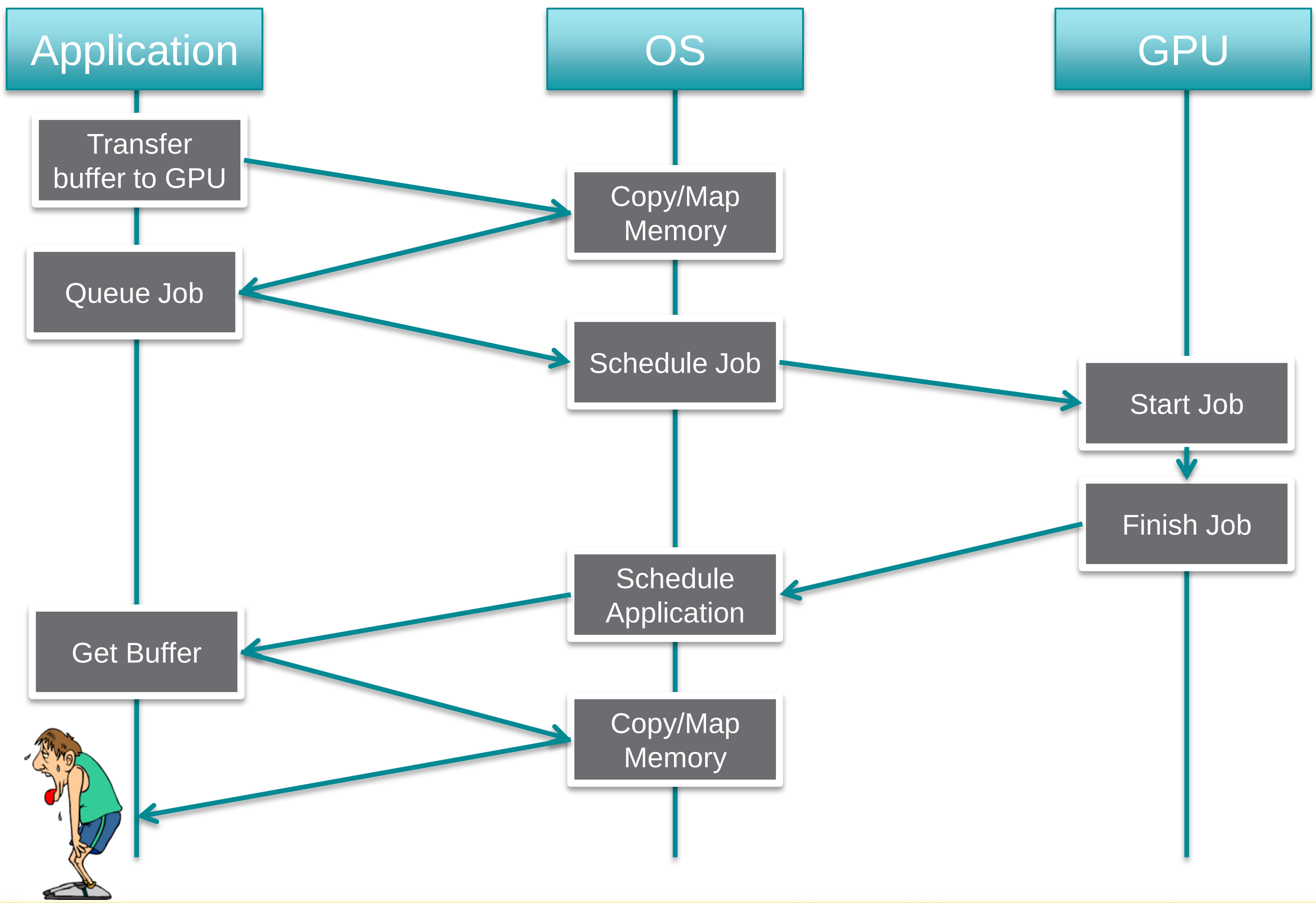
HSA QUEUEING

HOT CHIPS TUTORIAL - AUGUST 2013

IAN BRATT
PRINCIPAL ENGINEER
ARM

HSA QUEUEING, MOTIVATION

MOTIVATION (TODAY'S PICTURE)



