TELEDYNE SP DEVICES
Everywhereyoulook™

ADQ7 GPU Peer-To-Peer

User Guide

Author(s): Teledyne SP Devices
Document ID: 19-2241
Classification: Public
Revision: PA1
Print date: 2021-11-17

Contents

1 Introduction

This document describes how to perform peer-to-peer transfer from an ADQ7 digitizer to a GPU.

1.1 Definitions and Abbreviations

Table 1 lists the definitions and abbreviations used in this document and provides an explanation for each entry.

Table 1: Definitions and abbreviations used in this document.

Term	Explanation
API	Application programming interface
DMA	Direct memory access
GPU	Graphics processing unit (Graphics card)
OCT	Optical coherence tomography
P2P	Peer-to-peer
UL2	User logic 2—open FPGA area in the ADQ7 firmware.

2 Prerequisites

Hardware

- ADQ7 digitizer
- Peer-to-peer capable GPU
  - **Windows** AMD GPU with DirectGMA support
  - **Linux** Nvidia GPU with GPUDirect support
- Host computer capable of P2P streaming with two PCIe Gen3x8 slots

3 Working Principles

The peer-to-peer function for ADQ7 has been developed with special consideration for OCT applications and the triggering scheme commonly used in these applications. Data is collected with the help of two trigger signals: the *A-scan* trigger and the *B-scan* trigger.

3.1 Trigger and Data Alignment

A data collection is specified by a record length *M* and a line length. Where each line consists of *N* records. The A-scan trigger starts collection of a record. Each record is consecutively written directly to

GPU memory. In the normal case a line therefore takes up $M \times N$ samples of memory. When a line is full records are continued at the next line immediately after the last.

A B-scan trigger indicates the start of a new line. At the detection of a B-scan trigger the next record will be written at the start of the next line. If a B-scan trigger arrives before N records have been written this means one or several records are missing. In this case the line is marked as invalid but the next line will automatically be properly aligned in memory as if the previous line had been fully written.

Fig. 1 shows an example of the memory layout after a successful data collection. Each record is represented by a dash and corresponding A- and B-triggers are labeled. In this example the line length N would be four, meaning four A-triggers are followed by one B-trigger. At the right hand side of the data buffer, the *data valid buffer* is shown containing the number one for each valid line.

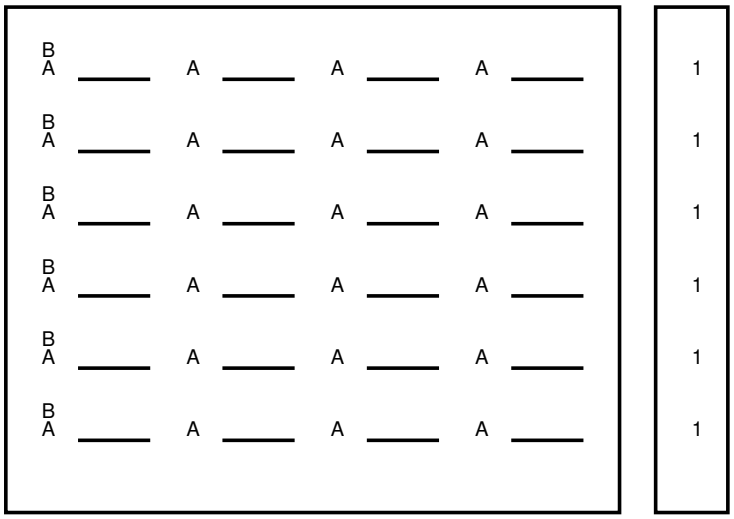

Figure 1: Records in GPU memory with labels indicating triggers. The data valid buffer indicates that all lines are valid.

Information about invalid lines is written as metadata to a separate part of the memory in the GPU. Fig. 2 shows the GPU memory with several A-scans missing, the B-scan has moved the following record to the next line and the data valid buffer indicates that the second line is not valid. The digitizer will write a zero to the buffer only when an invalid line is encountered, the buffer is expected to initially be filled with ones.

The resulting data buffer in GPU memory will always contain consistent data aligned in records and lines according to the trigger information received by the ADQ7, ready to be processed by the GPU.

3.2 Double Buffering and Kernel Scheduling

The example program employs a double buffering scheme when transferring data to GPU. The digitizer will write to one buffer and signal the host when the buffer is completely filled. At that signal the host may schedule processing of the data in the first buffer. Simultaneously the digitizer can start writing data to a second buffer. When data processing of the first buffer has completed the second buffer can be passed to the GPU and the process repeats again.

This method ensures that there is always a buffer available for data transfer, avoiding any wait time. Transfer rates above 7 GB/s have been measured using the attached example code.

