Study on Implementing OpenCL in Common GNURadio Blocks

PRESENTED BY:

GHOSTOP14
(HTTPS://GITHUB.COM/GHOSTOP14)

April, 2017
[Last Updated: May 2017]

Contents

Introduction

Software Defined Radio (SDR) has opened up a world of research and development into broad radio frequency (RF) communications and brought affordable hardware and open source software to the world. However as other real-world radio solutions continue to push for greater bandwidth such as satellite communications meeting or exceeding 100 Mbps and new wireless LAN technologies with 80 MHz-wide channels, these solutions implemented in hardware are a challenge to implement with the same throughput in SDR for real-time processing.

One option to provide additional throughput capabilities to SDR is to leverage those graphics cards that are present in many computers along with their highly capable, massively parallel GPU's to supplement CPU-based calculations. This project (named "gr-clenabled") is not the first project to evaluate this option. However, the unique goal of this project is to provide a more comprehensive and practical implementation of as many common blocks as possible along with providing quantitative timing analysis to study the benefit (or degradation) in performance while running on GPU's.

Gr-clenabled had a number of lofty goals at the project's onset. The goal was to go through as many GNURadio blocks as possible that are used in common digital communications processing (ASK, FSK, and PSK), convert them to OpenCL, and provide scalability by allowing each OpenCL-enabled block to be assigned to a user-selectable OpenCL device. This latter scalability feature would allow a system that has 3 graphics cards, or even a combination of GPU's and FPGA's, to have different blocks assigned to run on different cards all within the same flowgraph. This flexibility would also allow lower-end cards to drive less computational blocks and allow FPGA's to handle the more intensive blocks.

Simply implementing blocks in OpenCL does not guarantee that those blocks would perform better than CPU-based blocks. So this project also had to include a quantitative comparison of these new OpenCL-enabled blocks against the native GNURadio blocks. The results of each block could then be categorized into one of three groups: 1.) Those blocks that run faster in OpenCL than the native implementation [designated "accelerated"], 2.) those blocks where performance of the OpenCL blocks is very close to the native blocks or mixed across hardware and buffer sizes [designated "offloaded"], and 3.) those blocks whose OpenCL implementation performs worse than the native blocks [designated OpenCL "enabled", meaning they have been implemented].

There were a number of driving factors discussed in this paper that contributed to better or worse performance. For instance, those blocks that leveraged more CPU-intensive functions, specifically trigonometric functions such as sine, cosine, and inverse tangent calls, along with log functions showed the most improvement. Those functions that do

single-cycle or very low computation-intensive tasks such as a basic multiply function had the opposite affect where the cost of moving data to the OpenCL device and copying it back outweighed the benefit of the OpenCL processing resulting in worse performance. Others such as filters suffered dramatically in performance in OpenCL because of the way the signal processing math is executed with blocks and rolling carry-forward "tails".

In total the following list of blocks are implemented in this project and available online as part of an open source release. The goal being to further research and progress in this area.

1. Basic Building Blocks
    a. Signal Source
    b. Multiply
    c. Add
    d. Subtract
    e. Multiply Constant
    f. Add Constant
    g. Filters (Both FFT and FIR)
        i. Low Pass
        ii. High Pass
        iii. Band Pass
        iv. Band Reject
        v. Root-Raised Cosine
        vi. General tap-based
2. Common Math or Complex Data Functions
    a. Complex Conjugate
    b. Multiply Conjugate
    c. Complex to Arg
    d. Complex to Mag Phase
    e. Mag Phase to Complex
    f. Log10
    g. SNR Helper (a custom block performing divide->log10->abs)
    h. Forward FFT
    i. Reverse FFT
3. Digital Signal Processing
    a. Complex to Mag (used for ASK/OOK)
    b. Quadrature Demod (used for FSK)
    c. Costas Loop

Two general-purpose blocks were also built that allow a kernel described in a text file to process 1-to-1 and 2-to-1 input to output samples. This provides a level of future scalability without necessarily building new classes and recompiling code.

Note that while PSK was an initial design goal, after beginning the project it became clear than blocks with sequential calculations such as PSK and MM Clock Recovery did not lend themselves to OpenCL implementation because processing of each data point was not atomic enough. The

study also considered looking at instrumentation, but as noted in this study, FFT's actually performed worse in OpenCL for the block sizes used in real-time processing. Because of this fact, instrumentation sinks such as a Frequency Sink would also perform worse than their CPU counterpart. This was evident when testing gr-fosphor versus the native QT Frequency Sink. Monitoring CPU usage, even on new NVIDIA GTX 1070 hardware on new i7 processors showed much higher CPU usage than the CPU-only version.

The remainder of this paper proceeds through the methodology used in testing, the tools the gr-clenabled project provides, and some OpenCL lessons learned. Then each module along with the quantitative testing results are presented in detail, discussing the code used in each, along with the results and observations. The study then goes on to discuss the testing results of using multiple OpenCL blocks simultaneously in the same flowgraph, the effect of buffer sizes, and why some of the blocks required for digital processing (PSK and MM Clock Recovery) were not implemented. The study then wraps with general conclusions and observations.

Analysis

OPENCL

Open Computing Language (OpenCL) is a framework for writing code (kernels) that can run across a number of computing environments. This includes CPU's, Graphics Processing Units (GPU's), and Field Programmable Gate Arrays (FPGA's). A common language with which to write the code makes code more portable across all OpenCL-supported environments.

GPU's are used extensively on modern graphics cards for math functions generally related to image processing. However, they are not limited to image-related tasks. GPU's provide simultaneous processing of small kernels in levels of parallelism not possible on more general-purpose CPU's. This also makes them ideal for certain types of other tasks such as signal processing.

However, in the context of real-time software-defined radio, there are some caveats. First, GPU-offloading (from here-on out generically referred to as OpenCL to cover all OpenCL-capable devices) works most efficiently when calculations can be broken down into stand-alone (atomic) calculations working on each data point. That is not to say there are not techniques to deal with interactions between calculations, just that the greatest performance gains are achieved with atomic operations.

Next, each set of calculations applied to a data point should have some level of computing complexity to them. Basic operations like add, subtract, and multiply do not consume the same computing cycles as a log10 or inverse tangent (atan) calculation. Therefore, these basic calculations such as multiply tend to perform better on modern CPU's than on GPU's.

The last caveat specifically related to SDR is the OpenCL generally outperforms CPU's when working on very large data sets all at once. This is fundamentally a problem for real-time SDR as default buffer sizes used in GNURadio do not generally reach this "large data set" level for some calculations. For example, the default GNURadio buffer size for blocks is 8192. However during run-time generally about half that (4096) data points are sent. This does vary as controlled by a complex scheduling engine and can be adjusted by block parameters, however these data sets are not at levels such as 128K or 1M of data points where the offloading would become very evident.

In terms of OpenCL implementation, several lessons were learned during development. While they may be obvious to experienced OpenCL developers, it is worth mentioning them here.

First, there is a time price to pay to copy data to the graphics card for processing, then to copy the data back. In fact during testing as discussed later, a baseline case is defined that copies the data to the card and copies data back but performs no processing (the kernel

simply returns). The time it takes for this kernel to execute becomes the absolute best performance that could be achieved with OpenCL offloading. If CPU-based blocks do not take at least this long to process, then the CPU implementation will have better performance. This baseline is then expanded to simply do out=in to incorporate memory actions into the baseline.

Programmatically, creating buffers during each call to a GNURadio block's work (or general_work) function is costly and inefficient, and initial programming tests demonstrated this. Therefore this project takes a more efficient approach. An appropriately sized buffer is created once at block startup then reused throughout the course of operations. In the gr-clenabled implementation, safety checks ensure that the data passed in doesn't exceed the allocated buffer, and if it does resizes the buffer. This can happen in blocks where parameters can be changed at runtime such as constant blocks and filters.

The next important foundational memory concept is that OpenCL supports 2 mechanisms to move data back from a card. One is simply to enqueue a buffer read call while the other maps the underlying memory store to host address space with calls to map/unmap. During testing, using map/unmap was much slower than simply reading the return buffer. Therefore using map/unmap is not recommended.

Another important point about OpenCL implementations is that graphics cards have different memory types. Some are faster than others. Local memory and constant memory are present in the card, however smaller in size than global memory. For instance using the "clview" tool included with gr-clenabled shows constant and local memory sizes for an NVIDIA GTX 1070 card as 64K and 48K respectively.

```
Platform Name: NVIDIA CUDA
Device Name: GeForce GTX 1070
Device Type: GPU
Constant Memory: 64K (16384 floats)
Local Memory: 48K (12288 floats)
```

However with the size difference, local memory can be 15 times faster than global memory. Therefore every effort should be made to use this faster memory whenever possible. When looking at the numbers above in the context of GNURadio, 64K of constant memory. For complex data (2 floats) this translates to 8192 samples. While that seems to exactly match the GNURadio default buffer size, you have to remember that GNURadio's scheduler does not generally want to wait until the buffer is full, so will generally send about half that data. The first reaction may be to double the default buffer size in GNURadio, however this means that the scheduling engine MAY send more than 8192 so the OpenCL implementation has to be prepared for that and fall back to a global memory implementation. So there is a fine balance that must be struck.

One trick that can be used to further improve OpenCL performance is taking advantage of fixed values. If you are calling a kernel and passing it a parameter that will not change, there is still a cost to moving the argument to the kernel. One trick OpenCL coders use is to build the kernel with a #define rather than passing a fixed value as a parameter. Where appropriate some of the kernels here will use this approach.

Lastly, OpenCL devices have a concept of a preferred workgroup size multiple. An extensive discussion of this attribute will not be given here, however testing found that performance improved when calls to execute kernels used this preferred workgroup size. For instance on the NVIDIA GTX 1070 this value is 32. It was found during testing that calls to enqueueNDRangeKernel performed better when the workgroup size parameter was set to this value and the incoming data size was a multiple of this number. Interestingly, setting it to a higher multiple of this value (for example using 64 in this case) caused performance to drop slightly. Therefore in the absence of any other block requirements, gr-clenabled blocks set the output multiple to be this preferred size multiple and execute the kernel with this value for best performance.

THE CODE

The project containing all of the code for this study is in a GNURadio out-of-tree (OOT) module called gr-clenabled. This module can be found on GitHub at https://github.com/ghostop14/gr-clenabled.git. This study was meant to give back to the general SDR and open source community to provide a framework for moving towards ever-increasing digital processing speeds. The module does require clFFT be installed. Depending on your linux distro you may be able to 'sudo apt-get install libclfft-dev' to get all of the libraries and files to build the module. It can also be installed directly from source.

Also note that these modules were developed using GNURadio 3.7.10 (the latest at the time of development). Therefore if you run into any issues, first check that you have the latest GNURadio version and you have the latest Swig and Doxygen installed (see GNURadio's reference on building OOT modules for more details). Some people have issues building OOT modules in general when using older 3.7.9 on Ubuntu. The best approach is to install the latest GNURadio from pybombs.

You will also need an OpenCL implementation installed. The project does include some help on how to set this up on Ubuntu 16.04 and Debian/Kali for both NVIDIA cards and Intel drivers. Instructions are in the github repository in the setup_help subdirectory.

For CPU-based OpenCL, download the Intel software at https://software.intel.com/en-us/intel-opencl/download. intel_sdk_for_opencl_2016_ubuntu_6.3.0.1904_x64.tar.gz was used for CPU-based testing. While you may get an OS version issue, it should still install.

However running OpenCL on a CPU provides arguably a worse multithreading solution. The real benefit comes from running OpenCL on accelerated hardware. But this requires

the appropriate drivers and libraries. For instance on Linux running NVIDIA cards you may want to first 'sudo apt-get install nvidia-opencl-icd'. PLEASE READ THE DOCUMENTATION FOR GETTING OPENCL WORKING ON YOUR CARD AND VERSION OF LINUX BEFORE PROCEEDING! Obviously the OpenCL blocks won't compile or run until an OpenCL implementation is properly configured.

Once you have OpenCL set up, 'sudo apt-get install clinfo'. If you can run clinfo and see your card you are ready to proceed.

Now that you have OpenCL correctly set up, clfft installed, and a working GNURadio 3.7.10+ installation, make sure that gnuradio-dev is also installed if installing from a repo.

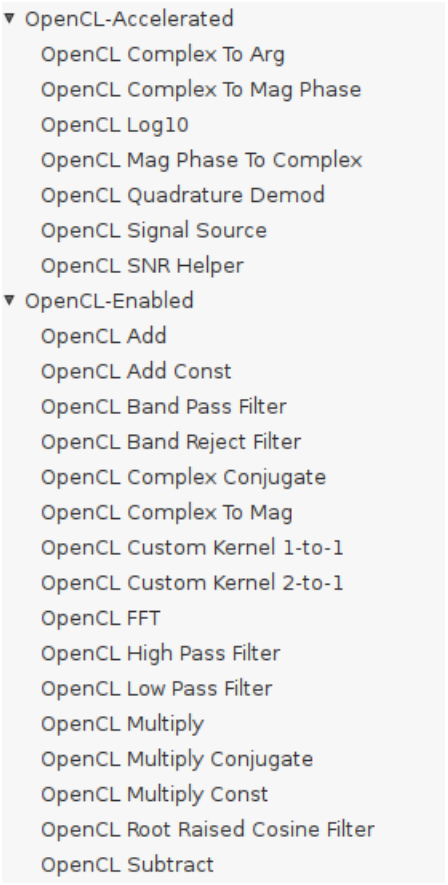
GNURadio Blocks

To build gr-clenabled, simply follow the standard module build process. Git clone it to a directory, close GNURadio if you have it open, then use the following build steps:

```
cd <clone directory>
mkdir build
cd build
cmake ..
make
sudo make install
sudo ldconfig
```

If each step was successful (don't overlook the 'sudo ldconfig' step).

Within GNURadio you will now have 2 new block groups as shown below:



Several command-line tools are also included in this project. These can be used to test performance on your specific system and were used to generate the data discussed in this document.

test-clenabled

after 'sudo make install' you can type 'test-clenabled --help' to get the help information below:

```
Usage: [--gpu] [--cpu] [--accel] [--any] [--device=<platformid>:<device
id>] [number of samples (default is 8192)]

where: --gpu, --cpu, --accel[erator], or any defines the type of OpenCL
device opened.

The optional --device argument allows for a specific OpenCL platform
and device to be chosen.  Use the included clview utility to get the
numbers.
```

The first few parameters allow you to choose from multiple GPU platforms, and if multiple cards are present define specifically what card you want to target. The easiest way to run it with a single GPU card is to simply type 'test-clenabled'. This will run with a default

8192 block size. This can be adjusted by running with a specific block size such as 'test-clenabled 4096'. In order to get the appropriate platform and device id, you can use clinfo, or the included clview tool which just provides a simpler view with the id numbers more easily identified.

clview

This tool just provides a simpler view than clinfo focused specifically on getting the correct platform and device id (highlighted in yellow below), and noting how much local and constant memory is present on your card. The output below shows the result for an NVIDIA GTX 1070 card:

```
Platform Id: 0
Device Id: 0
Platform Name: NVIDIA CUDA
Device Name: GeForce GTX 1070
Device Type: GPU
Constant Memory: 64K (16384 floats)
Local Memory: 48K (12288 floats)
OpenCL 2.0 Capabilities:
Shared Virtual Memory (SVM): Yes
Fine-grained SVM: No
```

The following output shows the result for the Intel CPU driver:

```
Platform Id: 0
Device Id: 0
Platform Name: Intel(R) OpenCL
Device Name: Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz
Device Type: CPU
Constant Memory: 128K (32768 floats)
Local Memory: 32K (8192 floats)
OpenCL 2.0 Capabilities:
Shared Virtual Memory (SVM): Yes
Fine-grained SVM: Yes
```

test-clfilter

Since filters are such a big part of signal processing and probably the first one may think of offloading, test-clfilter is a command-line tool to focus on filter performance. It tests filter performance with a given number of taps in 3 modes:

- OpenCL time-domain filter
- OpenCL frequency domain filter
- Native/CPU filter

For very small tap counts, you'll see that the time-domain filter will perform better. However as the number of taps increases, eventually the frequency domain version will

perform better. This tool gives you the opportunity to assess throughput on your hardware, and then make decisions for the "OpenCL Tap-Based Fir Filter" module which exposes the ability to select between time or frequency domain filtering.

The following shows some of the output for a small low pass filter with 241 taps on an NVIDIA 1070 card:

```
test-clfilter --ntaps=241
OpenCL Context: GPU
"Test Type                 "    throughput (sps)
"OpenCL Time Domain Filter"     130,682,400.00
"OpenCL Freq Domain Filter"     3,790,198.50
"CPU Freq Domain Filter"        191,092,784.00
```

The following shows with 1730 taps:

```
test-clfilter --ntaps=1730
OpenCL Context: GPU
"Test Type                  "    throughput (sps)
"OpenCL Time Domain Filter"     41,613,248.00
"OpenCL Freq Domain Filter"     13,019,512.00
"CPU Freq Domain Filter"        157,887,296.00
```

The following shows the help screen for the tool:

```
Usage: [--gpu] [--cpu] [--accel] [--any] [--device=<platformid>:<device id>] --
ntaps=<# of filter taps> [number of samples (default is 8192)]

where: --gpu, --cpu, --accel[erator], or any defines the type of OpenCL device
opened.

The optional --device argument allows for a specific OpenCL platform and device
to be chosen.  Use the included clview utility to get the numbers.



You can create a filter by hand and see how many taps it would create from an
interactive python command-line like this:



python

from gnuradio.filter import firdes

# parameters are gain, sample rate, cutoff freq, transition width for this
low_pass filter.

taps=firdes.low_pass(1, 10e6, 500e3, 0.2*500e3)

len(taps)
```

```
For this example 241 taps were created.
```

test-clkernel

Two blocks included in gr-clenabled are generic 1-to-1 and 2-to-1 kernel blocks. These blocks allow a designer to write their own kernel and save it to a file and select the appropriate data type (complex, float, etc.) and provide their own implementation to extend GNURadio. Test-clkernel provides the same level of timing testing and kernel testing from a command-line to make sure your kernel compiles and see how it performs.

'test-clkernel --help' will provide the parameters to provide as shown below:

```
Usage: <[--1to1] [--2to1]> <[--complex] [--float] [--int]> [--gpu] [--cpu] [--
accel] [--any] [--device=<platformid>:<device id>] [number of samples (default
is 8192)]
Where:
--1to1 says use the 1 input stream to 1 output stream module
--2to1 says use the 2 input streams to 1 output stream module
complex/float/int defines the data type of the streams (in matches out)
--fnname is the kernel function name to call in the provided kernel file (e.g.
what's on the __kernel line
--kernelfile is the file containing a valid OpenCL kernel matching the stream
format 2-in/1-out or 1-in/2-out
--gpu, --cpu, --accel[erator], or any defines the type of OpenCL device opened.
The optional --device argument allows for a specific OpenCL platform and device
to be chosen.  Use the included clview utility to get the numbers.
```

NOTESABOUT CUSTOM KERNELS:

If you use trig functions in your kernel, verify your hardware supports double precision math. You can do this with clview which will immediately tell you if your card supports it. Then in your kernel, you can still pass floats but make sure you typecast parameters to the trig functions as (double) first or you will notice too much variation in the calculated results.

The following shows a simple example kernel from the project's <project>/examples/ kernel1to1_sincos.cl file:

```
struct ComplexStruct {

      float real;

      float imag;

};



typedef struct ComplexStruct SComplex;
```

```
__kernel void fn_sin_cos(__global SComplex * restrict a, __global
SComplex * restrict c) {

    /* You have to be careful with trig functions and precision.

       If you call the float versions of sin/cos for example, it may
only be accurate to

       5-6 decimal places for CPU and 9-10 for GPU's which won't be
accurate enough

       for signal processing.  So make sure you use the double
versions.

    */


      size_t index =  get_global_id(0);

      c[index].real = cos((double)a[index].real);

      c[index].imag = sin((double)a[index].imag);

}
```

Printing Actual Block Sizes

During testing, several options were considered to understand specifically how much data GNURadio's scheduler was actually sending to the block. How true was the "half max buffer size" theory? While developing other blocks that could output the size of the blocks was considered, it made an assumption that all blocks would get the same block size. In order to avoid an incorrect assumption, in the GRCLBase.cpp file, there is a variable called CLPRINT_NITEMS. If this value is set to true and the module recompiled, enabling debug on a block will cause the block to output the size of the input items buffer for each iteration. Note that this will inevitably have an impact on performance so it should only be done if true block sizes are desired. The line below shows what the line of code looks like:

```
bool CLPRINT_NITEMS=false;
```

METHODOLOGY

As stated earlier, test-clenabled was used to generate the data for this study. Incremental block sizes were used from 2048 to 24576 in 2048 sample increments. While data could be extended much higher, within the context of GNURadio, larger block sizes can impact the real-time processing of flowgraphs. And since the scheduler generally sends about half the set buffer size to each block, 24576 of tested data points would correspond to a real-

world setting in GNURadio of twice that. Since this study is focused on practical implementations, data collection was capped at the 24576 processed buffer stream sizes, however test-clenabled is capable of running with any value.

The code that provides the actual testing is in test_clenabled.cc. This code goes through each of the block types discussed in this report and does timing tests both with the straight GNURadio CPU-only code as well as the OpenCL implementation. In order to ensure the integrity of the analysis, the code from GNURadio was used for the CPU-only test. Therefore if the GNURadio implementation used Volk, the CPU-only comparison is also against Volk. The goal was to provide an honest comparison of native GNURadio blocks versus the OpenCL equivalents.

In order to get good sample data, each test first starts with a single call that is not used in calculating performance. This is to remove any initialization performance issues from the first call from the run-time calculations. Each block is then run through 100 iterations and timed with the std::chrono::steady_clock object to provide a 100-run average. The code below shows one of these representative timing tests:

```
start = std::chrono::steady_clock::now();
// make iterations calls to get average.
for (i=0;i<iterations;i++) {
        noutputitems = test->testOpenCL(largeBlockSize,ninitems,inputPointers,outputPointers);
}
end = std::chrono::steady_clock::now();

elapsed_seconds = end-start;

elapsed_time = elapsed_seconds.count()/(float)iterations;
throughput = largeBlockSize / elapsed_time;
```

The chrono library provides several different types of timing clocks, however steady_clock is the chrono version recommended for measuring time intervals.

In any study, the data can sometimes show outliers or anomalies. While collecting data, if data points showed extreme anomalies, that data point was rerun several times to get a stable number. The assumption being that when that anomalous sample occurred, that the computer may have been executing another task that interrupted the data run.

It should also be noted that each module is tested in isolation. Meaning that multiple modules are not run simultaneously. One important note on using OpenCL and GPU's is that the assumption is that a module gets the full benefit of the card. If multiple blocks are trying to run multiple kernels simultaneously, it follows that the card's overall performance too needs to be shared, and if multiple OpenCL contexts are used, the card will need to account for running multiple contexts. The performance of multiple blocks running simultaneously is discussed later in this study.

