Hindawi Publishing Corporation
EURASIP Journal on Advances in Signal Processing
Volume 2009, Article ID 930619, 11 pages
doi:10.1155/2009/930619

*Research Article*

# GPU Boosted CNN Simulator Library for Graphical Flow-Based Programmability

**Balázs Gergely Soós,[1,2] Ádám Rák,[1] József Veres,[1] and György Cserey[3]**

[1] *Faculty of Information Technology, Pázmány Péter Catholic University, Práter u. 50/A, 1083 Budapest, Hungary*
[2] *Computer and Automation Research Institute of the Hungarian Academy of Sciences, Kende u. 13-17. , 1111 Budapest, Hungary*
[3] *Infobionic and Neurobiological Plasticity Research Group, Hungarian Academy of Sciences, Pázmány University and Semmelweis University, Práter u. 50/A, 1083 Budapest, Hungary*

Correspondence should be addressed to Balázs Gergely Soós, soos@digitus.itk.ppke.hu

Received 2 October 2008; Revised 13 January 2009; Accepted 12 March 2009

Recommended by Ronald Tetzlaff

A graphical environment for CNN algorithm development is presented. The new generation of graphical cards with many general purpose processing units introduces the massively parallel computing into PC environment. Universal Machine on Flows-(UMF) like notation, highlighting image flows and operations, is a useful tool to describe image processing algorithms. This documentation step can be turned into modeling using our framework backed with MATLAB Simulink and the power of a video card. This latter relatively cheap extension enables a convenient and fast analysis of CNN dynamics and complex algorithms. Comparison with other PC solutions is also presented. For single template execution, our approach yields run times 40x faster than that of the widely used Candy simulator. In the case of simpler algorithms, real-time execution is also possible.



## 1. Introduction

Nowadays there are many hardware implementations of the Cellular Neural Network Universal Machine (CNN-UM) [1] that can be divided into three major categories. The first and usually the most powerful is the mixed-mode (analog and digital) VLSI implementation like the Ace16K [2], the SCAMP chip [3], and the eye-RIS chip [4]. They are also referred to as focal plane processors since they have direct optical input. These architectures employ one-to-one mapping between pixels and processors. Vision Systems are created based on them like the Bi-i system [5] or the eye-RIS system [4] designed for industrial purposes. The second is the emulated digital class that splits into two subcategories. The first is the pipelined version such as the FALCON [6] architectures implemented on DSPs or FPGAs while the other is the coarse grained processor array (e.g., Xenon [7]) where $n$ pixels are mapped to $m$ processors (usually $n$ being much higher than $m$). The third category is the optical implementations like POAC [8] which has the advantage of the massive parallelism. These implementations give researchers a handful of tools for processing two or three dimensional sensory data flows which are received usually from visual or tactile sources. Manipulations of data flows need a special programming environment. Describing these kinds of algorithms can be effectively done by using a high-level Universal Machine on Flows (UMF) [9] flow-chart model and its UMF diagrams.

The advantage of our simulator system is that it combines functionality with clarity of UMF diagrams for description, since these graphically represented data flow algorithms can be run directly without the need for further coding. Furthermore, these diagrams can easily be drawn only using a simple drag-and-drop technique. This feature is very helpful in algorithmic development by speeding up the drawing process. The modeling environment gives fast test results before optimizing the algorithm for any specific hardware.

Our simulator environment is embedded into MATLAB Simulink from MathWorks Inc. [10]. The aim was to exploit its simulation ability for continuous time dynamic systems and utilize its construction method of simple additional

*Blocks* (basic functional units). For referring to this concept, capital letter is used to distinguish it from the one appearing in Section 3. We have created a high level Blockset for the CNN-UM programming structure including easy access to the linear-type, one-layer subset of the CNN template library [13]. The first version of our Blockset used the built-in partial differential equation solver of Simulink. Despite of its relatively slow speed it can be used efficiently for educational purposes.

By deploying the execution of the most computation expensive part to a fast external hardware component, we obtain a powerful development environment that combines the relatively high simulation speed with the advantages of convenient UMF modeling. The mostly data parallel structure of CNN tasks can be exploited by using hardware accelerators, either multicored CPUs or the Graphical Processing Units (GPUs) of graphical cards. We have chosen the GPU for our purposes, because these extremely fast developing systems have high performance and it is easily incorporated into common PCs. There are other research groups working on the development of the GPU powered acceleration of CNN template running [11], which underlines its significance.

The structure of this paper is as follows: after the introduction, Section 2 describes our graphical programming interface, SimCNN in detail. Section 3 gives an overview of the architecture of the recent generation of NVIDIA graphics cards that clarifies the reasons for our choice. Section 4 describes the hardware mapping of the dynamic CNN equations. In Section 5 a complex algorithmic example is presented. Finally, performance comparison with other simulating platforms is given.

Since the first publication [12], our framework has been improved in two regards: (i) we have added a special function for uncoupled templates, (ii) and improved the interfacing Blocks for data transfer between the memory units of the PC and that of the GPU. Thus the possible iteration number that can be calculated for a single template with 25 frames per second has increased from 80 to 90 with the same hardware configuration.

## 2. SimCNN and Its Graphical Flow-Based Programmability

*2.1. The Software Framework.* The simulator is based on Simulink from MathWorks Inc., which is a widely used software that has support for nearly any operating systems. Graphical flow-based programmability has many advantages compared to the standard imperative command-based ways. Algorithms are mapped to transformations realized by *Blocks* and links between them. *Source Blocks* are emitting *signals* that are transferred by the links, processed by numerous transformations, and finally stored or displayed by *sinks*. Blocks can handle multiple inputs or outputs connected to their *ports*. Simulink has built-in Blocks for realizing specific data flow structures, like parallel or conditional execution and signal routing. Simulation can be done for models with continuous states using differential updates, integrator Blocks, and feedbacks. Alternatively, discrete time models can be designed with memory and delay elements based on larger fix time-steps. Built-in solvers exist for both approaches.

In the first version of our SimCNN blockset we used built-in continuous time solvers. In this way multilayer CNNs can be covered, and the dynamics of the system can be visualized easily. In order to increase simulation speed we decided to use hardware acceleration. The current model works as follows: discrete time simulation is used in the top level and all templates are executed atomically, while the dynamic of a single template is handled by our algorithm running on the video card. After the evolution is calculated, the result is passed onto the connecting transformations, and the Simulink scheduler selects the new Block to evaluate. This method is closer to real CNN-UM hardware realizations.

