Parallel Computing Final Term: Kernel Image Processing

Lorenzo Giannella
lorenzo.giannella@stud.unifi.it
https://github.com/lore1379/SobelFilter

Abstract

The aim of this final project is the implementation of a sobel filter, an edge detection algorithm. There are 4 implementations, one sequential version and three parallel variants, obtained with CUDA. A simple performance study on speed up has been made, in order to analyze the benefits of parallelism. The developed code is in C/C++ and CUDA language, tests were made on a 1080p resolution image.

1. Introduction

The Sobel operator, sometimes called the Sobel–Feldman operator or Sobel filter, is used in image processing and computer vision, particularly within edge detection algorithms where it creates an image emphasizing edges. It is a discrete differentiation operator, which computes an approximation of the gradient of the image intensity function by convolving the input image with a 3x3 kernel in the horizontal and vertical directions, respectively. Such operator lies at the basis of several algorithms for pedestrian detection in autonomous driving systems.

2. Algorithm Description

Given an input RGB image *I*, the Sobel Filter algorithm proceeds according to the following phases:

1. Convert *I* to a gray-scale image *O* according to Formula 1. Such formula is applied to every pixel p of *I* to produce a corresponding gray-scale pixel o of *O*. p_r, p_g, p_b represent the R,G,B intensity of pixel p, respectively, and each one lies in the range [0,255].

Figure 1: Result of Sobel Filter application

$$o = 0.2126 * p_r + 0.7152 * p_g + 0.0722 * p_b \quad (1)$$

2. Compute G_h , an approximation of the horizontal gradient of *O*, by convolving *O* with a 3x3 kernel according to Formula 2. $\circledast$ denotes the 2-dimensional convolution operation.

$$G_h = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \circledast O \quad (2)$$

3. Compute G_v, an approximation of the vertical

gradient of O, by convolving O with a 3x3 kernel according to Formula 3.

$$G_v = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \circledast O \tag{3}$$

4. Compute G, the final gradient magnitude, by combining the horizontal gradient G_h and the vertical gradient G_v according to Formula 4.

$$G = \sqrt{G_h^2 + G_v^2} \tag{4}$$

3. Algorithm Implementation

I implemented a squential version of the Sobel Filter. Afterwards, 3 parallel GPU implementations in CUDA are compared to evaluate the benefit of parallelization. Tests were made on the hardware outlined in Table 1.

CPU	Intel(R) Core(TM) i7-8700 CPU@3.20GHz
GPU	NVIDIA GeForce GTX 1080
RAM	16,0 GB
Hard Disk	SSD - 256 GB

Table 1: Hardware used to test sequential CPU and parallel GPU code performance

In all of the implementations I used stb [1] for manipulating images. In particular we used:

- image loader: stb_image.h
- image writer: stb_image_write.h

To manage images I used a struct called "Image", containing following informations:

- data: extracted thanks to stb_image.h, it's an unsigned integer of 8 bits lenght. Used to store image bytes.
- channels, width and height: extracted thanks to stb_image.h. They are integer values.
- size: calculated as $width * height * channels$.
- constructor methods to read or write, create blank, copy or destroy an image.

3.1. Sequential Version

The sequential version of the Sobel Filter [2] was implemented according to the following steps:

1. **Convert input PNG/BMP/JPG image to gray-scale:** with the aid of the read utility (stb_image.h), a format-independent representation of the input image is produced in RGB format. In such format the image has three channels so every pixel p of image I has an R,G,B component: p_r, p_g, p_b. Then by applying formula 1, a conversion of every single pixel p of the RGB image is made in a sequential way to produce pixel o of the gray-scale image O.
2. **Compute Horizontal Gradient:** by applying formula 3, we compute the horizontal gradient G_h over all pixels o of O by taking a 3x3 region around o and convolving it with the 3x3 kernel.
3. **Compute Vertical Gradient:** by applying formula 2, we compute the vertical gradient G_v over all pixels o of O by taking a 3x3 region around o and convolving it with the 3x3 kernel.
4. **Combine Vertical Gradient and Horizontal Gradient:** we combine the horizontal and vertical gradient in a sequential way over all the pixel g_h of G_h and g_v of G_v to produce pixel g of G, according to formula 4.
5. **Save processed image to PNG:** image is finally saved in PNG format with the aid of the write utility (stb_image_write.h).

