Parallel Computing course

a.a. 2021–2022

Comparative Analysis of Kernel Image Processing using Sequential C++ and Parallel CUDA implementation modes

Leonardo Paroli
for the 2021-2022 Parallel Computing course by Prof.Bertini

Abstract. This study aims to assess the execution time of a Gaussian image processing kernel for both image blurring and enhancement, comparing sequential and parallel CUDA implementations, exploiting techniques for faster parallel memory loading, processing and thread organization to accelerate the convolution process and comparing results.

Key words. Parallel Computing - CUDA - Sequential coding - Kernel Image Processing - Gaussian Filtering - Kernel

1. Context

Kernel image processing is a fundamental technique that plays a pivotal role in various image manipulation tasks. One common application involves employing a Gaussian kernel for blurring images: This technique, often referred to as Gaussian blur, serves as a cornerstone for enhancing images by reducing noise and sharpness, thereby achieving a smoother and more visually appealing result. The Gaussian kernel, characterized by its bell-shaped curve, assigns higher weights to pixels closer to the center and gradually decreases weights for pixels farther away. This weighting scheme ensures that the blurred pixel value is an average of its neighbors, with a greater emphasis on those in close proximity.
The sequential implementation of the Kernel Image Processing technique serves as the foundation for building the parallel version, providing a a benchmark for evaluating the speedup brought about in terms of performance and scalability.
Parallel implementations of the Kernel Image Processing technique in CUDA mode focus instead on optimizing execution by utilizing the parallel processing units of GPUs.

2. Test structure and hardware

The objective is to measure the execution time of applying a Gaussian image processing kernel for image blurring and enhancement in both sequential and parallel modes. The first step for the comparative analysis is to develop the algorithm in different

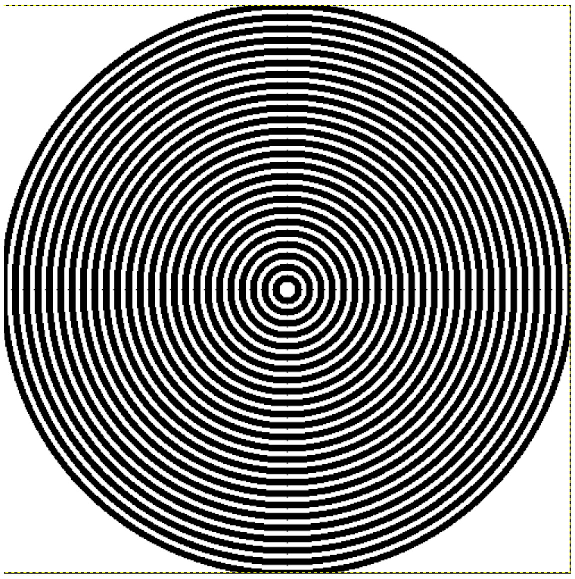

Fig. 1. Original image, 500x500 pixels.

modes and verify its effectiveness, ensuring correct convolution application to the image (as can be seen in figures 1, 2).

To begin, a Python script is used to generate four images in PPM format, each with increasing dimensions: 500x500, 1000x1000, 2000x2000, and 4000x4000 pixels, representing a sequence of pixelated concentric circles.hese images will be used to compare algorithm performances and demonstrate their effectiveness.
It's possible to visualize the images before and after applying the convolution to ensure the blurring operation is correctly executed.
The test also involves sequentially increasing the size of the kernel from 7x7 to 15x15 while maintaining an odd kernel size. The convolution execution is performed three times for each image-kernel pair across various dimensions, allowing the calculation of the average operation time.

The measurement of execution time starts when the algorithm receives the original image as an array of pixels and begins its iteration. It continues until the result is returned, which is a new array of pixels modified by the Gaussian kernel convolution.

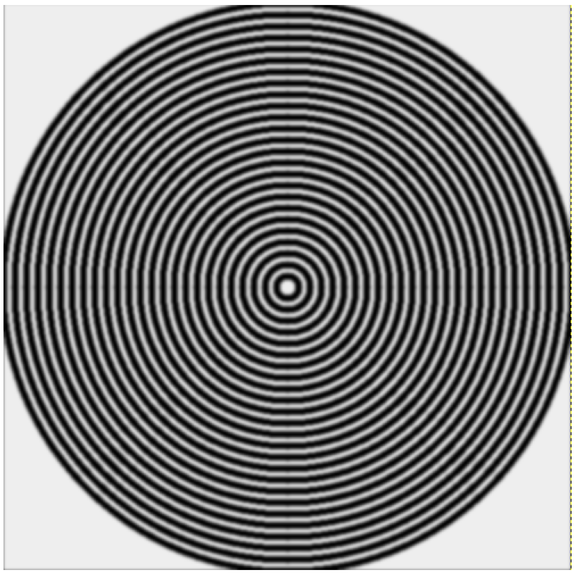

Fig. 2. Blurred image, resulting from the parallel, CUDA implementation of the guassian blur filtering, 500x500 pixels.

