Contributions to Aetherling

David Akeley

September 12, 2018

Abstract

This is a list of my contributions to the Aetherling project. I start with a section summarizing tasks I worked on, then I expand on the tasks in further sections.

0 Summary

Most of my work was on the Haskell portion of Aetherling. Aetherling supports a Haskell intermediate representation (IR) of hardware pipelines as an abstract syntax tree (AST) of ops. These ops include arithmetic ops (e.g. multiplication) and meta-ops that take other ops as children (e.g. `MapOp`, which duplicates an op in parallel; `ComposeSeq`, which wires the outputs of one op to the inputs of another). These ASTs can then be transformed by Aetherling passes, such as the speed-up pass, which automatically paralellizes an Aetherling pipeline.

1. Line Buffer Specifications

   I proposed a new specification of Aetherling's line buffer node and wrote a document ("The Line Buffer Manifesto") describing the benefits of this redesign. The previous line buffer design was hard to parallelize due to difficult-to-satisfy constraints on its parameters' divisibility, and also did not support downsampling ("stride"). The redesign addresses these issues.

2. Functional Simulator

   I wrote a functional simulator for Aetherling, which includes a simulation of the intended behavior of the redesigned line buffer. This allows the user to test the functionality of an Aetherling pipeline (in IR form) using pure Haskell code.

3. Demonstration Apps

   I designed two demonstration Aetherling pipelines: Gaussian blur (7×7 stencil) and mipmap generation. These pipelines can be tested using the functional sim-

ulator along with a library (`ImageIO.hs`) for converting between simulator values and png images on disk. These apps also demonstrate the applications of the redesigned line buffer.

4. Simplifying Ops & Helper Functions

   An op is a node of an Aetherling AST (along with its children). Previously there were multiple ways to express the same functionality. I simplified the ops to minimize redundancy, mainly by removing unneeded type parameters for arithmetic ops. Then, I wrote some Haskell IR helper functions for creating ops and simple patterns of ops. These helpers check for invalid parameters and substitute for functionality lost through the op simplification.

5. Ready-Valid Meta-Op

   By default, Aetherling pipelines are composed of ops representing circuits with synchronous timing: they wait for a certain number of warm-up cycles, then emit outputs on a repeating schedule (phase). I designed a `ReadyValid` op that represents the idea of wrapping a portion of an Aetherling pipeline with a ready-valid interface. I modified the compose operators (`|&|` and `|>>=|`) to properly handle the `ReadyValid` op. The user is prevented from composing an op with ready-valid timing with one with synchronous timing, and, when composing two ready-valid ops in sequence, the throughput-matching behavior of the Aetherling type system is suppressed.

6. ComposePar Retiming

   Aetherling includes a `ComposePar` op that represents placing circuits (child ops) in parallel. The user is not required to ensure that each parallel path has the same latency (sequential latency – the count of the number of register delays along a path). I wrote a pass that walks an Aetherling pipeline (in Haskell IR form), searching for `ComposePar` ops, modifying its child ops if needed such that all paths have the same latency. The pass finds an optimal solution that minimizes the number of register bits added to the circuit.

7. Fractional Underutilization and Phase

   In the Aetherling team, "phase" refers to the repeating pattern of valid and garbage values input to/generated by a synchronously timed op. For example, an op that generates 1 valid output every 3 cycles would have an output phase of `[True, False, False]` (Two out of every three cycles, the op generates garbage).

   The choice of phase for an op with integer underutilization (1 valid input/output per X clock cycles) is obvious, but with fractional underutilization (X valid per Y clocks), there are several reasonable phase choices. When several such underutilized ops are joined together, there's no reason to assume that their phase patterns will match.

Since the Aetherling type system only exposes type and throughput information to the user, it's vital that the system take care of phase matching automatically. I proposed a scheme that assigns each fractional utilization ratio a standardized phase. (The earlier phase corresponds to $\frac{1}{3}$ utilization). This allows the complexity of phase matching to be confined to one op in the system, `SequenceArrayRepack`. Along with the ComposePar retiming pass, this makes it so that users only have to be concerned with type and throughput matching, allowing them to view phase and latency as performance, rather than correctness, issues.

8. Tests Written

   I gained a lot of experience writing tests as part of my work on the Aetherling project. These tests include tests for the functional simulator, tests for the hardware line buffer David Durst is designing, tests for the compose operators (`|&|` and `|>>=|`), and tests for the `ComposePar` retiming passes.

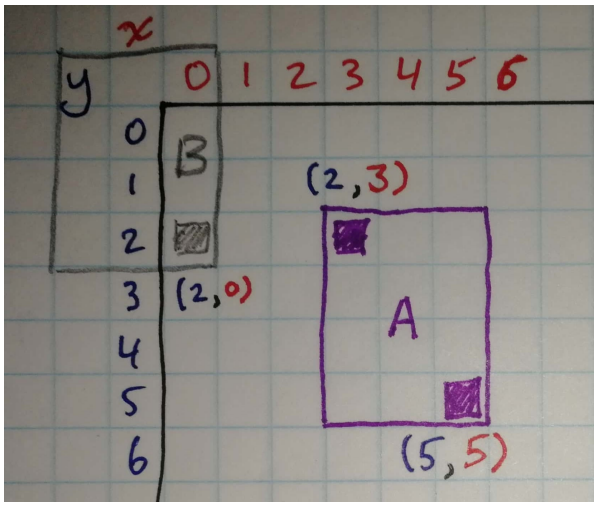
1 Line Buffer Specifications

One of the main goals of Aetherling is to facilitate automatic parallelization of hardware pipelines. To this end, Aetherling implements speed-up passes that determine the correct way to parallelize each type of op in an Aetherling pipeline. For example, an addition op can be parallelized by being wrapped in a map; an existing map can be parallelized by increasing the map's parallelism (lane count).

This parallelization turned out to be difficult for the original line buffer op Aetherling supported. The line buffer reads in an image as a stream of pixels (streamed in left-to-right, row-by-row), and emits as output a stream of rectangular portions of the image (we call these rectangles "windows"). Ops downstream from the line buffer can then base their results on pixels streamed in at different points in time.

The original line buffer was scheduled as follows: each time a pixel is streamed in, an output window with the streamed-in pixel at its bottom-right (highest x and y) is emitted, unless that window would include pixels outside of the image bounds. For example, if the window is 4 pixels tall and 3 pixels wide (4×3; I use (y, x) coordinates with $(0, 0)$ being the top-left pixel[1]), pixel $(5, 5)$ would be input on the same cycle that the window with corners $(2, 3)$ and $(5, 5)$ would appear as valid output, while input pixel $(2, 0)$ would have no corresponding valid output window since the corresponding corner $(-1, -2)$ is out-of-bounds. (To make the distinction between valid and invalid outputs concrete, we say that if the line buffer were part of a ready-valid circuit, valid would be asserted only on cycles with valid output).

[1] I advocated for Aetherling to use (y, x) coordinates so that pixels would be streamed in lexicographical order.



Window A (purple) is considered a valid output, but window B (gray) is not a valid output by the old line buffer's rules.

The original line buffer could be parallelized by increasing the number of pixels streamed in per clock cycle. Since (almost) each input pixel corresponds to one output window's lower-right corner, multiple output windows would be produced per cycle as well.

This plan seemed simple, but in practice such parallelized line buffers were difficult to schedule. The main issue was that, with multiple windows output per cycle, it's possible for some to be valid and some to be invalid at the same time. This would require some complicated lane-interleaving logic downstream from the line buffer. Furthermore, even without paralellization, this line buffer would be difficult to place in a synchronously-timed circuit (i.e. non-ready-valid), since it produces an irregular output pattern. The output is stalled each time the input "passes over" the left margin of the image being streamed; for example, a line buffer with a 4-pixel wide window would produce no valid output whenever one of the first 3 pixels of a row is input.

To make scheduling managable, the original line buffer had several divisibility requirements on its parameters. The most problematic restriction was that the input pixels-per-clock[2] had to divide both

1. Window width - 1
2. Image width - Window width + 1

These restrictions were needed to prevent both valid and invalid windows from being produced on the same cycle (Figure 1). In practice, few levels-of-parallelization (besides 1) could satisfy both restrictions. Stencil sizes are typically small, so "Window width - 1" already sets a low ceiling on pixels-per-clock. "Window width - 1" more likely than

[2]Strictly speaking, this parameter is horizontal pixels-per-clock. The old line buffer was also planned to support streaming multiple rows of pixels in per cycle, which I don't discuss here.

