Sudoko Solver in RISC V

Itbaan Safwan
Department of Computer Science
Institute of Business Administration
Karachi, Pakistan
i.safwan.26197@khi.iba.edu.pk

Abdullah Ahmedani
Department of Computer Science
Institute of Business Administration
Karachi, Pakistan
a.ahmedani.25896@khi.iba.edu.pk

Huzaifa Ahmed
Department of Computer Science
Institute of Business Administration
Karachi, Pakistan
h.khan.26485@khi.iba.edu.pk

Zain Sharjeel
Department of Computer Science
Institute of Business Administration
Karachi, Pakistan
z.sharjeel.26922@khi.iba.edu.pk

***Abstract*— This project presents an in-depth exploration of optimizing a Sudoku solver by converting it from a C implementation using recursive backtracking to an assembly implementation enhanced with vectorization. The project focuses on the translation process, comparing the performance of the non-vectorized and vectorized assembly code. Results demonstrate the significant reduction in iteration count through vectorization. Challenges and insights gained during the optimization for larger grids are discussed below.**

I. The algorithm

The recursive backtracking algorithm is a popular approach to solving Sudoku puzzles. It works by iteratively filling in possible values for each cell, and recursively exploring the consequences of each choice. If a cell's value leads to an invalid board state, the algorithm backtracks and tries a different value. This process continues until the puzzle is solved or all possibilities have been exhausted. The algorithm's recursive nature allows it to efficiently explore the vast solution space of Sudoku puzzles.

II. Flow of the algorithm

A. *High level overveiw of Algorithm*

- Start with an empty board: Begin with a blank Sudoku board, where each cell is empty.
- Choose an empty cell: Select an empty cell to fill in.
- Try a possible value: Assign a possible value to the chosen cell.
- Recursively explore consequences: Call the algorithm recursively to fill in the rest of the board based on the chosen value.
- Backtrack if necessary: If the chosen value leads to an invalid board state, backtrack and try a different value.
- Repeat until solved: Continue filling in cells and backtracking until the puzzle is solved or all possibilities have been exhausted

B. *Important functions in the Algorithm*

The algorithm iteratively checks the board to find an empty element in the board. After finding the element, it checks the possible values between 1 till 64 (for a 64x64 board). To check if the value is valid in that empty cell it uses the rules of sudoku to check it. So, our first important function is the validation function and we named it *"isSafe"*. This function iteratively checks if the element is unique in its row, column and sub grid. These checks are straight forward in C code but how we converted it into assembly scaler and then vector form will be discussed below.

Second important function is the actual solving function. Although it is straightforward in C code as it is checking each element of the board and recursively calling itself after finding such element, but how the recursive call works in the assembly code will also be discussed below. Also, this function needs to iterate each element in board. We will discuss later how we reduced these iterations in our vectorized code alongside other improvements.

III. Conversion from C to assembly: without vectorization

The scaler conversion from C to assembly is very identical to original C. For the isSafe function, the assembly code iteratively checks each element in row, col and sub-grid (8x8 for 64x64 board) much like the C code. The complexity only arises when many registers are used so saving their values in stack plays a crucial role whenever the function is called.

Similarly, the solve function is also identical to its C counterpart. Here the complexity arises in the recursive calls where you have to save the return addresses in the stack so the functions can return back to their respective callee functions. Other then this function is also iteratively checking each element in the board and placing a possible value there and do a recursive call. If all elements are checked then that means the board is solved, and if any empty element failed on all possible inputs, it rejects that call and back tracks.

IV. Conversion from C to assembly: with vectorization

Before discussing how we convert our assembly code to vectorize version, we will first discuss vector instructions that we used in our vectorize code.

VSETVLI**:** The vsetvli instruction sets the vector length and element width configuration for subsequent vector operations. By specifying the vector length multiplier and element width through the immediate value, this instruction

ensures that the vector unit is appropriately configured for the following operations.

VLE8.V**:** The vle8.v instruction loads 8-bit elements from memory into a vector register. It efficiently transfers a series of 8-bit data elements from a specified memory address into the destination vector register, facilitating vectorized data processing.

VMSEQ.VX**:** The vmseq.vx instruction performs a vector-scalar equality comparison. It compares each element in a vector register with a scalar value, setting the corresponding elements in the destination mask register to 1 if they are equal, or 0 if they are not.

VFIRST.M**:** The vfirst.m instruction identifies the first set element in a mask register. It searches through the mask register and returns the index of the first element set to 1, or a special value if no elements are set, which helps in quickly finding active elements.

VPOPC.M**:** The vpopc.m instruction counts the number of set bits (1s) in a mask register. This count of active elements is useful for determining the number of elements that meet specific criteria, aiding in various vector operations.

We used these vector instructions to aid us in row checks, column checks, sub-grid checks and to find empty element in the board. Now we will discuss on how we incorporate these instructions in each of these checks.

