20th International Conference on Artificial Neural Networks (ICANN), Thessaloniki, Greece, September 2010

Accelerating Large-Scale Convolutional Neural Networks with Parallel Graphics Multiprocessors

Dominik Scherer, Hannes Schulz, and Sven Behnke

University of Bonn, Institute of Computer Science VI,
Autonomous Intelligent Systems Group, Römerstr. 164, 53117 Bonn, Germany
{scherer|schulz}@ais.uni-bonn.de, behnke@cs.uni-bonn.de
http://www.ais.uni-bonn.de

Abstract. Training convolutional neural networks (CNNs) on large sets of high-resolution images is too computationally intense to be performed on commodity CPUs. Such architectures, however, achieve state-of-the-art results on low-resolution machine vision tasks such as recognition of handwritten characters. We have adapted the inherent multi-level parallelism of CNNs for Nvidia's CUDA GPU architecture to accelerate the training by two orders of magnitude. This dramatic speedup permits to apply CNN architectures to pattern recognition tasks on datasets with high-resolution natural images.

1 Introduction

Biologically-inspired convolutional neural networks (CNNs) have achieved state-of-the-art results for the recognition of handwritten digits [5, 11] and for the detection of faces [1, 8]. However, since gradient-based learning of CNNs is computationally intense, it would require weeks to train large-scale CNNs on commodity processors. It therefore remains a largely unexplored question whether CNNs are viable to categorize objects in high-resolution camera images.

In the 1990ies similar incentives to parallelize neuroapplications led to the development of both general-purpose and special-purpose neurohardware. Popular data-parallel techniques primarily relied on the parallel execution of nodes for a single input pattern [13]. Early approaches to implement CNNs on parallel graphics hardware [2] yielded speedups of up to 4.11×. More recently, multilayer perceptrons [3], locally-connected neural networks [12] and deep belief networks [9] have been adapted to GPUs with speedups by two orders of magnitude.

Modern graphics cards consist of several hundred parallel processing cores. Nvidia's scalable CUDA framework [7] (Compute Unified Device Architecture) makes it possible to harness their computational power of several hundreds of GigaFLOPS (floating point operations per second) without requiring to learn a specialized programming language. However, in order to accelerate an application, hardware-imposed memory access restrictions must be considered to efficiently exploit the fast on-chip shared memory. Both general neural networks and convolution operations are inherently parallel. Thus, CNNs, which combine both, are a particularly promising candidate for a parallel implementation.

To account for the peculiarities of such CUDA-capable devices, we had to slightly divert from common CNN architectures. However, our CNN model for CUDA's parallel hardware architecture is sufficiently flexible for a large range of machine vision tasks. We describe a fast parallel implementation of the backpropagation algorithm to train this network on GPU hardware. The program scales well on an arbitrary number of processors, and employs a circular buffer strategy to minimize data transfers from the device memory to the on-chip shared memory. This implementation achieves a speed-up factor between 95 and 115 over a serial, single-core CPU implementation and scales well with both the network and input size.

