Project 6 - Ozzy Simpson

Overall Performance of Optimized Code:

Performance improvements by time in milliseconds

Dimension	naive_rotate	my_rotate	Speedup	naive_smooth	my_smooth	Speedup
256				6799	1940	350.46%
512	1020	753	135.46%	30430	7879	386.22%
1024	8149	3280	248.45%	155057	34212	453.22%
2048	54719	45690	119.76%	834334	134984	618.10%
4096	627518	158877	394.97%			
		Average:	224.66%		Average:	452.00%

Performance improvements by time in milliseconds

Dimension	naive_rotate	my_rotate	Speedup	naive_smooth	my_smooth	Speedup
256				13630951	3881243	351.20%
512	2069471	1463788	141.38%	60866341	15760623	386.19%
1024	14532535	6999360	207.63%	311164828	68428119	454.73%
2048	120437954	66039451	182.37%	1671769911	270061108	619.03%
4096	860793384	306333113	281.00%			
		Average:	203.09%		Average:	452.79%

my_rotate():

Function Inlining

While it is a macro (and not a function), we can inline the RIDX calculation. This will become helpful later on when we do code motion.

Before:	*After:*
`dst[RIDX(dim-1-j, i, dim)] = src[RIDX(i, j, dim)];`	`dst[(dim-1-j) * dim + i] = src[i * dim + j];`

This change resulted in a slight increase in performance because the calculations are performed inline and the macro doesn't have to be called; on average, it increased the performance by 3%:

Dimension	naive_rotate	my_rotate	Speedup
512	1084	1029	105.34%
1024	7239	6904	104.85%
2048	51228	51614	99.25%
4096	549993	534463	102.91%
		Average:	103.09%

Dataflow Analysis and Optimizations

There were not many data flow optimizations that could be performed. The original rotate function does not include any dead code and there were not any opportunities to do constant folding or copy propagation. There are some common expressions that can be simplified, but this will be done in the code motion section below because we'll move common expressions outside of some loops.

Code Motion and Strength Reduction

We can notice that RIDX() is called with dim-1-j as the first argument every iteration of the inner loop, but dim-1-j only changes with each iteration of the outer loop. We can simplify this by moving this invariant computation outside of the inner loop, storing it in a temporary variable, and then using that as one of the arguments for the dst index calculation, like this:

Before:	*After:*

`for (j = 0; j < dim; j++)`
`    for (i = 0; i < dim; i++)`
`        dst[RIDX(dim-1-j, i, dim)] = src[RIDX(i, j, dim)];`
	`int in;`
	`for (j = 0; j < dim; j++) {`
	`    in = dim - 1 - j;`
	`    for (i = 0; i < dim; i++)`
	`        dst[RIDX(in, i, dim)] = src[RIDX(i, j, dim)];`
	`}`

This change (alone) resulted in a more significant increase in performance, which is to be expected since we're calculating *in* fewer times (*dim* fewer times, to be exact). On average, this increased performance by 7%:

Dimension	naive_rotate	my_rotate	Speedup
512	1011	961	105.20%
1024	7940	6438	123.33%
2048	62486	61511	101.59%
4096	504535	507486	99.42%
		Average:	107.38%

Furthermore, we can combine this change with the previous change and further eliminate common subexpressions to calculate part of the index completely out of the loops and as much as we can of the rest just in the outer loop, such that the for loops have been changed as follows:

Before:	*After:*

`for (j = 0; j < dim; j++)`
`    for (i = 0; i < dim; i++)`
`        dst[RIDX(dim-1-j, i, dim)] = src[RIDX(i, j, dim)];`
	`int in;`
	`in = dim * dim - dim;`
	`for (j = 0; j < dim; j++) {`
	`    for (i = 0; i < dim; i++) {`
	`        dst[in + i] = src[i * dim + j];`
	`    }`
	`    in -= dim;`
	`}`

This is possible because *(dim-1-j) * dim + i = dim * dim - dim - j * dim + i*. Anything not attached to *i* and *j* can be brought fully out of the loops, and any terms with just *j* can be put in the outer loop (but, can be simplified using strength reduction such that we subtract a *dim* at the end of each iteration of the outer loop rather than using the more costly multiplication).

This change gave the below performance results (average speedup from the original of 12%). This is because we calculate the more computationally expensive parts only once outside of the loops and just perform basic subtraction in the outer loop.

