Hardware-Accelerated Real-Time Stream Data Processing on Android with GNU Radio

Bastian Bloessl
Technical University of Darmstadt
mail@bastibl.net

Lars Baumgärtner
Technical University of Darmstadt
baumgaertner@cs.tu-darmstadt.de

Matthias Hollick
Technical University of Darmstadt
mhollick@seemoo.tu-darmstadt.de

ABSTRACT

With the ever-increasing performance of smartphones and tablets, they become viable platforms for applications that were, in the past, only possible on desktops or laptops. In this paper, we study their applicability for real-time stream-data processing, which is particularly interesting for Software Defined Radio (SDR) applications, enabling wireless measurement and experimentation campaigns on mobile platforms. To this end, we port GNU Radio, a state-of-the-art, open source, real-time stream-data processing framework, to Android and evaluate its performance. We show that it is possible to fully benefit from available accelerators, i.e., Single Instruction Multiple Data (SIMD) and the Graphics Processing Unit (GPU), which provide considerable speedups and allow for efficient implementations. As a general-purpose real-time data processing framework, GNU Radio can provide the base for a wide range of applications. To demonstrate its flexibility, we provide example applications that implement FM and Wireless LAN (WLAN). Our toolchain is published as open source software, thus serving as an enabler for highly mobile SDR applications.

CCS CONCEPTS

• **Networks** → **Network experimentation**; **Mobile networks**.

KEYWORDS

SDR, Android, GNU Radio, Experimentation, Wireless Networks

ACM Reference Format:
Bastian Bloessl, Lars Baumgärtner, and Matthias Hollick. 2020. Hardware-Accelerated Real-Time Stream Data Processing on Android with GNU Radio. In *14th International Workshop on Wireless Network Testbeds, Experimental evaluation & Characterization (WiNTECH'20), September 21, 2020, London, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3411276.3412184

1 INTRODUCTION

With the ever-increasing performance of mobile devices, like smartphones and tablets, they become powerful general-purpose compute platforms, featuring capable processor and GPUs. Today, smartphones support applications that go far beyond their initial use-cases, like browsing, emails, texting, or calls. In this paper, we explore their applicability for real-time stream-data processing, in particular, real-time signal processing for SDR applications.[1] To this end, we port GNU Radio [14], a state-of-the-art, real-time stream-data processing framework, to Android and evaluate its performance. Our port supports the two main architectures, ARMv7-A (`armeabi-v7a`, 32 bit) and ARMv8-A (`arm64-v8a`, 64 bit) and is integrated in Android to create full-featured SDR applications that can interface USB-based hardware frontends without the need to root the device. While Android applications are mostly implemented in Java, all data processing happens in C++ domain, driven by GNU Radio's heavily parallelized runtime environment. Our port takes full advantage of the available accelerators, i.e., SIMD instructions and the GPU, enabling efficient high-performance applications.

GNU Radio comes with a comprehensive library of optimized, state-of-the-art signal processing algorithms and can be further extended through third-party modules, many of which implement specific technologies, like WLAN, ZigBee, or LoRa. Porting the framework, therefore, does not enable a specific application but provides the base for a wide range real-time stream-data processing systems. Prime applications for our port are wireless measurement and experimentation campaigns. Here, smartphones provide distinct advantages: (1) They are portable, standalone platforms with significant compute power that will continue to improve in the future. (2) They are relatively cheap, considering their features and capabilities (e.g., CPU, GPU, storage, display, multi-touch input, GPS, audio, sensors, and camera). With these properties, they are well suited for field-testing novel technologies, like IEEE 802.11p, a WLAN variant for use in vehicle-to-everything communications, including vulnerable road user, like pedestrians or cyclists. In the medium-term, smartphones could even become standalone SDR platforms without requiring additional hardware. Schulz et al. [17] showed that it is possible to modify the firmware of WLAN chips from Broadcom (as used, for example, in the Nexus 5 smartphone) to transmit arbitrary waveforms. While these experiments are, at the moment, limited to transmitting pre-loaded baseband samples, a better understanding of the Direct Memory Access (DMA) controller might allow streaming samples to the device, enabling SDR-like capabilities with integrated hardware.

While GNU Radio includes many processing blocks that are targeted towards signal processing, it is in no way limited to this particular use-case. Its runtime environment is a general-purpose stream-data processing framework that can be used for audio, video, or sensor data processing. This can support, for example, augmented reality or virtual reality applications. Overall, the contributions can be summarized as follows:

WiNTECH'20, September 21, 2020, London, United Kingdom
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8082-9/20/09…$15.00
https://doi.org/10.1145/3411276.3412184

[1]Note that we use the term *real-time* as it is used in the signal processing context. It implies that the data can be processed live while the system is running. It does not imply latency or deadline guarantees.

- We present a state-of-the-art, stream-data processing framework for Android that supports the two most common architectures (ARMv7-A and ARMv8-A) and make it available as open source software.[2]
- We provide a performance evaluation of the GNU Radio runtime environment on Android, showing that both the ARM platform and the Android operating system are well suited for real-time data processing.
- We show the potential of hardware acceleration through SIMD instructions and data processing on the GPU.

