supra
-CON

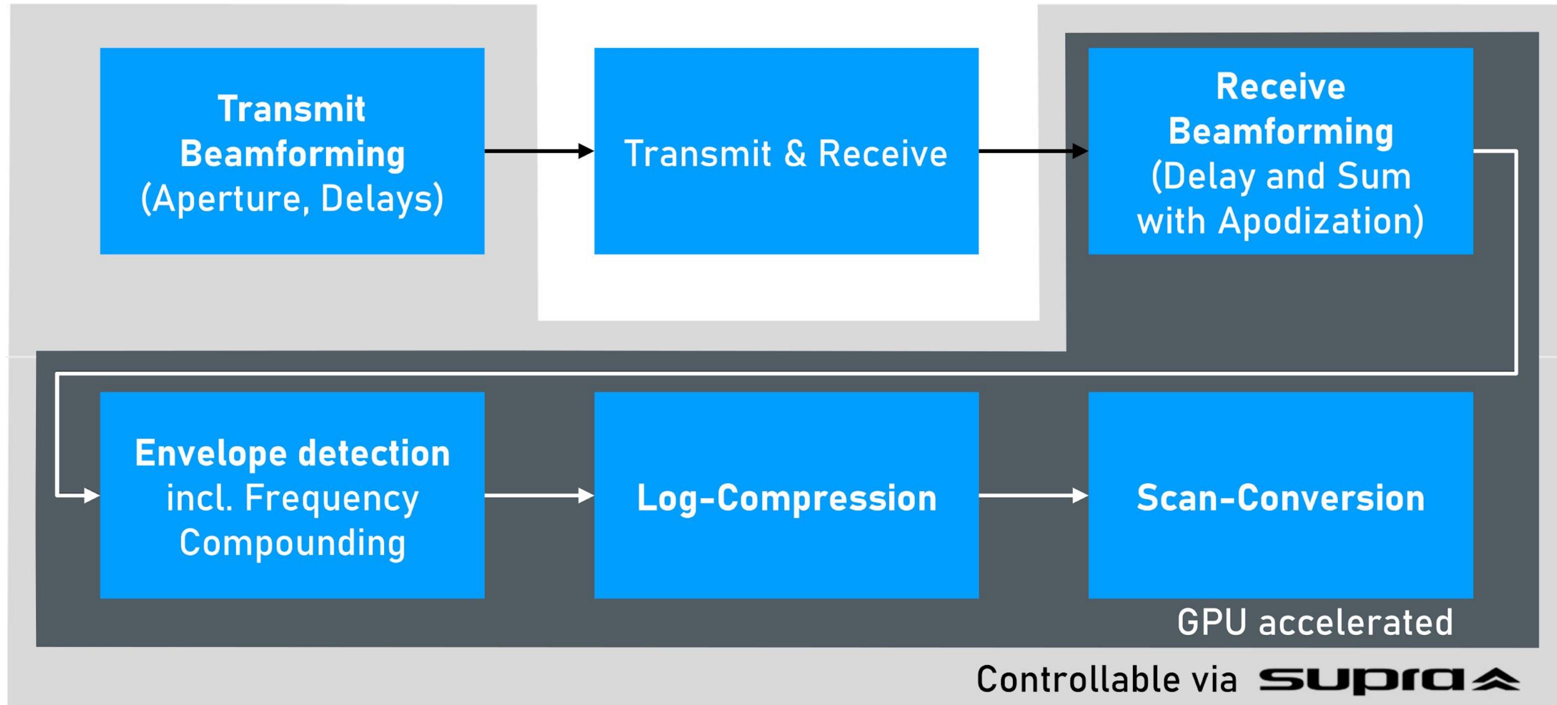
supra

What is SUPRA?