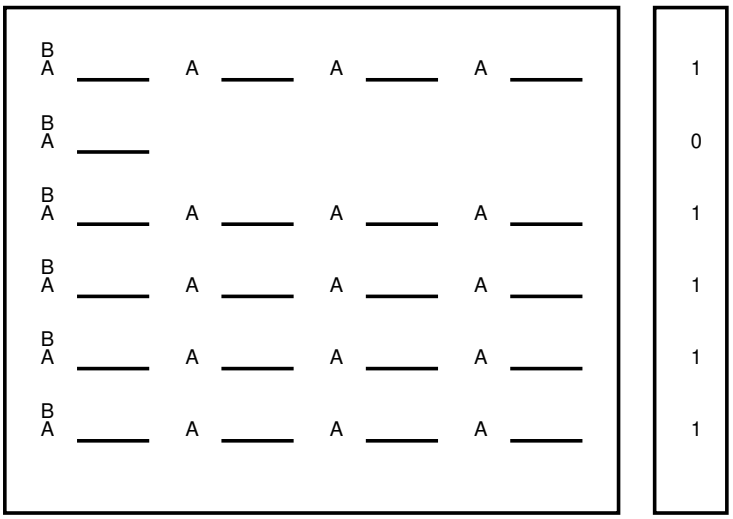

Figure 2: Records in GPU memory where several A-scans are missing, indicated in the data valid buffer.

3.3 Backplane Peer-To-Peer Transfer

Data is written from the digitizer directly to the GPU without going through the host CPU or host memory. This reduces requirement on the host system significantly while still utilizing the full transfer rate capability of the PCIe back plane. The ADQ7 supports up to eight PCIe generation 3.0 lanes.

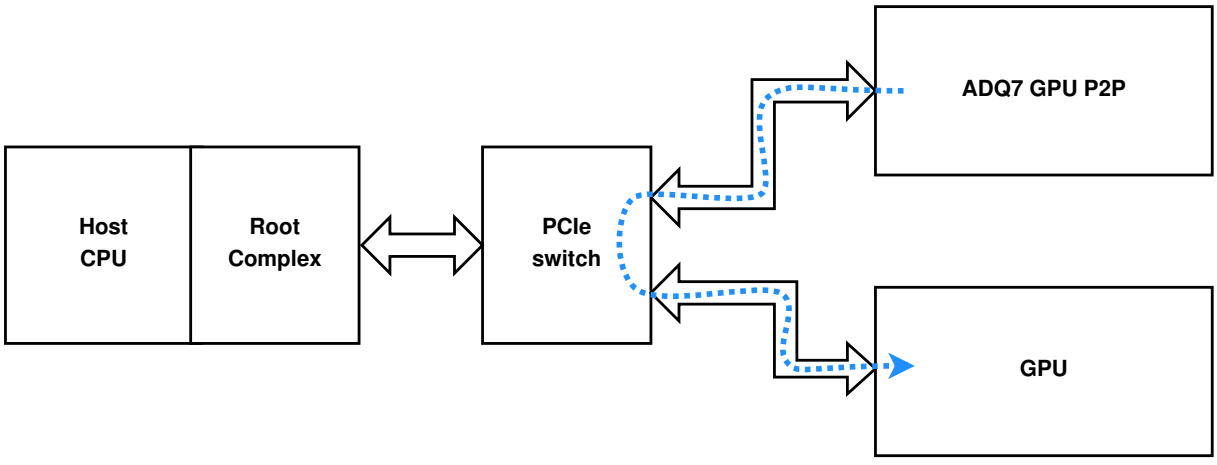

Figure 3: Peer-to-peer transfer from the digitizer to the GPU.

Fig. 3 shows transfer from a digitizer to a GPU. Note that transfer takes place via a PCIe switch. While peer-to-peer transfer is sometimes supported directly through the root complex without involving a switch it is not a mandatory function of the PCIe standard. Therefore, make sure that there is a PCIe switch between the two endpoints or that the root complex supports peer-to-peer transfer.

4 Setting up an Application

This section outlines the general steps for setting up P2P GPU streaming. Detailed information is given where deemed necessary. It is recommended to study example files `main.c` which shows all the listed steps and `gpu_streaming_defines.h` which contains macros for calculating buffer sizes, data access, and settings. Information about ADQ functions is found in the ADQAPI reference guide [1].

