FPGA^2: An Open Source Framework for FPGA-GPU PCIe Communication

Yann Thoma, Alberto Dassatti, Daniel Molla
Reconfigurable and Embedded Digital Systems Institute - REDS
HEIG-VD // School of Business and Engineering Vaud; HES-SO // University of Applied Sciences Western Switzerland
CH-1400 Yverdon-les-Bains, Switzerland
Corresponding author: Yann Thoma. E-mail: yann.thoma@heig-vd.ch. Phone: +41245576273. Web: http://reds.heig-vd.ch

Abstract—**In recent years two main platforms emerged as powerful key players in the domain of parallel computing: GPUs and FPGAs. Many researches investigate interaction and benefits of coupling them with a general purpose processor (CPU), but very few, and only very recently, integrate the two in the same computational system. Even less research are focusing on direct interaction of the two platforms [1].**

This paper presents an open source framework enabling easy integration of GPU and FPGA resources; Our work provides direct data transfer between the two platforms with minimal CPU coordination at high data rate and low latency. Finally, at the best of our knowledge, this is the first proposition of an open source implementation of a system including an FPGA and a GPU that provides code for both sides.

Notwithstanding the generality of the presented framework, we present in this paper an actual implementation consisting of a single GPU board and a FPGA board connected through a PCIe link. Measures on this implementation demonstrate achieved data rate that are close to the theoretical maximum.

I. Introduction

FPGAs and GPUs are two platforms capable of accelerating data processing, both of them being a good alternative to standard CPU computing for specific applications. The choice of one platform should depend on the type of calculation to be performed, and [2] gives a good overview of the characteristics that should drive such decision. On one end, FPGAs are excellent for fully pipelined computing, and for low-level bit-wise operations (cryptography applications, for instance). On the other end, GPUs are easier to program and are very good candidates for massive parallel computing applications showing a low dependency between data. Programming is normally performed on high level software languages such as C/C++ supported by a rich set of tools for profiling and debugging. Parallelism is achieved by means of specialized languages extensions and libraries, notable examples being CUDA [3] and OpenCL [4].

In this context, a lot of literature illustrates accelerators developed with GPUs or FPGAs platforms, but the conjunction of both has not been exploited until recently. The work presented in this paper aims to ease the development of such heterogeneous system, and could serve in different areas: for example, a video application with multiple HD cameras could let a FPGA handle the image capture and the first image processing, the GPU taking care of more advanced image processing. Another field of interest is medical imaging: in [5] the coupling of FPGA and GPU is proposed as an effective solution at high frame rate processing. Another possible application is in genomic processing. An ongoing project aims at accelerating the genome comparison by refactoring the genome compression format. For this application an FPGA would be perfect for the genome decompression, while the GPU could then handle part of the comparison. For all of these applications, the bottleneck resides in the data throughput that can be achieved through rapid communication lines.

GPUs are commonly plugged onto PCIe slots, and on the FPGA side this bus interface is becoming a common utility in recent chips. Therefore it is quite easy to build a setup using a standard PC in which a GPU board and an FPGA board are plugged on the same motherboard. The challenge is then to allow the fastest possible communication between the two platforms.

A. FPGA-GPU communication

Whenever we would like to exchange data between an FPGA and a GPU the standard flow requires to allocate a memory buffer for each device and one extra buffer in the CPU memory. Once the buffers are allocated, we need two memory copy operations: from the first device to the main memory and then from the main memory to the second device. Unfortunately this is not always the case and many actual implementations exhibit even worse scenarios. If we look at the CPU drivers we realize that each peripheral driver uses its own memory buffer and so a third memory copy is necessary from CPU memory to CPU memory. This flow of operations, depicted in figure 1(b) has three main effects:

- Increasing memory requirements, doubled compared with minimal requirement
- Reducing global throughput
- Increasing total latency

Although direct communication between FPGAs and GPUs is the natural evolution of such a system, the reality is quite discomforting. At our knowledge, there is no open enabling technology available today. Partial solution exists, however. NVIDIA proposed recently GPUDirect [6], a software package enabling RDMA, direct GPU-GPU communication and possibly Peer-to-Peer memory access. This solution was only announced (and not available) at the time of this project and the few information available on the integration of a custom peripheral, the FPGA in our case, was coming from APEnet+ project [7] where a direct support from NVIDIA was claimed. This solution is also limited to a specific family of devices, where a more general solution could better serve the vast area of potential applications.

