Efficient mapping of the training of Convolutional Neural Networks to a CUDA-based cluster

Jonatan Ward Sergey Andreev Francisco Heredia Bogdan Lazar Zlatka Manevska

Eindhoven University of Technology, The Netherlands
http://parse.ele.tue.nl
{j.j.ward, s.andreev, f.j.heredia.soriano, b.m.lazar, z.manevska}@tue.nl

Abstract

We propose a method to parallelize the training of a convolutional neural network by using a CUDA-based cluster. We attain a substantial increase in the performance of the algorithm itself. We research the feasibility of using batch versus online mode training and provide a performance comparison between them. Furthermore, we propose an implementation of an alternative algorithm to compute local gradients which increases the level of parallelism. To conclude, we give a set of best practices for implementing Convolutional Neural Networks on the cluster.

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: Parallel Architectures; C.4 [*Performance of Systems*]: Modeling Techniques

General Terms Performance

Keywords GPU, CNN, Neural Networks

1. Introduction

This paper presents a solution that uses GPUs to take advantage of the inherent parallelism available when training neural networks. The focus is put on a Convolutional Neural Network (CNN) used for visual object recognition. Online- and batch-mode learning methods were implemented and the results from the two methods are presented and compared.

1.1 Cluster description

The cluster used in this project contains four nodes. Each node has a Intel Core i7 960 (3.20 GHz) processor, 12 GB of main memory and four NVIDIA GTX570 graphical processor units (GPUs). The graphical processors are connected via PCI Express 2.0. The nodes are interconnected via Gigabit Ethernet (1Gb/s).

1.2 Convolutional neural networks

Artificial Neural Networks (ANNs) have emerged as a powerful technique in machine learning for solving various practical problems like pattern classification and recognition, medical imaging, speech recognition and control.

A Convolutional Neural Network (CNN) is an extension of an ANN optimized for two-dimensional pattern recognition because it uses shared weights and less connections, which greatly reduces the solution space. The architecture of a CNN is shown in Figure 1.

The biggest problem with using a CNN is the long training time since it is computationally intensive. Training with large data sets could take up to several days or weeks.

Since the training involves big amounts of floating-point operations in every training step, it is well suited for running on modern graphical processing units (GPUs).

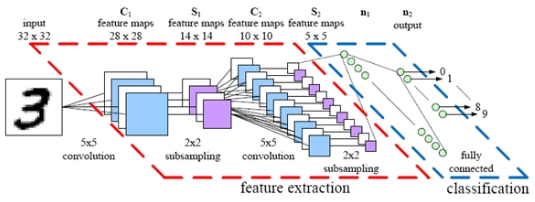

Figure 1. Convolutional neural network architecture [3]

2. Solution

The process of CNN online training consists of the following phases: convolution phase, compute local gradients phase and update weights phase. A training epoch is the exposition of a training set to the neural network. In this particular case, an epoch is comprised of a set of image patterns. Each pattern in the epoch goes through those phases and updates the network weights.

After one epoch iteration, the network is tested by running the convolution phase on test patterns and the error is computed.

After profiling the CPU reference implementation, the results showed that 59% of all computations goes to the convolution phase, 28% to the compute local gradients phase and 11% is spent on the update weights phase.

2.1 Used parallelism

The general approach to parallelization was splitting the image in tiles that are represented by thread blocks per output feature map. This division is represented in Figure 2. Each tile is analogous to a thread block, and each pixel is represented by a thread. We also use a three-dimensional grid: its *x*, *y* and *z* dimensions are represented by the feature maps' width, height, and quantity, respectively.

