Neural Network Implementation using CUDA and OpenMP

Honghoon Jang, Anjin Park, Keechul Jung
Department of Digital Media, College of Information Science, Soongsil University
{rollco82,Anjin,kcjung}@ssu.ac.kr

Abstract

Many algorithms for image processing and pattern recognition have recently been implemented on GPU (graphic processing unit) for faster computational times. However, the implementation using GPU encounters two problems. First, the programmer should master the fundamentals of the graphics shading languages that require the prior knowledge on computer graphics. Second, in a job which needs much cooperation between CPU and GPU, which is usual in image processings and pattern recognitions contrary to the graphics area, CPU should generate raw feature data for GPU processing as much as possible to effectively utilize GPU performance. This paper proposes more quick and efficient implementation of neural networks on both GPU and multi-core CPU. We use CUDA (compute unified device architecture) that can be easily programmed due to its simple C language-like style instead of GPGPU to solve the first problem. Moreover, OpenMP (Open Multi-Processing) is used to concurrently process multiple data with single instruction on multi-core CPU, which results in effectively utilizing the memories of GPU. In the experiments, we implemented neural networks-based text detection system using the proposed architecture, and the computational times showed about 15 times faster than implementation using CPU and about 4 times faster than implementation on only GPU without OpenMP.

1. Introduction

GPUs (graphic processing units) are much more effective in utilizing parallelism and pipelining than general purpose CPUs, as they are designed for high-performance rendering where repeated operations are common. In result, the GPUs have recently attracted a lot of attention in the field of computer vision and image processing with many repeated operations. For example, since there are many repeated operations in implementing NNs (neural networks), it can be quickly and effectively performed in GPU. Moreover, GPUs have recently became increasingly competitive as regards speed, programmability, and price. Therefore, many algorithms used in the fields of computer vision and image processing are translated into implementation on GPU [1-5].

Moreland and Angel [2] implemented FFT (fast Fourier transform) on GPU, and performed the FFT by executing a fragment program on every pixel at each step in a SIMD (single instruction, multiple data)-like fashion. Mairal et al. [3] implemented stereo matching algorithm on GPU. Geys and Gool [4] implemented view synthesis, and the efficiency was accomplished by the parallel use of the CPU and the GPU. The input images were projected on a plane sweeping through 3D space, using the hardware accelerated transformations available on the GPU and a max-flow algorithm on a graph was implemented on the CPU to ameliorate the result by a global optimization. Yang and Welch [5] implemented image segmentation and smoothing that is basic arithmetic of computer vision on GPU taking advantage of register combiner and blending technology. Moreover, Oh and Jung [1] implemented neural networks on GPU, which is one of popular algorithm of pattern recognition algorithm, and the GPU was used to implement the matrix multiplication of a neural network to enhance the time performance.

Above mentioned papers [1-5] showed faster computational performances compared with the implementations on CPU. However, implementation on GPU encounters two main problems.

First, the programmer should master the fundamentals of graphics shading languages that require the prior knowledge on computer graphics, as the implementation should be programmed using the shading languages, such as HLSL included in Direct X [6], Cg included in nVIDIA [7], and GLSL included in OpenGL 2.0 [8]. Although languages that can program on GPU more easily, such as Brook [9], are recently announced, these languages showed a slower execution time than previous shading languages [6-8]. Moreover, the shading languages on the GPU cannot easily access

978-0-7695-3456-5/08 $25.00 © 2008 IEEE
DOI 10.1109/DICTA.2008.82

general memories involved in GPU, as they should access through only texture memory [10].

