v. 4, n. 29, 2024

Journal of Engineering Research

Acceptance date: 27/12/2024
Submission date: 02/12/2024

TRAINING OF A DEEP NEURAL NETWORK USING CUDA PROGRAMMING

Patricia Pérez-Romero
National Polytechnic Institute, Cidetec, Mexico
https://orcid.org/0000-0003-3395-6239

Miguel Hernández Bolaños
National Polytechnic Institute, Cidetec, Mexico
https://orcid.org/0000-0002-5622-8747

Abstract: Deep neural networks have been successfully applied in the fields of computer vision, automatic image recognition and speech, among others. An important part of their architecture is the use of convolution operations that perform feature filtering at different levels of abstraction during the network training phase. This paper proposes the use of GPU graphics acceleration units to reduce the computational load based on its SIMT (*Single Instruction Multiple Thread*) architecture that exploits the intrinsic data parallelism of these applications.
Keywords: CUDA programming, graphics processing unit, training phase, neural network, spatial convolution, algorithm.

INTRODUCTION

According to Fujimoto, N. (2008), the internal structure of the deep neural network to carry out the recognition of objects in images can be made up of a series of operations that are performed sequentially, such as: spatial convolution, maxpooling and the different activation functions, see Table 1.

Considering that much of the computation in the neural network is consumed by the convolutional operations, since these are carried out for each element of the image, it was decided to improve performance by using a graphics processing unit (GPU), which allows through massive parallel processing to reduce the computation time required in the training phase of the network.

Layer	Function	Parameters
1	Convolution Space	16 characteristics Kernel 5x5
	Tanh	
	MaxPooling	Kernel 2x2
2	Convolution Space	256 features Kernel 5x5
	Tanh	
	MaxPooling	Kernel 2x2
3	Linear	Output 128
	Tanh	
	Linear	8 classes

Table 1. Characteristics of the deep neural network model

CONVOLUTION

Before defining what the convolution process is, it is necessary to clarify some concepts. In the same way that an image is represented as a two-dimensional matrix, there is a structure called a spatial filter. This structure is nothing more than a matrix of N x M, whose values are called filter coefficients. For the purpose of facilitating the calculations, N and M will always be considered to be odd, although in reality they can have arbitrary dimensions as expressed by Karimi et al (2010).

Figure 1 shows the image to be filtered and the respective filter. In this example, the portion of the image around the pixel to be filtered is taken and multiplied point by point to find the resulting matrix; then the sum of the values of this matrix is calculated and the value of the filtered pixel is obtained whose value is in the same position of the filtered image as it was in the original image. This process is known as *convolution*, which is the process by which a filter is moved over an image and the sum of the products at each position is calculated.

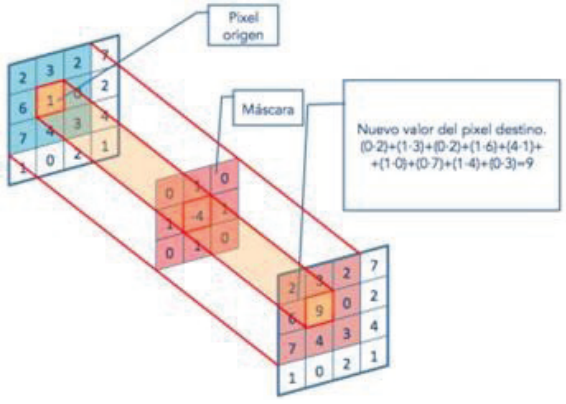


Figure 1. Application of a 3x3 filter to a pixel of an image (Own elaboration).

METHODOLOGY AND DEVELOPMENT

The methodology adopted for the work consisted of the following steps:

1. Definition of the development environment.
2. Handling of external libraries.
3. Time measurements and benchmarking.
4. Proposed algorithm.
5. Shared memory management.
6. Test stage.

