IEEE IUS 2023
International Ultrasonics Symposium
Palais des congrès de Montréal || September 3-8, 2023

us4us®

IPPT PAN

UNIVERSITY OF WATERLOO
CONCORDIA CUM VERITATE

GPU-Based Ultrasound Signal Processing

Real-Time Speed of Sound Estimation and CNN Inferencing Using GPU

Presenter: Dr Billy Yiu

Outline of Discussion

Execute your algorithm on GPU using existing libraries

Part I   Real-time speed of sound estimation using GPU

- How can we use CuPy to execute the algorithm in real-time/close to real-time

Part II   C++ Deployment of Machine Learning Algorithms

- How can we use ONNX and TensorRT to deploy a machine learning algorithm in a standalone program?

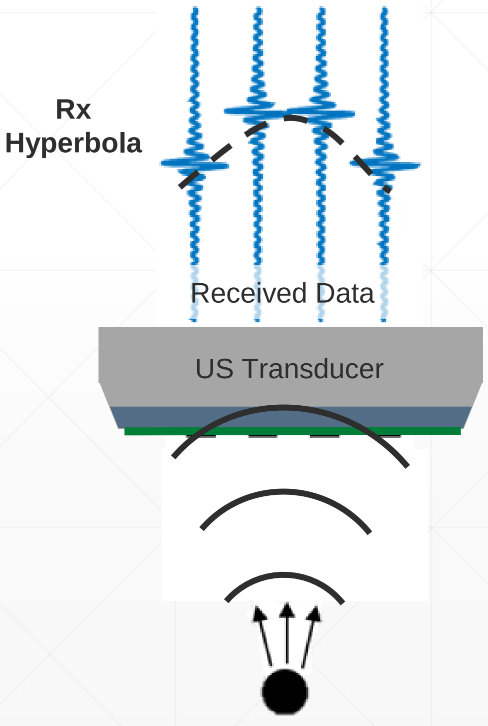
Real-Time Speed of Sound Estimation Using GPU

Di Xiao, Hassan Nahas, Billy Yiu

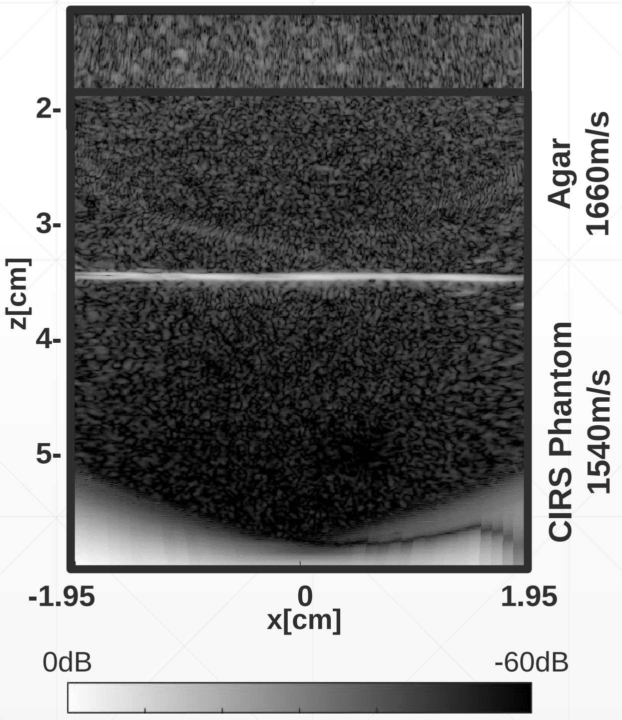
Ultrasound Imaging with Assumed Speed-of-Sound (SoS) Value

Challenge: Loss of image quality when beamformed with wrong SoS

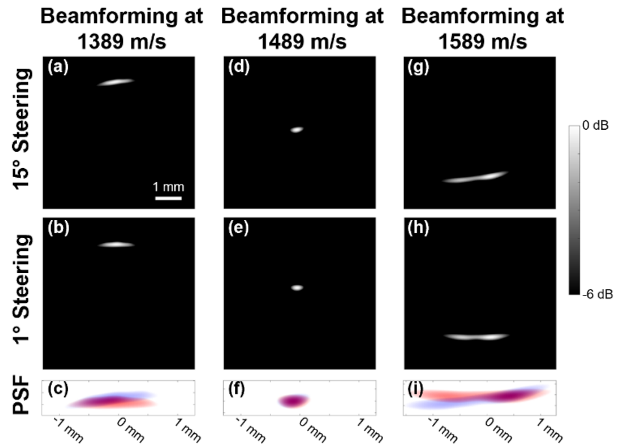
Plane Wave Imaging Forward Model



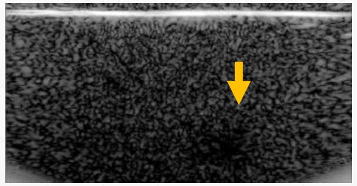
Effect of Different Beamforming SoS Values



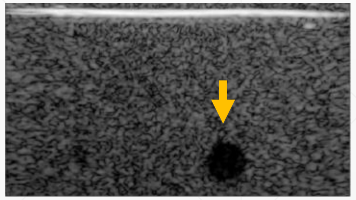
Point Spread Function of Wire Target in Water



Worst Estimate [1450m/s]



Best Estimate [1630m/s]

