Managing Accelerators: the Operating System Perspective

PTask: Operating System Abstractions to Manage GPUs as Compute Devices [SOSP, October 2011]

Chris Rossbach, Jon Currey – MSR,
Mark Silberstein – Technion,
Baishakhi Ray, Emmett Witchel – UT Austin

Pegasus (Coordinated Scheduling in Virtualized Accelerator-based Platforms) [Usenix ATC, June 2011]

Visakha Gupta, Karsten Schwan – Georgiatech,
Niraj Tolia – Maginatics,
Vanish Talwar, and Parthasarathy Ranganathan – HP Labs

Presentation
Kostas
Shantonu

Outline

- GPUs (and equivalent Accelerators): the software/hardware stack
- Why OS involvement is necessary
- Approaches
  - Programming model + OS abstractions (PTASK)
  - Virtualization (PEGASUS)
- Conclusions – Discussion

Accelerators as seen by ...

App Programmer

- Fit accelerate-able pattern
- Adapt algorithm to the programming model
- Optimize for specific accelerator architecture

System Programmer

- Maintain high-performance accelerator access
- Time-sharing (users/processes)
- Space-sharing (memory)
- Safe, controlled access

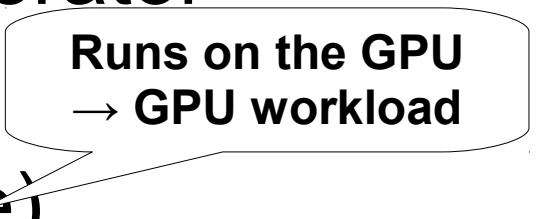
Focus of an **App** programmer

- Initialize the accelerator and its run-time
- Create **properly aligned** data **buffers**
- "Upload" data buffers to the accelerator
- Start the accelerator (e.g. GPUs: run the **optimized accelerator code**)
- "Download" result-buffers from accelerator
- Finalize the accelerator and its run-time

Runs on the GPU → GPU workload

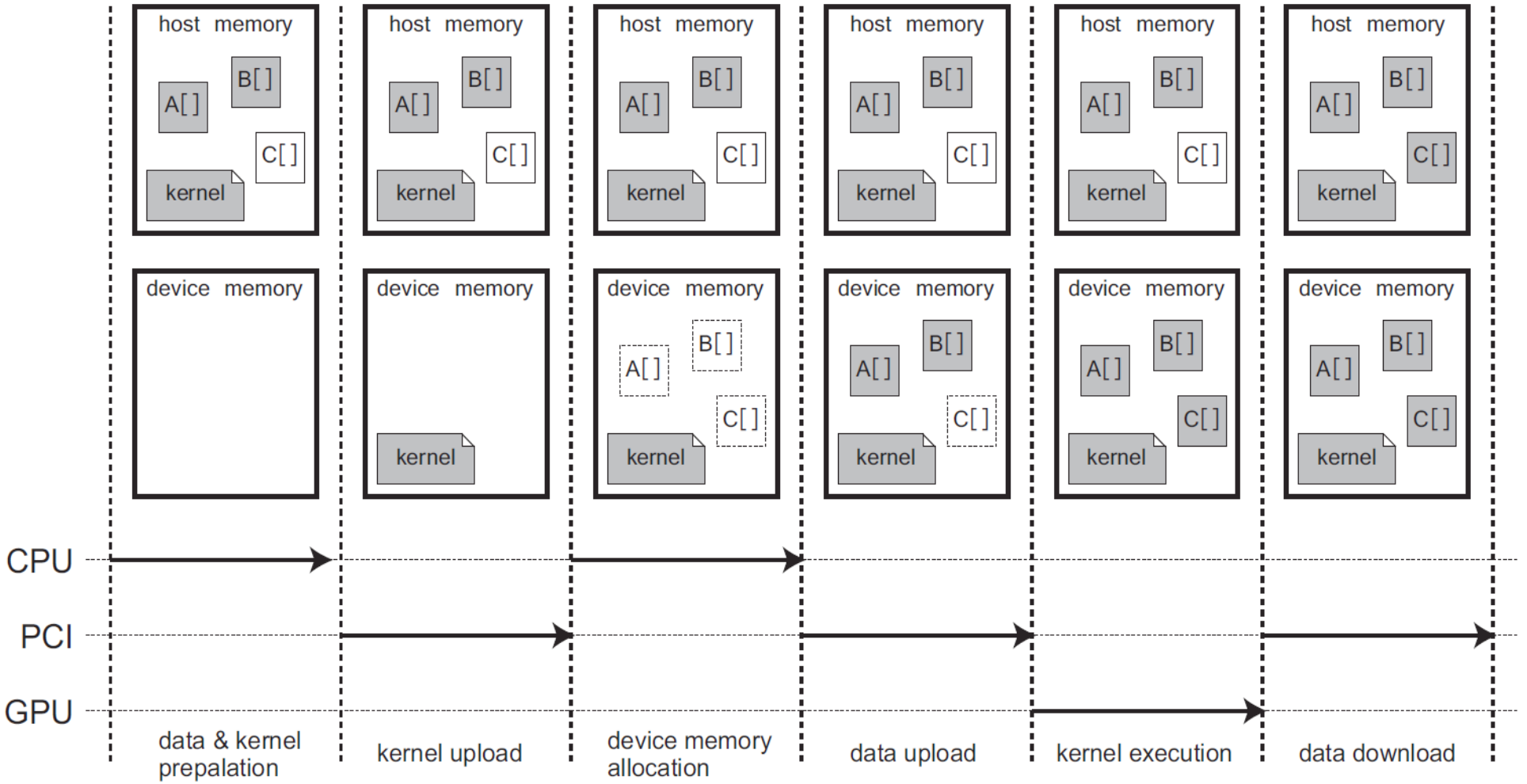
Focus of a **System** programmer

- **Initialize** the accelerator and its run-time
- Create properly aligned data buffers
- "**Upload**" data buffers to the accelerator
- **Start** the accelerator (e.g. GPUs: run the optimized accelerator code)



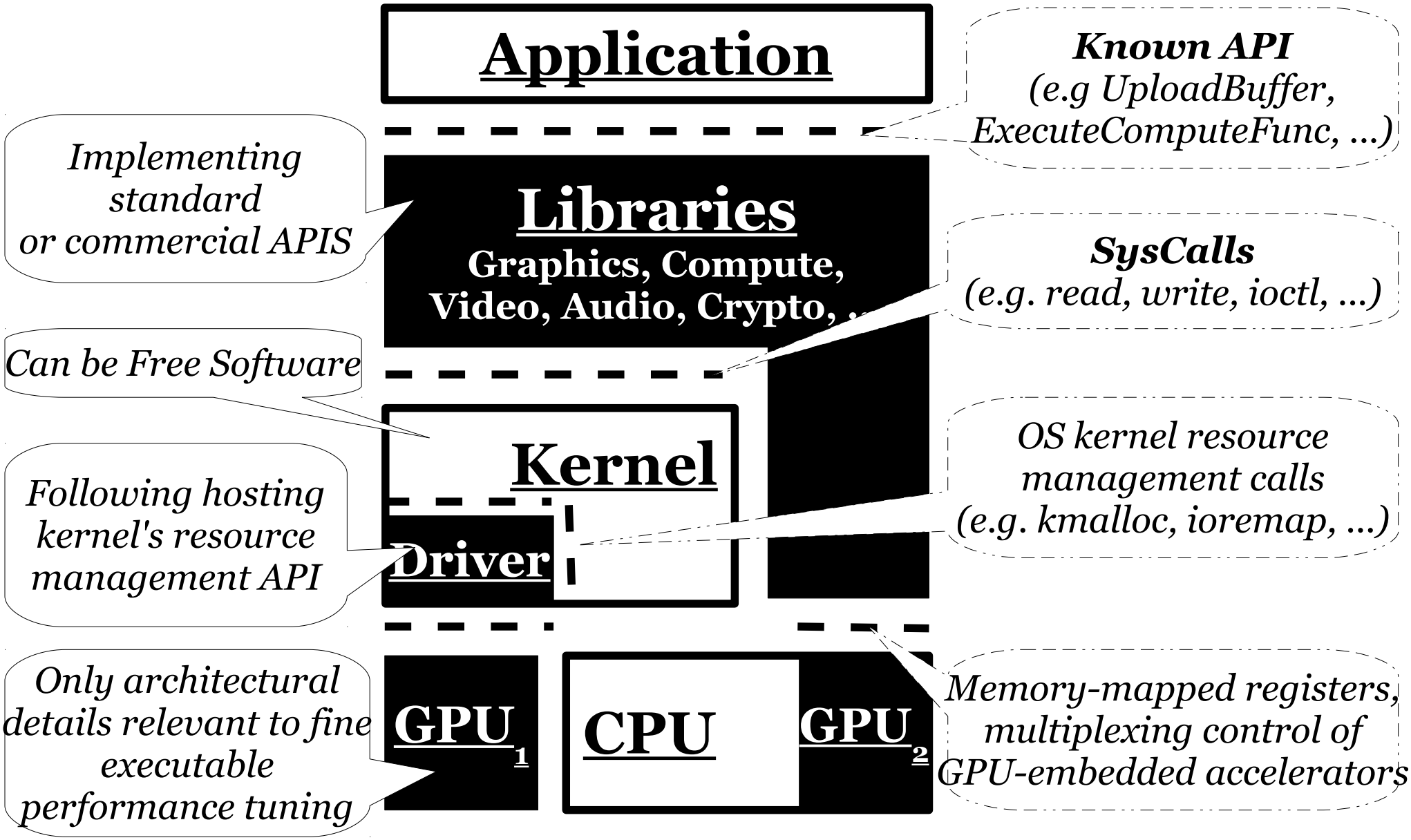
- "**Download**" result-buffers from accelerator
- **Finalize** the accelerator and its run-time

GPU execution flow example



Example of an execution flow of matrix multiplication $A[] \times B[] = C[]$

The GPU Software/Hardware Stack



Why involve the OS [1/2]

- Enhance system-wide objectives
  - Impact to whole-system performance, power/energy optimizations
- Provide scheduling guarantees
  - Fairness, isolation, safety
  - Meet real-time requirements
- Support accurate accounting
  - critical for cloud-level services

Why involve the OS [2/2]

- Extend the programming model's reach
  - support alternating between task (CPU) and data parallel (GPU) patterns
  - enable fast access to the GPU by other peripherals
  - use the GPU for kernel-level services (e.g. crypto, file-systems)
- Only the OS can realize run-time anomalies and enforce their safe management

Challenges [1/2]

- The GPU is not an I/O device but a Turing-complete co-processor
  - GPU workload not directly equivalent to CPU process
  - Unpredictable run-time of GPU workloads
  - The GPU doesn't run OS code
- Unlike CPUs, they can be non-preemptive
  - Hardware constraint
  - Only non-clean watchdog-based reset supported
- Disjoint address spaces likely
    - Automate (if possible) with data-movement
    - Maintain high-performance (optimize copying/migration)

Challenges [2/2]

- Limiting interfaces
  - GPU interface hidden behind “stack of black boxes”
  - Existing kernel/driver interface is mono-semantic (much described through a single ioctl call)
- Performance overheads might be unacceptable
  - E.g. games
- Vitalization is not straightforward
  - Hardware-specific software-stack hard to split to independent layers
  - MMIO adds another level of indirection

Goals

- Discuss different approaches to OS-involvement in GPU management
- Describe basic OS abstractions
- Suggest ways to promote GPUs to first-class scheduling entities
- Provide coherent motivation examples
- Demonstrate conclusive benefits over the “un-managed” case

PTask: Operating System Abstractions to Manage GPUs as Compute Devices

SOSP, October 2011

Chris Rossbach, Jon Currey – MSR,
Mark Silberstein – Technion,
Baishakhi Ray, Emmett Witchel – UT Austin

Ptask Outline

- Motivation:
  - make the GPU accessible to highly interactive applications
- Approach:
  - Focus on unnecessary data duplication (data-movement) and sub-optimal migration
  - Extend the existing programming model through a graph-based dependency representation
  - Extend the kernel interface (syscalls) to automate and optimize data placement and GPU workload scheduling
- Key contribution:
  - A semantically clear suggestion about how the OS can realize a GPU workload and adapt its scheduling policies