In the SimCNN Blockset, we defined necessary extensions for accessing the video card and to run templates on it (Figure 1). We also created a small user interface to template Blocks for browsing the linear one-layered subset of the template library. However, custom templates are also supported. The Blocks are implemented using the s-function extension interface. The *Video and Image Processing Blockset* covers common image processing tasks although in our case only I/O Blocks are used since algorithmic tasks are solved by CNN templates.

In all time instances, signals are scalars or matrices in the case of images. The memory space of the video card and the *host* computer are separate. The core algorithm of Simulink containing the scheduling and execution of Blocks and data transfer are running on the CPU. Template execution has a layer of code running on the CPU, invoking the layer running on the GPU. Once an image is uploaded to the memory of the device using the ≪Upload to GPU≫ Block, we can refer to it using a real value as ID, similar to pointers used in C language. Image data are handled on the video card, and only the IDs are transferred by the host from one template Block to another. The final results are loaded back to the CPU using the ≪Download from GPU≫ Block. For the first execution time instance, the structure is initialized using the parameters of the surrounding Blocks to determine image dimensions and memory requirements. After the initialization phase, template execution will be performed. Simulink calls event handler functions of each Block one by one in a good serialized order.

*2.2. UMF Description.* The UMF library [13] contains the basic image processing algorithms and numerous powerful ones like the CenterPointDetector or the GlobalConnectivityDetection. This library is an excellent starting point for developers to create their own programs. In the next paragraphs, mapping between UMF notations and Simulink elements marked with ≪symbol≫ are given for most important functionalities. Our aim was to implement the most important subset of the library to cover algorithms using one-layer linear CNN templates.

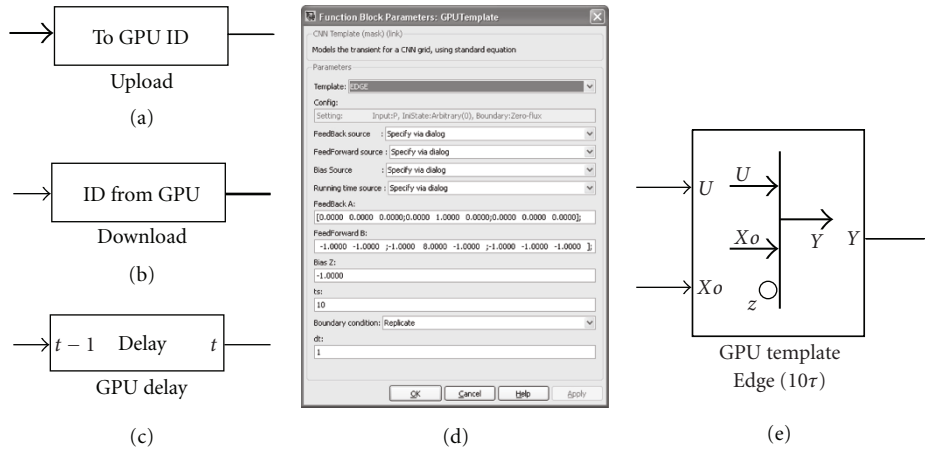


Figure 1: Basic Blockset of SimCNN. Interfacing Blocks (a, b) are required to transfer data to the graphical hardware. The dynamics of each template are calculated on the hardware in one atomic step. Template values, boundary condition, and simulation parameters like time step and running time can be set using a dialog box (d) for each template Block (e). A subset of the template library is also included and can be selected from a drop-down box. A special Block is also added with null transform to delay image flow with one step (c). The Blocks are implemented using C language.

(1) *Elementary instruction*: is the basic template execution that is realized by the Block ≪GPU Template≫. The parameters can be set using the user interface. The Block displays the execution time and also the name of the template if it is from the library otherwise labels it as "USER DEFINED."

(2) *Signals, variables*: simulink supports naming of signals directly. Variables can be created by the ≪GPU Delay≫ Block.

(3) *Boundary conditions*: can be set on the user interface of the corresponding template.

(4) *Continuous delay*: is out of scope since simulation is in discrete time.

(5) *Step delay*: can be realized with ≪GPU Delay≫, or they can be joined together to form longer delay line.

(6) *Parametric templates*: on the user interface of the templates external input can be selected for external parameters.

(7) *Algorithmic structures*: for creating the data path, basic Simulink knowledge is needed. Fortunately MATLAB Help is rather detailed and has good tutorials for learning. To realize "Triggers" ≪Unit Delay Enabled≫ Blocks can be applied. The input flow is blocked if the condition is not satisfied. Complex flow structures like conditional loops can be joined together using ≪MUX≫ (multiplexer) or ≪Switch≫ Blocks.

(8) *Decisions*: decisions on scalar values can be done using ≪Logic and Bit Operations≫, and scalars can be derived from images (like average or min, max) using the ≪ Math Operations≫ Blockset after downloading the image to the host.

(9) *Operators*: various operators are supported by Simulink but only for data in the host memory space. However, implementation of most important global operators can be expected in the near future.

(10) *Subroutines*: are element of the Simulink environment as well.

(11) *Triggers, Cycles*: ≪Enabled Subsystems≫ combined with ≪If≫ or ≪Switch≫ Blocks can be used for UMF-like branching, or ≪Iterator Subsystem≫ can be used for a slightly more readable notation.

In the following example the Center Point Detection algorithm from the UMF library is presented. This is an iterative method to detect mass center of objects in a binary image using series of erosions. Objects are shrinking around their perimeter one pixel in a specific direction in each step. Directions are repeated in clockwise order to slim patches symmetrically. The final result is a single pixel close to the original mass center. Figure 2 shows both UMF description and SimCNN model, and the evolution of the image. In Simulink a ≪Unit Delay≫ Block is needed to avoid direct loops. The ≪Switch≫ Block selects input image or the result from a previous iteration for further processing. For the used morphological templates 2 tau running time with 1 tau time-step is reliable for convergence (DT CNN model).