TEST PLATFORM

One goal of this project was to test across generally available computer configurations. This means not just a new computer, but also slightly older desktops, laptops, and even virtual machines. Testing with different graphics cards also increased the likelihood that if there were issues with older or mobile graphics platforms that those could be identified and addressed during development. The net result was that 4 platforms were tested. The table below shows each of those configurations. In the subsequent data sets, these platforms are designated by their graphics card version.

UPDATE: During testing, all systems used the same Debian/kali 4.9.0 kernel. After the study, some systems were switched to Ubuntu 16.04 LTS running the 4.4 kernel. On this 4.4 kernel, performance was slightly lower. Some research indicated that there are a number of performance improvements with some of the newer kernels. So this should be taken into account when selecting your own target platform. In other words even on the same hardware, performance may vary based on OS and kernel version.

Report Designator	VM	1000M	970	1070
Platform Description	Virtual Machine	Laptop	Older system	New System
OS	Debian/Kali Linux "Linux 4.9.0-kali3-amd64 #1 SMP Debian 4.9.18-1kali1"	Debian/Kali Linux "Linux 4.9.0-kali3-amd64 #1 SMP Debian 4.9.18-1kali1"	Debian/Kali Linux "Linux 4.9.0-kali3-amd64 #1 SMP Debian 4.9.18-1kali1"	Debian/Kali Linux "Linux 4.9.0-kali3-amd64 #1 SMP Debian 4.9.18-1kali1"
Hardware	Virtual Machine running on a Dell Precision M4700	Dell Precision M4600	Custom-build	Custom-build
CPU	Intel i7-3740QM @ 2.7 GHz 8 cores assigned to the VM	i7-2820QM CPU @ 2.30GHz	Intel i7 - 2700 @ 3.5 GHz	Intel i7 - 6700 @ 3.4 GHz
Memory	3 GB RAM assigned to VM	16 GB	8 GB	16 GB
OpenCL Platform	Intel CPU Driver	NVIDIA 1000M	NVIDIA GTX 970	NVIDIA GTX 1070

It should be noted that the tests were also initially tested on an NVIDIA GTX 730 card before that card was upgraded to the 1070 presented in this study. So a significant variety of NVIDIA hardware was tested.

BASELINE

No-action Kernel

In order to get a feel for the absolute best performance that could be achieved with OpenCL offloading, 2 initial tests were performed. Note that all tests outlined in this study were done against complex numbers with a few exceptions as noted in the appropriate section. However it should be noted that the actual blocks do support all data types such as complex, float, and int where appropriate.

A "no-action" kernel that simply returns as shown below:

```
struct ComplexStruct {
float real;
float imag;
};
typedef struct ComplexStruct SComplex;
__kernel void opconst_complex(__constant SComplex * a, const float multiplier,
__global SComplex * restrict c) {
      return;
}
```

Notice the ***__constant*** parameter specifier to use faster contant memory. Each block assesses the requested block size and the memory available on the card to determine if it has sufficient room to use constant memory. If not the kernel is automatically switched to use global memory as shown below:

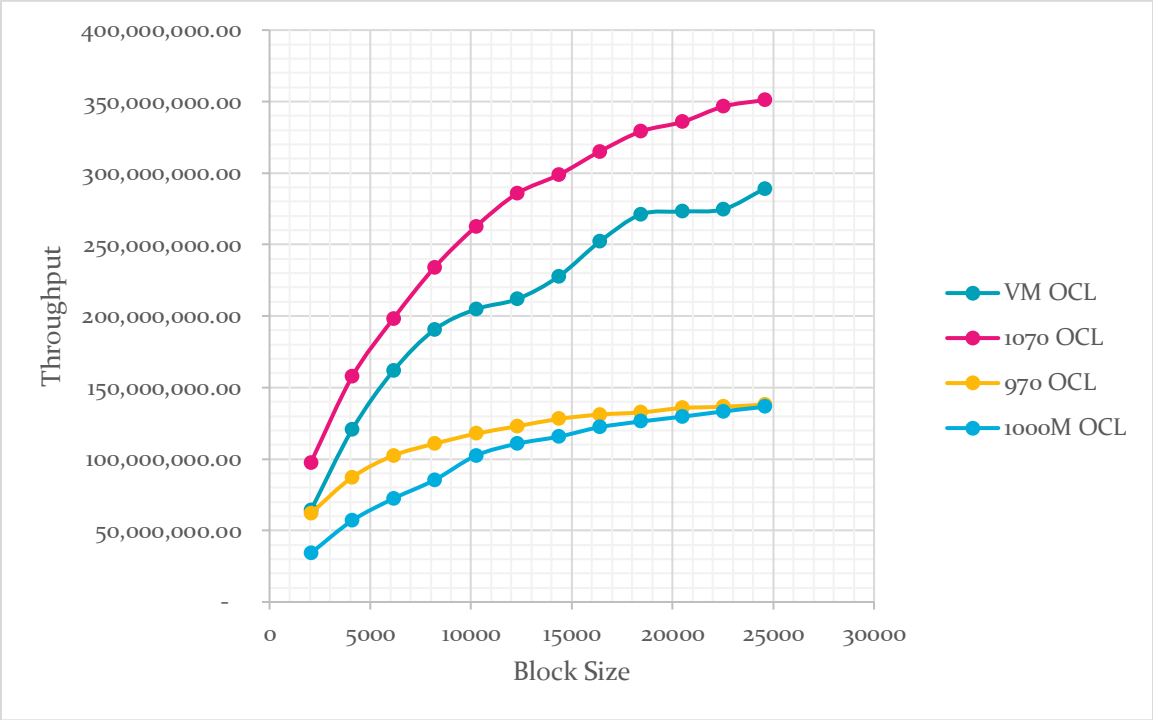
```
struct ComplexStruct {
float real;
float imag;
};
typedef struct ComplexStruct SComplex;
__kernel void opconst_complex(__global SComplex * restrict a, const float
multiplier, __global SComplex * restrict c) {
      return;
}
```

In each of the subsequent sections the results of each of the tests will be shown in 3 ways. The first is graphically. Then 2 tables will follow. The first table will contain the raw time-based measurements while the second table will show the data transformed based on the block size to a throughput number.

IMPORTANT: The throughput number should not be taken as the throughput that you will see from the block. It represents the absolute theoretical maximum possible from the block if data was continuously streamed to the function. For instance, the scheduler in GNURadio may wait for sufficient data to be available before sending it to the block and may adjust when blocks run. So continuous operation is unlikely.

However, since a continuous streaming approach was used across all OpenCL and CPU tests, the results provide benchmarks on absolute maximum throughputs for each block type as well as relative performance comparisons of OpenCL versus CPU implementations.

Data



In each of the following tables, rows highlighted in yellow are related to the default GNURadio configuration. 8192 is the default buffer size whereas 4096 would be the general expected actual bytes processed if no adjustments are made.

Timing in Seconds

Samples	VM OCL	1070 OCL	970 OCL	1000M OCL
2048	0.000032	0.000021	0.000033	0.000060
4096	0.000034	0.000026	0.000047	0.000072
6144	0.000038	0.000031	0.000060	0.000085
8192	0.000043	0.000035	0.000074	0.000096
10240	0.000050	0.000039	0.000087	0.000100
12288	0.000058	0.000043	0.000100	0.000111
14336	0.000063	0.000048	0.000112	0.000124
16384	0.000065	0.000052	0.000125	0.000134
18432	0.000068	0.000056	0.000139	0.000146
20480	0.000075	0.000061	0.000151	0.000158
22528	0.000082	0.000065	0.000165	0.000169
24576	0.000085	0.000070	0.000178	0.00018

Sample throughput based on time and block size

Samples	VM OCL	1070 OCL	970 OCL	1000M OCL
2048	64,000,000.00	97,523,809.52	62,060,606.06	34,133,333.33
4096	120,470,588.24	157,538,461.54	87,148,936.17	56,888,888.89
6144	161,684,210.53	198,193,548.39	102,400,000.00	72,282,352.94
8192	190,511,627.91	234,057,142.86	110,702,702.70	85,333,333.33
10240	204,800,000.00	262,564,102.56	117,701,149.43	102,400,000.00
12288	211,862,068.97	285,767,441.86	122,880,000.00	110,702,702.70
14336	227,555,555.56	298,666,666.67	128,000,000.00	115,612,903.23
16384	252,061,538.46	315,076,923.08	131,072,000.00	122,268,656.72
18432	271,058,823.53	329,142,857.14	132,604,316.55	126,246,575.34
20480	273,066,666.67	335,737,704.92	135,629,139.07	129,620,253.16
22528	274,731,707.32	346,584,615.38	136,533,333.33	133,301,775.15
24576	289,129,411.76	351,085,714.29	138,067,415.73	136,533,333.33

Observations

The good news from this baseline run is that the results agree with the expected outcome and knowledge of OpenCL. From the data above it becomes obvious that larger block sizes show better throughput. It also demonstrates the timing price to move data to and from a card. The 1070 for instance showed 35 microseconds for moving 8192 samples to the card and just returning.

It also shows the trend that newer cards have improved performance over older and mobile cards. The 1070 moved data at about twice the rate as the 970 and about 3 times

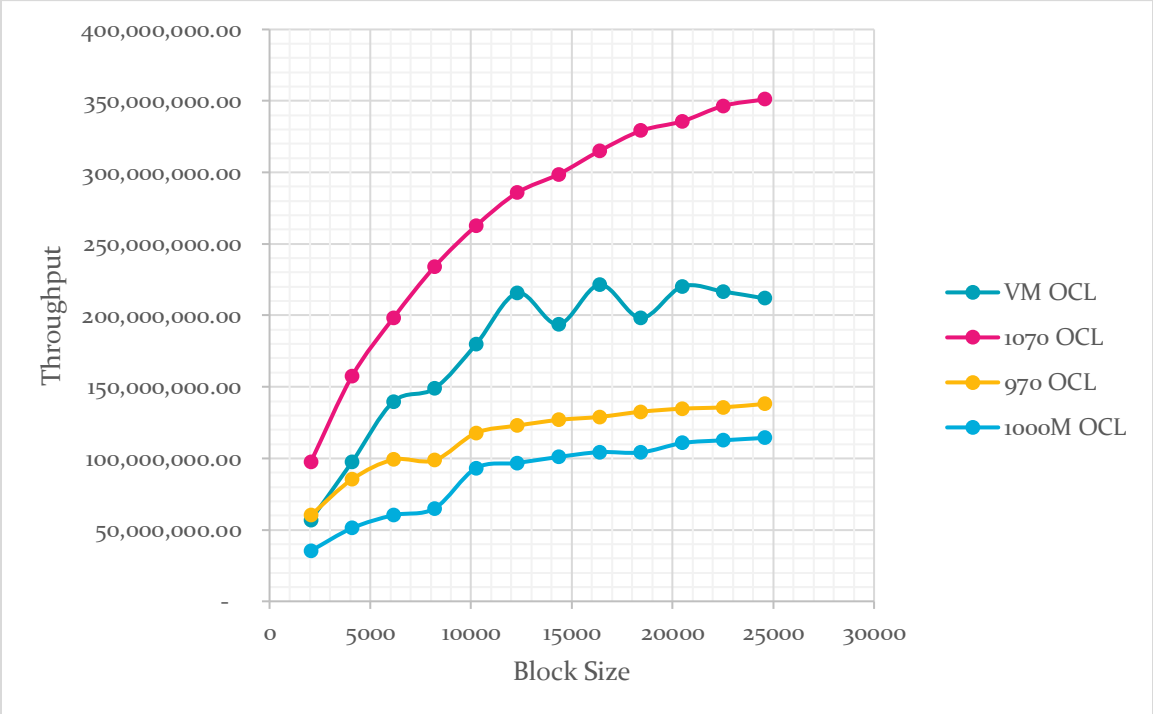
faster than the mobile chip. It also showed a trend evident in many of the tests with the VM and CPU driver. That is that the results show a good bit of variation. This could be due to a combination of both the virtualization of the processor as well as the fact that OpenCL is running on a general-purpose CPU.

Copy Kernel

The copy kernel was similar to the no-action kernel and was used as a baseline. This kernel simply assigned output=input to represent a data copy within the kernel then a return. The same constant/global separation was done automatically based on the memory available on the card. The following kernel shows the constant version:

```
struct ComplexStruct {
float real;
float imag;
};
typedef struct ComplexStruct SComplex;
__kernel void opconst_complex(__constant SComplex * a, const float multiplier,
__global SComplex * restrict c) {
       c.real = a.real;
       c.imag = a.imag;
}
```

Data



Timing in Seconds

Samples	VM OCL	1070 OCL	970 OCL	1000M OCL
2048	0.000036	0.000021	0.000034	0.000058
4096	0.000042	0.000026	0.000048	0.000080
6144	0.000044	0.000031	0.000062	0.000102
8192	0.000055	0.000035	0.000083	0.000126
10240	0.000057	0.000039	0.000087	0.000110
12288	0.000057	0.000043	0.000100	0.000127
14336	0.000074	0.000048	0.000113	0.000142
16384	0.000074	0.000052	0.000127	0.000157
18432	0.000093	0.000056	0.000139	0.000177
20480	0.000093	0.000061	0.000152	0.000185
22528	0.000104	0.000065	0.000166	0.000200
24576	0.000116	0.000070	0.000178	0.000215

Sample throughput based on time and block size

Samples	VM OCL	1070 OCL	970 OCL	1000M OCL
2048	56,888,888.89	97,523,809.52	60,235,294.12	35,310,344.83
4096	97,523,809.52	157,538,461.54	85,333,333.33	51,200,000.00
6144	139,636,363.64	198,193,548.39	99,096,774.19	60,235,294.12
8192	148,945,454.55	234,057,142.86	98,698,795.18	65,015,873.02
10240	179,649,122.81	262,564,102.56	117,701,149.43	93,090,909.09
12288	215,578,947.37	285,767,441.86	122,880,000.00	96,755,905.51
14336	193,729,729.73	298,666,666.67	126,867,256.64	100,957,746.48
16384	221,405,405.41	315,076,923.08	129,007,874.02	104,356,687.90
18432	198,193,548.39	329,142,857.14	132,604,316.55	104,135,593.22
20480	220,215,053.76	335,737,704.92	134,736,842.11	110,702,702.70
22528	216,615,384.62	346,584,615.38	135,710,843.37	112,640,000.00
24576	211,862,068.97	351,085,714.29	138,067,415.73	114,306,976.74

Observations

These results simply continued the trend observed in the no-action kernel. Note the lower throughput on the older and mobile platforms along with the variation in the OpenCL CPU version. Again clearly the new 1070 card significantly outperforms the older hardware and shows very consistent performance.

It was also interesting to note the impact of constant versus global memory in the curves across the different hardware platforms. What would have been expected would be a change in the performance curves representing a slight decrease in performance going to

global memory from constant memory above 8192 data points. However visually smoothing the curves shows that the performance curve seems somewhat unaffected. This is not to say that using constant memory does not provide any benefit, just that as the block sizes increase this benefit appears to be offset by overall processing of larger blocks.

ACCELERATED LIST

In any signal processing system, throughput is going to be limited in part by the slowest component. Where multiply blocks may be capable of exceeding 1,000 Msps, as soon as you enable a more complex processing block such as a log10 block the throughput may immediately drop to 40 Msps. Therefore when considering the overall throughput capacity of a given flowgraph, increasing the performance of the slowest blocks can mean the difference between maintaining high throughput or not being able to process the data in real-time. This section discusses the blocks that after analysis demonstrated significant throughput improvements in OpenCL implementations.

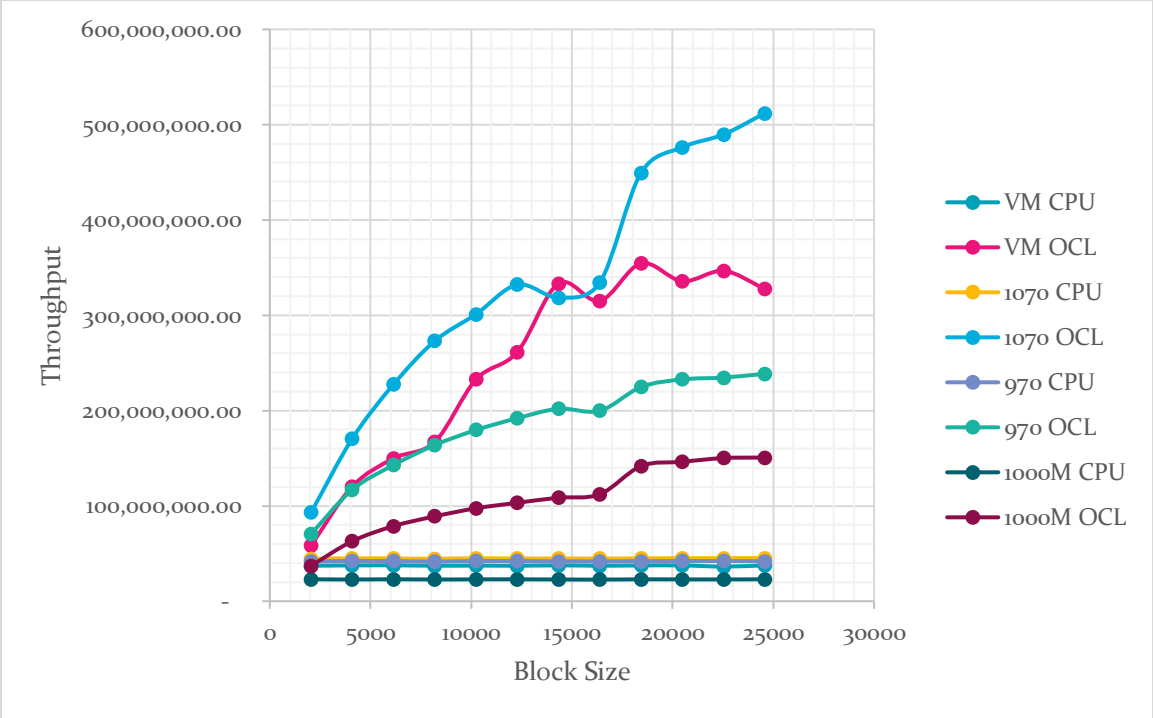
Log10

Log10 functions have all the hallmark of being good candidates for OpenCL acceleration. The calculations are atomic and require more CPU cycles than basic add/multiply/subtract operations. The Log10 block is one of the exceptions to using complex data points. This OpenCL block is designed to only work with float data. The block builds a kernel string with a few performance options. For instance, if n_val is passed as 1, don't even bother with the math. And since these values are not expected to change for this block, let's #define them rather than passing them as a parameter. The following code shows the kernel string being built.

```
	srcStdStr = "";
	if (n_val != 1.0) {
		srcStdStr += "#define n_val " + std::to_string(n_val) + "\n";
	}

	if (n_val != 1.0) {
		srcStdStr += "#define n_val " + std::to_string(n_val) + "\n";
	}

	if (k_val != 0.0) {
		srcStdStr += "#define k_val " + std::to_string(k_val) + "\n";
	}

	if (useConst)
		srcStdStr += "__kernel void op_log10(__constant float * a,
__global float * restrict c) {\n";
	else
		srcStdStr += "__kernel void op_log10(__global float * restrict a,
__global float * restrict c) {\n";

	srcStdStr += "    size_t index =  get_global_id(0);\n";

	if (k_val != 0.0) {
		if (n_val != 1.0) {
```

```
            srcStdStr += "    c[index] = n_val * log10(a[index]) + k_val;\n";
            }
            else {
            srcStdStr += "    c[index] = log10(a[index]) + k_val;\n";
            }
        }
        else {
            // Don't even bother with the k math op.
            if (n_val != 1.0) {
            srcStdStr += "    c[index] = n_val * log10(a[index]);\n";
            }
            else {
            srcStdStr += "    c[index] = log10(a[index]);\n";
            }
        }

        srcStdStr += "}\n";
```

Data



Timing in Seconds

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	0.000055	0.000035	0.000046	0.000022	0.000049	0.000029	0.000089	0.000055
4096	0.000109	0.000034	0.000091	0.000024	0.000097	0.000035	0.000180	0.000065
6144	0.000163	0.000041	0.000137	0.000027	0.000146	0.000043	0.000267	0.000078
8192	0.000220	0.000049	0.000184	0.000030	0.000197	0.000050	0.00036	0.000092
10240	0.000274	0.000044	0.000227	0.000034	0.000243	0.000057	0.000447	0.000105
12288	0.000330	0.000047	0.000274	0.000037	0.000292	0.000064	0.000535	0.000119
14336	0.000381	0.000043	0.000320	0.000045	0.000343	0.000071	0.000629	0.000132
16384	0.000440	0.000052	0.000366	0.000049	0.000392	0.000082	0.000722	0.000146
18432	0.000491	0.000052	0.000410	0.000041	0.000439	0.000082	0.000803	0.00013
20480	0.000545	0.000061	0.000454	0.000043	0.000486	0.000088	0.000895	0.000140
22528	0.00062	0.000065	0.000499	0.000046	0.000535	0.000096	0.000986	0.000150
24576	0.000653	0.000075	0.000542	0.000048	0.000586	0.000103	0.001069	0.000163

Sample throughput based on time and block size

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	37,236,363.64	58,514,285.71	44,521,739.13	93,090,909.09	41,795,918.37	70,620,689.66	23,011,235.96	37,236,363.64
4096	37,577,981.65	120,470,588.24	45,010,989.01	170,666,666.67	42,226,804.12	117,028,571.43	22,755,555.56	63,015,384.62
6144	37,693,251.53	149,853,658.54	44,846,715.33	227,555,555.56	42,082,191.78	142,883,720.93	23,011,235.96	78,769,230.77
8192	37,236,363.64	167,183,673.47	44,521,739.13	273,066,666.67	41,583,756.35	163,840,000.00	22,755,555.56	89,043,478.26
10240	37,372,262.77	232,727,272.73	45,110,132.16	301,176,470.59	42,139,917.70	179,649,122.81	22,908,277.40	97,523,809.52
12288	37,236,363.64	261,446,808.51	44,846,715.33	332,108,108.11	42,082,191.78	192,000,000.00	22,968,224.30	103,260,504.20
14336	37,627,296.59	333,395,348.84	44,800,000.00	318,577,777.78	41,795,918.37	201,915,492.96	22,791,732.91	108,606,060.61
16384	37,236,363.64	315,076,923.08	44,765,027.32	334,367,346.94	41,795,918.37	199,804,878.05	22,692,520.78	112,219,178.08
18432	37,539,714.87	354,461,538.46	44,956,097.56	449,560,975.61	41,986,332.57	224,780,487.80	22,953,922.79	141,784,615.38
20480	37,577,981.65	335,737,704.92	45,110,132.16	476,279,069.77	42,139,917.70	232,727,272.73	22,882,681.56	146,285,714.29
22528	36,335,483.87	346,584,615.38	45,146,292.59	489,739,130.43	42,108,411.21	234,666,666.67	22,847,870.18	150,186,666.67
24576	37,635,528.33	327,680,000.00	45,343,173.43	512,000,000.00	41,938,566.55	238,601,941.75	22,989,710.01	150,773,006.13

Observations

The results of this block clearly show relatively flat CPU throughput. Meaning that as the sample sizes double, so does the processing time such that the overall throughput is constant across block sizes.

