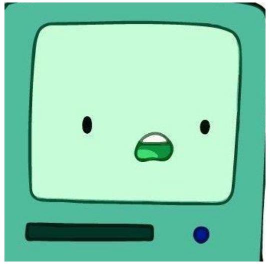
Day Final

Team 8 ELSA Robotics

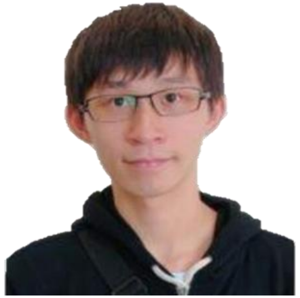
OpenACC
More Science, Less Programming

Team Members

Mentors
From NVIDIA



Frank Lin



Johnson Sun

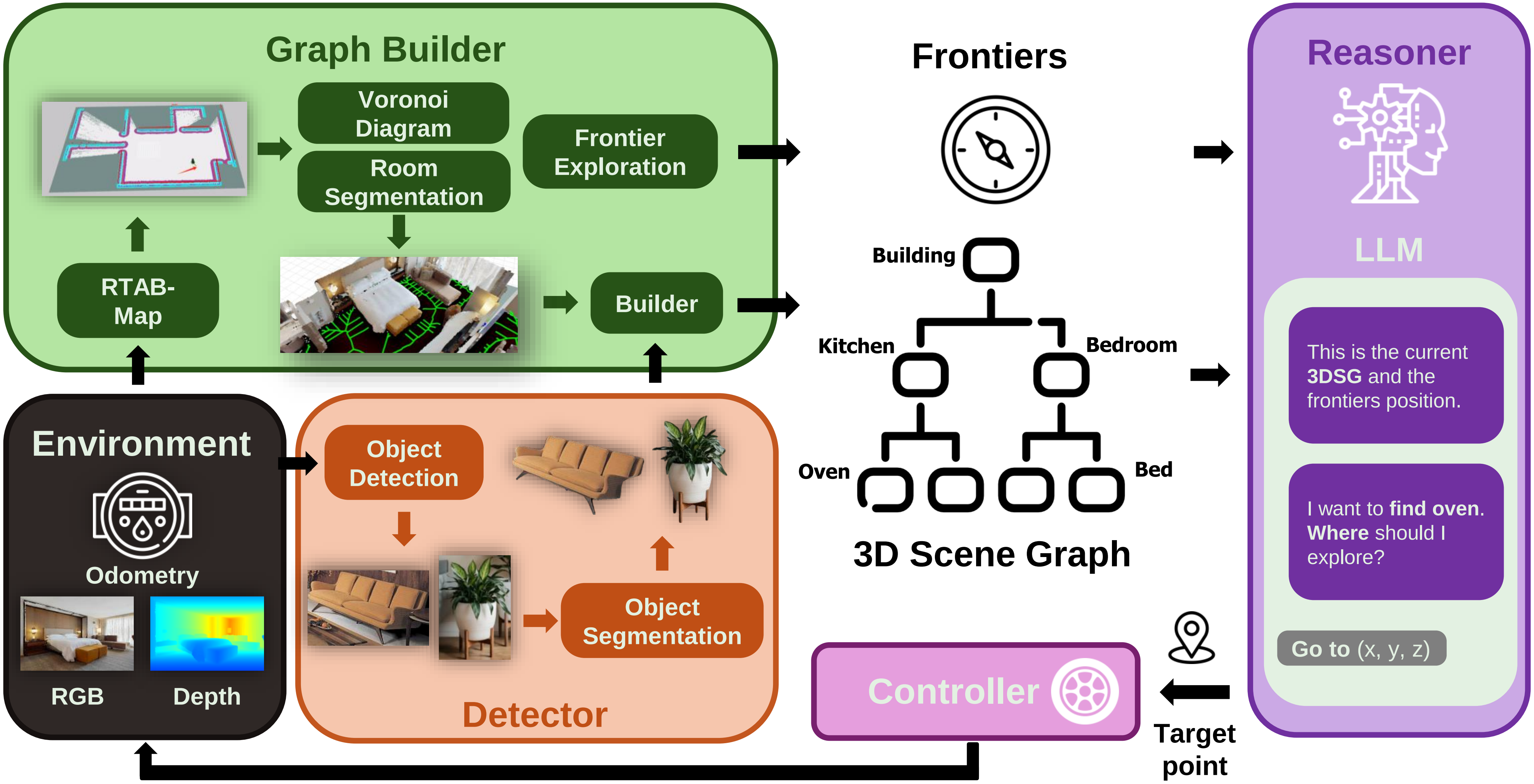
Team Members
From NTHU ELSA

Yu-Zhong Chen

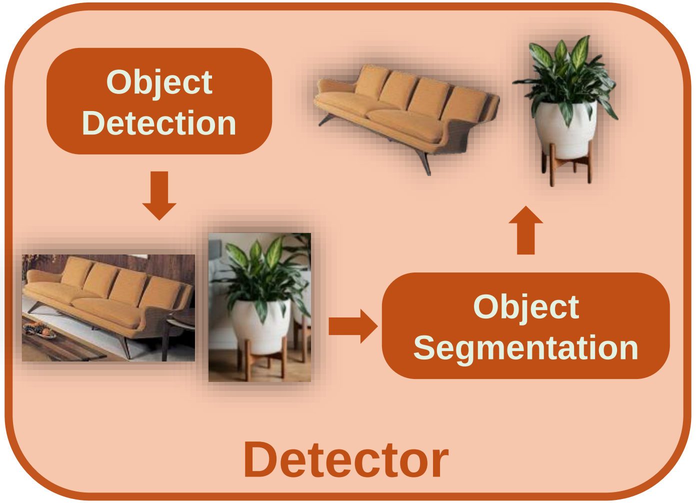
Yung-Shun Zhan

Chon-Hang Lam

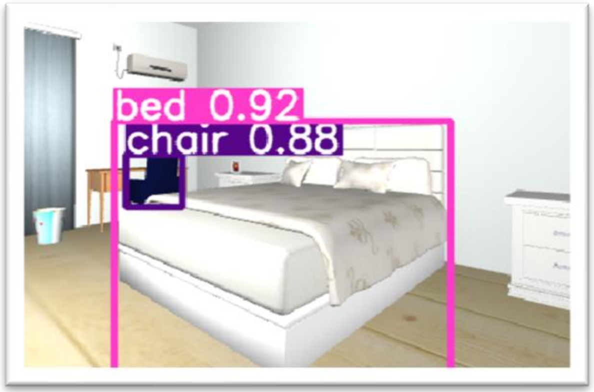
Architecture

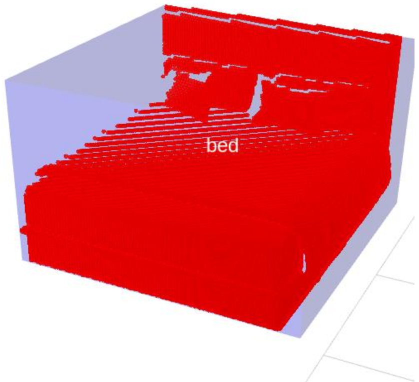


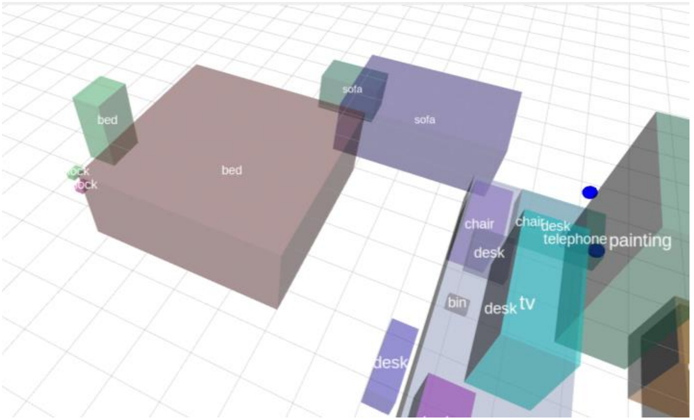
Object-Level



- 3D position of objects
- YOLO-World: Open-set object detector
- Segment object mask by SAM
- Project 2D bounding box to 3D by depth information







Total Speedup

Original pipeline

0.5 fps

Optimized pipeline

5.5 fps

Faster 11x

Optimized Objective

- (1) Depth to Point Cloud
- (2) YoloWorld

Evolution and Strategy

(1) Depth to Point Cloud

```python
for i in range(len(points)):
    # Transform the 2D point to a 3D point in map frame
    map_point = np.array([0.0, 0.0, 0.0, 1.0], dtype=np.float32)
    map_point[0] = (points[i][1] - cx) * depth[i] / fx
    map_point[1] = (points[i][0] - cy) * depth[i] / fy
    map_point[2] = depth[i]

    # Apply the transformation
    map_point = np.dot(transform, map_point)
```