2 RELATED WORK

There has been a broad interest in implementing real-time signal processing systems on embedded devices, in particular on ARM platforms [5, 9, 10, 18]. The most relevant platforms in this context are heterogeneous architectures that combine a CPU with accelerators, like DSP co-processors, Field-Programmable Gate Arrays (FPGAs), or GPUs. The main challenge is to integrate and exploit these accelerators, which can provide significant performance gains. Muniz [10] studies the performance of GNU Radio on a Texas Instruments TCI663x System on Chip (SoC), which includes co-processors for common signal processing functions. They show that offloading the computation of a Fast Fourier Transform (FFT) and decoding of Turbo Codes to the DSP can provide speedups of over 100 (for the FFT) and 40 (for the Turbo Decoder). Similarly, Fayez et al. [5] compare the performance of a Finite Impulse Response (FIR) filter that is computed on the CPU with a corresponding implementation on the DSP of a Texas Instruments OMAP3530 processor. Depending on the size of the input data, they measure speedups of up to 40.

Recently, FPGAs became more popular on embedded devices. This is mainly driven by Xilinx's Zynq series of devices, which combine an ARM processor with an FPGA on one chip. Marlow et al. [9], for example, present a framework to integrate Zynq FPGA accelerators into GNU Radio. A similar concept is used by RFNoC [4], a framework from Ettus Research that eases the use of FPGAs on recent SDRs, including their embedded series, which is also based on Xilinx Zynq processors. Like our paper, these works do not focus on specific applications but provide frameworks that serve as the base for actual applications.

Modern processor support SIMD instructions that operate on vectors of data in similar execution times than scalar operations. Ideally, these instructions would not require any explicit support from the application, as the compiler would use them whenever appropriate. In reality, many opportunities are not recognized. GNU Radio, therefore, uses the Vector-Optimized Library of Kernels (VOLK), which provides optimized implementations for common signal processing functions [15] that make explicit use of SIMD instructions. The library was later extended for Neon, the SIMD instruction set available on recent ARM platforms [18], which provided speedups of up to ten. In this paper, we integrate VOLK into our GNU Radio Android port and provide benchmarks for a smartphone.

Compared to SIMD instructions, GPUs provide an even higher level of parallelism by executing *kernels* that perform similar operations on vectors or arrays of data. The most popular frameworks

[2]The toolchain and example applications are available at https://github.com/bastibl/gnuradio-android.

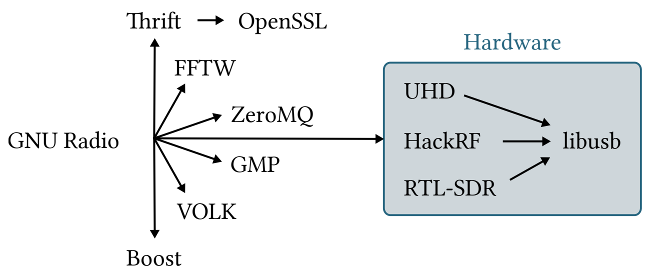

Figure 1: Library dependencies for GNU Radio. A → B indicates that A depends on B.

for general purpose computations on GPUs are Nvidia's Compute Unified Device Architecture (CUDA) and the Open Computing Language (OpenCL), which provides a vendor-independent standard. Both have been studied heavily also in the context of SDR [1, 6, 7, 12]. An issue that has been identified is that data cannot be processed directly by the GPU but has to be copied to GPU memory first. While there are OpenCL implementations that support zero-copy or shared virtual memory, these features require page aligned data or specifically allocated buffers. At the moment, this is not possible with GNU Radio. There is, however, ongoing work towards more flexible buffers [7]. In this paper, we focus on `gr-clenabled`[3] [12], a recent GNU Radio extension module that uses OpenCL, integrate it in our Android port and evaluate its performance.

We are not the first to study real-time signal processing on smartphones. Park et al. [11] presented IEEE 802.15.4 and IEEE 802.11p implementations for Android to interoperate with Internet of Things (IoT) devices. In contrast to our work, they demonstrate the feasibility by implementing specific applications, whereas we provide a general purpose, open source framework that can be used for arbitrary applications. Using GNU Radio, we can benefit from an ecosystem that provides third-party extensions that implement a wide range of technologies. Furthermore, our port integrates in Android and does not require rooting the device.

Our work is based on a GNU Radio Android port by Rondeau [13]. It provided a proof-of-concept, showing that it is possible to run GNU Radio on Android and interface SDR frontends without the need to root the device. This work also provided hardware drivers for Android sensors, audio, and SDR frontends. We extend this work, by updating it to GNU Radio v3.8, adding an automated build system, adding support for recent ARM architectures, and adding support for more SDR frontends. In addition, we contribute with a performance evaluation of the GNU Radio runtime on Android, full accelerator support (SIMD and GPU), and non-trivial example applications that demonstrate the potential of our work.

