Events Timing and Profiling

A Collaboration Between
David Kaeli, Northeastern University
Benedict R. Gaster, AMD

Instructor Notes

- Discusses synchronization, timing and profiling in OpenCL
- Coarse grain synchronization covered which discusses synchronizing on a command queue granularity
  - Discuss different types of command queues (in order and out of order)
- Fine grained synchronization which covers synchronization at a per function call granularity using OpenCL events
- Improved event handling capabilities like user defined events have been added in the OpenCL 1.1 specification
- Synchronization across multiple function calls and scheduling actions on the command queue discussed using wait lists
- The main applications of OpenCL events have been discussed here including timing, profiling and using events for managing asynchronous IO with a device
  - Potential performance benefits with asynchronous IO explained using a asymptotic calculation and a real world medical imaging application

Topics

- OpenCL command queues
- Events and synchronization
- OpenCL 1.1 and event callbacks
- Using OpenCL events for timing and profiling
- Using OpenCL events for asynchronous host-device communication with image reconstruction example

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

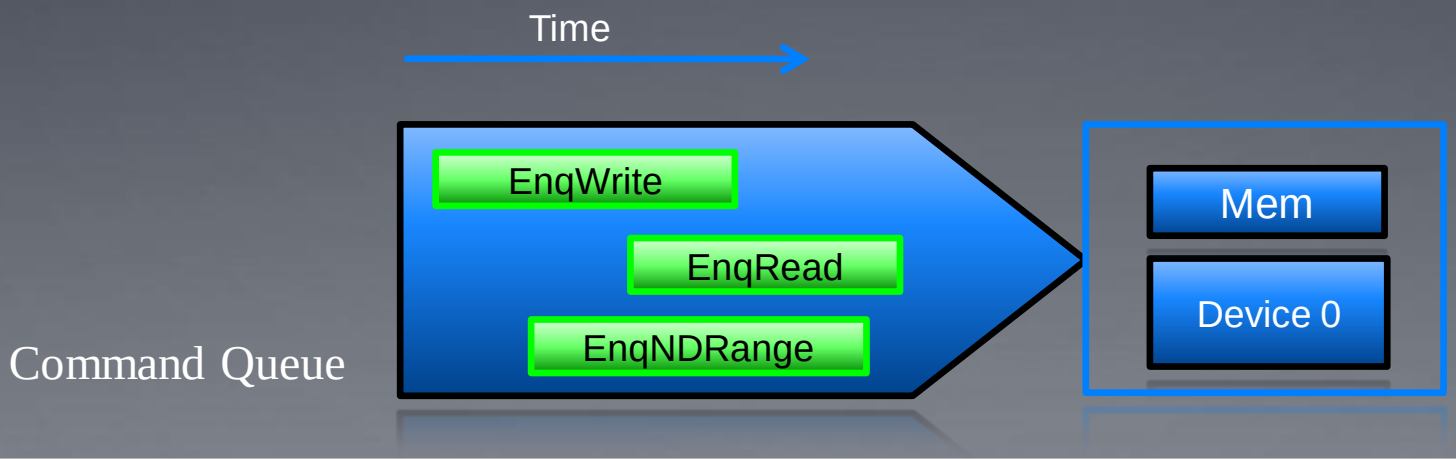
In-Order Execution

- 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

Commands To Submit
clEnqueueWriteBuffer (queue , d_ip, CL_TRUE, 0, mem_size, (void *)ip, 0, NULL, NULL)
clEnqueueNDRangeKernel (queue, kernel0, 2,0, global_work_size, local_work_size, 0,NULL,NULL)
clEnqueueReadBuffer(context, d_op, CL_TRUE, 0, mem_size, (void *) op, NULL, NULL,NULL);

