From Scratch to Extreme, Boosting Service Throughput by Dozens of Times with Step-by-Step Optimization

Gems Guo, NVIDIA DevTech APAC | GTC Spring 2024

Background

- More and more developers are looking to migrate their workloads to GPU and build GPU-based services.
- Compared to CPU, GPU services can usually lower the cost while handling the same number of requests.
- However, this also poses greater challenges. When a single GPU server replaces multiple CPU servers, the load pressure will significantly increase, which can easily affect GPU utilization and cause performance issue.
- At GTC Spring 2023, my colleague and I shared our experiences in optimizing services for Information Retrieval scenarios. The title of our session is **CUDA Implementation and Optimization for Boolean Model of Information Retrieval** (S51435).
- We continued our research and found that these optimization ideas and methods can also be applied to other scenarios, such as RecSys and LLM.
- Today, I will use a more general GPU workload as an example to discuss with you how to improve the throughput with bounded latency, and how to address the performance issue and enhance the GPU utilization.

Agenda

- Test Model and Evaluation Metrics
- Enhancing Throughput with CUDA Optimization Techniques Step-by-Step
- Technical Tips
- Summary & Takeaways

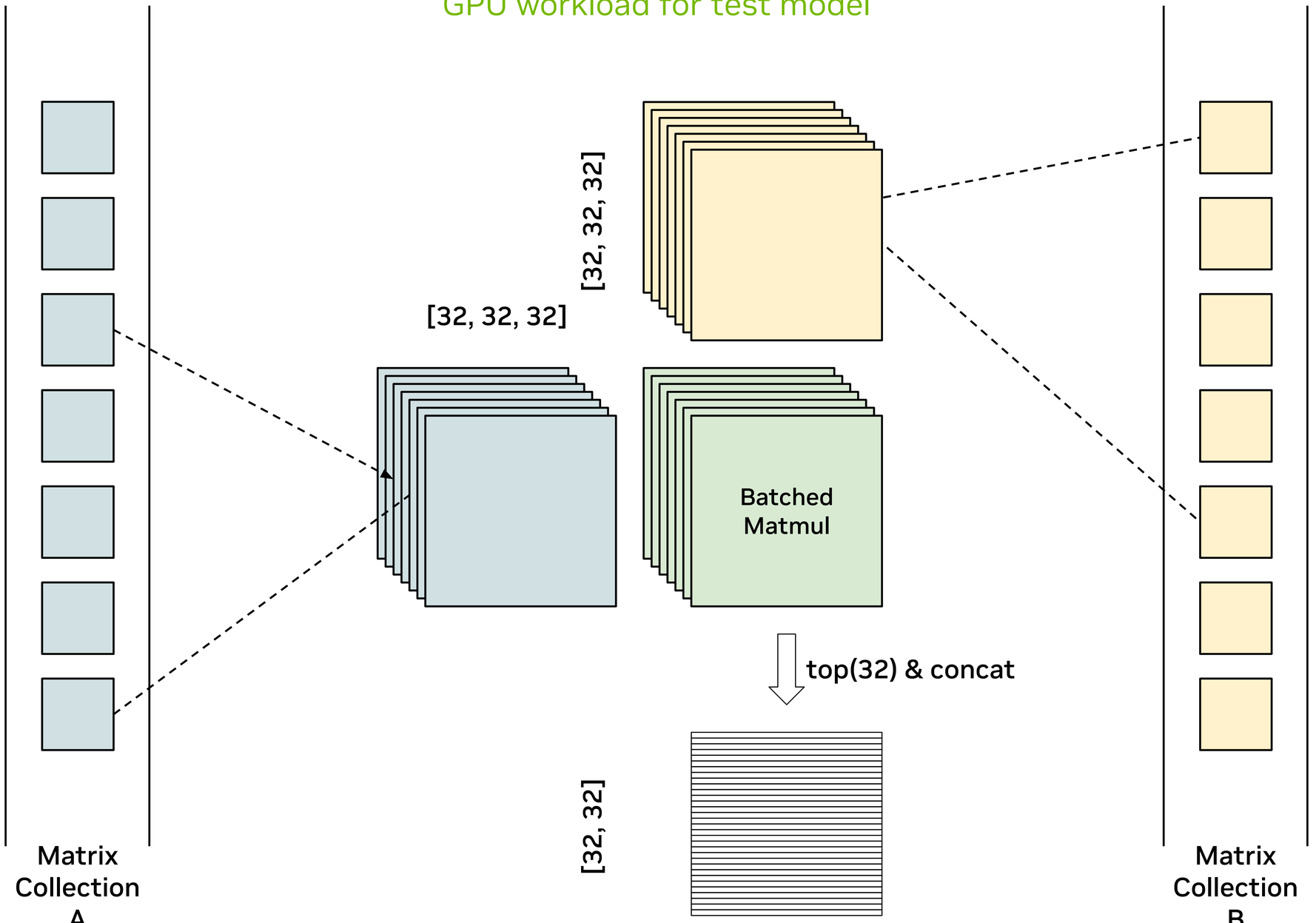
Test Model and Evaluation Metrics

Key features of test model

- The query’s main workload should run on GPU.
- Each query arrives independently, but the time intervals between queries follow a certain distribution.
- The query processing is asynchronous. This means that the system can start the next query without waiting for the previous one to finish, if there are enough resources.
- Each query has a relatively small workload, but the query frequency is high.

