Sorbonne Université – SESI M2

MU5IN160 – Parallel Programming

Hands-on Session 2 – SIMD Programming

Short introduction In this tutorial, we'll focus on writing efficient single-core code. To achieve this, we're going to use the SIMD instructions found in modern CPUs. In addition, we'll apply some source code source code transformations as in Session 1. Optimization will focus on a compute-intensive algorithm. This allow us to address computational optimization without the limitations from memory throughput. Keep in mind that SIMD instructions can be used in many other application domains.

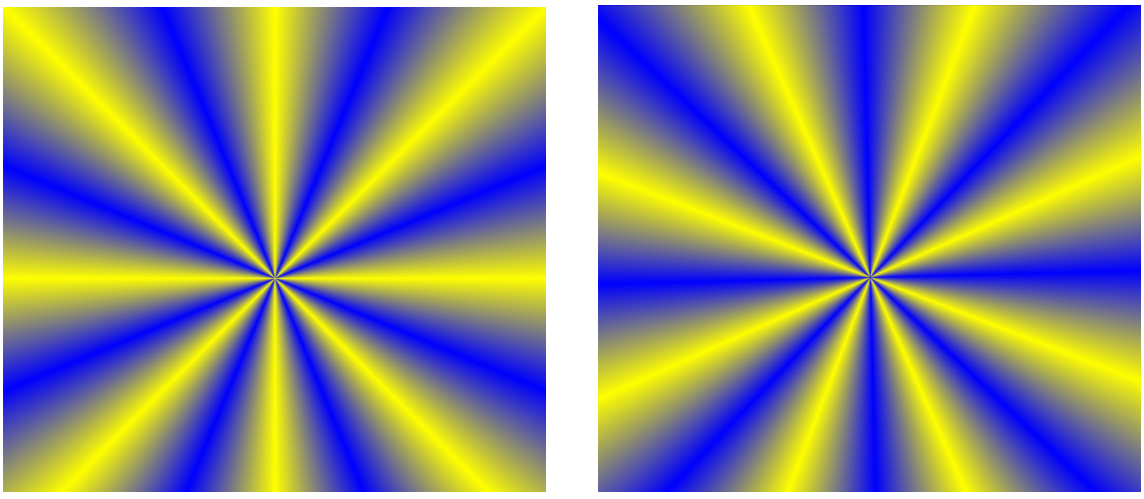
Very important, about the submission of your work At the end of this session you will have to upload your work (`spin.cpp` file, and only this file!) on Moodle. After that you will have 2 weeks (until October, 8) to complete your work and update your first submission. You have to work in group of two 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 `spin.cpp` file.

1 Appetizer

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

1.1 `spin` Kernel

In this session, we'll be using EasyPAP and, more specifically, we'll be work on the `spin` kernel. This kernel has the effect of rotating the image as it is iterated. Each pixel, depending on its position in the image and an angle angle, is assigned a color between yellow and blue. The angle is is common to all pixels and is incremented by one degree for each iteration. Figure 1 illustrates the *kernel* `spin`.



(a) Original image. (b) 25 degrees rotation (25 iterations).

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

Code 1 presents the current implementation of the `spin` algorithm. As you can see, the `spin_compute_seq` function is called first (this is the entry point for the EasyPAP kernel). Then, the `compute_color` and `rotate` functions are called.
You can now launch the kernel in EasyPAP:

```
./bin/easypap -k spin -i 100 -v seq
```

```
// Computation of one pixel
static unsigned compute_color(int i, int j) {
  float atan2f_in1 = (float)DIM / 2.f - (float)i;
  float atan2f_in2 = (float)j - (float)DIM / 2.f;
  float angle = atan2f(atan2f_in1, atan2f_in2) + M_PI + base_angle;
  float ratio = fabsf((fmodf(angle, M_PI / 4.f) - (M_PI / 8.f)) / (M_PI / 8.f));
  int r = color_a_r * ratio + color_b_r * (1.f - ratio);
  int g = color_a_g * ratio + color_b_g * (1.f - ratio);
  int b = color_a_b * ratio + color_b_b * (1.f - ratio);
  int a = color_a_a * ratio + color_b_a * (1.f - ratio);
  return rgba(r, g, b, a);
}

static void rotate(void) {
  base_angle = fmodf(base_angle + (1.f / 180.f) * M_PI, M_PI);
}

unsigned spin_compute_seq(unsigned nb_iter) {
  for (unsigned it = 1; it <= nb_iter; it++) {
    for (unsigned i = 0; i < DIM; i++)
      for (unsigned j = 0; j < DIM; j++)
        cur_img(i, j) = compute_color(i, j);

    rotate(); // Increase the base angle (+ 1 degree)
  }

  return 0;
}
```

Source code 1: `spin` kernel in EasyPAP.

