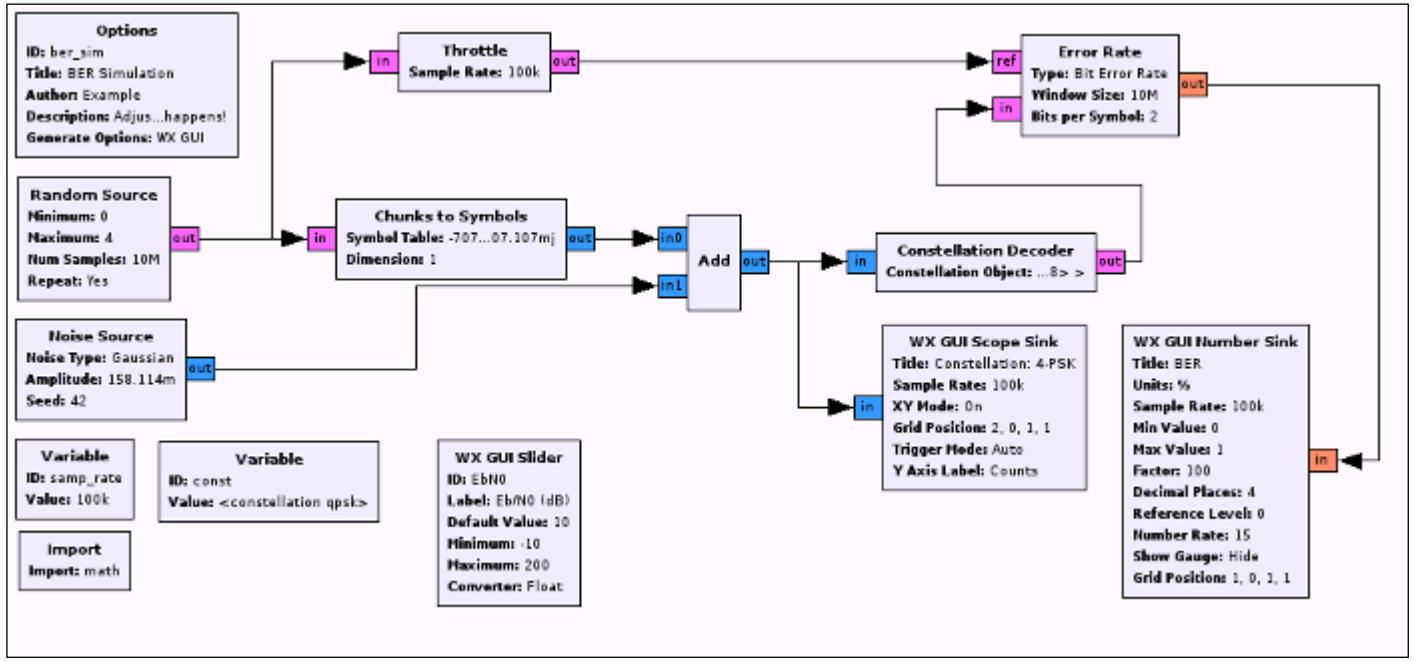
The State of GNU Radio Accelerator Device Support

Presented by: David Sorber

The Problem: Intro

GNU Radio is a software-defined radio (SDR) framework that uses a block-based interface. It includes a library of processing blocks that can be interconnected in various ways to create signal processing flowgraphs.



