ANDOR
an Oxford Instruments company

GPU Express

version 1.0 rev 20 May 2015

Large data solutions… simplified

Software Guide

Table of Contents

Contact Information: www.andor.com/contact_us/support_request

Document Revision History

Version	Released	Description
1.0	20 May 2015	Initial release for GPU Express v1.0

SECTION 1: INTRODUCTION

1.1 Introduction to GPU Express

The Andor GPU Express library has been created to simplify and optimise data transfers from camera to a CUDA enabled Nvidia Graphical Processing Unit (GPU) card to facilitate accelerated GPU processing during the acquisition pipeline. The initial release is designed to co-operate with Andor's Software Development Kit Version 3 (SDK3) library for utilisation with Andor's sCMOS camera range. The GPU Express library can also be utilised with Andor SDK2 based cameras, however, the example suite provided is set up for use with SDK3. (Linux is not currently supported).

Real-time processing during acquisition can provide the user with instantaneous feed-back and improved visualisation. It also provides a platform to save on memory resources via the use of standard compression algorithms and/or the removal of redundant information at acquisition, freeing up disk space to store only relevant data and reducing size of copies required to disk.

GPU Express provides a clean interface that integrates easily with Andor's SDK3 library to allow users to implement the above based on their specific requirements.

A CUDA accelerated version of Andor's utility library (CUDA_atutility.lib) is also provided for fast conversion between Pixel Encoding types and to provide unpacking of data due to granularity restrictions on the row width as is the case with Camera Link frame grabbers (in this case extra padding bytes are added to the end of each row to ensure that granularity limitations are met). See ''Andor Software Development Kit 3.pdf, Section 4.4" and Appendix C for further details.

1.2 Key Features

- Simple API to help reduce development time.
- Management of all required buffers in the GPU memory space.
- Management of intermediate CPU buffers for copy to GPU memory from camera.
- Provision of functions for safe allocation and de-allocation of user output buffers on the CPU side to store the result of GPU processing.
- Provision of functions for safe locking/pinning and unlocking/unpinning of user output buffers on the CPU side (as required for asynchronous memory copies) to store the result of GPU processing.
- Multiple camera support.
- Management of user-defined numbers and sizes of input and output GPU and CPU buffers to support varied user application requirements.
- Synchronisation call, to ensure that all previous copies and processing within a specified GPU Express Path are complete before continuing.
- Multiple copy functions for copies to/from GPU and CPU as required (dependent upon user's application).
- Management of CUDA streams to provide overlapping of copies to/from the GPU memory with accelerated GPU processing.
- Facilitates multiple GPU Processing.

1.3 Installation

Installation of Andor GPU Express library and examples (Windows 7 or later):

Notes:

- You must have sufficient privileges on your PC to perform the installation.
- GPU Express is not currently compatible with Linux.

1. Run the setup.exe file on the cd or from download.
2. Select the installation directory or accept the default when prompted by the installer.
3. Click on the Install button to confirm and continue with the installation.
4. Click on the Finish button when prompted.

1.4 Getting Started

This section demonstrates how to create a basic Andor GPU Express application that will test the library, ensuring that it can be successfully utilised with the Andor SDK3 library.

Running the examples that came with the installation

In the installation directory there is an examples directory “SDK3AcqExamples”. In that folder there are two further executable directories, ‘Win32’ and ‘x64’. These directories contain the executables required to run the provided examples in both the 32-bit Platform (in the ‘Win32’ folder) and the 64-bit Platform (in the ‘x64’ folder). The executables are provided within the “Release” folders contained in both ‘Win32’ and ‘x64’.

In the installation directory there are 2 folders containing all of the required library files, “lib64” for the 64-bit Platform and “lib32” for the 32-bit Platform. To run the examples, 1st copy the DLL files from the library folder (for the corresponding Platform) to the chosen executable’s ‘Release’ folder. The corresponding SDK3 DLL files shall also have to be copied to this directory. Then simply double click on the executables to run the examples in each case. Alternatively, open the Visual Studio Solution ‘SDK3AcqWithCCManagerExamples.sln’ to re-compile and run the examples from within the IDE (Note that in this case the provided executables shall be overwritten – a backup of each executable is therefore provided in the corresponding ‘Release’ folder under the ‘backup’ directory).

For further details on the examples see APPENDIX B.

Creating your own applications

With this installation you can create an application with a Microsoft compatible compiler. Perform the following steps to create your application.

1. Create an application with a CUDA supported C/C++ compiler (e.g. a recent version of Microsoft Visual Studio).

2. Add the SDK3 installation directory to the include path for the project. E.g. “C:\Program Files\Andor SDK3” for a 64-bit application or “C:\Program Files\Andor SDK3\win32” for a 32-bit application. Alternatively add the SDK2 installation directory to the include path if creating an SDK2 application, e.g. “C:\Program Files\Andor SOLIS\Drivers”.

3. Add the appropriate library from the SDK3 installation directory to your project.

- atcorem.lib for the Microsoft compiler

Alternatively add the appropriate library from the SDK2 installation to your project

- atmcd64m.lib for the Microsoft compiler (64-bit exe)
- atmcd32m.lib for the Microsoft compiler (32-bit exe)

4. Copy all the DLL's from the SDK3 or SDK2 installation directory to the directory that the executable is going to run from.

5. Add the GPU Express installation directory to the include path E.g. "C:\Program Files\Andor GPU Express\lib64" for a 64-bit application or "C:\Program Files\Andor GPU Express\lib32" for a 32-bit application.

6. Copy all the DLL's from the Andor GPU Express installation directory to the directory that the executable is going to run from.

7. Type or copy the code from one of the examples shown in the tutorial (SECTION 3:) into your projects main file (for an SDK3 example, otherwise create your own SDK2 example).

8. Update the code to include your own CUDA processing functions, add the required SDK3 code if acquiring data from an SDK3 camera, the required SDK2 code if acquiring from an SDK2 camera, compile and run the program.

1.5 Minimum Requirements

The minimum requirements for the GPU Express library are follows:

- C/C++ Compiler with full CUDA support.
- Andor SDK3 (version 3.8.30007.0) including relevant BitFlow Camera-Link Drivers, or Andor SDK2 (version 2.99.30001.0). ***Note that either the SDK3 or SDK2 binaries are required, even if not acquiring from an Andor camera.***
- CUDA Toolkit 6.0 Production Release. Note that the provided examples are built with CUDA 7.0, so CUDA 7.0 Production Release is required to run the examples. GPU Express is built with CUDA 6.0 to ensure support of 32-bit exes at this time.
- Nvidia CUDA enabled GPU card with Compute Capability of 2.0 :
    - Cards with Compute Capability below 2.0 may work with the library, but are not officially supported. Note that these cards are no longer supported by Nvidia.
    - (*Quadro card with Compute Capability of at least 3.0, and Dual Copy Engine are both STRONGLY RECOMMENDED to achieve maximum performance*).
    - *Compute Capability of at least 3.0 is required to run included 'DeblurNN' library examples.*
- Latest Nvidia Drivers for all installed Nvidia CUDA capable GPU cards.
- Windows 7 or 8: 32-bit or 64-bit Operating System. *Note GPU Express is not currently compatible with Linux.*

SECTION 2: API DESCRIPTION

2.1 Overview

The GPU Express API can be divided into several sections of functions, each controlling a particular aspect of the acquisition pipeline. There are sections in the API for:

- buffer management,
- synchronisation,
- CUDA stream management
- and for explicit copies.

First, we will look at a number of concepts introduced by CUDA and the GPU Express library.

2.1.1 Cuda Streams

CUDA streams are a concept introduced by Nvidia with the CUDA architecture and library. Essentially, a stream in CUDA holds a sequence of operations that execute on the device in the order in which they are issued by the host code. While operations within a stream are guaranteed to execute in the prescribed order, operations within different streams can be interleaved and, when possible, they can even run concurrently.

Utilising the same CUDA stream for a copy to GPU memory, processing on the GPU (within a CUDA 'kernel'), and copy back to CPU memory allows the 3 steps to be run asynchronously with respect to the calling host thread. The degree to which each CUDA stream's operations can be interleaved, and even overlapped, with regards to other CUDA stream operations are dependent on the GPU card specifications. For example, some (typically older) GPU cards have a single copy engine, so that we can only copy data to or from the GPU at a single point in time. Dual-copy engine GPU cards, however, allow copies to the GPU to be overlapped with copies from the GPU in separate copy engines (all properties of a particular GPU card can be queried via the CUDA library function 'cudaGetDeviceProperties'). How the user sets up their code, time taken for copies relative to the time taken for processing, and whether there are any implicit or explicit synchronisations within the CUDA processing code can all also effect the degree of overlap achievable. Detailed information is provided in the CUDA Programming guide.

See (http://docs.nvidia.com/cuda/cuda-c-programming-guide/#streams) and (http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/) for further details.

To achieve overlap, typically copies to the GPU, processing on the GPU, and copies from the GPU are called within user 'for' loops, such as in the example pseudo code below:

```
//--- Declare and create 'N number of CUDA streams

for (streamIndex = 0; streamIndex < N; streamIndex ++){
  //--- Copy to GPU in stream indexed by 'streamIndex'
  //--- process dataset on GPU in stream indexed by 'streamIndex'
  //--- Copy to CPU in stream indexed by 'streamIndex'
}

//--- Destroy 'N number of CUDA streams
```

Figure 1: Example pseudo code for CUDA processing with streams using single 'for' loop.

With a single copy engine GPU, the pseudo code shown in Figure 1 would result in a timeline similar to that shown in Figure 2:

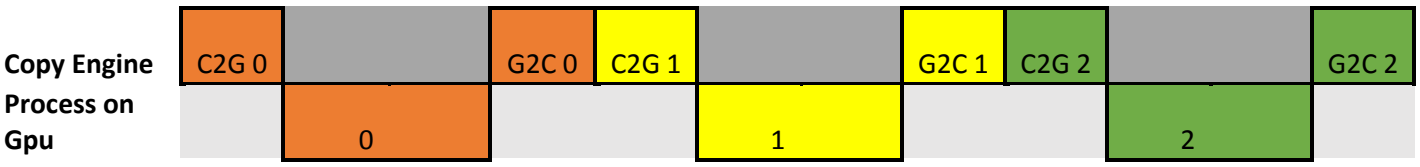


Figure 2: Typical operation timeline for a single copy engine GPU, using a single 'for' loop to call all operations. Each colour/number relates to a unique CUDA stream. 'C2G' signifies a copy from CPU to GPU and 'C2G' signifies a copy from CPU to GPU. This example uses a batch of 3 CUDA streams.

We can see here that although we are processing in different CUDA streams we do not get any overlap between the operations, as the copy operations are provided to the single copy engine in an order that prevents any overlap of GPU processing and copying. This is due to the fact that any copies input to the engine must occur in the order that they are queued up. Therefore, for example, the copy to the GPU card for stream 1 must wait until the copy from the GPU card for stream 0 is complete, which itself is dependent on the processing in stream 0 being completed.

A better approach with a single copy engine is to batch similar operations together for a set of CUDA streams, as shown in Figure 2.

```
//--- Declare and create 'N' number of CUDA streams

for (streamIndex = 0; streamIndex < N; streamIndex ++){
  //--- Copy to GPU in stream indexed by 'streamIndex'
}
for (streamIndex = 0; streamIndex < N; streamIndex ++){
  //--- process dataset on GPU in stream indexed by 'streamIndex'
}
for (streamIndex = 0; streamIndex < N; streamIndex ++){
  //--- Copy to CPU in stream indexed by 'streamIndex'
}

//--- Destroy 'N number of CUDA streams
```

Figure 3: Example pseudo code for CUDA processing with streams using multiple 'for' loops.

This pseudo code would kick off the copies to the GPU for each CUDA stream in turn in the calling host thread, then the processing for each, and finally the copies back to the CPU for each CUDA stream. All of the operations are asynchronous with respect to the calling host thread. What we should see is something similar to the timeline shown in Figure 4:

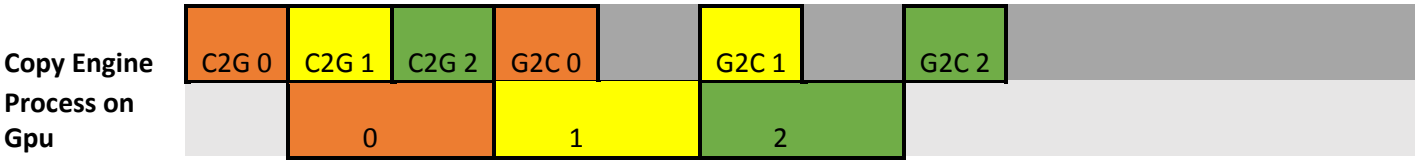


Figure 4: Typical operation timeline for a single copy engine GPU, using multiple 'for' loops to call the operations. Each colour/number relates to a unique CUDA stream. 'C2G' signifies a copy from CPU to GPU and 'C2G' signifies a copy from CPU to GPU. This example uses a batch of 3 CUDA streams.

Here, as soon as the copy to the GPU is complete for the 1st stream, we may then start the copy to the GPU for the 2nd stream, whereas in the previous example we had to wait for processing to finish before we could copy the result from the 1st stream back to the CPU.

With a dual copy engine GPU, the pseudo code shown in Figure 1 should result in a timeline similar to that shown in Figure 5:

Figure 5: Typical operation timeline for a dual copy engine GPU, using a single 'for' loop to call all operations. Each colour/number relates to a unique CUDA stream. This example uses a batch of 3 CUDA streams.

We can see in this case that we get overlapping between copies and processing on the GPU (within different streams), as here we have separate copy engines for copying to and from the GPU. All of the copies to the GPU are sent to one queue while the copies from the GPU are sent to another, so as soon as a copy in one direction is complete we can start the next copy in a different stream (assuming that processing is complete if copying back from the GPU).

With a dual copy engine GPU, it is possible that the pseudo code shown in Figure 3 may also result in a timeline similar to that shown in Figure 5 above. CUDA GPU cards with a Compute Capability of 2.0 or higher, however, also have the possibility to provide 'concurrent kernel execution' (a CUDA kernel is a function within the CUDA programming language that is ran in parallel on multiple threads when called), allowing CUDA kernel processing on the GPU within different CUDA streams to be overlapped. This, however, is dependent upon the availability of resources to be shared among the streams for processing. To enable concurrent kernels, kernels in different streams must be called sequentially (as shown in the pseudo code Figure 3). Normally, a signal is queued to launch the next operation in the same stream after an operation is complete. When a group of CUDA kernels are called sequentially, however, and concurrent kernel execution is available, this signal is delayed until after the last sequential CUDA kernel call, resulting in all copies back to the CPU to be delayed until all processing is complete. In the worst case scenario, where we only have sufficient resources to process one stream at a time, we may see a timeline similar to that shown in Figure 6:

Figure 6: Operation timeline for a dual copy engine GPU, using multiple 'for' loops to call the operations. Each colour/number relates to a unique CUDA stream. This example uses a batch of 3 CUDA streams. In this case there are not enough resources available to provide concurrent kernel execution.

If significant concurrency in processing is achieved, however, this can result in a shorter overall time for processing, resulting in a timeline similar to that shown in Figure 7:

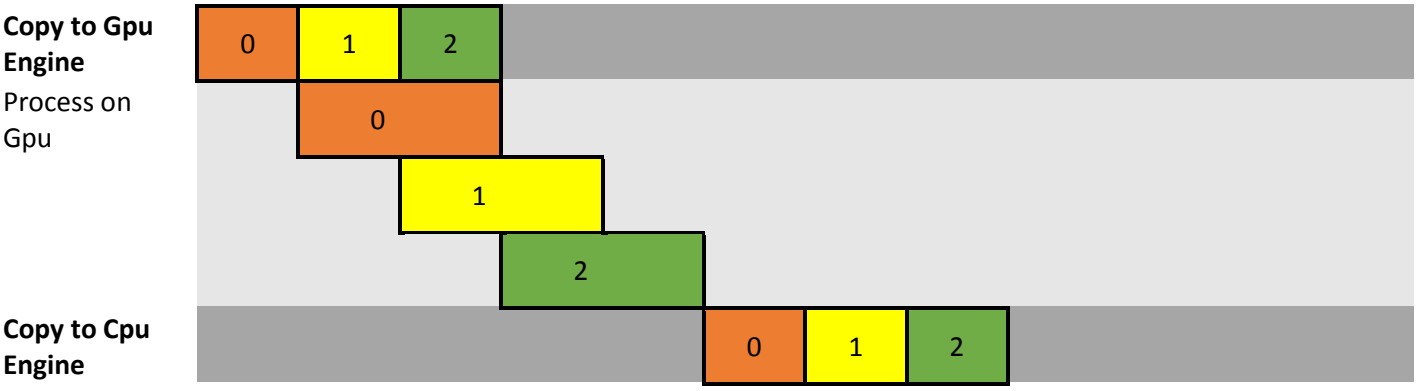


Figure 7: Operation timeline for a dual copy engine GPU, using multiple 'for' loops to call the operations. Each colour/number relates to a unique CUDA stream. This example uses a batch of 3 CUDA streams. In this case there are enough resources available to provide concurrent kernel execution.

