Parallel Computing Project 2 - Mathematical Morphology

Irene Campaioli
irene.campaioli@edu.unifi.it

Abstract

In this project we implement the four main operators of Mathematical Morphology (erosion, dilation, opening and closing), comparing the performances of a sequential approach and a parallel one using CUDA.

1. Introduction

Mathematical morphology is a non-linear theory of image analysis, based on set theory and topology. Originally defined only for binary images, it was later extended to grayscale images as well.

The basic idea of mathematical morphology is to probe an image with a simple, pre-defined shape (called 'structuring element') and see how the image responds to it. Mathematical morphology is a very popular framework in image processing and it's mostly used in applications like noise removal, feature extraction, image segmentation and many more. In this laboratory we implemented the operations of **erosion**, **dilation**, **opening** and **closing** in their grayscale-image form.

Binary Morphology

We first introduce the operators and formulae in their binary form.

The **Erosion** of on image A by the structuring element B is defined as:

$$A \ominus B = \{z | B_z \subseteq A\}$$

where B_z is the translation of B by the vector z.

Similarly, the **Dilation** of an image A by the structuring element B is:

$$A \oplus B = \{z | \hat{B}_z \cap A \neq \emptyset\}$$

where $\hat{B}_z$ refers to the *symmetric* of B, then translated. As the name suggests, erosion is an operator that thins the image, while dilation thickens it.

The **Opening** and **Closing** of A by B are defined respectively as:

$$A \circ B = (A \ominus B) \oplus B$$

$$A \bullet B = (A \oplus B) \ominus B$$

Opening is an erosion followed by a dilation (using the same structuring element!) and it's used to remove protrusions. Like erosion, it thins parts of the image. Closing is a dilation followed by an erosion, and it adds protrusions to the image.

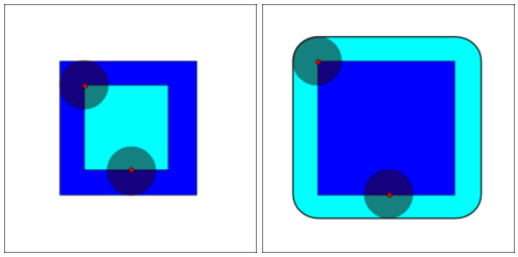

Figure 1. (Left). Erosion of the dark-blue square using a disk as a structuring element, resulting in the cyan square. (Right). Dilation of the dark-blue square by a disk, resulting in the bigger, rounded square in cyan.

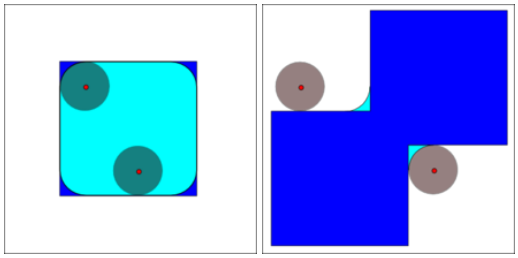

Figure 2. (Left). Opening of the dark-blue square. (Right). Closing of the dark-blue shape (union of two squares) by a disk, resulting in the union of the blue squares plus the cyan areas.

Lattice Morphology

Lattice is a term used to refer to grayscale images, although we still consider the structuring element to be binary. The operators can be re-defined as such: The **Erosion** and **Dilation** of an image f by the structuring element b are:

$$E(f, b)[x, y] = \min_{(u,v) \in S(b)} f(x + u, y + v)$$

Figure 3. Top-left: Erosion; Top-right: Dilation; Bottom-left: Opening; Bottom-right: Closing.

Figure 4. Original

$$D(f,b)[x,y] = \max_{(u,v)\in S(b)} f(x-u, y-v)$$

where $S(b)$ is the *support* of B, the set of point (u,v) such that $b(u,v) = 1$.

Figure 3 shows an example of all four operators on a grayscale image (Original: Fig.4).

We can see that the visual result is different from the binary case: the effect of erosion is a general darkening of the image, while dilation lightens it. Opening can be seen as an operator that removes bright objects smaller than b, while leaving the overall intensity and larger bright areas unaltered. Closing, on the other hand, removes small dark details.

2. Implementation

All code is available at `https://github.com/crashkeys`, divided in two folders for the sequential version and the CUDA one respectively. It requires the OpenCV library to handle images inputs.