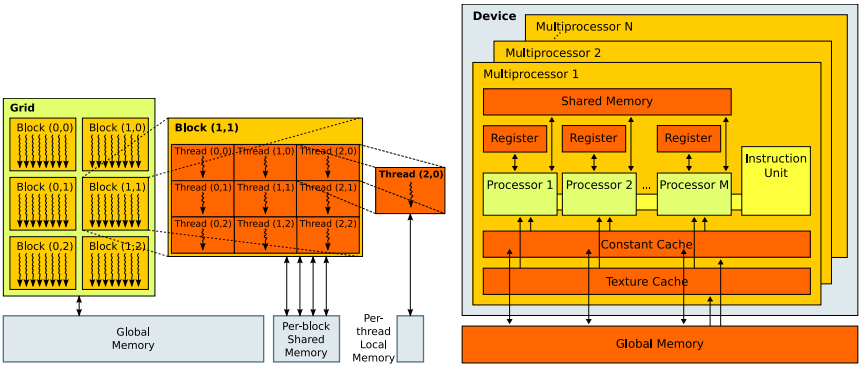
2 Massively Parallel Computing on GPUs

Modern graphics processing units (GPUs) have evolved from pure rendering machines into massively parallel general purpose processors. Recently, they exceeded 1 TeraFLOPS [7], outrunning the computational power of commodity CPUs by two orders of magnitude. GPUs employ the fundamentally different SPMD (Single Program, Multiple Data) architecture and are specialized for intense, highly parallel computations. More transistors are devoted to data processing rather than to data caching and control flow. The CUDA framework introduced by Nvidia allows the development of parallel applications through "C for CUDA", an extension of the C programming language.

The CUDA programming model considers a graphics card (*device*) as a physically separate co-processor to the CPU (*host*). Computations on the GPU are initiated through *kernel functions* which essentially are C-functions being executed in N parallel *threads*. Semantically, threads are organized in 1-, 2- or 3-dimensional groups of up to 512 threads, called *blocks*, as shown in Figure 1a. Each block is scheduled to run separately from all others on one multiprocessor. Blocks can be executed in arbitrary order – simultaneously or in sequence, depending on the system's resources. However, this scalability comes at the expense of restricting the communication between threads.

Threads have access to several memory spaces, as illustrated in Figure 1a. Each thread has a small private *local memory* space. In addition, all threads within the same block can communicate with each other through a low-latency *shared memory*. The much larger *global memory* has a higher latency and can be accessed by the CPU, thus being the only communication channel between CPU and GPU. The *GeForce GTX 285* graphics card running in our system consists of 30 multiprocessors with 8 stream processors each, resulting in a total of 240 cores and yielding a maximum theoretical speed of 1,063 GFLOPS. Each multiprocessor contains 16 KB of on-chip shared memory as well as 16,384 registers. The GPU-wide 1024 MB of global memory can be accessed with a maximum bandwidth of 159 GB per second.

In order to run hundreds of threads concurrently, the multiprocessors employ an architecture called SIMT (Single Instruction, Multiple Threads), visualized in Figure 1b. The SIMT unit of a multiprocessor creates, schedules and executes



(a) Thread hierarchy (b) SIMT architecture of CUDA-GPUs

Fig. 1: (a) Grids are subdivided into blocks, which consist of many parallel threads. Threads of the same block can cooperate through shared memory. (b) A device consists of several multiprocessors, each with their own memory spaces. (adapted from [7])

groups of 32 consecutive parallel threads. The highest efficiency can be achieved if all threads execute the same instruction path. Memory accesses from different threads can be *coalesced* into one memory transaction if consecutive threads access data elements from the same memory segment. Following such specific access patterns can dramatically improve the memory utilization and is essential for optimizing the performance of an application. Despite such optimizations, the CUDA framework is best suited for applications with a high *arithmetic density*, that is, where the number of memory transactions is small compared to the number of arithmetic operations.

3 CNN Architectures for SIMT Processors

The main concept of Convolutional Neural Networks (CNNs) is to extract local features at a high resolution and to successively combine these translation-invariant features into more complex features at lower resolutions. The loss of spatial information is compensated for by an increasing number of feature maps in the higher layers. Usually, CNNs consist of two altering types of layers, convolutional layers and subsampling layers. Each convolutional layer performs a discrete folding operation of its source image with a filter kernel. The subsampling layers reduce the size of the input by averaging neighboring pixels. Our architecture – shown in Figure 2 – diverges from this typical model because both the convolution and subsampling operations are performed in one step.

This modification was necessary due to the limited memory resources of the GPU hardware: During the backpropagation step it is required to store both the activities and the error signal for each feature map and each pattern. When combining both processing steps, the memory footprint of each feature map can thus be reduced by a factor of four when 2×2 subsampling is applied.

