International Journal of Innovative
Computing, Information and Control
Volume **7**, Number **7(A)**, July **2011**

ICIC International ©2011 ISSN 1349-4198
pp. **3825–3837**

BIO-MIMETIC CLASSIFICATION ON MODERN PARALLEL HARDWARE: REALIZATIONS ON NVIDIA® CUDA™ AND OPENMP™

Thomas Nowotny[1], Mehmet K. Muezzinoglu[2] and Ramon Huerta[2]

[1]School of Informatics
University of Sussex
Brighton, BN1 9QJ, United Kingdom
t.nowotny@sussex.ac.uk

[2]BioCircuits Institute
University of California, San Diego
La Jolla, CA 92093-0402, USA
mmuezzin@ucsd.edu; rhuerta@ucsd.edu

Received February 2010; revised June 2010

Abstract. *Both the brain and modern digital architectures rely on massive parallelism for efficient solutions to demanding computational tasks, such as pattern recognition. In this paper, we implement a parallel classification scheme inspired by the insect brain in two popular parallel computing frameworks, namely as an NVidia® CUDA™ implementation on a Tesla™ device and a brute force OpenMP™ parallel implementation on a quad-core CPU. When evaluating the systems on the MNIST data-set of handwritten digits, we can report that, compared with a standard serial implementation on a single CPU core, CUDA™ implementations of the bio-inspired classification provide a 7-to-11 fold speed-up, whereas the OpenMP™ implementation is 2-to-4 times faster. Our results are a proof of concept that suggests that modern parallel computing architectures and bio-mimetic algorithms are compatible and that the CUDA™ solution on an NVidia® Tesla™ C870 device at the time of writing has a small edge over an OpenMP solution on a recent quad core processor (3 GHz AMD® Phenom™ II X4 940).*

Keywords: Parallel hardware, Graphical processing unit, CUDA, OpenMP, Bio-mimetic systems, Classification, Insect brain, Olfactory system, MNIST data set, Multi-layer perceptron, Random connections

1. **Introduction.** The remarkable speed and accuracy of information processing in nature has been a strong motivation for computing research. A key characteristic of biological computing is architectural: the brain is organized in layers of neurons processing information in a highly parallel fashion. Bio-inspired algorithms, therefore, should naturally have the right structure to take advantage of highly parallel computing architectures. Although one can, in principle, implement such methods on conventional desktop computers in a serial manner, their full benefits can only be revealed on suitable parallel devices.

The practice of parallel computing dates back to the late 50s and early 60s, with the first parallel computer arguably being the D825 of Burroughs Corporation featuring 4 processors. Already in the early 80s, massively parallel computers like the "Connection Machine" with – depending on configuration – up to 65,536 processors were built. However, only recently, the domain of parallel computing has begun to expand from exotic super-computers towards the broader market in the form of multi-core processors, first

introduced by IBM® with the POWER4™ chip in 2001. Perhaps less noticed by the general public until recently, graphic processing units (GPUs) now offer a powerful, highly parallel computing medium as well.

With the introduction of graphics chips supporting the "Compute Unified Device Architecture" (CUDA™) application programming interface (API) extension to the C programming language [1], NVidia® has recently made the power of such GPUs more accessible to general-purpose computing. With hundreds of parallel cores per chip (e.g., 128 on the NVidia® Tesla™ C870 [2]) operating at competitive speeds in the GHz range (1.35 GHz on the Tesla™ C870), GPUs can considerably outperform conventional central processing units (CPUs) in many tasks. However, this potential can only be unleashed by suitable parallel algorithms developed for principally parallelizable problems.

In a series of recent works, we have formulated a model of pattern classification in the insect brain [3, 4, 5]. Biological algorithms of this kind not only have been proven to be effective by withstanding strong evolutionary pressures, but they can also be applied to situations beyond an insect's domain of perception, being nearly as successful as cutting-edge machine learning methods [5].

The CUDA architecture appears to be a natural choice for implementing bio-inspired parallel algorithms and tapping their potential in a variety of pattern recognition tasks. The architecture accommodates redundancy and individually inaccurate outcomes, two aspects that are intrinsic to neural computation but cannot be tolerated in a conventional digital design. With this motivation, we address in this paper the feasibility of neuromorphic computing on an NVidia® Tesla™ C870 GPU within the CUDA framework.

The idea of deploying graphics hardware in general purpose computing is not new. Earlier GPU implementations have mainly used the native graphics programming interfaces of the HLSL shader language in DirectX or the Cg shader language in OpenGL. Oh and Jung [24] have implemented a neural network (NN) utilizing matrix product implementations in OpenGL Cg fragment shaders and reported a 20-fold speed-up in a classification task over a CPU based implementation. Rolfes [6] reported on similar work within the DirectX/HLSL framework. These works did not include the training phase in the timing arguing and it is less time-critical. The enhanced execution speed in these examples can be attributed mainly to the efficiency of the inner product operation [7, 8] in fragment shader GPU implementations.

