Accelerating Convolutional Neural Networks
using General Purpose Processing
on Graphical Processing Unit

Michał Wojtera, Technical University of Lodz
(15-09-2009, prof. Zygmunt Ciat, Technical University of Lodz)

Abstract

General purpose processing is relatively new approach, which has recently become much more widely adopted, thanks to GPU manufacturers providing programming solutions and very recently thanks to OpenCL standard provided by Khronos group. CUDA used in the experiment described in this article is an extension of C programming language, enables programming heterogeneous systems.

Convolutional Neural Networks are biologically inspired. They are based on the concept of “simple” and “complex” neuron cells[9]. CNNs are designed with multiple layers of neurons, use weight sharing to reduce the number of free parameters[3]. They exhibit convolution which occurs between layers[4].

1. Convolutional Neural Networks

Traditional model of pattern recognition involves hand-designed feature extractor that gathers information from the input and filters out irrelevant variabilities, and a trainable classifier that categorizes feature vectors into classes. Convolutional Neural Networks eliminate need for feature extractor. Instead “raw” inputs (e.g. normalized images) are fed in to the network and back-propagation is responsible for turning the first few layers into a feature extractor.[1,8]

The neocognitron [7], which can be considered as the first realization of convolutional networks has been first presented by K.Fukushima[7]. In his paper he uses results of experiments conducted by Hubel and Wiesel[9]. Fukushima exploits notion of “simple” and “complex” cells to construct a multilayered neural network. This was the first extensive use of receptive fields in artificial neural networks. Fukushima applied his network primarily to handwritten digit recognition. Later variants of convolutional networks have been applied for example to large scale zip code recognition and face recognition.

Task of recognizing written characters is complex for several reasons. It could be accomplished using an ordinary fully connected feed-forward network with some success.

However, input images are large and contain several hundred variables. A fully-connected first layer with even 100 hidden units, would contain tens of tens of thousands of weights. We could deal with overfitting problems if the training data is very limited. Moreover that many weights require significant amount of memory.

Never the less, no built-in invariance to translations or local distortions is the main caveat of unstructured networks. Character images or other 2D signals must be normalized and centered in the input field, but this kind of transformation can’t be perfect. Firstly, handwriting is usually normalized a word level, this can cause size, slant and position variations of characters, which in result will cause the position of a feature in an input image to vary.

Although in principle big enough fully-connected network could be devised to produce invariant response, this would most probably result in many units with identical weight patterns at different places in the input. What is more, learning process for all these weight configurations would require very large and varied training set, considering the space of possible variations.

Model complexity and the number of weights can efficiently reduced by weight sharing. This is an advantage when images with high-dimensional input vectors should be presented directly to the network instead of explicit feature extraction and data reduction which is usually applied before classification.[1]

Forcing replication of weight configuration across space results in shift invariance.[1]

Fully-connected architectures ignore topology of the input. But nearby character images are highly correlated. Advantages of extracting and combining local features before recognizing objects are widely known[1].
Convolutional networks restrict the receptive fields of hidden units locally and therefore ensure extraction of local features.

CNN topology is more similar to biological networks based on receptive fields and improves tolerance to local distortions.

Concluding, multilayer back-propagation networks can learn complex, high-dimensional, non-linear mappings from large collections of examples. These features make them good candidates for image recognition or speech recognition[1].

CNNs exhibit convolution between layers and are resistant to input distortion and rotation.[10]

2. Graphical Processor Programming

General Processing on GPU is a relatively new approach to using raw processing power of these units. Currently both major GPU manufacturers provide solutions that enable using Graphical Processors for general computation. Recently OpenCL (Open Computing Language) standard for GPGPU was released, all considered writing programs for GPUs is becoming much more common. Technology used by the author, namely Compute Unified Device Architecture is an extension of C programming language. GPU and CPU are treated as separate devices. CUDA Serial portions of the code are run on CPU, while parallel portion is executed on device as kernels, GPU executes one kernel at a time, each kernel is executed in many threads.

Kernel is called on host and executed on the device.

CUDA threads are extremely lightweight, very little creation overhead, instant switching and therefore can be used to hide latency in memory loads and stores.