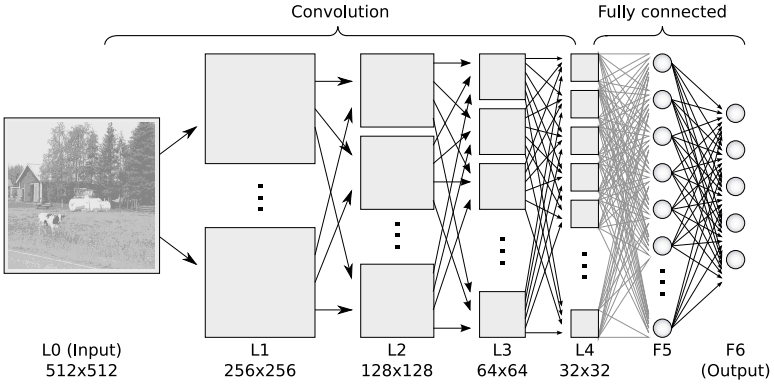


Fig. 2: Architecture of our CNN. The lower convolutional layers are locally connected with one convolutional kernel for each connection. The last convolutional layer L4 is followed by two fully connected layers.

The activity ${a_j}^{(l)}(x, y)$ of a neuron at position (x, y) in feature map j of layer l solely depends on the activities of neurons in a local receptive field of all feature maps from the preceding layer $l-1$. Each filter not only consists of its elements $w_{ij}(u, v)$ at position (u, v), but also of a bias weight b_{ij}. The net input $net_j(x, y)$ of feature map j is calculated over all source maps I:

$$net_j(x,y) = \sum_{i=0}^{I} \left(\left(\sum_{(u,v)} w_{ij}(u,v) \cdot a_i(2x+u, 2y+v) \right) + b_{ij} \right). \tag{1}$$

A non-linear sigmoid function – here we use hyperbolic tangent – is applied to determine the activity $a_j(x, y)$ of each neuron:

$$a_j(x,y) = f_{act}(net_j(x,y)) = \tanh(net_j(x,y)). \tag{2}$$

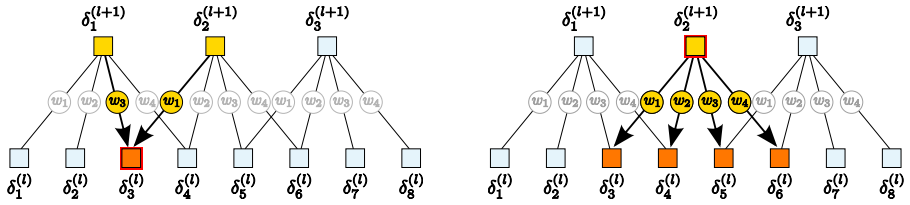
For our implementation, we used 8×8 kernels which are overlapping by 75% in each direction. The receptive field of a neuron increases exponentially on higher layers. Each neuron in layer L4 is influenced by an area of 106×106 pixels of the input image. A more technical advantage of this choice is that the $8 \times 8 = 64$ filter elements can be processed by 64 concurrent threads. In the CUDA framework, those 64 threads are coalesced into two *warps* of 32 concurrent threads each.

The filter weights are adapted with the gradient descent algorithm *backpropagation of error*. For one particular training pattern, the error signal δ_k of a neuron k in the output layer is calculated from the desired output t_k and the actual output o_k:

$$\delta_k = f'_{act}(net_k) \cdot (t_k - o_k). \tag{3}$$

For a fully connected hidden neuron j, this error is propagated backwards with

$$\delta_j = f'_{act}(net_j) \cdot \sum_k \delta_k w_{jk}. \tag{4}$$



(a) Pulling the error signals (b) Pushing the error signal

Fig. 3: Simplified example of the backpropagation pass for a one-dimensional case. (a) When the error signals are pulled by a neuron of the lower layer, discontinuous weights and neurons of the higher layer are involved. (b) When pushing the error signals, a neuron from the higher layer iterates the weights and accesses the error signals of continuous neurons.