To facilitate stream processing, all required CUDA streams are managed by the GPU Express library. The user therefore does not have to explicitly create or destroy any streams. An API function is also provided to return a reference to a specific CUDA stream (2.8.5 AT_GPU_GetStreamPtr).This stream can then be passed into the user's CUDA 'kernel' via the CUDA execution configuration syntax. The execution configuration syntax is the syntax required to call any CUDA kernel. It allows a CUDA user to specify how many parallel calls are required. It also allows the user to specify any shared memory that is required between CUDA threads, and is used to tell the CUDA compiler which CUDA stream the threads should be ran within (if not using the default stream). See (http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#execution-configuration) for more details.

Note that if a CUDA stream is not specified for a copy or a specific CUDA 'kernel' call for processing, then the copy or processing takes place within a CUDA managed 'default' stream.

2.1.2 GPU Express Path Concept

The 'GPU Express Path' concept provides a means to encapsulate a set of resources managed by the library to aid in the acquisition of a dataset, copying it to a GPU memory, processing on the GPU, and copying back to the CPU accessible memory.

Each GPU Express Path references a unique 'Input CPU Buffer' array, 'Input GPU Buffer' array, and CUDA stream managed by the library. The number of buffers in the 2 input arrays must be the same, so that if we have e.g. 2 'Input CPU Buffers' then we also have 2 'Input GPU Buffers'. Each Path may also reference an optional array of 'Output GPU Buffers', and an optional array of 'Output CPU Buffers'. Data from either an Input GPU Buffer or an Output GPU Buffer may be copied into an Output CPU Buffer via the GPU Express library API functions. It is possible for the user to specify that they require only Output GPU Buffers (for example, when utilising their own buffers for output to CPU memory), or to specify that they require only Output CPU Buffers (for example, if processing on the GPU in-place and copying the result to a single Output CPU Buffer). The size of the buffers within an array may vary, as required by the user's application, e.g. an 'Input CPU Buffer' array may consist of one buffer of size 2MB, and one buffer of size 1MB. Note, however, that the size of corresponding buffers in the 'Input CPU Buffer' and 'Input GPU Buffer' arrays must be the same. So with the above example, the 'Input GPU Buffer' array must also consist of one buffer of size 2MB, and one buffer of size 1MB.

Note that all 'Input GPU Buffers' and 'Output GPU Buffers' are memory buffers on the GPU card's RAM, and do not take up any space on the PC's CPU memory. Copies between buffers within the library are only allowed between buffers within the same GPU Express Path.

Each copy within a GPU Express Path takes place within the same CUDA stream. The CUDA stream may also be referenced by the user via an API function with the GPU Express Path's unique index. This provides the user with the means to process the buffers managed by a GPU Express Path within the unique CUDA stream that is also managed by that Path. *This is strongly recommended*. As shown in 2.1.1 Cuda Streams, if the copy and process operations of each group of buffers within a Path take place within their own unique CUDA stream, then it is possible to overlap the operations between different Paths. It also aids with synchronisation, as all operations within a CUDA stream are synchronous with respect to each other and guaranteed to execute in the prescribed order.

All copies that take place within a GPU Express Path are asynchronous, however, with respect to the calling Host thread. While operations within a CUDA stream are guaranteed to execute in the prescribed order, operations in different streams can be interleaved and, when possible, they can even run concurrently. As each GPU Express Path has its own unique CUDA stream, this provides us with the possibility to concurrently acquire and process multiple datasets within separate GPU Express Paths.

Figure 7 shows a number of possible GPU Express Path configurations with regards to the number of buffers and buffer size possible within a single GPU Express Path. Note that these are only a subset of all possible configurations:

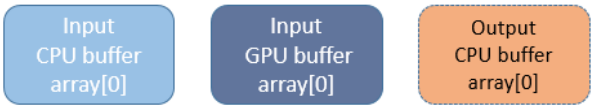


Figure 7a: Example layout of a 'GPU Express Path', containing 1 Input CPU Buffer, 1 Input GPU Buffer and 1 Output CPU Buffer.

Figure 7b: Example layout of a 'GPU Express Path', containing 1 Input CPU Buffer, 1 Input GPU Buffer, 1 Output GPU Buffer and 1 Output CPU Buffer. The output buffers in this case are of a different size to the input buffers.

Figure 7c: Example layout of a 'GPU Express Path', containing 1 Input CPU Buffer, 1 Input GPU Buffer, 2 Output GPU Buffers and 2 Output CPU Buffers. The output buffers in this case are of a different size to the input buffers, and the 2nd buffer in each output array is of a different size to the 1st buffer in the output arrays.

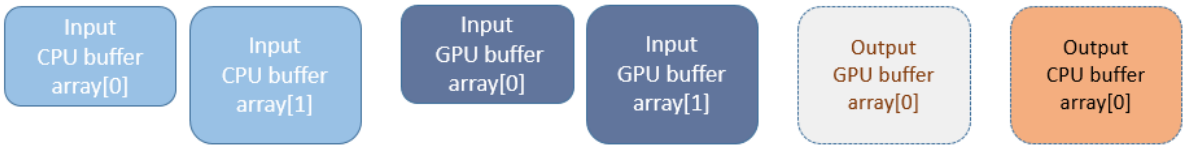


Figure 7d: Example layout of a 'GPU Express Path', containing 2 Input CPU Buffers, 2 Input GPU Buffers, 1 Output GPU Buffer and 1 Output CPU Buffer. The 2nd buffer in each input array is of a different size to the 1st buffer in the input arrays in this case.

2.1.3 DGPU Express Set Concept

GPU Express provides the user with a handle to a 'Set' of GPU Express Paths. Multiple Paths are required to manage the acquisition of a queue of buffers from the Andor SDK3 API, and to optimise performance via, for example, the interleaving of different Paths. When utilising the library with the SDK3, a GPU Express Set should be configured to manage a number of GPU Express Paths equal to the number of SDK3 buffers you wish to queue. This allows each GPU Express Path to iteratively manage 1 SDK3 buffer (i.e. each time an SDK3 buffer is updated it is processed by a GPU Express Path before being re-queued to the SDK3, updated again and processed by the same GPU Express Path until all acquisitions are complete).

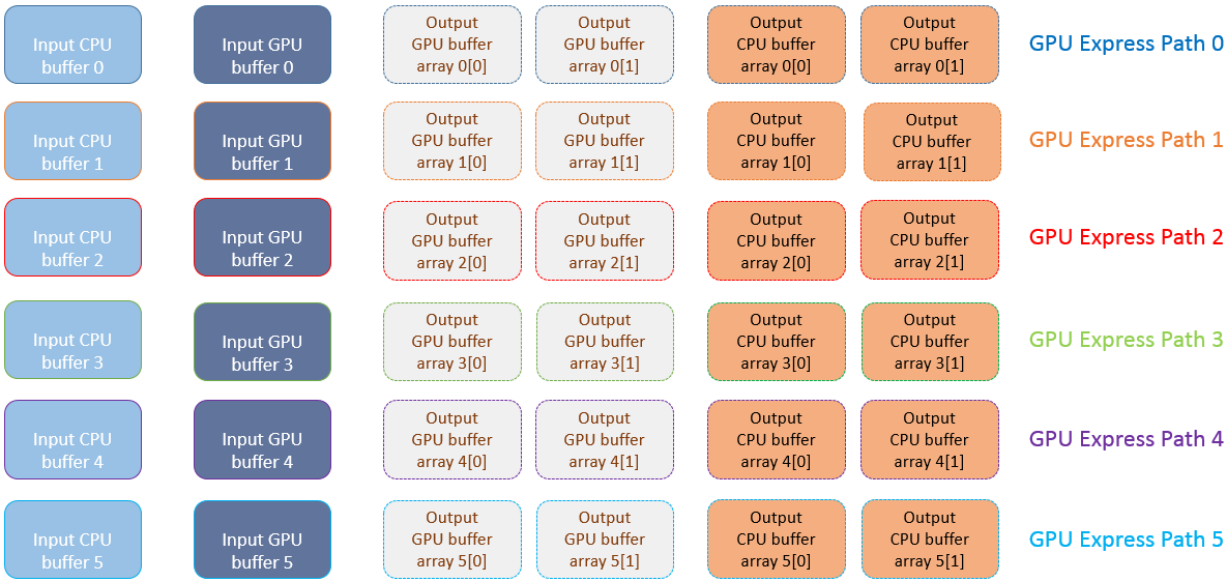


Figure 8: Example layout of 'GPU Express Set' containing 6 'GPU Express Paths'. Each GPU Express Path in this example contains 1 Input CPU Buffer, 1 Input GPU Buffer, 2 Output GPU Buffers and 2 Output CPU Buffers (all buffers are of equal size). All copies between any buffers must take place between buffers in the same GPU Express Path.

2.2 Function Listing

```
int AT_GPU_InitialiseLibrary();

int AT_GPU_FinaliseLibrary();



int AT_GPU_Open(AT_GPU_H * Hndl);

int AT_GPU_ConfigureSet(AT_GPU_H * Hndl, AT_GPU_U16 numPaths);

int AT_GPU_ConfigureGpuCardIdToPath(AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
              AT_GPU_U16 gpuCardId);

int AT_GPU_Close(AT_GPU_H Hndl);



int AT_GPU_ConfigureInputBuffers(AT_GPU_H Hndl,
              AT_GPU_U16 numInputBuffersPerPath,
              AT_GPU_U64 * inputBufferSizesInBytesArray);

int AT_GPU_ConfigureOutputGpuBuffers(AT_GPU_H Hndl,
              AT_GPU_U16 numOutputBuffersPerPath,
              AT_GPU_U64 * outputBufferSizesInBytesArray);

int AT_GPU_ConfigureOutputCpuBuffers(AT_GPU_H Hndl,
              AT_GPU_U16 numOutputBuffersPerPath,
              AT_GPU_U64 * outputBufferSizesInBytesArray);



int AT_GPU_GetInputCpuBufferArray (AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
              void*** bufferArrayPtr);

int AT_GPU_GetInputGpuBufferArray (AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
              void**** bufferArrayPtr);

int AT_GPU_GetOutputGpuBufferArray (AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
              void*** bufferArrayPtr);

int AT_GPU_GetOutputCpuBufferArray (AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
              void*** bufferArrayPtr);



int AT_GPU_CopyInputCpuToInputGpu(AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
              AT_GPU_U16 inputBufferArrayIndex);

int AT_GPU_CopyInputGpuToOutputCpu(AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
              AT_GPU_U16 bufferArrayIndex);


int AT_GPU_CopyInputGpuToUserCpu(AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
              AT_GPU_U16 inputBufferArrayIndex, void* userCpuBufPtr,
              AT_GPU_U64 userBufferSizeInBytes);

int AT_GPU_CopyOutputGpuToOutputCpu(AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
              AT_GPU_U16 outputBufferArrayIndex);
```

```
int AT_GPU_CopyOutputGpuToUserCpu(AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
              AT_GPU_U16 outputBufferArrayIndex, void* userCpuBufPtr,
              AT_GPU_U64 userBufferSizeInBytes);

int AT_GPU_CopyOutputCpuToUserCpu(AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
              AT_GPU_U16 outputBufferArrayIndex, void* userCpuBufPtr,
              AT_GPU_U64 userBufferSizeInBytes);


int AT_GPU_GetStreamPtr (AT_GPU_H Hndl, AT_GPU_U16 pathIndex, void** streamPtr);

int AT_GPU_CallUserFunction (AT_GPU_H _Hndl, AT_GPU_U16 pathIndex,
              AT_GPU_Callback userFunction, void * userData);

int AT_GPU_WaitPath (AT_GPU_H Hndl, AT_GPU_U16 pathIndex);

int AT_GPU_SetGpuCard(AT_GPU_H Hndl, AT_GPU_U16 gpuCardId);


//--- Auxiliary functions (no handle required)


int AT_GPU_GetErrorString(AT_GPU_U16 errorCode, AT_GPU_WC* stringData,
              AT_GPU_U16 stringLength);

int AT_GPU_CheckGpuCapability();

int AT_GPU_GetNumGpuCards(AT_GPU_U16 * numGpuCards);

int AT_GPU_GetBestGpuCard(AT_GPU_U16 * gpuCardID);

int AT_GPU_LockUserCpuBuf(void* dataPtr, AT_GPU_U64 bufferSizeInBytes);

int AT_GPU_UnlockUserCpuBuf(Hndl, void* dataPtr);
```

2.3 Library Initialisation

The library provides 2 functions for initialisation and finalisation.

2.3.1 AT_GPU_InitialiseLibrary

```
int AT_GPU_InitialiseLibrary()
```

must be called before any of the non-auxiliary functions in the API (see 2.2 Function Listing) for list of auxiliary functions). This allows the library to setup its internal data structures. The function takes no parameters.

2.3.2 AT_GPU_FinaliseLibrary

```
int AT_GPU_FinaliseLibrary()
```

must be called before your application closes or when you no longer wish to access the API. This function shall clean up all data structures held internally by the library. No parameters are taken by the function. Note that this function will reset all resources on each device immediately. It is the caller's responsibility to ensure that the device is not being accessed by any other host threads from the process when this function is called.

2.4 Initial Configuration

In order to utilise the resource handling features of the library a Handle of type 'AT_GPU_H' must first be obtained from the library. The Handle returned to the user is then subsequently associated with a unique set of GPU Express Paths (see 2.1.2 GPU Express Path Concept, which we refer to as a GPU Express Set).

2.4.1 AT_GPU_Open

```
int AT_GPU_Open(AT_GPU_H * Hndl)
```

provides a means to obtain a Handle to a GPU Express Set from the library. A reference to the Handle must be passed as the 1st parameter, which is updated and passed back to the user. The Handle returned to the user is then subsequently used to reference this unique Set of GPU Express Paths. The number of GPU Express Paths within a Set is configured by 2.4.2 AT_GPU_ConfigureSet.

Multiple Handles may be obtained from the library to provide management of multiple Sets of GPU Express Paths. This is useful, for example, if a number of differing acquisition setups requiring different numbers and/or sizes of buffers are required. *It is strongly recommended, however, that only 1 Handle is open at any moment in time*, to reduce the amount of Pinned/Locked memory required as page-locking too much memory may result in reduced system performance.

2.4.2 AT_GPU_ConfigureSet

```
int AT_GPU_ConfigureSet(AT_GPU_H Hndl, AT_GPU_U16 numPaths)
```

When called, this function configures the number of GPU Express Paths within the Set referenced by the Handle input as the 1st parameter. The number of Paths is input as the 2nd parameter.

Note that when utilising the GPU Express library with the Andor SDK3 library, it is recommended that the number of GPU Express Paths is specified to be equal to the number of buffers that shall be queued for subsequent acquisition of data from a camera by the SDK3 library. The 'Input CPU Buffer' of each path can then be utilised for this purpose. In a multiple camera setup, more than 1 'Input CPU Buffer' can be specified for each Path (using 2.5.1 AT_GPU_ConfigureInputBuffers). In this case, the 1st Input CPU Buffer of each Path can be queued for acquisition from the 1st camera, and the 2nd Input CPU Buffer of each Path can be queued for acquisition from the 2nd camera.

2.4.3 AT_GPU_ConfigureGpuCardIdToPath

```
int AT_GPU_ConfigureGpuCardIdToPath(AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
                                    AT_GPU_U16 gpuCardId)
```

When called, this function configures the input GPU Express Path indexed by the 2nd parameter to the GPU card indexed by the 3rd parameter, within the Set referenced by the Handle input as the 1st parameter. This tells the library that all allocations of buffers associated with this Path shall take place on the specified GPU card. Accordingly, all copies of buffers within this Path are to or from this GPU card, and all processing of the buffers shall take place on this card.

Once this function has been called, the number of Paths for the associated Handle is fixed, and can no longer be changed (i.e. 2.4.2 AT_GPU_ConfigureSet can no longer be called). Once ANY allocation of buffers has been completed (i.e. once any of the API functions detailed in 2.5 Buffer Handling have been called, the GPU card ID for each Path is fixed, and can no longer be changed.

A call to this function is only necessary if multiple GPU cards shall be used to process acquired data. By default, all GPU Express Paths are initially attached to the default GPU card (with gpuCardId 0).

2.4.4 AT_GPU_SetGpuCard

```
int AT_GPU_SetGpuCardId(AT_GPU_H Hndl, AT_GPU_U16 gpuCardId)
```

This function sets the calling Host (CPU) Thread to point to the GPU card indexed by the 2nd input parameter, within the Set referenced by the Handle input as the 1st parameter. Any user CUDA library calls (non-GPU Express calls) shall then be associated with this card. This is useful, for example, if allocating auxiliary memory within a user defined library on multiple GPUs. Note that GPU Express manages the GPU card associated with each Path internally, so *this function is only required for subsequent non-GPU Express CUDA library calls*.

2.4.5 AT_GPU_Close

```
int AT_GPU_Close(AT_GPU_H Hndl)
```

must be called once a handle is no longer required to clean up all library managed resources associated with the Set of GPU Express Paths referenced by the Handle passed in as the 1st parameter.

