Parallel computing mid term: morphological operators

Angelo D'Amante, Fabian Greavu

Università degli studi di Firenze
angelo.damante@stud.unifi.it fabian.greavu@stud.unifi.it

May 10, 2021

Abstract

In this project we've developed some implementations of main morphological operators on gray scale and binary images. First we are going to introduce the operators and their effect on the images. Then we show the c++ sequential implementation and two parallel versiones (using CUDA in c++ and Threads in JAVA). Finally we show the resulting speedup with respect to image size and used blocks/threads. All the code can be found on github at (1)

I. Introduction

Mathematical Morphology is a theory and technique for the analysis and processing of geometrical structures. Developed by *Matheron* and *Serra* to quantification of mineral characteristics, this technique resulted advancements in integral geometry and topology. (2)

a. Definitions

Morphological operators are designed to work with binaries images, however it is possible toextend them to grayscale and even colorized images. Every operator need:

- An *Image* to process with own height, width and spectrum.
- A *Structuring Element* represented by matrix shape to probe Image.

The probe element is centered on the pixel then all him foreground pixels are matched with the image's pixels.

b. Basic Operators

Let E an euclidean space, given an image I in $\mathbb{Z}$, a probe element M in $\mathbb{Z}$ and let M_z the translation of probe element by a vector z and $(M^s)_z$ its symmetrical. We define the follow basic operators:

Erosion: $I \ominus M = \{z \in E | M_z \subseteq I\}$

Dilatation: $I \oplus M = \{z \in E | (M^s)_z \cap I \neq \emptyset\}$

c. Composed Operators

Applying basic operators in a different order we obtain:

Closing: $I \bullet M = (I \oplus M) \ominus M$

Opening: $I \circ M = (I \ominus M) \oplus M$

II. Sequential Implementation

We present a simple sequential implementation with code executed by a CPU. The chosen programming language for this version is C++. Let Neighborhood [ ] be the set of neighboring foreground pixels. Dilatation operation select the max of Neighborhood and erosion operation select the min.

Show the pseudo-code:

Algorithm 1 Pseudo-Code sequential

```
1:  for rowImg = 0,1,...,imgH do
2:      for colImg = 0,1,...,imgW do
3:          for rowPrb = 0,1,...,probeH do
4:              for colPrb = 0,1,...,probeW do
5:                  update(Neighborhood[ ])
6:              end for
7:          end for
8:          update(Img[rowImg*imgW+colImg])
9:          with max or min of Neighborhood
10:     end for
11: end for
```

We can notice the presence of four nested cycles, two external to scan each image's pixel and two internal to scan his neighborhood and finally, compute the max or min value. The computational complexity is dominated by the size of the image, ending in a $\Theta(imgH \times imgW)$ time.

III. CUDA Implementation

This implementation defines a kernel function executed by the GPU threads independently. Each thread will compute a single image's pixel. Given the shape of the data to be processed, we have chosen a two-dimensional managing for blocks and threads. The image is divided into tiles. We assume

- **bx**, **by**: indexes of block along x,y axis.
- **tx**, **ty**: indexes of thread along x,y axis.

Algorithm 2 Naive-Parallel version

```
1: colImg = bx * blockDim.x + tx
2: rowImg = by * blockDim.y + ty
3: for rowPrb = 0,1,...,prbH do
4:     for colPrb = 0,1,...,prbW do
5:         update(Neighborhood[ ])
6:     end for
7: end for
8: update(Img[rowImg*imgW+colImg])
```

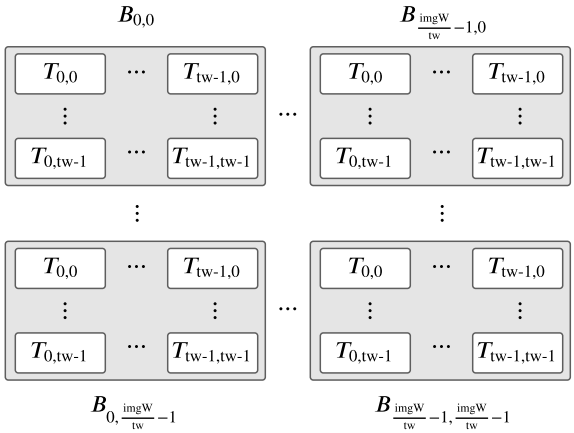

Figure 1: *Thread Blocks manage in GRID to map Image.*

Let tw be the tile width that define width and height of thread block. The image mapping is shown in the figure 1.

A. Naive implementation

Probe element is stored in constant memory and Image element is stored in global memory. This version has slowdowns caused by too many accesses to the global memory. The algorithm 2 shows a simple implementation and we can note the presence of only two cycles to map probe element in the image.

B. Optimized implementation

This version consists of reducing global memory accesses by working in shared memory. The *"2-batch loading"* has been implemented as follows: let be $w = TILE\_WIDTH + PROBE\_WIDTH - 1$, this technique consists in two phases.

