Flowgraph Acceleration with GPUs:
Analyzing the Benefits of Custom Buffers in GNU Radio

Seth D. Hitefield SETH.HITEFIELD@VT.EDU
T. Charles Clancy TCC@VT.EDU
Hume Center, Virginia Tech, 1991 Kraft Dr., Suite 2019, Blacksburg, VA 24060 USA

Abstract

Recently, there have been major improvements in graphics processing (GPU) hardware, making it an excellent tool for accelerating large parallel applications. Some discrete graphics cards can have thousands of individual cores and reach teraflops of computing power in contrast to a general purpose CPU. Since many digital signal processing operations are inherently parallelizable algorithms, GPUs are interesting candidates for also accelerating software radios.

However, there are a couple of challenges to utilizing GPUs in a software radio framework such as GNU Radio: specifically, memory management and kernel scheduling overhead. The custom buffer feature being developed for the GNU Radio runtime helps simplify these issues by allowing blocks to allocate specialized memory buffers that the runtime manages.

This paper discusses the challenges of integrating GPUs into applications and explores the benefits of using custom buffers with GPU based blocks. It presents some performance comparisons of several implementations of a very simple flowgraph that demonstrate the advantage of using custom buffers and GPUs for acceleration.

1. Introduction

Over the past few years there have been major advances in graphics processor hardware (GPUs) and software frameworks that can be used to accelerate many types of general purpose computing. GPUs can have different form factors such as an integrated, embedded/mobile system-on-chip, though the most common package is a discrete PCIe card. These cards (like NVIDIA's Titan X) contain thousands of computing cores (3,584) and reach teraflops of computational power in a relatively small, power efficient package (in comparison to the equivalent number of general purpose CPUs)(NVIDIA, 2016). GPUs are based on the single instruction, multiple data (SIMD) architecture and are perfect for accelerating many applications with highly parallel algorithms and datasets. They have been utilized in many different scientific disciplines, and in fact, many of the world's top supercomputers are now powered by GPUs.

Since many operations in digital signal processing (DSP) applications are inherently parallelizable (like a channelizer), they can be mapped to a SIMD architecture. There has already been significant work investigating flowgraph acceleration using SIMD hardware extensions available in many modern general purpose processors. Libraries such as the Vector-Optimized Library of Kernels (VOLK, 2016) allow developers to easily integrate these capabilities into their applications in a portable, platform agnostic way.

While these CPU hardware extensions are very useful, their capabilities can be limited in comparison to recent GPU platforms. GPUs' parallel architecture make them an interesting candidate for accelerating software defined radio applications, but there are several challenges with integrating them into existing frameworks like GNU Radio.

This paper will discuss some of those challenges and will briefly introduce the new custom buffer support being developed that allows blocks to allocate specialized memory buffers. The main focus of the paper is analyzing the benefits of using the custom buffer feature to integrate GPUs into a flowgraph. For example, a GPU using a custom buffer can significantly increase the overall throughput of the system without incurring major overhead from additional memory operations. While the concepts can be applied to any GPU platform, this paper will have a focus on embedded, heterogeneous platforms such as NVIDIA's Tegra X1 system-on-chip (NVIDIA, Jan. 2015) using the CUDA framework. Custom buffers are not limited to just GPUs and can also be useful for integrating other types of processors (such as FPGAs or DSP co-processors) into flowgraphs, but that is out of scope for this paper

Section 2.1 briefly discusses VOLK, which is an existing SIMD acceleration library for GNU Radio. Section 2.2 covers the challenges in integrating GPUs into flowgraphs.

Proceedings of the 6th *GNU Radio Conference*, Copyright 2016 by the author(s).

Section 3 will give an overview of the custom buffer support being added to the GNU Radio runtime, and Section 4 presents the performance comparison of VOLK kernel compared to a GPU implementation. The GPU kernels (and terminology) used for performance comparisons in this paper will be using CUDA framework (NVIDIA, Sept. 2015).