The previous command launches 100 iterations of the *kernel* `spin`. In addition, we've specified the version with the option `-v` (= `--version` param). This automatically calls `spin_compute_seq` line 18 in Code 1.
In this tutorial, we'll write several variants of the function `spin_compute_seq`. You'll be asked to copy and paste the same function and replace the end of the name (here `seq`) with the new name. You can then compare the execution times of the different versions using the `-v` parameter of EasyPAP.
The processing carried out by the *kernel* `spin` involves notions of trigonometry. You can see this in the calls to the function `atan2f` (defined in the header file `math.h`).
As in the previous session, the different components of a pixel are red, green, blue and the alpha channel (= opacity). Each of these components is stored on 8-bit. All these components are grouped together in a single 32-bit integer. The `rgba` function produces a pixel in the final image to be displayed (line 11 of Code 1).

Work to do Launch the *kernel* `spin` on your machine, understand the code.

1.2 Execution Time according to Optimization Options

The EasyPAP code compilation is defined in the `Makefile` file at the root. Compilation options are defined on line 28:

```
CFLAGS := -O3 -march=native
```

As you can see, the `-O3` option is present.

Work to do #1 Compile the code with the different optimization levels available (`-O0`, `-O1`, `-O2` and `-O3`) then compare and record the different execution times for 100 iterations. To see the execution time, use the `--no-display` option option in EasyPAP.

Work to do #2 As the *kernel* calls on various mathematical functions (`atan2f`, `fmodf` and `fabsf`), it's interesting to see the impact of the `-ffast-math` optimization. As a reminder, this optimization sacrifices the precision of floating-point calculations in favor of performance. Recompile your code by combining the different levels of optimization with the `-ffast-math` option. What do you observe?

For the rest of this session, use the optimization level "three" with the *fast math* option enabled (`-O3 -ffast-math` = all optimizations).

2 Optimize the Code to Reduce Execution Time

2.1 Approximation of Mathematical Functions (`approx`)

Work to do Starting from the initial version of the code (version `seq`), write a new version `approx` in which you replace the calls to the mathematical functions with the approximations given in Code 2. Test your code with EasyPAP. What do you notice? How do you explain it?

2.2 Preparing for Code Vectorization (`simd_v0`)

Now you can start vectorizing the code. To do this, you'll use the MIPP SIMD library. You don't need to install it, it's already done for you. However, you can (and should!) refer to the documentation available in the *readme* : `https://github.com/aff3ct/MIPP#list-of-mipp-functions`. The latter details all SIMD operations available through the library.

To help you get started, the code for the function `spin_compute_simd_v0` is given to you. Take your time to understand it. It's **VERY IMPORTANT** to see that the loop on line 9 of the Code 3 is different from the initial code. Indeed, the pitch of the mouth is `mipp::N<float>()`, the number of elements in a vector register, instead of 1 initially.
You'll notice that the prototype of the `compute_color_simd_v0` function (line 1 of Code 3) is different from the scalar implementations. This is normal, as from now on we'll be working with vector registers:

```
// arctangent, result between -pi/2 et pi/2 radians
static float atanf_approx(float x) {
  return x * M_PI / 4.f + 0.273f * x * (1.f - fabsf(x));
}
// arctangent, result between -pi et pi radians
static float atan2f_approx(float y, float x) {
  float ay = fabsf(y);
  float ax = fabsf(x);
  int invert = ay > ax;
  float z = invert ? ax / ay : ay / ax; // [0,1]
  float th = atanf_approx(z);           // [0,pi/4]
  if (invert) th = M_PI_2 - th;         // [0,pi/2]
  if (x < 0) th = M_PI - th;            // [0,pi]
  if (y < 0) th = -th;
  return th;
}
// remainder of floating-point division
static float fmodf_approx(float x, float y) {
  return x - trunc(x / y) * y;
}
```

Source code 2: `atanf`, `atan2f` and `fmodf` function approximations.

```
mipp::Reg<int> compute_color_simd_v0(mipp::Reg<int> r_i, mipp::Reg<int> r_j) {
  // TODO !
}

unsigned spin_compute_simd_v0(unsigned nb_iter) {
  int tab_j[mipp::N<int>()];
  for (unsigned it = 1; it <= nb_iter; it++) {
    for (unsigned i = 0; i < DIM; i++)
      for (unsigned j = 0; j < DIM; j += mipp::N<float>()) {
        for (unsigned jj = 0; jj < mipp::N<float>(); jj++) tab_j[jj] = j + jj;
        int* img_out_ptr = (int*)&cur_img(i, j);
        mipp::Reg<int> r_result =
            compute_color_simd_v0(mipp::Reg<int>(i), mipp::Reg<int>(tab_j));
        r_result.store(img_out_ptr);
      }
    rotate();
  }
  return 0;
}
```

Source code 3: Implémentation de la fonction `spin_compute_simd_v0`.

`mipp::Reg<int>` instead of `int` and `mipp::Reg<float>` instead of `float`. At present, MIPP doesn't support unsigned integers, so you'll have to **ABSOLUTELY AVOID** using `mipp::Reg<unsigned>`.

