Lecture 10: Building production ready CUDA libraries

A computer architecture aware programming perspective

Oscar Amoros Huguet

About me

BSC

- 3D convolutions in OpenCL, CUDA and OmpSs

UB

- Teaching assistant 9 years teaching OpenCL/CUDA and other CS topics
- 2 papers (OpenCL)
- First OpenSource library: SimpleOpenCL

StreamHPC

- Porting Boost Compute (OpenCL) to HSA and HAS-IL

AutomaticTV 2016 – Today

- AutomaticTV: soft real time, C++ and CUDA
- New OpenSource libraries cvGPUSpeedup and the Fused Kernel library -> paper on the way!

Lecture overview

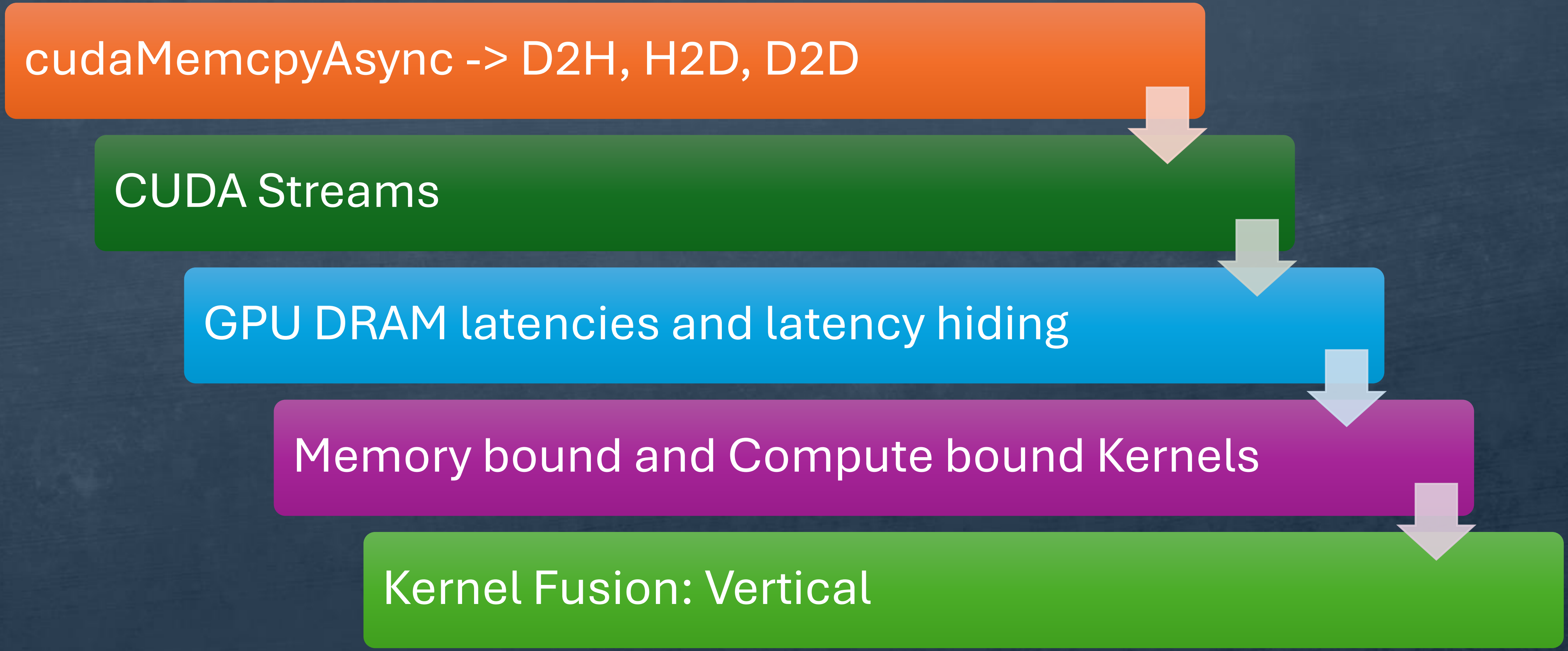
Summary of already seen concepts

Main story: why would a “CUDA ninja” want to create libraries?

Use cases:

- GPU communication utility
- GPU kernel libraries

Summary of already seen concepts



cudaMemcpyAsync -> D2H, H2D, D2D

CUDA Streams

GPU DRAM latencies and latency hiding

Memory bound and Compute bound Kernels

Kernel Fusion: Vertical

Lecture overview

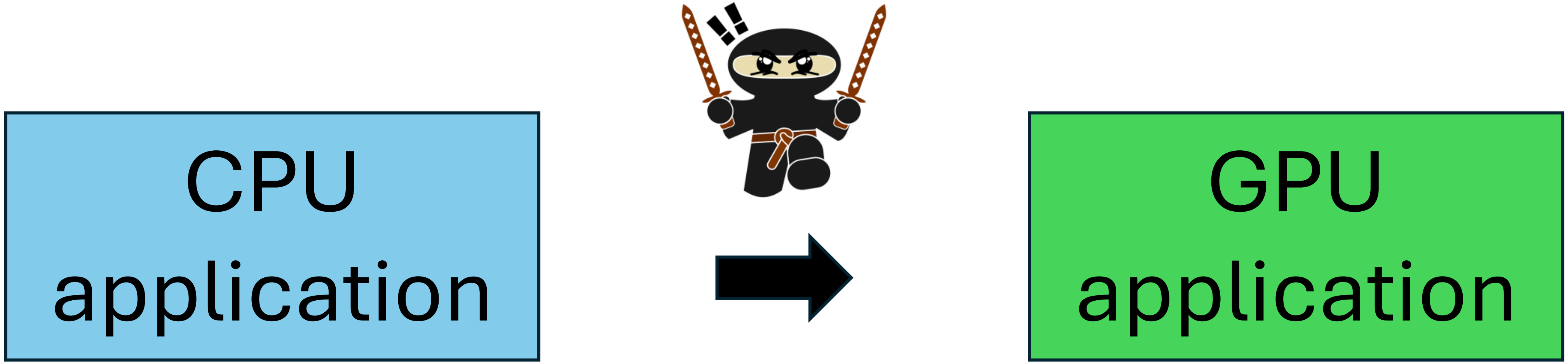
Summary of already seen concepts

Main story: libraries ★

Use cases:

- GPU communication utility
- GPU kernel libraries

Main story: 2008 approach



CPU application

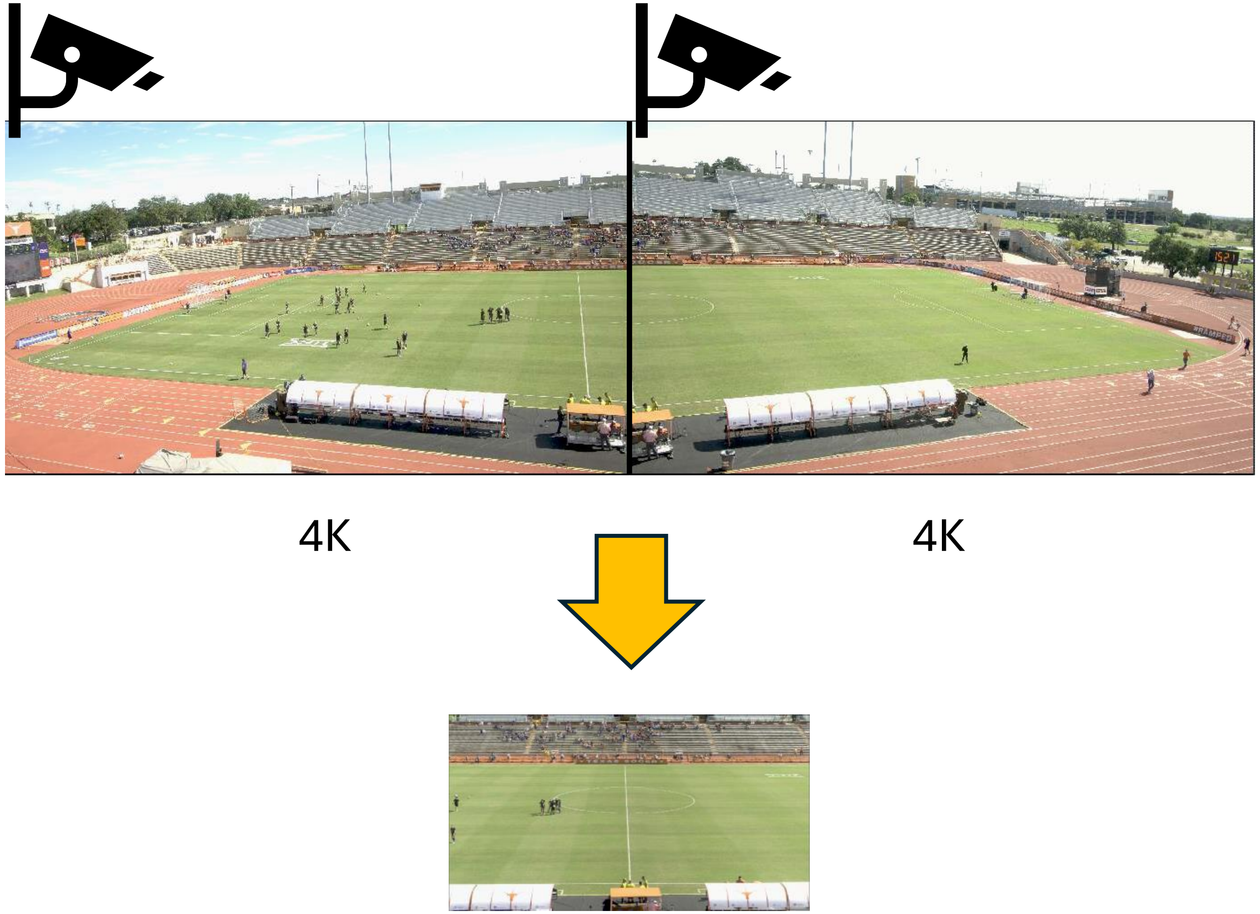
GPU application

Main story: most of my **C++/**CUDA exp.

