Sudoku Solver 64x64

Ramail Khan
Bachelors in Computer Science
Institute of Business Administration
Karachi, Pakistan
r.khan.26924@khi.iba.edu.pk

Ikhlas Khan
Bachelors in Computer Science
Institute of Business Administration
Karachi, Pakistan
i.khan.27096@khi.iba.edu.pk

Bilal Adnan
Bachelors in Computer Science
Institute of Business Administration
Karachi, Pakistan
m.adnan.27151@khi.iba.edu.pk

Muhammad Musab Suhail
Bachelors in Computer Science
Institute of Business Administration
Karachi, Pakistan
m.suhail.26923@khi.iba.edu.pk

This article presents a detailed approach to solving a 64x64 Sudoku puzzle using RISC-V vector instructions to enhance computational efficiency. We explore the design and implementation of the SolveBoard function, which iteratively attempts to place numbers in empty cells by validating placements with the isSafe function. The isSafe function ensures compliance with Sudoku rules by checking for duplicate numbers in rows, columns, and subgrids. To overcome the inefficiencies inherent in scalar processing for such a large dataset, we introduce vectorization techniques, leveraging RISC-V's vector capabilities to parallelize checks and significantly reduce computation time. Our method demonstrates substantial performance improvements, showcasing the superiority of vectorized instructions over scalar approaches in handling large-scale Sudoku puzzles.

Keywords—vectorization, recursion, backtracking, functions

Introduction

Sudoku, a popular number puzzle, becomes significantly more complex when scaled to a 64x64 grid. Traditional solving methods, which utilize recursive backtracking, are inefficient for such large grids due to the extensive number of entries that must be processed. To address this, we introduce vectorization, leveraging RISC-V vector instructions to enable parallel processing and improve computational efficiency. The SolveBoard function iterates through each cell, attempting to place numbers and using the isSafe function to validate these placements according to Sudoku rules. Vectorization optimizes this process by handling multiple data elements simultaneously, significantly reducing computation time. Specifically, vector instructions are employed to perform parallel checks on rows, columns, and subgrids. Key to our approach is the vsetvli command, which sets vector lengths and adapts vector operations to the specific data size. Transitioning from scalar to vector code involves managing memory alignment and ensuring efficient parallel data processing. Despite challenges such as handling large datasets and debugging vector operations, our method demonstrates significant performance improvements, making it an effective solution for solving large-scale Sudoku puzzles.

SolveBoard Function

The function iterates over each cell in the grid, searching for an empty cell (denoted by 0). When it finds an empty cell, it tries to place each number from 1 to the maximum valid number in that cell, checking with the isSafe function whether placing the number would violate Sudoku rules.

- If placing a number is safe, the function recursively calls itself to attempt to solve the rest of the board. If the recursive call returns true, it means the puzzle is solved, and the function returns true.
- If placing a number leads to no solution, it resets the cell to 0 (backtracks) and tries the next number.
- If no numbers can be placed in any empty cell, the function returns false, indicating the board is unsolvable with the current configuration.
- The function returns 1 (true) if the board is successfully solved.

isSafe Function

In the rules of Sudoku, any number that is placed must not be repeated by a similar number on any row or column or in the grid to which it belongs to.

The function isSafe checks whether placing a given number “num” in the specified cell at row and col in the Sudoku board is valid.

```
for (int i = 0; i < SIZE; i++) {
        if (board[i][col] == num) {
         return 0;
      }
   }
   for (int j = 0; j < SIZE; j++) {
      if (board[row][j] == num) {
         return 0;
      }
   }
   int startRow = row - row % 8;
   int startCol = col - col % 8;
   for (int i = 0; i < 8; i++) {
      for (int j = 0; j < 8; j++) {
         if (board[i + startRow][j + startCol] == num) {
            return 0;
         }
      }
   }
   return 1;
```

}

- **Column Check:** Checking through each row of the specified column(*col*). It checks if *num* is already present in that column. If found, it returns 0 indicating it's not safe to place *num* in that specific cell.

- **Row Check:** Checking through each column of the specified row(*row*). It checks if *num* is already present in that row. If found, it returns 0, indicating it's not safe to place *num* in that specific cell.
- **Subgrid Check:** It calculates the starting indices of the 8x8 subgrid that the specified cell belongs to, then checks through this subgrid if *num* is already present. If found, it returns 0, indicating it's not safe to place..
- If all checks fail to return 0, then return 1 indicating that it is safe to place.