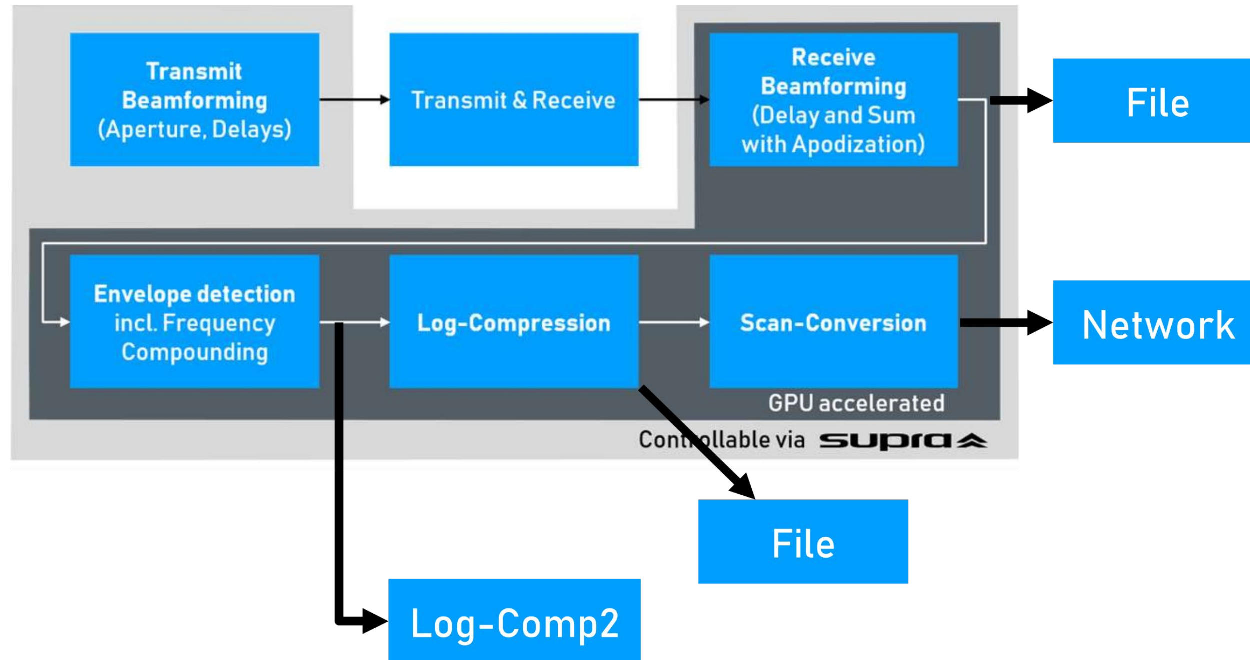
- Open-source pipeline for Ultrasound processing
- From Raw-data to final images
- Software-Defined for flexibility and accessibility
- CUDA-accelerated for real-time application
- Supports 2D and 3D imaging!

