Rhys Chambers
Computer Architecture
Professor Narahari

**Project 6: Code Optimization of an Image**

Part 1: Rotate

*Thought process to reach optimized code:*

To begin the project, I needed to understand what each variable meant. At first, I thought that dst was representing a two-dimensional array since we were working with i and j; however, I drew an example grid and then understood that dst[] was a one-dimensional array where the value inside of the brackets represented a particular pixel at location i, j. Next, I needed to understand what the two loops in rotate were actually doing so that I could better optimize the code. I noticed right away that there was a call to the function RIDX that we might be able to optimize using function inlining.

My first instinct was to expand the code inside the dst[RIDX([dim-1-j, i, dim)] and src[RIDX(i, j, dim)] so that I could remove one bookkeeping instruction. This resulted in: dst[RIDX((dim-1-j)(dim) + i)] = src[(i*dim) + j]. So, I also made a dimSquared variable so that I could distribute the operations inside the multiplication of dim so that it would only have to be calculated once. I also decided that I needed to trace through the code in a little bit more detail to see if there were any patterns that I was missing. So, I created tables in my notebook like the one below to show a few different combinations of i and j.

RIDX function (i, j, n): `((i)*(n)+(j))`

Dst[dim-1-j, i, dim]

| Dim (for example, 32) | j | i | RIDX: (dim-1-j)(dim) + i | change |
|---|---|---|---|---|
| 32 | 0 | 0 | 992 | +1 |
| 32 | 0 | 1 | 993 | +1 |
| 32 | 0 | 2 | 994 | |
| 32 | 1 | 0 | 960 | Original - dim |

| | | | | |
|---|---|---|---|---|
| 32 | 1 | 1 | 961 | |
| 32 | 2 | 0 | 928 | Original - j*dim |

Observation: for every one increase in the outer loop value, the value of RIDX decreases by dim

Src[i, j, dim]

| Dim (for example, 32) | j | i | RIDX: (i*dim) + j |
|---|---|---|---|
| 32 | 0 | 0 | 0 |
| 32 | 0 | 1 | 32 |
| 32 | 0 | 2 | 64 |
| 32 | 1 | 0 | 1 |
| 32 | 1 | 1 | 33 |
| 32 | 1 | 2 | 65 |
| 32 | 2 | 0 | 2 |
| 32 | 2 | 1 | 35 |
| 32 | 2 | 2 | 67 |

*Observation: each time the inner loop runs, the value of RIDX increases by dim

Here I could see that the value of dst depended on the outer loop, while the value of src depended on the inner loop. Still, I did not have enough of an understanding of how the rotation was actually happening. So, I drew a small example of a grid like the one below to help me visualize what a rotation would look like on the grid. For the sake of simplicity, I chose to visualize a grid of size 16; however, I know that the grid is a multiple of 32.

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

Grid rotated by 90 degrees:

| | | | |
|---|---|---|---|
| 15 | 14 | 13 | 12 |
| 7 | 10 | 9 | 8 |
| 11 | 6 | 5 | 4 |
| 3 | 2 | 1 | 0 |

In order to reach the fully rotated image, you must set the destination (let it be the top left corner) equal to the source (bottom right) pixel. Then, you need to shift every pixel following the destination pixel equal to the source. After one round, of shifts you would have the destination of the first pixel rotated correctly, followed by each subsequent pixel shifted to a space to the right. Once I made this realization, the dst and src values from above made more sense, and I was able to relate them to each other visually. In order to rotate an entire image you need to swap the destination pixel and the source pixel, shift all of the remaining pixels right (or +1), and then set the destination + 1 equal to the new source. This is when I realized that we could use blocking with the inner and outer loop as a method of improving cache hits/overall performance.
This inner process will happen as many times as the dimension. We can improve cache hit rate since we know that the size of the grid is a multiple of 32, so the outer loop will increment by 32 with each iteration.