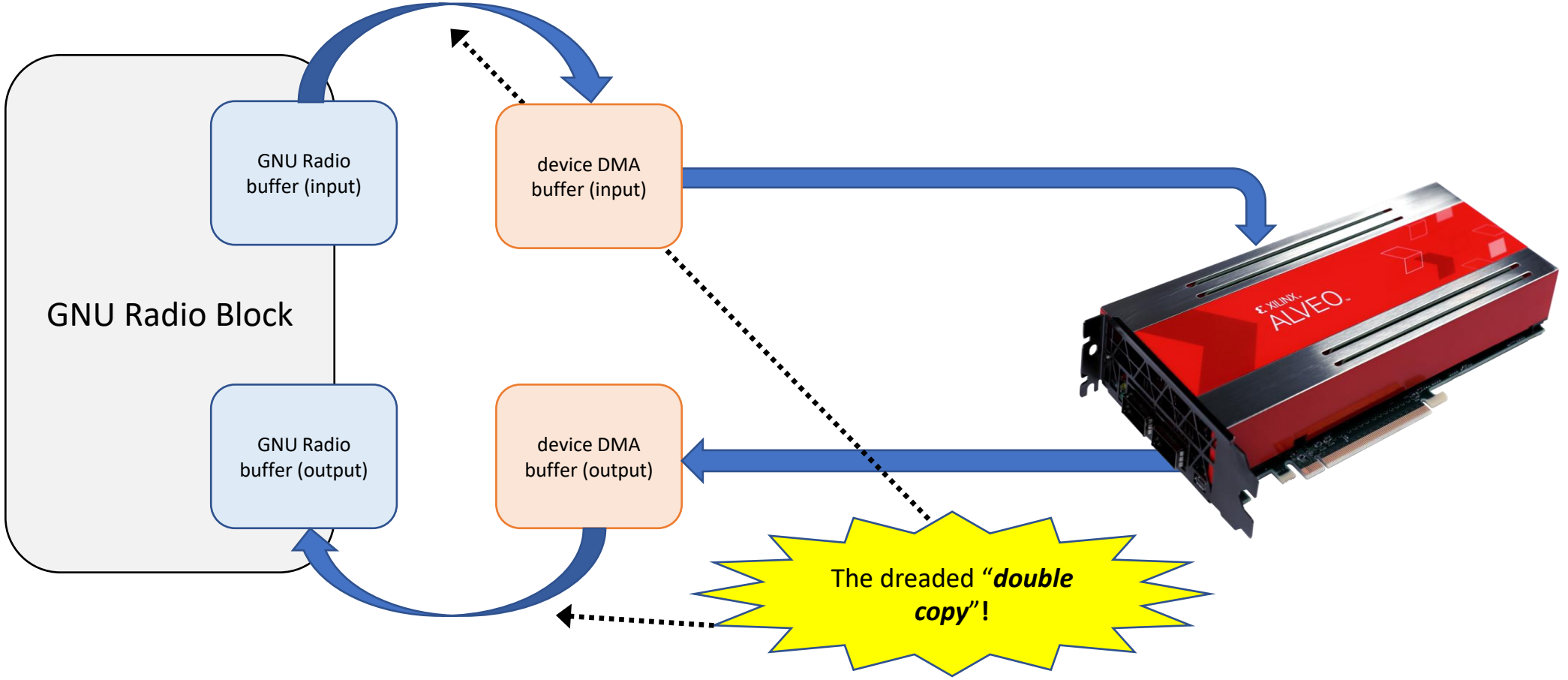
The Problem

Although not explicitly supported, the GNU Radio block-based interface allows signal processing blocks to offload their processing to external “accelerator” hardware including GPUS, FPGAs, and DSPs.

The efficiency of data transfer to and from accelerator devices is often suboptimal because the GNU Radio scheduler controls allocation of memory buffers.

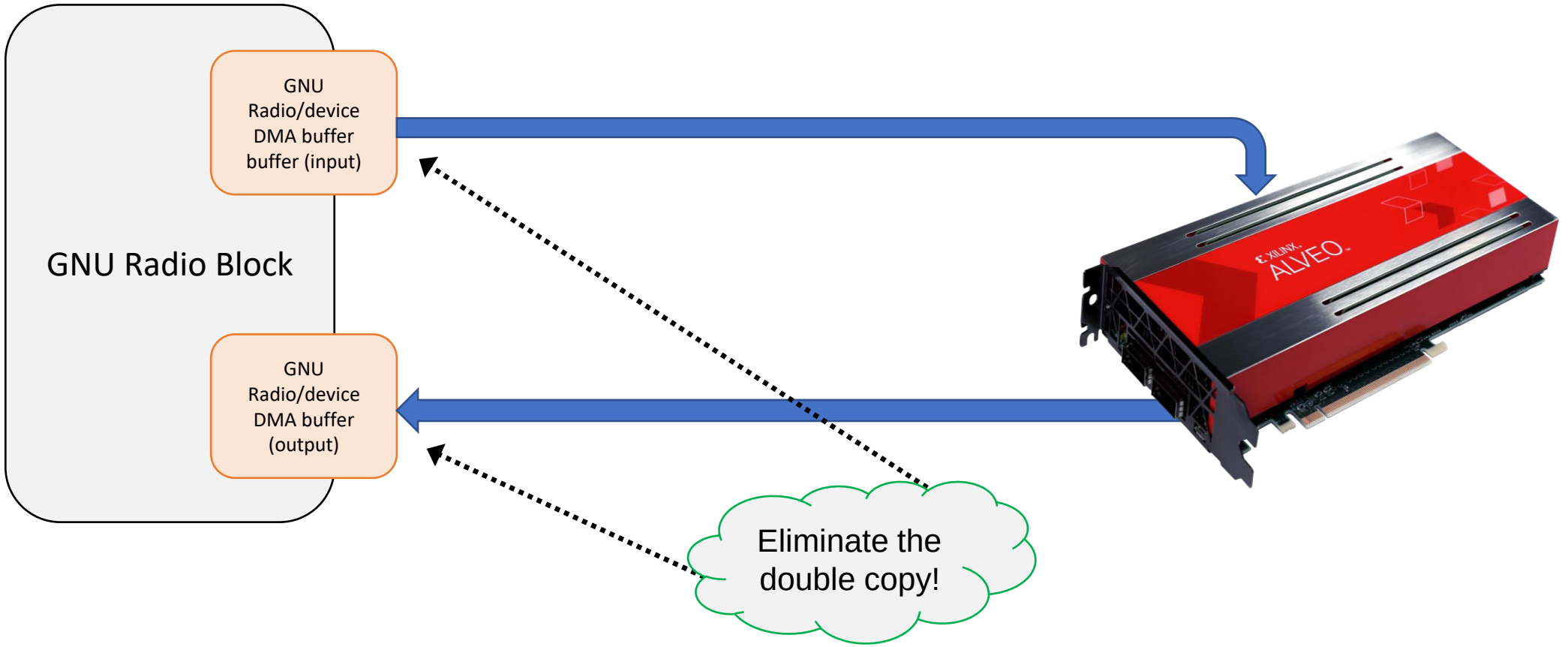
The Problem: Suboptimal

Many accelerator devices require dedicated memory buffers to transfer data; this leads to the double copy problem.



The Problem: A better way

Eliminate the double copy by allowing GNU Radio to manage “custom” (device, DMA, etc.) buffers.



