Neural Network Simulation

The recognition application

Panagiotis Afxentis
Eindhoven University of Technology
P.D.Afxentis.Axentis@tue.nl

Alicia Sánchez Crespo
Eindhoven University of Technology
A.Sanchez.Crespo@tue.nl

Ying Zhang
Eindhoven University of Technology
Y.Zhang@tue.nl

Abstract

This paper presents the GPU mapping of the recognition algorithm of a Convolution Neural Network (CNN). This work is based on a C-implementation of the application. The mapping to GPU was performed through different approaches which are explained in detail. The improvements achieved by each approach are presented as well as the overall speed up of the application. The GPU implementation demonstrates the feasibility to integrate the algorithm in a real-time system as the frame rate accomplished is 30fps. In order to examine the scalability of the application, the use of multiples GPUs was explored. The CPU implementation is executed on a 800MHz AMD Phenom. The GPU platform used for the experiment is an Nvidia GeForce GTX570.

1. Introduction

This paper will introduce the work of mapping a visual object (specially the road signs) recognition algorithm on a GPU platform(s), which is to accelerate the implementation for real-time application requirements. As a matter of fact, this algorithm is the recognition part of a vision system based on a Convolutional Neural Network which is based on a trained network structure gained from the training part of this vision system [1].

Generally speaking, the trained network structure consists of four layers, shown in Figure 1: the first two layers (depicted in red rectangle) are the feature extraction step and the last two layers (depicted in blue rectangle) are used for classification. Each layer is based on the same principle but with different workload. Every layer is composed by several outmaps. An outmap is the result of the convolution of an outmap from the previous layer with a predefined set of weights. A constant value, called bias, is added to the sum. To generate the final output value the previous sum is passed through a sigmoid activation function, see Eq.1. This process should be performed on an outmap as many times as the connections between this outmap and the previous layer, accumulating the intermediate results. The input of the algorithm is a 1280x720 HD video frame which will be processed by the trained network structure, generating an output that gives information about the existence of a sign specifying the detected sign type.

$$\varphi(x) = \frac{1}{1+\exp(-x)} \qquad (1)$$

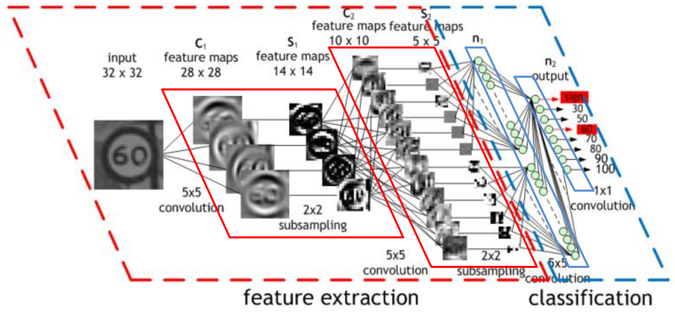

Figure 1. An example of CNN architecture

The recognition algorithm has the following two main features. Firstly, it is time-consuming with a large computational workload, especially for a high resolution image. Secondly it has a lot of potential parallelism, as it is a pixel based image processing algorithm. Furthermore, the algorithm is implemented using several nested loops which makes GPU a good candidate for gaining speed up as it supports data level parallel computing. Thus, the main challenge of this work is to exploit the parallelism of the algorithm as much as possible to make it run as fast as possible.

The content of the paper is as follows. In Section 2, the original implementation is profiled to find the time consuming functions. Section 3 focuses on the single GPU solution for the algorithm, which exploits the parallelism, and the corresponding profiled results. Section 4 presents the multiple GPUs solution for the algorithm based on OpenMP. In Section 5, the paper is summarized and concluded. Section 6 discusses future work that could be done.

2. Application profile

Given the C-implementation of the CNN, the application was profiled in order to identify the most time consuming functions. The application profile will help to determine which parts of the code will be mapped to a GPU.

In the code given, all the computations performed in one image were done by a single function. The total time to process one HD image is *3580ms*. Different parts of this function were measured in order to analyze where to exploit parallelism. It is worth to mention that for these measurements the compiler was not able to take advantage of all the possible optimizations. The time required to process every layer is shown in the second column of Table I, where it can be observed that the third layer is the most time consuming due to the network structure. In Table I, the third column shows the time needed to perform the convolution of a set of weights with an image from the previous layer. The differences between the layers are associated with the different size of the images and the different size of the convolution windows. In this application parallelism can be exploited using GPUs based on the fact that every outmap needs the convolution of several previous images which are independent, together with the fact that every layer is composed by several outmaps. It has also to be considered that the computation of every output pixel is independent of its neighbors. Moreover, it was measured the time needed to perform the activation function. The results obtained are shown in the fourth column of Table I. Using the special function units of the GPU may result in a faster computation of the non-linear sigmoid activation function [2].

