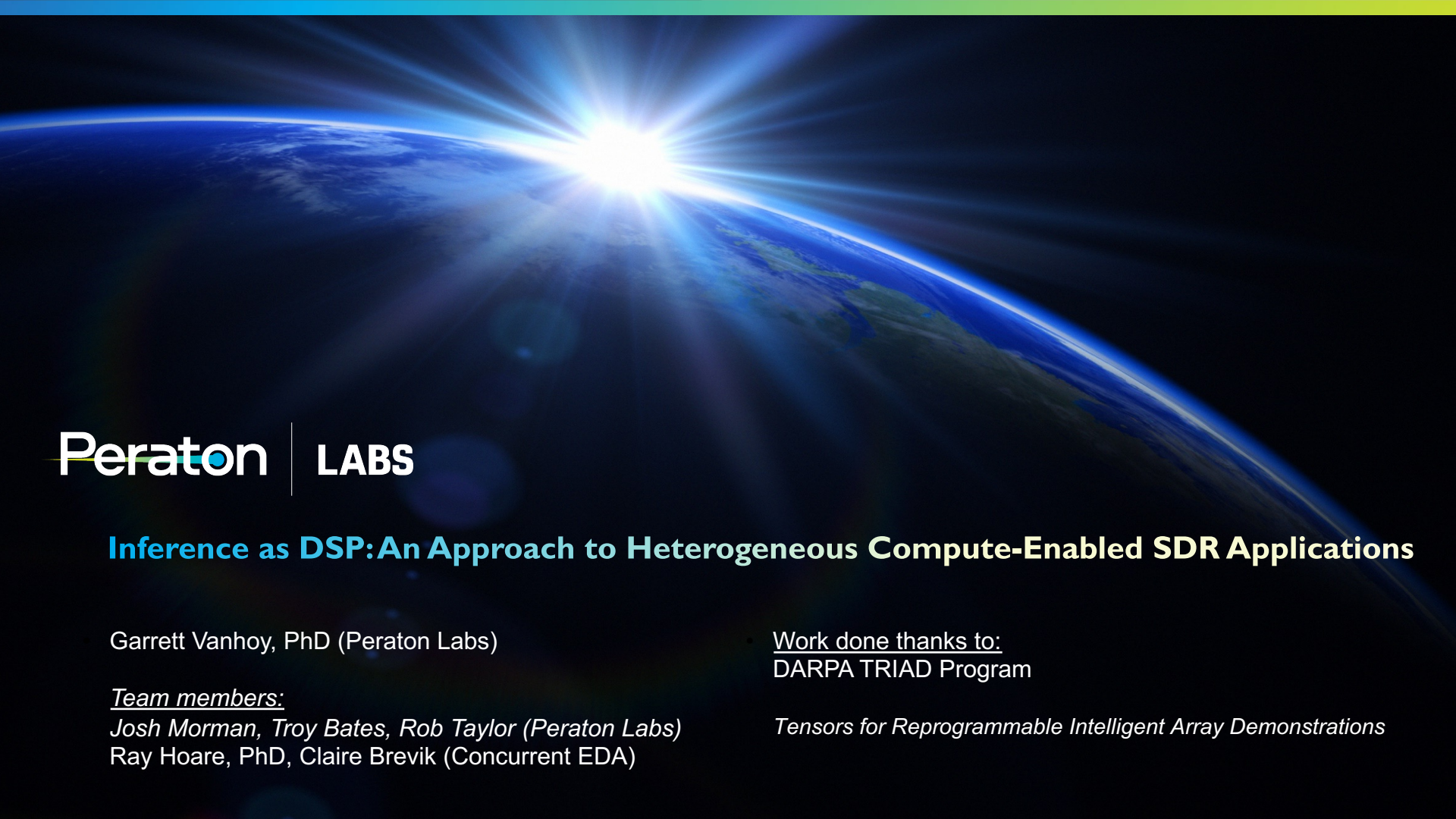


Inference as DSP: An Approach to Heterogeneous Compute-Enabled SDR Applications

Garrett Vanhoy, PhD (Peraton Labs)

Team members:
Josh Morman, 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

Peraton | LABS

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

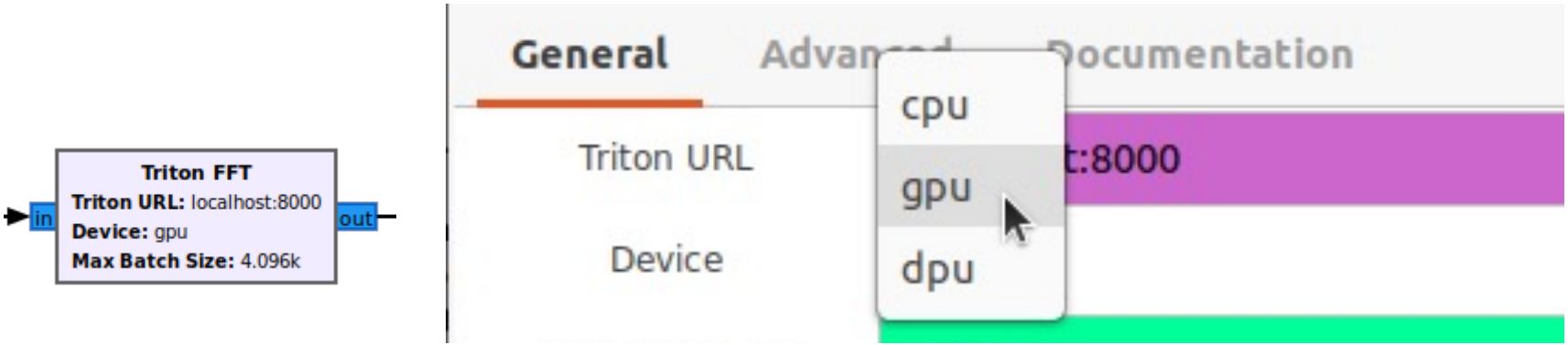
Peraton | LABS

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?

Peraton | LABS

- 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?

