GPU TECHNOLOGY CONFERENCE
April 4-7, 2016 | Silicon Valley

PERFORMANCE CONSIDERATIONS FOR OPENCL ON NVIDIA GPUS

Karthik Raghavan Ravi, 4/4/16

PRESENTED BY
NVIDIA.

THE PROBLEM

OpenCL is portable across vendors and implementations, but not always at peak performance

OBJECTIVE OF THIS TALK

Discuss

- common perf pitfalls in the API and ways to avoid them
- high performance paths for NVIDIA
- leveraging recent enhancements in the driver

AGENDA

EXECUTION

- Perf Knobs in the API
- Waiting for Work Completion

DATA MOVEMENT

- Better Copy Compute Overlap
- Better Interoperability with OpenGL
- Shared Virtual Memory

PERF KNOBS IN THE API

OCCUPANCY AND PERFORMANCE

Background

Occupancy = #active threads / max threads that could be active at a time

The goal should be to have enough active warps to keep the GPU busy computing stuff and hide the data access latency

Note: occupancy can only hide latency due to memory accesses; instruction computation latency needs to be hidden by providing enough independent instructions between dependent operations

OCCUPANCY AND PERFORMANCE

Older talks

"CUDA Warps and Occupancy" – Dr Justin Luitjens, Dr Steven Rennich. Deep dive into limiting factors for occupancy: http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf

"Better Performance at lower Occupancy" – Vasily Volkov. Argument for how performance can be extracted by improving instruction level parallelism: http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf

"GPU Optimization Fundamentals" – Cliff Woolley. Multiple strategies to analyze and improve performance of compute apps: https://www.olcf.ornl.gov/wp-content/uploads/2013/02/GPU_Opt_Fund-CW1.pdf

WORK-GROUP SIZES

OCCUPANCY AND PERFORMANCE

Work-group sizes

NDRange divided into work-groups

All work items in a work group execute on the same compute unit, share resources of the compute unit

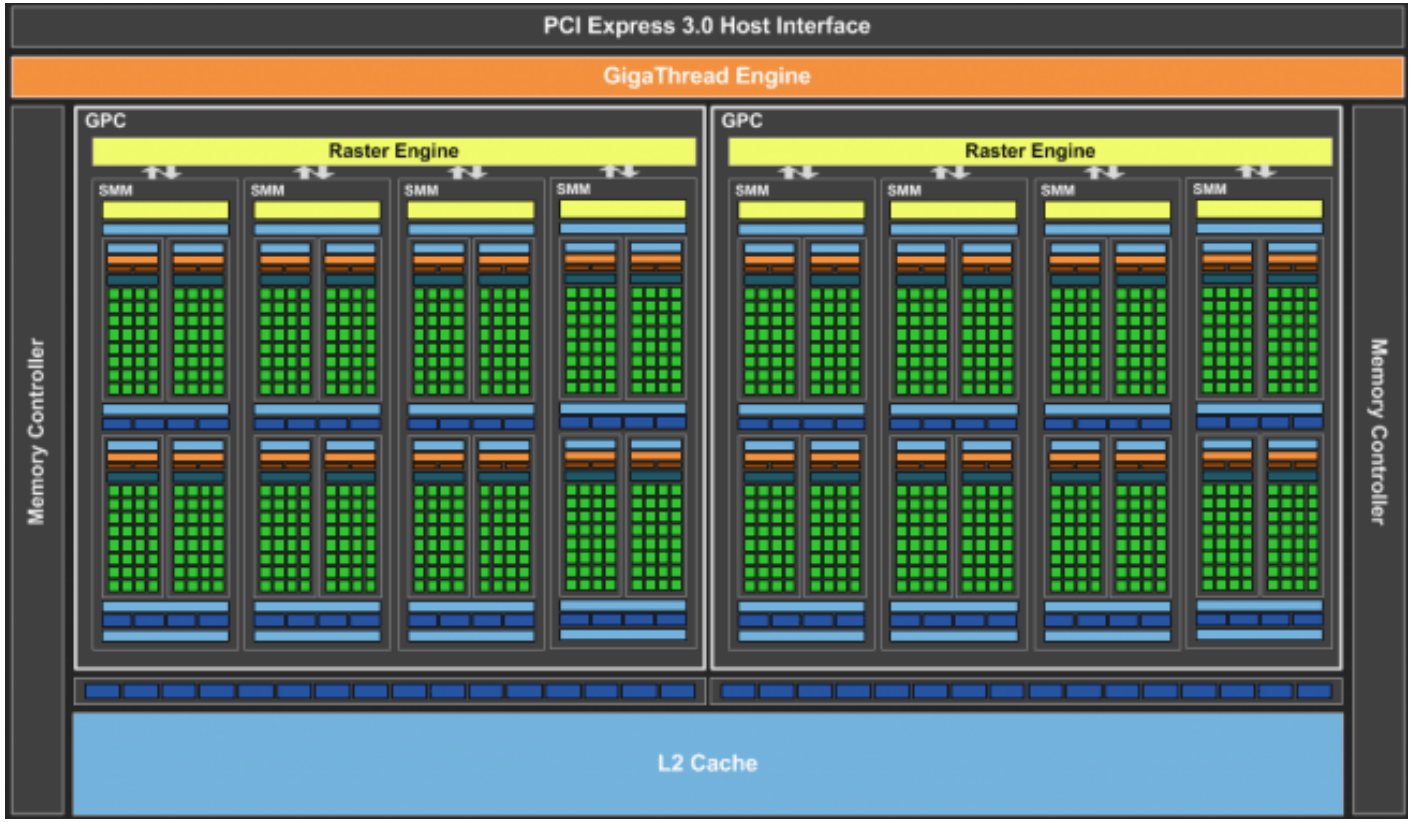
Multiple work-groups can be scheduled on the same compute unit

