

Complete Stream Fusion for Software-Defined Radio

Extended Abstract

Tomoaki Kobayashi
Tohoku University
Japan

tomoaki.kobayashi.t3@dc.tohoku.ac.jp

Oleg Kiselyov
Tohoku University
Japan
oleg@okmij.org

Abstract

Software-Defined Radio (SDR) is widely used not only as a practical application but also as a fitting benchmark of high-performance signal processing. We report using the SDR benchmark – specifically, FM Radio reception – to evaluate the recently developed single-thread stream processing library *strymonas*, contrasting it with the synchronous dataflow system *StreamIt*. Despite the absence of parallel processing or windowing as a core primitive, *strymonas* turns out to easily support SDR, offering high expressiveness and performance, approaching the peak single-core floating-point performance, sufficient for real-time FM reception.

1 Summary

Software-defined radio (SDR)¹ is performing all steps of radio signal processing (save for the antenna reception or transmission) not via analog electric circuits but digitally in software, typically running on an ordinary computer. GNU Radio² is a characteristic example. Besides being a widely used application, it also makes a good benchmark of high-performance signal processing, used as such in [Stewart et al. 2015; Thies 2009]. An example is FM Radio reception, diagrammed in Fig. 1 borrowed from [Thies 2009].

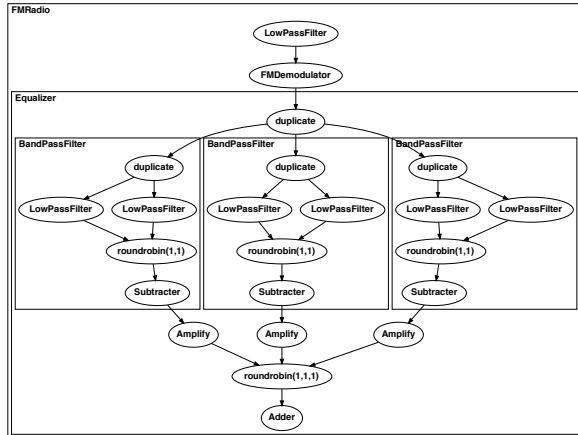


Figure 1. FM Radio reception diagram

1.1 Strymonas

Strymonas [Kiselyov et al. 2017] is an embedded DSL for single-thread stream processing, declaratively assembling stream processing pipelines like Xmas lights, from operators like *map*, *filter*, *zip*,

¹<https://hackrf.readthedocs.io/en/latest/index.html>

²<https://wiki.gnuradio.org/>

flat_map, etc. This work uses the further developed version of *strymonas*.

The characteristic of *strymonas* is the static guarantee of *complete* fusion: if each operator individually runs without any function calls and memory allocations however temporary, the entire streaming pipeline runs without calls and allocations. Thus *strymonas* per se introduces not even constant-size intermediary data structures. An example is *zip_with f*, defined as $\text{zip} \triangleright \text{map} (\text{fun } (x,y) \rightarrow f \ x \ y)$ where \triangleright is left-to-right function composition. A naive implementation would construct a tuple in *zip*, to be deconstructed in the subsequent mapping. In *strymonas*, no intermediate tuples are constructed. The processing loop thus may run without any GC or even stack allocations – of importance to SDR.

Strymonas has several backends to emit code in OCaml, Scala and C – each statically guarantying the generated code compiles without errors or warnings. Here we used the C backend, based on embedding of C in tagless-final style [Kiselyov 2022].

In this work we apply *strymonas* to the FM Radio reception, Fig. 1. There are immediate problems: *Strymonas* does not support *split/join* (or, *duplicate/join*) operations apparent in Fig. 1. Out of the box *strymonas* also does not support signal filtering, or any windowing for that matter. These problems turned out easily surmountable; *strymonas*, hence, is useful and performant for SDR.

2 FM Radio Reception in Strymonas

Although *strymonas* does not support windowing out of the box, it offers enough tools to implement

```
make_stream : float cstream → window stream
```

that converts a stream of float to a window stream, where *window* is an abstract data type with the operations *reduce* and *dot product*. Streams in *strymonas* do not have to be of base types. As an example, FM demodulation is

```
let fmDemodulator (sampRate:float) (max:float) (bndwdth:float)
  : float cstream → float cstream =
```

```
let gain =
```

```
  C.float (max *. sampRate /. (bndwdth *. Float.pi)) in
```

```
let (module Win) = Window.make_window C.tfloat 2 0 in
```

```
Win.make_stream
```

```
▷ map_raw' (Win.reduce C.(. *))
```

```
▷ map_raw C.(fun e k → letl (gain *. atan e) k)
```

The prefix *C* qualifies backend code generation combinators; *map_raw'* is the mapping operator and *map_raw* is its CPS version, used for generating a *let*-binding. We took advantage of OCaml's first-class modules to provide several implementations of the window abstract type, optimized for a particular window size, sliding/tumbling, etc. For example, for short windows we use ordinary variables to store past elements.

