While GPUs provide massive speedups for data-parallel workloads, they are difficult to share safely and efficiently across multiple virtual machines.

Existing approaches,

- GPU I/O pass-through
    - Expose GPU to guest device driver, minimize overhead of virtualization
    - Each GPU can only be owned by one VM.
- API remoting
    - Export API calls to outside of guest CMs.
    - The entire software stack must be rewritten to incoporate an API remoting mechanism.
- Para-virtualization,
    - Allow multiple VM access the GPU by providing an ideal device model through the hypervisor, the guest driver must be modified to support the device model.

The paper introduces GPUvm, hypervisor-level architecture for GPU virtualization, implemented on the Xen hypervisor. GPUvm allows multiple VMs to safely share a physical GPU while exposing a native GPU device model to guest OSes.

Model

The paper assumes a **typical server architecture**:

- A **multi-core CPU**
- A **discrete GPU** connected via PCI Express
- The GPU is treated as an **independent compute device**, not a CPU coprocessor
- The CPU cannot directly execute GPU code
- All communication happens through memory-mapped IO and DMA.

GPU kernel is a function offloaded from CPU to GPU.

- Executed by a large number of parallel threads on GPU cores
- A single application may launch many kernels, and mutliple kernels can be run concurrently.

GPU context represent

- The execution state of GPU computation
- Ownership of GPU resources
- A virtual address space on the GPU

GPU channel is a hardware mechanism used to

- Submit commands from CPU to GPU
- Isolate command streams from different contexts
- Each channel belongs to exactly one GPU context
- A GPU context may own multiple channels

GPU Page Table:

- GPU support paging just like CPU, but they have their own page tables for
    - Translates GPU virtual addresses
    - Map to
        - GPU physical memory
        - Host physical memory
    - So GPU code can directly access system RAM
    - DMA safety is enforced via GPU page tables

PCIe BARs(Base Address Registers):

- PCIe exposes GPU registers and memory via BARs
- BARs define MMIO windows
    - GPU control register
    - GPU memory apertures
    - Channel registers
- CPU controls the GPU by MMIO using BARs for reading/writing special memory address corresponding to the GPU registers.

Design

GPUvm exposes a **native-looking GPU device** to each VM, but **no VM ever touches the real GPU directly**.

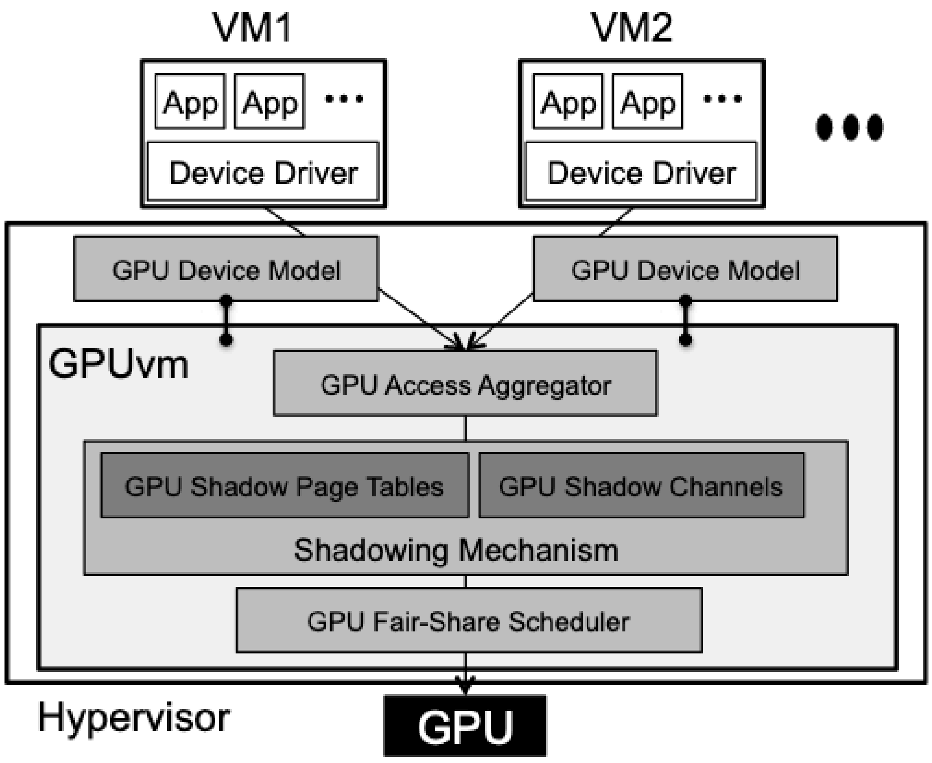


Figure 2: The design of GPUvm and system stack.