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 1.1

- OpenCL 1.1 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

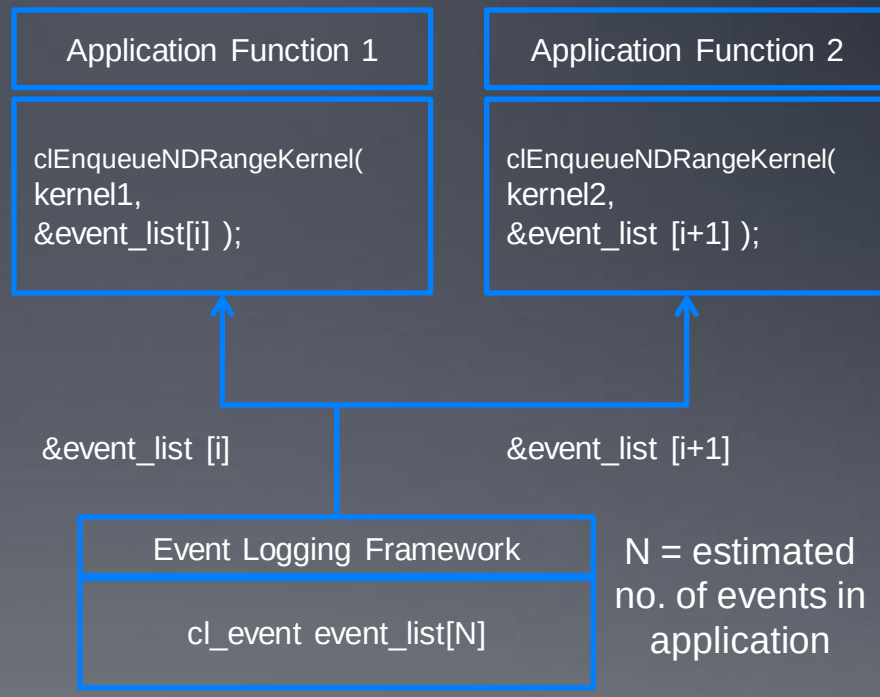
```
clGetEventProfilingInfo( event_time,
CL_PROFILING_COMMAND_START,
sizeof(cl_ulong), &starttime, NULL);
```

```
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

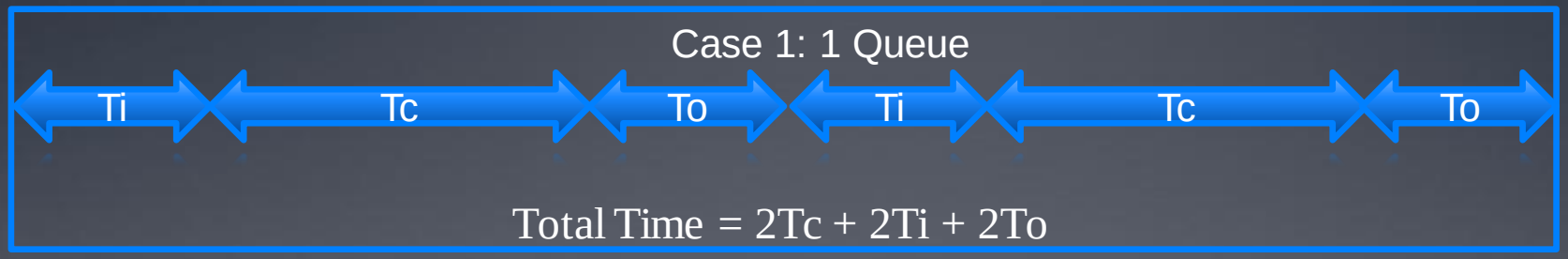
- A heterogeneous application can have multiple kernels and a large amount of host device IO
- Questions that can be answered by profiling using OpenCL events
  - We need to know which kernel to optimize when multiple kernels take similar time ?
    - Small kernels that may be called multiple times vs. 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

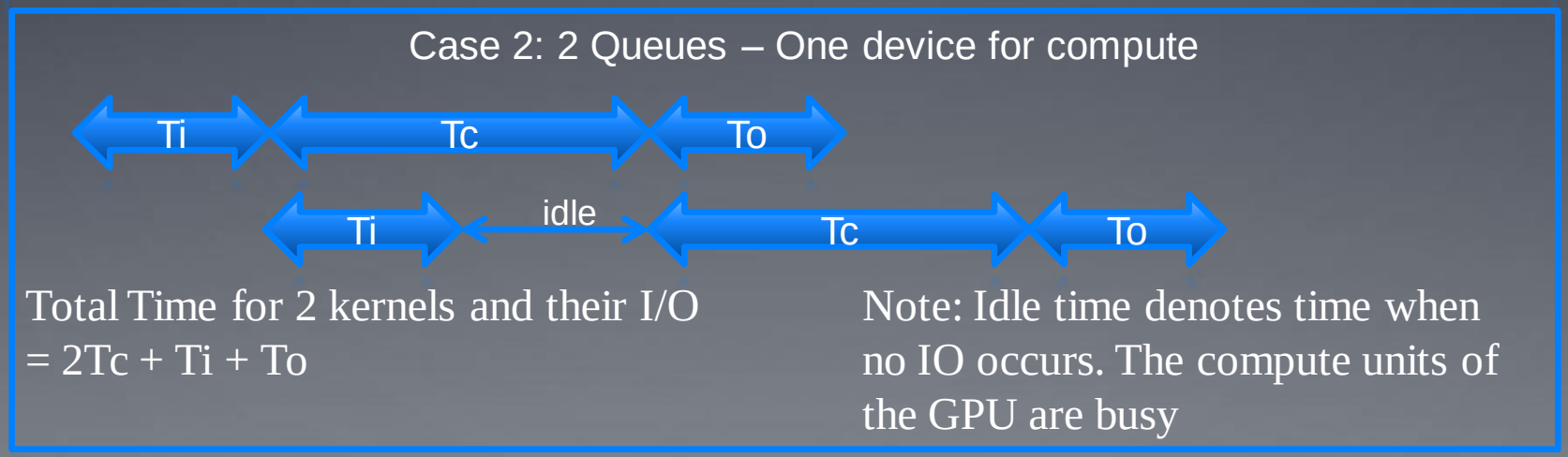
Asynchronous IO

- 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

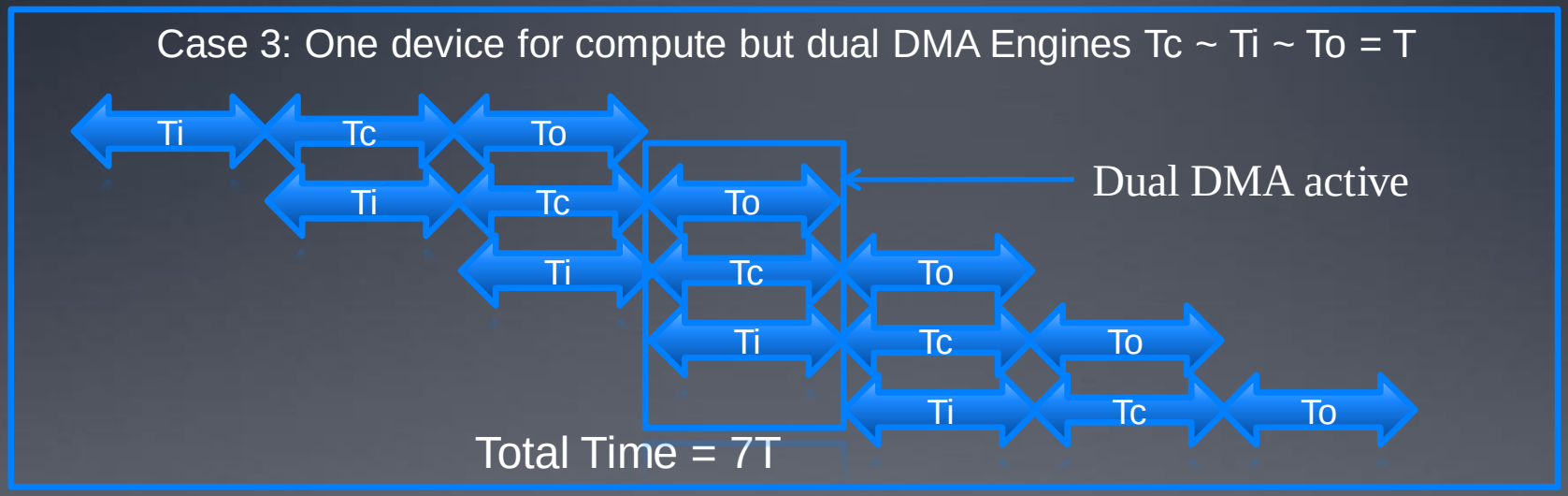
- Time with 1 Queue = 2Tc + 2Ti + 2To
  - No asynchronous behavior