DEVELOPMENT ENVIRONMENT

For the development of the programs, the compiler provided by NVIDIA CUDA Toolkit within the *Linux* development environment *NVIDIA CUDA Compiler Driver* (NVCC) was used. It is worth mentioning that the code produced can be compiled under other architectures and operating systems as long as it has hardware with CUDA architecture or a CUDA emulator, according to NVIDIA, (2010). The characteristics of the computer on which we worked are the following:

- Processor: Intel Core i5 M520 2.4 GHz.
- RAM memory: 6 GB.
- Video Card: NVIDIA GeForce 330 MB.
- With dedicated video memory: 1 GB.
- NVIDIA Driver: The video card driver is version 270.41.06 for 64-bit Linux operating system.

It is possible to estimate the theoretical performance of the board we worked on. It has 6 multiprocessors, with 8 processors each, for a total of 48 processing units. Each of these has an internal clock of 1.265 GHz, giving a theoretical performance of 60.72 Gflops. This is 60.72 billion floating point operations per second.

EXTERNAL LIBRARIES

Only one library external to the standard C libraries was used, and that was NVIDIA's *cuda.h*, as it allows the use of directives and functions to take full advantage of the video card. These range from memory copies between host and device to thread synchronization.

TIME MEASUREMENTS AND BENCHMARKING

NVIDIA Compute Visual Profiler version 3.2 is used to accurately measure kernel execution times and data copies from CPU memory to device memory and vice versa. This tool allows determining the execution times of kernels and memory copies, as well as determining the percentage of use of the video card's computational capacity, the amount of memory used for each type of graphics card memory, among other measurements that are useful for the optimization of the implemented programs.

PROPOSED ALGORITHM

In the case of a color image, three matrices are required, that is, there is a representation for the red, green and blue planes respectively. Likewise, the filters must correspond one for each plane, as shown in figure 2 .

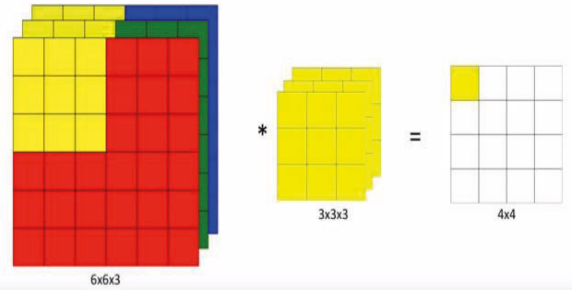


Figure 2. Color image convolution (Prepared by the authors)

A thread is the execution unit in which part of the data is processed and which can synchronize and share information with other threads. A thread in CUDA is similar to threa-

ds in any multitasking operating system, with the difference that the creation and synchronization of these threads does not involve a high computational cost, as it does in the case of these operating systems. The main reason lies in the execution model of the CUDA architecture, SIMT (Single Instruction Multiple Thread) in which the same instruction is executed by several threads simultaneously, so the cost of execution becomes almost unimportant, i.e. the architecture of the video card is optimized to perform tasks of this type. For more detail it should be clarified that the threads are grouped into blocks, which are an abstraction to denote an arrangement (up to 3 dimensions) of these. These blocks are grouped in turn, in a grid, which is an array (also up to 3 dimensions) of blocks, see figure 3.

```
_global_____void convolve(        . . .        ) {

for( ){        //Rojo, Verde, Azul
for( )         //filas máscara
for( )         //columnas máscara
suma parcial
Sustituir pixel por la suma sum = 0;
}
```

Figure 3. Pseudocode of the implementation (Prepared by me).

SHARED MEMORY MANAGEMENT

For local memory accesses without cache or global memory, latencies of 400 to 600 clock cycles can be expected. For this reason, memory access optimization is considered a high priority when optimizing a kernel, NVIDIA (2010). Shared memory should be used whenever possible and in cases where memory capacities (16 KB for 1.X devices and 48 KB for 2.0 and above devices) are exceeded by the amount of data, it is advisable to reorder and group memory accesses or consider the use of texture memory if the application allows it.

This can be seen in Figure 4, which also shows the memory model of the CUDA architecture.

