Kernel Launch

Prateek Shukla

Some important constraints and functions

`__cluster_dims__(x,y,z)` for compile-time thread-block cluster shape.

`__launch_bounds__(maxThreads[,minBlocksPerSM[,maxBlocksPerCluster]])`. The 3rd argument is the cluster-specific one that matters more on SM90.

`__maxnreg__(N)` to cap registers per thread.

`__grid_constant__` for read-only grid-lifetime kernel params. This is especially common for CUtensorMap / TMA descriptors.

`__forceinline__` Forces nvcc to inline the function within a single translation unit.

`__restrict__` tells nvcc that, for the lifetime of that pointer in the current scope, the pointed-to memory is not aliased by any other pointer used to access the same data

Need for multi kernel setups

The epilogue has a hard ceiling. Pushing past it destroys your performance. Fused operations compete with the mainloop's accumulator tiles which might cause register pressure.

Operations demanding cross-tile dependencies (LayerNorm, Softmax) are architecturally incompatible with the independent-tile processing of a GEMM epilogue. The math strictly requires a kernel boundary to reduce properly.

You must accept kernel boundaries and intermediate DRAM writes. The engineering challenge isn't forcing a monolithic kernel it's making the handoff between kernels virtually free

This is done by deploying dependent-launch protocols, L2 prefetch strategies, and cross grid mbarriers

The Stream Serialization Problem

Generally one grid must fully retire before the next grid on the same stream can begin issuing work. This strict completion ordering creates a utilization gap as the first grid enters its low-occupancy tail.

The hardware scheduler treats same-stream grids as a single ordered queue. It does not hand out thread-block slots for Kernel B until Kernel A reports global completion, even if some SMs are already underutilized.

As Kernel A nears completion, only a shrinking subset of SMs still has active work. The remaining SMs sit idle because no next-grid blocks are eligible under default stream rules.

The handoff is a full-grid retirement event, not a tile-by-tile transfer. That means all dependency-safe overlap opportunities are ignored unless explicit dependent-launch control is enabled.

The critical cost is tail latency amplification: silicon is present but not issuing useful instructions during the final phase of Kernel A.

Hardware Signaling (GDC Instructions)

GDC instructions control over grid handoff timing inside the GPU execution timeline. Its physical purpose is to separate “scheduler may start dependents” from “dependents may read dependency-sensitive memory.”

`griddepcontrol.launch_dependents` is a scheduler-facing signal emitted by running warps. It tells the dispatch logic that the dependent grid is now eligible to boot, even before the active grid has globally retired.

`griddepcontrol.wait` is a hard execution fence in the dependent grid. Warps reaching this instruction are held until the launch dependency condition is satisfied, preventing premature reads of unresolved data.

Together they form a two-part protocol: early boot permission plus memory-safety gating. The scheduler can overlap startup work while the fence preserves correctness for dependency-sensitive loads. These are physical control points in the instruction stream, so placement directly changes overlap timing and hardware contention.

9.7.13.13. Parallel Synchronization and Communication Instructions: `griddepcontrol`

`griddepcontrol`

Control execution of dependent grids.

Syntax

```
griddepcontrol.action;

.action   = { .launch_dependents, .wait }
```

Description

The `griddepcontrol` instruction allows the dependent grids and prerequisite grids as defined by the runtime, to control execution in the following way:

`.launch_dependents` modifier signals that specific dependents the runtime system designated to react to this instruction can be scheduled as soon as all other CTAs in the grid issue the same instruction or have completed. The dependent may launch before the completion of the current grid. There is no guarantee that the dependent will launch before the completion of the current grid. Repeated invocations of this instruction by threads in the current CTA will have no additional side effects past that of the first invocation. A *release fence* preceeding a `griddepcontrol.launch_dependents` in program-order *synchronizes* with the start of a dependent grid if either both grids have the same memory synchronization domain and the fence scope is `gpu`, or if the fence scope is `sys`.

`.wait` modifier causes the executing thread to wait until all prerequisite grids in flight have completed and all the memory operations from the prerequisite grids are performed and made visible to the current grid.

Note

If the prerequisite grid is using `griddepcontrol.launch_dependents`, then the dependent grid must use `griddepcontrol.wait` to ensure correct functional execution.

Host Launch Authority

Only the CPU launch packet can authorize relaxed stream behavior for dependent overlap. The physical purpose is to keep default stream semantics intact unless host software explicitly opts in.

