Operating System Abstractions for GPU Programming

Chris Rossbach, Microsoft Research
Emmett Witchel, University of Texas at Austin
September 23 2010

Motivation

- Broaden GPU application domains
  - Cheaper/simpler development cycles
  - Bigger deployed base
  - Better utilization in deployed systems

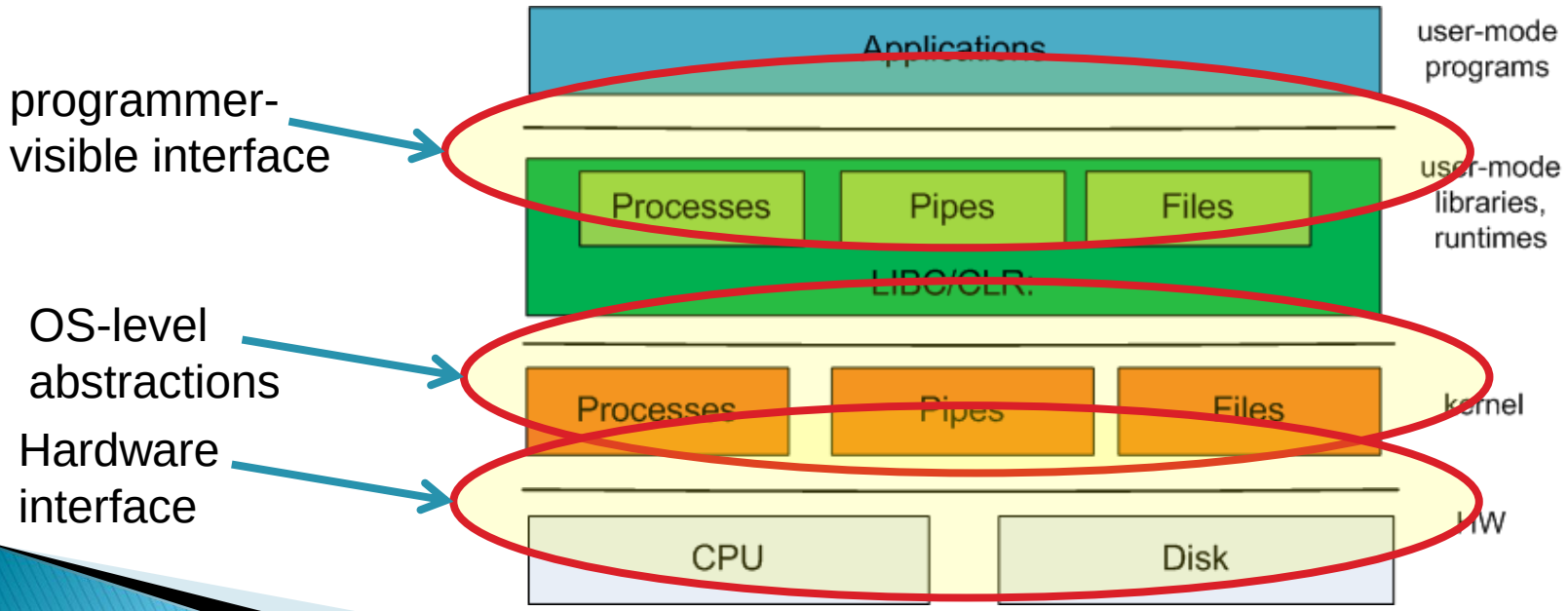
CUDA is a tremendous leap for parallel programming, but…

we need better OS-level abstractions

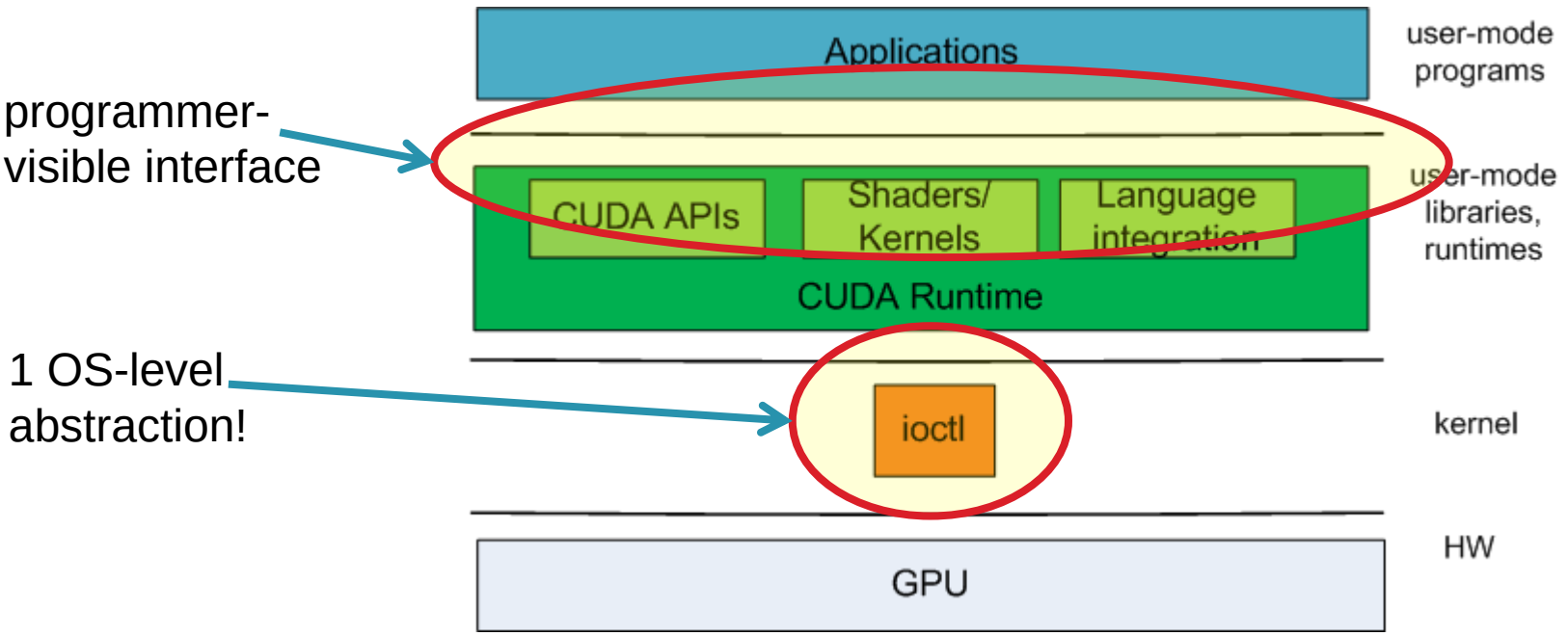
- *Programmers can express things whose implementations fall short*

Consider a simplified machine

```
int main(argc, argv) {
  FILE * fp = fopen(“quack”, “w”);
  // How do I program just a CPU and a disk?
  if(fp == NULL)
    fprintf(stderr, “failure\n”);
  …
  return 0;
}
```



GPU Abstractions

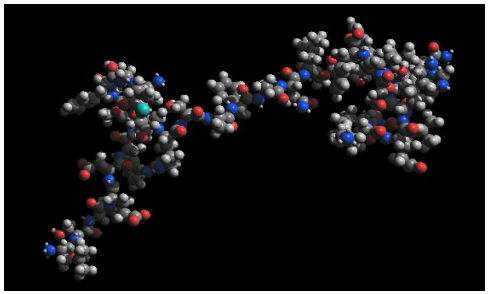


The programmer gets to work with great abstractions… so is this a problem?

Poor OS abstractions limit GPUs

Doing fine without OS support:

- Gaming/Graphics
  - Shader Languages
  - DirectX, OpenGL
- GPU Computing
  - user-mode/batch
  - scientific algorithms
  - Latency-tolerant
  - CUDA



- The application ecosystem is more diverse
- Poor OS abstractions → limited domains

Outline

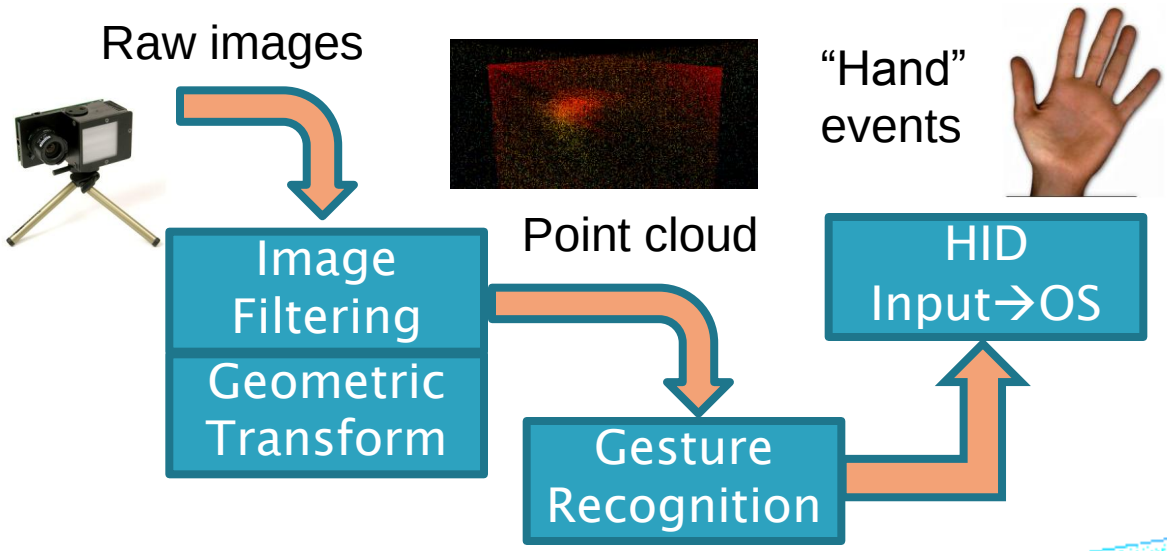
- Motivation
- The need for OS abstractions
- Why CUDA alone (currently) isn't enough
- New OS abstractions
- Related Work
- Conclusion