The measurement is performed using std::chrono::high_resolution_clock, and the result is reported in milliseconds. Once the algorithm is executed on the generated dataset, in both sequential and parallel modes, the results are calculated and recorded in a text file named ”results.txt”. Additionally, the speedup values are calculated and recorded in a separate text file named ”speedup.txt”.

The analysis was conducted on a laptop with the following hardware specifications:

- Processor:AMD Ryzen 7 5700U 8 cores
- L2 Cache :4MB
- L3 Cache :8MB
- 16 GB RAM, 3200 MHz
- GPU : NVIDIA GeForce RTX 3050 Laptop GPU , 4 GB VRAM, 8.6 computational power on the NVIDIA scale.

All tests have been developed and ran on Windows 10, using Clion-2023 prior to the integration of the Visual Studio 2022 toolchain for building, compiling, debugging and running the program.
Tests ran in parallel mode have been conducted through the use of Nvidia CUDA Toolkit 12.1.

3. Sequential implementation

The next step was writing the actual code for the Gaussian blur operation in sequential mode, briefly summarized as follows:

- The code sequentially applies Gaussian blur to an input image represented as an array of structures Pixel, with each Pixel struct composed of the red,green and blue channels.
- The Gaussian kernel is created, which serves as a weight matrix for the convolution to apply to the image. The kernel's size, an odd integer, determines the extent of the blur effect.
- A nested loop constructs the kernel by iteratively computing Gaussian distribution values for each element, taking into account its position within the kernel, while considering a sigma standard deviation, controlling the spread of the distribution and by doing so influencing the amount of blurring applied.

The formula for the Gaussian distribution in two dimensions is given by:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{1}$$

The following formula has been used to calculate the value of the Gaussian kernel at position (i, j) within a kernel of size $K \times K$:

$$kernel[i][j] = \frac{1}{2\pi\sigma^2} e^{-\frac{(i-kernelSize/2)^2+(j-kernelSize/2)^2}{2\sigma^2}} \tag{2}$$

- The computed kernel values are then normalized by dividing the sum of all kernel values, ensuring that the overall pixel intensity remains consistent, as follows:

$$kernel[i][j] = \frac{kernel[i][j]}{\sum_{i,j} kernel[i][j]} \tag{3}$$

- A new image array is created to store the resulting blurred image. Two nested loops iterate over each pixel in the original image. For each pixel, an inner loop traverses the Gaussian kernel, using its values to compute a weighted sum of neighboring pixels' color values, using accumulators for each channel.
  The implementation also checks whether the calculated offset positions fall within the image dimensions to avoid accessing out-of-bounds pixels while applying the kernel values.
- The accumulated color values are used to create a new pixel value for the blurred image. This process is repeated for every pixel in the image, ensuring that each pixel's color values are influenced by its neighboring pixels.
- The resulting blurred image is then returned as the function's output.

In the next chapter, we'll see how this sequential implementation serves as a foundational step for parallelization, where optimizations and thread-level parallelism are introduced to enhance the performance of the Gaussian blur operation.

4. CUDA Implementation

The purpose of this code is to accelerate the Gaussian filtering process compared to the sequential CPU implementation by exploiting the computational capabilities of the streaming processor.
The idea is to apply the convolution through multiple blocks of threads executing in parallel over the streaming processor, each working on a portion of the image, with careful attention on the thread organization, as to allow for efficient parallel computation and to avoid potential race conditions.

To do this, the firs step was to apply a parallel conversion of the Array of Structures Pixel into a Structure of Arrays, as to properly exploit the memory burst when loading and moving data from the main memory.
Once the data is ready, it is loaded through CUDA memory allocation and copy method, and then used in the kernel over a specific number of blocks composed of a number of threads (the exact number of thread blocks and the density of threads is key for achieving more or less speedup when comparing against the sequential mode, and in the best results we've used 256 threads per block),
The kernel used to apply the convolution first identifies the current thread's position in the grid of threads, also determining the global indices for the current thread based on the block and thread indices, effectively determining which pixels of the image the thread should process.