For the convolutional layers it is error-prone to calculate the error due to the dimension-reducing filter kernels: Because of the subsampling operation, the index arithmetics for the error signal computations are complex. Therefore, we are employing a technique that Simard *et al.* [11] call "pushing" the error signals, as opposed to "pulling". Figure 3 depicts the difference between those operations for a 1D case. When the error is "pulled" by a neuron from the lower layer, it is tedious to determine the indices of the weights and neurons involved. On the other hand, "pushing" the error signal can be considered as the inversion of the forward pass, which enables us to use similar implementation techniques.

The convolutional kernel is applied at every position of a feature map, thus the weights are shared between several connections. The partial derivative of the error with respect to a weight $w_{jk}(u, v)$ is not only summed over all patterns P, but also over all positions (x, y) of the feature map:

$$\frac{\partial E}{\partial w_{jk}(u,v)} = \sum_{p=1}^{P} \sum_{(x,y)} \left(-\delta_k^{(l+1)}(x,y) \cdot a_j^{(l)}(2 \cdot x + u, 2 \cdot y + v) \right). \tag{5}$$

The weight update $\Delta w_{kj}(u, v)$ can then be calculated as

$$\Delta w_{ji}(u,v) = -\eta \cdot \frac{\partial E}{\partial w_{jk}(u,v)}, \tag{6}$$

with an adjustable learning rate of $\eta > 0$.

Determining a suitable learning rate can be difficult. Randomly initialized deep networks with more than three hidden layers often converge slowly and reach an inadequate local minimum [4]. During our experiments, it proved impossible to empirically choose a suitable learning rate for each layer. We therefore implemented the RPROP algorithm [10] which maintains an adaptive learning rate for each weight.

4 Implementation

To accelerate an application with the CUDA framework, it is necessary to split the problem into coarse-grained sub-problems which can be solved independently. Each task is mapped to a multiprocessor. Within each block, this task is further divided into finer pieces that can be solved cooperatively in parallel by many threads. We decided to assign each training pattern to one block, which implies that a mini-batch learning scheme has to be applied. During one batch learning pass, every pattern uses the same network parameters and weights.

The number of training patterns to be processed in parallel is restricted by the limited amount of global memory. With 32-bit precision, a feature map of 512×512 pixels occupies 1 MB of memory. The activities of all layers and all feature maps are required in the training pass, hence they all contribute to the memory footprint of one training pattern. Depending on the number of feature maps per layer and the size of the feature maps, up to 10 MB might be required for each pattern. When using the GeForce 285 GTX this means that only a few dozen patterns can be processed concurrently. At a finer scale, i.e., within a block, we perform the convolution of one source feature map onto eight target feature maps simultaneously. With this setup, the activities of the source feature map need to be loaded only once for every eighth target feature map. This dramatically reduces the amount of memory transfers. To simplify the parallel implementation described here, we will only consider the case of 64×64 pixel source feature maps and 32×32 pixel target feature maps. It can be expanded for multiples of these sizes with little effort.

Because shared memory is extremely limited, it is critically important to reuse loaded data as often as possible. Even if the whole 16 KB of shared memory is used, it can only hold a small fraction of the source feature map. For this reason, we are utilizing the shared memory as a circular buffer which only holds a small region of the source feature map, as shown in Figure 4. During each iteration only two rows of this buffer need to be exchanged. A 70×8 window of