2.5 Buffer Handling

The library manages a user-defined number of Input CPU Buffers, Input GPU Buffers, Output GPU Buffers and Output CPU Buffers for each GPU Express Path.

The number and size of the buffers required may be configured dependent upon the end-users application requirements.

2.5.1 AT_GPU_ConfigureInputBuffers

```
int AT_GPU_ConfigureInputBuffers(AT_GPU_H Hndl, AT_GPU_U16 numInputBuffersPerPath,
              AT_GPU_U64 * inputBufferSizeInBytesArray)
```

is required to allocate the Input Buffers for each GPU Express Path within the Set referenced by the handle passed in as the 1st parameter. The 2nd parameter specifies the number of Input CPU *and* Input GPU Buffers to be allocated for each GPU Express Path within this Set.

This function allocates the Input CPU buffers (in CPU memory) and the Input GPU buffers (in GPU memory) for each GPU Express Path within the Set referenced by the Handle. The Input CPU buffers are page-locked to ensure faster copies to the corresponding Input GPU buffers, and to allow for asynchronous copies with respect to the CPU and operations taking place within other CUDA streams.

The 3rd parameter informs the library of the buffer sizes required for each of the CPU and GPU Input Buffers required in bytes. This array should contain a number of values equal to the number of buffers per path as specified by the 2nd parameter, and each value in this array is used to set the size of the corresponding buffer in the Input CPU and Input GPU buffer arrays, e.g. the 1st value in the input array is used to set the size of the 1st Input CPU and GPU buffers, and the 2nd value is used to set the size of the 2nd Input buffer in each Input CPU and GPU buffer array (if more than 1 input buffer per path has been specified with the 2nd parameter here).

Note that the buffer sizes should be based on the maximum possible size required for each buffer. For example, if acquiring data with the SDK3 from a Camera Link frame grabber, it is possible that unpacking of the data is required. The output from the unpacking step may require a larger number of bytes than the input, so to enable an in-place unpacking the number of bytes here should be set to the size of the output after unpacking. For further information see ‘APPENDIX C’.

Note that these Input buffers are de-allocated upon the occasion of a call to 2.4.5 AT_GPU_Close with the same input Handle. The function may only be called once for each Handle.

2.5.2 AT_GPU_ConfigureOutputGpuBuffers

```
int AT_GPU_ConfigureOutputGpuBuffers(AT_GPU_H Hndl,
             AT_GPU_U16 numOutputBuffersPerPath,
             AT_GPU_U64 * outputBufferSizeInBytes)
```

is required to allocate the Output GPU Buffers for each GPU Express Path within the Set referenced by the handle passed in as the 1st parameter. The 2nd parameter specifies the number of Output GPU Buffers to be allocated for each GPU Express Path within this Set.

The 3rd parameter informs the library of the buffer sizes required for each of the GPU Output Buffers required in bytes. This array should contain a number of values equal to the number of buffers per path as specified by the 2nd parameter, and each value in this array is used to set the size of the corresponding buffer in the Output GPU buffer array, e.g. the 1st value in the input array is used to set the size of the 1st Output GPU buffer, and the 2nd value is used to set the size of the 2nd Output GPU buffer (if more than 1 GPU Output buffer per path has been specified with the 2nd parameter here).

The function may only be called once for each Handle.

2.5.3 AT_GPU_ConfigureOutputCpuBuffers

```
int AT_GPU_ConfigureOutputCpuBuffers(AT_GPU_H Hndl,
             AT_GPU_U16 numOutputBuffersPerPath,
             AT_GPU_U64 outputBufferSizeInBytes)
```

is required to allocate the Output CPU Buffers for each GPU Express Path within the Set referenced by the handle passed in as the 1st parameter. The 2nd parameter specifies the number of Output CPU Buffers to be allocated for each GPU Express Path within this Set.

The 3rd parameter informs the library of the buffer sizes required for each of the CPU Output Buffers required in bytes. This array should contain a number of values equal to the number of buffers per path as specified by the 2nd parameter, and each value in this array is used to set the size of the corresponding buffer in the Output CPU buffer array, e.g. the 1st value in the input array is used to set the size of the 1st Output CPU buffer, and the 2nd value is used to set the size of the 2nd Output CPU buffer (if more than 1 CPU Output buffer per path has been specified with the 2nd parameter here).

The function may only be called once for each Handle.

2.6 User CPU Buffer Handling

To facilitate fast asynchronous copies from the library-managed Input GPU or Output GPU Buffers to user managed CPU buffers for subsequent processing or saving to disk, the library provides a number of auxiliary functions to page-lock/unlock user managed CPU Buffers.

Unlike the previous buffers, user CPU Buffers are explicitly managed by the caller of the library. The user must declare and allocate the buffers, and free the memory when they are no longer required.

The user CPU Buffers may be page-locked (pinned) with a specific CUDA library function before copying data to them from the GPU to optimise performance of the copy. This ensures that the buffers are stored in physical memory (RAM) preventing the need for additional copies from secondary storage (hard disk). It also ensures that they are mapped to the CUDA page table so the GPU can access it directly to improve performance of copies. Page-locking too much memory, however, may result in reduced system performance so it is important to only page-lock buffers that are required for a copy, and to page-unlock buffers when no longer required for this.

When locking and unlocking memory, note that locking for a single copy is inefficient, as there is overhead involved in the locking and unlocking, and these calls are not asynchronous with respect to the CPU host thread. User managed CPU buffers should therefore only be locked if being re-used for multiple copies from library managed GPU buffers.

Note that for buffers to be properly locked they must first be page-aligned by the user, i.e. on an Intel architecture the memory address should be a multiple of 4096. The simplest way to do this on Windows is to use the Windows 'VirtualAlloc' function. See (http://msdn.microsoft.com/en-us/library/windows/desktop/aa366887(v=vs.85).aspx) and (http://msdn.microsoft.com/en-gb/library/windows/desktop/aa366892(v=vs.85).aspx) for more details on this.

As the user CPU Buffers are not managed by the library, these functions are provided as auxiliary functions, and as such do not require Initialisation of the Library, and do not require a Handle to a GPU Express Path.

Note that the locking mechanisms used for these functions are portable to all CUDA enabled GPU cards on a system, so that improved performance of copies should be seen when copying to/from all available GPU cards.

2.6.1 AT_GPU_LockUserCpuBuf

```
int AT_GPU_LockUserCpuBuf(void* dataPtr, AT_GPU_U64 bufferSizeInBytes)
```

provides the user with a means to page-lock a user managed (page-aligned) CPU Buffer for optimised CUDA copies. The pointer to the buffer is required as the 1st parameter, and the size of the buffer in bytes is required as the 2nd parameter. No input parameter for a specific handle is required.

2.6.2 AT_GPU_UnlockUserCpuBuf

```
int AT_GPU_UnlockCPU(void* dataPtr, AT_GPU_U64 bufferSizeInBytes)
```

provides the user with a means to unlock a page-locked user CPU Buffer. The pointer to the buffer is required as the 1st parameter, and the size of the buffer in bytes is required as the 2nd parameter. Again no input parameter for a specific handle is required.

2.7 Copy Functions

The library provides a number of functions for fast asynchronous copies to/from buffers within a GPU Express Path. Copy functions for copying from the Input GPU Buffers and Output GPU Buffers within a GPU Express Path to a user managed CPU buffer are also provided.

All copies are asynchronous with respect to the CPU, and all copies are carried out within a CUDA stream associated with the input GPU Express Path index in each case.

2.7.1 AT_GPU_CopyInputCpuToInputGpu

```
int AT_GPU_CopyInputCpuToInputGpu(AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
                    AT_GPU_U16 inputBufferArrayIndex)
```

copies data from an Input CPU Buffer to an Input GPU Buffer within the GPU Express Path indexed by the 2nd parameter. The Set of GPU Express Paths to be indexed are referenced by the Handle input as the 1st parameter. The index of the input buffers within their respective arrays is input as the 3rd parameter.

2.7.2 AT_GPU_CopyInputGpuToOutputCpu

```
int AT_GPU_CopyInputGpuToOutputCpu (AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
                            AT_GPU_U16 outputBufferArrayIndex)
```

copies data from an Input GPU Buffer to an Output CPU Buffer within the GPU Express Path indexed by the 2nd parameter. The Set of GPU Express Paths to be indexed are referenced by the Handle input as the 1st parameter. The index of the Output CPU Buffer is input as the 3rd parameter.

2.7.3 AT_GPU_CopyInputGpuToUserCpu

```
int AT_GPU_CopyInputGpuToUserCpu (AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
                            AT_GPU_U16 inputBufferArrayIndex,
                            void* userBufPtr, AT_GPU_U64 userBufferSizeInBytes)
```

copies data from an Input GPU Buffer to a user managed CPU Buffer. The GPU Express Path of the Input GPU Buffer is indexed by the 2ndparameter. The Set of GPU Express Paths to be indexed are referenced by the Handle input as the 1st parameter. The index of the Input GPU Buffer is input as the 3rd parameter. A pointer to the user CPU buffer is required as the 4th parameter. The size of the user buffer must be entered as the 5th parameter. If this is smaller than the size of the Input GPU buffer, then an error shall be returned.

2.7.4 AT_GPU_CopyOutputGpuToOutputCpu

```
int AT_GPU_CopyOutputGpuToOutputCpu (AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
                            AT_GPU_U16 outputBufferArrayIndex)
```

copies data from an Output GPU Buffer to an Output CPU Buffer within the GPU Express Path indexed by the 2nd parameter. The Set of GPU Express Paths to be indexed are referenced by the Handle input as the 1st parameter. The index of the Output CPU Buffer and output GPU Buffer are input as the 3rd parameter.

2.7.5 AT_GPU_CopyOutputGpuToUserCpu

```
int AT_GPU_CopyOutputGpuToUserCpu (AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
                                   AT_GPU_U16 outputBufferArrayIndex, void* userCpuBufPtr,
                                   AT_GPU_U64 userBufferSizeInBytes)
```

copies data from an Output GPU Buffer to a user managed CPU Buffer. The GPU Express Path of the Output GPU Buffer is indexed by the 2nd parameter. The Set of GPU Express Paths to be indexed are referenced by the Handle input as the 1st parameter. The index of the Output GPU Buffer is input as the 3rd parameter. A pointer to the user CPU buffer is required as the 4th parameter. The size of the user buffer must be entered as the 5th parameter. If this is smaller than the size of the Input GPU buffer, then an error shall be returned.

2.7.6 AT_GPU_CopyOutputCpuToUserCpu

```
int AT_GPU_CopyOutputCpuToUserCpu (AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
                                   AT_GPU_U16 outputBufferArrayIndex, void* userCpuBufPtr,
                                   AT_GPU_U64 userBufferSizeInBytes)
```

copies data from an Output CPU Buffer to a user managed CPU Buffer. The GPU Express Path of the Output CPU Buffer is indexed by the 2nd parameter. The Set of GPU Express Paths to be indexed are referenced by the Handle input as the 1st parameter. The index of the Output CPU Buffer is input as the 3rd parameter. A pointer to the user CPU buffer is required as the 4th parameter. The size of the user buffer must be entered as the 5th parameter. If this is smaller than the size of the Input GPU buffer, then an error shall be returned.

2.8 Get Library Managed Resource Functions

The library provides a number of functions to provide the user with library managed resources for subsequent use within their own functions.

2.8.1 AT_GPU_GetInputCpuBufferArray

```
int AT_GPU_GetInputCpuBufferArray (AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
                                   void *** bufferArrayPtr)
```

passes a pointer to a pointer to an Input CPU Buffer array back to the caller via the 3rd parameter. The buffer belongs to the GPU Express Path indexed by the 2nd parameter, and the GPU Express Set referenced by the Handle input as the 1st parameter.

2.8.2 AT_GPU_GetInputGpuBufferArray

```
int AT_GPU_GetInputGpuBufferArray (AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
                                   void *** bufferArrayPtr)
```

passes a pointer to a pointer to an Input GPU Buffer array back to the caller via the 3rd parameter. The buffer belongs to the GPU Express Path indexed by the 2nd parameter, and the GPU Express Set referenced by the Handle input as the 1st parameter.

2.8.3 AT_GPU_GetOutputGpuBufferArray

```
int AT_GPU_GetOutputGpuBufferArray (AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
                                    void *** bufferArrayPtr)
```

passes a pointer to a pointer to an Output GPU Buffer array back to the caller via the 3rd parameter. The buffer array belongs to the GPU Express Path indexed by the 2nd parameter, and the GPU Express Set referenced by the Handle input as the 1st parameter.

2.8.4 AT_GPU_GetOutputCpuBufferArray

```
int AT_GPU_GetOutputCpuBufferArray (AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
              void *** bufferArrayPtr)
```

passes a pointer to a pointer to an Output CPU Buffer array back to the caller via the 3rd parameter. The buffer array belongs to the GPU Express Path indexed by the 2nd parameter, and the GPU Express Set referenced by the Handle input as the 1st parameter.

2.8.5 AT_GPU_GetStreamPtr

```
int AT_GPU_GetStreamPtr (AT_GPU_H Hndl, AT_GPU_U16 pathIndex,
              void ** streamPtr)
```

passes a pointer to a pointer to a CUDA stream back to the caller via the 3rd parameter. The CUDA stream belongs to the GPU Express Path indexed by the 2nd parameter, and the GPU Express Set referenced by the Handle input as the 1st parameter.

2.9 Synchronisation

The library provides an explicit synchronisation function, to ensure that all previous copies and processing within a specified GPU Express Path are complete before continuing.

2.9.1 AT_GPU_WaitPath

```
int AT_GPU_WaitPath (AT_GPU_H Hndl, AT_GPU_U16 pathIndex)
```

ensures that all previous copies and processing associated with the GPU Express Path indexed by the 2nd parameter are complete before continuing the calling CPU host thread. The GPU Express Path belongs to a Set referenced by the 1st parameter.

2.10 Calling a User Defined Function

The library provides an API function to allow the caller to pass their own function to the library to be called with specified library resources. With this approach, the caller does not have to declare any variables to hold references to the library managed resources, and they do not need to call any of the functions in 2.8 Get Library Managed Resource Functions. Instead they may create a callback function of type '*AT_GPU_Callback*', declared below (and in the library's header file):

```
typedef int (*AT_GPU_Callback)(void** InputGpuBufferArray,
                               void** outputGpuBufferArray,
                               void* streamPtr, void* userDefinedData)
```

Any user-created callback function must obey the above type. The 1st 3 parameters are library managed resources, namely the Input GPU Buffer array, Output GPU Buffer array, and CUDA stream pointer managed by the same GPU Express Path that shall be specified when the function is ultimately called. The 4th parameter allows the user to pass their own data through to the function via a void pointer.

As an example, a user may have a function with the following declaration that calls into a CUDA kernel:

```
int userExampleFunc(unsigned short * gpu_inputBuffer,
             unsigned short * gpu_outputBuffer, int inputInt, cudaStream_t *stream);
```

This function takes a pointer to an input buffer allocated on GPU memory as the 1st parameter, a pointer to an output buffer allocated on GPU memory as the 2nd parameter, an integer as the 3rd parameter, and a pointer to a CUDA stream as the 4th parameter.

In order to call this function with an Input GPU Buffer, Output GPU Buffer, and CUDA stream that are all managed by the same GPU Express Path, a callback function must be created that matches the *'AT_GPU_Callback'* type. The user can then simply pass a reference to their callback to the API function 2.10.1 AT_GPU_CallUserFunctionthat is described below.

For example, look at the following user defined callback function that simply calls into our example function declared above:

```
int userExampleFunc_Callback(void** InputGpuBufArray, void** OutputGpuBufArray,
                             void * streamPtr, void * userData)
{
  ret = userExampleFunc( reinterpret_cast<unsigned short*>(InputGpuBufArray[0]),
                        reinterpret_cast<unsigned short*>(OutputGpuBufArray[0])
                        *(reinterpret_cast<int*>(userData)),
                        reinterpret_cast<cudaStream_t*>(streamPtr) )
};
```

When a reference to this is passed through to 2.10.1 AT_GPU_CallUserFunction with a specific Path index, e.g:

```
int inputInt = 10;
AT_GPU_CallUserFunction(Hndl0, pathIndex, &userExampleFunc_Callback, (void*)(&inputInt));
```

the library calls '`userExampleFunc_Callback`' with the Input GPU Buffer array (1st parameter), Output GPU Buffer array (2nd parameter), and CUDA stream pointer (3rd parameter) specified by '`pathIndex`'. The user's own data (`inputInt`) is also passed through as the 4th parameter.

2.10.1 AT_GPU_CallUserFunction

```
int AT_GPU_CallUserFunction(AT_GPU_H _Hndl,
                            AT_GPU_U16 pathIndex,
                            AT_GPU_Callback userCallbackFunc,
                            void * userData);
```

This function provides a simple method to access the library managed resources associated with the GPU Express Path indexed via the 2nd input parameter. The Set of GPU Express Paths to be indexed are referenced by the Handle input as the 1st parameter.

