# Explicit Multi GPU Programming with DirectX 12

**Juha Sjöholm**
Developer Technology Engineer
NVIDIA

# Agenda

- What is explicit Multi GPU
- API Introduction
- Engine Requirements
- Frame Pipelining – Case Study

# Problem With Implicit Multi GPU

## Ideal situation

- Driver does its magic
- Developer doesn‘t have to care
- It just works

## Reality

- Driver needs lots of hints
  - Clears, discards
  - Vendor specific APIs
- Developer needs to understand what driver is trying to do
- It still doesn’t always fly

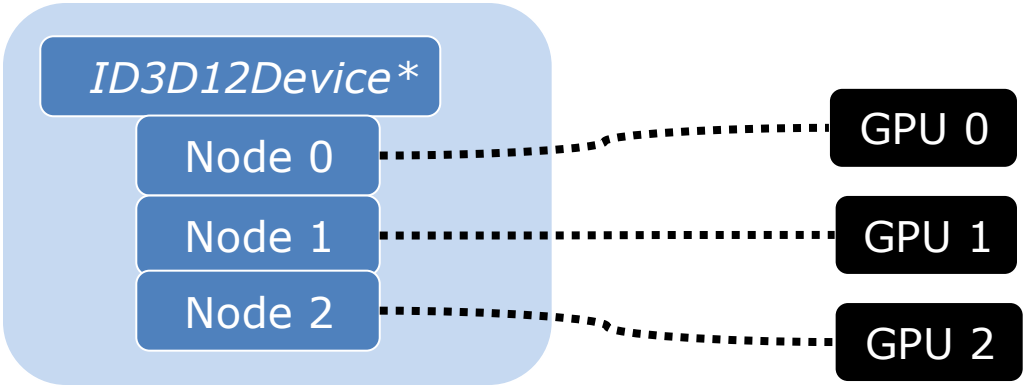
# What is Explicit Multi-GPU?

- Control cross GPU transfers
  - No unintended implicit transfers
- Control what work is done on each GPU
- Not just Alternate Frame Rendering (AFR)

# DX12 Explicit Multi GPU

- No more driver magic
- There is no driver level support for AFR
- Now you can do it better yourself, and much more!
- No vendor specific APIs needed

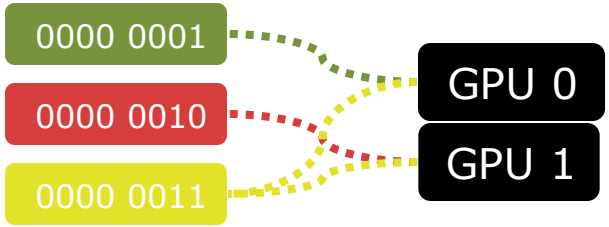
# Adapters – Linked Node Adapter



# Adapters – Multiple Adapters

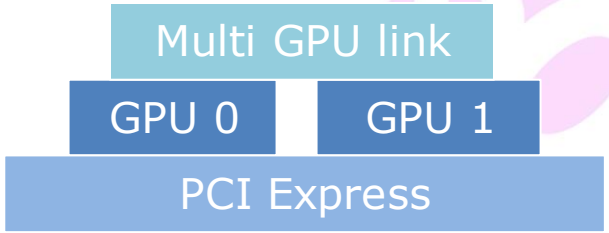
# Linked Node Adapter

- When user has enabled use of multiple GPUs in display driver, linked node mode is enabled
  - *IDXGIFactory::EnumAdapters1()* sees one adapter
  - *ID3D12Device::GetNodeCount()* tells node count
- Nodes (GPUs) are referenced with affinity masks
  - Node 0 = 0x1
  - Node 1 = 0x2
  - Node 1 and 2 = 0x3



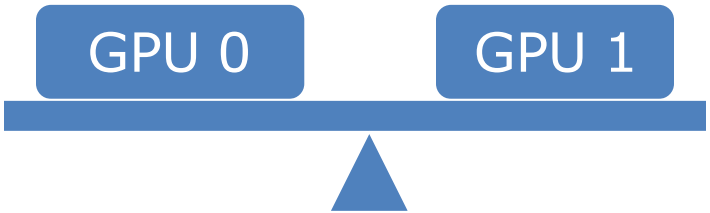
# Linked Node Features

- Resource copies directly from discrete GPU to discrete GPU – not through system memory
- Special support for AFR
  *IDXGISwapChain3::ResizeBuffers1()* allows utilization of other connections than PCIe when presenting frames
- Good for multiple discrete GPUs!



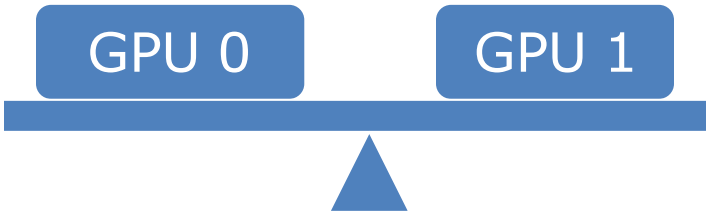
# Linked Node Load Balancing

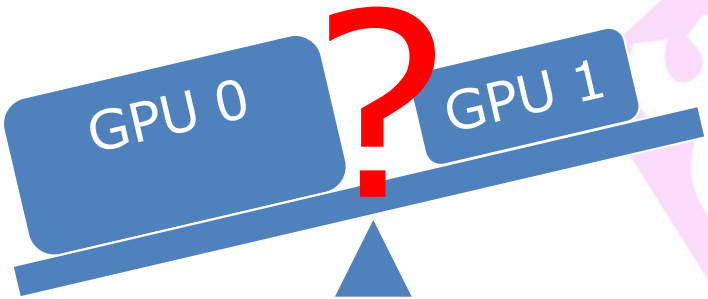
- It’s safe to assume that nodes are balanced for foreseeable future
  - Life is easy



# Linked Node Load Balancing

- It's safe to assume that nodes are balanced for foreseeable future
  - Life is easy
- Heterogeneous nodes may be available some day



