Christos Axelos, AEM 1814, 1st Assignment in High Performance computing 2017/18

OBSERVATIONS

- All measurements had the value of PSNR = inf

- At the end of the document there is a **graph** with all the measurements

- During the time measurements, the CPU **wasn't busy** with other "heavy" calculations, in order to measure the time more **precisely**

- The minimum **time resolution** is 1us and is implemented using the function **clock_gettime()** which is used before the start and after the end. Before measuring the end time, we have already compared the differences between original output and our output, in order to calculate the **PSNR**(Signal to Noise Ratio)

- The spreadsheet includes both **-fast** and **-O0** measurements. However this report includes only **the best** of these 2

- All the measurements have an **unique number** in parenthesis. This number corresponds to the number of measurement in spreadsheet

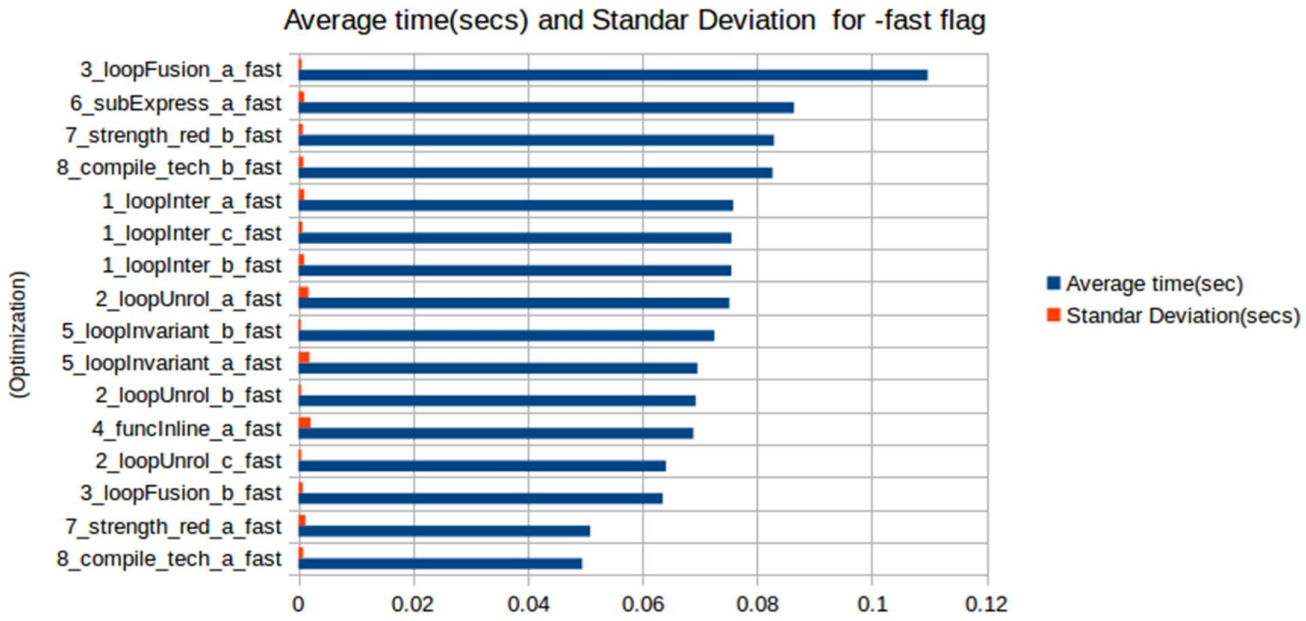
- In (c) Code of Unroll optimization, I assumed that the defined size is **4096**. If "#define SIZE 4096" changes, the loop must be rewritten

- In spreadsheet, with **cyan** color are the best variation per optimization technique for -O0 and with **green** are for -fast

- The optimization methods have been applied by the **order** in the pdfs:

