Connecting PCs by a custom torus network for QCD

DESY technical seminar

Zeuthen – 8 Nov 2011

Filippo Mantovani
filippo.mantovani@desy.de

HadronPhysics I3

(F. Mantovani, D. Pleiter, F. S. Schifano, H. Simma)

Overview:

- Introduction
- Architecture of custom network
- PCIe interface of the network processor
- High speed transceiver

Why a custom network?

Scalability ⇔ torus:

- ➜ cost and complexity (linear in # of procs)
- ➜ performance (in particular nearest neighbours)
- ➜ latency (in particular nearest neighbours)

Integration:

- ➜ size
- ➜ power

Tightly coupled computing nodes:

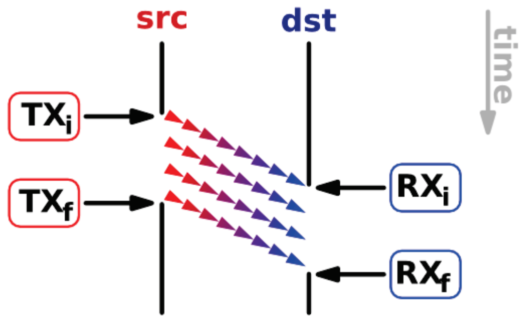
- ➜ close interface to CPU (N-P)
- ➜ light-weight protocol (N-N)

Previous examples

	unit	APEnext 2006	BG/P 2008	Cell/QPACE 2009
f_{clk}	[GHz]	0.13	0.85	3.2
# of cores	–	1	4	8
DP peak	[Gflops]	1	13.6	100
Power	[W/Gflop]	9	3	1.5
Memory bw	[GByte/s]	2	13.6	25
	[word/flop]	**1/4**	1/8	**1/32**
Network bw	[Gbyte/s]	0.67	2.55	5.5
	[word/flop]	**1/12**	1/42	**1/145**
Network latency	[ns]	$\sim$ 300	$\sim$ 800	$\sim$ 3000

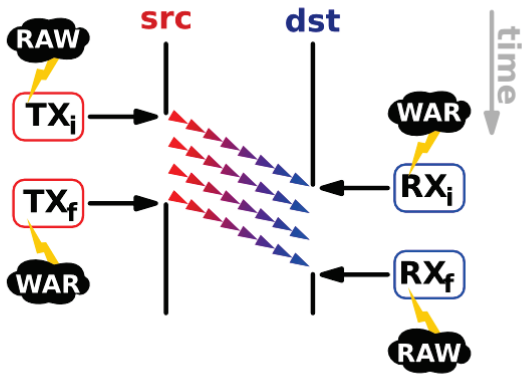
Two sided communication model

- Two sided ⇔ transport requires operations on source (TX) and on destination (RX).
- possibly non-blocking operations ⇔
  $TX = TX_i + TX_f$
  $RX = RX_i + RX_f$

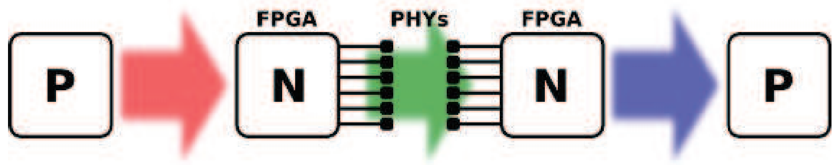


Two sided communication model

- Two sided ⇔ transport requires operations on source (TX) and on destination (RX).
- possibly non-blocking operations ⇔
  $TX = TX_i + TX_f$
  $RX = RX_i + RX_f$



Data paths



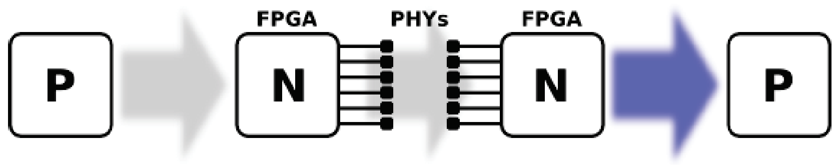
Elements involved:

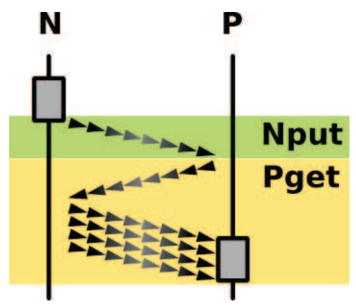
- **P:** Processor / CPU (general purpose)
- **N:** Network processor / NWP (implemented on FPGA)

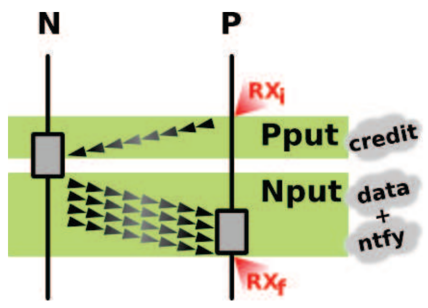
Operations:

- **put:** Initiator = source
- **get:** Initiator = destination

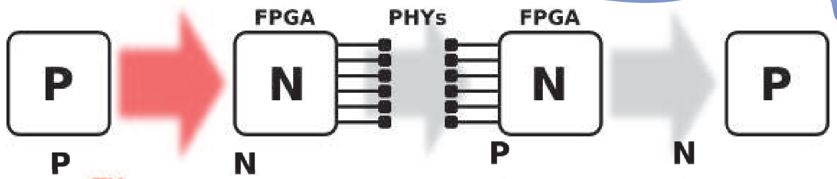
Communication model: RX

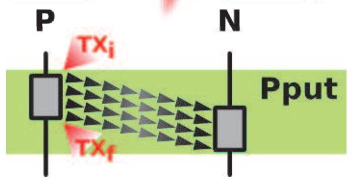


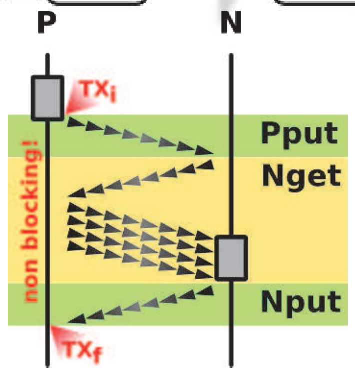




Communication model: TX







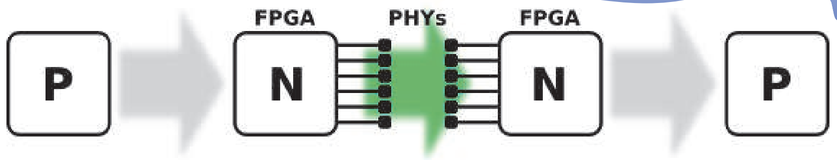
Used in QPACE

+ Low latency
– Needs backpressure handling

Non-blocking on Cell (DMA), but on Intel only PIO ⇒ blocking operation!

– Higher latency
+ Trivial backpressure handling

Communication model: Link-to-Link



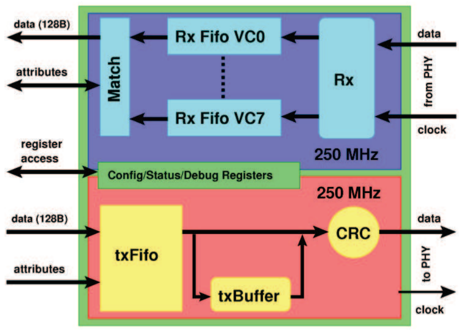
Light-weight protocol:

➜ datagram

Header	**4 B**
Payload	**128 B**
CRC	**4 B**

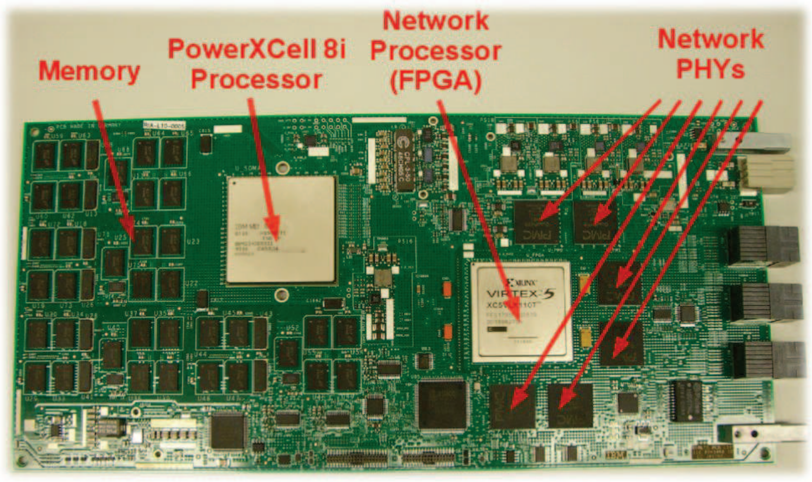
➜ commands ACK, NACK, RESTART

➜ 8 virtual channels (RX reorders packets by matching credits – data)