Peraton | LABS

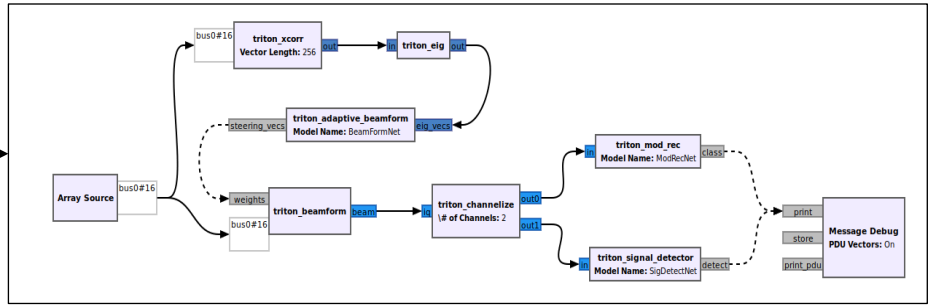
- Leverage Tensor-processing libraries such as PyTorch and Tensorflow to describe DSP operations
  - And also implement actual ML models too.
- Integrate GRC/GR with inference servers to leverage multiple devices and per-operation/block scheduling
- **The result:** write a "model" in Python, implement it on:
  - Multi-core CPU
  - GPU/Multi-GPU
  - FPGA/Multi-FPGA **(DPU)**

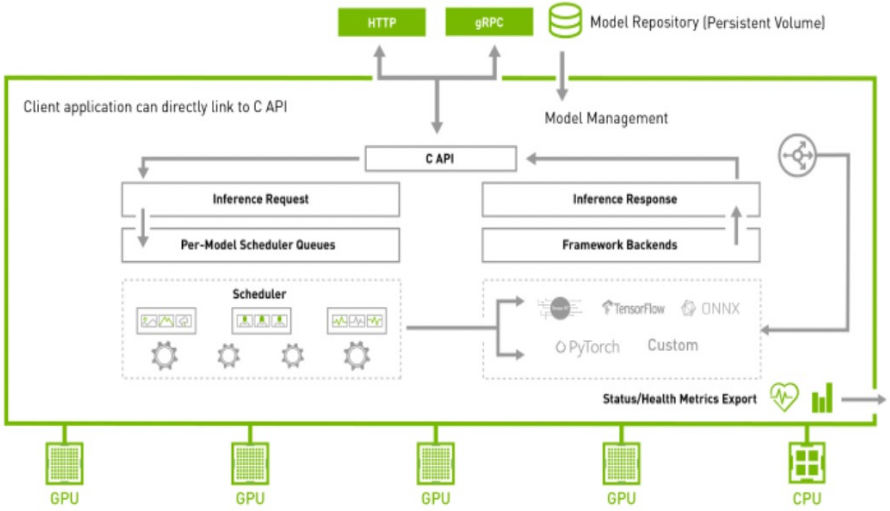
Architecture

Overall

Front End

Use GRC as a familiar front-end to develop blocks.





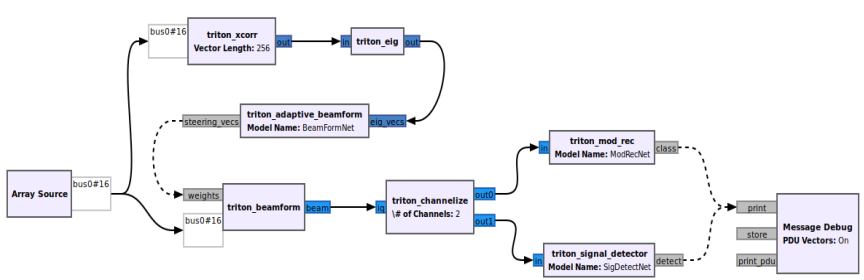
Application

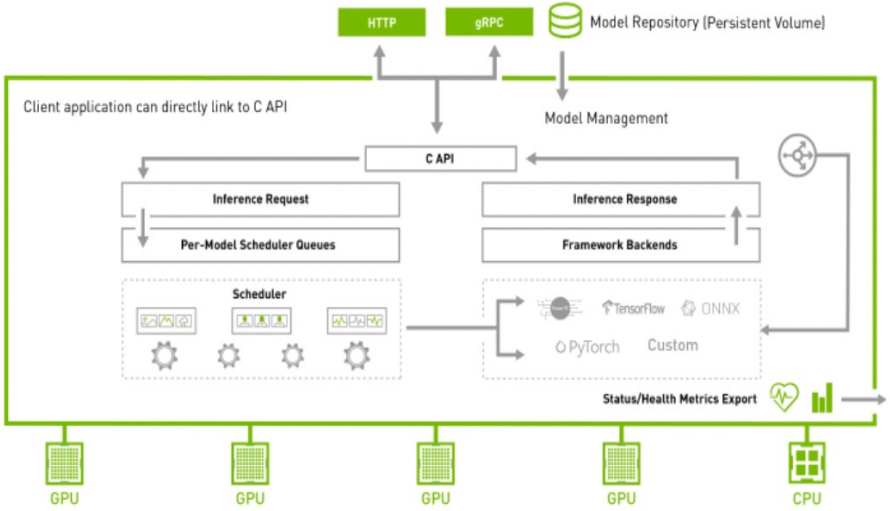
Software and Hardware

Generic GR "Model" block

GR blocks just call to the inference servers

```
int triton_block_impl::work(
    int noutput_items,
    gr_vector_const_void_star& input_items,
    gr_vector_void_star& 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_patch is fixed.
    auto num_items_per_batch =
        model_.get()->get_output_sizes()[0] / model_.get()->get_output_signature()[0];
    auto batch_size = noutput_items / num_items_per_batch;
    model_->infer_batch(in_ptrs, out_ptrs, batch_size);

    return noutput_items;
}
```





