



Enable AI & HPC to be Open, Safe and Accessible to All

# OpenCL Command-buffer Extension: Design & Implementation

EwA Crawford and JAck Frankland



IWOCL – 2022

# Company

Leaders in enabling high-performance software solutions for new AI processing systems

Enabling the toughest processors with tools and middleware based on open standards

Established 2002 in Scotland with ~80 employees

# Products

## Acoran

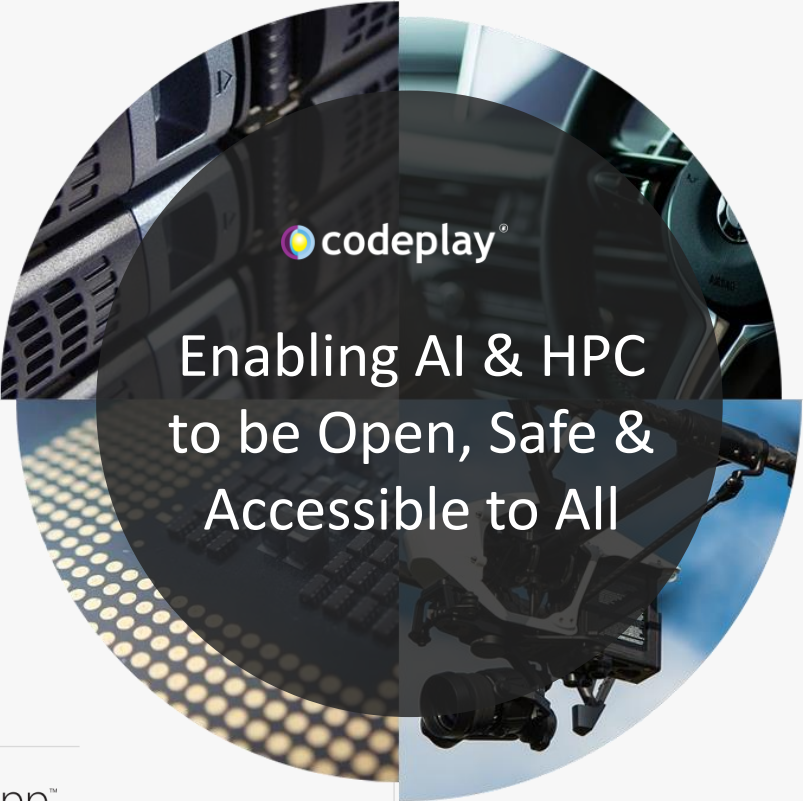
Integrates all the industry standard technologies needed to support a very wide range of AI and HPC

## ComputeAorta™

The heart of Codeplay's compute technology enabling OpenCL™, SPIR-V™, HSA™ and Vulkan™

## ComputeCpp™

C++ platform via the SYCL™ open standard, enabling vision & machine learning e.g. TensorFlow™



# Partners



And many more!

# Markets

High Performance Compute (HPC)  
Automotive ADAS, IoT, Cloud Compute  
Smartphones & Tablets  
Medical & Industrial

Technologies: Artificial Intelligence  
Vision Processing  
Machine Learning  
Big Data Compute

# Agenda

Background

Command-buffer Extension

Design Decisions

Implementation Experience

Next Steps

# Background

# Command List Construction

- OpenCL allows a programmer to offload a sequence of commands to a heterogeneous accelerator.
- The overhead of building a command sequence can be expensive for some hardware, e.g. embedded devices.
- When the same pipeline of commands are repeatedly enqueued this cost is incurred each iteration.

# Pipelined Workflows

- Waiting on the host to construct workload commands also introduces latency until workload can be issued for execution.
- Removing this resubmission latency would keep devices better occupied with work.
- Impacts performance in applications where the same command sequence is used to process different inputs, e.g. computer vision applications operating on images.

# OpenCL API

## Problem

*clEnqueue<Command>* both creates a command and schedules it for execution.

## Solution

Separate these concerns – Distinct API controlling command construction and scheduling commands for execution.

1. Command Construction - Only pay construction cost once.
2. Pipelined Workflow – Low overhead command submit entry-point.

# Proven Abstraction

Vulkan -  
vkCommandBuffer

Intel Level Zero –  
Command Lists

CUDA – CUDA  
Graphs

# Motivating Example

```
cl_mem frame_input, frame_output, tile_input, tile_output;

// Setup buffers, build program, set tile input/output as kernel args

for (size_t f = 0; f < num_frames; f++) {
    clEnqueueWriteBuffer(command_queue, frame_input, CL_TRUE, ...);

    for (size_t t = 0; t < num_tiles; t++) {
        clEnqueueCopyBuffer(command_queue, frame_input, tile_input,
                             frame_offset, 0, ...);
        clEnqueueNDRangeKernel(command_queue, kernel, ...);
        clEnqueueCopyBuffer(command_queue, tile_output, frame_output,
                             0, frame_offset ...);
    }
    clEnqueueReadBuffer(command_queue, frame_output, CL_TRUE, ...);
}
```

Code snippet from an application doing image processing with tiled memory.

Repeated sequence of commands - we want to avoid having to duplicate creating these commands each iteration.

# Command-buffer Extension

# cl\_khr\_command\_buffer

- cl\_khr\_command\_buffer extension defines an alternative API mechanism that separates command construction from execution.
- Created from contributions by many OpenCL working group members.

## 46. Command Buffers (Provisional)

This extension adds the ability to record and replay buffers of OpenCL commands.

### 46.1. General Information

#### 46.1.1. Name Strings

cl\_khr\_command\_buffer

#### 46.1.2. Version History

Date	Version	Description
2021-11-10	0.9.0	First assigned version (provisional).

[https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL\\_Ext.html#cl\\_khr\\_command\\_buffer](https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_Ext.html#cl_khr_command_buffer)

# Command-buffer Lifecycle

- Create command-buffer targeting a device.
- Record commands to command-buffer using new entry-points.
- Finalize command-buffer, at which point no more commands can be recorded.
- Submit command-buffer one or more times asynchronously.

Device queries available to report usage specifics.

# Command-buffer Example

```
cl_mem frame_input, frame_output, tile_input, tile_output;

// Setup buffers, build program, set tile input/output as kernel args

cl_command_buffer_khr command_buffer =
    clCreateCommandBufferKHR(1, &command_queue, nullptr, nullptr);

for (size_t t = 0; t < num_tiles; t++) {
    clCommandCopyBufferKHR(command_buffer, nullptr, frame_input,
                           tile_input, frame_offset, 0, ...);
    clCommandNDRangeKernelKHR(command_buffer, nullptr, nullptr, kernel, ...);
    clCommandCopyBufferKHR(command_buffer, nullptr, tile_output,
                           frame_output, 0, frame_offset, ...);
}

clFinalizeCommandBufferKHR(command_buffer);

for (size_t f = 0; f < num_frames; f++) {
    clEnqueueWriteBuffer(command_queue, frame_input, CL_TRUE, ...);
    clEnqueueCommandBufferKHR(0, nullptr, command_buffer, 0, nullptr,
                              nullptr);
    clEnqueueReadBuffer(command_queue, frame_output, CL_TRUE, ...);
}
```

