

# Direct GPU/FPGA Communication Via PCI Express

Ray Bittner, Erik Ruf

Microsoft Research

Redmond, USA

{raybit,erikruf}@microsoft.com

**Abstract**—Parallel processing has hit mainstream computing in the form of CPUs, GPUs and FPGAs. While explorations proceed with all three platforms individually and with the CPU-GPU pair, little exploration has been performed with the synergy of GPU-FPGA. This is due in part to the cumbersome nature of communication between the two. This paper presents a mechanism for direct GPU-FPGA communication and characterizes its performance in a full hardware implementation.

**Keywords**- GPU; FPGA; PCI Express; PCIe; Xilinx; CUDA; nVidia; Windows; Verilog

## I. INTRODUCTION

The world of computing is experiencing an upheaval. The end of clock scaling has forced developers and users alike to begin to fully explore parallel computation in the mainstream. Multi-core CPUs, GPUs and to a lesser extent, FPGAs, are being employed to fill the computational gap left between clock rate and predicted performance increases.

The members of this parallel trifecta are not created equally. Typical attributes are:

- CPUs - Ease of programming and native floating point support with complex and cumbersome memory systems, as well as significant operating system overhead.
- GPUs - Fine grain SIMD processing and native floating point with a streaming memory architecture and a more difficult programming environment.
- FPGAs - Ultimate flexibility in processing, control and interfacing, at the extreme end of programming difficulty and lower clock rates with only cumbersome floating point support.

Each has its strengths and weaknesses, thus motivating the use of multiple device types in heterogeneous computing systems. A key requirement of such systems is the ability to transfer data between components at high bandwidth and low latency. Several GPGPU abstractions [3][5][6] support explicit transfers between the CPU and GPU, and it has recently been shown that this is also possible between CPU and FPGA [1]. However, we know of no existing solution that enables direct GPU-FPGA communication.

Existing facilities may be used to implement GPU to FPGA communication by transferring data through CPU memory as illustrated by the red line in Fig 1. Data must

traverse through the PCI Express (PCIe) switch twice and suffer the latency penalties of both the operating system and the CPU memory hardware using the red indirect path. We refer to this as a GPU-CPU-FPGA transfer. This additional indirection adds communication latency and operating system overhead to the computation, as well as consuming bandwidth that could otherwise be used by other cluster elements sharing the same communication network.

In this paper, we describe and evaluate a mechanism for implementing the green line in Fig. 1 for direct, bidirectional GPU-FPGA communication over the PCIe bus [2]. As illustrated, data moves through the PCIe switch once and is never copied into system memory, thus enabling more efficient communication between these disparate computing elements. We refer to this as a direct GPU-FPGA transfer.

## II. THE SPEEDY PCIe CORE

Enabling FPGA based PCIe communication is not a simple task and, while there are numerous PCIe cores available, these often fall short of a complete implementation or are prohibitively expensive. Fortunately, the Speedy PCIe core has delivered a viable solution at no cost that can be

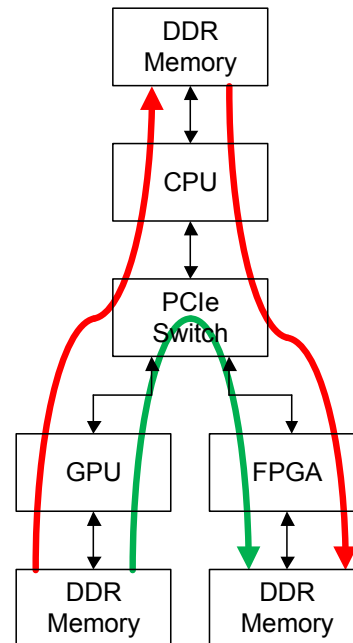


Figure 1. Two Conceptual Models Of GPU To FPGA Transfers

made to serve for the problem at hand.

The Speedy PCIe core is a soon to be published, freely downloadable, FPGA core designed for Xilinx FPGAs [1]. It builds on Xilinx PCIe IP [11] to provide the FPGA designer a memory-like interface to the PCIe bus that abstracts away the addressing, transfer size and packetization rules of PCIe. The standard distribution includes Verilog that turns this memory interface into a high speed DMA engine that, together with the supplied Microsoft Windows driver, delivers the full bandwidth potential of the PCIe bus between a PC's system memory and DDR3 that is local to the FPGA.

