Template Matching with Cache Friendly Code

COMP273 Assignment 4 - Winter 2023

School of Computer Science, McGill University

Submit electronically via MyCourses

http://pxlcon.jimmysomething.com

Template Matching

In this assignment, you will use template matching to find Waldo in a pixel art image. Waldo is easily recognizable because of his glasses and his red and white striped shirt and hat. Template matching is the simplest of a family of algorithms that are used for optical character recognition in scanned documents, and automatic face detection in images. It will be indeed face detection you will be doing in this assignment, although with a very small template of only 8-by-8 pixels. Using such a small template will allow your assignment to run at a reasonable speed in the simulator, in contrast to the larger template shown at right. This small fixed-size template matching process is also very closely related to the motion estimation step (optical flow computation) that video compression algorithms perform on pairs of successive frames.

There are many ways to compute the error of a match, but a common choice is the sum of absolute differences of pixel intensities,

$$e(x,y) = \sum_{i=0}^{w-1} \sum_{j=0}^{h-1} \big|I(x+i, y+j) - T(i,j)\big|. \tag{1}$$

The function e is the error for a given pixel location (x, y), and I and T are functions that provide the pixel intensities in the image and template, respectively. The double sum loops over the width w and height h of the template (in our case, $w = h = 8$). The better the match, the lower the error. If template and image intensities match exactly the error will be zero.

While we could consider doing a comparison of colours, it is simpler to only work with intensities. Thus the image and template provided in this assignment will be gray-scale images. That is, when you load the bitmap image, the red, green, and blue components of the word corresponding to a pixel will all be the same value, and we **only need to read one byte** (`lbu`) to know the pixel's brightness (a number between 0 and 255). This wastes memory, using four times what is required, but is convenient because it will allow use of the bitmap display for visualization and debugging.

Provided Data and Code

Useful functions are provided for you in the `templatematch.asm` file. You will implement two different versions of the template matching function by modifying and submitting this file. Be sure to enter your name and student number at the top of the file.

In the provided file, the data section reserves space for the display buffer, an error buffer, and a template buffer. The image and error buffer regions are placed at the beginning of the static data 0x10010000 such that you can easily visualize both in the memory mapped bitmap display.


```
.data
displayBuffer:  .space 0x40000 # space for 512x256 bitmap display
errorBuffer:    .space 0x40000 # space to store match function
templateBuffer: .s[illegible]e for 8x8 template
imageFileName:  [illegible]256cropgs.raw"
templateFileName: [illegible]8gs.raw"
# struct bufferInf[illegible]t width, int height, char* filename }
imageBufferInfo:  [illegible]er  512 128  imageFileName
errorBufferInfo:  [illegible]    512 128  0
templateBufferInf[illegible]fer 8   8    templateFileName
```

As you can see here, following the space for buffers, the provided code also sets up some simple data structures to store information relating to each buffer, specifically, the address in memory where it is found, the width, the height, and the filename if there is one. It is this buffer information structure that is used as an argument to the provided functions, and to the matchTemplate and matchTemplateFast functions that you will write. Thus, if $a0 contains the address imageBufferInfo, then structure members can be loaded with offsets:


```
lw $t0 0($a0)  % load address of displayBuffer (0x10010000) into $t0
lw $t1 4($a0)  % load width of image (512) into $t1
lw $t2 8($a0)  % load height of image (128) into $t2
lw $t3 12($a0) % load address of null terminated string imageFileName (0x10090100) into $t3
```


The following functions are provided.

`void loadImage( bufferInfo* imageBufferInfo )`
Loads an image or template from file (and reads structure members much like the example above).

`(offset, score) = findBest( bufferInfo errorBufferInfo )`
Finds the best match in the error buffer and returns the **offset in bytes** along with the score.

`void highlight( bufferInfo imageBufferInfo, int offset )`
Highlights in green the area corresponding to the best match in the image.

