北京大学高能效计算与应用中心

Center for Energy-efficient Computing and Applications

Introduction to Parallel & Distributed Computing

OpenCL: optimizations

Lecture 13, Spring 2014

Instructor: 罗国杰

gluo@pku.edu.cn

Outline

- Optimizations
  - Thread mapping
    - Choosing a proper mapping
    - Optimizing with local memory
  - Device occupancy
  - Vectorization
- Timing
- Debugging

Optimizations

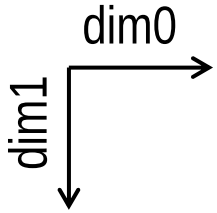
- The performance impact of mapping threads to data on the GPU is subtle but extremely important
- The number of threads that are active on the GPU can also play a large part in achieving good performance, and so the subtleties of GPU occupancy are discussed
- Vectorization is particularly important for AMD GPUs and is briefly discussed as well

Thread Mapping

- **Thread mapping determines which threads will access which data**
  - **Proper mappings can align with hardware and provide large performance benefits**
  - **Improper mappings can be disastrous to performance**
- **The paper *Static Memory Access Pattern Analysis on a Massively Parallel GPU* by Jang, et. al focuses on the task of effectively mapping threads to the data access patterns of an algorithm**

Thread Mapping

- **By using different mappings, the same thread can be assigned to access different data elements**
  - **The examples below show three different possible mappings of threads to data (assuming the thread id is used to access an element)**



Mapping

```
int tid =
get_global_id(1) *
get_global_size(0) +
get_global_id(0);
```

```
int tid =
get_global_id(0) *
get_global_size(1) +
get_global_id(1);
```

```
int group_size =
get_local_size(0) *
get_local_size(1);

int tid =
get_group_id(1) *
get_num_groups(0) *
group_size +
get_group_id(0) *
group_size +
get_local_id(1) *
get_local_size(0) +
get_local_id(0)
```

Thread IDs

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

*assuming 2x2 groups

Thread Mapping

- Consider a serial matrix multiplication algorithm

```
for (i1=0; i1 < M; i1++)
    for (i2=0; i2< N; i2++)
        for (i3=0; i3 < P; i3++)
            C[i1][i2] += A[i1][i3]*B[i3][i2];
```

- This algorithm is suited for output data decomposition
  - We will create *NM* threads
    - Effectively removing the outer two loops
  - Each thread will perform *P* calculations
    - The inner loop will remain as part of the kernel
- Should the index space be M×N or N×M?

Thread Mapping

- **Thread mapping 1: with an M×N index space, the kernel would be:**

```
int tx = get_global_id(0);
int ty = get_global_id(1);
for(i3=0; i3<P; i3++)
    C[tx][ty] += A[tx][i3]*B[i3][ty];
```

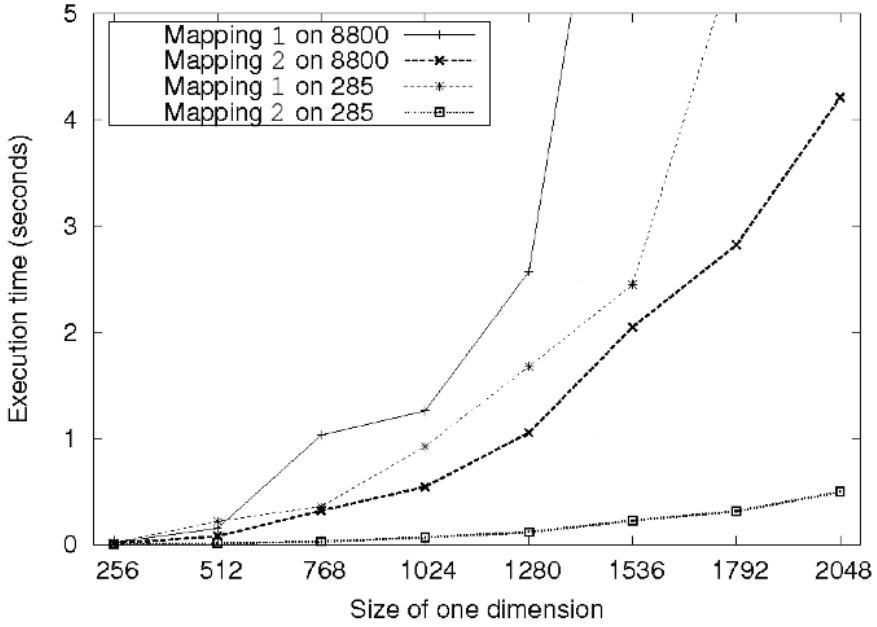
- **Thread mapping 2: with an N×M index space, the kernel would be:**

```
int tx = get_global_id (0);
int ty = get_global_id (1);
for(i3=0; i3<P; i3++)
    C[ty][tx] += A[ty][i3]*B[i3][tx];
```

- **Both mappings produce functionally equivalent versions of the program**

Thread Mapping

- This figure shows the execution of the two thread mappings on NVIDIA GeForce 285 and 8800 GPUs



- Notice that mapping 2 is far superior in performance for both GPUs

Thread Mapping

- **The discrepancy in execution times between the mappings is due to data accesses on the global memory bus**
  - **Assuming row-major data, data in a row (i.e., elements in adjacent columns) are stored sequentially in memory**
  - **To ensure *coalesced accesses*, consecutive threads in the same wavefront should be mapped to columns (the second dimension) of the matrices**
    - **This will give coalesced accesses in Matrices B and C**
    - **For Matrix A, the iterator *i3* determines the access pattern for row-major data, so thread mapping does not affect it**

Thread Mapping

- **In mapping 1, consecutive threads (*tx*) are mapped to different rows of Matrix C, and non-consecutive threads (*ty*) are mapped to columns of Matrix B**
  - **The mapping causes inefficient memory accesses**

```
int tx = get_global_id(0);
int ty = get_global_id(1);
for(i3=0; i3<P; i3++)
    C[tx][ty] += A[tx][i3]*B[i3][ty];
```

Thread Mapping

- **In mapping 2, consecutive threads (*tx*) are mapped to consecutive elements in Matrices B and C**
  - **Accesses to both of these matrices will be coalesced**
    - **Degree of coalescence depends on the workgroup and data sizes**

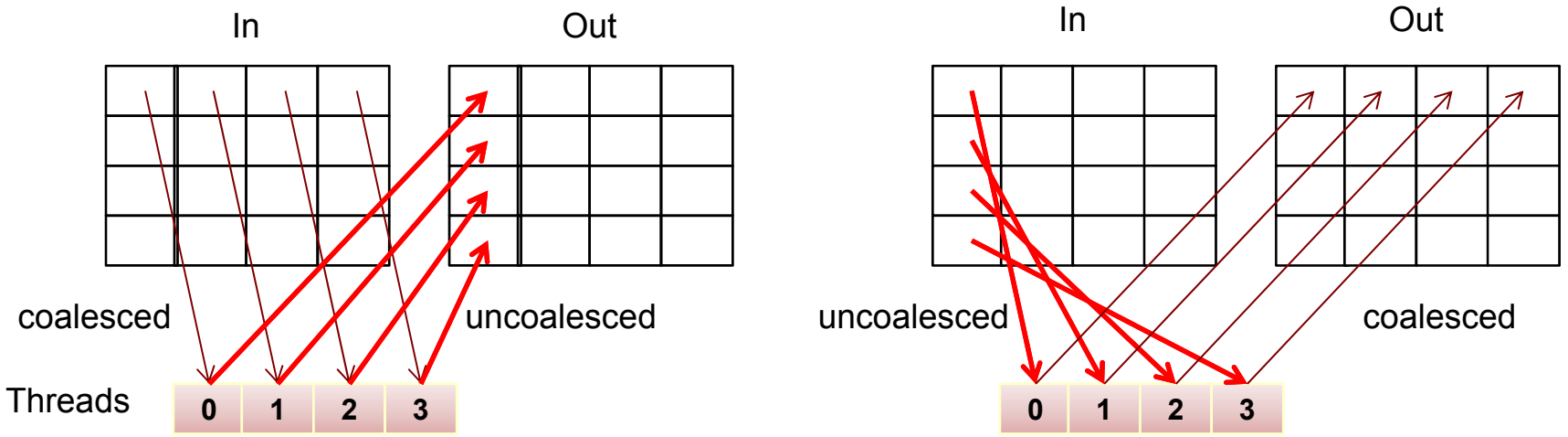
```
int tx = get_global_id (0);
int ty = get_global_id (1);
for(i3=0; i3<P; i3++)
    C[ty][tx] += A[ty][i3]*B[i3][tx];
```

