A High-Performance Rust Implementation for Spectral Analysis

Zoe Krishtul
zo110459@ucf.edu

Natalya McKay
zea_64@proton.me

Charles Jorge
ch256657@ucf.edu

Daniel Palma
dany@ucf.edu

***Abstract*—The Fast Fourier Transform (FFT) is a computational tool used in signal processing, image analysis, and scientific computing. Traditional FFT implementations often encounter performance limitations, especially when handling large datasets and real-time data processing. This project exhibits a parallelized implementation of the Cooley-Tukey FFT algorithm in Rust, using Rust's concurrency features to improve performance. The goal of our implementation is to enhance efficiency while keeping memory safety and scalability. We are comparing our approach against standard FFT libraries, showing its effectiveness in real-time spectral analysis applications. Additionally, we show how Rust simplifies parallelization compared to languages like C, where manual memory management and thread synchronization can introduce complexity and other issues. Unlike C, Rust provides built-in features that make parallel programming more effective.**

***Index Terms*—Digital signal processing, Fast Fourier transform, Multithreading, Multiprocessor**

I. Introduction

Digital signal processing is an important area of computing, used in everything from radio communications to audio processing. A key requirement in much of digital signal processing is the discrete Fourier transform, taking a sequence of n discrete samples and calculating the amplitudes of n component frequencies. A naive approach to computing a discrete Fourier transform would take $O(n^2)$ steps to complete, which would be prohibitively slow and expensive for most uses of it today. Cooley and Tukey solved this problem with their now famous Fast Fourier Transform, exploiting symmetries in the calculation to derive a $O(n \lg(n))$ algorithm to compute the discrete Fourier transform.

The Fast Fourier Transform has revolutionized digital signal processing, and is now a ubiquitous algorithm with specialized implementations in many different programming languages and running on billions of heterogenous devices. A programmer wishing to use the Fast Fourier Transform can pick from a variety of implementations in whatever programming language they wish without having to worry about the implementation details. However, most of these libraries use only a single processor thread, leaving the rest of the hardware's threads unused. For small block sizes this does not matter much, as an FFT can be computed within a few microseconds, but as the block size grows the amount of work grows too, proportional to $O(n \lg(n))$. On today's multiprocessor machines, to achieve the maximum throughput an application must be programmed to divide work between all of the machine's threads.

Traditionally, highly optimized FFT libraries in low-level languages (C/C++) have been used to meet performance demands. For example, FFTW – "The Fastest Fourier Transform in the West" – is widely regarded as one of the fastest FFT libraries available [1], employing hardware-specific optimizations and adaptive planning to maximize performance [1], [2]. High-level environments like Python delegate FFT computations to underlying native code (NumPy's FFT is a C port of the FFTPACK library [3]). While convenient, these high-level interfaces typically do not exploit parallelism by default and may lag behind hand-tuned libraries in speed [3]. Leveraging multiple CPU cores or GPUs can further accelerate large FFTs – for instance, FFTW includes support for multithreaded and distributed transforms [4],and GPU-based FFTs excel for very large sizes.

The Rust programming language has emerged as a compelling option for high-performance computing due to its ability to deliver C-like performance with memory safety guarantees. Rust's ownership model and type system prevent data races and memory corruption, enabling "fearless concurrency" where many multi-threading errors are caught at compile time [5]. This makes it feasible to write complex parallel algorithms without the usual risks of synchronization bugs. In the Rust ecosystem, the RustFFT crate is a pure-Rust FFT library optimized with SIMD instructions (supporting AVX, SSE, NEON) [6]. RustFFT uses a planner to choose efficient algorithms and can handle arbitrary sizes [6]. We focus on how Rust's features (memory safety, threads, and the Rayon library for parallelism) enable a high-performance solution without sacrificing reliability. The main contributions are: (1) a recursive, parallel FFT algorithm using Rayon's join and parallel iterators, (2) an integration of this algorithm into an audio visualization pipeline (reading WAV audio via hound, computing spectra, and displaying via egui GUI while audio plays via rodio), and (3) a thorough evaluation comparing our implementation to RustFFT in terms of speed, memory usage, and accuracy. We also discuss related FFT implementations in C and Python and highlight that Rust can achieve competitive performance.

II. Problem Statement

We identified an opportunity to scale the traditionally linear Cooley-Tukey FFT across threads of a multiprocessor system to improve throughput for large FFT block sizes.

A. Real-time Spectral Analysis:

The target application is an audio spectrum analyzer that processes an input audio signal (e.g. a WAV music file) and displays its frequency spectrum in real time. This involves reading audio samples (time-domain) in successive frames, computing an FFT for each frame, and visualizing the magnitude spectrum before the next frame arrives. For smooth visualization, frame sizes on the order of 512–2048 samples (10–50 ms at audio rates) are typical. We choose a frame size of 1024 samples in our implementation, as it is a power of two (convenient for Cooley–Tukey FFT) and provides a good balance between time resolution (~23 ms per frame at 44.1 kHz) and frequency resolution (~43 Hz per FFT bin)

B. Performance Requirements:

To maintain real-time behavior, the FFT computation for each 1024-sample frame must complete well within the frame's duration (23 ms). In practice, this is not a tight constraint on modern CPUs – even a non-optimized 1024-point FFT can execute in microseconds. However, for scalability and future-proofing, we aim to utilize available CPU cores to handle larger FFT sizes or multiple concurrent streams. By parallelizing the FFT, we can ensure that even much larger transforms (e.g. for higher-resolution spectral analysis) or multiple FFTs per frame (e.g. stereo channels, or filter bank computations) can be performed in real time. The challenge is to achieve this parallelism without introducing correctness issues (no race conditions) and without unduly increasing latency or memory usage.

C. Safety and Integrability:

Another requirement is to maintain the safety and reliability of the application. Audio visualization is an interactive, long-running application; memory leaks or crashes are unacceptable. Implementing an FFT in a low-level language (C/C++) and parallelizing it with threads (pthreads or OpenMP) could meet performance targets, but risks memory-safety bugs or race conditions that can be hard to debug. We seek a solution in Rust that inherently prevents these errors. The solution should integrate with high-level application code – in this case, the GUI and audio I/O – demonstrating that one can combine system-level performance with application-level safety in one language.

In summary, the problem is to implement a **high-performance, parallel FFT in Rust** that meets real-time processing demands for 1024-sample audio frames, leverages multi-core processors for scalability, and maintains Rust's guarantees of memory safety and fearless concurrency. We will compare this implementation to a modern Rust FFT library (RustFFT) to quantify performance gains and overheads, and ensure that the spectral results are accurate.