Steinkraus et al. [9] developed a parallel code for training a feed-forward neural network and achieved a $3\times$ speedup of the GPU implementation over a CPU implementations on 1000 training samples of the MNIST database. More recently, Brandstetter and Artusi [10] and Li et al. [11] extended the GPU domain for NN training and execution further by implementing a radial basis function neural network on a GPU. In this work, the training times of the GPU implementation was compared with a standard algorithm on Mathematica® 5 leading to a speed improvement of two orders of magnitude when fitting a sine function and Hermite functions.

Other works on implementing numerical methods within the OpenGL Cg vertex and fragment shader framework have included wavelet transforms [12, 13], self-organized maps and multi-layer perceptrons [14], a form of spiking neural network [15], simulations of the ocean surface [16] and a cellular neural network [17]. All authors report a speed-up over standard CPU implementations within the range of $3\times$ to $20\times$.

A different approach is the use of the novel and much more flexible CUDA framework, which we focus in this work. Catanzaro et al. [18] have translated the support vector machine classifier design procedure into the CUDA framework based on the sequential minimal optimization (SMO) method. They have reported a $9 - 35\times$ increase in speed compared with a regular CPU when training several large data sets. The trained classifier

itself runs two orders of magnitude faster than a CPU implementation in a more traditional approach [9, 10]. The key factor of their overall success lies in the parallelization of the SMO method. More recently, Nageswaran et al. [19] have implemented a configurable simulation environment for networks of Izhikevich neurons on CUDA reporting an up to 24× speedup on their hardware while Khanna and McKennon [20] have implemented a model for gravitational waves in OpenCL, finding speedups of the order of 30×. Other bioinspired motivated approaches can be found in ant colony optimization for hard combinatorial optimization problems which have also shown substantial speed improvement [21].

In the sequel, we consider a bio-inspired algorithm that is similarly suitable for parallelization as the problems highlighted above. Specifically, we implement a two-layer feed-forward neural network [3, 5] using CUDA parallel kernels in compliance with the memory and communication constraints of the device with appropriate data structures and handling. Our implementation incorporates learning in an incremental, stochastic and local fashion, mimicking the plasticity in the insect brain. As opposed to the conventional machine learning paradigm that isolates the training phase from the actual performance, the considered insect-inspired scheme learns "on-the-fly." Also, unlike earlier work on similar problems, the system we implement here deals with rather sparse matrices with random entries such that the parallel implementations have to compete with efficient sparse matrix methods rather than direct matrix multiplication. We test our system on a benchmark classification task, the MNIST database of handwritten digits [22], and compare the speed of the learning scheme on a standard (Intel®-based) serial processor, on a GPU system, and on an OpenMP parallelized version on a modern quad-core CPU. We observe a significant speed-up in the CUDA and OpenMP implementations over the serial implementations.

For decades, biological methods of classification have inspired the field of machine learning. Nevertheless, the complexity of the redundancy in these systems has always tempted the developers to abstract and simplify these architectures. Our view is that such complexity has a key role in the separation of classes. To the best of our knowledge, this work constitutes the first attempt to translate a massively sparse and tunable kernel expansion to a parallel machine.

The remainder of this paper is organized as follows: in the following section, we briefly outline the biological motivation, the classifier model, its training scheme and the benchmark problem; in Section 3, we evaluate the performance of CUDA, OpenMP and the serial implementations; we provide a brief discussion and outlook in Section 4.

2. **Methodology.**

2.1. **Classifier model.**

We consider a bio-inspired classifier model based on the insect brain described before [3, 5] and outlined in Figure 1. The main biological observations captured in this model are: (1) *The areas responsible for pattern recognition in the insect brain are layered and mainly feedforward;* (2) *The hidden layer (mushroom body calyx (MB)) code is sparse;* (3) *Input-to-MB connectivity is dense and unspecific:* the connectivity of the first layer is not reproducible from insect to insect; (4) *Output layer (MB lobe) neurons compete:* Mutual (lateral) inhibition is the most likely instrument for competition in the MB outputs; and (5) *Plasticity in the MB is tuned by reinforcement:* Behavioral studies and direct electrophysiological evidence indicate that the synaptic changes underlying conditioning (in reinforcement paradigms) take place in the synapses from the so-called Kenyon cells (KC) in the MB to the output neurons in the MB lobes [23].