Create the command-buffer

Record commands

End recording of commands

Execute commands for each frame

# Command-buffer Example

```
cl_mem frame_input, frame_output, tile_input, tile_output;

// Setup buffers, build program, set tile input/output as kernel args

cl_command_buffer_khr command_buffer =
    clCreateCommandBufferKHR(1, &command_queue, nullptr, nullptr);

for (size_t t = 0; t < num_tiles; t++) {
    clCommandCopyBufferKHR(command_buffer, nullptr, frame_input,
                           tile_input, frame_offset, 0, ...);
    clCommandNDRangeKernelKHR(command_buffer, nullptr, nullptr, kernel, ...);
    clCommandCopyBufferKHR(command_buffer, nullptr, tile_output,
                           frame_output, 0, frame_offset, ...);
}

clFinalizeCommandBufferKHR(command_buffer);

for (size_t f = 0; f < num_frames; f++) {
    clEnqueueWriteBuffer(command_queue, frame_input, CL_TRUE, ...);
    clEnqueueCommandBufferKHR(0, nullptr, command_buffer, 0, nullptr,
                              nullptr);
    clEnqueueReadBuffer(command_queue, frame_output, CL_TRUE, ...);
}
```

Create the command-buffer

```
cl_command_buffer_khr clCreateCommandBufferKHR(
    cl_uint num_queues,
    const cl_command_queue* queues,
    const cl_command_buffer_properties_khr* properties,
    cl_int* errcode_ret);
```

# Command-buffer Example

```
cl_mem frame_input, frame_output, tile_input, tile_output;

// Setup buffers, build program, set tile input/output as kernel args

cl_command_buffer_khr command_buffer =
    clCreateCommandBufferKHR(1, &command_queue, nullptr, nullptr);

for (size_t t = 0; t < num_tiles; t++) {
    clCommandCopyBufferKHR(command_buffer, nullptr, frame_input,
                           tile_input, frame_offset, 0, ...);
    clCommandNDRangeKernelKHR(command_buffer, nullptr, nullptr, kernel, ...);
    clCommandCopyBufferKHR(command_buffer, nullptr, tile_output,
                           frame_output, 0, frame_offset, ...);
}

clFinalizeCommandBufferKHR(command_buffer);

for (size_t f = 0; f < num_frames; f++) {
    clEnqueueWriteBuffer(command_queue, frame_input, CL_TRUE, ...);
    clEnqueueCommandBufferKHR(0, nullptr, command_buffer, 0, nullptr,
                              nullptr);
    clEnqueueReadBuffer(command_queue, frame_output, CL_TRUE, ...);
}
```

Create the command-buffer

```
cl_command_buffer_khr clCreateCommandBufferKHR(
    cl_uint num_queues,
    const cl_command_queue* queues,
    const cl_command_buffer_properties_khr* properties,
    cl_int* errcode_ret);
```

Only a single queue permitted for the moment

# Command-buffer Example

```
cl_mem frame_input, frame_output, tile_input, tile_output;

// Setup buffers, build program, set tile input/output as kernel args

cl_command_buffer_khr command_buffer =
    clCreateCommandBufferKHR(1, &command_queue, nullptr, nullptr);

for (size_t t = 0; t < num_tiles; t++) {
    clCommandCopyBufferKHR(command_buffer, nullptr, frame_input,
                           tile_input, frame_offset, 0, ...);
    clCommandNDRangeKernelKHR(command_buffer, nullptr, nullptr, kernel, ...);
    clCommandCopyBufferKHR(command_buffer, nullptr, tile_output,
                           frame_output, 0, frame_offset, ...);
}

clFinalizeCommandBufferKHR(command_buffer);

for (size_t f = 0; f < num_frames; f++) {
    clEnqueueWriteBuffer(command_queue, frame_input, CL_TRUE, ...);
    clEnqueueCommandBufferKHR(0, nullptr, command_buffer, 0, nullptr,
                              nullptr);
    clEnqueueReadBuffer(command_queue, frame_output, CL_TRUE, ...);
}
```

## Record commands

```
cl_int clCommandNDRangeKernelKHR(
    cl_command_buffer_khr command_buffer,
    cl_command_queue command_queue,
    const cl_ndrange_kernel_command_properties_khr* properties,
    cl_kernel kernel,
    cl_uint work_dim,
    const size_t* global_work_offset,
    const size_t* global_work_size,
    const size_t* local_work_size,
    cl_uint num_sync_points_in_wait_list,
    const cl_sync_point_khr* sync_point_wait_list,
    cl_sync_point_khr* sync_point,
    cl_mutable_command_khr* mutable_handle);
```

# Command-buffer Example

```
cl_mem frame_input, frame_output, tile_input, tile_output;

// Setup buffers, build program, set tile input/output as kernel args

cl_command_buffer_khr command_buffer =
    clCreateCommandBufferKHR(1, &command_queue, nullptr, nullptr);

for (size_t t = 0; t < num_tiles; t++) {
    clCommandCopyBufferKHR(command_buffer, nullptr, frame_input,
                           tile_input, frame_offset, 0, ...);
    clCommandNDRangeKernelKHR(command_buffer, nullptr, nullptr, kernel, ...);
    clCommandCopyBufferKHR(command_buffer, nullptr, tile_output,
                           frame_output, 0, frame_offset, ...);
}

clFinalizeCommandBufferKHR(command_buffer);

for (size_t f = 0; f < num_frames; f++) {
    clEnqueueWriteBuffer(command_queue, frame_input, CL_TRUE, ...);
    clEnqueueCommandBufferKHR(0, nullptr, command_buffer, 0, nullptr,
                              nullptr);
    clEnqueueReadBuffer(command_queue, frame_output, CL_TRUE, ...);
}
```

## Record commands

```
cl_int clCommandNDRangeKernelKHR(
    cl_command_buffer_khr command_buffer,
    cl_command_queue command_queue,
    const cl_ndrange_kernel_command_properties_khr* properties,
    cl_kernel kernel,
    cl_uint work_dim,
    const size_t* global_work_offset,
    const size_t* global_work_size,
    const size_t* local_work_size,
    cl_uint num_sync_points_in_wait_list,
    const cl_sync_point_khr* sync_point_wait_list,
    cl_sync_point_khr* sync_point,
    cl_mutable_command_khr* mutable_handle);
```

- Properties parameter for use in later extensions.
- mutable\_handle for future functionality to change kernel command configuration.
- Newly defined sync-points rather than events.