https://github.com/IFL-CAMP/supra

General Ultrasound Processing & SUPRA



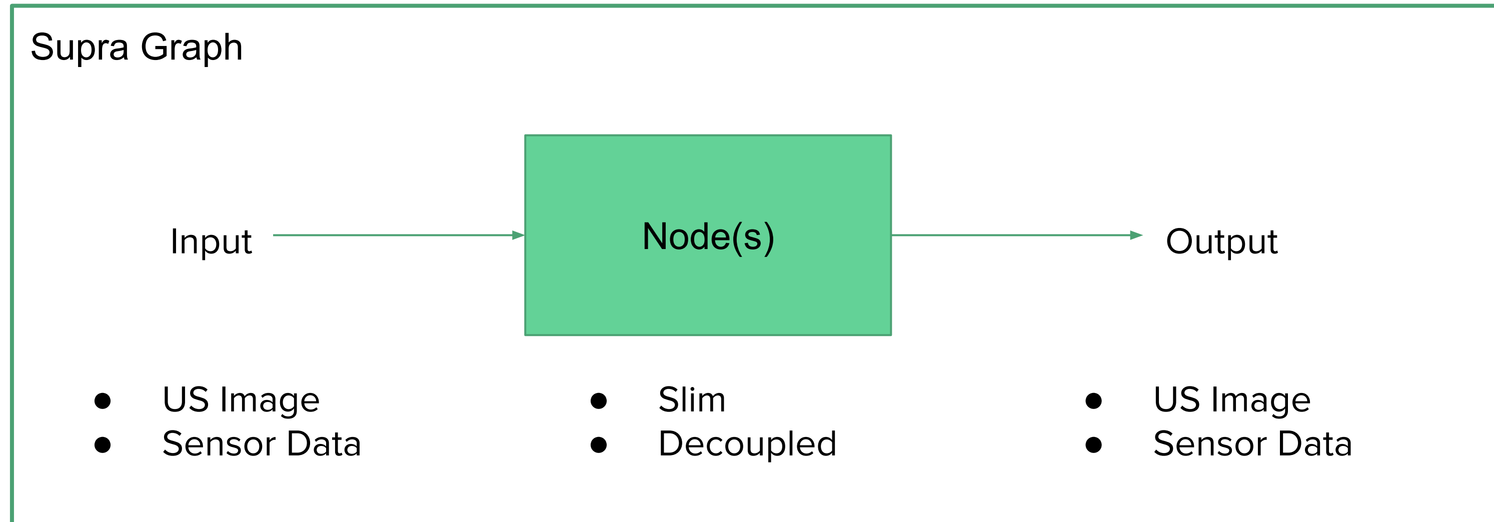
Flexibility in a Software-Defined Framework



SUPRA Design Goals

- Capability to map conventional ultrasound imaging pipeline in software
- Research platform for exploration of new methods
- Flexibility in use and extension
- Easy to use and extend
- Real-time use = High performance

Architecture - Compute Graph



Using SUPRA

What you can do with SUPRA?

- Data / Image acquisition
  - Compile and use it
- Realtime DeepLearning Inference

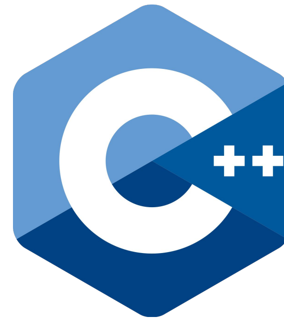
What you can do with SUPRA?

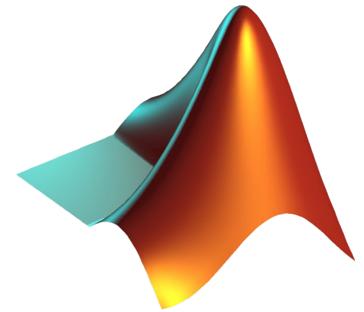
Current:

- Use WrappedPipeline from SupraLib

Upcoming:

- Matlab wrapper
- Python wrapper
- Other?





What you can do with SUPRA?

Add Features:

- New Reconstruction methods
- New Transducer Compatibility
- Advanced Ultrasound Filters
- Clinical/Research interfaces
- Further Tracking / Robotic Integrations
- AI integration

What you can do with SUPRA?

Development on framework itself

- Future architecture changes
- Addition and improvement of features
- Improved abstraction
- Additional GPU acceleration

Usage - Acquisition: Present Functionality

- 2D - Scanline imaging
  - Delay and Sum
  - Minimum Variance
- 3D Ultrasound
- Filters
  - Temporal
  - Bilateral
  - SRAD
- Pytorch integration
- Robotic tracking with ROS

https://github.com/IFL-CAMP/supra/wiki#processing-nodes

Usage - Acquisition: I/O (Cephasonics & Mock-files)

