OPENCL PROGRAMMING AND OPTIMIZATION – PART II

HAIBO XIE, PH.D.
haibo.xie@amd.com

OPENCL PERFORMANCE CONSIDERATION ON GPUS

- CPU + dGPU with OpenCL has obvious bottlenecks
  - CPU/GPU data movement is a side effect
  - dGPU has limited memory size
  - CPU + dGPU has seeable overhead of cooperation under OpenCL runtime

- Try to narrow the side effects down as much as possible
  - CPU/GPU data movement over PIC-E or other bus is the introduced overhead
  - Double buffering or APU platform is the ideal technology to reduce the overhead

- Ideas to tune overall system performance should be paid attention
  - Double buffering for dGPU
  - APU platform for eliminating CPU/GPU data movement
  - HSA technique gives CPU/GPU cooperation a more harmonious way

AGENDA

- OpenCL system performance
  - CPU/GPU data movement
  - OpenCL runtime overhead
- APU architecture and OpenCL optimization
- HSA and OpenCL optimization

CPU/GPU DATA MOVEMENT

- For normal CPU + dGPU platform, a single buffer for computing and data movement looks like the below

- There's additional time consuming for CPU <-> GPU data movement which is introduced side effect
- This side effect is even worse in the case that:
  - Data movement time is significantly larger than Kernel time
  - Or Data movement time is even larger than CPU computing time

OPENCL APPLICATION OPTIMIZATION

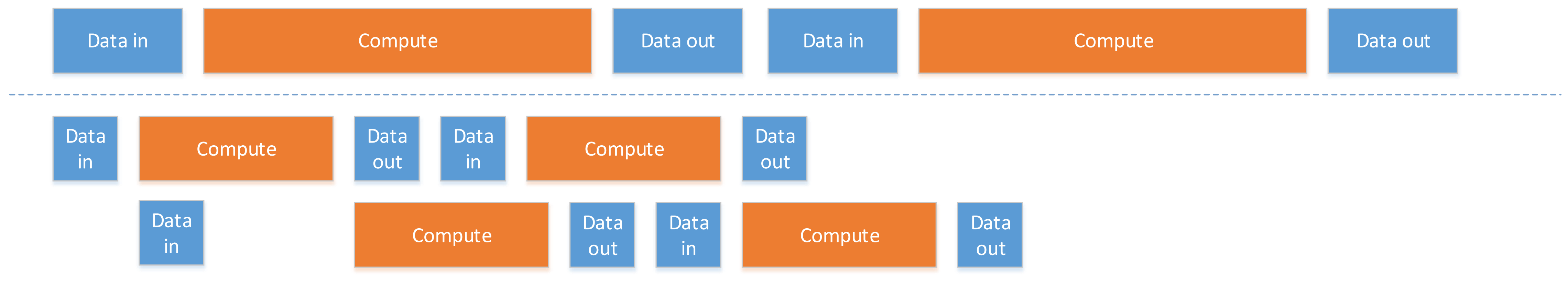
DOUBLE BUFFERING

- Very useful common technique
  - One buffer is computing while another buffer is filled in data
  - To overlap the time of computing and the time of CPU/GPU data movement
  - Especially useful for CPU + dGPU platform

- With AMD OpenCL implementation, DMA is asynchronous
  - Use two command queue, one for buffer en-queue operation and another for Kernel operation
  - Use event to synchronize

DOUBLE BUFFERING

- We can introduce double buffering technique for GPU offload computing mode



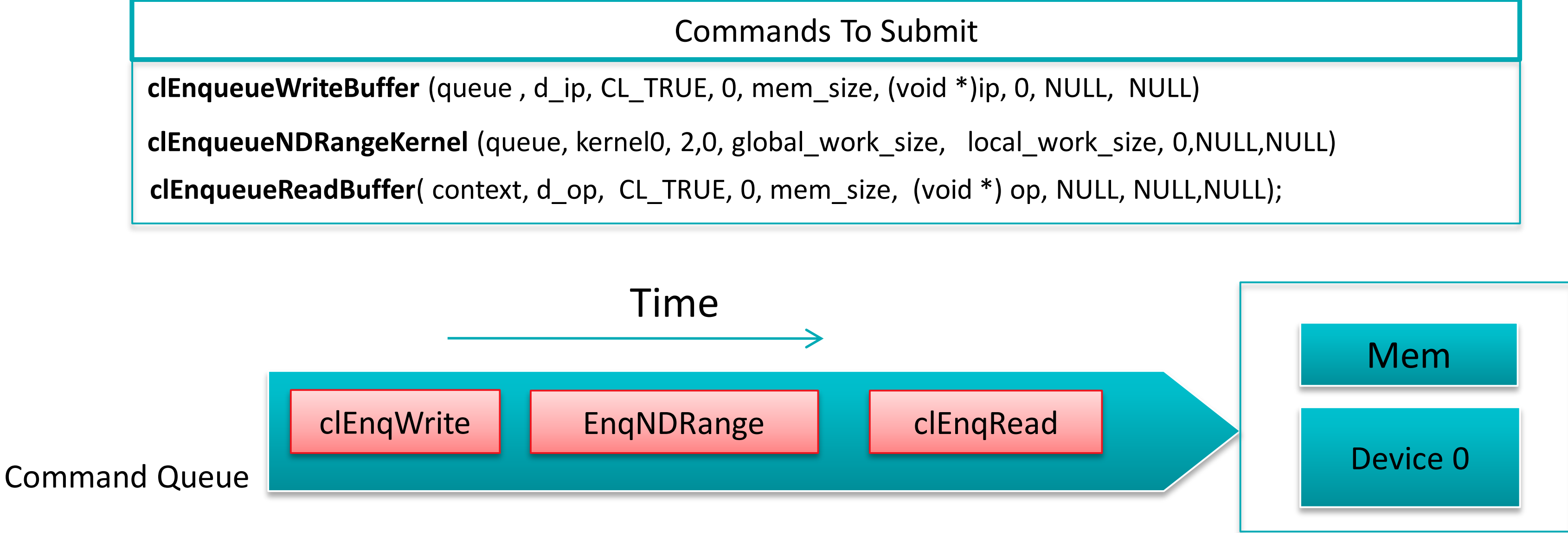
- For the above N times Kernel invocation
  - We can reduce N-1 times data movement time

COMMAND QUEUES

- We need to measure the performance of an application as a whole and not just our optimized kernels to understand bottlenecks
- This necessitates understanding of OpenCL synchronization techniques and events
- Command queues are used to submit work to a device
- Two main types of command queues
  – In Order Queue
  – Out of Order Queue

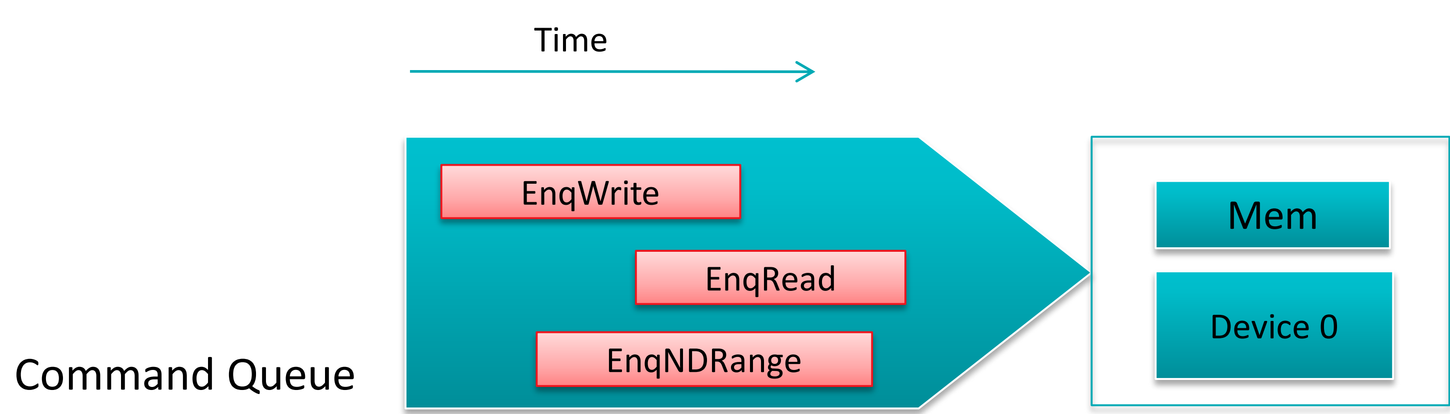
IN-ORDER EXECUTION

- In an in-order command queue, each command executes after the previous one has finished
  - For the set of commands shown, the read from the device would start after the kernel call has finished
- Memory transactions have consistent view



OUT-OF-ORDER EXECUTION

- In an out-of-order command queue, commands are executed as soon as possible, without any ordering guarantees
- All memory operations occur in single memory pool
- Out-of-order queues result in memory transactions that will overlap and clobber data without some form of synchronization
- The commands discussed in the previous slide could execute in any order on device