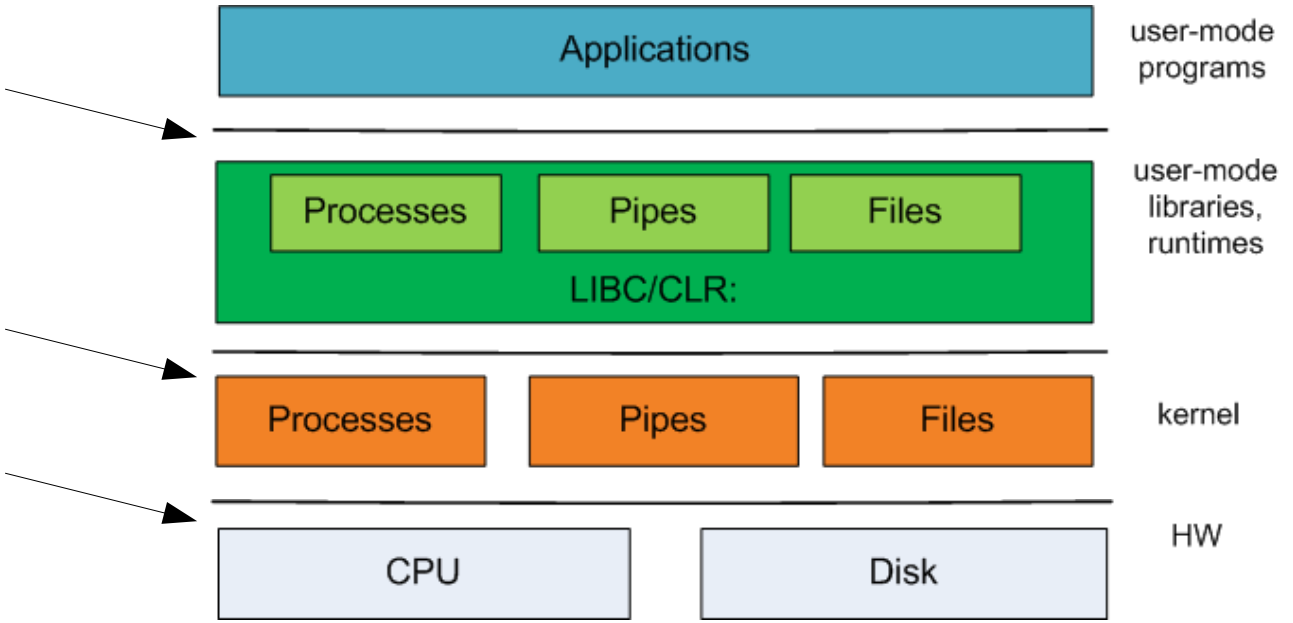
Motivation

- GPUs are everywhere
  - Top supercomputers
  - Workstations, PCs, smart-phones tablets
- Expanded, yet limited application domain
  - Games, Simulation
  - Highly data parallel applications (science)
- Current GPUs have challenging system limitations
  - GPU+main memory disjoint
  - Treated as I/O device by OS

Approach

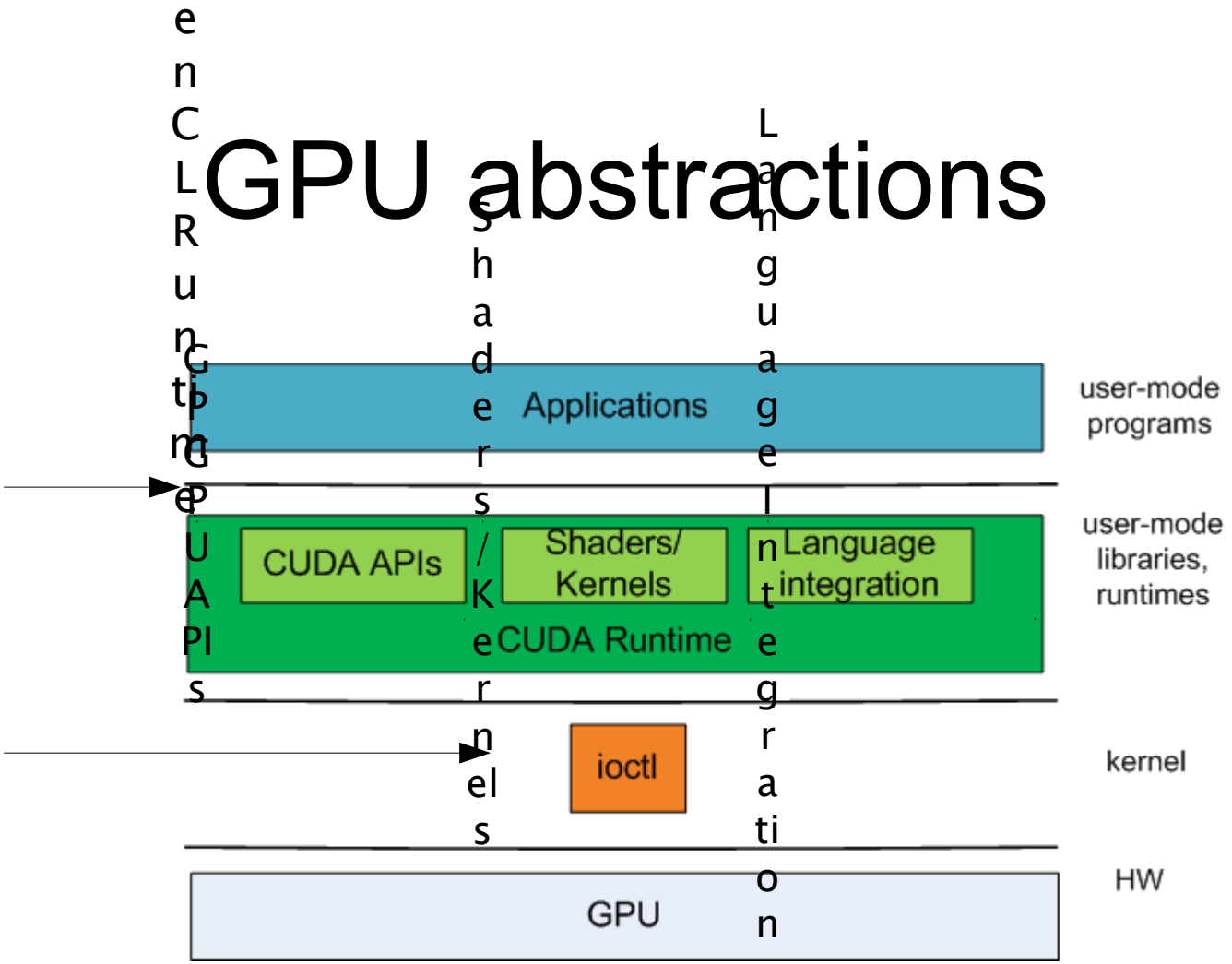
- Use OS abstractions (processes, files, …)
- Build modular tools that allow reuse, flexible combinations
- Hide GPU limitations (e.g. incoherent memory) without hindering performance
- Rely on a semantically clear description of GPU workloads from the OS perspective to make GPUs first-class scheduling devices

OS abstractions



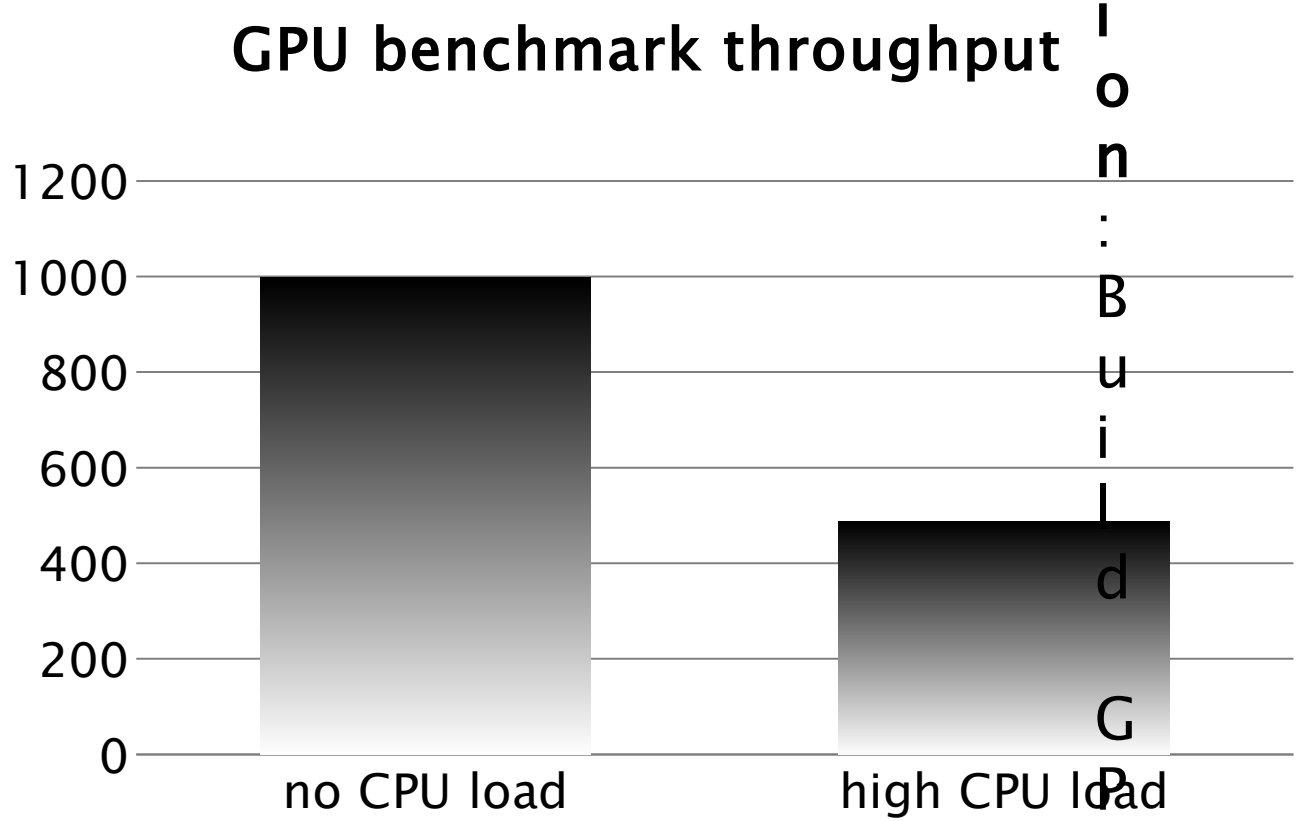
Rich API for system interactions (system-calls)
Kernel-level management of resources (CPU, RAM, Disk, ...)
Composable components ($ cat file | grep pattern)

GPU abstractions



No kernel-facing API
OS limited to safe device access support
Poor composability

CPU-bound processes hurt GPUs



can submit workloads to
solution: build GPU-aware

GPU-bound processes hurt CPUs

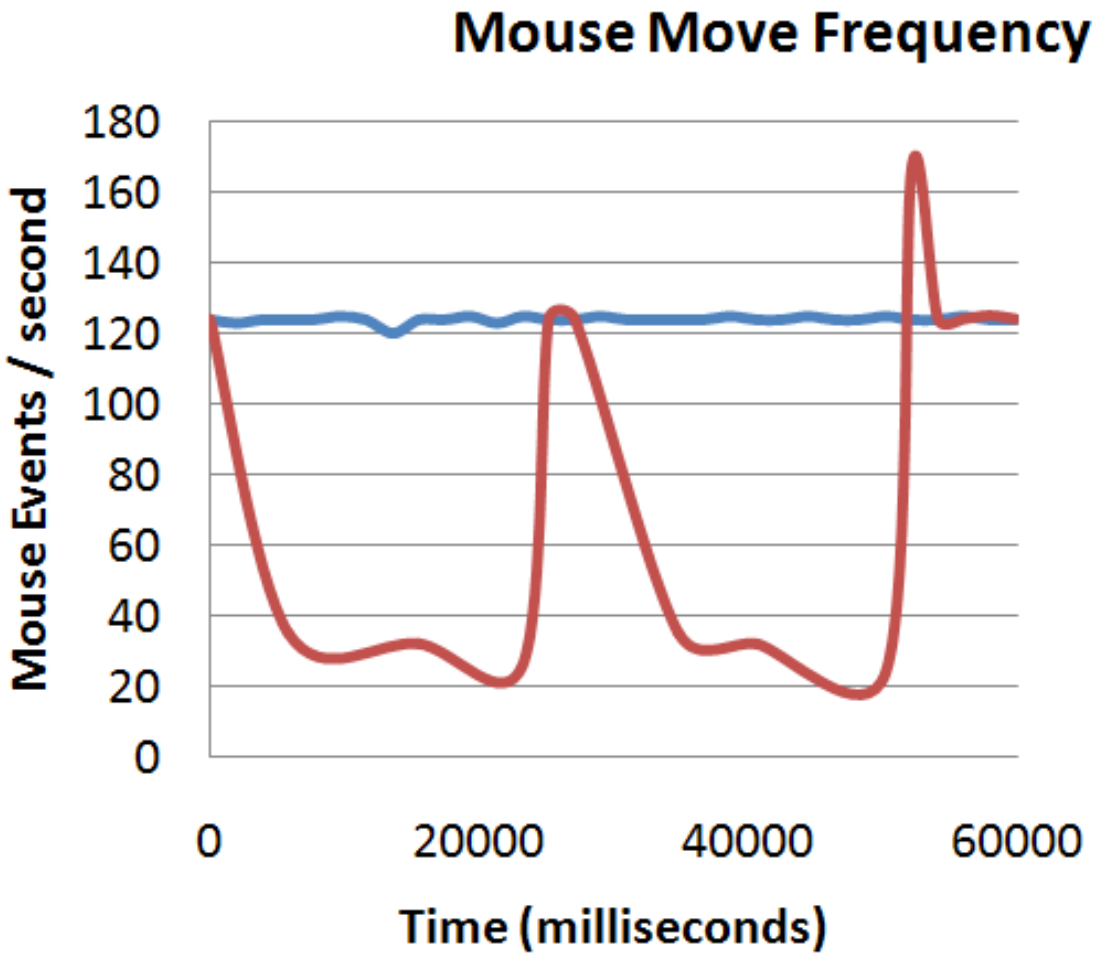
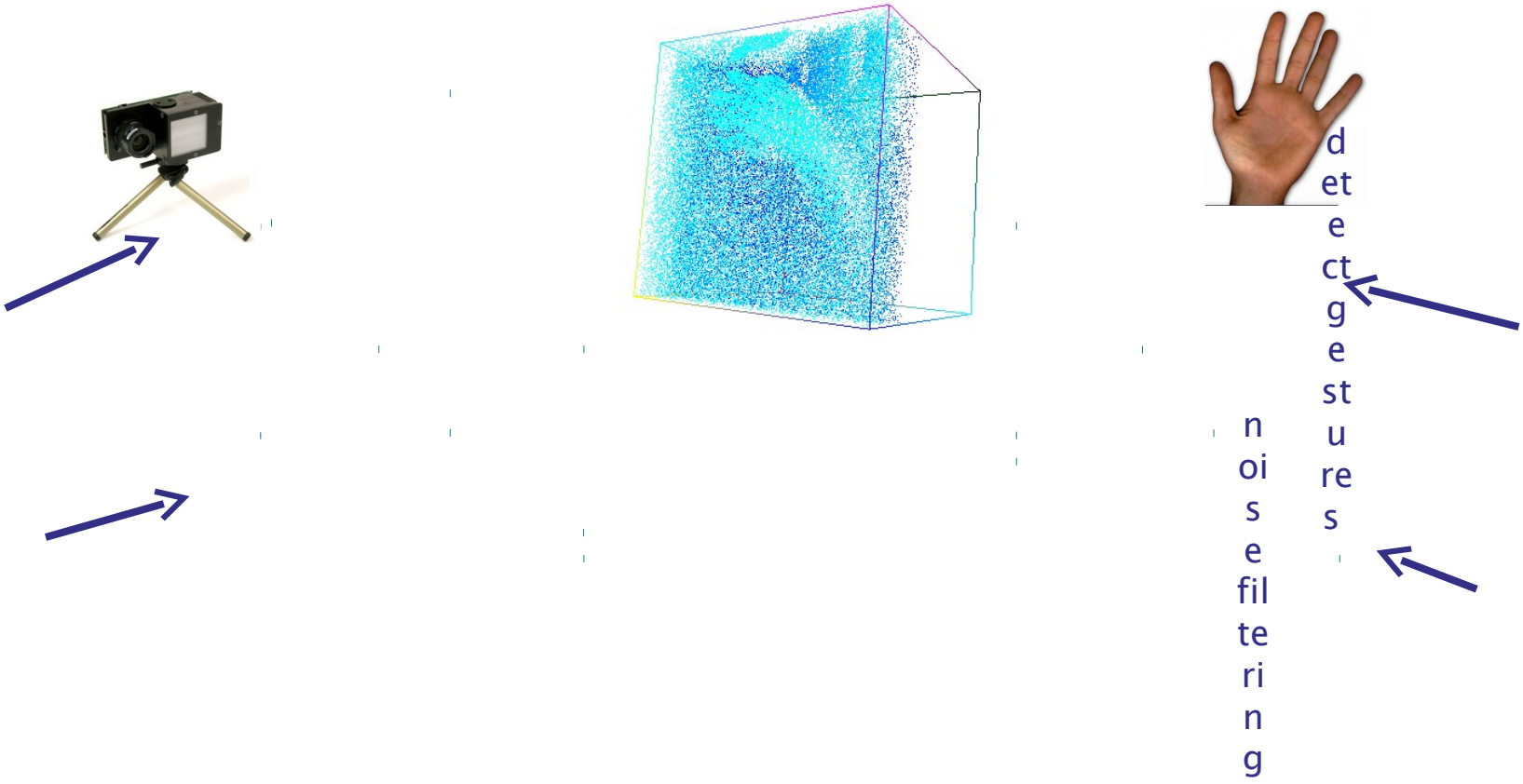