Thread Mapping

- In general, threads can be created and mapped to any data element by manipulating the values returned by the thread identifier functions
- The following matrix transpose example will show how thread IDs can be modified to achieve efficient memory accesses

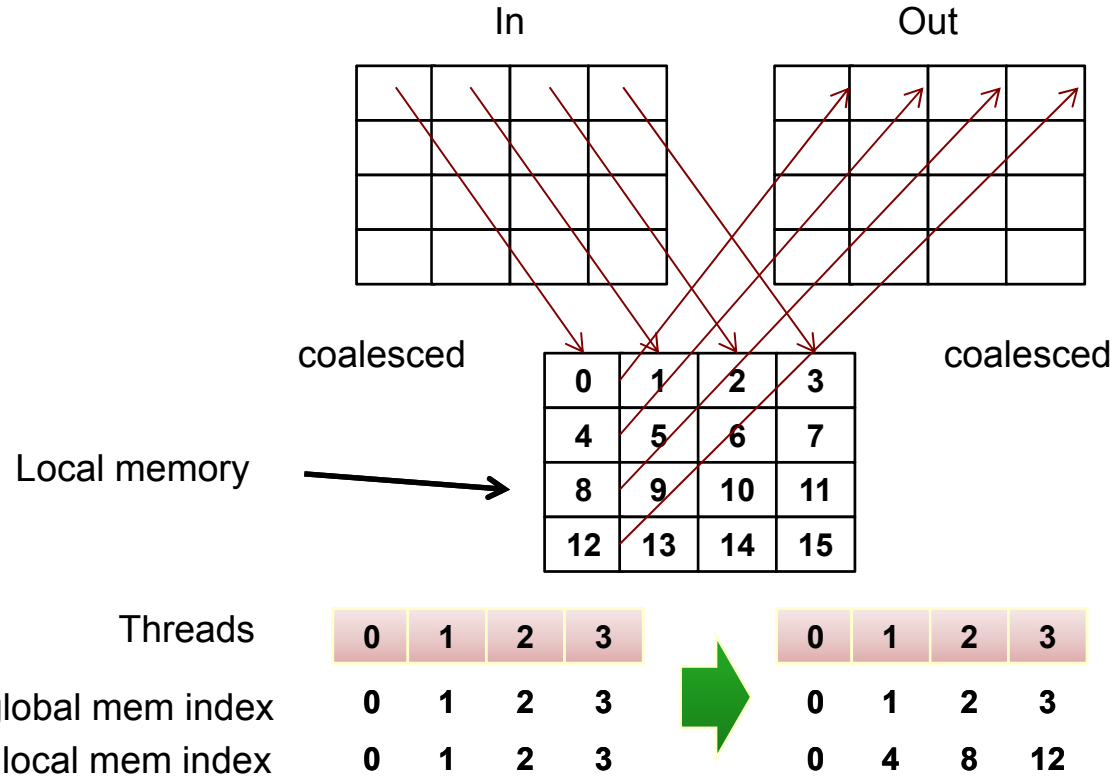
Matrix Transpose

- A matrix transpose is a straightforward technique
  - Out(x,y) = In(y,x)
- No matter which thread mapping is chosen, one operation (read/write) will produce coalesced accesses while the other (write/read) produces uncoalesced accesses
  - Note that data must be read to a temporary location (such as a register) before being written to a new location



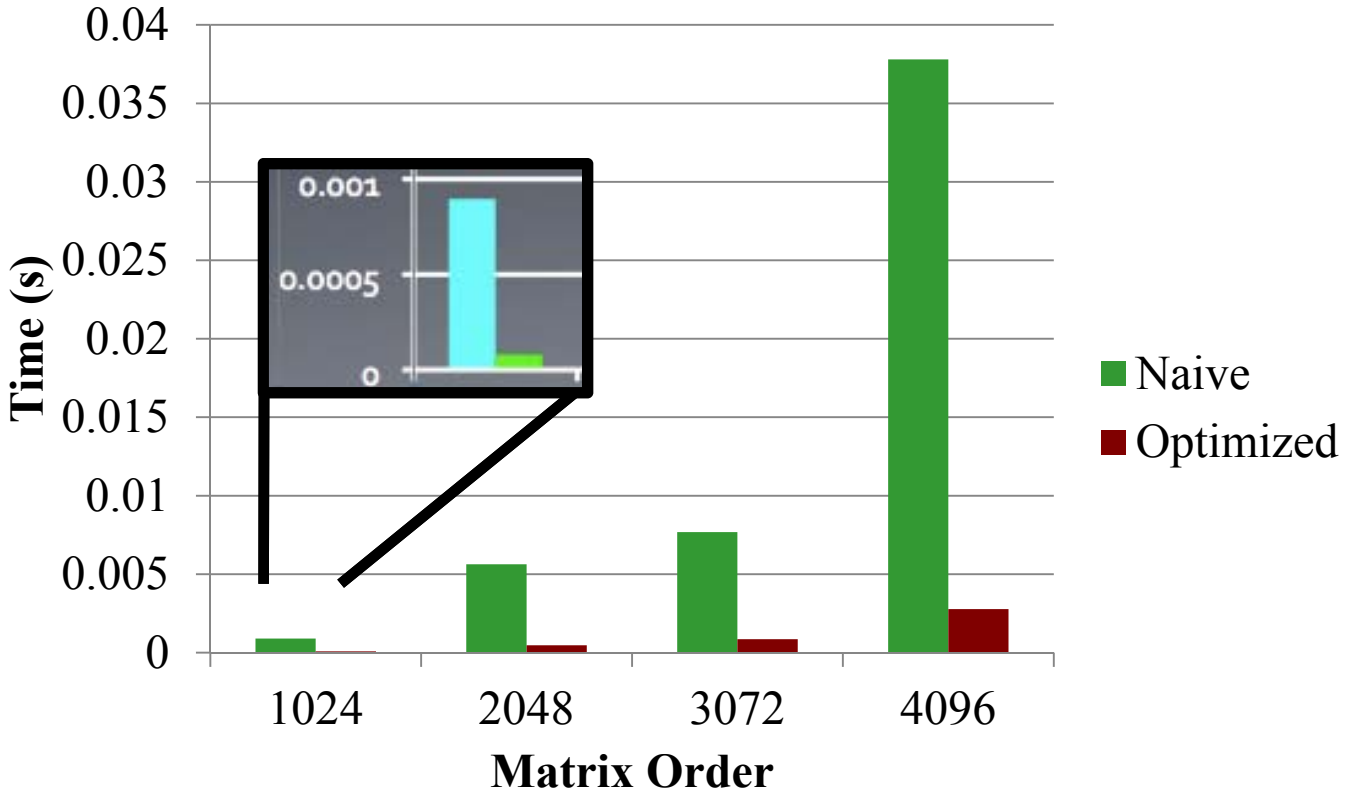
Matrix Transpose

- **If local memory is used to buffer the data between reading and writing, we can rearrange the thread mapping to provide coalesced accesses in both directions**
  - **Note that the work group must be square**