SYNCHRONIZATION IN OPENCL

- Synchronization is required if we use an out-of-order command queue or multiple command queues
- Coarse synchronization granularity
  – Per command queue basis
- Finer synchronization granularity
  – Per OpenCL operation basis using events
- Synchronization in OpenCL is restricted to within a context
- This is similar to the fact that it is not possible to share data between multiple contexts without explicit copying
- The proceeding discussion of synchronization is applicable to any OpenCL device (CPU or GPU)

OPENCL COMMAND QUEUE CONTROL

- Command queue synchronization methods work on a per-queue basis
- Flush: clFlush(cl_commandqueue)
  - Send all commands in the queue to the compute device
  - No guarantee that they will be complete when clFlush returns
- Finish: clFinish(cl_commandqueue)
  - Waits for all commands in the command queue to complete before proceeding (host blocks on this call)
- Barrier: clEnqueueBarrier(cl_commandqueue)
  - Enqueue a synchronization point that ensures all prior commands in a queue have completed before any further commands execute

SYNCHRONIZATION FOR CLENQUEUE FUNCTIONS

- Functions like clEnqueueReadBuffer and clEnqueueWriteBuffer have a boolean parameter to determine if the function is blocking
  - This provides a blocking construct that can be invoked to block the host
- If blocking is **TRUE**, OpenCL enqueues the operation using the host pointer in the command-queue
  - Host pointer **can** be reused by the application after the enqueue call returns
- If blocking is **FALSE**, OpenCL will use the host pointer parameter to perform a non-blocking read/write and returns immediately
  - Host pointer **cannot** be reused safely by the application after the call returns
  - Event handle returned by clEnqueue* operations can be used to check if the non-blocking operation has completed

OPENCL EVENTS

- Previous OpenCL synchronization functions only operated on a per-command-queue granularity
- OpenCL events are needed to synchronize at a function granularity
- Explicit synchronization is required for
  - Out-of-order command queues
  - Multiple command queues
- OpenCL events are data-types defined by the specification for storing timing information returned by the device

OPENCL EVENTS

- Profiling of OpenCL programs using events has to be enabled explicitly when creating a command queue
  - `CL_QUEUE_PROFILING_ENABLE` flag must be set
  - Keeping track of events may slow down execution
- A handle to store event information can be passed for all clEnqueue* commands
  - When commands such as `clEnqueueNDRangeKernel` and `clEnqueueReadBuffer` are invoked timing information is recorded at the passed address

USES OF OPENCL EVENTS

- Using OpenCL Events we can:
  - time execution of clEnqueue* calls like kernel execution or explicit data transfers
  - use the events from OpenCL to schedule asynchronous data transfers between host and device
  - profile an application to understand an execution flow
  - observe overhead and time consumed by a kernel in the command queue versus actually executing
- **Note:** OpenCL event handling can be done in a consistent manner on both CPU and GPU for AMD and NVIDIA's implementations

CAPTURING EVENT INFORMATION

```
cl_int clGetEventProfilingInfo (
        cl_event  event,                        //event object
        cl_profiling_info  param_name,          //Type of data of event
        size_t  param_value_size,               //size of memory pointed to by param_value
        void *  param_value,                    //Pointer to returned timestamp
        size_t * param_value_size_ret)          //size of data copied to param_value
```

- clGetEventProfilingInfo allows us to query cl_event to get required counter values
- Timing information returned as cl_ulong data types
  - Returns device time counter in nanoseconds

EVENT PROFILING INFORMATION

```
cl_int clGetEventProfilingInfo (
        cl_event  event,                      //event object
        cl_profiling_info  param_name,        //Type of data of event
        size_t  param_value_size,             //size of memory pointed to by param_value
        void *  param_value,                  //Pointer to returned timestamp
        size_t * param_value_size_ret)        //size of data copied to param_value
```

- Table shows event types described using cl_profiling_info enumerated type

Event Type	Description
CL_PROFILING_COMMAND_QUEUED	Command is enqueued in a command-queue by the host.
CL_PROFILING_COMMAND_SUBMIT	Command is submitted by the host to the device associated with the command queue.
CL_PROFILING_COMMAND_START	Command starts execution on device.
CL_PROFILING_COMMAND_END	Command has finished execution on device.

CAPTURING EVENT INFORMATION

```
cl_int clGetEventInfo (
        cl_event event,                      //event object
        cl_event_info param_name,            //Specifies the information to query.
        void * param_value,                  //Pointer to memory where result queried is returned
        size_t * param_value_size_ret)       //size in bytes of memory pointed to by param_value
```

- clGetEventInfo can be used to return information about the event object
- It can return details about the command queue, context, type of command associated with events, execution status
- This command can be used by along with timing provided by clGetEventProfilingInfo as part of a high level profiling framework to keep track of commands

USER EVENTS IN OPENCL

- OpenCL defines a user event object. Unlike clEnqueue* commands, user events can be set by the user

```
cl_event clCreateUserEvent (
                cl_context context,          //OpenCL Context
                cl_int *errcode_ret )        //Returned Error Code
```

- When we create a user event, status is set to CL_SUBMITTED
- clSetUserEventStatus is used to set the execution status of a user event object. The status needs to be set to CL_COMPLETE

```
cl_mem clSetUserEventStatus (
            cl_event event,                  //User event
            cl_int execution_status)         //Execution Status
```

- A user event can only be set to CL_COMPLETE once

USING USER EVENTS

- A simple example of user events being triggered and used in a command queue

```
//Create  user event which will start the write of buf1
user_event = clCreateUserEvent(ctx, NULL);
clEnqueueWriteBuffer( cq, buf1, CL_FALSE, ..., 1, &user_event , NULL);

//The write of buf1 is now enqued and waiting on user_event
X = foo(); //Lots of complicated host processing code

clSetUserEventStatus(user_event, CL_COMPLETE);
//The clEnqueueWriteBuffer to buf1 can now proceed as per OP of foo()
```

WAIT LISTS

- All clEnqueue* methods also accept event wait lists
  - Waitlists are arrays of `cl_event` type
- OpenCL defines *waitlists* to provide precedence rules

```
err = clWaitOnEvents(1, &read_event);
```

```
clEnqueueWaitListForEvents( cl_command_queue , int, cl_event *)
```

- Enqueue a list of events to wait for such that all events need to complete before this particular command can be executed

```
clEnqueueMarker( cl_command_queue, cl_event *)
```

- Enqueue a command to mark this location in the queue with a unique event object that can be used for synchronization

EXAMPLE OF EVENT CALLBACKS

```
cl_int clSetEventCallback (
        cl_event event,                                    //Event Name
        cl_int  command_exec_type ,                        //Status on which callback is invoked
        void (CL_CALLBACK  *pfn_event_notify)              //Callback Name
         (cl_event event, cl_int event_command_exec_status, void *user_data),
        void *  user_data)                                 //User Data Passed to callback
```

- OpenCL 1.1 allows registration of a user callback function for a specific command execution status
  - Event callbacks can be used to enqueue new commands based on event state changes in a non-blocking manner
  - Using blocking versions of clEnqueue* OpenCL functions in callback leads to undefined behavior
- The callback takes an cl_event, status and a pointer to user data as its parameters

AMD EVENTS EXTENSION

- OpenCL event callbacks are valid only for the `CL_COMPLETE` state
- The `cl_amd_event_callback` extension provides the ability to register event callbacks for states other than `CL_COMPLETE`
- Lecture 10 discusses how to use vendor specific extensions in OpenCL
- The event states allowed are `CL_QUEUED`, `CL_SUBMITTED`, and `CL_RUNNING`

USING EVENTS FOR TIMING

- OpenCL events can easily be used for timing durations of kernels.
- This method is reliable for performance optimizations since it uses counters from the device
- By taking differences of the start and end timestamps we are discounting overheads like time spent in the command queue

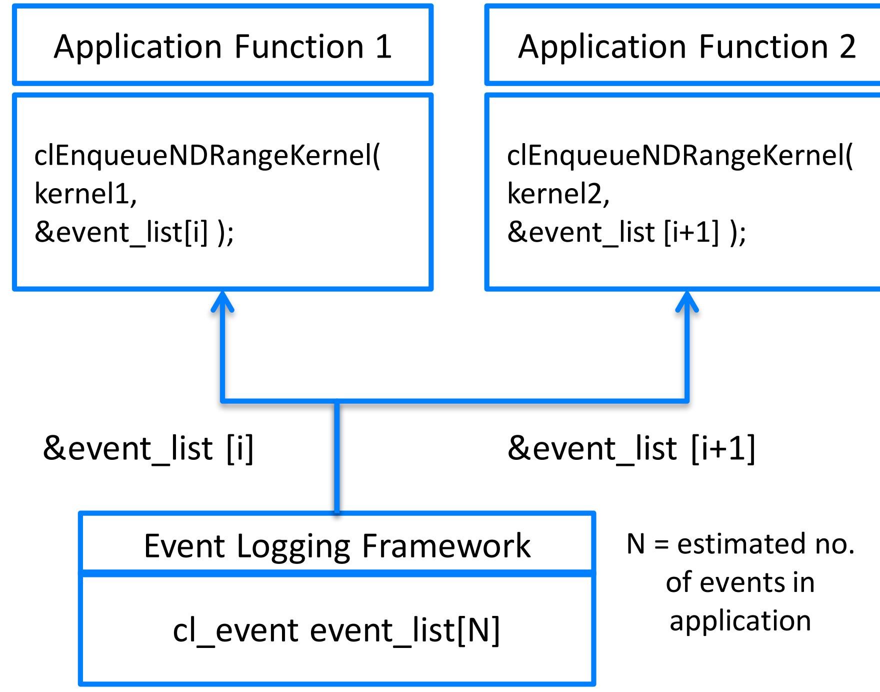
```
clGetEventProfilingInfo( event_time,
CL_PROFILING_COMMAND_START,
sizeof(cl_ulong), &starttime, NULL);
```

```
clGetEventProfilingInfo(event_time,
CL_PROFILING_COMMAND_END,
sizeof(cl_ulong), &endtime, NULL);
```

```
unsigned long elapsed =
(unsigned long)(endtime - starttime);
```

