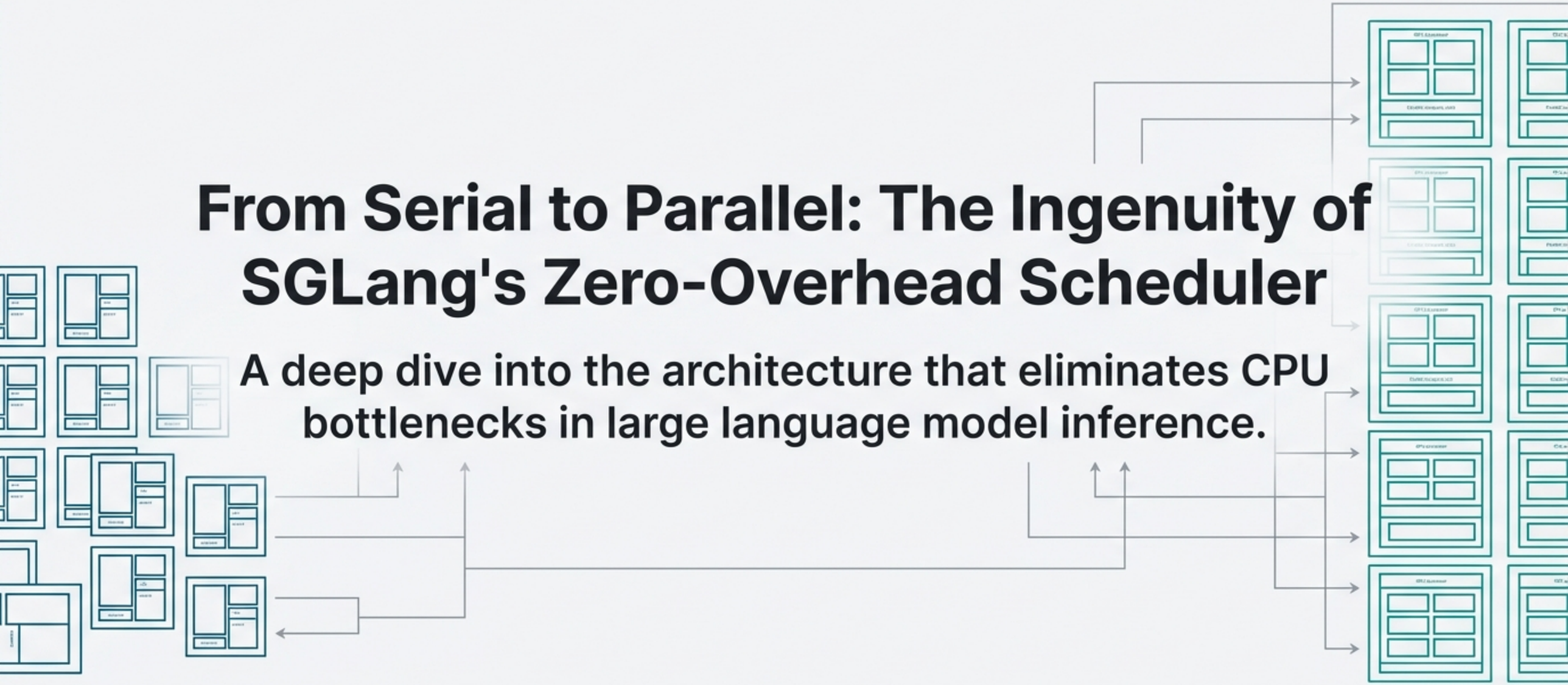
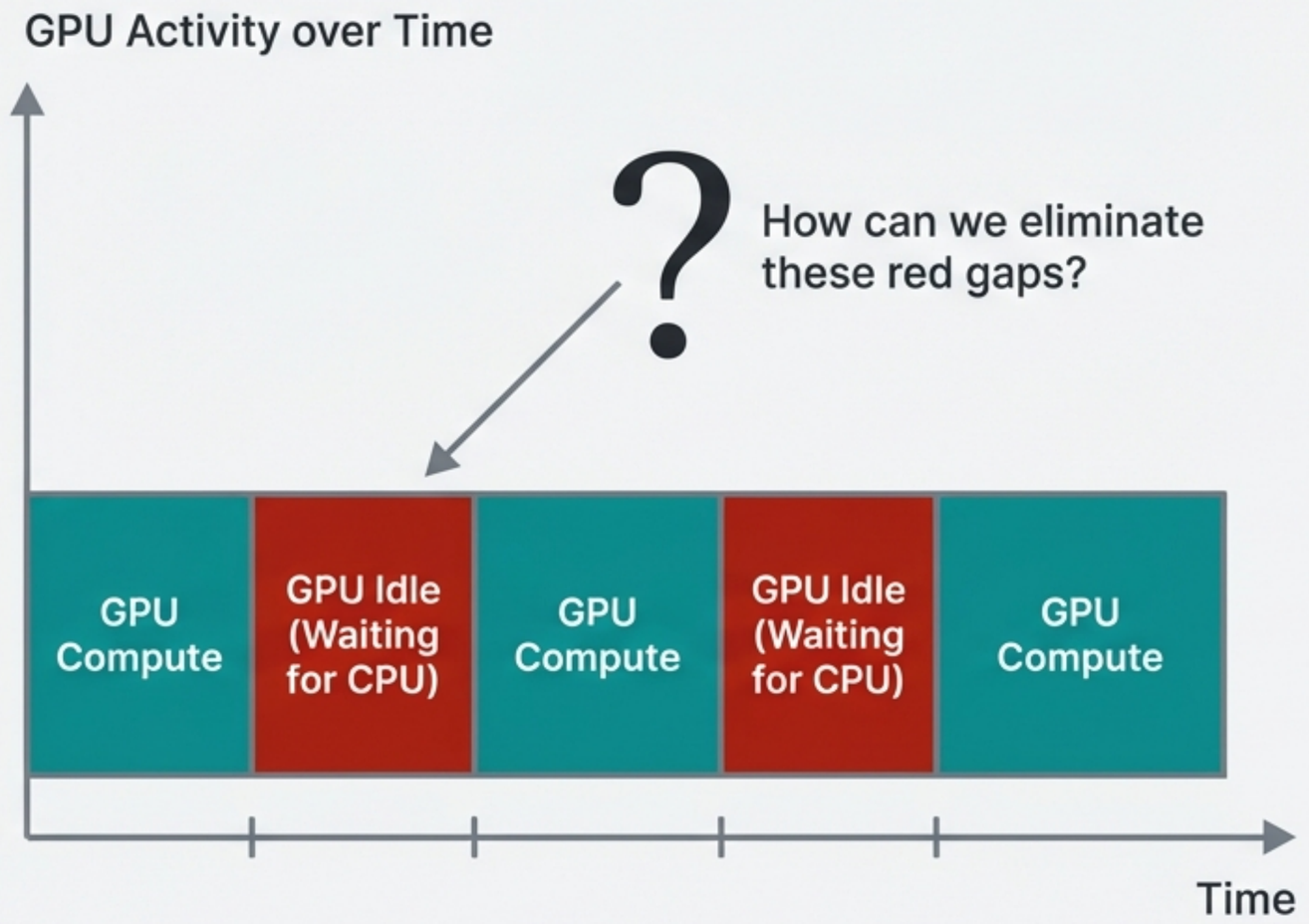