Virtualization Approaches

GPUvm supports

- Full Virtualization
    - Guest GPU driver is unmodified
    - Hypervisor intercept MMIO, shadows GPU data structures
        - GPUvm validate the contents of the shadow page table, so GPU memory can be shared safely by multiple guest VMs.
            - And ensures DMA initiated by GPU never access memory areas outside of those allocated to the VM.
        - The number of GPU channels is limited in hardware
            - GPUvm creates shadow channel to multiplex VMs on GPU channels.
            - GPUvm create and assign virtual channel to each VM, and maintain mapping between virtual and shadow channel.

Resource Partitioning

GPUvm partition physical memory space and MMIO space over PCIe BARs into multiple section of continuous address space, each is assigned to an individual VM.

Guest drivers assume:

- GPU memory starts at address 0
- BARs map directly to hardware

Static Partitioning (Prototype Choice)

- GPU memory is split into fixed regions
- Each VM gets a slice
- Easier to implement and reason about
- Dynamic allocation is possible and planned
- GPUvm
    - **Shifts and remaps addresses**
    - Uses **shadow page tables** to preserve the illusion

GPU Shadow Page Tables

- Guest driver maintains its own GPU page table
- GPUvm maintains a shadow page table
- The real GPU only uses the shadow page table
- GPUvm
    - Validate guest mappings
    - Filters out illegal pages
    - Prevent DMA to
        - Other VMs' memory
        - Hypervisor memory
        - GPUvm's own metadata
- The device driver flush TLB caches every time a page table entry is updated.
    - GPUvm can intercept TLB flush requests because those requests are issued from the host CPU through MMIO.
    - After interception, GPUvm updates the corresponding GPU shadow page table entry.

GPU Shadow Channels

- GPUs have **few hardware channels**
- Channels have **numeric IDs**
- Drivers assume exclusive ownership

GPUvm introduces **virtual channels**:

- Each VM sees its own channel numbers (0,1,2,…)
- GPUvm maps virtual channel to physical channel
- Mapping is hidden from guests
- Channel activation and command submission is done via MMIO

    - GPUvm intercepts these accesses
    - Redirects to correct physical channel
- Multiple VMs safely share limited channels
- Each VM believes it owns the GPU

GPU Fair-Share Scheduler

Use the BAND scheduler in Gdev paper, for mitigating non-preemptive problem.

Optimization Techniques

Lazy shadowing:

- Scanning page tables on every TLB flush is expensive
- GPUvm delay shadow updates until
    - GPU memory is accessed
    - Kernel execution begins
- So it have reduced number of page table scans.

BAR Remap:

- Every BAR access traps into hypervisor
- GPUvm pass through BAR accesses except for channel descriptors
    - Because it don't need to virtualize the values read from or written to the BAR area except for channel descriptors
    - The BAR accesses that got passed through is isolated using shadow page tables among multiple VMs.
- So it have fewer traps

Para-virtualization:

- Detecting page table change and shadowing page table is costly in full-virtualization
- GPUvm supports para-virtualization, guest driver cannot write page tables directly
    - They must use hypercalls, and GPUvm validates updates immediately
- Inspired from Xen.

Multicall:

- Hypercalls are expensive
- Multiple operation can be batched into one hypercall
- Inspired from Xen.

