Investigating Digital Signal Processing Libraries and Optimization Techniques for Windows on Arm Computers

Sami Saade
Queen Mary University of London
London, United Kingdom

Jean-Baptiste Rolland
Steinberg Media Technologies GmbH
Hamburg, Germany

George Fazekas
Queen Mary University of London
London, United Kingdom

***Abstract*—Since 2020, the personal computing scene was taken by storm as Apple had shifted its flagship MacBooks from Intel to Arm. Apple succeeded in their Arm integration as their MacBooks became more efficient and more responsive because of Arm's RISC based architecture. Now, we see other big tech manufacturers, such as Microsoft, looking to implement Arm in their new personal computers. With Windows on Arm still in its preliminary stages, audio software companies like Steinberg are investigating the architecture to establish the most performant techniques and libraries to use in their products and recommend to their clients. This motivates the task to evaluate different open-source digital signal processing libraries and analyze their implementations to form a better understanding on optimizing digital signal processing algorithms for the Armv8 instruction set. Thus, a micro-benchmark is created to rank the performance speeds of the Fast Fourier Transform (FFT) and biquadratic filter libraries; two widely used digital signal processing algorithms in music software and digital audio processing. The project focuses on Windows on Arm running and adapting the benchmark on a Windows Developer Kit computer with a Qualcomm Snapdragon® Gen 3 chip. The results show that vectorization and single instruction multiple data (SIMD) architecture through Arm's Neon extension significantly improve FFT applications. Furthermore, the study demonstrates the need for optimization on Windows on Arm to reach state-of-the-art digital signal processing libraries.**

***Index Terms*—benchmarking, digital signal processing, Armv8-A, Arm architecture, NEON, optimization, digital filters, Windows on Arm**

I. Introduction

Digital audio software such as digital audio workstations (DAWs), virtual instruments, and digital audio effects have made the world of music production, performance, and composition accessible to everyone with a personal computer and laptop (Bell 2018). Audio software is built using digital signal processing libraries that offer different algorithms that analyze or manipulate audio data. Important algorithms include Discrete Fourier Transforms typically used to analyze the frequency content of a signal and store it. Other important algorithms are IIR and FIR digital filters that manipulate the audio signal data attenuating and boosting certain frequencies (Mulgrew, Grant, and Thompson 2002). These algorithms are used to design most digital audio effects such as pitch shifting (Laroche and Dolson 1999), equalizers (Reiss 2010), and more. However, for these libraries to be successfully used in a musical context, they must be fast and efficient to avoid any lags in the sound output, especially if used in a real-time performance or recording setting. The performance of these libraries highly depends on their implementation into a computer program and the central processing units (CPU) architecture and speed of the computers running them.

Initially, the most popular CPU architecture for personal computers was x86 and x64 based on the Complex Instruction Set Computer (CISC). In CISC, instructions can perform multiple low-level arithmetic and memory operations which allow complex tasks to be performed by a minimal number of instructions. Another computer architecture is the Arm architecture based on the Reduced Instruction Set Computer (RISC) architecture. Arm uses simple finite-length instructions that can perform only one operation quickly and efficiently (Rahman, Khan, and Zaman 2024). While unpopular until 2019, the implementation of Arm in personal computers is not novel. In fact, the modern company Arm Ltd. is the offspring of Acorn Computers; a company established in 1976 aiming to provide cost-effective computers across the education sector in the UK. Due to monetary issues and the unpopularity of the Acorn computers, the company pivoted to selling its processor architecture to mobile manufacturers as Advanced RISC machine (Arm) from the year 1993. Arm then continued to dominate these markets expanding into banking, microcontrollers, and more. While companies like Microsoft still attempted to release Arm powered personal computers in 2012, none of the attempts were successful due to machine's poor performance and the lack of compatibility of software for Arm. However, this has changed in 2020 when Apple completely shifted their CPUs in their flagship MacBooks from Intel to Arm achieving better performance and power efficiency as a result (Gupta and Sharma 2021). As of 2023, Microsoft has reframed their focus on their Arm powered devices as they released their Microsoft Dev Kits equipped with Qualcomm's Snapdragon® 8cx Gen 3 compute platform chips which is used in this project. Microsoft also released their version of Windows on Arm which included proper emulation, an ARM64 compiler, SIMD Neon support, and other native Arm tools. With a new type of personal computer becoming more and more popular, it is important to investigate the implications of the different architectures on

existing software.