From Serial to Parallel: The Ingenuity of SGLang's Zero-Overhead Scheduler

A deep dive into the architecture that eliminates CPU bottlenecks in large language model inference.

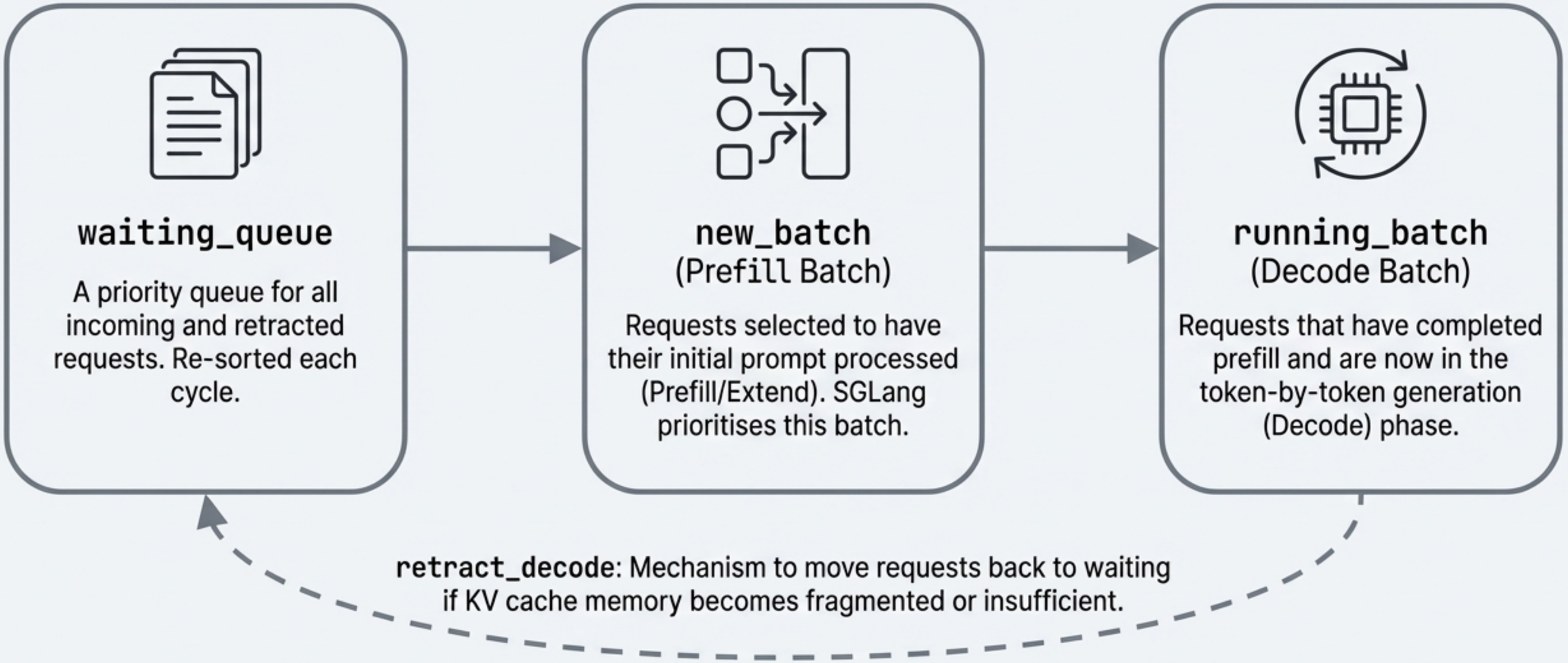
The Core Challenge: Maximising GPU Utilisation in LLM Serving

