A GRAPHIC PROCESSING UNIT FRAME WORK FOR CONVOLUTIONAL NEURAL NETWORK BASED CLASSIFICATION OF REMOTELY SENSED SATELLITE IMAGES

Rizwan Ahmed Ansari [1*], Winnie Thomas [2], Krishna Mohan Buddhiraju [1]

[1] Centre of Studies in Resources Engineering, Indian Institute of Technology Bombay, India - rizwan.vjti@ieee.org, bkmohan@csre.iitb.ac.in
[2] Department of Electrical Engineering, Indian Institute of Technology Bombay, India - winniethomas@ee.iitb.ac.in

Commission V, SS: Emerging Trends in Geoinformatics

KEY WORDS: GPU, GPGPU, convolutional neural networks, parallel processing, image classification

ABSTRACT:

Near real time processing and feature extraction from high-resolution satellite images aids in various applications of remote sensing including segmentation, classification and change detection. The latest generation of satellite sensors are able to capture the data at a very high spatial, spectral and temporal resolution. The processing time required for such a huge data is also large. Disaster monitoring applications such as forest fire monitoring, earthquakes require fast/real time processing of high resolution data to enable response activities. In general, due to the large size of satellite data, the computational time of feature calculation and training neural network is found to be very high. Therefore in order to achieve the aim of near real time processing of such huge data, we developed a parallel implementation. The implementation is performed on NVIDIA's Graphical Processing Unit. The performance improvement obtained is demonstrated by a GPU implementation on Resourcesat-1 data and compared with the traditional sequential implementation. The results show that the GPU implementation is found to achieve performance improvement in terms of execution time and speedup throughput as compared to the sequential implementation.

1. INTRODUCTION

Artificial Neural Networks (ANNs) perform very well on pattern recognition and classification problems with a large amount of training data. For image classification, like optical character recognition, Convolutional Neural Networks (CNNs) deliver state-of-the-art performance (Simard et al. 2003). CNNs are a variant of Multilayer Perceptron (MLP) neural networks optimized for two-dimensional pattern recognition. CNNs are used in many applications including handwriting recognition (LeCun 1998), face, eye and license plate detection (Lam and Eizenman 2008; Zhao et al. 2008), and in non-vision applications such as semantic analysis (Collobert and Weston 2008).The latest generation of satellite sensors are able to capture the data at a very high spatial, spectral and temporal resolution. The processing time required for such a huge data is also large. Disaster monitoring applications such as flood/forest fire monitoring, earthquakes require fast/real time processing of high resolution data to enable response activities. For The biggest drawback of CNNs, besides a complex implementation, is the long training time. Since CNN training is very compute- and memory-intensive, training with large data sets may take several days or weeks.

Traditionally most of the programs are written in sequential manner. A sequential program will run on only one Central Processing Unit (CPU) and will not become faster than the most powerful CPU in use today. This is a huge obstruction for an application developer because they will not be able to introduce different and new features to their software. Most of the software developers have relied on the advances in hardware to increase the speed of their applications under the hood. This trend has slowed since 2003 due to energy-consumption and heat-dissipation issues that have limited the increase of the clock frequency and the level of productive activities that can be performed in each clock period within a single CPU (Kirk and Wen-mei 2012).

However, the current CPU based approaches are not amenable for time critical applications, particularly when used on considerably large high-resolution imagery. There is an issue of scalability related to the remote sensing data. The recent emergence of Graphic Processing Units (GPU) provides a platform for such compute intensive problems and gives considerable performance improvement. Parallel programming is the only way that will give room for the performance improvement of applications in handling large data from different satellite sensors. In a parallel programming model multiple threads of execution cooperate to complete the work faster.

In general, due to the large size of satellite data, the computational time of feature calculation and training neural network is found to be very high. Hence in order to achieve the aim of near real time processing of such huge data we developed a parallel implementation. The implementation is performed on NVIDIA's Graphical Processing Unit. In particular, feature extraction and classification using neural networks have been explored in a GPU-based environment to study the significant gains achieved in their computational capability.

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 Graphic Processing Units) computation on current Graphic Processing Units (GPUs). The

[*] Corresponding author

main advantage of GPUs over CPUs is the high computational throughput at relatively low cost, achieved through their massively parallel architecture.

Using the GPU for general purpose applications requires some amount of understanding of the hardware architecture. The GPU programming platforms such as Compute Unified Device Architecture (CUDA) (NVIDIA 2009a) transforms the algorithms to be implemented into a graphics pipeline friendly format. The CUDA language bears resemblance to the C programming language and is therefore much simpler to program rather than writing the graphics API languages. Furthermore, unified shaders are better adapted to perform general computations than earlier architectures which results in a shorter training period, faster adoption and higher efficiency.

