CSC5001

Fiche 3 : Cache & SIMD

1 Cache reuse with `scrollup` kernel

We're going to modify the scrollup kernel of EasyPAP in order to provoke some cache faults and observe their impact on performance.

1. Modify the Makefile to compile with level of optimisation -O1. This disables automatic vectorization and will allow us to better distinguish the influence of the program's performance.
2. Duplicate the function `compute_seq()` by naming it `scrollup_compute_ji()` :

```
for (int i = 0; i < DIM; i++) {
  int src = (i < DIM - 1) ? i + 1 : 0;
  for (int j = 0; j < DIM; j++)
    next_img (i, j) = cur_img (src, j);
}
```

   and invert the order of loops that iterate over `i` and `j`. Also, compute the last line of the image out of the loop. Remove the test and put after the loop the following code :

```
for (int j = 0; j < DIM; j++)
  next_img (DIM - 1, j) = cur_img (0, j);
```

3. Use the image `shibuya.png` to check the result. Compare visually the fluidity of the animation obtained by the `seq` and `ji` versions of the `scrollup` kernel.
4. Use the script `plots/run-xp-scrollup.py` to run a series of experiments that will vary the image size and the number of iterations to maintain a constant computation time.

$$\text{nb\_iterations} = \frac{4096 \times 4096}{\text{DIM} \times \text{DIM}}$$

5. Use the folowing command in order to generate curve :

```
./plots/easyplot.py  -y time -x size --yscale log --xscale log \
--delete iterations -if scrollup.csv
```

6. Interprete the figure knowing that a pixel is encoded on 4 bytes and that the cache size of your machine is given by the commande `lstopo`.
7. Implement a tiled version of the initial row-major path. Play with the tile dimension. Compare performance.

   NOTE : DO NOT FORGET to put back the `-O3` level of compilation in the Makefile.

2 Auto-vectorization of `blur` kernel

This exercise deals with the optimisation of sequential code and vectorisation.

The `blur` kernel blurs an image by calculating, for each pixel, the average value of the pixels located in its immediate neighbourhood. However, two different pixels may have a different number of neighbours depending on their position in the image. Here is a code that treats all cases in the same way.

```
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);
      ...
    }
```

1. When a tile is far from the edges, a certain number of tests are unnecessary. Write a variant of the tiling function which treats edge tiles differently from the ones of the center, in order to drastically simplify their calculation.

   Here's what the start of this function might look like :

   ```
   // Optimized implementation of tiles
   int blur_do_tile_opt (int x, int y, int width, int height)
   {
     // Outer tiles are computed the usual way
     if (x == 0 || x == DIM - width || y == 0 || y == DIM - height)
     return blur_do_tile_default (x, y, width, height);

     // Inner tiles involve no border test
     ...
   ```

2. Check your program is working correctly by saving the calculated image after 50 iterations via the `--dump` option (you can work with the image `./data/img/shibuya.png`) and comparing them (command `diff`). Bump images are stored in directeory `data/dump`.

   ```
   ./run -l data/img/shibuya.png -k blur -v tiled -wt default -nt 32 -i 50 --dump

   ./run -l data/img/shibuya.png -k blur -v tiled -wt opt -nt 32 -i 50 --dump

   ```

3. Observe the application's behaviour by using the *monitoring* and pressing the `h` key to activate the *heat mode*. The light intensity of the tile is proportional to its duration (relative to the others). Use vary the grain.

```
./run -l data/img/shibuya.png -k blur -v tiled -wt opt -nt 32 -m
```

4. To compare the relative performance of these two tiling variants, it is possible to generate execution traces and compare them. For thus purpose, you can do for example :

```
# run regular version
./run -l data/img/shibuya.png -k blur -v tiled -wt default -nt 32 -i 10 -t -n \
-lb "Regular 32x32"
# run optimized version
./run -l data/img/shibuya.png -k blur -v tiled -wt opt -nt 32 -i 10 -t -n \
-lb "Optimized 32x32"
# compare last two traces
./view -c
```

5. Experimentally determine a good tile size. Doesn't this give us ideas for further optimising the programme ?
6. Implement the `blur_compute_omp_tiled` version. Load balance as best you can with the help the execution traces. How can you initiate the tile distribution in order to enhance this pathological pattern ?
7. We are now looking to optimise the cache reuse. For example, two images of size 1920×1920 will easily fit in the set of caches L3. Propose a distribution policy that encourages cache reuse.

3 Vectorization of `spin` kernel

The spin kernel, which performs a few simple trigonometric calculations, gives the compiler a hard time, as it is unable to vectorize it automatically. So we're going to manually vectorize the calculations using *intrinsics*.

The interactive Intel Intrinsics Guide will undoubtly become your best friend during this assignment !

3.1 Start of vectorization

The file `spin-codelet.c` contains pieces of code you copy and paste at the end of your `kernel/c/spin.c` file. Do not move `spin-codelet.c` into the the EASYPAP tree.

After recompiling the application, you now have an `avx` tiling variant which performs significantly better than the sequential version. To try it out :

```
./run -k spin -v tiled -wt avx
```

Examine the code to see that the vectorization is partial at this stage (look for the `// FIXME` in particular).

3.2 Arc tangent

Let's start with the `_mm256_atan_ps` function, which poses no any particular problems, since it does not contain any conditional statement.

Replace the sequential iterative code with vector instructions to calculate `AVX_VEC_SIZE_FLOAT` values (i.e. 8) of arc tangents simultaneously. Note that you will probably need to use the function `_mm256_abs_ps` which calculates the absolute of the elements of a vector : take a look at how it works!

Check that your code works in interactive mode, then measure the time saved compared with the old version (option `-i 100 -n`).

3.3 Bonus : Fonction `atan2`

Complete the vectorization of the function `_mm256_atan2_ps`. Notice how the first sequential instructions have been transformed into vector operations...

In particular, note that the test :

```
if (y[i] < 0) th[i] = -th[i];
```

can be read as 'propagate the sign bit from `y` to `th`.

Visually check that the calculations are still correct, and appreciate the performance gains you've made!