HSA QUEUEING: REQUIREMENTS

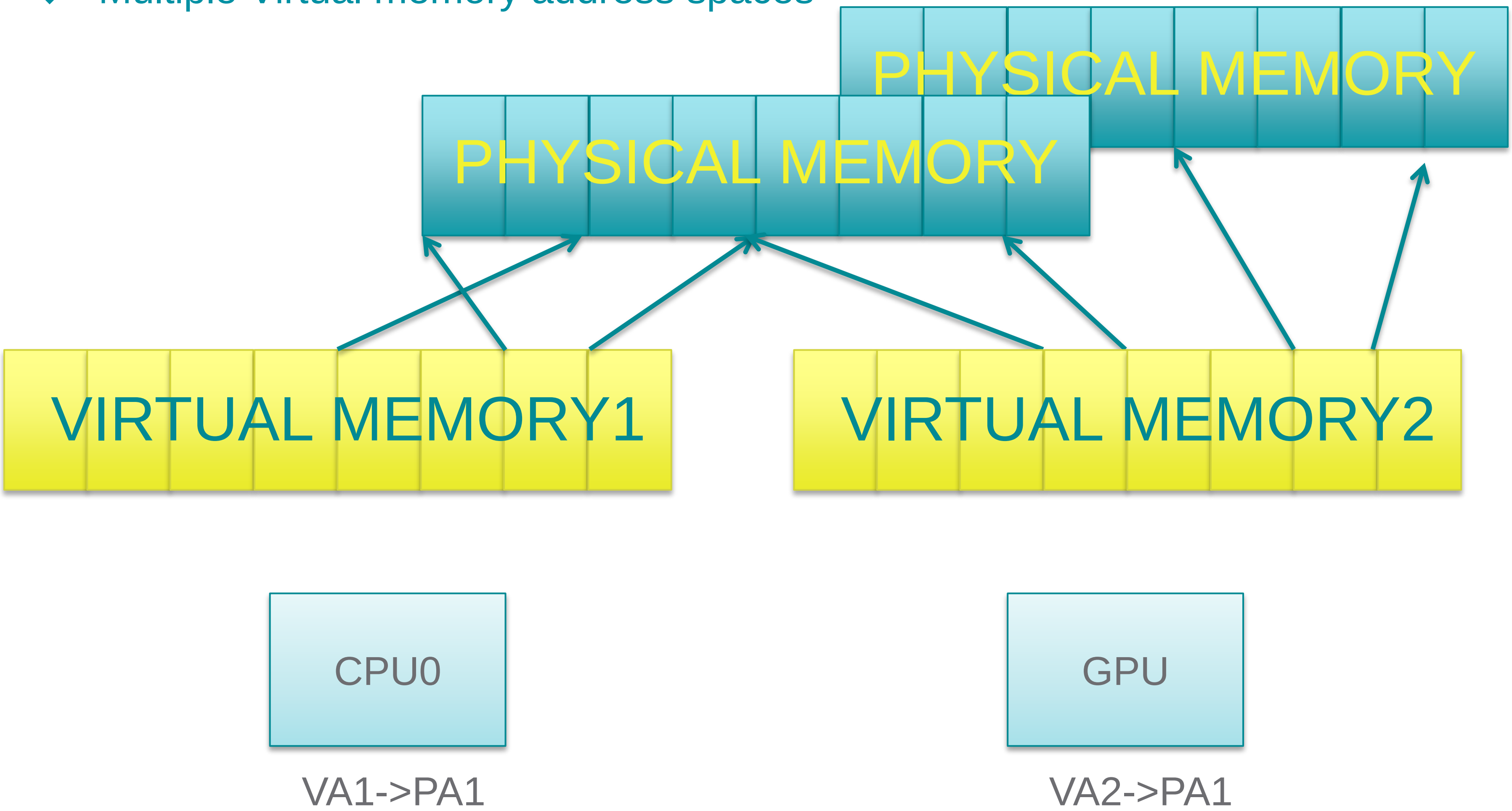
REQUIREMENTS

- Requires four mechanisms to enable lower overhead job dispatch.
  - Shared Virtual Memory
  - System Coherency
  - Signaling
  - User mode queueing

SHARED VIRTUAL MEMORY

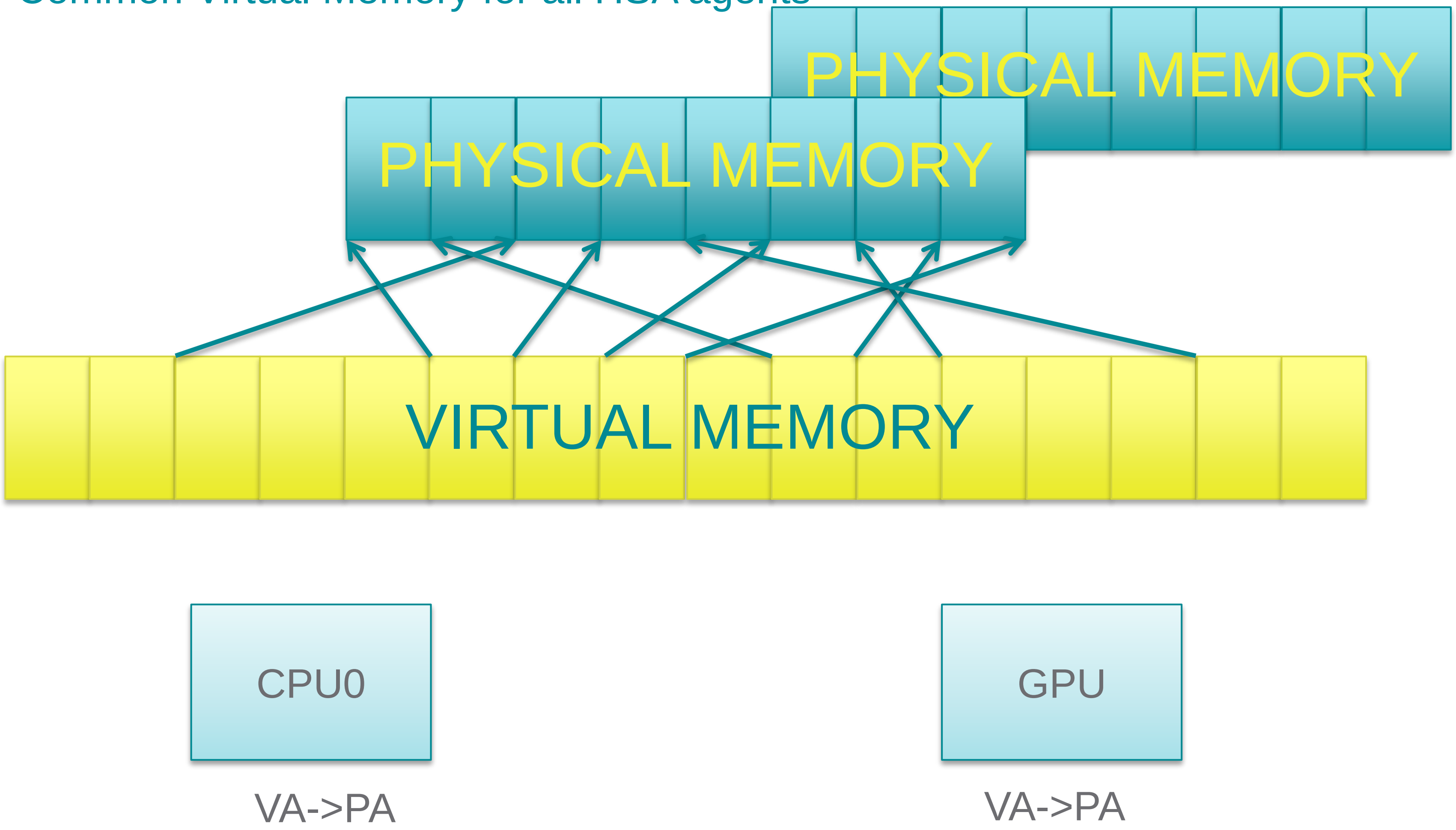
SHARED VIRTUAL MEMORY (TODAY)

- Multiple Virtual memory address spaces



SHARED VIRTUAL MEMORY (HSA)