This paper describes architecture of GPU in CNN frame work for remotely sensed image classification using textural analysis; and program pieces to study and compare process times and thus speed up and throughput of CPU and GPU. The program is written in CUDA C language and has led to more insight into some typical underlying architectural behavior of the GPU device for the application of textural segmentation of remotely sensed satellite images.

2. METHODOLOGY

2.1 Data Set

The dataset is from Resourcesat-1 satellite image of Kuwait city with a spatial resolution of 5.8m x 5.8m, shown in Figure 1. This resolution is well suited for texture analysis since a spatial resolution of this order is not adequate to extract individual buildings or narrow roads but groups of them render a visible checked pattern in dense urban areas. For general analysis where multiresolution coefficients follow a mixture of distribution families, a natural way to carry out the analysis is by using lower and higher order moments of the multiresolution coefficients for the unknown underlying distributions. We have used first four moments viz. mean, variance, skewness and kurtosis of wavelet coefficient. Three level 9/7 biorthogonal wavelet is used for MRA decomposition.

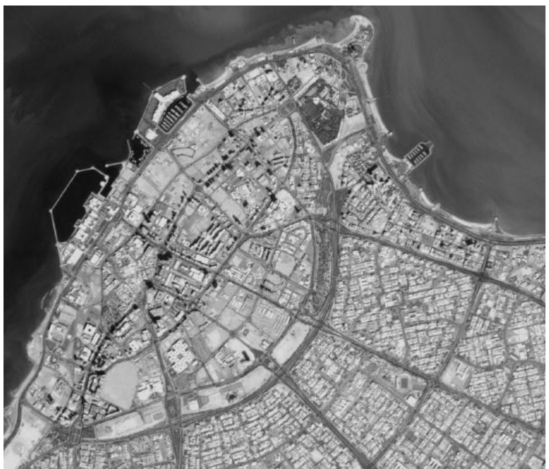


Figure 1. Original image covering Kuwait city (Resourcesat image)

2.2 Feature Extraction

Multiresolution analysis (MRA) has been successfully used in texture analysis of images. Texture is characterized by the spatial organization of gray level variations in a local area. It quantifies the local intensity variations in an image, observing properties such as fineness, coarseness and evenness. Co-occurrence matrices are frequently used in texture analysis as they capture the spatial relatedness of pixel intensities in a neighbourhood within an image (Unser, 1986; Unser 1995; Murray et al., 2010). These methods are constrained by the analysis of spatial arrangement over relatively small neighbourhoods on a given single scale. An object which is smaller than the spatial resolution of sensor system cannot be identified. As a result, performance of co-occurrence matrices is only suitable for micro-level textures (Unser, 1995). A multiresolution technique provides a coarse-to-fine and scale-invariant decomposition for interpreting the image information. At different scales, the details of an image vary according to the content of the image where, the lower resolution provides a global view, while the higher resolution provides the finer details of the scene. Texture and MRA are therefore required in analysis and segmentation because it is difficult to analyze the information content of an image directly from the pixel intensity. The local changes of the intensity of an image are more important than the gray level intensity of that image. Textured objects reveal different type of information as a function of the resolution of reference for analysis, which cannot optimally be observed at a single resolution for image analysis. Therefore, it is better to extract features of the objects from different segmentation levels.

The wavelet transform, is extensively used to describe images in multiple resolutions. Wavelets can be viewed as a projection of the signal on a specific set of scaling *$\phi(t)$* and wavelet basis *$\psi(t)$* functions in the vector space. The wavelet coefficients obtained represent these projection values. The discrete wavelet transform is realized with the help of filter banks. The basis functions are expressed with the help of dilation equations as

$$\phi(t) = \sum_{n\in\mathbb{Z}} h[n]\phi(2^m t - n) \tag{1}$$

$$\psi(t) = \sum_{n\in\mathbb{Z}} g[n]\phi(2^m t - n) \tag{2}$$

Where *h[.]* are low-pass filter coefficients, *g[.]* are high-pass filter coefficients of the filter bank, m is the scaling index, and n is the translating index.

By decomposing the image into a series of high-pass and low-pass filter bands, the wavelet transform extracts directional details that capture horizontal, vertical, and diagonal details. We have used moment and energy based texture features from wavelet coefficients as features to in the neural networks for classification task.

