Overview for Game of Life Implementation in Linux Kernel with Restrictive Memory

Bryce Palmer

October 2024

1 Introduction

This is a brief documentation for my implementation of Conway's Game of Life within a Linux kernel under restrictive memory constraints. I had previously developed a large scale Game of Life project[4] in Professor Bienz's Parallel Processing class which tested performance on both distributed memory and shared memory systems. In Matthew Fricke's HPC class at UNM, I was instructed to modify the kernel's output. After learning how to modify the VGA memory accessed by the kernel, I wanted to create a result more substantial than modifying text and colors.

2 Optimizations

The primary performance issues concerned memory usage and not speed. The performance behavior was unusual and difficult to track. It was observed that adding extra code lines and variables beyond what seemed to be arbitrary numbers caused parts of the program to run incorrectly. Such observations included patterns not showing in the screen, partial patterns appearing, incorrect state updates, VGA color mode crashing, and kernel mode crashing. When I use the word performance in this document, I am speaking of these unusual observations or lack thereof.

2.1 Compiler and Qemu Command Flags

The `-Os` flag for GCC compiler decreases code size; it uses all `-O2` optimizations, except those that increase code size. Removing this flag had immediate negative effects on performance.

The `-m` flag for `qemu-system-i386` specifies how much memory to allocate to

the emulator. I did not observe any positive benefits. I currently use `-m 500M`.

The `--mem` flag for `srun` specifies how much memory to allocate for a job. This also did not yield any positive benefits. I currently use `--mem 1000M`.

2.2 Code Reductions

A majority of the optimizations discussed in HPC class[3] were used, however most considerations were regarding memory use. A function that updated the state of the game which was called repeatedly inside the final *while(1)* loop was inlined to reduce the pressure on stack memory. Function names were shortened to single characters because function names are not abbreviated by the compiler and they are stored in read only data. Variables were reduced to the smallest size type compatible with their required operations. Variables were repurposed so as to require less variable management and allocation.

2.3 Pattern Encoding

Game of Life patterns, such as the Gosper Glider Gun[1], were copied from *conwaylife.com*. Each pattern represents alive cells with 'O' and dead cells with '.'. Rows often contain long stretches of dead cells, trailing up to a final alive cell, which significantly increases the size without contributing meaningful information.

Originally, I stored the Game of Life patterns in `char` arrays; however, this can be significantly reduced in size by using encoding schemes. I implemented two different encoding schemes, both of which performed better than storing the patterns without encoding. Oddly, the simplest but largest memory-requiring scheme returned the best performance.

The first encoding scheme I implemented was a direct character-to-2-bit scheme with a size on the order of the number of characters in the original pattern. There were 4 possible bit encodings: `0x0`, `0x1`, `0x2`, and `0x3`, which respectively correspond to `DEAD`, `ALIVE`, `NEWLINE`, and `EOF`. This is the encoding scheme used in the final product.

The second encoding scheme only stored locations of cells that are `ALIVE`. This scheme was designed to address the issue of the numerous `DEAD` cells in patterns, which could preferably be skipped. The number of bits to encode each `ALIVE` cell was `log(c)`, where `c = MAX(columns in pattern)`. Each bit encoding greater than 0 corresponds to the column in which the `ALIVE` cell is present. If the encoding is 0, it represents a `NEWLINE`. EOF encoding was not required as I recorded the total length of the encoding and passed the length to the decoding function.

I performed the initial encoding to each of the binary encodings manually. Then

I used a Python function to quickly translate binary into an array of hexadecimal values I could copy directly into the kernel file (page 4). I also wrote another Python function for the second encoding scheme that translated the column numbers into binary groupings of a given b bits (page 5).

2.4 Storing Game State in VGA

Instead of storing the game states in user defined arrays which requires additional memory, I utilized the existing VGA memory that is provided. Each cell on the displayable VGA screen corresponds to 16 bits in VGA memory.[2] The first 8 (0-7) bits correspond to the character displayed. Bits 8-11 dictate the foreground colour and bits 12-15 dictate the background colour. Setting the background bits to 0x0 sets the background black and 0x04 sets the background colour to red. Note when the background colour is red, this means bit 14 is set to 1. I used red to indicate a cell is alive and black to indicate the cell is dead. Hence I simplified updates to set bit 14 to 1 or 0.

Now recall that Game of Life usually requires storing 2 states of the game: the current state and the post state. The post state is determined by following the game rule update on the current state. After evaluation, the current state is typically swapped or set to the post state. In my case, I store the post state in the 1st bit of the foreground bits. Once the post state is completely evaluated, I update bit 14 for each cell which completes a state transition.

Python Code

```
def binary_to_hex(binary_str):
    # Pad the binary string to make sure its
   length is divisible by 8
    if len(binary_str) % 8 != 0:
        binary_str = binary_str.zfill(len(
   binary_str) + (8 - len(binary_str) % 8))

    # Convert each byte into hexadecimal and
   store as '0xXX'
    hex_list = []
    for i in range(0, len(binary_str), 8):
        byte = binary_str[i:i+8]  # Extract 8-bit
    chunk
        hex_value = hex(int(byte, 2))[2:].zfill
   (2)  # Convert to hex and ensure 2 digits
        hex_list.append(f"0x{hex_value.upper()}")
     # Format as C hex and use uppercase

    # Print the list of hex characters
    print(f"{{␣{',␣'.join(hex_list)}␣}};")

# Enter Binary String here!
binary_str = "01000000100000100001001110000000001"
    + "01110000101111000110111000000000000000"
binary_to_hex(binary_str)
```

Python Code

```
def numbers_to_binary_encode(numbers, bits):
    # Convert each number into binary. Keep
   leading zeroes. bits indicates #bits used per
   each number
    binary_string = ''.join(format(num, '0' + str
   (bits) + 'b') for num in numbers)
    return binary_string


# Enter numbers here
numbers = [2,0,1,0,1,0,2,3,4,0,0,5,6,0,5,7,0,6,7]

# Get the concatenated binary string
binary_string = numbers_to_bit_binary_string(
   numbers, 3)
print(binary_string)
```

References

[1] Conwaylife.com. *Gosper Glider Gun Cells*. Accessed: 2024-10-14. 2024. URL: https://conwaylife.com/patterns/gosperglidergun.cells.

[2] Brandon F. *Bran's Kernel Development Tutorial: Printing to screen*. Accessed: 2024-10-14. 2024. URL: http://www.osdever.net/bkerndev/Docs/printing.htm.

[3] Matthew Fricke. *CS491 2024 Lecture 11-Compiler Optimisaton*. Accessed: 2024-10-14. 2024. URL: https://fricke.co.uk/Teaching/CS491_2024/Lectures/Lecture_11-Compiler_Optimisation.pdf.

[4] Bryce Palmer et al. *Github Repository: Game of Life Parallel Computing Project*. Accessed: 2024-10-14. 2024. URL: https://github.com/kknowlson07/game_of_life/tree/master.