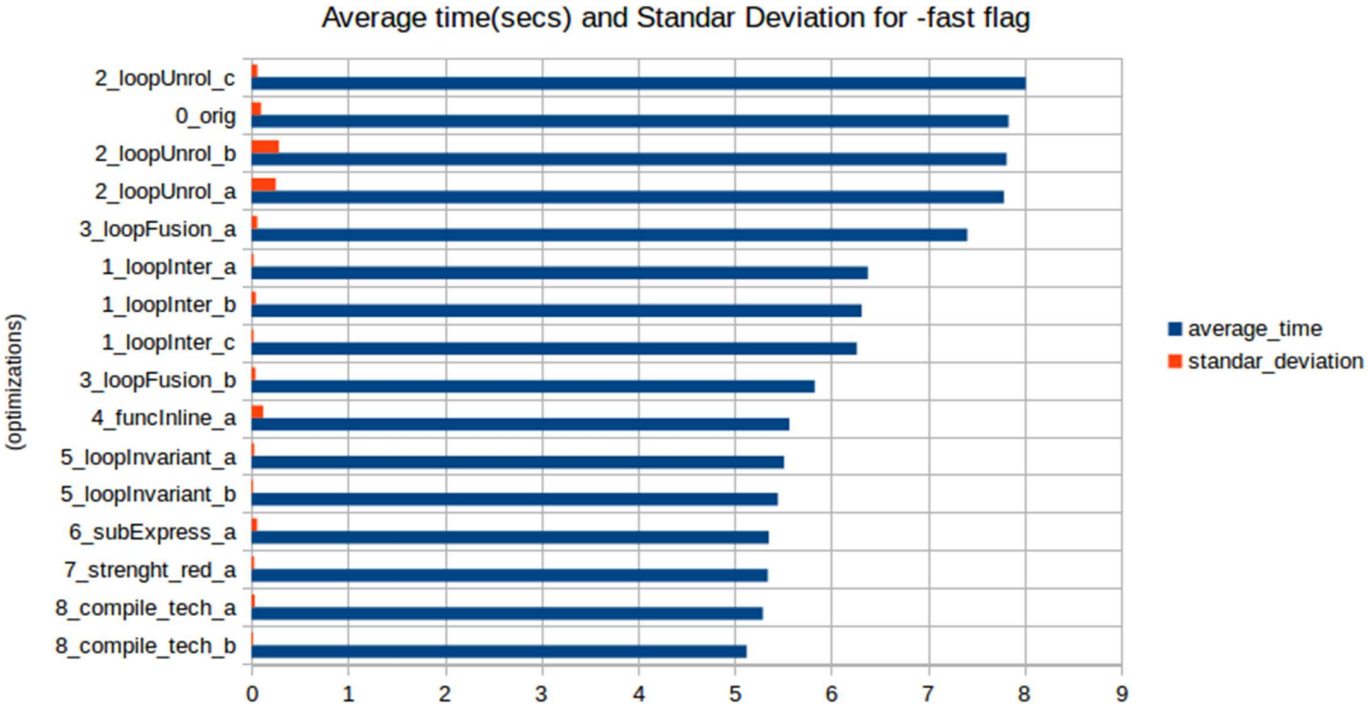
RESULTS

- After executing all the following optimizations for the **-fast** flag, we achieve the following performances



- The y axis corresponds to the the Optimization performed and x axis to **time**(in seconds)

- The same also for the **-O0** flag



- Now that we are done with the schematics, lets analyse each optimization

LOOP INTERCHANGE

- Optimizations began from the source code **0_orig.c**

- We can apply that optimization, because we dont **read** from memory after the value has been modified by a **previous** loop iteration: e.g for (i=0; i<N; i++)

a[i] = a[i-1] + b[i];

- In line 109, we change the loop order, so the variable "i" runs in the outer loop and variable "j" in the inner. That leads in **much better** performance, 0.075837375secs with the -fast flag enabled (Code **(a)**)

```
for (i=1; i<SIZE-1; i+=1) {
        for (j=1; j<SIZE-1; j+=1 ) {
                /* Apply the sobel filter and calculate the magnitude *
                 * of the derivative.
                p = pow(convolution2D(i, j, input, horiz_operator), 2) +
                        pow(convolution2D(i, j, input, vert_operator), 2);
                res = (int)sqrt(p);
                /* If the resulting value is greater than 255, clip it *
                 * to 255.
                if (res > 255)
                        output[i*SIZE + j] = 255;
                else
                        output[i*SIZE + j] = (unsigned char)res;
        }
}
```

- Including the previous optimization, we repeat the some methodology for the **convolution2D** function. That leads again to a better impletentation,317 microseconds **faster** than the (a) ,using the -fast flag(Code **(b)**)

```
for (i = -1; i <= 1; i++) {
        for (j = -1; j <= 1; j++) {
                res += input[(posy + i)*SIZE + posx + j] * operator[i+1][j+1];
        }
}
return(res);
```

- However, if we try to do the same thing in the nested loop in **line 127** the performance is getting **worser** per 12 microseconds than the previous one.
In the spreadsheet, number 7 uses the **-O0** and number 8 the **-fast**(Code **(c)**)

```
for (i=1; i<SIZE-1; i++) {
        for ( j=1; j<SIZE-1; j++ ) {
                t = pow((output[i*SIZE+j] - golden[i*SIZE+j]),2);
                PSNR += t;
        }
}
```

- In the spreadsheet, **code(a)** corresponds to numbers number 3 and 5, **code(b)** to 4 and 6 and finally **code(c)** to 7 and 8

- So we keep the **code(b)** implementation and continue to the next optimization method.

LOOP UNROLLING

- Optimizations began from the source code **1_loopInter.c**

- We start this optimization from the function **convolution2D.** This function contains the following nested loop:

```
for (i = -1; i <= 1; i++) {
        for (j = -1; j <= 1; j++) {
                res += input[(posy + i)*SIZE + posx + j] * operator[i+1][j+1];
        }
}
```

- Firsty we try to unroll the inner loop. So the new code looks like the above(Code **(a)**)

```
for (i = -1; i <= 1; i++) {
        res += input[(posy -1)*SIZE + posx -1] * operator[0][0];
        res += input[(posy -1)*SIZE + posx ] * operator[0][1];
        res += input[(posy -1)*SIZE + posx + 1] * operator[0][2];

}
```

- Now we measure with both **-O0** and **-fast** and we find 2 complete different results. Flag **-O0** seems not to decrease the execution time. On the contrary it increases it.
- On the other hand, **-fast** flags seems to create the best running time until now. So lets try to unroll the outer loop and make the nested loop fully unrolled (Code **(b)**)

```
res += input[(posy -1)*SIZE + posx -1] * operator[0][0];
res += input[(posy -1)*SIZE + posx ] * operator[0][1];
res += input[(posy -1)*SIZE + posx + 1] * operator[0][2];

res += input[(posy )*SIZE + posx -1] * operator[1][0];
res += input[(posy )*SIZE + posx ] * operator[1][1];
res += input[(posy )*SIZE + posx + 1] * operator[1][2];

res += input[(posy + 1)*SIZE + posx -1] * operator[2][0];
res += input[(posy + 1)*SIZE + posx ] * operator[2][1];
res += input[(posy + 1)*SIZE + posx + 1] * operator[2][2];
```

- While the automatic optimizations that compiler produces using the **-fast** decreases the latency, the **-O0** flag seem to increase the execution's time. Under that circumstances, compiler's optimization like register allocation seem to do great job

- Now, we continue with further unrolling. Now look the nested loop in **sobel** function

```
/* For each pixel of the output image */
for (i=1; i<SIZE-1; i+=1) {
        for (j=1; j<SIZE-1; j+=1 ) {
                /* Apply the sobel filter and calculate the magnitude *
                 * of the derivative.
                p = pow(convolution2D(i, j, input, horiz_operator), 2) +
                        pow(convolution2D(i, j, input, vert_operator), 2);
                res = (int)sqrt(p);
                /* If the resulting value is greater than 255, clip it *
                 * to 255.
                if (res > 255)
                        output[i*SIZE + j] = 255;
                else
                        output[i*SIZE + j] = (unsigned char)res;
        }
}
```

- Try to unroll the inner loop. Assuming that SIZE=4096, the inner loop executes **4094 times**. So we can unroll the loop N times, where 4094%N = 0. Possible values for N can be 2, or 8

- For N = 2, the execution time seems to be the same as the previous best time for both **-fast** and -**O0**. So no effect.

- If we try N =4, despite the fact that 4094%4 = 2, we get better time with PSNR=inf. The reason why choosing N = 4 isn't **illegal** is because we don't really count the last column and row of the array. So we keep N = 4. The following picture shows how the last executions for unroll factor = 4:

	J=0	…	4093	4094	4095
I=4090			1st	2nd	3rd
I=4091	4th		1st	2nd	3rd
I=4092	4th		1st	2nd	3rd
I=4093	4th		1st	2nd	3rd
I=4094	4th		1st	2nd	3rd
I=4095	4th				

- If we try to unroll our loop **4 times** instead of 2, the bellow code shows the unroll(Code **(c)**)

- We get an obvious decrease in total time using **-fast,** however for **-O0** flag we again don't take worthy results. Using the **-fast,** the total time has fallen to 0.0641150333 seconds.

```
for (i=1; i<SIZE-1; i+=1) {
        for (j=1; j<SIZE-1; j+=2) {
                //LOOP 1
                p = pow(convolution2D(i, j, input, horiz_operator), 2) +
                        pow(convolution2D(i, j, input, vert_operator), 2);
                res = (int)sqrt(p);
                if (res > 255)
                        output[i*SIZE + j] = 255;
                else
                        output[i*SIZE + j] = (unsigned char)res;

                //LOOP 2
                p = pow(convolution2D(i, j+1, input, horiz_operator), 2) +
                        pow(convolution2D(i, j+1, input, vert_operator), 2);
                res = (int)sqrt(p);
                if (res > 255)
                        output[i*SIZE + j+1] = 255;
                else
                        output[i*SIZE + j+1] = (unsigned char)res;
        }
}
```

- For **N = 8**, we get the same results as the previous one. So we stop at **N=4**

- Now, if we try to unroll the outer loop the code that we will be produced is going to increase **a lot** the size of the binary file. Also, after measurements for N = 2 or 4, the running time stays **stable** for -fast and gets even **worser** for -O0

- Now lets continue with the following loop:

```
for (j=1; j<SIZE-1; j++) {
        for ( i=1; i<SIZE-1; i++ ) {
                t = pow((output[i*SIZE+j] - golden[i*SIZE+j]),2);
                PSNR += t;
        }
}
```

```
                pow(convolution2D(i, j, input, vert_operator), 2);
        res = (int)sqrt(p);
        if (res > 255)
                output[i*SIZE + j] = 255;
        else
                output[i*SIZE + j] = (unsigned char)res;

        //LOOP 2
        p = pow(convolution2D(i, j+1, input, horiz_operator), 2) +
                pow(convolution2D(i, j+1, input, vert_operator), 2);
        res = (int)sqrt(p);
        if (res > 255)
                output[i*SIZE + j+1] = 255;
        else
                output[i*SIZE + j+1] = (unsigned char)res;

        //LOOP 3
        p = pow(convolution2D(i, j+2, input, horiz_operator), 2) +
                pow(convolution2D(i, j+2, input, vert_operator), 2);
        res = (int)sqrt(p);
        if (res > 255)
                output[i*SIZE + j+2] = 255;
        else
                output[i*SIZE + j+2] = (unsigned char)res;

        //LOOP 4
        p = pow(convolution2D(i, j+3, input, horiz_operator), 2) +
                pow(convolution2D(i, j+3, input, vert_operator), 2);
        res = (int)sqrt(p);
        if (res > 255)
                output[i*SIZE + j+3] = 255;
        else
                output[i*SIZE + j+3] = (unsigned char)res;
        }
}
```

- After measuring times, we don't have any improvement. I tried unrolling these 2 loops and separately and together for N = 2, 4 and 8 but there was no improvement. Also tried to change loop order in order to increase **locality** for the inner loop, but it had no effect.

- In the spreadsheet, **code(a)** corresponds to numbers number 9 and 10, **code(b)** to 11 and 12 and finally **code(c)** to 13 and 14

- So we keep the **code(c)** implementation and continue to the next optimization method.

LOOP FUSION

- Optimizations began from the source code 2**_loopUnrol.c**

- In function **sobel ,** we have the 2 following for nested for-loops:

(a)

```
for (i=1; i<SIZE-1; i+=1) {
        for (j=1; j<SIZE-1; j+=4) {
                //LOOP 1
                p = pow(convolution2D(i, j, input, horiz_operator), 2) +
                        pow(convolution2D(i, j, input, vert_operator), 2);
                res = (int)sqrt(p);
                if (res > 255)
                        output[i*SIZE + j] = 255;
                else
                        output[i*SIZE + j] = (unsigned char)res;

                //LOOP 2
                p = pow(convolution2D(i, j+1, input, horiz_operator), 2) +
                        pow(convolution2D(i, j+1, input, vert_operator), 2);
                res = (int)sqrt(p);
                if (res > 255)
                        output[i*SIZE + j+1] = 255;
                else
                        output[i*SIZE + j+1] = (unsigned char)res;

                //LOOP 3
                p = pow(convolution2D(i, j+2, input, horiz_operator), 2) +
                        pow(convolution2D(i, j+2, input, vert_operator), 2);
                res = (int)sqrt(p);
                if (res > 255)
                        output[i*SIZE + j+2] = 255;
                else
                        output[i*SIZE + j+2] = (unsigned char)res;

                //LOOP 4
                p = pow(convolution2D(i, j+3, input, horiz_operator), 2) +
                        pow(convolution2D(i, j+3, input, vert_operator), 2);
                res = (int)sqrt(p);
                if (res > 255)
                        output[i*SIZE + j+3] = 255;
                else
                        output[i*SIZE + j+3] = (unsigned char)res;
        }
}
```