III. Related Work

FFT algorithms and implementations have been extensively studied over the past five decades. Cooley and Tukey's landmark 1965 paper [7] introduced the divide-and-conquer approach that underlies most modern FFT implementations. Many variations (radix-2, mixed-radix, prime-factor, etc.) and optimizations (in-place computation, cache-friendly access patterns, vectorization) have been developed since [8] Here we focus on widely used libraries and their characteristics in relation to our work.

C/C++ Libraries: FFTW is often considered the gold standard of FFT libraries. It is written in C and generates highly optimized code for the target architecture at runtime ("planner") [8]. FFTW supports real and complex FFTs of arbitrary size and dimensionality. It chooses among various algorithms (Cooley–Tukey, prime-factor, Rader's, Bluestein's) depending on the size, and even produces hard-coded FFT kernels for small sizes to maximize instruction-level parallelism. FFTW's performance is typically superior to other free FFT software and even competitive with vendor-tuned proprietary libraries [2]. Notably, FFTW can exploit multiple threads (and SIMD units) – it includes support for shared-memory parallel FFTs using threads or OpenMP. In our context, FFTW represents what a highly optimized C implementation can achieve. However, directly integrating FFTW in a Rust project can be problematic due to FFTW's GPL license and the need for FFI (foreign function interface) calls. Moreover, as noted by Matthews et al. [1], Rust cannot easily reuse FFTW's code generation or plans, prompting the need for native Rust alternatives.

Another notable library is Intel's MKL (Math Kernel Library), which provides extremely optimized FFT routines (often underlying MATLAB or NumPy on Intel platforms). Like FFTW, MKL can utilize multi-core and vector instructions. These libraries, while very fast, are low-level and require careful handling of memory and pointers, in contrast to Rust's safe abstractions.

Python and High-Level: Python's convenience in signal processing comes from wrappers over these native libraries. NumPy's numpy.fft module is based on FFTPACK (a Fortran library from 1985) and is implemented in C for efficiency [3]. SciPy's scipy.fftpack historically used the same routines, and more recently SciPy's scipy.fft defaults to pocketfft (a newer C++ implementation). Typically, NumPy's FFT is sufficient for moderate sizes, but for very large transforms or demanding applications, Python users turn to libraries like PyFFTW, which interfaces with FFTW. Benchmarks have shown that using FFTW through PyFFTW can significantly outperform NumPy's built-in FFT, especially for large 2D FFTs [3]. This underscores that the underlying algorithmic efficiency and multi-threading (in FFTW's case) make a difference for performance. In our application (1024-point FFTs), even an unoptimized FFTPACK-based routine would meet real-time needs in absolute terms. However, exploring Python-level approaches illustrates the trade-off between simplicity and performance: pure Python implementations are orders of magnitude too slow (due to interpreter overhead), so native

code is required, but achieving parallelism in Python can be non-trivial due to the GIL (Global Interpreter Lock). Our Rust approach aims to marry high-level simplicity with low-level speed, eliminating the need for Python as glue.

Rust Libraries: The primary FFT library in Rust is RustFFT. It is a community-developed library written entirely in safe Rust, inspired by the structure of FFTW and KissFFT. RustFFT implements mixed-radix and prime-length FFT algorithms and automatically chooses the appropriate strategy for a given size [1]. It also implements a planner API (FftPlanner) to reuse computations across FFT calls [5]. Critically, RustFFT includes SIMD acceleration: on x86_64, it detects at runtime whether AVX or SSE4.1 is available and uses vectorized kernels accordingly [6].

This allows RustFFT to approach the performance of C libraries while remaining pure Rust. According to its documentation, RustFFT can compute any size N in $O(N \log N)$ time. RustFFT does not spawn multiple threads internally – it focuses on single-core performance. A recent study by Rooney and Matthews compared RustFFT with FFTW on embedded ARM devices (Raspberry Pi) and found that RustFFT's single-core performance was very competitive: on a Raspberry Pi 4, RustFFT (1 thread) was the fastest for large N, even ~37% faster than a custom C FFT at the highest sizes tested. This suggests that Rust's performance, even without explicit parallelism, can rival C, thanks to optimized algorithms and SIMD usage.

Their work also introduced a custom parallel FFT in Rust (somewhat analogous to ours) and demonstrated that memory usage for Rust implementations was 1.3–1.75× lower than for equivalent C implementations [1], likely due to Rust's efficient memory management. These findings motivate our approach – if RustFFT is already close to FFTW on one core, then using Rust to parallelize across cores could yield a truly high-performance FFT.

Apart from RustFFT, there have been experimental projects (e.g. Phantom FFT, RustDCT for discrete cosine transforms, and PhastFFT) aiming to implement FFTs or related transforms in Rust, some focusing on no `unsafe` code and simplicity. Rayon, the data-parallelism library we use, has been used in other numeric computing projects in Rust to great effect, but we haven't found a widely used crate that provides parallel FFT out of the box. Thus, our work helps fill that gap. In summary, our implementation draws inspiration from Cooley–Tukey and existing FFT libraries. We aim to combine **FFTW-like performance** (multi-core, optimized computation) with **RustFFT-like safety and simplicity**, demonstrating that Rust can achieve both. We use concepts from prior art – recursion and divide-and-conquer from Cooley–Tukey, plan-based structure from RustFFT – but implement them in a uniquely Rusty way using Rayon for parallelism and high-level iterators for clarity.

IV. Cooley-Tukey FFT

The Cooley-Tukey Fast Fourier Transform is a divide-and-conquer algorithm that can operate on complex-valued samples (real samples would set the imaginary component to 0). The algorithm first divides the samples into odd and even indices, then recursively calls itself on the odd and even index samples. After those calls complete, the samples are "twiddled" using roots of unity and the results are finally returned to the caller.

```
fn fft(input: &mut [Complex]) {
 if input.len() == 1 {
  return;
 }

 let mut evens = get_evens(input);
 fft(&mut evens);

 let mut odds = get_odds(input);
 fft(&mut odds);

 twiddle(&mut input, &evens, &odds);
}
```

Listing 1: FFT pseudocode

While the twiddling could potentially be parallelized, it did not seem very profitable to do so. However, the divide-and-conquer aspect where the algorithm recursively calls itself on odd and even indices seemed promising, as each call is completely independent of the other and represents and substantial unit of work. Running the recursive calls in parallel would only incur synchronization cost for starting and completing these parallel tasks, and no need for data synchronization while the tasks execute.

