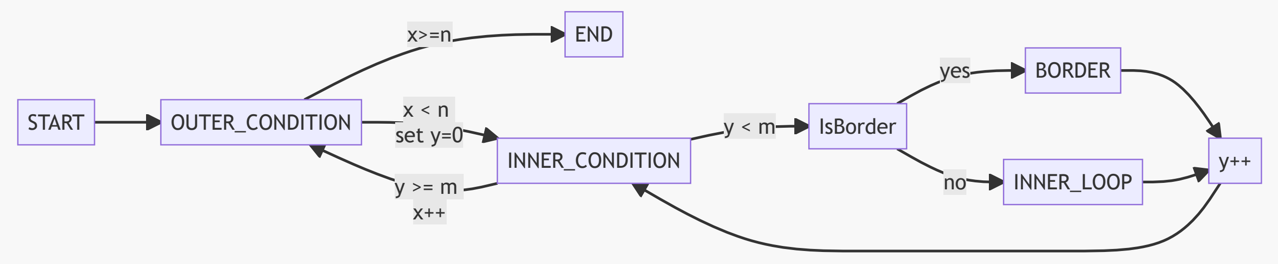
Design Document

Methodology

Initial assembly code organization: `lab1-riscv.S`

I started with `lab1-c.c`. Then I based on this `.c` program to write assembly code. I simply translated it into `lab1-riscv.S`.

I used the general structure for interleaved `for` loops:



I also unrolled the loops inside `lab1-c.c`:

```
/* lab1-c.c */
for(int i=0; i<3; i++){
    for(int j=0; j<3; j++){
        res += img[(x+i-1)*m + (y+j-1)] << K[i*3+j];
    }
}
```

into 9 seperate operations:

```
# lab1-riscv.S
# loop unrolling: compute the nine entries directly without using 9 loops
        addi    a7,zero,0       # a7: res
        # 1
        lw      t5,0(s0)        # t5: K[i][j]
        sub     t3,t4,s2
        addi    t3,t3,-1        # t3: x*m+y-m-1
        slli    t6,t3,2
        add     t6,t6,s3        # t6: &img[x*m+y-1]
        lw      a6,0(t6)        # a6: img[x*m+y-1]
        sll     a6,a6,t5
        add     a7,a7,a6
        # 2
        lw      t5,4(s0)
        lw      a6,4(t6)
        sll     a6,a6,t5
```

```
        add     a7,a7,a6
        # 3
        lw      t5,8(s0)
        lw      a6,8(t6)
        sll     a6,a6,t5
        add     a7,a7,a6
        # 4
        lw      t5,12(s0)
        slli    t3,s2,2         # s3 = 4*m
        add     t6,t6,t3        # t6: x*m+y-1
        lw      a6,0(t6)
        sll     a6,a6,t5
        add     a7,a7,a6
        # 5
        lw      t5,16(s0)
        lw      a6,4(t6)
        sll     a6,a6,t5
        add     a7,a7,a6
        # 6
        lw      t5,20(s0)
        lw      a6,8(t6)
        sll     a6,a6,t5
        add     a7,a7,a6
        # 7
        lw      t5,24(s0)
        add     t6,t6,t3
        lw      a6,0(t6)        # i = 2
        sll     a6,a6,t5
        add     a7,a7,a6
        # 8
        lw      t5,28(s0)
        lw      a6,4(t6)
        sll     a6,a6,t5
        add     a7,a7,a6
        # 9
        lw      t5,32(s0)
        lw      a6,8(t6)
        sll     a6,a6,t5
        add     a7,a7,a6
        # res/16
        srli    a7,a7,4
        slli    t4,t4,2
        add     t3,t4,s4
        sw      a7,0(t3)
```

During this process I also replaced `mul` for `slli` and `div` for `srli`.

Optimization: `lab1-risv-opt.S`

I removed the process of loading and storing the kernel matrix K, since in `lab1-riscv.S` we had done loop unrolling so wo didn't really need a place to store K.

