PTask: Operating System Abstractions To Manage GPUs as Compute Devices

C.J. Rossbach, J. Currey - Microsoft Research
B. Ray, E. Witchel - University of Texas
M.Silberstein - Technion

Presentation: Adam Karczmarz

Outline

1. Overview & motivation (a long one).
2. Design.
3. PTask API.
4. Implementation details.
5. Evaluation.

GPU as a computing resource

- **G**raphic **P**rocessing **U**nit
- Great for rendering graphics/gaming...
- ... but also heavy parallel computations.

General purpose GPU frameworks

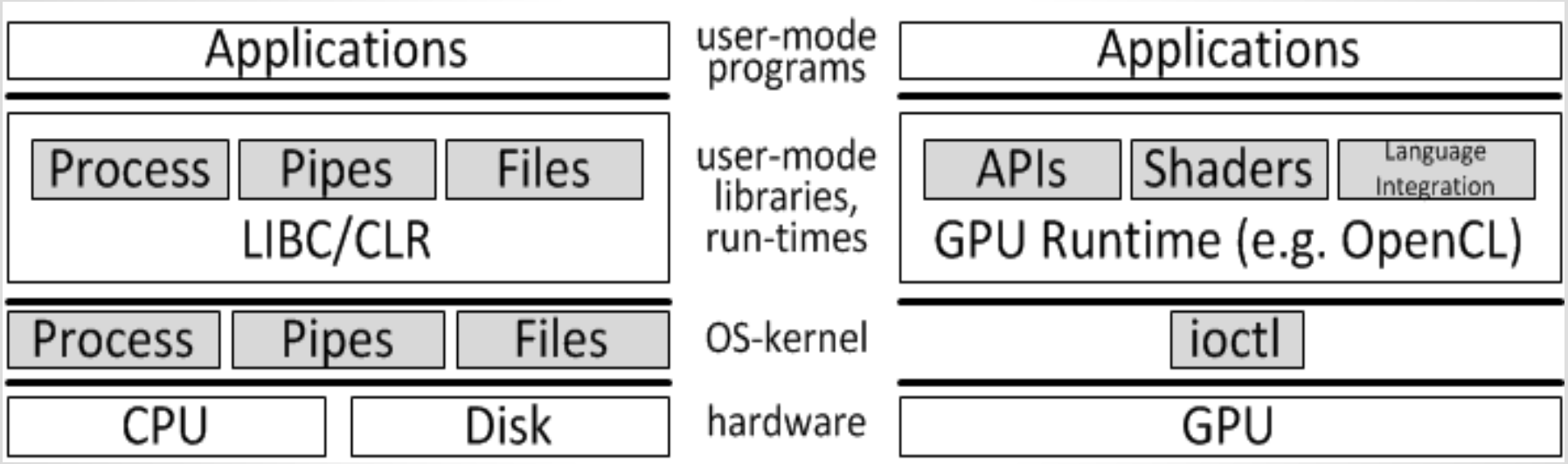
- sufficient for high-performance batch computations, e.g. scientific

New GPGPU applications?

- compute-intensive interactive apps: gestural input, real-time video recognition...
- own OS computation, such as encrypted file system.

- problem: lack of proper OS-level abstractions and treating GPU as a peripheral I/O device makes it hard...

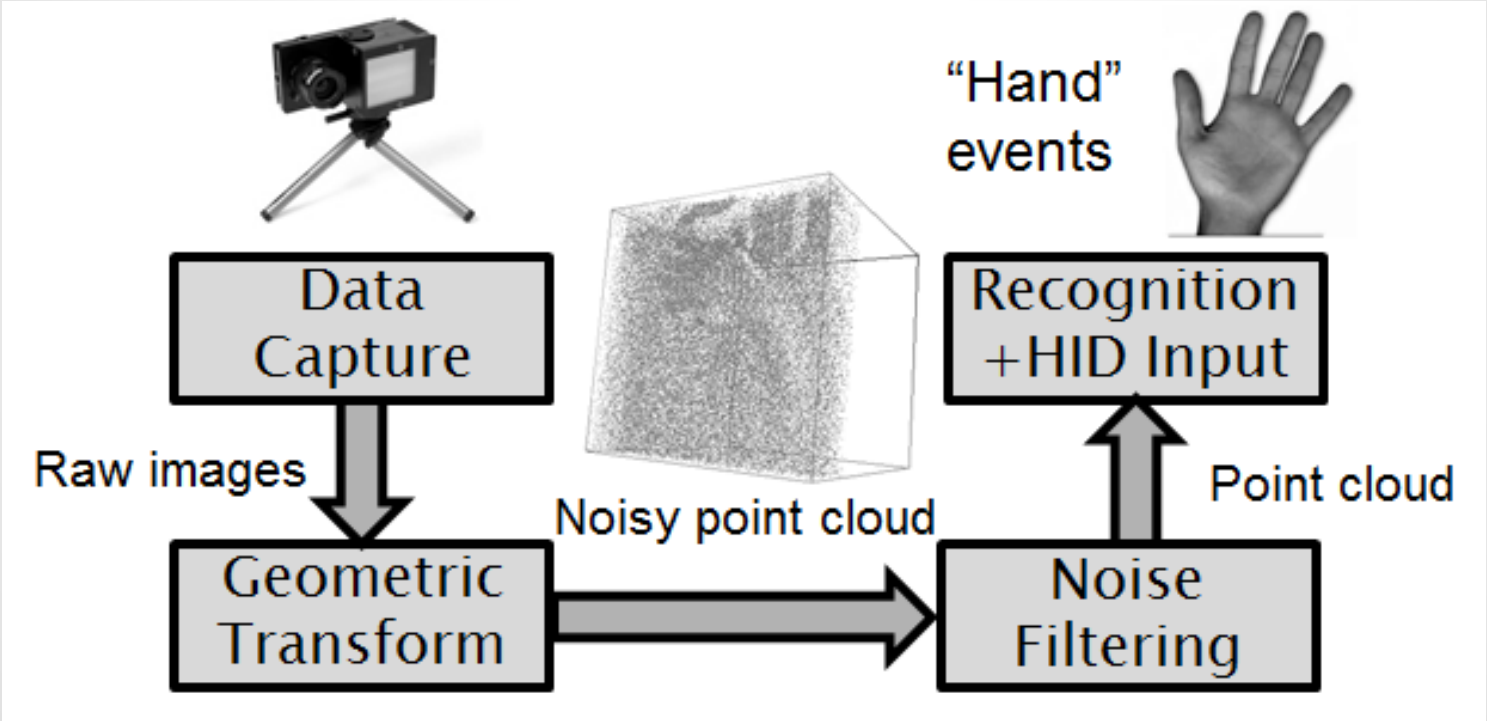
Technology stack for CPU vs GPU programs