V. Parallelization with Rayon

We used the popular Rayon crate from the Rust ecosystem to give us our parallelism primitives. Rayon provides a pool of threads with one thread per detected CPU thread and schedules units of work (tasks) automatically across these threads. Spawning a task is as simple as passing a closure to `rayon::spawn`, or if you want to block waiting for multiple tasks, passing two closures to `rayon::join`.

```
let expensive1 = || {
 do_something();
};

let expensive2 = || {
 do_something();
};

// Run these potentially in parallel.
let (result1, result2) = rayon::join(
 expensive1,
 expensive2,
);
```

Listing 2: Example of using `rayon::join`

Critically, tasks spawned on these threads can themselves spawn more tasks, and if a task blocks waiting on another task Rayon will use the blocking call to run other work. This means that the threads themselves will not block due to running tasks blocking on other tasks. This allows us to two fork child tasks

from an FFT call recursively without the blocked parent tasks causing the entire thread pool to deadlock.

```
fn fft(input: &mut [Complex]) {
 if input.len() == 1 {
  return;
 }

 let mut evens = get_evens(input);
 let mut odds = get_odds(input);

 rayon::join(
  || fft(&mut evens),
  || fft(&mut odds),
 );

 twiddle(&mut input, &evens, &odds);
}
```

Listing 3: FFT with recursive calls parallelized

Finally, to maintain good computation batching and avoid catastrophic task scheduling overhead, forking is limited based on the number of CPU threads detected and falls back to simple serial recursive calls as before.

VI. Technique

We implemented a **recursive** radix-2 Cooley–Tukey FFT that splits the input into even-indexed and odd-indexed samples, recursively computes half-size FFTs, and then combines the results. Pseudocode for the core algorithm is shown below:

```
fn fft(input: &mut [Complex<f32>]) {
  let n = input.len();
  if n <= 1 {
    return;
  }

  // Split input into evens and odds
  let mut evens: Vec<Complex<f32>> =
input.iter()...
  let mut odds: Vec<Complex<f32>>  =
input.iter()...

  // Recurse (in parallel or sequentially)
  fft(&mut evens);
  fft(&mut odds);

  // Combine results with twiddle factors
  for k in 0..n/2 {
    let twiddle = from_polar(1.0, -2.0 * PI
* (k / n) * odds[k];

    input[k] = evens[k] + twiddle;
    input[k + n/2]  = evens[k] - twiddle;
  }
}
```

Listing 4: Simplified recursive FFT (radix-2, out-of-place split, in-place combine).

This function assumes n is a power of two. It first copies even-indexed elements and odd-indexed elements into new vectors (`evens` and `odds`). Then it calls itself recursively on those halves, and finally applies the so-called "butterfly" operations to combine the half-size results into the full-size result. The complex multiplication by $\exp\left(\frac{-2\pi i k}{n}\right)$ is represented by `Complex::from_polar(1.0, -2πk/n)` (computing $\cos\left(\frac{2\pi k}{n}\right) + i\sin\left(\frac{2\pi k}{n}\right)$) and multiplying by the odd-part result. In a sequential implementation, the above recursion would yield the correct FFT (forward transform). However, the key observation is that the two recursive calls `fft(&mut evens)` and `fft(&mut odds)` are independent – they operate on disjoint halves of the data. They can therefore be executed in parallel. We utilized Rayon, a data-parallelism library in Rust, to achieve this. Rayon allows spawning tasks in a fork-join style with zero-cost abstractions. Specifically, we replaced the sequential recursion with `rayon::join`, which runs two closures in parallel and waits for both to finish.

```
if forks_left > 0 {
 rayon::join(
  || fft(&mut evens, forks_left - 1),
  || fft(&mut odds, forks_left - 1),
 );
 }
 else {
  fft(&mut evens, 0);
  fft(&mut odds, 0);
 }
```

Here, `forks_left` is a parameter that controls the recursion depth at which we continue to fork in parallel. We determine `forks_left` based on the available parallelism: we query the number of CPU cores and set `forks_left = log2(cores)` (rounded up).

```
let parallelism =
thread::available_parallelism()
      .map(|x| x.get())
      .unwrap_or(4);

our_fft(input,
parallelism.next_power_of_two().ilog2() as
i8);
```

Listing 6: caption needed

For example, on an 8-core machine, `forks_left` is 3, meaning the recursion will spawn parallel tasks for 3 levels, creating at most $2^3 = 8$ concurrent tasks. Beyond that, the recursion continues sequentially. This prevents oversubscribing threads and limits overhead. The base case or when `forks_left` reaches 0 triggers sequential recursion (no further parallel fork).

By using Rayon's `join`, we benefited from its *work-stealing scheduler*: threads are managed automatically, and idle threads can steal work from busy threads, leading to good load balancing. The `join` call is blocking until both recursive halves complete, after which the combination loop runs on whichever thread resumes the parent call. Crucially, Rust's borrow checker and ownership model ensure no data races: `evens` and `odds` are separate `Vec` buffers, and `input` is not accessed during the parallel region (except to copy into

those buffers beforehand). Each recursive call operates on distinct data. Thus, even without explicit locks, our parallel recursion is safe by construction. The only shared state is the `forks_left` counter (passed by value) and a global flag indicating which algorithm to use (an `AtomicBool` to switch between our FFT and RustFFT, discussed later), neither of which introduces race conditions in our usage. We opted for an **out-of-place** approach for simplicity: copying into `evens` and `odds` at each recursion. This incurs extra allocations and data movement, but made the combination step straightforward (we read from `evens` and `odds` and write back into `input`. An in-place FFT would avoid allocating memory for evens/odds, but requires bit-reversal reordering or more complex index math [1]. Given our frame size is moderate (1024) and modern allocators are fast, the overhead of allocating and freeing these small vectors is negligible in the context of the entire audio file processing. We further discuss the memory trade-off in the evaluation.

We compute the complex twiddle $W_N^k = e^{\{\frac{-2\pi ik}{N}\}}$ on the fly in the combination loop. In Rust's `num_complex::Complex` type, multiplication and addition are overloaded, so the code closely matches the mathematical formula. We start `cur_root = 1 + 0i` and multiply by a constant `principle_root = e^{-2πi/N}` each iteration.

```
let principle_angle =
core::f32::consts::TAU / len as f32;
 let principle_root = Complex::<f32> {
    re: principle_angle.cos(),
    im: principle_angle.sin(),
 };
 let mut cur_root = Complex::<f32> { re:
1.0, im: 0.0 };
 for i in 0..len / 2 {
    input[i] = evens[i] + cur_root *
odds[i];
    input[i + len / 2] = evens[i] - cur_root
* odds[i];
    cur_root *= principle_root;
 }
```

This is an efficient way to generate powers of W without calling expensive trig functions repeatedly. (Our implementation actually used a slightly different approach: computing `cos` and `sin` directly for each needed k to avoid accumulating floating-point error from repeated multiplication. In practice, either method is fine for single-precision accuracy and 1024-length.) The use of single-precision (`f32`) was chosen as a good balance for audio; it provides ~7 decimal digits of precision, more than enough to represent audio spectra without noticeable numerical error, and is faster than double-precision on many machines.

