ATS-GPU Programmer's Guide

Release 3.5.0

AlazarTech

Sep 18, 2017

Contents

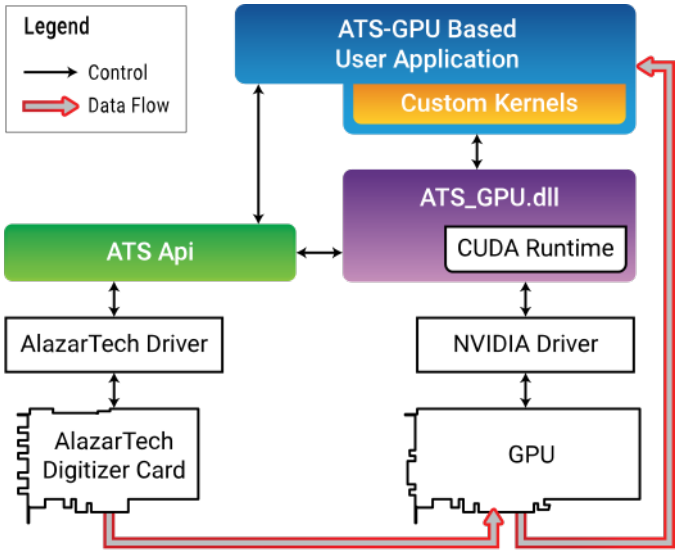
Introduction

The ATS-GPU SDK provides a framework to allow real-time processing of data from AlazarTech PCIe digitizers on a CUDA compatible GPU. This programmer's guide covers the use of ATS-GPU-BASE and ATS-GPU-OCT optional signal processing library.

This document assumes that the reader is familiar with ATS-SDK, the standard interface for programming AlazarTech digitizers. Having a copy of the ATS-SDK manual available can be helpful, since many references to ATSApi functions are done here. The latest version of the ATS-SDK manual can be downloaded free of charge from AlazarTech's website (www.alazartech.com).

In addition, expertise in CUDA programming is assumed. This is particularly important for users wishing to use ATS-GPU-BASE directly, because this task involves CUDA programming.

It is also essential for programmers to have in-depth knowledge of GPU architecture and parallel programming.



Prerequisites

System requirements

This software requires a PC with a CUDA-compatible GPU, and sufficient CPU resources to supply data to the GPU at the desired data acquisition rate. It was tested with Geforce GTX Titan X (Maxwell), Geforce GTX980 and Quadro P5000. DDR4 memory and a modern chipset (X99, X299) will greatly improve transfer speed and overall performance.

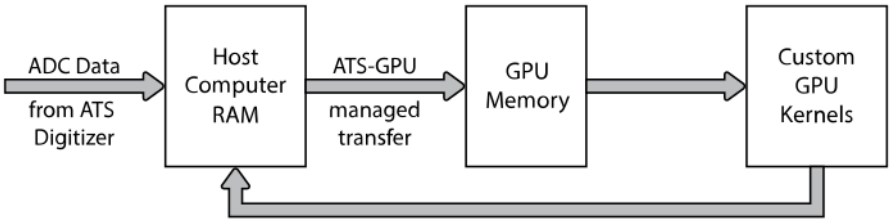
Supported operating systems Windows and Linux operating systems are supported. Please verify that your Linux distribution is supported by NVIDIA which supplies the CUDA toolkit required to use ATS-GPU.

Compiler support The C++ code was written with Microsoft Visual C++ 2015, and requires Microsoft Visual C++ 2015 or later. Please note that a Community Edition of Visual Studio is available for free. It is fully compatible with our code samples. CMake can also be used to build C++ code. CMake files are provided. A C++11 compiler is required to build the library. On older Red Hat distributions, a devtoolset can be obtained to use a more recent version of gcc that supports C++11. NVCC is required to compile the example code, this compiler is included with CUDA toolkit.

ATS-GPU-BASE

ATS-GPU-BASE is designed to provide highly efficient code to transfer data from an ATS digitizer to a GPU for processing. This transfer is done using multiple DMA transactions. The user application, which includes custom CUDA kernels, can then access data buffers on the GPU. The user is then responsible to perform data processing and copy data back to the CPU if required. A code example is provided as an example of a user application that performs very simple signal processing (data inversion).

