ECE454 LAB 5 REPORT **DECEMBER 6th, 2016**

Taylan Gocmen 1000379949 Gligor Djogo 1000884206

Optimization overview:
- eliminate redundant read accesses to the array: instead we store values locally and reuse them across iterations (we call this 'sliding window')

- loop peeling: we calculate the last column element along with the first column element, which allows us to only cross the 'toroidal' board boundary once during an iteration of the innermost loop, thus we are not replacing the cache unnecessarily often

- loop reordering to traverse entire columns (*Column Major Order): innermost loop increments rows, outermost loop increments columns, every time a new cell is computed there are three sequential reads from the array at the next column down, which may help with prefetching of cache lines by giving the prefetcher a consistent pattern to follow

- loop invariant code motion and common subexpression elimination: both are used to pre-calculate any constant boundary values outside of loops, eliminating small calculations repeated millions of times makes a difference

- multithreading: we divide up work by giving each thread a block of columns from the inner loop so that the time to compute the innermost loop is cut down, while generational dependencies are preserved

- using barrier instead of fork-join; this lowers the thread overhead when moving on to the next generation, synchronization is required to avoid the dependency of reading cells which have not yet been updated by another thread and are in the wrong generation cycle - loop unrolling innermost loop; allows processor to optimize the order of instructions for us, optimizing use of registers & alu componenets, and allows greater optimization at the level of the individual cpu architecture, minimizing the condition checks and jumps makes a difference since the loop is so highly used (triple nested)

- reduction in strength, eliminating any expensive modulo or 'mod' function by using the ternary operator to check for index conditions and catch the edge cases on the board

It is evident that in the original code, the main bottleneck is the game function due to the highly nested loops which traverse the game board and cycle through generations. Upon profiling the code with gprof and gcov, it was evident that the any small reduction in the amount of work inside the loops would result in significant speedups.

We tackled this problem by two means of optimization, multithreading and computing different sections of the board in parallel, as well as optimizing the inner loops to eliminate redundant operations and make the array access pattern more favorable on the cache.

For parallelization, we chose the simplest strategy to divide up the work. Multi-threading was done by simply assigning a block of consecutive columns to each thread. Threads were synchronized via barrier at the end of every generation such that any dependencies were avoided between generations. There was no dependency inside the generation, since all threads read from one board and wrote to separate sections of a second board.

The simple parallelization strategy was selected because we estimated there would not be any particular area of the game board that would require excessive computation, thus threads should be pretty well balanced. We concluded this from the input game boards, which are generated randomly and not preset in small clusters as usual GoL boards are. Also, looking at the final boards, the cells tend to live in stable patterns evenly distributed across the board.

Loop optimizations are listed above. The first change that we made was to realize that not all 9 cells need to be read every time to perform a calculation, since we can reuse previous values from the last iteration. This 'sliding window' approach only requires three three array reads and one write to compute each cell.

We then moved the computation of the last cell in a column from inside the loop to before it. This allowed us to set up the vars for our 'sliding window' more effectively and only cross the board border once per column. Also, we could eliminate some common subexpressions and move simple index boundary computations outside of the inner loops. We replaced the board edge check to a ternary condition rather than the 'mod' function which saved us the use of an expensive modulo op. We tried to reduce the operations as much as possible.

Finally, we did loop unrolling in the innermost loop. In the end we went with unrolling by eight, which means that eight cells are computed in each iteration. This is a standard technique which we saw could be exploited in our code, since the cells are updated in the preset pattern and it is easy to unroll by any amount. Loop unrolling cuts down on loop overhead and make use of cpu speedup when performing calculations. As we see it, the local variables are most likely held in registers, thus speeding up the computations even more. Thus, we achieved our maximum speedup of about 3.9 seconds (rounded).

Game of Life - Performance Timing Data

ORIGINAL sequential gol
time gol 10000 generations over 1k grid:
1k avg = 110.89
512 avg = 25.83
128 avg = 1.62

SWAP LOOP ORDER
Swapping the loop order, s.t. j is outer loop, i is inner loop (cols then rows)
1k = ~100 (1.11)
512 = 25.12 (1.03)
128 = 1.58 (1.02)

SLIDING WINDOW v1
Avoid unecessary reads from inboard by only performing reads for the cells on the leading edge of the sliding 3x3 game box
1k = 78.98 (1.40)
512 = 18.18 (1.42)
128 = 0.903 (1.79)

SLIDING WINDOW v2
Less local variables, combine nw+sw into tbw, etc.
Tried loop unrolling to 3 stages, time jumped to ~12s
1k = 47.73 (2.32)
512 = 11.09 (2.33)
128 = 0.32 (5.06)

PARALLEL sliding window
Parallelize with NUM_THREADS, assign block or rows to each thread_worker
1k = 11.3 (9.81)
512 = 3.22 (8.02)
128 = 0.66 (2.45)

BARRIERS parll slidingWdw
Add barriers between generations instead of fork-join every generation
1k = 4.61 (24.05)
512 = 1.21 (21.35)
128 = 0.15 (10.8)

LOOP UNROLL sliding window
Loop unrolled the sliding window
1k = 4.04 (27.45)
512 = 1.09 (23.69)
128 = 0.12 (13.5)

LOOP UNROLL x8
Similar to above
1k = 3.93 (28.2)
512 = 1.03 (25.1)
128 = 0.13 (12.5)