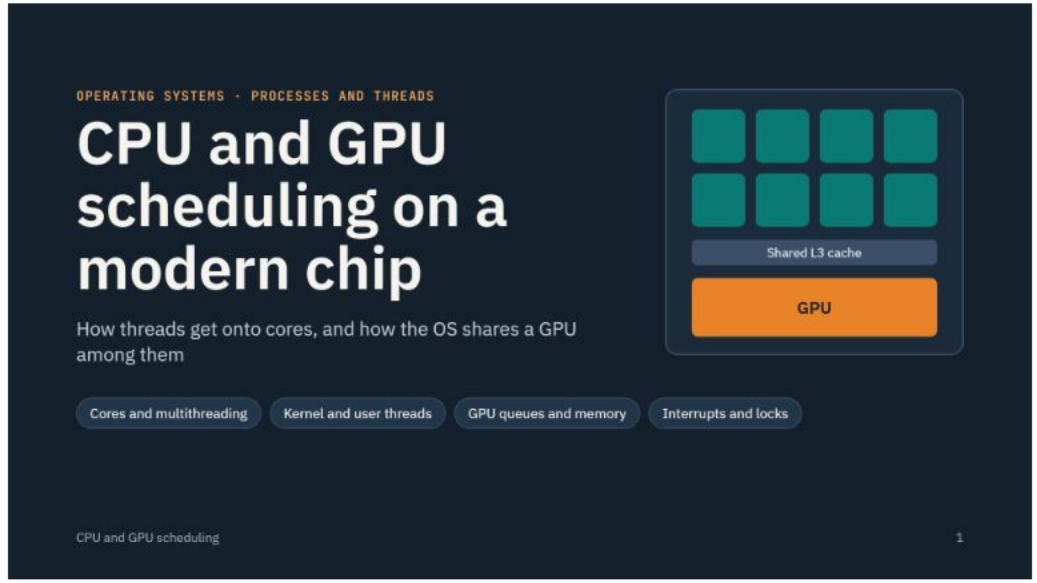
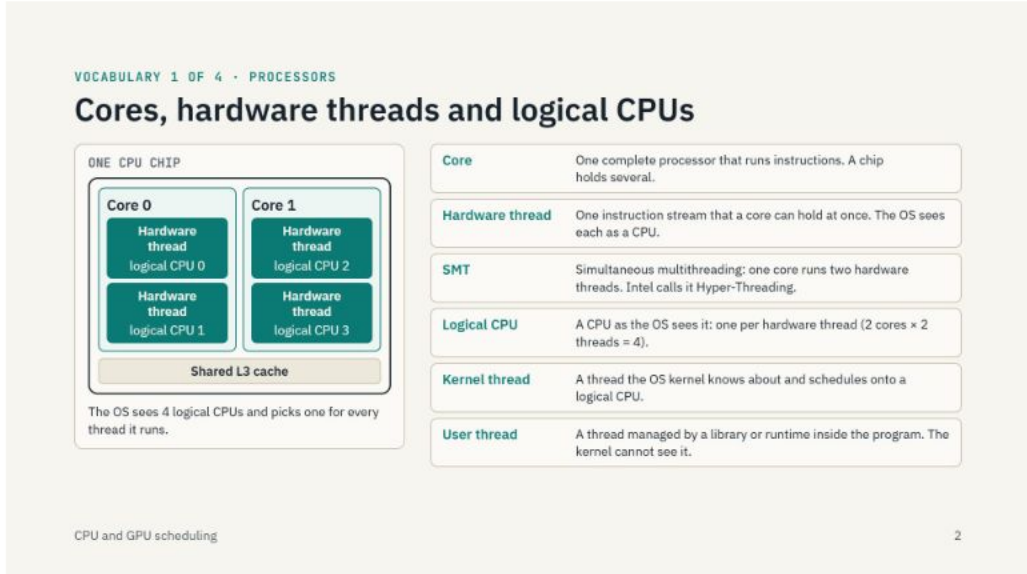
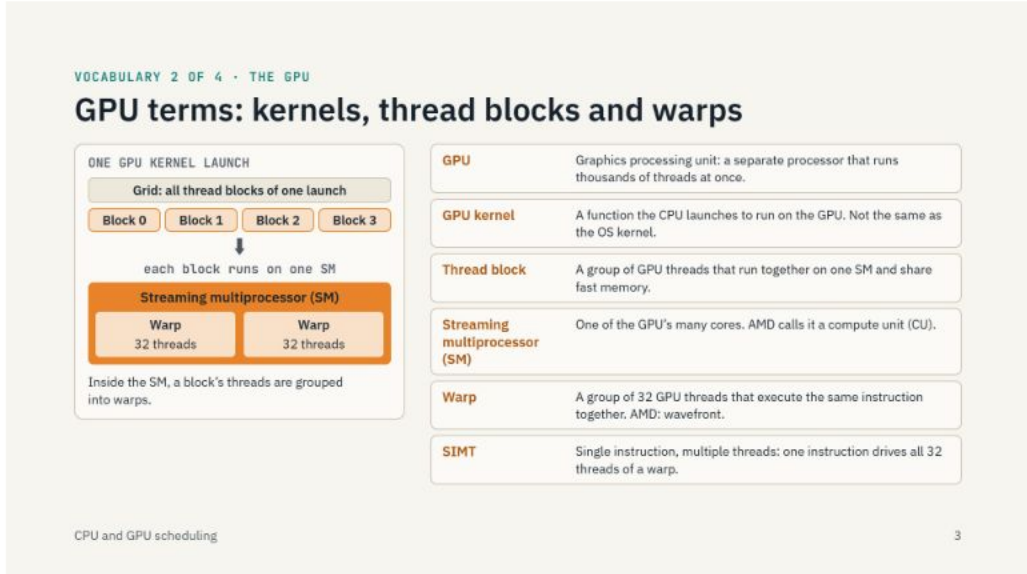