Even if it is not directly related with the application itself, the time needed to convert the image from unsigned char to float and vice versa was computed. The results obtained for the functions *uc2f and f2uc* are approximately 6ms and 0.7ms, respectively. These values don't seem important when comparing to the 3580ms needed to process every frame but once the application will be speeded up, they will become more significant. As according to Amdhal's law, the speed up of the application will be limited by the sequential part of the algorithm. For this reason, these two functions were also decided to be implemented in GPU.

TABLE I
TIME PERFORMANCE OF THE APPLICATION

Layer	Complete layer (ms)	Convolution with one feature map (ms)	Activation function (ms)
1	445	33	40
2	646	8	10
3	4350	6	10
4	357	0	10

3. Single CPU-GPU implementation

3.1 First approach

Based on the results mentioned above a first GPU implementation is made. This involved the gradual incorporation of the layers of the network into GPU kernels. Initially the most time consuming layer, the third one, was mapped on a GPU. More specifically, different kernels were created for the functions implemented into the layer such as :

- the convolution between an image of the previous layer and the corresponding set of weights, which produces an intermediate result for a feature map of the current layer.
- the accumulation of all the intermediate feature maps in order to obtain the final feature map.
- the addition of the bias value.
- the sigmoid activation function.

The convolution kernel, which involves one of the most time consuming function of the algorithm, is implemented in terms of exploiting the data level parallelism (DLP) in two ways. Firstly, an internal parallelization method is applied as, in every row of an intermediate feature map of the layer, all the pixels are computed in parallel by setting the appropriate number of threads per block. Secondly, by defining the number of blocks in the grid as the number of the temporary maps that produce one specific feature map, the computation of each of the intermediate maps is done in parallel. For that purpose, a vector structure is created for every outmap of the layer, see Figure 2. By applying these two DLP methods, each temporary map is not only *generated faster* because of the threads that compute its pixels in parallel, but is also computed *in parallel* with the other intermediate maps. Because of the CNN structure, the first layer will not profit of the gain of computing the intermediate outmaps in parallel, as the number of connections with the previous layer is only one. This results in a significant speedup in the generation of a single feature map in the layer.

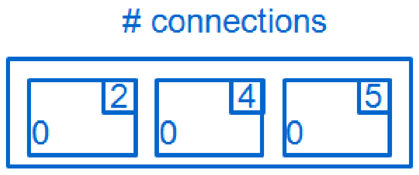

Figure 2. An example of a vector used for the computation of outmap0

The fast shared memory is used in order to avoid the data transactions to and from the slow global memory. Its limited size determines the priority given to the data that shall reside to it and the way it will be refreshed by new data. It was decided to store in shared memory as many lines of the image as the corresponding convolution window size. Despite this limiting factor, the use of shared memory is considered to be a critical improvement to this implementation due to the reusage of many data for the convolution kernel. Further speedup is achieved with the use of the function given by Eq.2 into the activation kernel. This function is implemented in the special functions units of the GPU and provides faster single-precision floating point division than the division operator. The use of this function might introduce inaccuracy but in this specific application the result was not degraded.

$$\_\_fdividef(1,1+exp(-x)) \qquad (2)$$

A last step for this first approach is the implementation of the functions that perform the image conversion in GPU by the corresponding kernels, thus exploiting the large amount of independent computations within them.

Initially, as mentioned before only the third layer was mapped on GPU. The fact that the code was kept generic made possible to map all the layers of the network on the GPU kernels described above. In this approach, a considerable speedup of *8.75 times* is depicted in the execution time, which is now *420ms* per image.

3.2 Second approach

The second approach was based on the previous one. As it has been explained above, one of the way used to exploit parallelism was the computation in parallel of all the previous images needed to generate one outmap. The idea of the second approach is to broaden this concept to all the outmaps. That is to say that all the intermediate outmaps needed to compute all the final outmaps will be processed in parallel.