Implementation

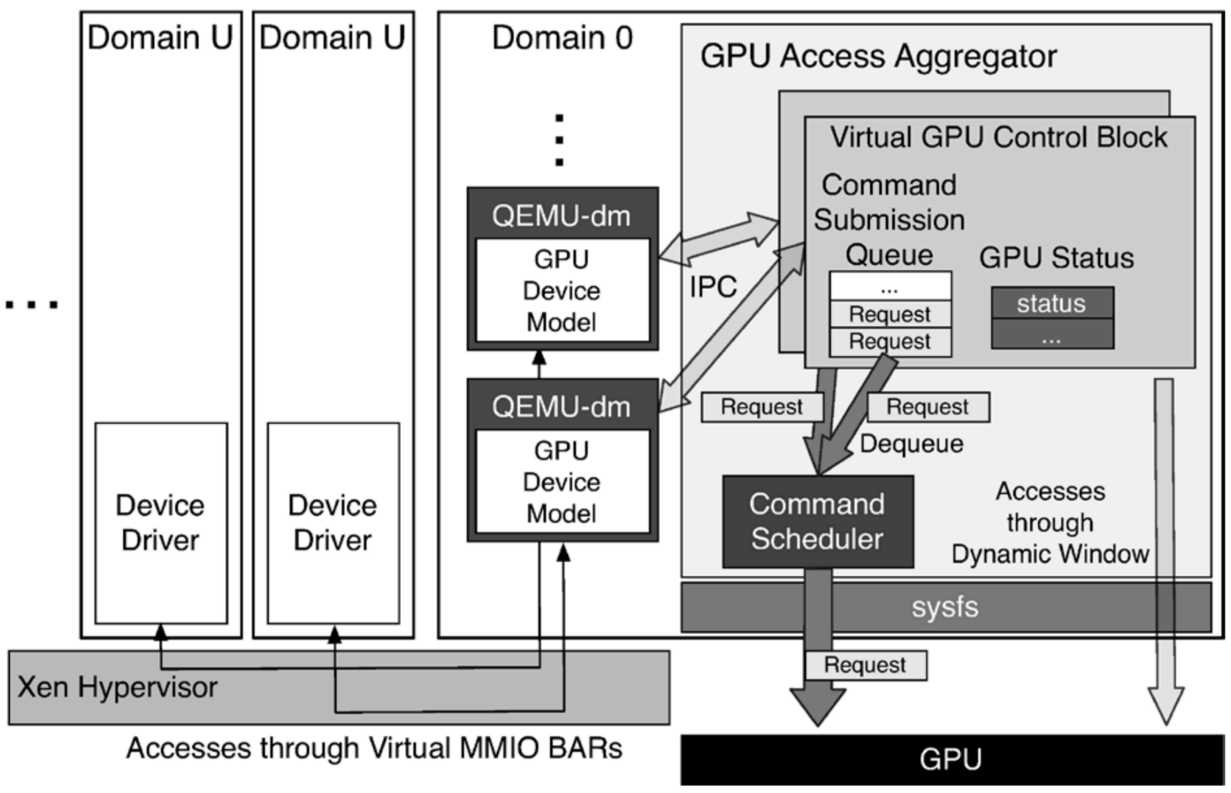


Figure 3: The prototype implementation of GPUvm.

GPUvm is implemented on Xen 4.2.0 hypervisor with Linux 3.6.5 running on Dom0, and DomU as guest VMs.

- GPU device model exposed in DomU is created by QEMU-dm, and it behaves as a vGPU.
- The Guest GPU driver runs on their VMs in DomU, and it consider the vGPU device model as normal GPU
- QEMU-dm exposes MMIO PCIe BARs, handling access to the BARs in Xen.
- GPU Access Aggregator is a user-space process in Dom0, and it manage shadow page tables, validate and schedule GPU commands received from each guest VMs.
    - GPU device model communicate with GPU Access Aggregator using POSIX IPC.
- Each VM has a vGPU control block at Dom0, that tracks virtual channels, page table state, command queues.
    - When guest driver changes GPU state or a command for writting MMIO register
    - The device model send IPC to update the corresponding control block.

Evaluation

Compares against

- Native (non-virtualized Linux 3.6.5)
- PT(pass-through provided by Xen's pass through feature)
- FV Naive(Full-virtualization without any optimization techniques)