Usage



ATS-GPU Data Flow

ATS-GPU-BASE offers several functions that behave similarly to ATSApi functions. Please refer to the ATS-SDK guide for more details about these APIs. Obtaining a board handle and configuring the board (sampling rate, trigger, input channels, etc.) is performed directly using functions from the ATS-SDK. By convention, the code samples define a `ConfigureBoard()` function that handles all these tasks.

```
if (!ConfigureBoard(boardHandle)) {
    // Error handling
}
```

During the lifetime of an application, multiple acquisitions can take place. If the board configuration parameters do not change, it is not necessary to call `ConfigureBoard()` again.

The next step is to select the CUDA-enabled GPU to use for the data transfer. This call is optional. If you only have one CUDA capable GPU on your computer, you can skip it.

```
rc = ATS_GPU_SetCUDAComputeDevice(boardHandle, deviceIndex);
// Error handling
```

We must then setup parameters of the acquisition to GPU. This function replaces the call to `AlazarBeforeAsyncRead()` in normal programs. Parameters were kept as close as possible to those of `AlazarBeforeAsyncRead()` to ease transition between standard acquisitions and ATS-GPU acquisitions. To maximize performance, sample interleave should be enabled with `ADMA_INTERLEAVE_SAMPLES`.

```
rc = ATS_GPU_Setup(boardHandle, channelSelect, transferOffset,
                   transferLength, recordsPerBuffer, recordsPerAcquisition,
                   autoDMAFlags, ATSGPUFlags);
// Error handling
```

We then allocate memory on the GPU for data to be transferred to, and we post those buffers to the board. For this purpose, we use `ATS_GPU_AllocBuffer()`. This function allocates a buffer on the GPU, and sets up all the intermediary state necessary for ATS-GPU to successfully transfer data. Please note that if you'd like to send data back from the GPU to your computer's RAM after having processed it, you will need to allocate memory independently of the AlazarTech APIs.

```
for (size_t i = 0; i < buffers_to_allocate; i++)
{
    buffers[i] = ATS_GPU_AllocBuffer(boardHandle, bytesPerBuffer);
```

```
    rc =  ATS_GPU_PostBuffer(boardHandle,
                             buffers[i],
                             bytesPerBuffer);
    // Error handling
}
```

We can then start the acquisition. The board will directly start acquiring data, assuming it receives triggers, and data transfer to posted GPU buffers will also start immediately.

```
rc =  ATS_GPU_StartCapture(HANDLE boardHandle);
// Error handling
```

Once acquisition is started, `ATS_GPU_GetBuffer()` must be called as often as possible to retrieve a buffer containing data already copied on the GPU. This buffer can then be processed by your custom kernel on the GPU. When a buffer is done being used (either data has been copied to a different buffer or processing is complete), the buffer needs to be posted back to the board.

```
for (size_t i; i < buffers_per_acquisition; i++)
{
    rc =  ATS_GPU_GetBuffer(boardHandle,
                            buffers[i],
                            timeout_ms,
                            nullptr);

    // TODO: Error handling

    // TODO: Process buffer. This is where you can call your own processing
    //       function that launches the GPU kernels, such as ProcessBuffer()
    //       in the code samples.
    ProcessBuffer(buffers[i], bytesPerBuffer);

    rc = ATS_GPU_PostBuffer(boardHandle, buffer, bytesPerBuffer);
}
```

When acquisition is complete, `ATS_GPU_AbortCapture()` must be called. Buffers allocated with `ATS_GPU_AllocBuffer()` should then be freed with `ATS_GPU_FreeBuffer()`.

```
RETURN_CODE ATS_GPU_AbortCapture(HANDLE boardHandle);

for (size_t i = 0; i < number_of_buffers; i++)
{
   rc = ATS_GPU_FreeBuffer(boardHandle, buffers[i]);
   // Error handling
}
```

Here is an example of what the function to process data on the GPU can look like. Since this contains code that is executed on the GPU, it needs to be located in a file with a `.cu` extension:

```
extern “C”__global__ void ProcessBuffer(void* buffer, bytesPerBuffer)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
```

```
    // TODO: Do processing here
}

Bool ProcessBuffer(void* buffer, U32 bytesPerBuffer)
{
    // Launch ProcessBuffer CUDA kernel
    ProcessBuffer<<<threadsPerBlock, BlocksPerGrid>>>(buffer, bytesPerBuffer);

    // Copy result from GPU memory to CPU memory
    cudaMemcpy(resultBuffer,buffer,bytesPerBuffer);
}
```

Performance guidelines

While GPU solutions are highly customizable and can reach very high processing speeds, care must be taken to preserve performance. The provided libraries use streams to maximise concurrency and hide latency associated with data transfers. The processing functions are optimized to provide the best performance and modifying them can result in a loss of performance. Refer to the CUDA best practices guide for more information on how to improve performance.