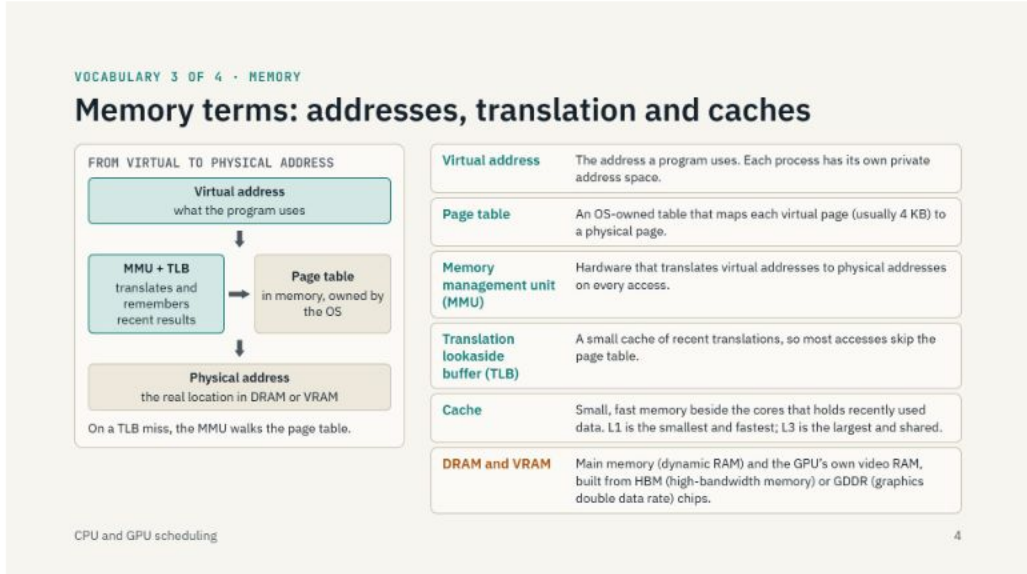
Opening slide. This lecture builds on heavyweight processes, threads, multi-threading and basic multi-core management. Today the question is what happens underneath: how a thread actually gets onto a hardware thread, and how a GPU, which is a separate computer with its own scheduler and memory, is shared among processes. The graphic on the right is a toy chip: 8 cores, a shared L3, and a GPU block.



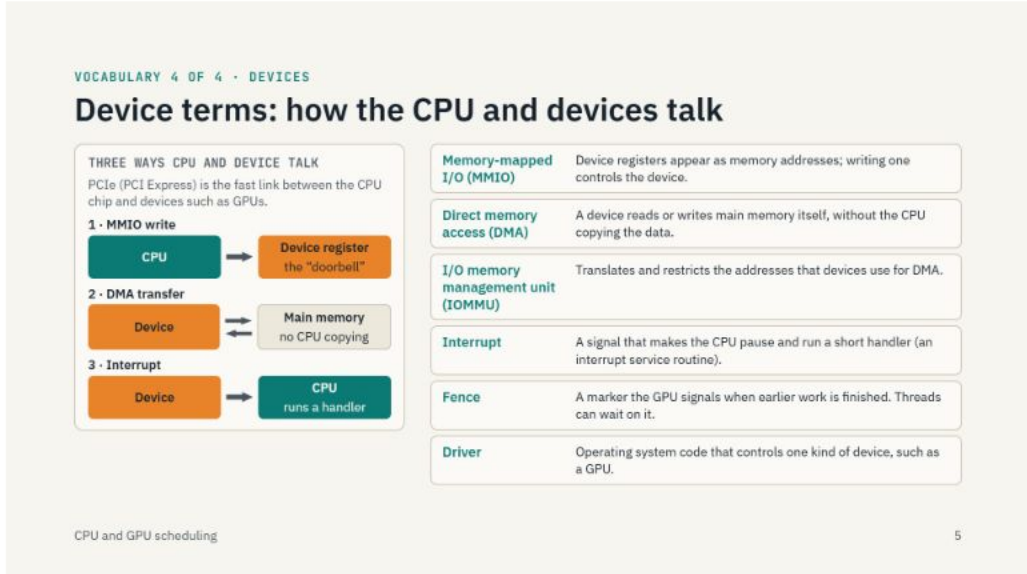
Vocabulary slide 1 of 4. Everything later in the lecture uses these words, so spend a minute here. A core is a full processor. With simultaneous multithreading (SMT; Intel calls it Hyper-Threading) a core keeps two instruction streams ready and can switch or mix between them, so the OS sees two logical CPUs per core. Note the layers: hardware threads and logical CPUs are hardware words; kernel threads and user threads are software words. The OS scheduler places kernel threads onto logical CPUs.



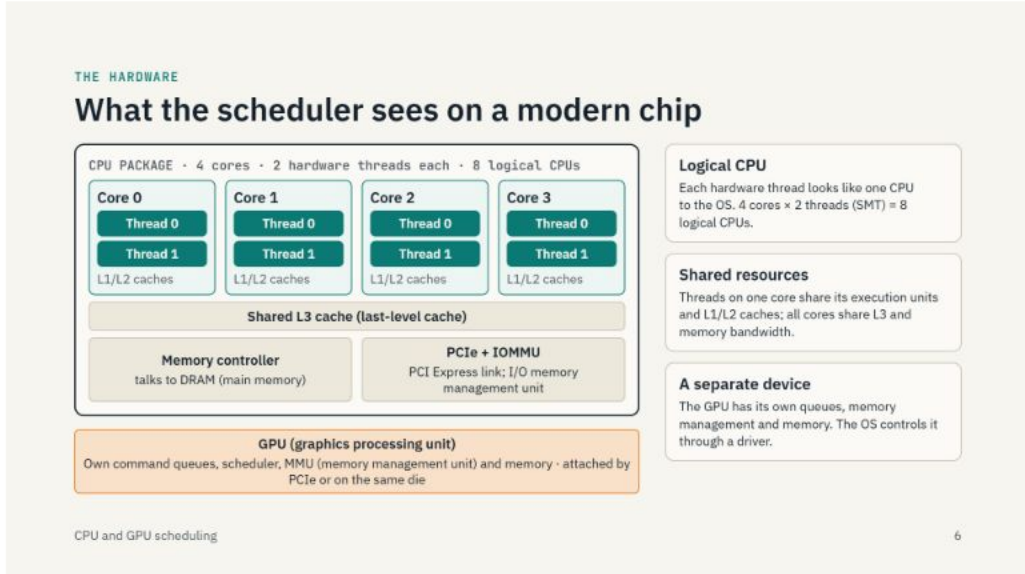
Vocabulary slide 2 of 4. Warn students about the word kernel: on the CPU side it means the operating system kernel; on the GPU side a kernel is a function that the CPU asks the GPU to run. A launch creates a grid of thread blocks. Each block is placed on one streaming multiprocessor (SM; AMD calls it a compute unit, CU) and stays there until it finishes. Inside the SM, threads execute in warps of 32 that share one instruction stream (single instruction, multiple threads, SIMT). AMD uses the word wavefront for a warp. Orange is used for GPU terms throughout the deck.



