Mastering NVIDIA CUDA: Expert GPU Programming

Table of Contents

45. Accelerating Deep Learning Inference with TensorRT
46. Custom Layers and Plugin Development for AI
47. Dynamic Programming and Sequence Alignment on GPUs
48. Monte Carlo Simulations and RNG Optimization
49. Cryptographic Primitives and Blockchain Acceleration
50. CFD and Molecular Dynamics: Real-world HPC Scaling
51. Ray Tracing Cores for Non-Graphics Workloads
52. Video Encoding and Decoding via NVENC and NVDEC
53. Optical Flow and Computer Vision Hardware Acceleration
54. Unified Memory: Advanced Prefetching and Advice
55. Heterogeneous Memory Management on Linux
56. Multi-GPU Load Balancing and Task Distribution
57. Zero-Copy Memory and PCIe Gen 6 Considerations
58. Building High-Performance Data Loaders for AI
59. Quantization Algorithms for Large Language Models
60. Kernel Fusion Strategies for Complex Pipelines
61. Autotuning Kernels for Cross-Architecture Portability
62. The Future of GPGPU: Beyond Blackwell
63. Hardware-Software Co-Design for Specialized Domains
64. Implementing Software-Managed Caches in Shared Memory
65. Bit-Manipulation Instructions and Advanced Integer Math
66. Handling Large Scale Data with NVIDIA Multi-Node Toolkit
67. Virtual Memory Management API Deep Dive
68. Cooperative Launch and Grid-Level Synchronization
69. Reducing Tail Latency in Real-time GPU Systems
70. Optimizing for the NVIDIA Grace Hopper Superchip
71. The Grace CPU and LPDDR5X Memory Integration
72. C++ Standard Parallelism for NVIDIA GPUs
73. Python Performance with CuPy and Numba
74. Integrating CUDA with Rust and Modern Systems Languages
75. Formal Verification of CUDA Kernels
76. Security Vulnerabilities in GPGPU Kernels
77. Energy-Efficient GPU Programming
78. Managing Thermal Throttling and Power Limits
79. CI/CD Pipelines for High-Performance Computing
80. Deploying CUDA at Scale: Kubernetes and Slurm
81. Debugging Race Conditions and Memory Corruption
82. Using Sanitizers for CUDA Memory Validation
83. Implementing Custom Allocators for GPU Memory
84. Exploring the CUDA Driver API vs Runtime API
85. Static Analysis Tools for Parallel Code
86. Performance Portability with Kokkos and RAJA
87. OpenMP Offloading for NVIDIA Hardware
88. The CUTLASS Library: Templates for Matrix Multiplies
89. CuTe: A New Abstraction for Tiled Layouts
90. NVIDIA Warp: A Framework for Physics Simulation
91. Optimizing Stencil Computations and Finite Difference
92. Sort Scan and Reduction: The Building Blocks of Parallelism

93. Designing Low-Latency Audio and Signal Processing Kernels
94. Exploiting Hardware-Based Data Compression
95. Case Study: Optimizing Llama-3 for Blackwell
96. Case Study: Climate Modeling at Petabyte Scales
97. Case Study: Real-time Ray Traced Scientific Visualization
98. Sustaining Performance in Long-Running GPU Services
99. Community Ecosystem and Staying Current with CUDA Research
100. Masterclass Conclusion: The Path to Zero-Overhead Computing

Example:

Chapter 26: CUDA Graphs: Eliminating CPU-side Overhead
In the pursuit of zero-overhead computing on Blackwell and Hopper architectures, the CPU is increasingly the primary bottleneck. As kernel execution times shrink to the microsecond range—common in real-time inference, high-frequency signal processing, and small-batch Transformer operations—the latency of the CUDA driver and the operating system's scheduling overhead becomes a dominant factor. Every cudaLaunchKernel call incurs a CPU-side cost involving argument validation, stack setup, and submission to the GPU's command processor. When a workload consists of thousands of short-lived kernels, the GPU often sits idle while the CPU struggles to feed it.

CUDA Graphs represent a paradigm shift from the imperative "submit-and-wait" model to a declarative "define-once, execute-many" model. By capturing the entire dependency structure of a workflow into a persistent graph object, you allow the CUDA driver to perform heavy-duty optimizations and hardware-specific scheduling upfront, reducing the cost of subsequent launches to a single, low-latency doorbell register write.

