, it survives to the next generation, and
4. if a dead cell has exactly three living neighbors, it becomes alive in the next generation.

As implied earlier, a grid of cells following these rules can be used to compute any computable function given enough generations and the appropriate initial conditions. In other words, the Game of Life is Turing-complete and is equivalent in power to a Turing machine (assuming an infinite grid of cells). However, finding appropriate initial conditions and interpreting the resulting output is a very difficult task.

This simple game forms the basis of the proof in [7], which assumes that universality of multi-stage CNNs and multi-layer CNNs implies the universality of the CNN-UM. This is a reasonable assumption, since the motivation behind the CNN-UM is a programmable multi-stage CNN. However, the multi-stage CNN is not equivalent to the CNN-UM:

1. multi-layer CNNs allow recurrent connections between layers; this is not supported by the CNN-UM,

2. each layer in a multi-layer CNN is updated simultaneously, whereas the CNN-UM and multi-stage CNN allow one instruction/stage to settle before starting the next,

3. multi-stage and multi-layer CNNs do not rely on storage devices to remember previous cell states; the CNN-UM has a an image store for saving and retrieving CNN states, and

4. CNN-UM includes logic operations which are not equivalent to CNN instructions.

For these reasons, we should look to prove that the CNN-UM presented here is universal in its own right, rather than relying on its similarity to multi-stage or muli-layer CNNs. It is believed that this is the first time a CNN-UM has been considered universal without this assumption.

Here we outline an informal proof which is similar to [7] and borrows heavily from that work; however, here we show an actual CNN-UM implementation of the Game of Life (see Figure 17). The CNN-UM implementation relies on three CNNs based on those presented in [7] for multi-stage CNN Game of Life. When interpreted by the CNN-UM program, the CDO script in Figure 17 performs one update step ("generation") of the Game of Life (see Figure 18). Successive applications of this algorithm (with the inclusion of a JUMP instruction, for example) produce further generations.

Since the Game of Life involves binary states in a 3x3 neighborhood, there is a finite, manageable number of possible neighborhoods for a given cell. We can enumerate all possible neighborhoods and show the correct state in the next generation according to the Game of Life. Doing a similar exhaustive analysis of the CNN-UM program shows that it produces identical generations as the Game of Life.

```
def atleast3total { } < 1

    0 0 0   0 1 0   0 0 0
    .2 .2 .2  .2 .2 .2  .2 .2 .2 >

def atmost3neighbors { } < 1

    0 0 0  0 1 0  0 0 0
    .2 .2 .2  .2  0 .2  .2 .2 .2 >

def b { } < 0 0 -1 >
INPUT $1 in
atleast3total in in A 15
atmost3neighbors in in B 15
b B B B 15
MULT A B out 10
OUTPUT out.png out
```

Figure 17: Game of Life CNN-UM algorithm

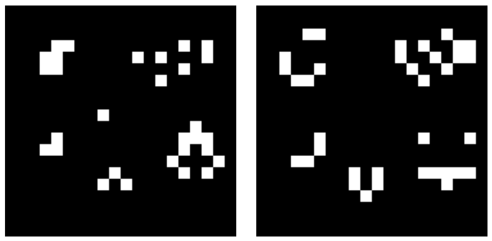

Figure 18: CNN-UM Game of Life output (one step)

Other implementations of the Game of Life are also possible with the software; for example, a MiniVM expression could be constructed to implement the Game of Life in one CNN instruction rather than three; however, the implementation presented above is sufficient proof that the CNN-UM interpreter is Turing complete. Consult [7] for a presentation of several Game of Life CNNs that rely on specialized output functions.

Part V

CNN IMAGE PROCESSING LIBRARY

```
-- snip --
-- remove small isolated objects from a binary image
-- reported in [21]
def removeSmallObjects { } < 0

    1 1 1  1 2 1  1 1 1
    0 0 0  0 0 0  0 0 0

>
-- soft edge detection for improved contrast of grayscale images
-- reported in [32]
def softEdgeContrast { } < -0.365


    0.01 -0.075 0.01  -0.075 1.28 -0.075  0.01 -0.075 0.01
    -0.01 -0.13 0.04  -0.12 0.71 -0.13  -0.04 -0.13 -0.04

>
-- snip --
```

Figure 19: image processing library as a CDO script

14 Implementation

A lightweight image processing library based on the CNN simulator was developed during this research. While not an extensive library by any means, the beginnings of a full-featured image processing library are evident. The library is broken into two parts:

1. a C API to the three CNN simulator backends (CPU, multi-core, and GPU); it is implemented as an extension to OpenCV
2. a CDO script which can be imported into other CDO scripts for use with the CNN-UM interpreter (something like a CDO header file)

An excerpt from the CDO script is given in Figure 19, and an example of its usage is shown in Figure 21.