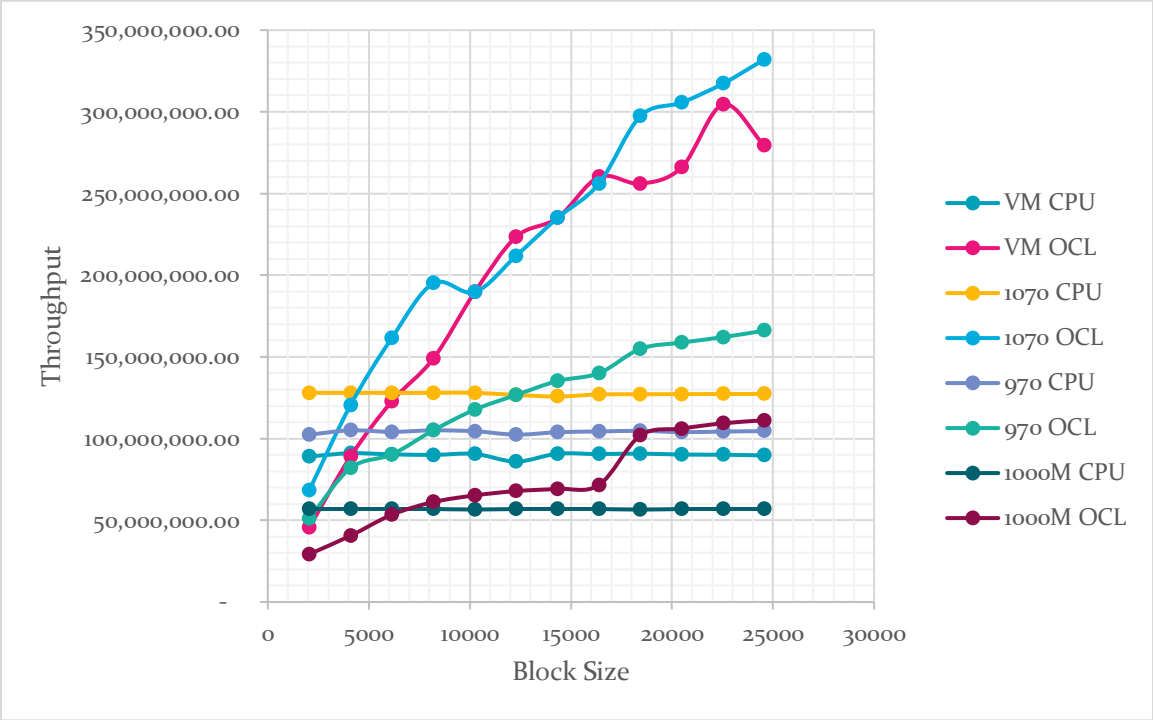
In contrast the OpenCL implementation clearly shows a performance improvement over the CPU implementation even at small block sizes. This performance improvement continues to increase as the block sizes increase. Where the CPU throughput for the 1070 platform stays constant around 44 Msps, the OpenCL implementation can easily exceed 150 Msps. In the case of the Log10 block, the OpenCL version provides a significant advantage over the CPU-only block.

SNR Helper

When discussing OpenCL, one way to gain more benefits from kernels is to put more operations in a single call. The SNR helper block is an example of such a block. This sequence of blocks: divide->log10->Abs could be used for a basic signal to noise ratio calculator. This block shows how a single kernel could combine these multiple functions into a single call and demonstrates the benefit gained from such an approach. The following code builds the kernel string:

```
srcStdStr += "#define n_val " + std::to_string(n_val) + "\n";
srcStdStr += "#define k_val " + std::to_string(k_val) + "\n";

if (useConst)
       srcStdStr += "__kernel void op_snr(__constant float * a, __constant float
* b, __global float * restrict c) {\n";
else
       srcStdStr += "__kernel void op_snr(__global float * restrict a, __global
float * restrict b, __global float * restrict c) {\n";

srcStdStr += "    size_t index =  get_global_id(0);\n";
srcStdStr += "    float tmpVal = a[index] / b[index];\n";
srcStdStr += "    tmpVal = n_val * log10(tmpVal) + k_val;\n";
srcStdStr += "    c[index] = fabs(tmpVal);\n";
srcStdStr += "}\n";
```

Data



Timing in Seconds

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	0.000023	0.000045	0.000016	0.000030	0.000020	0.000040	0.000036	0.000070
4096	0.000045	0.000046	0.000032	0.000034	0.000039	0.000050	0.000072	0.000101
6144	0.000068	0.00005	0.000048	0.000038	0.000059	0.000068	0.000108	0.000115
8192	0.000091	0.000055	0.000064	0.000042	0.000078	0.000078	0.000144	0.000134
10240	0.000113	0.000054	0.000080	0.000054	0.000098	0.000087	0.000181	0.000157
12288	0.000143	0.000055	0.000097	0.000058	0.000120	0.000097	0.000216	0.000181
14336	0.000158	0.000061	0.000114	0.000061	0.000138	0.000106	0.000252	0.000207
16384	0.000181	0.000063	0.000129	0.000064	0.000157	0.000117	0.000288	0.000229
18432	0.000203	0.000072	0.000145	0.000062	0.000176	0.000119	0.000326	0.000181
20480	0.000227	0.000077	0.000161	0.000067	0.000197	0.000129	0.000360	0.000193
22528	0.00025	0.000074	0.000177	0.000071	0.000216	0.000139	0.000396	0.000206
24576	0.000274	0.000088	0.000193	0.000074	0.000235	0.000148	0.000432	0.000221

Sample throughput based on time and block size

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	89,043,478.26	45,511,111.11	128,000,000.00	68,266,666.67	102,400,000.00	51,200,000.00	56,888,888.89	29,257,142.86
4096	91,022,222.22	89,043,478.26	128,000,000.00	120,470,588.24	105,025,641.03	81,920,000.00	56,888,888.89	40,554,455.45
6144	90,352,941.18	122,880,000.00	128,000,000.00	161,684,210.53	104,135,593.22	90,352,941.18	56,888,888.89	53,426,086.96
8192	90,021,978.02	148,945,454.55	128,000,000.00	195,047,619.05	105,025,641.03	105,025,641.03	56,888,888.89	61,134,328.36
10240	90,619,469.03	189,629,629.63	128,000,000.00	189,629,629.63	104,489,795.92	117,701,149.43	56,574,585.64	65,222,929.94
12288	85,930,069.93	223,418,181.82	126,680,412.37	211,862,068.97	102,400,000.00	126,680,412.37	56,888,888.89	67,889,502.76
14336	90,734,177.22	235,016,393.44	125,754,385.96	235,016,393.44	103,884,057.97	135,245,283.02	56,888,888.89	69,256,038.65
16384	90,519,337.02	260,063,492.06	127,007,751.94	256,000,000.00	104,356,687.90	140,034,188.03	56,888,888.89	71,545,851.53
18432	90,798,029.56	256,000,000.00	127,117,241.38	297,290,322.58	104,727,272.73	154,890,756.30	56,539,877.30	101,834,254.14
20480	90,220,264.32	265,974,025.97	127,204,968.94	305,671,641.79	103,959,390.86	158,759,689.92	56,888,888.89	106,113,989.64
22528	90,112,000.00	304,432,432.43	127,276,836.16	317,295,774.65	104,296,296.30	162,071,942.45	56,888,888.89	109,359,223.30
24576	89,693,430.66	279,272,727.27	127,336,787.56	332,108,108.11	104,578,723.40	166,054,054.05	56,888,888.89	111,203,619.91

Observations

Because this block builds upon the Log10 calculation by adding a divide and absolute value operation in a single kernel, the overall result of this block having a performance increase was as expected in that the OpenCL version showed significant improvement over the CPU-only version.

However, there was an anomaly in this block that was continuously reproduced in testing. That is that this block includes not only the log10 calculation, but also performs a divide and absolute value. However the CPU throughput was actually better than the log10-only block. The reason behind this was never identified, however the result was consistently reproduced.

Complex to Arg

The Complex To Arg block is really the phase calculation from the Complex To Mag Phase block isolated to only output the phase. The block takes the inverse tangent of the input complex number and outputs a float as calculated below:

```
c[index] = atan2(a[index].imag,a[index].real);
```

This block is the first block discussed that uses a trigonometric function and as such requires some additional discussion.

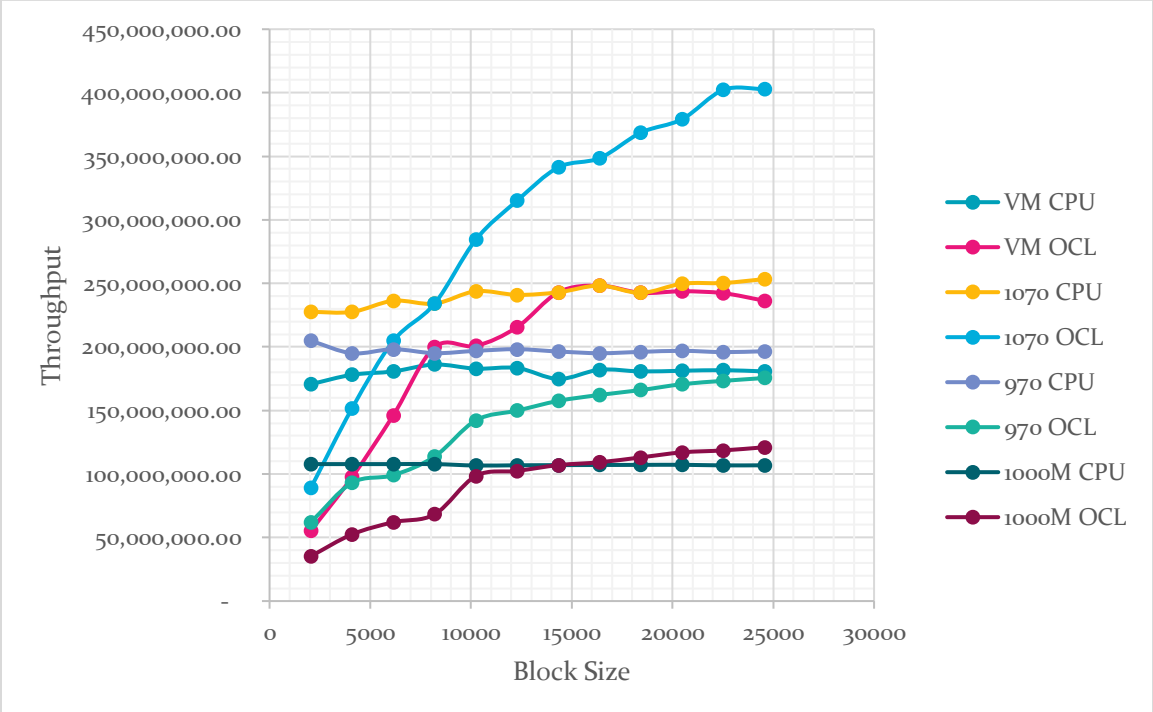
Trigonometric functions can be costly calls. To address this issue, the GNURadio designers took a good performance approach and rather than actually calculating the atan, they use a lookup table to approximate it. The GNURadio code uses the following line to perform the same calculation:

```
out[i] = fast_atan2f(in[i].imag(),in[i].real());
```

There is one word of caution about trig functions in OpenCL. Specifically around the precision supported by the device being used. If the card supports double precision (you can run "clview" and look for "Double Precision Math Support: Yes" or clinfo and look for the "Double Precision section", the resulting OpenCL trig functions will actually be slightly more precise. However, if the device only supports single precision (float), like older graphics cards, precision will actually suffer and it will show up as noise in the trig functions. Testing with the OpenCL Signal Source block will give a clear indication of performance. With double-precision, the OpenCL curve will actually not have some side frequencies present. With single precision the source will look "noisy".

Back to the data for this block.

Data



Timing in Seconds

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	0.000012	0.000037	0.000009	0.000023	0.00001	0.000033	0.000019	0.000058
4096	0.000023	0.000042	0.000018	0.000027	0.000021	0.000044	0.000038	0.000078
6144	0.000034	0.000042	0.000026	0.000030	0.000031	0.000062	0.000057	0.000099
8192	0.000044	0.000041	0.000035	0.000035	0.000042	0.000072	0.000076	0.000120
10240	0.000056	0.000051	0.000042	0.000036	0.000052	0.000072	0.000096	0.000104
12288	0.000067	0.000057	0.000051	0.000039	0.000062	0.000082	0.000115	0.000120
14336	0.000082	0.000059	0.000059	0.000042	0.000073	0.000091	0.000134	0.000134
16384	0.000090	0.000066	0.000066	0.000047	0.000084	0.000101	0.000153	0.000150
18432	0.000102	0.000076	0.000076	0.000050	0.000094	0.000111	0.000172	0.000163
20480	0.000113	0.000084	0.000082	0.000054	0.000104	0.000120	0.000191	0.000175
22528	0.000124	0.000093	0.000090	0.000056	0.000115	0.000130	0.000211	0.000190
24576	0.000136	0.000104	0.000097	0.000061	0.000125	0.000140	0.000230	0.000203

Sample throughput based on time and block size

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	170,666,666.67	55,351,351.35	227,555,555.56	89,043,478.26	204,800,000.00	62,060,606.06	107,789,473.68	35,310,344.83
4096	178,086,956.52	97,523,809.52	227,555,555.56	151,703,703.70	195,047,619.05	93,090,909.09	107,789,473.68	52,512,820.51
6144	180,705,882.35	146,285,714.29	236,307,692.31	204,800,000.00	198,193,548.39	99,096,774.19	107,789,473.68	62,060,606.06
8192	186,181,818.18	199,804,878.05	234,057,142.86	234,057,142.86	195,047,619.05	113,777,777.78	107,789,473.68	68,266,666.67
10240	182,857,142.86	200,784,313.73	243,809,523.81	284,444,444.44	196,923,076.92	142,222,222.22	106,666,666.67	98,461,538.46
12288	183,402,985.07	215,578,947.37	240,941,176.47	315,076,923.08	198,193,548.39	149,853,658.54	106,852,173.91	102,400,000.00
14336	174,829,268.29	242,983,050.85	242,983,050.85	341,333,333.33	196,383,561.64	157,538,461.54	106,985,074.63	106,985,074.63
16384	182,044,444.44	248,242,424.24	248,242,424.24	348,595,744.68	195,047,619.05	162,217,821.78	107,084,967.32	109,226,666.67
18432	180,705,882.35	242,526,315.79	242,526,315.79	368,640,000.00	196,085,106.38	166,054,054.05	107,162,790.70	113,079,754.60
20480	181,238,938.05	243,809,523.81	249,756,097.56	379,259,259.26	196,923,076.92	170,666,666.67	107,225,130.89	117,028,571.43
22528	181,677,419.35	242,236,559.14	250,311,111.11	402,285,714.29	195,895,652.17	173,292,307.69	106,767,772.51	118,568,421.05
24576	180,705,882.35	236,307,692.31	253,360,824.74	402,885,245.90	196,608,000.00	175,542,857.14	106,852,173.91	121,064,039.41

Observations

The newer GTX 1070 card when processing blocks of 10K or larger clearly shows a performance improvement. This is important due to the full atan calculation versus the table-based approximation approach in that the result implies that for actual processed blocks of 10K and bigger the OpenCL version is both faster and more precise than the CPU-based version.

Complex to Mag Phase

Complex to Mag Phase takes a complex input data stream and splits it into a magnitude and a phase. The magnitude calculation is a basic square root of squares calculation as shown below from the OpenCL kernel:

```
b[index] = sqrt((aval*aval)+(bval*bval));
```

In the CPU implementation this is accomplished by a Volk function as shown below:

```
volk_32fc_magnitude_32f_u(out0, in, noi);
```

The phase is then calculated with an atan call matching the Complex To Arg calculation as:

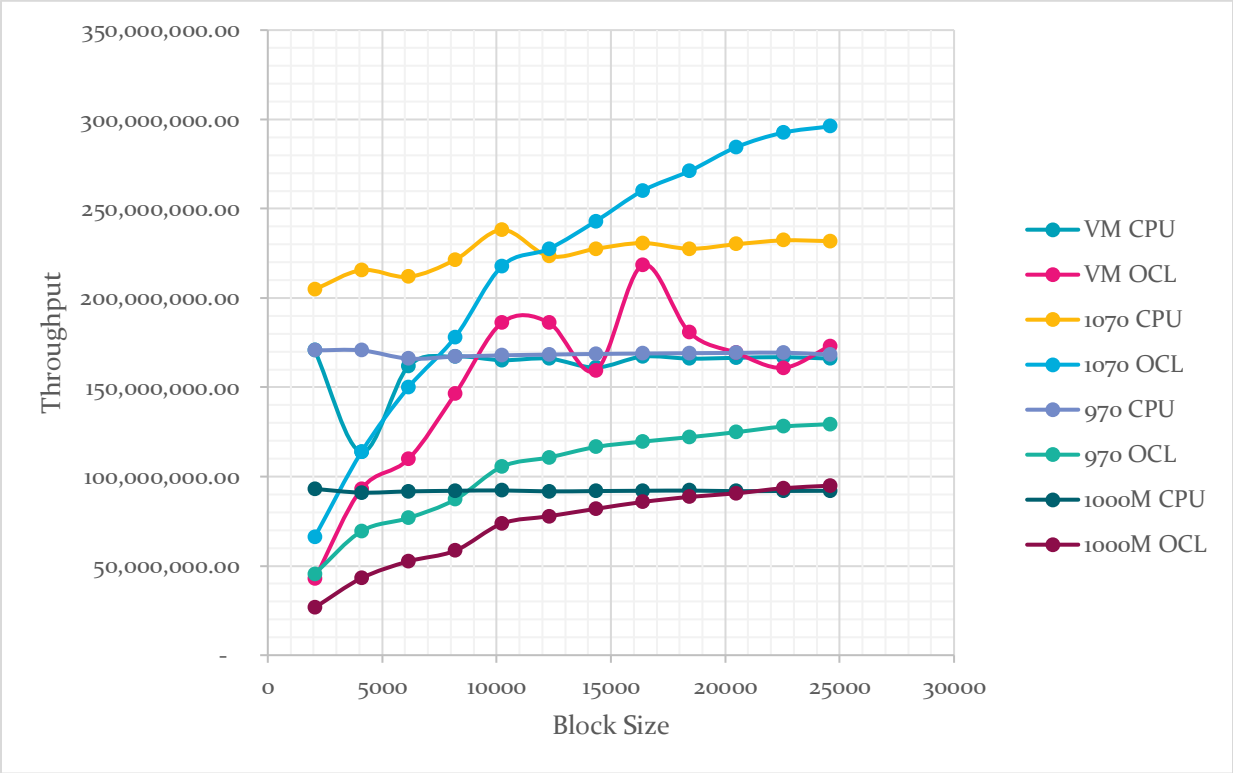
```
c[index] = atan2(a[index].imag,a[index].real);
```

or this for the CPU implementation:

```
out[i] = fast_atan2f(in[i].imag(),in[i].real());
```

Because this block not only includes the calculation from the Complex To Arg calculation, but also an additional magnitude calculation. It has all the hallmarks for OpenCL acceleration. Adding the magnitude calculation increases kernel complexity which generally leads to OpenCL kernel performance gains over CPU-only implementations. And this is exactly the results observed in the data below.

Data



Timing in Seconds

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	0.000012	0.000048	0.000010	0.000031	0.000012	0.000045	0.000022	0.000077
4096	0.000036	0.000044	0.000019	0.000036	0.000024	0.000059	0.000045	0.000095
6144	0.000038	0.000056	0.000029	0.000041	0.000037	0.00008	0.000067	0.000117
8192	0.000049	0.000056	0.000037	0.000046	0.000049	0.000094	0.000089	0.000140
10240	0.000062	0.000055	0.000043	0.000047	0.000061	0.000097	0.000111	0.000139
12288	0.000074	0.000066	0.000055	0.000054	0.000073	0.000111	0.000134	0.000158
14336	0.000089	0.000090	0.000063	0.000059	0.000085	0.000123	0.000156	0.000175
16384	0.000098	0.000075	0.000071	0.000063	0.000097	0.000137	0.000178	0.000191
18432	0.000111	0.000102	0.000081	0.000068	0.000109	0.000151	0.000200	0.000208
20480	0.000123	0.000121	0.000089	0.000072	0.000121	0.000164	0.000223	0.000226
22528	0.000135	0.000140	0.000097	0.000077	0.000133	0.000176	0.000245	0.000241
24576	0.000148	0.000142	0.000106	0.000083	0.000146	0.000190	0.000267	0.000259

Sample throughput based on time and block size

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	170,666,666.67	42,666,666.67	204,800,000.00	66,064,516.13	170,666,666.67	45,511,111.11	93,090,909.09	26,597,402.60
4096	113,777,777.78	93,090,909.09	215,578,947.37	113,777,777.78	170,666,666.67	69,423,728.81	91,022,222.22	43,115,789.47
6144	161,684,210.53	109,714,285.71	211,862,068.97	149,853,658.54	166,054,054.05	76,800,000.00	91,701,492.54	52,512,820.51
8192	167,183,673.47	146,285,714.29	221,405,405.41	178,086,956.52	167,183,673.47	87,148,936.17	92,044,943.82	58,514,285.71
10240	165,161,290.32	186,181,818.18	238,139,534.88	217,872,340.43	167,868,852.46	105,567,010.31	92,252,252.25	73,669,064.75
12288	166,054,054.05	186,181,818.18	223,418,181.82	227,555,555.56	168,328,767.12	110,702,702.70	91,701,492.54	77,772,151.90
14336	161,078,651.69	159,288,888.89	227,555,555.56	242,983,050.85	168,658,823.53	116,552,845.53	91,897,435.90	81,920,000.00
16384	167,183,673.47	218,453,333.33	230,760,563.38	260,063,492.06	168,907,216.49	119,591,240.88	92,044,943.82	85,780,104.71
18432	166,054,054.05	180,705,882.35	227,555,555.56	271,058,823.53	169,100,917.43	122,066,225.17	92,160,000.00	88,615,384.62
20480	166,504,065.04	169,256,198.35	230,112,359.55	284,444,444.44	169,256,198.35	124,878,048.78	91,838,565.02	90,619,469.03
22528	166,874,074.07	160,914,285.71	232,247,422.68	292,571,428.57	169,383,458.65	128,000,000.00	91,951,020.41	93,477,178.42
24576	166,054,054.05	173,070,422.54	231,849,056.60	296,096,385.54	168,328,767.12	129,347,368.42	92,044,943.82	94,888,030.89

Observations

The results were as expected in that the OpenCL versions outperformed the CPU-only implementations. However due to the extra calculations, the performance benefit happens slightly later in the curve. Looking at the 1070 platform, the CPU and OpenCL implementations are approximately even at 12,288 samples with the OpenCL performance benefit not showing up until 14336 samples.