CUDA uses 1000s of threads to achieve efficiency, essential oversubscribing processing elements, so when a thread is stalled by a memory access or synchronization, it can be swapped out without any performance penalty.[5]

When a CUDA code is launched it is executed as an array of threads. Each thread is running the same code and each is given a separate ID to compute memory addresses and make control decisions.

2.1 Thread Cooperation

Threads can share results to avoid redundant computation and can share memory accesses to reduce the bandwidth requirements.

Thread cooperation is available only within batches of threads.

2.2 Thread Batching

When a kernel is launched, it’s launched as a grid of thread blocks.

Threads within a block cooperate via shared memory and synchronize. Threads in different blocks cannot cooperate.

Transparent Scalability is achieved by allowing hardware to schedule thread blocks on parallel multiprocessor, where a multiprocessor is a processor that executes one or more groups of threads, below we can see that grid is divided into thread blocks to fit number of processors in the GPU.

This arrangement allows programs to scale on different GPUs.

CUDA kernel threads have access to a variety of memory spaces which differ in scope, size, lifetime and latency.

Each thread has read/write access to per thread memory, which is on-chip and exists for the lifetime of the thread. Unqualified variables that do not fit in register reside in local memory which is uncached in DRAM. Lifetime of both is the lifetime of the thread.

All threads within a block have access to fast on-chip shared memory. Lifetime of the shared memory is the lifetime of the block. (Size 16KB on 8 and 10 series)

All threads have read and write access to the same global memory, which is located in the device DRAM (up to 4 GB), it is uncached and exists for the lifetime of the application.

In addition host has read and write access to global memory, as such Global Memory is used to transfer data between host and the device.
2.3 Physical Memory Layout

Fast shared memory and registers are on the chip, in device DRAM are device Local and Global memories. Local memory is local in scope, but physically it resides in device DRAM. From performance perspective one wants to leverage Registers and Shared Memory in order to minimize use local and global memory. Device to Host memory transfers go through Global memory only.

![Fig.2. Memory configuration in CUDA](image)

3. The experiment

Conducted experiment consisted of running parallel program on the Graphical Processing Unit (GPU) and using the emulator. This results were then compared to the program run on CPU[10].

In order to get reliable results both programs for CPU and GPU used identical layout, training data and testing set. Training and testing was done using MNIST database of handwritten digits. Moreover, the same computer set was used to ensure accurate results. The computer was equipped with 2,66GHz Intel Core2 Duo processor with 6MB of level 2 cache, and 4GB of DDR2 Random Access Memory. The GPU used in the experiment was Nvidia GeForce 9600M GT, clocked at 1,25 GHz, with 512MB of dedicated RAM. Graphical Processor had 4 multiprocessors capable of running 512 threads per block. Each program was run in 1000 loop results were stored, mean and variance was computed.

![Fig.1. Simulation results](image)

4. Results

As we can see rewriting and recompiling the code for GPU enabled author to significantly reduce recognition time. As we can see in Figure 1, the recognition time was decreased four orders of magnitude. Another interesting fact is that even emulation takes more time it isn't significantly slower than CPU computation.

5. Conclusions

In this paper author has presented Convolutional Neural Networks and Compute Unified Device Architecture. As was shown and proven in the experiment CNNs are capable of recognizing handwritten digits without preprocessing as was also discussed[1-4]. Implementing the CNN in CUDA resulted in four orders of magnitude speed up of the recognition process, this leads author to a conclusion, that it is viable to implement larger networks or with bigger number of free parameters, while maintaining speed of computation. However promising this results are only preliminary and demonstrate capabilities of both CNNs and General Processing on GPU, authors goal of recognizing handwritten polish characters significantly increases complication of the problem as well as set of possible symbols. Moreover we can see that even with multicore CPU at 2,66 GHz, massive parallelism of the GPU gives much better recognition time, provided of course that code written for GPU is adapted to it's capabilities. Lastly we can see that emulation results are worse than program written natively for CPU and although it takes 40% more time to emulate, it can still be seen as a viable option for preparing CUDA solutions on computers where CUDA capable GPU is not available.

2. Bibliography


[5] **CUDA Programming Model Overview**


Authors:
MSc. Michał Wojtera
Department of Microelectronics and Computer Science
Technical University of Lodz
ul. Wólczańska 221/223 bud. B18
90-924 Łódź
email: mwojtera@dmcs.pl