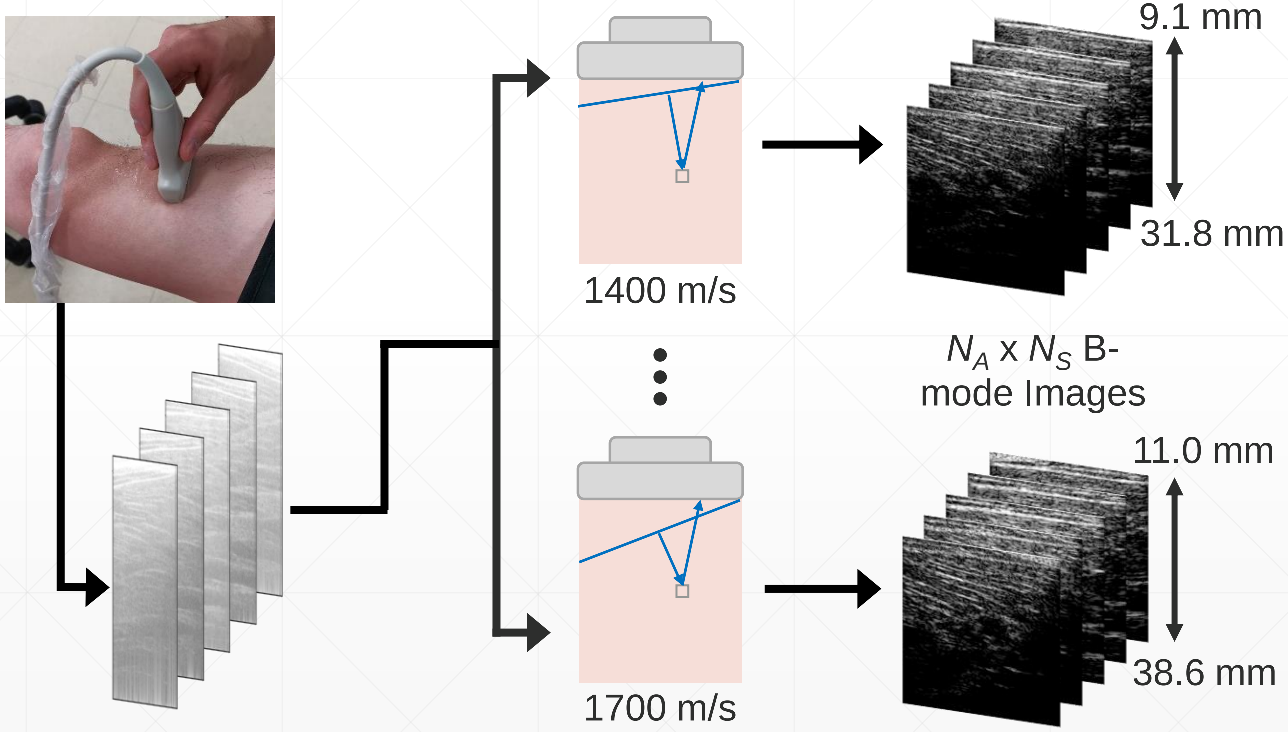


Global SoS Estimation: Modified Delay-and-Sum Beamforming

Goal: Provide a real-time estimate of the average (global) SoS of the imaged medium

1. Acquisition

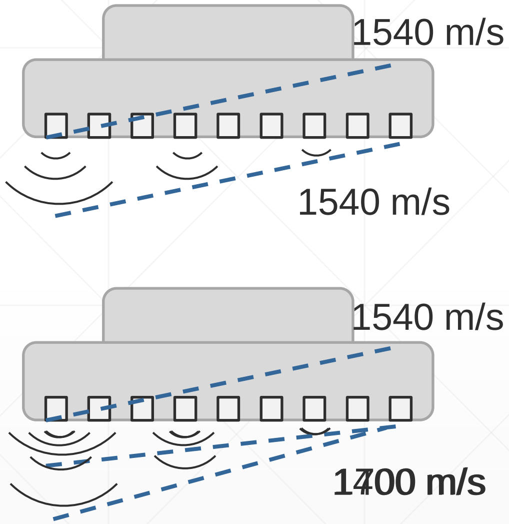
2. Speed-of-Sound Aware Beamforming



Plane Wave RF Data from N_A Angles

Beamform for a range of SoS values

Refraction Compensation



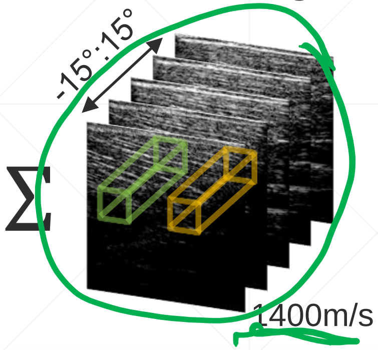
$$\theta_{\text{medium}} = \sin^{-1}\left(\sin\theta_{\text{probe}} \frac{v_{\text{medium}}}{v_{\text{assumed}}}\right)$$

Snell's Law

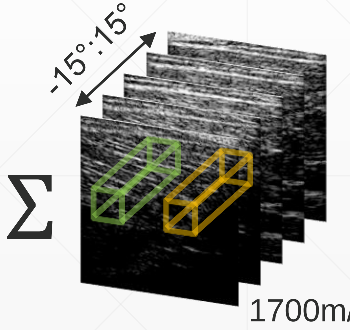
Global SoS Estimation:
Identifying Best Beamforming SoS

Goal: Provide a real-time estimate of the average (global) SoS of the imaged medium