# Infrastructure For Explicit M-GPU

- Renderer has to be aware of multiple GPUs
  - Expose multiple GPUs at right level
  - Wrap command queues, resources, descriptors, gpu virtual addresses etc. for multiple GPUs
- This can actually be the part that requires most effort
  - Once infrastructure exists, it’s easier to experiment

# Multi Node APIs

- With linked nodes, some things are very easy
- Some interfaces are omni node (no node mask)
  - Starting with *ID3D12Device*
- Some interfaces are multi node
  - Affinity mask can have more than one bit set
  - Root signatures, pipeline states and command signatures can be often just shared for all nodes

*ID3D12RootSignature**
*NodeMask* 0x3

*ID3D12PipelineState**
*NodeMask* 0x3

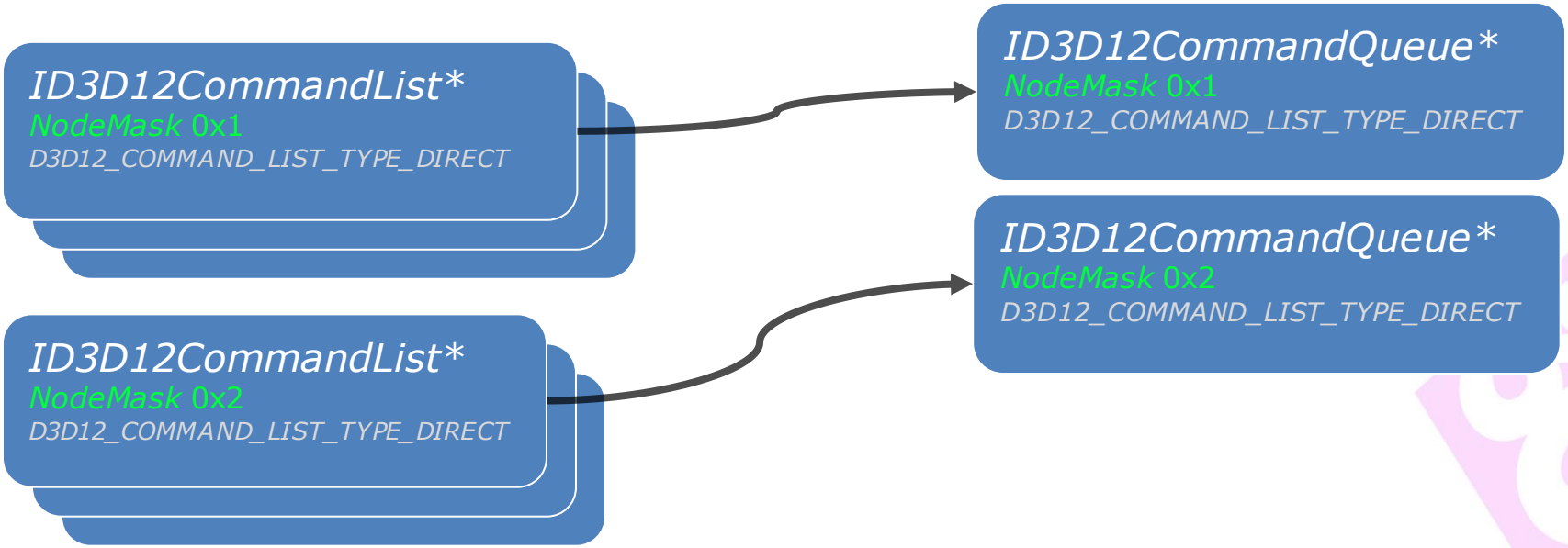
*ID3D12CommandSignature**
*NodeMask* 0x3

# Command Queues And Lists

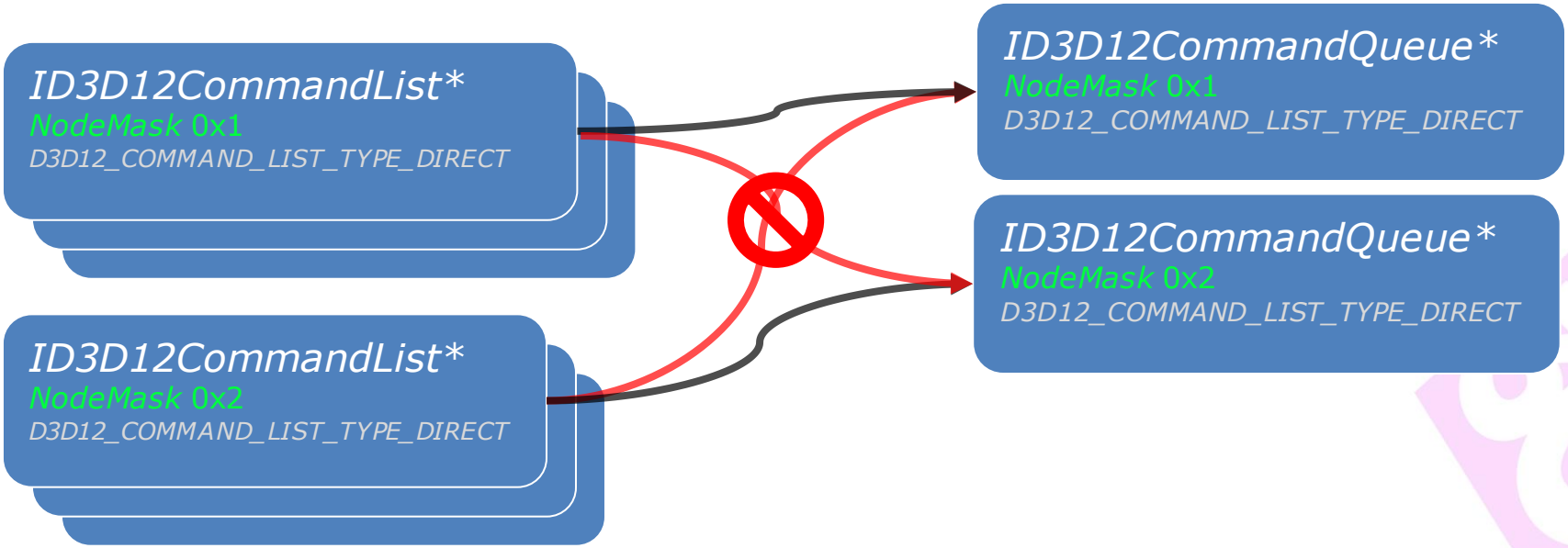
*ID3D12CommandQueue**
*NodeMask* 0x1
*D3D12_COMMAND_LIST_TYPE_DIRECT*

