Cuda Kernel Image Processing

Francesco Villi
francesco.villi@stud.unifi.it

Abstract

This project utilizes CUDA, a parallel computing platform and programming model developed by NVIDIA, to perform image processing tasks on a GPU. By implementing image processing algorithms as CUDA kernels, the project is able to significantly accelerate the processing time compared to a traditional CPU-based implementation. The goal of the project is to demonstrate the effectiveness of using CUDA for image processing and to show the potential for processing high-resolution images.

1. Introduction

The developed application can be used to perform a convolution between kernel and image. Kernel image processing refers to a technique used in image processing where an image is convolved with a small matrix called a kernel. The kernel is a fixed matrix of coefficients, which is used to modify the pixels in the image. The process of convolution applies the kernel to each pixel of the image by multiplying the pixel value with the corresponding value in the kernel, and then summing up the results. It can be written as:

$$g(x,y) = (f * g)(x,y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} k(i,j) f(x-i, y-j)$$

where f is the image, k the kernel, and g the output image. This process is applied to each pixel in the image, effectively transforming it. We can rotate by 180° the kernel k and compute the convolution using this formula that is easier to implement:

$$g(x,y) = (f * g)(x,y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} k'(i,j) f(x+i, y+j)$$

All kernels in the program are symmetric so we don't need rotation.

The output has a size smaller than the input due to we can't apply convolution in the borders; to solve that we have some techniques computationally expansive as increasing the kernel and image size:

- *zero-padding* where we add 0s around the border of the image
- *pixel mirroring* where we use the nearest border pixel as padding
- *pixel replication* where we copy border pixels.

The kernel matrix can be designed to perform different types of image processing operations such as image sharpening, blurring, edge detection.

Kernel-based image processing is a powerful technique that can be used to modify images in a variety of ways, but it has a high computational cost which can be time-consuming on a CPU. CUDA-based image processing allows to perform kernel-based image processing on a GPU, greatly reducing the time required for the operation. In particular this is an embarrassingly parallel problem because each output value depends on the source image and kernel coefficients so we don't need particular synchronization methods.

1.1. CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA for general purpose computing on NVIDIA GPUs (Graphics Processing Units). CUDA enables developers to leverage the massive parallelism of NVIDIA GPUs to perform complex computations faster than on a CPU alone.

In CUDA, there are different types of memories that can be used for storing data and performing computation. These include:

- *Global memory*: This is the largest memory available on the device and it is shared among all threads. It's the slowest available on GPU

- *Shared memory*: This is a smaller memory that is shared among threads within a block.It's the faster than global and constant memory.
- *Constant memory*: This is a read-only memory that is accessible by all threads and can be used to store constant data. It's faster then global.

In CUDA, there is also a distinction between host memory and device memory:

- *Host memory* refers to the memory that is available on the host (i.e. the CPU)
- *device memory* refers to the memory that is available on the GPU

Data must be transferred between host and device memory in order to be processed on the GPU.

2. Structure

The structure of the code is:

- **ConvolutionCPU** implements static functions to perform a convolution in parallel and sequential way.
- **PaddingImage** implements static function to perform padding according to the type padding (zero, pixel mirroring, pixel replication)
- **Image** class that contains image size and image values. It doesn't read a real image but, given height and width, it creates a random matrix with values from 0 to 255.
- **Kernel** class that contains all possible kernels that can be used such as edge detection (3x3), sharpen (3x3), constant blur (3x3) and gaussian blur (3x3,5x5,7x7).
- **ConvolutionCPU** implements static functions to perform a convolution using GPU. There are 3 possibilities: use only gloabl memory, global and constant memory or only shared memory.

2.1. CPU version

In this version we have 2 main steps:

1. The function gets in input the image, kernel and type padding. At the begin it computes the padding according to the type and returning a new image padded.
2. The second step is about performing convolution between the image padded and kernel returning the edited image.

I implemented a parallel version and a sequential version in order to compare them. The parallel version uses directive yielded by OpenMP.