2.3 Graphic Processing Unit

The demand for very high quality real time graphics in computer applications has been the inspiration behind the advancement of graphics hardware. A graphic programmer writes a single thread program that draws one pixel and GPU runs multiple copies of this program (thread) in parallel, drawing multiple pixels in parallel. Now graphic programs written in C or C++ with the CUDA model, scale transparently (Boyd 2008). Scalability has enabled GPUs to rapidly increase their parallelism and performance with increasing transistor density as GPU transistor counts are increasing exponentially doubling every eight months (LeCun 1998).

In 2001, NVIDIA GeForce introduced General shader programmability there by allowing the application developer to work with instruction of the floating point vertex engine. These programmability and floating point capability, extended to the pixel shader stage and made texture accessible from the vertex shader stage, e.g. ATI Radeon (2002), featured a programmable 24-bit floating point pixel shader processor programmed with DirectX9 and OpenGL. GeForce features 32-bit floating point pixel processors. GeForce 6800 (Kirk and Wen-mei 2012) and 7800 series introduced separate dedicated vertex and pixel processors. In 2005, Xbox 360 GPU achieved unified processing by allowing vertex and pixel shader to execute on the same processor.

GeForce 8800 GPU introduced in 2006, featured an array of unified processors. The unified processor supports dynamic partitioning of the array of processors to vertex shading stage, geometry processing (first introduced in GeForce 8800 GPU) and pixel processing stage.

2.4 General Purpose Computing on GPU

The GPUs were only capable to process graphic data. GPGPU allows the utilization of a GPU to perform computation in applications traditionally handled by the CPU. To access the computational resources the programmer had to use OpenGL or DirectX API calls (McReynolds 1998). The NVIDIA Tesla GPU architecture designers replace shader processors with fully programmable processors with instruction memory, cache and instruction sequencing control (Lindholm et al. 2008, Huang 2009). With NVIDIA developing CUDA C/C++ compiler libraries also, by now programmers can easily access the GPU. NVIDIA introduced Fermi GPU computing architecture in 2009 (NVIDIA 2009a). It increased double- precision performance, error correcting code (ECC) memory protection for large scale computing, 64 bit unified addressing, cached memory hierarchy and instruction for C, C++, Fortran, OpenCL and DirectCompute .

In 2010 September, NVIDIA introduced Kepler architecture, which added new features of dynamic parallelism and Hyper Q (NVIDIA 2013). Dynamic parallelism allows GPU to generate work for itself and to schedule that work through the best hardware path, without involving the CPUs. Hyper Q allows multiple CPU cores to call single GPU thereby dramatically increasing GPU utilization and significantly reducing CPU idle times (NVIDIA 2013).

2.5 Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) is software and hardware architecture for supporting heterogeneous parallel computing. It enabled the GPU to be programmed with a variety of high level languages. The programmer could now write C programs with CUDA extensions and target a general purpose, massively parallel processor.

CUDA 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. To a CUDA programmer, the computing system comprises a host which is a traditional CPU, such as an Intel architecture microprocessor in personal computers and GPU(s) which are massively parallel processors with large number of arithmetic execution units. The program section often consist of some data parallelism, that allow many arithmetic operations to be safely performed on streaming data in a simultaneous manner. Streaming data can be considered as a stream of data elements that are required to be processed by same task or instruction based on Single Program Multiple Data (SPMD) which is a data parallel model. The CUDA devices accelerate the execution of these applications by harvesting a large amount of data parallelism. Different texture classes from remotely sensed images at different resolution capture independent and simultaneous events.

The kernel functions typically generate a large number of threads to exploit data parallelism. These CUDA threads are of lighter weight than the CPU threads. These threads take few cycles to generate and schedule which is in contrast with CPU threads that typically take thousands of clock cycles to generate and schedule. The program execution always begins with host (CPU) execution. The execution is moved from host to device (GPU) when a kernel function is called. All the threads that are generated due to the launch of kernel are called a grid (Kirk and Wen-mei 2012). When all the threads complete their execution, the grid formed by threads also ends for that kernel. The remaining non kernel part of the program is executed on host till the next kernel is called by host.

2.6 CUDA Thread Hierarchy

The threads on CUDA are organized into a hierarchy of threads, thread blocks and grid of blocks as shown in Figure 2. Once a kernel is called or launched, the grid corresponding to the threads is generated. To assign the threads to execution resources, they are divided into blocks. In Fermi architecture the execution resources are in the form of streaming multiprocessors abbreviated as SMs. An SM consists of 8 or more streaming processors (SPs) also called as cores. For e.g., the NVIDIA GT630M GPU used for experimentation has two SMs shown in Figure 3. Each SM consists of 48 cores and 1 core processes single block. Hence both SMs can service 96 blocks at a time in GT630M as long as sufficient resources are available for all the thread blocks. If the available resources do not suffice to the need of all 48 blocks per SM, CUDA runtime system automatically reduces the number of blocks per SM. The runtime system keeps the record of the blocks that are needed to be executed and as soon as the previous blocks are serviced, the new blocks are assigned or mapped to SMs.

