GPUnet: networking abstractions for GPU programs

Mark Silberstein
Technion – Israel Institute of Technology

Sangman Kim, Seonggu Huh, Xinya Zhang
Yige Hu, Emmett Witchel
University of Texas at Austin

Amir Wated
Technion

What

A socket API for programs running on GPU

Why

GPU-accelerated servers are hard to build

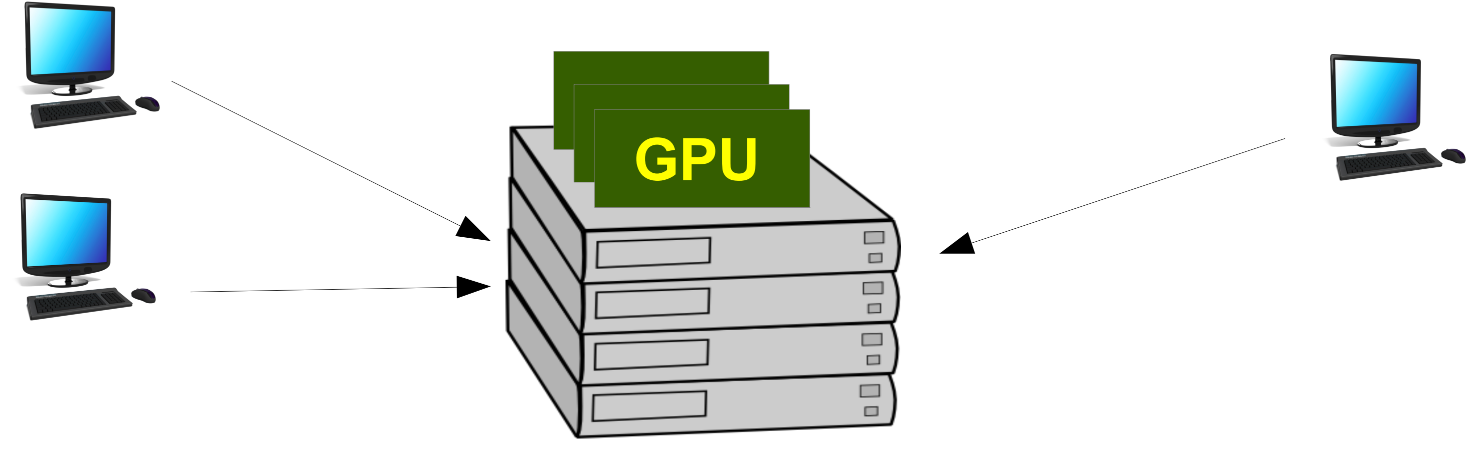
Results

GPU vs. CPU

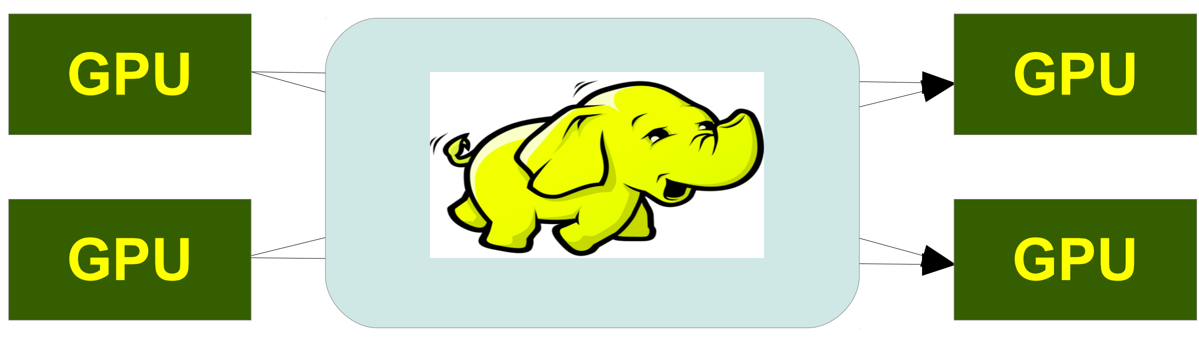
50%↑ throughput, 60%↓ **latency**, ½ LOC

Motivation: GPU-accelerated networking applications

Data processing server



MapReduce



Recent GPU-accelerated networking applications

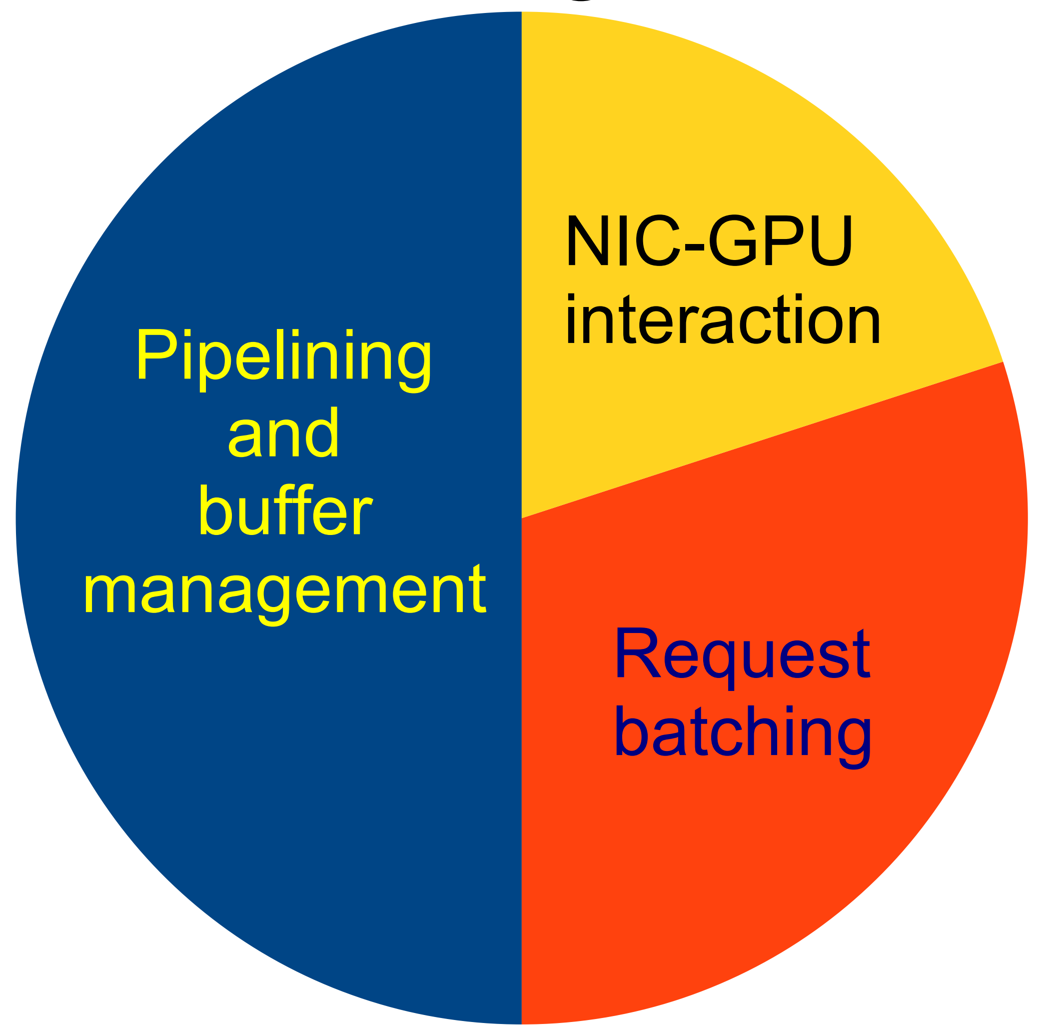
SSLShader (Jang 2011), GPU MapReduce (Stuart 2011), Deep Neural Networks (Coates 2013), Dandelion (Rossbach 2013), Rhythm (Agrawal 2014) ...

Recent GPU-accelerated networking applications

SSLShader (Jang 2011), GPU MapReduce (Stuart 2011), Deep Neural Networks (Coates 2013), Dandelion (Rossbach 2013), Rhythm (Agrawal 2014) ...