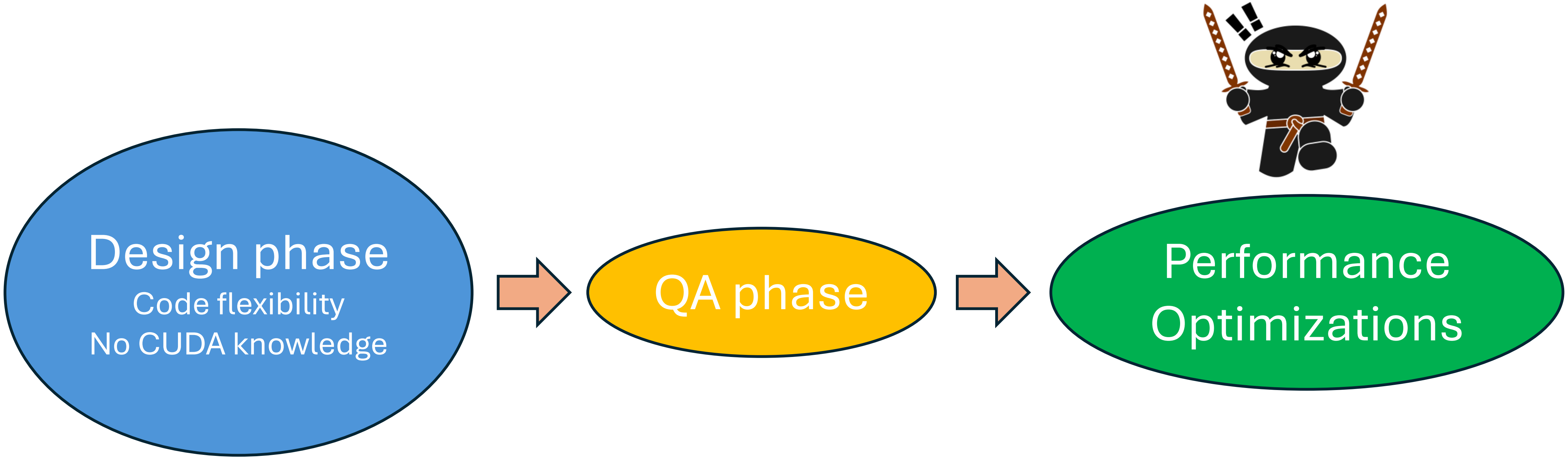
GRUP MEDIAPRO

AUTOMATICTV

https://automatic.tv/



Main story: ideal project organization



Design phase

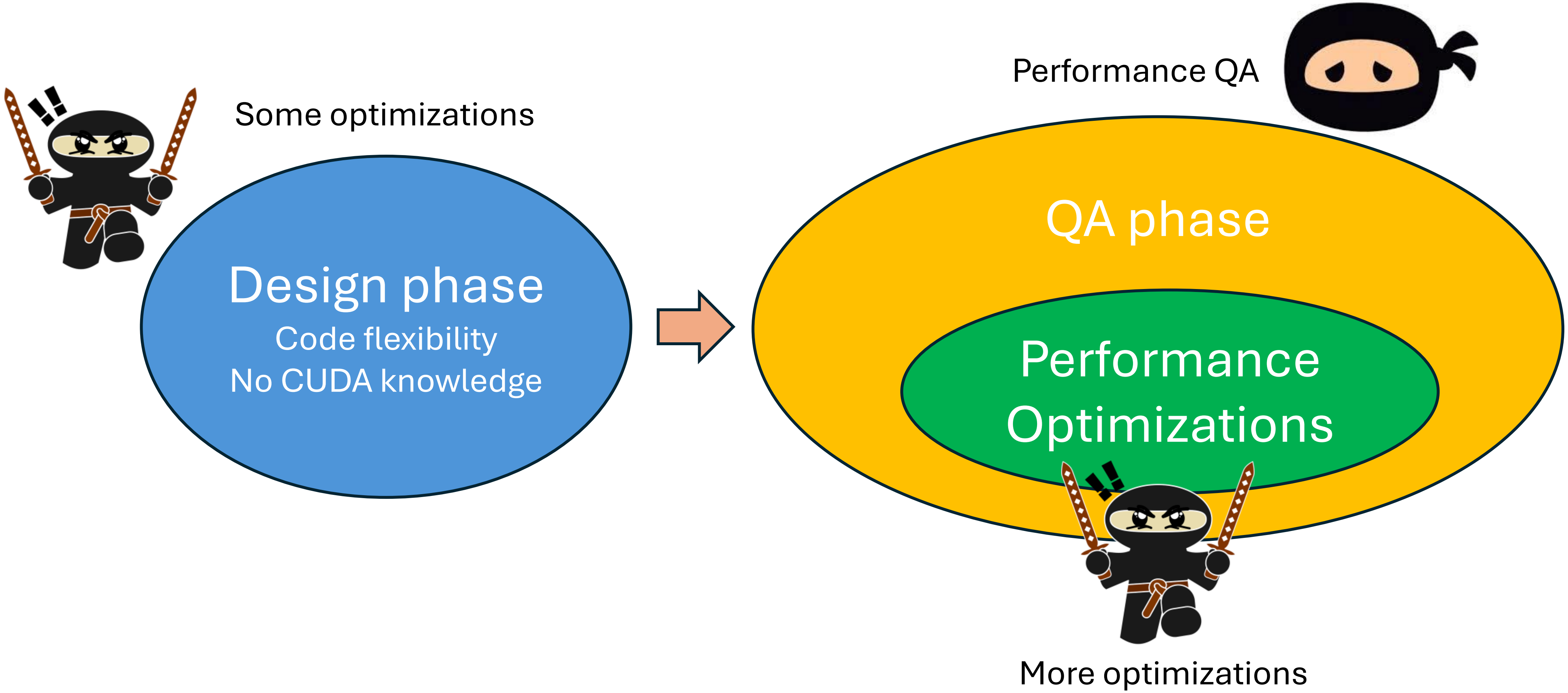
Code flexibility

No CUDA knowledge

QA phase

Performance Optimizations

Main story: reality



Some optimizations

Design phase

Code flexibility

No CUDA knowledge

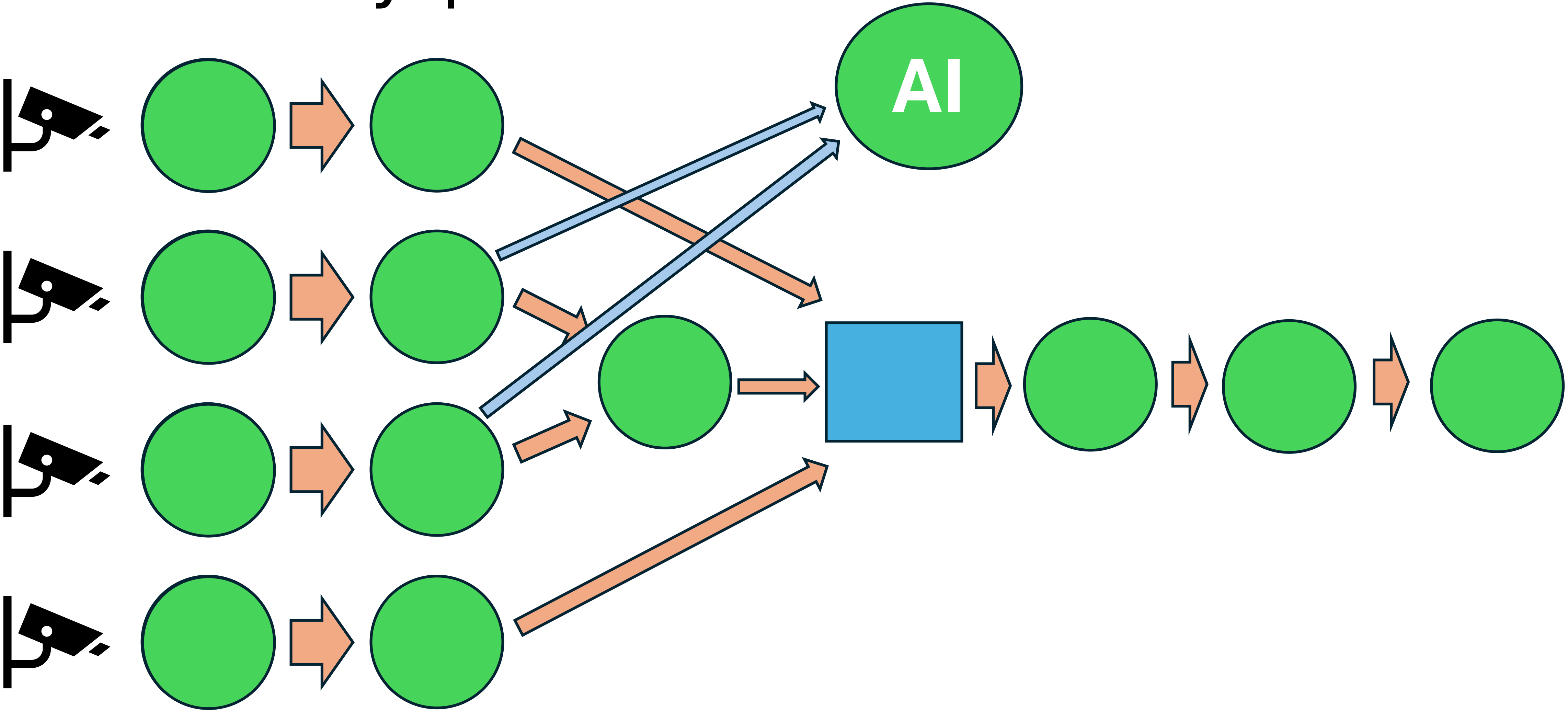
Performance QA

QA phase

Performance Optimizations

More optimizations

Main story: problem



Main story: tough balance

Abstraction

Performance

Main story: user requirements

Abstraction

Performance

Main story: what users care about

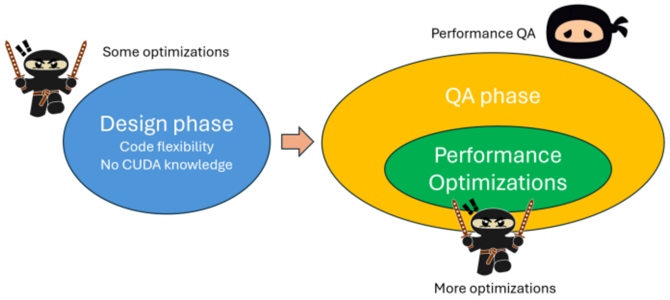
Learning curve

API

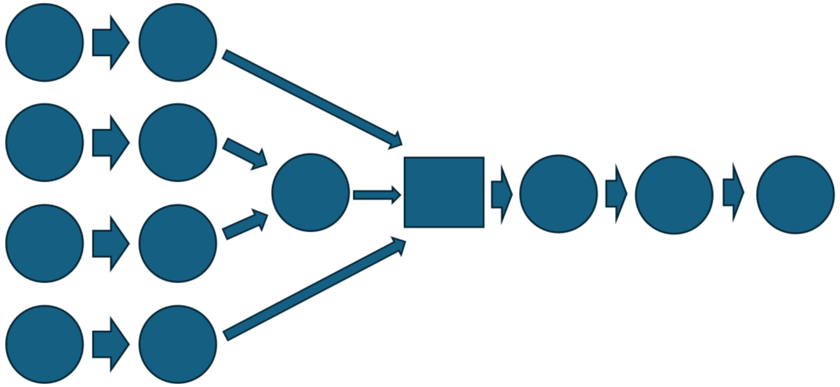
Main story: libraries challenges

Main story: summary

Motivation



Problem to solve



Challenges

Lecture overview

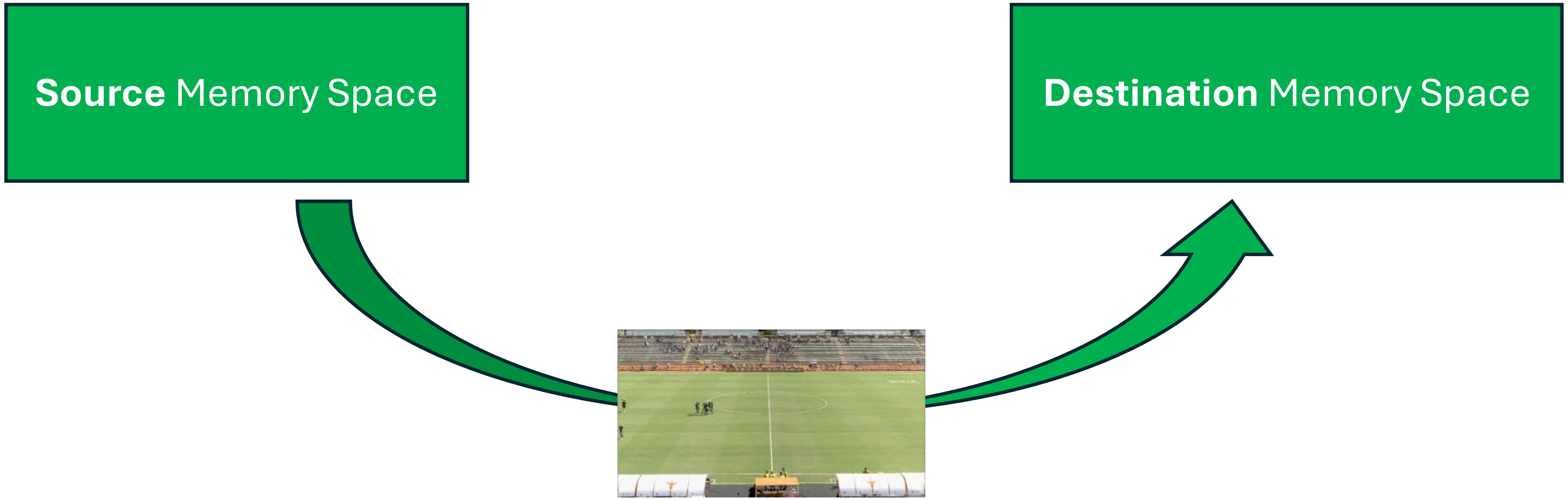
Summary of already seen concepts

Main story: libraries

Use cases:

- GPU communication utility
- GPU kernel libraries

GPU communication utility



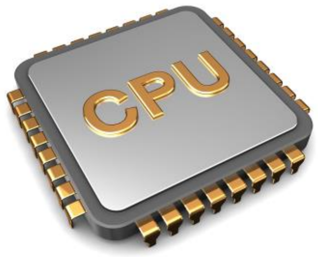
Source Memory Space

Destination Memory Space

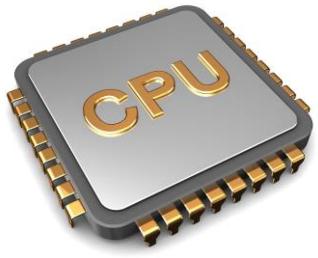
GPU communication utility

Memory Space Types

GPU



CPU
Pinned

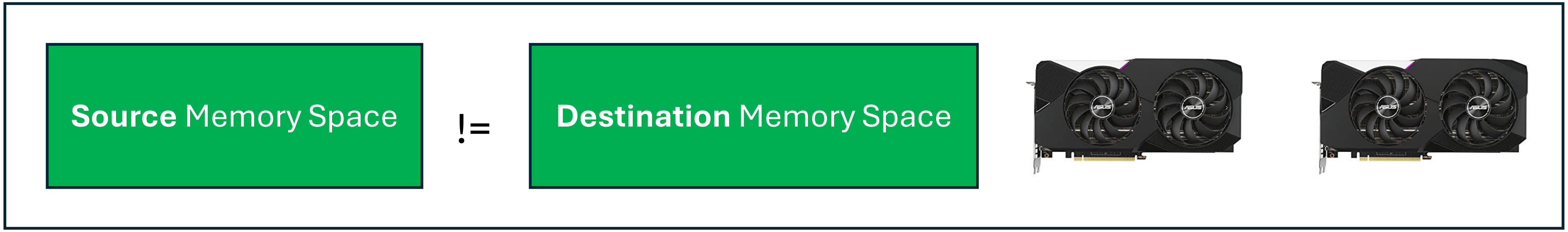


CPU

GPU communication utility

Pre-allocate all your memory

GPU communication utility

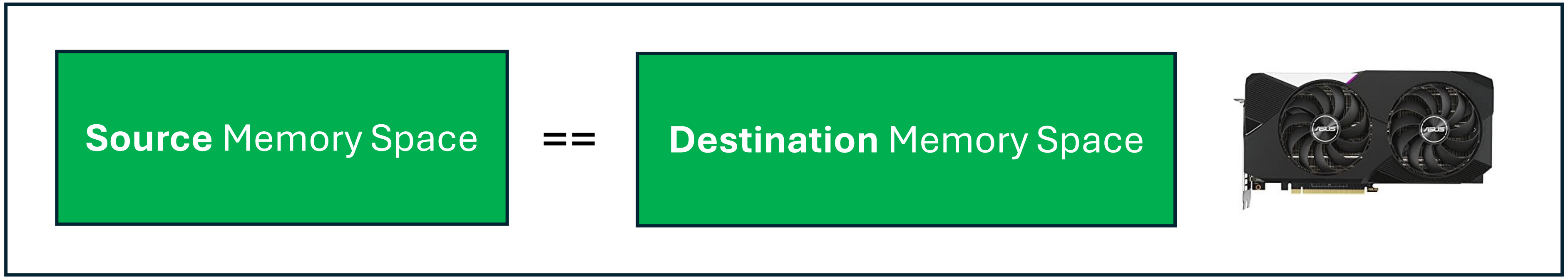


Source Memory Space

!=

Destination Memory Space

```
Manager.manage(sourcePtr, destinationPtr);
```



Source Memory Space

==

Destination Memory Space

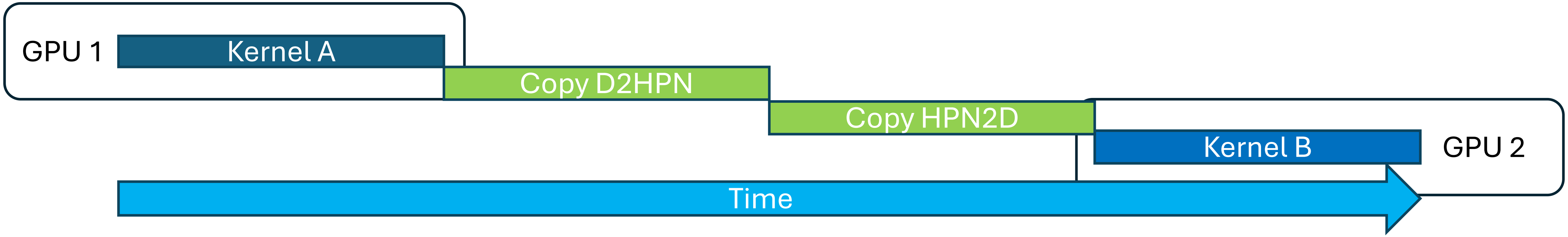
GPU communication utility

Source MS Type	Destination MS Type	Number of copies	
CPU	CPU	0	
CPU	CPU Pinned	1	
CPU	GPU	2 CPU -> Pinned -> GPU	
CPU Pinned	CPU	1	
CPU Pinned	CPU Pinned	0	
CPU Pinned	GPU	1	
GPU	CPU	2 GPU -> Pinned -> CPU	
GPU	CPU Pinned	1	
GPU	GPU	0	
GPU	Peer GPU	1	cudaMemcpyPeerAsync
GPU	Fake Peer GPU	2 GPU -> Pinned -> GPU	cudaMemcpyPeerAsync

Multi GPU

GPU communication utility

Problem: serial execution (GPU to Fake Peer GPU)



GPU 1

Kernel A

Copy D2HPN

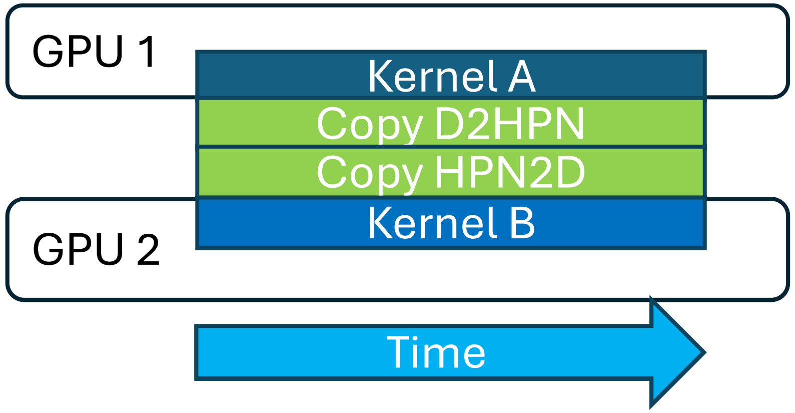
Copy HPN2D

Kernel B

GPU 2

Time

Goal: parallel execution



GPU 1

Kernel A

Copy D2HPN

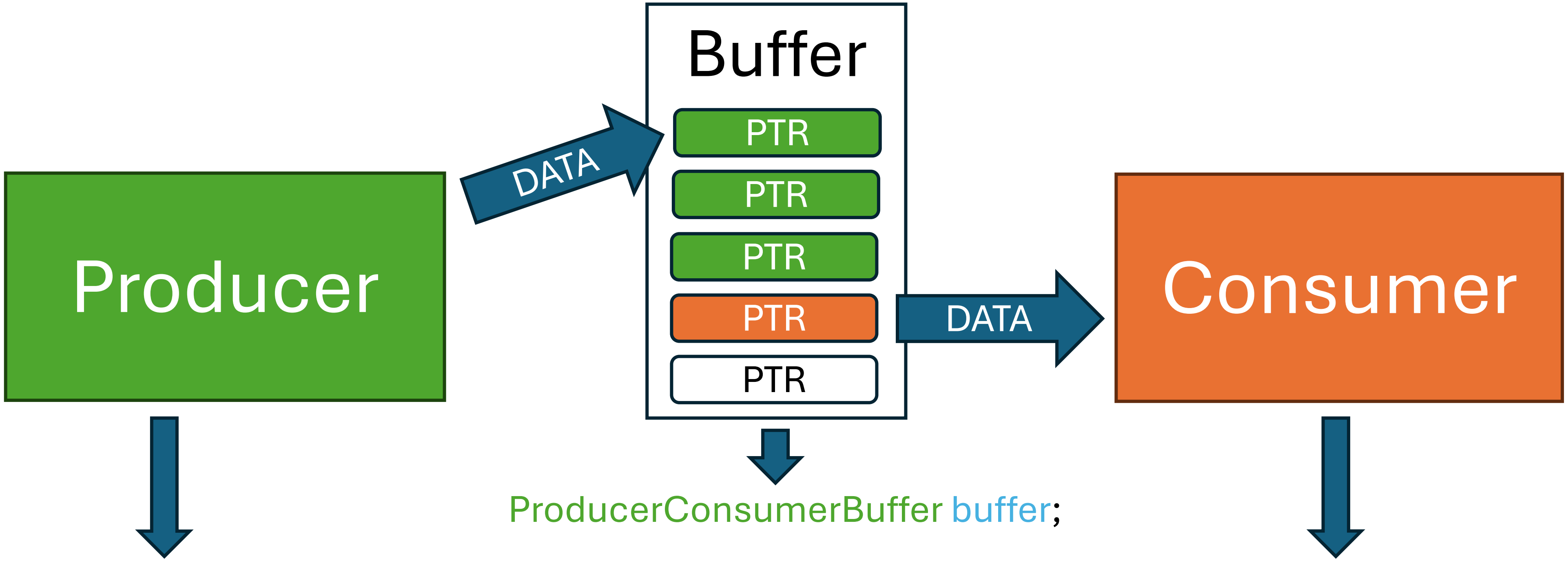
Copy HPN2D

Kernel B

GPU 2

Time

GPU communication utility



Producer

DATA

Buffer

PTR

PTR

PTR

PTR

PTR

DATA

Consumer

```
ProducerConsumerBuffer buffer;
```

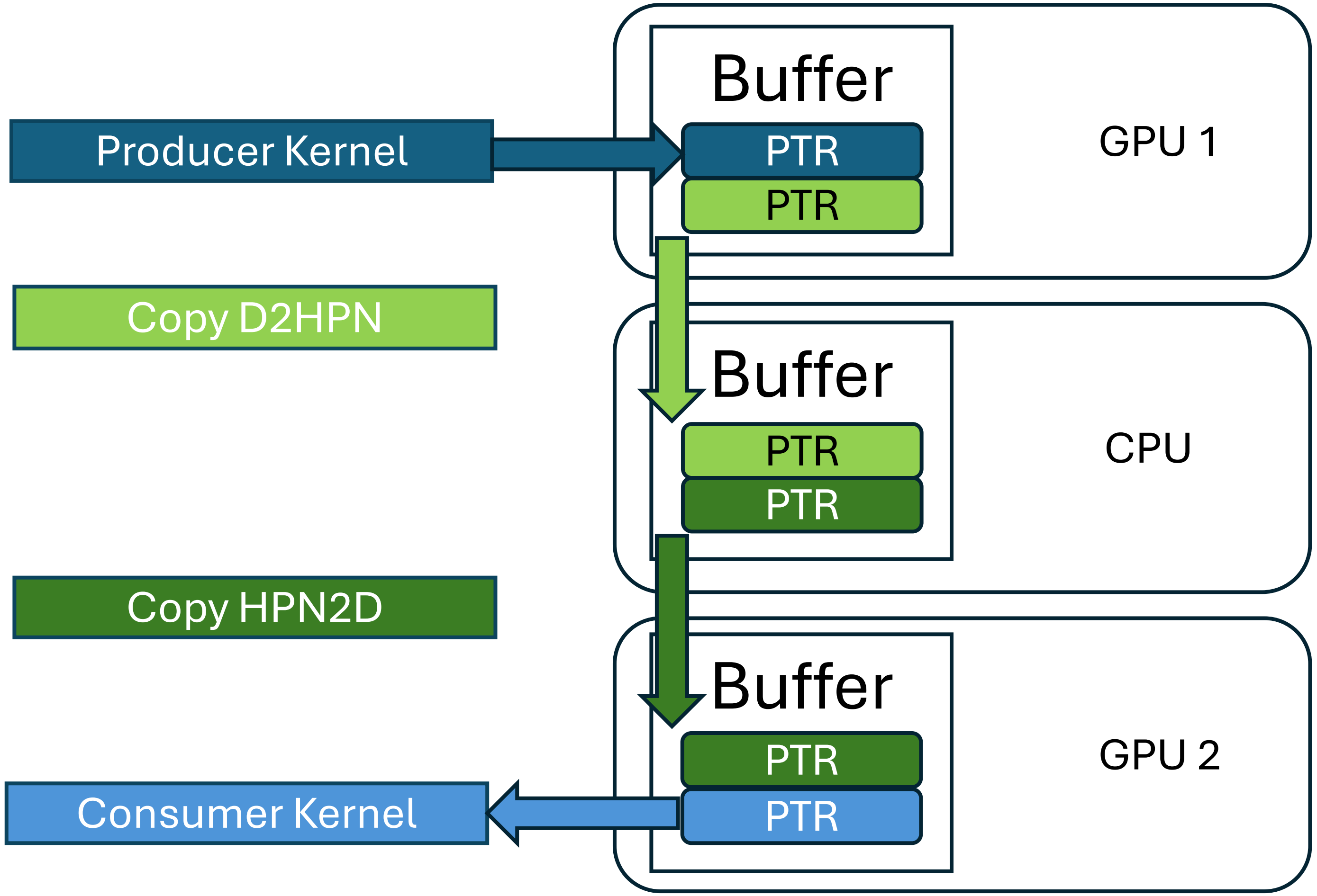
CPU THREAD 1

```
ptrToProduce = buffer.getPointerToProduce();
doProduce(ptrToProduce); // Modifying the data
buffer.setPointerAsProduced(producedData);
```