To start with, while it is thought that Arm is more power efficient than x86_64, and that x86_64 is better at complex applications than Arm, this hypothesis has been debunked by Blem, Menon, and Sankaralingam (2013) stating that performance and power efficiency are significantly governed by the microarchitecture design of the task instead. This raises the need to optimize existing libraries specifically to the architecture itself. This is also important as libraries that are designed to compile on Intel cannot run on Arm machines without X86_64 virtualization, which greatly affects the performance of these libraries on Arm computers (Dall et al. 2016). Thus, many popular digital signal processing libraries were updated to have native Arm support. However, few benchmarks of these libraries running on the computer Arm processors exist, especially on the Qualcomm processors used by Microsoft. Furthermore, with the early stages of Windows on Arm, not many libraries are compatible on the operating system. Therefore, and due to a significant share of users shifting to Arm based computers, audio software companies like Steinberg are interested in finding the best performing digital signal process libraries optimized for Arm computers. Moreover, investigating optimization best practices for Arm is still needed as the libraries would operate on two different operating systems - Windows on Arm and macOS - and on two main CPU types - Qualcomm Compute Platform and Apple Silicon. To achieve this, we have built a benchmark to test the different digital signal processing libraries focusing on Fast Fourier Transforms (FFTs) and biquadratic filter libraries. Specifically, the effects of different algorithm implementations and data structure usage were evaluated using a micro-benchmark framework. The micro-benchmarking framework was developed to evaluate the computation speeds of FFT and biquadratic filter applications to audio data of different buffer sizes. The micro-benchmark was built to be cross-compatible across different operating systems. This paper focuses on the Windows on Arm implementation, while another study by Steinberg intern, Blaise Hecquet (2024), focuses on the Apple implementation. The results are then ranked and compared to the ones obtained on the Apple machine.

II. Background

Optimizing digital signal processing for libraries for better performance could be achieved from a high-level and a low-level approach. In the high-level optimization layer, the algorithm itself and its code implementation are optimized to reduce its complexity. In the low-level optimization layer, intrinsic functions can be used to specify compiler level instructions that would help avoid memory overheads and improve arithmetic operation speeds. The following sections will explore the optimization techniques used for FFTs and biquadratic filters.

A. The Fast Fourier Transform

The first algorithm of interest is the Discrete Fourier Transform. The algorithm decomposes a discrete signal into sinusoidal components of various frequencies bins. In music, this technique is used to build visualizers or as an analysis step for more complex applications such as phase vocoders. The algorithm can be defined as:

$$X(k) = \sum_{n=0}^{N-1} x(n) exp(\frac{-jnk2\pi}{N})$$

X(k) are the frequency bin amplitudes, x(n) is the discrete signal input, and N is the length of the input signal. This original algorithm has the complexity of $O(n^2)$ as it must calculate over a matrix of size $(n * n)$ (Mulgrew, Grant, and Thompson 2002). This is very computationally demanding and is exponentially slower for larger buffer sizes. To surpass these limitations, many techniques have been used, the most popular being the Cooley-Tukey algorithm. Cooley and Tukey (1965) were able to reduce the complexity of the DFT from $O(n^2)$ to $O(N * log(N))$ by partitioning the DFT into two half-sized DFTs and butterfly operations. There are many different Fast Fourier Transforms (FFT) algorithms such as the Pease algorithm, the Transposed Stockham algorithm, and the Gentlemen-Sade algorithm. To optimize the algorithm further, Swarztrauber (1982) showed that, for buffers of size $N = 2^m$, the Cooley-Tukey algorithm and the Gentlemen-Sade algorithm could be vectorized in place or out of place without additional storage needs. This is ideal for digital audio and DAWs as the standard buffers are of size $N = 2^m$. This means that these FFT algorithms could be split and computed using parallel computation techniques without adding any memory operations.

B. The Biquadratic Filter

The second digital signal processing algorithm of interest is the Biquadratic filter. Biquadratic filters are 2nd order Infinite Impulse Response (IIR) filters with two zeros and two poles. They are also known as recursive functions due to their dependency on previous outputs. They can be represented by the following direct formula:

$$y[n] = b_0x[n] + b_2x[n-1] - a_1y[n-1] - a_2y[n-2]$$

Biquadic filters attenuate or boost a signal at a certain frequency range. They could be used in applications as simple as an equalizer (Reiss 2010) to a bank of biquadratic filters modeling audio distortion of popular guitar pedals (Nercessian, Sarroff, and Werner 2021).