Figure 3 shows a second example. A basic algorithm was designed to help counting worms in blood samples. Video flows were captured by a camera mounted to a microscope. The specimens are not thin enough to fit into the limited depth of field yielding partially blurred images. Blood particles are relatively dark blobs, haemolyzed particles are giving salt-and-pepper like noise factor. Worms are elongated dark structures. With thresholding, a binary image can be extracted covering the whole worms but also the blood particles. The parts of the worms in focus are black thus a stronger threshold value can be used to select them. The remaining tiny particles can be removed by a Small Object Killer template. A Recall wave can now be started from the

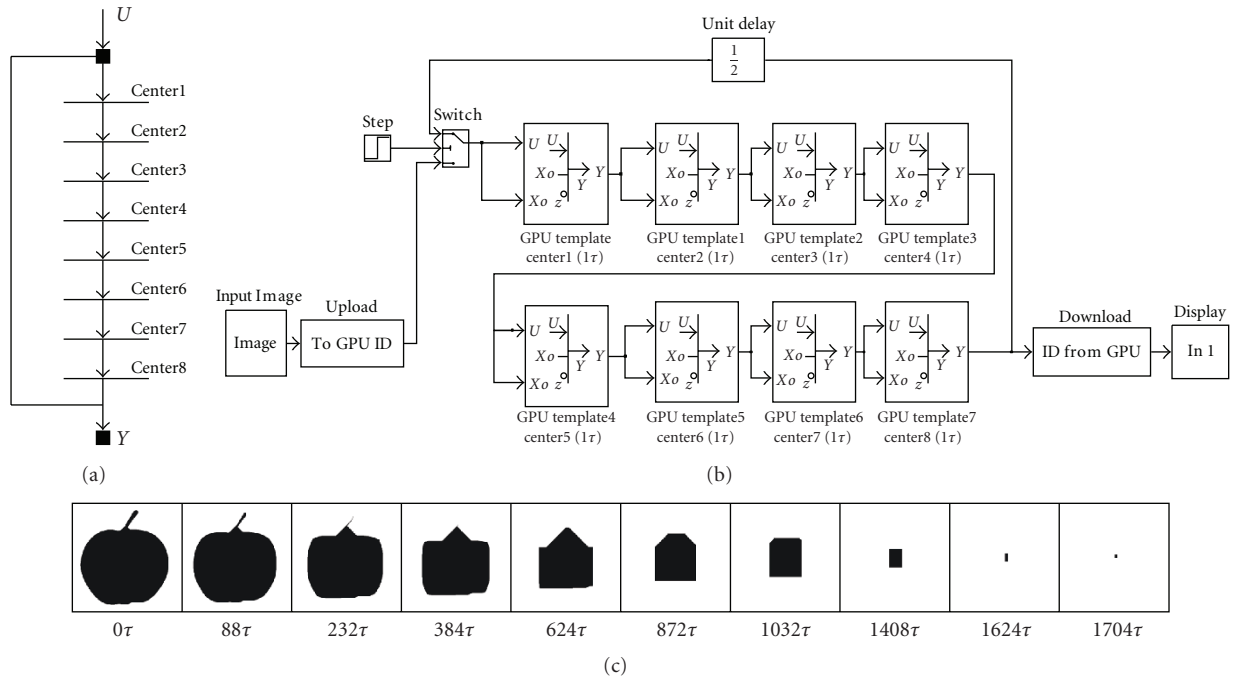


Figure 2: Center Point Detection algorithm. The UMF model (a) and the corresponding SimCNN representation (b) are given alongside the result for a $512 \times 512512 \times 512$ image of an apple (c). The algorithm consists of iterating a sequence of eight erosion templates for all eight direction. The SimCNN model comprises the ≪InputImage≫ and ≪Display≫ Blocks for reading the image file and displaying the result, the ≪Upload≫ and ≪Download≫ Blocks to transfer images to and from the graphic card, and the eight ≪GPUTemplate≫ Blocks. To realize the feedback a ≪Switch≫ Block can be used to select from the original or the updated image. The ≪Step≫ Block provides control signal for the first time instances to send the input into the loop.

cleaned mask to reconstruct worm bodies. The algorithm run video real time.

*2.3. Template Running: The Approximate Equation of the CNN.* CNN dynamics are encapsulated in subroutines running mostly on the graphical hardware.

Let us consider the standard CNN configuration with space invariant templates, connection radius $r$, cloning templates $A_{pq}, B_{pq}$ where $p, q \in \{-r, \ldots, r\}$, bias map $z_{ij}$ and output characteristic function $f$ [14]

$$\frac{dx_{ij}}{dt} = -x_{ij} + \hat{A}_{ij}(y) + \hat{B}_{ij}(u) + z_{ij}, \tag{1}$$

$$\hat{A}_{ij}(y) = \sum_{p,q\in\{-r,\ldots,r\}} A_{pq} * y_{i+p,j+q}(t), \tag{2}$$

$$\hat{B}_{ij}(u) = \sum_{p,q\in\{-r,\ldots,r\}} B_{pq} * u_{i+p,j+q}(t), \tag{3}$$

$$y_{ij} = f\left(x_{ij}\right), \tag{4}$$

$$f(a) = \frac{1}{2}(|a+1| - |a-1|). \tag{5}$$

The differential equation can be solved from a starting point $x_{ij}^0$ iteratively with Euler formula, (1) turns to be (6). More sophisticated solvers such as the Runge-Kutta method could also be used. Discussion will be presented in Section 4.2.

$$x_{ij}(t+dt) = x_{ij}(t) + dt * x'_{ij}. \tag{6}$$

As the input picture does not change over the template execution time, a cumulated bias map $W_{ij}$ can be precalculated. $dt * A_{pq}$ can also be calculated and stored to speed up calculation of the feedback part $V_{ij}$:

$$x_{ij}(t+dt) = (1-dt) * x_{ij}(t) + V_{ij}(x) + W_{ij}, \tag{7}$$

$$V_{ij}(x) = \sum_{p,q\in\{-r,..,r\}} \left[dt * A_{pq}\right] * f\left[x_{i+p,j+q}(t)\right], \tag{8}$$

$$W_{ij} = \left[\hat{B}_{ij}(u) + z_{ij}\right] * dt. \tag{9}$$

Altogether two 2D data sets: state matrix $x_{ij}$ and $W_{ij}$ are needed together with the constant template values to calculate the state update for the next iteration. The template execution can be simulated with adequately small time step for $n$ iterations to achieve steady state (in most cases) or some specific states for templates like heat-diffusion.

