The paper introduces Gdev, an operating-system–level framework(implemented as Linux kernel module) that treats GPUs as first-class computing resources, similar to CPUs. Unlike traditional GPU programming models that rely heavily on user-space runtime libraries, Gdev integrates GPU runtime support directly into the OS. This design enables secure, efficient, and controlled GPU sharing across multiple applications and even allows the OS kernel itself to execute GPU code.

Existing GPU software stacks suffer from several limitations:

- GPUs are managed primarily in user space by providing graphical/computing APIs, limiting OS control.
    - Prevents file system or network stack in OS from using GPUs directly.
- No robust multitasking support: poor isolation, no QoS guarantees.
    - You can launch any user-space program on GPU using `ioctl` system call directly.
        - GPUs should be protected by the OS as well as CPUs.
- No inter-process GPU memory sharing.
- GPU memory allocations cannot exceed physical device memory.

Gdev introduces:

1. Runtime-Integrated OS Design
   • Moves GPU runtime support into the OS kernel.
   • User-space applications and kernel modules use the same GPU API.
   • Prevents unprivileged programs from bypassing GPU resource management.
2. First-Class GPU Resource Management
   • Device memory management
   • Inter-process communication (IPC) via shared GPU memory
   • GPU scheduling and virtualization
   • Priority propagation from CPU tasks to GPU contexts
3. Open source implementation of GPU driver, runtime/API libraries, utility tools, and Gdev resource management primitives.

System Model

A GPU program typically follows these steps:

1. Allocate memory on the GPU (device memory)
2. Copy input data from CPU memory to GPU memory
3. Launch a GPU kernel (computation)
4. Copy results back to CPU memory
5. Free GPU memory

GPU Commands:

- CPU submit commands to the GPU, commands are placed into a FIFO queue associated with a GPU context(memory copy, kernel launch etc.)
- If the OS wants control, it must manage *when* commands are allowed to reach the GPU.

GPU Contexts and Channels:

- Each GPU application runs in a **GPU context**.
- Each context is assigned a **hardware channel**.
- Channels allow multiple contexts to *exist*, but:
    - Only **one channel can actively use the GPU's compute units at a time**
    - Contexts are switched by hardware, not by the OS

Address Space and Virtual Memory:

- Each GPU context has its own **virtual address space**
- Address translation is done by a **GPU Memory management unit**
- Page tables are configured by the device driver

Compute Units

- GPU threads are mapped internally to cores by the GPU hardware
- The OS **cannot see or control individual threads**
- Scheduling decisions must be made at the **context level**, not thread level
- Multiple kernels from the *same context* can overlap
- Kernels from *different contexts* cannot run simultaneously

DMA (Data Transfer) Units
There are two types of data transfers:

1. Synchronous DMA
    - Cannot overlap with computation
2. Asynchronous DMA
    - Can overlap with computation

Additional constraints:

- DMA operations are **non-preemptive**
- Once started, they must complete

Gdev Eco System

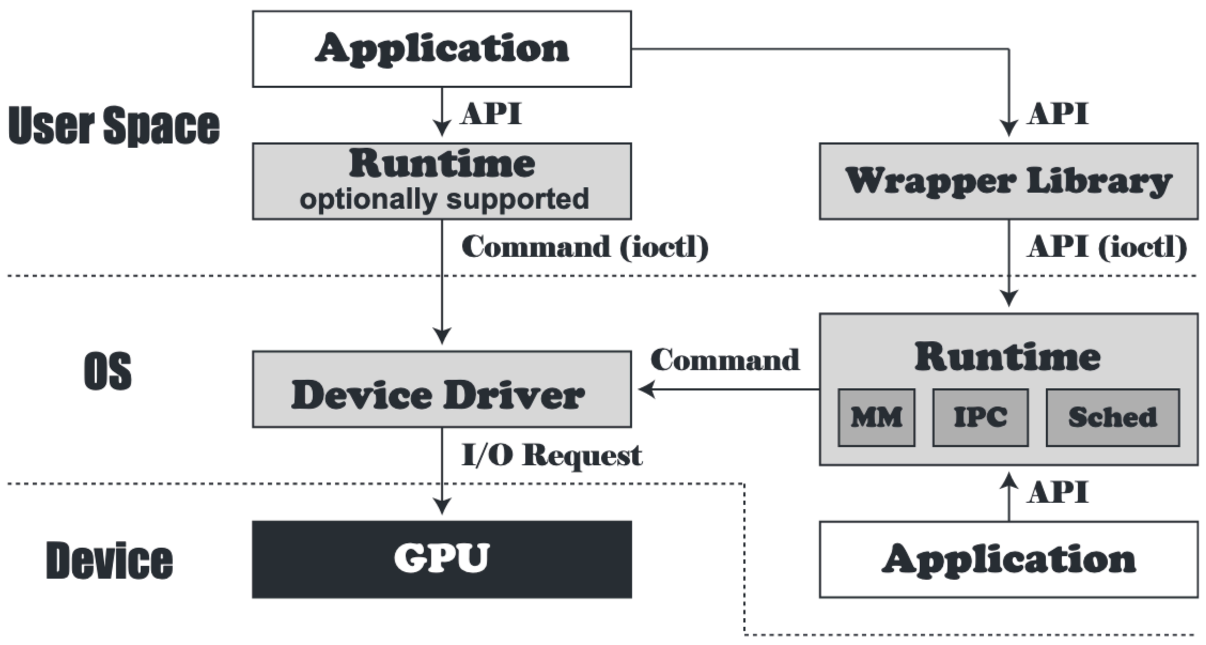


Figure 1: Logical view of Gdev's ecosystem.

Move GPU runtime support into the operating system, instead of keeping it only in user space.

Traditional Model (What's wrong)
In conventional systems:

- Applications call a **user-space runtime library** (e.g., CUDA, DirectX, Vulkan...)
- The runtime translates API calls into GPU commands
- The OS only sees **low-level commands**, not high-level intent

This causes two major problems:

1. **Limited OS control**
   - The OS cannot easily tell *what* the GPU is doing (compute vs copy, memory usage, priorities).
2. **Security and reliability issues**
   - Applications can bypass the runtime and issue GPU commands directly via system calls (e.g., ioctl)
   - This allows resource abuse and breaks isolation

Gdev introduces a **runtime-unified ecosystem**, shown conceptually in Figure 1 of the paper.

- The **GPU runtime lives inside the OS**
- Both user applications *and the OS itself* use the **same GPU API**
- User-space calls are forwarded through a **thin wrapper library**
- The OS controls all GPU access at the API level

Instead of scheduling and accounting at every GPU command:

- Gdev manages GPU resources **when API calls are made**
  - Like Memory copy API, kernel launch API...
- API calls are **semantically meaningful**
- The OS knows:
  - Which memory is being accessed
  - Whether the operation is compute or data transfer
  - Which context owns which resources

This avoids the heavy overhead seen in **command-driven schedulers**, which must intercept and analyze every low-level GPU command.

Gdev API:

- Gdev defines a **low-level GPU API**
- This API serves as a backend for higher-level APIs like CUDA
- Programmers can:
  - Use Gdev API directly, or
  - Use CUDA built on top of Gdev

In the paper’s prototype:

- CUDA Driver API 4.0 is implemented on top of Gdev
- Existing CUDA programs can run unchanged

Using the Gdev ecosystem, The OS kernel can execute GPU programs just like user applications.

This enables GPU acceleration for:

- File systems
- Network stacks
- OS services

Because the runtime is inside the OS, Gdev can implement features that are impractical in user space:

1. Shared Device Memory
   - Explicit APIs to share GPU memory between contexts
   - Enables fast inter-process communication (IPC)
2. Data Swapping
   - GPU memory allocations can exceed physical device memory
   - Excess data is transparently swapped to host memory
3. GPU Scheduling

   - Context-aware scheduling
   - Priority propagation from CPU tasks to GPU tasks
4. GPU Virtualization
   - One physical GPU → multiple logical GPUs
   - Each with controlled memory and bandwidth

Device Memory Management

In traditional GPU systems:

- GPU memory is **separate** from CPU memory
- Data must be copied back and forth explicitly
- GPU memory allocations **cannot exceed physical capacity**
- GPU contexts **cannot share memory**

These limitations make GPUs fragile and inefficient in multi-tasking environments.

Memory Copy Optimization - Split Transaction:

- A memory copy to communicate with GPU involves two stage:
  - Copy data inside host memory(main memory to pinned I/O memory)
  - Copy data between host I/O memory and GPU via DMA
  - These stages happens sequentially wasting time
- Gdev split large transfers into fixed-size chunks
  - Host-memory copy of chunk N+1
  - GPU DMA copy of chunk N
  - Transfer time is cut in half
- For small data sizes, setting up DMA is slower than the direct I/O access. So Gdev maps GPU memory into host I/O space directly, and read/write data one word at a time to avoid DMA setup overhead.

