Cluster Comput
DOI 10.1007/s10586-013-0280-9

Direct GPU/FPGA communication Via PCI express

Ray Bittner · Erik Ruf · Alessandro Forin

Received: 15 February 2013 / Accepted: 13 May 2013

Abstract We describe a mechanism for connecting GPU and FPGA devices directly via the PCI Express bus, enabling the transfer of data between these heterogeneous computing units without the intermediate use of system memory. We evaluate the performance benefits of this approach over a range of transfer sizes, and demonstrate its utility in a computer vision application. We find that bypassing system memory yields improvements as high as 2.2× in data transfer speed, and 1.9× in application performance.

Keywords GPU · FPGA · PCI express · PCIe · Xilinx · CUDA · nVidia · Windows · Verilog · Hand tracking · Computer vision

1 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, fill the computational gap left between clock rate and predicted performance increases.

The members of this parallel trifecta are not created equal. The main characteristics of each are:

- CPUs—ease of programming and native floating point support with complex multi-tiered 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 resource intensive 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, 5–7] support explicit transfers between the CPU and GPU, and it has recently been shown that this is also possible between CPU and FPGA [1]. In CUDA 5.0, nVidia recently announced GPUDirect RDMA [12], which enables its high-end professional GPUs to access the memory spaces of other PCIe [8] devices having suitably modified Linux drivers. The work in [15] uses an FPGA to implement peer-to-peer GPU communications over a custom interconnect. The work in [14] describes a high-level toolflow for building such applications that span GPU and FPGA; their implementation is currently bottlenecked precisely by the GPU–FPGA data transfer path.

Existing facilities may be used to implement GPU to FPGA communication by transferring data through CPU memory as illustrated by the red lines in Fig. 1. With this indirect approach, data must traverse the PCI Express (PCIe) switch twice and suffer the latency penalties of both the operating system and the CPU memory hardware. We refer to this as a GPU–CPU–FPGA transfer. This additional indirection adds communication latency and operating system over-

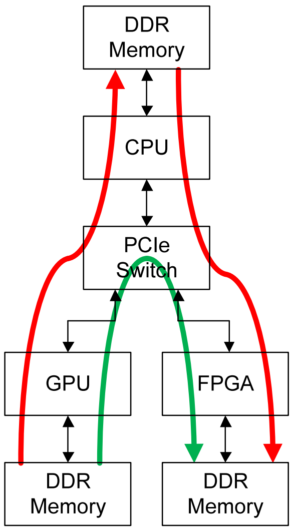
R. Bittner · E. Ruf (✉) · A. Forin
Microsoft Research, Redmond, USA
e-mail: erikruf@microsoft.com

R. Bittner
e-mail: raybit@microsoft.com

A. Forin
e-mail: sandrof@microsoft.com

Published online: 08 June 2013

Fig. 1 Two conceptual models of GPU to FPGA transfers



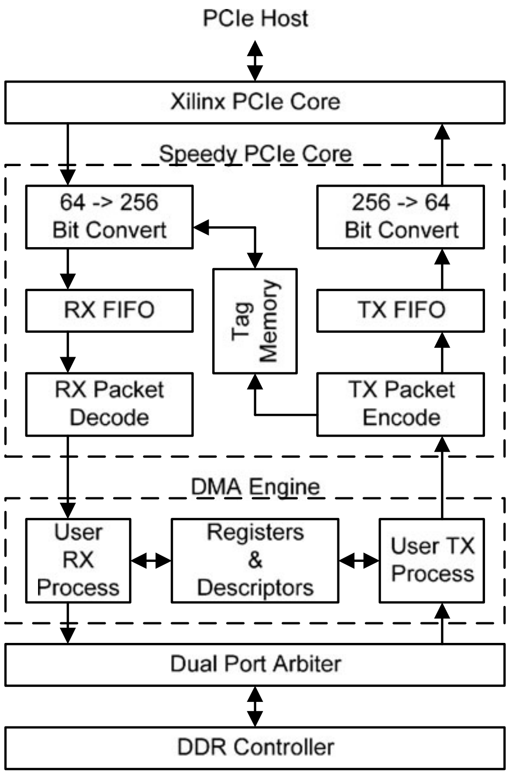


Fig. 2 Speedy PCIe core

head to the computation, as well as consuming bandwidth that could otherwise be used by other elements sharing the same communication network.