PROFILING USING EVENTS

- OpenCL calls occur asynchronously within a heterogeneous application
- A `clFinish` to capture events after each function introduces interference
- Obtaining a pipeline view of commands in an OpenCL context
  - Declare a large array of events in beginning of application
  - Assign an event from within this array to each clEnqueue* call
  - Query all events at one time after the critical path of the application



Event logging framework can query and format data stored in event_list

PROFILING WITH EVENT INFORMATION

- Before getting timing information, we must make sure that the events we are interested in have completed
- There are different ways of waiting for events:
  - `clWaitForEvents(numEvents, eventlist)`
  - `clFinish(commandQueue)`
- Timer resolution can be obtained from the flag `CL_DEVICE_PROFILING_TIMER_RESOLUTION` when calling `clGetDeviceInfo`()

EXAMPLE OF PROFILING

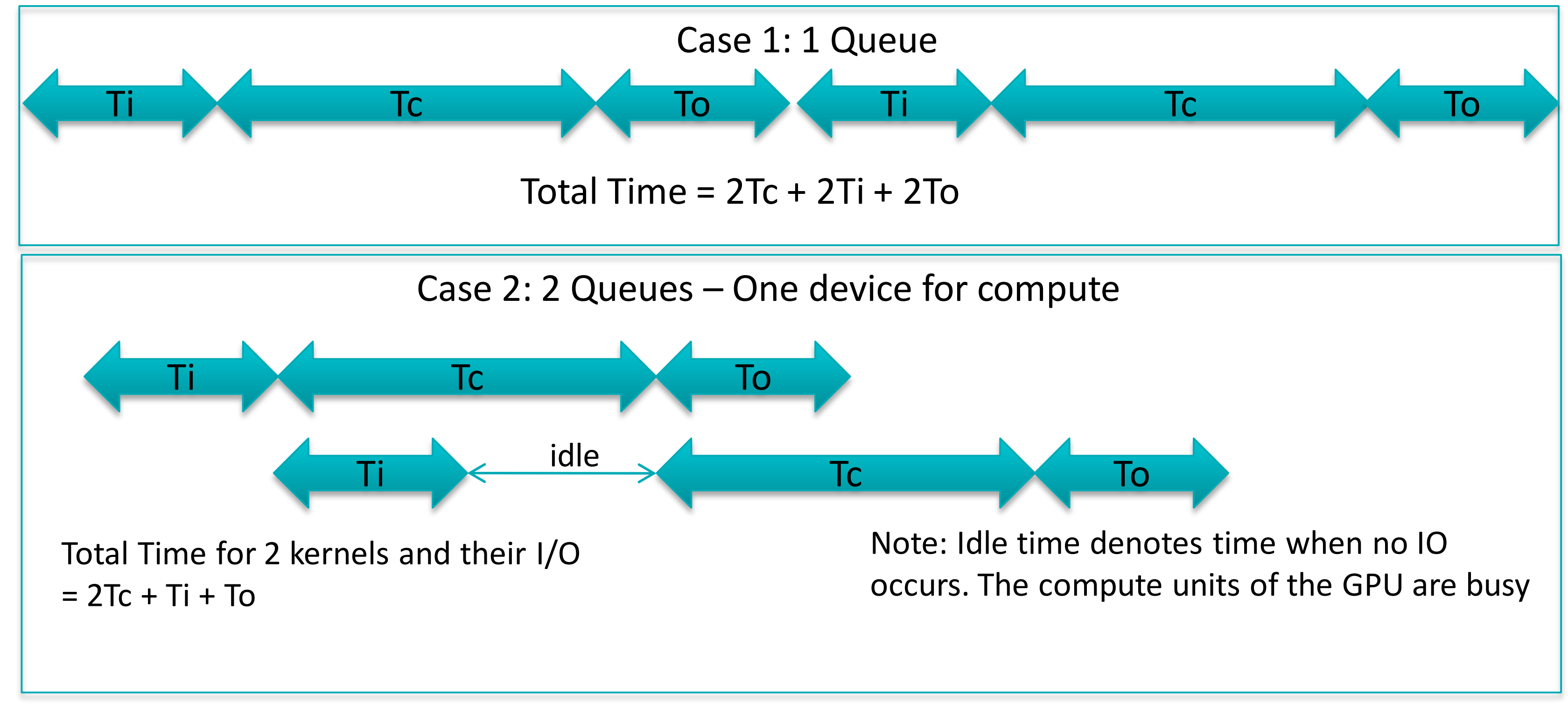
- A heterogeneous application can have multiple kernels and a large amount of host device IO
- Questions that can be answered by profiling using OpenCL events
  - We need to know which kernel to optimize when multiple kernels take similar time ?
    - Small kernels that may be called multiple times vs. large slow complicated kernel ?
  - Are the kernels spending too much time in queues ?
  - Understand proportion between execution time and setup time for an application
  - How much does host device IO matter ?
- By profiling an application with minimum overhead and no extra synchronization, most of the above questions can be answered

ASYNCHRONOUS I/O

- Overlapping host-device I/O can lead to substantial application level performance improvements
- How much can we benefit from asynchronous IO
- Can prove to be a non-trivial coding effort, Is it worth it ?
  - Useful for streaming workloads that can stall the GPU like medical imaging where new data is generated and processed in a continuous loop
  - Other uses include workloads like linear algebra where the results of previous time steps can be transferred asynchronously to the host
  - We need two command queues with a balanced amount of work on each queue

ASYNCHRONOUS I/O

- Asymptotic Approximation of benefit of asynchronous IO in OpenCL
  - Tc = Time for computation
  - Ti = Time to write I/P to device To = Time to read OP for device
  - (Assume host to device and device to host I/O is same)



ASYNCHRONOUS I/O

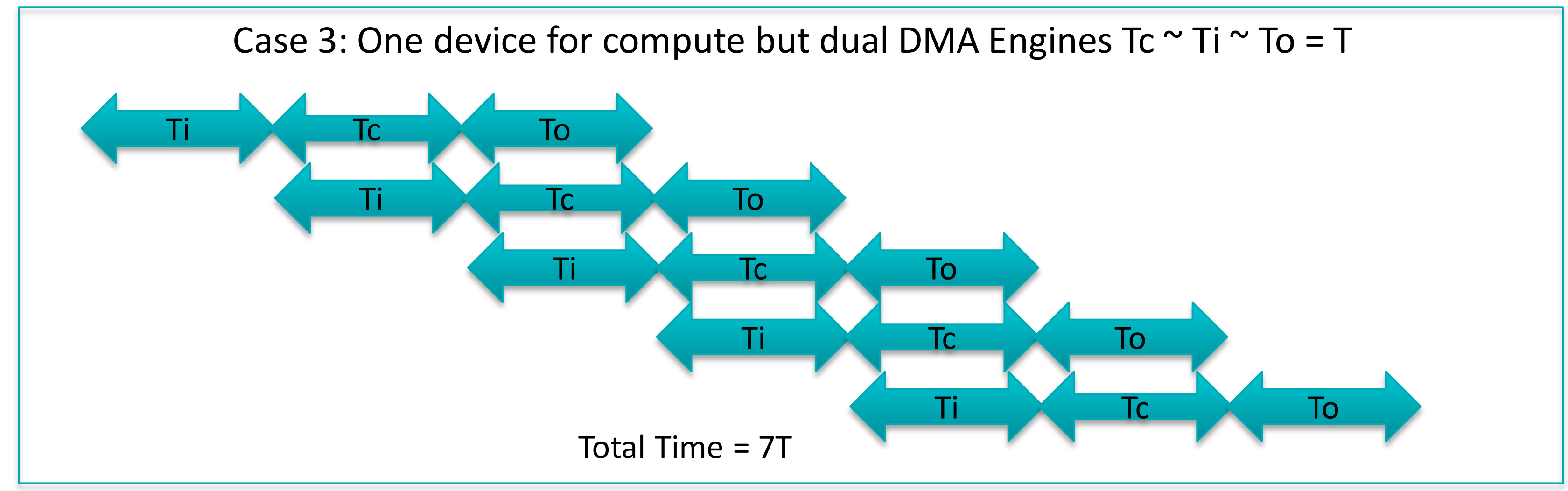
- Time with 1 Queue = 2Tc + 2Ti + 2To
  - No asynchronous behavior
- Time with 2 Queues = 2Tc + Ti + To
  - Overlap computation and communication

$$\text{Performance Benefit} = {(2Tc + Ti + To)}\Big/{(2Tc + 2Ti + 2To)}$$

- Maximum benefit achievable with similar input and output data is approximately 30% of overlap when Tc = Ti = To since that would remove the idle time shown in the previous diagram
- Host-device I/O is limited by PCI bandwidth, so it's often not quite as big a win