I removed the border cases:

```
/* lab1-c.c */
if(x == 0 || x == n-1 || y == 0 || y == m-1){
    result_img[x*m+y] = (pixel)img[x*m+y];
}
```

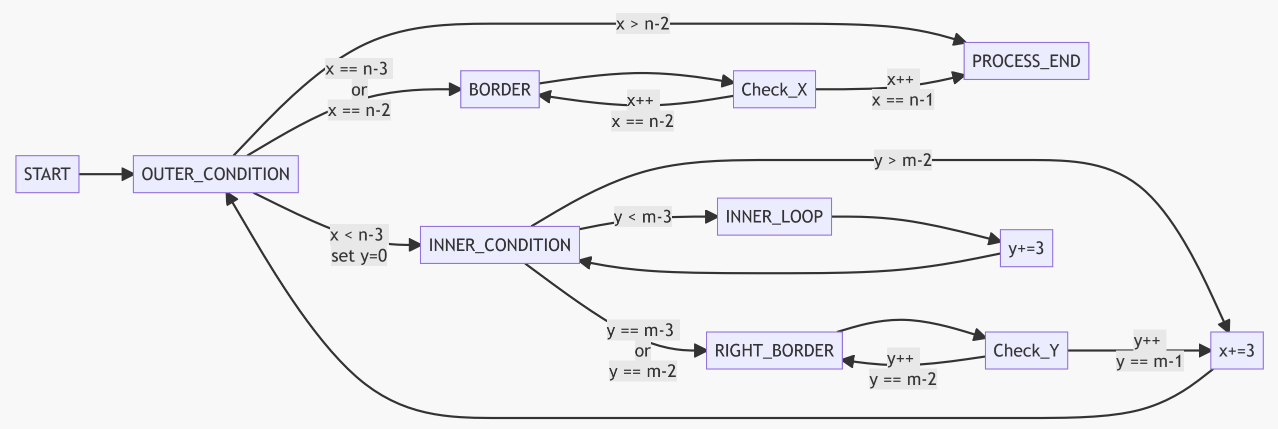
from the `OUTSIDE_LOOP`. Instead, I precomputed them and then computed non-border cases in `OUTSIDE_LOOP`:

```
# lab1-risv-opt.S
# border cases: x==0 || x== n-1 || y==0 || y==m-1
        # (cache - branch trade-off)
        # case1: x == 0 || n-1
        addi    t2,zero,0       # t2: 4*y
        slli    t3,s2,2
        addi    t3,t3,-3        # t3: 4*m-3
        mul     t6,s1,s2
        sub     t6,t6,s2        # t6: n*(m-1)
        slli    t6,t6,2         # t6: 4*n*(m-1)
        j CONDIITION_ONE
LOOP_ONE:
        # x==0
        add     t4,t2,s3        # t4: &img[y]
        add     t5,t2,s4        # t5: &result_img[y]
        lw      a7,0(t4)        # a7: img[y]
        sw      a7,0(t5)        # result_img[y] = img[y]
        # x== n-1
        add     t4,t4,t6        # t4: &img[(n-1)*m+y]
        add     t5,t5,t6        # t5: t5: &result_img[(n-1)*m+y]
        lw      a7,0(t4)        # a7: img[(n-1)*my]
        sw      a7,0(t5)        # result_img[(n-1)*my] = img[(n-1)*my]
        addi    t2,t2,4
CONDIITION_ONE:
        blt     t2,t3,LOOP_ONE

        # case2: y == 0 || m-1
        addi    t1,zero,0       # t1: 4*x*m
        addi    t3,s1,-1
        mul     t3,t3,s2
        slli    t3,t3,2         # t3: 4*(n-1)*m
        slli    t2,s2,2         # t2: 4*m
        addi    t4,s2,-1
        slli    t4,t4,2         # t4: 4*(m-1)
        j       CONDITION_TWO
LOOP_TWO:
        # y==0
        add     t5,t1,s3        # t5: &img[x*m]
        add     t6,t1,s4        # t6: &result_img[x*m]
        lw      a7,0(t5)        # a7: img[x*m]
```

```
        sw      a7,0(t6)        # result_img[x*m] = img[x*m]
        # y==m-1
        add     t5,t5,t4        # t5: &img[x*m+m-1]
        add     t6,t6,t4        # t6: &result_img[x*m+m-1]
        lw      a7,0(t5)
        sw      a7,0(t6)

        add     t1,t1,t2        # 4*x*m + 4*m
CONDITION_TWO:
        bge     t3,t1,LOOP_TWO  # 4*(n-1)*m >= 4*x*m
```

I applied loop tiling (a 3×3 block) to optimize `lab1-riscv.S`. When computing the 3×3 block in `result_img`, we need a 5×5 corressponding block of `img`.

