Inference as DSP: An Approach to Heterogeneous Compute-Enabled SDR Applications

Garrett Vanhoy, PhD (Peraton Labs)

Team members:
Joshorman, Troy Bates, Rob Taylor (Peraton Labs)
Ray Hoare, PhD, Claire Brevik (Concurrent EDA)

Work done thanks to:
DARPA TRIAD Program

Tensors for Reprogrammable Intelligent Array Demonstrations
Agenda

- Inference as DSP?
- Architecture
- Benchmarks/Results
- OOT Structure
Inference as DSP?
Why Inference as DSP?

- GNU Radio has a library of blocks implementing DSP on CPU, but what about other modern processors GPU, FPGA, etc?

- Ideally, as SDR application developers, we would like to have as much control about where and how DSP operations are run.
Why is that hard?

- Implementing many basic DSP operations on GPU are available
  - cuSignal, Eigen, PyTorch/Tensorflow, MatX, OpenCV, OpenCL, Intel oneAPI

- Libraries supporting FPGA-based DSP implementations:
  - Vitis Libraries, Ettus RFNoC, hls4ml

- Some questions:
  - What about when multiple devices are available?
  - How do you efficiently handle transfers between devices?
  - What is an appropriate model for the maintenance of a heterogeneous DSP library?
What’s our approach?

• Leverage Tensor-processing libraries such as PyTorch and Tensorflow to describe DSP operations
  • And also implement actual ML models too.

• Integrate GRC/GR with inference servers to leverage multiple devices and per-operation/block scheduling

• **The result:** write a “model” in Python, implement it on:
  • Multi-core CPU
  • GPU/Multi-GPU
  • FPGA/Multi-FPGA (DPU)
Architecture

Overall
Use GRC as a familiar front-end to develop blocks.
Generic GR “Model” block

GR blocks just call to the inference servers

```cpp
int triton::block::impl::work(
    int output_items,
    gr_vector const* & input_items,
    gr_vector void const* & output_items) {

    std::vector<const char*> in_ptrs;
    for (const auto & item : input_items)
        in_ptrs.push_back(static_cast<const char*>(item));

    std::vector<char*> out_ptrs;
    for (const auto & item : output_items)
        out_ptrs.push_back(static_cast<char*>(item));

    // num_items_per_batch is fixed.
    auto num_items_per_batch =
        model_get()->get_output_sizes()[0] / model_get()->get_output_signature()[0];
    auto batch_size = output_items / num_items_per_batch;
    model_->infer_batch(in_ptrs, out_ptrs, batch_size);

    return output_items;
}
```
Generic GR “Model” block

Each call is a batch of data

```cpp
int triton_block_impl::work(
    int output_items,
    gr_vector< const void* > & input_items,
    gr_vector< void* >& output_items
) {

    std::vector< const char* > in_ptrs;
    for (const auto & item : input_items)
        in_ptrs.push_back( static_cast< const char* >(item));

    std::vector< char* > out_ptrs;
    for (const auto & item : output_items)
        out_ptrs.push_back( static_cast< char* >(item));

    // num_items_per_batch is fixed.
    auto num_items_per_batch =
        model_.get()->get_output_sizes().size() / model_.get()->get_output_signature().size();
    auto batch_size = num_items_per_batch / num_items_per_batch;

    model_.infer_batch(in_ptrs, out_ptrs, batch_size);

    return noutput_items;
}
```
Architecture

Triton Inference Server:
Running multi-core CPU, GPU, and multi-GPU
TIS Internals

Here’s TIS’s overall architecture
Models are defined through a number of common frameworks
Client libraries are provided to issue Inference Requests
There is a supported way to make direct calls to inference rather than remote calls, to be more efficient.
Each “model” has inference requests has a scheduler.

The scheduler can be configured.
TIS Internals

```python
instance_group:
  count: 1
  kind: KIND_CPU

optimization:
  execution_accelerators:
  - name: "openvino"
```

Models are instantiated across a configurable number of devices.
Each block is defined in Python as a Pytorch Module.

class FFT(nn.Module):
    def __init__(self):
        super(FFT, self).__init__()

    def forward(self, x_real, x_imag):
        raw_iq = x_real + 1j*x_imag
        fft = torch.fft.fft(raw_iq, dim=1, norm="ortho")
        return torch.cat([fft.real, fft.imag], dim=1)
Architecture

AMD/Xilinx Inference Server and the Deep-Learning Processing Unit (DPU)
What is the DPU?

