Overview

Most designs require more than a data IO interface to transfer data back and forth—they need an interface that can transfer data as fast as possible, often while performing other resource intensive tasks such as data processing. Reaching high-throughput rates with QuickUSB in these scenarios requires an understating of how QuickUSB fits into a design and how to properly utilize QuickUSB at both the hardware and software level. This application notes helps to clarify the important design choices one needs to make when creating complex designs that demand high data rates using QuickUSB.

Throughput vs. Bandwidth

Before we delve into the details, it is important to have a good and clear understanding of throughput. Throughput is the average rate of successful data transmission, expressed in this document as megabytes/second (MB/s). With respect to QuickUSB, throughput denotes the rate at which data flows from/to a user-application, through the QuickUSB Driver, over USB, and over the QuickUSB general-purpose interface (GPIF) to/from interface hardware, accounting for all system latencies and protocol overhead. Bandwidth, on the other hand, is a measure of the physical rate of data across a communication interface, often expressed in megabits/seconds (Mb/s). USB 2.0 has a bandwidth of 480 Mb/s (60 MB/s). USB throughputs, however, are far less than the USB bandwidth because throughput accounts for the overhead of the USB protocol, as well as system latencies. Throughput is useful for determining the overall data rate of an entire system, and possibly the overall performance of a design.

Test Measurements

The throughput measurements used in all of the figures in this document were measured on three separate, but comparable, computers. Each computer was natively running one of the three operating systems supported by QuickUSB. Here are the relevant specifications of each test computer:

- Windows PC: Windows 7 x64 Ultimate SP1, Intel Core 2 Duo CPU T7500 @ 2.2 GHz, 4 GB RAM
- Linux PC: Ubuntu 10.10, Kernel v2.6.35-30 SMP x64, Intel Core 2 Duo CPU E8500 @ 3.16 GHz, 4 GB RAM
- Mac Book: Mac OS X 10.6.6, Intel Core 2 Duo P8600 @ 2.4 GHz, 4 GB RAM
  The specifications of a computer can have a large effect on throughput measurements. It is important to use the throughput data in this document only as a guide to estimate maximum expected data rates when implementing QuickUSB designs.

Hardware Considerations for High Data Throughput

Designing a Host-Driven Data Flow Architecture

USB is a host-driven protocol and it is important to seriously consider this when designing a project based around USB. "Host-Driven" means that requests to transfer data are initiated by the host and not by interface hardware. QuickUSB is a USB peripheral and requires a Windows, Linux, or Mac computer to act as the host. This important restriction means that your interface hardware cannot signal QuickUSB to send/receive data to/from the computer. Instead, a user application must make QuickUSB API function calls to initiate data transactions over USB. Those transactions are processed by the QuickUSB Library and Driver, sent over USB to the QuickUSB device, and sent over the GPIF (as specified by the IO Model firmware in QuickUSB device). Designs that understand and work with this

requirement will have increased throughput over those that work against it. The following sections will elaborate on designs that work with and against this design philosophy. Keep in mind that determining which design best suits your application is heavily dependent on the details of your application.

Host-Driven Design

This is the preferred design architecture for a project using QuickUSB. Data requests are initiated by software running on the host computer via the QuickUSB API. When those requests reach the QuickUSB interface, they are transferred using the selected IO Model. For example, the Simple IO Model will transmit data over the QuickUSB interface without regard to the readiness of the target hardware, which is good for designs that can send/receive data as fast as QuickUSB can delivery it. Designs that lend themselves well to this design architecture are those that can respond to data requests from the host, instead of trying to deliver unsolicited data to the host.

Polling Design

With some designs, it may be impossible to design a data flow architecture where the host is able to issue data requests without first checking, or polling, the current state of the target interface hardware. These designs will see lower throughput than their host-driven counterparts will, but if properly implemented the effect of polling can be minimized yielding a high-performing design. The key is to avoid traditional polling and instead rely on the timeout capability of QuickUSB. Instead of polling the state of a GPIO pin, reading device EP flags directly, etc. simply issue data requests as you would in a host-driven design: expecting them to be able to be serviced by your target hardware. If your design is using an IO Model that only transmits data over the QuickUSB interface if the target hardware is ready, such as FIFOHS, BlockHS, FullHS, etc., the QuickUSB firmware will attempt to fulfill the data request as dictated by the IO Model waveforms found in the QuickUSB User Guide. If it is unable to fulfill the request before a configurable timeout period has elapsed, then the request is marked as failed and a timeout error code will propagate back to the QuickUSB API. This allows your software to determine if requests have successfully made it to your target interface hardware without the need to poll the state of the target hardware by checking if the request has either completed or timed out and failed. Such a design works great with the FIFOHS model by eliminating the need to poll the state of the FIFO Empty and FIFO Full flags before issuing data requests. You must however, make sure your design can handle partially completed requests because, for example, a data write request for 64 KB could transfer only 32 KB before timing out and failing, perhaps because a buffer fills up or is empty. After the request times out it is reported as failed by the QuickUSB API but half of the failed requests data has been successfully transferred over the GPIF.