*Coding thought process:*
Figuring out how the rotation function worked with higher level programming

```
 k = 32;
           while(k >=0){
           *(dst + k) = *src;
           src += dim;
           k--;
           }
```

Starting to implement rotate by breaking it down with blocking

```
for (i = 0; i < dim; i += 32){
       // int dimSqrd = dim*(dim); moved to above, originally did not work
       // dst += dimSqrd  - dim;
       for (j = 0; j < dim; j++){
           *dst = *src;
           src += dim;

           k = 32;
```

```
while(k >=0){
*(dst + k) = *src;
src += dim;
k--;
}
```

Techniques used:

1. Inlining
    a. Inlining did not really improve code performance that much since there was only one function call within the loops, but it was an improvement nonetheless. Inlining improved my code performance by about 5%. I inlined the RIDX functions. Although it isn't technically inlining, I also broke up the dim*dim portion into its own variable, since performance would be improved by only calculating dim*dim once.
2. Improving cache performance
    a. Blocking
        i. After I made the first two tables above, blocking became more of a relevant technique to be used in my code optimization. I knew that if I could reduce the number of times that my outer loop ran, I could significantly improve the overall performance of the code by reducing cache misses. This is why I chose to increment my outer loop by 32.

3. Loop interchange
    a. I switched the i and j loops to improve cache access performance. It helped speed up my code by about 3%. Loop interchange helped speed up performance since the new code would be operating in row-major order, which improves the locality of the program.

| Dimension | naive_rotate | my_rotate | Speedup |
|---|---|---|---|
| **256** | 1068 | 687 | 155.46% |
| **512** | 8048 | 2916 | 275.99% |
| **1024** | 61551 | 21502 | 286.26% |
| **2048** | 418870 | 105006 | 398.90% |
| | | Average: | 279.15% |

**Smooth**

The smooth portion of the project was a lot harder for me to understand how to improve compared to the rotation. I also spent more time on the rotate portion so that I had a really good understanding of what was going on behind each line of code. I think I could have definitely spent more time trying to understand each individual step of smooth more, and I could have been able to effectively make improvements to the speed.

I started out by looking at the first function call that is encountered, which is RIDX. We have used RIDX before, so I inlined it with the actual function, (i*n)+j. Next, I needed to inline the avg function. The average function also calls initialize_pixel_sum, minimum, maximum, RIDX, accumulate_sum, and assign_sum_to_pixel. All of these functions could be inlined within the average function. I checked my work along the way by keeping the original function calls and commenting out other function calls to make sure that the original smooth function and my implementation were both working the same way. The most difficult part of inlining with these function calls was creating the new variables that would have originally been passed into the functions since we are switching between variable addresses and the dereferenced values in some of the function calls and implementations.

The only optimization technique that I tried was inlining. This optimization increased my code efficiency by 25-32% depending on the actual run time. I definitely think that there is a way that I could keep improving the code by using blocking and possibly breaking up the blurring by region, but I think that this improvement might take me another couple of days to figure out.

Functions inlined:

Calls to maximum became: `((j-1) > 0 ? (j-1) : 0 or (i-1) > 0 ? (i-1) : 0)`

Calls to minimum became: `(j+1 < dim -1 ? j+1 : dim-1) or ii <= (i+1 < dim -1 ? i+1 : dim-1)`

Call to initialize_pixel_sum became:

```
sum.red = sum.green = sum.blue = 0;
sum.num = 0;
```

Call to RIDX became: `(i*dim)+j`

Call to accumulate_sum became:

```
// pixel_sum *sumAcc = &sum;
    pixel p =  src[((ii*dim)+jj)]; //RIDX(ii, jj, dim)];

    sumAcc->red += (int) p.red;
    sumAcc->green += (int) p.green;
    sumAcc->blue += (int) p.blue;
    sumAcc->num++;
```

```
            }
```

