IEEE UFFC-JS 2024
IEEE Ultrasonics, Ferroelectrics, and Frequency Control Joint Symposium
Taipei Nangang Exhibition Center, Taipei, Taiwan || September 22 – 26, 2024

us4us®

IPPT PAN

University of Waterloo — Concordia cum Veritate

DTU

GPU-Based Ultrasound Signal Processing

Real-Time Speed of Sound Estimation CNN Inferencing Using GPU and Flow Detection using the NPP library

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?

Part III **Image Processing with NPP Libraries**

- How can we apply post-processing image filtering to the data

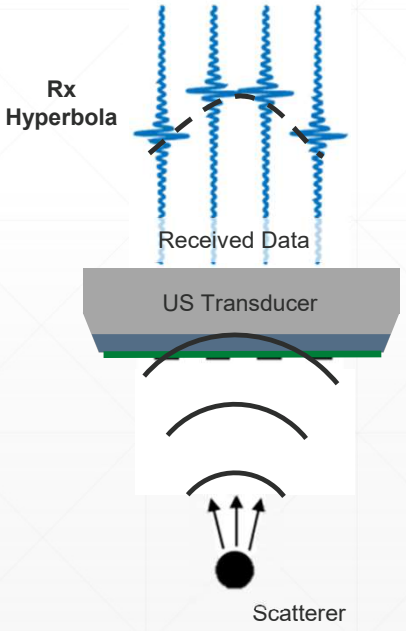
Real-Time Speed of Sound Estimation Using GPU

Di Xiao, Hassan Nahas, Billy Yiu

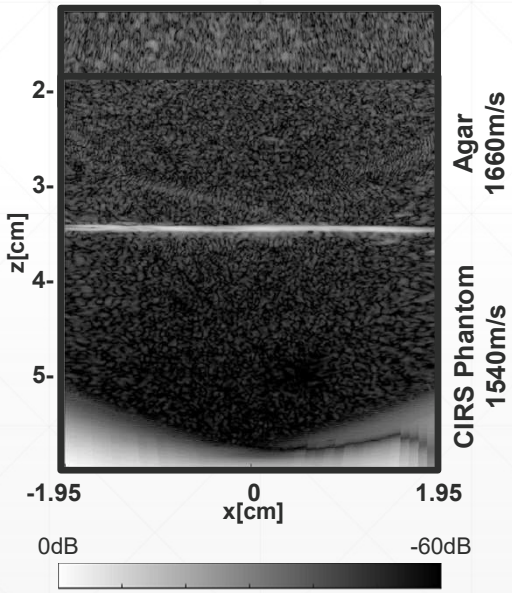
Ultrasound Imaging with Assumed Speed-of-Sound (SoS) Value

Challenge: Loss of image quality when beamformed with wrong SoS

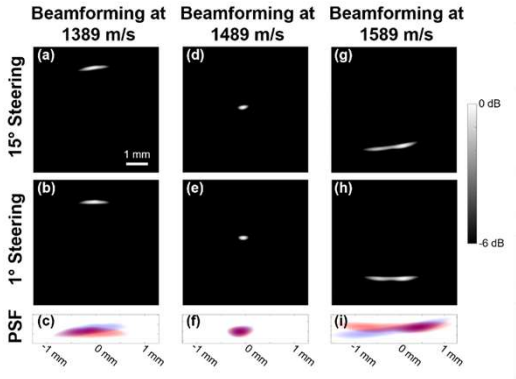
Plane Wave Imaging Forward Model



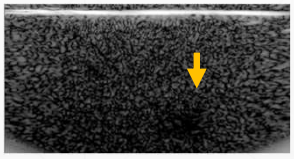
Effect of Different Beamforming SoS Values



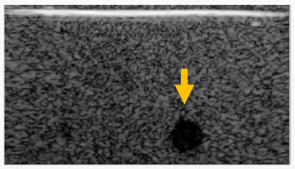
Point Spread Function of Wire Target in Water



Worst Estimate [1450m/s]