## 3. GPU as a Hardware Accelerator

*3.1. CPU versus GPU Comparison.* Nowadays more than a billion of transistors [15] could be found on a single digital chip. This is an outstanding opportunity for creating not only

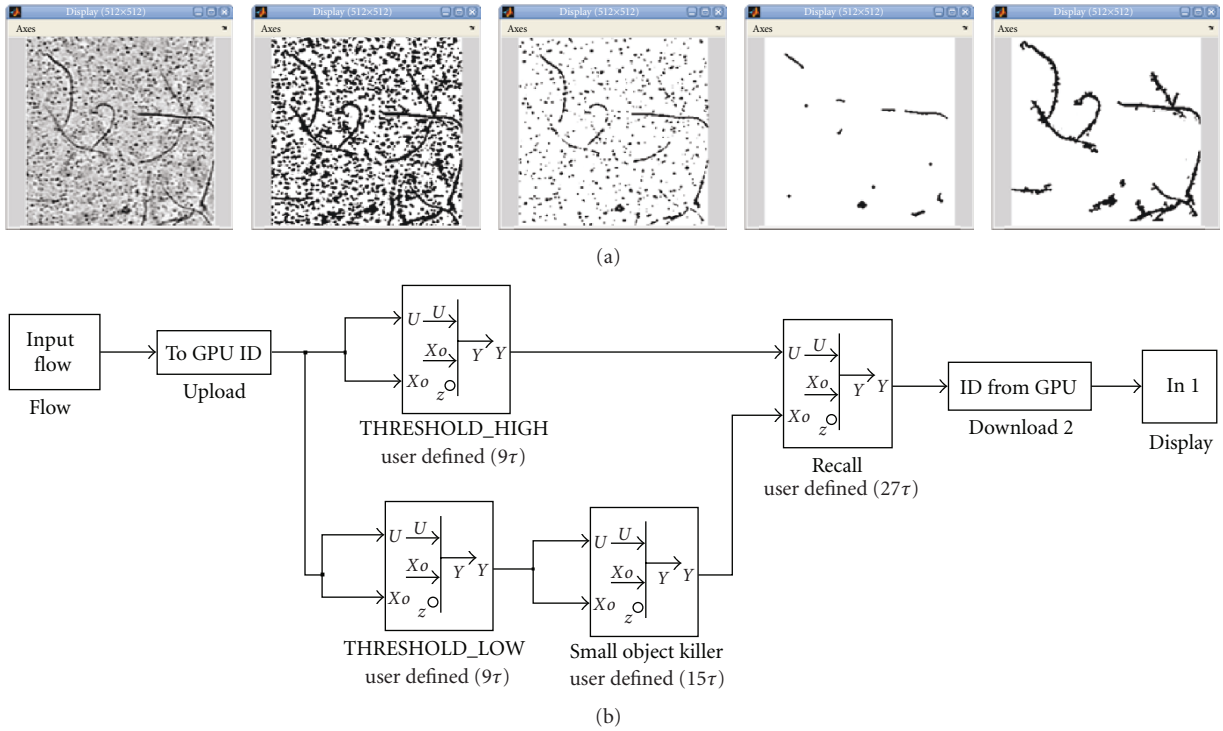


Figure 3: Worm detection algorithm, a realtime example. This figure shows the SimCNN model for the algorithm. It contains the UMF-like diagram flow chart of a sample algorithm (b) that can be run video real-time. This is a worm detection application, for extracting elongated structures from microscope images of blood samples. The upper first image on the upper panel (a) displays the input followed by the two thresholded intermediates, the one after running the Small Object Killer template and finally, the reconstructed image using Recall template that is the result of the algorithm. ≪Upload≫ and ≪Download≫ Blocks are used for interfacing the parts running on CPU and GPU, while templates are running mostly on GPU, represented by ≪GPUTemplate≫ Blocks.

single but dual, quad or many cores on a single die. This multicore tendency, however, had a higher impact on the Graphics Processing Units than on the Central Processing Units of PCs. This is confirmed by the fact that nowadays GPUs with 128 processors are widely available, but only dual- or quad-core CPUs are common on the market. Owens et al. show in their survey [16] that more than five times greater computational power can be achieved compared to nowadays' CPU used worldwide in personal computers. The main significant difference is the amount of transistors dedicated to the data processing elements versus the amount used in local cache circuits. The high percentage of the transistors in CPUs is responsible for flow control and for caching. The GPU needs more effort from the programmer to design efficient code with explicit care of data transfers but offers more computational power on the other hand.

If consider other cheap, compact solutions for scientific calculation than PCs, another possible solution is Sony PlayStation III based on Cell processor technology from the STI alliance (Sony, Toshiba, and IBM). This can be used with Linux operating system but the small amount of memory embarasses the use of graphical environments so this cannot replace a PC in everyday use. Cell is a very powerful digital processor but for some applications, GF8800GT overperforms it. While the Cell Broadband Engine Architecture with 8 Synergistic Processing Elements (SPEs) clocked at 3.2 Ghz is capable of theoretical 208.4 Gflops [17], the 8800 GTX with 128 stream processors clocked at 1.35 Ghz reaches 520 Gflops in peaks (above 300 Gflops in practice) [18] for nearly the same price.

Therefore, we can state that today's GPUs with their high level of parallelism are cost-efficient processors for performing the power resource extensive task of CNN simulator, namely, template running.

*3.2. Compute Unified Device Architecture (CUDA): NVIDIA's Newest Architecture.* In order to realize a nongraphic operation like template running on the graphics card, there must be a software and hardware solution to access the high computational power of the GPU. This need is fulfilled by NVIDIA's newest development, the Compute Unified Device Architecture (CUDA). This novel approach code can be built in C-like language without any knowledge about graphical programming like DirectX or OpenGL. It is available not only on the high-end category, but also on the whole NVIDIA's 8 series cards. The gaming market is huge, driving production in enormous quantities compared to any scientific equipment, making it available for a wide audience in the near future to benefit from its advantages for a low price. Before the technical implementation, we highlight the major hardware parts of the GPU's architecture.

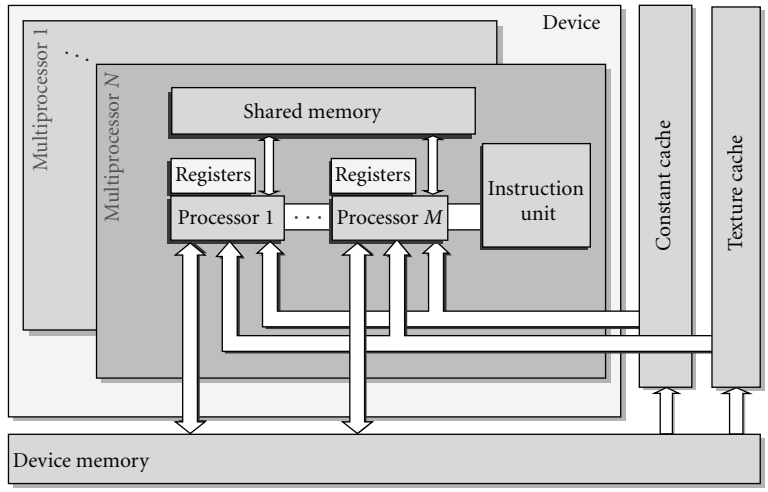


Figure 4: Hardware model of the GeForce 8 series cards [19]. It is a new unified hardware architecture with multiprocessors (MPs) that have dedicated shared memory accessed by a few scalar-based processors replacing the separate vertex and pixel shaders. Processors work with their own registers and are driven by a common Instruction Unit forming a single instruction multiple data architecture. Algorithms can run on multiple MPs, although communication between MPs via the Device Memory is relatively slow. Data can be loaded from the read-only Constant and Texture Cache as well.