- Each node has its own *ID3D12CommandQueue, i.e. "engine"*
- *ID3D12CommandLists* are also exclusive to single node
  - Command list pooling for each node is needed

# Command List Pooling



# Command List Pooling

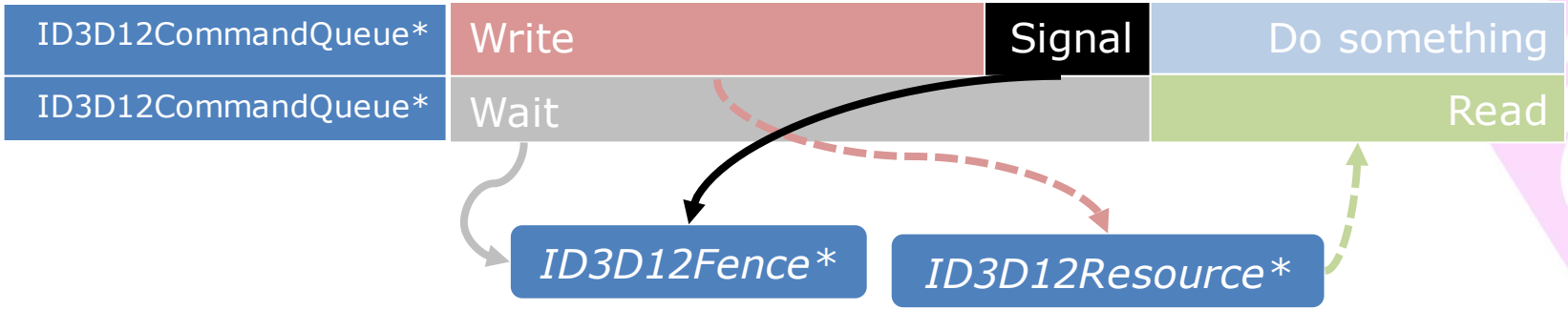


# Synchronization - Fences

- Different command queues need to be synchronized when sharing resources
- *ID3D12Fence* is the synchronization tool

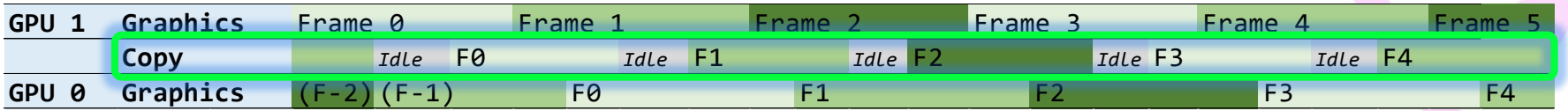
# Fences

- Application must avoid access conflicts
- Application must ensure that all engines see shared resources in same state



# Copy Engine(s)

- *ID3D12CommandQueue* with *D3D12_COMMAND_LIST_TYPE_COPY*
- Cross GPU copies *parallel* to other processing
- Remember to double buffer the resources



# Cross Node Sharing Tiers

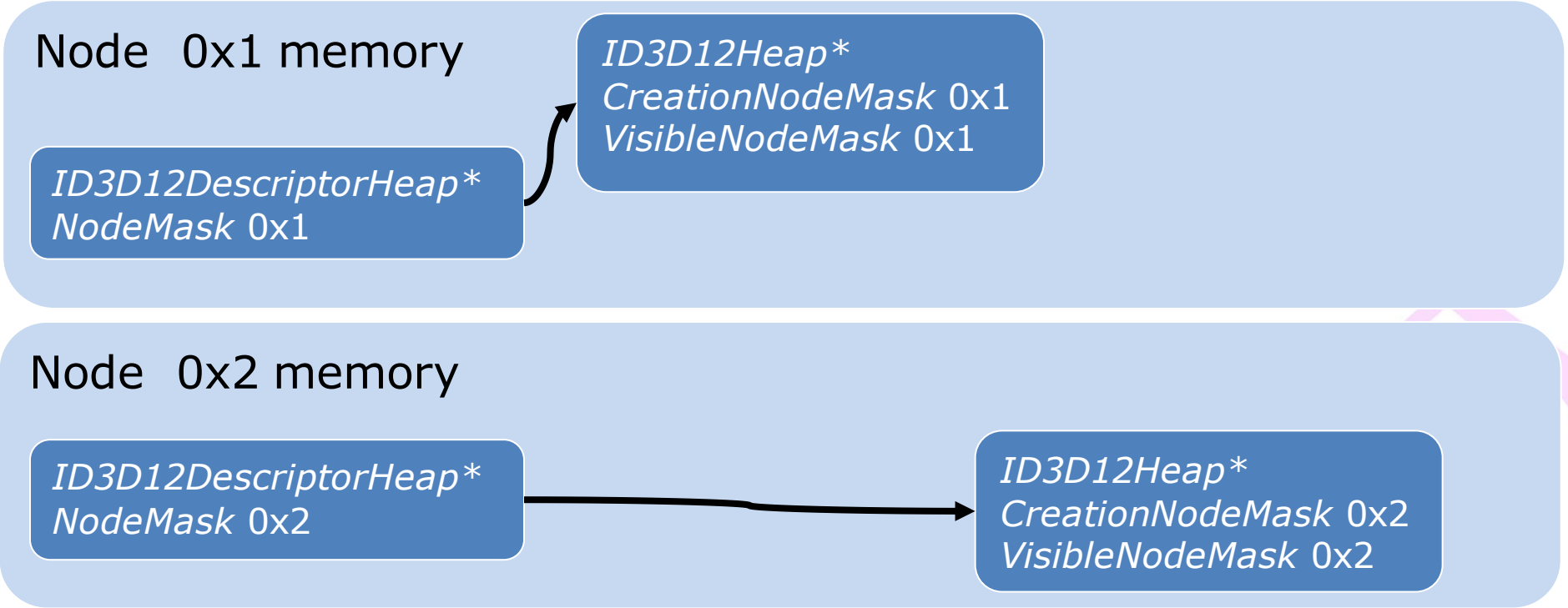
- *ID3D12Device* has tiers for cross node sharing
- Tier 1 supports only cross node copy operations
  - *ID3D12GraphicsCommandList::CopyResource()* etc
- Tier 2 supports cross node SRV/CBV/UAV access