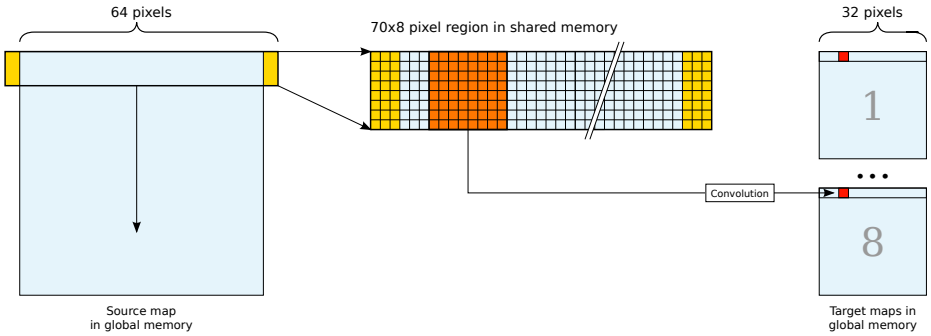


Fig. 4: Convolution operation. A region of the source map is loaded from global memory into the circular buffer in shared memory. Each thread performs a convolution on this section and subsequently writes the result into the target map in global memory.

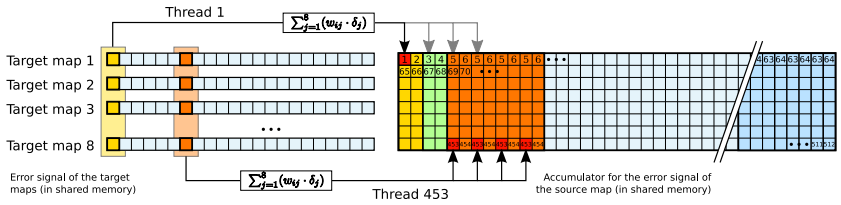


Fig. 5: Aggregation of the backpropagated error signals. Each thread first sums the partial error signal over one position in all eight target map rows. Then it writes the result to the accumulator in shared memory. As indicated by the thread numbers on the right, each thread of a warp writes to a different memory bank. Thus, bank conflicts are avoided. This step is repeated four times to cover each element of a filter.

the source feature map is maintained within shared memory. This area includes the 64 columns of the source map as well as a 3 pixel wide border on both sides. The 8×8 convolutional filter is applied at every other pixel, thus fitting exactly 32 times into this 70×8 window.

The source feature map is being iterated from top to bottom. During each iteration, three steps are performed: (1) Loading two rows of 70 consecutive pixels of the source feature map from global into shared memory with a coalesced transfer operation. (2) Iterating through each filter element to compute the convolution. (3) Writing 32 consecutive pixels, each computed by continuous threads, back to global memory. Since this kernel calculates the output of a 32 pixel wide row for eight target feature maps, it uses 256 threads. The source image window only requires $70 \times 80 \times 4 = 2240$ Byte of shared memory. Therefore, it is theoretically possible to run seven thread blocks on one multiprocessor at once. In practice, only four blocks can be scheduled simultaneously because each thread requires at least 16 registers. Data transfers are minimized as each loaded pixel is reused 128 times (at 16 positions for each of the 8 target maps). The result of our optimizations is that the runtime is bounded by the maximum arithmetic performance of the hardware as opposed to being bounded by the memory bandwidth.

During the backpropagation step, two tasks are performed simultaneously: the weight gradients are calculated for each element of the eight convolutional filters and the error signal is accumulated on the lower layer. For both operations, the error signal of the higher layer is required. Hence, it is reasonable to reuse this data once it is loaded. We are again handling eight convolutional filters during one device function call: 512 threads can be employed for this, one for each of the eight 8×8 filters. Similar to the forward pass, we are using a circular buffer to load the activities of the source map ${a_i}^{(l)}$ from global memory into shared memory. An additional circular buffer is used as an accumulator for the backpropagated error signals ${\delta_j}^{(l+1)}(x, y)$ of the target maps.

One thread runs through all 32 pixels of the row for one specific feature map. It multiplies the error signal with the activity of the source map pixel corresponding to its weight position and accumulates this value in a register.