- Choose parameters
- Decide if B-scan and data valid buffers should be used
- Initialize GPU driver
- Allocate and pin buffers in GPU
- Initialize ADQ
- Set up triggers
- Set up P2P GPU with `SetupDMAP2p2D()`
- Start streaming
- Wait for a completed buffer
- Detect and handle overflows
- Process received data and reset the data valid buffer
- Stop streaming
- User logic 2 considerations

4.1 Choosing Parameters

Streaming parameters should be chosen to at least satisfy required conditions. N is an integer $2 \leq N \leq 2^{22}$ and Ch is the number of channels used: 1 or 2.

1. Record length = N $\times$32 / Ch, required
2. A-scans per B-scan $\geq$ 2, required
3. Samples per B-scan $\leq 2^{31}$, required
4. 1 $\leq$ B-scans per buffer $\leq 2^{12}$, required
5. Samples per buffer $\leq 2^{33}$, required
6. N/2 $\times$Records per buffer = Integer, recommended

Condition 6 is recommended for minimum latency at buffer completion. See `gpu_streaming_defines.h` for macros to calculate correct buffer sizes from parameters. Please note that the GPU may impose other restrictions on buffer size. The average transfer speed should also be considered:

$$\text{Transfer speed [MB/s]} = \text{A-scans/s} \times \text{record length} \times \text{Ch} \times 10^{-6}$$

If the maximum throughput of the system is exceeded more than momentarily the ADQs buffer may overflow and data will be lost. Fig. 4 presents the throughput measured for a test setup and may be used as a rough guideline for maximum throughput. GPU load may affect throughput so it is recommended to determine the maximum throughput of the system with data processing active.

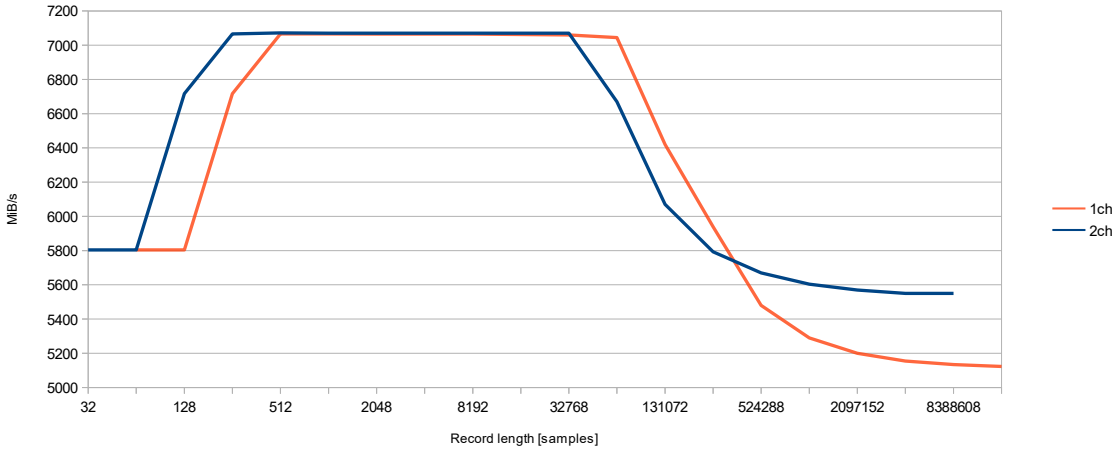

Figure 4: Maximum throughput in test setup for different record lenghts.

4.2 B-scan and Data Valid

If no B-scan signal is connected the system will behave as if a correct B-scan is present, no writes to data valid buffer should occur. If no data valid buffer is specified, please set data valid size to 0 in the call to `SetupDMAP2p2D()`. This blocks any unintentional B-scan trigger from initializing a write to an invalid address.

4.3 Set up P2P GPU with `SetupDMAP2p2D()`

The input of marker addresses is vendor specific and described bellow, For general information see the ADQAPI reference guide [1].

4.3.1 Nvidia

Marker memory is allocated by the ADQAPI. The marker address fields can be left empty.

4.3.2 AMD

The DMA buffers allocated with OpenCL has one marker address each. Since the application only needs 2 markers in total, use the marker addresses associated with the data buffers.

4.4 Wait for a Completed Buffer

Two markers are used to detect completed buffers. The marker associated with data and data valid buffers 0 will only take on odd values and the marker associated with data and data valid buffers 1 will take on even values. The initial marker value is 1 and the maximum marker value is $2^{32} - 1$ the succeeding marker value will be 0.

4.4.1 Nvidia

Completed buffers are detected by host. Call function `WaitforGPUMarker()` which will return as soon as a buffer is completed (since last function call) or when the specified timeout is reached. `WaitforGPUMarker()` will return the latest marker value via argument `marker_list`. The completed data and data valid buffers can be determined as:

$$\text{completed buffers} = (\text{returned marker} + 1) \mod 2$$

It is also recommended to compare the returned marker value with the expected marker value to detect missed buffers which typically means that streaming is faster than the application.