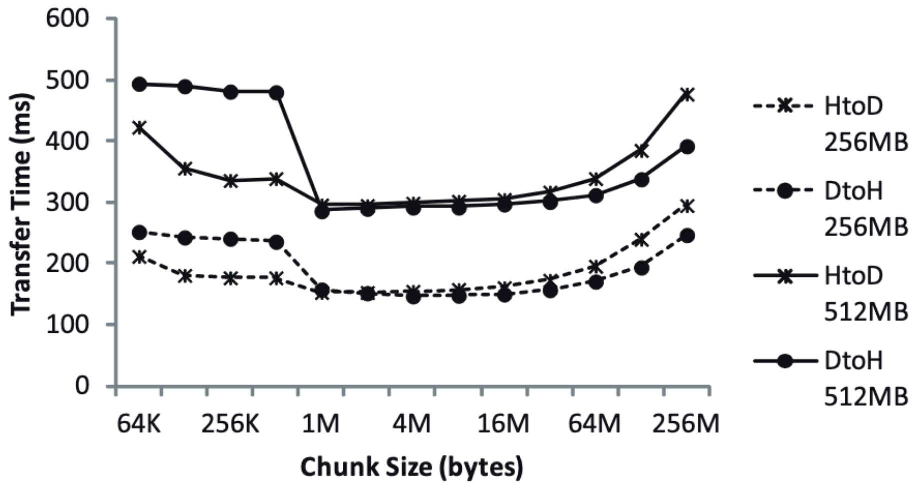


Figure 3: Impact of the chunk size on DMA speeds.

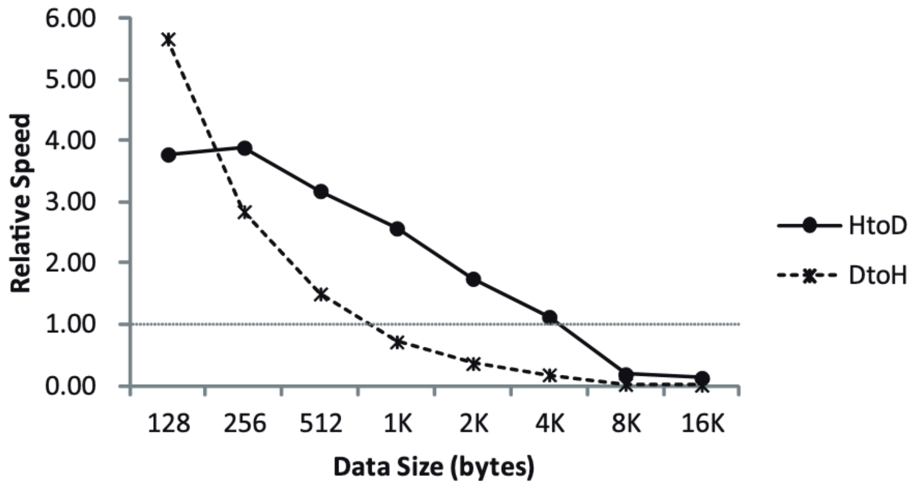


Figure 4: Relative speed of I/O access to DMA.