C. Compiler Level Digital Signal Processing Optimization

On the compiler level, techniques to improve the code implementation of FFTs and Biquadratic filters were suggested such as loop unrolling which results in less overhead generated by loops with a small number of operations. These loops are turned into straight line code that can be vectorized and used in parallel computations. While recursive filters are inherently dependent on loops, these vectorizing techniques are still efficient to be implemented in biquadratic filter applications (Kutil 2009). Parallel computing can be performed in different ways. Chaurasia et al. (2015) observed significantly

faster performance when performing recursive filter tasks on graphics processing units using Nvidia CUDA and the Graphics Processing Unit (GPU). Using the CPU, a way to parallelize computations is through single instruction multiple data architectures (SIMD). SIMD is a microarchitecture that allows a single instruction to process multiple data vectors. SIMD is generally successful and exploited in multimedia digital signal processing applications (Nguyen and John 1999). SIMD is accessible through assembly code or in C/C++ using intrinsic extensions such as Intel's SSE and Armv8's Neon.

D. Automatically Tuned Libraries

With the efficiency of auto-vectorization in FFTs, a search algorithm was designed to automatically rank multiple FFT implementations (J. Johnson et al. 2000). This study evolved as the search algorithm was included in a specialized compiler that translated digital signal processing algorithms from FORTRAN to C as part of the SPIRAL library (Xiong et al. 2001). The SPIRAL library, alongside others such as FFTW and ATLAS, used dynamic programming to create automatic performance tuning systems that maintain peak performance without depending on any processor architecture. The recent version of FFTW beat many of the vendor-tuned libraries while being on par with Intel's IPP – the fastest FFT library on intel processors (Frigo and S. G. Johnson 2005). While the automatically tuned libraries have seen success, Orozco et al. (2007) states that these libraries focus on top layer optimization techniques while better performance was achieved by optimizing kernel microarchitecture as well. This highlights the method of implementing machine-specific intrinsics and the nuance between hand-tuned and auto-tuned digital signal processing libraries.

E. Optimizations for Armv8 and Current Benchmarks

Neon is Arm's advanced SIMD architecture and is included in the Armv8-A architecture. Neon SIMD has 32 128-bit vector registers that could be used to process multiple data streams each. Being from the RISC architecture, Neon SIMD is different than Intel's SSE by having more compact implementations and less complex intrinsics. With the recent adoption of this technology in personal computers, not many libraries have ported to support Neon. The use of Neon intrinsics looks promising as it was shown to increase FFT performance than the typical FFTW performance on a Linux-based machine (Du and Huang 2020). With a plethora of benchmarks and support from state-of-the art digital signal processing libraries on Intel, a better understanding of parameters and optimization techniques is achieved, shedding light on limitations and best practices in computing (Ayala et al. 2021). Thus, this paper aims to fill the gap in performance data of digital signal processing algorithms performance on Windows on Arm machines to enable the research for computing best practices on Arm.

III. Methodology

A gray-box approach is adopted where we run different libraries on the Windows on Arm computer and obtain performance metrics. We then compare these results and analyze the best performing algorithms to understand which optimization approaches are more effective for the Armv8 architecture and the Windows on Arm kernel.

A. Micro-Benchmark for Open-Source Digital Signal Processing Libraries on Arm

To achieve the first part of the task, we used micro-benchmarking as a performance evaluation strategy. Micro-benchmarking is a common technique used to evaluate CPU performance of a specific isolated task. Using Google's Benchmarking library, we can run microbenchmarks that assess the speed of execution of the tasks in the designated benchmarking loop. The loop then repeats until a stable CPU time has been reached. In a practical setting, the initialization of the functions would happen as soon as the VST is loaded into a DAW or as soon as the program starts running. Thus, initializations are executed before the benchmarking loop to isolate the algorithm's performance metrics. The algorithms are applied to similar audio data buffers spanning from 4 samples to 16384 samples in length. The libraries that were chosen were all free and open-source C++ libraries apart from JUCE; a commercial library for VST development available for free under non-commercial use. FFT results are also converted to MFLOPS with $MFLOPS = 5 * N * log(N)/time(microseconds)$. This is suggested in the implementation of FFTW (Frigo and S. G. Johnson 2005) and is provided for a better comparison metric between processors and operating systems.