Interactive Applications

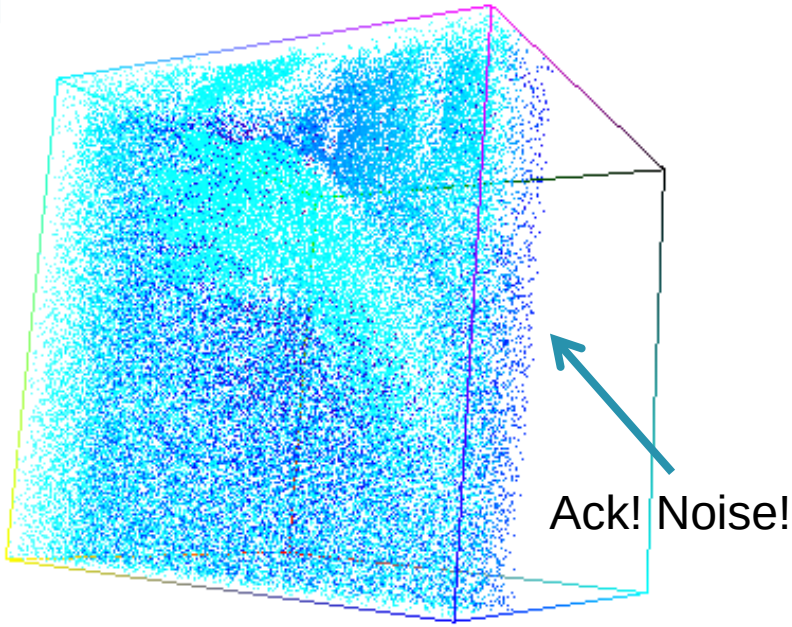
- Gestural Interface
- Brain-Computer Interface *(no apologies!)*
- Spatial Audio
- Image Recognition

Processing user input:
- need low latency, concurrency
- must be multiplexed by OS

Gestural Interface

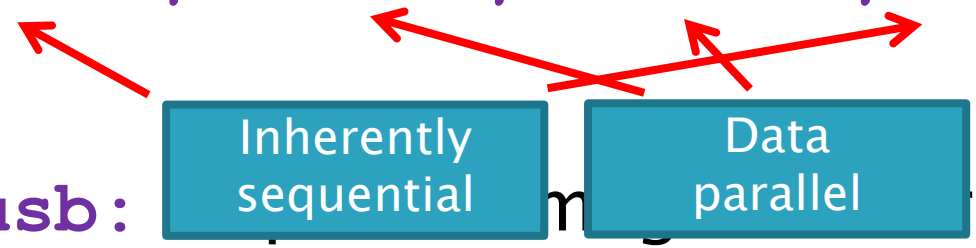


- High data rates
- Noisy input
- Data-parallel algorithms



What I wish I could do

```
#> catusb | xform | detect | hidinput &
```



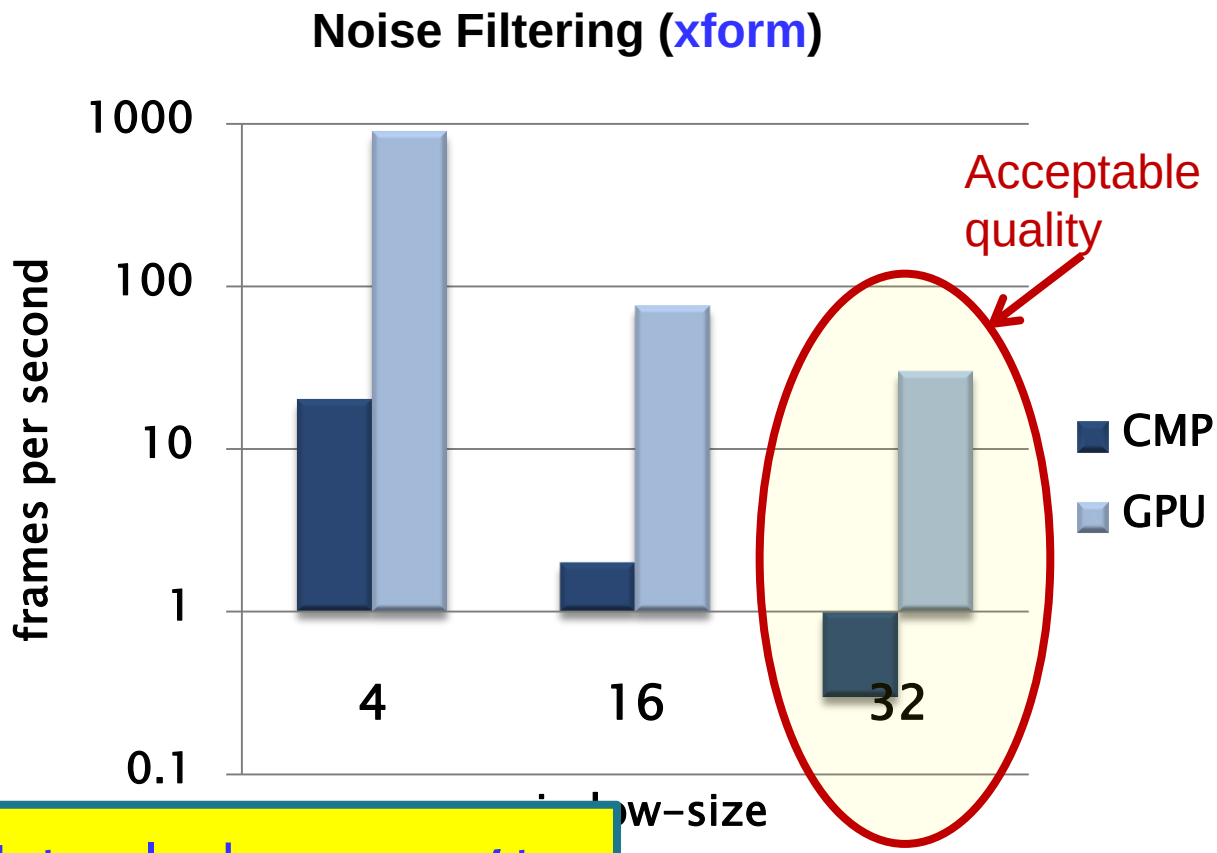
- **catusb:** m om usb
- **xform:**
  - Noise filtering
  - Geometric transformation
- **detect:** extract gestures from point cloud
- **hidinput:** send mouse events (or whatever)

Could parallelize on a CMP, but…

GPUs succeed where CPUs fail

Small window

Large window



Not only do we *want* to use the GPU, we *need* to!

sta 3GB RAM

- **4 cores**: Intel Core 2 Quad 2.40GHz
- Nvidia GeForce 9800 GX2

So use the GPU! (naïve approach)

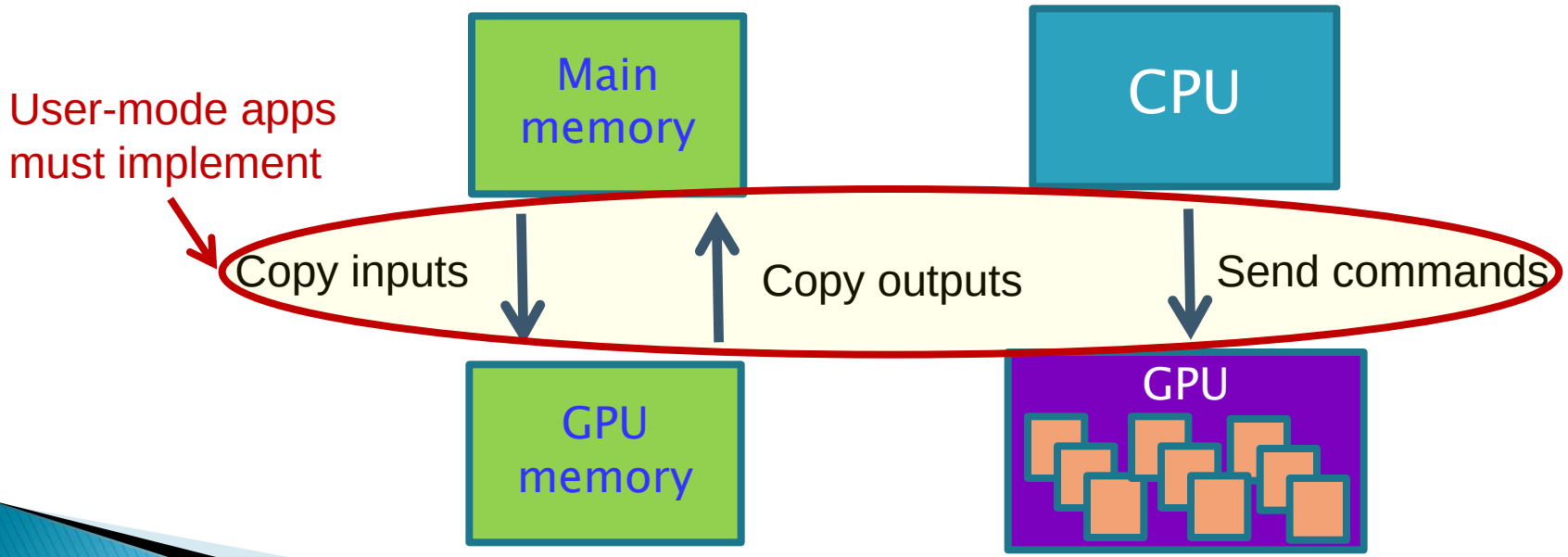
```
#> catusb | xform | detect | hidinput &
```

- Run `catusb` on CPU
- Run `xform` uses GPU
- Run `detect` uses GPU
- Run `hidinput:` on CPU

And use CUDA to write xform and detect!

