Complete Stream Fusion for Software-Defined Radio

Tomoaki Kobayashi
Tohoku University
Sendai, Japan
tomoaki.kobayashi.t3@dc.tohoku.ac.jp

Oleg Kiselyov
Tohoku University
Sendai, Japan
oleg@okmij.org

Abstract

Strymonas is a code-generation–based library (embedded DSL) for fast, bulk, single-thread in-memory stream processing – with the declarative description of stream pipelines and yet achieving the speed and memory efficiency of hand-written state machines. It guarantees complete stream fusion in all cases.

So far, strymonas has been used on small examples and micro-benchmarks. In this work, we evaluate strymonas on a large, real-life application of Software-Defined Radio – FM Radio reception, – contrasting and benchmarking it against the synchronous dataflow system StreamIt, and the state-of-the art: GNU Radio.

Strymonas, despite being declarative, single-thread single-core with no explicit support for SIMD, no built-in windowing or convolution, turns out to offer portable high performance, well enough for real-time FM Radio reception. It is on par with (or, on Raspberry Pi Zero, outstripping) GNU Radio, while providing static guarantees of complete fusion and type safety.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Domain specific languages**; *Source code generation.*

Keywords: stream fusion, stream processing, DSL, software-defined radio, code generation

ACM Reference Format:
Tomoaki Kobayashi and Oleg Kiselyov. 2024. Complete Stream Fusion for Software-Defined Radio. In *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation (PEPM '24), January 16, 2024, London, UK*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3635800.3636962

1 Introduction

Strymonas [3, 5] is the stream processing library that achieves the highest performance of existing OCaml, Java Stream, Scala, etc. streaming libraries, attaining the speed and memory efficiency of hand-written C state machines. It supports finite and infinite streams with the familiar interface: declaratively assembling stream processing pipelines like Xmas lights, from combinators like map, filter, zip, flat_map, take, take_while, drop, drop_while – with no restrictions so long as the types match. The combinators are in turn built on a lower-level, spartan but also declarative, interface of stateful streams, which, in addition, supports accumulating map, compression, etc.

Strymonas statically guarantees *complete fusion*: if each operation in a pipeline individually runs without any function calls and memory allocations, the entire streaming pipeline runs without calls and allocations. Thus strymonas per se introduces not even constant-size intermediary data structures. Complete fusion is mainly the space guarantee: the ability to run the processing loop without any GC or even stack allocations. Avoiding closures and the repeated construction/disposal of tuples, option values, etc. also notably improves performance, in our experience.

The complete fusion is achieved (see [3]) by the careful selection of core (or, 'raw') stream-processing primitives and the normalization-by-evaluation to bring a pipeline to a normal form, from which the completely fused code can be generated.

Strymonas is based on assured code generation, generating OCaml, Scala – and, relevant for the present paper, C99. The latter needs no particular runtime and can be automatically vectorized, as we shall see.

Previously, [3, 5], strymonas was evaluated on micro-benchmarks. The present paper is centered on a macro-benchmark: a large, realistic example of digital signal processing (DSP), specifically: Software-Defined Radio. 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. In DSP, the signal is a potentially infinite stream of samples. Therefore, processing is necessarily confined to a finite subsequence, of the most recent samples: the moving *window* [8, 13]. A particular case of window processing is digital filtering [9], a fundamental operation in DSP – which is also the principal component of SDR, with the most impact on performance.

Strymonas does not provide any windowing primitives. Therefore, the first research question is expressiveness: can windowing and digital filtering be efficiently implemented

PEPM '24, January 16, 2024, London, UK
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0487-1/24/01
https://doi.org/10.1145/3635800.3636962

with the current strymonas? If strymonas has to be extended, do the extensions compromise the complete fusion?

The most well-known and prototypical example of SDR is GNU Radio:[1] a pervasively-used SDR framework, not just by hobbyists but in industrial and government applications. GNU Radio also uses code-generation to a degree: it assembles C++ components (with some hand-written assembly kernels) using Python or Graphical interface. There is little type safety, and hence few guarantees that the assembled code will compile; bugs are easy to introduce and hard to find.[2]

Therefore, the main research question is: can strymonas, while providing type safety, static fusion guarantees, fine-grain declarative interface, also attain the performance of GNU Radio on a characteristic application, real-time FM radio reception?

Real-time FM reception is no simple matter: the HackRF One board[3] often used to acquire and digitize radio signals sends data at the rate of 2 to 20 million samples per second. A typical 3GHz CPU may then spend no more than 1500 CPU cycles per sample. For comparison, on a high-class Sandy Bridge class E Intel CPU, L3 cache access may take up to 38 cycles; main memory access more than 100 cycles. Keeping in mind that FM reception involves 64-tap (or 65-tap) filters with many floating-point operations (see §4.2), this is a rather tight cycle budget.

GNU Radio does not attain complete stream fusion: its processing pipelines have overhead due to function calls and accompanying register saves/restores and resets of floating-point processing, etc. GNU Radio compensates by using hand-written assembly kernels and exploring multiple cores and parallelism. Strymonas, on the other hand, is single-core single-thread library and relies on high-quality code generation for its performance. Whether this is sufficient to attain the performance needed for FM reception is the third question to answer in the present research.

Thus, our contributions are:

1. The design and implementation of windowing and FIR (Finite Impulse Response) filtering, with predictable and high performance: §3. Our design aimed to exploit static information (window size, sliding amount, padding) and to lend itself to automatic vectorization. The implementation uses the existing strymonas interface as is, thus confirming its expressivity – which also ensures the complete fusion by construction.
2. The implementation of a realistic, practical application – FM Radio reception – and its evaluation/macro-benchmarking: §4. We have actually implemented three related applications:
   (a) using the high-level algorithm from StreamIt: §4.1. StreamIt is a synchronous dataflow system whose design and operation differ significantly from strymonas. Nevertheless we are able to implement its high-level algorithm in strymonas, and confirm the correctness. We also check the performance against a StreamIt reference implementation: hand-written C code.
   (b) using the high-level GNU Radio algorithm based on quadrature frequency demodulation: §4.2. (To remind, in design and operation, GNU Radio and strymonas differ significantly.) We have benchmarked strymonas on an Intel (macOS) platform and verified that we attain the GNU Radio performance. We then confirmed that our implementation (using the HackRF One board for signal acquisition and ffplay for play-out) indeed achieves the real-time reception of a local FM radio station: §4.3.
   (c) To verify the portability of our implementation in (b) we recompiled it on the relatively low-end Raspberry Pi Zero, and repeated the benchmarks, thus confirming the competitive performance of strymonas. Although barely, we managed to achieve real-time FM reception with strymonas on that platform.

We start by recalling the background: the strymonas library and software-defined radio.

The code for the benchmarks, in particular, StreamIt benchmark and the code for the running example, is freely available in the strymonas repository http://strymonas.github.io/.

2 Background: Strymonas on an SDR Example

Strymonas is a DSL that generates high-performance single-core stream processing code from declarative descriptions of stream pipelines and user actions – something like Yacc. Unlike (ocaml)yacc, strymonas is a DSL embedded in OCaml.[4] Therefore, it integrates as is with the existing OCaml code and tools. Any typing or pipeline mis-assembling errors are reported immediately (even during editing).