OCCUPANCY AND PERFORMANCE

Work-group sizes

For NVIDIA,

- the compute unit is an SM
- the key shared resources are shared memory, registers



OCCUPANCY AND PERFORMANCE

Too small a local work-group size

Constraint: Work items of a local work-group are scheduled on to SMs in groups [SIMT], with the size of this set being architecture-defined [1]

Pitfall: A local work-group size of less than this number leaves some of the streaming processors unutilized but occupied

Have the work-group size to be at least the number of threads that get scheduled together

Larger work-group sizes ideally need to be a multiple of this number

[1] this can be obtained from the GPU manual/programming guide

OCCUPANCY AND PERFORMANCE

Too large a local work-group size

Constraint: All threads of a local work-group will share the resources of the SM

Pitfall: Having too large a local work-group size typically increases pressure on registers and shared memory, impacting occupancy

For contemporary architectures, 256 is a good starting point, but obviously each kernel is different and deserves investigation to identify ideal sizes

OCCUPANCY AND PERFORMANCE

Too large a local work-group size

Constraint: All threads of a local work-group will be scheduled on the same SM

Pitfall: If there are lesser work-groups than the number of SMs in the GPU, a few SMs will see high contention while a few SMs will run idle

Also consider the number of work-groups when trying to size your grid

OCCUPANCY AND PERFORMANCE

Good global work sizes

Constraint: local work-group size needs to be a divisor of the corresponding global work size dimension size in OpenCL 1.x

Pitfall: primes and small multiples of primes are bad (evil?) global work sizes

Consider resizing the NDRange to something that provides many work-group size options.

Depending on the kernel, having some threads early-out might be better than a poor size affecting all threads

OCCUPANCY AND PERFORMANCE

Runtime support for choosing a local work-group size

The OpenCL API allows applications to ask the runtime to choose an optimal size

The NVIDIA OpenCL runtime takes into account all the previous heuristics while choosing a local work-group size

This can serve as a good starting point for optimization. Do not expect this to be the best possible option for all the kernels out there.

The heuristic cannot violate constraints cited earlier!