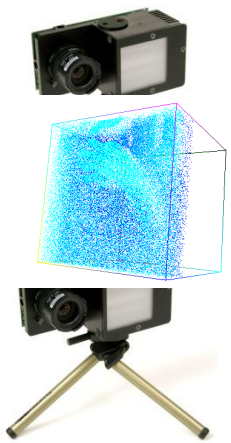
Running a program on a GPU

- GPUs cannot run OS: different ISA
- Disjoint memory space, no coherence*
- Host CPU must manage execution
  - Program inputs explicitly bound at runtime



Technology Stack View

- 12 kernel crossings
- 6 copy_to_user
- 6 copy_from_user
- Performance tradeoffs for runtime/abstractions



xform

detect

CUDA Runtime

User Mode Drivers (DXVA)

user

kernel

OS Executive

Kernel Mode Drivers

HAL

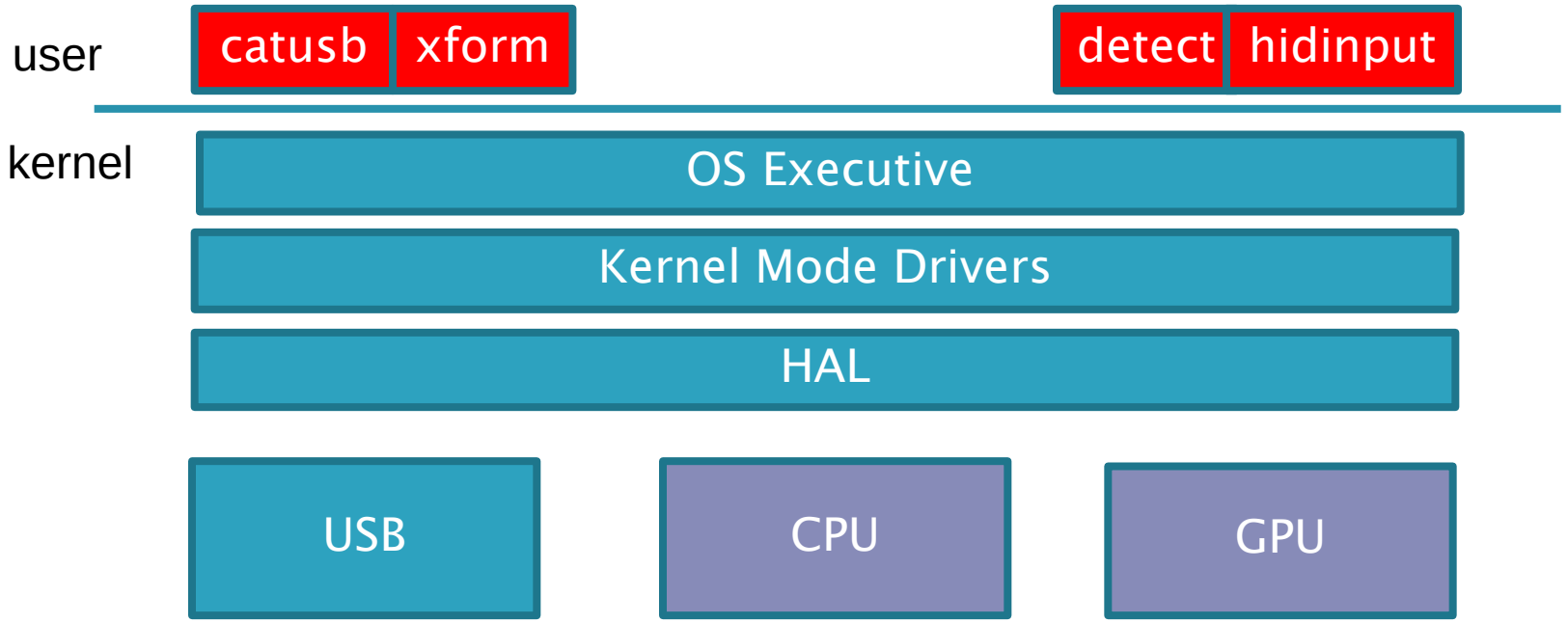
USB

CPU

GPU

Run Shader Program

So, big deal…do it all in the kernel



- No high level abstractions
- ***If*** you're MS and/or nVidia, this is tenable…
- Solution is specialized

but there is still a data migration problem…

Hardware View

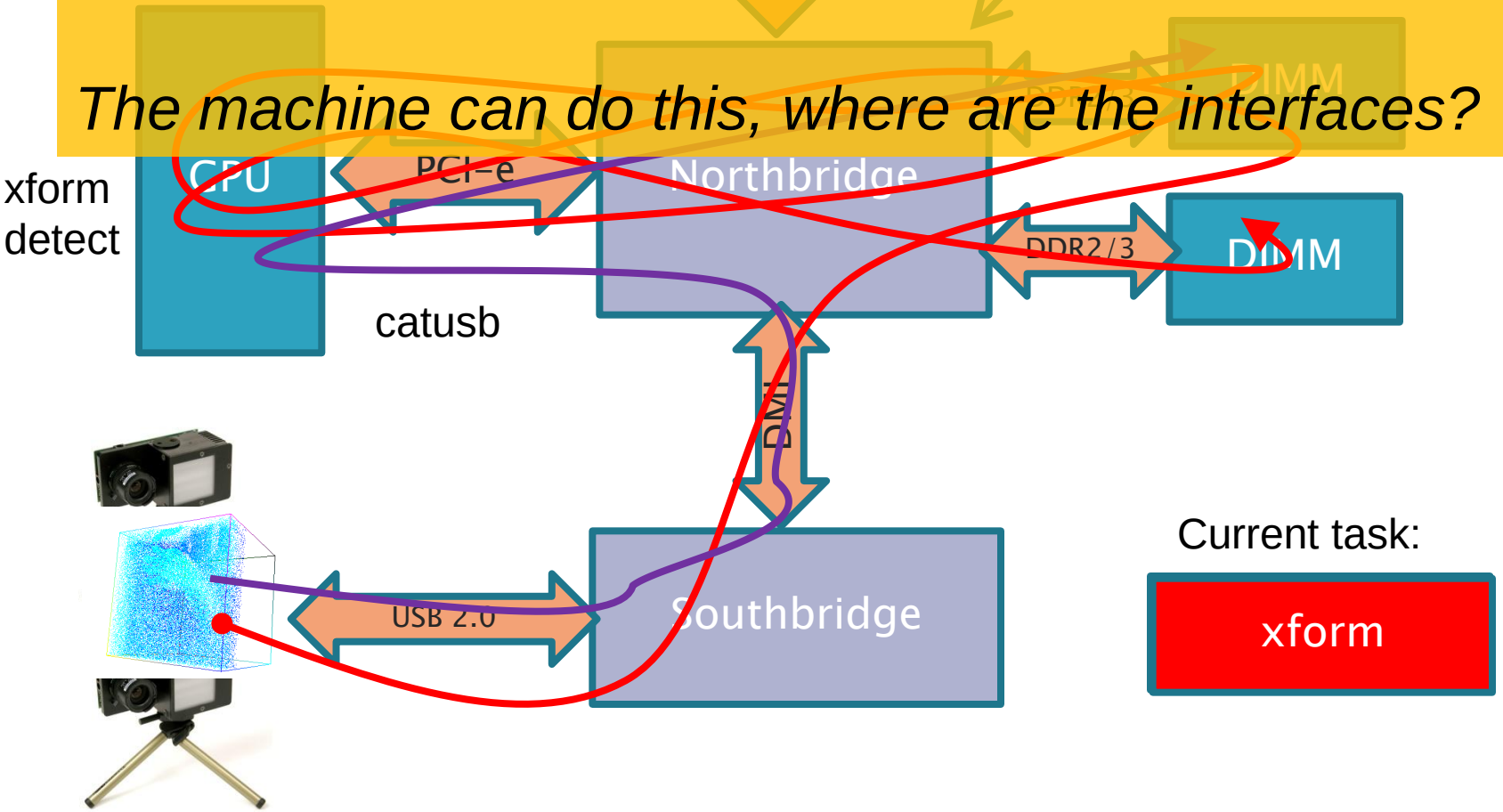
We'd prefer:

- catusb: USB bus → GPU memory
- xform, detect: **no** transfers
- hidinput: single GPU→main mem transfer

Cache pollution
Wasted bandwidth
Wasted power

hidinput

The machine can do this, where are the interfaces?



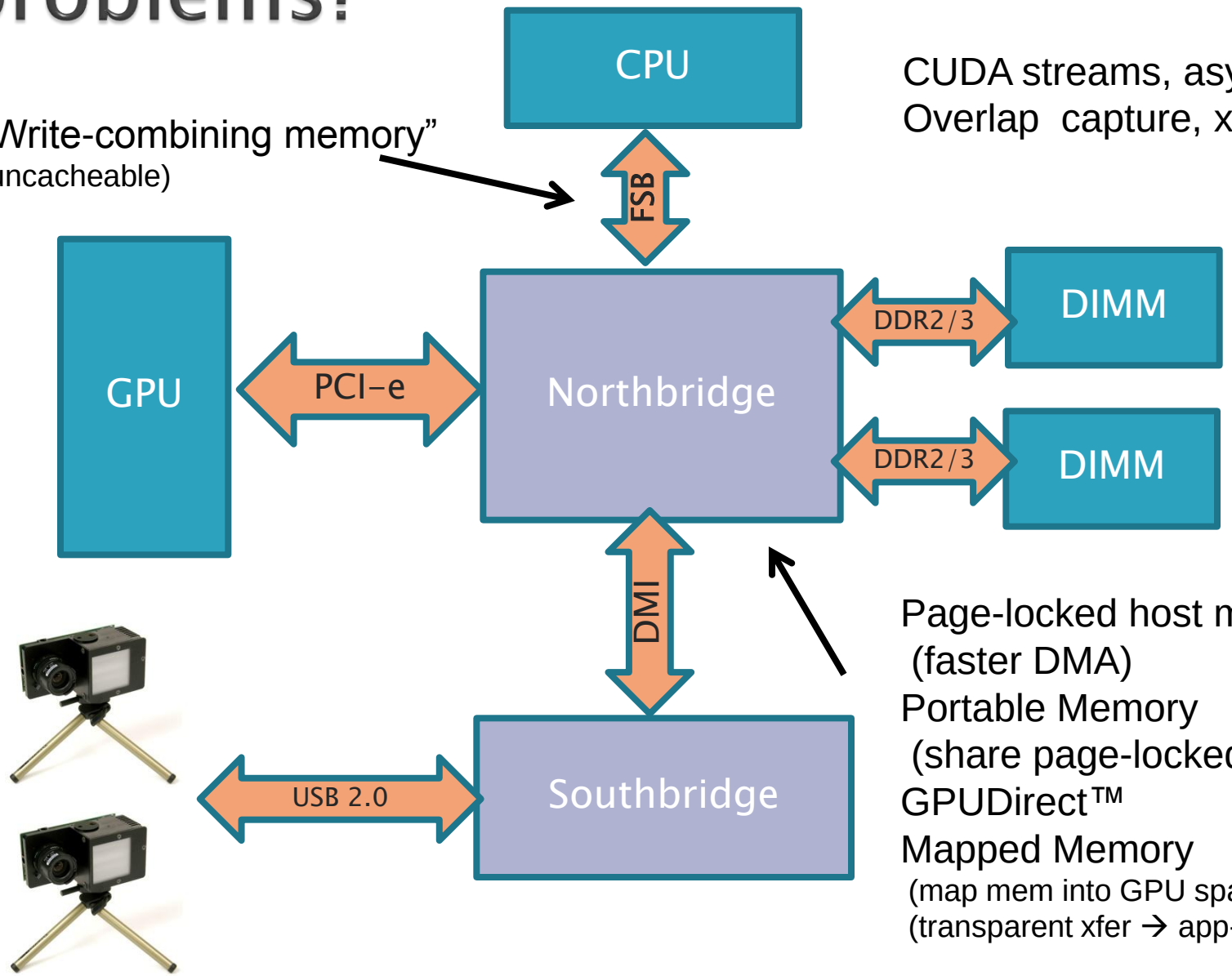
Current task:

xform

Outline

- Motivation
- The need for OS abstractions
- Why CUDA alone (currently) isn't enough
- New OS abstractions
- Related Work
- Conclusion

Doesn't CUDA address these problems?

