Evaluating FFT performance of the C and Rust Languages on Raspberry Pi platforms

Michael P. Rooney Jr.
Department of Electrical Engineering & Computer Science
U.S. Military Academy
West Point, NY, USA
michael.rooney@westpoint.edu

Suzanne J. Matthews
Department of Electrical Engineering & Computer Science
U.S. Military Academy
West Point, NY, USA
suzanne.matthews@westpoint.edu

***Abstract*—The Fast Fourier Transform (FFT) is perhaps the most consequential algorithm for real-time applications for digital signals processing. Given the increased importance of securing devices on the edge, memory safety becomes an increasing concern for FFT applications. This work compares the performance of four FFT implementations written in the C and the Rust languages, benchmarked on the Raspberry Pi 4 and the Raspberry Pi Zero W platforms. Our results suggest that FFTs implemented in Rust are up to 45% more energy efficient than those written in C, and that Rust FFT implementations execute up to 37% faster than corresponding FFTs implemented in C. These results suggest that real-time application designers should take a closer look at the Rust language to enhance the safety and performance of their FFT applications.**

***Index Terms*—Edge computing, Rust, C, Fast Fourier Transform, Raspberry Pi**

I. Introduction

The Fast Fourier Transform (FFT) is one of the most consequential algorithms of all time, and plays a critical role in most digital signal processing applications. Due the broad applicability of the FFT, much effort [1], [2] is made in the software and hardware domains to improve the performance of FFTs, and to pick the optimal FFT for a particular application. Well-known FFT libraries such as FFTW aid application writers by automatically selecting the “best” FFT to use based on the input size, N, which is typically a power of 2.

In recent years, researchers have also sought to measure and optimize the *energy consumption* of their FFT approaches, especially in edge computing applications [3], [4]. In edge computing, FFT approaches are typically used as part of a larger analysis that occurs in real-time, which is achieved when the time to analyze data (t_a) is no more than the time required to acquire the data (t_{in}) [4]. Due to the potential sensitivity of certain edge computing applications, security is becoming an increasing concern in edge computing design [5], [6].

Applications designed for embedded devices at the edge commonly use C or C++ [7], due to the compact and efficient nature of produced executables. However, C is a notoriously memory-unsafe language, making it easy to inadvertently introduce vulnerabilities in applications [7]. In recent years, Rust has emerged as a memory-safe programming language that does not compromise execution speed [8]–[10]. As such, researchers have begun to include Rust when benchmarking the relative performance of languages; recent work has suggested that Rust achieves similar performance to C on many classes of problems [11].

Supported by the Department of Defense MIPR #H98230C0154252

This paper compares the performance of Rust and C-based implementations of the FFT on two separate Raspberry Pi platforms. The Raspberry Pi is an extremely popular single-board computer (SBC) used in edge and IoT applications involving FFT, either directly or as a testbed [3], [12]–[15]. Our work is novel for two reasons. First, we are the first to compare the performance of C-based and Rust-based implementations of the FFT. Second, we benchmark FFT performance on the high-end Raspberry Pi 4B, and the Raspberry Pi Zero W, the smallest and most energy-efficient of the Raspberry Pi SBCs. To the best of our knowledge, no prior study of FFTs on the Raspberry Pi have considered the Rust language or the Raspberry Pi Zero W. To this end, our work seeks to answer the following three research questions:

- RQ1: How performant are different FFT problem sizes on the Raspberry Pi 4B vs the Raspberry Pi Zero W?
- RQ2: How well do Rust implementations of FFT compare to C implementations of FFT?
- RQ3: What benefit (if any) is gained from implementing a custom FFT over using pre-existing library methods?

The remainder of this paper is organized as follows. Section II discusses related work. Section III describes our methodology and experimental setup. Section IV discusses the results, and we summarize our findings in Section V.

II. Background & Related Work

The Fast Fourier Transform (FFT) provides an $O(N \times \log N)$ solution to the traditional $O(N^2)$ Discrete Fourier Transform. Since the genesis of this new algorithm, there has been a sustained, multi-pronged effort to achieve further speed-up by analyzing the algorithm itself [2], the use of additional hardware [16] with some specially made to optimize the computation of the FFT [17], [18] and increasing the energy efficiency of the FFT [19].