The Anatomy of CPU-Side Bottlenecks

Traditional stream-based execution is fundamentally serial from the perspective of the driver. Even when multiple kernels are queued in a stream, the CPU must still process each one individually. In a typical Blackwell-based environment, a single kernel launch might take 5–10 microseconds of CPU time. If the kernel itself only executes for 3 microseconds, the GPU utilization will never exceed 30%, regardless of the hardware's 20-petaflop capability.

CUDA Graphs eliminate this by moving the graph construction out of the "hot path." The driver analyzes the entire DAG (Directed Acyclic Graph) of kernels, memory copies, and synchronization events, creating a pre-baked binary command stream that the GPU's Command Processor (GCP) can ingest directly.

Capture, Instantiation, and Execution

The workflow for mastering CUDA Graphs involves three distinct phases: Capture, Instantiation, and Execution.

1. Stream Capture

The most ergonomic way to create a graph is via Stream Capture. This allows you to use existing stream-based code without a total rewrite.

```
cudaGraph_t graph;
cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);

// All work submitted to this stream is now being recorded into the graph
launch_kernel_A<<<grid, block, 0, stream>>>(...);
launch_kernel_B<<<grid, block, 0, stream>>>(...);
cudaMemcpyAsync(dst, src, size, cudaMemcpyDeviceToDevice, stream);

cudaStreamEndCapture(stream, &graph);
```

During capture, no work is actually dispatched to the GPU. Instead, the driver records the operations and their implicit dependencies (based on the stream order).

2. Instantiation (The Optimization Phase)

Instantiation is where the "magic" happens. The driver takes the cudaGraph_t and produces a cudaGraphExec_t. This is a heavyweight process where the driver:

Allocates hardware resources.

Validates all kernel parameters.

Determines the optimal execution order.

Pre-calculates command buffer offsets.

For Blackwell and Hopper, the instantiation process also optimizes for distributed shared memory (DSMEM) and TMA (Tensor Memory Accelerator) operations, ensuring that kernels sharing data within a Thread Block Cluster are scheduled on SMs that are physically close or linked via high-speed on-chip interconnects.

3. Execution

Launching the instantiated graph is a near-zero-cost operation:

```
cudaGraphLaunch(instance, stream);
```

This single call replaces dozens or hundreds of individual kernel launches, collapsing the CPU overhead into a single point.

Dynamic Graph Updates: Mastering Parameter Evolution

The most common criticism of CUDA Graphs is their perceived "static" nature. In real-world applications, pointers and parameters often change between iterations (e.g., swapping double buffers or updating learning rates). Re-instantiating a graph to reflect these changes is a performance anti-pattern, as instantiation is orders of magnitude more expensive than launch.

To achieve peak throughput, you must use Graph Executable Updates. This allows you to modify the parameters of a kernel node within a cudaGraphExec_t without re-optimizing the entire structure.

```
cudaGraphExecUpdateResult updateResult;
cudaGraphNode_t kernelNode; // Retrieved during capture or manual build
cudaKernelNodeParams updatedParams = {0};
updatedParams.func = (void*)my_kernel;
updatedParams.gridDim = newGrid;
updatedParams.blockDim = newBlock;
updatedParams.kernelParams = newArgs;

cudaGraphExecKernelNodeSetParams(instance, kernelNode, &updatedParams);
```

On Blackwell, this mechanism is further accelerated. The hardware supports a "fast path" for updating certain parameters, allowing for sub-microsecond updates that enable graphs to remain active even in highly dynamic workloads like Reinforcement Learning.

Conditional Nodes and In-Graph Control Flow

One of the most powerful features introduced for Hopper and Blackwell (CUDA 12.x+) is the ability to handle Conditional Nodes. Historically, any "if/else" logic based on GPU data required synchronization back to the CPU, effectively breaking the graph.

With Conditional Nodes, you can define sub-graphs that are only executed if a specific GPU-side condition is met. This is implemented via a "Condition Node" that reads a value from device memory.

IF/ELSE: Execute Graph A or Graph B based on a predicate.