- FV Optimized
- PV Naive(Para-virtualization without multicall)
- PV Multicall

Overhead

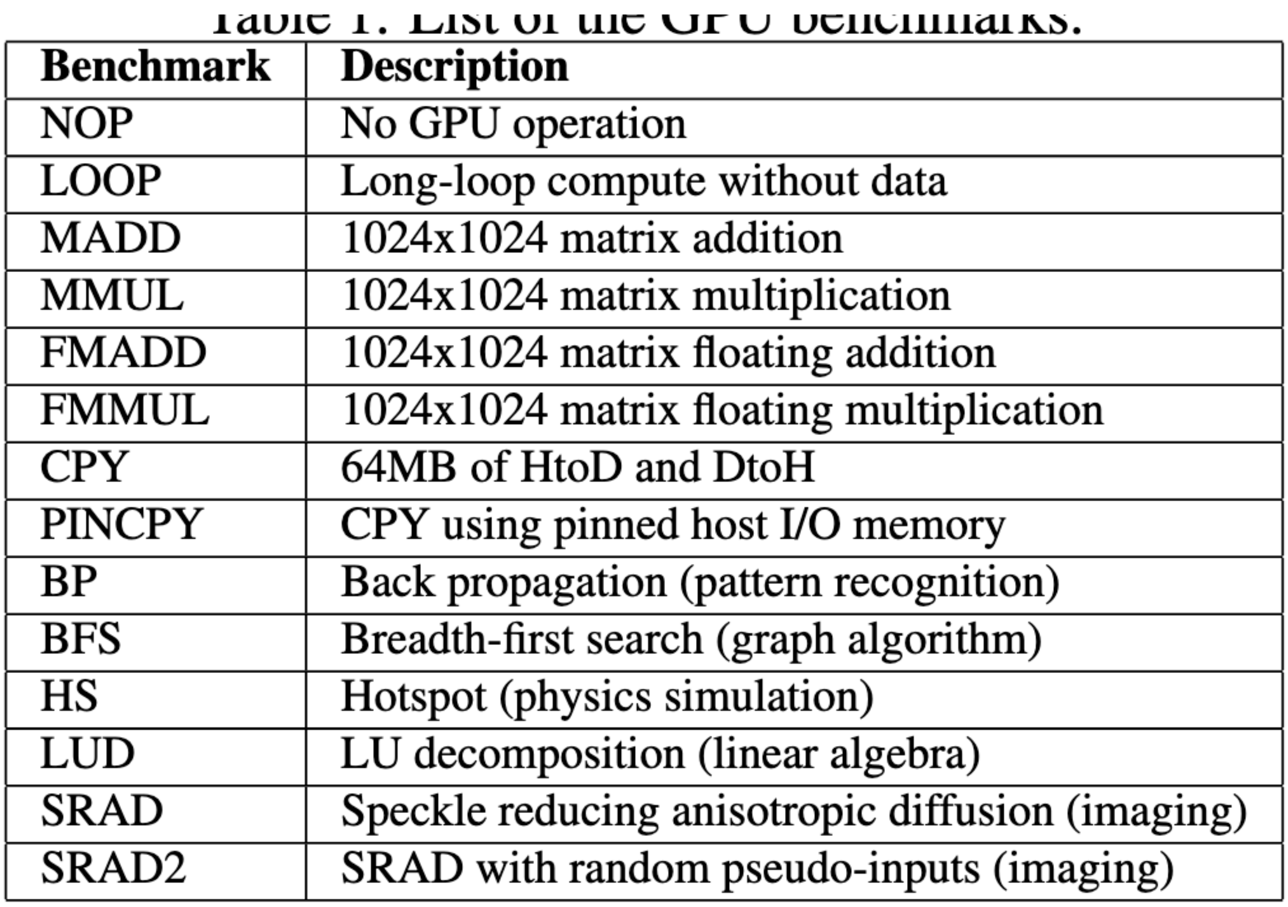


Table 1: List of the GPU benchmarks.

Benchmark	Description
NOP	No GPU operation
LOOP	Long-loop compute without data
MADD	1024x1024 matrix addition
MMUL	1024x1024 matrix multiplication
FMADD	1024x1024 matrix floating addition
FMMUL	1024x1024 matrix floating multiplication
CPY	64MB of HtoD and DtoH
PINCPY	CPY using pinned host I/O memory
BP	Back propagation (pattern recognition)
BFS	Breadth-first search (graph algorithm)
HS	Hotspot (physics simulation)
LUD	LU decomposition (linear algebra)
SRAD	Speckle reducing anisotropic diffusion (imaging)
SRAD2	SRAD with random pseudo-inputs (imaging)