2. Background

2.1. VOLK

Most modern CPU architectures support additional hardware SIMD extensions, such as Intel's SSE and AVX and Arm's NEON extensions. These extensions implement specialized instructions and have minimal overhead other than some requirements for memory alignment. There is a learning curve to implementing these extensions, and, once implemented, they can require significant hand tuning to optimize an operation.

VOLK is a subproject of GNU Radio designed to simplify the use of SIMD extensions in software radio applications. It provides an portable, platform agnostic interface to various SIMD-optimized mathematical operations that can be integrated into an application. Many core GNU Radio blocks already use VOLK to accelerate operations on supported systems.

There are two types of abstraction in VOLK: 1) *machines* represent supported architectures, and *kernels* represent specific DSP functions (Rondeau et al.). Machines abstract hardware platforms, which may contain one or more set of different SIMD extensions; also, newer architectures typically support the legacy extensions from previous hardware generations. A kernel consists of multiple *proto-kernels*: the optimized implementations of a DSP function for a specific machine. Dispatchers are automatically generated for each kernel that allow the runtime system to select and call the best available proto-kernel. Generic proto-kernel implementations also exist as a fall back if no other proto-kernel supports the current system. This design provides a platform agnostic method for integrating SIMD capability into DSP pipelines.

2.2. CUDA Programming Model

NVIDIA's CUDA framework provides both a programming model for their GPUs, as well as, the runtime support for general purpose computing on supported systems (NVIDIA, Sept. 2015). The programming model addresses specific concepts of programming GPUs, such as threads, memory access, and synchronization. A CUDA kernel is essentially a specialized *C* function that is executed in parallel based on the number of scheduled threads.

While the concept of a thread does not differ much from a traditional definition, GPU threads are scheduled in a very different manner than on a CPU. They are executed in groups of 32 (warps) on a Streaming Multiprocessor (SM), which could be compared to an individual core of a CPU. Threads can be organized into a multi-dimensional hierarchy composed of blocks and grids to help simplify the mapping of a problem to a parallel model. Each grid eventually gets scheduled to a single GPU and the blocks in the grid are mapped to the GPU's individual SMs; blocks are not split across multiple SMs. The number of cores in each SM depends on the hardware generation; NVIDIA's newest architectures typically have 128 cores/SM and cards like the newest Titan X have 28 SMs.

An example of a CUDA kernel is shown in Section A.1. This kernel is a GPU implementation of the *multiply_const* block, which multiplies input samples by some constant value. The *cuda_kexec()* wrapper function shows the required compiler (*NVCC*) notation for launching a kernel using the CUDA runtime.

2.2.1. Kernel Overhead

Because GPUs have their own execution pipeline separate from the host, launching a kernel can incur significant overhead. Many times, the workload for a GPU kernel is so immense that any overhead from launching kernels is negligible compared to the overall execution time of the kernel. For a software radio application where latency is a huge concern, not correctly accounting for this overhead can significantly reduce the performance of the executing kernels.

Table 1. Launch overhead for a CUDA kernel

Buffer Size	Overhead (μs)
8192	48.7
16384	51.7
32768	50.5
65536	49.9
131072	79.1
262144	146.1
524288	149.1
1048576	183.1

GPUs are capable of handling a significant amount of processing, but there is overhead from launching even the most simple of kernels. For example, on the Tegra X1 platform, benchmarking multiple kernel launches (*multiply_const*) showed an that the overhead for each launch ranged from approximately 50 μs to 180 μs (shown in Table 1) as the size of the scheduled kernel increased. There were sev-

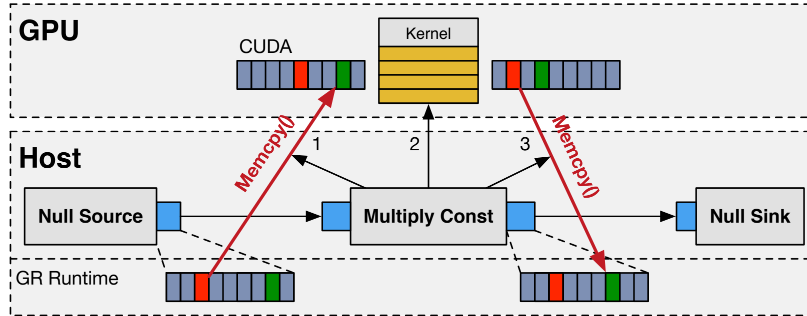

Figure 1. Example memory management for GPU integration in GNU Radio without custom buffer support: 1) Memory is copied to a device accessible buffer, 2) a kernel is executed, and 3) the results are copied back to a host buffer.

