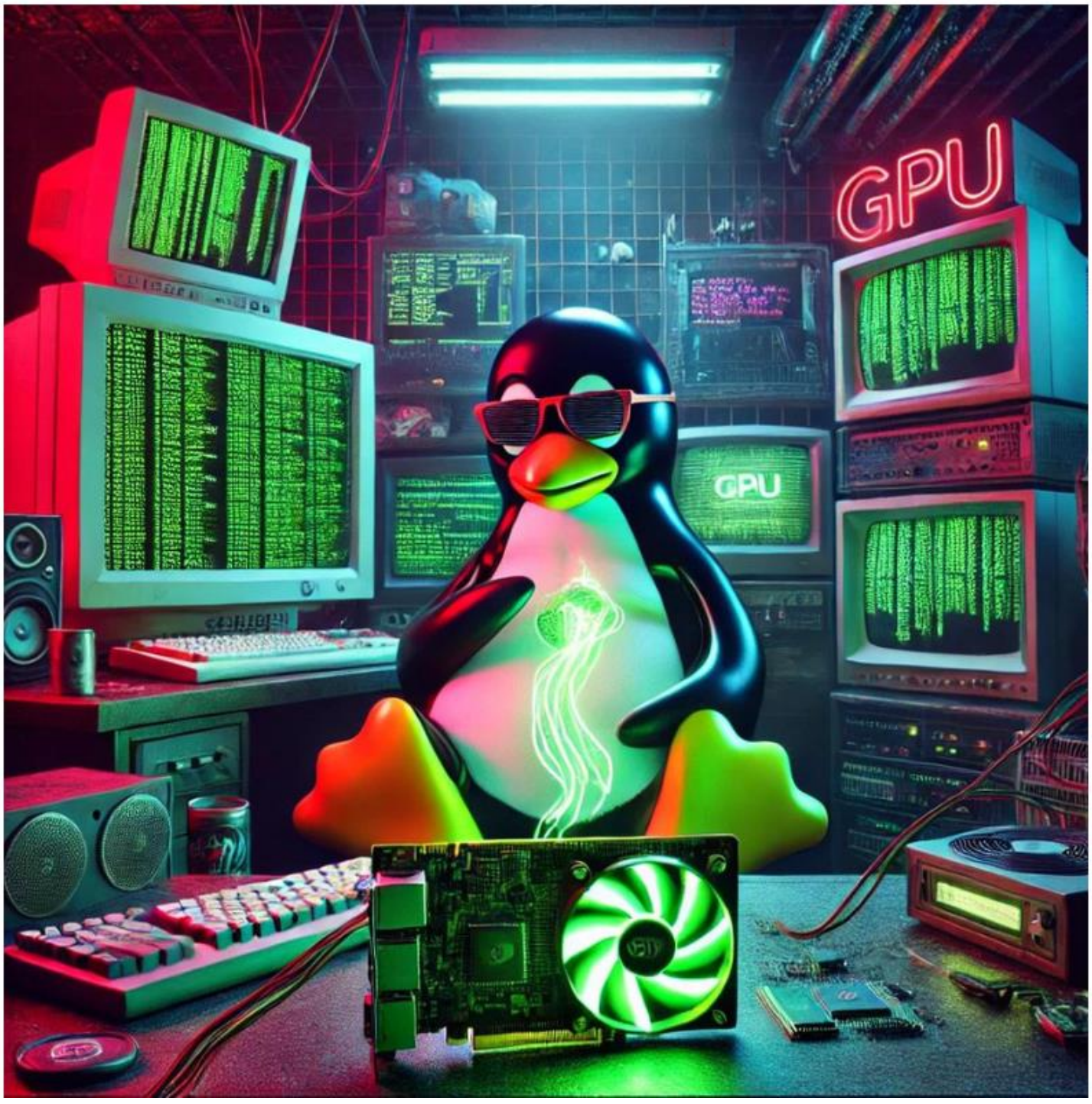
How do threads get scheduled when multiple threads are launched in a GPU!



In CUDA, thread scheduling is handled by the GPU's hardware and is influenced by the structure of threads, blocks, and warps.

1. Launch of a CUDA Kernel

When a CUDA kernel is launched, the following steps occur:

- Grid and Block Creation: The kernel launches a grid of blocks, where each block contains multiple threads. These threads are organized into warps of 32 threads each.
    - If you have 1024 threads in a block, this results in 1024 / 32 = 32 warps.
- Each block of threads is assigned to a streaming multiprocessor (SM) for execution. A warp consists of 32 threads, and this is the basic unit of execution on the GPU. All 32 threads in a warp execute the same instruction at any given time (SIMT - Single Instruction, Multiple Threads).