4.4.2 AMD

Completed buffers are detected by GPU. The function `clEnqueueWaitSignalAMD()` is used to make GPU wait until the marker associated with the specified buffer is equal to or greater than the specified value. Operations added to the queue after a `clEnqueueWaitSignalAMD()` will not be performed until the `clEnqueueWaitSignalAMD()` operation is finished. When operations are added to the queue they can be connected to a `cl_event`. Use function `clWaitForEvents()` to make host wait for one or several events. It should be noted that in some systems `clWaitForEvents()` only returns when all `clEnqueueWaitSignalAMD()` calls are finished.

4.5 Detect and Handle Overflows

It is recommended to frequently check for overflows with ADQ function `GetStreamOverflow()`, no detected overflows means that all data since last successful `GetStreamOverflow()` call is correct. If an overflow is detected streaming has to be restarted by calling ADQ functions `StopStreaming()`, `SetupDMAP2p2D()` and `StartStreaming()`. Other than loss of data overflow may cause unexpected change in throughput or a total halt of streaming.

4.6 Process Received Data and Reset the Data Valid Buffer

The samples are written in `int16_t` format into buffers. In two-channel mode samples from channel A and B are interleaved. Macros in `gpu_streaming_defines.h` shows how to access a given sample in a

buffer. After a set of buffers are completed user application has to reset data valid buffers by writing an 32-bit variable with value 1 in each position.

4.7 User Logic 2 Considerations

- Metadata insertion and sample interleaving for two-channel mode is done in UL2.
- If UL2 is bypassed GPU streaming will not work.
- Writing to UL2 registers may alter functionality.
- If UL2 is modified with the development kit, make sure that metadata insertion still behaves correctly.

5 Example Code

The ADQ7 GPU P2P function is delivered with two examples, one for Nvidia GPU's under Linux and another for AMD GPU's under Windows.[1]

5.1 Signal Connections

The example code can collect data from one or two analog inputs using the A-trigger and optional B-trigger as described in Section 3.1. Table 2 shows how the different input signals are labeled on the ADQ7 backplate.

Table 2: Signal connections for ADQ7 device

Signal	ADQ7 input
Analog channel A	A
Analog channel B	B (optional)
A-trigger	Trig
B-trigger	Sync (optional)

5.2 Nvidia Example

The example for Nvidia GPUs is written using GPUDirect with CUDA and OpenGL.

GPUDirect works by setting up a bus writable buffers in GPU memory. Additionally, *marker* buffers are setup in host memory used to synchronize writes to the GPU buffers with host program execution. When a GPU buffer has been completely written by the digitizer it writes an iterator to the associated marker buffer in host and sends an interrupt. The host program waits for the marker write and enqueues a CUDA kernel at that time. This sequence is repeated in an alternating pattern for two buffers.

[1] Nvidia and AMD are protected trademarks of their respective owners.

The example uses a CUDA kernel which computes an FFT for each record in the buffer, and draws a point to an OpenGL buffer indicating the peak frequency of the FFT. Fig. 5 shows a screen capture of the CUDA example.