(b)

```
for (j=1; j<SIZE-1; j++) {
        for ( i=1; i<SIZE-1; i++ ) {
                t = pow((output[i*SIZE+j] - golden[i*SIZE+j]),2);
                PSNR += t;

        }
}
```

- If we notice these 2 nested loops, we can implement the **Loop-Fusion** technique with some changes in the 2nd loop. Firstly change loop order and then unroll partialy by j = j + 4. Lets take it step by step:

a) By changing loop order at **(b)** we have the same time as before, so:

```
for (i=1; i<SIZE-1; i++) {
        for ( j=1; j<SIZE-1; j++ ) {
                t = pow((output[i*SIZE+j] - golden[i*SIZE+j]),2);
                PSNR += t;
        }
}
```

b) Now we merge the two following loops (**code(a)**)

```
for (i=1; i<SIZE-1; i+=1) {
        for (j=1; j<SIZE-1; j+=4) {
                //LOOP 1a
                p = pow(convolution2D(i, j, input, horiz_operator), 2) +
                        pow(convolution2D(i, j, input, vert_operator), 2);
                res = (int)sqrt(p);
                if (res > 255)
                        output[i*SIZE + j] = 255;
                else
                        output[i*SIZE + j] = (unsigned char)res;

                //LOOP 1b
                t = pow((output[i*SIZE+j] - golden[i*SIZE+j]),2);
                PSNR += t;



                //LOOP 2a
                p = pow(convolution2D(i, j+1, input, horiz_operator), 2) +
                        pow(convolution2D(i, j+1, input, vert_operator), 2);
                res = (int)sqrt(p);
                if (res > 255)
                        output[i*SIZE + j+1] = 255;
                else
                        output[i*SIZE + j+1] = (unsigned char)res;


                //LOOP 2b
                t = pow((output[i*SIZE+j+1] - golden[i*SIZE+j+1]),2);
                PSNR += t;



                //LOOP 3a
                p = pow(convolution2D(i, j+2, input, horiz_operator), 2) +
                        pow(convolution2D(i, j+2, input, vert_operator), 2);
                res = (int)sqrt(p);
                if (res > 255)
```

- If we measure again the time is almost the **double** than before, 0.111 seconds and **PSNR** is 37.4392. We don't continue more that optimization because we don't improve our results

- Now lets go back at out **2_LoopUnrol.c** code. If we try to merge the outer for-loops, the code looks like the following(**code(b)**)

```
        if (res > 255)
                output[i*SIZE + j] = 255;
        else
                output[i*SIZE + j] = (unsigned char)res;

        //LOOP 2
        p = pow(convolution2D(i, j+1, input, horiz_operator), 2) +
                pow(convolution2D(i, j+1, input, vert_operator), 2);
        res = (int)sqrt(p);
        if (res > 255)
                output[i*SIZE + j+1] = 255;
        else
                output[i*SIZE + j+1] = (unsigned char)res;

        //LOOP 3
        p = pow(convolution2D(i, j+2, input, horiz_operator), 2) +
                pow(convolution2D(i, j+2, input, vert_operator), 2);
        res = (int)sqrt(p);
        if (res > 255)
                output[i*SIZE + j+2] = 255;
        else
                output[i*SIZE + j+2] = (unsigned char)res;

        //LOOP 4
        p = pow(convolution2D(i, j+3, input, horiz_operator), 2) +
                pow(convolution2D(i, j+3, input, vert_operator), 2);
        res = (int)sqrt(p);
        if (res > 255)
                output[i*SIZE + j+3] = 255;
        else
                output[i*SIZE + j+3] = (unsigned char)res;
}

for (j=1; j<SIZE-1; j=j+1) {

        t = pow((output[i*SIZE+j] - golden[i*SIZE+j]),2);
        PSNR += t;
}
```

- Now, if we run **code(b)** t using the **-fast** flag we have a small improvement in time, a bit less than 0.064 seconds. Unlike code(a), in code(b) the **PSNR** remains INF

- Also, if we run **code(b)** using the **-O0** flag we have an **important** improvement in time. New best time for **-O0** flag is 5.8283991 seconds

- Finally, I tried to merge the nested in into a (SIZE-1)*(SIZE-1) iterations but without success

- In the spreadsheet, **code(a)** corresponds to numbers number 15 and 16, **code(b)** to 17 and 18

- So we keep the **code(b)** implementation and continue to the next optimization method.