2. How the GPU Schedules Warps

Warp Scheduler and the SM

- Each SM (Streaming Multiprocessor) has its own warp scheduler that decides which warp to execute at any given moment.
- Warp Scheduling is based on availability of resources and the readiness of the warp to execute. A warp becomes ready to execute when it has all necessary data in place (e.g., data from registers or shared memory).
- Multiple warps from different thread blocks can be assigned to an SM to maximize resource utilization.

The GPU's warp scheduler continuously monitors the status of each warp. The primary reasons a warp may stall and require the scheduler to switch to another are:

- Memory access latencies: If a warp issues a memory request (e.g., accessing global memory), the GPU may have to wait hundreds of clock cycles for the data to be fetched. During this wait, the warp is stalled, and the warp scheduler selects another warp to execute.
- Execution dependencies: Some warps might be waiting for the result of a previous instruction (e.g., waiting for a value produced by an earlier computation), so they cannot proceed until the result is available.
- Shared resources: A warp might be stalled if it is waiting for access to shared memory or registers that are being used by another warp.

The warp scheduler ensures that warps are dynamically switched out whenever they are stalled, ensuring the SM's computational units stay busy.

3. Warp Scheduling: Dynamic and Non-preemptive

Unlike traditional CPU scheduling, where individual threads may be preempted and swapped out, CUDA does not employ preemptive multitasking at the thread level. Instead:

Once a warp is selected for execution, it runs to completion or until it stalls (due to memory or execution dependencies).

The warp scheduler switches to other ready warps when one is stalled.

Example: If Warp 1 issues a global memory access request, it might need to wait for 400 cycles for the data to be available. During these 400 cycles, Warp 2, Warp 3, and others may be executed, filling the gap and ensuring the GPU's cores don't sit idle.

4. Occupancy and Efficient Scheduling

Occupancy is a key factor in determining the efficiency of warp scheduling. Occupancy refers to the ratio of active warps to the maximum number of warps that an SM can handle. High occupancy means more warps are available for execution, which allows the warp scheduler to better hide memory latencies by switching between warps.

Factors affecting occupancy:

- Number of registers per thread: The more registers each thread requires, the fewer warps can be loaded onto an SM.
- Shared memory per block: If a kernel requires a large amount of shared memory, this can limit the number of blocks (and thus the number of warps) that can run on an SM.
- Number of threads per block: Larger blocks (e.g., 1024 threads per block) result in more warps, but the hardware resources (registers, shared memory) must be divided among all threads. This can limit the overall number of blocks an SM can handle.

When an SM has higher occupancy, the warp scheduler has more warps to choose from. If one warp is stalled due to a memory dependency, another warp that's ready to execute can be selected, thereby hiding the memory latency and maximizing the usage of the computational units (ALUs).

The GPU architecture is designed to hide memory latency. When one warp is stalled due to memory access, the warp scheduler switches to another warp that is ready to execute. This context switching between warps is extremely efficient because:

All warps share the same execution context (no need to save or restore registers, as is typical in CPU thread context switching).

The GPU does not waste time, as another warp can be executed immediately while waiting for memory to be fetched.

In the SIMT model, all 32 threads in a warp execute the same instruction at the same time, but they might operate on different data. This works well when the threads are following the same control flow, but if threads

diverge (e.g., due to an if statement where some threads take one branch and others take another), the warp is split, and the scheduler handles these divergent branches one at a time (reducing efficiency).

As a conclusion,

When a CUDA kernel is launched:

- Threads are grouped into blocks, and each block is divided into warps of 32 threads.
- Warps are assigned to SMs for execution. Multiple warps can be executed concurrently on each SM.
- The warp scheduler on each SM manages which warps are active, switching between warps when one is stalled due to memory or execution dependencies.
- When one warp is waiting on data (e.g., memory access), the scheduler switches to another warp that is ready to execute, keeping the GPU's cores busy.
- Higher occupancy (more warps per SM) allows for better hiding of memory latencies, as more warps are available for scheduling.
- The SIMT execution model ensures that all threads in a warp execute the same instruction, but divergence can reduce performance, as the scheduler has to handle each divergent path separately.