Accelerators and accelerated systems

046278/236278

Mark Silberstein
EE
Technion
GPU networking

- Motivation
- In-GPU networking: GPU-based network sockets
- In-GPU networking: RDMA from GPUs
Hardware capabilities

- **GPUDirectRDMA**
  - enable direct data transfer (Peer-to-Peer) between the NIC and the GPU memory
- Eliminate the CPU logic from data path
- Eliminate extra buffering in CPU
- *Control path is still via the CPU*
On-GPU servers
Why GPU servers?

- Task-parallel workload
- High throughput
- Lower latency for parallel tasks
Why not RDMA for servers?

- Potentially not a bad idea
  - Hot research topic actually!
- Requires knowing the **data size in advance**
  - Messages instead of data stream
- Too low level interface
  - We'd like to use sockets!
Why not CPU sockets?

Example: GPU-accelerated server using CPU sockets
CPU server

- NIC
- CPU
- Memory

Functions:
- recv()
- compute()
- send()
Inside a GPU-accelerated server

Theory:

recv()
GPU_compute()
send()
Inside a GPU-accelerated server

Theory

```c
recv();
GPU_compute();
send();
recv();
batch();
```

Improve PCIe transfer performance
Inside a GPU-accelerated server

\begin{center}
\begin{tikzpicture}
  \node[draw, rectangle] (cpu) at (0,0) {CPU};
  \node[draw, rectangle, fill=blue!20] (nic) at (-3,-1.5) {NIC};
  \node[draw, rectangle, fill=blue!20] (mem) at (-3,-0.5) {Memory};
  \node[draw, rectangle, fill=green!50!black] (gpu) at (3,0) {GPU};
  \node[draw, rectangle, fill=green!50!black] (mem_gpu) at (3,-0.5) {Memory};

  \draw[->] (nic) -- (mem);
  \draw[->] (mem) -- (cpu);
  \draw[->] (cpu) -- (mem_gpu);
  \draw[->] (mem_gpu) -- (gpu);

  \node[draw, rectangle, fill=lightblue] at (0,-4) {Theory};
  \node[draw, rectangle, fill=lightblue] (recv) at (0,-4.5) {recv()};
  \node[draw, rectangle, fill=green!50!black] (gpu_compute) at (0,-5) {GPU\_compute()};
  \node[draw, rectangle, fill=lightblue] (send) at (0,-5.5) {send()};

  \node[draw, rectangle, fill=lightblue] (batch) at (3,-3) {batch();
    optimize();
    transfer();}

\end{tikzpicture}
\end{center}
Inside a GPU-accelerated server

```plaintext
invoke();
CPU
NIC
Memory
Memory
recv();
batch();
optimize();
transfer();
balance();
GPU_compute();
recv();
GPU_compute()
```

Theory

```plaintext
recv()
GPU_compute()
send()
```
Inside a GPU-accelerated server

```c
transfer();

CPU

Memory

GPU

Memory

NIC

recv();

batch();

optimize();

transfer();

balance();

GPU_compute();

recv();

send();

recv();

GPU_compute();

transfer();

cleanup();
```

Theory
Inside a GPU-accelerated server

send();

CPU

GPU

Memory

Memory

recv();

batch();

optimize();

transfer();

balance();

transfer();

cleanup();

dispatch();

send();

recv();

GPU_compute();

Theoretical
code:
recv();
batch();
optimize();
transfer();
balance();
GPU_compute();
transfer();
cleanup();
dispatch();
send();
Inside a GPU-accelerated server

Aggressive pipelining
Buffering, asynchrony, multithreading

recv();
batch();
optimize();
transfer();
balance();
GPU_compute();
transfer();
cleanup();
dispatch();
recv();
batch();
optimize();
transfer();
balance();
GPU_compute();
transfer();
cleanup();
dispatch();
send();
recv();
batch();
optimize();
transfer();
balance();
GPU_compute();
transfer();
cleanup();
dispatch();
send();
Where is the problem?

- Why don't we have these issues in CPU?
- Which part of the system takes care of them?
Sockets on GPU

- recv()
- GPU_compute()
- send()

GPUDirectRDMA
Sockets on GPU

No CPU (in data path)

NIC

GPU

Memory

PCIe bus

recv()

GPU_compute()

send()
Sockets on GPU

No request batching

recv()  send()

NIC

Memory

recv()

GPU_compute()

send()
Sockets on GPU

Transparent pipelining

Seamless buffer management

recv() → send()

recv() → send()

recv() → send()

recv() → send()
Designing a socket abstraction for GPUs
Goals

Simplicity
Reliable streaming abstraction for GPUs

Performance
NIC → GPU
Use: P2PDMA
Goals

**Simplicity**
Reliable streaming abstraction for GPUs

**Performance**
NIC → GPU
Use: P2PDMA

Problem: where to implement transport layer?
What does transport layer mean?