The Fastest Fast Fourier Transform in the West (FFTW) is a highly optimized, widely-used FFT library that is considered the fastest publicly available FFT library. It uses the characteristics of a particular machine’s compiler to select the "best" FFT algorithm and implementation strategy (or "plan") for a particular problem size at runtime. Additionally, it is largely autogenerated using *genfft*, which is written in OCaml [2]. Significantly, it is the underlying FFT used by the commercial Matlab software [20] and the open-source GNU Octave suite [21].

FFTW is written in ANSI C, the most widely-used low-level programming language in use today. Applications written on the edge for embedded devices typically also use C or C++ [7]. We note that C is a notoriously unsafe language, lacking array bounds checking and is weakly-typed, making it harder to secure programs written in the C language. However C is still widely chosen for performant applications due to the speed of its compiled executables. It is therefore perhaps unsurprising that despite C's inherent security flaws, it remains a popular language for researchers utilizing the FFT.

More recently, the Rust Programming Language has gained prominence as a "safer" alternative to C, providing enhanced safety features such as efficient static information flow analysis and strong type checking, without compromising performance [8]. Adoption of Rust is still relatively low, owing to the relative newness of the language [9].

Rust cannot leverage FFTW as it lacks support for Single Instruction, Multiple Data (SIMD) instructions required by FFTW3 [22], [23]. As a result, Rust developers created the rustfft crate [24] in pure Rust. Unlike FFTW, rustfft needs to place the buffer in bit-reversed order. Like FFTW, the rustfft crate includes a planner that automatically chooses the best algorithm for a particular buffer and N size.

Recent work has suggested that Rust is capable of outperforming C in certain classes of applications. Pereira et. al [11], [25] ranked 27 programming languages by energy efficiency over 10 problem types; Rust emerged as the most energy efficient language in three, and performed similarly to C in the rest. Sitepu [26] compared the performance of a Rust and C implementation of the QUIC protocol in the cloud environment. In this application space, the C implementation outperformed the Rust implementation. Noureddine [27] compared the performance of three implementations of a Ray-casting algorithm in C, Python and Java on Raspberry Pi and Intel systems. It was discovered that C and Python outperformed Java on all three systems. Notably, Rust was not included in the set of languages under analysis. Georgiou et. al [28] compared Rust and C (among other language) implementations of common sorting algorithms on an Intel machine and a Raspberry Pi 3B; it was concluded that C was more energy-efficient than Rust. We note that this paper is several years old, and that Rust has undergone several optimizations in the intervening years. Furthermore, the performance of Rust and C on FFT implementations was not considered in any of the aforementioned benchmarks.

This paper specifically focuses on comparing the performance of FFT implementations in the C and Rust languages on Raspberry Pi platforms. The Raspberry Pi is an extremely popular single-board computer (SBC) used for embedded applications on the edge [3], [13]–[15]. As such, there have been several papers comparing FFT performance on the Raspberry Pi. He et. al [14] benchmarked the run-time of six FFT approaches on the Raspberry Pi 3B and the Raspberry Pi 2B+ implemented in C and Python. The FFTW implementations were the fastest serial implementations, though the authors note that parallel methods utilizing the GPU work best at higher N values [14]. Faerman et. al [15] looked at the efficacy of the Raspberry Pi 3B+ and Raspberry Pi 4B for the real-time processing of acoustic signals using different FFT algorithms, all implemented in C++. CPU FFT computation was performed using FFTW; the researchers concluded that for $N < 2^{18}$, both the CPU and GPU FFT implementations were capable of real-time performance. For larger N-sizes, GPU methods were preferable. Abtahi et. al [29] used the Raspberry Pi 3B to compare the performance of a number of FFT-based convolutional techniques on the Arm Cotex A53 processor (which is the processor used on the Raspberry Pi 3B). Notably, the researchers use GNU Octave for implementing their FFTs, which commonly uses FFTW under the hood.