eral outliers in the timing data, but there was a clear trend that the launch overhead increased with the buffer size; this was expected as more kernel threads were being scheduled to execute. For single kernel launches 50-180μs is negligible, but the total overhead can significantly increase as the number of launches increases.

This creates a trade-off between the amount of work to be accomplished per kernel call and the amount of latency for the overall system. For example, the default GNU Radio buffer size (on the Tegra X1) for complex floats is 8,192 samples; the runtime attempts to call *work()* after half of the buffer is full. If a kernel were launched for every single *work()* iteration, the system would incur major overhead simply launching kernels. Increasing the amount of work per kernel reduces the overall number of launches but requires larger memory buffers within the system. In fixed rate systems (either input or output), larger buffers increase the overall latency for the entire pipeline; this may not be an optimal solution when low latency is a operational requirement.

Also, the specific dimensions of the grid and blocks for each launch can play a role in overhead and overall execution time of a kernel. If the scheduled dimensionality is greater than the work available, the runtime can waste time scheduling threads that will ultimately have no work to accomplish.

2.2.2. Memory Management

Memory management is also a concern for GPU development; when programming with the CUDA framework, every memory buffer needed must be allocated through CUDA and not through standard methods such as *malloc()*. With the current GNU Radio architecture, the runtime is responsible for allocating and managing the output buffer for each block. In order to use a GPU within this constraint, a developer must allocate one or more buffers accessible to the GPU and copy data back and forth between the GNU Radio and CUDA buffers (Figure 1). This additional copy operation can cause huge overhead and greatly reduce the overall performance of the application.

Handling memory for a discrete GPU can be more complex; in many cases, data must be copied from the host to the device itself before a kernel can be executed. For applications with large datasets and no latency requirements, the optimal solution is copying the entire dataset (or as much of it as possible) to the GPU before launching a kernel. However, for applications with latency requirements (like software defined radio), handling small memory transfers between the host and device for each *work()* iteration can quickly cause significant overhead; this problem would be further compounded by multiple GPU blocks in a flowgraph.

The GR-GPU project from the University of Maryland addressed some of these issues with a new dataflow approach for integrating GPUs into GNU Radio (Plishker et al., 2011). Their approach included two new blocks, *H2D* and *D2H*, which were responsible for managing memory transfers from the GNU Radio allocated buffers on the host to CUDA allocated memory on the GPU. The runtime buffers allocated (on the host) for each block were used to transfer GPU memory pointers between the other CUDA based blocks. Other blocks could then use these pointers when launching kernels without additional memory transfers or allocating other buffers. This approach would be very useful for consecutive CUDA blocks; once data was initially copied to device memory, each downstream block could execute its kernel without an unnecessary copy to host memory and back.

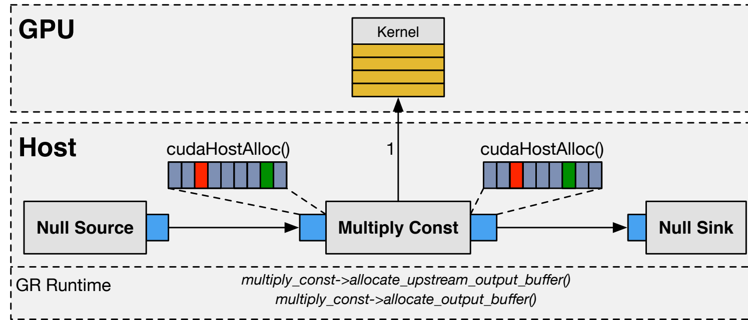

Figure 2. Example memory management for GPU integration with custom buffers. Buffers are allocated by the block in page-locked, host memory that is directly accessible by the GPU.

