2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing

Performance and Scalability of GPU-based Convolutional Neural Networks

Daniel Strigl, Klaus Kofler and Stefan Podlipnig
Distributed and Parallel Systems Group, Institute of Computer Science
University of Innsbruck, Technikerstrasse 21a, 6020 Innsbruck, Austria
{*Daniel.Strigl, Klaus.Kofler*}*@student.uibk.ac.at, Stefan.Podlipnig@uibk.ac.at*

***Abstract*—In this paper we present the implementation of a framework for accelerating training and classification of arbitrary Convolutional Neural Networks (CNNs) on the GPU. CNNs are a derivative of standard Multilayer Perceptron (MLP) neural networks optimized for two-dimensional pattern recognition problems such as Optical Character Recognition (OCR) or face detection. We describe the basic parts of a CNN and demonstrate the performance and scalability improvement that can be achieved by shifting the computation-intensive tasks of a CNN to the GPU. Depending on the network topology training and classification on the GPU performs 2 to 24 times faster than on the CPU. Furthermore, the GPU version scales much better than the CPU implementation with respect to the network size.**

***Keywords*-machine learning; convolutional neural networks; GPGPU; CUDA; performance; scalability;**

I. Introduction

Neural Networks (NNs) perform very well on pattern recognition tasks with a large amount of training data. For image classification, like Optical Character Recognition (OCR), Convolutional Neural Networks (CNNs) deliver state-of-the-art performance [1]. CNNs are a derivative of Multilayer Perceptron (MLP) neural networks optimized for two-dimensional pattern recognition. The area of applications for CNNs is widespread. They are used for handwriting recognition [1] [2], face, eye and license plate detection [3] [4] [5], and in non-vision applications such as semantic analysis [6].

The biggest drawback of CNNs, besides a complex implementation, is the long training time. Since CNN training is very compute- and data-intensive, training with large data sets may take several days or weeks. The huge number of floating point operations and relatively low data transfer in every training step makes this task well suited for GPGPU (General Purpose GPU) computation on current Graphic Processing Units (GPUs). The main advantage of GPUs over CPUs is the high computational throughput at relatively low cost, achieved through their massively parallel architecture.

In the past using the GPU for general purpose calculations required a deep understanding of the hardware architecture, where the problems had to be implemented using a graphics API like OpenGL or DirectX. Therefore, the algorithms had to be transformed into a graphics pipeline friendly format. With the emergence of CUDA (Compute Unified Device Architecture) [7] and the so-called unified shader architecture [8] things have changed. The CUDA language bears resemblance to the C programming language and is therefore much more common for a programmer than the graphics API languages. This results in a shorter training period, faster adoption and higher efficiency. Furthermore, unified shaders are better adapted to perform general computations than earlier architectures. Probably, the biggest drawback of CUDA is its limitation to the NVIDIA hardware, but future languages like OpenCL [9] or DirectX 11 Compute Shader [10] will solve this problem. It is expected that these manufacturer independent languages will make GPGPU computing even more popular.

In contrast to other classifiers like Support Vector Machines (SVMs) where several parallel implementations for CPUs [11] and GPUs [12] exist, similar efforts for CNNs are missing. Therefore, we implemented a high performance library in CUDA to perform fast training and classification of CNNs on the GPU. Our goal was to demonstrate the performance and scalability improvement that can be achieved by shifting the computation-intensive tasks of a CNN to the GPU. We will describe some scalability experiments in the following sections.

The paper is organized as follows: The next section introduces CNNs and describes their basic building blocks. Section III describes some characteristics of CUDA, followed by Section IV with a brief outline of other publications in this research area. Our implementations are presented in Section V, while Section VI presents our benchmarks and the obtained results. Finally, Section VII concludes this paper.

II. Convolutional Neural Networks

The traditional approach for two-dimensional pattern recognition is based on a feature extractor, the output of which is fed into a neural network. This feature extractor is usually static, independent of the neural network and not part of the training procedure. It is not an easy task to find a "good" feature extractor because it is not part of the training procedure and therefore it can neither adapt to the network topology nor to the parameters generated by the training procedure of the neural network.

CNNs make this difficult task part of the network and act as a trainable feature extractor with some degree of shift, scale, and deformation invariance [2]. They are composed of three different types of layers: convolutional layers, subsampling layers (optional), and fully connected layers. These layers are arranged in a feed-forward structure as shown in Fig. 1. The convolutional layers are responsible for the feature extraction (edges, corners, end points or non visual features in other signals), using the

1066-6192/10 $26.00 © 2010 IEEE
DOI 10.1109/PDP.2010.43

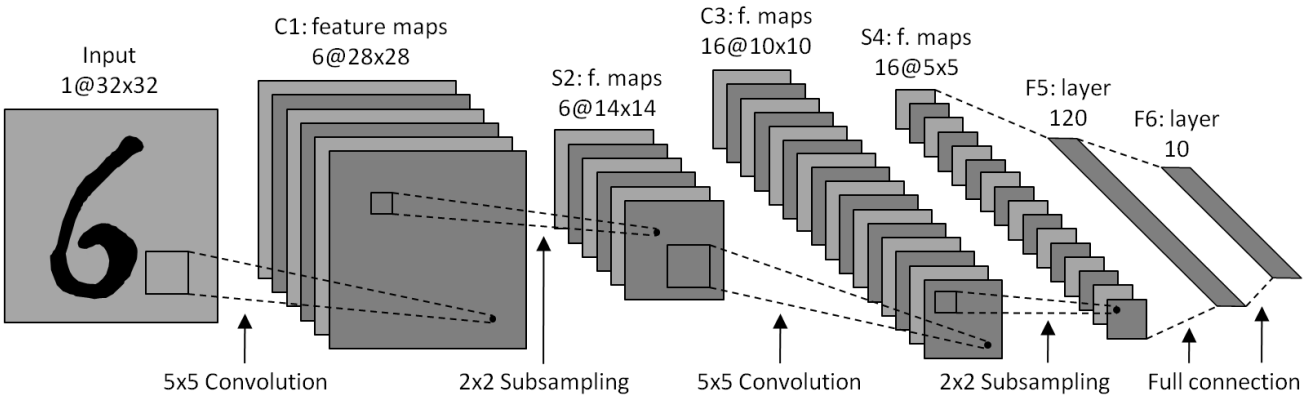


Figure 1. Architecture of the LeNet5 [2] variant used in this work (based on Fig. 2 in [2]).