Our work is *novel* for several reasons. To the best of our knowledge, we are the first to compare Rust and C implementations of the FFT on Raspberry Pi platforms. Second, we are also the first to include the Raspberry Pi Zero in our benchmarks. The Raspberry Pi Zero is the smallest Raspberry Pi, with 512MB of RAM and a 1 GHz ARM11 processor, consuming under a 2 watts of power under peak loads. The Raspberry Pi Zero W is the option of the Raspberry Pi Zero that has wireless connectivity; it has an identical processor to the Raspberry Pi Zero. The smaller processor of the Raspberry Pi Zero means that computations necessarily take longer than on other Raspberry Pi SBC models such as the Raspberry Pi 4B, the most recent model of the Raspberry Pi SBC. Unlike the Raspberry Pi Zero, the Raspberry Pi 4B consumes up to 6.4 Watts under peak loads, and boasts a more powerful 1.5 GHz A72 processor and 1 or more GB of RAM.

Lastly, our paper is one of the first to use the recently released PowerJoular [27] monitoring software for monitoring the energy consumption of our specific FFT processes. PowerJoular supports a variety of Raspberry Pi models, as well as PCs and servers using a RAPL supported Intel or AMD processor or an NVIDIA graphic card [30]. Unlike many other tools, PowerJoular is able to monitor the energy consumption of individual processes, allowing for more accurate energy estimates of specific programs.

III. Methods

The two specific Raspberry Pi platforms being investigated in this study are the Raspberry Pi Zero W Rev 1.1 (RPi ZW) and the Raspberry Pi 4B Rev 1.5 (RPi 4B) with 2 GB of RAM. The RPi ZW draws about 0.86 watts of power at idle and 1.75 watts under load. By contrast the Raspberry Pi 4B draws 2.4 watts at idle and 6.4 watts when under load [31].

Four implementations of the FFT were benchmarked over the RPi 4B and RPi ZW platforms. The first two FFT programs (`fftw` and `rustfft_crate`) are implemented using the standard FFTW (version 3.3.10) and rustfft (version

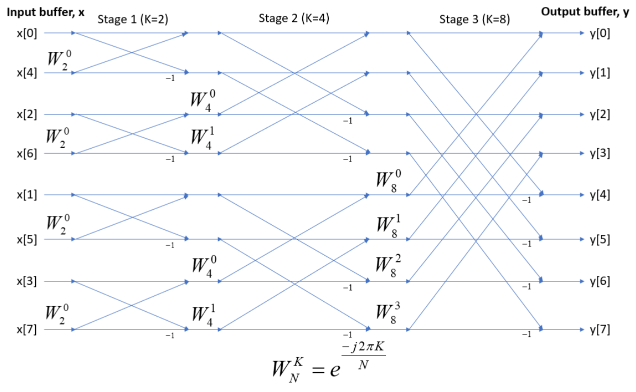


Fig. 1: Radix-2 8-point DIT FFT Butterfly Diagram

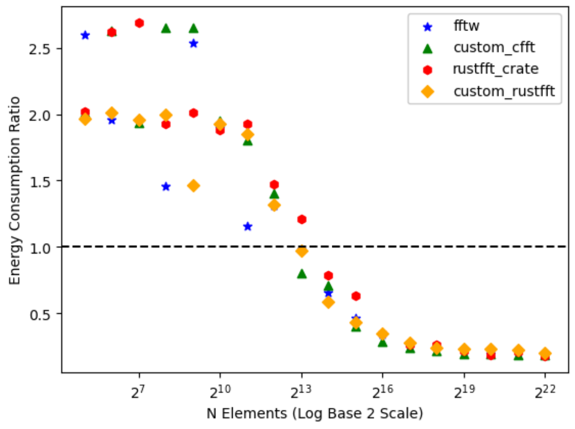


Fig. 2: Energy Consumption on Raspberry Pi 4B vs Raspberry Pi Zero W

6.1.0) libraries in C and Rust, respectively. The last two implementations (`custom_cfft` and `custom_rustfft`) are hand-implemented versions of the Radix-2 FFT algorithm (see Section III-A for more details) in C and Rust. For each implementation and platform, we measure the execution time, energy consumption, and the CPU and memory utilization.

All implementations were compiled using their respective languages optimization flags (`-O3` for C and `--release` for Rust) and were tested on N values ranging from 2^5 to 2^{22}. Additional work was required to create Rust binaries on the smaller RPi ZW, which cannot natively build Rust binaries. Instead, we use the Rust Cross-RS cross-compilation toolchain [32], maintained by the Rust Embedded Working Group Tools team, to build a 32-bit armv6 executable capable of running on the RPi ZW. Across the surveyed literature, an $N > 20$ was considered to be a large FFT requiring a lot of computational work; the largest N-value observed for a serial FFT implementation was $N = 2^{22}$ [14].