`void processError( bufferInfo errorBufferInfo )`
Processes the error into a score between 0 (bad match) and 255 (best match) for viewing in the bitmap display. The match error at each pixel will generally be much larger than 255, and when viewed in the memory mapped display, the unprocessed error will be broken up into fields for the red, green, and blue intensities in a manner which is difficult to interpret. This function will allow you to better visualize the quality of the match across the entire image, by re-scaling . A perfect match will show up as a bright green dot in the memory mapped display.

The provided functions make the assumption that the template is always 8 by 8, and **you can make the same assumption in your code too!** However, **do not assume that the image size is fixed at 512 by 128!** Having a variable image size allows it to be reduced when measuring performance in the last part of the assignment. You might notice that the `pxlcon512x256cropgs.raw` is actually 512 by 256, but we are only loading the first half of the file.

The image and template data files are stored in a `raw` binary format. This makes for a very simple `loadImage` function, which directly loads the contents of the file into memory. These files **must be placed in the directory where you launch the MARS**, otherwise they will not be found.

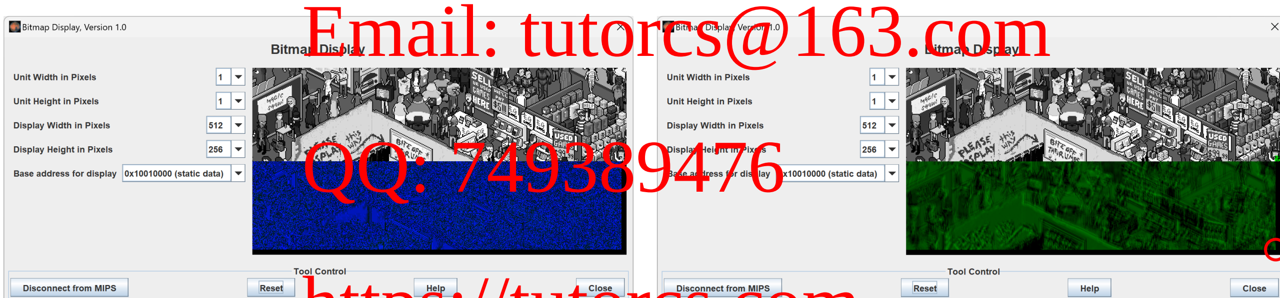
Naive Implementation (8 marks)

Complete the function `matchTemplate` in the `templatematch.asm` file. Assuming the sum of absolute differences array is initialized to zero (which is how the memory is initialized in MARS), the following pseudo-code will naively compute the sum of absolute differences, i.e., the error.

```
for ( int y = 0; y <= height - 8; y++ ) {     // loop through the image
    for ( int x =                x++ ) {
        for ( int                 {        // loop through the template image
            for (                  ++ ) {
                SA                  i][y+j] - T[i][j] );
            }
        }
    }
}
```

Note that the template is assumed to be 8 by 8, and that there will be a portion of the error buffer which is untouched because there is not a complete overlap of the template and image to test. This corresponds to the **last 7 words of each line, as well as 7 lines at the bottom of the error buffer** (notice the *less than equal* test in the for loops, instead of *less than*).

Change the template file to `template8x8gsLLtest.raw` to check that you can correctly find the lower right hand 8x8 corner of the 512x128 image using your naive implementation. Below left you is an example of what the bitmap display should look like before `processError`, and below right is what you should see as a result after the calls to `processError` and `highlight`. The red arrow points to the green highlighted region of the best match, and while hard to see, you might also notice a bright green dot in the center of the red circle.



Despite the fact that you will need to make a faster implementation of the template match in the next objective, you must still complete this part of the assignment so that you have both versions of the function to compare in the last objective of the assignment.

Faster Cache Friendly Implementation (8 marks)

The naive implementation will not work well with a small memory cache because of the inner loops which loads an entire 8 by 8 area of both the image and the template. The memory associated with an 8 by 8 area is 64 words, or 256 bytes, which means that 512 bytes of memory will be read for both template and image.