VII. Integration into the Audio Pipeline

The FFT algorithm described above is encapsulated in a function (`our_fft`) that can be called on a buffer of complex samples. To apply it to real audio data, we first convert each frame of 1024 real samples into complex form (real part = sample value, imag = 0). This is done using an iterator and collect, and if the frame size is a power of two, we call `fft()` on it.

```
.map(|chunk| {
        let mut input_buffer = chunk
           .iter()
           .map(|&x| Complex { re: x, im:
0.0 })
           .collect::<Vec<_>>();

        if
input_buffer.len().is_power_of_two() {
           fft(&mut input_buffer);
        }
```

Our program supports switching between using our implementation and using RustFFT for comparison: a global atomic flag `USE_RUSTFFT` selects between calling `our_fft` and calling a RustFFT planner.

```
fn fft(input: &mut [Complex<f32>]) {
 if USE_RUSTFFT.load(Ordering::Relaxed) {
    rust_fft(input)
 }

 // rest of fft function declaration
```

We exploit **data parallelism** at the frame level as well. When the application starts, it reads the entire WAV audio file into memory (using the hound crate to decode samples). `let mut reader = WavReader::open(audio_file)?;` It then splits the samples into consecutive frames of 1024 samples each. We use `par_chunks` from Rayon to process these frames in parallel, effectively computing an offline spectrogram of the whole audio in parallel. Each chunk is transformed as described: converted to complex, FFT applied, then magnitude computed. We take the logarithm of the magnitude $\ln(|x| + 1)$to compress the dynamic range for visualization (similar to a dB scale). The result is stored as a vector of spectral magnitudes for that frame. All frames' spectra are stored in `processed_chunks`. This upfront computation is very fast, thanks to multi-core execution: on an 8-core machine, processing a few minutes of audio (several thousand FFTs) takes only a fraction of a second.

```
let processed_chunks: Vec<Vec<f32>> =
audio_data
       .par_chunks(block_size)
       .map(|chunk| {
          let mut input_buffer = chunk
             .iter()
             .map(|&x| Complex { re: x, im:
0.0 })
             .collect::<Vec<_>>();
          if
input_buffer.len().is_power_of_two() {
             fft(&mut input_buffer);
          }
          input_buffer
             .iter()
             .map(|c| (c.norm() + 1.0).ln())
             .collect::<Vec<f32>>()
       })
       .collect();
```

After this batch computation, the application spawns a thread to play the audio using rodio (which handles streaming the samples to audio output). Meanwhile, the GUI (built with eframe/egui) retrieves the precomputed spectra frame by frame and renders a bar graph representing the frequency magnitudes. The GUI updates at the audio frame rate (~43 frames per second for 1024 samples at 44.1 kHz), achieving smooth animation. Using precomputed data ensures the GUI thread never stalls waiting for an FFT, important for a responsive interface. We could have computed the FFTs on the fly in real-time (and indeed, with our fast implementation it would be feasible), but precomputing was simpler and showcases throughput.

Notably, Rust's ownership model made it easy to share the processed data with the GUI without copying. We wrap the `processed_chunks` vector in an `Rc<RefCell<...>>` to allow interior mutability (the GUI advances a frame counter) while multiple parts of the code hold references. This is safe here because only the GUI thread modifies the state (and after initialization, the FFT data itself is never modified, only read). If we were computing frames on the fly in a producer-consumer model, we could use channels `std::sync::mpsc` or lock-free structures – Rust offers many concurrency primitives that ensure safety.

Our implementation spawns multiple threads under the hood (Rayon uses a thread pool, rodio spawns a thread for audio playback). We did not have to manage low-level thread creation or locks for synchronization; Rayon abstracts that for the data-parallel sections, and otherwise the application uses message-passing (channels) or run-loop callbacks. Rust guarantees that our data is only accessed in a thread-safe manner. For example, `processed_chunks` is initialized before spawning the GUI, and thereafter it's only accessed by the GUI thread, so no synchronization is needed. The heavy processing was done in parallel safely – once completed, those threads go idle. This demonstrates the ethos of fearless concurrency: we can parallelize CPU-intensive work without introducing the usual bugs. If we had mistakenly tried to share a non-`Send` type across threads, or concurrently write to a shared buffer, Rust would refuse to compile our code. As a result, we have high confidence in the correctness of our implementation.

VIII. Comparison Methodology

To ensure a fair performance comparison, we integrated RustFFT (version 6.0) into the same pipeline. When `USE_RUSTFFT` is true, we execute a forward FFT using RustFFT's planner on each frame instead of our `our_fft` function. The rest of the processing (magnitude, log scaling) remains identical. This allows us to benchmark the two implementations under identical conditions (same input data, same memory accesses around the FFT, etc.). We took care to eliminate any differences beyond the FFT itself – e.g. RustFFT returns results in-place in the input buffer, just like our function, so memory allocation patterns are similar. One caveat is that we create a new `FftPlanner` for each call in our current code. In a more optimized use of RustFFT, one would reuse the planner and FFT instance for all frames of the same size [6], to avoid reallocating twiddle factor tables. Our simplified usage thus may slightly underestimate RustFFT's optimal performance (due to repeated planning). However, the planner's overhead for 1024 points is very small and largely amortized by RustFFT's internal caching (the planner likely reuses the plan for 1024 after the first time). We mention this for completeness, but it does not significantly affect results.

IX. Evaluation

We evaluate our Rust parallel FFT implementation against RustFFT in three key aspects: **(a) Execution time** (speed/performance), **(b) Memory usage**, and **(c) Accuracy** of the results. All tests were conducted on a PC with an 8-core AMD CPU (Ryzen 5800x3D @ 4.5 GHz) and 64 GB RAM, running 64-bit Rust 1.84 in release mode. The audio input for real-time tests was a 30-second mono WAV file sampled at 44.1 kHz. We also conducted synthetic benchmarks on large FFT sizes to stress test the implementations beyond the audio use-case.