OCCUPANCY AND PERFORMANCE

Caveats

The resources per SM changes with architectures, and other parameters such as warp size are also architecture-specific

This means that a configuration ideal for one architecture may not be ideal for all architectures

Revalidate architecture-specific tuning for each architecture

REGISTER USAGE

RESTRICTING REGISTER USAGE

Only as many threads as there are resources for can be run

Occupancy might potentially be limited by register usage

Reducing this and improving occupancy might potentially* improve performance

Per-thread register usage can be capped via an NVIDIA OpenCL extension:
cl_nv_compiler_options

Play around with this knob to see if occupancy improves, and if improved occupancy provides gains

*See caveats

RESTRICTING REGISTER USAGE

Caveats

Reducing per-thread register usage will likely affect per-thread performance. Trading this off with increased occupancy needs to be resolved differently for different kernels

Better occupancy is equal to better performance only till memory latency is visible

This tuning is also architecture-specific. Changes in arch might move bottlenecks elsewhere and make tuning inapplicable

WAITING FOR WORK COMPLETION

WAITING FOR WORK COMPLETION

The Inefficient and Potentially Incorrect Way

Spinning on event status waiting for it to become CL_COMPLETE:

```
while(clGetEventInfo(myEvent, CL_EVENT_COMMAND_EXECUTION_STATUS) !=
CL_COMPLETE)

{}
```

WAITING FOR WORK COMPLETION

The Inefficient and Potentially Incorrect Way

Inefficient because external influences can cause a large amount of variance on when the app knows about event completion

Potentially Incorrect because event status becoming CL_COMPLETE is not a synchronization point. To quote the spec,

"There are no guarantees that the memory objects being modified by command associated with event will be visible to other enqueued commands"

WAITING FOR WORK COMPLETION

The Efficient and Correct Way

Use clWaitForEvents

- low latency, since the runtime already implements this call as a low-latency spin wait on internal work-tracking structures
- correct, since completion of this call guarantees that "*commands identified by event objects in event_list [are] complete*"

BETTER COPY COMPUTE OVERLAP

COPY COMPUTE OVERLAP

The false serialization problem

Independent workloads can serialize if they are contending for the same hardware resource (ex: copy engine)

CPU time is an important resource, and new work submission needs the CPU

Not all host allocations are the same. Copying data between host and GPU is slower and more work if the runtime thinks that host memory could be paged out

Put together, this is a common cause for false serialization between copies and independent work such as kernels

COPY COMPUTE OVERLAP

What's needed?

The runtime needs a guarantee that the memory will not be paged out by the OS at any time

malloc'ed memory does not provide that guarantee

The OpenCL API does not provide a mechanism to allocate page-locked memory, but the NVIDIA OpenCL implementation guarantees some allocations to be pinned on the host

Judicious use of this gives best performance

Read more about this in earlier cited talks

COPY COMPUTE OVERLAP

Allocating Pinned Memory – The Old Way

Allocating page-locked memory

```
dummyClMem = clCreateBuffer(ALLOC_HOST_PTR);

void *hostPinnerPointer = clEnqueueMapBuffer(dummyClMem);
```

Using page-locked memory

Use `hostPinnedPointer` as host memory for host-device transfers as you would malloc'd memory

COPY COMPUTE OVERLAP

Allocating Pinned Memory – The Old Way

In other words, make a host allocation by creating a device buffer and having the OpenCL runtime map it to the host

Not the most direct or intuitive of approaches

COPY COMPUTE OVERLAP

Allocating Pinned Memory – New Support

Map/Unmap calls now internally use pinned memory

To benefit from fast, asynchronous copies, use Map/Unmap instead of Read/Write

COPY COMPUTE OVERLAP

Allocating Pinned Memory – New Support

```
pMem = clEnqueueMapBuffer(clMem); // async call, returns fast
```

<opportunity to do other work on the host while data is being copied>

```
//use pMem once MapBuffer completes
```

```
clEnqueueUnmapMemObject(pMem); // async call, returns fast
```