Still, some designs may require polling and the timeout mechanism may not provide the needed hardware state information. A design that sends non-constant amounts of data to the PC, for example, is such a case because the user application must issue data requests with the exact amount of data to read. The amount of data to read, however, is not known until the application first queries the device for the number of bytes to read—perhaps the amount of data in a buffer—before issuing the read request. Such a design can be implemented with QuickUSB, but because you must first always query the device before performing data transactions, the design will suffer lower overall data throughput.

Selecting an Appropriate IO Model

The IO Model in a design can have a lot to do with the maximum throughput a design may achieve. This is because throughput is fundamentally limited by the rate at which data can physically travel

across the QuickUSB GPIF. The Simple IO Model and Pipe1 IO Model are the fastest IO Models because data is transferred over the GPIF without regard for the readiness of target hardware, and thus are not limited by the GPIF. The Synchronous/Asynchronous Slave FIFO IO Models are amongst the slowest because they solely depend on target hardware.

In General, IO Models that must check the readiness of target hardware before performing reads/writes do not perform as well as those that do not. Additionally, IO Models that must wait for interface hardware to be ready, as well as the Synchronous/Asynchronous Slave FIFO IO Models, are greatly limited by your interface hardware's ability to transfer data as QuickUSB demands it. Hardware interfaces that cannot keep up with QuickUSB negatively influence data throughput. It is very important to select an IO Model that will perform best for your application. Please consult additional Application Notes on how to properly select the correct IO Model for a given application.

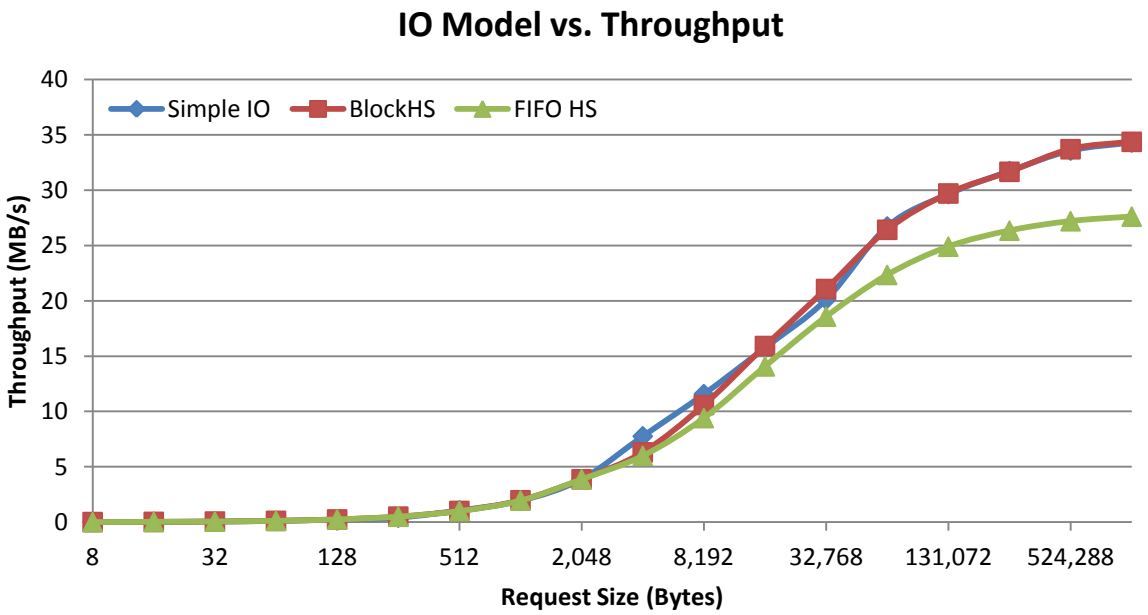

Figure 1: IO Model vs. throughput on windows performing synchronous data reads. Both the BlockHS and FIFOHS IO Model tests were configure to always have the FIFO nEMPTY and nFULL flags pulled high as to not stall data transfers.