3 GNU RADIO ON ANDROID

Android is, together with iOS, one of the major operating systems for mobile devices, like tablets and smartphones. On Android, applications are mainly written in Java or Kotlin, with the option to access shared libraries through the Java Native Interface (JNI). GNU Radio itself is implemented in C++. To use it on Android, we cross-compile the most recent version and all its direct and

[3]https://github.com/ghostop14/gr-clenabled

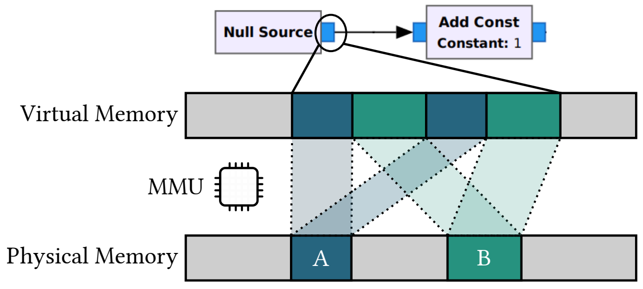

Figure 2: GNU Radio uses double-mapped circular buffers for stream-data processing. We provide an Android implementation that uses *Android Shared Memory.*

indirect dependencies using the Android Native Development Kit (NDK). Our toolchain supports the two most popular architectures, ARMv7-A (*armeabi-v7a*, 32-bit) and ARMv8-A (*arm64-v8a*, 64-bit). The dependency graph of the libraries is shown in Figure 1. Since most libraries do not support native Android builds, we had to modify their build framework and in some cases adapt the source code. To ease the use of our toolchain, we release build scripts that reference the modified libraries through submodules of the version control system. This avoids issues with compatibility between libraries and manual patching. The output of the build scripts are shared libraries that can be included in Android applications and accessed through the JNI. Note that while GNU Radio and the Android SDK require around 17 GByte on the development machine, GNU Radio Android applications only require tens of MByte.

To fully integrate GNU Radio on Android, we adapted the logging framework (which was changed from `log4cpp` to Android's native library) and the configuration subsystem (which considers Android's *external storage*). External storage, is a global directory that is accessible to all applications with the corresponding permission. This way, we have one configuration that is used by all GNU Radio applications. To access the phone's sensors, speakers, and the microphone within GNU Radio, we updated the existing GNU Radio module[4] that was developed as part of the earlier port.

Double-Mapped Circular Buffers

Efficient stream-data processing is at the core of GNU Radio. Its runtime environment implements a producer–consumer pattern with a ring buffer between subsequent processing blocks. Each block is executed in a separate thread, allowing for parallelized processing. The buffer is implemented as double-mapped circular buffer. When allocating a buffer, we use the Memory Management Unit (MMU) to map, not necessarily contiguous physical memory twice, back-to-back in the virtual address space of the application. An illustration is shown in Figure 2. In the example, the buffer comprises two physical memory regions (A and B) that are mapped into virtual memory following the pattern ABAB. Note that, as indicated in the figure, the buffer is associated with the output port, since there might be more than one downstream blocks that read from the same buffer. The total usable buffer size corresponds to the sum of the sizes of A and B. Since the MMU manages memory through memory pages, the buffer size is always a multiple of the page size. The advantage of this architecture is that operations on the buffer can use contiguous memory, i.e., functions like `memcpy` or filters do not have to deal with the case that the circular buffer wraps around. This case is handled by the MMU, which simplifies the implementation and allows us to use instructions that work on vectors more efficiently.

GNU Radio comes with four implementations for double-mapped circular buffers. Linux usually uses System V shared memory, which is, however, not available on Android. The only compatible implementation uses a temporary file that is memory mapped into the virtual address space. The drawback of this implementation are the need to create temporary file and potential writes to disk that are not under the control of the application. Android, however, introduced Shared Memory (*ashmem*) with API level 26, which we used to implement a double-mapped circular buffer. The implementation was verified with the GNU Radio's unit tests and is used by default by our Android port.

Runtime Optimizations

On a normal Linux system, GNU Radio allows fine-tuning parameters to optimize performance, many of which are not supported by Android. For example, the CPU affinity of the threads cannot be controlled individually, i.e., we cannot assign processing blocks to specific CPU cores. Another limitation is that Android applications cannot spawn threads with real-time priority. On Linux, these threads are handled by dedicated task schedulers (e.g., `sched_rr` or `sched_fifo`), which can improve performance [2]. On Android, all threads are served by the Completely Fair Scheduler (CFS), which allows adjusting the *nice* value to assign more CPU time to threads with higher priority. By default, Android prioritizes threads that are relevant to the user experience, i.e., rendering and audio. In our case, we prioritize data processing and, therefore, assign the smallest nice value, corresponding to the highest priority, to all GNU Radio threads.

Apart from standard Linux process configurations, there are also Android-specific optimizations. Most Android systems feature a *Game Mode*, which further prioritizes the application running in the foreground. While there is no standard implementation, this mode usually allocates more CPU time and suspends background applications to disk to free memory. Further optimizations, like prioritizing network traffic or disabling secondary SIM cards are also typical but less relevant in this context.

