ATS-GPU-BASE

Version 4.1.0
June 17, 2020

AlazarTech

CONTENTS

CHAPTER

ONE

LICENSE AGREEMENT

1.1 Important

CAREFULLY READ THIS SOFTWARE LICENSE AGREEMENT. BY CLICKING THE APPLICABLE BUTTON TO COMPLETE THE INSTALLATION PROCESS, YOU AGREE TO BE BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT WISH TO BECOME A PARTY TO THIS AGREEMENT AND BE BOUND BY ITS TERMS AND CONDITIONS, DO NOT INSTALL OR USE THE SOFTWARE, AND RETURN THE SOFTWARE (WITH ANY ACCOMPANYING MEDIA) WITHIN THIRTY (30) DAYS OF RECEIPT. ALL RETURNS TO ALAZAR TECHNOLOGIES INC. ("ALAZARTECH") WILL BE SUBJECT TO ALAZARTECH'S THEN-CURRENT POLICY. IF YOU ARE ACCEPTING THESE TERMS ON BEHALF OF AN ENTITY, YOU AGREE THAT YOU HAVE AUTHORITY TO BIND THE ENTITY TO THESE TERMS.

1.2 Ownership

AlazarTech retains the ownership of ATS-GPU software ("Software"). It is licensed to you for use under the following conditions:

1.2.1 Grant of License

You may only concurrently use the Software on the computers that have an AlazarTech waveform digitizer card plugged in (for example, if you have purchased one AlazarTech card, you have a license for one concurrent usage). If the number of users of the Software exceeds the number of AlazarTech cards you have purchased, you must have a reasonable process in place to assure that the number of persons concurrently using the Software does not exceed the number of AlazarTech cards purchased.

This license is non-transferable.

1.2.2 Restrictions

You may not copy the documentation or Software except as described in the installation section of the Software manual. You may not distribute, rent, sub-lease or lease the Software or documentation, including translating or decomposing. You may not modify, reverse-engineer, decompile, or disassemble any part of the Software or documentation, or produce any derivative work other than software applications that communicate with AlazarTech hardware using the published Application Programming Interface (API).

You may not remove, block, or modify any titles, logos, trademarks, copyright and/or patent notices, digital watermarks, disclaimers, or other legal notices that are included in the Software.

1.2.3 Termination

This license and your right to use this Software automatically terminates if you fail to comply with any provision of this license agreement.

1.3 Rights

AlazarTech retains all rights not expressly granted. Nothing in this agreement constitutes a waiver of AlazarTech's rights under the Canadian and U.S. copyright laws or any other Federal or State law.

1.4 Limited Warranty

Although AlazarTech has tested the Software and reviewed the documentation, ALAZARTECH MAKES NO WARRANTY OF REPRESENTATION, EITHER EXPRESSED OR IMPLIED, WITH RESPECT TO THIS SOFTWARE OR DOCUMENTATION, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS SOFTWARE AND DOCUMENTATION IS LICENSED "as is" AND YOU, THE LICENSEE, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE. IN NO EVENT WILL ALAZARTECH BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS SOFTWARE OR DOCUMENTATION, even if advised of the possibility of such damages. In particular, AlazarTech shall have no liability for any data acquired, stored or processed with this Software, including the costs of recovering such data.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESSED OR IMPLIED. No AlazarTech dealer, agent or employee is authorized to make any modifications or additions to this warranty.

Information in this document is subject to change without notice and does not represent a commitment on the part of AlazarTech. The Software described in this document is furnished under this license agreement. The Software may be used or copied only in accordance with the terms of the agreement.

Some jurisdictions do not allow the exclusion of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights, which vary from jurisdiction to jurisdiction.