In this paper, we describe and evaluate a mechanism for implementing the green line in Fig. 1, creating a direct, bidirectional GPU–FPGA communication over the PCIe bus [2]. With this new approach, data moves through the PCIe switch only once and it is never copied into system memory, thus enabling more efficient communication between these two computing elements. We refer to this as a direct GPU–FPGA transfer. This capability enables scalable heterogeneous computing systems, algorithm migration between GPUs and FPGAs, as well as a migration path from GPUs to ASIC implementations.

The remainder of the paper is structured as follows. Section 2 describes the FPGA's PCIe implementation. Section 4 describes the mechanism that enables direct GPU–FPGA transfers. Section 4 describes our experimental methodology. Section 5 presents the experimental results, which are discussed in Sect. 6. Section 7 describes how a real-world computer vision application benefits from the direct path. Sections 8 and 9 describe future work and the conclusion.

2 The speedy PCIe core

Interfacing an FPGA to the PCIe bus 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 [1] has delivered a viable solution at no cost that can be made to serve for the problem at hand.

The Speedy PCIe core shown in Fig. 2 is a freely downloadable FPGA core designed for Xilinx FPGAs. It leverages the Xilinx PCIe IP [11] to provide the FPGA designer a memory-like interface to the PCIe bus that abstracts the addressing, transfer size and packetization rules of PCIe. The standard distribution includes Verilog source code that turns this memory interface into a high speed DMA engine that, together with the supplied Microsoft Windows driver, delivers up to 1.5 GByte/s between a PC's system memory and DDR3 memory that is local to the FPGA.

The Speedy PCIe core 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 in the range of 250 ns to 1 µs, while reads are in the range of 1 µs to 2.5 µs. The savings offered by the Speedy PCIe core directly contribute to lower latency transfers at the application level, as we will show later.

Lastly, the Speedy PCIe core was designed with peer to peer communication in mind. By exposing the FPGA's DDR memory on the PCIe bus, peer devices may initiate master reads and writes directly into that memory; a capability that

Table 1 Master/slave relationships

Transfer	PCIe master	PCIe slave
GPU–CPU	GPU	CPU
FPGA–CPU	FPGA	CPU
GPU–FPGA	GPU	FPGA

we exploit when enabling direct GPU–FPGA communication.

3 Enabling the missing link

On the GPU side, we have somewhat less control, because all of the hardware functionality remains hidden behind an opaque, vendor-supplied driver and its associated Application Programming Interfaces (APIs). Typically such APIs support only transfers between GPU and CPU memories, not between GPU memory and that of arbitrary devices. In CUDA 4.x, nVidia added a peer-to-peer (GPU-to-GPU) memory transfer facility to its professional-level Quadro and Tesla product lines, but transactions involving arbitrary PCIe devices, such as our FPGA development board, are not supported.

In an implementation such as ours, which targets consumer-level GPUs, the GPU must always be the bus master in any transfer in which it is involved. To use the GPU as a slave, we would need access to the its internal structures. If the GPU must always be the bus master, it follows that in the direct GPU–FPGA data path the FPGA must always be the slave. This requires the FPGA to map its memory (on chip or otherwise) onto the PCIe bus so that the GPU may read or write directly to it as needed. Luckily, 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. The ensuing master/slave relationships are summarized in Table 1 for each transfer type.

We found 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 *malloc*()-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. Note that the routine succeeds 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 Speedy PCIe 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 memcpy()-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 memory 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.

