Parallel Kernel Image Processing Report

Michael Bartoloni
Matricola 7152862

Abstract

The purpose of this project is an exercise in parallel programming techniques using the CUDA framework. The project features the development of two kernel processing methods for a dataset of 2K resolution images, first using a sequential C++ implementation, then using an optimized parallel version using CUDA, and a comparison between sequential and parallel execution times.

1. Introduction

Kernel image processing is a fundamental technique in image processing and computer vision, used for tasks such as filtering, edge detection, and noise reduction.
A separable kernel and a nonseparable kernel were tested in a sequential and in a highly parallel environment. The purpose is to test the improvement in performance with different complexities of the algorithm. The benchmark is performed on a selection of 30 images from the DIV2K dataset. The parallel benchmark measures the execution of the kernels for increasing block sizes, comparing them with the sequential time.

1.1. The Dataset

DIV2K is a collection of images in 2K resolution and was featured in various image processing competitions. The images are in RGB format, but since the use of three channels only adds redundancy that requires management, which is really not interesting for the objectives of this exercise, we load them in a grayscale format. This allows for the simplest data structure for this task, i.e. an array of pixel values.

1.2. Kernel Image Processing

Kernel image processing involves applying a convolution filter to an image. In our case, the convolution is performed using a 9×9 kernel. The convolution operation involves sliding the kernel over the image and computing a weighted sum of pixel values within the kernel window. Although separable kernels can reduce the computational cost by breaking a 2D filter into two 1D filters, the nonseparable version processes all 81 pixels per convolution. This exercise provides insight into the performance gains achievable when parallelizing computationally intensive tasks on GPUs. In [Figure 1], a visualization of the task is shown.

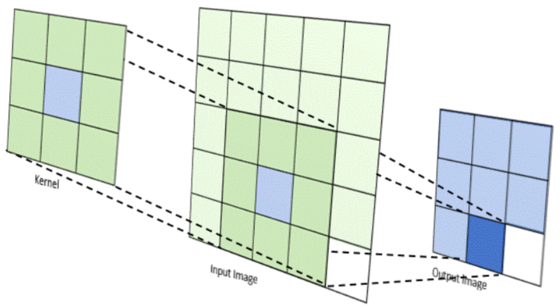

Figure 1. Convolution visualization

1.3. The Algorithm

The algorithm performs a convolution operation, a data-intensive process in which each output pixel is computed as a weighted sum of a neighborhood of input pixels. This requires management of memory accesses, particularly because the convolution window must be applied across the entire image, including its borders.
The separable kernel applies a gaussian blur to the image, and the nonseparable kernel applies an image detection filter.
The choice for handling the boundary conditions is to clamp the pixel indices to valid ranges. This is not the only strategy for boundaries, but for the types of kernels implemented (mainly because of the edge detection kernel), it is a simple and effective solution.

2. Sequential Benchmark

The sequential benchmark begins by loading all the test images from the dataset. Once the images are loaded, the separable kernel is applied across each image, and the elapsed time is recorded. Following this, the nonseparable

Separable Kernel	Nonseparable Kernel
0.7578 s	5.6661 s

Table 1. Sequential execution times (s)

kernel is executed on the same set of images, with its execution time measured similarly.
To mitigate variability in performance measurements, each kernel is executed 100 times, and the average elapsed time is computed for a more stable benchmark.

2.1. Results

The results [Table 1] show a marked difference in execution times between the separable and nonseparable kernel implementations. As expected, the more complex nonseparable kernel runs slower, since it requires more computation in comparison with the separable kernel.
These results provide a clear baseline for assessing the performance improvements achieved by parallelization.

3. Parallel Benchmark

The parallel benchmark features an optimized version of the algorithm, executed on an NVIDIA GeForce RTX 3060 (12GB). In this setup, the algorithm is benchmarked using increasing block sizes, ranging from 32 to 1024 threads per block. We do not need to test beyond this range because the SM can manage a maximum of 1536 threads.