Project: Objectives

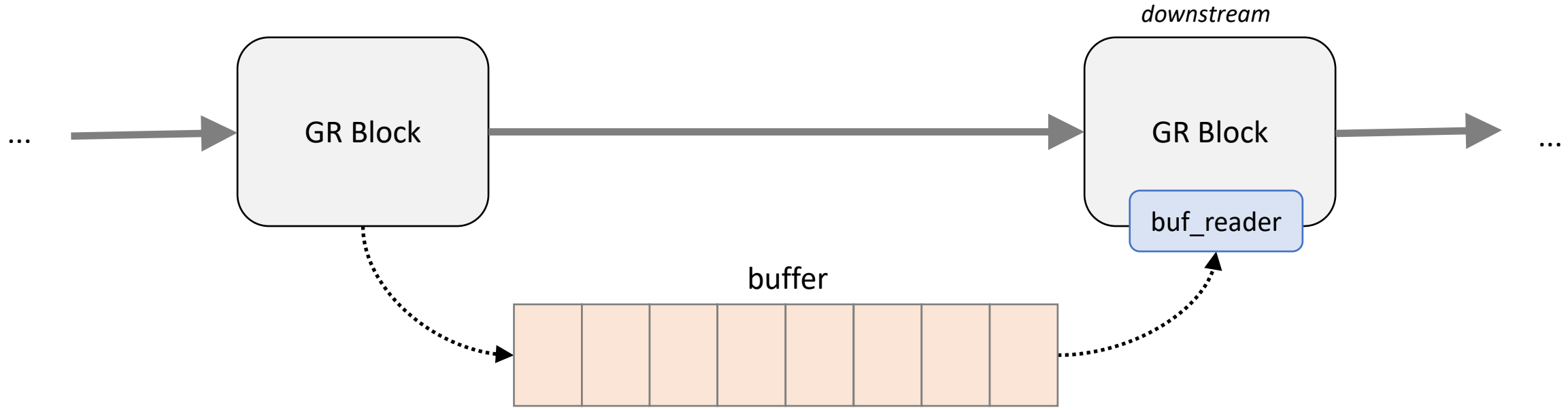
- Maintain backwards compatibility with all existing blocks both in-tree and from OOT modules (*important!*)
- Create flexible interface for creating "custom buffers" to support accelerated devices
    - Custom buffer interface should provide necessary hooks to allow the runtime to handle data movement
- Provide infrastructure to support "*insert signal processing here*" paradigm for common accelerator devices such as NVidia GPUs
- Work with GNU Radio development team; upstream changes

Background: Block interconnect

GR blocks are connected to create a flow graph.

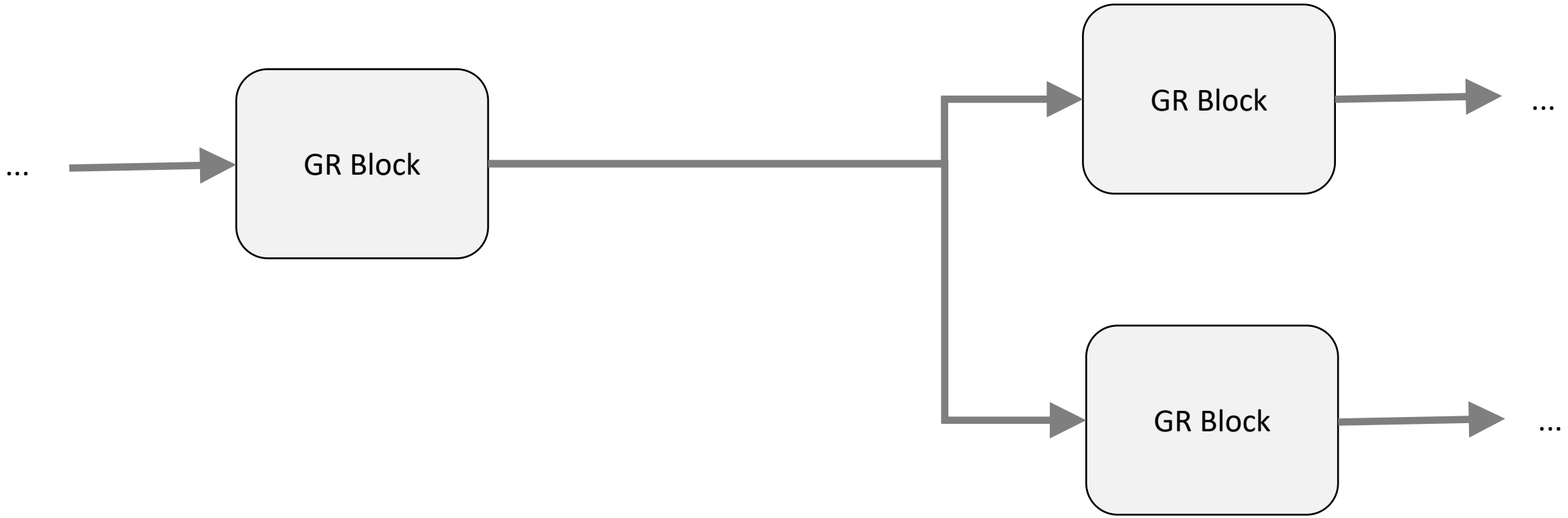
Background: Block interconnect pt. 2

"Under the hood" a block writes its output data to a buffer. A *downstream* buffer reader consumes (reads) data from the same buffer. Each underlying buffer has a wrapper that allows it to be used in a circular fashion.