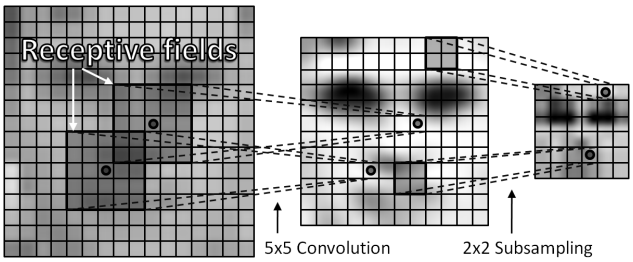


Figure 2. Illustration of the convolution and subsampling process inside a CNN (based on Fig. 3.5 in [13]).

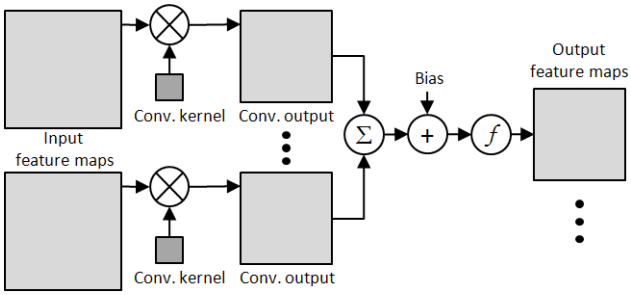


Figure 3. The convolutional layer of a CNN (from [14]).

two key concepts of local receptive fields and shared weights. Since the exact location of the detected features are less important but the relative position to other features are relevant, the succeeding subsampling layer performs a local averaging and subsampling. This reduces the shift and distortion sensibility of the detected features. The fully connected layer acts as a normal classifier similar to the layers in traditional MLP networks. A detailed description of the architectural concepts behind these layers is given in [2], while a brief explanation of the composition and the mathematical model of these layers is given in the following subsections.

A. Convolutional Layer

The convolutional layers are the core of any CNN. A convolutional layer consists of several two-dimensional planes of neurons, the so-called feature maps. Each neuron of a feature map is connected to a small subset of neurons inside the feature maps of the previous layer, the so-called receptive fields (see also Fig. 2). The receptive fields of neighboring neurons overlap and the weights of these receptive fields are shared through all the neurons of the same feature map. The feature maps of a convolutional layer and its preceding layer are either fully or partially connected (either in a predefined way or in a randomized manner).

First, the convolution between each input feature map and the respective kernel is computed. Corresponding to the connectivity between the convolutional layer and its preceding layer these convolution outputs are then summed up together with a trainable scalar, known as the bias term. Finally, the result is passed through an activation function (e.g. tanh). An illustration of this process is given in Fig. 3.

The output $y_n^{(l)}$ of a feature map n in a convolutional layer l is given by

$$y_n^{(l)}(x,y) = f^{(l)}\left(\sum_{m \in M_n^{(l)}} \sum_{(i,j) \in K^{(l)}} w_{mn}^{(l)}(i,j) \cdot y_m^{(l-1)}(x \cdot h^{(l)} + i, y \cdot v^{(l)} + j) + b_n^{(l)} \right) \quad (1)$$

where $K^{(l)} = \left\{ (i,j) \in \mathbb{N}^2 \mid 0 \leq i < k_x^{(l)}; 0 \leq j < k_y^{(l)} \right\}$, $k_x^{(l)}$ and $k_y^{(l)}$ are the width and the height of the convolution kernels $w_{mn}^{(l)}$ of layer l, and $b_n^{(l)}$ is the bias of feature map n in layer l. The set $M_n^{(l)}$ contains the feature maps in the preceding layer $l-1$ that are connected to feature map n in layer l. The values $h^{(l)}$ and $v^{(l)}$ describe the horizontal and vertical step size of the convolution in layer l (usually 1), while $f^{(l)}$ is the activation function of layer l.

B. Subsampling Layer

To reduce the size of consecutive feature maps a subsampling layer is usually placed between two convolutional layers. This type of layer reduces the outputs of a certain number of adjacent neurons (normally a square of 2×2 neurons) of a feature map in the previous layer to a single value. Afterwards it multiplies a single weight to this value, adds a bias and passes the result through an activation function to obtain the result of the output feature

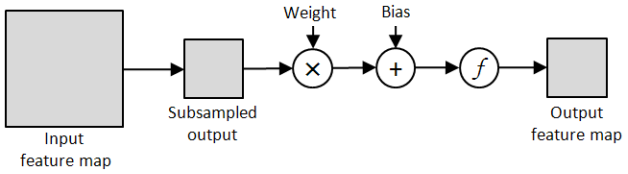


Figure 4. The subsampling layer of a CNN (from [14]).

map. Subsampling layers have the same number of feature maps as the preceding convolutional layer, where each feature map of the subsampling layer is always connected to the corresponding one in the previous convolutional layer (1-to-1 connection).