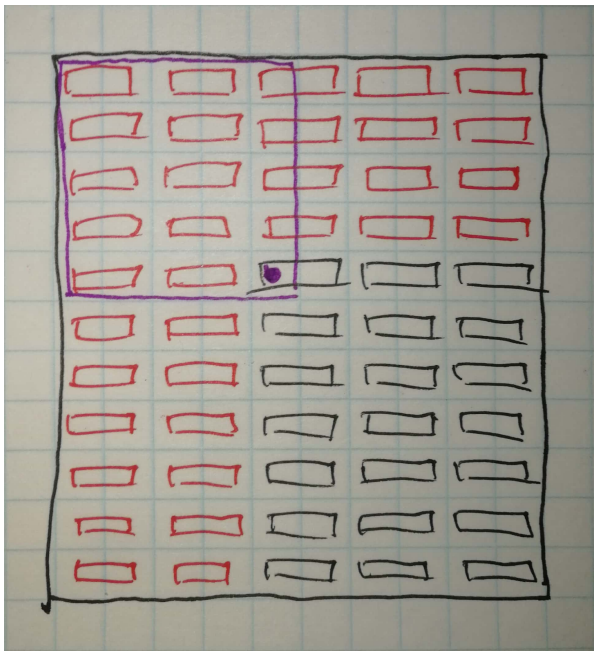


Figure 1: Visualization of how input pixels tile an image. This line buffer takes 2 pixels per clock, has a 5×5 window size, and reads a 11×10 image. Red inputs produce no valid output windows; black inputs produce only valid output windows (under old line buffer rules).

not would be a prime number, and, if "Image width - Window width + 1" did not share that prime factor, the line buffer cannot be parallelized at all.

A further drawback of the original line buffer is that it did not support an output "stride". Each output window is offset by 1 pixel from its nearest vertical and horizontal neighbors. This prevents mipmapping and pyramid-based image processing algorithms from being expressed as Aetherling pipelines. (I think that Darkroom had a similar issue).

The main problem with the old line buffer was that invalid "garbage" outputs could have two causes. It could be caused by out-of-bounds pixel access, or it could be caused by scheduling (`ReduceOp` is an example of an op that produces garbage due to scheduling. A reduce that sums up 4-sequences of integers would produce garbage outputs 3-out-of-4 cycles waiting for the full sequence to be made available).

For the new line buffer that David Durst is implementing in hardware, I proposed that invalid values should only result from scheduling, not from out-of-bounds pixel access. If a window is scheduled to be output, it should still be considered valid even if it contains garbage out-of-bounds pixels. Global bounds analysis can be performed at the end of an Aetherling pipeline to crop out the margins of the final output image that were produced from garbage pixel values. By moving the responsibility of tracking

out-of-bounds pixel access from the individual line buffers to a single node at the end of the circuit, we can get rid of the two most restrictive divisibility requirements listed above.

I also proposed that the new line buffer should support output stride (horizontal and vertical distance between windows) and origin (the top-left corner of the initial output window). Vertical stride in particular produces some scheduling difficulties that I proposed (but did not implement in hardware) a solution to.

1.1 New Line Buffer Parameters

The new line buffer takes these parameters as pairs of $(height, width)$.

1. Input pixels per clock ($pxPerClk$)

   This defines the "shape" of the 2D-array input bus, and therefore how much more of the input image we see per cycle.[3]

   I originally proposed that the height of this parameter must be 1, but the new line buffer may be extended to support streaming in multiple rows per clock cycle. In this case, the height of this parameter will be greater than 1, and the parameter's width must match the image width.

2. Output window size ($window$)

3. Image size ($image$)

4. Output stride ($stride$)

   The vertical and horizontal distance between the upper-left corners of adjacent output windows.

5. Output origin ($origin$)

   The upper-left coordinate of the initial output window (itself the upper-left most window).

The line buffer also takes a type parameter (e.g. it could be parameterized with `tInts[3]` to process RGB pixels), and there's talk of supporting a boundary condition parameter: I proposed that the line buffer simply emit some garbage when it's scheduled to output a window that overhangs the image boundaries, but it may be useful to provide the line buffer with well-defined behavior when it conceptually accesses an out-of-bounds pixel (something akin to OpenGL's `GL_TEXTURE_WRAP` parameter).

[3]For simplicity I talk as if the line buffer op is fully utilized. If the line buffer is underutilized, then "each cycle" should be taken to mean "each cycle when the line buffer is scheduled to advance its state".

1.2 Input-Output Format

The input is a 2D-array of pixels, with height and width given by the pixels-per-clock parameter. Pixels are stored and streamed in reading order: left-to-right, then top-to-bottom. (If the width of the input array equals the width of the image, then the image is streamed-in one (or more) row at a time, top-to-bottom).

Example: For a 9-pixel wide image and a $(1, 3)$ pixels-per-clock parameter, the first 4 cycles will see these pixels as input:

cycle	input[0][0]	input[0][1]	input[0][2]
0	(0,0)	(0,1)	(0,2)
1	(0,3)	(0,4)	(0,5)
2	(0,6)	(0,7)	(0,8)
3	(1,0)	(1,1)	(1,2)

The output is a 3D-array of pixels: the inner 2 dimensions correspond to the height and width of a window, and the outer dimension is for parallelism (multiple output windows packed together, left-to-right).

For each image streamed in, the line buffer should produce

$$\frac{image_y}{stride_y} \tag{1}$$

rows of valid output windows, each of which consists of

$$\frac{image_x}{stride_x} \tag{2}$$

valid output windows.

Conceptually, stride is used for downsampling, so we expect that the number of outputs should be divided by the stride. For example, if we're using a line buffer to downsample a 2048×1024 image by 2 with Gaussian smoothing step, we would use a line buffer with a $(2, 2)$ stride, and expect $\frac{2048}{2}$ rows of output windows, $\frac{1024}{2}$ windows per row, yielding a total of 1024×512 windows, each of which gets convolved to yield 1 pixel of the 1024×512 output image.

1.3 Throughput and Schedule

We define the "window throughput" as the average number of valid output windows per cycle. I wanted to spread out the new line buffer's outputs evenly through time to give the new line buffer a regular output schedule. So, I calculate the throughput by dividing the total number of output windows (per image) with the number of cycles it takes the line buffer to stream in an entire image.

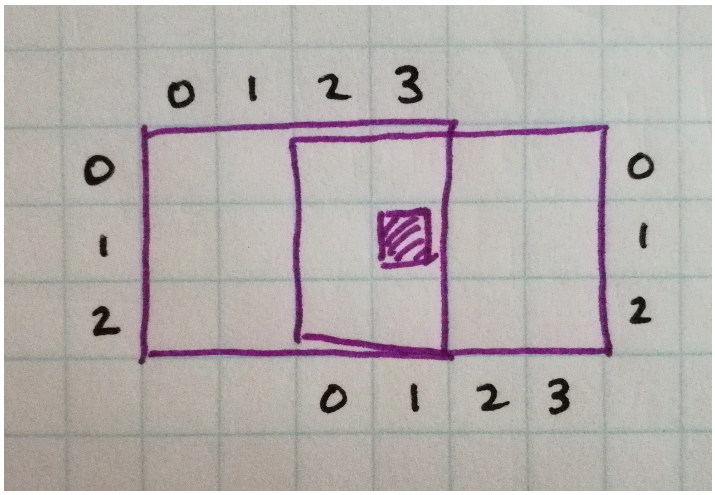


Figure 2: Figure representing one cycle's output of a line buffer with a $(3, 4)$ output window size, an x-stride of 2, and 2 output windows per clock. The output array dimensions are `[2,3,4]`, since there's a count of 2 output windows per cycle, and each window has a size of $(3, 4)$. The highlighted pixel has coordinates `[0,1,3]` and `[1,1,1]`, because it's at coordinate $(1, 3)$ of the upper-left-most window (`O[0]`) and at coordinate $(1, 1)$ of the `O[1]` window (the window 1 stride to the right of the `O[0]` window).

It takes

$$\frac{image_x image_y}{pxPerClk_x pxPerClk_y} \tag{3}$$

cycles to stream in the entire image, and in that amount of time[4] we must emit

$$\frac{image_x image_y}{stride_x stride_y} \tag{4}$$

valid output windows. Dividing the two quantities, we have

$$\text{Window throughput} = \frac{pxPerClk_x pxPerClk_y}{strides_x strides_y} \tag{5}$$

The parallelism of the output array (i.e. its outermost dimension) will be set to accomodate this many output windows per cycle (1 for window throughput ≤ 1).