- LLM inference is a constant negotiation between stateful, dynamic requests and the need for high-throughput, parallel processing on the GPU.
- The primary performance limiter is often not the GPU's computation speed, but the 'bubbles' of idle time created while the CPU prepares the next batch.
- CPU-bound tasks include: tokenising inputs, managing KV Cache memory, preparing complex metadata tensors, and processing outputs.
- **The Goal:** To hide this CPU scheduling overhead completely behind the GPU's computation, achieving near-100% utilisation.



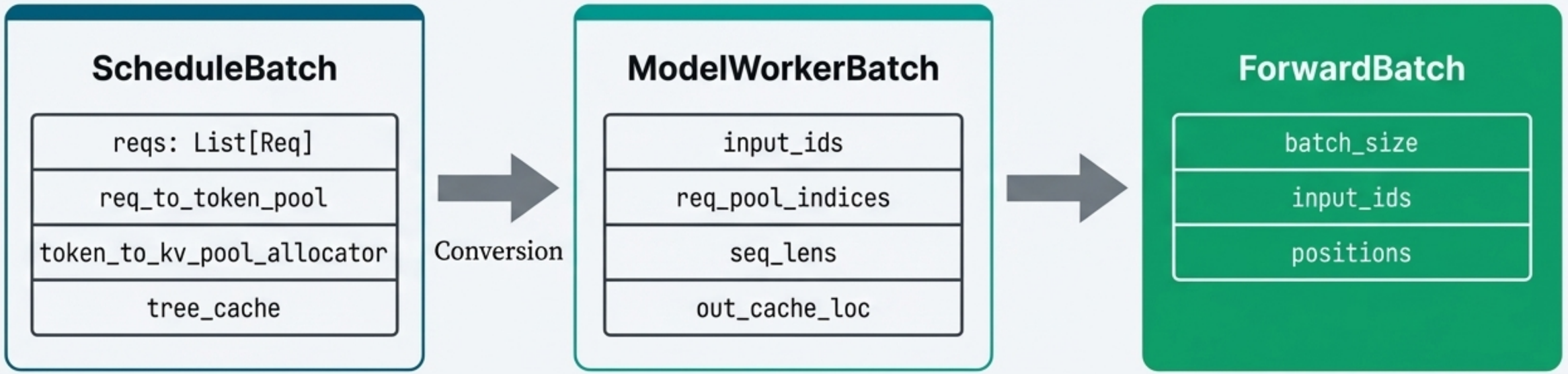
Managing the Request Lifecycle: Queues and Batches

The Scheduler orchestrates all active requests using a system of queues, each representing a distinct stage of processing.



From CPU Logic to GPU Execution: The Transformation of a Batch ModelWorkerBatch

A ‘batch’ is not a single object. It transforms as it moves from the CPU-based scheduler to the GPU-based model runner, shedding high-level logic for low-level tensor data.



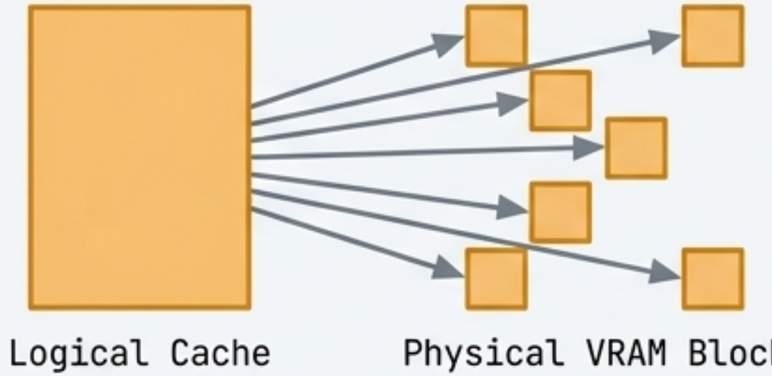
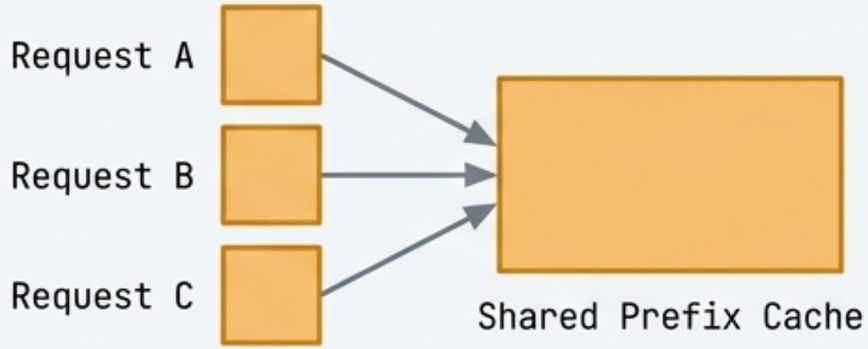
Managed by the Scheduler. Contains high-level logic and CPU-resident data.