The output $y_n^{(l)}$ of a feature map n in the subsampling layer l is calculated according to

$$y_n^{(l)}(x,y) = f^{(l)}\left(w_n^{(l)} \cdot \sum_{(i,j)\in S^{(l)}} y_n^{(l-1)}(x\cdot s_x + i, y\cdot s_y + j) + b_n^{(l)}\right) \quad (2)$$

where $S^{(l)} = \left\{(i,j) \in \mathbb{N}^2 \mid 0 \le i < s_x^{(l)}; 0 \le j < s_y^{(l)}\right\}$, $s_x^{(l)}$ and $s_y^{(l)}$ define width and height of the subsampling kernel of layer l, and $b_n^{(l)}$ is the bias of feature map n in layer l. The value $w_n^{(l)}$ is the weight of feature map n in layer l and $f^{(l)}$ the activation function of layer l.

Fig. 4 illustrates the process that is performed by a subsampling layer. As subsampling layers are optional a simpler CNN can be build by replacing the consecutive convolutional and subsampling layers with a convolutional layer that uses a step size greater than 1 (for an example see [1]).

C. Fully Connected Layer

After the convolutional and subsampling layers one or more fully connected layers follow. These layers are always used in a CNN and are similar to the layers in a standard MLP. In those layers the outputs of all neurons in layer $l-1$ are connected to every neuron in layer l.

The output $y^{(l)}(j)$ of neuron j in a fully connected layer l is given by

$$y^{(l)}(j) = f^{(l)}\left(\sum_{i=1}^{N^{(l-1)}} y^{(l-1)}(i)\cdot w^{(l)}(i,j) + b^{(l)}(j)\right) \quad (3)$$

where $N^{(l-1)}$ is the number of neurons in the preceding layer $l-1$, $w^{(l)}(i,j)$ is the weight for the connection from neuron i in layer $l-1$ to neuron j in layer l, and $b^{(l)}(j)$ is the bias of neuron j in layer l. As for the other two layers, $f^{(l)}$ represents the activation function of layer l.

Fig. 2 illustrates the process of a 5×5 convolution followed by a 2×2 subsampling inside a CNN, while an example for a possible composition of convolutional, subsampling and fully connected layers is shown in Fig. 1.

CNNs are usually trained with a variant of the gradient-based backpropagation method [15]. All training patterns along with the expected outputs are fed into the network. Afterwards the network error (the difference between the actual and expected output) is backpropagated through the network and used to compute the gradient of the network error w.r.t. the weights. This gradient is then used to update the weight values according to a specific rule (e.g. stochastic, momentum, etc.) [13] [14] [15]. For a description of the backpropagation algorithm in CNNs the reader is referred to [13] and [14].

III. Compute Unified Device Architecture

CUDA (Compute Unified Device Architecture) [7] describes a proprietary language by NVIDIA which is based on C and contains some special extensions to enable efficient programming of NVIDIA's graphic processors since the G80 (released 2006). The extensions mainly cover commands to enable multithreading on GPU and to access the different types of memory on the GPU.

In contrast to the traditional approach of programming GPUs through graphics shading languages like OpenGL or DirectX, a programmer does not need experiences in computer graphics to be able to write GPGPU applications using CUDA. However, certain knowledge of the hardware is indispensable to write efficient and fast GPGPU programs. Every CUDA device consists of a certain number of so-called Streaming Multiprocessor (SM). Each SM contains eight Shader Units (SUs), a Multithreaded Instruction Unit and on-chip Shared Memory that can be accessed by all eight SUs. Every SU can perform one multiplication and one MAD operation (a floating point multiplication followed by a floating point addition on the result) every clock cycle, but the whole SM can only perform the same piece of code on different data using multiple threads. This parallel computing architecture is called SIMT (Single Instruction, Multiple Threads). Furthermore, a SM can issue a new command only every fourth clock cycle of the SU, which means that the same command has to be executed at least 32 times in distinct threads to totally utilize a single SM.

The CUDA programming model reflects the specific hardware topology of these GPUs. Fig. 5 shows how threads are grouped and mapped to the hardware in CUDA. At startup of a function on the GPU (a so-called kernel) the system creates a certain number of threads defined by the programmer. The entirety of all those threads is called the grid. The grid is composed of a specific number of thread blocks. These blocks are arranged in a two-dimensional manner on the grid with a maximum size of 65,535×65,535 blocks. Each block is assigned to one SM and the threads in a block are arranged in a three-dimensional array. The maximum size of each dimension of a block is 512×512×64, but the maximum number of threads in a block cannot exceed 512 threads (for performance reasons each block should contain at least 32 threads). Each thread has access to various kinds of memory with different characteristics (see also Fig. 5). Using the most appropriate memory the right way (e.g. coalesced access to global memory, avoiding bank conflicts in shared memory) is one of the most