Original

- Convert depth point to 3D
- Dot product for **each point**

$$([p_{i1} \quad p_{i2}] - [c_x \quad c_y]) \times \begin{bmatrix} \frac{d_i}{f_x} & 0 \\ 0 & \frac{d_i}{f_y} \end{bmatrix}$$

- Need to optimize for-loop

Evolution and Strategy

(1) Depth to Point Cloud - Optimization 1 / 3

```
# Precompute constants for the coordinate transformation
x_offsets = (points[:, 1] - cx) / fx
y_offsets = (points[:, 0] - cy) / fy

# Construct 3D points before transformation
map_points = np.vstack((
    x_offsets * depth,
    y_offsets * depth,
    depth,
    np.ones_like(depth),
)).T

# Apply the transformation matrix
map_points = (transform @ map_points.T).T
```

Vectorization

- Calculate point offset for input image (**constant** part)
- Construct map point matrix
- Perform dot product for **all the point** in the matrix

Evolution and Strategy

(1) Depth to Point Cloud - Optimization 2 / 3

```
# Precompute constants for the coordinate transformation
x_offsets = (points[:, 1] - cx) / fx
y_offsets = (points[:, 0] - cy) / fy

# Transfer data to GPU
depth = cp.asarray(depth)
x_offsets = cp.asarray(x_offsets)
y_offsets = cp.asarray(y_offsets)
transform = cp.asarray(transform)

# Construct 3D points before transformation
map_points = cp.vstack((
    x_offsets * depth,
    y_offsets * depth,
    depth,
    cp.ones_like(depth),
)).T

# Apply the transformation matrix
map_points = (transform @ map_points.T).T
```

