ELEC-H-473 — Exercise 01

C and first optimisations using SIMD – image threshold: Simple digital image-processing algorithm

1. Description of the lab

Objectives

1. Learn how to use Visual Studio form MS (editor, compile etc.).
2. Write elementary C programs and how to use SIMD with in-line assembly.
3. Learn how to use the debugging environment.
4. For the proposed problem, implement C and SIMD version of the program.
5. Measure execution times to compare different implementations and understand the advantages of SIMD.

2. Preliminary

- Start by writing Hello World program
- Modify and add simple arithmetical operations
- Launch the program in DEBUG mode and execute the program step-by-step
- Learn to how to observe register file changes

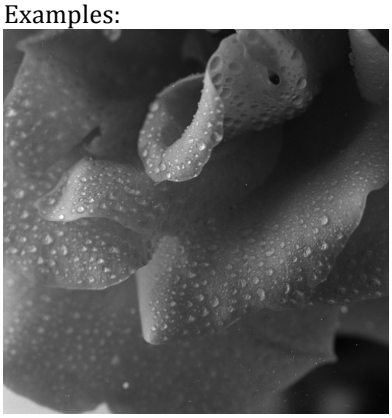
3. Introduction

Digital images are represented in form of a 2 dimensional matrix with les indexes *i,j* represent the spatial coordinates of the image. For a colour image there are 3 matrices, one matrix per colour component R, G and B (Red, Green, Blue). For a grey level image, only one matrix is necessary (R, G, B components have the same value). Depending on image quantification we can have more or less colours. Most of the time grey level images are coded using 8 bits of information per pixel. This means 256 colours (from 0 to 255 → $256=2^8$) with:

- 0 black $(00000000)_2 = (0)_{10}$ and
- 255 white $(11111111)_2 = (255)_{10}$.

In image processing applications it is often useful to work with binary images that use only black and white colours. For such images the pixel value is given only with a binary choice: black (0) or white (1). For images coded with 8 bits, this means 0 or 255 (0x00 or 0xFF in hexadecimal).
In order to convert one grey level image into a binary image we need to fix a threshold value. If the current pixel value is smaller then the threshold value, we will set the new pixel value to 0; otherwise it will be fixed to 255. Depending on the input image and the value of the threshold we will obtain different resulting images (see Figure bellow). For this exercise you can use the value of the threshold as you like. We are only interested in the way computation is being performed (and the execution time). Explain why?

Examples:


a) Original image


b) Threshold at 75


c) Threshold at 128

Here is the pseudo-code of what you will have to program:

```
for (all_images) {
 read_image_from_file
 start_time = get_time ()
 for (all_pixels_in_the_image) {
   if (curent_pixel_value < treshold) then new_pixel_value = 0
      else new_pixel_value = 255 }
 end_time = get_time ()
 out >> « Computing time » end_time – start_time
 write_image_result
}
```

4. Implementation C/C++

Implement the program using plain C language.

Help

- How to measure time in C:

```
#include time.h
…
time_t start, end ;
float dt ;
…
start_time = clock ();
… code to mesure …
end_time = clock ();
dt = (stop_time-start_time)/(float)(CLOCKS_PER_SEC) ;
```

- How to correctly print dt?
- Compile the same programme for DEBUG (default) and RELEASE target. Compare the execution time and explain the difference if any.

5. Implementation in SIMD

Before programming think on how you would solve this problem on paper. The idea is to look for the vector operation that could perform required computation. Some SIMD operations (google for the others):

- `pand xmm0, xmm1`
  - Logical AND operation between 2 registers
- `pcmpeqb xmm0, xmm1`
  - Compares 2 values in mode BYTE. If TRUE, all bits of the destination register are set to 1, 0 otherwise
- `pminub xmm0, xmm1 (ou pmaxub)`
  - Performs the comparison in en mode BYTE between xmm0, xmm1. Puts minumun of tw in `xmm0`.

Help

Few indications:

- How to include assembly code inside C ? Here is the example of the loop — the loop counter is stored in `ecx`:

```
fonctionC()
{
…
_asm{
// Code assembleur SIMD y compris
…
label1 :
…
sub ecx , 1 ;
jnz label1
}
…
emms;
}
```

  For compatibility reasons, finish your program using `emms` (*Empy MMX State*) instruction.

- Data movement → pointer into the register:
  `mov esi, ptrin`
  - Loads the value of the pointer `ptrin` into register `esi`
- Move the value from the pointer the register into central memory at the address stored in the pointer `edi`:
  `movdqu xmm7, [edi]`
  - Loads a vector of 128 bits in `xmm7` from register. Inversely, it will move the register value to the central memory.
- We suppose that you have knowledge of: a) pointers b) dynamic memory allocation, c) how to read write into a file (if not we will have to cover this on the fly).