We illustrate strymonas on the running example chosen to show off its facilities, and also illustrate DSP. One may regard the example as a simple but representative example of radio processing – specifically, AM Radio. The full code of the example is included in the strymonas repository.[5]

Radio, originally, is transmitting audio over long distances. The audio (also called message) signal has relatively low frequency and, hence, propagates poorly and is difficult to effectively transmit and receive. Therefore, radio uses a high-frequency carrier wave, whose parameters (amplitude, phase) are the function of (i.e., modulated by) the message signal

[1] https://wiki.gnuradio.org/

[2] see, for example, https://github.com/gnuradio/gnuradio/issues/6103#issuecomment-1279650539

[3] https://hackrf.readthedocs.io/en/latest/index.html

[4] There is also a Scala version.

[5] http://strymonas.github.io/

[6]. Our first task then is to generate the carrier wave: to build a sine wave generator.[6]

```
let sine_wave (F:int) (f:float) : F32.t cstream =
  iota (C.int 0) ▹
  map F32.(fun i → of_int i /. lit (Float.of_int F)) ▹
  map F32.(( *. ) (lit Stdlib.(τ *. f))) ▹
  map F32.sin.invoke
```

where F is the sampling rate (Hz, or samples per second), f is the frequency (Hz), and τ is 2π. (For clarity, we explicitly write type annotations although none are needed.)

Like Yacc, strymonas uses two languages: one to describe the structure of the stream pipeline, and the other to specify the semantic actions such as mapping transformations, etc. Since strymonas is an embedded DSL, both languages are represented by OCaml functions (combinators), but from two different namespaces (signatures). Stream structure combinators such as iota and map, Fig. 1, produce, consume, or transform values of the type α cstream, where α is a base type (α cstream is a convenient specialization of the more general α stream, where α can be a tuple, structure, etc. type). The iota combinator produces an infinite stream of consecutively increasing integers starting with the given one; map should be self-explanatory. The combinators are typically composed via ▹: the right-to-left application operator.

Semantic actions (i.e., the arguments of stream combinators) are described via backend combinators, Fig. 2, which build values of the abstract type α exp representing the expressions and α stm for statements of the target code: C, OCaml, etc. A backend hence is a first-order statement-oriented imperative language with mutable references and arrays. We shall assume one such backend in scope, as the module named C. It has a submodule F32 for short-float expressions of the target language, of the abstract type F32.t (we abbreviate C.F32 as F32). In contrast, Stdlib and its submodule Float refer to the OCaml standard library; therefore, their operations are performed at the code-generation time, with the result becoming the literal number in the backend code (F32.lit operation).[7] The notation F32.(*exp*) is the abbreviation for let open F32 in *exp*, that is, making F32 operations available within *exp* without qualification.

The code sine_wave clearly corresponds to the mathematical expression $\sin(2\pi f i/F)$, for $i = 0, 1, \ldots$, but written in the left-to-right information-flow style common in electrical engineering.

[6]The left-associative infix operator ▹ of low precedence is the inverse application. The related right-associative infix operator @@ of low precedence, appearing later, is application: x + 1 ▹ f is the same as f @@ x + 1 and is the same as f (x + 1) but avoids the parentheses.

[7]In particular, in F32.of_int i, i is a statically unknown backend integer value; whereas in F32.lit (Float.of_int j), j is the statically known OCaml integer.

```
type α stream
type α cstream = α exp stream
type α emit = (α → unit stm) → unit stm

val iota : int exp → int cstream
val from_to : int exp → int exp → int cstream
val of_arr : α arr → α cstream
val infinite : α emit → α stream
val initializing_ref : ζ exp →
                       (ζ mut → α stream) → α stream

val map : (α exp → β exp) → α cstream → β cstream
val flat_map : (α exp → β stream) →
               α cstream → β stream
val map_accum :
  (ζ exp → α exp →
    (ζ exp → β exp → unit stm) → unit stm) →
  ζ exp → α cstream → β cstream
val map_raw : ?linear:bool → (α → β emit) →
                α stream → β stream
val zip_with : (α exp → β exp → γ exp) →
               (α cstream → β cstream → γ cstream)

val iter : (α → unit stm) → α stream → unit stm
```

Figure 1. Stream combinators (abbreviated and simplified for presentation). Types exp, stm, arr and mut refer to expressions, statements, arrays and mutable references of the target code, see Fig. 2. In map_raw, the notation ?linear:bool means an optional boolean argument. If not specified, the default values (true in this case) is used.

AM medium-wave radio has the frequency band 531–1,602 kHz with 9 kHz spacing. Let's then make our carrier at AM 540 (kHz):

```
let carrier : F32.t cstream = sine_wave F_c 540_000.
```

where the sampling rate F_c is 3,360 kHz (a bit more than twice the highest frequency in the medium-wave band).

We take the message signal to be of unit amplitude and given as an array (which may be an mmap-ed array):

```
let message : F32.t C.arr → F32.t cstream = of_arr
```

It is a low-frequency signal, and hence has a much lower sampling rate, F_m, which we take to be 48 kHz: the standard DVD/HDMI sampling rate.[8]

The carrier and the message signals are sampled at different rates. Before attempting modulation, we have to bring

[8]This sampling rate is an overkill for the real AM Radio, which does not possess bandwidth for the DVD-quality transmission. Amplitude modulation, however, is used in other contexts, as we shall see in §4.2.

```
type α exp          (* Abstract type of expressions *)
type α stm          (* Statements *)
type α arr          (* Arrays *)
type α mut          (* Mutable references *)
type α tbase        (* Type representation for base types *)

val tint : int tbase
val int   : int → int exp
val ( + ) : int exp → int exp → int exp
...

(* sequencing *)
val (@.) : unit stm → α stm → α stm

val cond : bool exp → α exp → α exp → α exp
val if_ : bool exp → unit stm → unit stm → unit stm

(* let-binding *)
val letl : α exp → ((α exp → ω stm) → ω stm)

val dref : α mut → α exp
val (:=) : α mut → α exp → unit stm
val incr : int mut → unit stm
val decr : int mut → unit stm

(* foreign function type: backends for the constructors *)
type 'sg ff = private {invoke : 'sg}

module F32 : sig    (* short floats *)
  type t
  val tbase     : t tbase          (* type representation *)
  val lit       : float → t exp
  val of_int    : int exp → t exp
  val ( +. )    : t exp → t exp → t exp
  val ( *. )    : t exp → t exp → t exp
  val ( /. )    : t exp → t exp → t exp
  val sin       : (t exp → t exp) ff
end
```

Figure 2. Backend code-generating combinators for the target language (abbreviated). In the present paper, they are assumed to be in a module named C. We abbreviate C.F32 as F32.

them to the same rate: that is, upsample the message signal. We use for the sake of example the simplest left-neighbor upsampling. For higher fidelity one would use linear, etc. upsampling – which are actually a form of filtering, to be discussed in detail later.

```
let upsample (F_s:int) (F_d:int) : α stream → α stream =
  let k = F_d / F_s in assert (k > 1 && k * F_s = F_d );
  flat_map (fun s → from_to C.(int 0) (C.int (k - 1)) ▹
                  map (Fun.const s))
```

The left-neighbor upsampling by the factor k is thus mapping each sample s into a finite stream (chunk) of length k containing just the sample s, and concatenating all these chunks. The mapping-concatenating is also called flat-mapping. A chunk is created from a 0..k−1 stream (build by from_to) by replacing each element with s. Although intuitive, the whole pipeline (especially mapping by const s) seems quite naive – enough to start worrying about performance. As we shall see later, the worry is unfounded. The strength of strymonas is that pipelines may be naive, but still fully fused and performant.