At launch time, the CPU writes attributes into the command descriptor consumed by the GPU command processor. One attribute grants permission for dependent-grid scheduling before full predecessor retirement.

If that permission bit is present, scheduler state machines honor launch_dependents and wait as dependency control instructions. If it is absent, the scheduler enforces normal serialization and the instructions have no enabling effect.

The handoff authority is therefore host-originated and hardware-enforced. Device-side signaling cannot override stream policy unless launch metadata authorizes it.

This protects correctness across mixed workloads, where some streams need strict ordering and others need controlled overlap.

```cpp
// launch primary normally
primary_kernel<<<gridA, blockA, 0, stream>>>(...);

// launch secondary with the special attribute
cudaLaunchAttribute attr[1];
attr[0].id = cudaLaunchAttributeProgrammaticStreamSerialization;
attr[0].val.programmaticStreamSerializationAllowed = 1;

cudaLaunchConfig_t cfg{};
cfg.gridDim  = gridB;   // for secondary_kernel
cfg.blockDim = blockB;  // for secondary_kernel
cfg.attrs    = attr;
cfg.numAttrs = 1;

cudaLaunchKernelEx(&cfg, secondary_kernel, ...);
```

The Dependent’s Wall (Wait)

This phase defines the early-boot stall point for producer warps in the dependent grid. Its physical purpose is to let the dependent kernel reserve execution context early while blocking unsafe memory traffic.

The dependent grid can be admitted and begin executing setup instructions on available SMs. Producer warps run until they reach `griddepcontrol.wait`, where they are parked by hardware.

While parked, these warps cannot issue dependency-sensitive global reads such as activation tensors produced by the active kernel. This prevents cache fill and memory ordering violations from reading not-yet-final data.

Once the dependency signal is satisfied, the fence releases and those same warps resume issuing loads. The result is immediate forward progress without paying full cold-start delay after producer completion.

The tradeoff is reserved resources: parked warps still consume scheduling slots and can reduce instantaneous headroom for other work.

The Producer’s Green Light (Launch)

This phase is where the active kernel emits the exact signal that opens dependent scheduling. Its physical purpose is to trigger overlap at the last correctness-safe moment that still exposes startup latency hiding.

Producer warps in the active grid execute `griddepcontrol.launch_dependents` when their remaining writes are sufficiently advanced for dependent boot. This signal targets scheduler eligibility, not immediate memory-read permission.

In persistent kernels, correct placement is tied to the final scheduled work tile’s lifecycle. The signal should be aligned with true pipeline completion, not merely the lexical end of a code region.

After the signal, dependent blocks can be dispatched while the active grid drains residual work. The dependent grid’s wait fence still guards dependency-sensitive reads until the required condition is met.

Signaling too late wastes overlap; signaling too early increases concurrent pressure and can reduce net throughput.

The L2 Prefetch Trick (Split DMA)

This phase implements overlap by dividing roles across warps in the dependent kernel. Its physical purpose is to keep memory fabric busy with dependency-free transfers while dependency-bound producers remain fenced.

One producer warp reaches the dependency barrier and pauses before activation reads. A different prefetch warp continues running and issues DMA-style requests for static weights that are independent of producer completion.

Those requests are targeted to populate L2 ahead of compute demand, reducing later miss penalties when the dependency fence opens. The memory system can therefore warm useful cache lines during otherwise stalled dependency time.

The handoff is asymmetric but safe: activation traffic waits behind the fence, weight prefetch does not. When the fence releases, producers see improved effective latency because part of the working set is already resident.

This strategy only works if prefetched data has high reuse and survives in L2 until consumption.

Tuning the Overlap Window

The overlap ratio is set by the timing gap between dependent-launch signal emission and true dependency readiness. A larger gap increases potential hiding, but only if resources remain uncongested.

If signaling is too early, both kernels compete for SM issue slots, register file capacity, shared memory allocation, and memory ports. This can throttle each kernel enough that total time increases instead of decreases. If prefetch is too aggressive, L2 lines churn before use and DRAM traffic spikes from refills. Bandwidth is then spent moving evicted data back in, erasing the gain from prefetch overlap.

Practical tuning uses hardware counters: launch at the latest safe point, prefetch only stable /reused tensors, and watch L2 hit rate plus memory-queue pressure as the primary guardrails.