Therefore the aim of the project presented here consisted in the realization of an open source framework allowing an efficient FPGA⇔GPU communication through PCIe. As stated above, a standard data transfer requires too many memory operations, as illustrated in figure 1(a). The $FPGA^2$ project proposes a solution where data do not need to go through the main memory, but only requires direct DMA transfer between both devices, as shown in figure 1(c).

In this setup, the central CPU only serves as a controller to start memory operations, and to synchronize the GPU and the FPGA. Focusing on the PCIe network, this setup enables transfers through the South bridge of the motherboard, allowing the CPU to access the central memory while easing congestion situations.

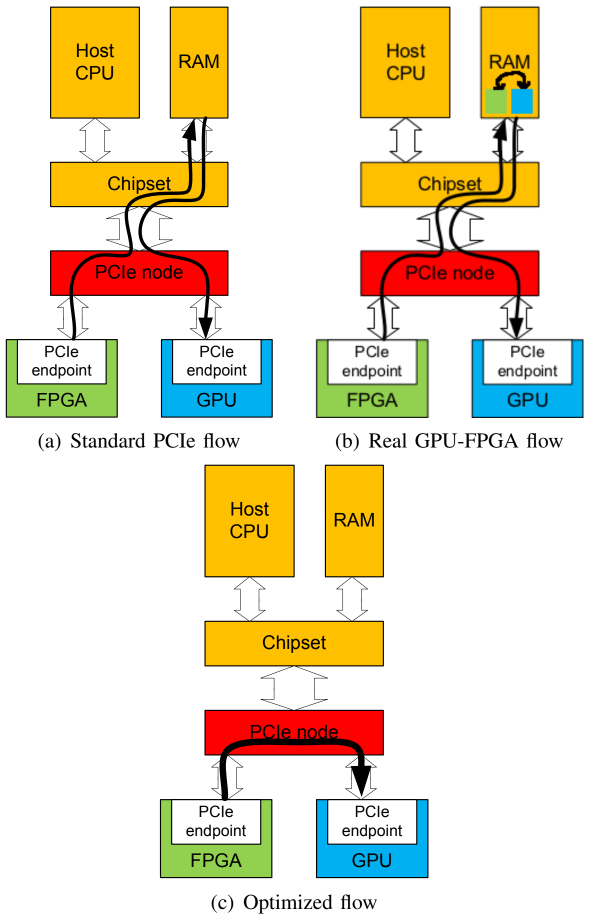


Fig. 1. Standard data flow and optimized transfers

Considering the vast set of scenarios that could benefit from our framework, the design presents two interfaces to the developer: the FPGA being seen as a simple memory, or as a FIFO.

1) FPGA seen as a memory: Both at the hardware and software side, the framework offers to simply view the FPGA as a memory region, the software driver being able to access data through addresses, with direct memory access or via DMA transfers.

2) FPGA seen as a FIFO: As some application would require data streaming, the FPGA can be seen as two FIFOs (input and output). In this configuration, accessing the FPGA via a special address range forces the FPGA design to supply data to an input FIFO (in case of a write access) or to get data from an output FIFO (reading access).

II. Hardware Setup

For the first communication tests, two boards provided by Xilinx have been used:

- The ML506 contains especially a 1-Lane connector for PCI Express Designs and a *Virtex 5 XC5VSX50T* FPGA.
- The ML555 contains especially a flash, a 8-Lane connector for PCI Express Designs and a *Virtex 5 XC5VLX50T* FPGA. Only one lane of the connector was used in the project.