Image-convolution in CUDA

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

preemptible; not a solution: Build GPU-awar

Composition: Gestural Interface



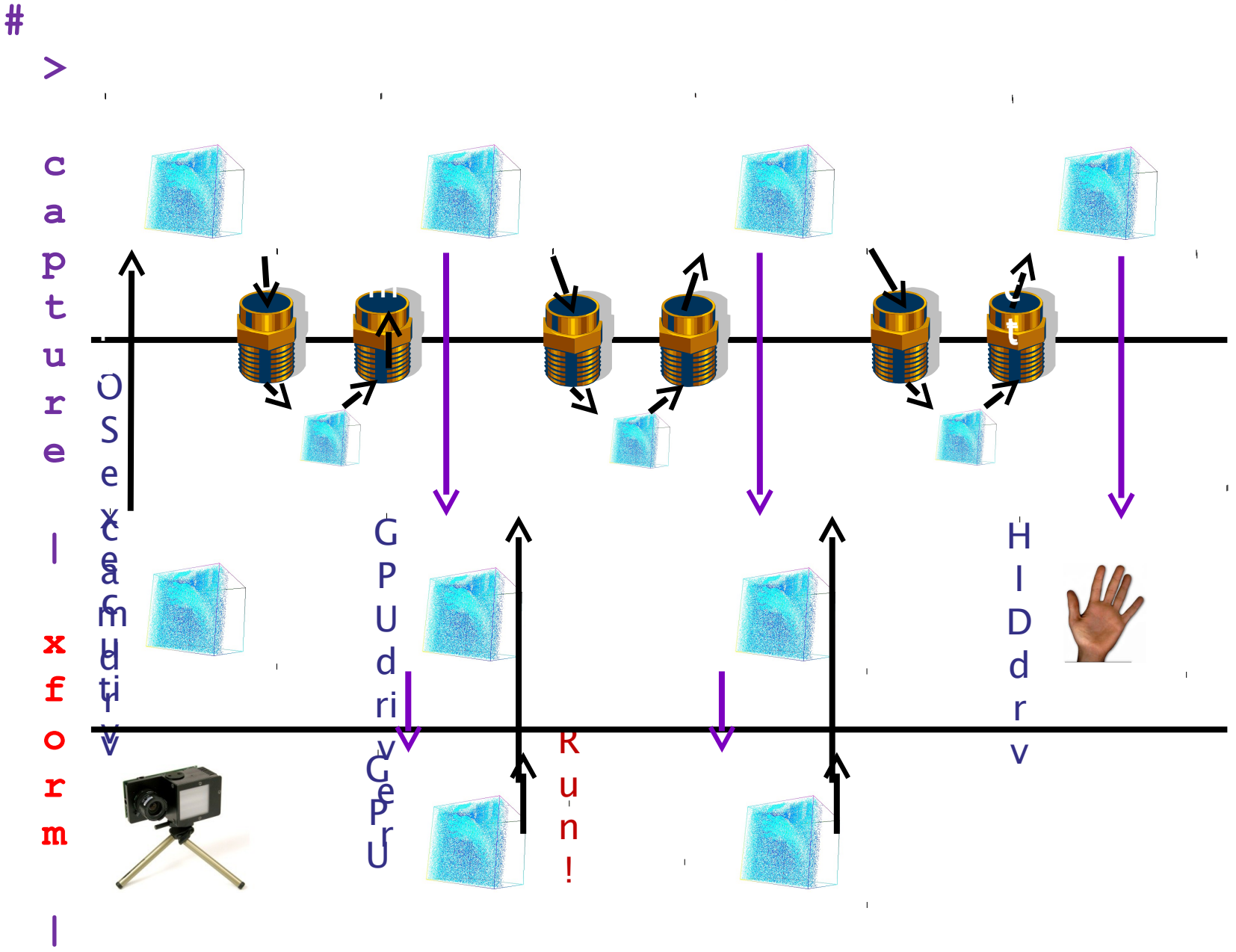
- peripheral device direct input (e.g. camera
  → data-parallel (GPU-fit) components
  → non-GPU-fit component (e.g. human interface)

- catusb | xform | filter | hdinput & # CPU,GPU *composition*

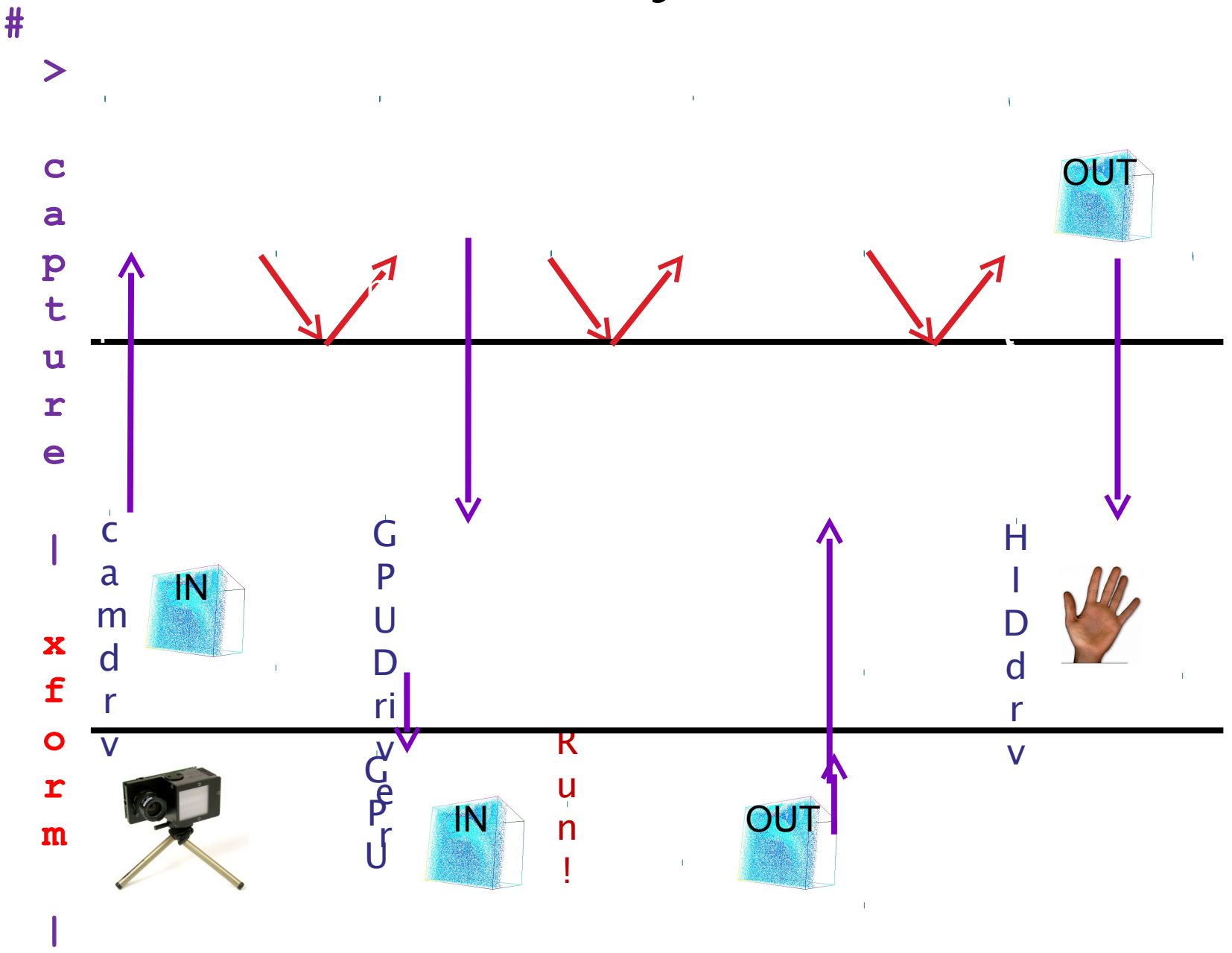
Existing limitations

- The programmer must control where (which device) to allocate a buffer as they code their program
  - Double-buffering optimization can hurt
  - Streaming buffers across processes impossible
- The programmer must control when to invoke a kernel, while the system asynchronously satisfies this request (command batching)
- Host (CPU) orchestrates data transfers and access to the device
- Data-replication will happen even on integrated CPU/GPU systems (coherent memory)

Conceptually ...



Ideally ...

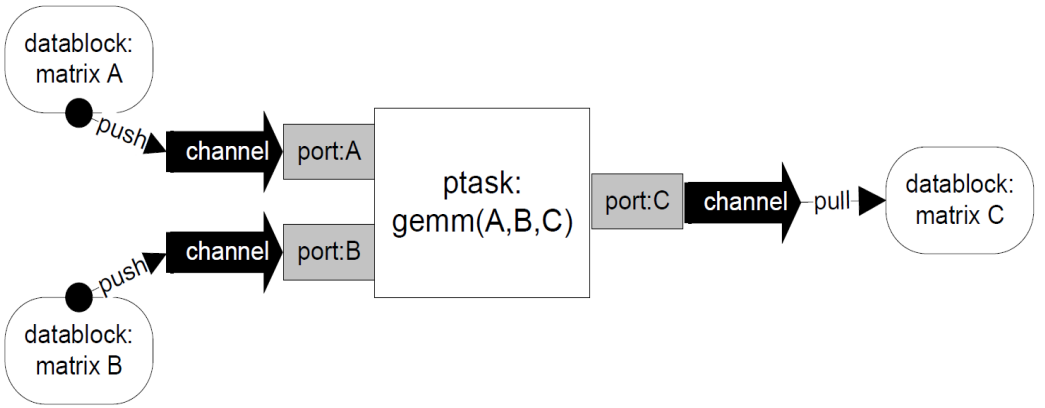


Data-flow-based GPU abstractions

- **ptask** (parallel task)
  - Analogous to a process for GPU workload
  - List of input/output resources (*e.g. stdin, stdout…*)
- **ports**
  - Type: Input, Sticky, Output
  - State: Occupied, Unoccupied
  - Bound to GPU-side or consumed by run-time
- **channels**
  - Similar to pipes, connect arbitrary ports
  - Type: Input (1 channel), Output (>=1 channel)
  - Capacity: fifo queue of data-blocks
- **data-blocks**
  - Memory-space transparent buffers
- **templates**
  - Abstract buffer dimension and access pattern descriptor
- **graph**
  - DAG: connected ptasks, ports, channels

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

PTask API system calls

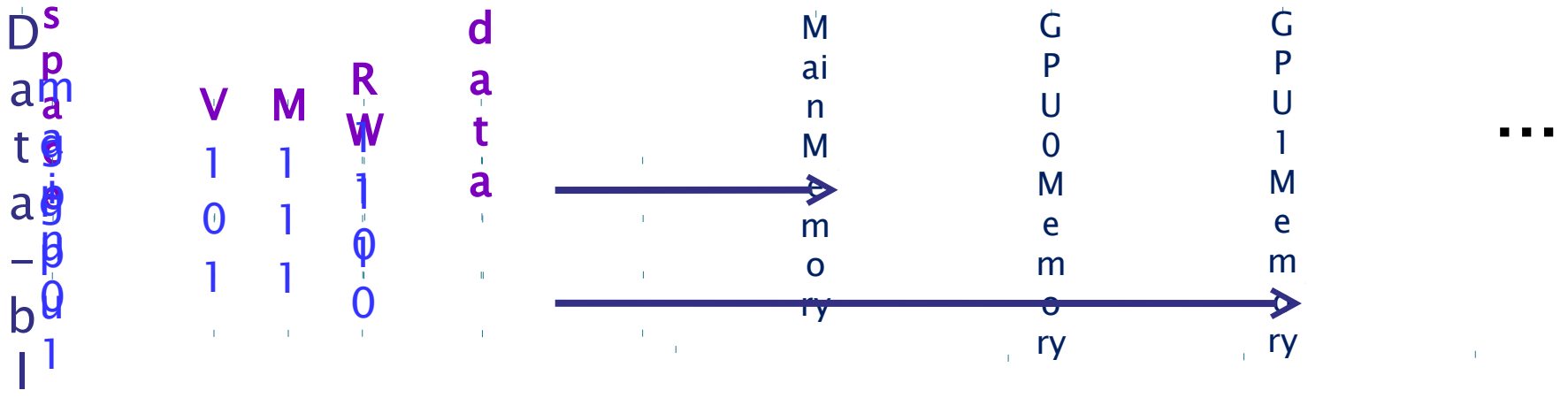


A dataflow graph for matrix multiplication.