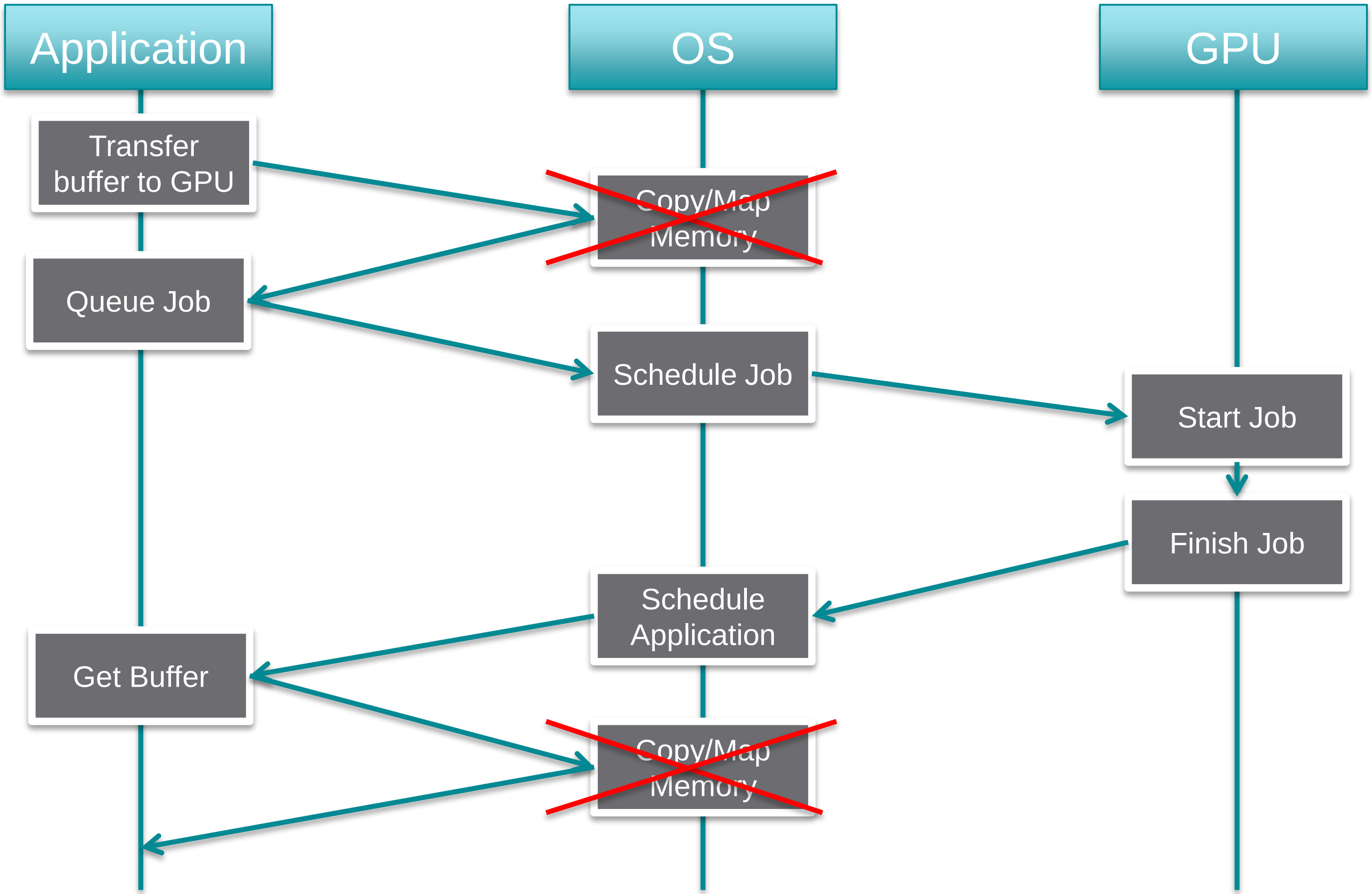
- Common Virtual Memory for all HSA agents



SHARED VIRTUAL MEMORY

- Advantages
  - No mapping tricks, no copying back-and-forth between different PA addresses
  - Send pointers (not data) back and forth between HSA agents.
- Implications
  - Common Page Tables (and common interpretation of architectural semantics such as shareability, protection, etc).
  - Common mechanisms for address translation (and servicing address translation faults)
  - Concept of a process address space (PASID) to allow multiple, per process virtual address spaces within the system.

GETTING THERE …



SHARED VIRTUAL MEMORY

- Specifics
  - Minimum supported VA width is 48b for 64b systems, and 32b for 32b systems.
  - HSA agents may reserve VA ranges for internal use via system software.
  - All HSA agents other than the host unit must use the lowest privelege level
  - If present, read/write access flags for page tables must be maintained by all agents.
  - Read/write permissions apply to all HSA agents, equally.

CACHE COHERENCY

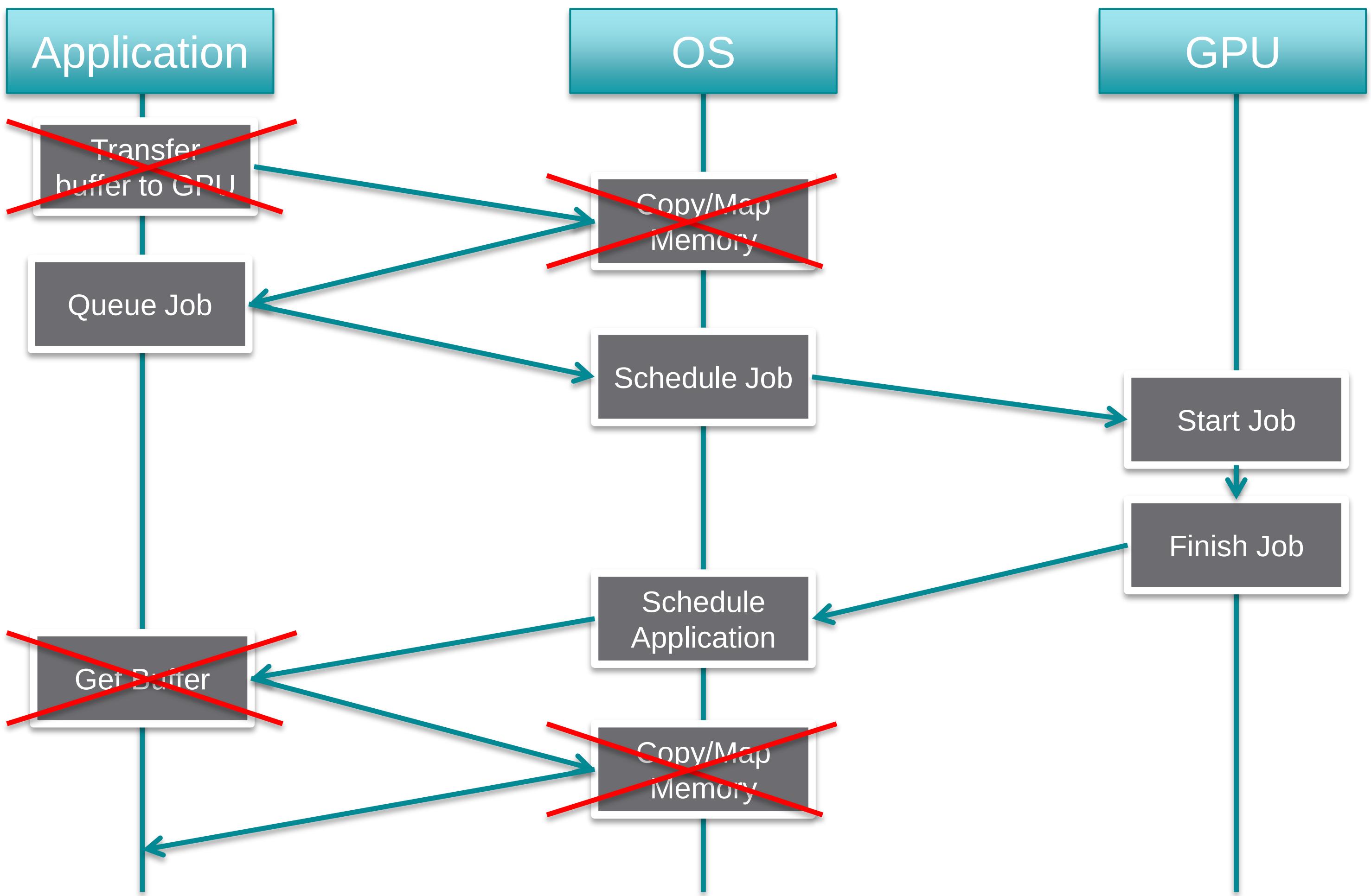
CACHE COHERENCY DOMAINS (1/3)