Motivation - case study: gestural recognition

- computationally demanding task,
- real-time constraints,
- rich with data-parallel algorithms...
- a great fit for GPU acceleration!

Gesture recognition system



Gesture recognition system

- Ideally, we should be able to decompose the system into four (separate program) components:
  - `catusb:` captures data from USB cameras,
  - `xform`: perform geometric transformations to transfer multiple camera perspectives into a single point cloud. Data-parallel phase,
  - `filter`: Noise filtering on the point cloud data. Data parallel,
  - `hidinput`: detect gestures and send them to the OS as human interface device (HID) input. Not data-parallel.

Gesture recognition system - usage

- The nice modular design makes the components reusable,
- Just type:
  - `catusb | xform | filter | hidinput &`

  and enjoy gestural control

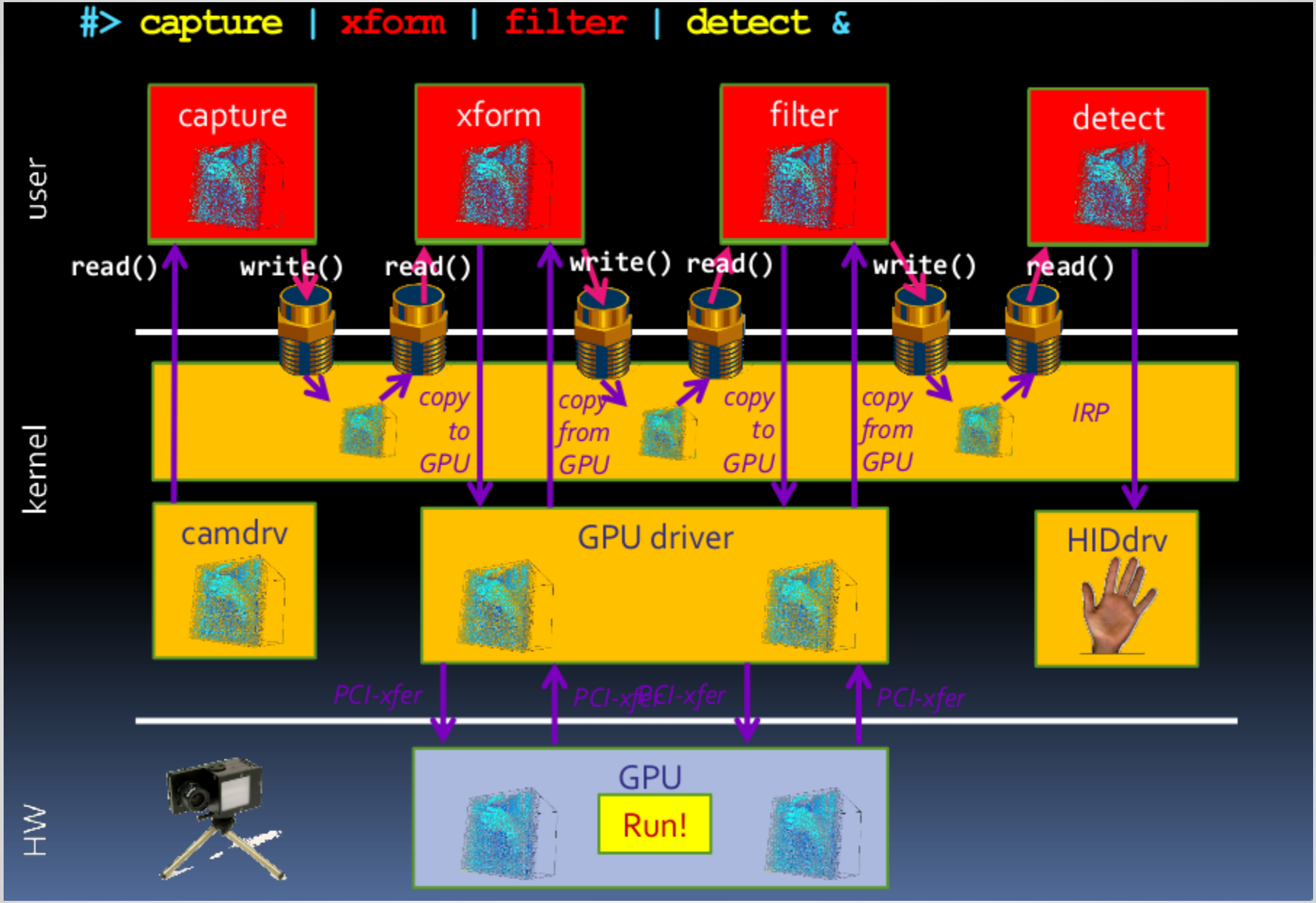
Gestural recognition system

- prototype `xform` and `filter` implementations show that running them on GPU is a great speed-up...
- actually, GPU acceleration is **required** for each of them: 4-core multiprocessor is unable to maintain real-time frame rates, consuming nearly 100% of available CPU
- GPU implementation has minimal effect on CPU utilization.

