Sorbonne Université

SESI M2 – ProgPar

Parallel Programming – TP3: Advanced SIMD Programming

Short introduction In this session, we'll focus on writing efficient single-core code. To achieve this, we're going to use the SIMD instructions found in modern CPUs. Optimization will be focused on a classic image processing algorithm (application of a convolution matrix, 2D filtering or 2D stencil). Contrary to the Session 2, we will not use MIPP but we will directly call NEON SIMD intrinsic functions.

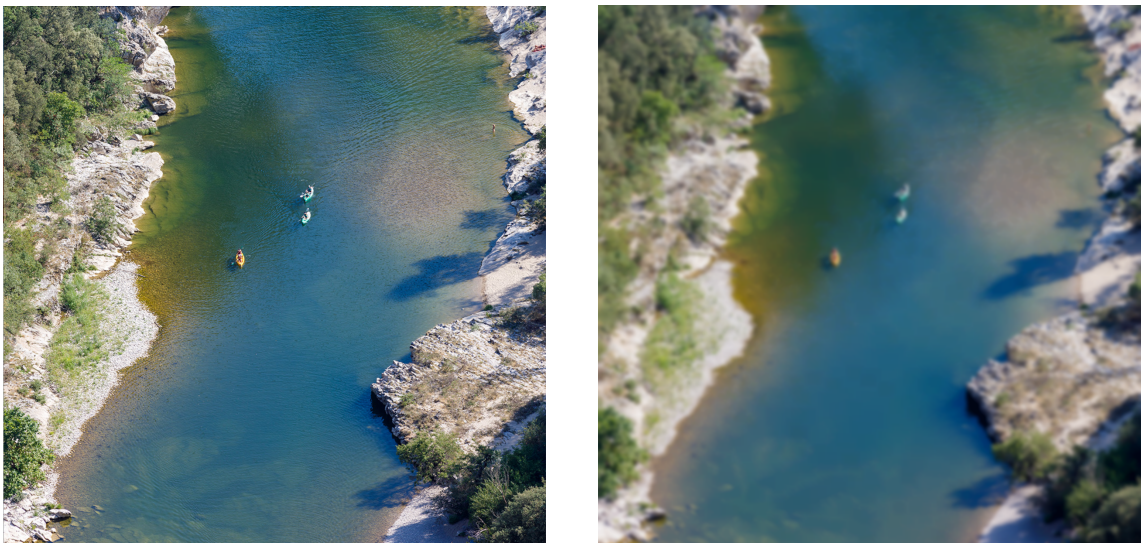
Very important, about the submission of your work At the end of this session you will have to upload your work (`blur2.c` file, and only this file!) on Moodle. After that you will have 2 weeks (until October, 15) to complete your work and update your first submission. You have to work in group of two people but each of you will have to upload the file on Moodle. Finally, please write your name plus the name of your pair at the top of the `blur2.c` file.

1 Appetizer

First, you need to update your EasyPAP **version for this session**: `git pull origin master`

1.1 `blur2` Kernel

In this session, we'll be using EasyPAP and, more specifically, we'll be work on the `blur2` kernel. This is the same kernel (`blur`) as in Session 1. Figure 1 illustrates the *kernel* `blur2`.



(a) Original image. (b) Blurred image after 20 iterations.

Figure 1: Illustration of the `blur2` kernel on images.

Code 1 presents an optimized version to compute a tile (here the size of the tile is the entire image). x and y axis of the 3×3 stencil have been unrolled. Moreover, to limit the number of loads and avoid recompute, the columns are reduced and a rotation of variables is performed on them. The computations of the borders have been separated in a dedicated `compute_borders` function (it is given to you).