Specifically, the function call sequence needed to initiate a GPU to FPGA transfer is shown in Fig. 5. First, *cudaMalloc*() is called to allocate memory on the GPU. At this point, the user executes the GPU code *run_kernel*() that mutates this memory. The following two steps make the FPGA target address visible to the GPU. The call to *DeviceIOControl*() causes the FPGA device driver to return the virtual pointer *fpga_ptr* in the user's address space that directly maps to the physical addresses for the FPGA's memory. As mentioned, this pointer can be used just like any other; resulting in PIO accesses to the FPGA's memory on the PCIe bus when accessed programmatically from the CPU. Subsequently, *cudaHostRegister*() is called on the *fpga_ptr* pointer to request page locks on the memory address range associated with it. This is guaranteed to succeed since the FPGA's driver locks the memory range before it is returned to the user. Finally, *cudaMemcpy*() is called to cause the GPU's driver to initiate a bus master DMA from the GPU memory to the FPGA memory. In practice, the CPU's involvement is minimal, being only used to provide the virtual to physical address mapping between the two memory regions.

4 Test procedure

For our testing, we chose an nVidia GeForce GTX 580, a high-end consumer GPU that supports the whole CUDA 4.1 API, with the sole exception of the 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/s of throughput.

The FPGA platform used in our tests is a Xilinx ML605 development board with an integrated V6LX240T-1 Xilinx FPGA. This unit supports 8 generation 1.0 PCIe lanes, with a maximum throughput of approximately 1.6 GByte/s [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 × 16.

Transfers between CPU and FPGA memories are implemented using the native functionality of the Speedy PCIe driver, allowing the user application to transfer arbitrary memory buffers between the CPU and FPGA, using DMA. The Speedy PCIe driver exposes this capability by providing the user with a file handle that represents the FPGA. The user calls the standard file system *Read* and *Write* operations with this file handle in order to induce DMA-based memory 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 are accomplished via the same *cudaMemcpy*() API described before, but this time it is used in the traditional sense of transferring to and from a user buffer in system main memory. In this case the GPU acts as the bus master. Transfers between GPU and FPGA memories were performed by using *cudaMemcpy*() as described in the code sample of Fig. 5, with the GPU acting as the PCIe bus master.