- While SRV/CBV/UAV access may seem convenient, try whether using parallel copy engines would be more efficient

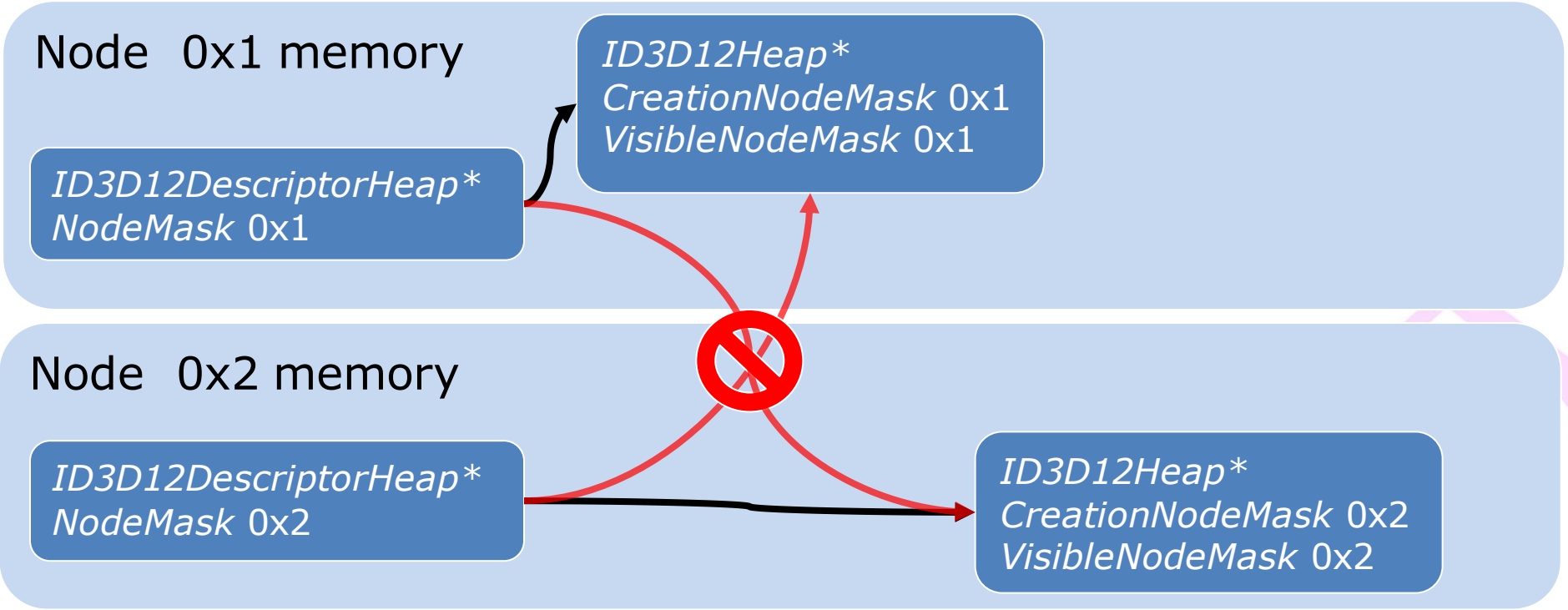
# Resources

- Resources and descriptors need most attention
- Resources/heaps have two separate node masks
  - *CreationNodeMask* is single node mask
  - *VisibleNodeMask* is multi node mask
- Descriptor heap is exclusive to single node