Algorithm 3 Optimized-Parallel version

```
1: __shared__ float tileDS[w][w]
2: Load TILE_WIDTH × TILE_WIDTH
3: Load the data outside of the
   TILE_WIDTH × TILE_WIDTH
4: __synchtreads()
5: Compute MM operation in tileDS[][]
6: __synchtreads()
```

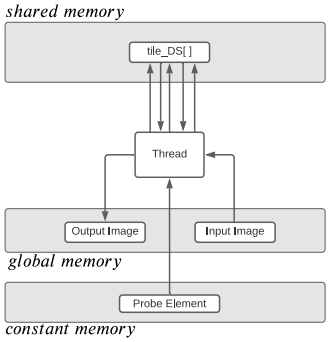

Figure 2: *Policy of optimized version.*

Padding policy was adopted to handle tile boundaries. Moreover, we can notice that in the first phase, all threads works and in the second phase, only some threads work. An idea to quantify accesses is shown in figure 2.

IV. JAVA Implementation

For better comparison an additional sequential JAVA version have been created and results from parallel version will be compared to it instead of the C++ version.
The parallel version takes advantage of the points independence when executing kernel calculation. The entire image is divided into chuncks of rows. Each chunk will be given to a single thread that will compute necessary operation on it.
This algorithm does not work in place. Results from the given operation will be written into a new matrix and finally converted into an output image.
In order to synchronize all the jobs, a barrier is inserted after all threads execution.
Assuming:

- **width**: width of image.
- **height**: height of image.
- **num_ths**: Given threads number.

The algorithm is showed in 4.

V. Results

In this section are represented speedups obtained about the parallel version with CUDA and Java. Five image resolutions were analyzed to compare the results, from 1280×720 to 7680×4320. Ten images were processed for each resolution to obtain mean and max speedup evaluation.

Algorithm 4 JAVA-Parallel/sequential version

```
1: int output[] = new int[width*height];
2: ExecutorService taskExecutor = Executors.newFixedThreadPool(num_ths);
3: int r = (int)Math.floor(height/num_ths);
4: for int y; y < height; y += r do
5:     MaskApplier t = new MaskApplier(...);
6:     taskExecutor.execute(t);
7: end for
8: waitExecutors(taskExecutor);
9: return CreateImage(...,output)
```

A. Speedups for CUDA version

The tests for the sequential version were obtained with CPU Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz. The tests for parallel version were obtained with GPU GeForce GTX 1050 Ti.

In according to CUDA rule that consists in *"number of threads launched per block must be multiple of warp-size = 32"* [1], we highlight the results varying tile dimension (8/16/24/32) and obtain (64/256/576/1024) threads per block respectively.

The mean speedups result are in table 1 and table 2. The max speedups obtained are in table 3.

Resolution	Tile Width = 8		Tile Width = 16	
	Naive	Opt	Naive	Opt
1280x720	41.16	42.44	48.88	51.06
1920x1080	44.68	47.77	53.61	57.98
2560x1440	45.62	47.44	59.22	61.44
3840x2160	43.18	44.89	62.44	65.33
7680x4320	38.28	39.61	55.45	59.46

Table 1: *Mean Speedups for MM Operation with 64 and 256 threads per block.*

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Resolution	Tile Width = 24		Tile Width = 32	
	Naive	Opt	Naive	Opt
1280x720	67.57	73.21	63.17	64.58
1920x1080	69.55	74.44	65.47	67.88
2560x1440	72.13	75.95	67.41	69.71
3840x2160	75.87	79.87	67.45	68.35
7680x4320	78.21	79.60	68.23	69.47

Table 2: *Mean Speedups for MM Operation with 576 and 1024 threads per block.*

Resolution	8	16	24	32
1280x720	45.10	53.24	85.54	70.74
1920x1080	49.21	61.08	83.41	75.44
2560x1440	51.21	63.47	88.98	77.74
3840x2160	46.00	69.21	91.10	73.57
7680x4320	41.63	61.00	89.94	73.48

Table 3: *Max Speedups for MM Operation for optimized version using shared memory.*

B. Speedups for JAVA Version

The java sequential version is done by calling the parallel version with num_ths = 1. Tests were done on CPU Intel(R) Core(TM) i7-9700K CPU @ 4.90GHz using thread number n^2 from 2 to 128. Having only 8 cores (physical and logical) we removed bigger thread numbers because speedups did not increase.
Table 4 shows speedups for Dilation operation. Maximum speedup achieved is 2.636 with 8 cores. Similar results to other Morphological operators.

Resolution	2	3	4	6	8
1280x720	1.58	1.96	2.19	2.50	2.61
1920x1080	1.55	1.91	2.13	2.42	2.50
2560x1440	1.55	1.89	2.12	2.41	2.64
3840x2160	1.52	1.87	2.09	2.38	2.56
7680x4320	1.50	1.84	2.04	2.33	2.47

Table 4: *Max Speedups for MM Operation for optimized version using shared memory.*

References

[1] Fabian Greavu Angelo D'Amante. morphological-image-processing-in-parallel. `https://github.com/AngeloDamante/morphological-image-processing-in-parallel`, 2021.

[2] Wikipedia contributors. Mathematical morphology — Wikipedia, the free encyclopedia, 2021. [Online; accessed 10-May-2021].