The kernel then declares a block-level shared memory array sharedMemory, which is used to efficiently store portions of the image, allowing for faster memory access compared to global memory.
Since the shared memory is relatively smaller than global memory, the image pixels (red, green, blue channels) are loaded into the shared memory by each thread by using a ”tiling” approach, where threads load data in smaller ”tiles”.
The kernel also makes use of batch loading, where the batches variable determines how many tiles are required to cover the kernel window (kernelSize x kernelSize).
When loading data into shared memory, the code takes care of handling edge cases: if a pixel lies outside the image boundaries, the code ensures that a valid pixel is fetched. This avoids artifacts caused by attempting to access pixels beyond the image borders.In our case, the ”extend” method was used,

which simply repeates the last valid pixel for half of the gaussian kernel size in each direction.

After loading data into shared memory, each thread performs the convolution operation for the pixel it corresponds to in the output image. It loops over the Gaussian kernel, computing the weighted sum of neighboring pixel values and the corresponding kernel coefficients.

The computed convolutions for the red, green, and blue channels are stored in the output image SoA arrays.

The __syncthreads() function has been used to ensure synchronization of all threads within a block and to prevent race conditions when accessing shared memory.

5. Results

Using the sequential implementation of the Gaussian Filtering as a baseline, tests have been conducted to determine the performance of the CUDA parallelized code.
The following tables help visualize the performance improvement when using all the precedently described parallization techniques: the first ones (1 and 2) have as values the mean execution time of the algorithm in the sequential and parallel modes, in milliseconds.

The resulting speedup in table 3 confirm that the parallel CUDA version can fully exploit threads to parallelize the process, growing more and more efficient as the number of pixels per image or the dimension of the kernel increase.

This is also due to the fact that many techniques like Tiling, Shared Memory and Batch Loading have been used to speed up the kernel when loading data for each thread, lowering the overhead by applying methods to reduce the impact of data transfers between the main memory and device memory, aswell as fully exploiting the fast and small shared memory within the GPU device instead of using the global memory.
The use of Structure of Arrays for the red,green and blue channels of the image's pixels also helped, exploiting memory bursts when loading data.

Table 1. Image Processing - Sequential execution results in milliseconds.

Image Size	Kernel size 7	Kernel size 9	Kernel size 11	Kernel size 13	Kernel size 15
Image500	(89)	(146.667)	(217.667)	(303.667)	(397.333)
Image1000	(353.667)	(586)	(869.667)	(1205.67)	(1602.33)
Image2000	(1420.33)	(2360.33)	(3501.33)	(4839)	(6455.67)
Image4000	(5691)	(9477.67)	(14011.3)	(19425)	(25812)

Table 2. Image Processing - Parallel execution results in milliseconds.

Image Size	**Kernel size 7**	**Kernel size 9**	**Kernel size 11**	**Kernel size 13**	**Kernel size 15**
Image500	(8)	(13)	(17.6667)	(25.6667)	(32)
Image1000	(23)	(38)	(51.3333)	(85.3333)	(110.667)
Image2000	(79.6667)	(142.667)	(195.667)	(332.333)	(433)
Image4000	(302.667)	(559)	(771.333)	(1322)	(1500.67)

Table 3. Image Processing - Speedup results

Image Size	**Kernel size 7**	**Kernel size 9**	**Kernel size 11**	**Kernel size 13**	**Kernel size 15**
Image500	(11.125)	(11.2821)	(12.3208)	(11.8312)	(12.4167)
Image1000	(15.3768)	(15.4211)	(16.9416)	(14.1289)	(14.4789)
Image2000	(17.8285)	(16.5444)	(17.8944)	(14.5607)	(14.9092)
Image4000	(18.8029)	(16.9547)	(18.1651)	(14.6936)	(19.2560)

6. Conclusions

The comparative analysis made on the various parallel versions of gaussian filtering process show that using CUDA parallelization methods lead to a great performance increase in the execution time, while maintaining the same efficacy of the sequential version.
CUDA streaming processors can exploit the higher number of threads and computational cores to efficiently execute the algorithm convolution over large images, with gains increasing exponentially to the number of pixels and the dimension of the kernel. As a side note, GPUs have usually lower memory to work with than the actual main memory a CPU can utilize to do its computations, a detail not to be overlooked when dealing with large datasets, that may require much more careful handling and partitioning if processed by using CUDA kernels.