The Speedy PCIe design emphasizes minimal system impact while delivering maximum performance. Data transfers may be initiated from the CPU via a single write across the PCIe bus after the setup of a number of transfer descriptor records that are maintained in the host's system memory. Since system memory has much lower latency and higher bandwidth for the CPU, this arrangement offloads work from the processor and ultimately results in higher performance by avoiding operating system overhead. Minimizing the number of CPU initiated reads and writes across the PCIe bus is also helpful because in practice the execution time for a single 4 byte write is often in the range of 250 ns to 1  $\mu$ s, while reads are often in the range of 1  $\mu$ s to 2.5  $\mu$ s. This overhead savings offered by the Speedy PCIe core directly contributes to lower latency transfers as will be shown later.

### III. ENABLING THE MISSING LINK

On the GPU side, we have somewhat less control, as all hardware functionality remains hidden behind an opaque, vendor-supplied driver. Typically such APIs support only transfers between GPU and CPU memories, not between GPU memory and that of arbitrary devices. One notable exception is the peer-to-peer memory transfer facility provided by the nVidia CUDA library on the professional-level Quadro and Tesla product lines. This enables GPU-GPU data transfers, but does not explicitly support transactions involving arbitrary PCIe devices, such as our FPGA development board. At present, the GPU must always be the bus master in any transfer in which it is involved.

If the GPU must always be the bus master, it follows that the FPGA must always be the slave order to enable the direct GPU-FPGA data path. This requires the FPGA to map its memory (on chip or otherwise) onto the PCIe bus so that the GPU may read or write it directly as needed. This functionality is already enabled in the user example supplied with the Speedy PCIe design, which demonstrates how to map DDR3 physical memory addresses onto the PCIe bus. This arrangement forces the master/slave relationships summarized in Table I for each transfer type.

Our primary discovery is that some of the CUDA operations intended for CPU memory access can be repurposed for GPU-FPGA transfers. In particular, the CUDA API supports the concept of page-locked CPU memory, which maintains a constant physical address and can thus be efficiently accessed by the GPU's bus-mastering DMA controller. CUDA provides `mmap()`-like functionality

for allocating and freeing blocks of such memory. Crucially, recent versions of CUDA also provide a routine for page-locking existing CPU virtual address ranges (succeeding only when the operating system has allocated contiguous physical pages for the specified virtual address range). We have found that this routine does not distinguish between virtual addresses mapped to physical CPU memory and those mapped to FPGA memory by the SpeedyPCIe driver. Furthermore, since the driver maps FPGA pages in locked mode, the CUDA locking routine does not fail on these ranges. Thus, the mapped pointer can be passed to various `mempcpy()`-style operators in CUDA that require page-locked CPU memory pointers as arguments.

Using this to our advantage, we modified the Speedy PCIe driver to allow a user application to obtain a virtual pointer to the physical DDR3 mapped by the FPGA onto the PCIe bus. Using this pointer, it is possible to directly access the FPGA's DDR3 memory using the standard C `*ptr` notation or other programmatic forms of direct manipulation. It is also possible to pass this virtual memory pointer to the CUDA page-locking and memory copy routines, causing the GPU to directly write or read data to/from the FPGA's DDR3 memory.

The CPU is involved in the transfer, but only for initial setup, such as virtual to physical address mapping or GPU DMA configuration in the driver, but the data itself is moved directly between the GPU and FPGA over the PCIe bus.

### IV. TEST PROCEDURE

For our testing, we chose an nVidia GeForce GTX 580, a high-end consumer GPU that supports the CUDA 4.1 API (with the exception of peer-to-peer functionality restricted to the more expensive Quadro and Tesla GPUs). This unit can make use of up to 16 generation 2.0 PCIe lanes, reaching up to 6.2 GByte/Sec of throughput.

