Higher Throughput: GNURadio Let me Re-Introduce you to GPU's

By ghostop14

April, 2017

About 4 months ago I decided to take on a project that I had wished existed for some time. With all of the code available for using graphics cards for signal processing why were there not a wealth of GPU-accelerated blocks for GNURadio? Really leveraging my new graphics card (an NVIDIA GTX 1070), couldn't I drive 80 MSPS or higher through if I had hardware that could supply it? (I know USB 2.0 bus speeds, some decoders require hardware for speed, etc. but an SDR enthusiast can still dream)

My idea seemed simple enough. Why not develop OpenCL versions of the most common blocks used in digital data processing? I may not hit my throughput goal but I bet I can really accelerate my flowgraphs. And since I can dream up whatever I want before I have to actually make it, why not make it even more scalable? Why not be able to take full advantage of multiple graphics cards in a system by being able to assign different blocks to run on different cards?

I know, that's a lot of questions, but sounds great if it existed right? What I didn't realize was the scope of the box I was about to open. My first task at hand was to learn OpenCL and REALLY dig into the depths of the GNURadio code. Turns out not all signal processing algorithms lend themselves nicely to the way massively parallel processing works. And there's a time price to pay to move data to a PCI card for processing then retrieve the results that has to be considered. Some native blocks take longer than this transfer time to run and can benefit from offloading, while others are so fast they're done before a GPU even gets the data. But I'm getting ahead of myself here.

Back to the project…. so to put my curiosity (and anyone else's) to rest I took on the daunting task of implementing these blocks in OpenCL:

1. Basic Building Blocks
    a. Signal Source
    b. Multiply
    c. Add
    d. Subtract
    e. Multiply Constant
    f. Add Constant
    g. Filters
        i. Low Pass
        ii. High Pass
        iii. Band Pass
        iv. Band Reject
        v. Root-Raised Cosine
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)

I figured with all of those blocks I should really be able to move some data through a flowgraph. I have access to multiple systems with different levels of NVIDIA graphics cards from a laptop with an NVIDIA 1000M to a slightly older GTX 970 (2 in the same system in fact) to a new 1070, so I could really do some old-versus-new comparisons and work out any issues with different cards during the project as well.

So I started developing the code one module at a time and as I started writing and timing the OpenCL code, I was starting to question if all of these OpenCL blocks would actually perform better than the CPU, and if I would still have overall acceleration if I tried to use multiple OpenCL blocks in the same flowgraph. Now not only did I have a development project on my hands but this idea was turning into a full-fledged study! Not what I was expecting. I could now see why nobody else had tackled this before. But I'm stubborn so I decided to push on.

Time to get serious: a framework had to be built to be able to get timing information on the original GNURadio modules outside of the GNURadio scheduling flowgraph and the new OpenCL modules. I had to be able to accurately time the execution of both sets of code and know that nothing else like the scheduling engine logic was impacting the results. Then I could try to group the blocks into one of 3 categories: 1. Those that ran faster on the GPU (classified as accelerated), 2. Those that ran about the same on the GPU or had mixed performance based on different graphics cards and/or block sizes (classified as offloading), and 3. Those that ran slower (classified as enabled). So I built some command-line tools into the project to do just that. I could feed it different block sizes and call the working code directly along with timers and averaging loops to get an idea of how long each routine took to run. From the results this last group actually had two subcategories: those that still ran fast enough for most SDR implementations even though they were slower than their CPU counterparts, and those that ran so slow I wouldn't bother but let's implement them because they seemed intuitively obvious and this study would put the question of their applicability to rest.

If you're already wondering how it turned out, the full project and code is up on github at https://github.com/ghostop14/gr-clenabled.git for everyone to enjoy. And a full methodical study along with the data can be found in the docs folder at https://github.com/ghostop14/gr-clenabled/blob/master/docs/Study%20on%20Implementing%20OpenCL%20in%20GNURadio.pdf.

The cliff notes summary is that there were a number of blocks that relied on trigonometric and log functions that really benefitted from OpenCL. However some blocks that seemed obvious for OpenCL implementation (FFTs and filters specifically) actually performed horribly. Not because they can't be implemented in OpenCL efficiently, but because of the real-time nature of SDR and the relatively small block sizes we work with in relation to what could be done in offline modes. Because we have to process data in "real-time" we can't take huge blocks of data and process them all at once; something

GPU's can excel at. Instead we take much smaller blocks to keep the data moving. This is where the tradeoff between the time it takes us to move the data to the card and back can mean the difference between a block performing better in OpenCL or on the native CPU.

There was in fact a price to pay for running multiple blocks on the same card in the same flowgraph as well. To address this there are a few workable alternatives: 1. Run different blocks on different OpenCL devices (glad I built in the ability to run different blocks on different cards), 2. Use the custom kernel blocks to write a single kernel that aggregates a number of functions together, or 3. Write a new class based on the framework that can take advantage of re-using buffers from one kernel call to another (again aggregating multiple functions into a single block).