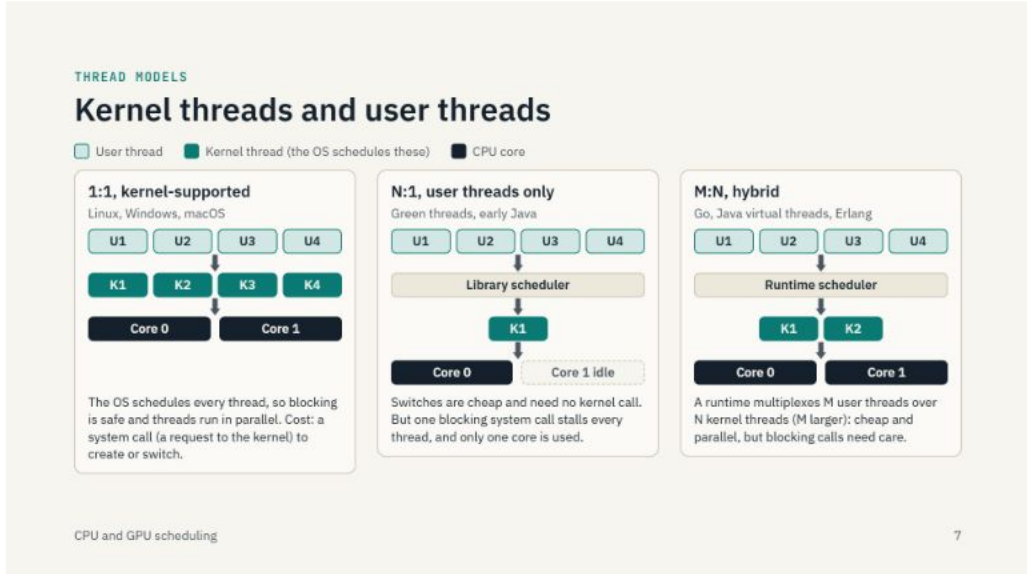
Vocabulary slide 3 of 4. The figure is the path every memory access takes: the program uses a virtual address, the memory management unit (MMU) translates it, and the translation comes from the page table, which lives in memory and is owned by the OS. Because walking the page table is slow, the MMU keeps recent results in the translation lookaside buffer (TLB). Caches are a different idea: they hold copies of data, not translations. A GPU repeats the whole picture: it has its own MMU, TLB and page tables, and its own memory (VRAM), built from HBM or GDDR chips.



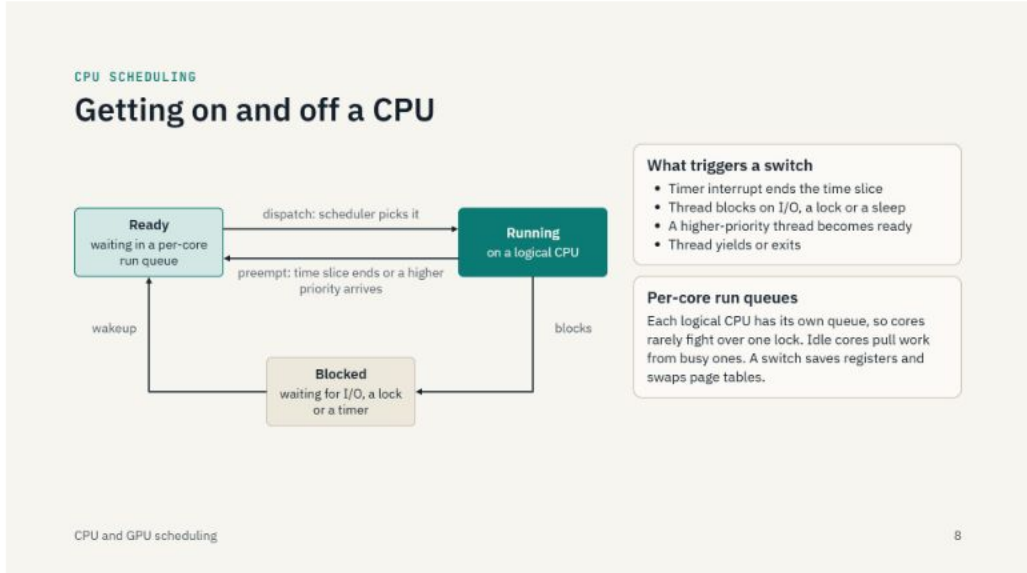
Vocabulary slide 4 of 4. A device such as a GPU is not part of the CPU, so the two need ways to communicate. The CPU talks to the device by writing to special addresses (memory-mapped I/O, MMIO); the GPU's doorbell is one such register. The device moves bulk data by itself using direct memory access (DMA), and the I/O memory management unit (IOMMU) translates and limits the addresses it may use. When the device finishes, it signals the CPU with an interrupt. A fence is the software-visible marker for that completion. All of these connect over a link called PCIe (PCI Express), unless the GPU is on the same chip.



