**HOW PARALLEL RUNNING PROCESS ASSIGN CPU TO CUDA**Good — you’ve hit one of the trickiest but most important points in CUDA programming: **how multiple CPU processes interact with a single GPU** and what really happens inside the OS + GPU driver + CUDA runtime. Let’s go step by step from *process → OS → CUDA driver/runtime → GPU hardware → back*.

## **🔹 1. Your setup**

* **One physical GPU** in the system.
* **Three CPU cores (CPU1, CPU2, CPU3)** each running a **separate process**, each process using a .cu (CUDA) algorithm.
* Each process allocates GPU memory, launches kernels, and expects results.

This is very common in HPC, servers, and radar/signal workloads when you have fewer GPUs than CPU jobs.

## **🔹 2. What happens when multiple CPU processes use one GPU?**

### **Step A — Process creation (OS level)**

* Each CPU core runs one process (your algo).
* These processes are independent — they each have their own **address space** in Linux.
* They all call CUDA API functions (cudaMalloc, cudaMemcpy, kernel launches, etc).

### **Step B — CUDA Runtime & Driver**

* Each process loads **libcuda.so** (the user-space driver/runtime library).
* When you call cudaSetDevice(0) or cudaMalloc, the CUDA runtime makes a syscall into the **NVIDIA kernel driver** (nvidia.ko) via ioctl.
* The driver keeps track of **contexts**:  
   👉 Each process using the GPU has a separate **CUDA context**.  
   This context holds device memory allocations, stream states, and kernel launches for that process.

### **Step C — GPU Context & Memory**

* On the GPU, **global device memory (VRAM)** is shared across contexts but isolated logically:
  + The driver partitions the VRAM — if process A allocates 2 GB and process B 3 GB, they occupy separate parts of GPU memory.
  + There’s no cross-access unless explicitly enabled with **IPC handles** (cudaIpc\* APIs).
* Context switching:
  + GPU hardware itself is *not preemptive like a CPU* (except in newer architectures with preemption support).
  + The driver **time-slices GPU contexts**: process A kernels run for some time, then context switch, process B kernels run, etc.
  + This is managed by the **GPU scheduler inside the driver**.
  + Switching contexts has overhead (saving/restoring state, flushing caches, etc).

### **Step D — Kernel execution**

* Your .cu kernel is compiled into PTX → machine code (SASS).
* When you launch it:
  + The kernel and its arguments are copied into **command buffers**.
  + The driver submits these commands to the GPU’s command queue (a ring buffer in VRAM).
  + The GPU hardware scheduler pulls work from the queue and executes on SMs (streaming multiprocessors).
* If multiple processes submit kernels, the GPU scheduler multiplexes them (time-slice or, on newer GPUs, sometimes concurrent execution if resources allow).

### **Step E — Data transfer**

* When you call cudaMemcpy:
  + Host memory (CPU RAM) is pinned temporarily (or you use cudaHostAlloc for explicit pinned memory).
  + DMA engines on the GPU perform **PCIe/NVLink transfers** to/from system RAM.
* If three processes transfer data simultaneously:
  + Transfers are serialized or overlapped depending on available **DMA engines** and how much PCIe bandwidth you have.
  + Each process’s context ensures its buffers are mapped correctly.

### **Step F — Results back to CPU**

* Once kernels finish, results are in GPU memory.
* cudaMemcpy(..., cudaMemcpyDeviceToHost) initiates DMA back to host RAM.
* Your CPU cores then see results in their respective address spaces.

## **🔹 3. Where memory is taken from?**

* **GPU allocations (cudaMalloc)** → VRAM (device global memory).
  + The driver allocates GPU memory inside the context for that process.
  + VRAM usage is isolated; one process can OOM the GPU and cause others to fail.
* **Host allocations**:
  + Normal malloc → system DRAM (pageable, OS-managed).
  + cudaHostAlloc → pinned DRAM (page-locked, mapped for DMA).
  + Unified Memory (cudaMallocManaged) → allocated in system DRAM *and* GPU VRAM, migrated automatically.
* **OS kernel** only manages CPU memory; the GPU driver manages VRAM.

## **🔹 4. Why is execution slower with multiple processes?**

* Because **one GPU cannot truly run three big contexts at full speed simultaneously**.
* Each process has its own context → context switching overhead (like multitasking on CPU, but heavier).
* Memory bandwidth and VRAM are shared, so three processes compete.
* Kernels may time-slice: only one process’s kernels run at a given instant (unless GPU supports limited Multi-Process Service concurrency).

