CS 2461 - Lauren Schmidt
Project 6: Code Optimization Report

Overall Optimization Results:

	Time in Milliseconds				Time in Milliseconds		
Dimension	Naive_rotate	My_rotate	Speedup	Dimension	Naive_smooth	My_smooth	Speedup
512	1031	905	113.9226519	512	7053	3890	181.311054
1024	7608	3880	196.0824742	1024	30908	16011	193.0422834
2048	63734	30523	208.8064738	2048	148354	65747	225.6437556
4096	516868	134177	385.2135612	4096	784018	262802	298.3303019
Avg			226.0062903				224.5818487

Overall, I was able to optimize my_rotate by 126% and my_smooth by 124% on the best runs. The speedup averages varied slightly each time I ran the code, however, the results were always over 100% faster than the naive versions of the methods.

I only measured the time in milliseconds as reported by driver.c, and although cycle counts are important, they were not used as the primary measure of efficiency in this report. To calculate the increase in efficiency, I averaged together the speedup calculations for each of the different dimensions.

Optimizations for my_rotate:

I. Replaced RIDX with Inline Calculations

When I replaced each call to RIDX with a manual calculation, it increased the efficiency of my_rotate by 7%. This slight increase in efficiency occurs because the function now has improved locality, and does not have to access a method defined elsewhere in memory. Instead, it can quickly compute the calculations done by RIDX all while staying within the my_rotate method.

1. Replaced RIDX with Inline Calculations			
	Time in Milliseconds		
Dimension	Naive_rotate	My_rotate	Speedup
512	974	957	101.7763845
1024	9139	7477	122.2281664
2048	62228	58438	106.485506
4096	456510	455067	100.3170962
Avg			107.7017883

Code implemented:

```
for (j = 0; j < dim; j++)
   for (i = 0; i < dim; i++)
       dst[(dim - 1 - j) * (dim) + (i)] = src[(i) * (dim) + (j)];
```

II. Implemented Memory Blocking

Memory blocking rewrites the code to process individual blocks of data at a time. Instead of parsing through the entire 2-D matrix of pixel values all at once (which takes a larger amount of time), the program can tackle the computations block by block. Because the test dimensions of the matrices begin at 512 for my_rotate(), I decided to work up through different powers of 2 for the block size until I found the most efficient for this dataset.

A. Trial 1: Blocking by Size 4

2. Implemented memory blocking, Increment by 4			
	Time in Milliseconds		
Dimension	Naive_rotate	My_rotate	Speedup
512	1089	1159	93.96031061
1024	7424	5679	130.7272407
2048	52591	41812	125.7796805
4096	593904	275552	215.5324585
Avg			141.4999226

Blocking by a size of 4 resulted in a 41% increase in efficiency.

Code implemented:

```
   // Replace common subexpression dim-1
```

```
int x = dim - 1;

// Memory block by 16 at a time
int ii;
int jj;

for (j = 0; j < dim; j += 4)
    for (i = 0; i < dim; i += 4)
        for (jj = j; jj < j + 4; jj++)
            for (ii = i; ii < i + 4; ii++)
                dst[(x - jj) * (dim) + (ii)] = src[(ii * dim) + (jj)];
```

B. Trial 2: Blocking by Size 16

2. Implemented memory blocking incrementing by 16

	Time in Milliseconds		
Dimension	Naive_rotate	My_rotate	Speedup
512	1058	977	108.2906858
1024	7628	5586	136.5556749
2048	67871	33258	204.0742077
4096	501133	135551	369.7007031
Avg			204.6553179

Blocking by a size of 16 resulted in the largest increase in efficiency by far, with code that was 104% more efficient than naive_rotate.

Running the same code again, for a second trial to confirm results after running `clean` and `make` again resulted in a 126% increase in efficiency:

2. Second trial + 16

	Time in Milliseconds		
Dimension	Naive_rotate	My_rotate	Speedup
512	1031	905	113.9226519
1024	7608	3880	196.0824742
2048	63734	30523	208.8064738
4096	516868	134177	385.2135612
Avg			226.0062903

Code implemented:

```
// Replace common subexpression dim-1
int x = dim - 1;

// Memory block by 16 at a time
int ii;
int jj;
```

```
for (j = 0; j < dim; j += 16)
    for (i = 0; i < dim; i += 16)
        for (jj = j; jj < j + 16; jj++)
            for (ii = i; ii < i + 16; ii++)
                dst[(x - jj) * (dim) + (ii)] = src[(ii * dim) + (jj)];
```

C. Trial 3: Blocking by Size 32

2. Implemented memory blocking + 32			
	Time in Milliseconds		
Dimension	Naive_rotate	My_rotate	Speedup
512	1157	1112	104.0467626
1024	7756	4952	156.6235864
2048	60977	36115	168.8412017
4096	587444	194431	302.1349476
Avg			182.9116246

Because of the efficiency of the results for blocking with a size of 16, I decided to up the increments in the for-loops to a size of 32. However, this actually *decreased* the efficiency of my program, and as a result, I decided to test no higher than a block size of 16. The results for a block size 32 are pictured, however, I stuck with the size-16 block as it was the most efficient. 16 works the best because, in this case, that is the size of the cache.