WHILE: Loop a sub-graph until a GPU-resident counter reaches a limit.

This allows the GPU to handle complex logic internally—such as early-exit in iterative solvers or dynamic pruning in LLM inference—without ever involving the CPU's interrupt latency.

Architectural Nuance: Blackwell’s Hardware-Accelerated Scheduling

Blackwell introduces enhancements to the GPU's command processor that specifically target the latency between nodes in a CUDA Graph. In previous architectures, even with graphs, there was a small gap—the "tail-to-start" latency—where the GPU might momentarily starve while the next node was dispatched.

Blackwell’s hardware-level command streaming reduces this to near-zero. By utilizing the Hardware Command Processor, Blackwell can pre-fetch the next node's descriptors while the current kernel is still occupying the SMs. This is particularly vital for kernels that saturate the GPU’s compute throughput but finish very quickly.

Advanced Memory Management with Graphs

A common pitfall when using CUDA Graphs is the interaction with cudaMalloc and cudaFree. These calls are synchronous and extremely slow. To solve this, CUDA Graphs integrate with Stream-Ordered Memory Allocators.

When you capture a cudaMallocAsync call inside a graph, the driver treats the memory as a "graph-managed resource." This allows the memory to be reused between graph executions without the overhead of returning it to the system-wide pool. For a senior engineer, this is the key to building "zero-allocation" loops where the entire memory lifecycle is handled within the optimized command stream.

Practical Implementation Strategy for High-Throughput Pipelines

To master CUDA Graphs on the latest hardware, follow this performance-oriented blueprint:

Identify the Iterative Core: Find the most repetitive part of your pipeline (e.g., the forward pass of a transformer layer or a single timestep in a CFD simulation).

Minimize Synchronization: Ensure there are no cudaStreamSynchronize or cudaDeviceSynchronize calls within the sequence you intend to capture.

Warm-up and Capture: Run the stream once to warm up the driver and caches, then capture the graph.

Leverage Whole-Graph Dependencies: Instead of manual cudaEvent_t synchronization, let the graph's dependency edges define the execution order. The driver can often find more parallelism than the developer by analyzing these edges.

Use the "Update or Patch" Pattern: If pointers change every iteration, use cudaGraphExecUpdate. If only scalar values (like a threshold) change, use cudaGraphExecKernelNodeSetParams or, better yet, have the kernel read those values from a dedicated "parameter buffer" in device memory that you update via cudaMemcpyAsync.

Real-World Case Study: Micro-Batch Inference

In an LLM inference scenario on a B200, processing a single token might involve 50+ small kernels (normalization, attention projections, bias additions). Without CUDA Graphs, the CPU overhead of launching these 50 kernels can exceed the actual GPU execution time by 2x.

By capturing these 50 kernels into a single CUDA Graph, the total end-to-end latency can be reduced by 40-60%. Furthermore, by utilizing Conditional Nodes, the graph can handle the KV-cache management (deciding whether to grow the cache or reuse slots) entirely on the GPU, eliminating the PCIe round-trip latency that typically kills real-time responsiveness.

Summary of Actionable Takeaways

Move Beyond Capture: While cudaStreamBeginCapture is easy, manual graph construction using cudaGraphAddNode provides finer control over node priorities and can lead to better scheduling on Blackwell's complex GPC/TPC hierarchy.

Monitor Graph Overhead: Use NVIDIA Nsight Systems to visualize the "gap" between kernels. If you see white space between nodes in a graph, it often indicates a dependency bottleneck or an SM-to-SM synchronization delay that can be optimized by adjusting block sizes or memory layouts.

Prioritize Persistence: The goal is to instantiate once and launch millions of times. If your application logic forces frequent re-instantiation, rethink your data structures to allow for graph updates instead.

Chapter 27: Stream-Ordered Memory Allocation and Management

In the pursuit of deterministic latency and maximum throughput on the Blackwell and Hopper architectures, the traditional memory management model—characterized by synchronous cudaMalloc and cudaFree calls—is a relic of a previous era. For senior engineers building exascale simulations or real-time LLM inference engines, these calls represent "stop-the-world" events. A standard cudaMalloc must synchronize the entire device, wait for all pending work to complete, and then negotiate with the operating system’s kernel-mode driver to modify the GPU page tables. This process can take hundreds of microseconds, an eternity when your compute kernels are finishing in ten.