Vectorization with CuPy

- Replace Numpy with CuPy

Evolution and Strategy

(1) Depth to Point Cloud - Optimization 3 / 3

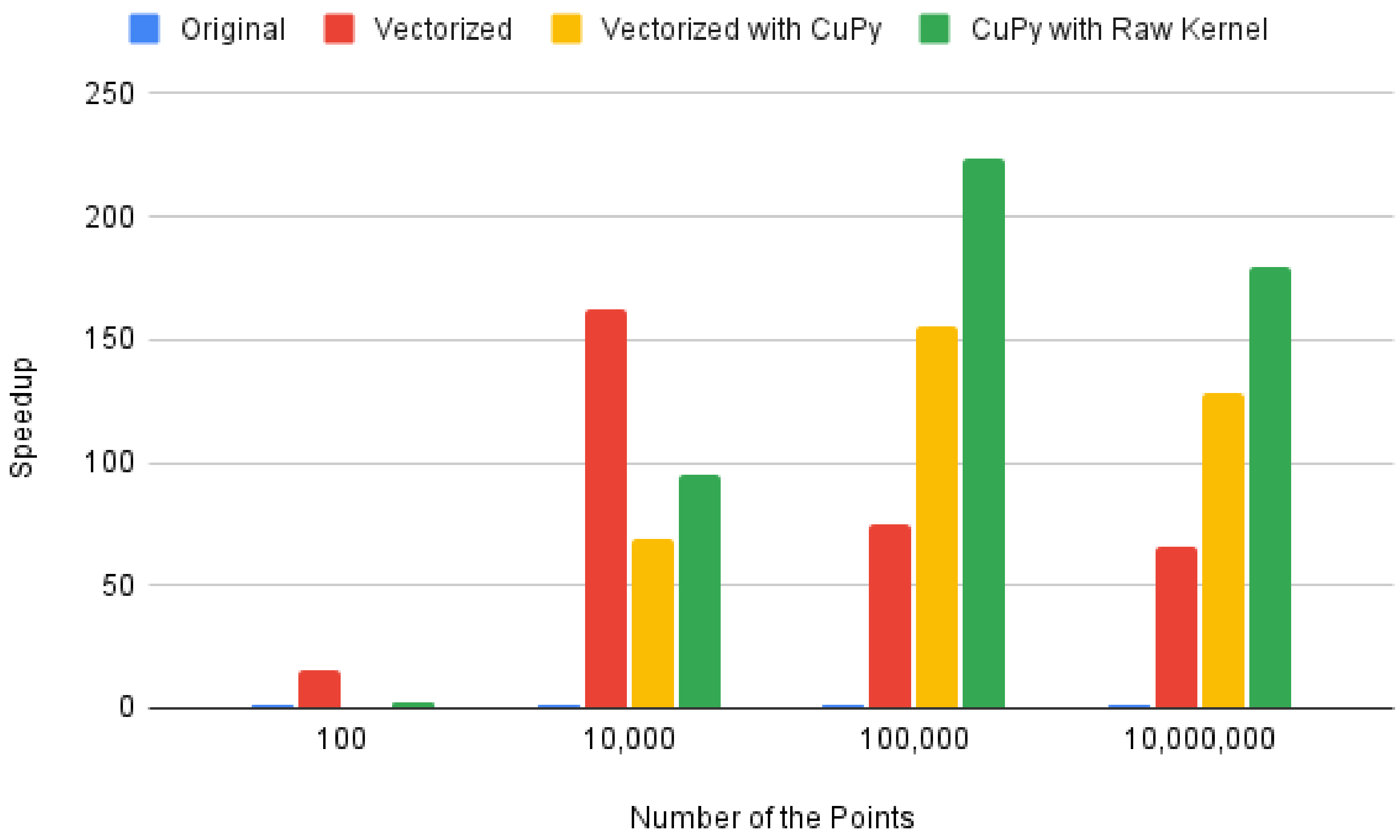
```
# CUDA kernel
kernel_code = """
extern "C" __global__
void Transform2DPointsToMap(const int* points, const float* depth, ...) {
    // Get the index of the point
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    float point[4] = {0.0, 0.0, 0.0, 1.0};
    point[0] = (points[idx * 2 + 1] - cx) * depth[idx] / fx;
    point[1] = (points[idx * 2] - cy) * depth[idx] / fy;
    point[2] = depth[idx];

    // Dot product
    float map_point[4] = {0.0, 0.0, 0.0, 1.0};
    for (int i = 0; i < 4; i++) {
        map_point[i] =
            transform[i * 4 + 0] * point[0] +
            transform[i * 4 + 1] * point[1] +
            transform[i * 4 + 2] * point[2] +
            transform[i * 4 + 3] * point[3];
    }

    // Store the result
    map_points[idx * 3 + 0] = map_point[0];
    map_points[idx * 3 + 1] = map_point[1];
    map_points[idx * 3 + 2] = map_point[2];
}
"""

# Compile the CUDA kernel
module = cp.RawModule(code=kernel_code, backend="nvrtc")
```

CuPy with Raw Kernel

- Optimized the orignal for-loop with raw kernel

Evolution and Strategy

Result (Speedup)



Evolution and Strategy

(2) YoloWorld

Inference Time

PyTorch → ONNX → TensorRT

206.942 ms | 179.633 ms | 71.153 ms

Faster 2.9x

Evolution and Strategy

(2) YoloWorld - Before optimization

```python
# Allocate Memory
d_input = cuda.mem_alloc(1 * 3 * 640 * 640 * 4)
d_output = cuda.mem_alloc(1 * 8400 * 13 * 4)
np_output = np.empty(1 * 8400 * 13, dtype=np.float32)

# Set Input and Output Addresses
context.set_tensor_address(tensor_names[0], int(d_input))
context.set_tensor_address(tensor_names[1], int(d_output))

# Inference
cuda.memcpy_htod_async(d_input, image, stream) # Transfer data to device
context.execute_async_v3(stream.handle) # Execute model
cuda.memcpy_dtoh_async(np_output, d_output, stream) # Transfer predictions back to host

# Create Torch Tensor from NumPy
torch_output = torch.from_numpy(output).to("cuda:0")

# Postprocess ...
```

GPU

CPU

- Memory allocation

Evolution and Strategy

(2) YoloWorld - Before optimization

```python
# Allocate Memory
d_input = cuda.mem_alloc(1 * 3 * 640 * 640 * 4)
d_output = cuda.mem_alloc(1 * 8400 * 13 * 4)
np_output = np.empty(1 * 8400 * 13, dtype=np.float32)

# Set Input and Output Addresses
context.set_tensor_address(tensor_names[0], int(d_input))
context.set_tensor_address(tensor_names[1], int(d_output))

# Inference
cuda.memcpy_htod_async(d_input, image, stream) # Transfer data to device
context.execute_async_v3(stream.handle) # Execute model
cuda.memcpy_dtoh_async(np_output, d_output, stream) # Transfer predictions back to host

# Create Torch Tensor from NumPy
torch_output = torch.from_numpy(output).to("cuda:0")

# Postprocess ...
```