- Time with 2 Queues = 2Tc + Ti + To
  - Overlap computation and communication

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

- 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

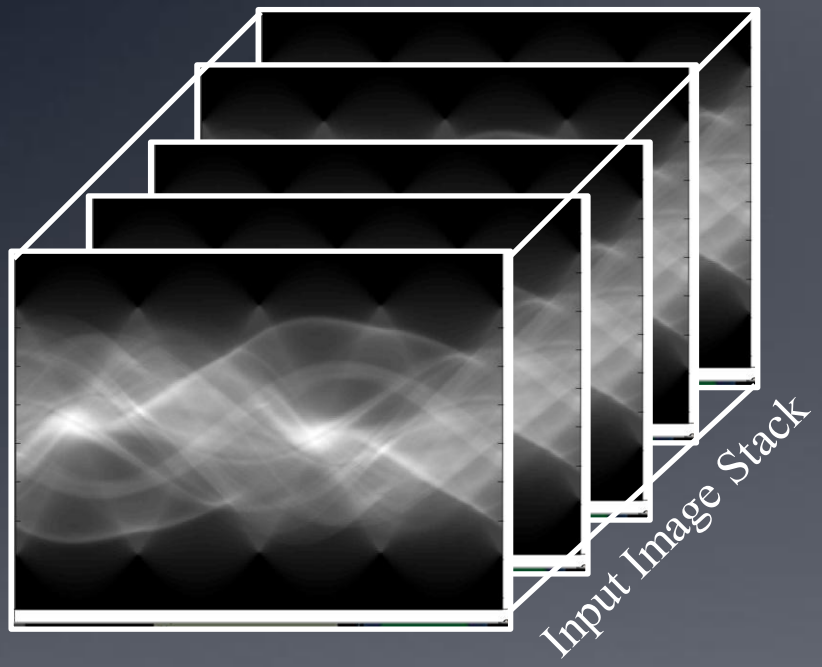
Dual DMA Engine Case

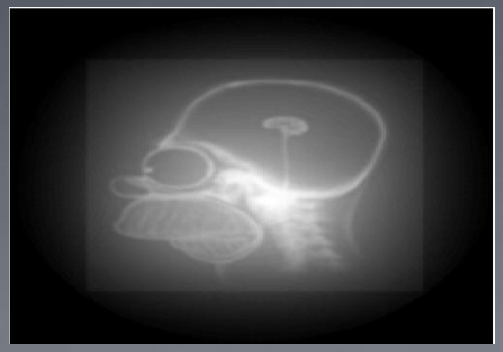
- In the Nvidia Fermi GPUs, dual DMA engines allow simultaneous bidirectional IO.



- 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

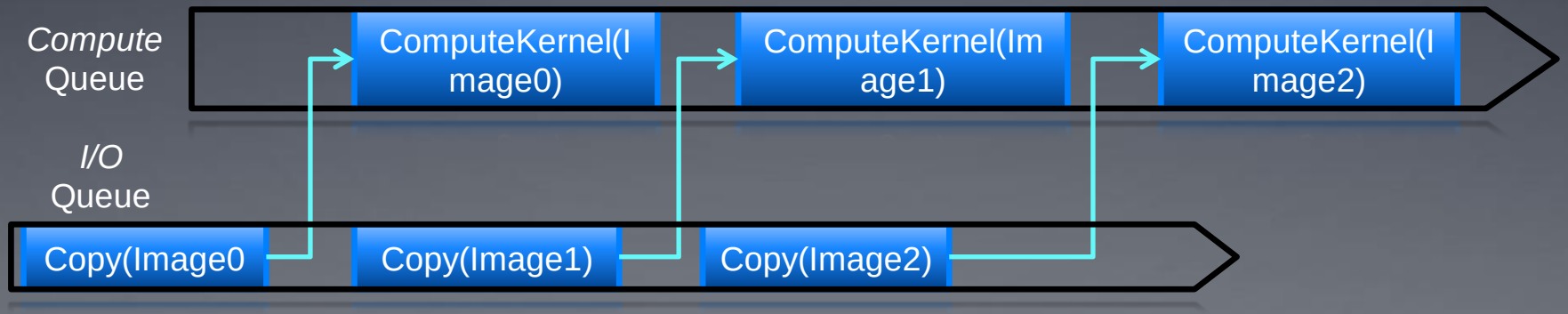
- Single Command Queue: N images = N(tcompute + transfer)

- 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 IO 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



Backprojection

- The Backprojection kernel isn't relevant to this discussion
  - We simply wish to show how kernels and buffer writes can be enqueued to different queues to overlap IO and computation
- More optimizations possible like buffering strategies and memory management to avoid declaring N buffers on device

