A METHODOLOGY FOR OPTIMIZING DATA TRANSFER IN OPENCL™

Hervé CHEVANNE Dr. Ing.
AMD
SMTS
The OpenCL framework is divided into platform API and runtime API:

- The platform API:
  - Allows application to query for OpenCL devices
  - Manages OpenCL devices through a context

- The runtime API:
  - Makes use of contexts to manage the execution of kernels on OpenCL devices
**OPENCL MEMORY OBJECTS**

**Buffer Object**: A memory object that stores a linear collection of bytes. Buffer objects are accessible using a pointer in a *kernel* executing on a *device*. Buffer objects can be manipulated by the host using OpenCL API calls. A *buffer object* encapsulates the following information:

- Size in bytes.
- Properties that describe usage information and which region to allocate from.
- Buffer data.

**Image Object**: A *memory object* that stores a two- or three-dimensional structured array. Image data can only be accessed with read and write functions. The read functions use a *sampler*.

The *image object* encapsulates the following information:

- Dimensions of the image.
- Description of each element in the image.
- Properties that describe usage information and which region to allocate from.
- Image data.

The elements of an image are selected from a list of predefined image formats.

- **Opaque objects (2D or 3D)**
  - Can only be accessed via `read_image()` and `write_image()`.
  - Can either be read or written in a kernel, but not both.

- **Contiguous chunks of memory** stored sequentially and can be accessed directly (arrays, pointers, structures).
- **Read/write capable**
CREATING MEMORY OBJECTS

```c
cl_mem clCreateBuffer (cl_context context,
                       cl_mem_flags flags,
                       size_t size,
                       void *host_ptr,
                       cl_int *errcode_ret)
```

```c
cl_mem clCreateImage2D (cl_context context,
                        cl_mem_flags flags,
                        const cl_image_format *image_format,
                        size_t image_width,
                        size_t image_height,
                        size_t image_row_pitch,
                        void *host_ptr,
                        cl_int *errcode_ret)
```

```c
cl_mem clCreateImage3D (cl_context context,
                        cl_mem_flags flags,
                        const cl_image_format *image_format,
                        size_t image_width,
                        size_t image_height,
                        size_t image_depth,
                        size_t image_row_pitch,
                        size_t image_slice_pitch,
                        void *host_ptr,
                        cl_int *errcode_ret)
```
MEMORY FLAGS

- Memory flag field in `clCreateBuffer()` allows to define characteristics of the buffer object

<table>
<thead>
<tr>
<th>CL_MEM Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_MEM_READ_WRITE</td>
<td>Kernel can read and write to the memory object</td>
</tr>
<tr>
<td>CL_MEM_WRITE_ONLY</td>
<td>Kernel can write to memory object. Read from the memory object is undefined</td>
</tr>
<tr>
<td>CL_MEM_READ_ONLY</td>
<td>Kernel can only read from the memory object. Write from the memory object is undefined</td>
</tr>
<tr>
<td>CL_MEM_USE_HOST_PTR</td>
<td>Specifies to OpenCL implementation to use memory reference by host_ptr (4th arg) as storage object</td>
</tr>
<tr>
<td>CL_MEM_COPY_HOST_PTR</td>
<td>Specifies to OpenCL to allocate the memory and copy data pointed by host_ptr (4th arg) to the memory object</td>
</tr>
<tr>
<td>CL_MEM_ALLOC_HOST_PTR</td>
<td>Specifies to OpenCL to allocate memory from host accessible memory</td>
</tr>
</tbody>
</table>
TRANSFERRING DATA

```c
err = clEnqueueReadBuffer(
    command_queue,  // valid command queue
    output,         // memory buffer to read from
    CL_TRUE,        // indicate blocking read
    0,              // the offset in the buffer object to read from
    (sizeof(float) * DATA_SIZE), // size in bytes of data being read
    results,        // pointer to buffer in host mem to store read data
    0,              // number of event in the event list
    NULL,           // list of events that needs to complete before this executes
    NULL);          // event object to return on completion

err = clEnqueueWriteBuffer(
    command_queue,  // valid command queue
    input,          // memory buffer object to write to
    CL_TRUE,        // indicate blocking write
    0,              // the offset in the buffer object to write to
    (sizeof(float) * DATA_SIZE), // size in bytes of data to write
    host_ptr,       // pointer to buffer in host mem to read data from
    0,              // number of event in the event list
    NULL,           // list of events that needs to complete before this executes
    NULL);          // event object to return on completion
```
TRANSFERRING DATA (CONT.)

```c
err = clEnqueueReadImage (  
    command_queue, // valid command queue
    image, // valid image object to read from
    blocking_read, // blocking flag, CL_TRUE or CL_FALSE
    origin_offset, // (x,y,z) offset in pixels to read from z=0 for 2D image
    region, // (width,height,depth) in pixels to read from, depth=1 for 2D image
    row_pitch, // length of each row in bytes
    slice_pitch, // size of each 2D slice in the 3D image in bytes, 0 for 2D image
    host_ptr, // host memory pointer to store write image object data to
    num_events, // number of events in events list
    event_list, // list of events that needs to complete before this executes
    &event // event object to return on completion
 );

err = clEnqueueWriteImage (  
    command_queue, // valid command queue
    image, // valid image object to write to
    blocking_read, // blocking flag, CL_TRUE or CL_FALSE
    origin_offset, // (x,y,z) offset in pixels to write to z=0 for 2D image
    region, // (width,height,depth) in pixels to write to, depth=1 for 2D image
    row_pitch, // length of each row in bytes
    slice_pitch, // size of each 2D slice in the 3D image in bytes, 0 for 2D image
    host_ptr, // host memory pointer to store read data from
    num_events, // number of events in events list
    event_list, // list of events that needs to complete before this executes
    &event // event object to return on completion
 );
```