Matrix Transpose

- **The following figure shows a performance comparison of the two transpose kernels for matrices of size N×M on an AMD 5870 GPU**
  - **"Optimized" uses local memory and thread remapping**



Occupancy

- On current GPUs, work groups get mapped to compute units
  - When a work group is mapped to a compute unit, it cannot be swapped off until all of its threads complete their execution
- If there are enough resources available, multiple work groups can be mapped to the same compute unit at the same time
  - Wavefronts from another work group can be swapped in to hide latency
- Resources are fixed per compute unit (number of registers, local memory size, maximum number of threads)
  - Any one of these resource constraints may limit the number of work groups on a compute unit
- The term *occupancy* is used to describe how well the resources of the compute unit are being utilized

Occupancy – Registers

- **The availability of registers is one of the major limiting factor for larger kernels**
- **The maximum number of registers required by a kernel must be available for all threads of a workgroup**
  - **Example: Consider a GPU with 16384 registers per compute unit running a kernel that requires 35 registers per thread**
    - **Each compute unit can execute at most 468 threads**
    - **This affects the choice of workgroup size**
      - A workgroup of 512 is not possible
      - Only 1 workgroup of 256 threads is allowed at a time, even though 212 more threads could be running
      - 3 workgroups of 128 threads are allowed, providing 384 threads to be scheduled, etc.

Occupancy – Registers

- **Consider another example:**
  - **A GPU has 16384 registers per compute unit**
  - **The work group size of a kernel is fixed at 256 threads**
  - **The kernel currently requires 17 registers per thread**
- **Given the information, each work group requires 4352 registers**
  - **This allows for 3 active work groups if registers are the only limiting factor**
- **If the code can be restructured to only use 16 registers, then 4 active work groups would be possible**

Occupancy – Local Memory

- GPUs have a limited amount of local memory on each compute unit
  - 32KB of local memory on AMD GPUs
  - 32-48KB of local memory on NVIDIA GPUs
- Local memory limits the number of active work groups per compute unit
- Depending on the kernel, the data per workgroup may be fixed regardless of number of threads (e.g., histograms), or may vary based on the number of threads (e.g., matrix multiplication, convolution)

Occupancy – Threads

- GPUs have hardware limitations on the maximum number of threads per work group
  - 256 threads per WG on AMD GPUs
  - 512 threads per WG on NVIDIA GPUs
- NVIDIA GPUs have per-compute-unit limits on the number of active threads and work groups (depending on the GPU model)
  - 768 or 1024 threads per compute unit
  - 8 or 16 warps per compute unit
- AMD GPUs have GPU-wide limits on the number of wavefronts
  - 496 wavefronts on the 5870 GPU (~25 wavefronts or ~1600 threads per compute unit)

Occupancy – Limiting Factors

- The minimum of these three factors is what limits the active number of threads (or occupancy) of a compute unit
- The interactions between the factors are complex
  - The limiting factor may have either thread or wavefront granularity
  - Changing work group size may affect register or shared memory usage
  - Reducing any factor (such as register usage) slightly may have allow another work group to be active
- The CUDA occupancy calculator from NVIDIA plots these factors visually allowing the tradeoffs to be visualized

CUDA Occupancy Calculator

- **CUDA occupancy calculator:**

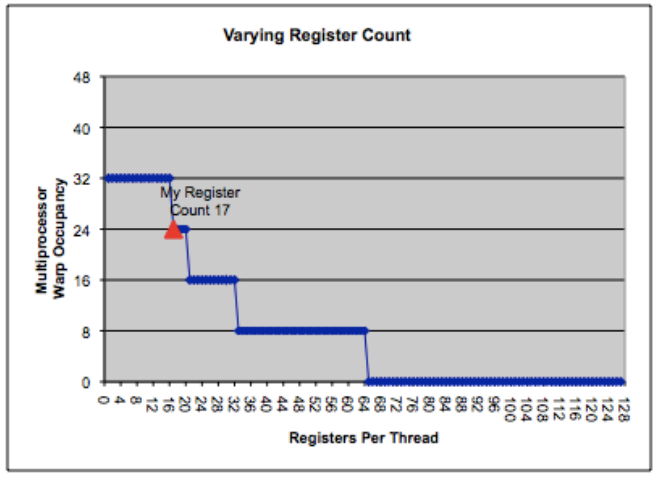
1. Enter hardware model and kernel requirements

1.) Select Compute Capability (click):	2.0
2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	8
Shared Memory Per Block (bytes)	1024

2. Resource usage and limiting factors are displayed

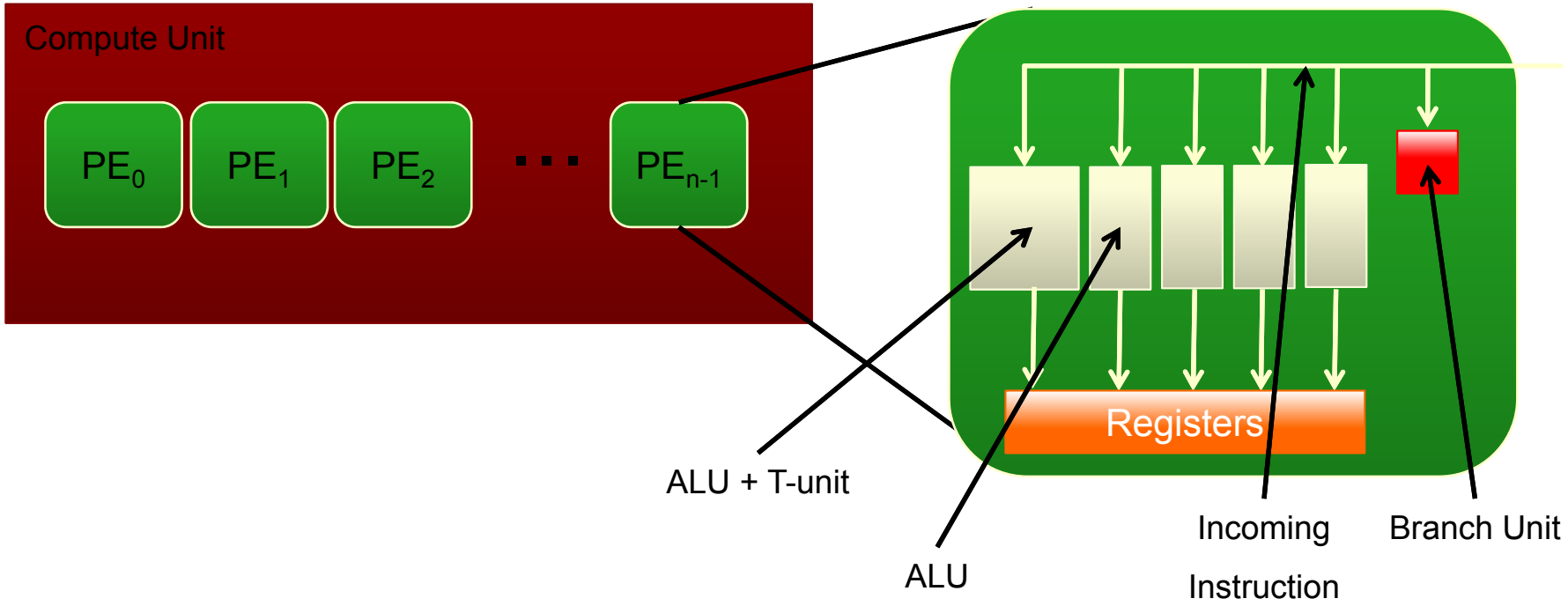
Allocation Per Thread Block	
Warps	8
Registers	4608
Shared Memory	1024
These data are used in computing the occupancy data in blue	
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Blocks per Multiprocessor	6
Limited by Registers per Multiprocessor	7
Limited by Shared Memory per Multiprocessor	48