If the window throughput is 1 or greater, then it should be an integer N (more on this later), and the line buffer is scheduled to start with a certain number of latency cycles, where no valid outputs are produced, then proceed to generate N valid outputs every cycle.

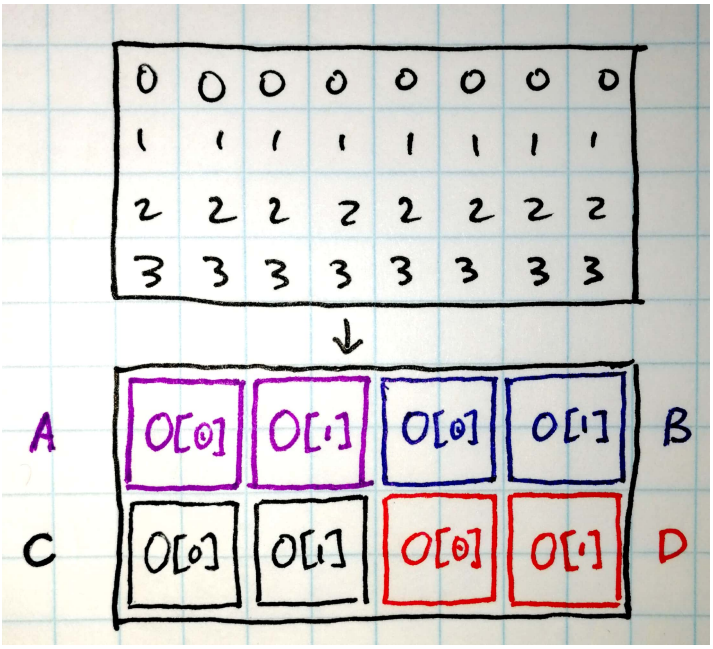
If the window throughput is a fraction less than 1, then the line buffer will generate at most 1 valid output per clock cycle.

[4] "in that *amount* of time", because there may be added latency.

These output windows are packed into the parallel output left-to-right, then streamed left-to-right, row-by-row.

Especially for line buffers with $stride_y > 1$, evenly spreading out the output windows through time breaks the relationship between a given cycle's inputs and outputs. Unlike the old line buffer, where each output window's lower-right pixel corresponds to a pixel input on the same cycle, in the new line buffer there may be no clear relationship between one clock cycle's outputs and inputs.

As an example, consider a line buffer with image size $(4, 8)$, stride $(2, 2)$, and $pxPerClk$ $(1, 8)$.



Top: Image pixels labelled with the clock cycle they were input on.

Bottom: Output window schedule. Each color represents one cycle's output.

This line buffer reads in an entire row of the image per clock cycle, and, dividing (1×8) by (2×2), we deduce that it should emit 2 valid output windows per clock cycle. After waiting a certain number of latency cycles (depends on implementation), the line buffer should output the 2-window arrays A, B, C, and D over 4 clock cycles with no pauses in between (unless the line buffer has been wrapped in an underutilization op).

1.4 Remaining Divisibility Requirements

The remaining divisibility requirements are that

1. The stride width (height) must divide the image width (height).

2. The $pxPerClk$ width must divide the image width.
3. The window throughput must be an integer or a reciprocal of an integer.
4. If $pxPerClk$ height is not 1, then
   (a) $pxPerClk_x = image_x$.
   (b) $pxPerClk_y$ must divide $image_y$.
   (c) $pxPerClk_y$ must divide $stride_y$, or the other way around.

The first two divisibility restrictions depend on the image dimensions, so I expect them not to be too restrictive since typical images have fairly composite dimensions. Furthermore, if multiple line buffers are chained together with no downsampling ($stride = (1, 1)$), then all of them will have the same image size and will not impose additional constraints on one another.

The $pxPerClk$ restrictions are justified by the difficulty of implementing a line buffer that accepts portions of 2 different rows in one clock cycle as input, or emits windows from two different rows as output in one clock cycle.

The *stride* restrictions have both conceptual and pragmatic justifications. The conceptual justification is that stride is used for downsampling, and we expect downsampling by a factor of x should scale the number of output samples (windows) by exactly x^{-1} – impossible if the image dimension is not itself a multiple of x.

The pragmatic justification is that scheduling becomes difficult without this stride assumption. The window throughput calculations were dependent on the total number of output windows being exactly

$$\frac{image_x image_y}{stride_x stride_y} \tag{6}$$

This cannot happen if the strides do not divide the image dimensions.[5]

The window throughput restriction is a bit more punishing, although not as punishing as the old line buffer's divisibility restrictions. This restriction was also motivated by scheduling concerns. It's not clear how to schedule a line buffer that produces a

[5]In case this sounds more-or-less the same as the conceptual reason, the low-level justification is that, to keep the internal buffer size bounded, we need the number of input pixels per clock to match, on average, with the number of "new" output pixels per clock ("new" in the sense that this pixel was not output as part of an earlier output window). It turns out that the average number of new pixels per window is equal to $stride_x stride_y$, leading to the constraint

$pxPerClk_x pxPerClk_y = (\text{Window Throughput}) stride_x stride_y$

(rearrange to get the original Window Throughput calculation). The issue is that this averaging doesn't work if the strides do not divide their respective image dimensions; the left margin's windows will have a greater-than-average number of new pixels, which will only be balanced out by the right margin's windows having a lower-than-average number of new pixels if the strides divide image dimensions. See figure 3 for visualization.

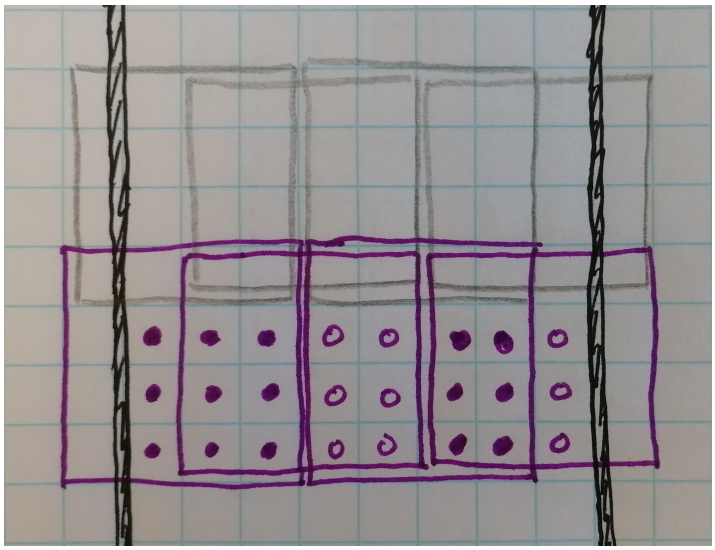


Figure 3: Footnote example with image width 8, output window size $(4, 4)$ and strides $(3, 2)$. There are 4 windows per row of windows since image width 8 divided by $stride_x$ 2 is 4. Notice that the left-most window includes 9 new pixels and the right-most 3, but on average each window requires emitting 6 new pixels, which matches the product of $stride_x$ and $stride_y$.