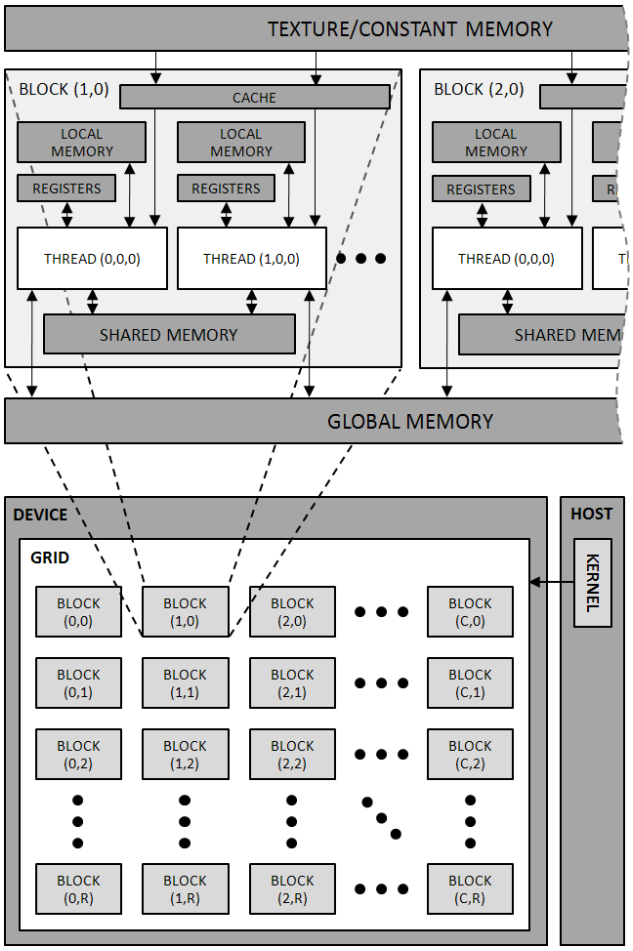


Figure 5. CUDA threading and memory topology.

effective means of improving performance [16].

Because of the manycore structure of actual GPUs they are very well suited for any application with a lot of floating point operations that can be processed in parallel. Some important performance numbers and a comparison to a CPU are listed in Table I. This table describes the actual hardware used for our experiments. Compared to the CPU the GPU numbers look quite impressive, but it should be considered that the peak performance of 1010.8 GFLOPS/s promoted by NVIDIA can only be achieved if every SM can execute 8 multiplications and 8 MAD operations at the same time.

One drawback of today's GPGPU programming is the rather slow data transfer from the GPU's memory to the main memory of the CPU. This is not a big disadvantage when running a neural network because the amount of data that has to be copied from and to the GPU (the network's input and its output) is relatively small in comparison with the big matrices that must be handled inside the network.

Another downside of the actual NVIDIA GPUs is that they either do not support double precision (DP) floating point numbers (G80 – G98, compute capability 1.0 and 1.1) or peak performance drops to approximately one tenth of single precision (SP) performance (on GT200, compute capability 1.2). Furthermore, these GPUs are not fully IEEE-754 (IEEE Standard for Floating-Point Arithmetic) compliant [7]. However, this does not seem to be any problem when it comes to neural networks. Our tests showed no difference between single and double precision nor between CPU and GPU implementation in terms of the classification rates of the tested networks.

IV. Related Work

During the last years much work has been investigated to improve the performance of neural networks on the GPU. Some of the first GPU implementations of neural networks were presented in [17] and [18]. While the first work only enhanced the performance of the classification part of a neural network using the vertex and pixel shader of a GPU, the latter one also accelerated the training of a neural network.

The first porting of a CNN to the GPU has been released in [19]. It was limited to the network proposed in [1] using the outdated architecture of vertex and pixel shader and implemented through the DirectX API.

In [20] one of the first neural networks for the new unified shader architecture using CUDA was implemented. A GPU implementation of a so-called Neocognitron neural network, which is quite similar to a CNN, was presented in [21]. This work only focuses on the improvement of the recognition part on the GPU, training of this network is not considered.

However, as far as we know, no prior effort was made to build a complete framework for accelerating training and classification of arbitrary CNNs on modern GPUs.

V. Implementation

We implemented a high performance but still flexible library in C++ and CUDA to accelerate the training and classification process of arbitrary CNNs. Due to the fact that the ideal parameters of a neural network (network structure, initial value range, learning method, learning rate, etc.) can only be determined by testing and evaluating, shortening the training time often leads to better results. We started with a straight forward implementation without any manual parallelization or vectorization ($\mathrm{CPU}_{\mathrm{triv.}}$). To fairly compare the GPU with the CPU variant of our library, we optimized this implementation using functions from Intel's Performance Libraries IPP (Integrated Performance Primitives, ver. 6.1) [22] and MKL (Math Kernel Library, ver. 10.2) [23] ($\mathrm{CPU}_{\mathrm{opt.}}$). Those libraries take the full advantage of the newest Streaming SIMD Extensions (SSE) of the CPU. These enhancements resulted in a quite fast implementation.

The GPU implementation (GPU) using CUDA exchanges the mathematical vector and matrix operations with functions either from NVIDIA's CUBLAS Library [24] if appropriate functions are available there or our own implementations otherwise. Each kernel-function performs one mathematical operation, e.g. a matrix-vector multiplication or the summation of all elements in a vector. In order to not affect the C++ class structure of the CPU implementation each CUBLAS or self implemented function (kernel-call) is embedded in a templatized C++

Table I
TECHNICAL SPECIFICATIONS OF THE HARDWARE USED FOR PERFORMANCE MEASUREMENTS IN THIS PAPER

	Core i7 860	GeForce GTX 275
Processsor core clock	2800 MHz	633 MHz
ALU clock	5600 MHz	1404 MHz
Memory size	4096 MB	896 MB
Bandwidth core ↔ memory	21.3 GB/s	127.0 GB/s
Number of processor cores	4	30
Local cache per core	64 KB L1 + 512 KB L2 + 2048 KB L3	16 KB (shared) + 8 KB (texture) + 8 KB (constant)
SP FLOPS / core and clock cycle	4 MUL or ADD	8 MUL and 8 MAD
Total SP FLOPS peak performance	89.6 GFLOPS/s	1010.8 GFLOPS/s
Termal Design Power	95 Watt	216 Watt

function. This means that after every operation on the GPU the program control is passed back to the CPU, even if no data transfer between CPU and GPU is necessary. Although this introduces some small overheads the general reusable structure of our library is preserved. During the tests of the networks running on the GPU we noticed a CPU load of approx. 12% on our (virtual) eight-core test machine, which means that one core was fully occupied.

The following paragraphs describe some characteristics of the implemention, while the interested reader is referred to our implementation. In order to encourage further development in this research area we have made our source code publicly available [25].

