Report | Architecture Lab 1

Jiarui Yao, 2020011031, Yao Class 02

I. Methodology

- First I finished the `c` program which is a straightforward implementation.
- Then according to the `c` program, I finished the first assembly code, which is very slow without doubt.
- Then according to the lab1 handout and other optimization materials, I gradually improved the assembly implementation and finally met the requirement.

II. Optimization methods

- Use shift instructions (e.g., `srai`, `slli`) instead of multiplication/division. Since the kernel is

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} \tag{2}$$

  and every element is some power of 2, we can just use `slli a`(`srai a`) to replace the multiplication(division) by 2^a.

  The code just changes from pattern `muli a1, a2, a3` to `alli a1, a2, i` where `i` is an immediate number.

  This should have improved the performance, but it only played a tiny role, which maybe results from the tiny portion of multiplication instructions.

- Improve the ratio of register reuse. Since the array `img` and `result_img` contains many elements and it takes much longer time to load data from memory than from register. So increasing the ratio of register reuse will help improve the performance of our riscv code.

  The code appears with more registers and I nearly make advantage of all registers.

  This strategy is combined with other strategies.

- Optimize memory accesses through loop optimization techniques.

  1. Loop unrolling and loop blocking to fully utilize the cache.

     Loop tiling partitions a loop's iteration space into smaller chunks or blocks, so as to help ensure data used in a loop stays in the cache until it is reused. And since cache can be reached faster than memory, we can improve the performance of our riscv code.

     This is implemented together with the last two strategies.

- Move the edge cases from the double loop out to a new single loop, which decreases the number of conditional instructions and helps improve the performance.

  Add two extra loops `Loopx_pre` and `Loopy_pre` at the beginning and move the edge case judgements from the main `x,y` loops.

  This helps reduce instructions from 628×10^4 cycles to 621×10^4 cycles ($\approx 1.1\%$)

- Unfold the kernel multiplication loop to 9 times of iteration

  Just break the `i,j` loops up and explicitly implement how the result pixel is obtained.

  This is implemented together with the following strategy.

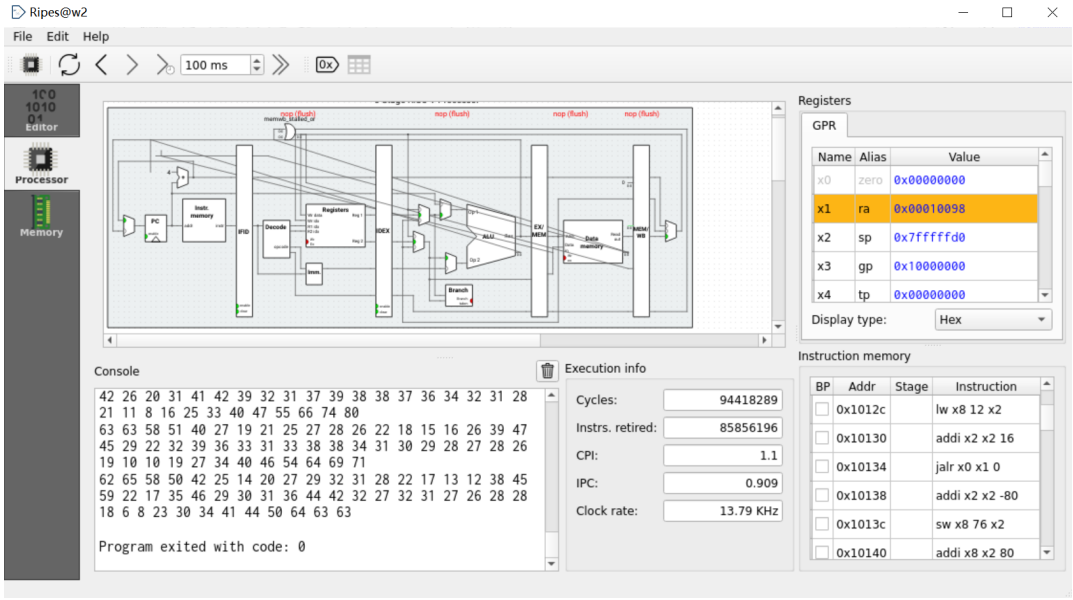
- Load 3×3 pixels at the beginning of each row, and move the kernel window 1 pixel right and load 3 new pixels with the iteration of column index.

  This can decrease the number of `load` instructions per update from 9 to 3 (approximately, since there are edge cases). And adjacent `lw` can also utilize the ability of cache.

