CUDA streams and processing frameworks

Piotr Jarosik, Marcin Lewandowski, Billy Yiu

GPU Short Course, IEEE UFFC-JS 2024

Agenda

- CUDA streams and events
- Graph of operations
- GPU Frameworks

CUDA Stream: introduction

We can run a sequence of GPU kernels to implement the complete e.g. B-mode imaging:

```
# HtoD
rf_gpu.set(rf_cpu)
iq = ddc_kernel(rf_gpu)
hri = delay_and_sum_kernel(iq)
envelope = envelope_kernel(hri)
bmode = log_compression_kernel(envelope)
# DtoH
bmode_cpu = bmode.get()
```

CUDA Streams: introduction

Refresher: asynchronous vs synchronous operations:

- kernel invocation
  - **asynchronous**
  - we have no guarantee that the kernel execution will be finished (or even started) by the next line of our script/host application code
- data transfers (DtoH, HtoD, DtoD): can be **synchronous** or **asynchronous** (it's up to the programmer)

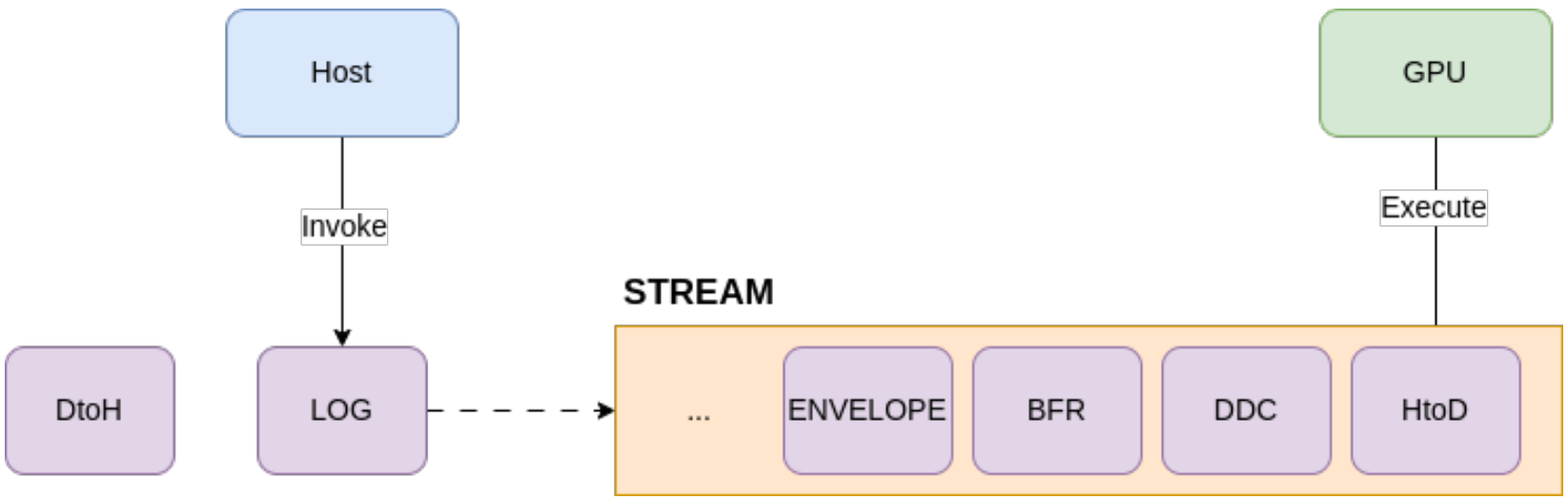
CUDA Streams: example

Refresher:

- host code puts GPU operations into **CUDA Streams**
- **CUDA Streams** $\rightarrow$ a *queue* (FIFO) of operations
- To wait for all the GPU kernels in the stream to finish: call 'synchronize' on the stream.

