AMD Accelerated Parallel Processing
TECHNOLOGY

SAMPLE

BufferBandwidth

1 Overview

1.1 Location

`$(AMDAPPSDKSAMPLESROOT)\samples\opencl\cl\app`

1.2 How to Run

See the *Getting Started* guide for how to build samples. You first must compile the sample.

Use the command line to change to the directory where the executable is located. The pre-compiled sample executable is at `$(AMDAPPSDKSAMPLESROOT)\samples\opencl\bin\x86\` for 32-bit builds, and `$(AMDAPPSDKSAMPLESROOT)\samples\opencl\bin\x86_64\` for 64-bit builds.

Type the following command(s).

1. `BufferBandwidth`
   This runs the program with the default options: -T 0, -i 1,-r 1, -k 10, -x 1048576 (1MB), -s 0, -w 7, -l 0, -O 0, -C 5, -C 0.
2. `BufferBandwidth -h`
   This prints the help file.

1.3 Command Line Options

Table 1 lists, and briefly describes, the command line options.

Table 1 Command Line Options

Short Form	Long Form	Description
`-h`	`--help`	Shows all command options and their respective meaning.
	`--device`	Devices on which the program is to be run. Acceptable values are cpu or gpu.
`-q`	`--quiet`	Quiet mode. Suppresses all text output.
`-e`	`--verify`	Verify results against reference implementation.
`-t`	`--timing`	Print timing.
	`--dump`	Dump binary image for all devices.
	`--load`	Load binary image and execute on device.
`-d`	`--deviceId`	Select deviceId to be used (0 to N-1, where N is the number of available devices).
	`--flags`	Specify compiler flags to build the kernel.
`-p`	`--platformId`	Select platformId to be used (0 to N-1, where N is the number of available platform).
`-x`	`--size`	Size in bytes.
`-i`	`--iterations`	Number of timing loops.
`-s`	`--skip`	Skip the first n interations for average.
`-k`	`--kernelLoops`	Number of loops in the kernel.

Short Form	Long Form	Description
`-w`	`--wavefronts`	Number of wavefronts per compute unit.
`-I`	`--inMemFlag`	Memory flags for the input buffer. 0 CL_MEM_READ_ONLY 1 CL_MEM_WRITE_ONLY 2 CL_MEM_READ_WRITE 3 CL_MEM_USE_HOST_PTR 4 CL_MEM_COPY_HOST_PTR 5 CL_MEM_ALLOC_HOST_PTR 6 CL_MEM_USE_PERSISTENT_MEM_AMD
`-r`	`--repeats`	Repeat each timing n times.
`-T`	`--TypeOfTest`	Type of test. 0 clEnqueue[Map,Unmap] 1 clEnqueue[Read/Write] 2 clEnqueueCopy
`-O`	`--outMemFlag`	Memory flags for the output buffer. Values are the same as for option `-I`.
`-C`	`--copyMemFlag`	Memory flags for the copy buffer. Values are the same as for option `-I`.
`-m`	`--mapping`	Always maps as `MAP_READ | MAP_WRITE`.
`-D`	`--disable`	Disable host mem bandwidth baseline.
`-l`	`--log`	Prints complete timing log.

2 Introduction

This sample measures a complete round trip loop of data transfer steps to, and from, an OpenCL device. It also assesses the bandwidth characteristics of a given system, including GPU memory and interconnect (for example: PCIe) bandwidth, achievable in OpenCL.

It can run the following tests:

- Create a simple baseline for host memory read and write performance. This can be used to ensure sanity of device buffer access performance numbers created by the other tests.
- Benchmark a round-trip chain of synchronous, serialized transfer steps between the host and the device.
- The sample can create a log over many iterations to locate one-time effects or variations over time.

The following transfer paths can be tested via command line option:

```
clEnqueueMap/UnmapBuffer
clEnqueueRead/WriteBuffer
clEnqueueCopyBuffer
```

This sample allows selection of any of the various CL buffer creation attributes for the source and destination buffers of the transfer chain.

3 Implementation Details

The following are experiments that can be done with this sample.