III. Working Optimized Code

The final optimized code ran 126% more efficiently than the naive code, and the complete implementation can be seen below:

```
// Replace common subexpression dim-1
int x = dim - 1;

// Memory block by 16 at a time
int ii;
int jj;

for (j = 0; j < dim; j += 16)
    for (i = 0; i < dim; i += 16)
        for (jj = j; jj < j + 16; jj++)
            for (ii = i; ii < i + 16; ii++)
                dst[(x - jj) * (dim) + (ii)] = src[(ii * dim) + (jj)];
```

Optimizations for my_smooth:

I. RIDX Replacement, Loop Interchange

Loop interchange and calculating RDIX inline			
	Time in Milliseconds		
Dimension	Naive_smooth	My_smooth	Speedup
256	7360	7237	101.6995993
512	31609	29548	106.9750914
1024	155496	119496	130.1265314
2048	850731	577262	147.373463
Avg			121.5436713

Similarly to the my_rotate() method, I replaced the RIDX method calls with the actual calculation done by the calls in order to increase the overall efficiency of the program.

I also changed the i and j loop order. Previously, the j loop came first, which was causing the program to run less efficiently because C stores in row-major order, and the loop was accessing the matrix in column-major order. Changing the order of the i and j loops and writing the RIDX methods inline resulted in a 21% total increase in efficiency.

Code implemented:

```
for (i = 0; i < dim; i++)
   for (j = 0; j < dim; j++)
       dst[(i) * (dim) + (j)] = avg(dim, i, j, src);
```

II. Replacing Method Calls with Inline Calculations

The change that made the largest difference in program efficiency was the replacement of method calls with inline calculations. The naive_smooth method contain a call to the average function, which contains nested calls to other functions. Because the computer has to access several different locations in memory to access each individual function, the program has poor locality and takes longer to run.

If we replace each of the following method calls with their inline equivalents and perform all calculations in the main method, everything can be accessed by the driver in fast memory, and the program will run faster.

A. average and assign_sum_to_pixel

First, I implemented average in the main method by copying

Replacing Inline Functions Avg & Assign_sum_to_pixel			
	Time in Milliseconds		
Dimension	Naive_smooth	My_smooth	Speedup
256	6848	5898	116.107155
512	29672	24579	120.7209406
1024	146828	102448	143.3195377
2048	746189	399600	186.733984
Avg			141.7204043

over the code called in the average function. I did the same for the assign_sum_to_pixel function. I had to create some new variables (like sum and current_pixel) to accommodate for this change, and ended up with a nested for-loop structure that looked similar to the memory blocking from before. This change, which still made calls from main to accumulate_sum, minimum, and maximum, increased the efficiency of the program from 21% faster than naive_smooth to 41% faster than naive_smooth.

Code implemented:

```
   int ii, jj;
   pixel_sum sum;
   pixel current_pixel;

   for (i = 0; i < dim; i++)
   {
       for (j = 0; j < dim; j++)
       {
           pixel_sum *newSum = &sum;
           newSum->red = newSum->green = newSum->blue = newSum->num = 0;
           for (ii = maximum(i - 1, 0); ii <= minimum(i + 1, dim - 1);
ii++)
           {
               for (jj = maximum(j - 1, 0); jj <= minimum(j + 1, dim -
1); jj++)
               {
                   accumulate_sum(&sum, src[RIDX(ii, jj, dim)]);
               }
           }

           pixel *ptr = &current_pixel;
           ptr->red = (unsigned short)(sum.red / sum.num);
           ptr->green = (unsigned short)(sum.green / sum.num);
           ptr->blue = (unsigned short)(sum.blue / sum.num);
           dst[i * dim + j] = current_pixel;
       }
   }
```

B. accumulate_sum
Similar to the way that I implemented the previous two functions, I copied over the calculations done by accumulate_sum into the main function to improve the program's locality. This included creating a new pixel variable (accumulated) that

Replacing accumulate_sum

	Time in Milliseconds		
Dimension	Naive_smooth	My_smooth	Speedup
256	7077	5858	120.8091499
512	30748	24833	123.8191117
1024	147593	99421	148.4525402
2048	744826	398795	186.769142
Avg			144.9624859

would take the value of src[RIDX(ii, jj, dim)]. The accumulated sum would need this new pixel's values added to it.

Overall, implementing accumulate_sum inline instead of via method call increased the efficiency to 44%. Although it was a minor improvement, I still found it necessary to accomplish over 100% efficiency.