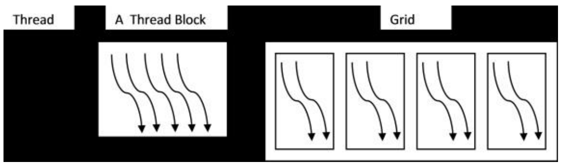


Figure 2. CUDA thread hierarchy

In Fermi architecture, once a block is assigned to an SM, it is further divided into 32 threads units called warps that mean the execution of a thread block is divided into warp execution. The core in the SM services a block in warp by warp basis.

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. Figure 2 shows how threads are grouped and mapped to the hardware in CUDA. At start of a function on the GPU (also called as a 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 65,535x65, 535 blocks in this case (GTM630). 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 512x512x64, 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 as shown in Figure 4. Using the most appropriate memory in the right way (e.g. coalesced access to global memory, avoiding bank conflicts in shared memory) is one of the most effective means of improving performance (NVIDIA 2009b).

Because of the many core 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 1. 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 1010.8GFLOPS/s promoted by NVIDIA can only be achieved if every SM can execute 8 multiplications and 8 MAD operations at the same time.

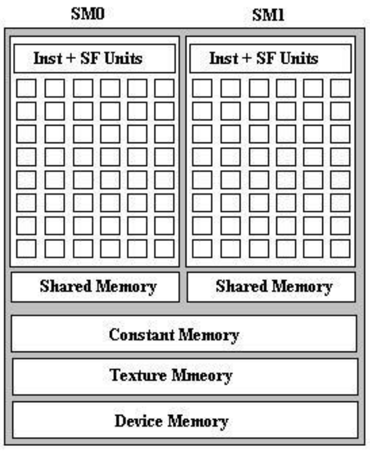


Figure 3. SM in GT630M

Table 1. Technical Specification

	Intel Core i5-3210	GeForce GTX 275
Processor core clock	2.5 GHz	800 MHz
Memory size	32GB/DDR3	2048MB/GDDR5
Bandwidth core: memory	25.3 GB/s	32.0 GB/s
Number of cores	2	96
Number of threads	4	Upto 512
SP FLOPS / core and clock cycle	4 MUL or ADD	8 MUL and 8 MAD
Total SP FLOPS peak performance	1.61GFLOPS/s	307.2 GFLOPS/s

2.7 Convolutional Neural Networks

The general method for two-dimensional pattern recognition task is based on a feature extractor, the output of which is fed into a neural network. This feature extractor is usually static and independent of the neural network. It is difficult to find a suitable feature extractor because it is not part of the training procedure and therefore it can neither adapt to the network used nor to the parameters of the training procedure.

2.7.1 Image processing layer: The image processing layer is an optional pre-processing layer of predefined filters that are kept fixed during training. Thus additional information besides the raw input image can be provided to the network, such as boundaries and gradients. CNNs make this difficult task part of the network and act as a trainable feature extractor with some degree of shift, scale, and rotation invariance (Gonalez 2007). 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. The convolutional layers are responsible for the feature extraction (edges, corners, end points or non-visual features in other signals), using the two key concepts of local receptive fields and shared weights. The fully connected layer acts as a normal classifier similar to the layers in traditional MLP networks (Figure 4). A brief explanation of the composition and the mathematical model of these layers are described in the following sections.

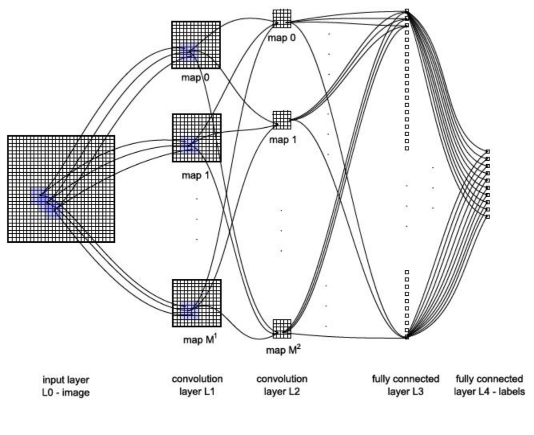


Figure 4. Structure of CNN (Ciresan et al. 2011)

