GPU

**Warp-swapping** is the major latency-hiding mechanism used by all GPUs.

Say we have two thousand threads to be executed. **Warps** on NVIDIA GPUs contain 32 **threads**. This yields 2000/32 = 62.5 warps, which means that 63 warps are allocated, one warp being half empty. The shader program is executed in lock-step on all 32 processors.

When a memory fetch is encountered, all threads encounter it at the same time, because the same instruction is executed for all. The fetch signals that this warp of threads will stall, all waiting for their (different) results. Instead of stalling, the warp is swapped out for a different warp of 32 threads, which is then executed by the 32 cores. This swapping is just as fast as with our single processor system, as no data within each thread is touched when a warp is swapped in or out. Each thread has its own registers, and each warp keeps track of which instruction it is executing.Swapping in a new warp is just a matter of pointing the set of cores at a different set of threads to execute; there is no other overhead. Warps execute or swap out until all are completed.

The more registers needed by the shader program associated with each thread, the fewer threads, and thus the fewer warps, can be resident in the GPU.

Warps that are resident are said to be “in flight,” and this number is called the **occupancy**. High occupancy means that there are many warps available for processing, so that idle processors are less likely. Low occupancy will often lead to poor performance.

Another factor affecting overall efficiency is **dynamic branching**, caused by “if” statements and loops. Say an “if” statement is encountered in a shader program. If all the threads evaluate and take the same branch, the warp can continue without any concern about the other branch. However, if some threads, or even one thread, take the alternate path, then the warp must execute both branches, throwing away the results not needed by each particular thread [530, 945].

**Static flow control** branches are based on the values of uniform inputs. This means that the flow of the code is constant over the draw call. The primary benefit of static flow control is to allow the same shader to be used in a variety of different situations (e.g., a varying numbers of lights). There is no thread divergence, since all invocations take the same code path.

**Dynamic flow control** is based on the values of varying inputs, meaning that each fragment can execute the code differently. This is much more powerful than static flow control but can cost performance, especially if the code flow changes erratically between shader invocations.