<opportunity to do other work on the host while data is being copied>

COPY COMPUTE OVERLAP

Caveats

Pinned memory is a scarce system resource, also required for other activities

Heavy use of pinned memory might slow down the entire system or have programs killed unpredictably

Use this resource judiciously

COPY COMPUTE OVERLAP

Avoiding copies

Use CL_WRITE when you want the mapped region to have the latest bits before writing. This involves a device-to-host copy

Use CL_WRITE_INVALIDATE when you know that the mapped region is going to be overwritten by the host. This skips the device-to-host copy altogether, and can give significant performance benefit

BETTER INTEROPERABILITY WITH OPENGL

PAST PAIN POINTS

Multithreaded robustness, API latency

Context and other state is explicit for OpenCL while implicit for OpenGL => lots of trouble with interop for OpenCL implementations, particularly in multithreaded cases

API latency used to be very high, in orders of a few milliseconds instead of tens of microseconds

Fixing such issues enabled better overlap of interop and other work, opening up more subtle improvement opportunities

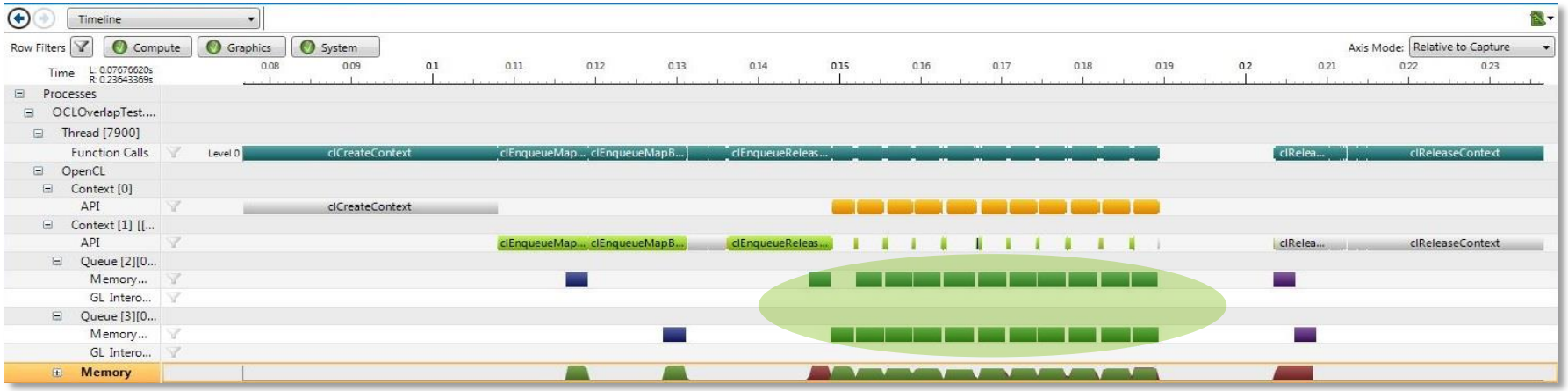
UNEXPECTED SERIALIZATION

UNEXPECTED SERIALIZATION

Consider the following code, running on a GPU with dual copy engines:

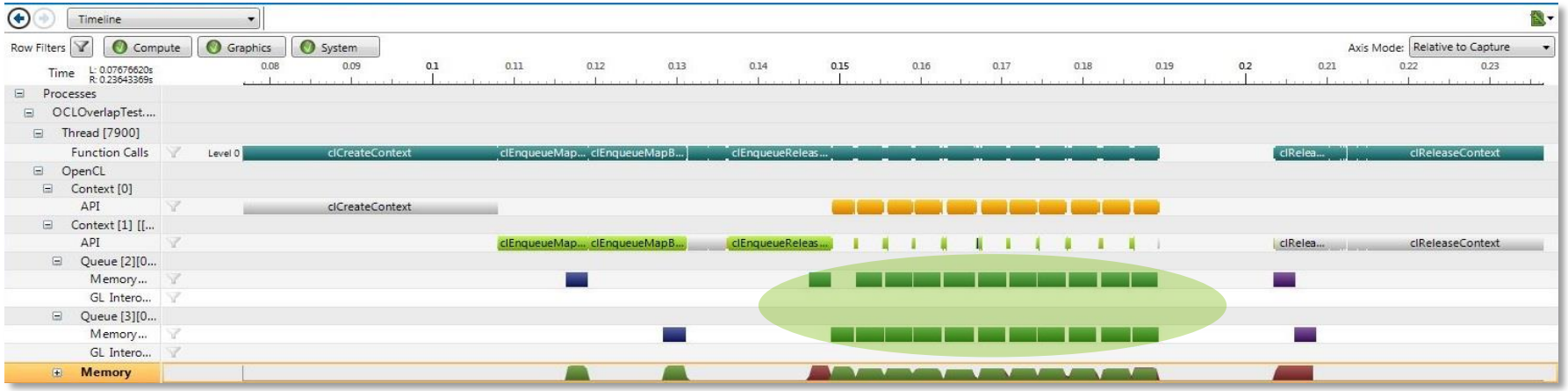
```
while(1) {
    EnqueueAcquireFromGL(memory1, queue1)
    EnqueueWrite(memory1, queue1)
    EnqueueReleaseToGL(memory1, queue1)

    EnqueueAcquireFromGL(memory2, queue2)
    EnqueueRead(memory2, queue2)
    EnqueueReleaseToGL(memory2, queue2)
}
```

UNEXPECTED SERIALIZATION



EXPECTED

UNEXPECTED SERIALIZATION



EXPECTED

ACTUAL

UNEXPECTED SERIALIZATION

What's going on?

```
while(1) {
     EnqueueAcquireFromGL(memory1, queue1)
     EnqueueWrite(memory1, queue1)
     EnqueueReleaseToGL(memory1, queue1)

     EnqueueAcquireFromGL(memory2, queue2)
     EnqueueRead(memory2, queue2)
     EnqueueReleaseToGL(memory2, queue2)
}
```

queue1 and queue2 are *OpenCL* queues and not necessarily backed by separate OpenGL queues, since the OGL context is the same

UNEXPECTED SERIALIZATION

What's going on?

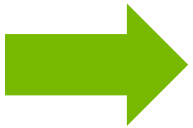
```
while(1) {
     EnqueueAcquireFromGL(memory1, queue1)
     EnqueueWrite(memory1, queue1)
     EnqueueReleaseToGL(memory1, queue1)

     EnqueueAcquireFromGL(memory2, queue2)
     EnqueueRead(memory2, queue2)
     EnqueueReleaseToGL(memory2, queue2)
}
```

False dependency!

UNEXPECTED SERIALIZATION

Fixing it

```
while(1) {
      EnqueueAcquireFromGL(memory1, queue1)
      EnqueueWrite(memory1, queue1)
      EnqueueReleaseToGL(memory1, queue1)

      EnqueueAcquireFromGL(memory2, queue2)
      EnqueueRead(memory2, queue2)
      EnqueueReleaseToGL(memory2, queue2)
}
```



```
while(1) {
      EnqueueAcquireFromGL(memory1, queue1)
      EnqueueAcquireFromGL(memory2, queue2)

      EnqueueWrite(memory1, queue1)
      EnqueueRead(memory2, queue2)

      EnqueueReleaseToGL(memory1, queue1)
      EnqueueReleaseToGL(memory2, queue2)
}
```

UNEXPECTED SERIALIZATION

Fixing it

```
while(1) {
      AcquireFromGL(memory1, queue1)
      Write(memory1, queue1)
      ReleaseToGL(memory1, queue1)

      AcquireFromGL(memory2, queue2)
      Read(memory2, queue2)
      ReleaseToGL(memory2, queue2)
}
```

False dependency!