The implementation template of our benchmarking functions are as follows:

Algorithm 1 Benchmark implementation for FFTs and biquadraticLibraries in C++

```
template T //to set the precision type
void bm_library(benchmark::State& state){
 initialize-fft or biquad
 initialize-audio vector for inputs
 initialize-output vector with length
    depending on benchmark state variable
 populate-fill audio vector
    with complex data

 //this is the benchmark loop
 while (state.KeepRunning()) {
    perform the fft or biquad operation
    benchmark::DoNotOptimize(out);
 }
}
BENCHMARK(bm_jucefft)->ALLARG;
```

B. Project Structure using CMake

Using CMake, we manage to implement the libraries in one program by linking their dependencies. This allows us to generate a build file for our project according to the

source code's structure and organization. CMake also allows us to make the benchmark cross-compatible across macOS and Windows on Arm. This enables us to compare the performance of the libraries across the two different operating systems and processors. We then create headers for each library containing the benchmark functions. Finally, we can call one benchmark that runs all these functions within main.cpp. This is indicated by the CMakeLists.txt file. We have used FetchContent to fetch the libraries from their online repositories where applicable. In addition, we have set flags using the option command to indicate which libraries should be linked when creating the build file. This allows us to control which libraries are on or off by adding the -DCMAKE_ENABLE_libraryName=OFF/ON when generating the build files with CMake. This project structure allows further benchmarking of more Arm supported digital signal processing libraries on multiple Arm based computers and operating systems.

2 Benchmark Project Structure

```
|-- fft-biquad-benchmark/
| |-- Include/
| | |-- bm_{library_name}.h*
| |-- src/
| | |-- main.cpp
| |-- CMakeLists.txt
```

C. Library Compatibility for Windows on Arm

As mentioned, support for Windows on Arm software is still a work in progress. Thus, throughout the project, the task to make libraries compatible with windows was a necessity in order to gather enough data. The most prominent issue encountered during the task is the instability of the native Clang compiler on the Windows Dev Kit. This meant that most libraries that required Clang could not be tested on the computer yet. Thus, some of the majorly used libraries are still to be supported on WoA such as FFTW3, FFMPEG, KFR, and GSL. Another complication was that most libraries are not up to date with Windows or Windows on Arm compatibility. They ignore the appropriate C library headers required for Windows Visual Studio. Also, some libraries overlooked Windows CMake guidelines making the libraries incompatible with Visual Studio's CMake tools. To circumvent that, CMake Windows ARM64 was installed and used through the command prompt to generate the builds for the project. With these implementations, we successfully fixed 4 libraries to be compatible with Windows on Arm. Table I describes the compatibility of each library and their fixes.

D. Complex Data Forms

FFTs and biquadratic filter operations depend on complex data vectors. The same vector of complex numbers could be stored in two different array structures: Interleaved or Separated. Interleaved arrays have real and imaginary components coinciding with each other while the split array form has the real components stored in the beginning of the array and the complex numbers stored after that. This means that for interleaved complex number arrays, only one pointer is needed to the real part of a number and the complex number can be understood to be the one in the next memory location. Meanwhile, for the split complex number format, two pointers are used to specify the real and complex part of a number. A list of the tested libraries and their complex data forms are listed in table II.

TABLE I
FFT AND BIQUADRATICLIBRARY SATUS ON WINDOWS ON ARM

library	Status	Fix
pffft	stable	Added support for $< math.h >$ Added proper $\_M\_Arm64$ preprocessor macros to target AMR64 for Windows on Arm
SST	stable	Added support for $< math.h >$ Replaced $aligned\_alloc()$ from $< stdlib.h >$ to support windows with $\_align\_malloc()$
Pocketfft	stable	Added support for $< math.h >$ Added CMake support for Windows on Arm
SND	stable	Added support for $< math.h >$
KFR	Unstable	Requires Clang Compiler
vDSP	Incompatible	Apple exclusive library
FFMPEG	Unstable	Requires Clang Compiler
KissFFT	Unstable	Requires Clang Compiler
GSL	Unstable	Requires Clang Compiler
MeowFFT	stable	supports Windows on Arm
muFFT	stable	supports Windows on Arm
Q_library	stable	supports Windows on Arm
Juce	stable	supports Windows on Arm
Cmsis_dsp	stable	supports Windows on Arm
IIR1	stable	supports Windows on Arm
Bela	stable	supports Windows on Arm
Maximilian	stable	supports Windows on Arm
Moog	stable	supports Windows on Arm