The C API provides a set of functions to configure the data structures used by the CNN simulator. For example, the cvCNNSpot function initializes a CvCNN

```
include ip.cdo      -- include IP CNNs
INPUT $1 in         -- filename from command-line args
-- perform computations
checkerboard in in out1 10
ripple in in out2 0.5
spot in in out3 10
edge in in out4 2
-- save output images
OUTPUT out1.jpg out1
OUTPUT out2.jpg out2
OUTPUT out3.jpg out3
OUTPUT out4.jpg out4
```

Figure 20: example usage of ip.cdo

data structure so that cvCNNProcessMC will perform the SPOT image processing algorithm reported in [21].

```
// new functions are in italics
IplImage *in = cvLoadIamge (filepath, CV_LOAD_IMAGE_COLOR);
U = cvCreateMat (in->height, in->width, CV_32FC3);
cvConvert (in, U);
cvBipolarThreshold (U, U, 0.0);
CvCNN *cnn = cvCreateCNN (5, CV_32FC3);
cnn->T = 10.0;
cnn->DeltaT = 0.02;
CvCNNState *cur = cvCreateCNNState (in->height, in->width, CV_32FC3);
cvCopy (U, cur->X, NULL);
cvCNNSpot (cnn);   // CNN SPOT algorithm reported in [21]
// simulate using Multi-Core backend
cvCNNProcessMC (cnn, U, cur, NULL);
CvMat *output = cvCreateMat (in->height, in->width, CV_32FC3);
cvCopy (cur->Y, output, NULL);
```

Figure 21: example usage of CNN-IP C API

Part VI

FUTURE WORK AND CONCLUSION

15 Distributed CNN Universal Machine

The CUDA platforms provides no novel approach to clustering or distributed computing. Even single computers with multiple CUDA GPUs (using SLI[4], for example) provide no automatic scaling of computing power beyond one GPU. To take advantage of multiple CUDA GPUs (whether on the same computer or distributed in a cluster), the programmer must rely on existing technologies such as network socket communication, MPI, PVM, shared memory, etc. This is often surprising to new CUDA programmers, who mistakenly view CUDA as an escape from these existing technologies.

CNN simulations have been performed on clusters utilizing single-processor computers, with some success [29]. CNN simulations using GPUs (such as the one presented here and [13]) show more promise at a *much* lower monetary cost, however. The logical extension of this technology would be to employ multiple GPUs on multiple computers to build a "CUDA cluster" such as the one once operated by NVIDIA[5].

CNNs can be used to take advantage of a cluster of CUDA devices in at least two ways. Firstly, higher-level software can be used to distribute a single, large CNN over multiple GPUs on the same computer or on multiple computers. This would allow for efficient processing of larger images (more cells) by allowing more threads to run simultaneously. Considering that consumer digital cameras can produce images with several millions of pixels, there is definitely an immediate use for larger CNNs (more cells) in the domain of image processing. Secondly, it is possible to imagine a "distributed universal machine" that uses each GPU as a separate CNN but uses several CNN "instructions" simultaneously (see Figure 22 on page 57). This is reminiscent of stream processing architectures such as the Cell processor [24], except each syn-

[4]"Scalable Link Interface" allows multiple NVIDIA GPUs to render different parts of the same scene. CUDA uses SLI to share a PCI bus between two cards, but does not allow two cards to behave as one.

[5]NVIDIA at one point gave outside researchers access to their "CUDA cluster" for free. The cluster is no longer operational, according to email correspondences.

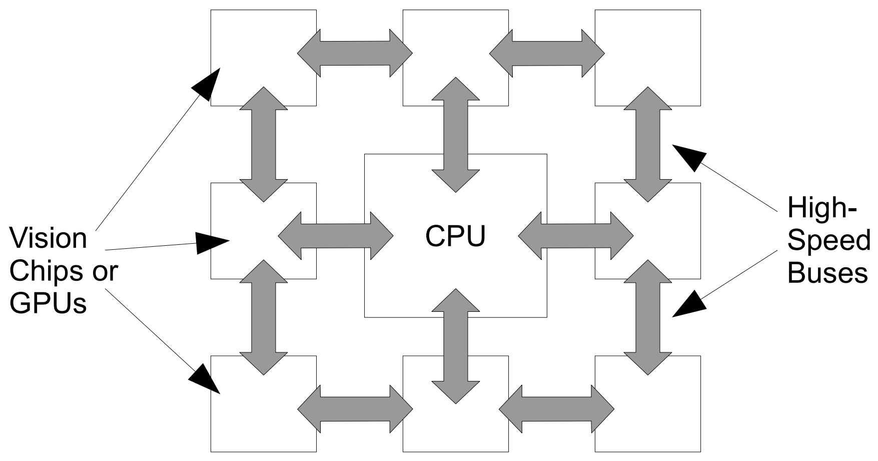

Figure 22: CNN stream processor architecture

ergistic processing element is replaced with a simulated CNN. By extension, several CNN “vision chips” could be used in concert as high-throughput stream processors. Such CNN stream processors would likely excel at the same tasks as existing stream processors, such as video processing and encoding [23].