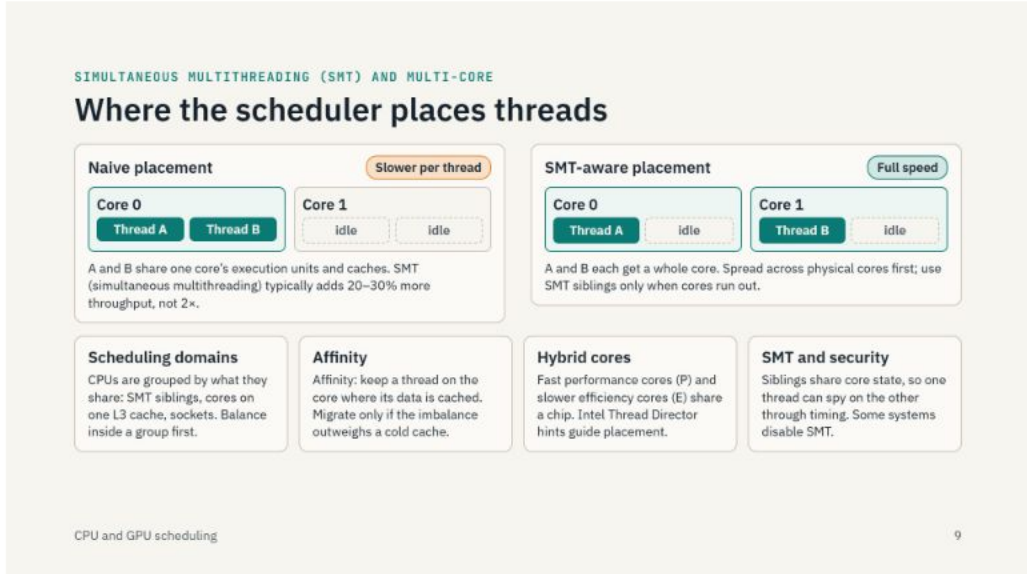
The scheduler does not see "a chip". It sees a list of logical CPUs. With simultaneous multithreading (SMT; Intel calls it Hyper-Threading) each physical core exposes two hardware threads that share execution units, L1 and L2 caches. All cores share the L3 cache and the memory controller. The GPU is drawn separately on purpose: it is a different processor with its own scheduler, page tables and memory. It may be a discrete card across PCI Express (PCIe) or an integrated GPU on the same die, but the OS still treats it as a device managed by a driver. Terms are defined on the four vocabulary slides.



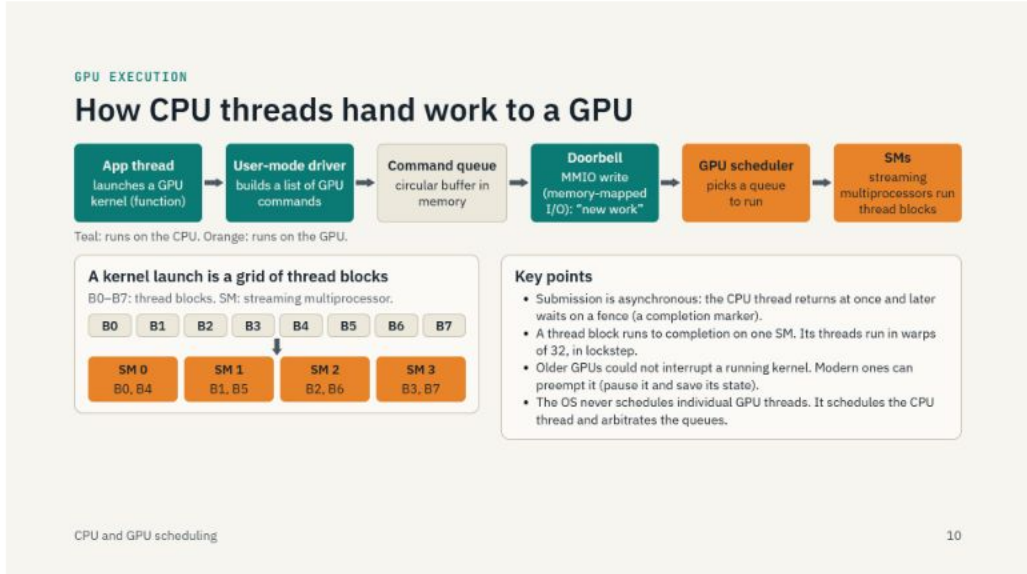
Three ways to map user-visible threads to the kernel's schedulable entities. In 1:1 every thread is a kernel thread, so the OS scheduler sees it, can run it on any core, and can block it independently. In N:1 a user-level library does the switching; the kernel sees one thread, so one blocking system call blocks everything and only one core is used. M:N runtimes such as Go's scheduler multiplex many lightweight threads over a small pool of kernel threads; the hard part is what to do when a user thread blocks inside the kernel (Go hands the kernel thread's queue to another thread). The unit the OS schedules onto a CPU is always a kernel thread.



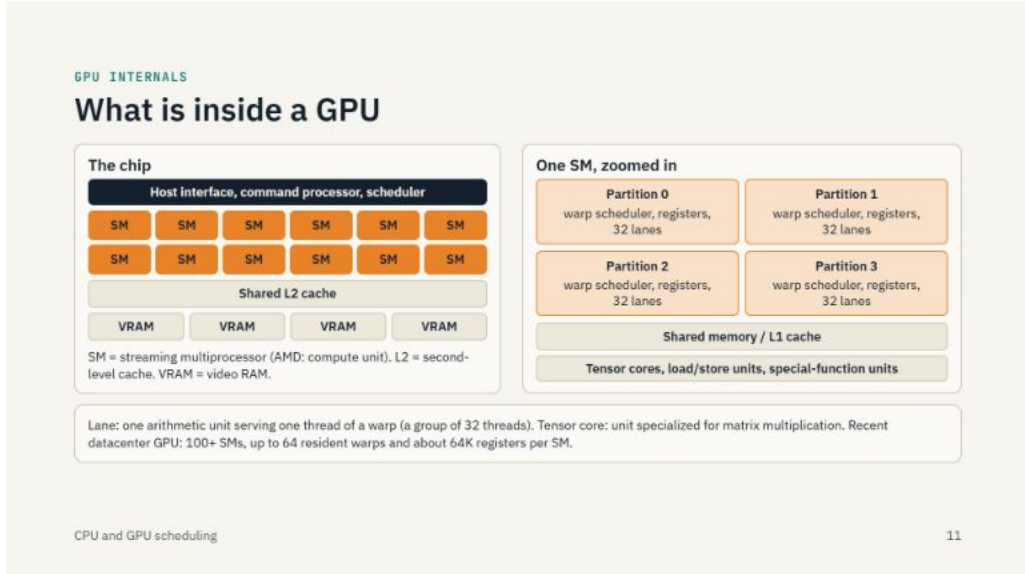
This is the state machine students already know, now tied to hardware. A thread is Ready when it sits in a run queue, Running when a logical CPU executes it, Blocked when it waits for an event. The scheduler runs on a core in three situations: when the running thread blocks or exits, when a timer interrupt fires and the time slice is used up, and when a wakeup makes a higher-priority thread ready. Modern kernels keep one run queue per logical CPU to avoid a global lock, and a load balancer migrates threads between queues. A context switch saves registers, switches the page-table base register, and often leaves caches and TLB entries cold for the next thread.



