Vivek Sivakumar
UID 303652195

Lab 1 Report

Optimizing mmul1

My first step in optimizing mmul1 was parallelizing each of the 3 for loops.

	1024 x 1024	2048x2048	4096x4096
Inner k loop	Didn't finish	Didn't finish	N/A
Middle j loop	1.17 Gflop/s (2.3x)	0.89 GFlop/s (3.5X)	N/A
Outer i loop	1.16 GFlop/s (2.1x)	0.88 GFlop/s (3.7x)	N/A

Parallelizing the for loops had a minimal speedup effect, and the speed of the program actually decreased as I increased the matrix size. This told me that the parallelism benefit was not great enough to dominate the execution time equation, so I did not attempt these approaches with a 4096x4096 matrix. When I parallelized the innermost k for loop the performance actually got worse, to the point where it could not finish in a reasonable amount of time. I suspect this is because of the overhead of creating 32 threads for each matrix multiplication operation and then synchronizing them after the operation is complete. I initially hoped to replace C[i][j] += A[i][k] + B[k][j] with an accumulator variable via reduction, but since the innermost loop could not be efficiently parallelized this approach was not possible.

My next approach was to improve the spatial locality of my program, part of the problem with the initial algorithm is that it accesses columns of B for each matrix operation. In memory B is laid out in row major order, meaning the L1 cache is much more likely to have members of the same row in a block rather than members of the same column. The frequency of L1 cache misses from B column accesses was killing performance, to solve this I simply swapped the inner k loop with the middle j loop.

I chose to parallelize the outermost loop because it reduces thread overhead and increases the granularity on which each thread works.

	1024 x 1024	2048 x 2048	4096 x 4096
Sequential	6.53 GFlops/s (12.0X)	6.12 GFlops/s (10.2X)	N/A
Parallel outer loop	30.52 GFlop/s (50.5x)	47.33 GFlop/s (69.4x)	43.28 GFlops/s

The loop swap has a massive effect on performance both in the sequential version and the parallelized version, suggesting spatial locality is the big performance killer in the original algorithm.

The parallelized performance on 2048 is almost 50% greater than at 1024 but the same gains aren't seen between 2048 and 4096. The jump in performance from 1024 to 2048 makes sense because the amount of parallelizable computation increases with matrix size so this starts to dominate the execution time equation. I'm not sure why at least some gains were seen at 4096, my only thought is that this was run at a more impacted time on the server which might have affected performance.

I didn't run sequential at 4096 because of the time it would take, and it was clearly not going to outperform the parallelized version.

Optimizing mmul2

The first step to optimizing blocked matrix multiplication was figuring out how to do blocked matrix multiplication. My sequential algorithm is conceptually similar to the original sequential algorithm. There are 3 outer loops moving at the block level and doing block multiplication, and 3 inner loops moving at the float granularity doing the actual multiplication of the matrix values. Because of the improved spatial locality of the blocked approach, a pure sequential algorithm produced between a 1 to 2x speedup.

<u>Blocked Multiplication Optimizations</u>

- 32 threads
- Block Size: 32 block size

	1024 x 1024	2048 x 2048	4096 x 4096
Sequential	0.73 GFlops/s (1.9x)	0.72 GFlops/s (1.7x)	N/A
Parallel Outer Loop	6.21 GFlops/s (10.1x)	9.24 GFlops/s (14.1x)	8.97 GFlops/s
Swap jj,kk loops	15.25 GFlop/s (31.9x)	32.20 GFlops/s (46.4x)	32.43 GFlops/s
Swap j,k loops	24.74 GFlops/s (46.8x)	29.07 GFlops/s (74.9x)	26.05 GFlops/s

My first optimization was to parallelize the outermost for loop (I used the results from mmul1 to determine that this was the best loop to parallelize). This parallelization gave me a pretty good speedup (up to 14x at 2048), this makes sense because different threads are able to work on different blocks without intruding on each other, making the program much faster. For my final 2 optimizations I attempted to increase spatial locality by employing the same strategies from mmul1, namely swapping the jj and kk loops to increase row accesses and lower L1 cache misses. I saw huge gains in performance with these optimizations, reaching 32.43 GFlop/s with swapping the innermost jj and kk loops at 4096.