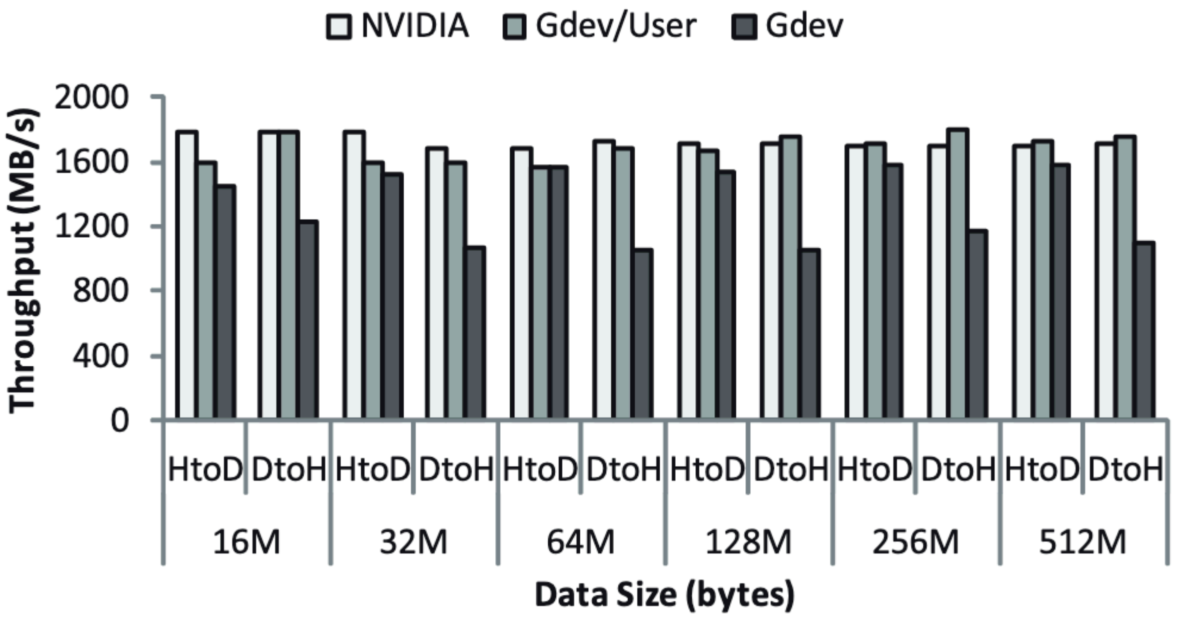
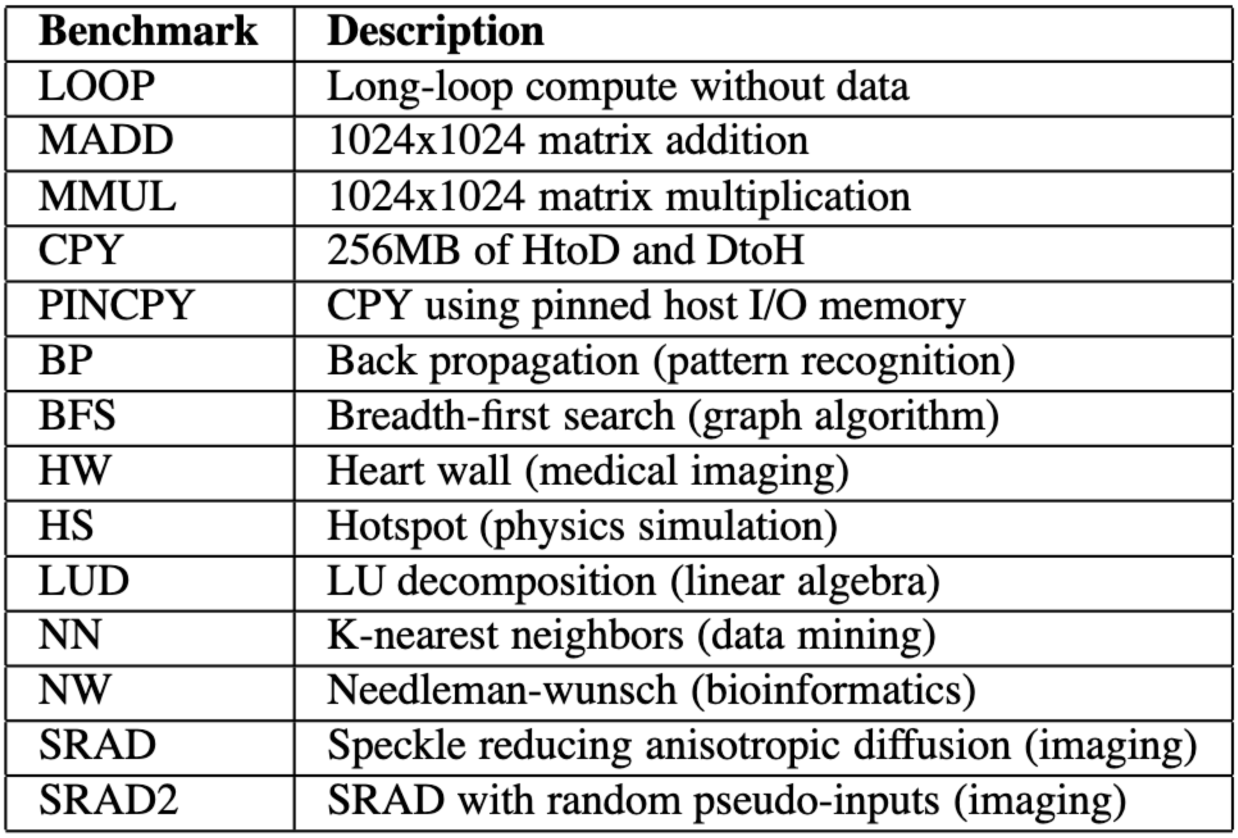


Figure 5: Memory-copy throughput.