LOOP INLINING

- Optimizations began from the source code 3**_loopFusion.c**

- In function **sobel ,** we call the function **convolution2D** two times. If we take out the optimizations with the loop unrolling and if we avoid calling the **convolution2D** function, the code looks like this: **(code (a))**

```
for (i=1; i<SIZE-1; i+=1) {
        for (j=1; j<SIZE-1; j+=1) {

                //horizontal
                tmp1 = 0;
                tmp1 += input[(i - 1)*SIZE + j -1] * horiz_operator[0][0];
                tmp1 += input[(i - 1)*SIZE + j + 0] * horiz_operator[0][1];
                tmp1 += input[(i - 1)*SIZE + j + 1] * horiz_operator[0][2];

                tmp1 += input[(i + 0)*SIZE + j - 1] * horiz_operator[1][0];
                tmp1 += input[(i + 0)*SIZE + j + 0] * horiz_operator[1][1];
                tmp1 += input[(i + 0)*SIZE + j + 1] * horiz_operator[1][2];

                tmp1 += input[(i + 1)*SIZE + j - 1] * horiz_operator[2][0];
                tmp1 += input[(i + 1)*SIZE + j + 0 ] * horiz_operator[2][1];
                tmp1 += input[(i + 1)*SIZE + j + 1] * horiz_operator[2][2];

                tmp2 = 0;
                //vertical
                tmp2 += input[(i - 1)*SIZE + j -1] * vert_operator[0][0];
                tmp2 += input[(i - 1)*SIZE + j + 0] * vert_operator[0][1];
                tmp2 += input[(i - 1)*SIZE + j + 1] * vert_operator[0][2];

                tmp2 += input[(i + 0)*SIZE + j - 1] * vert_operator[1][0];
                tmp2 += input[(i + 0)*SIZE + j + 0] * vert_operator[1][1];
                tmp2 += input[(i + 0)*SIZE + j + 1] * vert_operator[1][2];

                tmp2 += input[(i + 1)*SIZE + j - 1] * vert_operator[2][0];
                tmp2 += input[(i + 1)*SIZE + j + 0 ] * vert_operator[2][1];
                tmp2 += input[(i + 1)*SIZE + j + 1] * vert_operator[2][2];


                p = pow( tmp1, 2) +
                        pow(tmp2, 2);
                res = (int)sqrt(p);
                if (res > 255)
                        output[i*SIZE + j] = 255;
                else
                        output[i*SIZE + j] = (unsigned char)res;
```

- If we count the time needed to execute, we have a new best time for **-O0** flag, at 5.56secs but the **-fast** time is worser

- Now if we try to unroll the inner loop with factor=2, we don't have either any improvement in **-fast** or **-O0**

- At this point, we create a branch in our code, because -fast uses different code from -O0 to run fast. So we are going to use **4_functInline.c** for the -fast flag and **3_loopFusion.c** for the -O0 flag

- In the spreadsheet, **code(a)** corresponds to numbers number 19 and 20

- So we keep the **code(a)** for -O0 implementation and continue to the next optimization method.

LOOP INLINING

- Optimizations began from the source code **3_loopFusion.c** for the -fast flag and **4_funcInline.c** for -O0 flag
4_funcInline.c

- Now start from the file that we had the fastest time for **-O0** flag

- If we look the **code (a)** from Loop Inlining, we will find out that some values are reused. We are going to put these values out of this loop, so the code will look like the below code(**code (a))**

```
for (i=1; i<SIZE-1; i+=1) {
        iMinus1 = (i-1)*SIZE;
        iMinus0 = (i)*SIZE;
        iPlus1   = (i+1)*SIZE;
        for (j=1; j<SIZE-1; j++) {
                jMinus1 = j -1;
                jMinus0 = j;
                jPlus1 = j + 1;

                //LOOP 1
                //horizontal
                tmp1 = 0;
                tmp1 += input[iMinus1 + jMinus1] * horiz_operator[0][0];
                tmp1 += input[iMinus1 + jMinus0] * horiz_operator[0][1];
                tmp1 += input[iMinus1 + jPlus1 ] * horiz_operator[0][2];

                tmp1 += input[iMinus0 + jMinus1] * horiz_operator[1][0];
                tmp1 += input[iMinus0 + jMinus0] * horiz_operator[1][1];
                tmp1 += input[iMinus0 + jPlus1 ] * horiz_operator[1][2];

                tmp1 += input[iPlus1 + jMinus1] * horiz_operator[2][0];
                tmp1 += input[iPlus1 + jMinus0] * horiz_operator[2][1];
                tmp1 += input[iPlus1 + jPlus1] * horiz_operator[2][2];

                //vertical
                tmp2 = 0;
                tmp2 += input[iMinus1 + jMinus1] * vert_operator[0][0];
                tmp2 += input[iMinus1 + jMinus0] * vert_operator[0][1];
                tmp2 += input[iMinus1 + jPlus1] * vert_operator[0][2];

                tmp2 += input[iMinus0 + jMinus1] * vert_operator[1][0];
                tmp2 += input[iMinus0 + jMinus0] * vert_operator[1][1];
                tmp2 += input[iMinus0 + jPlus1] * vert_operator[1][2];

                tmp2 += input[iPlus1 + jMinus1] * vert_operator[2][0];
                tmp2 += input[iPlus1 + jMinus0] * vert_operator[2][1];
                tmp2 += input[iPlus1 + jPlus1] * vert_operator[2][2];
```

- Measuring with -O0 flag we get a better time. However -fast does not make improve any improve.
- Now as you can see we have 3 accesses in memory per access. We can reduce it to 1 access and 2 moves. The following code shows that(**code (b))**

```
saveUpLeft = input[iMinus1 + jMinus1];
saveUp = input[iMinus1 + jMinus0];
saveUpRight = input[iMinus1 + jPlus1 ];
saveCentralLeft = input[iMinus0 + jMinus1];
saveCentral =  input[iMinus0 + jMinus0];
saveCentralRight = input[iMinus0 + jPlus1 ];
saveDownLeft = input[iPlus1 + jMinus1];
saveDown = input[iPlus1 + jMinus0];
saveDownRight = input[iPlus1 + jPlus1];

//LOOP 1
//horizontal
tmp1 = 0;
tmp1 += saveUpLeft * horiz_operator[0][0];
tmp1 += saveUp * horiz_operator[0][1];
tmp1 += saveUpRight * horiz_operator[0][2];

tmp1 += saveCentralLeft * horiz_operator[1][0];
tmp1 += saveCentral * horiz_operator[1][1];
tmp1 += saveCentralRight * horiz_operator[1][2];

tmp1 += saveDownLeft * horiz_operator[2][0];
tmp1 += saveDown * horiz_operator[2][1];
tmp1 += saveDownRight * horiz_operator[2][2];

//vertical
tmp2 = 0;
tmp2 += saveUpLeft * vert_operator[0][0];
tmp2 += saveUp * vert_operator[0][1];
tmp2 += saveUpRight  * vert_operator[0][2];

tmp2 += saveCentralLeft * vert_operator[1][0];
tmp2 += saveCentral * vert_operator[1][1];
tmp2 += saveCentralRight * vert_operator[1][2];

tmp2 += saveDownLeft * vert_operator[2][0];
tmp2 += saveDown * vert_operator[2][1];
```