The input buffers are complex numbers, all having zero for their imaginary component at the beginning of execution. FFT execution time was measured using the `clock` function in the `time.h` header file in C, and the `Instant` function in the `std::time` module in Rust. Each N-value and implementation combination were run 5 times, and the average execution time is reported. Valgrind's `massif` tool was used to measure the peak memory utilized each program's execution.

A. *Fast Fourier Transform - Custom Implementations*

Both the FFTW and the rustfft libraries are capable of selecting a different FFT algorithm based on the nature of the input and a particular N-value. For simplicity, we chose to implement the Radix-2 algorithm for our custom FFT implementations (see Fig. 1). Prior work [4] suggests that the Radix-2 algorithm has a faster execution time than other approaches, though it may not be the most energy efficient for all N-values. The custom implementations are implemented identically in C and Rust, and use an iterative butterfly method.

Each custom implementation includes separate functions to bit reverse the input and generate the twiddle factors for each stage of the Radix-2 algorithm. When computing a given stage of the FFT, computed values are placed on a stack to preserve the buffer and then overwrite their respective values once computation for that section is complete. The implementations were made to compute complex-to-complex FFTs.

B. *PowerJoular*

PowerJoular was used to measure the energy consumption and CPU utilization of all implementations on the RPi 4B and RPi ZW. On Raspberry Pi models, PowerJoular uses custom-built polynomial regression models to make its power estimates [27]. PowerJoular is supported as-is on the RPi ZW and RPi 4B running a 32-bit OS [33]; however, a one-line change was made to Power Joular's `os_utils.adb` file to recognize the the latest revision number of the RPi 4B's 32-bit OS. We consulted with PowerJoular's author about the validity of this change, who confirmed that there should be no impact on accuracy.

Using the best practices laid out in prior work [27], each program computed the FFT on the given input data 30 times to gain a more accurate representation of the amount of power required to compute a single iteration of the FFT. After gaining the total energy over 30 iterations, that value was then divided by 30 to represent the amount of energy required for one FFT iteration. PowerJoular's `-f` flag was used to measure the CPU utilization of each implementation on a particular N-value.

IV. Results and Discussion

Fig. 2 shows the energy consumption ratio of the RPi 4B to the RPi ZW for each FFT implementation. A ratio greater than 1 at a particular N value indicates that the RPi ZW is a more energy-efficient platform for computing the FFT for that N. A ratio below 1 indicates when the RPi 4B is the more energy efficient platform. When the ratio is equal to 1, there is no difference in energy consumption between the two platforms. Our data suggests that $N = 2^{13}$ represents an inflection point at which the RPi 4B becomes a more energy-efficient platform