Buffering Data

Data transfers over USB occur in bursts. Each burst contains a data packet of at most 512 bytes (High-Speed) or 64 bytes (Full-Speed) of data. This size is known as the USB Packet Size. Data requests for larger sizes are broken down into multiples of the USB Packet Size. Because QuickUSB implements data transfers using USB Bulk Endpoints, the latency between USB Data Packets can, and will, vary. These latencies, along with software latencies in issuing successive data requests, are visible on the GPIF interface. If your design is sensitive to the time between successive USB Data Packets then you may consider inserting a data buffer, such as a FIFO, between your hardware and QuickUSB (for FPGA designs, this FIFO could exist inside the FPGA). This form of buffering helps shield these visible latencies from target hardware. This has the added benefit that it adds a physical layer between QuickUSB and your target hardware that can potentially increase throughput by allowing for higher request sizes with interface hardware that would otherwise not allow larger request sizes.

Software Considerations for High Data Throughput

Variations Between Platforms

If you are creating a design that must operate under multiple platforms (Windows, Linux, and/or Mac), then it differences in throughput across platforms can have a significant impact on a design. Additionally, hardware variations between systems can greatly influence the performance of a design. It is very important to ensure that your design will operate correctly on all systems you intend to support by testing across low-performance and high-performance hardware on all supported operating systems.

Reads vs. Writes

Data reads using QuickUSB are typically faster than data writes across all platforms. This is an important difference to note in designs that have minimum throughput requirements for reading and writing data.

The Effects of Request Size

Request size is the most important parameters when it comes to throughput with QuickUSB. Request size is the amount of data request in a single data transaction (i.e. call to the QuickUSB API). Each time you call the QuickUSB API to issue a data request, the request is sent to the QuickUSB Driver where it is broken into one or more USB requests and sent further down the USB Driver Stack until it eventually makes its way over the USB cable. Each step along the way introduces software latencies that have a detrimental effect on maximum performance. To minimize these software latencies it is important to issue data requests in large chunks rather than small pieces.

The following figures show the effects that request size can have on throughput using the QuickUSB Synchronous API. The tests were run on Windows, Linux, and Mac to show variations between platforms, which can be quite substantial. No data processing was performed on the transferred data.

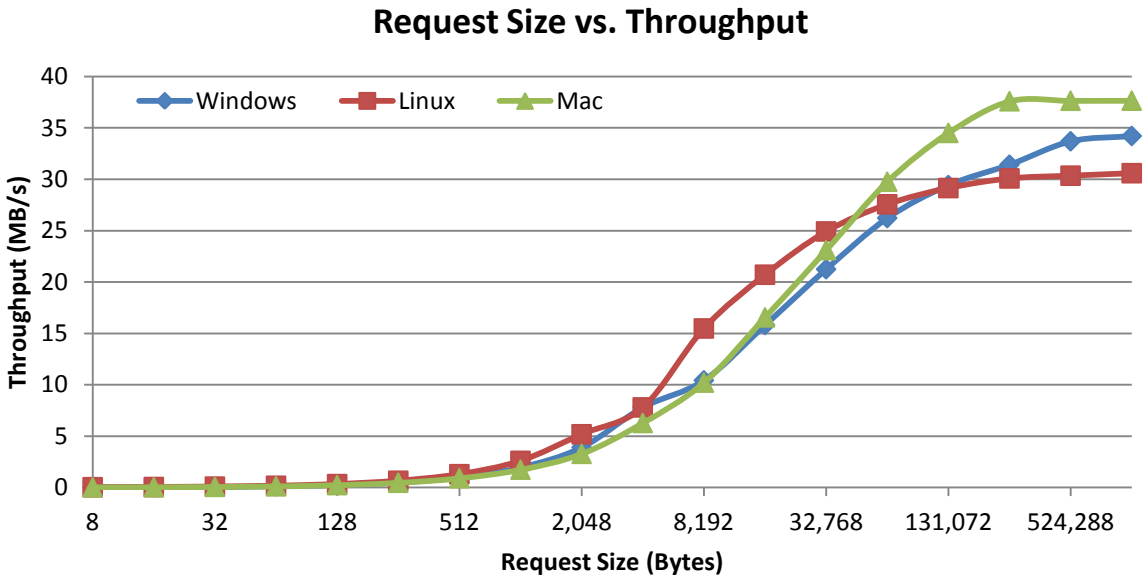

Figure 2: Request size vs. throughput using the Simple IO Model and performing synchronous data reads.