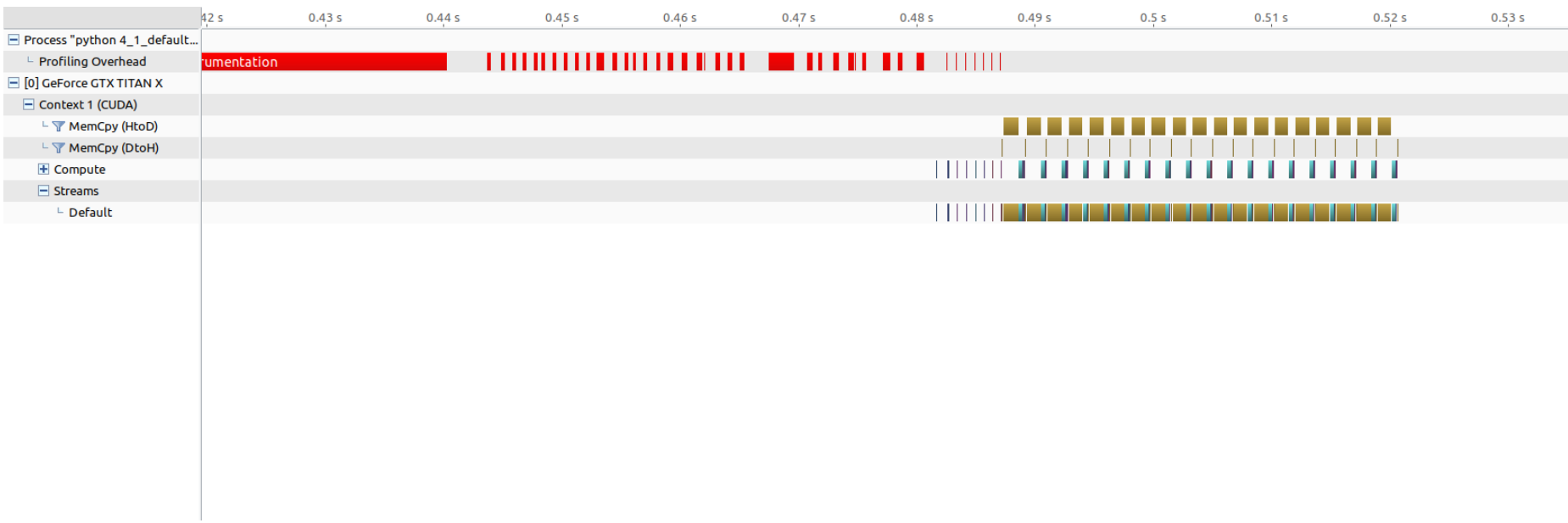
CUDA Streams: example

```
rf_gpu.set(rf_cpu)    # HtoD
iq = ddc_kernel(rf_gpu) # DDC
hri = delay_and_sum_kernel(iq)  # BFR
envelope = envelope_kernel(hri)  # ENVELOPE
bmode = log_compression_kernel(envelope) # LOG
bmode_cpu = bmode.get() # DtoH
```



CUDA Streams: example

```
rf_gpu.set(rf_cpu)    # HtoD
iq = ddc_kernel(rf_gpu) # DDC
hri = delay_and_sum_kernel(iq)  # BFR
envelope = envelope_kernel(hri)  # ENVELOPE
bmode = log_compression_kernel(envelope) # LOG
bmode_cpu = bmode.get() # DtoH
```



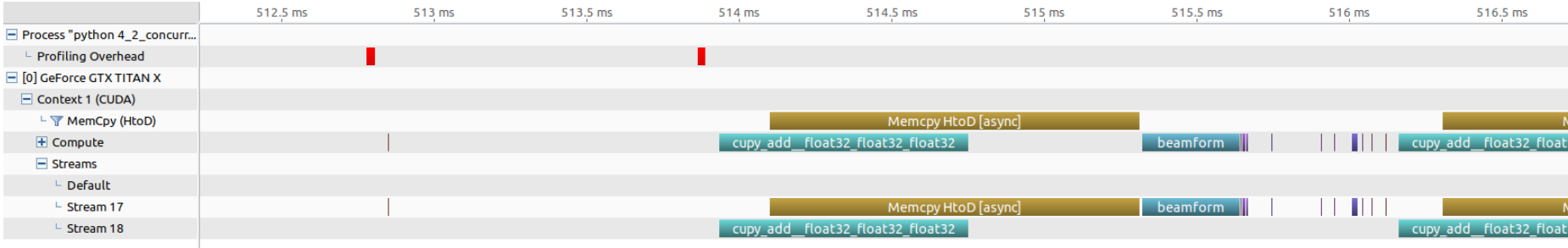
CUDA Streams: multiple streams

Is it possible to run GPU operations in parallel?