VECTORIZATION

Once the understanding of the Sudoku code is complete, the task shifts to creating its relevant RISC-V scalar code. However, this scalar code may not be very efficient, especially when dealing with the extensive entries of a 64x64 Sudoku board. This inefficiency arises from the necessity to process each entry individually, resulting in slower computation. which is where the concept of vector comes, where we can process multiple data entries simultaneously, harnessing data parallelism to expedite computation.

a) Implementation of Vectorization

We introduced vectorization at three main points in the code: when checking rows, columns, and subgrids. Remember, for the 64x64 Sudoku board, each subgrid is now 8x8. By using vectorization, we could process multiple pieces of data at once in these areas. This made our algorithm much faster and improved its overall performance.

Before delving into vectorization within the code, it's crucial to understand the **vsetvli** command, which plays a pivotal role. This command sets the length of the vector, determining how many data elements it can handle at once. For instance, if we have the command

vsetvli t0, a4, e8, m1, it signifies that each element is 8 bits long, and our RISC-V vector register is 256 bits wide. Therefore, we can accommodate (256/8) = 32 elements of 8 bits (1 byte) each in one vector register.

However, if a4 = 4, it means we only want to utilize the vector register's capacity for 4 elements. In such a case, vsetvli considers the minimum of t0 and a4 and sets the vector register length accordingly. In our case for 64x64 sudoku board, a4 = 32 and t0 = 32, a4=32 and t= also 32 and max a vector register can support is 32 elements but our row/column size is 64 so we will be using 2 vectors to store the whole row/column data

1. Row check:

- To perform the row check, we first obtain the base address for the start of the respective row where we want to insert a new value. This base address is then passed to vector register v0, which stores the first 32 elements of that row using the command **vle8.v v0, (t1)**. Here, t1 represents the base address of the row.'
- Next, we compare each element of the row stored in v0 with the value we want to insert, denoted by a3. The command **vmseq.vx v1, v0, a3** compares each element of v0 with a3, assigning 1 if there's a match and 0 if there isn't. This results in a vector v1 with a mixture of 0s and/or 1s, indicating the presence or absence of the number in the respective row.
- To determine if the number has occurred in the row, we use the command **vpopc.m t2, v1**, which counts the number of occurrences of 1s in v1 and stores the result in scalar register t2. If t2 is nonzero, indicating that the number has occurred in the row, the program branches to the label "notSafe". However, if t2 is zero, we move onto the next 32 elements.
- Since we have just checked the first 32 elements now we will do addi t1,t1,32 and repeat the whole row check again to check the next 32 elements of the row as well if t2 is still zero then we move on towards the column check.

```
#Row check with vector registers
    mul t1, a1, s2
    add t1, s1, t1
    vle8.v v0, (t1)
    vmseq.vx v1, v0, a3
    vpopc.m t2, v1
    bnez t2, notSafe
    addi t1,t1,32
    vle8.v v0, (t1)
    vmseq.vx v1, v0, a3
    vpopc.m t2, v1
    bnez t2, notSafe
```

2. Column check:

- One simplification we made for ease in column check was converting the entire board into its transpose. This exchange of rows and columns allowed us to perform row checks on the transposed matrix. This approach was necessary because vectors inherently handle values in an ascending fashion, such as 0, 1, 2, 3, 4, 5, etc., in bytes.
- To perform the row check on the transposed matrix, we first obtained the base address for the start of the respective col where we want to insert a new value. This base address is then passed to vector register v0, which stores the first 32 elements of that row using the command **vle8.v**

v0, (t1). Here, t1 represents the base address of the start of that row.

- Next, we compare each element of the row stored in v0 with the value we want to insert, denoted by a3. The command **vmseq.vx v1, v0, a3** compares each element of v0 with a3, assigning 1 if there's a match and 0 if there isn't. This results in a vector v1 with a mixture of 0s and/or 1s, indicating the presence or absence of the number in the respective row.
- To determine if the number has occurred in the row, we use the command **vpopc.m t2, v1**, which counts the number of occurrences of 1s in v1 and stores the result in scalar register t2. If t2 is nonzero, indicating that the number has occurred in the row, the program branches to the label "notSafe". However, if t2 is zero, we move onto the next 32 elements.
- Since we have just checked the first 32 elements now we will do addi t1,t1,32 and repeat the whole row check again to check the next 32 elements of the row as well if t2 is still zero then we move on towards the subgrid check.

```
#Col Check with vector registers
   mul t1, a2, s2
   add t1, s5, t1
   vle8.v v0, (t1)
   vmseq.vx v1, v0, a3
   vpopc.m t2, v1
   bnez t2, notSafe
   addi t1,t1,32
   vle8.v v0, (t1)
   vmseq.vx v1, v0, a3
   vpopc.m t2, v1
   bnez t2, notSafe
```