Interfacing Hardware

USB-based SDRs like the RTL-SDR dongle, the HackRF, or the Ettus Research B200 series are interesting hardware frontends for small portable setups. Using USB On-The-Go (OTG), the smartphone can power and interface the SDR without any additional hardware. The drivers for these devices are based on `libusb`. On Linux, they interact directly with USB device nodes that are usually mounted under `/dev/bus/usb` or `/sys/bus/usb`. While Android is based on Linux, it implements a stricter security model than typical desktop distributions. This model prevents direct access to the device nodes, which is why Park et al. [11] use a rooted phone to circumvent the problem.

[4]https://github.com/trondeau/gr-grand

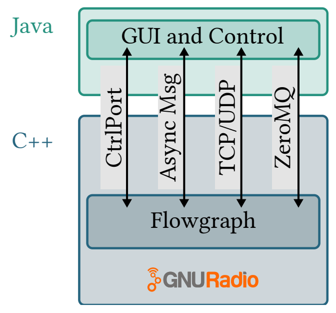

Figure 3: GNU Radio provides four interfaces to communicate between Java and C++ domains.

We use an alternate approach and integrate the driver into Android's security model. To this end, the application requests a file descriptor for the device from Android's *UsbManager* and forwards it to the driver. We, therefore, need to adapt the driver to accept a file descriptor and `libusb` to use this descriptor instead the device nodes. For `libusb` there is a modified version available[5], which we used in our toolchain. Furthermore, we added support for three popular SDRs frontends: the RTL-SDR, the HackRF and the Ettus Research B200 series. While the RTL-SDR and HackRF are USB 2.0 devices, the B200 series also supports USB 3.0, which is available on more recent smartphone SoCs, like the Snapdragon 855 series.

Connecting C++ and Java

With the discussed toolchain, we can setup and run GNU Radio applications using the JNI and process all data in C++ domain. While this is great in terms of performance, it requires methods to interact with the flowgraph from Java, for example, to set parameters or visualize the signal. GNU Radio comes with a wide range of interfaces (cf. Figure 3). The most flexible option is *ControlPort* [16], which exposes a Remote Procedure Call (RPC) interface to the flowgraph and individual processing blocks. The interface is based on Apache Thrift, an RPC protocol that GNU Radio uses on top of TCP/IP, allowing us to interface a running flowgraph via network sockets. An alternate method is to use GNU Radio's asynchronous message passing interface, which allows thread-safe message injection from code outside the flowgraph. This interface is, however, limited to posting messages, i.e., it is not possible to subscribe to updates or read parameters.

A limitation of both ControlPort and asynchronous messages is that the interfaces cannot be used to stream data into or out of the flowgraph. This can be achieved with TCP/UDP sockets or ZeroMQ, a higher-layer networking library with support for messaging patterns like request-reply or pub-sub.

4 RUNTIME PERFORMANCE

Having a full port of GNU Radio, we were interested in assessing the performance of SDR applications on smartphones. In particular, we were curious if Android interferes with CPU intensive, multi-threaded applications like GNU Radio, which spawns a thread for each processing block in the flowgraph. To this end, we benchmark the GNU Radio runtime similar to a previous study that focused on the performance on desktop PCs [2]. We create a flowgraph with a structure as shown in Figure 4, that allows us to change the number of *pipes* (i.e., parallel streams) and *stages* (i.e., blocks per stream) programmatically. Since we focus on the performance of the runtime environment, as opposed to the performance of individual blocks, we just copy data through the flowgraph without any additional processing. To minimize the impact of the measurement on the result, we pipe a given amount of data into the flowgraph and measure the execution time, i.e., the time it takes to propagate the data through the flowgraph. This way, we do not have to instrument the flowgraph, which would inevitably cause overhead and potentially impact its runtime behavior. We, furthermore, create and setup the whole flowgraph in advance to only measure the time it takes to propagate the data.

[5]https://github.com/videgro/libusb.git

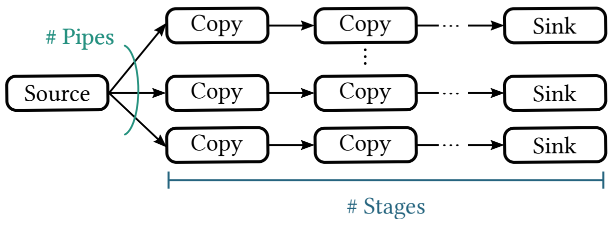

Figure 4: Flowgraph topology, used to evaluate runtime performance of GNU Radio.

The following benchmarks are conducted with a OnePlus 5T smartphone, featuring an octa-core Qualcomm Snapdragon 835 processor with an Adreno 540 GPU, which was released in the first quarter of 2017. The phone runs Android 9 (API level 28). It was fully charged before the measurement and remained connected to the PC during the measurement. To ensure that the benchmark application remains in the foreground, we configured the phone to stay active (i.e., not go into standby mode) while connected via cable. We, furthermore, put the phone in *Flight Mode* to minimize interference from other system services. The GNU Radio library and the Android application were compiled in release mode. We set the *nice* value of the GNU Radio threads to highest priority, exceeding the default priorities of rendering threads, and enable *Game Mode* to prioritize the application over other system services. Using this setup, we evaluate both the stream-data and the message passing interface.

