PC-2018/19 Course Project - Morphological Operators

Riccardo Malavolti
riccardo.malavolti@stud.unifi.it

Abstract

In the image processing context, morphological operators are the precursors of modern image segmentation systems. Today these methods are replaced by specifically-designed algorithms, more efficient than these general-purpose operators. Still, they are not abandoned: their low computational cost and their generality are excellent reasons to use them in the preprocessing phases. Could be useful study and implement efficient implementations, in order to understand the multi-core architectures impact. In this work a CPU and three GPU implementations will be presented, to show how the quality of the code could affect execution time.

1. Introduction

The theory behind matematical morphology came from mining industry: Matheron and Serra needed a method to analyze mineral characteristics from samples obtained with cross section. They initially developted the hit-or-miss transform, then deducting the others operators.

1.1. Definitions

Morphological operators are designed to work with binarized images, however it is possible to extend them to grayscale and even colorized images. Every operator need an image and a *structuring element*: a pattern represented in a matrix (usually 3x3, 5x5, 7x7) used to compute every image pixel's value. The pattern is centered on the pixel then all him foreground pixels are matched with the image's pixels.

1.1.1 Basic operators

We define two basic operators:
Erosion Given an image A in $\mathbb{Z}$, a structuring element B in $\mathbb{Z}$ and E, an euclidean space, we can define the erosion operator as:

$$\ominus(A, B) = \{z \in E | B_z \subseteq A\}$$

Where B_z is the structured element translated by a vector z.
Dilatation Given an image A in $\mathbb{Z}$, a structuring element B in $\mathbb{Z}$ and E, an euclidean space, we can define the erosion operator as:

$$\oplus(A, B) = \{z \in E | (B_z)^s \cap A \neq \varnothing\}$$

Where $(B_z)^s$ is the symmetric of B_z defined above.

1.1.2 Composed operators

Erosion and dilatation can be composed in order to create noise removal operators, top-hat and bottom-hat transforms can be used as edge revealers.
Opening Defined as the dilatation of the erosion of A by B.

$$\Omega(A, B) := \oplus(\ominus(A, B), B)$$

Dilatation Defined as the erosion of the dilatation of A by B.

$$\Delta(A, B) := \ominus(\oplus(A, B), B)$$

Top Hat transform Defined as the difference between the original image and his opened version.

$$\Theta(A, B) := A - \Omega(A, B)$$

Bottom Hat tranform Defined as the difference between the closed version and the original image.

$$\beta(A, B) := \Delta(A, B) - A$$

2. Implementation

In this section, will be presented a sequential and a parallel implementation, this last one in three variants.
In every implementation was implemented erosion, dilatation, opening, closing, top hat and bottom hat methods. Composed operators were implementated using method calls, so a little overhead should be considered: i.e.

```
Image_t* opening(Image_t* input, StructElem*
    structElem){

    return dilatation(erosion(input,
        structElem), structElem);
}
```

2.1. CPU sequential implementation

The sequential implementation was none but the classic one of erosion and dilatation algorithms.

```
// Pseudocode for erosion algorithm
for row in (0,imgH):
    for col in (0,imgW):
        for i in (0, strelH):
            for j in (0, strelW):
                x=row+i-strelRad
                y=col+j-strelRad
                neighborhood.add(img[x*imgW+y])
            end for
        end for
        output[row*imgW+col]=max(neighborhood)
    end for
end for
```

This above, is a simplified version of the original code**??**, in order to improve readability. It's composed by four for cycles innested, the external two scan every input image's pixel while the internal two scan his neighborhood. The computational complexity is dominated by the size of the input image, ending in a θ(Height $\times$ Width) time.

2.2. GPU parallel implementations

The implementation of every parallel code was written in CUDA. In all of them, thread blocks were mapped on the input tile, so there was necessary an adequate strategy to manage the padding.

2.2.1 Naive implementation

In this version the kernel code simply reads neighborhood's pixels from global memory for each thread, and writes max or min value (respectively for erosion and dilatation) in the correct location of output image. Code is similar to CPU version:

```
// Pseudocode for erosion algorithm:
for i in (0, strelH):
    for j in (0, strelW):
        x=threadX+i-strelRad
        y=threadY+j-strelRad
        neighborhood.add(img[x*imgW+y])
    end for
end for
output[imgX*imgW+imageY]=max(neighborhood)
```

of course the two external for cycles are absent due to GPU architecture: each tread is mapped on a single input image's pixel.

2.2.2 Shared Memory

In this variant the input tile was loaded in shared memory, the first and the last thread (NW and SE) load 50% of padding each, filling with pixel of the image or a special value for out-of-border zones. First thread loads his own pixel, then proceeds to fill the padding with the pixel above and to the left of the tile. Last thread loads pixels on the bottom and to the right of the tile. See fig.1 for a schematic of this policy.

This approach obviously charges a lot of work on the two corner threads, so higher times should be expected during images elaboration.