Stream-ordered memory allocation, introduced via cudaMallocAsync and cudaFreeAsync, fundamentally changes the contract between the application and the CUDA driver. It treats memory allocation not as a blocking management task, but as a first-class asynchronous operation that can be pipelined alongside compute and data movement.

The Mechanism of Asynchronous Allocation

Stream-ordered allocation operates on the principle of memory pools (cudaMemPool_t). When you call cudaMallocAsync, you are not necessarily requesting the driver to map new

physical pages from the OS. Instead, you are requesting a slice of memory from a pre-allocated pool that is managed by the CUDA driver in user-space.

The "stream-ordered" nature means that the allocation and deallocation are scoped to a specific timeline. If you allocate memory in stream A, that memory is guaranteed to be available for any subsequent operation in stream A. Crucially, the driver can reuse that same memory for another allocation in stream B, but only after it has inserted the necessary internal hardware dependencies to ensure that the work in stream A is finished. This removes the need for the CPU to manually synchronize streams just to recycle memory buffers.

Memory Pools: The Engine of Performance

To master this system, you must move beyond the default pool and implement custom cudaMemPool_t configurations tailored to your workload's memory footprint.

1. Configuring Reuse Policies

The primary advantage of a memory pool is the ability to bypass the OS-level "de-allocate and re-allocate" cycle. You can control this behavior using the cudaMemPoolAttrReleaseThreshold. By setting a high threshold, you instruct the driver to keep "freed" memory resident in the GPU’s physical VRAM rather than returning it to the system.

```
cudaMemPool_t mempool;
cudaDeviceGetDefaultMemPool(&mempool, device_id);
uint64_t threshold = 1024LL * 1024LL * 1024LL; // 1GB
cudaMemPoolSetAttribute(mempool, cudaMemPoolAttrReleaseThreshold, &threshold);
```

On Blackwell, where HBM3e bandwidth is measured in terabytes per second, minimizing the frequency of page table updates is critical. Every time the OS has to intervene in memory management, it risks invalidating TLB (Translation Lookaside Buffer) entries on the GPU, leading to "TLB shootdowns" that stall the memory controllers. By maintaining a high release threshold, you ensure that the GPU’s MMU (Memory Management Unit) stays "warm" and optimized for your specific memory addresses.

2. Cross-Stream Reuse and Event Dependencies

One of the most sophisticated features of stream-ordered allocation is the cudaMemPoolReuseFollowEventDependencies attribute. By default, memory freed in stream A might not be immediately available to stream B unless the driver is certain they don't overlap. By enabling this attribute, you allow the allocator to look at cudaEvent_t dependencies. If stream B is waiting on an event recorded in stream A, the allocator knows it is safe to immediately reassign stream A's freed buffers to stream B.

Integration with CUDA Graphs

Chapter 26 discussed the elimination of CPU overhead via CUDA Graphs. Stream-ordered allocation is the final piece of that puzzle. When capturing a graph, you can include cudaMallocAsync and cudaFreeAsync nodes.

During the instantiation phase, the driver analyzes the entire lifecycle of these allocations within the graph. It can "pre-plan" the memory addresses, effectively turning the dynamic allocation into a static offset calculation. This is known as Graph Memory Management. It allows a complex pipeline—for example, a multi-layer transformer—to reuse the same scratch buffers for each layer's intermediate activations without ever truly "allocating" memory during the execution phase. The result is a zero-overhead memory lifecycle that survives millions of graph iterations.

Inter-Process Communication (IPC) and Memory Sharing

In multi-GPU environments (e.g., an H100/B200 NVLink cluster), stream-ordered allocation extends to IPC. Traditionally, sharing memory between processes required a complex dance of exporting and importing handles.

With stream-ordered pools, you can use cudaMemPoolExportPointer and cudaMemPoolImportPointer. This allows a "producer" process to allocate a buffer asynchronously, and a "consumer" process to begin using it as soon as the associated stream-ordered signal is reached. In the context of NCCL (NVIDIA Collective Communications Library), this enables "Zero-Copy" collective operations where the communication buffers are managed by the stream-ordered allocator, reducing the latency of All-Reduce operations in distributed training.