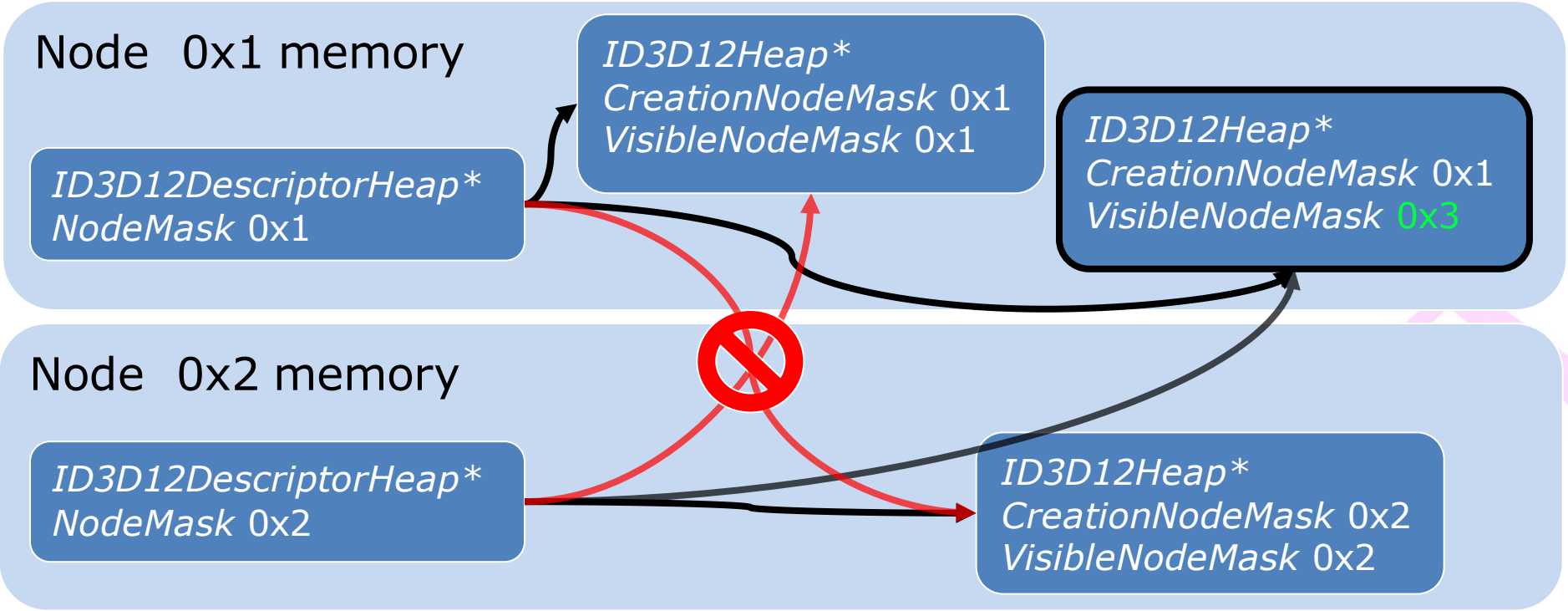
# Resources - Visibility



# Resources - Visibility



# Resources - Visibility



# Resources - Assets

- Upload art assets (vertex data, textures etc.) to nodes that need them
  - It's often convenient to upload your assets to all nodes for easy experimentation
  - AFR needs assets on all nodes
- Create a unique resource for each node, not just one that would be visible to others (with proper *VisibleNodeMask*)

# Resources - AFR Targets

- AFR requires all render targets be duplicated for each node
  - Need robust cycling mechanism
- Again, a unique resource for each node, not one resource visible to all nodes

# AFR Isn’t For Everyone…

- Temporal techniques make AFR difficult
  - Too many inter-frame dependencies can kill the performance
  - Explicit or implicit

# AFR Workflow Problem

## Ideal

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **GPU 1** | Frame 0 | | Frame 2 | | Frame 4 | | Frame 6 | | Frame 8 | |
| **GPU 0** | | *Frame 1* | | *Frame 3* | | Frame 5 | | Frame 7 | | Frame 9 |
| ***Screen*** | (F-2) | (F-1) | F0 | *F1* | F2 | *F3* | *F4* | *F5* | *F6* | *F7* | *F8* |

# AFR Workflow Problem

## Ideal

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **GPU 1** | Frame 0 | | Frame 2 | | Frame 4 | | Frame 6 | | Frame 8 | | |
| **GPU 0** | | *Frame 1* | | *Frame 3* | | Frame 5 | | Frame 7 | | Frame 9 | |
| ***Screen*** | (F-2) | (F-1) | F0 | *F1* | F2 | *F3* | *F4* | *F5* | *F6* | *F7* | *F8* |

## Dependencies between frames

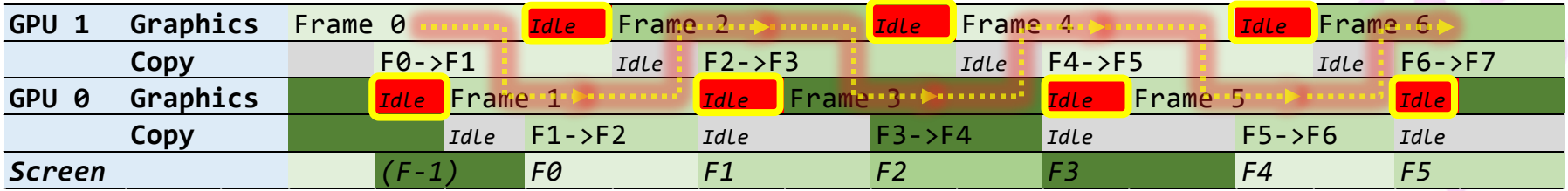
| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **GPU 1** | **Graphics** | Frame 0 | | | *Idle* | Frame 2 | | | *Idle* | Frame 4 | | | *Idle* | Frame 6 | |
| | **Copy** | | F0->F1 | | *Idle* | | F2->F3 | | *Idle* | | F4->F5 | | *Idle* | | F6->F7 |
| **GPU 0** | **Graphics** | | *Idle* | Frame 1 | | *Idle* | | Frame 3 | | *Idle* | | Frame 5 | | *Idle* | |
| | **Copy** | | | *Idle* | F1->F2 | | *Idle* | | F3->F4 | | *Idle* | | F5->F6 | | *Idle* |
| ***Screen*** | | | *(F-1)* | | *F0* | | *F1* | | *F2* | | *F3* | | *F4* | | *F5* |

# AFR Workflow Problem

## Ideal

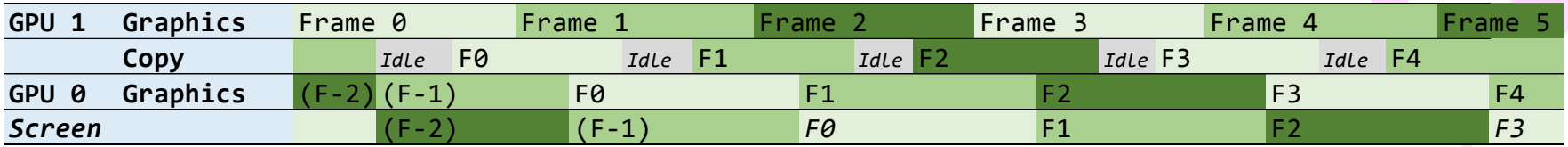
| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| GPU 1 | Frame 0 | | Frame 2 | | Frame 4 | | Frame 6 | | Frame 8 | | |
| GPU 0 | | Frame 1 | | Frame 3 | | Frame 5 | | Frame 7 | | Frame 9 | |
| Screen | (F-2) | (F-1) | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 |