Application

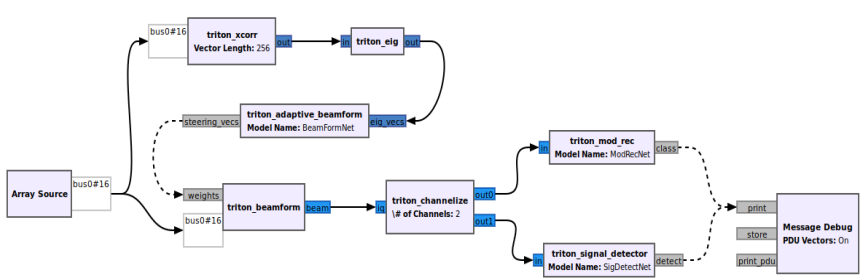
Software and Hardware

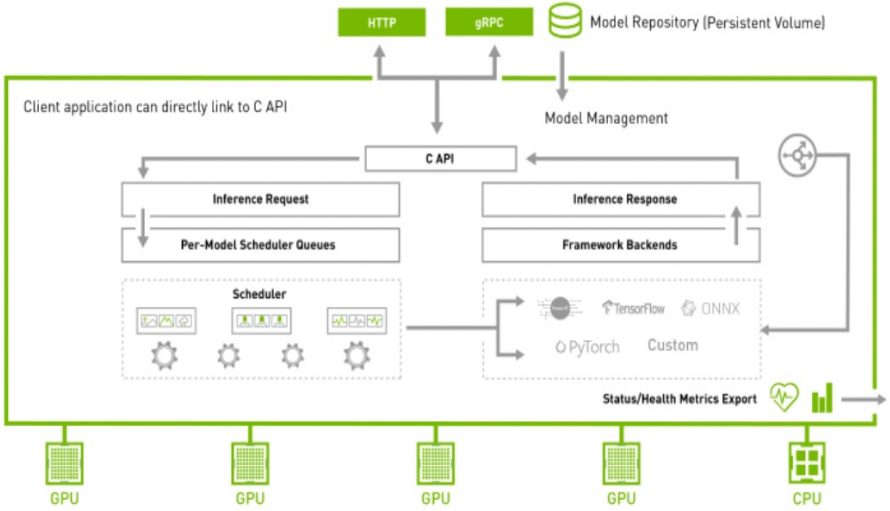
Generic GR "Model" block

Peraton | LABS

Each call is a batch of data

```
int triton_block_impl::work(
    int noutput_items,
    gr_vector_const_void_star& input_items,
    gr_vector_void_star& 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_patch is fixed.
    auto num_items_per_batch =
        model_.get()->get_output_sizes()[0] / model_.get()->get_output_signature()[0];
    auto batch_size = noutput_items / num_items_per_batch;
    model_->infer_batch(in_ptrs, out_ptrs, batch_size);

    return noutput_items;
}
```