These two boards allowed to validate the FPGA design responsible for the PCIe communication. The design currently only exploits one lane, but further improvements could lead to a multi-lane solution. After the first validations, only the ML555 board has been used during the development phase.

The board was plugged into a standard PC, the central CPU being an Intel Core i7. The GPU used consisted in a NVIDIA GeForce 8400 GS, from the NV50 family.

The FPGA design is based on a PCIe IP core offered by Xilinx through Coregen. This IP is responsible for handling the low-level PCIe communication and offers a synchronous interface to the user design. Transaction Level Packets (TLPs) are transmitted using a 64-bits data interface, the control being pretty easy to handle.

Figure 2 represents a schematic view of the FPGA design. The arrows correspond to the data lines, while all the control paths are not shown.

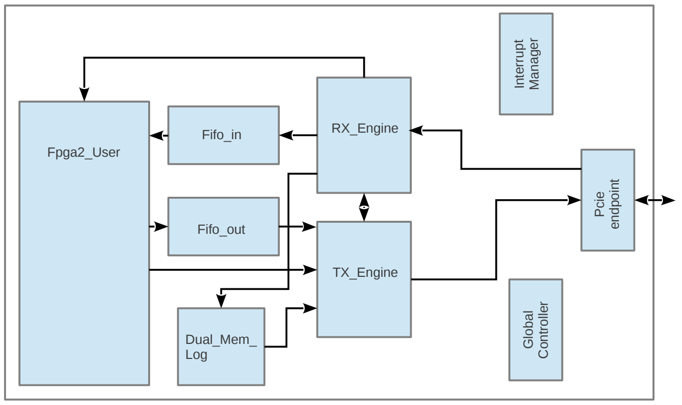


Fig. 2. Schematic view of the FPGA design

The most important blocks are the following:

1) **PCIe endpoint**: This endpoint consists in an IP core generated with Coregen. It instantiates the Multi-Gigabit Transceiver (MGTs) and is responsible for managing the PCIe link and packets. Its interface with the FPGA subsystem is mainly a simple protocol for the RX engine and the TX engine, allowing them to get and send packets in an simplified fashion. It also exchanges some control signals with the interrupt manager, as everything goes through this endpoint for any kind of communication with the rest of the system.
2) **Interrupt manager**: This component is responsible for the management of the interrupts. It holds registers that allow to raise/clear/mask interrupts. It also contains a small state machine that implements a simple protocol to manage the interrupts sent by the PCIe endpoint. We can also notice that the user logic can generate interrupts, as well as the RX and TX engines.
3) **TX engine**: This module is responsible for sending packets to the PCIe infrastructure. It is directly interfaced with the PCIe core, and is therefore dependent on this core definition. The sending of packets is triggered by other modules: The user application, the RX engine in case of read requests that need an answer, and the global controller for DMA transfers.
4) **RX engine**: This module is responsible for getting packets from the PCIe endpoint. It processes these packets and acts accordingly. Typically the types of packets correspond to a standard write (to the Fifo, control registers, interrupt manager, or user logic), a read request, or a read completion. It is then responsible for sending the right command to the right component.
5) **Fifo In**: This FIFO serves as a simple way to see a stream of data, from the user logic point of view. Data comes from the PCIe fabric, during standard writes or during DMA read completions. The user logic can then retrieve its data.
6) **Fifo Out**: This FIFO serves to send a stream of data, from the user logic point of view. It has the capability of generating an interrupt when a programmable level of fullness is reached. With this feature enabled the user logic can send data in it and the host CPU can be triggered to execute standard reads or DMA transfers from FPGA to GPU. Once the programming is done, it is then the TX engine that takes care of retrieving the data from the FIFO.
7) **Dual Mem Log**: This is a dual port memory that keeps trace of incoming packets. Basically it can log the first 64-bits of any packet received by the RX engine. Its size can be easily parametrized and the content of the memory can then be retrieved by means of read operations. This log memory was particularly useful during the development process because it is the only effective way to view the Transaction Level Packets [9] transferred on the physical link. This module is optional and can be removed if not needed.
8) **FPGA2 User**: This module represents the user logic. It is currently almost empty. It contains a simple dual-port memory as example, but is really the part that will have to be modified by any developer who would like to implement a new application. It can raise interrupts, get data from the input FIFO, send data through the output FIFO, and can be accessed like a standard 1-cycle latency memory.