Work to do #1 Implement the `compute_color_simd_v0` function. To do this, in this first implementation, write a loop that iterates over the elements of the input vector registers (`r_i` and `r_j`). Each element of a vector register can be individually accessed using the `[]` operator (as for an array). The computation inside the loop will be sequential for the moment (the same as in the `compute_color_approx` function implemented earlier). You'll save the n pixels in an array of integers of size `mipp::N<int>()`, then return a register that you'll initialize by loading the previous array (see line 13 of Code 3 for loading an array).

Work to do #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?

2.3 "Et zé bartiii" for the Vectorization (`simd_v1`)

Work to do #1 Write a SIMD version of the remainder of the division in the function `fmodf_approx_simd`.

Work to do #2 Starting from the version `compute_color_simd_v0` you wrote earlier, implement a new version `compute_color_simd_v1` that calls `fmodf_approx_simd`.

Work to do #3 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?

Tips #1 The loop that iterates over the register elements must be split in two. elements into two, with only one call to the function `fmodf_approx_simd`.

Tips #2 Some useful SIMD MIPP functions:

- `mipp::trunc`,
- `mipp::abs`,
- `mipp::sub` (or operateur `-`),
- `mipp::mul` (or operateur `*`),
- `mipp::div` (or operateur `/`).

2.4 "One more step" in Vectorizing (`simd_v2`)

Work to do #1 Write a SIMD version of the `rgba` function function in the `rgba_simd` function. As a reminder, the implementation of the function can be found in the file `include/img_data.h`.

Work to do #2 Starting from the `compute_color_simd_v1` you wrote earlier, implement a new version `compute_color_simd_v2` that calls on `rgba_simd`.

Work to do #3 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?

Tips #1 The second `for`-loop in the function `compute_color_simd_v1` has to disappear in favor of SIMD-only instructions in the new version `compute_color_simd_v2`.

Tips #2 Some useful SIMD MIPP functions:

- `mipp::lshift` (or operateur `<<`),
- `mipp::orb` (or operateur `|`),
- `mipp::add` (or operateur `+`),
- `mipp::cvt<float, int>` (to convert a floating vector register to an integer vector register).

2.5 "À pieds joints" into the Vectorization (`simd_v3`)

Work to do #1 Write SIMD versions of the `atanf_approx` and `atan2f_approx` functions in the `atanf_approx_simd` and `atan2f_approx_simd` functions.

Work to do #2 Starting from the version `compute_color_simd_v2` you wrote earlier, implement a new version `compute_color_simd_v3` that calls `atan2f_approx_simd`. With this version, you should no longer have a loop in the `compute_color_simd_v3` function.

Work to do #3 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?

Tips Some useful SIMD MIPP functions:

- `mipp::Msk<mipp::N<float>()>` (declaration of a "mask vector register"),
- `mipp::blend` (implementation of a ternary condition, or select in SIMD),
- `mipp::cvt<int, float>` (to convert an integer vector register into a floating-point vector register).

2.6 "À fond les ballons" into the Vectorisation (`simd_v4`)

Work to do #1 Starting from the previous version, write a new version `compute_color_simd_v4` in which you completely inline the function call `atan2f_approx_simd`. In other words, there must be no more calls to `atanf_approx_simd` and `atan2f_approx_simd` in the `compute_color_simd_v4` function.

Work to do #2 Some registers containing constants can be eliminated, delete duplicates.

Work to do #3 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?

2.7 "Ça pique les yeux" the Vectorization (`simd_v5`)

Work to do #1 Starting from the previous version, write a new version of `spin_compute_simd_v5`, in which you completely inline the `compute_color_simd` function call.

Work to do #2 Take the constants out of the loops, and check whether some computations are not made too many times. For instance, some computations don't depend on `j`... You'll also need to find multiplication and addition sequences and replace them with FMA-type operations (*Fused Multiply and Add*). This type of operations is now widely used in SIMD instruction sets, and theoretically double the efficiency.

Work to do #3 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?

Tips Some useful SIMD MIPP functions:

- `mipp::fmadd(a,b,c)` (FMA: $a \times b + c$),
- `mipp::fnmadd(a,b,c)` (FMA: $-(a \times b) + c$).

2.8 "Cerise (de Groupama) sur le gâteau", unrolling (`simd_v6`)

In this ultimate version, we're going to unroll the loop on the `j`. But before we get too carried away, it's important to put things in order, otherwise you won't make it... Go back to version 5, and copy/paste for version 6. Postfix all your variables that depend on `j` with `_j0`. For example, if you have a vector register `mipp::Reg<float> r_atan2f_in2` you need to rename it to `mipp::Reg<float> r_atan2f_in2_j0`.

Work to do #1 Starting from the previous version, write a new version `spin_compute_simd_v6u2` in which you will perform a 2-order unrolling on `j`.

Work to do #2 Starting from the previous version, write a new version `spin_compute_simd_v6u4` in which you will perform a 4-order unrolling on `j`.

Work to do #3 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?