Application

Software and Hardware

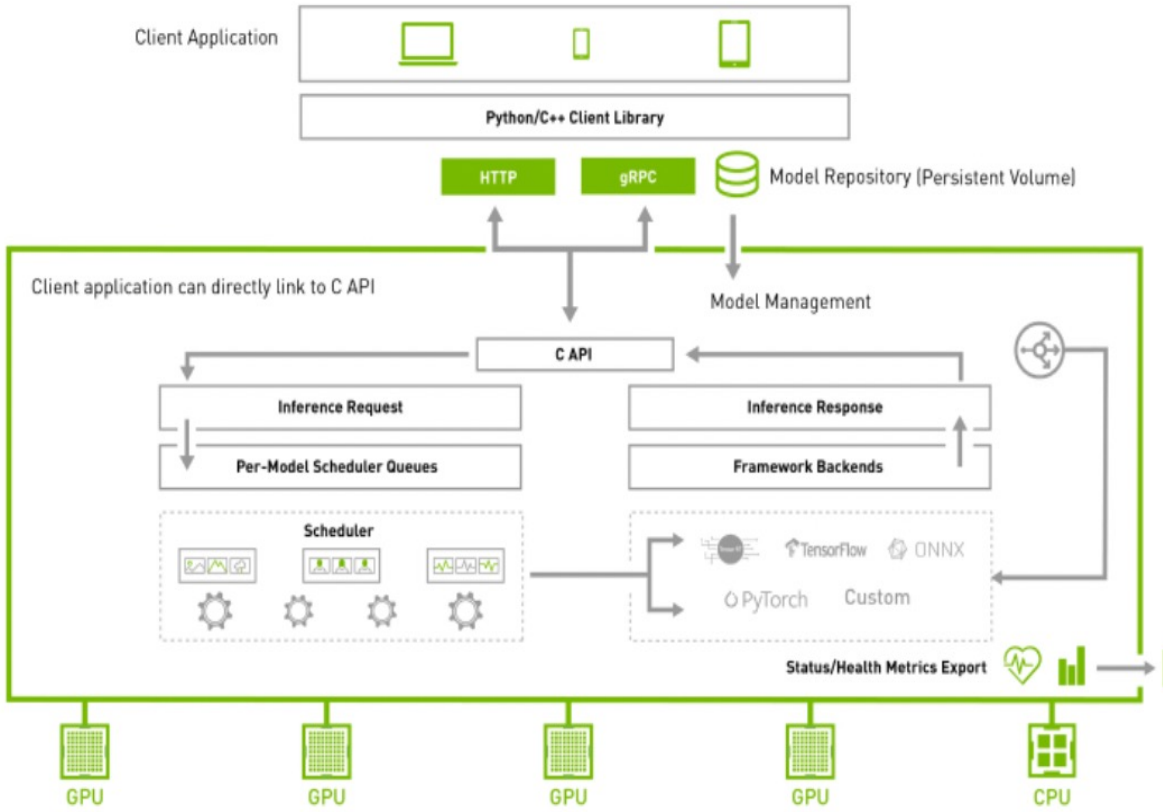
Peraton | LABS

Architecture

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

TIS Internals

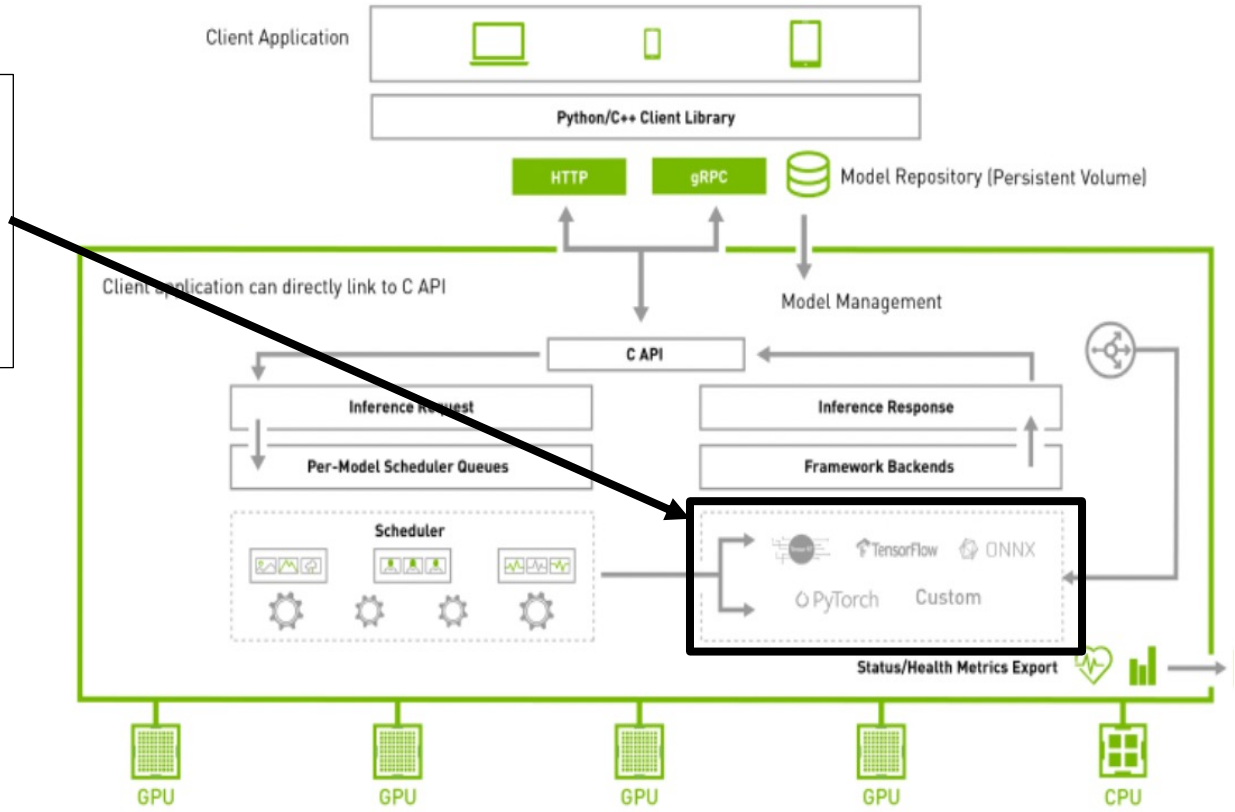
Here's TIS's overall architecture



TIS Internals

Peraton | LABS

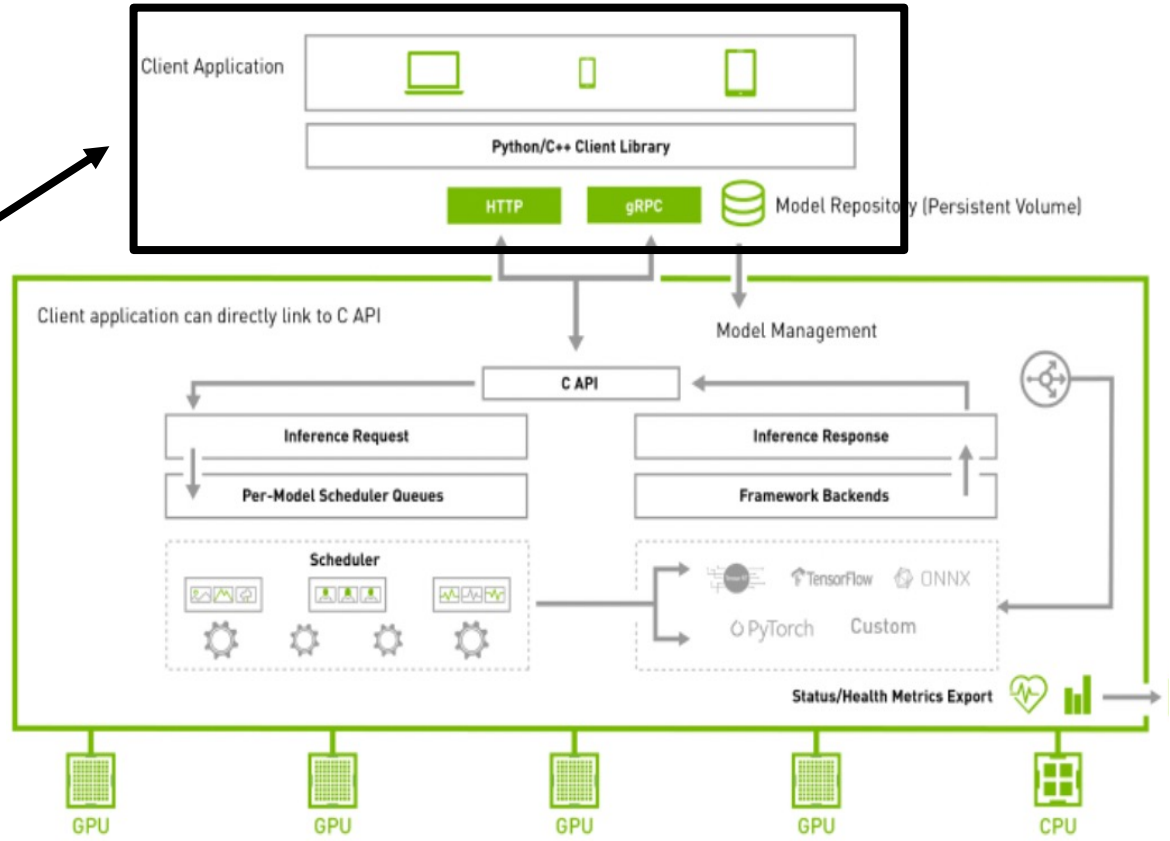
Models are defined through a number of common frameworks