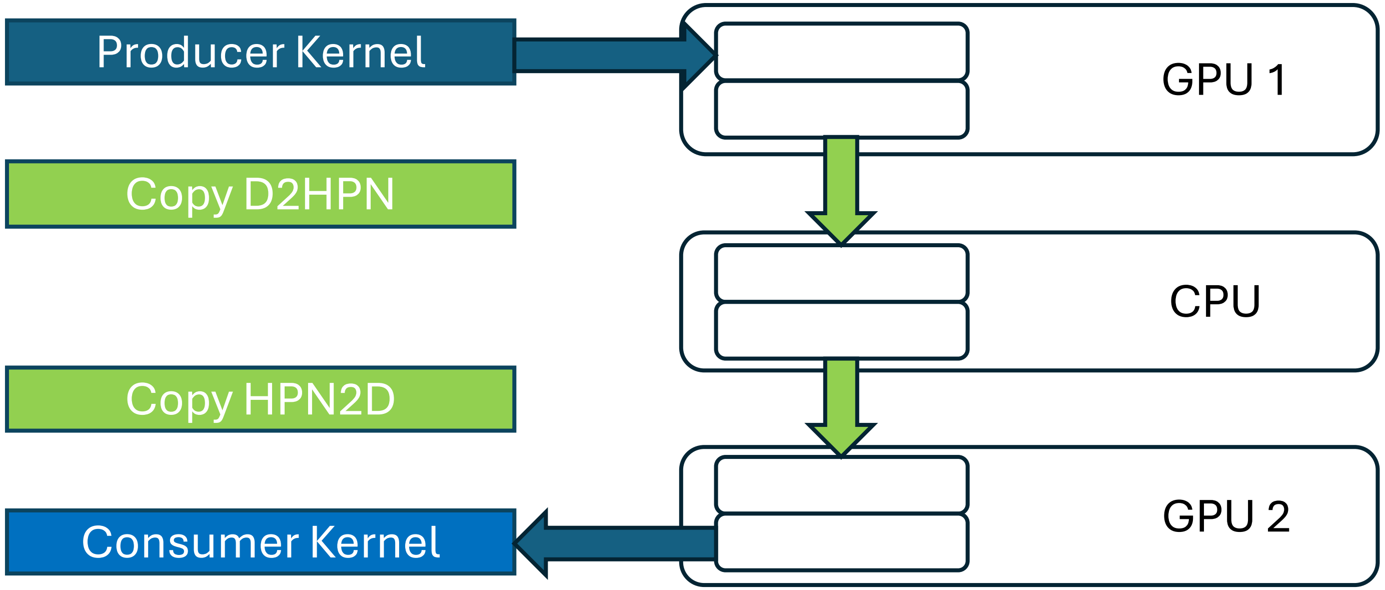
CPU THREAD 2

```
ptrToConsume = buffer.getPointerToConsume();
doConsume(ptrToConsume); // Reading the data
buffer.setPointerAsConsumed(producedData);
```

GPU communication utility



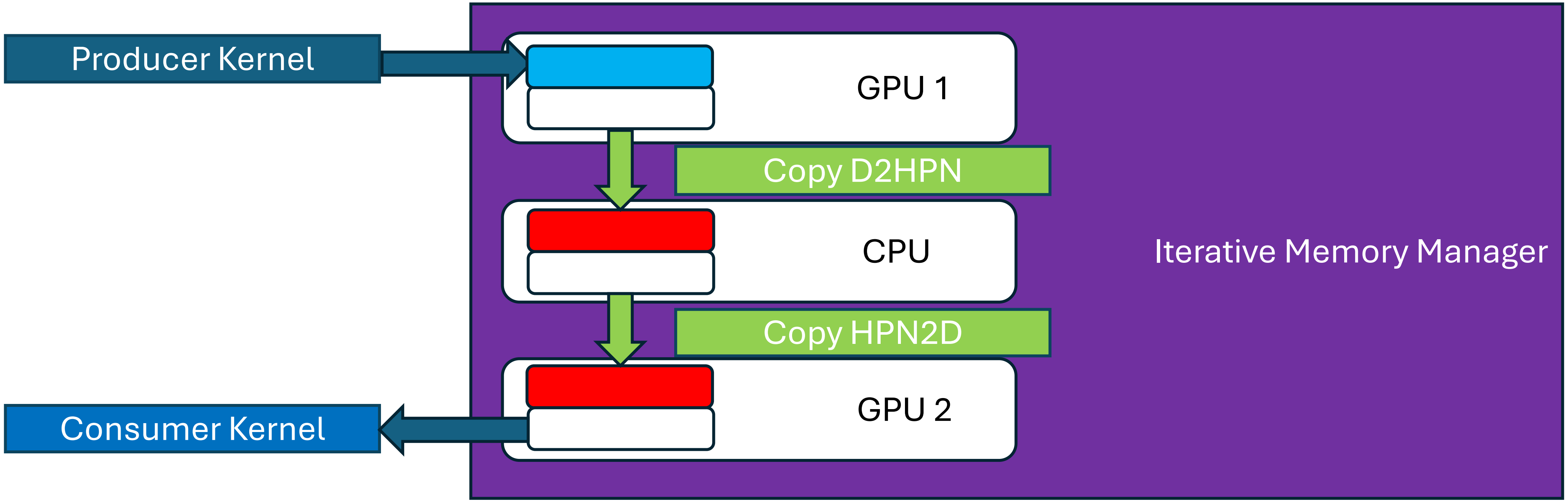
GPU communication utility



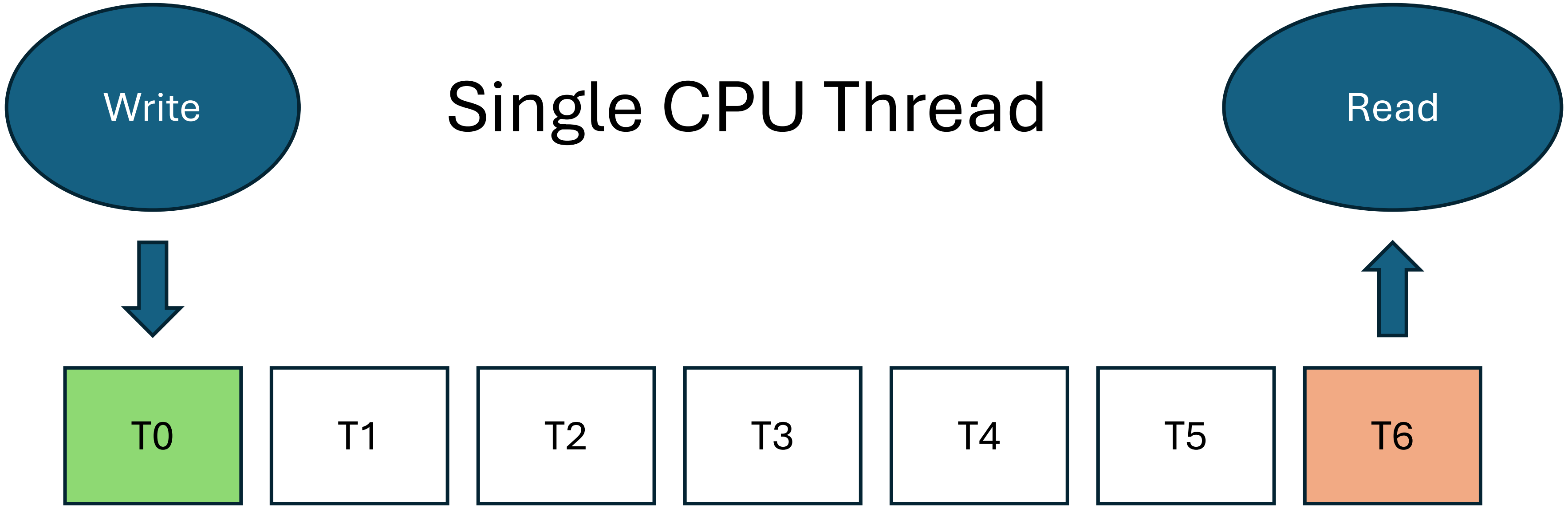
Iteration: [illegible]