3. Custom Buffers

The main rationale for supporting custom memory buffers in GNU Radio is allowing blocks to allocate specialized buffers that will be owned by the block but managed by the runtime without changing the underlying flowgraph model presented to developers. A small downside to the GR-GPU approach is that the input/output pointers passed to the *work()* function no longer point to data, but rather point to a list of GPU pointers. This breaks the current flowgraph model, which the custom buffer feature avoids; the call to a block's *work()* function passes a pointer to the data no matter what type of buffer or where it was allocate. There are several use cases for custom buffers, with GPU acceleration being a major example.

An immediate benefit can be seen on platforms with integrated GPUs such as the Tegra X1. Both the CPU and GPU share access to global memory, so memory access is simplified; there is no need to copy data from the host to the device. However, without the custom buffer feature, a memory copy would still be required to move data into a CUDA allocated buffer. With custom buffers, a block can allocate both its upstream and downstream buffers using the CUDA framework, from which the corresponding blocks would write and read. When a downstream CUDA block's *work()* function is called, the block could immediately launch a kernel without requiring extra memory management (Figure 2).

Dealing with discrete GPUs is more complex than an integrated embedded platform, but the custom buffer feature can also drastically simplify memory management in this use case. The CUDA framework can allocate page-locked, host memory (*cudaHostAlloc()*) that can be accessed from both the CPU and the GPU. Using this method, an explicitly copy from host to device is no longer needed, since memory is already accessible directly from the device. If a flowgraph contained multiple, consecutive CUDA blocks, the first block could read from page-locked memory and write to another custom buffer allocated on the device's global memory. Each successive block can read and write using device memory until the final block writes back to another page-locked buffer in host memory.

3.1. Runtime Changes

There were several required changes in the GNU Radio runtime to enable support custom buffers for blocks. A new flag MEM_BLOCK_ALLOC was added to the IO signature for blocks to inform the runtime they need to allocate a special buffer. If this flag is passed for either an input or output signature in the block's constructor, the runtime will call the corresponding *allocate_output_buffer()* or *allocate_upstream_output_buffer()* functions. An example of this is shown in Section A.2.

3.2. Single Mapped Buffers

One of the characteristics of the core runtime buffers is the doubly mapped address space that implements a circular buffer. This simplifies the runtime code managing the memory buffers and simplifies implementing features such as sample history. Implementing the same doubly mapped memory may not be possible for many use cases. For example, the CUDA framework and GPUs themselves do not support this feature, which makes implementing these custom buffers more complex.

Since a guarantee of the runtime is that any pointer (and items to read/write) passed to a *work()* function will always be valid, additional support must be added to the buffer to ensure the scheduler does not accidentally break this requirement.

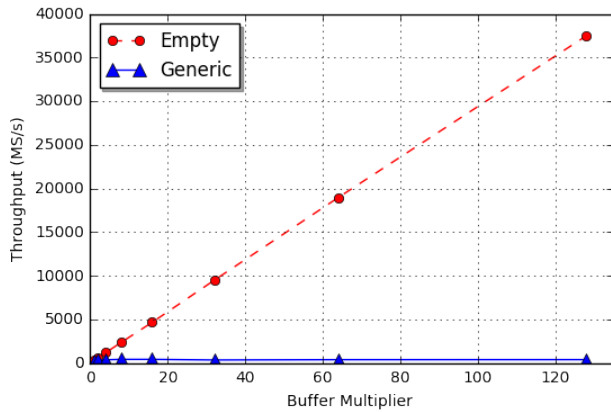

Figure 3. Average throughput of the empty flowgraph (Null Source → Null Sink) vs. the generic implementation (Null Source → Multiply_Const → Null Sink). The number of samples = 8192 * multiplier (Power of 2)