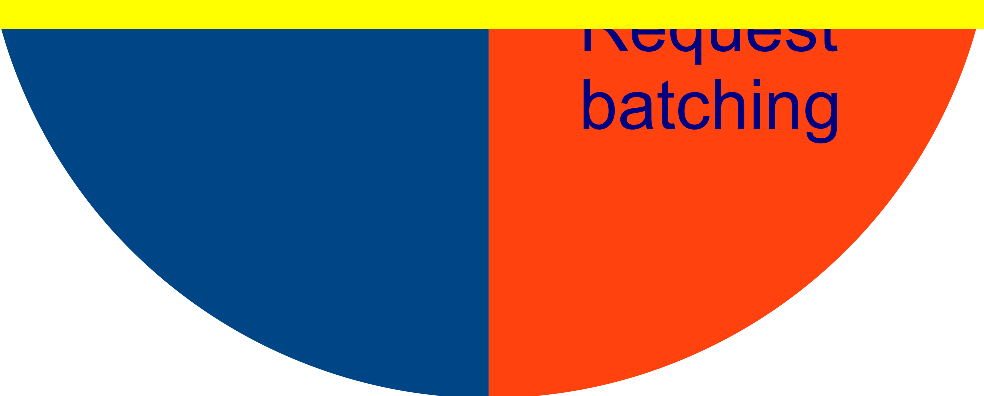
required **heroic** efforts

GPU-accelerated networking apps: Recurring themes

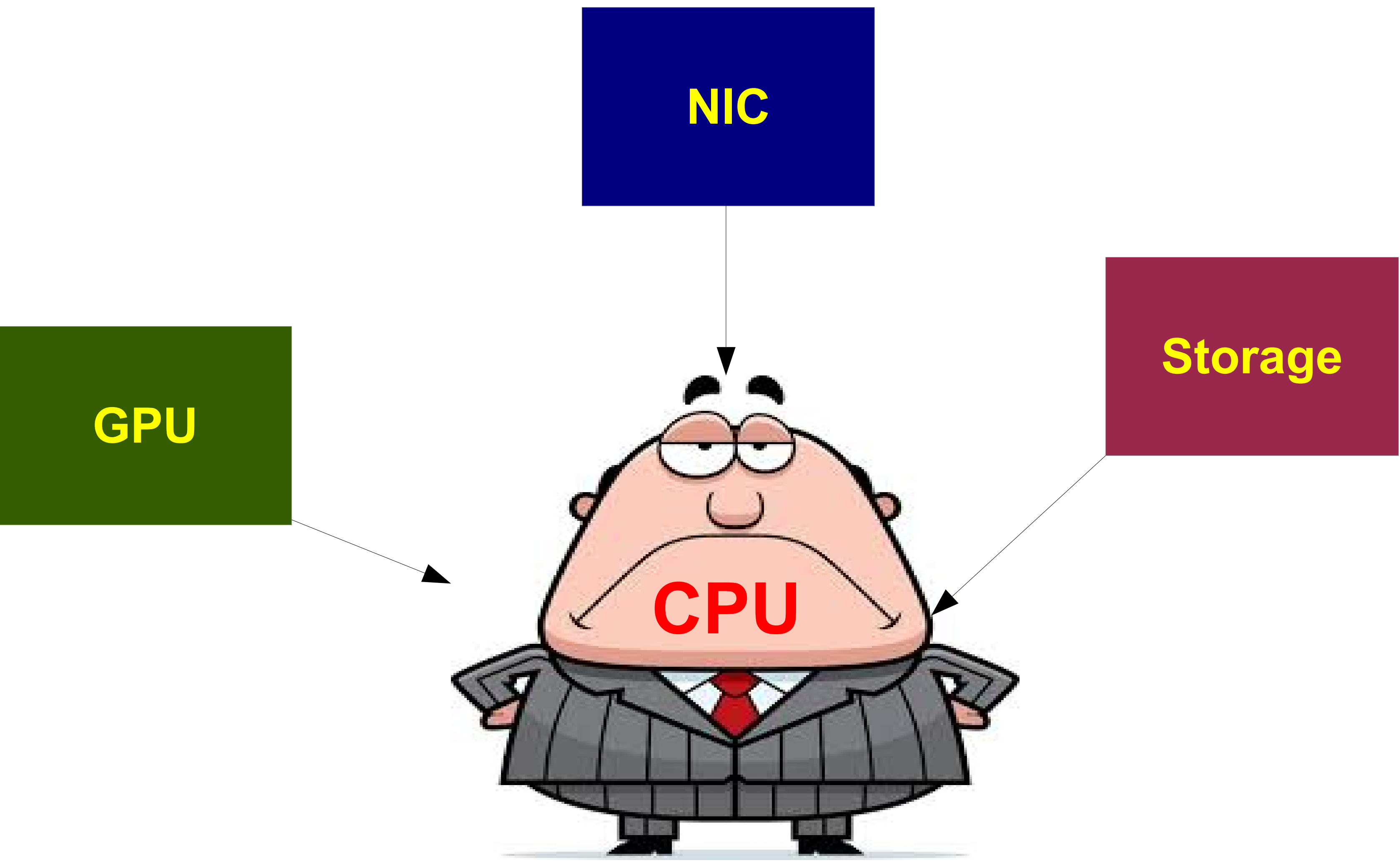


GPU-accelerated networking apps: Recurring themes

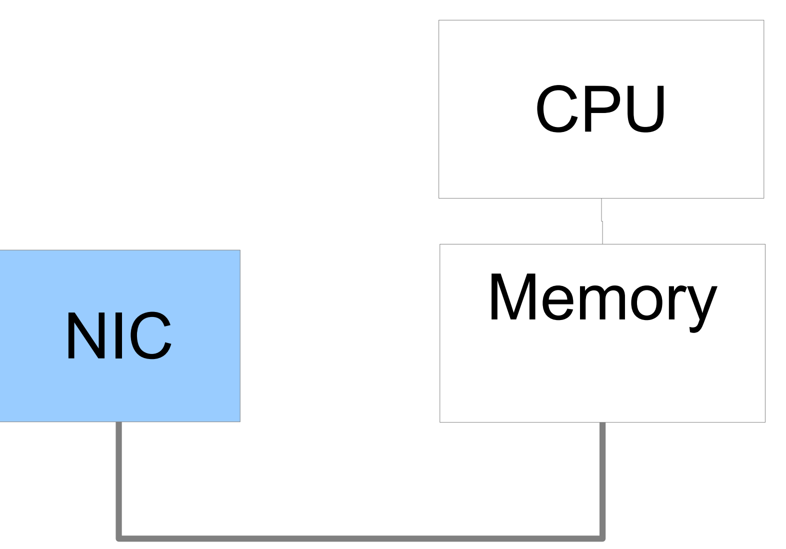
We will sidestep these problems



The **real** problem:
CPU is the ***only*** boss

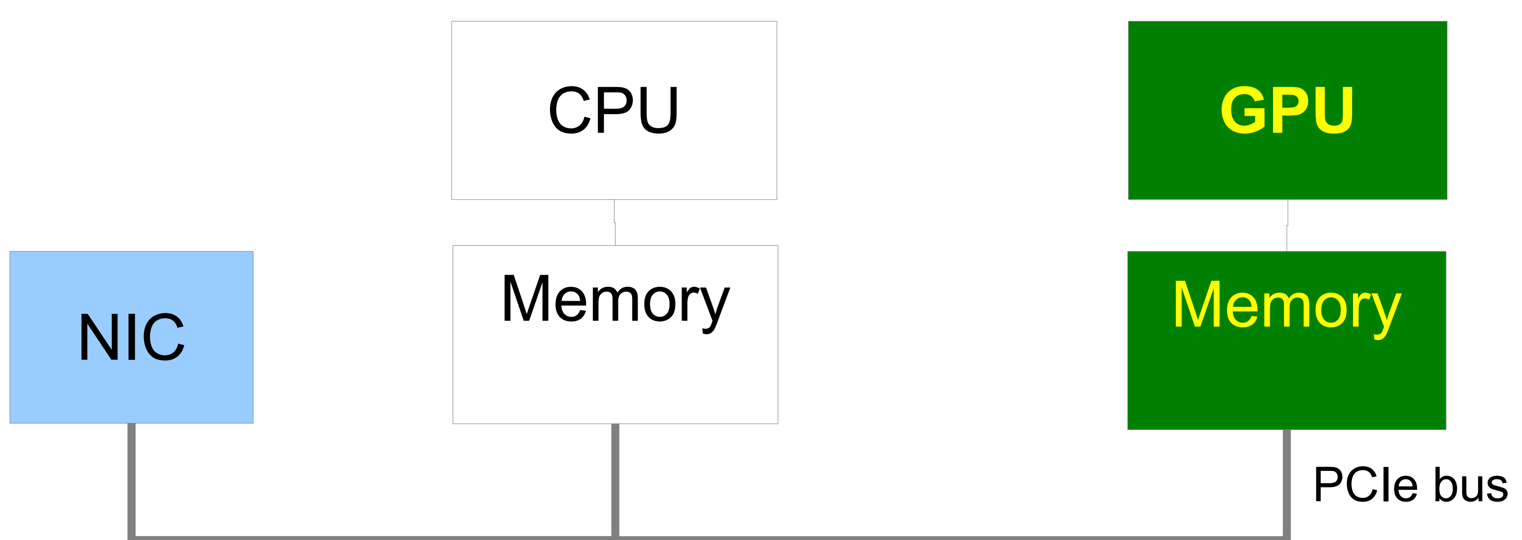


Example: CPU server




```
recv()
compute()
send()
```

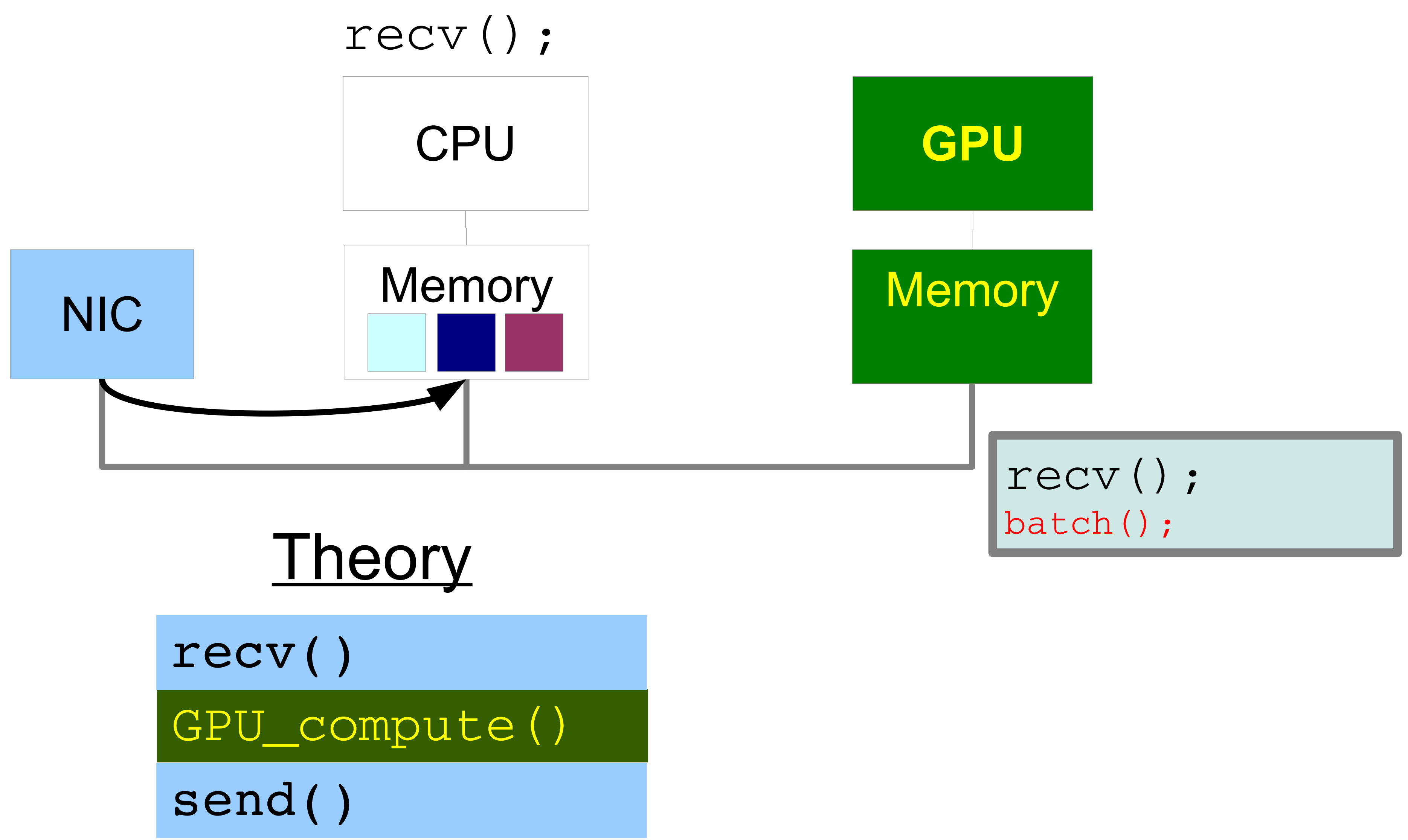
Inside a GPU-accelerated server



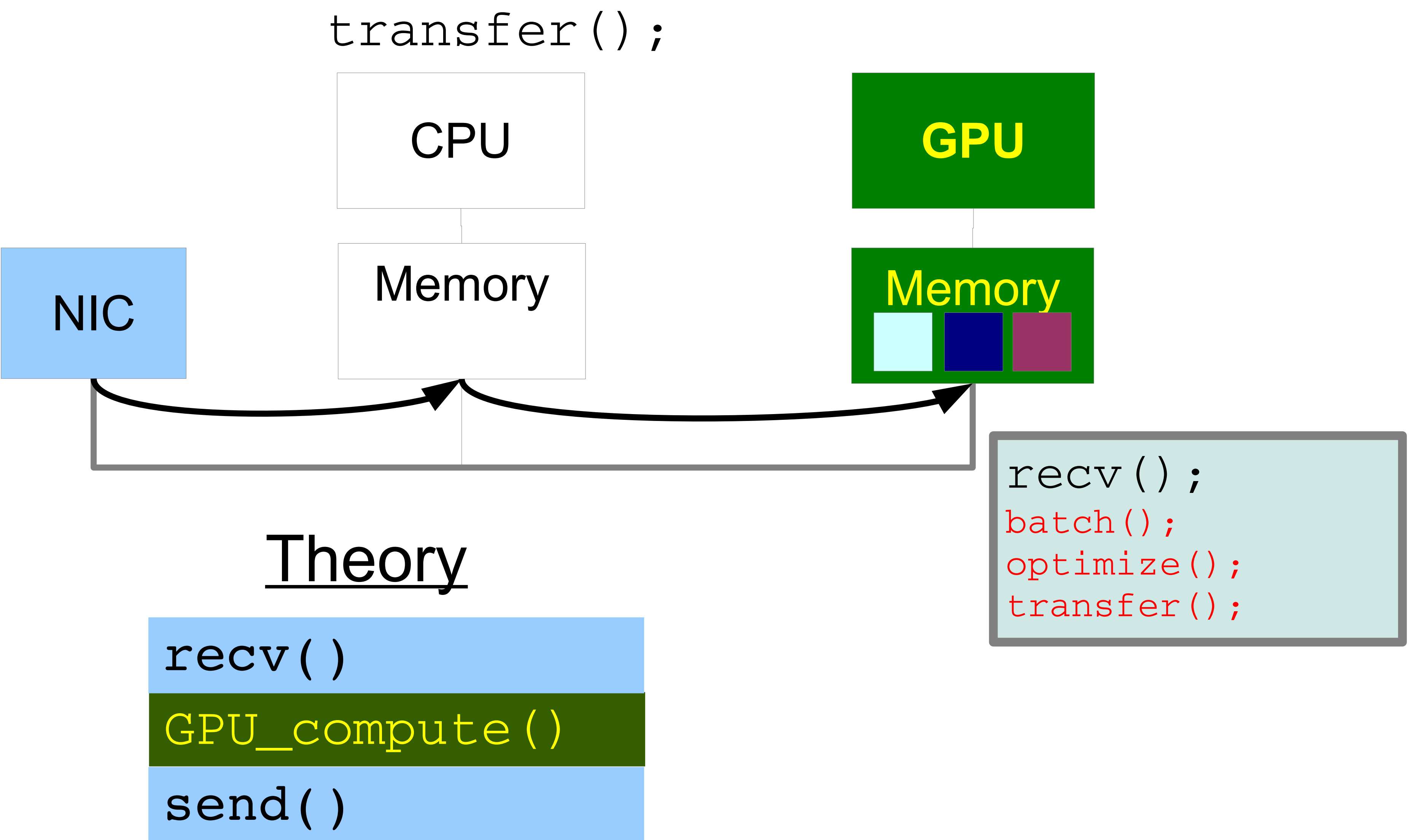
Theory

```
recv()
GPU_compute()
send()
```