It is postulated that the cost of the extra buffer in/out copy to produce both the magnitude and the phase as output is the reason for the delayed benefit.

OFFLOAD LIST

Some blocks showed performance with mixed results depending on the card. For instance all blocks in this list were actually accelerated on the NVIDIA 1070 card, however the 970 and 1000M cards showed better CPU performance.

Given the time of writing, NVIDIA has released their 1070 and 1080 cards to address the higher performance requirements for virtual reality. It is clear that these cards outperform their predecessors based on their results in this study. In terms of SDR and OpenCL accelerating GNURadio blocks, this means that these cards can mean the difference between OpenCL implementations outperforming their CPU equivalents versus not.

While an intuitive conclusion, it is recommended that in order to get the most benefit in OpenCL that the newest hardware possible be used. As demonstrated in the data below these "offload" blocks can actually be classified as "accelerated" on the newer 1070+ cards and are grouped that way in the GNURadio block groups.

Mag Phase to Complex

The Mag Phase to Complex block performs they opposite function as the Complex to Mag Phase block. Rather than a single atan call, this block uses 2 trigonometric functions (sine and cosine) to reverse the process. The OpenCL kernel below shows the process:

```
struct ComplexStruct {
float real;
float imag; };
typedef struct ComplexStruct SComplex;

__kernel void magphasetocomplex(__constant float * a, __constant float * b,
__global SComplex * restrict c) {
    size_t index =  get_global_id(0);
    float mag = a[index];
    float phase = b[index];
    float real = mag*cos(phase);
    float imag = mag*sin(phase);
    c[index].real = real;
    c[index].imag = imag;
}
```

For data blocks larger than constant memory size the following kernel is used:

```
struct ComplexStruct {
float real;
float imag; };
typedef struct ComplexStruct SComplex;

__kernel void magphasetocomplex(__global float * restrict a, __global float *
restrict b, __global SComplex * restrict c) {
    size_t index =  get_global_id(0);
    float mag = a[index];
    float phase = b[index];
    float real = mag*cos(phase);
    float imag = mag*sin(phase);
```

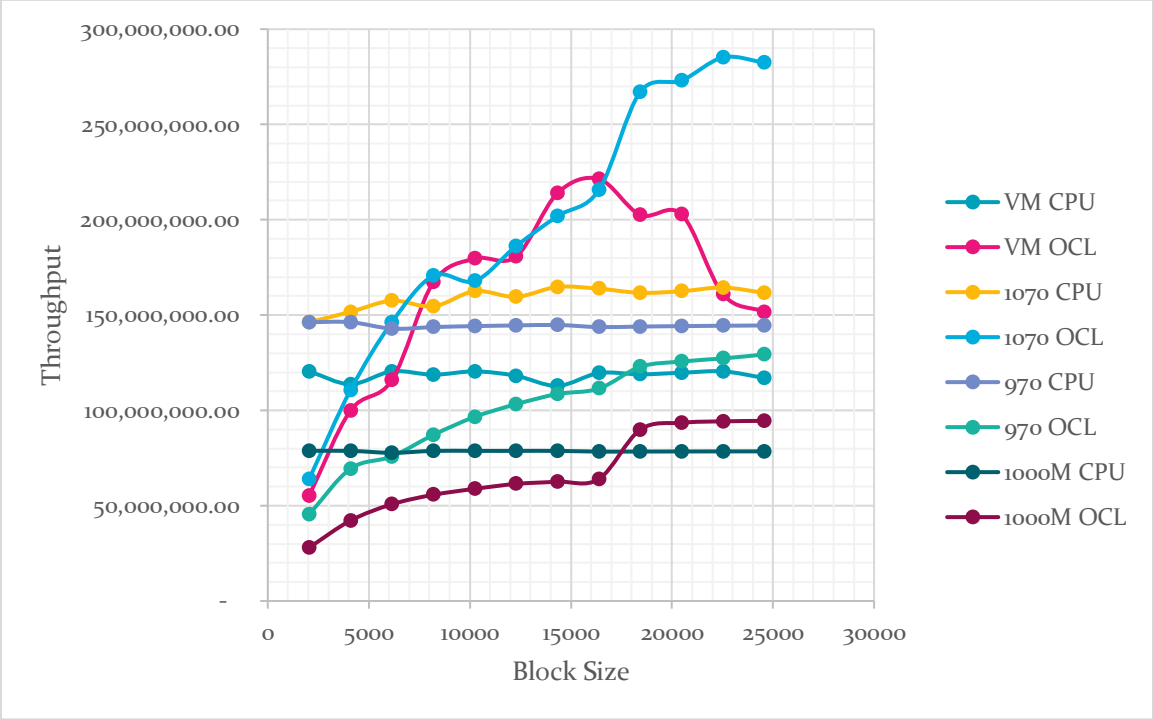
```
    c[index].real = real;
    c[index].imag = imag;
}
```

These kernels can be compared to the GNURadio implementation that does the calculations inline while creating a new complex sample:

```
out[j] = gr_complex (mag[j]*cos(phase[j]),mag[j]*sin(phase[j]));
```

Because of the cost of processing trigonometric functions the expected result is that the OpenCL kernel should have performance benefits over the CPU version with sufficient block sizes.

Data



Timing in Seconds

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	0.000017	0.000037	0.000014	0.000032	0.000014	0.000045	0.000026	0.000073
4096	0.000036	0.000041	0.000027	0.000037	0.000028	0.000059	0.000052	0.000097
6144	0.000051	0.000053	0.000039	0.000042	0.000043	0.000081	0.000079	0.000121
8192	0.000069	0.000049	0.000053	0.000048	0.000057	0.000094	0.000104	0.000147
10240	0.000085	0.000057	0.000063	0.000061	0.000071	0.000106	0.000130	0.000174
12288	0.000104	0.000068	0.000077	0.000066	0.000085	0.000119	0.000156	0.000200
14336	0.000127	0.000067	0.000087	0.000071	0.000099	0.000132	0.000182	0.000229
16384	0.000137	0.000074	0.000100	0.000076	0.000114	0.000147	0.000209	0.000256

18432	0.000155	0.000091	0.000114	0.000069	0.000128	0.000150	0.000235	0.000205
20480	0.000171	0.000101	0.000126	0.000075	0.000142	0.000163	0.000261	0.000219
22528	0.000187	0.00014	0.000137	0.000079	0.000156	0.000177	0.000287	0.000239
24576	0.00021	0.000162	0.000152	0.000087	0.000170	0.000190	0.000313	0.000260

Sample throughput based on time and block size

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	120,470,588.24	55,351,351.35	146,285,714.29	64,000,000.00	146,285,714.29	45,511,111.11	78,769,230.77	28,054,794.52
4096	113,777,777.78	99,902,439.02	151,703,703.70	110,702,702.70	146,285,714.29	69,423,728.81	78,769,230.77	42,226,804.12
6144	120,470,588.24	115,924,528.30	157,538,461.54	146,285,714.29	142,883,720.93	75,851,851.85	77,772,151.90	50,776,859.50
8192	118,724,637.68	167,183,673.47	154,566,037.74	170,666,666.67	143,719,298.25	87,148,936.17	78,769,230.77	55,727,891.16
10240	120,470,588.24	179,649,122.81	162,539,682.54	167,868,852.46	144,225,352.11	96,603,773.58	78,769,230.77	58,850,574.71
12288	118,153,846.15	180,705,882.35	159,584,415.58	186,181,818.18	144,564,705.88	103,260,504.20	78,769,230.77	61,440,000.00
14336	112,881,889.76	213,970,149.25	164,781,609.20	201,915,492.96	144,808,080.81	108,606,060.61	78,769,230.77	62,602,620.09
16384	119,591,240.88	221,405,405.41	163,840,000.00	215,578,947.37	143,719,298.25	111,455,782.31	78,392,344.50	64,000,000.00
18432	118,916,129.03	202,549,450.55	161,684,210.53	267,130,434.78	144,000,000.00	122,880,000.00	78,434,042.55	89,912,195.12
20480	119,766,081.87	202,772,277.23	162,539,682.54	273,066,666.67	144,225,352.11	125,644,171.78	78,467,432.95	93,515,981.74
22528	120,470,588.24	160,914,285.71	164,437,956.20	285,164,556.96	144,410,256.41	127,276,836.16	78,494,773.52	94,259,414.23
24576	117,028,571.43	151,703,703.70	161,684,210.53	282,482,758.62	144,564,705.88	129,347,368.42	78,517,571.88	94,523,076.92

Observations

What was observed from this run was that the OpenCL version actually performed worse than the CPU version for all but the new 1070 card. In fact for the 970 and 1000M cards, the OpenCL performance never exceeded the CPU performance. However the 1070 card started to exceed the CPU performance at 8192 processed data samples.

Signal Source

The signal source block provides a number of capabilities within a GNURadio flowgraph. While it can be used as a source in and of itself, it is also used to shift signals in the frequency domain with a complex multiply block.

The most common uses are producing sine and cosine waves of specific amplitudes. However interestingly when producing complex signals, the sine and cosine signals use the same code (data.real = cos(), data.imag = sin()).

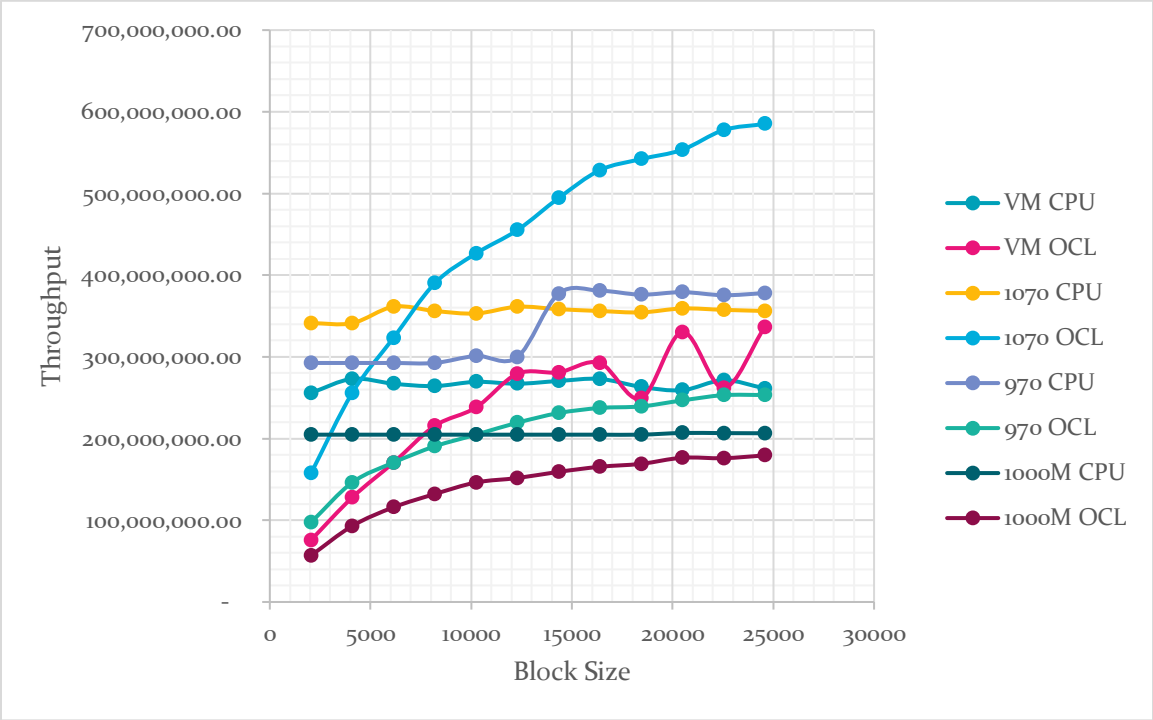
The OpenCL implementation of this function for complex data points is shown below:

```
struct ComplexStruct {
float real;
float imag;
};

typedef struct ComplexStruct SComplex;

__kernel void sig_complex(const float phase, const float
phase_inc, const float ampl, __global SComplex * restrict c) {
    size_t index =  get_global_id(0);
    float dval =  (float)(phase+(phase_inc*(float)index));
     srcStdStr += "    c[index].real = (float)(cos(dval) *
ampl);
     srcStdStr += "    c[index].imag = (float)(sin(dval) *
ampl);
}
```

This can be compared against the GNURadio CPU-only implementation shown below:

```
output[i] = gr_complex(gr::fxpt::cos(d_phase) * d_ampl,
gr::fxpt::sin(d_phase) * d_ampl);
```

Data



Timing in Seconds

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	0.000008	0.000027	0.000006	0.000013	0.000007	0.000021	0.000010	0.000036
4096	0.000015	0.000032	0.000012	0.000016	0.000014	0.000028	0.000020	0.000044
6144	0.000023	0.000036	0.000017	0.000019	0.000021	0.000036	0.000030	0.000053
8192	0.000031	0.000038	0.000023	0.000021	0.000028	0.000043	0.000040	0.000062
10240	0.000038	0.000043	0.000029	0.000024	0.000034	0.000050	0.000050	0.000070
12288	0.000046	0.000044	0.000034	0.000027	0.000041	0.000056	0.000060	0.000081
14336	0.000053	0.000051	0.000040	0.000029	0.000038	0.000062	0.000070	0.000090
16384	0.000060	0.000056	0.000046	0.000031	0.000043	0.000069	0.000080	0.000099
18432	0.00007	0.000074	0.000052	0.000034	0.000049	0.000077	0.000090	0.000109
20480	0.000079	0.000062	0.000057	0.000037	0.000054	0.000083	0.000099	0.000116
22528	0.000083	0.000086	0.000063	0.000039	0.000060	0.000089	0.000109	0.000128
24576	0.000094	0.000073	0.000069	0.000042	0.000065	0.000097	0.000119	0.000137

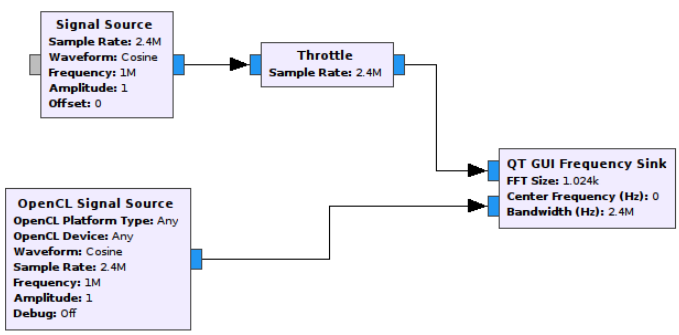
Sample throughput based on time and block size

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	256,000,000.00	75,851,851.85	341,333,333.33	157,538,461.54	292,571,428.57	97,523,809.52	204,800,000.00	56,888,888.89
4096	273,066,666.67	128,000,000.00	341,333,333.33	256,000,000.00	292,571,428.57	146,285,714.29	204,800,000.00	93,090,909.09
6144	267,130,434.78	170,666,666.67	361,411,764.71	323,368,421.05	292,571,428.57	170,666,666.67	204,800,000.00	115,924,528.30
8192	264,258,064.52	215,578,947.37	356,173,913.04	390,095,238.10	292,571,428.57	190,511,627.91	204,800,000.00	132,129,032.26
10240	269,473,684.21	238,139,534.88	353,103,448.28	426,666,666.67	301,176,470.59	204,800,000.00	204,800,000.00	146,285,714.29
12288	267,130,434.78	279,272,727.27	361,411,764.71	455,111,111.11	299,707,317.07	219,428,571.43	204,800,000.00	151,703,703.70
14336	270,490,566.04	281,098,039.22	358,400,000.00	494,344,827.59	377,263,157.89	231,225,806.45	204,800,000.00	159,288,888.89
16384	273,066,666.67	292,571,428.57	356,173,913.04	528,516,129.03	381,023,255.81	237,449,275.36	204,800,000.00	165,494,949.49
18432	263,314,285.71	249,081,081.08	354,461,538.46	542,117,647.06	376,163,265.31	239,376,623.38	204,800,000.00	169,100,917.43
20480	259,240,506.33	330,322,580.65	359,298,245.61	553,513,513.51	379,259,259.26	246,746,987.95	206,868,686.87	176,551,724.14
22528	271,421,686.75	261,953,488.37	357,587,301.59	577,641,025.64	375,466,666.67	253,123,595.51	206,678,899.08	176,000,000.00
24576	261,446,808.51	336,657,534.25	356,173,913.04	585,142,857.14	378,092,307.69	253,360,824.74	206,521,008.40	179,386,861.31

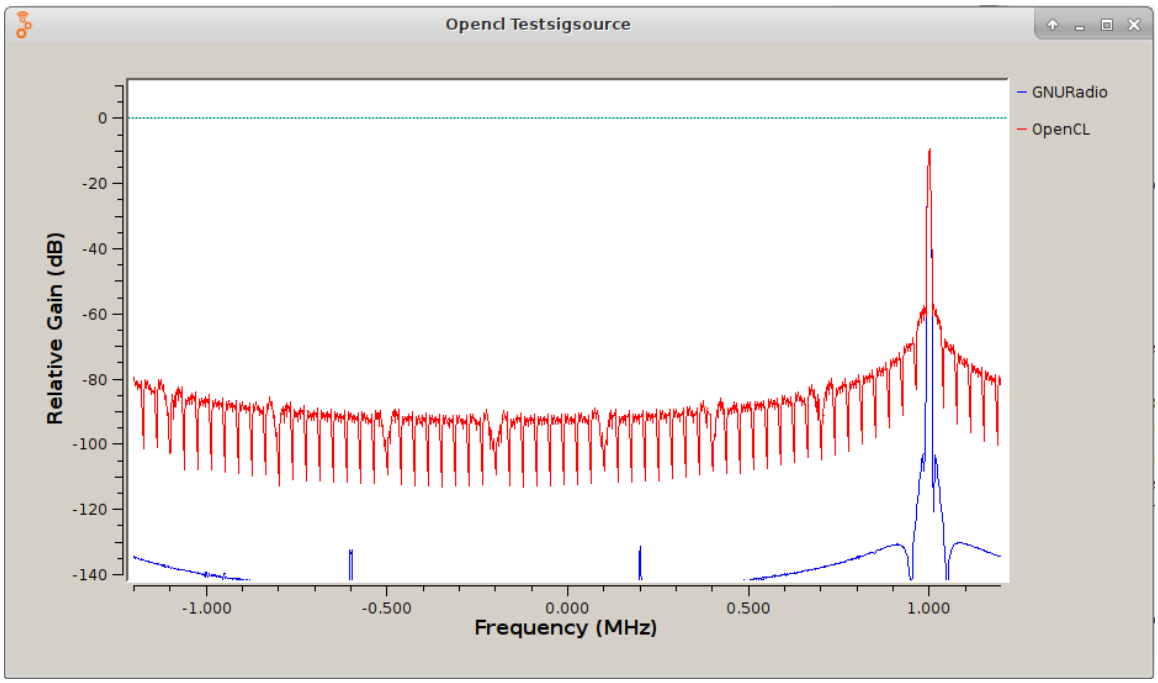
Observations

In this block, both the 970 and 1000M cards showed worse performance in the OpenCL implementation. However the 1070 card outperformed even the faster CPU starting at 819s samples processed.