2.7.2 Convolutional Layer: The convolutional layers are the core of any CNN. A convolutional layer consists of several two-dimensional planes of neurons which serve as feature maps. Each neuron of a feature map is connected to a small subset of neurons inside the feature maps of the previous layer, and works as receptive fields. The receptive fields of neighbouring neurons overlap and the weights of these receptive fields are shared through all the neurons of the same feature map. A convolutional layer is parametrized by the size and the number of the maps, kernel sizes, skipping factors, and the connection table. Each layer has M maps of equal size (Mx, My). A kernel of size (Kx, Ky) is shifted over the valid region of the input image (i.e. the kernel has to be completely inside the image). The skipping factors Sx and Sy define how many pixels the filter/kernel skips in x- and y-direction between subsequent convolutions. The size of the output map is then defined as in (Ciresan et al. 2011)

$$M_x^n = \frac{M_x^{n-1} - K_x^n}{S_x^n + 1} + 1; \qquad M_y^n = \frac{M_y^{n-1} - K_y^n}{S_y^n + 1} + 1 \tag{3}$$

where index *n* indicates the layer. Each map in layer L^n is connected to at most M^{n-1}maps inlayer L^{n-1}. Neurons of a given map share their weights but have different receptive fields.

The feature maps of a convolutional layer and its preceding layer are either fully or partially connected. 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 a nonlinear activation function viz. sigmoidal, hyperbolic tangent.

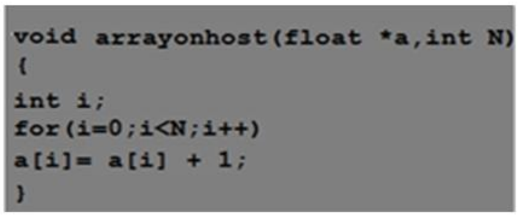
2.7.3 Classification layer: Kernel sizes of convolutional filters and max-pooling rectangles as well as skipping factors are chosen such that either the output maps of the last convolutional layer are down-sampled to 1 pixel per map, or a fully connected layer combines the outputs of the topmost convolutional layer into a 1D feature vector. The top layer is always fully connected, with one output unit per class label.

2.8 GPU Implementation

We implemented a high performance but still flexible library in CUDA to accelerate the training and classification process of arbitrary CNNs. Due to the fact that the ideal parameters of a neural network 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 (CPUtriv). 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) (Intel 2009a) and MKL (Math Kernel Library, ver. 10.2) (Intel 2009b) (CPUopt).Those libraries take the full advantage of the newest Streaming SIMD Extensions (SSE) of the CPU. These enhancements resulted in a quite fast implementation.

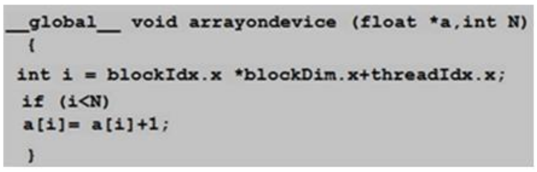
The GPU implementation (GPU) using CUDA exchanges the mathematical vector and matrix operations with functions either from NVIDIA’s CUBLAS Library (NVIDIA 2009c) if appropriate functions are available there or our own implementations otherwise. Each kernel-function performs one mathematical operation, e.g. a matrix-vector multiplication or the summation of all elements in a vector.

Figure 5 shows the implementation of a routine to increment each element of an array of certain length. Figure 5a shows sequential and Figure 5b shows parallel implementation of the routine to be executed on the CPU and CUDA device respectively. The host used for experimentation is 3rd generation i5-3210 processor, with 2 cores and with the memory of 4 GB RAM. The CUDA device used is Fermi Architecture Based NVIDIA GT630M and has dedicated memory of 2GB RAM. The overall architecture is a discrete heterogeneous architecture ,where in CPU and GPU or any other processor are connected through PCI bus on different chips each with their own global memory. All the cores of the CPU while running task were active so that the performance comparison of GPU and CPU is fair as much as possible. CPU supports multithreading with 4 threads and each core operates at 2.5 GHz. GPU operates at 0.95 GHz.



```
void arrayonhost(float *a,int N)
{
int i;
for(i=0;i<N;i++)
a[i]= a[i] + 1;
}
```

(a) Host Routine for CPU



```
__global__ void arrayondevice (float *a,int N)
 {
int i = blockIdx.x *blockDim.x+threadIdx.x;
 if (i<N)
 a[i]= a[i]+1;
 }
```

(b) Kernel for the GPU

Figure 5. Host routine for CPU and Kernel of GPU

The objective of this routine is to measure the process time taken by the CPU and the GPU and to compare their execution time and throughput by recording the speed up. Throughput of a device is the number of tasks it performs in unit time. The elements of array are generated in host with a simple *for loop*. The length of the array is varied. The kernel for GPU is called by the host (CPU). Block size is also varied to analyze the impact of number of threads per block on performance and ability of the CUDA GPU.