Managed by the Model Worker. Contains only the data needed for the GPU forward pass.

Managed by the Model Runner. Low-level tensor data ready for the compute kernel.

The Memory Foundation: An Operating System Analogy for KV Cache

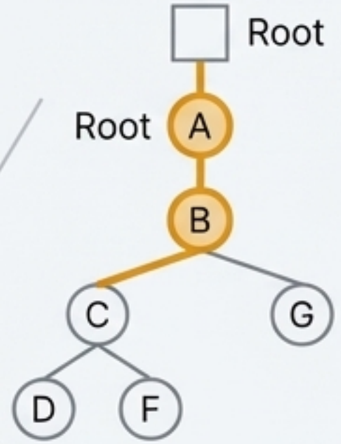
SGLang's throughput advantage comes from a sophisticated, two-level approach to KV Cache management, analogous to memory management in a modern OS.

PagedAttention (The Addressing Layer)	RadixCache (The Strategy Layer)
OS Analogy: Page Table Problem Solved: Memory Fragmentation. Allows logically continuous KV Cache to be stored in physically non-contiguous VRAM. Focus: 'How to store' data efficiently.  	OS Analogy: Shared Libraries / File Cache Problem Solved: Redundant Computation. Allows multiple requests with a common prefix to share the same underlying KV Cache data. Focus: 'What to store' to maximise reuse. 

PagedAttention and RadixCache are not competitors; they are complementary.
RadixCache leverages the flexibility of PagedAttention to enable extreme cache reuse.

A Concrete Look at the Three-Layer Cache System

```
R1: ['A', 'B', 'C', 'D']
R2: ['A', 'B', 'C', 'F']
R3: ['A', 'B', 'G', 'H']
```



Layer 1: RadixCache (Logical Tree)

Maintains a tree of token prefixes and their physical slot locations. A new request finds the longest matching prefix. Nodes have a `lock_ref` to prevent eviction while in use.

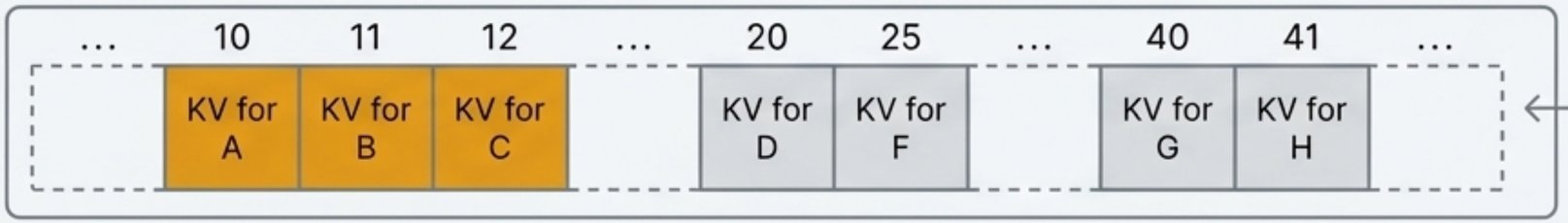
Layer 2: ReqToTokenPool (The "Page Table")

A 2D tensor mapping `[request_id`, `[request_id, token_position]` to a physical slot index. Shared prefixes point to the same indices (10, 11).