END

GPU communication utility



GPU communication utility

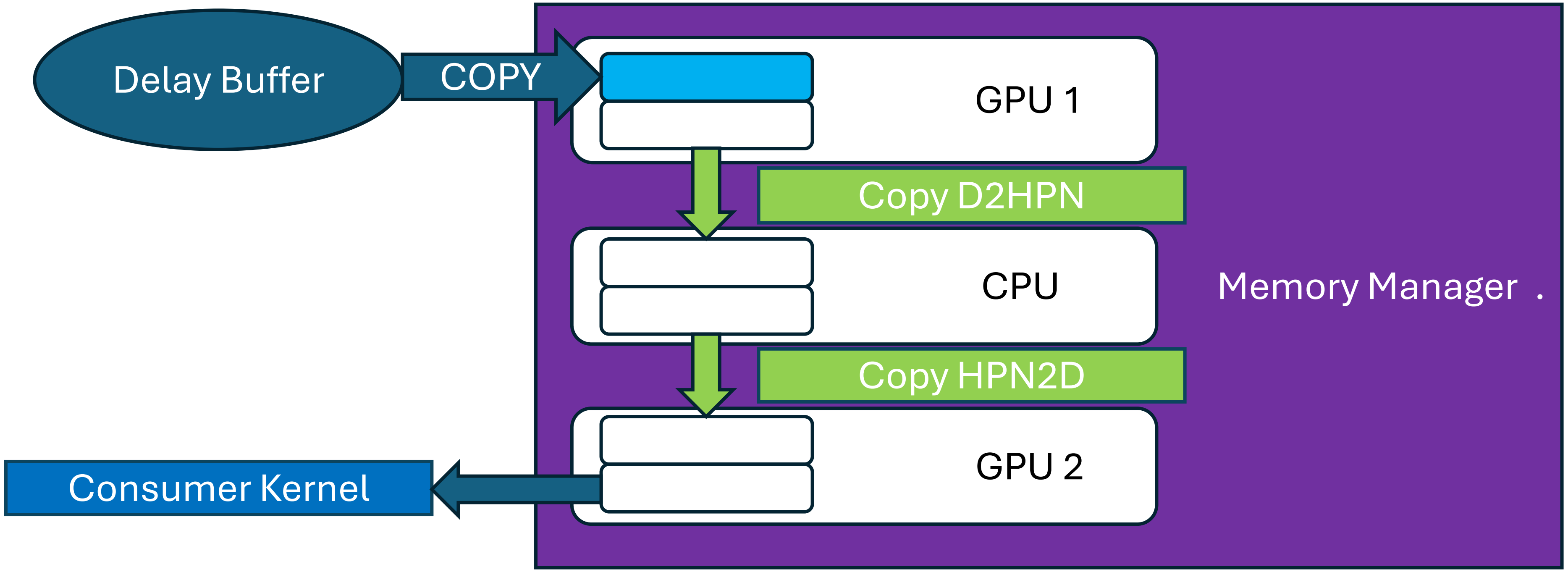


Single CPU Thread

Delay Buffer

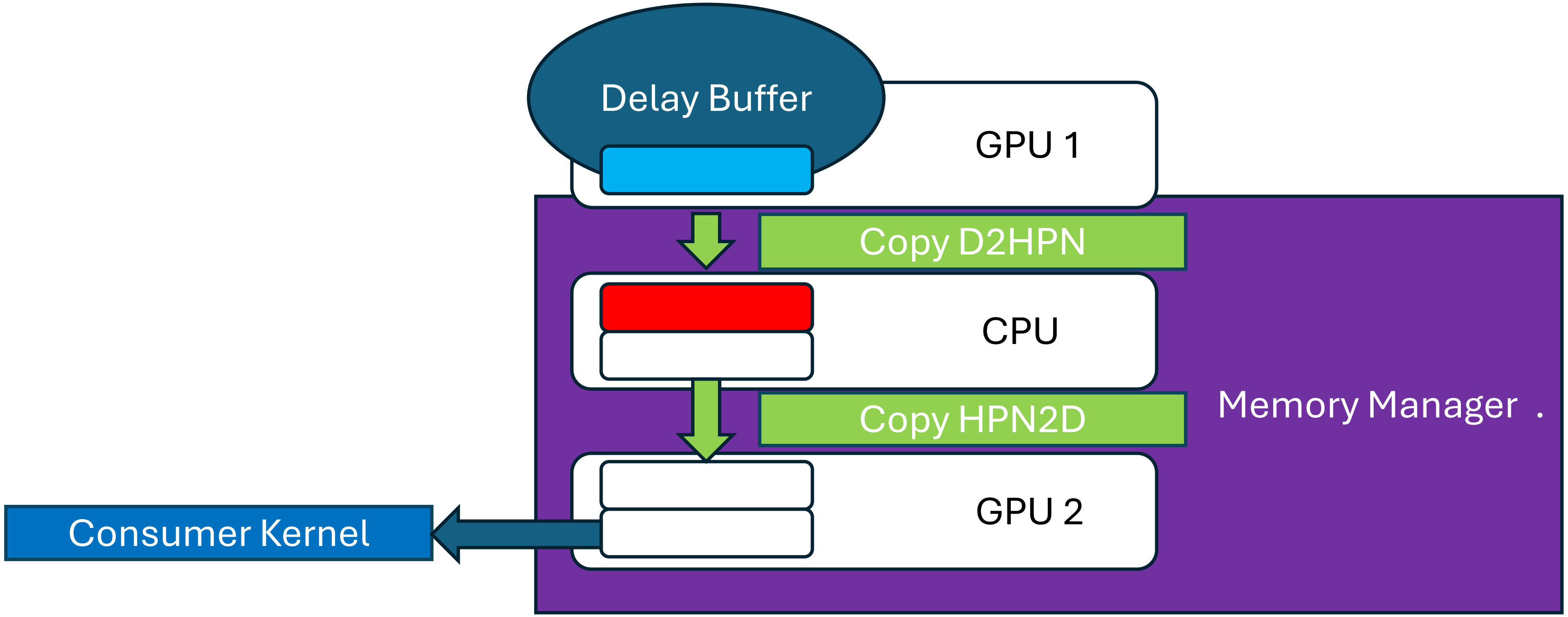
GPU communication utility

Memory manager: minimum delay possible

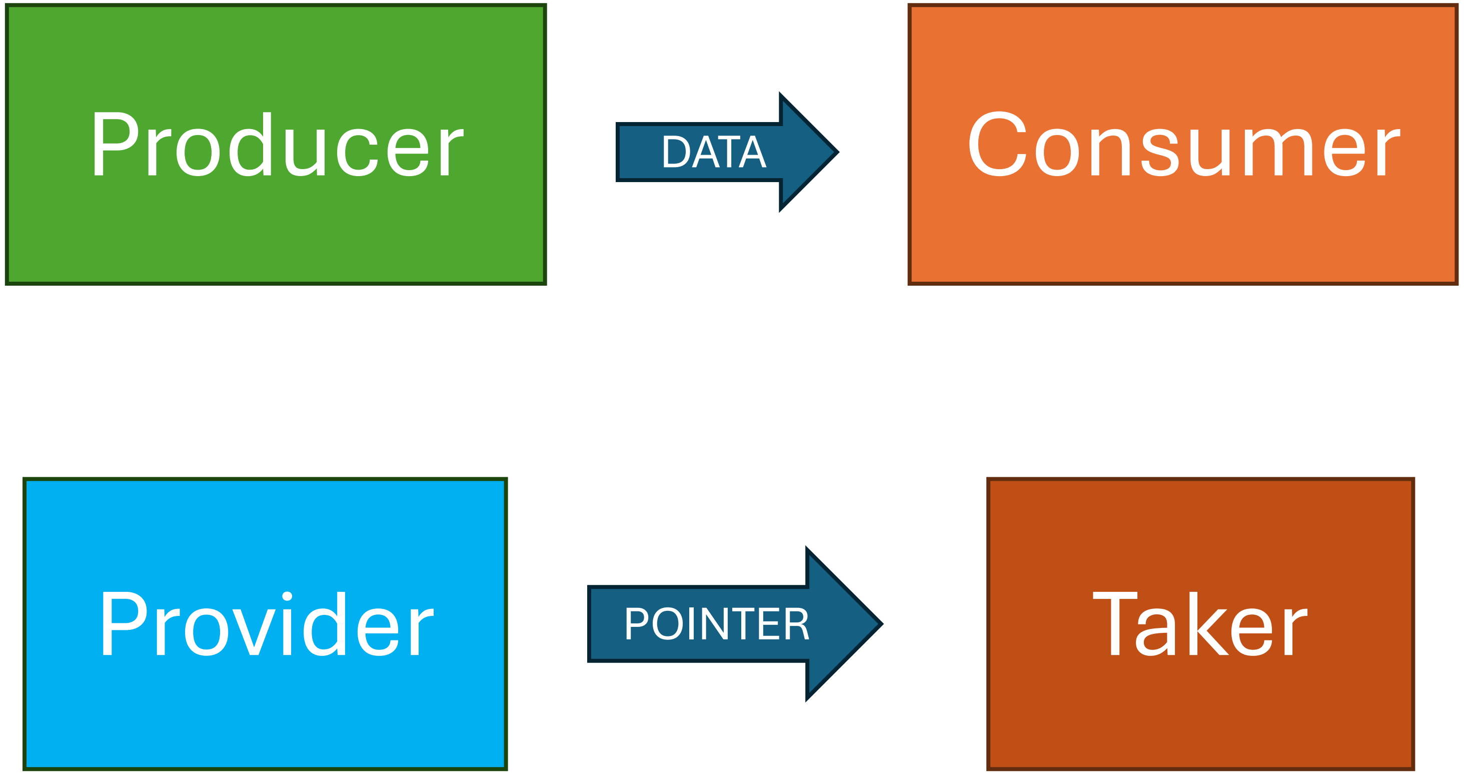


GPU communication utility

Memory manager: minimum copies and delay possible



GPU communication utility



Producer

DATA

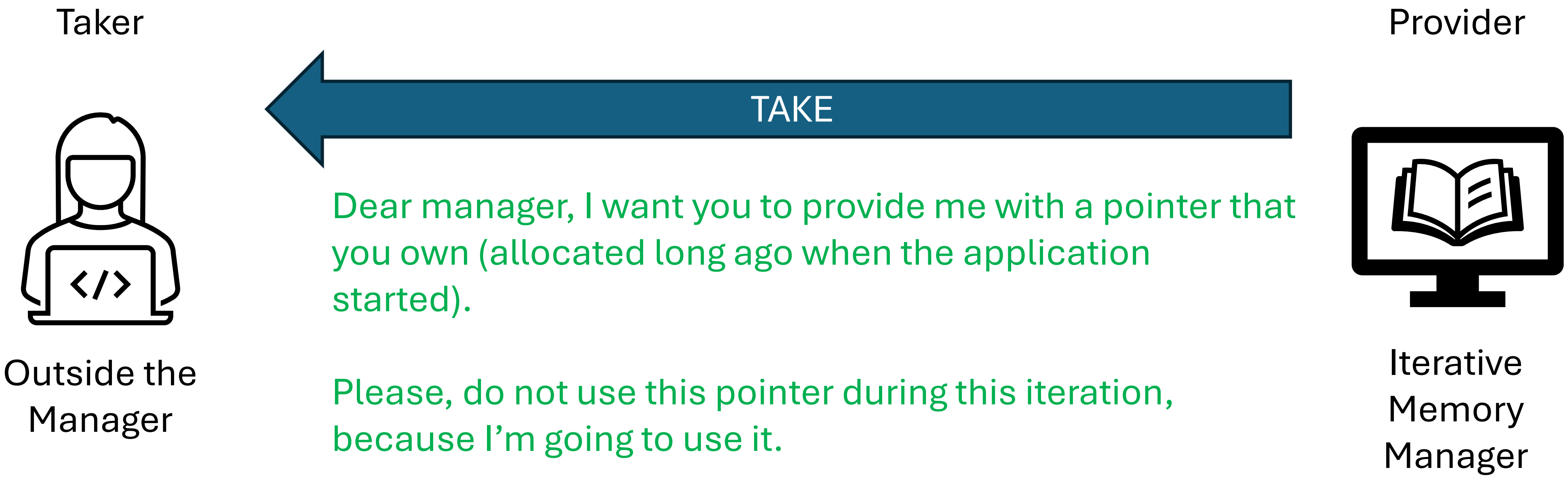
Consumer

Provider

POINTER

Taker

GPU communication utility



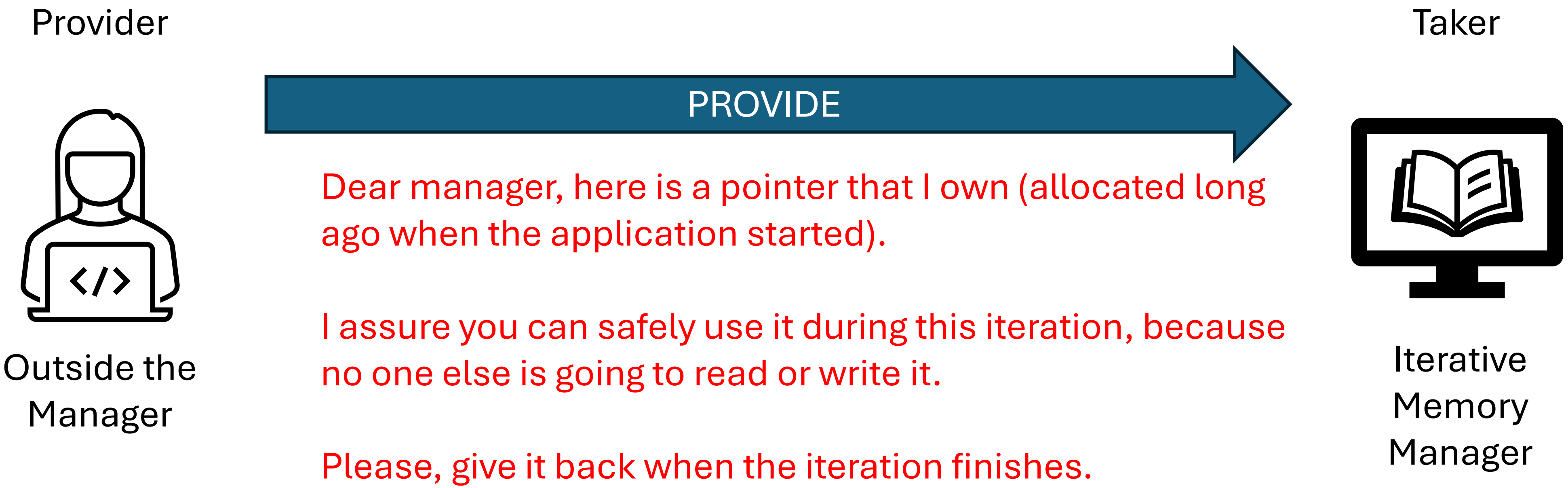
Dear manager, I want you to provide me with a pointer that you own (allocated long ago when the application started).

Please, do not use this pointer during this iteration, because I'm going to use it.

After this iteration, you can do whatever you need with it.

During this iteration, do your stuff with other pointers.

GPU communication utility



Provider

Outside the Manager

PROVIDE

Dear manager, here is a pointer that I own (allocated long ago when the application started).

I assure you can safely use it during this iteration, because no one else is going to read or write it.

Please, give it back when the iteration finishes.

Taker

Iterative Memory Manager

GPU communication utility

Producer Memory Space	Consumer Memory Space
ProducerTakes	ConsumerTakes
ProducerProvides	ConsumerTakes
ProducerTakes	ConsumerProvides
ProducerProvides	ConsumerProvides

GPU communication utility

Source MS	Destination MS	Flag	Number of copies
CPU	CPU	HPG2HPG	0
CPU	CPU Pinned	HPG2HPN	1
CPU	GPU	HPG2D	2 CPU -> Pinned -> GPU
CPU Pinned	CPU	HPN2HPG	1
CPU Pinned	CPU Pinned	HPN2HPN	0
CPU Pinned	GPU	HPN2D	1
GPU	CPU	D2HPG	2
GPU	CPU Pinned	D2HPN	1
GPU	GPU	D2D	0
GPU	Peer GPU	D2PD	1
GPU	Fake Peer GPU	D2FPD	2 GPU -> Pinned -> GPU

×

Producer MS	Consumer MS
Take	Take
Provide	Take
Take	Provide
Provide	Provide

GPU communication utility

```
struct DataInfo {
        int numElements;
        int elemSizeInBytes;
        MemorySpace memSpace;
};

DataInfo producerDataInfo{1024, 4, HostPageable};
DataInfo consumerDataInfo{1024, 4, Device_1};

Data ptrToProduce(producerDataInfo);
Data ptrToConsume(consumerDataInfo);

enum Actions { ProducerProvides, ProducerTakes, ConsumerProvides, ConsumerTakes};
```

GPU communication utility

```
//Initialization
MemoryManager<ProducerProvides, ConsumerProvides> manager(producerDataInfo, consumerDataInfo);

int delay = manager.getTotalDelay(); // Query delay generated by the manager

manager.manage(ptrToProduce, ptrToConsume); // Usage
```

GPU communication utility

```
// Initialization
MemoryManager<ProducerTakes, ConsumerTakes> manager(producerDataInfo, consumerDataInfo);

auto [ptrToProduce, ptrToConsume] = manager.manage(); // Usage
```

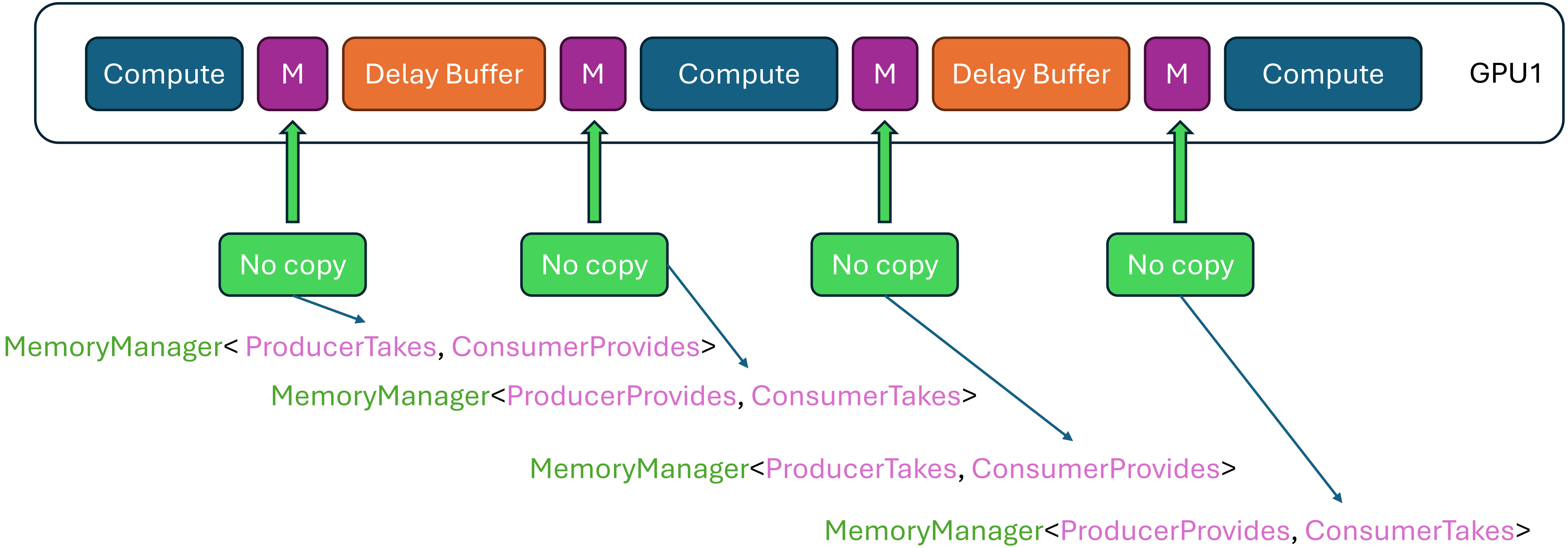
GPU communication utility

```
// Initialization
MemoryManager<ProducerTakes, ConsumerProvides> manager(producerDataInfo, consumerDataInfo);

ptrToProduce = manager.manage(ptrToConsume); // Usage
```

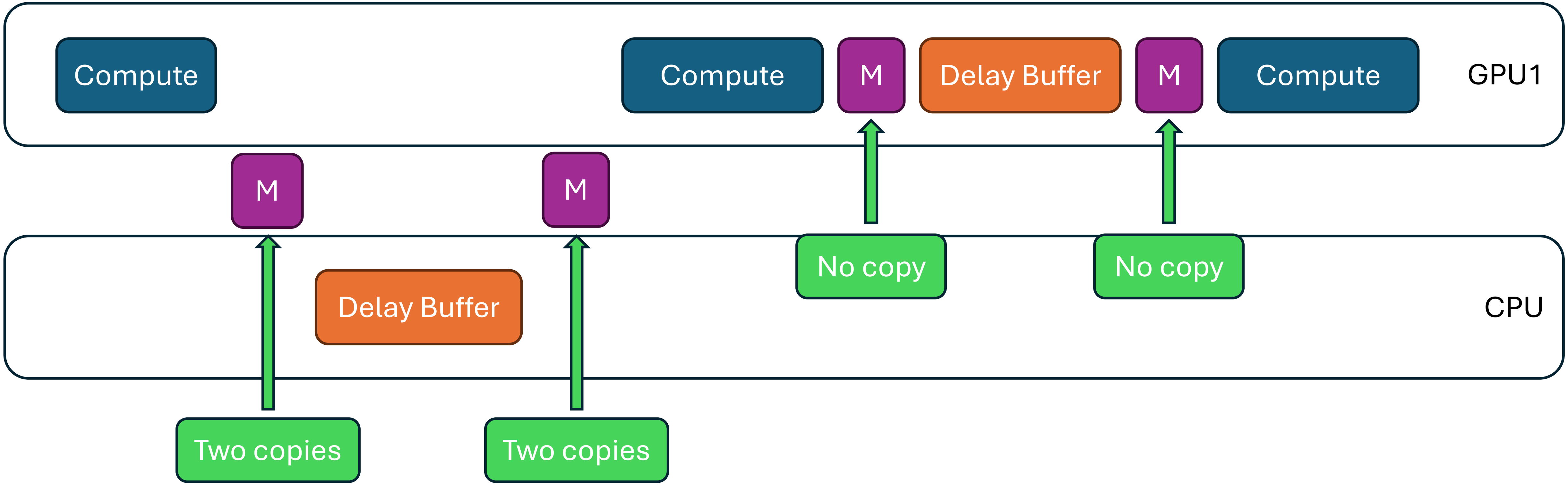
GPU communication utility

```
// Initialization
MemoryManager<ProducerProvides, ProducerTakes> manager(producerDataInfo, consumerDataInfo);

ptrToConsume = manager.manage(ptrToProduce); // Usage
```

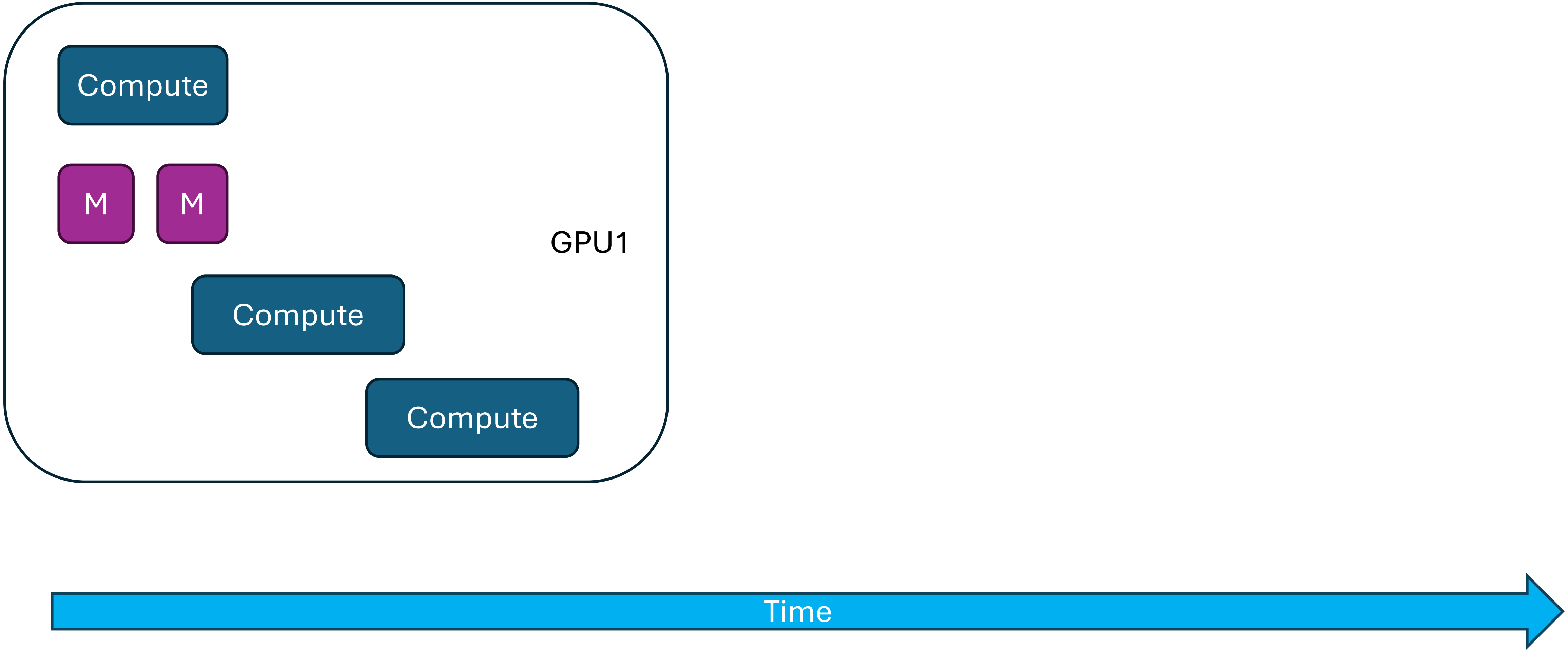
GPU communication utility


Compute
M
Delay Buffer
M
Compute
M
Delay Buffer
M
Compute
GPU1
No copy
No copy
No copy
No copy
MemoryManager< ProducerTakes, ConsumerProvides>
MemoryManager<ProducerProvides, ConsumerTakes>
MemoryManager<ProducerTakes, ConsumerProvides>
MemoryManager<ProducerProvides, ConsumerTakes>