## Dependencies between frames



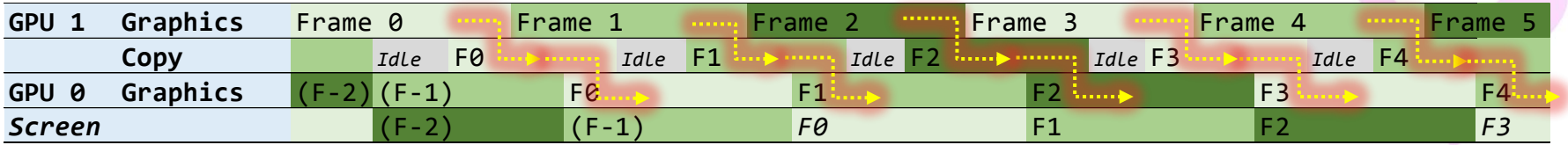
# New Possibility - Frame Pipelining

- Pipeline rendering of frames
  - Begin frame on one GPU
  - Transfer work to next GPU to finish rendering and present
  - The GPUs and copy engines form a pipeline



# New Possibility - Frame Pipelining

- Pipeline rendering of frames
  - Begin frame on one GPU
  - Transfer work to next GPU to finish rendering and present
  - The GPUs and copy engines form a pipeline

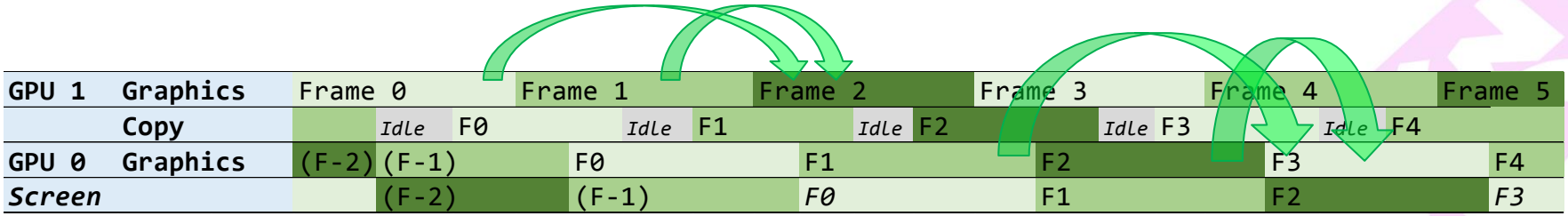


# Pipelining – Simple Dependencies

- No back and forth dependencies between GPUs
  - Helps to minimize waits
  - Easier to do large cross GPU data transfers without reducing frame rate
  - Unless copying takes longer than actual work, it affects only latency, not frame rate

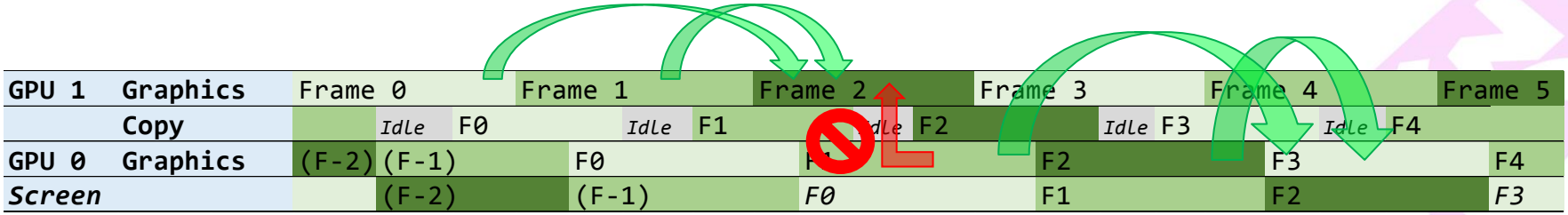
# Pipelining – Temporal techniques

- Temporal techniques allowed without penalties



# Pipelining – Temporal techniques

- Temporal techniques allowed without penalties
- Limitation: GPUs at beginning of pipeline cannot use resources produced further down the pipeline



# Pipelining – Something More

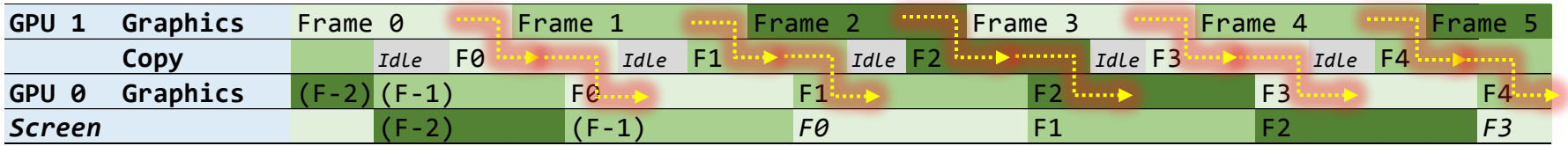
- Instead doing the same faster, do something more
  - GI
  - Ray tracing
  - Physics
  - Etc.