2.1. Sequential solution

The core of the program is in `MMops.cpp`, in which we define both the `erosion` and `dilation` operations for one pixel, given its coordinates `(i,j)`. We will show only the implementation of the first one, as the dilation can easily be derived from the definition.

```
int erode(Mat& frame, Mat& element, int i, int j){
    int value = 255;
    for (int u=0; u<element.rows; u++) {
        for (int v=0; v<element.cols; v++) {
            if (element.at<uchar>(u,v) == 1 ) {
                int temp = frame.at<uchar>(i+u, j+v);
                value = min(temp, value);
            }
        }
    }
    return value;
}
```

where `frame` refers to the input image and `element` to the structuring element. Both are `cv::Mat` objects. To generate the complete eroded and dilated images, we will need to call this function once for every pixel in our `main` function (two extra `for` loops). Additionally, we'll need to repeat the scan of the whole input image a second time

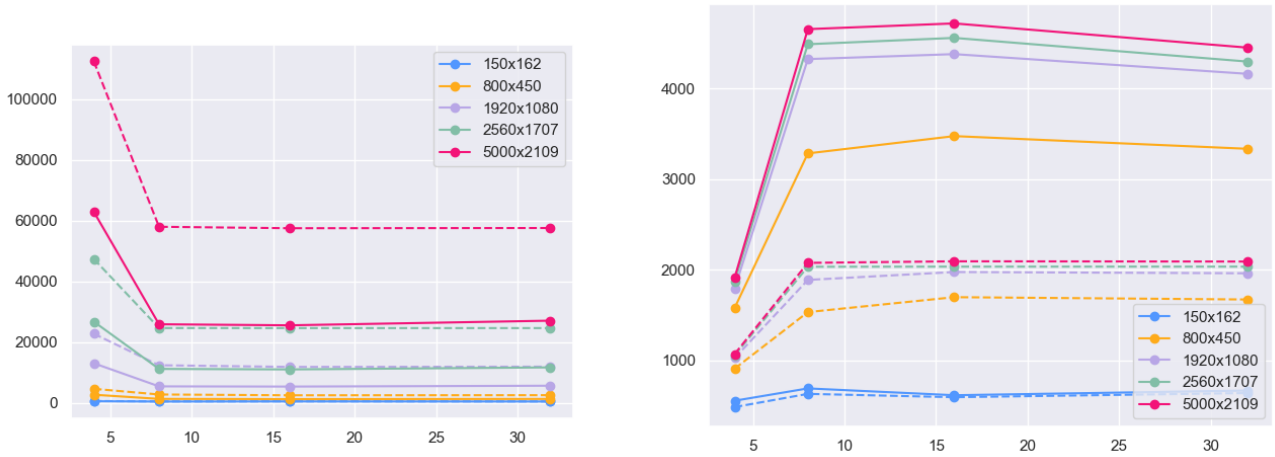

Figure 5. On the left, Exectution time (in microseconds) for images of different sizes, varying the block size. On the right, speedups comparing the CUDA times with the sequential one. For both graphs, dotted lines refer to the *naive* version, while full lines to the shared memory one. For the exact values refer to Tab.1 and Tab.2.

for the operations of opening and closing, since we need the finished eroded and dilated frames.

Despite the operations of erosion and dilation themselves on a single pixel being easy, the multitude of `for` loops severely slows down the execution time, especially for very big images. That's why in the next section we tackle the problem with a parallel approach using CUDA and the GPU's power.

2.2. CUDA solution

The parallel approach completely removes the two external `for` loops and generates enough threads so that each can perform the operations on one pixel independently.

```
__global__ void erode(uchar* d_input, uchar* d_output,
    int width, int height) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < height && col < width) {
        uchar value = 255;

        for (int el_id = 0; el_id < ELEM_SIZE*ELEM_SIZE;
     el_id++) {
            int r = floorf(el_id/ELEM_SIZE);
            int c = el_id % ELEM_SIZE;

            int newRow = row + r;
            int newCol = col + c;

            if (newRow>=0 && newRow<height && newCol>=0
    && newCol<width) {
                if (ELEM[el_id] == 1) {
                    uchar temp = d_input[newRow*width +
    newCol];
                    value = value < temp ? value : temp;
      //MIN
                }
            }
        }
        d_output[row * width + col] = value;
    }
}
```

To further improve the performance, we moved the structuring element in the constant memory.