Important system parameters

- Assign host (CPU) helper threads from a pool to ptasks – used for data movement, GPU dispatch
- *Buffer-map:* Pro-actively or lazily copy/migrate data to fit locality requirements (e.g. GPU expecting input in dedicated RAM)
- Templates:
  - Buffer dimensions + stride → iteration space
  - Buffer type (fixed size, variable size, opaque buffer)
  - Run-time binding description (CPU, GPU)
  - Passed through meta-data channels for irregular data handling

Data-blocks



- Logical buffer
  - backed by multiple physical buffers
  - buffers created/updated lazily
  - mem-mapping used to share across process boundaries
- Track buffer validity per memory space
  - writes invalidate other views
  - Use reference counters
- Flags for access control/data placement

Run-time

- State:
  - Waiting (for input; not all input ports occupied)
  - Queued (inputs available, read data-blocks occupying input/sticky ports, waiting for GPU)
  - Executing (running on the GPU)
  - Completed (finished execution, waiting for output ports to transition to occupied → waiting)
- Run-time scheduling of queued tasks
  - Respect and optimize data locality

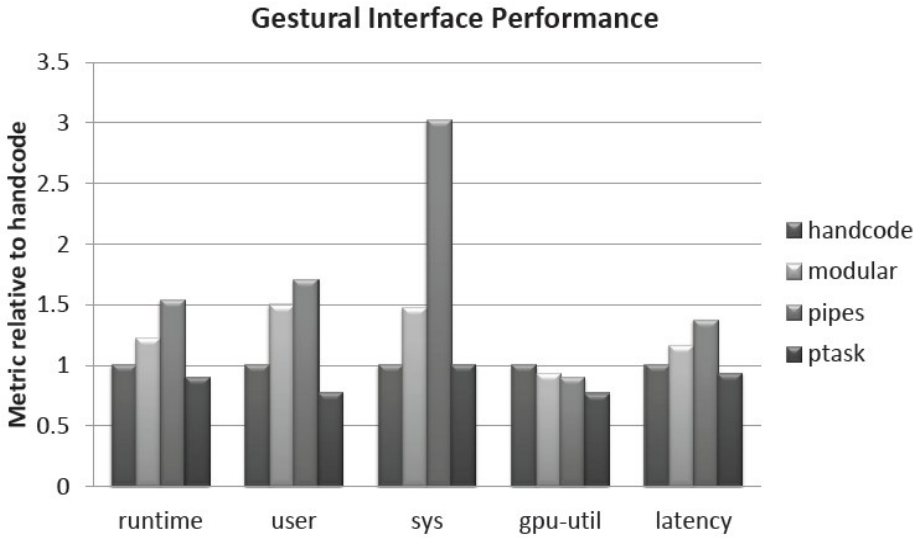
PTask Scheduling

- First-available: manager threads compete for lock-protected accelerator(s)
- Fifo: first-available with queuing
- Priority: enhanced ptasks with priorities + device selection
  - Static (programmer-requested) + Proxy (assigned to managed-thread to avoid *priority laundering*)
  - Priority boosting based on current & average wait-time, average run-time
  - Effective priority: f(static, proxy, boosting)
  - Device selection: GPU strength, task fitness (simple heuristics)
- Data-aware: Priority + Prefer GPUs whose memory spaces contain most up-to-date inputs
  - Not work-conserving but can save costly data-migrations

Implementation

- GPU programmable through user-level libraries
  → user-level component necessary for demonstrated use
  → (in practice) subject to user-attacks
- Windows:
  - Ptask API transparently described through ioctl extensions and user-mode drivers (Windows-specific driver-development tools)
- Linux:
  - Blocking sys-calls to pass information to the kernel; instrument applications
  - Event-based (invoke, complete) tocken-bucket based algorithm enhancing CPU task scheduler
  - Priority-adjusted, time-replenished budget

Evaluation 1/3



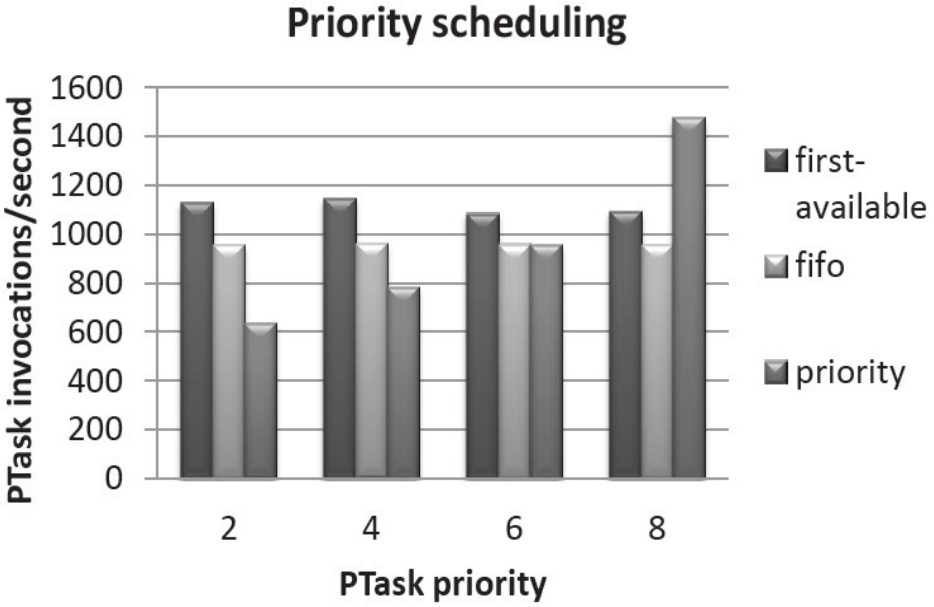
- Compare hand-coded, pipes-based, non-pipe modular and ptask for latency (real-time) throughput (unconstrained)
- Ptask optimizes both metrics while being as or almost as efficient a hand-optimized code

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 2/3

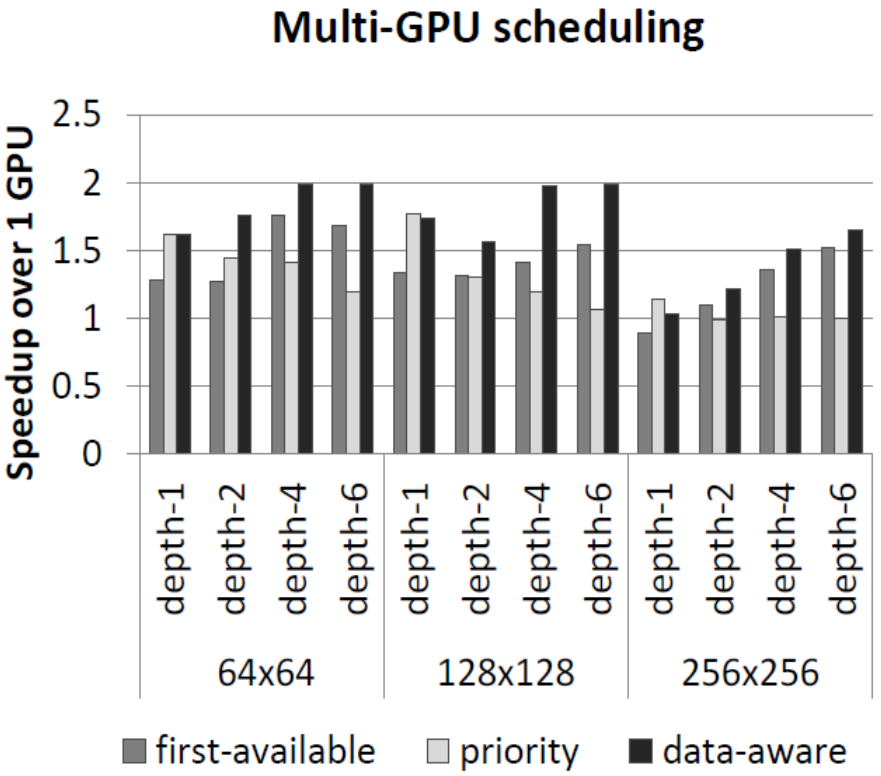
- Linux FUSE
  - AES EncFS
  - GPU-accelerated reads: 17% faster
  - GPU-accelerated writes: 28% faster
  - Priority inversion possible without Ptask
  - Nice-levels carrying to GPU tasks with Ptask
- Micro-benchmarks
  - Variation of access patterns and input sizes in ptask graphs
  - Ptask outperforms hand-optimized data/compute overlapping and composable alternative
  - Reason: data copying

Evaluation 3/3



4 graphs, 36 ptasks each, 1 GPU

- Priority throughput increases linearly with the priority of the graph



Range of graph sizes, matrix dimensions

- Data-aware policy is scalable
- Oblivious ptask assignment (e.g. priority) hurts performance

Summary

- Operating-system inspired accelerator programming model based on data-flows
- Performance stems from avoiding unnecessary data replication across address spaces (devices, process)
- GPU abstractions equivalent to familiar OS abstractions
- Abstractions built from the application programmer's perspective – a Ptask is a compute/graphics/...-kernel instance
- Proves scheduling is better than un-managed GPUs

Pegasus (Coordinated Scheduling in Virtualized Accelerator-based Platforms)

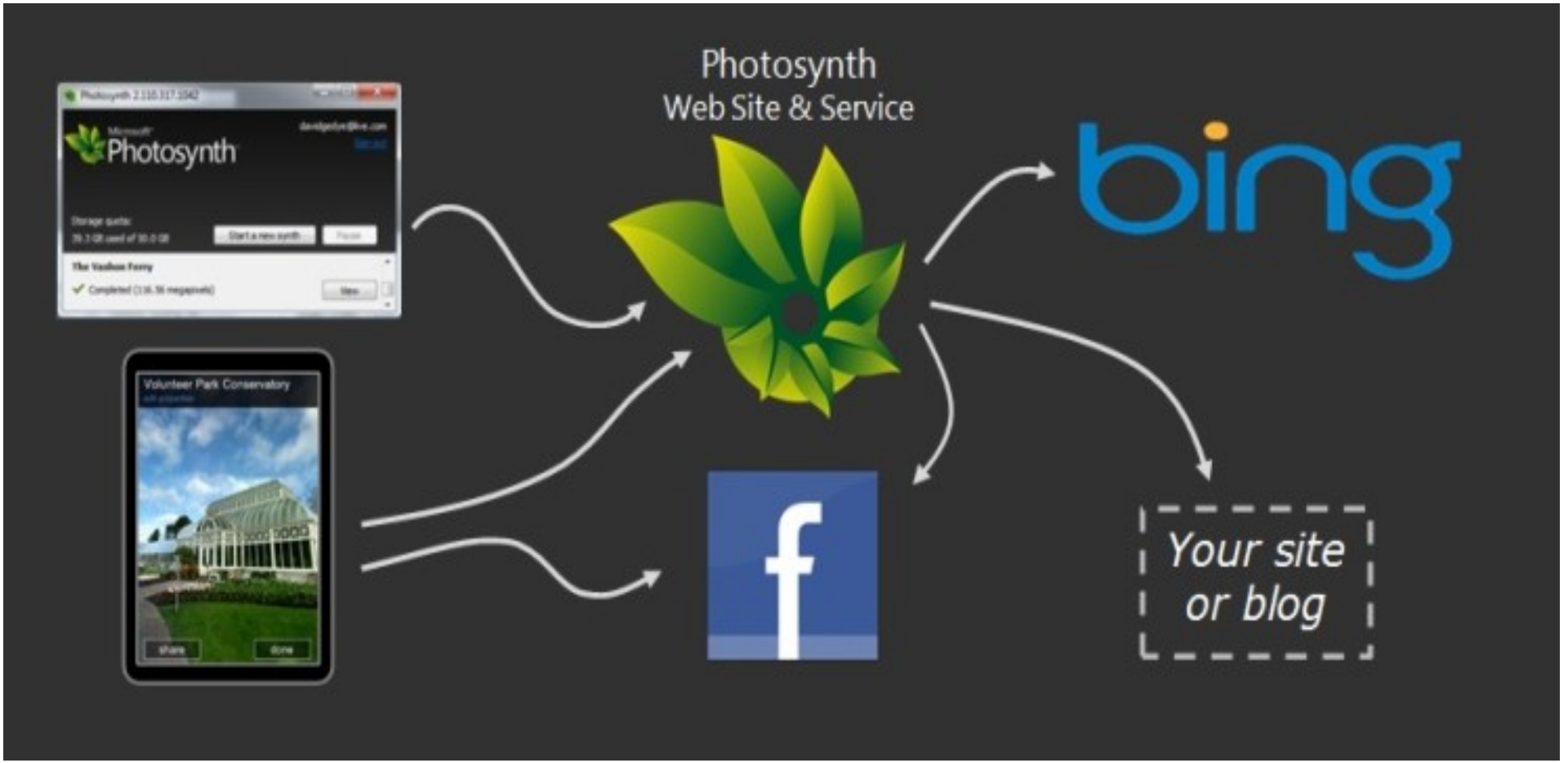
Usenix ATC, June 2011

Visakha Gupta, Karsten Schwan – Georgiatech,
Niraj Tolia – Maginatics,
Vanish Talwar, and Parthasarathy Ranganathan – HP Labs

Re-thinking Accelerator-based Systems

- Accelerators as first class citizens
  - Why treat such powerful processing resources as devices?
  - How can such heterogeneous resources be managed especially with evolving programming models, evolving hardware and proprietary software?
- Sharing of accelerators
  - Are there efficient methods to utilize a heterogeneous pool of resources?
  - Can applications share accelerators without a big hit in efficiency?
- Coordination across different processor types
  - How do you deal with multiple scheduling domains?
  - Does coordination obtain any performance gains?