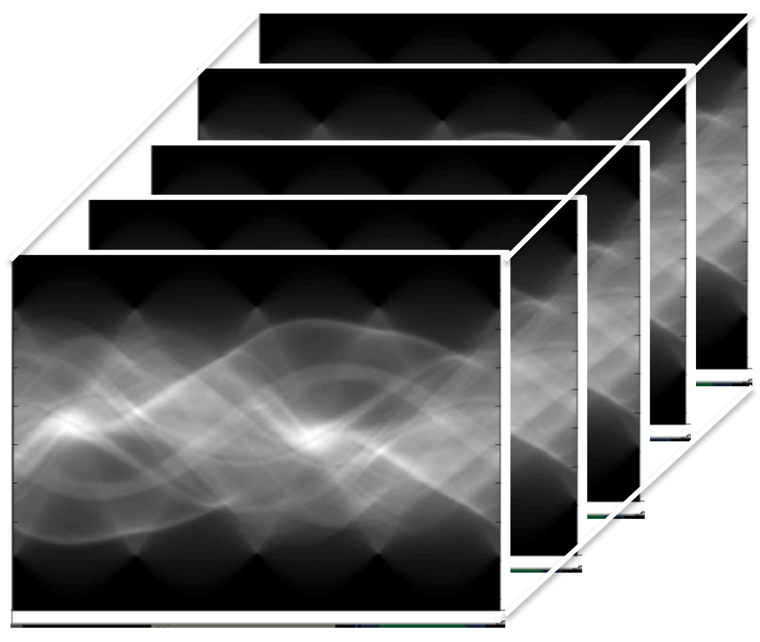
DUAL DMA ENGINE CASE

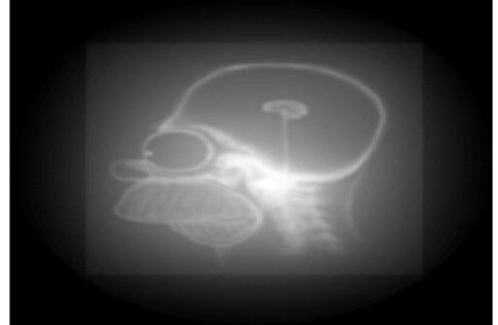
- Dual DMA engines allow simultaneous bidirectional IO.



- Possible Improvement with dual DMA Engines
  - Baseline with one queue = 3*5 = 15T
  - Overlap Case = 7T
- Potential Performance Benefit ~ 50%

EXAMPLE: IMAGE RECONSTRUCTION





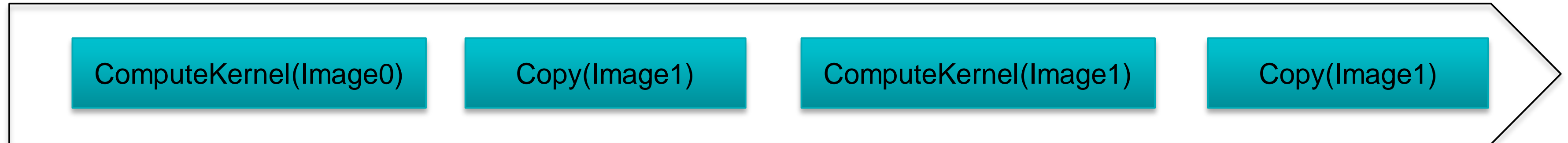
Reconstructed OP Image

- Filtered Back-projection Application
- Multiple sinogram images processed to build a reconstructed image
- Images continuously fed in from scanner to iteratively improve resultant output
- Streaming style data flow

Image Source:
hem.bredband.net/luciadbb/Reconstruction_presentation.pdf

WITHOUT ASYNCHRONOUS I/O

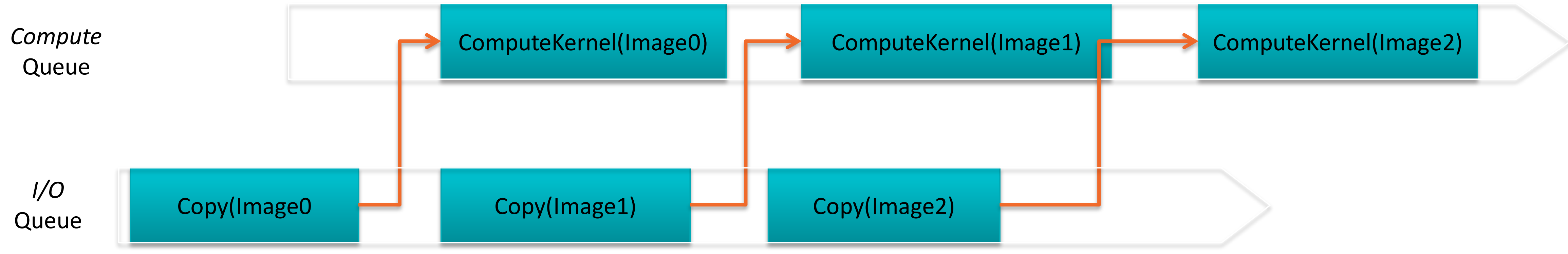
- Single Command Queue: N images = N(tcompute + transfer)



- Inefficient for medical imaging applications like reconstruction where large numbers of input images are used in a streaming fashion to incrementally reconstruct an image.
- Performance improvement by asynchronous IO is better than previously discussed case since no IO from device to host after each kernel call. This would reduce total IO time per kernel by ½
  - Total time per image = Ti + Tc = 2T
  - Overlapped Time = T (if Ti = Tc) shows ~ 50% improvement scope

EVENTS FOR ASYNCHRONOUS I/O

- Two command queues created on the same device
  - Different from asymptotic analysis case of dividing computation between queues
  - In this case we use different queues for I/O and compute
  - We have no output data moving from Host to device for each image, so using separate command queues will also allow for latency hiding



PINNED OR NON-PINNED MEMORY

- OpenCL runtime use pinned memory for DMA transfer
  - <= 32 KB: for transfers from the host to device, the data is copied by the CPU to a runtime pinned host memory buffer, then DMA to device
  - > 32 KB and <= 16 MB: the host memory physical pages are pinned, then DMA
  - > 16MB: OpenCL runtime pins host memory in stages of 16MB block, then DMA
- Pin/unpin operation has heavy overhead
  - Try to use pre-pinned memory to reduce the overhead
  - The performance is even worse with non-256B alignment memory
- Pre-pinned memory
  - Use CL_MEM_ALLOC_HOST_PTR flag with clCreateBuffer() to generate pre-pinned memory buffer
    - Then use clEnqueueCopyBuffer() to transfer data
  - Or Use clEnqueueMapBuffer() to get a host pointer to the pre-pinned memory object
    - Then use clEnqueueWriteBuffer() to transfer data

AGENDA

- OpenCL system performance
  - CPU/GPU data movement
  - OpenCL runtime overhead
- APU architecture and OpenCL optimization
- HSA and OpenCL optimization

RE-THINKING CPU+dGPU

Current CPUs and GPUs have been designed as separate processing elements and do not work together efficiently...

Serial/Task-Parallel Workloads

Graphics Workloads

Other Highly Parallel Workloads

We need consideration of overall system efficiency

Today’s efficiency problems result from the way computers have evolved

Typically platform builders create innovative new hardware and offer an API for software to access it

That tired thinking has only ever had niche success!

APU ARCHITECTURE

APP Accelerated Software Applications

Accelerated Processing Unit with
Latency Compute Unit (LCU)
Throughput Compute Unit (TCU)

Graphics Workloads

Data Parallel Workloads

Serial and Task Parallel Workloads

DDR3 Controller
UNB
Channel
GMC
AMD HD Media Accelerator
L2 Cache
L2 Cache
GPU
Dual Core x86 Module
Dual Core x86 Module
PCIe
PCIe
PCIe
Display PLL
DP / HDMI
Display Controller

KEY ARCHITECTURE EVOLUTION FROM dGPU

- Host and Device share the same physical memory
  - Both Host (CPU) and Device (GPU) has their own TLB
- On the APU, one of the key parts of the system is the data path between the GPU and memory
  - Provides low latency access for CPU cores (optimized around caches)
    - Random access, branchy, single threaded, scalar code
  - Provides high throughput access for GPU cores (optimized around latency hiding)
    - Streaming, vectorized, massively multithreaded, data-intensive cod
- Two new on-chip buses are introduced
  - AMD Fusion Compute Link (ONION)
    - This bus is used by the GPU when it needs to snoop the CPU cache, so is a coherent bus
    - This is used for cacheable system memory
  - Radeon Memory Bus (GARLIC):
    - This bus is directly connected to memory and can saturate memory bandwidth, so is a non coherent bus
    - This is used for local memory and USWC (uncached speculative write combine)