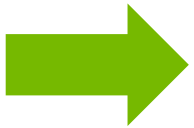
```
while(1) {
      AcquireFromGL(memory1, queue1)
      AcquireFromGL(memory2, queue2)

      Write(memory1, queue1)
      Read(memory2, queue2)

      ReleaseToGL(memory1, queue1)
      ReleaseToGL(memory2, queue2)
}
```

False dependency still exists between the OpenGL operations, but this dependency no longer separating heavyweight copy operations like before, so they're now free to overlap

UNEXPECTED SERIALIZATION

Fixing it

```
while(1) {
      AcquireFromGL(memory1, queue1)
      Write(memory1, queue1)
      ReleaseToGL(memory1, queue1)

      AcquireFromGL(memory2, queue2)
      Read(memory2, queue2)
      ReleaseToGL(memory2, queue2)
}
```



```
while(1) {
      AcquireFromGL(memory1, queue1)
      AcquireFromGL(memory2, queue2)

      Write(memory1, queue1)
      Read(memory2, queue2)

      ReleaseToGL(memory1, queue1)
      ReleaseToGL(memory2, queue2)
}
```

MORE EFFICIENT CL/GL SYNCHRONIZATION

MORE EFFICIENT CL/GL SYNCHRONIZATION

The problem

Applications need to segregate accesses from the two APIs

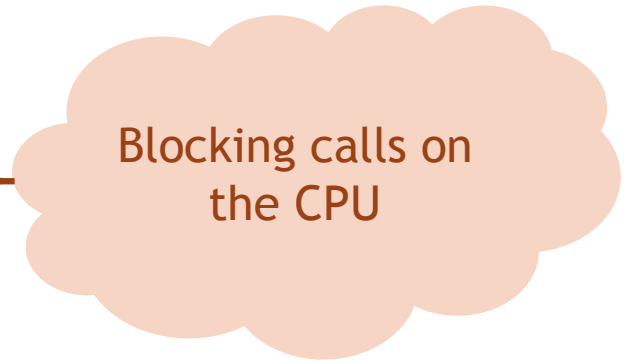
The only portable way to do this in the core OpenCL API is with clFinish()/glFinish() at each handover

This causes bubbles in the pipeline

MORE EFFICIENT CL/GL SYNCHRONIZATION

The problem

```
glFinish()
AcquireFromGL(mem)

doCLWork(mem)

ReleaseToGL(mem)
clFinish()

doGLWork()
```



MORE EFFICIENT CL/GL SYNCHRONIZATION

cl_khr_gl_event and GL_ARB_cl_event

These extensions provide better coordination between OpenCL and OpenGL by

1. offloading synchronization responsibility on to the OpenCL runtime, or
2. providing new calls to translate events of one API to a form waitable on by the other API. This is typically an advanced optimization strategy.

Heads-up: interop behaviour is different for single threaded and multi threaded use cases

MORE EFFICIENT CL/GL SYNCHRONIZATION

Single threaded application

Acquire and release calls are synchronous without any effort from the application

```
~~glFinish()~~
AcquireFromGL(mem)

doCLWork(mem)

ReleaseToGL(mem)
~~clFinish()~~

doGLWork()
```

This synchronization happens on the GPU
The CPU calls are non-blocking, freeing up the app to do other work while waiting for GPU work to be done

Also simplifies code

MORE EFFICIENT CL/GL SYNCHRONIZATION

Multi threaded application

OpenCL thread

OpenGL thread

```
                                                                    doGLWork()
                                                          glFence = createGLFence()

clEventFromGLFence = clCreateEventFromGLSyncKHR(glFence)
//param below is a dependency
clEnqueueAcquireGLObjects(clEventFromGLFence)
doCLWork()
clEvent = clEnqueueReleaseGLObjects()

                                        GLSyncFromCLEvent = CreateSyncFromCLeventARB(clEvent)
                                                                  glWaitSync(GLSyncFromCLEvent)
                                                                    doGLWork()
```