For that purpose it was necessary to define in every layer a matrix structure that contains the corresponding set of weights needed for every final outmap. The size of this matrix is determined by the number of outmaps as well as the maximum number of connections with the previous layer. Another matrix structure is needed to store the intermediate feature maps resulting from the convolution, see Figure 3. In order to implement this approach it was necessary to add a second dimension to the grid definition in the convolution kernel. This dimension is set to the number of outmaps so as one thread block computes one intermediate feature map.

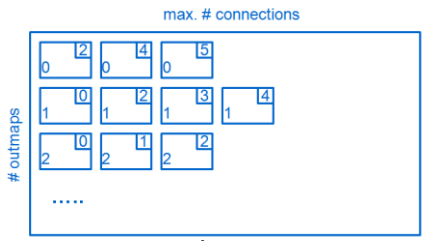

Figure 3. An example of a matrix structure

In this approach, the accumulation kernel was modified as now it has to add together the intermediate results associated to each final outmap. With the implementation of this approach, a speed up of *3.5 times* was achieved, being now the total execution time *120ms* per image.

3.3 Third approach

The downside of the previous approach was the need for extra GPU memory space for intermediate results in order to exploit parallelism when generating the outmaps. The main optimized points of the new approach are as follows.

Firstly, in each CNN layer, it is needed to allocate new temporary GPU memory space, as well as transfer data to this space and free it afterwards. Undoubtedly, these operations are time consuming and unnecessary. In order to avoid this extra GPU memory space for storing the intermediate results in each CNN layer, the corresponding memory space is just allocated once (i.e. image, weight, conmat and bias) large enough for the second approach implementation. The GPU memory space allocated for the image is used for both intermediate and final outmaps. In this way, it is possible to avoid the previous unnecessary operations. Applying these changes, the execution time for each frame gets 2 times faster.

Secondly, based on the cuda profile the occupancy of the convolution is relatively low indicating that more parallelism can be exploited in this kernel. Based on this observation, the 2D CUDA grid for convolution kernel was extended to a 3D grid so that the image height is divided in tiles computed in parallel, using a greater number of thread blocks. After experimenting, it was measured that the convolution kernel has the best performance when dividing the height into groups of 64 lines. Each thread block processes one group of lines sequentially.

Additionally, another relatively significant performance enhancement for convolution kernel could be reached by loop unrolling with template. The convolution window size was set as a template variable so as to unroll the loops needed to perform the convolution. It was also needed to take into account the different sizes in every layer.

Last but not least, several milliseconds enhancements on the total execution time per frame were achieved by applying other optimizations. For example, in the first approach bias addition and activation computation in each CNNs layer are in separated kernels and finally they are merged into the convolution kernel and accumulation kernel respectively.

After these optimizations on single GPU, the final execution time for each frame is around *3.5 times* less than the second approach, being now *33ms*, which corresponds to *30fps*.

3.4 Results

Table II summarizes the speedup achieved in the performance of the application in every approach. The total speedup represents the time improvements with respect to the original version. As it can be observed, the total speed up obtained with the final approach is 108 times. It is necessary when analyzing these results to take into account that the original version could be further optimized with software techniques such as SSE, loop unrolling, subword parallelism.

TABLE II
TIME PERFORMANCE OF THE DIFFERENT APPROACHES

Version	Execution time (ms)	Speedup	Total speedup
Original	3580	--	--
First approach	420	8.75 x	8.75 x
Second approach	120	3.5 x	30 x
Third approach	33	3.5 x	108 x

For the last approach, Cuda profiler was used to further analyze the kernel characteristics, such as the memory throughput, occupancy and the kernel execution time. Table III highlights these characteristics. Based on the results, in the convolution kernel it was observed that the third layer remains the most time consuming as noticed in the application profile of the original version. The low memory bandwidth of this kernel, together with the fact that the accesses to memory are coalesced, indicates that the convolution kernel is limited by the big amount of computations needed. This is not the case in the accumulation kernel which has a high memory bandwidth. The number of computations in this kernel is significantly less than in the convolution. Thus, this kernel is limited by the memory accesses. Regarding the occupancy, both kernels have a relatively high occupancy close to the theoretical limit of 1. The lower result of the first layer in the convolution kernel is related to its CNN structure which requires less number of thread blocks for the computation.