TIS Internals

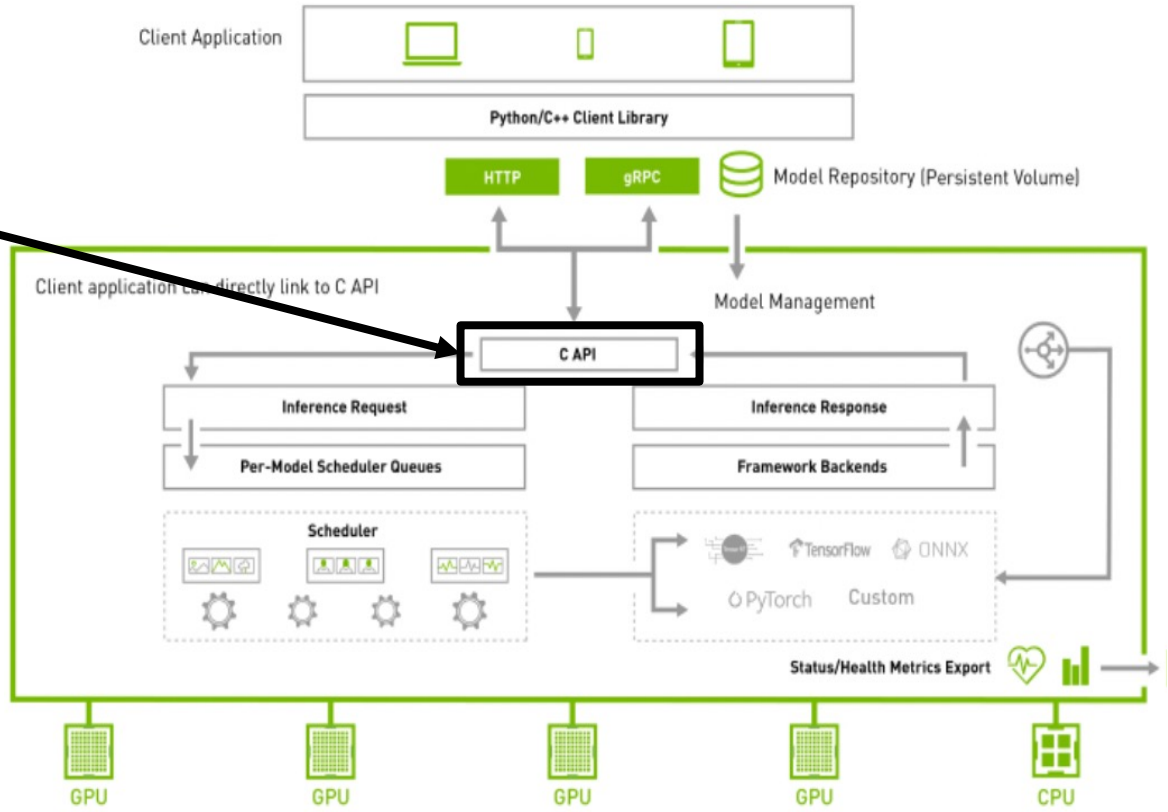
Peraton | LABS

Client libraries are provided to issue Inference Requests



TIS Internals

There is a supported way to make direct calls to inference rather than remote calls, to be more efficient

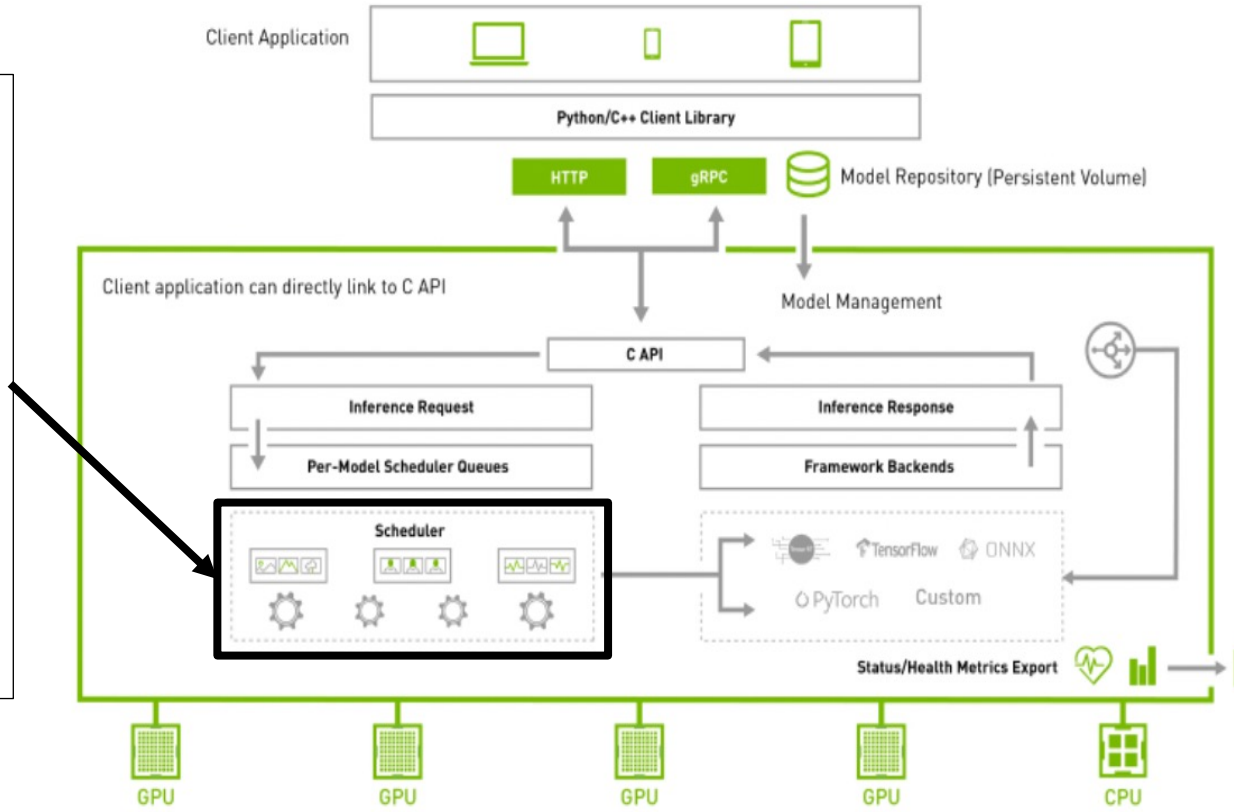


TIS Internals

Peraton | LABS

Each “model” has inference requests has a scheduler.

