Ralph Paul

Bhagirath Narahari

Computer Architecture

16 December, 2021

Project 6 Report

In this project we were working with image processing and specific functions that deal with altering an image's appearance. These functions are the rotate operation as well as the smooth operation. In this case our *rotate* function rotates our image, whose dimensions are NxN, by 90 degrees, counter-clockwise, and our *smooth* function changes the blur of our image. With this assignment I was tasked with implementing optimization techniques within the given functions to make our program run faster. The performance was measured by comparing the number of (processor) cycles it takes to run as well as the execution time in milliseconds. With this report I will take you through my optimization process, explain the different techniques that worked and didn't, and how these techniques help in optimizing the functions.

First we will begin by talking about our *rotate* function. I tried a number of different techniques in this one. They include: code motion, function inlining, and cache blocking. The first technique that I tried was code motion. I did this by creating a variable that stores the function (dim-1). I noticed that this operation produced a constant number every time so I decided to store it in a variable outside of the nested loops and just use my variable inside the loop. This did help improve my performance a tiny bit in that it reduced the amount of times that the computation was performed especially since it was within a loop so there could have been a lot of unnecessary computations occurring. This of course didn't help as much as others because

it was a small mathematical operation so it didn't drastically change my performance. The next technique I decided to use was function inlining. In this case we had a macro of RIDX which took in 3 inputs and performed a mathematical operation. Instead of calling my RIDX macro at the call site I wrote out the operation wherever I saw RIDX being called. This eliminates bookkeeping instructions for better optimization. And since the macro does a pretty simple procedure, the procedure call overhead can increase the execution time of the application considerably. So using this technique did really help improve the performance of my rotate function. The final technique that I used for my optimization was cache blocking. This works by accessing our data through blocks of code instead of iterating through entire rows and columns to only access one piece of data at a time. Without using blocking over the course of the outer loop, so many different places in memory will be accessed that the cache block associated with our dst pixel will be gone from cache, so, when our inner loop is incremented to the next value, it would still be in the same cache block, and that block is no longer in cache and has to be loaded again. So using blocking improves our temporal locality significantly, and as a result our performance is also improved significantly. I would say this definitely improved my speed the most because it improved how quickly I was accessing data. I implemented this by adding in two more nested loops that loop through a block of a specified length and once it is done my outer nested loops go to the next block. Another thing that helped performance was changing the tile size of my blocking. I knew that all my dimensions were divisible by 32 so I tried a lot of different multiples and factors of 32 to see what would work best. And it seemed like using 16 tiles works the best. This could be because this is a more accurate factor of all of the dimensions.

Testing Rotate:						
	Time in milliseconds		Cycles used			
Dimension	naive_rotate	my_rotate	naive_rotate	my_rotate	time ratio	improvement
512	1024	937	2049066	1875062	1.09284952	9%
1024	7322	4442	14649805	8885561	1.648356596	64%
2048	71698	32630	143400981	65652437	2.197303095	119%
4096	666087	138254	1382884266	283137532	4.817849755	388%

Now looking at our *smooth* function I tried a number of different techniques in this one as well. They include: function inlining, and cache blocking, and loop interchanging. First I tried function inkling, I did most of this in my avg function that is called by my *smooth* function. Inside of the avg function there were a lot of other functions that were called, and I realized that they were not really needed because I could just write them. The *maximum* and *minimum* functions were both one line of code, so I knew having those function calls would only increase my execution time with the overhead bookkeeping instructions. And I did the same thing for the *initialize_pixel_sum* function as well. This indeed helped improve my performance. But one thing I noticed was when I tried using function inlining for the *assign_sum_to_pixel* function the performance was worse. I believe this could be because the size of my code was increased so that could have caused it to go slower in that more instructions are being carried out in the function. Next technique I used was cache blocking. As described in my *rotate* function I iterated through blocks of data at a time this keeps my temporal locality accessible because we are accessing data within close proximity so our cache does not have to reset, so this reduces our misses by a lot and ultimately betters our performance significantly. I also used 16 as my block size. The last technique that I implemented was loop interchanging. In the original function we were given, we are iterating in column major order, and from what I have learned through the lectures C stores 2d arrays in row major order. So knowing that I interchanged the loops to help improve the

performance. This is because with row major order consecutive elements of a row are contiguous in memory. So reading memory in contiguous locations is faster than jumping around among locations. As a result, if the array is stored in row-major order, then iterating through its elements sequentially in row-major order was shown to be faster than iterating through its elements in column-major order.

Testing Smooth:						
	Time in milliseconds		Cycles used			
Dimension	naive_smooth	my_smooth	naive_smooth	my_smooth	time ratio	improvement
256	5405	5251	10811879	10503149	1.029327747	3%
512	23369	21081	46739073	42163757	1.108533751	10%
1024	112723	85566	225449031	171931943	1.317380735	31%
2048	565665	340670	1132730404	681426443	1.660448528	66%