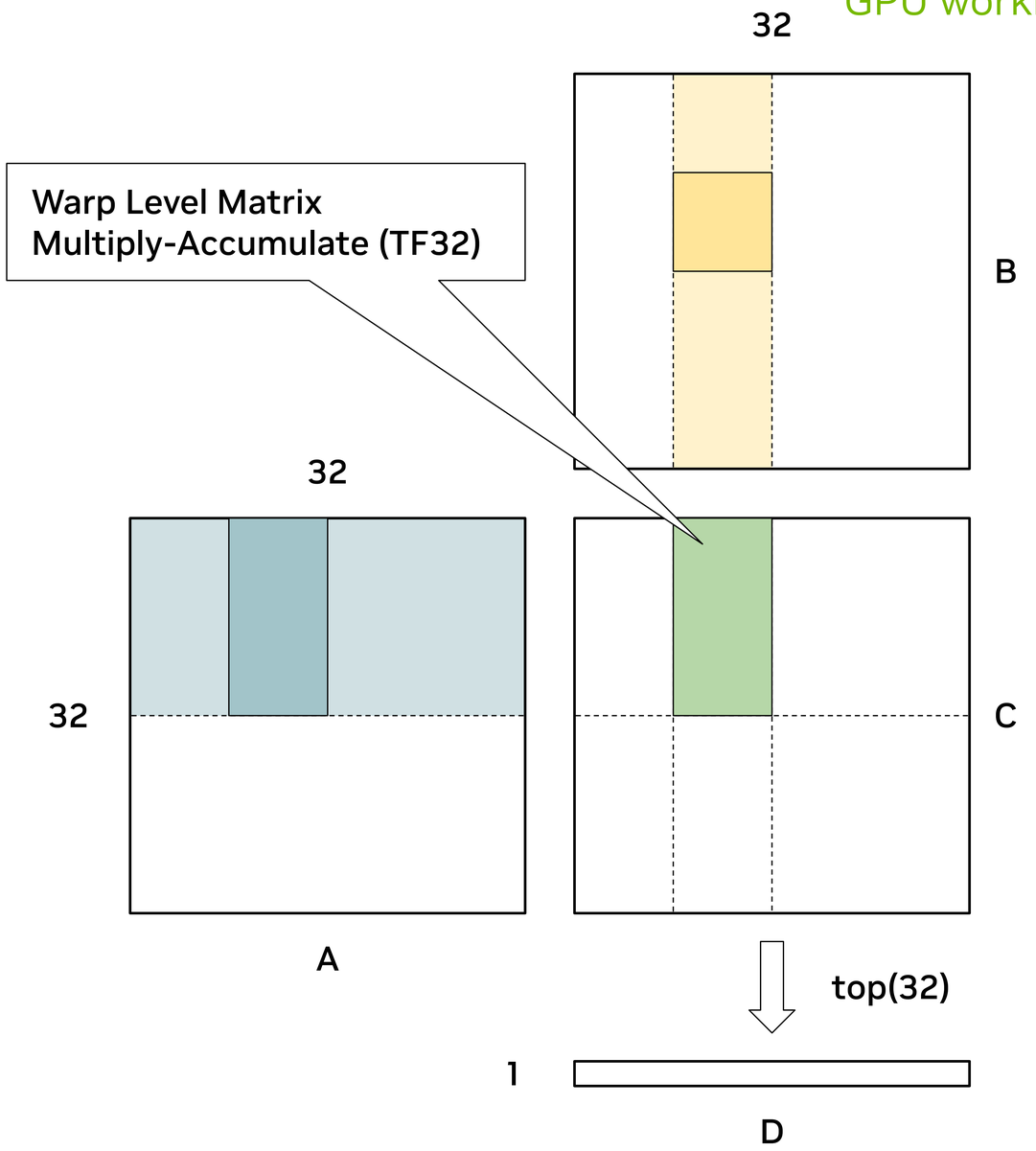
Test Model and Evaluation Metrics

GPU workload for test model



Test Model and Evaluation Metrics

GPU workload for test model



The pointer to the index list for the matrix A and B in their collection, the data type is int32, the shape is [32, 2]

The pointer to the weights of the matrix A collection and matrix B collection that reside in the global memory, the data type is float32

matmul<<<32, 128, 0, stream>>>

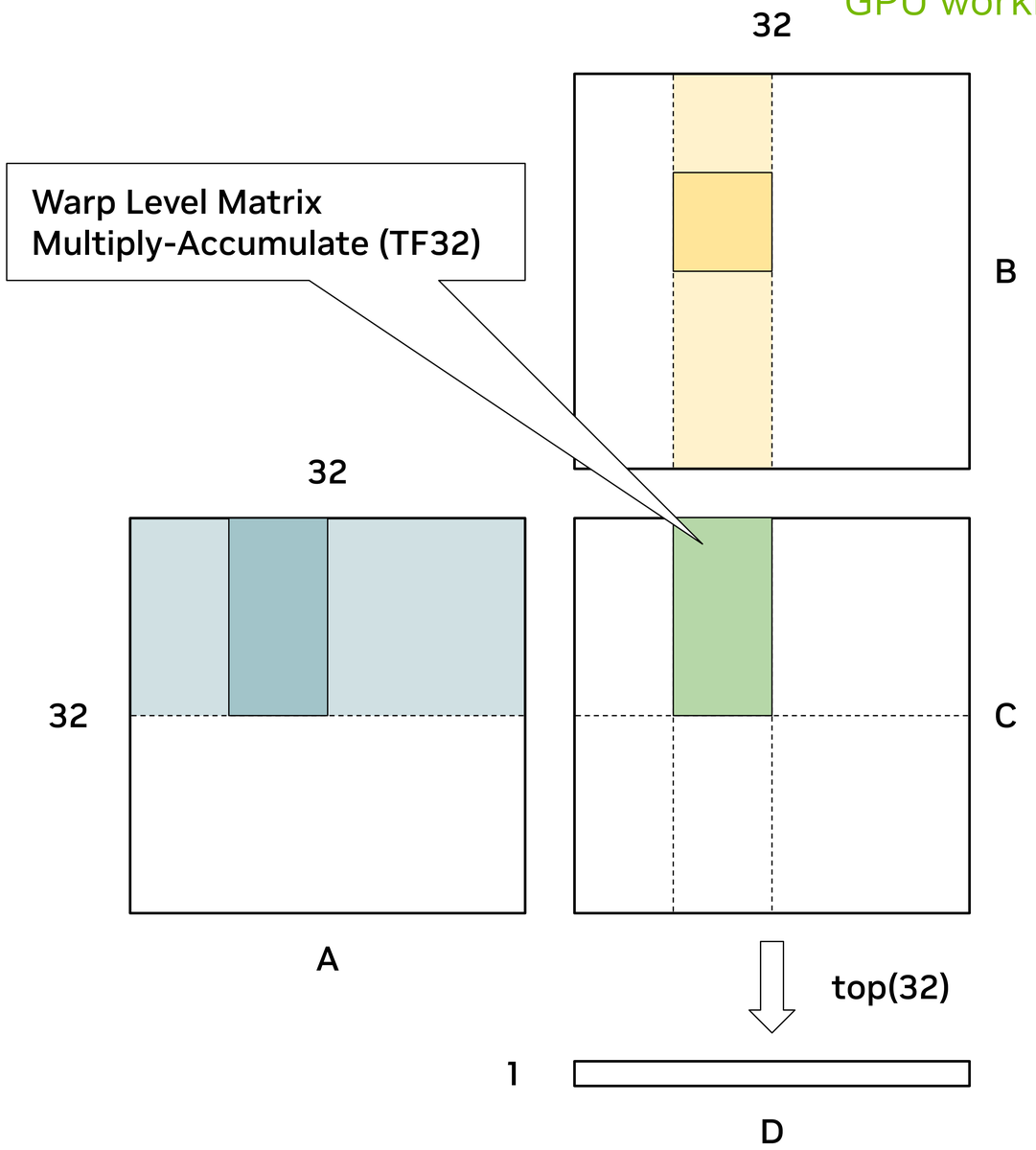
The pointer to the result matrix of matmul, the data type is float32, the shape is [32, 32, 32]