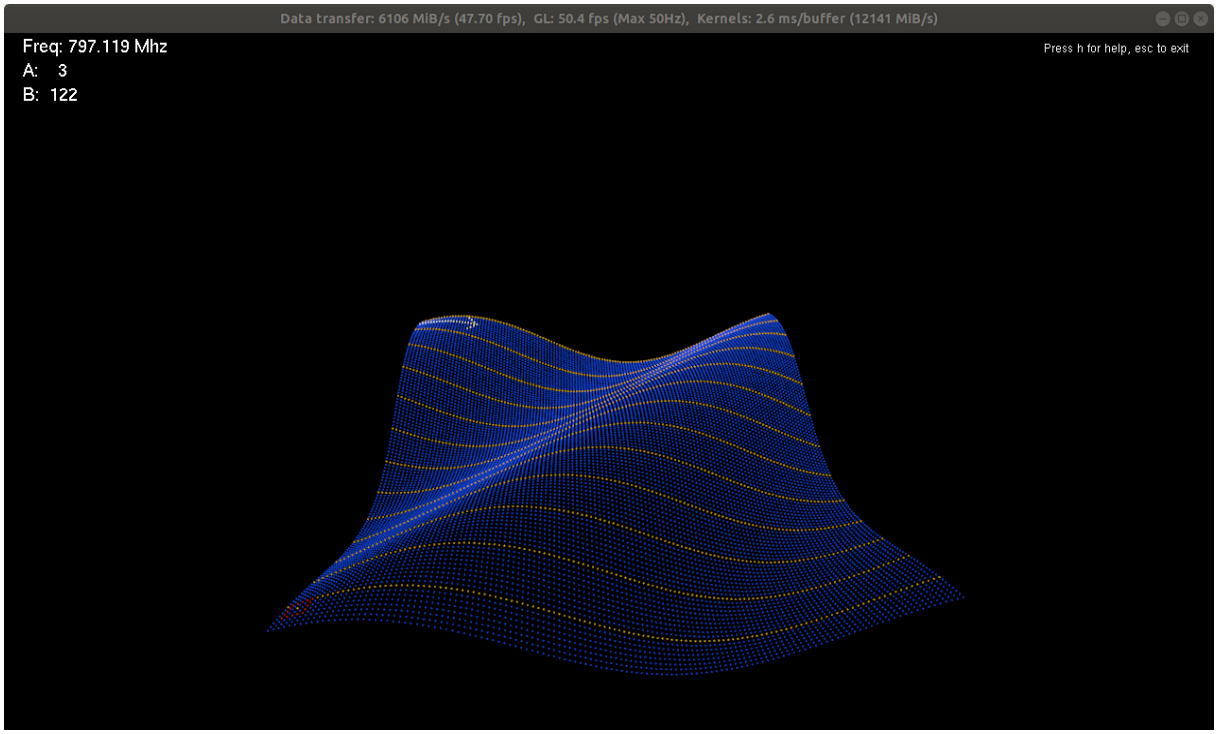

Figure 5: Screen capture from the CUDA example.

5.2.1 Running the Example

1. Make sure CUDA toolkit including examples are installed
2. Make sure the current user is in the 'adq' group: `groups`
3. Make sure the ADQ7 device driver is loaded: `ls /dev/adq*`
4. Go to `ADQ7_GPUDirect_example/source/gdrdrv`
5. Build kernel module for GPUDirect: `make`
6. Load kernel module: `sudo ./insmod.sh`
7. Go to directory `ADQ7_GPUDirect_example/source`
8. Build example: `make`
9. Run example: `./cuda_example`

Once the example is running press `h` for information about mouse and keyboard controls and `i` for information.

5.2.2 Adjusting Example Settings

The file `gpu_streaming_defines.h` contains macros for settings and debug printouts. When two-channel mode is active (`STREAM_CHANNELS = 2`) the channel used for generating graphics can be chosen by replacing `GET_SAMPLE_CH_A` macro in function `cast_and_validate_kernel` located in `OCT_func.cu`. Remember to rebuild example (`make`) for the changes to take effect.

5.2.3 Known Bugs

1. Resizing of the window causes a high number re-scaling events leading to a temporary slowdown of the GPU, this can be avoided by freezing the frame (`f`) before resizing.
2. Transfer speed is calculated from buffer size, the size of skipped samples because of invalid lines is not subtracted, thus the figure is only 100% accurate when all lines are valid.

5.3 AMD Example

The example for AMD GPUs is written using DirectGMA with OpenCL. DirectGMA works by setting up a bus writable buffers with *markers* in GPU memory. Data is written into a buffer and when it is full an iterator is written to the associated marker. A wait command is added to the OpenCL queue which blocks the queue until a marker write is detected. At that time a kernel enqueued after the marker wait can run. This sequence is repeated in an alternating pattern for two buffers. The enqueued operations copy the content of the data and `data_valid` buffers to another set of buffers and reinitializes data valid buffer.

5.3.1 Running the example

1. Make sure the ADQ7 device driver is installed.
2. Make sure AMD driver is installed and directGMA activated
3. Make sure Visual studio 2017 or never is installed
4. Install OCL-SDK: OCL-SDK installer
5. Go to directory examples/amd/example
6. Open visual studio project file and build release x64
7. Run example: `ADQ7_DirectGMA_example`
8. For more details see included README.

5.3.2 Adjusting Example Settings

The file `gpu_streaming_defines.h` contains macros for settings and debug printouts. Remember to rebuild example for the changes to take effect.

References

[1] Teledyne Signal Processing Devices Sweden AB, *14-1351 ADQAPI Reference Guide*. Technical Manual.

TELEDYNE SP DEVICES
Everywhereyoulook™

Worldwide Sales and Technical Support
spdevices.com

Teledyne SP Devices Corporate Headquarters
Teknikringen 6
SE-583 30 Linköping
Sweden
Phone: +46 (0)13 645 0600
Fax: +46 (0)13 991 3044
Email: spd_info@teledyne.com