GPU programming
LeNet-1 Project Report

Dalmasso Luca
Computer Engineering, Embedded Systems master degree
Politecnico of Turin

01/07/2022

Contents

0.1 Introduction 1
0.2 Background 1
0.2.1 Convolution 2
0.2.2 Activation Function 2
0.2.3 Pooling Function 3
0.3 LeNet-1 Architecture 4
0.3.1 Description of the first hidden layer 5
0.3.2 All layers 6
0.3.3 Other comments about LeNet 6
0.4 Implementation 7
0.4.1 CPU implementation & serial benchmarks 7
0.4.2 GPU implementations 8
0.4.3 Naïve version benchmarks 8
0.4.4 Constant memory version benchmarks 9
0.4.5 Shared & Constant memory version benchmarks 10
0.5 Block Concurrency 11
0.6 References 13

List of Figures

1 Typical CNN 1
2 Sigmoid activation function 3
3 Anatomy of the LeNet-1 4
4 First convolutional layer, first hidden layer 5
5 Third convolutional layer, Fully connected to output layer 6
6 CPU performance profile 7
7 CPU vs GPU performances, version 1 9
8 CPU vs GPU performances, version 2 10
9 CPU vs GPU performances, version 3 11
10 GPU acceleration as a function of blocks running in the kernel . 12

List of Tables

1 Naïve version Global memory profile 8
2 Naïve version performances 8
3 Constant memory version Global memory profile 9
4 Constant memory performances 9
5 Constant + Shared version, Global memory profile 10
6 Constant + Shared version, Shared memory profile 11
7 Constant memory performances 11

Abstract

The project consisted in developing the LeNet-1 Convolutional Neural Network with the aim of speeding it up in a GPU through the CUDA programming language.
In this report are presented some example of possible implementations of the CNN algorithm on the GPU that start from a very naïve version and shows what are the optimisations that were implemented in order to get better performances and better usage of the GPU resources.

0.1 Introduction

In the domain of Machine Learning, Convolutional Neural Networks (CNNs) are particular class of Depp Learning algorithms specifically designed to process pixel data and so widely used for image classification and object recognition. The LeNet-1 is a CNN architecture introduced in 1998 by Yann LeCun and other researchers as an attempt to classify 2D images, in particular with the purpose of recognising handwritten digits in ZIP codes.

0.2 Background

A very general architecture of a CNN is the one showed in Figure 1 where it is possible to see that, from a general point of view, a CNN is a sequence of layers. Each layer is composed of a set of feature maps and each feature map is processed by some mathematical operations and transformed into another feature map and so on. The number of layers as well as the number of feature maps of each layer and their size really depends on what kind of image you would like to classify and what kind of accuracy you need and also how many resources you have.
Layers in a CNN are divided into three categories:

1. Input Layer: is the first layer of a CNN and it is the input image that needs to be classified. An example of input layer can be the robot in the Figure 1 .

2. Hidden layer: any middle layer in a CNN is also called hidden layer and they are a set of feature maps. They are called hidden because their inputs and outputs are masked by convolutions , activation function and pooling or subsampling functions. Each hidden layer is used as an input of the next hidden layer until the last one, which is further processed to become the output layer. An example of hidden layer can be the first four feature maps of the Figure 1 that are obtained starting from the input image convolved with four different filters.

3. Output layer: contains the results of the classification.

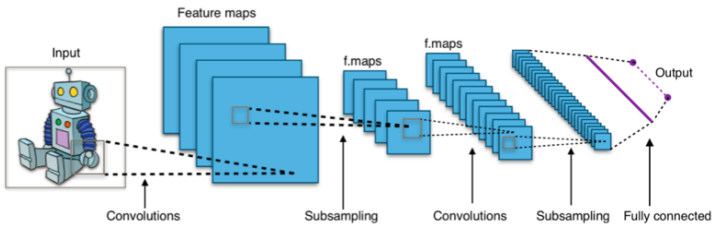

Figure 1: Typical CNN

The three operations mentioned before (convolution, activation function and pooling) are the set of mathematical ingredients that are used in any CNN to process and classify images.