Host ← Device

Host → Device
**OPENCL PROFILING CAPABILITIES**

- The OpenCL runtime provides a built-in mechanism for timing the execution of kernels by setting the `CL_QUEUE_PROFILING_ENABLE` flag when the queue is created.

- The OpenCL runtime automatically records timestamp information for every kernel and memory operation submitted to the queue.
### EVENT PROFILING INFORMATION

```c
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**

<table>
<thead>
<tr>
<th>Profiling Data</th>
<th>Return Type</th>
<th>Information Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_PROFILING_COMMAND_QUEUED</td>
<td>cl_ulong</td>
<td>A 64-bit counter in nanoseconds when the command is enqueued in a command queue</td>
</tr>
<tr>
<td>CL_PROFILING_COMMAND_SUBMIT</td>
<td>cl_ulong</td>
<td>A 64-bit counter in nanoseconds when the command that has been enqueued is submitted to the compute device for execution</td>
</tr>
<tr>
<td>CL_PROFILING_COMMAND_START</td>
<td>cl_ulong</td>
<td>A 64-bit counter in nanoseconds when the command started execution on the compute device.</td>
</tr>
<tr>
<td>CL_PROFILING_COMMAND_END</td>
<td>cl_ulong</td>
<td>A 64-bit counter in nanoseconds when the command has finished execution on the compute device.</td>
</tr>
</tbody>
</table>
**USING EVENT PROFILING IN OPENCL**

```c
myCommandQ = clCreateCommandQueue (... , CL_QUEUE_PROFILING_ENABLE, NULL);
...
cl_event myEvent;
cl_ulong startTime, endTime;
clEnqueueNDRangeKernel(myCommandQ,
    ...
    &myEvent);
...
clFinish(myCommandQ); // wait for all events to finish
clGetEventProfilingInfo(myEvent,
    CL_PROFILING_COMMAND_START,
    sizeof(cl_ulong),
    &startTime,
    NULL);
clGetEventProfilingInfo(myEvent,
    CL_PROFILING_COMMAND_END,
    sizeof(cl_ulong),
    &endTime,
    NULL);
cl_ulong kernelExecTime = endTime - startTime;
```
MEASURING ELAPSED TIME IN LINUX®: CLOCK_GETTIME

Name

clock_gettime - Return the current timespec value of tp for the specified clock

Synopsis

int clock_gettime(clockid_t clk_id, struct timespec *tp);

Description

The function clock_gettime() retrieve the time of the specified clock clk_id.

All implementations support the system-wide realtime clock, which is identified by
CLOCK_REALTIME. Its time represents seconds and nanoseconds since the Epoch.

CLOCK_REALTIME

System-wide realtime clock. Setting this clock requires appropriate privileges.
QueryPerformanceCounter Function
Retrieves the current value of the high-resolution performance counter.

Syntax
BOOL WINAPI QueryPerformanceCounter( __out LARGE_INTEGER *lpPerformanceCount );

Parameters
lpPerformanceCount [out]
Type: LARGE_INTEGER*
A pointer to a variable that receives the current performance-counter value, in counts.
IMPLEMENTATION ON LINUX® AND WINDOWS®

