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 vectorisation 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 but after the loop, the following code :

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

3. En utilisant l'image `shibuya.png`, vérifier visuellement le résultat. Comparer « visuellement » la fluidité de l'animation obtenues par les versions `seq` et `ji` du noyau `scrollup`.
4. Visually check the result on the image `shibuya.png`. Compare visually the fluidity of the animation obtained by the `seq` and `ji` versions of the `scrollup` kernel.
   Use the script `plots/run-xp-scrollup.py` to run a series of experiments a series of experiments that will vary the image size and the number of iterations to maintain a constant of 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 `./images/shibuya.png`) and comparing them (command `diff`).

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.
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 images/shibuya.png -k blur -v tiled -wt default -nt 32 -i 10 -t -n \
-lb "Regular 32x32"
# run optimized version
./run -l images/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 Vectorisation of `spin` kernel

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

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

3.1 Start of vectorisation

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 vectorisation 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 Fonction `atan2`

Complete the vectorisation 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 !

4 Back to the tasks : search for related components in an image

This exercise is about finding related components in an image in OpenMP. We will use the max kernel. A sequential version is provided in EasyPAP.

4.1 Introduction

The aim is to identify the non-transparent objects present in an image. We are working here with 4-connexity : two pixels are connected if they are adjacent by an edge (north, south, east, west). Two pixels belong to the same object if they are related or if there is a there is a chain of related pixels (non empty) connecting them.

To identify objects, we start by assigning a unique colour to each non-zero pixel. We then propagate the maximum in the neighbourhood of the coloured pixels by iterating the calculation.

At the iteration end (bounded by the number of pixels), the pixels belonging to the same object have all acquired the same colour, that of the pixel with the greatest colour. When propagation has reached a stable, the calculation is complete.

To achieve this propagation, it is natural to iterate a scan of all the pixels by assigning to each non-zero pixel the maximum between the current pixel and its four neighbouring pixels until stagnation. The efficiency of such conventional scan can be significantly improved by replacing it by two scans :

```
1 // first scan in the usual path
2 for (int i = 0; i < DIM; i++)
3   for (int j = 0; j < DIM; j++)
4 ...
5
6 // then a second scan in the reverse path
7 for (int i = DIM-1; i >= 0 ;i--)
```

```
8   for (int j = DIM-1; j >= 0 ;j--)
9 ...
```

It therefore seems pointless to calculate the maximum on all the 4 neighbouring pixels, but simply on those most likely to propagate the maximum.

By convention, let's assume that the downward path corresponds to the path of the table from left to right and then from top to bottom. The max is downwards by consulting the value of the west and north cells. Going up corresponds to scanning from right to left, then from bottom to top. The max is brought up by consulting the south and east cells.

This algorithm is implemented by the function `max_compute_seq` (in `max.c`).

You can test it with the command :

```
./run -l images/spirale.png -k max
```

The propagation can be carried out in parallel without any care because the order of the calculations does not matter as long as a stable state is reached. Thus, parallelize the code using OpenMP loops without respecting the data dependencies induced by sequential algorithm. Here, the aim is to parallelize the functions `tile_down_right` and `tile_up_left` (**copy them** in order to save sequential versions) really simply by adding only two OpenMP directives.

— Check that the code works correctly on the image `spirale.png` :

```
./run -l images/spirale.png -k max -v omp
```

— Measure the performance obtained for `DIM` = 2048 with spirals of 100 turns :

```
./run -s 2048 -k max -v omp -a 100 -n
```

Experimentally, we can see that if we simply straightforward parallelise the for loops, the parallel version performs significantly more iterations than the sequential version when image is large. Indeed, if all the pixels are opaque, then the parallel version will require as many iterations as there are threads to raise the max, instead of just one for the sequential version.

4.2 Parallelization with tasks

One way of improving the efficiency of parallelization is to lightly sacrifice the parallelism to maintain good transmission of the max. In figure 1 below, each square represents a 'tile' of pixels. The contents of each tile are processed sequentially.

