Assignment #2

Low-level Programming

Due Date:	as specified on Moodle
Topics covered:	Program optimization from low-level perspective
Deliverables:	`edge.cpp`
Objectives:	Streaming SIMD Programming

Implementing the Kirsch Edge Detection Filter

The aim of this assignment is to implement an optimized version of the Kirsch Filter, which is a kind of filter used in image processing for edge detection. The purpose of edge detection is to identify points in the image where there are discontinuities or major shifts in colour change (hence detecting the "edges" of the image). This is useful for applications such as feature extraction or face detection etc. For example, the following pictures show the effect of edge detection. We show the effect of two different edge detection filters, both Sobel and Kirsch.

Before Edge Detection

Sobel Edge Detection

Kirsch Edge Detection

The BMP file format

The 24-bit colour bitmap file consists of two parts. A 54-byte header that contains information such as image height and width, and pixel array that contains the colour information of each pixel. Every row in the pixel array has the following format:

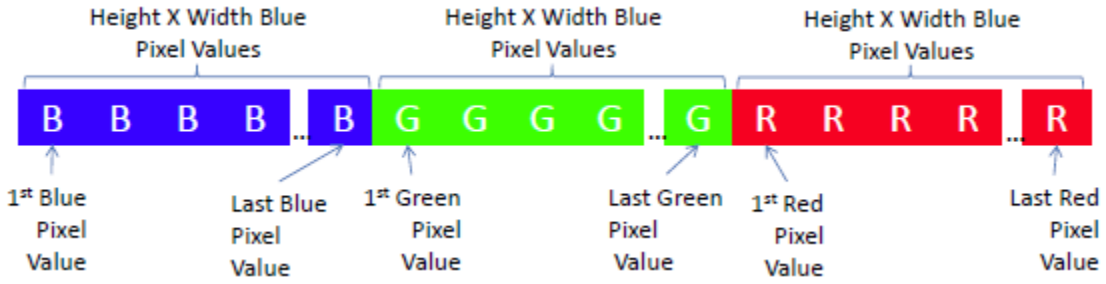
Each pixel contains 3 8-bits value for Blue, Green and Red respectively. Each of these 8-bits value should be considered as an unsigned byte ranging between 0 to 255 in values.

Different combinations of the RGB values determine the colour of the particular pixel. The number of pixels in a row is determined by the width of the picture. For example, if the picture has a width of 50 pixels, the pixel row will be $50 \times 3B + 2B = 152B$. The additional 2 bytes of padding was including to make entire row a multiple of 4 bytes. Of course, the height of a picture will determine the number of rows that we have.

However, the original format of the file is not convenient for data manipulation. We have included a function in the file bmp.c that has the following two functions:

```
void bmp_read(char *filename, bmp_header *header, unsigned char **data);
void bmp_write(char *filename, bmp_header *header, unsigned char *data);
```

The `bmp_read` takes in a file name and fill in the header information into the header object passed by reference into the function. It also allocates an array for the data variable and fills it in with a reformatted pixel information in one single array. In particular, the array is presented in the following form:



All the paddings have been removed and now instead, we have a single array instead. The bmp write function performs the reverse of the bmp read function and writes a given single array into the bitmap file with the correct format. Therefore, the following code simply copies the bitmap file from one to another:

```
bmp_header h;
unsigned char *data;

bmp_read("in.bmp", &h, &data );
bmp_write("out.bmp", &h, &data);
```

The definition of the bmp header and the functions are given in the files bmp.cpp and bmp.h.

Sobel Edge Detection Algorithm

At the heart of the Sobel edge detection algorithm lies two filter masks that are applied to each pixel.

