Stefen Pegels sgp62
ECE 5720
2/17/2021

HW 1 Writeup

1) Memory Strided Access Time

Cache Info From Index0 on sys/devices/system/cpu/cpu0/cache/index0:
Size of Cache: 32K
Number of Sets: 64
Associativity: 8-way
Line Length: 64

Here is an example output image from my .eps file. The data changes slightly with each compilation, but the trend remains the same. (Note that array sizes are superimposed plots on each other, with red being 2^{16} up to light blue being 2^{20})

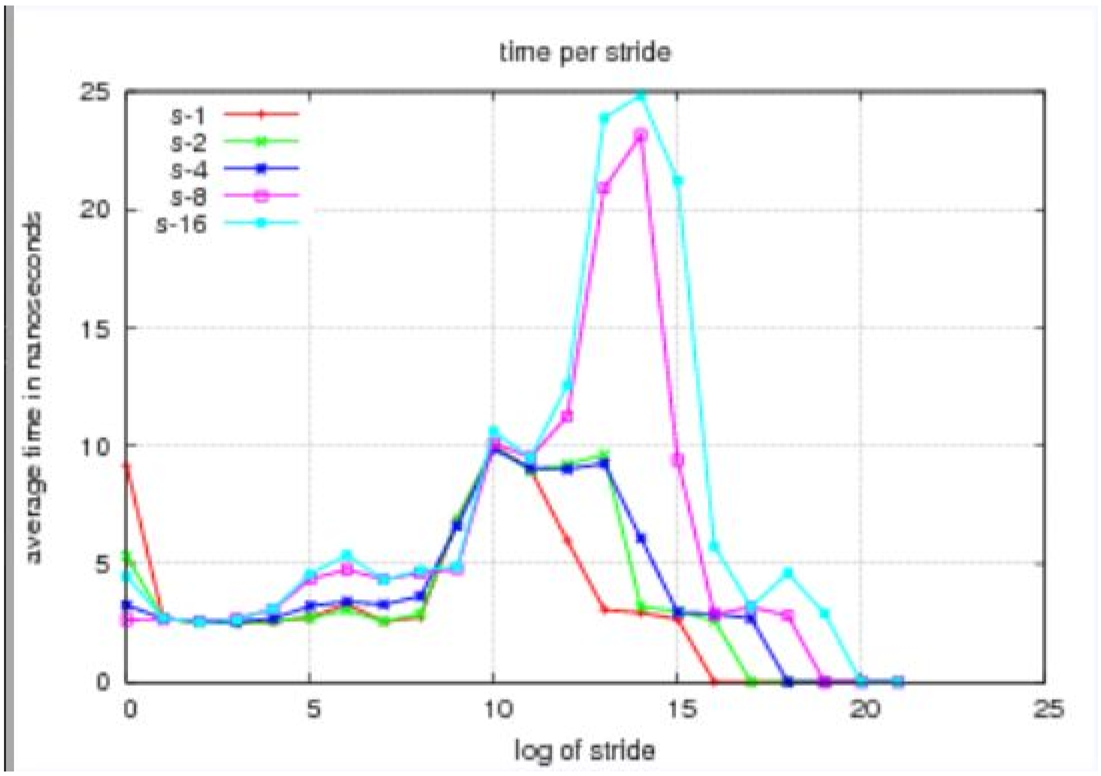

Figure 1: Graph of Log of Stride vs Nanosecond Time for Cache Operations

The graph supports the trend that the average time to access array elements (I was just doing +=1 to each element) decreases as stride length increases, before leveling off mostly at 2^4 stride length. Array size does not seem to be significant to the trend. Note that array sizes went from 2^{16} to 2^{20}.

The plot seems to correlate with the **L1 cache line length** of 64, because the lowest time occurs at stride length 16 (4 on the x axis), which makes sense because when our line length is 64, a cache line brings in

16 4-byte elements, which means the 16-length stride will bring in every array element in order, making repeated loop iterations generate more cache hits, driving down the average time.

