Sorbonne Université

SESI M2 – ProgPar

Parallel Programming – TP4: Introduction to OpenMP

Short introduction In this session, we'll focus on writing efficient multi-core codes. To achieve this, we're going to use OpenMP directives to create multi-threaded programs. Optimization will be focused on a classic image processing algorithm and we will use the EasyPAP framework.

Very important, about the submission of your work At the end of this session you will have to upload your work (`jetsontx2_specs.txt`, `blur2.c` and `mandel.c` files, and only these files!) on Moodle. After that you will have 2 weeks (until October, 22) 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 files on Moodle. Finally, please write your name plus the name of your pair at the top of the files.

1 Appetizer

First, you need to update your EasyPAP version for this session:

```
git pull origin master
```

1.1 Nvidia Jetson TX2 Peak Performance

In this exercise, you will note your answers in a text file. You will call this file `jetsontx2_specs.txt`.

1.1.1 Cores Peak Performance

Work to do According to the micro-architecture of the ARM Cortex-A57 and Nvidia Denver 2 cores presented in the first session, compute the theoretical CPU peak performance in single and double precision. You will detail your calculations in the text file (reported results have to be in Giga floating-point operations per second or Gflop/s).

1.1.2 Memory Peak Performance

Clone the `bandwidth` micro-benchmark on your `~/Workspace` directory:

```
cd ~/Workspace && git clone https://github.com/alsoc/bandwidth.git
```

To compile `bandwidth`, follow the explanations given in the README.

In the next questions, you will have to copy-paste the `bandwidth` outputs in `jetsontx2_specs.txt` file.

Work to do #1 Run `bandwidth` to estimate the **single-core** bandwidth of the Cortex-A57 cluster. Do the same for **single-core** Nvidia Denver 2 cluster. Then, generate the corresponding plots with the cache sizes (`plot.py` script). In the `jetsontx2_specs.txt` file, you will note the measured throughput of each cache and of the RAM (considering only the `triad` kernel).

Tips #1 `bandwidth` is an OpenMP code. Like most of the OpenMP codes, if you want to run it on a single-core you need to set the `OMP_NUM_THREADS` environment variable to 1 (if not set, by default, OpenMP uses all the available cores).

Tips #2 Remember that you can run a code on a specific core with the `taskset -c <core>` command. $\{0, 3, 4, 5\}$ are the Cortex-A57 cores and $\{1, 2\}$ are the Denver 2 cores. You can even specify a set of cores if you want to run a multi-threaded code.

Tips #3 The cache sizes of the Cortex-A57 and of the Denver 2 clusters are given in the slides of the first session.

Work to do #2 Run `bandwidth` to estimate the **multi-core** bandwidth of the Cortex-A57 cluster (= 4 cores). Do the same for **multi-core** bandwidth of the Nvidia Denver 2 cluster (= 2 cores). Then, generate the corresponding plots with the cache sizes (`plot.py` script). In the `jetsontx2_specs.txt` file, you will note the measured throughput of each cache and of the RAM (considering only the `triad` kernel).

Work to do #3 Run `bandwidth` to estimate the **multi-core** bandwidth of the entire CPU (all the 6 cores). Then, generate the corresponding plot with the cache sizes (`plot.py` script). In the `jetsontx2_specs.txt` file, you will note the measured throughput of the RAM (considering only the `triad` kernel).

2 Main Dish: OpenMP

2.1 `blur2 blur2` always `blur2`

In this exercise, we'll be using EasyPAP and, more specifically, we'll work on the `blur2` kernel. This is the same kernel as in Session 3. Figure 1 illustrates the `blur2` kernel.

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

Figure 1: Illustration of the `blur2` kernel on the `1024.png` image.

Work to do #1 The first objective is to write a tiled variant in witch you will use OpenMP to parallelize the `y-loop`. You will call the new variant: `blur_compute_omp_tiled`. You can copy the `blur_compute_tiled` and start from there.

Tips #1 In the `do_tile` function, replace the last argument (0) by the `omp_get_thread_num()` function call. This way EasyPAP will know which thread computes which part of the frame.
You can choose to execute the tile you want, but it is recommended to use the fastest tile you have ;-).

Here is an example of run you can launch:

```
1  export OMP_NUM_THREADS=4
2  ./run -k blur2 -l images/1024.png -v omp_tiled -wt urrot2_simd_div8_u16 -th 256 -tw 1024 -m
```

In the previous example the code is run on 4 threads (`export OMP_NUM_THREADS=4`) and with the monitoring enable (`-m` option). The variant is the newly `omp_tiled` implemented just before and the tile is the `urrot2_simd_div8_u16` version (the fastest version from the previous session). We defined a tile height of 256 pixels (`-th 256`) and a tile width of 1024 pixels (`-tw 1024`). In other term, we split the image in 4 horizontal tiles. With the monitoring windows you can confirm the horizontal splits and you can see cores usage in real time.