A. Making CNNs simpler

In most implementations of standard MLP networks each layer forwards its activation to the next layer during forward propagation and "pulls" the error back from the following layer during backpropagation. This has two major drawbacks for CNNs: First, it is very complicated to implement it for CNNs because, caused by border effects, the number of outgoing connections is not equal for all neurons in a convolutional layer. Second, every layer has to know the type of its subsequent layer. This is quite cumbersome and not beneficial for a flexible library. Therefore, we used the easier and more flexible approach to "push" the error back to the previous layer as described in [1]. The advantage of this method is that the number of incoming connections of each neuron in a layer, even in a convolutional one, is always constant. Furthermore, this technique does not require any knowledge about the neighboring layers and the resulting regular structure of the backpropagation process can be better optimized.

B. Making CNNs faster

Convolutions are not easy to optimize because of their irregular memory access pattern. The data is not accessed in the same order as it resides in memory and not all values are accessed equally often. To make access patterns more linear we used the "unfolding" technique (see [19], section 2). The input is copied to a matrix where the elements of each convolutional kernel form one row (in [19]) or one column (in our implementation [25]). Therefore, most of the input elements appear several times in this matrix. Once the unfolding is done the forward- and backpropagation of the convolutional layer can be implemented as a matrix product. This does not only result in an easier but also a much better optimizable implementation.

C. Debugging

Since incorrect implementations of a neural network sometimes yield reasonable results we proved the correctness of our implementations with two methods. We computed the Jacobian matrix (see [26], page 148) using the backpropagation function and an arbitrarily accurate estimate of it by adding small variations to the input and calling the forward propagation function. Next, we applied the comparison described in [1]. Furthermore, we proved the correctness of the gradients computed by the backpropagation function by comparing them with gradients computed via finite differences (see [27], subsection 4.4).

VI. BENCHMARKS AND RESULTS

All benchmarks in this paper were performed in single precision on an Intel Core i7 860 with a GeForce GTX 275 running Ubuntu 9.04 (system price in fall 2009 approx. € 900). The technical specifications of these two processors are shown in Table I. All benchmarks consisted of 1,000 training iterations of the networks described in subsection VI-A. Such a training iteration is composed of one forward propagation (a training pattern is fed into the network and produces some output), one backpropagation (based on the difference between the actual and the desired output, a gradient for every single weight in the network is calculated) and the weights update (the gradients calculated during backpropagation are multiplied with the learning rate and added to the actual weights). In case of the CUDA version the time to copy the training pattern to the GPU and the result to the main memory is also considered. For each iteration a separate input pattern was used. Note that we were interested in the scaling behavior of our implementations. To test this behavior with different settings we decided to restrict our tests to the basic case (one iteration for each pattern) and a small number of input patterns (1,000).

For our performance and scalability tests we performed measurements on two different networks: The network proposed by Simard et al. in [1] (called SimardNet from now on) and the LeNet5 [2]. In all benchmarks we compared the three different implementations explained in the previous section ($\mathrm{CPU}_{\mathrm{triv.}}$, $\mathrm{CPU}_{\mathrm{opt.}}$, GPU). As compiler we used the GNU C++ Compiler (g++, version 4.3.3) to generate the CPU executables and NVIDIA's

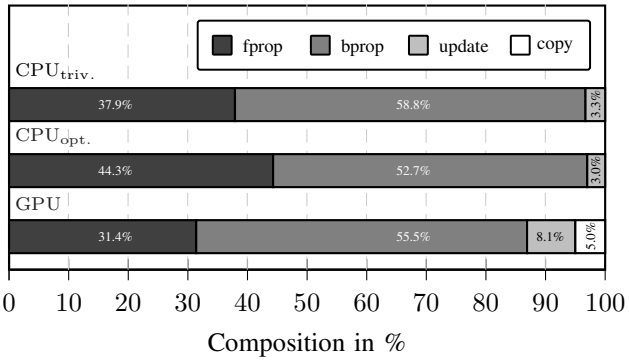


Figure 6. Execution time composition of the three different implementations performing 1,000 learning iterations of a LeNet5.

CUDA Compiler (nvcc, version 2.3) to generate the code that runs on the GPU. Compiler optimizations were turned on for the g++ compiler (option -O3). Because we did not observe any performance improvements when turning on the compiler optimization for the nvcc compiler we used the standard settings. The basic patterns for our input data stem from the well known MNIST database [28]. It is one of the most widely known pattern classifier benchmarks and consists of size-normalized and centered images of handwritten digits $(0-9)$.

A. Tested Networks

The SimardNet has been suggested by Simard et al. as a fast and simple CNN for OCR [1]. It consists of two convolutional and two fully connected layers. The convolutional layers use a step size of two, which makes subsampling layers superfluous. The typical recognition rate on the MNIST database (without any extension of the dataset) is about 99.1% [29].

The LeNet5 has been proposed by Yann LeCun, the inventor of CNNs, in [2]. The variant tested in this paper is composed of three convolutional, two subsampling and two fully connected layers as shown in Fig. 1. The typical recognition rate on the MNIST database [28] (without any extension of the dataset) is about 98.8%.

B. Composition of Execution Time

In our first experiment we investigated the partition of the LeNet5's training time for the three different CNN versions. The result is shown in Fig. 6. As one can clearly see, the backpropagation part (bprop) takes most of the overall time in all three versions. The forward propagation (fprop) is although quite time consuming. The weight update (update) consumes less time. Although the absolute execution time is quite different the percent distribution of the different parts is rather similar (except the weight update for the GPU version). The overhead caused by copying data (copy) to and from the GPU is quite small in the GPU version (a few percentages of the overall execution time).