The plot also correlates with the **L1 cache size** of 32K, because beyond 2^{15} array size, the entire array can no longer fit inside the cache, and we will create many cache misses as we jump around the array, especially at larger cache sizes. This explains the increase in access time as log of stride increases beyond 16, with larger array sizes suffering the greater performance impacts. These larger step sizes create many more cache misses, which shows us that we do not have a direct correlation between step size and average time, because it depends more on our cache hit rate. However, as stride length approaches array size /2, the average time drops back down again, as there are so few elements to access that we can partially negate the increase in time from our cache misses.

The plot shows no direct correlation to the **L1 associativity** of 8, but we can assume that this helps explain the lower times at stride lengths of less than 2^9 which had many cache hits. As the stride length became larger, similar tag bits were encountered less often and the associativity wasn't able to be taken advantage of.

Works Cited
Source for POSIX include command to not throw errors in VSCode for given structures like CLOCK_MONOTONIC
https://stackoverflow.com/questions/40515557/compilation-error-on-clock-gettime-and-clock-monotonic/40515669
General Note on Code Submission: I included the .gp files along with my .c files due to slight edits that might make my graph structure slightly different than that of the solution (ex different array lengths). I also included the .csv and .eps files referenced in this report, to corroborate my analysis and for grader's reference.

2) **Matrix-Matrix Multiplication**

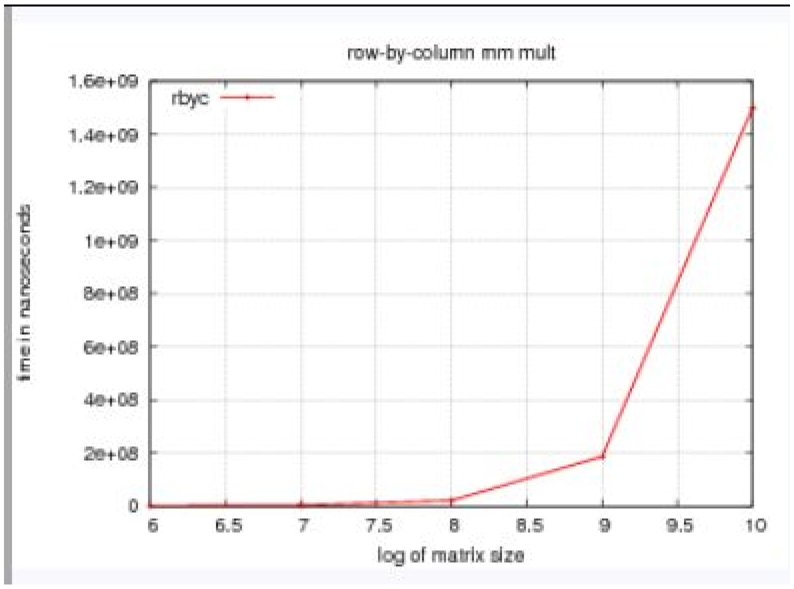

Figure 2: Log of Matrix Size vs Nanosecond Time for Row by Column Matrix Multiply (to N = 10)

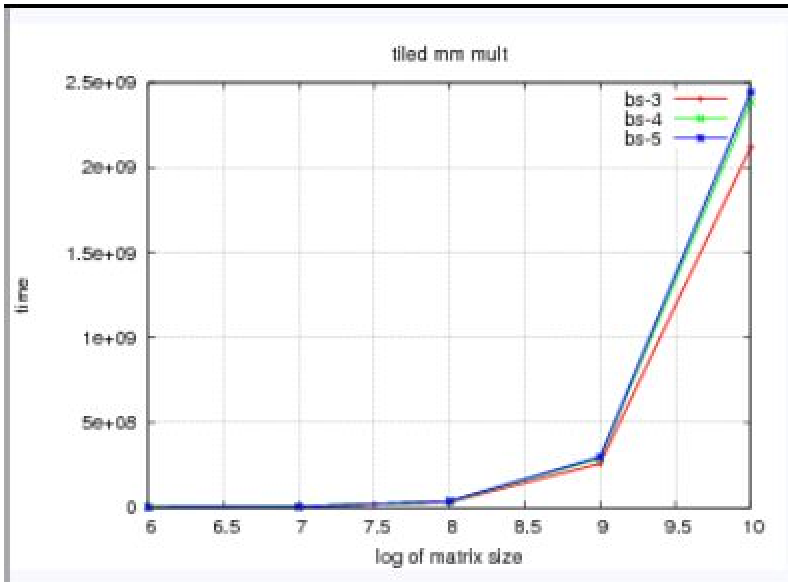

Figure 3: Log of Matrix Size vs Nanosecond Time for Block Matrix Multiply (to N = 10)