- *Data accesses to global memory segment from all HSA Agents shall be coherent without the need for explicit cache maintenance.*

CACHE COHERENCY DOMAINS (2/3)

- Advantages
  - Composability
  - Reduced SW complexity when communicating between agents
  - Lower barrier to entry when porting software
- Implications
  - Hardware coherency support between all HSA agents
  - Can take many forms
    - Stand alone Snoop Filters / Directories
    - Combined L3/Filters
    - Snoop-based systems (no filter)
    - Etc …

GETTING CLOSER …



CACHE COHERENCY DOMAINS (3/3)

- Specifics
  - No requirement for instruction memory accesses to be coherent
  - Only applies to the Primary memory type.
  - No requirement for HSA agents to maintain coherency to any memory location where the HSA agents do not specify the same memory attributes
  - Read-only image data is required to remain static during the execution of an HSA kernel.
    - No double mapping (via different attributes) in order to modify. Must remain static

SIGNALING

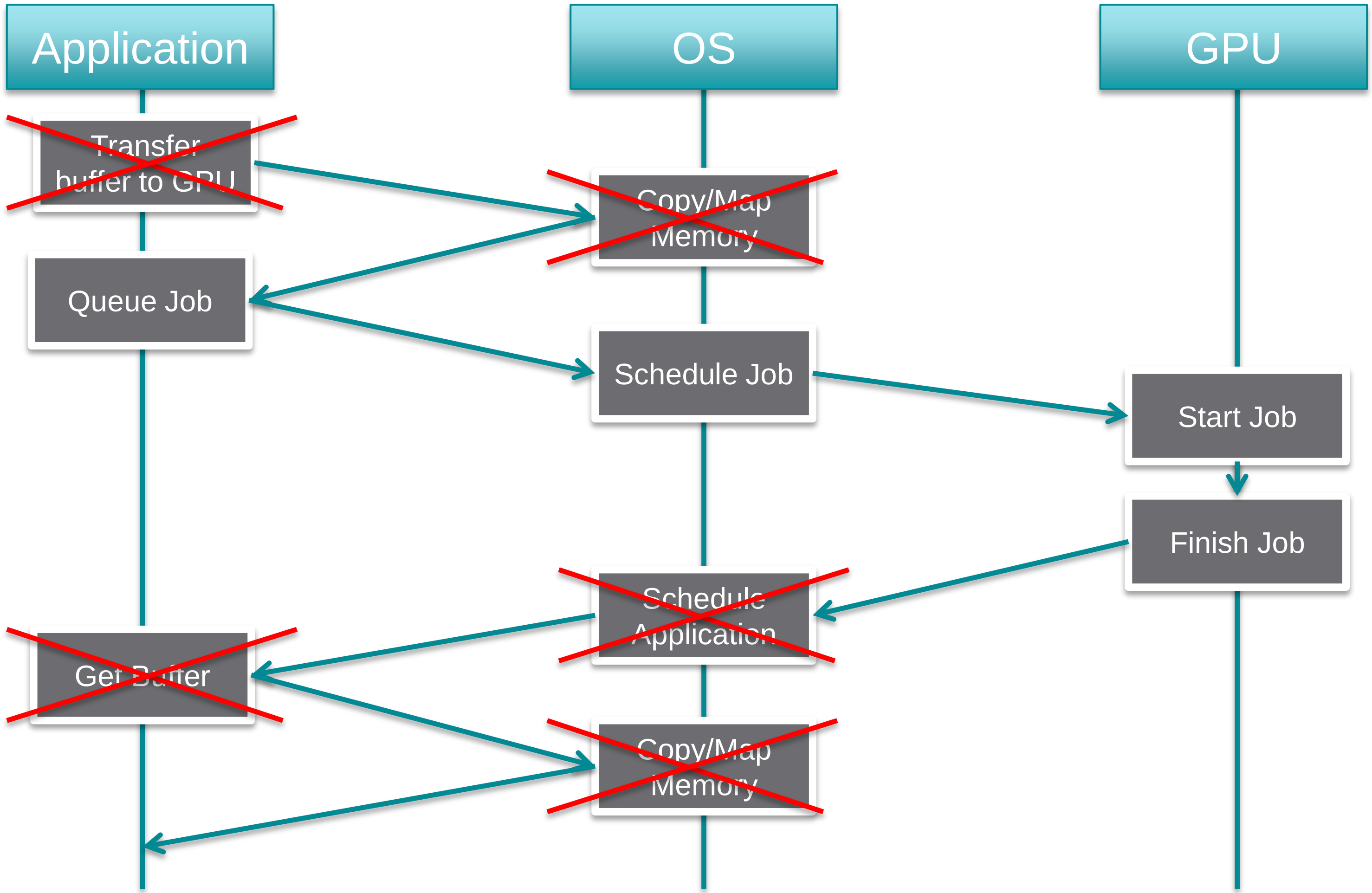
SIGNALING (1/3)

- HSA agents support the ability to use signaling objects
  - All creation/destruction signaling objects occurs via HSA runtime APIs
    - Object creation/destruction
  - From an HSA Agent you can directly accessing signaling objects.
    - Signaling a signal object (this will wake up HSA agents waiting upon the object)
    - Query current object
    - Wait on the current object (various conditions supported).