# Command-buffer Example

```
cl_mem frame_input, frame_output, tile_input, tile_output;

// Setup buffers, build program, set tile input/output as kernel args

cl_command_buffer_khr command_buffer =
    clCreateCommandBufferKHR(1, &command_queue, nullptr, nullptr);

for (size_t t = 0; t < num_tiles; t++) {
    clCommandCopyBufferKHR(command_buffer, nullptr, frame_input,
                           tile_input, frame_offset, 0, ...);
    clCommandNDRangeKernelKHR(command_buffer, nullptr, nullptr, kernel, ...);
    clCommandCopyBufferKHR(command_buffer, nullptr, tile_output,
                           frame_output, 0, frame_offset, ...);
}

clFinalizeCommandBufferKHR(command_buffer);

for (size_t f = 0; f < num_frames; f++) {
    clEnqueueWriteBuffer(command_queue, frame_input, CL_TRUE, ...);
    clEnqueueCommandBufferKHR(0, nullptr, command_buffer, 0, nullptr,
                              nullptr);
    clEnqueueReadBuffer(command_queue, frame_output, CL_TRUE, ...);
}
```

End recording of commands

```
cl_int clFinalizeCommandBufferKHR(cl_command_buffer_khr command_buffer);
```

# Command-buffer Example

```
cl_mem frame_input, frame_output, tile_input, tile_output;

// Setup buffers, build program, set tile input/output as kernel args

cl_command_buffer_khr command_buffer =
    clCreateCommandBufferKHR(1, &command_queue, nullptr, nullptr);

for (size_t t = 0; t < num_tiles; t++) {
    clCommandCopyBufferKHR(command_buffer, nullptr, frame_input,
                           tile_input, frame_offset, 0, ...);
    clCommandNDRangeKernelKHR(command_buffer, nullptr, nullptr, kernel, ...);
    clCommandCopyBufferKHR(command_buffer, nullptr, tile_output,
                           frame_output, 0, frame_offset, ...);
}

clFinalizeCommandBufferKHR(command_buffer);

for (size_t f = 0; f < num_frames; f++) {
    clEnqueueWriteBuffer(command_queue, frame_input, CL_TRUE, ...);
    clEnqueueCommandBufferKHR(0, nullptr, command_buffer, 0, nullptr,
                             nullptr);
    clEnqueueReadBuffer(command_queue, frame_output, CL_TRUE, ...);
}
```

## End recording of commands

```
cl_int clFinalizeCommandBufferKHR(cl_command_buffer_khr command_buffer);
```

- Provides the runtime with optimization opportunities based on knowledge of command dependencies.
- Explicit entry-point gives users control of when to incur any synchronous latency.

# Command-buffer Example

```
cl_mem frame_input, frame_output, tile_input, tile_output;

// Setup buffers, build program, set tile input/output as kernel args

cl_command_buffer_khr command_buffer =
    clCreateCommandBufferKHR(1, &command_queue, nullptr, nullptr);

for (size_t t = 0; t < num_tiles; t++) {
    clCommandCopyBufferKHR(command_buffer, nullptr, frame_input,
                           tile_input, frame_offset, 0, ...);
    clCommandNDRangeKernelKHR(command_buffer, nullptr, nullptr, kernel, ...);
    clCommandCopyBufferKHR(command_buffer, nullptr, tile_output,
                           frame_output, 0, frame_offset, ...);
}

clFinalizeCommandBufferKHR(command_buffer);

for (size_t f = 0; f < num_frames; f++) {
    clEnqueueWriteBuffer(command_queue, frame_input, CL_TRUE, ...);
    clEnqueueCommandBufferKHR(0, nullptr, command_buffer, 0, nullptr,
                              nullptr);
    clEnqueueReadBuffer(command_queue, frame_output, CL_TRUE, ...);
}
```

Execute commands for each frame

```
cl_int clEnqueueCommandBufferKHR(
    cl_uint num_queues,
    cl_command_queue* queues,
    cl_command_buffer_khr command_buffer,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

# Command-buffer Example

```
cl_mem frame_input, frame_output, tile_input, tile_output;

// Setup buffers, build program, set tile input/output as kernel args

cl_command_buffer_khr command_buffer =
    clCreateCommandBufferKHR(1, &command_queue, nullptr, nullptr);

for (size_t t = 0; t < num_tiles; t++) {
    clCommandCopyBufferKHR(command_buffer, nullptr, frame_input,
                           tile_input, frame_offset, 0, ...);
    clCommandNDRangeKernelKHR(command_buffer, nullptr, nullptr, kernel, ...);
    clCommandCopyBufferKHR(command_buffer, nullptr, tile_output,
                           frame_output, 0, frame_offset, ...);
}

clFinalizeCommandBufferKHR(command_buffer);

for (size_t f = 0; f < num_frames; f++) {
    clEnqueueWriteBuffer(command_queue, frame_input, CL_TRUE, ...);
    clEnqueueCommandBufferKHR(0, nullptr, command_buffer, 0, nullptr,
                              nullptr);
    clEnqueueReadBuffer(command_queue, frame_output, CL_TRUE, ...);
}
```

Execute commands for each frame

```
cl_int clEnqueueCommandBufferKHR(
    cl_uint num_queues,
    cl_command_queue* queues,
    cl_command_buffer_khr command_buffer,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

Queue must be compatible with queue used to create command-buffer, i.e same device and properties.

# Design Decisions

# API Design Alternative

```
cl_command_buffer_khr command_buffer;
clBeginQueueRecording(command_queue, command_buffer);

for (size_t t = 0; t < num_tiles; t++) {
    clEnqueueCopyBuffer(command_queue, frame_input, tile_input,
                        frame_offset, 0, ...);
    clEnqueueNDRangeKernel(command_queue, kernel, ...);
    clEnqueueCopyBuffer(command_queue, tile_output, frame_output,
                        0, frame_offset ...);
}

clEndQueueRecording(command_queue, command_buffer);

for (size_t f = 0; f < num_frames; f++) {
    clEnqueueWriteBuffer(command_queue, frame_input, CL_TRUE, ...);
    clEnqueueCommandBufferKHR(0, nullptr, command_buffer,
                             0, nullptr, nullptr);
    clEnqueueReadBuffer(command_queue, frame_output,
                        CL_TRUE, ...);
}
```

Alternative that uses existing command-queue entry-points for recording

Introduces state to command-queue – where queue can be put into a “recording” state

Advantage: Easier for users to update existing applications to use extension

# Implications of Stateful Design

## Maintainability

- If a new command is added to core OpenCL spec it can be immediately used by extension if desired.
- However, if we don't want to allow the new command then still need to update extension spec with error wording forbidding it.
- Reverse of the maintenance situation of current design, new commands we'd like to introduce would need added (possibly as a layered extension) but free to ignore new commands not introduced.

## User Readability

- Smaller API footprint & less duplication of entry-points, so easier to use in existing applications.
- However, may be harder for users to reason about code as they have to keep mental note of command-queue state

# New Entry-points

## Constrain Scope

- Haven't allowed commands inside a command-buffer to interaction with host
  - No read/write/map buffer commands
  - No `cl_event` which allows host callbacks and host synchronization
  - Introduce `cl_sync_point_khr` for synchronization within command-buffer **only**

## Control

- Able to add extra parameters to new command recording entry-points
  - Properties parameter to kernel commands
  - Mutable handle parameter to allow every command in a command-buffer to be referenced with a handle, rather than having to get an application writer to remember indices

# Vulkan Comparison

- Vulkan distinguishes between **primary** and **secondary** command-buffers
  - Primary command-buffers are submitted to queues.
  - Secondary command-buffers can only be executed from another command-buffer.
  - OpenCL command-buffers are all **primary** command-buffers.
- vkResetCommandBuffer
  - Resets a command-buffer to initial state to avoid the overhead of frequent creation and destruction
  - No equivalent in OpenCL extension – could be added later if cost of creation/destruction is found to be prohibitive to performance.

