Final-Term Project : Kernel Image Processing

Hulduz Djanbekov
hulduz.djanbekov@etu.univ-nantes.fr

Abstract

Image processing often involves applying kernels to modify image data for purposes such as blurring, sharpening, or detecting edges. These tasks can benefit greatly from GPU parallelization to enhance performance. This study explores a comparison between a sequential method and a GPU-accelerated approach using CUDA, NVIDIA's platform for parallel computing. I implemented a convolution operation in CUDA and compared its performance against two alternatives: an optimized sequential method provided by OpenCV and a simple, unoptimized Python implementation. The results demonstrate that the CUDA implementation achieves remarkable speedups over the sequential methods, particularly when processing larger image sizes.

1. Introduction

Kernel image processing is a method where a small matrix, referred to as a kernel or filter, is applied to an image to perform various operations like blurring, sharpening, edge detection, or other transformations by using convolution.

You can find the code of my project here: GitHub Repository

1.1. Mathematical tools for kernel image processing

Mathematically, the convolution of two continuous functions $f(t)$ and $g(t)$ is defined as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) \cdot g(t - \tau)\, d\tau$$

For two discrete functions $f[n]$ and $g[n]$, the convolution is given by:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m] \cdot g[n - m]$$

In the context of image processing, where $I(x, y)$ is the image and $K(m, n)$ is the kernel, the two-dimensional discrete convolution is defined as:

$$G(x, y) = (I*K)(x, y) = \sum_{m=-a}^{a} \sum_{n=-b}^{b} I(x-m, y-n) \cdot K(m, n)$$

where $G(x, y)$ is the resulting image, and a and b represent the dimensions of the kernel K.

1.2. Kernel Image Processing

Kernel image processing involves moving a kernel (a small matrix of weights) over an image, where the kernel's values are multiplied by the corresponding image pixel values in the overlapping region. The products are then summed, and the result is assigned to the corresponding pixel in a new image. As shown in Figure 1, this operation is repeated for each pixel in the image, altering the original image based on the kernel's pattern. It's important to note that the image's border will be affected unless edge handling techniques are applied.

1.3. Simple convolution algorithm

Convolution is a process used in image processing where a small matrix called a kernel is applied to each pixel in the image to modify it. Here's a simplified breakdown of how the algorithm works:

```
For each row in input image:
For each pixel in image row:
     set acc to 0
     For each row in kernel:
       For each element in kernel
            row:
          If element position
              matches pixel
              position then:
               multiply element
                   value by pixel
                   value
               add result to acc
          End If
     End For
     set output image pixel to
         acc
```

1.3.1 Explanation of this algorithm

1. **Process each row of the input image:** The algorithm starts by looking at each row of pixels in the image one at a time.

2. **Process each pixel in the row:** For each pixel, the following steps are applied to calculate its new value.

   (a) **Initialize an accumulator:** An accumulator (denoted as `acc`) is set to 0. This will store the result of the calculations for each pixel.

   (b) **Iterate through the kernel rows:** The kernel is a small matrix that is used to adjust the pixel values. The algorithm moves through each row of the kernel.

   (c) **Process each kernel element:** For each element of the kernel, the algorithm checks if the position of the kernel element matches the position of the pixel in the image.

   (d) **Apply the kernel element:** If the positions match, the kernel element value is multiplied by the corresponding pixel value, and the result is added to the accumulator.

   (e) **Update the output pixel:** Once all kernel values have been applied to a pixel, the accumulated value is assigned to the corresponding pixel in the output image.

This process is repeated for every pixel in the image, ultimately transforming the input image according to the kernel's pattern.

1.4. Applications

Convolution, also known as kernel image processing, is used for various tasks like blurring, sharpening, and edge detection. It plays a crucial role not only in image manipulation but also in artificial intelligence, particularly in Convolutional Neural Networks (CNNs). CNNs utilize convolutional layers to automatically learn and extract hierarchical features from images, which allows for advanced pattern recognition.

1.5. Parallelization potential

Image convolution is naturally parallelizable because the computation of each output pixel depends solely on the input image and kernel, and is confined to a small neighborhood of input pixels. This feature enables the simultaneous processing of multiple pixels, making it ideal for parallel execution. As a result, pixel-level parallelization is straightforward, and hardware acceleration can take advantage of this to achieve substantial computational speedup. Furthermore, due to the localized nature of the computation, spatial decomposition can be utilized to share memory between threads within the GPU.

2. Implementation

This script conducts benchmarking to measure and compare the execution times of various

convolution techniques applied to images. Below is a breakdown of each section of the code.

