Sorbonne Université

SESI M2 – ProgPar

Parallel Programming – TP1: Single-core CPU Optimizations

Short introduction In this hands-on session, we'll be focusing on writing efficient single-core CPU code. To achieve this, we'll apply source code transformations to take advantage of modern CPU architecture. Optimization will be focused on a classic image processing algorithm (application of a convolution matrix, 2D filtering or 2D stencil). Keep in mind that these same optimizations can be transposed to other application domains.

Very important, about the submission of your work At the end of this session you will have to upload your work (`blur.c` file, and only this file!) on Moodle. After that you will have 2 weeks (until October, 1) 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 `blur.c` file.

1 Appetizer

1.1 Reminder of Compiler Optimization Options

The compiler transforms the source code (which you write) into a machine-readable binary. This step is necessary to run your code on a CPU. In addition to translating source code into binary, **the compiler can also optimize the code to reduce runtime execution time and energy consumption**. To achieve this, **the compiler provides optimization options**. The best-known start with `-O` and define a global optimization level:

- `-O0` No optimization,
- `-O1` Enables a series of optimizations to reduce binary size and execution time, while keeping compilation time relatively low,
- `-O2` Enables all possible optimizations (except those requiring a compromise between efficiency and binary size), this option requires a longer compilation time than `-O1`,
- `-O3` Enables all possible optimizations that don't compromise computation accuracy, compilation time is increased compared to `-O2`.

These different levels of optimization are valid for GNU compilers (`gcc`) and Clang compilers. They may vary depending on the compiler you use. so it's up to the developer to be familiar with his compiler. For more details on the various compiler optimizations compiler optimizations, go to `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

1.2 `blur` Kernel

In this session we'll be using EasyPAP and, more specifically, we'll work on the `blur` kernel. This kernel has the effect of blurring an image. It's an algorithm often found in image processing software such as Photoshop. The Figure 1 illustrates the processing we'll be applying of this tutorial.
You can download an images bank compatible with EasyPAP here: `http://raymond.namyst.emi.u-bordeaux.fr/no_link/EasyPAP/images.zip`.

Code 1 shows the current implementation of the `blur` kernel. As you can see, the `blur_compute_seq` function is called first (this is the entry point for the EasyPAP kernel), then the `blur_do_tile_default` function is called at each iteration.
You can now launch the kernel in EasyPAP:

```
./bin/easypap -k blur --load-image images/1024.png -i 100 -v seq -wt default
```

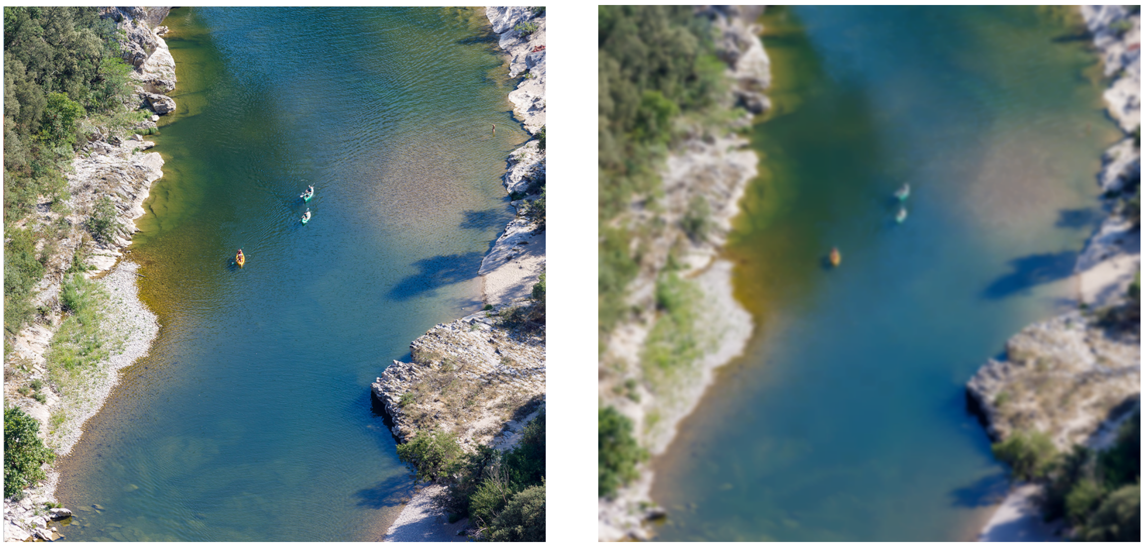
(a) Original image. (b) Blurred image after 20 iterations.