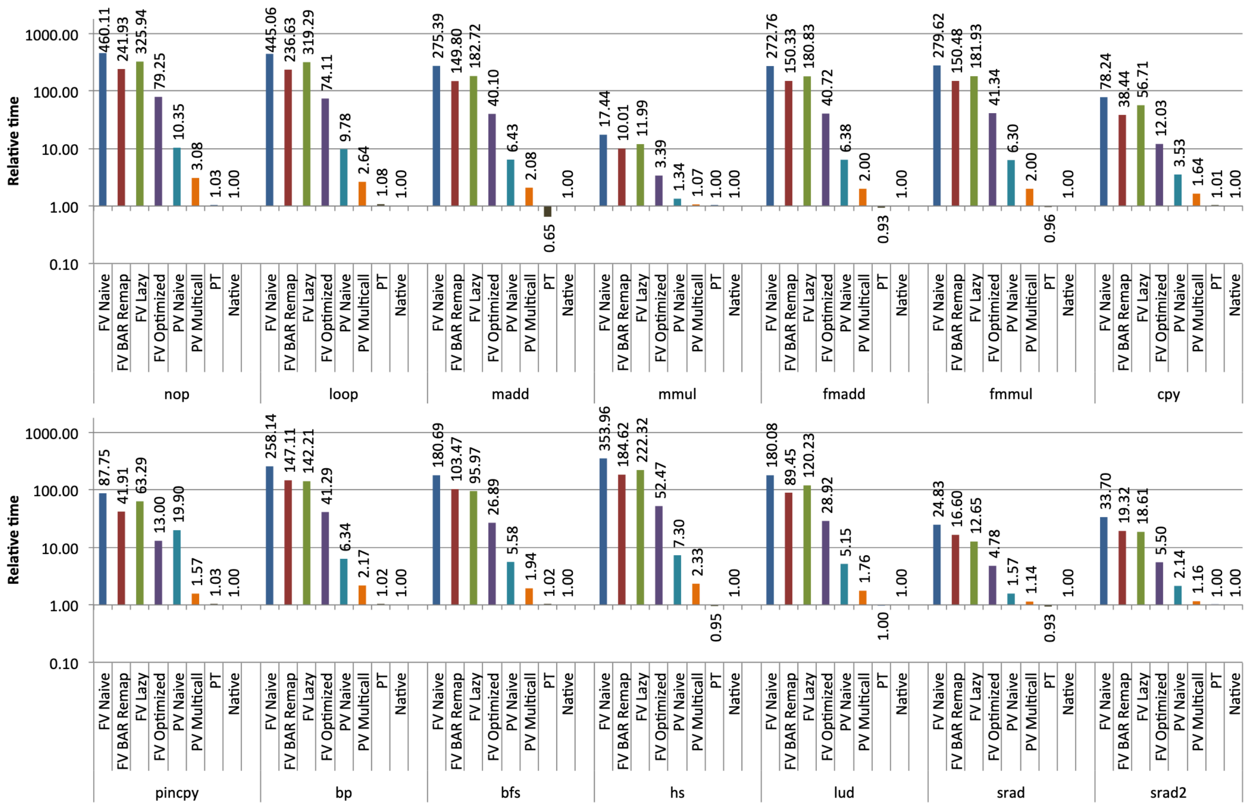


Figure 4: Execution time of the GPU benchmarks on the eight platforms.

- The overhead of Naive Full Virtualization is unacceptable
    - It's 100 times slower in nine benchmarks than Native.
    - These overhead can be mitigated by optimization
        - Combining the optimization techniques together achieves more performance gain.
    - In madd, PT is faster than Native, the author says "This is a GPU's mysterious behavior"
    - PV Naive is 3-10 times slower than Native
        - PV Multicall often within 2-3 times of native and sometimes close to passthrough

---

Each benchmark is divided into

- Init - GPU setup
- HtoD - Host to Device memory copy
- Launch - GPU execution
- DtoH
- Close - teardown