- Memory allocation
- Setting address

Evolution and Strategy

(2) YoloWorld - Before optimization

```python
# Allocate Memory
d_input = cuda.mem_alloc(1 * 3 * 640 * 640 * 4)
d_output = cuda.mem_alloc(1 * 8400 * 13 * 4)
np_output = np.empty(1 * 8400 * 13, dtype=np.float32)

# Set Input and Output Addresses
context.set_tensor_address(tensor_names[0], int(d_input))
context.set_tensor_address(tensor_names[1], int(d_output))

# Inference
cuda.memcpy_htod_async(d_input, image, stream) # Transfer data to device
context.execute_async_v3(stream.handle) # Execute model
cuda.memcpy_dtoh_async(np_output, d_output, stream) # Transfer predictions back to host

# Create Torch Tensor from NumPy
torch_output = torch.from_numpy(output).to("cuda:0")

# Postprocess ...
```

- Memory allocation
- Setting address
- Perform inference and Move back to CPU

Evolution and Strategy

(2) YoloWorld - Before optimization

```python
# Allocate Memory
d_input = cuda.mem_alloc(1 * 3 * 640 * 640 * 4)
d_output = cuda.mem_alloc(1 * 8400 * 13 * 4)
np_output = np.empty(1 * 8400 * 13, dtype=np.float32)

# Set Input and Output Addresses
context.set_tensor_address(tensor_names[0], int(d_input))
context.set_tensor_address(tensor_names[1], int(d_output))

# Inference
cuda.memcpy_htod_async(d_input, image, stream) # Transfer data to device
context.execute_async_v3(stream.handle) # Execute model
cuda.memcpy_dtoh_async(np_output, d_output, stream) # Transfer predictions back to host

# Create Torch Tensor from NumPy
torch_output = torch.from_numpy(output).to("cuda:0")

# Postprocess ...
```

- Memory allocation
- Setting address
- Perform inference and Move back to CPU
- Create Torch Tensor (GPU) and post-processing

Evolution and Strategy

(2) YoloWorld - After optimization

```
# Allocate Memory
d_input = torch.zeros([1 * 3 * 640 * 640], dtype=torch.float32, device="cuda:0")
d_output = torch.zeros([8400, 13], dtype=torch.float32, device="cuda:0")

# Set Input and Output Addresses
context.set_tensor_address(tensor_names[0], d_input.data_ptr())
context.set_tensor_address(tensor_names[1], d_output.data_ptr())

# Inference
d_input.copy_(torch.from_numpy(image)) # Transfer input data to device
context.execute_async_v3(stream_handle=torch.cuda.current_stream().cuda_stream) # Execute model

# Postprocess ...
```

- Memory allocation

Evolution and Strategy

(2) YoloWorld - After optimization

```python
# Allocate Memory
d_input = torch.zeros([1 * 3 * 640 * 640], dtype=torch.float32, device="cuda:0")
d_output = torch.zeros([8400, 13], dtype=torch.float32, device="cuda:0")

# Set Input and Output Addresses
context.set_tensor_address(tensor_names[0], d_input.data_ptr())
context.set_tensor_address(tensor_names[1], d_output.data_ptr())

# Inference
d_input.copy_(torch.from_numpy(image)) # Transfer input data to device
context.execute_async_v3(stream_handle=torch.cuda.current_stream().cuda_stream) # Execute model

# Postprocess ...
```