- Input: Open Device Interface
  - Supports Cephasonics *cuSDK*
  - Further support planned
- Input & Output: “Replay” Interface
  - config and mock file as input
- Output: ImFusion Stream
  - Contact ImFusion :)
- Output: MetaImage (MHD)
- Output: IGTL
- Output: ROS messages

https://github.com/IFL-CAMP/supra/wiki#inputs

Usage - Acquisition: Config File

- XML file
- specifies nodes
- specifies input and output
- Loaded at runtime

```xml
<?xml version="1.0" encoding="utf-8"?>
<supra_config>
  <devices>
    <inputs>
      <input type="UltrasoundInterfaceRawDataMock" id="US-Mock">
        <param name="mockDataFilename" type="string">data/linearProbe_IPCAI_128-2_0.raw</param>
        <param name="mockMetaDataFilename" type="string">data/linearProbe_IPCAI_128-2.mock</param>
      </input>
    </inputs>
    <outputs>
      <output type="MetaImageOutputDevice" id="MHD">
        <param name="createSequences" type="bool">1</param>
        <param name="filename" type="string">mhd_output</param>
      </output>
    </outputs>
    <nodes>
      <node type="BeamformingNode" id="BEAM">
        <param name="windowParameter" type="double">0.5</param>
        <param name="windowType" type="string">Hamming</param>
      </node>
      <node type="HilbertFirEnvelopeNode" id="DEMO" />
      <node type="FrequencyLimiterNode" id="LIMIT">
        <param name="maxFrequency" type="double">50</param>
      </node>
```

Usage - Acquisition: Runtime Graph

- Interactive parameter changes
- Dynamic Graph construction
- Load/Save as Config Files

Interfaces

- GUI → Online processing
- Executor → Offline processing
- ROS → “Headless” online processing, control through ROS
- REST interface → Access & change parameters through web API
- WrappedPipeline → Use a SUPRA pipeline in other software

Using - Summary

- Nodes
- Configuration
  - Runtime
  - Config XMLs
- Interfaces

Adding Functionality

Important Data Structures

- SupraGraph
- AbstractNode
- Parameter system
- Containers
- RecordObject
  - USImage
  - USRawData
- USImageProperties
- RxBeamformerParameters

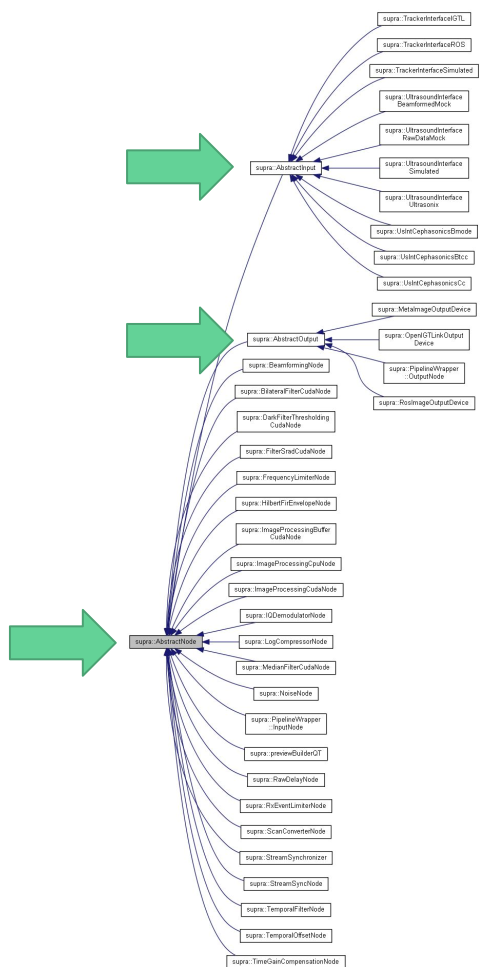
➜ Doxygen documentation (can be activated in CMake!)