However...

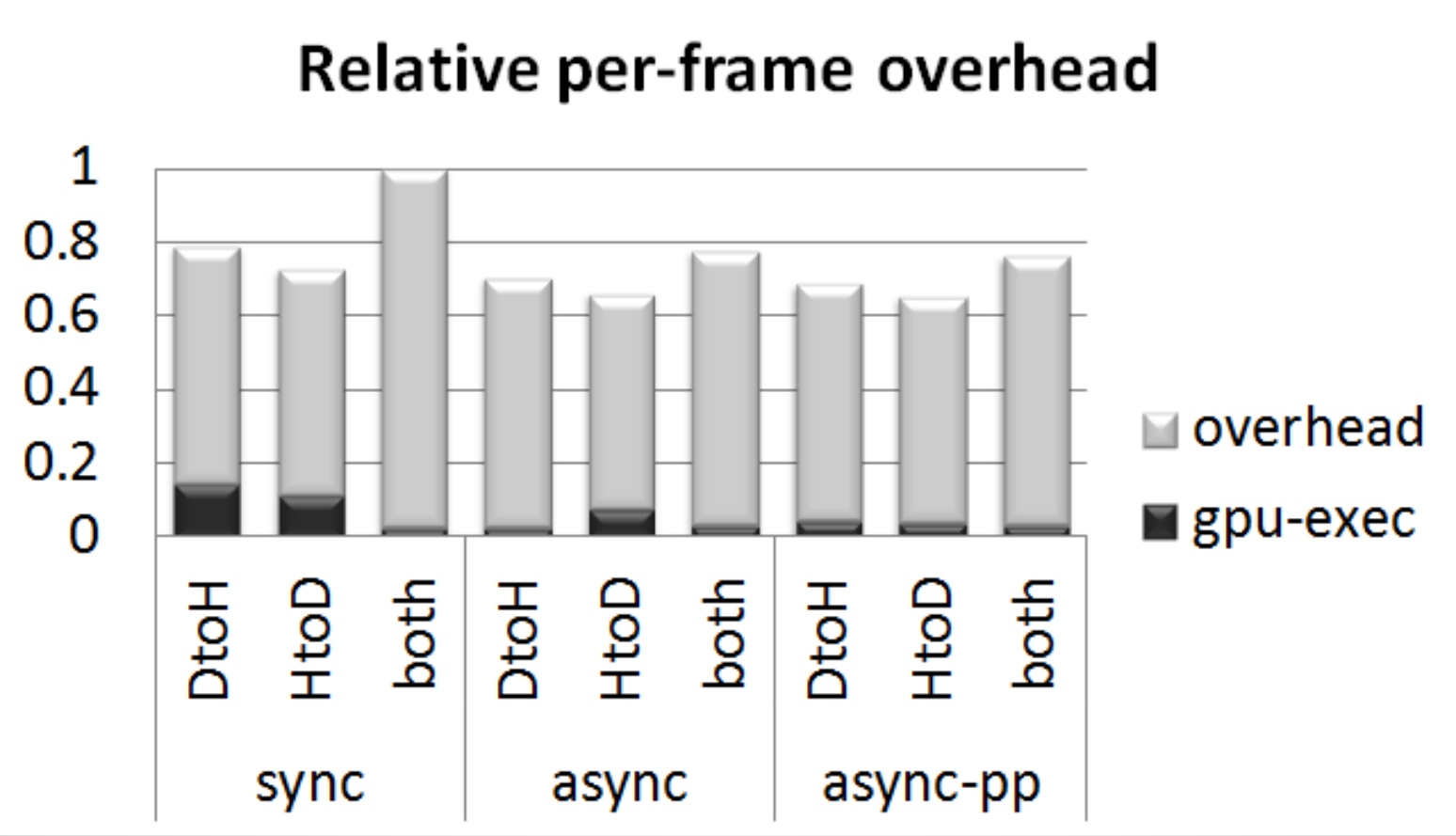
- Any of the GPGPU frameworks requires the main memory data to be transferred to the device before the computation and then back to the host to be read so...
- Running our nice pipeline
  - `catusb | xform | filter | hidinput &`

  suffers from excessive data movement - both across the user-kernel boundary and from main memory to GPU memory.