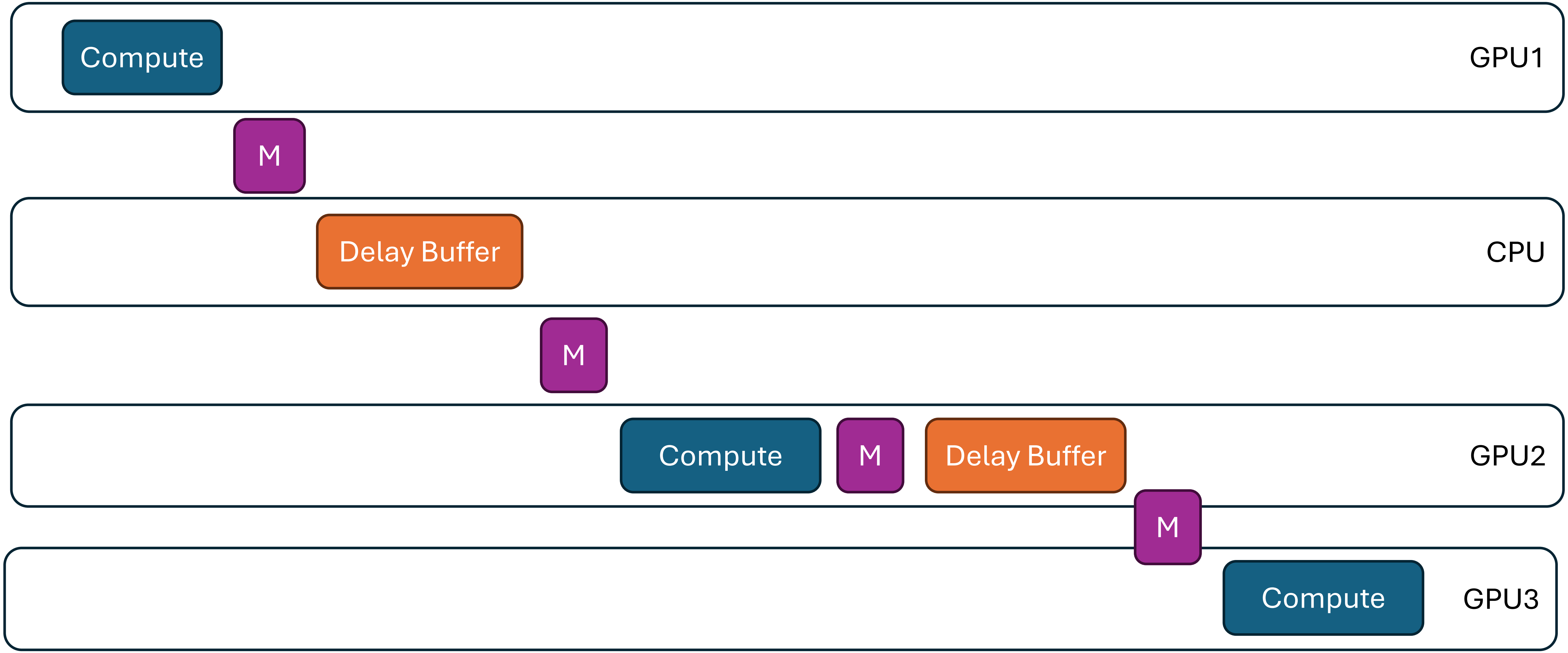
GPU communication utility



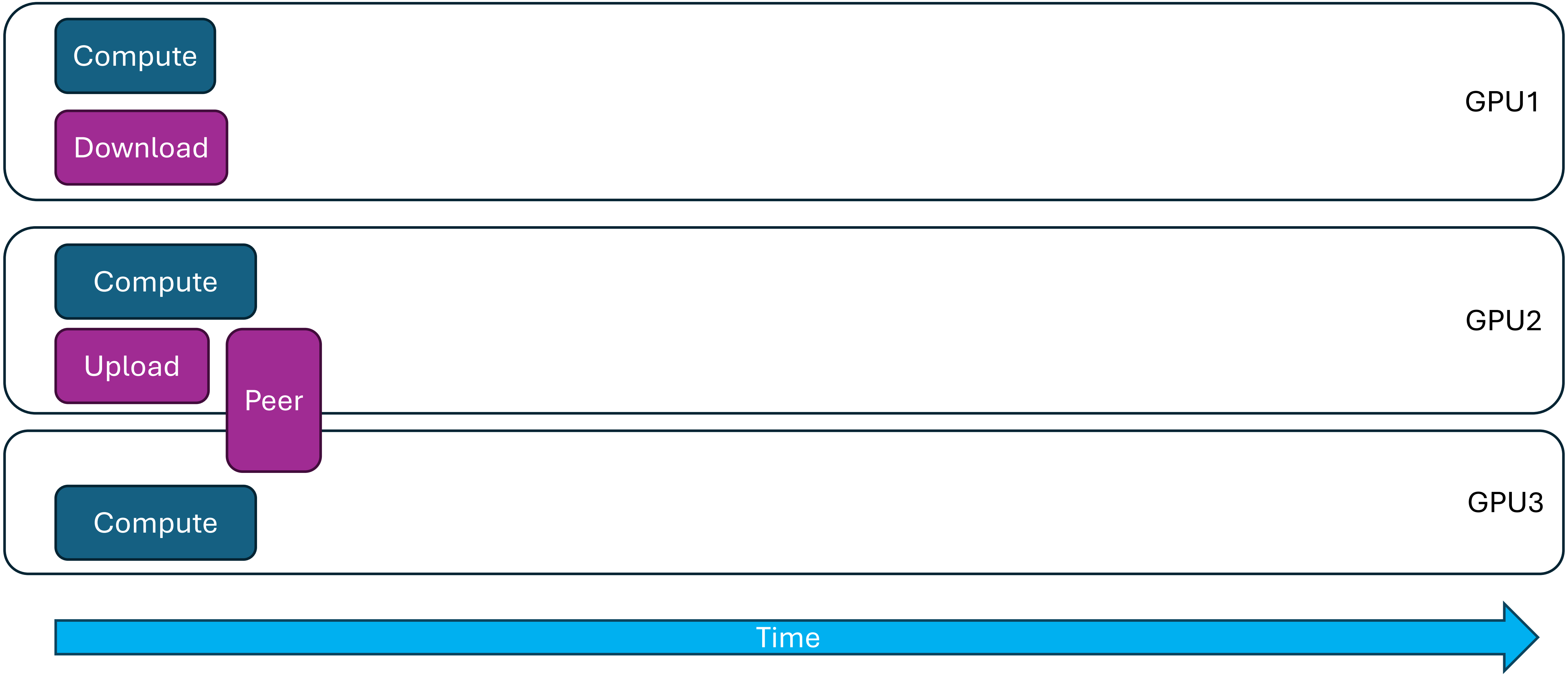
GPU communication utility



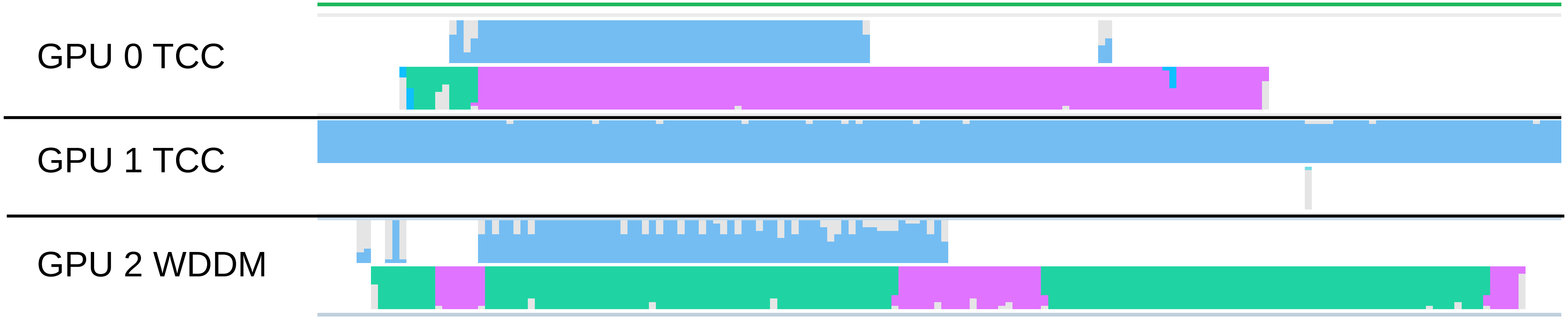
GPU communication utility

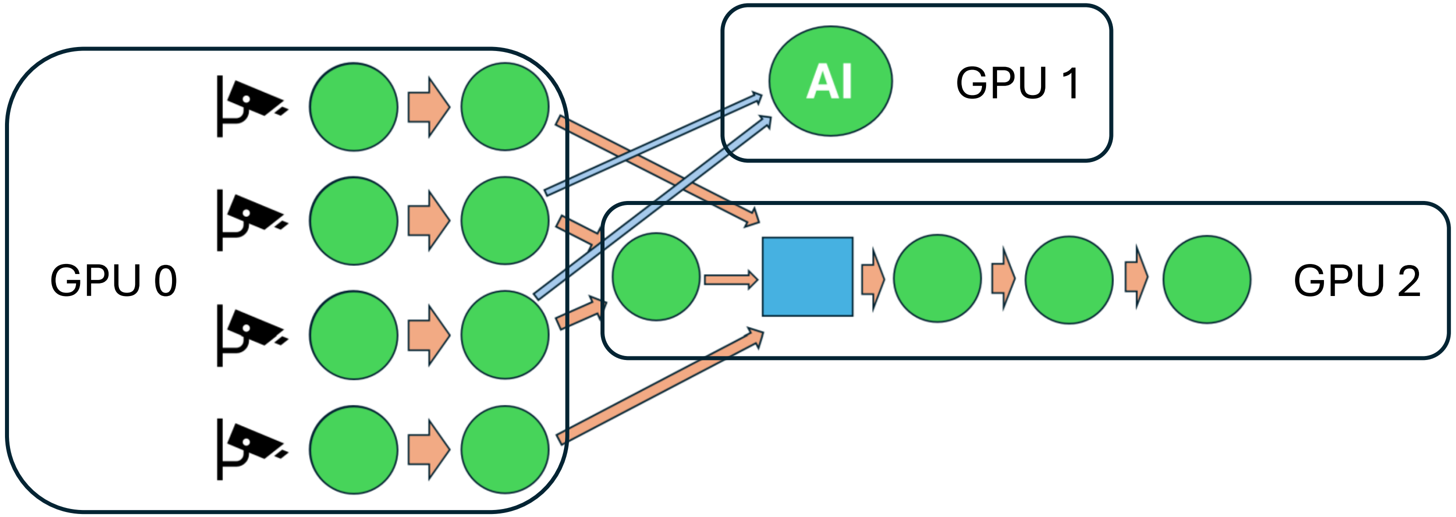


GPU communication utility



GPU communication utility





GPU communication utility

Libraries that provide solutions to similar problems

Image processing graphs:

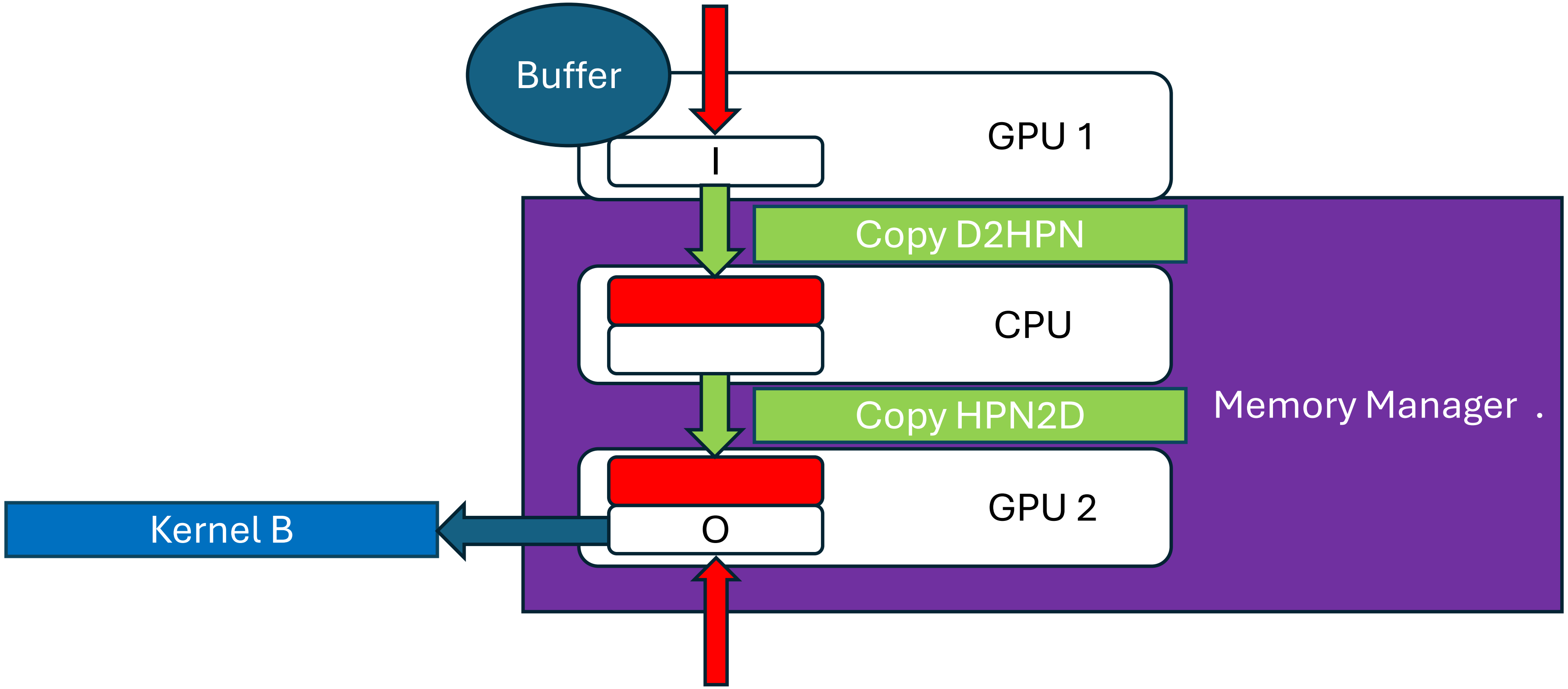
DeepStream SDK | NVIDIA Developer | NVIDIA Developer

Multi-gpu and multi-node CUDA programs (training DNNs and more):