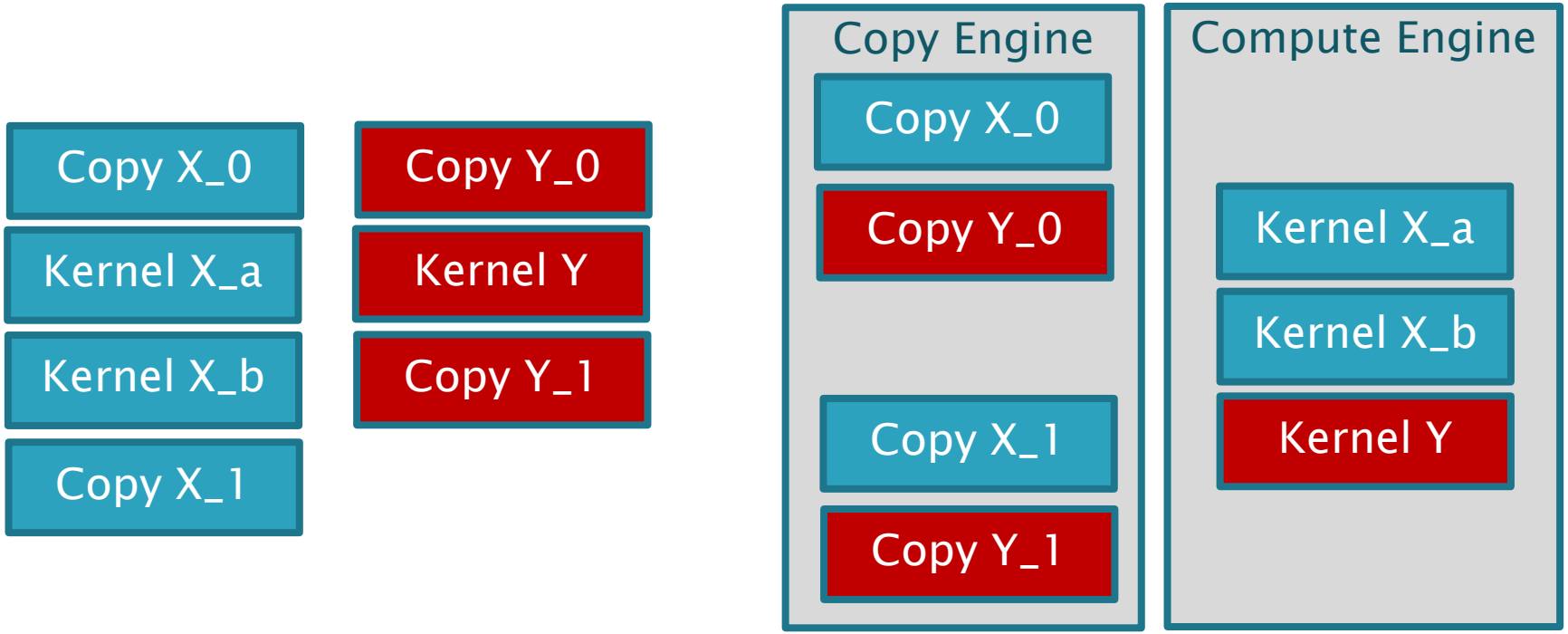


CUDA streams, async:
Overlap capture, xfer, exec

Page-locked host memory
(faster DMA)
Portable Memory
(share page-locked)
GPUDirect™
Mapped Memory
(map mem into GPU space)
(transparent xfer → app-level upcalls)

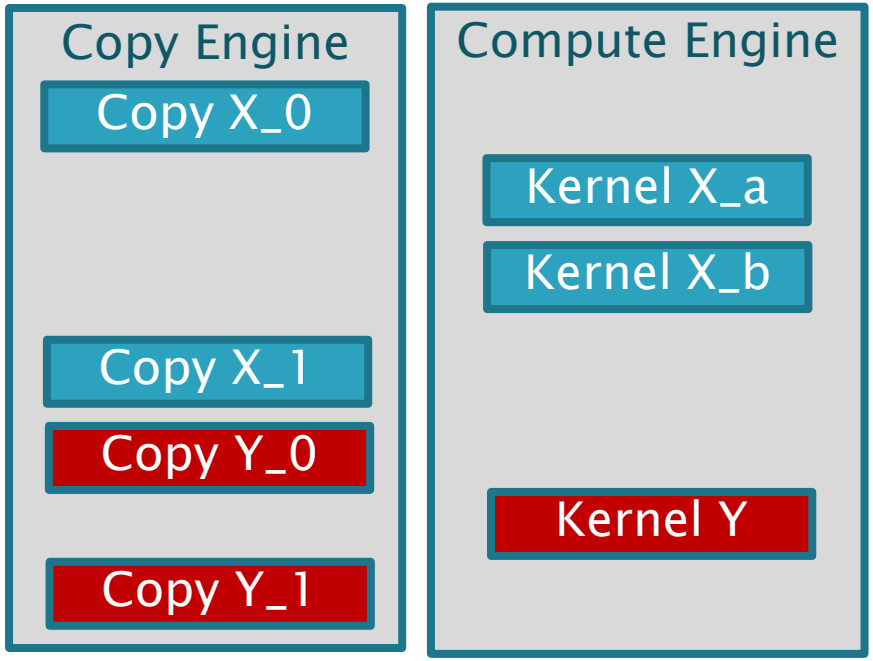
CUDA Streams

- Overlap Communication with Computation



Streams: explicitly scheduled

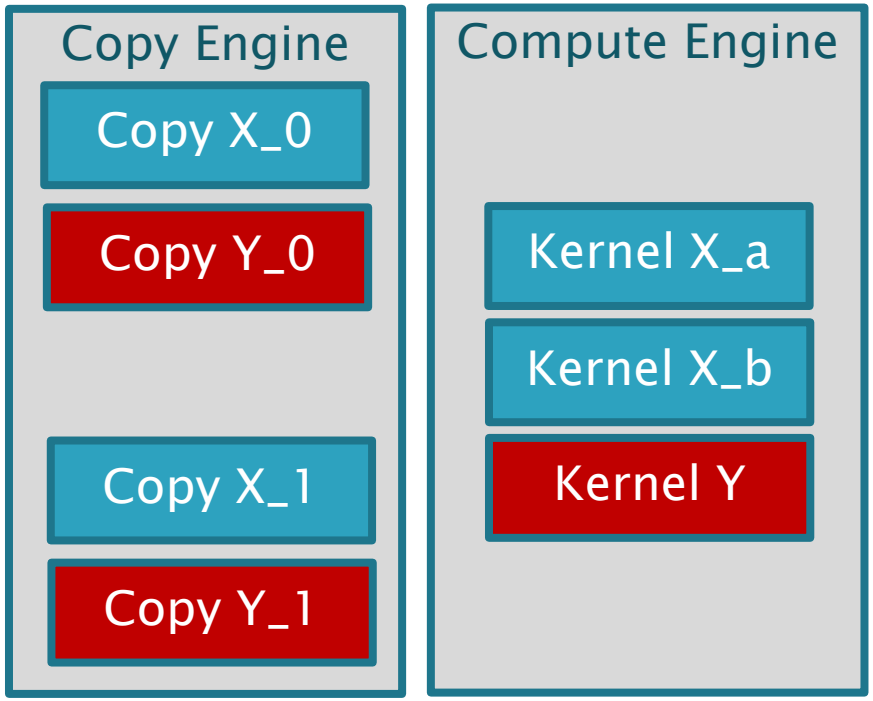
```
CudaMemcpyAsync(X_0…);
KernelX_0<<<…>>>();
KernelX_1<<<…>>>();
CudaMemcpyAsync(X_1…)

CudaMemcpyAsync(Y_0);
KernelY_0<<<…>>>();
CudaMemcpyAsync(Y_1);
```



Each stream proceeds serially, different streams overlap
Naïve programming eliminates potential concurrency

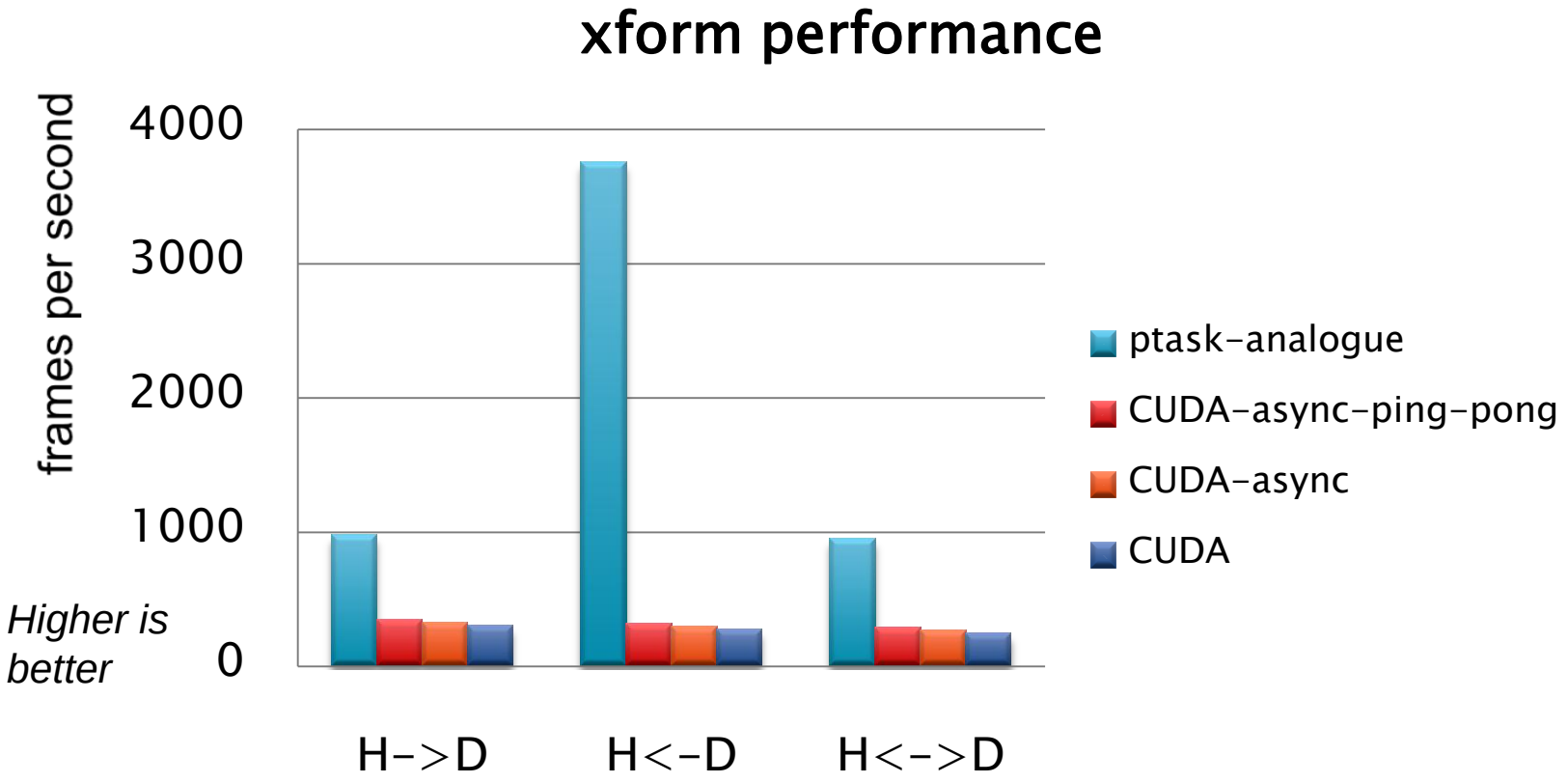
Reorder Code→better schedule

```
CudaMemcpyAsync(X_0…);
KernelX_0<<<…>>>();
KernelX_1<<<…>>>();

CudaMemcpyAsync(Y_0);
KernelY_0<<<…>>>();

CudaMemcpyAsync(X_1…)
CudaMemcpyAsync(Y_1);
```



- Order sensitive
- Applications must statically determine order
- ***Couldn't a scheduler with a global view do a better job dynamically?***

Asynchrony & CUDA

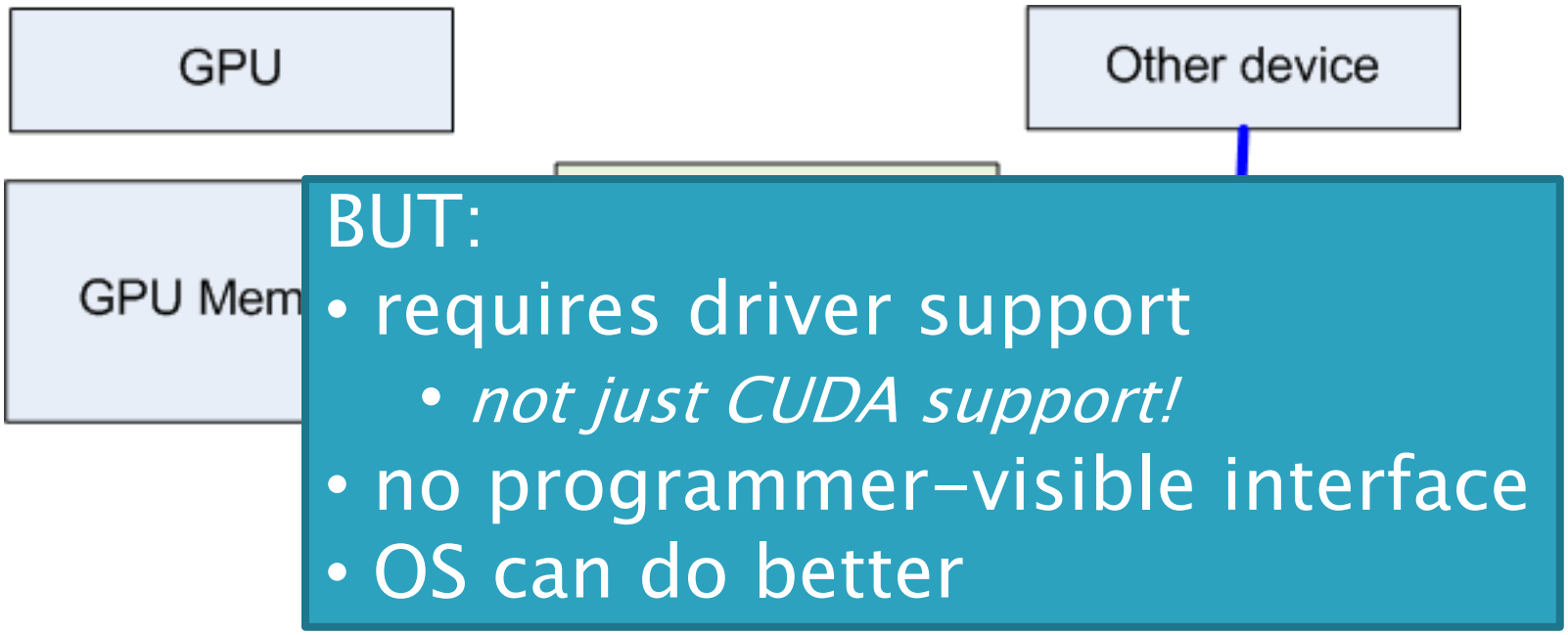


H→D: Host-to-Device only
H←D: Device-to-Host only
H ←→D: duplex communication

- Windows 7 x64 8GB RAM
- Intel Core 2 Quad 2.66GHz
- Nvidia GeForce GT230

GPUDirect™

- "Allows 3rd party devices to access CUDA memory": (eliminates data copy)

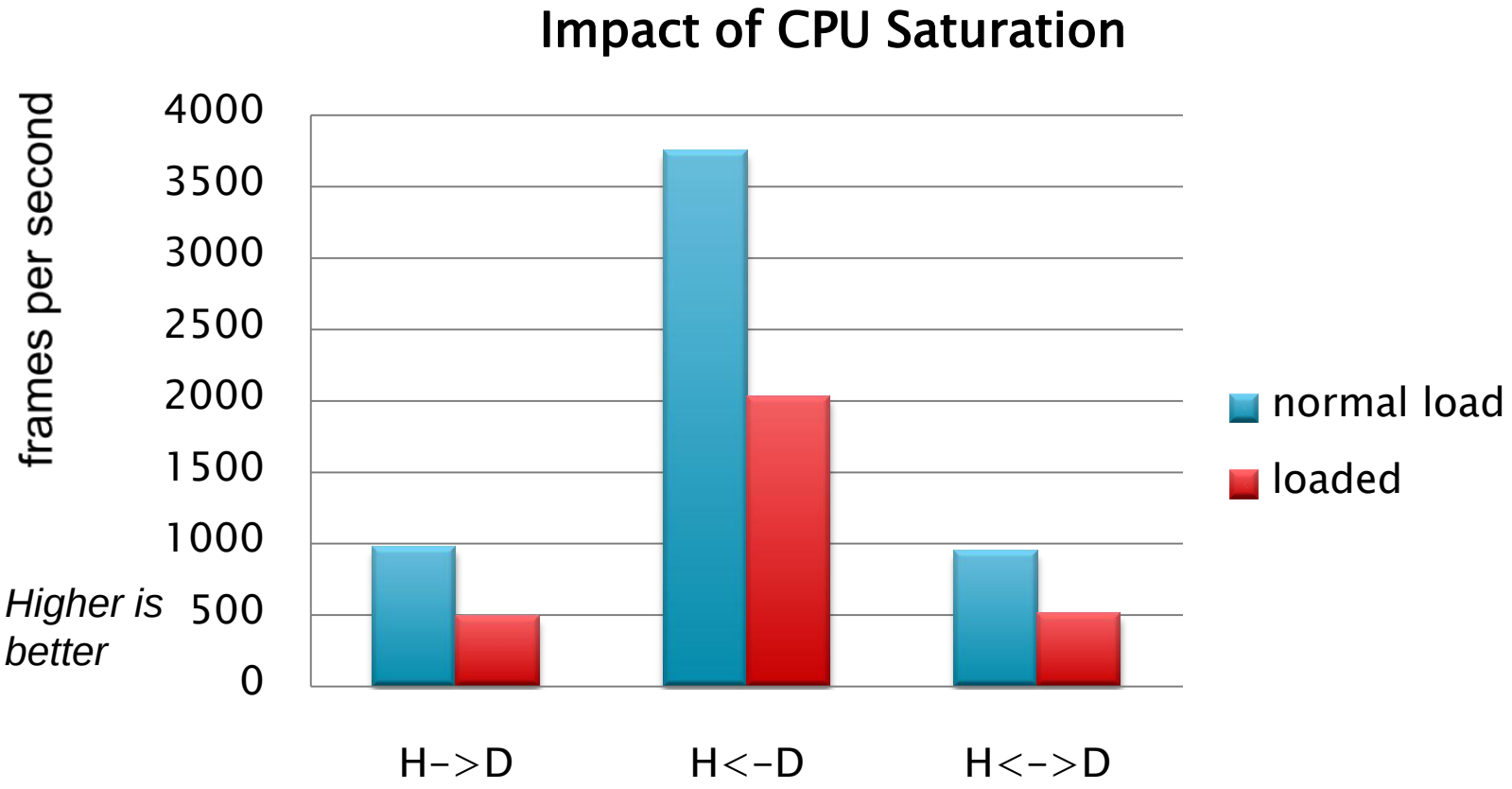


Problems CUDA cannot solve

Traditional OS guarantees:

- Fairness
- Isolation
- User-space runtime cannot provide these!
  - (Although they can stop your uncle Phil from violating them!)