topk<<<32, 128, 0, stream>>>

The pointer to all the largest 32 weights after concating, the data type is float32, the shape is [32, 32]

Test Model and Evaluation Metrics

GPU workload for test model



matmul<<<...>>>

- Asynchronous Data Copies
- Warp Level Matrix Multiply-Accumulate (TF32)

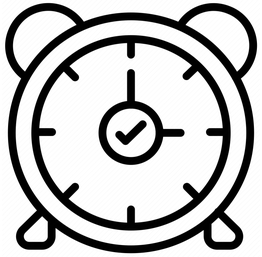
topk<<<...>>>

- Asynchronous Data Copies
- Warp Merge Sort via cub
- Warp Collectives

Test Model and Evaluation Metrics

Key indicators for evaluation

There are two important indicators for evaluating the implementation



Latency

The unit is milliseconds

VS

Throughput

The unit is QPS (queries per seconds)

Test Model and Evaluation Metrics

Metrics excerpted from the introduction of NVIDIA Triton Inference Server

How Throughput is Calculated

Perf Analyzer calculates throughput to be the total number of requests completed during a measurement, divided by the duration of the measurement, in seconds.

How Latency is Calculated

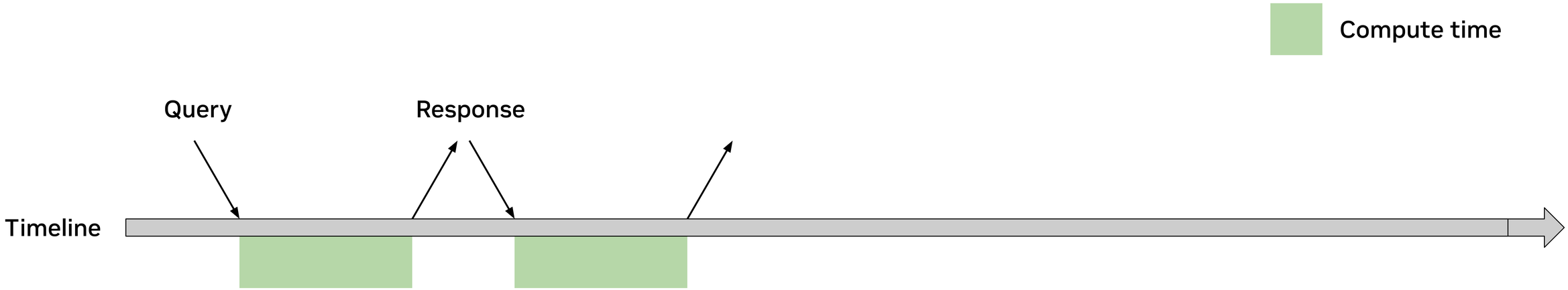
The server latency measures the total time from when the request is received at the server until when the response is sent from the server. Because of the HTTP libraries used to implement the server endpoints, total server latency is typically more accurate for HTTP requests as it measures time from the first byte received until last byte sent. For HTTP the total server latency is broken-down into the following components:

- ***queue*** - The average time spent in the inference schedule queue by a request waiting for an instance of the model to become available.
- ***compute*** - The average time spent performing the actual inference, including any time needed to copy data to/from the GPU.
- ***overhead*** - The average time spent in the endpoint that cannot be correctly captured in the send/receive time with the way the gRPC and HTTP libraries are structured.

https://github.com/triton-inference-server/client/blob/main/src/c%2B%2B/perf_analyzer/docs/measurements_metrics.md

