Design Document - Lab 1

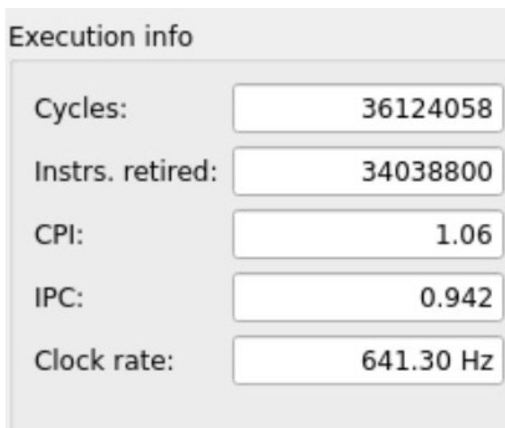
Yaxin Tu, 2019012341

March 27, 2021

1 Methodology and Performance

1.1 C implementation

The C implementation of Gauss Filter is a nested for loop that enumerates every pixel in the result image, judges whether it is on the border and gives it the corresponding gray scale. The cycles it uses to process the picture of Turing is

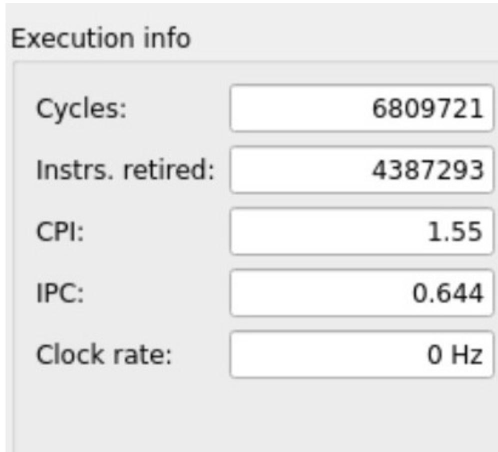


1.2 Unoptimized RISCV implementation

My original RISCV code is basically translating C code to RISCV code. It is mainly partitioned into *Outloop*, *Inloop*, *Border* and *Inner*, corresponding to what in C code: outer-loop, inner-loop, computation given current pixel is a border pixel and computation given current pixel is an inner pixel, respectively.

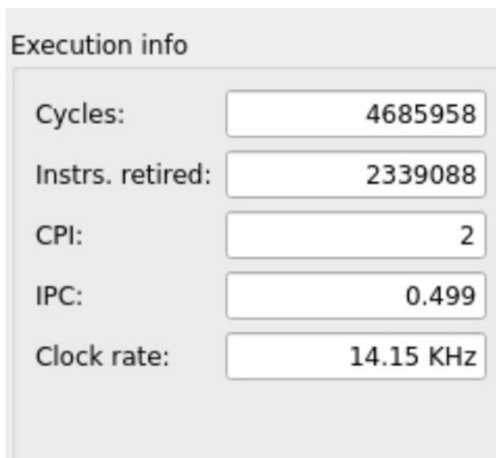
A small optimization in this original RISCV code is that instead of computing $(i+r)\cdot m+(j+t)$ where $r,t\in\{-1,0,1\}$ during each iteration, the RISCV code gives $im+j$ along the loop (it is just the number of iterations that have been processed) then plus some constant value, namely $\pm m\pm 1$ or so.

The performance of the unoptimized RISCV code is



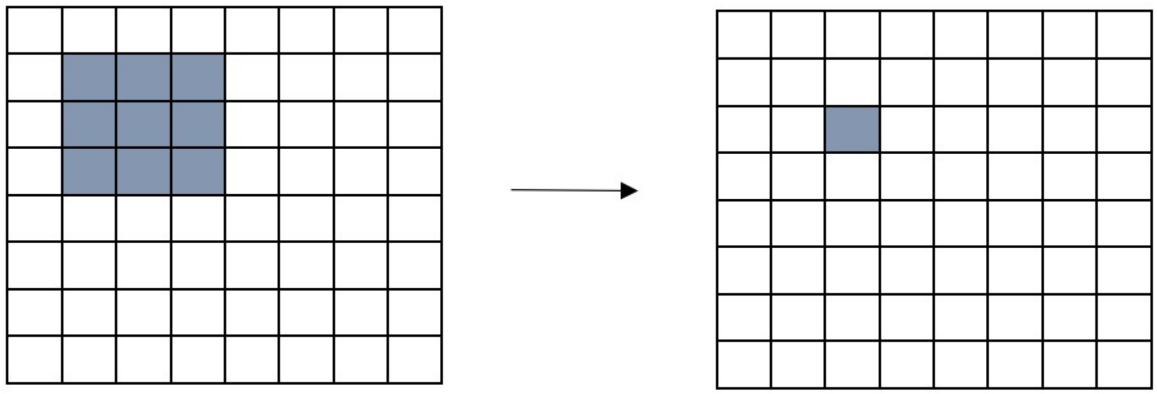
1.3 Optimized RISCV implementation

The optimization on the original RISCV code is loop blocking plus some detail optimizations (reversing the structure of loop condition and loop body, reducing registers and so on). I'll mainly explain how loop blocking is used to optimize the original RISCV code.