Figure 1: Illustration of the `blur` kernel on an image.

```c
unsigned blur_compute_seq (unsigned nb_iter) {
  for (unsigned it = 1; it <= nb_iter; it++) {
    do_tile (0, 0, DIM, DIM, 0);
    swap_images ();
  }
  return 0;
}

int blur_do_tile_default (int x, int y, int width, int height) {
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++) {
      unsigned r = 0, g = 0, b = 0, a = 0, n = 0;
      int i_d = (i > 0) ? i - 1 : i;
      int i_f = (i < DIM - 1) ? i + 1 : i;
      int j_d = (j > 0) ? j - 1 : j;
      int j_f = (j < DIM - 1) ? j + 1 : j;
      for (int yloc = i_d; yloc <= i_f; yloc++)
        for (int xloc = j_d; xloc <= j_f; xloc++) {
          unsigned c = cur_img (yloc, xloc);
          r += extract_red (c);
          g += extract_green (c);
          b += extract_blue (c);
          a += extract_alpha (c);
          n += 1;
        }
      r /= n;
      g /= n;
      b /= n;
      a /= n;
      next_img (i, j) = rgba (r, g, b, a);
    }
  return 0;
}
```

Code source 1: Source code of the `blur` kernel in EasyPAP.

It presupposes that you have extracted the `images.zip` file from the EasyPAP root. The previous command run the `blur` kernel over 100 iterations. Moreover, we specified the tile type with the `-wt` option (or `--with-tile`). It allows to automatically replace the `do_tile` call line 3 in the Code 1 by the function entitled `blur_do_tile_default` line 9. Note that EasyPAP do that for you.

Important Notes for Jetson TX2
To run the code on a Denver 2 core you can do:

```
taskset -c 1 ./bin/easypap -k blur --load-image images/1024.png -i 100 -v seq -wt default
```

And to run the code on a Cortex-A57 core you can do:

```
taskset -c 0 ./bin/easypap -k blur --load-image images/1024.png -i 100 -v seq -wt default
```

By default the code run on a Denver 2 core! Be careful, you could have some surprising results on Denver 2 cores... Remember that there is an hardware code translator that can perform code simplifications on these cores...

In this session, we'll write several variants of the `blur_do_tile_default` function. To do this, you'll be asked to copy and paste the same function, replacing the end of the name (in this case `default`) with the new name. You can then compare the execution times of the different versions using EasyPAP's `-wt` parameter.

The processing performed by the `blur` kernel is an average of the 9 pixels surrounding a pixel (the surrounded pixel is included in the average). This has the effect of diffusing the color with a 3x3 neighborhood, resulting in a blurred effect. The same process is applied to each pixel in the image, and the newly calculated pixel is stored in the next image. The more you iterate, the more blurred the image becomes.

The different components of a pixel are red, green, blue and the alpha channel (= opacity). Each of these components is stored on 8 bits. All these components are grouped together in a single 32-bit integer. The `extract_red`, `extract_green`, `extract_blue`, `extract_alpha` and `rgba` functions enable you to easily manipulate the different components of a pixel.

Work to do You will find the source code of the `blur` kernel in `kernel/c/blur.c`. Run `blur` kernel on your Jetson TX2 and understand the code.

1.3 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 Re-read the Section 1.1, compile the code with the different levels of optimization available, then compare and note the different execution times for 100 iterations. To see the execution time, use the `--no-display` option in EasyPAP. Note the results as comments above the `blur_do_tile_default` in the `blur.c` file.