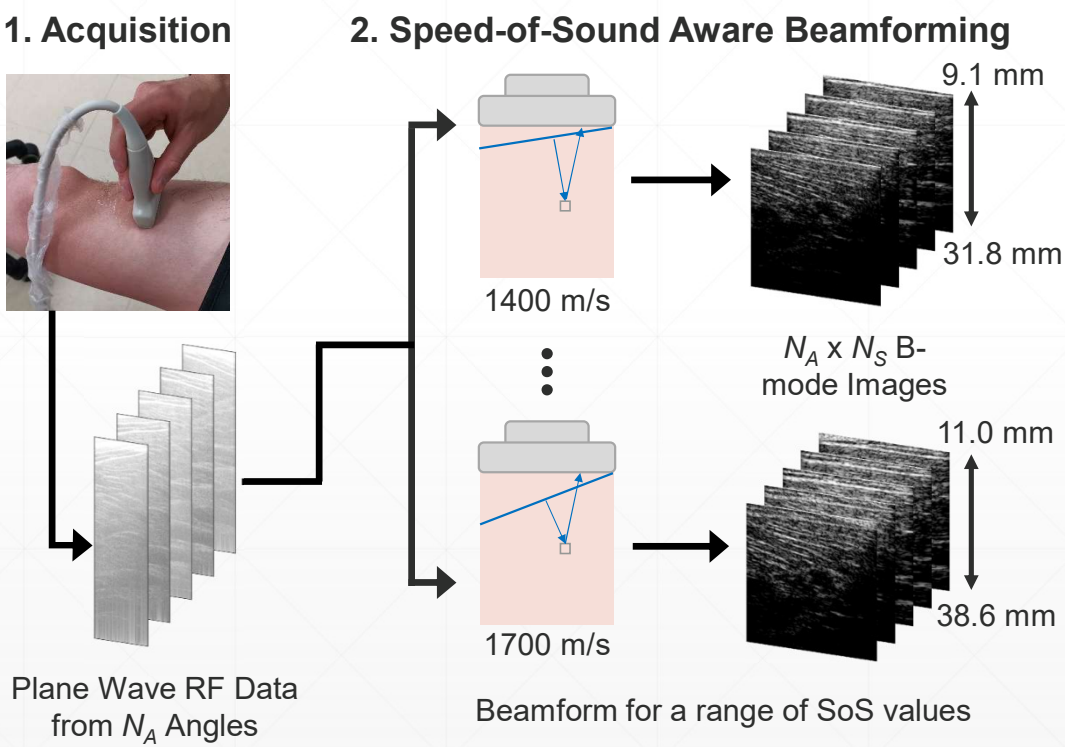


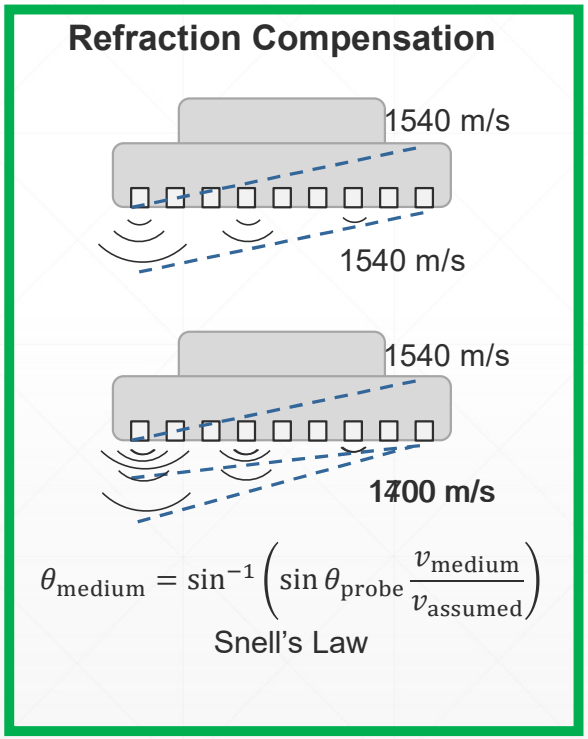
Best Estimate [1630m/s]



Global SoS Estimation: Modified Delay-and-Sum Beamforming

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

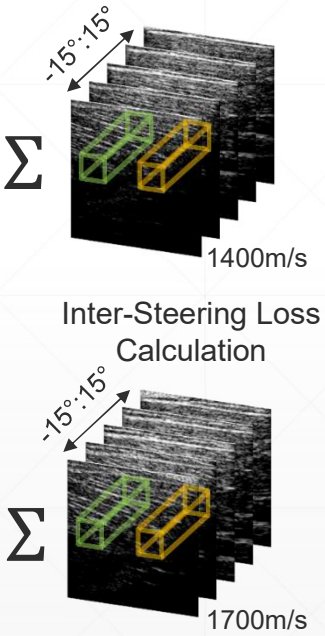




Global SoS Estimation: Identifying Best Beamforming SoS

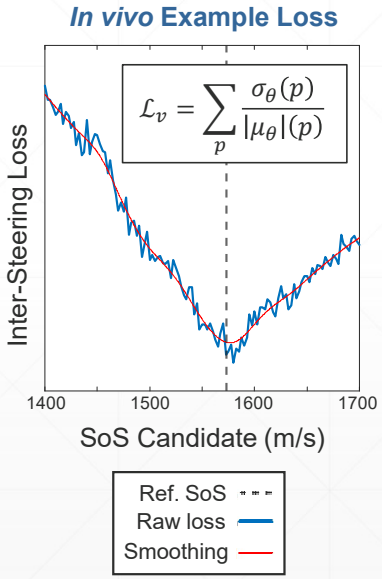
Goal: Provide a real-time estimate of the average (global) SoS of the imaged medium