When called, the Input GPU Buffer array, Output GPU Buffer array and CUDA stream pointer associated with the input GPU Express Path index are passed through to the user's callback function that is input to the library via a pointer to the function in the 3rd parameter. The callback function may then pass these through to a CUDA kernel function for processing on the GPU. The user's own data may also be passed through via the 4th parameter.

2.11 Error Checking

The library returns an error code from every function call to the API (see 2.13 Error Codes) for a listing of the codes). Each error code has a unique error string associated with it.

2.11.1 AT_GPU_GetErrorString

```
int AT_GPU_GetErrorString(AT_GPU_U16 errorCode, AT_GPU_WC* String,
                          AT_GPU_U16 StringLength)
```

retrieves an error string associated with the input code in the 1st parameter passed to the function. The string is returned in the memory pointed to by the 2nd parameter. The character type used by the API is a 16 bit wide character defined by the '*AT_GPU_WC*' type, which is compatible with the 'wchar_t' type. An example of converting wide character strings to char strings can be found in 'APPENDIX D - Conversion between char* and AT_GPU_WC*'.

2.12 GPU card Utility Functions

A number of functions are included to check for CUDA compatibility, and to provide information on the currently installed CUDA compatible cards.

2.12.1 AT_GPU_CheckGpuCapability

```
int AT_GPU_CheckGpuCapability()
```

may be used to verify that your system is compatible with the CUDA requirements for the library (this checks both the CUDA driver and installed cards). The function returns an error message indicating the cause of the incompatibility if applicable; otherwise it returns the library's 'AT_GPU_SUCCESS' code.

2.12.2 AT_GPU_GetNumGpuCards

```
int AT_GPU_GetNumGpuCards()
```

returns the total number of CUDA compatible cards connected to the current system.

2.12.3 AT_GPU_GetBestGpuCard

```
int AT_GPU_GetBestGpuCard()
```

returns the index of the CUDA compatible card with the maximum possible GFLOPs. The index returned refers to the physical position of the card's PCI slot, and can be used to …

2.13 Error Codes

The following section outlines the error codes and description.

Success		
AT_GPU_SUCCESS	(0)	Function call has been successful

Library Initialisation		
AT_GPU_ERR_NOTINITIALISED	(1)	The library has not yet been initialised
AT_GPU_ERR_PREINITIALISED	(2)	The library is currently initialised, and cannot be re-initialised
AT_GPU_ERR_GETHNDL	(3)	The function was unable to provide a handle
AT_GPU_ERR_INVALIDHNDL	(4)	An invalid handle was passed to the function

Path Handling		
AT_GPU_ERR_PATHNONINIT	(5)	The number of GPU Express Paths required has not yet been set
AT_GPU_ERR_PATHREINIT	(6)	The number of GPU Express Paths cannot be reset once buffers have been allocated
AT_GPU_ERR_NUMPATHS	(7)	The number of paths passed to the function MUST be between 4 and 12 inclusive
AT_GPU_ERR_PATHINDEX	(8)	Invalid GPU Express Path index entered
AT_GPU_ERR_CARDTOPATH	(9)	GPU card for this GPU Express Path can no longer be changed (as buffers have previously been allocated).

Buffer Handling		
AT_GPU_ERR_INCOMPATBUFSIZE	(10)	Destination buffer size is smaller than number of bytes to be copied
AT_GPU_ERR_BUFNONINIT	(11)	Input and/or Output GPU Buffers have not yet been allocated
AT_GPU_ERR_BUFREINIT	(12)	Input and/or Output GPU Buffers may not be re-allocated
AT_GPU_ERR_NUMBUFFERS	(13)	Invalid number of buffers entered. Maximum for any type is 4
AT_GPU_ERR_BUFFERSIZE	(14)	Invalid buffer size entered.
AT_GPU_ERR_BUFFERINDEX	(15)	Invalid buffer index entered.

CUDA Compatibility		
AT_GPU_ERR_CUDACONTEXT	(16)	Could not set CUDA context on the requested GPU card
AT_GPU_ERR_CUDASTREAM	(17)	Could not create a CUDA stream
AT_GPU_ERR_GPUCARDID	(18)	Invalid GPU card ID has been entered
AT_GPU_ERR_NOCUDACARD	(19)	No compatible CUDA card was found
AT_GPU_ERR_CUDADRIVER	(20)	No compatible CUDA driver was found

CUDA Error Handling		
AT_GPU_ERR_CUDAINIT	(21)	The function was unable to initialise the CUDA driver API
AT_GPU_ERR_CPUMEMALLOC	(22)	The function was unable to allocate page aligned memory on the CPU
AT_GPU_ERR_CUDAMEMALLOC	(23)	The function was unable to allocate global memory on the CUDA card
AT_GPU_ERR_CUDAMEMLOCK	(24)	The function was unable to register a CPU buffer as pinned to CUDA API
AT_GPU_ERR_CUDAKERNEL	(25)	An error was returned when attempting to process a CUDA kernel
AT_GPU_ERR_CUDAMEMCOPY	(26)	The function was unable to complete a CUDA memory copy
AT_GPU_ERR_CUDASYNC	(27)	The function was unable to complete a CUDA synchronisation call (DEBUG only)
AT_GPU_ERR_CUDALIBCALL	(28)	The function was unable to complete a CUDA library call (DEBUG only)

OS Specific Errors		
AT_GPU_ERR_OPENPROCESS	(29)	The function failed to retrieve the process handle (Windows)
AT_GPU_ERR_GTPROCWRKSETSZ	(30)	The function failed to get the current process working set size (Windows)
AT_GPU_ERR_STPROCWRKSETSZ	(31)	The function failed to set the current process working set size (Windows)
AT_GPU_ERR_NONCOMPATOS	(32)	The library in not compatible with the current OS (Mac OSX)

License Error		
AT_GPU_ERR_LICENSE	(33)	No valid license was found for the library. Please contact Andor Tech.

SECTION 3: TUTORIAL

This section will explain how to develop a program to use the API to optimise data transfers to your CUDA enabled GPU card for processing while acquiring data with the Andor SDK3 library. It shall guide the user through the functionality that is required in all cases, and also detail how to use the API for a number of differing applications.

The first thing that must be done is to add the appropriate library files to the project.

During the SDK3 installation a Borland/Embarcadero library (*atcore.lib*) and a Microsoft library (*atcorem.lib*) are made available. You should also add the include path of the SDK3 installation directory to your project. The *'atcore.h'* file from this directory needs to be included in any C/C++ or CUDA file that requires access to the SDK3 type definitions and/or functions. Note that the list of supported Host Compilers for CUDA can be found here: 'http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#supported-host-compilers'. The Borland/Embarcadero compiler at the time of writing is not supported, although an application built with the compiler can be linked to a library that utilises CUDA processing.

Similarly, during the GPU Express installation, the Microsoft library '*atGpuExpress.lib*' is provided. You should also add the corresponding library include path from the GPU Express installation directory to your project (dependent upon your chosen Platform this shall be either *'lib32/Release'* or *'lib64/Release'*). The '*AT_IGpuExpress.h*' file found under the '*GPU Express'* folder in the GPU Express installation directory shall need to be included in any C/C++ or CUDA file that requires access to the GPU Express type definitions and/or functions. The '*AT_CUDA_atutility.h*' file (also found under the '*GPU Express'* folder) must be included in any C/C++ or CUDA file that requires access to the unpacking functionality of the supplied CUDA Utility library (See '*APPENDIX C – CUDA UTILITY LIBRARY*').

It is assumed at this point that the reader has read the 'Andor Software Development Kit 3.pdf' document and is familiar with the Andor SDK3 API.

Note that the code snippets below are not complete, and are provided to familiarise the reader with the library and its functionality. The reader would be required to add code for acquisition of data to run these examples. An example Microsoft Visual Studio test suite [APPENDIX B – INCLUDED EXAMPLE SUITE] is provided to the user that includes all of the necessary SDK3 code to build and run a set of examples for initial testing of the library.

SDK3 pseudo-code is presented in red in the following code snippets, code comments are shown in green.

3.1 Initialisation

The first call to the GPU Express API should be 2.3.1 AT_GPU_InitialiseLibrary and the last call should be 2.3.2 AT_GPU_FinaliseLibrary. These functions will prepare the API to manage the internal resources required, and free resources when no longer needed.

```
//--- Initialise SDK3 (if acquisition is required from an SDK3 camera)

AT_GPU_InitialiseLibrary( );

AT_GPU_FinaliseLibrary( );

//--- Finalise SDK3 (if acquisition is required from an SDK3 camera)
```

3.2 Error Checking

Every function will return an error code when called. It is recommended that a user check every return code before moving on to the next statement. If the 2.3.1 AT_GPU_InitialiseLibrary function call fails there is no point continuing on with the program as every proceeding function call will also fail (apart from those listed as 'Auxiliary functions' in 2.2 Function Listing.

Each return code that could possibly be returned is listed in the '*AT_IGpuExpress.h'* file and documented in '*2.13 Error Codes*'. A string describing the error related to each error code can also be retrieved via the use of the library function 2.11.1 AT_GPU_GetErrorString.

As GPU Express is designed to assist GPU processing, the API provides a function to verify that a compatible GPU card is attached and that a suitable driver is installed on the current system. Note that the library only supports Nvidia's CUDA (and by extension Nvidia's CUDA enabled cards and drivers) although support for openCL may be added in a future release. The function 2.12.1 AT_GPU_CheckGpuCapability returns an error if there are any issues with the current system configuration in this regard.

So now the program becomes:

```
int i_returnCode;

//--- Initialise SDK3
i_returnCode = AT_GPU_InitialiseLibrary( );
if (i_returnCode == AT_GPU_SUCCESS) {
  i_returnCode = AT_GPU_CheckGpuCapability( );
  if (i_returnCode == AT_GPU_SUCCESS) {
    //---continue with program
  }
}
AT_GPU_FinaliseLibrary( );
if (i_returnCode != AT_SUCCESS) {
  //--- error FinaliseLibrary
}

//--- Finalise SDK3
```

Error Checking

It is highly recommended that you check return codes from every function in case of error. For the purpose of this tutorial the error checking will be kept to a minimum to reduce the length of the program.

3.3 Obtaining a Handle

Selecting Exit will close all open Datasets and exit the DBV application.

The next stage of the program is to obtain a handle to a GPU Express Set of Paths from the GPU Express library. This is achieved with the 2.4.1 AT_GPU_Open function. Each Handle returned to the user acts as an index to a unique Set of GPU Express Paths (See 2.1.2 GPU Express Path Concept). Multiple Handles may be used, for example, to facilitate the use of multiple cameras if required.

Each handle should be cleared before the end of the user program, with the GPU Express API call 2.4.5 AT_GPU_Close.

Now the program is:

```
int i_returnCode;
AT_GPU_H gpuHndl;

//--- Initialise SDK3 and get Camera Handle

i_returnCode = AT_GPU_InitialiseLibrary( );
if (i_returnCode == AT_GPU_SUCCESS) {
  i_returnCode = AT_GPU_CheckGpuCapability( );
  if (i_returnCode == AT_GPU_SUCCESS) {
    i_returnCode = AT_GPU_Open(&gpuHndl);
    //---continue with program
    i_returnCode = AT_GPU_Close(gpuHndl);
    if (i_returnCode != AT_GPU_SUCCESS){

     //--- error clearing handle

    }
  }
}
AT_GPU_FinaliseLibrary( );
if (i_returnCode != AT_SUCCESS) {
  //--- error FinaliseLibrary
}

//--- Finalise SDK3
```

3.4 Initial Buffer and Memory Handling

Now we will add code to the program to allocate the required buffers necessary on both the CPU and GPU, to prepare for acquisition and subsequent processing.

After initialisation of the SDK3 library, the user must set up acquisition parameters with the SDK3 API. It is then possible to find a value for the number of bytes required for each buffer, based on the acquisition frame dimensions and Pixel Encoding chosen by the user. It is useful to bear in mind that the unpacking process (See APPENDIX C) can result in a larger frame size after unpacking, so this must be taken into account when calculating the number of bytes required for each buffer.

Once calculated, we can then use the GPU Express API function 2.4.2 AT_GPU_ConfigureSet to set the number of GPU Express Paths required for the acquisition. The number of GPU Express Paths should be set to be equal to the number of buffers to be queued to the SDK3.

The GPU Express API function 2.5.1 AT_GPU_ConfigureInputBuffers is then required to allocate the required number of (both CPU and GPU) Input buffers for each GPU Express Path, using the pre-calculated required acquisition size in bytes.

For the first tutorial, we do not request any Output Buffers. Once allocated, we can assign the Input CPU Buffers to the SDK3 as normal, using 2.8.1 AT_GPU_GetInputCpuBufferArray to return a pointer to each allocated Input CPU buffer in turn.

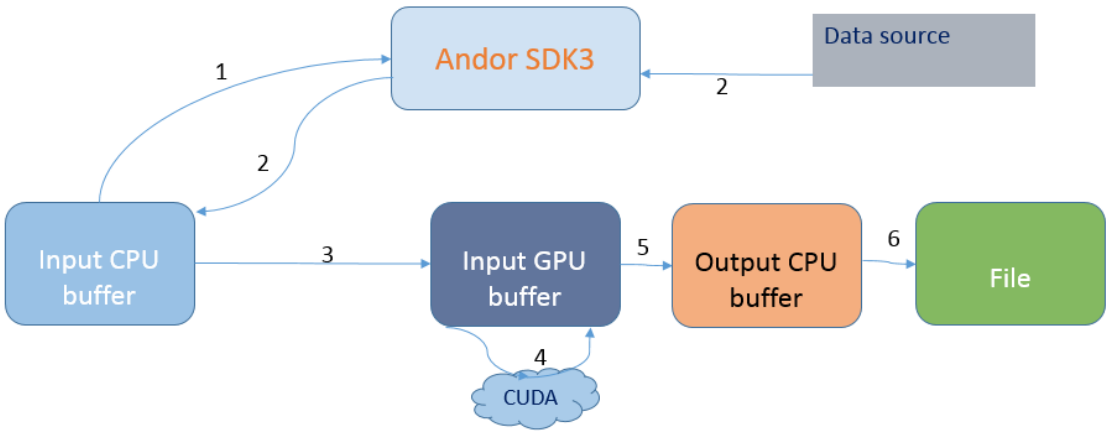
No error checking will be shown in the following example to help with clarity but it is recommended that in final code all return codes are checked.

```
int i_returnCode;
AT_GPU_H gpuHndl;
void ** cpuBufferArray;
AT_GPU_U64 * acqBufferSizeArray;

//--- Initialise SDK3 and get Camera Handle

AT_GPU_InitialiseLibrary( );
AT_GPU_CheckGpuCapability( );
AT_GPU_Open(&Hndl);

//--- Configure Input Parameters and get required buffer size(s)

AT_GPU_ConfigureSet(gpuHndl, numRequiredPaths);

AT_GPU_ConfigureInputBuffers(gpuHndl, 1, acqBufferSizeArray);

for (int i = 0; i < numRequiredPaths; i++) {
  AT_GPU_GetInputCpuBufferArray(gpuHndl, i, &cpuBufferArray);
  //--- assign reinterpret_cast<unsigned char *>(cpuBufferArray[0])
  //--- to the SDK3 acquisition queue
}

//--- continue with program

AT_GPU_Close(gpuHndl);
AT_GPU_FinaliseLibrary( );
//--- Finalise SDK3
```

3.5 Simple Acquisition Cycle

This section will show how to get data from an acquisition cycle of 100 frames, and process each acquired frame on the GPU. Once we have started the acquisition with the SDK3 API, we can calculate a value for the GPU Express Path index for subsequent calls to the GPU Express API for each acquisition. We simply iterate through the indices from 0 to 1 less than the total number of Paths set for the current Handle. Once a frame has been acquired from the SDK3 we copy from the Input CPU buffer to the Input GPU buffer using 2.7.1 AT_GPU_CopyInputCpuToInputGpu ***[3]***, and call the requested user defined function with the required library managed resources ***[4]***. With this example, we then copy back from the Input GPU buffer to an Output CPU buffer using 2.7.2 AT_GPU_CopyInputGpuToOutputCpu ***[5]***. Example code relating to this example can be found in the provided 'SimpleSDK3AcqExamples' Visual Studio solution, within the 'simpleAcquistionCycle' project. See B.1 Simple SDK3 Acquisition Examples for further details.