3. Inter-steering Loss



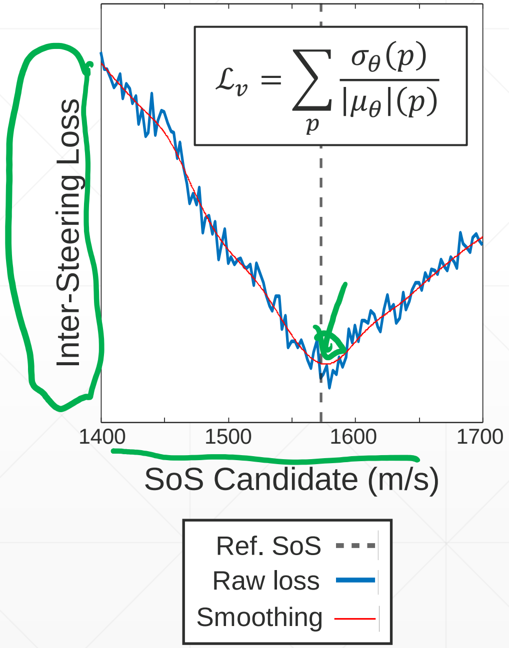
Inter-Steering Loss
Calculation



Calculate a loss for each SoS:
differences between steering angles

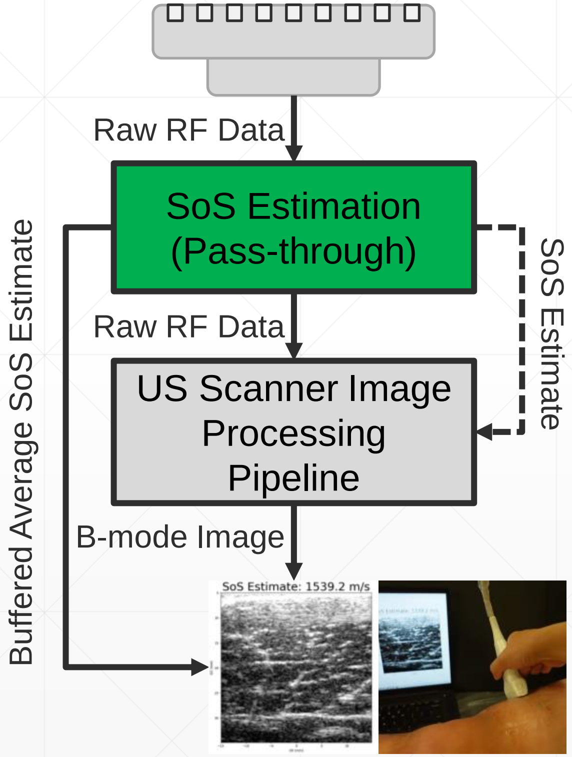
4. SoS Estimation

***In vivo* Example Loss**



Estimate the SoS by finding
the smoothed loss minimum

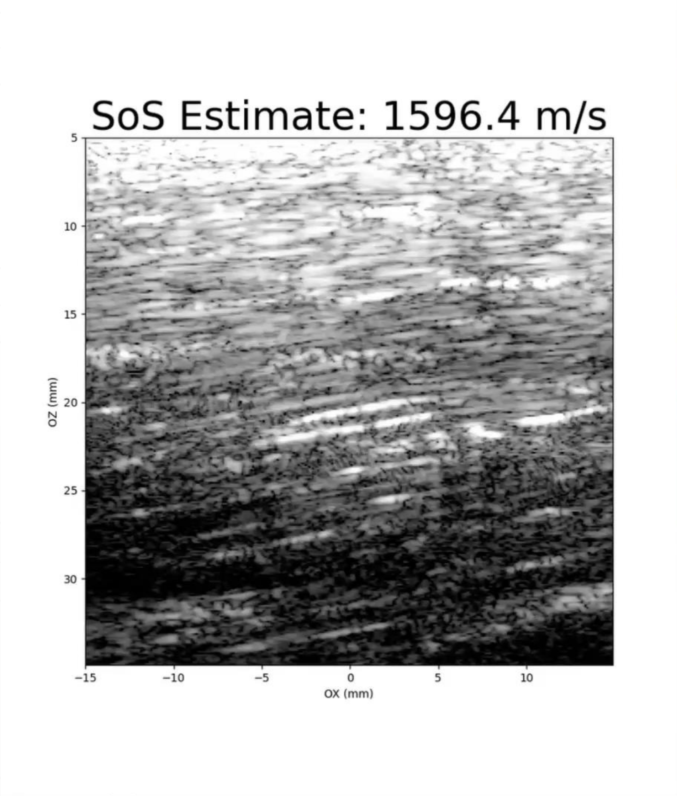
Real-Time Global SoS Estimation

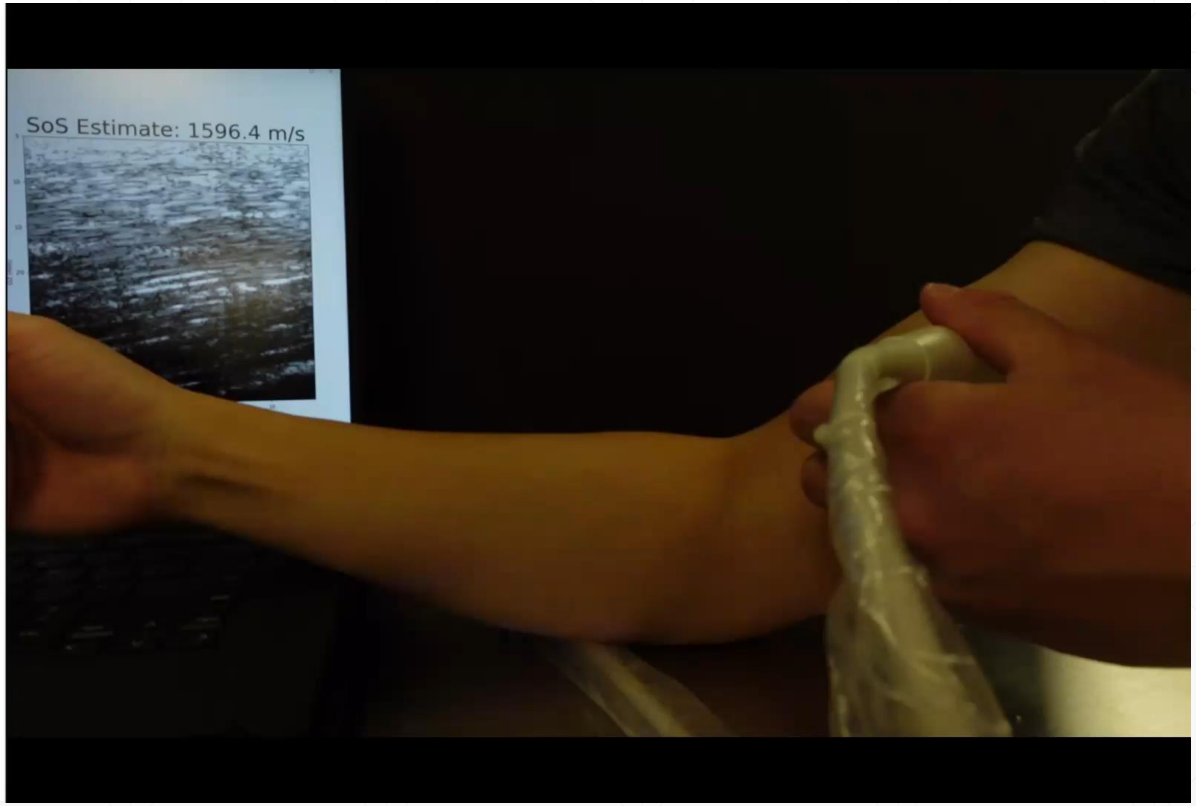


Display the SoS estimate in
real-time alongside B-mode

A High-Level Overview of the GPU Implementation of the SOS Estimation Algorithm

RF Data → **Pre-Filtering** → **Beamforming** → **SOS Loss Computation** → **Display SOS on Screen**



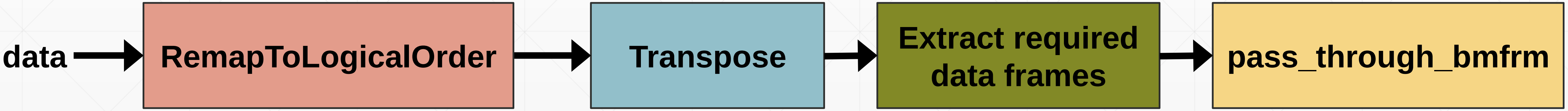


Python Code Structure for Real-Time SoS Estimation