There is one other very important observation from developing the signal source block. That is that the accuracy of the sin() and cos() functions implemented by OpenCL platforms varies. The anomaly showed up during testing when comparing the OpenCL version of a signal source to the native version as implemented in this flowgraph:



The net result on the VM was the following curves:



At first the assumption was there was an error with the code, however after doing some research it turns out it is an OpenCL anomaly. Running accuracy tests on different platforms showed the following results as output from test-clenabled:

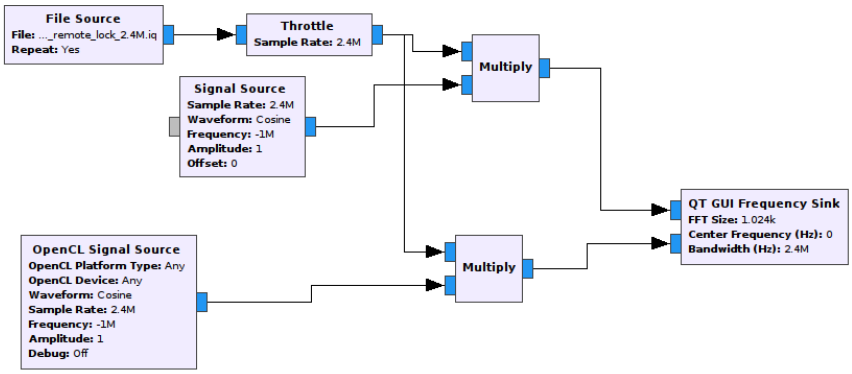
This is the run result from running on the Intel CPU OpenCL driver:

```
maximum error OpenCL versus gnuradio table lookup cos/sin: 0.000056/0.000054
```

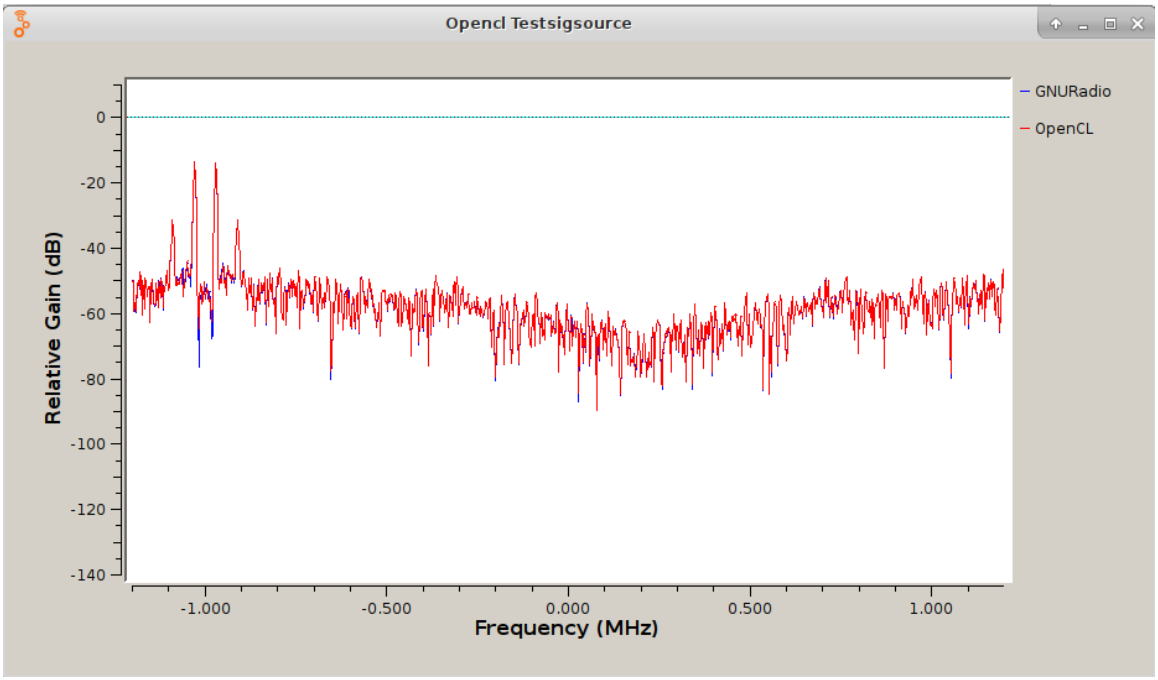
Running the same tests on the 1000M and 1070 card produced the following outcome:

```
maximum error OpenCL versus gnuradio table lookup cos/sin: 0.000009/0.000009
```

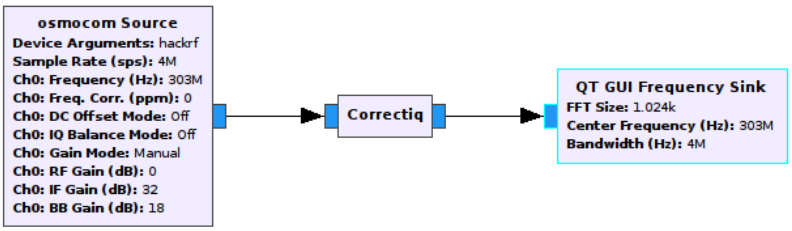
According to the OpenCL specification, sin and cos accuracy are implementation-specific. Which means that the OpenCL code was correct however the precision of the results may vary. This did not have a notable visual impact when shifting signals as in the following comparative flowgraph:



This produced the following frequency curves showing the shift is correct:



However it means that if precision is important, the native block may be preferred, and a more efficient way of getting rid of the center frequency DC spike on devices that do not remove it in their driver would be to use something like my correctiq block (found here https://github.com/ghostop14/gr-correctiq.git) which has been timed at 280+ MSPS and can skip the signal source and multiply requirement or overhead of combining those functions in the XLATING FIR filter by simply doing this:



Quadrature Demod

The quadrature demod block is used for Frequency Shift Keying decoding, one of the three primary digital data transmission modes (with the other two being amplitude shift keying [ASK or On/Off Keying (OOK)] and Phase Shift Keying (PSK). This block performs the following calculations to produce a demodulated signal (note some optimizations for if gain is set to 1 or some other value):

```
srcStdStr = "";
if (f_gain != 1.0) {
       srcStdStr += "#define GAIN " + std::to_string(f_gain) + "\n";
}

srcStdStr += "struct ComplexStruct {\n";
srcStdStr += "float real;\n";
srcStdStr += "float imag; };\n";
srcStdStr += "typedef struct ComplexStruct SComplex;\n";

srcStdStr += "__kernel void quadDemod(__global SComplex * restrict a, __global
float * restrict c) {\n";

srcStdStr += "    size_t index =  get_global_id(0);\n";
srcStdStr += "    float a_r=a[index+1].real;\n";
srcStdStr += "    float a_i=a[index+1].imag;\n";
srcStdStr += "    float b_r=a[index].real;\n";
srcStdStr += "    float b_i=-1.0 * a[index].imag;\n";
srcStdStr += "    float multCCreal = (a_r * b_r) - (a_i*b_i);\n";
srcStdStr += "    float multCCimag = (a_r * b_i) + (a_i * b_r);\n";
if (f_gain != 1.0)
       srcStdStr += "    c[index] = GAIN * atan2(multCCimag,multCCreal);\n";
else
       srcStdStr += "    c[index] = atan2(multCCimag,multCCreal);\n";
srcStdStr += "}\n";
```

This can be compared to the GNURadio implementation which uses a Volk block to do the calculations. The Volk block ultimately performs the following calculation:

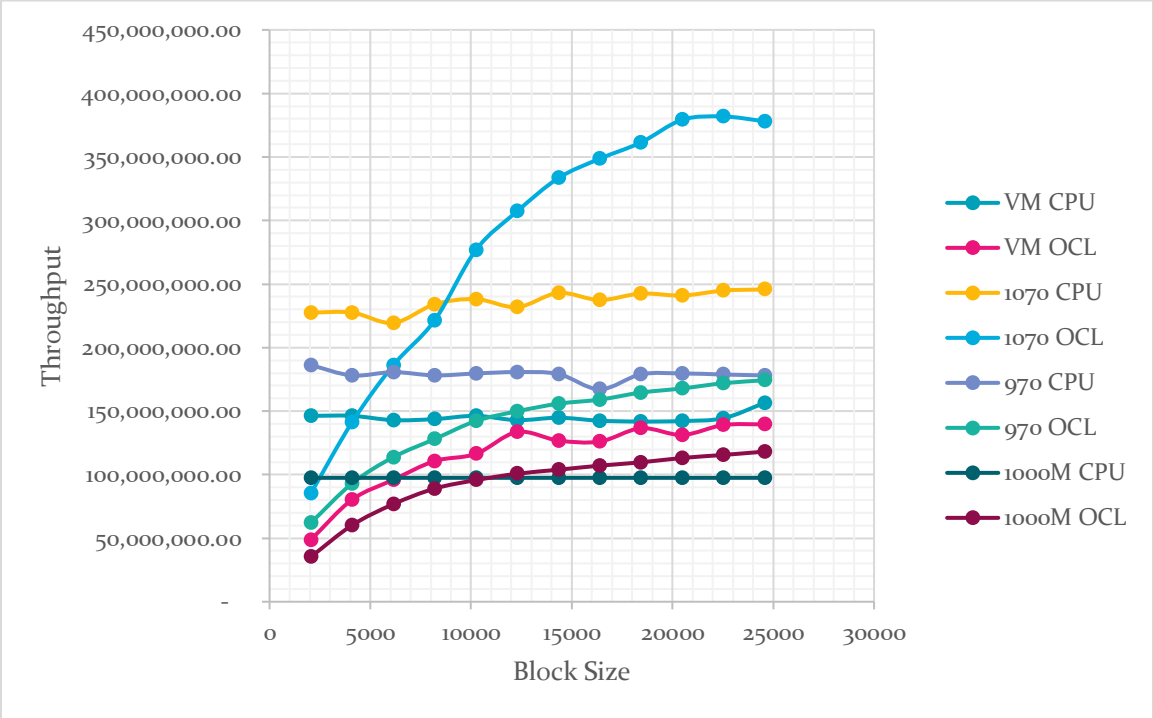
```
        volk_32fc_x2_multiply_conjugate_32fc(&tmp[0], &in[1], &in[0],
noutput_items);
        for(int i = 0; i < noutput_items; i++) {
          out[i] = f_gain * gr::clenabled::fast_atan2f(imag(tmp[i]),
real(tmp[i]));
        }
```

Where the volk_32fc_x2_multiply_conjugate_32fc function performs the following calculation:

```
volk_32fc_x2_multiply_conjugate_32fc_generic(lv_32fc_t* cVector, const
lv_32fc_t* aVector,
                                             const lv_32fc_t* bVector,
unsigned int num_points)
{
  lv_32fc_t* cPtr = cVector;
  const lv_32fc_t* aPtr = aVector;
  const lv_32fc_t* bPtr=  bVector;
  unsigned int number = 0;

  for(number = 0; number < num_points; number++){
    *cPtr++ = (*aPtr++) * lv_conj(*bPtr++);
  }
}
```

Note highlighted in the code the in[1] and in[0] parameters passed to the Volk function. This implies that each sample is actually multiplied by the conjugate of the subsequent data point to produce the output. This makes sense because mathematically multiplying a complex number by its complex conjugate simply produces a real number (the imaginary part cancels itself out to zero). This subsequent conjugate calculation is reflected in the OpenCL implementation with the index+1 reference to match the Volk calculation. Care was taken in the code when creating the buffers to allow for the potential for datasize+1 to be accessed as in the CPU implementation.

Data



Timing in Seconds

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	0.000014	0.000042	0.000009	0.000024	0.000011	0.000033	0.000021	0.000058
4096	0.000028	0.000051	0.000018	0.000029	0.000023	0.000044	0.000042	0.000068
6144	0.000043	0.000064	0.000028	0.000033	0.000034	0.000054	0.000063	0.000080
8192	0.000057	0.000074	0.000035	0.000037	0.000046	0.000064	0.000084	0.000092
10240	0.000070	0.000088	0.000043	0.000037	0.000057	0.000072	0.000105	0.000107
12288	0.000086	0.000092	0.000053	0.000040	0.000068	0.000082	0.000126	0.000122
14336	0.000099	0.000113	0.000059	0.000043	0.000080	0.000092	0.000147	0.000138
16384	0.000115	0.00013	0.000069	0.000047	0.000098	0.000103	0.000168	0.000153
18432	0.00013	0.000135	0.000076	0.000051	0.000103	0.000112	0.000189	0.000168
20480	0.000144	0.000156	0.000085	0.000054	0.000114	0.000122	0.000210	0.000181

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
22528	0.000156	0.000162	0.000092	0.000059	0.000126	0.000131	0.000231	0.000195
24576	0.000157	0.000176	0.000100	0.000065	0.000138	0.000141	0.000252	0.000208

Sample throughput based on time and block size

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	146,285,714.29	48,761,904.76	227,555,555.56	85,333,333.33	186,181,818.18	62,060,606.06	97,523,809.52	35,310,344.83
4096	146,285,714.29	80,313,725.49	227,555,555.56	141,241,379.31	178,086,956.52	93,090,909.09	97,523,809.52	60,235,294.12
6144	142,883,720.93	96,000,000.00	219,428,571.43	186,181,818.18	180,705,882.35	113,777,777.78	97,523,809.52	76,800,000.00
8192	143,719,298.25	110,702,702.70	234,057,142.86	221,405,405.41	178,086,956.52	128,000,000.00	97,523,809.52	89,043,478.26
10240	146,285,714.29	116,363,636.36	238,139,534.88	276,756,756.76	179,649,122.81	142,222,222.22	97,523,809.52	95,700,934.58
12288	142,883,720.93	133,565,217.39	231,849,056.60	307,200,000.00	180,705,882.35	149,853,658.54	97,523,809.52	100,721,311.48
14336	144,808,080.81	126,867,256.64	242,983,050.85	333,395,348.84	179,200,000.00	155,826,086.96	97,523,809.52	103,884,057.97
16384	142,469,565.22	126,030,769.23	237,449,275.36	348,595,744.68	167,183,673.47	159,067,961.17	97,523,809.52	107,084,967.32
18432	141,784,615.38	136,533,333.33	242,526,315.79	361,411,764.71	178,951,456.31	164,571,428.57	97,523,809.52	109,714,285.71
20480	142,222,222.22	131,282,051.28	240,941,176.47	379,259,259.26	179,649,122.81	167,868,852.46	97,523,809.52	113,149,171.27
22528	144,410,256.41	139,061,728.40	244,869,565.22	381,830,508.47	178,793,650.79	171,969,465.65	97,523,809.52	115,528,205.13
24576	156,535,031.85	139,636,363.64	245,760,000.00	378,092,307.69	178,086,956.52	174,297,872.34	97,523,809.52	118,153,846.15

Observations

This block shows mixed performance based on processed block size. It isn't until the block size reaches 10240 that the OpenCL implementation actually starts to outperform the CPU implementation. And a much more significant performance gain is achieved on the newer 1070 hardware.

ENABLED LIST

The modules discussed in this section represent functions that are or may be used in common flow graphs such as a basic complex multiply or constant multiply block but whose performance was not categorized as accelerated or offloaded. In the original design goal of this project, and prior to having a more thorough understanding of both GNURadio operations and OpenCL, it was not understood why these blocks had not been implemented in OpenCL.

Therefore to address others having the same thought processes as mine prior to the study, these blocks are included along with the performance comparisons of each.

Multiply/Add/Subtract/Multiply Conjugate

Within the gr-clenabled code, there are a number of blocks that are all implemented in a single class. These blocks share common traits such as 2 inputs and 1 output, and only differ in the calculation performed. While only the Multiply operation is presented here, the block also provides the following operations:

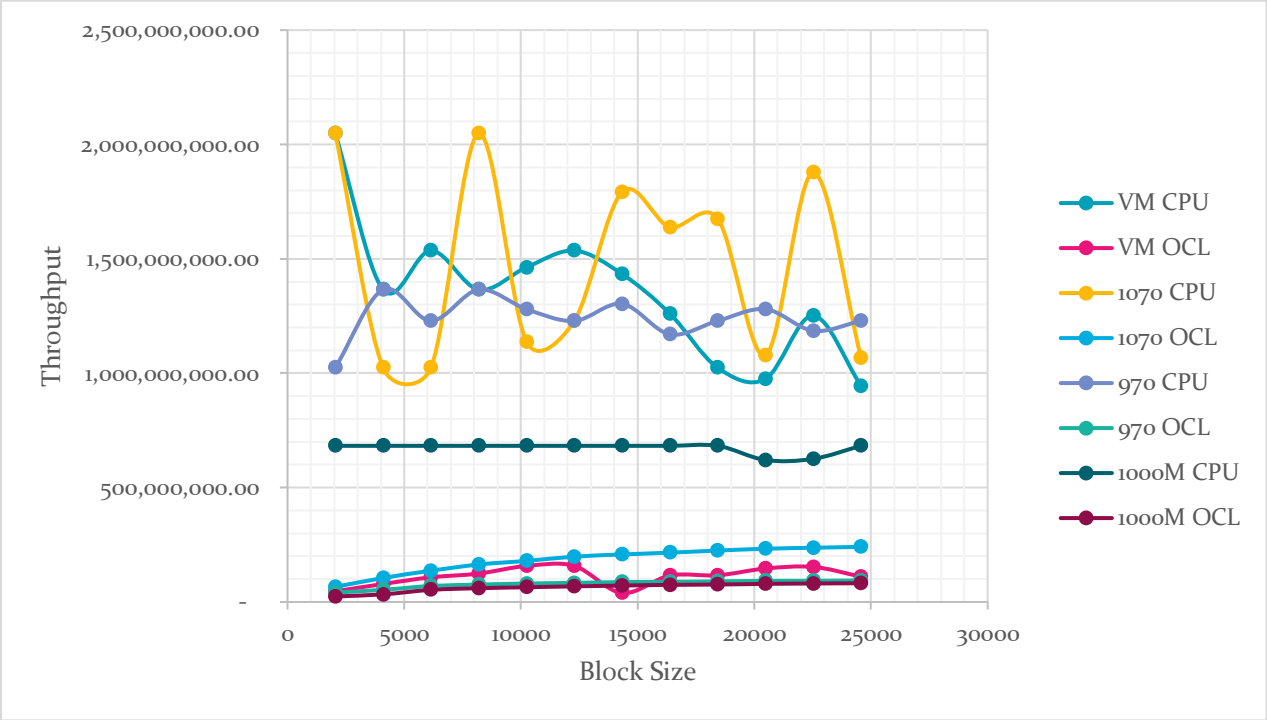
- Add
- Subtract
- Multiply Conjugate

The following code shows the OpenCL kernel construction for the complex data types:

```
                    srcStdStr = "struct ComplexStruct {\n";
                    srcStdStr += "float real;\n";
                    srcStdStr += "float imag; };\n";
                    srcStdStr += "typedef struct ComplexStruct SComplex;\n";

                    fnName = "op_complex";

                    if (useConst)
                          srcStdStr += "__kernel void op_complex(__constant
SComplex * a, __constant SComplex * b, __global SComplex * restrict c) {\n";
                    else
                          srcStdStr += "__kernel void op_complex(__global
SComplex * restrict a, __global SComplex * restrict b, __global SComplex *
restrict c) {\n";

                    if (d_operatorType != MATHOP_EMPTY)
                          srcStdStr += "    size_t index =
get_global_id(0);\n";
                    switch (d_operatorType) {
                    case MATHOP_MULTIPLY:
                    srcStdStr += "    float a_r=a[index].real;\n";
                    srcStdStr += "    float a_i=a[index].imag;\n";
                    srcStdStr += "    float b_r=b[index].real;\n";
                    srcStdStr += "    float b_i=b[index].imag;\n";
                    srcStdStr += "    c[index].real = (a_r * b_r) -
(a_i*b_i);\n";
```

```
                    srcStdStr += "    c[index].imag = (a_r * b_i) + (a_i *
b_r);\n";
                    break;
                    case MATHOP_ADD:
                    srcStdStr += "    c[index].real = a[index].real +
b[index].real;\n";
                    srcStdStr += "    c[index].imag = a[index].imag +
b[index].imag;\n";
                    break;
                    case MATHOP_SUBTRACT:
                    srcStdStr += "    c[index].real = a[index].real -
b[index].real;\n";
                    srcStdStr += "    c[index].imag = a[index].imag -
b[index].imag;\n";
                    break;

                    case MATHOP_MULTIPLY_CONJUGATE:
                       numParams = 2;
                    fnName = "op_complex";

                    srcStdStr = "struct ComplexStruct {\n";
                    srcStdStr += "float real;\n";
                    srcStdStr += "float imag; };\n";
                    srcStdStr += "typedef struct ComplexStruct SComplex;\n";

                    if (useConst)
                           srcStdStr += "__kernel void op_complex(__constant
SComplex * a, __constant SComplex * b, __global SComplex * restrict c) {\n";
                    else
                           srcStdStr += "__kernel void op_complex(__global
SComplex * restrict a, __global SComplex * restrict b, __global SComplex *
restrict c) {\n";

                    srcStdStr += "    size_t index =  get_global_id(0);\n";
                    srcStdStr += "    float a_r=a[index].real;\n";
                    srcStdStr += "    float a_i=a[index].imag;\n";
                    srcStdStr += "    float b_r=b[index].real;\n";
                    srcStdStr += "    float b_i=-1.0 * b[index].imag;\n";
                    srcStdStr += "    c[index].real = (a_r * b_r) -
(a_i*b_i);\n";
                    srcStdStr += "    c[index].imag = (a_r * b_i) + (a_i *
b_r);\n";
                    numConstParams = 2;
                    break;
                    }
                    srcStdStr += "}\n";
```

Data



Timing in Seconds

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	0.000001	0.000044	0.000001	0.000031	0.000002	0.000050	0.000003	0.000085
4096	0.000003	0.000052	0.000004	0.000039	0.000003	0.000077	0.000006	0.000123
6144	0.000004	0.000057	0.000006	0.000045	0.000005	0.000088	0.000009	0.000116
8192	0.000006	0.000066	0.000004	0.000050	0.000006	0.000108	0.000012	0.000136
10240	0.000007	0.000065	0.000009	0.000057	0.000008	0.000127	0.000015	0.000159
12288	0.000008	0.000078	0.000010	0.000062	0.000010	0.000147	0.000018	0.000180
14336	0.000010	0.000365	0.000008	0.000069	0.000011	0.000165	0.000021	0.000200
16384	0.000013	0.000142	0.000010	0.000076	0.000014	0.000186	0.000024	0.000219
18432	0.000018	0.000158	0.000011	0.000082	0.000015	0.000204	0.000027	0.000242
20480	0.000021	0.00014	0.000019	0.000088	0.000016	0.000222	0.000033	0.000259
22528	0.000018	0.000147	0.000012	0.000095	0.000019	0.000243	0.000036	0.000279
24576	0.000026	0.000222	0.000023	0.000102	0.000020	0.000261	0.000036	0.000300

Sample throughput based on time and block size

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	2,048,000,000.00	46,545,454.55	2,048,000,000.00	66,064,516.13	1,024,000,000.00	40,960,000.00	682,666,666.67	24,094,117.65
4096	1,365,333,333.33	78,769,230.77	1,024,000,000.00	105,025,641.03	1,365,333,333.33	53,194,805.19	682,666,666.67	33,300,813.01
6144	1,536,000,000.00	107,789,473.68	1,024,000,000.00	136,533,333.33	1,228,800,000.00	69,818,181.82	682,666,666.67	52,965,517.24
8192	1,365,333,333.33	124,121,212.12	2,048,000,000.00	163,840,000.00	1,365,333,333.33	75,851,851.85	682,666,666.67	60,235,294.12
10240	1,462,857,142.86	157,538,461.54	1,137,777,777.78	179,649,122.81	1,280,000,000.00	80,629,921.26	682,666,666.67	64,402,515.72
12288	1,536,000,000.00	157,538,461.54	1,228,800,000.00	198,193,548.39	1,228,800,000.00	83,591,836.73	682,666,666.67	68,266,666.67
14336	1,433,600,000.00	39,276,712.33	1,792,000,000.00	207,768,115.94	1,303,272,727.27	86,884,848.48	682,666,666.67	71,680,000.00
16384	1,260,307,692.31	115,380,281.69	1,638,400,000.00	215,578,947.37	1,170,285,714.29	88,086,021.51	682,666,666.67	74,812,785.39
18432	1,024,000,000.00	116,658,227.85	1,675,636,363.64	224,780,487.80	1,228,800,000.00	90,352,941.18	682,666,666.67	76,165,289.26
20480	975,238,095.24	146,285,714.29	1,077,894,736.84	232,727,272.73	1,280,000,000.00	92,252,252.25	620,606,060.61	79,073,359.07
22528	1,251,555,555.56	153,251,700.68	1,877,333,333.33	237,136,842.11	1,185,684,210.53	92,707,818.93	625,777,777.78	80,745,519.71
24576	945,230,769.23	110,702,702.70	1,068,521,739.13	240,941,176.47	1,228,800,000.00	94,160,919.54	682,666,666.67	81,920,000.00

Observations

This basic multiply block is very revealing. It clearly shows that for very simple operations the cost of moving the data to the graphics card and back exceeds the amount of time it takes for the CPU to perform the multiply operation. The graph clearly shows the CPU outperforms OpenCL by almost an order of magnitude.

After reviewing the data, the cyclic nature of the CPU results appears to be attributed to the fact that the measurements down to 1 microsecond accuracy may be the cause. Some blocks only differ from their predecessor for the 1070 hardware by 1-2 microseconds. If the timers are rounding up/down to produce an integer, it would explain the wild swings in throughput. In other words 1 microsecond versus 2 microseconds in timing would give the impression of doubling or halving the throughput rates.

In either case, this block could be capable of processing 1 – 2 GSPS due to the simplicity of the calculations. This can be compared against the GPUs that can only process 100-200 MSPS.