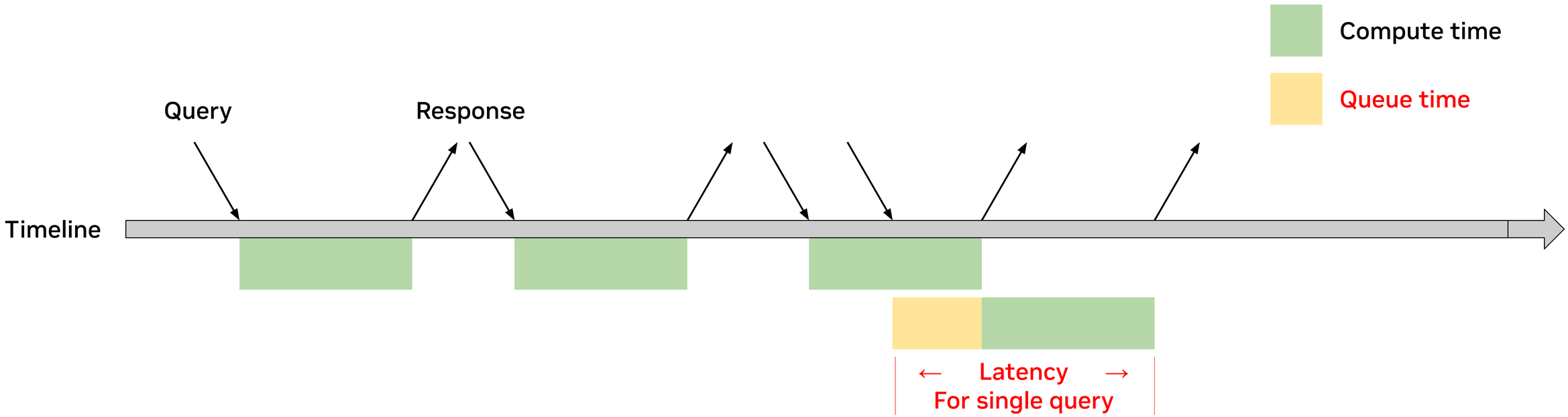
Test Model and Evaluation Metrics

Understanding the indicators on the timeline



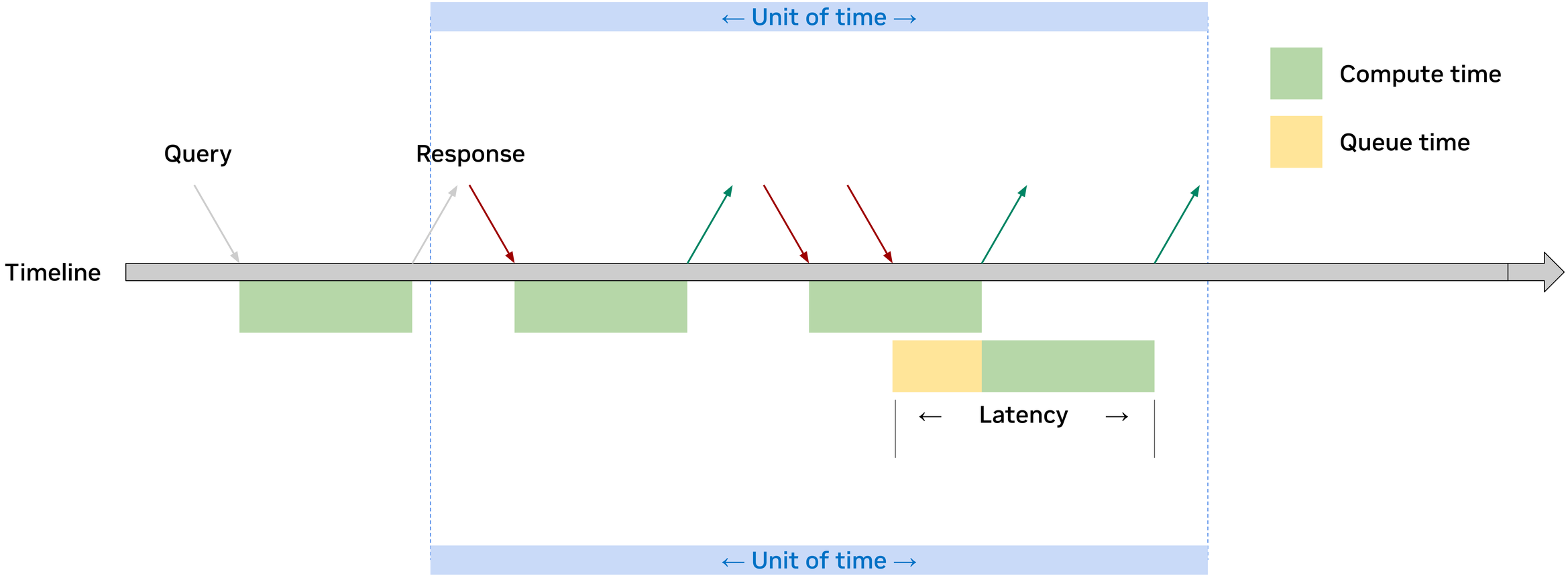
Test Model and Evaluation Metrics

Understanding the indicators on the timeline



Test Model and Evaluation Metrics

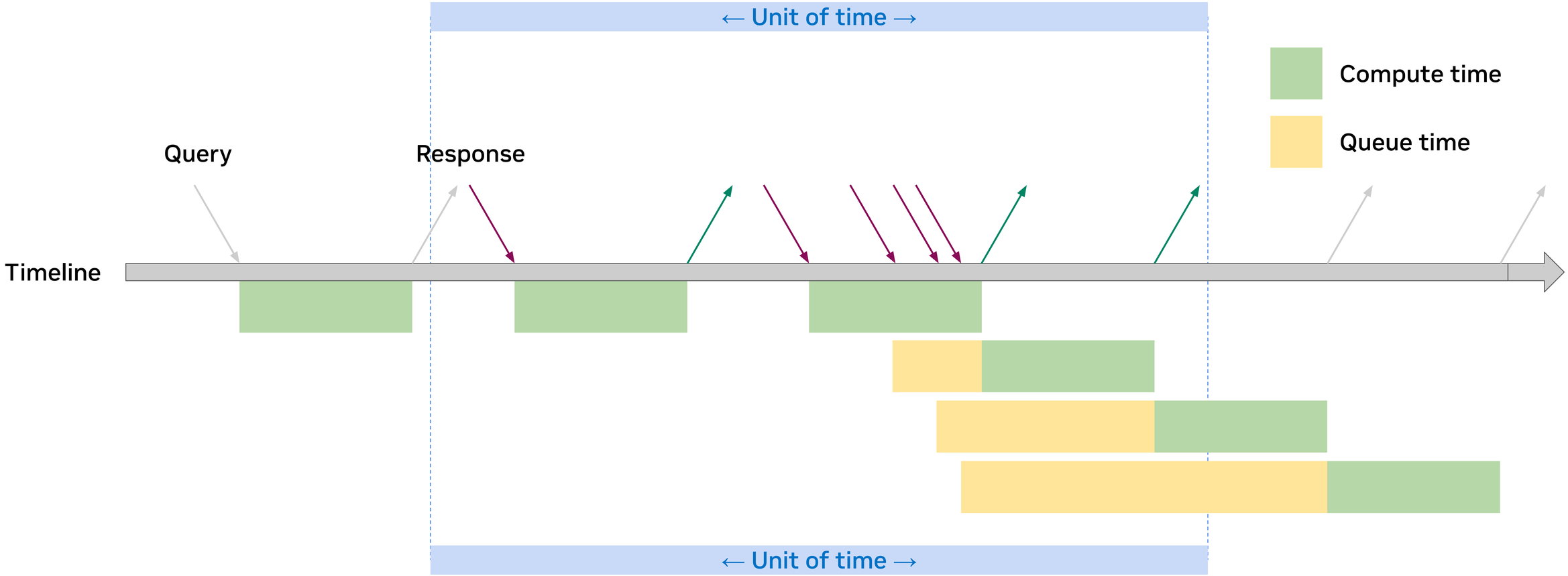
Understanding the indicators on the timeline



Incoming throughput is the number of queries **arriving** per unit of time.
Measured throughput is the number of queries **processed** per unit of time.