Req ID	Pos 0	Pos 1	Pos 2	Pos 3
R1	10	11	12	20
R2	10	11	12	25
R3	10	11	40	41

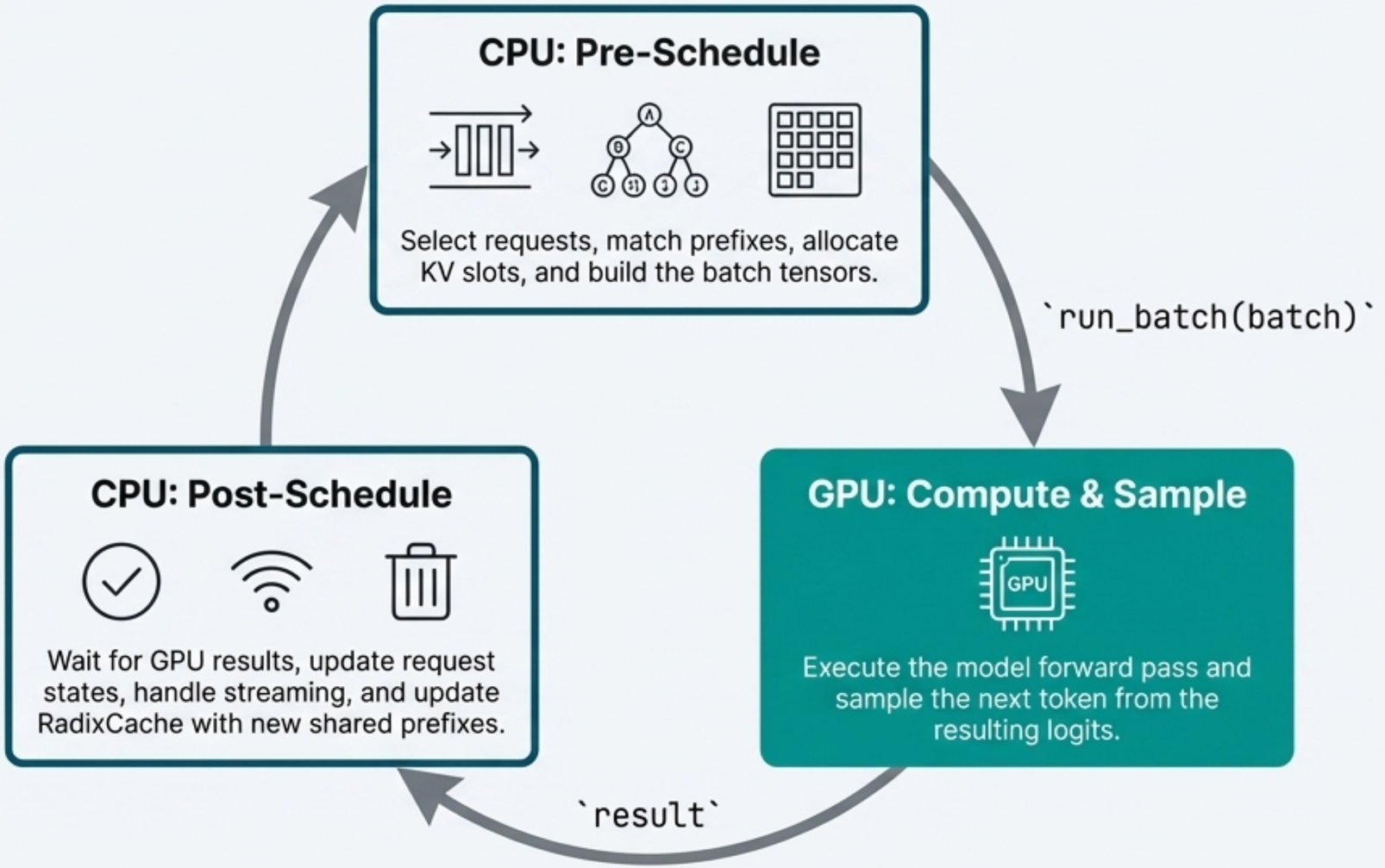
Layer 3: TokenToKVPool (Physical VRAM)

The actual GPU memory where KV Tensors reside. PagedAttention allows the GPU to gather these non-contiguous blocks using the indices from `ReqToTokenPool`.