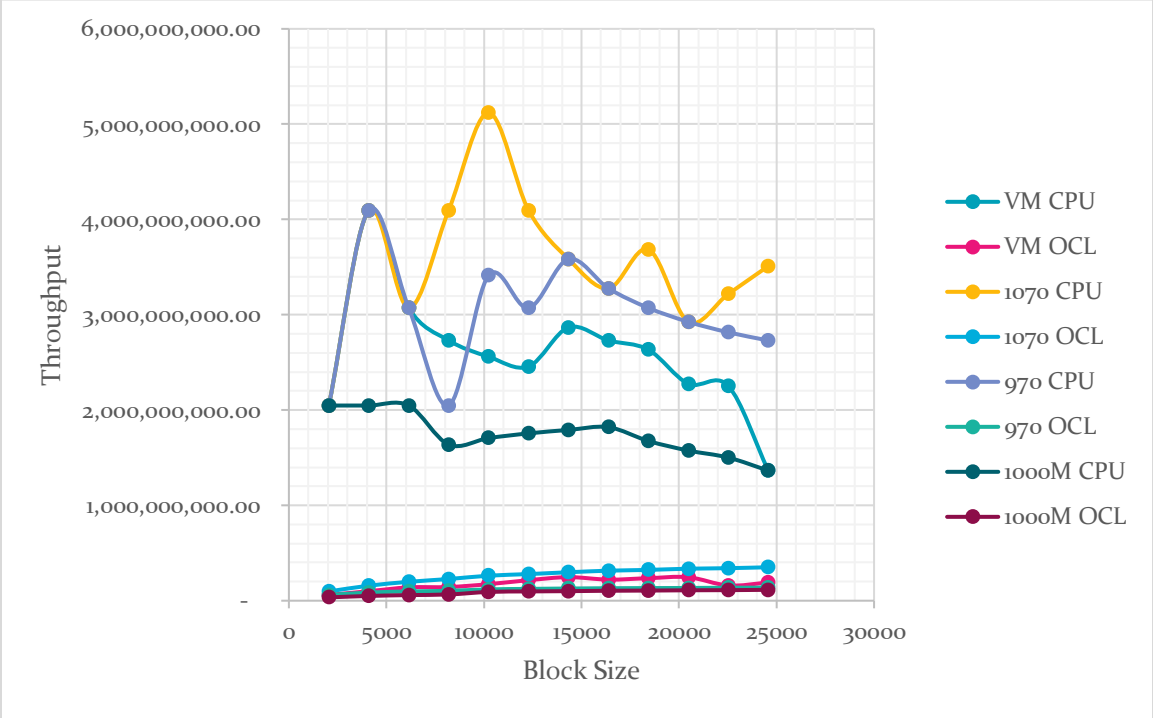
One could argue that for current SDR hardware capable of sampling at 2.4 – 20 MSPS, that this 100-200 MSPS rate is sufficient to offload processing to a GPU. However as noted

later there is a price to pay for running multiple blocks with separate contexts in the same flowgraph which could actually decrease performance. So this is not recommended.

The other observation from the results is in regard to the OpenCL CPU results, in other words OpenCL running on the Intel CPU driver. The effect of this driver really amounts to a less efficient version of multi-threading. What is evident from the result is that the OpenCL CPU version slightly outperformed the CPU-only version for the VM version. This implies that this block could benefit from a true multithreading implementation if true acceleration were desired.

Multiply/Add Const

Data



Timing in Seconds

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	0.000001	0.000034	0.000001	0.000021	0.000001	0.000034	0.000001	0.000059
4096	0.000001	0.000041	0.000001	0.000026	0.000001	0.000048	0.000002	0.000082
6144	0.000002	0.000044	0.000002	0.000031	0.000002	0.000063	0.000003	0.000104
8192	0.000003	0.000057	0.000002	0.000036	0.000004	0.000079	0.000005	0.000125
10240	0.000004	0.000059	0.000002	0.000039	0.000003	0.000086	0.000006	0.000111
12288	0.000005	0.000057	0.000003	0.000044	0.000004	0.000099	0.000007	0.000126
14336	0.000005	0.000058	0.000004	0.000048	0.000004	0.000112	0.000008	0.000143
16384	0.000006	0.000074	0.000005	0.000052	0.000005	0.000125	0.000009	0.000157
18432	0.000007	0.000078	0.000005	0.000057	0.000006	0.000139	0.000011	0.000173
20480	0.000009	0.000083	0.000007	0.000061	0.000007	0.000153	0.000013	0.000187
22528	0.00001	0.000141	0.000007	0.000066	0.000008	0.000165	0.000015	0.000203
24576	0.000018	0.000129	0.000007	0.000070	0.000009	0.000179	0.000018	0.000215

Sample throughput based on time and block size

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	2,048,000,000.00	60,235,294.12	2,048,000,000.00	97,523,809.52	2,048,000,000.00	60,235,294.12	2,048,000,000.00	34,711,864.41
4096	4,096,000,000.00	99,902,439.02	4,096,000,000.00	157,538,461.54	4,096,000,000.00	85,333,333.33	2,048,000,000.00	49,951,219.51
6144	3,072,000,000.00	139,636,363.64	3,072,000,000.00	198,193,548.39	3,072,000,000.00	97,523,809.52	2,048,000,000.00	59,076,923.08
8192	2,730,666,666.67	143,719,298.25	4,096,000,000.00	227,555,555.56	2,048,000,000.00	103,696,202.53	1,638,400,000.00	65,536,000.00
10240	2,560,000,000.00	173,559,322.03	5,120,000,000.00	262,564,102.56	3,413,333,333.33	119,069,767.44	1,706,666,666.67	92,252,252.25
12288	2,457,600,000.00	215,578,947.37	4,096,000,000.00	279,272,727.27	3,072,000,000.00	124,121,212.12	1,755,428,571.43	97,523,809.52
14336	2,867,200,000.00	247,172,413.79	3,584,000,000.00	298,666,666.67	3,584,000,000.00	128,000,000.00	1,792,000,000.00	100,251,748.25
16384	2,730,666,666.67	221,405,405.41	3,276,800,000.00	315,076,923.08	3,276,800,000.00	131,072,000.00	1,820,444,444.44	104,356,687.90
18432	2,633,142,857.14	236,307,692.31	3,686,400,000.00	323,368,421.05	3,072,000,000.00	132,604,316.55	1,675,636,363.64	106,543,352.60
20480	2,275,555,555.56	246,746,987.95	2,925,714,285.71	335,737,704.92	2,925,714,285.71	133,856,209.15	1,575,384,615.38	109,518,716.58
22528	2,252,800,000.00	159,773,049.65	3,218,285,714.29	341,333,333.33	2,816,000,000.00	136,533,333.33	1,501,866,666.67	110,975,369.46
24576	1,365,333,333.33	190,511,627.91	3,510,857,142.86	351,085,714.29	2,730,666,666.67	137,296,089.39	1,365,333,333.33	114,306,976.74

Observations

This block performed very similarly to the Multiply block.

Complex to Mag

Complex to Mag is a block used for a number of applications, however in digital data processing it is most commonly used to demodulate ASK/OOK signals. Given the goal of covering all three primary digital data modes (ASK/FSK/PSK), it was logical to include this block in the study.

This block is simply implemented in OpenCL as a square root of squares as shown in the code below which builds the kernel:

```
       srcStdStr = "";
       srcStdStr += "struct ComplexStruct {\n";
       srcStdStr += "float real;\n";
       srcStdStr += "float imag; };\n";
       srcStdStr += "typedef struct ComplexStruct SComplex;\n";

       if (useConst)
              srcStdStr += "__kernel void complextomag(__constant SComplex * a,
__global float * restrict c) {\n";
       else
              srcStdStr += "__kernel void complextomag(__global SComplex *
restrict a, __global float * restrict c) {\n";

       srcStdStr += "    size_t index =  get_global_id(0);\n";
       srcStdStr += "    float aval = a[index].imag;\n";
       srcStdStr += "    float bval = a[index].real;\n";
       srcStdStr += "    c[index] = sqrt((aval*aval)+(bval*bval));\n";
       srcStdStr += "}\n";
```

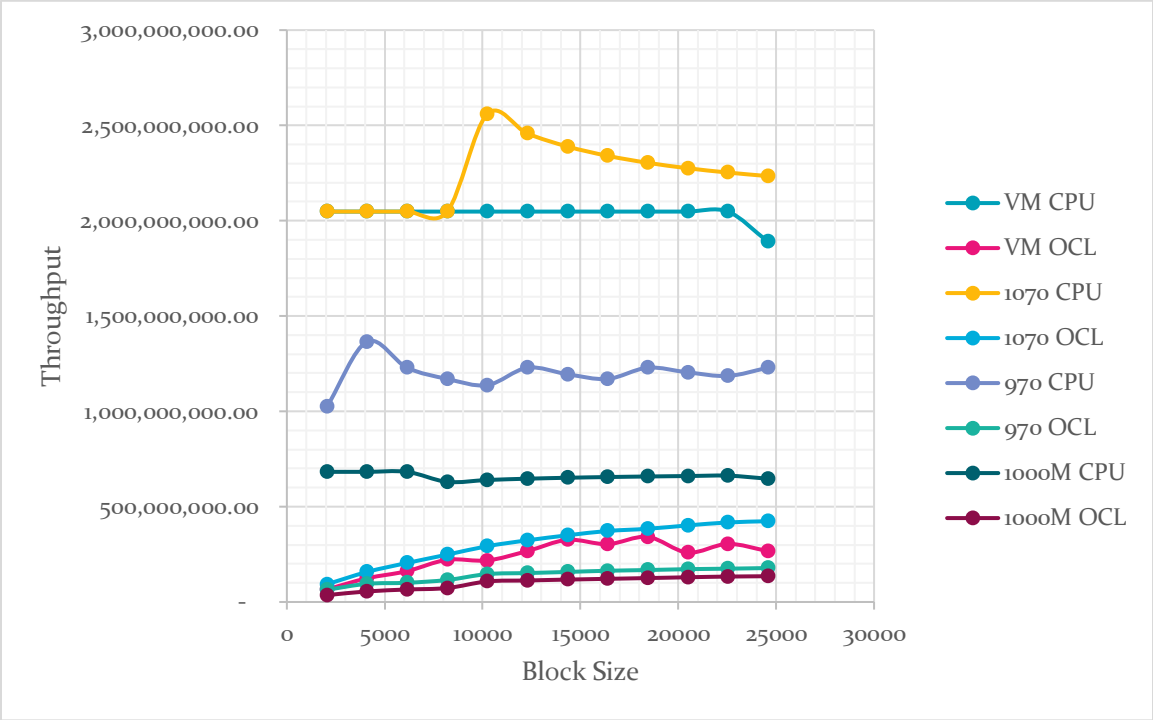
This can be compared against the GNURadio implementation which uses the following Volk call:

```
volk_32fc_magnitude_32f_u(out, in, noutput_items);
```

This calculation can ultimately be seen in the generic version of the Volk call:

```
volk_32fc_magnitude_32f_generic(float* magnitudeVector, const lv_32fc_t*
complexVector, unsigned int num_points)
{
  const float* complexVectorPtr = (float*)complexVector;
  float* magnitudeVectorPtr = magnitudeVector;
  unsigned int number = 0;
  for(number = 0; number < num_points; number++){
    const float real = *complexVectorPtr++;
    const float imag = *complexVectorPtr++;
    *magnitudeVectorPtr++ = sqrtf((real*real) + (imag*imag));
  }
}
```

Data



Timing in Seconds

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	0.000001	0.000031	0.000001	0.000022	0.000002	0.000032	0.000003	0.000057
4096	0.000002	0.000033	0.000002	0.000026	0.000003	0.000043	0.000006	0.000074
6144	0.000003	0.000038	0.000003	0.000030	0.000005	0.000061	0.000009	0.000094
8192	0.000004	0.000037	0.000004	0.000033	0.000007	0.000071	0.000013	0.000112
10240	0.000005	0.000047	0.000004	0.000035	0.000009	0.000070	0.000016	0.000095
12288	0.000006	0.000046	0.000005	0.000038	0.000010	0.000081	0.000019	0.000109
14336	0.000007	0.000044	0.000006	0.000041	0.000012	0.000091	0.000022	0.000122
16384	0.000008	0.000054	0.000007	0.000044	0.000014	0.000100	0.000025	0.000135
18432	0.000009	0.000054	0.000008	0.000048	0.000015	0.000110	0.000028	0.000147
20480	0.000010	0.000079	0.000009	0.000051	0.000017	0.000119	0.000031	0.000158
22528	0.000011	0.000074	0.000010	0.000054	0.000019	0.000129	0.000034	0.000169
24576	0.000013	0.000092	0.000011	0.000058	0.000020	0.000138	0.000038	0.000182

Sample throughput based on time and block size

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	2,048,000,000.00	66,064,516.13	2,048,000,000.00	93,090,909.09	1,024,000,000.00	64,000,000.00	682,666,666.67	35,929,824.56
4096	2,048,000,000.00	124,121,212.12	2,048,000,000.00	157,538,461.54	1,365,333,333.33	95,255,813.95	682,666,666.67	55,351,351.35
6144	2,048,000,000.00	161,684,210.53	2,048,000,000.00	204,800,000.00	1,228,800,000.00	100,721,311.48	682,666,666.67	65,361,702.13
8192	2,048,000,000.00	221,405,405.41	2,048,000,000.00	248,242,424.24	1,170,285,714.29	115,380,281.69	630,153,846.15	73,142,857.14
10240	2,048,000,000.00	217,872,340.43	2,560,000,000.00	292,571,428.57	1,137,777,777.78	146,285,714.29	640,000,000.00	107,789,473.68
12288	2,048,000,000.00	267,130,434.78	2,457,600,000.00	323,368,421.05	1,228,800,000.00	151,703,703.70	646,736,842.11	112,733,944.95
14336	2,048,000,000.00	325,818,181.82	2,389,333,333.33	349,658,536.59	1,194,666,666.67	157,538,461.54	651,636,363.64	117,508,196.72
16384	2,048,000,000.00	303,407,407.41	2,340,571,428.57	372,363,636.36	1,170,285,714.29	163,840,000.00	655,360,000.00	121,362,962.96
18432	2,048,000,000.00	341,333,333.33	2,304,000,000.00	384,000,000.00	1,228,800,000.00	167,563,636.36	658,285,714.29	125,387,755.10
20480	2,048,000,000.00	259,240,506.33	2,275,555,555.56	401,568,627.45	1,204,705,882.35	172,100,840.34	660,645,161.29	129,620,253.16
22528	2,048,000,000.00	304,432,432.43	2,252,800,000.00	417,185,185.19	1,185,684,210.53	174,635,658.91	662,588,235.29	133,301,775.15
24576	1,890,461,538.46	267,130,434.78	2,234,181,818.18	423,724,137.93	1,228,800,000.00	178,086,956.52	646,736,842.11	135,032,967.03

Observations

Again the results were very comparable to the Multiply and Multiply Const blocks with the CPU outperforming the GPU. There was one anomaly in the data that was most evident in the 1070 hardware and that was the unusual jump going from 8192 to 10240 data points. The test was rerun several times and each time the result was consistent. The reason for this jump is unknown.

POOR PERFORMANCE LIST

In general, there are implementations of various core signal processing functions in OpenCL. Most specifically these are basic Fast Fourier Transforms (FFT), both forward and reverse, along with signal filters. Given all of the materials available on the Internet regarding executing FFT's in OpenCL, the clFFT OpenCL FFT library, and vendor-specific implementations such as cuFFT, the expected result was that the OpenCL implementations would outperform the CPU implementations. However the data indicates a different outcome within the context of real-time SDR processing. And once the underlying root causes were understood it did make sense. As a result, this group provided the greatest surprise in the test data results.

Before reviewing the data, some speculation in terms of why these blocks performed worse is be in order. Many of the sample code about doing FFT transforms in OpenCL were geared towards offline processing. In offline processing, blocks or batches of data could be processed at one time, which takes advantage of OpenCL's performance gains with large data blocks. However for real-time processing these blocks are much smaller and must be processed in a time-sensitive way. For GNURadio this equates to processing the blocks provided by the scheduler.

Because an FFT transform works on a complex data set matching in size to the FFT size (in other words a 2048 point FFT would process 2048 data points at a time), moving these blocks individually to the GPU and back pays the performance cost of the memory copies mentioned earlier. It is possible that more focus on batch processing within the code for the FFT and reverse FFT blocks would improve performance, however problems were encountered with clFFT throwing exceptions while using batches. This may be more programmatic errors in this implementation than problems with the library.

However any improvements in the FFT blocks would not be realized in the filter code since the filter calculations have a carry-forward "tail" necessitating FFT blocks be done sequentially. In terms of filters, several options were explored. Given the filter transition width and type, a series of taps are generated. Smaller transition widths produce more taps which require more processing and adjustments to the block sizes. These taps can be applied in either a time domain or frequency domain, however in general the frequency domain application does not take as much processing power and produces exactly the same result. However what this requires is a forward FFT to get the data into the frequency domain, application of the taps, then a reverse FFT to get the transformed signal out. This process produces that "tail" that was mentioned that needs to be applied to the next block before it can be processed, which means processing needs to be done in sequential chunks. Not an optimal implementation for OpenCL in that buffer read/writes need to be executed to move each block to the card. This incurs quite a time penalty over the CPU implementation.

These root causes do account for the lower-than-expected performance seen in the data presented in the next subsections.

FFT Forward

The forward FFT calculation was implemented in OpenCL using the clFFT library as shown below.

Setup:

```
        /* Setup clFFT. */
        clfftSetupData fftSetup;
        err = clfftInitSetupData(&fftSetup);
        err = clfftSetup(&fftSetup);

        err = clfftCreateDefaultPlan(&planHandle, (*context)(), dim,
clLengths);

        /* Set plan parameters. */
        err = clfftSetPlanPrecision(planHandle, CLFFT_SINGLE);

        if (dataType==DTYPE_COMPLEX) {
            err = clfftSetLayout(planHandle, CLFFT_COMPLEX_INTERLEAVED,
CLFFT_COMPLEX_INTERLEAVED);
        }
        else {
            err = clfftSetLayout(planHandle, CLFFT_REAL, CLFFT_REAL);
        }

       clfftSetPlanScale(planHandle, CLFFT_BACKWARD, 1.0f); // By default the
backward scale is set to 1/N so you have to set it here.

        //err = clfftSetResultLocation(planHandle, CLFFT_INPLACE);  // In-place
puts data back in source queue.  Not what we want.
        err = clfftSetResultLocation(planHandle, CLFFT_OUTOFPLACE);

        // using vectors we don't want to change the output multiple since 1
item will be an fft worth of data.
        //          set_output_multiple(fftSize);

        /* Bake the plan. */
        err = clfftBakePlan(planHandle, 1, &(*queue)(), NULL, NULL);
```

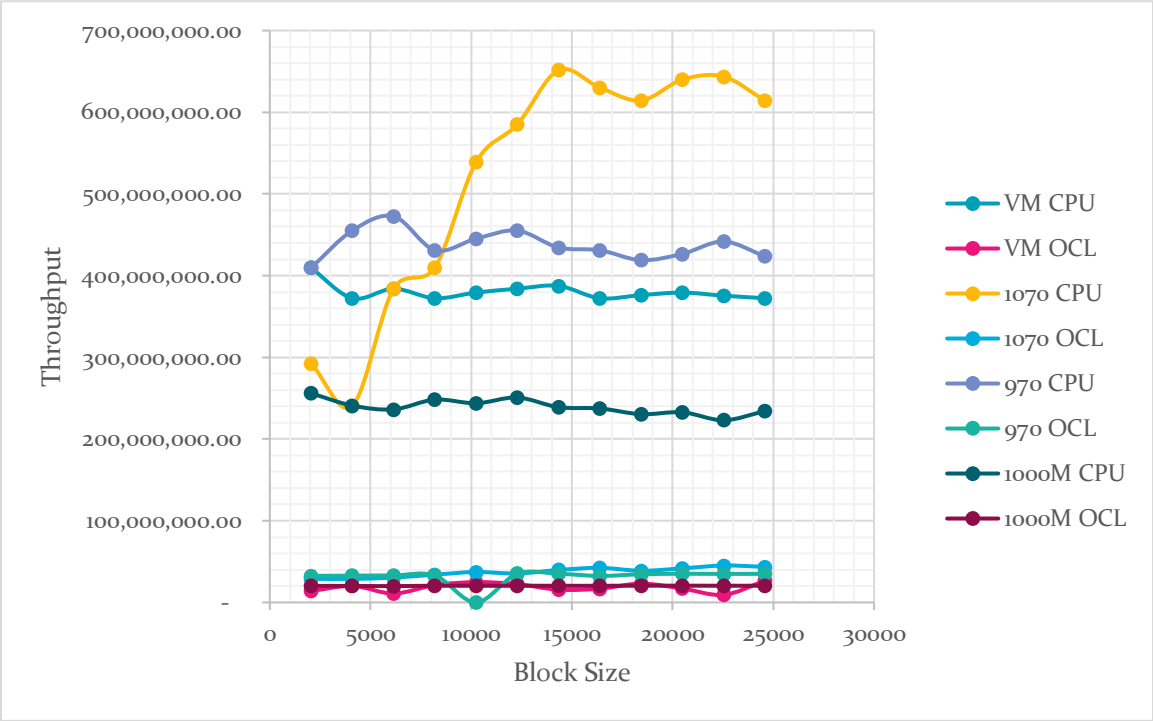
Note the highlighted line required for reverse transforms. For some reason the clFFT library applies a scaling factor by default to reverse transforms. This factor is 1/N where N is the FFT size. This can be overridden with the highlighted line but must be done explicitly or the results would be scaled down and not match the CPU-only implementation results.

The actual transform is then executed at runtime with the following code:

```
            err = clfftEnqueueTransform(planHandle, fftDir, 1,
&((*queue)()), 0, NULL, NULL, &((*aBuffer)()), &((*cBuffer)()), NULL);
```

Note the same code is used for both forward and reverse transforms, the only difference is the fftDir flag which indicates whether the transform is forward or reverse.