Leverage the customizable receive processing pipeline

```
scheme = Scheme(
    tx_rx_sequence=sequence,
    processing=Pipeline(
        steps=(
            Lambda(lambda data: (data_timestamps.append(time_ns()), data)[1]),
            RemapToLogicalOrder(),
            Transpose(axes=(2, 3, 1, 0)),
            Lambda(lambda data: (data_buffer_rf.append(data.get()), data[:,:,:Nang,:])[1]),
            Lambda(pass_through_bmfrm,lambda metadata:  metadata.copy(input_shape=(Nz,Nx),dtype="float32")),
            ),
            placement="/GPU:0"
        ))
```

<u>Pipeline</u>: Automatically execute the specified functions sequentially



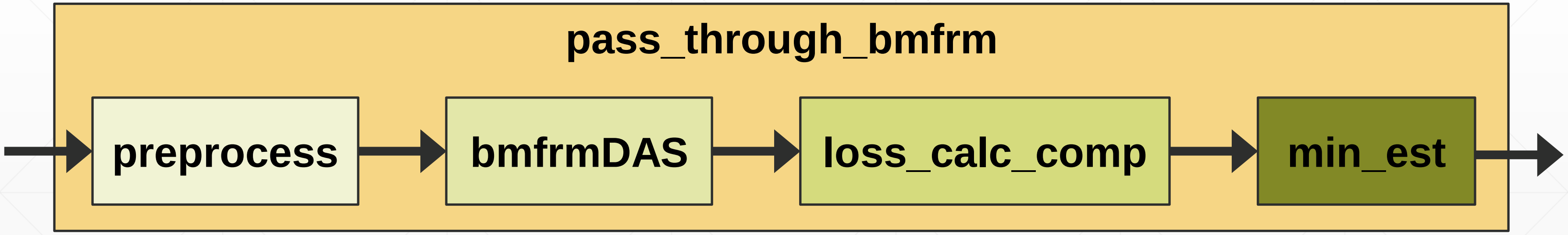
<u>Lambda function</u>: A function handler that the pipeline can call