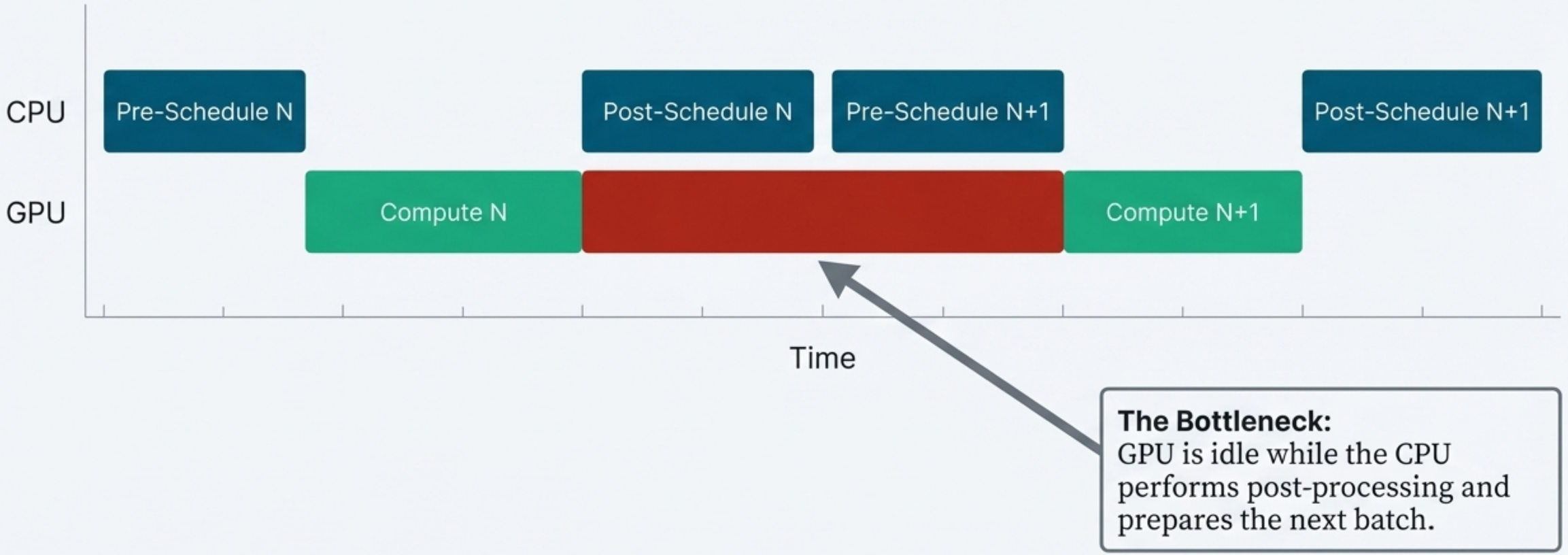


The Normal Scheduler: A Sequential, Blocking Workflow

The standard scheduling loop processes batches in a strictly sequential, blocking manner. The CPU prepares a batch, hands it to the GPU, and waits for the result before starting the next cycle.



The GPU CoPU Computes, The CPU Waits



The Bottleneck is Python I/O, Not Scheduling Logic

Key Insight

Profiling reveals that the core scheduling algorithms are fast. **The significant overhead comes from Python's heavy I/O and data preparation tasks.**

Pre-Schedule Costs

Building complex input tensors, preparing sampling metadata.

Post-Schedule Costs

Detokenisation and checking for stop conditions (EOS tokens).

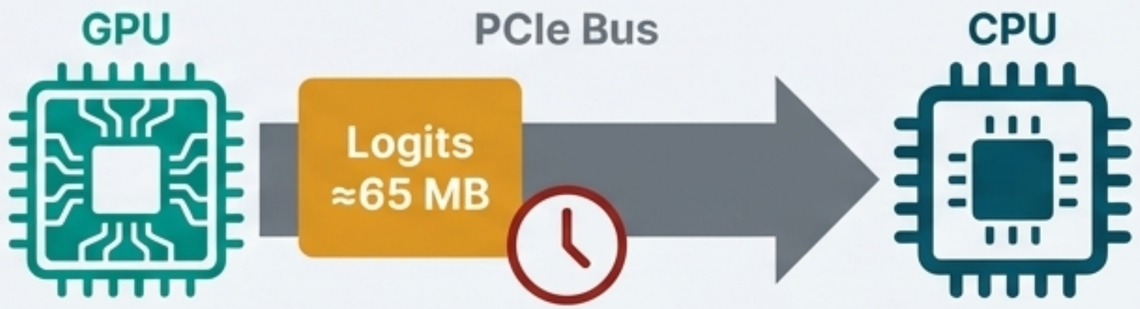
Why not just move sampling to the CPU?

For a batch size of 256 and a vocab size of 128,000 (Llama-3), the logits tensor is `[256, 128000]`.

At bf16 (2 bytes/value), this is **≈65 MB of data per step.**

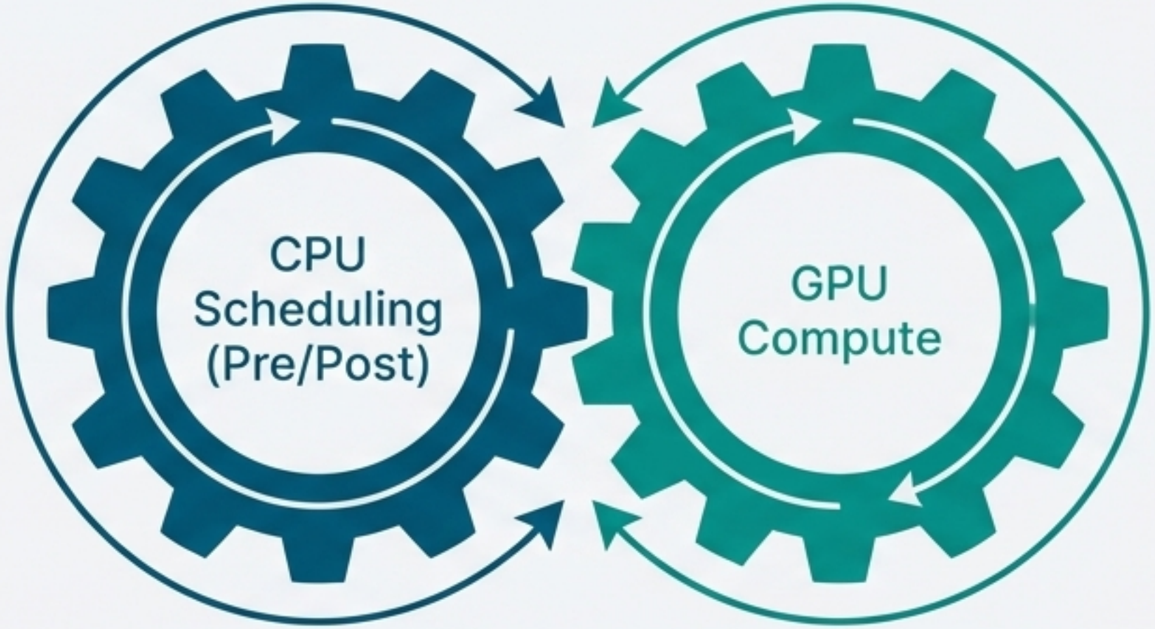
Transferring this over PCIe adds more latency than the inference step itself.