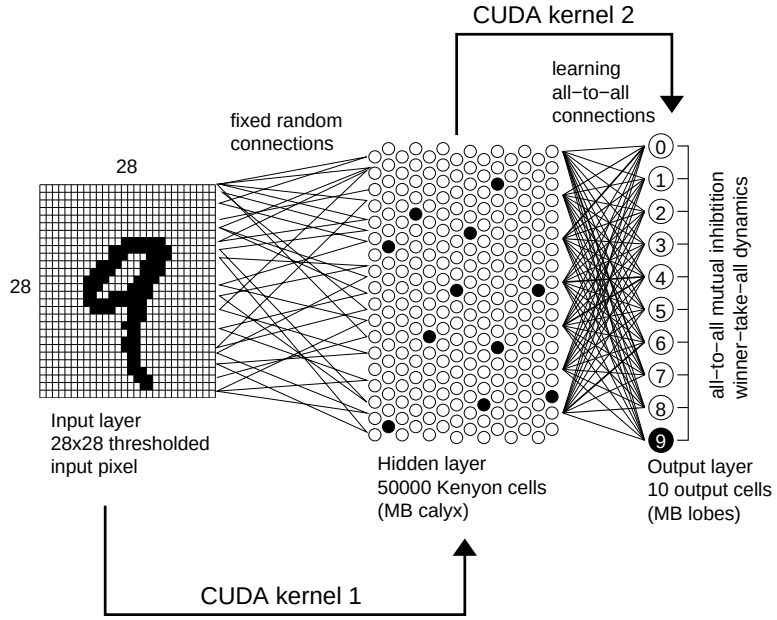

FIGURE 1. Diagram of the classifier model derived from [5]. The system consists of three layers: an input layer of N_{in} binary nodes, a hidden layer corresponding to the mushroom body (MB) calyx and containing $N_{KC} = 50,000$ Kenyon cells (KC), and an output layer corresponding to the MB lobes, each denoting a class identity. The connection from input to calyx are random and from calyx to output are all-to-all and learning. The output neurons compete by lateral inhibition exhibiting winner-take-all dynamics.

In the light of these observations, our model consists of three layers corresponding to the AL (input layer), the calyx of the MB (hidden layer) and the output regions of the MB (output layer) (see Figure 1).

The aim of this work is, in contrast to other recent work on GPU acceleration of artificial neural networks [6, 9, 10, 11, 14, 17, 18, 24], to be strictly bio-mimetic and not adjust the model in non-biologically plausible ways to increase performance. This restriction applies both to increasing performance of the classification achieved by the system and to increasing computational (speed) performance of the different implementations discussed below.

We denote by x_i and y_i the ith unit in the list of N_{in} input nodes and the activity of the ith unit in the layer of N_{KC} hidden nodes, respectively. The model accepts binary inputs, i.e., $x_i \in \{0, 1\}$, $i = 1, \ldots, N_{in}$.

We consider each KC unit as a McCulloch-Pitts (MP) neuron [25] defined by the input-output relation:

$$y_i = \Theta\left(\sum_{j=1}^{N_{in}} v_{ij} x_j - \theta_i\right), \quad i = 1, \ldots, N_{KC} \tag{1}$$

where $\Theta(\cdot)$ is the Heaviside step function and the $\theta_i \in \mathbb{R}$ represent the individual firing thresholds.

We further assume that all synaptic weights from the input to the hidden layer take binary values, i.e., $v_{ij} \in \{0, 1\}$, and are assigned randomly, yielding a random connectivity. This constraint is adopted to reflect the non-specific structure of the MB as a connectionist system. In the construction of the network, we use a Bernoulli process that introduces

an input-to-KC connection with a constant probability p_c. Once the connectivity is determined, the thresholds $\theta_i \in \mathbb{R}$, $i = 1, \ldots, N_{KC}$, are tuned to ensure that each of the N_{KC} units responds (i.e., outputs 1) for only a certain fraction of all inputs (conditioning phase). Specifically, this is achieved by initializing θ_i, $i = 1, \ldots, N_{KC}$, at a common arbitrary value, and then gradually increasing them for cells that fire too frequently and decreasing them for cells not firing enough until the requirement is met. Following the construction and conditioning, the parameters v_{ij} and the thresholds θ_i are fixed for all $i \in \{1, \ldots, N_{KC}\}$ and $j \in \{1, \ldots, N_{in}\}$, establishing the first layer as a fixed random binary kernel.

We designate each one of the N_{out} output nodes to a particular class and determine the recognized class as follows: First, the total input I_i for all N_{out} output neurons is calculated as:

$$I_i = \sum_{j=1}^{N_{KC}} \tanh(w_{ij}/\beta) y_i, \quad i = 1, \ldots, N_{out}, \tag{2}$$

where the unfiltered weights w_{ij} are integer-valued output layer weights and β is a gain parameter. Then, the output class assignment is made by $z = \arg\max_{1 \le i \le N_{out}} I_i$. Note that, since this assignment is based on the arg max operator, only the ordering (not the actual values) of I_i is required. Such an implementation of the lobe constitutes a realistic abstraction (i.e., binary decision) of the stimulus that can be easily processed by further layers, such as action generators (pre-motor areas), in the insect brain [23].

The key feature underlying the success of this system (and also the difficulty in implementing it on a serial machine) is the large number of neurons (≈ 50000) in the hidden layer. This layer has a very sparse activity (i.e., very few "on" units at any given time) which boosts the separability of patterns. Since there are no recurrent (feedback) connections within this layer nor from other layers, the output of the massive number of hidden units can be calculated in parallel.

2.2. **Training.** The MB is a learning classifier that adapts to new stimuli (even to new classes of stimuli) while in actual performance, thus its training and performance should co-occur online. The feedback required for such adaptation is provided in the form of a low-resolution reinforcement signal, similar to the reward signals a natural environment provides to an insect.