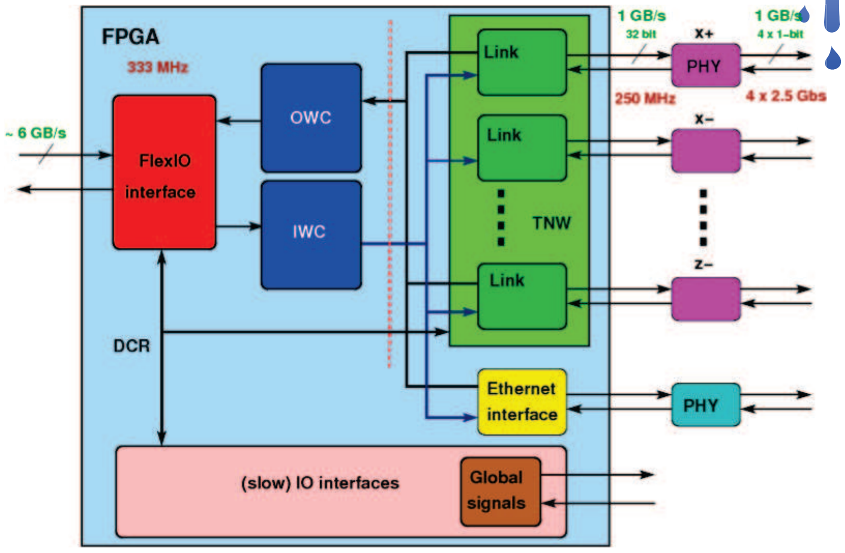


ftnw (M. Pivanti, S. F. Schifano, H. Simma) as used in QPACE

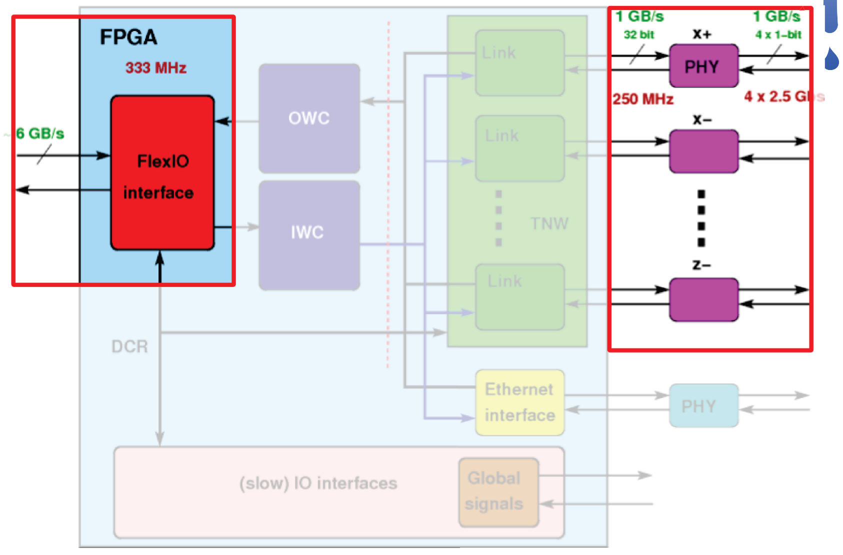
Closer view of last example in the history: QPACE



Where I did my work?



Where I did my work?



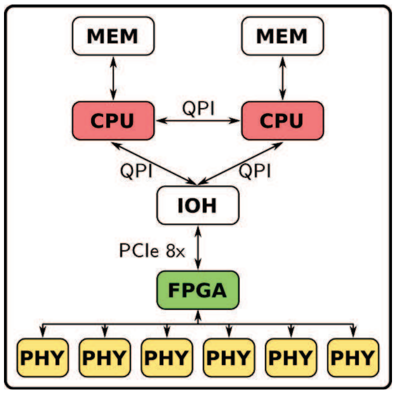
Goals

- Implement PCIe-based interface of the network processor
- Micro-benchmark of a communication model between CPU and FPGA
- Replace external PHYs by high speed transceivers within FPGA
- Test stability of the physical link with internal transceivers

Hardware setup:

Aurora (in comparison with QPACE):

- ➜ Cell → Intel Nehalem X5570@2.93 / E5540@2.53
- ➜ FlexIO → QPI + IOH + PCIe



Hardware setup:

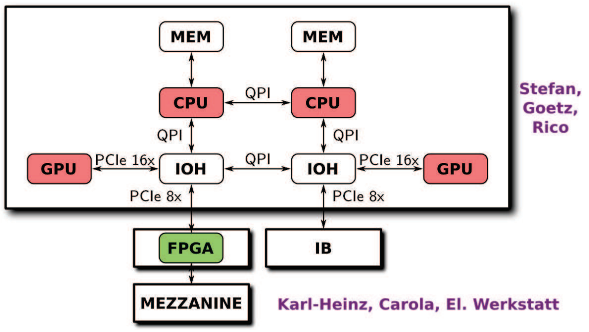
Gpu1

- ➜ 2 quad-core Intel Westmere CPUs X5667@3.06 GHz
- ➜ 2 IOH Intel 5520 (Tylersburg)
- ➜ 2 Nvidia Fermi GPU C2050 3 GB

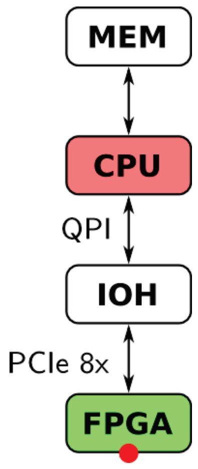
Gpu2

- ➜ 2 six-core Intel Westmere CPUs X5675@3.07 GHz
- ➜ 2 IOH Intel 5520 (Tylersburg)
- ➜ 2 Nvidia Fermi GPU C2070 6 GB

Both equiped with 1 FPGA Altera StratixIV GX230 dev-kit + 1 mezzanine card (DESY)



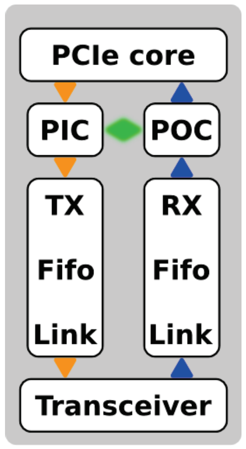
CPU – network processor setup:



We have focused on this setup:

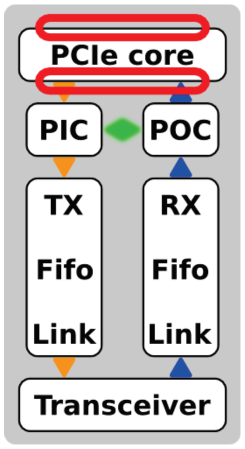
- ➜ use different hardware:
  Aurora board, gpu1-like,
- ➜ develop an FPGA firmware implementing an interface with the PCIe (on Altera Stratix IV GX230),
- ➜ write a Linux driver and a library layer allowing user applications to access the FPGA,
- ➜ measure bandwidth and latency in such setup with various communication models.

Sub-parts of the design:



Network processor
FPGA

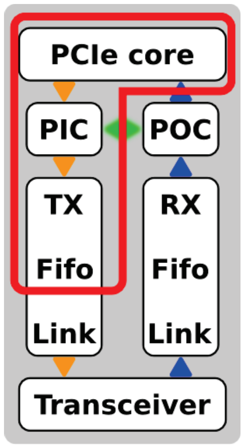
Sub-parts of the design:



Network processor
FPGA

1. PCI-express architecture in a nutshell;

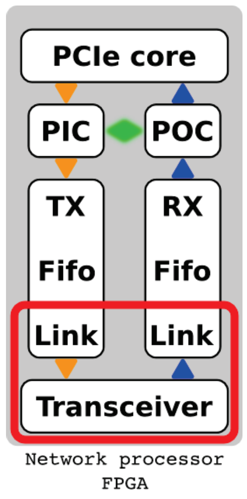
Sub-parts of the design:



Network processor
FPGA

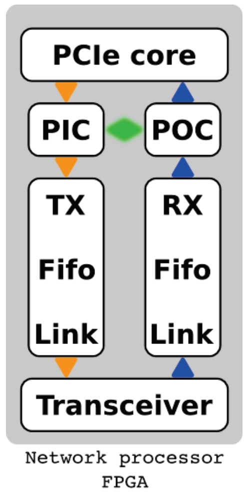
1. PCI-express architecture in a nutshell;
2. The FPGA design implementing an **Nget** engine to fetch data for transmission;

Sub-parts of the design:

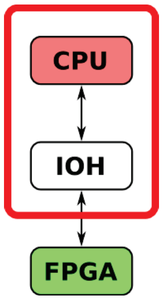


1. PCI-express architecture in a nutshell;
2. The FPGA design implementing an **Nget** engine to fetch data for transmission;
3. Transceiver;

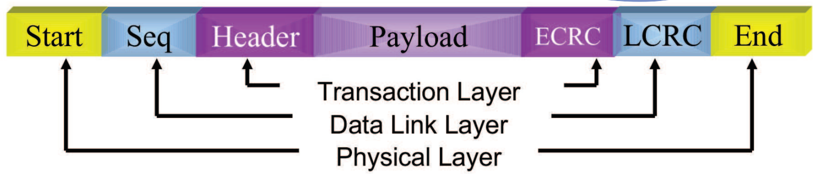
Sub-parts of the design:



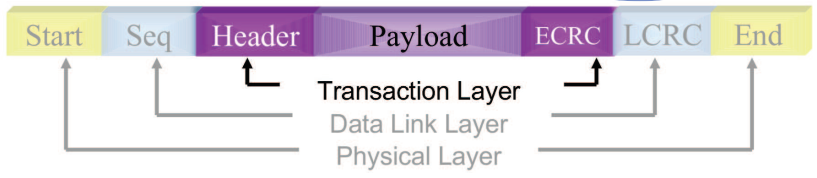
1. PCI-express architecture in a nutshell;
2. The FPGA design implementing an **Nget** engine to fetch data for transmission;
3. Transceiver;
4. The Linux driver.



PCIe architecture –1–



PCIe architecture –1–



Transaction Layer is responsible for:

- ➜ Storing negotiated and programmed configuration information
- ➜ Managing link flow control
- ➜ Enforcing ordering and Quality of Service (QoS)
- ➜ Power management control/status

Header information may include:

- ➜ Address/Routing
- ➜ Data transfer Length
- ➜ Transaction descriptor

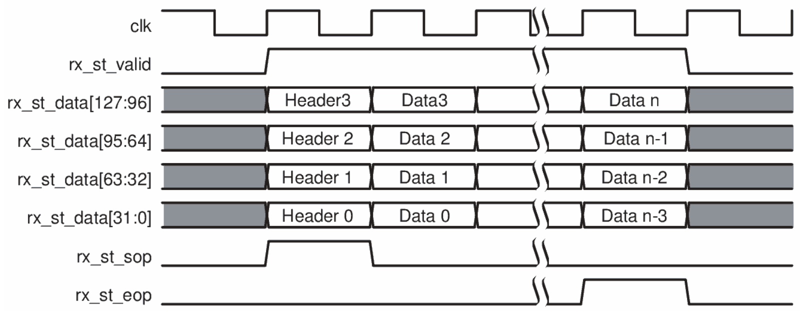
End to End CRC checking provides additional security (optional)

PCIe architecture –2–

Relevant **T**ransaction **L**ayer **P**ackets in our talk are:

- ➜ Memory Read – **MemRd**: 16 Byte header, no payload;
- ➜ Memory Write – **MemWr**: 16 Byte header, max payload 1024 Byte;
- ➜ Completion w Data – **CplD**: 16 Byte header, max payload 1024 Byte;

Interface Altera PCIe Core ⇔ **Avalon Bus**



PCIe architecture –3–

In general:

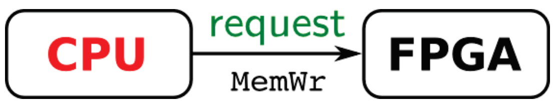
- ➜ PCIe Gen1 250 MB/s per lane;
- ➜ PCIe Gen2 500 MB/s per lane:

In our project:

- ➜ we use PCIe Gen2 8x (i.e. 8 lanes) ⇒ 4 GB/s
- ➜ Altera Avalon-ST 128bit bus (250 MHz).

The Nget engine –1–

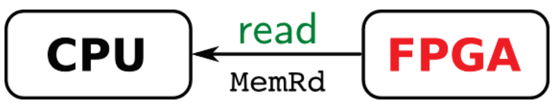
A communication scheme in which:



1. The processor triggers a read operation sending a read request to the network processor;

The Nget engine –1–

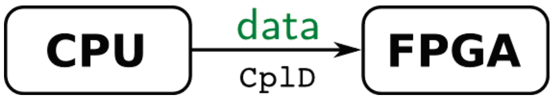
A communication scheme in which:



1. The processor triggers a read operation sending a read request to the network processor;
2. The FPGA processes requests and sends read command to the processor;

The Nget engine –1–

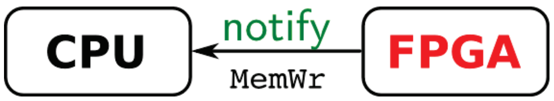
A communication scheme in which:



1. The processor triggers a read operation sending a read request to the network processor;
2. The FPGA processes requests and sends read command to the processor;
3. The processor answers with the data to send through the network;

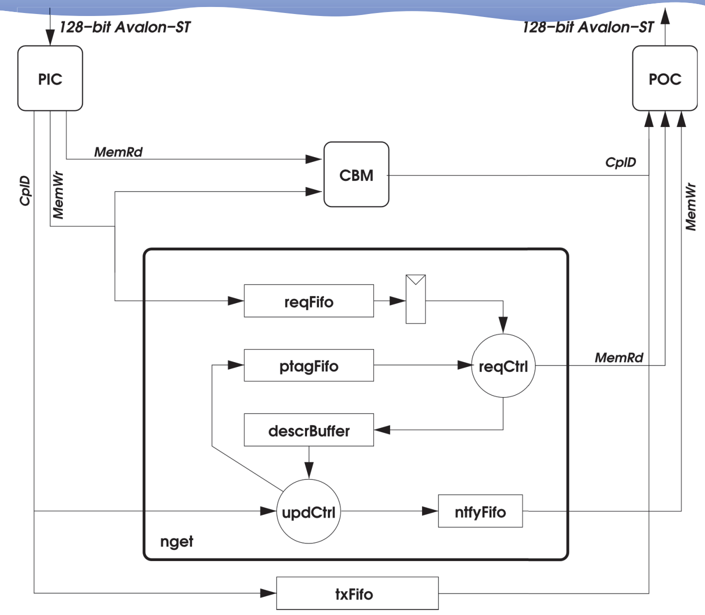
The Nget engine –1–

A communication scheme in which:



1. The processor triggers a read operation sending a read request to the network processor;
2. The FPGA processes requests and sends read command to the processor;
3. The processor answers with the data to send through the network;
4. When all data are arrived, the FPGA writes a notification message in a memory location so that the processor can detect the end of the operation.

The Nget engine –2–



The gpe driver

Driver:

- read/write via IOCTL and/or memory map;
- polling on the notify locations.

Low level library:

- init/release/reset;
- read, write (IOCTL/mm);
- nget, nget_wait;

Communication library:

- TX_i (trigger a send $P \to N$) $\to$ (nget)
- RX_i (issue credit to receive data $N \to P$) $\to$ (write)
- TX_f (test notification about completed TX) $\to$ (nget_wait)
- RX_f (test notification about completed RX) $\to$ (poll)

Driver modes

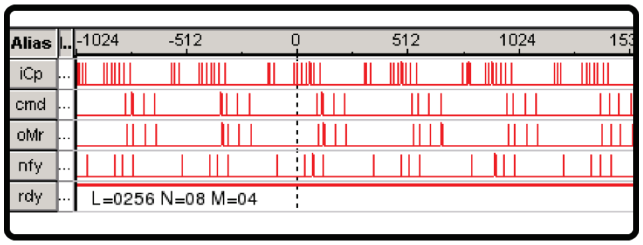
- Data buffer allocation:
  1. User-space;
  2. Kernel-space (copy required);
  3. Kernel-space (memory map);
- Write Nget requests via:
  1. IOCTL operations;
  2. Write operations on locations that are memory mapped;
- To detect the end of an **Nget** operation the network processor writes a location in CPU's main memory.
  In order to detect the notification the application triggering the Nget operation can:
  1. allocate a memory location in **user space** and poll it;
  2. leave to the driver the task to allocate memory for notification in **kernel space** and check for memory update using a standard polling method;
  3. use the Intel macro **monitor/mwait** (in kernel-space).

Algorithm of the Nget micro-benchmark

To benchmark Nget design transactions are started in a loop such that:

- Inside the main loop up to N transactions are in flight.
  $N \leq 64 \rightarrow$ `max # of PCIe tags supported by the macro`
- During each loop iteration M new transactions are started and then the algorithm waits for M outstanding transactions to complete.
- Two concurrently active transactions differ in (lnk,vc,tag).

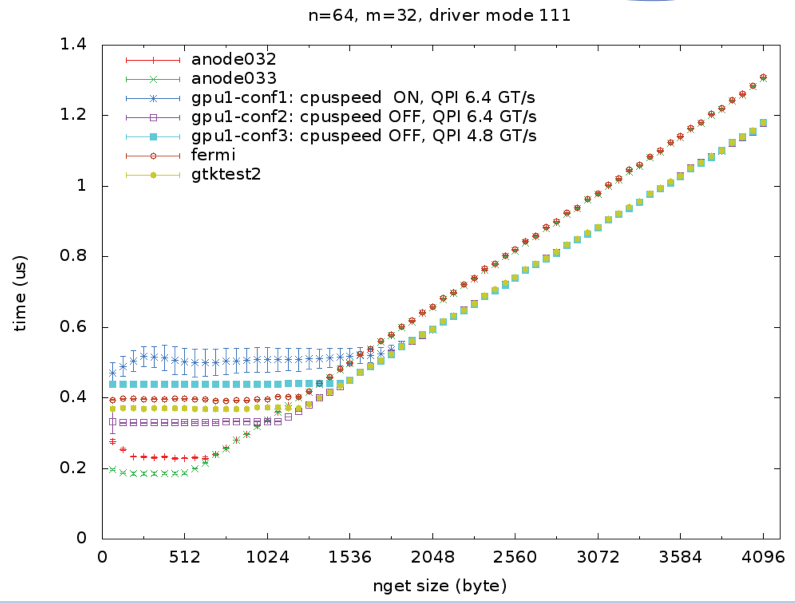
All the Nget involved are of the same size (L bytes)



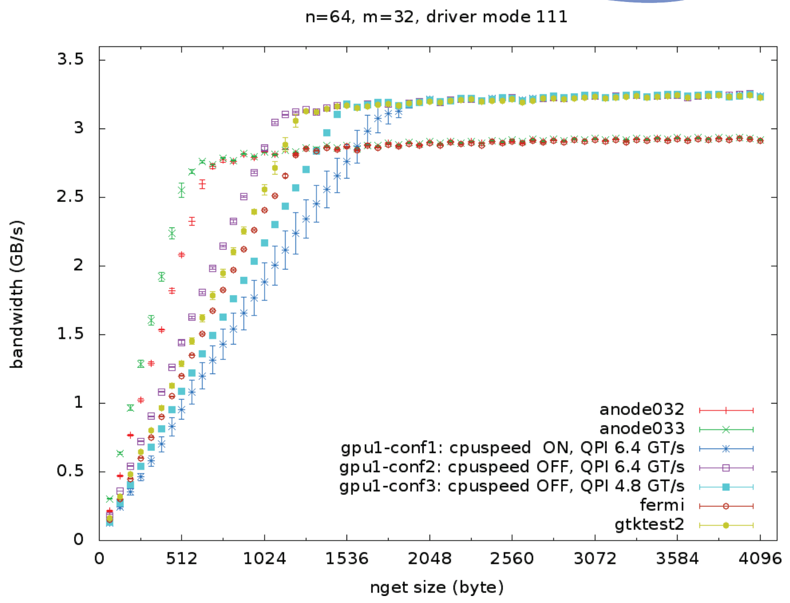
Pseudo-code of the Nget micro-benchmark

```
get time stamp #S
/* Start-up transactions */
for (i=0; i<N-M; i++) {
  update (lnk,vc,ttag)
  start nget(lnk, vc, ttag, dmabufhp[j], L);
}
/* Main loop */
for (k=0; k<K; k++) {
  for (i=0; i<M; i++) {
    update (lnk,vc,ttag)
    start nget(lnk, vc, ttag, dmabufhp[j], L);
  }
  for (i=0; i<M; i++) {
    update (lnk,vc,ttag)
    start nget_notify_wait(lnk, vc, ttag);
  }
  get time stamp #k
}
/* Drain */
for (i=0; i<N-M; i++) {
  update (lnk,vc,ttag)
  start nget_notify_wait(lnk, vc, ttag);
}
get time stamp
print time stamps #E
```

