# Vectorization

D. Jiménez, E. Morancho and À. Ramírez

1516-Q2

# Index

# 1

# SIMDization/Vectorization

1. The `vector_add.c` program (at `vector_add` directory) implements the scalar and vector addition of two vectors for different data types. Compile using the Makefile we have provided you. This Makefile compiles the same program `vector_add.c` using different define flags and the compiler options `-O2 -march=native -fno-inline`.

   On one hand, in order to generate the SIMD version of a code we should compile with the `-DVECTORIAL` option. In this case we can specify three more flags: `INTRINSIC_UNALIGNED` to use the vector code using aligned load/store instrinsics, `INTRINSIC_ALIGNED` to use the vector code using unaligned load/store intrinsics, and `UNALIGNED` to indicate that we force unaligned memory alloc for the vector version. Otherwise, aligned memory is allocated and the vector code let the compiler do the vector load/store operations.

   On the other hand, compiling with `-DCHAR`, `-DSHORT` and `-DINT` options you can indicate which is the data type of the elements of the vector in the case of the scalar version, and the size of the elements of the vector register for the SIMD version. Independently of the data type, each execution performs the same number of element-to-element additions. We are using `SSE2` vector extensions.

   (a) Do `make`.

   (b) Run `script.sh` and look at the execution time `vector_add.e8, vector_add.e16, vector_add.e32` scalar versions and `vector_add_ma_compiler.v8,v16,v32` vector versions.

   (c) Have you observed any relation between the execution time evolution of the vector version and the data type size of each element? Justify the answwer.

   (d) Have you observed any relation between the execution time evolution of the scalar version and the data type size of each element? Justify the answwer.

2. We have also provided you with two different vectorization alternatives to the one of the `vector_add.c` as we commented above. With this, you should evaluate the performance difference and problems derivated from the alignment or not alignment of the allocated data.

   - Execute `make` and do timing of the `vertor_add_m*_i*.v32` versions (`script.sh` run them).
   - Reason the results you have obtained compared to the `vector_add_ma_compiler.v32` version.

3. Finally, run `vector_add.e32.3` and compare its performance to the best version of `vector_add_m*.v32` with `-O2`.

   - Explain a little bit the performance difference based on the assembler code.
   - Modify the vector code so that you can achieve equal or better performance than the scalar code version compiled with `-O3`. Hint: use unrolling and pointer induction variables.

4. In this exercise the objective is to program a matrix transposition that exploits data locality and SIMD instructions, following the idea of Eklund's Matrix transposition algorithm commented during the lecture. Please, do the following steps for a matrix size of $4096 \times 4096$:

(a) Intrument the naive matrix transposition we have provided to do timing. Note: timing instrumentation should only measure the execution time of the transpose code. In addition, it would be good to include the necessary code to check that correctness of your code.

(b) Modify the matrix transposition so that it exploits/improves the data locality. Do timing and profiling. Compute the speedup achieved compared to the previous version (only the instrumented part-transpose function).

(c) Modify previous version so that you follow the Eklund's Matrix transposition idea. Do timing and profiling. Compute the speedup achieved compared to the previous versions. This step is a intermediate step so that you can vectorize each of the submatrices using vector instructions (next step).

(d) Vectorize previous version so that you can exploit the SIMD instructions of your computer. Do timing and profiling. Compute the speedup achieved compared to the previous versions. Hint: look for transpose/shuffle/unpack, etc. instrinsics at one of the URLs given in the bibliography of lecture 6.

5. Program a vector version of your best `swap` program (lab 5).

# 2

# Image processing

Most of the vector extensions that processors have are oriented to multimedia data managment (image, video, sound).

For instance, a typical image processing operation is the saturated add, as shown below. That is, increasing a value that cannot be bigger than a certain maximum value. This operation is used, for example, when we want to increase the brightness of an image. The operation consists on increasing all the colour components (red, green, blue) of an image, being conscious that 255 is the maximum value that each component can/should reach.

```
typedef struct s_pixel_rgb {
    unsigned char r, g, b;
} t_pixel_rgb;

void
increase_brightness (t_pixel_rgb *rgb, int len, unsigned char inc, int N_iter)
{
    int i,j;

    for (j=0; j<N_iter; j++)
        for (i=0; i<len; i++) {
            if ((rgb[i].r + inc) > 255) {
                rgb[i].r = 255;
            } else {
                rgb[i].r += inc;
            }

            if ((rgb[i].g + inc) > 255) {
                rgb[i].g = 255;
            } else {
                rgb[i].g += inc;
            }

            if ((rgb[i].b + inc) > 255) {
                rgb[i].b = 255;
            } else {
                rgb[i].b += inc;
            }
        }
}
```

6. Note that in a 128-bit vector exactly fits 16 R,G and B values (`rgbrgbrgbrgbrgbr`). The first step in vectorizing function `increase_brightness` (found in file `increase_brightness.c` ) is to apply *unrolling* to the code so that 16 R,G, and B values are processed in each loop iteration.