0.2.1 Convolution

given an image $I \epsilon R^{n \cdot m}$ and a filter $K \epsilon R^{k \cdot k}$ the resulting pixel of this operation $I_{x,y} * K$ (which is a 2D convolution) is the result of the following equation:

$$Ic_{(x,y)} = I_{x,y} * K = \sum_{u=-k/2}^{k/2} \sum_{v=-k/2}^{k/2} K_{u,v} \cdot I_{x-u,y-v} \tag{1}$$

The equation (1) requires to flip the filter K and to center it on the target pixel $I_{x,y}$, if you consider the filter as a set of weights then the resulting pixel $Ic_{(x,y)}$ is just a weighted sum of the input pixel $I_{x,y}$ and the surrounding ones.
Due to the fact that in a CNN the filter can be considered already inverted and not all input pixels are convoluted but only those in the central portion of the image where the filter doesn't go out of border:

$$x \in [k/2, m - k/2], y \in [k/2, n - k/2]$$

the operation can be simplified as a standard dot product:

$$Ic_{(x,y)} = \sum_{u=0}^{k} \sum_{v=0}^{k} K_{u,v} \cdot I_{x+u,y+v} \tag{2}$$

Just to be clear, the following example makes use of the equation (1) where for simplicity the filter K is the identity filter (in this case the inverted version K' is the same K):

$$I \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \\ 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 & 36 \end{pmatrix} * K \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \\ 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 & 36 \end{pmatrix}$$

The following example makes use of the equation (2) with the same I, K:

$$I \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \\ 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 & 36 \end{pmatrix} * K \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 8 & 9 & 10 & 11 \\ 14 & 15 & 16 & 17 \\ 20 & 21 & 22 & 23 \\ 26 & 27 & 28 & 29 \end{pmatrix}$$

This is the formula used to compute the output matrix's size Os starting from input matrix's size Is and filter's size Ks:

$$Os = Is - Ks + 1 \tag{3}$$

0.2.2 Activation Function

The activation function, sometimes also called transfer function, defines how the weighted sum of the input (so the result of a convolution) is transformed

into an output value.
There are different types of activation functions and usually many of them are nonlinear functions, and the choice of which activation function to use can really affects the performances and the capabilities of a neural network.
Typically every pixel after being convoluted is also activated, this operation is performed by the convolutional layers.
Without going into many details, the LeNet-1 makes use of the Sigmoid function which is the following nonlinear function:

$$S(x) = \frac{1}{1 + e^{-x}} \tag{4}$$

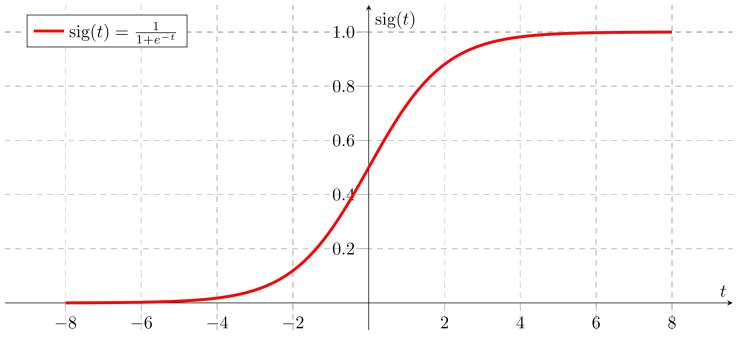

Figure 2: Sigmoid activation function

0.2.3 Pooling Function

Another widely used operation in a CNN is the Pooling function also called Subsampling function.
Pooling is a sample-based discretisation process with the aim of down-sample an input representation (image, hidden-layer, generic matrix, etc) and so reducing it's dimensionality.
The basic procedure of pooling is very similar to the convolution operation. You select a filter, you slide it over the input representation and you obtain a new output representation.
In this case the most commonly used filter size is 2x2 and the filter is slid over the input pixels with a stride of 2.