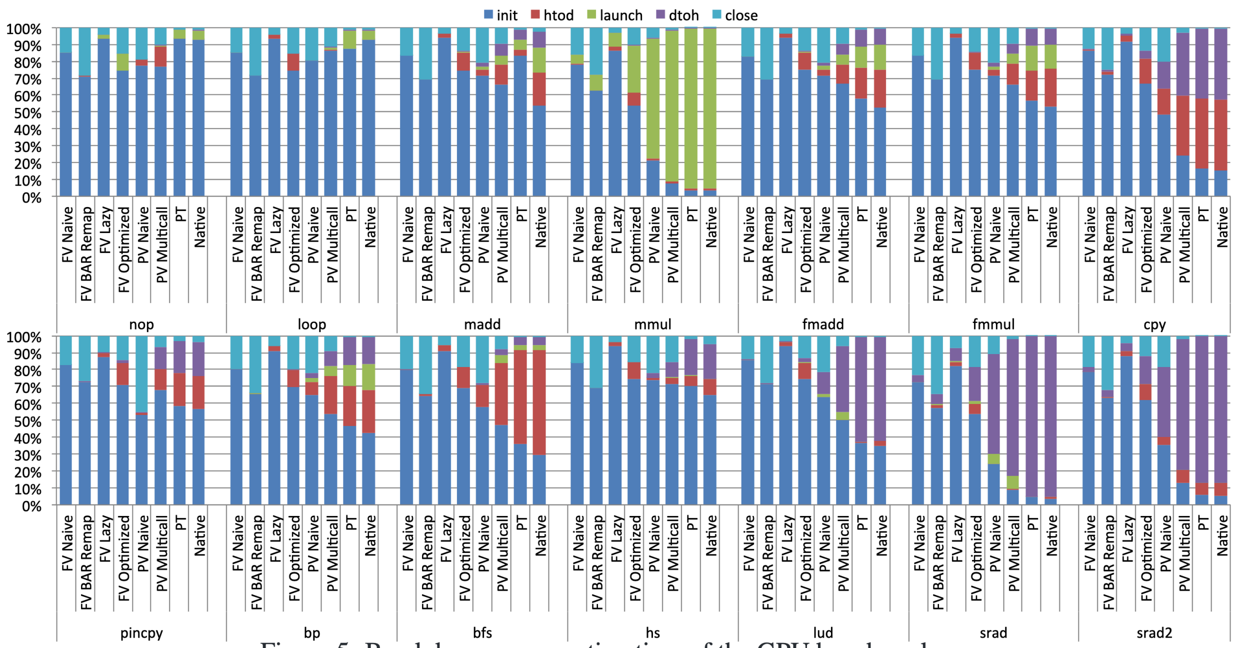


Figure 5: Breakdown on execution time of the GPU benchmarks.

In FV:

- Init and Close dominates(more than 90% execution time)
    - Using optimization techniques, these phase's ratio are lowered.
- GPU execution itself is fast
- Virtualization overhead is mostly control, not compute

BARremap helps benchmark with heavy BAR writes by avoiding hypervisor traps on every MMIO

Lazy shadowing reduces shadow page table updates to about 7 per benchmark.

Table 3: Update count for GPU shadow page tables.

	nop	loop	madd	mmul	fmadd	fmmul	cpy	pincpy	bp	bfs	hs	lud	srad	srad2
FV Naive	30	30	34	34	34	34	26	28	40	42	34	30	52	40
FV Optimized	7	7	7	7	7	7	6	6	7	7	7	7	7	7

Table 4: The number of hypercall issues.

	nop	loop	madd	mmul	fmadd	fmmul	cpy	pincpy	bp	bfs	hs	lud	srad	srad2
PV Naive	1230	1230	1420	1420	1420	1420	2010	34784	1628	1993	1169	1429	1985	2681
PV Multicall	93	93	97	97	97	97	81	218	117	117	97	89	149	107

For one VM with multiple GPU contexts, FV scales poorly, the init/cose cost grows with number of contexts.

- PV performance is similar to passthrough, and scales well.