The amplitude modulation is defined as $(1 + \mu m(t))c(t)$ where $m(t)$ is the message signal, $c(t)$ is carrier, and μ (chosen to be close but less than 1) is the modulation index.

```
let am_modulate (ms:F32.t cstream) (F_m:int)
    (cs:F32.t cstream) (F_c:int) (μ:float) : F32.t cstream =
  zip_with (fun m c → F32.((lit 1. +. lit μ *. m) *. c))
    (upsample F_m F_c ms) cs
```

The combinator zip_with performs a binary operation (specified by its first argument) on two streams elementwise.

The pipeline is completed by terminating it by a stream consumer, such as fold, reducing the stream to a single value. Such full accumulation is rare in signal processing, however. More common is outputting the processed signal samples: to be stored in a file, sent over the network or pushed to a sound card. For output, we rely on the existing backend function (which could be inlined), invoked via the FFI:

```
val write_s16_le : (F32.t C.exp → unit C.stm) C.ff
```

It writes a signed 16-bit little-endian integer to the standard output, which can be pipelined to a tool like HackRF for converting to an analog signal and transmitting.

Putting it all together while adding gain of 30000 gives

```
let ammod =
  C.one_arg_fun @@ fun arr →
  am_modulate (message arr) F_m carrier F_c 0.9 ▹
  map F32.( ( *. ) (lit 30000.)) ▹
  iter write_s16_le.invoke
```

The code can be improved, which gives us the chance to introduce further strymonas facilities. Recall, sine_wave maps sin over the infinite stream of (scaled) samples 0,1,.... There is a real danger that samples eventually overflow the 32-bit integer counter. Applying sin to progressively bigger and bigger arguments seems like a waste. A bit more sophisticated generator solves both problems:

```
let sine_wave (F:int) (f:float) : F32.t cstream =
  let δ = (τ *. f) /. float_of_int F in
  infinite (fun k → F32.lit δ ▹ k) ▹
  map_accum F32.(fun acc' s k →
    let- acc = C.letl (acc' +. s) in
    let- acc = C.letl (C.cond (acc < lit τ)
                              acc (acc -. lit τ)) in
    k acc acc') (F32.lit 0.) ▹
  map F32.sin.invoke
```

Here, infinite creates an infinite stream of δ (phase change per sample) and map_accum integrates this stream while keeping the samples within $0..\tau$. At the end, we could have used a more efficient version of sin that takes advantage of that fact. The function map_accum is the first demonstration that strymonas streams may keep their own state. The so-called let-operator let- is defined as

```
let (let-) c k = c k
```

It makes the backend operation C.letl for local bindings in the target language look like a let-expression.

Strymonas has several backends to emit code in OCaml, Scala and C – each statically guaranteeing the generated code compile without errors. In this paper we use the C backend, based on embedding of C in tagless-final style described in detail in [2]. The generated C code for modulation (with the more sophisticated sine-wave generator) is as follows:

```
void ammod(int const n_1, float * const a_2) {
  float x_3 = 0.;
  for (int i_4 = 0; i_4 < n_1; i_4 += 1) {
    float const t_5 = a_2[i_4];
    for (int i_6 = 0; i_6 < 70; i_6 += 1) {
      float const t_7 = x_3;
      float const t_8 = t_7 + 1.0097976;
      float const t_9 = (t_8 < 6.2831853 ?
                          t_8 : t_8 - 6.2831853);
      x_3 = t_9;
      float const t_10 = sinf(t_7);
      float const t_11 = 30000. * ((1. + (0.9 * t_5)) * t_10);
      write_s16_le(t_11);
    }
  }
}
```

This is what a competent programmer would have written by hand. Although the pipeline is purely declarative, with first-class (the argument of flat_map and zip_with) and higher-order functions the generated code is imperative, with no closures. We have built the AM modulation pipeline using map as we pleased (recklessly, one may say), as well as the complicated-looking combinators like map_accum and flat_map. The generated code, however, is as simple as it gets, with no overhead. Map fusion should be expected of any stream processing library. Strymonas, however, performs fusion in all other cases, some of which are rather challenging (such as involving both zipping and flat-map).

The reverse process, demodulation, cuts off the negative portion of the signal (so-called diode, or envelop detection) and downsamples it:

```
let decimate (n:int) : α stream → α stream = fun st →
  assert (n > 1);
  let skip = n - 1 in
  let- z = initializing_ref C.(int skip) in
  st ▹ Raw.map_raw ~linear:false
    C.(fun e k →
      if_ (dref z ≤ int 0) ((z := int skip) @. k e) (decr z))

let demodulate (k:int) : F32.t cstream → F32.t cstream =
  map F32.(fun x → C.cond (x > lit 0.) x (lit 0.)) ▹
  decimate k
```

(where ▹ is the left-to-right function composition). The function decimate is the left inverse of upsample, keeping every $n-1$-th sample of the input. The state of the stream is now very explicit. All streams in strymonas are stateful; programmers may introduce their own state, but it must be declared, using initializing_ref. The combinator map_raw in a lower-level (internal) strymonas interface is a more general version of map that permits skipping of samples (in which case the optional argument ~linear has to be specified, as false[9]).

More examples can be found in the strymonas repository; particularly relevant to signal processing are run-length-encoding and decoding.

3 Windowing

Whereas the AM radio transmission pipeline described in §2 was complete, the reception was not. First, the signal acquired from the antenna contains transmissions from all sources (radio, TV, cellular) plus the environment noise. Therefore, before demodulation we have to remove all signals but the one within the transmission band of the desired station. The demodulated signal still contains a high-frequency component, which would produce audible noise after decimation due to aliasing. It has to be filtered out. Filtering is the fundamental operation of DSP.

GNU Radio and StreamIt use so-called FIR filters – which is what we adopt as well. FIR (Finite Impulse Response) filters have the following general form:

$$y_i = \sum_{k=0}^{m-1} w_k \, x_{i-k}$$

[9]which means that that the stream may advance its state but produce no sample. Such non-linear streams are a special case upon zipping, see [3] for detail.

In other words, a sample of the output signal y_i is a weighted sum of the m most recent input signal samples $x_i, x_{i-1}, \ldots, x_{i-m+1}$. The weights w_k are the filter coefficients. The number $m-1$ is called the order of the filter, and m itself is called the number of taps. See [9] for much more detail.

Filtering is clearly a form of window processing: viewing a potentially infinite stream through a finite window of the most recently seen elements. The window may slide over the stream, or tumble over (i.e., move without overlapping). Strymonas provides only the most basic stream operations: map, filter (not to be confused with filtering as used in DSP), zip, flat_map. The limited set of operations is chosen so to support the complete fusion. The richer interface in Fig. 1 is implemented in terms of the basic. This section shows that windowing is also implementable in terms of the basic operations, and hence enjoys complete fusion by construction. We will further see that windowing code is vectorizable.