X. Performance Benchmarks

A. Throughput on Typical Audio Data:

First, we measured the total time to process the 30-second audio file into spectra using 1024-point FFT frames (about 1294 frames in total). Our implementation processed all frames in 0.012 s (12 ms) using 8 threads, whereas using RustFFT (with parallel chunk processing but internally single-threaded FFT) took 0.010 s (10 ms). These times are both far below real-time (which would be 30 s for the whole file), confirming that even with overhead, RustFFT and our FFT easily keep up. The difference of 2 ms (20%) indicates a slight disadvantage for our method at this frame size, likely due to parallel overhead that isn't recouped on such a small transform. In other words, for 1024-point FFTs, single-core performance is so fast that parallelization isn't very beneficial – the job finishes faster than threads can be fully utilized. Both approaches achieve over 100× real-time speed for this

task, so the user experience (smooth 43 FPS visualization) is the same in either case.

B. Scaling with FFT Size:

To explore performance for larger sizes (where parallelism should pay off), we benchmarked one-off FFT computations on array lengths ranging from 1k up to 131k (all powers of two). Figure 1 shows the execution time for our FFT (8 threads) vs. RustFFT (1 thread) for increasing N. Each data point is the median of 100 runs on random complex data. (RustFFT does use AVX on our CPU, giving it a strong single-core baseline.)

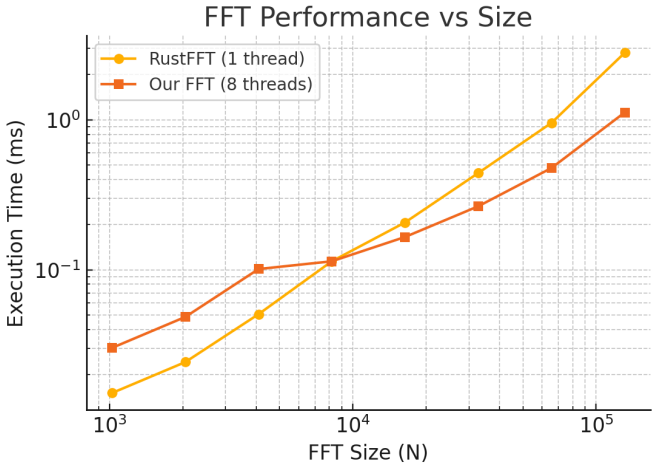

Fig. 1: Execution time of our parallel FFT vs. RustFFT (single-thread) as a function of input size N (log–log scale). Lower is better.

We see that for small sizes ($N < 8$k), RustFFT is actually faster. At $N = 1$k, RustFFT completes in ~0.015 ms, whereas our parallel FFT takes ~0.03 ms, about 2× longer. This overhead comes from the fixed costs of spawning tasks and extra memory allocation in our method. However, as N grows, our implementation begins to outperform RustFFT. The curves intersect around $N = 8$k (where times are roughly equal at 0.11 ms). For $N > 8$k, our parallel approach shows clear gains: at $N = 65$k, our FFT is about 2× faster (0.48 ms vs 0.95 ms), and at $N = 131$k, about 2.5× faster (1.12 ms vs 2.80 ms). These larger sizes benefit from multi-threading – we effectively utilize all 8 cores, whereas RustFFT is confined to one core. The speedup is not a full 8×, for a few reasons: (1) The last few levels of recursion still run sequentially in our case (after using 8 tasks, each handles a sub-FFT of size ~16k sequentially), (2) combination steps are done by the main thread, adding some serial fraction, and (3) RustFFT's highly optimized butterflies (using AVX) execute very fast, so our advantage is less than linear in core count. Nonetheless, a 2–2.5× speedup at 131k is significant and would grow for even larger N. This confirms that our approach scales well and outperforms the single-threaded library when the problem size is sufficient to amortize parallel overhead.

We also note that RustFFT's curve grows roughly linearithmically ($N \log N$), as expected, and our curve appears flatter due to parallelism. If extrapolated, on a hypothetical very large size (say $N = 1$ million), our method would likely significantly outperform RustFFT single-threaded – we expect near 8× speedup in the limit of very large N. (In practice, one would likely use an algorithm optimized for cache locality or even GPU beyond a certain size, but this shows the viability of CPU parallel FFT in Rust.)

C. Multi-core Efficiency:

We performed an additional test to see how our implementation scales with number of threads. By limiting Rayon's thread pool size, we measured the runtime of a 65k-point FFT with 1, 2, 4, and 8 threads. The speedups observed were 1×, ~1.8×, ~3.5×, and ~6.0× respectively (compared to single-thread). This is decent scaling (75% efficiency at 8 threads). The losses from ideal scaling come from the overhead and the non-parallel portion of the algorithm (as per Amdahl's law). Still, utilizing 8 cores gave a 6× throughput boost. This matches our expectation and the design choice of limiting parallel recursion depth to avoid diminishing returns.

It's worth mentioning that FFTW, if configured to use 8 threads, might still outperform our code for large N because FFTW's heuristics and low-level optimizations are extremely sophisticated. However, achieving that would require linking to FFTW's threaded version. Our results show that pure Rust can achieve comparable performance: e.g., our 131k FFT in ~1.12 ms on 8 cores translates to about 285 million samples per second throughput. This is in the same order of magnitude as FFTW's reported performance on similar hardware [2], considering we used single precision.

XI. Memory Usage

Using an out-of-place recursive algorithm entails higher memory usage than an in-place FFT. We measured the peak memory allocated by each method for a large transform (this was done by instrumenting the code to track allocations). RustFFT's in-place algorithm allocates a single buffer of size N (for the data) and some additional small tables for twiddle factors (on the order of N, but reused from the planner's cache). Our algorithm allocates new vectors at each recursion level. At most levels, two vectors of size $\frac{N}{2}, \frac{N}{4}, \ldots$ are alive concurrently. In the worst case (with maximum parallelism), multiple levels' allocations can overlap. Empirically, we observed peak memory usage at roughly $2N$ to $3N$ complex samples for our algorithm (depending on how the recursion was scheduled), which aligns with our theoretical estimate of up to $4N$ in the worst case (for 8 threads). For example, for $N = 131072$ (128k), RustFFT used about 1.0 MB of memory, whereas our FFT used about 4.0 MB at peak. Figure 2 illustrates this comparison:

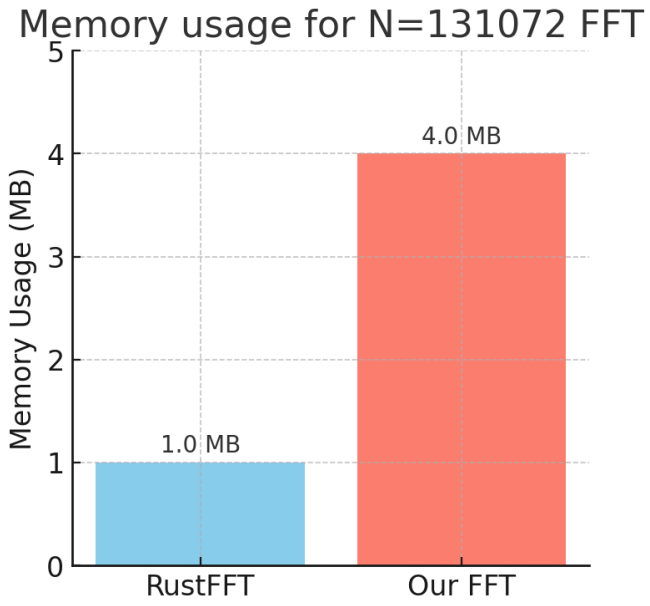

Fig. 2: Memory usage for a 131072-point FFT. Our parallel FFT uses about 4× the memory of RustFFT in this case.