NVIDIA Collective Communications Library (NCCL) | NVIDIA Developer

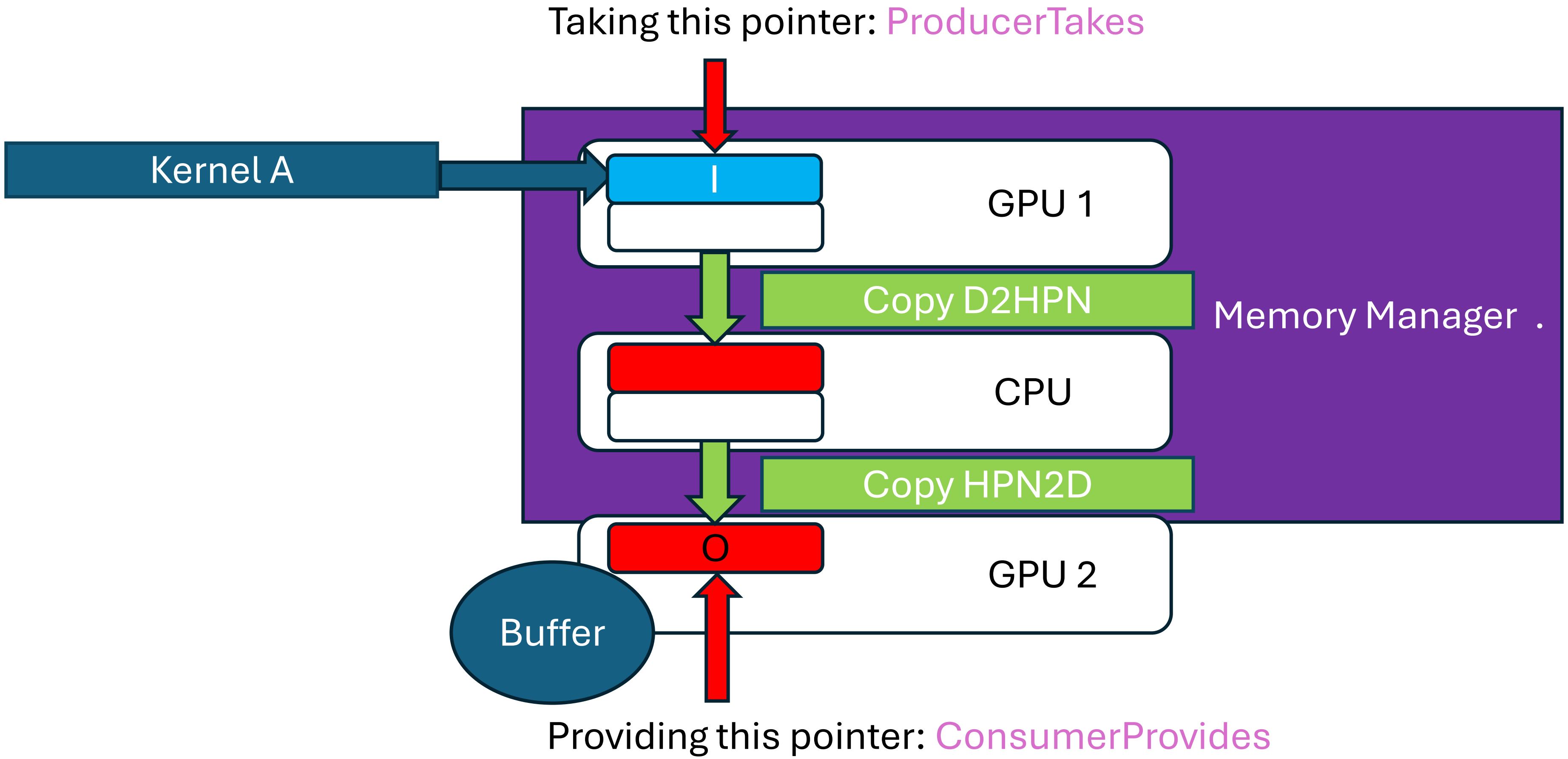
GPU communication utility

Providing this pointer: ProducerProvides

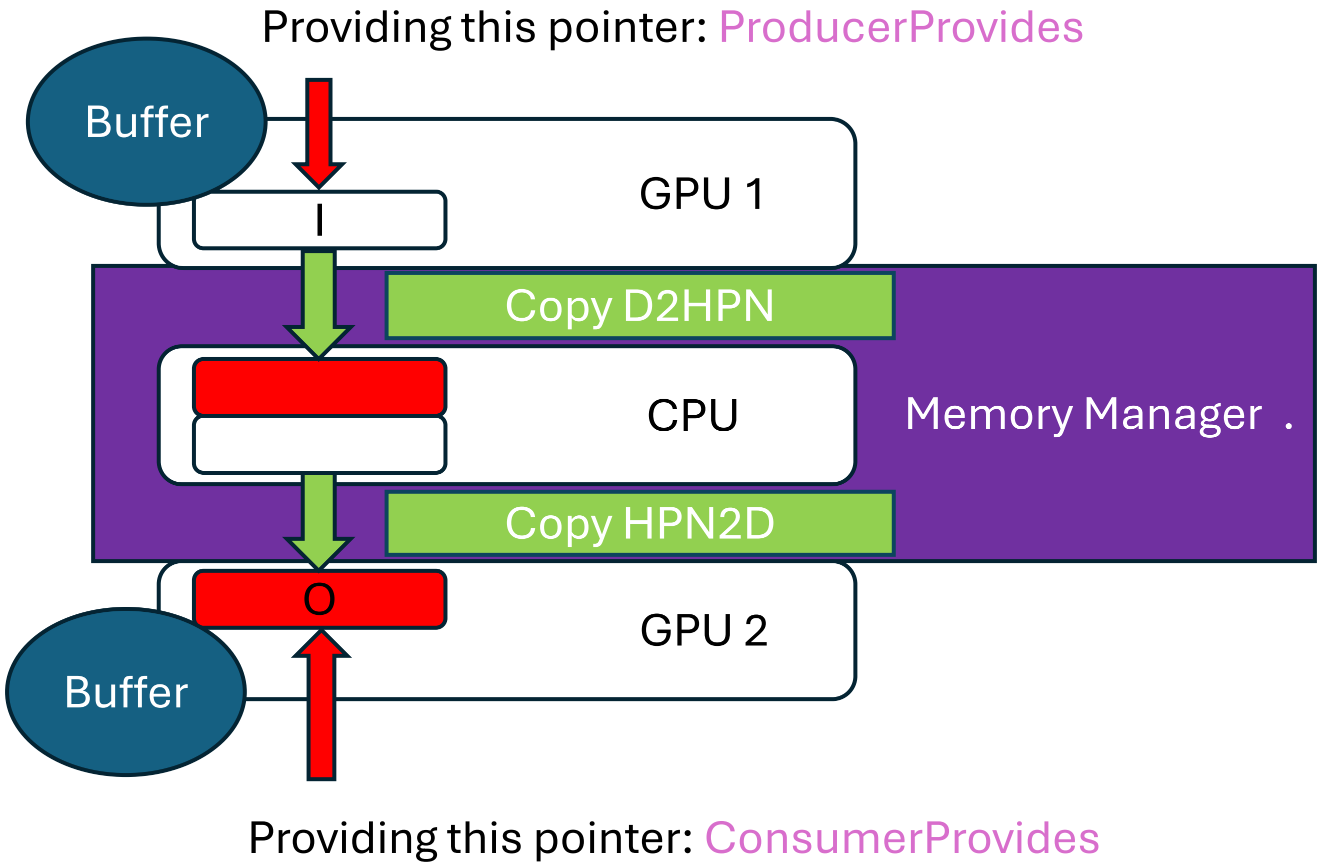


Taking this pointer: ConsumerTakes

GPU communication utility

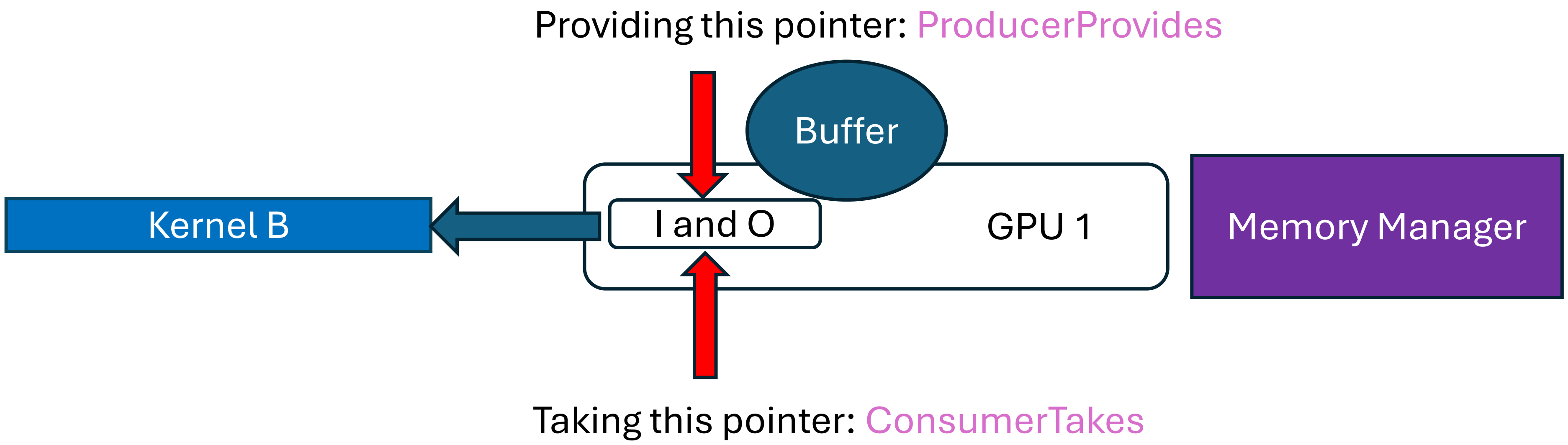

Taking this pointer: ProducerTakes
Kernel A
I
GPU 1
Copy D2HPN
Memory Manager .
CPU
Copy HPN2D
O
GPU 2
Buffer
Providing this pointer: ConsumerProvides

GPU communication utility


Providing this pointer: ProducerProvides
Buffer
GPU 1
I
Copy D2HPN
CPU
Memory Manager .
Copy HPN2D
O
GPU 2
Buffer
Providing this pointer: ConsumerProvides

GPU communication utility

Zero copy case


Providing this pointer: ProducerProvides
Buffer
Kernel B
I and O
GPU 1
Memory Manager
Taking this pointer: ConsumerTakes

GPU communication utility

Rules to know when to use **Provide**:

- If the Producer code it’s a Delay Buffer.
- Equivalent to saying: if the Producer code **does not modify** the data during this iteration.
- Then Producer **must** Provide: ProducerProvides
- If the Consumer code it’s a Delay Buffer.
- Equivalent to saying: if the Consumer code **does not modify** the data during this iteration.
- Then Consumer **must** Provide: ConsumerProvides

GPU communication utility

Rules to know when to use **Take**:

- If the Producer code it’s a Kernel or asynchronous CPU code.
- Equivalent to saying: if the Producer code **does modify** the data during this iteration.
- Then Producer **must** Take: ProducerTakes
- If the Consumer code it’s a Kernel or asynchronous CPU code.
- Equivalent to saying: if the Consumer code **does modify** the data during this iteration.
- Then Consumer **must** Take: ConsumerTakes

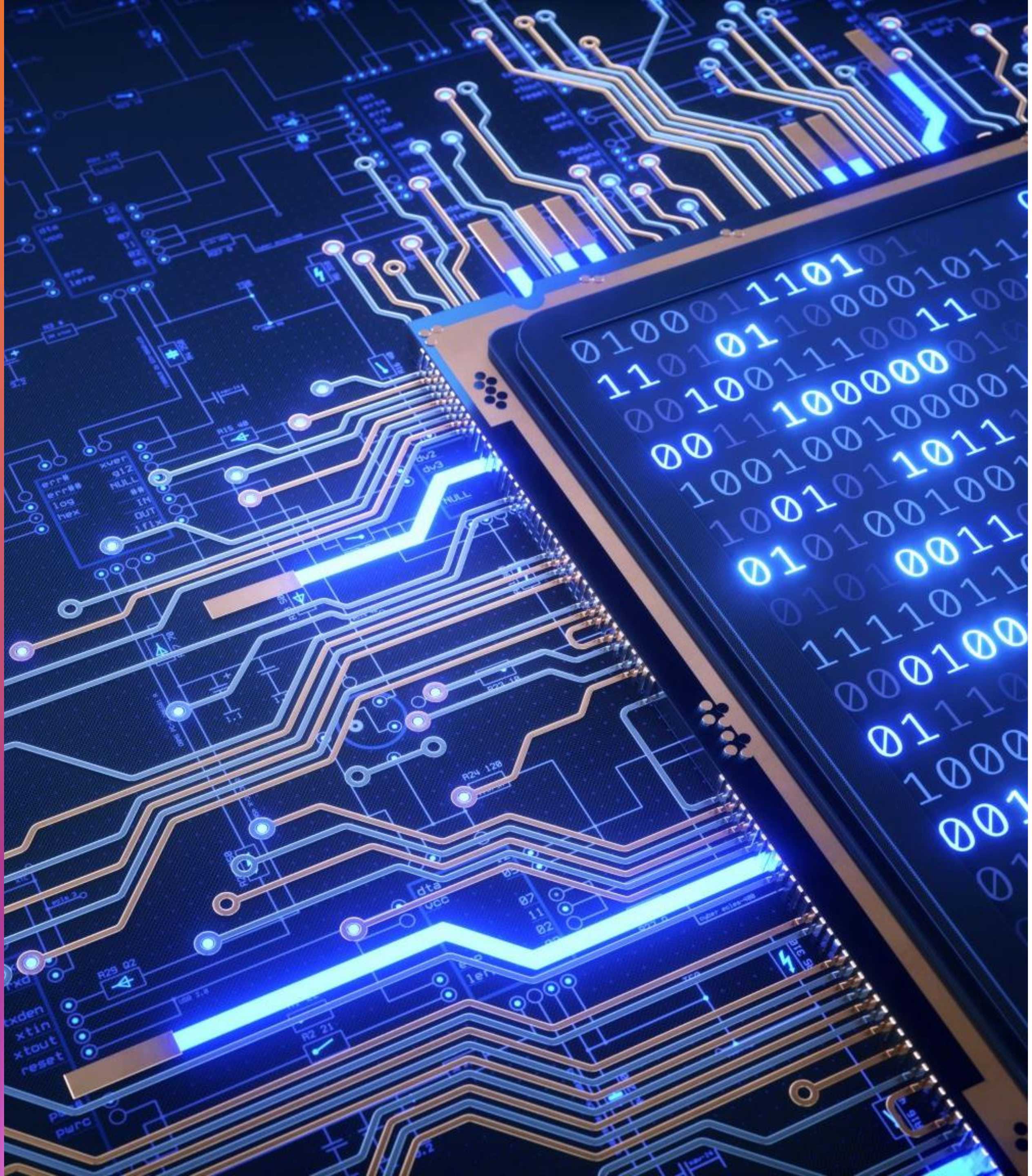
Lecture overview

Summary of already seen concepts

Main story: libraries

Use cases:

- GPU communication utility
- GPU kernel libraries



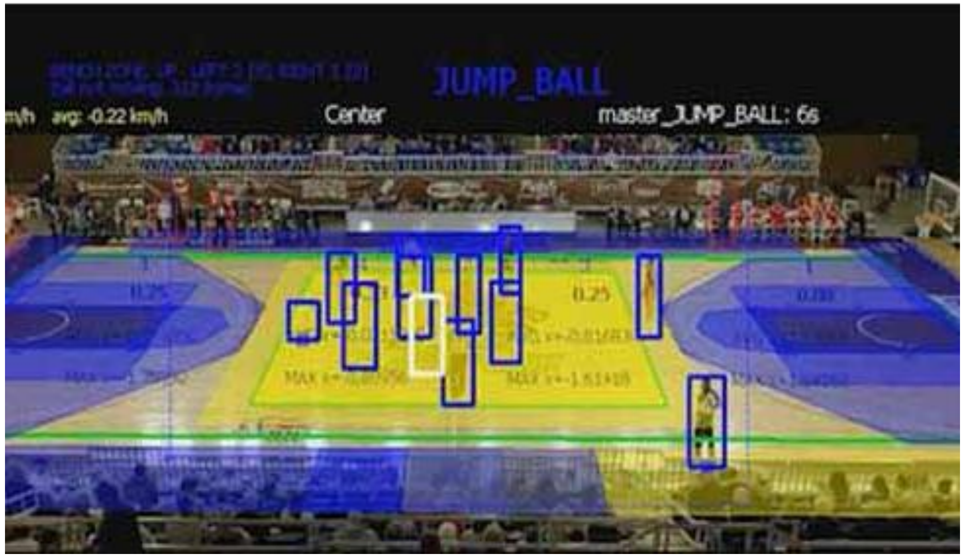
GPU kernel libraries

OpenSource personal project:

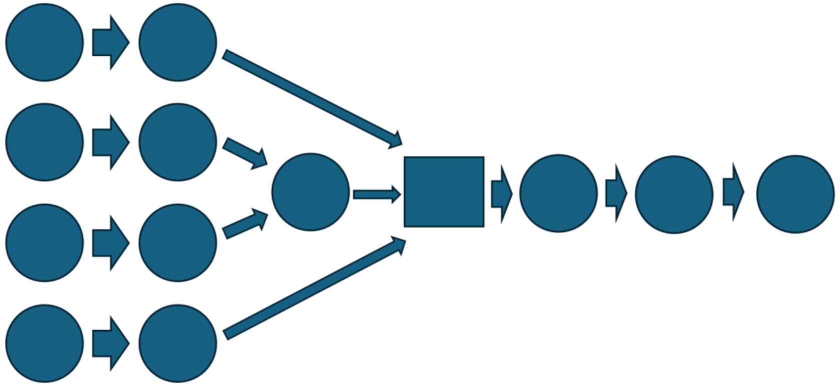
morousg/cvGPUSpeedup: A faster implementation of OpenCV-CUDA that uses OpenCV objects, and more! (github.com)