- AMD/Xilinx offers a Deep Learning Processor Unit (DPU) implemented in Programmable Logic
  - Effectively works as co-processor specialized for convolutional neural networks

- Supported on:
  - Zynq-7000 SoC and Zynq Ultrascale+ MPSoC

- DPU’s are a customizable IP Block
  - AMD/Xilinx provides baseline images with DPU access
  - Can have up to four DPU’s with different configurations
What’s Special about the DPU?

• Rapid Development/Prototyping
  • Place-and-route not required when to implement a new “model”

• Parallelism
  • Highly parallelized architecture for convolutions can be exploited for efficient DSP/ML

• Simplified Development Interface
  • Python → FPGA implementation
Working with the DPU

- Similar to CPU/GPU, a model is **defined in** PyTorch

- **Vitis AI Tools** creates xmodel files to run on DPU

- **Vitis-AI Runtime (VART)** provides Python interfaces to actuate DPU’s with provided data
DPU Implementation

Working with the DPU

- DPU employs three independent means of parallelism

- 2D convolutions can be used to implement many common DSP operations
  - And also ML layers
Some Caveats

- Fixed point vs Floating point
- Most of the current DPU configurations process 8-bit data, not 16-bit
Throughput Measurements

Triton Inference Server on CPU/GPU/multi-GPU
TIS Throughput Measurements

• Purpose
  • Expected maximum throughput if transfer overhead (double-copies, HTTP RTT) is eliminated

• Using *perf_analyzer* provided by TIS
  • Configurable: # threads, batch size
  • Fixed: transfer via shared memory, HTTP

• Hardware
  • Four (4) nVidia 4500 (20 GB)
  • AMD EPYC 7513 with 32 Cores
TIS Throughput: FFT

- GPU and CPU perform similar for smaller FFT
- Too large FFT and batch size result in overhead from TIS
TIS Throughput: Convolve

- GPU and CPU still perform similarly for smaller convolutions
- Multi-GPU is closer to linear in speed-up per GPU
TIS Throughput: Dot Product

- GPU outpaces CPU for larger batch sizes
- Multi-GPU outpaces single GPU for yet larger batch sizes
Throughput Measurements

The DPU (and eventually the AMD/Xilinx Server)
DPU Benchmarks

- **Purpose**
  - Expected maximum throughput if transfer overhead (double-copies, HTTP RTT) is eliminated

- **Hardware**
  - KRIA KV260
  - Single DPUCZDX8G in B4096 configuration running at 300 MHz

- **Using a simple Python program on the ARM processor**
  - AMD/Xilinx server to be eventually benchmarked
• Using a 2D convolution operation to represent an 8-element complex dot-product in 8-bit fixed-point format.
  • We “batch” in the input-channel, height, and width dimensions
  • Effective “batch” size is 32768

• This can represent an 8-element narrowband beamform:
  • ~60 MS/s throughput per-aperture
DPU Throughput: Real-Only Filter

- Using a 2D convolution operation to represent a
  - 1024-tap real filter operating on real data
  - 8-bit fixed-point format
  - decimating by a factor of 64

- We “batch” in the input-channel, height, and width dimensions
  - Effective “batch” size is 4096

- Measured throughput: ~31 MS/s
• Using a 2D convolution operation to represent a
  • 256-element complex FFT
  • 8-bit fixed-point format