The FPGA platform used in our tests was a Xilinx ML605 development board with an integrated V6LX240T-1 Xilinx FPGA. This unit supports x8 lane generation 1.0 PCIe with a maximum throughput of approximately 1.6 GByte/Sec [1] (a factor of four slower than the GPU). Both the graphics and FPGA development boards were plugged into a commercial PC backplane running a modern Intel six core CPU and supporting PCIe generation 2.0 x16.

Transfers between CPU and FPGA memories were implemented by mapping FPGA memory ranges to CPU virtual memory ranges using the Speedy PCIe driver as discussed above. The virtual address ranges are then registered with the operating system as memory-mapped files. Performing standard file system *Read* and *Write* operations on such files induces DMA-based memory

TABLE I. MASTER/SLAVE RELATIONSHIPS

Transfer	PCIe Master	PCIe Slave
GPU-CPU	GPU	CPU
FPGA-CPU	FPGA	CPU
GPU-FPGA	GPU	FPGA

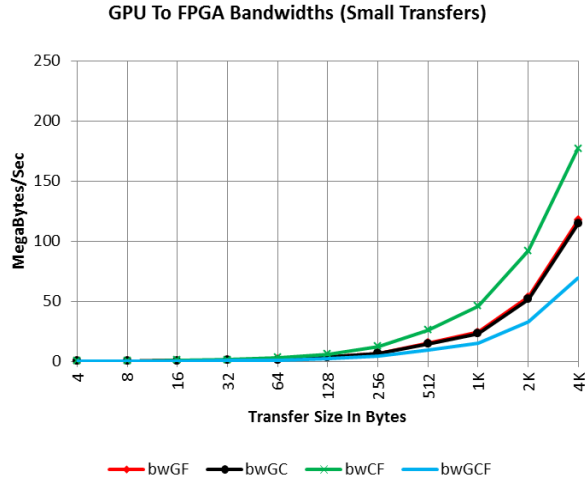


Figure 2. Performance Of Small GPU To FPGA Transfers

transfers between the CPU and FPGA. In this scenario, the FPGA is always acting as a PCIe bus master, sending or requesting data as required.

Transfers between GPU and CPU memories were accomplished via the `cudaMemcpy()` interface. Because we allocate the CPU memory in page-locked mode, the resulting transfers make use of DMA, with the GPU acting as master.

Transfers between GPU and FPGA memories were performed by first passing the FPGA memory range to `cudaHostRegister()`, which makes CUDA treat the FPGA memory as page-locked CPU memory. Memory transfers are performed using `cudaMemcpy()` in exactly the same manner as the GPU-CPU transfers described above, with the GPU acting as the PCIe bus master.

## V. RESULTS

Knowing that we had asymmetric transfer characteristics, we created separate graphs for each direction, and separate graphs for small and large transfers, as shown in Figs. 2-5. Each transfer was performed ten times, with the mean elapsed time used to compute the bandwidth at each transfer size. As we would expect, the graphs show an increase in bandwidth as the transfer size increases until reaching an asymptotic value.

Figs. 2-3 show the achieved bandwidths in the GPU to FPGA direction in four curves:

- Red (bwGF) - The direct data path from GPU to FPGA.
- Black (bwGC) - GPU to CPU bandwidth.
- Green (bwCF) - CPU to FPGA bandwidth.
- Blue (bwGCF) - GPU to CPU to FPGA cumulative bandwidth.

Of these, the red GPU to FPGA and the blue GPU to CPU to FPGA lines are the most interesting, as they compare the benefit of direct GPU to FPGA transfers vs. the trip through system memory, respectively. For the large transfers