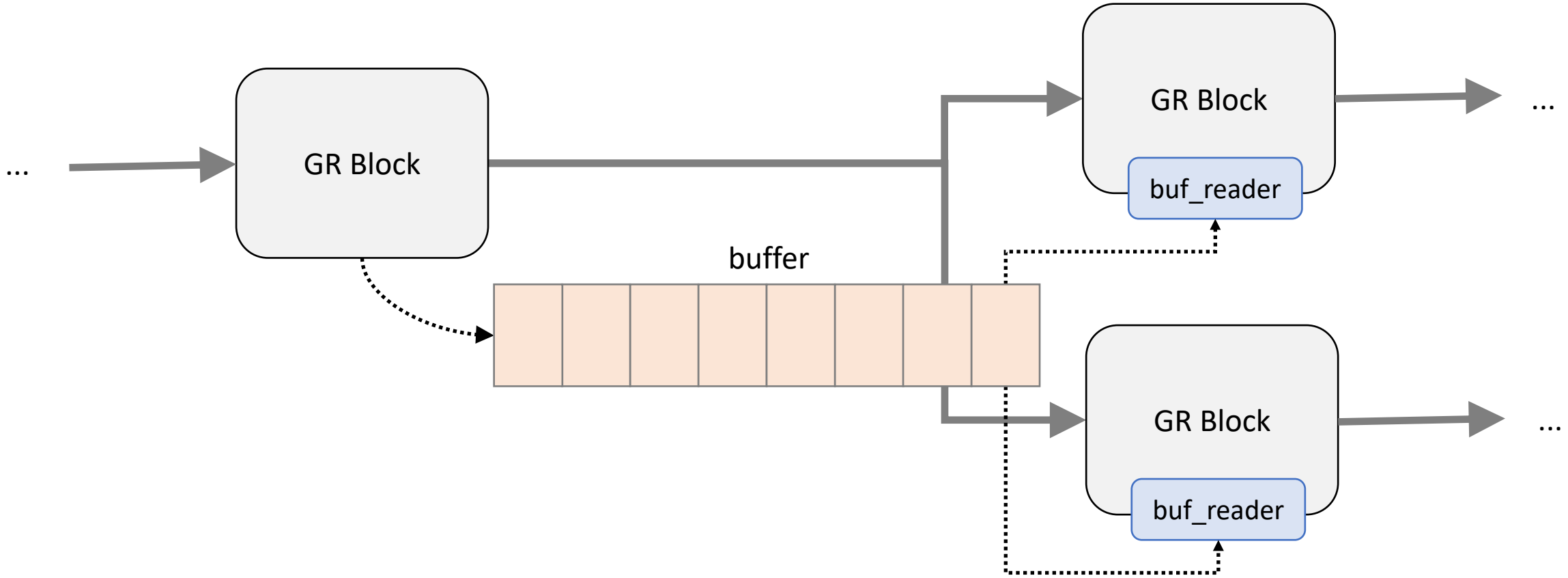
Background: Block interconnect pt. 3

One upstream block and feed multiple downstream blocks.



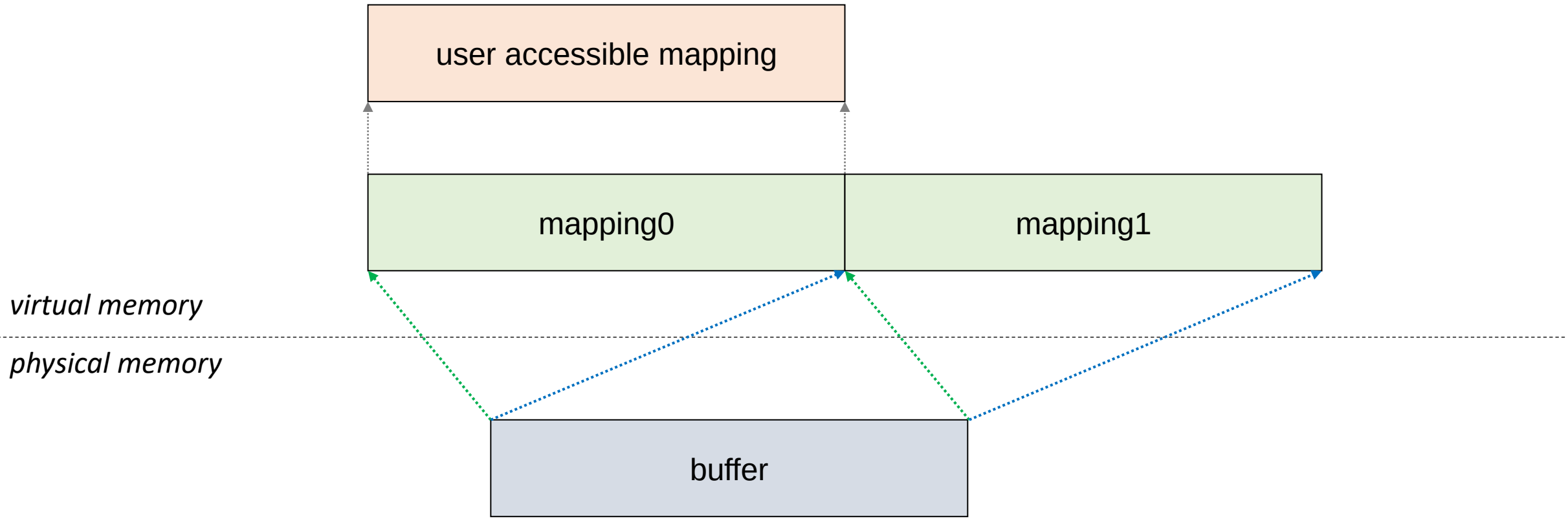
Background: Block interconnect pt. 4

Each downstream block has a separate read pointer.



Background: Double-mapped buffers

GR circular buffers (`vmcircbuf`) use a very clever double mapping scheme.