Second, it is essential to avoid data transfer between the CPU and GPU as much as possible to take advantage of efficiency of GPU. Almost all applications or algorithms used in computer vision and image processing, which involve a high capacity of data, cannot be completed in one step on GPU due to the limited memory of the GPU. Due to this reason, Fung and Mann [11] explored the creation of a parallel computer architecture consisting of multiple GPU built entirely from commodity hardware to simultaneously process all the data on multiple memories of GPUs. However, the architecture is not general, as almost all computers have only one graphics hardware. As an another approach, the algorithms implemented on CPU and GPU can be executed in parallel by using a multi-threaded implementation, which means the next operation can be processed while the previous one is still processed [4]. However, it also has a significant problem that the computational time on GPU is much faster than on CPU, and thus the GPU waits for completing the process on CPU. Therefore, it is essential to transfer data as much as possible from CPU to GPU to take advantage of efficiency of GPU. However, in this case, many overheads should be occurred when the CPU generates the data as much as possible.

This paper proposes more quick and efficient implementation on both commodity graphics hardware and multi-core CPU. We use a new GPU language CUDA (compute unified device architecture) recently released from NVIDIA, as the CUDA code is C language style and has less computational restriction, while the traditional GPGPU could be programmed through only a graphics API that requires much special knowledge on computer graphics. Moreover, we design the NN using inner product operation in parallel to be suitable to the CUDA. To reduce the computational time on CPU, which generates data as much as possible that will be performed in GPU, we implement feature extraction module for the NNs using OpenMP (Open Multi-Processing), which can help to concurrently process multiple data with single instruction on multi-core CPU while processing only one data on GPU. Therefore, the proposed method minimizes differentiation between two computational times on only one graphics hardware.

Based on the proposed architecture, we implement a NN, involving the main problem that is the computational complexity in the testing stage. Fig. 1 shows an overall flow chart. Given input image, the function GetConfiguration() extracts features. Here, feature extraction is processed on the multi-core CPU, which is performed in parallel to reduce the computational time on CPU, and the set of extracted features is transferred to the CUDA. CUDA performs main operations of NN composed of inner-product operations and an active function, and we design two operations using multi-thread and shared memories to

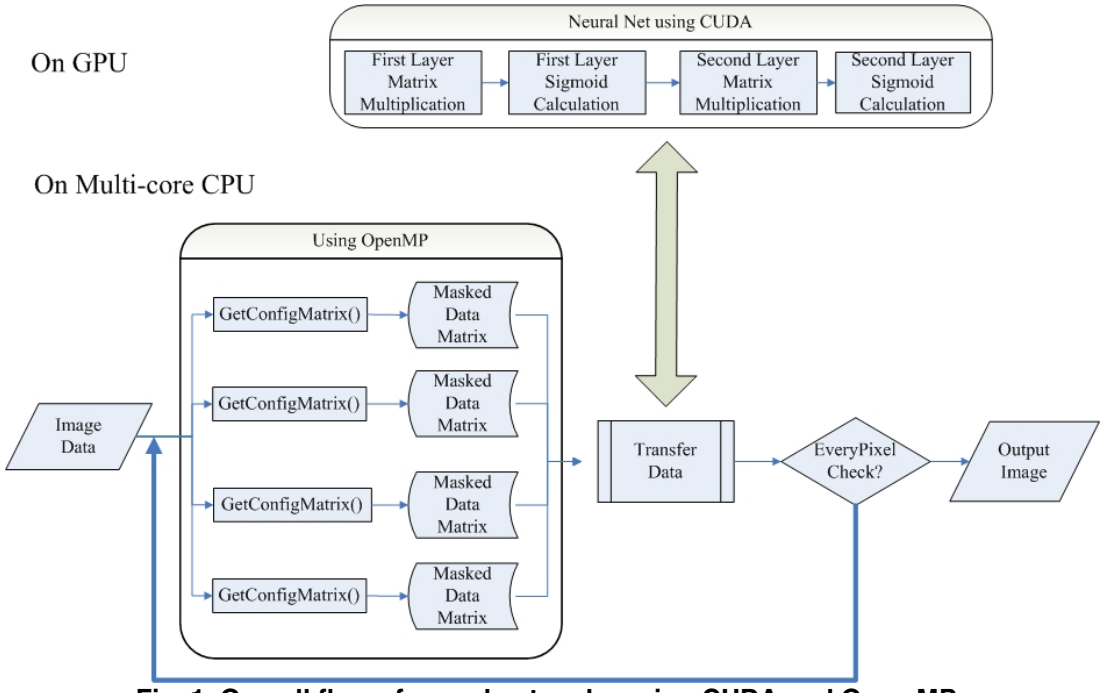


Fig. 1. Overall flow of neural networks using CUDA and Open MP.

be suitable to the CUDA. In the experiments, we implemented NN-based text detection using the proposed architecture, and the computational times showed about 20 times faster than implementation using CPU and about 5 times faster than on only GPU.