Test Model and Evaluation Metrics

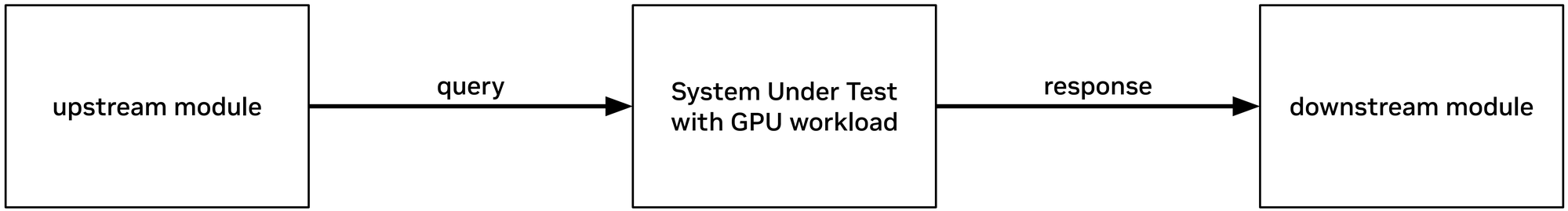
Understanding the indicators on the timeline



The measured throughput is less than or equal to the the incoming throughput.

Test Model and Evaluation Metrics

Explain module relationships and how to get the indicators

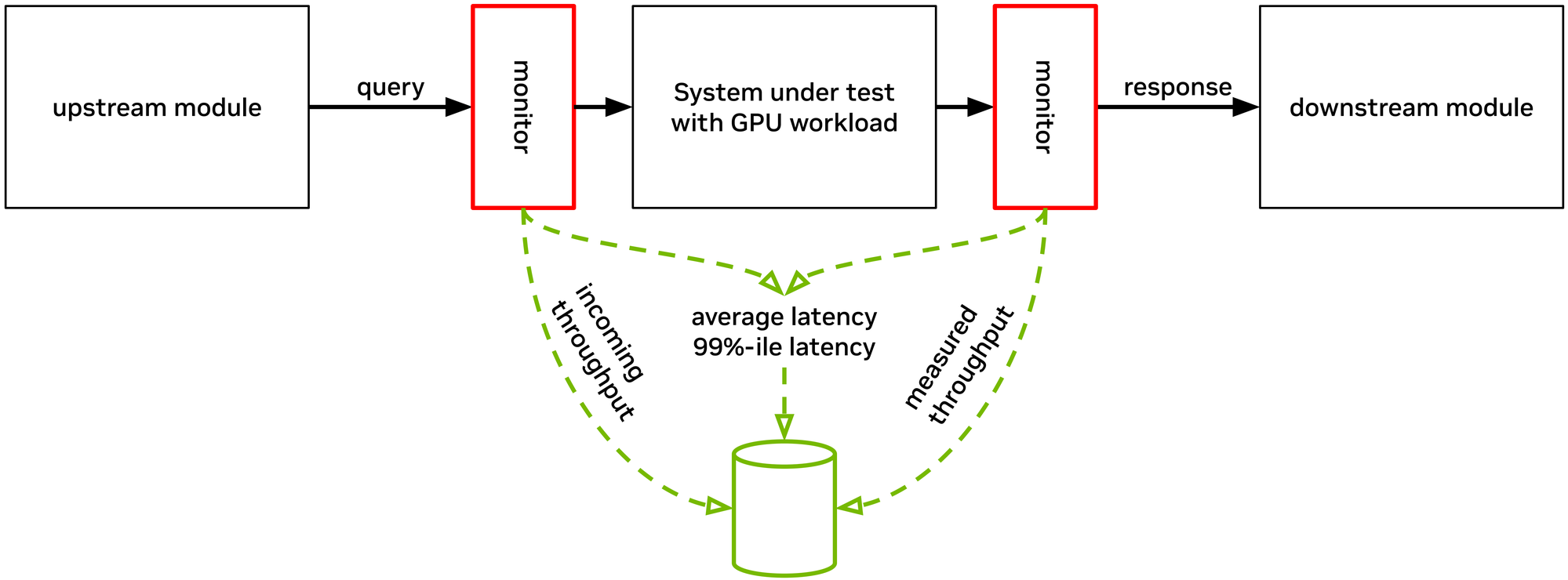


Communication between the upstream/downstream module and the System Under Test can be in 2 ways:

- Intra-process communication, via function calls
- Inter-process communication, via HTTP/gRPC/WebSocket/COM/POSIX, etc.