# Vulkan Comparison

VkCommandBufferUsageFlagBits	Semantics	In OpenCL command-buffers
VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT	Indicates that the command-buffer may only be submitted once	Not represented
VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT	Only relevant for render passes and secondary command-buffers	Not represented
VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT	Indicates that the command-buffer can submitted while a current submission is in flight	Available <ul style="list-style-type: none"><li>• Optionally supported by devices</li><li>• Set on command-buffer creation as a property</li><li>• Discussion ComputeAorta implementation in later slides.</li></ul>

# Layering

- Command-buffer extension unlocks possibility of extending functionality in different directions.
- Rather than combine functionality into a single extension with lots of optional capabilities, decided to layer the extension functionality across multiple extensions with `cl_khr_command_buffer` as the base.
  - Quicker release of base extension, allowing for earlier feedback from community & implementors
  - Simpler base extension reduces effort to implement minimum functionality.
  - Standalone extension documentation is more readable.
- Layered extensions being developed and with target provisional release in 2022.

# Layering

## Mutable Kernel Commands

- Kernel commands in a command-buffer may be modified between command-buffer enqueues.
- Being able to modify commands is the rationale behind the unused mutable-handle output parameter specified in command recording entry-points.

## Multi-device command-buffer

- Individual commands in a command-buffer can be recorded to queues targeting different devices
- Rationale behind unused queue parameter in command recording entry-points.

# Implementation Experience

# ComputeAorta

- Codeplay's toolkit for building heterogeneous compute runtimes
- Amongst other components consists of an OpenCL implementation built on top of Codeplay's proprietary ComputeMux API
- For more details on ComputeAorta see 2020 IWOCCL talk



# ComputeMux

- ComputeMux is Codeplay's bare metal compute API
- ComputeMux already has concept of `mux_command_buffer_s` object
- Commands within `mux_command_buffer_s` execute in-order

# mux\_command\_buffer\_s

## OpenCL Command Buffers

clEnqueueReadBuffer  
clEnqueueCopyBuffer  
clEnqueueWriteBuffer  
clEnqueueNDRangeKernel

...

...

...

## ComputeMux

muxCommandReadBuffer  
muxCommandCopyBuffer  
muxCommandWriteBuffer  
muxCommandNDRange

...

...

...

# mux\_command\_buffer\_s

## OpenCL Command Buffers

clCommandCopyBufferKHR  
clCommandCopyBufferRectKHR  
clCommandCopyBufferToImage  
clCommandCopyImageKHR  
clCommandCopyImageToBufferKHR  
clCommandFillBufferKHR  
clCommandFillImageKHR  
clCommandNDRangeKernelKHR

## ComputeMux

muxCommandCopyBuffer  
muxCommandCopyBufferRegions  
muxCommandCopyBufferToImage  
muxCommandCopyImage  
muxCommandCopyImageToBuffer  
muxCommandFillBuffer  
muxCommandFillImage  
muxCommandNDRange

# Command Batching

- mux\_command\_buffer\_s already go some way to reducing overhead of building command streams in vanilla OpenCL
- As regular OpenCL commands are enqueued to a cl\_command\_queue they are "batched" into mux\_command\_buffer\_s objects according to certain constraints
  - "pending dispatch"
- Command batches are then dispatched when a blocking event or flush occurs in OpenCL, avoiding the cost of building a command stream for every individual command

# Batching Algorithm

Wait events associated with a single pending dispatch

Push command to the associated command-buffer

Wait events associated with multiple pending dispatches

Get an unused command-buffer

No wait events or wait events with no associated pending dispatches (already dispatched)

Get an unused command-buffer

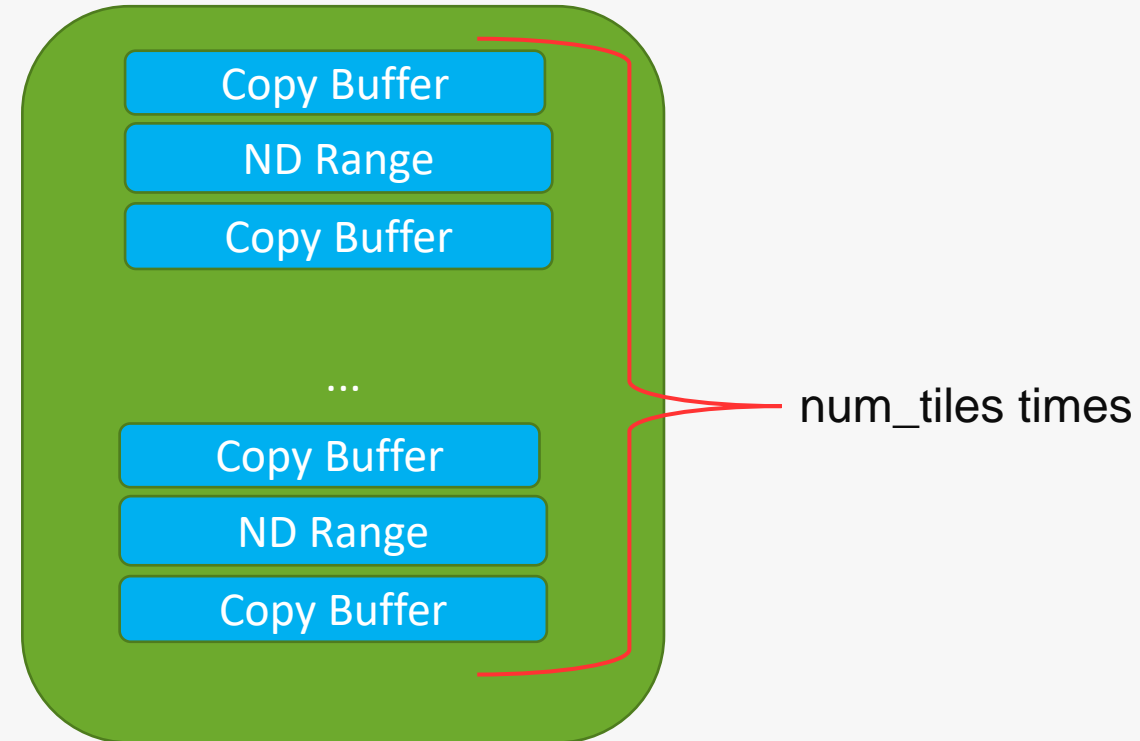
# Batching Algorithm

## cl\_command\_queue responsibilities

- Creating mux\_command\_buffer\_s objects
- Destroying mux\_command\_buffer\_s objects
- May reset command buffers via muxResetCommandBuffer and put them in a cache to avoid wasted overhead of resource allocation/deallocation
- Creating/destroying/caching and signalling mux\_semaphore\_s objects used to express dependencies between mux\_command\_buffer\_s objects
- Signalling and waiting on OpenCL cl\_events

# Problem: Batching cl\_command\_buffer\_khr

```
for (size_t t = 0; t < num_tiles; t++) {  
    clCommandCopyBufferKHR(...);  
    clCommandNDRangeKernelKHR(..);  
    clCommandCopyBufferKHR(...);  
}  
  
clFinalizeCommandBufferKHR(...);
```