GPU kernel libraries

- The inference image preprocessing

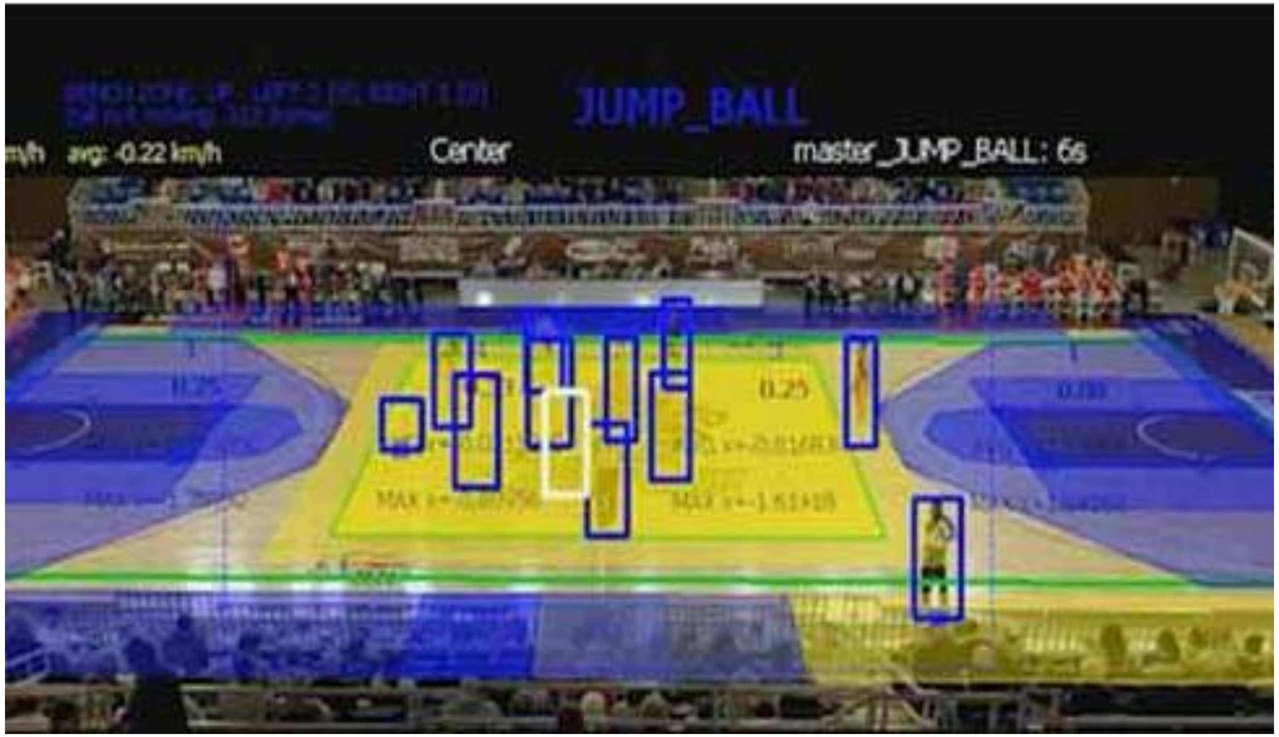


- The image processing pipeline

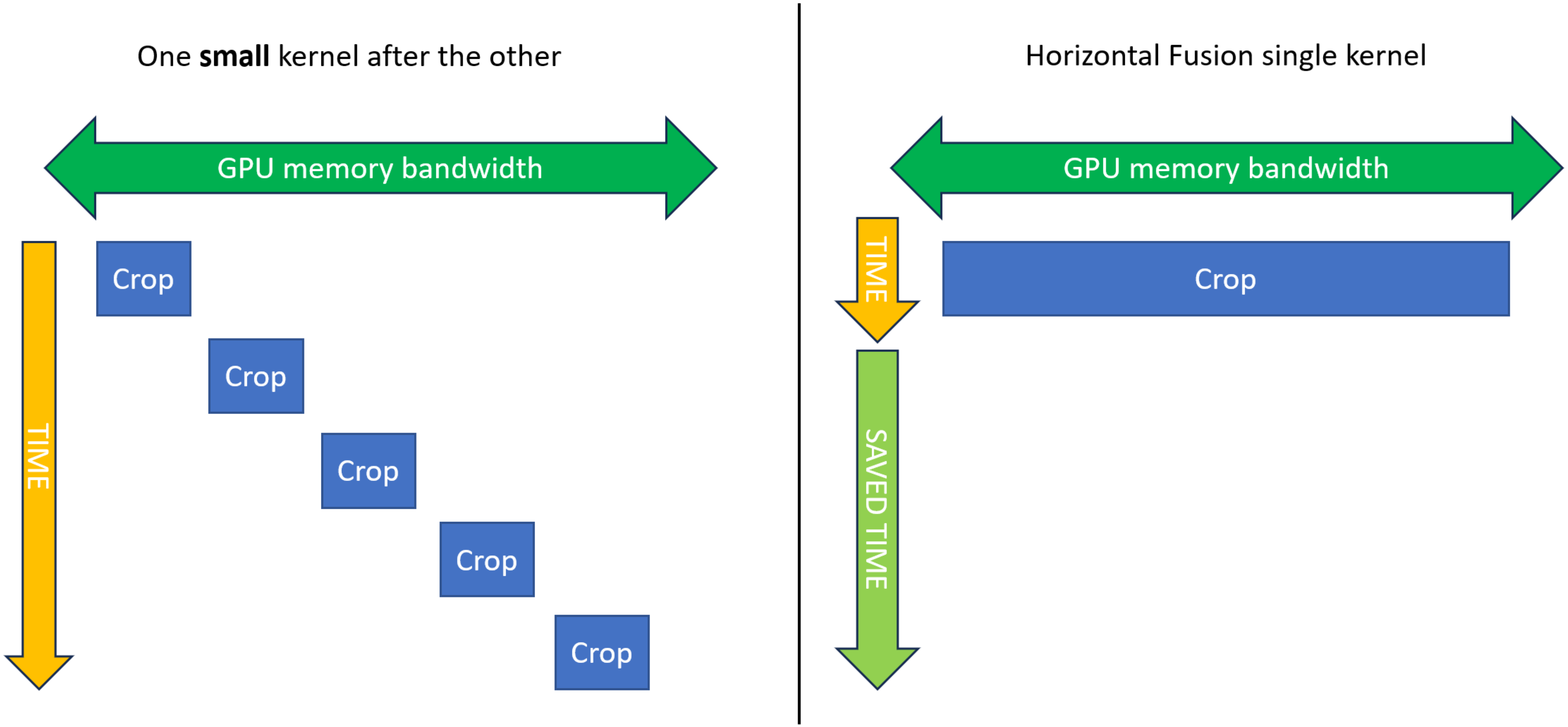


GPU kernel libraries:

- AI team uses OpenCV-CUDA -> extremely memory bound kernels
- Many small independent and dependent kernels



GPU kernel libraries: Horizontal Fusion



One **small** kernel after the other

GPU memory bandwidth

TIME

Crop

Crop

Crop

Crop

Crop

Horizontal Fusion single kernel

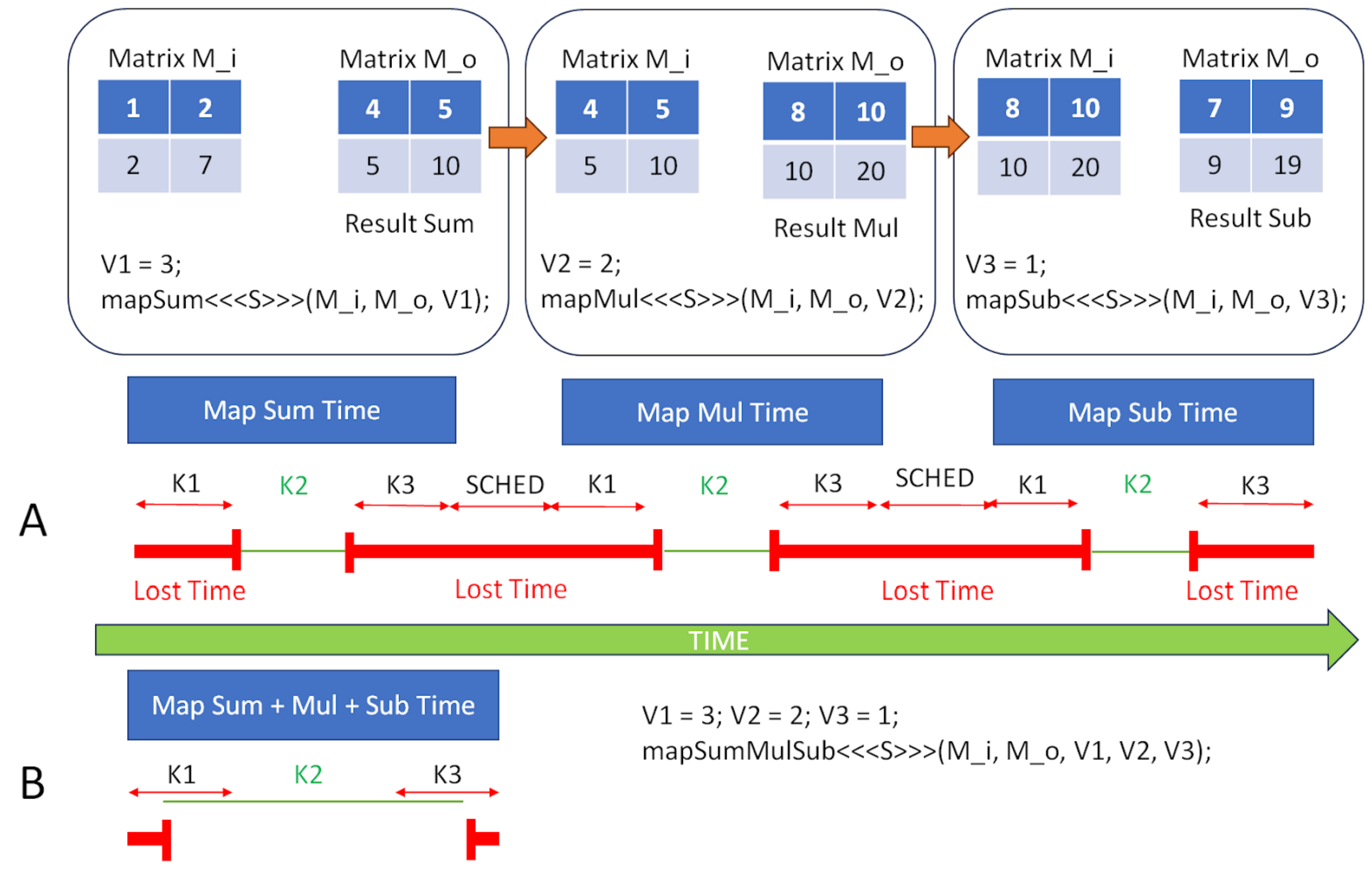
GPU memory bandwidth

TIME

Crop

SAVED TIME

GPU kernel libraries: Vertical Fusion



Matrix M_i

1	2
2	7

Matrix M_o

4	5
5	10

Result Sum

```
V1 = 3;
mapSum<<<S>>>(M_i, M_o, V1);
```

Matrix M_i

4	5
5	10

Matrix M_o

8	10
10	20

Result Mul

```
V2 = 2;
mapMul<<<S>>>(M_i, M_o, V2);
```

Matrix M_i

8	10
10	20

Matrix M_o

7	9
9	19

Result Sub

```
V3 = 1;
mapSub<<<S>>>(M_i, M_o, V3);
```

```
V1 = 3; V2 = 2; V3 = 1;
mapSumMulSub<<<S>>>(M_i, M_o, V1, V2, V3);
```

GPU kernel libraries: Fusion

Vertical Fusion for Data Dependent Kernels

Horizontal Fusion for Data Independent Kernels

GPU kernel libraries: novel concepts implemented in cvGPUSpeedup library

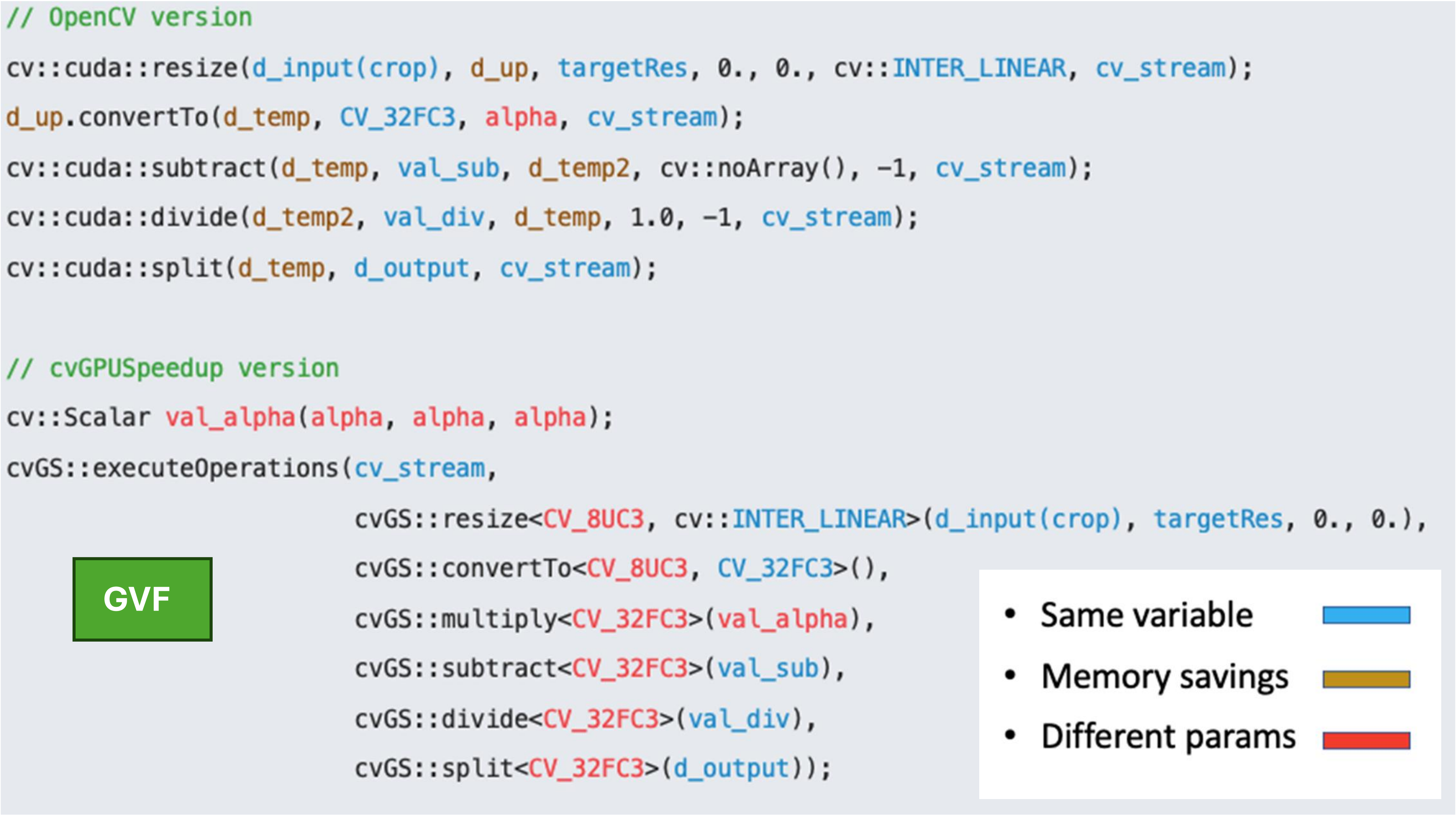
- Generic Vertical Fusion (GVF)
- Divergent Horizontal Fusion (DHF)
- Backwards Generic Vertical Fusion (BGVF)
- Automatic Thread Coarsening (ATC)

GPU kernel libraries

- **Typical approach for Vertical Fusion:**
  - Library defines a set of Vertically Fused Kernels (__global__ functions)
  - Explicitly chosen by the user
  - Implicitly chosen by a runtime
- **Our proposal:**
  - The library provides a set of fusionable __device__ functions
  - The final user, defines the chain of functions that go into the kernel, **without knowing it**.
  - We call this **Generic Vertical Fusion**

GPU kernel libraries: Generic Vertical Fusion

No Fusion

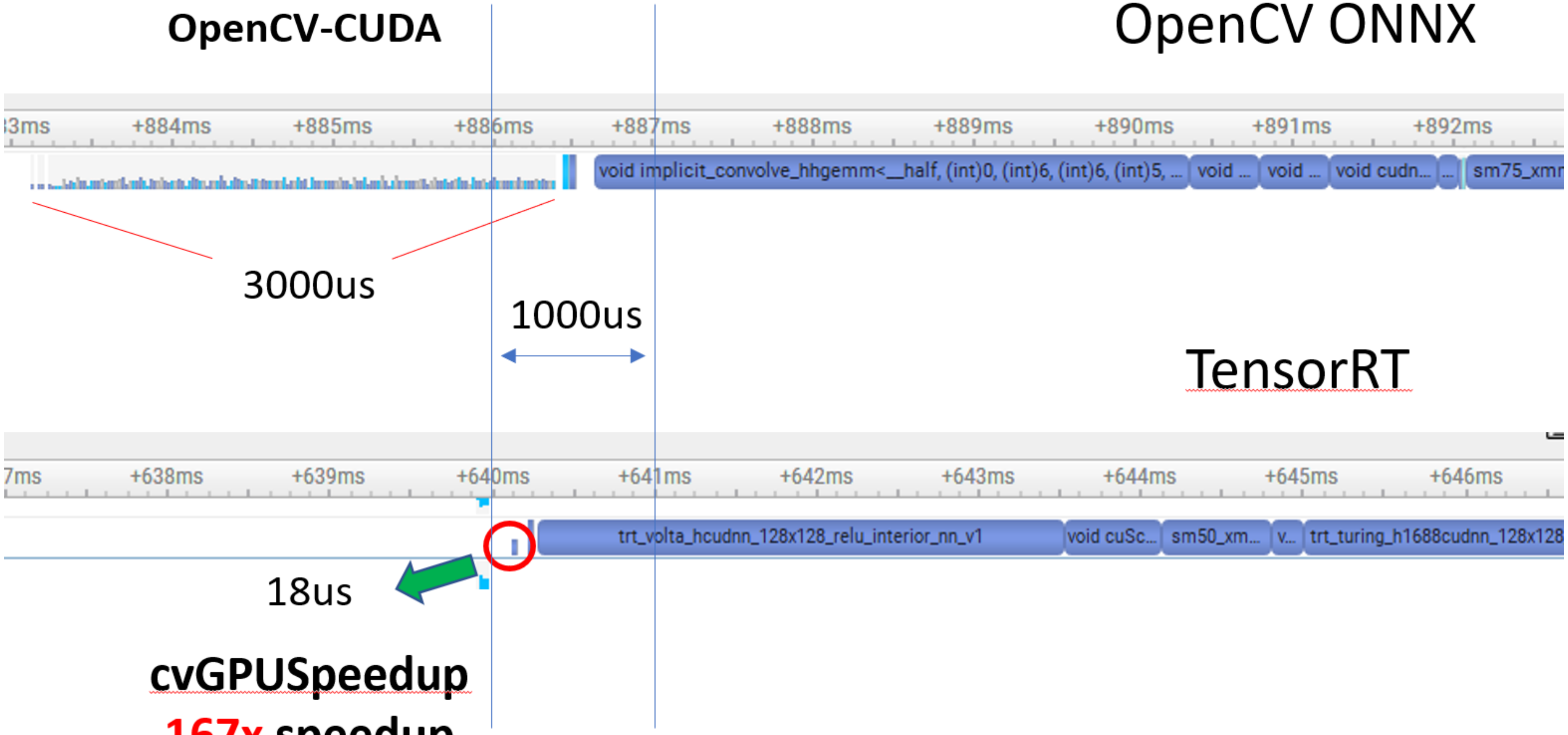


```
// OpenCV version
cv::cuda::resize(d_input(crop), d_up, targetRes, 0., 0., cv::INTER_LINEAR, cv_stream);
d_up.convertTo(d_temp, CV_32FC3, alpha, cv_stream);
cv::cuda::subtract(d_temp, val_sub, d_temp2, cv::noArray(), -1, cv_stream);
cv::cuda::divide(d_temp2, val_div, d_temp, 1.0, -1, cv_stream);
cv::cuda::split(d_temp, d_output, cv_stream);

// cvGPUSpeedup version
cv::Scalar val_alpha(alpha, alpha, alpha);
cvGS::executeOperations(cv_stream,
            cvGS::resize<CV_8UC3, cv::INTER_LINEAR>(d_input(crop), targetRes, 0., 0.),
            cvGS::convertTo<CV_8UC3, CV_32FC3>(),
            cvGS::multiply<CV_32FC3>(val_alpha),
            cvGS::subtract<CV_32FC3>(val_sub),
            cvGS::divide<CV_32FC3>(val_div),
            cvGS::split<CV_32FC3>(d_output));
```