2.2. GPU version

In the GPU version we have 3 different types of implementation:

1. Use only *global memory*: I allocate space in the global memory for kernel, original image and output image; then I compute the output image calculating the convolution between image and kernel reading all values from the global memory and storing the new values in the output image.
2. Use *constant memory*: I allocate space in the global memory for original image and output image whereas the kernel has been stored in the constant memory. Now to compute the convolution I have to read image values from the global memory and kernel values from the constant memory that is faster.
3. Use *shared memory*: I allocate space in global memory for original image,kernel and output image; then for each block I have to load a piece of the input image due to an high resolution image can't be stored in a single block. To do that we have to copy for each block that represents a part of the total image also the elements ”outside the border” (i.e. if a block is 16x16 and the kernel size is 3x3 the sub-image that we copy is 18x18 because we have to consider the pixel near the border to calculate correctly the output value).

2.3. Other implementation details

Every time before executing the convolution we create the padded image and then we compute the convolution. All kernels are integers and only after the sum we apply the normalization factor to achieve the max precision.

3. Environment settings

All tests have been run in a PC with intel core intel i7-12700H, 16GB of RAM DDR5, GPU NVIDIA GeForce RTX 3050 with 4GB of memory. The compiler used is Visual Studio Community version 17.4.3 and NVIDIA toolkit 12.0, OpenMP v2 has been used to parallelize the CPU code. All tests have been run in relase mode.

4. Comparison

Firstly we'll compare the CPU version and GPU version to check who can elaborate faster an image; then, we will analyze three different versions of a GPU implementations that utilize different types of memory. These variations include a version that uses only global memory, another that uses constant memory for the kernel, and a third that uses only shared memory. We will be examining the performance, and overall effectiveness of each version to determine the best option for the given application. By comparing these different types of memory, we aim to provide insight into the advantages and disadvantages of each. We will evaluate their performance under varying image sizes, block size and kernel size. We will incrementally increase one parameter at a time and we measured the time it takes for each version to complete the task, i.e. load a matrix into memory, complete the convolution and return the new elaborated image. This will provide insight into how each type of memory handles larger workloads and how it may impact the overall performance of the system. The kernel used for tests is a gaussian filter.

5. Results and Discussion

5.1. CPU

The parallel version uses the max amount of threads (20) and to balance the workload I used a dynamic scheduling because the i7-12000H has 4 efficient cores and 8 high performance cores.

Size	Sequential	Parallel
1280x720	0.006941	0.003161
1920x1080	0.025705	0.003122
2560x1440	0.034740	0.003746
8120x4568	0.245991	0.056058

Table 1. 3x3 kernel processing speed using CPU(values in seconds)

Size	Sequential	Parallel
1280x720	0.022662	0.003011
1920x1080	0.051387	0.006721
2560x1440	0.034740	0.003746
8120x4568	0.577013	0.084652

Table 2. 5x5 kernel processing speed using CPU(values in seconds)

We observe in the CPU case that the parallelization improves the general performance, especially increasing the kernel's size.
In particular:

1. Using a 3x3 kernel (tab. 1) the speed up is low (2) when the image and kernel size are small whereas speed up grows until 10 for the median sizes and then it stabilizes around 7 for high values.

Size	Sequential	Parallel
1280x720	0.042273	0.005194
1920x1080	0.085983	0.012568
2560x1440	0.128911	0.019509
8120x4568	1.0025	0.128868

Table 3. 7x7 kernel processing speed using CPU(values in seconds)

2. Using a 5x5 kernel (tab. 2) the speed up is 6 for small image, then it reaches 10 for the median sizes and it stabilizes around 7 again.
3. Using a 7x7 kernel (tab. 3) the speed up is constantly around 9

We notice that in all cases the parallel version performs better than the sequential. The unique case takes in exam that performs worst is the first case in table 1 due to the additional time and resources required to coordinate and manage parallel operations, as opposed to running a single operation sequentially (overhead). This includes tasks such as dividing a problem into smaller subproblems, communicating data between processing units.
In all other cases the improvements are significant but not enough to elaborate a millions of images in rational time. So we need to introduce GPU that has thousands of small processing cores that can work together to perform the same operation on multiple pieces of data at the same time.

5.2. GPU

We measured computational time for each implementation according to the memory used.