Dimension	naive_rotate	my_rotate	Speedup
512	1068	1003	106.48%
1024	7449	6711	111.00%
2048	63086	50613	124.64%
4096	436503	407986	106.99%
		Average:	112.28%

Improving memory access/locality

There are four main ways in which we can improve memory access/locality: merging arrays, loop interchange, loop fusion, and blocking. Merging arrays, loop interchange, and loop fusion is not possible for this function (we can't have a single array, changing the nesting of loops doesn't change the fact that we need to use both row-major and column-major order, and there aren't loops to fuse together), but blocking can be used to improve performance since we cannot avoid accessing data in column-major order. Since the image dimensions are a multiple of 32, we can use blocks of 32 to improve the locality of our memory accesses. In each block, our accesses to dst are in order such that after we access dst[0] we access dst[1]; for our access to src, we're accessing src[0], src[32], src[64], … and then src[1], src[33], src[65], … instead of something like src[0], src[512], src[1024], …. This means we have a higher cache hit rate because it's more likely the elements we're accessing are in the cache.

The modified code, taking advantage of (similar) improvements we performed in previous sections, now looks like:

Before:

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

After:

```
int dim2, k;
dim2 = dim * dim;
dst += dim2 - dim;
for (j = 0; j < dim; j += 32) {
	for (i = 0; i < dim; i++) {
		for (k = 0; k < 31;
k++) {
			dst[k] = *src;
			src += dim;
		}
		dst[k] = *src;
		dst -= dim;
		src -= dim*31 - 1;
	}
	dst += 32 + dim2;
	src += dim * 31;
}
```

We can see that this change made a more significant change to the performance of the my_rotate() function, with an average improvement of 101%::

Dimension	naive_rotate	my_rotate	Speedup
512	1016	691	147.03%
1024	7827	4750	164.78%
2048	63091	35682	176.81%
4096	585942	185253	316.29%
		Average:	201.23%

With a block size of 16, though, the performance is further improved (to a ~125% increase), which indicates that we are increasing memory locality and reducing cache misses over the block size of 32 (likely due to the size of the cache):

Dimension	naive_rotate	my_rotate	Speedup
512	1020	753	135.46%
1024	8149	3280	248.45%
2048	54719	45690	119.76%
4096	627518	158877	394.97%
		Average:	224.66%

my_smooth():

Function Inlining

There are many opportunities to inline functions with the smoothing function. We can inline the RIDX() macro, the avg() function, and the functions that the avg() function calls (initialize_pixel_sum(), maximum(), minimum(), accumulate_sum(), and assign_sum_to_pixel()) which will improve performance by avoiding the overhead that comes from functions (activation records, return instructions, etc.). The code now looks like this:

Before:

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

After:

```
int ii, jj;
pixel_sum sum;
pixel current_pixel;
for (j = 0; j < dim; j++) {
    for (i = 0; i < dim; i++) {
        sum.red = sum.green = sum.blue = 0;
        sum.num = 0;
        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++) {
                sum.red += (int)src[(ii * dim + jj)].red;
                sum.green += (int)src[(ii * dim + jj)].green;
                sum.blue += (int)src[(ii * dim + jj)].blue;
                sum.num++;
            }

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

The performance improvements from just inlining all of the functions is significant, with an average speedup of nearly 56%:

Dimension	naive_smooth	my_smooth	Speedup
256	7320	4687	156.18%
512	32177	20004	160.85%
1024	155367	99789	155.70%
2048	820113	548003	149.65%
		Average:	155.59%

Dataflow Analysis and Optimizations, Code Motion

Using the code from the previous optimization as a starting point, we can perform some dataflow analysis and do constant folding and eliminate common subexpressions:

- We notice that the bound for the *ii* for loop is calculated each time the loop iterates; we can improve performance by calculating it once before the loop and storing the value in a temp variable so that it doesn't need to be calculated each time *ii* is incremented. We can do the same for the bound for the for loop that uses *jj* as an iterator.
- We also notice that *ii * dim + jj* is calculated 3 times each time the *jj* loop iterates; we can calculate *ii * dim* outside of the *jj* loop since it doesn't rely on *jj* and then just add *jj* to that value inside of the *jj* loop.
- We also see that *i + 1* and *j + 1* are calculated more than once per loop of their respective loops, so we store those calculated values in temp variables to avoid re-calculating them and improve performance. The same can be said for *i - 1* and *j - 1*.
- *dim - 1* is also calculated many times, so we'll take that out of all of the for loops entirely since it does not depend on any iterators and only calculate it once per function call.

After:

```c
int ii, jj, iidim, iidimjj, iiEnd, jjEnd, dimMinusOne, iPlusOne, jPlusOne, iMinusOne, jMinusOne;
pixel_sum sum;
pixel current_pixel;
dimMinusOne = dim - 1;

for (j = 0; j < dim; j++) {
    jPlusOne = j + 1;
    jMinusOne = j - 1;
    for (i = 0; i < dim; i++) {
        iPlusOne = i + 1;
        iMinusOne = i - 1;
        sum.red = sum.green = sum.blue = 0;
        sum.num = 0;
        iiEnd = (iPlusOne < dimMinusOne ? iPlusOne : dimMinusOne);
        for (ii = (iMinusOne > 0 ? iMinusOne : 0); ii <= iiEnd; ii++) {
            iidim = ii * dim;
            jjEnd = (jPlusOne < dimMinusOne ? jPlusOne : dimMinusOne);
            for (jj = (jMinusOne > 0 ? jMinusOne : 0); jj <= jjEnd; jj++) {
                iidimjj = iidim + jj;
                sum.red += (int)src[iidimjj].red;
                sum.green += (int)src[iidimjj].green;
                sum.blue += (int)src[iidimjj].blue;
                sum.num++;
            }
        }

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

These changes net us an average speedup from the original function of ~74%:

Dimension	naive_smooth	my_smooth	Speedup
256	7368	3758	196.06%
512	31024	18107	171.34%
1024	158816	88126	180.21%
2048	824952	549482	150.13%
		Average:	174.44%

Improving memory access/locality

It is not possible to merge arrays with this function since we don’t have two arrays to merge. It is also not possible to benefit from loop fusion since there are no loops that have the same looping with overlapping variables.

The best we can do with memory blocking is using row-major access for the dst pixels. We notice that the smooth function iterates through every pixel and smooths each one. However, the order of nesting of the for loops results in the image being smoothed column-by-column, rather than row-by-row, which is unnecessary. Since C uses row-major order, this is computationally heavy and can result in cache misses and more memory accesses. We can use loop interchange to build on our last improvements and improve locality and reduce cache misses. This also gives us the best form of memory blocking, since we're accessing in row-major order. (We also can now calculate *i* * *dim* outside of the *j* loop, making that calculation run significantly fewer times—from dim^2 times to just *dim* times.)

After:

```
int ii, jj, iDim, iiDim, iidimjj, iiEnd, jjEnd, dimMinusOne, iPlusOne, jPlusOne, iMinusOne, jMinusOne;
pixel_sum sum;
pixel current_pixel;
dimMinusOne = dim - 1;

for (i = 0; i < dim; i++) {
    iPlusOne = i + 1;
    iMinusOne = i - 1;
    iDim = i * dim;
    for (j = 0; j < dim; j++) {
        jPlusOne = j + 1;
        jMinusOne = j - 1;
        sum.red = sum.green = sum.blue = 0;
        sum.num = 0;
        iiEnd = (iPlusOne < dimMinusOne ? iPlusOne : dimMinusOne);
        for (ii = (iMinusOne > 0 ? iMinusOne : 0); ii <= iiEnd; ii++) {
            iiDim = ii * dim;
            jjEnd = (jPlusOne < dimMinusOne ? jPlusOne : dimMinusOne);
            for (jj = (jMinusOne > 0 ? jMinusOne : 0); jj <= jjEnd; jj++) {
                iidimjj = iiDim + jj;
                sum.red += (int)src[iidimjj].red;
                sum.green += (int)src[iidimjj].green;
                sum.blue += (int)src[iidimjj].blue;
                sum.num++;
            }
        }

        current_pixel.red = (unsigned short)(sum.red / sum.num);
        current_pixel.green = (unsigned short)(sum.green / sum.num);
        current_pixel.blue = (unsigned short)(sum.blue / sum.num);
        dst[(iDim + j)] = current_pixel;
    }
}
```

By swapping the order of the *i* and *j* loops, we now smooth row-by-row, which, as predicted, increases performance significantly (average of 135% from the original function):

Dimension	naive_smooth	my_smooth	Speedup
256	7415	4014	184.73%
512	32998	16032	205.83%

1024	153912	67257	228.84%
2048	803120	249253	322.21%
		Average:	235.40%

Strength Reduction

Finally, we can use strength reduction to further improve performance. We notice that *iidim* and *iidimjj* increase by *dim* and *1*, respectively, in each iteration of their respective loops. Rather than fully recalculating them each time, we can calculate their initial value outside of their loops and then increment them in their loops.