- Generally, it's possible, by using multiple streams.
- In practice: it depends what operations we would like to run in parallel:
  - executing GPU kernels simultaneously – not always guaranteed (it depends on the GPU occupancy, kernel scheduler implementation, etc.)
  - GPU kernels and data transfers can be overlapped – data transfers can be handled by a separate GPU component **DMA**, that can run in parallel with SMs

CUDA Streams: overlapping data transfer with kernels

```
stream_17 = cp.cuda.Stream(non_blocking=True)
stream_18 = cp.cuda.Stream(non_blocking=True)
with stream_17:
   gpu_array.set(rf)
   img       = beamformer.process(gpu_array)
   envelope  = to_envelope.process(img)
   bmode     = to_bmode.process(envelope)
with stream_18:
   a+b
```



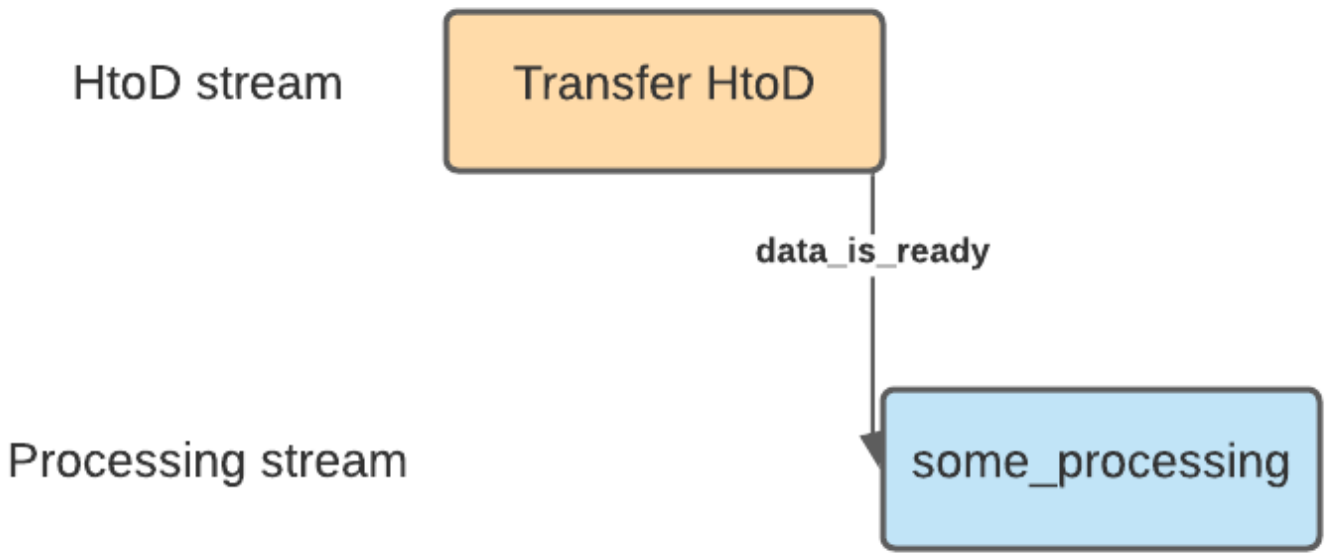
CUDA Streams synchronization: Events

How to make sure, that GPU kernel will not start until the HtoD transfer is finished?

- We can use CUDA **Events**.
- **CUDA event** object represents an event in the CUDA stream execution timeline. An example of such an event is the completion of a GPU kernel, or that the data transfer from host to GPU has finished.

