Timothy Lytkine George Washington University

CSCI 2461 Professor Leontie

Project 6: Code Optimization Report

rotate Function

Optimization techniques used:

- Loop interchange
- Blocking

Rationale behind choosing techniques and elaboration on performance improvement:

First, the loop interchange technique of compiler optimization was implemented. This was done by swapping variables i and j in the nested for loops. Since the image which the operation is being performed on is a 2D array of pixels, this means that by default C will store it in row major order. Initially, with variable j in the outer for loop and variable i in the inner for loop, the image is accessed using column major order. This is inefficient and results in more cache misses. By swapping the for loop variables, the image is now accessed in row major order which decreases the number of cache misses thereby increasing the number of cache hits. Furthermore, spatial locality is improved since adjacent elements in a row have nearby memory addresses due to being stored in row major order.

The blocking technique of compiler optimization was also implemented. In order to do so, a macro named rotateFunction(num) was defined. This function-like macro essentially takes the contents of the nested for loops in the naive_rotate function but adds the parameter of num to the value of i each time. The original contents of the nested for loop (still present in naive_rotate) were replaced with the rotateFunction being called from 0 to 15 (takes into consideration the

index starting at 0). In addition to this, the first for loop was modified to increment i by 16 rather than 1. The purpose of this was to implement the blocking compiler optimization technique. The pixels are accessed as blocks of 16. The description of the project stated that it can be assumed that the image dimension is a multiple of 32. Since 16 is a factor of 32, all the pixels of the image are accessed and rotated. 32 was not used in the for loop since we have to account for potential cache misses that can occur when initially fetching the block of memory.

By accessing blocks of memory at a time in the order that they are stored (row major order), values are accessed from the cache rather than the main memory. This results in less cache misses and more cache hits. Hence, by using the optimization methods of loop interchange and blocking, the performance of the optimized code in the my_rotate function was more efficient than the code present in the naive_rotate. The improvement in performance can be seen in the table below where for dimension sizes of 1024 and up there is a 57.46%-75.63% decrease in time and cycles.

Table of Performance Results *Same percentage change for time and cycles

Dimension	Time (ms)		Cycles used		Percentage change *
	naive_rotate	my_rotate	naive_rotate	my_rotate	
512	3799	2799	10773130	7636041	26.32 %
1024	55871	23766	156348492	66505680	57.46 %
2048	290643	100762	812294521	281457012	65.33 %
4096	1302470	317429	3639938499	886911065	75.63 %

smooth Function

Optimization techniques used:

- Loop interchange
- Code motion
- Strength reduction
- Function inlining
- Dead code elimination

Rationale behind choosing techniques and elaboration on performance improvement:

First, the loop interchange technique of compiler optimization was implemented. This was done by swapping the i and j variables of the nested for loop. This change was made in order to access blocks of memory in the image consisting of a 2D array (matrix) of pixels in row major order rather than column major order. C stores 2D arrays in row major order so accessing the image the way that it is stored leads to increased spatial locality, more cache hits and less cache misses.

The rest of the changes to the my_smooth function were based heavily on utilizing the function inlining compiler optimization method by inling the code and methods from the avg method, inlining the functions contained within the avg method and plugging in the corresponding parameters. The avg method contained many other methods that were called within it such as initialize_pixel_sum, accumulate_sum and assign_sum_to_pixel. The code contained in all of the methods excluding accumulate_sum was added (inlined) into the my_smooth function with their corresponding parameters. It is worth noting that there were some

slight adjustments to account for whether a pointer or reference to a memory address was being used. Each time another function inlining optimization was performed, the performance was improved. An explanation for the improvement in performance is that since the functions are not called but rather the bodies of the functions are inlined into the my_smooth function, there are no subroutine calls. These function calls do not have to be popped onto the stack and spatial locality is improved.