Why Coordinated Scheduling is Needed?

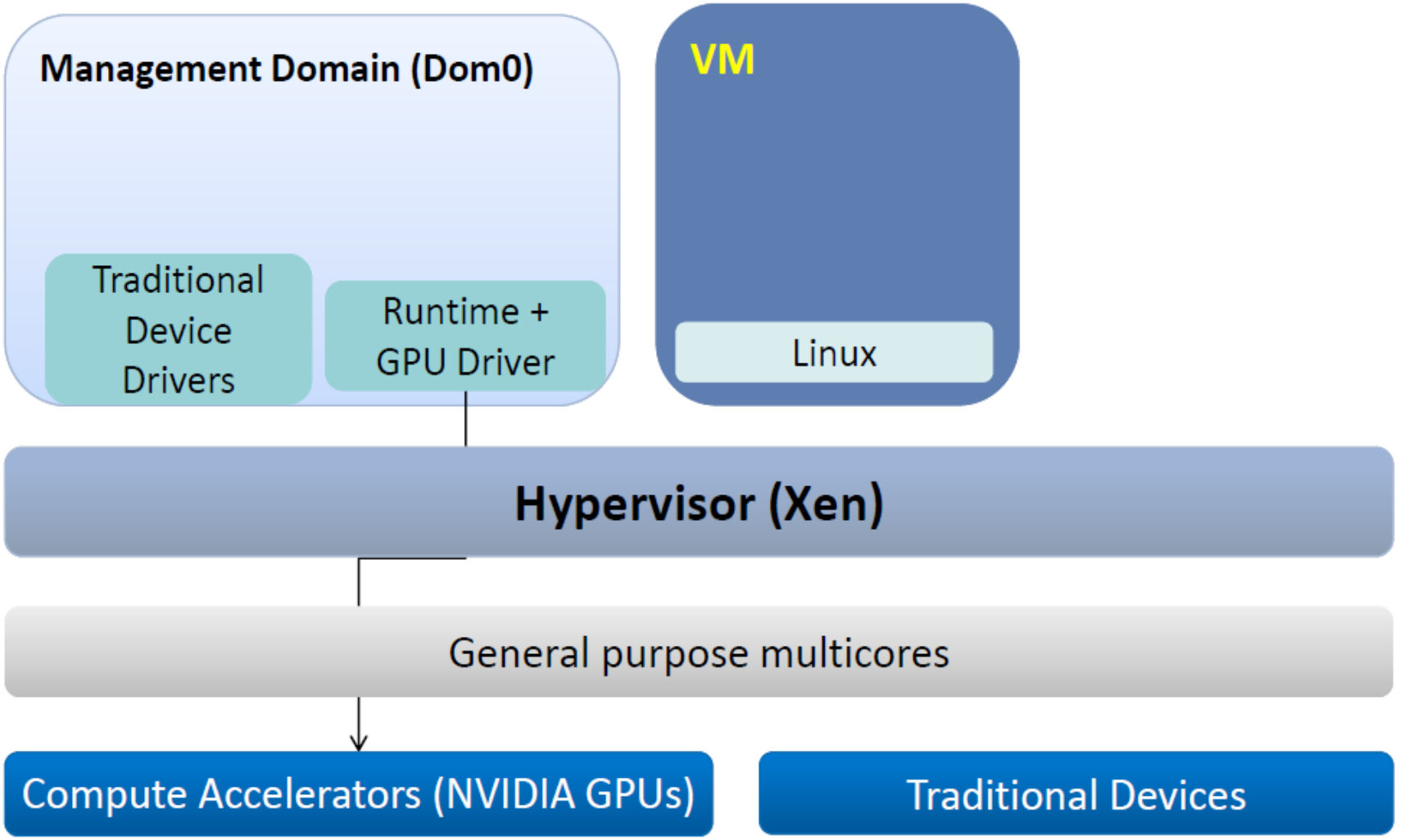


Objectives of Pegasus

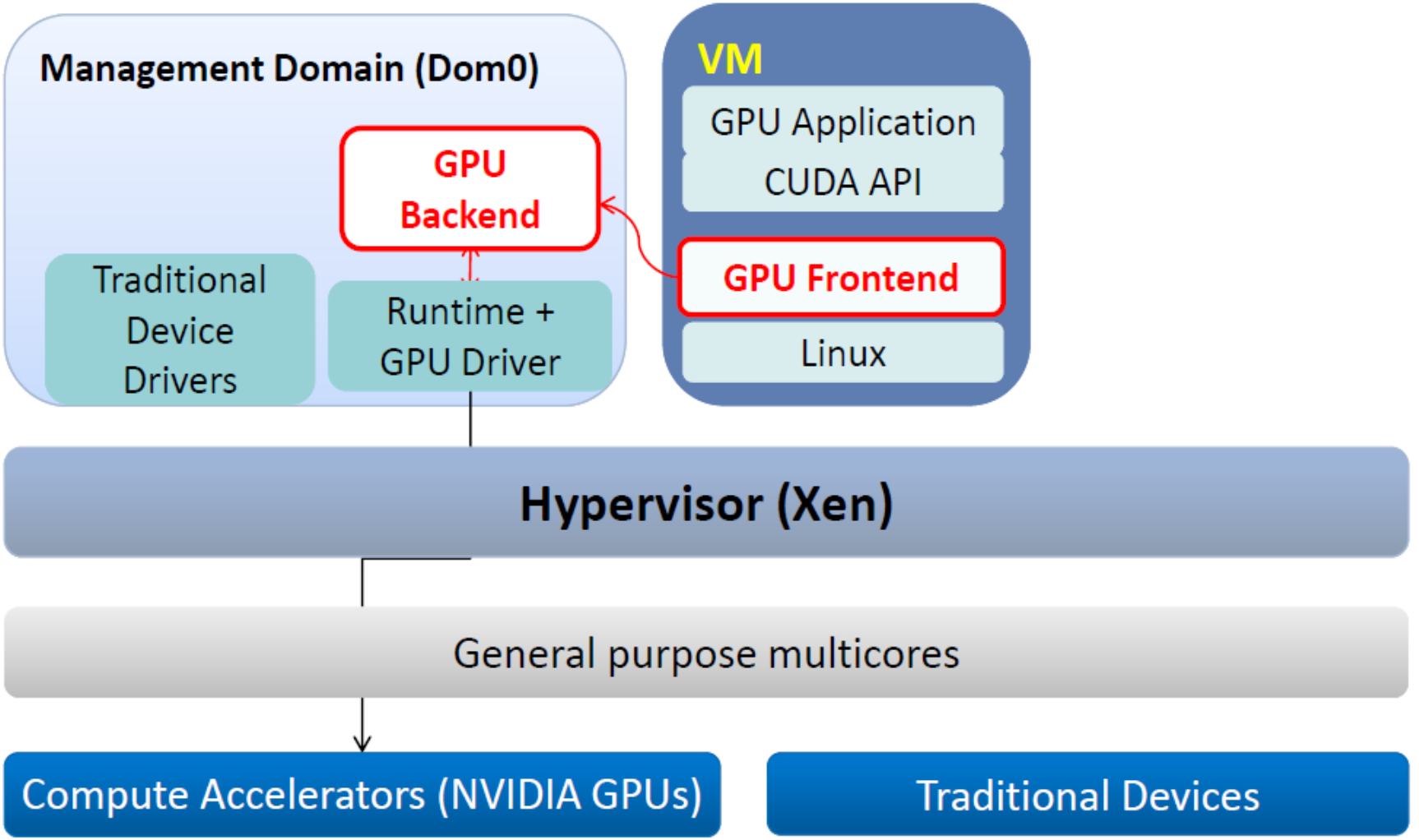
- Treats accelerators as first class schedulable entities
  - Supports scheduling methods that efficiently uses both general purpose and accelerator cores
  - shares accelerators fairly and efficiently across multiple virtual machines (VM)
- Provides system support for efficient accelerator management via virtualization
  - Operates at a hypervisor level (VMM)
  - Uses a uniform resource model for all cores on heterogeneous CMP