3. Graphs are shown to visualize limiting factors



Vectorization

- **On AMD GPUs, each processing element executes a 5-way VLIW instruction**
  - **5 scalar operations or**
  - **4 scalar operations + 1 transcendental operation**



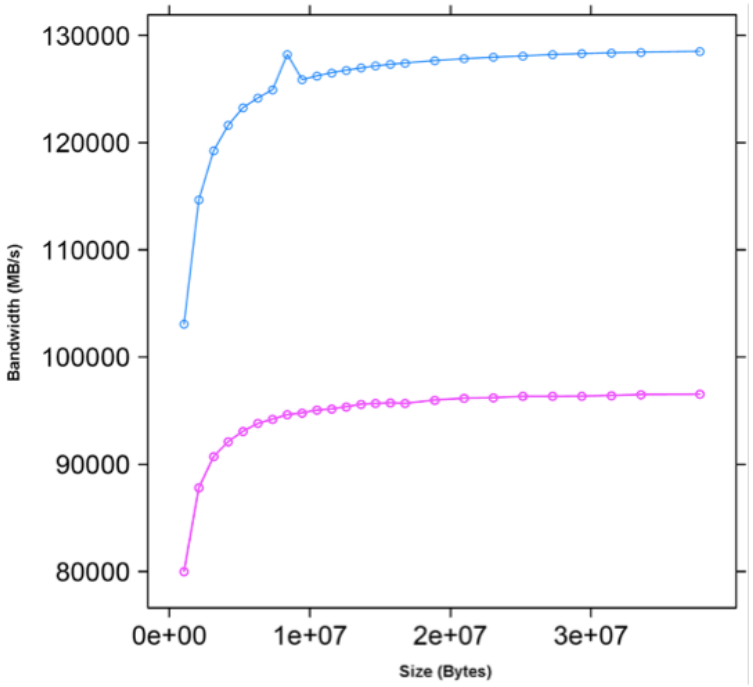
Vectorization

- Vectorization allows a single thread to perform multiple operations at once
- Explicit vectorization is achieved by using vector datatypes (such as `float4`) in the source program
  - When a number is appended to a datatype, the datatype becomes an array of that length
  - Operations can be performed on vector datatypes just like regular datatypes
    - Each ALU will operate on different element of the float4 data

Vectorization

- **Vectorization improves memory performance on AMD GPUs**
  - **The *AMD Accelerated Parallel Processing OpenCL Programming Guide* compares `float` to `float4` memory bandwidth**

```
__kernel void
Copy4(__global const float4 * input,
      __global float4 * output)
{
  int gid = get_global_id(0);
  output[gid] = input[gid];
  return;
}
__kernel void
Copy1(__global const float * input,
      __global float * output)
{
  int gid = get_global_id(0);
  output[gid] = input[gid];
  return;
}
```



Vectorization: Example

- **Non-vectorized sum**

```
float x[0] = v[0]+w[0];
float x[1] = v[1]+w[1];
float x[2] = v[2]+w[2];
float x[3] = v[3]+w[3];
```

- **Vectorized sum**

```
float4 x = (float4)(v[0],v[1],v[2],v[3])
         + (float4)(w[0],w[1],w[2],w[3]);
```

Optimizations: Summary

- Although writing a simple OpenCL program is relatively easy, optimizing code can be very difficult
  - Improperly mapping loop iterations to OpenCL threads can significantly degrade performance
- When creating work groups, hardware limitations (number of registers, size of local memory, etc.) need to be considered
  - Work groups must be sized appropriately to maximize the number of active threads and properly hide latencies
- Vectorization is an important optimization for AMD GPU hardware
  - Though not covered here, vectorization may also help performance when targeting CPUs

Timing

- 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

Timing: 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

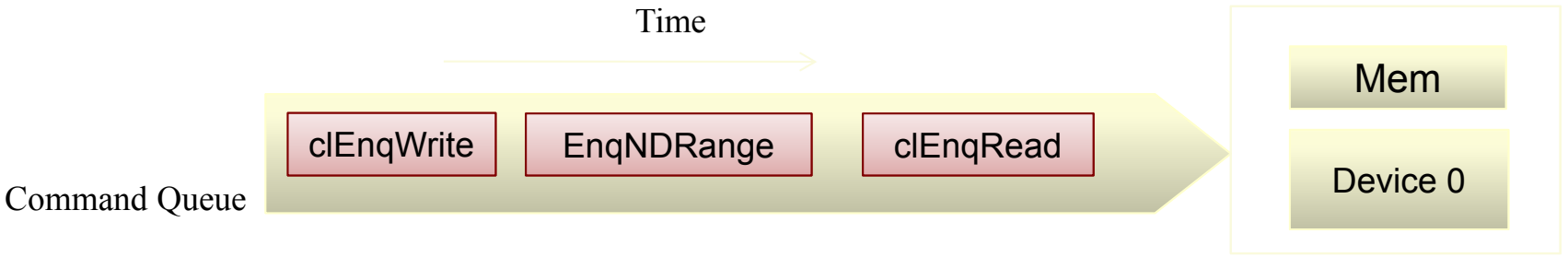
Command Queues

- 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

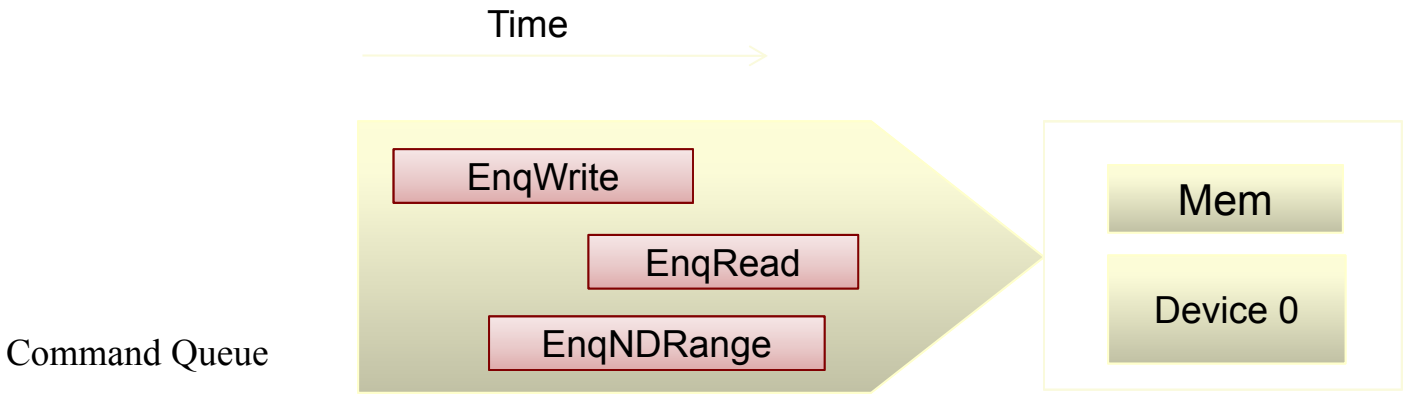
- In an in-order command queue, each command executes after the previous one has finished
  - For the set of commands shown, the read from the device would start after the kernel call has finished
- Memory transactions have consistent view