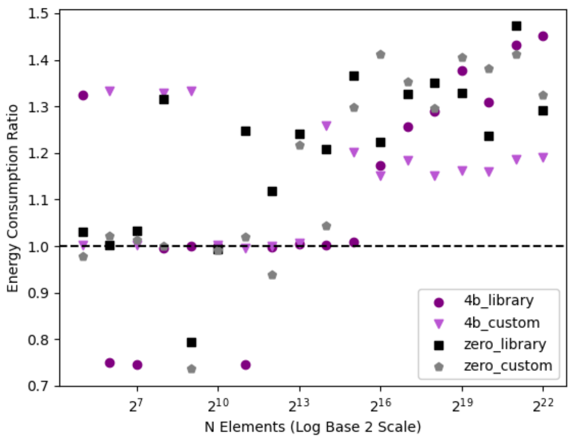


Fig. 3: Energy Consumption of C vs Rust

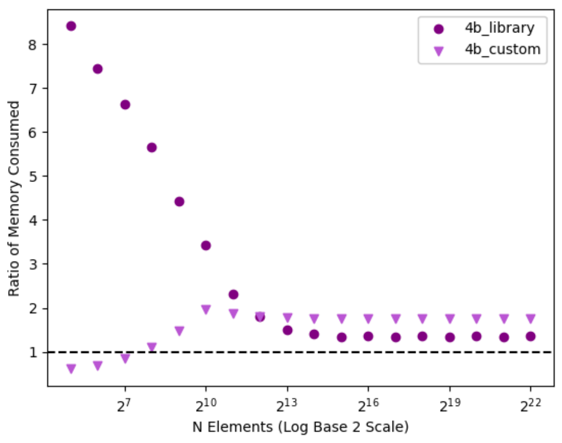


Fig. 4: Memory Consumption of C vs Rust

for FFT computation. We note that at $N = 2^{13}$, the RPi ZW's CPU utilization hits 100% across all approaches, and remains at 100% across all approaches for all higher N values. In contrast, the RPi 4B's CPU utilization never reaches 100%, suggesting its processor is never fully under load. Our results suggest that for all but the highest N sizes, the smaller RPi ZW is a better option from a purely energy perspective, consuming roughly 1.85 times less energy at N values of less than 2^{11}.

Fig. 3 shows the energy consumption ratio of each C language FFT implementation to the corresponding Rust language implementation on each platform. A ratio greater than 1 at a particular N indicates that the Rust language was more energy efficient at computing the FFT than the corresponding C language implementation for that N. A ratio less than 1 indicates that the C language implementation was more energy-efficient, while a ratio of 1 indicates no difference in energy efficiency. Regardless of whether a custom FFT or a library-selected FFT was employed, our results suggest that the Rust implementations are either typically equally or more energy-efficient than the C implementations on the Raspberry Pi platforms, with the Rust library implementations up to 45% more energy-efficiency at higher N values on the RPi 4B, and 41% more energy efficient on the RPi ZW.

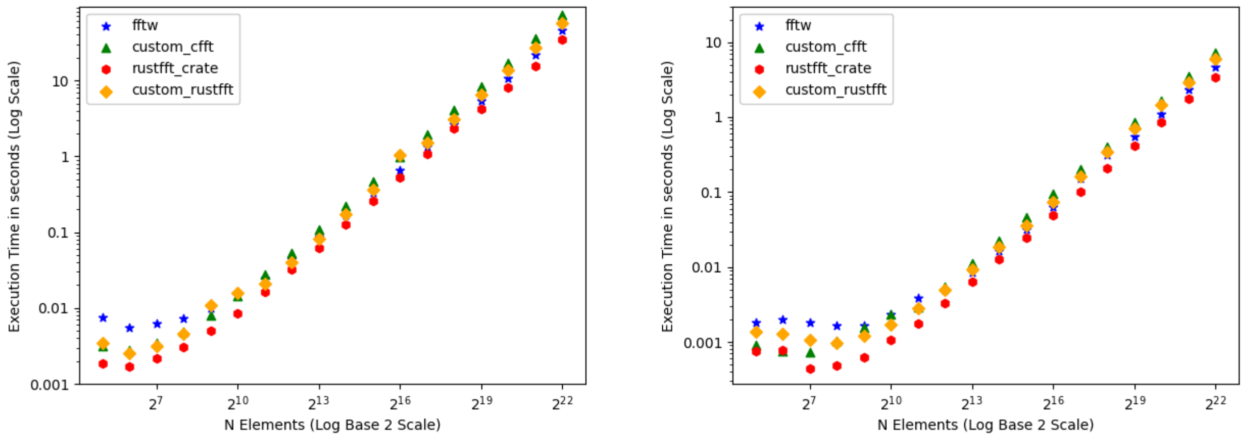
For real-time applications, executions speed is often tantamount. Fig. 5a and Fig. 5b depict the execution time of each FFT approach on the RPi ZW and the RPi 4B respectively. Unsurprisingly, FFT approaches take significantly longer on to execute on the weaker RPi ZW, executing up to 10.5 times slower than on the RPi 4B. Interestingly however, on virtually all N sizes, the `rustfft_crate` implementation had the fastest execution time, regardless of platform, executing up to 37% faster than the C implementations at the highest N values. These results suggest that the standard rustfft library is a better choice for implementing FFT approaches on the Raspberry Pi.