The remainder of this paper is organized as follows. The brief introductions of CUDA and OpenMP are described in section 2, and implementation of NN on the proposed architecture is described in section 3. Some experimental results are presented in section 4, and the final conclusions are given in section 5.

2. Proposed Architecture

The proposed architecture mainly consists of CUDA that allows us to program an algorithm executed on GPU in a C programming language style and OpenMP that concurrently processes multiple data with single instruction. Therefore, brief introductions of CUDA and OpenMP are described in section 2.1 and 2.2, respectively.

2.1. CUDA

The mechanism of general computation using a GPU is as follows. The input data is transferred to the GPU as *textures* or *vertex* values. The computation is then performed by the *vertex shader* and *pixel shader* during a number of rendering passes. The vertex shader performs a routine for every vertex that involves computing its position, color, and texture coordinates, while the pixel shader is performed for every pixel covered by polygons and outputs the color of the pixel. The reason why the programmer used texture or vertex values and vertex or pixel shader is the GPU could only be programmed through a graphics API, imposing complex knowledge on computer graphics and the overhead of an inadequate API to the non-graphics application.

A CUDA is a new GPU programming language recently released from NVIDA [10]. The CUDA code is written in the standard C language with some extensions related to GPU computation, and thus it can help to easily program the general computation on GPU if a programmer has basic knowledge on the standard C language. Moreover, since the CUDA do not use the graphics API generally, the overhead for the non-graphics application, such as basic operations, should also be reduced.

Moreover, GPU programs can gather data elements from any part of DRAM, but could not be written in a general way, which means GPU programs cannot scatter information to any part of DRAM. It results in removing a lot of the programming flexibility readily available on the CPU[10].

The CUDA provides general DRAM memory addressing for more programming flexibility: both scatter and gather memory operations. From a programming perspective, this translates into the ability to read and write data at any location in DRAM, like on a CPU. CUDA features a parallel data cache or on-chip shared memories with very fast general read and write access, that threads use to share data with each other. Thus, applications can take advantage of it by minimizing overfetch and round-trips to DRAM.

We explain basic terminologies to easily understand the CUDA. As a *thread* is a basic execution unit, many threads on GPU are created, and they execute a same function in parallel. A *thread block* is a batch of threads that can cooperate together by efficiently sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses.

However, CUDA cannot share two memories of CPU and GPU, which means GPU receives input data from the CPU to implement operations. To take advantage of efficiency of GPU, it is essential to avoid data transmission between the CPU and GPU as much as possible, and to do this, CPU should generate data as much as possible. However, in this case, many computation times to make maximum data in CPU are required, and thus hinder the effective use of CPU and GPU architecture. We solve this problem by using OpenMP that performs the operations in parallel implementation of CPU, which will be described in section 2.2.

2.2. OpenMP

The OpenMP is a set of directives for C, C++, and Fortan programs that make it easier to express shared-memory parallelism, which was released in 2005 [12]. The advent of commodity inexpensive multi-core processors and corresponding OpenMP-capable complier has recently increased the popularity of OpenMP. The OpenMP consists of two teams: *Master* and *Slave*, and an implementation of multithreading whereby the master “thread” forks a specified number of slave “threads” and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors. Fig. 2 shows an illustration of multithreading where the master thread forks off a number of threads that execute blocks of code in parallel.