Warning: When developing CUDA code, it is very important to check memory accesses with a dedicated tool, as bad memory accesses will not necessarily trigger an error but will lead to bad behavior and can cause a crash. The CUDA toolkit provides the necessary memory checking utilities.

Because data is DMA'd from ATS board to host memory then to GPU memory, speed of host computer memory will influence performance. DDR4 memory and a modern chipset (X99, X299, etc.) will greatly improve transfer speed and overall performance.

Benchmarks

Performance benchmarks using the optional OCT signal processing library and NVIDIA GeForce GTX Titan X (Maxwell) GPU on an Asus X99 Deluxe motherboard with an Intel i7 5930K 3.5 GHz CPU, and 2133 MHz DDR4 memory (64 GB RAM):

PCIe Link Speed	Transfer Rate
Gen 3: ATS9373	Up to 4 GB/s
Gen 2: ATS9360, ATS9416	Up to 3 GB/s
Gen 1: ATS9870, ATS9350, ATS9351, ATS9625, ATS9626, ATS9440	Up to 1.6 GB/s
Gen 1: ATS9462	Up to 720 MB/s

API Reference

RETURN_CODE **ATS_GPU_Setup**(HANDLE *boardHandle*, U32 *channelSelect*, long *transferOffset*, U32 *transferLength*, U32 *recordsPerBuffer*, U32 *recordsPerAcqusition*, U32 *autoDMAFlags*, U32 *ATSGPUFlags*)

Prepares the ATS board and GPU for acquisition.

This function calls AlazarBeforeAsyncRead internally and most parameters are passed directly to it. In addition, it sets up the GPU for DMA transfers

Parameters

- `boardHandle` - Handle to the board. Set to `NULL` for data validation mode.
- `channelSelect` - Channel mask with each channel identifier OR'd
- `transferOffset` - pass a negative integer for pretrigger samples
- `transferLength` - Number of samples in a record or transfer
- `recordsPerBuffer` - Number of records in a buffer, 1 for triggered streaming and continuous streaming modes.
- `recordsPerAcquisition` - Total number of records in the acquisition. Pass 0x7FFFFFF for infinite.
- `autoDMAFlags` - ATSApi flags for AlazarBeforeAsyncRead
- `ATSGPUFlags` - ATS-GPU specific flags

RETURN_CODE **ATS_GPU_Setup**(HANDLE *boardHandle*, U32 *channelSelect*, long *transferOffset*, U32 *transferLength*, U32 *recordsPerBuffer*, U32 *recordsPerAcqusition*, U32 *autoDMAFlags*, U32 *ATSGPUFlags*)

Prepares the ATS board and GPU for acquisition.

This function calls AlazarBeforeAsyncRead internally and most parameters are passed directly to it. In addition, it sets up the GPU for DMA transfers

Parameters

- `boardHandle` - Handle to the board. Set to `NULL` for data validation mode.
- `channelSelect` - Channel mask with each channel identifier OR'd
- `transferOffset` - pass a negative integer for pretrigger samples
- `transferLength` - Number of samples in a record or transfer
- `recordsPerBuffer` - Number of records in a buffer, 1 for triggered streaming and continuous streaming modes.
- `recordsPerAcquisition` - Total number of records in the acquisition. Pass 0x7FFFFFF for infinite.
- `autoDMAFlags` - ATSApi flags for AlazarBeforeAsyncRead
- `ATSGPUFlags` - ATS-GPU specific flags

void *__ATS_GPU_AllocBuffer__(HANDLE *boardHandle*, U32 *bytesPerBuffer*, void **reserved*)

Allocates page-aligned pinned memory for ATS and GPU boards.