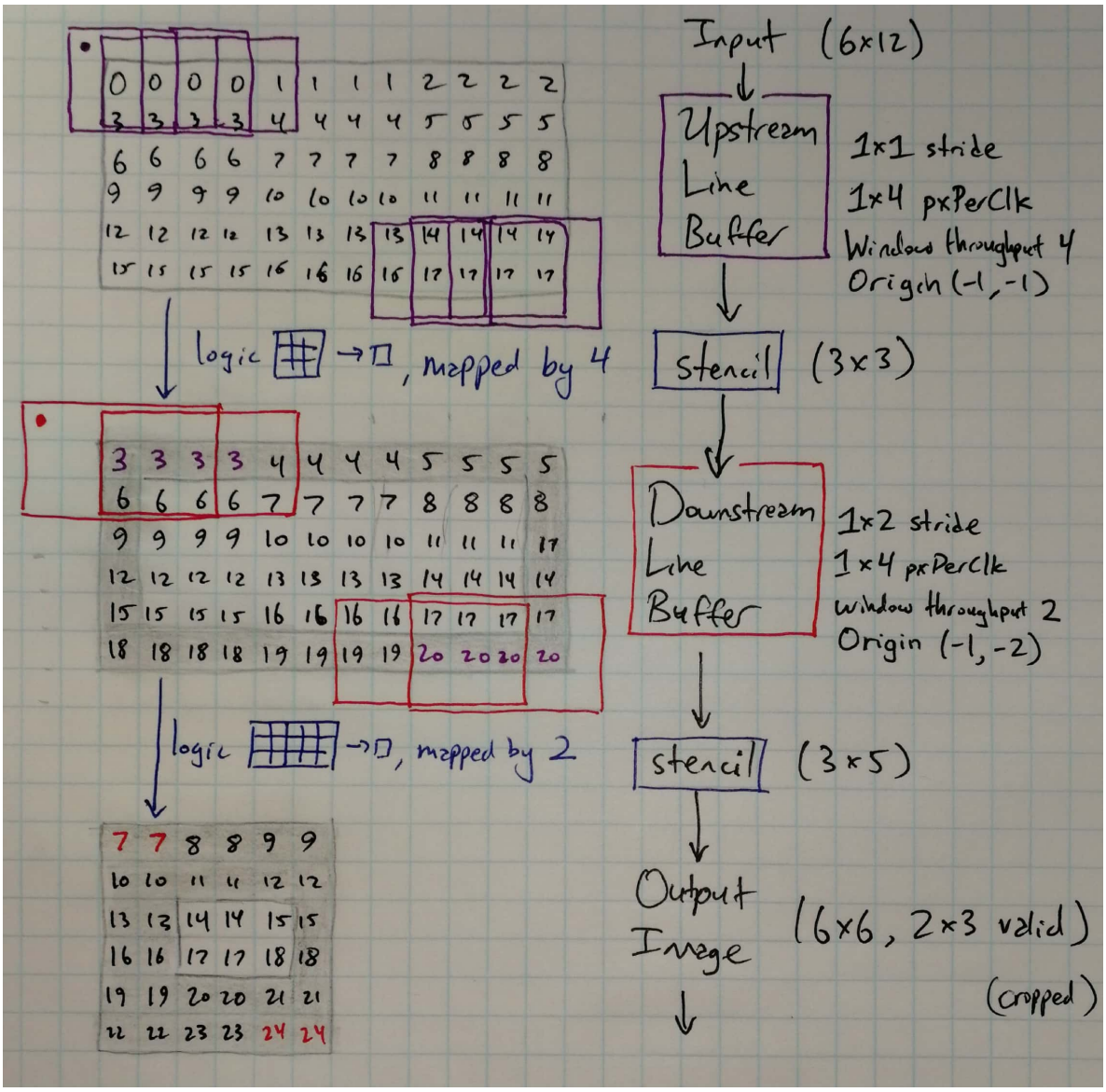
fractional number of valid output windows per cycle (with window throughput > 1), and, at the time I made my line buffer proposal, if window throughput were less than 1, it also wasn't clear how to schedule line buffers with window throughput not of the form $\frac{1}{N}$, N :integer. My phase proposal might make it feasible to lift the window throughput restrictions when window throughput is below 1.

1.5 Chain of Line Buffers Example

Consider a pipeline made up of two line buffers with combinational stencil logic inserted between (3×3 stencil) and after (3×5) the line buffers. The two line buffers have parameters[6]

	pxPerClk	*window*	*image*	*stride*	*origin*	`sequentialLatency`
Upstream line buffer	(1,4)	(3,3)	(6,12)	(1,1)	(-1,-1)	3
Downstream line buffer	(1,4)	(3,5)	(6,12)	(1,2)	(-1,-2)	4

[6]Errata: It turns out that the upstream line buffer's latency is impossibly low – the bottom-right pixel of the rightmost window each cycle has to travel back in time by 1 cycle.



For the upstream line buffer, we have that

$$\text{Window throughput} = \frac{pxPerClk_x pxPerClk_y}{stride_x stride_y} = 4 \tag{7}$$

So, there are 4 output windows per cycle (the initial and final batch of windows are highlighted in purple), and, since both strides are 1, the input and output images have the same size – height 6, width 12. The 0^{th} output window of the initial valid output cycle consists of the 9 pixels from the origin (-1,-1) to (1,1); this window is still considered a valid output even though 5 of its pixels are garbage values.

The stencil logic connecting the two line buffers must be mapped by 4, since the upstream buffer produces 4 output windows per clock cycle. Each pixel is labelled with

the clock cycle that it is input to the line buffer, and pixels whose values are based on windows with garbage pixels are shaded in gray.

The downstream line buffer has the same $pxPerClk$ but a different stride – (1,2). This halves the width of its output image (now 6×6), and halves its window throughput to 2. The stencil logic downstream of this line buffer, which produces the final output image, is only mapped by 2 then.

Based on my propose schedule for such a pipeline, the output image has size 6×6 (18 valid outputs of 2 pixels each). However, at the end of the pipeline, we may change our definition of "valid" to exclude garbage pixel values. Using bounds analysis we can conclude that the valid portion of the output image has size 2×3.

2 Functional Simulator

Note: `src/Core/Aetherling/Simulator/README.md` in the `aetherlingHaskellIR` source tree is the main source of documentation for the functional simulator.

The functional simulator (`simulateHighLevel` function) was the first contribution I made to the Aetherling project. At the time, the plan for timing Aetherling ops wasn't clear, and many of the ops were not realized in hardware yet, so it seemed important to write a high-level simulator that did not concern itself with precise hardware details like timing.

The data type that the simulator operates on is a "stream" of `ValueType`: this represents the values of a port's valid outputs through time, with no timing details included. Input is passed to a simulated op with multiple ports as a list of streams, represented in Haskell as a 2D-list of `ValueType`, with the outer dimension corresponding to ports and the inner dimension corresponding to time.

As a simple example, consider simulating an adder:

```
inputs = [
    [V_Int 0, V_Int 2, V_Int 4],
    [V_Int 30, V_Int 20, V_Int 10]]
simulateHighLevel (addInts T_Int) inputs [] -- empty memory argument

outputs: ([V_Int 30, V_Int 22, V_Int 14], []) -- empty memory result
```

(`V_Int` constructs a `ValueType` representing a simulated integer value).

Assuming the adder is timed to perform one add per cycle, this simulation corresponds to an adder calculating

cycle	port 0 input	port 1 input	result
0	0	30	$0 + 30 = 30$
1	2	20	$2 + 20 = 22$
2	4	10	$4 + 10 = 14$

over 3 clock cycles.

The reason I chose the outer dimension of the input list to correspond to port and the inner dimension correspond to time is that it's not possible to represent inputs the other way without knowing details of the simulated pipeline's timing.

Consider a pipeline with two input port. This pipeline reads a 4-sequence of integers from port 0 and a single integer from port 1; sums up the 4-sequence, multiplies the single integer by 16, and outputs the maximum of those two values. We can write this pipeline as

```
add4stream = arrayReshape [T_Int] [tInts [1]] |>>=| ReduceOp 1 4 (addInts T_Int)
pipeline = (add4stream |&| underutil 4 (Shl 4)) |>>=| underutil 4 (maxInts T_Int)
```

Simulate the pipeline with 2 input sequences using

```
inputs =
    [ -- port 0 input
      [V_Int 1, V_Int 2, V_Int 3, V_Int 4, V_Int 5, V_Int 6, V_Int 7, V_Int 8],
      -- port 1 input
      [V_Int 0, V_Int 10] ]
simulateHighLevel pipeline inputs []
```

The port output should be `[10, 160]`: each input sequence results in a single output. The 0^{th} 4-sequence on port 0 is `[1, 2, 3, 4]`, and the 0^{th} 1-sequence on port 1 is `[0]`, so we expect output `[0]` to be

$$\max(1 + 2 + 3 + 4, 0 \text{ shl } 4) = \max(10, 0) = 10 \tag{8}$$

Output number `[1]` should be based on the 1^{st} input 4- and 1- sequences on ports 0 and 1, so we expect to see

$$\max(5 + 6 + 7 + 8, 10 \text{ shl } 4) = \max(26, 160) = 160 \tag{9}$$

The important point is that entry `[1]` of the port 1 input (`...V_Int 10]`) is automatically matched with 4-sequence input number `[1]` on port 0, `...V_Int 5, V_Int 6, V_Int 7, V_Int 8]`. This could not have been done if the input 2D-list's dimensions were swapped without knowing the precise timing of the circuit. In this case, the timing will be for port 1's input to come in as the last (3^{rd}) input of port 0's 4-sequence comes in, so, if the simulator were written with the input dimensions swapped (time outer, port inner dimension), the input may look something like

```
-- port 0 (4-sequence)
--   V
--   V    port 1 (1-sequence), X denotes invalid input
--   V      V
[ -- V      V
  [V_Int 1, X],       -- cycle 0 input
  [V_Int 2, X],       -- cycle 1 input
  [V_Int 3, X],       -- cycle 2 input
  [V_Int 4, V_Int 0], -- cycle 3 input
  [V_Int 5, X],       -- cycle 4 input
  [V_Int 6, X],       -- cycle 5 input
  [V_Int 7, X],       -- cycle 6 input
  [V_Int 8, V_Int 10] -- cycle 7 input
]
```

In general though, at the time the simulator was written, it would not have been possible to deduce exact timing in all cases.