Convolution phase

Convolution is done using the general approach described above. Each convolution operation is applied per convolution kernel,

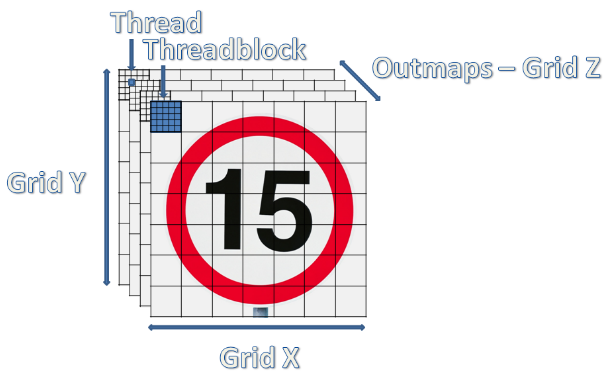

Figure 2. Image tile processing

therefore adding additional loops inside each CUDA kernel was required. As a result of this, the performance on the GPU is affected. In order to mitigate this and given the relatively small size of the loops, they were unrolled using boost preprocessor library.

The convolution phase applies the sigmoid activation function at the end. That function updates the value of every pixel using formula (1).

$$\frac{1}{1+e^{-x}} \tag{1}$$

Since there are no dependencies to update the value of a pixel using the sigmoid function, the image was processed in flat thread blocks (lines) in order to coalesce the memory access.

Compute local gradients phase

The calculation of the weights gradients is done in two steps. First the local gradients are calculated by back-propagation from the succeeding layers. Second, the local gradients are used to compute the gradients of the weights.

The original implementation of local gradients calculation gave poor results on the GPUs due to the fact that it was only possible to parallelize per convolution kernels instead of image tiles.

The algorithm utilizes the connections of the Feature Extraction Layers (FELs) from each layer to the FELs of its immediately precedent layer. Information about these connections is contained in the *connection matrix*. This behaviour is denoted as *Push* calculation, since a layer *pushes* the data to its precedent layer for calculation purposes.

In Figure 3 it is shown that, for a loop index *i*, data is pushed from the FEL$_0$ in layer *i* to FEL$_0$, FEL$_1$ and FEL$_3$ in layer *i - 1*. The rest of the connections have been ommited for clarity, however the remaining FELs in layer *i* also have connections to the layer on the left.

Since inside every iteration *i* the algorithm looped through the FELs in the layer *i*, it may happen that FELs in layer *i - 1* were written in different (and unpredictable) points in time. It is important to notice that the connection matrix can have infinite network-dependent connection combinations. Thus it was not possible to determine when or how many times the pixels in a FEL in the preceding layer would be updated.

As a result of this, the kernel could not be parallelized based on the pixel of each FEL. Not parallelizing based on the FELs meant that the sum of the gradients was serialized and did not improve the overall processing time.

In order to overcome this problem, the concept of *inverted connection matrix* was introduced. This new matrix, which is created when parsing the network, contains the relevant information to calculate the gradients of the FELs from a given layer, by pulling relevant data from the FELs of the succeeding layer. This behaviour was denoted as *Pull* calculation.

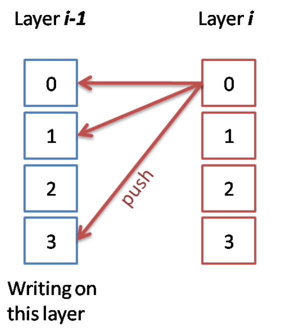

Figure 3. Push calculation

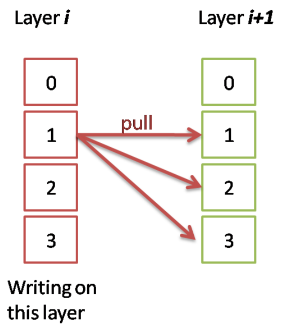

Figure 4. Pull calculation

As shown in Figure 4, FEL$_1$ in layer *i pulls* all the information it needs from layer *i* + *1*. By using this pulling mechanism, it could be garanteed that the pixels in FEL$_1$ are only updated when reading the *inverted connection matrix* that corresponds to that FEL, avoiding memory access conflicts.

The pulling computation of the local gradients enables the parallelization per FEL in each layer.

Update weights phase

Update weights phase consists of two parts: summation of local gradients followed by an update of the weights and calculation of bias values per layer. Calculating bias values is done using the standard optimized way of summing arrays on GPUs.