```c
void TimerStart(void)
{
    #ifdef _WIN32
        QueryPerformanceCounter((LARGE_INTEGER *) &start);
        QueryPerformanceFrequency((LARGE_INTEGER *) &freq);
    #else
        struct timespec s;
        assert(clock_gettime(CLOCK_REALTIME, &s) == CL_SUCCESS);
        start = (i64)s.tv_sec * 1e9 + (i64)s.tv_nsec;
        freq = 1000000000;
    #endif
}

void TimerReset(void)
{
    iclock = 0;
}

void TimerStop(void)
{
    i64 n;
    #ifdef _WIN32
        QueryPerformanceCounter((LARGE_INTEGER *) &n);
    #else
        struct timespec s;
        assert(clock_gettime(CLOCK_REALTIME, &s) == CL_SUCCESS);
        n = (i64)s.tv_sec * 1e9 + (i64)s.tv_nsec;
    #endif
    n -= _start;
    start = 0;
    iclock += n;
}

double GetElapsedTime(void)
{
    return (double)iclock / (double)freq;
}
```
COPYING DATA HOST → DEVICE
COPYING DATA HOST → DEVICE THE “NATURAL” WAY

- Transfer “size” Bytes from the CPU to the GPU using a NULL pointer:
  
  ```
  hostMem = malloc(size);
  cl_mem_flags flags = CL_MEM_READ_WRITE;
  cl_mem buffer = clCreateBuffer(context, flags, size, NULL, &err);
  int err = clEnqueueWriteBuffer(commandQueue, buffer, CL_TRUE, 0, size, hostMem, 0, NULL, NULL);
  ```

- Transfer “size” Bytes from the CPU to the GPU using a memory pointer (CL_MEM_USE_HOST_PTR):
  
  ```
  hostMem = malloc(size);
  cl_mem_flags flags = CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR;
  cl_mem buffer = clCreateBuffer(context, flags, size, hostMem, &err);
  int err = clEnqueueWriteBuffer(commandQueue, buffer, CL_TRUE, 0, size, hostMem, 0, NULL, NULL);
  ```
COPYING DATA HOST → DEVICE

CL_MEM_READ_WRITE flag

Buffers aligned on page boundaries

// 1st case: NULL_ptr;
printf("1 - Testing NULL_ptr:\n---------------------\n");
// Allocate device memory
cl_mem_flags flags = CL_MEM_READ_WRITE;
for (int sizeCount=0; sizeCount < NSIZES; sizeCount++)
{
#ifdef _WIN32
    unsigned char* hostMem = (unsigned char*) _aligned_malloc (memSize[sizeCount],
    pageSize);
    unsigned char* validMem = (unsigned char*) _aligned_malloc (memSize[sizeCount],
    pageSize);
#else
    unsigned char* hostMem = (unsigned char*) memalign(
    pageSize, memSize[sizeCount]);
    unsigned char* validMem = (unsigned char*) memalign(
    pageSize, memSize[sizeCount]);
#endif
}
for (int iterCount=0; iterCount < NITERS; iterCount++)
{
    // Create buffer on the GPU
    devBuffer = clCreateBuffer(_deviceContext, flags, memSize[sizeCount], NULL, &err);
    assert(err == CL_SUCCESS);

    // Generate a random value in [0,7] range, but different from the previous one
    do{
        value_old = value;
        value = (unsigned char) rand() % 8;
    } while (value_old == value);

    // Initialize arrays in host space with new values
    for (int i=0; i<memSize[sizeCount]; i++)
    {
        hostMem[i] = value;
        validMem[i] = zero;
    }
}
COPYING DATA HOST → DEVICE (CONT.)

// Initialize device memory
cl_event* my_events = (cl_event*) malloc((numIter[iterCount]+1)*sizeof(cl_event));
err = clEnqueueWriteBuffer(_commandQueue, devBuffer, CL_TRUE, 0,
    memSize[sizeCount], hostMem, 0, NULL,
    &my_events[0]);

assert(err == CL_SUCCESS);
err = clEnqueueWaitForEvents(_commandQueue, 1, &my_events[0]);

TimerReset();
TimerStart();
for(int i=0;i<numIter[iterCount];i++)
{
    err = clEnqueueWriteBuffer(_commandQueue, devBuffer, CL_TRUE, 0,
        memSize[sizeCount], hostMem, 1, &my_events[i],
        &my_events[i+1]);

    assert(err == CL_SUCCESS);
}

// Wait for dust to settle
clFinish(_commandQueue);
TimerStop();
double Time = GetElapsedTime();
double rate = (double(memSize[sizeCount]/MB) * double(numIter[iterCount])) / Time;
COPYING DATA HOST → DEVICE (CONT.)

//Check if the transfers went OK
err = clEnqueueReadBuffer(_commandQueue, devBuffer, CL_TRUE, 0,
  memSize[sizeCount], validMem, 0, NULL, NULL);
assert(err == CL_SUCCESS);
err = CL_SUCCESS;
for (int i=0; i<memSize[sizeCount]; i++)
  if(validMem[i] != value) err = -10;
if(err != CL_SUCCESS)
  printf("Warning: the transfer is not valid\n");
assert(clReleaseMemObject(devBuffer) == CL_SUCCESS);
cFinish(_commandQueue);

#ifdef _WIN32
  _aligned_free(hostMem);
  _aligned_free(validMem);
#else
  free(hostMem);
  free(validMem);
#endif

Check the data transferred to the GPU

Synchronization

Release memory on the GPU and on the host
**TEST CONFIGURATION**

- Fujitsu Celsius M470 workstation
  - 2 Intel Xeon X5550 (2.66GHz)
  - 6GB of DDR3 memory
  - OpenSuSE 11.2 / gcc 4.4.1
    - fglrx 8.832-110310a-115047E-ATI
  - Windows 7 Professional / VS 2008
    - fglrx 8.841-110405a-116675E
  - SDK 2.4
  - ATI FirePro™ V9800
    Professional Graphics (Cypress)
COPYING DATA HOST → DEVICE - PERFORMANCE – LINUX®

Bandwidth (Mbytes/s)

Buffer size (Bytes/s)

1 iteration
10 iterations
100 iterations
COPYING DATA HOST → DEVICE - PERFORMANCE – WINDOWS®7

Bandwidth (Mbytes/s)

Buffer size (Bytes/s)

1 iteration

10 iterations

100 iterations
COPYING DATA HOST → DEVICE - THE MAP/UNMAP WAY

- Map a “size” Bytes long memory area of the GPU into the CPU address space
- CL_MEM_USE_HOST_PTR + CL_MEM_USE_PERSISTENT_MEM_AMD:

```c
hostMem = malloc(size);

cl_mem_flags flags = CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR | CL_MEM_USE_PERSISTENT_MEM_AMD;

cl_mem buffer = clCreateBuffer(context, flags, size, NULL, &err);

void *mem = clEnqueueMapBuffer(commandQueue, buffer, CL_TRUE, CL_MAP_READ, 0, size, 0, NULL, NULL, &err);

memcpy(mem, hostMem, size);

er = clEnqueueUnmapMemObject(commandQueue, buffer, mem, 0, NULL, NULL);
```
void *mem;

TimerReset();
TimerStart();
for(int i=0;i<numIter[iterCount];i++)
{
    mem = clEnqueueMapBuffer(_commandQueue, devBuffer, CL_TRUE,
        CL_MAP_READ, 0, memSize[sizeCount], 0,
        NULL, NULL, &err);
    assert(err == CL_SUCCESS);
    memcpy(mem, hostMem, memSize[sizeCount]);
    assert(CL_SUCCESS == clEnqueueUnmapMemObject (_commandQueue,devBuffer,
      mem, 0, NULL, NULL));
}
// Wait for dust to settle
clFinish(_commandQueue);
TimerStop();

double Time = GetElapsedTime();
double rate = (double(memSize[sizeCount]/MB) * double(numIter[iterCount])) / Time;
COPYING DATA HOST → DEVICE - MAP/UNMAP PERFORMANCE – LINUX®

Bandwidth (Mbytes/s)

Buffer size (Bytes/s)

1 iteration
10 iterations
100 iterations
COPYING DATA HOST → DEVICE - MAP/UNMAP PERFORMANCE – WINDOWS®7

Bandwidth (Mbytes/s) vs. Buffer size (Bytes/s)

- 1 iteration
- 10 iterations
- 100 iterations

Buffer size (Bytes/s)
READING DATA DEVICE → HOST
READING DATA DEVICE $\rightarrow$ HOST THE TEST CODE