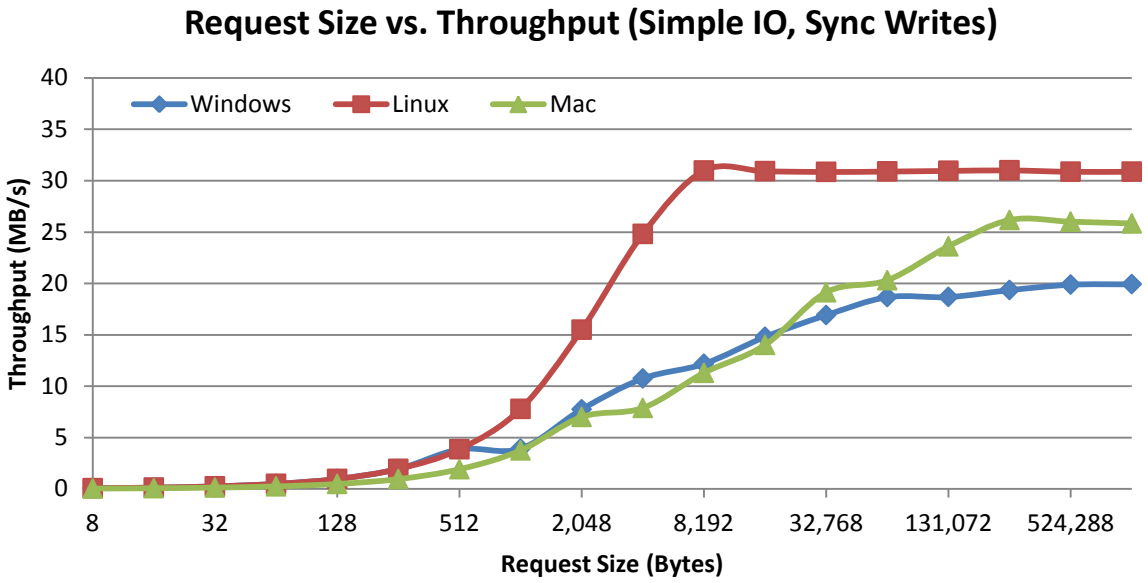

Figure 3: Request size vs. throughput using the Simple IO Model and performing synchronous data writes.

Synchronous vs. Asynchronous Requests

Increasing request size can greatly increase throughput by minimizing software latencies, but it does not eliminate software latencies. Additionally, the USB stack is designed is such a way that USB requests are queued up and sent over USB as quickly as possible by the low-level USB host driver. Not only does issuing requests synchronously not allow the USB host driver to queue requests and minimize delays between USB packets, latencies still exist in getting data requests from the QuickUSB Library to the USB host driver. These latencies add up and can have a significant impact on throughput.

The QuickUSB API provides two mechanisms to remove these latencies: the Asynchronous API and the Streaming Data API. Both APIs allow you to issue multiple asynchronous requests at without waiting for those requests to complete. This substantially reduces system latencies and has the added benefit that you can perform background tasks while USB requests are processed—a very important feature in GUI applications and designs that require real-time data processing.

The Asynchronous API

The Asynchronous API gives you complete control over issuing asynchronous data requests. You may issue any number of asynchronous requests (up to the system limit) and then perform other processing tasks as those requests are processed. Once you are ready to check if a given asynchronous request has completed you just issue a QuickUSB API function call.

The Asynchronous API provides two mechanisms for determining when an issued request has completed (either successfully or unsuccessfully): 1) you may query the request to see if it has completed yet with a call to the QuickUSB API (optionally blocking or non-blocking); and/or 2) you may provide a callback function to be executed when the request data completed. The callback routine, or completion routine, is called in the context of the main application thread unless multithreading, in which case the completion routines are call in the context of worker threads created by the QuickUSB API. Note that if you opt for a single-threaded asynchronous design your application must first wait for the request to complete (by making a call to the QuickUSB API) before the completion routine for that

request will be executed. This is not required for multithreaded designs as worker threads can asynchronously process requests and call completion routines without any interaction from the main thread.