(code (b))

- Now we get a much better time for **-O0** flag. The time using **-fast** flag doesn't improve its best performance**.**

- If we try something smarter like the picture below we dont take better results, so we don’t count that attempt

```
iMinus1 = -SIZE;
iMinus0 = 0;
iPlus1 = SIZE;
for (i=1; i<SIZE-1; i+=1) {
        iMinus1 = iMinus0;
        iMinus0 = iPlus1;
        iPlus1 = iPlus1 + SIZE;
        jMinus1 = -1;
        jMinus0 = 0;
        jPlus1 = 1;

        for (j=1;jMinus0<SIZE-1; j++) {
                jMinus1 = j -1;
                jMinus0 = j;
                jPlus1 = j + 1;

                jMinus1 = jMinus0;
                jMinus0 = jPlus1;
                jPlus1 = jPlus1 + 1;

                saveUpLeft = input[iMinus1 + jMinus1];
                saveUp = input[iMinus1 + jMinus0];
                saveUpRight = input[iMinus1 + jPlus1 ];
```

- So we keep **code(b)** as the fastest implementation of **-O0** until now

3_loopFusion.c

- Now start from the file that we had the fastest time for **-fast** flag
- So, Lets recall that code:

```
for (i=1; i<SIZE-1; i+=1) {
        for (j=1; j<SIZE-1; j+=4) {
                //LOOP 1
                p = pow(convolution2D(i, j, input, horiz_operator), 2) +
                        pow(convolution2D(i, j, input, vert_operator), 2);
                res = (int)sqrt(p);
                if (res > 255)
                        output[i*SIZE + j] = 255;
                else
                        output[i*SIZE + j] = (unsigned char)res;


                //LOOP 2
                p = pow(convolution2D(i, j+1, input, horiz_operator), 2) +
                        pow(convolution2D(i, j+1, input, vert_operator), 2);
                res = (int)sqrt(p);
                if (res > 255)
                        output[i*SIZE + j+1] = 255;
                else
                        output[i*SIZE + j+1] = (unsigned char)res;


                //LOOP 3
                p = pow(convolution2D(i, j+2, input, horiz_operator), 2) +
                        pow(convolution2D(i, j+2, input, vert_operator), 2);
                res = (int)sqrt(p);
                if (res > 255)
                        output[i*SIZE + j+2] = 255;
                else
                        output[i*SIZE + j+2] = (unsigned char)res;



                //LOOP 4
                p = pow(convolution2D(i, j+3, input, horiz_operator), 2) +
                        pow(convolution2D(i, j+3, input, vert_operator), 2);
                res = (int)sqrt(p);
                if (res > 255)
```

- If we try to apply here Loop Invariant code optimization using the **-fast,** we don't achieve better timing, so we keep that code without changing, hoping that next optimizations using **-fast** will improve the performance

- To summarize, the best time until now for **-fast** flag is 0.063534175 seconds(number 20 in spreadsheet and file: "3_loopFusion/3_loopFusion.c") and for **-O0** flag is 5.4476366667 seconds(number 25 in spreadsheet and file: "5_loopInvariant/5_loopInvariant.c")

- In the spreadsheet, **code(b)** corresponds to numbers number 25 and 26

- So we keep **code(b)** for the next optimization method

COMMON SUBEXPRESSION ELIMINATION

- Optimizations began from the source code **3_loopFusion.c** for the -fast flag and 5**_loopInvariant.c** for -O0 flag

- Starting from file 3_loopFusion.c where we had the fastest time for **-fast** flag, we have the following code

```
int convolution2D(int posy, int posx, const unsigned char *input, char operator[][3]) {
        int res;

        res = 0;


        res += input[(posy -1)*SIZE+ posx -1] * operator[0][0];
        res += input[(posy -1)*SIZE + posx ] * operator[0][1];
        res += input[(posy -1)*SIZE + posx + 1] * operator[0][2];

        res += input[(posy )*SIZE + posx -1] * operator[1][0];
        res += input[(posy )*SIZE + posx ] * operator[1][1];
        res += input[(posy )*SIZE + posx + 1] * operator[1][2];

        res += input[(posy + 1)*SIZE + posx -1] * operator[2][0];
        res += input[(posy + 1)*SIZE + posx ] * operator[2][1];
        res += input[(posy + 1)*SIZE + posx + 1] * operator[2][2];


        return res;
```

- As we can see, some calculations are repeated at 3 times each. Let's rewrite that code substituting the obsolete calculations with variable assignments

```
int convolution2D(int posy, int posx, const unsigned char *input, char operator[][3]) {
        int res;
        int value1, value2,value3;

        value1 = (posy -1)<<12;
        value2 = (posy)<<12;
        value3 = (posy +1)<<12;

        res = 0;
        res += input[value1+ posx -1] * operator[0][0];
        res += input[value1 + posx ] * operator[0][1];
        res += input[value1 + posx + 1] * operator[0][2];

        res += input[value2 + posx -1] * operator[1][0];
        res += input[value2 + posx ] * operator[1][1];
        res += input[value2 + posx + 1] * operator[1][2];

        res += input[value3 + posx -1] * operator[2][0];
        res += input[value3 + posx ] * operator[2][1];
        res += input[value3 + posx + 1] * operator[2][2];


        return res;
```

- However we don't have any improvement in the timing neither in **-fast** or **-O0** flag. So we don't take that into account.
- I also tried to reduce the numbers of calls to **convolution2D** function by merging the into one call the vertical and horizontal calculation, however there wasn't any improvement