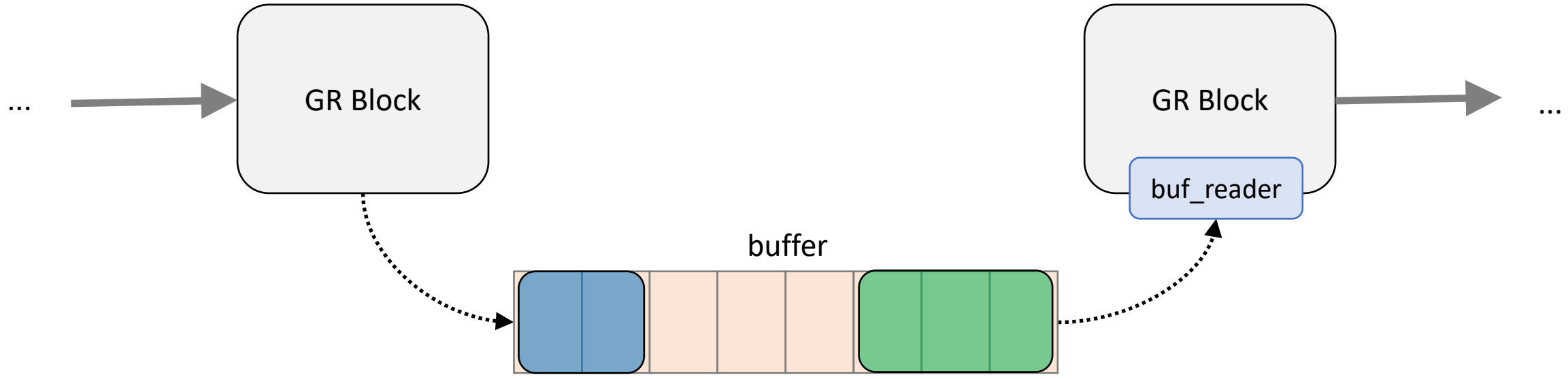


Design: Development Approach

- **Milestone1**
  - Create single mapped buffer abstraction
  - Create initial simplified custom buffer interface
    - Eliminate double copy problem
    - Blocks responsible for data movement
  - Test, benchmark, refine
- **Milestone2**
  - Refine custom buffer interface
    - Offload data movement from block to runtime
    - Support device zero copy transfers
  - Test, benchmark, refine

Design: Single-mapped buffer

- Abstraction created to provide encapsulation for underlying buffers who lack a virtual memory mapping or whose virtual memory mapping cannot be manipulated
- Unlike double mapped buffers, must manage wrapping explicitly

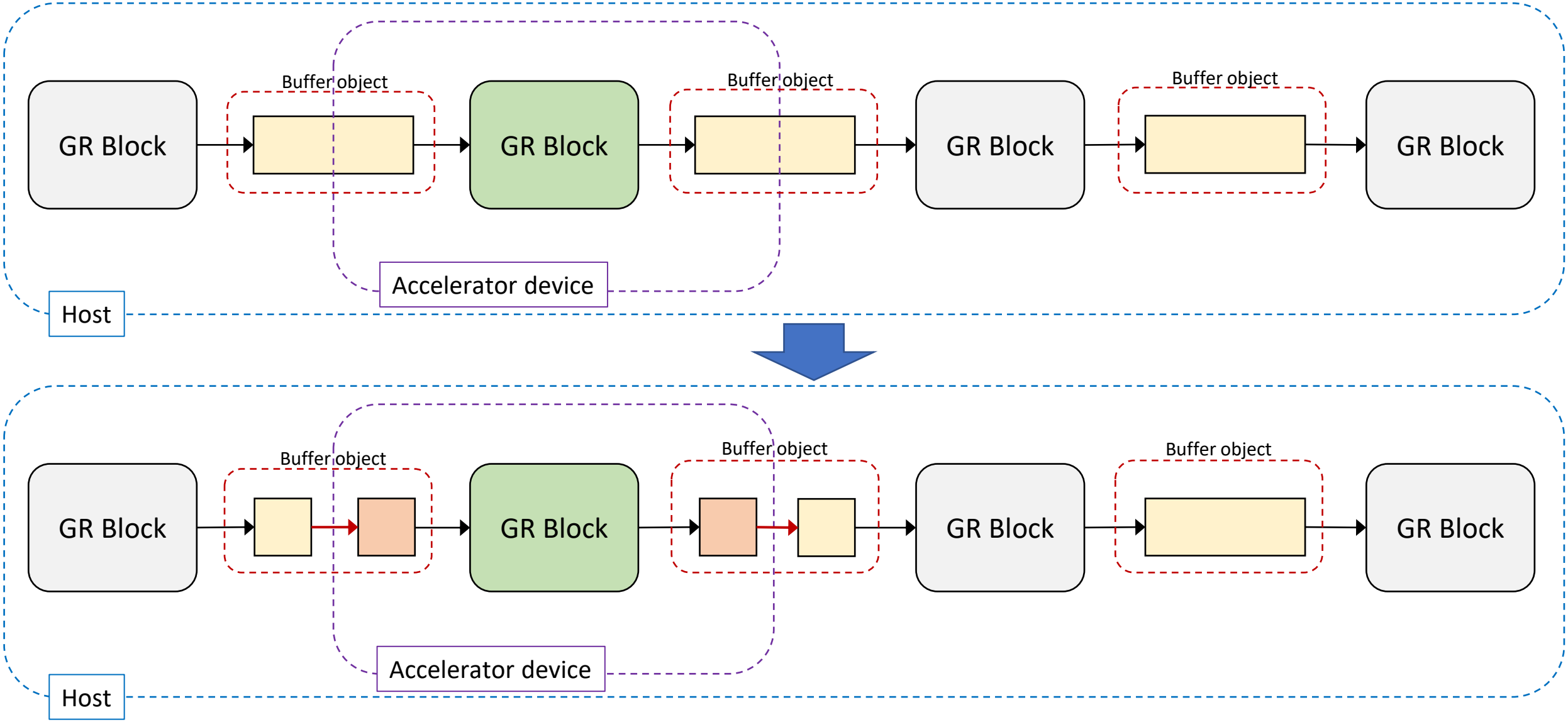


- The read and write granularity of the upstream and downstream blocks may be different! This causes problems when managing wrap around case.