There were a few remaining blocks used in digital data processing that were not implemented in the final product, specifically MM Clock Recovery, Costas Loops, and PSK demodulation. I wanted to include them, however as I learned throughout this project these blocks use processing algorithms that have sequential calculations (the value for one output point depends on the previous one). Because of this they don't implement well in a massively parallel architecture where all of the calculations want to happen *independently and simultaneously*. However if folks have ideas I didn't think of, I've built in room to grow and experiment in the project. There are two generic extensible blocks for 1 input-to-1 output, and 2 inputs-to-1 output that can be passed an OpenCL kernel as a block parameter so folks can make their own kernels. And all of the timing tools I used along with one to allow the same type of timing analysis on generic blocks is included in the project (see the study doc for tool details).

Overall I think the project was a pretty big success. Of the blocks implemented. these 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

These 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)

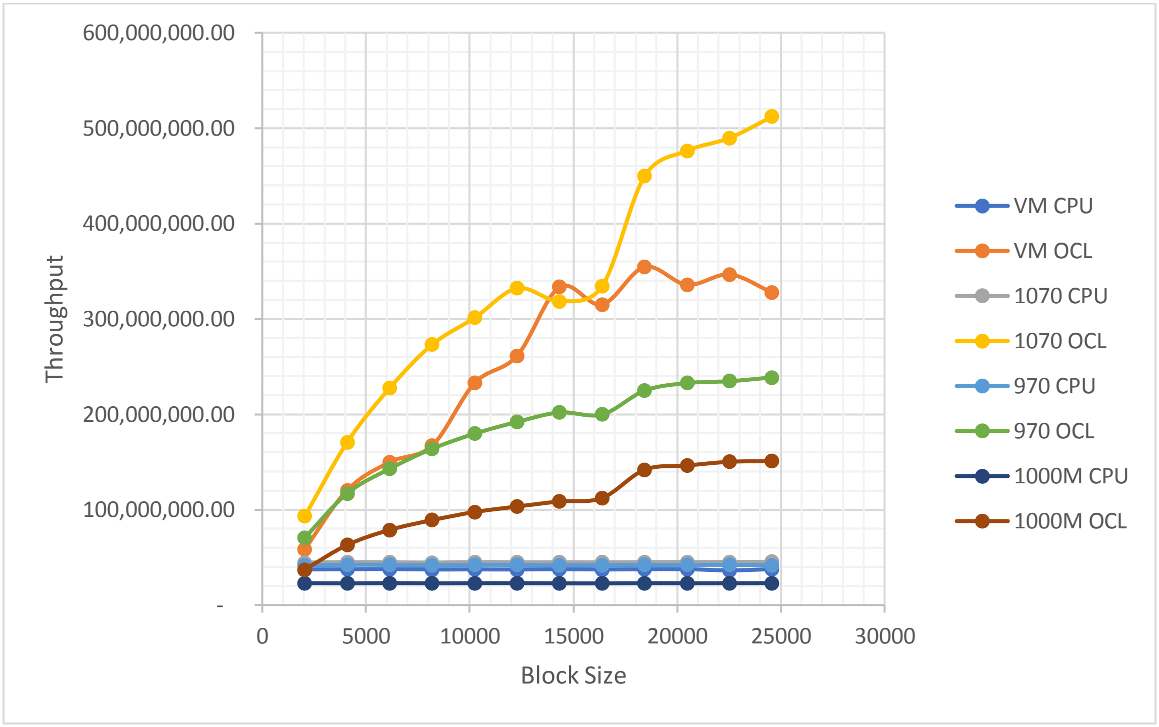
The remaining blocks tested showed worse throughput in OpenCL implementations. These blocks were:

1. Multiply
2. Add
3. Subtract
4. Complex Conjugate
5. Multiply Conjugate
6. Multiply Constant
7. Add Constant
8. Complex to Mag

And these blocks performed incredibly poorly with the block sizes used in real-time processing. (The why's identified during testing are all discussed in the study paper).

1. Forward FFT
2. Reverse FFT
3. Filters

The chart below for the log10 block shows a quick sample of what's in the full study:



Wrapping up this phase of the project is only the beginning, not the end. In fact it's already spawned a follow-up project called gr-lfast (also up on github) to try to focus on those other digital processing blocks that didn't translate well to OpenCL and look for ways to optimize the CPU code. The Clock Recovery block had great CPU throughput on its own, while the Costas Loop has been improved by 54% for a 2nd order loop in the gr-lfast module from 24 MSPS max throughput to 37 MSPS on my test platform.

There are a number of new blocks, both CPU-based and OpenCL in the works to continue to push the performance envelope, so keep an eye out for more blocks in the future!