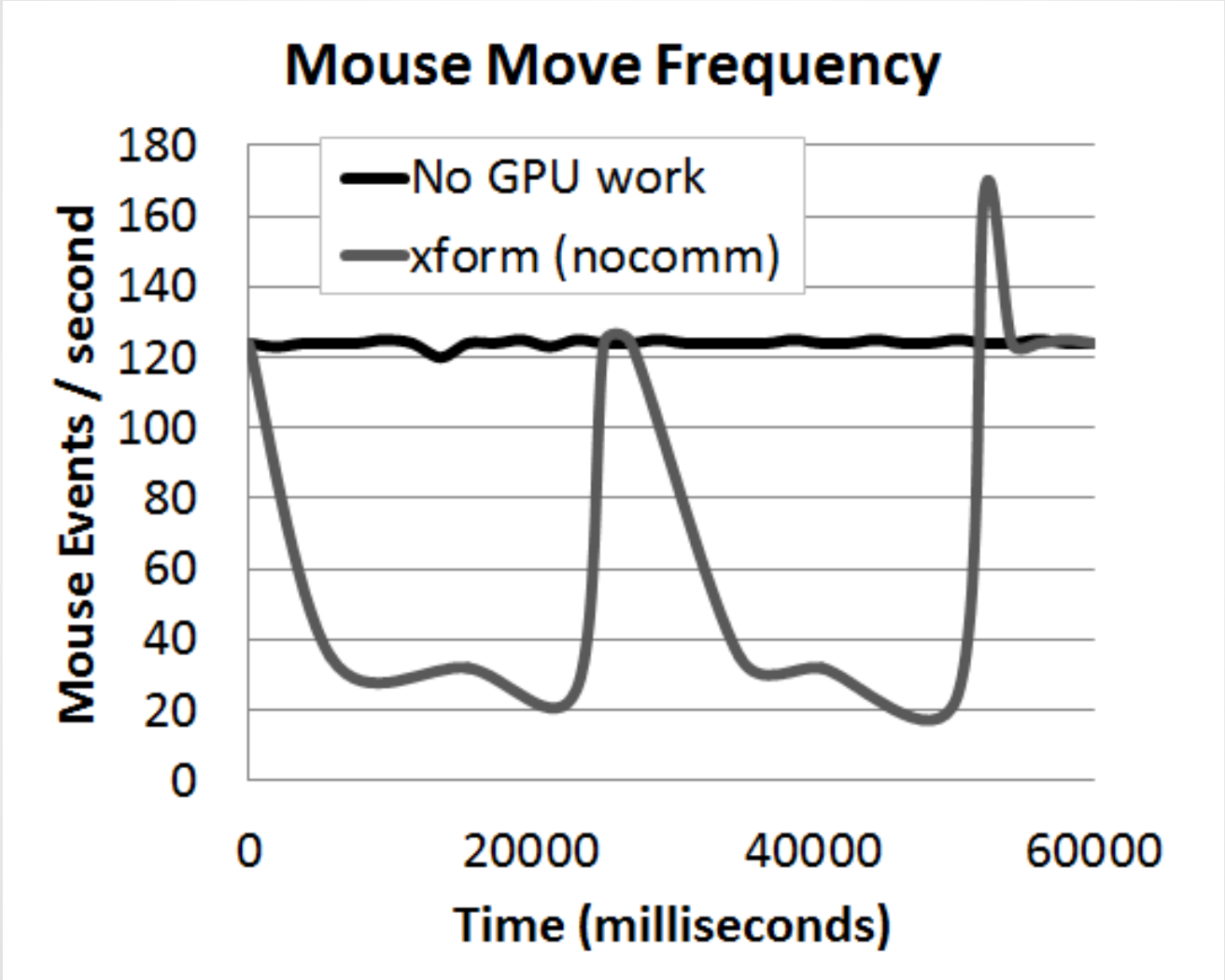
From the presentation PTask: OS Support for GPU Dataflow Programming by C. Rossbach, J. Currey

xform - memory movement overhead CUDA-based implementation



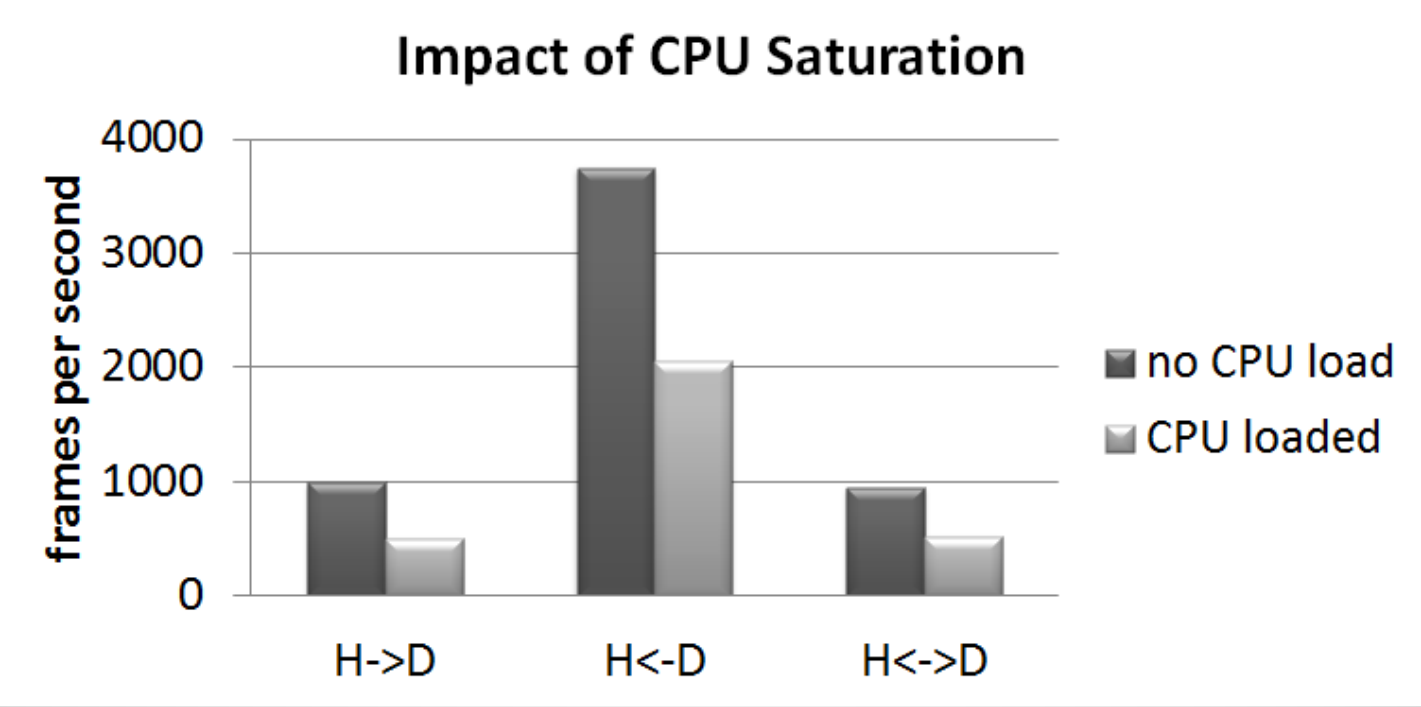
Scheduling problem #1

- Example: Windows 7 uses GPU for its own computation (Aero interface) and maintains screen refresh rates but...
- It relies on cancellation to prioritize its work.
- But the GPU I/O requests cannot be preempted once started. Running many GPU-bound tasks in a batch makes the system unresponsive...