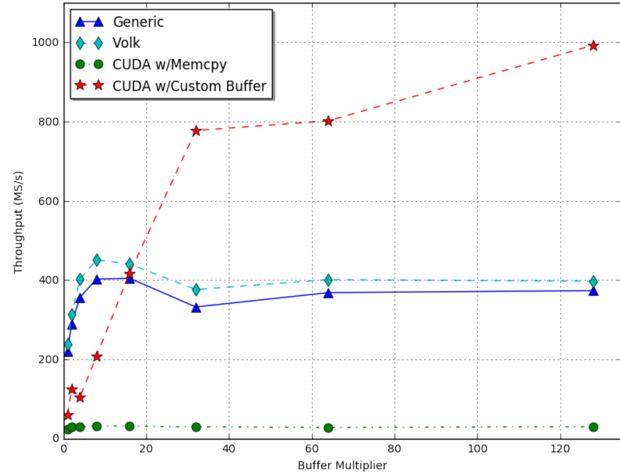

Figure 5. Average throughput comparison for different implementations of the test flowgraph.

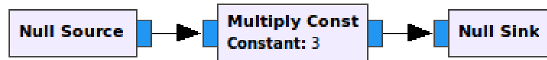

Figure 4. GRC example of the C++ test flowgraph

4. Performance

In order to evaluate the overall performance gains of GPU acceleration with custom buffers, several different implementations of the same basic flowgraph were compared by their overall throughput. The maximum throughput is the main metric being considered simply to determine how much an application can benefit from GPU acceleration. All tests were performed on NVIDIA's Tegra X1 embedded processor (set to maximum performance) that combines a quad-core ARM processor with a 256 core (2 SMs) Maxwell generation GPU (NVIDIA, Jan. 2015).

The test flowgraph is written in C++ and is very simple, consisting of a 'Null Source', a 'Multiply Const', and a 'Null Sink' (a GRC representation is shown in Figure 4). The reasoning for using such a basic flowgraph is removing all other possible sources of overhead in the system other than the GNU Radio runtime itself in order to get an accurate comparison of the implementation being benchmarked. This is not characteristic of a real world flowgraph and is only designed to demonstrate the benefits of the custom buffer feature. Factors such as flowgraph back-pressure and hardware clock rates are not being considered for this paper, as the goal of these tests is executing as quickly as possible to compare the maximum possible throughput of each implementation.

Five different implementations of the test flowgraph were compared including: a generic implementation, a VOLK kernel, a CUDA block with a copy between buffers, the same block with the CUDA kernel disabled, and a CUDA block using the custom buffers. The generic implementation consists of a loop that iterates over the incoming samples and executes a simple multiply; the VOLK implementation ('multiply_const_cc') uses the 'volk_32fc_s32fc_multiply_32fc' kernel that supports the Tegra X1's NEON extensions (West et al., 2015). The two different CUDA implementations were compared to show the overhead caused by constant memory copies between the default GNU Radio buffers and a CUDA accessible buffer. Additionally, the same memory copy CUDA implementation was tested without actually executing the kernel to demonstrate the overhead due to the memory management alone. Each test flowgraph was benchmarked using several different buffer sizes to analyze the impact of increasing the work per iteration for the CUDA kernels (Figure 5). The starting buffer size of each test was 8,192 samples (default) and the size was doubled for each successive test, up to a maximum size of 1,048,576 samples (128x increase).

Also, an empty version (null source to null sink) of the flowgraph was benchmarked to evaluate the maximum throughput of the system based only on the runtime overhead. Figure 3 and Table 2 show the throughput of the empty flowgraph based on the allocated buffer size. These results show that the overall throughput doubled as the buffer doubled in size for each successive test. This makes

Table 2. Average throughput comparison of the different test implementations (MS/s)

Buffer	Empty	Generic	Volk	Memcpy	CUDA	
					Memcpy	Buffer
8192	294.47	219.24	237.94	34.57	23.75	59.92
16384	584.25	289.51	312.54	36.14	29.17	125.00
32768	1170.24	356.13	402.25	36.98	28.99	105.01
65536	2348.78	402.11	451.04	37.37	31.86	207.76
131072	4648.74	404.33	440.47	34.67	30.86	417.23
262144	9467.53	332.01	376.00	30.59	29.35	777.45
524288	18945.50	368.08	400.80	31.41	27.84	801.83
1048576	37476.80	373.15	397.04	31.99	29.76	992.77