The structure of the process is:



The corresponding code inside each block is:

```
# lab1-riscv-opt.S
# BLOCK TILING: 3 x 3 BLOCK
        addi    t3,zero,0    # t3: store res for result_img[x*m+y]
        addi    t4,zero,0    # t4: store res for result_img[x*m+(y+1)]
        addi    t5,zero,0    # t5: store res for result_img[x*m+(y+2)]
        addi    t6,zero,0    # t6: store res for result_img[(x+1)*m+y]
        addi    s5,zero,0    # s5: store res for result_img[(x+1)*m+(y+1)]
        addi    s6,zero,0    # s6: store res for result_img[(x+1)*m+(y+2)]
        addi    s7,zero,0    # s7: store res for result_img[(x+2)*m+y]
        addi    s8,zero,0    # s8: store res for result_img[(x+2)*m+(y+1)]
        addi    s9,zero,0    # s9: store res for result_img[(x+2)*m+(y+2)]
        # (x-1,y-1): a7 - 4*m - 4 #
        sub     a4,a7,a6
        addi    a4,a4,-4
        lw      a5,0(a4)
        add     t3,t3,a5
        # (x-1,y): a7 - 4*m #
        addi    a4,a4,4
        lw      a5,0(a4)
```

```
        add     t4,t4,a5
        slli    a5,a5,1
        add     t3,t3,a5
        # (x-1,y+1): a7 - 4*m + 4 #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     t3,t3,a5
        add     t5,t5,a5
        slli    a5,a5,1
        add     t4,t4,a5
        # (x-1,y+2): a7 - 4*m + 8 #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     t4,t4,a5
        slli    a5,a5,1
        add     t5,t5,a5
        # (x-1,y+3): a7 - 4*m + 12 #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     t5,t5,a5
        # (x,y-1): a7 - 4 #
        addi    a4,a7,-4
        lw      a5,0(a4)
        add     t6,t6,a5
        slli    a5,a5,1
        add     t3,t3,a5
        # (x,y): a7 #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     s5,s5,a5
        slli    a5,a5,1
        add     t4,t4,a5
        add     t6,t6,a5
        slli    a5,a5,1
        add     t3,t3,a5
        # (x,y+1): a7 + 4 #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     t6,t6,a5
        add     s6,s6,a5
        slli    a5,a5,1
        add     t3,t3,a5
        add     t5,t5,a5
        add     s5,s5,a5
        slli    a5,a5,1
        add     t4,t4,a5
        # (x,y+2): a7 + 8 #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     s5,s5,a5
        slli    a5,a5,1
        add     t4,t4,a5
        add     s6,s6,a5
        slli    a5,a5,1
```

```
        add     t5,t5,a5
        # (x,y+3): a7 + 12 #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     s6,s6,a5
        slli    a5,a5,1
        add     t5,t5,a5
        # (x+1,y-1): a7 + 4*m - 4 #
        add     a4,a7,a6
        addi    a4,a4,-4
        lw      a5,0(a4)
        add     t3,t3,a5
        add     s7,s7,a5
        slli    a5,a5,1
        add     t6,t6,a5
        # (x+1,y): a7 + 4*m #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     t4,t4,a5
        add     s8,s8,a5
        slli    a5,a5,1
        add     t3,t3,a5
        add     s5,s5,a5
        add     s7,s7,a5
        slli    a5,a5,1
        add     t6,t6,a5
        # (x+1,y+1): a7 + 4*m + 4 #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     t3,t3,a5
        add     t5,t5,a5
        add     s7,s7,a5
        add     s9,s9,a5
        slli    a5,a5,1
        add     t4,t4,a5
        add     t6,t6,a5
        add     s6,s6,a5
        add     s8,s8,a5
        slli    a5,a5,1
        add     s5,s5,a5
        # (x+1,y+2): a7 + 4*m + 8 #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     t4,t4,a5