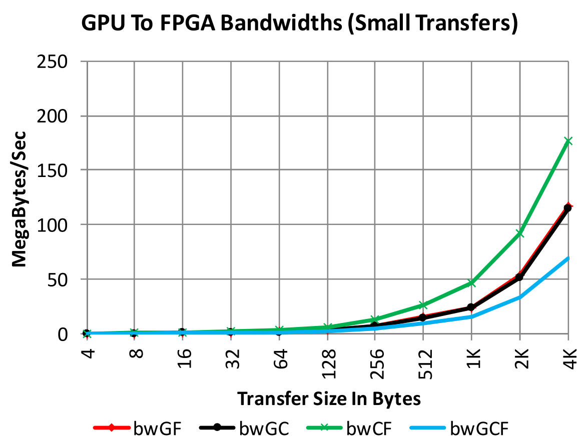


Fig. 3 Performance of small GPU to FPGA transfers

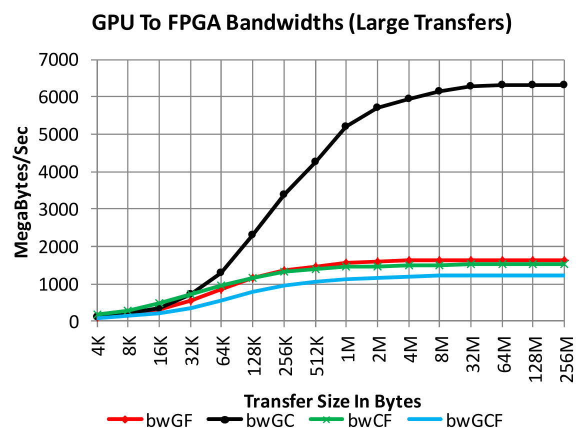


Fig. 4 Performance of large GPU to FPGA transfers

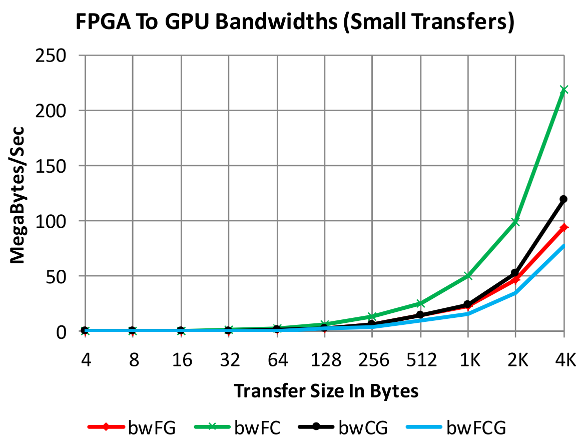


Fig. 6 Performance of small FPGA to GPU transfers

5 Results

The data transfer characteristics are asymmetric. In Figs. 3, 4, 5, 6, 7, we show separate graphs for each transfer direction, and separate graphs for small and large transfers. 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.

Figures 3 and 4 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)—the indirect data path from GPU to CPU to FPGA.

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 shown in Fig. 4, the black line indicating the GPU to CPU bandwidth dominates the graph as the GPU supports generation 2.0 × 16 lane PCIe. Since the FPGA (green line) we used only supports generation 1.0 × 8 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. 3, for small transfers the CPU to FPGA path dominates due to the smaller latencies as measured in Table 2.

Figures 6 and 7 show the achieved bandwidths in 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 bandwidth (red line) is markedly lower than in the GPU to FPGA case.

We measure the transfer latency by the elapsed time of the smallest possible PCIe data transfer of 4 bytes as mea-

Table 2 Four byte transfer latencies

Transfer points and direction	Latency (μs)
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

sured at the level of the CUDA API. These measured latencies are summarized in Table 2. 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 μs) over the direct GPU–FPGA transfers in both directions (Fig. 8).

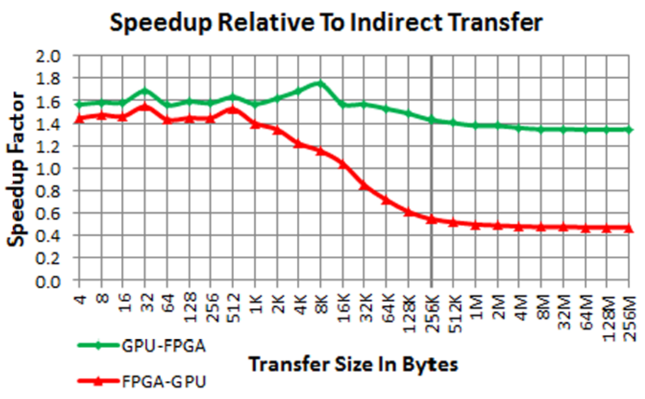


Fig. 7 Speedup of GPU–FPGA relative to GPU–CPU–FPGA transfers

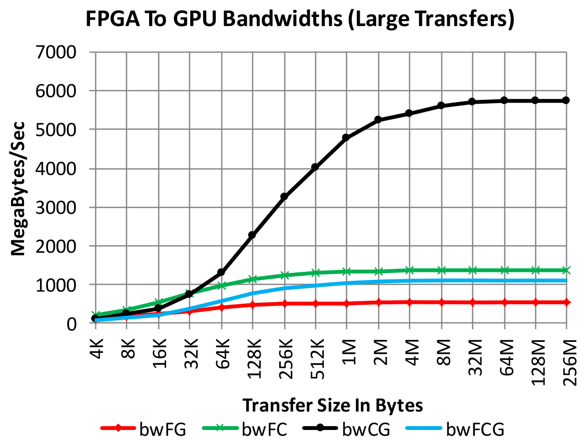


Fig. 8 Performance of large FPGA to GPU transfers

6 Discussion

The primary limiting factor in our implementation is the bandwidth supported by the FPGA. The FPGA is only operating in a generation 1.0, ×8 lane mode, therefore its performance reaches a maximum of 1.6 GByte/s regardless of the transfer partner. It is 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 × 16 lane operation; this gives it a 4× speed advantage over the FPGA with a measured maximum of 6.2 GByte/s.