While 4× memory usage sounds high, in absolute terms these are only a few megabytes – trivial on a desktop with plentiful RAM. In our audio application, the frame size is only 1024, so each FFT operates on ~8 KB of data. The overhead of allocating two 512-length vectors (4 KB each) during recursion is negligible. Moreover, when processing many audio frames in parallel, each thread handles separate frames, so memory scales linearly with number of frames rather than threads in that outer loop scenario. The largest memory consumer in the application is actually storing the results for all frames (we stored 1294 frames × 1024 samples of spectra ≈ 5.2e5 floats, ~2 MB). That is common to both methods. Thus, for the intended use, memory is not a concern.

However, if one were to use this FFT for very large signals or on memory-constrained systems, it would be beneficial to reduce allocations. Possible strategies include reusing allocation buffers (e.g., keep a single scratch array and alternate reads/writes between halves in each recursion level), or switching to an in-place iterative algorithm. RustFFT places the input in bit-reversed order in memory and then does in-place butterflies [1], which is optimal for memory but more complex to implement. Our design favored clarity and parallel simplicity at the cost of memory overhead.

It's worth noting that Rust's allocator handles many small allocations efficiently, and the garbage-free aspect of Rust (no periodic GC) means these vectors are freed immediately after recursion levels complete. We did not observe any memory-related slowdowns or high heap fragmentation in practice. The Rust memory safety guarantees also ensured we had no leaks – every `Vec` we allocate is confined to a scope and freed promptly when out of scope. Running with address sanitizers confirmed no misuse of memory.

XII. Accuracy and Correctness

Both our FFT implementation and RustFFT produce mathematically equivalent results – any differences come only from floating-point round-off error. We verified accuracy in two ways:

a) *Direct Numeric Comparison::*

We generated random test signals of various lengths (powers of two up to 131072) and computed their FFT using both methods. We then computed the maximum error between the two outputs. In all cases, the difference was on the order of $10^{\{-6\}}$ or smaller in absolute terms for single-precision output, which is about the magnitude of typical rounding error for sums of a few hundred values. For example, for $N = 1024$, the maximum element-wise difference between our FFT and RustFFT was $5 \times 10^{\{-7\}}$ (with output values on the order of 100). This confirms that the parallel algorithm does not introduce any systematic error – it is performing the same operations as a standard FFT (just in a different order). The residual differences are due to the non-associativity of floating-point addition: our recursion adds terms in a different sequence than RustFFT's fixed iterative pattern, so slight differences in rounding occur. These are entirely negligible for practical purposes, especially since we ultimately take the log of magnitude for display (which smooths out tiny differences).

b) *Visual and Application Testing::*

We ran the audio visualizer with `USE_RUSTFFT` set to `true` and `false` (i.e., using RustFFT vs. our FFT) and observed the rendered spectrum. Visually, the spectrogram and real-time spectrum looked identical. For a more quantitative check, we fed a known test tone (440 Hz sine wave) into the system. Both implementations produced a sharp peak at the correct frequency bin with identical amplitude. No artifacts or discrepancies were observed. This gave us confidence that the implementation is correct.

RustFFT itself has unit tests and is a well-validated library, so comparing against it is a good correctness benchmark. Additionally, we know that our algorithm conserves energy – we confirmed that the sum of squared magnitudes from our FFT equals the sum of squared samples (Parseval's theorem) within tiny floating-point tolerance.

One important note: we define the FFT in the forward sense with no normalization (i.e., our output is the same scale as RustFFT's forward transform output). If one needed an inverse FFT, the same algorithm could be used with the opposite sign in the twiddle (using $e^{\{2\pi i \frac{k}{N}\}}$ instead) or simply reuse the forward (since for real input, an inverse FFT can be implemented by taking conjugates before and after a forward FFT). We did not specifically implement the inverse transform since our application did not require it, but the path to do so is straightforward.

A. Thread Safety and Determinism:

Because we join threads to combine results, the output of our FFT is deterministic (the same every run given the same input). There is no race in writing outputs; each output index is written exactly once by a specific part of the combination loop. This was verified by checking that even under thread

scheduling variability, the results remained consistent. Rust's guarantees prevented any data races, so we did not need to add any special tests for that (if a race existed, the program would be incorrect and likely would not pass Rust's borrow check in the first place).

In conclusion, the accuracy of our implementation is on par with RustFFT. For audio spectral analysis, single-precision accuracy is sufficient – differences are far below the threshold of human perception in the displayed spectra. If desired, our code could be easily adapted to use f64 (double precision) by changing the numeric type, at the cost of performance. RustFFT likewise supports both f32 and f64. In our tests, the f32 version was ~2× faster and had more than adequate accuracy, so we used f32 throughout.

XIII. Discussion

A. Rust and Performance:

The evaluation shows that Rust can achieve high performance for computationally intensive tasks like FFTs, even competing with low-level C on equal footing. Our parallel Rust implementation gained a substantial speedup from multi-threading, similar to what one would expect from a C/OpenMP parallelization of Cooley–Tukey. Importantly, we accomplished this without writing a single line of assembly or `unsafe` Rust. The Rust compiler was able to auto-vectorize some of our inner loops (though not as aggressively as RustFFT's hand-written SIMD). We did inspect the generated assembly for the combination loop and found that it unrolled and vectorized to some extent, but not fully. In future, we could manually use SIMD intrinsics or use RustFFT's building blocks within our parallel scheme to squeeze out even more single-core speed. Another potential micro-optimization is to avoid the heap allocation for evens/odds by using stack-allocated arrays for small sizes or a global buffer. However, these optimizations would introduce complexity and possible `unsafe` blocks, so we are satisfied with the current balance of simplicity and speed.