SIGNALING (2/3)

- Advantages
  - Enables asynchronous interrupts between HSA agents, without involving the kernel
  - Common idiom for work offload
  - Low power waiting
- Implications
  - Runtime support required
  - Commonly implemented on top of cache coherency flows

ALMOST THERE…



SIGNALING (3/3)

- Specifics
  - Only supported within a PASID
  - Supported wait conditions are =, !=, < and >=
  - Wait operations may return sporadically (no guarantee against false positives)
    - Programmer must test.
  - Wait operations have a maximum duration before returning.
  - The HSAIL atomic operations are supported on signal objects.
  - Signal objects are opaque

USER MODE QUEUEING

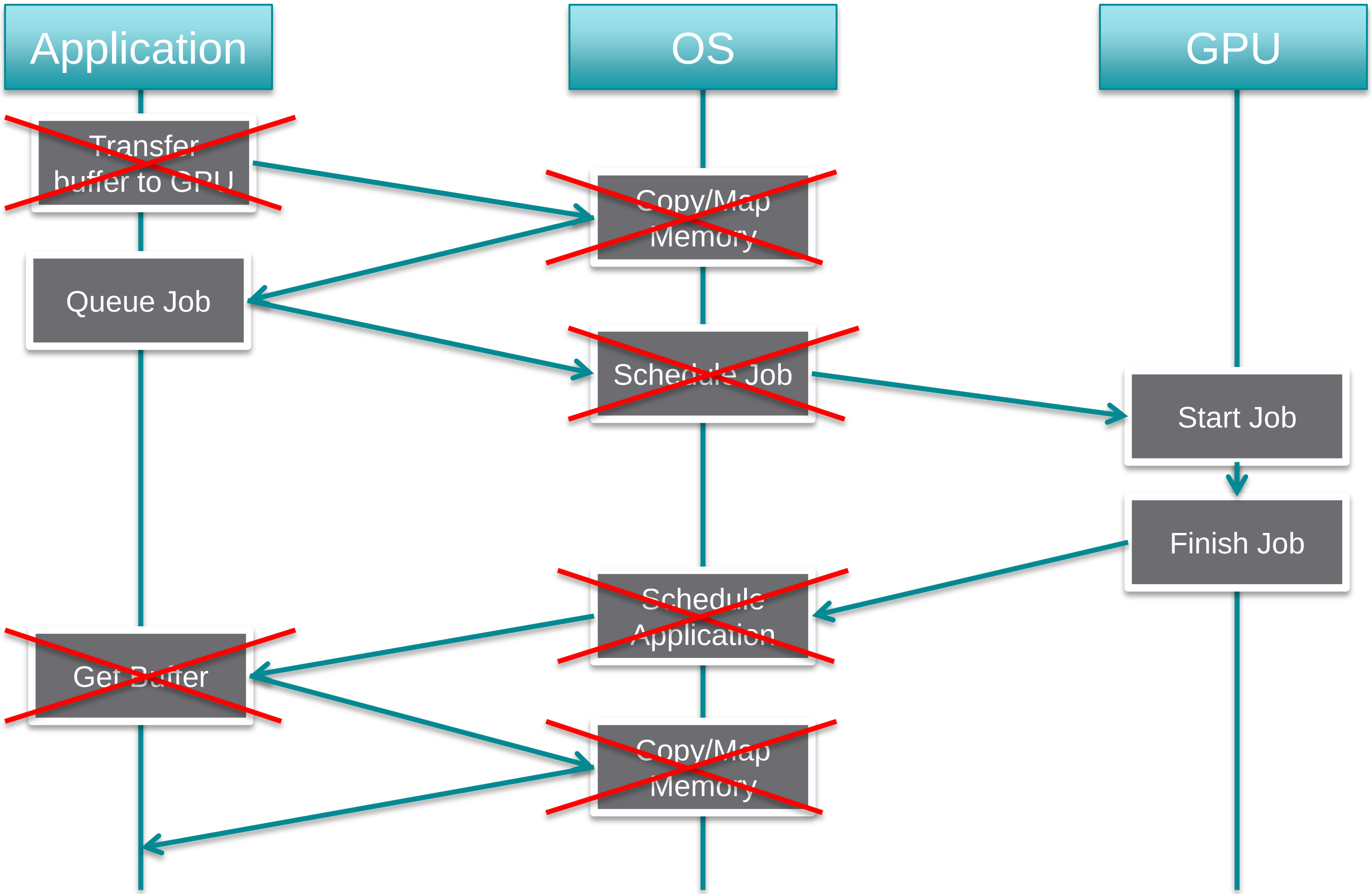
USER MODE QUEUEING (1/3)

- User mode Queueing
  - Enables user space applications to directly, without OS intervention, enqueue jobs ("Dispatch Packets") for HSA agents.
    - Dispatch packet is a job of work
  - Support for multiple queues per PASID
  - Multiple threads/agents within a PASID may enqueue Packets in the same Queue.
  - Dependency mechanisms created for ensuring ordering between packets.

USER MODE QUEUEING (2/3)