For applications that must perform real-time processing on data, performing the data processing in a completion routine and allowing the QuickUSB API to automatically multithread the processing can improve throughput when system processing is bottlenecking throughput. See the Maintaining High Data Throughput with Real-Time Data Processing section of this document for more information.

The Streaming API

The Asynchronous API is great for performing low-latency bursts of asynchronous data requests, but if you need to continually read or write to a QuickUSB device for a prolonged period of time then the Streaming API can be very beneficial. The Streaming Data API is built on top of the Asynchronous Data API, but the complexity of waiting for requests to complete and then re-issuing those requests is moved into the QuickUSB API. Additionally, latencies in manually having to wait and re-issue completed requests are slightly reduced. With the Streaming Data API, you must provide a callback function to be executed when each request completes. Note that if you opt for a single-threaded streaming design your application must periodically call the QuickUSB API function QuickUsbProcessStream so that requests on the stream may be processed and completion routines may be executed. This is not required for multithreaded designs as worker threads can asynchronously process requests and call completion routines without any interaction from the main thread.

For applications that must perform real-time processing on data, performing the data processing in a completion routine and allowing the QuickUSB API to automatically multithread the processing can improve throughput when system processing is bottlenecking throughput. See the Maintaining High Data Throughput with Real-Time Data Processing section of this document for more information.

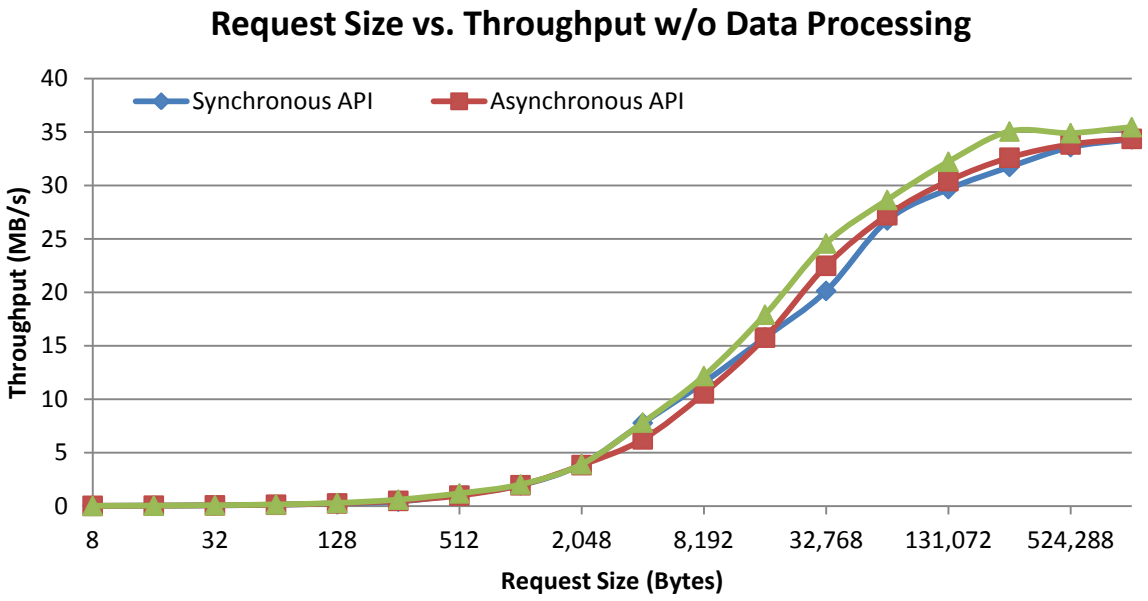

Figure 4: Request size vs. throughput using the Simple IO Model and performing synchronous/asynchronous data reads on Windows without any data processing.

Maintaining High Data Throughput with Real-Time Data Processing

Challenges of Real-Time Data Processing

Processing data in real-time adds an additional timing constraint on the design. Now, instead of just, for example, reading data from a disk and transferring it over USB or receiving data and writing it to disk, you must perform some processing on the data which takes some finite amount of time. The goal is to achieve a high data rate without hindering performance by under-utilizing the system while performing both data transfer and data processing. The Asynchronous and Streaming Data APIs lend themselves well to data processing through use of completion routines. When a data request completes, a completion routine is (optionally) called to notify your application that either:

1) The read request has completed and the data buffer associated with the request contains the read data, or
2) The previous write request has completed and you must fill the data buffer associated with the request up with new data to write.