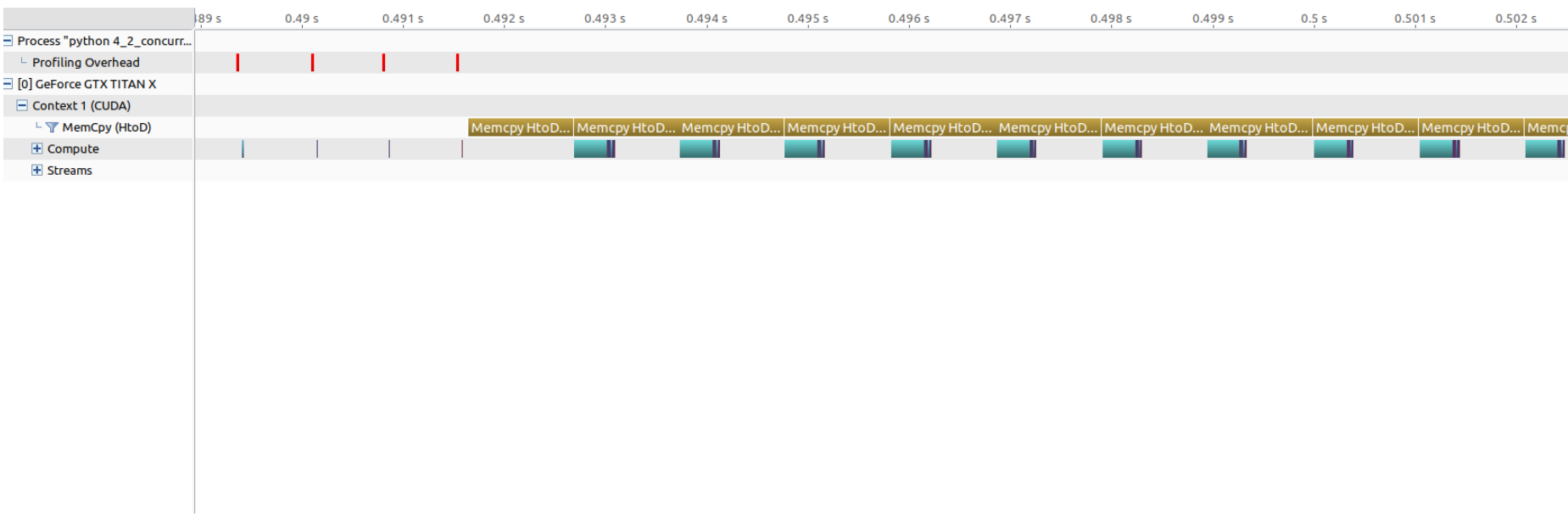
CUDA Streams synchronization: Events

```
data_is_ready = cp.cuda.Event()
gpu_array.set(rf, stream=HtoD_Stream)
data_is_ready.record(stream=HtoD_Stream)
processing_stream.wait_event(data_is_ready)
with processing_stream:
  some_processing(gpu_array)
```



CUDA Streams synchronization: Events

```
data_is_ready = cp.cuda.Event()
gpu_array.set(rf, stream=HtoD_Stream)
data_is_ready.record(stream=HtoD_Stream)
processing_stream.wait_event(data_is_ready)
with processing_stream:
  some_processing(gpu_array)
```