Mouse Move Frequency
180
160
140
120
100
80
60
40
20
0
Mouse Events / second
No GPU work
xform (nocomm)
0
20000
40000
60000
Time (milliseconds)

Scheduling problem #2

- CPU work interferes with GPU throughput - Windows fails to load balance unrelated CPU-bound and GPU-bound tasks.



Conslusion

- New OS abstractions needed!
- Fairness and performance isolation needed!
- Reduction of redundant data movement needed! Also abstracting away the details of data movement and I/O to let the programmer focus on algorithms and high level data flow.
- Support for modular code needed, without much loss in performance.

PTask - design

- Set of OS abstractions for GPU programming addressing our conclusions.
- A dataflow programming model.
- Many GPUs transparent to the programmer.
- GPU tasks organized by the programmer into a DAG featuring:
  - vertices corresponding to tasks (called ptasks),
  - edges representing data flow, connecting the inputs and outputs(ports) of nodes (called channels).

PTask - efficiency vs modularity

- Imagine that we want to multiply two matrices A, B with the GPU:

```
matrix mult(A, B) {
   matrix res = new matrix();
   copyToDevice(A);
   copyToDevice(B);
   invokeGPU(mult_kernel, A, B,
res);
   copyFromDevice(res);
   return res;
}
```

PTask - efficiency vs modularity

- Now, imagine that we want to multiply three matrices A, B, C. The modular solution would be...

```
matrix modularSlowAxBxC(A, B, C) {
   matrix AxB = mult(A, B);
   matrix AxBxC = mult(AxB, C);
   return AxBxC;
}
```

PTask - efficiency vs modularity

- ... but the efficient one would be:

```
matrix nonmodularFastAxBxC(A, B, C) {
    matrix intermed = new matrix();
    matrix res = new matrix();
    copyToDevice(A);
    copyToDevice(B);
    copyToDevice(C);
    invokeGPU(mult_kernel, A, B, intermed);
    invokeGPU(mult_kernel, intermed, C, res);
    copyFromDevice(res);
    return res;
}
```

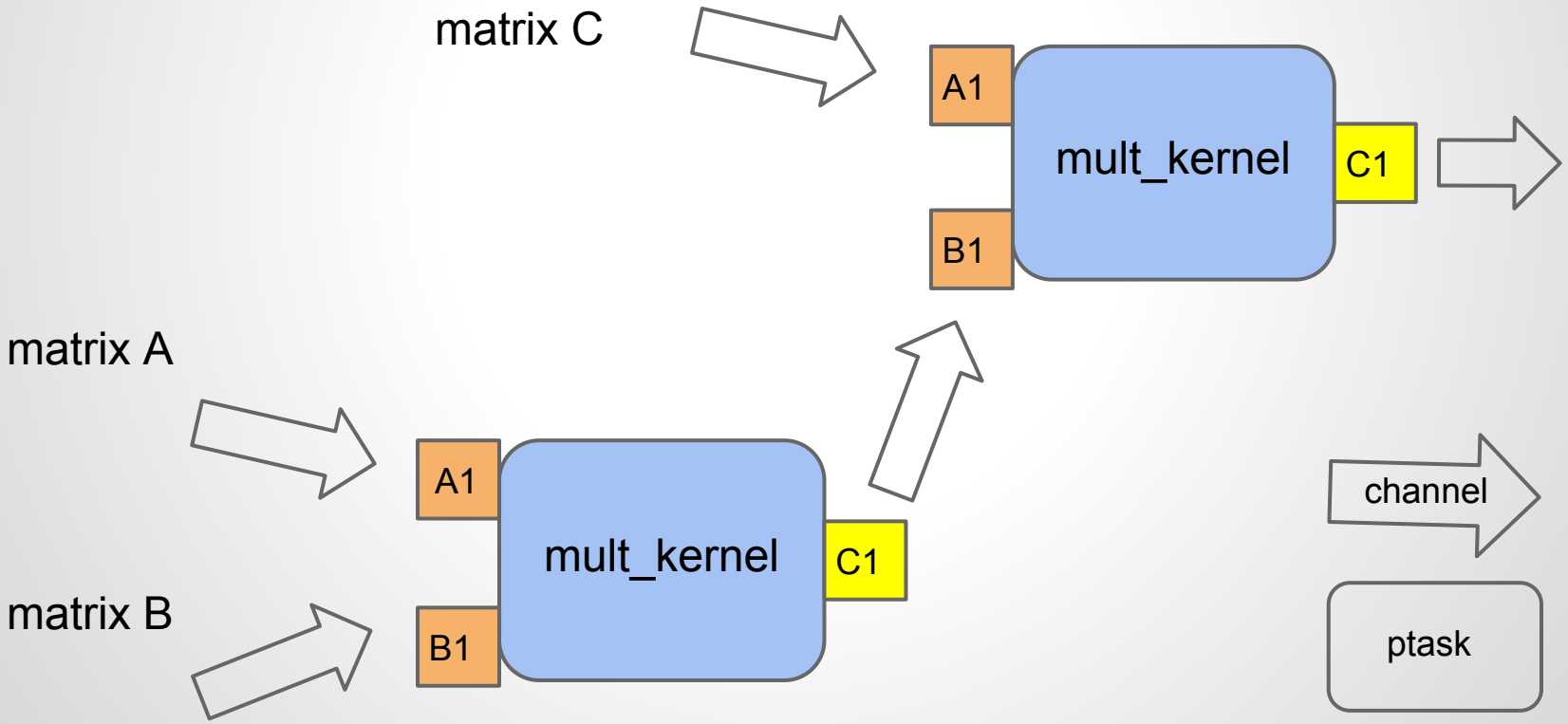
- this code is no longer modular and reusable.