After performing the function inlining optimization, the code motion technique of compiler optimization was utilized in order to reduce the frequency of computations performed. This was only applicable after inserting the newly inlined code into the my_smooth function. An example of code motion being used were when the integer variables x1, x2, y1, and y2 that were created as part of the function inline optimization were moved outside of the loops they were invariant in. Instead of computing these values each time in their initial respective inner for loops, they were computed in the outer for loop since they were invariant to the inner one. This resulted in fewer computations, improved temporal locality and less clock cycles. Hence, the function showed an improvement in performance.

The maximum and minimum methods are great examples of strength reduction and are worth mentioning despite them already being optimized initially. The operations used in these methods are less costly than ones used in other implementations of finding the maximum and minimum that I have observed. The variables previously mentioned (x1,x2,y1,and y2) are computed efficiently using the bodies of the maximum and minimum methods. Combining the strength reduction already present in these methods and inlining these functions lead to a notable improvement in spatial and temporal locality. The last optimization used was dead code

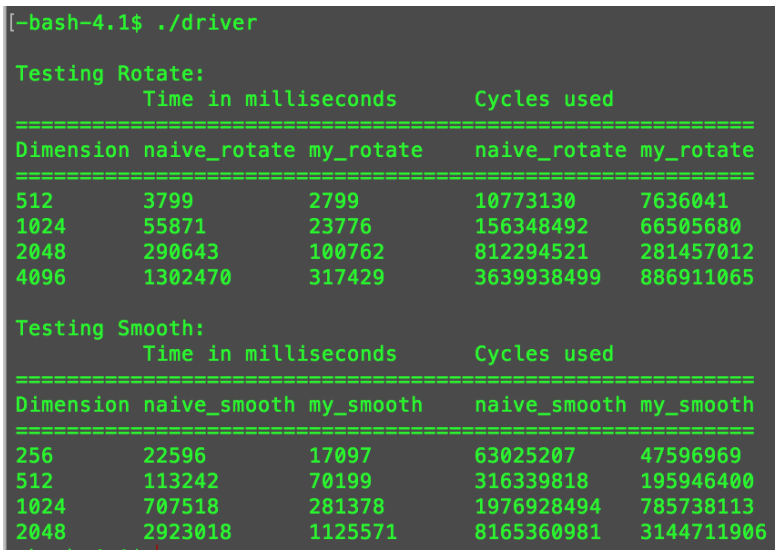
elimination. There was one line where this was done which was part of the body of the initialize_pixel_sum method: sum->red = sum->green = sum->blue = 0 = sum->num = 0. (sum->num was added to the same line and removed as a separate statement) While it was only one line of code, it is still worth noting since in other cases of optimizing code, a lot of lines of dead code can have a significant negative impact on the performance of a program.

The loop interchange, code motion and function inlining optimization methods lead to the most notable improvements in performance. As seen in the table below, for dimension sizes 1024 and up, there was more than a 60% decreases in cycles and time.

Table of Performance Results *Same percentage change for time and cycles

Dimension	Time (ms)		Cycles used		Percentage change *
	naive_smooth	my_smooth	naive_smooth	my_smooth	
256	22596	17097	63025207	47596969	24.34 %
512	113242	70199	316339818	195946400	38.01 %
1024	707518	281378	1976928494	7875738113	60.23 %
2048	2923018	1125571	8165360981	3144711906	61.49 %

Screenshot of performance on SEAS server:



```
[-bash-4.1$ ./driver

Testing Rotate:
          Time in milliseconds      Cycles used
============================================================
Dimension naive_rotate my_rotate    naive_rotate my_rotate
============================================================
512       3799         2799         10773130     7636041
1024      55871        23776        156348492    66505680
2048      290643       100762       812294521    281457012
4096      1302470      317429       3639938499   886911065

Testing Smooth:
          Time in milliseconds      Cycles used
============================================================
Dimension naive_smooth my_smooth    naive_smooth my_smooth
============================================================
256       22596        17097        63025207     47596969
512       113242       70199        316339818    195946400
1024      707518       281378       1976928494   785738113
2048      2923018      1125571      8165360981   3144711906
```