3.2. Parallel Version - CUDA

CUDA [1] (Compute Unified Device Architecture) is a C/C++ parallel computing platform and API model created by NVIDIA that facilitates the process of writing GPU code: the CUDA

[1] github.com/nothings/stb, single-file public domain libraries for C/C++.

[2] Implementation is available at https://github.com/lore1379/SobelFilter/tree/main/SobelFilte

platform is a software layer that fives direct access to the GPU's instruction set and parallel computational elements, for the execution of *kernels*. A kernel is executed in parallel by an array of threads that runs the same code, arranged as a grid of threads blocks. Threads from the same block have access to a shared memory and their execution can be synchronized. Each thread has a specific ID in the form of "blockIdx * blockDim + threadIdx":

- threadIdx: Id of the thread in a block.
- blockIdx: Id of the block in a grid.
- blockDim: Dimension of a block

In this project three versions of the CUDA code have been produced: a naive version, a version that only use constant memory for the convolution mask and a tiled one that uses constant and shared memory. The first is the simple "transcription" of the sequential algorithm to CUDA, while the second and the third take advantage of optimizations, ideally speeding up the calculations:

- constant memory: read-only memory that resides in device memory. It is cached in a dedicated, per-SM, constant cache.
- shared memory: on-chip, thus it has a much higher bandwidth and much lower latency than global memory.

The parallel GPU version of the Sobel Filter was implemented with the aid of CUDA framework, so the computations of Section 3.1 occur in parallel over every single pixel p of input image I or pixel o of gray-scale image O. Such parallel abstraction is implemented concretely in three CUDA kernels having the following common characteristics: kernel_name <<< dimGrid, dimBlock >>> (argument_1, argument_2, ..., argument_n). dimGrid represents the amount of blocks in the CUDA grid, while dimBlock is the amount of threads per CUDA block, in a particular direction. In this manner, every single pixel is allocated in one single thread.

3.2.1 Naive parallel version

The naive version of the algorithm [3] is a "no-optimized transcription" of the sequential algorithm and it's composed by three kernels:

- ConvertImageToGrayGpu: extracts the thread ids (2D indexing) along x and y axis and combines them to obtain two 1D ids. The first index is for the destination array of pixels of the gray-scale image and the second, multiplied by channels number, is for the source array of pixel of the input image. Then proceeds with the conversion according to Formula 1 in a parallel manner.
- gradientConvolution: performs convolutions (formula 2 and 3) in a parallel manner taking into consideration image borders and, if it's outside, it performs no operation.
- sobelFilterGPU: performs formula 4 in a parallel manner.

Memory accesses in the naive version are performed on the global memory. Following optimizations are performed on the convolution part of the application.

3.2.2 Constant memory parallel version

The first optimization strategy proposed is to load masks in constant memory. In fact masks are never changed and are perfect candidates for the faster read-only memory. This can be done with the use of __constant__ .

3.2.3 Shared memory parallel version

The second optimization strategy is to add the use of shared memory. Data is sliced into tiles and loaded sequentially into shared memory. The accesses are then performed on this local cache, amortizing the more expensive cost of accessing global memory. Convolution is the process of adding each element of the image to its local neighbors, weighted by the mask. So we need a

[3] https://github.com/lore1379/SobelFilter/tree/main/NaiveParallelSobel

special treatment for the border of the image: we need to load not only the pixels of the image but also to add a 0 padding around the image itself. To do this we use the technique of 2-batch loading: we reuse threads of the block to load pixels of the image plus the 0 padded pixels needed to perform convolution, then evey thread writes the result in the output image in a one to one relastionship.