What we just showed will be referred from now on as a 'naive' solution, because it's the simplest implementation in CUDA. While, as we'll see in the experiment section, this will still be giving exceptional results, we also provided another solution that makes use of *shared memory* between threads of the same block. The shared memory is defined as such:

```
size_t sharedArray = (blockSize+ELEM_SIZE-1)*(blockSize+
    ELEM_SIZE-1)*sizeof(uchar);
```

and it will be filled with data of the input image that's overlapping between threads. So when we want to read the pixel value like in line:

```
uchar temp = d_input[newRow*width + newCol];
```

we can instead refer to the shared memory, saving some time.

```
uchar temp = sharedMem[(sharedRow + r) * paddedSize + (
    sharedCol + c)];
```

Due to come tweaking to the definition of the kernel, we can perform the operations of erosion and dilation at the same time, since they are independent. Therefore, in our main we will call the kernel by passing explicitly the shared memory and the two outputs.

```
MM_ops<<<dimGrid, dimBlock, sharedArray>>>(d_input,
    d_output_EROSION, d_output_DILATION, width, height)
    ;
```

3. Experimental Results

All tests were performed on an AMD Ryzen 5 7600 and an NVIDIA GeForce RTX 3060 GPU. All tests were repeated fifty times to validate the measured data. Note that

Image Size	Sequential	Naive				SharedMemory			
		4	8	16	32	4	8	16	32
150x162	274	0.565	0.434	0.461	**0.428**	0.494	**0.396**	0.444	0.410
800x450	4163	4.582	2.716	**2.453**	2.490	2.637	1.268	**1.198**	1.248
1920x1080	23299	22.704	12.342	**11.80**	11.88	13.0	5.388	**5.321**	5.599
2500x1707	49983	47.33	24.59	**24.57**	24.57	26.68	11.14	**10.97**	11.63
5000x2109	120243	112.38	57.92	**57.45**	57.52	63.85	25.83	**25.49**	27.015

Table 1. Completion time (in ms) for images of different sizes, varying the block-size

Image Size	Naive				SharedMemory			
	4	8	16	32	4	8	16	32
150x162	485	631	594	**640**	555	**691**	616	668
800x450	909	1533	**1698**	1672	1579	3284	**3474**	3335
1920x1080	1026	1888	**1975**	1961	1793	4324	**4379**	4162
2500x1707	1056	2033	**2035**	2035	1873	4488	**4558**	4297
5000x2109	1070	2076	**2093**	2091	1913	4654	**4718**	4451

Table 2. Speedups (Higher is better).

all the following results do not just refer to the execution time of the operations, but the time includes the allocation of the objects as well. This is because we expect the GPU kernel execution to be much faster, but we must also consider the time spent in creating objects and moving all the data on the graphics card.

3.1. Varying the block size

Choosing the right block size is the main step in a highly performative parallel code. We used the following formulae to choose the number of blocks and the number of threads-per-block:

```
int blockSize;
dim3 dimBlock(blockSize, blockSize);
dim3 dimGrid((width + blockSize - 1)/blockSize , (
 height + blockSize - 1)/blockSize);
```

We know from literature that the optimal choice for `blockSize` is a multiple of 32 (warp-size), therefore we chose $[4, 8, 16, 32]$ for block-size, for a total of $[16, 64, 256, 1024]$ threads-per-block.

Table.1 shows that the best block-size is almost always 16, for a total of 256 threads-per-block, except for very small images. This is further confirmed by the occupancy calculator (using NVIDIA Nsight Compute), that gives us a 100% used occupancy when using this number of threads. Note how the shared memory version is always better than the naive one, confirming the usefulness of a proper memory management.

3.2. Speedups

We compare the results of the CUDA version from the section above and the sequential approach. Tab 2 shows the speedups for the parallel approach: it becomes increasingly better as the size of the image grows. Yet again, we can see how much the correct use of a shared memory improves on the result.

4. Conclusion

This type of exercise is what we call ”embarrassingly parallel”, since each operation on a pixel is independent, and this is evident by the very high speedups we observed in our experiments. Even for smaller images, it’s still convenient to transfer the data to the graphics card and take advantage of its parallelization.