Code implemented:

```
   pixel_sum sum;
   pixel current_pixel;

   for (i = 0; i < dim; i++)
   {
       for (j = 0; j < dim; j++)
       {
           pixel_sum *newSum = &sum;
           newSum->red = newSum->green = newSum->blue = newSum->num = 0;
           // ii and jj loops replace the "average" function call and
make it inline instead.
           for (ii = maximum(i - 1, 0); ii <= minimum(i + 1, dim - 1);
ii++)
           {
               for (jj = maximum(j - 1, 0); jj <= minimum(j + 1, dim -
1); jj++)
               {
                   pixel accumulated = src[(ii) * (dim) + (jj)];
                   newSum->red += (int)accumulated.red;
                   newSum->green += (int)accumulated.green;
                   newSum->blue += (int)accumulated.blue;
                   newSum->num++;
                   // accumulate_sum(&sum, src[RIDX(ii, jj, dim)]);
               }
           }
           // Putting the assign_sum_to_pixel inline insetad of making a
method call.
```

```
            pixel *ptr = &current_pixel;
            ptr->red = (unsigned short)(sum.red / sum.num);
            ptr->green = (unsigned short)(sum.green / sum.num);
            ptr->blue = (unsigned short)(sum.blue / sum.num);
            dst[i * dim + j] = current_pixel;
        }
    }
```

C. Minimum and maximum
My final inline replacement had to do with the minimum and maximum methods, which were still being called in the ii and jj for-loops. I simply copied over the calculations done by both of those methods into the conditions for the for-loops, and the resulting efficiency was 122%.

Replacing minimum, maximum

	Time in Milliseconds		
Dimension	**Naive_smooth**	**My_smooth**	**Speedup**
256	6659	4018	165.7292185
512	29481	16276	181.1317277
1024	149355	62624	238.4948263
2048	773966	253310	305.5410367
Avg			**222.7242023**

Code implemented:

```
int ii, jj;

   pixel_sum sum;
   pixel current_pixel;

   for (i = 0; i < dim; i++)
   {
       for (j = 0; j < dim; j++)
       {
           pixel_sum *newSum = &sum;
           newSum->red = newSum->green = newSum->blue = newSum->num = 0;

           // ii and jj loops replace the "average" function call and make it
inline instead.
           for (ii = (i - 1 > 0 ? i - 1 : 0); ii <= (i + 1 < dim-1 ? i + 1 :
dim-1); ii++)
           {
               for (jj = (j - 1 > 0 ? j - 1 : 0); jj <= (j + 1 < dim-1 ? j + 1
: dim-1); jj++)
```

```
                {

                    // Inline "accumulate_sum" function instead of making a
method call outside of main.
                    pixel accumulated = src[(ii) * (dim) + (jj)];
                    newSum->red += (int)accumulated.red;
                    newSum->green += (int)accumulated.green;
                    newSum->blue += (int)accumulated.blue;
                    newSum->num++;
                }
            }
            // Putting the assign_sum_to_pixel inline insetad of making a method
call.
            pixel *ptr = &current_pixel;
            ptr->red = (unsigned short)(sum.red / sum.num);
            ptr->green = (unsigned short)(sum.green / sum.num);
            ptr->blue = (unsigned short)(sum.blue / sum.num);
            dst[i * dim + j] = current_pixel;
        }
    }
```

III. Replacing Common Subexpressions

A. (dim - 1)

This was one of the final changes that I made to my_smooth. I noticed that the subexpression (dim-1) was appearing often in the code, so I replaced it with the integer variable x. With this change, the value of x would be pre-calculated, and would not have to be computed inside of each for-loop every time it was needed for a conclusion. The effect on the efficiency was small, but overall made the program 124% more efficient than naive_smooth.

Replacing common subexpressions dim-1 = x

	Time in Milliseconds		
Dimension	**Naive_smooth**	**My_smooth**	**Speedup**
256	7053	3890	181.311054
512	30908	16011	193.0422834
1024	148354	65747	225.6437556
2048	784018	262802	298.3303019
Avg			**224.5818487**

Code implementation:

See working optimized code below.

IV. Working Optimized Code

The final optimized code ran 124% more efficiently than the naive code, and the complete implementation can be seen below:

```
int ii, jj;

   pixel_sum sum;
   pixel current_pixel;

   // Replace common subexpression dim-1
   int x = dim - 1;

   for (i = 0; i < dim; i++)
   {
       for (j = 0; j < dim; j++)
       {
           pixel_sum *newSum = &sum;
           newSum->red = newSum->green = newSum->blue = newSum->num = 0;

           // ii and jj loops replace the "average" function call and make it
inline instead.
           for (ii = (i - 1 > 0 ? i - 1 : 0); ii <= (i + 1 < x ? i + 1 : x);
ii++)
           {
               for (jj = (j - 1 > 0 ? j - 1 : 0); jj <= (j + 1 < x ? j + 1 :
x); jj++)
               {

                   // Inline "accumulate_sum" function instead of making a
method call outside of main.
                   pixel accumulated = src[(ii) * (dim) + (jj)];
                   newSum->red += (int)accumulated.red;
                   newSum->green += (int)accumulated.green;
                   newSum->blue += (int)accumulated.blue;
                   newSum->num++;
               }
           }
           // Putting the assign_sum_to_pixel inline insetad of making a method
call.
           pixel *ptr = &current_pixel;
           ptr->red = (unsigned short)(sum.red / sum.num);
           ptr->green = (unsigned short)(sum.green / sum.num);
           ptr->blue = (unsigned short)(sum.blue / sum.num);
           dst[i * dim + j] = current_pixel;
```

```
        }
    }
```