Table I shows the raw execution times of our different FFT approaches in seconds. We provide this table in hopes that it helps others determine if a particular platform is suitable for their analyses. For example, a sensor operating at a frequency of 60 HZ would report data every 16.67 ms. For problem sizes where $N < 2^{11}$, our data shows that a real-time computation of the FFT is possible on the RPi ZW, while consuming less energy than a similar computation on the RPi 4B. In contrast, for larger problem sizes between 2^{11} and 2^{13}, real-time computation of the FFT is feasible only on the RPi 4B.

Lastly, we compared the peak memory consumption of the Rust and C language implementations on our separate platforms. The maximum amount of memory utilized by any of the methods was 176 MiB, which is far below the memory capacity of both the RPi ZW and the RPi 4B. Each implementation used the same amount of memory whether run on the RPi 4B or RPi ZW; as such, only the memory usage on the RPi 4B is reported. Unsurprisingly, the custom versions of the algorithms consumed less overall memory than their library counterparts. Fig. 4 depicts the ratio of the measured memory consumption of each C language FFT implementation to the corresponding Rust language implementation. A ratio greater than 1 indicates that the Rust language implementation consumes less memory than the corresponding C implementaton, while a ratio of 1 indicates no difference. For all but the smallest N values, the Rust implementations was more memory-efficient than the C implementations, regardless of whether a custom FFT or a library-selected FFT was used. Most interestingly, 2^{12} became an inflection point at which both the `custom_rustfft` and the `rustfft_crate` implementations utilized roughly equal amounts of memory, consuming approximately 1.75 and 1.33 times less memory than the `custom_cfft` and `fftw` implementations respectively. Our results strongly support the notion that Rust implementations of the FFT are more memory-efficient than the C versions.

V. Conclusion

The Fast Fourier Transform plays a critical role in digital signals processing applications, with much work being done to improve the performance and energy consumption of the



(a) Raspberry Pi Zero W (b) Raspberry Pi 4B

Fig. 5: FFT execution time on Raspberry Pi platforms

TABLE I: Total Execution Time (s)

N	FFTW3		custom_cfft		rustfft_crate		custom_rustfft	
	RPi 4B	RPi ZW	RPi 4B	RPi ZW	RPi 4B	RPi ZW	RPi 4B	RPi ZW
5	0.0018	0.0075	0.0009	0.0031	0.0007	0.0018	0.0013	0.0035
6	0.0019	0.0056	0.0007	0.0027	0.0007	0.0016	0.0012	0.0025
7	0.0018	0.0062	0.0007	0.0034	0.0004	0.0021	0.001	0.0031
8	0.0016	0.0074	0.001	0.0048	0.0004	0.003	0.0009	0.0045
9	0.0016	0.0098	0.0015	0.0079	0.0006	0.005	0.0011	0.011
10	0.0023	0.0149	0.0023	0.0143	0.001	0.0084	0.0017	0.0158
11	0.0038	0.0243	0.0029	0.0275	0.0017	0.0161	0.0028	0.0207
12	0.0049	0.0441	0.0054	0.0538	0.0033	0.0323	0.0049	0.0408
13	0.0084	0.0865	0.0111	0.1098	0.0063	0.0624	0.0092	0.0822
14	0.0165	0.1666	0.0222	0.2209	0.0125	0.1269	0.0183	0.1723
15	0.0315	0.3315	0.0456	0.4716	0.0248	0.2611	0.0362	0.3599
16	0.064	0.6522	0.0948	0.9671	0.0496	0.5303	0.0737	1.0458
17	0.1503	1.3344	0.2031	1.9678	0.1021	1.0873	0.1602	1.5139
18	0.3097	2.667	0.4022	4.0993	0.2074	2.3514	0.3399	3.1034
19	0.5429	5.3709	0.8418	8.3462	0.4203	4.2411	0.6983	6.5872
20	1.0918	10.6836	1.6767	17.1514	0.8555	8.1077	1.4401	13.5054
21	2.3252	21.6724	3.5584	35.6269	1.7617	15.5024	2.9386	27.182
22	4.7176	45.7268	7.1744	73.88	3.4193	34.6656	6.0205	57.6972

underlying implementations. This paper compares the performance of Rust and C implementations of FFT implementations on the Raspberry Pi 4B and Raspberry Pi Zero W platforms, a popular class of SBCs for edge and IoT applications. We compare the implementations across language and platform, measuring the run time, energy consumption, memory consumption, and CPU utilization of our different approaches.