The OpenMP indicates how to process the code block by compiler indicators. The most basic indicator is ‘#pragma omp parallel’ to indicate parallel regions. The OpenMP uses a fork-join model as a parallel

operation model. The fork-join model starts with initial single thread, and then two procedures are iteratively performed; if the parallel regions are reached, additional thread is constructed, and then operations are performed on the thread, and if the parallel regions are closed, the constructed thread are destroyed. Moreover, the OpenMP provides several useful routines for the thread: information of activated threads, setting the number of threads to be used for parallelization, and the number of maximum threads. Above mentioned functionalities are collectively defined in the specification of the OpenMP API. This specification provides a model for parallel programming that is portable across shared memory architectures from different vendors. Compliers from numerous vendors support the OpenMP API. More information about OpenMP can be found at the following web site: http://www.openmp.org/

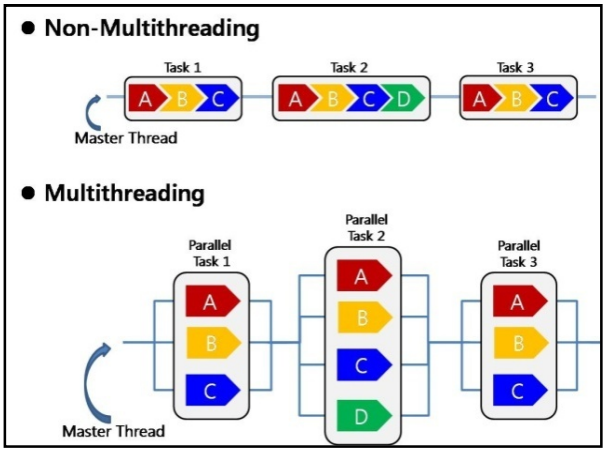


Fig. 2. Architecture of OpenMP.

In the experiments, when using only CUDA without OpenMP, computational times that gather data to be transferred to the GPU take 4/5 of total computational times. Therefore, the gathering times need to be reduced as many as possible. The solution to do this is the OpenMP, as it can help to process multiple data with same operations in parallel, and thus it can reduce the overhead for gathering times. In the case of NN, one operation to extract features is concurrently run with multiple input data using OpenMP, and then extracted features are simultaneously sent to the CUDA.

3. NN Implementation

NN is based on the concept of the workings of the human brain. There are many different types of NN, with the more popular being a multilayer perceptron (MLP), learning vector quantization, radial basis function, Hopfiled, and self-organizing map.

The current study focuses on implementing the test stage of the MLP using a CUDA and OpenMP. The MLP consists of one input layer, one output layer, and one or more than hidden layer. Nodes of adjacent layers are usually fully connected, and the mechanism of general computation for adjacent two layers in the testing stage consists of two steps: 1) inner-product operation between weights and input vectors of each layer (Eq. 1) and 2) then activate function (Eq. 2).

$$m_j = \sum w_{ij} x_i + b_j \quad (1)$$

$$r_j = (1 + e^{-m_j})^{-1} \quad (2)$$

In the Eq (1) and (2), the subscript j indexes nodes in the current layer to be calculated, i indexes the node of the lower layer connected with the jth node, and w_{ij} denotes the weight at the connection between the ith and jth nodes. x_i is value inputted to ith node, b_j the is the bias term of the jth node, and r_j is the output value of the jth node. This general operation is continually performed from the first hidden layer to the output layer. Since another NN is also calculated by the general operation, the inner-production operation and activate function, this operation can be easily applied to another NN.

Moreover, since many inner-product operations can be replaced with a matrix multiplication, the MLP is more appropriate for CUDA implementation. As such, the computation-per-layer can be written as follows:

$$\mathcal{W} = \begin{bmatrix} w_{10} & w_{11} & \cdots & w_{1N} \\ w_{20} & w_{21} & \cdots & w_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ w_{M0} & w_{M1} & \cdots & w_{MN} \end{bmatrix} = \begin{bmatrix} \mathcal{W}_1 \\ \mathcal{W}_2 \\ \vdots \\ \mathcal{W}_M \end{bmatrix},$$

$$\mathcal{X} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_{11} & x_{12} & \cdots & x_{1L} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{NL} \end{bmatrix} = [\mathcal{X}_1 \quad \mathcal{X}_2 \quad \cdots \quad \mathcal{X}_L],$$