- Memory allocation
- Setting address

Evolution and Strategy

(2) YoloWorld - After optimization

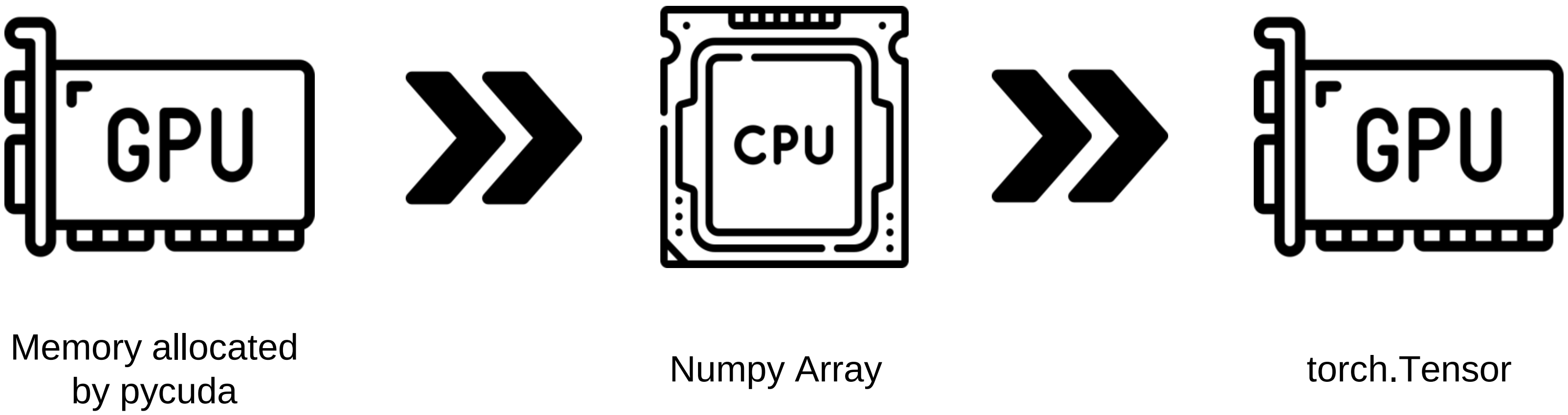
```python
# Allocate Memory
d_input = torch.zeros([1 * 3 * 640 * 640], dtype=torch.float32, device="cuda:0")
d_output = torch.zeros([8400, 13], dtype=torch.float32, device="cuda:0")

# Set Input and Output Addresses
context.set_tensor_address(tensor_names[0], d_input.data_ptr())
context.set_tensor_address(tensor_names[1], d_output.data_ptr())

# Inference
d_input.copy_(torch.from_numpy(image)) # Transfer input data to device
context.execute_async_v3(stream_handle=torch.cuda.current_stream().cuda_stream) # Execute model

# Postprocess ...
```

- Memory allocation
- Setting address
- Perform inference and post-processing

Evolution and Strategy

(2) YoloWorld

Original



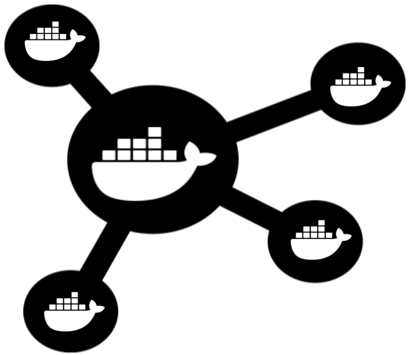
> Solve with only use PyTorch, and use tensor.data_ptr() to get the pointer.

2.9x → 3.1x

What problems have you encountered?