Strategic Implementation: The "Transient Buffer" Pattern

For senior HPC engineers, the most effective use of stream-ordered allocation is the Transient Buffer Pattern. This is used for temporary workspace memory required by libraries like cuFFT or cuSOLVER.

The Anti-Pattern:

```
// Bad: Synchronizes the device, high overhead
void* workspace;
cudaMalloc(&workspace, size);
launch_kernel<<<...>>>(workspace);
cudaFree(workspace);
```

The Master-Level Pattern:

```
// Good: Zero synchronization, fully pipelined
void* workspace;
cudaMallocAsync(&workspace, size, stream);
launch_kernel<<<...>>>(workspace, stream);
cudaFreeAsync(workspace, stream);
// The SM starts the kernel immediately; the driver handles the workspace lifecycle
```

In this model, the "allocation" is just another command in the GPU’s command queue. If the pool has available memory, the "allocation" takes effectively 0ns of CPU time. If the pool is empty, the driver will perform a one-time expansion, which is still faster than a standard cudaMalloc because it doesn't require a global device sync.

Memory Fragmentation and Blackwell's HBM3e

A common concern with long-running GPU services is fragmentation. Over time, frequent allocations and deallocations of varying sizes can leave "holes" in the memory space. The stream-ordered allocator is designed with a Best-Fit strategy that is significantly more robust than the old runtime allocator.

On Blackwell, the physical memory is organized into multiple stacks of HBM3e. The stream-ordered allocator is "topology-aware," meaning it attempts to allocate memory in a way that maximizes the efficiency of the memory controllers. When you use cudaMallocAsync, the driver can prioritize allocating memory that is physically "closer" to the SMs that are currently active in that stream, reducing internal crossbar contention within the GPU.

Hardware-Level Nuance: The Role of the GigaThread Engine

The GigaThread Engine on Hopper and Blackwell is responsible for dispatching work to the SMs. It also plays a role in stream-ordered management. When a cudaFreeAsync is encountered in the command stream, the GigaThread Engine tracks the completion of all kernels that were launched before that free command.

Once the "last" kernel using that memory finishes, the GigaThread Engine signals the memory controller that the range is now "available for reuse." This hardware-software co-design ensures that the "hand-off" between two kernels using the same memory happens at the hardware signal level, bypassing the OS kernel entirely. This is why stream-ordered allocation is the only viable path for achieving sub-10-microsecond tail latencies in high-frequency trading or real-time sensor processing.

Practical Takeaways for Performance Engineering

Default to Async: In any performance-critical path, cudaMalloc should be considered a bug. Replace it with cudaMallocAsync.

Warm Your Pools: At application startup, perform a large "dummy" cudaMallocAsync and cudaFreeAsync to grow the pool to the expected high-water mark of your application. This avoids OS-level allocations during the hot path.

Leverage Snapshots: Use cudaMemPoolGetAttribute with cudaMemPoolAttrReservedMemCurrent and cudaMemPoolAttrUsedMemCurrent to profile your memory usage without using external tools.

Align with Streams: Ensure that the stream used for allocation is the same stream used for the compute. If you must use different streams, use cudaStreamWaitEvent to establish a clear dependency before the second stream accesses the memory.

By mastering stream-ordered memory, you move from a model where the CPU manages the GPU, to a model where the CPU merely provides a high-level schedule, and the GPU hardware—assisted by the low-level driver—self-orchestrates its resources for maximum efficiency.

Chapter 28: Multi-Instance GPU for High-Density Computing

In the landscape of modern data centers, the monolithic utilization of a GPU is often an architectural inefficiency. While a Blackwell B200 or a Hopper H100 possesses staggering throughput, many workloads—ranging from micro-service inference to CI/CD kernel validation—cannot saturate the massive compute and memory bandwidth of these chips. Multi-Instance GPU (MIG) is the architectural answer to this fragmentation, allowing a single physical GPU to be partitioned into multiple, hardware-isolated instances. Unlike software-level partitioning or time-slicing, MIG provides dedicated hardware resources, including SMs, memory controllers, and L2 cache, ensuring that a "noisy neighbor" in one instance cannot degrade the performance or deterministic latency of another.