The two graphs show that the two data directions have asymmetric bandwidth characteristics. This is visualized in Fig. 9, which graphics the relative speedups comparing direct GPU–FPGA transfers and indirect GPU–CPU–FPGA transfers. 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 performance. In the GPU to FPGA case, the performance improvement over GPU to CPU to FPGA settles at 34.6 % for

Fig. 5 GPU to FPGA DMA code sample

```
// Allocate GPU memory
gpu_ptr = cudaMalloc( MEM_SIZE );

// Perform GPU computations that modify GPU memory
run_kernel(gpu_ptr, …);

// Map FPGA memory to CPU virtual address space
fpga_ptr =
DeviceIoControl(IOCTL_SPEEDYPCIE_GET_DIRECT_MAPPED_POINTER);

// Present FPGA memory to CUDA as CPU locked pages
cudaHostRegister( fpga_ptr, MEM_SIZE );

// DMA from GPU memory to FPGA memory
cudaMemcpy( fpga_ptr, gpu_ptr, MEM_SIZE,
cudaMemcpyDeviceToHost );
```

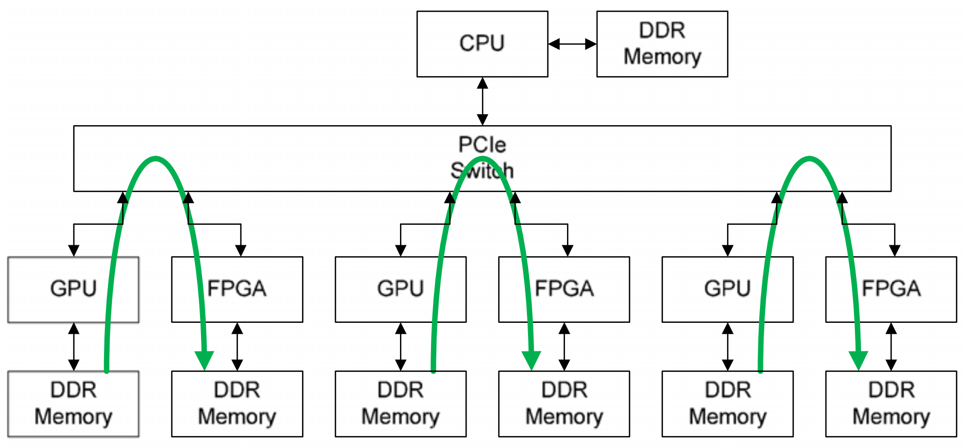


Fig. 9 Simultaneous DMA datapaths

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. 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 our 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 absorb these at full speed (1.6 GByte/s). 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/s.

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/s. In the opposite direction from CPU to GPU, the GPU is initiating bus master reads and the resulting bandwidth falls to 5.61 GByte/s. 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 2 are interesting because they show that the path between the CPU and FPGA has 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 4× the transfer bandwidth of the FPGA and so the latency is expected to be lower. We do not have a definitive explanation of this phenomenon, due to the above mentioned opacity of the GPU APIs. 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 these performance comparisons are slightly deceptive in that they do not account for potential bottlenecks in the CPU's system memory, or in 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 will circumvent the system memory bottleneck by taking the direct routes through a local PCIe switch. Figure 9 illustrates this topology in a system that uses three pairs of GPU–FPGA devices, each pair transferring data simultaneously over the green line paths. The direct GPU–FPGA communication scales better, and attains much higher speedups because multiple transfers are happening simultaneously. While this possibility is predicated on the hardware details of the particular host platform being used, such independent data paths are also possible through the use of suitable switches. One such source of PCIe switches is PLDA [10].