In Figure 4 the hardware model of this architecture is described briefly. One can realize that dedicated pixel and vertex shaders were replaced with unified, scalar-based processors. Eight of these processors are grouped with an Instruction Unit to form a single-instruction-multiple-data (SIMD) multiprocessor (MP). The number of MPs varies from 1 to 16, yielding 8 to 128 parallel processors. Global memories are much faster than those in use on mainstream motherboards nevertheless, these would be still too slow for supplying the data for all processors. Therefore, three ways for caching were introduced. A 16 KB per MP high-speed universal purpose shared memory is available beside the constant (1D) and texture (2D) caching memory. I/O instructions accessing device memory through cache converges to 1-2 cycles while direct access costs 100–200 clock cycles.

CUDA supports implementing parallel algorithms with topological processing, running the same code segment on all nodes, which is similar to the CNN architecture. Since the number of the physical computing elements (processors) is typically less than the number of nodes for one-to-one mapping, an automatic serialization is applied. All processors run a thread at a given time, which is its execution context. The atomic algorithm part covering an execution of a subroutine on the nodes is called kernel. Groups of topologically associated threads called blocks (notation with small starting letter) are also arranged in a *grid* topology (Figure 5). The low-level cluster is especially important because all threads in a given block are executed on the same MP, so they have a common high-speed, low-latency (1-2 clock cycles) shared memory that can be used for creating local interconnection. It is tempting to group all nodes to a single block in order to form a fully connected network, a block, however, can encapsulate maximum 256 threads. Moreover, all threads need space for storing their data, and the amount of memory available for a block is only 8 Kbytes, therefore a hard limitation is encountered. Since there are only eight processing elements in a Multiprocessor unit, when large number of threads are associated, frequent context switching also slows down execution [19].

The communication between the threads can be realized only through memory transfers. Synchronization is needed in order to preserve checkpoints to avoid inconsistencies in communication. We have to minimize data exchange between threads that are mapped to distinct blocks, because the only way to do that is accessing the high-latency (100–200 clock cycles) global memory. Since there is no support for the synchronization outside the blocks, the whole kernel should be finished to be sure that all the threads are ready. This means splitting algorithms into smaller kernels. Programs in the CUDA framework consist of kernels compiled into binary code, executed on the graphic card and controlling code running on the host computer.

## 4. Mapping

*4.1. Input/Output Requirements.* The actual topological problem is the CNN dynamics equation. CNN cells are arranged in topological grid. Similar configuration can be created from threads. Analysing the state equations, one could see that this problem is a memory intensive one. To calculate the state update for a pixel, the constants for the template values, the nine state variables from the neighbourhood, and the nine values from the cumulated bias map ($x_{ij}$, $W_{ij}$) are needed. The memory fetch from global memory can cost 100–200 machine cycles compared to 1-2 cycles needed for arithmetic calculations.

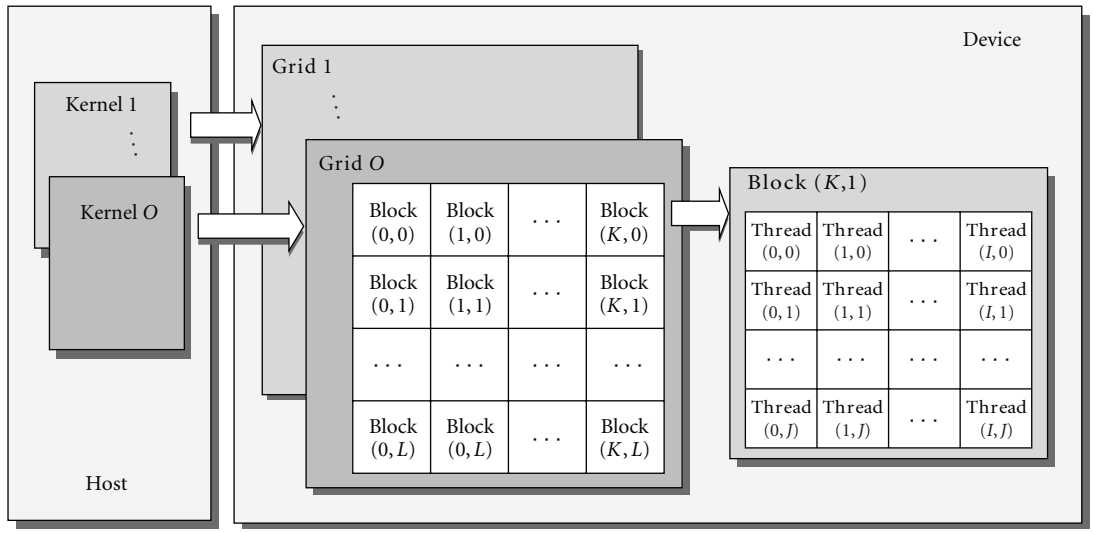


Figure 5: Software running in CUDA [19]. Programs are coordinated by the host computers. Functions that can be executed on GPUs are special data-parallel multithreaded segments called kernels. A 2-level topology (1D/2D/3D), or Grid configuration is assigned to threads. At low level threads are organized into blocks. This grouping is relevant because they mapped to a common MP. Threads run the same code but different data can be referred using index parameters.

If all pixels are accessing their nine neighbors separately, that will not be efficient. The 2D aware texture cache can be applied to assist joint prefetch, but the cache hit rate is less than 100%. Unnecessary redundant memory access can be eliminated with direct copying of all referred states for threads in a block into the shared memory. The small amount of constant values can be fit into the constant cache. After the reading commands, a checkpoint can be placed for synchronization before starting the calculation.

*4.2. Tiling.* Rectangular parts of the image can be assigned to a block and calculated by threads running on the same Multiprocessor, so they can synchronize with each other. After the state update, blocks should write results to global memory to be accessible for neighbours and for the host computer. To calculate derivatives, neighbouring values are also needed, thus overlapped tiling is necessary when covering the full image. The overlapping regions are read from both tiles but only the nonoverlapped regions are updated.

For the Euler method only the $r$ direct neighbours are needed for a state update step, while for 4th-order Runge-Kutta (RK4) to calculate immediate substeps $4 \times r$ cells are needed to be read from $x_{ij}$ image in each direction along the diameter. Mapping squared regions offer the best area perimeter ratio.