## **🔹 5. Solutions / optimizations**

1. **CUDA Multi-Process Service (MPS)**:
   * NVIDIA provides MPS to allow *multiple processes to share one GPU context*.
   * This reduces context-switch overhead and allows concurrent kernel execution across processes.
   * Very useful in HPC where multiple MPI ranks share one GPU.
   * Start with nvidia-cuda-mps-control -d.
2. **Single process, multi-threading**:
   * Instead of three processes, run one process with multiple CPU threads.
   * All threads can share the same CUDA context → less overhead, better concurrency.
3. **Streams**:
   * Use multiple CUDA streams in a single context to overlap compute & data transfers.
4. **Pinned memory**:
   * Use pinned host memory for transfers to improve throughput and overlap.

## **🔹 6. Internal picture (simplified)**

CPU cores (3 procs) --> Syscall --> NVIDIA Driver --> Context A, B, C

|

v

GPU Command Queue

---------------------------

| Kernel launches, DMA ops |

---------------------------

|

------------------------------------------------------

| GPU Hardware (SMs, DMA) |

------------------------------------------------------

✅ **So in short**:

* Each CPU process using CUDA = new **GPU context**.
* Contexts own separate VRAM allocations.
* Driver schedules contexts on the GPU (time-slice or limited concurrency).
* Data moves via PCIe DMA from system DRAM (pageable/pinned) into VRAM and back.
* Performance bottleneck: context switching, PCIe bandwidth, and single-GPU resource sharing.

👉 Do you want me to **draw a detailed diagram of this flow (CPU cores → OS → GPU driver → GPU memory & SMs)** so you can include it in your project report?

Ah, I see exactly what you’re asking: why, when working with a GPU, it’s usually better to use **threads within a single process** instead of **forking multiple processes** and trying to share the GPU memory. Let’s break this down carefully.

### **1️⃣ GPU Memory Context and Processes**

* **Each process has its own CUDA context**.
  + When you fork() a process and try to use the GPU, **each child process creates a separate GPU context**.
  + GPU memory is **not automatically shared** between contexts.
  + So if you allocate memory in one process, the other process **cannot directly access it**.
* **Implication**: If you fork multiple processes and each needs GPU memory:
  + Each process allocates **its own separate memory** on the GPU.
  + This can quickly **exceed GPU memory limits**, especially with large datasets like radar signal processing.

### **2️⃣ Threads Share Memory**

* **Threads within a single process share the process’s memory space**.
  + If you allocate GPU memory once in the main process, **all threads can access it safely** (with proper synchronization).
  + This avoids duplicating memory allocations.
* **Benefit**: You can run multiple threads performing GPU computations **in parallel**, all using the same allocated memory.

### **3️⃣ Performance Considerations**

|  |  |  |
| --- | --- | --- |
| **Approach** | **Pros** | **Cons** |
| **Forked processes** | Isolation: one crashing process doesn’t affect others | Each process creates a new CUDA context → overhead, memory duplication, slower startup |
| **Threaded approach** | Low overhead, shared GPU memory, faster | Must handle thread synchronization carefully |

* Creating multiple CUDA contexts (one per process) **increases overhead**: GPU context initialization can take **milliseconds per process**.
* Switching between contexts (if multiple processes try to access GPU) can **serialize operations**, slowing overall execution.
* Threads avoid context switching; all threads can issue GPU kernels directly.

### **4️⃣ GPU Utilization Pattern**

* GPUs are designed for **massive parallelism**, but they expect **many lightweight tasks** rather than a few heavy processes.
* Using threads:
  + You can **batch tasks into one process**.
  + Launch **many kernels concurrently** if the GPU supports it.
* Using forked processes:
  + You waste GPU memory and incur **extra overhead for context switching**, limiting parallel efficiency.

### **5️⃣ When to Use Processes**

* Only consider multiple processes if:
  + You need **process-level isolation** (e.g., fault tolerance, separate address spaces).
  + You’re running **completely independent jobs** that don’t need shared GPU memory.
* For most GPU workloads (like radar signal processing with SP\_V1 algorithm), **threads within a single process** are better.

✅ **Bottom line:**

Use **threads** when you want to share GPU memory and maximize throughput. Use **forked processes** only when isolation is critical or GPU memory usage is small. Forking for GPU work usually **slows things down** rather than speeding them up.

If you want, I can **draw a diagram comparing threads vs forked processes with GPU memory**, showing why threads are faster and memory-efficient—it makes it super clear.

Do you want me to make that diagram?