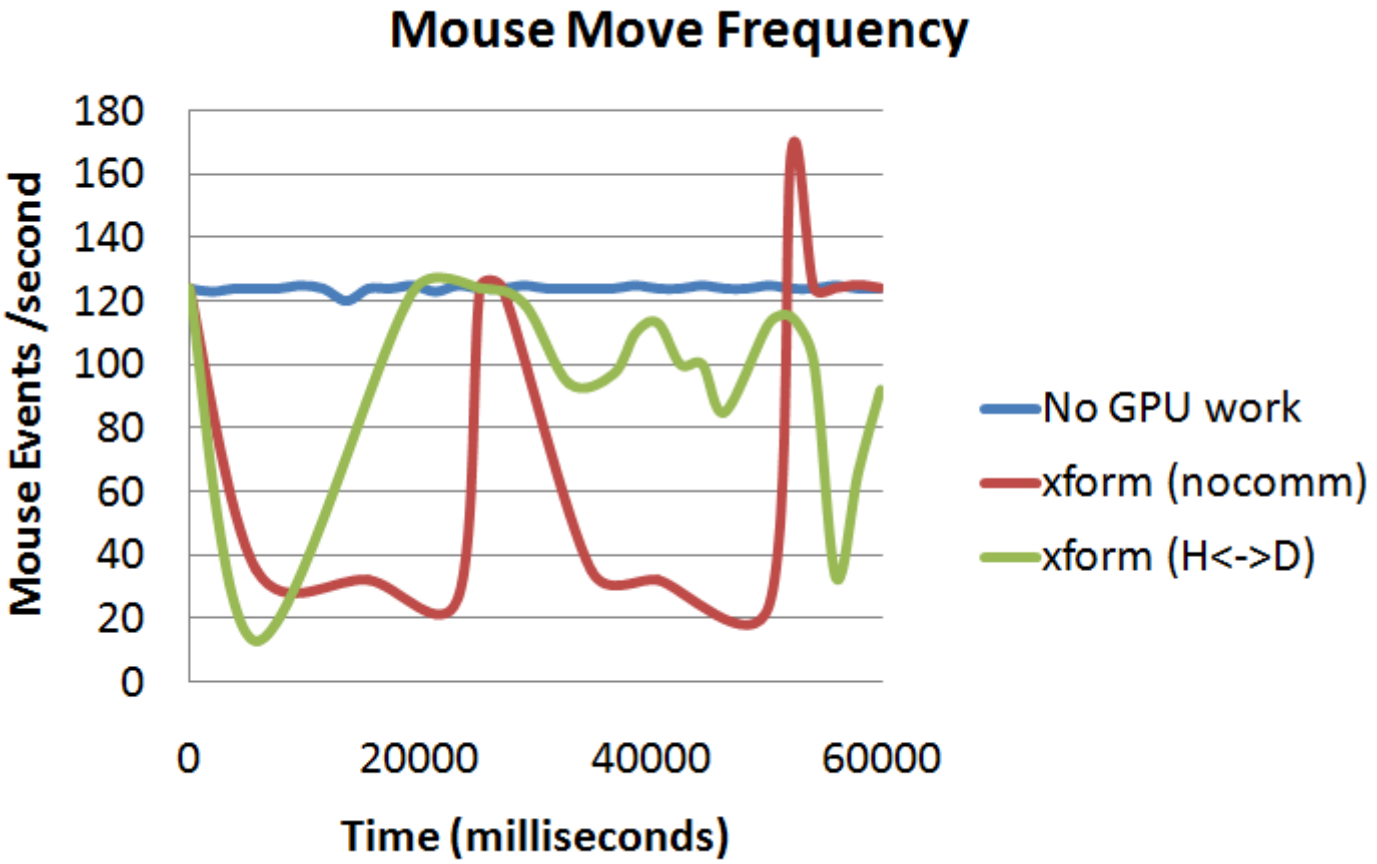
CPU-bound processes hurt GPUs



H→D: Host-to-Device only
H←D: Device-to-Host only
H ←→D: duplex communication

- Windows 7 x64 8GB RAM
- Intel Core 2 Quad 2.66GHz
- Nvidia GeForce GT230

GPU-bound processes hurt CPUs



Flatter lines
Are better

H→D: Host-to-Device only
H←D: Device-to-Host only
H ←→D: duplex communication

- Windows 7 x64 8GB RAM
- Intel Core 2 Quad 2.66GHz
- Nvidia GeForce GT230

Meaningful “GPU Computing” implies GPUs should be managed like CPUs

- Process API analogues
- IPC API analogues
- Scheduler hint analogues
- Must integrate with existing interfaces
  - CUDA/DXGI/DirectX
  - DRI/DRM/OpenGL

Outline

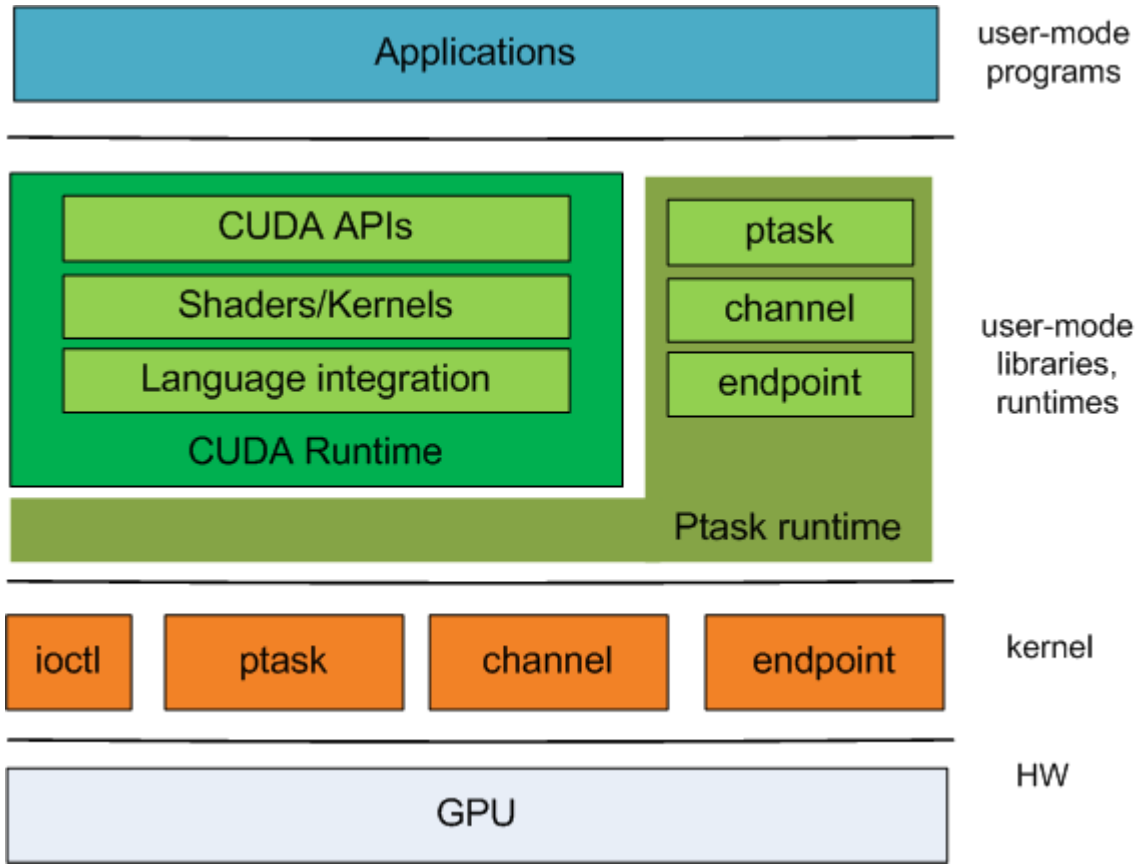
- Motivation
- The need for OS abstractions
- Why CUDA alone (currently) isn't enough
- New OS abstractions
- Related Work
- Conclusion

Proposed OS abstractions

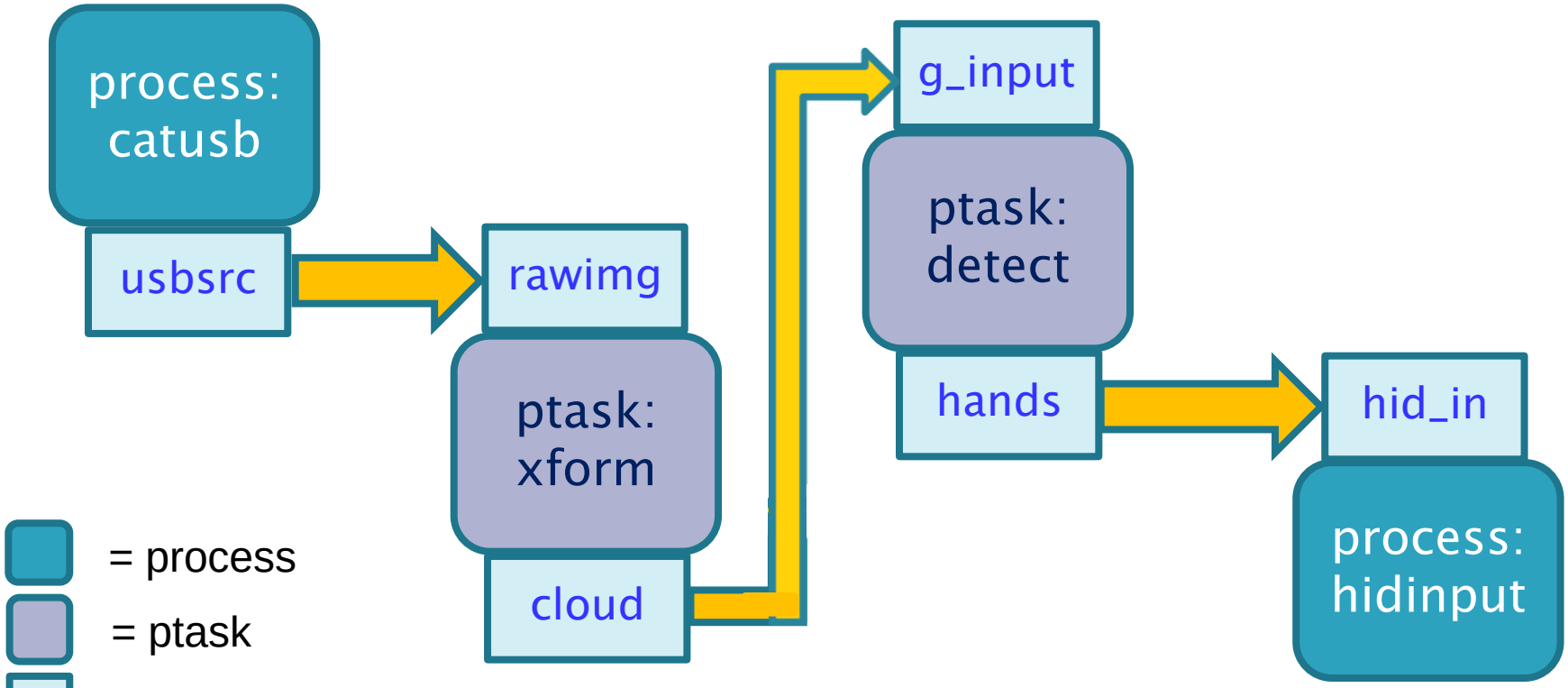
- **ptask**
  - Like a process, thread, ***can exist without user host process***
  - OS abstraction…not a full CPU-process
  - List of mappable input/output resources
- **endpoint**
  - Globally named kernel object
  - Can be mapped to ptask input/output resources
  - A data source or sink (e.g. buffer in GPU memory)
- **channel**
  - Similar to a pipe
  - Connect arbitrary endpoints
  - 1:1, 1:M, M:1, N:M
  - Generalization of GPUDirect™ mechanism