7 Hand tracking application

We have chosen computer vision based hand tracking as a practical test of the direct GPU–FPGA data path. Figure 10 shows the data transfers through the system. On the left side, a depth camera captures raw IR data from a scene containing the user's hands. The camera is connected to the PC system through USB, so the IR data is first transferred to the system DDR memory, and from there to the GPU's memory. The GPU decodes the IR image and uses a proprietary algorithm to create a depth map. The picture with the green background located on the left side of Fig. 10 shows the result of the GPU computation. In the direct GPU–FPGA case, the GPU uses master writes to transfer the depth map directly through the PCIe switch into the FPGA's memory. In the indirect case, this data goes to system DDR first, and then from there to the FPGA's DDR. The FPGA processes the depth image via a decision tree algorithm [13] followed by ak-means based centroid calculation. The FPGA uses bus master writes to transfer the candidate centroids through the PCIe switch again to the CPU. The resulting centroids are shown in the upper-right side of Fig. 10, as colored dots identifying the various parts of a hand. Lastly, the CPU applies a model fitting algorithm to determine the best candidate centroids, identify the left and right hands, and label the centroids with the correct joints names. This results in the skeletal representation of the user's hands in the original scene which is shown at the bottom-right of Fig. 10. Note that the only difference between the direct and indirect path realizations is the way the depth map is transferred to the FPGA; everything else in the application remains the same.

The original GPU algorithms that decode the camera image were written in Microsoft Direct Compute, part of the DirectX 11 toolkit. DirectX eschews direct pointer manipulation in favor of abstract buffer and view objects that are transparently copied between devices as needed, meaning that (given our level of knowledge of the API) we were unable to directly obtain a pointer to the resulting depth map stored on the GPU. To make this possible, we used CUDA's interoperability API to view the DirectCompute result (on the GPU) as a cudaArray object which serves as an input to a CUDA kernel. The kernel also executes on the GPU—no physical copying takes place. After some additional filtering, the CUDA kernel writes its results to a GPU buffer. This final buffer is analogous to the buffer allocated with *cudaMalloc*() in Fig. 5, and it is therefore sent to the FPGA via the same operations shown in Fig. 5.

To allow for meaningful comparisons between the direct and indirect paths, we use the same mechanism to send the filtered data to the CPU, e.g. with a CPU memory address rather than an FPGA address. By instrumenting the copy code, we are able to capture more detailed measurements, as shown in Fig. 11.

The application system was implemented using the same Xilinx ML605 development board with an integrated V6LX240T-1 Xilinx FPGA as described in Sect. 4. As before, the FPGA utilizes PCIe v1.0 $\times 8$ lane transfers with an expected maximum throughput of approximately 1.6 GByte/s. The GPU is different, this time it is an nVidia GTX 480. Compared with the GTX 580 of our previous experiment, it has less compute power but similar PCIe bus bandwidth.

In this application, GPU–FPGA transfers are from the GPU into the FPGA, corresponding to GPU master write operations, which is the preferred direction for our FPGA implementation and therefore we do not incur the penalty described in Sect. 6. The input image from the camera has a resolution of 640 by 480 with 16 bits per pixel of encoded depth data. This forms an image of size 614,400 bytes that must be transferred from the camera to the GPU via the CPU. The GPU processing reduces the image size to 320×240 16-bit pixels, so 153,600 bytes are transferred from the GPU to the FPGA. The centroid data from the FPGA going to the CPU is small; amounting to 10K bytes or less per image.

We have measured the computation and transfer times for the application, averaging the results over 200 frames. The results are shown in the diagram in Fig. 11. The top half of the diagram provides the measurements for the indirect case, where the data is transferred through the CPU's memory using the GPU–CPU–FPGA path. The bottom half of the diagram provides the measurements when the direct GPU–FPGA DMA path is utilized. The largest contributor is the FPGA processing time, which is unfortunately data dependent. Times here range from a few microseconds in the idle case, to 20 milliseconds in the worst case. A two-hands image such as shown in Fig. 10 requires about 5 milliseconds of processing time. The GPU computation time is instead fixed at 60 µs. If we ignore the FPGA processing time, the rest of the pipeline requires 555 µs and 285 µs respectively for the two cases. The best application speedup possible is therefore 1.9×, in the idle case. The speedup reduces quickly with the FPGA processing time growing, and in the average case we only get a 5 % improvement. In all cases, the data transfer times are reduced from 495 µs to 225 µs, a factor of 2.2×.