TABLE II
COMPLEX DATA STRUCTURES OF DIFFERENT DIGITAL SIGNAL PROCESSING LIBRARIES

Library	Algorithm	Precision	Structure
pffft	FFT	float	interleaved
pocketFFT	FFT	float, double	interleaved
KissFFT	FFT	float, double	interleaved
muFFT	FFT	float	interleaved
Juce	FFT, Biquad	float	interleaved
meowFFT	FFT	float	interleaved
Q	FFT, Biquad	FFT(double)	interleaved
FFTW3	FFT	float, double	split
cmsis-DSP	Biquad	float, double	N/A
Bela	Biquad	float	N/A
Maximilian	Biquad	float, double	N/A
SND	Biquad	float	N/A
moog Ladder	Biquad	float	N/A
IIR1	Biquad	double	N/A
SST	Biquad	float	N/A
vDSP	Biquad	float, double	split, interleaved

Within our implementation, and according to I, the libraries that use split complex data structures are not compatible with Windows on Arm as of yet. The impact of the different forms is investigated through the Apple benchmarks by assessing their implementation in the different libraries (Hecquet, Grabit, and Rolland 2024).

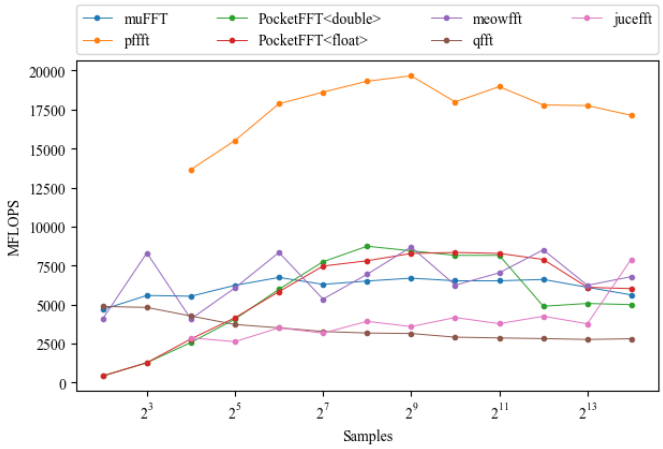

Fig. 1. Benchmark of FFT libraries on a Snapdragon® 8cx Gen 3 compute platform and Windows on Arm in MFLOPS

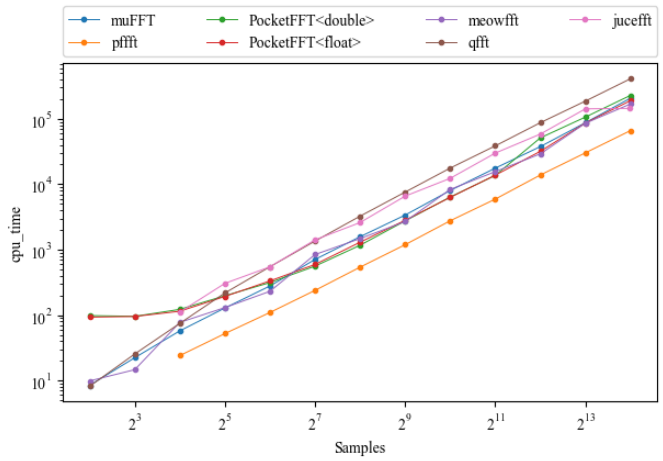

Fig. 2. Benchmark of FFT libraries on a Snapdragon® 8cx Gen 3 compute platform and Windows on Arm in nanoseconds

IV. Results

The benchmark is run on the Windows Dev Kit computer with 32GB LPDDR4x RAM, 512GB fast NVMe storage, and a Snapdragon® 8cx Gen 3 compute platform Software on a Chip (SoC). The results are output into JSON format for parsing using the –benchmark_out=path/to/results.json and –benchmark_out_format=json commands while executing the solution in Visual Studio. The results are then tabulated and compared. A Jupyter notebook and python library were made to easily parse and visualize the data.

A. FFT Results

Figure 1 shows that the fastest library is pffft, a library based on the FFTPACKv4 algorithm which follows the split vectorization method suggested by Swarztrauber (1985). The library performed twice as fast than all other libraries consistently over all buffer sizes N. In addition, pffft is the only library that maintains a linear behavior with respect to N as seen in figure 2. The complexity of the algorithm is