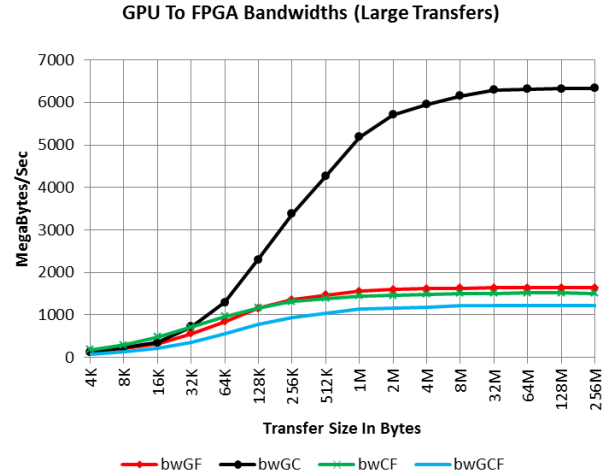


Figure 3. Performance Of Large GPU To FPGA Transfers

shown in Fig. 3, the black line indicating the GPU to CPU bandwidth dominates the graph as the GPU supports generation 2.0 x16 lane PCIe. Since the FPGA (green line) we used only supports generation 1.0 x8 lane PCIe, it is expected to support roughly  $\frac{1}{4}$  of the bandwidth that can be achieved with the GPU. Although as shown in Fig. 2, for small transfers the CPU to FPGA path dominates due to the smaller latencies as measured in Table II.

Figs. 4-5 show the results of the FPGA to GPU transfer direction. The color coding and line symbols have the same meanings as before, except that data is moving in the opposite direction. Note that the FPGA to GPU (red line) bandwidth is markedly lower than in the GPU to FPGA case.

We approximate the transfer latency by the elapsed time of the smallest possible PCIe data transfer of 4 bytes as measured at the level of the CUDA API. These measured latencies are summarized in Table II. The CPU-FPGA latencies and bandwidths seen here agree with those reported in [1], though it is interesting that the GPU-FPGA and GPU-CPU transfers suffer approximately twice the latency of the CPU-FPGA transfers. As a result, the indirect transfers going through CPU memory have an increased total latency of approximately 50% (20  $\mu$ s) over the direct GPU-FPGA transfers in both directions.

## VI. DISCUSSION

The primary limiting factor in this implementation is the supported bandwidth of the FPGA. Since it is only operating in a generation 1.0 x8 lane mode, its performance reaches a maximum of 1.6 GByte/Sec regardless of transfer partner. It should be possible to alleviate this bottleneck by using a faster -2 speed grade part, however, such parts are not normally populated on the ML605 Xilinx development board. The GPU supports generation 2.0 x16 lane operation; giving it an inherent 4x speed advantage over the FPGA with a measured maximum of 6.2 GByte/Sec.

The two graphs show that the two data directions have asymmetric bandwidth characteristics. This is better