The FPGA resources consumed by the full design are summarized in Table 3. The first line indicates the total resource count available on the device. While we consume a relatively small fraction of most of the FPGA's resources,

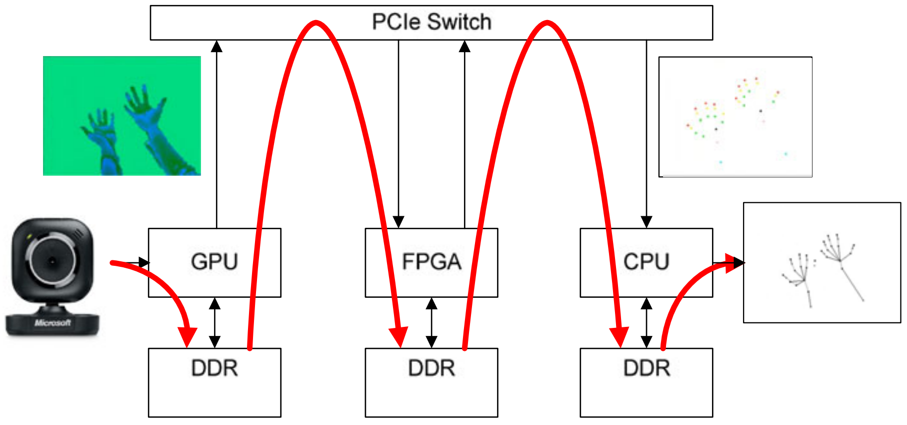


Fig. 10 Hand tracking application dataflow

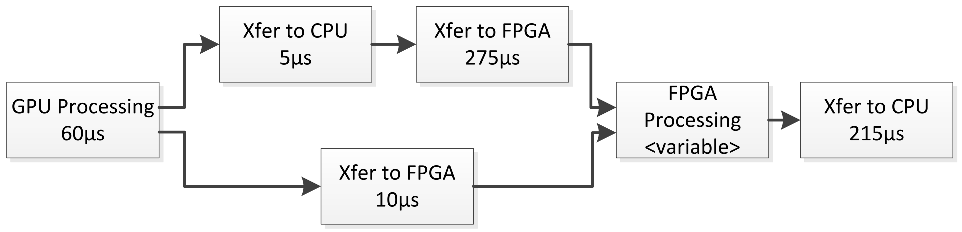


Fig. 11 Application timings, with (*bottom*) and without (*top*) the direct path

Table 3 FPGA resource utilization

Design unit	Slices	Flip flops	LUTs	DSP48	36Kb BRAMs
V6LX240T Available	37,680	301,440	150,720	768	416
Complete Design	29,942	58,406	40,823	23	138
Entire Image Processor	10,149	19,878	14,289	23	20
Decision Tree Eval.	766	1267	1318	6	0
Centroid Calc.	3239	5518	5523	17	16
DMA Engine & SIRC	10,776	23,695	12,959	0	89
Speedy PCIe Core	3386	6024	6868	0	13
Xilinx PCIe Core	598	941	971	0	16
MIG-based DDR3	4967	7756	5702	0	0

the "Complete Design" line indicates that most of the slices are occupied. Because of this, we had many problems in coercing the Xilinx tools to meet timing. We had to use a combination of general floor planning, pipelining and individual location constraints for the block RAMs and DSPs in order to generate repeatable successful routes using the Xilinx ISE 14.4 tools. The most sensitive parts of the design are the decision tree and centroid computation logic that run at 200 MHz (to correlate well with the 400 MHz speed of the local DDR3 memory bus), and the Xilinx core PCIe interface that must run at 250 MHz.

These problem areas were dealt with by running Xilinx's SmartExplorer with multiple parallel threads iterating on the cost table initialization variable. Typically, we obtained 2 or 3 acceptable routes out of 100 runs using this approach. Once a good route had been found, we used PlanAhead to extract the locations of all of the BRAM and DSP blocks and then used those LOC constraints as guides for further compilations. Routing was also much easier to achieve with a reduced clock rate for the application logic, rather than 200 MHz a clock rate of 175 MHz was much more achievable. We made use of the slower clock rate to ease routing during the debugging phase.