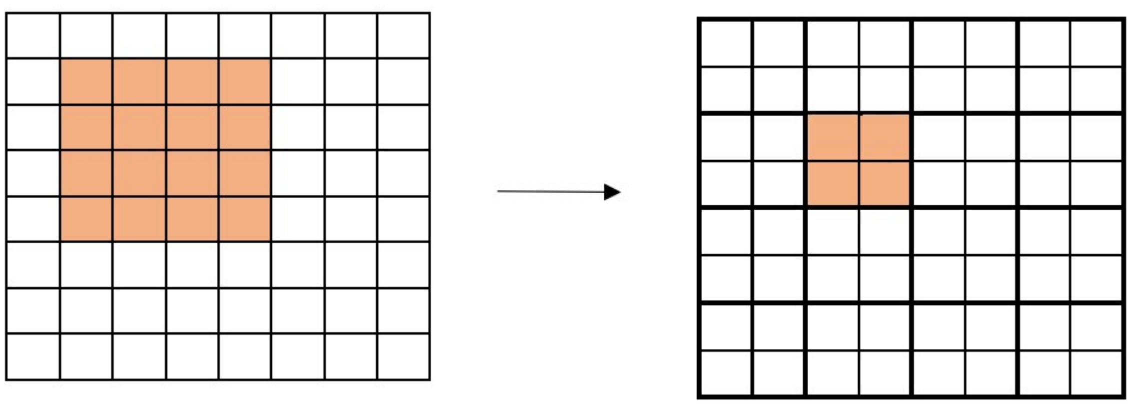
Loop blocking is done by computing 4 result image pixels at a time instead of originally only compute 1 result image pixel. The illustration is as follows:

In the original RISCV code, during each loop we take a 3×3 area of original image and compute the corresponding one result pixel, as shown below.



By this method, nearly every pixel in the result image is going to need 9 load instructions and corresponding arithmetic instructions.

In the optimized RISCV code, during each loop we take a 4×4 area of original image and use them to compute 4 result image pixels, as shown below.



In the optimized process, nearly every pixel of the original image will be loaded and computed only 4 times.

According to the above analysis, the most overt advantage of the optimized version over the original version is that it theoretically reduces the number of load instructions and arithmetic instructions to a fraction of $\frac{4}{9}$. This is also the reality, since the original RISCV code takes 4387293 instructions and the optimized one takes 2339088 instructions, approximately the fraction of $\frac{4}{9}$. However the optimization somehow make CPI increases about $\frac{1}{3}$, possibly due to the complexity of border occasions. (See code, when processing border pixels there are many cases, so a lot of judgement, branch and jump are added, which raises CPI).

In conclusion, when the processed image is large enough the optimized code could reduce the number of instructions to $\frac{4}{9}$ of the original needed. In the case of processing Turing's picture, the optimized code however raises CPI but eventually still reduced the total number of cycles to 67% of the original.

2 Question answering

2.1

My RISCV program is extremely better than my C code on both number of instructions and the total number of cycles. 3 reasons are listed here:

1. First see this segment of assembly implementation of C:

```
10444:      fe842783      lw x15 -24 x8
10448:      00f687b3      add x15 x13 x15
1044c:      000126b7      lui x13 0x12
10450:      87868693      addi x13 x13 -1928
10454:      00279793      slli x15 x15 2
10458:      00f687b3      add x15 x13 x15
1045c:      0007a783      lw x15 0 x15
10460:      00179793      slli x15 x15 1
10464:      00f70733      add x14 x14 x15
10468:      fec42783      lw x15 -20 x8
1046c:      00178793      addi x15 x15 1
10470:      00078693      addi x13 x15 0
10474:      000127b7      lui x15 0x12
10478:      8747a783      lw x15 -1932 x15
1047c:      02f686b3      mul x13 x13 x15
10480:      fe842783      lw x15 -24 x8
10484:      00f687b3      add x15 x13 x15
10488:      00178793      addi x15 x15 1
1048c:      000126b7      lui x13 0x12
10490:      87868693      addi x13 x13 -1928
10494:      00279793      slli x15 x15 2
10498:      00f687b3      add x15 x13 x15
1049c:      0007a783      lw x15 0 x15
104a0:      00f70733      add x14 x14 x15
104a4:      fec42683      lw x13 -20 x8
104a8:      000127b7      lui x15 0x12
104ac:      8747a783      lw x15 -1932 x15
104b0:      02f686b3      mul x13 x13 x15
104b4:      fe842783      lw x15 -24 x8
```

   The read lines mark the same instruction of loading the value of m to $x5$. This operation is done many times in the assembly implementation of C. However in RISCV we just use a register to store the value of m in the first place as follows:

```
lw s0, n
lw s1, m
```

   The RISCV version is of course much more efficient.