A. User Application

As the goal of the project was to offer a framework for easing the realization of new applications, we embedded a module that can be adapted to the need of any application.

It basically offers four interfaces to the application, as shown in figure 2:

1) A **standard memory-like access**, the user application being seen like a memory
2) An **input FIFO**
3) An **output FIFO**
4) A possibility to trigger **interrupts** transmitted to the host CPU

Through the memory interface, a user design can implement any memory map, and so any kind of functionality. This interface is fully synchronous, the only limitation being the 1-clock cycle latency of the read operation.

Despite the fact that the memory interface can be sufficient for any application, two FIFO interfaces have been included in order to ease the realization of streaming applications. The input and output FIFOs are directly accessible by the user logic, and can trigger interrupts when being in a certain state (typically half full).

Finally, the user logic can trigger interrupts through 16 lines, allowing it to signal interesting events to the software. It is mainly useful to indicate the end of a particular processing, letting then the software reacts, launching, for example, a data transfer.

B. Synthesis results

The FPGA design was synthesized and placed/routed with Xilinx ISE 13.3, with a user logic consisting in a dual-port memory. The results after placement/routing, for a design embedding input and output FIFOs of 1024 dwords, a 1024 dwords Log memory, and a user memory of 512 dwords are presented in table I.

Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	4,090	32,640	12%
Number of Slice LUTs	6,234	32,640	19%
Number of occupied Slices	2,820	8,160	34%
Number of BlockRAM/FIFO	8	132	6%
Number of BUFG/BUFGCTRLs	4	32	12%

TABLE I. PLACEMENT/ROUTING RESULTS FOR AN EXAMPLE DESIGN

These results illustrate the minimal resources needed by the interface logic. These numbers are for a full featured system and can be further reduced if some components, for instance the streaming interfaces, are not needed.

III. Software setup

Choosing an open source operating system, as Linux for example, provides a complete control over the available hardware. GPUs are the main exception to this assertion. While today all major hardware components of consumer electronics are perfectly supported by open source drivers, this is only seldom the case for GPUs. GPU vendors historically preferred an obfuscated interface between the kernel and their chips in the aim of unveil as less as possible of their intellectual property. While this caused some issues with the diffusion of open source systems this is a major limitation to the spread of new ideas and research due to the minimal or none commercial interest of a major company in supporting niche markets. Considering the interest of using these chips in open source systems, many independent groups investigated the matter and today open source drivers are available and are in some extent fully functional. Their performance are not always as good as the proprietary implementation, but they allow easier integration and modification and the gap is shrinking with time. In the chosen platform we faced this issues in the attempt of supporting NVIDIA GPUs and their programming paradigm: CUDA. NVIDIA GPUs are supported by a proprietary driver and CUDA is supported by a proprietary, closed source, library. Our first attempt was using these two components to enable direct communication between GPU and FPGA. Making use of CUDA memory allocation functions it is straight forward to transfer data from and to the GPU with excellent performances. While this approach works it has a major drawback: it requires data to be in the main system memory and we are unable to instruct the DMA engines of the GPU to work directly with buffers allocated in the FPGA. These are major limitations in a framework for high performance computing. The second attempt was adding DMA engines to the FPGA and then use them to master the transfer. This is useful to transfer data from the FPGA to the main memory, but with the proprietary driver there is no way to retrieve the physical address of an allocated buffer in the GPU and consequently it is impossible to drive a direct transfer between the two platforms. These three phases process (see figure 1(b)), transferring data from FPGA to main memory, then from main memory to GPU and finally elaborate them, can be partially overlapped reducing data-rate bottlenecks [1]; unfortunately, nothing can be done to alleviate the latency penalty and the memory waste of this approach.