2.1 Implementation Notes

Aetherling (at the time of writing) defines two memory ops: `MemRead` and `MemWrite`. Unlike port inputs and outputs, which only appear at the beginning and end of a pipeline, memory inputs and outputs could appear at any stage of an Aetherling pipeline. Distributing `MemRead` values and collecting `MemWrite` values properly was difficult to implement in the simulator.

The first problem is that the individual `MemRead` and `MemWrite` ops have no names, making it hard to even specify the inputs and outputs. The solution I came up with for the time being is to number each `MemRead` and `MemWrite` based on the order they would be visited by a depth-first search of an Aetherling AST.[7]

Internally, the simulator works by specializing the `simhl`[8] function for each op offered by Aetherling.

[7] Strictly speaking, the memory ops are numbered based on the order they would be visited by a DFS of an Aetherling AST transformed by duplicating ops by the number of times they actually appear in a physical circuit. Aetherling defines ops like `MapOp n` that have one child node that represents `n` copies of the actual op. If that op contains memory ops, each needs to be numbered separately.

[8] *sim*ulate *h*igh *l*evel

3 Demonstration Apps

4 Simplifying Ops & Helper Functions

Previously, many Aetherling arithmetic ops (e.g. `Add`) took a type parameter, which could be an array type. This meant that there were multiple ways to express the same operation. For example, a bit xor could be expressed as `XOr T_Bit` or `Add T_Bit`, and elementwise addition of two 4-arrays-of-int could be expressed as `Add $ T_Array 4 T_Int` (4-array type parameter) or as a `MapOp 4` over a scalar `Add T_Int`.

Since each op of the Aetherling Haskell IR will eventually need to be implemented in hardware, it would be best to minimize unneeded functionality as much as possible. I eliminated the type parameter from arithmetic ops, splitting the op into two ops if bit and integer versions are both needed (e.g. `And` for boolean *and*; `AndInt` for bitwise *and*).

Since this change makes vectorized arithmetic harder to express directly, I wrote some helper functions for expressing array types and array operations. For example, a 16×16 matrix of integers can be expressed as `tInts[16,16]`, and an op performing elementwise addition of two matrices can be expressed as `addInts $ tInts[16,16]`. Internally, the IR represents vectorized ops as scalar ops wrapped by `MapOp`. In this example, `addInts` will return

```
MapOp 16 (MapOp 16 Add)
```

The helper functions also serve the purpose of checking for invalid parameters. For example, the function for creating a line buffer op[9] checks that the divisibility requirements are satisfied, and the `arrayReshape` function, which creates an `ArrayReshape` op that reinterprets inputs as a different type, checks that there's a reasonable mapping between input and output types.[10]

[9] Temporarily named `manifestoLineBuffer`.

[10] Examples: `arrayReshape [tBits[2]] [T_Bit, T_Bit]` (conversion of 2-array-of-bit to two separate bits) is valid, while `arrayReshape [tBits[3]] [T_Bit]` is not since the input has more bits than the output.

5 ReadyValid Op

6 ComposePar Retiming

7 Fractional Underutilization and Phase

Originally Aetherling only supported the concept of integer underutilization. After waiting for a certain number of warmup cycles, synchronous ops could accept inputs or create outputs on a repeating schedule of 1 valid input/output every `N` clock cycles, where `N` is an integer.[11]

The problem is that we wanted to lift the integer restriction on underutilization and allow Aetherling ops to have any fractional throughput (between 0 and 1) ("fractional underutilization"). The benefit of the original integer underutilization scheme is that two ops are guaranteed to work together when composed in sequence given that their linked ports have the same throughput ($\frac{1}{N}$). As long as the downstream op waits for L cycles before accepting the first input, where L is the latency in clock cycles of the upstream op, the timings of the two ops will match, with the upstream op feeding input to the downstream op on cycles L, $L+N$, $L+2N$, $L+3N$...

To maintain this simplicity with fractional underutilization, I proposed that each fractional throughput should be assigned one standardized phase pattern (repeating schedule of valid and garbage inputs/outputs). The details of this assignment algorithm are not important yet. The important points of this plan are that

1. Each op has one `sequentialLatency` value – the difference in clock cycles between the time when its first input arrives and its first output is produced.[12]
2. Each port of an op is assigned a phase pattern based on its throughput. (In Aetherling, this throughput is calculated by dividing `seqLen`, a property of the port, with `cps` (clocks per sequence), a property of the entire op).
3. Each output port of an op, in isolation, behaves like this: it emits garbage for `sequentialLatency` cycles, then cycles through its assigned phase pattern of valid and garbage outputs. (If the op is not in isolation, we'll have to wait additional cycles equal to the wait time for the first input to arrive – this is equivalent to the sum of the upstream ops' `sequentialLatency`s). Note that a consequence of

[11] Strictly speaking, Aetherling supported reciprocal-integer throughputs, which I refer to as integer underutilization since most ops can only support such throughputs through underutilization.

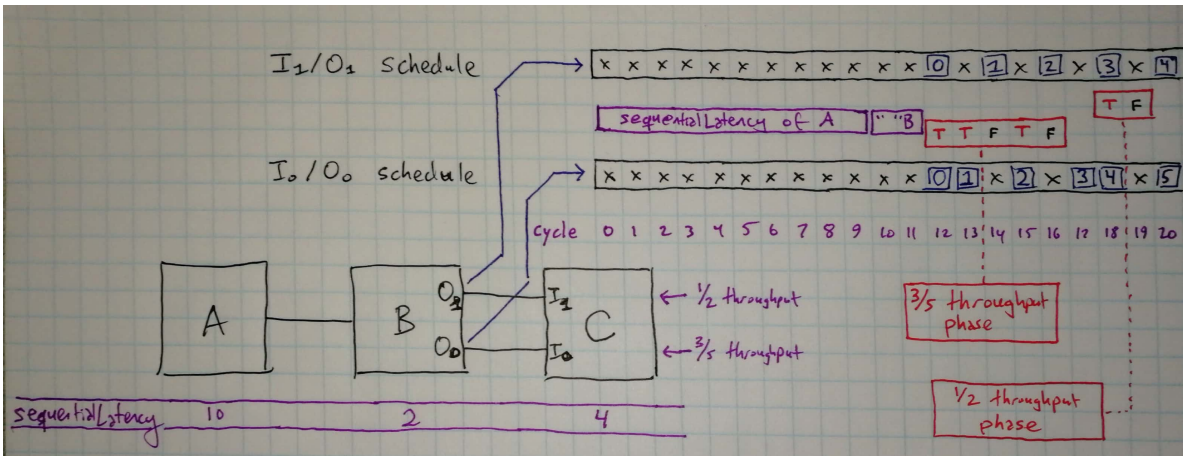
[12] This is still well-defined if the op has multiple ports with varying throughputs. Each phase pattern starts with a valid input/output, so the initial batch of inputs are synchronized across all ports. Same with the initial batch of outputs.

the earlier definition of `sequentialLatency` as the difference in time between the first input and first output, the phase pattern must start with a valid output.

With these standardized phases, given that we properly delay downstream ops, we can be certain that two ports' schedules will match if their fractional throughputs match (checked by the Aetherling type system), just as the case was for the earlier system that only supported integer-reciprocal throughputs.

7.1 Pipeline Example

Consider the example pipeline `A |>>=| B |>>=| C` (`|>>=|` means sequential compose, i.e. wire the outputs of left op to inputs of right op). Suppose that `A` has `sequentialLatency` 10, `B` has `sequentialLatency` 2, `C` has `sequentialLatency` 4, `B` has two output ports O_0 and O_1 with throughputs $\frac{3}{5}$ and $\frac{1}{2}$ respectively, which matches input ports I_0, I_1 of `C`. The standard phase for $\frac{3}{5}$ throughput is `[True, True, False, True, False]`, and for $\frac{1}{2}$, `[True, False]`. (`True` denotes a clock cycle with valid input/output; `False` denotes garbage).