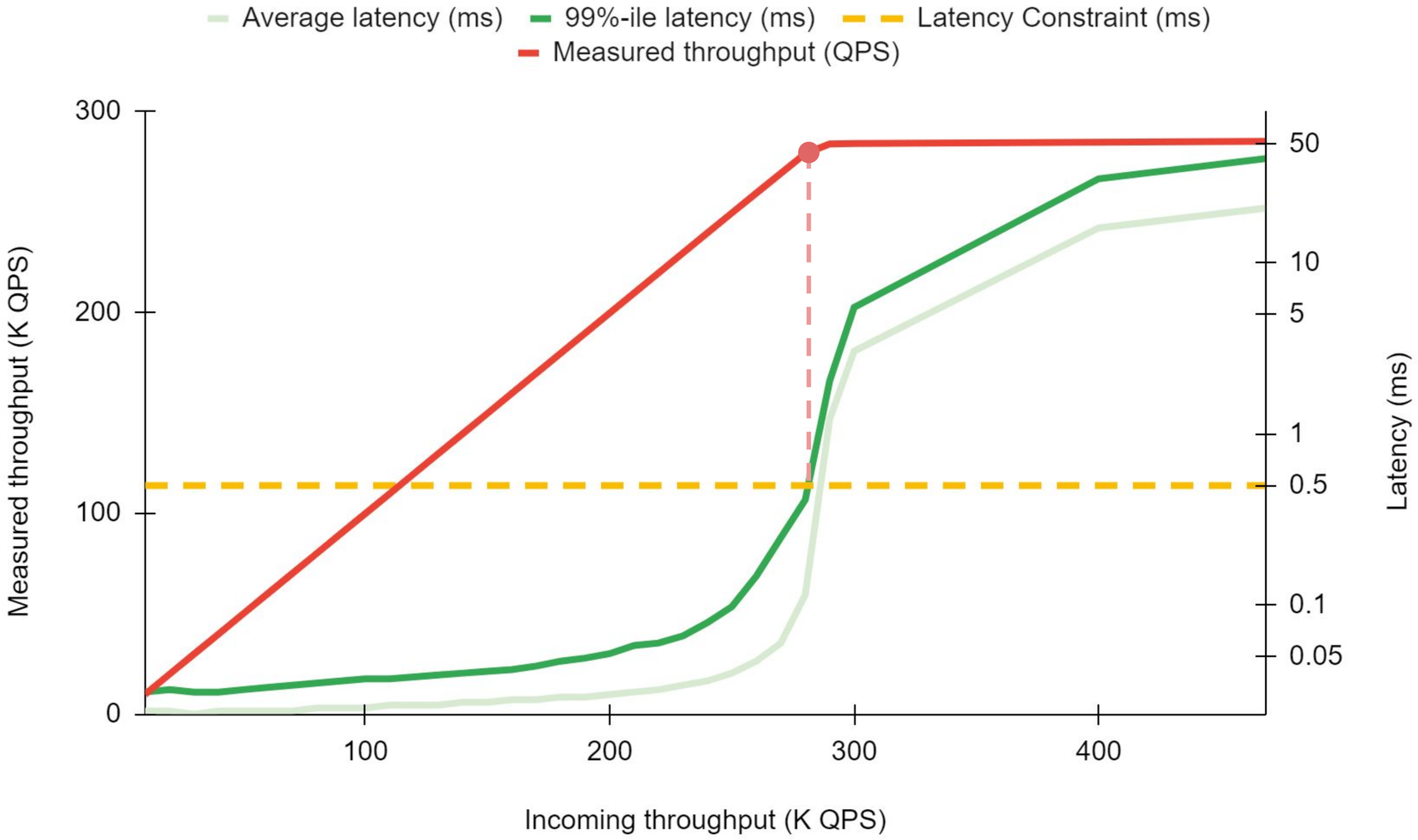
Test Model and Evaluation Metrics

Explain module relationships and how to get the indicators



Test Model and Evaluation Metrics

The relationship between latency and incoming throughput and measured throughput



Test Model and Evaluation Metrics

Hardware environment

NVIDIA L40

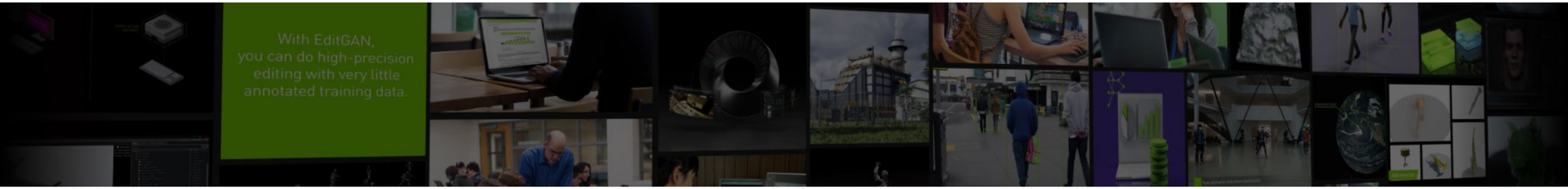
Architecture: Ada Lovelace
Memory: 48GB GDDR6
Memory clock: 9,001MHz
Peak memory bandwidth: 864 GB/s
GPU boost clock: 2,490MHz
# of CUDA Cores: 18,176
# of Tensor Cores: 568
FP32 TFLOPS: 90.5
TF32 TFLOPS: 90.5
PCIe interface: PCIe Gen4 ×16

AMD EPYC 7313P

Architecture: Zen3
# of Sockets: 1
# of CCD (Core Chiplet Die): 4
# of CCX (Core Complex) per CCD: 1
# of CPU Cores per CCX: 4
# of CPU Cores: 16
# of Threads: 32
Max. boost clock: Up to 3.7GHz
L3 cache: 128MB
PCIe version: PCIe Gen4 × 128