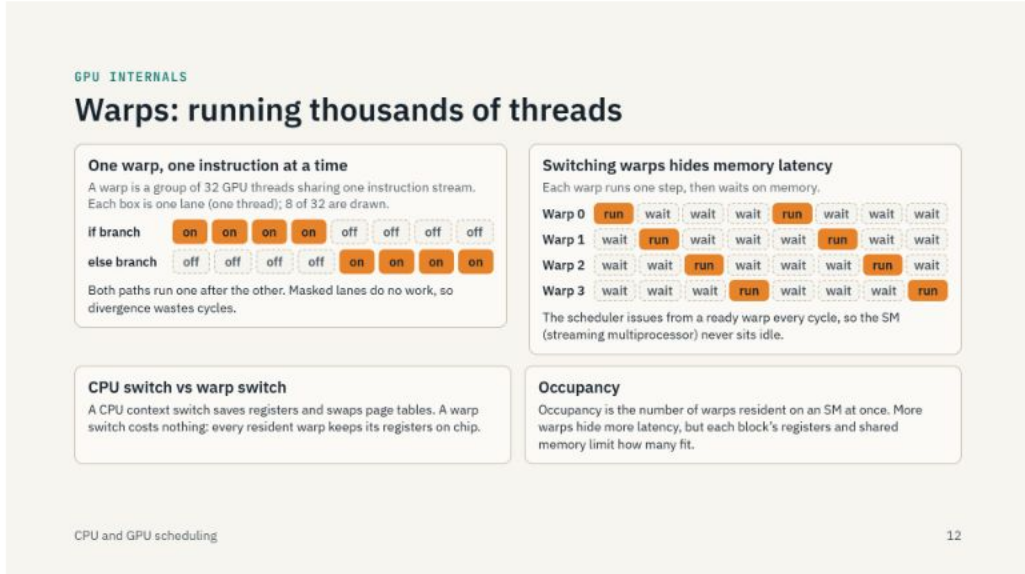
Multi-core and SMT look similar to the scheduler (both give more logical CPUs) but they are not equal. Two threads on SMT siblings compete for the same execution units, L1 and L2, so each runs slower than it would alone; the typical total gain is on the order of 20 to 30 percent, not double. A good scheduler therefore fills one thread per physical core first. Linux expresses this with scheduling domains (SMT, cache-sharing cores, package, NUMA node); load balancing happens within a domain before crossing it. Affinity keeps a thread near its warm caches. On hybrid chips (P-cores and E-cores, or big.LITTLE on ARM) the scheduler needs per-core capacity information, and Intel's Thread Director supplies hardware hints. Because SMT siblings share microarchitectural state, some deployments disable SMT or use core scheduling so that only mutually trusting threads share a core.



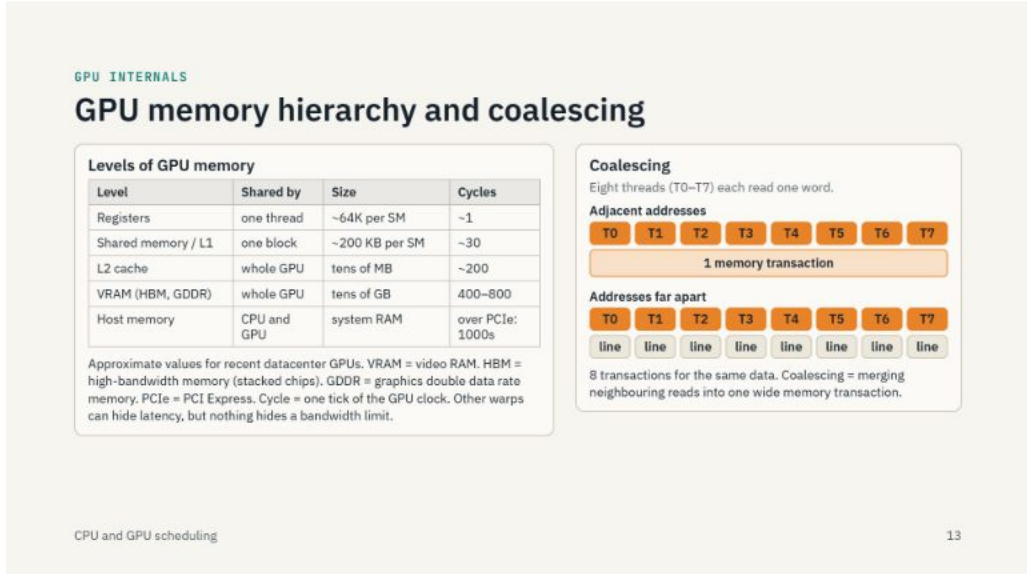
The path from a CPU thread to GPU execution. The application calls a runtime (CUDA, Vulkan, OpenCL); the user-mode driver encodes the request in a command buffer and appends it to a queue that lives in memory shared with the GPU. A write to a doorbell register (a memory-mapped I/O, MMIO, write) tells the GPU there is new work, with no system call needed on modern drivers. The GPU's own front end and scheduler pick a queue and split a kernel into thread blocks that are placed on streaming multiprocessors (SMs; AMD calls them compute units, CUs). Blocks do not migrate once started. Because submission is asynchronous the CPU thread keeps running, and synchronizes later with a fence or event. Preemption on GPUs was historically coarse or absent; recent hardware can save a running kernel's state at thread-block or instruction granularity. The OS scheduler never sees individual GPU threads.