At this stage we looked at alternatives to the closed source drivers. An open source alternative exists and is actively developed: nouveau [10]. This driver is functional and supports many hardware acceleration, unfortunately it is unable to cope with the CUDA extension proprietary library. Once again the open source community comes up with an alternative implementation of these functionalities in the project gdev [11]. Coupling the two open source projects we are able to use CUDA with the GPU and, more interestingly, gdev provides the functionality of retrieve the physical address of a buffer allocated in the GPU memory. In order to enable this feature, the allocated buffer has to be placed in the PCIe aperture of the card, and consequently the maximum size of this type of buffer is limited compared with the amount of available board memory. This is only partially a critical limit considered that we can transfer data from buffers within the GPU card at very high speed, in our case in excess of $5GB/s$. Once we have the physical address of our buffer we can instruct the FPGA to initiate a DMA transfer to the GPU memory, providing a direct link between the two platforms, limiting the use of the CPU to interrupt handling.

On the other side our FPGA implementation is feature rich and a specific driver was written to fully support it in the system. This driver delivers standard `FILE` abstraction operations to read and write from the memory mapped region as well as some special `ioctls` wrapped in a user space library to interact with the streaming interface.

[1]only when the two platforms sit on different PCIe paths

IV. Results

We fully benchmarked the proposed design. Results of this measure campaign are resumed in tables II and III were two scenarios are compared. $FPGA^2$ becomes useful when we would like to transfer data between the two devices without the intervention of the host machine as explained in detail in section I. If $FPGA^2$ is not used the only alternative is to copy the data from one device to the main memory and then from there to the second device.

Transferring data within the $FPGA^2$ framework requires 2 buffers, one allocated in each device. On the other hand, without $FPGA^2$ we need at least one extra buffer in the host machine but, due to actual GPU driver implementation limitations, practically two extra buffers are needed, doubling the memory requirements.

Data transferred per second are also slightly reduced working outside the $FPGA^2$ framework: two memory operations and an extra interruption are needed and this increases the time necessary to complete the transfer. Transferring data from the FPGA to the GPU can be accomplished in two ways: using $FPGA^2$ direct DMA transfers or using a combination of DMA transfers from the FPGA to the host memory and then employing CUDA function `cuMemCpyHtoD` to transfer from the host to the GPU. Data rates measured in the two situations are presented in tables II and III for various sizes of data transfer. It is important to notice that the GPU has more than one single PCIe lane. This explains the data rate achievable with `cuMemCpyHtoD`. It is important to stress the difference in performance achievable in reading and writing. Writing to the FPGA memory is slower because we use the FPGA's DMA engine to fetch data. This approach requires an extra PCIe packet for each DMA transfer, limiting the global throughput. We presented this approach instead of using the system DMA or the GPU DMA for this transfer because it is the only one available to transfer data to both targets (GPU or host memory).

Figures 3 and 4 present the same data graphically. From figure 3 we can highlight how at a size of $256k$ bytes we have a peek in the performances, while, for larger sizes, performance degrades. Repeating the measure multiple times and mean the results confirmed that this behaviors is consistent. We can only make the hypothesis that the reason behind this degradation is due to the GPU read request response time and we have no control nor visibility over the GPU internals. In order to make the comparison between the two systems easier for the reader, table II and III present a column with the GPU data

