Complete Stream Fusion for Software-Defined Radio

Extended Abstract

Tomoaki Kobayashi
Tohoku University
Japan
tomaki.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 Radio\(^2\) 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](image)

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, 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 performance introduces not even constant-size intermediary data structures. An example is.zip_with C.\(f\), defined as.zip \(\triangleright\) map \((f\, x\, y) \rightarrow f\, x\, y\) 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 guaranteeing 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

\[
\text{let fmDemodulator (sampRate:float) (max:float) (bndwdth:float)} \rightarrow \text{float cstream} = \text{let gain} = \text{C.} \,(\text{max} \triangleright\, \text{sampRate} \div\, (\text{bndwdth} \div\, \text{Float.pi})); \text{in} \text{let (module Win)} = \text{Window.make_window C.float 2 0 in} \text{Win.make_stream} \triangleright\, \text{map_raw C.} \,(\text{fmDemodulator} \triangleright\, \text{atan e}) \rightarrow \text{letl} (\text{gain} \triangleright\, \text{atan e} e) k) \]

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:

```ocaml
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_width C.float taps decimation in Win.make_stream ▷ map_raw C.(letl (Win.dot tfloat coeff (+ .) (* .)))
```

Then low-pass filtering is the straightforward

```ocaml
let lowPassFilter (rate:float) (cutoff:float) (taps:int) (decimation:int) : 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

```ocaml
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 project\(^3\) 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.

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).

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


\(^3\)http://groups.csail.mit.edu/cag/streamit/apps/benchmarks/fm/c/fmref.c