Considering $16 \times 16$ pixel blocks and a template with $r = 1$ neighborhood, for Euler neighbor constraint rises $(17 \times 17 - 16 \times 16) = 33$ read overhead compared to 256 updated cell (13 percent), while for RK4 $(20 \times 20 - 16 \times 16) = 144$ extraread is needed (56 percent).

The first version of our system relies on the built in solvers of Simulink so we could test both methods. The linear subset of the templates that can be used in present hardware implementations consists of robust ones that converge relatively fast even with large (0.5 tau, 0.6 tau) steps or even 1 tau for binary templates. The RK method gives more accurate solution, converges faster, and a larger time-step can be used [20]. Although considering the penalty for extra memory access, the advantage can be taken only for more complicated dynamics like polynomial or multilayered CNNs. The aim of this work was to give a tool for algorithmic testing with many templates rather then to write a new high precision solver. The Euler method and (7) will be used to estimate the dynamics.

*4.3. Grid Configuration.* The CUDA gives support for four element vectors beside the standard data types. If a thread is responsible for more pixels, the outputs can be packed using vector representations. On the other hand, simple algorithms—minimizing the branching—are needed for efficient parallel execution. Therefore, four pixels from a row are associated to a thread. To minimize read overhead around perimeters, squared image parts are required. Considering the maximal 256 threads allowed in a block, 4 threads in a row, and 16 rows configuration $(4 \times 16 = 64$ threads) is used for handling tiles with $16 \times 16$ pixel efficient regions.

*4.4. Data Format.* In Simulink the preferred data format is *double* (double precision floating point represented on 64 bit) especially for ≪Video Display≫ Block in $[0, 1]$ interval for images. To store data on the graphic card, *short* (16 bit integer) format was selected. The signal range of CNN $[-1, 1]$ was mapped into $[-2048 \cdots 2047]$, allowing $[-8 \cdots 8]$ value to be represented in a short value assigned to the state variable of a pixel, which is large enough to cover cell dynamics and also a compact form to keep memory consumption low. The arithmetic calculations are

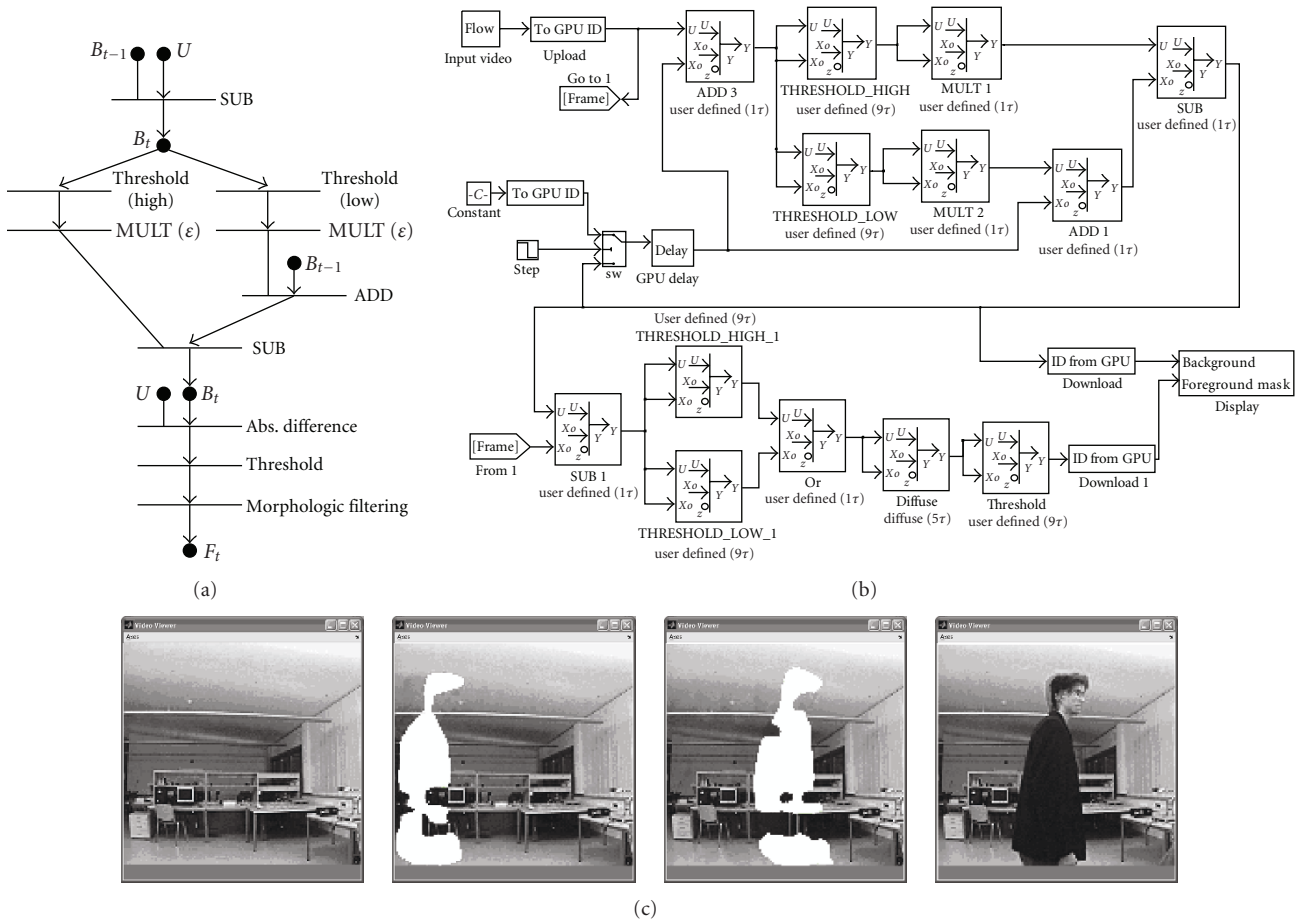


FIGURE 6: Implementation of a complex problem. UMF diagram of the Adaptive Background and Foreground Estimation subroutine (a) is shown with the corresponding SimCNN model (b). The algorithm can process input flow captured with a web camera on the fly. The static part of the input flow is estimated in a feedback loop (Background). Parts with significant difference in gray level are considered to be moving objects (Foreground). Abs. differences and Threshold are implemented using two Thresholds ≪GPUTemplates≫ and Morphological Filtering with a Diffusion and Threshold. During the experiment a person is coming into the screen from the left and stops in the middle. Characteristic screenshots are shown on the upper panel (c). Foreground parts are overlapped to the input flow.

done in *float* (single precision floating point represented on 32 bit) to fit hardware capabilities of the GPU. Format conversion is done automatically during memory access. Range conversion is handled by the interfacing Simulink Blocks implemented by kernels running on GPU.