2.8.1 Forward Propagation (FP): A straight forward way of parallelizing FP is to assign a thread block to each map that has to be calculated. For maps bigger than 512 neurons, the task is further split into smaller blocks by assigning a block to each line of the map, because the number of threads per block is limited (512 for GT630M). A one to one correspondence between threads and the map's neurons is assumed. Due to weight sharing, threads inside a block can access data in parallel, in particular the same weights and inputs from the previous layer. Each thread computes by initializing its sum with the bias, then loops over all map connections, convolving the appropriate patch of the input map with the corresponding kernel. The output is obtained by passing the sum through a scaled nonlinear activation function, and then written to device memory.

2.8.2 Backward Propagation (BP): The thread grid assigns a thread block to each map in the previous layer and a thread to each neuron in every map. Similar to FP, for maps with more than 1024 neurons, the 2D grid is further split into smaller 1D blocks by assigning a 2D block to each row of the map. Each thread computes the delta of its corresponding neuron by pulling deltas from the current layer. For every neuron in the previous layer we have to determine the list of neurons in the current layer which are connected to it. Let us consider neuron (i, j) from a map in layer L^{n-1}, and then assume that (x, y) are the coordinates of neurons in maps of L^n that contribute to the delta of neuron (I, j). The (x, y) neuron is connected to kernel size number neurons (K_x x K_y) from each connected map in the previous layer. The indices in $L^{n-1}$of the neurons connected through a kernel to the (x, y) neuron are:

$$\begin{aligned} x(S_x+1) \le i \le x(S_x+1)+K_x-1 \\ y(S_y+1) \le j \le y(S_y+1)+K_y-1 \end{aligned} \quad (4)$$

The inequalities are computed as

$$\begin{aligned} \frac{i-K_x+1}{S_x+1} \le x \le \frac{i}{S_x+1} \\ \frac{j-K_y+1}{S_y+1} \le y \le \frac{j}{S_y+1} \end{aligned} \quad (5)$$

With final inequalities computed as (x, y) is inside the map (Ciresan 2011);

$$\begin{aligned} \max\left(\left\lceil \frac{i-K_x+1}{S_x+1} \right\rceil, 0\right) \le x \le \min\left(\left\lfloor \frac{i}{S_x+1} \right\rfloor, M_x-1\right) \\ \max\left(\left\lceil \frac{j-K_y+1}{S_y+1} \right\rceil, 0\right) \le y \le \min\left(\left\lfloor \frac{j}{S_y+1} \right\rfloor, M_y-1\right) \end{aligned} \quad (6)$$

These inequalities suggest that the delta of neuron (i, j) from L^{n-1} is computed from deltas of neurons in a rectangular area in maps of L^n. After summing up the deltas, each thread multiplies the result by the derivative of the activation function.

2.8.3 Adjusting weights: FP and BP have a grid on the list of maps, but the thread grid is on the list of kernels or filters between maps of two consecutive layers. The 1D grid has a block for each connection between two maps. Thread blocks are 2D, with a corresponding thread for every kernel weight. The bias weights included as an entire row of threads, thus requiring thread blocks to have (K_x+1) times K_y threads. Most of the time these additional Ky threads will do nothing, thread (0, 0) being activated only for blocks that have to process the bias.

3. RESULTS AND DISCUSSION

All benchmarks in this paper were performed in single precision on an Intel Core i5-3210 with a GeForce GT630M running Windows 7. The technical specifications of these two processors are shown in Table I. All benchmarks consisted of 1,000 training iterations of the networks as described above. Such a training iteration is composed of one forward propagation (a training pattern is fed into the network and produces some output), one back-propagation (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 back-propagation 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 is used.

For our performance and scalability tests we performed measurements on two different networks: Simard et al. (2003) and LeCun (1998). In all benchmarks we compared the three different implementations explained in the previous section (CPUtriv, CPUopt, GPU).

3.1 Tested Networks

We have used the network proposed by Simard et al. (2003) which provides a fast and simple CNN implementation for vision applications. It consists of two convolutional and two fully connected layers. The convolutional layers use a step size of two, which makes subsampling layers superfluous. Another variant the LeNet5 (LeCun 1998) is used. It is composed of three convolutional, two subsampling and two fully connected layers.