# Pipelining – Workload Distribution

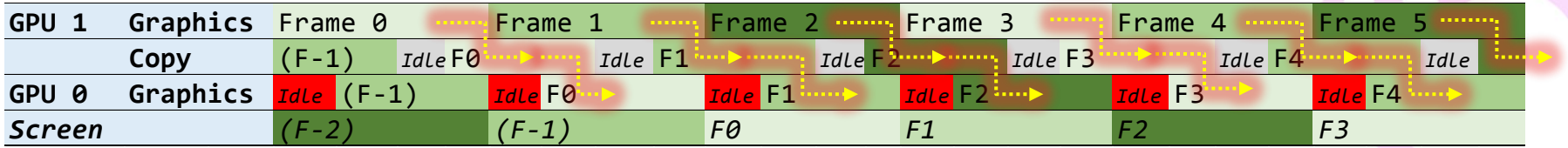
- Needs a good point to split the frame
  - Cross GPU copies are slow regardless of parallel copy engines
    - <8 GB/s on 8xPCIe3, 64 MB consumes at least 8 ms
- Doing some passes on both GPUs instead of transferring the results can be an option

# Frame Pipelining Workflow

## Ideal

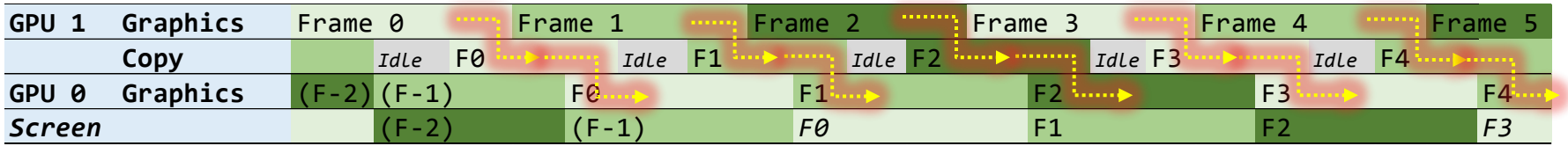



## Unbalanced work

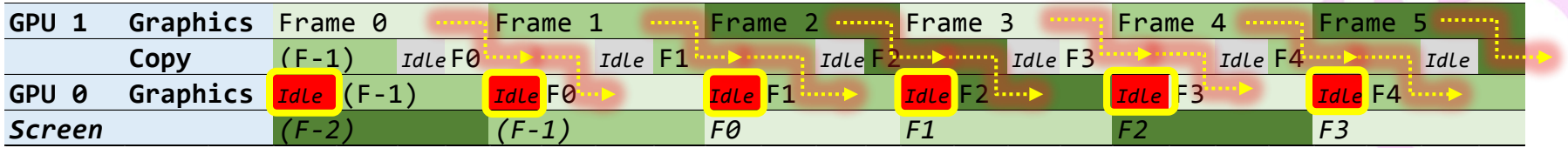


# Frame Pipelining Workflow

## Ideal




## Unbalanced work



# Pipelining – Possible Problems

- Workload balance between GPUs depends also on scene content
  - It's never perfect, but can be reasonable
- Latency can be a problem like in AFR
- Scaling for 3 or 4 GPUs requires separate solutions

# Frame Pipelining Case Study

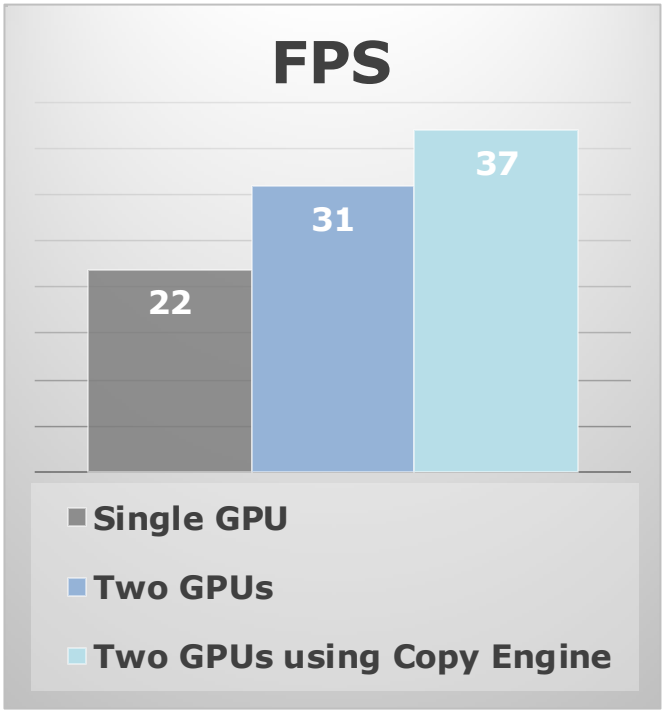
- Microsoft DX12 miniengine
  - Pre-depth
  - SSAO
  - Sun shadow map
  - Primary pass
  - Particles
  - Motion blur
  - Bloom
  - FXAA

# Frame Pipelining Case Study

- As a stress test, 3840x2160 screen and 4k by 4k sun shadow map resolutions were used
- Generated on first GPU:

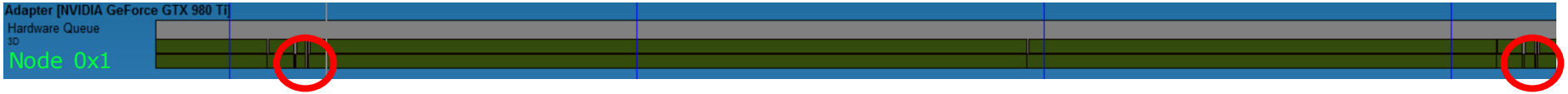
| | | | |
|---|---|---|---|
| Predepth | D32_FLOAT | 31.6 MB | 5.3 ms |
| Linear Depth | R16_FLOAT | 15.8 MB | 2.6 ms |
| SSAO | R8_UNORM | 7.9 MB | 1.3 ms |
| Sun Shadow Map | D16_UNORM | 32 MB | 5.3 ms |
| **Total** | | **87.3 MB** | **14.6 ms** |