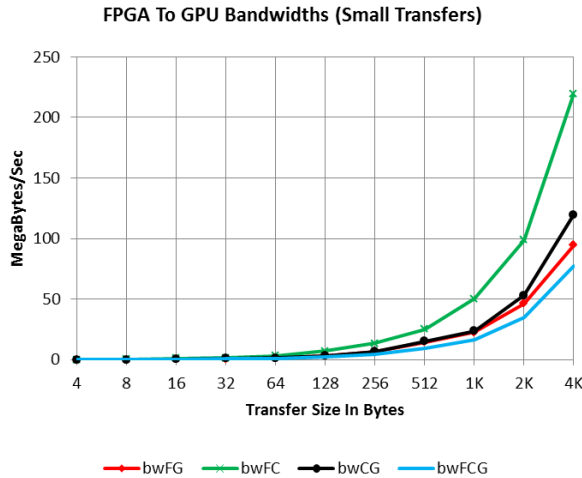


Figure 4. Performance Of Small FPGA To GPU Transfers

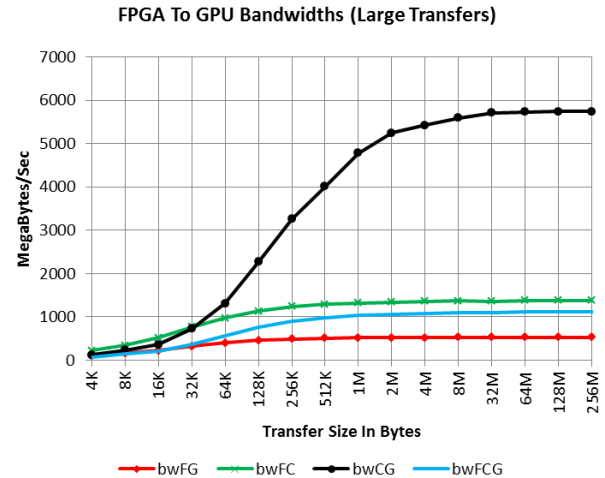


Figure 5. Performance Of Large FPGA To GPU Transfers

visualized in Fig. 6 where the relative speedups comparing direct GPU-FPGA transfers and the indirect GPU-CPU-FPGA paths have been computed. These speedup numbers are computed in the traditional sense where numbers greater than 1 indicate relative improvement of the direct GPU-FPGA path, and numbers less than 1 indicate that the GPU-FPGA path degrades transfer performance. In the GPU to FPGA case, the performance improvement over GPU to CPU to FPGA settles at 34.6% for large transfers. On the other hand, the FPGA to GPU case actually lowers performance by 52.6% as compared to the FPGA to CPU to GPU path.

We believe that this is due to the implementation of the Speedy PCIe user example Verilog code. This code gives the example design its “personality” and determines the exact features that the FPGA implementation will support including memories, peripherals and transfer characteristics. At the time that the user example was written, it was believed that all high bandwidth traffic would be initiated as bus master writes on the PCIe bus since master writes have inherently lower overhead in the PCIe protocol.

However, in the GPU-FPGA situation, the GPU always demands to be the bus master. This is ideal when data is being transferred from the GPU to the FPGA as the GPU initiates master writes with data and the FPGA can digest

these at full speed (1.6 GByte/Sec). When data is being transferred from the FPGA to the GPU, the GPU initiates master read requests over the PCIe bus and the FPGA faithfully sends back the data as requested. However, a bottleneck arises because this slave read data path is not fully optimized in the FPGA, resulting in a disappointing 0.514 GByte/Sec.

It should also be noted that the GPU-CPU transfers themselves also show some degree of asymmetric behavior. In the case of a GPU to CPU transfer, where the GPU is initiating bus master writes, the GPU reaches a maximum of 6.18 GByte/Sec. In the opposite direction from CPU to GPU, the GPU is initiating bus master reads and the resulting bandwidth falls to 5.61 GByte/Sec. In our observations it is typically the case that bus master writes are more efficient than bus master reads for any PCIe implementation due to protocol overhead and the relative complexity of implementation. While a possible solution to this asymmetry would be to handle the CPU to GPU direction by using CPU initiated bus master writes, that hardware facility is not available in the PC architecture in general.

The transfer latencies shown in Table II are interesting in that the path between the CPU and FPGA is shown to have half of the latency of the path between the CPU and GPU in both transfer directions. This is despite the fact that the GPU hardware supports 4x the transfer bandwidth of the FPGA and so the latency is expected to be lower. The longer GPU latency could be caused by more OS overhead, presumably in the time that the driver needs to setup the transfer. This could be due to pure software overhead, or it may be that the GPU hardware requires more I/O reads/writes from the driver in order to setup the transfer, which may be costly as described earlier. It isn’t possible to determine the exact cause without GPU driver source code, GPU hardware specifications or a PCIe bus analyzer; none of which are easily obtainable.

One final observation is that the performance comparisons are deceptive in that they do not account for

TABLE II. FOUR BYTE TRANSFER LATENCIES

Transfer Points and Direction	Latency ( $\mu$ s)
GPU to CPU	41.9
CPU to FPGA	20.7
GPU to CPU to FPGA (Total)	62.6
GPU to FPGA	40
FPGA to CPU	18.9
CPU to GPU	40.4
FPGA to CPU to GPU (Total)	59.3
FPGA to GPU	41.1