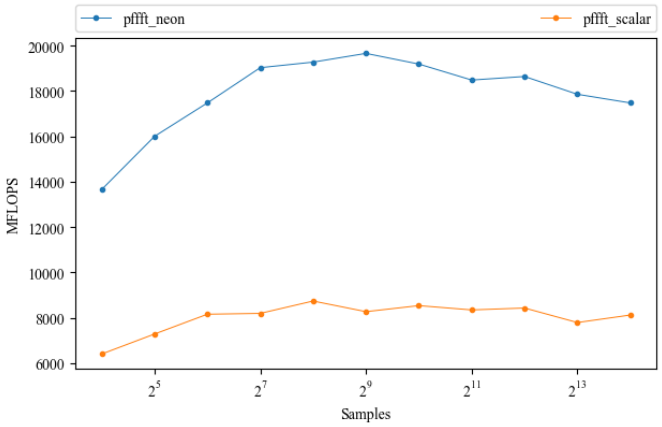

Fig. 3. Benchmark of pffft with a scalar implementation and with Neon SIMD support

$5 * N * log(N)$ for a complex buffer input. The library is also using Arm Neon SIMD for its vectorized implementation. As seen in figure 3, the library performs half as well when Neon support is disabled. In general, the open-source libraries have outperformed the commercial libraries with JUCE ranking second to last across most buffer sizes.

B. Biquadratic Filter Results

In the biquadratic filter benchmarks, we found that the bela library was the highest performing library with IIR1 and CMSIS-DSP in close second and third. We also find that the Arm provided library does perform better starting with a buffer length of 64 samples. In most audio processing use cases, the buffer length usually spans from 64 samples and more depending on the application, making CMSIS-DSP a suitable candidate to be used on Windows Dev Kit.

TABLE III
Benchmark results of top 3 biquadratic filter performing libraries on a Snapdragon® 8cx Gen 3 compute platform and Windows on Arm in nanoseconds

Buffer Size	bela	cmsis_dsp_float	Iir
4	9.78	12.77	10.24
8	19.66	23.0639	20.36
16	39.44	41.5488	41.29
32	79.67	80.5782	81.34
64	159.39	157.28	163.77
128	320.92	313.42	325.05
256	641.35	613.18	654.04
512	1267.2	1229.60	1339.46
1024	2541.6	2587.67	2632.70
2048	5076.4	5079.24	5203.31
4096	10177.4	9920.79	10454.87
8192	20319	19825.20	20775.33
16384	40731	39771.26	41981.22

C. Apple Silicon Benchmark Results

While no data was gathered for automatic tuning libraries on Windows on Arm, they were successfully implemented on an Apple Silicon MacBook. As reported by Hecquet, Grabit,

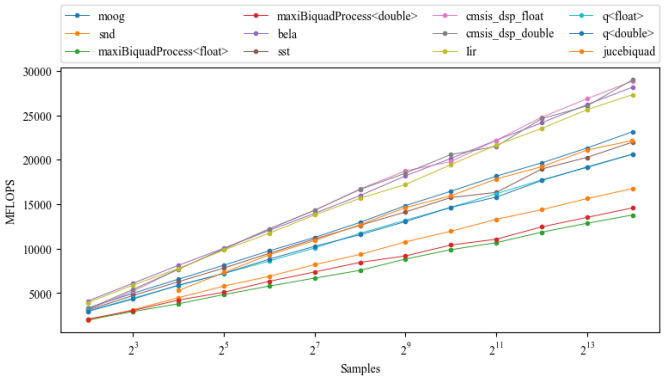

Fig. 4. Benchmark of biquadratic filter libraries on a Snapdragon® 8cx Gen 3 compute platform and Windows on Arm in MFLOPS

and Rolland (2024) FFTW3 only performs better than vDSP in the smallest buffer size of 4 samples. Interleaved library KFR performs the best in float precision and smaller buffer sizes ranging to 64 samples. The library that performed the best across FFTs and biquadratic filters in general was Apple's vDSP. Pffft still ranked well in the FFT benchmark on Apple Silicon while ranking first in Windows on Arm. Moreover, we can observe in table IV that split format FFT libraries generally perform better than interleaved libraries, with vDSP 1.25x to 1.8x faster while using split data structures.

TABLE IV
BENCHMARK RESULTS OF TOP FFT LIBRARIES ON APPLE SILICON IN NANOSECONDS

Buffer Size	vDSP	vDSP interleaved	pffft	KFR	FFTW
4	8.95	N/A	N/A	4.06	3.84
64	49.8	61.8	56.1	47	111
1024	578	1041	1096	1485	3263
16384	15067	N/A	29386	36629	16384

Furthermore, as seen in table V, CMSIS-DSP is second to vDSP in terms of performance. We can also observe from table III that CMSIS-DSP performs better on the Snapdragon® 8cx Gen 3 compute platform than on Apple silicon. Lastly, we compare the top Windows on Arm FFT library to its performance on Apple. As shown by the results in table VI, pffft on Apple performs twice as fast as on Windows on Arm. Pffft is the 3rd ranked library on Apple.