There are several approaches to pooling, the most commonly used are the following two:

- Max Pooling:
  the filter simply selects the maximum pixel value in the 2x2 region
  example:

$$\begin{pmatrix} 8 & 9 & 10 & 11 \\ 14 & 15 & 16 & 17 \\ 20 & 21 & 22 & 23 \\ 26 & 27 & 28 & 29 \end{pmatrix} \mapsto \begin{pmatrix} 15 & 17 \\ 27 & 29 \end{pmatrix}$$

- Average Pooling:
  in this case the pooling works by calculating the average value of the pixels in the 2x2 region
  example:

$$\begin{pmatrix} 8 & 9 & 10 & 11 \\ 14 & 15 & 16 & 17 \\ 20 & 21 & 22 & 23 \\ 26 & 27 & 28 & 29 \end{pmatrix} \mapsto \begin{pmatrix} 11.5 & 13.5 \\ 23.5 & 25.5 \end{pmatrix}$$

Right now, many implementations use max pooling because it is less expensive from a computationally point of view and it can helps in speeding up the CNN.

0.3 LeNet-1 Architecture

Now that all the mathematical basics were briefly introduced the discussion can finally introduce the forward propagation algorithm of the LeNet-1 CNN.

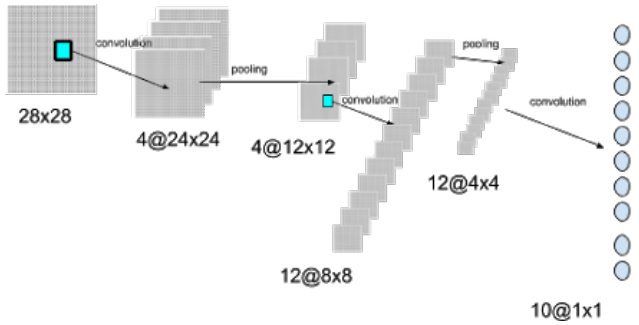

Figure 3: Anatomy of the LeNet-1

From Figure 3 it is possible to observe that, starting from the input layer, the overall architecture is composed of 3 convolutional layers and 2 pooling layers for a total amount of 4 hidden layers and a final output layer.

1. Input Layer:
   consists of a 28x28 image that represents the first 784 input neurons.

2. First Hidden layer:
   First convolutional layer composed of four 24x24 feature maps that forms a group of 2304 neurons.

3. Second Hidden layer:
   First average pooling layer composed of four 12x12 feature maps that forms a group of 576 neurons.

4. Third Hidden layer:
   Second convolutional layer composed of twelve 8x8 feature maps that forms a group of 768 neurons.

5. Fourth Hidden layer:
   Second and last average pooling layer composed of twelve 4x4 feature maps that forms a group of 192 neurons.

6. Output layer:
   Contains the result of the classification of the input image. This CNN as said in the Introduction is meant to be used to classify digits between 0-9, so the output layer contains the classification on a 10x1 array.

0.3.1 Description of the first hidden layer

Starting from the input image, four filters K1, K2, K3, K4 of size 5x5 (25 weights each) are convoluted to generate the four feature maps.
The size of each feature map is 24x24 and can be easily computed with the equation (3) where Is=28 and Ks=5.
The visual representation of the first convolutional layer is showed in Figure 4.
The values of the first layer are instead given by the following equation:

$$F^s_{x,y} = \sigma(bias + \sum_{u=0}^{4}\sum_{v=0}^{4} K^s_{u,v} \cdot I_{x+u,y+v}) \tag{5}$$

Where $F^s_{x,y}$ is the (x,y) neuron of the s-th feature, σ is the Sigmoid activation function in the equation (4) that transform the result of convolution (equation (2)) plus a bias term.
The remaining parts of the equation are the term $K^s_{u,v}$ that is the (u,v) weight of the s-th filter, and finally $I_{x+u,y+v}$ is the (x+u,y+v) pixel of the input image.

