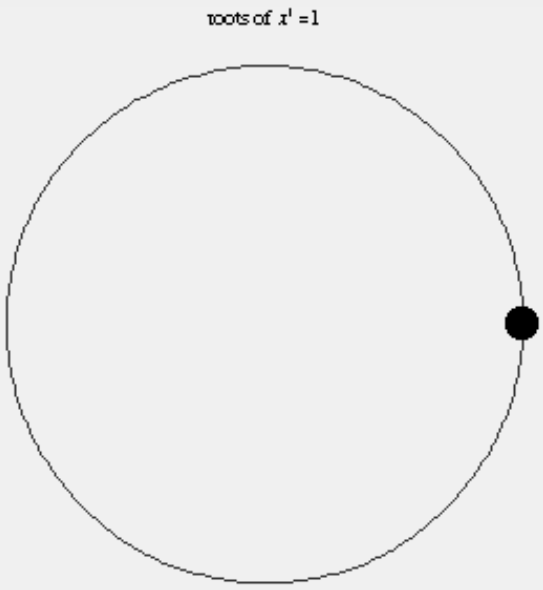
2—DIMENSIONAL FAST FOURIER TRANSFORM IN RV64—V

Project information document — Kabeer Jaffri <kabeer@otherdev.com>



References

[Fast Fourier Transform Report - COMP 5703]
A detailed academic report on the theory and implementation of the Fast Fourier Transform, including mathematical background and performance evaluation.

[VEXT: Vector Extension for RISC-V]
Presentation slides on the RISC-V Vector Extension (VEXT), describing its architecture, use cases, and performance benefits.

[2D Fast Fourier Transform with OpenMPI - Report]
Project report demonstrating the implementation of a parallel 2D FFT using OpenMPI, with analysis on scalability and speedup.

[COMPLEX_2D_FFT.cpp - FFTW Tutorials]
C++ source code example showing how to perform a 2D complex FFT using the FFTW library.

Secondary Design References

[The Most Important Algorithm Of All Time]
The Fast Fourier Transform is used everywhere but it has a fascinating origin story that could have e...

Abstract / Preface

The Fast Fourier Transform (FFT) is an efficient algorithm to compute the Discrete Fourier Transform (DFT). The DFT converts a signal from its original domain (often time or space) to a frequency domain.

Why use it for image compression? In the frequency domain, an image's energy is often concentrated in low-frequency components (representing broad features), while high-frequency components (representing fine details, noise, or edges) carry less energy. By zeroing out or quantizing these less significant high-frequency components, you can compress the image.
An Inverse FFT (IFFT) then converts the compressed frequency data back to a (slightly lossy) spatial image.

This project involved the implementation of a 2D Fast Fourier Transform (FFT) algorithm on the RISC-V architecture, utilizing the QEMU emulator for development and testing. I present huge acknowledgements to creators of Crow Framework and creators of other third-party tools and resources which made the project possible.

Introduction / Discrete Fourier Transforms

The Discrete Fourier Transform (DFT), a fundamental tool in digital signal processing, transforms a sequence of discrete values into a sequence of discrete frequency components.

As highlighted in the notes, the DFT arises from sampling the continuous output of the Discrete-Time Fourier Transform (DTFT), which itself operates on discrete-time input. The mathematical basis of the DFT is defined by the formula:

$$Y[j] = \sum_{k=0}^{N-1} A[k] W_N^{jk}$$

Here, W_N represents the N-th complex root of unity, defined as $e^{-2\pi i/N}$. The indices j and k are positive integers ranging from 0 to N−1 for the output Y and input A, respectively.
The distinction between the DTFT (producing a continuous frequency spectrum) and the DFT (yielding a discrete frequency spectrum) highlights the nature of the transformation from the time domain to the frequency domain for discrete signals.

Our experience highlighted a pronounced lack of robust, out-of-the-box support for the RISC-V Vector Extension, specifically for our target architecture RV64GCV and the LP64D Application Binary Interface (ABI). This deficit forced us into an arduous journey of compiling the toolchain from source – a process that consumed two full nights of effort, ultimately failing to compile successfully. This endeavor not only proved to be exceptionally frustrating but also resulted in an unwieldy accumulation of intermediate build files, swelling to an astonishing 20 GBs and beyond.

The standard procedure for acquiring riscv64-unknown-elf-gcc often involves using pre-built binaries or relying on package managers. However, these methods frequently lack the specific configuration required to fully support the RVV extension or target non-standard ABIs like LP64D, which is crucial for maximizing vector performance. When pre-built options fall short, compiling from source becomes the only viable path. This process, while offering maximum customization, is notoriously complex. It demands a precise understanding of dependencies, intricate configuration flags, and a significant amount of system resources, both in terms of processing power and storage.