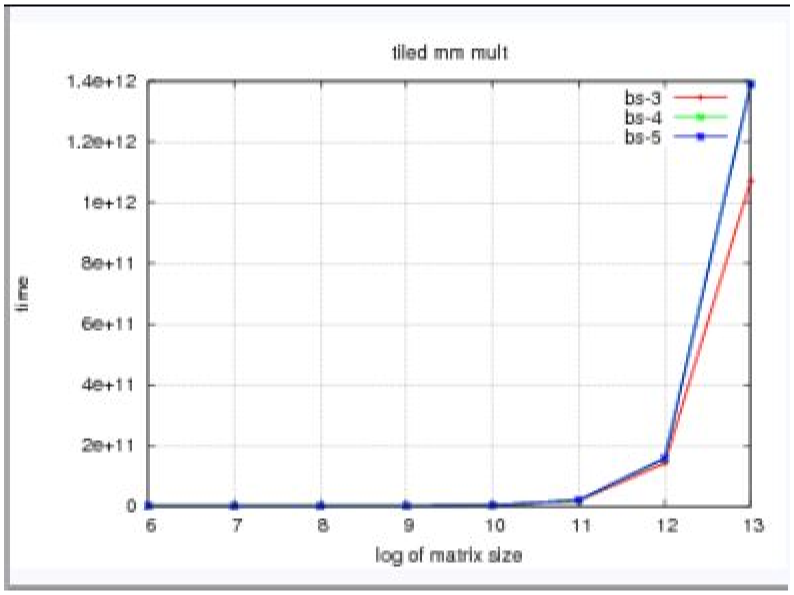

Figure 4: Log of Matrix Size vs Nanosecond Time for Block Matrix Multiply (to N = 13)

From the graphs, it appears that when matrix size is doubled, it follows *O(n^3)* conventions for each step, in some cases slightly larger or smaller gaps were recorded (ex row-by column went from 2.0e8 to 1.5e9 nanoseconds from N=9 to N=10, only a 7.5x increase)(**Figure 2**).

From comparing the graphs that go up to N=10 for both the row-by-column matrix and the block matrix operations, it appears that the row-by-column multiply was faster, with a comparison of **1.5e9 vs 2.1e9** nanoseconds for block size 3 at N = 10(**Figure 3**). This is a speedup of **40%**. Larger block sizes performed even worse (**Figure 4**). This seems to be the fault of bringing the blocks into the cache in an inefficient way. As block size increased, performance got worse, which explains why the rbyc approach was the most efficient, since the block size is just one in this case. I believe that block multiplication should actually perform better than row by column, but I think inefficiencies with my code design caused the slowdown(see lines **139-158 in sgp62_hw1_3.c**). My code has an inefficient way of populating each tile which I believe has dragged down the performance. With a few performance tweaks, block multiply should perform better with increasing block size, since we can bring multiple entries into the cache with a single read from memory, minimizing memory accesses and thus average memory access latency.

In doing analysis per single flop, we can take the absolute time at a specific array index and divide by the array size ^3. For rbyc at array size 2^6, we have 3.999e5 nanoseconds, corresponding to **3.999e5 / 2^{18} = 1.53 nanoseconds** per floating point operation. For other array indices of row by column we get the following:

Array Size	2^6	2^7	2^8	2^9	2^{10}
Total Time(ns)	3.999e5	3.212e6	2.092e7	1.867e8	1.500e9
Time per flop(ns)	1.525	1.531	1.247	1.391	1.397

Figure 5: Time for a Single Flop vs Total Execution Time for rbyc Multiplication

As the array dimensions changed, the time per single flop remained (mostly) constant. Note that the measurements for block multiplication mirror this phenomenon, but the figures are omitted for brevity. The constant behavior of the flop time demonstrates that as the program grows in size and execution time, the time for individual flops is unchanged and remains unaffected by the growing scope of the actively running program.