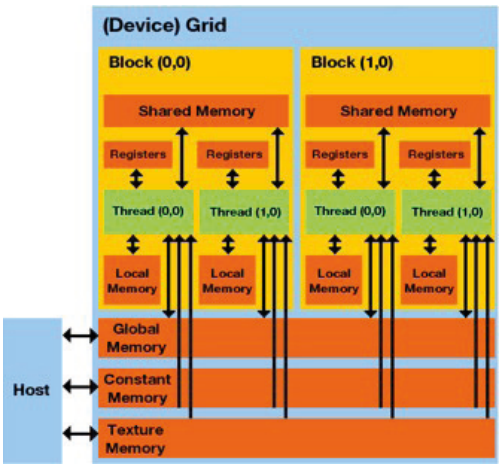
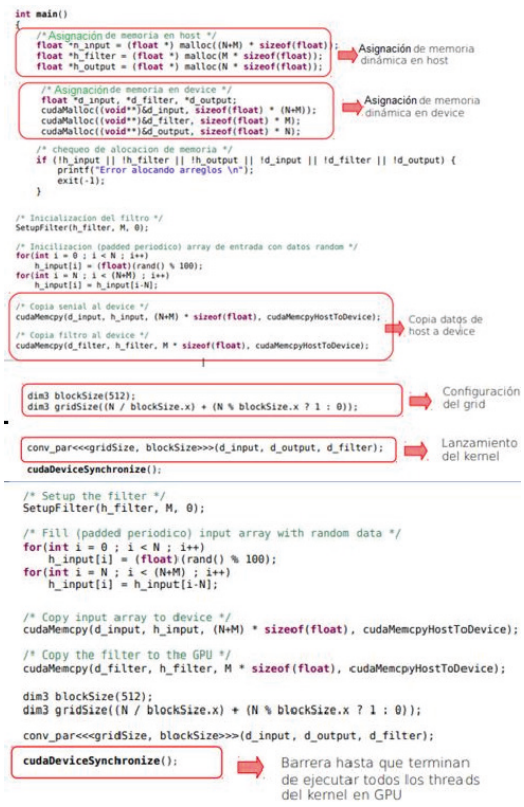


Figure 4. Available memories in the CUDA architecture. (Original image taken from http://www.ks.uiuc.edu)

KERNEL EXECUTION (TEST STAGE)

It can be seen that a kernel is declared just as a C function is declared with the difference that the global declaration is used to indicate that it will be executed on the GPU. Additionally, the kernel parameters are specified between the braces <<<...>>>. The elaborated code is shown in figure 5.



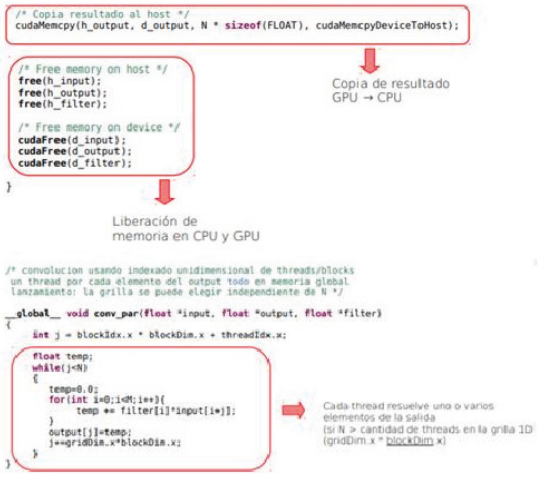


Figure 5. CUDA code of the application (Own elaboration).

RESULTS

Four different image sizes were chosen, being of square proportions. Their sizes in ascending order are: 2800x2800px, 5600x5600px, 11200x11200px, 22400x22400px.

In addition, there were 3 different types of masks that apply the convolution algorithm of sizes 3x3, 5x5 and 7x7.

Each of the possible combinations between image and filter were executed in each version of the algorithm*: sequential algorithm, CUDA algorithm, CUDA algorithm using shared memory*. The above can be seen in Table 3, which additionally shows the number of threads used in each case.