The Virtualization Approach

Extending Xen for Closed NVIDIA GPUs

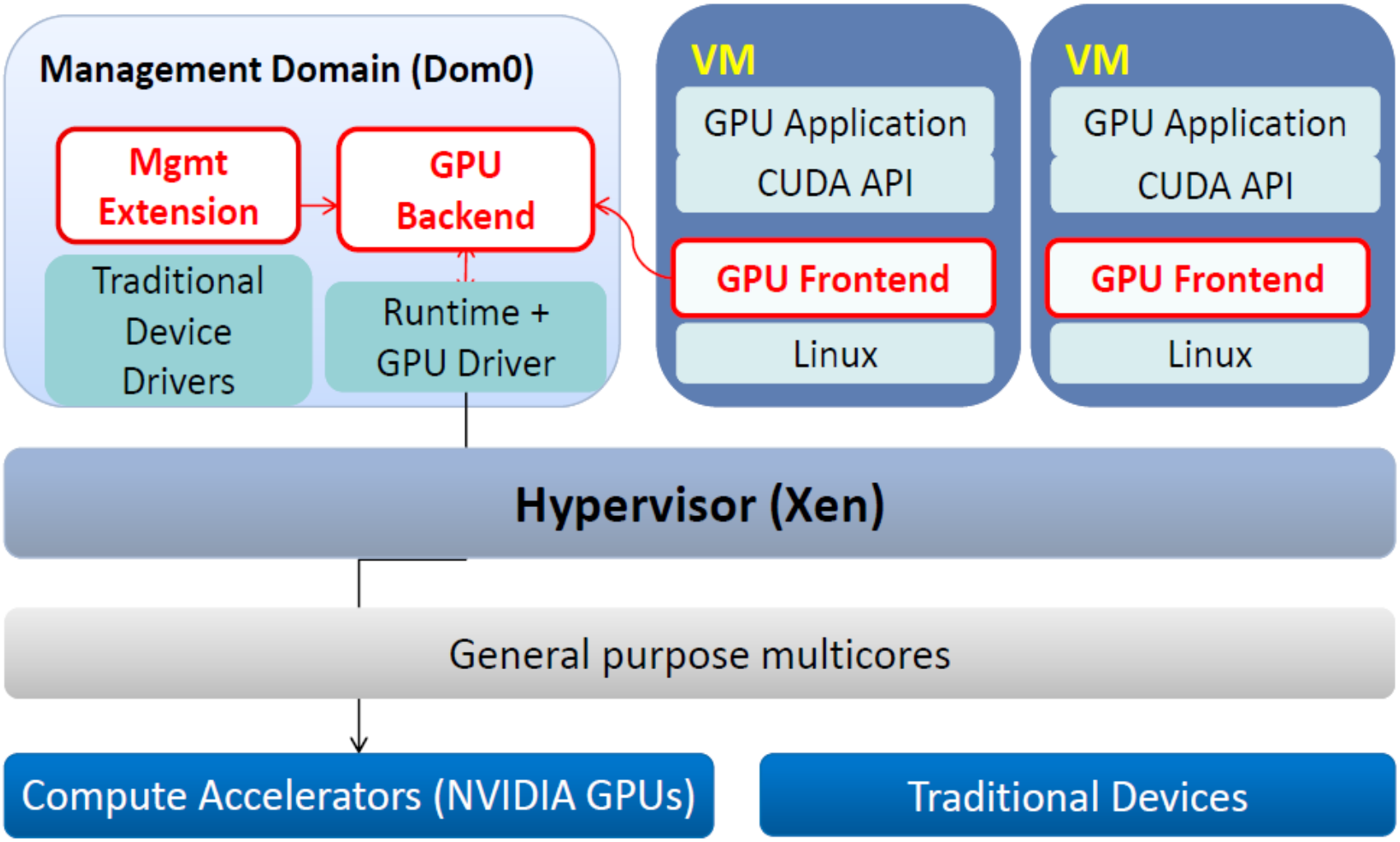


Extending Xen for Closed NVIDIA GPUs



NVIDIA's CUDA – Compute Unified Device Architecture for managing GPUs

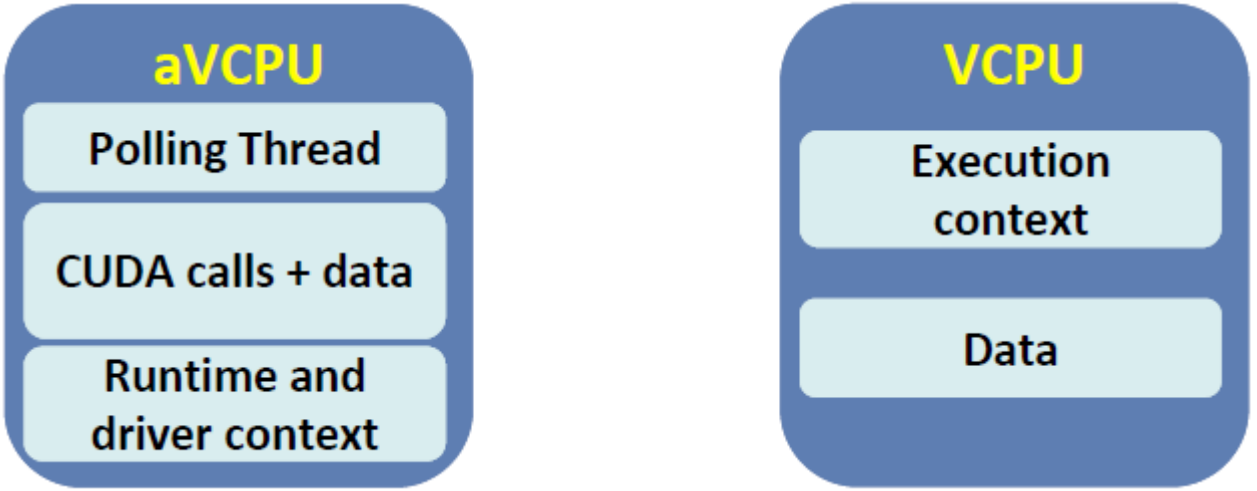
Extending Xen for Closed NVIDIA GPUs



NVIDIA's CUDA – Compute Unified Device Architecture for managing GPUs

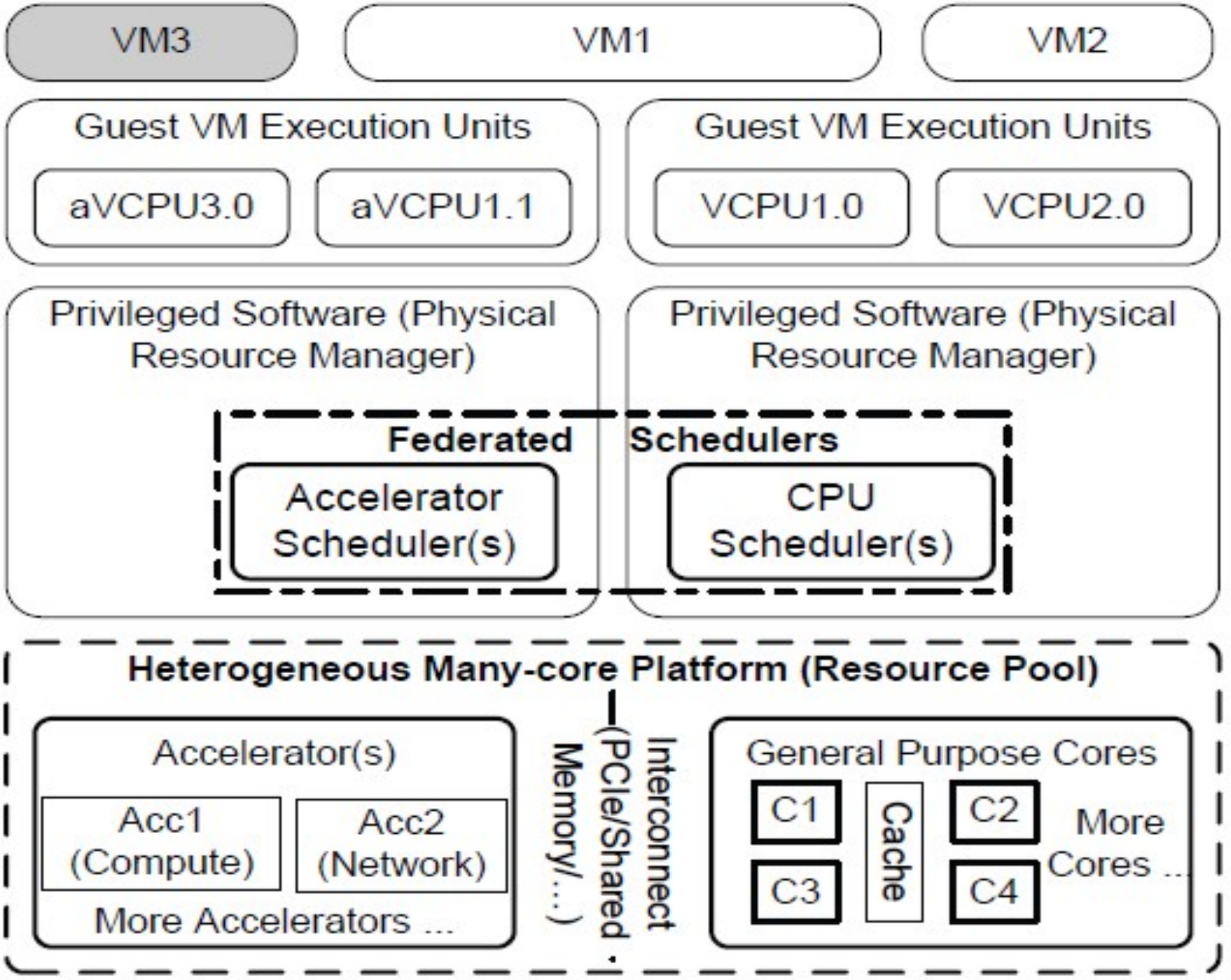
Accelerators as First-Class Schedulable Entities

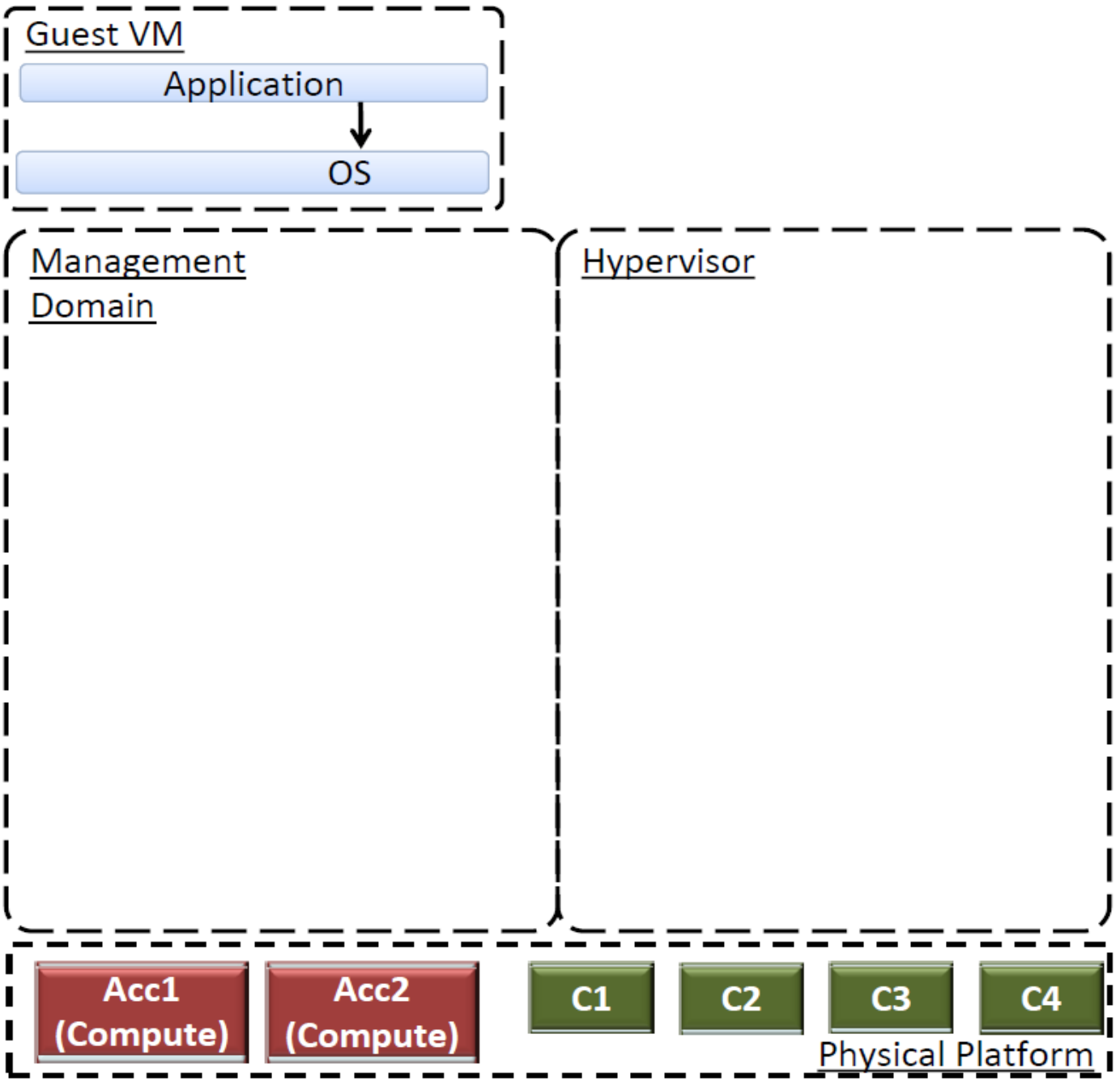
New Abstraction for Accelerator



- VCPU (virtual CPU) – first class schedulable entity on physical CPU
- aVCPU (accelerator virtual CPU) - first class schedulable entity on GPU
  - has execution context on both, CPU (polling thread, runtime, driver context) and GPU (CUDA kernel)