Now, you can launch the kernel in EasyPAP:

```
./run -l images/1024.png -k blur2 -i 100 -wt urrot1
```

```
int blur2_do_tile_urrot1 (int x, int y, int width, int height)
{
  // loop over y (start from +1, end at -1 => no border)
  for (int i = y + 1; i < y + height - 1; i++) {
    uint16_t c_0_r = 0, c_0_g = 0, c_0_b = 0, c_0_a = 0; // col 0 -> 4 color components {r,g,b,a}
    uint16_t c_1_r = 0, c_1_g = 0, c_1_b = 0, c_1_a = 0; // col 1 -> 4 color components {r,g,b,a}

    // read 3 pixels of column 0
    unsigned c_0_0 = cur_i(i - 1, x + 0), c_1_0 = cur_i(i + 0, x + 0), c_2_0 = cur_i(i + 1, x + 0);
    // read 3 pixels of column 1
    unsigned c_0_1 = cur_i(i - 1, x + 1), c_1_1 = cur_i(i + 0, x + 1), c_2_1 = cur_i(i + 1, x + 1);

    // reduction of the pixels of column 0 (per components {r,g,b,a})
    c_0_r += ext_r(c_0_0); c_0_g += ext_g(c_0_0); c_0_b += ext_b(c_0_0); c_0_a += ext_a(c_0_0);
    c_0_r += ext_r(c_1_0); c_0_g += ext_g(c_1_0); c_0_b += ext_b(c_1_0); c_0_a += ext_a(c_1_0);
    c_0_r += ext_r(c_2_0); c_0_g += ext_g(c_2_0); c_0_b += ext_b(c_2_0); c_0_a += ext_a(c_2_0);

    // reduction of the pixels of column 1 (per components {r,g,b,a})
    c_1_r += ext_r(c_0_1); c_1_g += ext_g(c_0_1); c_1_b += ext_b(c_0_1); c_1_a += ext_a(c_0_1);
    c_1_r += ext_r(c_1_1); c_1_g += ext_g(c_1_1); c_1_b += ext_b(c_1_1); c_1_a += ext_a(c_1_1);
    c_1_r += ext_r(c_2_1); c_1_g += ext_g(c_2_1); c_1_b += ext_b(c_2_1); c_1_a += ext_a(c_2_1);

    // loop over x (start from +1, end at -1 => no border)
    for (int j = x + 1; j < x + width - 1; j++) {
      uint16_t c_2_r = 0, c_2_g = 0, c_2_b = 0, c_2_a = 0; // col 2 -> 4 color components {r,g,b,a}

      // read 3 pixels of column 2
      unsigned c_0_2 = cur_i (i - 1, j + 1);
      unsigned c_1_2 = cur_i (i + 0, j + 1);
      unsigned c_2_2 = cur_i (i + 1, j + 1);

      // reduction of the pixels of column 2 (per components {r,g,b,a})
      c_2_r += ext_r(c_0_2); c_2_g += ext_g(c_0_2); c_2_b += ext_b(c_0_2); c_2_a += ext_a(c_0_2);
      c_2_r += ext_r(c_1_2); c_2_g += ext_g(c_1_2); c_2_b += ext_b(c_1_2); c_2_a += ext_a(c_1_2);
      c_2_r += ext_r(c_2_2); c_2_g += ext_g(c_2_2); c_2_b += ext_b(c_2_2); c_2_a += ext_a(c_2_2);

      // compute the sum of all the columns 0,1,2 per components {r,g,b,a}
      uint16_t r = 0, g = 0, b = 0, a = 0;
      r = c_0_r+c_1_r+c_2_r; g = c_0_g+c_1_g+c_2_g; b = c_0_b+c_1_b+c_2_b; a = c_0_a+c_1_a+c_2_a;
      // compute the average (sum = sum / 9)
      r /= 9; g /= 9; b /= 9; a /= 9;

      // variables rotations (col0 <- col1 and col1 <- col2)
      c_0_r = c_1_r; c_0_g = c_1_g; c_0_b = c_1_b; c_0_a = c_1_a;
      c_1_r = c_2_r; c_1_g = c_2_g; c_1_b = c_2_b; c_1_a = c_2_a;

      // write the current pixel
      next_i(i, j) = rgba (r, g, b, a);
    }
  }

  // left-right borders size
  uint32_t bsize = 1;
  // compute the borders
  compute_borders(x, y, width, height, bsize);

  return 0;
}
```

Source code 1: `blur2` [unroll+reduction+rotation] kernel version 1 in EasyPAP. `ext_r`, `ext_g`, `ext_b` and `ext_a` are contractions of `extract_red`, `extract_green`, `extract_blue` and `extract_alpha`, resp. And, `cur_i` and `next_i` are contractions for `cur_img` and `next_img`, resp.