Expand system call interface:
- process API analogues
- IPC API analogues
- scheduler hints

Revised technology stack



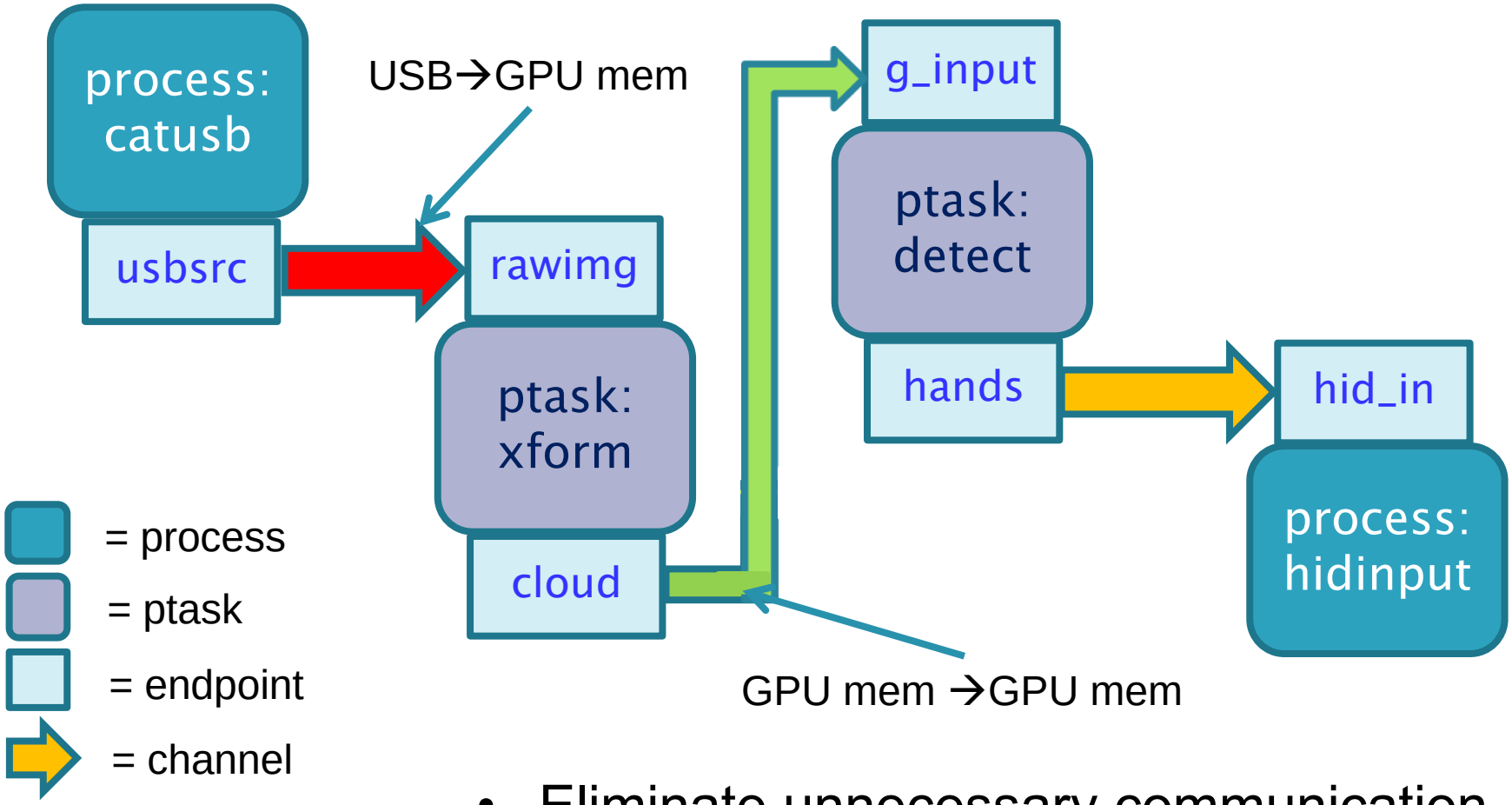
Gestural interface revisited



Computation expressed as a graph

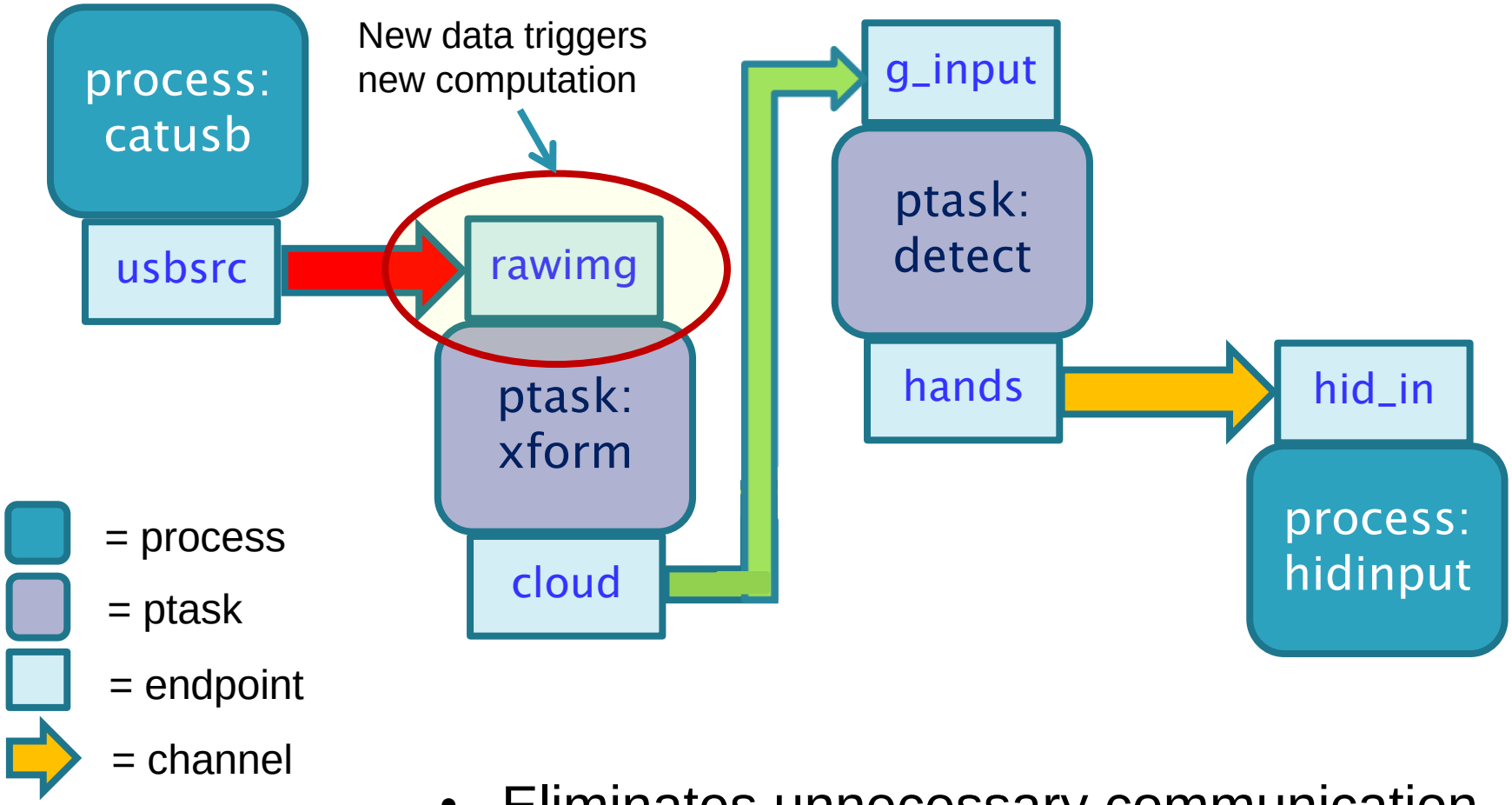
- *Synthesis* **[Masselin 89]** (streams, pumps)
- Dryad **[Isard 07]**
- Steamlt **[Thies 02]**
- Offcodes **[Weinsberg 08]**
- others…