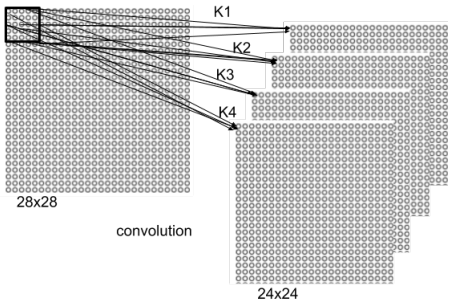

Figure 4: First convolutional layer, first hidden layer

0.3.2 All layers

Stated that LeNet-1 has an overall of three convolutional layers and two pooling layers, the network can be described by the following flow:

$$INPUT \mapsto C1 \mapsto S1 \mapsto C2 \mapsto S2 \mapsto C3 \mapsto OUTPUT$$

Once the input has been convoluted into four feature maps, each feature map is pooled into 4 corresponding feature maps of 12x12 each.
In the next layer, the third hidden layer, the convolution is done again by the same equation (5) but this time there are only 3 filters 5x5. Since the three filters are convoluted on each of the four feature maps, this multiplies the amount of feature maps of the next layer to 12. Now the new layer is pooled again and by doing that the last hidden layer is created and the size is reduced from 768 to 192 neurons that correspond to the last 12 4x4 features in the hidden layers. To fully connect the last hidden layer to the 10 output neurons one last convolution is used, in particular each Out_i can be computed using this formula:

$$Out_i = \sigma(bias + \sum_{j=0}^{11} K_i * F_j) \tag{6}$$

where σ is the usual Sigmoid, K_i is the i-th filter over the total 10 of the last convolutional layer and F_j is the j-th feature map of the last hidden layer. The following image can help to understand how the last hidden layer is fully connected to the ouput:

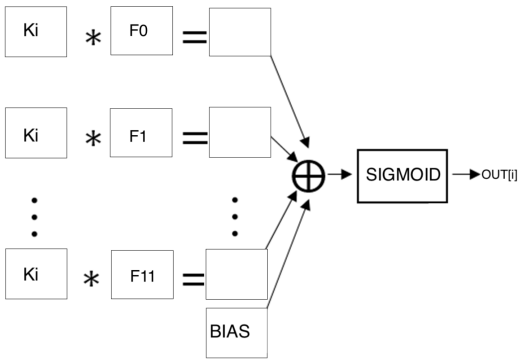

Figure 5: Third convolutional layer, Fully connected to output layer

0.3.3 Other comments about LeNet

Even if the architecture of the LeNet-1 CNN is a very simple example without too many hidden layers and features it is important to underline that every forward propagation (every image classification) requires a big set of operations that are sequences of multiplications, additions, divisions and exponentials. All the mentioned operations repeated for every neuron inside the network will require hundreds of thousands of floating point operation and this render the problem heavy from a computational point of view!

0.4 Implementation

The board used for implementing and testing the forward propagation algorithm is the NVIDIA Jetson Nano.
First of all the forward propagation was implemented on the CPU (host) to have a reference benchmark of the algorithm executed serially, then the code was ported on the GPU (device).
Concerning the device implementation there are three solution proposed:

1. heavy usage of global memory, all the weights are saved in global memory and all feature maps are processed in global memory.
2. filters are moved in constant memory space, in such a way to reduce the global memory usage.
3. soft usage of global memory, all computations in shared memory, filters in constant memory.

0.4.1 CPU implementation & serial benchmarks

The host code is very simple, it serially executes this series of transformation already discussed in the previous sections:

$$Image28x28 \mapsto C1_{4x24x24} \mapsto S1_{4x12x12} \mapsto C2_{12x8x8} \mapsto S2_{12x4x4} \mapsto C3_{10x1x1} \mapsto OUTPUT$$

The following graph is showing the execution times for 10 consecutive forward propagations randomly selected among 50000 recorded executions of the algorithm with random images and random weights:

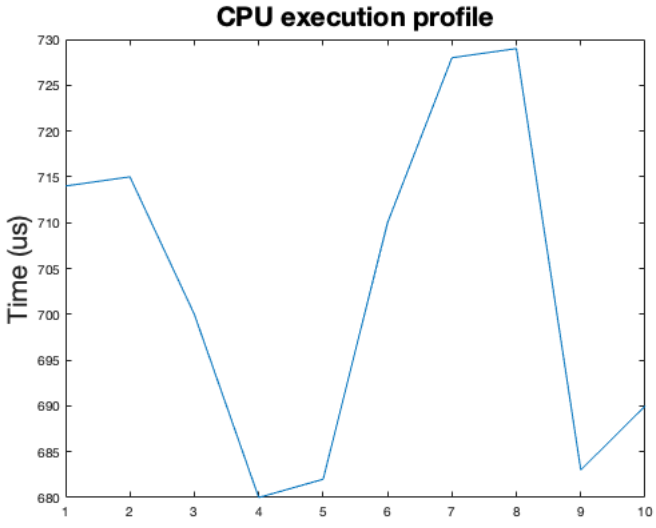

Figure 6: CPU performance profile

0.4.2 GPU implementations

Considering that the propagation through the layers is sequential it is still possible to accelerate the overall performances of the network by speeding-up the internal operations of each layer.
Once the features of the layer i are ready, all the features of the layer $i+1$ can be processed in parallel because each neuron can be computed independently from all others in the same layer.
As introduced before, the implementation starts from a very naïve version that processes data using heavily the global memory, the goal was to move as much as possible the data in shared memory in such a way to reduce as much as possible the amount of global memory transactions.
The algorithm is executed by a single block of 28x28 threads and it is the same in all the 3 versions, the only differences are in how the data are arranged in the memories.

0.4.3 Naïve version benchmarks

All filters are saved in global memory, all features are saved in global memory. Data arranged like this will produce a lot of transactions because each feature, once computed, needs to be stored in global memory in such a way that can be used to feed the next layer when the propagation moves forward, and also the efficiency of the transaction will not be optimal because of the strided accesses by convolution and pooling operations.
The following metrics were collected by using the profiler:

Kernel Name	**load efficiency**	**store efficiency**	**load transactions**	**store transactions**
deviceForwardV1	49.17%	76.15%	124610	632

Table 1: Naïve version Global memory profile

In the following table are shown the performances of the kernel collected again with the profiler.
Note that not only the pure kernel execution time is considered but also the CUDA APIs calls are considered in the final execution time. The API calls are the following ones: $memcpyHostToDevice$, $memcpyDeviceToHost$ and the $cudaDeviceSynchronize$.
The total execution time is then the time the processor needs to wait in order to be able to use the GPU results.

Kernel Name	**Avg. kernel exec time**	**Avg. APIs overhead**	**Avg. total exec time**
deviceForwardV1	370us	540us	900us

Table 2: Naïve version performances

Is possible to see that even if the algorithm is executed faster on the GPU (370us against 700us in Figure 6) the overall execution time is expected to be on average higher than the CPU one.

The following picture shows a comparison between CPU and Naïve version performances (again using 10 random execution times among a collection of 50000):

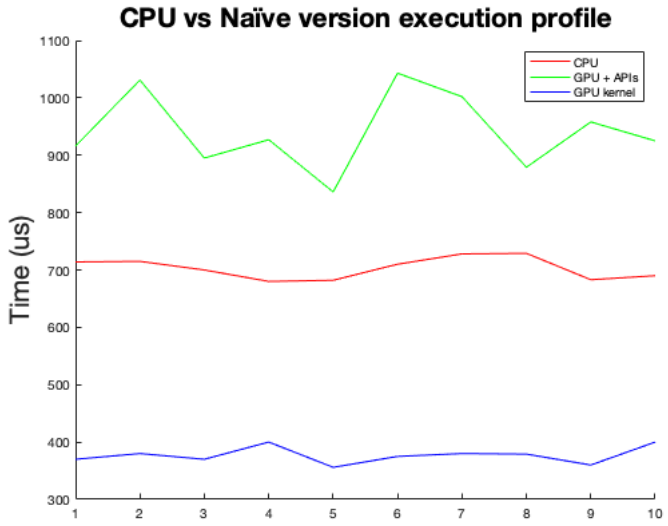

Figure 7: CPU vs GPU performances, version 1

0.4.4 Constant memory version benchmarks

Considering that once a network is trained all the filters are maintained constant, is possible to take advantage of the Constant Memory.
The only data that needs to be moved in and out form the global memory are then only the features, the profiler already shows a significant reduction of the global transactions:

Kernel Name	**load efficiency**	**store efficiency**	**load transactions**	**store transactions**
deviceForwardV2	59.45%	76.15%	64610	632

Table 3: Constant memory version Global memory profile

The performances are now expected to be better, in fact the profiler shows the following results:

Kernel Name	**Avg. kernel exec time**	**Avg. APIs overhead**	**Avg. total exec time**
deviceForwardV2	283us	370us	650us

Table 4: Constant memory performances

The new version is roughly 30% faster than the Naïve one but even if overall execution time is not much better that the CPU's.
The results confirms that the approach of reducing the global memory usage is

the right one!