3.1. Optimizations

3.1.1 Constant Memory and Shared Tiling

The CUDA implementation leverages constant memory to store the convolution kernel, ensuring that these values are cached and quickly accessible by all threads. To further improve performance, shared memory is used to implement tiling: sections of the image are loaded into a shared buffer that includes extra border pixels necessary for the convolution window. This indexing enables each thread to efficiently compute the weighted sum over its local neighborhood using the kernel stored in constant memory. The shared memory loading phase is the only part requiring explicit synchronization among threads, meaning that massive speedups from great numbers of threads can take place in this setting.

3.1.2 Indexing threads

To account for variable image sizes, the logical view of the image is padded with the block's dimensions, to ensure that the correct number of threads is spawned. In this design, each thread calculates an output tile and is responsible for allocating more than one pixel to the shared memory. The way this is achieved is by calculating an enlargement factor given by the number of pixels in the tile plus the halo for the tile. Then, each thread in the block calculates the current pixel position in the shared tile and loads the actual pixel, based on a local index coalescing the threads' accesses to optimize the distribution of workload.

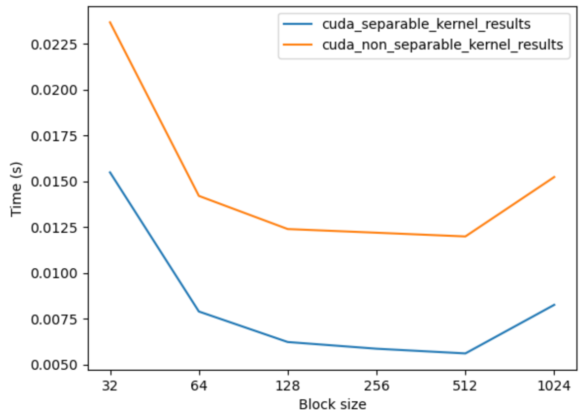

Figure 2. Execution times for separable and nonseparable kernels with increasing block sizes.

3.1.3 Synchronization

Since shared memory is used for tiling, it is essential to synchronize threads after loading the data into the shared buffer. A barrier is used to ensure that all threads in a block have completed the memory load before any thread begins the convolution computation. Once this barrier is passed, each thread operates on its own shared copy of the required data, eliminating the need for further synchronization during the computation phase.

3.2. Results

In [Figure 2], execution times are plotted for both the separable and nonseparable kernels as the block size increases. The separable kernel reaches its best performance at 512 threads per block. In contrast, the nonseparable kernel, while also showing improved performance at higher block sizes, consistently requires more time, with the best performance also observed at 512 threads per block.

4. Conclusions

The parallel benchmark results clearly demonstrate that the use of GPUs for kernel image processing leads to substantial performance improvements over a sequential approach. The highly parallel nature of the GPU enables a dramatic reduction in execution times, even though the speedup is sublinear relative to the increase in threads per block.

Block Size	Separable Time (s)	Nonseparable Time (s)
32	0.01548	0.02367
64	0.00789	0.01420
128	0.00623	0.01239
256	0.00586	0.01219
512	0.00560	0.01199
1024	0.00825	0.01523

Table 2. Execution times for separable and nonseparable kernels with varying block sizes.

The kernels incur different computational overheads, due to their different complexity in processing the convolution: even if a thread elaborates only a pixel, the number of operations is linear for the separable kernel and quadratic for the nonseparable kernel (with respect to the kernel's width). Despite these differences, both implementations show massive overall improvements when executed in parallel, and their differences are mitigated.
Looking at the execution times, the worst result is obtained using a block size of 32. This was expected, as 32 threads is basically a warp, and blocks work better when they have more than one warp.
The second worst result, using 1024 threads per block, is due to the SM's maximum number of threads, maxed out at 1536. This means that each streaming multiprocessor is underutilized in this setting Since limited synchronization is needed, using massively parallel machines greatly enhances performance, showing how aggressive multithreading can lead to huge improvements for tasks requiring less communication between threads, slicing computation times to a fraction.