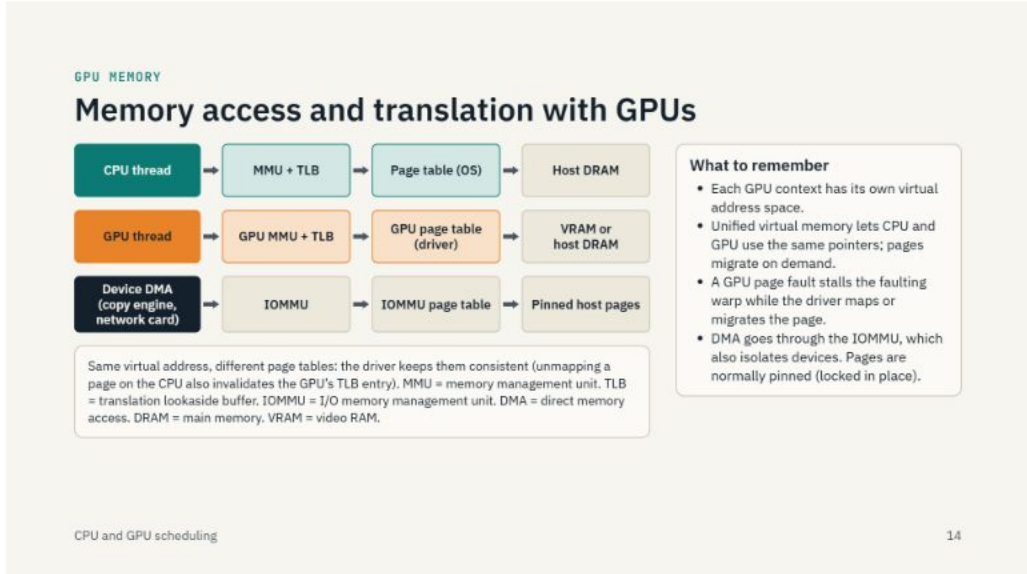
A GPU is a throughput machine built from many streaming multiprocessors (SMs; AMD calls them compute units). The front end (host interface, command processor and scheduler) reads queues and hands work to SMs. All SMs share one L2 cache and a set of memory channels to VRAM (HBM or GDDR). Inside an SM there are typically four partitions, each with its own warp scheduler, a slice of the register file and a set of 32 execution lanes, plus shared memory or L1 and specialized units such as tensor cores. The numbers are for recent NVIDIA datacenter parts (for example 132 SMs, 64 resident warps and 64K 32-bit registers per SM on an H100); other vendors and models differ, so treat them as typical rather than universal.



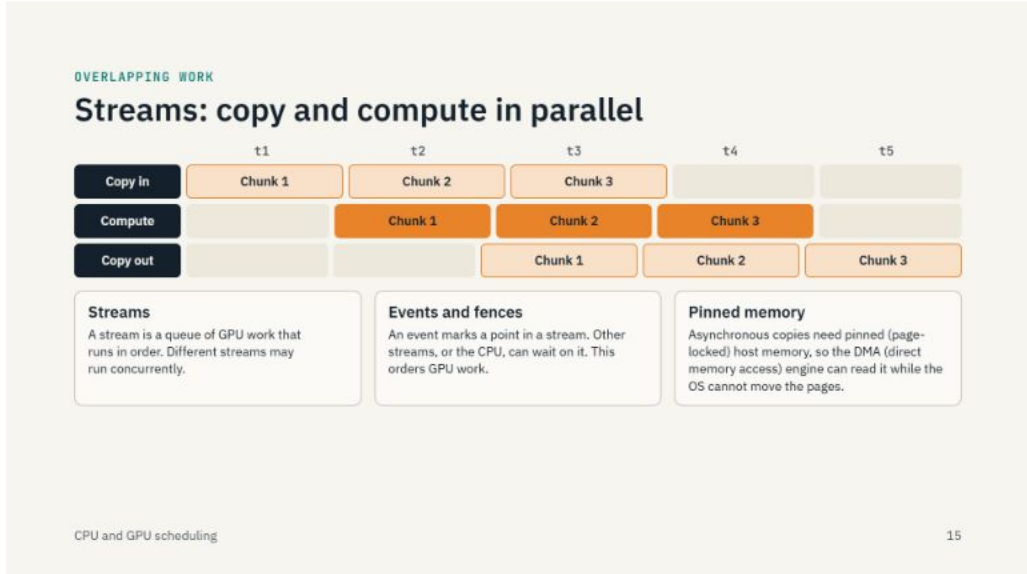
The unit of execution on an NVIDIA GPU is the warp: 32 threads that share one instruction stream (SIMT). When threads in a warp take different sides of a branch, the hardware runs each path in turn with the other lanes masked off, so divergence costs throughput. The key trick is latency hiding. A memory access takes hundreds of cycles, so instead of waiting, the SM's warp scheduler issues from another ready warp. That switch is free because all resident warps keep their registers in the SM's large register file; nothing has to be saved or restored. The diagram simplifies to one run slot then three wait slots per warp, so four warps keep the SM fully busy. AMD wavefronts are similar (32 or 64 lanes). Contrast this with the CPU context switch from earlier slides, which costs thousands of cycles. The price is that registers and shared memory are carved up among resident warps, which limits occupancy.