2.2.3 Shared Memory - Optimized padding loading

In order to lighten the workload on NW and SE thread, in this variant each thread on the tile border loads his pixels along his normal. There still are four special cases: NW/NE/SE/SW corners

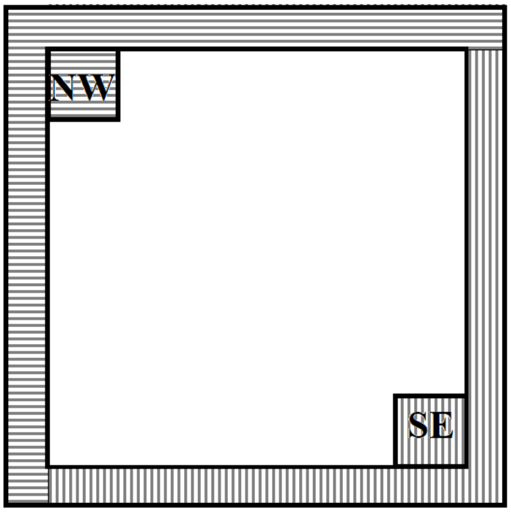

Figure 1. Padding loading policy for shared memory variant.

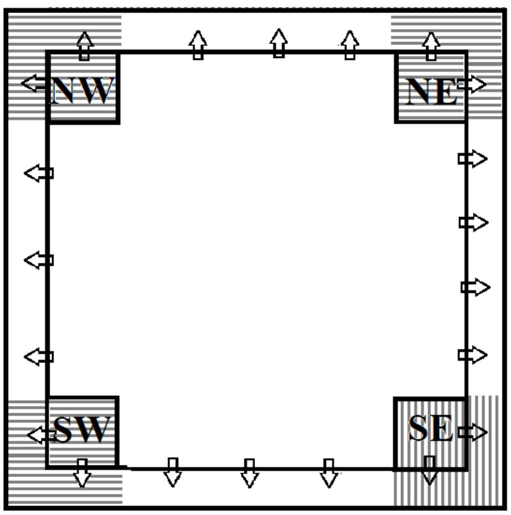

Figure 2. Optimized padding loading policy for shared memory variant.

have to do a little more work, loading a quarter of the structuring element's area. Fig.2 provide a schematic of this approach.

3. Results

All implementations were tested with five images (See Fig.3), resolutions are indicated in Table 1; each image has been binarized previously and saved in a .ppm file.

Test were conducted running the program ten times for each implementation, measuring times for every operator.GPU version was tested varing tile dimension (8/16/32), and in this measura-

Figure 3. Test images. From top-left, clockwise order: apple_adam, logitech_bill, macintosh, two_bites_better, micro-pro_wordstar.

ments, time was recorded on the kernel call, so the overhead due to memory allocation and copy is not included. Sequential version was tested on a AMD Ryzen 1600 CPU (3.2/3.7 GHz) equipped with DDR4 2400MHz RAM, GPU version was tested on a Nvidia K80. Table 2 shows times in seconds for three cases, Table 3 shows speedup.

It is clear that GPU implementation is faster than CPU one; it's interesting to note tile's size timings suggest that a bigger tile does not equal to lower times: 1024 threads per block are too much in order to mantain 2 thread blocks on the same Stream Multiprocessor. If this is true in the naive version, in the other flavours this effect is exagerrated by a bad padding loading policy, wich increases the execution time of some threads (even the optimized version is affected by this phenomen), thus a SM is occupied for a longer time.

4. Resources

Code is aviable on my github page:

- Sequential code (C++): github.com/rickie95/MorphOpsCPP
- Parallel code (CUDA):

Name	Resolution	N. of pixels
logitech_bill	432x596	257 472
micropro_wordstar	778x1088	846 464
apple_adam	995x1314	1 307 430
two_bytes_better	2340x3228	7 553 520
macintosh	4871x6466	31 495 886

Table 1. Test images details.

Image	**Operator**	**CPU**	**GPU**								
			Shared			**Opt. Shared**			**Naive**		
			8	16	32	8	16	32	8	16	32
logitech_bill	Erosion	0,0188	0,0007	0,0005	0,0010	0,0006	0,0004	0,0005	0,0005	0,0003	0,0003
	Opening	0,0380	0,0014	0,0011	0,0020	0,0011	0,0008	0,0009	0,0009	0,0006	0,0007
apple_adam	Erosion	0,4960	0,0190	0,0142	0,0260	0,0153	0,0112	0,0124	0,0124	0,0082	0,0085
	Opening	1,0288	0,0380	0,0284	0,0520	0,0307	0,0225	0,0248	0,0248	0,0164	0,0169
macintosh	Erosion	2,0761	0,0777	0,0584	0,0982	0,0626	0,0464	0,0512	0,0511	0,0339	0,0348
	Opening	4,1218	0,1535	0,1180	0,1732	0,1258	0,0935	0,1045	0,1022	0,0678	0,0698

Table 2. Elaboration times. Times are in seconds.

Image	**Operator**	**GPU**								
		Shared			**Opt. Shared**			**Naive**		
		8	16	32	8	16	32	8	16	32
logitech_bill	Erosion	27	35	19	34	44	41	41	59	56
	Opening	28	36	19	35	45	42	42	61	58
apple_adam	Erosion	26	35	19	32	44	40	40	60	59
	Opening	27	36	20	34	46	41	41	63	61
macintosh	Erosion	27	36	21	33	45	41	41	61	60
	Opening	27	35	24	33	44	39	40	61	59
Mean		27	**35**	20	33	**45**	41	41	**61**	59
Max		28	**36**	24	35	**46**	42	42	**63**	61

Table 3. Speedup GPU vs CPU.

github.com/rickie95/MorphOpsCUDA

- Wikipedia:
  en.wikipedia.org/wiki/Mathematical_morphology
- A page on CS Departement of Auckland Univerisity (New Zeland):
  cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm
- Kirk, Hwu - *Programming Massively Parallel Processors: A Hands-on Approach*