        add     s8,s8,a5
        slli    a5,a5,1
        add     t5,t5,a5
        add     s5,s5,a5
        add     s9,s9,a5
        slli    a5,a5,1
        add     s6,s6,a5
        # (x+1,y+2): a7 + 4*m + 12 #
        addi    a4,a4,4
        lw      a5,0(a4)
```

```
        add     t5,t5,a5
        add     s9,s9,a5
        slli    a5,a5,1
        add     s6,s6,a5
        # (x+2,y-1): a7 + 8*m - 4 #
        add     a4,a7,a6
        add     a4,a4,a6
        addi    a4,a4,-4
        lw      a5,0(a4)
        add     t6,t6,a5
        slli    a5,a5,1
        add     s7,s7,a5
        # (x+2,y): a7 + 8*m #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     s5,s5,a5
        slli    a5,a5,1
        add     t6,t6,a5
        add     s8,s8,a5
        slli    a5,a5,1
        add     s7,s7,a5
        # (x+2,y+1): a7 + 8*m + 4 #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     t6,t6,a5
        add     s6,s6,a5
        slli    a5,a5,1
        add     s5,s5,a5
        add     s7,s7,a5
        add     s9,s9,a5
        slli    a5,a5,1
        add     s8,s8,a5
        # (x+2,y+2): a7 + 8*m + 8 #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     s5,s5,a5
        slli    a5,a5,1
        add     s6,s6,a5
        add     s8,s8,a5
        slli    a5,a5,1
        add     s9,s9,a5
        # (x+2,y+2): a7 + 8*m + 12 #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     s6,s6,a5
        slli    a5,a5,1
        add     s9,s9,a5
        # (x+3,y-1): a7 + 12*m - 4 #
        add     a4,a7,a6
        add     a4,a4,a6
        add     a4,a4,a6
        addi    a4,a4,-4
        lw      a5,0(a4)
        add     s7,s7,a5
```

```
        # (x+3,y): a7 + 12*m #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     s8,s8,a5
        slli    a5,a5,1
        add     s7,s7,a5
        # (x+3,y+1): a7 + 12*m + 4 #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     s7,s7,a5
        add     s9,s9,a5
        slli    a5,a5,1
        add     s8,s8,a5
        # (x+3,y+2): a7 + 12*m + 8 #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     s8,s8,a5
        slli    a5,a5,1
        add     s9,s9,a5
        # (x+3,y+3): a7 + 12*m + 12 #
        addi    a4,a4,4
        lw      a5,0(a4)
        add     s9,s9,a5
        # save t3,t4,t5,t6,s5,s6,s7,s8,s9 to result_image
        # res/16
        srli    t3,t3,4
        srli    t4,t4,4
        srli    t5,t5,4
        srli    t6,t6,4
        srli    s5,s5,4
        srli    s6,s6,4
        srli    s7,s7,4
        srli    s8,s8,4
        srli    s9,s9,4
        # a7 = 4*(x*m+y) + result_img
        sub     a7,a7,s3
        add     a7,a7,s4
        # save t3
        sw      t3,0(a7)
        # save t4
        addi    a7,a7,4
        sw      t4,0(a7)
        # save t5
        addi    a7,a7,4
        sw      t5,0(a7)
        # save s6
        add     a7,a7,a6
        sw      s6,0(a7)
        # save s5
        addi    a7,a7,-4
        sw      s5,0(a7)
        # save t6
        addi    a7,a7,-4
        sw      t6,0(a7)
```

```
        # save s7
        add     a7,a7,a6
        sw      s7,0(a7)
        # save s8
        addi    a7,a7,4
        sw      s8,0(a7)
        # save s9
        addi    a7,a7,4
        sw      s9,0(a7)
        # update y
        addi    t2,t2,3
        # update a7
        sub     a7,a7,s4
        add     a7,a7,s3
        addi    a7,a7,4
        sub     a7,a7,a6
        sub     a7,a7,a6
```