To convolve a stream with a given array `arr` we likewise wake a window stream and reduce the window to a float by the dot-product with `arr`:

```
let apply_filter ?(decimation=0) (arr: float array)
  : float cstream → float cstream =
let coeff = Array.map C.float arr in
let taps = Array.length coeff in
let (module Win) =
  Window.make_window C.tfloat taps decimation in
Win.make_stream
▷ map_raw C.(letl (Win.dot tfloat coeff ( + . ) ( * . )))
```

Then low-pass filtering is the straightforward

```
let lowPassFilter (rate:float) (cutoff:float) (taps:int)
  (decimation:int) : float cstream → float cstream =
  apply_filter (lpf_coeff rate cutoff taps) ~decimation
```

where `lpf_coeff rate cutoff taps : float array` computes the coefficients of the low-pass filter with the given parameters.

Equalization is splitting the signal into several bands, amplifying by a band-specific gain and re-combining. This looks, see Fig. 1, far more complex than `lowPassFilter`. However, convolutional filters are linear and so we can perform the equalization on filter coefficients instead. Our equalizer hence is, like `lowPassFilter`, a mere `apply_filter` operation.

Overall, the FM Radio reception is literally

```
lowPassFilter samplingRate cutoffFrequency nTaps 4
▷ fmDemodulator samplingRate maxAmplitude bandwidth
▷ equalizer samplingRate bands eqCutoff eqGain nTaps
```

where `samplingRate` (set to 250MHz), `cutoffFrequency` (108MHz), `bandwidth` (10KHz), `nTaps` (64), etc. are the standard FM Radio parameters.

3 Evaluation

We have verified the correctness of our implementation by writing a naive (and hence obviously correct) `StreamIt` interpreter and checking that the interpretation of all steps of the diagram Fig. 1 gives the same output as the `strymonas` implementation. We also generated a sample FM Radio signal by modulating a sine wave, fed into our implementation and checked the result by ear.

The C code generated by `strymonas` for the FM radio reception spans 216 sparsely-filled lines, half of which are filter coefficients. It is completely vectorized when compiled by GCC.

For performance, we compare with the C reference implementation of FM radio from the `StreamIt` project³ as the baseline, with the same benchmark set-up: 1 million synthetic samples, measuring total processing time. The evaluation platform is 1.8GHz dual-core Intel Core i5, 8 GB DDR3 main memory, macOS Big Sur 11.6. The C code was compiled with GCC 11.2 given the flags `"-O3 -march=native -mfpmath=both -fno-math-errno"` (to be called F1) or with `"-ffast-math"` added (to be called F2). Fig. 2 shows the processing time of 1 million samples, in ms, measured as an average of 20 runs after 5 warmup runs. Compiled with F2 flags, `strymonas` code hence processes in excess of 10M samples/sec, which is the average sample rate of the HackRF SDR board.

³<http://groups.csail.mit.edu/cag/streamit/apps/benchmarks/fm/c/fmref.c>

We also checked memory profile with `valgrind`. Both the baseline and our code showed constant memory use throughout the entire processing, with no heap allocation. However, whereas the baseline used 590 Kib of stack, our code needed 2KiB (for flags F1) or close to zero (for flags F2).

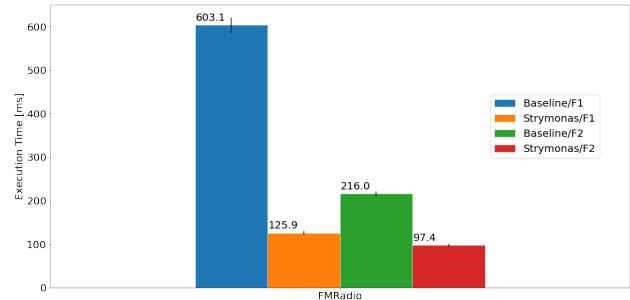


Figure 2. Benchmarking against the baseline

4 Related work

The FM Radio benchmark was borrowed from `StreamIt` [Thies 2009]: a synchronous dataflow programming language with static scheduling. In contrast, `strymonas` is designed for processing on a single core in a single thread, with all operators fused rather than scheduled. Although `strymonas` does not support `split/join`, they are not actually necessary in SDR. For example, the complex equalization in Fig. 1 (implemented in `StreamIt` as drawn) is reducible to the single ordinary FIR filtering.

GNU Radio includes FM Radio as a standard application. Unlike GNU Radio, `strymonas` is typed. Therefore, ill-formed pipelines are rejected before any code is generated, with error messages describing ill-fitting operators. The generated code is statically guaranteed to be well-formed and well-typed, and compiled even without warnings. Stream elements in `strymonas` are not limited to base types: they may be tuples and arbitrary records and objects (such as window seen earlier in the demodulator). The complete fusion is ensured regardless.

We are currently benchmarking against GNU Radio.

References

- Oleg Kiselyov. Generating C. In Michael Hanus and Atsushi Igarashi, editors, *Functional and Logic Programming*, volume 13215 of *Lecture Notes in Computer Science*, pages 75–93. Springer International Publishing, 2022. doi: 10.1007/978-3-030-99461-7_5.
- Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream fusion, to completeness. In *POPL '17: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 285–299, New York, January 2017. ACM Press. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.
- Gordon Stewart, Mahanth Gowda, Geoffrey Mainland, Bozidar Radunovic, Dimitrios Vytiniotis, and Cristina Luengo Agullo. Ziria: A DSL for wireless systems programming. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, pages 415–428, New York, NY, USA, 2015. ACM. doi: 10.1145/2694344.2694368.
- William Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2009. AAI0821753.