transfer size	cuMemCpyHtoD [MB/s]	cuMemCpyHtoD [MB/s] per lane	dma_from_fpga [MB/s]
32b	2.628	0.164	0.677
64b	5.303	0.331	1.276
128b	10.387	0.649	2.542
256b	20.455	1.278	5.463
512b	41.851	2.615	10.338
1k	83.217	5.201	20.216
2k	66.821	4.176	30.356
4k	132.330	8.270	59.347
8k	260.338	16.271	79.417
16k	258.906	16.181	96.404
32k	816.972	51.060	130.668
64k	1275.952	79.747	170.828
128k	1770.036	110.627	185.038
256k	1979.053	123.690	195.066
512k	2488.911	155.556	199.422
1M	2465.544	154.096	201.783
2M	1977.173	123.573	202.539
4M	3191.864	199.491	203.417
8M	4065.098	254.068	203.895

TABLE II. FPGA TO GPU TRANSFER COMPARISON DATA

transfer size	cuMemcpyDtoH [MB/s]	cuMemcpyDtoH [MB/s] per lane	dma_to_fpga [MB/s]
32b	2.525	0.157	1.039
64b	4.773	0.298	1.449
128b	7.832	0.489	3.852
256b	11.624	0.726	7.607
512b	15.088	0.943	14.758
1k	18.010	1.125	27.744
2k	115.296	7.206	35.527
4k	224.833	14.052	66.290
8k	415.359	25.959	77.336
16k	641.631	40.101	116.767
32k	1000.832	62.552	122.422
64k	1492.145	93.259	168.454
128k	1929.458	120.591	181.472
256k	1350.322	84.395	189.261
512k	2861.639	178.852	149.113
1M	2858.155	178.634	149.965
2M	1930.869	120.679	150.802
4M	3199.055	199.940	151.511
8M	3262.074	203.879	151.783

TABLE III. FPGA FROM GPU TRANSFER COMPARISON DATA

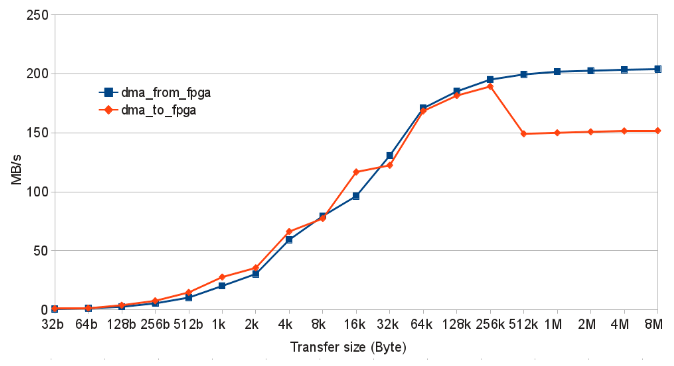


Fig. 3. FPGA DMA transfer rates

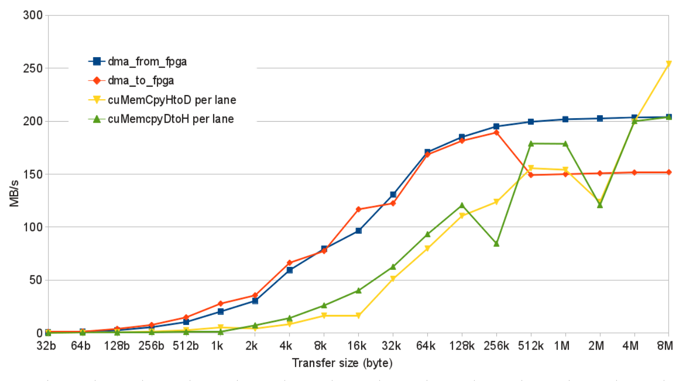


Fig. 5. FPGA and GPU per lane data transfer rates

rates per lane. These data can be easier comparable with the performance on the FPGA side of the system and figure 5 gives a graphical view of the data side by side. It is evident from these data our solution outperform standard solution for all transfer sizes smaller than $8MB$. We would like to remark another difference between the two PCIe sub-systems: GPU uses a version 2.0 of the protocol, while the FPGA implement the version 1.1. This is not a limitation for the communication being the two version compatible, but the data rate per lane is limited in the PCIe 1.1 to $250MB/s$ while it is doubled to $500MB/s$ in the more recent version [9]. We would like to recall this is the speed of the transceivers at the physical layer; this fact implies achievable logical data rates are upper bounded by these numbers multiplied by the available lanes. This final remark confirms that the last value in the second column of table II, where we find a value higher than $250MB/s$, is not a measurement error, but it is coherent with the system under test.