3. Subgrid check:

- The subgrid check presents a bit of a challenge, especially with a 64x64 Sudoku board where each subgrid is 8x8. To accommodate this, we adjust the vector length to 8 elements at a time. We use a similar formula to locate the start of the correct subgrid (t2=row - row % 8) and

  (t5=col - col% 8). Where t2 is multiplied by 8 to generate values like 0, 8, 16, 24, etc. This value is then combined with the column address (t5) and then eventually added with s1(board address) to arrive at the correct address for the subgrid.

- Once t2 holds the start address of the subgrid, we apply the row check strategy on it. This process repeats 8 times, checking each row of the subgrid. The strategy for row check remains the same as before, only with adjusted parameters.

```
#BOX CHECK
   addi a4, zero, 8
   vsetvli t0, a4, e8, m1
   addi t1, zero, 8
   rem t2, a1, t1
   sub t2, a1, t2
   rem t5,a2,t1
   sub t5,a2,t5
addi t3, zero, 0
   boxCheckLoop:
            bge t3, t1, safe
            add t2, t2, t3
            mul t2, s2, t2
            add t2, t2, t5
            add t2, s1, t2
            vle8.v v3, (t2)
            vmseq.vx v4, v3, a3
            vpopc.m t4, v4
            bnez t4, notSafe
            addi t3, t3, 1
            rem t2, a1, t1
            sub t2, a1, t2
            j   boxCheckLoop
```

CHALLENGES

1. The installation also seemed to be quite a challenge since the total file size for the gnu toolchain was more than 10 GB which was a hectic to download.
2. Managing the large data size for both the 64x64 board, with 4096 entries, and its transposed version.
3. Frequently adjusting vector lengths to ensure accurate checks.
4. Implementing subgrid checks by determining the base address of the starting element in each subgrid.
5. Debugging was more challenging than scalar code since we relied solely on log files to interpret and identify issues.
6. Ensuring data alignment in memory to optimize vector operations.
7. The code was too large which made it difficult to understand and interpret it.

ACHIEVEMENTS

1. We were successfully able to run our 64x64 sudoku solving code on VeeR simulator.
2. Achieved a significant boost in performance compared to the scalar code, which took too long to run.
3. Data was processed in parallel, greatly increasing efficiency.
4. Handled large chunks of data effectively, for example, the elements in the row were stored in the vector which made it easier to perform the row check operation.
5. The solver, using vector instructions, can now solve boards of any dimensions, demonstrating scalability.

6. Made effective use of RISC-V vector instructions such as **vle8.v, vmseq.vx, vpopc.m,** etc.
7. Demonstrated that RISC-V vector instructions can be used to solve complex computational problems in less time with fewer computations.