Inside the Pass_Through_Bmfrm…

```python
def pass_through_bmfrm(data):
    data_preprocess = preprocess(data)
    b_img = bmfrmDAS(data_preprocess)
    sos_est = min_est(loss_calc_comp(data_preprocess).get(),sosBmfrm,sosTest)
    data_buffer_sos.append(sos_est)
    print("FPS: {:3.1f} fps".format(fps_calc(data_timestamps)),end='\r')
    return b_img
```

sos aware BF

Modularize the processing pipeline



These modules can be CuPy built-in or custom functions

Inside the Pass_Through_Bmfrm…

```python
def pass_through_bmfrm(data):
    data_preprocess = preprocess(data)
    b_img = bmfrmDAS(data_preprocess)
    sos_est = min_est(loss_calc_comp(data_preprocess).get(),sosBmfrm,sosTest)
    data_buffer_sos.append(sos_est)
    print("FPS: {:3.1f} fps".format(fps_calc(data_timestamps)),end='\r')
    return b_img
```

These modules can be CuPy built-in or custom functions

- **Divide the overall process into multiple steps**
  - Allows optimized implementation for each step
  - All steps are implemented using CuPy functions
  - Some of the functions can be implemented using custom kernels
- **Same as C++ implementation**
  - Use pre-built libraries for convenience
  - Built custom implementation for mission-critical steps

Implementation of the PreProcess Module

```python
def preprocess(rfTensor):
    data_preprocess = hilbert_fft(bandpass_filter(rfTensor.astype('float32')))
    data_preprocess = cp.reshape(data_preprocess,(Nsamp*Nchan*Nang,),order='F')
    return data_preprocess
```

bandpass_filter: Suppress out-of-band noise

```python
def bandpass_filter(rfTensor):
    return cp.flip(convolve1d(cp.flip(convolve1d(rfTensor,cp.squeeze(filtcoeff_gpu), axis = 0),axis = 0), cp.squeeze(filtcoeff_gpu), axis = 0),axis = 0)
```

Perform forward-backward FIR filtering using convolve1d from *cupyx.scipy*

hilbert_fft: Convert the RF data into its analytic form

```python
def hilbert_fft(rfTensor):
    rf_fft = cufft.fft(rfTensor,axis = 0)
    rf_fft[(rfTensor.shape[0]/2):,:,:] = 0
    return cufft.ifft(rf_fft,axis = 0)
```

Perform analytic signal conversion using Hilbert transform (zero-out the –ve side of the frequency spectrum)

reshape: reshape the filtered data into a column vector format for the next step

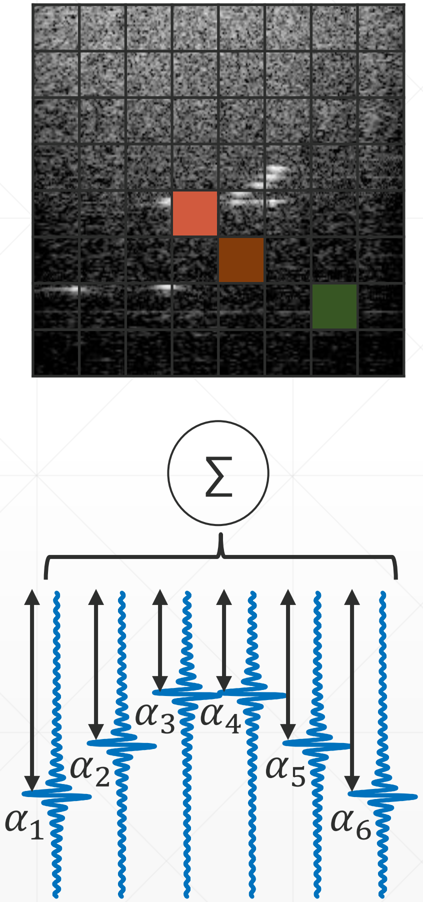
Implementation of the Beamforming Module

```
def bmfrmDAS(data_preprocess):
    img=cp.reshape(sp_csr_bmfrm.dot(data_preprocess),(Nz*Nx,Nang),order='F')
    img[img==0] = cp.asarray(np.nan*0j)
    img = 20*cp.log10(cp.abs(cp.nanmean(img,axis=1)))
    img = cp.nan_to_num(img, copy=False, nan=-100)
    img = cp.reshape(img,(Nz,Nx),order='F')
    return img
```

Sparse matrix beamforming – Multiply the RF data matrix with pre-computed delay matrices