To achieve high performance, we rely on the general principle of partial evaluation: exploiting the available static information. This is often easier said than done – especially if we want assured performance. The challenge is to identify the most useful static information, and to design the interface to easily, hopefully 'naturally', specify it without burdening the user.

Our first design decision is decomposing the filtering into the creation of a window and its reduction to a single value, per the following schema to be made concrete later:

```
val make_stream : F32.t exp stream → Win.t stream
let fir_filter weights : F32.t cstream → F32.t cstream =
  make_stream ⊳ map (dot weights)
```

With the guaranteed complete fusion, such a decomposition has zero run-time cost. Here make_stream converts a stream of floats to a stream of windows. A window, of the type Win.t, is a representation of the finite history of the original stream. The operation dot weights: t → F32.t exp performs the dot-product of the window with the given array of weights. This decomposition separates the tasks of maintaining the window (stream history) and using the history – making it easier to write and understand the code. It also lets us easily develop other operations on windows.

The generated code deals with integers, floats, etc. Strymonas, however, lets us pretend that stream elements may be composite: tuples, records, etc. Such structured elements are exclusively the generation-time abstraction. The generated code does not keep creating and disposing tuples or records per element: we are statically assured of that because the backend, Fig. 2, simply does not have facilities to build tuples of records.

Concretely, the window interface (the type t and the operations on it) is presented in Fig. 3. The window type is abstract, which allows for different concrete implementations. The operation reduce ⊙ reduces the window with a user-specified associative operation ⊙. That is,

```
module type window = sig
  type e                          (* element type: base type *)
  type t                          (* window type *)
  val etyp : e tbase              (* type descriptor of e *)
  val size : int                  (* statically known, >0 *)
  val make_stream : e exp stream → t stream

  val reduce : (e exp → e exp → e exp) →
            t → (e exp → ω stm) → ω stm

  val dot : α tbase → (β → α exp) → β array →
          (e exp → e exp → e exp) →
          (α exp → e exp → e exp) →
          t → (e exp → ω stm) → ω stm
end

type α window = (module window with type e = α)
(* make a window with the given element type,
   size and slide *)
val make_window : α tbase → int → int → α window
```

Figure 3. Windowing interface

```
make_stream ⊳ map_raw (reduce ⊙)
```

effectively computes $x_i \odot x_{i-1} \odot \ldots \odot x_{i-m+1}$ where m is the window size. As the name implies, dot trep lift $w \oplus \otimes$ computes the dot-product of the window $[x_i, \ldots, x_{i-m+1}]$ with the given array w, with $\oplus$ and $\otimes$ as the additive and multiplicative operations. Therefore,

```
make_stream ⊳ map_raw (reduce trep lift w ( + ) ( * ))
```

performs the convolution, that is, FIR filtering. The weights are specified as a β array: an OCaml rather than the target language array. In other words, the interface requires the weights be statically known (as is usually the case in DSP); an implementation can take advantage of it.

Since the window type t is abstract, there may be different realizations of it. The concrete instances of the window interface are created by make_window, which takes the element type descriptor trep, the window size and slide, and returns the first-class module of the window interface. Depending on size and slide, make_window returns different window implementations, optimized for these parameters. For example, when the size is two, the implementation (slightly abbreviated) is

```
type t = {prev: e mut; curr: e exp}

let make_stream : e exp stream → t stream = fun st →
  let– inited = initializing_ref (C.bool false) in
  let– prev   = initializing_ref (C.tbase_zero etyp) in
```

```
   st ▹ map_raw ~linear:false C.(fun e k ⟶
     if_ (dref inited) (
       k {prev; curr=e} @.
       if slide then prev := e
       else inited := bool false
     ) (
       (prev := e) @.
       (inited := bool true)
     ))

let dot _typ lift weights op mul {prev;curr} =
   assert (Array.length weights = 2);
   C.(op (mul (lift weights.(0)) curr)
         (mul (lift weights.(1)) (dref prev))) ▹
   C.letl
```

Since the window is short, it is represented by the current element and the reference cell holding the previous element. The boolean slide tells if the window is sliding, which affects the generated code. This is an example of a conditional code generation.

One may notice the assert statement, which checks that the array of weights has two elements (and hence matches the window size). The assertion failure is a run-time error – in the *generator* rather than the generated code. It may be raised when generating DSP code. If the code is successfully generated, the check must have passed and the code relies on the invariant without further run-time checks and errors. In effect, code generation lets us achieve a form of dependent typing: see [4] for more detailed discussion.

As an example, the following example code builds the stream of differences between consecutive input samples: $x_i - x_{i-1}$. The input is given as an array a:

```
of_arr a
 ▹ Win.make_stream
 ▹ map_raw F32.(Win.dot tbase lit [|−1.;1.|] ( +. ) ( *. ))
 ▹ iter write_s16_le.invoke
```

The generated C code shows the expected complete fusion:

```
void fn(int const n_41, float * a_42) {
  bool x_43 = false;
  float x_44 = 0.;
  for (int i_46 = 0; i_46 < n_41; i_46 += 1) {
    float const t_47 = a_42[i_46];
    if (x_43) {
      float const t_48 = (−t_47) + x_44;
      write_s16_le(t_48);
      x_44 = t_47;
    } else {
      x_44 = t_47;
      x_43 = true;
    }
  }
}
```

The function make_window has many more implementations: for tumbling windows, for windows sliding by 1, for large windows, etc. One can see them all in the source code available in the strymonas repository. Below we show only one example: low-pass filtering to use after demodulation. It is produced by the following fir_filter (this is now the concrete code rather than the schema):

```
let fir_filter ?(decimation=0) (weights:float array) :
    F32.t cstream ⟶ F32.t cstream =
  let ntaps = Array.length weights in
  let (module Win) =
      make_window F32.tbase ntaps (decimation+1) in
  Win.make_stream
  ▹ map_raw F32.(Win.dot tbase lit weights ( +. ) ( *. ))
```

Decimation (see decimate in §2) is integrated with filtering: indeed, it is effected by sliding the window by a larger amount. When we pass to fir_filter the weights for a 64-tap low-pass filter (and complete the pipeline) we obtain the following C code:

```
int x_3 = 0;
int x_4 = 62;
float a_5[126] = {0.,0.,...};
float const t_17 = /* current sample */
if (x_3 < 63) {
  x_4 = (x_4 == 0 ? 62 : x_4 − 1);
  (a_5[x_4]) = t_17;
  (a_5[x_4 + 63]) = t_17;
  x_3++;
} else {
  static float a_18[64] =
   {1.5361169e−06,1.5800085e−06,...};
  float x_19 = (a_18[0]) * t_17;
  for (int i_20 = 0; i_20 < 63; i_20 += 1)
    x_19 = x_19 + ((a_18[i_20 + 1]) * (a_5[x_4 + i_20]));
  write_s16_le(x_19);
  x_4 = (x_4 == 0 ? 62 : x_4 − 1);
  (a_5[x_4]) = t_17;
  (a_5[x_4 + 63]) = t_17;
}
```