C. Scaling Input Size

In this benchmark we scaled the input size of the training patterns fed to a LeNet5. Increasing the input size automatically increases the number of neurons in the convolutional and subsampling layers and the number of trainable parameters (weights, biases). The input's side length was increased stepwise by eight pixels as shown in Table II. Fig. 7(a) shows the execution time of all three implementations with different input sizes. While the CPU version using Intel's Performance Libraries clearly outperforms the trivial implementation, the CUDA version is not only the fastest one but it also scales best with the input size. This is underlined by Fig. 7(b) which shows the speedup of the GPU version in comparison to the trivial CPU version ($\text{CPU}_{\text{triv.}}/\text{GPU}$) and to the optimized CPU version ($\text{CPU}_{\text{opt.}}/\text{GPU}$). The speedup grows with the input size and the GPU version definitely scales better than the CPU versions with large input sizes.

D. Scaling Feature Maps and Inner Neurons

The last benchmark shows how the implementations scale with the number of neurons inside the network. We tested the SimardNet with all possible combinations resulting from doubling the feature maps/neurons of all inner layers. The input size remained fixed at 29×29, while the output size was set to 10. The properties of the tested network's variants are listed in Table III. As shown in Fig. 8(a) and 8(b) the optimized CPU variant scales better than the trivial one when doubling the feature maps in one of the first two inner layers or the neurons in the third inner layer. However, the GPU version indisputable scales best with the size of any inner layer.

VII. Conclusion and Future Work

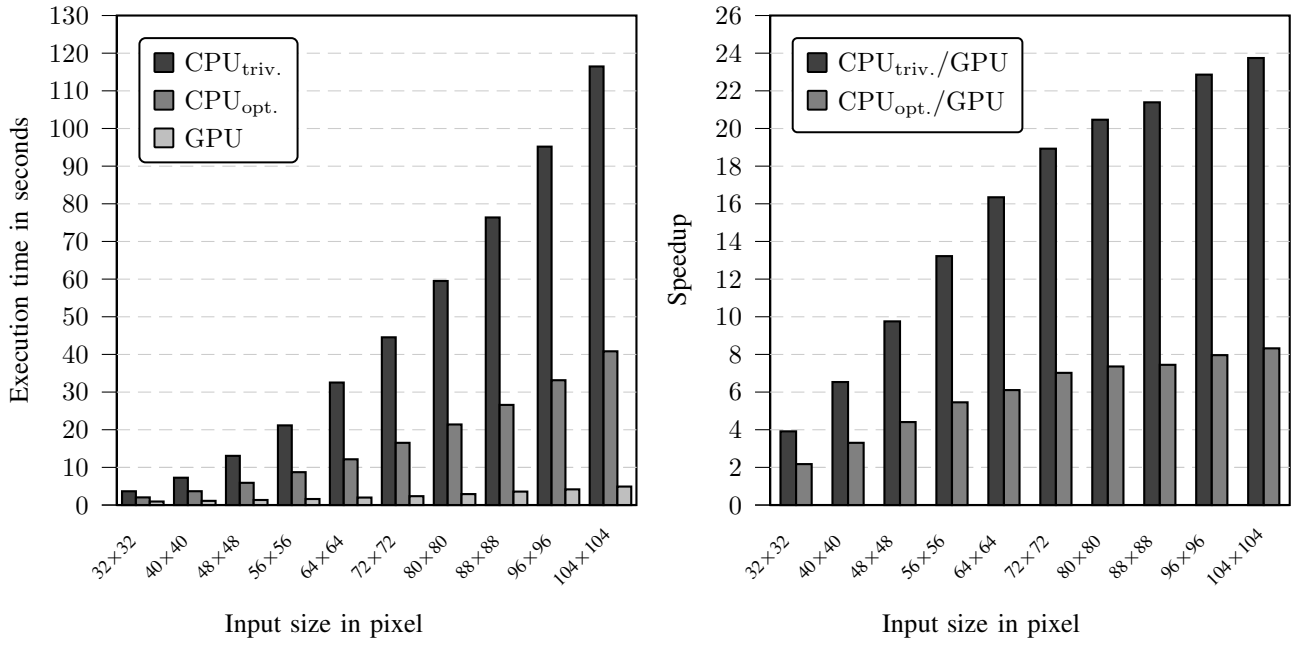
This article shows that GPUs work quite well for convolutional neural networks. The relatively low amount of data to transfer to the GPU for every pattern and the big matrices that have to be handled inside the network seem to be appropriate for GPGPU processing. Furthermore, our experiments showed that the GPU implementation scales much better than the CPU implementations with increasing network size.

For future work we plan to use the GPU version to accomplish complex experimental tests with different network topologies and parameter settings (e.g. different learning rates). The whole MNIST database includes 60,000 training patterns and we have to perform several training iterations for each training pattern. Therefore, one single test run (training and evaluation) can take days with the trivial CPU version. The GPU version will enable a faster execution of these tests and facilitate experiments with alternative data sets and larger input patterns.

Another aspect to evaluate is the power consumption. The wattage of a system as the one used for testing in this paper nearly doubles when using the GPU instead of the CPU for computations. Because of the enormous speedup that can be achieved the whole training process should consume less power when running on the GPU. However, further experiments are needed for an accurate evaluation.

Acknowledgment

The authors would like to thank Dr. Alfred Strey and the anonymous reviewers for their helpful comments.



(a) Execution time

(b) Speedup

Figure 7. Execution time (a) and corresponding speedup (b) of the three different implementations performing 1,000 learning iterations of a LeNet5.