In the top-down phase (Figure 1 left) , the top left tile is processed first. Then, as a general rule, a tile can be processed a tile after its north and west neighbours have been processed.

Symmetrically, in the rising phase (Figure 1 right) , the tile at the bottom right is processed first. Then, as a general rule, it can be computed after its southern and eastern neighbours have been processed.

The 'tiled' version of this algorithm is implemented by the function `max_compute_tiled` (in `max.c`), with the number of tiles being set by `NB_TILES_X` × `NB_TILES_Y`. To simplify things, square tiles can be used by setting an identical number horizontally and vertically using the option `--nb-tiles` (or `-nt`). Example using 16 × 16 tiles :

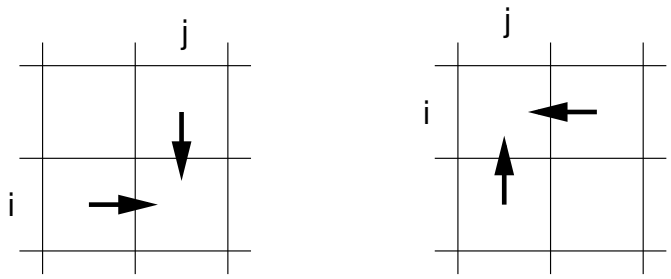

FIGURE 1 – Dependencies between tiles during descent (a) and when rising (b).

```
./run -s 2048 -k max -v tiled -a 100 -nt 16
```

Parallelize the tiled version (by creating a new variant `max_compute_task`) by computing each tile by an OpenMP task. Then use the depend clause to constrain the execution order of the tasks in order to maintain the sequential order of max propagation. To simplify the expression of dependencies, you can use an appendix table which will only be used to express the dependencies :

```
int tile[NB_TILES_Y][NB_TILES_X + 1] __attribute__ ((unused));
```

Once again, check that the code is working correctly on the image `spirale.png` :

```
[my-machine] ./run -k max -v task -l images/spirale.png -nt 16 -n
Using kernel [max], variant [task]
Computation completed after 284 iterations
```

Also check, thanks to an execution trace, that the dependencies between tasks are respected. For this purpose, you can use this sequence of commands :

```
# generate thumbnails
[my-machine] ./run -k max -v task -l images/stars.png -nt 16 -tn -n
# trace
[my-machine] ./run -k max -v task -l images/stars.png -nt 16 -t -n
# observe
[my-machine ./view
```

If you move the mouse over the Gantt chart, from left to right, you should see a 'front' of tasks progressing towards the bottom right-hand corner (see Figure 2), then to the top left corner, etc.

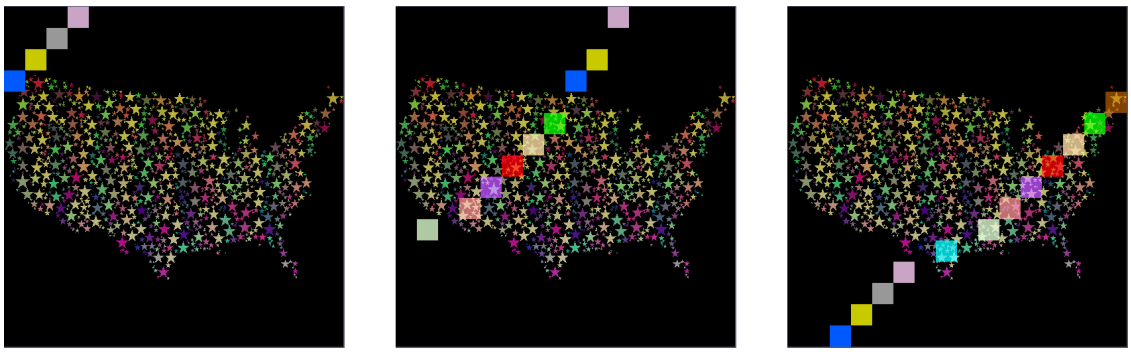

FIGURE 2 – Progression of the task front during the descending phase of maximum propagation.