3. Inter-steering Loss



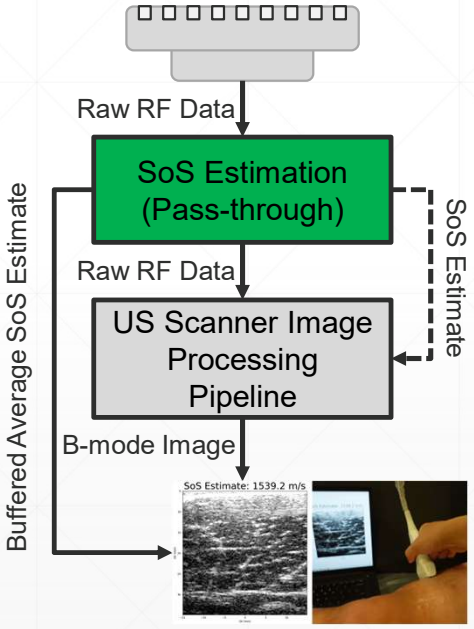
Calculate a loss for each SoS: differences between steering angles

4. SoS Estimation



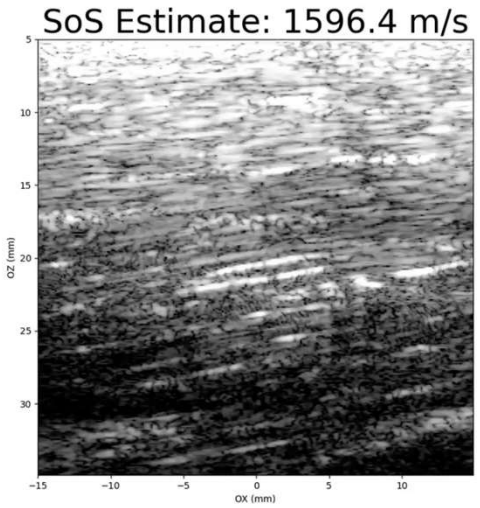
Estimate the SoS by finding the smoothed loss minimum