Inside a block, I reduced the number of `lw` by realizing the fact that we can load each `img` entry once in each block (i.e., each inner loop) and reused them. For example, the center of each block `img[(x+1)*m+(y+1)]` would be reused for nine times, thus we needed to load it for only one time in the whole `image_process` process. However, in the naive `lab1-riscv.S` implementation, we need to load it for nine times (in nine inner loops). Hence this trick greatly improved the efficiency by cutting down memory access overheads. This helped speed up the initial implementation about 4 times faster, though it increased CPI and reduced IPC .

I also tried loop unrolling: unrolled the inner `y`'s loop. However, this only cut 1 cycles from the previous implementation. So I didn't implement it after all.

Results

cycles of 3 programs:

- `lab1-c.c`: 70909765
- `lab1-riscv.S`: 8796200
- `lab1-riscv-opt.S`: 2315602

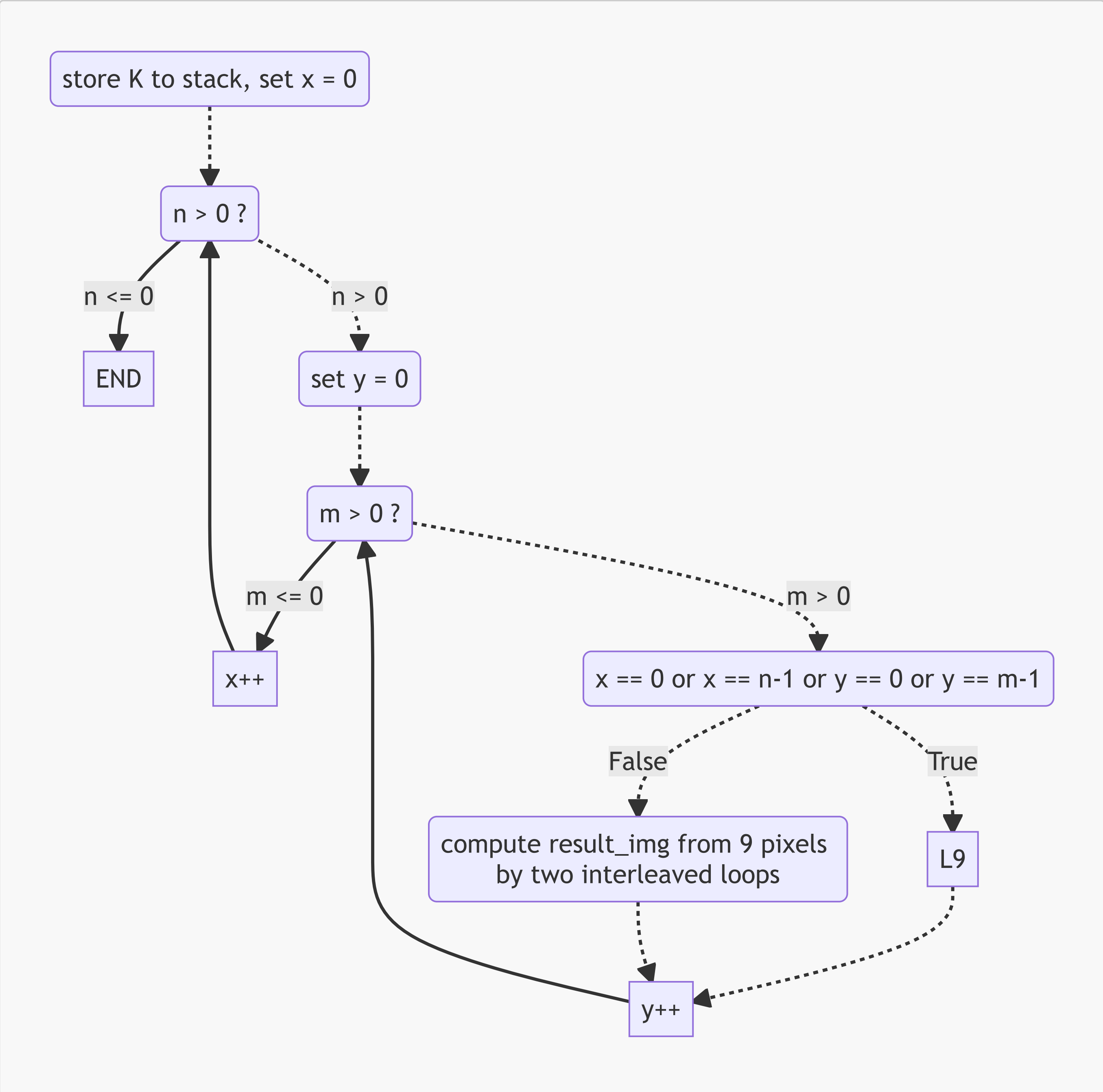
Question Answering

The source code `lab1-c.c` is :

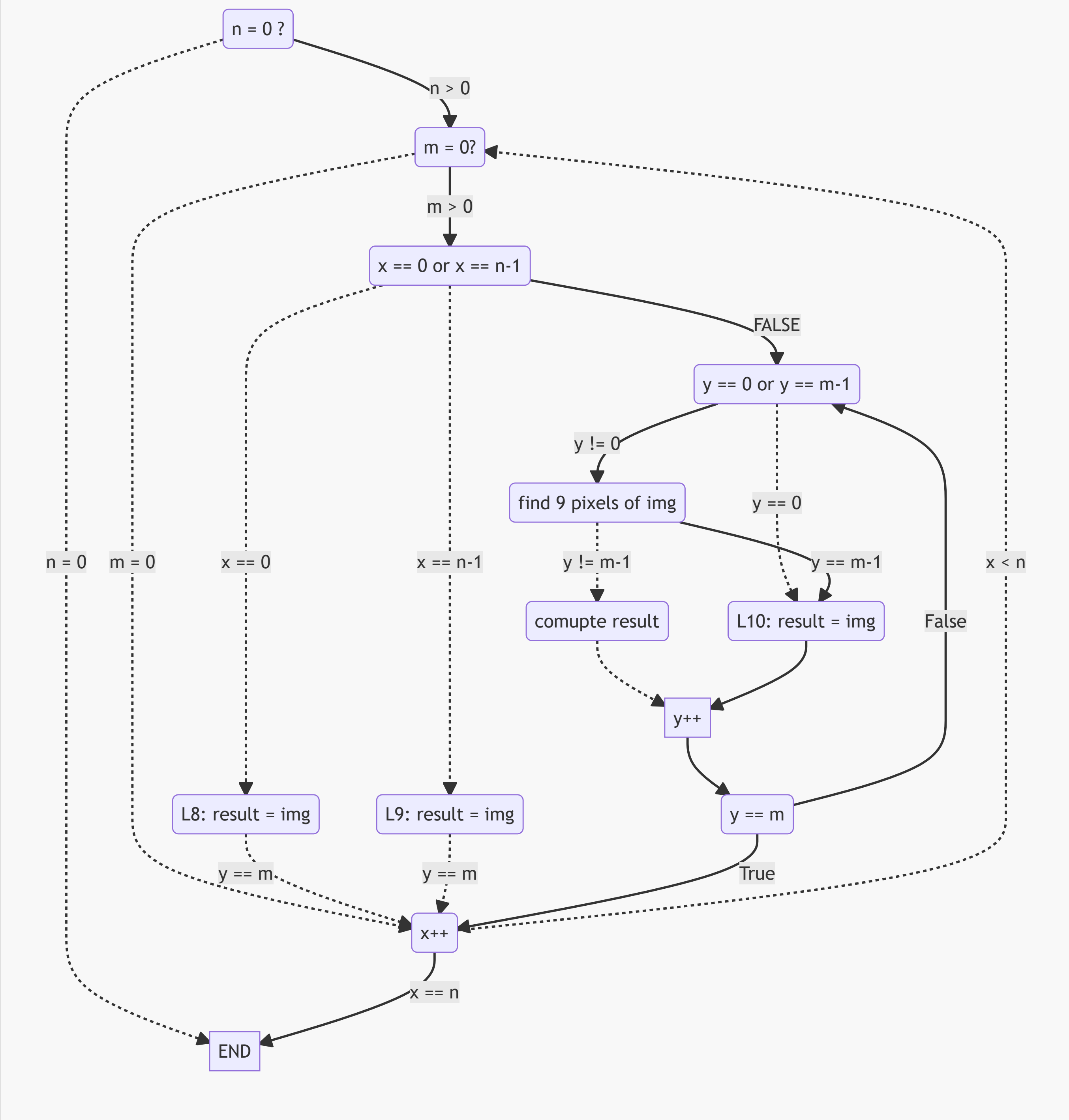
```c
void image_process() {
    for(int x=0; x<n; x++){
        for(int y=0; y<m; y++){
            if(x == 0 || x == n-1 || y == 0 || y == m-1){
                result_img[x*m+y] = (pixel)img[x*m+y];
            }
            else{
                int res = 0;
                for(int i=0; i<3; i++){
                    for(int j=0; j<3; j++){
```

```
                        res += img[(x+i-1)*m + (y+j-1)] << K[i*3+j];
                    }
                }
                result_img[x*m+y] = (pixel)(res>>4);
            }
        }
    }
    return;
}
```

