A Standalone Package for Bringing Graphics Processor Acceleration to GNU Radio: GRGPU

William Plishker
University of Maryland
plishker@umd.edu
Outline

• Introduction

• GPU Background

• Graphics Processor Integration with GNU Radio

• Evaluation

• Summary
Motivation

• High-Performance prototyping with GPUs is tempting…
  – Potentially fast design times
  – Flop/$

• …but hard.
  – Learn a new language
  – Expose or refactor parallelism
The GPU Difference

Detailed GPU

Device
- Multiprocessor N
- Multiprocessor 2
- Multiprocessor 1
  - Shared Memory
  - Registers (Processor 1, Processor 2, ..., Processor M)
  - Instruction Unit
  - Constant Cache
  - Texture Cache
  - Device Memory

Control
- ALU
- ALU
- ALU

Cache

DRAM

CPU

GPU

DRAM
A typical high end modern GPU: GTX 580

- Released 10/2010
- 3B transistors, 40nm
- 512 Cores @700MHz
- 1.5 TFLOPs
- 192 GB/s Memory BW
- 244W
- $500
CUDA FIR vs. Stock FIR
A brief history of GPU Programming

- Cg, GLSL, HLSL (early 2000s)
- Sh, Brook (2004)
- CUDA (2006)
- OpenCL (2008)
Related Work – MATLAB

```matlab
>> % Create arrays that reside on the GPU
>> A = gpuArray(rand(1000, 1000));
>> b = gpuArray(rand(1000, 1));

>> % Use GPU-enabled MATLAB functions
>> x_gpu = A \ b; % "\" is GPU-enabled

>> f_gpu = fft(A);

>> % Bring data back from GPU memory into MATLAB workspace
>> x = gather(x_gpu);
>> f = gather(f_gpu);
```

8/25
Related Work – PyCUDA

```python
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule('"
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}"
"

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1), grid=(1,1))

print dest-a*b
```
Background – GNU Radio Design Flow

```python
class top(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)
        ntaps = 256
        # Something vaguely like floating point ops
        self.flop = 2 * ntaps * options.npipelines
        src = gr.null_source(gr.sizeof_float)
        head = gr.head(gr.sizeof_float, int(options.nsamples))
        self.connect(src, head)

        for n in range(options.npipelines):
            self.connect(head, pipeline(options.nstages, ntaps))
```
Proposed Design Flow

- Start from an unambiguous application description based on the application area itself
  - Dataflow
- Use tools to optimize scheduling, assignment
- Generate an accelerated functional description
GNU Radio for Graphics Processing Units (GRGPU)

- C++ Block with call to GPU Kernel (.cc)
- GPU Kernel in C++ (.cc)
- CUDA Synthesizer (nvcc)
- libtool
- Stand alone python package

CUDA Libraries

GNU Radio

CUeda Libraries

libcuda
libcutil
Build Process

• make-cuda : .cu $\rightarrow$ .cc
  – nvcc as a software synthesizer
  – transform cuda code into target specific GPU code
  – wrap functions in extern “C”

• make
  – Resulting cc’s are just more inputs to existing build structure
  – link against CUDA runtime (cudart) and CUDA utils (cutil)
Using GRGPU

- Used just like gr in GNU Radio flowgraphs
  - import grgpu
  - op = grgpu.op_cuda()

- Dependent blocks
  - Host/device data movement handled by other blocks
  - Has device pointers as interface
class GPU_Thread(threading.Thread):
    """GPU Thread""
    def init(self):
        self.tb = gr.top_block()
        src_data = list(math.sin(x) for x in range(TOTAL_LENGTH))
        src = gr.vector_source_f(src_data)

        h2d = grgpu.h2d_cuda()
        taps = [0.1]*60
        op = grgpu.fir_filter_fff_cuda(taps)
        d2h = grgpu.d2h_cuda()

        self.dst = gr.vector_sink_f()
        self.tb.connect(src, h2d)
        self.tb.connect(h2d, op)
        self.tb.connect(op, d2h)
        self.tb.connect(d2h, self.dst)
GRGPU Example

source → H2D → FIR (GPU accel) → D2H → sink

Host: Data → GPU Ptrs → GPU Ptrs → Data

GPU: Data → Data
Other GRGPU features

- Fifo reads/writes are chunky and have history
  - Designed to make use of burst transfers
  - Creates in data contiguous memory for kernels

- Growing library
  - add_const, mul_const, FIR, resampler, FFT
  - floats, complexes

- Typical GR conveniences
  - Create new GRGPU block script
  - Doxygen, unit tests, GR build process
Evaluation

• Benchmark GPU options with GNU Radio multicore benchmark (mp-sched)
  – 2 Intel Xeons (3GHz) and 1 GTX 260

• GPU and GPP accelerated FIR
  – CUDA and SSE acceleration

• Vary: size and assignment
• Measure: total execution time
Evaluation Application

![Diagram showing the flow of evaluation application with stages and pipelines.](image-url)
Evaluation Application – “Half CPU”
Evaluation Application – “One GPP”
Evaluation Application – “One GPU”
Results

![Graph showing speed-up relative to All GPP solution for different configurations.](image)
Conclusions

- GRGPU provides GPU acceleration to GNU Radio as a stand alone library
  - Facilitates design space exploration

- Status
  - Planned to be contributed back
  - Few but growing number of CUDA accelerated blocks

- Ongoing Work
  - Automated Mapping
  - MultiGPU
Questions?

Citations:
