The fastest interconnect for hundreds of CPUs, GPUs and FPGAs (make a supercomputer)

April 19, 2017

About the fastest inter-server and inter-device communication with bandwidth more than 100 Gbit/sec and latency 0.5 usec

1. Possible protocols and benefits

We all know the stack of data transfer protocols via the network - TCP/IP, which includes the transport level protocols TCP and UDP, which use IP addresses to address computers. In programming languages, these

protocols use the API: BSD Socket.

- TCP-protocol requires a connection setup and ensures delivery of a data package, and if the package is not delivered, it tells us about it.

- UDP-protocol does not require a connection setup and does not ensure delivery of a data package, but it can broadcast.

These protocols are used for data exchange in the Internet global network and in local networks.

Even if there were faster protocols, then their use in the Internet would be very difficult, because all routers in the Internet use the TCP / IP stack, the TCP and UDP transport protocols, and the IP network protocol.

However, work with Internet connections can still be speeded up by means of using IRQ-coalescing setting, by processing the TCP/IP hardware-based stack on the Ethernet-TOE-adapter, by using kernel-bypass approach Onload, Netmap/PF_RING/DPDK or by processing TCP/IP on the GPU with remapping Netmap into GPU-UVA using Intel Ethernet-card. GPU-Cores copy the data direct from the PCIe-BAR, to which the physical Ethernet-adapter buffers are mapped, with a total additional round-trip latency of 1 - 4 usec, and the GPU uses thousands of physical cores to process network packets and tens of thousands of threads for hide-latency ["Implementing an architecture for efficient network traffic on modern graphics hardware", Lazaros Koromilas, 2012].

Recall that the TCP and UDP exchange from the TCP/IP stack by means of using Ethernet equipment is called IPoE (IP over Ethernet).

However, on the local network, we can dramatically accelerate data exchange by using faster protocols and equipment:

- **Protocols**: IPoEth, IPoIB, IPoPCIe, Onload, SDP, VMA, RDS, uDAPL, VERBS, SISCI

- **Hardware** (adapter and wiring): Ethernet (TOE, RoCE, iWarp), Intel Omni-Path, Infiniband, PCIe

The difference between these protocols:

- **IPoEth, IPoIB** and **IPoPCIe -** it is the same standard TCP/IP stack with TCP and UDP protocols, but it uses equipment Ethernet (Eth), Infinband (IB) and PCI-Express (PCIe) for the exchange,

respectively.

- **Onload** - user-space implementation of TCP/IP-stack with highly optimized TCP, UDP and IP protocols using the standard API BSD Sockets, that reduces CPU utilisation and latency (app-to-app 1.7 usec), and can be used on the Solarflare Ethernet-adapter at one end of the network, even if all the rest are the conventional Etnernet equipment.

- **SDP** and **VMA -** (Socket Direct Protocols) and (Voltaire Message Accelerator) are highly optimized TCP and UDP protocols with the standard API BSD Sockets, for the equipment that allows for direct RDMA-exchange (memory to memory) at the hardware level.

- **RDS** (Reliable Datagram Sockets) **-** is a highly optimized UDP-based protocol but with guaranteed delivery of datagrams, this is especially effective on lossless devices with hardware-based reliable delivery (Infiniband, PCIe, …), RDS was included in the Linux kernel 2.6.30.

- **uDAPL** (userspace Direct Access Programming Library) **-** is a standard API for the fast data exchange on the network by means of using RDMA transport from user-space on network devices with VERBS / RDMA support.

- **VERBS -** is the fastest API (libibverbs.so) standardized by the OpenFabric alliance for using RDMA on the following devices: RoCE, iWARP, Infinband (Mellanox, QLogic, IBM, Chelsio).

- **SISCI** (Software Infrastructure Shared-Memory Cluster Interconnect) **-** is the fastest way to exchange data, requiring the API only for optimal allocation of memory, but not for accessing the data, because data exchange occurs by using ordinary assembler MOV-instructions, i.e. CPU-0 accesses by ordinary pointer to the memory of another CPU-1 located on another server via PCI Express (similarly as on a multiprocessor server via QPI / HyperTransport but without cache coherency). I.e. the physical memory of one server-0 is mapped into the physical address space of another server-1 and is re-mapped into the process user-space. You can use two ways: Programmed IO (PIO) - CPU cores by using pointer directly accesses to remote memory with the lowest latency, or (RDMA) -

DMA-Controller of PCIe-NTB copies data from remote memory to local memory with the highest bandwidth.

RDMA - (Remote Direct Memory Access) - hardware support to provide direct access to the RAM of another computer by using network equipment.
Typically, to ensure the maximum performance, application programmers, instead of VERBS and uDAPL, use a more standard API - MPI library (MPI-2 standard) that works over the uDAPL protocol, uses the uDAPL-API and has a larger built-in functionality: gather, scatter, scan, reduce, rma-operations, ...

2. Physical interfaces and benefits

Network adapters:

- **Ethernet -** the most widely used network cards for data exchange that allow to effectively build complex topologies and duplicate channels; the maximum length of the optical cable is 40 km (IEEE 802.3ba - 100GBase-ER4), currently the maximum speed is 100 Gbit/sec (12.5 GB/sec).

- **Ethernet TCP offload engine** (TOE) **-** is a similar Ethernet-adapter, but here, the TCP/IP stack processing is hardware-based and is executed in the adapter chip direct, instead of software-based processing in linux-kernel, this reduces packages delivery latencies and reduces the CPU load (e.g. Chelsio, Broadcom).

- **RoCE** (RDMA over Converged Ethernet v2) **-** is an Ethernet-adapter, in addition to TCP/IP, supporting hardware RDMA - transfer of data from the memory of one server to the memory of another server without unnecessary intermediate copies by using the UDP protocol as transport. It works via normal Ethernet-connections and Ethernet-switches / routers, but requires RoCE-adapters to be installed on all the computers using RDMA. It supports RDMA multicast (e.g. Mellanox ConnectX-3 Pro Ethernet).

- **Ethernet iWARP -** is also an Ethernet adapter, in addition to TCP/IP, supporting hardware RDMA, but using TCP and SCTP protocols as transport. It works via normal Ethernet connections and Ethernet-switches/routers and requires installation of iWARP-adapters to support RDMA (e.g. Chelsio T5/T6).

- **Infiniband -** is used mainly for data exchange in the local network for High-performance-computing clusters and super computers from top500.org, at the hardware level, it guarantees delivery of packages even for multicasting (hardware-based reliable multicast) and allows you to build complex network topologies, the maximum length of the optical cable is 200 m (Mellanox MFS1200-E200), the maximum speed for the Infiniband EDR 12x is 300 Gbit/sec (37.5 GB/sec), latency 1.5 usec (e.g. Mellanox, Intel).

- **Intel Omni-Path -** is used for HPC clusters and supercomputers top-10 from top500.org and allows you to build complex network topologies, as well as combines the advantages of Ethernet/iWARP/Infiniband, the maximum length of the optical cable is 30 meters; working at a speed of 100 Gbit/sec, it has latency of 56% lower than a similar implementation of InfiniBand, latency 1 usec.

- **PCI Express** (PCIe) **-** is a standard interface for connecting to the CPU devices: GPU, FGPA, Ethernet, Infiniband, Intel Omni-Path - the chips of each of these devices have a built-in PCIe-adapter. PCIe can also be used for connection of one CPU to another CPU via the NTB bridge or for combining multiple CPU servers into a single cluster by using multiple PCIe-Switches from PLX or IDT. The maximum length of the optical cable is 30 meters, the maximum actual throughput of PCIe 3.0 16x is 100 Gbit/sec (12.5 GB/s) with a latency of 0.5 usec. In all Intel Server (Xeon) and High-end desktop (Core) processors Intel there is 40-lanes PCI Express 3.0 with a total throughput of 256 Gbit/sec (32 GB/s) and a latency of 0.5 sec (e.g. IDT PCIe-Switch, Broadcom / PLX PCIe-Switch).

- **NVLink -** is NVidia's built-in GPU interface supporting up to 8 GPUs in a single network, and these 8 GPUs are additionally connected to the Intel CPU by PCIe. The nVidia GPU Volta features 4-Ports

NVLink 2.0 with a total performance of 1600 Gbit/sec (200 GB/s). I.e. NVLink 2.0 4-Ports are 6.5 times faster than PCIe 3.0 40-Lanes, but NVLink allows you to connect only 8 GPUs.

As PCI Express is located on the CPU chip direct and any network adapters (Ethernet, Infiniband, Omni Path, ...) are connected to the CPU via the PCIe interface, the CPUs cannot exchange the data via any of the above adapters faster than via PCI Express.

The exception is not the above-described QPI interface, which is also located on Xeon-Intel server CPU chips, provides something like a cache-coherent RDMA, where CPU-0 can access to RAM of CPU-1. But it allows to combine no more than 8 CPUs into a single server - i.e. the system does not look like a cluster of multiple servers, but as a single multi-core server (ccNUMA). Typically, server prices with QPI connections grow exponentially, depending on the number of CPUs.

The PCI Express has the maximum throughput and the minimum latencies. It is integrated into most devices and can combine hundreds of CPUs, GPUs, FPGAs, Ethernet, Infiniband and other devices supporting PCI Express into the same network by using PCIe-Switches. PCIe is the fastest solution for building HPC clusters from hundreds of different devices.

All network devices, except for the standard Ethernet, allow using any of the protocols listed above, for example:

Computer1 -> uDAPL -> iWARP -> Ethernet -> iWARP -> uDAPL -> Computer2

You can read about each of the protocols and network adapters in Wikipedia, and you can find them by their names. There are other less common protocols and interfaces not described here.

To run the program using the Onload, SDP and VMA protocols instead of TCP and UDP, it is not necessary to change the source code of the program. The program (binary executable program) compiled for work with TCP and UDP can be made to work by using Onload, SDP or VMA protocols. To do this, run the program as follows:

LD_PRELOAD = libonload.so program.bin,

LD_PRELOAD = libsdp.so program.bin or LD_PRELOAD = libvma.so program.bin

OpenOnload retains the TCP/IP protocol so can be used at single end, even if on other computers on the network, the usual Ethernet-adapters are installed and the usual Linux-kernel stack TCP/IP is used. However, for SDP or VMA, it should be taken into account that all the parties involved in data exchange (computers) should work via the same protocols - at both ends only SDP or only VMA. I.e. the program

trying to establish a connection via TCP will not be able to establish a connection with the program that accepts connections via SDP.

General recommendations:

(These recommendations can be used to process client connections from the Internet, without changing the source code of your program, and without changing the hardware – but it is recommended to use Solarflare Ethernet 40 Gbit to use OpenOnload – TCP offload engine)

1. Install 2 utilities by using: aptitude install / sudo apt-get install
   - **numactl**: http://linuxcommand.org/man_pages/numactl8.html
   - **ethtool**: http://www.linuxcommand.org/man_pages/ethtool8.html

   Then check that these utilities are installed:
   - numactl --hardware
   - ethtool -S eth0
   - ethtool -k eth0

   Also, you can get other useful information:
   - ifconfig
   - lspci -t -n -D -v
   - cat /proc/interrupts
   - cat /proc/net/softnet_stat
   - perf top

2. If you have a server with 2 or more CPUs on the same motherboard (i.e. NUMA), then connect the network adapters to the PCIe of different CPUs evenly - i.e. each CPU should has the same number of network cards.

3. Run a individual instance of your application for each CPU (NUMA-node) as follows, so that each instance will be rigidly bound to a specific NUMA-node, example for 2 x Xeon CPU Server:

   - service irqbalance stop (or set ENABLED=0 in the file /etc/default/irqbalance)
   - numactl --localalloc --cpunodebind=0 ./program.bin
   - numactl --localalloc --cpunodebind=1 ./program.bin

   These instances can communicate with each other via sockets by a local connection.

However, If you necessarily want to use only one instance of your program, then run your program as follows: numactl --interleave=all ./program.bin

4. Use RSS (Receive Side Scaling) – hardware based – usually enabled by default

   - For Intel use: echo "options igb RSS=0,0" >>/etc/modprobe.d/igb.conf
   - For Solarflare: echo "options sfc rss_cpus=cores" >>/etc/modprobe.d/scf.conf

   Then reload driver:
   - rmmod sfc
   - modprobe sfc

5. Choose the optimal number of queues of network card up to 4 – 32, but not more than the number of Cores per one CPU, example for 2 x Ethernet cards:

   - ethtool -L eth0 combined 4
   - ethtool -L eth1 combined 4

   (The most efficient high-rate configuration is likely the one with the smallest number of receive queues where no receive queue overflows due to a saturated CPU)

6. Bind interrupts of each network card to the CPU (NUMA-node) to which it is connected:

   - service irqbalance stop (or set ENABLED=0 in the file /etc/default/irqbalance)
   - echo 1 >/proc/irq/[eth0-irq-num]/node
   - echo 2 >/proc/irq/[eth1-irq-num]/node
   - echo 1 >/sys/class/net/eth0/device/numa_node
   - echo 2 >/sys/class/net/eth1/device/numa_node

7. Distribute interrupts of queues of network card to the different Cores of CPU to which this network card is attached, by using bit-mask in HEX (1,2,4,8,10,20,40,80,100,200,…):

   - echo 1 >/proc/irq/[eth0-queue0-irq-num]/smp_affinity
   - echo 2 >/proc/irq/[eth0-queue1-irq-num]/smp_affinity
   - echo 4 >/proc/irq/[eth0-queue2-irq-num]/smp_affinity
   - echo 8 >/proc/irq/[eth0-queue3-irq-num]/smp_affinity

- echo 1 >/proc/irq/[eth1-queue0-irq-num]/smp_affinity
- echo 2 >/proc/irq/[eth1-queue1-irq-num]/smp_affinity
- echo 4 >/proc/irq/[eth1-queue2-irq-num]/smp_affinity
- echo 8 >/proc/irq/[eth1-queue3-irq-num]/smp_affinity

Or by using – set_irq_affinity.sh: https://gist.github.com/SaveTheRbtz/8875474

8. Enable RPS (Receive Packet Steering) – software based – it can use any number of CPU-Cores in contrast to the described above RSS, which can usually only use 8 Cores. Also RPS allows us to enable following RFS.

   Note: RPS requires a kernel compiled with the CONFIG_RPS kconfig symbol (on by default for SMP).

   - echo 1 > /sys/class/net/eth0/queues/rx-0/rps_cpus
   - echo 2 > /sys/class/net/eth0/queues/rx-1/rps_cpus
   - echo 4 > /sys/class/net/eth0/queues/rx-2/rps_cpus
   - echo 8 > /sys/class/net/eth0/queues/rx-3/rps_cpus

   - echo 1 > /sys/class/net/eth1/queues/rx-0/rps_cpus
   - echo 2 > /sys/class/net/eth1/queues/rx-1/rps_cpus
   - echo 4 > /sys/class/net/eth1/queues/rx-2/rps_cpus
   - echo 8 > /sys/class/net/eth1/queues/rx-3/rps_cpus

   RedHat 7 – 7.3.6. Configuring Receive Packet Steering (RPS): https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Performance_Tuning_Guide/sect-Red_Hat_Enterprise_Linux-Performance_Tuning_Guide-Networking-Configuration_tools.html

9. Enable RFS (Receive Flow Steering) – software based – then TCP/IP-stack in kernel-space will be executed on the same CPU-Core as thread of your Application in user-space that processes the same packet - good cache locality.

   - ethtool -K eth0 ntuple on
   - ethtool -K eth1 ntuple on
   - sudo sysctl -w net.core.rps_sock_flow_entries=32768
   - echo 32768 > /proc/sys/net/core/rps_sock_flow_entries
   - echo 4096 > /sys/class/net/eth0/queues/rx-0/rps_flow_cnt
   - echo 4096 > /sys/class/net/eth0/queues/rx-1/rps_flow_cnt

- echo 4096 > /sys/class/net/eth0/queues/rx-2/rps_flow_cnt
- echo 4096 > /sys/class/net/eth0/queues/rx-3/rps_flow_cnt

- echo 4096 > /sys/class/net/eth1/queues/rx-0/rps_flow_cnt
- echo 4096 > /sys/class/net/eth1/queues/rx-1/rps_flow_cnt
- echo 4096 > /sys/class/net/eth1/queues/rx-2/rps_flow_cnt
- echo 4096 > /sys/class/net/eth1/queues/rx-3/rps_flow_cnt

(Set the value of this file to the value of rps_sock_flow_entries divided by N, where N is the number of receive queues on a device)
RedHat 7 – 7.3.7. Configuring Receive Flow Steering (RFS): https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Performance_Tuning_Guide/sect-Red_Hat_Enterprise_Linux-Performance_Tuning_Guide-Networking-Configuration_tools.html

10. Enable XPS (Transmit Packet Steering) – software based – the same as RPS, but XPS used for transmitting packets instead of receiving. Can use any number of CPU-Cores to process packets on the TCP/IP-stack in kernel-space.

    Note: XPS requires a kernel compiled with the CONFIG_XPS kconfig symbol (on by default for SMP).

    - echo 1 > /sys/class/net/eth0/queues/tx-0/xps_cpus
    - echo 2 > /sys/class/net/eth0/queues/tx-1/xps_cpus
    - echo 4 > /sys/class/net/eth0/queues/tx-2/xps_cpus
    - echo 8 > /sys/class/net/eth0/queues/tx-3/xps_cpus

    - echo 1 > /sys/class/net/eth1/queues/tx-0/xps_cpus
    - echo 2 > /sys/class/net/eth1/queues/tx-1/xps_cpus
    - echo 4 > /sys/class/net/eth1/queues/tx-2/xps_cpus
    - echo 8 > /sys/class/net/eth1/queues/tx-3/xps_cpus

11. If interrupts use too much CPU time and if more bandwidth (GB/sec) is required, but the minimal latency (usec) is not critical, then use Interrupt coalescing – increase up to 100 or 1000.
    (However, if you want to achieve low latency, then set these values to 0)
    - ethtool -c eth0
    - ethtool -c eth1
    - ethtool -C eth0 rx-usecs 100

- ethtool -C eth1 rx-usecs 100
- ethtool -C eth0 tx-usecs 100
- ethtool -C eth1 tx-usecs 100
- ethtool -C eth0 rx-frames 100
- ethtool -C eth1 rx-frames 100

12. If a lot of packets are lost – this you can see using the command:
    - ethtool -S eth0

    Then see the size of the current and maximum rx/tx buffers of your network card:
    - ethtool -g eth0

    Then try to increase current value to maximum by using:
    - ethtool -G eth0 rx 4096 tx 4096

    Details: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-network-common-queue-issues.html

13. Disable SSR (slow-start restart) for long-lived TCP connections:

    - sudo sysctl -w net.ipv4.tcp_slow_start_after_idle=0

14. Change CPU governor to 'performance' for all CPUs/cores, run as root:

    for CPUFREQ in /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor; do [-f $CPUFREQ] || continue; echo -n performance > $CPUFREQ; done

    service cpuspeed stop

    More details: https://www.servernoobs.com/avoiding-cpu-speed-scaling-in-modern-linux-distributions-running-cpu-at-full-speed-tips/

15. Use Accelerated RFS/SARFS – hardware based RFS – to bind network-packets to network-stack on that CPU-Core on which this packet will be processed – this is cache-friendly:

- Enable A-RFS if you are using Mellanox with mlx4-driver (or Intel Ethernet adapter if it supports A-RFS). Accelerated RFS is only available if the kernel is complied with the kconfig symbol CONFIG_RFS_ACCEL enabled.

- Enable SARFS (Solarflare Accelerated Receive Flow) if you are using Solarflare Ethernet adapters – supported since Solarflare Linux network driver v4.1.0.6734 Create file: /etc/modprobe.d/sfc.conf
  Then add this content to this file:

  sxps_enabled
  sarfs_table_size
  sarfs_global_holdoff_ms
  sarfs_sample_rate

  3.18 Solarflare Accelerated RFS (SARFS) – page 81:
  https://support.solarflare.com/index.php/component/cognidox/?file=SF-103837-CD-19_Solarflare_Server_Adapter_User_s_Guide.pdf&task=download&format=raw&id=165
  Also, see "A Guide to Optimising Memcache Performance on SolarFlare":
  http://10gigabitethernet.typepad.com/files/sf-110694-tc.pdf

16. Use processing TCP/IP-stack in the user-space (kernel bypass) if possible – this will: increase throughput, reduce latency (-10 usec), and reduce CPU usage.

    If used Solarflare Ethernet card then use OpenOnload – run your program as follows without recompile:
    - LD_PRELOAD=libonload.so numactl --localalloc --cpunodebind=0 ./program.bin
    - LD_PRELOAD=libonload.so numactl --localalloc --cpunodebind=1 ./program.bin

17. Try to use fast memory allocator tcmalloc or jemalloc: https://github.com/jemalloc/jemalloc/wiki/Getting-Started

    - LD_PRELOAD=libonload.so:jemalloc.so.1 numactl --localalloc --cpunodebind=0 ./program.bin
    - LD_PRELOAD=libonload.so:jemalloc.so.1 numactl --localalloc --cpunodebind=1 ./program.bin

18. Find the best combination for performance of your system by switching on/off two parameters in the BIOS: 1. C-sleep state (C3 or greater), 2. Hyper Threading.

19. Increase socket limits:

    - cat /proc/sys/fs/file-max
    - echo "100000" > /proc/sys/fs/file-max

(or equivalently: sysctl -w fs.file-max=100000)

20. Try to check the effect of offload engine parameters on performance – this offloads the execution part of the algorithms to the NIC:

    - Look at the parameters: ethtool -k eth0
    - And try to turn them on or off: ethtool -K eth0 rx on tx on tso on gso on lso on lro on

21. Try to check the effect of these parameters on performance: https://stackoverflow.com/a/3923785/1558037

    - sudo sysctl net.core.somaxconn=16384
    - ifconfig eth0 txqueuelen 5000
    - sudo sysctl net.core.netdev_max_backlog=2000
    - sudo sysctl net.ipv4.tcp_max_syn_backlog=4096

Also read:

- Scaling in the Linux Networking Stack: https://www.kernel.org/doc/Documentation/networking/scaling.txt
- Monitoring and Tuning the Linux Networking Stack: Receiving Data: https://blog.packagecloud.io/eng/2016/06/22/monitoring-tuning-linux-networking-stack-receiving-data/
- Monitoring and Tuning the Linux Networking Stack: Sending Data: https://blog.packagecloud.io/eng/2017/02/06/monitoring-tuning-linux-networking-stack-sending-data/
- How to achieve low latency with 10Gbps Ethernet: https://blog.cloudflare.com/how-to-receive-a-million-packets/

Recommendations for Application:

1. Use thread-pool: i.e. don't create and destroy threads for each connection, but create so much threads as the number of CPU-Cores, evenly distribute connections over threads and bind threads to CPU-Cores by using: sched_setaffinity(), pthread_setaffinity_np()

2. Use memory-pool: i.e. repeatedly re-use pre-allocated memory areas to avoid the overhead of extra memory allocation.

3. Use platform-specific optimal demultiplexing approaches: epoll (Linux), kqueue (Open/Net/FreeBSD, MacOS, iOS), IOCP (Windows), IOCP/POLL_SET (AIX), /dev/poll (Solaris)

4. If your application is working with a large number of short connections, then in each thread use individual own epoll_wait() processing cycle and individual own acceptor-socket with the same <ip:port> and enabled SO_REUSEADDR and SO_REUSEPORT (Linux kernel 3.9 or later), as it is used in Ngnix: https://www.nginx.com/blog/socket-sharding-nginx-release-1-9-1/

```
epoll_event event, ret_events[N];
event.data.fd = socket(AF_INET, SOCK_STREAM, 0);
bind(event.data.fd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
event.events = EPOLLIN | EPOLLET;
int enable = 1; setsockopt(event.data.fd, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int));
enable = 1; setsockopt(event.data.fd, SOL_SOCKET, SO_REUSEPORT, &enable, sizeof(int));
fcntl(event.data.fd, F_SETFL, fcntl(sfd, F_GETFL, 0) | O_NONBLOCK);
listen(event.data.fd, SOMAXCONN);
int efd = epoll_create1(EPOLL_CLOEXEC);
epoll_ctl(efd, EPOLL_CTL_ADD, event.data.fd, &event);
while(1) { int events_num = epoll_wait (efd, ret_events, N, -1); for(int i=0; i<events_num; ++i){...} }
// https://linux.die.net/man/2/epoll_wait
// call stack with URLs: epoll_wait() ->kernel-> SYSCALL_DEFINE4() -> ep_poll()

// or polling for UDP: while(1) { int r = recvmsg(socket_desc, msg, MSG_DONTWAIT); if (r > 0) {...} }
```

5. Use polling: for low-latency and a small number of long connections, then in each thread use individual own poll() processing cycle and individual own acceptor-socket with the same <ip:port> and enabled SO_REUSEADDR and SO_REUSEPORT.

```
pollfd pfd[N]; while(1) { int events_num = poll(pfd, N, -1); if(events_num > 0) { read(...); }  } //
https://linux.die.net/man/2/poll
// call stack with URLs: poll() ->kernel-> do_sys_poll() -> do_poll()
// call stack with URLs: select() ->kernel-> core_sys_select() -> do_select()
```

6. If your application is working with a large number of long connections, then accept connections using the acceptor-socket in only one thread, and then transfer batches of the received connection sockets to

other threads using a thread-safe 1P-MC queues (wait-free or write-contention-free). In each thread use its own epoll_wait() processing cycle.

7. Use work stealing: for high throughput and long connections, if the CPU-Cores usage becomes uneven over time. I.e. if there is too much work for the one of thread, the thread sends extra work to the thread-safe queue. Use individual 1P-MC queues for each thread – so busy-thread pushes extra works to its own queue with relaxed-ordering using only Write-operations, but free-threads which have too little work concurrently get works using RMW-operations with acq-rel-ordering from any of these queues). Work – is a connection socket descriptor or something else.

8. Use batching for sending data: i.e. if required high throughput, then prepare large block of data in your buffer and then send this entire block to the socket

9. Use batching for receiving and sending data over UDP: i.e. use recvmmsg() and sendmmsg() instead of recv() and send()

10. To filter traffic use libs: PF_RING or (Netmap or DPDK on Intel-Eth only) – this will give the highest possible performance. Comparison: https://www.net.in.tum.de/fileadmin/bibtex/publications/theses/2014-gallenmueller-high-speed-packet-processing.pdf

    Alternatively, use RAW-sockets with: batching, zero-copy and fan-out.

```
int packet_socket = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL)); // raw-sockets

int version = TPACKET_V3;  // read/poll per-block instead of per-packet
setsockopt(packet_socket, SOL_PACKET, PACKET_VERSION, &version, sizeof(version));

// map socket buffer from kernel to user-space for zero-copy
struct tpacket_req3 req;
setsockopt(packet_socket, SOL_PACKET , PACKET_RX_RING , (void*)&req , sizeof(req));
mapped_buffer = (uint8_t*)mmap(NULL, req.tp_block_size * req.tp_block_nr, PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_LOCKED, packet_socket, 0);

// to avoid locks when used one socket from many threads
int fanout_arg = (fanout_group_id | (PACKET_FANOUT_CPU << 16));
setsockopt(packet_socket, SOL_PACKET, PACKET_FANOUT, &fanout_arg, sizeof(fanout_arg));
```

11. Accept connections to different listening IP addresses on different NUMA-nodes – i.e. for each process use only that Ethernet-card which connected to this NUMA-node on which executed this

process

Recommendations for LAN:

1. Enable Jumbo-frames – i.e. increase MTU:

- ifconfig eth0 mtu 32000
- ifconfig eth1 mtu 32000
- ip route flush cache
- ping -M do -c 4 -s 31972 192.168.0.1 (try to ping each server in LAN)

2. If you are using Ethernet Mellanox adapter, then use VMA

- LD_PRELOAD=libvma.so numactl --localalloc --cpunodebind=0 ./program.bin
- LD_PRELOAD=libvma.so numactl --localalloc --cpunodebind=1 ./program.bin

(If you are using iWarp offload adapter or RoCE adapter, then try to use SDP)

3. Try to use Infininband (switches, wires and adapters): Mellanox or Intel
(Use preferably VMA for Mellanox, or SDP for Intel, otherwise use IPoIB)

For SDP (Sockets Direct Protocol):

- modprobe ib_sdp
- LD_PRELOAD=libsdp.so numactl --localalloc --cpunodebind=0 ./program.bin
- LD_PRELOAD=libsdp.so numactl --localalloc --cpunodebind=1 ./program.bin

4. Try to connect servers direct via PCI Express by using IDT or PLX Switches with NTB, virtual NIC and TWC, for example: PEX9797

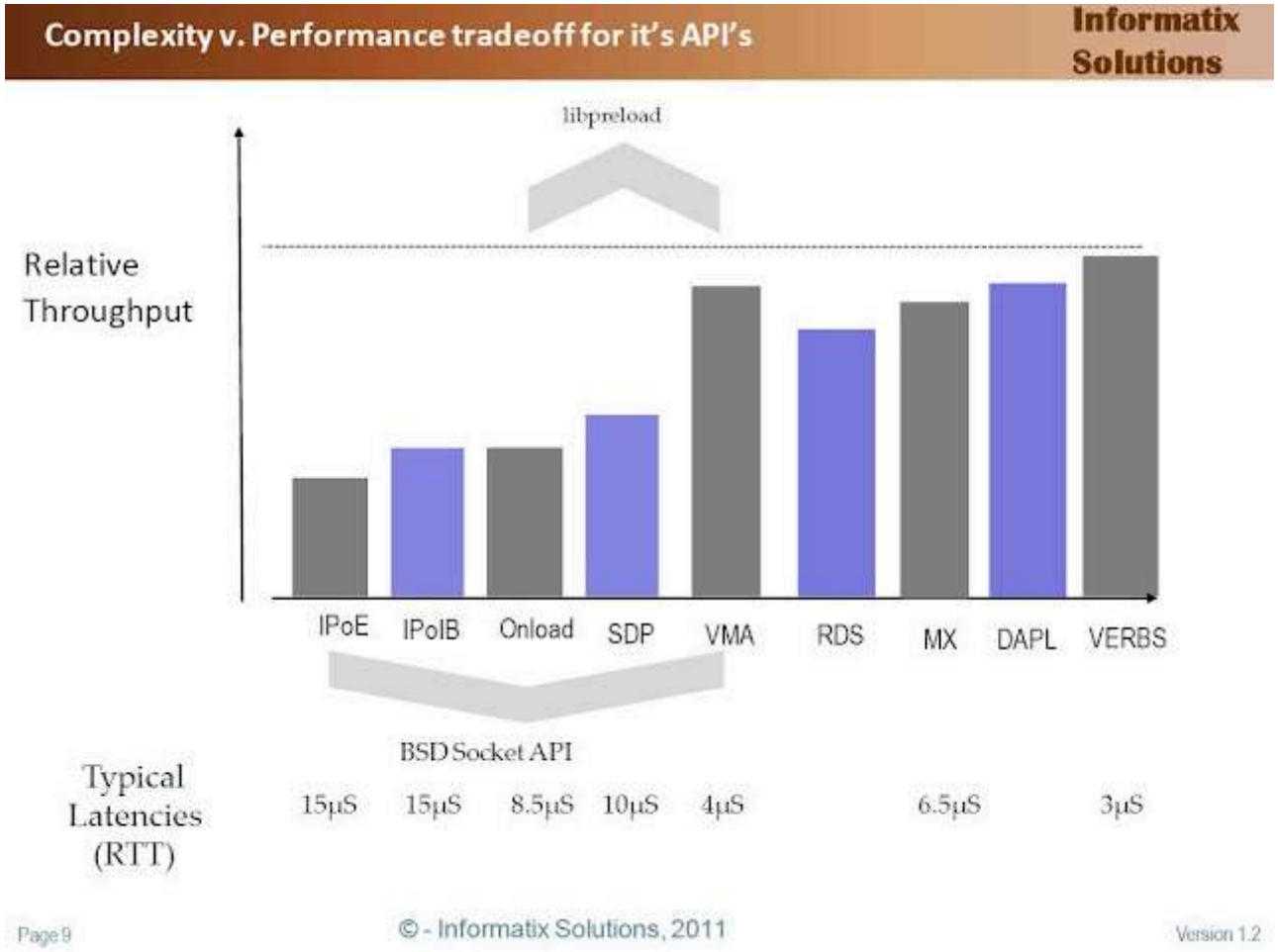
I.e. use PEX9700 series or more modern PCIe-switches which support virtual Ethernet NICs, NIC DMA and Tunneled Window Connection (TWC): https://docs.broadcom.com/docs/AV00-0327EN

5. If you are using network adapters that support RDMA (iWarp, RoCE, Infiniband, PCIe NTB-Switches) then try to use API uDAPL, MPI (MPI-2 over uDAPL) or SISCI in your program instead of BSD Sockets

Screenshots and Quotes:

Next, we give a lot of screenshots from various articles and links to them - the screenshots show page numbers on the basis of which they are easy to find in the articles. We'll show which problems are solved by using PCIe interface, DAPL protocol and MPI software interface.

We'll show how the throughput and latency of various network protocols are related in general, and which of them use the standard API - BSD Socket:



https://www.informatix-sol.com/docs/TCP_bypass_overview.pdf

Openonload retains the TCP/IP protocol so can be used single ended.

This can be used in any of three modes each with decreasing latency but with increasing API complexity:

- The simplest is to take an existing TCP/IP socket based program, it can even be in binary format and preload onload before it is started.
- The next is to use TCPdirect, which uses a simplified socket API referred to as zsockets. This requires refactoring of existing Linux socket 'C' code.
- The best latency however, for Onload, is to re-write your application to use their EF_VI API.

Kernel based TCP/IP has it's limitations, you can typically half its contribution to latency by adopting a User space implementation such as OpenOnload. This can be used in any of three modes each with decreasing latency but with increasing API complexity. The simplest is to take an existing TCP/IP socket based program, it can even be in binary format and preload onload before it is started. The next is to use TCPdirect, which uses a simplified socket API referred to as zsockets. This requires refactoring of existing Linux socket 'C' code. The best latency however, for Onload, is to re-write your application to use their EF_VI API. This is asynchronous, using completion events so it usually requires a complete rewrite of the send/receive modukes.

Where you have control of both ends of the wire then lowest latency and jitter is obtained by bypassing TCP altogether. There are three main candidates for this - InfiniBand, RoCE and iWARP. These all share a common set of API's known as the VERB's so it's possible to develop applications that will run on any of them. As with Openonload, SDP and Mellanox's VMA all preload to accelerate an existing TCP/IP socket program. Openonload retains the TCP/IP protocol so can be used single ended. SDP and VMA both map to VERBS so must be deployed on both ends of the wire. Best latency with Openonload is achieved by receive polling, this sacrifices a core just to receieve packets on this socket but does avoid the kernel wakeup delay of the user thread. VERB based programs can run be run on Ethernet, InfiniBand or OmniPath. Verb programs over Ethernet use either RoCE or iWARP. RoCE relies on PFC's to limit senders and the non-drop queuing for any RDMA Ethertype packets queued in the switches to provide the underlying reliability, whilst iWARP uses TCP (implemented with offload engines). Unfortunately, whilst the DCB standard includes the mechanisms to enable this, the current generation of Ethernet switches typically only enable dropless behavior for the FCoE Ethertype. Whiles RoCE programs may appear to work any drops may be undetected and result in data corruption. Large scale RDMA Ethernet deployments also need L2 mesh support to replace Spanning tree. There are a number of proprietary approaches appearing to solve this, whilst the DCB group are focusing on TRIL which we should see emerge during 2012.

TOE (TCP/IP Offload Engine) Ethernet Adapter:
Solarflare Flareon SFN7002F Dual-Port 10GbE PCIe 3.0 Server I/O Adapter - Part ID: SFN7002F

- Hardware Offloads: LSO, LRO, GSO; IPv4/IPv6; TCP, UDP checksums

Solarflare Flareon SFN7002F Dual-Port 10GbE PCIe 3.0 Server I/O Adapter - Part ID: SFN7002F

SKU: SFN7002F
Manufacturer: Solarflare

Solarflare Flareon SFN7002F Dual-Port 10GbE PCIe 3.0 Server I/O Adapter More details...

Price: $395.00

1 Add To Cart

Add to Wishlist Tell a Friend

Share this product:

Tweet G+1 0

Product Details

Flareon SFN7002F Dual-Port 10GbE PCIe 3.0 Server I/O Adapter

Flareon SFN7002F provides the best overall performance with low CPU utilization for 10GbE networking. With a PCIe 3.0 host interface, an all new internal data-path micro-architecture, the addition of a hardware switch fabric, and featuring a full set of stateless offloads, SFN7002F reduces CPU processing overhead for the most demanding network tasks in data center, enterprise, cloud, and network-attached storage networks. SFN7002F also accelerates database, social networking, email and other storage-intensive applications with high bandwidth and low latency network access to SAN and NAS environments.

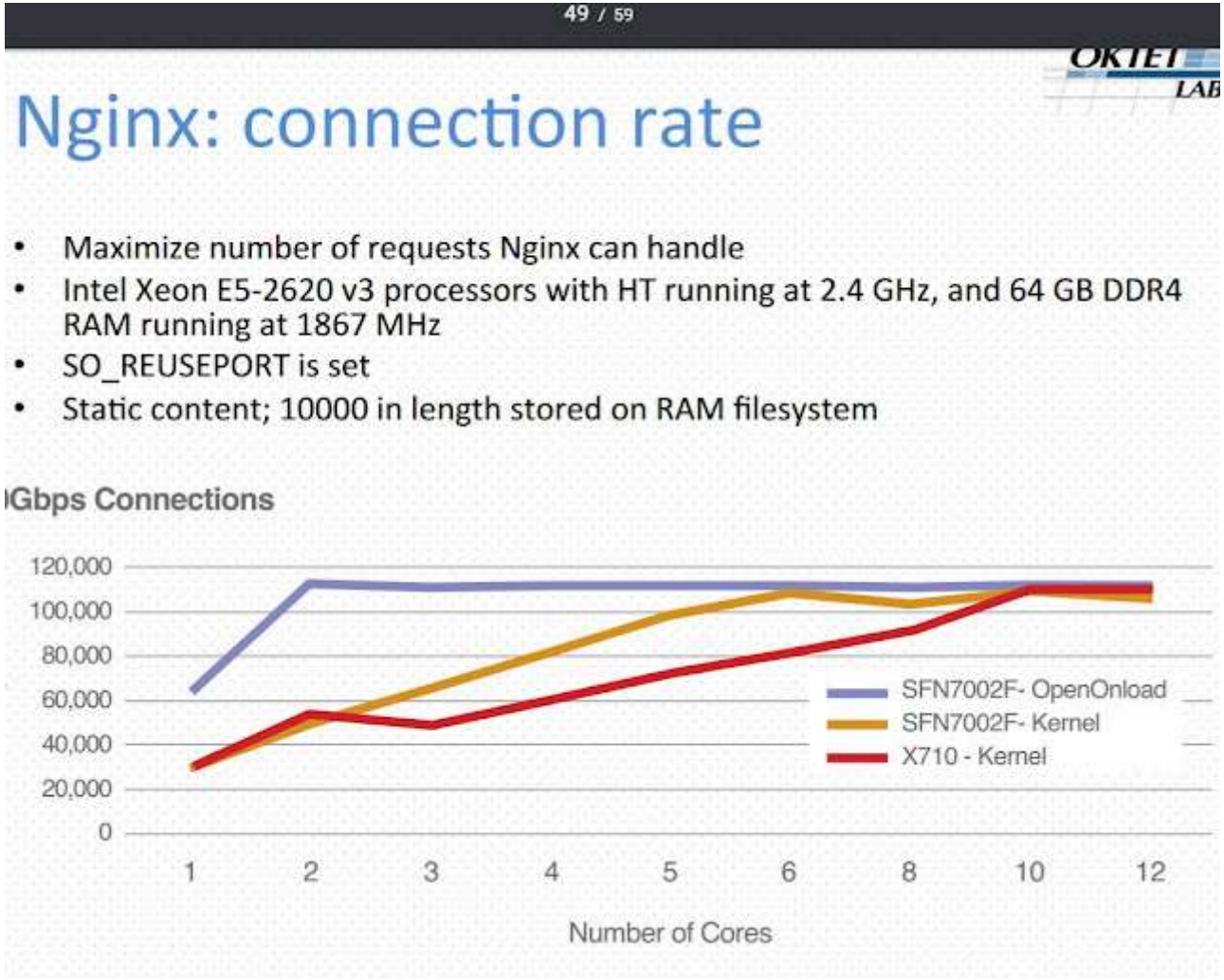
Unique Application Performance

SFN7002F's unique hardware-accelerated virtualized NIC (vNIC) architecture provides concrete benefits to multi-core computing. In many cases, network traffic that is directed to a single CPU core can overload the core and create a performance bottleneck. SFN7002Fcan use up to 2048 vNICs and 240 virtual functions to implement Receive Side Scaling (RSS) and accelerate Receive Flow Steering (RFS) to distribute network traffic across multiple cores. These technologies allow the networking I/O processing to be spread across the available CPU cores, eliminating processing bottlenecks and dramatically improving application performance and scalability. SFN7002F also improves data intensive key value storage applications such as NoSQL, memcached, Couchbase and other big data applications by accelerating throughput and reducing latency for compute and Ethernet storage. SFN7002F also features hardware acceleration to optimize packet delivery, such as integrated layer 2 switching capability and VLAN insertion/removal, along with TCP segmentation offload (TSO) to reduce CPU load.

http://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=2131

Comparison on Ngnix:

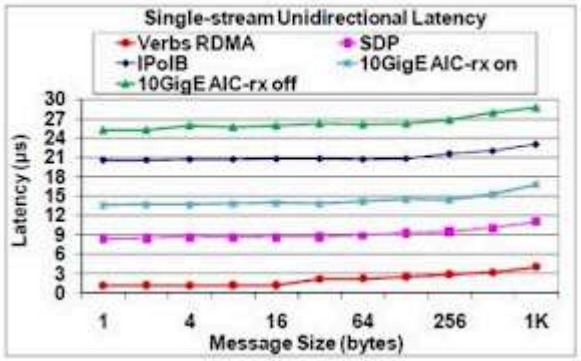
- SFN7002F - OpenOnload = TOE(hardware TCP/IP) + Onload (kernel-bypass)
- SFN7002F - Kernel = TOE(hardware TCP/IP)
- X710 - Kernel = Conventional Ethernet Adapter + Linux Kernel

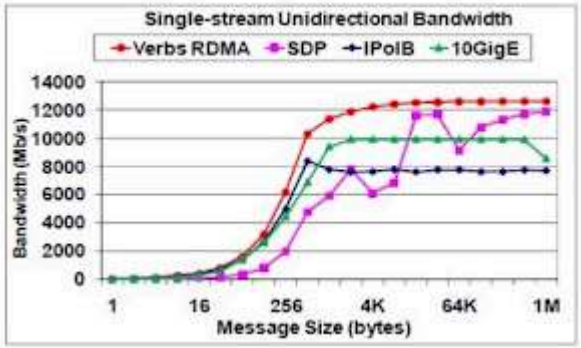


http://it-events.com/system/attachments/files/000/001/097/original/OpenOnload-for-LinuxPiter.pdf

Here are the results of the test in 2010 to compare different protocols: TCP, SDP, VERBS.
On a single ConnectX network device, in one thread, different protocols are run in two modes:

- Ethernet 10 GigE – TCP with on/off AIC-Rx
- Infiniband – TCP (IPoIB), SDP, VERBS





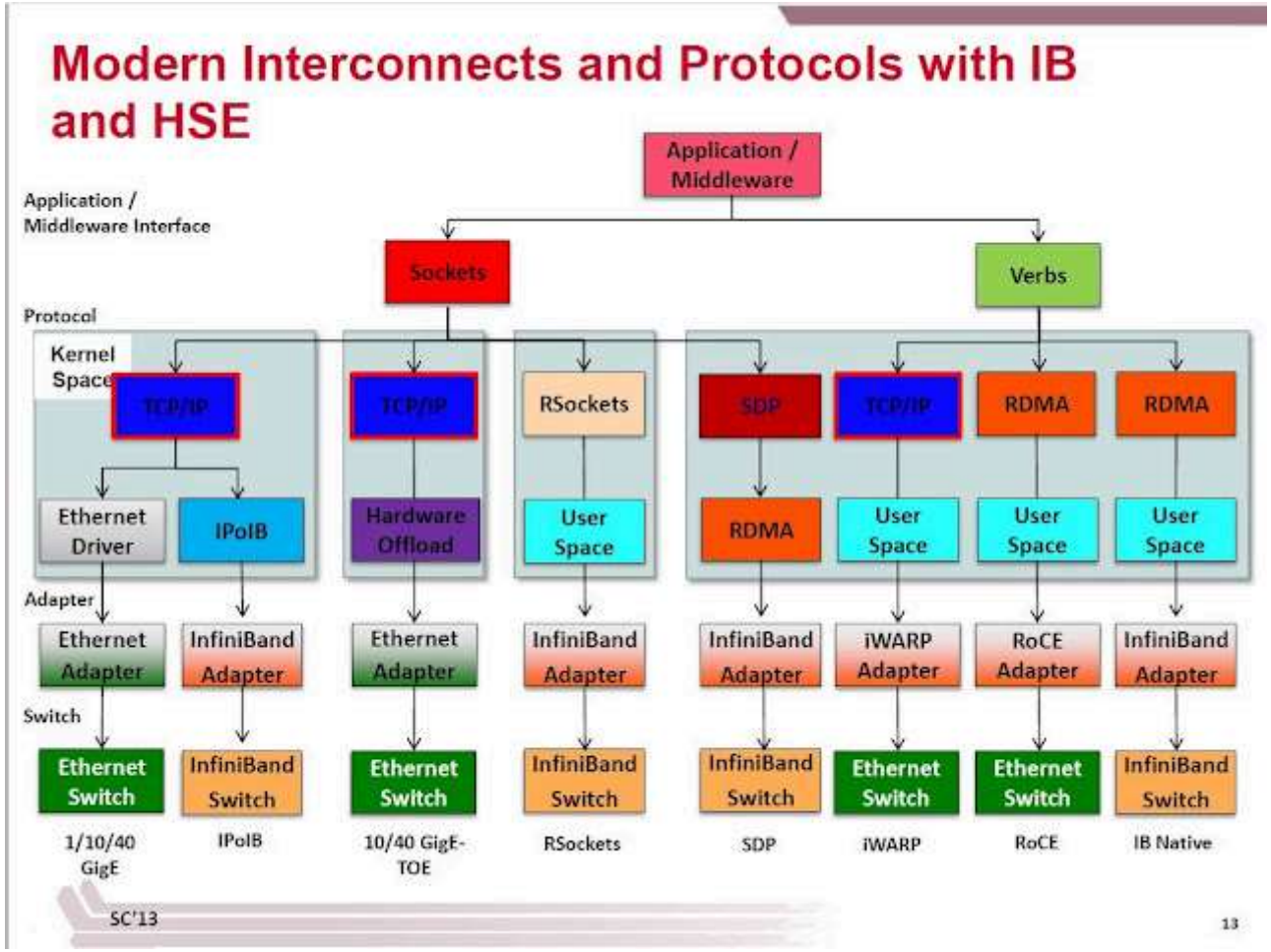
AIC-Rx - Adaptive Interrupt Coalescing (Receiving)

"Evaluation of ConnectX Virtual Protocol Interconnect for Data Centers"; R. Grant, A. Afsahi, P.Balaji; Ontario University and Argonne National Laboratory
Testing same ConnectX card in both 40g InfiniBand and 10g CNEE modes with jumbo frames for throughput

https://www.informatix-sol.com/docs/TCP_bypass.pdf

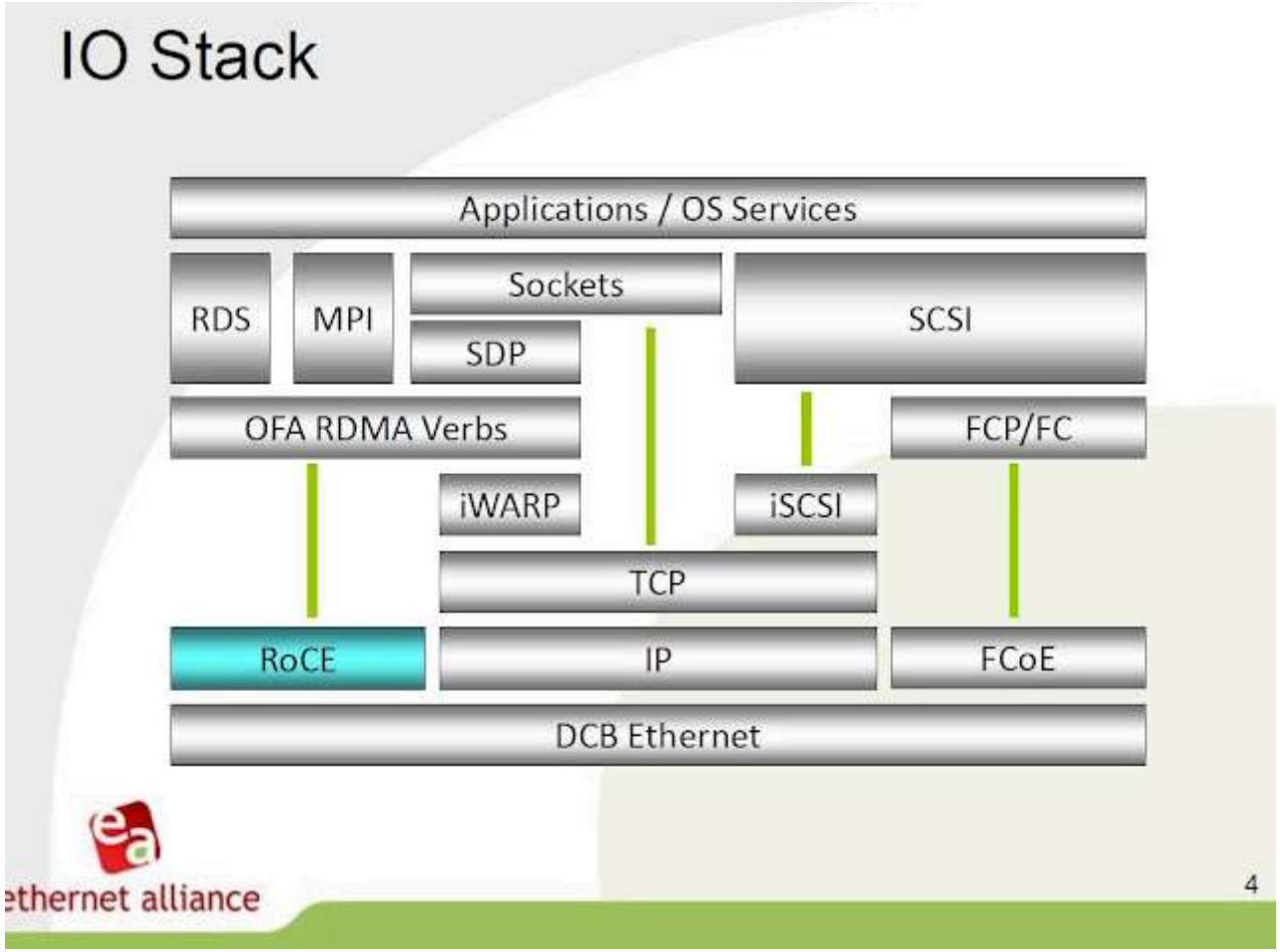
We'll show what protocols and which adapters can be used, as well as what connections can be used for exchange, for example:

Ethernet-Switch <-> RoCE-adapter <-> Verbs-API.



http://ircc.fiu.edu/download/sc13/AdvInfiniband_Slides.pdf

Which APIs and protocols can be used for an Ethernet connection:



http://www.ethernetalliance.org/wp-content/uploads/2011/10/document_files_RoCE_Intro_and_Update_100716.pdf

Feature comparison: Infiniband, iWARP, RoE, RoCE

IB and HSE: Feature Comparison

	IB	iWARP/HSE	RoE	RoCE
Hardware Acceleration	Yes	Yes	Yes	Yes
RDMA	Yes	Yes	Yes	Yes
Congestion Control	Yes	Optional	Optional	Yes
Multipathing	Yes	Yes	Yes	Yes
Atomic Operations	Yes	No	Yes	Yes
Multicast	Optional	No	Optional	Optional
Data Placement	Ordered	Out-of-order	Ordered	Ordered
Prioritization	Optional	Optional	Optional	Yes
Fixed BW QoS (ETS)	No	Optional	Optional	Yes
Ethernet Compatibility	No	Yes	Yes	Yes
TCP/IP Compatibility	Yes (using IPoIB)	Yes	Yes (using IPoIB)	Yes (using IPoIB)

http://www.ics.uci.edu/~ccgrid11/files/ccgrid11-ib-hse_last.pdf

Comparison of the throughput of two Ethernet types with RDMA support: iWARP and RoCE

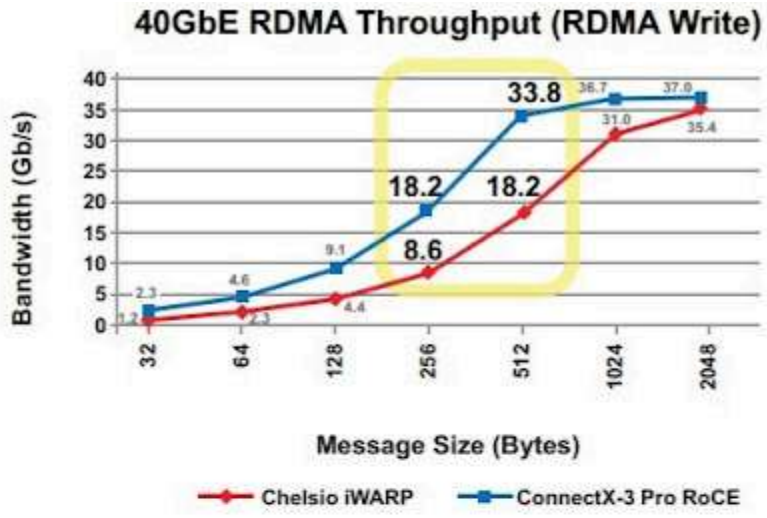


Figure 5. *40Gb Ethernet Throughput Benchmark*

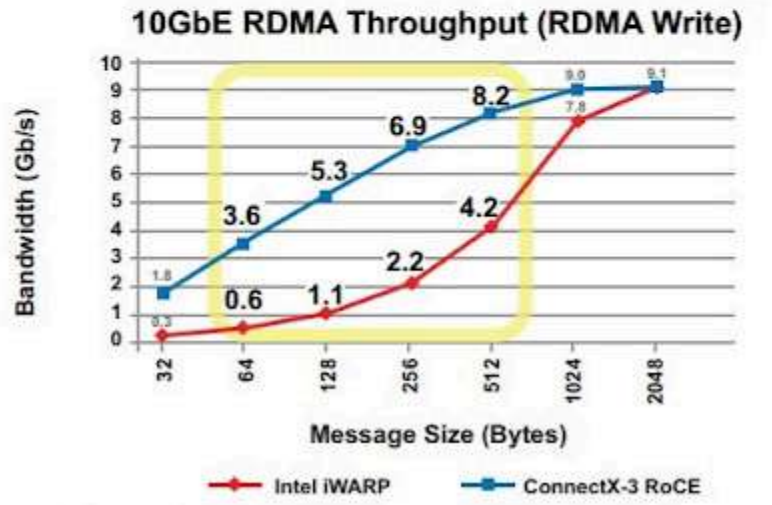
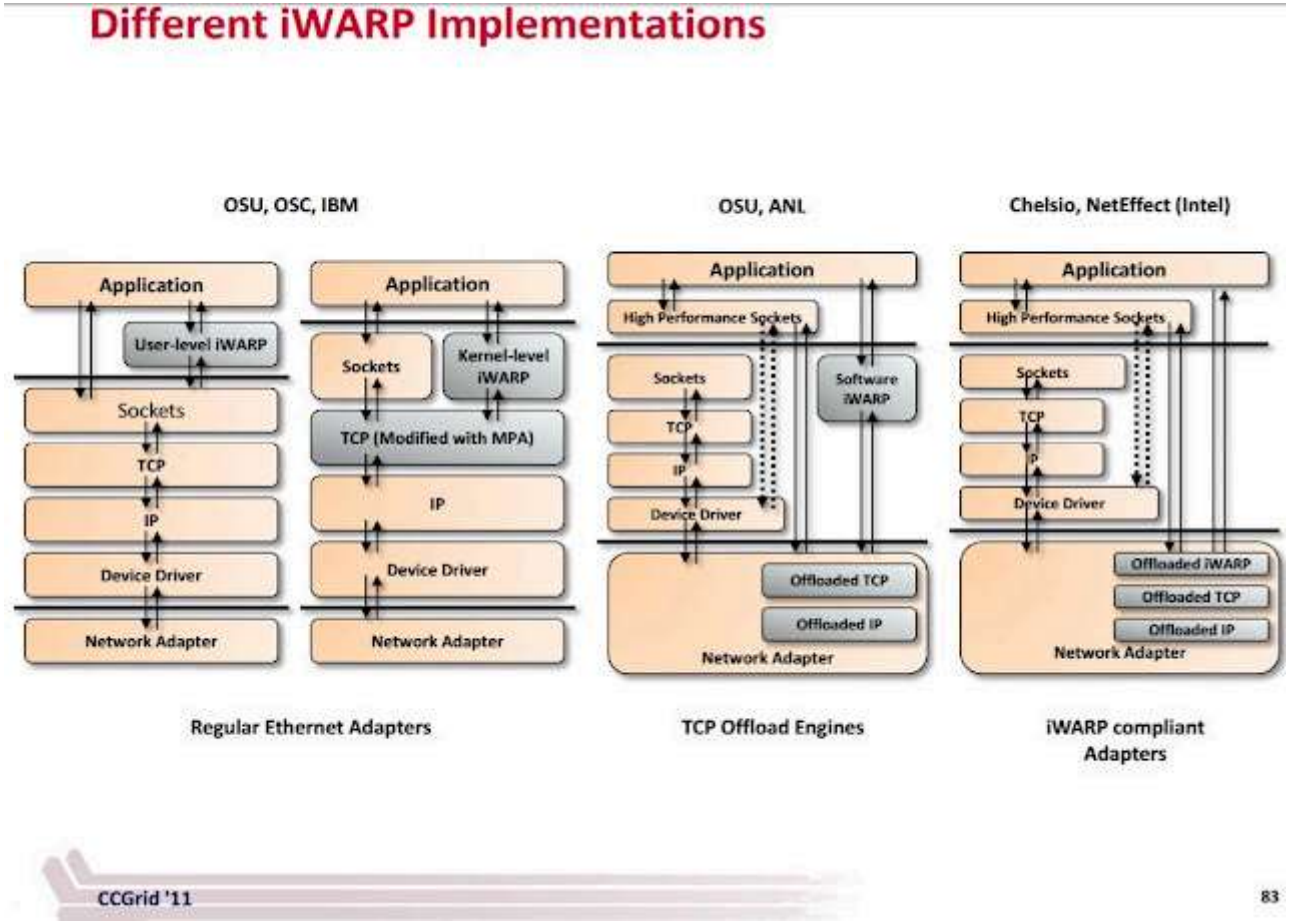


Figure 6 *10Gb Ethernet Throughput Benchmark*

http://www.mellanox.com/related-docs/whitepapers/WP_RoCE_vs_iWARP.pdf

There are many implementations of iWARP:

- **Software-iWARP** using standard Ethernet-adapter and Ethernet TCP Offload
- **Hardware-iWARP** using adapter with Offload-iWARP hardware support



http://www.ics.uci.edu/~ccgrid11/files/ccgrid11-ib-hse_last.pdf

Infiniband adapter supports hardware multicasting with guaranteed delivery.

ConnectX® Single/Dual-Port Adapter Devices with Virtual Protocol Interconnect®

page 2

and protection for virtual machines (VM) within the server. ConnectX gives data center managers better server utilization and LAN/SAN unification while reducing costs, power, and complexity.

Storage Acceleration — A consolidated compute and storage network achieves significant cost-performance advantages over multi-fabric networks. Standard block and file access protocols can leverage InfiniBand RDMA for high-performance storage access. Fibre Channel frame encapsulation (FCoIB or FCoE) and hardware offloads enable simple connectivity to Fibre Channel SANs.

Software Support

All Mellanox adapter devices are compatible with TCP/IP and OpenFabrics-based RDMA protocols and software. They are also compatible with InfiniBand and cluster management software available from OEMs. Adapters based on the devices are compatible with major operating system distributions.

OPENFABRICS ALLIANCE

FEATURE SUMMARY

INFINIBAND
- IBTA Specification 1.2.1 compliant
- 10, 20, or 40Gb/s per port
- RDMA, Send/Receive semantics
- Hardware-based congestion control
- Atomic operations
- 16 million I/O channels
- 256 to 4Kbyte MTU
- 2GB messages
- 9 virtual lanes: 8 data + 1 management

ENHANCED INFINIBAND
- Hardware-based reliable transport
- Hardware-based reliable multicast
- Scalable Reliable Connected transport
- Enhanced Atomic operations
- Fine grained end-to-end QoS

ETHERNET
- IEEE Std 802.3ae 10 Gigabit Ethernet
- EEE Std 802.3ak 10GBASE-CX4
- IEEE Std 802.3ap Backplanes
- IEEE Std 802.3ad Link Aggregation and Failover
- EEE Std 802.3x Pause
- IEEE Std 802.1Q VLAN tags
- EEE Std 802.1p Priorities
- Multicast
- Jumbo frame support (10KB)
- 128 MAC/VLAN addresses per port
- MAC and VLAN based filtering
- Class Based Flow Control / Per Priority Pause

HARDWARE-BASED I/O VIRTUALIZATION
- Single-Root IOV
- Address translation and protection
- Multiple queues per virtual machine
- VMware NetQueue support
- PCISIG IOV compliance

ADDITIONAL CPU OFFLOADS
- TCP/UDP/IP stateless offload
- Intelligent interrupt coalescence
- Compliant to Microsoft RSS and NetDMA

STORAGE SUPPORT
- T10-compliant Data Integrity Field support
- Fibre Channel over InfiniBand or Ethernet

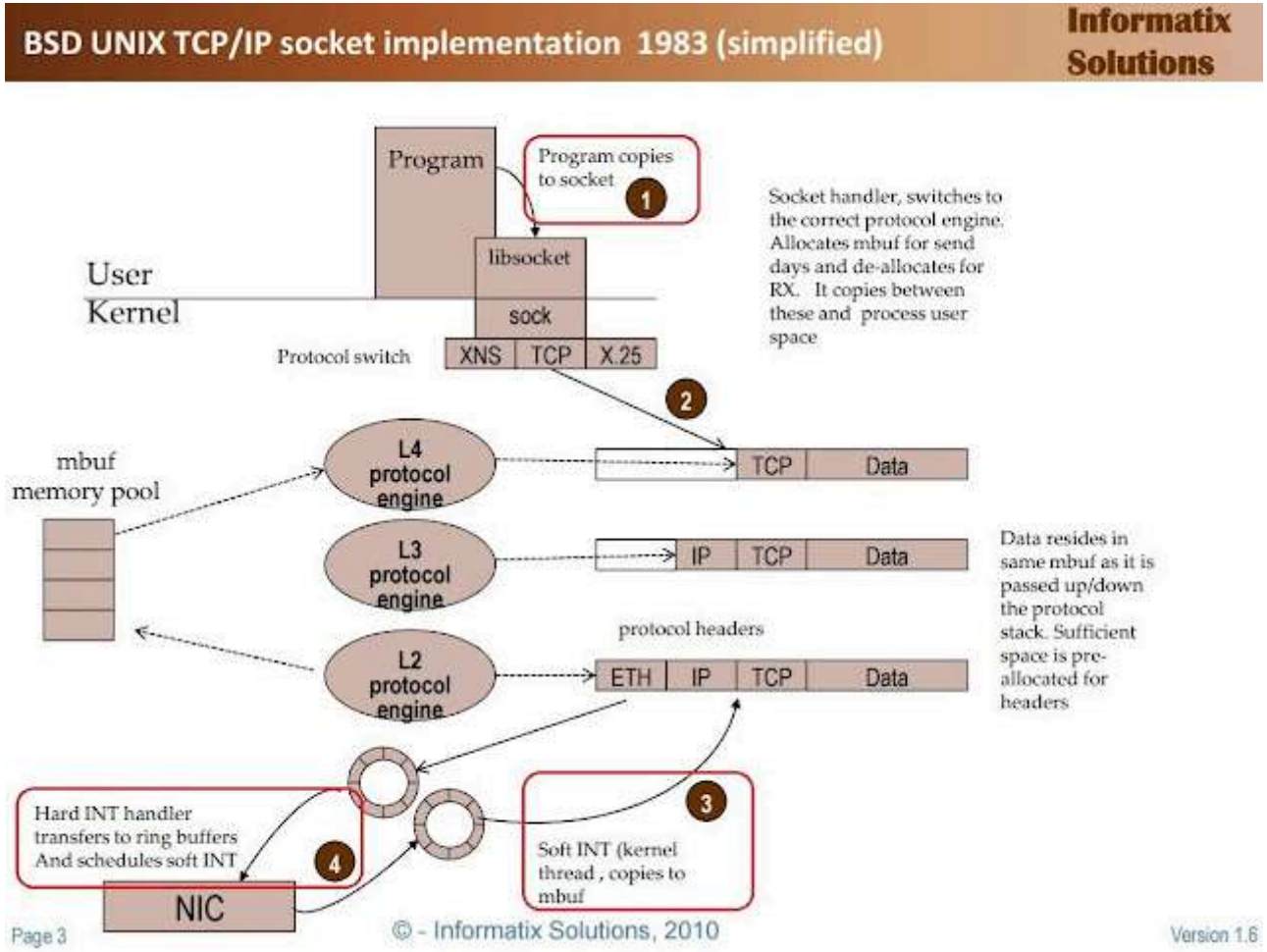
COMPATIBLITY

CPU
- AMD X86, X86_64
- Intel X86, EM64T, IA-32, IA-64
- SPARC
- PowerPC, MIPS, and Cell
- Firmware and debug tools (MFT, IBDIAG)

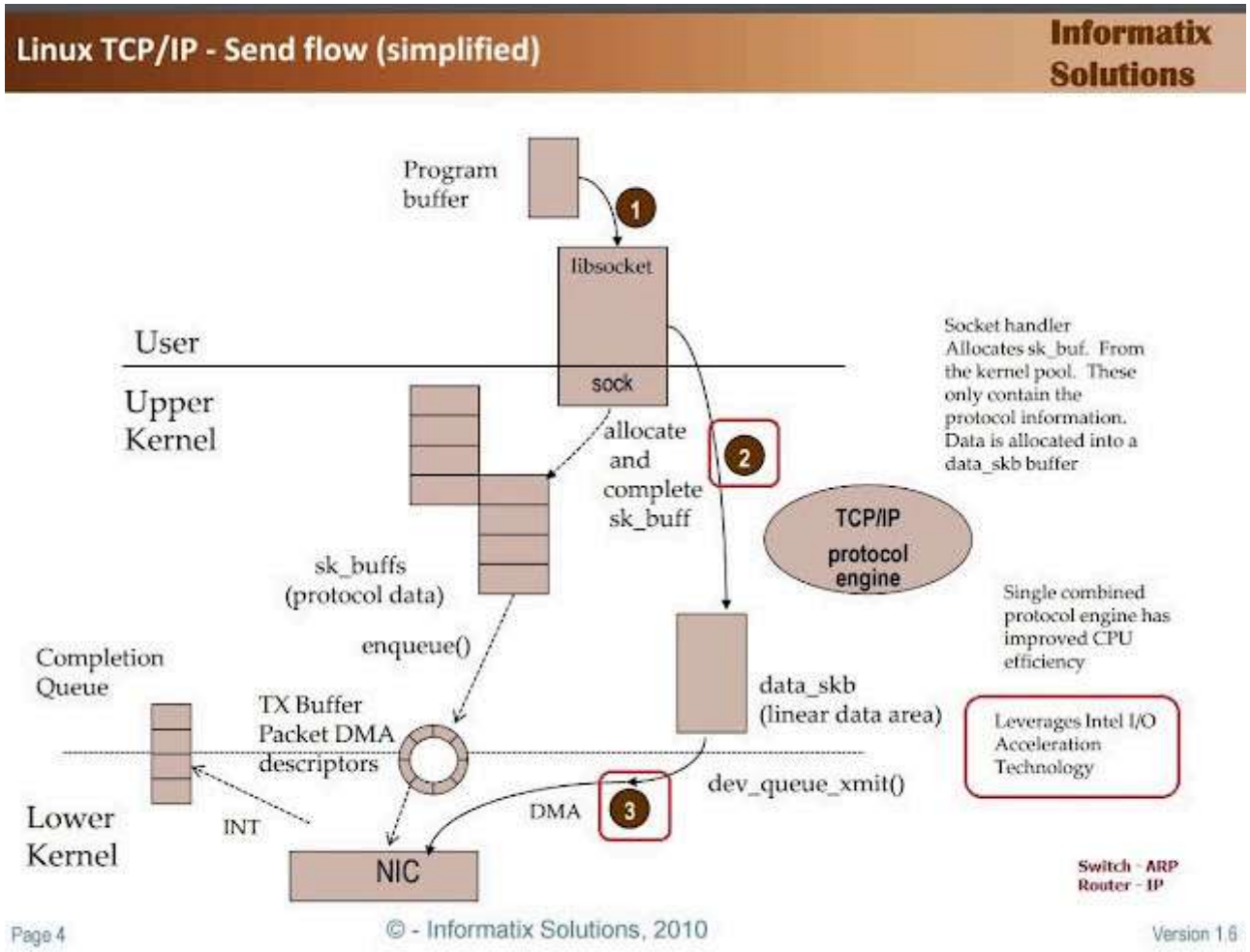
Ethernet
- MIB, MIB-II, MIB-II Extensions, RMON, RMON 2
- Configuration and diagnostic tools

http://www.mellanox.com/related-docs/prod_silicon/PB_ConnectX_Silicon.pdf

We'll show 4 screenshots of the overhead costs of the standard operation of the TCP/IP stack in the Linux kernel - these overheads are avoided in more optimized protocols: SDP / VMA, uDAPL.



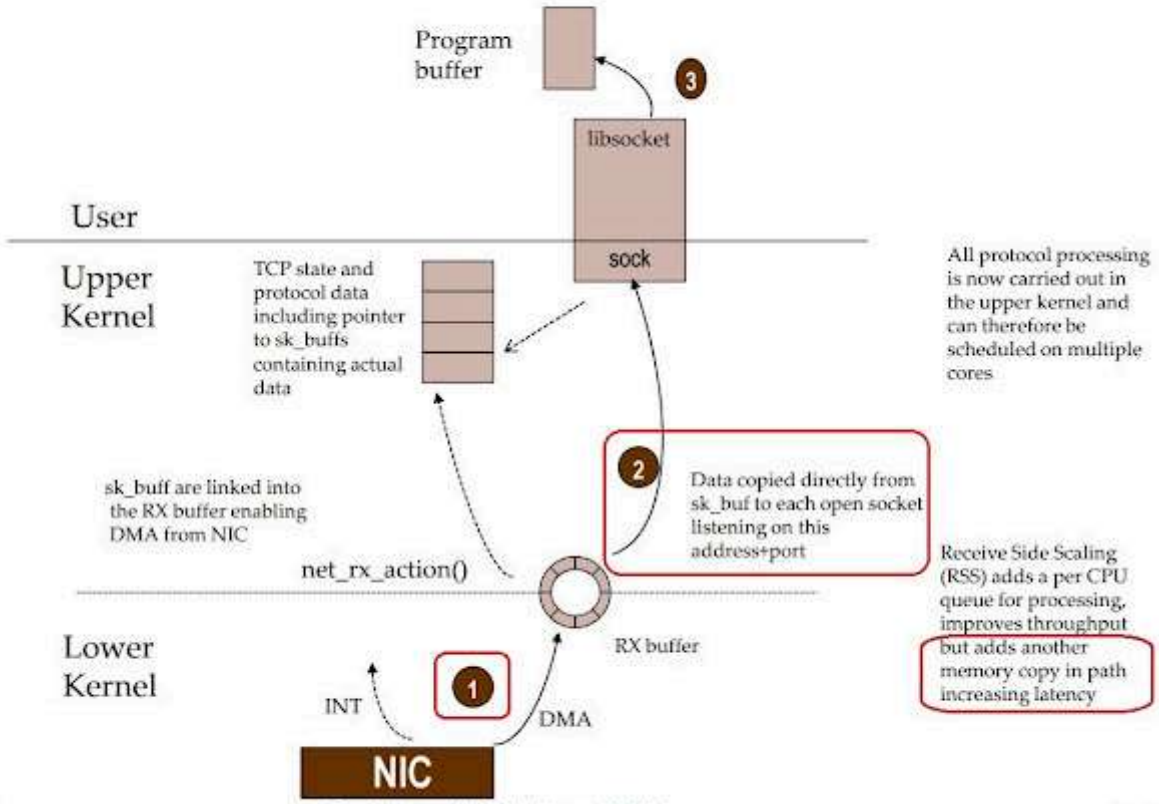
https://www.informatix-sol.com/docs/TCP_bypass.pdf



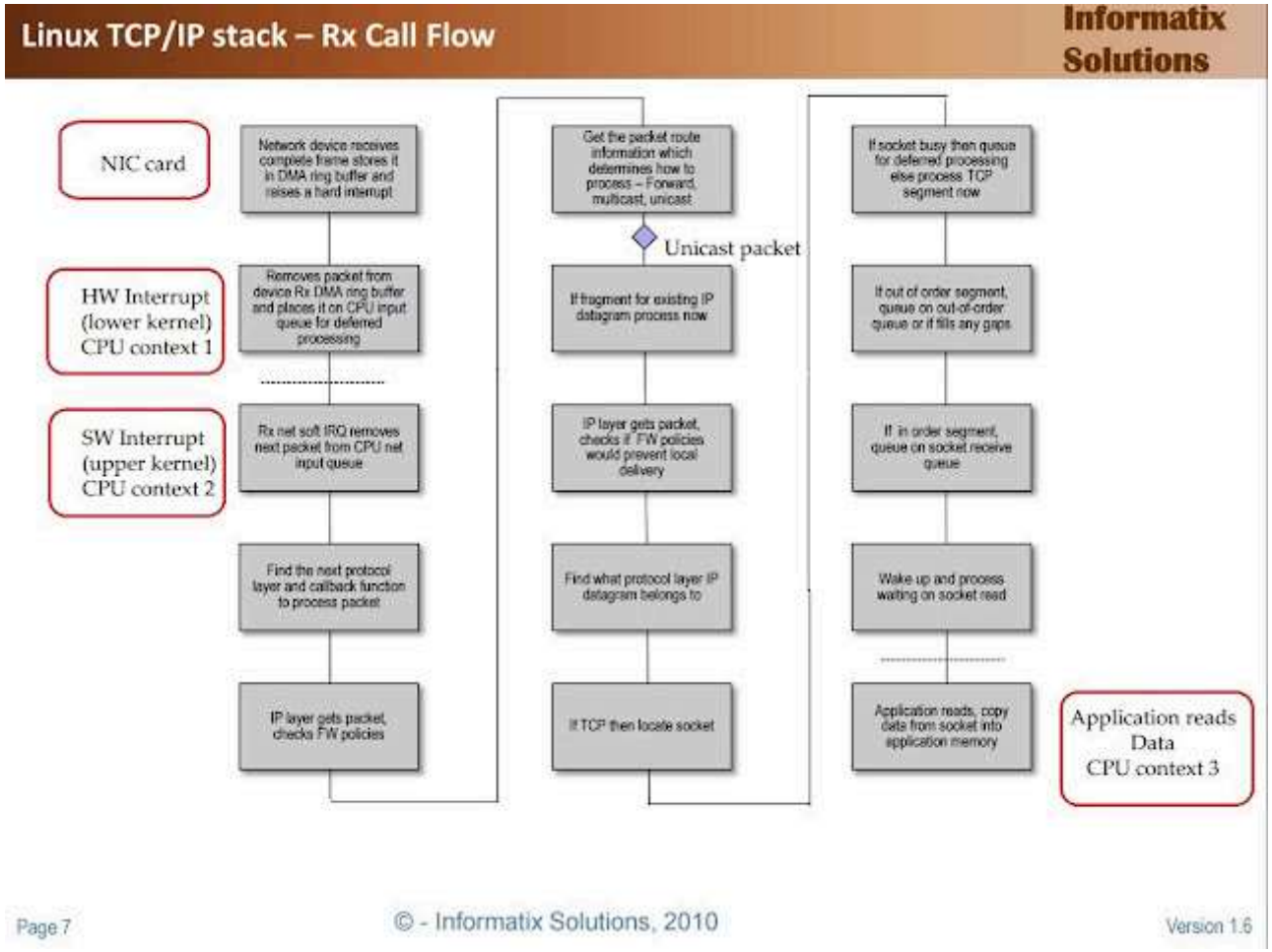
https://www.informatix-sol.com/docs/TCP_bypass.pdf

Linux TCP/IP – receive flow (simplified)

Informatix Solutions



https://www.informatix-sol.com/docs/TCP_bypass.pdf



https://www.informatix-sol.com/docs/TCP_bypass.pdf

VMA (Voltaire Messaging Accelerator) - is libvma.so library, which can be used to speed up data exchange not only for the UDP protocol, but also for TCP.

(page 2): http://www.mellanox.com/related-docs/whitepapers/HP_Mellanox_Low%20Latency_Benchmark%20Report%202012.pdf

Finally, to maximize the benefits of low latency networking hardware for the end user application, the Mellanox Messaging Accelerator (VMA) Linux library has been enhanced in 2012. ***VMA has long been capable*** *of performing socket acceleration via OS/kernel bypass* ***for UDP unicast and multicast*** *messages (typically associated with market data feeds from Exchanges), without rewriting the applications.*
In ***version 6*** *this dynamically-linked user-space library can now* ***also accelerate any application using TCP*** *messages (typically associated with orders/acknowledgements to and from the Exchanges), passing TCP traffic directly from the user-space application to the network adapter. Bypassing the kernel and IP stack interrupts delivers extremely low-latency because context switches and buffer copies are minimized. The result is high packet-per-second rates, low data distribution latency, low CPU utilization and increased application scalability.*

SDP (Socket Direct Protocol) - is a protocol and the libsdp.so loadable library, which can be used to speed up data exchange via the TCP protocol.

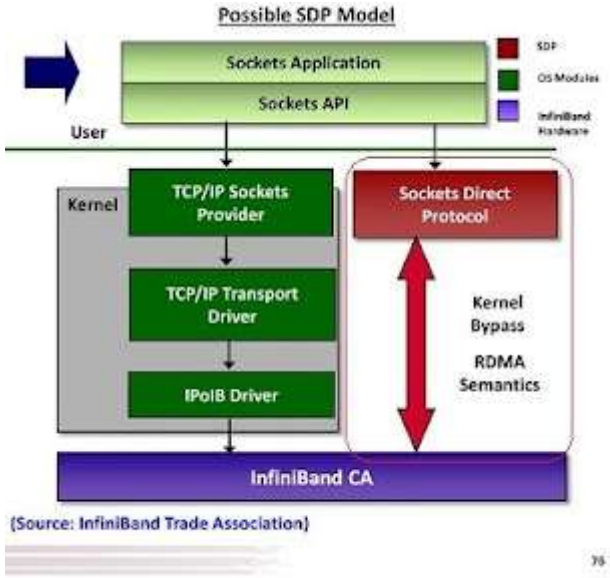
(page 9): https://www.suse.com/docrep/documents/rhvs3sxk4t/RMDS6-Novell-IntelXeon-HPBL460c-VoltaireIB-GbE_en.pdf

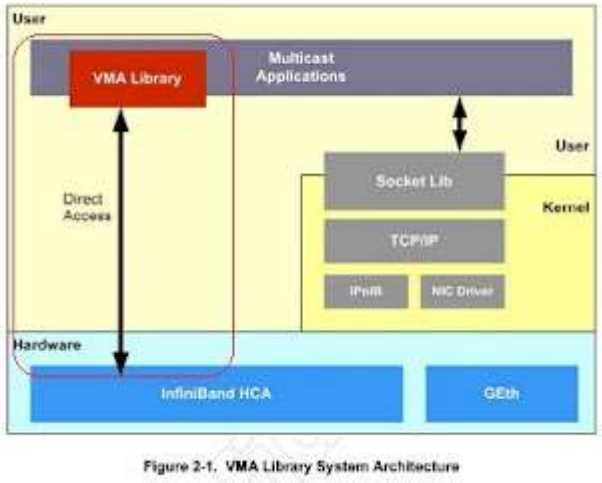
Latency tests on InfiniBand used the VMA (Voltaire Messaging Accelerator) library for UDP traffic and the SDP library was used for TCP traffic.

The way of SDP and VMA protocols' reducing the overhead for data exchange by working from user-space direct with the Infiniband device bypassing the kernel-space:

- **SDP**: Page 76: http://ircc.fiu.edu/download/sc13/Infiniband_Slides.pdf

- **VMA**: Page 15 figure 2-1: http://lists.openfabrics.org/pipermail/general/attachments/20081016/3fe4fd45/attachment.obj





Left**:** **InfiniBand and High-Speed Ethernet for Dummies.pdf**

Right**:** **Voltaire Messaging Accelerator (VMA)Library for Linux.pdf**

Socket Direct Protocol (SDP) over PCI Express interconnect: Design, implementation and evaluation.

Simplified scheme of operation of SDP-protocol (used instead of TCP):

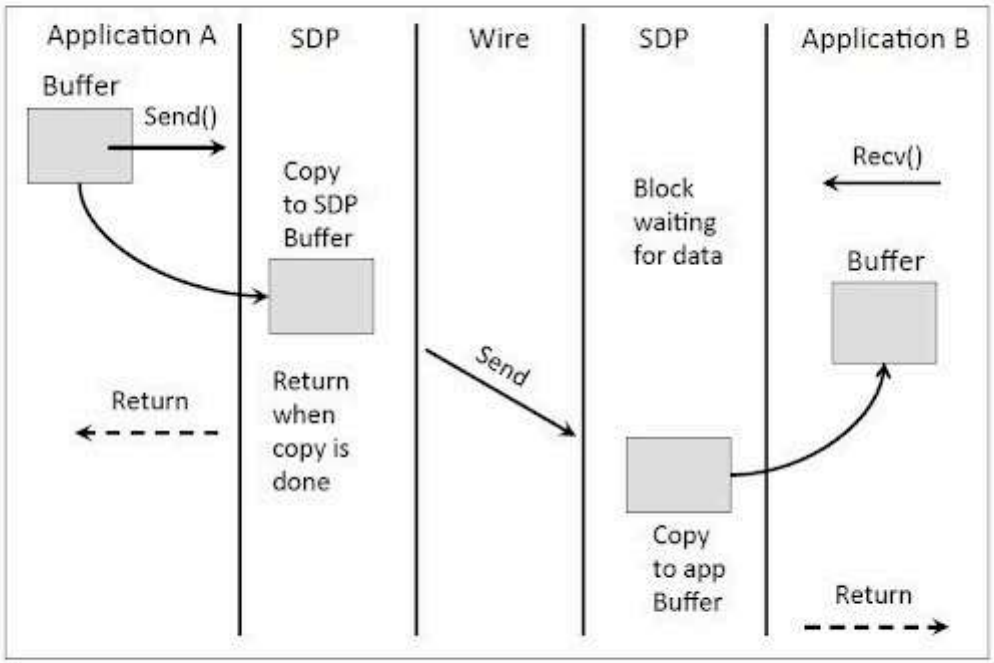


Figure 2.7: Bcopy data flow [16].

https://cs-nsl-wiki.cs.surrey.sfu.ca/theses/bu-khamsin12.pdf

What changes are needed to pass from TCP to SDP.

It's not necessary to change the code. I remind you that a binary executable program compiled for TCP and UDP can be made to work by using SDP or VMA protocols. For this purpose, it is necessary to run the program as follows: LD_PRELOAD=libsdp.so program.bin or LD_PRELOAD=libvma.so program.bin

```c
int main(int argc, char *argv[])
{
    int sockfd;
    int len;
    struct sockaddr_in address;
    int result;
    //Create socket for client.
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr("127.0.0.1");
    address.sin_port = htons(7734);
    len = sizeof(address);
    result = connect(sockfd, (struct sockaddr *)&address, len);
    ...
```

```c
int main(int argc, char *argv[])
{
    int sockfd;
    int len;
    struct sockaddr_in address;
    int result;
    //Create socket for client.
    sockfd = socket(AF_INET_SDP, SOCK_STREAM, 0);
    address.sin_family = AF_INET_SDP;
    address.sin_addr.s_addr = inet_addr("127.0.0.1");
    address.sin_port = htons(7734);
    len = sizeof(address);
    result = connect(sockfd, (struct sockaddr *)&address, len);
    ...
```

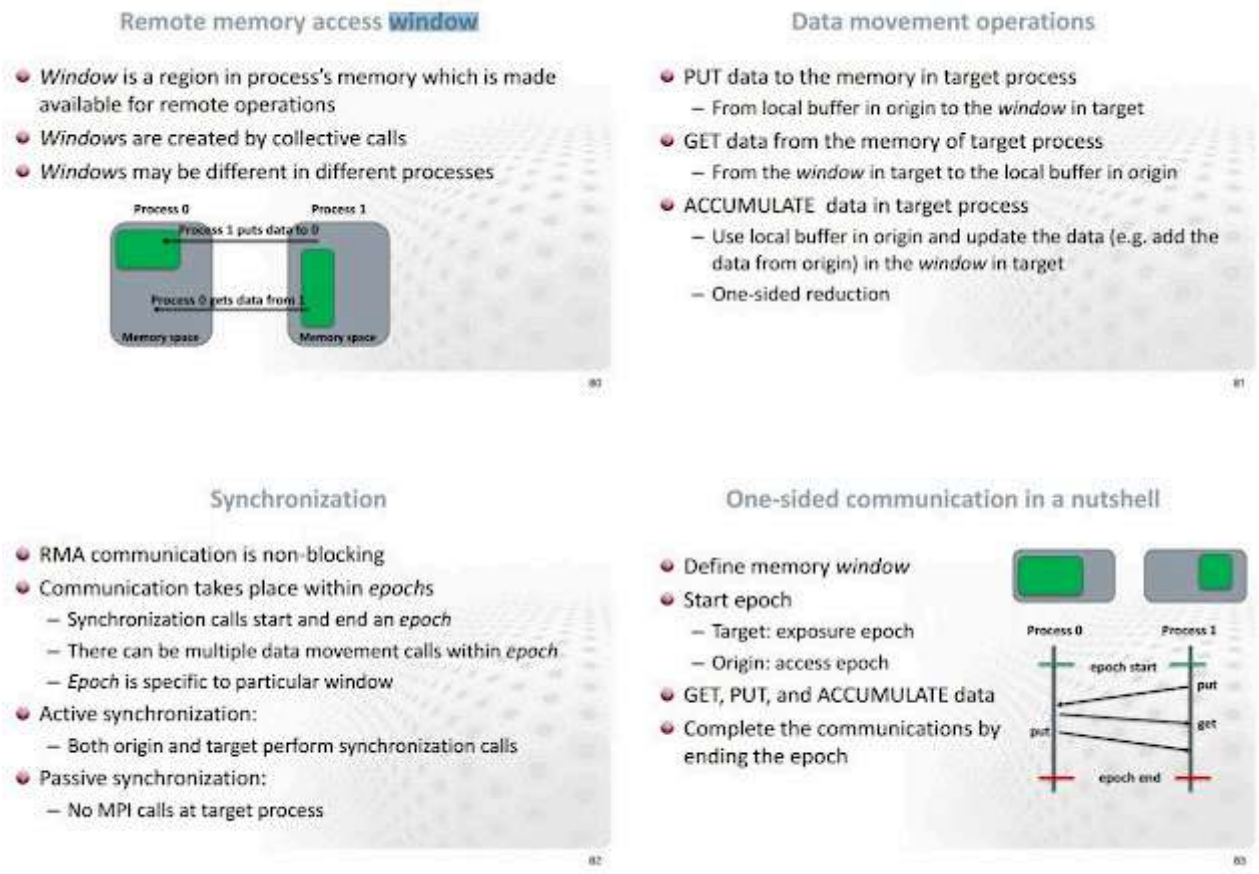
Figure 4.2: Code snippet in C shows the difference between TCP socket (left) and SDP socket (right).

number, or IP address. We use a similar approach as SDP Infiniband implementation. We register our implementation to the same address family used by Infiniband implementation of SDP and use the same preloaded library. An applications developer also has the option to natively support SDP and skip the use of this component by using `AF_INET_SDP` directly as shown in Figure 4.2.

https://cs-nsl-wiki.cs.surrey.sfu.ca/theses/bu-khamsin12.pdf

MPI (Message Passing Interface) - is an API that can use the uDAPL protocol and physical interfaces (RoCE, Infiniband, Intel Omni-Path, PCIe - that support RDMA and uDAPL protocol) for direct reading from the memory of another server without requiring it to send this data and even without a mandatory call to any hardware interruptions.

This interaction is more and more similar to one multiprocessor server with shared NUMA (Non-uniform memory access) memory, but without cache coherency.

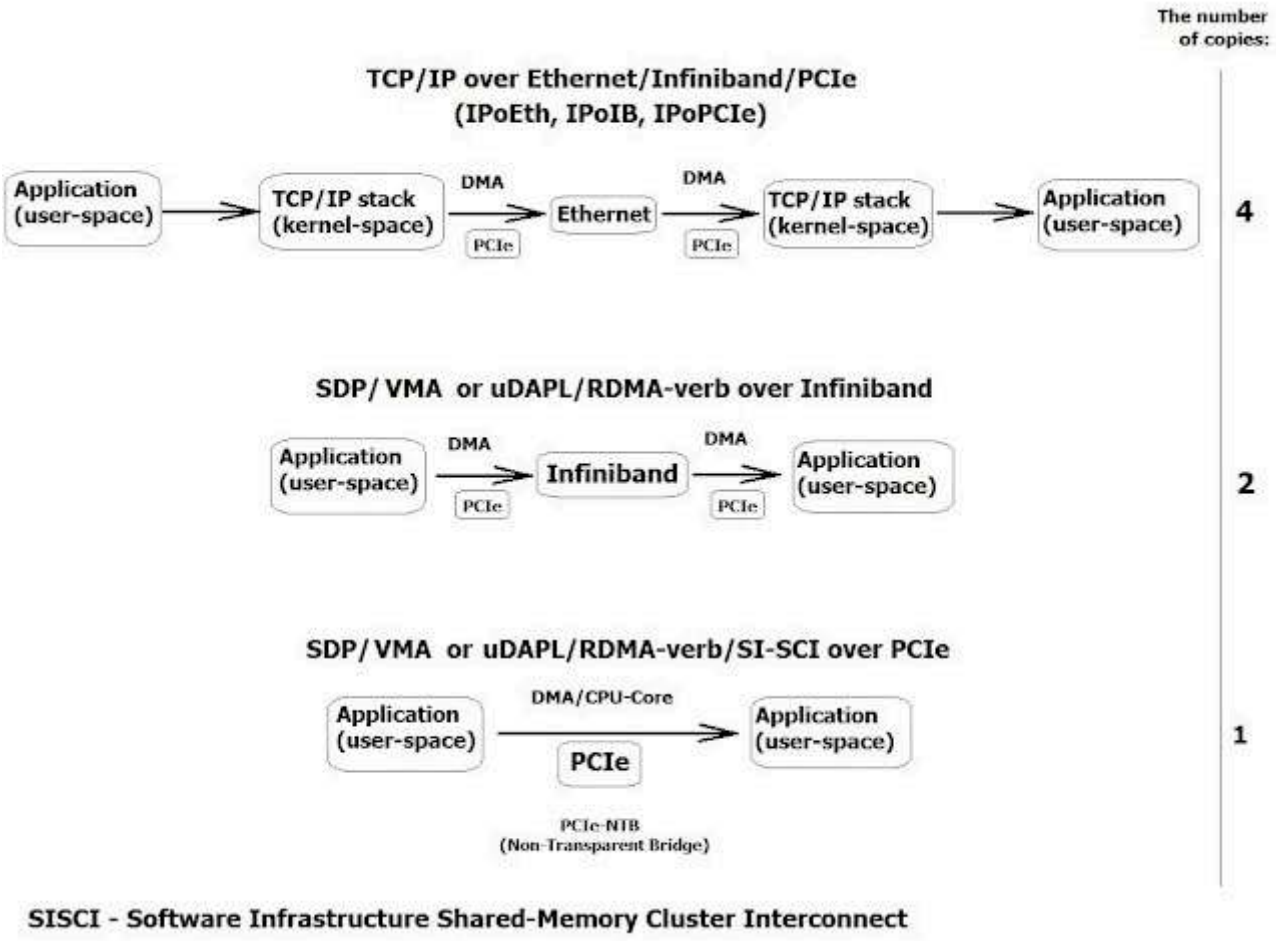


http://www.training.prace-ri.eu/uploads/tx_pracetmo/advanced_parallel_handoutAPPC16.pdf

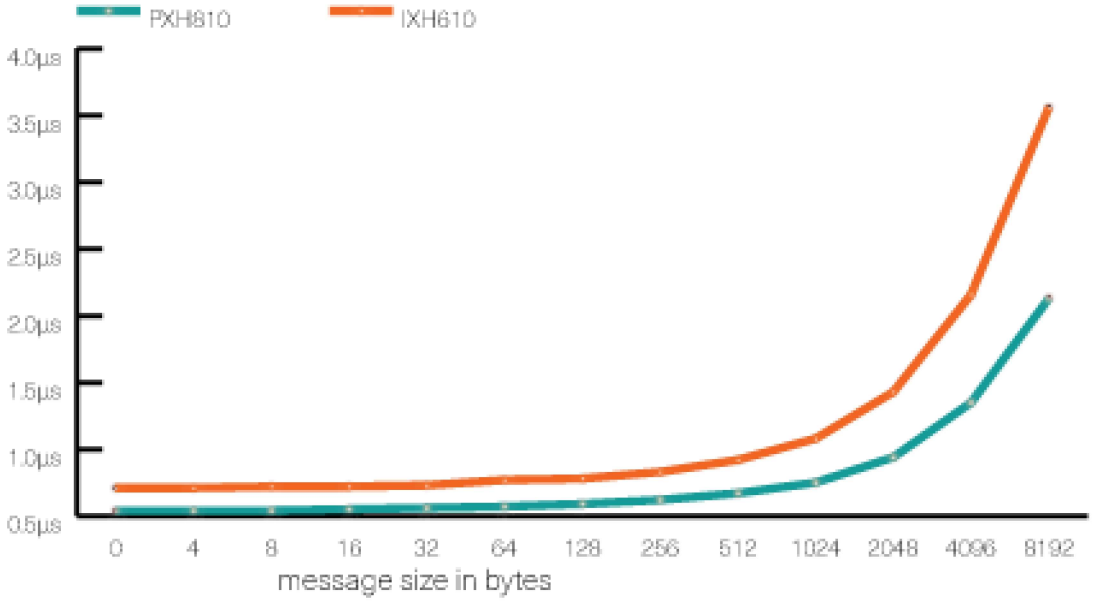
3. The fastest and most available interface – PCIe

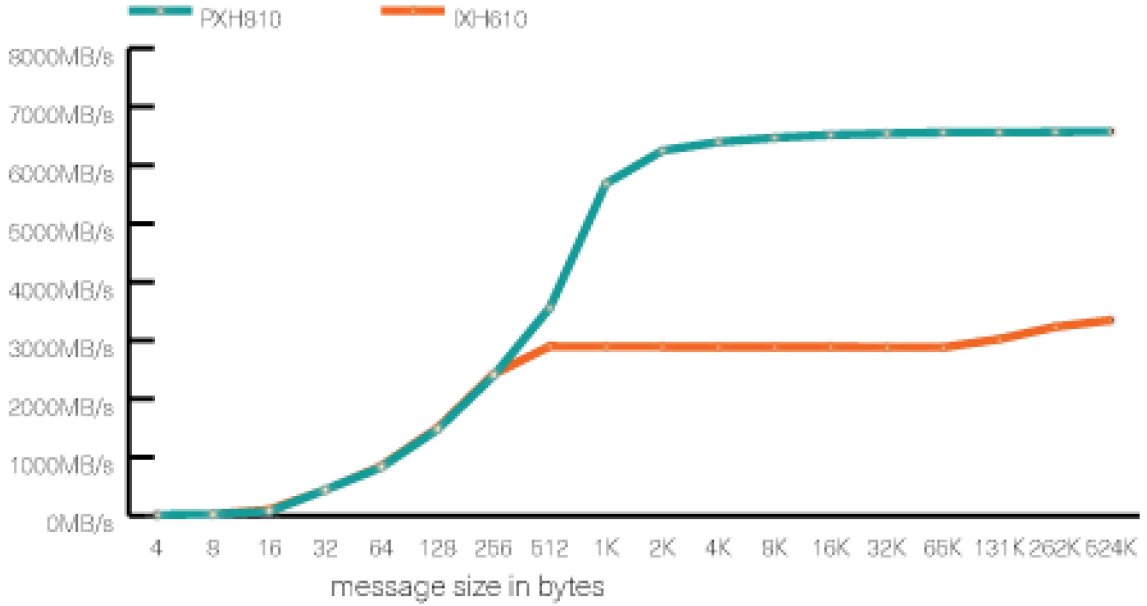
The number of redundant copies inside the devices and between them when data is exchanged between two CPUs (servers) is shown schematically. This is a short summary of the above.

- bypassing the kernel-space
- bypassing the network-adapter



Latencies from 0.5 microseconds when exchanging the data via PCI Express 2.0 16x with a speed up to 7 GB/s (56 Gbit/sec) (bandwidth of modern PCIe 3.0 with 16-lanes up to 14 GB/s (112 Gbit/sec))





"Shared-Memory Cluster Interconnect (SISCI) API makes developing PCI Express® Network applications faster and easier."

http://www.dolphinics.com/products/embedded-sisci-developers-kit.html

Starting with version 2.0, PCI Express supports compound atomic operations: FetchAdd, CAS, Swap

Part II

Detailed Description of the change

Modify Terms and Acronyms as shown

AtomicOp	One of 3 architected atomic operations, where a single PCI Express transaction targeting a location in Memory Space reads the location's value, potentially writes a new value to the location, and returns the original value. This read-modify-write sequence to the location is performed atomically. AtomicOps include FetchAdd, Swap, and CAS.
CAS	Compare and Swap. An AtomicOp where the value of a target location is compared to a specified value, and if they match, another specified value is written back to the location. Regardless, the original value of the location is returned.
FetchAdd	Fetch and Add. An AtomicOp where the value of a target location is incremented by a specified value using two's complement arithmetic ignoring any carry or overflow, and the result is written back to the location. The original value of the location is returned.
Swap	Unconditional Swap. An AtomicOp where a specified value is written to a target location, and the original value of the location is returned.