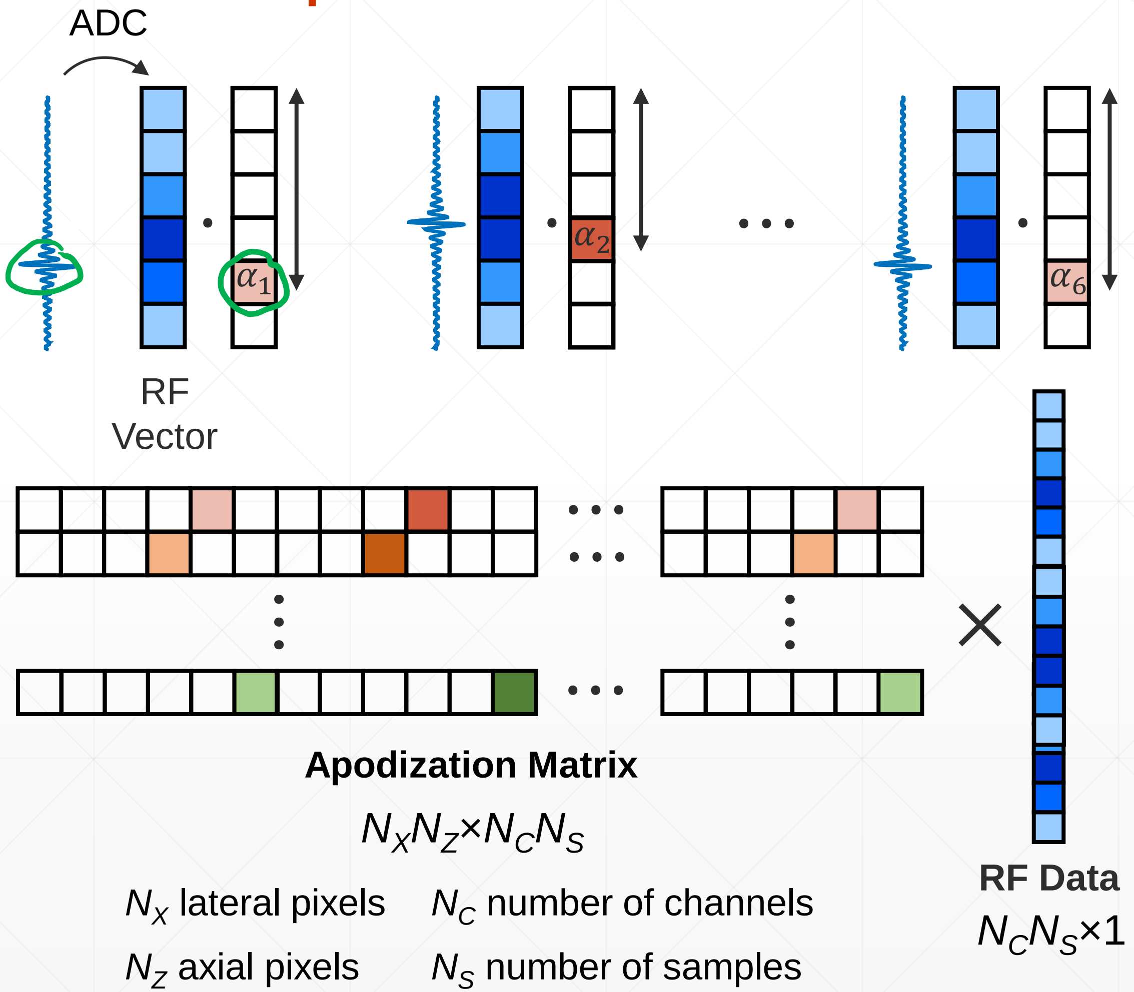
Brief Summary of Sparse Matrix Beamforming

Pixel by Pixel Approach



Analytic RF Data

Expression in Matricial Form



Implementation of the Beamforming Module

```python
def bmfrmDAS(data_preprocess):
    img=cp.reshape(sp_csr_bmfrm.dot(data_preprocess),(Nz*Nx,Nang),order='F')
    img[img==0] = cp.asarray(np.nan*0j)
    img = 20*cp.log10(cp.abs(cp.nanmean(img,axis=1)))
    img = cp.nan_to_num(img, copy=False, nan=-100)
    img = cp.reshape(img,(Nz,Nx),order='F')
    return img
```

Sparse matrix beamforming – Multiply the RF data matrix with pre-computed delay matrices

1. **Perform matrix multiplication (dot product) between the delay-encoded sparse matrix with the RF data column vectors**
   - sp_csr_bmfrm is an CuPy sparse matrix
   - Apodization is also encoded into the sparse matrix as well

2. **Log compress the output vector and reshape it into a 2D matrix (i.e., an image)**

Easy and straightforward to implement

Implementation of Loss Function Computation

```
def loss_calc_comp(data_preprocess):
    bimgs = cp.reshape(sp_csr.dot(data_preprocess),(Npix*Npix*Nsos,Nang),order='F')
    bimgs[bimgs==0] = cp.asarray(np.nan*0j)
    std_comp = cp.sqrt(cp.square(cp.nanstd(cp.real(bimgs),axis=1,ddof=1))
        +cp.square(cp.nanstd(cp.imag(bimgs),axis=1,ddof=1)))
    coefvar = cp.divide(std_comp,cp.nanmean(cp.abs(bimgs),axis=1))
    loss = cp.nansum(cp.reshape(coefvar,(Npix*Npix,Nsos),order='F'),axis=0)
    return cp.divide(cp.squeeze(loss),norm_factor)
```

1. **Use sparse matrix beamforming to generate images with different SoS**
    - sp_csr is an CuPy sparse matrix that accounts for SoS when encoding the delay

2. **Compute the coefficient of variation (CoV) across the angles**

3. **Cumulate the CoV across the image as the loss for different pre-defined SoS**

Estimating the SoS from the Loss Function

```
def min_est(loss_est,sosBmfrm,sosTest):
    spl = UnivariateSpline(sosBmfrm, loss_est, s=0)
    int_points = spl(sosTest)
    min_sos = sosTest[np.argmin(int_points)]
    return min_sos
```

1. **Interpolate the loss function with univariate spline function**
   - Done in the CPU

2. **Find and return the SoS at minimum**

These modules can be CuPy built-in functions or custom functions

Comparison Between Python and C++ Implementation

Python Implementation

CuPy makes GPU implementation of your algorithms much easier

- Similar to MatLab script
- No need to implement everything by yourselves
- Speed up your algorithms for a real-time /close-to-real-time performance
- Based on cuBLAS, cuRAND, cuSOLVER, cuSPARSE, cuFFT, cuDNN
- Easy display options

C++ Implementation

C++ provides a standalone program for the execution

- Greater control on implementation
- Need to implement many things by yourselves
- Optimize the implementation specific to your algorithms for a real-time performance
- Libraries like cuBLAS, cuRAND, cuSOLVER, cuSPARSE, cuFFT, cuDNN are available
- Greater control on display rendering