If you perform data processing from within the completion routine then the only factor you must focus on is ensuring that your data processing algorithm does not take so long to process that the data requests cannot successively issued quickly enough to keep up with the desired throughput rate. In such a case you may potentially increase performance by increasing the number of data requests issued at once (i.e. the number of buffers) and/or multithreading the design, as long as you are not attempting to over-utilize the system.

The following figures show how data rates can vary between the Synchronous, Asynchronous, and Streaming Data APIs once you start processing data in real-time as a specific processing rate, measured here in megabytes/millisecond (MB/MS). It is easy to see that the Synchronous Data API takes the largest hit because each request must be issued, completed, and processed before the next request may be issued.

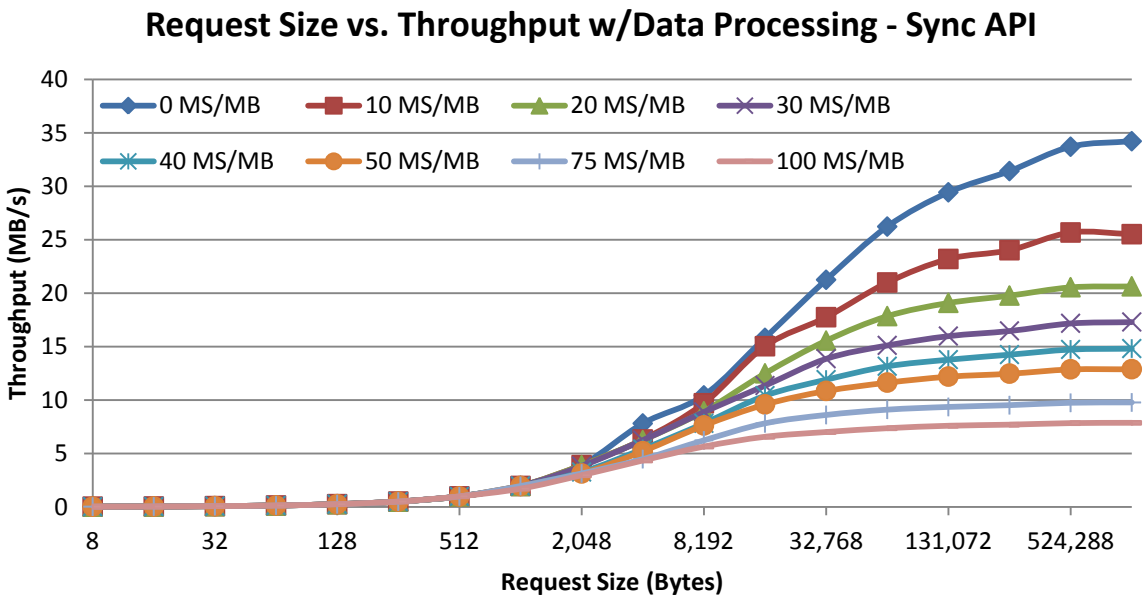

Figure 5: Request size vs. throughput using the Simple IO Model and performing synchronous data reads on Windows with data processing. The data processing rates, measured in milliseconds/megabyte, express the amount of processing time spend on data.