A flow chart for this acquisition cycle is shown below for a single GPU Express Path. The steps are repeated for each Path index in turn within the 'while' loop in the below code snippet until all iterations are complete. The steps listed are included in the code snippet for clarification:



1. ***Input CPU buffer passed to SDK3 (via SDK3 function) and queued for acquisition***
2. ***Frame acquired from Data Source to Input CPU buffer***
3. ***GPU Express copy to Input GPU Buffer [AT_GPU_CopyInputCpuToInputGpu] (asynchronous wrt calling Host Thread)***
4. ***CUDA Processing (asynchronous wrt calling Host Thread – if correct CUDA stream is passed to user function)***
5. ***GPU Express copy to Output CPU Buffer [AT_GPU_CopyInputGpuToOutputCpu] (asynchronous wrt calling Host Thread)***
6. ***Result copied to file***

Figure 9: Flow chart for a single GPU Express Path during a Simple Acquisition

```
void **cpuInitBufferArray;
AT_GPU_U64 * acqBufferSizeArray;

AT_GPU_H gpuHndl;
```

```
extern void addConstantOnGpuFunc(unsigned short * inputBuffer, unsigned short inputConst, unsigned
short bufferWidth, unsigned short bufferHeight, cudaStream_t *stream);

int main(int argc, char* argv[])
{
  //--- Initialise SDK3 and get Camera Handle
  //--- Configure acq. parameters and get required buffer size(s), width and height

  AT_GPU_InitialiseLibrary();
  AT_GPU_CheckGpuCapability();
  AT_GPU_Open(&gpuHndl);

  AT_GPU_ConfigureSet(gpuHndl, numRequiredPaths);
  AT_GPU_ConfigureInputBuffers(gpuHndl, 1, acqBufferSizeArray);

  for (unsigned int i = 0; i < numRequiredPaths; i++) {
    AT_GPU_GetInputCpuBufferArray(gpuHndl, i, &cpuInitBufferArray);
    //--- assign reinterpret_cast<unsigned char *>(cpuInitBufferArray[0])          [1]
    //--- to the SDK3 acquisition queue
  }

  //--- Allocate a single Output CPU Buffer per Path (same size as input)
  int AT_GPU_ConfigureOutputCpuBuffers(gpuHndl, 1, acqBufferSizeArray);

  //--- Start acquisition
  unsigned int iterationCount = 0;
  AT_GPU_U16 pathIndex;
  while ( iterationCount < 100 )
  {
    void *streamPtr;
    void **cpuBufferArray, **cpuOutputBufferArray, **gpuBufferArray;

    pathIndex = iterationCount%numRequiredPaths;
    //--- Wait for SDK3 Buffer to be acquired and Trigger next acquisition         [2]

    AT_GPU_CopyInputCpuToInputGpu(gpuHndl, pathIndex, 0);                          [3]

    AT_GPU_GetInputGpuBufferArray(gpuHndl, pathIndex, &gpuBufferArray);
    AT_GPU_GetStreamPtr(gpuHndl, pathIndex, &streamPtr);
    unsigned short inputVal = 10;
    addConstantOnGpuFunc(reinterpret_cast<unsigned short *>(gpuBufferArray[0]),
       inputVal, width, height, reinterpret_cast<cudaStream_t*>(streamPtr));       [4]

    AT_GPU_CopyInputGpuToOutputCpu(gpuHndl, pathIndex, 0);                         [5]

    AT_GPU_WaitPath(gpuHndl, pathIndex);
    AT_GPU_GetInputCpuBufferArray(gpuHndl, pathIndex, &cpuBufferArray);
    //--- Re-assign reinterpret_cast<unsigned char *>(cpuBufferArray[0])           [1]
    //--- to the SDK3 acquisition queue

    AT_GPU_GetOutputCpuBufferArray(gpuHndl, pathIndex, &cpuOutputBufferArray);
    //--- Save reinterpret_cast<unsigned short *>(cpuOutputBufferArray[0]) to file [6]

    iterationCount++;
  }

  //--- Finalise SDK3

  AT_GPU_Close(gpuHndl);
  AT_GPU_FinaliseLibrary( );

  return 0;
}// end main
```

3.6 User CPU Buffer Output

The following example applies the same processing as 3.5 Simple Acquisition Cycle, but in this case the result of the GPU processing is copied to a user managed CPU buffer for subsequent copying to file.

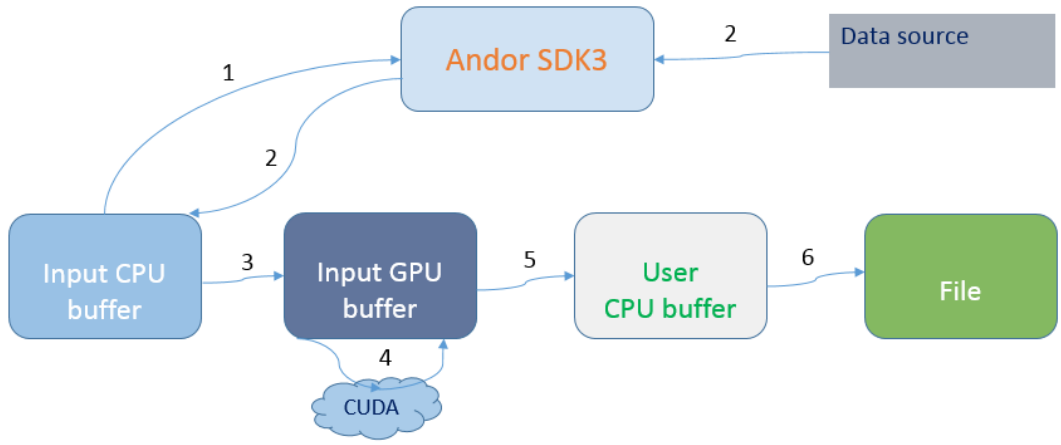
Note that the functions relating to the locking of the user CPU buffers do not require an input Handle. They may also therefore be called before initialisation of the library and do not require a Handle to be returned from the library. The functions are provided to the user simply to ensure that user managed buffers can be locked and unlocked for optimised copies from the specified GPU Buffers to user CPU memory. It is up to the user to ensure that these CPU Buffers are properly managed.

Note that in this example we also delay the copying to file of each processed output until the next iteration, this reduces time required for synchronisation within the acquisition cycle.

Updated code compared to 3.5 Simple Acquisition Cycle is highlighted in the code snippet below.

Example code relating to this example can be found in the provided 'SimpleSDK3AcqExamples' Visual Studio solution, within the 'userCpuBufferOutput' project. See B.1 Simple SDK3 Acquisition Examples for further details.

A flow chart for this acquisition cycle is shown below for a single GPU Express Path. The steps are repeated for each Path index in turn within the 'while' loop in the below code snippet until all iterations are complete. The steps listed are included in the code snippet for clarification:



1. *Input CPU buffer passed to SDK3 (via SDK3 function) and queued for acquisition*
2. *Frame acquired from Data Source to Input CPU buffer*
3. *GPU Express copy to Input GPU Buffer [AT_GPU_CopyInputCpuToInputGpu] (asynchronous wrt calling Host Thread)*
4. *CUDA Processing (asynchronous wrt calling Host Thread – if correct CUDA stream is passed to user function)*
5. *GPU Express copy to User CPU Buffer [CopyInputGpuToUserCpu] (asynchronous wrt calling Host Thread)*
6. *Result copied to file [in a subsequent iteration]*

Figure 10: Flow chart for a single GPU Express Path during an acquisition with User CPU Buffer

```
unsigned short ** p_outputDataPtr1 = NULL;
```

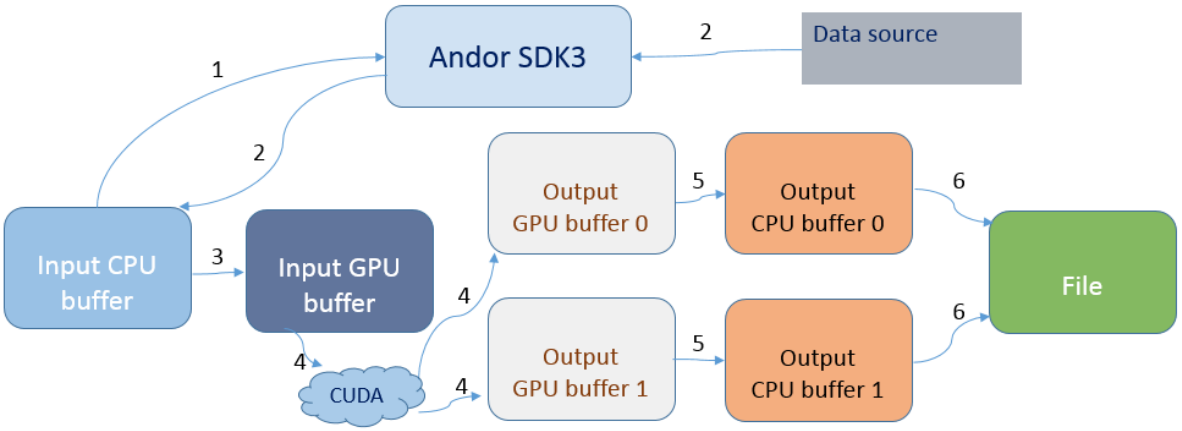
```
void **cpuInitBufferArray;

AT_GPU_H gpuHndl;

extern void addConstantOnGpuFunc(unsigned short * inputBuffer, unsigned short inputConst, unsigned
short bufferWidth, unsigned short bufferHeight, cudaStream_t *stream);

int main(int argc, char* argv[])
{
  //--- Initialise SDK3 and get Camera Handle
  //--- Configure acq. parameters and get required buffer size, width and height

  AT_GPU_InitialiseLibrary();
  AT_GPU_Open(&gpuHndl);

  AT_GPU_ConfigureSet(gpuHndl, numRequiredPaths);
  AT_GPU_ConfigureInputBuffers(gpuHndl, 1, acqBufferSizeArray);

  for (int i = 0; i < numRequiredPaths; i++) {
    AT_GPU_GetInputCpuBufferArray(Hndl, i, &cpuInitBufferArray);
    //--- assign reinterpret_cast<unsigned char *>(cpuInputBufferArray[0])          [1]
    //--- to the SDK3 acquisition queue
  }

  for (unsigned int i = 0; i < numRequiredPaths; ++i)  { // allocate all user buffers
    //--- Allocate p_outputDataPtr[i] with 'acqBufferSizeArray[0]' bytes, page aligned
    AT_GPU_LockOutputCpuBuf(p_outputDataPtr[i], acqBufferSizeArray[0]);
  }

  //--- Start acquisition
  int iterationCount = 0;
  AT_GPU_U16 pathIndex, prevPathIndex;
  while ( iterationCount < 100 )
  {
    void **cpuBufferArray, **gpuBufferArray, *streamPtr;

    int pathIndex = iterationCount%numRequiredPaths;
    //--- Wait for SDK3 Buffer to be acquired and Trigger next acquisition          [2]

    AT_GPU_CopyInputCpuToInputGpu(gpuHndl, pathIndex, 0);                          [3]

    AT_GPU_GetInputGpuBufferArray(gpuHndl, pathIndex, &gpuBufferArray);
    AT_GPU_GetStreamPtr(gpuHndl, pathIndex, &streamPtr);

    unsigned short inputVal = 10;
    addConstantOnGpuFunc(reinterpret_cast<unsigned short *>(gpuBufferArray[0]),
    inputVal, width, height, reinterpret_cast<cudaStream_t*>(streamPtr));          [4]

    AT_GPU_CopyInputGpuToUserCpu(gpuHndl, pathIndex, 0, p_outputDataPtr[pathIndex],
        acqBufferSizeArray[0])                                                     [5]

    if (iterationCount > 0) {
      //--- calculate path index used in previous iteration and save in        'prevPathIndex'
      AT_GPU_WaitPath(gpuHndl, prevPathIndex);
      AT_GPU_GetInputCpuBufferArray(Hndl, prevPathIndex, &cpuBufferArray);
      //--- Re-assign reinterpret_cast<unsigned char *>(cpuBufferArray[0])          [1]
      //--- to the SDK3 acquisition queue

      //--- Save p_outputDataPtr[prevPathIndex] to file                            [6]
    }
    iterationCount++;
  }
```

```
  //--- Finalise SDK3

  AT_GPU_Close(gpuHndl);
  AT_GPU_FinaliseLibrary( );

  if (p_outputDataPtr != NULL) {
    for (unsigned int i = 0; i < numRequiredPaths; ++i) {
      AT_GPU_UnlockOutputCpuBuf (p_outputDataPtr[i]);
      //--- Free 'p_outputDataPtr[i]'
    }
  }

  return 0;
}// end main
```

3.7 Output GPU Buffers – Multiple Outputs

The following example performs processing for a CUDA function that provides 2 outputs. In this case the results from the GPU processing are saved into each GPU Express Path's Output GPU Buffers, and are then copied to the GPU Express Path's Output CPU buffers for subsequent saving to disk. Updated code compared to 3.5 Simple Acquisition Cycle with respect to the use of Output GPU Buffers is highlighted below.
Example code relating to this example can be found in the provided 3.5 Simple Acquisition Cycle Visual Studio solution, within the 'outputGpuBuffersMultipleOutputs' project. See B.1 Simple SDK3 Acquisition Examples for further details.

A flow chart for this acquisition cycle is shown below for a single GPU Express Path. The steps are repeated for each Path index in turn within the 'while' loop in the below code snippet until all iterations are complete. The steps listed are included in the code snippet for clarification:



1. *Input CPU buffer passed to SDK3 (via SDK3 function) and queued for acquisition*
2. *Frame acquired from Data Source to Input CPU buffer*
3. *GPU Express copy to Input GPU Buffer [AT_GPU_CopyInputCpuToInputGpu] (asynchronous wrt calling Host Thread)*
4. *CUDA Processing (asynchronous wrt calling Host Thread – if correct CUDA stream is passed to user function)*
5. *GPU Express copy to Output CPU Buffers [AT_GPU_CopyOutputGpuToOutputCpu] (asynchronous wrt calling Host Thread)*
6. *Result copied to file [in a subsequent iteration]*

Figure 11: Flow chart for a single GPU Express Path during an acquisition with multiple outputs

```
//--- Initialise SDK3 and get Camera Handle

void **cpuInitBufferArray;

AT_GPU_H gpuHndl;

extern void addMultipleConstantsOnGpuFunc(unsigned short * inputBuffer,
       unsigned short * outputBuffer0, unsigned short * outputBuffer1,
       unsigned short inputConst0, unsigned short inputConst1,
       unsigned short bufferWidth, unsigned short bufferHeight, cudaStream_t *stream);

int main(int argc, char* argv[])
{
  //--- Initialise SDK3 and get Camera Handle
  //--- Configure acq. parameters and get required buffer size, width and height

  AT_GPU_InitialiseLibrary();
  AT_GPU_Open(&gpuHndl);

  AT_GPU_ConfigureSet(gpuHndl, numRequiredPaths);
  AT_GPU_ConfigureInputBuffers(gpuHndl, 1, acqBufferSizeArray);

  for (int i = 0; i < numRequiredPaths; i++) {
    AT_GPU_GetInputCpuBuffer(gpuHndl, i, &cpuInitBufferArray);
    //--- assign reinterpret_cast<unsigned char *>(cpuInitBufferArray[0])          [1]
    //--- to the SDK3 acquisition queue
  }
  AT_GPU_ConfigureOutputGpuBuffers(gpuHndl, 2, acqBufferSizeArray);
  AT_GPU_ConfigureOutputCpuBuffers(gpuHndl, 2, acqBufferSizeArray);

  //--- Start acquisition
  int iterationCount = 0;
  AT_GPU_U16 pathIndex, prevPathIndex;
  while ( iterationCount < 100 )
  {
    void *streamPtr;
    void **cpuBufferArray, **cpuOutputBufferArray, **gpuInputBufferArray,
         **gpuOutputBufferArray;

    int pathIndex = iterationCount%numRequiredPaths;                              [2]

    //--- Wait for SDK3 Buffer to be acquired and Trigger next acquisition
    AT_GPU_CopyInputCpuToInputGpu(gpuHndl, pathIndex, 0);                         [3]

    AT_GPU_GetInputGpuBuffer(gpuHndl, pathIndex, &gpuInputBufferArray);
    AT_GPU_GetOutputGpuBufferArray(gpuHndl, pathIndex, &gpuOutputBufferArray);
    AT_GPU_GetStreamPtr(gpuHndl, pathIndex, &streamPtr);

    unsigned short inputVal0 = 10;
    unsigned short inputVal1 = 20;
    addMultipleConstantsOnGpuFunc(
              reinterpret_cast<unsigned short *>(gpuInputBufferArray[0]),
              reinterpret_cast<unsigned short *>(gpuOutputBufferArray[0]),
              reinterpret_cast<unsigned short *>(gpuOutputBufferArray[1]),
              inputVal0, inputVal1, width, height, reinterpret_cast<cudaStream_t*>(streamPtr));
                                  [4]
    AT_GPU_CopyOutputGpuToOutputCpu(gpuHndl, pathIndex, 0);                       [5]
    AT_GPU_CopyOutputGpuToOutputCpu(gpuHndl, pathIndex, 1);                       [5]

    if (iterationCount > 0) {
      //--- calculate path index used in previous iteration and save in
      //---'prevPathIndex'
      AT_GPU_WaitPath(gpuHndl, prevPathIndex);
```

```
      AT_GPU_GetInputCpuBuffer(gpuHndl, prevPathIndex, &cpuBufferArray);
      //--- Re-assign reinterpret_cast<unsigned char *>(cpuBufferArray[0])        [1]
      //--- to the SDK3 acquisition queue

      AT_GPU_GetOutputCpuBufferArray(gpuHndl, prevPathIndex, &cpuBufferArray);
      //- Save reinterpret_cast<unsigned short *>(cpuOutputBufferArray[0]) to file [6]
      //- Save reinterpret_cast<unsigned short *>(cpuOutputBufferArray[1]) to file [6]
    }

    iterationCount++;
  }

  //--- Finalise SDK3

  AT_GPU_Close(gpuHndl);
  AT_GPU_FinaliseLibrary( );

  return 0;
}//end main
```

3.8 Dual camera

The following example applies the same processing as 3.5 Simple Acquisition Cycle, but in this case we process the output acquired from 2 cameras within the same cycle. The following code assumes a Master/Slave camera set up, with the Slave being triggered externally by the Master to acquire each frame. Updated code compared to 3.5 Simple Acquisition Cycle is highlighted below.