Like in mmul1 the performance on 2048 is much better than the performance at 1024, but the same gains aren't seen between 2048 and 4096. The jump between 1024 and 2048 is most likely for similar reasons as in mmul1. The lack of increased performance at 4096 is interesting, it could mean that 2048 is right around the sweet spot where parallel gains are maximized and parallel overhead is minimized, and increasing size after that gains parallelization but increases overhead resulting in a net neutral effect.

Blocked Multiplication With Different Thread Counts

- Block size: 32
- jj and kk loops swapped
- parallel outermost for loop

N Threads	2048 x 2048
2	6.70 GFlops/s (9.5x)
4	9.93 GFlops/s (18.7x)
8	12.67 GFlops/s (18.2x)
16	20.73 GFlops/s (29.5x)
32	32.20 GFlops/s (46.4x)

The program seems to scale as expected with more threads. At lower thread counts there is sometimes a 50% increase in performance for just adding 2-4 threads. Whereas at higher thread counts the number of threads needs to be increased by 16 to get a similar percentage of speedup. With the increased overhead of doubling the thread count, as well as the decreasing work for each thread to do, the diminishing returns for increasing thread count makes sense.

Blocked Multiplication With Varying Block Size

Before considering which block size to use I had to think about what makes a block size more efficient than another. I came up with two possibilities: the first is that one block of A and one block of B fit perfectly in the L1 cache so that multiplication of 2 blocks has no L1 misses. The second is that there are as many blocks as processors so that each thread operates on a block and they all theoretically finish together.

To fit a block of A and a block of B in the cache I used the following info:

- x86 floats are 4 bytes
- The L1 cache is 32Kb or 32768 bytes

Block size = sqrt(L1 Cache / (float size * blocks in cache))
Block size = sqrt (32768 / (4 * 2)) = 64

To have as many blocks as processors

Matrix size / n processors = 2048 / 32 = 64

These numbers ended up being the same! So 64 ended up being my target block size, I ran an experiment with ½ times and 2 times the block size just to see the results.

Results

- 32 threads
- jj and kk loops swapped
- parallel outermost for loop

	2048 x 2048
16	22.17 GFlops/s (37.6x)
32	32.20 GFlops/s (46.4x)
64	29.79 GFlops/s (42.7x)
128	24.06 GFlops/s (34.4x)

Surprisingly 32 ended up performing the best! I think this is because 32 allows a block of A, B, and C to fit in the L1 cache at the same time. Blocks of 64 were still giving me some read misses while blocks of 32 prevent those.

Discussion

I was surprised that most of the performance gains for matrix multiplication did not come from parallelizing the for loop, but instead from improving the spatial locality of the program by reordering the for loops that made frequent column accesses. Looking at it know, this makes sense because much of the matrix multiplication algorithm is not parallelizable, there are basically only 2 possible parallelizations in the basic sequential algorithm and only a few more in the blocked algorithm. This problem shows the importance of getting the best sequential algorithm before trying to parallelize, in both standard and blocked multiplication it is possible to beat a parallel algorithm with a sequential algorithm if it harnesses better spatial locality.

One interesting trend I saw was the effect checking loop conditions had on the speed of my program, in order to ensure correctness for all cases in blocked matrix multiplication I had to make sure the multiplication of 2 blocks did not overrun the block size and did not overrun the matrix itself. This second check was necessary if the block size does not divide evenly into the matrix size. Just by taking this second condition out of the innermost for loop, I saw massive gains in performance (~10 GFlops/s more). If I was truly optimizing for speed I would remove these checks and only use block sizes that divide evenly into the matrix size.