16 Conclusion

The CNN-UM presented here provides a universal abstraction of massively parallel computation which fits nicely within the limitations of CUDA GPUs. A single CNN-UM algorithm can be simulated on various platforms (CPU, multi-core CPU, GPU) without modifying the algorithm and source code. A simple CNN-UM language can leverage the computational power of GPUs and multi-core CPUs without requiring the user to understand the underlying hardware. The abstraction hides many of the difficulties involved with GPU programming, and GPU-based CNN-UM CDO scripts and CNN simulations demonstrate impressive performance comparable to highly optimized CPU routines in OpenCV.

The GPU-based CNN simulation library we have introduced offers a significant performance gain over CPU-based simulators–in some cases we see a 100x run-time improvement when using GPUs instead of CPUs to simulate CNNs. 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 results presented here are encouraging. As more complex incarnations of massively parallel processors and GPUs are developed, the CNN and CNN-UM may prove increasingly useful as a universal model of cellular computation and image processing.

References

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

[2] N.N. Aizenberg, I.N. Aizenberg, and G.A. Krivosheev. Multi-valued and universal binary neurons: mathematical model, learning, networks, application to image processing and pattern recognition. *Pattern Recognition, 1996., Proceedings of the 13th International Conference on*, 4:185–189 vol.4, Aug 1996.

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

[4] G. Bradski. The OpenCV Library. *Dr. Dobbs Journal November 2000, Computer Security*, 2000.

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

[6] LO Chua and L. Yang. Cellular neural networks: theory. *Circuits and Systems, IEEE Transactions on*, 35(10):1257–1272, 1988.

[7] KR Crounse and LO Chua. The CNN Universal Machine is as universal as a Turing Machine. *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on [see also Circuits and Systems I: Regular Papers, IEEE Transactions on]*, 43(4):353–355, 1996.

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

[9] CM Dominguez-Matas, FJ Sainchez-Femaindez, R. Carmona-Galan, and E. Roca-Moreno. Experiments on Global and Local Adaptation to Illumination

Conditions based on Focal-Plane Average Compuutation. In *10th International Workshop on Cellular Neural Networks and Their Applications*, pages 1–6.

[10] 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.

[11] 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.

[12] H.A. Firpi and E.D. Goodman. Designing Templates for Cellular Neural Networks Using Particle Swarm Optimization. In *Applied Imagery Pattern Recognition Workshop, 2004. Proceedings. 33rd*, pages 119–123, 2004.

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

[14] 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.

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

[16] C.C. Lee and JP de Gyvez. Color image processing in a cellular neural-network environment. *Neural Networks, IEEE Transactions on*, 7(5):1086–1098, 1996.

[17] 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.

[18] M. Mitchell, J.P. Crutchfield, and R. Das. Evolving cellular automata with genetic algorithms: A review of recent work. In *Proceedings of the First International Conference on Evolutionary Computation and Its Applications (EvCA'96)*, 1996.

[19] O. Moreira-Tamayor and J.P. de Gyvez. Subband coding and image compression using cnn. *Int. J. Circ. Theor. Appl*, 27:135–151, 1999.

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

[21] 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.

[22] JA Nossek. Design and learning with cellular neural networks. In *Cellular Neural Networks and their Applications, 1994. CNNA-94., Proceedings of the Third IEEE International Workshop on*, pages 137–146, 1994.

[23] Jonghan Park and Soonhoi Ha. Performance analysis of parallel execution of h.264 encoder on the cell processor. *Embedded Systems for Real-Time Multimedia, 2007. ESTIMedia 2007. IEEE/ACM/IFIP Workshop on*, pages 27–32, Oct. 2007.

[24] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and

K. Yazawa. The design and implementation of a first-generation cell processor - a multi-core soc. *Integrated Circuit Design and Technology, 2005. ICICDT 2005. 2005 International Conference on*, pages 49–52, May 2005.

[25] T. Roska and LO Chua. The CNN universal machine: an analogic array computer. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, 40(3):163–173, 1993.

[26] 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.

[27] R.T. Wainwright. Life is universal! In *Proceedings of the 7th conference on Winter simulation-Volume 2*, pages 449–459. ACM New York, NY, USA, 1974.

[28] G.K. Wallace et al. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991.

[29] 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.

[30] Wouter van Oortmerssen. The false programming language, Apr 2009. `http://strlen.com/false/false.txt/`.

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

[32] F. Zou, S. Schwarz, and JA Nossek. Cellular neural network design using a learning algorithm. In *1990 IEEE International Workshop on Cellular Neu-*

ral Networks and their Applications, 1990. CNNA-90 Proceedings., pages 73–81, 1990.