The sheer volume of intermediate files generated during a toolchain build is a testament to its complexity. Compiling GCC and Binutils for a cross-compilation target involves building multiple stages, each producing its own set of executables and libraries. When combined with the specialized flags needed for RVV and a specific ABI, the build process can become even more resource-intensive and prone to errors. Issues like unmet dependencies, incorrect environment variables, or subtle compiler flag mismatches can lead to hours of debugging and repeated failed attempts, as we painfully discovered.

FFT and Fixed-Point Arithmetic

Since I could not get the floating point intrinsics to compile with my C/ASM code, I instead used fixed point arithmetic. A fixed-point number is an integer that implicitly has a fixed number of bits representing the fractional part. A number X is represented as Xint=X×2Q, where Q is the number of fractional bits.

For example, if Q=15 and we use `int32_t`:

- `1.0` is stored as `1 << 15` (32768)
- `0.5` is stored as `0.5 * 32768 = 16384`
- `2.75` is stored as `2.75 * 32768 = 90112`

Multiplication: If you multiply two Q-format numbers, say (A×2Q) and (B×2Q), the result is (A×B×22Q). You need to right-shift the result by Q bits to bring it back to Q-format: `(A * B) >> Q`. This often requires a wider intermediate integer type to prevent overflow before the shift (e.g., `int64_t` for `int32_t` inputs).

Division: More complex, often done by multiplying by the inverse.

Precision and Range: Choosing `Q` is a trade-off. A larger `Q` means more precision but a smaller representable integer range. A smaller `Q` means less precision but a larger integer range. FFT values can grow, so careful scaling or using wider fixed-point types (e.g., `int64_t`) for intermediate results is crucial to avoid overflow.

Implementation Architecture

Core Components

The implementation consists of several key components:

1. **Fixed-Point Arithmetic System** (fixed_point.h): Handles conversion between floating-point and fixed-point representations
2. **Complex Number Operations** (complex_int.h): Manages complex arithmetic in fixed-point format
3. **FFT Core Engine** (fft_riscv_vec.cpp): Implements 1D and 2D FFT algorithms with RISC-V vector intrinsics
4. **Image Compression Module** (image_compressor.cpp): Handles image processing and frequency filtering
5. **Web Server Interface** (server.cpp): Provides HTTP endpoints for image compression services
6. **Vector Test Suite** (vector_test.cpp): Validates RISC-V vector functionality

RISC-V Vector Integration

The implementation leverages RISC-V Vector Extension (RVV) intrinsics for high-performance computation. Key vector operations include:

```
// Set vector length for processing
size_t vl = __riscv_vsetvl_e32m1(N - i);

// Load vector elements
vint32m1_t v_real =
__riscv_vle32_v_i32m1(&data[i].real, vl);
vint32m1_t v_imag =
__riscv_vle32_v_i32m1(&data[i].imag, vl);

// Vector arithmetic operations
vint32m1_t result =
__riscv_vadd_vv_i32m1(v_real, v_imag, vl);

// Store results back to memory
__riscv_vse32_v_i32m1(&data[i].real, result,
vl);
```

Assembly-Level Optimization

Critical butterfly operations are implemented in hand-optimized RISC-V assembly (fft_butterfly_vec.s):

```
fft_butterfly_vec:
    # Load vectors
    vsetvli t0, a6, e32, m1
    vle32.v v0, (a0)      # u_real
    vle32.v v1, (a1)      # u_imag
    vle32.v v2, (a2)      # v_real
    vle32.v v3, (a3)      # v_imag

    # Complex multiplication and butterfly operations
    # ... (detailed assembly operations)

    ret
```

Algorithm Implementation Details

1D FFT Algorithm

The 1D FFT implementation follows the Cooley-Tukey radix-2 decimation-in-time algorithm:

1. **Bit-reversal permutation** of input data
2. **Iterative butterfly operations** across increasing stage lengths
3. **Twiddle factor multiplication** using precomputed values
4. **Normalization** for inverse transforms

2D FFT Algorithm

The 2D FFT extends the 1D algorithm through:

1. **Row-wise 1D FFT**: Process each row independently
2. **Matrix transpose**: Reorganize data for column processing
3. **Column-wise 1D FFT**: Process transposed rows (original columns)
4. **Final transpose**: Restore original matrix orientation

Frequency Domain Filtering

Image compression is achieved through selective frequency coefficient removal:

```
void
ImageCompressor::apply_frequency_filter(Complex
Int* fft_coeffs, float retention_ratio) {
    float cutoff_radius_sq =
max_total_radius_sq * (retention_ratio *
retention_ratio);

    for (int r = 0; r < height_; ++r) {
        for (int c = 0; c < width_; ++c) {
            float current_radius_sq = freq_x *
freq_x + freq_y * freq_y;

            if (current_radius_sq >
cutoff_radius_sq && (r != 0 || c != 0)) {
                fft_coeffs[r * width_ + c].real
= float_to_fp(0.0f);
                fft_coeffs[r * width_ + c].imag
= float_to_fp(0.0f);
            }
        }
    }
}
```