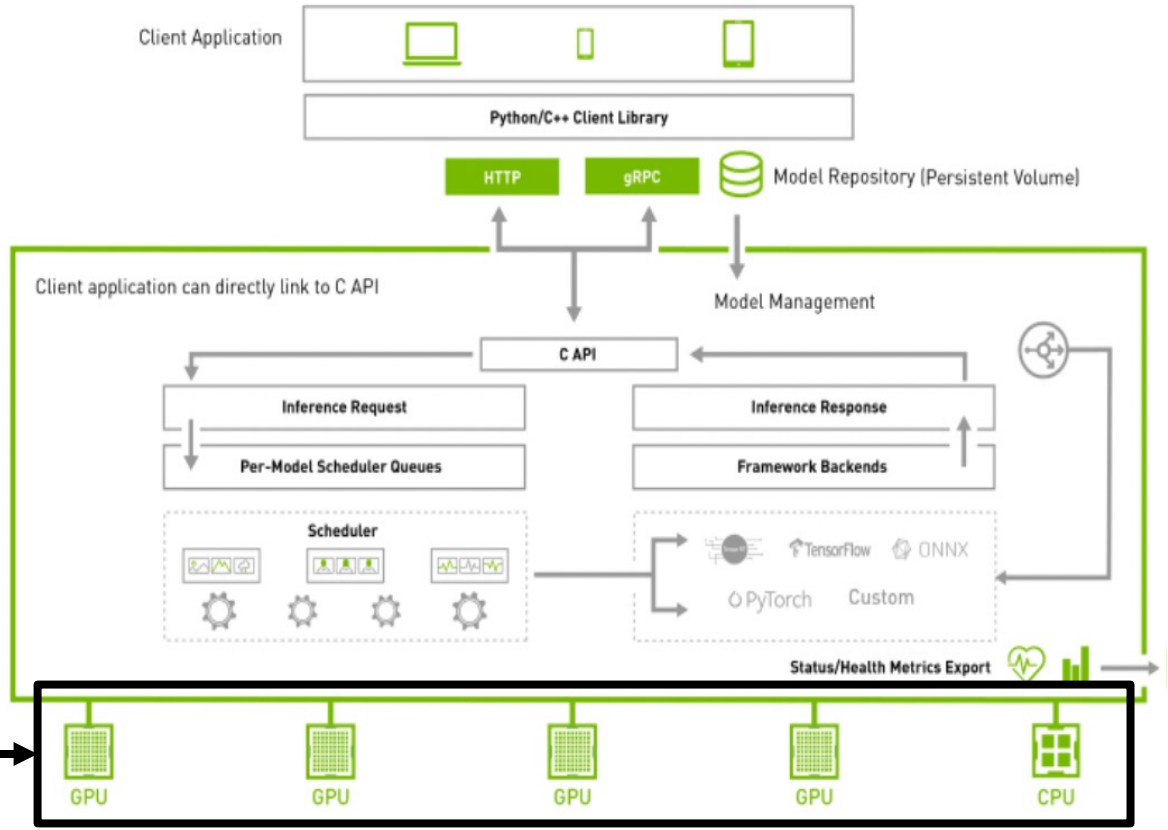
The scheduler can be configured.



TIS Internals

```
instance_group [
        {
                count: 1
                kind: KIND_CPU
        }
]
optimization { execution_accelerators {
  cpu_execution_accelerator : [ {
    name : "openvino"
  }]
}}
```

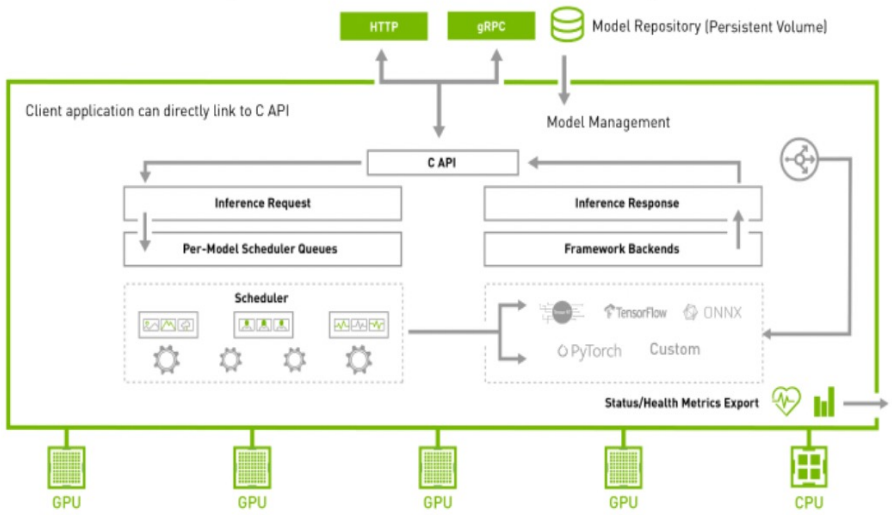
Models are instantiated across a configurable number of devices.



TIS Integration

```python
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)
```

Each block is defined in Python as a Pytorch Module.



Peraton | LABS

Architecture

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

What is the DPU?

Peraton | LABS

- 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?