APU PRIOR TO HSA

- Memory visibility in APU system
  - Both CPU and GPU have their own set of page tables and TLB
  - Both CPU and GPU can directly access the each other’s memory
  - The memory is generally not coherent
  - The GPU can probe the CPU cache …
  - … but the CPU relies on the driver for synchronization (map/unmap, lock/unlock flush GPU caches)

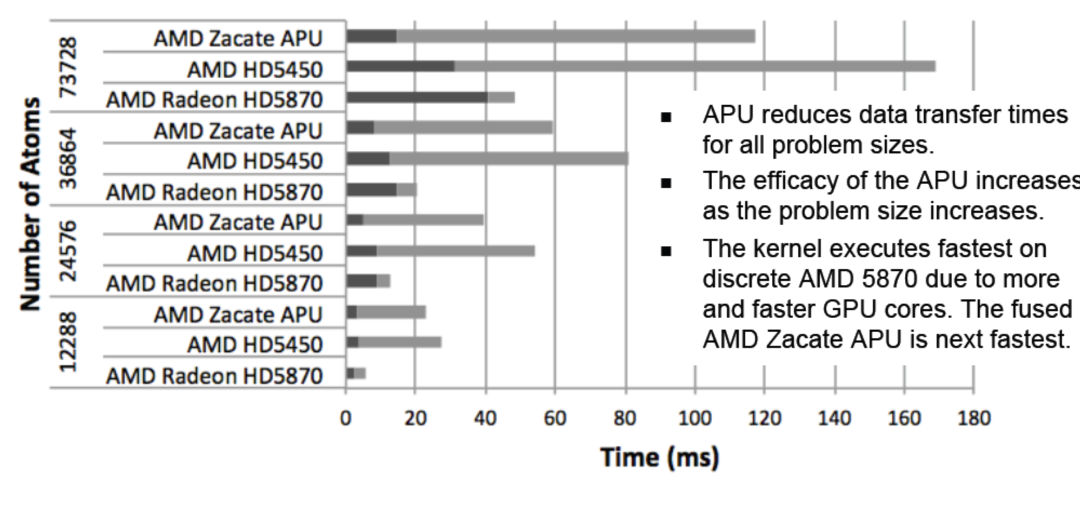
- The current programming model is a direct consequence
  - CPU access will page fault on a single access, and the OS will page in/out on demand
  - GPU access is known upfront, and the driver or OS will page in/out on scheduling

- However, CPU/GPU can perform “zero-copy”

ZERO-COPY

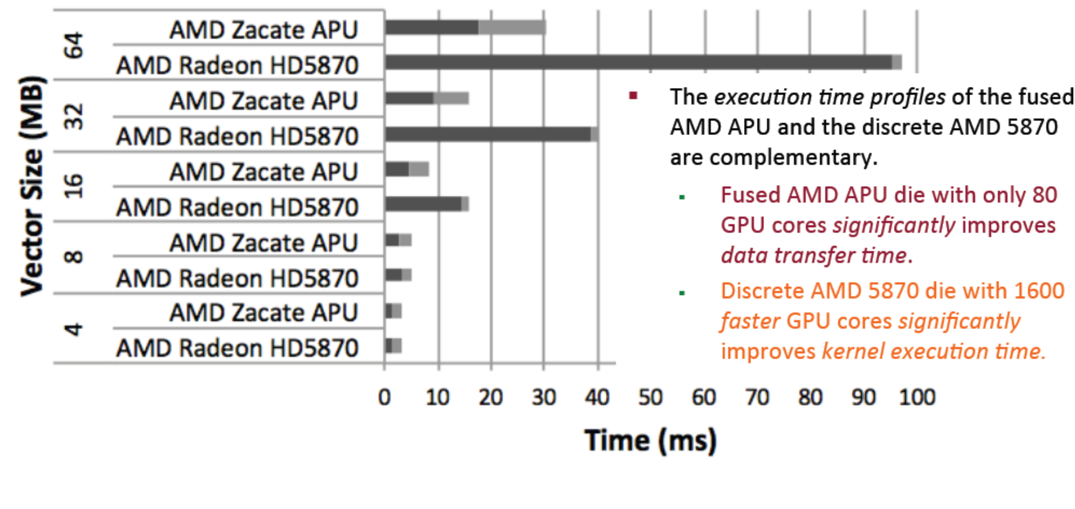
- Many different meanings
  - A kernel access system memory directly for either read or write
  - A DMA transfer access system memory directly without copying into USWC
  - The CPU directly writes into local memory without doing any DMA
- OpenCL offers several mechanisms to effectively reduce extra copying
  - On APU , this matters even more than on discrete because bandwidth is shared
  - OpenGL has some driver optimizations and some proprietary extensions

Performance: Molecular Dynamics (N-Body)



Compute-bound

Performance: Reduction (Dense Linear Algebra)



I/O-bound

DIFFERENT DATA LOCATION AND PATH

- clCreateBuffer will give OpenCL a memory object
  - All GPU Kernel memory access reply on this object
- clEnqueuMapBuffer to access the memory

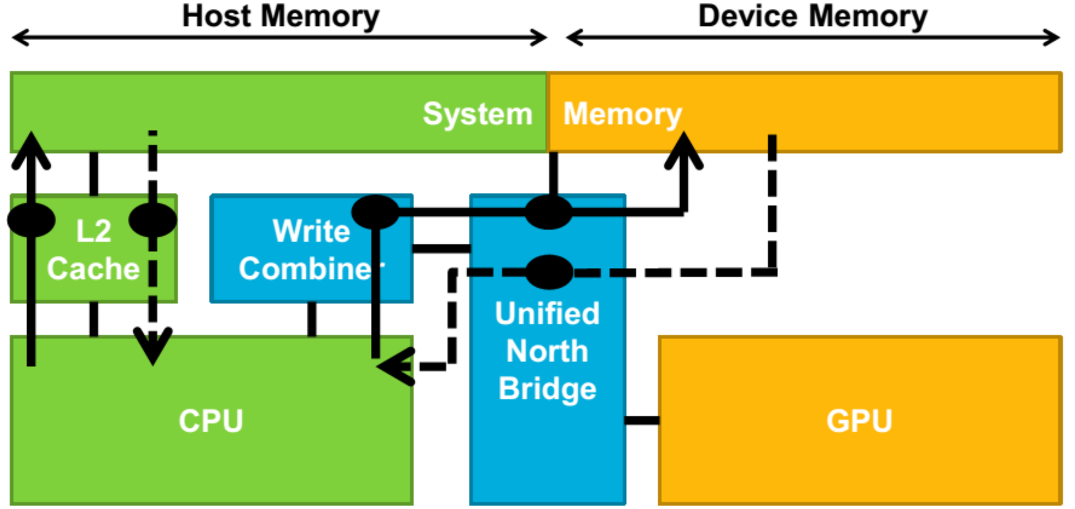
- Data transfer is the key point for system level performance optimization
  - Different flags to clCreateBuffer will result in different heap location of the memory object
  - Different heap location results in different transfer speed
  - Different memory object type results in different OpenCL runtime overhead
  - On APU, memory object decides zero-copy or not

- Data location
  - Host memory
    - Cacheable memory
    - Uncached memory
  - Device memory

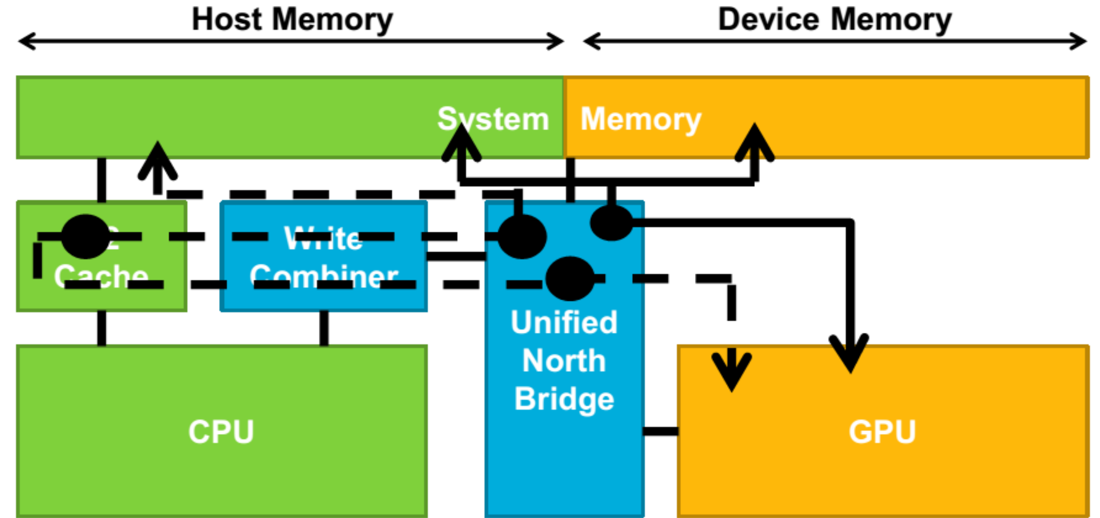
- Data path
  - CPU access to device memory
  - ~~CPU access to uncached memory~~
  - CPU access to cacheable memory
  - GPU access to device memory
  - GPU access to uncached memory
  - GPU access to cacheable memory