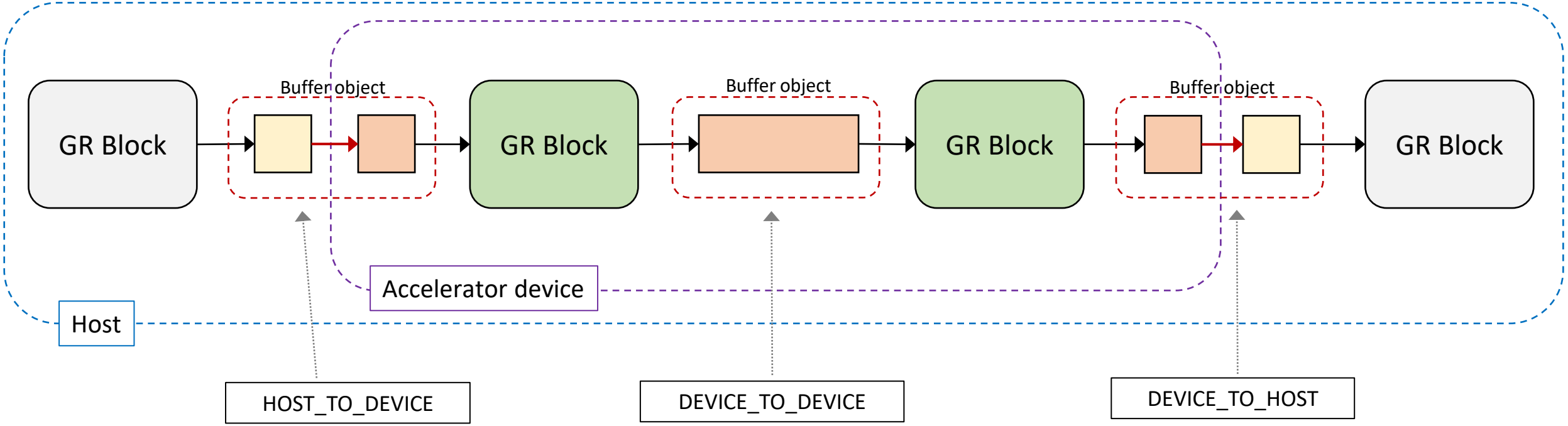
Design: Custom Buffer Interface

- Each custom buffer wrapper is really a pair of buffers:
  - A single mapped host buffer (might be unused)
  - A device buffer
- The custom buffer interface provides:
  - Hook to control device memory allocation
  - Hook to allow the runtime to move data back and forth across the host/device boundary. The "`post_work()`" function is called by the runtime after `general_work()` to perform data movement.
  - Templated callback functions to handle input/output (reader/writer) blocking cases for the single mapped buffer.
- Nearly all interface functions are marked `virtual` so they can be overridden (fully customized) if needed.

Design: Custom Buffer Interface



Design: Back-to-back Accelerated Blocks



Design: Device Runtimes/Frameworks

- **What works with custom buffer interface:**
  - NVidia CUDA
  - AMD HIP
  - custom embedded driver + library
  - *any runtime or framework using pointer-based data manipulation should work*

- **What doesn’t work with custom buffer interface:**
  - OpenCL (still works with legacy double copy)

Benchmarking: Platforms

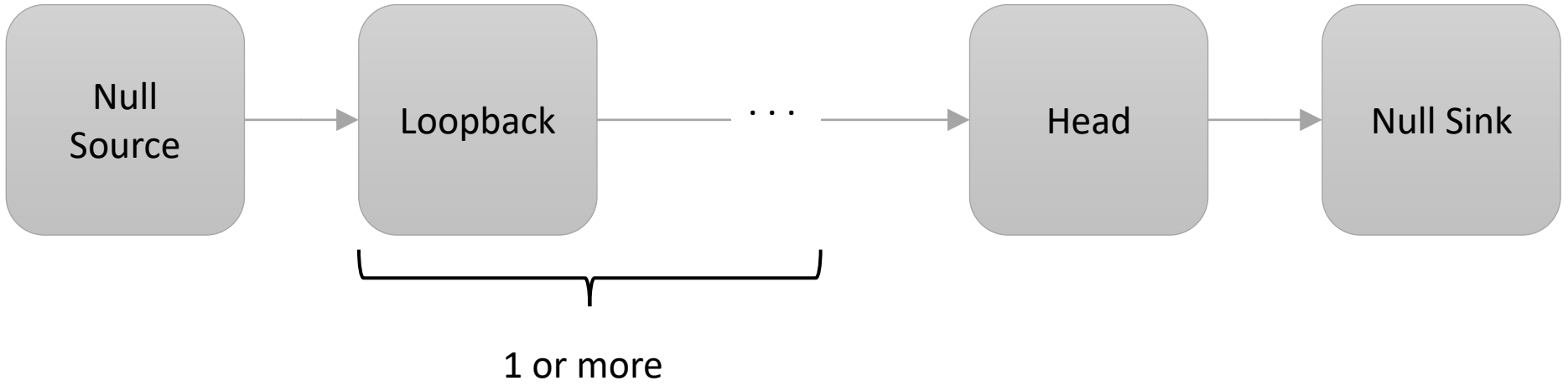
- **Test Systems**
  - Dell r740 server + NVidia V100
  - ~~Dell r740 server + AMD Instinct MI50~~ – hardware issue with card
  - Dell XPS15 laptop with discrete NVidia GTX 1650 GPU
  - NVidia Jetson AGX Xavier
  - Xilinx ZCU106 development board (custom driver and FPGA firmware)

NVidia V100

NVidia Jetson AGX Xavier

Xilinx ZCU106

Benchmarking: Methodology



- Run using standardized benchmark scripts
- Three cases using one or more loopback blocks:
  - stock GR 3.9.2.0 + legacy (double copy)
  - ngsched + legacy (double copy)
  - ngsched + custom buffer (single mapped)
- Durations measured in ms; average of 10 runs

Remember Amdhal's Law when thinking about overall speedup!