CHAPTER

TWO

INTRODUCTION

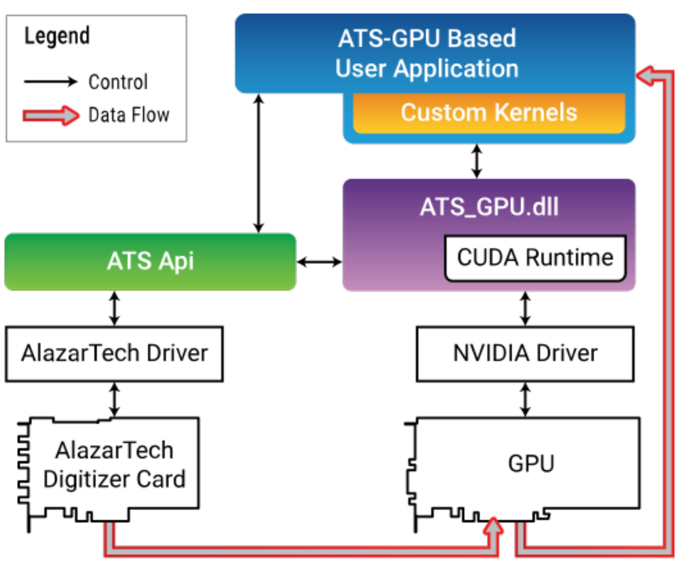
The ATS-GPU SDK provides a framework to allow real-time processing of data from AlazarTech PCIe digitizers on a CUDA-enabled GPU. This programmer's guide covers the use of ATS-GPU-BASE.

ATS-GPU-BASE internally calls ATS-CUDA, which is a wrapper library for simple CUDA calls. ATS-CUDA is described in more detail later in this guide in the section *ATS-CUDA*.

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.

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

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



CHAPTER

THREE

PREREQUISITES

3.1 System requirements

This software requires a PC with a CUDA-enabled 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 64-bit Windows and 64-bit 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. On Linux, 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.

CUDA driver requirements In order to use ATS-GPU, you must install the appropriate driver for your CUDA-enabled GPU. Drivers can be downloaded at https://www.nvidia.com/Download/index.aspx.

3.2 Programming experience

Users who wish to use ATS-GPU-BASE to create high-performance custom kernels must have expertise in CUDA programming.

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

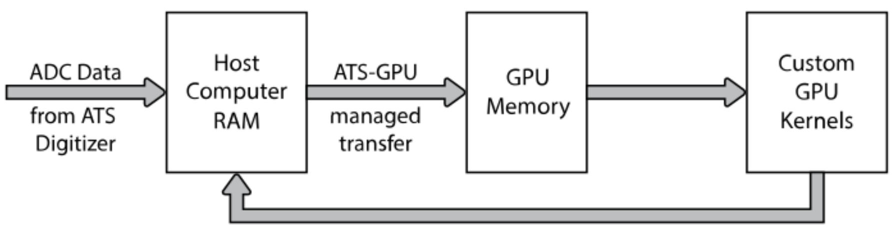
CHAPTER

FOUR

ATS-GPU-BASE

ATS-GPU-BASE is designed to provide highly efficient code to transfer data from an ATS PCIe digitizer to a CUDA-enabled 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).

4.1 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 would 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

```

(continues on next page)

(continued from previous page)