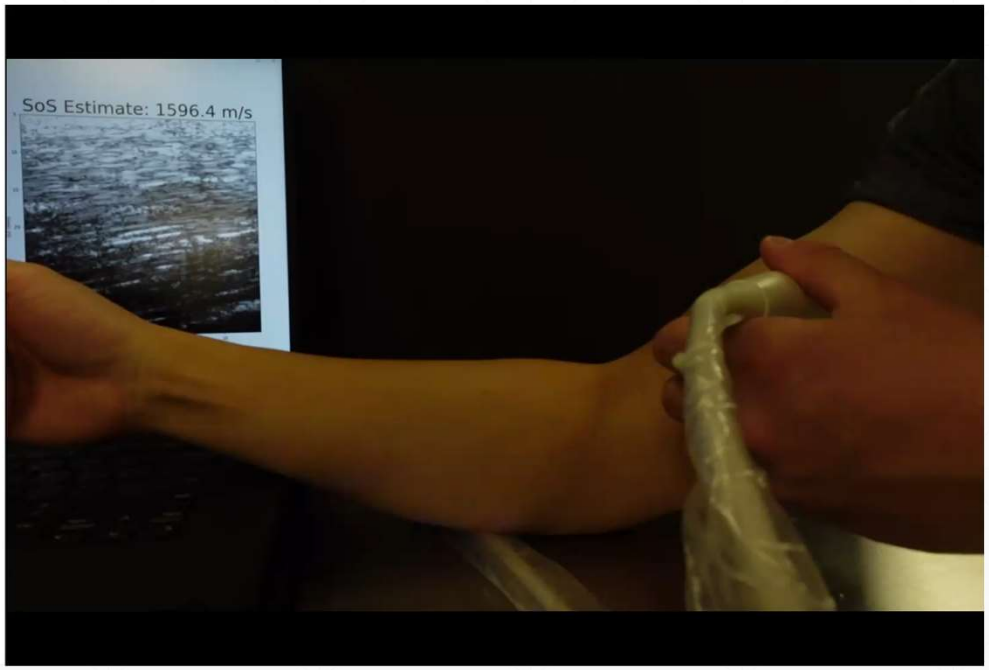
Real-Time Global SoS Estimation



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

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



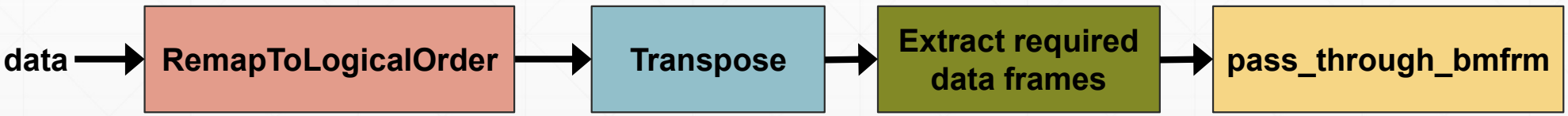


Python Code Structure for Real-Time SoS Estimation

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"
        ))
```

Pipeline: Automatically execute the specified functions sequentially

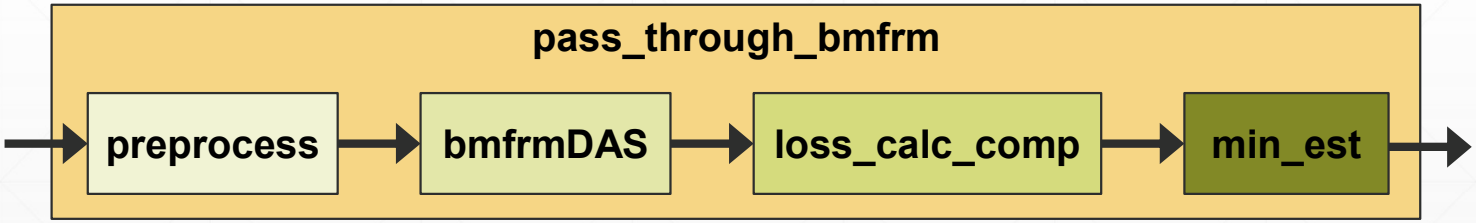


Lambda function: A function handler that the pipeline can call

Inside the Pass_Through_Bmfrm…

```
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
```

Modularize the processing pipeline



These modules can be CuPy built-in or custom functions

Inside the Pass_Through_Bmfrm…

```
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

```
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

```
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

```
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

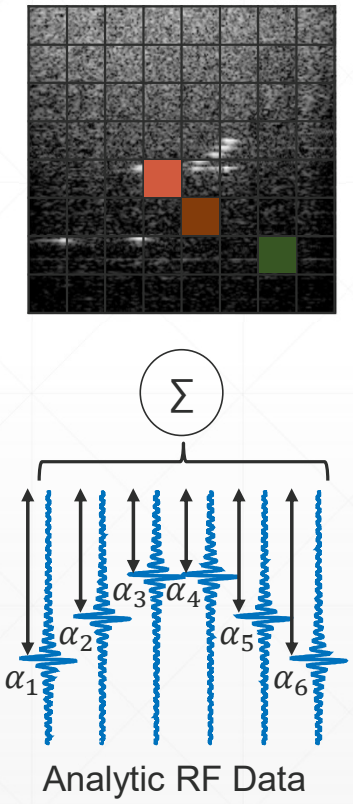
Implementation of the Beamforming Module