SupraGraph

- The collection of Nodes and Connections
- Represents the “pipeline”
- Used by interfaces (frontends? GUI etc.) to access and modify nodes
- Usually only one
- Can read and write configuration XML
    - Nodes
    - Parameters
    - Connections

AbstractNode

- Base class for all nodes
  (Special base classes AbstractInput and AbstractOutput predefine some behavior)
- Contains the parameter definitions and values
  - ValueRangeDictionary
  - ConfigurationDictionary
- Defines interface to access and change them



AbstractNode

- Base class for all nodes
  (Special base classes AbstractInput and AbstractOutput pre-define some behavior)
- Contains the parameter definitions and values
  - ValueRangeDictionary
  - ConfigurationDictionary
- Defines interface to access and change them

Need to overwrite functions:

```
getNumInputs()

getNumOutputs()

getInput(size_t index)

getOutput(size_t index)
```

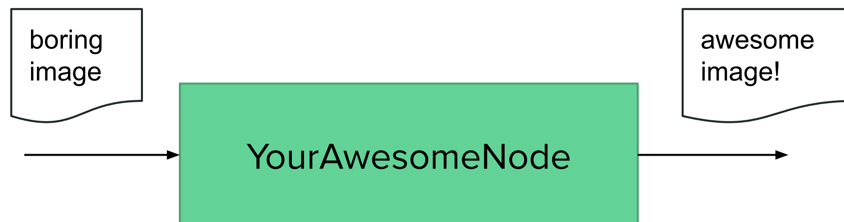
Mostly generic

```
configurationChanged()

configurationEntryChanged(
    string configKey)
```

Node specific logic to react to new parameter values

How do Nodes do stuff?



How is YourAwesomeAlgorithm called?

Usual case: 1-Input, 1-Output

```
YourAwesomeNode::YourAwesomeNode(...)
{
    // Create the underlying tbb node for handling the message passing.
    m_node = unique_ptr<NodeTypeQueueing>(new NodeTypeQueueing(graph, 1,

        [this](shared_ptr<RecordObject> inObj) -> shared_ptr<RecordObject> {
            return YourAwesomeAlgorithm(inObj); }

    ));
```

Callback on arrival of input message!

Parameter System: ValueRangeDictionary

Every node has one!

Defines the parameters including

- Type
- Name (key)
- Display name
- Valid values
    - **Unrestricted**
    - Range
    - Discrete

```
template<typename ValueType>
void ValueRangeDictionary::set(
    const string & key,
    const ValueType & defaultValue,
    const string & displayName )
```

Example:

```
m_valueRangeDictionary.set<bool>(
    "additiveGaussCorrelated",
    false,
    "additive Gauss Correlated");
```

Parameter System: ValueRangeDictionary

Every node has one!

Defines the parameters including

- Type
- Name (key)
- Display name
- Valid values
    - Unrestricted
    - **Range**
    - Discrete

```
template<typename ValueType>
void ValueRangeDictionary::set(
    const string & key,
    const ValueType & lowerBound,
    const ValueType & upperBound,
    const ValueType & defaultValue,
    const string & displayName)
```

Example:

```
m_valueRangeDictionary.set<double>(
    "speckleScale",
    0.0, 100.0, 25.0,
    "Speckle Scale");
```

Parameter System: ValueRangeDictionary

Every node has one!

Defines the parameters including

- Type
- Name (key)
- Display name
- Valid values
    - Unrestricted
    - Range
    - **Discrete**

```
template<typename ValueType>
void ValueRangeDictionary::set(
    const string & key,
    const vector<ValueType> & value,
    const ValueType & defaultValue,
    const string & displayName)
```

Example:

```
m_valueRangeDictionary.set<DataType>(
    "outputType",
    { TypeFloat, TypeUint8, TypeInt16 },
    TypeFloat,
    "Output type");
```

Parameter System: ConfigurationDictionary

Contains the actual parameter values

Used to get and set the params

Example:

```
m_speckleScale =
    m_configurationDictionary.
     get<double>("speckleScale");

m_subArraySize =
    m_configurationDictionary.
     get<uint32_t>("subArraySize");
```

Container<T>

Thin wrapper around memory block

Handles allocation / deallocation

Follows RAII (Resource acquisition is initialization)

Keeps track:

- Location (Host / GPU)
- Associated CUDA stream
- Buffer size
- Data type

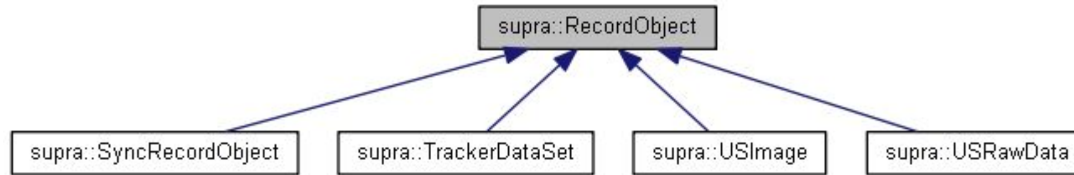
Hints:

- Use when you need GPU memory
- Also for moving data GPU ↔ Host

```
shared_ptr<Container<float> > data = ...;
if (!data->isGPU())
{
    data = make_shared<Container<float> >
     (LocationGpu, *data);
}
```

RecordObject

- The base class for data passing through the graph
- Contains timestamp



USImage

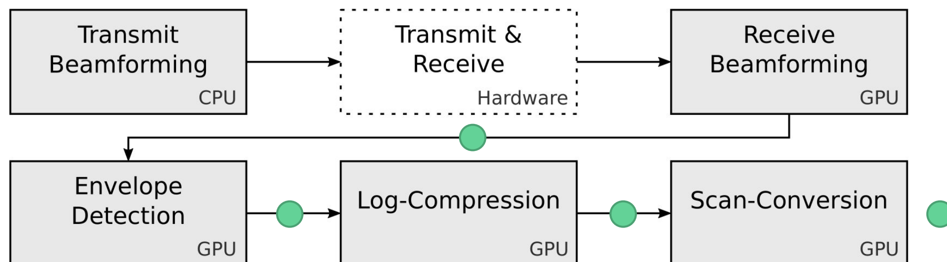
Describes an ultrasound image after beamforming

- Size
- USImageProperties
- Image data Container

Arbitrary element (“pixel”) datatypes

Possible states:

- Beamformed RF
- Envelope detected
- PreScan (Logcompressed scanline data)
- Scanconverted



USRawData

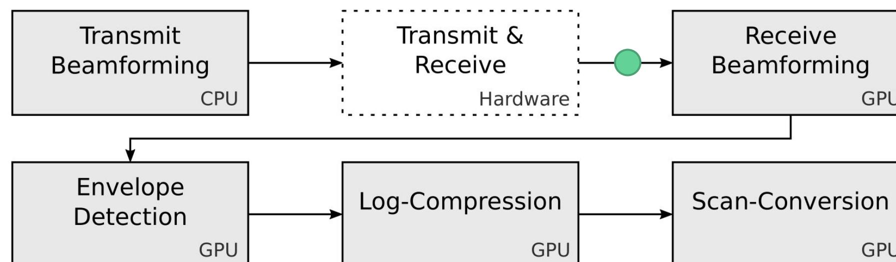
Describes an ultrasound image before beamforming

- Size i.e. #Scanlines, #elements, #temporal samples
- Sampling frequency
- USImageProperties
- RXBeamformerParameters
- Raw data Container

Arbitrary element (“Sample”) datatypes

Possible states:

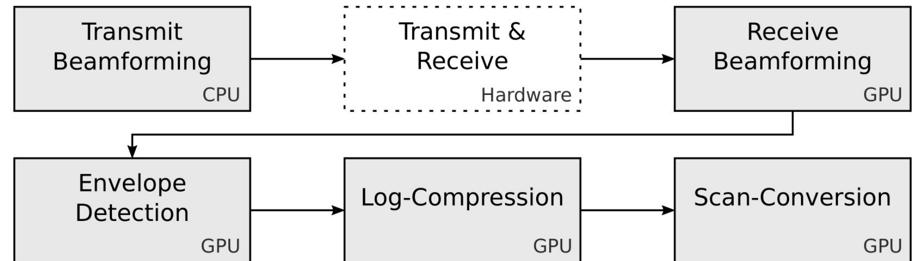
- Raw data / Channel data
- Delayed Raw



USImageProperties

Contains all the metadata to describe the meaning of an ultrasound image

- State (e.g. beamformed RF)
- Transducer type
- # Scanlines & geometry
- # Samples
- Acquisition depth