Table II
PROPERTIES OF THE TESTED LENET5 VARIANTS

Input size [a]	Network properties				Execution time (in seconds)			Speedup	
	Input area	Neurons	Trainable parameters	Connections	CPU$_{\text{triv.}}$	CPU$_{\text{opt.}}$	GPU	$\frac{\text{CPU}_{\text{triv.}}}{\text{GPU}}$	$\frac{\text{CPU}_{\text{opt.}}}{\text{GPU}}$
32×32	1,024	8,010	51,046	331,114	3.6567	2.0317	0.9345	3.9132	2.1742
40×40	1,600	13,770	60,646	956,842	7.2532	3.6693	1.1103	6.5326	3.3048
48×48	2,304	21,130	79,846	2,047,210	13.0610	5.9001	1.3388	9.7554	4.4068
56×56	3,136	30,090	108,646	3,602,218	21.1530	8.7290	1.6000	13.2206	5.4556
64×64	4,096	40,650	147,046	5,621,866	32.5300	12.1500	1.9901	16.3463	6.1054
72×72	5,184	52,810	195,046	8,106,154	44.5400	16.5210	2.3532	18.9274	7.0207
80×80	6,400	66,570	252,646	11,055,082	59.4970	21.3960	2.9072	20.4650	7.3595
88×88	7,744	81,930	319,846	14,468,650	76.3710	26.5900	3.5705	21.3894	7.4471
96×96	9,216	98,890	396,646	18,346,858	95.1780	33.1480	4.1635	22.8601	7.9616
104×104	10,816	117,450	483,046	22,689,706	116.4700	40.8270	4.9055	23.7427	8.3227

[a] width×height of the input pattern

REFERENCES

[1] P. Y. Simard, D. Steinkraus, and J. C. Platt, "Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis," in *Proc. of the 7th Int. Conference on Document Analysis and Recognition*, 2003, pp. 958–962.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," in *Proc. of the IEEE*, vol. 86, no. 11, 1998, pp. 2278–2324.

[3] C. Garcia and M. Delakis, "Convolutional Face Finder: A Neural Architecture for Fast and Robust Face Detection," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 26, no. 11, pp. 1408–1423, 2004.

[4] J. C. L. Lam and M. Eizenman, "Convolutional Neural Networks for Eye Detection in Remote Gaze Estimation Systems," in *Proc. of the Int. MultiConference of Engineers and Computer Scientists*, vol. 1, 2008, pp. 601–606.

[5] Z. Zhao, S. Yang, and X. Ma, "Chinese License Plate Recognition Using a Convolutional Neural Network," in *Proc. of the Pacific-Asia Workshop on Computational Intelligence and Industrial Application*, vol. 1, 2008, pp. 27–30.

[6] R. Collobert and J. Weston, "A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning," in *Proc. of the 25th Int. Conference on Machine Learning*, vol. 307, 2008, pp. 160–167.

[7] NVIDIA, "NVIDIA CUDA – Programming Guide," http://www.nvidia.com/object/cuda_home.html, August 2009.

[8] D. Luebke and G. Humphreys, "How GPUs Work," *IEEE Computer*, vol. 40, no. 2, pp. 96–100, 2007.

[9] The Khronos Group, "OpenCL Overview," http://www.khronos.org/opencl, August 2009.

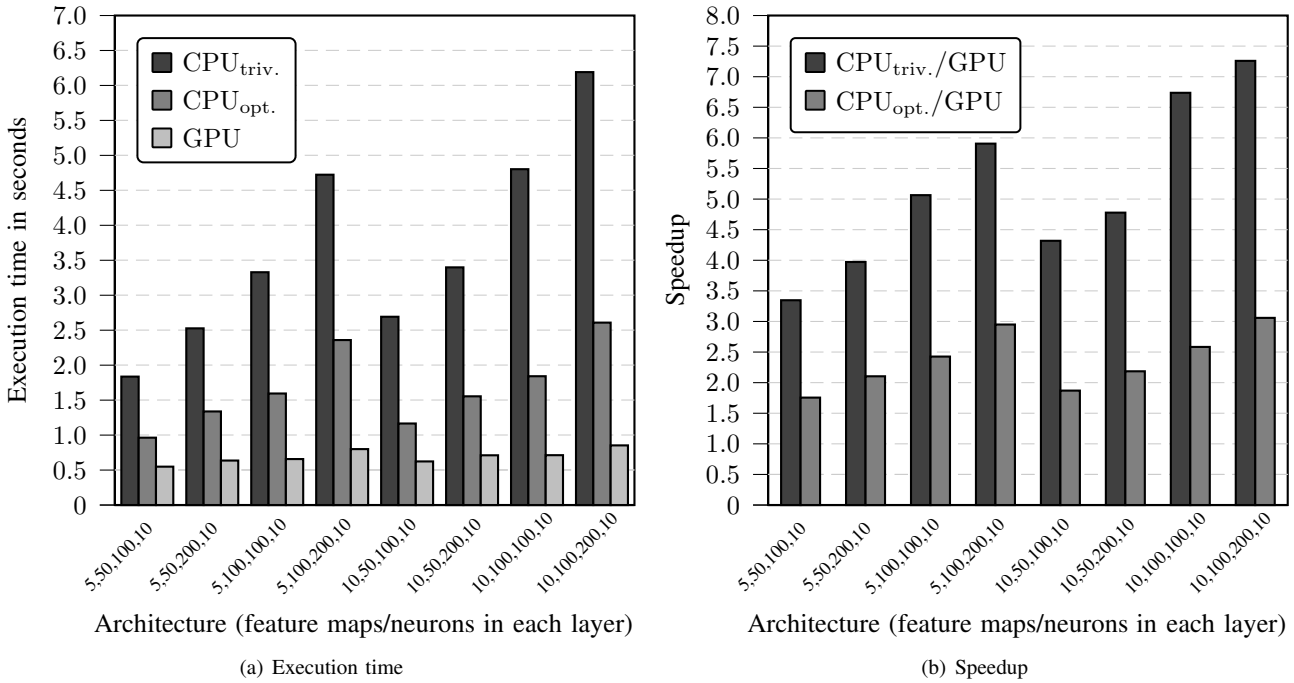
[10] C. Boyd, "DirectX 11 Compute Shader," in *The 35th Int. Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2008)*, http://s08.idav.ucdavis.edu/boyd-dx11-compute-shader.pdf.