Work to do #2 Let's improve our `blur_compute_omp_tiled` a little bit. First we can say to OpenMP to merge the `y-loop` and the `x-loop` for the indices distribution with the `collapse(2)` directive after the `for` in the OpenMP pragma. Try it!

Now we can also split the frame vertically. Here is an example of run you can launch:

```
1 export OMP_NUM_THREADS=4
2 ./run -k blur2 -l images/1024.png -v omp_tiled -wt urrot2_simd_div8_u16 -th 256 -tw 512 -si -m
```

In the previous command we split the image in 2D blocks of 512×256 pixels.

Work to do #3 Let's improve again our `blur_compute_omp_tiled`. OpenMP scheduler (`static` or `dynamic`) can be set directly in the pragma directive but for our experiments there is an alternative: setting the scheduler at runtime from the command line. To achieve this you can add the following directive at the end of the OpenMP `for` pragma: `schedule(runtime)`.

Now we can specify the scheduler from the command line. Here is an example of run you can launch:

```
1 export OMP_NUM_THREADS=4
2 export OMP_SCHEDULE="static,1"
3 ./run -k blur2 -l images/1024.png -v omp_tiled -wt urrot2_simd_div8_u16 -th 256 -tw 512 -si -m
4 export OMP_SCHEDULE="dynamic,3"
5 ./run -k blur2 -l images/1024.png -v omp_tiled -wt urrot2_simd_div8_u16 -th 256 -tw 512 -si -m
```

Work to do #4 Explore 1) OpenMP schedulers, 2) tiles height/width, 3) the number of threads and 4) the tile version to **find a configuration that gives the lower execution time** (on the `1024.png` image). Report this configuration in the `blur2.c` file with the achieved execution time.

Work to do #5 From the previous configuration you found, plot a speedup graph. You can use, Matplotlib, Libre Office, Gnuplot, Pgfplot...

Work to do #6 For the `1024.png` image, find in which memory you are performing most of the loads and stores. Estimate how many accesses you are performing in the RAM for each pixel computation. From this estimation, give the operational intensity of your code. According to the Roofline model, is it memory bound or compute bound? You will write your answers in the `blur2.c` file.

2.2 Finally, a new one: the `mandel` Kernel

The MANDELBROT set is a famous compute intensive fractal structure (see Figure 2). The CPU sequential implementation is given in the `kernel/c/mandel.c` file. Note that the amount of computations per pixel is irregular in this kernel...

Work to do #1 The first objective is to write a tiled variant in witch you will use OpenMP to parallelize the `y-loop` and `x-loop`. You will call the new variant: `mandel_compute_omp_tiled`. You can copy the `mandel_compute_tiled` and start from there.

Work to do #2 Explore 1) OpenMP schedulers, 2) tiles height/width, 3) the number of threads and 4) the tile version to **find a configuration that gives the lower execution time** (for a size of 1024 pixels, `-s 1024`). Report this configuration in the `mandel.c` file with the achieved execution time.

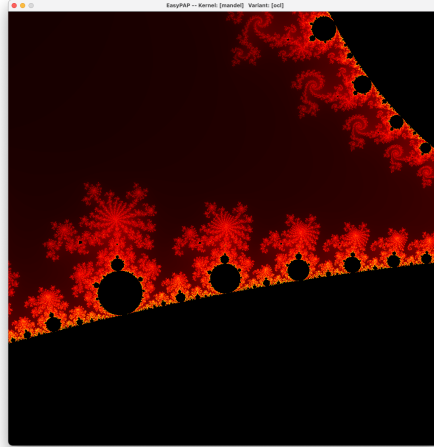

Figure 2: Illustration of the `mandel` kernel after 100 iterations.

Work to do #3 From the previous configuration you found, plot a speedup graph. You can use, Matplotlib, Libre Office, Gnuplot, Pgfplot...

Work to do #4 For the a 1024×1024 image, find in which memory you are performing most of the loads and stores. Estimate how many accesses you are performing in the RAM for each pixel computation. From this estimation, give the operational intensity of your code. According to the Roofline model, is it memory bound or compute bound? You will write your answers in the `mandel.c` file.

2.3 ~~Bonus~~ Challenge: Fast `mandel` Kernel

Work to do Only for the SIMD lovers... Implement a vectorized tile version of the `mandel` kernel with SIMD instructions! This is not too complicated but you will need to perform a reduction on a mask SIMD register to implement the early exit in the inner loop, **this is new**. You can use MIPP or NEON, the choice is yours ;-).