A block is made of $TILE\_WIDTH * TILE\_WIDTH$ threads. Since the shared memory area $(TILE\_WIDTH + Mask\_width - 1) * (TILE\_WIDTH + Mask\_width - 1)$ is larger than the block size $TILE\_WIDTH * TILE\_WIDTH$ and it is smaller than $2 * TILE\_WIDTH * TILE\_WIDTH$, then each thread should move at most two elements from global memory to shared memory, so it is convenient to split this loads in a " two stages" process. A synchronization barrier __syncthreads() is then used in order to wait the completion of this loading phase. The threads of the tile now can proceed to the computation and, after applying the convolution mask, are the final output tiles whose size is $TILE\_WIDTH * TILE\_WIDTH$.

4. Performance Analysis

In order to understand if these parallel implementations could improve the performances with respect to the sequential version, a simple speedup analysis has been performed. For testing purpose, a 1080p resolution image was used. Both the sequential and the parallel executions have been recorded 10 times and then the average was taken, in order to obtain a better estimation.

Speedup $Sp(n)$, is defined as the ratio of the execution time $T(l)$ on a sequential computer to the corresponding execution time $T(n)$ on the parallel computer for some class of tasks of interest:

$$Sp(n) = \frac{T(l)}{T(n)} \tag{5}$$

Tests were made on the hardware outlined in Table 1. The operating system is Windows 10. The GPU used for this analysis has the following properties:

- Compute capability: 6.1
- 2560 CUDA core
- Multiprocessor count: 20
- Max blocks per multiprocessor: 32
- Max threads per block: 1024

Comparison against CUDA will be done considering the speed up of parallel implementations with respect to sequential approach and the benefit of implementing the aforementioned optimizations. However we also want to see the effect of the variation in the number of threads in a kernel. Usually you want to choose the size of your blocks based on your GPU architecture, with the goal of maintaining 100% occupancy on the Streaming Multiprocessor (SM). For example, this GPUs can run 2048 threads per SM, and up to 32 blocks per SM, but each block can only have up to 1024 threads. Test will be done considering a 100% occupancy (figure 2) according to CUDA Occupancy Calculator [4]:

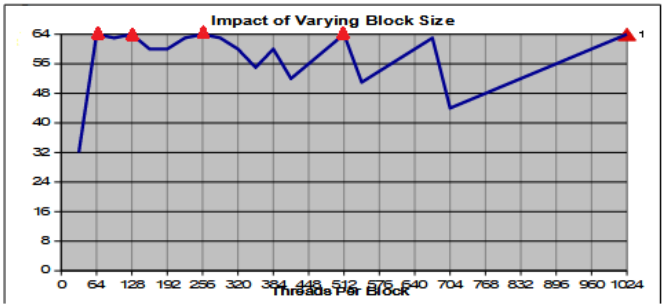
- 64 threads per block: 32 active Thread Blocks per Multiprocessor
- 128 threads per block: 16 active Thread Blocks per Multiprocessor
- 256 threads per block: 8 active Thread Blocks per Multiprocessor
- 512 threads per block: 4 active Thread Blocks per Multiprocessor
- 1024 threads per block: 2 active Thread Blocks per Multiprocessor

4.1. Experiment on a Full HD image - 1920 x 1080

The image we take into consideration is 1080p resolution, characterized by 1,920 pixels displayed across the screen horizontally and 1,080 pixels down the screen vertically. For 128 and 512 threads per block we considered blocks with

[4] https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html

Figure 2: CUDA Occupancy Calculator results



larger width with respect to height because experiments showed better results for the calculation of the gradient, as we can see in Table 2, performed in the naive implementation. Other operations performed similarly.

128 Threads		512 Threads	
16x8	8x16	32x16	16x32
0.957 ms	1.093 ms	0.864 ms	0.934 ms

Table 2: Gradient convolution times using different orientation of blocks for naive implementation

As we can see in Figure 3 we obtain a considerable speed up performing the 3 main operations of the algorithm. In particular for the grayscaling operation due to the fact that the original coloured image has 3 channels to compute. Although using more threads in a block doesn't change execution times significantly, grayscaling and combination of gradients get slightly worst. This is due to the fact that the two operations are computationally intensive and don't take advantage in changing block shape so the limiting factor is the computation on SMs and having more threads to manage can cause an overhead lowering performances while using smaller blocks better exploit the use of streaming multiprocessors. On the contrary, calculation of gradient performs slightly better using more threads per block. That is because having larger blocks helps with better memory bandwidth utlization slightly lowering the latency of memory operations.