In our model, the locus of learning is the output layer weights w_{ij}. Learning is implemented by a local Hebbian-type learning rule that is gated by a reward signal. Specifically, the synaptic weights w_{ij} are adjusted according to the following two rules. If the input was recognized correctly, i.e., a positive reward signal was received, then

$$w_{ij}(n+1) = \begin{cases} w_{ij}(n) + 1, \ \eta < p_+, \ y_j = 1, \ i = z_n, \ I_{\max} \le k I_{2\text{nd}} \\ w_{ij}(n) - 1, \ \eta < p_-, \ y_j = 0, \ i = z_n, \ I_{\max} \le k I_{2\text{nd}}. \end{cases} \tag{3}$$

Otherwise, if recognition was not correct and no reward was received, then

$$w_{ij}(n+1) = \begin{cases} w_{ij}(n) - 1, & \eta < p_+, \ y_j = 1, \ i = z_n \\ w_{ij}(n), & \text{otherwise} \end{cases} \tag{4}$$

where $\eta \in [0, 1]$ is a uniformly distributed random variable and $n \in \mathbb{N}$ denotes (discrete) time. In other words, a synaptic weight w_{ij} is increased by 1 with probability p_+ when a reward signal is received and if (i) the j-th KC is active, (ii) the recognized class is $z = i$, and (iii) the total input to I_i is not greater than the sub-maximum output node's input $I_{2\text{nd}}$ times a constant margin $k > 1$. The third rule provides a safeguard against over-training.

In the opposite direction, the synapse strength w_{ij} is decreased in two cases:
Case 1: When a reward signal is received, w_{ij} is decreased by 1 with probability p_- if (i) the j-th KC is *not* active, (ii) the active output neuron (recognized class) is i, and (iii) the total input I_i is not greater than the sub-maximum output node's value $I_{2\text{nd}}$ times a constant margin $k > 1$;
Case 2: When *no* reward is received, w_{ij} is decreased by 1 with probability p_+ if (i) the j-th KC is active and (ii) the output neuron i was the result of the classifier. In all other cases, the original value of the synaptic weight is retained.

This set of rules prescribes an incremental learning procedure and can be applied on-the-fly, i.e., while the classifier is actually performing. Note, that the only form of supervision utilized in training is whether an input stimulus is recognized correctly or not, i.e., whether a reward signal is received or not. The algorithm, therefore, falls in the category of reinforcement learning. The performance of the resulting classification system has been investigated in detail in [5].

2.3. **Benchmark problem: The MNIST data set.** A large data set is necessary to illustrate the learning algorithm under heavy memory usage, which is one of the main limitations of the current GPU systems. The MNIST database of handwritten digits meets these conditions [22]. This database contains $60,000$ training samples and $10,000$ test samples of handwritten digits in the form of 28×28 gray scale pictures. It is a popular benchmark in the pattern recognition community.

In the following, we use thresholded black-and-white versions of the images, and both the original images and their inverse for reasons explained in our earlier work [5].

2.4. **Parallel and serial implementations on GPU and CPU.** For the same MNIST task, we devised 8 different parallel CUDA implementations, two conventional serial implementations, and an OpenMP parallelized version of the more successful of the two serial implementations of the classification algorithm[1]. These were then tested separately on the MNIST classification problem and ranked with respect to their speed.

The need for parallelism in the algorithm arises primarily in the evaluation of the $50,000$ binary units in the hidden layer of the architecture. Recall that this layer is not subject to training; the computational challenge is the evaluation of such an expansive binary kernel.

2.4.1. *CUDA implementation.* We programmed a simple CUDA kernel evaluating a single binary unit. Each such unit receives inputs from a random subset of the $28 \times 28 \times 2 = 1568$ input neurons, which each represent one of the 784 pixels of the black-and-white image of the input digits or of the 784 pixels of the inverse image. Therefore, within the kernel, the states of the 1568 input neurons, and which inputs the KC unit is connected to, need to be known. We used the maximum of 512 threads per parallel block (a hardware constraint in Tesla$^{\text{TM}}$ C870), and thus evaluated 512 KC units in parallel. Each thread downloads its connections into local memory. Then, the threads download the state of input neurons to shared memory in a parallel fashion (each thread downloading at most 4 states) and calculate the resulting output (0 or 1). This value is written back to the device memory. The process of downloading input states, calculating the KC unit's state, and writing it back to device memory is repeated for a set of $N = 1000$ inputs at a time. Figure 3 illustrates the main steps of the KC evaluation kernel.

We devised a variety of implementations that differed in the storage format and memory handling of the data, including bit-wise storage of KC patterns in chars or ints and

[1]The source codes of the three covered implementations are available at http://www.informatics.sussex.ac.uk/users/tn41/code/MNISTonCUDA.html.