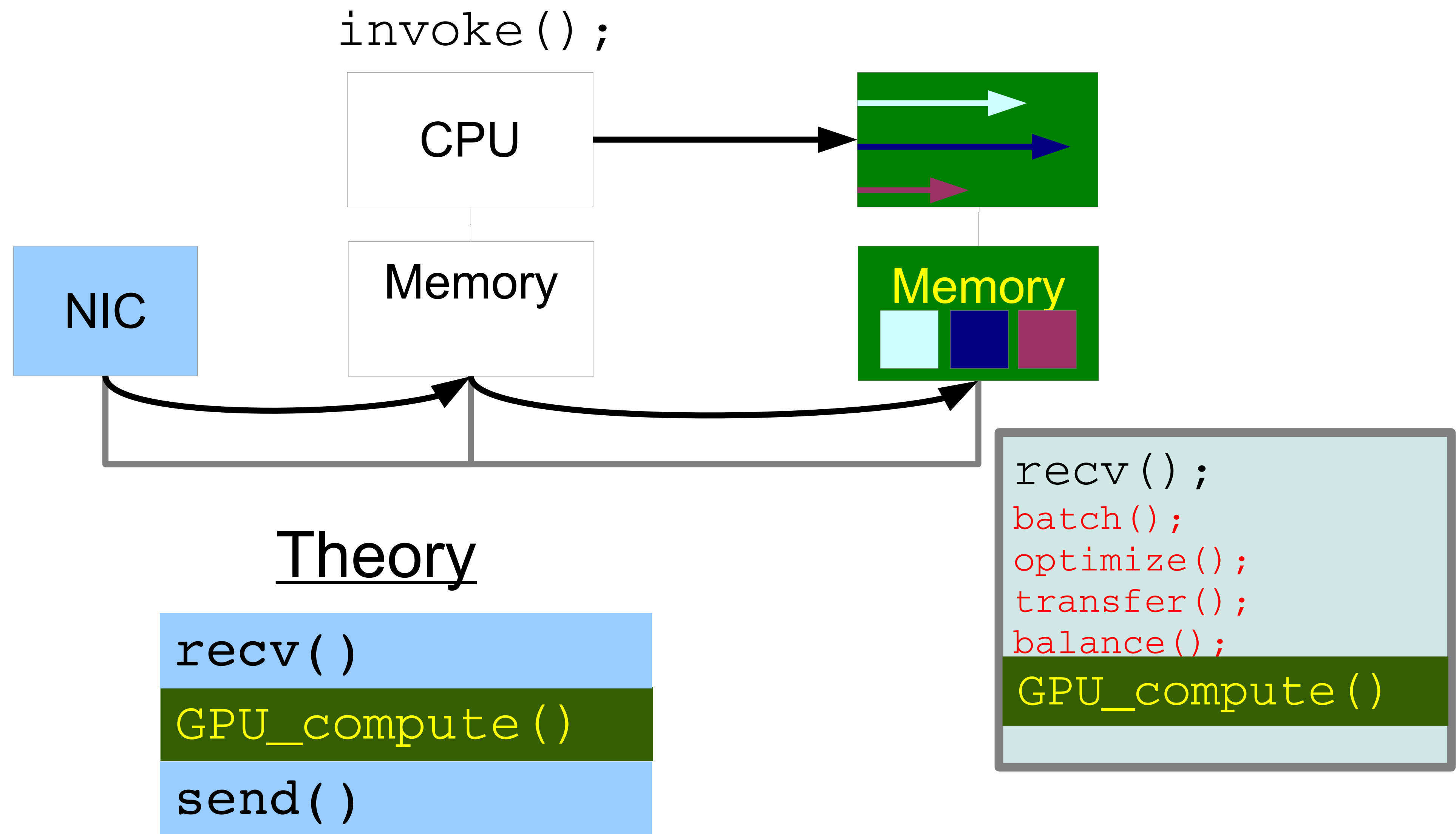
Inside a GPU-accelerated server



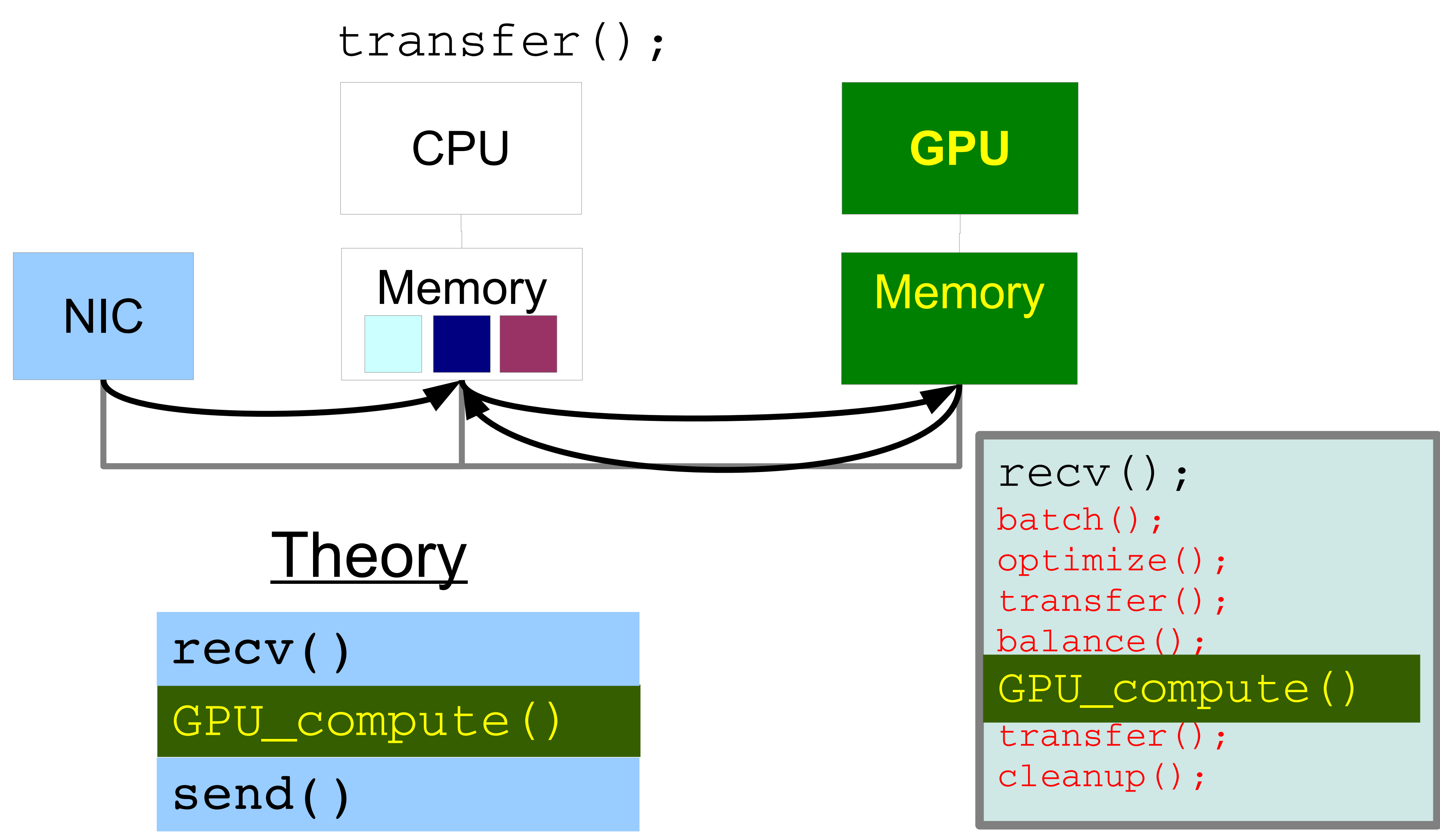
Inside a GPU-accelerated server



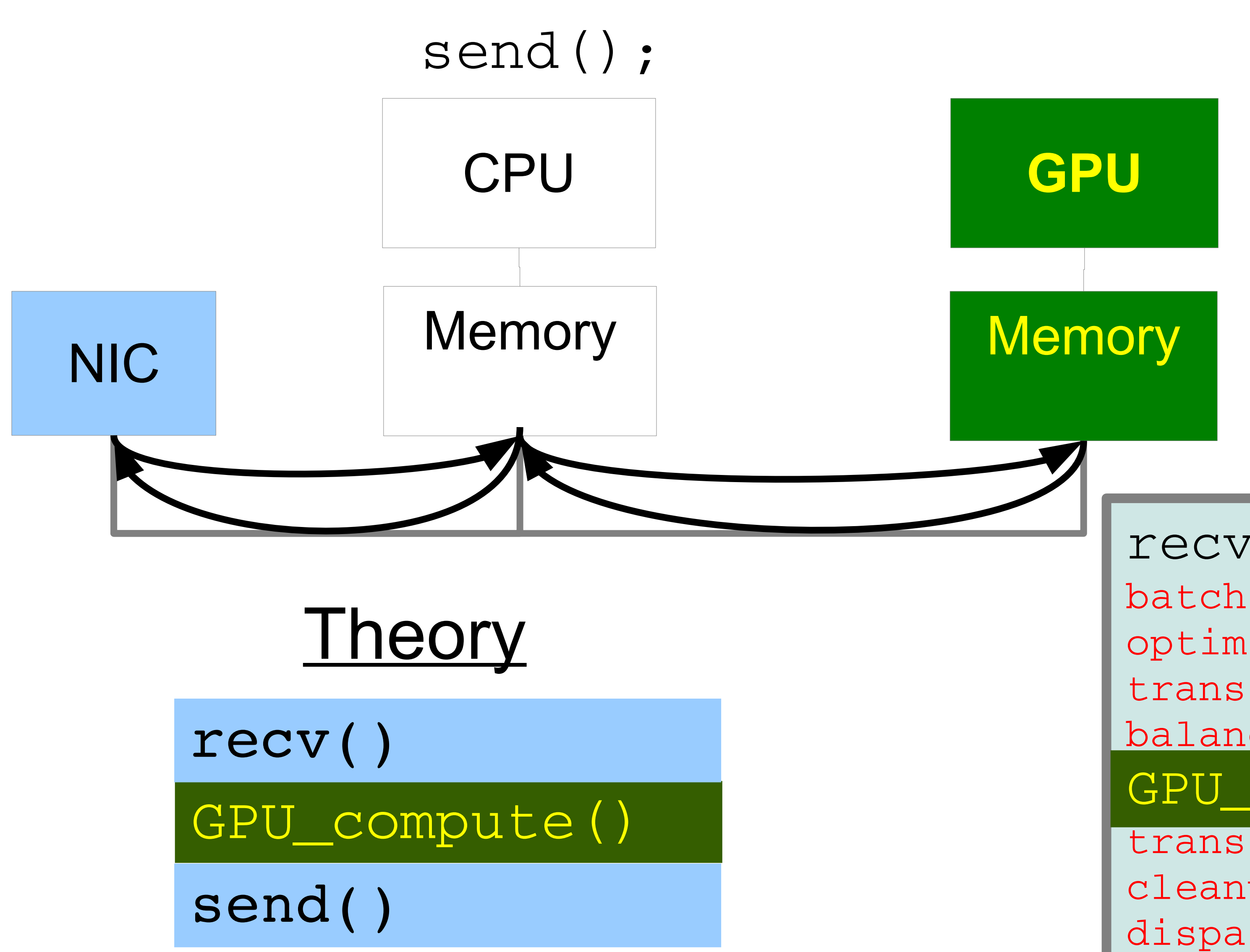
Inside a GPU-accelerated server



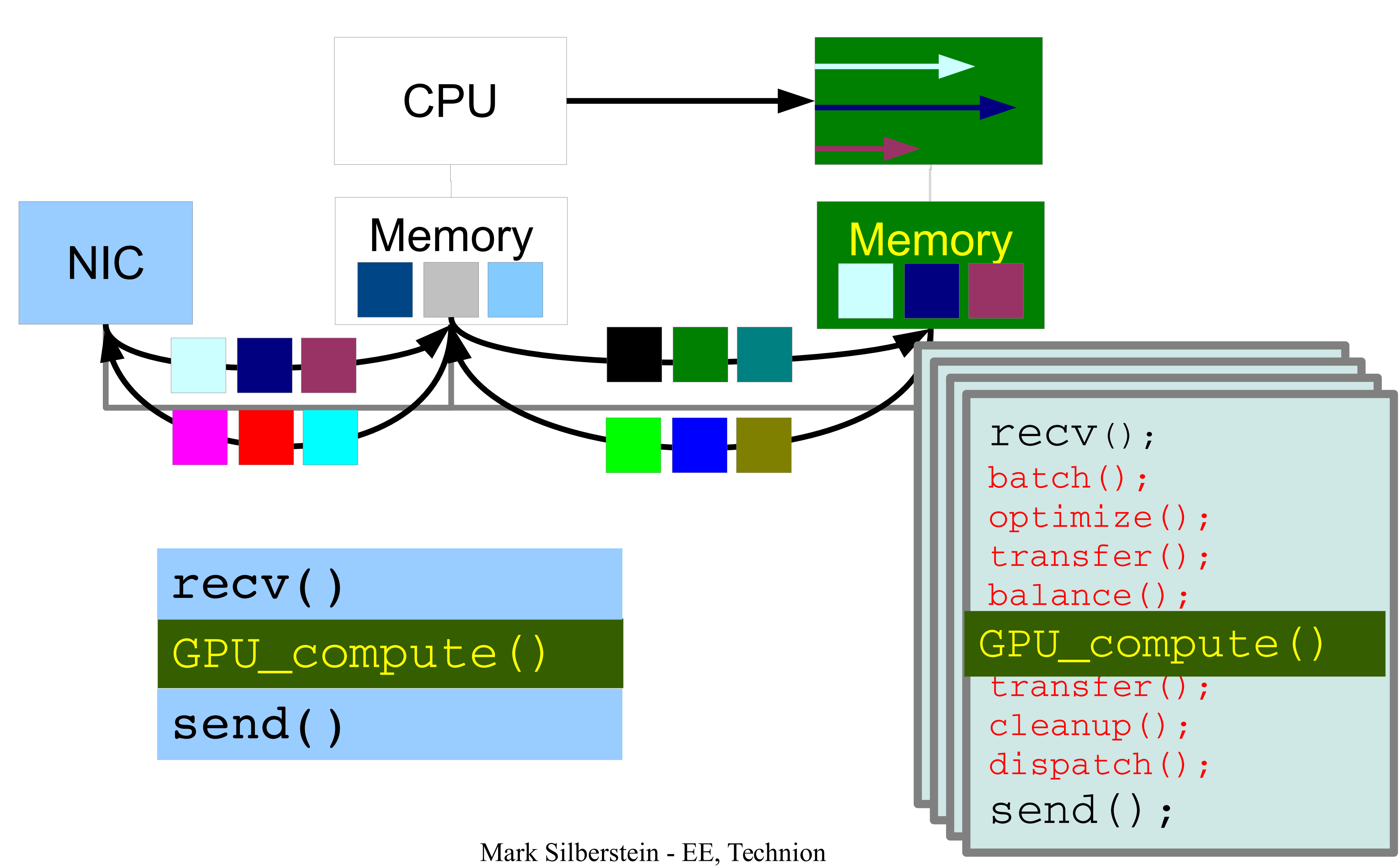
Inside a GPU-accelerated server



Inside a GPU-accelerated server



Aggressive pipelining
Double buffering, asynchrony, multithreading



This code is for a CPU to manage a GPU

```
batch();
optimize();
transfer();
balance();
GPU_compute()
transfer();
cleanup();
dispatch();
```

GPUs are not ***co-***processors

GPUs are **peer-**processors

They need I/O **abstractions**

File system I/O – [GPUfs ASPLOS13]
Network I/O – this work

GPUnet: socket API for GPUs
Application view

node0.technion.ac.il

GPU ***native*** server

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

GPUnet

Network

CPU client

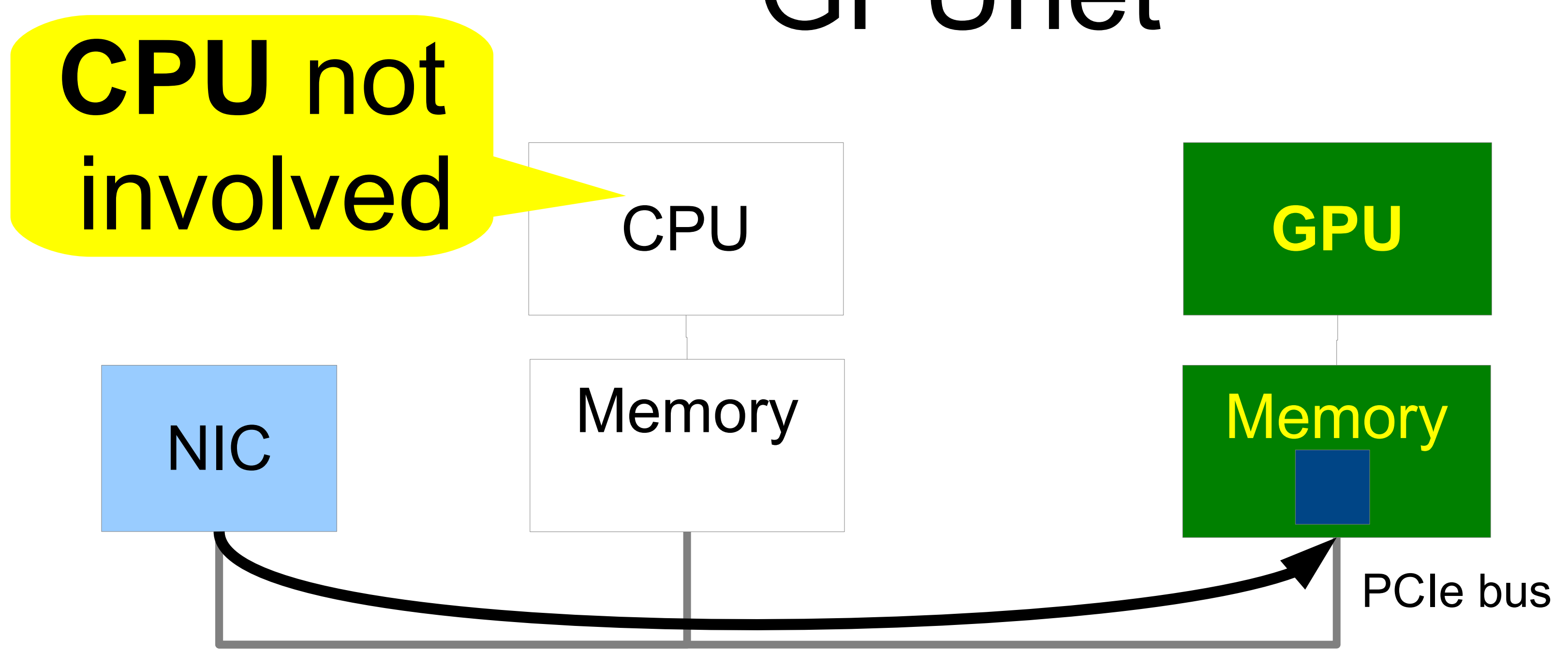
```
socket(AF_INET,SOCK_STREAM);
connect("node0:2340")
```

GPU ***native*** client

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

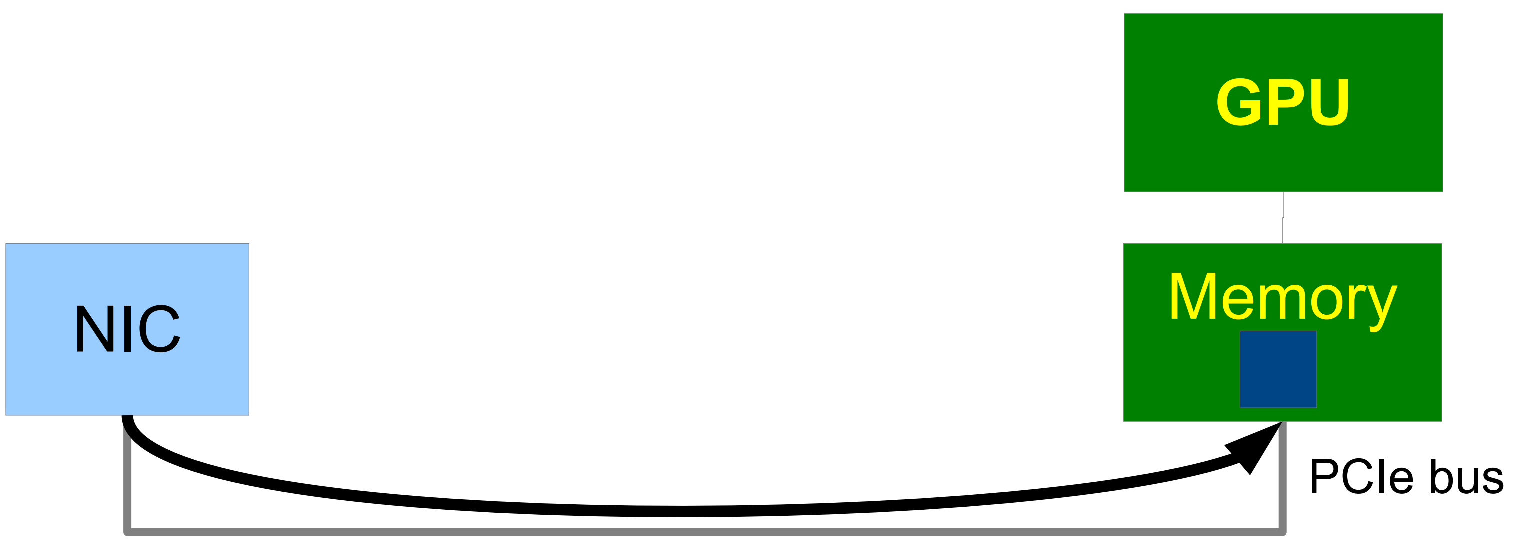
GPUnet

GPU-accelerated server with GPUnet




```
recv()
GPU_compute()
send()
```

GPU-accelerated server with GPUnet




```
recv()
GPU_compute()
send()
```

GPU-accelerated server with GPUnet

No request batching

recv()

send()

NIC

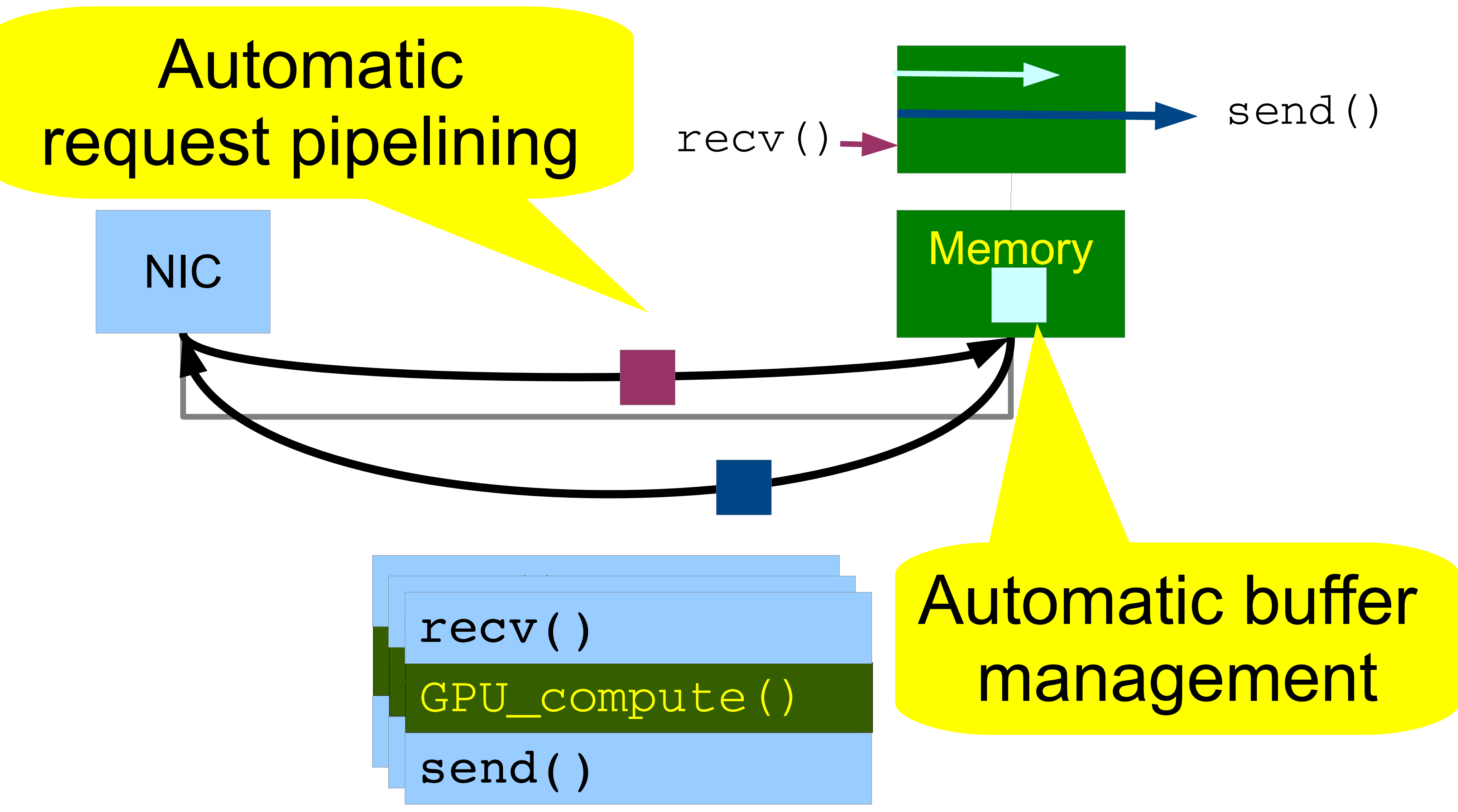
Memory

recv()

GPU_compute()

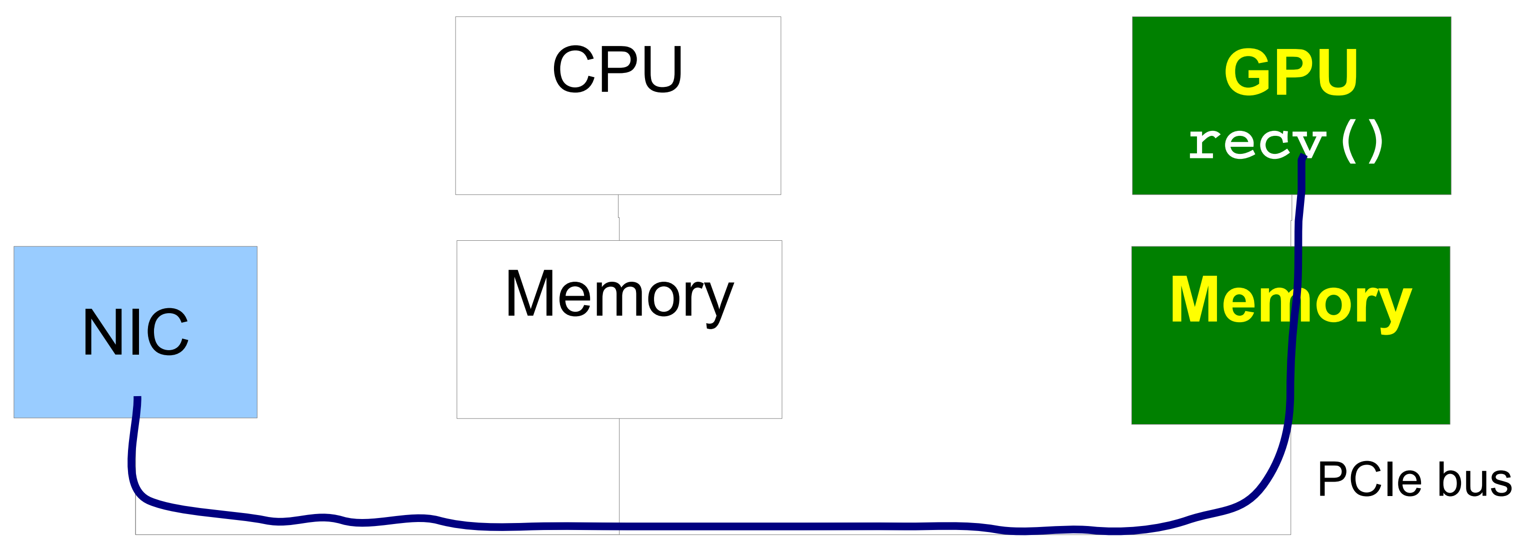
send()