Peraton | LABS

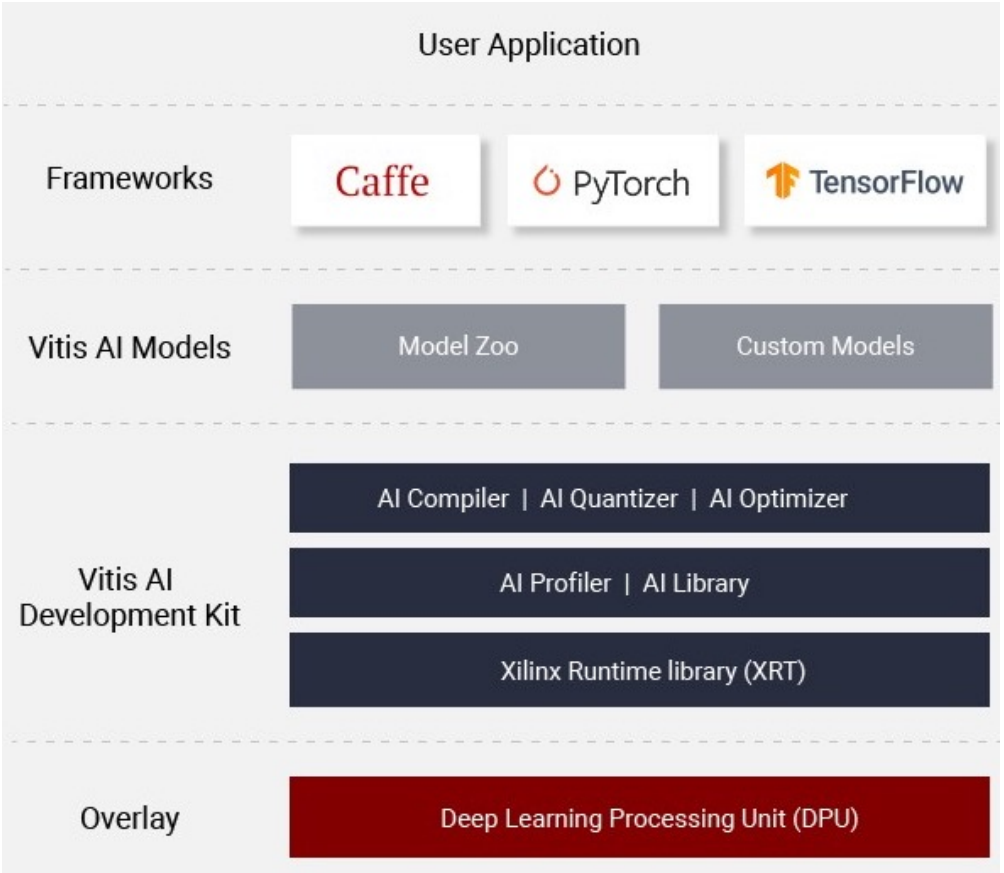
- Rapid Development/Prototyping
  - Place-and-route not required when to implement a new "model"
- Parallelism
  - Highly parallelized architecture for convolutions can be exploited for efficient DSP/ML
- Simplified Development Interface
  - Python → FPGA implementation