$$M = \mathcal{W} \times \mathcal{X} = \begin{bmatrix} \mathcal{W}_1 \cdot \mathcal{X}_1 & \mathcal{W}_1 \cdot \mathcal{X}_2 & \cdots & \mathcal{W}_1 \cdot \mathcal{X}_N \\ \mathcal{W}_2 \cdot \mathcal{X}_1 & \mathcal{W}_2 \cdot \mathcal{X}_2 & \cdots & \mathcal{W}_2 \cdot \mathcal{X}_N \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{W}_M \cdot \mathcal{X}_1 & \mathcal{W}_M \cdot \mathcal{X}_2 & \cdots & \mathcal{W}_M \cdot \mathcal{X}_N \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & \cdots & m_{1L} \\ m_{21} & m_{22} & \cdots & m_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ m_{M1} & m_{M2} & \cdots & m_{ML} \end{bmatrix}.$$

$$R = \text{sigmoid}(M) = \begin{bmatrix} 1 + e^{-m_{11}} & 1 + e^{-m_{12}} & \cdots & 1 + e^{-m_{1L}} \\ 1 + e^{-m_{21}} & 1 + e^{-m_{22}} & \cdots & 1 + e^{-m_{2L}} \\ \vdots & \vdots & \ddots & \vdots \\ 1 + e^{-m_{M1}} & 1 + e^{-m_{M2}} & \cdots & 1 + e^{-m_{ML}} \end{bmatrix}$$

where M is the number of nodes in the current layer, N is the number of nodes in the lower layer, and x_{ij} is the

*i*th feature value of the *j*th input vector. The result R_{ij} is the output of the *i*th output node for the *j*th input vector. Here, the subscript 0 means the bias term, and this is to make one matrix multiplication without the summation term in Eq 1.

When implementing the NN using CUDA, all input feature data for the NN cannot be transferred into the memories of the GPU, due to the limited memories of the GPU. Therefore, the proposed architecture divides the whole process into two parts. The first part is to make a suitable size of feature data for the memory of the GPU, and can also include a feature extraction step to extract features for the NN. However, this part is much slower than implementation on the GPU. Therefore, the OpenMP is used for parallel implementation of the first step, i.e. making feature data is concurrently performed on the multi-core CPU. The second step is to implement the NN using feature data received from the CPU. Then computational times of two parts are similar to each other, compared with implementation without OpenMP. Therefore, the efficiency of the proposed architecture is accomplished by the parallel use of the CPU and GPU.

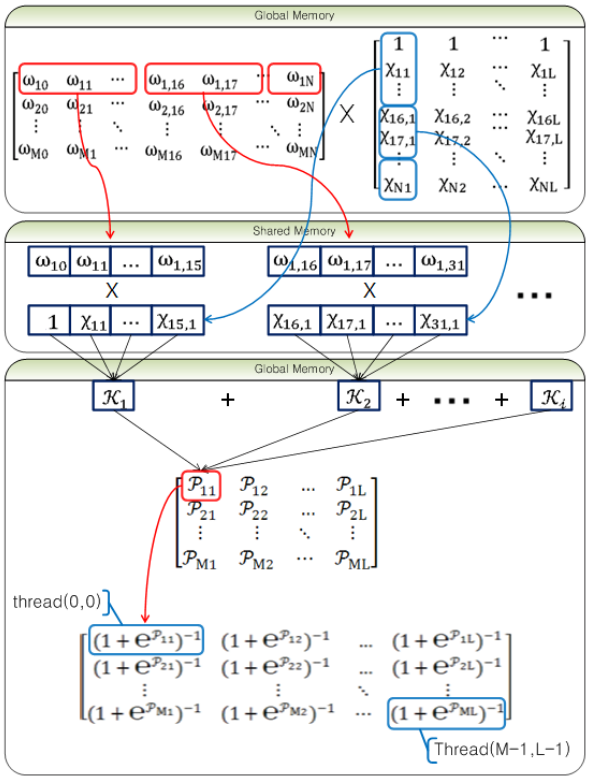


Fig. 3. Operations of NN using CUDA.