To evaluate stream-data performance, we use a *Null Source* and a *Head* block to stream 200×10^6 32 bit floats into the flowgraph. We scale the number of pipes and stages jointly, i.e., an x-value of 25 corresponds to five pipes and five stages (cf. Figure 4). The average execution time of ten runs per configuration are shown in Figure 5. The error bars indicate the confidence interval of the mean for a confidence level of 95 %. To put the performance of the smartphone into context, we conducted similar measurements on a laptop with an Intel i7-8565U that features eight CPUs (four cores with hyperthreads), running Ubuntu 19.10 and the same GNU Radio version as the phone.

The results show that the execution times of both the smartphone and the laptop scale about linearly, matching the amount of data that has to be passed through memory. This is a positive

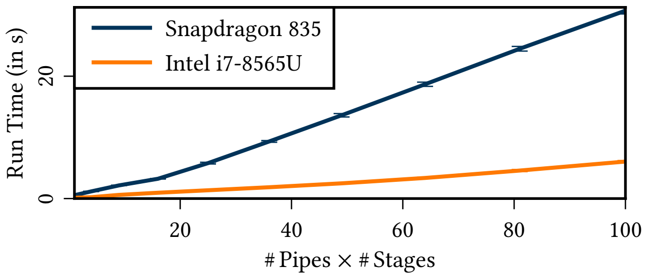

Figure 5: Execution time, required to stream 200 × 10^6 samples through the flowgraph. Pipes and stages are scaled jointly, i.e., an x-axis value of 25 corresponds to five pipes and five stages.

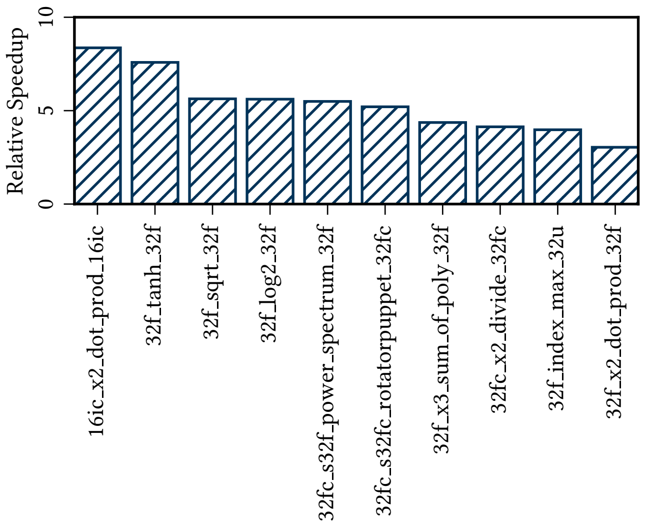

Figure 6: Top 10 SIMD accelerated functions with highest speedup compared to C implementations.

result, as it suggests that Android does not interfere with real-time signal processing applications. Even the configuration with 100 threads runs without getting throttled by the operating system. Yet, in comparison to the laptop, the execution time on the phone is significantly higher with a factor of about five. Similar results were observed in an analog experiment for the asynchronous message passing interface of GNU Radio (data not shown).

While the performance of the smartphone is lower, it is far from rendering the platform impractical for SDR, especially considering that we compare a top smartphone SoC from 2017 with a top laptop CPU from 2018. Furthermore, the octa-core Snapdragon does not feature eight similar CPUs like the laptop but uses ARM's *big.LITTLE* architecture, which combines four high-performance cores with four energy-efficient slower cores. With the continuous performance improvements of smartphones, we believe that more and more SDR applications become feasible on mobile platforms.

5 SIMD ACCELERATION

ARMv7 introduced SIMD acceleration through the NEON instruction set extension and a corresponding co-processor. To exploit SIMD instructions, GNU Radio relies on VOLK[6], a library of common functions (called *kernels*) that can benefit from vectorized instructions. Every kernel comes with a generic C implementation and optional optimized versions that use, for example, assembly instructions, compiler intrinsics, or lookup tables. VOLK chooses the most efficient implementation during runtime based on profiling data, generated with *volk_profile*, a tool that benchmarks all implementations of a kernel that are supported by the platform. To evaluate the potential of SIMD on a smartphone, we created an Android version of *volk_profile* and adapted the library to read the benchmarking results from Android's external storage. We profiled both the ARMv7-A and the ARMv8-A implementation, using the default parameters, where each function processes a vector of 131 071 items 1987 times.

The results are shown in Figure 6, where we plot the speedup of the Top 10 functions that provide the highest speedup relative to a generic C implementation for ARMv8-A. VOLK differentiates between aligned and unaligned kernels. The former allows exploiting more efficient instructions for aligned input data. We evaluated the differences between these versions but found that only the computation of the complex conjugate showed different performance with a speedup of approximately two. Figure 6 shows the results for aligned implementations. Note that VOLK functions follow the naming scheme `<input data type> [number of inputs] function name <output data type>`. The data types encode the number of bits and the type (`i` for integer, `ic` for a complex number consisting of two integers, `u` for unsigned integer, `f` for floats, `fc` for a complex number consisting of two floats). Most functions work on vectors of data, however, some also accept scalar inputs (e.g., `multiply` supports multiplying a vector with a scalar). These scalars are prefixed with an `s`.