Updating delta weights requires writing to the same memory address a number of times equivalent to the size of the outmap, rendering the use of loop unrolling untenable. Hence, the basic idea pursued was to map those parts of the algorithm that wrote to the same destination address to a single thread block. The remaining CUDA kernel dimensions are then used to handle outputmaps, algorithm kernels and inputs per output map; since then each threadblock would then write to a different address for each outmap, etc. The thread blocks were used this way in order to avoid synchronization issues between different CUDA threads attempting to write to the same memory address. Inside the update delta weight kernel function, each thread iterates over a set of pixels of the input associated with the output map. Once all threads in the block complete, their results are added and stored in the final weight location.

2.2 Performance gains

Table 1 describes the performance gains from each function based on the reference implementation on the CPU. It can be seen that *compute_error* actually decreases the overall performance but it is required to be executed in the GPU. This choice is less time-consuming than copying the information to the CPU, executing the CPU implementation of the *compute_error* function and then copying the information back to the GPU.

Function name	CPU[ms]	GPU[ms]	Performance gain
uchar2float	0.0736	0.0477	1.54X
uchar2shiftedfloat	0.0603	0.0389	1.55X
run_convolution_layer	3.6957	0.1742	21.21X
compute_local_gradients	0.5722	0.1139	5.02X
update_weights	1.5660	0.1604	9.77X
compute_error	0.0023	0.0478	0.05X

Table 1. Execution times comparison

The function *run_convolution_layer* has the largest gain, mostly because the sigmoid function takes a considerable amount of time in the CPU version. Since it doesn't have data dependency, when it runs in the GPU it is completely parallelized.

The GPU execution times for *compute_local_gradients*, *run_convolution_layer* and *update_weights* are similar to each other because the paralellism is obtained in the same manner (pixels per output map). The main difference in the gain for each one is the level of optimization with respect to the CPU version.

Table 2 shows the execution time for the operations when run after the first time. We used static variables to avoid allocating and freeing memory for each execution of the GPU functions.

Function name	CPU[ms]	GPU[ms]	Performance gain
uchar2float	0.0494	0.0234	2.11X
uchar2shiftedfloat	0.0600	0.0299	2.01X
run_convolution_layer	3.6648	0.1694	21.63X
compute_local_gradients	0.5723	0.1033	5.54X
update_weights	1.5590	0.1420	10.98X
compute_error	0.0023	0.0309	0.08X

Table 2. Execution time using static variables

3. Batch learning

Another approach to neural networks training is batch mode. The main difference with respect to the online mode is the frequency with which the weights and bias values are updated.

In online mode, the weights and bias values are updated every time a single image is sent through the network. In other words, every time an image passes through the network, the delta values for bias and weights are calculated and added to the previous bias and weights values, respectively.

In batch mode, the network is trained on a batch of images. For each batch of images, delta values for bias and weights are accumulated. It is only after the batch of images has been processed that the previous values for bias and weights are updated. This leads to the open issue that batch learning may take more iterations to reach the desired error minimum. However, since the convergence speed to this value relies on many factors such as learning rate, batch size, number of neuron layers, and so forth, no general conclusion can be reached regarding wich mode is better.

Batch training maps naturally to the use of multiple GPUs. First, all the GPUs start with the same weights and bias values. Then, each GPU runs the batch of images. These two steps comprise the setup for batch training.

Once the batch is loaded, the GPUs are ready to process it and generate the delta values locally. After each GPU finishes processing, it sends the generated delta values to the GPU that is in charge of summing the results.