Gdev/User employs runtime library in user-space, Gdev integrates runtime support into OS kernel.

Table 1: List of benchmarks.



Benchmark	**Description**
LOOP	Long-loop compute without data
MADD	1024x1024 matrix addition
MMUL	1024x1024 matrix multiplication
CPY	256MB of HtoD and DtoH
PINCPY	CPY using pinned host I/O memory
BP	Back propagation (pattern recognition)
BFS	Breadth-first search (graph algorithm)
HW	Heart wall (medical imaging)
HS	Hotspot (physics simulation)
LUD	LU decomposition (linear algebra)
NN	K-nearest neighbors (data mining)
NW	Needleman-wunsch (bioinformatics)
SRAD	Speckle reducing anisotropic diffusion (imaging)
SRAD2	SRAD with random pseudo-inputs (imaging)

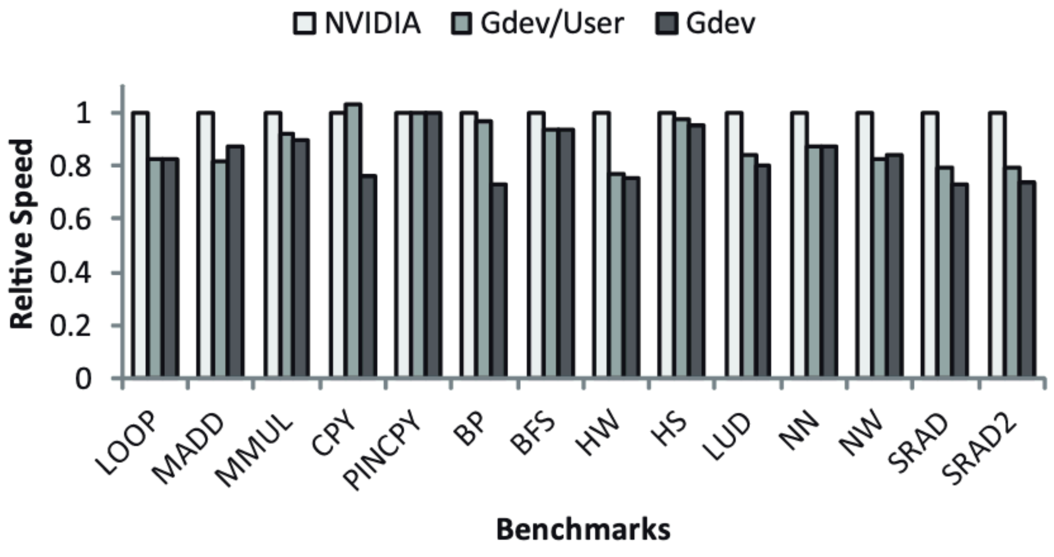


Figure 6: Basic standalone performance.