C++ Deployment of Machine Learning Algorithms

Di Xiao, Hassan Nahas, Billy Yiu

Machine Learning in Ultrasound Imaging

Involved in every stage of ultrasound image processing

Upstream Process

RF Data Transfer

Channel Recovery

Compressive Sensing

Midstream Process

Beamforming

ML beamforming

Spatial compounding

Harmonic imaging

Super resolution imaging

Downstream Process

Color Doppler

Dealiasing

Clutter filtering

Image Processing

Image despeckling

Quantitative analysis

Speed of sound

Speckle distribution

Object identification

Lesion detection

Organ labelling

Deploying a Trained Neural Network in C++ Environment

Not a straightforward task

TensorFlow C++

TensorFlow API directly in C++

Supports inference on CPU and GPU

TensorRT

API for optimized inference on NVIDIA devices.

May be used with other deep learning packages (e.g.: Pytorch)

TensorFlow-TensorRT

Direct integration between TF and TRT

TensorRT for Model Deployment

Save the model in TensorRT readable format: ONNX

TensorFlow → TensorFlow Model → **TF2ONNX** → ONNX Model → **TensorRT** → Deployment

Python

Trained model in TensorFlow-Python

```
# TF2 model development
model = …
model.train(…)
# saving
save_model(model, …)
```

↓

Convert to ONNX

```
# using tf2onnx
>> python -m tf2onnx.convert …
```

TF2ONNX is open-source and available at https://onnxruntime.ai/docs/tutorials/tf-get-started.html

TensorRT for Model Deployment

Save the model in TensorRT readable format: ONNX

TensorFlow → TensorFlow Model → **TF2ONNX** → ONNX Model → **TensorRT** → Deployment

C++

Parse ONNX model using TensorRT

▼

TensorRT engine available for inference

```
// Define necessary TRT builders, config, etc.
…
parser->parseFromFile(…) // Parse ONNX model
…
Return mEngine // TRT engine for inference
```

```
context = mEngine->createExecutionContext(…)
…
context->executeV2(….data()) // results on buffers
```

Deploying a CNN Model Trained with MNIST Dataset: A Brief Walkthrough

1. Define the TF-Keras model

```
########################################
# Creates the model.

model = tf.keras.models.Sequential([
    tf.keras.Input((28, 28,1),batch_size=1),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10),
    tf.keras.layers.Softmax()
])
```

2. Train the model with MNIST dataset

```
####################################
# Train the model on MNIST

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0


loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False)

model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])


model.fit(x_train, y_train, epochs=5)

model.evaluate(x_test,  y_test, verbose=2)
```

*Based on sample codes by NVIDIA and ONNX project

Deploying a CNN Model Trained with MNIST Dataset: A Brief Walkthrough

3. Save the TensorFlow model

```
########################################
# Saves the model for ONNX conversion.

if not os.path.exists("simple_nn"):
    os.mkdir("simple_nn")
tf.keras.models.save_model(model, "simple_nn")
```

4. Convert the saved TensorFlow model into an ONNX one

```
########################################
# Run the command line for ONNX conversion.

proc = subprocess.run('python -m tf2onnx.convert --saved-model simple_nn '
                      '--output ../cpp/sample_mnist_data/simple_nn.onnx --opset 15'.split(),
                      capture_output=True)
```

*Based on sample codes by NVIDIA and ONNX project

Deploying a CNN Model Trained with MNIST Dataset: A Brief Walkthrough

5. Load the ONNX model into C++ environment

```cpp
// create builder object
auto builder = SampleUniquePtr<nvinfer1::IBuilder>(nvinfer1::createInferBuilder(sample::gLogger.getTRTLogger()));

// create network object
const auto explicitBatch = 1U << static_cast<uint32_t>(NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
auto network = SampleUniquePtr<nvinfer1::INetworkDefinition>(builder->createNetworkV2(explicitBatch));

// create config object
auto config = SampleUniquePtr<nvinfer1::IBuilderConfig>(builder->createBuilderConfig());

// create parser object
auto parser = SampleUniquePtr<nvonnxparser::IParser>(nvonnxparser::createParser(*network, sample::gLogger.getTRTLogger()));

// parse ONNX model from filepath
auto parsed = parser->parseFromFile(onnxModelPath.c_str(), static_cast<int>(sample::gLogger.getReportableSeverity()));
```

*Based on sample codes by NVIDIA and ONNX project

Deploying a CNN Model Trained with MNIST Dataset: A Brief Walkthrough

6. Configure the CNN engine

```
// Configure DLA
samplesCommon::enableDLA(builder.get(), config.get(), dlaCore);

// Build serialized network and return it
SampleUniquePtr<IHostMemory> plan{ builder->buildSerializedNetwork(*network, *config) };

// Get inference runtime
mRuntime = std::shared_ptr<nvinfer1::IRuntime>(createInferRuntime(sample::gLogger.getTRTLogger()));

// Get inference engine for network
mEngine = std::shared_ptr<nvinfer1::ICudaEngine>(
    mRuntime->deserializeCudaEngine(plan->data(), plan->size()), samplesCommon::InferDeleter());
```