- Data abstraction: message or stream
- Reliability: ensure delivery of all sent data
  - or not, if unreliable
- Flow control: stop sending if receive buffers are full
- Multiplexing: support multiple concurrent applications/threads
Design option 1: Transport layer processing on CPU

Memory

Data must be received in CPU

GPU controls the flow of data

recv()
Design option 1: Transport layer processing on CPU

- Extra CPU-GPU memory transfers
Design option 2: Transport layer processing on GPU

- CPU
- Memory
- GPU
  - `recv()`
  - Transport processing
- Network buffers
- P2P DMA
- NIC
Design option 2: Transport layer processing on GPU

CPU
recv()

GPU
Transport processing
Network buffers

TCP/IP on GPU?
P2P DMA

NIC

CPU applications access network through GPU?
Not CPU, Not GPU

We need help from NIC hardware
RDMA: partially offloading transport layer processing to NIC

Diagram:
- CPU: Streaming
  - Message buffers
- GPU: Streaming
  - Message buffers
- Reliable RDMA
  - NIC
GPUnet layers

- GPU Socket API
- Reliable in-order streaming
- Reliable channel
- RDMA Transports
  - Infiniband
- Use RDMA RC
- Streaming over RDMA
- recv/send buffers multiple sockets
GPUnet layers

- GPU Socket API
- Reliable in-order streaming
- Reliable channel

RDMA Transports
- Infiniband

Non-RDMA Transports
- UNIX Domain Sockets, TCP/IP
Full compatibility with the CPU endpoints

node0.technion.ac.il

**GPU native** server

```c
socket(AF_INET,SOCK_STREAM);
listen(:2340)
```

GPUnet

CPU client

```c
socket(AF_INET,SOCK_STREAM);
connect("node0:2340")
```

GPUnet

**GPU native** client

```c
socket(AF_INET,SOCK_STREAM);
connect("node0:2340")
```
Sockets on GPU: details
Semantics

- Socket is an end-point for connection-based communications
- Provides *streaming* abstraction
- Has recv/send system buffers, flow control counters
  - If no place left in recv buffer, blocks the sender

- Shared among all the GPU threads
- **Socket API** calls: synchronization point
  - Invoked by all the threadblock threads at once!
Combining GPU Data path and CPU Control

- NIC (HCA)
- NIC driver send/recv API
- GPUdirectRDMA
- PCIe
- GPU Memory
- Send/recv buffers
- CPU Memory
How does NIC know it should start doing something?
How does NIC know it should start doing something?

- Relay via the CPU! CPU runs a “network service server”
- GPU calls “recv()”
  - GPU writes to an RPC queue in CPU-GPU shared memory
  - CPU \textit{daemon thread} picks up
  - CPU Issues (non-blocking) recv()
  - GPU polls for completion
  - CPU updates the RPC queue
  - Call completes
- Same for send/open/close/shutdown
Implementing flow control

- Sender keeps track of the amount of empty space in the *receive* buffer

- Receiver: report successful recv to sender
The anatomy of recv()
The anatomy of recv()
Using GPUnet for building a server
Different GPU server architectures

GPU-accelerated server

CPU
- Daemon Thread
- Worker Thread

GPU
- Worker TB

accept
send
compute
send
copy
send
close

GPUNet server
Daemon architecture

GPU
- Daemon TBs
- Worker TBs

accept
recv
compute
send
close

GPUNet server
Independent architecture

GPU
- Server TBs

accept
recv
compute
send
close
GPU servers – new optimization challenges

- How many TBs to allocate for I/O, how many for worker TBs?

- Matrix product server

- Light/Medium/Heavy inputs: 64x64, 256x256, 1024x1024

- Light/Medium/Heavy configs: 16, 8, 4 daemon TBs

Performance degradation due to misconfiguration

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Light</th>
<th>Workload Medium</th>
<th>Heavy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light</td>
<td>92%</td>
<td>81%</td>
<td>74%</td>
</tr>
<tr>
<td>Medium</td>
<td>44%</td>
<td>99%</td>
<td>88%</td>
</tr>
<tr>
<td>Heavy</td>
<td>12%</td>
<td>44%</td>
<td>100%</td>
</tr>
</tbody>
</table>
Face verification server

CPU client (unmodified) via rsocket

GPU server (GPUnet)

memcached (unmodified) via rsocket

Infiniband

recv()
GPU_features()
query_DB()
GPU_compare()
send()
Face verification: NVIDIA CUDA (no batching)

Throughput (KReq/sec)

Latency (μsec)

1 GPU (no GPUnet)

99th%

25th-75th%

Median

23
Face verification: Different implementations

Throughput (KReq/sec)

Latency (μsec)

1 GPU (no GPUnet)

CPU 6 cores

1 GPU GPUnet

1.9x throughput
1/3x latency
(500usec)
½ LOC
Face verification: Different implementations

Throughput (KReq/sec)

Latency (μsec)

Large variability in latency
Face verification on all processors
2xGPU + 10xCPU

Throughput (KReq/sec)

Latency (μsec)

CPU 6 cores

1 GPU
GPUnet

Similar latency
4.5x throughput

2xGPUnet + 10xCPU

Latency optimized

Throughput optimized

164

186