After:

```
int ii, jj, iDim, iiDim, iidimjj, iiEnd, jjEnd, dimMinusOne, iPlusOne, jPlusOne, iMinusOne, jMinusOne, iiStart,
jjStart;
pixel_sum sum;
pixel current_pixel;
dimMinusOne = dim - 1;

for (i = 0; i < dim; i++) {
    iPlusOne = i + 1;
    iMinusOne = i - 1;
    iDim = i * dim;
    for (j = 0; j < dim; j++) {
        jPlusOne = j + 1;
        jMinusOne = j - 1;
        sum.red = sum.green = sum.blue = 0;
        sum.num = 0;
        iiStart = (iMinusOne > 0 ? iMinusOne : 0);
        iiDim = iiStart * dim;
        iiEnd = (iPlusOne < dimMinusOne ? iPlusOne : dimMinusOne);
        for (ii = iiStart; ii <= iiEnd; ii++) {
            jjStart = (jMinusOne > 0 ? jMinusOne : 0);
            iidimjj = iiDim + jjStart;
            jjEnd = (jPlusOne < dimMinusOne ? jPlusOne : dimMinusOne);
            for (jj = jjStart; jj <= jjEnd; jj++) {
                sum.red += (int)src[iidimjj].red;
                sum.green += (int)src[iidimjj].green;
                sum.blue += (int)src[iidimjj].blue;
                sum.num++;
                iidimjj++;
            }
            iiDim += dim;
        }

        current_pixel.red = (unsigned short)(sum.red / sum.num);
        current_pixel.green = (unsigned short)(sum.green / sum.num);
        current_pixel.blue = (unsigned short)(sum.blue / sum.num);
        dst[(iDim + j)] = current_pixel;
    }
}
```

This resulted in a slight increase in performance from our previous optimization, bringing the overall performance gain to ~140%. This improvement is because addition is less costly than multiplication and just iteratively adding to something already calculated is also less costly than recalculating the whole value.

Dimension	naive_smooth	my_smooth	Speedup
256	7164	3830	187.05%
512	30933	15172	203.88%
1024	148973	59943	248.52%
2048	761234	236379	322.04%
		Average:	240.37%

Lastly, we notice that the above loops require logic and branching to check if the current pixel is a corner (which only averages 4 pixels), an edge (which averages 6), or an inside pixel that averages 9 pixels. This results in more computations being done for each pixel. To avoid this, we can separate calculating averages for the corners, edges, and inside from each other. And, to improve memory locality and increase cache hits, we can do the sides in a clockwise direction (starting at the top left corner, then the top edge, then the top right corner, then right side, and so on). For the corners, we also notice that we divide by 4 to calculate the average, so we can do a right shift of 2 instead since multiplying is more costly than a bit-shift. We can also use dataflow analysis to reduce common subexpressions and reduce the number of calculations. Putting all of these together, the code now is much longer but significantly more efficient.