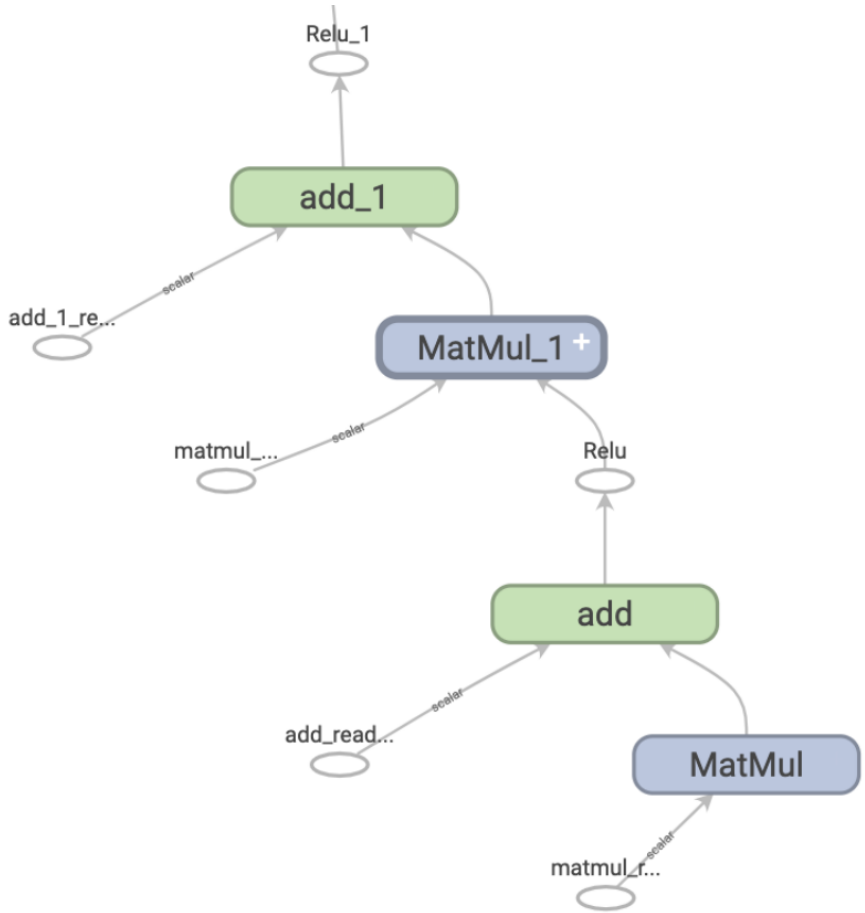


Graph o operations

It's possible to use CUDA kernels and events to implement a graph of operations.
Graph of operations: a Graph $G(V, E)$, where:

- V: **operations**
- E: **dependencies** between operations (input/output arrays)