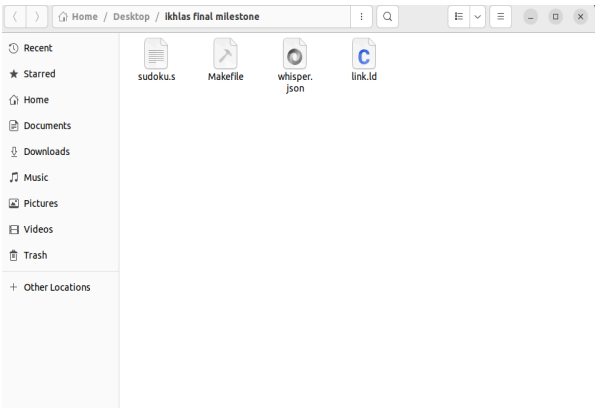
Advantages of Vectorization

- **Parallel Processing:** Vectorized instructions excel in parallel processing compared to scalar instructions. While scalar instructions handle one data element at a time, vectorized instructions process multiple data elements simultaneously, significantly accelerating the solving process of a 64x64 Sudoku board.
- **Efficiency:** Vectorized instructions offer a notable advantage in efficiency over scalar instructions. While scalar instructions rely on large loops to process each element individually, vectorized instructions execute complex operations on multiple elements simultaneously with fewer instructions. This difference in approach significantly reduces processing time, making the solver for a 64x64 Sudoku board more efficient compared to scalar-based approaches.
- **Data Handling:** Vectors have the upper hand in data handling compared to scalars. They efficiently manage the extensive dataset of a 64x64 Sudoku board, enabling rapid traversal and manipulation of board elements, which scalar instructions struggle to achieve with their sequential processing approach.
- **Scalability:** Vectorized instructions provide superior scalability in solving Sudoku boards, particularly the 64x64 variant, compared to scalar instructions. They seamlessly adapt to larger problem sizes, such as handling more rows, columns, or subgrids, making the solver scalable without sacrificing performance.
- **Optimization:** Vectorized operations are inherently optimized for performance, offering significant advantages over scalar instructions. By leveraging hardware capabilities and executing operations in parallel, they minimize computational overhead and achieve faster execution speeds, making the solver for a 64x64 Sudoku board more optimized and efficient.

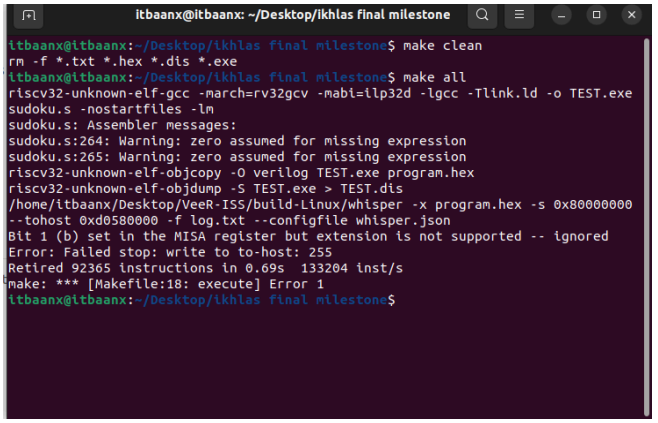
Simulation

This section will demonstrate how to run our vector code for a 64x64 sudoku solver using the VeeR simulator that we previously installed.

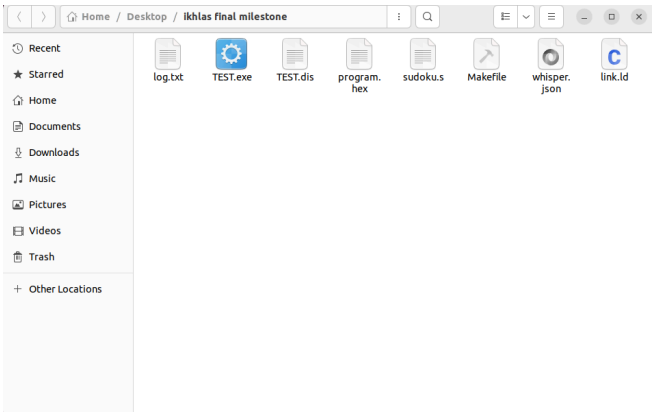
1. According to our submission you will have these files listed in a folder,



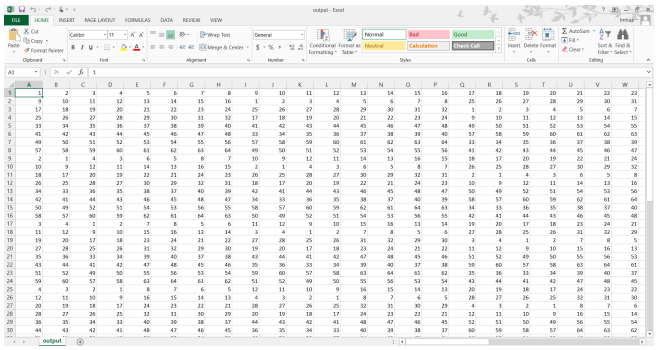
2. Next right click on this folder here to open the terminal and type in these commands:,
    - make clean
    - make all



3. Upon doing so, these files should appear in your sudoku folder.



4. The results of our output can be clearly seen in the log file but for more clarity we extracted the fully solved board from the log file using a python script which produced output as shown below. This process confirmed the proper functionality of our assembly code for Sudoku.



ACKNOWLEDGMENT

We would like to acknowledge Sir Zainuddin and Sir Salman Zafar for their direction and expertise regarding the teaching of Computer Architecture and Assembly Language.This project and it's milestones has increased the understanding regarding the use and execution of Assembly language in a specific hardware while also introducing the usefulness of vectorization in such an environment.

REFERENCES

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

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

Drive Link for the Project Content:

https://drive.google.com/open?id=1BwcUdITJSQg1yPoXXxP9azWYB0qUbokn&usp=drive_fs