GPU-accelerated server with GPUnet



Building a socket abstraction for GPUs (High level points)

Goals



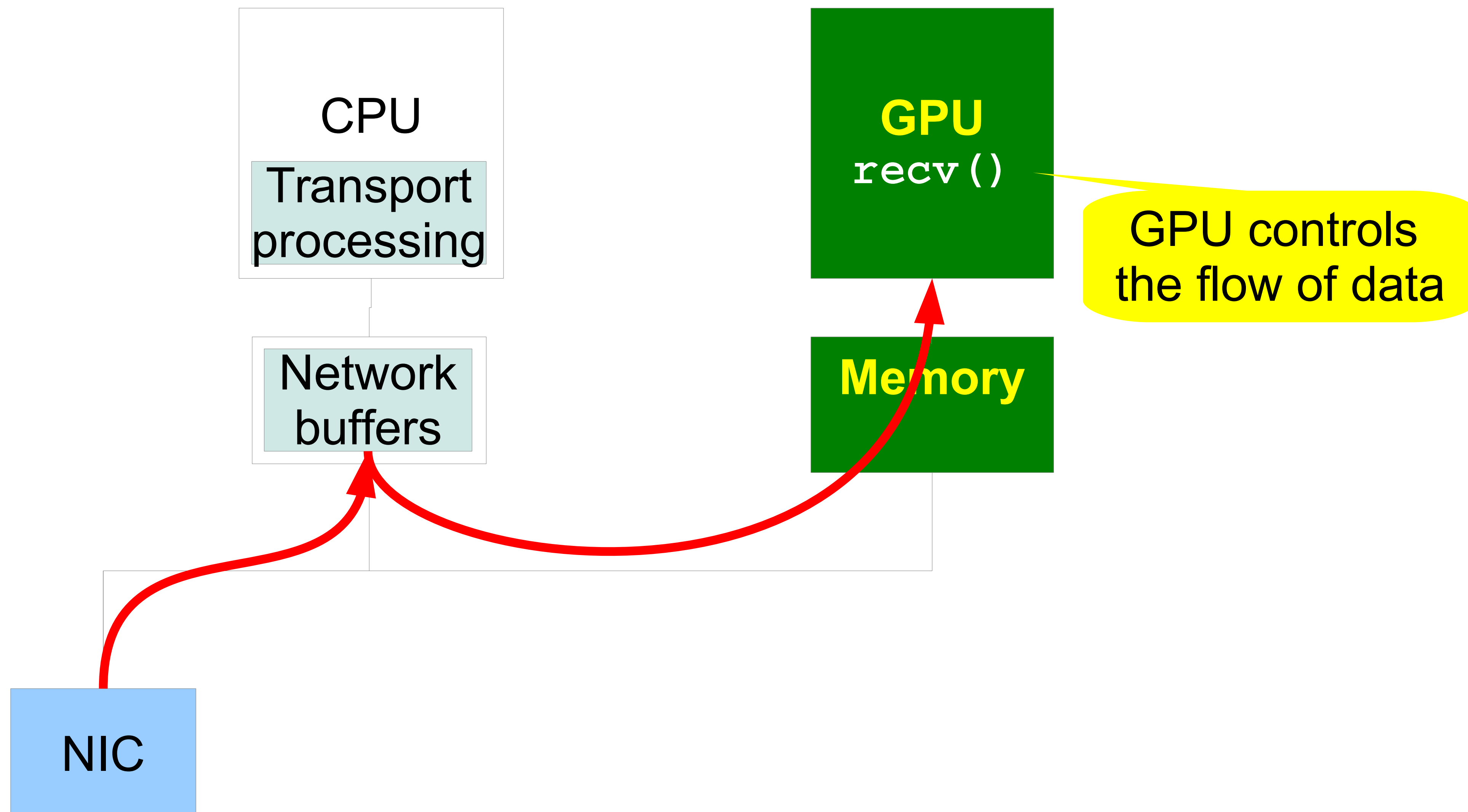
Simplicity

Reliable streaming abstraction for GPUs

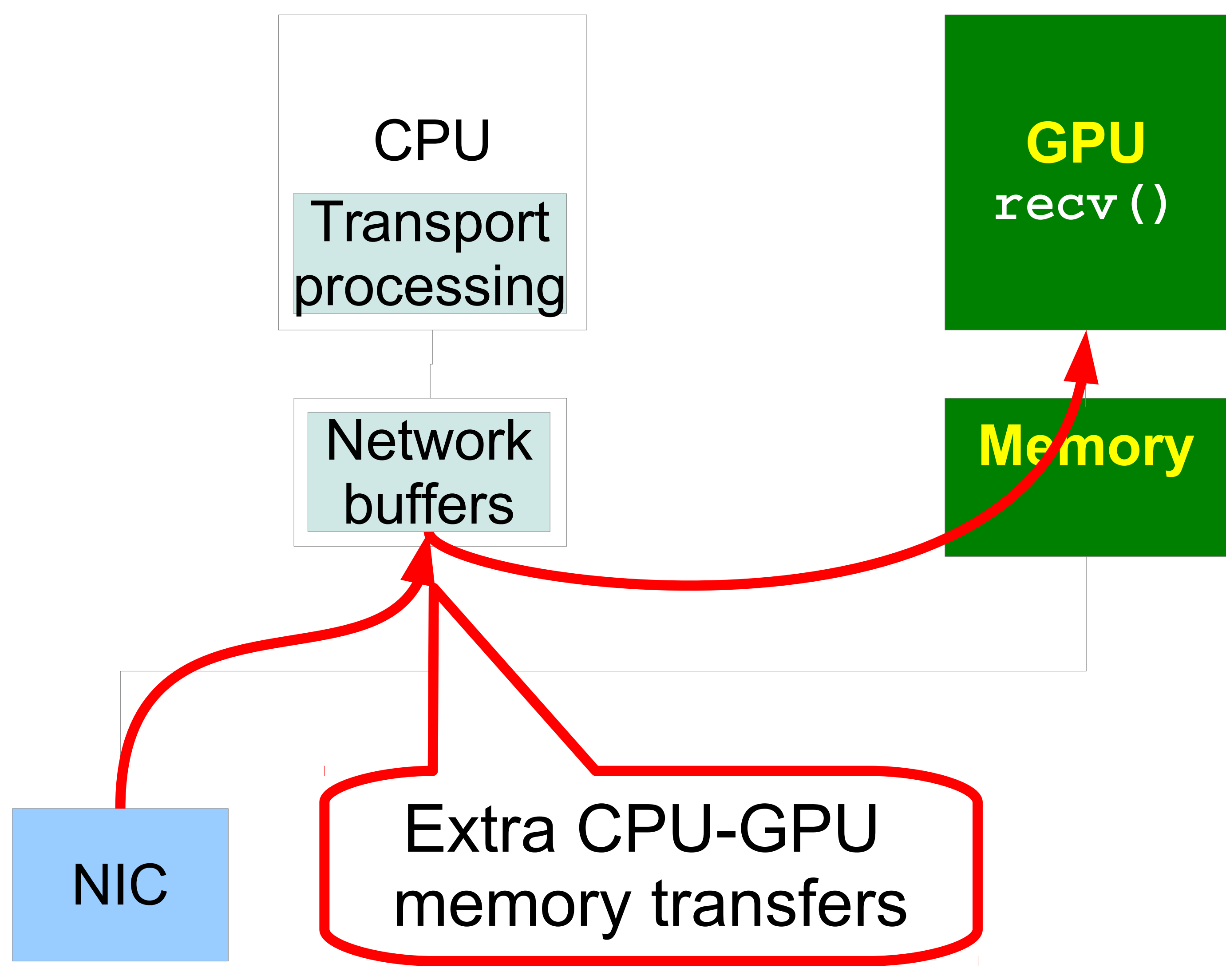
Performance

NIC → GPU data path optimizations

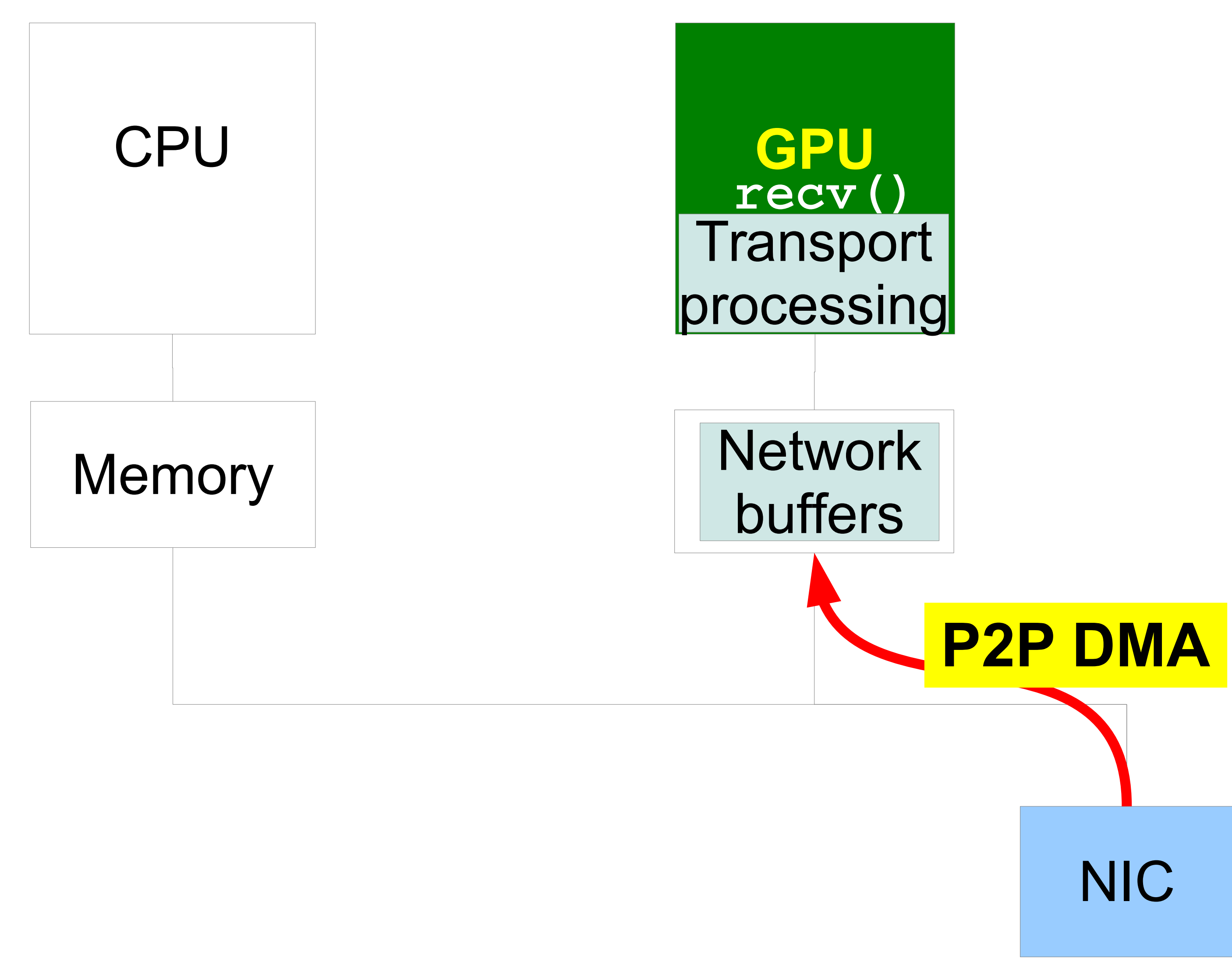
Design option 1: Transport layer processing on CPU



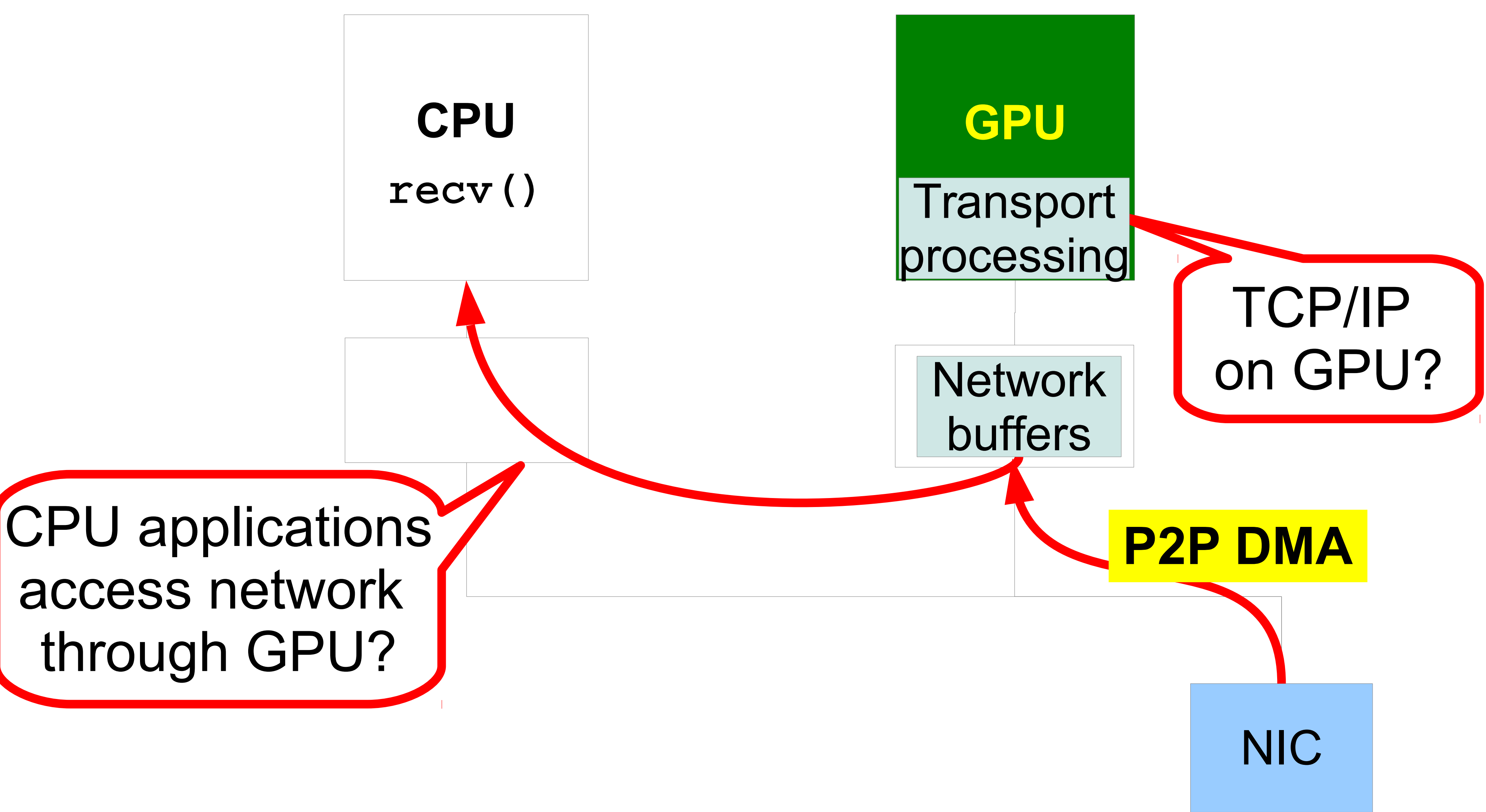
Design option 1: Transport layer processing on CPU



Design option 2: Transport layer processing on GPU



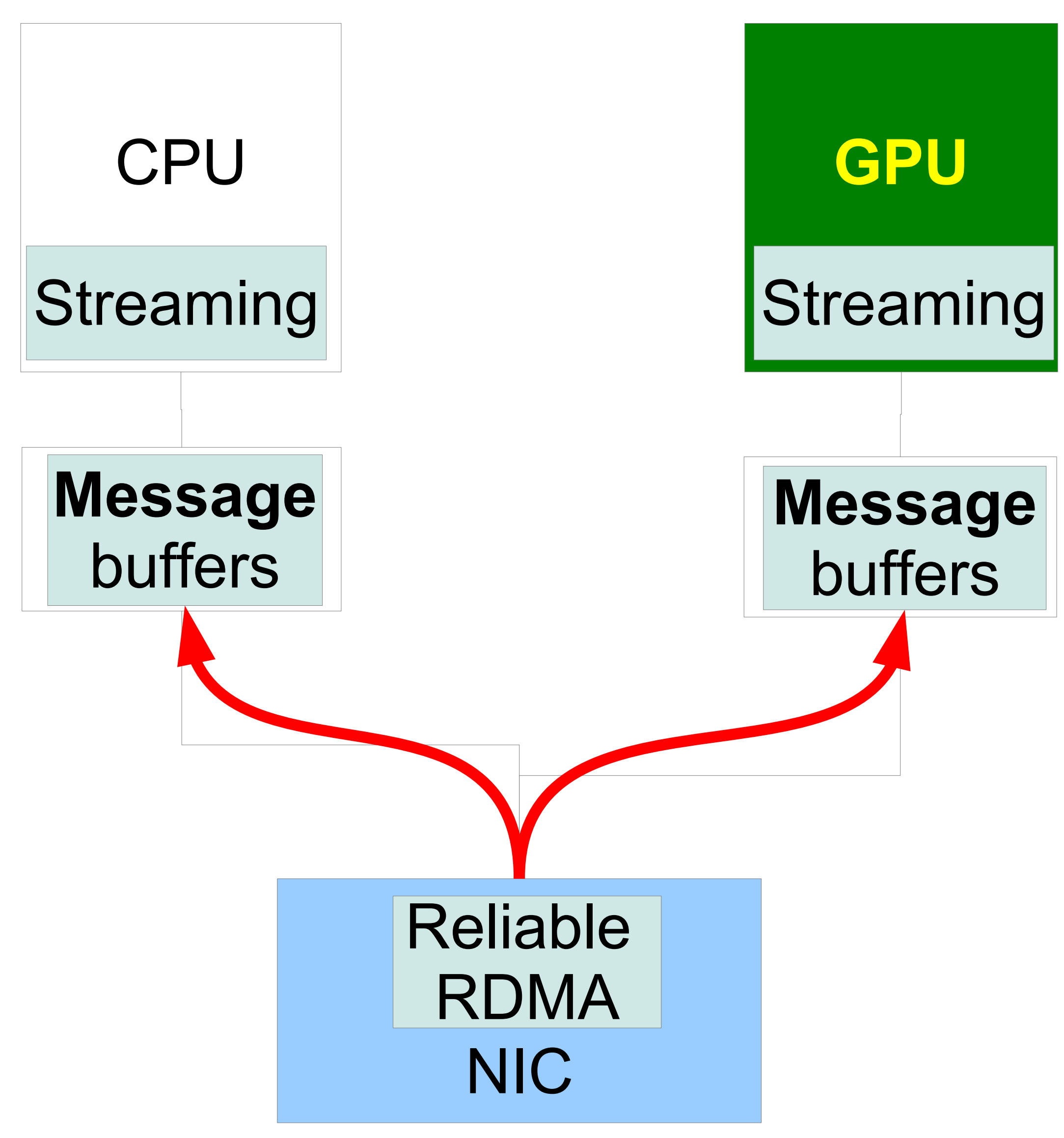
Design option 2: Transport layer processing on GPU