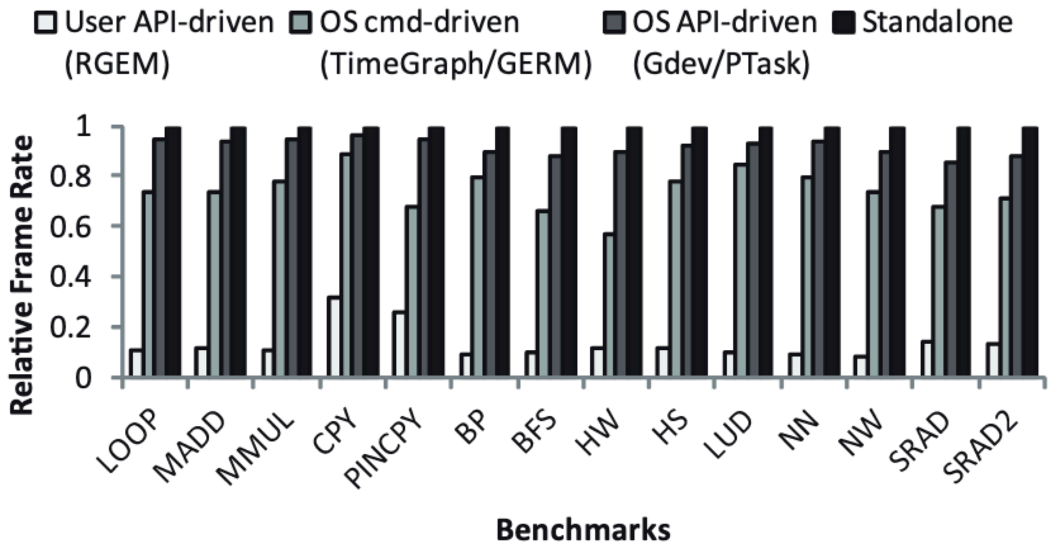


Figure 7: Unconstrained real-time performance.

- Some compute-intensive workloads show up to ~20% slowdown due to:
    - Nvidia driver has some proprietary “performance mode” for GPU? So the compute task runs faster in NVIDIA driver.

Shared Device Memory:

- In conventional system, GPU contexts cannot share memory, and Data to be shared between the context must be copied from GPU to CPU and vice versa.
- Gdev introduces explicit GPU shared memory, modeled after POSIX shared memory using POSIX IPC functions of `cuShmGet, cuShmAt, cuShmDt, cuShmCtl` in their CUDA implementation.
    - So CUDA applications can easily use Gdev's shared device memory without modification.

- Shared device memory enables:
  - Fast inter-process GPU communication
  - Efficient dataflow pipelines

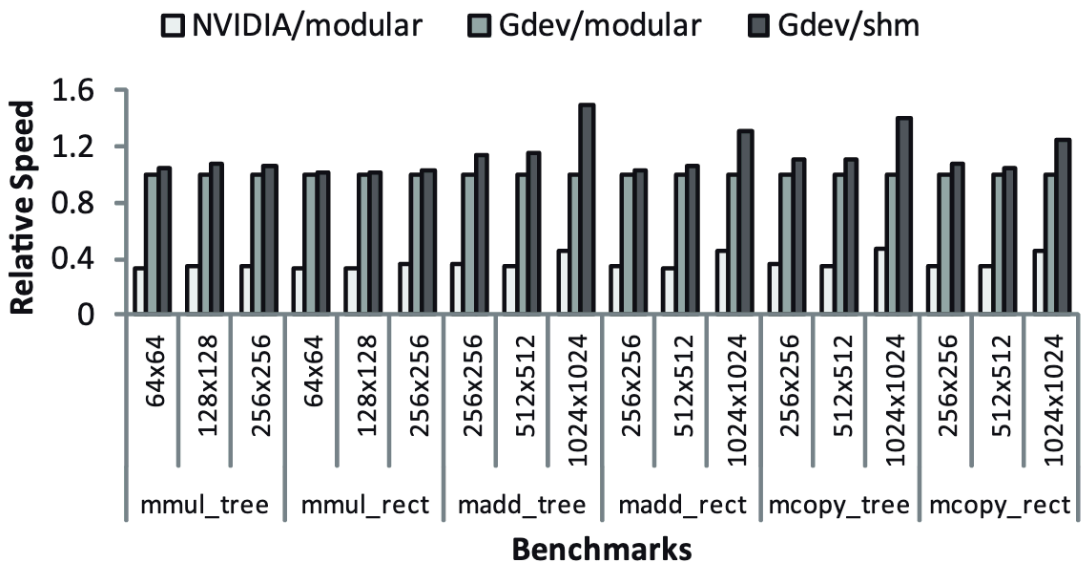


Figure 11: Impact of shared memory on dataflow tasks.

- Dataflow benchmarks show up to **49% speedup**
- Gains come from eliminating redundant host-device copies
- Especially effective for large intermediate data

Data Swapping:

- Most GPU runtimes fail when memory allocation exceeds physical GPU memory
- Some proprietary driver swap implicitly, but without control or guarantees
- Gdev introduce GPU memory swapping so the context can exceed physical device memory, just like virtual memory for CPUs.
  - When allocation fails:
    - Gdev selects a **victim memory object**
      - Preferably from a low-priority context
    - Data is evicted(swapped) to host memory
    - When needed again, data is restored transparently
- But memory swapping between GPU and host is slow, so Gdev reserve a portion of GPU memory as temporary swap space.
  - First evict data to this fast on-device space
  - Later move it to host memory in the background
  - Gdev Use a **dedicated GPU context** for swap transfers
    - Overlap eviction with computation

    - If the context is requesting evicted data that haven't been swap to the host yet, it can be retrieve directly from this on-device temporary swap space.