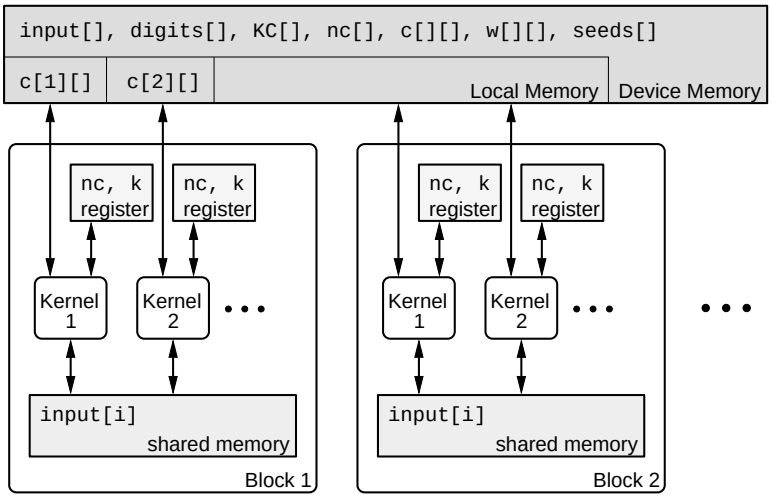

FIGURE 2. CUDA memory model and its use in our implementation. All input-output data passes through the device memory. In our case, these are the input patters (inputs), the corresponding classes (digits), the state of the hidden layer (KC), number of connections to the hidden layer (nc), the input neurons each KC is connected to (c), the weights for the all-to-all connections to the output layer (w), and random seeds for individual random number generation in each kernel. Parts of these data are "pulled down" to shared memory (the input digits) or registers (nc, k) while other data is reorganized into local memory (the PN-KC connectivity c) to allow faster, coalesced memory access. Note that memory access to registers is very fast, to shared memory fast and to device memory, including the so-called local memory, fairly slow. The right use of the memory spaces may make or break a competitive algorithm.

```
determine thread index k
copy connectivity to local memory c[]
for i=1 ... N
  pull pattern i to shared memory AL[]
  synchronize
  sum= 0
  for j=1 ... nc
    sum= sum+AL[c[j]]
  end
  if (sum > theta)
    then KC[i][k]= 1
    else KC[i][k]= 0
  synchronize
end
```

Pseudo code for hidden layer Kernel

Each thread copies 3 or 4 of the input pixels to local memory

calculate the summed input to each KC

threshold the response

FIGURE 3. Pseudo code for the hidden layer kernel. The input patterns are pulled from device memory to shared memory in a parallel fashion, each of the 512 kernels in a block copying 3 or 4 of the pixels. The connectivity for each KC is copied once into local memory and used for N input digits producing N KC patterns for each grid invocation.

assembled locally or in device memory as well as different order schemes how the 1000 patterns of 50000 KC values each are arranged in a 1D array. We ensured that only the

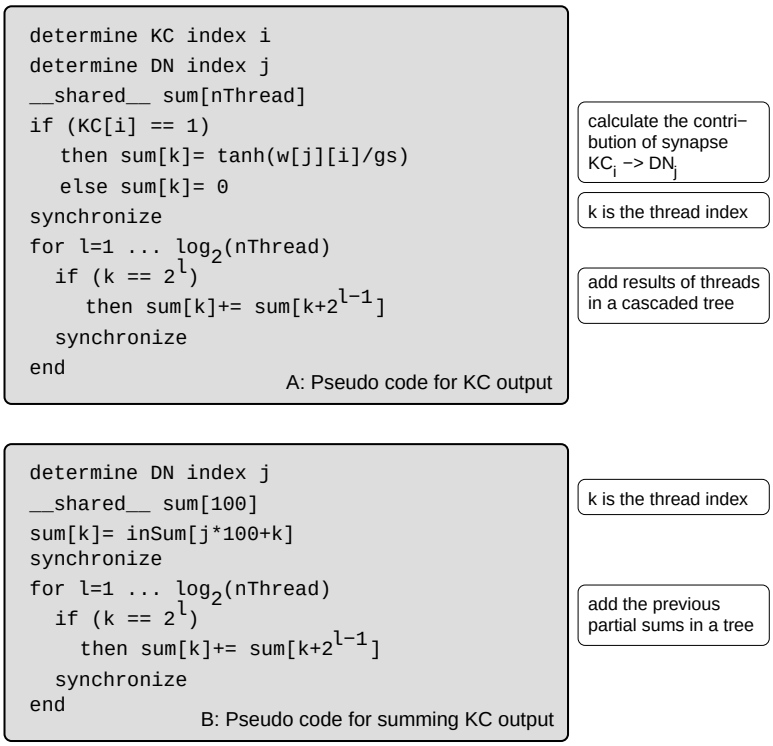

FIGURE 4. Pseudo code for the kernels calculating KC output. First, outputs are calculated and summed in blocks of 500 (A). The resulting 100 partial sums of the inputs to each output neuron are then summed in a separate kernel (B). Splitting the task into many small kernels like this improved performance over a monolithic kernel that does it all at once.

combinations of storage types and strategies were used that avoided explicit memory conflict by design such that the classification performance of all implementations is identical, i.e., the algorithm is executed correctly in all cases.

The second stage of the algorithm, the evaluation of the outputs (i.e., the activity in the third layer) and the modification of connections according to the learning rule are performed in three additional CUDA kernels. Intensive testing revealed that in this case, batch-processing is not advantageous, mainly because there is no benefit in storing connectivity locally with the all-to-all connections from the hidden layer to the output layer.

In brief, in the first kernel grid, which is invoked in blocks of 500 threads, each kernel reads the state of a KC from device memory, and the corresponding contribution of one synapse is deposited into shared memory. These values are then summed in a cascade to give the summed output of 500 synapses per block. A second kernel grid then performs the remaining summation over 100 partial sums of synaptic input to each of the 10 output neurons. Figure 4 illustrates the code of these two kernels.