RXBeamformerParameters

The geometric, temporal and other low-level parameters to facilitate receive beamforming

- Location & direction of scanlines
- Transducer element locations
- Speed of sound
- Elements used for each transmit / receive event

Example Nodes

Perform same task

“Scale image intensities”

in slightly different ways

- ImageProcessing**Cpu**Node
- ImageProcessing**Cuda**Node
- ImageProcessing**BufferCuda**Node

Highly recommended!

Adding Nodes - Summary

- 1-Input, 1-Output is easy ➔ Callback with your function
- Example nodes are good basis
- Use Containers!
- Make everything a parameter
- Use Buffer2/3 instead of manual index calculation

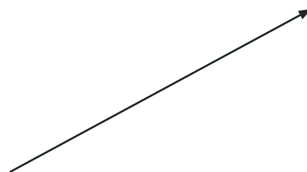
CUDA-basics in two slides (1)

CPU 2D operation

```
for (size_t y = 0; y < height; y++) {
  for (size_t x = 0; x < width; x++) {
    WorkType inPixel =
        pInputImage[x + y*width];
    WorkType value = inPixel * m_factor;
    pOutputImage[x + y*width] = value;
} }
```

CUDA 2D operation

```
dim3 blockSize(32, 4, 1);
dim3 gridSize(
  (size.x + blockSize.x - 1) / blockSize.x,
  (size.y + blockSize.y - 1) / blockSize.y);
processKernel <<<gridSize, blockSize, 0,
inImageData->getStream() >>> (
  inImageData->get(),
  size,
  factor,
  outImageData->get());
cudaSafeCall(cudaPeekAtLastError());
```



Kernel call

CUDA-basics in two slides (2)

"Equivalent" loops for kernel

implicit in kernel call, order unknown!

```
for (blockIndex.x = 0; blockIndex.x < gridSize.x; blockIndex.x++) {
 for (blockIndex.y = 0; blockIndex.y < gridSize.y; blockIndex.y++) {
  for (threadIndex.x = 0; threadIndex.x < blockDim.x; threadIndex.x++) {
   for (threadIndex.y = 0; threadIndex.y < blockDim.y; threadIndex.y++) {
      size_t x = blockDim.x*blockIdx.x + threadIdx.x;
      size_t y = blockDim.y*blockIdx.y + threadIdx.y;
      if (x < width && y < height)
      {
         WorkType inPixel = pInputImage[x + y*width];
         WorkType value = inPixel * m_factor;
         pOutputImage[x + y*width] = value;
      }
} } } }
```

Code in kernel function

CUDA tips

- Start from example CUDA node!
- Ensure all containers are in GPU memory

```
data->isGPU()
```

- Avoid GPU synchronization where possible
    - Except for debugging
- Add error checking after every kernel call

```
cudaSafeCall(cudaPeekAtLastError());
```