```
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
```

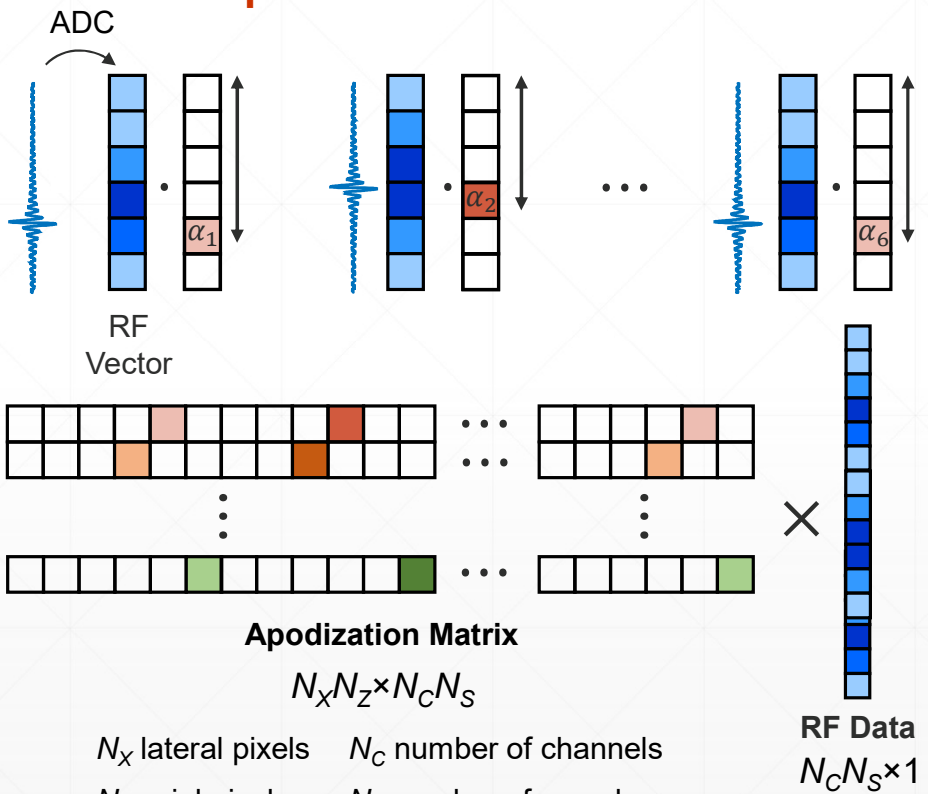
Sparse matrix beamforming – Multiply the RF data matrix with pre-computed delay matrices

Brief Summary of Sparse Matrix Beamforming

Pixel by Pixel Approach



Expression in Matricial Form



N_X lateral pixels N_C number of channels

N_Z axial pixels N_S number of samples

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

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

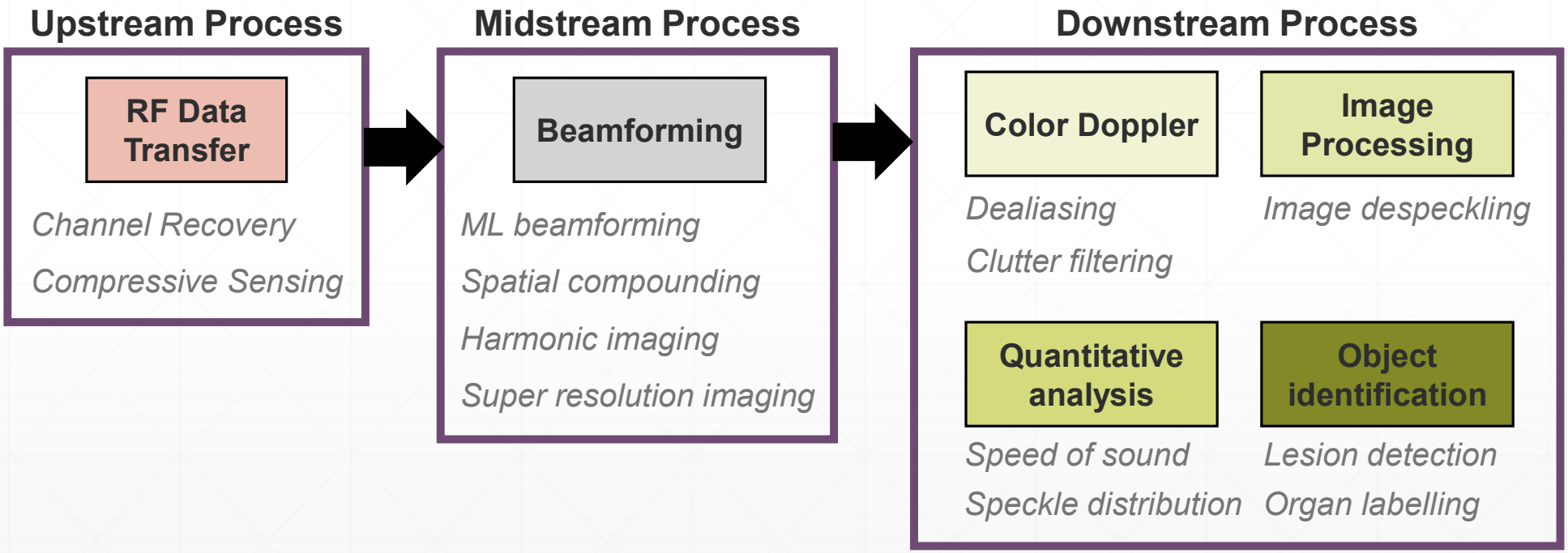
- Greater control on implementation
- Need to implement many things by yourselves
- Optimize the implementation specific to your algorithms for a real-time performance
- Libraries like cuBLAS, cuRAND, cuSOLVER, cuSPARSE, cuFFT, cuDNN are available
- Greater control on display rendering

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



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.

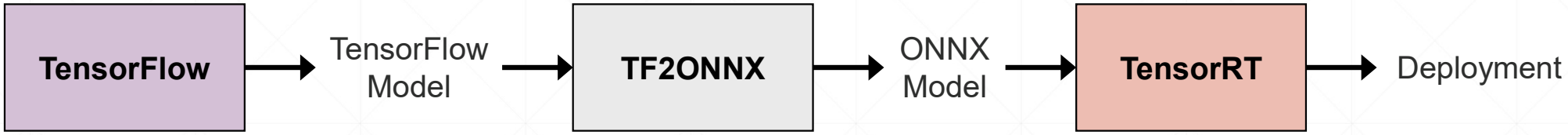
May be used with other deep learning packages (e.g.: Pytorch)

TensorFlow-TensorRT

Direct integration between TF and TRT

TensorRT for Model Deployment

Save the model in TensorRT readable format: ONNX



Python

Trained model in TensorFlow-Python

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

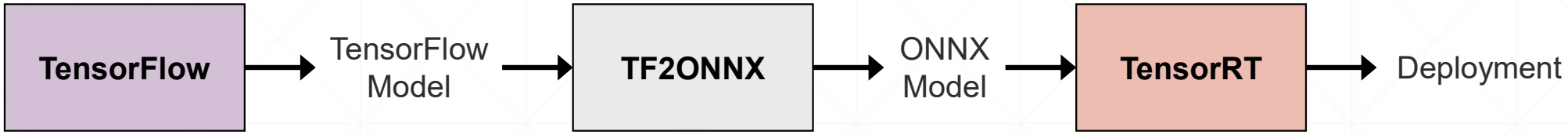
Convert to ONNX