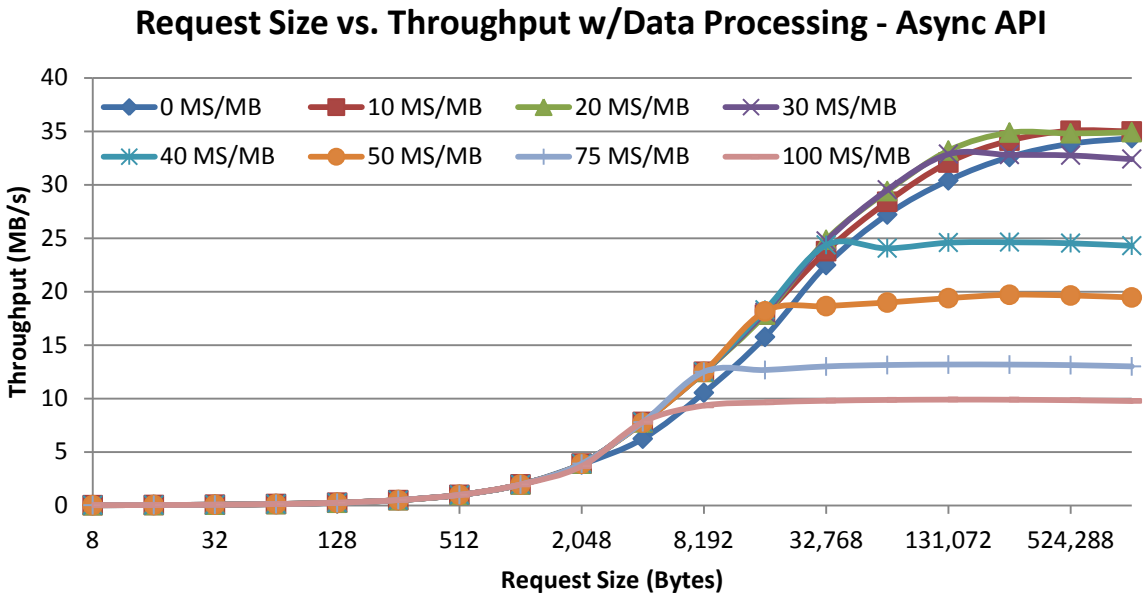

Figure 6: Request size vs. throughput using the Simple IO Model and performing asynchronous data reads on Windows with data processing. The test is setup to run with 4 asynchronous transactions continually being issued. The data processing rates, measured in milliseconds/megabyte, express the amount of processing time spend on data.

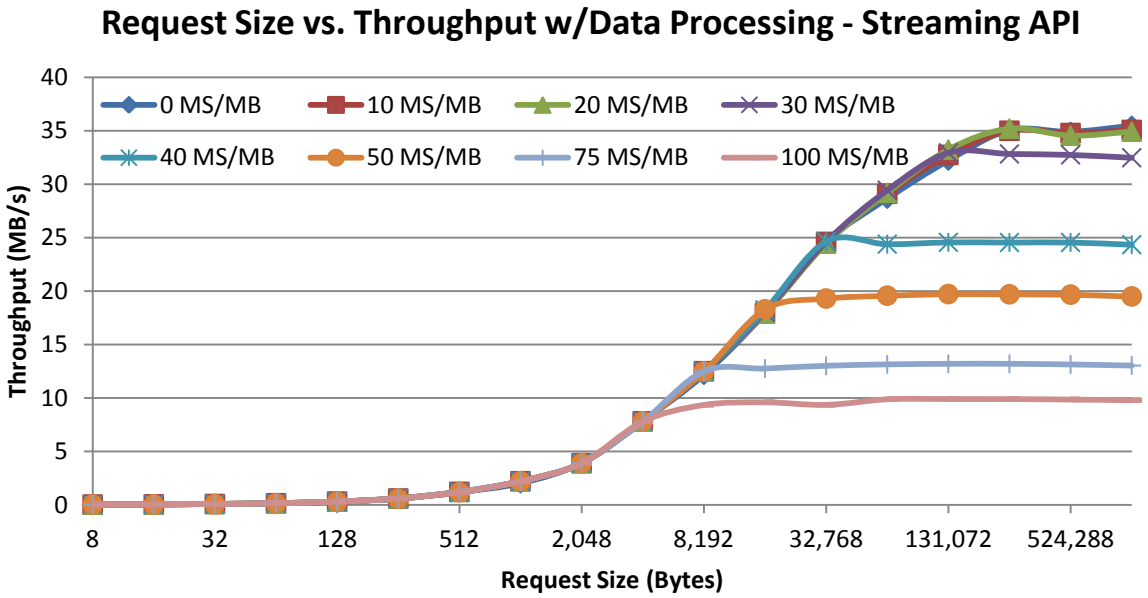

Figure 7: Request size vs. throughput using the Simple IO Model and performing streaming data reads on Windows with data processing. The read stream is configured to have 4 buffers and no multithreading. The data processing rates, measured in milliseconds/megabyte, express the amount of processing time spend on data.

Benefits of Multithreading

If you find that your processing algorithm is taking up too much time to execute and is lowering your overall system throughput, then multithreading your design may help increase your system throughput. Both the Asynchronous Data API and Streaming Data API both can both be automatically multithreaded by the QuickUSB Library with minimal increase in design complexity. The Asynchronous Data API

may be multithreaded by calling an additional QuickUSB API function to set the number of threads to use and to set the thread concurrency. The Streaming Data API may be multithreaded by setting the number of threads to use and the thread concurrency parameters to non-zero values. Setting the number of threads to a value greater than zero instructs the QuickUSB Library to create internal worker threads to handle issuing requests, processing requests, and calling completion routines. This means that completion routines are called from the context of worker threads when multithreading. The thread concurrency specifies the number of threads allowed to concurrently execute a completion routine, and must have a value of at least one when multithreading. If you set the thread concurrency to a value greater than one, you must take the appropriate precautions to protect shared data within your completion routine, as well as handle the case where completion routines are executed in parallel, thus potentially out of order from their respective data requests.