# Problem: Batching cl\_command\_buffer\_khr

```
for (size_t f = 0; f < num_frames; f++) {  
    clEnqueueWriteBuffer(...);  
    clEnqueueCommandBufferKHR(...);  
    clEnqueueReadBuffer(...);  
}
```

- Batching command appends subsequent regular commands to command-buffer
- cl\_command\_queue will reset or destroy command-buffer once it has finished executing

Pending Dispatches

# Problem: Batching cl\_command\_buffer\_khr

```
for (size_t f = 0; f < num_frames; f++) {  
    clEnqueueWriteBuffer(...);  
    clEnqueueCommandBufferKHR(...);  
    clEnqueueReadBuffer(...);  
}
```

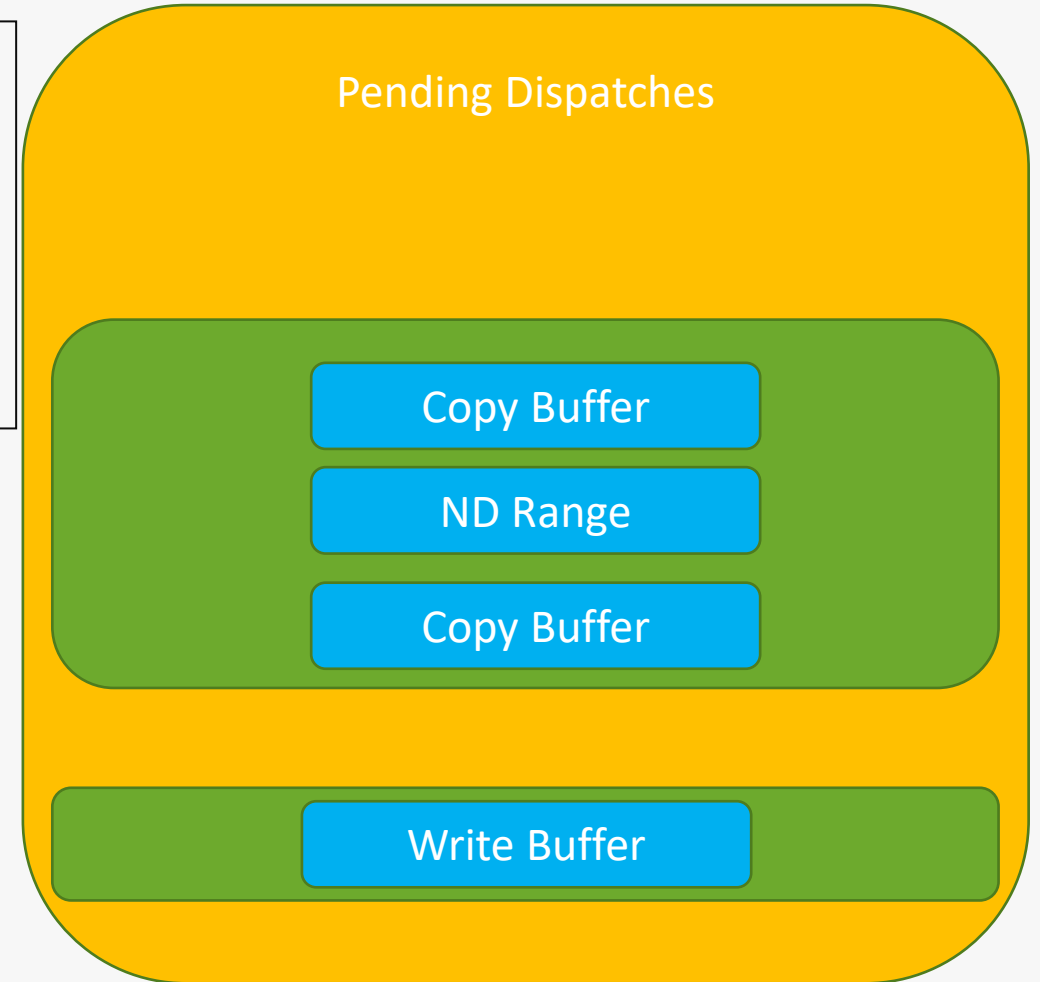
- Batching command appends subsequent regular commands to command-buffer
- cl\_command\_queue will reset or destroy command-buffer once it has finished executing



# Problem: Batching cl\_command\_buffer\_khr

```
for (size_t f = 0; f < num_frames; f++) {  
    clEnqueueWriteBuffer(...);  
    clEnqueueCommandBufferKHR(...);  
    clEnqueueReadBuffer(...);  
}
```

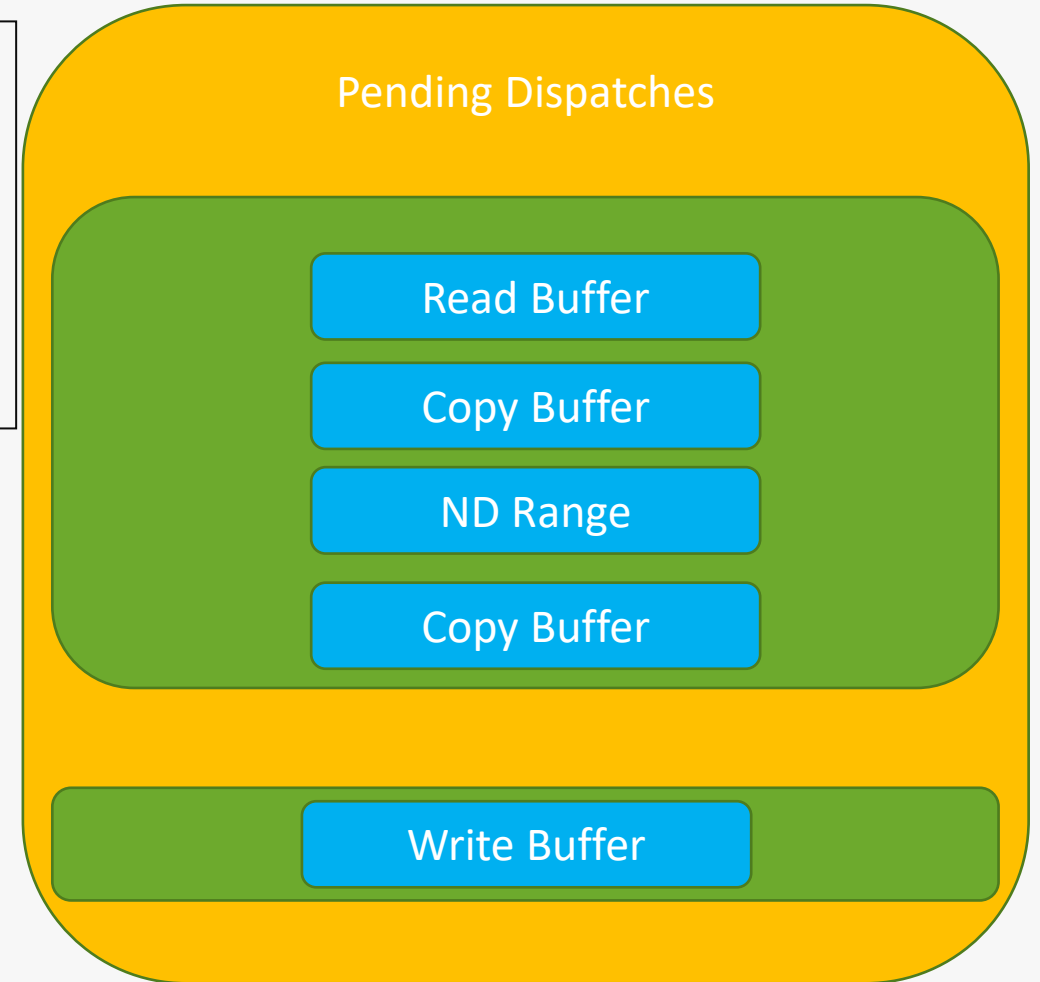
- Batching command appends subsequent regular commands to command-buffer
- cl\_command\_queue will reset or destroy command-buffer once it has finished executing