Modify Section 2.1.1.1. as shown

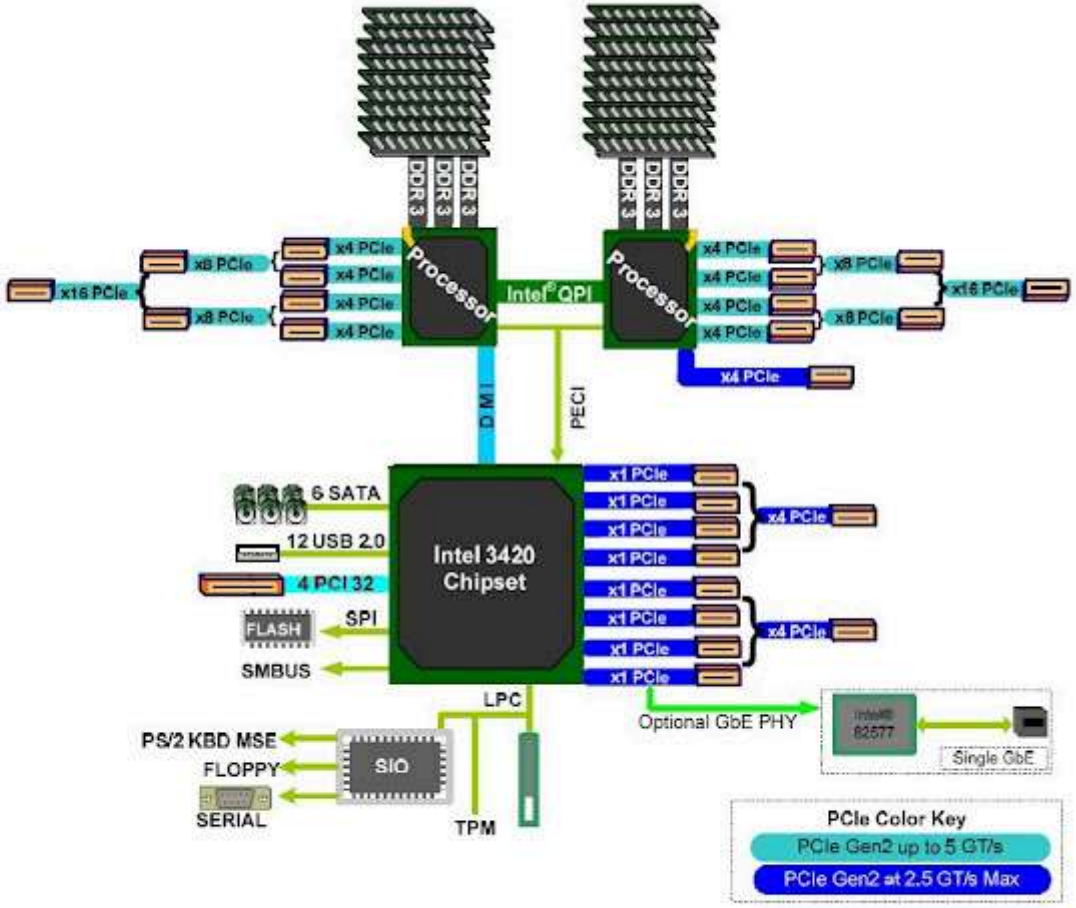
2.1.1.1. Memory Transactions

https://pcisig.com/sites/default/files/specification_documents/ECN_Atomic_Ops_080417.pdf

4. CPU – CPU interconnect

An example of 2 CPUs via QPI connection to a single server (remember, you can connect a maximum of 8 CPUs, and the price of the system grows exponentially)

Figure 1. Intel® Xeon® Processor C5500/C3500 Series Dual-Socket-Based System Block Diagram

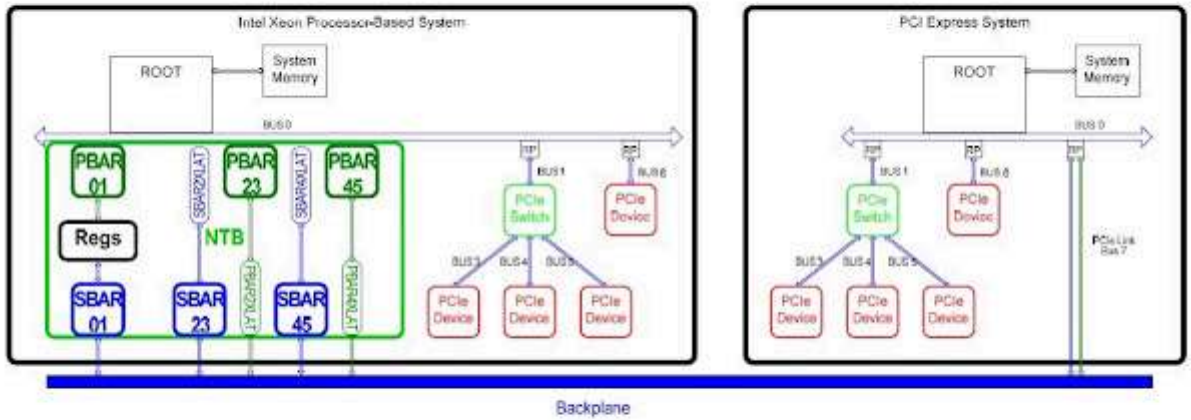


http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/xeon-c5500-c3500-non-transparent-bridge-paper.pdf

Connection of 2 CPU Intel Xeon via PCI Express interface instead of QPI.
(The used PCIe BARs are shown).

Intel® Xeon® Processor C5500/C3500 Series NTB Hardware

Figure 5. Data Paths



2.4 NTB Primary

The view into the NTB from the processor on the primary side of the NTB is comprised of its PCI Configuration space and three BARs: PBAR01, PBAR23 and PBAR45. PBAR01 contains the MMIO base address of the NTB internal registers while PBAR23 and PBAR45 are used to gain access into the memory map of the system connected to the secondary side of the NTB.

http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/xeon-c5500-c3500-non-transparent-bridge-paper.pdf

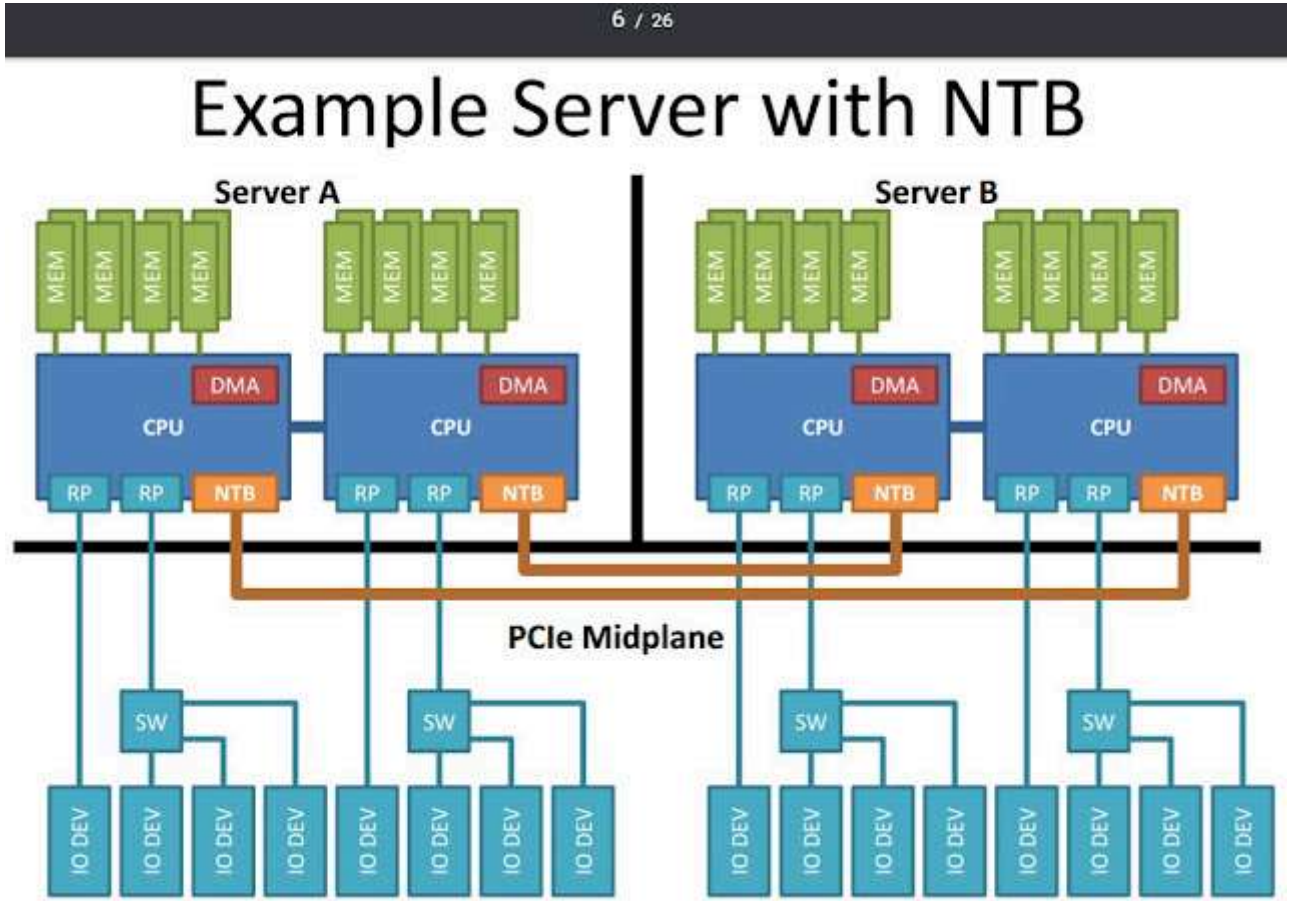
1. PCIe “Non-Transparent” Bridge

- Forwards PCIe traffic between busses like a bridge

- CPU sees the bridge as an end-point device
- CPU does not see devices on the other side
- Other side is typically attached to another CPU

2. NTB Features

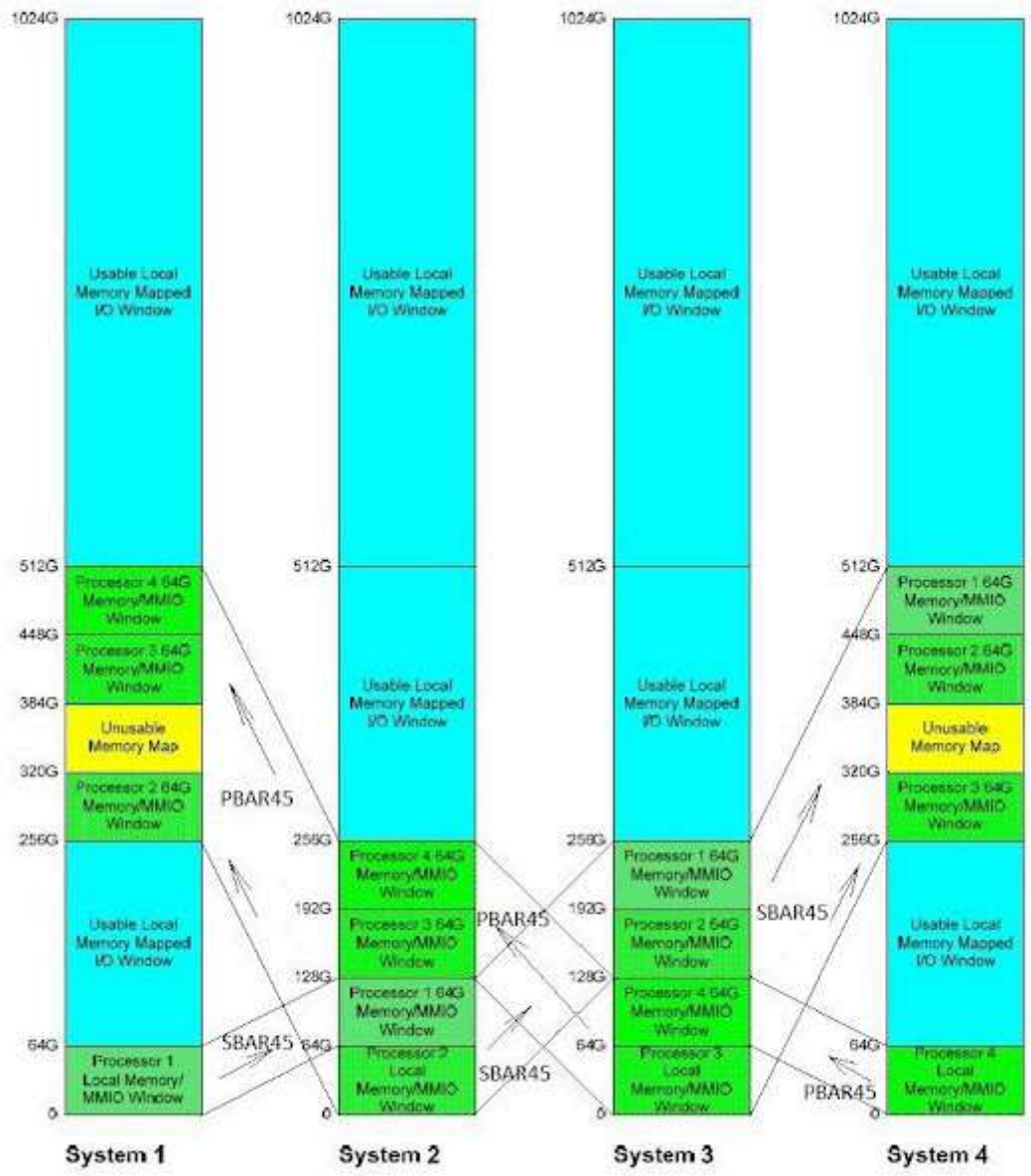
- Low Cost: Already present on CPU or PCIe bridge chips
- High Performance – PCIe wire speed: NTB connects PCIe buses
- Internally Wired – Not accessible to customer, low maintenance
- External setup also supported (redriver and cable)



http://events.linuxfoundation.org/sites/events/files/slides/Linux%20NTB_0.pdf

The memory of 4 CPUs connected via PCIe is mapped to their physical address spaces.

Figure 10. Memory Layout



http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/xeon-c5500-c3500-non-transparent-bridge-paper.pdf

Bandwidth of PCIe 3.0 16-lanes:

- for raw-data is 16 GB/s (128 Gbit/sec) by specification
- for actual-data 14 GB/s (112 Gbit/sec) by benchmark

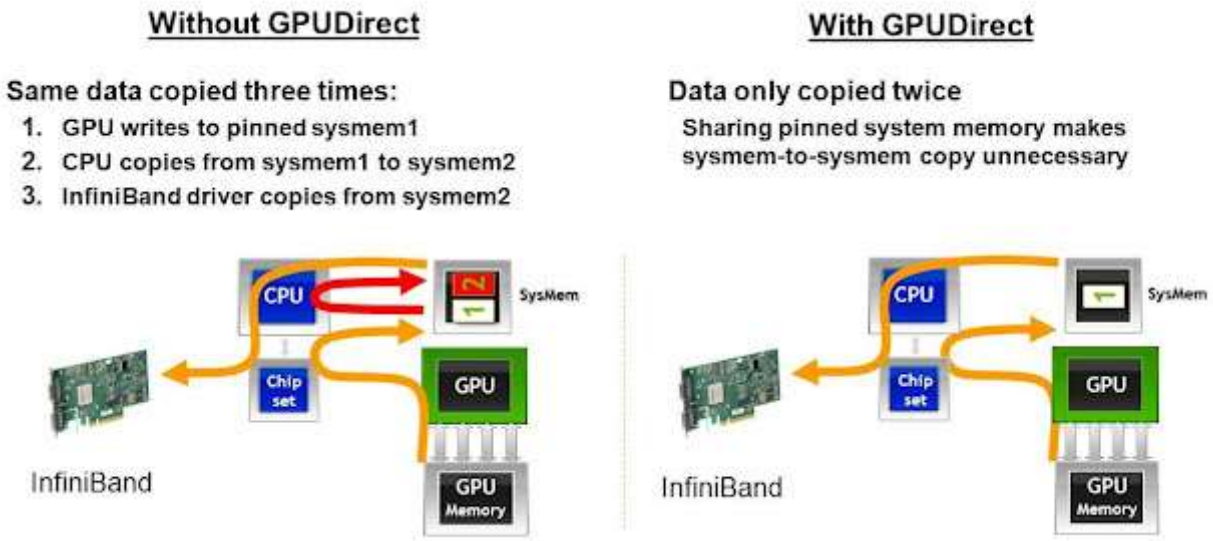
Total bandwidth for 3 PCIe-connections 40-lanes=(16+16+8) on **one CPU is 45 GB/s (280 Gbit/sec)**.