[11] L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, *Large-Scale Kernel Machines*. MIT Press, 2007, ch. Large-Scale Parallel SVM Implementation, I. Durdanovic, E. Cosatto, and H. P. Graf, pp. 105–138.

[12] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast Support Vector Machine Training and Classification on Graphics Processors," in *Proc. of the 25th Int. Conference on Machine Learning*, vol. 307, 2008, pp. 104–111.

[13] S. Duffner, "Face Image Analysis With Convolutional Neural Networks," Ph.D. dissertation, Albert-Ludwigs-Universität Freiburg, 2007.

[14] S. L. Phung and A. Bouzerdoum, "MATLAB Library for



(a) Execution time

(b) Speedup

Figure 8. Execution time (a) and corresponding speedup (b) of the three different implementations performing 1,000 learning iterations of a SimardNet.

Table III
PROPERTIES OF THE TESTED SIMARDNET VARIANTS

Architecture [a]	Network properties				Execution time (in seconds)			Speedup	
	Input area	Neurons	Trainable parameters	Connections	CPU$_{\text{triv.}}$	CPU$_{\text{opt.}}$	GPU	$\frac{\text{CPU}_{\text{triv.}}}{\text{GPU}}$	$\frac{\text{CPU}_{\text{opt.}}}{\text{GPU}}$
5,50,100,10	841	2,205	132,540	305,580	1.8364	0.9632	0.5487	3.3470	1.7556
5,50,200,10	841	2,305	258,640	431,680	2.5266	1.3383	0.6361	3.9722	2.1040
5,100,100,10	841	3,455	263,840	588,080	3.3291	1.5949	0.6574	5.0643	2.4262
5,100,200,10	841	3,555	514,940	839,180	4.7236	2.3592	0.7998	5.9062	2.9499
10,50,100,10	841	3,050	138,920	483,800	2.6923	1.1659	0.6235	4.3184	1.8701
10,50,200,10	841	3,150	265,020	609,900	3.3987	1.5546	0.7113	4.7785	2.1857
10,100,100,10	841	4,300	276,470	922,550	4.8025	1.8419	0.7129	6.7369	2.5838
10,100,200,10	841	4,400	527,570	1,173,650	6.1908	2.6085	0.8529	7.2589	3.0586

[a] number of feature maps in layer 1, 2 and number of neurons in layer 3, 4

Convolutional Neural Network," ICT Research Institute, Visual and Audio Processing Laboratory, University of Wollongong, Tech. Rep., 2009, http://www.elec.uow.edu.au/staff/sphung.

[15] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient BackProp," in *Neural Networks: Tricks of the Trade*, G. B. Orr and K.-R. Müller, Eds., vol. 1524. Springer, 1998, pp. 9–50.

[16] NVIDIA, "NVIDIA CUDA C Programming – Best Practices Guide," http://www.nvidia.com/object/cuda_home.html, August 2009.

[17] K.-S. Oh and K. Jung, "GPU implementation of neural networks," *Pattern Recognition*, vol. 37, no. 6, pp. 1311–1314, 2004.

[18] D. Steinkraus, I. Buck, and P. Y. Simard, "Using GPUs for Machine Learning Algorithms," in *Proc. of the 8th Int. Conference on Document Analysis and Recognition*, vol. 2, 2005, pp. 1115–1120.

[19] K. Chellapilla, S. Puri, and P. Y. Simard, "High Performance Convolutional Neural Networks for Document Processing," in *Proc. of the 10th Int. Workshop on Frontiers in Handwriting Recognition*, 2006.

[20] S. Lahabar, P. Agrawal, and P. J. Narayanan, "High Performance Pattern Recognition on GPU," in *Proc. of the National Conference on Computer Vision, Pattern Recognition, Image Processing and Graphics*, 2008, pp. 154–159.

[21] G. Poli, J. H. Saito, J. F. Mari, and M. R. Zorzan, "Processing Neocognitron of Face Recognition on High Performance Environment Based on GPU with CUDA Architecture," in *Proc. of the 20th Int. Symposium on Computer Architecture and High Performance Computing*, 2008, pp. 81–88.

[22] Intel, "Intel Integrated Performance Primitives (IPP)," http://software.intel.com/en-us/intel-ipp, August 2009.

[23] Intel, "Intel Math Kernel Library (MKL)," http://software.intel.com/en-us/intel-mkl, August 2009.

[24] NVIDIA, "NVIDIA CUBLAS Library," http://www.nvidia.com/object/cuda_home.html, August 2009.

[25] D. Strigl and K. Kofler, "CNNLIB: A library for fast convolutional neural network training and classification," http://cnnlib.sourceforge.net, October 2009.

[26] C. M. Bishop, *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[27] J. V. Bouvrie, "Notes on Convolutional Neural Networks," November 2006, Unpublished, http://web.mit.edu/jvb/www/papers/cnn_tutorial.pdf.

[28] Y. LeCun and C. Cortes, "MNIST Handwritten Digit Database," http://yann.lecun.com/exdb/mnist, August 2009.

[29] A. Graves, S. Fernández, and J. Schmidhuber, "Multi-dimensional Recurrent Neural Networks," in *Proc. of the 17th Int. Conference on Artificial Neural Networks*, 2007, pp. 549–558.