```
    // 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);
}
```

4.2 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.

4.3 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 i9-7900X 3.3 GHz CPU, and 2133 MHz DDR4 memory (32 GB RAM):

PCIe Link Speed	Transfer Rate
Gen 3x8: ATS9373, ATS9371	Up to 6.9 GB/s
Gen 2x8: ATS9360, ATS9416	Up to 3.5 GB/s
Gen 2x4: ATS9352 Gen 1x8: ATS9870, ATS9350, ATS9351, ATS9625, ATS9626, ATS9440	Up to 1.6 GB/s
Gen 1x4: ATS9462	Up to 720 MB/s
Gen 1x1: ATS9146, ATS9130, ATS9120	Up to 200 MB/s

4.4 API Reference

RETURN_CODE **ATS_GPU_Setup**(HANDLE *boardHandle*, U32 *channelSelect*, long *transferOffset*, U32 *transferLength*, U32 *recordsPerBuffer*, U32 *recordsPerAcquisition*, 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 `0x7FFFFFFF` for infinite.
- `autoDMAFlags`: ATSApi flags for AlazarBeforeAsyncRead
- `ATSGPUFlags`: Combination of elements from *ATS_GPU_SETUP_FLAG* OR'd together. Pass 0 for default

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 *ATS_GPU_SETUP_FLAG* values used in the setup.

Parameters

- `boardHandle`: Handle to the board
- `bytesPerBuffer`: Total number of bytes to allocate per buffer
- `reserved`: Reserved value. Pass `NULL`

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`: Handle 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 occurred.

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_GetVersion**(U8 **major*, U8 **minor*, U8 **revision*)

Get ATS-GPU version number.

Parameters

- `major`: ATS-GPU major version number.
- `minor`: ATS-GPU minor version number.
- `revision`: ATS-GPU revision number.

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

Function to get the number of available CUDA devices.

Return `ApiSuccess` if it succeeded.

Return `ApiFailed` if the GPU 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` if it succeeded.

Return `ApiFailed` if the GPU 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.

enum ATS_GPU_SETUP_FLAG

GPU data transfer configuration options.

Values:

ATS_GPU_SETUP_FLAG_CPU_BUFFER = 0x1

Makes ATS-GPU deliver CPU buffers instead of GPU ones. Useful for debugging

ATS_GPU_SETUP_FLAG_MAPPED_MEMORY = 0x2

Can only be used with `ATS_GPU_SETUP_FLAG_CPU_BUFFER`. Makes the API map the CPU buffers returned to GPU buffers.

ATS_GPU_SETUP_FLAG_DEINTERLEAVE = 0x4

De-interleave the data in the returned GPU buffer. Does not apply in conjunction with `ATS_GPU_SETUP_FLAG_CPU_BUFFER`

ATS_GPU_SETUP_FLAG_UNPACK = 0x8

Unpack the data in the returned GPU buffer. It is required for the allocated buffers to be large enough to accommodate unpacked data. Does not apply in conjunction with `ATS_GPU_SETUP_FLAG_CPU_BUFFER`

CHAPTER

FIVE

ATS-CUDA

The ATS-CUDA SDK provides a framework to allow users to perform simple manipulations on CUDA-enabled GPUs. ATS-CUDA is designed to be used with ATS-GPU-BASE, but can also be used independently. This section of the programmer's guide covers the use of ATS-CUDA.

As with ATS-GPU-BASE, using ATS-CUDA requires expertise in CUDA programming because this involves writing custom CUDA kernels.

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

5.1 API Reference

enum ALAZAR_PACKING

Types of data packing.

Values:

_16_bits_per_sample

_12_bits_per_sample

_8_bits_per_sample

struct UNPACK_DEINTERLEAVE_OPTIONS

Structure used to set up unpacking and deinterleaving kernel used in ATS_CUDA_BaseProcessBuffer().

Public Members

bool **unpack**

Flag to activate unpacking;.

bool **deinterleave**

Flag to activate deinterleaving.

U32 **transferLength**

Number of samples per record per channel.

U32 **recordsPerBuffer**

Number of records per buffer per channel.

U32 **channelCount**

channelCount Number of active channels

ALAZAR_PACKING **input_pack_mode**

A member of ALAZAR_PACKING indicating the data packing mode of input buffer

ALAZAR_PACKING **output_pack_mode**

A member of ALAZAR_PACKING indicating the desired output data packing. Ignored if unpack is set to 0.

void ***ATS_CUDA_AllocCPUBuffer**(U32 *bytesPerBuffer*)

Allocates page-locked memory on the host computer.

This function is used to allocate host memory and is accessible to the device. Memory can be accessed directly by the device and can be written or read at high bandwidth.

Return This function returns a CPU buffer pointer.

Parameters