Example code relating to this example can be found in the provided 'SimpleSDK3AcqExamples' Visual Studio solution, within the 'dualCamera' project. See B.1 Simple SDK3 Acquisition Examples for further details.

```
void **cpuInitBufferArray;

AT_GPU_H gpuHndl;

extern void addConstantOnGpuFunc(unsigned short * inputBuffer, unsigned short inputConst, unsigned
short bufferWidth, unsigned short bufferHeight, cudaStream_t *stream);

int main(int argc, char* argv[])
{
  //--- Initialise SDK3 and get Camera Handle
  //--- Configure acq. parameters and get required buffer sizes, width and height

  AT_GPU_InitialiseLibrary();
  AT_GPU_Open(&gpuHndl);

  AT_GPU_ConfigureSet(gpuHndl, numRequiredPaths);
  AT_GPU_ConfigureInputBuffers(gpuHndl, 2, acqBufferSizeArray);
  for (int i = 0; i < numRequiredPaths; i++) {
    AT_GPU_GetInputCpuBufferArray(gpuHndl, i, &cpuBufferArray);
    //--- assign reinterpret_cast<unsigned char *>(cpuBufferArray[0])
    //--- to the 1st SDK3 acquisition queue

    AT_GPU_GetInputCpuBufferArray(gpuHndl, i, &cpuInitBufferArray);
    //--- assign reinterpret_cast<unsigned char *>(cpuInitBufferArray[1])
    //--- to the 2nd SDK3 acquisition queue
  }
```

```
  //--- Allocate 2 output CPU Buffers per Path (1 output for each camera)
  //--- (same size as input)
  AT_GPU_ConfigureOutputCpuBuffers(gpuHndl, 2, acqBufferSizeArray);

  //--- Start acquisition
  int iterationCount = 0;
  AT_GPU_U16 pathIndex;
  while ( iterationCount < 100 )
  {
    void *streamPtr;
    void **gpuBufferArray, **cpuBufferArray, **cpuOutputBufferArray;
    int pathIndex = iterationCount%numRequiredPaths;
    //--- Wait for Buffers to be acquired and Trigger next acquisition
    AT_GPU_CopyInputCpuToInputGpu(gpuHndl, pathIndex, 0);
    if (iterationCount > 0) {
      AT_GPU_CopyInputCpuToInputGpu (gpuHndl, pathIndex, 1);
    }

    unsigned short inputVal = 10;

    AT_GPU_GetInputGpuBufferArray(gpuHndl, pathIndex, &gpuBufferArray);
    AT_GPU_GetStreamPtr(gpuHndl, pathIndex, &streamPtr);

    addConstantOnGpuFunc(reinterpret_cast<unsigned short *>(gpuBufferArray[0]),
       inputVal, width, height, reinterpret_cast<cudaStream_t*>(streamPtr));

    addConstantOnGpuFunc(reinterpret_cast<unsigned short *>(gpuBufferArray[1]),
       inputVal, width, height, reinterpret_cast<cudaStream_t*>(streamPtr));

    AT_GPU_CopyInputGpuToOutputCpu(gpuHndl, pathIndex, 0);
    AT_GPU_CopyInputCpuToOutputCpu(gpuHndl, pathIndex, 1);

    AT_GPU_WaitPath(gpuHndl, pathIndex);
    AT_GPU_GetInputCpuBufferArray(gpuHndl, pathIndex, &cpuBufferArray);
    //--- Re-assign reinterpret_cast<unsigned char *>(cpuBufferArray[0])
    //--- to the SDK3 acquisition queue

    //--- Re-assign reinterpret_cast<unsigned char *>(cpuBufferArray[1])
    //--- to the SDK3 acquisition queue

    AT_GPU_GetOutputCpuBufferArray(gpuHndl, pathIndex, &cpuBufferArray);
    //--- Save reinterpret_cast<unsigned short *>(cpuOutputBufferArray[0]) to file
    //--- Save reinterpret_cast<unsigned short *>(cpuOutputBufferArray[1]) to file

    iterationCount++;
  }

  //--- Finalise SDK3

  AT_GPU_Close(gpuHndl0);
  AT_GPU_FinaliseLibrary( );

  return 0;
}//end main
```

3.9 Callback Functionality

This section gives an example of the callback functionality provided by the library via the library function 2.10.1 AT_GPU_CallUserFunction. With this function, the user does not have to declare any variables to hold references to library managed resources, and less function calls are required to pass the library resources through to the user defined function. Updated code compared to 3.5 Simple Acquisition Cycle with respect to the use of Output GPU Buffers is highlighted below.

Example code relating to this example can be found in the provided 'SimpleSDK3AcqExamples' Visual Studio solution, within the 'callbackFunctionality' project. See B.1 Simple SDK3 Acquisition Examples for further details.

```
//--- Initialise SDK3 and get Camera Handle

void ** cpuInitBufferArray;

AT_GPU_H gpuHndl;

extern void addConstantOnGpuFunc(unsigned short * inputBuffer,
       unsigned short inputConst, unsigned short bufferWidth,
       unsigned short bufferHeight, cudaStream_t *stream);

struct userDataStruct{
       int inputConst;
       unsigned short bufferWidth;
       unsigned short bufferHeight;
} myStruct;

int addConstantOnGpu_Callback(void** InputGpuBufArray, void** OutputGpuBufArray,
                              void * streamPtr, void * userDataStr)
{
  addConstantOnGpuFunc( reinterpret_cast<unsigned short*>(InputGpuBufArray[0]),
       reinterpret_cast<userDataStruct*>(userDataStr)->inputConst,
        reinterpret_cast<userDataStruct*>(userDataStr)->bufferWidth,
        reinterpret_cast<userDataStruct*>(userDataStr)->bufferHeight,
        reinterpret_cast<cudaStream_t*>(streamPtr) );
  return 0;
};

int main(int argc, char* argv[])
{
  //--- Initialise SDK3 and get Camera Handle
  //--- Configure acq. parameters and get required buffer size, width and height

  AT_GPU_InitialiseLibrary( );
  AT_GPU_Open(&gpuHndl);

  //--- Configure acquisition Parameters and get required buffer size
  AT_GPU_ConfigureSet(gpuHndl, numRequiredPaths);
  AT_GPU_ConfigureInputBuffers(gpuHndl, 1, acqBufferSizeArray);

  for (int i = 0; i < numRequiredPaths; i++) {
    AT_GPU_GetInputCpuBuffer(gpuHndl, i, &cpuInitBufferArray);
    //--- assign reinterpret_cast<unsigned char *>(cpuInitBufferArray[0])
    //--- to the SDK3 acquisition queue
  }

  //--- Allocate a single Output CPU Buffer per Path (same size as input)

  int AT_GPU_ConfigureOutputCpuBuffers(gpuHndl, 1, acqBufferSizeArray);
```

```
  //--- Start acquisition
  int iterationCount = 0;
  AT_GPU_U16 pathIndex;
  while ( iterationCount < 100 )
  {
    void **cpuBufferArray, **cpuOutputBufferArray;
    int pathIndex = iterationCount%numRequiredPaths;
    //--- Wait for SDK3 Buffer to be acquired and Trigger next acquisition

    AT_GPU_CopyInputCpuToInputGpu(gpuHndl, pathIndex, 0);
    myStruct.inputVal = 10;
    myStruct.width = width;
    myStruct.height = height;
    AT_GPU_CallUserFunction(gpuHndl, pathIndex, &addConstantOnGpu_Callback,
                            (void*)(&myStruct));

    AT_GPU_CopyInputGpuToOutputCpu(gpuHndl, pathIndex, 0);

    AT_GPU_WaitPath(gpuHndl, pathIndex);
    AT_GPU_GetInputCpuBuffer(gpuHndl, pathIndex, &cpuBufferArray);
    //--- Re-assign reinterpret_cast<unsigned char *>(cpuBufferArray[0])
    //--- to the SDK3 acquisition queue

    AT_GPU_GetOutputCpuBufferArray(Hndl, pathIndex, &cpuOutputBufferArray);
    //--- Save reinterpret_cast<unsigned short *>(cpuOutputBufferArray[0]) to file
    iterationCount++;
  }

  //--- Finalise SDK3

  AT_GPU_Close(gpuHndl0);
  AT_GPU_FinaliseLibrary( );

  return 0;
}//end main
```

3.10 CUDA Streaming (single copy engine approach)

The following example applies the same processing as 3.5 Simple Acquisition Cycle, but runs the GPU Express Paths in ‘batches’ to provide a basis to overlap the copies and processing within each Path. This is possible as each Path carries out its copies within its own unique CUDA stream, and can return a reference to this stream to the user (so they can apply processing on the GPU within this stream for the current Path).

Note that this type of setup is only recommended to overlap copies and processing on a single copy engine GPU card.

Note that to see improved results with CUDA streaming during acquisition it is necessary to use either an Internal or External Trigger, or if using a Software Trigger to utilise it with an SDK3 acquisition callback. Note also that any (implicit or explicit) synchronisation in the CUDA code to be called shall minimise any performance improvement.

The SDK3 callback can be used to trigger another acquisition on end-of-exposure event. In order to prevent the on-board buffers on the camera overflowing in this case (in order to provide a more consistent output without the need to flush the buffers and re-start the acquisition, the SDK3 callback should include functionality to restrict the number of Software Triggers). An example of such a callback is included below for complete-ness. See ‘Andor Software

Development Kit 3.pdf', and the included example test suite for further details. Example code relating to this example can be found in the provided 'SimpleSDK3AcqExamples' Visual Studio solution, within the 'simpleAcquistionCycle' project. See B.1 Simple SDK3 Acquisition Examples.

```
//-----------------------------------------------------------------
//--- example SDK3 acquisition callback with restricted SW Trigger
//-----------------------------------------------------------------

int AT_EXP_CONV acquisitionCallback(AT_H Hndl, const AT_WC* Feature, void* Context)
{
  if (!b_firstAccess) {
    eventContext * context = reinterpret_cast<eventContext *>(Context);
    if (context->triggerCount <= context->frameCount + context->evNumBuffers ) {
      context->b_cbSuccess = true;
      context->triggerCount++;
      AT_Command(Hndl, L"Software Trigger");
    }
    else {
      context->b_cbSuccess = false;
    }
  }
  else {
    b_firstAccess = false;
  }
  return AT_CALLBACK_SUCCESS;
}


//-----------------------------------------------------------------
//--- CUDA Streaming acquisition Example
//-----------------------------------------------------------------

//--- Initialise SDK3 and get Camera Handle

void **cpuInitBufferArray;

AT_GPU_H gpuHndl;

extern void addConstantOnGpuFunc(unsigned short * inputBuffer,
              unsigned short inputConst, unsigned short bufferWidth,
              unsigned short bufferHeight, cudaStream_t *stream);

int main(int argc, char* argv[])
{
  //--- Initialise SDK3 and get Camera Handle
  //--- Configure acq. parameters and get required buffer size, width and height

  AT_GPU_InitialiseLibrary( );
  AT_GPU_Open(&Hndl);

  //--- Configure acquisition Parameters and get required buffer size
  AT_GPU_ConfigureSet(gpuHndl, numRequiredPaths, acqBufferSize);
  AT_GPU_ConfigureInputBuffers(gpuHndl, 1, acqBufferSizeArray);

  for (int i = 0; i < numRequiredPaths; i++) {
    AT_GPU_GetInputCpuBuffer(gpuHndl, i, &cpuInitBufferArray);
    //--- assign reinterpret_cast<unsigned char *>(cpuInitBufferArray[0])
    //--- to the SDK3 acquisition queue
  }

  //--- Allocate a single Output CPU Buffer per Path (same size as input)
```

```
int AT_GPU_ConfigureOutputCpuBuffers(gpuHndl, 1, acqBufferSizeArray);

unsigned int batchSize = 4;

glb_evContext.evNumBuffers = static_cast<int>(numRequiredPaths);
glb_evContext.frameCount = 0;
glb_evContext.triggerCount = 0;
glb_evContext.b_cbSuccess = true;

//--- Start acquisition
int iterationCount = 0;
AT_GPU_U16 pathIndex, prevPathIndex, pathBatchStartIndex, pathBatchEndIndex;

while ( iterationCount < (100/batchSize)
{
  void *streamPtr;
  void **cpuBufferArray, **cpuOutputBufferArray, gpuBufferArray;

  pathBatchStartIndex = glb_evContext.frameCount % numRequiredPaths;
  pathBatchEndIndex = pathBatchStartIndex + batchSize;

  //--- 1st 'batch operation' - copy from GPU to CPU within each Path in Batch
  for (pathIndex = pathBatchStartIndex; pathIndex < pathBatchEndIndex; pathIndex++)
  {
    //--- Wait for Buffer to be acquired
    glb_evContext.frameCount++;
    AT_GPU_CopyInputCpuToInputGpu(gpuHndl, pathIndex, 0);
  }

  unsigned short inputVal = 10;
  //--- 2nd 'batch operation' - process on GPU for each Path in Batch
  int inputInt = 10;
  for (pathIndex = pathBatchStartIndex; pathIndex < pathBatchEndIndex; pathIndex++)
  {
    AT_GPU_GetInputGpuBuffer(gpuHndl, pathIndex, &gpuBufferArray);
    AT_GPU_GetStreamPtr(gpuHndl, pathIndex, &streamPtr);
    addConstantOnGpuFunc(reinterpret_cast<unsigned short *>(gpuBufferArray[0]),
     inputVal, width, height, reinterpret_cast<cudaStream_t*>(streamPtr));
  }

  //--- 3rd 'batch operation' - copy from CPU to GPU within each Path in Batch
  for (pathIndex = pathBatchStartIndex; pathIndex < pathBatchEndIndex; pathIndex++)
  {
    AT_GPU_CopyInputGpuToOutputCpu(gpuHndl, pathIndex, 0);
  }

  if (iterationCount > 0) {
     for (pathIndex = pathBatchStartIndex; pathIndex < pathBatchEndIndex;
             pathIndex++)
     {
       //--- calculate path index used in previous iteration and save in
       //--- 'prevPathIndex'

       AT_GPU_WaitPath(gpuHndl, prevPathIndex);
       AT_GPU_GetInputCpuBuffer(gpuHndl, prevPathIndex, &cpuBufferArray);
       //--- Re-assign reinterpret_cast<unsigned char *>(cpuBufferArray[0])
       //--- to the SDK3 acquisition queue

       AT_GPU_GetOutputCpuBufferArray(gpuHndl, prevPathIndex,
            &cpuOutputBufferArray);
       //- Save reinterpret_cast<unsigned short *>(cpuOutputBufferArray[0]) to file
     }
  }
  if (!glb_evContext.b_cbSuccess) {
```

```
      glb_evContext.triggerCount++;
      //--- Call SDK3 Software Trigger
    }
    iterationCount++;
  }

  //--- Finalise SDK3

  AT_GPU_Close(gpuHndl);
  AT_GPU_FinaliseLibrary();

  return 0;
}//end main
```

3.11 Multiple GPU Processing

The following example applies the same processing as 3.5 Simple Acquisition Cycle, but in this case we process the output buffer acquired from a single camera on one of 2 GPU cards, dependent on Path index. Updated code compared to 3.5 Simple Acquisition Cycle is highlighted below.