For the remainder of this session, you should use optimizations level "zero" (`-O0` = no compiler optimization), to better see the impact of the various code transformations we'll be performing. At the end, AND ONLY AT THE END, you'll be able to compare execution times with higher optimization levels.

2 Optimize the Code to Reduce Execution Time

2.1 `blur` Kernel WITHOUT Border (`default_nb`)

In Code 1, pixels at the borders of the image are computed differently. Indeed, at the borders of the image, it's not possible to average over 9 pixels, as some pixels simply don't exist. This computation of pixels at the borders is provided by lines 13-16 where limits are computed.

Work to do To simplify future transformations, you need to write a `default_nb` version (`nb` for *no border*) in which there will be a simpler treatment for borders: a simple copy of the previous pixel. This will simplify the body of the computational loop (between L10 and L31) and lines 13-16 should disappear. Test your code with EasyPAP, you should clearly see that the borders don't change and there should be no blinking.

2.2 Loop Unrolling on X-axis (`optim1`)

Work to do Starting from the `default_nb` version you wrote before, write a new version of the code (which you'll call `optim1`) that manually unroll loop line 18. In other words, the L18 loop should disappear. Compare execution times between the versions `default_nb` and `optim1`. What do you notice? How do you explain it?

No improvement? Did you try on a Cortex-A57 core?

2.3 Loop Unrolling on Y-axis (`optim2`)

Work to do Starting from the `optim1` version written before, write a new version of the code (which you'll call `optim2`) that manually unroll the loop on line 17. In other words loop L17 must disappear. Compare execution times between the versions `optim1` and `optim2`. What do you notice? How How do you explain it?

No improvement? Did you try on a Cortex-A57 core?

2.4 Inline Function Calls (`optim3`)

You should now have a code with:

- 9 calls to the function `extract_red`,
- 9 calls to the function `extract_green`,
- 9 calls to the function `extract_blue`,
- 9 calls to the function `extract_alpha`,
- 1 call to the function `rgba`.

That's 37 function calls in the body of the deepest loop. These functions are defined in the `include/img_data.h` file. You'll notice that these functions are defined in a header file and preceded by the keyword `inline`. Normally, when one of these functions is called, the compiler is able to replace the call with the function code (*inlining*). However, we're in `-O0` mode, so the compiler doesn't apply this optimization.

Work to do Starting from the `optim2` version written before, write a new version of the code (which you'll call `optim3`) that replaces the function calls listed above with their their code. Compare execution times between versions `optim2` and `optim3` versions. What do you notice? How do you explain it?

No improvement? Did you try on a Cortex-A57 core?

2.5 Variables Rotation (`optim4`)

Some memory accesses are redundant from one pixel to the next. When scanning pixels, we first move along the x-axis. In this case accesses to the right-hand column and the middle column (in the 3x3 matrix representing our stencil).

Work to do Starting from the `optim3` version, write a new version of the code (which you'll call `optim4`) in which you rotate variables on the x-axis. Compare execution times between versions `optim3` and `optim4` versions. What do you notice? How do you explain it?

No improvement? Did you try on a Cortex-A57 core?

2.6 [Bonus] Manage Borders (`optim5`)

Work to do Starting from the `optim4` version written before, write a new version of the code (which you'll call `optim5`) where you'll restore the edge management as in the version of the code (function `blur_do_tile_default`). To do this, don't modify your main loop, just replace the pixel copy of the edges with the the original edge management code. You can copy and paste the code. Compare execution times between versions `optim4` and `optim5` versions. What do you notice? How do you explain it?

2.7 [Bonus] Reduction (`optim6`)

Work to do Starting from the `optim5` version written before, write a new version of the code (which you'll call `optim6`) where you'll apply a column reduction to reduce the number of computation in the kernel. What do you notice? How do you explain it?

No improvement? Did you try on a Cortex-A57 core?

2.8 [Bonus] Compare with Different Levels of Compiler Optimization

Work to do To complete this session, you now need to enable the different compiler optimization levels (e.g. `-O1`, `-O2` and `-O3`). Compare the execution times of the different optimized versions depending on the level of optimization. Is the optimized code faster? If so, by how much?

No improvement? Did you try on a Cortex-A57 core?