Fig. 3 shows matrix multiplication and computation of sigmoid function using CUDA. Since the CUDA can effectively compute matrix multiplication by using shared memories. In general operation in GPU, about 400-600 cycles are required to assess the global memories, but in memory environment of CUDA, only 4 cycles is required to access the shared memories. Therefore, the shared memories of the CUDA help to effectively compute the operations. The sigmoid function can be performed in parallel by allocating thread equal to the number of elements of matrix and then compute the operation in each thread independently.

4. Experimental Results

All experiments were carried out on an Intel core 2 Quad Q6600 CPU (2.4 GHz) and GeForce 8800 GTX graphics hardware. OpenMP help to process four sets of data on CPU in parallel. We evaluated the proposed method through the NN-based text detection application, section 4.1 describes the text detection, and section 4.2 shows result images and time complexity.

4.1. NN-based Text Detection

Recently, researchers have attempted text-based retrieval of image and video data using several image processing techniques [13]. As such, an automatic text detection algorithm for image data and video documents is important as a preprocessing stage for optical character recognition, and an NN-based text detection method has several advantages over other methods [13].

Therefore, this subsection briefly describes such a text detection method, and readers are referred to the author's previous publication for more detail [13]. In the proposed architecture, an NN is used to classify the pixels of input images, whereby the feature extraction and pattern recognition stage are integrated in the NN. The NN then examines local regions looking for text pixels that may be contained in a text region. Therefore, 1) gray values of the pixels at predefined positions inside an M×M window over an input frame is received as the input and 2) a classified image is generated as the output. After the feature passes the network, the value of the output node is compared with a threshold value and the class of each pixel determined, resulting in a classified image. Experiments were conducted using an 11×11 input window size, with the number of nodes in a hidden layer set at 30.

In the proposed architecture, the first step of previous sentence was performed on the multi-core CPU using OpenMP for parallel implementation, and Fig. 4 shows pseudo codes for the first step. The

second step was performed on the GPU, and Fig. 5 shows pseudo codes. In Fig. 5, two threads were allocated to perform the first step, thus the pseudo code including two indicators '**#pragma omp section**' is to allocate the thread.

```
for(check everyPixel of image)
{
 //Parallel Implementation using OpenMP
 #pragma omp parallel section
 {
    #pragma omp section
    {
       //pixel check in window range
       GetConfigMatrix(cpuData1);
    }
    #pragma omp section
    {
       //pixel check in window range
       GetConfigMatrix(cpuData2);
    }
 }
 //calculate neural net using CUDA
 ForwardCUDA(cpuData1,outputCUDAData);
 SaveOutputData(outputCUDAData);
 ForwardCUDA(cpuData2,outputCUDAData);
 SaveOutputData(outputCUDAData);
}
```

Fig. 4. Pseudo code for OpenMP performed on multi-core CPU.

```
//memory copy from CPU to GPU
cublasSetMatrix(CPUData, CUDAData );

//Result 0 = Weight0 * GPUData
//matrix multiplication of first layer
cublasSgemm(Weight0, CUDAData, Result0);
// sigmoid calculation of first layer
Sigmoid(Result0);

//Result1 = Weight1 * Result0;
// matrix multiplication of second layer
cublasSgemm(Weight1, Result0, Result1);
//sigmoid calculation of second layer
Sigmoid(Result1);

//memory copy from GPU to CPU
cublasGetMatrix(Result1, outputCPUData);
```

Fig. 5. Pseudo code for NN performed on GPU.

4.2. Result of Text Detection

Fig. 6 shows the result images according to the image sizes: (a,b) 320×240, (c,d) 571×785, and (e,f) 1152×15466. Figs. 6(b,d,f) show the pixel classification result for the input image Figs. 6(a,c,e), where a black pixel denotes a text pixel. The classification using a GPU produced almost the same results as without a GPU.