```
# 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



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

```
// 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

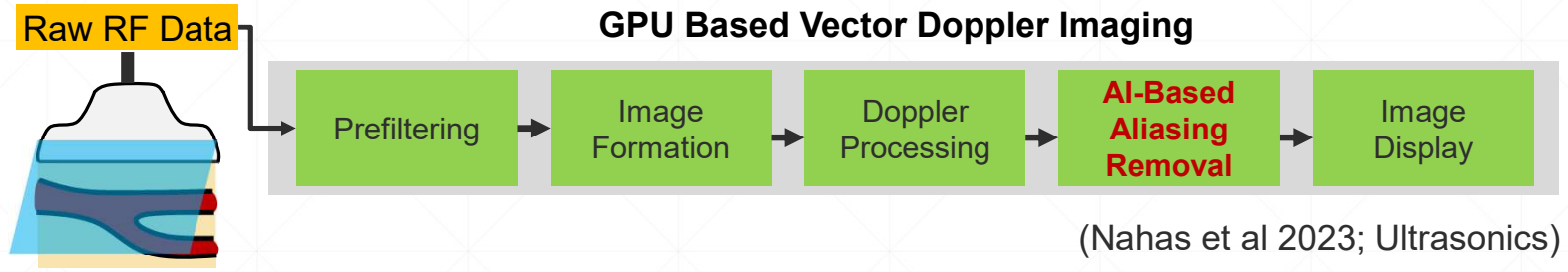
Deploying a CNN Model Trained with MNIST Dataset: A Brief Walkthrough

7. Execute the CNN for inferencing

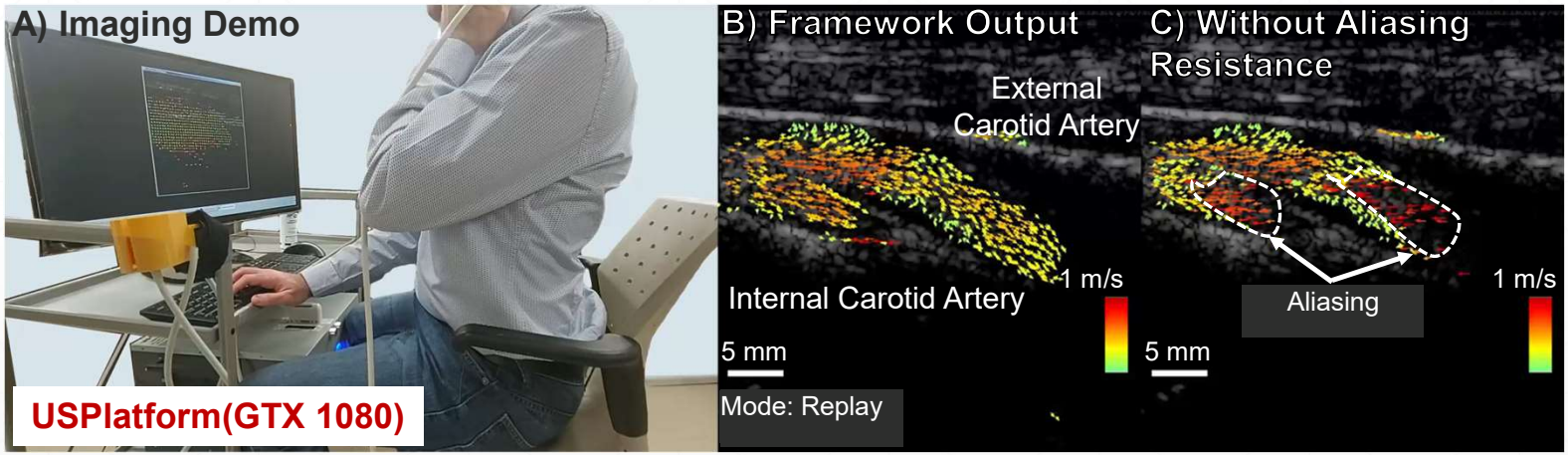
```
// buffers for input/output of model
samplesCommon::BufferManager buffers(mEngine);