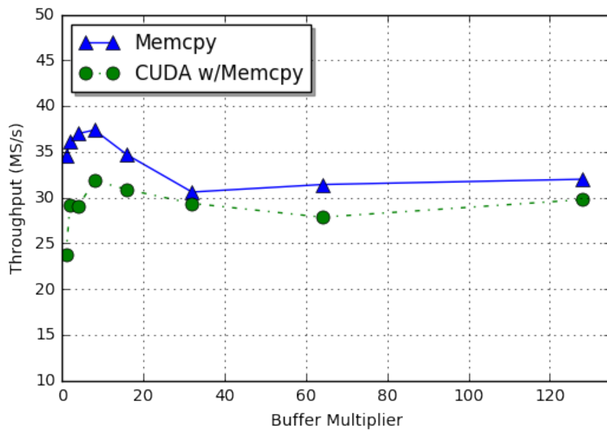

Figure 6. Comparison of the CUDA block with *memcpy()* operations to just the *memcpy()* overhead.

Table 3. Percent speedup over generic implementation

Buffer Size	VOLK	CUDA	
		Memcpy	Buffer
8192	9%	-92%	-73%
16384	8%	-90%	-57%
32768	13%	-92%	-71%
65536	12%	-92%	-48%
131072	9%	-92%	3%
262144	13%	-91%	134%
524288	9%	-92%	118%
1048576	6%	-92%	166%

sense considering that the number of samples produced for each call to *work()* should be doubling from the previous test.

Figure 5 and Table 2 show the comparison of the average throughputs of the various implementations discussed above. As expected, the VOLK implementation performed better than the generic implementation for all buffer sizes; it averaged about 10% better throughput than the generic test. Interestingly, both implementations have a maximum throughput at the 8x buffer size.

The performance of the two CUDA implementations was very different; the maximum throughput of the first (with the memory copy) never exceeded 32 MS/s, which is a 90% drop in performance in comparison to the generic version. Most of this overhead is due to the memory transfers from GNU Radio buffers to the CUDA managed buffer and back. Figure 6 shows the results of executing the same flowgraph with the CUDA kernel both enabled (green) and disabled (blue). When the kernel is disabled, the maximum throughput measured was about 37 MS/s; this shows that the dual memory copies per *work()* are the cause of the significant loss of performance.

The CUDA implementation using the custom buffer feature has vastly better performance. Initially, its performance is terrible compared to both the generic and VOLK implementations (73% drop in throughput); this is expected because of the large number of kernel launches occurring due to smaller buffer sizes, as previously discussed. A significant amount of execution is being spent launching the kernels, which is reducing the execution time available to actually process the incoming samples. However, as the buffer sizes and work per kernel increase, the performance of the custom buffer implementation quickly catches (16x buffer) and surpasses (32x) the other implementations. The performance of the custom buffer implementation continues to increase as buffer sizes increase and has a throughput of approximately 1000 MS/s (166% increase) at the maximum buffer size tested. Additional tests suggest the perfor-

mance continues to increase beyond the 128x buffer size. Of course, there is a trade-off with the overall latency of the system; as the buffer size increases, the overall latency is also increases. Whether this additional latency is acceptable depends entirely on the specific application.

5. Conclusion

The parallel architecture of a GPU system makes it an interesting candidate for accelerating software radio applications. However, there are challenges to integrating GPUs into a DSP pipeline: mainly this includes overhead from launching kernels and managing memory buffers. If not correctly accounted for, these issues can result in significant overhead that degrades the overall performance of the system.

The custom buffer feature reduces this complexity and allows for blocks to create specialized buffers that are used by the GNU Radio runtime without changing the underlying flowgraph model. A GPU implementation of the 'Multiply Const' using custom buffers showed varying performance characteristics in comparison to the same generic and VOLK implementations based on the block's buffer size. For small buffer sizes, the large number of kernel executions caused a 73% drop in throughput; however, as the buffer sizes increased the CUDA implementation was able to match and exceed the throughput performance of the generic implementation by 150%.