Size	Global	Constant	Shared
1280x720	$1.9e-05$	$3.77e-05$	$2.6e-05$
1920x1080	$3.77e-05$	$4.04e-05$	$2.68e-05$
2560x1440	$2.26e-05$	$2.32e-05$	$2.61e-05$
8120x4568	$1.52e-04$	$1.01e-04$	$6.51e-05$

Table 4. 3x3 kernel processing speed using GPU and various memories without data transfer(values in seconds)

Size	Global	Constant	Shared
1280x720	$4.36e-05$	$4.2e-05$	$2.45e-05$
1920x1080	$3.68e-05$	$3.82e-05$	$3.89e-05$
2560x1440	$3.15e-05$	$2.89e-05$	$2.24e-05$
8120x4568	$3.35e-05$	$3.39e-05$	$2.99e-05$

Table 5. 5x5 kernel processing speed using GPU and various memories without data transfer(values in seconds)

Size	Global	Constant	Shared
1280x720	$2.85e-05$	$3.53e-05$	$3.24e-05$
1920x1080	$3.2e-05$	$3.23e-05$	$2.83e-05$
2560x1440	$3.45e-05$	$3.75e-05$	$3.02e-05$
8120x4568	$3.7e-05$	$4.07e-05$	$3.45e-05$

Table 6. 7x7 kernel processing speed using GPU and various memories without data transfer(values in seconds)

We note that in general from table 4, 5 and 6 that the shared memory version has the best performance while the constant memory version is a little better than the global memory version. The difference in absolute terms isn't much between all three versions but we have to remember that we will apply convolution to millions of images and so a small speed improvement for a single image can mean a big speed improvement for elaborating millions of images.
Comparing these results with the results obtained using a CPU we notice a speed up ranging from 100 to 3500; bigger is the image and kernel, and greater is the speed up.
Using a GPUs, one of the key challenges is transferring data between the host (i.e. the CPU) and the device (i.e. the GPU). This process, known as data transfer or data movement, can have a significant impact on the performance of the system. Considering also the time needed for the data transfer the results in terms of time become:

Size	Global	Constant	Shared
1280x720	0.112076	0.0009342	0.0009855
1920x1080	0.0017446	0.0013102	0.0014553
2560x1440	0.0027054	0.0022875	0.0026714
8120x4568	0.0162387	0.0178803	0.0198277

Table 7. 3x3 kernel processing speed using GPU and various memories with data transfer(values in seconds)

Size	Global	Constant	Shared
1280x720	0.0010611	0.0009583	0.0010926
1920x1080	0.0017095	0.0016164	0.0017338
2560x1440	0.0028429	0.0025105	0.0033403
8120x4568	0.0215968	0.0204387	0.034397

Table 8. 5x5 kernel processing speed using GPU and various memories with data transfer(values in seconds)

We note that the issue is the data transfer, in particular in shared memory we have to allocate also the memory in each block for the partial image and the kernel.
So, the bottleneck in this process is often the transfer of data from the CPU to the GPU, as the CPU must send the data to the GPU before computation can begin and retrieve the results once computation is complete. This transfer can take a significant amount of time, limiting the overall speedup gained by using a GPU. Despite this bottleneck, the use of GPUs can still result in significant performance gains in certain types of computations like this one.

Size	Global	Constant	Shared
1280x720	0.0011533	0.0010757	0.0012704
1920x1080	0.0019003	0.0019199	0.0024117
2560x1440	0.003266	0.0033103	0.0038257
8120x4568	0.0277996	0.0207564	0.0395129

Table 9. 7x7 kernel processing speed using GPU and various memories with data transfer(values in seconds)

So, considering the data transfer we have to update the previous deductions.
Analyzing tab 7, 8 and 9 we observe for each size the shared memory returns the worst result in terms of time; the best outcome has been reached by constant memory, in particular it's more time efficient as kernel size increases. In shared memory we have to load a partial image and the kernel so we must allocate memory for each block and it is time consuming.
Re-calculating the speed up, considering the data transfer, it ranges between 3.5 and 6.3; we notice the data transfer from/to device/host is very onerous, in particular as the size of the image and kernel increases.

6. Conclusion

In conclusion, this project has demonstrated that GPU's are a superior choice for kernel image processing compared to CPU's. The parallel processing capabilities of GPU's allow for faster processing times, even when faced with the added overhead of transferring data from the host to the device and viceversa. The results of this project show a significant performance boost when using GPU's for kernel image processing, making them a valuable tool for tasks such as image filtering and feature extraction.
Overall, it is clear that GPU's are a powerful and efficient choice for image processing tasks.