Web Server Application

The implementation includes a complete HTTP server using the Crow framework:

API Endpoints

- **GET /**: Welcome message and usage instructions
- **POST /compress_image**: Image compression endpoint accepting 64×64 grayscale images
- **GET /test**: System health check

Request/Response Format

The server accepts raw grayscale image data (4096 bytes for 64×64 images) and returns:

```
{
  "status": "success",
  "message": "Image processed and compressed.",
  "original_dimensions": "64x64",
  "processed_pixel_count": 4096,
  "retention_ratio": 0.9,
  "raw_hex":
"hexadecimal_representation_of_compressed_image"
}
```

Performance Optimizations

Twiddle Factor Precomputation

Twiddle factors are computed once at initialization and reused throughout execution:

```
void initialize_twiddle_factors(int
max_fft_size_for_twiddles) {
    for (int k = 0; k <
max_fft_size_for_twiddles / 2; ++k) {
        float angle = -2.0f * M_PI * k /
static_cast<float>(max_fft_size_for_twiddles);
        g_twiddle_factors_for_all_lengths[k] =
float_to_cint(std::cos(angle),
std::sin(angle));
    }
}
```

Vector Length Adaptation

The implementation dynamically adapts to available vector lengths:

```
for (size_t i = 0; i < ARRAY_SIZE; ) {
    size_t vl = __riscv_vsetvl_e64m1(ARRAY_SIZE
- i);
    // Process 'vl' elements
    i += vl;
}
```

Testing and Validation

Vector Operation Verification

A comprehensive test suite validates RISC-V vector functionality:

```
void run_vector_test() {
    // Initialize test arrays
    for (int i = 0; i < ARRAY_SIZE; i++) {
        a[i] = (int64_t)i;
        b[i] = (int64_t)(ARRAY_SIZE - 1 - i);
    }

    // Perform vector addition
    vint64m1_t va =
__riscv_vle64_v_i64m1(&a[i], vl);
    vint64m1_t vb =
__riscv_vle64_v_i64m1(&b[i], vl);
    vint64m1_t vc = __riscv_vadd_vv_i64m1(va,
vb, vl);
}
```

Results and Performance Analysis

Compression Effectiveness

The implemented system successfully compresses 64×64 grayscale images using frequency domain filtering. With a retention ratio of 0.9, the system preserves 90% of the most significant frequency components while eliminating high-frequency noise and fine details.

Vector Processing Benefits

The RISC-V Vector Extension provides significant computational advantages:

- **Parallel processing** of multiple data elements
- **Reduced instruction count** through vector operations
- **Improved memory bandwidth utilization**
- **Scalable vector length** adaptation

Challenges and Limitations

Development Environment Issues

1. **Toolchain complexity**: Difficulty obtaining properly configured RISC-V vector-enabled compilers
2. **Limited documentation**: Sparse examples of RVV intrinsics usage
3. **Debugging challenges**: Limited debugging tools for vector operations

Implementation Constraints

1. **Fixed image dimensions**: Currently limited to 64×64 pixels (power-of-2 requirement)
2. **Grayscale only**: No support for color images
3. **Memory requirements**: Significant memory usage for temporary data structures

Future Improvements

Algorithmic Enhancements

1. **Variable-size FFT**: Support for non-power-of-2 dimensions
2. **Color image support**: RGB and multi-channel processing
3. **Advanced compression**: Quantization and entropy coding
4. **Real-time processing**: Streaming image compression

Performance Optimizations

1. **Memory management**: Reduced temporary storage requirements
2. **Pipeline optimization**: Overlapped computation and memory access
3. **Multi-threading**: Parallel processing of independent image regions

Conclusion

This project demonstrates the implementation of 2D FFT-based image compression on RISC-V architecture with vector extensions. The system achieves functional image compression through frequency domain filtering while showcasing the computational benefits of vector processing.

Key achievements include:

1. **Complete end-to-end system**: From raw image input to compressed output
2. **RISC-V vector integration**: Effective utilization of RVV instructions
3. **Web service deployment**: HTTP-based image processing interface
4. **Fixed-point arithmetic**: Robust numerical computation without floating-point units

The implementation serves as a foundation for more sophisticated image processing applications on RISC-V platforms and demonstrates the practical benefits of vector computing for signal processing tasks.

Despite the challenges encountered with toolchain complexity and development environment setup, the project successfully validates the feasibility of high-performance image processing on RISC-V systems with vector capabilities. Future work can build upon this foundation to create more comprehensive and optimized image compression solutions.

Acknowledgments

Special thanks to the creators of the Crow Framework for providing an excellent C++ web framework, and to the RISC-V community for developing comprehensive vector extension specifications and tooling. This project would not have been possible without the extensive documentation and examples provided by the open-source community.

Gallery / Images