Sampling must happen on the GPU, but in the Normal Scheduler, the CPU still blocks waiting for the resulting token IDs.



The Breakthrough: Hiding CPU Overhead Behind GPU Computation

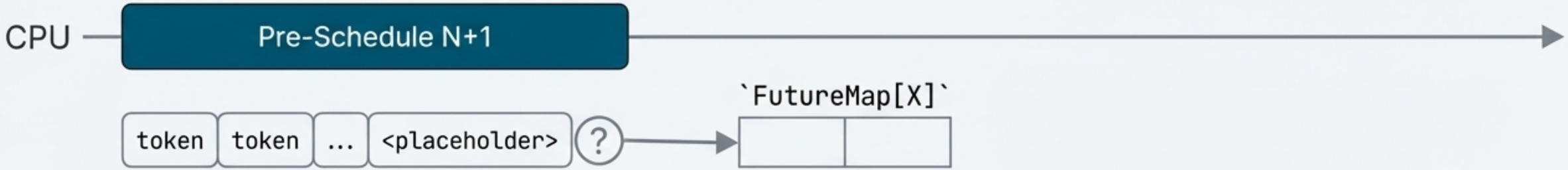
The Core Idea: Instead of making the CPU faster or scheduling less often (like multi-step scheduling, which loses flexibility), SGLang's Overlap Scheduler masks the CPU's work by performing it in parallel with the GPU's work.



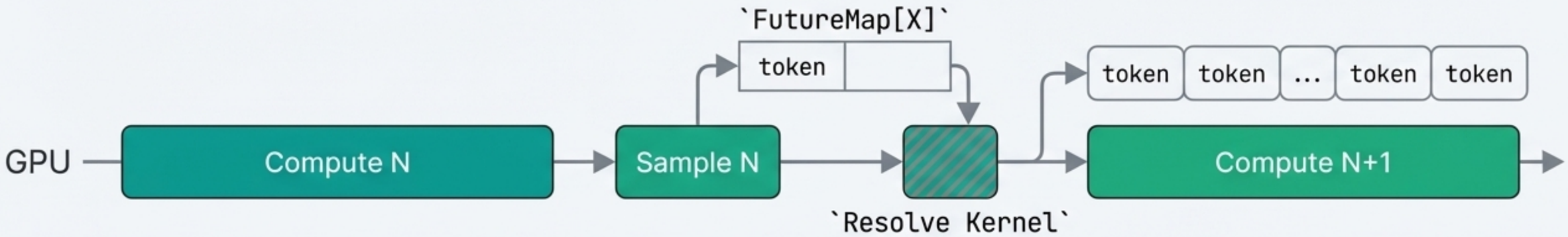
Key Principle: Decouple the control flow dependency (the CPU can launch the next batch) from the data dependency (it doesn't need the result of the current batch to do so).

The Mechanism: Symbolic Linking with a FutureMap

Overlap is possible because the CPU can prepare batch N+1 without knowing the actual output token from batch N. It uses a placeholder system.

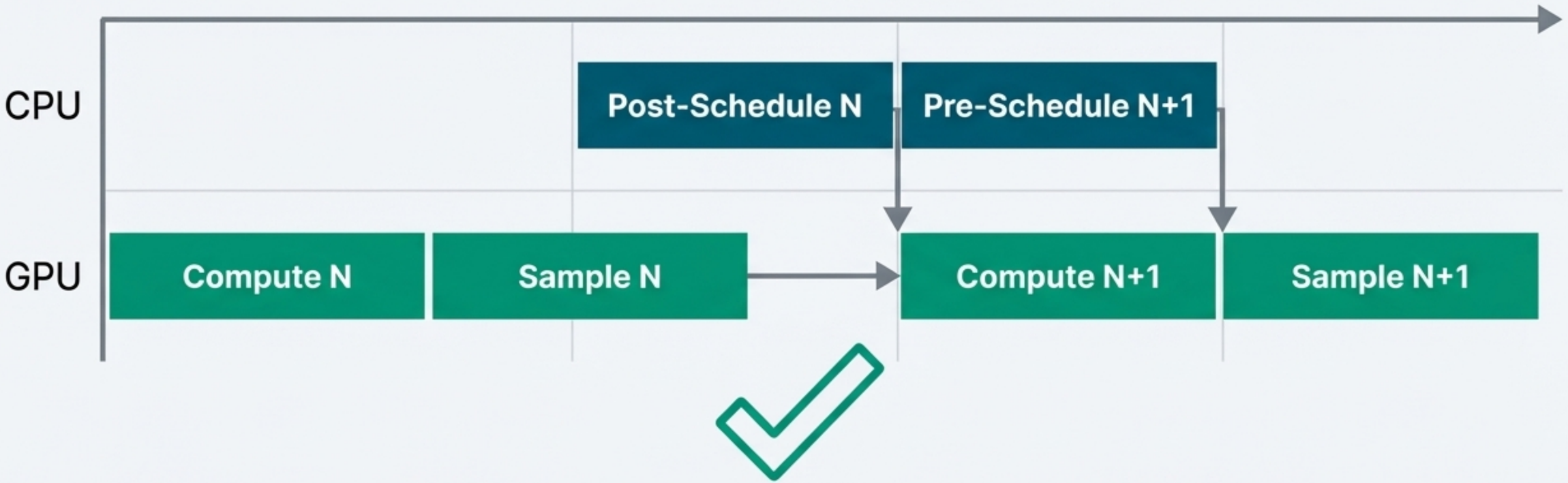


The CPU reserves a slot in a GPU memory buffer (`FutureMap`) and tells the N+1 batch to read its input from that future location.