// 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)

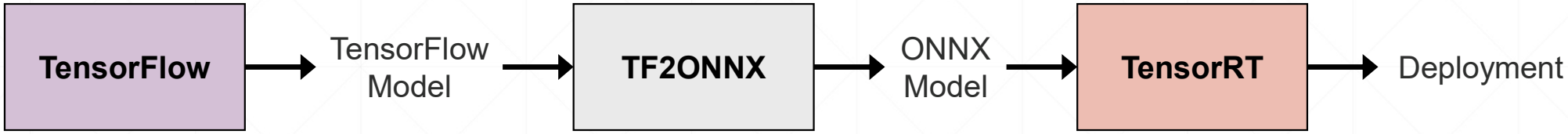


Results:

- 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

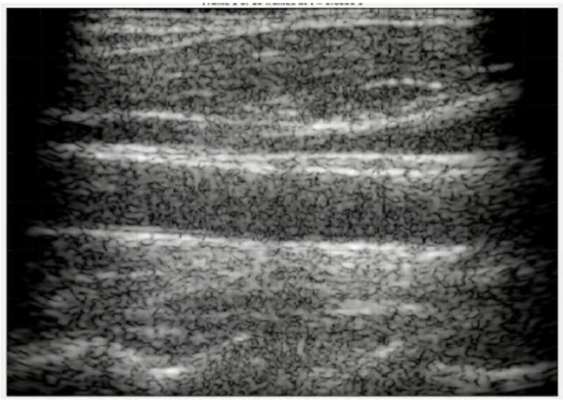
Flow Detection Using NPP Library

Billy Yiu

NVIDIA Performance Primitives (NPP): A Powerful Library for Basic Image Processing

Involved in the later stage of all ultrasound processing pipelines

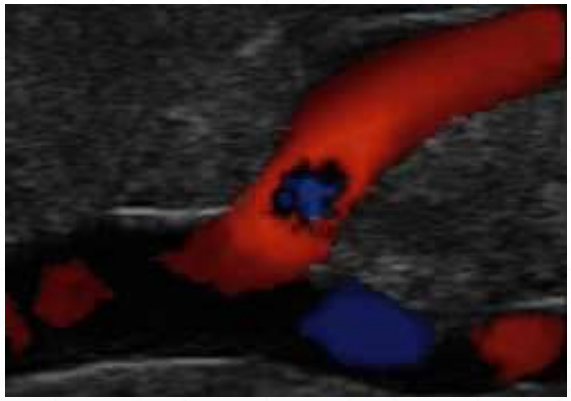
B-Mode Imaging



Speckle Filtering

Scan Conversion

Color Flow Mapping



Flow Detection

Median Filtering

Gaussian Filtering

Image Persistence

A convenient tool to translate Matlab pipeline to C++ pipeline

Flow Detection in Color Doppler Mapping

Objective: Generate a flow-detected mask from the power Doppler map

```
void flowDetection(float* flowDetectedMap, float* powerDopplerMap,
                   int input_size_x, int input_size_z, float threshold)
{
    //Apply a 5x5 mean filter across the image before flow detection
    meanFilter(powerDopplerMap, input_size_x, input_size_z, 5, 5);
    // Apply thresholding to the powerDopplerMap
    flowDetection(flowDetectedMap, powerDopplerMap, threshold);
    // Apply a 7x7 median filter to smooth out the edges and fill the holes
    medianFilter(flowDetectedMap, input_size_x, input_size_z, 7, 7);
    // You can apply other image processing steps to further smooth the flow detected map
    ...
}
```

1. Spatial smoothing of the power Doppler map
2. Generate the flow-detected mask using thresholding
3. Smooth the edges and fill the holes using median filtering

Deploying a Median Filtering Using NPP: A Brief Walkthrough – 1

Encapsulate all the GPU operations into a function

```
void medianFilter(float* input, int input_size_x, int input_size_z, int x, int y)
```

1. Define the median filtering operation

```
// Define the median mask
NppiSize oMaskSize = { y, x };
// Handling the edges
NppiSize oROISize = { input_size_z - oMaskSize.width, input_size_x - oMaskSize.height };
// Define the center of the mask
NppiPoint oAnchor{ oMaskSize.width / 2,oMaskSize.height / 2 };
```