The previous command launches 100 iterations of the *kernel* `blur2`. In addition, we've specified the tile version with the option `-wt` (= `--with-tile` param). This automatically calls the optimized tile presented in Code 1.

In this tutorial, we'll write several variants of the tile function `blur2_do_tile_urrotx`. You can then compare the execution times of the different versions using the `-wt` parameter of EasyPAP.

In this session you have to compile the code with all the optimizations enabled (`-O3`).

Work to do Launch the *kernel* `blur2` (`--wt urrot1`) on your Jetson TX2, **understand the code**. Note the time for 100 iterations and without visualization (`-n` parameter). This will be the starting point for the following optimizations.

2 Optimize the Code to Reduce Execution Time

ARM provides an extensive documentation of the NEON intrinsic functions:

- `https://developer.arm.com/architectures/instruction-sets/intrinsics/`.

Feel free to use it. You can also refer to the slides of the class. Normally you should find all the required information inside.

2.1 SIMDv1 with Division by 9 on 32-bit Floats (`urrot1_simd_div9_f32`)

In this exercise, you will use ARM NEON SIMD intrinsic functions to implement a first SIMD version. There is no instruction for division on integers on ARM NEON, this is why we will need to compute the division on 32-bit floating-point numbers. Also, we will accumulate the neighbors of a pixel in 16-bit integers: **this is a mixed-precision (u8, u16, f32) implementation.**

In order to guide you, here are the main steps of the expected implementation:

1. Write the prologue code, after the `y-loop` and before the `x-loop` (you can do this at the end).
2. Read (= load) SIMD registers in 8-bit unsigned integers (`uint8x16_t`) from `cur_img` (only the pixels corresponding to the column 2).
3. Use the `vld4q_u8` instructions to perform a deinterleving of the four pixel components $\{r, g, b, a\}$, consequently you will read 4 $\times$ `uint8x16_t` at a time (they will be stored in a `uint8x16x4_t` structure). **In other word, each loop step should process 16 pixels.**
4. Promote the 8-bit components into 16-bit components to perform the accumulation (you will have 8 "`low` and `high`" registers per column (2 $\times$ the number of pixel components)).
5. Compute the reduction of the column 2 (perform the addition of the lines per component).
6. Use the "left-right" pattern to perform the sum between the columns 0, 1 and 2. You should have $2 \times 2 \times 4 = 16$ `vextq_u16` intrinsic calls (for low, high, column 0, column 2 and each time for the 4 pixel components).
7. First, promote the results to `uint32x4_t` registers. Then, convert the `uint32x4_t` registers into `float32x4_t` registers.
8. Perform the division by 9 on 32-bit floating-point.
9. Convert back the `float32x4_t` resulting registers into `uint32x4_t` registers.
10. Convert back the `uint32x4_t` registers into `uint16x8_t` registers.
11. Convert back the `uint16x8_t` registers into `uint8x16_t` registers.
12. Write (= store) `uint8x16x4_t` SIMD registers in `next_img` with interleaving (`vst4q_u8`).
13. Perform the columns rotation: $c0 \leftarrow c1$ and $c1 \leftarrow c2$ (on 16-bit registers).

Steps 2 to 13 are made in the `x-loop`. **This is an important task! Do it carefully and try to organize your code well.** Rather than using each time 4 registers for the 4 pixel components, prefer to use the `x4_t` structures (`uint8x16x4_t`, `uint16x8x4_t`, `uint32x4x4_t` and `float32x4x4_t`) offered by ARM NEON.

For debugging purpose, some functions are given to you to print the lanes of the SIMD registers:

- `print_reg_u8`,
- `print_reg_u16`,
- `print_reg_u32`,
- `print_reg_f32`.

Do not forget that a generic `compute_borders` function is given to you. In SIMD you will have larger border than 1 on the left and on the right of the image. You can specify the left and right borders size you want in the `compute_borders` function.

Work to do Implement the previous pseudo code in SIMD. Visually check that the code still works when you run it. Compare the execution time with previous versions. What do you notice? How do you explain it?