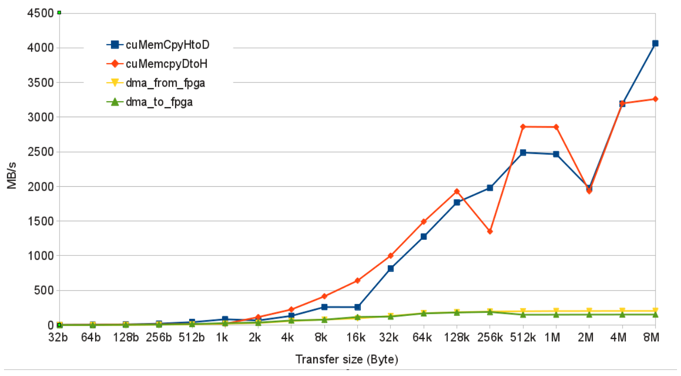
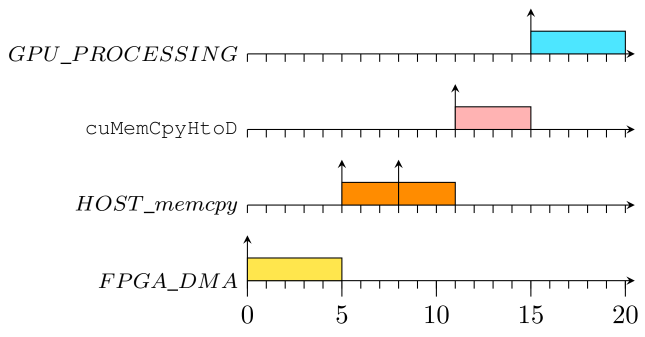
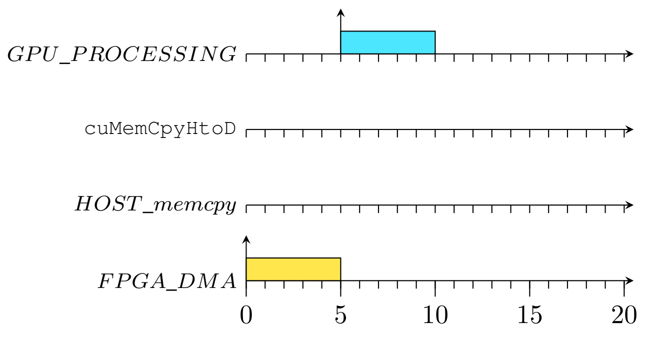


Fig. 4. FPGA and GPU data transfer rates

Latency is another interesting aspect of the communication. Working without the framework leads to an increased latency. From the time when a data buffer is available we need to notify the host processor, then, if we use $FPGA^2$, a DMA transfer takes place and, at its end, data are available for processing on the target device. Without $FPGA^2$ the first notification triggers a DMA transfer to the host memory; at the operation end a copy between host memory buffers takes place, and then a second DMA transfer data to the second device is performed. Only at the end of the latter operation, data are available for processing. While a partial overlap of operation is possible, and recommended, to reduce the impact on the throughput, there is no possible alleviation of latency increasing. The two schedule examples shown on Figure 6(a) and 6(b) can better help to compare the two situations.



(a) FPGA to GPU transfer without $FPGA^2$



(b) FPGA to GPU transfer with $FPGA^2$

Fig. 6. FPGA to GPU transfer with and without $FPGA^2$

Similarly, if data are available in the GPU and we want to process them in the FPGA we have the same response time.