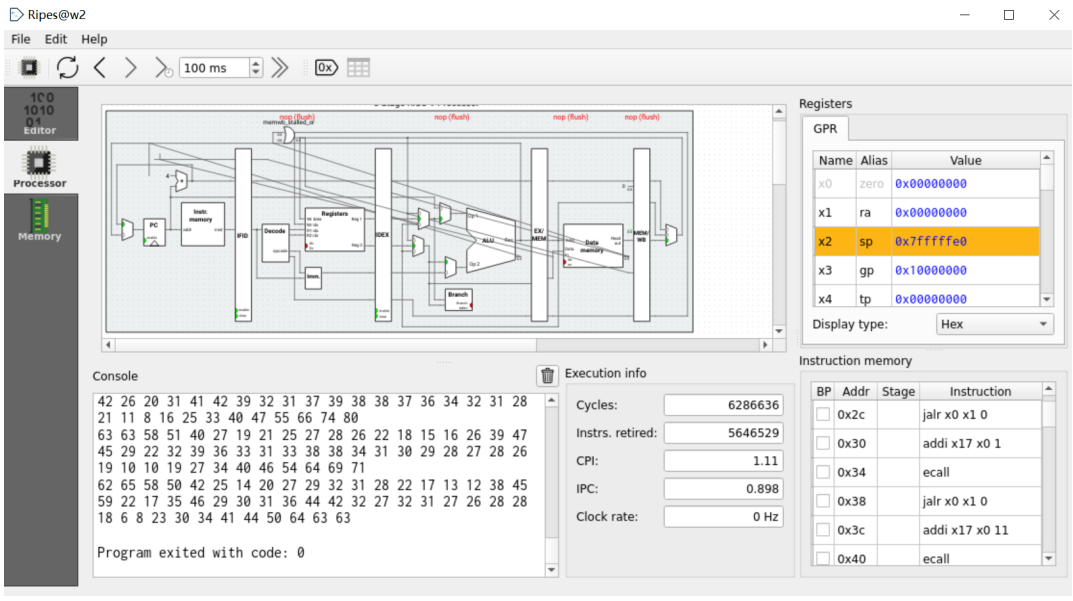
  This helps reduce instructions from 580×10^4 cycles to 245×10^4 cycles ($\approx 55.7\%$).

III. Results

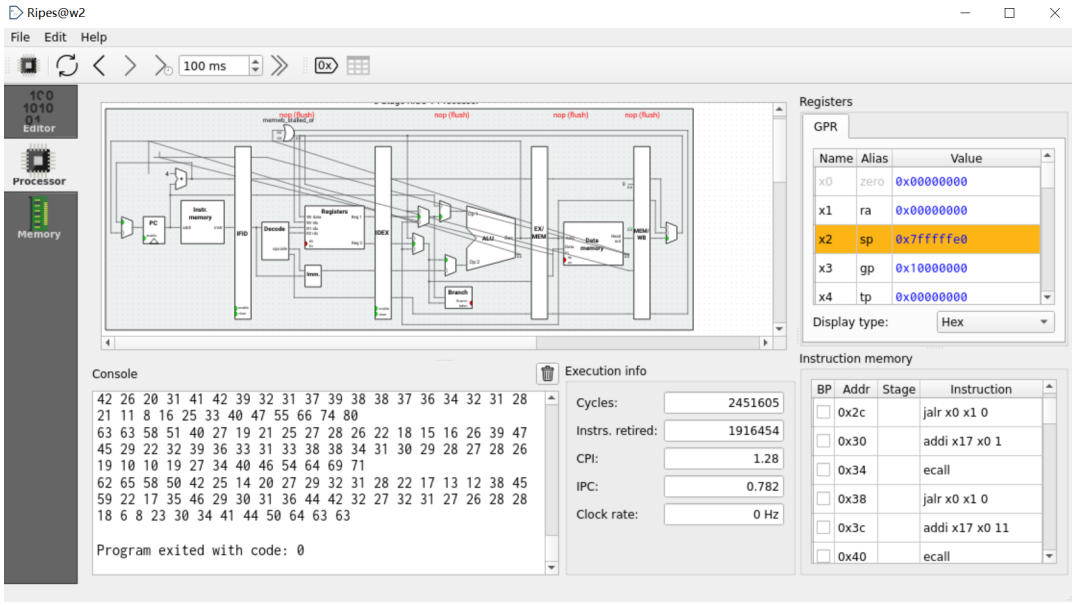
- 9441×10^4 to 628×10^4 to 245×10^4 cycles optimization in total
  - `c` program version



  - Unoptimized assembly version



  - Final optimized assembly version



IV. Question Answering

- Optimization used by GCC O3
    1. Perform loop optimization. Remove constant expression from the loops and simplify the conditions of loops' termination.
    2. Copy memory data to registers before arithmetic calculation, which makes all memory references potentially common expressions, resulting in codes with higher performance. (Store commonly used constants at registers to accelerate the access speed).
    3. Reorder the instructions to eliminate the latency caused by unprepared data. Similarly the complier may reorder the basic blocks.
    4. Remove invariant conditional branches from loops and substitute it with the result.
    5. Make advantage of registers to decrease the times of data loading, such as using blocks to update instead of updating an element every time.
- Limitations of my optimized code
    1. Too many conditional statements are executed, which are slower than ordinary instructions.

       Solution: merge similar condition judgements together and try to move invariant variables out from the loop to accelerate the code

    2. Use redundant registers

       Solution: Try to replace some registers with previous invalid registers

    3. Not making enough advantage of cache

       Solution: Try to load data which is adjacent in physical address simultaneously

    4. Execution of too many `load/store` instructions

       The ideal case is use $n \times m$ `load` instructions, while my current code uses approximately $3n \times m$ `load` instructions. Maybe we can use linear update instead of rectangular matrix update to achieve this goal.

V. Others

Problems and solutions

- Endless loop
  - I once wrote `addi t0, t0, 1` to `addi t0, t0, -1` by mistake, causing the loop to run forever since the iteration variable will always be less than the terminal value
  - Solution: just replace `-1` in the instruction with `1`
- Wrong terminal condition
  - When I first implemented the riscv code, I put the update instruction of iteration variable (such as `i` in the loop `for (int i = 0; i < n; ++i)`) just behind the `Loop` statement, resulting in using a wrong number in the subsequent calculation.
  - Solution: just move the update statement just before the `Cond` statement

Debugging process

- Write a script to generate test data, but the ending symbol at each line contains a redundant blank, so the `image_process` function runs on incorrect data.

  Solution: Rewrite the script and delete the extra blank at the end of each line.

Acknowledgement

- Thank Zihan Hao for discussing with me and motivate me to use different ideas to optimize my assembly code.