Visualization of scheduling based on `sequentialLatency` wait and repeating phase pattern.

Consider the output pattern of O_0. We'll start with 12 cycles of garbage, since 12 is the sum of `A` and `B`'s `sequentialLatency`s, then O_0 will create valid outputs on cycles 12, 13, 15, 17, 18, 20, 22... (since the $\frac{3}{5}$ phase indicates the $0^{th}, 1^{st}$, and 3^{rd} cycles in each 5-cycle pattern is valid). Meanwhile, valid output will be sent through O_1 on cycles 12, 14, 16, 18, 20, 22...

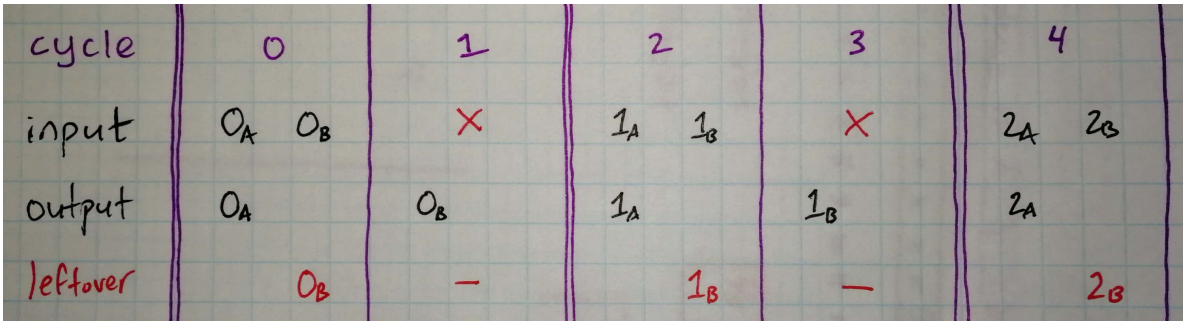
This matches what `C`'s input ports I_0 and I_1 expect. They'll wait 12 cycles since 12 is the sum of the upstream ops' `sequentialLatency`s, then, by looking up the same phase pattern, `C` can reproduce the exact schedule that `B` used. (I_0 expects input on cycles 12, 13, 15, 17...; I_1 on cycles 12, 14, 16...).

Finally, note that the initial inputs and initial outputs are always synchronized across ports. O_0 and O_1 start output on cycle 12, and I_0 and I_1 start expecting input on cycle 12. This means that the definition of `sequentialLatency` is well-defined. Using the definition, we expect that each input port of `B` should start accepting input on cycle 10, and each output port of `C` should emit its first valid output on cycle 16.

7.2 Phase Assignment Algorithm

The throughput-to-phase assignment algorithm is based on the `SequenceArrayRepack` op. This op converts between sequences of arrays of different sizes. (In Aetherling terms, a sequence is a stream of valid values delivered across multiple, not-necessarily-contiguous clock cycles – in the earlier example, port O_0 delivers a 3-sequence of outputs on cycles 12, 13, and 15).

For example, a (fully utilized) `SequenceArrayRepack` that converts from 1-sequences of 2-arrays to 2-sequences of 1-arrays would take a valid 2-array input on cycles 0, 2, 4... and, using values unpacked from the input arrays, generate a 1-array output on each clock cycle. On each even-numbered cycle, one entry of the input 2-array won't fit in the output 1-array, and will be "leftover" (i.e. buffered) for output on the next clock cycle.



`SequenceArrayRepack (1,2) (2,1)` repacking 2-arrays to 1-arrays.

Each sequence gets its own digit.

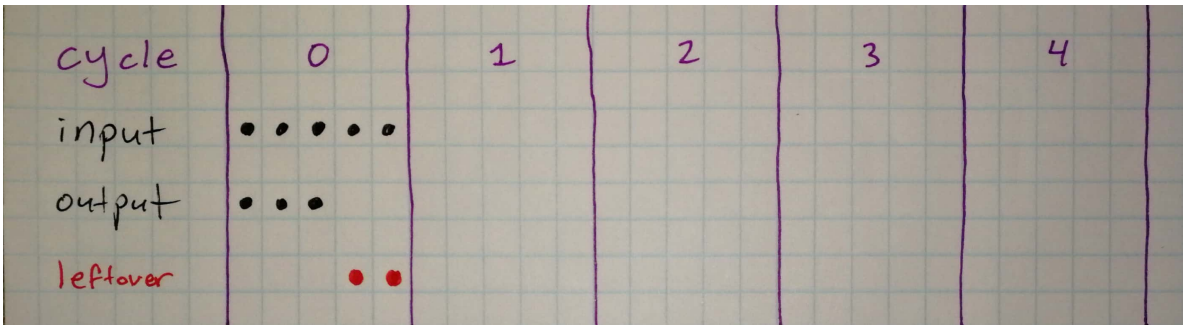
To determine the pattern for an $\frac{x}{Y}$ throughput, we figure out what pattern would be most convenient for the input of a fully-utilized `SequenceArrayRepack` converting x-sequences of Y-arrays to Y-sequences of x-arrays. This is a "narrowing" repack: the output arrays are narrower than the input arrays since

$$\text{throughput} < 1 \implies x < Y$$

The repack is fully utilized, so the output comes out at maximum rate: Y outputs over Y clock cycles (in Aetherling terms, output `seqLen=cps=Y`).

It would be most convenient for the `SequenceArrayRepack` if each input array came in as soon as it was needed, but no sooner. (This is convenient in the sense that it

minimizes internal buffering). Earlier, I set the $\frac{3}{5}$ throughput phase to `[True, True, False, True, False]`. This pattern can be reproduced by working through what a fully-utilized 3-sequence, 5-array to 5-sequence, 3-array `SequenceArrayRepack` input schedule should be. Make a table with 5 columns, one for each clock cycle 0-4 (fully-utilized $\implies$ `cps=5`), and have entries for inputs, outputs, and leftovers in each column.



Visual aid. Fill in the chart for the next example.

`SequenceArrayRepack (3,5) (5,3)`

On each clock cycle, one 3-array output must be produced. Represent this by drawing 3 dots in the output cells. This 0^{th} cycle's output must come from some input, so one 5-array input must come in on the 0^{th} cycle, leaving 2 array entries left over that must be stored in `SequenceArrayRepack`'s internal buffer. Draw 5 dots in the 0^{th} input cell and 2 dots in the 0^{th} leftover cell to represent this.

2 array entries alone are not enough to populate the 1^{st} cycle's 3-array output, so on that cycle we must read in another 5-array input. By cycle 1 then, the repack will have read in 10 and output 6 array entries, leaving 4 stored internally. Draw 5, 3, and 4 dots in the first column to represent this (input, output, leftover).

Now that there's 4 array entries buffered, we can produce another 3-array output on the 2^{nd} cycle without reading in another array. By the earlier rule, we defer reading in another valid input for now, producing the 3-array output using buffered inputs. Fill in 3 output dots and 1 leftover dots in column 2 to represent this.

Fill in the last 2 columns using the same rule. There should be input 5-arrays on cycles 0, 1, and 3, leading to the phase pattern `[True, True, False, True, False]`. For each cycle 0-4, there should be 2, 4, 1, 3, and finally 0 leftover entries.[13]

[13]That there are 0 leftover entries at the end is notable: we would just repeat if we continued the phase pattern for another 5 cycles. If the throughput fraction $\frac{x}{Y}$ were not in reduced form, we would still get the same phase (recall that phase repeats, so `[True, False]`-repeated $\equiv$ `[True, False, True, False]`-repeated. Although I've explained this as a fraction-to-phase mapping, in practice it's a mapping of `seqLen`$\equiv x$ and `cps`$\equiv Y$ to phase, so it's not actually guaranteed *a priori* that two ops with matching fractional throughputs will have matching phase if the $\frac{\text{seqLen}}{\text{cps}}$ fractions aren't in reduced form. That the phase is still the same without reduced fractions is pretty important to this scheme, but no one else seemed to be worried about this so it's in this footnote here.

7.3 SequenceArrayRepack Example

I propose that all ops conform to this phase scheme, including `SequenceArrayRepack` itself. It's a bit hard to explain my proposed `SequenceArrayRepack`'s behavior because of pontential confusion between the concrete `SequenceArrayRepack` being analyzed and the imaginary `SequenceArrayRepack`s used to determine phase patterns.