DPU Implementation

Peraton | LABS

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

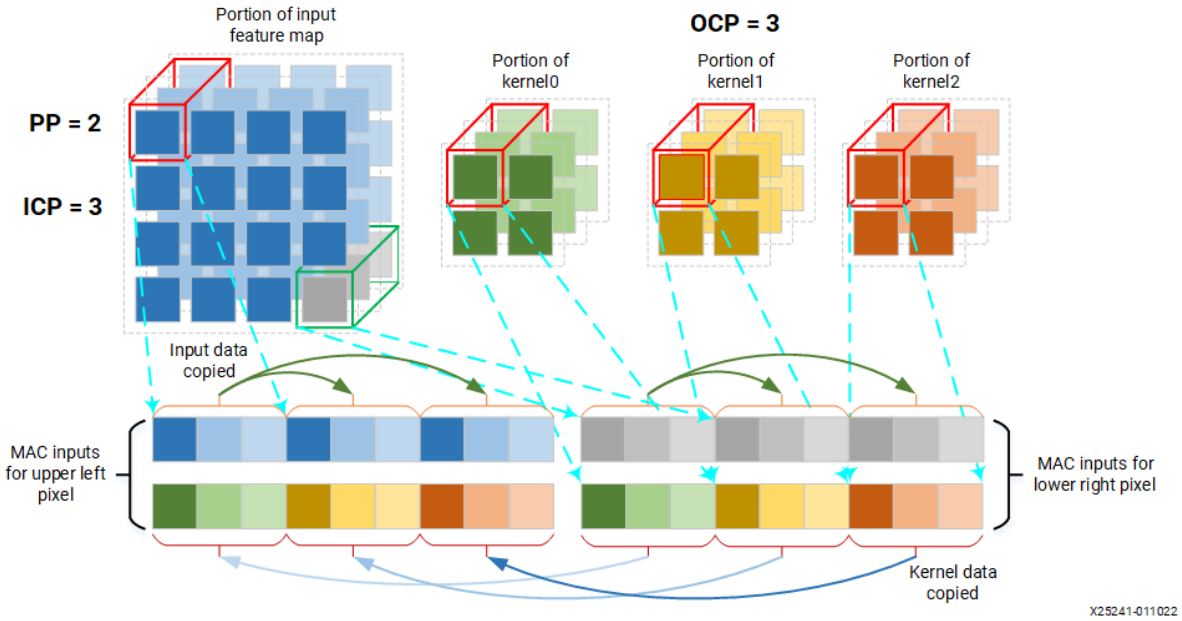


DPU Implementation

Peraton LABS

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



DPU Implementation

Peraton | LABS

Some Caveats

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

Peraton | LABS

Throughput Measurements

Triton Inference Server on CPU/GPU/multi-GPU

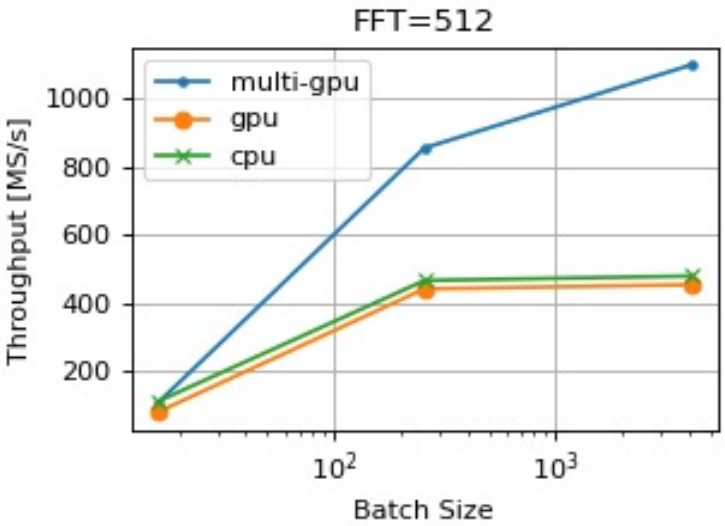
TIS Throughput Measurements

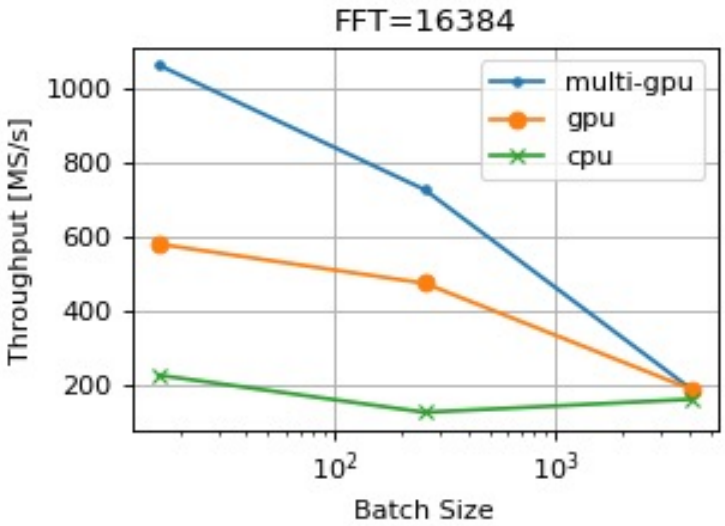
Peraton | LABS

- 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

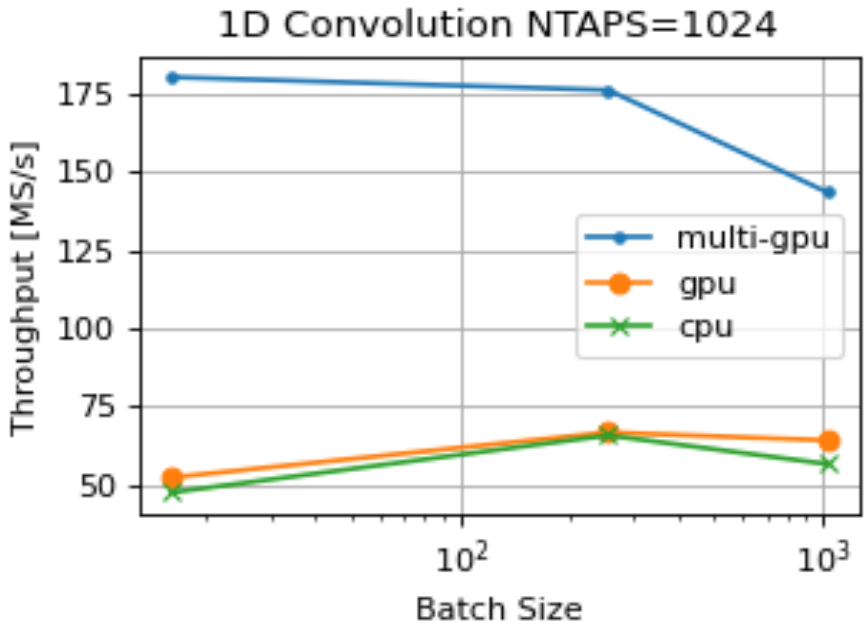
- GPU and CPU perform similar for smaller FFT
- Too large FFT and batch size result in overhead from TIS





TIS Throughput: Convolve

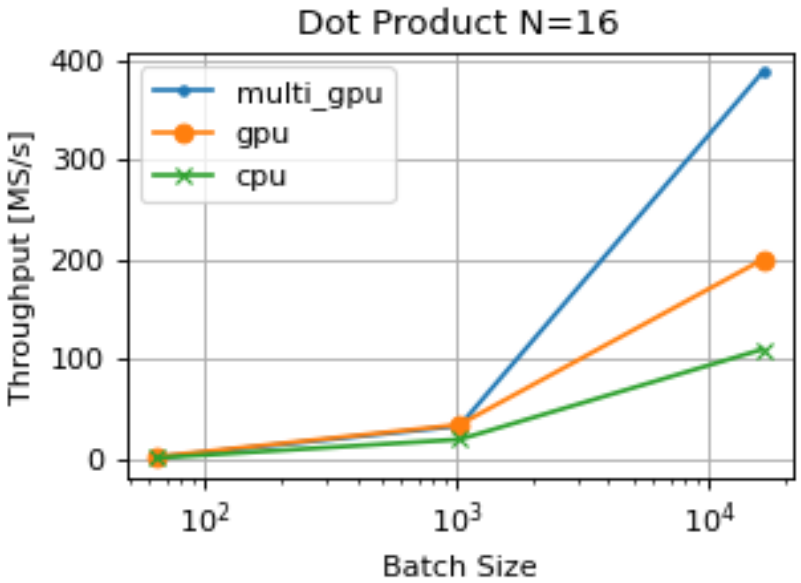
- GPU and CPU still perform similarly for smaller convolutions
- Multi-GPU is closer to linear in speed-up per GPU



TIS Throughput: Dot Product

Peraton | LABS

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



Peraton | LABS

Throughput Measurements

The DPU (and eventually the AMD/Xilinx Server)

DPU Benchmarks

Peraton | LABS

- **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

DPU Throughput: Dot Product

Peraton | LABS

- 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

Peraton | LABS

- 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

DPU Throughput: FFT

Peraton LABS

- 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

Peraton | LABS

Parameter	AlexNet	GoogLeNet	Inception3	MobileNet V2	Resnet101	Resnet152	Resnet34	Resnet50	VGG16
Calibration time (s)	17.9	47.0	97.1	63.5	173.8	243.1	51.3	115.4	106.2
Top 1 Accuracy	47.3	85.6	69.5	36.3	46.9	80.2	81.7	44.9	39.8
Top 5 Accuracy	51.9	98.7	81.1	57.3	59.0	97.4	94.3	61.7	46.2
Loss	6.7	0.5	4.9	6.1	5.4	0.7	2.5	5.2	6.7
GPU Throughput (ms)	5.2	11.6	17.5	10.3	19.0	26.7	9.3	11.3	11.2
GPU Throughput FPS	190.8	86.4	57.1	97.6	52.5	37.5	107.3	88.2	89.6
GPU Latency (ms)	106.0	67.2	105.9	46.8	133.6	186.8	58.3	87.5	249.2
FPGA Latency (ms)	43.0	15.1	44.7	39.1	47.3	41.7	41.1	43.1	65.5
FPGA FPS	27.3	224.5	27.1	27.9	27.3	28.8	27.4	26.8	20.3
FPGA Throughput (ms)	36.7	4.5	36.9	35.8	36.7	34.7	36.6	37.3	49.2
FPGA Performance (GOP/s)	141.5	679.3	726.9	187.4	644.9	665.9	886.5	597.5	629.4
FPGA Speedup (Throughput)	0.143	2.600	0.474	0.286	0.519	0.769	0.255	0.304	0.227
Image Size	224x224	224x224	299x299	224x224	224x224	224x224	224x224	224x224	224x224

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

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