Not CPU, Not GPU

We need help from NIC hardware

RDMA: offloading transport layer processing to NIC



GPUnet layers

GPU Socket API	
Reliable in-order streaming	
Reliable channel	
RDMA Transports *Infiniband*	Non-RDMA Transports *UNIX Domain Socket, TCP/IP*

GPUnet layers

Simplicity

GPU

GPU Socket API

Reliable in-order streaming

Reliable channel

RDMA Transports
Infiniband

Non-RDMA Transports
UNIX Domain Socket, TCP/IP

NIC

CPU

Performance

Technical details

CTA-wide socket API

```
_global__ void gpuclient(struct sockaddr_in *addr, int* tb_alloc_tbl, int nr_tb) {
        __shared__ int sock;
        __shared__ uchar buf[BUF_SIZE];
        int ret, i;

        while ((sock = gconnect_in(addr)) < 0) {};
        assert(sock >= 0);

        for (i = 0; i < NR_MSG; i++) {
                int recved = 0, sent = 0;

                do {
                        ret = gsend(sock, buf + sent, BUF_SIZE - sent);
                        if (ret < 0) {
                                goto out;
                        } else {
                                sent += ret;
                        }
                } while (sent < BUF_SIZE);

                __syncthreads();
……..
```

What's hidden behind each socket

```
struct socket{
    flow control counters
    volatile gpumem* receive and send buffers
    open/close flags
}
```

- Socket table keeps track of all the sockets
- The buffers are registered to HCA
- Added read-only / write-only sockets to save space

Streaming over RDMA

- Sender: keep track of the amount of empty space in the receive buffer
  - IB HCA notifies on successful send completion (send buffer reuse)
- Receiver: report progress to sender
  - IB HCA notifies the sender that new data was received

Flow control implementation challenge

- Data stored in GPU memory, but control still goes through a CPU
  - Completions are received by a CPU
- Flow control logic spread across CPU and GPU
  - Shared counters
- Requires atomic updates of shared variables over PCI

Flow control implementation challenge

- Data stored in GPU memory, but control still goes through a CPU
  - Completions are received by a CPU
- Flow control logic spread across CPU and GPU
  - Shared counters
- Requires atomic updates of shared variables over PCI

Does not work

Producer-consumer idea

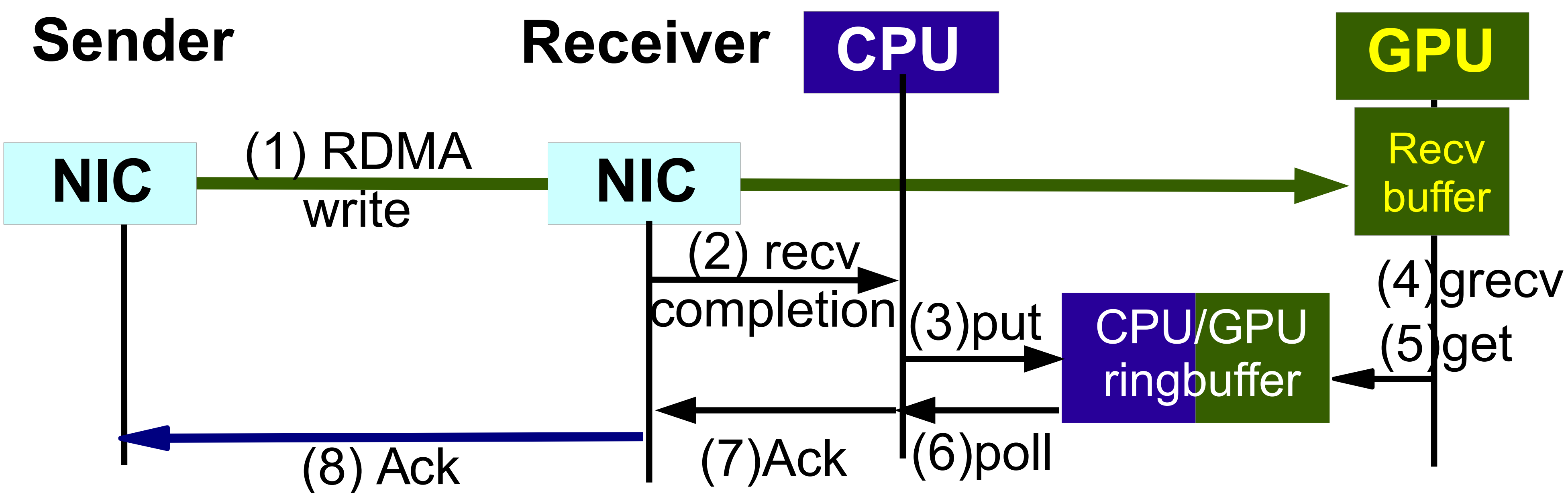
- Two producer-consumer instances mediated by a CPU

  Send: Producer – GPU, Consumer – Remote Host

  Receive: Producer – HCA, Consumer – GPU

- CPU updates counters in GPU-accessible memory on behalf of the other party
- Each instance is implemented using CPU-GPU shared ring-buffers – no locks, no atomics

The anatomy of recv()



The anatomy of send()

```
__device__ gsend(uchar* user_buf, uint size){
  uint sent=0;
  uint free=0;
   while(count<size){
      while(!( free=rb->available())); // wait for space
      copy_to_nw_buf(user_buf+sent,free);
      rb->push(free);
      count+=free;
   }
}
```

The anatomy of send()

```
__device__ gsend(uchar* user_buf, uint size){
  uint sent=0;
  uint free=0;
   while(count<size){
      while(!( free=rb->available())); // wait for space
       copy_to_nw_buf(user_buf+sent,free);
      rb->push(free);
      count+=free;
   }
}
```

Send throughput critically depends on the *latency* of the shared RB and memcpy

Memcpy issue

- Single CTA memcpy
- Maximum of ~14 GB/s (1024 threads) but HUGE number of registers – lower overall performance
- Naive – 2.4 GB/s – the main bottleneck!
- 6.9 GB/s with 16 byte reads, 256/512 threads
- Hack
  - nvcc 6.0 fails on compile `volatile double2*`
  - Use ptx instead

No longer a bottleneck

Send ringbuffer challenge

updated by GPU

updated by CPU

```
struct rb{
  volatile uchar head, tail;
  empty() { return tail==head; }
  full() { return (head+1)%SIZE == tail;}
  push() {
    while (full()); head=(head+1)%SIZE;
  }
}
```

Send ringbuffer challenge

stored in write-shared CPU memory

```
struct rb{
  volatile uchar head, tail;
  empty() { return tail==head; }
  full() { return (head+1)%SIZE == tail;}
  push() {
    while (full());                    E;
  }
}
```

These remote reads (from GPU) are expensive!!!

Send ringbuffer challenge

We can cache the head

```
struct rb{
  volatile uchar head, c_head, tail;
  empty() { return tail==c_head; }
  full() { return (c_head+1)%SIZE == tail;}
  push() {
    while (full()); c_head=(c_head+1)%SIZE;
    head=c_head;
  }
}
```

Send ringbuffer challenge

We can cache the head

```
struct rb{
  volatile uchar head, c_head, tail;
  empty() { return tail==c_head; }
  full() { return (c_head+1)%SIZE == tail;}
  push() {
    while (full()); c_head=(c_head+1)%SIZE;
    head=c_head;
  }
}
```

But what about tail?

Never *read* from remote memory

- Lets put both in GPU memory!
  - Map GPU memory into CPU address space
  - Simple kernel module using GPUDirectRDMA interface
- Reads become stale, so larger network buffers required to hide update latency
- Similar hack for recv(), but less critical

64 KB send call breakdown

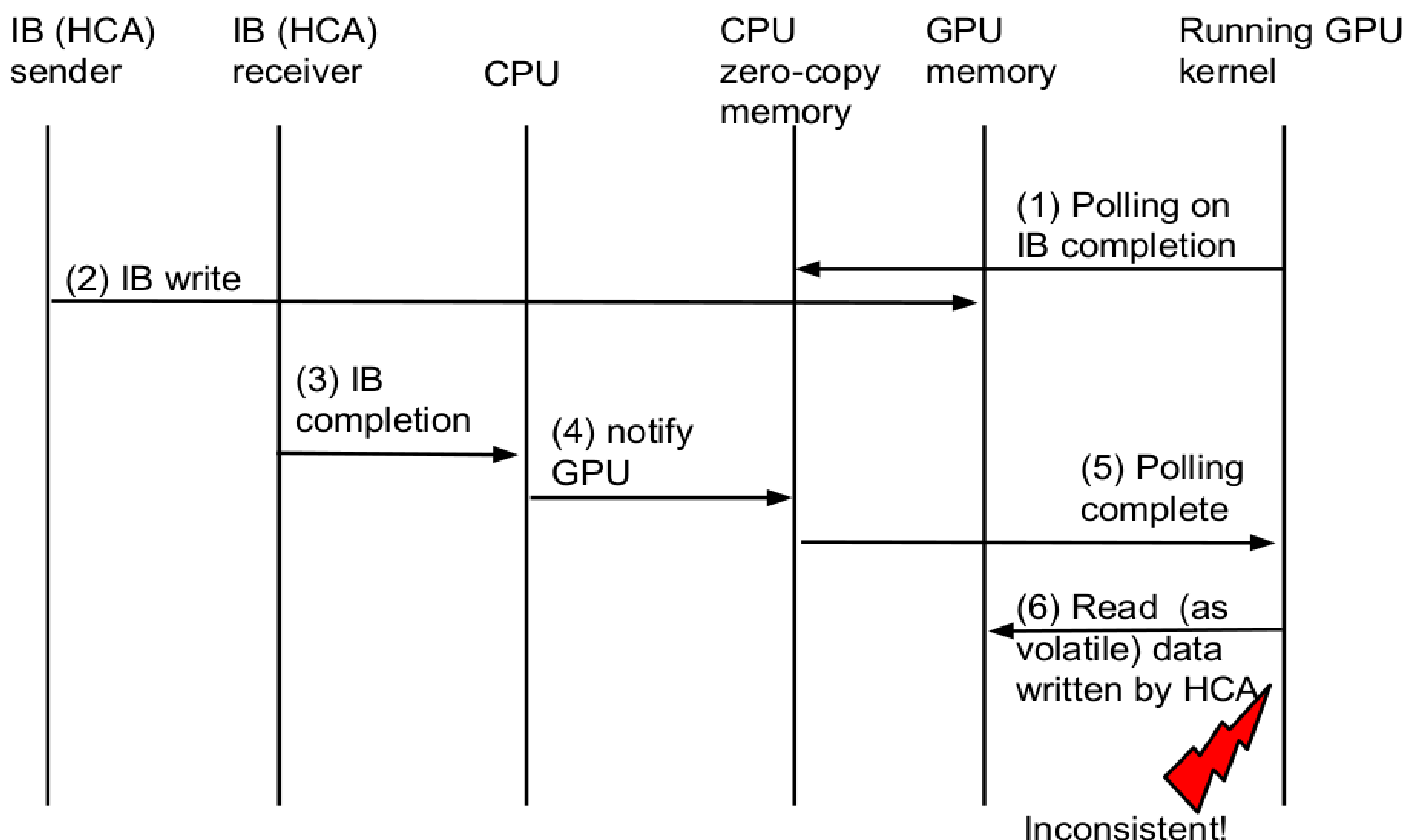
Profiling using clock64() for timing

Steps	Latency (μsec)	
T_1 GPU ring buffer	1.4	
T_2 GPU copies buffer	15.7	3.8GB/s
T_3 GPU requests to CPU	3.8	
T_4 CPU reads GPU request	2.5	
T_5 CPU RDMA write time to completion	22.2	2.68GB/s
Total one-way latency	45.6	

Memory management

- 220MB of GPU memory registered via GPUDirectRDMA
- Statically chopped up into 512 or 256KB buffers
- Each socket gets two buffers
- Tested with 480 sockets

Consistency problem



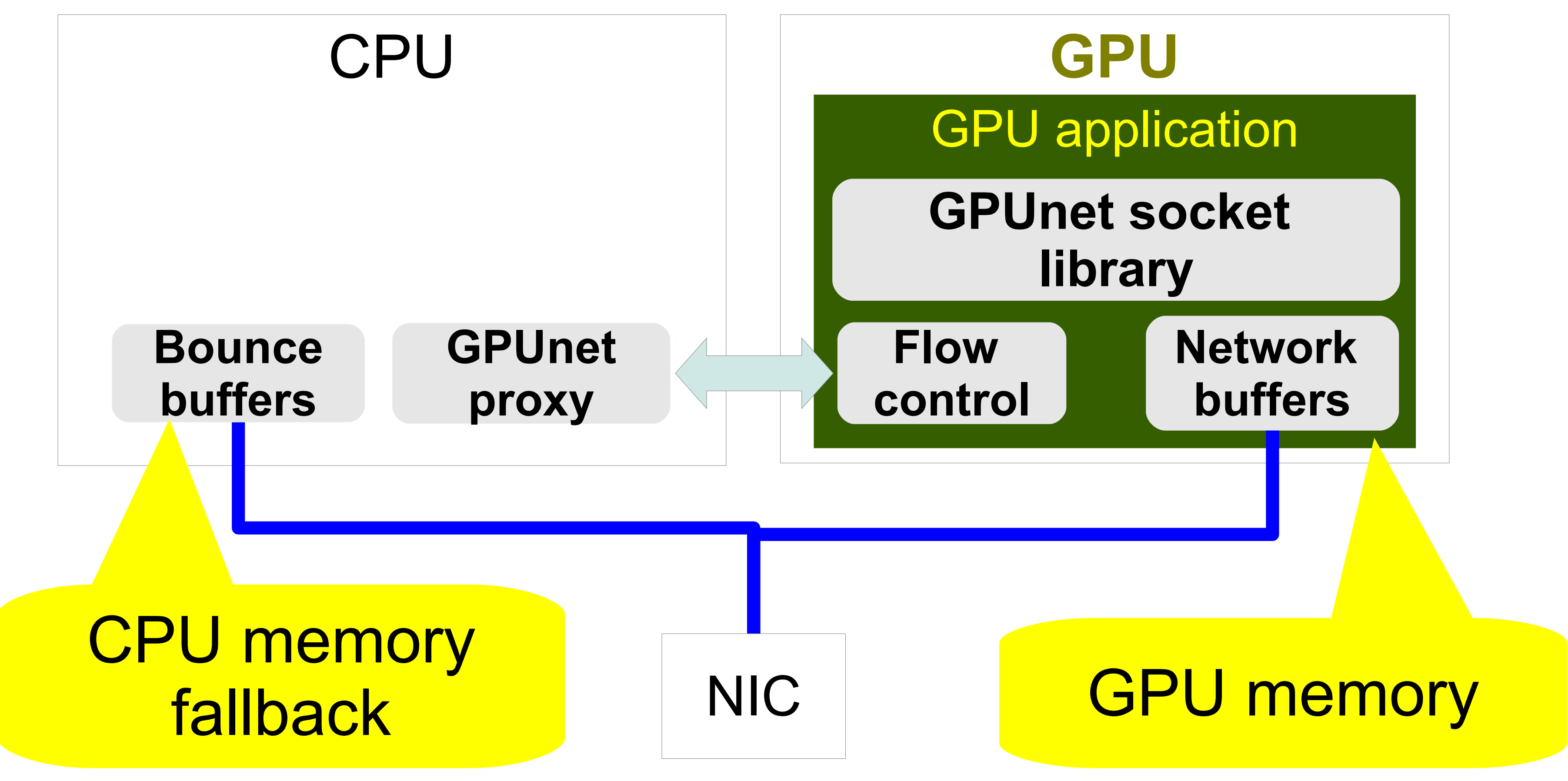
No guarantees but works in practice

- We added CRC32 on every n/w transaction
- Found no consistency violations
  - 10% overhead
- *PCIe consistency works* ***probably*** because of GPU-CPU notify in the last stage
  - But GPU might screw it up in internal buffers
- Surfaced GPUnet bug ;-)

Implementation

- Standard API calls, blocking/nonblocking
- **libGPUnet.a**: AF_INET, Streaming over Infiniband RDMA
  - Fully compatible with CPU **rsocket** library
- **libUNIXnet.a**: AF_LOCAL: Unix Domain Sockets support for inter GPU/CPU-GPU