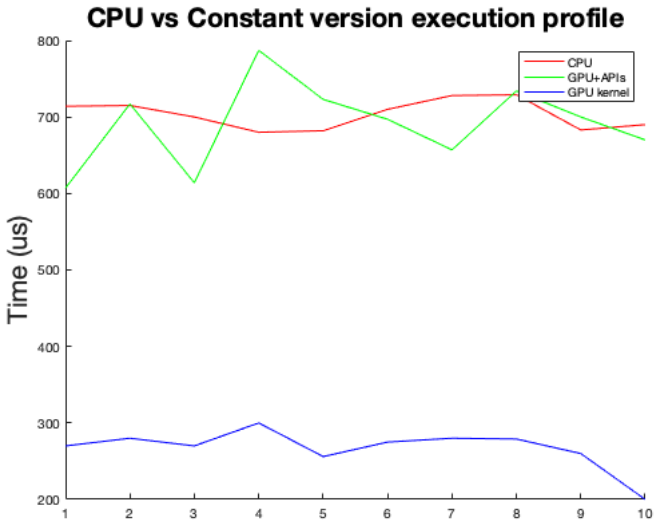

Figure 8: CPU vs GPU performances, version 2

0.4.5 Shared & Constant memory version benchmarks

This is the final implementation of the algorithm, in the previous one the filters have been removed from global memory, now considering that:

1. there is only a single block running on the GPU, this means that all threads can easily work together using the same data in shared memory.
2. all the features together occupy a total amount of: $(4x24x24 + 4x12x12 + 12x8x8 + 12x4x4) \cdot 4 \approx 16Kbytes$, our device has 49 Kbytes of shared memory per block so the entire features can be stored in shared memory.
3. the kernel just needs the input image and needs to provide to the CPU just the output layer.

all the computations can be done using just the shared + constant memory, the overall usage of global memory is negligible because only transactions issued are for loading the input layer in shared memory and storing the output layer in global memory:

Kernel Name	**load efficiency**	**store efficiency**	**load transactions**	**store transactions**
deviceForwardV3	100.00%	62.50%	394	2

Table 5: Constant + Shared version, Global memory profile

ld. transactions/request	st. transactions/request	ld. bank conflicts	st. bank conflicts
1.026161	1.023599	0	8

Table 6: Constant + Shared version, Shared memory profile

Also the shared memory usage showed good results in terms of bank conflicts, the following metrics and events have been profiled:

The performances showed overall better results:

Kernel Name	Avg. kernel exec time	Avg. APIs overhead	Avg. total exec time
deviceForwardV3	172us	230us	400us

Table 7: Constant memory performances

The new version is expected to roughly 2x faster that the CPU one.