8 Future work

Our next order of business will be to address the 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 other applications that can 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 image 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 likely require the involvement of GPU vendors, as the relevant mechanisms are presently hidden behind black-box driver code.

9 Conclusions

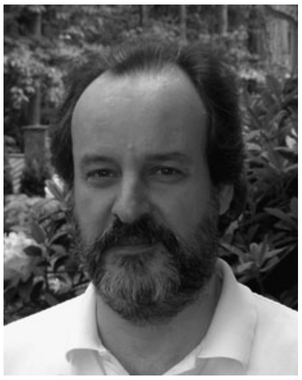
We have presented a mechanism for direct GPU–FPGA communications via PCI Express, and analyzed its performance characteristics and ease of use. Our hope is that this opens the door to new computation synergies and architectures that were previously unsuitable or perhaps not considered practical. We have presented a case study with a computer vision application, where the data transfer times are cut in half when using this new direct path. Our implementation is available as a free download, making it possible for other researchers to expand on our initial steps.

References

1. Bittner, R.: Speedy bus mastering PCI express. In: 22nd International Conference on Field Programmable Logic and Applications (2012)
2. Goldhammer, A., Ayer, J. Jr.: Understanding performance of PCI express systems. Xilinx WP350 (Sept. 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//directxompute.aspx
6. nVidia Corporation: nVidia CUDA API reference manual, version 4.1. Available at: http://ww.nvidia.com/CUDA
7. nVidia Corporation: nVidia CUDA C programming guide, version 4.1. Available at: http://ww.nvidia.com/CUDA
8. PCI express base specification, PCI SIG: Available at http://www.pcisig.com/specifications/pciexpress
9. Whitted, T., Kajiya, J., Ruf, E., Bittner, R.: Embedded function composition. In: Proceedings of the Conference on High Performance Graphics (2009)
10. PLDA Corporation: http://www.plda.com/prodetail.php?pid=175
11. Xilinx Corporation: PCI express. Available at: http://www.xilinx.com/technology/protocols/pciexpress.htm
12. nVidia GPUDirect: http://developer.nvidia.com/gpudirect
13. Oberg, J., Eguro, K., Bittner, R., Forin, A.: Random decision tree body part recognition using FPGAs. In: International Conference on Field Programmable Logic and Applications, August (2012)
14. da Silva, B., Braeken, A., D'Hollander, E., Touhafi, A., Cornelis, J.G., Lemiere, J.: Performance and toolchain of a combined GPU/FPGA desktop. In: 21st International Symposium on Field Programmable Gate Arrays (FPGA'13), Monterey, CA, February (2013)
15. Rossetti, D., et al.: GPU peer-to-peer techniques applied to a cluster interconnect. In: Proceeding of the Third Workshop on Communication Architecture for Scalable Systems (2013)

Ray Bittner enjoyed his time at Virginia Tech, obtaining a B.S. degree in Computer Engineering in 1991, as well as M.S. (1993) & Ph.D. (1997) degrees in Electrical Engineering. Since graduating he has worked at Microsoft on various embedded products, eventually joining Microsoft Research where he has concentrated on the uses and applications of FPGAs as well as portable embedded platforms.

Erik Ruf received the M.S. (1989) and Ph.D. (1993) degrees in Electrical Engineering from Stanford University. He is a Senior Researcher and a member of the Embedded Systems group at Microsoft Research, Redmond, WA, USA. His research interests span program analysis and transformation, compilers, acceleration technologies, and embedded systems. He is a member of the ACM and the IEEE.



Alessandro Forin received the M.Sc. degree in Electrical Engineering in 1982 and the Ph.D. degree in Computer Science in 1987, both from the University of Padova, Padova, Italy. He is currently a Principal Researcher at Microsoft Research, Redmond, WA, USA, and an Adjunct Professor with the Department of Computer Science at Texas A&M University, College Station, TX, USA. His research interests include reconfigurable computing, embedded systems, programming languages, debuggers, and operating systems. He is member of the Association for Computing Machinery (ACM).