```
//N is the number of images we wish to use
cl_event event_list[N];
//Start First Copy to device
clEnqueueWriteBuffer (queue1 , d_ip[0], CL_FALSE,
                      (void *)ip, 0, NULL, &event_list[0] ) ;

for ( int i = 1 ;  i <N ; i++)
{       //Wait till IO is finished  by specifying a wait list of one element which denotes the previous I/O
        clEnqueueNDRange(queue0, Kernel, …
                      1,&event_list[i-1], NULL); //clEnqueueND will return asynchronously and begin IO

        //Enque a Non Blocking Write using other command queue
        clEnqueueWriteBuffer (queue1
                , d_ip [i-1], CL_FALSE,
                (void *)ip, 0, NULL, &event_list[i])
}
```

Summary

- OpenCL events allow us to use the execution model and synchronization to benefit application performance
  - Use command queue synchronization constructs for coarse grained control
  - Use events for fine grained control over an application
- OpenCL 1.1 allows more complicated event handling and adds callbacks. OpenCL 1.1 also provides for events that can be triggered by the user
- Specification provides events to understand an application performance.
- Per kernel analysis requires vendor specific tools
  - Nvidia OpenCL Profiler
  - AMD Accelerated Parallel Processing (APP) Profiler and Kernel Analyzer