The default settings of the *data cache simulation tool* is 128 bytes. With this default cache, suppose we complete the two inner loops for the pixel at (0,0) and we advance x to the next pixel at (1,0). Our first loads to access the image and template will result in cache misses. In fact, while we will still want most of the blocks that are currently in the cache (later in the inner loops), they will all be flushed out of the cache as the least recently used blocks before the inner loop code tries to load from them.

Increasing the size of the cache can help, but there is also a very simple code modification that will help performance, even when the cache is small. Consider first (we will do better shortly) what happens when the order of the loops is changed.

```
for ( int j = 0; j < 8; j++ )
    for ( int y = 0; y <= height - 8; y++ ) {

        for ( int x = 0; x <= width - 8; x++ ) {
            for ( int i = 0; i < 8; i++ ) {
                SAD[x,y] += abs( I[x+i][y+j] - T[i][j] );
            }
        }

    }
}
```

Here, the inner loop proce[illegible]mplate pixels. Thus the two inner loops (blue) only use a very small amount of memory, s[illegible]he image, and 8 words of the template, and 8 words of the error buffer SAD, or 96 bytes. In [illegible]ache, all of these blocks *can still be* in the cache! However, the wording here is weak (*can* [illegible] *will still be*) because there could be conflict misses depending on the cache organization.

In this assignment, we will mostly use the default settings of the cache. That is, we will always assume the cache holds a total of 8 blocks, with blocks being 4 words each, for a total of 128 bytes, and we will adjust the settings to consider direct, 2 way set associative, and fully associative caches.

To improve on the loop reordering, consider also unrolling the inner most loop. It only runs 8 times, and the 8 loads of the template can be stored in registers. This speeds up your code by reducing the total instruction count (i.e., avoiding the comparison and branching for the inner loop), but more importantly reduces the number of memory accesses needed in the inner loop, which can also help reduce conflict misses!

```
for ( int j = 0; j < 8; j++ ) {
    int t0 = T[0][j];
    int t1 = T[1][j];
    int t2 = T[2][j];
    int t3 = T[3][j];
    int t4 = T[4][j];
    int t5 = T[5][j];
    int t6 = T[6][j];
    int t7 = T[7][j];
    for ( int y = 0; y <= height - 8; y++ ) {
        for ( int x = 0; x <= width - 8; x++ ) {
            SAD[x,y] += abs( I[x+0][y+j] - t0 );
            SAD[x,y] += abs( I[x+1][y+j] - t1 );
            SAD[x,y] += abs( I[x+2][y+j] - t2 );
            SAD[x,y] += abs( I[x+3][y+j] - t3 );
            SAD[x,y] += abs( I[x+4][y+j] - t4 );
            SAD[x,y] += abs( I[x+5][y+j] - t5 );
            SAD[x,y] += abs( I[x+6][y+j] - t6 );
            SAD[x,y] += abs( I[x+7][y+j] - t7 );
        }
    }
}
```

Implement this cache friendly version of the template matching loop in the `matchTemplateFast` function of the `templatematch.asm` file. This can be the algorithm described here, but you are also free to explore other ideas in implementing the algorithm in any way that improves cache performance. Advice for success: if you are trying different strategies, keep all versions of your code so that you can always fall back to your best working version. It is also a good idea to keep things simple. Please provide clear comments that document your code and optimization strategies. Finally, be sure to **read the rest of this assignment before starting this objective!**

Measuring Cache Performance and Questions (4 marks)

Once you have implemented both regular and fast versions of the template matching function, you will test your implementation and document its behaviour by tabulating performance numbers in the file `stats.csv` (see details below). Because the *data cache simulator* and the *instruction counter* tools slow down execution of your program, you will reduce the image and error buffer sizes to only 16 lines for measurements.

```
imageBufferInfo: [illegible]er  512 16   imageFileName
errorBufferInfo: [illegible]    512 16   0
```

Replace these lines **only d**[illegible]! Be sure to submit your code with a value of 128 in the height of the imageBufferInfo and [illegible]

Measure only the instru[illegible]`late` **to run**. Set one breakpoint at `jal matchTemplate` and another at the next line. Ru[illegible] breakpoint, press the *connect to MIPS* button on the instruction counter, press the run button to continue execution, then make note of the *instructions so far* when you reach the next breakpoint. Measure the instruction count for both your naive and cache friendly implementations.