The loop has simple boundaries (in fact, constants) and its indexing expressions are all simple: the loop variable with an offset. Such loops are easily vectorizable – and indeed vectorized by GCC, on an Intel x86-64 platform (given options: -O3 -march=native -mfpmath=both -ffast-math), as the following GCC-generated assembly code demonstrates.

```
vmulps  32(%r10), %ymm14, %ymm3
vmulps  96(%r10), %ymm12, %ymm15
vmovups 64(%r10), %ymm17
vmovups 128(%r10), %ymm18
decl    %edi
vfmadd231ps     (%r10), %ymm13, %ymm3
vfmadd231ps     .LC17(%rip), %ymm17, %ymm15
vaddps  %ymm15, %ymm3, %ymm3
vmovaps .LC18(%rip), %ymm15
vmulps  160(%r10), %ymm15, %ymm15
vfmadd231ps     .LC19(%rip), %ymm18, %ymm15
vaddps  %ymm15, %ymm3, %ymm3
```

The loop has been completely unrolled. We also see packed (SIMD) addition and multiplication vaddps, vmulps, and the fused multiply-add vfmadd231ps.

4 Evaluation

Using the just presented windowing and filtering operations, we have implemented FM radio reception applications and used them as macro-benchmarks to evaluate strymonas, contrasting it with StreamIt and GNU Radio.

All evaluation, on Intel platform, has been performed on 1.8GHz dualcore Intel Core i5, 8 GB DDR3 main memory, macOS Monterey 12.7. Strymonas-generated C code was compiled by GCC version 13.2.0. GNU Radio C/C++ code was compiled by Clang version 14.0.0. The GNU Radio version is 3.10.7.0.

4.1 StreamIt Benchmark

We first evaluate and contrast strymonas with the synchronous dataflow system StreamIt [11–13] using FM radio reception as a benchmark.

Like the AM radio described in §2, FM radio also modulates the high-frequency carrier with the message signal. However, rather than affect the amplitude of the carrier, it affects the frequency. That is, the instantaneous frequency of the output $f_i(t)$ is the deviation from the carrier frequency f_c by the message signal $m(t)$:

$$f_i(t) = f_c + \Delta f m(t)$$

where Δf is the peak frequency deviation; its ratio to the message signal frequency (or its sampling rate) is called the modulation index. The output signal is hence

$$\sin\left(\tau f_c t + \tau \Delta f \int_0^t m(t) dt\right)$$

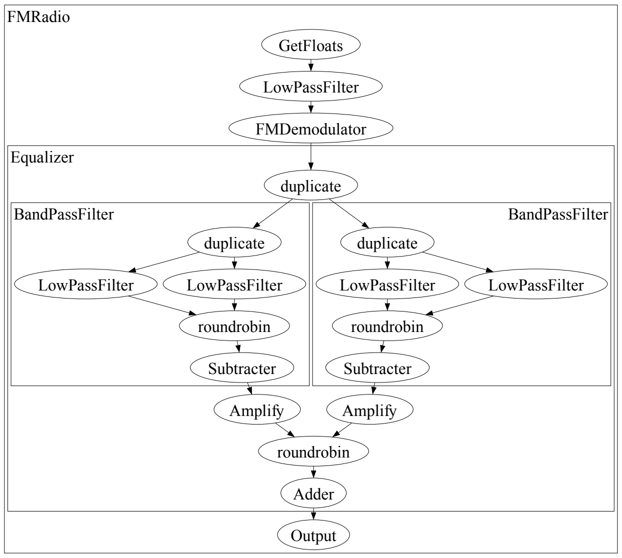

Figure 4. StreamIt's FM radio reception diagram

Demodulation therefore involves a form of differentiation, followed by the envelop detection.[10] For the sake of comparing with StreamIt, we used the same demodulation algorithm, which is expressible in strymonas as:

```
let fmDemodulator (F:float) (M:float) (b:float) :
        F32.t cstream ⟶ F32.t cstream =
  let gain = F32.lit (M *. (F /. (b *. π))) in
  let (module Win) =
      Window.make_window F32.tbase 2 1 in
  Win.make_stream
  ▷ map_raw (Win.reduce F32.( *. ))
  ▷ map_raw C.(fun e ⟶ letl F32.(gain *. atan.invoke e))
```

where M is the the max amplitude of the output, b is the bandwidth and F is the sampling rate. Demodulation is too a form of window processing.

Overall, FM radio reception is the filtering to select the suitable band, demodulation and another filtering to remove noise and improve the quality of the resulting audio[11] – as diagrammed in Fig. 4 borrowed from [12].

For meaningful comparison, we implement the same algorithm with the same parameters and the set-up. There is an immediate problem however: strymonas does not support split/join (or, duplicate/join) operations apparent in Fig. 4 in equalization. Equalization is splitting the signal into several bands, amplifying by a band-specific gain and re-combining. This looks far more complex than LowPassFilter. However, FIR filters are linear: a sum or a difference of FIR filters is

[10]Differentiation also amplifies high-frequency noise, and hence degrades the signal-to-noise ratio of the audio signal at higher frequencies. To compensate, broadcasters perform so-called pre-emphasis, artificially boosting the high-frequency part of the message signal.
[11]and also to compensate for the pre-emphasis

also a FIR filter (with added or subtracted weights, resp.). The entire equalizer block in Fig. 4 is hence reducible to a single FIR filter, which is what we use.

Thus the FM reception is literally the following three lines:

fir_filter (lowPassFilter F f_{cut} m) ~decimation:4
▷ fmDemodulator F M b
▷ fir_filter (equalizer F bands eqCutoff eqGain m)

where the sampling rate F (set to 250 MHz), the cut-off frequency f_{cut} (108 MHz), the bandwidth b (10 kHz), the number of taps m (64), etc., are the standard FM radio parameters. Here lowPassFilter F f_{cut} m : float array is a pure OCaml (that is, with no relation to code generation) function that computes the coefficients (the weights) of the low-pass filter with the given parameters. Likewise, equalizer computes the weights of the fused equalization filters.

Our first comparison with StreamIt is checking that we reproduce its input/output behavior. We wrote a naive (and hence obviously correct) StreamIt interpreter and used it on the StreamIt FM benchmark code[12] to confirm that the interpretation of every step of the diagram in Fig. 4 gives the same, up to numeric precision, output as the strymonas implementation (on the same inputs): the maximum relative error for the whole pipeline is 3e-7 (in particular, 6.6e-7 for the low-pass filter and 5.3e-7 for the equalization, when compared as independent steps.) We remind that the code uses 32-bit floats. Thus, despite the completely different design (and the different implementation of the equalizer, for one), strymonas reproduces the input/output behavior of the StreamIt FM reception program. As an additional check of correctness, we generated a sample FM radio signal by modulating a sine wave, fed into the strymonas implementation and checked the result by ear.

The FM benchmark of StreamIt includes the reference C code:[13] a hand-written and optimized implementation of the StreamIt pipeline. It is also single-threaded, so that the comparison with strymonas (on the same synthetic input) is meaningful. Fig. 5 shows the results: the processing time of 1 million synthetic samples, measured as an average of 20 runs after 5 warmup runs, on the platform described earlier. We compiled both StreamIt C code and strymonas-generated C code using GCC given the flags "-O3 -march=native -mfpmath=both -fno-math-errno" (to be called F1) or with "-ffast-math" added (to be called F2).