Test Model and Evaluation Metrics

Software environment



OS: Ubuntu 22.04

CUDA Toolkit Version: 12.1

CUDA Driver Version: 530.30.02

CPU and GPU are locked to their boost clock frequency

The time intervals of queries follow a Poisson Distribution

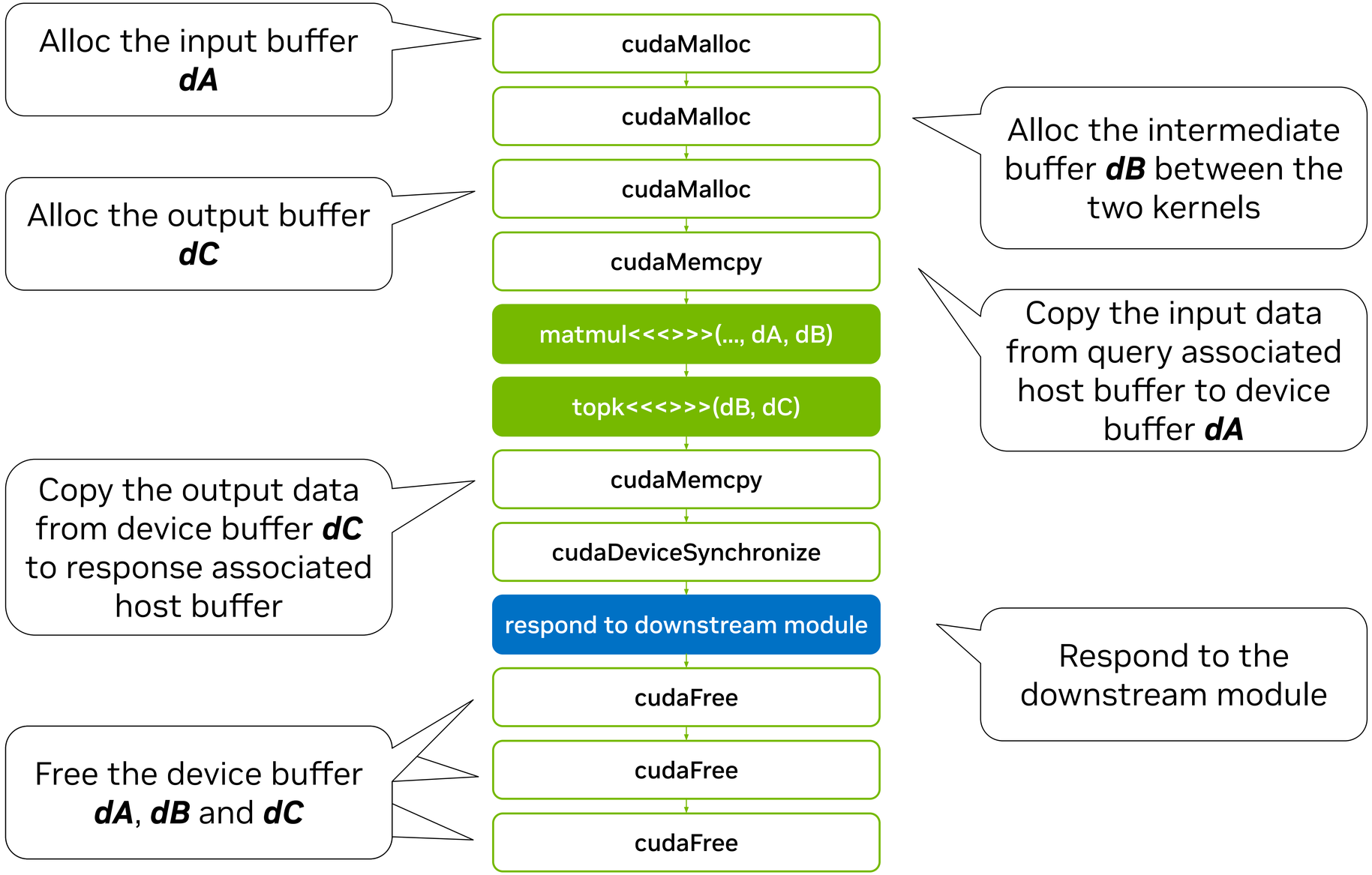
Latency constraint: 0.5ms for 99%-ile latency