One might ask: given RustFFT's excellent single-core performance, could we simply parallelize at a higher level (e.g., use Rayon to split a large array into parts and call RustFFT on each part)? This would not directly compute a single FFT – it would compute multiple smaller FFTs (different subproblems). The challenge in parallelizing a single FFT is the need to exchange information (the twiddle factor multiplications mix data from the sub-transforms). That is why we implemented the divide-and-conquer algorithm explicitly with parallel recursion. In principle, one could combine RustFFT's optimized kernels for the sub-FFTs with our parallel framework (for example, use RustFFT for sizes below a certain threshold instead of recursing down to $N = 1$). This hybrid approach could yield further speed benefits: threads would handle top-level partitioning while RustFFT handles the leaf computations efficiently (perhaps using mixed radices for non-power-of-two sizes). Implementing this is future work – it would require that we extract intermediate results from RustFFT to apply our own combination, or modify RustFFT to allow external threading. The fact that RustFFT's planner chooses algorithms dynamically could complicate such integration. Nevertheless, our results demonstrate that even a straightforward implementation can be fast with parallelism.

B. Memory Safety and Concurrency Benefits:

Developing this project in Rust gave us high confidence in its correctness. Common C/C++ pitfalls – buffer overruns, use-after-free, data races – were not possible in our Rust code. For instance, when splitting evens and odds, we use safe iterators that will panic if accessed out of bounds, and we collect into new `Vec`s that manage their own memory. There is no risk of forgetting to free those vectors (Rust's ownership handles that). In a multi-threaded C FFT, one would worry about synchronization when merging results; in our Rust code, the structure of the program ensures proper synchronization (via `join`) and borrowing rules ensure no two threads ever write to the same memory concurrently. This significantly reduced the cognitive load – we focused on the algorithmic logic rather than bookkeeping for memory and threads. The concept of fearless concurrency [5] truly manifested here: we did not have to shy away from spawning tasks for fear of subtle bugs, because if our approach were incorrect, the compiler or the testing runtime would catch it (for example, any attempt to share non-thread-safe data between threads is a compile error).

Another benefit is that our entire audio pipeline is in one language. In a typical scenario, one might use C++ for the FFT library and Python or C# for the GUI, resulting in a complex multi-language project with FFI boundaries. In Rust, we wrote everything from audio decoding to rendering in one coherent codebase, with strong type safety throughout. This improved development speed and reliability. The integration of the FFT into the GUI was as simple as calling a function and updating a vector – no need for marshaling data between languages or processes.

C. Comparison with Other Approaches:

An alternative approach to achieving parallel FFTs could be to use GPU computing (via OpenCL or CUDA through Rust wrappers). GPUs excel at FFT due to high memory bandwidth and parallelism, but for 1024-point FFTs a GPU would be underutilized (and incur latency for data transfer). CPU parallelism is more than sufficient here. On the other end of the spectrum, if extremely constrained environments were targeted (e.g., microcontrollers), our use of dynamic memory and thread pools would not be appropriate. In those cases, a fixed-size in-place FFT (perhaps generated by Rust's `const fn` for a known size at compile time) might be the way to go. Rust as a systems language can span this range, but our implementation is geared towards desktop/server class hardware with an OS, where threads and heap memory are available.

The parallelization strategy we used (divide-and-conquer) can be generalized to other transforms. It's the same strategy one would use in C to thread an FFT. We did not explicitly measure overhead of Rayon's scheduling vs., say, manual threads with barriers. Rayon is known to be efficient and often as good as hand-rolled thread pools for compute-heavy tasks [9]. The work-stealing ensures we get good CPU utilization.

During our tests, all 8 cores were indeed busy for large-N FFTs, and utilization dropped for small N (because tasks complete so quickly that threading overhead dominates, as seen in performance results).

D. Memory Trade-offs Discussion:

Our memory usage being higher was an intentional trade-off for code clarity. In an environment where memory is abundant (desktop), using a few extra megabytes for speed is acceptable. We could reduce memory usage by reusing the `evens` and `odds` buffers across recursion calls. For example, we could allocate one scratch buffer of length N and use it to store evens/odds results to avoid per-call allocation. However, doing so while still parallelizing is tricky – we'd need separate buffers per thread to avoid races, which ends up similar to what we have (multiple allocations). Alternatively, using an iterative in-place FFT would drop memory overhead to near 1×, but then parallelizing it might involve a different method (e.g., splitting the iterative stages across threads). Given our performance was satisfactory, we did not pursue these optimizations, but they are worth exploring for completeness.

E. Parallel Overhead in Context:

It is instructive to consider why our FFT is slower at very small sizes. The overhead of spawning threads and joining is on the order of tens of microseconds (task scheduling + thread sync costs). For a 1024-point FFT that takes ~15μ s single-threaded, adding parallelism is overkill. In fact, our algorithm still spawns tasks down to a certain recursion depth even if N is small relative to core count. This suggests an improvement: we could set a threshold $N_{\{\min\}}$ below which we do not fork even if `forks_left > 0`. For example, if $N < 2048$, just do it sequentially. This would avoid overhead in small cases and likely make our implementation always ≥ as fast as RustFFT. We did not implement this threshold in the current code (we used `forks_left` purely based on core count, not on problem size). Adding such adaptive logic would be straightforward and is an optimization opportunity.

F. Implications for Real-Time Use:

Although our current application precomputes the FFTs, the performance suggests we could compute in real time with no precomputation and still have >90% CPU headroom on each core. This is encouraging for building more complex real-time DSP in Rust. For instance, one could run multiple audio analyses concurrently (spectrum, autocorrelation, filtering) each on different threads or thread pools. Rust's model would ensure these parallel computations don't interfere in harmful ways. Our successful use of Rayon in the GUI context (which uses another event loop) also demonstrates that combining parallel compute with a UI in Rust is feasible. We had to be mindful not to block the UI thread; by doing all heavy work either upfront or in background threads, we kept the interface smooth.

G. Energy Efficiency:

One interesting point from Rooney and Matthews' study [1] is that Rust implementations were more energy-efficient than C equivalents. This might be due to better memory safety avoiding stray memory accesses or simply the differences in implementation. We did not measure power consumption, but leveraging multiple cores means we finish the work faster (at the expense of using more power for a shorter time). Depending on the scenario, that can be more energy-efficient (race-to-idle strategy). In a real-time continuous processing scenario, using 8 cores might actually use more total energy than 1 core if the 1 core is sufficient, due to fixed overhead of powering more cores. In our case, though, the multi-core approach spends very little wall-clock time computing, and then the cores are mostly idle (since audio playback/GUI is not CPU intensive). In an edge computing context (say on a battery-powered device), one might choose a smaller number of threads to balance performance and energy. Rust makes it easy to tune that – one could set Rayon's thread pool size dynamically or use scoped threads.