DIFFERENT DATA LOCATION AND PATH

- Data path
  - CPU <-> Host memory
    - Write/Read: through cache
  - CPU <-> device memory (zero-copy)
    - Write: WC
    - Read: uncached read
  - CPU <-> device memory (non zero-copy)
    - DMA



- Data path
  - GPU <-> cacheable host memory
    - Write/Read: Onion bus
  - GPU <-> uncached host memory
    - Write/Read: Garlic bus
  - GPU <-> device memory
    - Write/Read: Garlic bus



THE FIRST GLANCE OF MEMORY OBJECT PERFORMANCE

LLANO APU

- Performance (may vary based on system/driver)

	LOCAL	UNCACHED	CACHEABLE
GPU READ	17 GB/s	6-12 GB/s	4.5 GB/s
GPU WRITE	12 GB/s	6-12 GB/s	5.5 GB/s
CPU READ	< 1GB/s	< 1GB/s	8-13 GB/s
CPU WRITE	8 GB/s	8-13 GB/s	8-13 GB/s

CPU COPY TO DEVICE MEMORY

DMA

- DMA is performed via PCI-E bus
  - For PCI-E 2.0 x 16 lane, theoretical peak bandwidth is 8GB/s in each direction
  - For PCI-E 3.0, theoretical peak bandwidth is 16GB/s in each direction
  - The effective bandwidth depends on the data size, usually far away from the theoretical performance

- Two buffer existing at both Host and Device side
  - Zero-copy has only one buffer existing on either Host or Device side
  - Explicit data copy between the sides

- Double buffering should be used to hide the time of data movement
  - Very common on CPU + dGPU platform

CPU ACCESS TO DEVICE MEMORY

ZERO COPY

- CPU writes to device memory
  - On Llano, this can peak at 8GB/s
  - The data first goes through the WC buffers on the CPU, then goes to the GPU core in order to get physical address
- CPU reads from device memory
  - Very slow!
  - Access are uncached
  - Only one single outstanding read is supported
- Sample code
  - Create the buffer with the CL_MEM_USE_PERSISTENT_MEM_AMD flag

Create buffer

```
clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_PERSISTENT_MEM_AMD, bufSize, 0, &error);
```

Access buffer

```
clEnqueueMapBuffer(cmd_queue, buffer, CL_TRUE, CL_MAP_WRITE, 0, bufSize, 0, NULL, NULL, &error)
```

CPU ACCESS TO CACHEABLE HOST MEMORY

ZERO COPY

- CPU accesses to cacheable memory
  - This is the typical case in C++ code (no difference to discrete)
  - Single threaded performance: ~8GB/s for either read or write
  - Multi-threaded performance: ~13GB/s for either read or writeCPU reads from device memory
- The memory can be accessed by the GPU:
  - Pages need to be made resident by the OS, and locked to prevent paging
  - Physical pages need to be programmed into the GPU HW virtual memory page tables
  - Implications:
    - The two operations are done by the device driver (compute/graphics)
    - They take time, so should be done atinitialization time if possible
    - There is a limit to how much cacheable memory can be accessed, because itis removed from normal OS usage

Create buffer

```
clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PRT, bufSize, 0, &error);
```

Access buffer

```
clEnqueueMapBuffer(cmd_queue, buffer, CL_TRUE, CL_MAP_WRITE, 0, bufSize, 0, NULL, NULL, &error)
```

GPU ACCESS TO DEVICE MEMORY

ZERO COPY

- GPU reads from device memory
  - This is the optimal path to memory:
    - Radeon Memory Bus (GARLIC) avoids any cache snooping
    - Memory is interleaved to increase throughput efficiency
  - Kernels and shaders can saturate dram bandwidth (measured at ~17GB/s)
- GPU writes to device memory are similar
  - Kernels and shaders can saturate dram bandwidth (measured at ~13 GB/s)

Create buffer

```
clCreateBuffer(context, CL_MEM_READ_WRITE, bufSize, 0, &error);
```

Access buffer

Directly use in the Kernel

GPU ACCESS UNCACHED HOST MEMORY

- GPU accesses to uncached memory
  - This uses the Radeon Memory Bus (GARLIC)
  - Memory does not have the same interleaving granularity as local memory
  - So slightly lower performance than local memory, but faster than cacheable memory
  - Reads can saturate dram bandwidth (measured at 12 GB/s)
  - Writes are similarly fast but …
    - Usually avoided, however, since CPU readsare really slow from uncached space

Create buffer

clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_ALLOC_HOST_PRT, bufSize, 0, &error);

Access buffer

Directly use in the Kernel

GPU ACCESS CACHABLE HOST MEMORY

- GPU accesses to cacheable memory
  - This can be used directly by a kernel or for data upload to the GPU
  - Uses the AMD Fusion Compute Link (ONION), so snoops the cache
  - Reads measured at ~4.5 GB/s and writes at ~5.5 GB/s
  - Often used for sharing data with the CPU
  - Or CPU share data among multiple GPU devices

Create buffer

```
clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PRT, bufSize, 0, &error);
```

Access buffer

Directly use in the Kernel

OTHER EXPERIMENTAL DATA

	CPU R	CPU W	GPU Shader R	GPU Shader W	GPU DMA R	GPU DMA W
Host Memory	10 - 20	10 - 20	9 - 10	2.5	11 - 12	11 - 12
GPU Memory	.01	9 - 10	230	120 -150	n/a	n/a

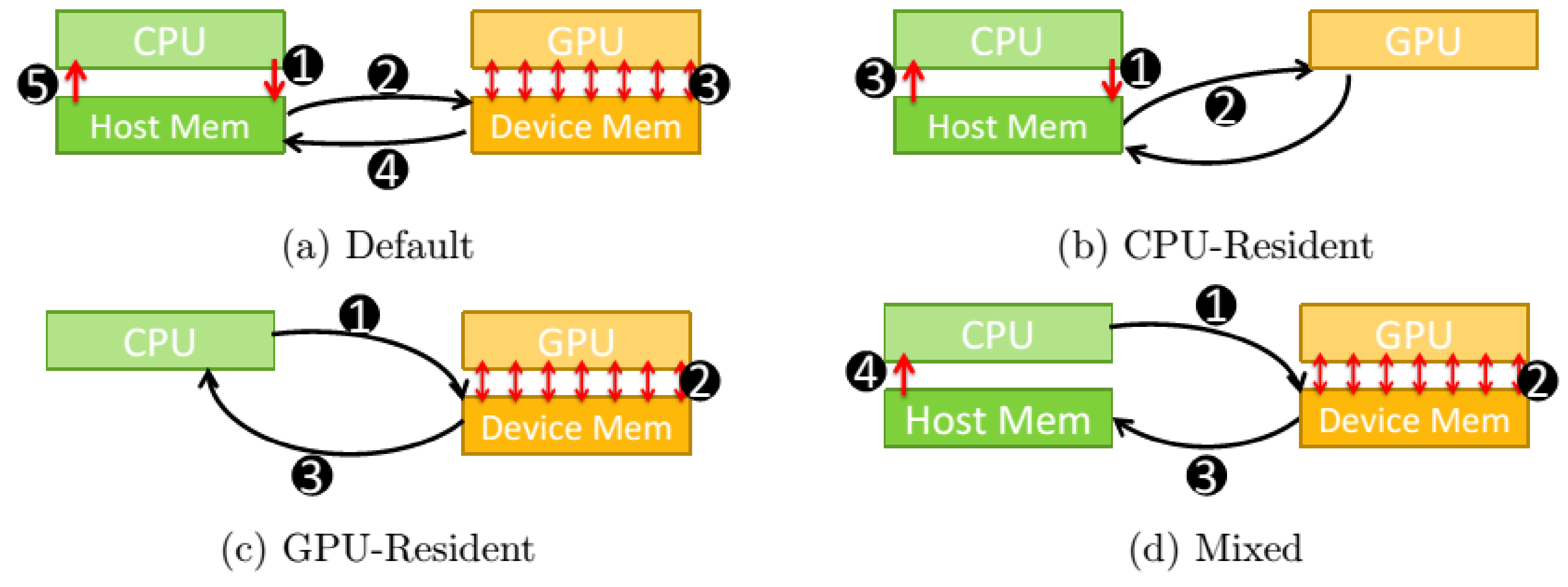
Quoted from *AMD Accelerated Parallel Processing OpenCL™ Programming Guide*

Transfer Type	Zacate	Llano	Discrete
Host Buffer → Device Buffer	1.15 GB/s	2.61 GB/s	1.25 GB/s
Host Buffer ← Device Buffer	1.18 GB/s	3.17 GB/s	1.39 GB/s
CPU ← Host Buffer (Read)	0.75 GB/s	5.67 GB/s	5.64 GB/s
CPU → Host Buffer (Write)	1.75 GB/s	5.46 GB/s	5.44 GB/s
GPU ← Host Buffer (Read)	6.49 GB/s	16.26 GB/s	1.46 GB/s
GPU → Host Buffer (Write)	3.66 GB/s	4.96 GB/s	1.28 GB/s
CPU ← Device Buffer (Read)	0.01 GB/s	0.01 GB/s	0.01 GB/s
CPU → Device Buffer (Write)	1.98 GB/s	7.49 GB/s	1.52 GB/s
GPU ← Device Buffer (Read)	6.75 GB/s	17.54 GB/s	128.74 GB/s
GPU → Device Buffer (Write)	4.78 GB/s	14.31 GB/s	98.60 GB/s