A. *Row Check*

In Sudoko the element to be placed must be unique in its row. The basic approach that was used in C code and scaler assembly code was to iteratively check each element in the row and compare it with the value that you are going to place, if any element in the row matches then return invalid and if no element matches, the value to be putted is valid according to its row.

Instead of checking each element in the row, the vector instruction vle8.v allows us to put the whole row in a vector. We then used the vmseq.vx instruction to compare the value we want to put with the elements in the row. If any element was matched the instruction would set 1 in that position in a new vector register. Vpopc.m instruction is then used to count the number of 1s in the new vector register and it returns the counter in a new scaler register. Now all we need to see if the returned value in scaler register is non-zero or not, if it is zero then nothing matched meaning the value was unique in the row, if it is non-zero we return invalid.

. One thing to note is that the max length the vector can contain if each element is of size 8-bit is 32 elements (256 / 8). Problem comes when we solve for 64x64 board where each row will have 64 elements. To mitigate this we first took first 32 elements of the row, store it in vector and do the comparison. We then store the next 32 elements in the vector and repeat the comparison.

Fig.1 shows the process of row check visually on a 4x4 board.

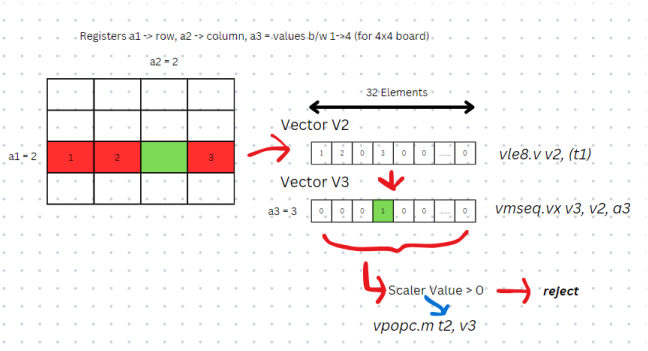

Fig.1. Example of row check operation on a 4x4 sudoko board. In our assembly code we have set a1 register to represent the row, a2 for column and a3 for value that we want to put in the blank place.

B. *Column Check*

In this check the new element must be unique in its column too. Intuitively it is very similar to row check and the vector code for this should be very similar to it, but the problem arises on how can we store sequential elements of column like we did for row. If we want each element of column sequentially, we will have to give an offset of 64 (for 64x64 board). We couldn't find any relevant vector instruction that would allow us to do so.

To address this problem, we transposed the grid clock-wise (see Fig.2) so that the columns become rows in new transposed grid. For this we had to make a separate copy of board in transposed form. Now we did the same row check in this new transposed board which served as the column check for the original board. The steps to check the valid value is identical to row check except in this case the elements of rows are loaded from transposed board instead of the original board.

Although this required storing duplicates of same board in memory and also extra computational steps to convert the board but it saved a lot of iterations in each of the recursive calls.

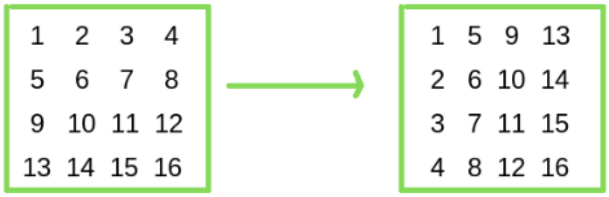

Fig.2. Transposition operation on the grid. Note the grid in the figure does not represen the sudoko board but transpostion effect still remains same for the sudoko board too.

C. *Sub-Grid Check*

In this check we first set the vector length to 8 elements (since each element is 8 bits) using the instruction vsetvli t0, a4, e8, m1. The variable is initialized to 8, representing the size of the sub-grid. The row and column starting points of the sub-grid are calculated using the instructions in Fig.3:

```
rem t2, a1, t1 // row%8
sub t2, a1, t2 // row - row%8
addi t3, zero, 0

rem t5, a2, t1 // col%8
sub t5, a2, t5 // col - col%8
```

Fig.3. Row and Column starting points calculations for sub-grid.

t3 is initialized to 0, representing the current row within the sub-grid. The base address for the sub-grid row is calculated by combining the row start t2, the size s2, and the column offset t5, then adding the base address s1 of the board. vle8.v v4, (t2) loads 8 elements (one row of the sub-grid) into the vector register v4. vmseq.vx v5, v4, a3 compares each element in v4 with a3 (the number being checked), setting the corresponding element in v5 to 1 if they match. vpopc.m t4, v5 counts the number of matches and stores the result in t4. If any match is found (bnez t4, notSafe), the function jumps to the notSafe label, indicating the number cannot be placed safely in the sub-grid. Increment and Loop: addi t3, t3, 1 increments the row index within the sub-grid. The loop continues until all 8 rows of the sub-grid are checked. Safe and Not Safe: If no duplicates are found within the sub-grid, the function proceeds to the safe label. If a duplicate is found, the function jumps to the notSafe label.