```c
// Initialize device memory
err = clEnqueueWriteBuffer(_commandQueue, devBuffer, CL_TRUE, 0,
                          memSize[sizeCount], hostMem, 0, NULL, NULL);
assert(err == CL_SUCCESS);
clFinish(_commandQueue);
cl_event* my_events = (cl_event*) malloc((numIter[iterCount]+1)*sizeof(cl_event));

TimerReset();
TimerStart();
for(int i=0;i<numIter[iterCount];i++)
{
   err = clEnqueueReadBuffer(_commandQueue, devBuffer, CL_TRUE, 0,
                            memSize[sizeCount], validMem, 1, &my_events[i],
                            &my_events[i+1]);
   assert(err == CL_SUCCESS);
}

// Wait for dust to settle
clFinish(_commandQueue);
TimerStop();
double Time = GetElapsedTime();
double rate = (double(memSize[sizeCount]/MB) * double(numIter[iterCount])) / Time;
```

Synchronous transfer

Synchronization

Bandwidth assessment
READING DATA DEVICE → HOST - PERFORMANCE – LINUX®

Bandwidth (Mbytes/s)

Buffer size (Bytes/s)

- 1 iteration
- 10 iterations
- 100 iterations
READING DATA DEVICE → HOST - PERFORMANCE – WINDOWS®7

Bandwidth (Mbytes/s)

Buffer size (Bytes/s)

1 iteration

10 iterations

100 iterations
void *mem;
TimerReset();
TimerStart();
for(int i=0;i<numIter[iterCount];i++)
{
    mem = clEnqueueMapBuffer(_commandQueue, devBuffer, CL_TRUE,
    CL_MAP_WRITE, 0, memSize[sizeCount], 0,
    NULL, NULL, &err);
    assert(err == CL_SUCCESS);
    memcpy(validMem, mem, memSize[sizeCount]);
    assert(CL_SUCCESS == clEnqueueUnmapMemObject (_commandQueue,devBuffer,
    mem, 0, NULL, NULL));
}
// Wait for dust to settle
clFinish(_commandQueue);
TimerStop();

double Time = GetElapsedTime();
double rate = (double(memSize[sizeCount]/MB) * double(numIter[iterCount])) / Time;
READING DATA DEVICE → HOST - MAP/UNMAP PERFORMANCE – LINUX®

Bandwidth (Mbytes/s)

Buffer size (Bytes/s)

1 iteration
10 iterations
100 iterations

0 500 1000 1500 2000 2500 3000

1K 2K 4K 8K 16K 32K 64K 128K 256K 512K 1M 2M 4M 8M 16M 32M 64M 128M 256M 512M 1G
READING DATA DEVICE → HOST - MAP/UNMAP PERFORMANCE – WINDOWS®7

Bandwidth (Mbytes/s)

Buffer size (Bytes/s)

- 1 iteration
- 10 iterations
- 100 iterations
SUMMARY AND CONCLUSIONS

- Use an appropriate timer (i.e. monotonic and accurate),
- Warm-up the GPU before making measurements,
- Ensure the system is quite and increase the priority of the job,
- Performance behavior depends on:
  - The version of the driver,
  - The version of SDK,
  - The operating system,
  - The amount of data transferred,
  - The nature of the transfer (upload vs. read back, buffer vs. image, …),
  - The system memory configuration,
  - The mother-board,
  - …
QUESTIONS
Disclaimer & Attribution
The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. There is no obligation to update or otherwise correct or revise this information. However, we reserve the right to revise this information and to make changes from time to time to the content hereof without obligation to notify any person of such revisions or changes.

NO REPRESENTATIONS OR WARRANTIES ARE MADE WITH RESPECT TO THE CONTENTS HEREOF AND NO RESPONSIBILITY IS ASSUMED FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

ALL IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT WILL ANY LIABILITY TO ANY PERSON BE INCURRED FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREA, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD, the AMD arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. All other names used in this presentation are for informational purposes only and may be trademarks of their respective owners.
OpenCL is a trademark of Apple Inc. used by permission of Khronos.
Linux is a registered trademark of Linus Torvalds.
Windows is a registered trademark of Microsoft Corporation.

© 2011 Advanced Micro Devices, Inc.