# Problem: Batching cl\_command\_buffer\_khr

```
for (size_t f = 0; f < num_frames; f++) {  
    clEnqueueWriteBuffer(...);  
    clEnqueueCommandBufferKHR(...);  
    clEnqueueReadBuffer(...);  
}
```

- Batching command appends subsequent regular commands to command-buffer
- cl\_command\_queue will reset or destroy command-buffer once it has finished executing

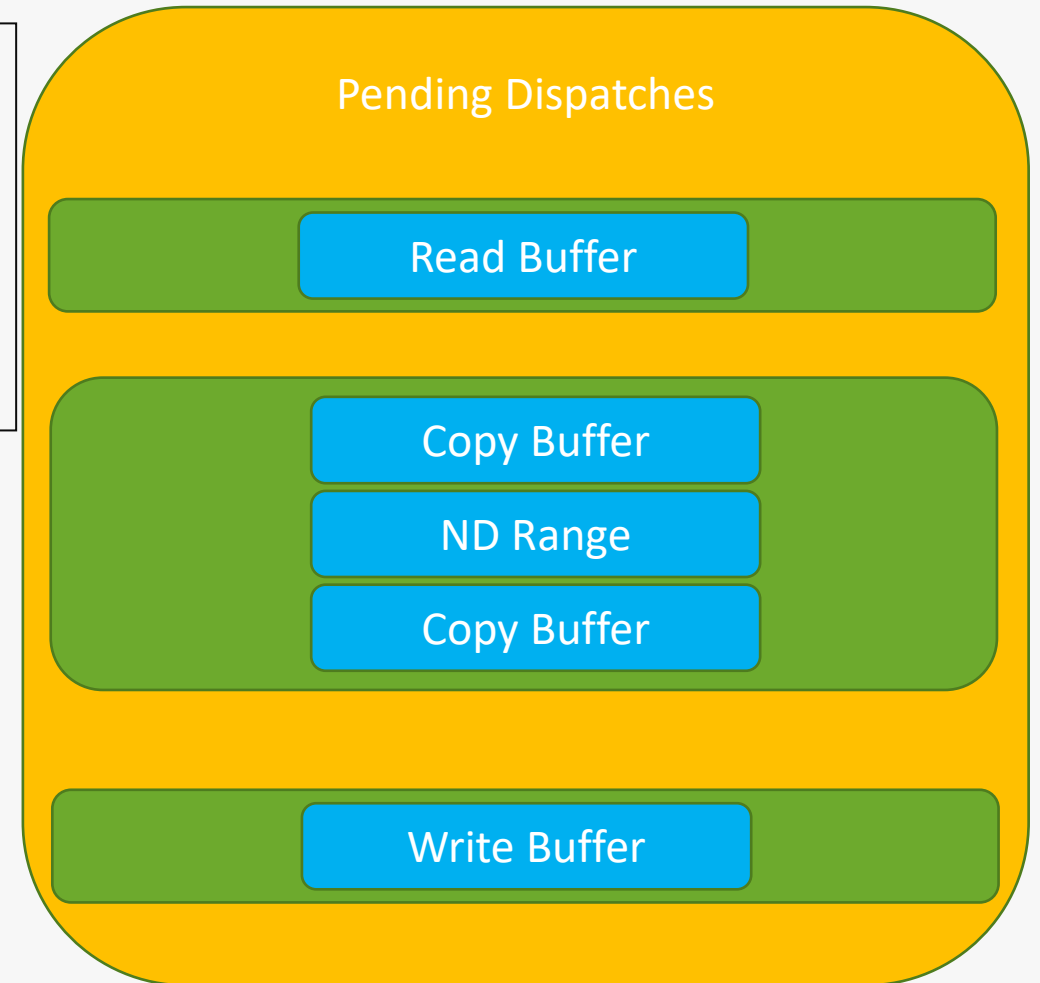


# Problem: Batching cl\_command\_buffer\_khr

```
for (size_t f = 0; f < num_frames; f++) {  
    clEnqueueWriteBuffer(...);  
    clEnqueueCommandBufferKHR(...);  
    clEnqueueReadBuffer(...);  
}
```

## “User Command-Buffer”

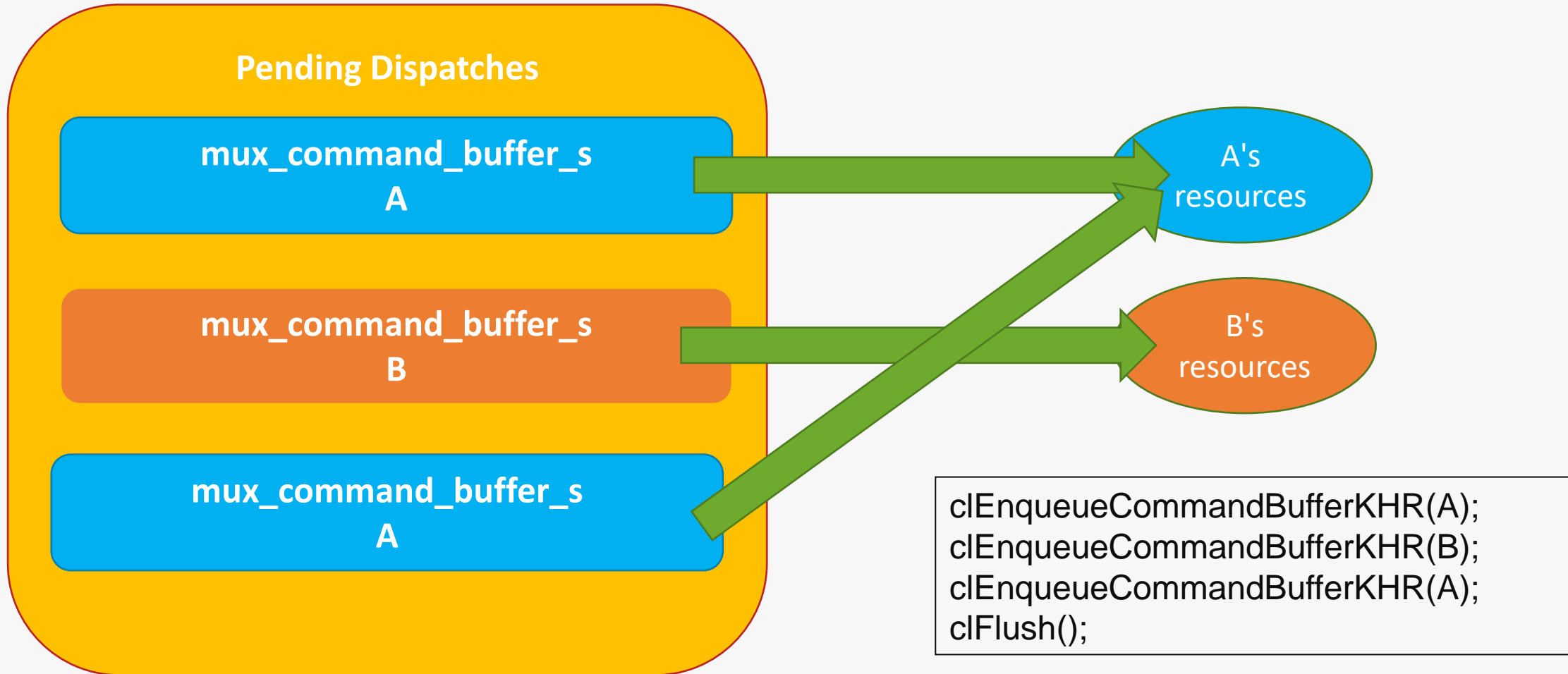
- Can't be appended to – will always cause subsequent regular commands to get a new `mux_command_buffer_s`
- Won't be reset or destroyed by `cl_command_queue`
- Will outlive the `cl_command_queue`