Example code relating to this example can be found in the provided 'SimpleSDK3AcqExamples' Visual Studio solution, within the 'multiGpu' project. See B.1 Simple SDK3 Acquisition Examples for further details.

```
void **cpuInitBufferArray;
AT_GPU_U64 * acqBufferSizeArray;

AT_GPU_H gpuHndl;

extern void addConstantOnGpuFunc(unsigned short * inputBuffer, unsigned short inputConst, unsigned
short bufferWidth, unsigned short bufferHeight, cudaStream_t *stream);

int main(int argc, char* argv[])
{

  AT_GPU_U16 numDevices;
  ret = AT_GPU_GetNumGpuCards(&numDevices);
  if (numDevices < 2 || ret != AT_GPU_SUCCESS) {
    return -1; //-- return if number of cards is less than 2 in this case
  }


  //--- Initialise SDK3 and get Camera Handle
  //--- Configure acq. parameters and get required buffer size(s), width and height

  AT_GPU_InitialiseLibrary();
  AT_GPU_CheckGpuCapability();
  AT_GPU_Open(&gpuHndl);

  AT_GPU_ConfigureSet(gpuHndl, numRequiredPaths);
  //--- Set Gpu card id for each Path
  for (unsigned int i = 0; i < numRequiredPaths; ++i) {
    if (i >= (numRequiredPaths/2)) {
      ret = AT_GPU_ConfigureGpuCardIdToPath(gpuHndl, i, 1);
    }
  }
  AT_GPU_ConfigureInputBuffers(gpuHndl, 1, acqBufferSizeArray);

  for (unsigned int i = 0; i < numRequiredPaths; i++) {
    AT_GPU_GetInputCpuBufferArray(gpuHndl, i, &cpuInitBufferArray);
    //--- assign reinterpret_cast<unsigned char *>(cpuInitBufferArray[0])          [1]
```

```
    //--- to the SDK3 acquisition queue
  }

  //--- Allocate a single Output CPU Buffer per Path (same size as input)
  int AT_GPU_ConfigureOutputCpuBuffers(gpuHndl, 1, acqBufferSizeArray);

  //--- Start acquisition
  unsigned int iterationCount = 0;
  AT_GPU_U16 pathIndex;
  while ( iterationCount < 100 )
  {
    void *streamPtr;
    void **cpuBufferArray, **cpuOutputBufferArray, **gpuBufferArray;

    pathIndex = iterationCount%numRequiredPaths;
    //--- Wait for SDK3 Buffer to be acquired and Trigger next acquisition          [2]

    AT_GPU_CopyInputCpuToInputGpu(gpuHndl, pathIndex, 0);                          [3]

    AT_GPU_GetInputGpuBufferArray(gpuHndl, pathIndex, &gpuBufferArray);
    AT_GPU_GetStreamPtr(gpuHndl, pathIndex, &streamPtr);
    unsigned short inputVal = 10;
    addConstantOnGpuFunc(reinterpret_cast<unsigned short *>(gpuBufferArray[0]),
       inputVal, width, height, reinterpret_cast<cudaStream_t*>(streamPtr));       [4]

    AT_GPU_CopyInputGpuToOutputCpu(gpuHndl, pathIndex, 0);                         [5]

    AT_GPU_WaitPath(gpuHndl, pathIndex);
    AT_GPU_GetInputCpuBufferArray(gpuHndl, pathIndex, &cpuBufferArray);
    //--- Re-assign reinterpret_cast<unsigned char *>(cpuBufferArray[0])           [1]
    //--- to the SDK3 acquisition queue

    AT_GPU_GetOutputCpuBufferArray(gpuHndl, pathIndex, &cpuOutputBufferArray);
    //-- Save reinterpret_cast<unsigned short *>(cpuOutputBufferArray[0]) to file [6]

    iterationCount++;
  }

  //--- Finalise SDK3

  AT_GPU_Close(gpuHndl);
  AT_GPU_FinaliseLibrary( );

  return 0;
}// end main
```

3.12 Note on Synchronization

Synchronization between copies occurs implicitly within the GPU Express library. That is, no copy or processing on the GPU for a specified GPU Express Path shall take place before all previous copies **AND** processing have been completed within the unique GPU Express Path associated with that buffer.

The synchronization occurs only on a per GPU Express Path basis, i.e. all copies are asynchronous with respect to the CPU host thread. When batching GPU Express Paths, this allows data associated with different CUDA streams to be copied / processed in parallel (assuming that any user defined CUDA functions take the correct CUDA stream as a parameter).

As such, in the previous tutorial code snippets, the

```
AT_GPU_CopyInputCpuToInputGpu(Hndl, pathIndex);
```

call acts as a synchronization point for the indexed GPU Express Path to the previous iteration's (within the 'while' loop) CUDA calls (copies and processing) with respect to that GPU Express Path.

```
AT_GPU_WaitPath(Hndl, pathIndex);
```

is therefore only required to ensure that all processing and copies previously requested for a particular GPU Express Path are complete, e.g. in the above examples the CPU host thread waits at this point before re-assigning the CPU buffer for the indexed Path to the SDK3 engine.

APPENDIX A DeblurNN LIBRARY

A.1 Introduction

Examples 1 and 2 in the included test suite utilise the included 'DeblurNN' library. This library must be dynamically loaded by the calling program, and the 'DeblurNN.dll' must therefore be present in the calling executable's directory. Two files to facilitate loading the DLL is provided to the user (*DeblurNN_DLLLoader.cpp* and *DeblurNN_DLLLoader.h*). These can be found under the '*GPU Express*' folder in the installation directory.

Once loaded, the library provides a number of functions implemented in CUDA to apply the well-known 'Nearest Neighbour' and 'No Neighbour' algorithms for 3D and 2D input datasets respectively. The output from an intermediate de-noising stage of the algorithm is also provided as an option to the end-user.

The Nearest Neighbour algorithm is based upon improving an image by sharpening the edges of structures, and falls under the category of a de-blurring algorithm. It is a specific type of sharpening filter called an unsharp mask.

A.2 Nearest-Neighbour Deblurring

The Nearest-Neighbour filter starts with a 3D stack of 2D images. For each 2D image, a Gaussian smoothing step is applied. The sharp image can then be recovered by subtracting the neighbouring two smoothed images from the middle blurred (original) image. The Gaussian smoothing, called the kNN filter, is given by

$$g(x)=\frac{1}{C(x)}\int_{\Omega(x)} y(u)e^{|u-x|^2/r^2}e^{|y(u)-y(x)|^2/h^2}du \qquad \textit{Equation 1}$$

Where $u(x)$ is the original blurred image, *h* is noise level parameter, *r* is the traditional Gaussian blur coefficient, $C(x)$ is the normalizing coefficient, and $\Omega(x)$ is the neighbourhood area around the pixel *x*. From Equation 1 we can see that the kNN filter smoothes the image according to the location (1st exponential term) and pixel value (2nd exponential term) distances between the centre pixel and its surrounding neighbourhood pixels. The second step is image subtraction:

$$o(x)=y(x)-w\times\left[g(x'')+g(x')\right] \qquad \textit{Equation 2}$$

Where *x*, x' and x'' denote the coordinates of the *z*, *z+1* and *z-1* planes respectively.

The assumption behind this method is that most of blur in the focal plane (middle) image is contributed by the neighbouring image planes. Although it is simple and fast, there are several disadvantages, e.g., it can't remove noise and introduces undesirable artefacts [2].

A.3 Nearest-Neighbour Algorithm

The flow chart for the Nearest-Neighbour algorithm is shown in Figure A.2. Firstly, the *z+1* and *z-1* z-planes are smoothed using a kNN filter. The pseudo-code of the kNN filter is shown in Figure A. 1.

***Input**: Noisy image N; Parameters: r; h.*

***Output**: Smoothed image.*

Main algorithm:

For each pixel P in N

1. Get pixels in surrounding area (r × r)

2. Calculate distances: location distance $\frac{|y-x|^2}{r^2}$ *and pixel value distance* $\frac{|u(y)-u(x)|^2}{h^2}$

3. Calculate weights $e^{\frac{|y-x|^2}{r^2}} e^{\frac{|u(y)-u(x)|^2}{h^2}}$

4. Calculate normalizing coefficient $C(x)$ as $\sum_{\Omega(x)} e^{\frac{|y-x|^2}{r^2}} e^{\frac{|u(y)-u(x)|^2}{h^2}}$

5. Calculate the new pixel according to Equation 1.

6. Blend the new generated pixel and original pixel into final pixel based on some heuristics

Figure A. 1: Pseudo code for the Gaussian smoothing algorithm

In step 1, the pixels in the kNN neighbourhood are acquired, eg. 7x7. Two distances are then calculated. One is to do with the geometric distance of each pixel in the neighbourhood from the central pixel. The other distance is in intensity space, and is the intensity difference between the neighbourhood pixel and the central pixel. In step 3 the weights for each neighbourhood pixel are calculated. In step 4 these are then summed over the neighbourhood to give the normalising coefficient.

In step 5, the new pixel is calculated as weighted sum of the intensities divided by the normalising coefficient. When the algorithm returns the final output pixel, step 6, it utilizes some heuristics which involves three parameters; lerp coefficient, weight threshold and counter threshold. Firstly, the weight is compared to the KNN_WEIGHT_THRESHOLD. The percentage of weights that are greater than KNN_WEIGHT_THRESHOLD is accumulated into the fCount variable. Secondly, when blending the pixel, a simple check determines whether the block is “smooth”: if fCount exceeds the KNN_LERP_THRESHOLD, the block is considered to be smooth enough, so the filtered pixel value should have more weight in the output than the original (noisy) input pixel value. Otherwise the block is presumed to contain edges or small features, and, in order to give more weight to the original input pixel value in the output and retain important visual image properties, the blending quotient lerpC is subtracted from one. Following smoothing of the planes, the final step is image subtraction as defined by Equation 2.

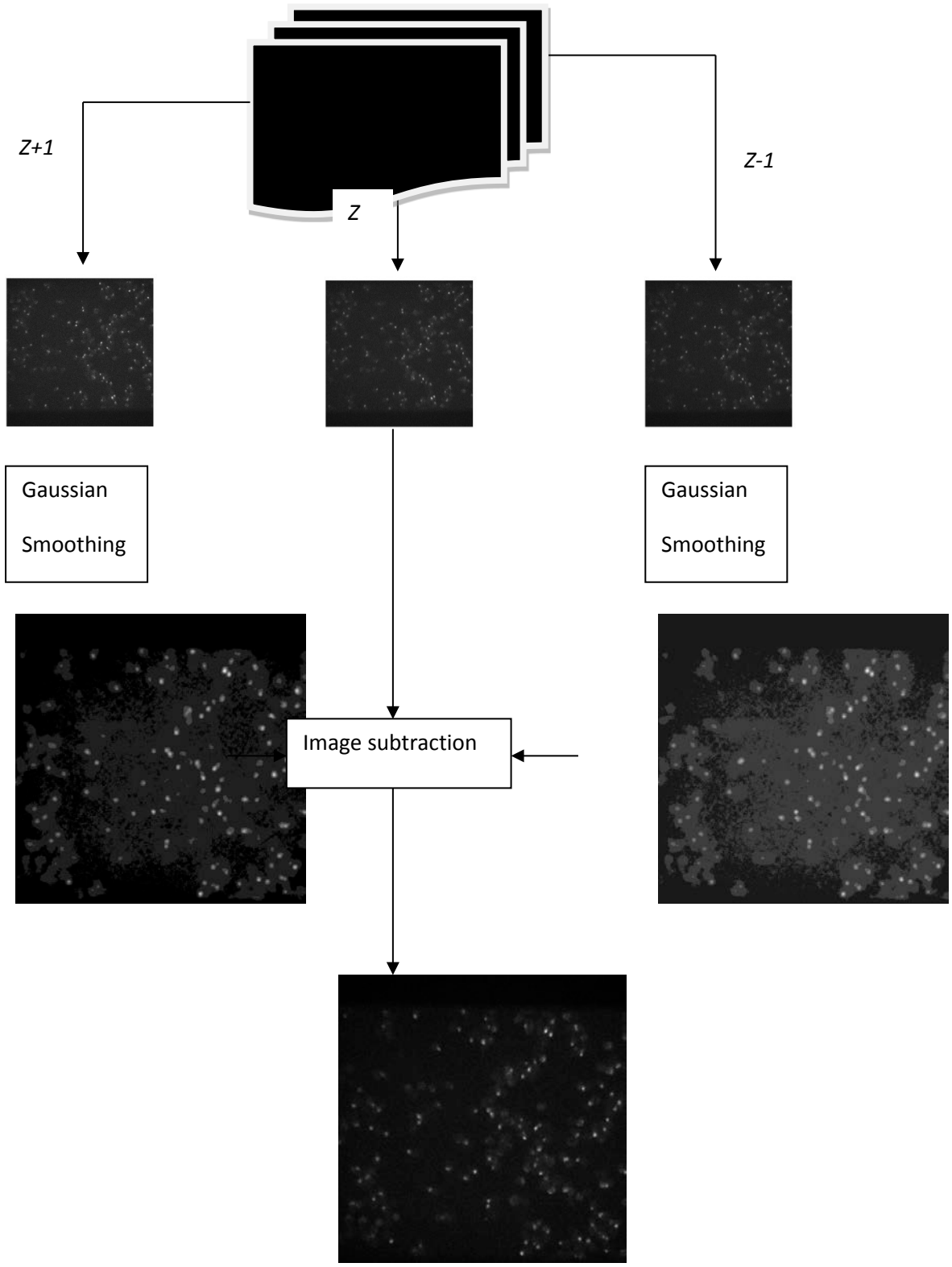


Figure A.2: Flow-chart of the nearest-neighbour deconvolution algorithm.

A.4 No-Neighbour Algorithm

As opposed to the Nearest-Neighbour algorithm, the No-Neighbour simply takes a single frame as input, applies Gaussian smoothing, and subtracts this result from itself. This is therefore applicable to 2D datasets.

A.5 Algorithm Parameters

A.6 References

[1] P. Pankajakshan, *et al*. Parametric blind deconvolution for confocal laser scanning microscopy (CLSM)-proof of concept. Research Report 6493, INRIA, Sophia-Antipolis, France. 2008.

[2] P. Sarder and A. Nehorai. Deconvolution methods for 3D fluorescence microscopy images: An overview. IEEE Signal Processing Magazine, vol. 23, no. 3, pp. 32-45, May 2006.

[3] D.A. Agard. Optical sectioning microscopy: Cellular architecture in three Dimensions. Ann. Rev. Biophys. Bioeng., vol. 13, pp. 191–219, 1984.

A.7 Nearest-Neighbour API Description

A.7.1 Function Listing

```
int AT_NN_SetDimensions( int _i_imageWidth, int _i_imageHeight, int _i_imageDepth);

int AT_NN_ClearDimensions();

int AT_NN_CheckGpuCapability();

char * AT_NN_GetLastError();

int AT_NN_ApplyNearestNeighbourDeblur(unsigned short * _pus_gpuImage,
int _i_knnWindowRadius, float _f_nnWeight, bool _b_applyScaling,
cudaStream_t _stream, int _i_strmIndex, int _i_numCUDAStreams);

int AT_NN_ApplyNoNeighbourDeblur(unsigned short * _pus_gpuImage,
int _i_knnWindowRadius, float _f_nnWeight, bool _b_applyScaling,
cudaStream_t _stream, int _i_strmIndex, int _i_numCUDAStreams);

int AT_NN_ApplyKNNDenoising(unsigned short * _pusgpuImage, int _i_knnWindowRadius,
cudaStream_t _stream, int _i_strmIndex, int _i_numCUDAStreams);
```

A.7.2 AT_NN_SetDimensions

```
int AT_NN_SetDimensions (int _i_imageWidth, int _i_imageHeight, int _i_imageDepth);
```

is the first call required into the DeblurNN library. This allocates all required memory and sets up all internal data structures required to process an image with the specified input parameter dimensions.

The 3 parameters are simply the width, height and depth of subsequent datasets to be processed. To process a dataset of different size, 'AT_NN_ClearDimensions' must be called followed by another call into this 'AT_NN_SetDimensions' function.

A.7.3 AT_NN_ClearDimensions

```
int AT_NN_ClearDimensions ();
```

deallocates all allocated memory and cleans up all resources within the library.

A.7.4 AT_NN_CheckGPU

```
int AT_NN_CheckGPU ();
```

may be used to verify that your system is compatible with the CUDA requirements for the library. The function returns an error message indicating the cause of the incompatibility if applicable; otherwise it returns a success return code.

A.7.5 AT_NN_GetLastError

```
char * AT_NN_GetLastError()
```

retrieves an error string associated with the last error found by the library.

A.7.6 AT_NN_ApplyNearestNeighbourDeblur

```
int AT_NN_ApplyNearestNeighbourDeblur(unsigned short * _pusgpuImage,
int _i_knnWindowRadius, float _f_nnWeight, bool _b_applyScaling,
cudaStream_t _stream, int _i_strmIndex, int _i_numCUDAStreams);
```

applies the Nearest-Neighbour deblurring algorithm to the input dataset on the GPU, as pointed to by the 1st parameter.

The 2nd parameter provides the radius (in terms of number of pixels) of the Gaussian smoothing window used for the kNN Gaussian smoothing. The 3rd parameter is the weighting used for the final subtraction in the Nearest Neighbour algorithm. This should be a 'float' value between 0.0f and 1.0f. See A.3 Nearest-Neighbour Algorithm for more information on these values.

The 4th parameter applies a scaling to the final output if set to true. The scaling keeps track of the smallest value in the output during processing, and adds this to all pixels at the end of the processing if less than zero. Otherwise, all negative values are set to zero.

The 5th parameter passes through a user (or GPU Express) managed CUDA stream, the 6th parameter is the index of the stream (within a set of overlapping streams), and the 7th parameter is the total number of overlapping CUDA streams to be used for processing.

For a non CUDA-overlapping approach, the final 2 parameters should simply be set to integer values of 0 and 1 respectively. Note that this function expects the input image to have a depth of at least 3 Z-planes.