Graph o operations: example

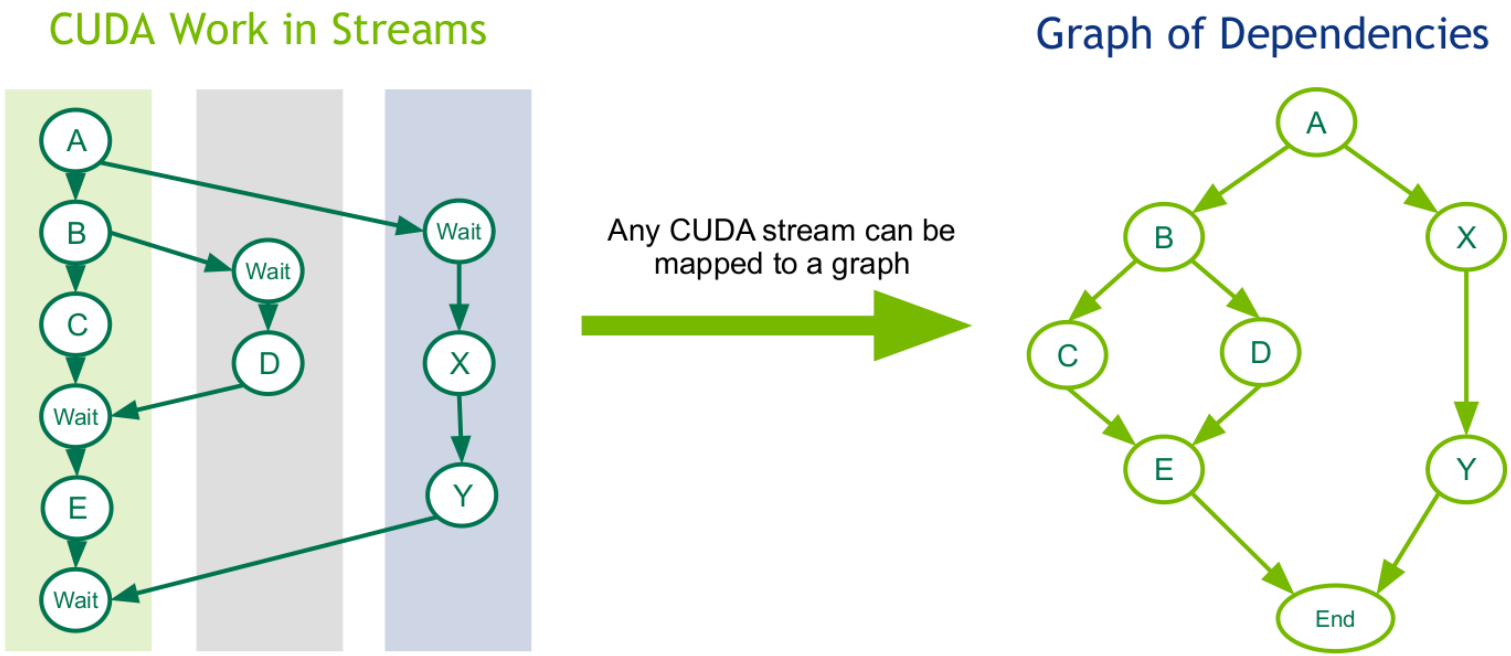


Source: "Tensorflow: introduction to graphs, `https://www.tensorflow.org/guide/intro_to_graphs`"

Graph of operations: frameworks

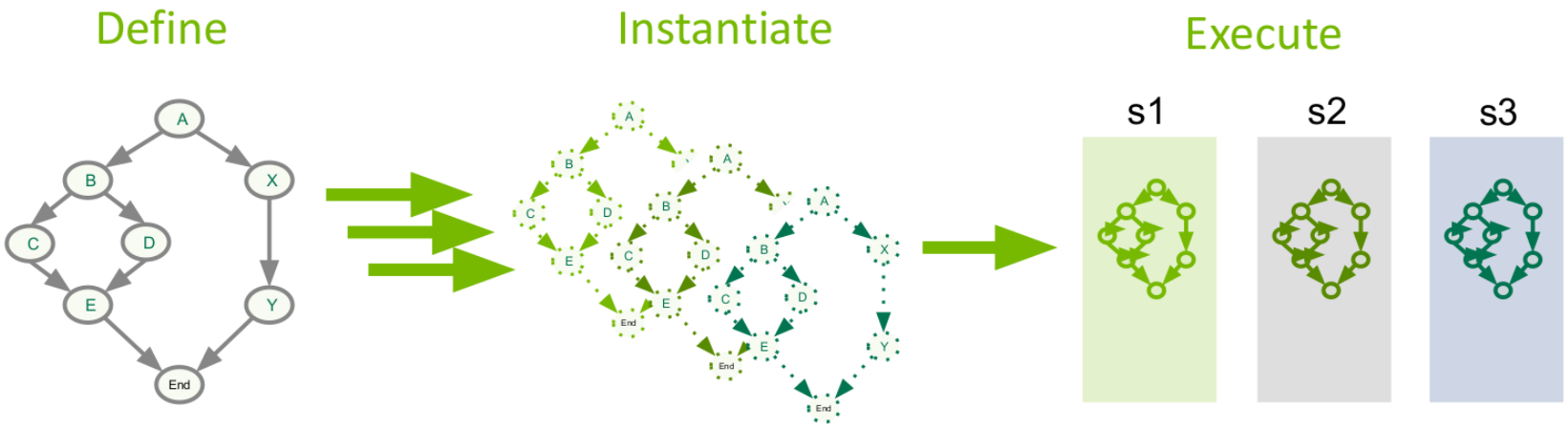
- tensorflow (based on XLA)
- ...
- NVIDIA CUDA Graphs
- NVIDIA CUDA Holoscan SDK