PTask - efficiency vs modularity

- The modularity problem could be easily solved within one program.
- But in our example, data moves between many programs and resources, so modularity is an OS-level issue.

- PTask abstracts away data movements completely, automatically avoiding redundant copies.

Matrix multiplication - PTask way



PTask limitations

- PTask graph needs to be acyclic,
- We cannot explicitly express loops or recursion,
- The graph cannot be changed once run.

PTask API abstractions

- Abstractions used in detail:
  - ptask - a process analogue, runs substantially on a GPU,
  - port - a object in the kernel namespace that can be bound to input and output resources.
  - channel - analogous to a POSIX pipe, connects ports to each other or to other data sources/sinks in the system. An input port can connect to only one channel, while the output port to multiple ones.
  - graph - a bunch of ptasks connected by channels. There can be many graphs running at once.

PTask API abstractions cont'd

- Abstractions used in detail:
  - datablock - a virtual buffer that stores information about where does the up-to-date version of the piece of data reside in the main/gpu memory. This information allows to avoid redundant data movements.
  - template - a metadata that describes raw data in a datablock: type of the resource, dimensions and layout of the data.

PTask invocation

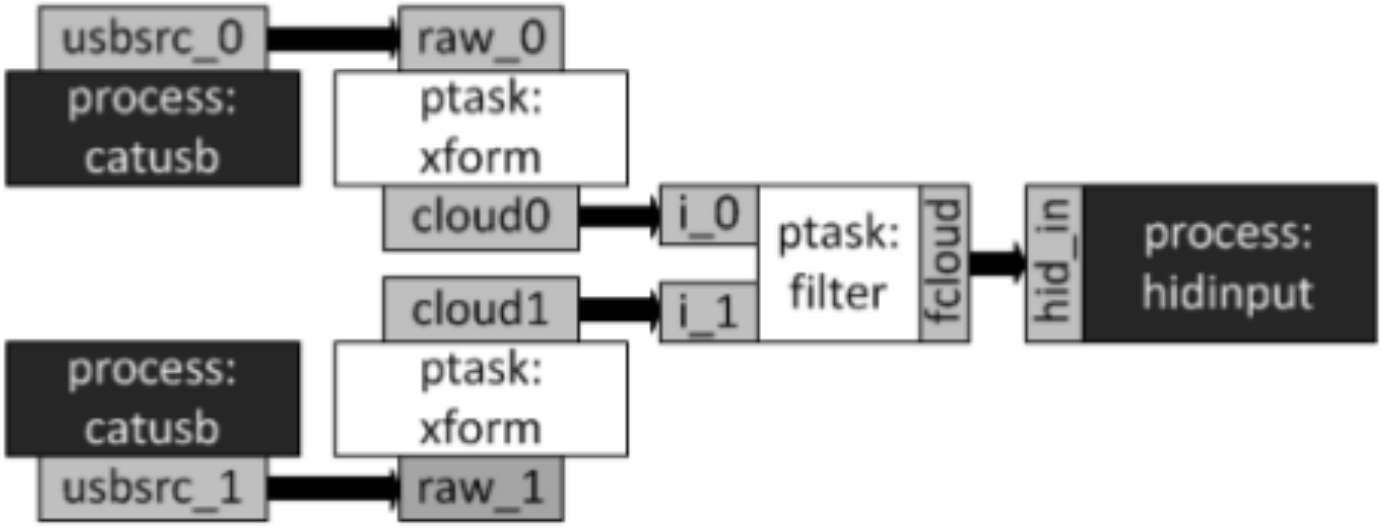
- A ptask can be in one of four states:
  - Waiting (for inputs),
  - Queued (inputs available, waiting for GPU),
  - Executing,
  - Completed (waiting for output consumption).

- A PTask is invoked if it's at the head of the queue and a capable device is available.

PTask API system calls

PTask system call	Description
`sys_open_graph`	Create/open a graph
`sys_open_port`	Create/open a port
`sys_open_ptask`	Create/open a ptask
`sys_open_channel`	Create and bind a channel
`sys_open_template`	Create/open a template
`sys_push`	Write to a channel/port
`sys_pull`	Read from a channel/port
`sys_run_graph`	Run a graph
`sys_terminate_graph`	Terminate graph
`sys_set_ptask_prio`	Set ptask priority
`sys_set_geometry`	Set iteration space

Gestural interface PTask graph



PTask implementation - scheduling

- Challenges:
  - non-preemptive GPUs, no context switches,
  - no OS interface to control the GPU in Windows,
  - in case of many GPUs in the system, parallel execution may not be profitable, as the data migration overhead may be greater than the latency reduction coming from concurrency.

Implemented scheduling policies - Windows

- first-available,
- fifo,
- priority mode - every task is assigned a static priority, its manager thread - proxy priority, GPUs are chosen based on its strength,
- data aware - same as above, but the GPU is chosen based on how many ptask's input are up-to-date in the corresponding device's memory. Based on its priority, a ptask could be queued to wait for a preferred GPU.

Limitations of the PTask prototype

- It assumes that all the GPU computations use PTask API.
- It does not address the problem of memory demands exceeding GPU physical memory - GPUs support virtual memory, but also allow allocation of unswappable memory and PTask uses that.

Evaluation 1 - gestural interface on Windows 7

- All the following measures were taken on a 64bit Windows 7 desktop, 4-core Xeon@2.67Ghz, 6GB RAM, GTX 580 GPU with 512 cores and 1.5GB memory.
- Five gestural interface implementations compared:
  - host-based - GPU not used at all,
  - handcode - non-modular implementation, GPU heavily used, optimized data movements,
  - pipes - `catusb | xform | filter | hidinput`, xform and filter use GPU,
  - modular - the same as pipes, but implemented as a single program to eliminate data migrations between processes,
  - PTask.