For ARMv8-A, the speedup is up to eight, with many functions providing a speedup between two and six. For ARMv7-A, we see qualitatively similar results, albeit the functions that benefit most do not match one-to-one between the architectures (data not shown). VOLK includes 54 functions with optimized implementations for both ARMv7-A and ARMv8-A. Overall, 51 and 36 out of these functions show best performance for ARMv7-A and ARMv8-A, respectively. This shows that VOLK is still relevant, as compilers do not recognize all situations where SIMD instructions can be used. Optimized implementations, using compiler intrinsics or assembly, can, therefore, still provide significant performance benefits, as even the compiler for the more recent architecture misses many opportunities to optimize implementations.

6 GPU ACCELERATION

GPUs are another interesting option to accelerate stream-data processing. While Android does not support *OpenCL* through native APIs, the GPUs of most modern smartphones do and the corresponding libraries are part of their system image. The Snapdragon 835 processor, for example, comes with an Adreno 540 GPU that supports OpenCL 2.0 *Full Profile*. Using the GPU on Android is

[6]http://libvolk.org/

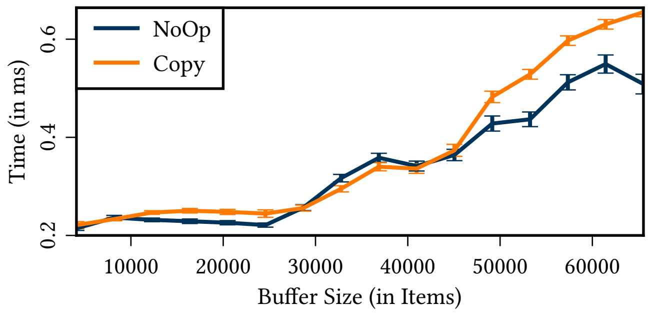

Figure 7: OpenCL baseline benchmark for the average execution times of a *NoOp* and a *Copy* kernel for different buffer sizes.

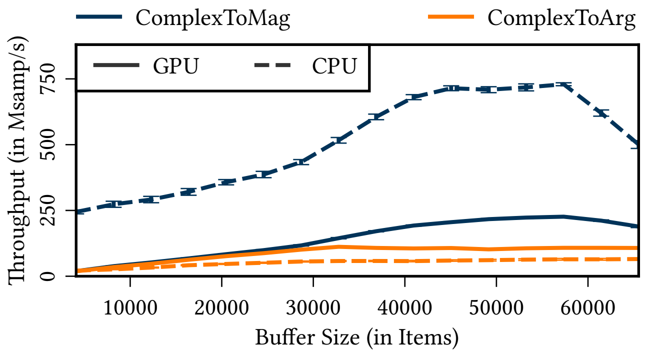

Figure 8: Throughput when computing the magnitude (*ComplexToMag*) and the argument (*ComplexToArg*) of a complex number on the CPU and the GPU.

possible by linking against the OpenCL libraries of the chip vendor, which also ties the application to the platform. This is unavoidable given the lack of native OpenCL support on Android.

Whether a flowgraph can benefit from GPU acceleration, depends on whether the potential processing gain of the GPU is worth the overhead of launching OpenCL kernels and copying the data back-and-forth. As earlier works already noticed, there is a non-trivial relationship between the buffer size and the throughput [1, 7, 12]. We, therefore, conduct baseline measurements similar to Piscopo [12], where we measure the average executing time for the *NoOp* and *Copy* kernels. The *NoOp* kernel copies data to GPU memory and returns immediately, while the *Copy* kernel copies the input to the output. Both kernels operate on 8 Byte complex floating point numbers (two 4 Byte floats) and distribute the input data according to the *preferred work group size* of the OpenCL implementation. The results are shown in Figure 7. We repeated all measurements 200 times, measuring 100 consecutive runs per repetition to increase the precision, i.e., each data point in the graph is based on 20 000 runs. To avoid profiling the initial overhead to set up buffers, we do warm-up runs that are not taken into account. The error bars indicate the confidence intervals of the mean for a confidence level of 95 %.

There are two main insights, we can gain from the results: First, we can assess whether a block could benefit from moving its computation to the GPU. If its execution time on the CPU is faster than the *Copy* kernel, it will not provide a benefit in terms of maximum throughput. It might, nevertheless, make sense to offload the block and free CPU resources. Second, we can see that both the *NoOp* and the *Copy* kernel do not scale linearly with the buffer size. This indicates that the overhead is not only the memory transfer (which we would assume to scale approximately linear) but also the overhead from distributing the work to GPU threads and launching the OpenCL kernel. The impact of launching the kernel is, furthermore, emphasized by the fact that the *NoOp* kernel (with a one-way data transfer) is not twice as fast as the *Copy* kernel (with a two-way data transfer).