TABLE V
BENCHMARK RESULTS OF TOP BIQUADRATIC FILTER LIBRARIES ON APPLE SILICON IN NANOSECONDS

Buffer Size	vDSP	CMSIS-DSP	Q
4	13.8	13.6	12.5
64	67.9	168.0	201.0
1034	969.0	2666.0	3200.0
16384	15338.0	44160.0	51259.0

V. DISCUSSION

The fast performance of pffft across Windows on Arm and Apple shows the successful implementation of parallelization

TABLE VI
PFFFT PERFORMANCE ON SNAPDRAGON® 8CX GEN 3 COMPUTE PLATFORM VS APPLE SILICON IN MFLOPS

Buffer Size	64	1024	16384
macOS	34224.59	47237.67	39028.11
Windows on Arm	17523.22	18526.84	17360.50

suggested by Swarztrauber (1982) through SIMD Neon on Armv8. Pffft is using hand tuned Neon intrinsics, according to its documentation, improving performance as discussed by Orozco et al. (2007). Though, while ranking 3rd in Apple silicon's benchmark, pffft's performance on Apple was still 2x faster than that of the library on Windows on Arm. This shows that, even though the library is the current fastest for Windows on Arm, a more optimized library could be more achieved. The difference between auto-tuned and vendor-tuned algorithms is yet to be investigated on Windows on Arm due to compatibility issues. However, the data provided by the Apple benchmark suggest that Apple's vendor-tuned library vDSP performed better than the auto-tuning library FFTW3. Moreover, libraries with split complex data structures are more optimal on Apple silicon. Again, this has yet to be tested on Windows on Arm since there are no implementations of this in any of the compatible FFT libraries.

Looking into the biquadratic filter results, the top performing library for Windows on Arm is the Bela library which is designed for the Bela circuit modules. These modules are built to have low-latency performance and multiple assignable input and output pins. Bela is typically used as the micro-controller for digital music instruments, interactive installations, and audio development research and education. While the library has been designed for performance, it has shown to perform well on larger Arm processors. This hints that some libraries optimized for low-powered applications are still suitable on the Armv8 architecture in larger processors such as Qualcomm's. Furthermore, the performance of CMSIS-DSP was not anticipated, as the library is built for computer vision applications and automotive. Moreover, the library was primarily built to be an education focused library, and not a performance focused library. The difference between computer vision and audio is the fact that image processing is multi-axial while audio processing is only 1-dimensional. Furthermore, audio data bit depth usually ranges from 16 to 32 bits while image data bit depth is usually between 8 and 24 bits. This shows the lack of libraries capable of high-end audio manipulation on high bit rates for Windows on Arm. The parallelizing of cascading biquadratic filters has not yet been successfully implemented with Neon SIMD on Windows on Arm, as the fastest libraries still do not support Neon. While CMSIS-DSP does support Neon, it does not properly implement it on biquadratic filter functions. Thus, improvements for biquadratic libraries on Windows on Arm are still possible through the usage of parallel computing as suggested by Chaurasia et al. 2015.

Finally, looking into commercial audio processing libraries, JUCE has proven to have poor performance on Windows

on Arm. While Intel Windows machines posses extremely efficient libraries, such as iPP and MKL, and macOS having their own exclusive high performing library, Windows on Arm devices are yet to have a similar state-of-the art library optimized for the Windows on Arm kernel and Qualcomm processor.

VI. Conclusion

We implemented a benchmark of multiple FFT and biquadratic filter libraries and provided metrics of their performance on a Windows on Arm machine running a Snapdragon Compute Chip. The results have been compared with another Armv8 MacBook to further understand the architecture. The effects of SIMD Neon intrinsics on FFT performance is evident across both platforms as seen with pffft. We also demonstrated the need for optimization of digital signal processing libraries on Windows on Arm.

VII. Future Work

There is a need for more libraries to be made compatible with Windows on Arm and more investigations to take place at a lower-level. As Windows is planning on providing native Arm tools such as a native Clang compilers to their newly released computers, these tools would help future implementation of native libraries and cross-compatibility. Thus, future implementations of this benchmark framework and optimization findings could be used to test the newly compatible libraries and evaluate their performances. Furthermore, these computers will run on the newer Arm processor, the Snapdragon® Elite X compute platforms, and will have an added Armv9 instruction architecture. The benchmark and findings could be used to assess Neon SIMD implementations in the Armv9 architecture and processors, and the nuances of library performance between Armv8 and Armv9. While this project focused on micro-benchmarks, the libraries might behave differently in a broader context application. This calls for macro-benchmarking the libraries using tools like Windows performance recorder to analyze the CPU thread while running the program. Finally, the techniques and libraries discussed within this paper are helpful in the search of a state-of-art digital signal processing library for Windows on Arm.