Because GPU streams are FIFO, the `Resolve Kernel` for N+1 is guaranteed to run after `Sample N` has completed, ensuring the data is ready.

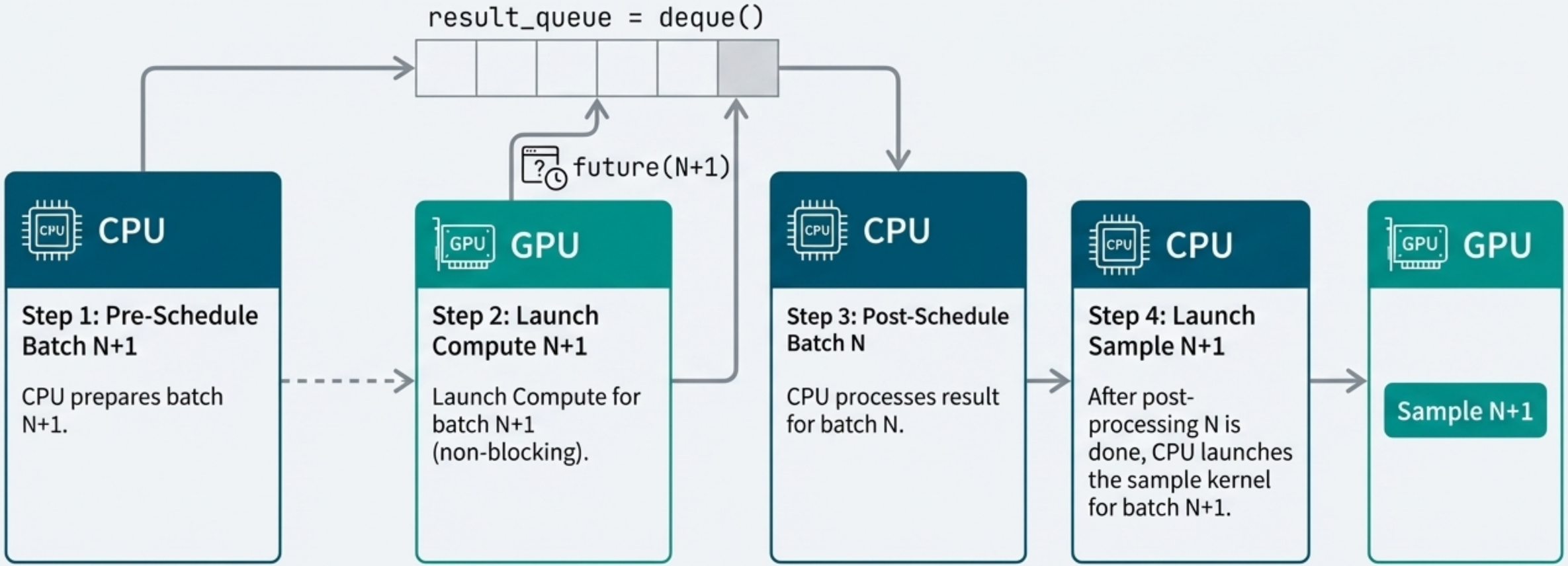
The Result: A Perfectly Pipelined System



By decoupling the launch of a batch from the result of the previous one, the GPU pipeline remains full, effectively achieving zero scheduling overhead.

The Overlap Event Loop in Action

The overlap event loop orchestrates this parallelism using a result queue and non-blocking calls.



The CPU is **always working on the results of batch N** while the GPU is computing **batch N+1**.

A Comparison of Scheduling Strategies

Strategy	CPU Overhead	Flexibility (EOS, Preemption)	Implementation Complexity
Normal Scheduling	High (blocking)	High (single-step control)	Low
Multi-step Scheduling	Low (amortised)	Low (delayed response)	Medium
Overlap Scheduling	Zero (hidden)	High (single-step control)	High

Overlap Scheduling offers the best of both worlds: the low overhead of multi-step scheduling combined with the fine-grained control and responsiveness of single-step scheduling, at the cost of higher implementation complexity using CUDA streams and future mapping.

The SGLang Scheduler: From Serial Dependency to Parallel Ingenuity

Core Insight Recap

- SGLang transforms a serial dependency (`CPU waits for GPU` in JetBrains Mono) into a fully asynchronous, pipelined workflow.
- It achieves this by separating the *control flow* (launching the next step) from the *data flow* (accessing the token result), using a system of symbolic placeholders and lazy resolution on the GPU.

The Impact

- The result is a "zero-overhead" scheduler that maximises hardware utilisation, enabling higher throughput and lower latency for LLM inference.