Gestural interface revisited



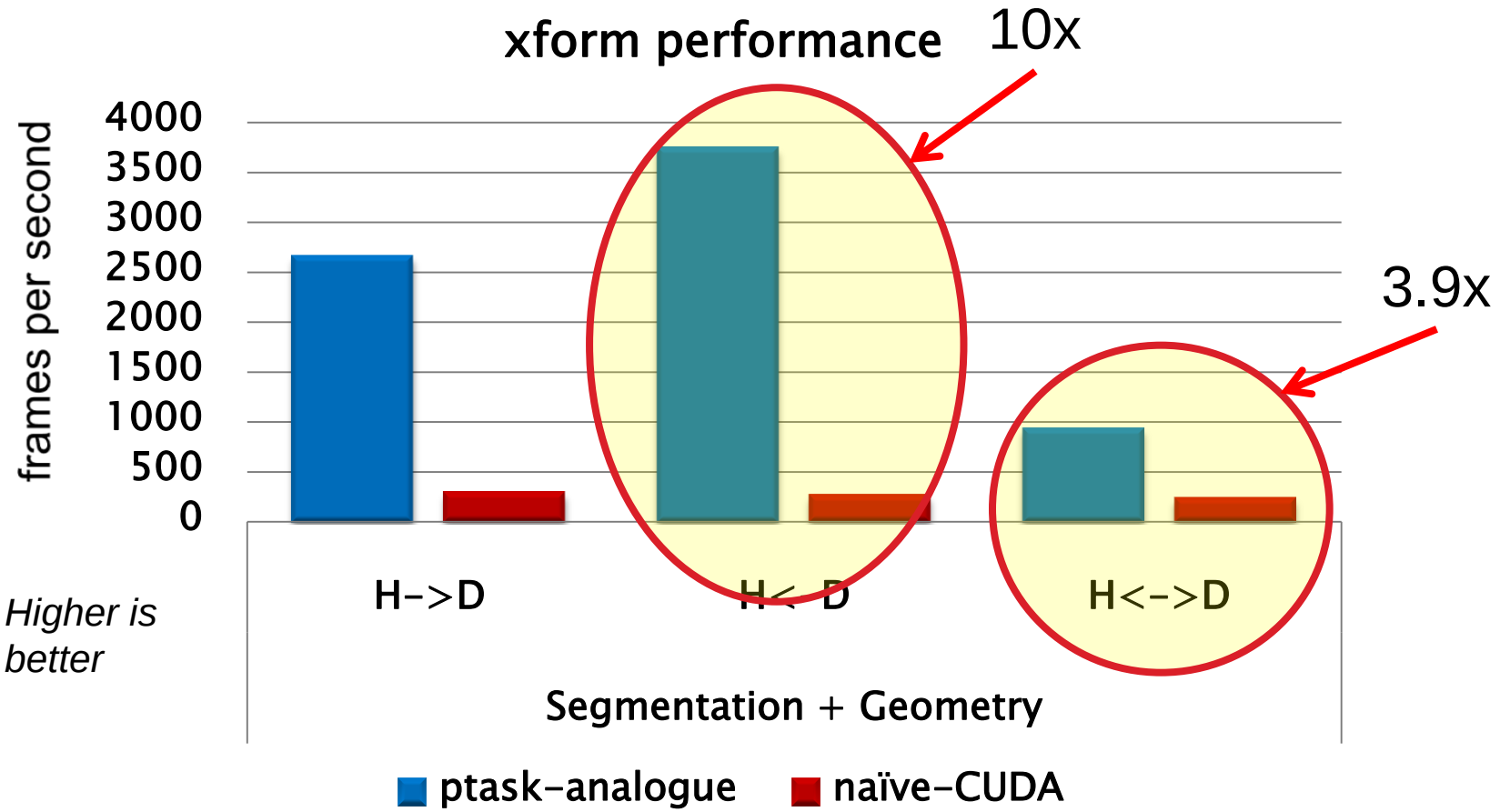
- Eliminate unnecessary communication…

Gestural interface revisited



- Eliminates unnecessary communication
- Eliminates u/k crossings, computation

Early Results: potential benefit



H→D: Host-to-Device only
H←D: Device-to-Host only
H ←→D: duplex communication

- Windows 7 x64 8GB RAM
- Intel Core 2 Quad 2.66GHz
- Nvidia GeForce GT230

Outline

- Motivation
- The need for OS abstractions
- Why CUDA alone (currently) isn't enough
- New OS abstractions
- Related Work
- Conclusion

Related Work

- OS support for Heterogeneous arch:
  - Helios [Nightingale 09]
  - BarrelFish [Baumann 09]
  - Offcodes [Weinsberg 08]
- Graph-based programming models
  - Synthesis [Masselin 89]
  - Monsoon/Id [Arvind]
  - Dryad [Isard 07]
  - StreamIt [Thies 02]
  - DirectShow
  - TCP Offload [Currid 04]
- GPU Computing
  - CUDA, OpenCL

Conclusions

- CUDA: programming interface is right,
  - ***but*** OS must get involved
- Need fairness, isolation
- Current interfaces waste data movement
- Current interfaces inhibit modularity/reuse
- Current interfaces limiting
  - interactive apps

Questions?

Backup slides…

Offcodes [Weinsberg 08]

- Similarities:
  - OS + GPU
  - Motivated by similar data migration issues
  - Graph-based programming model
- Differences
  - Host OS +target device firmware must support the same offcode API/runtime: device must run offcode runtime
  - NIC: TCP-Offload focused
  - didn't evaluate GPU
  - no scheduler integration
  - GPU still not a first class resource

Anatomy of a GPU shader program

- Cannot run OS: different ISA
- Host CPU must orchestrate execution
  - Disjoint memory space, no coherence
  - Program inputs explicitly bound at runtime

```
PS_OUTPUT DepthTransform( PS_INPUT In )
{
    PS_OUTPUT res;
    float4 sample = g_ABPhaseTexture.Sample(s, In.Tex);
    float4 xyzEntry = g_XYZCalibration.Sample(s, In.Tex);
    float abValue = sample[0];
    float zValueRaw = sample[1];
          …
    res[1] = xyzEntry[1] * zAdjust;
    res[2] = zAdjust;
    res[3] = abValue;
    return res;
}
```

fxc compiler

```
ps_4_0
dcl_input linear v1.xy
dcl_output o0.xyzw
dcl_constantbuffer cb0[276], dynamicIndexed
dcl_resource_texture2d ( float , float , float , float ) t0
dcl_resource_texture2d ( float , float , float , float ) t1
dcl_sampler s0, mode_default
dcl_sampler s1, mode_default
dcl_temps 2
sample r0.xyzw, v1.xyxx, t0.xyzw, s0
sample r1.xyzw, v1.xyxx, t1.xyzw, s1
mul r0.y, r0.y, r1.z
mul r0.y, r0.y, l(0.000061)
if_nz cb0[264].w
  ine r0.z, cb0[264].z, l(0)
  lt r0.x, r0.x, cb0[0].w
  and r0.x, r0.z, r0.x
  if_nz r0.x
    mov o0.xyzw, l(0,0,0,0)
    ret
  endif
  mul r0.xz, r1.xxyx, r0.yyyy
  mul r0.z, r0.z, cb0[266].z
  mad r0.x, cb0[265].z, r0.x, r0.z
  mad r0.x, cb0[267].z, r0.y, r0.x
  add r0.x, r0.x, cb0[267].w
  lt r0.x, l(0.000000), r0.x
  if_nz r0.x
    mov o0.xyzw, l(0,0,0,0)
    ret
  endif
endif
ftoi r0.x, r0.y
imin r0.x, l(255), r0.x
mov o0.xyzw, cb0[r0.x + 3].xyzw
ret
// Approximately 27 instruction slots used
```

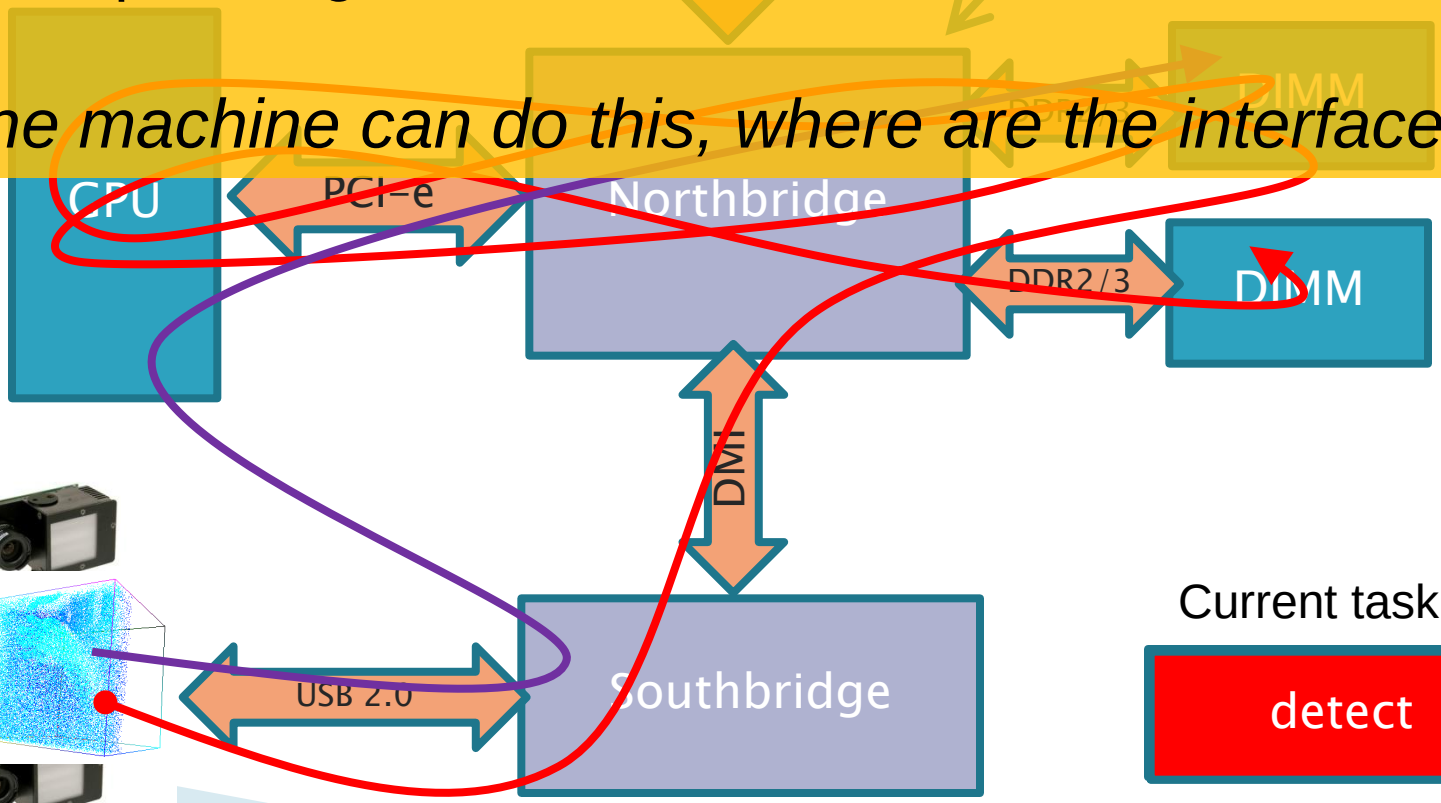
Hardware View

We'd prefer:

- catusb → GPU memory
- xform → detect no transfers
- hidinput: single GPU→main mem transfer

The machine can do this, where are the interfaces?

Cache pollution
Wasted bandwidth
Wasted power



Current task:

detect

Revised technology stack