2.2 SIMDv1 with Division by 9 on 16-bit Integers (`urrot1_simd_div9_u16`)

It is true that there is no SIMD intrinsic function to compute division by 9 on integers in NEON. However, it is possible to emulate this operation from other elementary operations (see Code 2).

```
1 uint16_t vdiv9_u16(const uint16_t n) {
2   uint16_t q1 = n - (n >> 3);
3            q1 += (q1 >> 6);
4   uint16_t q2 = q1 >> 3;
5   uint16_t  r = n - (q1 + q2);
6             r = q2 + ((r + 7) >> 4);
7   return r;
8 }
```

Source code 2: Emulated implementation of the 16-bit integer division by 9. Optimized for $n < 4096$.

Work to do Implement a new SIMD version with the integer division by 9. Visually check that the code still works when you run it. Compare the execution time with previous versions. What do you notice? How do you explain it?

2.3 SIMDv2 with Division by 9 on 32-bit Floats (`urrot2_simd_div9_f32`)

An alternative version (called `urrot2`, or version 2) is given in Code 3. In this version 2, there is **no more per-component extractions** (`extract_*` and `rgba`). All the components are stores in small arrays of 4 elements. Additional `for-loops-4` perform operations component-wise.

Work to do #1 Compare this new kernel (`--wt urrot2`) with the previous version (`--wt urrot1`), **understand the code**. Note the time for 100 iteration and without visualization (`-n` parameter). This will be the starting point for the following optimizations.

You will need to implement a new SIMD version starting from the given `urrot2` code. Here are the steps you need follow (**in bold** what has changed since version 1):

1. Write the prologue code, after the `y-loop` and before the `x-loop` (you can do this at the end).

```c
int blur2_do_tile_urrot2 (int x, int y, int width, int height)
{
  // small arrays (4 elements each) to store all the components of 1 pixel
  uint8_t c_0_0[4], c_1_0[4], c_2_0[4];
  uint8_t c_0_1[4], c_1_1[4], c_2_1[4];
  uint8_t c_0_2[4], c_1_2[4], c_2_2[4];

  // loop over y (start from +1, end at -1 => no border)
  for (int i = y + 1; i < y + height - 1; i++) {
    uint8_t* cur_i_ptr;
    // load the 3 pixels of the column 0 (no extraction of the components)
    cur_i_ptr = (uint8_t*)&cur_i(i - 1, x + 0); for(int c=0;c<4;c++) c_0_0[c] = cur_i_ptr[c];
    cur_i_ptr = (uint8_t*)&cur_i(i + 0, x + 0); for(int c=0;c<4;c++) c_1_0[c] = cur_i_ptr[c];
    cur_i_ptr = (uint8_t*)&cur_i(i + 1, x + 0); for(int c=0;c<4;c++) c_2_0[c] = cur_i_ptr[c];

    // load the 3 pixels of the column 1 (no extraction of the components)
    cur_i_ptr = (uint8_t*)&cur_i(i - 1, x + 1); for(int c=0;c<4;c++) c_0_1[c] = cur_i_ptr[c];
    cur_i_ptr = (uint8_t*)&cur_i(i + 0, x + 1); for(int c=0;c<4;c++) c_1_1[c] = cur_i_ptr[c];
    cur_i_ptr = (uint8_t*)&cur_i(i + 1, x + 1); for(int c=0;c<4;c++) c_2_1[c] = cur_i_ptr[c];

    // column 0 and column 1 reduction
    uint16_t c_0[4], c_1[4];
    for(int c=0;c<4;c++) c_0[c] = c_0_0[c] + c_1_0[c] + c_2_0[c];
    for(int c=0;c<4;c++) c_1[c] = c_0_1[c] + c_1_1[c] + c_2_1[c];

    // loop over x (start from +1, end at -1 => no border)
    for (int j = x + 1; j < x + width - 1; j++) {
      // load the 3 pixels of the column 2 (no extraction of the components)
      cur_i_ptr = (uint8_t*)&cur_i(i - 1, j + 1); for(int c=0;c<4;c++) c_0_2[c] = cur_i_ptr[c];
      cur_i_ptr = (uint8_t*)&cur_i(i + 0, j + 1); for(int c=0;c<4;c++) c_1_2[c] = cur_i_ptr[c];
      cur_i_ptr = (uint8_t*)&cur_i(i + 1, j + 1); for(int c=0;c<4;c++) c_2_2[c] = cur_i_ptr[c];

      // column 2 reduction
      uint16_t c_2[4] = {0, 0, 0, 0};
      for(int c=0;c<4;c++) c_2[c] += c_0_2[c] + c_1_2[c] + c_2_2[c];

      // add column 0, 1 and 2 and compute the avg (div9)
      uint16_t avg[4] = {0, 0, 0, 0};
      for(int c=0;c<4;c++) avg[c] = (c_0[c] + c_1[c] + c_2[c]) / 9;

      // variables rotations
      for(int c=0;c<4;c++) c_0[c] = c_1[c];
      for(int c=0;c<4;c++) c_1[c] = c_2[c];

      // store the resulting pixel (no need for the 'rgba' function)
      uint8_t* next_i_ptr = (uint8_t*)&next_i (i, j);
      for(int c=0;c<4;c++) next_i_ptr[c] = avg[c];
    }
  }

  // left-right borders size
  uint32_t bsize = 1;
  // compute the borders
  compute_borders(x, y, width, height, bsize);

  return 0;
}
```