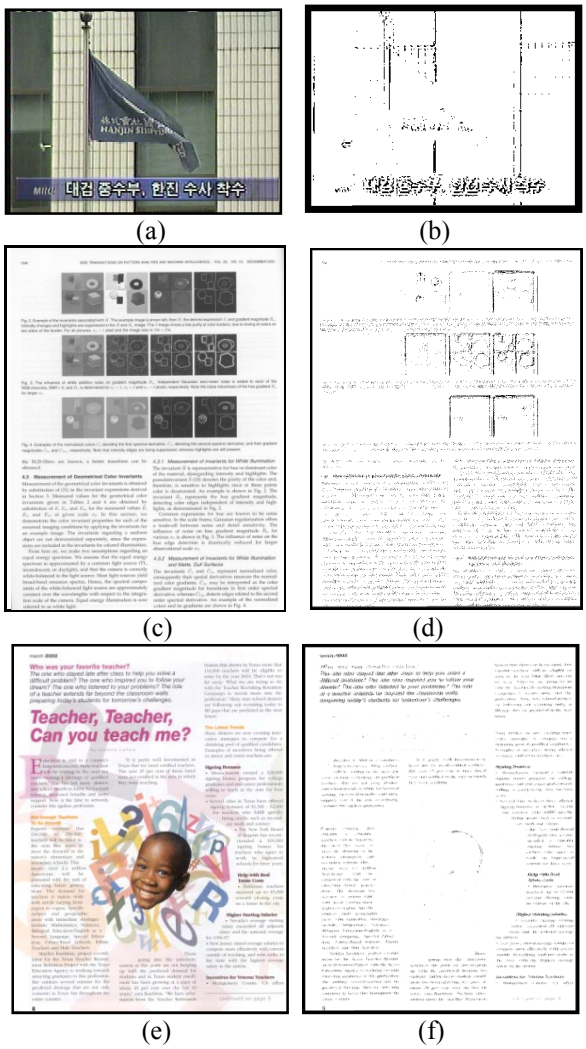
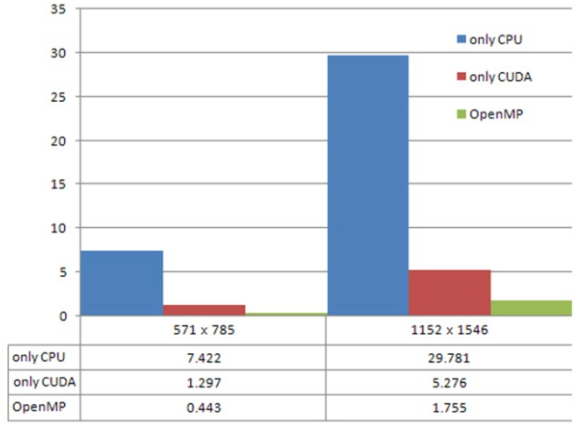


Fig. 6. Result images: (a,c,e) input images, and (b,d,f) result images.

Fig. 7 shows the computational times of Fig. 6, where x-axis indicates image sizes and y-axis indicates computational times (sec). As shown in Fig. 7, the proposed architecture showed about 20 times faster than the CPU-only and about 5 times faster than the GPU-only, and the computational times for pixel classification were significantly reduced using the proposed method. The reason why the proposed method showed faster computational times than GPU-only is we reduced the computational times to generate the data, which will be processed in GPU, in multi-core

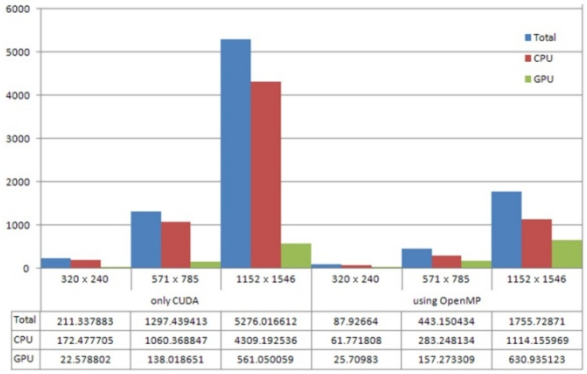
CPU using OpenMP. Therefore, we analyzed the computational times when using OpenMP.