GPU Scheduling

CPU schedulers(round-robin, CFS) do not work well for GPUs because CPU threads and GPU context works fundamentally different.

- CPU threads are preemptive, while GPU context are non-preemptive
- Threads are fine-grained, contexts are long running coarse grained.

Gdev schedule GPU access when API calls occur, not when individual GPU commands are issued, this allows the OS to

- Understand the intent of the API call(compute vs DMA)
- Reduce scheduler overhead

Compute and DMA has separate scheduler thread and separate waiting queue.

- So compute and DMA can overlap
    - Compute and DMA can be run simultaneously on the GPUs.
- Each resource accounted independently

GPU Virtualization

Gdev allows one physical GPU to appear as **multiple virtual GPUs (vGPUs)**.
Each vGPU is assigned:

- **Memory share** (space)
- **Compute bandwidth** (time)
- **Memory-copy bandwidth** (time)

Each application is bound to a specific vGPU.

Gdev introduces the BAND(Bandwith-Aware Non-preemptive Device) scheduler based on the credit scheduler

- Bandwith-aware credit handling
    - Credits are reduces as the allocated GPU time is exhausted and actual utilization of GPU is exceeding its bandwith.
    - Compensates for non-preemptive over-runs
    - vGPU with least usage of credit gets priority
- Time buffering
    - Scheduler waits a short period of time before dispatching the next vGPU
    - Allows short jobs to "cut in"
    - Reduce unfairness caused by long-running kernels

- New arrivals can immediately grab the GPU if eligible
- Avoids unnecessary idling

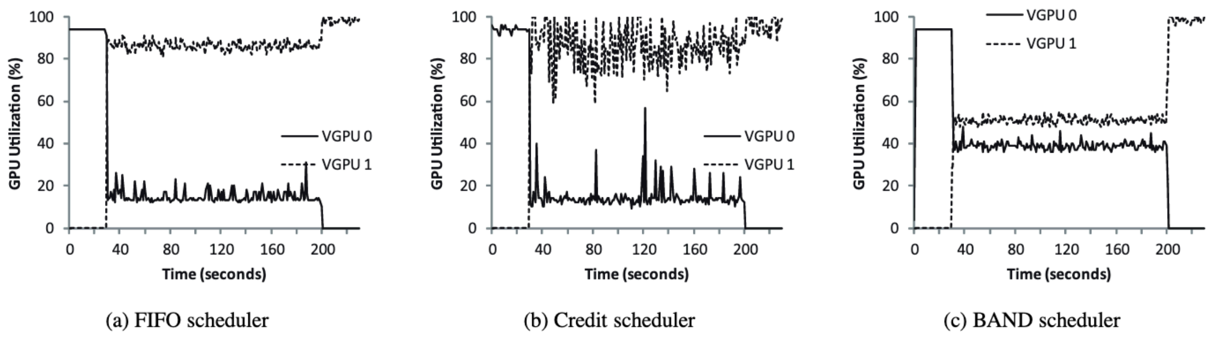


Figure 14: Util. of virtual GPUs under unfair workloads.