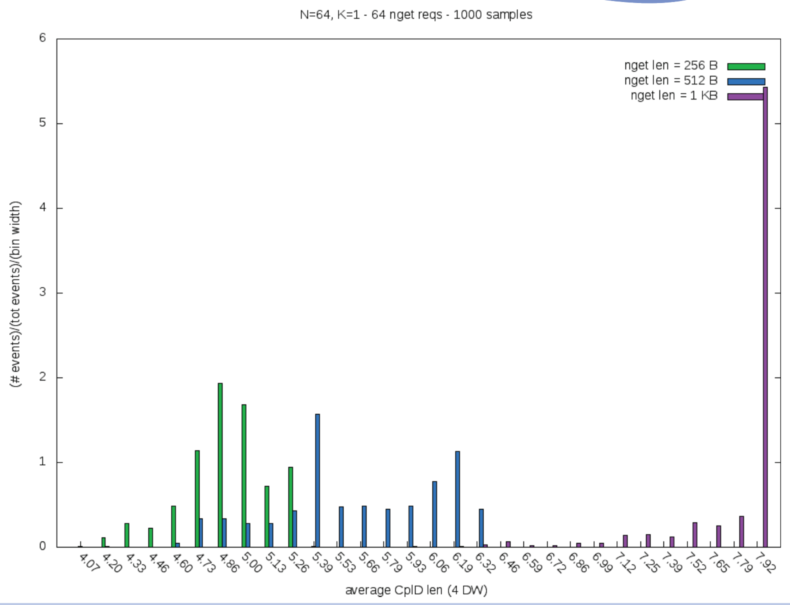
Nget duration – Machine comparison



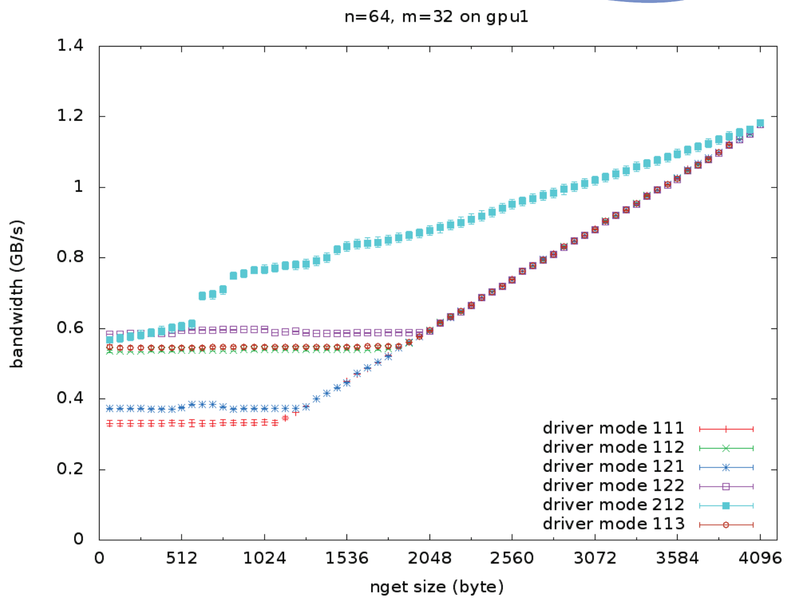
Nget bandwidth – Machine comparison



PIC occupation: packet fragmentation by IOH

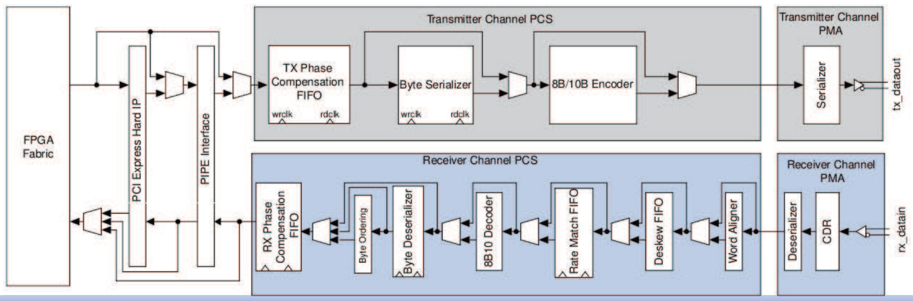


Driver modes



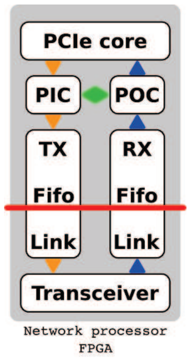
Transceivers (PIPE interface)

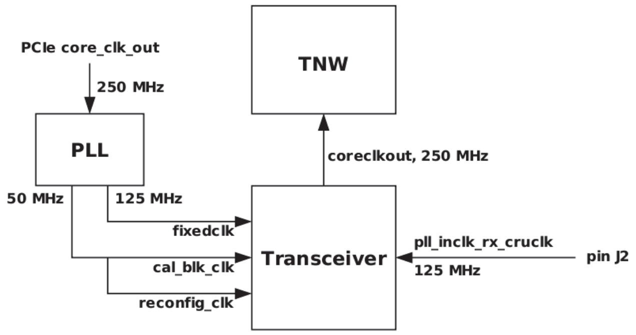
- 10 bit PMA-PCS interface width
  (between the PMA and PCS layer, i.e. after 10/8 encoding).
- Serial link data rates: 2.5 – 5 Gbps.
- Supported channel bonding x1, x4, x8 (x4).
- Automatic word aligner.
- Manual word deskew (implemented in VHDL).
- Correct byte misalignments due to byte SerDes.
- 8 or 16 bit per lane.
- Frequency 250 MHz.



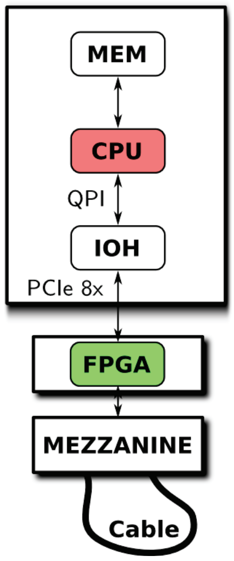
Transceivers (clock tree)

- `coreclkout`: transcv output, AL input, clock for TX inside the transmitter (250 MHz);
- `PCIe_core_clk_out`: clock from PCIe core (250 MHz);
- `cal_blk_clk`: calibration clock (10-125 MHz);
- `fixed_clk`: RX PIPE interface (125 MHz);
- `reconfig_clk`: for transcv dynamic reconfiguration (37.5-50 MHz).



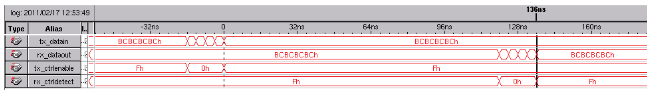


Transceivers test-bench



- Tested with GEN1 and GEN2;
- Tested with software data generation $\rightarrow$ OK for hours;
- Tested with hardware data generation $\rightarrow$ OK for hours;

One of the devkit seems to have an electrical problem still not solved.



Physical link latency (trcv – cable – trcv):
136 ns ($\simeq 1/2$ of external PHYs)

Conclusion remarks

- ➜ PCIe protocol overhead is relatively low (5% of the theoretical bandwidth).
- ➜ Data fragmentation due to IOH configuration can degrade the bandwidth.
- ➜ Micro-banchmarks of latency and bandwidth has been performed using the **Nget** engine exercise in several environments.
- ➜ **Nget** engine communication scheme has pros and cons. Pros: keep the CPU free during the read. Cons: introduce latency due to the need of request/notification for each message transmission.
- ➜ High latency to detect notification.
- ➜ Peak bandwidth $\sim 82\%$ of the theoretical bandwidth (for payloads ≥ 1 KB).
- ➜ Embedded high-speed transceiver allows to double bandwidth (GEN2), to reduce latency and to simplify hardware implementation of the physical link.

Outlook

- ➜ Tune parameters (preemphasis and VOD) of the transceiver;
- ➜ Extract EyeQ diagram from transceiver configurator block;
- ➜ Interconnect gpu1 $\Leftrightarrow$ gpu2.
- ➜ Insert the **ftnw** link modules;

N.B.: The number of transceivers in the generation of FPGA used in our tests are not enough to implement a whole network (6 high speed links for network + 1 bus PCIe)