Before the kernel function terminates, the final value for the weight gradient is written to global memory. Access to the error signals of the target maps is coalesced because all 32 pixels of a row are loaded simultaneously. Similarly, loading the error signals from shared memory is optimal. All of the 64 threads (two warps) of a specific filter access the same memory location, which enables the CUDA driver to broadcast this value to all threads. Write-access to global memory is coalesced as well, because all 64 elements of a filter are stored in succession.

As described in Section 3, we invert the forward pass to propagate the error signal $\delta_j^{(l+1)}(x, y)$ from the target layer $l + 1$ back to the source layer l. The first step is to accumulate the partial error signal in a circular buffer in shared memory. Once these values are pushed out of the circular buffer, they are added to the partial error signals in a temporary memory space in global memory.

This part of the algorithm reuses the error signals of the target maps which already reside in shared memory from the weight gradient computation. In order to avoid bank conflicts when accessing shared memory, our thread indexing scheme is not straight-forward, as illustrated in Figure 5. Each thread calculates and sums the partial error signal at a specific position for all eight target feature map rows. This sum is then accumulated in the shared memory circular buffer. As indicated in the figure, consecutive threads write to consecutive memory elements, thus avoiding bank conflicts. This step is repeated four times.

The ratio between memory transactions and arithmetic operations for the backpropagation pass is high because the error signals of the target maps are reused 64 times.

5 Evaluation

To verify the correctness of our implementation, we evaluated our CNN architecture on the normalized-uniform NORB dataset [6]. This dataset consists of 24,300 binocular grayscale training images and the same number of testing images, depicting toy figures of five categories. After training on the raw pixel data for 360 epochs (291,000 mini-batch updates), an error rate of 8.6% on the test set was achieved. To evaluate our architecture on the MNIST dataset of handwritten digits [5], we scaled the input images from 28×28 pixels to 256×256, because our implementation requires large input patterns. After 10 epochs of backpropagation, this network achieved a test error rate of 1.38%.

The speed of our implementation was benchmarked on a system with an *Intel Core i7 940* (2.93 GHz) CPU and a *Nvidia GeForce GTX 285*. For a comparison of the runtime, we implemented a functionally equivalent single-threaded, sequential CPU version. The most critical components of the parallel implementation are the forward pass and the the backpropagation of error. For both, several patterns are processed in parallel, as described in Section 4. Thus, it is to be expected that a larger speedup can be achieved for an increasing number of patterns. Figure 6 shows the runtime of the setup described above as a function of the number of patterns processed in parallel. For any value between 1 and 30 patterns, the

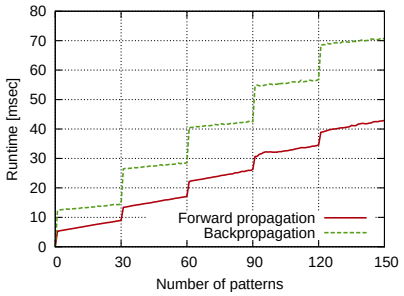


Fig. 6: Runtime of the forward and backpropagation passes as a function of the number of training patterns.

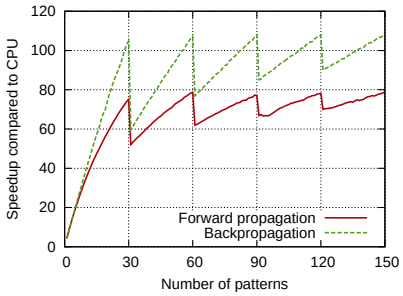


Fig. 7: Speedup of the parallel GPU implementation compared to a serial CPU version.

runtime remains almost constant, with a sudden jump at 31 patterns which is repeated every 30 patterns. Thus, as shown in Figure 7, the highest speedup was achieved when the number of patterns is a multiple of 30. This discontinuity is owed to the hardware: the GTX 285 GPU consists of 30 multiprocessors, each processing one pattern. If more then 30 patterns are scheduled, the remaining patterns are queued until another multiprocessor has finished.