The determination of the winner MB lobe in the competition of the 10 output neurons is performed on the CPU, and a third kernel grid then updates the synapses to the winning, active output neuron according to the learning rules (3) and (4).

All implemented algorithms were tested on the NVIDIA® Tesla™ C870 system and are compatible with the constraints of its particular technical specifications.

2.4.2. *Serial and OpenMP implementation.* Two serial versions of the system were implemented in C (gcc version 4.3.3, Linux kernel 2.6.28-13) on a modern PC (with 3 GHz

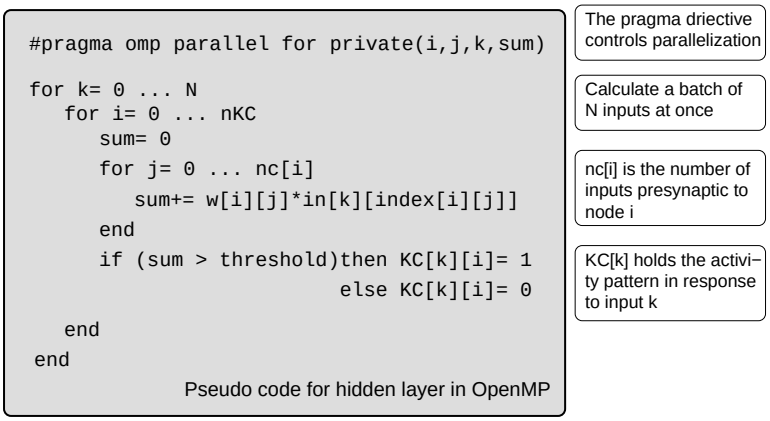

FIGURE 5. An OpenMP implementation of a single layer of binary units for all input patterns. The pragma directive will evenly split the k loop into pieces that run on each of the available CPU cores of the computer. The private variables are those that are not shared by the spawned processes. Parallelization of such simple serial codes is straightforward in OpenMP.

AMD® Phenom™ II X4 940 quad core processor and 8GB RAM) with different data ordering of the 1000 times 50000 bit KC data. Apart from the parallelization-specific code, the serial implementations are functionally identical to the CUDA implementations, including identical integer and floating point data types, the same random number generator, and equivalent math functions. The serial implementations were compiled with –O3 optimization option.

Finally, we parallelized the more efficient of the two serial implementations using the OpenMP API [26] and tested its performance on the host's quad-core CPU (3 GHz AMD® Phenom™ II X4 940). OpenMP is a very easy-to-use add-on supported by recent versions of the GNU C compiler. By introducing a few pragma directives into the code (see Figure 5 for example), it is possible to quickly turn a serial code into a parallel program. The most basic approach to parallelize a C code is to split the 'for' loops into several threads. In our particular case, we had to loop over different input patterns, all input layer units x_j, $j = 1, \ldots, N_{in}$ and all middle layer units y_i, $i = 1, \ldots, N_{KC}$. The best approach is to make long threads that require minimal synchronization. In the implementation, we chose to spawn threads on the iteration over the input patterns, see Figure 5.

3. **Results.**

3.1. **Timing of CUDA and CPU implementations.**

All implementations were tested on the same data set and training protocol with an increasing number of 1000 to 191,000 trained digits. The algorithm includes an initial conditioning (construction and KC-adjustment) phase and the performance measurement phase, which lead to a constant offset of the calculation times that otherwise grow linearly with the number of trained input patterns. The results of the timing trials are given in Figure 6. The values shown in the figure are of the best performing implementations for GPU, CPU and OpenMP versions.

The best serial CPU implementation performs a factor 11.8 worse for conditioning and classification performance measurement (the y-intercept of our linear time – sample number relationships) and a factor 7.7 worse for the time per trained sample (the slope of the linear timing relationships) compared to the best performer in the CUDA group (see Figure 6). The difference in speed-up between the conditioning and the training phases

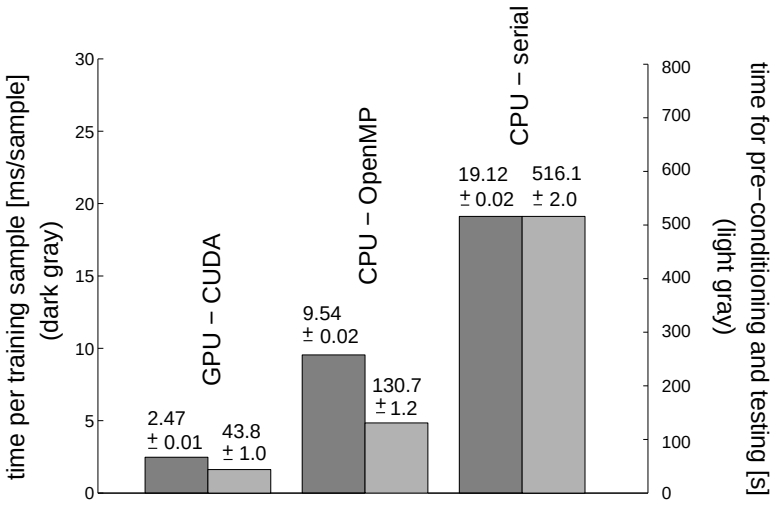

FIGURE 6. Timing results for the three different types of implementations. We tested different versions of the CUDA implementation and single core implementation. The values shown are for the overall best performing implementation in each category. The execution time increases linearly with the number of trained digits (data not shown). We calculated the slope (dark gray) and the offset (light gray) of these linear speed dependencies. The slope is the time needed for training per training sample and the offset is the time needed for initialization, conditioning of the hidden layer, processing of the test set (performance measurement) and for input/output operations. The intervals given as $\pm x$ above the bars, are confidence intervals for the linear regression of the timing curves at a confidence level of 1%.