Imports and Setup Several libraries are imported in this script:

- `os`, `csv`, `cv2`, `numpy`: These libraries are used for managing files, manipulating images, and performing numerical computations.
- Convolution modules (`OpenCVConvolution`, `CudaConvolution`, `SimpleConvolution`): These are custom-built modules that implement different convolution methods, such as OpenCV for CPU, CUDA for GPU acceleration, and a basic CPU-based method.
- Utility functions (`resize_images_incrementally`, `generate_output_path`): These functions handle resizing images in incremental steps and generating output paths for saving processed images.

Generating the Gaussian Kernel The function `generate_gaussian_kernel` creates a 2D Gaussian kernel with a specified size and standard deviation (`sigma`). It first generates a 1D Gaussian kernel using OpenCV's `getGaussianKernel` function, and then produces the 2D kernel by taking the outer product of the 1D kernel.

Setting Up Convolution Methods The `setup_convolution_methods` function initializes each convolution method (`OpenCVConvolution`, `CudaConvolution`, `SimpleConvolution`) with the Gaussian kernel. It then returns a dictionary where each method is mapped to its corresponding object.

Benchmarking the Convolution Methods The `perform_benchmark` function benchmarks the execution times of different convolution methods applied to a set of images. The steps performed include:

- For every image in the `images` list, the execution time for each convolution technique (OpenCV, CUDA, and Simple) is measured.
- The results are saved in a CSV file (`results_file`), which includes the execution times (in seconds) for each method.
- The script outputs the execution times for each method and compares the CUDA method to the others by showing how much faster it is. The "Simple" method is handled by skipping it if its execution time exceeds a given threshold.

2.1. Sequential implementation

The SimpleConvolution class offers a Python-based implementation of kernel convolution, but it has slow performance. The convolve method applies convolution to a given image array by iterating over each pixel. It extracts the necessary image patch for each computation and multiplies it pixel by pixel, and color channel by color channel, with the convolution kernel. The result is then placed into the output image. While this method is functional, being written in an interpreted language significantly limits its performance compared to compiled algorithms.

In contrast, OpenCV's optimized sequential convolution, implemented in the OpenCVConvolution class, is much faster. This makes it an excellent candidate for testing our parallel implementation.

2.2. Parallel implementation

CUDA is highly efficient for convolution tasks because it can parallelize computations across thousands of GPU cores at the pixel level. With PyCuda, we can integrate a CUDA C kernel directly into our Python code, making the overhead of using an interpreted language like Python virtually negligible.

3. Results

All the approaches yielded the same results and successfully performed kernel image processing. However, the OpenCV method can apply the kernel even to the borders of the images by using fake pixels to multiply with the kernel elements. This feature is not implemented in the Python and CUDA versions, resulting in minor differences between the output images.

3.1. Testing methodology

I didn't measure the entire execution of the program, as certain steps, such as initialization, are not relevant and would unfairly penalize approaches that involve more Python code, particularly the CUDA implementation. Therefore, only the actual convolution process and the handling of input data and output retrieval were measured.

3.2. Hardware precision

The tests were conducted on a Lenovo Ideapad 3 laptop, running at maximum power with Linux. The hardware specifications are as follows:

- CPU: AMD Ryzen 5
- RAM: 8GB DDR4 Memory
- GPU: Integrated AMD Radeon Vega Graphics

The tests were performed using a Google kernel.

3.3. Tests results

Every approach was measured 10 times on one image in a large number of size variations, from a size of 100 pixels by 100 pixels, to a size of 4000 pixels by 4000 pixels. The averaged execution time was used to measure accurately every approach.

Image Size	OpenCV (ms)	CUDA (ms)	Python (ms)
100	0.30	0.20	90.50
500	5.40	0.70	2,800
1000	23.00	2.10	too long
2000	85.00	10.50	too long
3000	190.00	16.50	too long
4000	370.00	34.00	too long

Table 1. Performance comparison of different convolution methods.

We see that OpenCV and CUDA have almost the same execution time for very small images. It is proven by the fact that the computations are too simple to make any difference. But when the computation time get longer, the CUDA approach is very effective speedup spoken.

As expected, Python approach is extremely long, and is not able to compete with the other two approaches (it is stopped for images when image size ¿ 1000).

4. Conclusion

In conclusion, the tests evaluated the performance of OpenCV, CUDA, and pure Python methods for kernel-based image processing. The results underscored CUDA's exceptional efficiency, particularly with larger image sizes, even without additional optimizations. Performance could be further enhanced by leveraging shared memory within CUDA blocks for faster data access. Overall, the hardware-accelerated approach demonstrated its effectiveness for kernel image processing tasks.