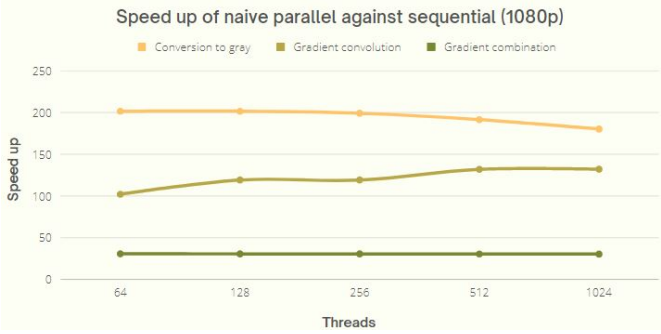


Figure 3: Speed up of naive parallel implementation against sequential

4.1.1 Testing optimizations against naive implementation

For this experiment we only take into consideration the convolution operation because it is the only one that takes advantage of aforementioned optimizations. As we can see in Figure 4 the two optimizations give a great speed up improvement over the naive implementation. Thanks to Nvidia profiler we could analize the behaviour of the two optimizations. Constant memory got a significant improvement in speed up from 64 threads per block to 128 and on: that's because we lower the memory throttle percentage given by pending memory operations that prevent further forward progress. Surprisingly adding tiling to constant memory optimization didn't improve the speed up further: the profiler, thanks to a sample distribution, highlighted an increasing memory throttle and increasing need of synchronization with respect to the increasing number of threads per block, lowering the speedup. Furthermore it showed that instruction and memory latency is limiting the performance of the kernel, when the GPU does not have enough work to keep busy. This is because instruction execution is stalling excessively.

4.2. Final Observations

As we can see a parallel approach in image processing is always better than a sequential approach and can drammatically improve performance for some operations. In fact we peak at

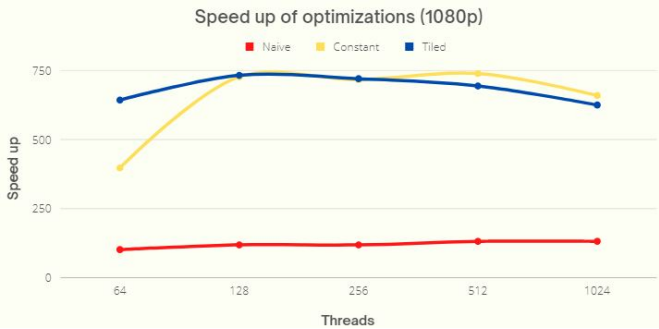


Figure 4: Speed up of parallel implementations

a speedup of approximately 750 times for an optimized convolution operation. However results will vary depending on the number of threads per block we take in consideration and the shape of the block itself. For the experiment we only considered blocks that achieved 100% theoretical occupancy to get better results. Summarizing we observed that:

- chosing larger width with respect to height always improves execution times.
- increasing the number of threads per block improves convolution process in the naive parallel implementation at the cost of slightly worst performance in computing grayscale image and in the final combining operation of gradients.
- optimization, like the usage of constant memory, greatly improves speed up with respect to naive implementation.
- adding tiling, with the use of shared memory, improved speed up with respect to the only usage of constant memory for the mask when 64, 512 and 1024 threads per block were used.
- major limiting factor in the usage of tiling comes from the load process from global to shared memory. The source of bad performance is due to bank conflicts during shared memory accesses, which would be solved by using vectorized loads and stores of 4B, the bank size.

References

[1] NVIDIA, P. Vingelmann, and F. H. Fitzek. Cuda, release: 11. 2020.