The Architectural Philosophy of Hardware Partitioning

Historically, sharing a GPU meant relying on the OS scheduler (time-slicing) or the Multi-Process Service (MPS). Time-slicing introduces latency spikes as the context of a 100GB+ memory state is swapped, and MPS, while efficient for compute sharing, lacks true hardware isolation; a single memory-hungry kernel in one MPS process can still stall the entire memory pipeline for others.

MIG shifts this paradigm by physically carving the GPU’s silicon. On a Hopper H100 or Blackwell B200, the hardware is divided into GPU Instances (GI). Each GI contains a specific number of GPCs (Graphics Processing Clusters) and their associated Streaming Multiprocessors (SMs), as well as a dedicated slice of the L2 cache and HBM memory controllers. Within a GI, you can further define Compute Instances (CI), which allow for even finer granularity in compute resource allocation while sharing the memory and cache of the parent GI. This hierarchy ensures that the path from the SM to the local HBM stack is physically isolated from other instances, providing a "Hard QoS" (Quality of Service) that is essential for multi-tenant environments.

Profiles and Resource Slicing on Blackwell

The granularity of MIG is dictated by "profiles." On the Blackwell architecture, these profiles have reached a new level of density. A typical profile is denoted as [C]g.[M]gb, where C is the number of compute slices and M is the amount of HBM memory.

For instance, a 1g.10gb profile on an H100 provides 1/7th of the GPU's compute and memory. On a Blackwell B200, however, the increased SM count and HBM3e bandwidth mean that even a single-slice instance can outperform a full-die Ampere-generation GPU.

The senior engineer must view these slices not as "restricted" versions of the hardware, but as "right-sized" execution environments.

The configuration of these slices is managed through the NVIDIA Management Library (NVML) or nvidia-smi. A critical nuance in the Blackwell era is the introduction of dynamic reconfiguration without a full driver reload, although existing instances must still be idle to change the partitioning scheme.

```
# Example: Partitioning a B200 into two 3g.40gb instances and one 1g.20gb instance
nvidia-smi mig -cgi 3g.40gb,3g.40gb,1g.20gb -C
```

This command doesn't just logically separate the GPU; it reconfigures the internal crossbar and memory controller mappings to ensure that the 3g.40gb instance has physically zero access to the address space or execution units of its neighbors.

MIG vs. MPS: Choosing the Right Multi-Tenancy Strategy

A common mistake at the senior level is treating MIG and MPS as interchangeable. They are complementary technologies with distinct trade-offs:

Fault Isolation: In MIG, if a kernel in one instance triggers a GPU reset (e.g., due to an out-of-bounds memory access), only that instance is affected. In MPS, a fatal error in one process can crash the entire GPU, affecting all other shared processes.

Resource Contention: MPS allows for "oversubscription," where multiple processes can compete for the same SMs to maximize total utilization. MIG is "strict"—if an instance is idle, its assigned SMs cannot be used by a neighboring instance.

Latency Determinism: For real-time applications, MIG is the gold standard. Since the L2 cache and memory paths are partitioned, there is zero cache-thrashing between tenants.

In high-density computing, the strategy is often to use MIG for coarse-grained isolation (e.g., separating three different customers) and then MPS within a single MIG instance to further pack small, related tasks (e.g., multiple micro-batch inference workers for the same customer).

Memory Hierarchy and Inter-Instance Communication

When a GPU is in MIG mode, the memory address space is strictly partitioned. This has significant implications for CUDA development. Each MIG instance appears to the CUDA driver as a unique CUdevice handle with its own limited memory capacity.

A critical constraint to master is that NVLink and P2P (Peer-to-Peer) communication are generally disabled between MIG instances on the same physical die. Because the hardware is focused on isolation, the high-speed cross-instance synchronization required for NVLink-based collective operations (like ncclAllReduce) is restricted. If your workload requires multi-GPU scaling within a single node, you must ensure that your MIG profile

covers the entire GPU or use GPUDirect RDMA over an external fabric to bridge instances, though this incurs a latency penalty.