# Frame Pipelining Case Study - Performance




# Pipelining Case Study - GPUView

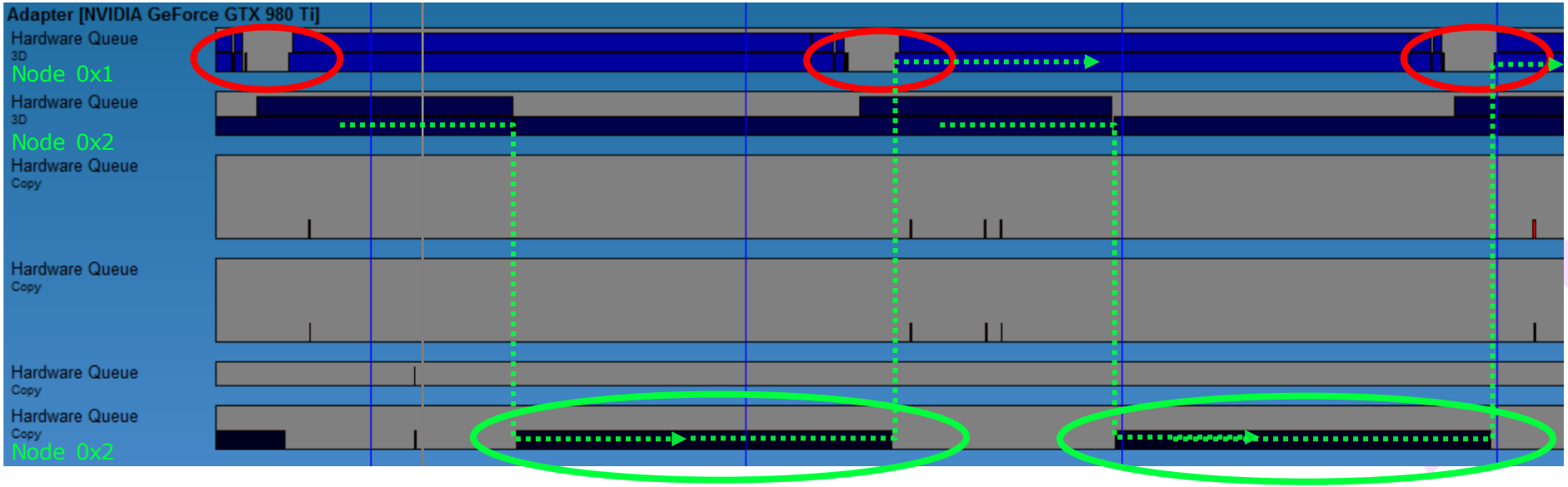
Original single GPU workflow




Two GPUs pipelined without copy engine

# Pipelining Case Study - GPUView

Two GPUs pipelined with copy engine



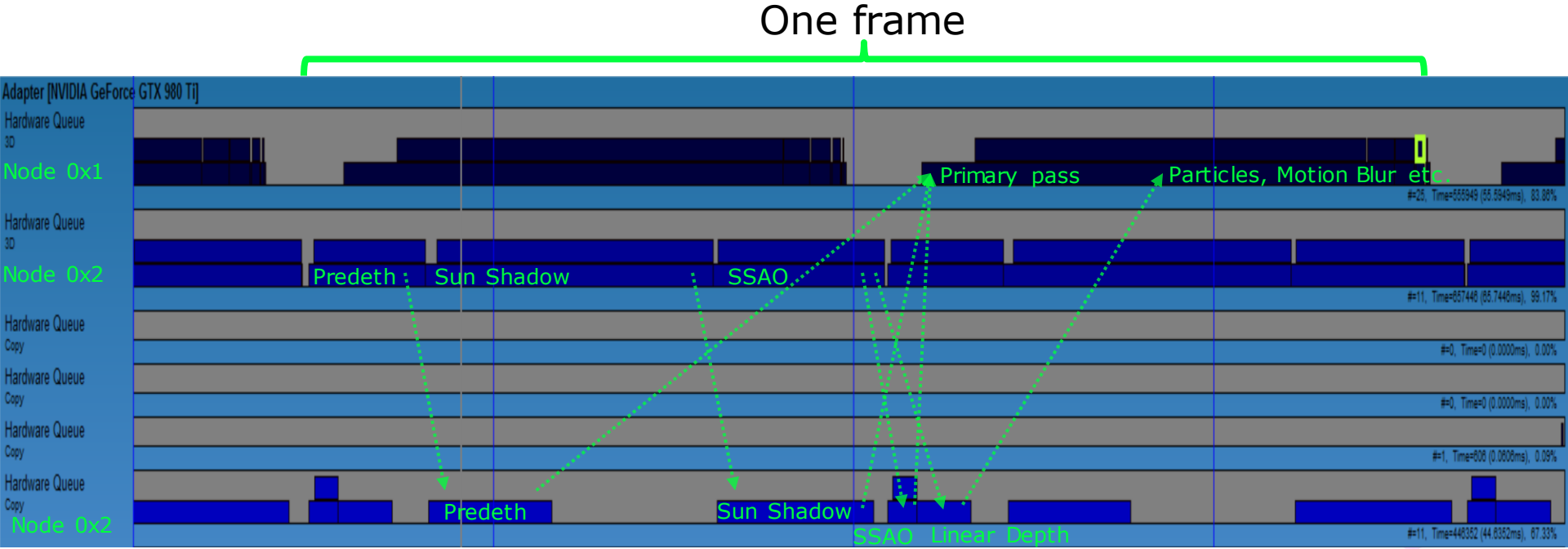
# Frame Pipelining Case Study

- *1.7x framerate from single to dual GPU*
  - Pretty even workload distribution, but it's content dependent
- Cost of copying step would limit frame rate to about 60 fps on 8xPCIe 3.0 system

# Pipelining – Hiding Copy Latency

- Break up copy work into smaller chuncks
  - Overlap with other work for the *same* frame
  - More and smaller command lists
  - *Remember guidelines from the "Practical DirectX 12"*
- In the case study, the ~15 ms extra latency from copies can be almost entirely hidden

# Hiding Copy Latency - GPUView



# Summary

- No more driver magic
- You‘re in control of AFR
- Try pipelining with temporal techniques!
- Remember copy engines!
- You can do anything you want with that extra GPU - Surprise us!

# Questions?

- jsjoholm@nvidia.com