TABLE III
CUDA PROFILE RESULTS

Kernel	Layer	Execution time (µs)	Memory Bandwidth (GB/s)	Occupancy
Convolution	1	1834.7	9.8	0.667
	2	3792.9	17.3	0.833
	3	17262.0	17.5	0.833
	4	5721.3	53.4	0.833
Accumulation	1	150.0	73.6	0.833
	2	183.4	93.5	0.833
	3	1503.5	103.8	0.833
	4	1419.5	111.5	0.833

4. Multiple CPU-GPU implementation

Having optimized the single CPU-GPU implementation reaching an execution time of *33ms* per frame, the possibility of extending the execution of the application to multiple GPUs was analyzed. Due to the data dependencies among the different layers of the CNN and the large amount of frames needed to be processed, it was decided to further exploit the parallelism in terms of frames. Using OpenMP, the amount of frames is equally distributed among the GPUs available. This approach speeds up the total execution time whereas the time to process one frame remains the same. To have the benefits of this implementation, a large set of input images should be processed.

In order to evaluate the advantages of using multiple GPUs, the application was run using a set of 120 input images. The results were compared with those obtained using the single GPU implementation and the original not-optimized CPU version. The results are shown in Table IV where it can be seen the significant speed up achieved by the multiple GPUs approach.

TABLE IV
COMPARISON OF THE DIFFERENT IMPLEMENTATIONS

Version	Execution time (ms)	Speedup	Total speedup
CPU	466368	--	--
Single GPU	4190	108x	108x
4 GPUs	1270	3.3x	357x

Given that the maximum number of GPUs that can be used with OpenMP is four, the number of GPUs was varied from 1 to 4 to measure the different performance in each case. For these measurements the set of 120 images was used. Figure 4 illustrates the results obtained.

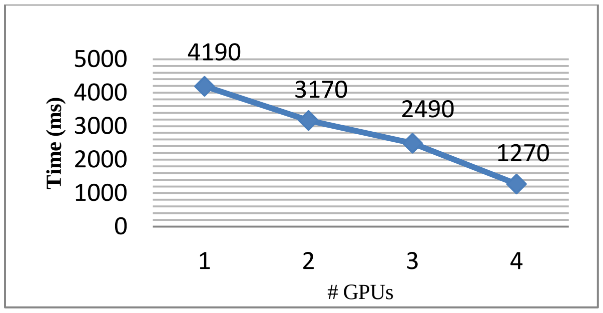

Figure 4. Time performance using different # of GPUs

5. Conclusions

This paper has presented the mapping to GPU of the recognition algorithm of a vision system based on a Convolutional Neural Network (CNN). The starting point of this work was the C-implementation of the algorithm executed in CPU. In order to evaluate this algorithm, the application was profiled. Based on the results, it was decided which parts could be mapped to the GPU. The next steps were aimed to improve the performance of the application using a single GPU. A total speed up of *108 times* was achieved, being now *33ms* the execution time per 1280x720 HD frame. Further improvements in the total execution time regarding a large amount of frames that can be processed were accomplished, with the use of OpenMP to split the workload among the GPUs available in the cluster. The total to process 120 HD frames is *1270ms, 357 times* faster than using the original CPU implementation.

This application has highly potential for parallelism techniques due to the fact that the algorithm is based on independent pixel computations. Moreover, the CNN structure offers the possibility to exploit parallelism in the computation of the outmaps. For these reasons, the application is considered suitable for GPU mapping. The results obtained confirm the previous assumption. The speed up achieved allows the application to satisfy the real-time requirements. It is also suitable for multiple GPUs as the division of the workload is done in terms of frames. The more GPUs available will result in a lower total time.

6. Future work

The performance of the application could be improved by applying different techniques. For instance, in the current implementation one of the drawbacks is that an image is read as many times as the number of connections with the next layer. The algorithm can be modified so as the image is read only once and be used by all the outmaps that require it. By implementing this approach, memory accesses would be reduced. Data locality could be improved by the use of smaller tiles when performing the convolution. Additionally, the use of half-precision floating-point instead of single-precision would result in a speedup of the application without a significant degradation of the accuracy. Lastly, the use of the maximum number of CPU processors in the node can be considered to as another option to exploit parallelism. Thus, CPU would process a number of frames concurrently.

References

[1] Peemen, M., Mesman, B., Corporaal, H.: Efficiency Optimization of Trainable Feature Extractors for a Consumer Platform. , in Proceedings of the 13th International Conference on Advanced Concepts for Intelligent Vision Systems (ACIVS'11), Ghent, Belgium, (2011)

[2] Nvidia Corporation, "NVIDA CUDA C Programming Guide 4.0",2011.