- (Processor E5 v4 Family) Intel® **Xeon**® Processor E5-4660 v4 - **PCI Express Lanes: 40**
  https://ark.intel.com/products/93796/Intel-Xeon-Processor-E5-4660-v4-40M-Cache-2_20-GHz
- (High End **Desktop Processor**) Intel® Core™ i7-6850K - **PCI Express Lanes: 40**
  https://ark.intel.com/products/94188/Intel-Core-i7-6850K-Processor-15M-Cache-up-to-3_80-GHz

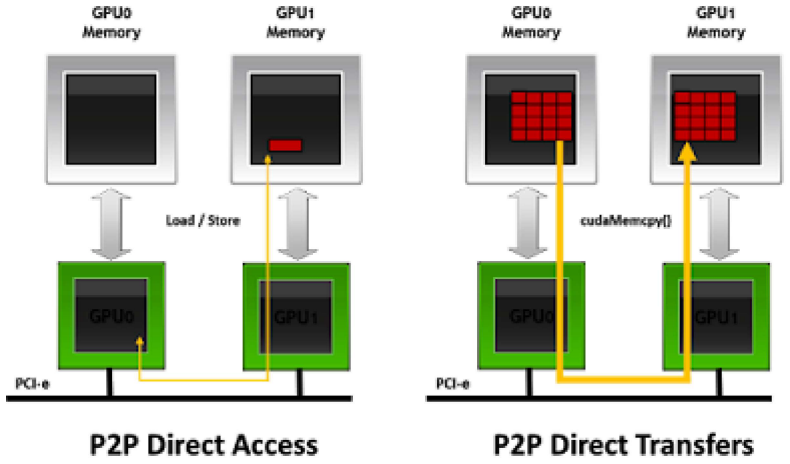
5. CPU – GPU – FPGA interconnect

The computing cores (ALUs) of the GPU0 can read the memory locations from the GPU1 direct, without accessing the CPU, by using GPU-Direct 2.0.

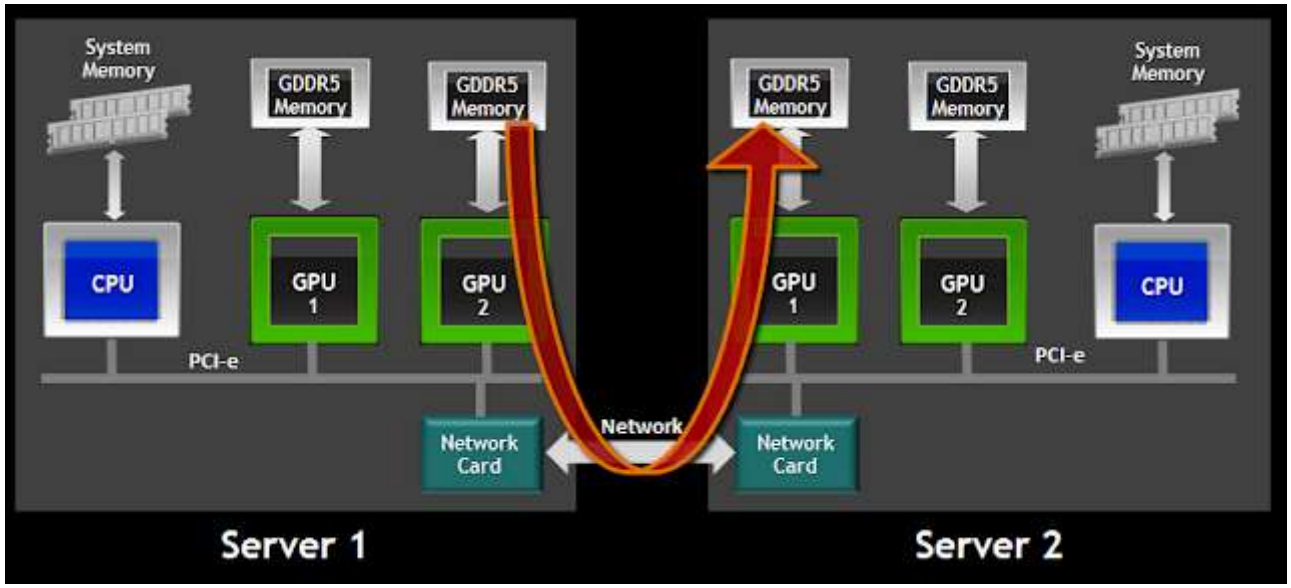
- **GPU Direct 1**



- **GPU Direct 2**



- **GPU Direct 3**

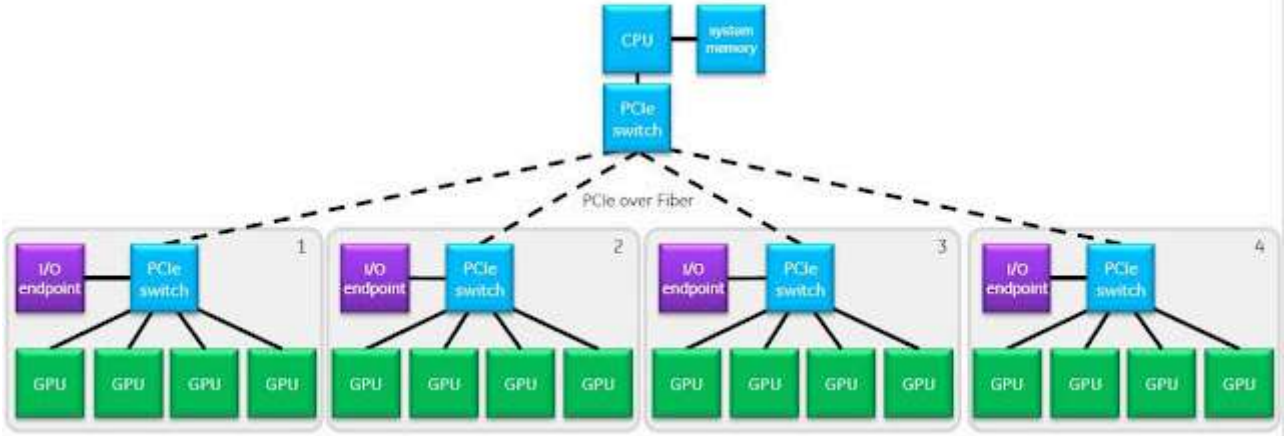


https://developer.nvidia.com/gpudirect

It is possible to connect tens and hundreds of GPUs into a single cascaded network by using a lot of PCI-Switches from PLX or IDT. (In this network there can also be hundreds of CPUs (Intel, PowerPC ...), FPGA, Ethernet, Infiniband ...).

PCIe over Fiber

- SWaP is our new limiting factor
- PCIe over Fiber-Optic can interconnect expansion blades and assure GPU:CPU growth
- Supports PCIe gen3 over at least 100 meters



GFLOPS/watt				
GPU:CPU ratio	1 to 1	4 to 1	8 to 1	16 to 1
SGEMM	8.45	10.97	11.61	11.96
DGEMM	3.61	4.64	4.91	5.04

imagination at work

http://on-demand.gputechconf.com/gtc/2013/presentations/S3266-GPUDirect-RDMA-Green-Multi-GPU-Architectures.pdf

(Page 16)

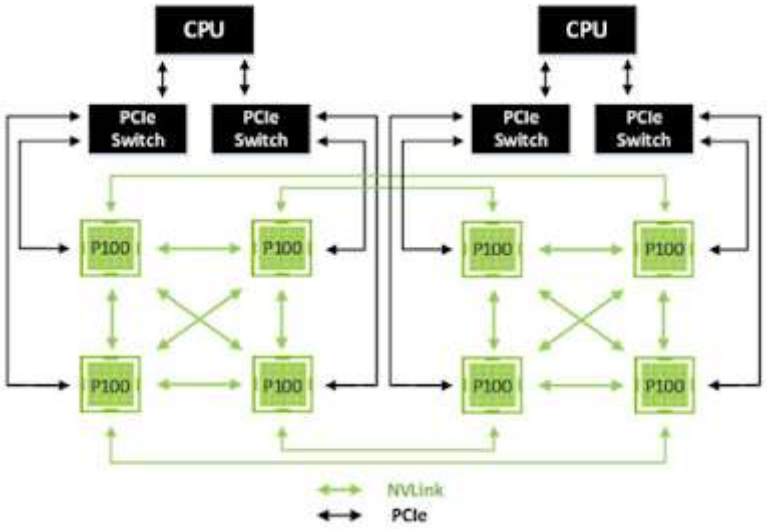
Functions for initializing direct access from the GPU to the CPU memory, into which the memory of any devices connected to the CPU can be mapped: FPGA, Ethernet, ...

http://people.maths.ox.ac.uk/gilesm/cuda/MultiGPU_Programming.pdf

(Page 14)

Connection scheme of several CPUs and GPUs in one HPC computer nVidia DGX-1:

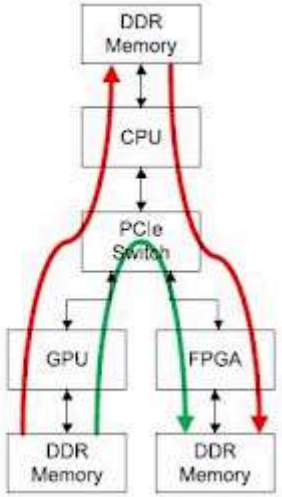
- **Black** - PCI Express connections; any CPU and GPU devices are connected
- **Green** - NVLink connections; 8 x nVidia GPU P100 (or new CPU Power PC) only



http://www.anandtech.com/show/10229/nvidia-announces-dgx1-server

PCI Express - direct exchange among CPU, GPU and FPGA is carried out by direct data transfer without occupying the interface of neighboring modules and without interfering with them.

Fig. 1 Two conceptual models of GPU to FPGA transfers



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

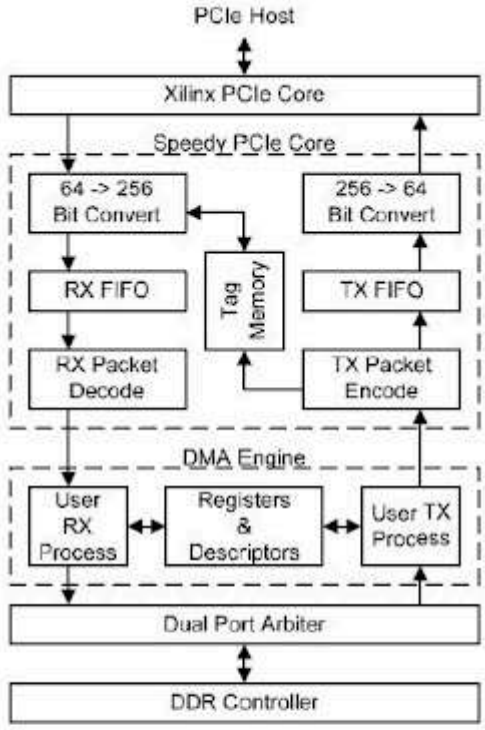


Fig. 2 Speedy PCIe core

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

https://www.microsoft.com/en-us/research/wp-content/uploads/2013/05/20130513-Cluster-Computing-Journal-Springer-Final-10.1007_s10586-013-0280-9.pdf

Through what PCI Express BARs the memory windows for the exchange are configured:

- **P2P** (GPU Direct 2.0) – for GPU-GPU exchange
- **RDMA** (GPU Direct 3.0) – for exchange between GPU and any other device: FPGA, Ethernet

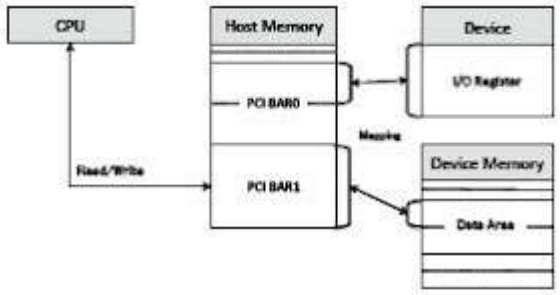


Fig. 5. Memory-mapped read and write.

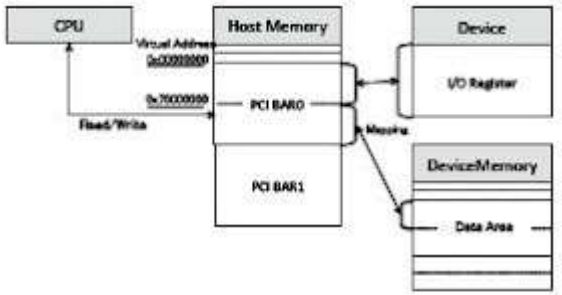


Fig. 6. Memory-window read and write.

RDMA

P2P

The GPU directly uses the Ethernet-adapter via PCIe for efficient network traffic processing on modern graphics hardware.

A prerequisite for the GPU to read the data direct from the Ethernet-adapter, the physical memory of which is mapped to the physical addressing of the CPU, - is that this memory should be mapped to virtual CPU addressing from there by using vm_insert_page instead of remap_pfn_range.

Processing is carried out like in case of Netmap: https://github.com/luigirizzo/netmap

tion has to be non-pageable and also it must be added to `nvidia` driver's internal tracking mechanism. For CUDA v3.2 or lower this meant that the memory had to be allocated using the API provided by the framework (`cudaMallocHost`). With the release of CUDA v4.0 the addition of the `cudaHostRegister` function call gave a solution. Now memory regions can be registered as memory eligible for accelerated DMA transfers. Buffers that that do not fulfill these requirements are first copied to some staging memory, which slows things down significantly. Before the new functionality came out, a workaround was used in this work. A dummy buffer of the same size with the target buffer was allocated using the proprietary CUDA API, and the page table of the process was altered to make the dummy buffer point to the physical pages of the original target buffer. With this hack, the GPU was tricked to perform accelerated transfers from/to generic non-pageable memory. The memory translation and re-mapping facilities were provided by a Linux kernel module that was developed for this exact purpose.

Either way, the zero-copy approach needs the packet storage of the capturing framework to be registered as GPU buffers. In the case of netmap's exposed NIC rings this was not possible "out-of-the-box". The problem seems to originate from the `nvidia` driver and the flags and/or structure associations it expects to be present on the virtual memory area passed to `cudaHostRegister`. However, the driver comes as a binary blob with no source code available. In order to overcome this problem, the netmap kernel module, and more specifically the code that implements the `mmap(3)` functionality, was modified. The patch simply changes the routines used to export the allocated pages to userspace. Apparently, the problem here is the use of `remap_pfn_range` which produces raw PFN (Page Frame Number) mappings and the pages do not have a `struct page` associated with them. Using `vm_insert_page` for every page instead works.

[2] `http://developer.download.nvidia.com/compute/cuda/4_2/rel/toolkit/docs/online/sync_async.html`

As various CPU и GPU calculators can be combined into a single network, then we will briefly show the types of multithreading that are found on these calculators:

- **CPU** Intel Core/Xeon uses: **SMT** (for Hyper-Threading) and **CMP** (for Cores)
- **GPU** uses: **Fine-grained** (for Threads/Cores) and **CMP** (for Blocks/SM)

Multithreaded/Multicore Processors

MT Approach	Resources shared between threads	Context Switch Mechanism
None	Everything	Explicit operating system context switch
Fine-grained	Everything but register file and control logic/state	Switch every cycle
Coarse-grained	Everything but I-fetch buffers, register file and con trol logic/state	Switch on pipeline stall
SMT	Everything but instruction fetch buffers, return address stack, architected register file, control logic/state, reorder buffer, store queue, etc.	All contexts concurrently active; no switching
CMT	Various core components (e.g. FPU), secondary cache, system interconnect	All contexts concurrently active; no switching
CMP	Secondary cache, system interconnect	All contexts concurrently active; no switching

- Many approaches for executing multiple threads on a single die
  - Mix-and-match: IBM Power7 CMP+SMT

Mikko Lipasti-University of Wisconsin 9

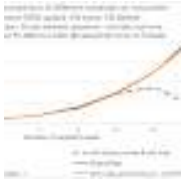
http://ece757.ece.wisc.edu/lect03-cores-multithread.pdf

Enter Comment

Popular posts from this blog

C++: Thread-safe std::map with the speed of lock-free map

April 27, 2017



Introduction: Examples of use and testing of a thread-safe pointer and contention-free shared-mutex. In this article we will show additional optimizations, examples of use and testing of a thread-safe pointer developed by us with an optimized shared mutex contfree_safe_ptr<T> - this is ec ...

READ MORE

C++: We make any object thread-safe

April 27, 2017

Introduction: Smart pointer that makes any object thread-safe for any operations, with the performance equal to that of optimized lock-free containers. In these 3 articles I'll tell in detail about atomic operations, memory barriers and the rapid exchange of data between threads, as well as about the "sequence-points" by the example of "execute-arour ...

READ MORE

Powered by Blogger

Theme images by Michael Elkan

ALEXEYAB

VISIT PROFILE

Archive