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 enqueued 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);
```

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

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

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

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

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**

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**

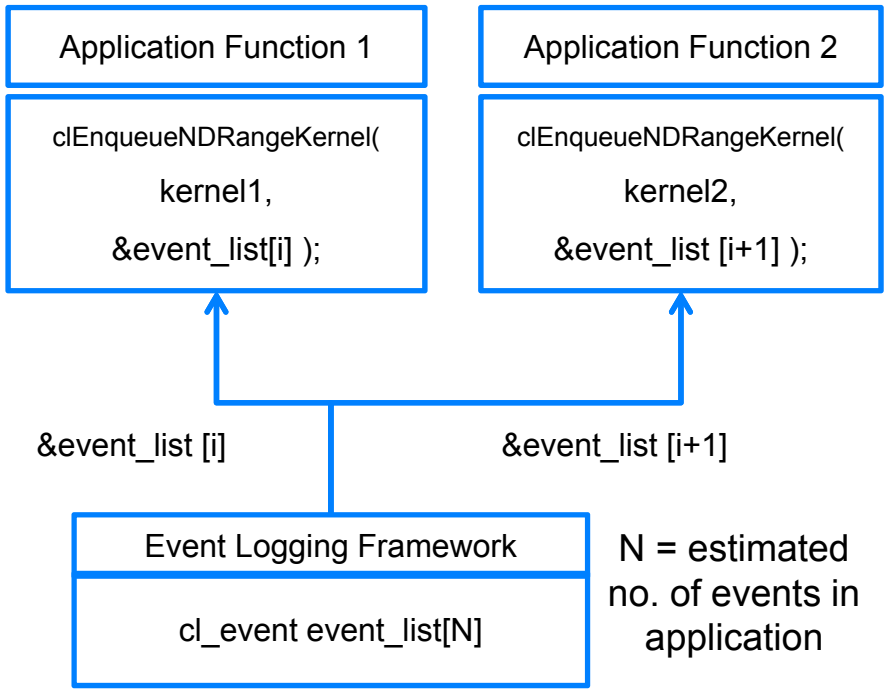
```
clGetEventProfilingInfo( event_time,
CL_PROFILING_COMMAND_START,
sizeof(cl_ulong), &starttime, NULL);
```

```
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

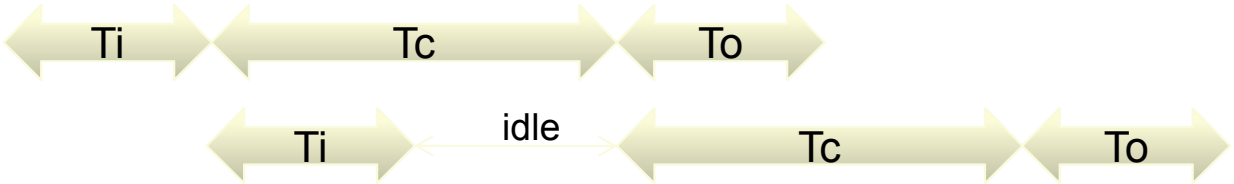
Asynchronous I/O

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)

Case 1: 1 Queue

Total Time = 2Tc + 2Ti + 2To

Case 2: 2 Queues – One device for compute



Total Time for 2 kernels and their I/O

= 2Tc + Ti + To

Note: Idle time denotes time when no IO occurs. The compute units of the GPU are busy

Asynchronous I/O

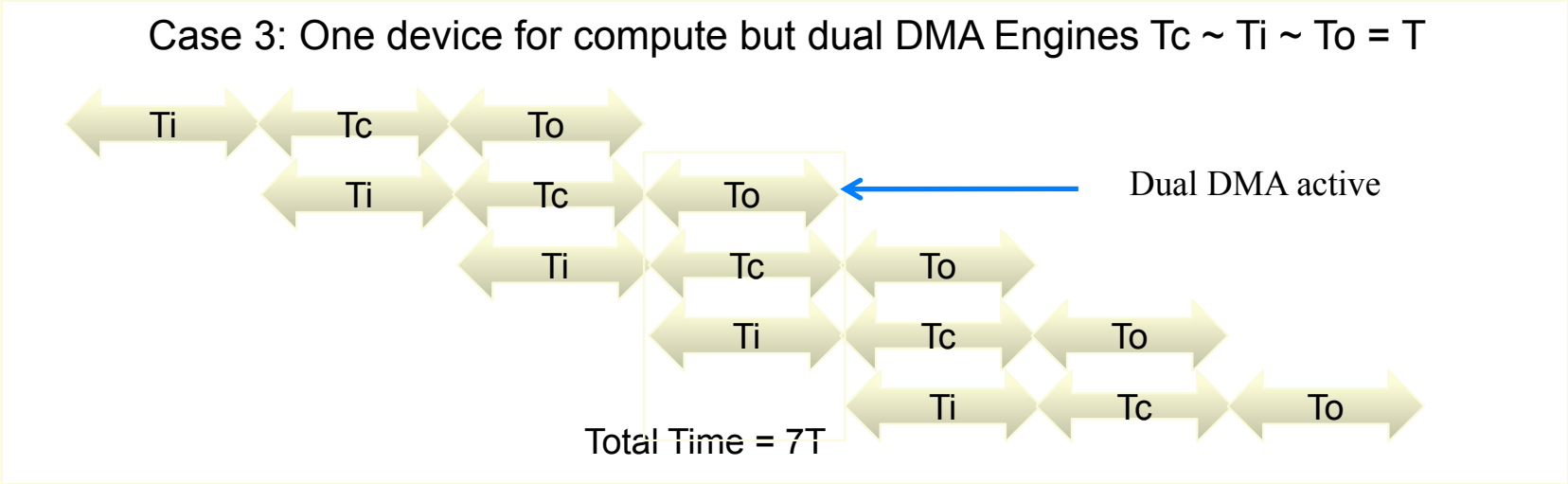
- **Time with 1 Queue = 2Tc + 2Ti + 2To**
  - **No asynchronous behavior**
- **Time with 2 Queues = 2Tc + Ti + To**
  - **Overlap computation and communication**

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

- **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**

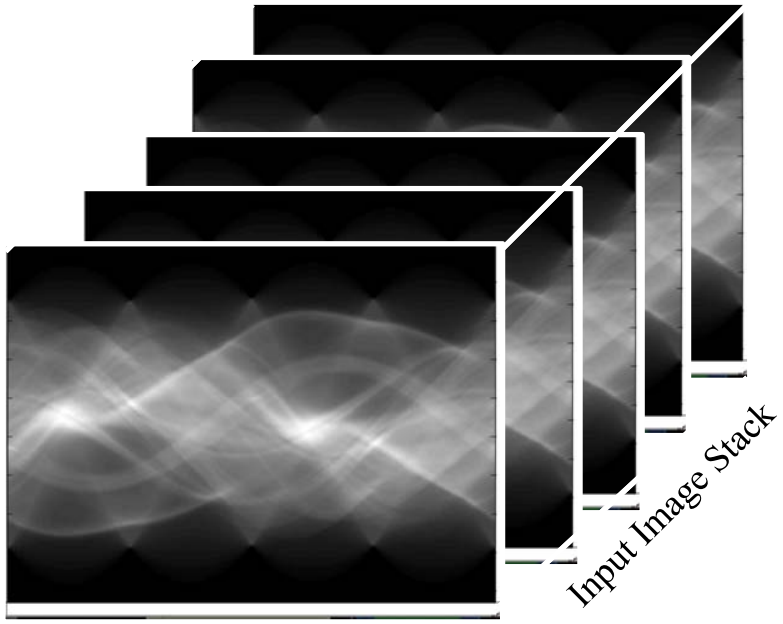
Dual DMA Engine Case

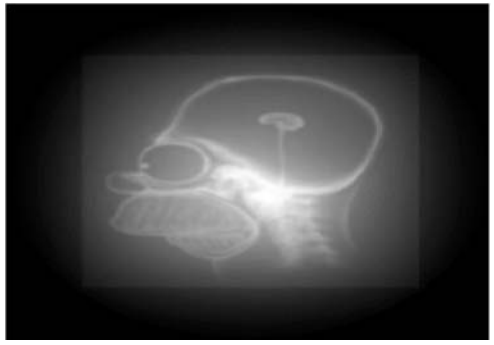
- **In the Nvidia Fermi GPUs, dual DMA engines allow simultaneous bidirectional IO.**