Benchmarking: Dell r740 + NVidia V100

nblocks	batch size	bytes	GR39+legacy	ngsched+_legacy	delta	ngsched+custbuff	delta
1	1024	8192	1.417	1.412	0.34%	0.898	57.78%
1	2048	16384	0.934	0.932	0.18%	0.718	30.07%
1	4096	32768	1.135	1.137	-0.24%	0.559	103.12%
1	8192	65536	0.680	0.679	0.12%	0.328	107.11%
1	16384	131072	0.453	0.437	3.62%	0.206	120.24%
1	32768	262144	0.371	0.369	0.42%	0.164	125.69%
1	65536	524288	0.327	0.328	-0.06%	0.151	116.68%
2	1024	8192	2.838	2.842	-0.15%	0.975	191.05%
2	2048	16384	1.808	1.807	0.01%	0.808	123.79%
2	4096	32768	1.928	1.924	0.16%	0.607	217.44%
2	8192	65536	1.221	1.220	0.16%	0.390	213.26%
2	16384	131072	0.851	0.852	-0.08%	0.266	220.40%
2	32768	262144	0.750	0.752	-0.28%	0.210	257.76%
2	65536	524288	0.659	0.662	-0.40%	0.192	242.46%
4	1024	8192	5.469	5.574	-1.89%	1.455	275.81%
4	2048	16384	2.957	2.962	-0.15%	1.116	165.12%
4	4096	32768	2.705	2.685	0.75%	0.718	276.73%
4	8192	65536	1.825	1.825	0.03%	0.463	293.77%
4	16384	131072	1.252	1.217	2.87%	0.317	294.62%
4	32768	262144	0.992	0.992	0.03%	0.242	310.88%
4	65536	524288	0.867	0.868	-0.08%	0.212	309.53%
16	1024	8192	25.150	25.073	0.31%	4.857	417.80%
16	2048	16384	13.092	12.836	2.00%	3.139	317.09%
16	4096	32768	7.652	7.586	0.87%	1.623	371.41%
16	8192	65536	5.452	5.237	4.10%	0.858	535.68%
16	16384	131072	4.378	3.837	14.11%	0.543	705.86%
16	32768	262144	3.217	2.929	9.81%	0.341	844.68%
16	65536	524288	2.795	2.955	-5.42%	0.288	868.98%

Benchmarking: Dell XPS15 + NVidia GTX 1650

nblocks	batch size	bytes	GR39+legacy	ngsched+legacy	delta	ngsched+custbuff	delta
1	1024	8192	1.268	1.273	-0.44%	0.780	62.46%
1	2048	16384	1.239	1.241	-0.15%	0.681	81.98%
1	4096	32768	1.004	1.006	-0.28%	0.529	89.66%
1	8192	65536	0.648	0.649	-0.02%	0.352	84.30%
1	16384	131072	0.558	0.563	-0.88%	0.258	116.24%
1	32768	262144	0.452	0.451	0.14%	0.205	119.93%
1	65536	524288	0.398	0.396	0.51%	0.178	123.50%
2	1024	8192	2.565	2.562	0.10%	0.790	224.75%
2	2048	16384	1.941	1.925	0.83%	0.637	204.50%
2	4096	32768	1.770	1.825	-3.03%	0.553	219.85%
2	8192	65536	1.192	1.209	-1.41%	0.353	238.05%
2	16384	131072	1.060	1.062	-0.23%	0.265	300.14%
2	32768	262144	0.912	0.908	0.44%	0.209	336.07%
2	65536	524288	0.835	0.839	-0.50%	0.183	356.28%
4	1024	8192	4.450	4.427	0.51%	0.951	368.12%
4	2048	16384	2.631	2.623	0.30%	0.745	253.35%
4	4096	32768	2.324	2.394	-2.95%	0.585	297.33%
4	8192	65536	1.782	1.752	1.73%	0.382	366.85%
4	16384	131072	1.378	1.384	-0.38%	0.273	404.54%
4	32768	262144	1.210	1.191	1.57%	0.218	456.03%
4	65536	524288	1.089	1.080	0.86%	0.186	486.44%
16	1024	8192	18.211	18.303	-0.50%	2.889	530.31%
16	2048	16384	11.216	11.212	0.04%	1.944	477.05%
16	4096	32768	7.373	7.394	-0.28%	1.116	560.69%
16	8192	65536	5.487	5.496	-0.16%	0.623	780.73%
16	16384	131072	4.241	4.219	0.52%	0.391	984.43%
16	32768	262144	3.629	3.631	-0.04%	0.280	1198.38%
16	65536	524288	3.377	3.377	0.02%	0.232	1358.07%

Benchmarking: NVidia Jetson AGX Xavier

nblocks	batch size	bytes	GR39+legacy	ngsched+legacy	delta	ngsched+custbuff	delta
1	1024	8192	8.619	8.993	-4.16%	5.280	63.25%
1	2048	16384	5.118	5.460	-6.28%	4.798	6.67%
1	4096	32768	4.209	4.415	-4.67%	3.471	21.27%
1	8192	65536	2.834	2.946	-3.80%	2.111	34.28%
1	16384	131072	1.612	1.664	-3.14%	1.373	17.44%
1	32768	262144	1.148	1.168	-1.73%	0.879	30.59%
1	65536	524288	0.920	0.952	-3.39%	0.717	28.30%
2	1024	8192	22.320	23.707	-5.85%	9.487	135.28%
2	2048	16384	11.483	11.885	-3.38%	6.752	70.07%
2	4096	32768	6.399	6.708	-4.61%	4.067	57.34%
2	8192	65536	4.400	4.707	-6.52%	2.334	88.53%
2	16384	131072	3.170	2.942	7.76%	1.545	105.14%
2	32768	262144	2.130	2.135	-0.25%	0.896	137.76%
2	65536	524288	1.626	1.715	-5.17%	0.752	116.30%
4	1024	8192	54.087	52.698	2.64%	21.004	157.51%
4	2048	16384	27.688	27.346	1.25%	12.161	127.68%
4	4096	32768	15.297	14.228	7.51%	6.790	125.27%
4	8192	65536	9.427	8.966	5.14%	3.655	157.92%
4	16384	131072	6.734	6.727	0.10%	2.105	219.86%
4	32768	262144	4.481	4.532	-1.12%	1.292	246.86%
4	65536	524288	3.380	3.322	1.75%	0.759	345.40%
16	1024	8192	218.267	214.857	1.59%	82.309	165.18%
16	2048	16384	112.618	108.324	3.96%	47.730	135.95%
16	4096	32768	62.136	63.191	-1.67%	25.109	147.47%
16	8192	65536	39.597	38.548	2.72%	13.519	192.89%
16	16384	131072	25.492	26.775	-4.79%	6.938	267.44%
16	32768	262144	17.554	16.992	3.31%	3.096	466.95%
16	65536	524288	10.431	10.259	1.68%	1.546	574.79%