Evaluation 1 - gestural interface on Windows 7

- Two comparison modes:
  - real-time - we measure utilization and end-to-end latency,
  - unconstrained - 1000 camera frames are replayed from memory, we measure throughput.

- We measure fps, throughput, latency, user and kernel CPU utilization, GPU utilization, GPU memory usage, additional threads and memory increase over the handcode version.

	impl		fps	tput (MB/s)	lat (ms)	user	sys	gpu	gmem	thrds	ws-delta
Core2-Quad	host-based	real-time	20.2	35.8	36.6	78.1	3.9	–	–	1	–
GTX580	handcode	real-time	30	53.8	10.5	4.0	5.3	21.0	138	1	–
		unconstrained	138	248.2	–	2.4	6.4	41.8	138	1	–
	modular	real-time	30	53.8	12.2	6.0	8.1	19.4	72	1	0.8 (1%)
		unconstrained	113	202.3	–	5.7	8.6	55.7	72	1	0.9 (1%)
	pipes	real-time	30	53.8	14.4	6.8	16.6	18.9	72	3	45.3 (58%)
		unconstrained	90	161.9	–	12.4	24.6	55.4	76	3	46.3 (59%)
	ptask	real-time	30	53.8	9.8	3.1	5.5	16.1	71	7	0.7 (1%)
		unconstrained	154	275.3	–	4.9	8.8	65.7	79	7	1.4 (2%)

Evaluation 2 - microbenchmarks: benefits of dataflow

- We compare four implementations of algorithms listed below on various dataflow graphs. The implementations are:
  - single-threaded modular - perform each task on the GPU, copying all the data after each step,
  - modular - same as above but with some overlap of computation and data movement,
  - handcode - optimized data movement at the cost of modularity,
  - PTask.

Evaluation 2 - microbenchmarks: benefits of dataflow

bnc	description	P_{min}	P_{max}	C_{min}	C_{max}
gemm	matrix multiply	4	60	14	190
sort	bitonic sort	13	41	29	84
madd	matrix addition	4	60	14	190
mcopy	matrix copy	2	30	7	95
fdtd	3d stencil	30	60	85	170
pca	principal components	229	1197	796	4140
grpby	group by	7	7	13	13

Table 5: Micro-benchmarks evaluated across a range of PTask graph sizes and shapes. P_{min} and P_{max} are the number of ptasks, while C_{min} and C_{max} are the number of channels in the smallest and largest graphs.

Evaluation 2 - microbenchmarks: benefits of dataflow

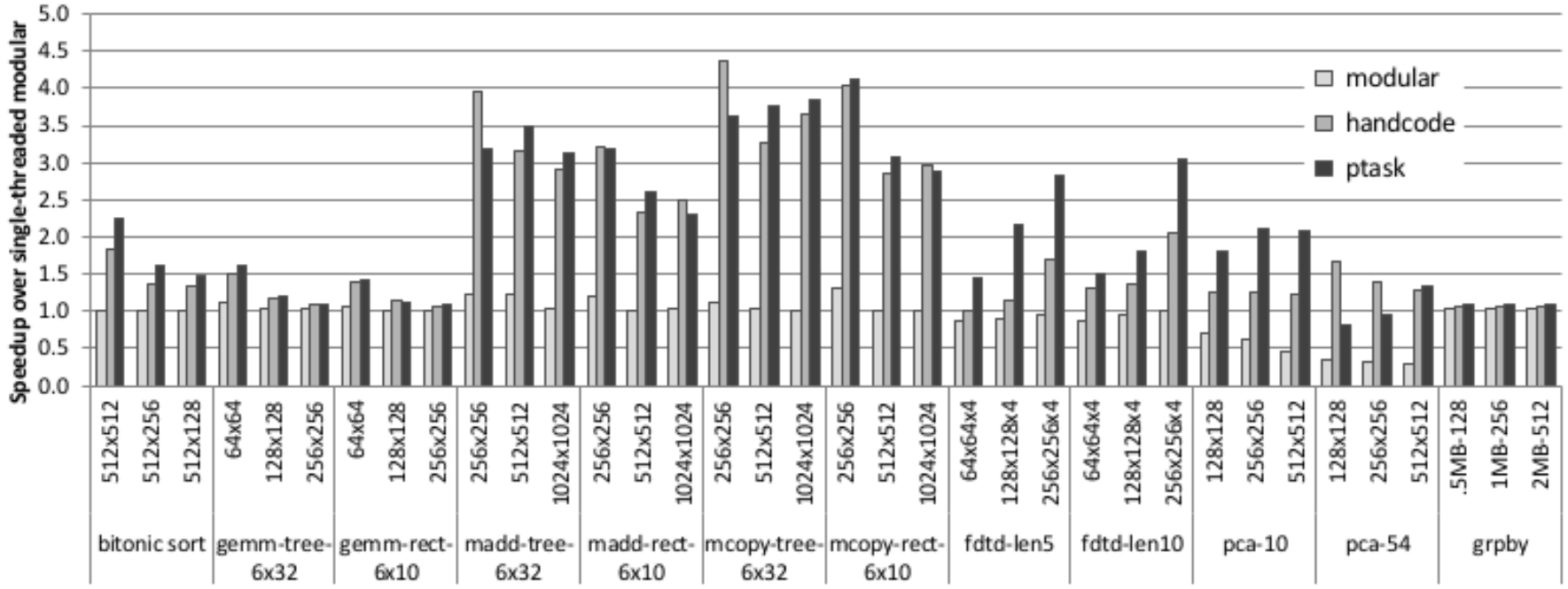
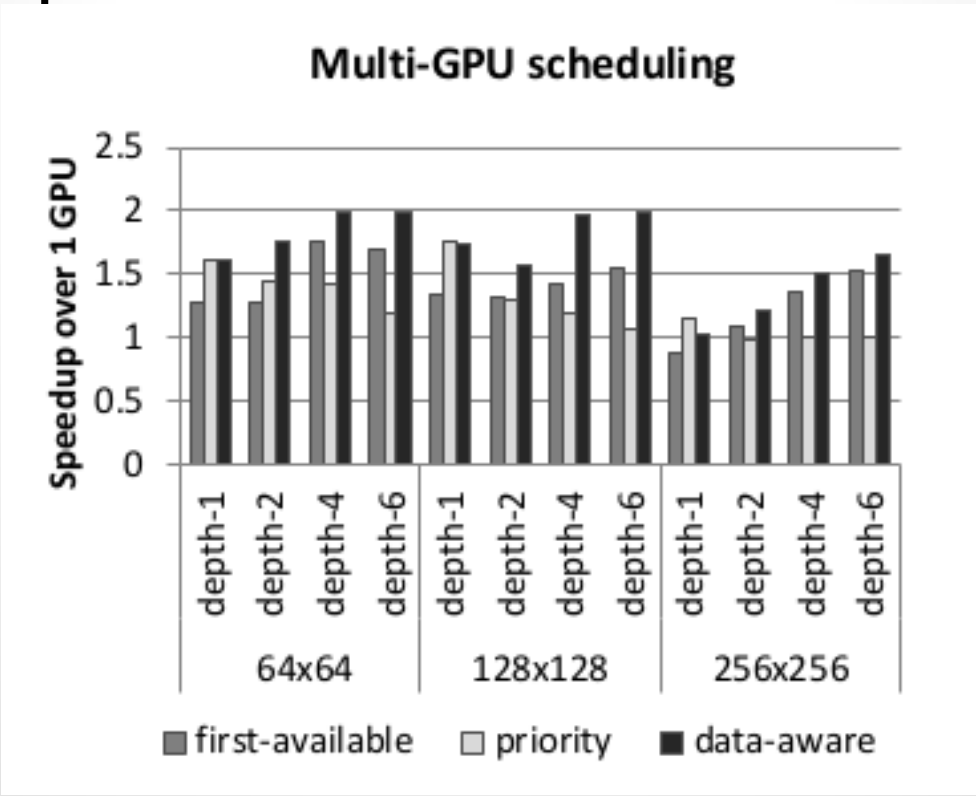