The execution of a template has two stages: calculating the cumulated bias map ($W_{ij}$) and iterating the state update kernels. Between state updates handling of the boundary condition is also needed. Three appropriate kernels were created with a controlling host function. This function is invoked by the ≪GPUTemplate≫ block.

The main objective was to handle coupled templates. Uncoupled templates for thresholding, logic operations, and for addition are also frequently used in algorithms, so a special kernel with embedding host function was also designed. This kernel can calculate the whole dynamic of a cell since no synchronization is needed between blocks. Using this method gives moderate speed-up, but this is not extremely remarkable since templates from this class need only a small number of iterations to converge (about 10) with small execution time compared to the overhead of executing a Block.

## 5. Complex Algorithm

To demonstrate the possibility of handling complex algorithm in SimCNN the implementation of the Adaptive Background and Foreground Estimation subroutine from the UMF library is presented (Figure 6). The UMF diagram refers to two input variables, namely, U for the actual input image, and $B_i$ for the actual background image. The background model is used from the previous time instant ($B_{i-1}$) that implies a feedback in the processing and a Delay element. Starting value is needed (e.g., 0) in the SimCNN model for initializing $B_0$. Image parts that are significantly brighter in the input than the model are increased, significantly darker regions are decreased in the model image. In case of static scene the model converges.

Table 1: Comparison of test results with other implementations.

| | T7200 | CELL | **8800GT** | Candy | XC3S5000 |
|---|---|---|---|---|---|
| Frequency (MHz) | 2000 | 3200 | **1350** | 2130 | 150 |
| No. of Processors | 2 | 8 | **120** | 1(2) | 34 |
| M cell iteration | 48 | 3627 | **590** | 15 | 1700 |
| Power (W) | 34 | 86 | **160** | 65 | 10 |

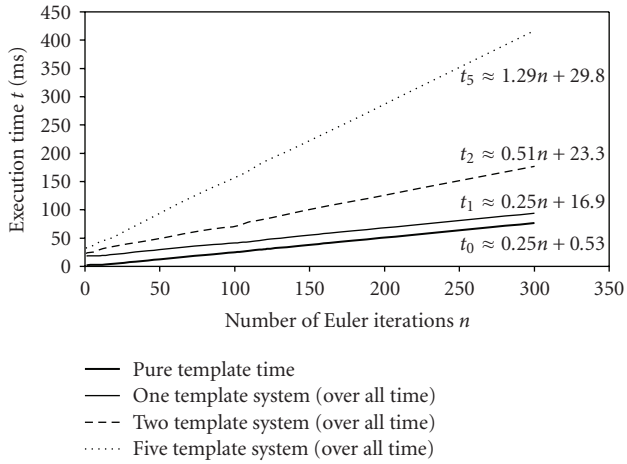


Figure 7: Execution speed of a series of template each running for n iterations. The measured value for a single template running influenced by all necessary interface Blocks is displayed with straight thin line ($t_1$) and its theoretical maximum without overhead with straight bold line ($t_0$). Results for two and five serially joined templates with overhead are indicated by striped ($t_2$) and dotted line ($t_5$) , respectively.

The updated model is compared to the image again in the detection part of the algorithm. Abs. Difference and Thresholding are implemented with multiple Threshold Blocks. Diffuse and a further Threshold Blocks are used to form connected foreground mask. This result is similar in functionality to a series of morphological operators but it is more effective in SimCNN due to the lover number of Blocks.

In the experiments the camera was inspecting our laboratory. Person entering the static scene is detected while he is moving. After he stops the background model is shortly updated. Small changes are not highlighted, like the slowly turning head. Sensitivity can be tuned in the ≪THRESHOLD_LOW_1≫ and ≪THRESHOLD_HIGH_1≫ modules, the learning speed for the model with the multiplication factors.

## 6. Experimental Results

The simulation platform for the SimCNN toolkit was an NVIDIA GeForce 8800 GT GPU connected to an Intel Core-2 2 × 2.13 GHz CPU (E6400) via PCI Express 16x. Our applications were tested both in Linux and Windows environments with NVIDIA driver version 169.07 (32bit) and CUDA Tool kit 1.1. The embedding MATLAB version was 2007 b with Simulink 7.0.

Experiments were conducted on 512 × 512 sized images. The simulation steps were as follows: image loading from hard drive, Upload to GPU, a series of template operation on GPU, Download from GPU, visualization, and frame rate estimation. The built-in profiler in Simulink reliably measures in the millisecond range, so for fast running Blocks it is quite unreliable, but the averaged measurement for large number of independent template executions can be used (1000 inputs). It converges to 0.25 milliseconds for a single Euler-iteration. This would allow for 4000 Euler-iterations per second (ips), but the execution of the interface Blocks, and the embedding layer to Simulink environment present significant additional load. In addition, Simulink also introduces small overhead for joining the Blocks (i.e., communication). The turn around speed for evaluating a single input image of this minialgorithm was also recorded (in frames per second-fps) with the ≪Frame Rate Display≫ Block in the function of Euler-iteration numbers. It is an overall speed indicator that includes the total overhead. For comparison we evaluated networks with a single template, two templates and five templates joined in a line (Figure 7).

Since the first report of our work small improvement has been achieved in the interface Blocks. As a result, real-time processing (25 input images in a second) can be achieved with up to 90 iterations for a frame together with displaying the outputs (590 mega cell iteration per second).

Furthermore, we have compared the results to other simulator systems. For educational purposes, one of the most common tools is Candy from Analogic Computers Inc. [21]. It also offers an algorithmic framework with the template iteration time of 17.5 milliseconds for a 512×512 sized image running on a 2.13 Ghz Intel Core2 Duo (E6400 with 2 MB L2 cache) computer with 14.98 million cell iteration per second. Since this software does not use multithread computing there was no significant effect of the multicore architecture. Measurement for 1.86 Ghz Intel Pentium 4 (M750 with 2 MB L2 cache) gave 12.48 million cell iteration per second. Taken everything together, almost 40x speed-up has been reached.

A comprehensive comparison can be found in [22] for the actual CNN simulators and emulators that we extended with the result of our measurements (Table 1). The first column shows characteristic values for Intel Core Duo (T7200 with 2 MB L2 cache) using the highly optimized Intel Performance Library [23] to implement convolution. The next column stands for the Cell Broadband Engine hosted by a PlayStation III. The last column shows results for an FPGA board with Xillinx Spartan3 chip (XC3S5000). Cell processor gives the best performance but integration into our desktop computer is not yet supported. In the Cell and FPGA setups a variation of the Falcon architecture was used based

on the Euler method. We can conclude that our solution is twelve times faster than that of the CPU of cutting-edge PCs.