When choosing a suitable number of patterns, the forward pass can be accelerated by a factor of 80. A maximum speedup factor of 110 is achieved for the backpropagation pass. In addition to the forward and backward pass, memory transfers are necessary to copy the training patterns from host memory to device memory. The CPU version has the advantage of using hardly any explicit memory transfers and memory access is accelerated by caching. On the contrary, for parallel GPU implementations memory transactions are often a bottleneck. To ensure a realistic comparison it is therefore necessary to include memory transfers in the total runtime of our GPU implementation.

We timed the total runtime of both our serial and parallel implementations for networks with a varying number of feature maps on each layer and with varying input sizes. Dispite memory transactions being included in the benchmark results in Table 1 speedups of two orders of magnitude are achieved. Thus, we conclude that data transfer times are negligible compared to forward and backward propagation. We can also deduce that neither network size, nor the size of the input has a significant influence on the speedup of our implementation. In all cases, the gradient descent mini-batch learning was accelerated by a factor ranging from 95 to 115.

6 Conclusion

Our work shows that current graphics cards programming frameworks with their hierarchy of threads are very well suited for a parallel implementation of CNNs. In comparison to a serial CPU implementation, we were able to significantly

Input Size	Feature maps	CPU [ms]	GPU [ms]	Speedup
256×256	1-8-8-8	14,045	141	99.8
256×256	2-8-16-32	44,242	383	115.7
256×256	4-16-64-64	278,010	2,583	107.6
512×512	1-8-8-8-8	53,495	561	95.4
512×512	2-8-16-32-64	225,110	2,045	110.1
512×512	4-8-32-64-64	545,803	5,143	106.1

Table 1: Benchmarks for one epoch with 60 training patterns, including all memory transfers. The networks have two fully connected layers with 100 and 10 neurons.

speed up the learning algorithm for a large convolutional neural network by a factor ranging from 95 to 115 on a single GPU. Until now, it was impossible to study deep CNNs with high-resolution input images due to the tremendous computational power required for their training. The aim of our future work is to extend this approach to systems with multiple GPUs and to apply large-scale CNNs to pattern recognition tasks using datasets with millions of high-resolution natural images.

References

1. S. Behnke. *Hierarchical Neural Networks for Image Interpretation*, volume 2766 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
2. K. Chellapilla, S. Puri, and P. Simard. High Performance Convolutional Neural Networks for Document Processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule, France, 2006. Université de Rennes.
3. S. Lahabar, P. Agrawal, and P. J. Narayanan. High Performance Pattern Recognition on GPU. In *Proc. of National Conference on Computer Vision, Pattern Recognition, Image Processing and Graphics*, January 2008.
4. H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin. Exploring Strategies for Training Deep Neural Networks. *Journal of Machine Learning Research*, 2009.
5. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
6. Y. LeCun, F. Huang, and L. Bottou. Learning Methods for Generic Object Recognition with Invariance to Pose and Lighting. In *Proc. of CVPR'04*, 2004.
7. Nvidia Corporation. CUDA Programming Guide 3.0, February 2010.
8. M. Osadchy, Y. LeCun, and M. Miller. Synergistic Face Detection and Pose Estimation with Energy-Based Models. *Journal of ML Research*, 2007.
9. R. Raina, A. Madhavan, and A. Y. Ng. Large-scale deep unsupervised learning using graphics processors. ACM Intl. Conference Proceeding Series, 2009.
10. M. Riedmiller and H. Braun. RPROP – A fast adaptive learning algorithm. In *Proc. of the Int. Symposium on Computer and Information Science VII*, 1992.
11. P. Y. Simard, D. Steinkraus, and J. C. Platt. Best Practice for Convolutional Neural Networks Applied to Visual Document Analysis. In *ICDAR*, 2003.
12. R. Uetz and S. Behnke. Large-scale Object Recognition with CUDA-accelerated Hierarchical Neural Networks. In *Proc. of ICIS*, 2009.
13. N. Šerbedžija. Simulating artificial neural networks on parallel architectures, 1996.