1. Interconnect (for example: PCIe) bandwidth achievable at application level

The bandwidth reported by `clEnqueueUnmapMemObject()` on a regular device buffer that was mapped as `CL_MAP_WRITE` usually is very close to the interconnect peak bandwidth to the GPU. Similarly, the bandwidth reported for `clEnqueueMapBuffer` (`CL_MAP_READ`) of a device buffer usually is very close to the interconnect peak bandwidth from the GPU. Running `BufferBandwidth` without command line arguments shows both.

2. Optimized paths for `CL_MAP_WRITE` and `CL_MAP_READ`

   The CL runtime tries to omit unnecessary copies between host and device when buffers are mapped as either `CL_MAP_READ` or `CL_MAP_WRITE`, but not both. A buffer mapped as `CL_MAP_WRITE` is transferred at `clEnqueueUnmapMemObject()` time. A buffer mapped as `CL_MAP_READ` is transferred at `clEnqueueMapBuffer()` time.

3. Zero copy buffers

   If supported on the platform, the following two buffer types show zero copy behavior (meaning they are not transferred without explicit request by the application):

   CL_MEM_ALLOC_HOST_PTR

   – Can be directly accessed across the interconnect bus by a GPU CL kernel.

   – Can be directly accessed by the host at host memory bandwidth.

   – Can be copied to, and from, a GPU device buffer at interconnect peak bandwidth through `clEnqueueCopyBuffers`.

   – Map and unmap are low cost.

   Try this path using `BufferBandwidth -I 5 -O 5`

   CL_MEM_USE_PERSISTENT_MEM_AMD

   – Can be accessed by the GPU like a regular device buffer.

   – Can be written to by the host at interconnect peak bandwidth.

   – Can be directly read by the host, but typically at low bandwidth.

   – Can be copied at high bandwidth from the device to the host through `clEnqueueCopyBuffer`.

   – Map and unmap are low cost.

   Try this path using `BufferBandwidth -I 6 -O 6`

   Zero copy buffers are useful for sparse accesses across the interconnect bus; they can be used to get around DMA start-up latency. Zero copy buffers of type `CL_MEM_USE_PERSISTENT_MEM_AMD` allow use of the CPU for transfer between the host and the device; they also allow overlap transfers with GPU kernel execution. Zero copy buffers of type `CL_MEM_ALLOC_HOST_PTR` type allow inclusion of transfer latency directly in the GPU kernel execution, and use of the GPU shader engines to perform the transfer. If zero copy is not supported, these buffers fall back to a meaningful, but lower-performing, default behavior.

4. Recommended fast paths

   The recommended paths to achieve peak transfer bandwidth are:

   – map and unmap of device buffers using `CL_MAP_READ` or `CL_MAP_WRITE`, as described in 2) above.

   – `clEnqueueCopyBuffers` from a zero copy `ALLOC_HOST_PTR` buffer to a device buffer.

5. The `-w` option permits optimizing the bandwidth of the GPU read and write kernels on a given platform.

4 Notes and Caveats

All transfer steps are executed synchronously to ensure accurate bandwidth measurement. The application code should not follow this model, but submit as many commands to a CL queue as possible before forcing the queue to drain.

- Do not run graphics applications while benchmarking compute or transfer operations.
- The `-l` option can be used to identify some of the one-time costs that exist for a given transfer chain. For instance, during the first 1 or 2 iterations, the GPU and CPU achieve maximum clock rates. Also, buffers are allocated and transported to their final location. These costs show up as increased execution times for the first few OpenCL calls.
- The read and write GPU kernels are written for clarity, and should achieve around 85% of HW peak with the right number of threads.
- The CPU baseline does not represent absolute host memory peak, as it is executed single threaded.
- The data verification used is basic.
- The smallest supported buffer size in this sample is 1024 bytes, corresponding to a single wave front. Buffer sizes supplied by `-x` are adjusted to a multiple of 1024 bytes.

Contact

Advanced Micro Devices, Inc.
One AMD Place
P.O. Box 3453
Sunnyvale, CA, 94088-3453
Phone: +1.408.749.4000

For AMD Accelerated Parallel Processing:
URL: developer.amd.com/appsdk
Developing: developer.amd.com/
Support: developer.amd.com/appsdksupport
Forum: developer.amd.com/openclforum