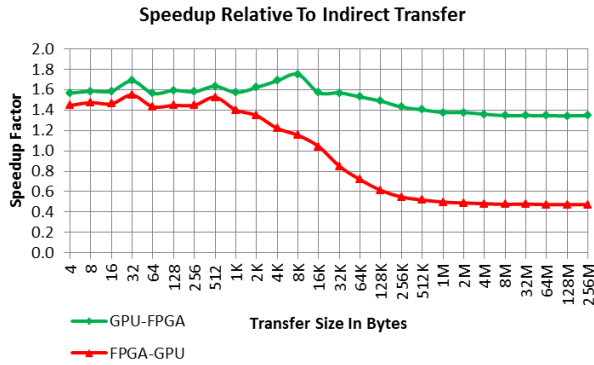


Figure 6. Speedup Of GPU-FPGA Relative To GPU-CPU-FPGA Transfers

potential bottlenecks in the CPU’s system memory or the PCIe switch as illustrated in Fig. 1 since only one GPU-FPGA pair was tested. All of the traffic for an indirect route through the CPU must go through these structures, which will ultimately saturate if a sufficient number of transfers are occurring simultaneously. Since PCIe is constructed with a tree topology and all transfers are point to point through switches, the direct GPU-FPGA path may circumvent the system memory bottleneck by taking the direct route through a local PCIe switch. Thus it is possible that direct GPU-FPGA communication could scale and attain much higher speedups in systems where multiple transfers are happening simultaneously. While this possibility is subject to the hardware details of the particular host platform being used, such independent data paths are possible through the use of suitable switches. One such source of PCIe switches is PLDA [10].

### VII. FUTURE WORK

Our next order of business will be to address the glaring bottleneck in the FPGA to GPU transfer direction. This will require a detailed analysis of the slave read data path within the FPGA likely followed by a number of changes to the Verilog code. With those changes, we hope to see the bandwidth rise to be commensurate with the GPU to FPGA direction.

Concurrently, we are exploring applications that could benefit from the close synergy between GPU and FPGA that this technique enables. The GPU offers relative ease of programming and floating point support while the FPGA offers extreme flexibility, bit manipulation and interfacing possibilities. An ongoing project in our lab [9] is making use of our technology to implement mixed GPU/FPGA strategies for distributed graphics rendering. This application in particular may benefit from reduced usage of system memory bandwidth and we intend to characterize this further.

Other potential investigations include the extension of our approach to non-nVidia GPUs and to GPU-FPGA interactions beyond memory transfers, such as synchronization, that are presently mediated by the CPU.

The former may be at least partially possible via OpenCL, which is supported on some AMD GPU devices. The OpenCL specification [4] hints at initializing device (e.g. GPU) buffers with “host-accessible (e.g. PCIe) memory,” so it is conceivable that our CPU virtual pointer to the FPGA DDR3 memory can be used as a source, if not as a target.

Bypassing the CPU for other interactions will require the involvement of GPU vendors, as the relevant mechanisms on are presently hidden behind black-box driver code.

### VIII. CONCLUSION

We have presented a mechanism and analysis for direct GPU-FPGA communications via PCI Express. Our hope is that this opens the door to new computation synergies and architectures that were previously unsuitable or perhaps not considered practical. There is one notable wart in the system that we hope to remove in the near future. However, the possibility remains for exploration in this brave new parallel world.

### REFERENCES

- [1] Ray Bittner, “Speedy Bus Mastering PCI Express”, 22nd International Conference on Field Programmable Logic and Applications, Aug 2012, in press.
- [2] Alex Goldhammer, John Ayer Jr, “Understanding Performance of PCI Express Systems,” Xilinx WP350, Sept 4, 2008
- [3] Khronos Group, “OpenCL: The open standard for parallel programming of heterogeneous systems”. Available at <http://www.khronos.org/opencl/>.
- [4] Khronos Group, “OpenCL API Registry”. Available at <http://www.khronos.org/registry/cl>.
- [5] Microsoft Corporation, “DirectCompute”. Available at <http://blogs.msdn.com/b/chuckw/archive/2010/07/14/directxcompute.aspx>.
- [6] nVidia Corporation, “nVidia CUDA API Reference Manual, Version 4.1”. Available from <http://www.nvidia.com/CUDA>.
- [7] nVidia Corporation, “nVidia CUDA C Programming Guide, Version 4.1”. Available from <http://www.nvidia.com/CUDA>.
- [8] PCI Express Base Specification, PCI SIG. Available: <http://www.pcisig.com/specifications/pciexpress>.
- [9] Turner Whitted, Jim Kajiya, Erik Ruf, Ray Bittner, “Embedded Function Composition,” Proceedings of the Conference on High Performance Graphics, 2009.
- [10] PLDA Corporation, <http://www.plda.com/prodetail.php?pid=175>.
- [11] PCI Express, Xilinx Corporation. Available: <http://www.xilinx.com/technology/protocols/pciexpress.htm>.