Acknowledgements

The authors would like to recognize Doug Geiger and Johnathan Corgan for their initial work with the custom buffer feature.

Source Code

As of publication, the custom buffer feature is still under development in the GNU Radio runtime. Examples and source code will be released once the feature is fully integrated into the GNU Radio code base.

References

NVIDIA. NVIDIA Titan X Specs, 2016. URL http://www.geforce.com/hardware/10series/titan-x-pascal#specs.

NVIDIA. Tegra X1 - NVIDIA's New Mobile Superchip. Technical report, Jan. 2015. URL http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf.

NVIDIA. CUDA C Programming Guide. Technical report, Sept. 2015. URL http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

Plishker, W., Zaki, G. F., Bhattacharyya, S. S., Clancy, C., and Kuykendall, J. Applying graphics processor acceleration in a software defined radio prototyping environment. In *2011 22nd IEEE International Symposium on Rapid System Prototyping*, pp. 67–73, May 2011. doi: 10.1109/RSP.2011.5929977.

Rondeau, T., McCarthy, N., and O'Shea, T. SIMD Programming in GNU Radio: Maintainable and User-Friendly Algorithm Optimization with VOLK.

VOLK. Vector-Optimized Library of Kernels, 2016. URL https://libvolk.org.

West, N., Geiger, D., and Scheets, G. Accelerating software radio on arm: Adding neon support to volk. In *2015 IEEE Radio and Wireless Symposium (RWS)*, pp. 174–176, Jan 2015. doi: 10.1109/RWS.2015.7129727.

A. Appendix

A.1. An Example CUDA Kernel

```
\* CUDA multiply constant kernel *\
__global__ void multiply_const(cuFloatComplex *input, cuFloatComplex *output,
                               float value, int samples) {
  // Get the thread index
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  // Make sure the current thread has work
  if (idx < samples) {
    output[idx].x = input[idx].x * val;
    output[idx].y = input[idx].y * val;
  }
}

// Wrapper for launching the CUDA kernel.
void cuda_kexec(void *input, void *output, float val, int samples) {
  // Launch with 8 blocks of 1024 threads
  multiply_const<<<8, 1024>>>((cuFloatComplex*)input, (cuFloatComplex*)output,
                              value, samples);
  cudaDeviceSynchronize();
}
```

A.2. An Example Block Using Custom Buffers

```
// This is an example of a block using the custom buffer feature
multiply_const_impl::multiply_const_impl(float p_value, long p_samples)
  : gr::sync_block("multiply_const",
    gr::io_signature::make(1, 1, sizeof(gr_complex),
                           gr::io_signature::MEM_BLOCK_ALLOC), // Upstream flag
    gr::io_signature::make(1, 1, sizeof(gr_complex),
                           gr::io_signature::MEM_BLOCK_ALLOC)) // Output flag
{
  samples = p_samples;
  value = p_value;
}

// Allocates an output buffer for this block (called by runtime)
buffer_sptr multiply_const_impl::allocate_output_buffer(int port)
{
  int item_size = this->output_signature()->sizeof_stream_item(port);
  block_sptr block_ptr = cast_to_block_sptr(shared_from_this());

  // Create the cuda pinned buffer and return it to the runtime
  cuda_pinned_sptr buf = make_cuda_pinned(samples, item_size, block_ptr);
  return boost::dynamic_pointer_cast<gr::buffer>(buf);
}

// Allocates the output buffer of an upstream block (aka this block's input buffer)
buffer_sptr multiply_const_impl::allocate_upstream_output_buffer(int port)
{
  int item_size = this->input_signature()->sizeof_stream_item(port);
  block_sptr block_ptr = cast_to_block_sptr(shared_from_this());
  cuda_pinned_sptr buf = make_cuda_pinned(samples, item_size, block_ptr);
  return boost::dynamic_pointer_cast<gr::buffer>(buf);
}
```