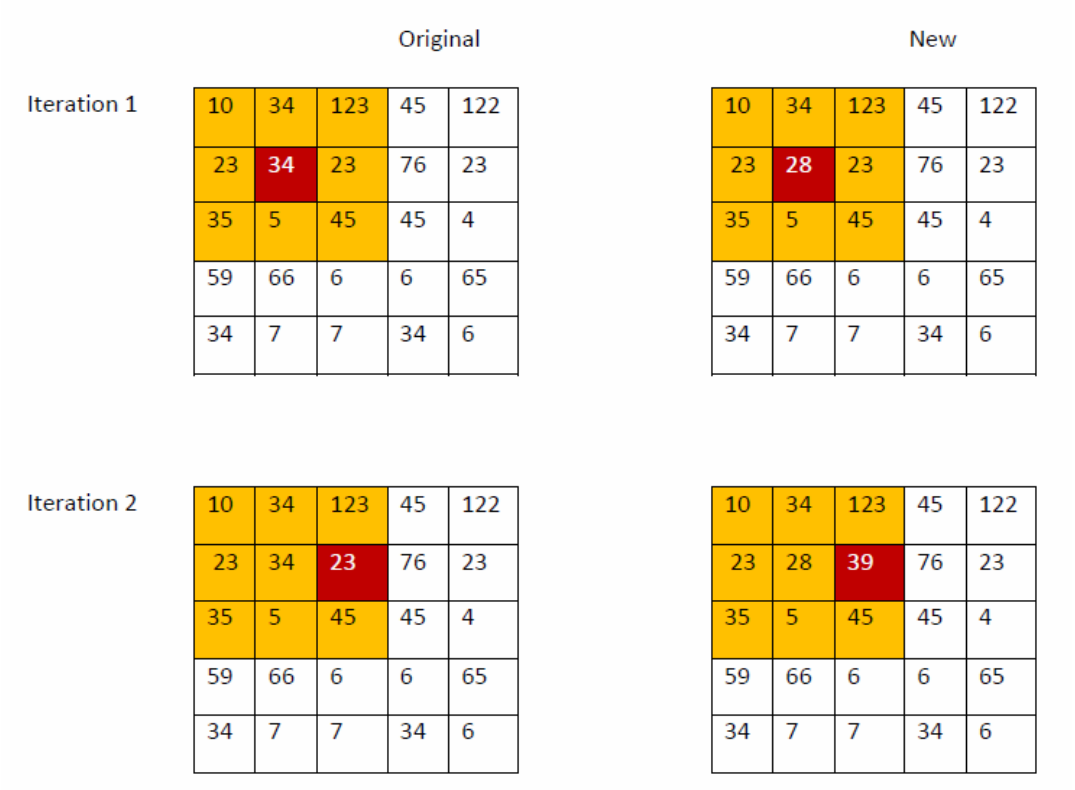
The Sobel Vertical Mask is defined by the matrix

$$Sobel_v = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}.$$

The Sobel Horizontal Mask is defined by the matrix

$$Sobel_h = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}.$$

The sobel filter works by applying both the horizontal and vertical masks as convolution matrices on every 3×3 pixels in the image. The following image shows how the algorithm works in general:



We make the following observations:

- We maintain two images, the new and the original image. Initially, the new image is the same as the original image.
- The highlighted area shows the a 3×3 pixel area, where the masks will be applied.
- After both masks are applied, a new value is obtained for the pixel. The new value is written into the new image.
- After one iteration, the masks moved row-wise to perform the same operation on the next set of pixels. When the masks moved to the end of the row, it will move one row downwards and repeat again until all pixels have been processed accordingly.

To obtain the value of the new pixel, we apply the horizontal mask as a convolution to the matrix

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & l \end{pmatrix}$$

and obtain this scalar value: $Sum_h = -a + c - (2 \times d) + (2 \times f) - g + l$. Applying the vertical mask, we will obtain $Sum_v = -a - (2 \times b) - c + g + (2 \times h) + l$. The new value is simply

$$\text{pixel}_{new} = \frac{abs(Sum_h) + abs(Sum_v)}{8}$$

. Please note that this is an integer division.

The student is given a sample code of the sobel edge detection and the improved `sobel_edge_detection_avx` as an example for students on how to optimize such an algorithm.

Kirsch Edge Detection Algorithm

At the heart of the Kirsch edge detection algorithm is one mask that is rotated 8 times and applied.

$$g^{(1)} = \begin{pmatrix} +5 & +5 & +5 \\ -3 & 0 & -3 \\ -3 & -3 & -3 \end{pmatrix}, g^{(2)} = \begin{pmatrix} +5 & +5 & -3 \\ +5 & 0 & -3 \\ -3 & -3 & -3 \end{pmatrix}, g^{(3)} = \begin{pmatrix} +5 & -3 & -3 \\ +5 & 0 & -3 \\ +5 & -3 & -3 \end{pmatrix}, g^{(4)} = \begin{pmatrix} -3 & -3 & -3 \\ +5 & 0 & -3 \\ +5 & 5 & -3 \end{pmatrix},$$

$$g^{(5)} = \begin{pmatrix} -3 & -3 & -3 \\ -3 & 0 & -3 \\ +5 & +5 & +5 \end{pmatrix}, g^{(6)} = \begin{pmatrix} -3 & -3 & -3 \\ -3 & 0 & +5 \\ -3 & 5 & 5 \end{pmatrix}, g^{(7)} = \begin{pmatrix} -3 & -3 & +5 \\ -3 & 0 & +5 \\ -3 & -3 & +5 \end{pmatrix}, g^{(8)} = \begin{pmatrix} -3 & +5 & +5 \\ -3 & 0 & +5 \\ -3 & -3 & -3 \end{pmatrix}.$$

We apply the mask to each 9 squares in a similar fashion to how we apply the two masks of the sobel filter. The difference is that we take the maximum convolution value to be the final value of the center pixel. The student is invited to consult `kirsch_operator_basic` to find out more details.