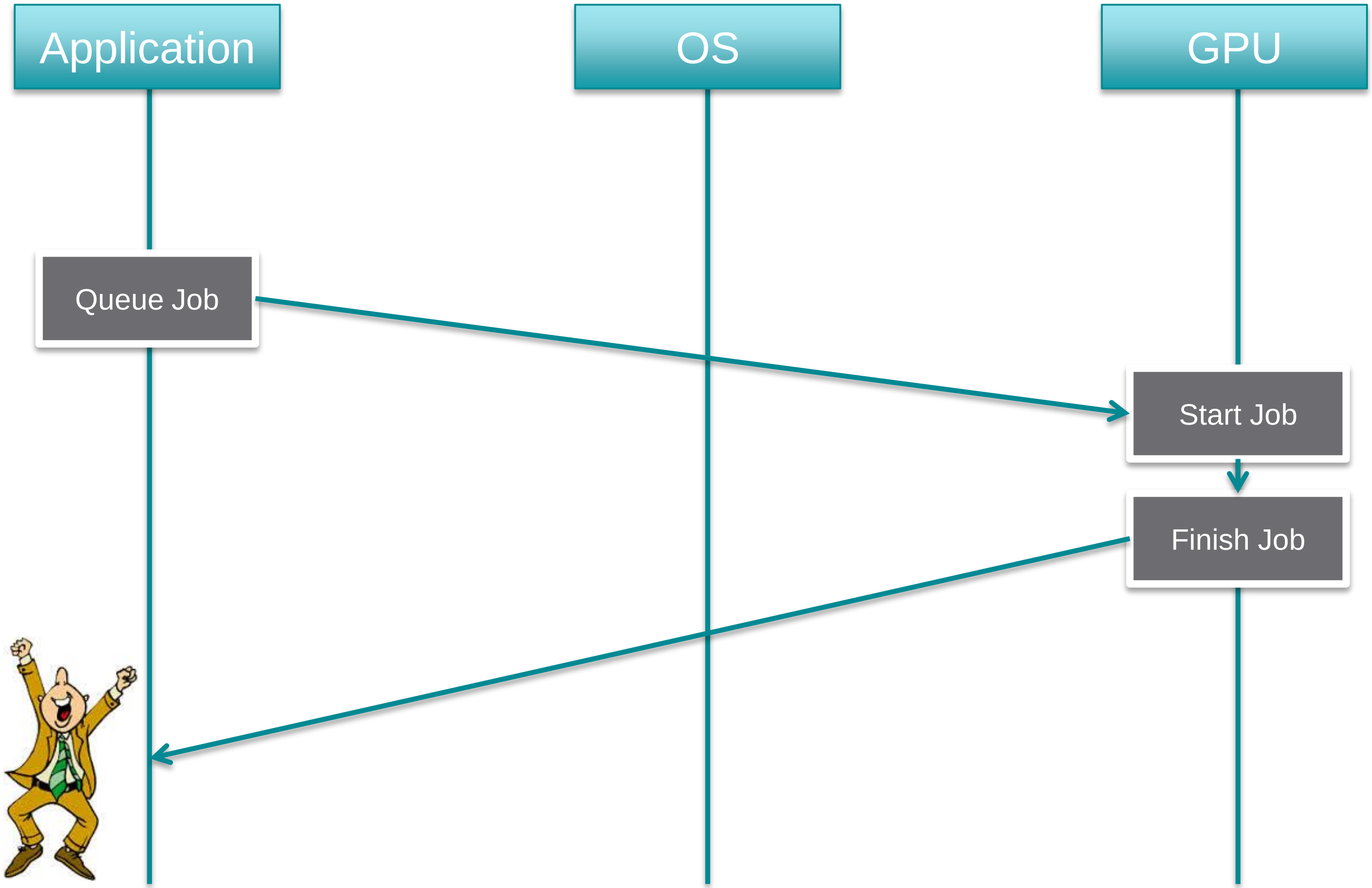
- Advantages
  - Avoid involving the kernel/driver when dispatching work for an Agent.
  - Lower latency job dispatch enables finer granularity of offload
  - Standard memory protection mechanisms may be used to protect communication with the consuming agent.
- Implications
  - Packet formats/fields are *Architected* – standard across vendors!
    - Guaranteed backward compatibility
  - Packets are enqueued/dequeued via an Architected protocol (all via memory accesses and signalling)
  - More on this later……

SUCCESS!



Application
OS
GPU
Transfer buffer to GPU
Copy/Map Memory
Queue Job
Schedule Job
Start Job
Finish Job
Schedule Application
Get Buffer
Copy/Map Memory

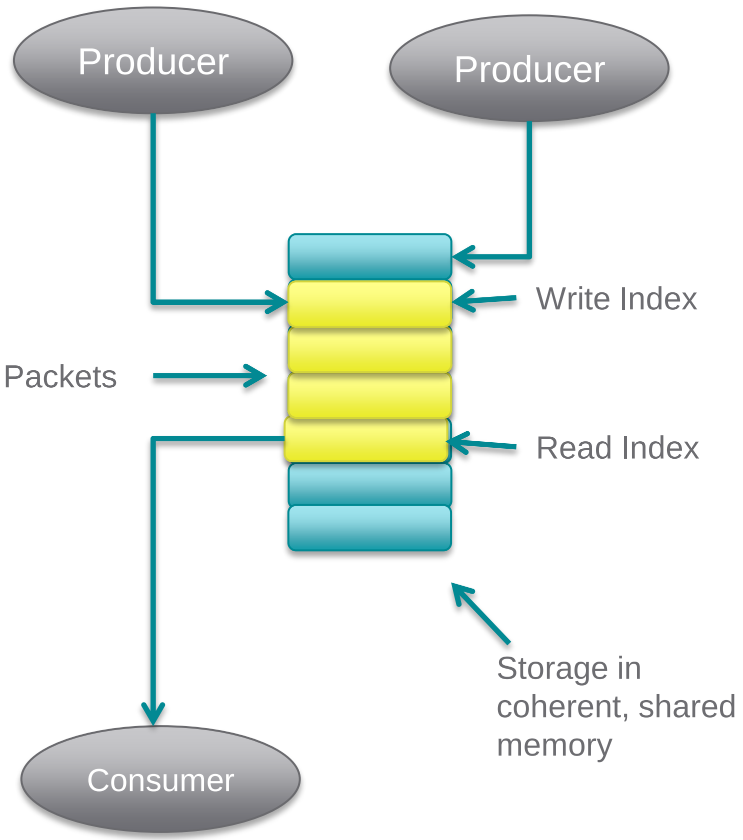
SUCCESS!



ARCHITECTED QUEUEING LANGUAGE, QUEUES

ARCHITECTED QUEUEING LANGUAGE

- HSA Queues look just like standard shared memory queues, supporting multi-producer, single-consumer
  - Support is allowed for single-producer, single-consumer
- Queues consist of storage, read/write indices, ID, etc.
- Queues are created/destroyed via calls to the HSA runtime
- “Packets” are placed in queues directly from user mode, via an *architected* protocol
- Packet format is architected



ARCHITECTED QUEUEING LANGUAGE

- Once a packet is enqueued, the producer signals the doorbell
  - Consumers are not required to wait on the doorbell – the consumer could instead be polling.
  - The doorbell is not the synchronization mechanism (the shared memory updates ensure the synchronization).
- Packets are read and dispatched for execution from the queue in order, but may complete in any order.
  - There is no guarantee that more than one packet will be processed in parallel at a time
- There may be many queues. A single agent may also consume from several queues.
- A packet processing agent may also enqueue packets.