However, the Shared Memory Controller (SMC) in Hopper and Blackwell has been optimized to handle MIG-level partitioning with minimal impact on effective bandwidth. Even when the HBM3e is sliced into seven parts, the hardware ensures that each slice maintains its proportional share of the peak theoretical bandwidth, preventing a single instance from saturating the global memory bus.

Programming Considerations for MIG-Aware Applications

From the perspective of a CUDA C++ application, a MIG instance is largely transparent. Calls to cudaGetDeviceCount will return the number of MIG instances visible to the process (usually restricted via CUDA_VISIBLE_DEVICES). However, deep-level optimizations must account for the reduced resource pool.

Occupancy Calculations: A kernel optimized for a full B200 may fail to launch or perform poorly on a 1g.20gb instance because the available register file and shared memory are proportional to the slice size. You must use cudaOccupancyMaxActiveBlocksPerMultiprocessor to dynamically adapt block sizes to the specific MIG instance's SM count.

L2 Cache Sensitivity: In a full GPU, a kernel might rely on 50MB+ of L2 cache to keep a large lookup table resident. In a MIG slice, that L2 cache might be restricted to 5-10MB. Performance engineers must monitor l2_hit_rate via Nsight Compute when migrating kernels from full GPUs to MIG instances.

Case Study: High-Density Inference on Blackwell

Consider a scenario where a B200 is deployed in a cloud environment to serve LLM inference requests. A single B200 might have enough memory and compute to handle a 70B parameter model, but if the incoming request volume consists of small, sporadic prompts, the GPU will spend most of its time waiting for the next request.

By partitioning the B200 into seven 1g.20gb instances, you can host seven independent instances of a smaller 7B parameter model.

Throughput: The total aggregate throughput (tokens/sec) across all seven instances often exceeds the throughput of a single 7B model running on the full GPU because the overhead of managing a massive grid is avoided.

Latency: Each customer gets a dedicated slice. A heavy request from "Customer A" cannot delay the "prefill" phase of "Customer B."

Security: MIG provides a hardware-enforced boundary. Even if a tenant successfully exploits a software-level vulnerability in the inference engine, they remain trapped within the 1/7th

slice of the GPU, unable to read the weights or activations of the models running in other instances.

Mastering NVML for Dynamic Orchestration

For engineers building the orchestration layer (e.g., a custom Kubernetes scheduler for GPUs), mastering the NVML API is required. You cannot simply launch a kernel and hope for the best; you must query the nvmlDeviceGetMigMode and nvmlDeviceGetGpuInstanceById to ensure the instance matches the workload's requirements.

When an instance is created, the system generates a unique MIG UUID. This UUID is the persistent handle used to map containers to hardware.

```
nvmlReturn_t result;
nvmlGpuInstancePlacement_t placement;
nvmlGpuInstance_t gpuInstance;
nvmlGpuInstanceProfileInfo_t profileInfo;

// Retrieve profile info for a 2-slice instance
result = nvmlDeviceGetGpuInstanceProfileInfo(device, profileId, &profileInfo);

// Create the hardware instance
result = nvmlDeviceCreateGpuInstance(device, profileId, &gpuInstance);
```

At this level of engineering, the goal is to build a "Self-Healing Fabric" where the orchestrator monitors the SM utilization of MIG instances and, during periods of low demand, collapses multiple small instances into a single large instance to handle a heavy batch-training job, then re-partitions them for inference during peak hours.

Performance Monitoring in Partitioned Environments

Monitoring a MIG-enabled GPU requires a shift in telemetry strategy. Global metrics like gpu_utilization are no longer sufficient because they aggregate data across all instances. You must utilize MIG-specific telemetry via Nsight Systems or the DCGM (Data Center GPU Manager).

On Hopper and Blackwell, the hardware includes dedicated performance counters for each GI. This allows you to track memory bandwidth saturation and pipe utilization per instance. If you see high sm_efficiency in Instance 0 but high mem_stall in Instance 1, you can conclude that Instance 1's kernel is poorly optimized for the reduced memory controller count, rather than assuming a global hardware bottleneck. This granular visibility is what allows senior engineers to achieve 90%+ total device utilization in high-density environments without sacrificing the latency of individual tasks

[THE END].