D. Empty Check

This is essentially the solving function but unlike the scaler version that iterates each element of the board to find an empty element, this function does that by putting each row in a vector and then look for empty elements.

Here is a run down of the process: The loop counter t0 is initialized to 0, representing the first row. The loop continues until all rows (s2, which is 64 for a 64x64 board) are processed. Within the loop, the base address of the current row is calculated by multiplying the row index (t0) by the board size (s2) and adding it to the starting address of the board (s1). The vle8.v instruction is used to load the elements of the current row into a vector register (v0).

This operation transfers 32 elements (8-bit each) from memory into the vector register, enabling parallel processing. The vmseq.vx instruction then compares each element in the vector register (v0) with zero. This comparison sets corresponding elements in a mask register (v1) to 1 if they are equal to zero, indicating empty cells. The vfirst.m instruction scans the mask register (v1) to find the index of the first set bit (1). This index represents the position of the first empty cell in the current row.

If an empty cell is found (the index returned by vfirst.m is non-negative), the function branches to the processEmpty label to handle the empty cell. If no empty cell is found in the first 32 elements of the row, the function checks the remaining elements. It increments the base address by 32 to access the next set of elements and repeats the vector load and comparison operations. If no empty cells are found in the entire row, the loop counter (t0) is incremented to process the next row. The loop repeats until all rows are checked.

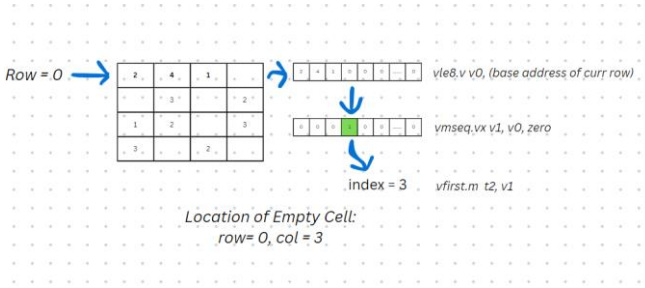

Fig.4. An example of empty check in 4x4 board for first row. All rows are similarly checked.

IMPROVEMENTS GAINED BY THE VECTOR CODE

The vectorized code greatly improves upon number of iterations. For example in row check normally we would have to do 64 iteration for each element, but in the vector all we had to do was use few instruction to copy entire row into a vector and then vmseq.vx instruction would parallelly compare all elements in the vector. This reduced our iterations from 64 to just 1 iteration.

For column check we pre-loaded a transposed version of the board alongside with the original board this saved us the extra computational steps to convert it to transposed version. Hence, we got similar efficiency gains like in row check by reducing iterations from 64 to 1 in each call of isSafe function.

For sub-grid checks we reduce our iterations 64 iterations (for a 8x8 sub-grid) to 8 iterations for 8 rows. Similarly in empty check we reduced the iterations from 4096 (for a 64x64 board) to 64 iterations for 64 rows in the board.

Mind that these reductions save a lot of steps as the recursive function is called a lot times when the board has many empty spaces. These reductions in iterations had a significant impact on the running time of our code as shown in Fig.5.

Fig.5. Comparision of running time and instructions excuted for scaler code and vectorized code for 64x64 sudoko board.

CLOSING COMMENTS

In conclusion, our project successfully demonstrated the significant performance improvements achieved by vectorizing a Sudoku solver originally implemented in C using recursive backtracking and then converting it to RISC-V assembly language. The introduction of vector instructions notably reduced the iteration count and, consequently, the execution time, particularly for larger grids like 64x64.

We faced and overcame several challenges, including the need to transpose the grid for column checks and the management of recursive function calls in assembly language. These optimizations and the use of vector instructions such as **vsetvli**, **vle8.v**, **vmseq.vx**, **vfirst.m**, and **vpopc.m** were pivotal in enhancing the solver's efficiency.

The improvements in the vectorized code highlight the potential of vectorization in solving computationally intensive problems and provide valuable insights for future optimizations in similar applications. The results, as depicted in our performance comparison, underscore the importance of leveraging advanced assembly instructions for optimizing algorithms. However, it is important to note that the code's performance significantly degrades when the number of empty spaces in the Sudoku board is high. The increased number of recursive calls and backtracking steps required to fill a large number of empty cells leads to longer execution times, making it impractical for boards with extensive empty spaces.

[1] https://inst.eecs.berkeley.edu/~cs152/sp20/handouts/sp20/riscv-v-spec.pdf

[2] https://github.com/annasshaikh/Sudoku-Veer-ISS-Log-to-Memory-Converter