We also checked the memory profile with valgrind. Both StreamIt and our code showed constant memory use throughout the entire processing, with no heap allocation. However, whereas the former used 590 KiB of stack, our code needed 1 KiB (for flags F1) or close to zero (for flags F2). One may perhaps lower the memory usage of the StreamIt reference

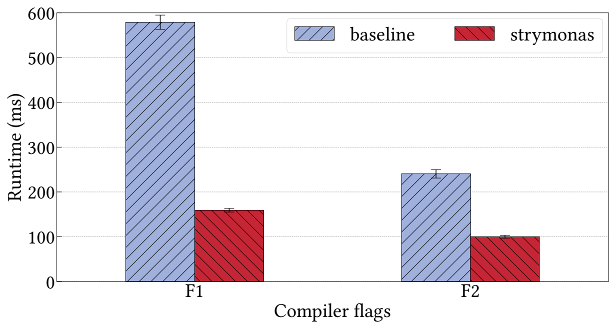

Figure 5. FM reception Benchmark of strymonas-generated C code against the baseline: StreamIt hand-written reference C code. The graph shows processing time (in milliseconds, lower is better) of 1 million synthetic samples, with 95% CIs. See text for the explanation of different compiler flags.

code by tuning its parameter IN_BUFFER_LEN, which however may affect performance or correctness. Since it is not our code, we left its parameters as they were set.

All in all, C code generated by strymonas from declarative pipelines shows competitive performance. We now check and compare performance against the state of the art: GNU Radio, on the real FM radio application.

4.2 GNU Radio Benchmark

GNU Radio is the state-of-the-art in SDR: a massive toolkit with many components to assemble a wide variety of SDR applications, including FM reception with HackRF One board as input. In fact, an instance of FM reception appears as the first example in the GNU Radio Beginner Tutorial.[14] The main research question is whether strymonas is able to approach or attain the GNU Radio performance on the example of FM reception.

HackRF One, as many other boards to acquire and digitize radio signals uses the so-called quadrature demodulation [7]. Quadrature (de)modulation is the technique based on the observation that any sort of modulation can be represented as an amplitude modulation of *two* carrier waves, of the same frequency and amplitude but shifted by 90° in phase, which are then added up. The two carriers (typically called I and Q) are the real and imaginary components of the complex carrier $e^{i\tau f_c t}$, and 'modulating the carriers and adding them up' is in fact complex multiplication. Hence the amplitude modulation with the message signal $m(t)$ is $e^{i\tau f_c t} \times (1+\mu m(t))$ and the frequency modulation is $e^{i\tau f_c t} \times \exp\left(i\tau\Delta f \int_0^t m(t)dt\right)$. The HackRF One board performs demodulation: in effect divides the received signal by $e^{i\tau f_c t}$ and returns the result as a stream of complex numbers.[15] Amplitude, frequency or other demodulation are performed on this stream. The

[12] http://groups.csail.mit.edu/cag/streamit/apps/benchmarks/fm/streamit/FMRadio.str
[13] http://groups.csail.mit.edu/cag/streamit/apps/benchmarks/fm/c/fmref.c
[14] https://wiki.gnuradio.org/index.php?title=What_Is_GNU_Radio
[15] Although obvious, somehow neither various tutorials and textbooks like [7], nor the HackRF documentation put it this way.

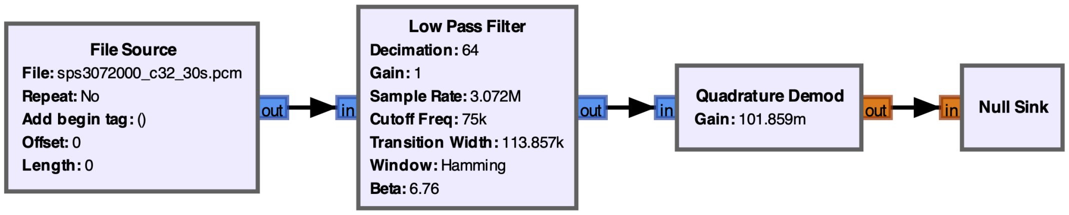

Figure 6. GNU Radio FM reception diagram, used as a benchmark

output from HackRF is therefore lower in frequency than the carrier signal – but still tens of times higher than the audio frequency, and further decimation is needed.

A typical FM reception in GNU Radio is described by the diagram in Fig. 6 – which is an instance of the diagram from the GNU Radio tutorial, but adjusted for much higher sampling sampling and wide-band FM. It also matches the StreamIt diagram, Fig. 4: filtering and demodulation. Since the source is quadrature (complex-number) samples, however, the filtering and demodulation is now in complex domain. That is not a problem for strymonas: its backend also has the interface C32, quite similar to F32 on Fig. 2, for 32-bit complex numbers (that is, pairs of 32-bit floating point-numbers representing the real and imaginary parts of a complex number). The FM reception proper, the central part of Fig. 6, is expressed in strymonas rather literally, as:

```
fir_filter_ccf (Fir.lowPassFilter 1. $F_{iq}$ $f_{cut}$ $f_{tr}$)
                          ~decimation:64
▷ demod_quad_fm (($F_{iq}$ /. 64.) /. ($\tau$ *. $\Delta f$))
```

where fir_filter_ccf is the C32.t cstream → C32.t cstream analogue of fir_filter in §3, using the operations of C32 rather than F32. The pure OCaml function Fir.lowPassFilter computes the weights as a float array, using the same algorithm as GNU Radio's and the same parameters: gain 1., sampling rate F_{iq} of 3.072 MHz and the transition width f_{tr} of 114 kHz (which determines the number of taps in the filter: 65). The peak frequency deviation Δf (75 kHz) and the cut-off frequency f_{cut} of 75 kHz are the parameters of FM Broadcasting. Down-sampling F_{iq} by 64 gives the output rate of 48 kHz, which is the standard DVD/HDMI audio sampling rate F_m.

The quadrature frequency demodulation code is as follows:

```
let demod_quad_fm (gain : float) :
    C32.t cstream → F32.t cstream =
  let (module Win) =
      Window.make_window C32.tbase 2 1 in
  Win.make_stream
  ▷ map_raw (
     Win.reduce C32.(fun $x_i$ $x_{i-1}$ → $x_i$ *. conj $x_{i-1}$))
  ▷ map F32.(fun e →
     lit gain *. fast_atan2f.invoke C32.(imag e) C32.(real e))
```

It takes quadrature (complex-number) samples but produces real-value samples (which can then be fed to an audio playback program such as ffplay).