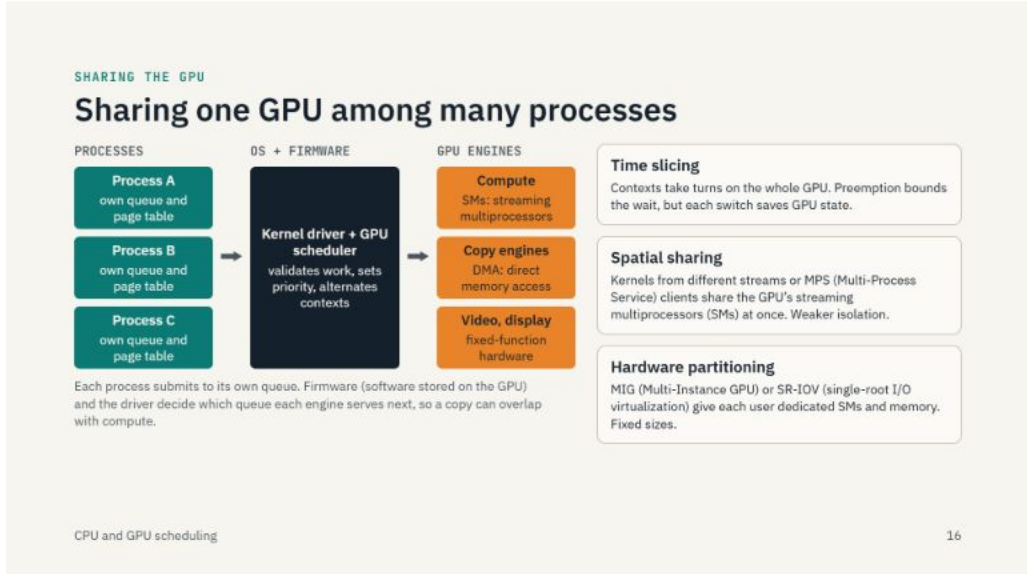
The GPU has a deeper explicit hierarchy than a CPU. Registers are private to a thread and enormous in total. Shared memory (or the L1 it shares space with) is on the SM and visible to one thread block; programmers manage it directly. L2 is shared by the whole chip, and VRAM (HBM or GDDR) is the main memory, with bandwidth of up to several TB/s on HBM parts. Host memory sits across PCIe or the on-die fabric and is orders of magnitude slower. The cycle counts are rough, commonly quoted figures that vary by architecture. Coalescing: when the 32 threads of a warp read neighbouring words, the hardware merges them into one or a few wide memory transactions; scattered addresses split into many transactions and waste bandwidth. For an OS course the point is that the GPU hides latency with parallelism, not with big caches and speculation like a CPU.



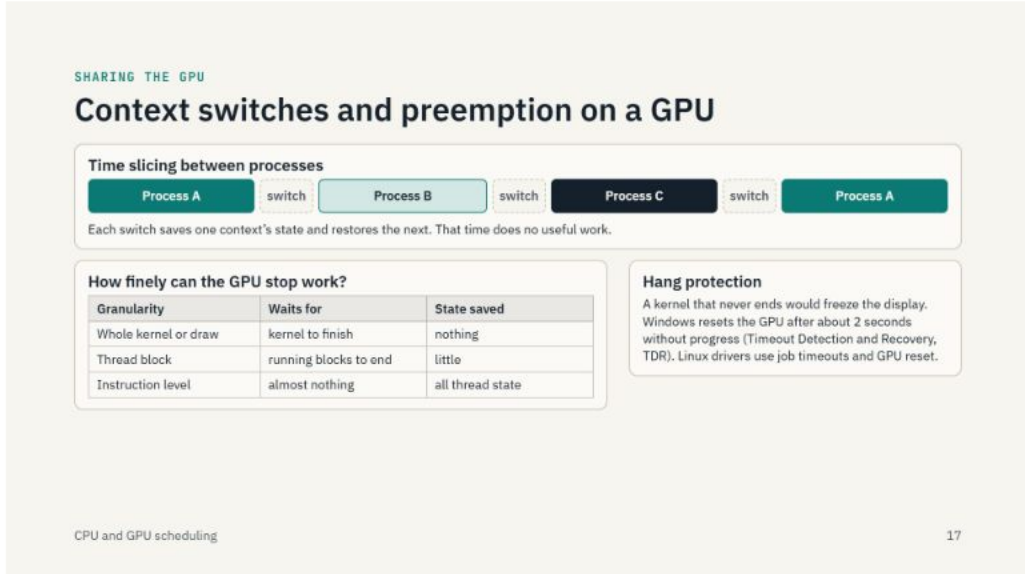
Three translation paths. A CPU thread goes through the core's MMU and TLB using page tables the OS owns. A GPU thread goes through the GPU's own MMU and TLB, using a per-context page table that the driver (or GPU firmware) maintains; its physical target can be on-card VRAM or host memory across PCIe. Device DMA, such as a copy engine or a network card, goes through the IOMMU (Intel VT-d, AMD-Vi, ARM SMMU), which both translates and restricts what a device can touch; DMA buffers are normally pinned so the pages cannot move mid-transfer. With unified virtual memory, CPU and GPU can share pointers: a GPU access to a non-resident page raises a replayable page fault, the faulting warp waits, and the driver migrates or maps the page. Because two page tables describe one address space, the OS and driver must keep them coherent, for example by invalidating GPU TLB entries when the CPU unmaps a page.



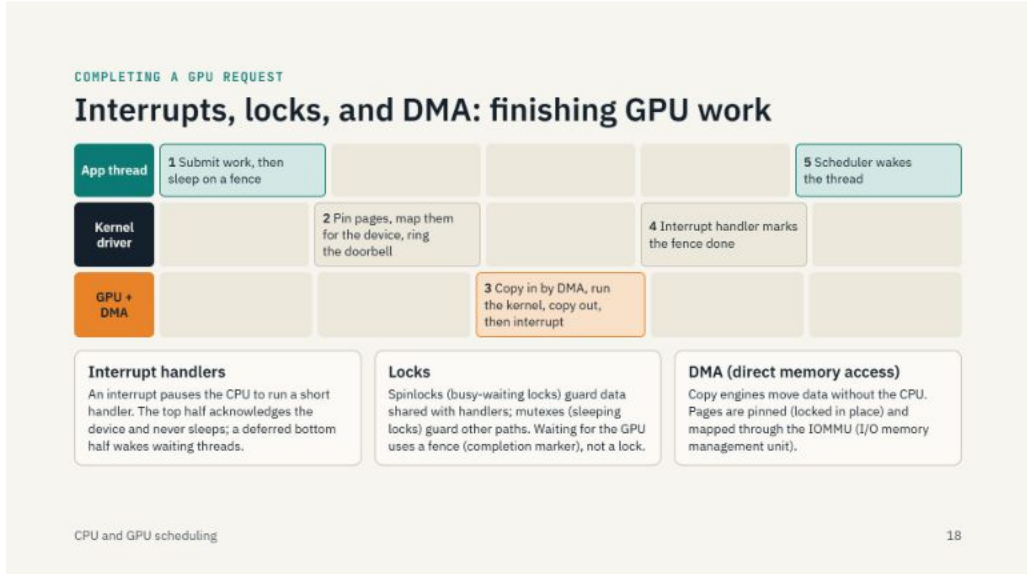
Because the GPU has independent engines (copy in, compute, copy out), a pipeline can keep all of them busy. Split the data into chunks and put them in different streams: while chunk 2 is being copied in, chunk 1 is computing, and so on. After the pipeline fills at t3, all three engines work at once. Streams give ordering inside and freedom between; events and fences express dependencies across streams and between the CPU and GPU. Asynchronous host-to-device copies need pinned (page-locked) host memory so that DMA can proceed without the OS paging or moving the buffer. Many GPUs have separate copy engines for each direction, which is what allows copy in and copy out to overlap. Stream priorities are also available so latency-sensitive work can jump ahead in the queue.