Implementation



Evaluation

- Analysis of GPU-native server design
  - Matrix product server
- In-GPU-memory MapReduce
- Face verification server

2x6 Intel E5-2620, NVIDIA Tesla K20Xm GPU, Mellanox Connect-IB HCA, Switch-X bridge

Single socket bandwidth and latency

	CPU-CPU	CPU-GPU	CPU-GPU (BB)	GPU-GPU
GB/s	3.44	3.44	3.44	3.38
RTT 64 byte (usec)	2.86	26.9	60.3	50.0

Single socket bandwidth and latency

	CPU-CPU	CPU-GPU	CPU-GPU (BB)	GPU-GPU
GB/s	3.44	3.44	3.44	3.38
RTT 64 byte (usec)	2.86	26.9	60.3	50.0

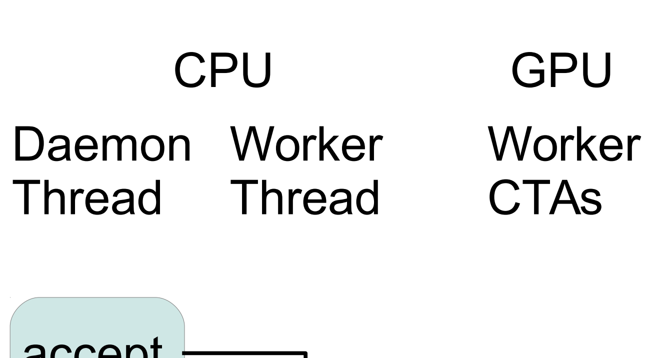
CPU-GPU interaction

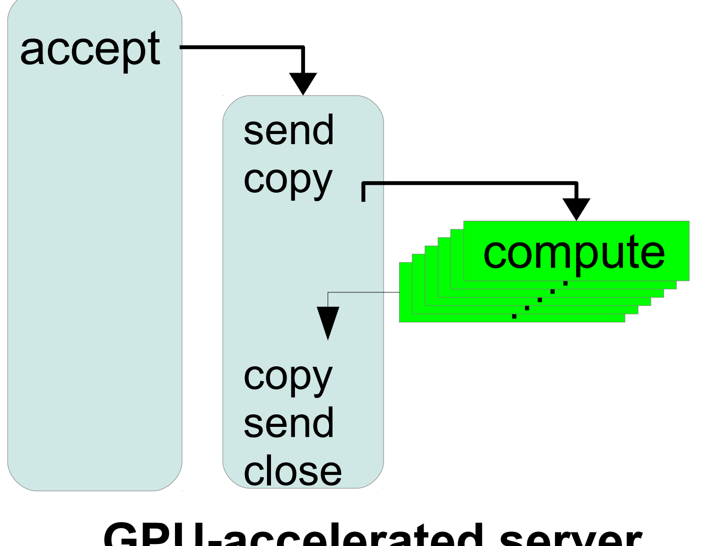
Single socket bandwidth and latency

	CPU-CPU	CPU-GPU	CPU-GPU (BB)	GPU-GPU
GB/s	3.44	3.44	3.44	3.38
RTT 64 byte (usec)	2.86	26.9	60.3	50.0

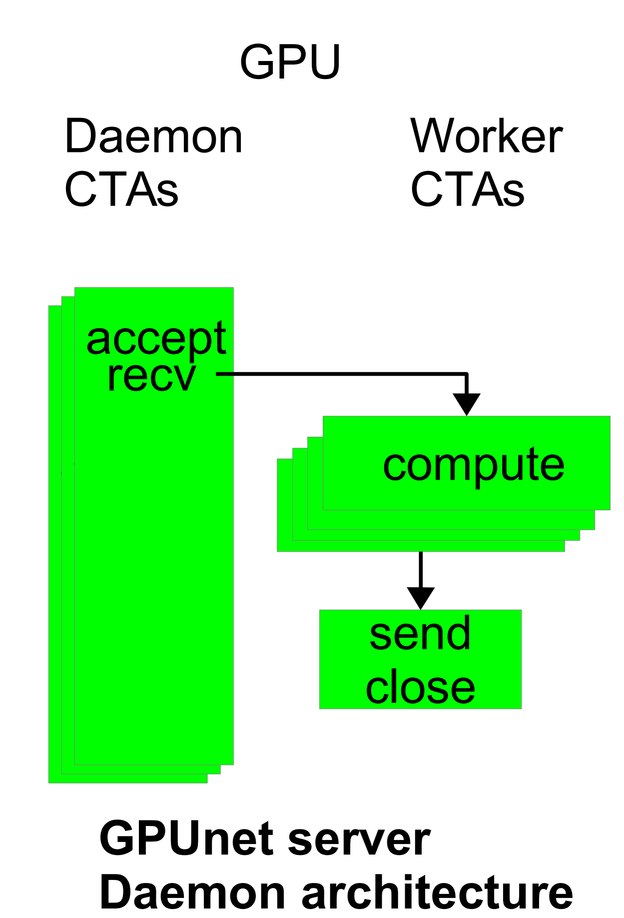
Extra-memory copy via CPU

Different GPU server architectures

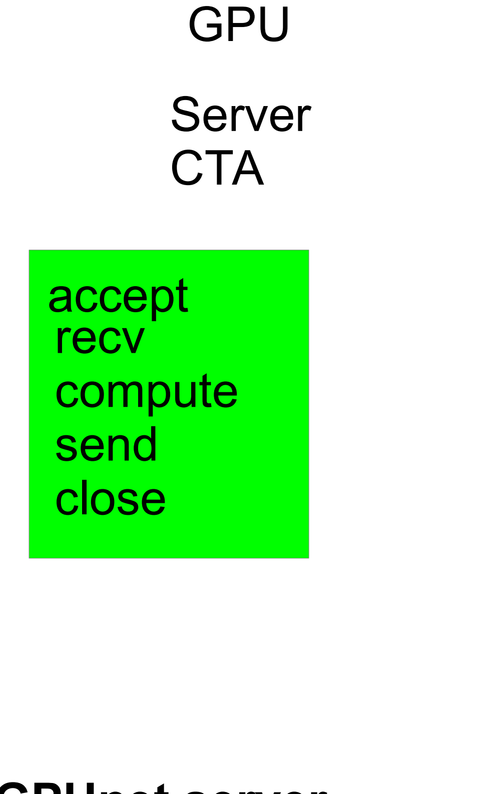




GPU-accelerated server



GPUnet server
Daemon architecture



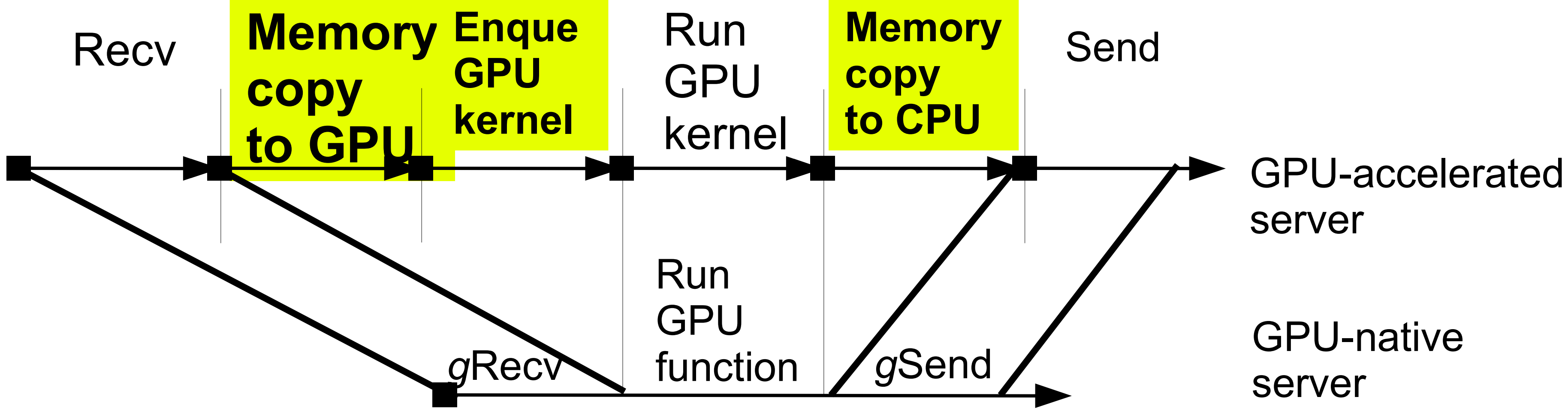
GPUnet server
Independent architecture

Different GPU server architectures

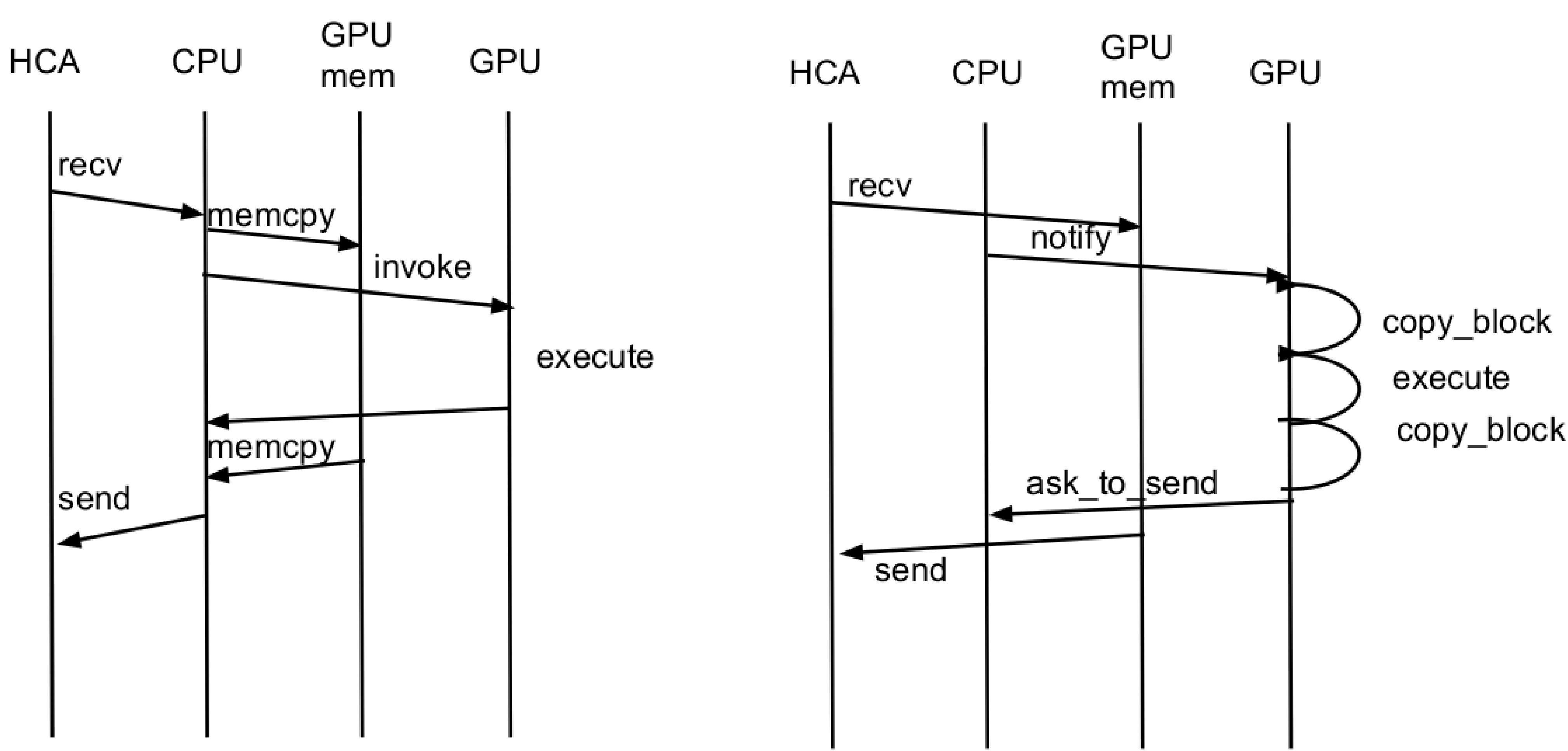
- Daemon: Listen → Fork
  - Partition GPU resources: I/O and worker CTAs
  - Worker CTAs invoked dynamically (can't use nested parallelism)
  - Pros: support dynamic workload
  - Cons: invocation overhead

- Independent: Listen → **Invoke**
  - GPU runs many persistent CTAs
  - Switch Pros/Cons of the daemon design

Why GPUnet is faster



Why GPUnet is faster



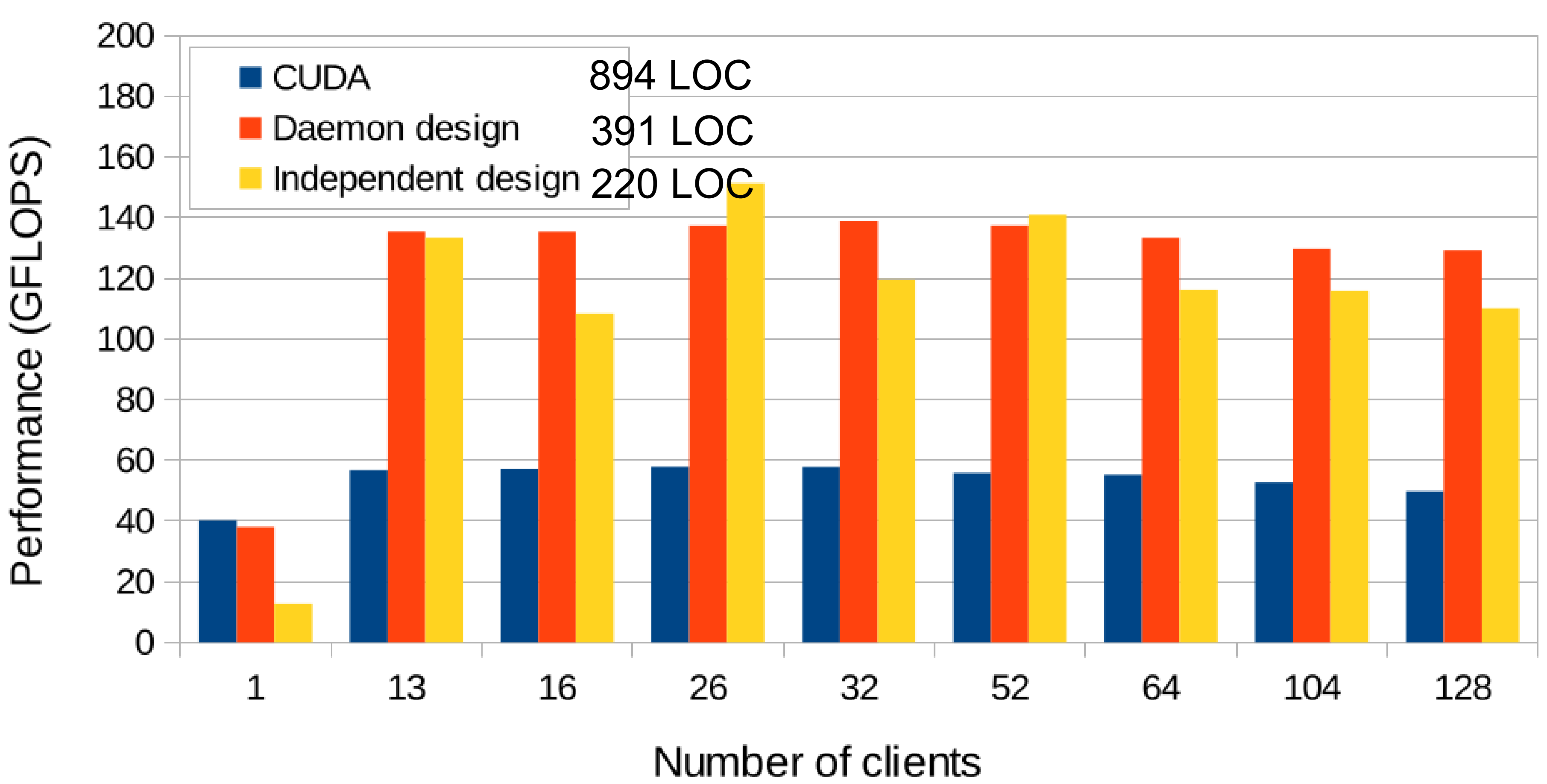
Daemon design: Determining the I/O-compute split

- Matrix product server
- Light/Medium/Heavy inputs: 64x64, 256x256, 1024x0124
- Light/Medium/Heavy configs: 16, 8, 4 daemon CTAs