2. During each pixel computation we need to locate at a central position $i \times m + j$. In RISCV code this value is computed along with the loop, so no more computations are needed to compute $i \times m + j$. (See the part of RISCV code below.)

```
Outloop:
        addi t0, t0, 1
        addi t1, zero, -1
        blt s0, t0, exit
Inloop:
        addi t1, t1, 1
        blt s1, t1, Outloop
        addi a0, a0, 4 #increment locating address
        addi a1, a1, 4
```

However in the assembly implementation of C, it has to use instructions to compute $i \times m + j$ whenever a require of locating position $i \times m + j$ comes up. (See the segment of assembly implementation of C below.)

```
104bc:      00475713        srli x14 x14 4
104c0:      001066b7        lui x13 0x106
104c4:      ab868693        addi x13 x13 -1352
104c8:      00279793        slli x15 x15 2
104cc:      00f687b3        add x15 x13 x15
104d0:      00e7a023        sw x14 0 x15
```

Here $x14$ is the final result that going to be stored. To store $x14$ to $res\_img[im + j]$, the assembly code computes $im + j$.

3. To complete the computation in each loop, more positions are needed to locate, namely $(i-1)m + j - 1, (i-1)m + j, \cdots, (i+1)m + j + 1$. Together there are 9 positions, and whenever the assembly implementation of C encounters one of these, say $(i-1)m + j$, it simply computes $(i-1)m + j$. See a segment that does this:

```
else{
    result_img[i * m + j] = (img[(i-1)*m + (j-1)]
                          + 2*img[(i-1)*m + j]
                          + img[(i-1)*m + (j+1)]
                          + 2*img[i * m + (j-1)]
                          + 4*img[i * m + j]
                          + 2*img[i*m + (j+1)]
                          + img[(i+1)*m + (j-1)]
                          + 2*img[(i+1) * m + j]
                          + img[(i+1) * m + (j+1)]) >> 4;
```

```
1047c:      02f686b3        mul x13 x13 x15
10480:      fe842783        lw x15 -24 x8
10484:      00f687b3        add x15 x13 x15
10488:      00178793        addi x15 x15 1
1048c:      000126b7        lui x13 0x12
10490:      87868693        addi x13 x13 -1928
10494:      00279793        slli x15 x15 2
10498:      00f687b3        add x15 x13 x15
1049c:      0007a783        lw x15 0 x15
104a0:      00f70733        add x14 x14 x15
```

However RISCV doesn't need to do this. In RISCV what it does is to add fixed values to the central position. For example for $(i-1)m + j$ it computes $(im + j) - m$ where $im + j$ and m are stored in other registers. See a corresponding segment that does this:

```
sub a0, a0, s4 #upper line
lw t2, 0(a0)
slli t2, t2, 1
add t3, t3, t2

lw t2, -4(a0)
add t3, t3, t2

lw t2, 4(a0)
add t3, t3, t2
```

2.2

1. One limitation of my optimized code is analyzed in section 1.3, namely there are many occasion when faced with a border block in the result image. My code generally goes through several branches to each single occasion and deal with it. However this process needs additional branches and also makes the code much longer (since there are so many occasions). So any branch or jump instructions should jump across a large segment of code, which definitely decreases CPI. Ideas to overcome this are: (1) somehow merge several occasions to one to reduce number of branches as well as number of instructions; (2) reorder the border cases and inner pixel case so that each branch or jump instruction won't jump across a very large segment of code.

2. Another limitation of my optimized code is that it doesn't make full use of all registers. The current version is to compute a 2×2 block each time using 4×4 original pixels. However this can be generalized to computing a $k\times k$ block each times using $(k+2)\times(k+2)$ original pixels. In the latter case, computing each result image pixel is only going to need $\frac{(k+2)^2}{k^2}$ load instructions and corresponding arithmetic instructions, strictly smaller than $\frac{16}{4}$ of my optimized code. The bigger k, the more register needed and the more complex the border occasion. There exists a trade-off. However since my optimized code which takes $k=2$ only used a small fraction of registers and didn't do well plan for border occasions, it obviously hasn't reached the balance of this trade-off. In other words, k could be even bigger to increase performance.

3. As discussed above, though my optimization did well on inner occasion, border occasions are very annoying. There must be other methods of loop unrolling that has simpler border occasion. An idea is to combine my blocking method with another one with simpler border occasion to improve performance.

4. My optimization only uses loop unrolling method and some local optimization. Still more optimization method could join in and improve the performance.