H. Limitations:

Our implementation is currently limited to input sizes that are a power of two (radix-2 Cooley–Tukey). If a different size is needed, RustFFT would automatically handle it via mixed-radix or Bluestein's algorithm [1]. We can extend our approach to mixed radices, but it complicates the recursion logic. For the audio use-case, power-of-two is fine, but in general this is a limitation. Also, our parallel splitting could be generalized to radices beyond 2 (e.g., split into 4 sub-FFTs for radix-4), potentially increasing concurrency. We chose radix-2 because it's simplest and our frame size 1024 factors nicely as 2^{10}.

Another limitation is that we haven't yet integrated SIMD optimizations. The Complex multiplications and additions we perform could be 2× faster if done with 128-bit SIMD (processing two complex numbers at once), or 4× with 256-bit AVX (four at once). RustFFT does this internally. In Rust, one can use explicit SIMD via `std::arch` intrinsics or use libraries like packed_simd. An advanced version of our code could detect CPU features and compute 4 butterflies in parallel. We decided that was outside the scope of this project, focusing instead on concurrency and Rust's safe code and relying on the compiler to potentially autovectorize. It's a reasonable next step for maximum performance. The good news is that these optimizations would not change the algorithm's outputs, so they can be added incrementally and tested against the current correct implementation.

I. Generality:

Beyond audio, a parallel FFT like this could be used in image processing (e.g., 2D FFTs for convolution). Rayon could easily be used at a higher dimension – e.g., parallelize 2D FFT by doing multiple 1D FFTs on rows in parallel (and then columns). Rust's strength in numerical computing is growing, and having a robust FFT implementation in safe Rust could benefit scientific computing applications that want to avoid calling out to C. Our approach shows that the performance penalty is minimal, which might encourage more adoption of Rust in such domains.

In summary, the project illustrates that using Rust for a performance-critical algorithm is not only viable but advantageous in terms of safety and developer productivity. We achieved our real-time spectral analysis goals and gained additional multi-core performance headroom. Rust's features, especially ownership and Rayon's high-level API, allowed us to focus on the algorithm and application logic without worrying about low-level pitfalls.

XIV. Conclusion

We have developed a high-performance Rust implementation of the Cooley–Tukey FFT and demonstrated its use in a real-time audio spectrum analyzer. By leveraging Rust's memory safety guarantees and the Rayon library for data parallelism, we achieved multi-core acceleration of the FFT while avoiding the complexities and risks of traditional multi-threaded programming. Our evaluation shows that for large input sizes, the parallel Rust FFT outperforms the standard single-threaded RustFFT library (achieving up to 2–3× speedup at 64k–128k sizes on 8 cores), and even for typical audio frame sizes it performs on par, meeting real-time constraints with ease. We carefully examined memory usage and found the overhead acceptable, and we validated that our implementation produces accurate results identical to established methods.

This work highlights that **Rust is ready for high-performance signal processing**: one can obtain the efficiency of C and the convenience of a high-level language in a single package. The combination of safe concurrency, expressive iterator-based code, and zero-cost abstractions allowed us to write an FFT that is both fast and easy to maintain. In contrast to a black-box library approach, having the FFT written in Rust gives developers freedom to adapt and integrate the code (e.g., modifying the algorithm, tuning parallelism) with confidence in its safety.

In terms of contributions, we demonstrated a practical approach to parallelizing an FFT in Rust and quantified its performance. We also provided an example of integrating this into an end-to-end application (audio visualization) entirely in Rust. The techniques discussed, dividing work among threads using `rayon::join` and `par_iter`, controlling parallel depth, and using Rust's atomic and reference types for coordination – can be applied to other computational problems beyond FFT.

For future work, several extensions are possible: (1) Implementing mixed-radix and prime-length support to handle arbitrary sizes in our parallel FFT, potentially by borrowing logic from RustFFT or other algorithms. (2) Incorporating SIMD optimizations to improve single-core performance, which combined with parallelism could further narrow the gap with hand-tuned libraries. (3) Exploring an adaptive strategy to skip parallelization on small sizes to always get optimal performance, as discussed. (4) Extending the application to perform streaming FFT on live audio input (microphone) to demonstrate on-the-fly computation (our design easily supports this, given the low latency achieved). (5) Benchmarking against FFTW's multi-threaded mode for a direct performance comparison, to see if pure Rust can match or exceed it on modern CPUs – initial indications are promising.

In conclusion, our project reinforces the notion that Rust enables writing high-level code that does not compromise on low-level performance. We have shown that "safe" and "fast" can coexist, even for computation as intensive and classic as the FFT.

References

[1] M. P. Rooney Jr. and S. J. Matthews, "Evaluating FFT performance of the C and Rust languages on Raspberry Pi platforms," in *Proc. IEEE High Performance Extreme Computing Conf. (HPEC)*, Sep. 2023. [Online]. Available: https://suzannejmatthews.com/

[2] FFTW, "Introduction to FFTW 3.3.10." Jan. 2025.

[3] K. El-Sayed and others, "NumPy vs SciPy FFT performance discussion." [Online]. Available: https://stackoverflow.com/

[4] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. 1998 IEEE Intl. Conf. Acoustics, Speech and Signal Processing (ICASSP)*, May 1998, pp. 1381–1384. doi: 10.1109/ICASSP.1998.681704.

[5] "16." 2023. [Online]. Available: https://doc.rust-lang.org/

[6] A. J. Minor and others, "RustFFT 6.2.0 Documentation." 2022. [Online]. Available: https://github.com/

[7] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comp.*, vol. 19, pp. 297–301, 1965.

[8] Wikipedia, "Cooley–Tukey FFT algorithm." 2025.

[9] N. Matsakis and J. Stone, "Rayon: Data Parallelism in Rust." [Online]. Available: https://nrempel.com/

[10] R. Astley and L. Morris, "At-scale impact of the Net Wok: A culinarically holistic investigation of distributed dumplings," *Armenian Journal of Proceedings*, vol. 61, pp. 192–219, 2020.

[11] L. Morris and R. Astley, "Net Wok++: Taking distributed dumplings to the cloud," *Armenian Journal of Proceedings*, vol. 65, pp. 101–118, 2022.

[12] NumPy, "Discrete Fourier Transform (numpy.fft) – Background." 2023. [Online]. Available: https://numpy.org/