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

Example: Image Reconstruction





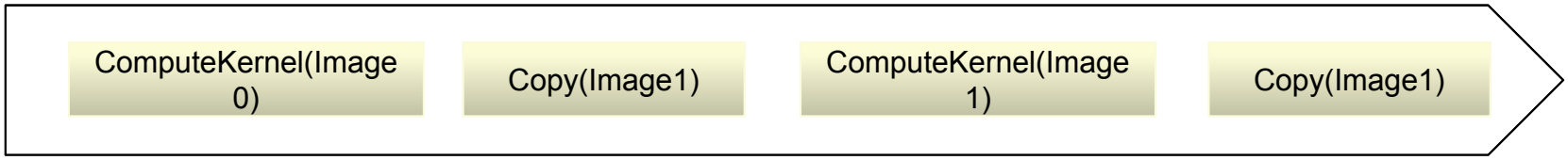
Reconstructed OP Image

- **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

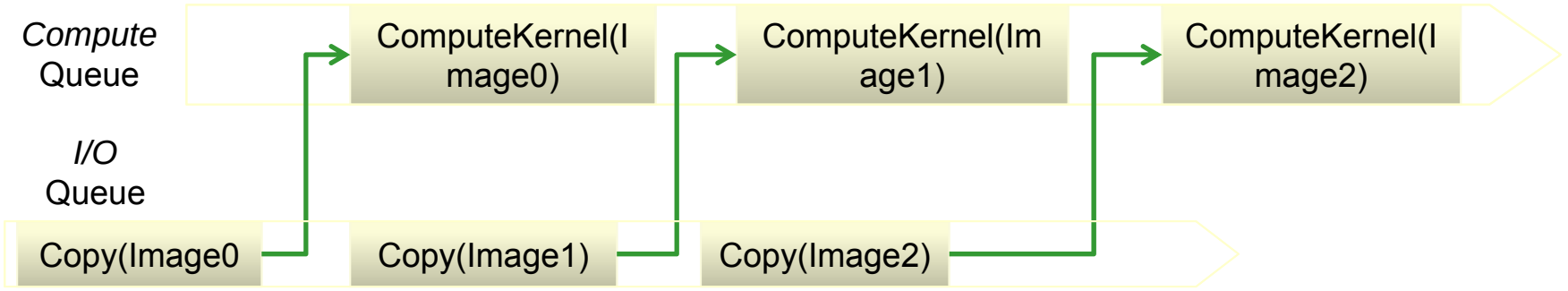
- **Single Command Queue: N images = N(tcompute + transfer)**



- **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])
}
```

Timing: 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

Debugging Techniques

- Compiling for x86 CPU
  - Debugging with GDB
- GPU printf
- Live debuggers
  - Parallel Nsight
  - gDEBugger

CPU Debugging

- OpenCL allows the same code to run on different types of devices
  - Compiling to run on a CPU provides some extra facilities for debugging
  - Additional forms of IO (such as writing to disk) are still not available from the kernel
- AMD's OpenCL implementation recognizes any x86 processor as a target device
  - Simply select the CPU as the target device when executing the program
- NVIDIA's OpenCL implementation can support compiling to x86 CPUs if AMD's installable client driver is installed

CPU Debugging with GDB

- **Setting up for GDB**
  - **Pass the compiler the "-g" flag**
    - **Pass "-g" to `clBuildProgram()`**
    - **Set an environment variable `CPU_COMPILER_OPTIONS="-g"`**
  - **Avoid non-deterministic execution by setting an environment variable `CPU_MAX_COMPUTE_UNITS=1`**

CPU Debugging with GDB

- Run gdb with the OpenCL executable

```
> gdb a.out
```

- Breakpoints can be set by line number, function name, or kernel name
- To break at the kernel `hello` within gdb, enter:

```
(gdb) b __OpenCL_hello_kernel
```

  - The prefix and suffix are required for kernel names

- OpenCL kernel symbols are not known until the kernel is loaded, so setting a breakpoint at `clEnqueueNDRangeKernel()` is helpful

```
(gdb) b clEnqueueNDRangeKernel
```

CPU Debugging with GDB

- **To break on a certain thread, introduce a conditional statement in the kernel and set the breakpoint inside the conditional body**
  - **Can use gdb commands to view thread state at this point**