Filter	Image size (square)	Sequential	CUDA	CUDA Shared Memory	Threads
3x3	2800	6100	1,98	7,97	7840000
3x3	5600	24128	9,00	54,88	31360000
3x3	11200	100096	35,98	201,78	125440000
3x3	22400	404130	139,98	775,97	265932800
5x5	2800	15948	5,01	14,87	7840000
5x5	5600	59248	20,97	92,90	31360000
5x5	11200	259414	83,89	330,87	125440000
5x5	22400	1290042	297,01	1299,85	265932800
7x7	2800	31997	9,90	24,96	7840000
7x7	5600	168782	38,94	142,10	31360000
7x7	11200	723987	145,86	536,78	125440000
7x7	22400	3085853	547,67	1430,75	265574400

Average time (ms) of the 4 runs launched by each version of the algorithm, mask size and image size.

Figure 6 shows the different execution times obtained for the different versions of the algorithm. It can be seen that the scaling is logarithmic, so the times improve exponentially as the image size increases. It is also evident that the times obtained with the version of CUDA that uses shared memory were not as efficient as expected, these results are explained due to the type of data used in each pixel of the image, since the image has been represented using bytes to maximize the capacity of the GPU and this caused collisions in the access to the memory of the device. On the other hand, the time cost of bringing the data from global memory to local memory is not compensated by the operations performed on each data, so the times of the shared memory version are not the best.

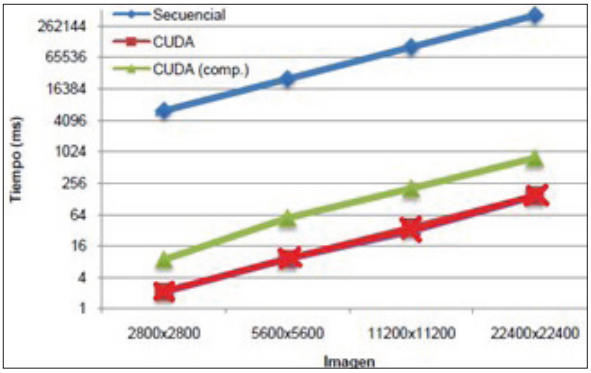


Figure 6. Times obtained with respect to different image sizes with a 3x3 mask in logarithmic scale base 2.

CONCLUSIONS

In the study carried out, it can be seen how the convolution operation is highly parallelizable allowing an allocation of GPU resources that accept to exploit the properties of both shared memory and global memory, with which high-growth linear gains can be obtained for the processing of any image. It was observed that the larger the image size, the longer the time required for its processing, however, with respect to the sequential version, the parallel version manifests high performance for any size, taking into account that it is a highly parallelizable algorithm and that it can be easily adapted to the parallel architecture of a GPU. It is also clear that the times obtained with the version of CUDA using shared memory is not as good as expected, results that are explained due to the type of data used in each pixel of the image, since the image has been represented using bytes to maximize the capacity of the GPU and this causes collisions in the access to the memory of the device. On the other hand, the time cost of bringing the data from global memory to local memory is not compensated by the operations performed on each piece of data, so the times of the shared memory version are not the best.

REFERENCES

Fujimoto, N. (2008). Dense Matrix-Vector Multiplication on the CUDA Architecture. *Parallel Processing Letters*, Vol. 18(4), pp. 511-530.

Karimi, K.; Dickson, N. G. & Hamze, F. A. (2010). Performance Comparison of CUDA and OpenCL. *CoRR*, Vol. abs/1005.2581.

Kirk, D. B. & Hwu, W. (2010).Programming Massively Parallel Processors: A Hands-on Approach. *Morgan Kaufmann*.

NVIDIA. (2010). CUDA C Best Practices Guide.

Version 3.2. *NVIDIA*.

Wolfe, M. (2010). Understanding the CUDA Data Parallel Threading Model: A Primer. The Portland Group *Technical News*.