This function can be called after ATS_GPU_Setup to perform the necessary memory allocations. This function returns a GPU or CPU buffer pointer depending on ATSGPUFlags used in the setup.

Parameters

- `boardHandle` - Handle to the board
- `bytesPerBuffer` - Total number of bytes to allocate per buffer

RETURN_CODE **ATS_GPU_PostBuffer**(HANDLE *boardHandle*, void **buffer*, U32 *bytesPerBuffer*)

Signal the library a particular buffer can be used for data transfer.

This function is the equivalent of AlazarPostAsyncBuffer for ATS_GPU. Buffers posted must have previously been allocated with ATS_GPU_AllocBuffers.

Parameters

- `boardHandle` - Hande to the board
- `buffer` - Pointer to a previously allocated buffer
- `bytesPerBuffer` - Size in bytes of the buffer, must be the same size as setup for the acquisition.

RETURN_CODE **ATS_GPU_GetBuffer**(HANDLE *boardHandle*, void **buffer*, U32 *timeout_ms*, void **reserved*)

Get processed buffer.

This function must be called at average rate that is equal to or greater than the rate at which DMA buffers complete. This function returns the GPU-processed buffer.

Return `ApiSuccess` (512) if the board received sufficient triggers to fill a DMA buffer.

Return `ApiNotInitialized` if `ATS_StartCapture` was not called before calling this function, or it was called and failed.

Return `ApiInvalidHandle` The `boardHandle` parameter is not valid.

Return `ApiBufferOverflow` if the board filled all the available DMA buffers and its on-board memory. This may happen if the acquisition rate exceeds the bus bandwidth or the GPU processing bandwidth.

Return `ApiWaitTimeout` if the timeout interval expired before the board received a sufficient number of triggers to fill a buffer.

Return `ApiFailed` if a system of internal error occured.

Parameters

- `boardHandle` - Handle to the board
- `buffer` - Pointer to the buffer
- `timeout_ms` - Time the board will wait for a trigger before throwing an error.
- `reserved` - Reserved for future use. Pass NULL.

RETURN_CODE **ATS_GPU_AbortCapture**(HANDLE *boardHandle*)

Stops the acquisition.

Aborts an acquisition, stops data processing, and releases resources allocated by ATS_GPU_Setup()

Return ApiSuccess

Parameters

- `boardHandle` - Handle to the board

RETURN_CODE **ATS_GPU_FreeBuffer**(HANDLE *boardHandle*, void **buffer*)

Free buffers allocated with ATS_GPU_AllocBuffers();.

Parameters

- `boardHandle` - Handle to the board
- `buffer` - Buffer pointer allocated by ATS_GPU_AllocBuffers()

RETURN_CODE **ATS_GPU_QueryCUDADeviceCount**(U32 **pDeviceCount*)

Function to get the number of available CUDA devices.

Return `ApiSuccess` it it succeeded.

Return `ApiFailed` if the OpenCL driver returned an error.

Parameters

- `pDeviceCount` - Outputs the number of devices detected on the system.

RETURN_CODE **ATS_GPU_QueryCUDADeviceName**(U32 *deviceIndex*, char **deviceName*, int *maxChars*)

Function to get the name of a specific CUDA device.

Return `ApiSuccess` it it succeeded.

Return `ApiFailed` if the OpenCL driver returned an error.

Return `ApiInvalidIndex` if the index provided is greater than the number of platforms or devices available.

Parameters

- `deviceIndex` - 0-based index to the device.
- `deviceName` - Char array to output the name of the device.
- `maxChars` - Size of the char array.

RETURN_CODE **ATS_GPU_SetCUDAComputeDevice**(HANDLE *boardHandle*, U32 *deviceIndex*)

CUDA-specific function used to associate a CUDA-enabled GPU device with a digitizer board.

Allows you to specify which GPU should be used to process sample data from a digitizer, if more than one GPU is available.

Return `ApiSuccess` if it succeeded.

Return `ApiFailed` if it failed. See `%TEMP%/ATS_GPU.log` (`/tmp/ATS_GPU.log` under Linux) for more information.

Parameters

- `boardHandle` - Handle to the ATS board.
- `deviceIndex` - 0-based index to the CUDA device.

Index

A