```
...
if(get_global_id(1) == 20 &&
   get_global_id(0) == 34) {
   ;  // Set breakpoint on this line
}
```

GPU Printf

- AMD GPUs support printing during execution using `printf()`
  - NVIDIA does not currently support printing for OpenCL kernels (though they do with CUDA/C)
- AMD requires the OpenCL extension *cl_amd_printf* to be enabled in the kernel
- `printf()` closely matches the definition found in the C99 standard

GPU Printf

- **`printf()` can be used to print information about threads or check help track down bugs**
- **The following example prints information about threads trying to perform an improper memory access**

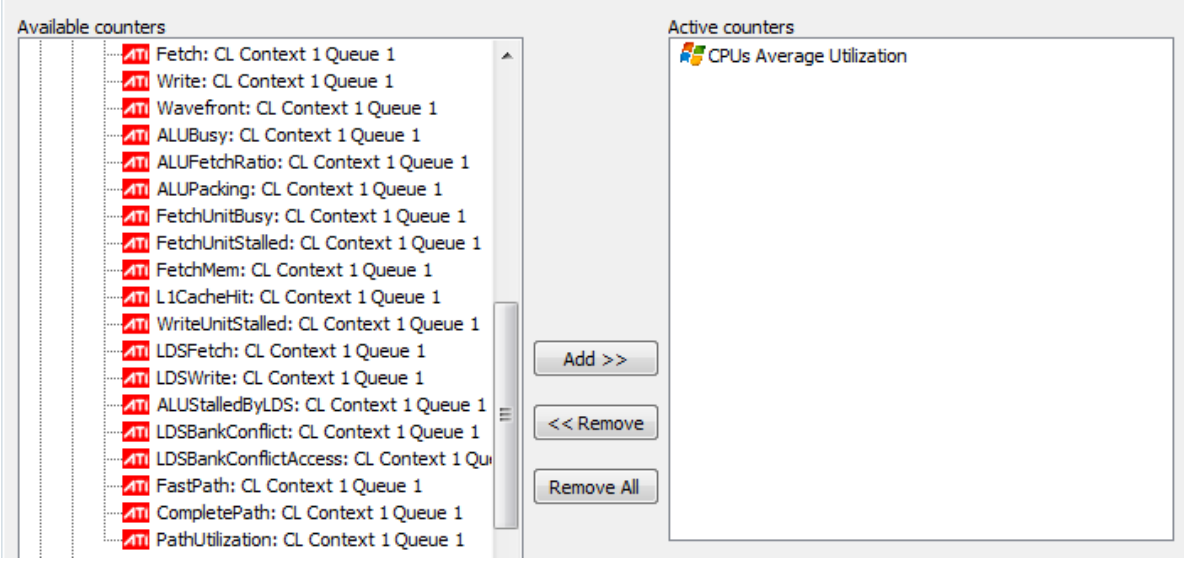
```
int myIdx = ... // index for addressing a matrix
if(myIdx < 0 || myIdx >= rows || myIdx >= cols) {
    printf("Thread %d,%d: bad index (%d)\n",
    get_global_id(1), get_global_id(0), myIdx));
}
```

GPU Printf

- `printf()` works by buffering output until the end of execution and transferring the output back to the host
  - It is important that a kernel completes in order to retrieve printed information
  - Commenting out code following `printf()` is a good technique if the kernel is crashing

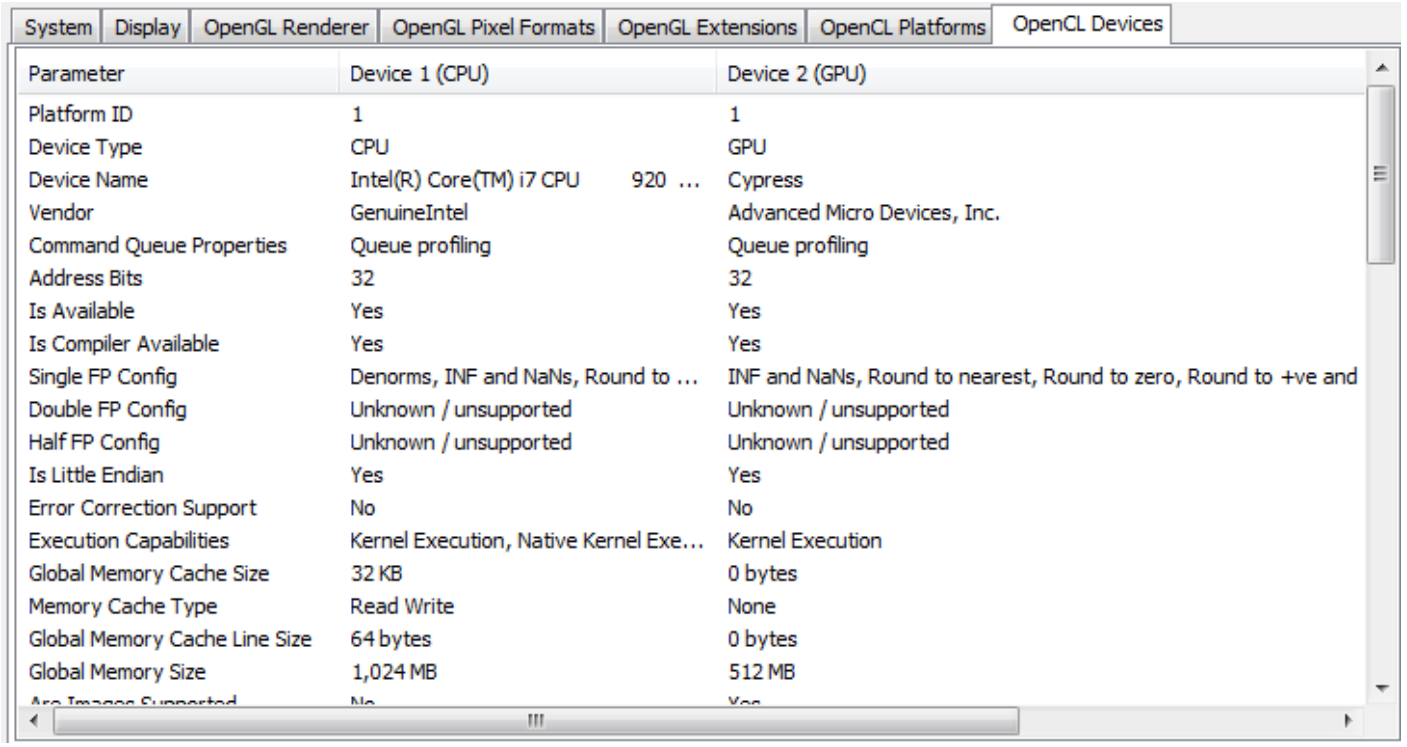
gDEBugger

- Developed by Graphic Remedy
  - Cost: not free
- Debugger, profiler, memory analyzer
- Integrated with AMD/ATI and NVIDIA performance counters



gDEBugger

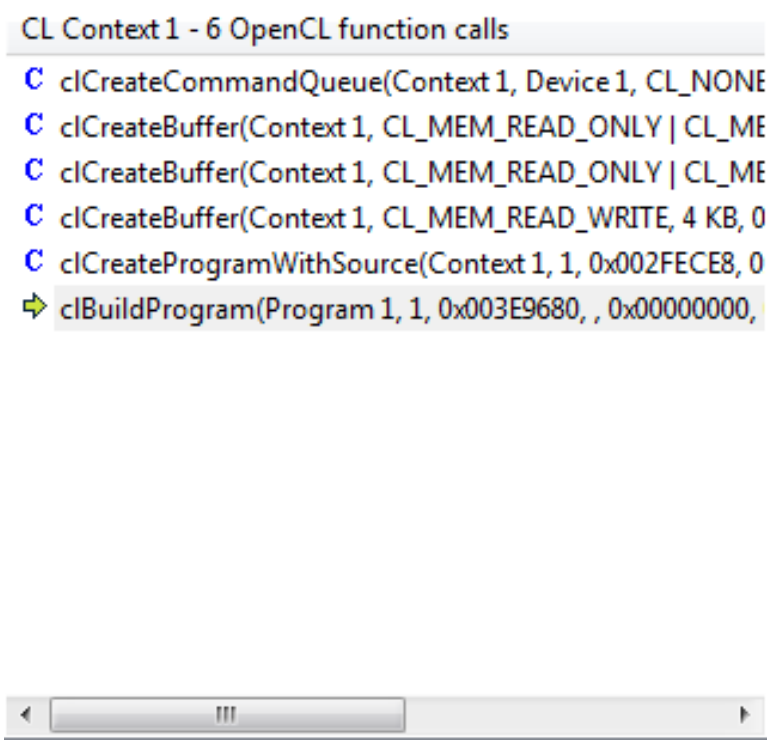
- **Displays information about OpenCL platforms and devices present in the system**

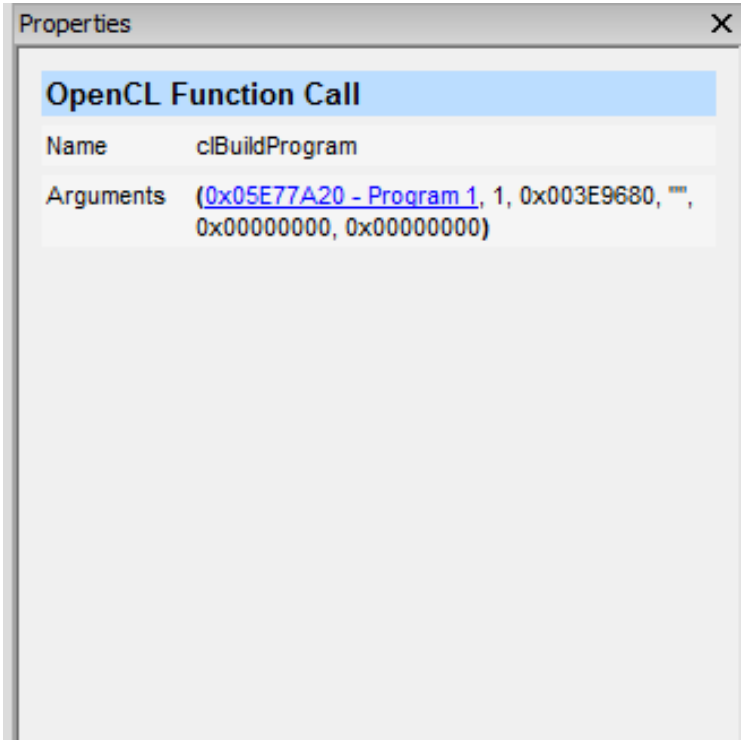


Parameter	Device 1 (CPU)	Device 2 (GPU)
Platform ID	1	1