Figure 11: Speedup of various GPU-accelerated implementations of our benchmarks over single-thread, modular GPU-based implementations, for input matrix and image sizes ranging from 64x64 through 1024x1024. The PTask implementation has geometric mean speedup of 10% over handcoded, and $1.93\times$ over modular implementation.

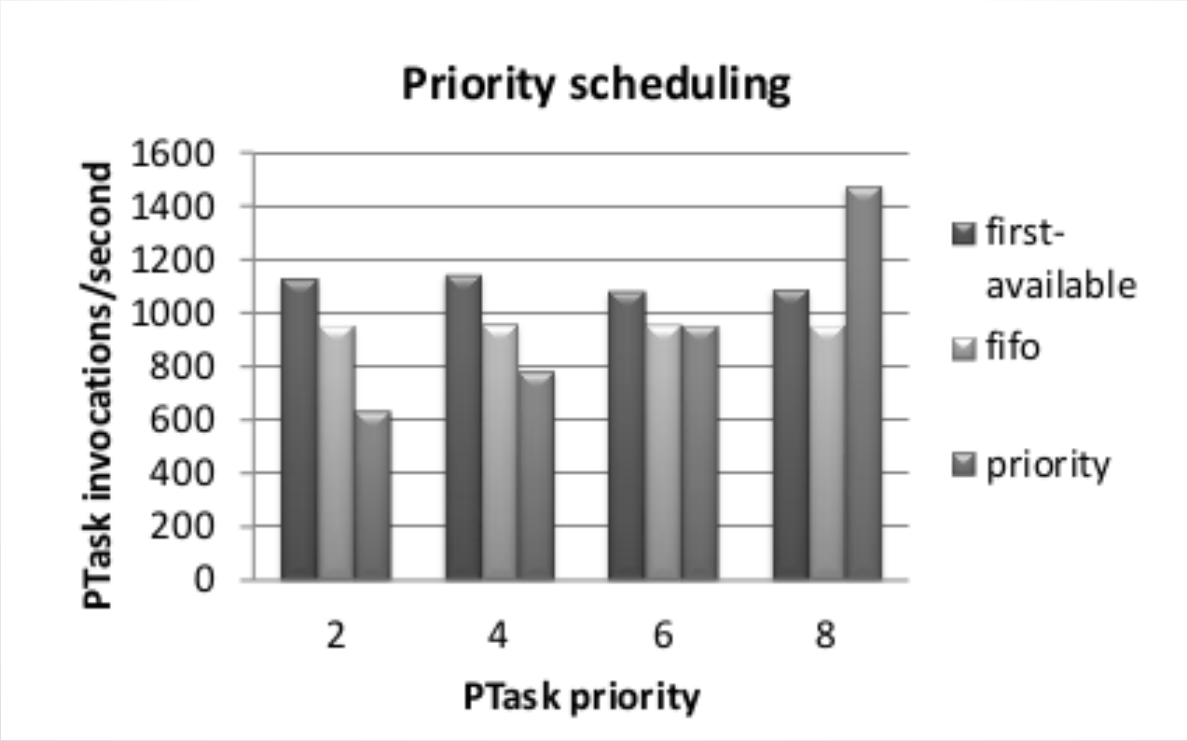
Evaluation 3 - PTask scheduling

- We add another GPU (of the same kind) and measure speedup over only one GPU with different policies:



Evaluation 3 - PTask scheduling

- To test the priority scheduling, we measure number of ptask invocations per second depending on the ptask priority:



Thank You!