Pegasus Architecture

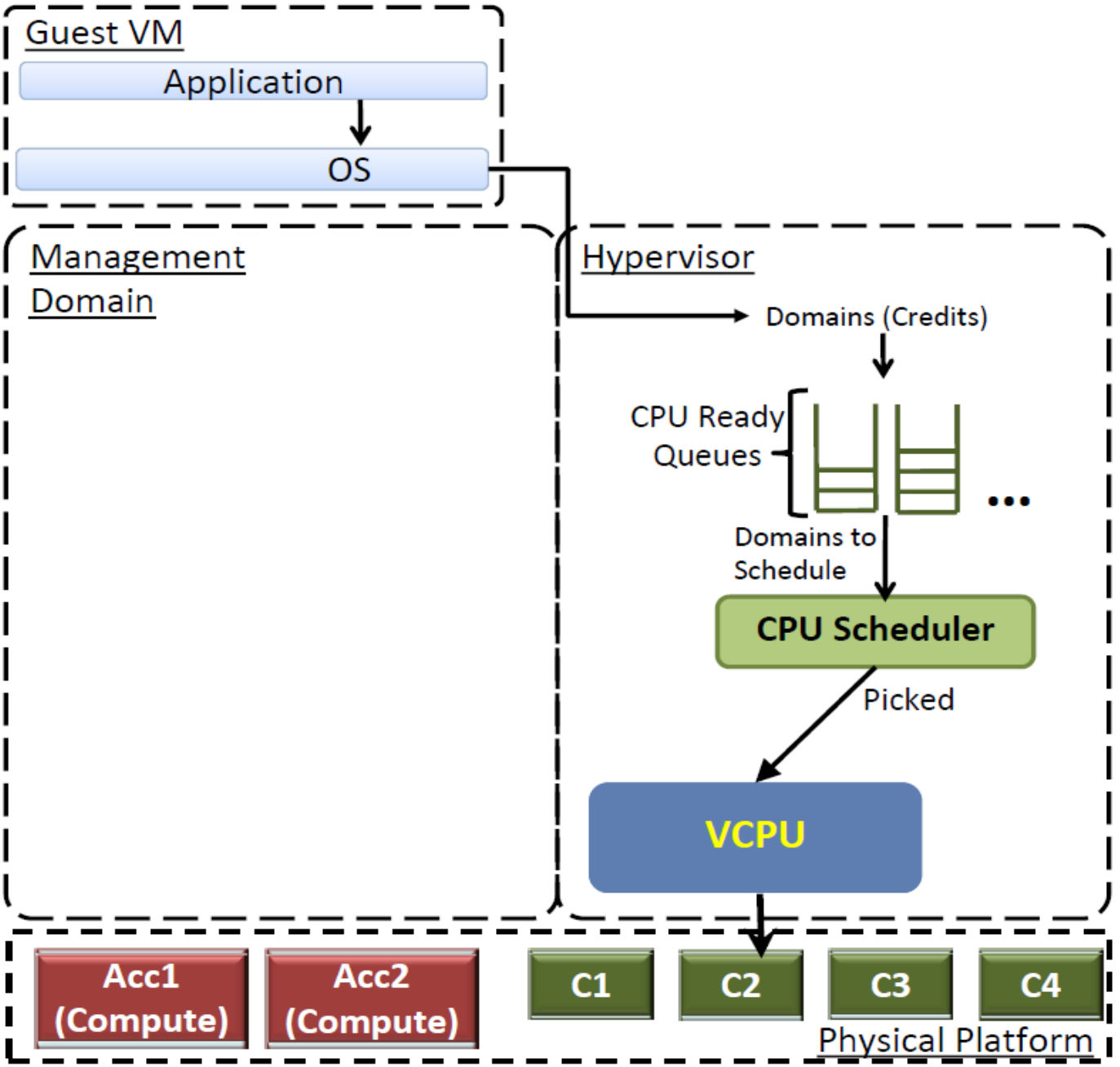




Logical View of the Pegasus Resource Management Framework

Georgia Tech College of Computing

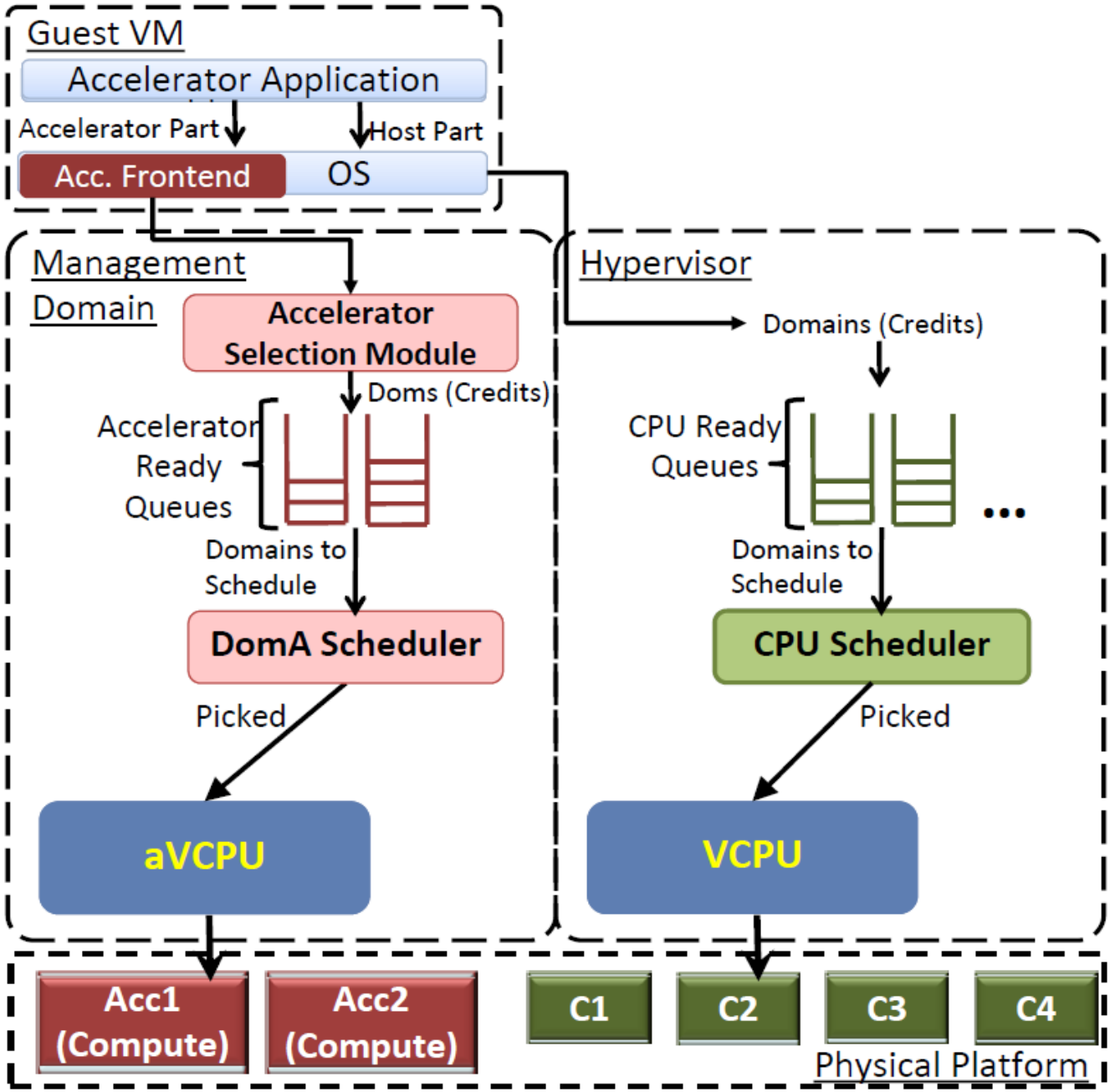
LABS hp



Logical View of the Pegasus Resource Management Framework

Georgia Tech College of Computing

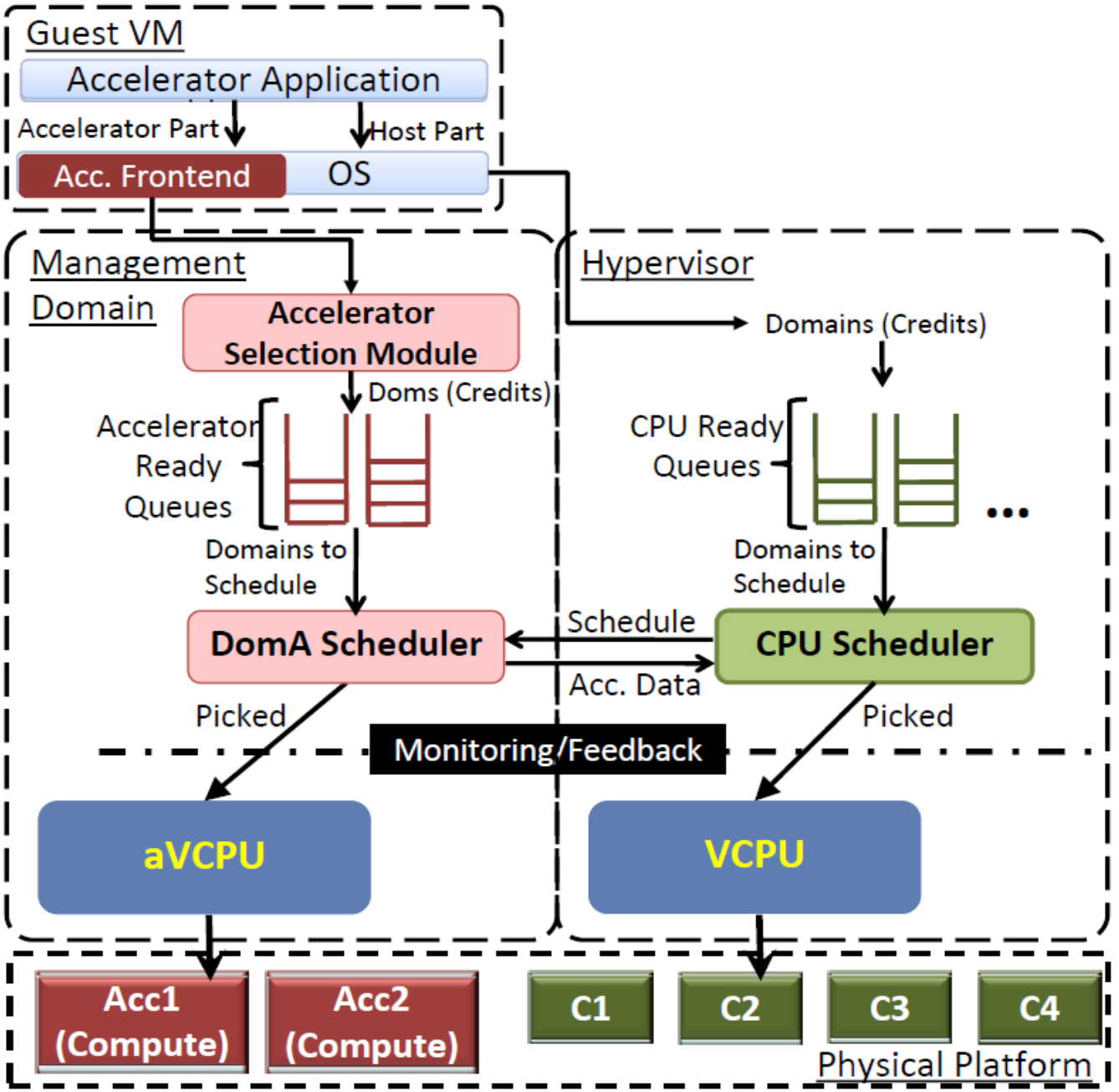
LABS hp



Logical View of the Pegasus Resource Management Framework

Georgia Tech College of Computing

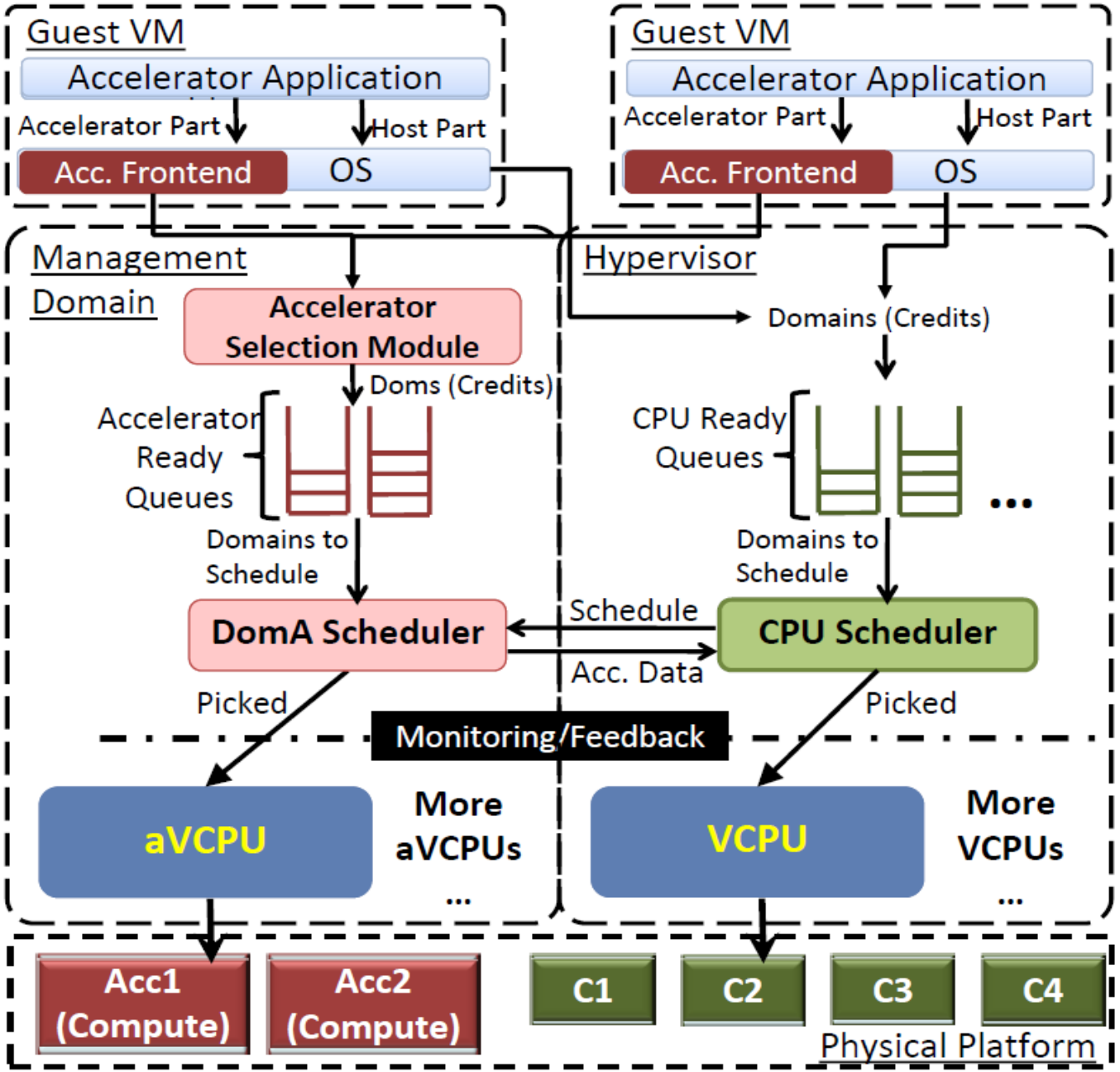
LABS hp



Logical View of the Pegasus Resource Management Framework

Georgia Tech College of Computing

LABS hp



Logical View of the Pegasus Resource Management Framework

Georgia Tech College of Computing

LABS hp

Pegasus Scheduling Policies

No Co-ordination

- **Round Robin (RR)** – relies on the GPU runtime/driver layer to handle all requests
- **AccCredit (AccC)** – Instead of using the assigned VCPU credits for scheduling aVCPUs, new accelerator credits are defined based on static profiling
  - Proportional fair-share

Hypervisor Controlled Policy

- **Strict Co-Scheduling (CoSched)**
  - Hypervisor has the complete control on scheduling
  - Hypervisor scheduling determines which domain should run on a GPU depending on the CPU schedule
  - Accelerator core are treated as slaves to host cores
  - Latency reduction by occasional unfairness
  - Possible waste of resources e.g. if domain picked for GPU has no work to do

Hypervisor Coordinated Policies

- **Augmented Credit-based Scheme (AugC)**
  - Scan the hypervisor CPU schedule to temporarily boost credits of domains selected for CPUs
  - Pick domain(s) for GPU(s) based on GPU credits and remaining CPU credits from hypervisor (augmenting)
  - Throughput improvement by temporary credit boost
- **Feedback-based proportional fair share(SLAF)**
  - Periodic scanning can lead to adjustment in the timer ticks assigned to aVCPUs if they do not get or exceed their assigned/expected time quota