Use a similar process for measuring the cache performance of `matchTemplateFast`. Record the memory accesses and cache misses for your naive and cache friendly implementations. Test with fully associative, a 2-way set associative, and direct mapped caches. Be sure to reset the cache simulator between measurements!

To make your cache friendly version work well with a direct mapped cache, you will want to consider if there are conflict misses that can be avoided.

1. Do the base addresses of image and error buffers fall into the same block of the direct mapped cache?
2. For the templateMatchFast and a direct mapped cache, does it matter if the template buffer base address falls into the same block as the image or error buffer base address?

Altering the memory layout in your data segment by adding `.space` directives as necessary to make your fast implementation work well with the direct mapped cache. At the top of your asm file, **add comment to answer the two questions above and provide a brief explanation.**

Your csv file **must exactly match the required filename and format**. Use the MARS editor to ensure you are editing the file as a plain text file using only ASCII characters (do not edit with Excel!). You may include comments in the file by starting a line with #, but otherwise there are six lines in this file to complete with six *comma separated values*, or fields, on each line. These fields consist of your student number, the test name (Naive or Fast implementation, with fully-associative (FA), 2-way set associative (2way), or Direct mapped cache), the instruction count, the number of memory accesses, the number of cache misses, and an execution time in microseconds. Here follows an example, with values shown being plausible for a correct solution!

```
# StudentID, Case, InstCnt, MemAccess, Misses, MicroSeconds
260123456, NaieveFA,     2308937, 627243, 177209, 20029
260123456, Naieve2Way,   2308937, 627243, 190466, 19277
260123456, NaieveDirect, 2308937, 627243, 246806, 26989
260123456, FastFA,       1963978, 363685,  18385,  3802
260123456, Fast2Way,     1963978, 363685,  18385,  3802
260123456, FastDirect,   1963978, 363685,  18385,  3802
```

The first entry of each line must be set to your student ID. While you will copy the numbers for entries 3 through 5 on each line from MARS tools, you must compute the time in microseconds for the last entry on each line. Assume a processor that runs at 1 GHz and executes one instruction every cycle, and assume the cache miss penalty to be 100 cycles. In the example above, the fast implementation uses fewer instructions and would be 88 μs faster if we were only counting instructions, but there are also far fewer cache misses, which can make the fast implementation almost 6.3 times faster than the naive implementation!

If you are unable to complete either of the first two objectives, you will not be able to receive full marks on this objective. To assist in marking, please either omit the file from your submission, or omit the appropriate lines in the `stats.csv` file, or include a comment line in the file to signal that either one or both of your functions is incomplete or has bugs.

Bonus (4 marks)

Bonus marks will be awar[illegible]he best direct mapped cache performance using the 100 cycle cache miss penalty describ[illegible] will be given to the top 10% of solutions that **respect register conventions and are sub[illegible]**

Submission Instru[illegible]

All work must be your owr[illegible]ed by MyCourses. **Include your name and student number in your source file**. You m[illegible]**vo files:** `templatematch.asm` and `stats.csv`. Use comments to explain any optimizations you make beyond those suggested in the assignment. **Do not use an archive**, and do not submit any of the other provided data files. Be sure to **check your submission** by downloading your submission from the server and checking that it was correctly submitted. You will not receive marks for work that is incorrectly submitted. *Note that you are encouraged to discuss assignments with your classmates, but not to the point of sharing code and answers. All code and written answers must be your own.*