Shmuel Newmark

Optimized Game of Life

19 June 2022

OVERVIEW

Design an algorithm to perform n generations of Conway's Game of Life that operates as quickly as possible

GOALS

1. Learn techniques on how to optimize algorithms to improve performance
2. Gain insights into how the JVM works on a low level and how to use the JNI in a practical way.

SPECIFICATIONS

The game will follow the classical game of life rules: if a cell has fewer than 2 live cells it will die from loneliness in the next generation. Exactly 2 and it will keep on living. Exactly 3 and it will keep on living or birth even if it was previously dead. Any more and it dies from overpopulation. Finally, the board is made toroid such that the leftmost cell is also adjacent to the rightmost cell.

Every technique must implement GameOfLifeAbstract abstract class which includes a constructor that accepts a 2d boolean array as the game board and a method *getNGeneration* that accepts an integer argument *n* and outputs a 2d array. The performance will be assessed based on the runtime of this method as n and the dimensions of the array increase.

Three starting positions will be used, one small: 96x96, one medium: 400x400, and one large: 1000x1000. Implementations are allowed to take advantage of the fact that all the boards are square and divisible by 16, but should not depend on anything else.

Speed tests are conducted on a 64-bit Windows 10 computer with an i7-6800K@3.40GHZ processor, and 32GB of RAM.

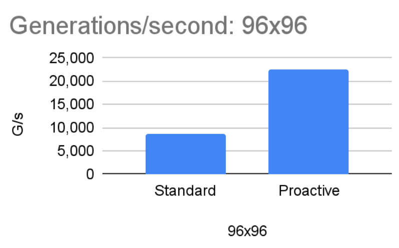
TECHNIQUES

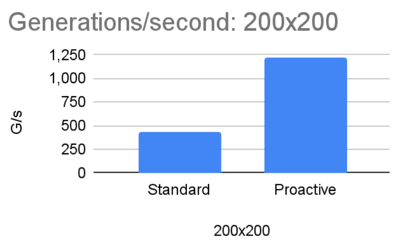
Basic Technique

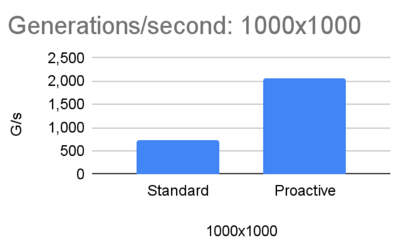
The most basic way to implement Conway's Game of Life is to iterate through every value in the array counting the adjacent cells' live states and setting the value in the cell. I accomplish this using a 2nd boolean array of the same dimensions as a buffer.

Optimization #1: *Proactive*

Another technique is to convert our buffer into an integer type instead of a boolean, then for each live cell, we increment the respective adjacent values in the buffer array. When done we set the bits in the boolean array based on the count in the buffer. This has the advantage of dodging the pricey 8-adjacent search for cells that aren't alive.



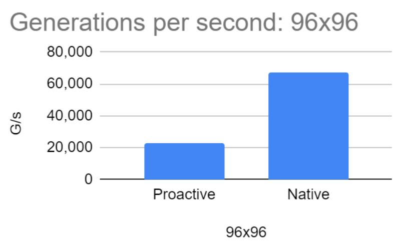


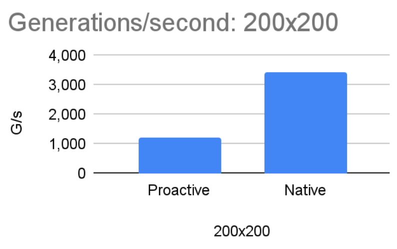


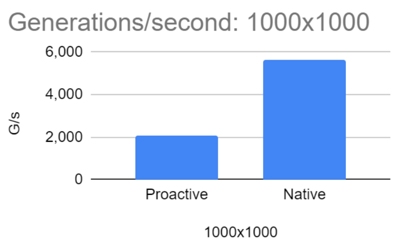
Average percent increase: 295%

Optimization #2: *Native*

We're going native! The implementation here is functionally identical to the proactive approach but programmed in C instead of Java. We pass the 2d boolean array into a native function call and then have the JVM 'pin' the array and perform raw memory accesses for a potential speed boost.







Average percent increase: 285%

This is much faster than the Java implementation. At first brush, this is a little surprising, the main inefficiency for array looping in Java is the need to do bounds checking on each access, but in theory, the JIT should optimize that away in tight loops. My best guess as to why that's not happening is due to the fact that in the 2d array my code will access values 1 above and below the x dimension without directly bounds checking. I know that to be safe because I ensure the array is a rectangle but I have no way to hint that fact to the JVM, forcing a check on each access.