Assign_sum_to_pixel became:

```
pixel* curPix;
   curPix = &current_pixel;


          curPix->red = (unsigned short) (sum.red/sum.num);
          curPix->green = (unsigned short) (sum.green/sum.num);
          curPix->blue = (unsigned short) (sum.blue/sum.num);
```

Finally, instead of returning current_pixel, I set dst[((i*dim)+j)] = current_pixel;

Optimizations tried:

1. Inlining
    a. Inlining was the primary optimization technique that I used to improve my code. I inlined every function in the average function. Then, once I could call average and my smooth function ran correctly, I took my inlined code from average and put it directly into the loop in smooth where the average function was originally called. Inlining was pretty effective since it improved my code performance by about 25-32% depending on run time.
2. Loop interchange
    a. Improved performance by about 5%. I interchanged the ii and jj loops so that I could improve the locality of the inlined average function. This loop interchange did not improve performance a lot because i and j can never be more than dim apart, and the image is a square. Therefore, there are limitations to how much the interchange can actually improve performance.
3. Blocking–not used but I could see how it might be implemented
    a. I thought about blocking by section, such as the left corner, right corner, and bottom corners, but I wasn't sure how to break my loops up enough for it to be effective.

| Dimension | nalive_smooth | my_smooth | Speedup |
|---|---|---|---|
| **256** | 5211 | 3816 | 136.56% |
| **512** | 23473 | 19800 | 118.55% |
| **1024** | 118330 | 92373 | 128.10% |
| **2048** | 601386 | 509675 | 117.99% |
| | | Average: | 125.30% |

*Coding thought process:*

Functions inside of average have been inlined, comments show some trial and error

```
static pixel avg(int dim, int i, int j, pixel *src)
{
   int ii, jj;
   pixel_sum sum;
   pixel_sum*sum2 = &sum;

   pixel current_pixel;
   pixel* curPix = &current_pixel;

   // pixel p;


   //inlined function
   sum.red = sum.green = sum.blue = 0;
   sum.num = 0;

   // initialize_pixel_sum(&sum);
   // for(jj = ((j-1) > 0 ? (j-1) : 0); jj <= (j+1 < dim -1 ? j+1 : dim-1); jj++)

       // ii = maximum(i-1, 0); ii <= minimum(i+1, dim-1); ii++)
           for(jj = ((j-1) > 0 ? (j-1) : 0); jj <= (j+1 < dim -1 ? j+1 : dim-1);
jj++){
               for(ii = ((i-1) > 0 ? (i-1) : 0); ii <= (i+1 < dim -1 ? i+1 : dim-1);
ii++){
               // jj = maximum(j-1, 0); jj <= minimum(j+1, dim-1); jj++)

               // p = src[RIDX(ii, jj, dim)];

               // sum2->red += (int) p.red;
               // sum2->green += (int)p.green;
               // sum2->blue += (int) p.blue;
               // sum2->num++;

                   // sum2->red += (int) p.red;
                   // sum2->green += (int) p.green;
                   // sum2->blue += (int) p.blue;
                   // sum2->num++;
```

```
                    // pixel_sum *sumAcc = &sum;
                    pixel p =  src[((ii*dim)+jj)]; //RIDX(ii, jj, dim)];

                    sumAcc->red += (int) p.red;
                    sumAcc->green += (int) p.green;
                    sumAcc->blue += (int) p.blue;
                    sumAcc->num++;

                // ridxScore = ((ii)*(dim)+(jj));

                // sum.red += (int) src[RIDX(ii, jj, dim)].red;

                // sum.green += (int) src[RIDX(ii, jj, dim)].green;


                // sum.blue += (int) src[RIDX(ii, jj, dim)].blue;




                // p = src[((ii)*(dim)+(jj))];
                }
            }

    // assign_sum_to_pixel(&current_pixel, sum);
    curPix->red = (unsigned short) (sum.red/sum.num);
    curPix->green = (unsigned short) (sum.green/sum.num);
    curPix->blue = (unsigned short) (sum.blue/sum.num);
    return current_pixel;

}
```