Device Type	CPU	GPU
Device Name	Intel(R) Core(TM) i7 CPU 920 ...	Cypress
Vendor	GenuineIntel	Advanced Micro Devices, Inc.
Command Queue Properties	Queue profiling	Queue profiling
Address Bits	32	32
Is Available	Yes	Yes
Is Compiler Available	Yes	Yes
Single FP Config	Denorms, INF and NaNs, Round to ...	INF and NaNs, Round to nearest, Round to zero, Round to +ve and
Double FP Config	Unknown / unsupported	Unknown / unsupported
Half FP Config	Unknown / unsupported	Unknown / unsupported
Is Little Endian	Yes	Yes
Error Correction Support	No	No
Execution Capabilities	Kernel Execution, Native Kernel Exe...	Kernel Execution
Global Memory Cache Size	32 KB	0 bytes
Memory Cache Type	Read Write	None
Global Memory Cache Line Size	64 bytes	0 bytes
Global Memory Size	1,024 MB	512 MB

gDEBugger

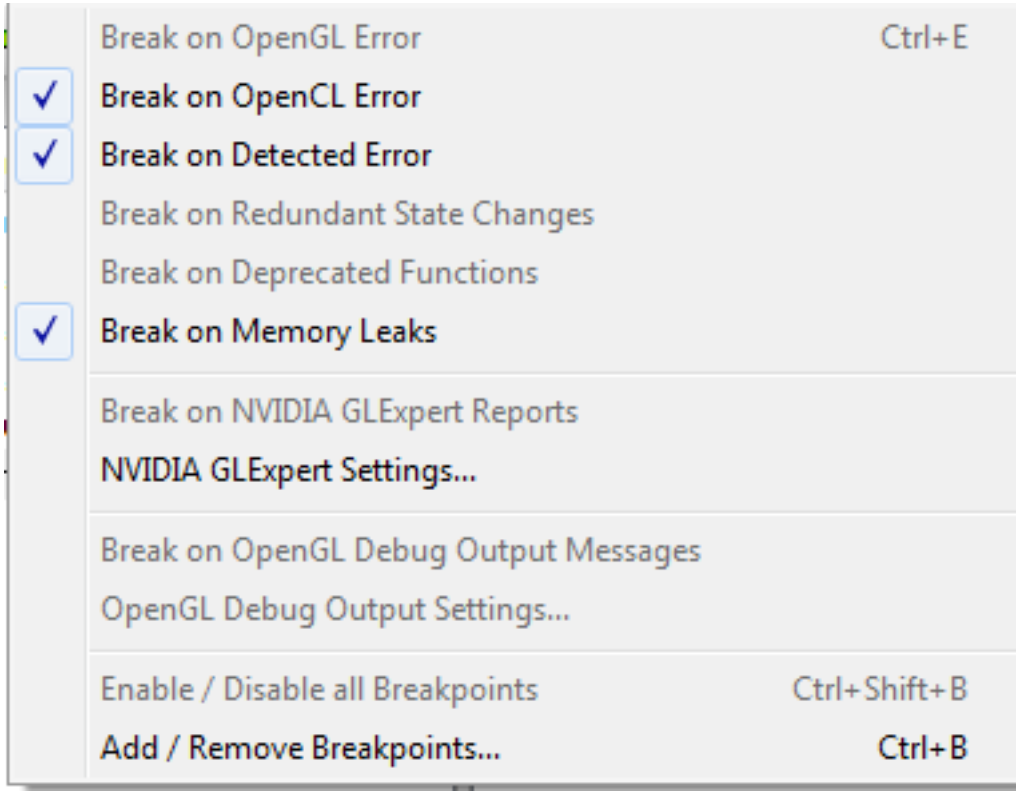
- **Can step through OpenCL calls, and view arguments**
  - **Links to programs, kernels, etc. when possible in the function call view**





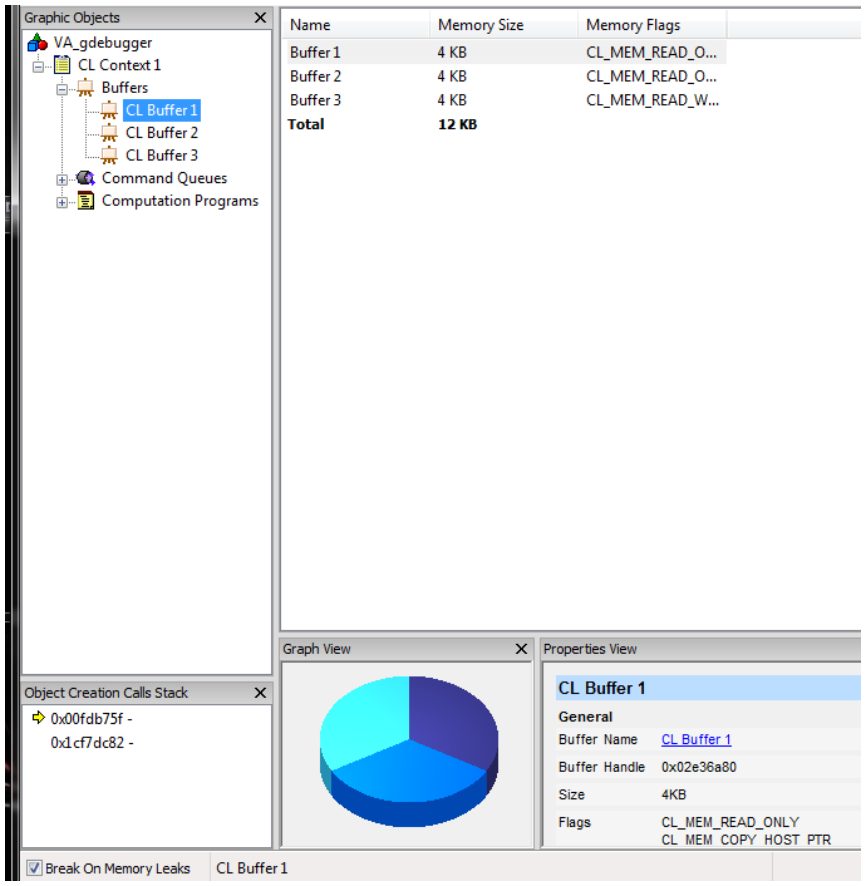
gDEBugger

- **Automatically detects OpenCL errors and memory leaks**



gDEBugger

- **Displays contents of buffers and images present on OpenCL devices**
  - **View live**
  - **Export to disk**



Debugging: Summary

- **GPU debugging is still immature**
  - **NVIDIA has a live debugger for Windows only**
  - **AMD and NVIDIA allow restrictive printing from the GPU**
  - **AMD allows code to be compiled and run with gdb on the CPU**
  - **Graphic Remedy (gDEBugger) provides online memory analysis and is integrated with performance counters, but cannot debug on a thread-by-thread basis**