Our work explored three research questions. In regards to RQ1, our results suggest that on problem sizes less than $N = 2^{11}$, the Raspberry Pi Zero W is a more energy-efficient platform. One of the novelties of our work compared to prior work is that we include the Raspberry Pi Zero W in our benchmarking; these results suggest that perhaps scientists should take a closer look at the Raspberry Pi Zero (W) for FFT applications, especially for problem sizes less than $N = 2^{11}$. In regards to RQ2, our results strongly suggest that Rust-based FFT implementations are typically more energy-efficient, memory-efficient, and run faster than their C counterparts, suggesting that scientists should seriously explore the Rust language for their FFT needs. For RQ3, hand-written FFT implementations may be more useful if it is necessary to optimize for memory; unless someone writes a highly-optimized hand-written FFT for a particular N-size however, using a library implementation will typically yield as good (or better) performance.

We believe that our results are significant and important to the scientific community, as it supports the notion that, at least for the FFT applications on SBCs like the Raspberry Pi, scientists should consider porting C-based FFT applications to the Rust language. While Rust's single-ownership memory model makes it much more challenging language to approach than C [7], [8], we believe that the increased memory protections that Rust offers, coupled with its fast execution and low energy consumption, makes the Rust language difficult to ignore for FFT applications. Future work will concentrate on evaluating the performance of larger real-time applications implemented in Rust and to C, especially those that leverage the FFT.

Acknowledgment

Funding for this work was provided by the Department of Defense MIPR # H98230C0154252. We are extremely grateful to Dr. Adel Noureddine for providing feedback on using PowerJoular. The views expressed in this article are those of the authors and do not reflect the official policy or position of the Department of the Army, Department of Defense or the U.S. Government.

References

[1] P. Duhamel and M. Vetterli, "Fast fourier transforms: A tutorial review and a state of the art," *Signal Processing*, vol. 19, no. 4, pp. 259–299, 1990. [Online]. Available: https://www.sciencedirect.com/science/article/pii/016516849090158U

[2] M. Frigo and S. G. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[3] A. Chakraborty, U. Gupta, and S. R. Das, "Benchmarking resource usage for spectrum sensing on commodity mobile devices," in *Proceedings of the 3rd Workshop on Hot Topics in Wireless*, ser. HotWireless '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 7–11. [Online]. Available: https://doi.org/10.1145/2980115.2980129

[4] K. Adámek, J. Novotný, J. Thiyagalingam, and W. Armour, "Efficiency near the edge: Increasing the energy efficiency of ffts on gpus for real-time edge computing," *IEEE Access*, vol. 9, pp. 18 167–18 182, 2021.

[5] M. Caprolu, R. Di Pietro, F. Lombardi, and S. Raponi, "Edge computing perspectives: Architectures, technologies, and open security issues," in *2019 IEEE International Conference on Edge Computing (EDGE)*, 2019, pp. 116–123.

[6] Y. Xiao, Y. Jia, C. Liu, X. Cheng, J. Yu, and W. Lv, "Edge computing security: State of the art and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1608–1631, 2019.

[7] M. Noseda, F. Frei, A. Rüst, and S. Künzli, "Rust for secure iot applications: why c is getting rusty," in *Embedded World Conference 2022, Nuremberg, 21-23 June 2022*. WEKA, 2022.

[8] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk, "System programming in rust: Beyond safety," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017, pp. 156–161.

[9] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and drawbacks of adopting a secure programming language: Rust as a case study," in *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. USENIX Association, Aug. 2021, pp. 597–616. [Online]. Available: https://www.usenix.org/conference/soups2021/presentation/fulton

[10] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating c to safer rust," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–29, 2021.

[11] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Ranking programming languages by energy efficiency," *Science of Computer Programming*, vol. 205, p. 102609, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642321000022

[12] J. D. Brock, R. F. Bruce, and M. E. Cameron, "Changing the world with a raspberry pi," *Journal of Computing Sciences in Colleges*, vol. 29, no. 2, pp. 151–153, 2013.

[13] A. Buzachis, A. Galletta, A. Celesti, M. Fazio, and M. Villari, "Development of a smart metering microservice based on fast fourier transform (fft) for edge/internet of things environments," in *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, 2019, pp. 1–6.