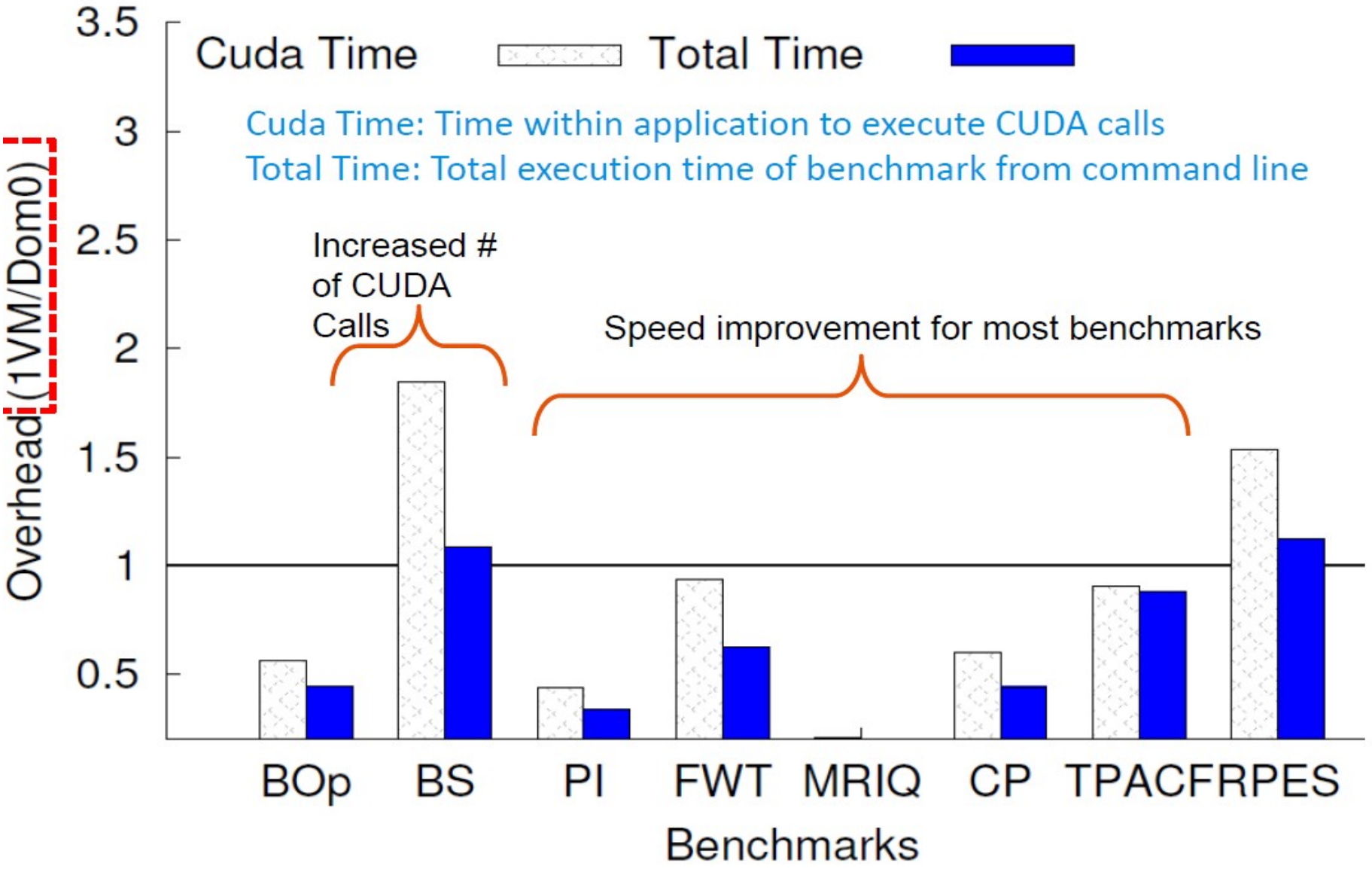
Experimental Evaluation

Experimental Set-up

- Xeon quad core @3GHz, 3GB RAM and NVIDIA 9800 GTX card with 2 GPUs
- Xen 3.2.1 hypervisor
- Linux kernel as guest domains with 512MB memory, 1VCPU each
- Benchmarks

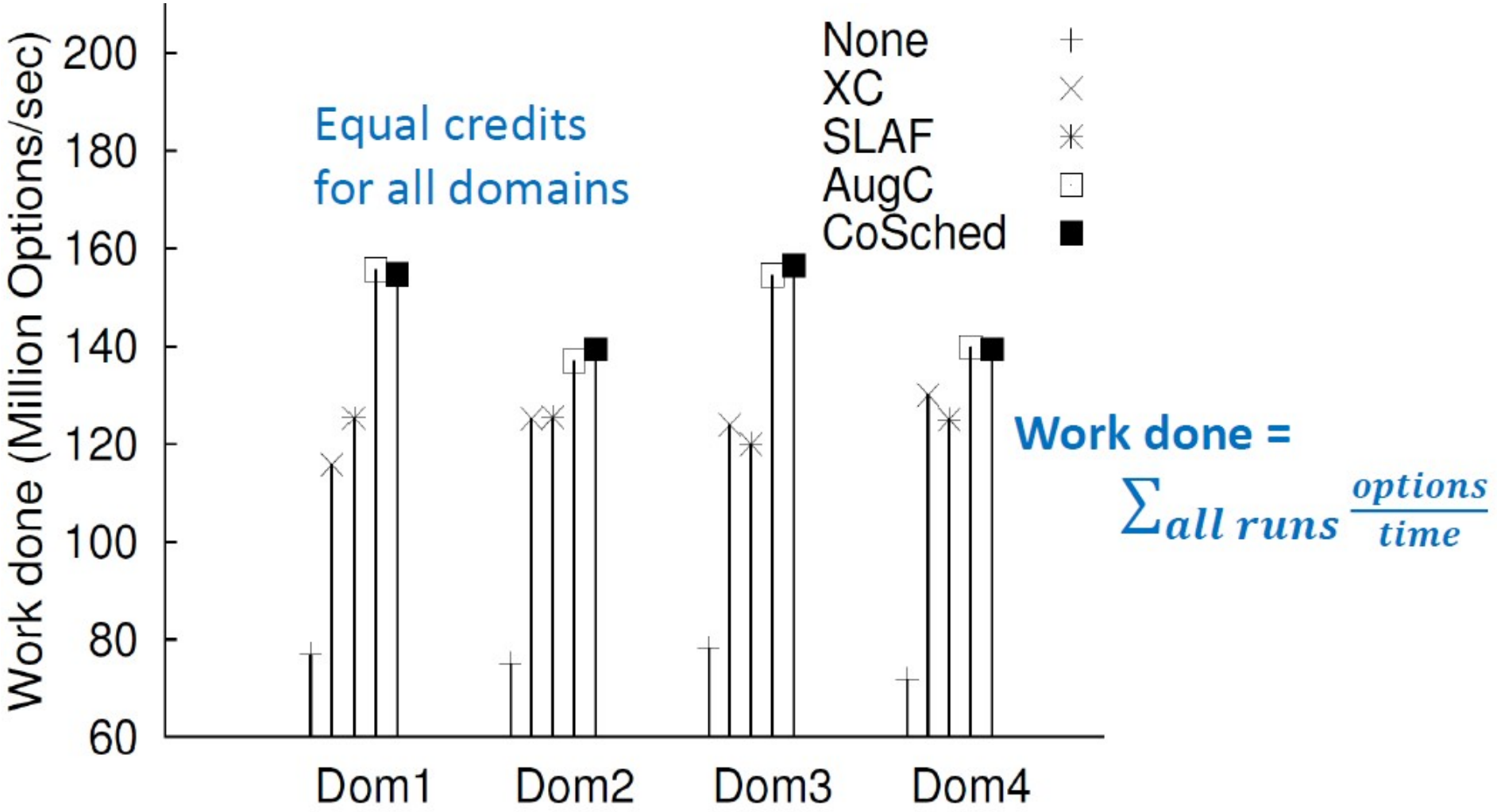
category	Source	
Financial	Cuda SDK	Bionomial(BOp), BlackScholes(BS), Mente-Carlo(MC)
Media-processing	Cuda SDK/parboil	ProcessImage(PI)=matrix multiply+DXTC, MRIQ, FastWalshTransform(FWT)
Scientific	parboil	CP, TPACF, RPES

Virtualization Overhead



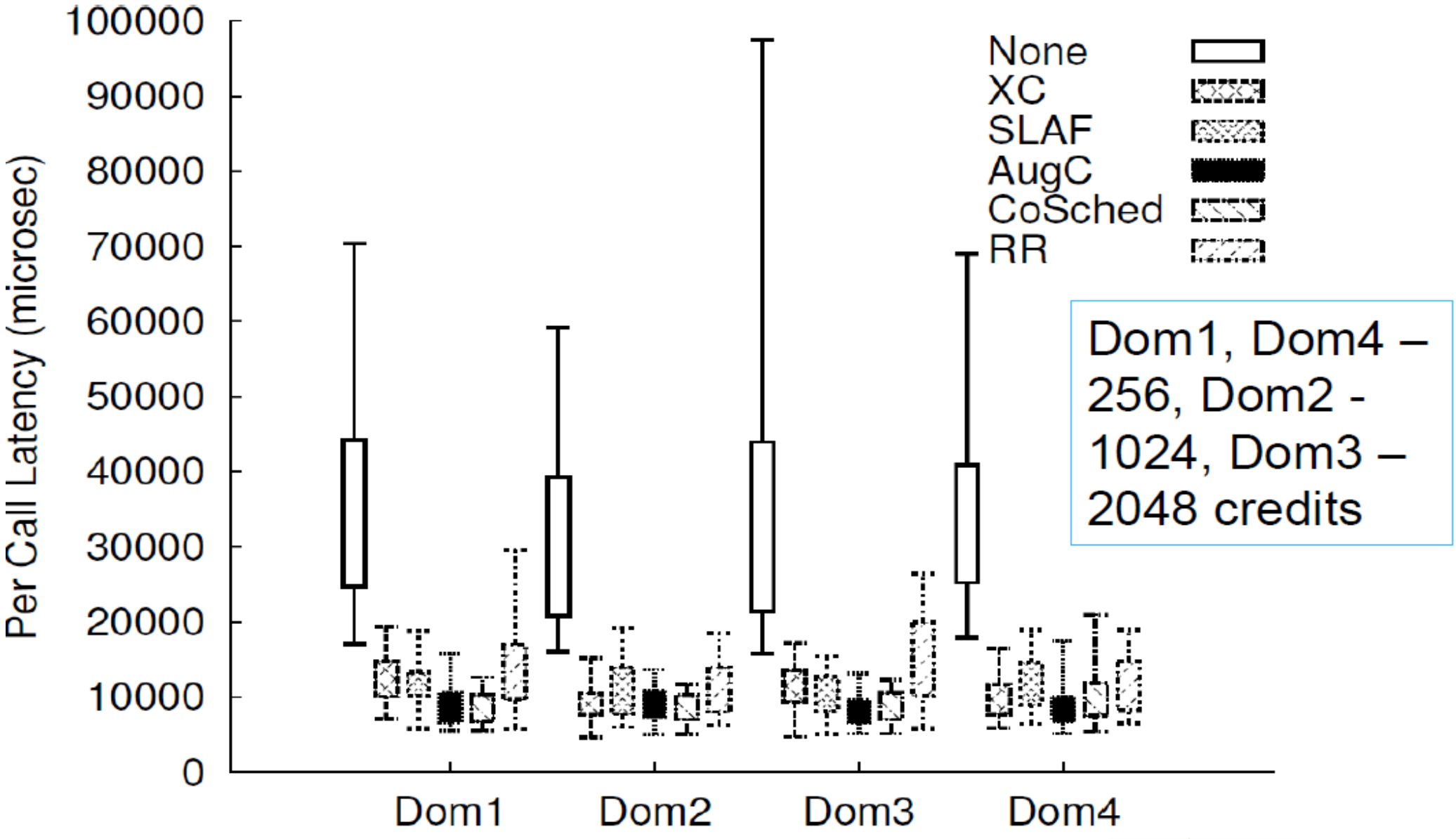
Scheduling Performance

Black Scholes – Latency and throughput sensitive



Scheduling Performance

FWT – Latency sensitive



Take Away from Pegasus

- Pegasus approach utilizes accelerators as first class schedulable entities via virtualization
- Coordinated scheduling is effective
  - Even basic accelerator request scheduling can improve sharing performance
  - Scheduling lowers degree of variability caused by un-coordinated use of the NVIDIA driver

Conclusions - Discussion

- Focus shift towards the systems aspect of GPUs/Accelerators
- What other opportunities open up by managing accelerators at the operating system level?
- Drawbacks? Is the system very sensitive to performance overheads (see Ptask micro-benchmarks)
- Are Ptask and Pegasus the right levels of abstraction?

Sources & References

- **PTask: Operating System Abstractions to Manage GPUs as Compute Devices** [SOSP, 2011]
- **Pegasus (Coordinated Scheduling in Virtualized Accelerator-based Platforms)** [Usenix ATC, 2011]
- **Operating Systems Challenges for GPU Resource Management** [OSPERT 11]
- **Graphic Engine Resource Management** [TR] Mikhail Bautin, Ashok Dwarakinath, Tzi-cker Chiueh
- **GPU Virtualization on VMware's Hosted I/O Architecture** [USENIX Workshop on I/O Virtualization, 2008]
- **TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments** [Usenix ATC, 2011]