Institute of Business Administration (IBA) Karachi

School of Mathematics and Computer Science

Implementation of 1D FFT in RISC-V Assembly with Vectorization

Group Name: High Risc High Reward

Group Members:

Lamaan Ali Buksh 29233

Gaurav Khatwani 29213

Abdullah Irfan 29266

Ibrahim Irfan Nazar 29225

Abstract

This project implements the 1D Fast Fourier Transform (FFT) in RISC-V Assembly using both scalar and vectorized approaches. The work focuses on optimizing FFT operations through vectorization, recursion, and mathematical approximations.

This implementation is split across two standalone codebases: one non-vectorized (scalar) and the other vectorized using RISC-V's vector instructions simulated on Veer (Whisper) in VSCode in a virtual machine.

Core functions like bit-reversal, twiddle factor generation, and butterfly operations are written from scratch, with focus on converting scalar instructions to vector ones, turning iterative functions into recursive forms, and efficiently calculating trigonometric functions using Taylor series expansion.

1. Introduction

The Discrete Fourier Transform (DFT) transforms time-domain signals into frequency-domain components but suffers from high computational cost O(N^2). Fast Fourier Transform (FFT) reduces this to O(N log N).

In this project, we implemented the Radix-2 FFT on RISC-V using two versions:

- A scalar-only implementation for simplicity and debug visibility.
- A vectorized implementation using the RISC-V Vector Extension to accelerate computation on large data sizes.
- Both these methods can be scaled up and down regarding the data size.
- Both these methods had 3 main components:

  - Array Index Bit Reversal: This reverse index values and swaps their data. This help divide the array into its even and odd constituients, for ease in loading during butterfly.
  - Butterfly Operations: These divide the original array x of N size into 2 and calculate the final output X using a divide-and-conquer strategy (sort of like merge-sort) in either recursive or iterative calls to the following equations:

    - ❖ $X[k] = X_{\text{even}}[k] + W_N^k + X_{\text{odd}}[k]$
    - ❖ $X[k + N/2] = X_{\text{even}}[k] - W_N^k X_{\text{odd}}[k]$

  - Twiddle factor Computations: Twiddle factor W^k_n rotates the complex numbers in the DFT computation, allowing efficient decomposition. The twiddle factor reduces redundant calculations by exploiting symmetry and periodicity in DFT and helps in our divide and conquer FFT approach. It is calculated using:

    - ❖ W^{fn}_N = $x[n] = \cos\left(\frac{2\pi fn}{1024}\right) + j\sin\left(\frac{2\pi fn}{1024}\right)$

    where:
    - ➢ *N* is the total number of points, here it is 1024, due to our requirement.
    - ➢ *f* is the frequency index,
    - ➢ *n* is the time index.

2. Scalar FFT Implementation

We started approaching this project using scalar RISC-V ISA. Both the twiddle function and the butterfly operations were firstly written in iterative formats for simplicity. The scalar version consists of basic FFT operations written in conventional iterative style. It uses:

- **BitRevFunc** and **BitRev** to reorder input using bit-reversed indices.

- **GenTwdl** was used to compute cosine (real) and sine (imaginary) values via Taylor series expansion, set to 16 iterations per term for better accuracy. The number of iterations can be increased or decreased as needed.

- **FFTMain** performs the butterfly operations in a traditional iterative loop.

The twiddle factor calculation function was converted to recursive thereafter with the help of online resources and AI, for memory efficiency and better flow control. However the twiddle function was thereafter removed to adopt a much simpler logic, that employed storing the twiddle factors in the Data Memory. This version was tested with an 8-point FFT and scaled up to 1024 points.

3. Vectorized FFT Implementation

The vectorized version leverages SIMD capabilities of RISC-V Vector ISA in Veer(Whisper). Major functions include:

- **BitRevFunc** reused for input reordering.

- **GenTwdl** uses scalar recursion but stores vector-compatible results.

- **FFTMain** restructured to perform butterfly operations using vector registers and vector ALU instructions (e.g., “vadd.vv” (vector add) , “vmul.vv” (vector multiply)). This allows for ease in operations and efficiency, as we can process chunks of Data.

For arrays with 1024 elements, the speed improvement is much more noticeable. The “vsetvli” instruction helps achieve SIMD parallelism by automatically adjusting the vector length to match the hardware's capabilities.

4. Function Descriptions

- **BitRevFunc** & **BitRev**: Compute the bit-reversed indices and perform in-place swaps in the array. These functions are used in both versions.
- **GenTwdl**: Generates twiddle factors using Taylor series for sine and cosine. The function handles scalar approximations of sin and cos through multiple floating-point iterations.
- **FFTMain**: This is the central FFT loop. In the scalar version, it uses nested loops; in the vectorized version, it leverages vectorized loading, multiplication, and butterfly operations.

- **printToLogVectorized**: A utility designed to load vectorized FFT output into memory to assist in debugging or logging using external scripts. It shows efficient logging of real and imaginary parts using vector loads and pointer arithmetic. It also shows the final output of the FFT in a printed matrix form.

5. Challenges and Observations

- Conversion from scalar to vector required realignment of memory and restructuring of butterfly operation loops to suit vector lengths.
- Conversion from iteration to recursion was essential for Taylor-based trigonometric computation due to need of high precision hence higher number of terms required to compute them. For similar reasons we decide to convert the Butterfly operations into recursive to help in memory efficiency and remove chances of garbage values effecting the final output.
- Register cleanup was required in iterative approach due to limitation of only 32 registers in RISC V 32-bit ISA.
- Precision issues were addressed by using 16 iterations in the Taylor series expansion for both sine and cosine.
- Had to learn and understand RISC V Vector ISA to aid in conversion from scalar to vector instructions.
- Debugging and writing codes for 8 data point was much easier and allowed scaling up much quicker, similarly iteratively writing the functions and debugging them, allowed for problems to be mitigated in recursive methods.
- Bit-reversal and vector butterfly operations had to be coordinated precisely to avoid memory overlap or indexing issues.
- Debugging vector instructions in VEER was more complex due, leading to understanding `` ` ``**printToLogVectorized**`` ` `` for checkpointing results and then understanding the log file to help in debugging.

6. Remaining Work:

- Scalar FFT approach works correctly, however still some debugging is required, due to some incorrect values in output, most likely caused by register not being cleaned.

- Individual fragments of the FFT were developed using scalar instructions and tested on Venus online simulator. These were then vectorized and stitched together, hence it is currently not working Properly on VEER simulator.

7. Conclusion

Two standalone implementations of the 1D FFT were developed in RISC-V: scalar and vectorized. Both versions highlight different trade-offs.

- The scalar version is easy to debug and readable.
- The vectorized version is optimized for speed and parallelism.

Together, they demonstrate progressive optimization in low-level 1D FFT computation, from basic logic to advanced SIMD-based acceleration.