To show the potential for typical signal processing blocks, we provide exemplary results that compare the performance of GPU-accelerated blocks with their corresponding GNU Radio blocks, which use VOLK to select the most efficient SIMD implementation for the platform. While GNU Radio allows us to define a maximum size for the ring buffers, the scheduler launches the block as soon as input data is available. The actual input data of a block might be much smaller, which would affect the measurement. We, therefore, instrument the blocks function without launching a GNU Radio flowgraph, always running it with the same buffer size. The results are shown in Figure 8, where we plot the throughput of a block that computes the magnitude (*ComplexToMag*) and the argument (*ComplexToArg*) of a complex number. Computing the magnitude on the GPU does not increase the throughput for any buffer size, while computing the argument on the GPU provides similar or better performance. For example, for a buffer size of 32 768 items, the throughput can be increased by 94 %. Overall, this shows that it is possible to use the GPU of a smartphone for real-time stream-data processing. Similar experiments for other functions showed that computations of the logarithm and trigonometric functions benefit in particular. Even though we cannot exploit the full potential of the GPU due to the current limitations of the GNU Radio runtime (i.e., the lack of zero-copy and shared virtual memory), the GPU can provide significant improvements, especially for large buffer sizes.

7 EXAMPLE APPLICATIONS

GNU Radio aims to be a general purpose framework that provides the base for real-time signal processing systems. The focus is not on implementing specific technologies or standards but to provide optimized, state-of-the-art implementations for common functions, like filters or synchronization algorithms. There are, however, a wide range of third-party GNU Radio extension modules available on the *Comprehensive GNU Radio Archive Network*[7]. To demonstrate that it is easily possible to use the extensions, we integrated two of them in our toolchain. They come with corresponding Android applications that implement an FM receiver and an IEEE 802.11a/g/p receiver [3]. Using these modules on Android does not require any modifications, i.e., the developer does not have to maintain separate versions.

Figure 9 shows an exemplary setup for the IEEE 802.11 receiver. It shows the OnePlus 5T smartphone that uses a B200mini from Ettus Research as SDR frontend. The smartphone is able to power

[7] https://www.cgran.org/

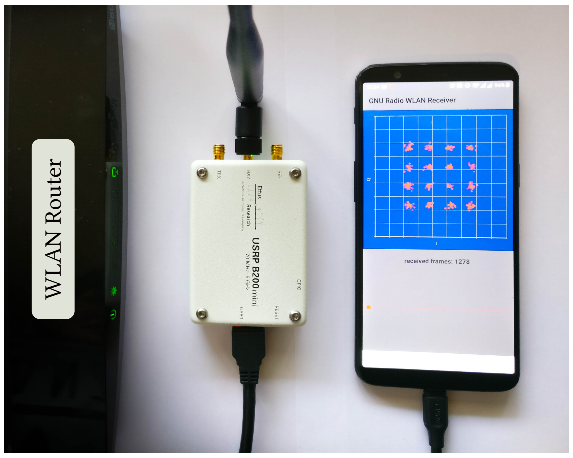

Figure 9: GNU Radio Android application that decodes WLAN frames with an Ettus Research B200mini.

the device, i.e., there is no external power required, providing a very portable setup. Since the phone supports only USB 2.0, i.e., a maximum bandwidth of 8 MHz at full sample resolution, we reduced the bandwidth of the WLAN signal to 5 MHz. This mode is supported by some commercial cards, such as the Atheros AR9582, used in our TP-Link WDR3600 access point that is shown on the left of Figure 9. To switch the card to 5 MHz channels, we used OpenC2X [8], an open source prototyping platform for vehicular communications.

The smartphone shows the equalized constellation points of the Orthogonal Frequency Division Multiplexing (OFDM) signal, which are polled from the flowgraph using GNU Radio's ControlPort interface. The frames are completely decoded and forwarded to Java domain via a ZeroMQ socket. We send 200 Byte frames at a rate of ten frames per second without any losses. This experiment gives an idea about what is possible on smartphones even today. It, furthermore, shows the capabilities of GNU Radio on Android, demonstrating that non-trivial applications are possible.

8 CONCLUSION

We have presented and released a complete toolchain to build GNU Radio and all its dependencies on Android, enabling SDR applications on mobile platforms. We have shown that it is possible to fully integrate GNU Radio and access hardware like microphone, speakers, or SDR frontends without rooting the device. To highlight that Android does not interfere with real-time data processing, we have conducted comprehensive experiments to measure the performance of the GNU Radio runtime environment. Moreover, we have demonstrated the ability to use available hardware accelerators (SIMD and GPU offloading) and benchmarked their performance. Finally, we have presented example applications to show the extensibility of GNU Radio and demonstrate integration with controls and visualization in the Android domain. We believe that this work can serve as the base for a wide range of real-time stream data processing applications, in particular highly mobile SDR implementations for wireless measurement and experimentation.