Data



Timing in Seconds

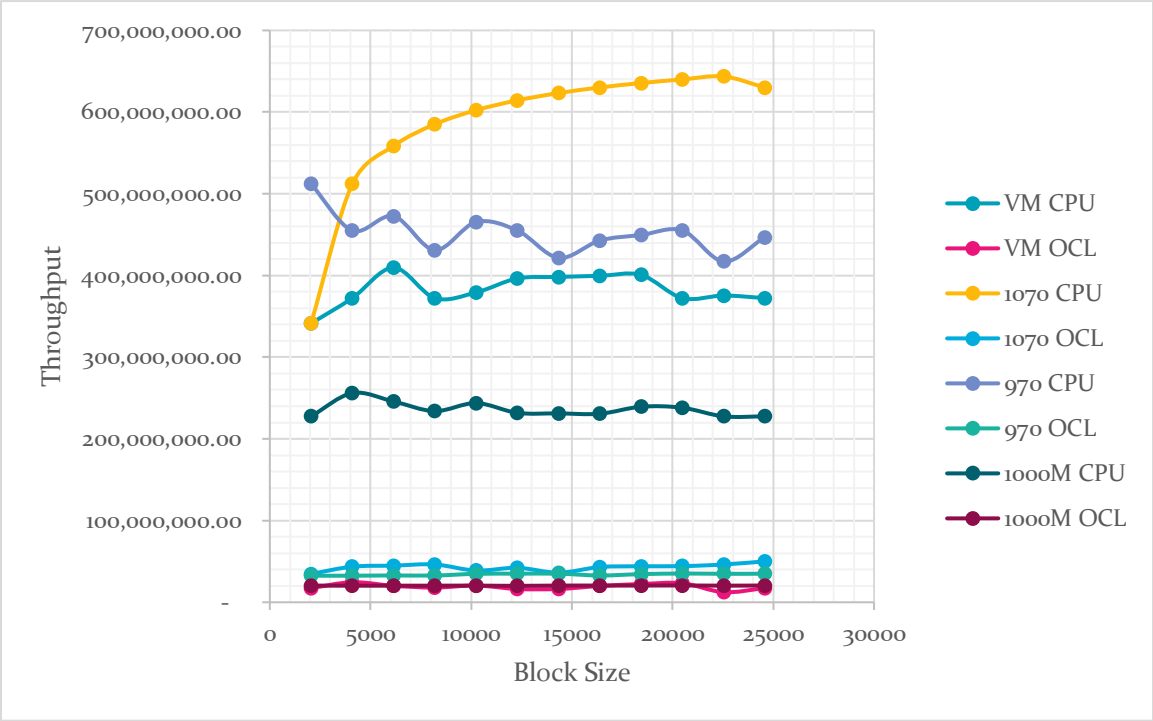
Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	0.000005	0.000147	0.000007	0.000071	0.000005	0.000063	0.000008	0.000101
4096	0.000011	0.000203	0.000017	0.000142	0.000009	0.000125	0.000017	0.000203
6144	0.000016	0.000562	0.000016	0.000202	0.000013	0.000186	0.000026	0.000308
8192	0.000022	0.000386	0.000020	0.000241	0.000019	0.000249	0.000033	0.000401
10240	0.000027	0.000413	0.000019	0.000275	0.000023	0.000292+G10	0.000042	0.000503
12288	0.000032	0.000547	0.000021	0.000347	0.000027	0.000350	0.000049	0.000602
14336	0.000037	0.000922	0.000022	0.000361	0.000033	0.000409	0.000060	0.000704
16384	0.000044	0.000975	0.000026	0.000387	0.000038	0.000505	0.000069	0.000806
18432	0.000049	0.000801	0.000030	0.000476	0.000044	0.000535	0.000080	0.000900
20480	0.000054	0.001200	0.000032	0.000491	0.000048	0.000584	0.000088	0.000999
22528	0.000060	0.002420	0.000035	0.000499	0.000051	0.000646	0.000101	0.001104
24576	0.000066	0.000931	0.000040	0.000564	0.000058	0.000701	0.000105	0.001202

Sample throughput based on time and block size

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	409,600,000.00	13,931,972.79	292,571,428.57	28,845,070.42	409,600,000.00	32,507,936.51	256,000,000.00	20,277,227.72
4096	372,363,636.36	20,177,339.90	240,941,176.47	28,845,070.42	455,111,111.11	32,768,000.00	240,941,176.47	20,177,339.90
6144	384,000,000.00	10,932,384.34	384,000,000.00	30,415,841.58	472,615,384.62	33,032,258.06	236,307,692.31	19,948,051.95
8192	372,363,636.36	21,222,797.93	409,600,000.00	33,991,701.24	431,157,894.74	32,899,598.39	248,242,424.24	20,428,927.68
10240	379,259,259.26	24,794,188.86	538,947,368.42	37,236,363.64	445,217,391.30	#VALUE!	243,809,523.81	20,357,852.88
12288	384,000,000.00	22,464,351.01	585,142,857.14	35,412,103.75	455,111,111.11	35,108,571.43	250,775,510.20	20,411,960.13
14336	387,459,459.46	15,548,806.94	651,636,363.64	39,711,911.36	434,424,242.42	35,051,344.74	238,933,333.33	20,363,636.36
16384	372,363,636.36	16,804,102.56	630,153,846.15	42,335,917.31	431,157,894.74	32,443,564.36	237,449,275.36	20,327,543.42
18432	376,163,265.31	23,011,235.96	614,400,000.00	38,722,689.08	418,909,090.91	34,452,336.45	230,400,000.00	20,480,000.00
20480	379,259,259.26	17,066,666.67	640,000,000.00	41,710,794.30	426,666,666.67	35,068,493.15	232,727,272.73	20,500,500.50
22528	375,466,666.67	9,309,090.91	643,657,142.86	45,146,292.59	441,725,490.20	34,873,065.02	223,049,504.95	20,405,797.10
24576	372,363,636.36	26,397,422.13	614,400,000.00	43,574,468.09	423,724,137.93	35,058,487.87	234,057,142.86	20,445,923.46

FFT Reverse

Data



Timing in Seconds

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	0.000006	0.000118	0.000006	0.000059	0.000004	0.000063	0.000009	0.000100
4096	0.000011	0.000167	0.000008	0.000094	0.000009	0.000126	0.000016	0.000202
6144	0.000015	0.000305	0.000011	0.000137	0.000013	0.000187	0.000025	0.000302
8192	0.000022	0.000455	0.000014	0.000177	0.000019	0.000250	0.000035	0.000403
10240	0.000027	0.000500	0.000017	0.000264	0.000022	0.000294	0.000042	0.000502
12288	0.000031	0.000755	0.000020	0.000291	0.000027	0.000351	0.000053	0.000611
14336	0.000036	0.000880	0.000023	0.000396	0.000034	0.000410	0.000062	0.000701
16384	0.000041	0.000821	0.000026	0.000380	0.000037	0.000499	0.000071	0.000801
18432	0.000046	0.000826	0.000029	0.000418	0.000041	0.000535	0.000077	0.000901
20480	0.000055	0.000878	0.000032	0.000463	0.000045	0.000582	0.000086	0.000995
22528	0.000060	0.001817	0.000035	0.000487	0.000054	0.000647	0.000099	0.001107
24576	0.000066	0.001424	0.000039	0.000491	0.000055	0.000703	0.000108	0.001197

Sample throughput based on time and block size

Samples	VM CPU	VM OCL	1070 CPU	1070 OCL	970 CPU	970 OCL	1000M CPU	1000M OCL
2048	341,333,333.33	17,355,932.20	341,333,333.33	34,711,864.41	512,000,000.00	32,507,936.51	227,555,555.56	20,480,000.00
4096	372,363,636.36	24,526,946.11	512,000,000.00	43,574,468.09	455,111,111.11	32,507,936.51	256,000,000.00	20,277,227.72
6144	409,600,000.00	20,144,262.30	558,545,454.55	44,846,715.33	472,615,384.62	32,855,614.97	245,760,000.00	20,344,370.86
8192	372,363,636.36	18,004,395.60	585,142,857.14	46,282,485.88	431,157,894.74	32,768,000.00	234,057,142.86	20,327,543.42
10240	379,259,259.26	20,480,000.00	602,352,941.18	38,787,878.79	465,454,545.45	34,829,931.97	243,809,523.81	20,398,406.37
12288	396,387,096.77	16,275,496.69	614,400,000.00	42,226,804.12	455,111,111.11	35,008,547.01	231,849,056.60	20,111,292.96
14336	398,222,222.22	16,290,909.09	623,304,347.83	36,202,020.20	421,647,058.82	34,965,853.66	231,225,806.45	20,450,784.59
16384	399,609,756.10	19,956,151.04	630,153,846.15	43,115,789.47	442,810,810.81	32,833,667.33	230,760,563.38	20,454,431.96
18432	400,695,652.17	22,314,769.98	635,586,206.90	44,095,693.78	449,560,975.61	34,452,336.45	239,376,623.38	20,457,269.70
20480	372,363,636.36	23,325,740.32	640,000,000.00	44,233,261.34	455,111,111.11	35,189,003.44	238,139,534.88	20,582,914.57
22528	375,466,666.67	12,398,459.00	643,657,142.86	46,258,726.90	417,185,185.19	34,819,165.38	227,555,555.56	20,350,496.84
24576	372,363,636.36	17,258,426.97	630,153,846.15	50,052,953.16	446,836,363.64	34,958,748.22	227,555,555.56	20,531,328.32

FILTERS

Filters turned out to show quite a variety of performance variation in testing. The number of taps in the specified filter along with FIR versus FFT versions both affected the ultimate throughput of the OpenCL implementation.

Before discussing the results, it's important to understand how taps can vary in a real implementation. We'll use a firdes.low_pass filter as an example. This filter takes gain, a sampling rate, a cutoff frequency, and a transition frequency as the minimum parameters. As seen below, the number of taps varies with sample rate, cutoff, and transition values. Each set was generated with gain 1 (however changing the gain did not impact the taps):

Description: 10 MSPS, 100 KHz Filter, 20% transition
Filter: firdes.low_pass(1, 10e6, 100e3, 0.2*100e3)
Taps: 1205

Description: 2.4 MSPS, 100 KHz Filter, 20% transition
Filter: firdes.low_pass(1, 2.4e6, 100e3, 0.2*100e3)
Taps: 289

Description: 10 MSPS, 50 KHz Filter, 20% transition
Filter: firdes.low_pass(1, 10e6, 50e3, 0.2*50e3)
Taps: 2409

Description: 10 MSPS, 50 KHz Filter, 30% transition
Filter: firdes.low_pass(1, 10e6, 50e3, 0.3*50e3)
Taps: 1607

Description: 10 MSPS, 50 KHz Filter, 10% transition
Filter: firdes.low_pass(1, 10e6, 50e3, 0.1*50e3)
Taps: 4819

The point of this exercise is that changing any one of the parameters can have an impact on the number of taps. So to understand filter performance, one must measure it based on the number of taps, then consider that for each of the filters and sampling rates in use to determine appropriate performance. Luckily, gr-clenabled includes a command-line tool called test-clfilter that takes the number of taps as a command-line parameter and measures performance across all 4 filter types:

1. OpenCL FIR Filter
2. GNURadio FIR Filter
3. OpenCL FFT Filter
4. GNURadio FFT Filter

The first draft of this report did not take enough into account. This section has been reworked to support a more thorough filter analysis. The one change over the rest of the document was that the NVIDIA 1070 system was changed to Ubuntu 16.04 LTS in between the first round of testing due to issues with the PFB Arbitrary Resampler on the version of Debian (potentially due to the 4.9 kernel or kali linux optimizations). This does have an impact on GNURadio/filter results in that CPU calculations did appear to run slightly slower on the 4.4 kernel over the 4.9 kernel.

The following OpenCL implementation was used for the time domain calculations. It shows a number of optimizations such as defining the number of taps as a constant rather than passing it as a parameter (if the number of taps changes, the kernel is recompiled). Note that this version does attempt to optimize if the hardware supports Fused Multiply/Add (FMA) operations. Also, if the number of taps supports it, the kernel is optimized to pass the taps in constant memory for improved speed.

```
"#define K "+ std::to_string(d_ntaps)

struct ComplexStruct {
      float real;
```

```
      float imag;
};
typedef struct ComplexStruct SComplex;

__kernel void td_FIR_complex
(  __global const SComplex * restrict InputArray, // Length N
__constant float * FilterArray, // Length K
__global SComplex *restrict OutputArray // Length N+K-1
)
{
  size_t gid=get_global_id(0);
      // Perform Compute
      SComplex result;
      result.real=0.0f;
      result.imag=0.0f;
      for (int i=0; i<K; i++) {
```

Code to check if FMA support is present

```
      if (hasSingleFMASupport) {
            // gid+i doesn't crash because we pass the larger buffer to
the device and zero out the additional memory
            kernelCode +="          result.real = fma(FilterArray[K-1-
i],InputArray[gid+i].real,result.real);\n";
            kernelCode +="          result.imag = fma(FilterArray[K-1-
i],InputArray[gid+i].imag,result.imag);\n";
      }
      else {
            kernelCode +="          result.real += FilterArray[K-1-
i]*InputArray[gid+i].real;\n";
            kernelCode +="          result.imag += FilterArray[K-1-
i]*InputArray[gid+i].imag;\n";
      }
```

```
      }
      OutputArray[gid].real = result.real;
      OutputArray[gid].imag = result.imag;
}
```

There were a number of examples on the Internet of implementing time-domain filtering with barriers and copying the data to local memory, however this approach was focused on FPGA implementations. When this approach was attempted on GPU's in combination with the constant memory, the kernels would throw exceptions at runtime, presumably due to lack of on-board memory. Therefore this approach with constant memory for the taps and global memory for the filter data was the best that could be achieved on the GPU's.

Given the flexibility in the gr-clenabled code it is possible to use a faster implementation when the OpenCL type is set to Accelerator, however an expensive FPGA was not available to test for this study so this implementation was left in the code for FPGA's as well for compatibility.

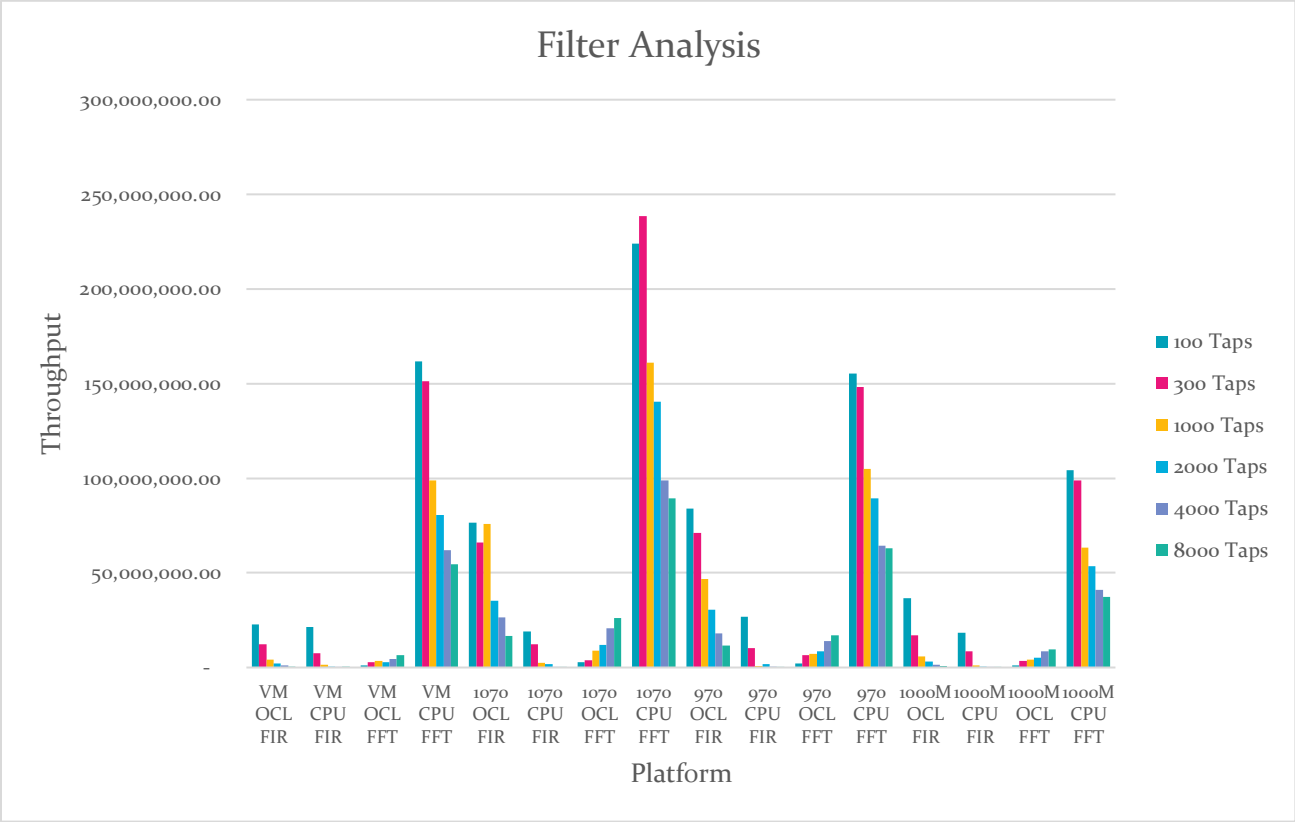
The frequency domain implementation can be seen in the following run-time routine leveraging the clFFT forward and reverse transforms. Note that since it had previously been proven in the Multiply block that the CPU multiply function was faster, the FFT calculations are done in OpenCL and the multiply to apply the taps is done on the CPU. In the code below, the GNURadio block code is commented out with the OpenCL implementation immediately following it.

```
      for(int i = 0; i < ninput_items; i += d_fft_filter->d_nsamples) {
        // Move block of data to forward FFT buffer
      /*
        memcpy(d_fwdfft->get_inbuf(), &input[i], d_fft_filter->d_nsamples *
sizeof(gr_complex));

        // zero out any data past d_fft_filter->d_nsamples to fft_size
        for(j = d_fft_filter->d_nsamples; j < d_fft_filter->d_fftsize; j++)
             d_fwdfft->get_inbuf()[j] = 0;
        // Run the transform
        d_fwdfft->execute();     // compute fwd xform
    */

        queue->enqueueWriteBuffer(*aBuffer,CL_TRUE,0,d_fft_filter-
>d_nsamples*dataSize,(void *)&in[i]);
        queue->enqueueWriteBuffer(*aBuffer,CL_TRUE,d_fft_filter-
>d_nsamples*dataSize,(d_fft_filter->d_fftsize-d_fft_filter-
>d_nsamples)*dataSize,(void *)zeroBuff);
        err = clfftEnqueueTransform(planHandle, CLFFT_FORWARD, 1, &(*queue)(),
0, NULL, NULL, &(*aBuffer)(), &(*cBuffer)(), NULL);
        err = clFinish((*queue)());

        // Get the fwd FFT data out
    //  gr_complex *a = d_fwdfft->get_outbuf();
        queue->enqueueReadBuffer(*cBuffer,CL_TRUE,0,d_fft_filter-
>d_fftsize*dataSize,(void *)tmpFFTBuff);
        gr_complex *a;
        a=(gr_complex *)tmpFFTBuff;

        gr_complex *b = d_fft_filter->d_xformed_taps;

        // set up the inv FFT buffer to receive the complex multiplied data
    //  gr_complex *c = d_invfft->get_inbuf();
        gr_complex *c;
        c=(gr_complex *)ifftBuff;

        // Original volk call.  Might as well use SIMD / SSE
        // I've tried, but this VOLK CALL JUST CRASHES!  DON"T USE IT UNTIL I
KNOW WHY
        //volk_32fc_x2_multiply_32fc_a(c, a, b, d_fft_filter->d_fftsize);

        for (k=0;k<d_fft_filter->d_fftsize;k++) {
              c[k] = a[k] * b[k];
        }


 //            memcpy(d_invfft->get_inbuf(),(void *)c,d_fft_filter-
>d_fftsize*dataSize);
        queue->enqueueWriteBuffer(*aBuffer,CL_TRUE,0,d_fft_filter-
>d_fftsize*dataSize,(void *)ifftBuff);

        // Run the inverse FFT
```

```
        //  d_invfft->execute(); // compute inv xform
        err = clfftEnqueueTransform(planHandle, CLFFT_BACKWARD, 1, &(*queue)(),
0, NULL, NULL, &(*aBuffer)(), &(*cBuffer)(), NULL);
        err = clFinish((*queue)());

        // outdata = (gr_complex *)d_invfft->get_outbuf();
        queue->enqueueReadBuffer(*cBuffer,CL_TRUE,0,d_fft_filter-
>d_fftsize*dataSize,(void *)tmpFFTBuff);
        gr_complex *outdata;
        outdata=(gr_complex *)tmpFFTBuff;

        // ----------------------------------------------------------------
        // Unmodified GNURadio flow
        // add in the overlapping tail
        for(j = 0; j < d_fft_filter->tailsize(); j++)
             outdata[j] += d_fft_filter->d_tail[j];

        // copy d_fft_filter->d_nsamples to output buffer and increment for
decimation!
        j = dec_ctr;
        while(j < d_fft_filter->d_nsamples) {
             *output++ = outdata[j];
             j += decimation();
        }
        dec_ctr = (j - d_fft_filter->d_nsamples);

        // ----------------------------------------------------------------
        // stash the tail
        // memcpy(&d_tail[0], outdata + d_fft_filter->d_nsamples,tailsize() *
sizeof(gr_complex));
        memcpy(&d_fft_filter->d_tail[0], outdata + d_fft_filter-
>d_nsamples,d_fft_filter->tailsize() * dataSize);
      }
```

Before presenting the data, there is another dimension to filter testing. FFT filters require specific block sizes to match up with FFT bins. Therefore as the number of taps changes, so does this block size. The test tool test-clfilter takes this into account, however what it means is that the data below was run for time domain samples with the number of samples specified. Whereas the FFT transforms were run with their necessary block size which may be higher or lower. Test-clfilter does tell you the block size used for each run in its output for further analysis. This is important in the context of OpenCL and the number of bytes transferred to hardware. Because more bytes can mean higher throughput, one can see some variations in the FFT OpenCL transforms that at first glance may not make sense. However if one considers that the block sizes changed for the FFT transforms, it makes sense.