A.7.7 AT_NN_ApplyNoNeighbourDeblur

```
int AT_NN_ApplyNoNeighbourDeblur(unsigned short * _pusgpuImage,
int _i_knnWindowRadius, float _f_nnWeight, bool _b_applyScaling,
cudaStream_t _stream, int _i_strmIndex, int _i_numCUDAStreams);
```

applies the No-Neighbour deblurring algorithm to the input dataset on the GPU, as pointed to by the 1st parameter.

The 2nd parameter provides the radius (in terms of number of pixels) of the Gaussian smoothing window used for the kNN Gaussian smoothing. The 3rd parameter is the weighting used for the final subtraction in the Nearest Neighbour algorithm. This should be a 'float' value between 0.0f and 1.0f. See A.3 Nearest-Neighbour Algorithm for more information on these values.

The 4th parameter applies a scaling to the final output if set to true. The scaling keeps track of the smallest value in the output during processing, and adds this to all pixels at the end of the processing if less than zero. Otherwise, all negative values are set to zero.

The 5th parameter passes through a user (or GPU Express) managed CUDA stream, the 6th parameter is the index of the stream (within a set of overlapping streams), and the 7th parameter is the total number of overlapping CUDA streams to be used for processing.

For a non CUDA-overlapping approach, the final 2 parameters should simply be set to integer values of 0 and 1 respectively.

A.7.8 AT_NN_ApplyKNNDenoising

```
int AT_NN_ApplyKNNDenoising(unsigned short * _pusgpuImage,
int _i_knnWindowRadius,
cudaStream_t _stream, int _i_strmIndex, int _i_numCUDAStreams);
```

applies simply the 1st stage of the No/Nearest Neighbour deblurring algorithm (i.e. the kNN Gaussian smoothing) to the input dataset on the GPU, as pointed to by the 1st parameter. This is applicable for both a 2D or 3D dataset.

The 2nd parameter provides the radius of the Gaussian smoothing window.

The 3rd parameter passes through a user (or GPU Express) managed CUDA stream, the 4th parameter is the index of the stream (within a set of overlapping streams), and the 5th parameter is the total number of overlapping CUDA streams to be used for processing.

For a non CUDA-overlapping approach, the final 2 parameters should simply be set to integer values of 0 and 1 respectively.

APPENDIX B INCLUDED EXAMPLE SUITES

B.1 Simple SDK3 Acquisition Examples

The Simple SDK3 Acquisition Examples are found in the 'SimpleSDK3AcqExamples' folder in the installation directory. The suite contains 8 sample console applications (8 Visual Studio Projects within the 'SimpleSDK3AcqExamples' Visual Studio solution), illustrating a number of features available with the GPU Express library. Each application relates directly to a section in the above tutorial (SECTION 3:), apart from 'multiGpuMultiThread' which is an additional example providing a multi-threaded version of the 'multiGpu' project (this example requires a C++11 compliant compiler). A C++ file with the same name as each project includes the code for the relevant section of the tutorial (with added code for SDK3 calls and error checking).

The examples are simple console applications, which prompt the user for input parameters before running the required processing. All the examples require at least 1 Andor sCMOS camera to be attached and running.

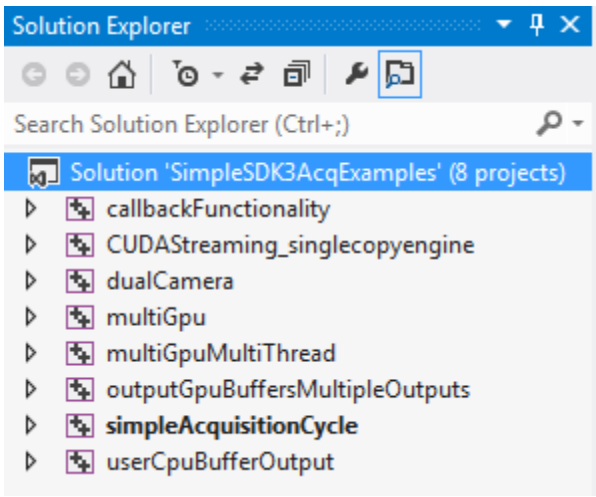


Figure B.3 Screen-shot of Simple SDK3 Acquisition Example Suite

B.2 Advanced Examples

Also included is an example suite that contains 7 further sample console applications (7 Visual Studio Projects within the 'AdvancedSDK3ACqExamples' Visual Studio solution), illustrating use with a 3rd party library. These examples are found in the 'AdvancedSDK3AcqExamples' folder in the installation directory.

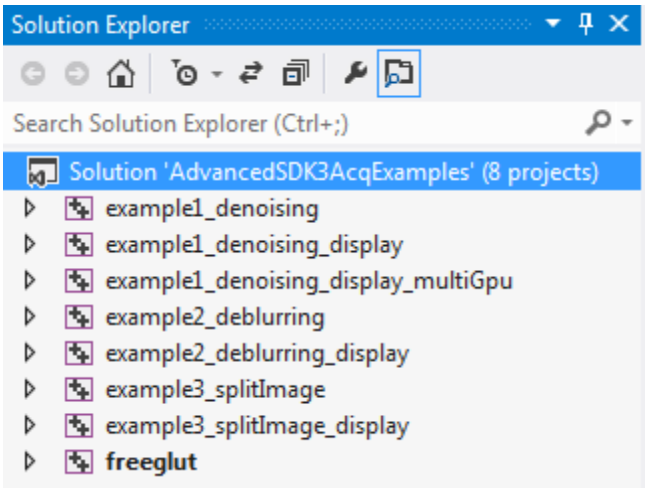


Figure B.2 Screen-shot of Advanced SDK3 Acquisition Example Suite

Again, the examples are simple console applications, which prompt the user for input parameters before running the required processing. All the examples require at least 1 Andor sCMOS camera to be attached and running.

All 3 examples above also have a 'display' version, which applies the same processing in each case, and also outputs the live or processed image(s) on screen if requested. These examples utilise openGL and the included 'freeglut' utility library for output for images on-screen. The included 'BFGlutUI.cpp' file and associated header file provide all the required functionality.

Examples 1 and 2 use the provided Andor CUDA enabled library 'DeblurNN' (see APPENDIX A for further details). In both cases the examples use buffers managed exclusively by the GPU Express library for all copies.

Example 1 acquires the number of frames specified by the user and applies a Gaussian de-noising algorithm on the GPU (A.7.8 AT_NN_ApplyKNNDenoising) to each before saving the result to file if requested. Note that this example does not apply any unpacking to the acquired images so that granularity limitations must be met by adjusting the size of the Region of Interest through the console input to meet the limitations as set by the Camera Link frame grabber if applicable (for example, the width must be a multiple of 10 for a 10-tap Camera Link frame grabber and 3 for a 3-tap Camera Link frame grabber). Both examples follow the flow chart shown in 3.5 Simple Acquisition Cycle.

The display version of Example 1 also shows an example usage of 2.10 Calling a User Defined Function.

A multiple GPU version of Example 1 has also been added. This example splits the processing of alternate GPU Express Paths utilising a 3rd party library between 2 GPU cards, and provides examples of 2.4.3 AT_GPU_ConfigureGpuCardIdToPath and 2.4.3 AT_GPU_SetGpuCard in use.

Example 2 similarly acquires the number of frames specified by the user and in this case applies a full de-blurring algorithm on the GPU to each (A.7.7 AT_NN_ApplyNoNeighbourDeblur) before again saving the result to file if required. This example includes a call to the CUDA unpacking library (See Appendix C for further details). Both examples again follow the flow chart shown in 3.5 Simple Acquisition Cycle.

As opposed to examples 1 and 2 which have 1 input and 1 output frame of equal dimensions, example 3 takes each input frame and splits it into 2 separate images on the GPU. In this case, 2 output GPU buffers are required which are again managed by the GPU Express library.

In the non-display example, both Output GPU buffers are copied to Output CPU buffers via 2.7.4 AT_GPU_CopyOutputGpuToOutputCpu before a final copy to disk if specified by the user. This example follows the flow chart shown in 3.7 Output GPU Buffers – Multiple Outputs. In the display example, both Output GPU buffers are copied to user allocated and locked buffers on the CPU before a final copy to disk if specified by the user. The flow chart for this example is an amalgam of 3.6 User CPU Buffer Output and 3.7 Output GPU Buffers – Multiple Outputs.

B.3 Default Parameters

In order to get started, it is recommended to choose the 'default parameters' option when prompted in each case.

The default for each parameter is as shown in Table B.1 Default Example Parameters:

Pixel Readout Rate:	280MHz (with a sCMOS camera this equates to 560MHz with the 2 outputs)
Exposure Time:	0.01s
Number of Buffers:	8
CUDA Streaming:	False
Width:	2560
Height:	2160
Sensor Height:	2160 (sCMOS 5.5MP)
Save Images to Disk:	false
acquisition Trigger:	Software

Table B.1 Default Example Parameters

A number of options can only be changed by manually updating the relevant 'SDK3Setup.h' header file (for either the Simple or Advanced examples) as shown in Figure B.2 Screen-shot of 'SDK3Setup.h'. The options in this case are set via C++ #define directives.

The 1st value is for the SDK3 'CycleMode'. This configures whether the camera will acquire a fixed length sequence or a continuous sequence. In Fixed mode the camera will acquire 'FrameCount' number of images and then stop automatically. In Continuous mode the camera will continue to acquire images indefinitely until the 'AcquisitionStop' command is issued. Here, we define this mode as 'CONTINUOUS'.

The 2nd value is the default value for the Sensor Height. It is defined as '2160', the height of a sCMOS 5.5MP camera. This can be updated here by simply changing this value to e.g. 2048 for a sCMOS 4.2MP camera.

Finally, the input and output Pixel Encoding values are defined as INPUT_MONO_16 and OUTPUT_MONO_16 in each case. This configures the format of the data stream. See Section 4.4 of 'Andor Software Development Kit.pdf' for further details. The possible options for the input are INPUT_MONO_32, INPUT_MONO_16, INPUT_MONO_12 and

INPUT_MONO_12PACKED. The possible output options are OUTPUT_MONO_32 and OUTPUT_MONO_16. Note that unpacking must be utilised when the input and output Pixel Encodings do not match. See APPENDIX C for further details.

```
/**************************************************************************
*
* SDK3Setup.h
*
* Copyright (c) 2014 Andor Technology.
* All Rights Reserved.
*
* The following variables and functions provide functionality
* for calling into the Andor SDK3 library, and setting user options
* for an acquisition.
*
**************************************************************************/

#ifndef SDK3SETUP_H
#define SDK3SETUP_H

//--- Include SDK3 library
#include "atcore.h"

// Standard libraby includes.
#include <iostream>
#include <sstream>
#include <vector>

using namespace std;

//--- Define Cyle Mode
#define CONTINUOUS

//--- Define Sensor Height
#define SENSOR_HEIGHT 2160

//--- define Pixel Encoding Methods (for input and output)
#define INPUT_MONO_16
#define OUTPUT_MONO_16
```

Figure B.3 Screen-shot of 'SDK3Setup.h'

B.4 Save to Disk Options

When the 'Saving To Disk' option is chosen, note that data is saved in raw binary format. The number of output files is equal to the number of buffers chosen by the user (or alternatively the default number of buffers - 16). We recommend using ImageJ or a similar tool to open the saved data files for inspection and subsequent saving to a format of your choice. See Figure B.4 imageJ Import Raw Binary Data Screen-shots. A final option is output to the user when this option is chosen, 'Save All Images?' When chosen, the example applications append the processed acquisitions from each buffer index to the same file, so that the 1st output file contains all the acquisitions that were processed with the 1st buffer, the 2nd output file contains all the acquisitions that were processed with the 2nd buffer etc. Otherwise, the outputs from each buffer are over-written, so that each output file contains a single processed acquisition (the last to be processed with each buffer). The outputs are saved to the current user's '**My Documents**' directory, within a folder created by the installation named '**GPU Express Output**'. Note also that the 'Save to Disk' functionality is not optimised, and is provided to inspect the acquired data only.

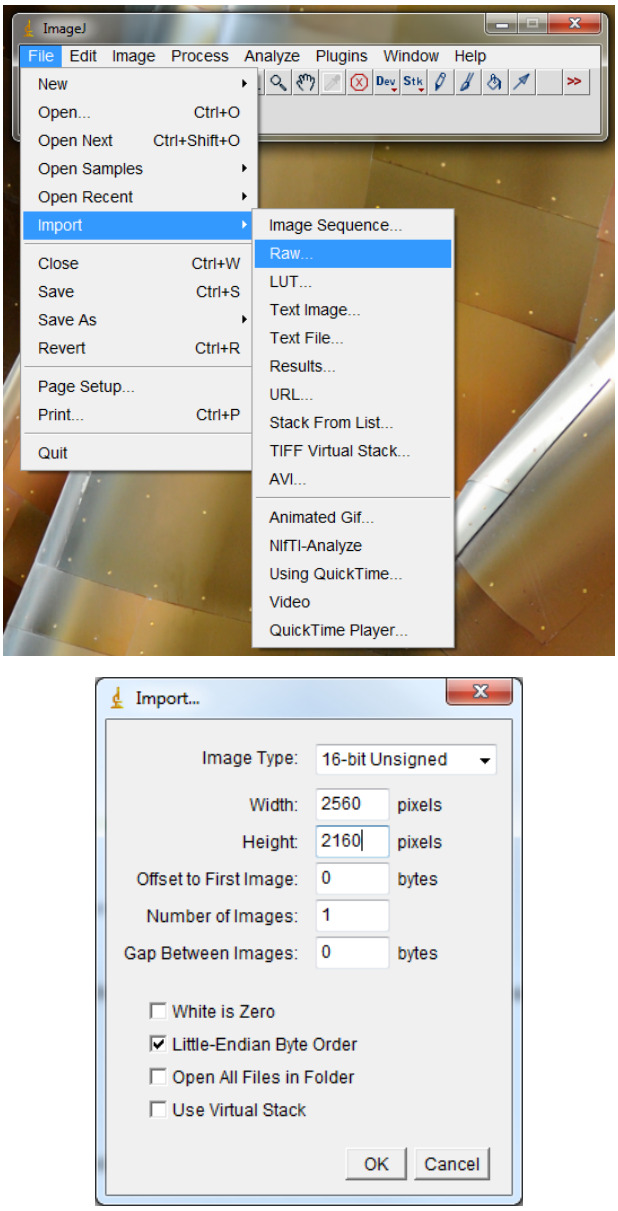


Figure B.4 imageJ Import Raw Binary Data Screen-shots

APPENDIX C CUDA UTILITY LIBRARY

The CUDA Utility library (CUDA_atutility.lib) is included as part of the installation with the GPU Express library. As mentioned in APPENDIX B this provides unpacking functionality to configure the format of the data streams for output from the Andor SDK3. More information is available in Section 4.4 of 'Andor Software Development Kit.pdf'.

A single library call is required to provide fast conversion between Pixel Encoding types and to provide unpacking of data due to granularity restrictions on the row width as is the case with Camera Link frame grabbers (in this case extra padding bytes are added to the end of each row to ensure that granularity limitations are met).

C.1 AT_CudaConvertBuffer

```
int AT_CudaConvertBuffer(AT_U8* inputBuffer, AT_64 inputBufferSize,
AT_U8* auxBuffer, AT_64 auxBufferSize, AT_64 width, AT_64 height, AT_64 stride, const AT_WC *
inputPixelEncoding, const AT_WC * outputPixelEncoding, cudaStream_t * stream);
```

applies all processing required to the data buffer passed into the 1st parameter. This buffer is updated and returned to the caller. The buffer should be large enough to cover the possibility of the output dataset requiring a greater number of bytes to the input (e.g. in the case of a 16-bit input and 32-bit output if the input Pixel Encoding is MONO_16 and the output Pixel Encoding is Mono_32). The size of the input buffer in bytes is input as the 2nd parameter. The 3rd parameter is a required auxiliary buffer which should be (at least) of equal size to the 1st buffer. The size of the auxiliary buffer in bytes is input as the 4th parameter. This may be larger than the input buffer, if for example, using a single large buffer to be shared amongst all required GPU Express paths, see the 'AdvancedSDK3AcqExamples' projects for an example of this approach. Parameters 5 and 6 are simply the width and height of the frames being acquired, and parameter 7 is the stride. This is the number of bytes per line for the image stored in the input buffers. It can be determined by using the SDK3 integer feature "AOI Stride". Parameters 8 and 9 are the input and output Pixel Encodings, and the 10th parameter is the CUDA stream within which to process the function.

APPENDIX D CONVERSION BETWEEN char* AND AT_GPU_WC*

The following code shows an example of how to convert a char* null terminated string to the equivalent wchar_t*.

```
#include <stdlib.h>
char szStr[512];
AT_GPU_WC wczStr[512];
mbstowcs(wcszStr, szStr, 512);
```

and from wide character string to char*

```
#include “stdlib.h”
char szStr[512];
AT_GPU_WC wczStr[512];
wcstombs(szStr, wcszStr, 512);
```