# Problem: Simultaneous Use

- mux\_command\_buffer\_s don't support simultaneous use
- Not possible to have more than one mux\_command\_buffer\_s in flight at a time before cl\_khr\_command\_buffer use case
- Resources used by mux\_command\_buffer\_s means enqueueing it more than once corrupts the queue

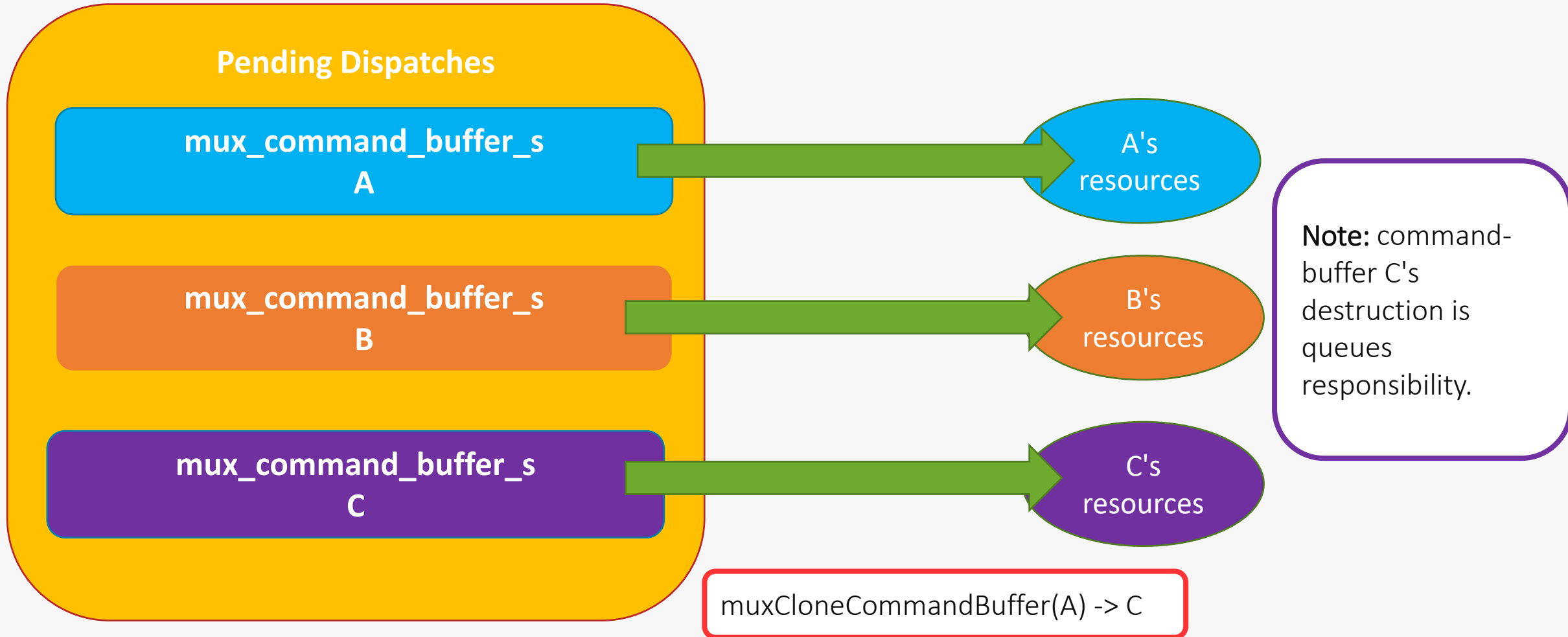
# Problem: Simultaneous Use



# Solution: Simultaneous Use

- `CL_COMMAND_BUFFER_CAPABILITY_SIMULTANEOUS_USE_KHR` introduced to make this optional so vendors can avoid situation altogether
- Introduced `muxCloneCommandBuffer` entry point
  - Copies of a command buffer, returning an identical but independent `mux_command_buffer_s`.
  - Allows user to create command buffers with `CL_COMMAND_BUFFER_SIMULTANEOUS_USE_KHR`
  - If set will result in call to `muxCloneCommandBuffer` at enqueue time when pending count exceeds 1