VIII. Acknowledgments

This work was done in partnership with Steinberg GmbH. I would like to thank my advisor Jean-Baptiste Roland for his guidance along this work. I would also want to extend my thanks for Steinberg for giving me the right tools to complete this work. Finally, I would like to thank Dr. George Fazekas for the oportunity and counsel.

References

Ayala, Alan et al. (2021). "Interim report on benchmarking FFT libraries on high performance systems". In: *University of Tennessee, ICL Tech Report ICLUT-21-03* 7, p. 2021.

Bell, Adam Patrick (2018). *Dawn of the DAW: The studio as musical instrument*. Oxford University Press.

Blem, Emily, Jaikrishnan Menon, and Karthikeyan Sankaralingam (2013). "Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, pp. 1–12.

Chaurasia, Gaurav et al. (2015). "Compiling high performance recursive filters". In: *Proceedings of the 7th conference on high-performance graphics*, pp. 85–94.

Cooley, JW and JW Tukey (1965). "An algorithm for the machine computation of the complex fourier series, in mathematics of computation. 9: 297–301". In: *April*.

Dall, Christoffer et al. (2016). "ARM virtualization: performance and architectural implications". In: *ACM SIGARCH Computer Architecture News* 44.3, pp. 304–316.

Du, Qi and Hui Huang (2020). "Research on the realization and optimization of FFTs in ARMv8 platform". In: *IOP Conference Series: Materials Science and Engineering*. Vol. 768. 7. IOP Publishing, p. 072114.

Frigo, Matteo and Steven G Johnson (2005). "The design and implementation of FFTW3". In: *Proceedings of the IEEE* 93.2, pp. 216–231.

Gupta, Khushi and Tushar Sharma (2021). "Changing trends in computer architecture: A comprehensive analysis of arm and x86 processors". In: *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* 7.

Hecquet, Blaise, Yvan Grabit, and Jean-Baptiste Rolland (2024). *DSP OPTIMIZATION FOR ARM PROCESSORS*.

Johnson, Jeremy et al. (2000). "Searching for the best FFT formulas with the SPL compiler". In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer, pp. 112–126.

Kutil, Rade (2009). "Short-vector SIMD parallelization in signal processing". In: *Parallel Computing: Numerics, Applications, and Trends*. Springer, pp. 397–433.

Laroche, Jean and Mark Dolson (1999). "New phase-vocoder techniques for pitch-shifting, harmonizing and other exotic effects". In: *Proceedings of the 1999 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics. WASPAA'99 (Cat. No. 99TH8452)*. IEEE, pp. 91–94.

Mulgrew, Bernard, Peter Grant, and John Thompson (2002). *Digital signal processing: concepts and applications*. Bloomsbury Publishing.

Nercessian, Shahan, Andy Sarroff, and Kurt James Werner (2021). "Lightweight and interpretable neural modeling of an audio distortion effect using hyperconditioned differentiable biquads". In: *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, pp. 890–894.

Nguyen, Huy and Lizy Kurian John (1999). "Exploiting SIMD parallelism in DSP and multimedia algorithms using the AltiVec technology". In: *Proceedings of the 13th international conference on Supercomputing*, pp. 11–20.

Orozco, Daniel et al. (2007). "Experience of optimizing FFT on Intel architectures". In: *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, pp. 1–8.

Rahman, Tahmid Noor, Nusaiba Khan, and Zarif Ishmam Zaman (2024). “Redefining Computing: Rise of ARM from consumer to Cloud for energy efficiency”. In: *arXiv preprint arXiv:2402.02527*.

Reiss, Joshua D (2010). “Design of audio parametric equalizer filters directly in the digital domain”. In: *IEEE transactions on audio, speech, and language processing* 19.6, pp. 1843–1848.

Swarztrauber, Paul N (1982). “Vectorizing the ffts”. In: *Parallel computations*. Elsevier, pp. 51–83.

Xiong, Jianxin et al. (2001). “SPL: A language and compiler for DSP algorithms”. In: *ACM SIGPLAN Notices* 36.5, pp. 298–308.