3.1.1 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.

Figure 6 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 versions not only the fastest one but it also scales best with the input size. This is underlined by Figure 7 which shows the speedup of the GPU version in comparison to the trivial CPU version (CPUtriv:=GPU) and to the optimized CPU version (CPUopt:=GPU). The speedup grows with the input size and the GPU version definitely scales better than the CPU versions with large input sizes.

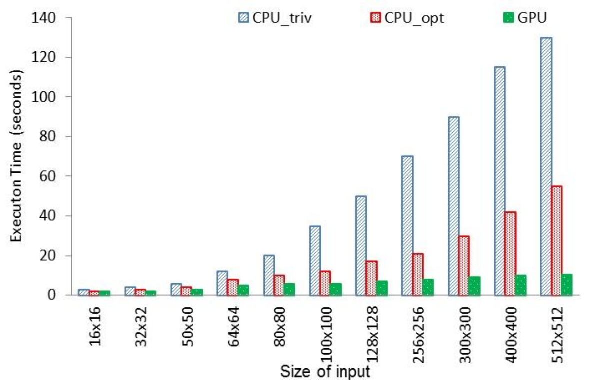


Figure 6. Execution time

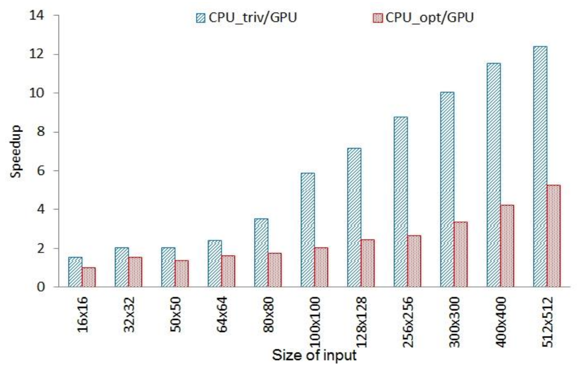


Figure 7. Speedup performance

3.2 Segmentation Result

Figures 8 and 9 show the reference window and segmented image respectively. Exact re-occurring patterns of built-up areas and roads are very different from water bodies and open areas, and therefore this is a good candidate image for texture analysis. Here, five classes are considered; water, shallow water, built-up area, open area and roads. The reference image is used to evaluate the classification result in terms of the kappa coefficient for the dataset considered in this study. Class-wise accuracy in terms of confusion matrix and kappa coefficient are computed (Table 2) to demonstrate the ability of wavelet based features for segmentation.

Table 2. Classification Accuracy

	User's Accuracy (%)	Producer's Accuracy (%)	Error of commission	Error Omission
Water	96.4	98.3	0.0353	0.0161
Shallow Water	969	93.9	0.0309	0.0605
Built-up area	82.8	81.9	0.1711	0.1803
Open	909	91.8	0.0903	0.0815
Road	85.0	86.2	0.1497	0.1379
Overall accuracy	90.5		Kappa	0.88