SHARED VIRTUAL MEMORY

SHARED VIRTUAL MEMORY

Address space is shared by host and all devices in a context

An address is “understood” the same way by host and all devices in a context

=> Programs can use pointer-containing structures such as graphs in device kernels

TYPES OF SHARED VIRTUAL MEMORY

COARSE-GRAINED BUFFER

Sharing happens at granularity of regions of OCL memory objects

Updates between host and devices happen explicitly, through map and unmap calls

FINE-GRAINED BUFFER

Sharing happens at granularity of bytes of OCL memory objects

Updates between host and device happen implicitly, with consistency maintained at synchronization points

FINE-GRAINED SYSTEM

Sharing happens at granularity of bytes anywhere in host memory

Updates between host and device happen implicitly, with consistency maintained at synchronization points

TYPES OF SHARED VIRTUAL MEMORUY

COARSE-GRAINED BUFFER

Sharing happens at granularity of regions of OCL memory objects

Updates between host and devices happen explicitly, through map and unmap calls

FINE-GRAINED BUFFER

Sharing happens at granularity of bytes of OCL memory objects

Updates between host and device happen implicitly, with consistency maintained at synchronization points

FINE-GRAINED SYSTEM

Sharing happens at granularity of bytes anywhere in host memory

Updates between host and device happen implicitly, with consistency maintained at synchronization points

TYPES OF SHARED VIRTUAL MEMORY

COARSE-GRAINED BUFFER

Sharing happens at granularity of regions of OCL memory objects

Updates between host and devices happen explicitly, through map and unmap calls

FINE-GRAINED BUFFER

Sharing happens at granularity of bytes of OCL memory objects

Updates between host and device happen implicitly, with consistency maintained at synchronization points

FINE-GRAINED SYSTEM

Sharing happens at granularity of bytes anywhere in host memory

Updates between host and device happen implicitly, with consistency maintained at synchronization points

SHARED VIRTUAL MEMORY

Fine Grained varieties and dGPU

Fine-grained SVM allows the same memory object to be shared across host and device

On a discrete GPU world, this means that one side has to pay the penalty of access over PCIe

This is bad for performance!

SHARED VIRTUAL MEMORY

Coarse Grain Buffer

- behaves exactly like regular OpenCL memory, but also
- allows host and devices to share pointer-containing data structures

SHARED VIRTUAL MEMORY

Coarse Grain Buffer – not magic

While the virtual address space is shared between the host and device, the physical address space need not necessarily be shared

This means that on a dGPU world, data will still need to be moved around between host and device just like regular buffers

SVM CGB is a great programming convenience for certain use-cases and allows richer algorithms, but it cannot magically reduce or eliminate existing data migration cost

SHARED VIRTUAL MEMORY

Performance Characteristics of SVM CGB on NVIDIA GPUs

Access latency of SVM CGB memory from the GPU is the same as that of regular buffers, for both clustered as well as sparse accesses

Cost of updation: updating SVM CGB buffers will cost only as much as the size of region being updated. Minimizing data traffic results in savings just as it would on regular buffers

API latency of SVM Map and Unmap calls will be comparable to regular Map and Unmap calls

Launch latency does not increase if SVM memory is used

SHARED VIRTUAL MEMORY

Performance Characteristics of SVM CGB on NVIDIA GPUs

The performance characteristics of SVM CGB and APIs affected by SVM CGB closely match that of regular memory

SUMMARY

EXECUTION

Use perf knobs in the API to tune programs

Waiting for completion can be efficient

DATA MOVEMENT

Copy can overlap with other work

Interop with OpenGL is more efficient with new features

Shared Virtual Memory = regular buffers + ability to have pointers

GPU TECHNOLOGY CONFERENCE

April 4-7, 2016 | Silicon Valley

THANK YOU

JOIN THE CONVERSATION

#GTC16

JOIN THE NVIDIA DEVELOPER PROGRAM AT developer.nvidia.com/join

PRESENTED BY

NVIDIA