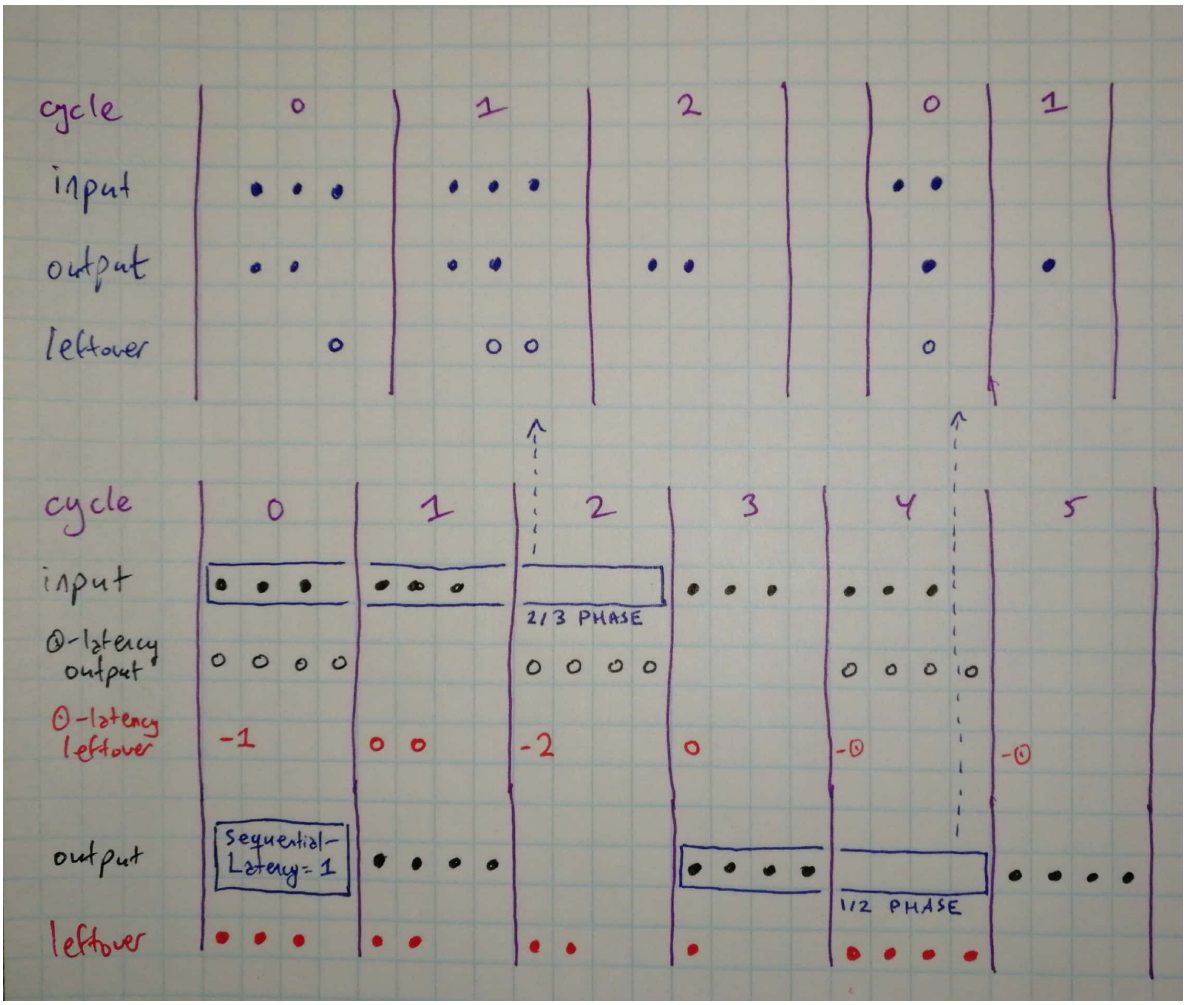
Let's work through the behavior of a repack that accepts as input 4-sequences of 3-arrays, produces 3-sequences of 4-arrays, and has `cps=6`. This repack has an input throughput of $\frac{4}{6} = \frac{2}{3}$ and an output throughput of $\frac{3}{6} = \frac{1}{2}$.[14]

Determining the input and output phases requires us to construct a couple of `SequenceArrayRepack`s that don't really resemble the repack we're analyzing – analyzing their input schedule as we did above, we'll get an input phase of `[True, True, False]` and an output phase of `[True, False]`. Focusing back on the original `SequenceArrayRepack` we're analyzing, if we just repeat this phase pattern without any added latency, some array entries will be scheduled for output before they were scheduled for input, so we need to have `sequentialLatency=1`.

Actually, there's one last thing to note, which is that an X-array to Y-array `SequenceArrayRepack` has the same schedule (all else being equal) as an nX-array to nY-array repack, n :integer. This is the other half of the phase being the same without reducing fractions.

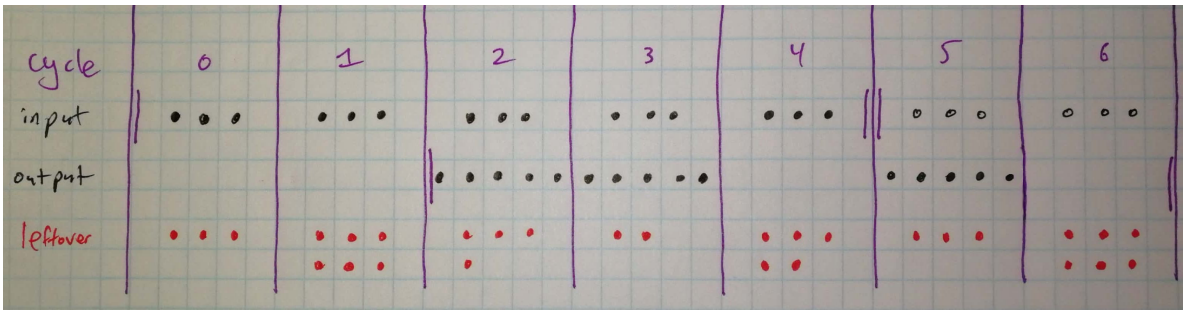
If you think about the dot chart from earlier, each input, output, and leftover cell will have a multiple-of-n number of dots. The chart would be equivalent if we represented each group of n dots as a single dot, matching the original X-array to Y-array repack's chart, and therefore its schedule.

[14] As explained in another footnote it doesn't really matter if these fractions are reduced, but it makes the example cleaner.



Example of underutilized 3-sequence of 4-array to 4-sequence of 3-array `SequenceArrayRepack` (`cps=6`). The bottom chart shows the schedule of the actual repack being analyzed (negative entries on the 0-latency leftover row illustrate the problem of array entries travelling backwards in time without added latency). The top two charts represent the imaginary line buffers used to determine the input and output phase.

As another example, consider the widening counterpart of the first line buffer example. This widening line buffer converts from 5-sequences of 3-arrays to 3-sequences of 5-arrays, with `cps=5`. (Figure 4)



Widening repack schedule (`SequenceArrayRepack (5,3) (3,5)`).

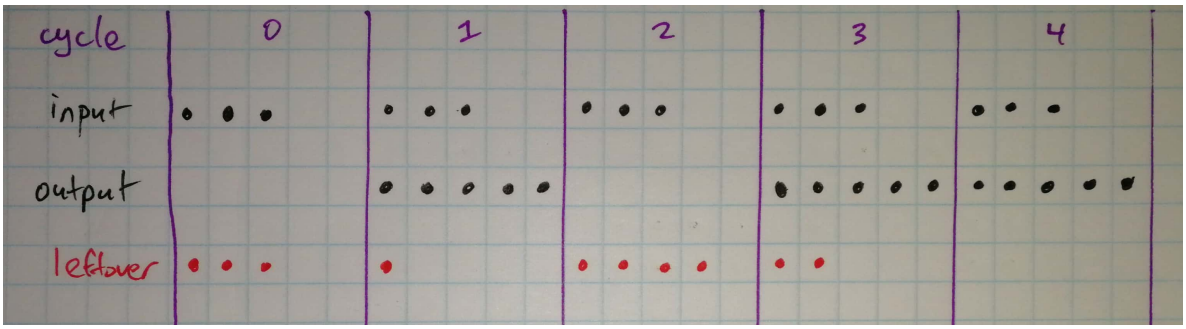
Figure 4: `sequentialLatency=2`; the initial valid output comes 2 cycles after the initial valid input.