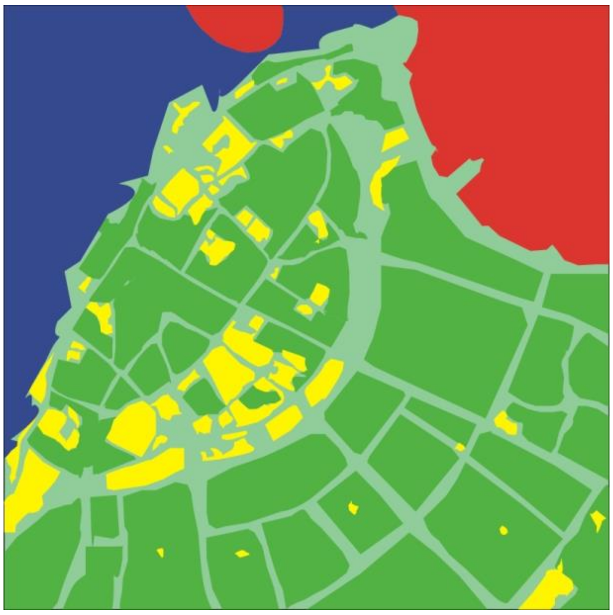


Figure 8.Reference Image

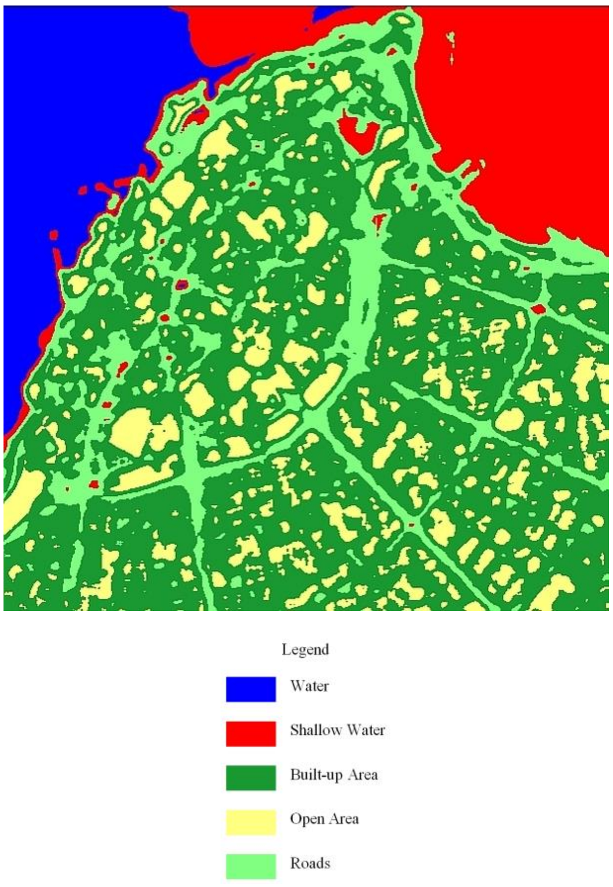


Figure 9. Segmented image

4. CONCLUSION

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 large matrices that have to be handled inside the network seem to be appropriate for GPGPU processing. Furthermore, our experiments demonstrated that the GPU implementation scales much better than the CPU implementations with increasing input size. One single test run (training and evaluation) can take hours 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 consumes less power when running on the GPU. However, further experiments are needed for an accurate evaluation.

REFERENCES

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

Cireşan, D.C., Meier, U., Masci, J., Gambardella, L.M. and Schmidhuber, J., 2011. High-performance neural networks for visual object classification. *arXiv preprint arXiv:1102.0183*.

E. Lindholm et al., 2008, NVIDIA Tesla: A Unified Graphics and Computing Architecture, IEEE Micro, vol. 28, no. 2, pp. 39-55.

Gonzalez, Rafael C and Woods, Richard E., 2007, Digital Image Processing, 3ed, Prentice Hall, India.

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

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

J. C. L. Lam and M. Eizenman, 2008, 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.

J.H. Huang, 2009: The GPU Computing Tipping Point, Proc. IEEE Hot Chips 21, http://www.hotchips.org/archives/hc21

Kirk, David B., and Wen-mei W. Hwu, 2012, Programming Massively Parallel Processors: A Hands-on Approach, Amsterdam: Elsevier/Morgan Kaufmann.

McReynolds, Tom and Blythe, David and Grantham, Brad and Nelson, Scott, 1998, Advanced Graphics Programming Techniques Using OpenGL." *Siggraph 1998 Course Notes*. Citeseer.

Murray, H., Lucieer, A. and Williams, R., 2010. Texture-based classification of sub-Antarctic vegetation communities on Heard Island. International Journal of Applied Earth Observation and Geoinformation, 12(3), pp.138-149.

NVIDIA 2013, NVIDIA's Next Generation CUDATM Compute Architecture: Kepler TM K110, http:// www.nvidia.com/ content/ PDF/kepler/ NVIDIA-Kepler-GK110- Architecture-Whitepaper.pdf Web. 07 July. 2013.

NVIDIA, 2009a, Unveils next Generation CUDA GPU Architecture--codenamed 'Fermi'. In Advanced Imaging, Oct. 2 2009.

NVIDIA, 2009b, NVIDIA CUDA – Programming Guide," http: //www.nvidia.com/object/cuda home.html, August 2009b.

NVIDIA, 2009c, NVIDIA CUBLAS Library," http://www.nvidia. com/object/cuda home.html, August 2009c.

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

R. Collobert and J. Weston, 2008, 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, pp. 160–167.

Unser M., 1995. Texture classification and segmentation using wavelet frames, IEEE Trans. Image Process. 4, 1549–1560.

Unser, M., 1986. Local linear transforms for texture measurements. Signal Process. 11, 61–79.

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

Z. Zhao, S. Yang, and X. Ma, 2008, Chinese License Plate Recognition Using a Convolutional Neural Network," in Proc. of the Pacific-Asia Workshop on Computational Intelligence, Vol. 1, pp. 27-30.