Nick Benevento

Computer Architecture CS 2461

December 11, 2018

Project 6 Report

Rotate Function Optimizations:

The first thing I noticed when working on optimizing this function was that it called a different function in the inner for loop (RIDX). I started by **inlining** this function, so that the program did not have to put excess records and variables on the stack and didn't have to jump to different lines of code. Simply inlining the function exactly where it was called reduced the run time by about 2.8%.

Next, I saw that part of the computation that was done in the inner for loop was based on things that did not change within the for loop. This optimization method is the **code motion** from the lecture notes. To be specific, the line of code was (dim-1-j)*dim + i. Expanding this out, we get dim^2 - dim - dim*j +i. Now, since dim is a set number that is passed in, I put the first part of it, dim^2 - dim, outside of the for loops, so it was only called once. I replaced the dim*j with a variable that adds the value of dim to it every loop of j (instead of using the multiplication operator, which takes more time - **strength reduction**). However, doing this requires new variables to be declared, and the amount of computations isn't reduced by that much, so it only resulted in a 1.1% increase in efficiency by itself. Total, the increase in efficiency was 3.9%.

The next and most successful optimization (I thought) that I tried was accessing memory that was closer together. The original structure of the for loops was in column-major order, so I switched the two for loops to make it row-major order, and on my local machine, this resulted in

a 40% increase in efficiency. However, once I tested this on the seas shell, it caused a slow down of 40%. This may be due to differences in the compiler from my local machine to the seas shell, so I did not use this optimization for the rotate method.

After these changes, I realized something major must have been missing, because the code was not getting optimized very much by these small changes. So thinking back to what Dr. Narahari said in class, I started to try moving the pixels in blocks instead of rows/columns, so that way the memory is accessed closer together. To do this, I reverted back to the RIDX function because it was easier to change the i and j variables to (i) (j+1), (i+1) (j+1), etc. After some drawing and trial and error, I was able to rotate the picture in blocks of 4x4. This alone made a MAJOR improvement for the efficiency of the program. In total, the increase in efficiency was around 61.7%. I also moved one of the static computations, dim-1, outside of the loops and set it to a variable (**elimination of a common subexpression**). It was a small change, but increased the efficiency to bring the total up to 63%. Now, instead of blocking by 4, it is possible to block by 8, or 16, or 32, and these changes may bring the efficiency up even more. However, I am tired, and I believe the margin of efficiency that I would get is not worth writing that much more code, or figuring out how to do it with more for loops. So, I have stuck with the blocking by 4.

Smooth Function Optimizations:

Inlining: The first changes I made were inlining the ridx function as well as the average method. After that, I continued to inline some of the more complex functions that the average function called (methods that had a lot of parameters). After doing this and changing the order of some function definitions around (so that they weren't re-declared in the for loop multiple

times), I got an increase in efficiency of about 8% (**code motion**). After coming back to this after trying some other methods, I decided to inline the minimum and maximum functions that were called, just to see what would happen. As it turns out, doing this resulted in an additional ~10% increase in efficiency. Since minimum and maximum were being called so many times within the for loops, this inlining cased a larger increase in efficiency.

Since I was getting success from this, I decided to inline all the functions that were called in smooth. After rearranging some variables and their declarations around, I was able to increase the efficiency by another ~15%.

The next, and most major change, was switching the order of the for loops so that the memory was accessed in row-major order. After trying it on the rotate function and getting unsuccessful results, I was not hopeful, but trying it out for the smooth function resulted in an increase in efficiency of about 41%. Combined with the other changes I made before, the increase in efficiency reached about 66%.

Dimension	naive_rotate	my_rotate	naive_rotate	my_rotate
512	961	883	2499706	2297485
1024	5934	4173	15432425	10852195
2048	42057	30036	109352195	78098109
4096	721733	268127	1876758646	697135146

Efficiency increase = ~63%

Dimension	naive_smooth	my_smooth	naive_smooth	my_smooth
512	6399	3588	16638945	9330384
1024	27471	14777	71426369	38422602
2048	123413	59063	320874652	153568410

4096	699192	238814	1817906486	620920260

Efficiency increase = ~66%