The Assignment

In the given file edge.c, we have a basic implementation of the kirsch edge detection in a function with the following prototype:

```
void kirsch_operator_basic(unsigned char *data_out,
                           unsigned char *data_in,
                           unsigned height,
                           unsigned width)
```

Please read the function and understand the code in the light of what was described in the previous section.

You are supposed to use the SSE/AVX/AVX2 SIMD instructions provided by Intel IntrinsicsGuide to optimize the basic implementation. An empty function `kirsch_edge_detection_avx` is supposed to be filled in by you. Please consider the following hints (with reference to the given sobel example):

- Using the YMM registers, it is possible to perform the computation of 8 pixel values in one iteration.

- The operations must be done at least in 16 bits to accomodate for overflow/underflow arithmetic.
- Use SSE/AVX/AVX2 intrinsics to write the code. Please refer to the intrinsics reference uploaded on Moodle. For the necessary header files for the intrinsics, please consult the documentation.
- In particular, these intrinsics are suggested for your consideration (You may choose to use any others, but here are some to help you out):
  - `__m256i` - the 256 bit integer type that can hold 16×16 bits values.
  - `_mm256_load_si2568, _mm_store_si128`. Loading and storing 256 bit variables.
  - `_mm256_sign_epi16`. An instruction for negating (or not) each component 16 bits value.
  - `__m256i _mm256_max_epi16 (__m256i a, __m256i b)` - Compare packed 16-bit integers in `a` and `b`, and store packed maximum values in `dst`.
  - `_mm256_add_epi16, _mm_sub_epi16`. Adding or subtracting all 16 bits values in 1 instruction.
  - `_mm_srli_si128` - shifting the 256-bits value right by a certain number of bits.
  - `_mm_slli_si128` - shifting the 256-bits value left by a certain number of bits.
  - `_mm256_srai_epi16` - shift each 16 bits values right by a certain number of bits.
  - `_mm256_slai_epi16` - shift each 16 bits values left by a certain number of bits.
  - `_mm256_packus_epi16` - takes the 16-bit signed integers in the 256-bit vectors `a` and `b` and converts them to a 256-bit vector of 8-bit unsigned integers. The result contains the first 8 integers from `a`, followed by the first 8 integers from `b`, followed by the last 8 integers from `a`, followed by the last 8 integers from `b`. Values that are out of range are set to 255 or 0.
  - `__mm128i _mm256_extracti128_si256(__m256i a, const int imm8)` - extract 128 bits (composed of integer data) from `a`, selected with the given `imm8`, and store the result.
  - `_mm256_set_m128i` - set packed `__m256i` vector with the supplied values.
- Please note that I did not mention any multiplication/division intrinsics. That is deliberate.
- Please note that there is no shift AVX/AVX2 instruction for 256-bit lane. You may first use `__mm128i _mm256_extracti128_si256(__m256i a, const int imm8)` to extract two 128 bits from 256 bits vector and then use bit shift and or operation instruction e.g. `_mm_or_si128, _mm_slli_si128, _mm_srli_si128` to construct two correct 128 bit values. Finally, you can use `_mm256_set_m128i` to set a 256-bit vector.
- Please start as early as you can. You may need up to 8 hours to complete this assignment.

Rubrics

This assignment will be graded over 100 points. Here is the breakdown:

- If your code fails to compile, zero is awarded immediately.
- If you did not use AVX/AVX2 SIMD instructions to implement the optimized kirsch edge detection, zero is awarded immediately.
- Comments/Code Readability. Comments are important for optimized code to enhance readability. Otherwise, optimized code cannot be easily maintained. Up to 15 points may be deducted if there is no proper comments.
- Correctness - 10 correctness test cases. You must ensure that the code works for images of all sizes, especially if the image width and height are not divisible by 16.
- Speedup - 5 efficiency test cases. At least 80% speedup on the computers in the VPL environment is required for full credit here. No credits for speedup test if your submission fails to pass the correction test 6, 7, 8, 9, 10. Speedup is defined as (time consumed by baseline C code - time consumed by SIMD c code)/(time consumed by baseline C code).