ACKNOWLEDGMENTS

This work has been co-funded by the LOEWE initiative (Hessen, Germany) within the emergenCITY center, as well as the German Research Foundation (DFG) in the Collaborative Research Center (SFB) 1053 MAKI. Furthermore, it has been co-funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research, and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

REFERENCES

[1] Christopher Becker, Aniqua Baset, Sneha Kasera, Kurt Derr, and Samuel Ramirez. 2018. Experiences with using GNU Radio for Real-time Wireless Signal Classification. In *8th GNU Radio Conference*. GNU Radio Project, Henderson, NV.

[2] Bastian Bloessl, Marcus Müller, and Matthias Hollick. 2019. Benchmarking and Profiling the GNU Radio Scheduler. In *9th GNU Radio Conference*. GNU Radio Project, Huntsville, AL.

[3] Bastian Bloessl, Michele Segata, Christoph Sommer, and Falko Dressler. 2018. Performance Assessment of IEEE 802.11p with an Open Source SDR-based Prototype. *IEEE Transactions on Mobile Computing* 17, 5 (May 2018). https://doi.org/10.1109/TMC.2017.2751474

[4] Martin Braun, Jonathan Pendlum, and Matt Ettus. 2016. RFNoC: RF Network-on-Chip. In *6th GNU Radio Conference*. GNU Radio Project, Boulder, CO.

[5] Almohanad S. Fayez, Nicholas J. Kaminski, Alexander R. Young, and Charles W. Bostian. 2012. Embedded SDR System Design Case Study: An Implementation Perspective. In *13th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2012)*. IEEE, San Francisco, CA. https://doi.org/10.1109/WoWMoM.2012.6263759

[6] Jake Gunther, Hyrum Gunther, and Todd Moon. 2017. GPU Acceleration of DSP for Communication Receivers. In *7th GNU Radio Conference*. GNU Radio Project, San Diego, CA.

[7] Seth Hitefield and T. Charles Clancy. 2016. Flowgraph Acceleration with GPUs: Analyzing the Benefits of Custom Buffers in GNU Radio. In *6th GNU Radio Conference*. GNU Radio Project, Boulder, CO.

[8] Florian Klingler, Gurjashan Singh Pannu, Christoph Sommer, Bastian Bloessl, and Falko Dressler. 2017. Field Testing Vehicular Networks using OpenC2X. In *15th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys 2017), Poster Session*. ACM, Niagara Falls, NY. https://doi.org/10.1145/3081333.3089322

[9] Ryan Marlow, Chris Dobson, and Peter Athanas. 2014. An Enhanced and Embedded GNU Radio Flow. In *24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Munich, Germany. https://doi.org/10.1109/FPL.2014.6927427

[10] Alfredo Muniz. 2015. Enhancing GNU Radio with Heterogeneous Computing. In *2015 Workshop on Software Radio Implementation Forum (SRIF '15)*. ACM, Paris, France. https://doi.org/10.1145/2801676.2801689

[11] Yongtae Park, Seungho Kuk, Inhye Kang, and Hyogon Kim. 2016. Overcoming IoT Language Barriers Using Smartphone SDRs. *IEEE Transactions on Mobile Computing* 16, 3 (May 2016). https://doi.org/10.1109/tmc.2016.2570749

[12] Michael Piscopo. 2017. Study on Implementing OpenCL in Common GNURadio Blocks. In *7th GNU Radio Conference*. GNU Radio Project, San Diego, CA.

[13] Thomas Rondeau. 2015. GNU Radio on Android. Talk at 5th GNU Radio Conference. http://www.trondeau.com/grcon15-presentations

[14] Thomas W. Rondeau. 2015. On the GNU Radio Ecosystem. In *Opportunistic Spectrum Sharing and White Space Access: The Practical Reality*, Oliver Holland, Hanna Bogucka, and Arturas Medeisis (Eds.). Wiley.

[15] Thomas W. Rondeau, Nicholas McCarthy, and Timothy O'Shea. 2013. SIMD Programming in GNU Radio: Maintainable und User-Friendly Algorithm Optimization with VOLK. In *SDR-WInnComm 2013*. Wireless Innovation Forum, Washington, DC.

[16] Thomas W. Rondeau, Timothy O'Shea, and Nathan Goergen. 2013. Inspecting GNU Radio Applications with ControlPort and Performance Counters. In *2nd ACM SIGCOMM Workshop of Software Radio Implementation Forum (SRIF 2013)*. ACM, Hong Kong, China.

[17] Matthias Schulz, Jakob Link, Francesco Gringoli, and Matthias Hollick. 2018. Shadow Wi-Fi: Teaching Smartphones to Transmit Raw Signals and to Extract Channel State Information to Implement Practical Covert Channels over Wi-Fi. In *16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys 2018)*. ACM, Munich, Germany. https://doi.org/10.1145/3210240.3210333

[18] Nathan West, Douglas Geiger, and George Scheets. 2015. Accelerating Software Radio on ARM: Adding NEON Support to VOLK. In *IEEE Radio and Wireless Symposium (RWS)*. IEEE, San Diego, CA. https://doi.org/10.1109/RWS.2015.7129727