V. Conclusion

This paper presented the implementation of an open source framework allowing to rapidly develop applications requiring high data rate and low latency links between an FPGA and a GPU, on a PCIe infrastructure. The achieved data rate is close to the theoretical maximum, and only requires minimal interactions with the central CPU.

The results of this work are available on the following web page: reds.heig-vd.ch/en/rad/Projets_en_realises/FPGA2.aspx. Considering the state of this project, the following steps envisioned are:

1) Support for multi-lane PCIe communication. Currently only a 1-lane endpoint is implemented, and in order to fully take advantage of GPUs that show up to 16x lanes, the FPGA design has to be adapted in that direction.
2) Full streaming support. While on the FPGA side the FIFOs allow to deal with streams of data, on the software side the GPU communication would require some enhancements in order to fully exploit streams abstraction accordingly to the CUDA specification.
3) Other devices support. The current implementation exploits an IP core supplied by Xilinx. This allowed to efficiently build a demonstrator. The design of $FPGA^2$ could then be adapted to support other devices by embedding other IP cores for the low-level PCIe management, without modifying the main architecture.

Finally, the outcome of this project opens a new way of thinking calculation in terms of heterogeneous architectures. Thanks to the DMA transfers implemented between the FPGA and the GPU, the data rate could reach up to 150MB/s and 200MB/s. Such data rates, coupled with potential multi-lane implementations, could lead to systems able to perform fast computing mixing FPGAs and GPUs in a single computer.

Acknowledgment

The authors would like to thank Jérôme Stadelmann and Yanick Saugy for their work in the early stage of the project, and the HES-SO for the funding support of this project, approved by the ISYS (http://isys.hes-so.ch) network of competences.

References

[1] R. Bittner and E. Ruf, "Direct gpu/fpga communication via pci express," in *1st International Workshop on Unconventional Cluster Architectures and Applications (UCAA 2012)*, 2012.

[2] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," in *Application Specific Processors, 2008. SASP 2008. Symposium on*, 2008, pp. 101–107.

[3] NVIDIA Corporation, *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.

[4] K. Opencl and A. Munshi, "The opencl specification version: 1.0 document revision: 48," 2009.

[5] R. K. Pingfan Meng, Matthew Jacobsen, "Fpga-gpu-cpu heterogenous architecture for real-time cardiac physiological optical mapping," in *Field-Programmable Technology (FPT), 2012 International Conference on*, 2012, pp. 37 – 42.

[6] NVIDIA, "Gpudirect," Tech. Rep., 2012, https://developer.nvidia.com/gpudirect.

[7] INFN Roma 1, "Apenet+ project," Tech. Rep., 2012, http://apegate.roma1.infn.it/mediawiki/index.php/APEnet%2B_project.

[8] Xilinx Inc., "Ml555 user guide," Tech. Rep., http://www.xilinx.com/support/documentation/boards_and_kits/ug201.pdf.

[9] R. Budruk, D. Anderson, and E. Solari, *PCI Express System Architecture*. Pearson Education, 2003.

[10] Diverse authors, "nouveau," Tech. Rep., 2012, http://nouveau.freedesktop.org/wiki/.

[11] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-class gpu resource management in the operating system," Tech. Rep., 2012, uSENIX Annual Technical Conference (USENIX ATC'12).

[12] Asus Inc., "P5q pemium user guide," Tech. Rep., http://www.cizgi.com.tr/resource/vfiles/cizgi/pms_file/73/p5qpremium_en.pdf.

[13] Xilinx Inc., "Ml505 user guide," Tech. Rep., http://www.xilinx.com/support/documentation/boards_and_kits/ug201.pdf.

[14] ——, "Endpoint block plus v1.14 user guide," Tech. Rep., 2010, http://www.xilinx.com/support/documentation/ip_documentation/pcie_blk_plus_ug341.pdf.