POTENTIAL MULTI-PRODUCER ALGORITHM

```
// Read the current queue write offset
tmp_WriteOffset = WriteOffset;

// wait until the queue is no longer full.
while(tmp_WriteOffset == ReadOffset + Size) {}

// Atomically bump the WriteOffset
if (WriteOffset.compare_exchange_strong(tmp_WriteOffset, tmp_WriteOffset + 1,
                                         std::memory_order_acquire){

    // calculate index
    uint32_t index = tmp_WriteOffset & (Size -1);

    // copy over the packet, the format field is INVALID
    BaseAddress[index] = pkt;

    // Update format field with release semantics
    BaseAddress[index].hdr.format.store(DISPATCH, std::memory_order_release);

     // ring doorbell, with release semantics (could also amortize over multiple packets)
    hsa_ring_doorbell(tmp_WriteOffset+1);
  }
```

POTENTIAL CONSUMER ALGORITHM

```
// spin while empty (could also perform low-power wait on doorbell)
while (BaseAddress[ReadOffset & (Size - 1)].hdr.format == INVALID) { }

// calculate the index
uint32_t index = ReadOffset & (Size - 1);

// copy over the packet
pkt = BaseAddress[index];

// set the format field to invalid
BaseAddress[index].hdr.format.store(INVALID, std::memory_order_relaxed);

// Update the readoffset
ReadOffset.store(ReadOffset + 1, std::memory_order_release);
```

ARCHITECTED QUEUEING LANGUAGE, PACKETS

DISPATCH PACKET

- Packets come in two main types (Dispatch and Barrier), with architected layouts
- Dispatch packet is the most common type of packet
- Contains
  - Pointer to the kernel
  - Pointer to the arguments
  - WorkGroupSize (x,y,z)
  - gridSize(x,y,z)
  - And more……
- Packets contain an additional “barrier” flag. When the barrier flag is set, no other packets will be launched until all previously launched packets from this queue have completed.

DISPATCH PACKET

Offset	Format	Field Name	Description
0	uint32_t	format:8	AQL_FORMAT: 0=INVALID, 1=DISPATCH, 2=DEPEND, others reserved
		barrier:1	If set then processing of packet will only begin when all preceding packets are complete.
		acquireFenceScope:2	Determines the scope and type of the memory fence operation applied before the job is dispatched.
		releaseFenceScope:2	Determines the scope and type of the memory fence operation applied after kernel completion but before the job is completed.
		invalidateInstructionCache:1	Acquire fence additionally applies to any instruction cache(s).
		invalidateROImageCache:1	Acquire fence additionally applies to any read-only image cache(s).
		dimensions:2	Number of dimensions specified in gridSize. Valid values are 1, 2, or 3.
		reserved:15	
4	uint16_t	workgroupSize.x	x dimension of work-group (measured in work-items).
6	uint16_t	workgroupSize.y	y dimension of work-group (measured in work-items).
8	uint16_t	workgroupSize.z	z dimension of work-group (measured in work-items).
10	uint16_t	reserved2	
12	uint32_t	gridSize.x	x dimension of grid (measured in work-items).
16	uint32_t	gridSize.y	y dimension of grid (measured in work-items).
20	uint32_t	gridSize.z	z dimension of grid (measured in work-items).
24	uint32_t	privateSegmentSizeBytes	Total size in bytes of private memory allocation request (per work-item).
28	uint32_t	groupSegmentSizeBytes	Total size in bytes of group memory allocation request (per work-group).
32	uint64_t	kernelObjectAddress	Address of an object in memory that includes an implementation-defined executable ISA image for the kernel.
40	uint64_t	kernargAddress	Address of memory containing kernel arguments.
48	uint64_t	reserved3	
56	uint64_t	completionSignal	Address of HSA signaling object used to indicate completion of the job.

BARRIER PACKET

- Used for specifying dependences between packets
- HSA will not launch any further packets from this queue until the barrier packet signal conditions are met
- Used for specifying dependences on packets dispatched from any queue.
  - Execution phase completes only when all of the dependent signals (up to five) have been signaled (with the value of 0).
  - Or if an error has occurred in one of the packets upon which we have a dependence.

BARRIER PACKET

Offset	Format	Field Name	Description
0	uint32_t	format:8	AQL_FORMAT: 0=INVALID, 1=DISPATCH, 2=DEPEND, others reserved
		barrier:1	If set then processing of packet will only begin when all preceding packets are complete.
		acquireFenceScope:2	Determines the scope and type of the memory fence operation applied before the job is dispatched.
		releaseFenceScope:2	Determines the scope and type of the memory fence operation applied after kernel completion but before the job is completed.
		invalidateInstructionCache:1	Acquire fence additionally applies to any instruction cache(s).
		invalidateROImageCache:1	Acquire fence additionally applies to any read-only image cache(s).
		dimensions:2	Number of dimensions specified in gridSize. Valid values are 1, 2, or 3.
		reserved:15	
4	uint32_t	reserved2	
8	uint64_t	depSignal0	Address of dependent signaling objects to be evaluated by the packet processor.
16	uint64_t	depSignal1	
24	uint64_t	depSignal2	
32	uint64_t	depSignal3	
40	uint64_t	depSignal4	
48	uint64_t	reserved3	
56	uint64_t	completionSignal	Address of HSA signaling object used to indicate completion of the job.

DEPENDENCES

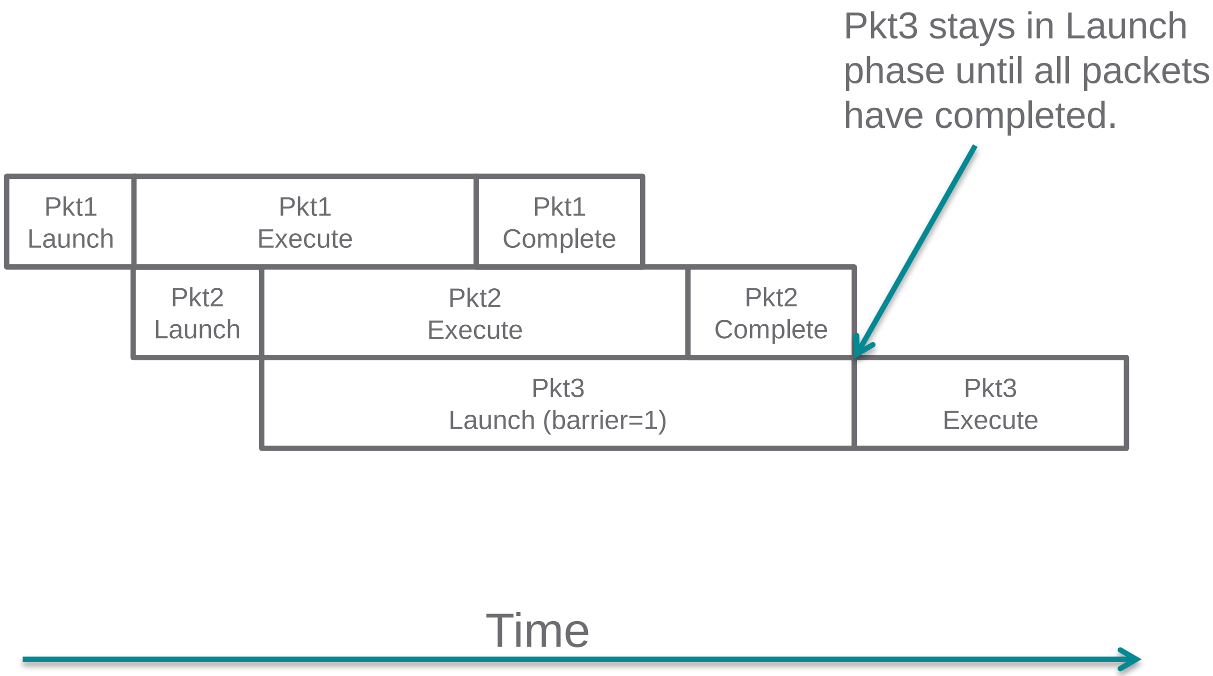
- A user may never assume more than one packet is being executed by an HSA agent at a time.
- Implications:
  - Packets can’t poll on shared memory values which will be set by packets issued from other queues, unless the user has ensured the proper ordering.
  - To ensure all previous packets from a queue have been completed, use the Barrier bit.
  - To ensure specific packets from any queue have completed, use the Barrier packet.