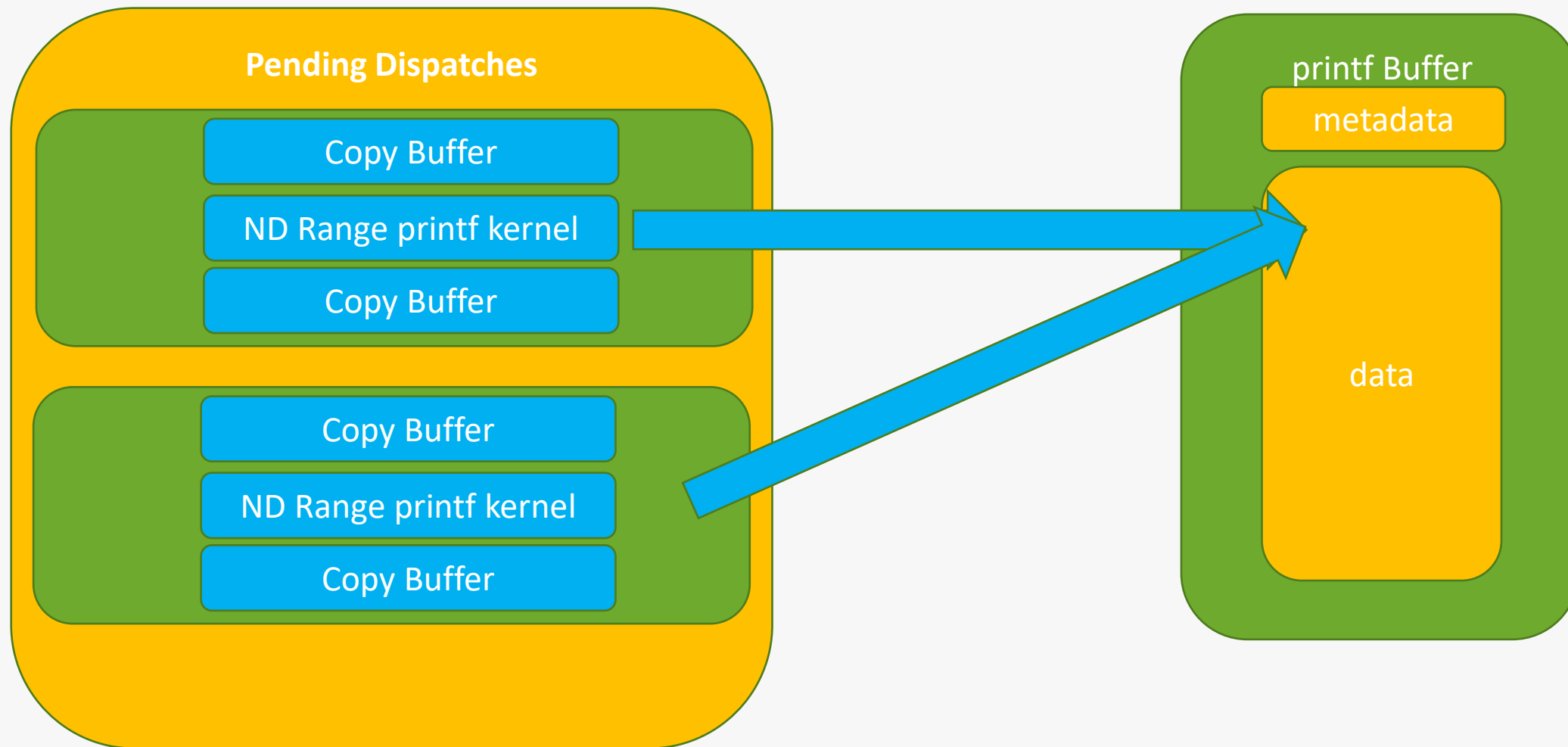
# Solution: Simultaneous Use



# Problem: printf

- Kernel containing printf gets an implicit buffer added to it, printf writes into this buffer
- When kernel is enqueued an implicit callback is added to read the buffer and printf its content on host
- If implementation supports simultaneous-use, printf call may clobber one another

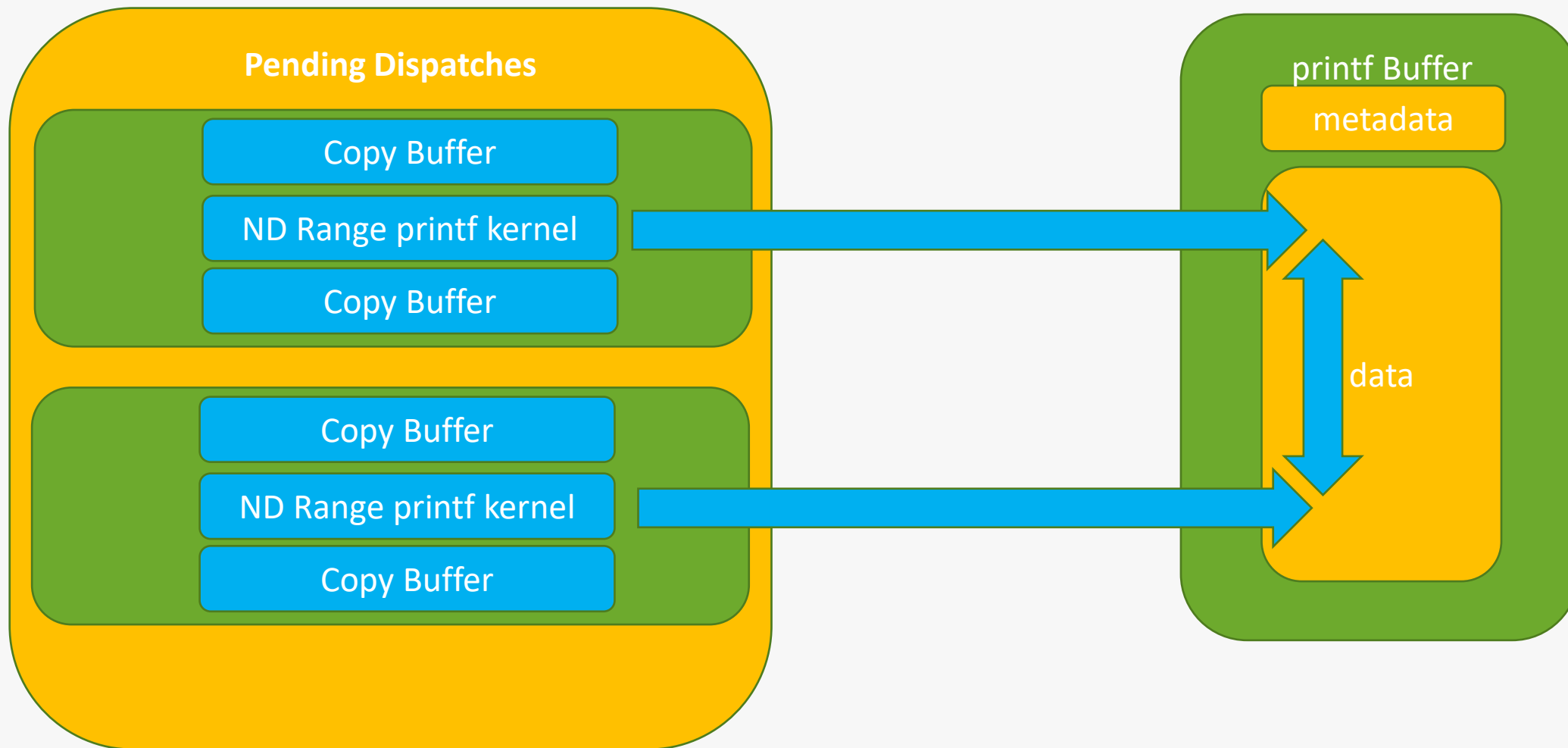
# Problem: printf



# Solution: printf

- `CL_COMMAND_BUFFER_CAPABILITY_KERNEL_PRINTF_KHR` allows implementation to opt out of supporting printf in kernels in `cl_command_buffer_khr` objects
- ComputeAorta works around this by offsetting into buffer for each subsequent printf call

# Problem: printf



# Next Steps

- Khronos OpenCL Working Group
  - Release layered extensions.
  - Finally ratified extension rather than provisional.
- Codeplay
  - Prototyping SYCL functionality on top of the OpenCL extension.
  - Implement layered extensions.

Feedback on the extension greatly appreciated!  
<https://github.com/KhronosGroup/OpenCL-Docs/issues>

We're  
Hiring!  
[codeplay.com/careers/](https://codeplay.com/careers/)



Enable AI & HPC to be Open, Safe and Accessible to All

# Thank you for watching

[ewan@codeplay.com](mailto:ewan@codeplay.com)

[jack@codeplay.com](mailto:jack@codeplay.com)



[@codeplaysoft](https://twitter.com/codeplaysoft)



[info@codeplay.com](mailto:info@codeplay.com)



[codeplay.com](https://codeplay.com)