A sum of the delta values from all GPUs is calculated. The result of the sum is added to the previous bias and weights values. Finally, the updated bias and weights values are copied back to all GPUs. That process is shown in Figure 5

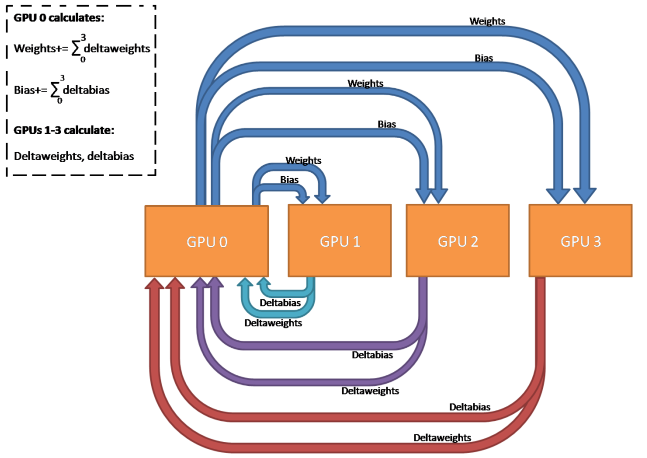

Figure 5. Batch training

The weights and bias values update was done on one GPU due to the nature of the algorithm. CUDA 2.x introduces a memory copying mechanism between devices which gives a substantial speedup. By using one device as a master, it is possible to avoid unnecessary overhead given by copying between host and device memories.

In order to mitigate the complexity of extracting and sending delta weights/bias values from the neural network data structure, it was decided to allocate delta weights/values for all layers in a continuous memory block as shown in Figure 6. Doing it in this way provides two main advantages. First, it allows to easily copy the values between devices by providing the pointer to the memory block and size of the block. Second, it makes parallelizing the summation of values from all devices straightforward: the values are in the same position in each block, as shown in Figure 7.

3.1 Performance gains

The batch process is splitted in three computational blocks:

1. **training** the network with the batch of images,
2. **copying** delta values from GPU devices to the GPU master and sending updated weights and bias back to the devices, and
3. **summing** delta values with previous values for weights and bias.

Table 3 shows the results of processing four images using batch method on four GPUs (one image per device).

Given that the amount of data we need to transfer and sum on the master GPU is independent of the number of images, and training of images is not dependent on other devices, batch method speed-up increases linearly with the number of images that should be processed per GPU. However, the accuracy of the algorithm decreases subsequently.

Processing the same number of images using online method takes 2.68745 ms and it increases linearly to the input.

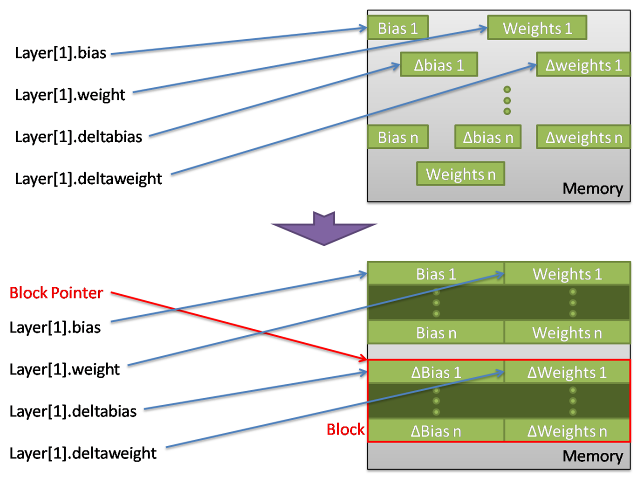

Figure 6. Block structure

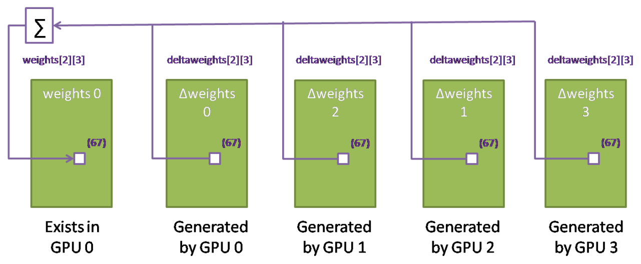

Figure 7. Delta weights/bias summation

Computational block	Time[ms]
Training	1.5379
Copying	0.1467
Summing	0.1276
Total	1.7743

Table 3. Batch processing results

One of the biggest bottlenecks in the proposed solution is that all data has to be copied to a master GPU which will compute the weights and bias values. The copying operation is blocking the GPU master: every GPU device has to wait until the operation is finished before its deltas can be processed. The same applies when the master needs to replicate updated values to the GPUs: the master has to wait until the operation is finished before it sends the updated values to the next device. However, the latency introduced by synchronization is smaller than the gains obtained from a faster training.