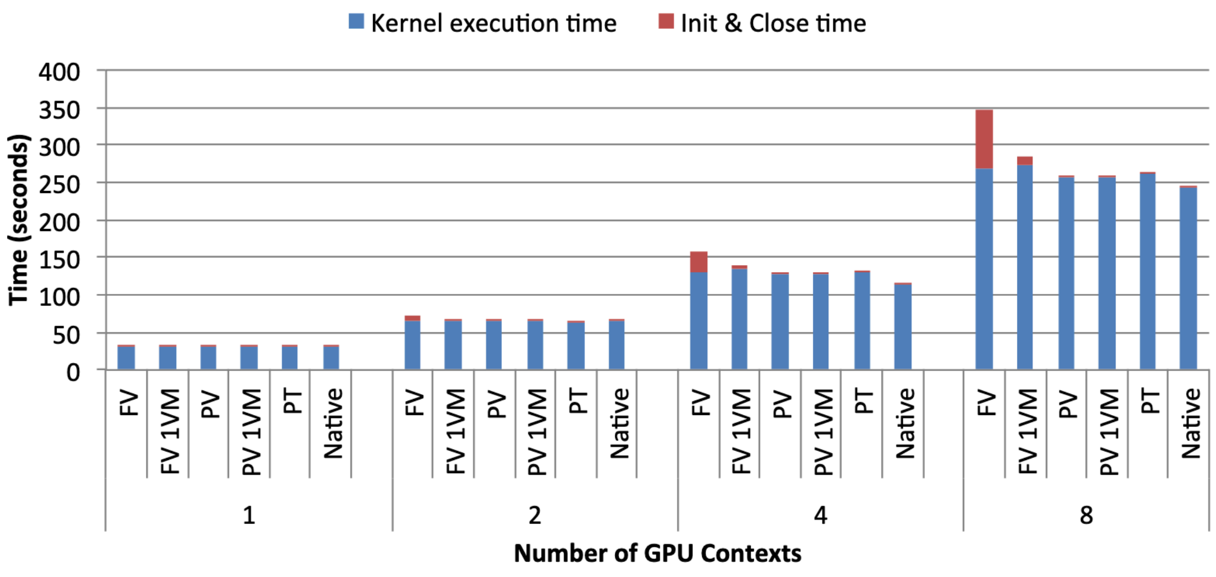


Figure 6: Performance across multiple VMs.