CUDA Graphs

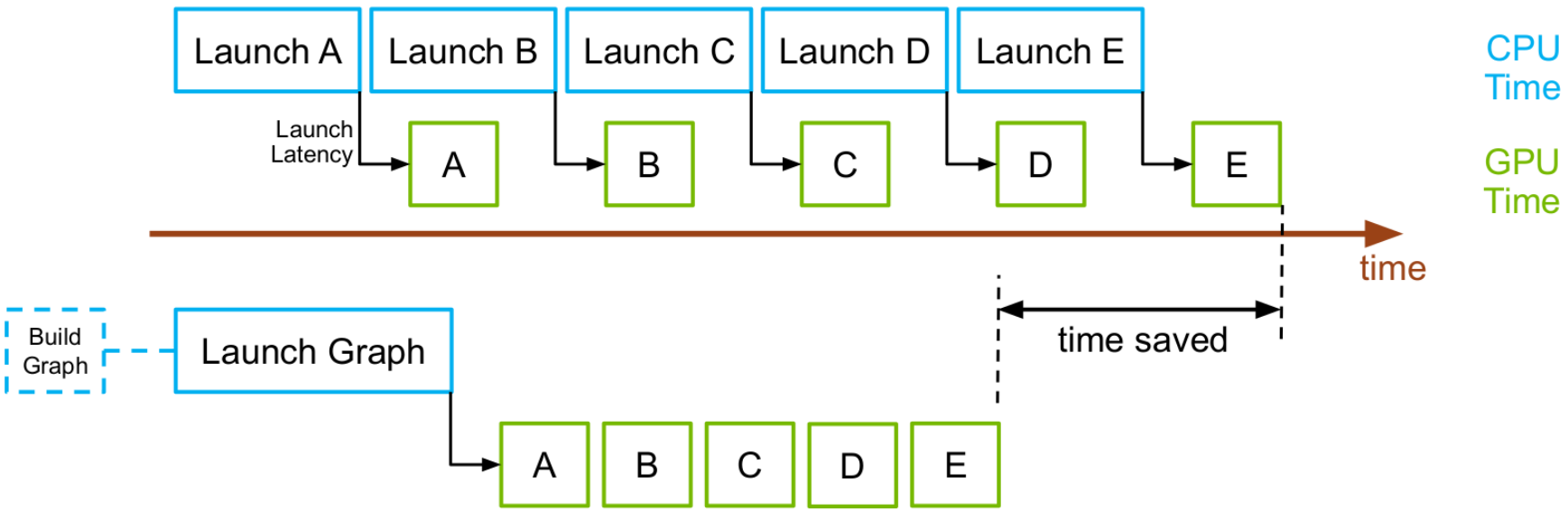


Source: "013 NVIDIA CUDA Graphs", `https://www.olcf.ornl.gov/wp-content/uploads/2021/10/013_CUDA_Graphs.pdf`

CUDA Graphs: Execution model



CUDA Graphs: kernel invocation overhead reduction



Source: "013 NVIDIA CUDA Graphs", `https://www.olcf.ornl.gov/wp-content/uploads/2021/10/013_CUDA_Graphs.pdf`

NVIDIA Holoscan SDK

NVIDIA Holoscan is the AI sensor processing platform that combines:

- hardware systems for low-latency sensor and
- network connectivity,
- optimized libraries for data processing and AI,
- ...

NVIDIA Holoscan SDK is a development kit that provides:

- C++ and Python API,
- Built-in, common operators (AI inference, etc.),
- examples, a remote reopsitory of operators, performance tools...

Source: "013 NVIDIA CUDA Graphs", `https://www.olcf.ornl.gov/wp-content/uploads/2021/10/013_CUDA_Graphs.pdf`