Optimization #3: *Native Packed*

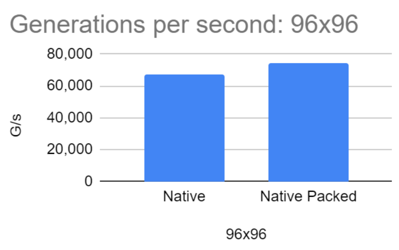
After playing around with how the Java arrays look in C I came to the realization that the boolean arrays actually occupy a byte each instead of what I presumed until now: a bit (goal 2 accomplished!). This is very space inefficient which got me thinking: what if I 'packed' 8 boolean values into a single byte value treating each bit as a boolean. Also, because our integral buffer array only needs to count up to 8 (each cell has 8 adjacent cells) we technically only need 8 bits allowing us to pack two integral counts into each byte. This has the advantage of being ~3 times more space efficient but the drawback of forcing me to perform significantly more bitwise operations on each iteration. Aside from being more space efficient, theoretically, the code can also be more performant due to better cache locality.

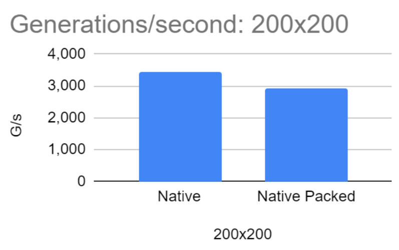
Standard:	0000000**0**	0000000**1**	0000000**0**	0000000**1**	0000000**1**	0000000**0**	0000000**1**	0000000**1**

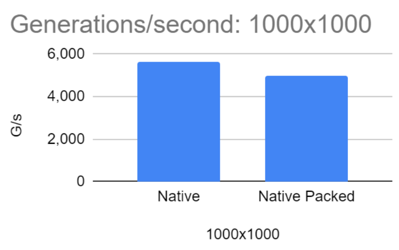
Despite the boolean being 8 bits (1 byte) only the last bit has any significance,

Packed:	**01011011**

Using some bit shifting we can pack 8 booleans in each byte.







Average percent increase: 86% (Decrease)

The fact that it's faster at all is still a little surprising. The code is longer and requires *significantly** more bit-wise operations but is still 5-10% faster on the smallest boards. The cheapness of bitwise operations on the main registers on x86-64 shows though, and avoiding a fair number of L1 cache misses cinches the advantage. Conclusion: doing lots of work isn't necessarily slower than sitting around waiting, even if it feels like it should be.

**(For reference, the inner loop of the standard version compiles down to 20 assembly instructions, the packed version requires 200! Almost all Shifts, Ands, Ors, and Masks).*

Optimization #4: *Native Dirty Bit*

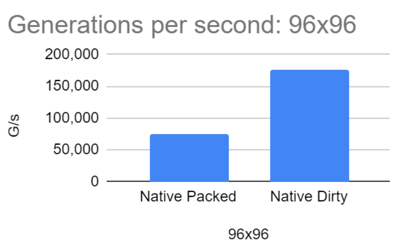
Building off my packing technique and proactive approach from earlier I tried to divine another strategy. My guiding intuition was to limit the number of times that the board needs updating which led me to a realization: on any given generation change, most of the cells are unchanged. For a given cell if all 8 of its neighbors are unchanged, it also can not change. Inverting that statement: “a cell only needs updating if on the previous generation one of its neighbors needed updating”. I, therefore, created a boolean array each representing a “line” of 8 cells. If a line’s values change, the corresponding value in the array is set to 1 as are the adjacent values. Then on the next iteration, only those lines whose corresponding bit is set in the dirty bit array are considered. The key idea here is that changes propagate, but also terminate. If a line goes an iteration without updating or having any adjacent updates its dirty bit is cleared and the line is static until an adjacent cell is changed again.

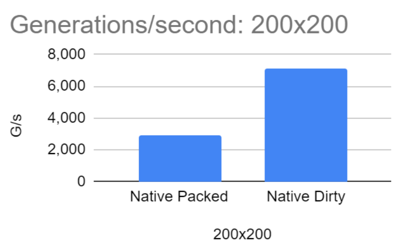
	🟩	🟥	🟩			🟩	🟥	🟩
Previous generation:	🟩	🟥 10001110	🟩		Current generation:	🟩	🟥 10001**0**10	🟩
	🟩	🟩	🟩			🟩	🟥	🟩

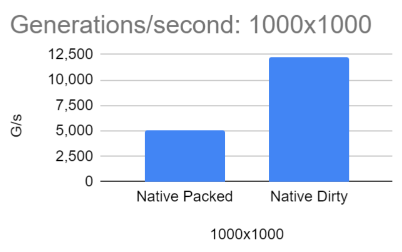
As you can see, the 4th value is changed between generations. This change can propagate, so all adjacent cells are marked as dirty. Because it’s a middle bit, that includes only those packed cells directly above and below.