2. Allocate buffers for the filtering operation

```
// Allocate the output and scratch buffer
NppiSize imgSize = { input_size_z, input_size_x };
Npp32f* nppMap; // temporary output
Npp8u* tmpBuffer; // scratch buffer
cudaMalloc((Npp32f**)&nppMap, input_size_z * input_size_x * sizeof(Npp32f));
cudaMalloc((Npp8u**)&tmpBuffer, input_size_z * input_size_x * sizeof(Npp8u) * x*y*2);
```

Need to make a scratch buffer for the median operation

Deploying a Median Filtering Using NPP: A Brief Walkthrough – 2

3. Make a copy of the input image and apply the filter

```
// Make a copy of the input and apply the filter
int status = nppiCopy_32f_C1R((Npp32f*)input, input_size_z * sizeof(Npp32f),
                              nppMap, input_size_z * sizeof(Npp32f), imgSize);

status = nppiFilterMedian_32f_C1R((Npp32f*)(input), input_size_z * sizeof(Npp32f),
    nppMap, input_size_z * sizeof(Npp32f), oROISize, oMaskSize, oAnchor, tmpBuffer);
```

nppiFilterMedian_32f_C1R(*dest_buffer*, *dest_stripe_size*,
src_buffer, *src_stripe_size*, *ROI_size*, *mask_size*, *center_pos*, *scratch_buffer*)

dest_stripe_size: The number of bytes for each column of the destination image buffer
src_stripe_size: The number of bytes for each column of the source image buffer

4. Copy the filtered image to the input image and release buffers

```
// Copy the result to the input image and free the temporary buffers
cudaMemcpy(input, nppMap, sizeof(Npp32f) * input_size_x * input_size_z, cudaMemcpyDeviceToDevice);
cudaFree(nppMap);
cudaFree(tmpBuffer);
```

Deploying a Mean Filtering Using NPP: A Brief Walkthrough – 1

Encapsulate all the GPU operations into a function

```
void meanFilter(float* input, int input_size_x, int input_size_z, int x, int y)
```

1. Define the mean filtering operation

```
// Define the mean mask
NppiSize oMaskSize = { y, x };
// Define the ROI/handle the edges
NppiSize oROISize = { input_size_z - y, input_size_x};
// Define the center of the mask
NppiPoint oAnchor{ oMaskSize.width / 2,oMaskSize.height / 2 };
```

2. Allocate a buffer for the filtering operation

```
// Allocate the temporary buffer
Npp32f* nppMap;
NppiSize imgSize = { input_size_z, input_size_x };
cudaMalloc((Npp32f**)&nppMap, (input_size_x + 2*x) * input_size_z * sizeof(Npp32f));
```

No need to make a scratch buffer as in median filtering

Deploying a Median Filtering Using NPP: A Brief Walkthrough – 2

3. Make a copy of the input image and apply the filter

```
// Make a copy of the input and apply the filter
int status = nppiFilterBox_32f_C1R(nppMap, input_size_z * sizeof(Npp32f), (Npp32f*)input, input_size_z * sizeof(Npp32f), oROISize,
                                   oMaskSize, oAnchor);
if (status != NPP_SUCCESS)
    std::cout << "mean filter failed\n";
```

nppiFilterBox_32f_C1R(*dest_buffer*, *dest_stripe_size*,
src_buffer, *src_stripe_size*, *ROI_size*, *mask_size*, *center_pos*)

dest_stripe_size: The number of bytes for each column of the destination image buffer
src_stripe_size: The number of bytes for each column of the source image buffer

4. Copy the filtered image to the input image and release the buffer

```
// end of mean filtering, go back to the original pipeline
cudaMemcpy(input, nppMap, sizeof(Npp32f) * input_size_x * input_size_z, cudaMemcpyDeviceToDevice);
cudaFree(nppMap);
```

Example of Using the Median and Mean Filtering

Generate a flow-detected mask using the power Doppler map

```
void flowDetection(float* flowDetectedMap, float* powerDopplerMap,
                   int input_size_x, int input_size_z, float threshold)
{
    //Apply a 5x5 mean filter across the image before flow detection
    meanFilter(powerDopplerMap, input_size_x, input_size_z, 5, 5);
    // Apply thresholding to the powerDopplerMap
    flowDetection(flowDetectedMap, powerDopplerMap, threshold);
    // Apply a 7x7 median filter to smooth out the edges and fill the holes
    medianFilter(flowDetectedMap, input_size_x, input_size_z, 7, 7);
    // You can apply other image processing steps to further smooth the flow detected map
    ...
}
```

You can cascade other NPP functions, and they can be called in a similar way

- **Edge detection**
- **Gaussian filtering**
- **Convolutions**

Summary on Using Existing Libraries

1. CuPy for quick deployment
2. ONNX and TensorRT for deep learning applications
3. NPP for color flow detection