For multiple VM with one GPU context,

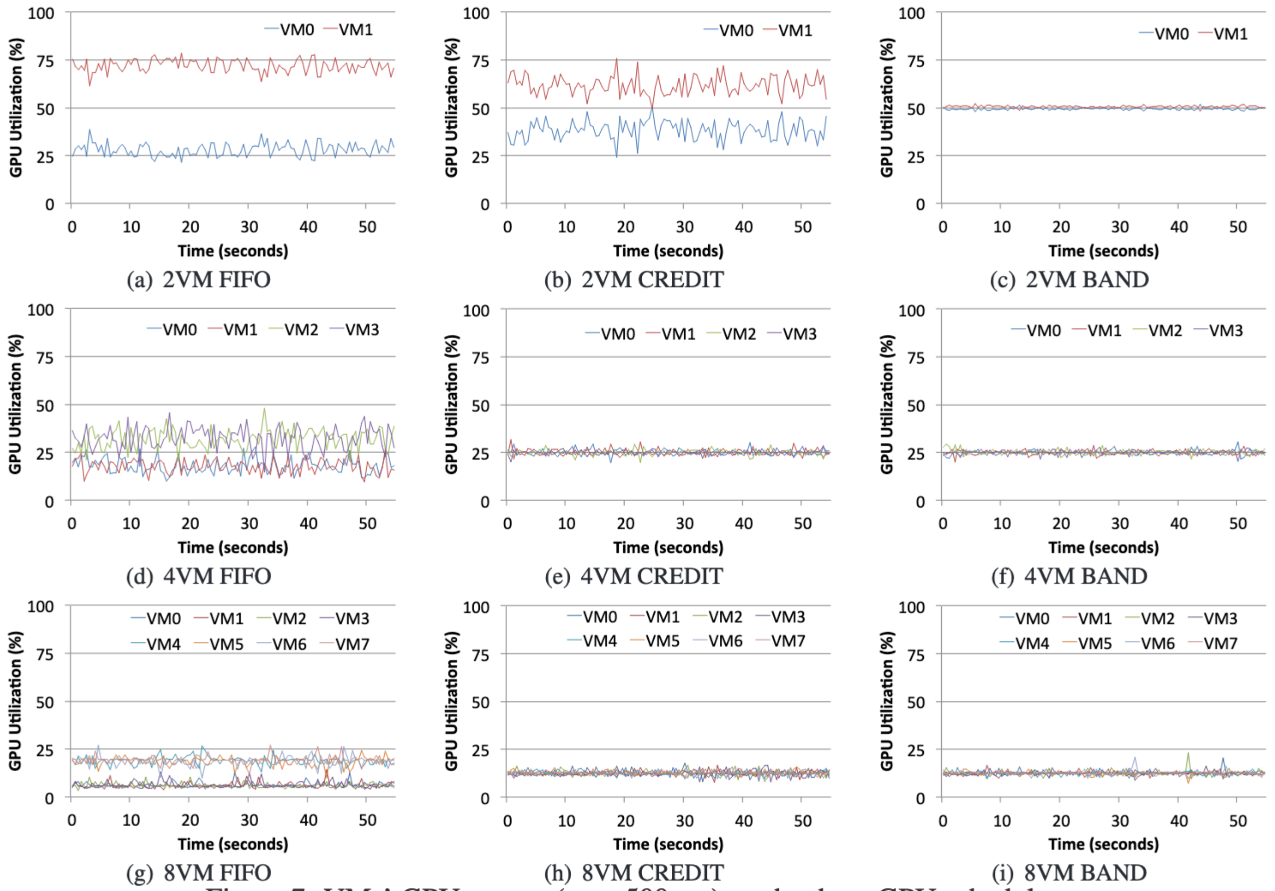
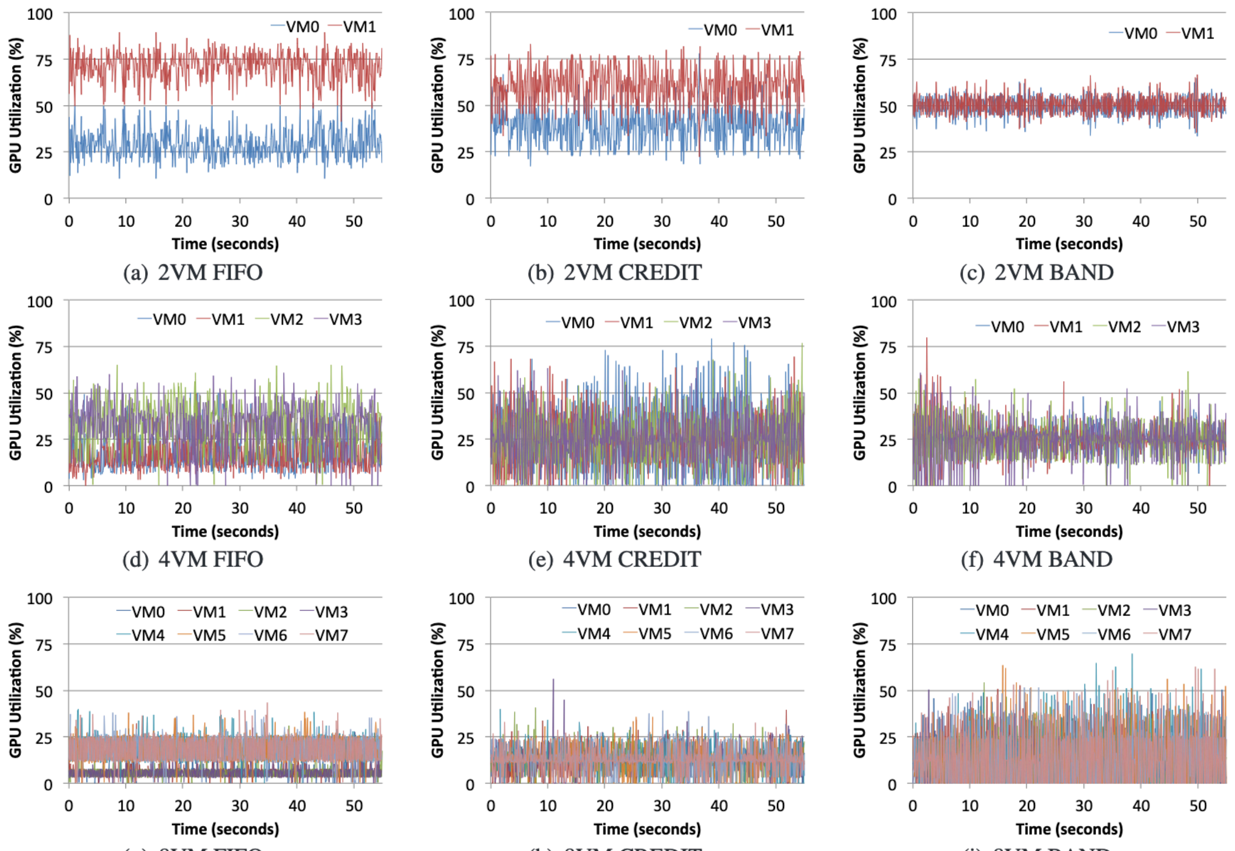


Figure 7: VMs' GPU usages (over 500 ms) on the three GPU schedulers.



(a) 2VM FIFO (b) 2VM CREDIT (c) 2VM BAND
(d) 4VM FIFO (e) 4VM CREDIT (f) 4VM BAND
(g) 8VM FIFO (h) 8VM CREDIT (i) 8VM BAND

Figure 8: VMs' GPU usages (over 100 ms) on the three GPU schedulers.

BAND scheduler is the best performing one, with best fairness in all cases. GPU usages are roughly equal for all VMs.