Evaluation

GPU Acceleration for the OS

GPU-accelerated encrypted file system eCryptfs

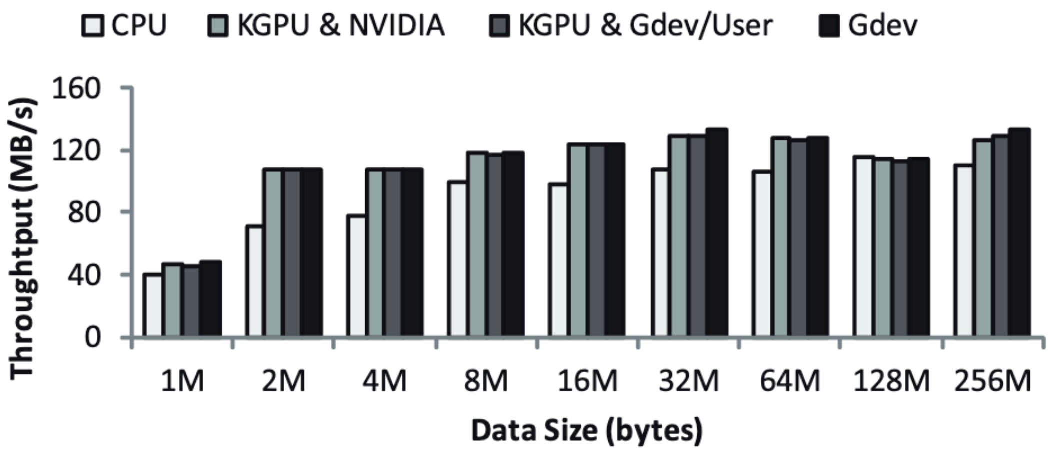


Figure 8: eCryptfs read throughput.

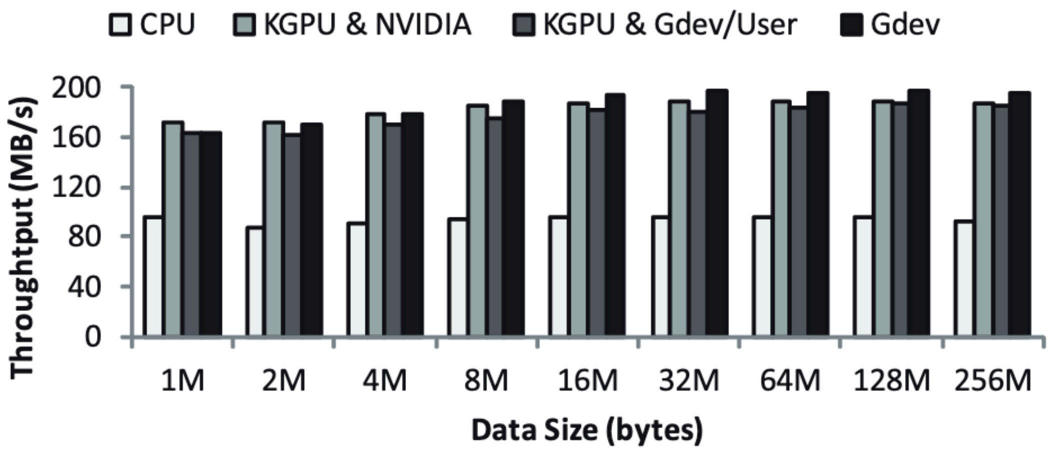


Figure 9: eCryptfs write throughput.

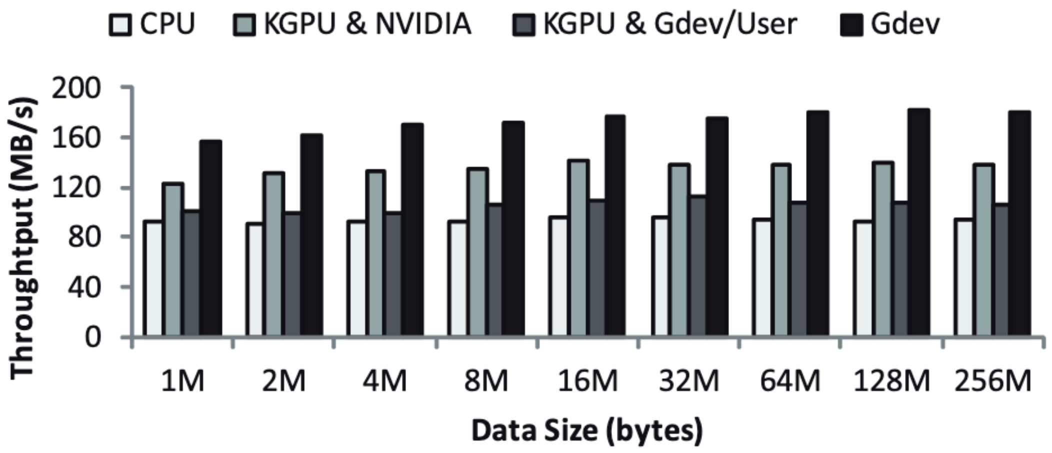


Figure 10: eCryptfs write throughput with priorities.

Compared against CPU-only, KGPU(user-space daemon), Gdev

- GPU acceleration provides **large throughput gains**
- Gdev performs similarly to KGPU in single-task scenarios

- In multi-task scenarios:
  - Gdev avoids priority inversion
  - High-priority OS tasks retain GPU performance