Benchmarking: Xilinx ZCU106 dev board

nblocks	batch size	bytes	GR39+legacy	ngsched+legacy	delta	ngsched+custbuff	delta
1	1024	8192	8.622	8.638	-0.18%	6.021	43.20%
1	2048	16384	7.461	7.552	-1.20%	6.309	18.28%
1	4096	32768	7.432	7.527	-1.26%	6.213	19.63%
1	8192	65536	6.657	6.680	-0.34%	6.307	5.55%
1	16384	131072	6.250	6.251	-0.01%	6.089	2.66%
1	32768	262144	6.152	6.161	-0.15%	6.024	2.11%
1	65536	524288	6.296	6.349	-0.83%	5.989	5.12%
2	1024	8192	10.815	10.924	-1.00%	6.614	63.51%
2	2048	16384	10.655	10.728	-0.68%	6.516	63.52%
2	4096	32768	8.685	8.798	-1.28%	6.562	32.36%
2	8192	65536	7.408	8.609	-13.95%	6.579	12.60%
2	16384	131072	6.939	7.272	-4.57%	6.243	11.16%
2	32768	262144	7.575	7.595	-0.26%	6.158	23.01%
2	65536	524288	7.796	7.936	-1.77%	6.090	28.00%
4	1024	8192	14.268	14.429	-1.12%	7.643	86.67%
4	2048	16384	14.129	14.129	0.00%	6.528	116.45%
4	4096	32768	11.900	12.422	-4.20%	6.570	81.13%
4	8192	65536	10.060	10.692	-5.91%	6.560	53.34%
4	16384	131072	9.977	10.347	-3.57%	6.386	56.24%
4	32768	262144	9.847	10.868	-9.40%	6.336	55.41%
4	65536	524288	9.906	9.961	-0.55%	6.247	58.56%
8	1024	8192	26.841	27.182	-1.26%	11.750	128.43%
8	2048	16384	23.495	23.859	-1.52%	8.952	162.46%
8	4096	32768	21.552	21.986	-1.97%	7.341	193.58%
8	8192	65536	20.294	22.196	-8.57%	7.192	182.19%
8	16384	131072	19.783	20.971	-5.67%	6.736	193.68%
8	32768	262144	19.148	19.349	-1.04%	6.679	186.70%
8	65536	524288	18.959	19.079	-0.63%	6.598	187.35%

How to Use a Custom Buffer: A very quick tutorial

1. Install OOT module containing custom buffer class. For example, gr-cuda_buffer (**NOTE:** “ngsched” changes are required)
2. In the desired block source file add include for buffer class header file:

```
#include <cuda_buffer/cuda_buffer.h>
```

3. Within the block’s constructor update the `gr::iosignature` to include the desired buffer’s type:

```
new_block_impl::new_block_impl()
    : gr::block("my_new_block",
                gr::io_signature::make(1 /* min inputs */, 1 /* max inputs */,
                                       sizeof(input_type), cuda_buffer::type),
                gr::io_signature::make(1 /* min outputs */, 1 /*max outputs */,
                                       sizeof(output_type), cuda_buffer::type))
{
    . . .
}
```

How to Use a Custom Buffer: A very quick tutorial

4. The pointers passed to the block's work function will now be of the selected type.

```
int new_block_impl::general_work(int noutput_items,
                                 gr_vector_int& ninput_items,
                                 gr_vector_const_void_star& input_items,
                                 gr_vector_void_star& output_items)
{
    // NOTE: in and out are *not* host accessible
    const auto in = reinterpret_cast<const input_type*>(input_items[0]);
    auto out = reinterpret_cast<output_type*>(output_items[0]);

    // Launch kernel passing in and out as parameters
    . . .
}
```

How to Create a Custom Buffer: Not a tutorial

- The custom buffer interface allows creation and use of custom buffer classes
- Custom buffer classes can be defined in OOT modules (no longer just for blocks)
- Subclass `buffer_single_mapped` and use the `MAKE_CUSTOM_BUFFER_TYPE()` macro
- Simple example custom buffer called “custom_buffer”
  - See `custom_buffer.h` and `custom_buffer.cc` in gr-blnxngsched

The Code

- Available here:
  - In the “merge_ngsched” branch of my GNU Radio fork:
    - https://github.com/dsorber/gnuradio
  - Also here:
    - https://github.com/gnuradio/gnuradio-ngsched/
- Detailed technical documentation:
  - https://github.com/gnuradio/gnuradio-ngsched/wiki
- Pull request created:
  - https://github.com/gnuradio/gnuradio/pull/5028

Additional Related Code

- gr-cuda_buffer - contains an OOT module containing the `cuda_buffer` class which is a "custom buffer" supporting the CUDA runtime for NVidia GPUs. This module is intended to be a base CUDA buffer implementation for CUDA blocks and can be used directly when writing CUDA accelerated blocks for NVidia GPUs.
  - https://github.com/BlackLynx-Inc/gr-cuda_buffer

- gr-blnxngsched - contains an OOT module containing various examples of the accelerated block interface (aka "ngsched") changes. Note that the CUDA-related blocks in this OOT require cuda_buffer from gr-cuda-buffer.
  - https://github.com/BlackLynx-Inc/gr-blnxngsched

Conclusion

- Thank you for listening!
- David Sorber
  - david.sorber@gmail.com
  - dsorber on matrix

- Thanks to Josh Morman and Seth Hitefield for their support, feedback, and assistance during this project.