Hard to profile ROS with Nsight System

- Profiling with NVTX tag



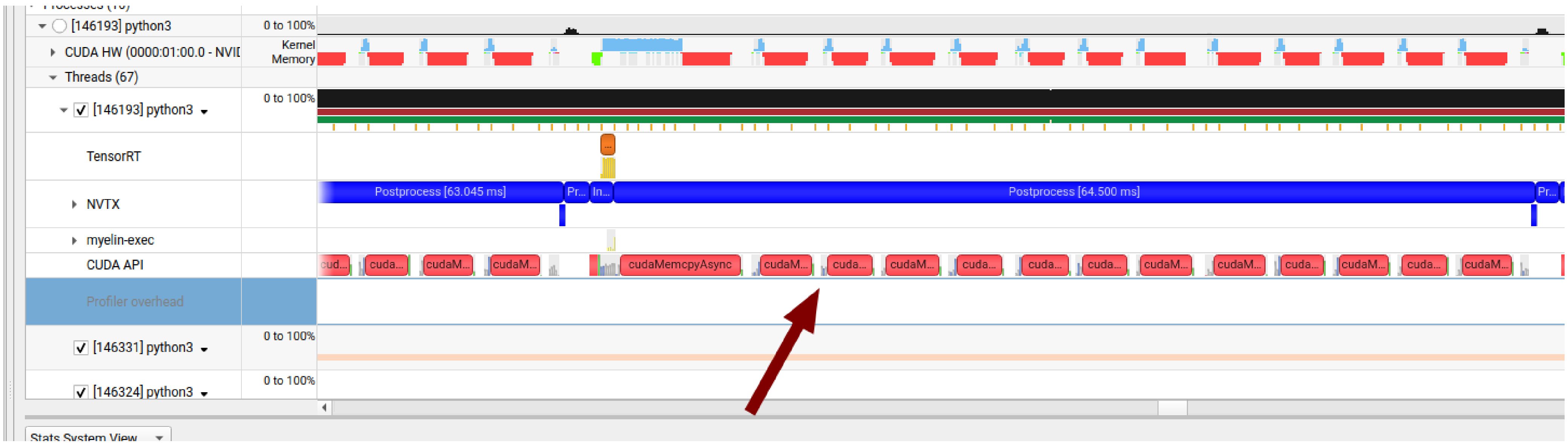
Profiling across multi-container

- Profiling each process with each container

Future Work

YoloWorld

Image post-processing was time-consuming because it **moved data between the GPU and the CPU periodically.**

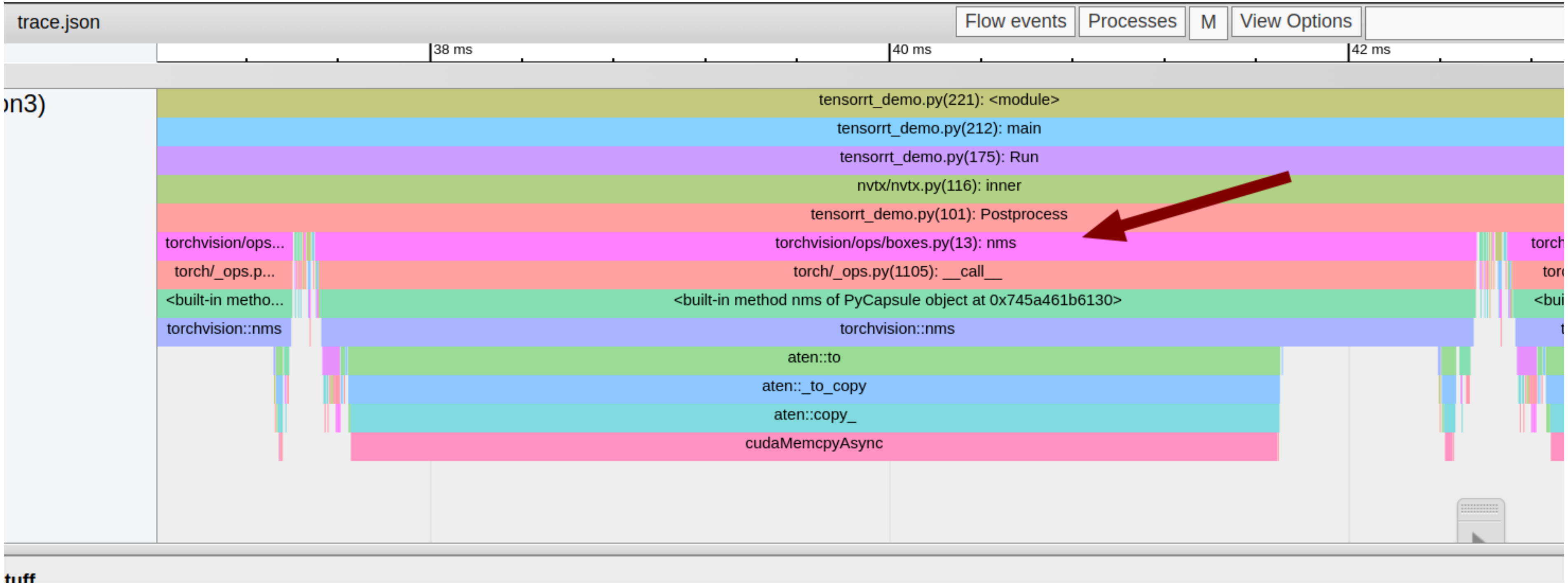


Need to remove the bubble between these commands.

Future Work

YoloWorld

By profiling result, the **cudaMemcpyAsync** was executed from Torchvision's NMS.



Thank You

OpenACC

More Science, Less Programming