7. As the R,G and B values are already consecutive in memory, you can load them into a 128-bit vector and perform 16 operations in parallel.

   (a) Optimize the code using the `_mm_adds_epu8()` intrinsic function in order to process 16 values (R, G and B values) in parallel.

   (b) Compare the performance of the scalar and the vector version.

   (c) Using `oprofile`, figure out how many misprediction branches are produced in the vector and scalar version. Why this difference?

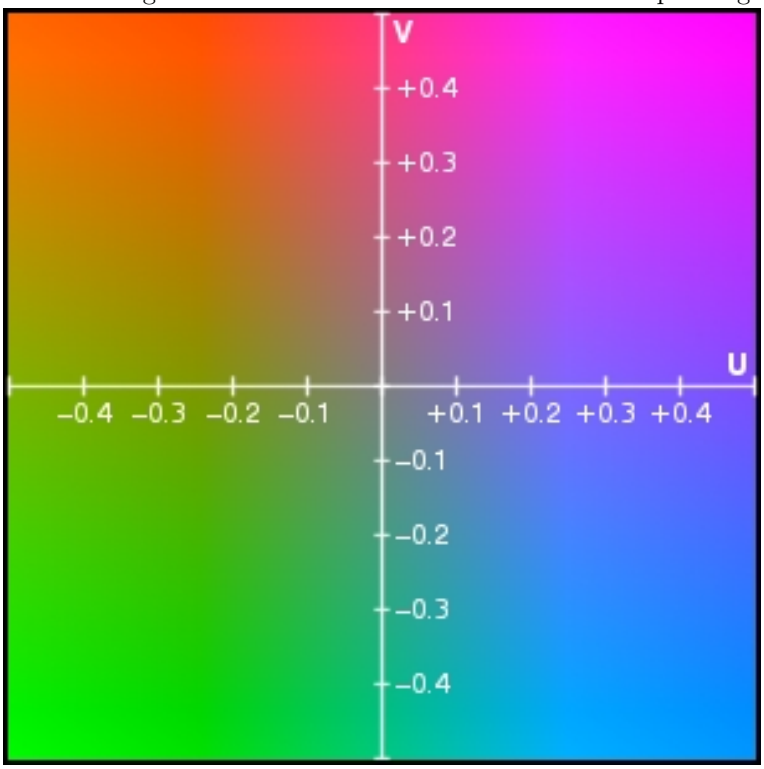
8. Now, the component R has to be processed using the following code. What do you have to do in order to vectorize that code?

```
if ((rgb[i].r < inc) {
    rgb[i].r = 0;
} else {
    rgb[i].r -= inc;
}
```

# 3

# Color Conversion

The most frequent way of codifying the colour is indicating their three colour components: Red, Green, and Blue. However, several transformations of the images are done using the YUV code. The YUV code is used in the TV signal in Europe. The Y signal, Light, indicates the gray level of the pixel (light level), and U and V signals determine the colour within the corresponding litgh plane (see figure)




In order to convert a RGB pixel to the YUV format we can use the following expressions (supposing that R,G and B are values between 0 and 1, being 0 the minimum, and 1 the maximum)

```
Y  = 0.299R + 0.587G + 0.114B
U  = 0.492 (B - Y) = - 0.147R - 0.289G + 0.436B
V  = 0.877 (R - Y) =   0.615R - 0.515G - 0.100B
```

That operation can be seen as a Matrix per vector product.

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

The following code (`rgb2yuv.c`) performs colour conversion of an image:

```
typedef struct s_pixel_rgb {
    float r, g, b;
} t_pixel_rgb;

typedef struct s_pixel_yuv {
    float y, u ,v;
} t_pixel_yuv;

void
rgb_to_yuv(t_pixel_rgb *rgb, t_pixel_yuv *yuv, int len, int N_iter)
{
    int i,j;

    for (j=0; j<N_iter; j++)
        for (i=0; i<len; i++) {
            yuv[i].y = 0.299 * rgb[i].r + 0.587 * rgb[i].g + 0.114 * rgb[i].b;
            yuv[i].u = 0.492 * (rgb[i].b - yuv[i].y);
            yuv[i].v = 0.877 * (rgb[i].r - yuv[i].y);
        }
}
```

9. Note that previous expression has a data dependency (`yuv[i].y` is used after its computation). Change the code in order to remove that data dependency. Note that this modification in the code may change the precision of the floating point operations, and it is fine for us. The code should follow the idea that it can be seen as a Matrix per vector product, in previous page.

10. Once you have removed the data dependency, unroll inner loop of the code to process 8 pixels in each iteration loop.

Once we have removed the data dependency and processed 8 pixels per iteration loop, we can use vector instructions in order to perform 8 operations in parallel. The problem is that the 8 values should be consecutive in memory (aligned if it is possible). That condition is not true in the case of the data structure we are using in this exercise.

11. Modify the data structures so that the 8 data you are processing can be consecutive in memory. Analyze if this mofication can affect the spatial locality of the program, and then its performance.

12. Optimize the performance of the code using vector instructions in order to perform 8 operations in parallel.