Source code 3: `blur2` [unroll+reduction+rotation] kernel version 2 in EasyPAP. `cur_i` and `next_i` are contractions for `cur_img` and `next_img`, resp.

2. Read (= load) SIMD register in 8-bit unsigned integers (`uint8x16_t`) from `cur_img` (only the pixels corresponding to the column 2).
3. **Use the `vld1q_u8` instructions to load 16 elements (= 4 pixels) at a time (they will be stored in a `uint8x16_t` register). In other words, each loop step should process 4 pixels.**
4. Promote the 8-bit components into 16-bit components to perform the accumulation (**you will have only two "`low` and `high`" registers per column**).
5. Compute the reduction of the column 2 (perform the addition of the lines per component).
6. Use the "left-right" pattern to perform the sum between the columns 0, 1 and 2. **You should have $2 \times 2 = 4$ `vextq_u16` intrinsic calls (for low, high, column 0 and column 2)**.
7. First, promote the results to `uint32x4_t` registers. Then, convert the `uint32x4_t` registers into `float32x4_t` registers.
8. Perform the division by 9 on 32-bit floating-point.
9. Convert back the `float32x4_t` resulting registers into `uint32x4_t` registers.
10. Convert back the `uint32x4_t` registers into `uint16x8_t` registers.
11. Convert back the `uint16x8_t` registers into `uint8x16_t` registers.
12. **Write (= store) `uint8x16_t` SIMD register in `next_img` (`vst1q_u8`).**
13. Perform the columns rotation: $c0 \leftarrow c1$ and $c1 \leftarrow c2$ (on 16-bit registers).

Steps 2 to 13 are made in the `x-loop`.

Work to do #2 Implement a new SIMD version with the floating-point division by 9. Visually check that the code still works when you run it. Compare the execution time with previous versions. What do you notice? How do you explain it?

2.4 SIMDv2 with Division by 9 on 16-bit Integers (`urrot2_simd_div9_u16`)

Work to do Implement a new SIMD version with the integer division by 9 (see Section 2.2). Visually check that the code still works when you run it. Compare the execution time with previous versions. What do you notice? How do you explain it?

2.5 SIMDv2 with Division by 8 on 16-bit Integers (`urrot2_simd_div8_u16`)

The division by 9 generates, in any cases, a lot of extra-instructions. In this exercise we consider to modify a little bit the code of the `blur2` 3×3 stencil algorithm to be more **SIMD-friendly**. Indeed, we propose to compute the **average of 8 pixels** instead of 9. We will ignore the centered pixel. Thus, we can perform a **division by 8** instead of a division by 9...

However, during the column reduction, we are accumulating the centered pixel. In this new version, we need to **subtract the centered pixel** in the last sum before to divide by 8. **An additional rotation of the centered pixel** is required to perform this subtraction.

Work to do Implement a new SIMD version with the integer division by 8. Visually check that the code still works when you run it. Compare the execution time with previous versions. What do you notice? How do you explain it?