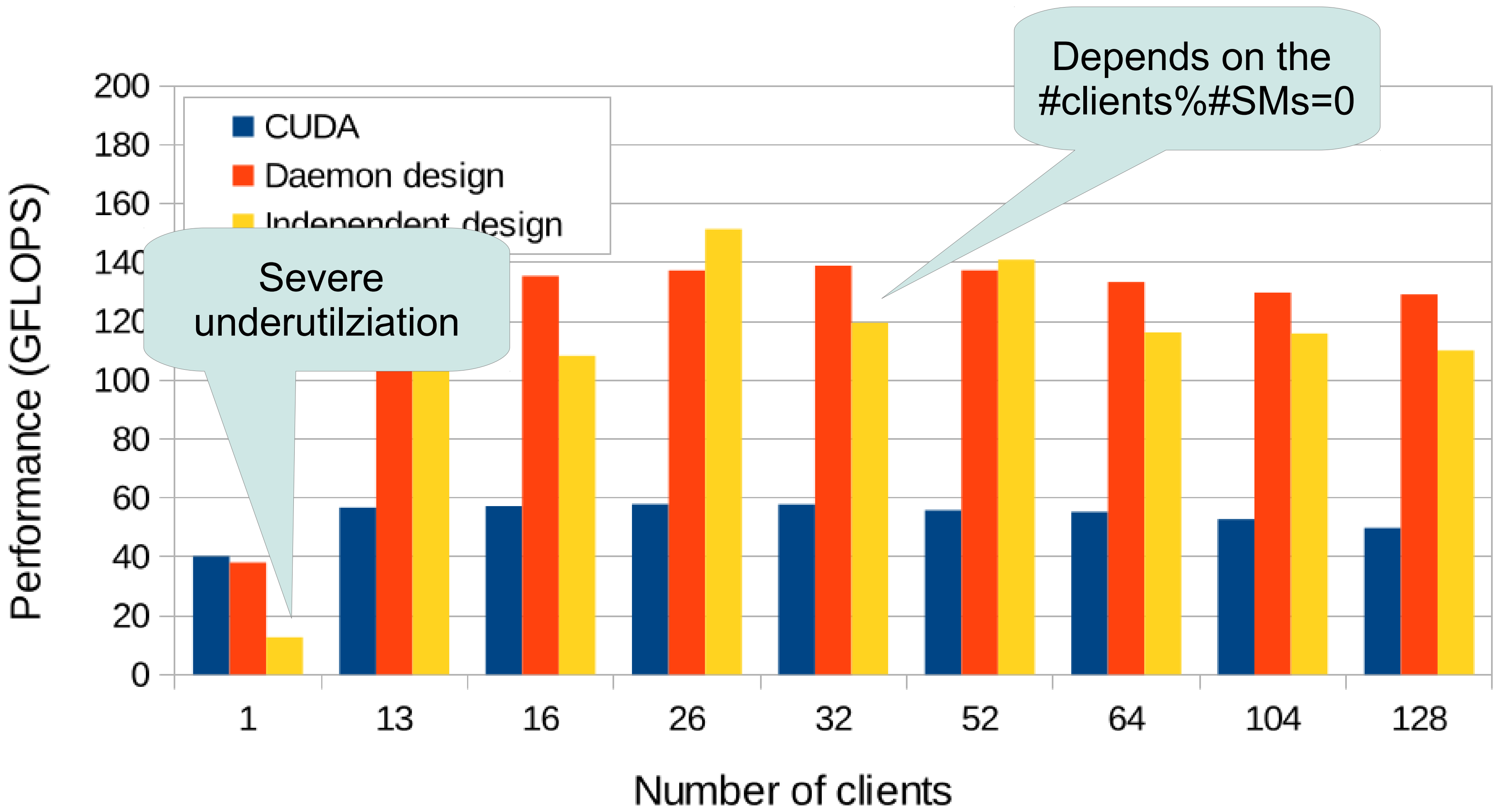
Cost of misconfiguration

Configuration	Workload Light	Medium	Heavy
Light	92%	81%	74%
Medium	44%	99%	88%
Heavy	12%	44%	100%

Different server designs (256x256 matrices)



Different server designs (256x256 matrices)

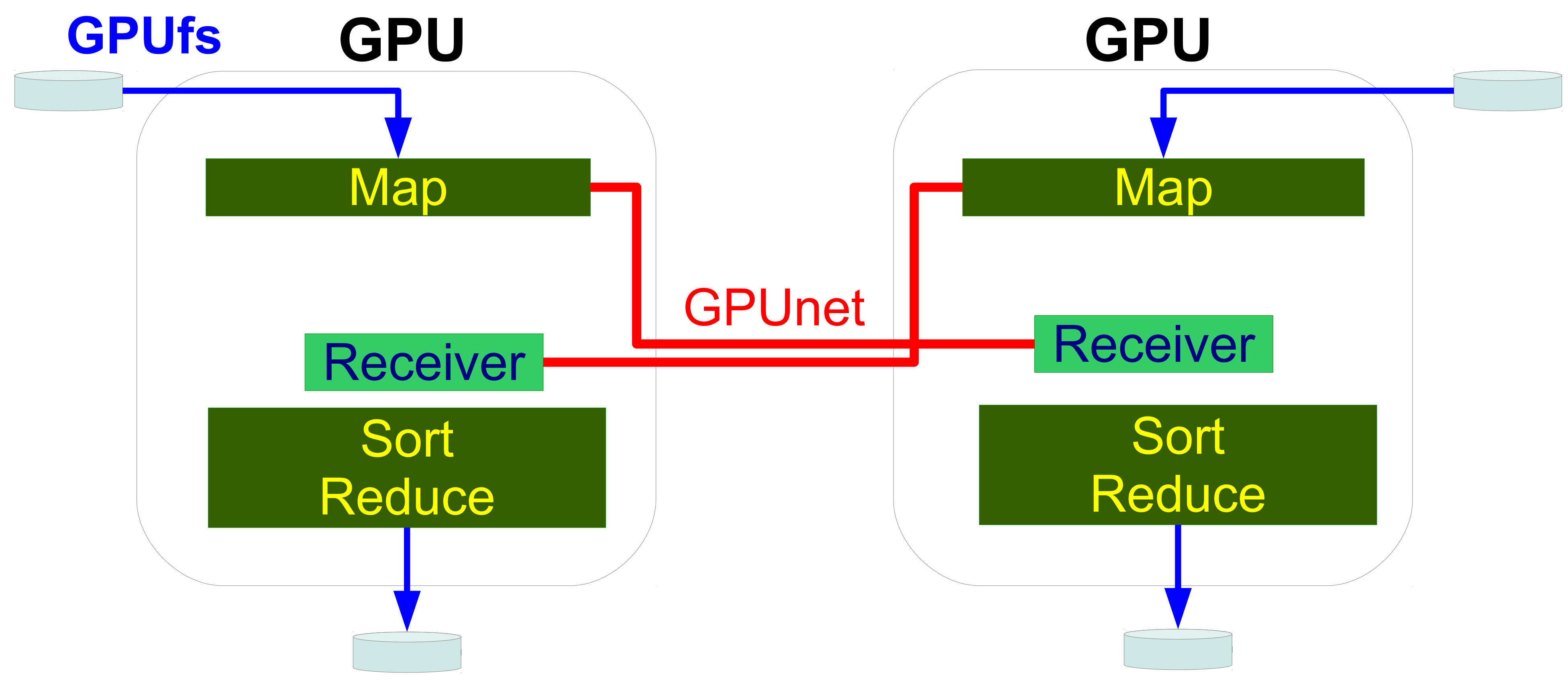


Daemon vs. Independent design

Server design	Light workload	Medium workload	Heavy workload
Daemon (GFLOPS)	11	137	201
Independent (GFLOPS)	37 (3.4×)	151 (1.1×)	207 (1.01×)

Independent design is best for (very) short kernels

In-GPU-memory MapReduce



In-GPU-memory MapReduce: Scalability

	1 GPU (no network)	4 GPUs (GPUnet)
K-means	5.6 sec	1.6 sec (**3.5x**)
Word-count	29.6 sec	10 sec (**2.9x**)

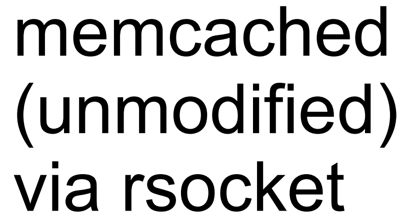
GPUnet enables scale-out
for GPU – accelerated systems

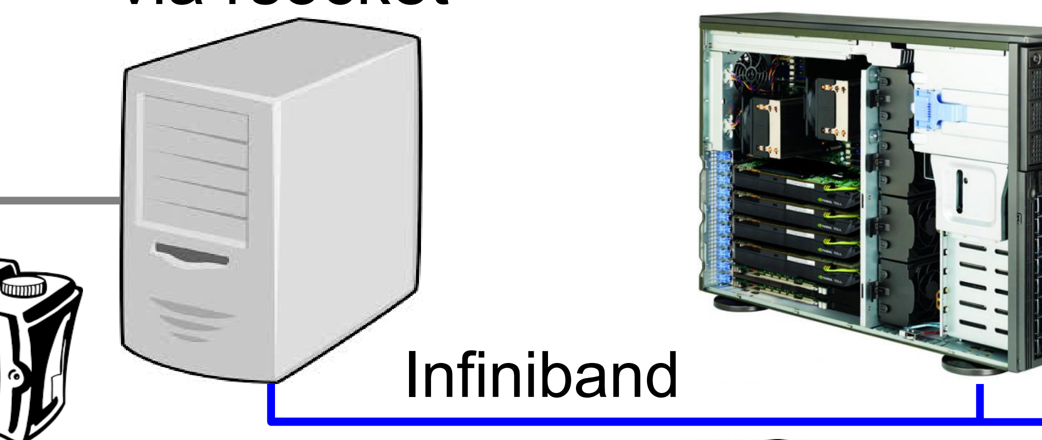
Face verification server

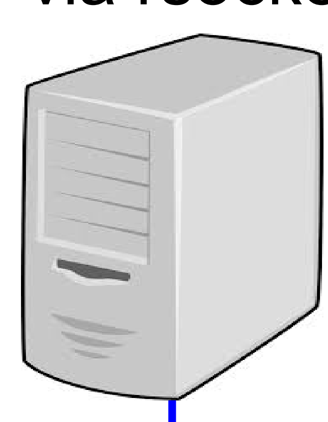
CPU client (unmodified) via rsocket

GPU server (GPUnet)

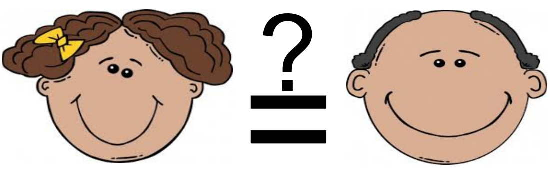
memcached (unmodified) via rsocket





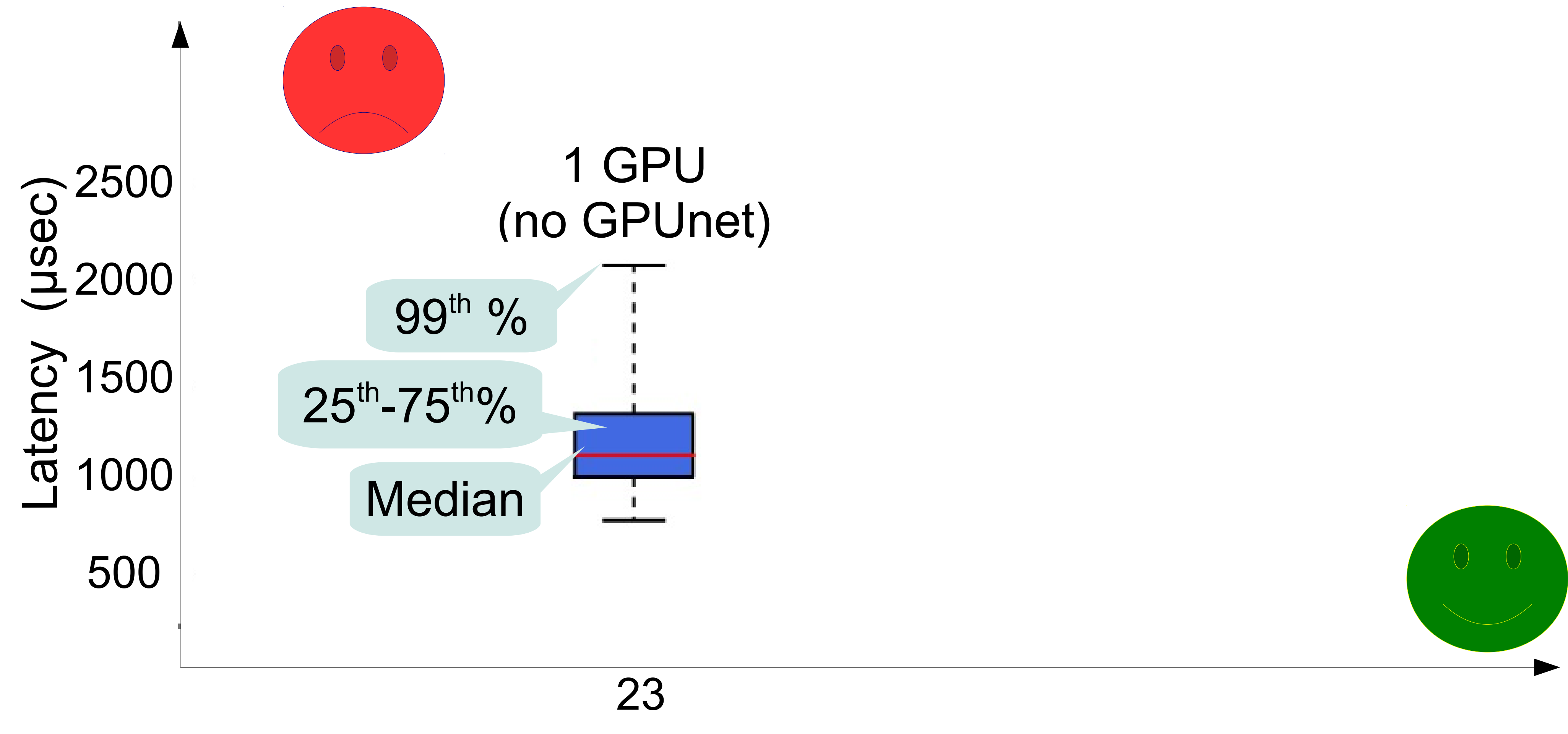


Infiniband

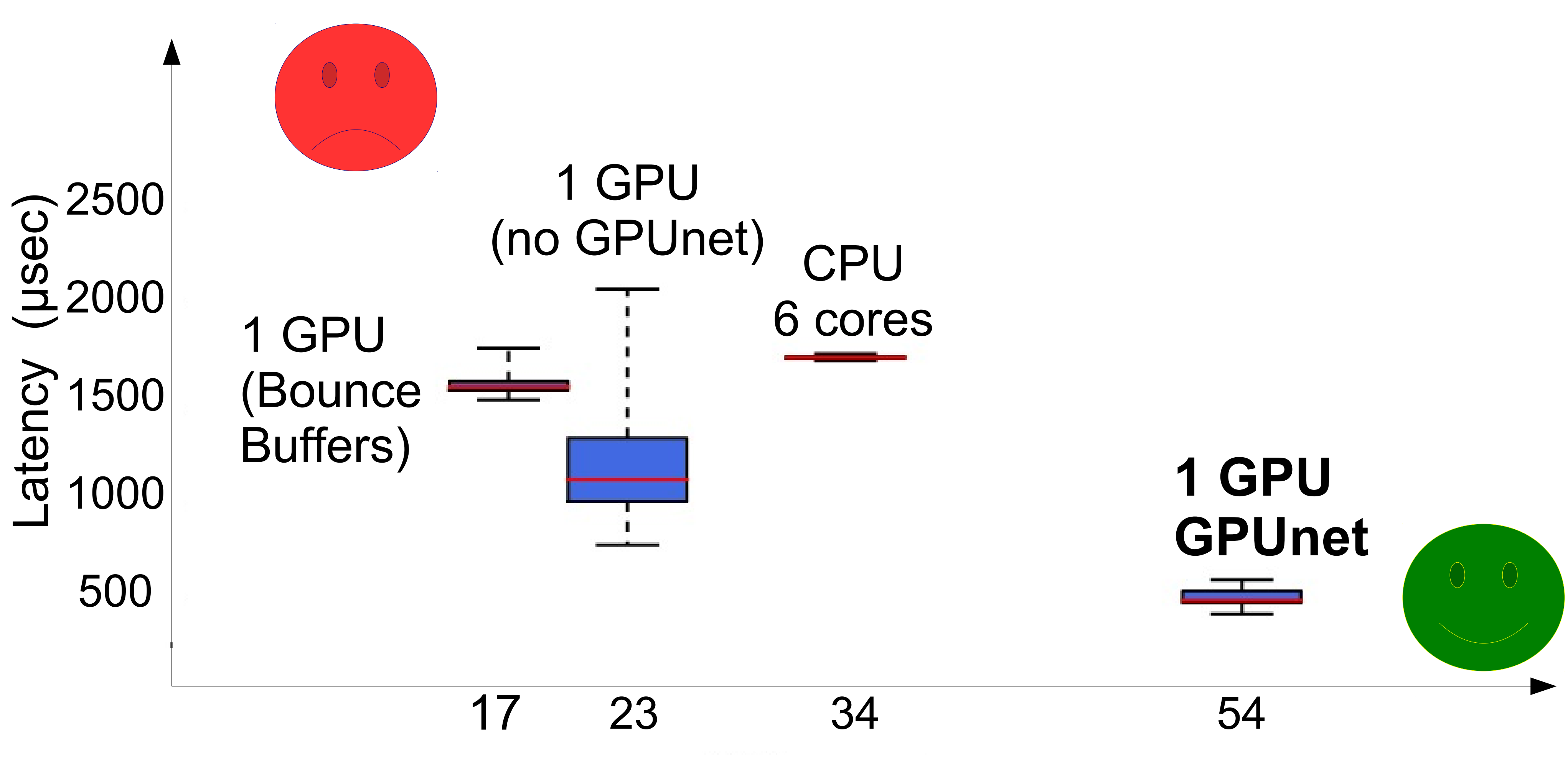


```
recv()
GPU_features()
query DB()
GPU_compare()
send()
```