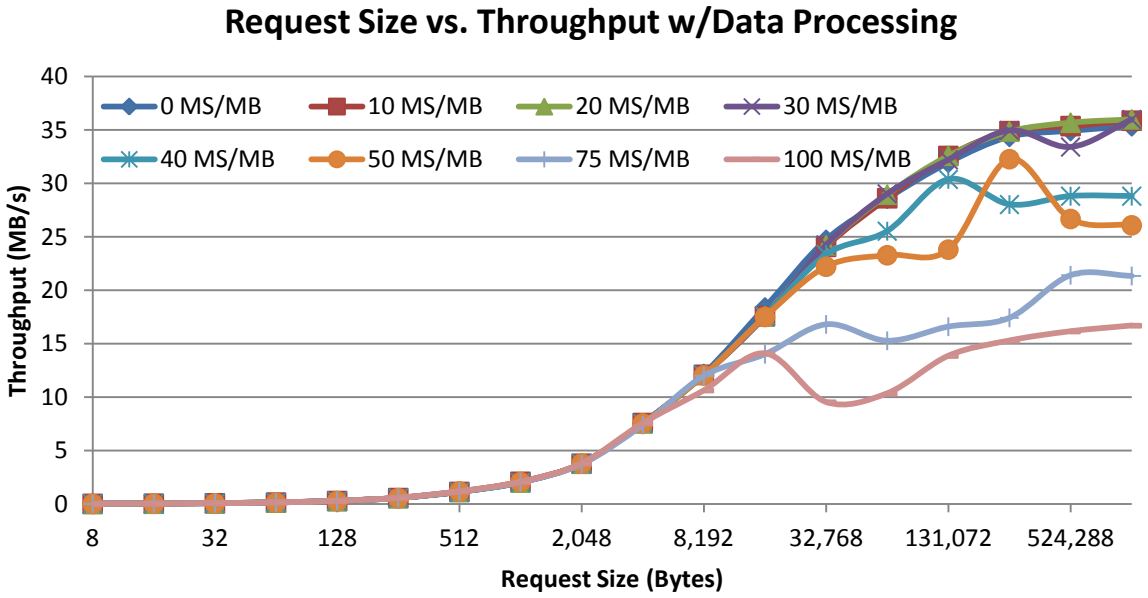

Figure 8: Request size vs. throughput using the Simple IO Model and performing streaming data reads on Windows with data processing. The read data stream is configure to run with 4 worker threads and a thread concurrency of 2. The data processing rates, measured in milliseconds/megabyte, express the amount of processing time spend on data. The variations in throughput are a result of real-time operating system scheduling and system usage. The throughput rates, in general, should be equal to or greater than their asynchronous implementation.

Multithreading the Asynchronous and Streaming APIs allows the operating system to better serve processing intensive processes, especially on multi-core and multiprocessor systems, by allowing multiple threads to process and reissue requests at the same time. The QuickUSB Library automatically ensures that data requests are processed and reissued in order to maintain data integrity. There is added complexity if you use a thread concurrency greater than one because then the QuickUSB Library will allow multiple threads to execute completion routines at the same time, which introduces the possibility the shared data may be accessed at the same time and that the completion routines execute out of order from their associated data requests. In such a case you must use synchronization objects to protect shared data and handle out-of-order execution, thus adding some complexity to the design. To reduce the initial complexity of a design, it is a good idea to first write your software to be single-threaded. Then, once you determine the throughput of the system is limited by the completion routines (potentially performing data processing), switch to a multithreaded design but keep the thread concurrency to one. If the throughput of the system is still limited by the completion routines, increase the thread concurrency

but keep in mind that a thread concurrency greater than one requires data protection through the use of synchronization objects within the completion routines and, if improperly implemented, can decrease performance rather than increase performance.

Bitwise Systems
6489 Calle Real, Suite E
Goleta, CA 93117
V (805) 683-6469
F (805) 683-4833
info@bitwisesys.com