Fig. 8 shows effectiveness of using OpenMP, where y-axis indicates computational times (msec). If only CUDA without OpenMP were used to implement NN, there is no little in differentiation of the computational times between CPU and GPU, i.e. computational times of GPU is 8 faster than CPU. The performance of GPU is maximized by accumulating a large number of input vectors that is dependent on the GPU configuration, thus the CPU generates the input vectors as much as possible, which will processed by GPU in one step. The OpenMP helped to reduce computational times processed in CPU, thus can reduce the differentiation of the computation times. Consequently, the OpenMP helped to reduce the bottleneck between the CPU and GPU.



	571 x 785	1152 x 1546
only CPU	7.422	29.781
only CUDA	1.297	5.276
OpenMP	0.443	1.755

Fig. 7. Computational times of three architectures.



	320 x 240	571 x 785	1152 x 1546	320 x 240	571 x 785	1152 x 1546
	only CUDA			using OpenMP		
Total	211.337883	1297.439413	5276.016612	87.92664	443.150434	1755.72871
CPU	172.477705	1060.368847	4309.192536	61.771808	283.248134	1114.155969
GPU	22.578802	138.018651	561.050059	25.70983	157.273309	630.935123

Fig. 8. Differentiation of computational times with and without OpenMP.

5. Conclusions

This paper proposed faster and more efficient multi-threaded implementation on both commodity graphics hardware and multi-core CPU. A CUDA was used, as the CUDA code is C language style and has less computational restriction while the traditional GPU could be programmed though only a graphics API that requires much special knowledge about computer graphics. Moreover, OpenMP, which can help to concurrently process more than two data with single instruction on multi-core CPU while processing only one data on GPU, was used to minimize difference between two computational times on only one graphics hardware. Based on the proposed architecture, we implemented neural network, where feature extraction is processed on multi-core CPU and main operation of NN consisting of inner-product operations and a activate function is processed on CUDA. The experiments evaluated the proposed implementation through NN-based text detection, and showed faster computational times on the proposed architecture than on only CUDA or CPU.

Acknowledgment: This work was supported by grant No.(R01-2006-000-11214-0) from the Basic Research Program of the Korea Science.

6. References

[1] K.S Kyong and K. Jung. "GPU Implementation of Neural Network", *Pattern Recognition*, Vol. 37, Issue 6, pp. 1311-1314, 2004.
[2] K. Moreland and E. Angel. "The FFT on a GPU", *Proceedings of SIGGRAPH Conference on Graphics Hardware,* pp. 112-119, 2003.
[3] J. Mairal, R. Keriven, and A. Chariot. "Fast and Efficient Dense Variational Stereo on GPU", *Proceedings of International Symposium on 3D Data Processing, Visualization, and Transmission,* pp. 97-704, 2006.
[4] R. Yang and G. Welch. "Fast Image Segmentation and Smoothing using Commodity Graphics hardware", *Journal of Graphics Tools*, Vol. 17, issue 4, pp. 91-100, 2002.
[5] J. Fung and S. Man. "OpenVIDIA: Parallel GPU Computer Vision", *Proceedings of ACM International Conference on Multimedia,* pp. 849-852, 2005.
[6] http://ati.amd.com/developer/
[7] http://developer.nvidia.com/object/cg_toolkit.html/
[8] http://www.opengl.org/documentation/glsl/
[9] http://graphics.stanford.edu/projects/brookgpu/
[10] http://www.nvidia.com/object/cuda_home.html/
[11] J. Fung and S. Mann. "OpenVIDIA: Parallel GPU Computer Vision", *Proceedings of ACM International Conference on Multimedia*, pp. 849-852, (2001).
[12] http://www.openmp.org/
[13] K. Jung. "Neural Network-based Text Localization in Color Images", *Pattern Recognition Letters*, Vol. 22, issue 4, pp. 1503-1515, (2001).