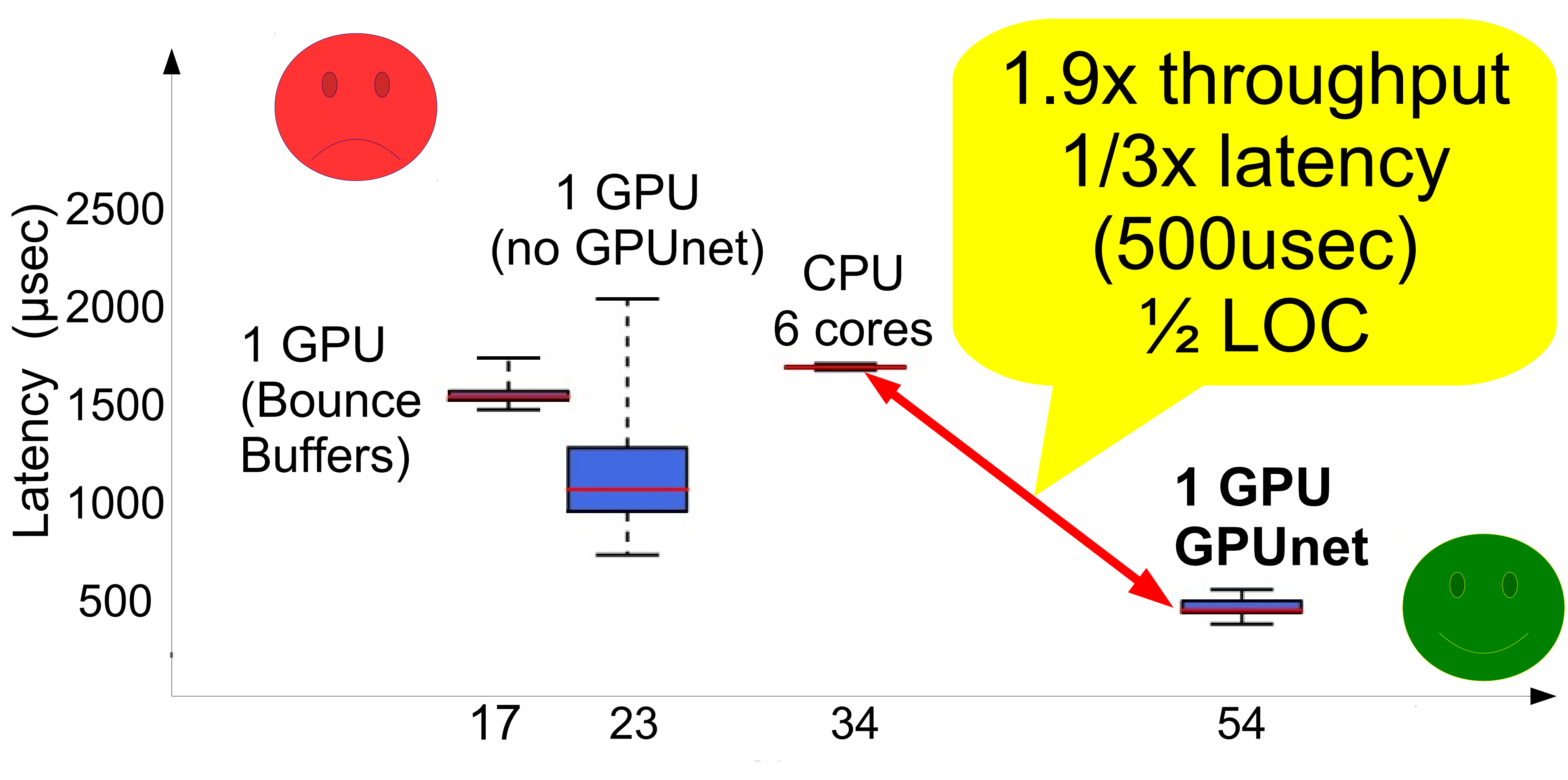
Face verification: NVIDIA CUDA (no batching)



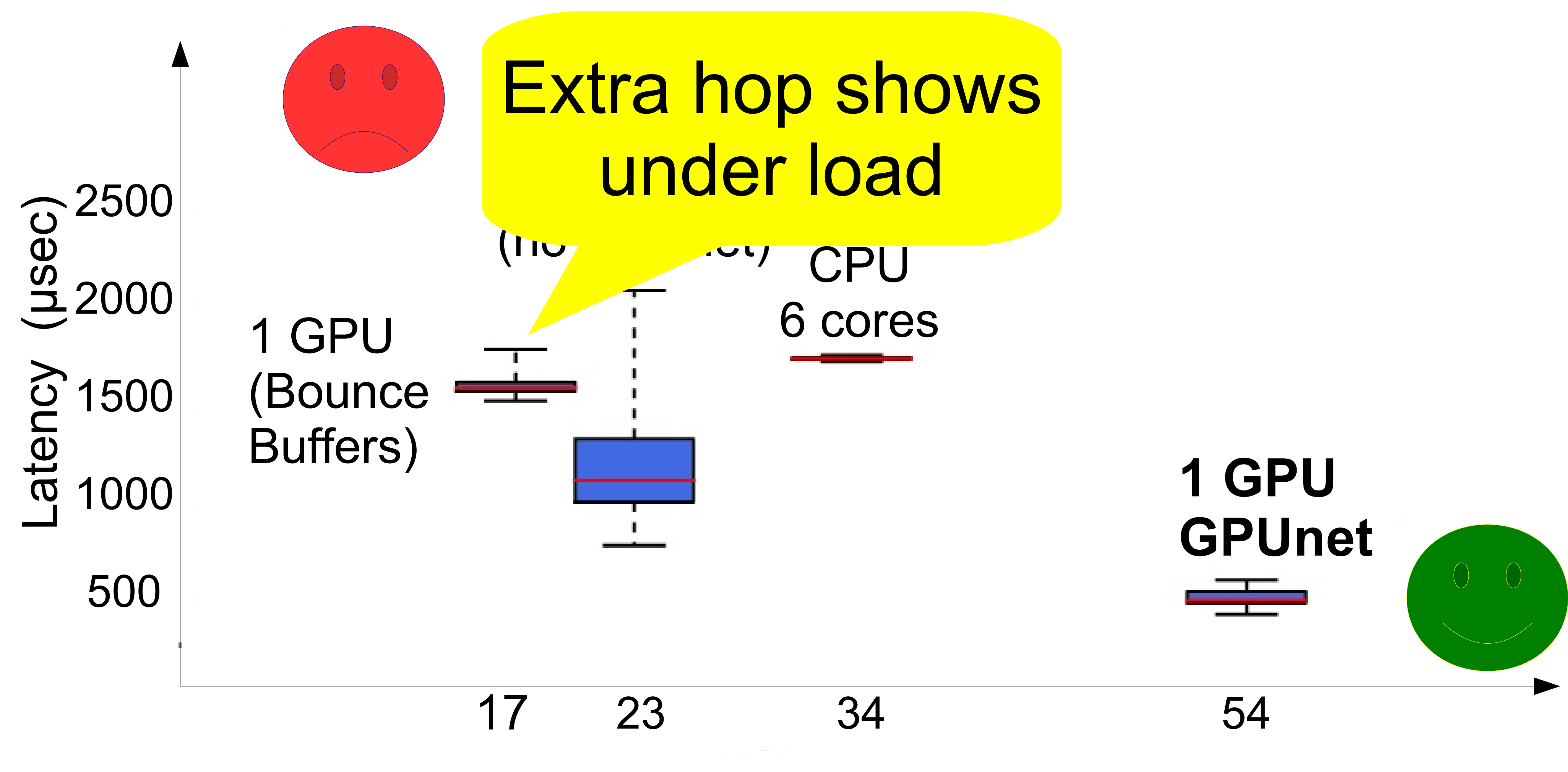
Face verification: Different implementations



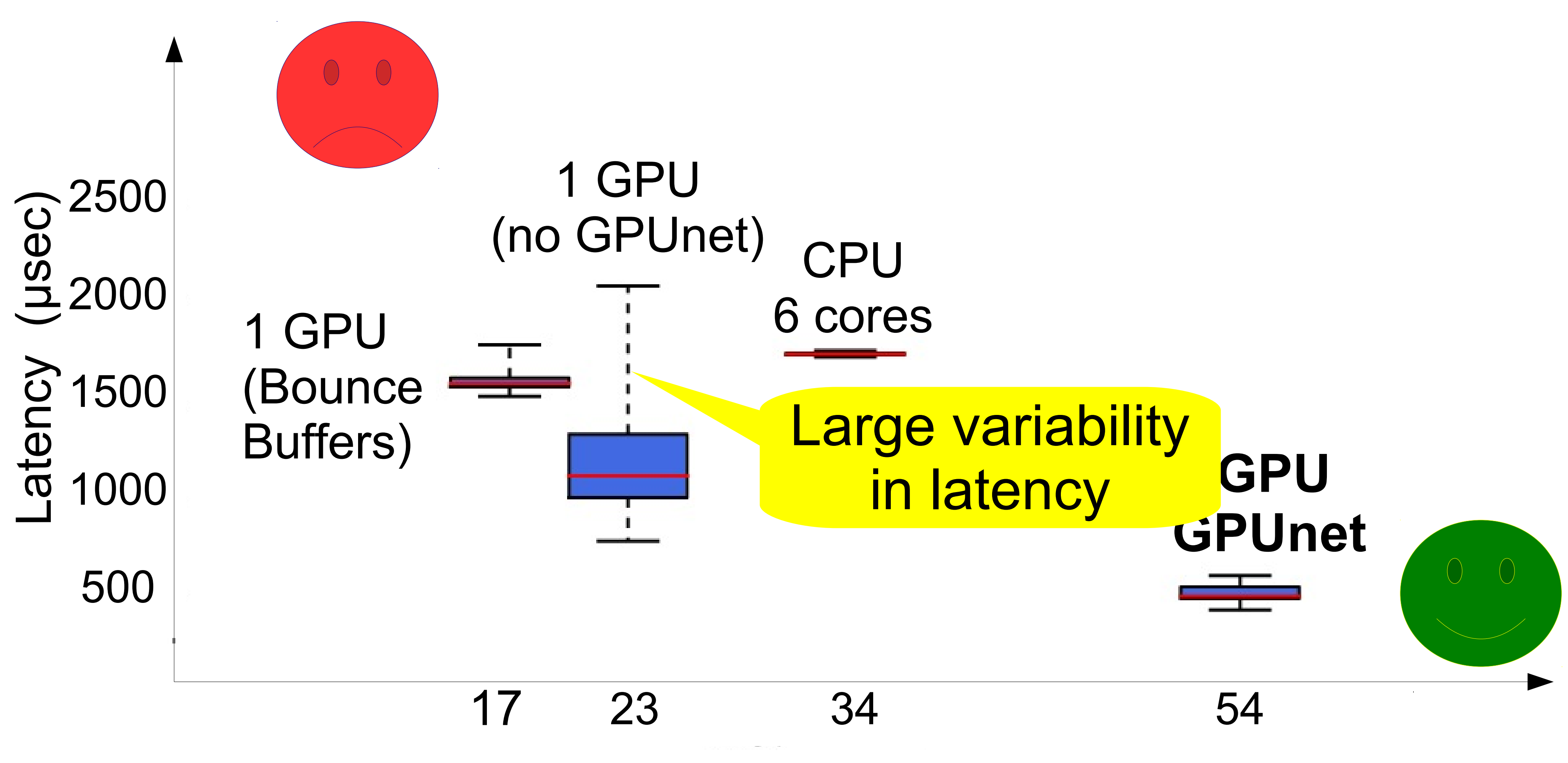
Face verification: Different implementations



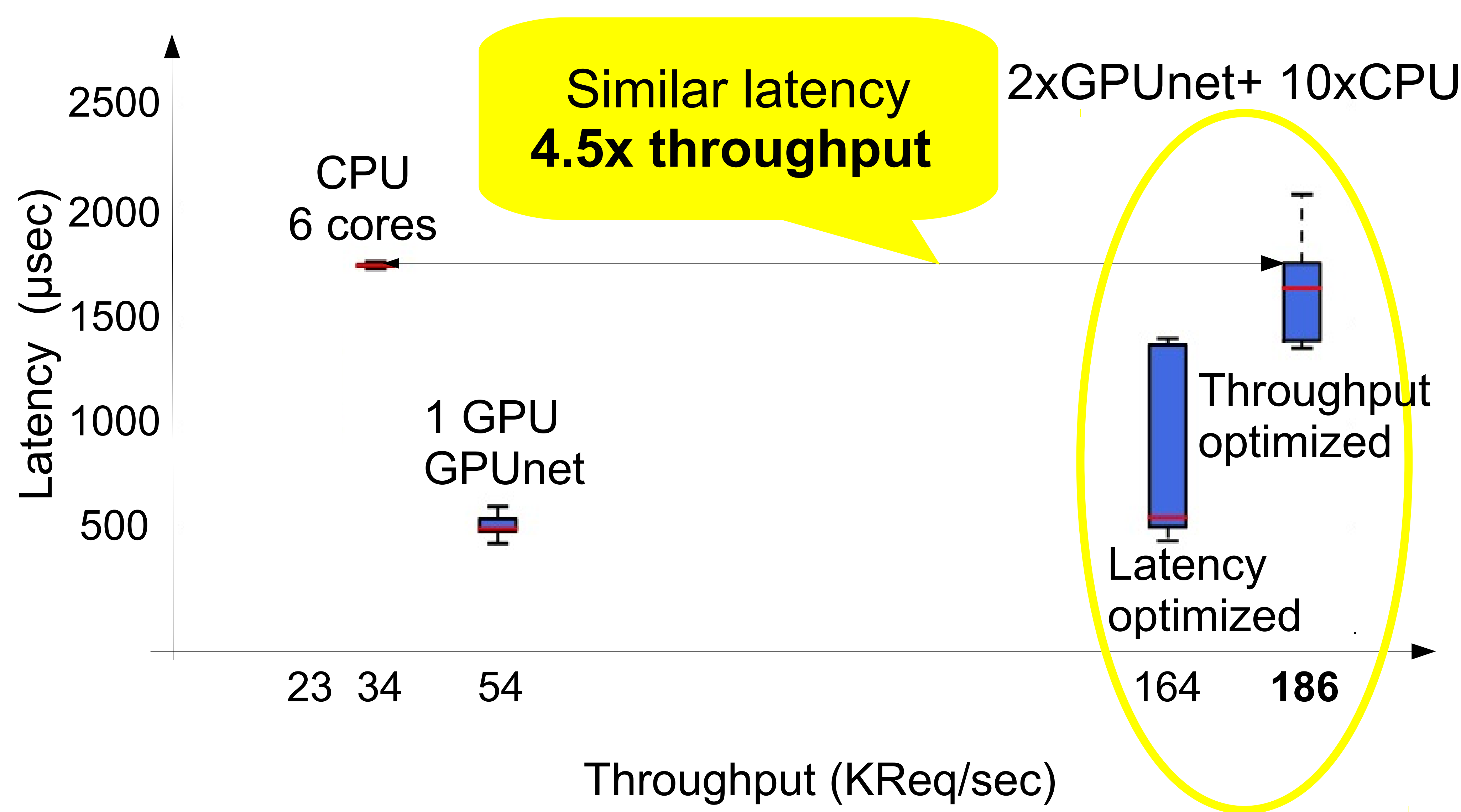
Face verification: Different implementations



Face verification: Different implementations



Face verification on all processors 2xGPU + 10xCPU



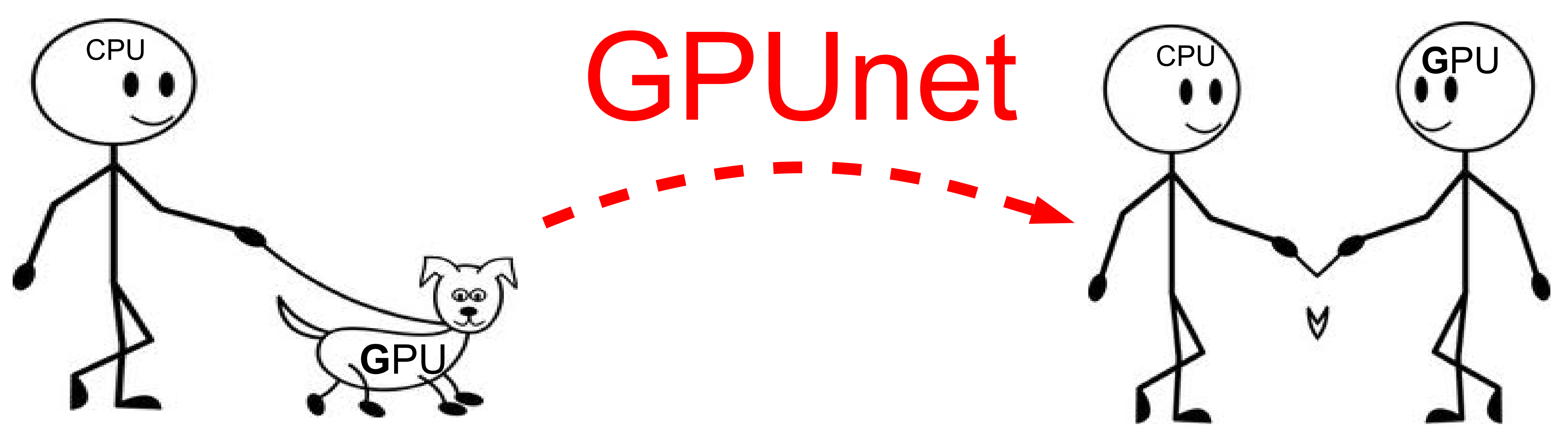
Future work

- Goal: 100Gb/s
  - Must map MMIO and HCA CQs
  - Zero-copy API
- Basic MPI support
- High-concurrency/ low latency servers
- Can GPUs leverage other accelerators?
- I/O on integrated GPUs

Wish-list

- GPU memcpy DMA
- GPU-CPU events (polling is a power hog)
- Mapping MMIO into GPU address space
- Consistency, consistency, consistency
- Inter-block barriers
- Kernel invocation with fire-forget semantics
- MWAIT-like calls
- Better debug/profile visualization
  - Report assertions more reliably
- Call GL and video encode/decode from a kernel
- Memory attestation
  - or ~secure boot
- clock64() precision (?)
- memory protection bits

Set GPUs free!



GPUnet is a library providing networking abstractions for GPUs

https://github.com/ut-osa/gpunet

mark@ee.technion.ac.il