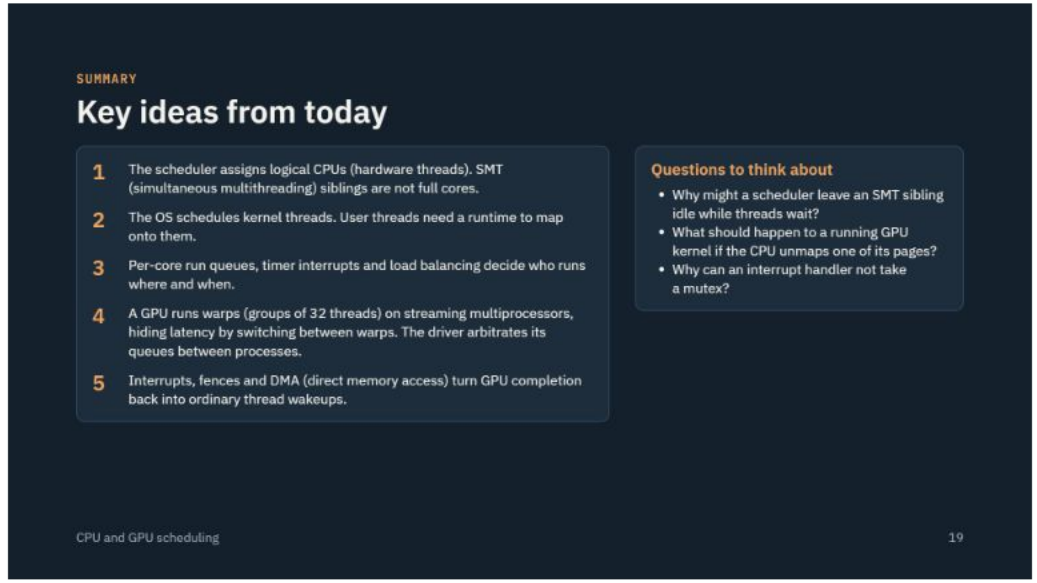
Each process that uses the GPU gets its own GPU context: a private virtual address space and its own queues, so one process cannot read another's memory. The kernel driver validates submissions and, together with GPU firmware and hardware, decides which context's queue gets which engine next. The GPU has separate engines: compute or graphics (the streaming multiprocessors, SMs), copy engines that do direct memory access (DMA), and video or display engines, so a copy for one process can overlap with a kernel from another. Three sharing styles: time slicing (contexts alternate on the whole GPU, with preemption bounding latency), spatial sharing (kernels from several streams, or from several processes through NVIDIA MPS, run concurrently on different SMs), and hardware partitioning (MIG on recent NVIDIA datacenter GPUs, or SR-IOV virtual functions) that gives each tenant dedicated SMs and memory.



Time slicing: the driver and firmware let each process's context run for a slice, then switch. Unlike a CPU switch, a GPU switch can mean saving a lot of on-chip state: a large GPU holds tens of megabytes of registers and shared memory. So the finer the preemption granularity, the lower the latency to stop a running task but the more state must be saved. Early GPUs could only switch at kernel or draw-call boundaries, so a long kernel blocked everyone. Later hardware preempts at thread-block granularity, and NVIDIA Pascal and newer, and AMD with its compute wave save and restore, can preempt at instruction level. Because a runaway kernel can freeze the desktop, operating systems add watchdogs: Windows Timeout Detection and Recovery resets the GPU after roughly two seconds by default, and Linux DRM drivers have job timeouts followed by GPU reset. This is the GPU analogue of the CPU's timer interrupt, but it is implemented by the driver and firmware rather than by a hardware tick.



Follow one request left to right. (1) The app thread submits work and waits on a fence; a waiting thread sleeps, so it leaves the CPU and the scheduler runs something else. (2) The driver takes a short lock to protect the queue, pins and maps the buffers (through the IOMMU), writes the command, and rings the doorbell. (3) The GPU's copy engine DMAs input in, the kernel runs, and results are DMAed out; DMA moves data without CPU involvement. When finished, the GPU signals the fence and raises an interrupt (usually MSI-X). (4) The interrupt handler runs with interrupts masked, so the top half only acknowledges the device and schedules deferred work; a bottom half or threaded handler marks the fence done. Interrupt context cannot sleep, so it cannot take a mutex; spinlocks are used for data shared with handlers. (5) Marking the fence done makes the sleeping thread runnable again, and the scheduler eventually dispatches it.



Wrap-up. The thread through the whole lecture: the OS schedules CPU threads onto logical CPUs using per-core queues and balancing, and it treats the GPU as a device with its own scheduler and memory management that it shares through a driver. The three questions work as a short discussion or as exam-style review. Suggested answers: (1) to avoid slowing two threads that would otherwise each have a whole core, or for security; (2) the GPU's page table and TLB must be updated and the GPU may fault and retry, which is why drivers keep the two in sync; (3) interrupt handlers cannot sleep, and a mutex may sleep.