*Based on sample codes by NVIDIA and ONNX project

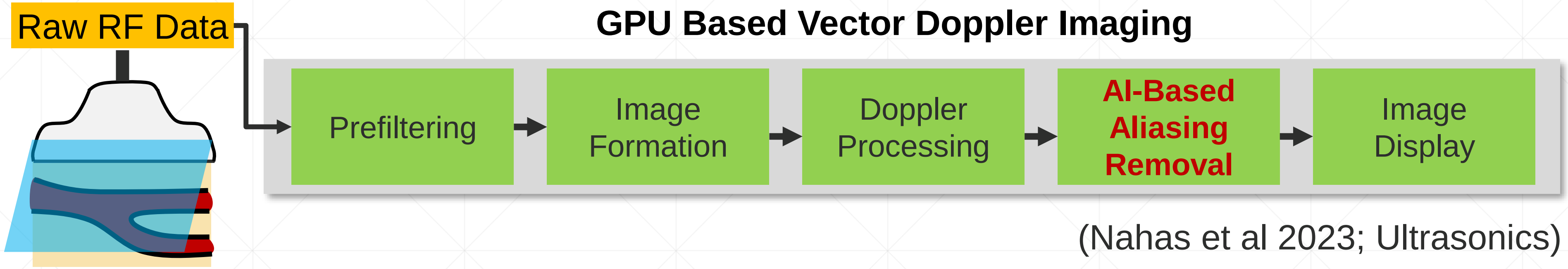
Deploying a CNN Model Trained with MNIST Dataset: A Brief Walkthrough

7. Execute the CNN for inferencing

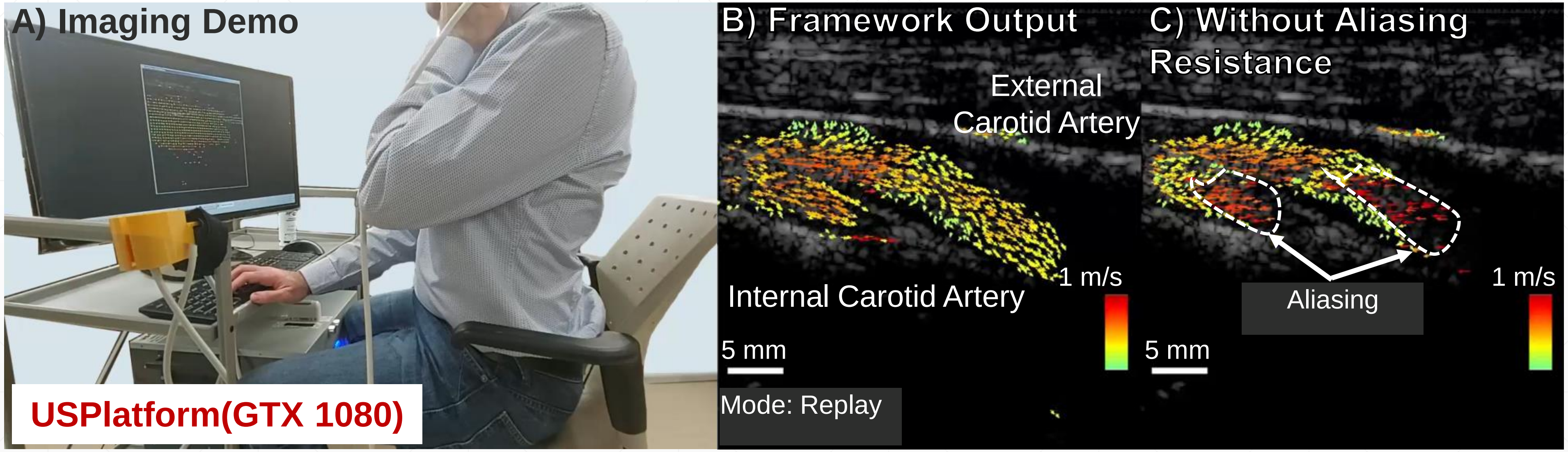
```
// buffers for input/output of model
samplesCommon::BufferManager buffers(mEngine);

// Create execution context for inference from engine.
auto context = SampleUniquePtr<nvinfer1::IExecutionContext>(mEngine->createExecutionContext());

// Read the input data into the managed buffers
processInput(buffers);

// Memcpy from host input buffers to device input buffers
buffers.copyInputToDevice();

// Run inference once (for warming)
bool status = context->executeV2(buffers.getDeviceBindings().data()); // synronous inference on GPU

// Memcpy from device output buffers to host output buffers
buffers.copyOutputToHost();
```

*Based on sample codes by NVIDIA and ONNX project

Demonstration: Real-time Aliasing Correction Using Deep Learning



(Nahas et al 2023; Ultrasonics)

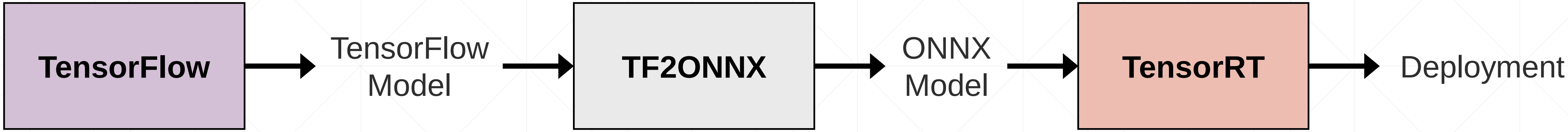


Results:

- Live vector Doppler imaging at 16 fps used for guiding examination
- On-site aliasing-resistant playback at average 60 fps with a screen update rate of 30 fps

Short Summary on TensorRT Implementation

Use ONNX convertor to translate the network into a TensorRT readable format



ONNX

- Able to interface with many different packages
- Open source

TensorRT

- Able to leverage on the dedicated inference hardware on Nvidia GPU
- Make the NN readily available for existing processing pipeline