GVF

- Same variable 🟦
- Memory savings 🟧
- Different params 🟥

GPU kernel libraries: Cr,Res,Nor for 50 Crops



OpenCV-CUDA

OpenCV ONNX

3000us

1000us

TensorRT

18us

cvGPUSpeedup

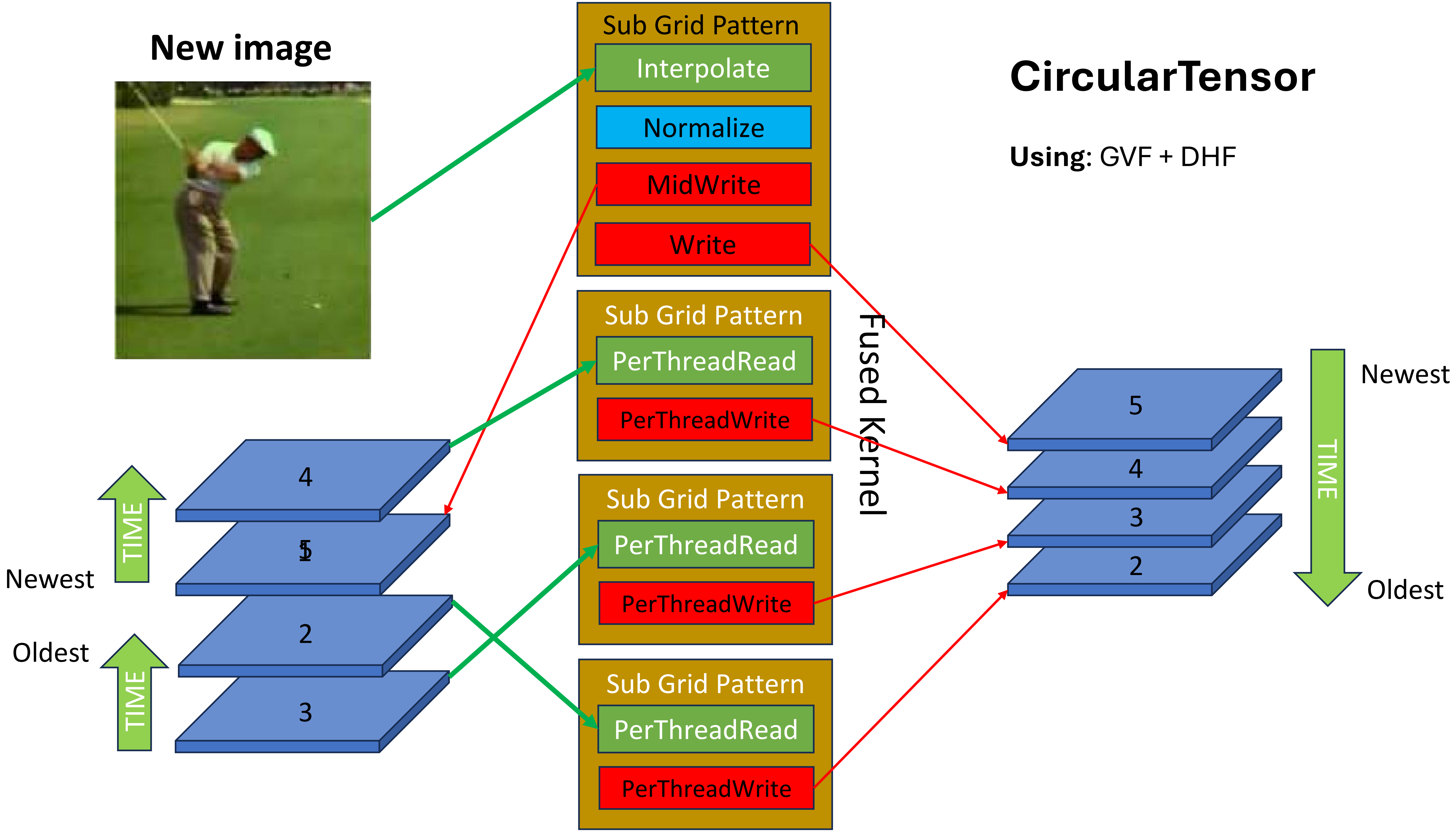
167x speedup

GPU kernel libraries: CircularTensor

```
cvGS::CircularTensor<InputCVType, CircularTensorCVType, NUM_CHANNELS, BATCH,
                     cvGS::CircularTensorOrder::NewestFirst> myTensor(WIDTH, HEIGHT);


myTensor.update(cv_stream,
                cvGS::resize<…>(newImage, …),
                cvGS::convertTo<…>(…),
                cvGS::split<…>(myTensor.ptr().data)); // We may look for a way to avoid this


// Now you can send the raw data to inference
network.forward(myTensor.ptr().data, cv_stream);
```

CircularTensor

Using: GVF + DHF



New image

Sub Grid Pattern

Interpolate

Normalize

MidWrite

Write

Sub Grid Pattern

PerThreadRead

PerThreadWrite

Sub Grid Pattern

PerThreadRead

PerThreadWrite

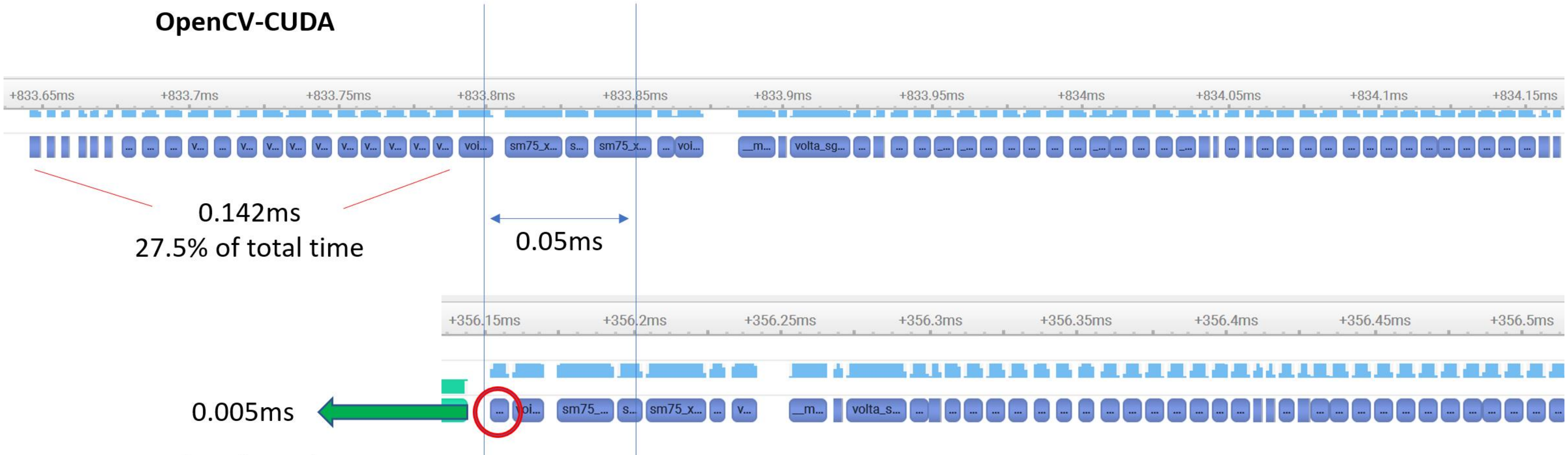
Sub Grid Pattern

PerThreadRead

PerThreadWrite

Fused Kernel

GPU kernel libraries



OpenCV-CUDA

0.142ms
27.5% of total time

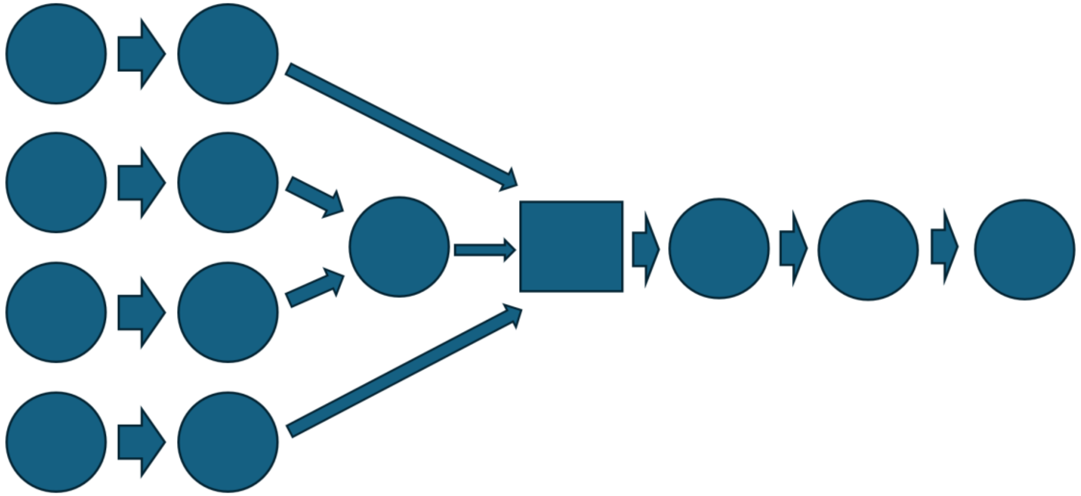
0.05ms

0.005ms

cvGPUSpeedup
28.4x speedup
0.9% of total time

GPU kernel libraries: image processing

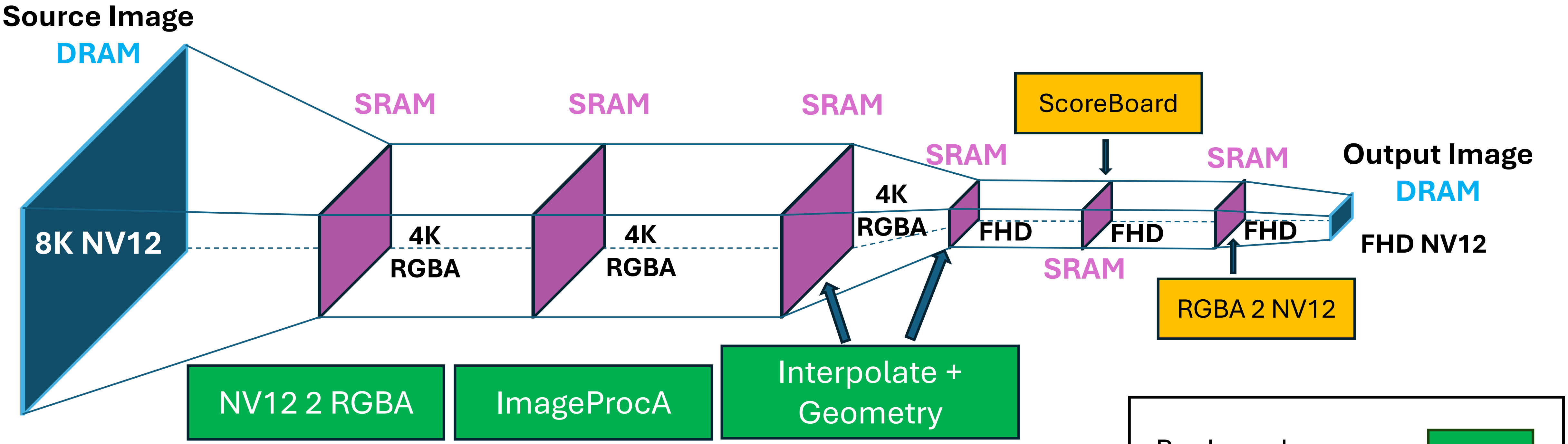
- Hand made kernels
- Mostly memory bound -> repetitive optimizations
- Read uchar4, process float4, store uchar4 -> loss of image quality



GPU kernel libraries: graph total fusion

```
using HowToReadAPixel = Binary<ReadYUV<NV12>, ConvertYUVToRGB<NV12, Full, bt709, AddAlpha, float4>, ImageProcA<float4>>;

HowToReadAPixel readDF;
get_params<0>(readDF) = d_nv12Image; // Source 8K image (ReadYUV Device Function)
get_params<2>(readDF) = imgProcAParams; // Parameters required by the ImageProcA Device Function.
// Applying Backwards Generic Vertical Fussion (BGVF)
auto howToInterpolateAPixelDF = resize<HowToReadAPixel, INTER_LINEAR>(readDF.params,
                                                 Size(d_nv12Image.dims().width, d_nv12Image.dims().height),
                                                 Size(targetWidth, targeHeight));

// Not present in the OpenSource library, approximated code, more BGVF
auto howToTransformAPixelDF = geometryFuntionBuilder(howToInterpolateAPixelDF, stitchParams);
auto generateOutputImageWithScoreBoardDF = scoreBoardFunctionBuilder(howToTransformAPixelDF, scoreBoardParams);

// Launch a single CUDA kernel, that does everything
executeOperations(stream, generateOutputImageWithScoreBoardDF, // Generic Vertical Fusion after the BGVF
                          Binary<ConvertRGBAToYUV<…>>{},
                          Write<ChromaSubSampling<NV12>,…>>{d_outputImage});
```

GPU kernel libraries: graph total fussion



4x less pixels processed than non fused kernels, thanks to BGVF

Pixels behind the ScoreBoard will not be processed by BDFs, thanks to BGVF

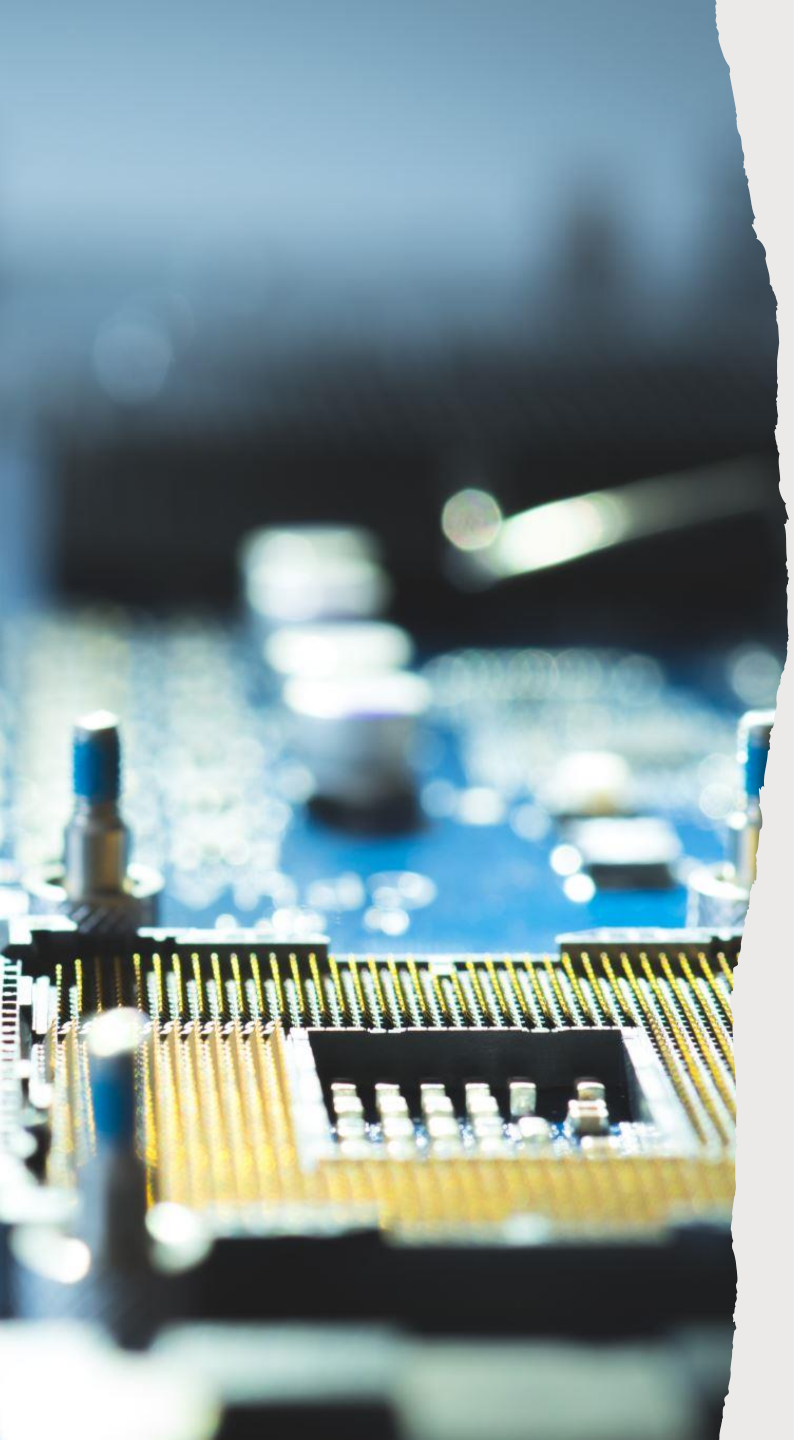
Backwards Device Functions 🟩 BDF

Forwards Device Functions 🟨 FDF

GPU kernel libraries: graph total fussion

Any one imagined a fused neural network with the previous slide?

It will require extra work, but yes, we have many ideas on how to get there, (including reductions, MMA etc...)



GPU kernel libraries

- **Automatic Thread Coarsening**:
  - Compile time detection of the possibility/convenience to apply it or not.
  - Requires zero user intervention (non CUDA programmers)
  - Requires CUDA ninjas to include it or not in their Read and Write Device Functions
  - Around 2x speedups for very memory bound kernels
  - Currently only active for 1 or 2 Byte data types:
    - Bigger types do not seem to give any speedup.
    - Further analysis will look into it.