The structure of `lab1-c-o1.S` of optimization level `-O1` is:



The structure of `lab1-c-o3.S` of optimization level `-O3` is:



optimization technique 1: use registers more. In `lab1-c-o1.S`, temporary variables in `.c` code `x`,`y`,`i`,`j`,`res` were all stored in its stack frame, and every time they were needed/updated, the assembly code loaded/stored them from/to the stack. However, in `lab1-c-o2.S`, temporary variables were stored in registers. For example, `x` was stored in `t1` register and `y` was stored in `a2` register in `L12` code block. This reduced the load/store overheads.

optimization techniqe 2: reduce branches. In `lab1-c-o1.S`, condition

```
x == 0 || x == n-1 || y == 0 || y == m-1
```

was checked each time when either `x` or `y` were updated. Each check took a branch and thus the assembly code took many branches for border check. However, in `lab1-c-o2.S`, `x` was checked only when `x` was updated. When only `y` was updated and `x` was kept unchanged, the code didn't check `x`'s border condition. This reduced number of branches a lot for large `m`.

optimization technique 3: not store the kernel matrix K. In `lab1-c-o1.S`, K was stored at the beginning and was fetched from the stack everytime $K[i][j]$ was needed. However, in `lab1-c-o2.S`, K didn't really existed as the code computed it directly.

optimization technique 4: loop unrolling. There are two loop unrollings used. In `lab1-c-o2.S`, the core of the process

```
res += img[(x+i-1)*m + (y+j-1)] << K[i*3+j];
```

was directly computed without using a 3×3 loop (as in `lab1-c-o1.S`). This not only saved `lw`/`sw` overheads and branches, but also got the code rid of storing K (optimization technique 4). Another loop unrolling was used in the border case:

```
result_img[x*m+y] = (pixel)(res>>4);
```

In `lab1-c-o2.S` code blocks `L8` and `L9`, `y` is updated by step size of 2, and in each loop, the code did the calculation for `y` and `y+1`. This helped fufill the pipeline.

optimization technique 5: reuse values. In `lab1-c-o1.S`, when computing `img[x*m+y]` and `result_img[x*m+y]`, the value `x*m+y` was computed twice for `img` and `result_img` seperately. However, in `lab1-c-o2.S`, this value was reused in code block `L12` when finding the 9 surrounding blocks of `img[x*m+y]`.

Limits in my optimized code: I got stuck on its current clock cycle number. I tried loop unrolling but the effect was negligible. I also thought of using 4×4 block for loop tiling, but there seemed lack of registers. Maybe I could use better code structure to reduce branches. I could also utilize cache to speed it up.