is due to the fact that in conditioning only the hidden layer is evaluated – the task where parallelization has its greatest impact – while the actual training (and the performance measurement) involve the whole network with the less parallelizable gathering and learning stages between the hidden layer and the output layer. If measured on its own, the hidden layer kernel is 21 times faster than the CPU code.

The parallelized OpenMP CPU version performed markedly better than the serial CPU implementation, reaching about a quarter of the speed of the CUDA version.

4. **Discussion and Conclusions.** All tested CUDA implementations were of similar speed and substantially faster than the serial CPU implementation(s) while achieving the exact same recognition performance.

When comparing CUDA implementations to OpenMP parallelization on a multi-core CPU the picture is less clear. While the CUDA was significantly faster, we only tested OpenMP on a quad core processor. At this point, we, therefore, cannot satisfactorily answer the question which technology, GPU or multicore CPU, provides a better computational environment. For the particular problem investigated here, the CUDA implementation achieves four times the speed of the multicore implementation (OpenMP). However, at current market prices a NVidia® GTX285 is priced at about $400 while the AMD® Phenom™ II X4 940 Processor plus 1 GB of memory is valued slightly above $200. Thus, one could argue that the price of computational power is presently only a factor two different for both technologies. If one factors in the extended developer time

for producing the CUDA-specific code, one may well call it a tie. Interesting challenging problems lie ahead for these two approaches.

We also investigated a more traditional approach to general purpose GPU computing using OpenGL and the Cg shader language. However, the use of random numbers in the algorithm and the restrictions of this framework to rectangular, float-type texture arrays for data storage and communication between shader invocations and the GPU and CPU did not allow competitive performance. It is also worth noting that using a traditional shader language framework requires *even more* restructuring and redesigning of the algorithm than using the CUDA framework and, obviously, the additional effort is *magnitudes greater* compared to an OpenMP parallelization. From our limited experience for this particular application, we do not foresee a major future role for shader languages in the general purpose computing arena in the medium to long term.

The kernel evaluation parallelized in this paper involves redundancy in the form of many KC nodes that possibly convey no information about the class identity. In contrast to the predominant view, we consider this feature as a "capacity" that can be utilized for accommodating changes in class representations, as well as new classes. The CUDA device memory constraint stands as the only limitation on this capacity.

GPU acceleration of artificial neural networks has recently been addressed by several authors [6, 10, 14, 21]. In contrast to these works we here have focused on an algorithm that is explicitly bio-mimetic [19], mimicking the olfactory system of insects. This implies a few limitations.

Firstly, and foremost, the system is limited to biologically realistic, local processing like random synaptic connections, Hebbian learning rules and mutual inhibition. More abstract transformations like radial basis function kernels or more formal learning methods based, for example, on gradient descent are excluded in this approach.

Secondly, we are not at freedom to optimize the size and connectivity of the modules in the model (input, hidden and output layers and their connectivities) to suit the architecture of the GPU system used. This reduces the reported speedup recognizably but offers a more realistic view on what to expect from GPU acceleration when applying it to a generic research question.

Besides GPU acceleration other approaches to hardware acceleration exist [27], including the cell processor architecture [28] and field programmable gate arrays (FPGAs) [29]. The suitability of any of these technologies will depend on the particular problem to be solved, the nature of the task (research or technical/commercial application) and the balance between costs of development time and runtime. The CUDA™ platform allows implementing quick and efficient solutions to a diverse set of problems at a low cost.

In summary, we have given an additional proof of concept of the suitability of bio-mimetic algorithms based on massively parallel computing and the new trend of ever more parallel computing devices. These two appear to be a natural match. In the future, more general tools for GPU acceleration of bio-mimetic systems similar to the system of [19] have to be developed beyond the custom code solutions used in this proof of concept study.

The brain computes multiple threads and not necessarily waits until all the computations are finished to take a decision. There is ongoing work on parallel implementations considering not to synchronize threads and using partial evaluation [30]. This is likely similar to what the multi-tasking ability of the brain rests on and would be an interesting future extension of our work.

Acknowledgment. The work has been funded by the Office of Naval Research (Grant No. N00014-07-1-0741) and the Biotechnology and Biological Sciences Research Council (Grant No. BB/F005113/1).