• We “batch” in the height and width dimensions
  • Effective ”batch” is 256

• Measured throughput: ~300 MS/s
# GPU/DPU Benchmarks for ML

<table>
<thead>
<tr>
<th>Parameter</th>
<th>AlexNet</th>
<th>GoogLeNet</th>
<th>Inception3</th>
<th>MobileNet V2</th>
<th>Resnet101</th>
<th>Resnet152</th>
<th>Resnet34</th>
<th>Resnet50</th>
<th>VGG16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calibration time (s)</td>
<td>17.9</td>
<td>47.0</td>
<td>97.1</td>
<td>63.5</td>
<td>173.8</td>
<td>243.1</td>
<td>51.3</td>
<td>115.4</td>
<td>106.2</td>
</tr>
<tr>
<td>Top 1 Accuracy</td>
<td>47.3</td>
<td>85.6</td>
<td>69.5</td>
<td>36.3</td>
<td>46.9</td>
<td>80.2</td>
<td>81.7</td>
<td>44.9</td>
<td>39.8</td>
</tr>
<tr>
<td>Top 5 Accuracy</td>
<td>51.9</td>
<td>98.7</td>
<td>81.1</td>
<td>57.3</td>
<td>59.0</td>
<td>97.4</td>
<td>94.3</td>
<td>61.7</td>
<td>46.2</td>
</tr>
<tr>
<td>Loss</td>
<td>6.7</td>
<td>0.5</td>
<td>4.9</td>
<td>6.1</td>
<td>5.4</td>
<td>0.7</td>
<td>2.5</td>
<td>5.2</td>
<td>6.7</td>
</tr>
<tr>
<td>GPU Throughput (ms)</td>
<td>5.2</td>
<td>11.6</td>
<td>17.5</td>
<td>10.3</td>
<td>19.0</td>
<td>26.7</td>
<td>9.3</td>
<td>11.3</td>
<td>11.2</td>
</tr>
<tr>
<td>GPU Throughput FPS</td>
<td>190.8</td>
<td>86.4</td>
<td>57.1</td>
<td>97.6</td>
<td>52.5</td>
<td>37.5</td>
<td>107.3</td>
<td>88.2</td>
<td>89.6</td>
</tr>
<tr>
<td>GPU Latency (ms)</td>
<td>106.0</td>
<td>67.2</td>
<td>105.9</td>
<td>48.8</td>
<td>133.6</td>
<td>186.8</td>
<td>58.3</td>
<td>87.5</td>
<td>249.2</td>
</tr>
<tr>
<td>FPGA Latency (ms)</td>
<td>43.0</td>
<td>15.1</td>
<td>44.7</td>
<td>39.1</td>
<td>47.3</td>
<td>41.7</td>
<td>41.1</td>
<td>43.1</td>
<td>65.5</td>
</tr>
<tr>
<td>FPGA FPS</td>
<td>27.3</td>
<td>224.5</td>
<td>27.1</td>
<td>27.9</td>
<td>27.3</td>
<td>28.8</td>
<td>27.4</td>
<td>26.8</td>
<td>20.3</td>
</tr>
<tr>
<td>FPGA Throughput (ms)</td>
<td>36.7</td>
<td>4.5</td>
<td>36.9</td>
<td>35.8</td>
<td>36.7</td>
<td>34.7</td>
<td>36.6</td>
<td>37.3</td>
<td>49.2</td>
</tr>
<tr>
<td>FPGA Performance (GOP/s)</td>
<td>141.5</td>
<td>679.3</td>
<td>726.9</td>
<td>187.4</td>
<td>644.9</td>
<td>685.9</td>
<td>886.5</td>
<td>597.5</td>
<td>629.4</td>
</tr>
<tr>
<td>FPGA Speedup (Throughput)</td>
<td>0.143</td>
<td>2.600</td>
<td>0.474</td>
<td>0.286</td>
<td>0.519</td>
<td>0.769</td>
<td>0.255</td>
<td>0.304</td>
<td>0.227</td>
</tr>
<tr>
<td>Image Size</td>
<td>224x224</td>
<td>224x224</td>
<td>299x299</td>
<td>224x224</td>
<td>224x224</td>
<td>224x224</td>
<td>224x224</td>
<td>224x224</td>
<td>224x224</td>
</tr>
</tbody>
</table>

Batch Sizes - GPU Training : 10, GPU Quantization Calibration : 10, GPU Quantization Validation : 1, FPGA Quantization Validation : 1
GPU used: GTX 1080, DPU used: one B4096 with 4 Threads
OOT Structure
1. examples
   1. Beamforming flowgraph
   2. FFT flowgraph
   3. ML flowgraph

2. Models
   • Triton
     • models
   • dpu
     • models
     • architecture

3. Available blocks
   • FIR Filter
   • FFT
   • Beamform
Information on OOT

- Available at https://github.com/gvanhoy/gr-torchdsp