One of the slowest operations in the FM pipeline is the arc tangent computation (atan2f) during demodulation. The CPU cycle budget per sample is tight already, but computing arc tangent takes many cycles and involves switching to FPU (all other floating-point operations in the pipeline are performed in xmm registers, on the SSE/AVX unit). Therefore, GNU Radio implements a simplified atan2f, using table look-up with linear interpolation and under the assumptions of –ffast–math (that is, assuming all floating-point numbers normal, and zero unsigned). These are all sound assumptions common in DSP: after all, samples are inherently noisy, and high-precision is not necessary. The GNU Radio implementation of fast_atan2f leaves however much room for improvement. Using the same basic idea, we re-implemented it from scratch. The average error of our approximate fast_atan2f (compared to atan2f of the C standard library) is 5e−7 with the maximum relative error of 4e−6. Looking at the GCC-compiled code on an Intel x86-64 platform, we see only xmm registers and even occasionally packed (vector) operations.

Typically input comes from a board like HackRF One. For reproducibility of the benchmark, however, we use a pre-recording: 30 seconds of broadcast from a local station 82.5 FM, acquired using HackRF/ffmpeg and saved to a file. With the sampling rate F_{iq} of 3.072 MHz, the file contains 92.16 million 32-bit complex samples; the file size is hence 737.28 MB. The output of the benchmark is the null sink (which for strymonas means calling a dummy function for each output sample, with no buffering).

Fig. 7 presents the results of the benchmark.[16] The input is the file of pre-recorded samples. There are several ways to access them however. One can pre-load the whole file into a pre-allocated array and process it in-memory (pre_load in Fig. 7). For GNU Radio, the array is used through the standard Vector Source block. Alternatively, one can incrementally load and process the file (inc_load in Fig. 7): in the case of GNU Radio, using the standard File Source block; in the case

[16]For GNU Radio we measured the runtime from gr::top_block::start() to gr::top_block::wait() by monotonic clock in the main thread while the top_-block object had already been created.

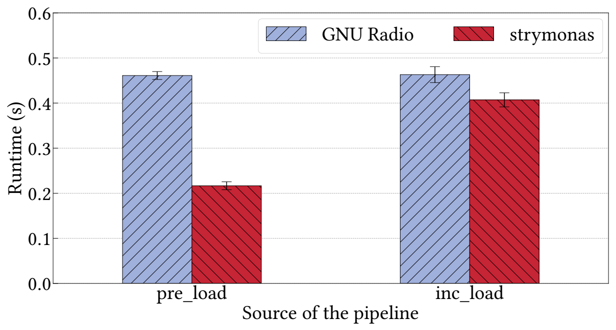

Figure 7. GNU Radio FM reception benchmark: processing time (in seconds, lower is better) of a pre-recorded 30-second FM signal, with 95% CIs. GNU Radio is the baseline. See text for the explanation of different sources.

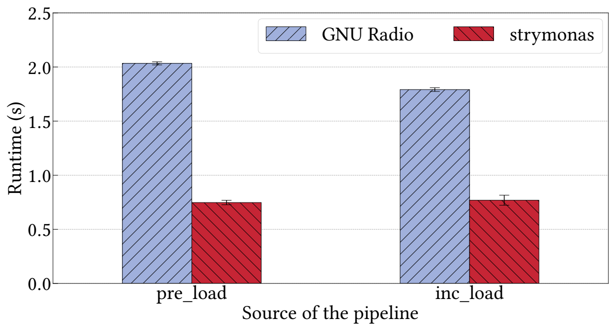

Figure 8. GNU Radio FM reception benchmark on Raspberry Pi: processing time (in seconds, lower is better) of a pre-recorded 3-second FM signal, with 95% CIs. GNU Radio is the baseline. See text for the explanation of different sources.

of strymonas, using the standard C library function fread (whose buffer holds 1024 eight-byte samples).

All the results are the averages over 20 runs after 5 warm-up runs. The platform is macOS as described earlier. GNU Radio code (which is C++) was compiled as using clang++ as recommended, and even necessary for the hand-tuned DSP kernels, using flags -Ofast -march=native. Strymonas-generated code, which is pure C, is compiled by GCC, with flags -Ofast -march=native. (The flag –Ofast implies –O3 and –ffast–math.)

The benchmark shows that even using the incremental loading with an inefficient and synchronous fread and the null sink with no blocking, strymonas is quite competitive with GNU Radio. The larger difference between memory pre-loading and incremental loading for strymonas compared to GNU Radio tells that the sample processing in strymonas is more efficient. §5 discusses the differences between GNU Radio and strymonas, which may help understand these results.

4.3 Real-time FM Radio Reception

Real-time FM reception is one of the "Hello world"–type applications for GNU Radio – which we have built, to confirm the set-up of GNU Radio and HackRF, on the macOS platform described earlier. Whether strymonas is capable of real-time reception is a research question.

Strymonas pipeline for real-time reception uses the same filtering and demodulation as that in the benchmark (Fig. 6), with the input coming from HackRF this time and the output being ffplay. The input is similar to inc_load for strymonas in the benchmark (fread whose buffer size is 1024), only instead of a file we read from a pipe connected to the hackrf_transfer tool. The sampling rate is 3.072 MHz. For the output, instead of a dummy function we call a function that writes a short float on the standard output with no explicit buffering (in binary little-endian format). The standard output is piped to ffplay.

The strymonas-generated code was compiled by GCC with flags -Ofast -march=native.

The playback was stable (no sound skipping) but noisy. We confirmed that this noise problem was not specific to strymonas or our approach: GNU Radio, with the same pipeline, gives playback of the same sound quality.

4.4 FM Radio Reception on Raspberry Pi

To check how portable strymonas performance is, we have repeated the GNU Radio benchmark and the real-time reception experiments on a low-end single-board computer Raspberry Pi Zero.

The board has the 1 GHz single-core 32-bit ARMv6 CPU (AArch32) and 512 MB memory. OS is Raspbian GNU/Linux 11 (bullseye). All C/C++ code was compiled by GCC version 10.2.1. The version of GNU Radio is 3.9.5.0: it is older than the version we used on macOS, but it is the last version of GNU Radio fully supported on 32-bit Raspbian as far as we can check. The strymonas-generated code and the GNU Radio code were compiled using the flags -Ofast -mfpu=vfp -mfloat-abi=hard -march=armv6zk -mtune=arm1176jzf-s.

Fig. 8 presents the results of the benchmark, which, as on macOS, used pre-recorded samples and the null sink. The file of the pre-recorded samples, pre_load, and inc_load are the same as those on the macOS platform. However, since the Raspberry Pi Zero has only 512 MB of memory, pre-loading the whole file in case of pre_load causes swapping, leading to a significant decrease in performance. Therefore, we used a 3-second initial segment of the file (9.216 million samples, 73.728 MB) for the Raspberry Pi benchmark.

The real-time FM strymonas reception used the HackRF One board source and the audio sink. Our Raspberry Pi runs in headless mode (no connected speakers), therefore we set up PulseAudio to transmit the output over a WiFi network to a macOS laptop for playback (using aplay, as in §4.3). This complicated play-out adds its own overhead and latency. The playback had same sound quality as in §4.3: a little unstable and noisy, but intelligible. Thus, although with some difficulty, we managed to use strymonas for real-time FM reception even on a lower-end Raspberry Pi. We also tried the real-time playback with GNU Radio with essentially the same pipeline: it was unstable, noisy, and unintelligible.

5 Related Work

StreamIt [11–13] is a synchronous dataflow programming language with static scheduling. Each operator in StreamIt has to specify how many stream elements it uses and produces. For example, demodulation 'pops' one element from the input stream, pushes one and peeks two: produces one element in the output per element of the input, within the size-2 window. From this rate information, the StreamIt compiler determines the execution schedule of operators and computes the necessary buffer space (besides performing a host of optimizations). Strymonas in contrast uses what amounts to a dynamic scheduling. StreamIt is designed for parallelism: split/join operations in Fig. 4 is a clear sign. Strymonas is intended for single-code single-thread processing. StreamIt and strymonas hence make very different trade-offs. Both are capable of implementing FM radio reception, as we have seen.

GNU Radio[17] is the state of the art in SDR, and is the most used SDR framework. It is a library of components ('blocks') for all common DSP operations (such as filtering, FFT, (de)modulation, wave generation) as well as display, acquisition and playback. The components can be wired together in a flowchart to build a complete DSP application. The wiring is performed by writing C++ code, or, more conveniently, in Python or via a graphical interface of GNU Radio Companion. GNU Radio uses a sophisticated dynamic scheduling of blocks using threads. The components are not fused, and hence incur the overhead of function calls, (FP) register saves and restores, as well as dynamic typing in some cases. They cannot be optimized together. GNU Radio hence relies on parallelism and its sophisticated scheduling for performance (as well as VOLK – hand-written and tuned processing kernels, on some platforms). Strymonas is single-thread single-core. It guarantees complete fusion: all processing occurs within one monolithic main loop and optimized as a whole by a compiler. Splitting a complicated processing step as a composition of smaller blocks has overhead in GNU Radio but no overhead in strymonas. Strymonas is also typed: all mismatches in interfaces are flagged as errors at the construction time. The generated code is statically guaranteed to be well-typed, and hence compiles without errors.

Ziria [10] is a standalone DSL for wireless system programming. Like strymonas it is structured as a typed higher-layer language for composing stream pipelines and a lower-level imperative language for describing operations on stream data. The latter is somewhat like C but restricted to ensure the generation of vectorized code. The Ziria compiler also implements a variety of domain-specific optimizations. Ziria is a stand-alone language, whereas strymonas is embedded. Therefore, strymonas takes the full advantage of its host language: its module system, standard library, etc. An example of it was using OCaml to compute weights of various filters. Embedding also gives conditional code generation 'for free'.

[17] https://github.com/gnuradio/gnuradio

6 Conclusions

We have implemented and evaluated a realistic application of the declarative stream processing library strymonas: FM Radio reception. In the process we demonstrated that windowing processing (underlying digital filtering) can be retrofitted into existing strymonas, without compromising its complete fusion guarantees and attaining high and portable performance (across Intel x86-64 and ARM AArch32 architectures).

The first lesson, on the concrete example of SDR, is that type-safety and declarative interface do not have to preclude high-performance. The second take-away message is that paying attention to the generated code and using specialization so to generate code that is easily vectorizable – as well the complete fusion provided by the strymonas – can compensate for single-core–only processing. Single- vs. multi-core processing is a trade-off. We have demonstrated how this trade-off can be evaluated within a particular area of real-time high-frequency signal processing such as FM radio reception.

As future work, we contemplate further back-ends (code-generation targets), in particular, Wasm and GPGPU. It is natural to consider infinite impulse response (i.e., feedback) (IIR) filters. An interesting research direction is high-level, (semi-) automatic fusing of FIR/IIR filters. There are also tasks for strymonas itself: declaratively and concisely separating initialization (filling-in windows, estimating signal DC component and power, etc.) from steady-processing.

Acknowledgments

We thank Aggelos Biboudis and Nick Palladinos for discussions, comments and encouragement. Suggestions and comments by the anonymous reviewers are gratefully acknowledged. This work was partially supported by JSPS KAKENHI Grants Numbers 21K11821, 22H03563, and 20H04161.

References

[1] Martin Ewing AA6E. 2012. *The ABC's of Software Defined Radio.* The American Radio Relay League, Inc.

[2] Oleg Kiselyov. 2023. Generating C: Heterogeneous Metaprogramming – System Description. *Science of Computer Programming* 103015 (2023), 103015. https://doi.org/10.1016/j.scico.2023.103015

[3] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *POPL '17: Conference Record of the Annual ACM Symposium on Principles of Programming Languages* (Paris, France). ACM Press, New York, 285–299. https://doi.org/10.1145/3009837

[4] Oleg Kiselyov and Keigo Imai. 2020. Session Types without Sophistry. In *Functional and Logic Programming (Lecture Notes in Computer Science, Vol. 12073)*. Springer International Publishing, 66–87. https://doi.org/10.1007/978-3-030-59025-3_5

[5] Oleg Kiselyov, Tomoaki Kobayashi, Aggelos Biboudis, and Nick Palladinos. 2022. Highest-performance Stream Processing. https://doi.org/10.48550/arXiv.2211.13461 ACM SIGPLAN OCaml workshop.

[6] Tony R. Kuphaldt. 2023. *Lessons in Electric Circuits*. Vol. Practical Guide to Radio-Frequency Analysis and Design. All About Circuits. https://www.allaboutcircuits.com/textbook/radio-frequency-analysis-design/

[7] Tony R. Kuphaldt. 2023. *Radio Frequency Demodulation*, Chapter 5. Volume Practical Guide to Radio-Frequency Analysis and Design of modulation [6]. https://www.allaboutcircuits.com/textbook/radio-frequency-analysis-design/radio-frequency-demodulation/understanding-i-q-signals-and-quadrature-modulation/

[8] David Maier, Jin Li, Peter A. Tucker, Kristin Tufte, and Vassilis Papadimos. 2005. Semantics of Data Streams and Operators. In *Database Theory - ICDT 2005, 10th International Conference* (Edinburgh, UK) *(Lecture Notes in Computer Science, Vol. 3363)*, Thomas Eiter and Leonid Libkin (Eds.). Springer, 37–52. https://doi.org/10.1007/978-3-540-30570-5_3

[9] Julius O. III Smith. 2007. Introduction to Digital Filters with Audio Applications. https://ccrma.stanford.edu/~jos/filters/filters.html

[10] Gordon Stewart, Mahanth Gowda, Geoffrey Mainland, Bozidar Radunovic, Dimitrios Vytiniotis, and Cristina Luengo Agullo. 2015. Ziria: A DSL for Wireless Systems Programming. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. ACM, New York, NY, USA, 415–428. https://doi.org/10.1145/2694344.2694368

[11] streamit@cag.csail.mit.edu. 2006. StreamIt Language Specification, Version 2.1. (Sept. 2006). http://groups.csail.mit.edu/cag/streamit/papers/streamit-lang-spec.pdf.

[12] William Thies. 2009. *Language and Compiler Support for Stream Programs*. Ph. D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. Advisor(s) Amarasinghe, Saman. AAI0821753.

[13] William Thies and Saman P. Amarasinghe. 2010. An empirical characterization of stream programs and its implications for language and compiler design. In *Proc. 19th International Conference on Parallel Architecture and Compilation Techniques (19th PACT'10)*. ACM Press, Vienna, Austria, 365–376. https://doi.org/10.1145/1854273.1854319

Received 2023-10-20; accepted 2023-11-20