	🟩	🟥	🟩			🟩	🟥	🟥
Previous generation:	🟩	🟥 10001110	🟩		Current generation:	🟩	🟥 1000111**1**	🟥
	🟩	🟩	🟩			🟩	🟥	🟥

In this case, a ‘corner’ bit has been changed, those changes can also propagate horizontally so we must mark the adjacent vertical cells dirty as well.





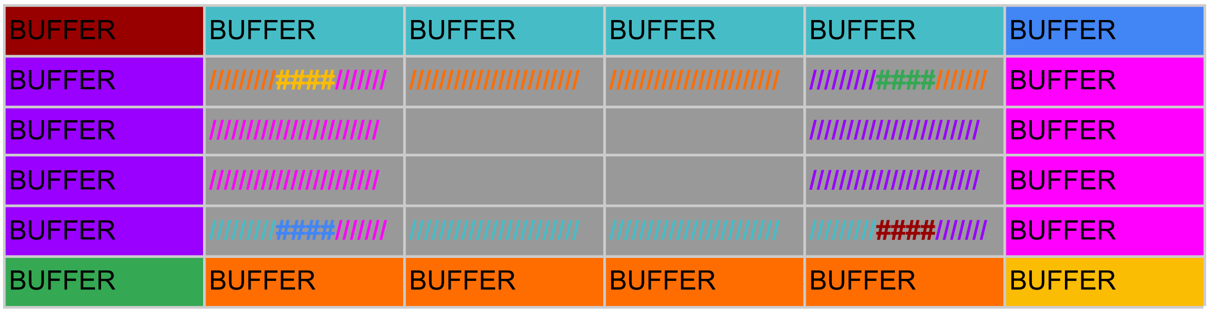


Average percent increase: 219%

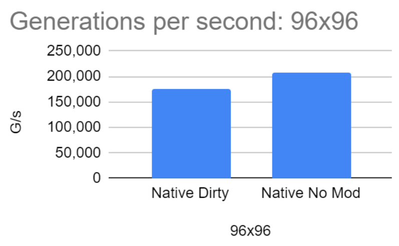
Additional considerations: my current implementation conceptualizes the 8 cells into a single 8x1 unit. A given change to that unit results in on average, 2.75 adjacent dirty bits set (5 for either of the 2 ends, 2 if in the middle 6). This can be improved by making the unit a 2x4 which would instead need 2 adjacent on average (3 for any of the 4 corners, 1 for the remaining 4 middles) but would necessitate a conceptual "splitting" of the char word, with each word occupying 2 x positions, i.e. the high bits having a different x value than the low bits. Manually crafting code for each bit instead of looping should smooth away the speed concerns but implementing it will still be painful because it can't be macro'd.

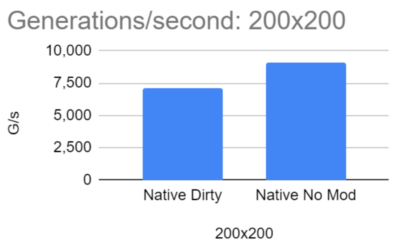
Optimization #5: *Native No Mod*

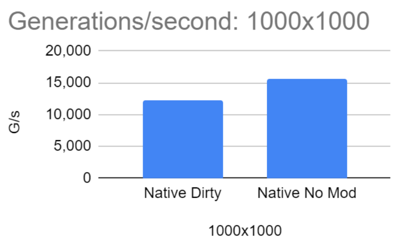
On the x86-64 mod is performed by doing an integral division operation. The rounded result is stored in the rax register while the remainder is stored in the rdx. Integral division however is a relatively expensive operation costing ~10 CPU cycles. Unfortunately, until now my code makes extensive use of mod to accomplish the toroid nature of my game board. To work around that, I extend the game board by 1 in all 4 directions to eliminate the mod operator, allowing code increment/decrement directly to the rectangular 'overflow buffer' at the end of the board. Then, at the end of each generation, I add the edges to their corresponding count.



Graph depicting the relation between the rectangular buffer and what gets added to where on the actual game board.







Average percent increase: 130%