Quoted from *Characterization and Exploitation of GPU Memory Systems, Kenneth S. Lee*

BEST DATA LOCATION AND PATH

- Allocate buffer at Device side, and access it using garlic bus
  - This gives you best performance for GPU <-> GPU, CPU -> GPU
  - For CPU <- GPU, it's very slow, so use clEnqueueCopyBuffer to get a copy to read from CPU



OPENCL PERFORMANCE ON APU

- Zero-copy benefits
  - Avoid data copy, directly use
  - Faster bus
  - Easy to programming
  - Use clEnqueueMapBuffer() can get direct benefits on both APU and dGPU

- Zero-copy also works for AMD dGPU
  - Limited by PCI-E bandwidth

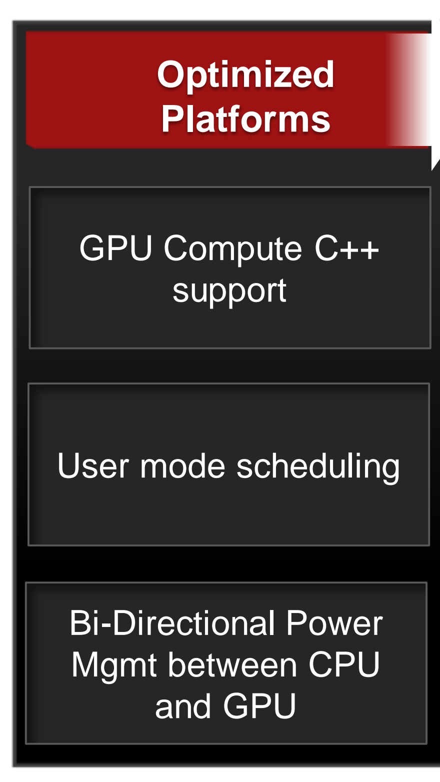
AGENDA

- OpenCL system performance
  - CPU/GPU data movement
  - OpenCL runtime overhead
- APU architecture and OpenCL optimization
- HSA and OpenCL optimization

WE'RE IN THE "ARCHITECTURAL INTEGRATION" STAGE

Physical Integration	Optimized Platforms	Architectural Integration	System Integration
Integrate CPU & GPU in silicon	GPU Compute C++ support	Unified Address Space for CPU and GPU	GPU compute context switch
Unified Memory Controller	User mode scheduling	GPU uses pageable system memory via CPU pointers	GPU graphics pre-emption
Common Manufacturing Technology	Bi-Directional Power Mgmt between CPU and GPU	Fully coherent memory between CPU & GPU	Quality of Service

HSA COMPLIANT FEATURES

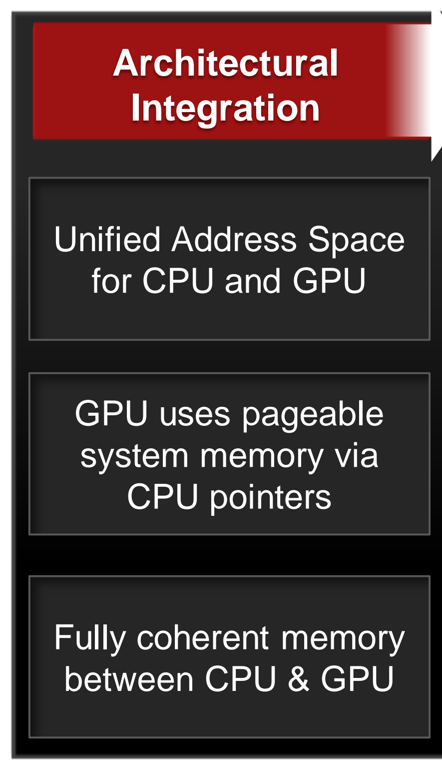


Support OpenCL C++ directions and Microsoft's upcoming C++ AMP language. This eases programming of both CPU and GPU working together to process parallel workloads.

Drastically reduces the time to dispatch work, requiring no OS kernel transitions or services, minimizing software driver overhead

Enables "power sloshing" where CPU and GPU are able to dynamically lower or raise their power and performance, depending on the activity and which one is more suited to the task at hand.

HSA COMPLIANT FEATURES

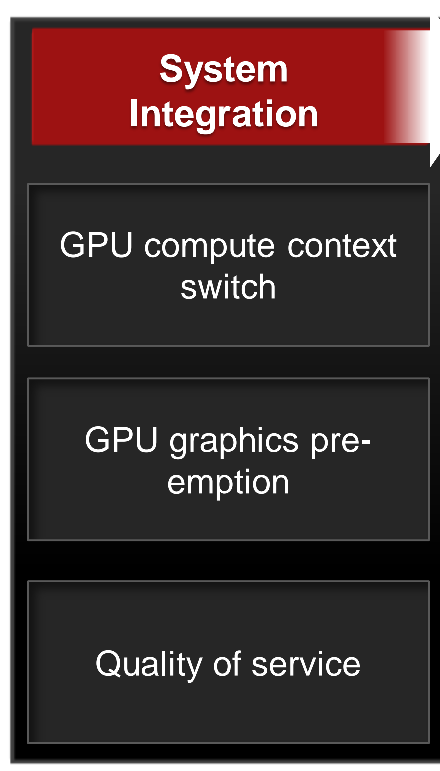


The unified address space provides ease of programming for developers to create applications. For HSA platforms, a pointer is really a pointer and does not require separate memory pointers for CPU and GPU.

The GPU can take advantage of the CPU virtual address space. With pageable system memory, the GPU can reference the data directly in the CPU domain. In prior architectures, data had to be copied between the two spaces or page-locked prior to use. And, NO GPU memory size limitation!

Allows for data to be cached by both the CPU and the GPU, and referenced by either. In all previous generations, GPU caches had to be flushed at command buffer boundaries prior to CPU access. And unlike discrete GPUs, the CPU and GPU in an APU share a high speed coherent bus.

FULL HSA FEATURES



GPU tasks can be context switched, making the GPU a multi-tasker. Context switching means faster application, graphics and compute interoperation. Users get a snappier, more interactive experience.

As more applications enjoy the performance and features of the GPU, it is important that interactivity of the system is good. This means low latency access to the GPU from any process.

With context switching and pre-emption, time criticality is added to the tasks assigned to the processors. Direct access to the hardware for multi-users or multiple applications are either prioritized or equalized.

KAVERI FROM OPENCL PROGRAMMER'S PERSPECTIVE

- Fundamental architectural changes!
  - Bring "HSA" into reality, giving you infinite imaginary space
    - First time, OpenCL programmer can take advantage of entire pageable memory space
    - Access data directly without moving it
    - Supporting more data structure and programming pattern
    - Better performance with less programming efforts
- First generation APU with "GCN" GPU architecture
  - Optimized for generation purpose computing workloads
  - Increasing GPU horsepower
- Essential architecture for all AMD platform, a "OpenCL style"
  - Covering from embedded, mobile, desktop to server
- Taking performance/watt as the first priority

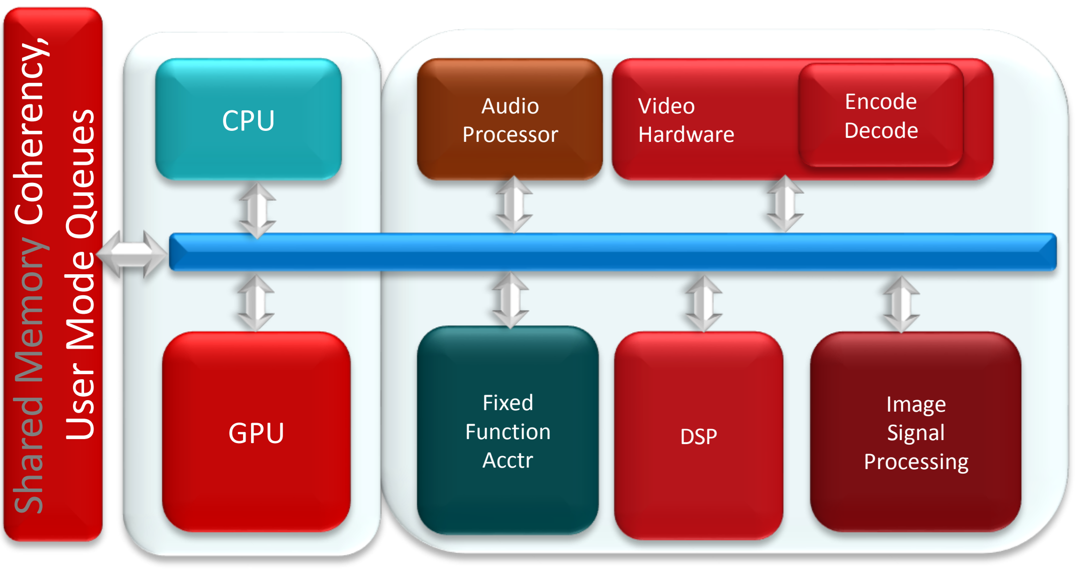
INTRODUCING HSA

An ***intelligent computing architecture*** that enables CPU, GPU and other processors to work in ***harmony*** on a single piece of silicon by ***seamlessly*** moving the right tasks to the best suited processing element