[14] Q. He, V. Weaver, and B. Segee, "Comparing power and energy usage for scientific calculation with and without gpu acceleration on a raspberry pi model b+ and 3b," pp. 3–9, 2018, copyright - Copyright The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp) 2018; Last updated - 2020-04-16. [Online]. Available: https://login.usmalibrary.idm.oclc.org/login

[15] V. Faerman, V. Avramchuk, K. Voevodin, and M. Shvetsov, "Real-time correlation processing of vibroacoustic signals on single board raspberry pi computers with hifiberry cards," in *High-Performance Computing Systems and Technologies in Scientific Research, Automation of Control and Production*, V. Jordan, I. Tarasov, and V. Faerman, Eds. Cham: Springer International Publishing, 2022, pp. 55–71.

[16] D. Efnusheva and A. Tentov, "Integrating processing in ram memory and its application to high speed fft computation," in *Proc. International Conference on Information Society and Technology, Serbia*, 2014.

[17] S. Lee, D. bai Kim, and S.-C. Park, "Power-efficient design of memory-based fft processor with new addressing scheme," in *IEEE International Symposium on Communications and Information Technology, 2004. ISCIT 2004.*, vol. 2, Oct 2004, pp. 678–681 vol.2.

[18] X. Chen, Y. Lei, Z. Lu, and S. Chen, "A variable-size fft hardware accelerator based on matrix transposition," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 10, pp. 1953–1966, 2018.

[19] M. A. Jaber, D. Massicotte, R. A. Jaber, and K. Nesmith, "An efficient data parallelization of the radix-23 (carbon) fft on gpu/cpu," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019, pp. 1–5.

[20] Mathworks, "Fast fourier transform - MATLAB fft," https://www.mathworks.com/help/matlab/ref/fft.html, accessed: 2022-12-6.

[21] The Octave Project Developers, "Gnu octave ch. 31: Signal processing," https://docs.octave.org/v7.3.0/Signal-Processing.html, accessed: 2022-12-6.

[22] L. Brenna, I. S. Singh, H. D. Johansen, and D. Johansen, "Tfhe-rs: A library for safe and secure remote computing using fully homomorphic encryption and trusted execution environments," *Array*, vol. 13, p. 100118, 2022.

[23] M. Frigo and S. G. Johnson, "3.1 SIMD alignment and fftw_malloc," http://www.fftw.org/doc/SIMD-alignment-and-fftw_005fmalloc.html, accessed: 2022-10-25.

[24] E. Mahler, "Crate rustfft," https://docs.rs/rustfft/latest/rustfft/, accessed: 2022-11-1.

[25] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. a. P. Fernandes, and J. a. Saraiva, "Energy efficiency across programming languages: How do energy, time, and memory relate?" in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 256–267. [Online]. Available: https://doi.org/10.1145/3136014.3136031

[26] S. F. A. Vedaalana, "Performance evaluation of various quic implementation: Performance and sustainability of quic implementations on the cloud," Master's thesis, University of Lorraine, France, 2022.

[27] A. Noureddine, "Powerjoular and joularjx: Multi-platform software power monitoring tools," in *18th International Conference on Intelligent Environments*, 2022.

[28] S. Georgiou, M. Kechagia, and D. Spinellis, "Analyzing programming languages' energy consumption: An empirical study," in *Proceedings of the 21st Pan-Hellenic Conference on Informatics*, ser. PCI 2017. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3139367.3139418

[29] T. Abtahi, C. Shea, A. Kulkarni, and T. Mohsenin, "Accelerating convolutional neural network with fft on embedded hardware," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 9, pp. 1737–1749, 2018.

[30] A. Noureddine, "Powerjoular," https://gitlab.com/joular/powerjoular, 2022, accessed: 2022-11-1.

[31] J. Geerling, "Power consumption benchmarks," http://www.pidramble.com/wiki/benchmarks/power-consumption.

[32] Rust Embedded Working Group Tools team, "Cross-rs/cross: "zero setup" cross compilation and "cross testing" of rust crates," https://github.com/cross-rs/cross, accessed: 2022-12-6.

[33] A. Noureddine, "Powerjoular," https://www.noureddine.org/research/joular/powerjoular, 2022, accessed: 2022-11-30.