Here, the input phase is `[True]` based on input throughput 1, and the output phase is `[True, True, False, True, False]` based on looking up the phase for output throughput $\frac{3}{5}$. This phase matches what the corresponding narrowing repack (`SequenceArrayRepack (3,5) (5,3)`) expects – indeed, under this plan, this widening repack can have its schedule matched with any other op with throughput $\frac{3}{5}$.

7.4 Rationale & Alternatives

The issue with requiring a consistent phase pattern across ops is that an op may be forced to have more latency (and therefore more buffer space needed) than theoretically required. For `SequenceArrayRepack (3,5) (5,3)`, (Figure 4) the `sequentialLatency` cannot be decreased to 1 with this output phase (we'd have negative leftover inputs at cycle 2). With `sequentialLatency=2`, the widening repack always has at least one value buffered on each clock cycle, and, on cycles 1,4,6... will have a full output array's worth of leftover outputs. In principle latency and buffering could be reduced by emitting arrays at cycles 1, 3, and 4 instead of 2, 3, and 5, adopting the alternate phase

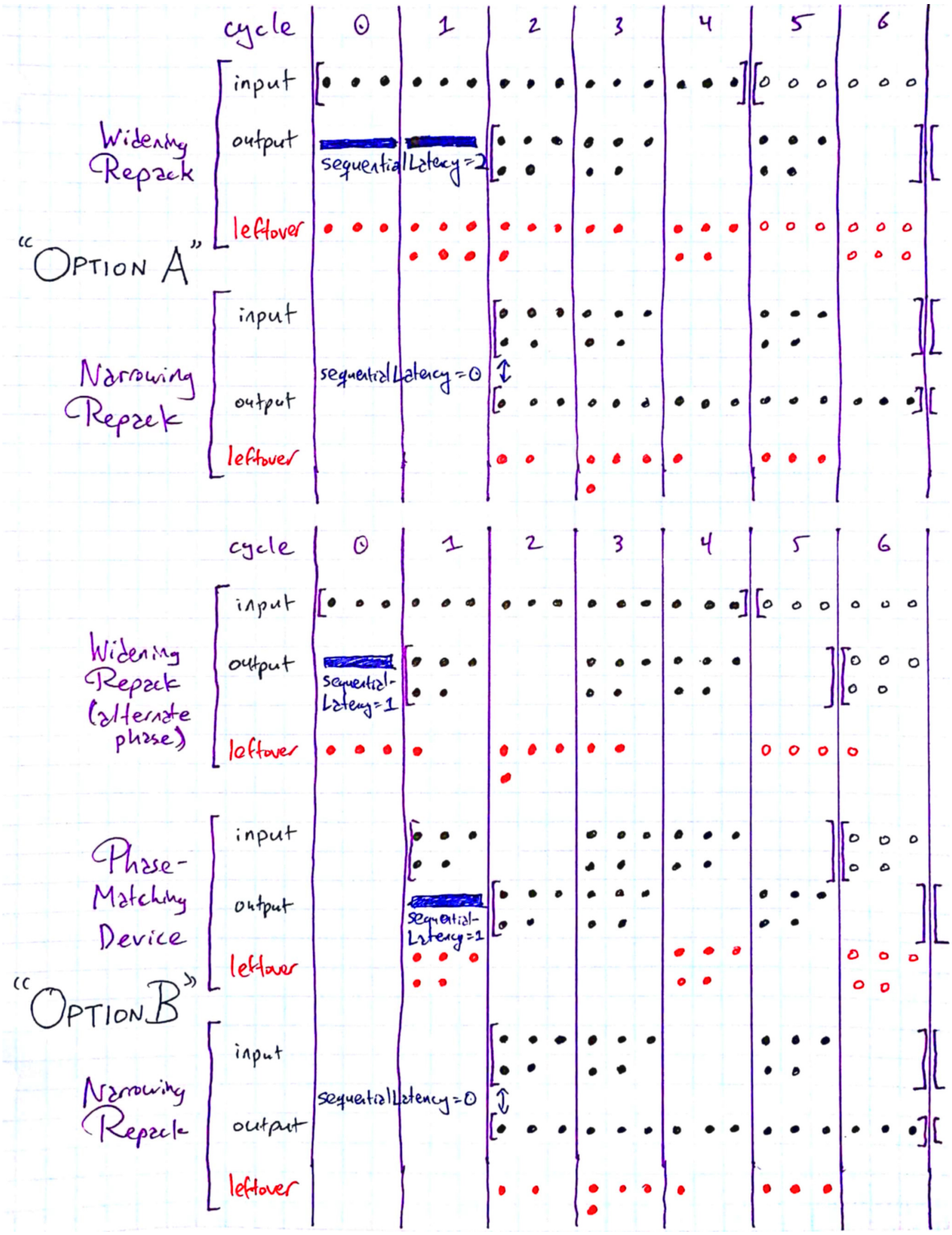
```
[True, False, True, True, False]
```
(10)



The problem with not matching phase is that, while translating Haskell IR to actual hardware, we'll have to add additional phase-matching buffers between ops with matching latencies but different phase patterns.

As an example, consider two options for wiring a widening repack to its corresponding narrowing repack.[15] Option A is for the widening repack to adopt the phase that the narrowing repack expects (as I propose). Option B is for the widening repack to adopt the alternate output phase at (10), and for a phase-matching device to be placed between the two repack ops.

[15]This is a contrived example, but the costs illustrated also apply to more realistic circuits where there is logic inserted between the two `SequenceArrayRepack`s.



Comparison of two options: top is my proposal, botton is rejected alternative.

Information flows from top-to-bottom.

First, at least in this example, the latency decrease of the alternate widening `SequenceArrayRepack (5,3) (3,5)` is cancelled out by the 1 latency increase introduced by the phase matching device. I suspect that any circuit area decrease by reducing buffer space in the widening repack would also be cancelled out by the buffers in the phase matching device. In fact, the area of option B is probably higher that option A, because the phase matching device would need its own duplicate of the muxes and counters that the repack devices presumably use in their implementation. Think of it this way: conceptually, the widening repack in option A combines the functionality of the phase-matching device and widening repack in option B, using only one counter and set of muxes to do so.

In my view though, the major issue with automatically adding phase-matching devices is that this adds extra cost and latency not visible in the Haskell IR, which

1. makes it more difficult for the user to reason about the latency and area of a circuit represented in Haskell IR form.

2. makes my `ComposePar` retiming passes less useful (or much more difficult to write), with the true latency of different paths of the circuit impossible to compute from the Haskell IR alone, which does not provide phase information.

Another alternative I considered and rejected was a simpler throughput-to-phase mapping (or at least a mapping simpler to explain). The phase pattern for a $\frac{x}{Y}$ throughput could have just been x valid values followed by $Y - x$ garbage values. The issues with this approach are:

1. This would increase the buffer space needed in a `SequenceArrayRepack` (which is ultimately the source of fractional underutilization). Intuitively, even for widening repacks and underutilized repacks, for which my proposed phase would not be the one that is theoretically most efficient, it would still be more expensive to have all inputs or outputs scheduled at the start or end of a pattern rather than spread out through time, as my proposed phases require.

2. Using this mapping, we could only assume that two ports with the same throughput will have matched phases if the throughput fraction $\frac{x}{Y}$ is always in reduced form. The actual throughput-to-phase mapping I propose does not require fractions to be in reduced form; for non-reduced fractions $\frac{nx}{nY}$ my proposed mapping naturally creates a phase that just repeats itself n times. This seemed more elegant to me than manually reducing fractions.[16]

[16] In hindsight this isn't all that great a reason (the buffer space reason is more practical), but I include it because this is probably the real reason I prefer the more complicated-to-explain throughput-to-phase mapping I proposed.

8 Test Writing

The tests I designed include

1. Haskell tests for the functional simulator I designed.
2. Python tests for the hardware line buffer David Durst is implementing in Magma based on "The Line Buffer Manifesto" specifications. These tests use the CoreIR simulator.
3. Tests for the sequential compose (`|>>=|`) and parallel compose (`|&|`) Haskell Aetherling operators. These tests check that the produced Haskell IR node is correct given valid operands, and check that the operators reject invalid operands (mismatched port types, mismatched synchronous and ready-valid timing, and mismatched throughputs with synchronous timing).
4. Tests for the `ComposePar` retiming passes. There are 18+ test cases, some of which have compose ops nested several layers deep meant to check for corner cases.