After:

```
   int dim2, dimMinusOne, dimPlusOne, iMinusOne, iPlusOne, k, l, m, n, o, p, q, r, s;
   dim2 = dim * dim;
   dimMinusOne = dim - 1;
   dimPlusOne = dim + 1;
   pixel current_pixel;

   // top left corner
   current_pixel.red = (src[0].red + src[1].red + src[dim].red + src[dimPlusOne].red)
>> 2;
   current_pixel.green = (src[0].green + src[1].green + src[dim].green +
src[dimPlusOne].green) >> 2;
   current_pixel.blue = (src[0].blue + src[1].blue + src[dim].blue +
src[dimPlusOne].blue) >> 2;
   dst[0] = current_pixel;

   // top side
   iMinusOne = -1;
   iPlusOne = 1;
   l = dim;
   k = dim - 1;
```

```
    j = dim + 1;
    for (i = 1; i < dimMinusOne; i++) {
        iMinusOne++;
        iPlusOne++;
        l++;
        k++;
        j++;
        current_pixel.red = (src[iMinusOne].red + src[i].red + src[iPlusOne].red +
src[k].red + src[l].red + src[j].red) / 6;
        current_pixel.green = (src[iMinusOne].green + src[i].green +
src[iPlusOne].green + src[k].green + src[l].green + src[j].green) / 6;
        current_pixel.blue = (src[iMinusOne].blue + src[i].blue + src[iPlusOne].blue +
src[k].blue + src[l].blue + src[j].blue) / 6;
        dst[i] = current_pixel;
    }

    // top right corner
    j = dimMinusOne + dim;
    k = j - 1;
    iMinusOne++;
    current_pixel.red = (src[iMinusOne].red + src[dimMinusOne].red + src[k].red +
src[j].red) >> 2;
    current_pixel.green = (src[iMinusOne].green + src[dimMinusOne].green + src[k].green
+ src[j].green) >> 2;
    current_pixel.blue = (src[iMinusOne].blue + src[dimMinusOne].blue + src[k].blue +
src[j].blue) >> 2;
    dst[dimMinusOne] = current_pixel;

    // right side
    k = dim2 - 1;
    n = j - 1;
    l = n - dim;
    m = j - dim;
    p = j + dim;
    o = p - 1;
    for (i = j; i < k; i += dim) {
        current_pixel.red = (src[l].red + src[m].red + src[n].red + src[i].red +
src[o].red + src[p].red) / 6;
        current_pixel.green = (src[l].green + src[m].green + src[n].green +
src[i].green + src[o].green + src[p].green) / 6;
        current_pixel.blue = (src[l].blue + src[m].blue + src[n].blue + src[i].blue +
src[o].blue + src[p].blue) / 6;
```

```
        dst[i] = current_pixel;
        l += dim;
        n += dim;
        m += dim;
        o += dim;
        p += dim;
    }

    // bottom right corner
    l = k - 1;
    m = k - dim;
    n = m - 1;
    current_pixel.red = (src[n].red + src[m].red + src[l].red + src[k].red) >> 2;
    current_pixel.green = (src[n].green + src[m].green + src[l].green + src[k].green)
>> 2;
    current_pixel.blue = (src[n].blue + src[m].blue + src[l].blue + src[k].blue) >> 2;
    dst[k] = current_pixel;

    // bottom side, from right
    j = m + 1;
    m = l - dim;
    k = m - 1;
    n = m + 1;
    o = l - 1;
    p = l + 1;
    for (i = l; i > j; i--) {
        current_pixel.red = (src[k].red + src[m].red + src[n].red + src[o].red +
src[i].red + src[p].red) / 6;
        current_pixel.green = (src[k].green + src[m].green + src[n].green +
src[o].green + src[i].green + src[p].green) / 6;
        current_pixel.blue = (src[k].blue + src[m].blue + src[n].blue + src[o].blue +
src[i].blue + src[p].blue) / 6;
        dst[i] = current_pixel;
        m--;
        k--;
        n--;
        o--;
        p--;
    }

    // bottom left corner
    j = i - dim;
```

```
    k = j + 1;
    l = i + 1;
    current_pixel.red = (src[j].red + src[k].red + src[i].red + src[l].red) >> 2;
    current_pixel.green = (src[j].green + src[k].green + src[i].green + src[l].green)
>> 2;
    current_pixel.blue = (src[j].blue + src[k].blue + src[i].blue + src[l].blue) >> 2;
    dst[i] = current_pixel;

    // left side, from bottom
    k = j - dim;
    l = k + 1;
    m = j + 1;
    n = j + dim;
    o = n + 1;
    for (i = j; i > 0; i -= dim) {
        current_pixel.red = (src[k].red + src[l].red + src[i].red + src[m].red +
src[n].red + src[o].red) / 6;
        current_pixel.green = (src[k].green + src[l].green + src[i].green +
src[m].green + src[n].green + src[o].green) / 6;
        current_pixel.blue = (src[k].blue + src[l].blue + src[i].blue + src[m].blue +
src[n].blue + src[o].blue) / 6;
        dst[i] = current_pixel;
        k -= dim;
        l -= dim;
        m -= dim;
        n -= dim;
        o -= dim;
    }

    // inside
    k = dim;
    l = k - 1;
    m = k + 1;
    n = l - dim;
    o = k - dim;
    p = o + 1;
    q = l + dim;
    r = q + 1;
    s = r + 1;
    for (i = 1; i < dimMinusOne; i++) {
        for (j = 1; j < dimMinusOne; j++) {
            k++;
```

```
            l++;
            m++;
            n++;
            o++;
            p++;
            q++;
            r++;
            s++;
            current_pixel.red = (src[l].red + src[k].red + src[m].red + src[n].red +
src[o].red + src[p].red + src[q].red + src[r].red + src[s].red) / 9;
            current_pixel.green = (src[l].green + src[k].green + src[m].green +
src[n].green + src[o].green + src[p].green + src[q].green + src[r].green +
src[s].green) / 9;
            current_pixel.blue = (src[l].blue + src[k].blue + src[m].blue + src[n].blue
+ src[o].blue + src[p].blue + src[q].blue + src[r].blue + src[s].blue) / 9;

            dst[k] = current_pixel;
        }
        k += 2;
        l += 2;
        m += 2;
        n += 2;
        o += 2;
        p += 2;
        q += 2;
        r += 2;
        s += 2;
    }
```

Smooth now sees a performance improvement of over 350%, with the performance increasing as the size of the image increases:

Dimension	naive_smooth	my_smooth	Speedup
256	6799	1940	350.46%
512	30430	7879	386.22%
1024	155057	34212	453.22%
2048	834334	134984	618.10%
		Average:	452.00%