Table 4 shows the comparison between the execution times for different training modes. The CPU reference corresponds to a sequential execution of all the algorithms. Single GPU is the optimized version of the CPU implementation for parallel execution in the cluster node. Batch includes parallel execution in four GPUs within a cluster node.

The results were obtained using a training set of ten epochs, ten images per epoch, and ten test patterns. The main differences between a small and a big network are the size of the input images (96 x 96 pixels for the former, 1080 x 1080 pixels for the latter), the number of layers (5 for the former, 10 for the latter).

Training type	Execution times [ms] Small network	Execution times [ms] Big network
CPU reference	961	1235991
Single GPU	60	19159
Batch	46	5232

Table 4. Time execution comparison with different modes

The feasibility of distributing the batch learning process over multiple cluster nodes was analyzed. The batch training has to be run in every node. Then, the nodes have to sum the deltas locally before sending them to the master node. It will sum the received deltas with the current bias and weights values, and transfer the updated values to the cluster nodes in order to start the next training iteration.

The theoretical numbers for copying the data were obtained taking into account the current configuration of the nodes, which are interconnected via 1Gb/s Ethernet. It was calculated that the time to copy the small-sized network's data back and forth will be around 16 ms, which considerably affects the overall training performance. Thus batch learning using different cluster nodes is not recommended unless the bus bandwidth between them is increased.

Moreover, the batch learning has better performance when running more images per device. However, the batch size must be chosen carefully, since using a large number of images per batch would affect the accuracy of the training. Unfortunately, there are no state-of-art techniques or heuristics to determine that parameter since that field is still under research.

4. Conclusion and future Work

In conclusion, Convolutional Neural Networks (CNNs) are highly suitable for parallelization on Graphical Processing Units (GPUs). That allows to significantly reduce the training time from several days or weeks, to the order of minutes and hours, depending on the network size, and the image resolution. Considering the single GPU implementation with respect to the CPU reference implementation, a speed-up of factors 16 and 64 are obtained for a small and a big network, respectively.

The standard implementation of the compute local gradients phase does not exploit the full parallelism capabilities that a CUDA device provides due to the nature of the pushing algorithm. In contrast, the pulling version that was introduced in this paper achieves an improvement of 30 times in the execution time.

By using batch mode learning, the training was distributed over multiple GPUs within a single node in the cluster. Running the training on multiple GPUs decreases the overall execution time even further with respect to the results obtained with the online mode training, which uses a single GPU. Considering the batch implementation with respect to the CPU reference implementation, a speed-up of factors 20 and 236 are obtained for a small and a big network, respectively.

However, the potential performance improvement achievable by running batch learning in several cluster nodes would be negligible due to the overhead that is generated by copying the data between them. For small-sized networks, copying the data between nodes is around 11 times slower than the actual batch training on a single node.

The optimal solution would be to train a different network in each cluster node using batch mode. Since these networks would be independent, different configurations are possible. After training, these networks can be compared in order to decide which one has the highest accuracy. Such an approach would utilize the full computational power of the cluster.

References

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[2] Y. Lecun, F. Huang, and L. Bottou. Learning methods for generic object recognition with invariance to pose and lighting. *Proceedings of CVPR04*, 2004.

[3] M. Peemen, B. Mesman, and H. Corporaal. Efficiency Optimization of Trainable Feature Extractors for a Consumer Platform. *Proceedings of the 13th International Conference on Advanced Concepts for Intelligent Vision Systems*, 2011.

[4] S. Suresh, S. Omkar, and V. Mani. Parallel Implementation of Back-Propagation Algorithm in Networks of Workstations. *IEEE Transactions on Parallel and Distributed Systems*, 16(1):24–34, 2005.

[5] D. R. Wilson and T. R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16:1429–1451, 2003.