The worm detection algorithm that was presented in Section 2.2 consists of four templates, namely, two Threshold templates, a Small object killer, and a Recall template. They converge in 9 tau, 15 tau, and 27 tau, respectively, with 1 tau step size. The complete algorithm consumes 60 iterations for each input image, thus with about 25 milliseconds of total overhead it runs with 25 fps.

The complex algorithmic example presented in Section 5 can run with 5 fps. The online capturing and the large number of connected templates give significant overhead. The processing speed is low but still enough to catch moving people in a live demonstration.

## 7. Conclusion

We have presented an algorithmic development framework for designing and testing CNN algorithms using UMF diagram-like notation. Simulink gives an environment for handling various data streams and for creating complex flow structures to build algorithms. With the additional Simulink Blocks of SimCNN we can get an efficient CNN simulator for an affordable price. We have stated our design considerations for optimal mapping between the CNN and the GPU architecture, resulting in a competitive speed that is twelve times faster than the implementation using high-level optimization on a cutting-edge CPU for a single template. To apply for this SimCNN software package, visit [24].

## Acknowledgments

The Office of Naval Research (ONR), the Operational Program for Economic Competitiveness (GVOP KMA), the NI 61101 Infobionics, Nanoelectronics, Artificial Intelligence Project, and the National Office for Research and Technology (NKTH RET 2004) which supports the Multidisciplinary Doctoral School at the Faculty of Information Technology of the Pázmány Péter Catholic University are acknowledged. The work was sponsored by NVIDIA Corporation and Tateyama Laboratory Hungary Ltd. The authors are also grateful to Professor Tamás Roska, Kristóf Karacs, and the members of the Robotics Lab for the fruitful discussions and their suggestions. Special thanks go to Zsófia Cserey and Éva Bankó.

## References

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

[2] A. Rodríguez-Vázquez, G. Liñán-Cembrano, L. Carranza, et al., "ACE16k: the third generation of mixed-signal SIMD-CNN ACE chips toward VSoCs," *IEEE Transactions on Circuits and Systems I*, vol. 51, no. 5, pp. 851–863, 2004.

[3] P. Dudek and P. J. Hicks, "A general-purpose processor-per-pixel analog SIMD vision chip," *IEEE Transactions on Circuits and Systems I*, vol. 52, no. 1, pp. 13–20, 2005.

[4] AnaFocus, "eye-RIS 1.1 leaflet of AnaFocus ltd.," 2007, http://www.anafocus.com.

[5] A. Zarándy and C. Rekeczky, "Bi-i: a standalone ultra high speed cellular vision system," *IEEE Circuits and Systems Magazine*, vol. 5, no. 2, pp. 36–45, 2005.

[6] Z. Nagy and P. Szolgay, "Configurable multilayer CNN-UM emulator on FPGA," *IEEE Transactions on Circuits and Systems I*, vol. 50, no. 6, pp. 774–778, 2003.

[7] P. Földesy, A. Zarándy, C. Rekeczky, and T. Roska, "Digital implementation of cellular sensor-computers," *International Journal of Circuit Theory and Applications*, vol. 34, no. 4, pp. 409–428, 2006.

[8] S. Tőkés, L. Orzó, C. Rekeczky, T. Roska, and A. Zarándy, "An optical CNN implementation with stored programmability," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '00)*, vol. 2, pp. 136–139, Geneva, Switzerland, May 2000.

[9] T. Roska, "Computational and computer complexity of analogic cellular wave computers," in *Proceedings of the 7th IEEE International Workshop on Cellular Neural Networks and Their Applications (CNNA '02)*, pp. 323–338, Frankfurt, Germany, July 2002.

[10] The MathWorks, "Simulink—Simulation and Model-Based Design," 2008, http://www.mathworks.com/products/simulink.

[11] A. Fernandez, R. S. Martin, E. Farguell, and G. E. Pazienza, "Cellular neural networks simulation on a parallel graphics processing unit," in *Proceedings of the 11th IEEE International Workshop on Cellular Neural Networks and Their Applications (CNNA '08)*, pp. 208–212, Santiago de Compostela, Spain, July 2008.

[12] B. G. Soós, Á. Rák, J. Veres, and G. Cserey, "GPU powered CNN simulator (SIMCNN) with graphical flow based programmability," in *Proceedings of the 11th IEEE International Workshop on Cellular Neural Networks and Their Applications (CNNA '08)*, pp. 163–168, Santiago de Compostela, Spain, July 2008.

[13] L. Kek, K. Karacs, and T. Roska, "Cellular wave computing library v2.1," Tech. Rep. CSW-1-2007, Cellular Sensory Wave Computers Laboratory, Computer and Automation Research Institute, Hungarian Academy of Science, Budapest, Hungary, 2007.

[14] L. O. Chua and T. Roska, *Cellular Neural Networks and Visual Computing*, Cambridge University Press, Cambridge, UK, 2002.

[15] Intel, "The world's first 2-Billion transistor microprocessor," 2008, http://www.intel.com/technology/architecture-silicon/2billion.htm?iid=homepage+news_itanium_platform.

[16] J. D. Owens, D. Luebke, N. Govindaraju, et al., "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[17] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4-5, pp. 589–604, 2005.

[18] NVIDIA, "GeForce 8800 GPU architecture overview," Technical Brief TB-02787-001_v1.0, NVIDIA, Santa Clara, Calif, USA, November 2006.

[19] NVIDIA, "CUDA compute unified device architecture—programming guide 1.1," November 2007, http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf.

[20] R. Kunz, R. Tetzlaff, and D. Wolf, "SCNN: a universal simulator for cellular neural networks," in *Proceedings of the*

*4th IEEE International Workshop on Cellular Neural Networks and Their Applications (CNNA '96)*, pp. 255–259, Seville, Spain, June 1996.

[21] C. S. W. C. Laboratory, "Candy," 2000, http://lab.analogic.sztaki.hu/index.php?a=downloads&c=2.

[22] Z. Nagy, *Implementation of emulated digital CNN-UM architecture on programale logic devices and its applications*, Ph.D. thesis, Department of Image Processing and Neurocomputing, University of Pannonia, Veszprém, Hungary, 2007, http://twilight.vein.hu/phd_dolgozatok/ls.php?s_dir=nagyzoltan.

[23] "Intel performance libraries," 2008, http://www.intel.com/software/products/perflib.

[24] RobotLab, "SimCNN," 2008, http://robotlab.itk.ppke.hu/simcnn.