- Now we continue with file 5**_loopInvariant.c**

```
iMinus1 = -SIZE;
iMinus0 = 0;
iPlus1 = SIZE;
for (i=1; i<SIZE-1; i+=1) {
        iMinus1 = iMinus0;
        iMinus0 = iPlus1;
        iPlus1 = iPlus1 + SIZE;
        jMinus1 = -1;
        jMinus0 = 0;
        jPlus1 = 1;

        for (j=1;jMinus0<SIZE-1; j++) {
                jMinus1 = j -1;
                jMinus0 = j;
                jPlus1 = j + 1;

                jMinus1 = jMinus0;
                jMinus0 = jPlus1;
                jPlus1 = jPlus1 + 1;

                saveUpLeft = input[iMinus1 + jMinus1];
                saveUp = input[iMinus1 + jMinus0];
                saveUpRight = input[iMinus1 + jPlus1 ];
```

- We start changing that code, so that we reduce **memory accesses**

- The idea is that in every iteration we **shift** a 3x3 array one position right, until the end of the line. So we dont't have to access the memory for every single access, but we access only one new. Lets see the modified code. **((code a1))**

```
for (i=1; i<SIZE-1; i+=1) {
        saveUpLeft = input[(i-1)*SIZE];
        saveUp = input[(i-1)*SIZE+1];
        saveUpRight = input[(i-1)*SIZE+2];
        saveCentralLeft = input[i*SIZE];
        saveCentral =  input[i*SIZE+1];
        saveCentralRight = input[i*SIZE+2 ];
        saveDownLeft = input[(i+1)*SIZE];
        saveDown = input[(i+1)*SIZE+1];
        saveDownRight = input[(i+1)*SIZE+2];


        for (j=1;j<SIZE-1; j++) {
```

- Here we initialize our 3x3 tetragon for every new value of i and we shift the tetragon for every new value of j at the end of (i,j) iteration**((code a2))**

```
        if (res > 255)
                output[i*SIZE + j] = 255;  //output[iMinus0 + jMinus0]
        else
                output[i*SIZE + j] = (unsigned char)res;


        saveUpLeft = saveUp;
        saveUp = saveUpRight;
        saveUpRight = input[(i-1)*SIZE +j+2];
        saveCentralLeft = saveCentral;
        saveCentral =  saveCentralRight;
        saveCentralRight = input[(i)*SIZE +j+2];
        saveDownLeft = saveDown;
        saveDown = saveDownRight;
        saveDownRight = input[(i+1)*SIZE +j+2];
```

- Now if we measure the timing, we achieve new best time, 5.3542seconds for **-O0** flag. No improvements using the **-fast** flag

In the spreadsheet, **code(a)** corresponds to numbers number 27 and 28

- So we keep **code(a)** and we continue to the next optimization

Strength Reduction

- Optimizations began from the source code **3_loopFusion.c** for the **-fast** flag and **6_subExpression.c** for **-O0** flag

- Starting from file 3_loopFusion.c where we had the fastest time for **-fast** flag, we have the following **code(a1,a2)**, where we substitute all our multiplications by **12-bit-left-shift**, assuming that SIZE=4096

```
    res += input[((posy -1)<<12)+ posx -1] * operator[0][0];
    res += input[((posy -1)<<12) + posx ] * operator[0][1];
    res += input[((posy -1)<<12) + posx + 1] * operator[0][2];

    res += input[((posy )<<12) + posx -1] * operator[1][0];
    res += input[((posy )<<12) + posx ] * operator[1][1];
    res += input[((posy )<<12) + posx + 1] * operator[1][2];

    res += input[((posy + 1)<<12) + posx -1] * operator[2][0];
    res += input[((posy + 1)<<12) + posx ] * operator[2][1];
    res += input[((posy + 1)<<12) + posx + 1] * operator[2][2];


    return res;
```

```
..
p = pow(convolution2D(i, j, input, horiz_operator), 2) +
        pow(convolution2D(i, j, input, vert_operator), 2);
res = (int)sqrt(p);
if (res > 255)
        output[(i<<12) + j] = 255;
else
        output[(i<<12) + j] = (unsigned char)res;


//LOOP 2
p = pow(convolution2D(i, j+1, input, horiz_operator), 2) +
        pow(convolution2D(i, j+1, input, vert_operator), 2);
res = (int)sqrt(p);
if (res > 255)
        output[(i<<12) + j+1] = 255;
else
        output[(i<<12) + j+1] = (unsigned char)res;
```

- If we measure time, the performance is not improved at all. However, if we try to avoid the multiplications with the **horizontal** and **vertical** matrices, then the code will look like the following

```
int convolution2D(int posy, int posx, const unsigned char *input) {
        int resA, resB;
        resA = 0;
        resB = 0;
        //horizontal
        resA -= input[((posy -1)<<12)+ posx -1];//-1
        resA += input[((posy -1)<<12) + posx + 1];//1

        resA -= (input[((posy )<<12) + posx -1] <<1);//-2
        resA += (input[((posy )<<12) + posx + 1] << 1);//2

        resA -= input[((posy + 1)<<12) + posx -1];//-1
        resA += input[((posy + 1)<<12) + posx + 1];//1


        //vertical

        resB += input[((posy -1)<<12)+ posx -1];//1
        resB += (input[((posy -1)<<12) + posx ]<<1);//2
        resB += input[((posy -1)<<12) + posx + 1];//1

        resB -= input[((posy + 1)<<12) + posx -1] ;//-1
        resB -= (input[((posy + 1)<<12) + posx ] <<1);//-2
        resB -= input[((posy + 1)<<12) + posx + 1];//-1


        return pow(resA,2) + pow(resB,2);
```