GPU
CPU
hUMA (MEMORY)
AMD
PARALLEL WORKLOADS
SERIAL WORKLOADS
HSA
APU
ACCELERATED PROCESSING UNIT

HSA ARCHITECTURE

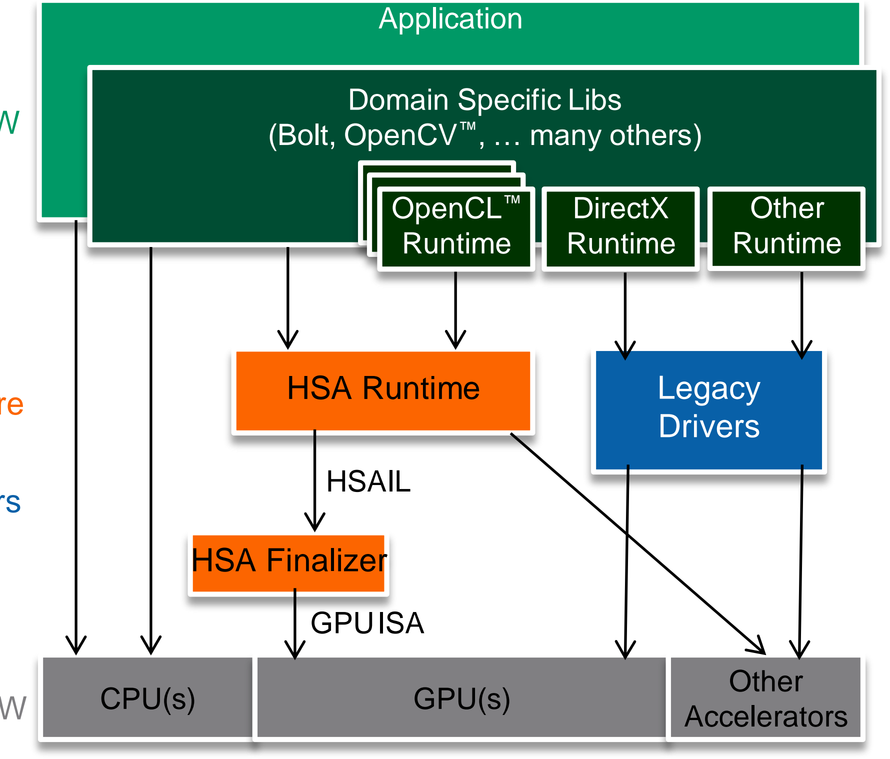
- Full programming language support
- User mode queueing
- Heterogeneous unified memory access (hUMA)
- Pageable memory
- Bibirectional coherency
- Compute context swtith and preemption



HSA SOLUTION STACK

Computing hardware
A software compilation stack
A user - space runtime system
Kernel - space system components

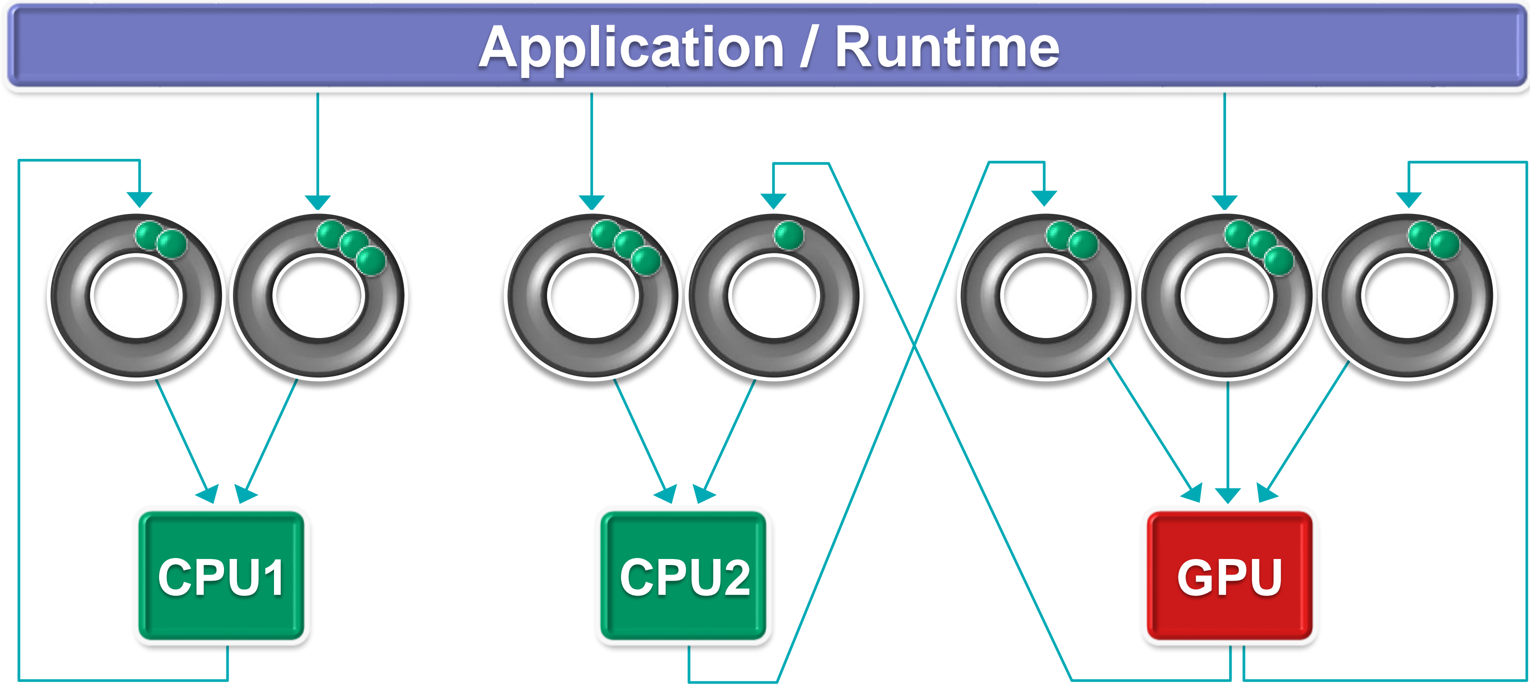
- Make GPU easily accessible
  - Support mainstream languages, expandable to domain specific languages
  - Complete GPU tool-chain, Programming & debugging & profiling like CPU does
- Make compute offload efficient
  - Direct path to GPU (avoid Graphics overhead)
  - Eliminate memory copy, Low-latency dispatch
- Make it ubiquitous
  - Drive HSA as a standard through HSA Foundation
  - Open Source key components



GET BENEFITS FROM HSA-FEATURED KAVERI

KERNEL ENQUEUE

- Enables the HSA CU to address new workload classes, beyond the classic static gridded algorithms.



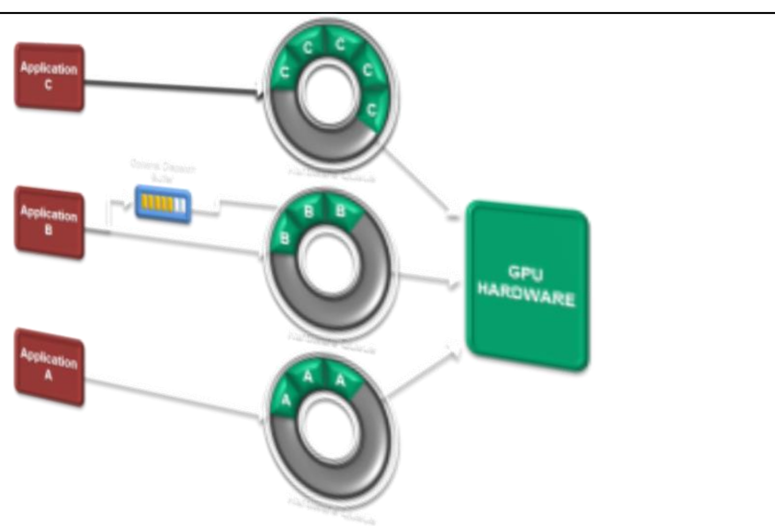
BENEFITS FROM HSA-FEATURED KAVERI

ACCELERATED DISPATCH LATENCY

*How compute dispatch operates today in the **driver model***

---

*How compute dispatch improves tomorrow **under HSA***



KAVERI-ORIENTED ALGOS
GENERAL IDEAS

- Those algo. who need large data set
- Those algo. who has heavy CPU/GPU data movement
- Those algo. who has less Kernel time with larger CPU/GPU communication time
- Those software who has complex and settled data structures
- Those software who want more programming model
  - For example, a producer/consumer pattern

THANKS!