Data



Throughput

Taps	VM OCL FIR	VM CPU FIR	VM OCL FFT	VM CPU FFT
100 Taps	22,885,616.00	21,404,476.00	1,075,628.00	161,624,064.00
300 Taps	12,312,318.00	7,563,402.00	2,938,967.75	151,332,832.00
1000 Taps	4,089,839.75	1,325,223.12	3,470,441.00	98,806,104.00
2000 Taps	2,051,149.62	362,435.88	2,928,042.25	80,466,240.00
4000 Taps	1,065,685.25	97,455.12	4,383,777.00	62,140,540.00
8,000 Taps	545,208.06	433,145.47	6,661,933.50	54,585,396.00

Taps	1070 OCL FIR	1070 CPU FIR	1070 OCL FFT	1070 CPU FFT
100 Taps	76,546,064.00	18,979,310.00	2,743,087.75	224,000,912.00
300 Taps	66,045,096.00	12,235,497.00	3,736,149.50	238,450,976.00
1000 Taps	75,889,568.00	2,359,690.00	8,852,358.00	160,946,704.00
2000 Taps	35,378,960.00	1,693,756.00	11,923,301.00	140,503,680.00
4000 Taps	26,616,328.00	209,474.69	20,767,786.00	98,766,816.00
8,000 Taps	16,599,522.00	204,850.78	26,284,392.00	89,369,280.00

Taps	970 OCL FIR	970 CPU FIR	970 OCL FFT	970 CPU FFT
100 Taps	83,844,648.00	26,843,200.00	2,084,594.38	155,215,264.00
300 Taps	71,128,776.00	10,141,627.00	6,673,140.50	148,326,640.00
1000 Taps	46,703,180.00	808,101.50	7,343,295.50	104,844,624.00
2000 Taps	30,493,560.00	1,790,411.00	8,603,805.00	89,385,872.00
4000 Taps	18,013,690.00	375,857.19	14,012,486.00	64,487,392.00
8000 Taps	11,443,021.00	26,349.91	16,937,466.00	63,031,420.00

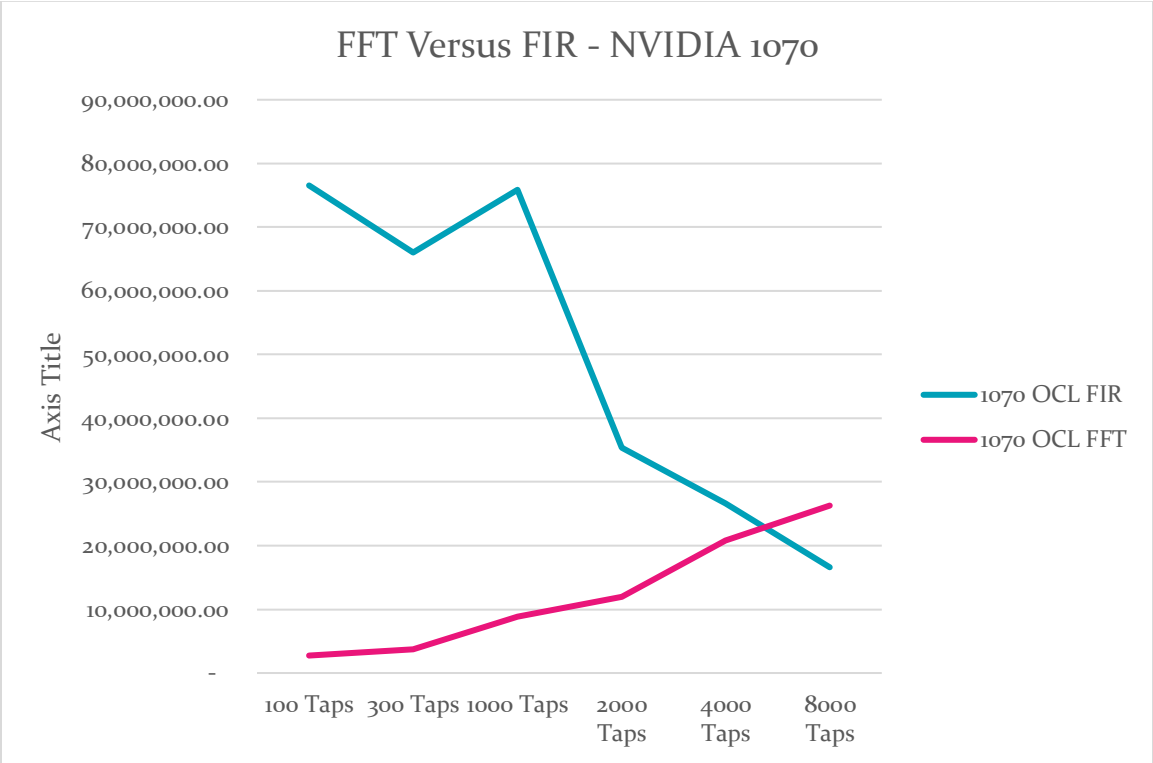
Taps	1000M OCL FIR	1000M CPU FIR	1000M OCL FFT	1000M CPU FFT
100 Taps	36,492,792.00	18,272,636.00	1,098,577.50	104,363,320.00
300 Taps	16,995,168.00	8,430,986.00	3,586,630.50	98,818,272.00
1000 Taps	5,948,145.50	1,122,462.25	4,286,220.50	63,277,876.00
2000 Taps	3,082,568.50	379,239.84	5,294,612.50	53,444,016.00
4000 Taps	1,555,993.25	91,996.01	8,386,413.50	40,870,868.00
8000 Taps	766,855.38	43,131.77	9,624,576.00	37,386,076.00

Observations

As is probably expected, an entire report could be done on filters. In general, the CPU-based FFT filters showed an order of magnitude better performance over the other filters. And as expected, the OpenCL implementation of the FIR filters outperformed their CPU equivalents. However the OpenCL-based FFT performance continued to improve as the number of taps increased, indicating that for very large tap sets the OpenCL FFT filter may be the best.

There was also decreasing performance with FIR filters as tap count increased countered by increasing FFT speed. This again is as expected in that the number of loops in time-based convolution increase with tap size taking more calculations per point. This reaches a tradeoff point in number of taps where the FIR filter is faster up to an equilibrium tap

size, then after which the FFT is faster. The graph below shows this tradeoff for the NVIDIA 1070.



In general the best recommendation is to treat taps as a design point in your flowgraphs. Take a look at the filter parameters and determine the number of taps in the specific filter(s) you are using and use the test-clfilter tool to profile each of the 4 filters to select the one with the best performance.

Costas Loop

In the original study, a Costas Loop was not included. This was due to the nature of the algorithm with sequential calculations that do not lend themselves to massively parallel processing. However after this study was completed, a follow-up project (gr-lfast) was started to look at ways to optimize CPU-based blocks for some additional performance improvements. Since a 50-70% speed gain was achieved in that project on the Costas Loop, the code was at least considered and translated in its optimized form into OpenCL to see how it would perform.

In order to implement this block in OpenCL, the method of OpenCL processing had to be different. Rather than parallel data processing, an OpenCL task-based processing approach was taken with a work queue size of 1 and all of the data passed as a single-dimensional input array. The result is that the algorithm runs as a single thread on a single core.

Performance was as expected (poor) on any of the GPU's. The data below shows the timing on the NVIDIA 1070:

```
Testing Costas Loop performance with 8192 items...

OpenCL Context: GPU

OpenCL Run Time:      0.011528 s  (710622.437500 sps)

CPU-only Run Time:      0.000462 s  (17721346.000000 sps)
```

One can see that it only achieved about 710,622 samples/sec. Not enough to even process in realtime at low sampling rates.

Because the code runs as a single task on a single core, increasing the data set size does not impact the overall throughput. In other words the throughput rate stays flat similar to when running on the CPU. This can be seen when moving to a 16384 block size:

```
Testing Costas Loop performance with 16384 items...

OpenCL Context: GPU

OpenCL Run Time:      0.023313 s  (702798.437500 sps)

CPU-only Run Time:      0.000922 s  (17774404.000000 sps)
```

As seen above, the throughput stays right around that 700,000 samples/sec range. This result agrees with expectations in running on a single core.

One rationale behind ultimately including this in this study was that some FPGA's also support OpenCL and the gr-clenabled code includes the ability to run on this OpenCL-enabled hardware as long as it appears as a standard OpenCL Accelerator device. It is possible that performance could be different on these alternate platforms, so the block was ultimately included.

A NOTE ON FRAMES AND SAMPLES / SEC

There are a number of notes of caution in actually using OpenCL accelerated blocks. In some cases, the size of the data blocks processed may need to be at a minimum of 8192, 10240, or larger to achieve OpenCL performance gains. This means making the GNURadio buffers at least twice that so that the scheduler would actually pass that much data to the block.

The GNURadio documentation cautions against adjusting these values unless you know specifically what you are doing and how it will impact the flowgraph. This section is

strictly conjecture (yet to be tested), however it stands to reason that the real value in OpenCL block usage comes from SDR hardware at higher data rates (in other words cards sampling at 20 MSPS or higher).

In real-time processing there is a concept of frames and frames per second. In this case a frame can be thought of as a single pass on a block of data provided to a block. If you assume a linear progression as is generally observed with CPU-only blocks, doubling the block size doubles the processing time but halves the frame rate. This can be best understood with an example:

If it takes .01 seconds to process a block of data, this translates to 100 frames per second (1/0.01). Multiply that times the size of the block and that would give you your throughput. So if 8192 data points are received, 8192 samples / frame x 100 frames/sec = 819,200 samples / sec. In many implementations both the throughput and frame rate are important. For instance, the human eye may require 20-30 frames/sec in a video stream to see smooth motion. If you decrease this to 5 frames/sec the video gets very choppy. The same concept can apply for real-time signal processing in that if we want to process our data in real time, we have to not only be aware of throughput but also frame rate. In other words if we made our block size so big that we were only processing 1 frame / sec, our flowgraphs and our output would "look" choppy just like the video example.

If you take a slower SDR producing 2.4 MSPS, increasing the buffer size per frame may make the signal choppy as in the video example. Moving to a 16K buffer over an 8K buffer would generally cut the frame rate in half.

However, it stands to reason that if you go from a 2.4 MSPS data stream on an rtl-sdr dongle to a 20 MSPS data stream on a HackRF you are increasing the required throughput per second by 8.3 times. Or on an Airspy running at 10 MSPS the throughput would be 4.17 times the 2.4 MSPS rate.

This implies that given the same buffer size, GNURadio needs to process more frames / sec to maintain throughput. In a real-world example, if the default 8192 buffer setting is used, lets assume the scheduler sends us half that every frame (it's not that linear but just for argument's sake and to make the math easy let's use this). That means that for 2.4 MSPS at 4096 samples, GNURadio would process about 586 frames / sec. For 20 MSPS, this increases to 4883 frames / sec!

In this 20 MSPS case, it may be okay to increase the buffer size comparably which would produce the same frame rate. In other words 20 MSPS with 8192 samples/frame * 8.3 factor (20/2.4) equals a buffer setting of 67164 to produce the same frame rate. Using this setting would produce the same frames / sec at 20 MSPS as 8192 at 2.4 MSPS but pass more data per frame which would be more conducive to OpenCL acceleration. Suddenly adjusting the buffer to 32K or 64K to get OpenCL block sizes above 10K doesn't seem unrealistic.

Therefore, while speculation, it stands to reason that increasing buffer sizes to 32K for 10 MSPS or 64K for 20 MSPS. Because we simply took the default setting of 8192 to get these numbers, remember that the scheduler would generally send about half that, which means that for 10 MSPS, a 32K buffer setting would generally produce 16K frame sizes. For 20 MSPS and 64K buffer sizes this means generally that blocks may be around 32K samples / frame. These line up much better with the performance curves where OpenCL outperforms the CPU.

Based on this line of thought, it stands to reason that one could use larger buffer sizes without impacting GNURadio flowgraphs and gain the benefit of OpenCL with larger block sizes, but only at higher SDR sample rates. This actually stands to reason in that if your goal is to process higher throughput through a flowgraph, it is because you are feeding it more samples per second. Otherwise at low sample rates the CPU-only performance for 2.4 MSPS is sufficient and wouldn't even require acceleration.

MULTIPLE SIMULTANEOUS BLOCKS

While one may think that using as many OpenCL-enabled blocks in a single flowgraph as possible would improve performance, running multiple kernels on a single graphics card does impact performance. A moment of reflection and this makes intuitive sense. Running a CPU-intensive application along with another CPU-intensive application will not only effect each but the system as a whole as they will compete to take full advantage of the CPU.

The same applies to GPU's. Performance benefits are calculated on an individual isolated basis taking full advantage of GPU hardware. Attempting to run multiple blocks simultaneously, each attempting to take full advantage of a GPU will inevitably effect each other. The following tests with test-clenabled demonstrate this effect.

This result is for the no-action kernel running with an 8192 block size and nothing else running on the 1070 hardware. The result is 36 microsecond runtimes.

```
Testing no-action kernel (return only) constant operation to measure
OpenCL overhead
This value represent the 'floor' on the selected platform.  Any CPU
operations have to be slower than this to even be worthy of OpenCL
consideration unless you're just looking to offload.
OpenCL INFO: Math Op Const building kernel with __constant params...
Max constant items: 8192
OpenCL INFO: Math Op Const building kernel with __constant params...
OpenCL Context: GPU
OpenCL Run Time:      0.000036 s  (225498704.000000 sps)
```

Running test-clenabled in a continuous while loop to load the GPU like this:

```
while true; do test-clenabled 128000; done
```

Then running test-clenabled in a separate command-prompt immediately shows the impact of the other running calculations:

Testing no-action kernel (return only) constant operation to measure OpenCL overhead

This value represent the 'floor' on the selected platform. Any CPU operations have to be slower than this to even be worthy of OpenCL consideration unless you're just looking to offload.

```
OpenCL INFO: Math Op Const building kernel with __constant params...
Max constant items: 8192
OpenCL INFO: Math Op Const building kernel with __constant params...
OpenCL Context: GPU
OpenCL Run Time:      0.000116 s  (70560704.000000 sps)
```

The result was an increased runtime to 116 microseconds… over 3.2 times slower. And this only represents the impact of one other block running at any given time. Running additional blocks could only have additional negative impacts on performance.

Tests were then run on the 970 hardware which actually contained 2 GTX 970 cards. The baseline no-action kernel test for 8192 data points was 73 microseconds shown below.

```
Testing no-action kernel (return only) constant operation to measure OpenCL
overhead
This value represent the 'floor' on the selected platform.  Any CPU operations
have to be slower than this to even be worthy of OpenCL consideration unless
you're just looking to offload.
OpenCL INFO: Math Op Const building kernel with __constant params...
Max constant items: 8192
OpenCL INFO: Math Op Const building kernel with __constant params...
OpenCL Context: GPU
OpenCL Run Time:      0.000073 s  (111518528.000000 sps)
```

The same while loop was executed using the second graphics card with the following line:

```
while true; do test-clenabled --device=0:1 128000;done
```

test-clenabled was then simultaneously run on the first graphics card to test the impact of running 2 blocks on 2 different cards. Then net result was a minimal impact as shown below:

```
Testing no-action kernel (return only) constant operation to measure
OpenCL overhead
This value represent the 'floor' on the selected platform.  Any CPU
operations have to be slower than this to even be worthy of OpenCL
consideration unless you're just looking to offload.
OpenCL INFO: Math Op Const building kernel with __constant params...
Max constant items: 8192
OpenCL INFO: Math Op Const building kernel with __constant params...
OpenCL Context: GPU
OpenCL Run Time:      0.000077 s  (105756552.000000 sps)
```

Note the increase from 73 to 77 microseconds. Multiple runs do vary by a few microseconds so this can be within within normal performance variances, however it stands to reason that other shared resources within the computer may contribute to some delay while both are running simultaneously.

In either case these tests demonstrate that using multiple OpenCL blocks on multiple cards can maintain the performance gains and allow multiple blocks to be used simultaneously.

CLOCK RECOVERY

MM Clock Recovery was another digital data block that was up for OpenCL implementation consideration. In reviewing the GNURadio code, the block appeared to have sequential calculations in that each successive iteration requires the previous iteration's calculation like the Costas Loop which does not lend itself to the massively parallel architecture of OpenCL processing. In addition, some other timing tests on the native block showed very high throughput rates, making it not worth OpenCL conversion at this time.

The sequential nature of the code can be seen in the code below taken from the GNURadio clock recovery general_work function:

```
      if(write_foptr) {
      while(oo < noutput_items && ii < ni) {
        d_p_2T = d_p_1T;
        d_p_1T = d_p_0T;
        d_p_0T = d_interp->interpolate(&in[ii], d_mu);

        d_c_2T = d_c_1T;
        d_c_1T = d_c_0T;
        d_c_0T = slicer_0deg(d_p_0T);

        x = (d_c_0T - d_c_2T) * conj(d_p_1T);
        y = (d_p_0T - d_p_2T) * conj(d_c_1T);
        u = y - x;
        mm_val = u.real();
        out[oo++] = d_p_0T;

        // limit mm_val
        mm_val = gr::branchless_clip(mm_val,1.0);
        d_omega = d_omega + d_gain_omega * mm_val;
        d_omega = d_omega_mid + gr::branchless_clip(d_omega-
d_omega_mid, d_omega_lim);

        d_mu = d_mu + d_omega + d_gain_mu * mm_val;
        ii += (int)floor(d_mu);
        d_mu -= floor(d_mu);

        // write the error signal to the second output
        foptr[oo-1] = mm_val;
```

```
        if(ii < 0) // clamp it.  This should only happen with bogus
input
          ii = 0;
      }
      }
```

If any readers have OpenCL implementations for these remaining blocks they could be worth timing and incorporating into gr-clenabled.

INSTRUMENTATION AND GR-FOSPHOR

This study specifically did not attempt to implement any instrumentation as the GR-FOSPHOR block already exists to provide an OpenCL implementation. Also, by the time the study got to the point to consider instrumentation, the poor performance of FFT transforms in OpenCL for real-time processing were understood and were speculated to show up in instrumentation like a Frequency Sink as well.

However the study did take a cursory look at the performance of gr-fosphor with the hindsight of the results of this study. What was observed was that gr-fosphor has a significant negative impact on a system. Tests were performed with a flowgraph with a constant source feeding a 2MSPS throttle block into gr-fosphor on the 1070 system. The net result was that 2 of the CPU cores still went from near zero to 100% utilization. This may be unacceptable if other high-CPU blocks are in a flowgraph.

For comparison, the same test was then performed with the standard QT Frequency sink on the 1070 system and the CPU usage was considerably lower. CPU's showed only nominal activity, low by any estimation.

This indicates that CPU-based visualization is actually more efficient and the use of the GPU for visualization, at least based on FFT transforms, is less efficient and would actually put a higher load on the system. This makes sense in the context of the results of this study where FFT transforms, fundamental to frequency displays perform better on the CPU than on the GPU.

If the overall goal of a flowgraph with OpenCL blocks is to process higher throughput or relieve some of the CPU-intensive calculations, one alternative in providing visualization with frequency, time, and/or phase plots to consider if this is the case is to use a TCP or UDP source/sink combination to send the data to a second computer dedicated to visualization. That way signal instrumentation would not impact processing of the same signal.

Conclusions

Software Defined Radio provides a great opportunity for researchers and hobbyists to engage in a field of study that has historically been too costly. However as technologies emerge, more and more bandwidth and throughput are required. Generalized CPU's continue to evolve and as seen in this study on the newer i7 processor in the 1070 hardware, throughput continues to improve as CPUs become faster.

However as computers provide the platform to enable SDR, all hardware at the radio designer's disposal should be able to be used. This includes graphics cards with powerful GPU's, and FPGA cards for higher-end applications. GNURadio is arguably one of the most utilized open source SDR platforms available. Therefore in the process of evolution it becomes a next logical step to evaluate which types of digital signal processing blocks could benefit from CPU offloading through OpenCL. This should not just be limited to a single card or a single type of card within a computer, but should be scalable to allow all cards to be utilized with the discretion of the designer to determine which blocks run on which cards.

This is the goal of the gr-clenabled project:

- ✓ The ability to use OpenCL for the most common GNURadio blocks used in digital data processing
- ✓ The ability to take full advantage of all OpenCL-capable hardware simultaneously
- ✓ The ability to have the flowgraph designer determine which blocks run on which cards
- ✓ Develop a solid understanding of which blocks can benefit from OpenCL, and by design which blocks will not

With the software developed during this study and the data collected, several important conclusions were reached. Some of these conclusions were obvious and simply confirmed assumptions, while others were quite revealing. First, not all blocks implemented in OpenCL running on GPU hardware demonstrate acceleration. In fact blocks could be categorized in 3 categories: 1. Those that provide acceleration, 2. Those that provide offloading or mixed results based on hardware and/or block size, and 3. Those implemented in OpenCL but exhibiting performance worse than their CPU version.

Of the blocks implemented in this project the following blocks showed acceleration when executed in OpenCL:

1. Log10
2. Complex To Arg
3. Complex To Mag/Phase
4. A custom Signal To Noise Ratio Helper that executes a divide->Log10->Abs sequence

The following blocks showed mixed or offload performance:

1. Mag/Phase To Complex (OpenCL performed better only for blocks above 8K for the 1070, and 18K for the 970 and 1000M)
2. Signal Source (OpenCL outperformed CPU only for the 1070 for 8K blocks and above)
3. Quadrature Demodulation (OpenCL performed better only for blocks above 10K)
4. FIR Filters

The remaining blocks tested showed worse throughput in OpenCL implementations. These blocks were:

1. Multiply
2. Add
3. Subtract
4. Complex Conjugate
5. Multiply Conjugate
6. Multiply Constant
7. Add Constant
8. Complex to Mag
9. Forward FFT
10. Reverse FFT
11. FFT Filters

Instrumentation is also a very important part of most flowgraphs. GR-Fosphor has been around for some time and actually uses a fifo and a separate application along with OpenGL and OpenCL to offload FFT transforms along with graphics rendering. However testing gr-fosphor with a constant source and a 2 MSPS throttle into both a standard QT Frequency Sink and a gr-fosphor sink on the 1070 hardware clearly demonstrated that using OpenGL and OpenCL for visualization puts an even higher load on a CPU than GPU offloading. This is actually inline with the findings of this study. Since FFT transforms are fundamental to frequency displays and it has been demonstrated here that FFT transforms can perform worse than CPU implementations for SDR block sizes, poorer performance from gr-fosphor should be expected.

The real benefit of OpenCL acceleration will come from higher sampling rates where increased buffer sizes can be used without negatively effecting frame rates. This can be important in realizing OpenCL acceleration in that some blocks only outperform their CPU counterparts when provided sufficiently large block sizes. This makes intuitive sense in that OpenCL acceleration may be most beneficial at higher throughput rates where the load on the CPU may increase to unacceptable levels. This should be kept in mind along with how GNURadio and its scheduler handle maximum and operational block sizes where the scheduler will [generally] attempt to send about half the maximum buffer size to a block during each frame. This default buffer size is 8192, so the default block size that

could be expected without making any changes would be 4096. This may not be sufficient to achieve OpenCL acceleration for some blocks.

Overall this study proved very enlightening. With the exception of improving PSK and MM Clock Recovery processing which had sequential calculations not conducive to OpenCL parallel processing, the study met all of its goals of providing implementations of most common blocks that could run on user-selectable OpenCL devices and studied the process from signal source through visualization for both ASK and FSK signals. However the study also demonstrated that not all blocks experience better performance on OpenCL hardware versus their CPU-only counterparts for a variety of reasons.

One final conclusion could be reached on appropriate use of OpenCL blocks. The greatest benefit would be derived from identifying all blocks used in your flowgraph that are in the OpenCL accelerated list. From this short list, start with the block that provides the most benefit and use that block on your OpenCL hardware. If you have multiple cards repeat this process with one block for each card. If you have multiple versions of the same card, selection will not matter. However if you have different cards of varying performance you may want to assign the most computationally intensive blocks to the best-performing hardware. Test-clenabled can be used to determine actual throughput for the block in question.