- If we count the time using **-fast** flag, we get a great improvement in performance, reaching 0.0508895 seconds. We can also add a small change in the computation of PSNR, in case that our PSNR is often **INF**

```
        if (PSNR == 0)
                PSNR = (double)1/0;
        else {
                PSNR /= (double)((SIZE<<12));
                PSNR = 10*log10(65536/PSNR);
        }
```

- If we count using the **-O0** flag, the execution time is now slightly better than before. So we keep that implementation and we continue with **6_subExpression.c,** which achieved the best time for -O0 flag

6_subExpression.c

- Now let's recall our best code for **-O0** flag in our previous optimization, were we have achieved best time = 5.4476366667seconds

```
for (i=1; i<SIZE-1; i+=1) {
        //iMinus1 = (i-1)*SIZE;
        //iMinus0 = (i)*SIZE;
        //iPlus1   = (i+1)*SIZE;
        saveUpLeft = input[(i-1)*SIZE];
        saveUp = input[(i-1)*SIZE+1];
        saveUpRight = input[(i-1)*SIZE+2];
        saveCentralLeft = input[i*SIZE];
        saveCentral =  input[i*SIZE+1];
        saveCentralRight = input[i*SIZE+2 ];
        saveDownLeft = input[(i+1)*SIZE];
        saveDown = input[(i+1)*SIZE+1];
        saveDownRight = input[(i+1)*SIZE+2];

        for (j=1;j<SIZE-1; j++) {

                //LOOP 1
                //horizontal
                tmp1 = 0;
                tmp1 += saveUpLeft * horiz_operator[0][0];
                tmp1 += saveUp * horiz_operator[0][1];
                tmp1 += saveUpRight * horiz_operator[0][2];

                tmp1 += saveCentralLeft * horiz_operator[1][0];
                tmp1 += saveCentral * horiz_operator[1][1];
                tmp1 += saveCentralRight * horiz_operator[1][2];

                tmp1 += saveDownLeft * horiz_operator[2][0];
                tmp1 += saveDown * horiz_operator[2][1];
                tmp1 += saveDownRight * horiz_operator[2][2];

                //vertical
                tmp2 = 0;
                tmp2 += saveUpLeft * vert_operator[0][0];
                tmp2 += saveUp * vert_operator[0][1];
                tmp2 += saveUpRight  * vert_operator[0][2];
```

- Let’s do the same arithmetic optimizations as before: **(code (b))**

```
for (j=1;j<SIZE-1; j++) {
        //LOOP 1

        //horizontal
        tmp1 = 0;
        tmp1 -= saveUpLeft;
        tmp1 += saveUpRight;
        tmp1 -= (saveCentralLeft<<1);
        tmp1 += (saveCentralRight<<1);
        tmp1 -= saveDownLeft;
        tmp1 += saveDownRight;

        //vertical
        tmp2 = 0;
        tmp2 += saveUpLeft;
        tmp2 += (saveUp<<1);
        tmp2 += saveUpRight;
        tmp2 -= saveDownLeft;
        tmp2 -= (saveDown<<1);
        tmp2 -= saveDownRight;


        p = pow( tmp1, 2) +
                pow(tmp2, 2);
        res = (int)sqrt(p);
        if (res > 255)
```

- Now the time measured is 5.1141308, improving our **-O0** best execution time. We have also achieved to reduce the code size, because some multiplications produce zero. So we keep that implementation for the -O0 flag and we c

- In the spreadsheet, **code(a1,a2)** corresponds to numbers number (29,30) and **code(b)** to numbers(31,32)

- So we keep **code(a1,a2)** for -fast flag, which is in **7_strengthReduction_fast.c** and **code(b)** for the next optimization method, which is in **7_strengthReduction_O0.c**

Compiler Optimization Techniques

- Optimizations began from the source code **7_strengthReduction_fast.c** for the **-fast** flag and **7_strengthReduction_O0.c** for **-O0** flag

- Lets begin with **7_strengthReduction_fast.c.** Despite the fact that our code isn't appropriate for adding **restrict** or **const** parameters to variables, we are going to use the **register** keyword for variables **i**nt resA, resB, int j; int j; int p; **(code (a))**

```
double sobel(unsigned char *input, unsigned char *output, unsigned char *golden)
{
        double PSNR = 0, t;
        register int i, j;
        register unsigned int p;
        register int res;
        struct timespec  tv1, tv2;
        FILE *f_in, *f_out, *f_golden;
```

- If we count the average time, the performance increases the performance for -fast flag, reaching now 0.0494872417seconds. Now its time to see if there is any optimization our 2nd file, improving the **-O0** time

7_strengthReduction_O0.c

- Let's try to do the similar process, putting the **register** keyword in variables that are often used. The folllowing code explains that **(code(b))**

```
double sobel(register unsigned char *input, register unsigned char *output, unsigned char *golden)
{
        double PSNR = 0, t;
        int i, j;
        unsigned int p;
        register int tmp1, tmp2;

        int saveUpLeft;
        int saveUp;
        int saveUpRight;
        int saveCentralLeft;
        int saveCentral;
        int saveCentralRight;
        int saveDownLeft;
        int saveDown;
        int saveDownRight;

        register int res;
```

- However, after applying that optimization to our code using the **-O0** flag the performance gets worser reaching 5.12582seconds, regardless of how many variables are defined as registers. I also tried to put also the 9 saveVariables(see above these), but there was no improvement in performance. So we omit that optimizations

- So we keep only the code that implements faster the **-fast** flag, which exists in file 8_compilerTechniques_fast.c

- The best code for the **-O0** flag remains the file 7_strengthReduction_O0.c

TO SUMMARIZE...

-fast

- Until the **Loop Fusion** technique, every optimization in our code gives us improvement in performance. The next improvement has been noticed in the **Strength Reduction** technique and finally achieves best time in **Compiler Techniques**
-O0

- Every technique from the beginning improved the time in our code , except for **Loop Unroll** and **Common Subexpression Elimination**