HSA QUEUEING, PACKET EXECUTION

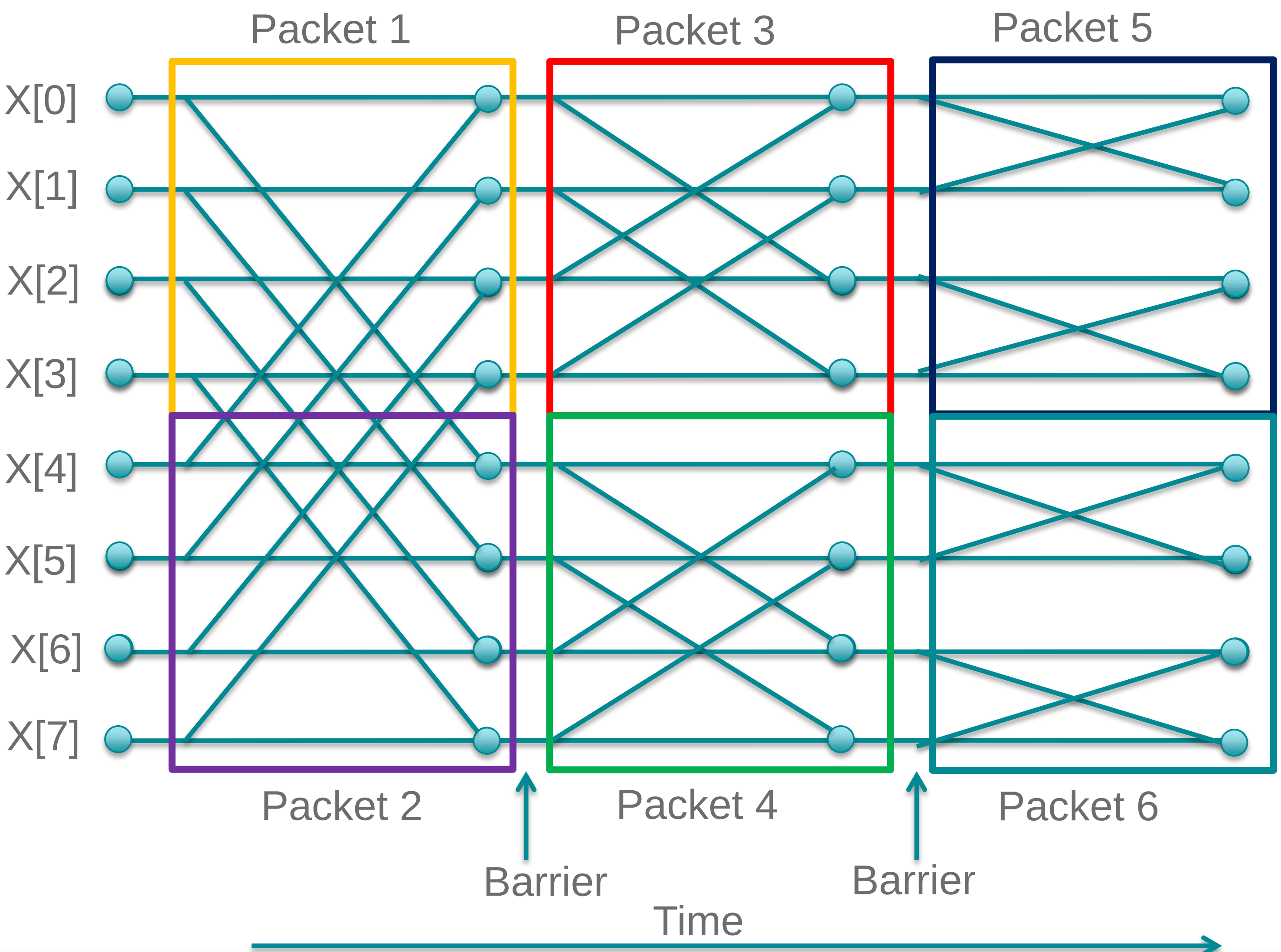
PACKET EXECUTION

- Launch phase
  - Initiated when launch conditions are met
    - All preceeding packets in the queue must have exited launch phase
    - If the barrier bit in the packet header is set, then all preceding packets in the queue must have exited completion phase
- Active phase
  - Execute the packet
  - Barrier packets remain in Active phase until conditions are met.
- Completion phase
  - First step is memory release fence – make results visible.
  - CompletionSignal field is then signaled with a decrementing atomic.

PACKET EXECUTION



PUTTING IT ALL TOGETHER (FFT)



PUTTING IT ALL TOGETHER

AQL Pseudo Code

```
// Send the packets to do the first stage.
aql_dispatch(pkt1);
aql_dispatch(pkt2);

// Send the next two packets, setting the barrier bit so we
//know packets 1 &2 will be complete before 3 and 4 are
//launched.
aql_dispatch_with _barrier_bit(pkt3);
aql_dispatch(pkt4);

// Same as above (make sure 3 & 4 are done before issuing 5
//& 6)
aql_dispatch_with_barrier_bit(pkt5);
aql_dispatch(pkt6);

// This packet will notify us when 5 & 6 are complete)
aql_dispatch_with_barrier_bit(finish_pkt);
```

QUESTIONS?