- `bytesPerBuffer`: Total number of bytes to allocate per buffer void ***ATS_CUDA_AllocGPUBuffer**(U32 *bytesPerBuffer*)

Allocates memory on the device.

This function is used to allocate memory on the device.

Return This function returns a GPU buffer pointer.

Parameters

- `bytesPerBuffer`: Total number of bytes to allocate per buffer

RETURN_CODE **ATS_CUDA_BaseProcessBuffer**(void **GPUBufferIn*, void **GPUBufferOut*, cudaStream_t *stream*, *UNPACK_DEINTERLEAVE_OPTIONS opt*)

Launches on the GPU a kernel to unpack and/or deinterleave a buffer acquired with an AlazarTech digitizer.

Parameters

- `GPUBufferIn`: Pointer to a GPU buffer to on which to apply unpacking/deinterleaving.
- `GPUBufferOut`: Pointer to a GPU buffer where data is to be outputted.
- `stream`: Stream identifier on which processing is to take place
- `opt`: Structure that defines how the unpacking and deinterleaving kernel is to be configured. See *UNPACK_DEINTERLEAVE_OPTIONS*.

RETURN_CODE **ATS_CUDA_CopyDeviceToHost**(void **GPUBuffer*, void **CPUBuffer*, U32 *bytesPerBuffer*, cudaStream_t *stream*)

Copies data between host and device.

Parameters

- `GPUBuffer`: Pointer to the GPU source memory address
- `CPUBuffer`: Pointer to the CPU destination memory address
- `bytesPerBuffer`: Size in bytes of the buffer to copy
- `stream`: Stream identifier on which the copy takes place

RETURN_CODE **`ATS_CUDA_CopyHostToDevice`**(void **GPUBuffer*, void **CPUBuffer*, U32 *bytesPerBuffer*, cudaStream_t *stream*)

Copies data between host and device.

Parameters

- `GPUBuffer`: Pointer to the GPU destination memory address
- `CPUBuffer`: Pointer to the CPU source memory address
- `bytesPerBuffer`: Size in bytes of the buffer to copy
- `stream`: Stream identifier on which the copy takes place

cudaStream_t **ATS_CUDA_CreateStream**()

Create a synchronous stream.

This function returns a pointer to the new stream identifier.

RETURN_CODE **ATS_CUDA_DestroyStream**(cudaStream_t *stream*)

Destroys and cleans up an asynchronous stream.

Parameters

- `stream`: Stream identifier.

RETURN_CODE **ATS_CUDA_FreeCPUBuffer**(void **CPUBuffer*)

Frees page-locked memory.

This function is used to free host memory allocated by ATS_CUDA_AllocCPUBuffer().

Parameters

- `CPUBuffer`: Pointer to the memory to free

RETURN_CODE **ATS_CUDA_FreeGPUBuffer**(void **GPUBuffer*)

Frees memory on the device.

This function is used to free GPU memory allocated by ATS_CUDA_AllocGPUBuffer().

Parameters

- `GPUBuffer`: Pointer to the device memory to free

RETURN_CODE **ATS_CUDA_GetVersion**(U8 **major*, U8 **minor*, U8 **revision*)

Get ATS-CUDA version number.

Parameters

- `major`: ATS-CUDA major version number.
- `minor`: ATS-CUDA minor version number.
- `revision`: ATS-CUDA revision number.

RETURN_CODE **ATS_CUDA_GetComputeCapability**(U32 *deviceIndex*, int **major*, int **minor*)

Function to get the compute capability of specified GPU.

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

- `deviceIndex`: 0-based index to the device.
- `major`: Major compute capability version number.
- `minor`: Minor compute capability version number.

RETURN_CODE **ATS_CUDA_QueryDeviceCount**(U32 **pDeviceCount*)

Function to get the number of available CUDA devices.

Return `ApiSuccess` if it succeeded.

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

Parameters

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

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

Function to get the name of a specific CUDA device.

Return `ApiSuccess` if it succeeded.

Return `ApiFailed` if the CUDA 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_CUDA_SetComputeDevice**(U32 *deviceIndex*)

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

- `deviceIndex`: 0-based index to the device.

RETURN_CODE **ATS_CUDA_StreamSynchronize**(cudaStream_t *stream*)

Waits for a stream to complete.

This function blocks the host thread until stream has completed all operations.

Parameters

- `stream`: Stream identifier.

bool **ATS_CUDA_StreamQuery**(cudaStream_t *stream*)

Queries a synchronous stream for completion status.

This function blocks the host thread until stream has completed all operations.

Return This function returns 1 if all operations in stream have completed.

Return This function returns 0 if not.

Parameters

- `stream`: Stream identifier.

INDEX

Symbols

A

U