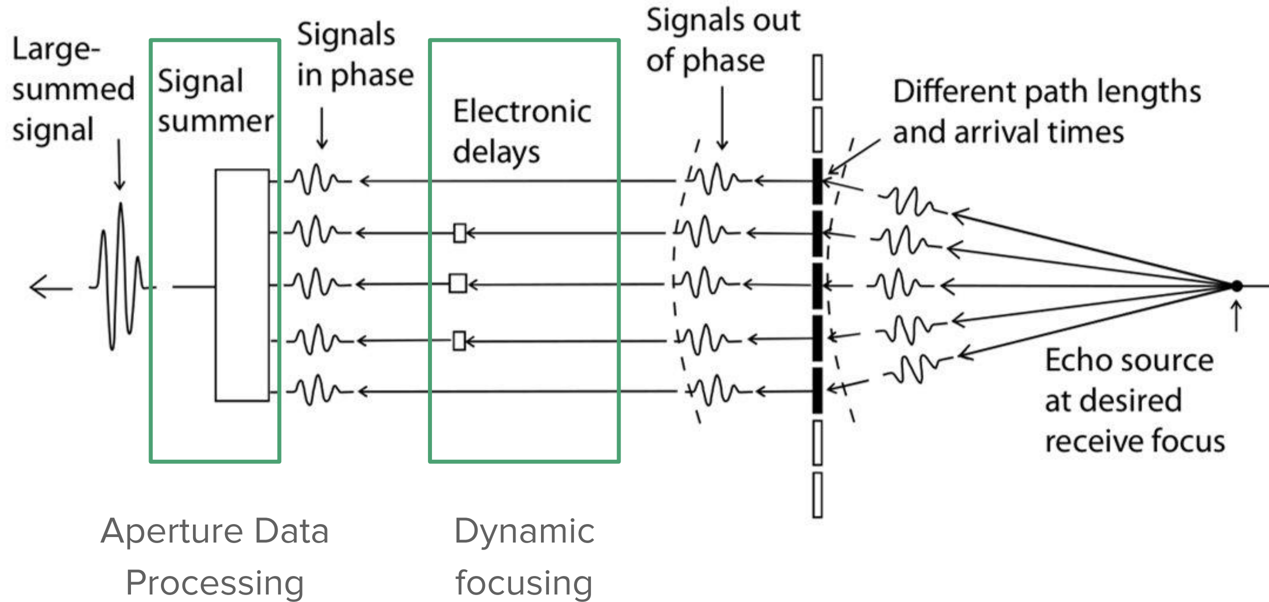
- Don’t worry about optimization at the beginning

Beamforming in SUPRA

Beamforming in two slides (1)

Hoskins, P. R., Martin, K. R., & Thrush, A. R. (2019). *Diagnostic Ultrasound: physics and equipment* (3rd ed.). page 8: CRC Press.

Beamforming in two slides (2)



From Hoskins, P. R., Martin, K. R., & Thrush, A. R. (2019). *Diagnostic Ultrasound: physics and equipment* (3rd ed.). page 49: CRC Press.

Beamforming: what I left out

- Transmit beamforming (TX)
- Acoustic properties of tissue
- Acoustic interactions
- and more…

See “Diagnostic Ultrasound: Physics and Equipment (3rd ed.)” for more

Beamforming in SUPRA

- Focusing and Aperture data processing combined
  - Delay-and-Sum
  - Coherence Factor weighting
- Focusing and Aperture data processing separate
  - Minimum Variance
  - DeepFormer

The Future

Architecture / Future Development goals

Architectural

Functionality

Compatibility

Architecture / Future Development goals

Architectural

- Option for grouping Parameters
  - apply several parameters at the same time while ensuring valid parameter state at all times
- Transducer-specification in files instead of code

Architecture / Future Development goals

Functionality

- Plane wave / Diverging wave (2D & 3D)
- Ultrasound File Format (UFF) input and output (HDF5 based)
- Pre-beamformed processing
- Increase modularity in fused Rx beamformer

Architecture / Future Development goals

Compatibility

- Wider variety of compatible devices
- “Clinical” interfaces via REST-API

Roadmap: Proposal for v0.1

- Current state
- Extended documentation
  - Improve Doxygen “low-level”
  - High-level documentation (“Getting started, using…”)
- Code file structure rework

Roadmap: Proposal for v0.2

- Complete REST-API
- MATLAB Interface
- Mock files replaced with HDF5
- Support for Cephasonics curvi-linear probe

Architecture / Future Development goals

What do you think is missing?