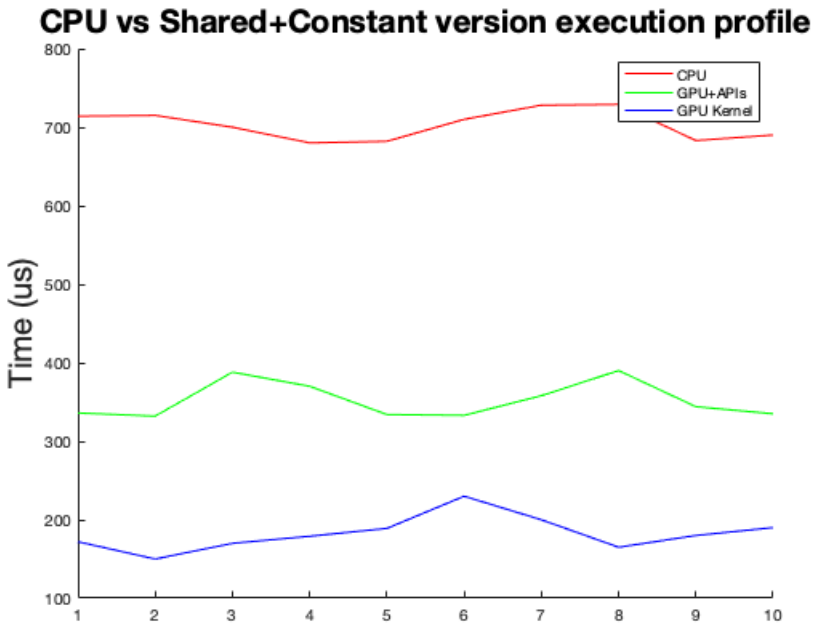

Figure 9: CPU vs GPU performances, version 3

0.5 Block Concurrency

The different implementations can show how is possible to speed up an algorithm by just take advantage of the different memory hierarchies in the device. The issue behind the implementations presented before is connected to the device occupancy, which is a very important metric that indicates how many warps

are active out of the total amount supported by the multiprocessor(s) on the GPU. Even if the final version is optimised from a memory point of view, the overall occupancy is low, less then 40%, because the more the propagation moves across the layers the less threads will work.
Another aspect that has not been taken into consideration is the possibility to use multiple blocks for accelerating more the algorithm.
In order to solve these two problems the idea is to have multiple blocks running the same code on different images, like having multiple instances of the CNN running "in parallel" on the device where basically each block is used to classify an image.
In the previous implementations the kernel is a single block that receives as input an image, runs the forward propagation and saves back the result, this is repeated *iteratively* for every image that the application wants to classify. In this new implementation the kernel receives as input the entire (or a part of it, depending on the total amount of memory) collection of images that the application wants to classify and each block is in charge of performing the forward propagation on one image.
Having multiple blocks running can not only lead to higher performances but also increase the overall device occupancy. The following picture shows what are the speed up of this implementation with respect to the CPU, as a function of blocks (images) running in the GPU:

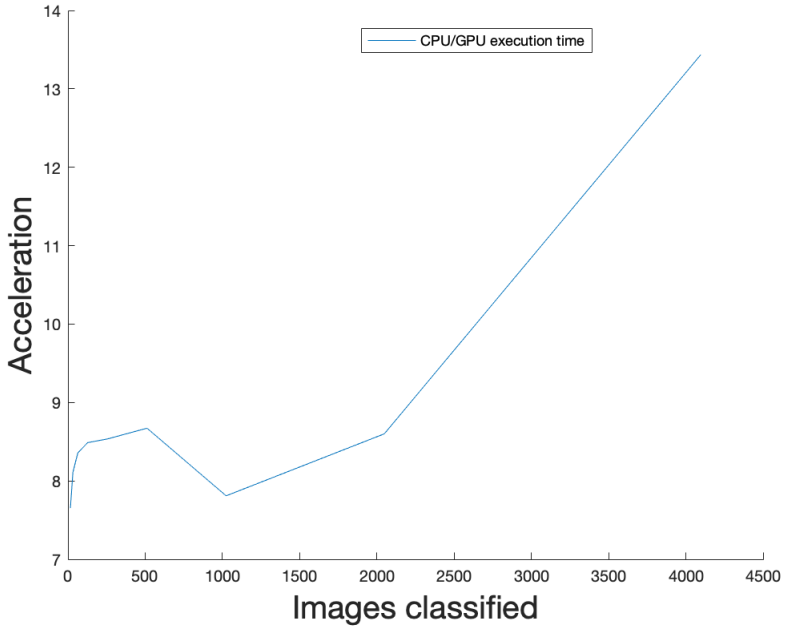

Figure 10: GPU acceleration as a function of blocks running in the kernel

The profiler also reported that the occupancy has settled around 70%, starting from the previous $35 \sim 40$ %.

0.6 References

1. Anatomy of the LeNet-1 Neural Network: https://acodez.in/anatomy-of-the-lenet-1-neural-network/

2. Review LeNet-1: https://sh-tsang.medium.com/paper-brief-review-of-lenet-1-lenet-4-lenet-5-boosted-lenet-4-image-classification-1f5f809dbf17

3. Anatomy of the LeNet-1 CNN: https://www.skyradar.com/blog/anatomy-of-the-lenet-1-convolutional-network-and-how-it-can-be-used-in-order-to-classify-radar-data

4. Professional CUDA C Programming