REFERENCES

[1] *NVIDIA® CUDA™ Documentation*, http://www.nvidia.com/object/cuda_home.html, 2008.
[2] *NVIDIA® TESLA™ Documentation*, http://www.nvidia.co.uk/page/tesla_gpu_processor.html, 2008.
[3] R. Huerta, T. Nowotny, M. Garcia-Sanchez, H. D. I. Abarbanel and M. I. Rabinovich, Learning classification in the olfactory system of insects, *Neural Computation*, vol.16, pp.1601-1640, 2004.
[4] T. Nowotny, R. Huerta, H. D. I. Abarbanel and M. I. Rabinovich, Self-organization in the olfactory system: One shot odor recognition in insects, *Biological Cybernetics*, vol.93, pp.436-446, 2005.
[5] R. Huerta and T. Nowotny, Fast and robust learning by reinforcement signals: Explorations in the insect brain, *Neural Computation*, vol.21, no.8, pp.2123-2151, 2009.
[6] T. Rolfes, *Neural Networks on Programmable Graphics Hardware*, Charles River Media, Boston, 2004.
[7] J. Krueger and R. Westermann, *GPUGems 2: Programming Techniquesor High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, Reading, 2005.
[8] A. Moravanszky, *Dense Matrix Algebra on the GPU*, http://www.shaderx2.com/shaderx.PDF, 2003.
[9] D. Steinkraus, I. Buck and P. Y. Simard, Using GPUs for machine learning algorithms, *Proc. of the 8ht International Conference on Document Analysis and Recognition*, 2005.
[10] A. Brandstetter and A. Artusi, Radial basis function networks GPU-based implementation, *IEEE Transactions on Neural Networks*, vol.19, pp.2150-2154, 2008.
[11] J. Li, X. Han, X. Hu and H. Tan, A parallel algorithm of handwritten digits recognition based on artificial neural network with GPU-acceleration, *ICIC Express Letters*, vol.3, no.4(A), pp.1093-1100, 2009.
[12] A. Garcia and H.-W. Shen, GPU-based 3D wavelet reconstruction with tileboarding, *The Visual Computer*, vol.21, no.8, pp.755-763, 2005.
[13] T.-T. Wong, C.-S. Leung, P.-A. Heng and J. Wang, Discrete wavelet transform on consumer-level graphics hardware, *IEEE Transactions on Multimedia*, vol.9, no.3, pp.668-673, 2007.
[14] Z. Luo, H. Liu and X. Wu, Artificial neural network computation on graphic process unit, *Proc. of IEEE International Joint Conference on Neural Networks*, vol.1, pp.622-626, 2005.
[15] F. Bernhard and R. Keriven, Spiking neurons on GPUs, *Lecture Notes in Computer Science*, Berlin, vol.3994, pp.236-243, 2006.
[16] X. Yang, X. Pi, L. Zeng and S. Li, GPU-based real-time simulation and rendering of unbounded ocean surface, *The 9th International Conference on Computer Aided Design and Computer Graphics*, 2005.
[17] T.-Y. Ho, P.-M. Lam and C.-S. Leung, Parallelization of cellular neural networks on GPU, *Pattern Recognition*, vol.41, no.8, pp.2684-2692, 2008.
[18] B. Catanzaro, N. Sundaram and K. Keutzer, Fast support vector machine training and classification on graphics processors, *Proc. of International Conference on Machine Learning*, 2008.
[19] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau and A. V. Veidenbaum, A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors, *Neural Netw.*, vol.22, no.5-6, pp.791-800, 2009.
[20] G. Khanna and J. McKennon, *Numerical Modeling of Gravitational Wave Sources Accelerated by OpenCL*, 2010.
[21] J. Li, X. Hu, Z. Pang and K. Qian, A parallel ant colony optimization algorithm based on fine-grained model with gpu-acceleration, *International Journal of Innovative Computing, Information and Control*, vol.5, no.11(A), pp.3707-3716, 2009.
[22] Y. LeCun and C. Cortes, *The Mnist Database*, http://yann.lecun.com/exdb/mnist, 1998.
[23] M. Heisenberg, Mushroom body memoir: From maps to models, *Nat. Rev. Neurosci.*, vol.4, pp.266-275, 2003.
[24] K.-S. Oh and K. Jung, GPU implementation of neural networks, *Pattern Recognition*, vol.37, no.6, pp.1311-1314, 2004.
[25] W. S. McCulloch and W. Pitts, Logical calculus of ideas immanent in nervous activity, *B Math Biophys*, vol.5, pp.115-133, 1943.
[26] *OpenMP Specifications*, http://openmp.org/wp/openmp-specifications/, 2010.

[27] T.-Y. Lee, Z. Ji and M. Hu, Hardware-software partitioning for embedded multiprocessor FPGA systems, *International Journal of Innovative Computing, Information and Control*, vol.5, no.10(A), pp.3071-3084, 2009.

[28] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer and D. J. Shippy, Introduction to the cell multiprocessor, *IBM Journal of Research and Development*, vol.49, no.4/5, pp.589-603, 2005.

[29] J. Wang, Z. Ji and M. Hu, High-performance multi-pattern matching structure in hardware network firewall, *ICIC Express Letters*, vol.4, no.1, pp.137-142, 2010.

[30] T. Halfhill, Parallel processing with CUDA, *Microprocessor*, 2008.