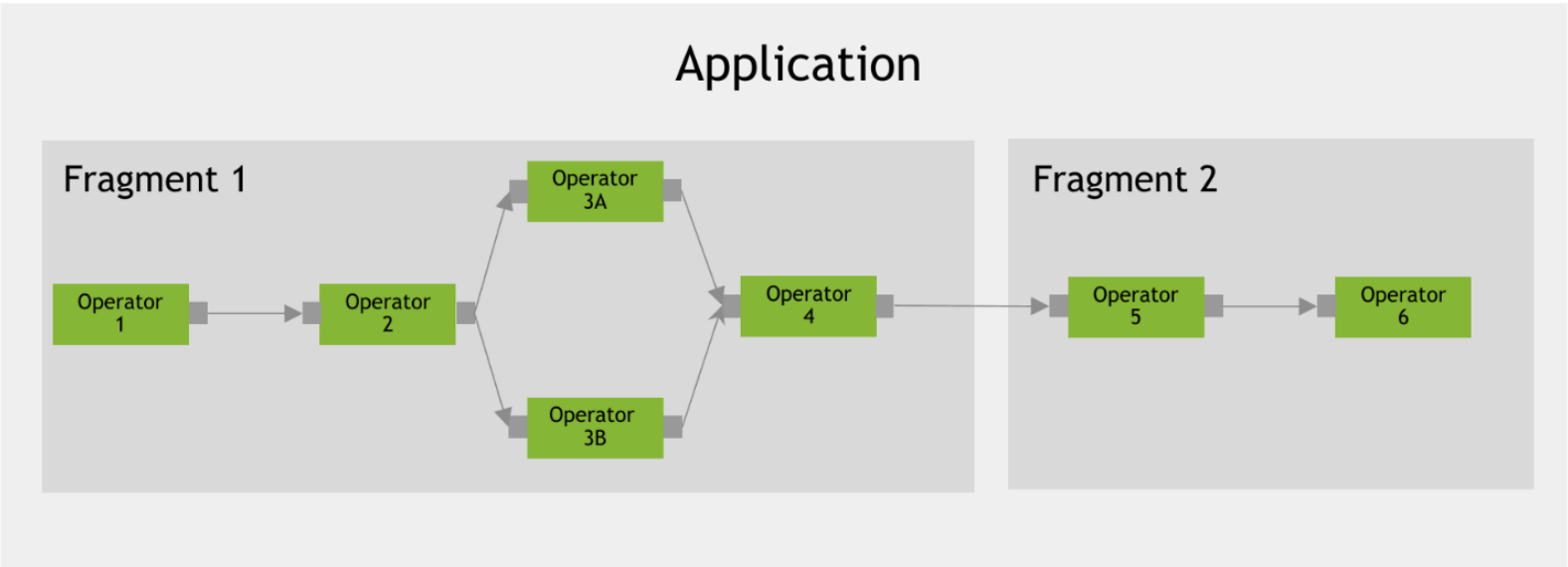
NVIDIA Holoscan SDK: Core concepts

Hoscan SDK implements a custom graph framework: **Graph Execution Framework (GXF)**.
Also provides a native support for:

- GPU Direct RDMA
- TensorRT
- ...

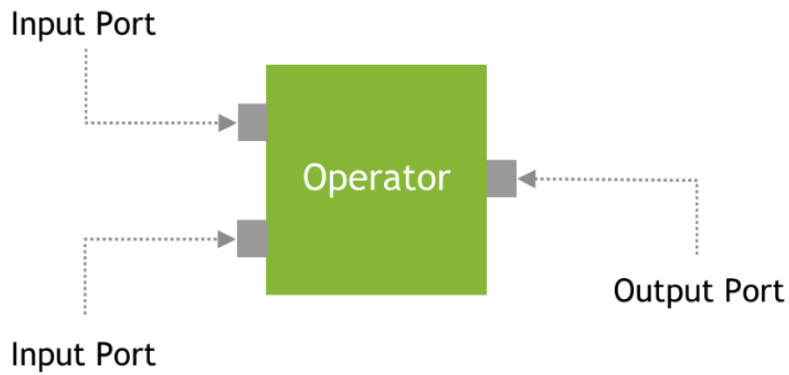
NVIDIA Holoscan SDK: Core concepts

An **Application** is composed of **Fragments**, each of which runs a sub-graph of **Operations**.



NVIDIA Holoscan SDK: Core concepts

Operator is a basic unit of work. An operator receives streaming data at an input port, processes it, and publishes it to one of the output ports.



NVIDIA Holoscan SDK

HoloHub: a central repository to share reusable operators and applications with the Holoscan community.
In preparation:

- us4us us4R $\longleftrightarrow$ NVIDIA Holoscan SDK integration
- GPU RCA image reconstruction using NVIDIA Holoscan SDK (offline processing)

Please make sure to subscribe to the ‘github.com/lab4us/gpu-short-course‘ repository by pressing ’Star’ button. This way you will get notification when the above are ready. Thanks!