① Single CPU Thread & Single CUDA Stream

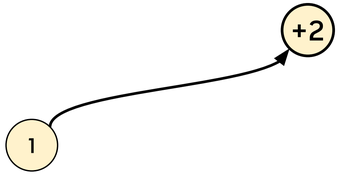
1

Single CPU Thread & Single CUDA Stream

Implementation

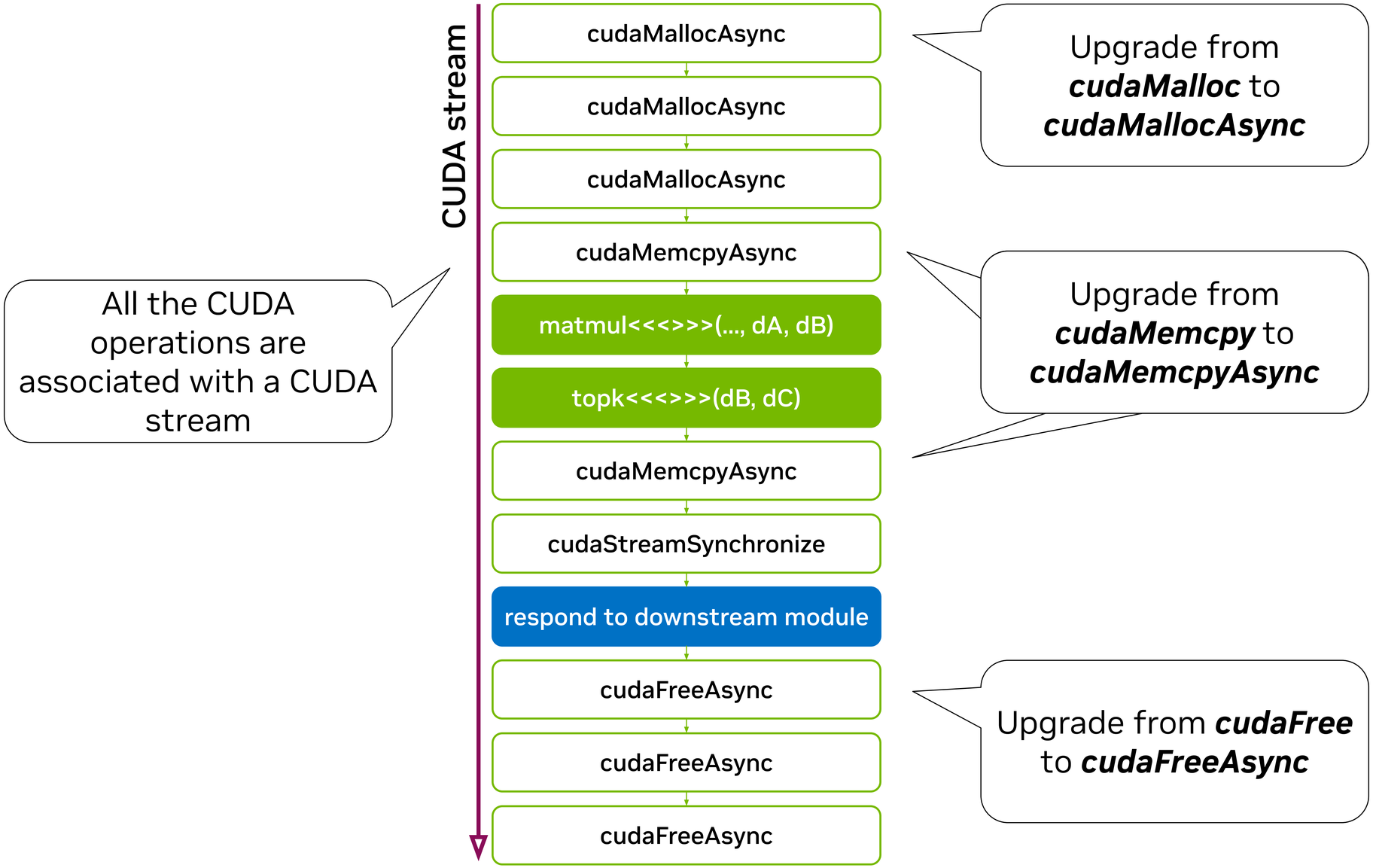


① Single CPU Thread & Single CUDA Stream
② **Stream Ordered Memory Allocator**



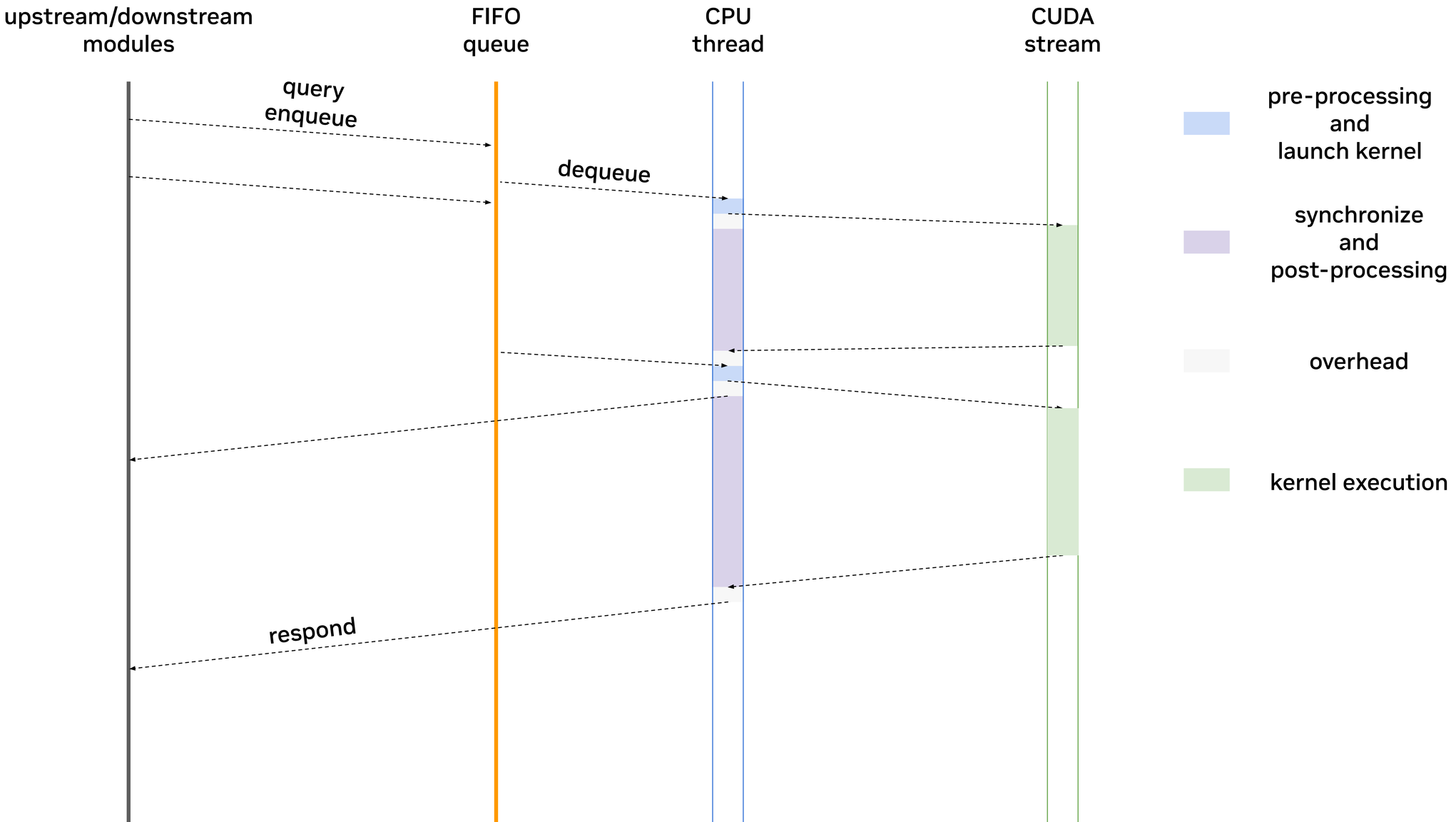
Stream Ordered Memory Allocator

Implementation

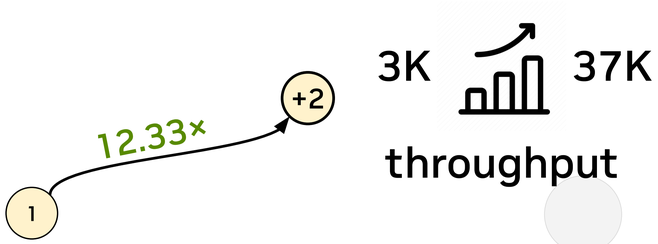


Stream Ordered Memory Allocator

Workflow

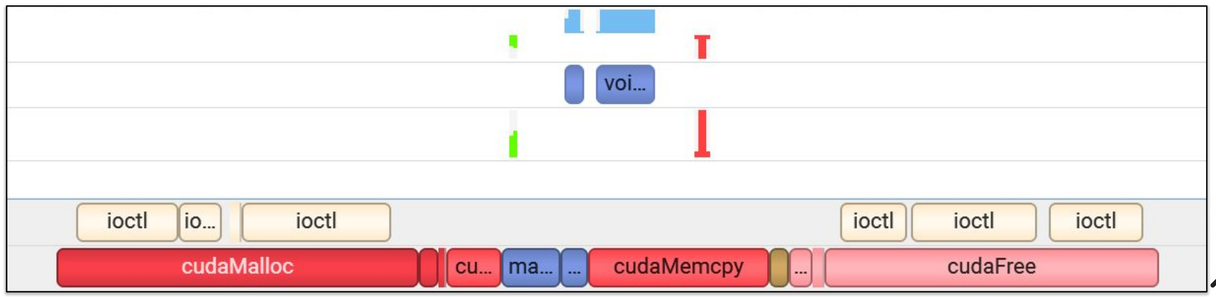


① Single CPU Thread & Single CUDA Stream
② **Stream Ordered Memory Allocator**

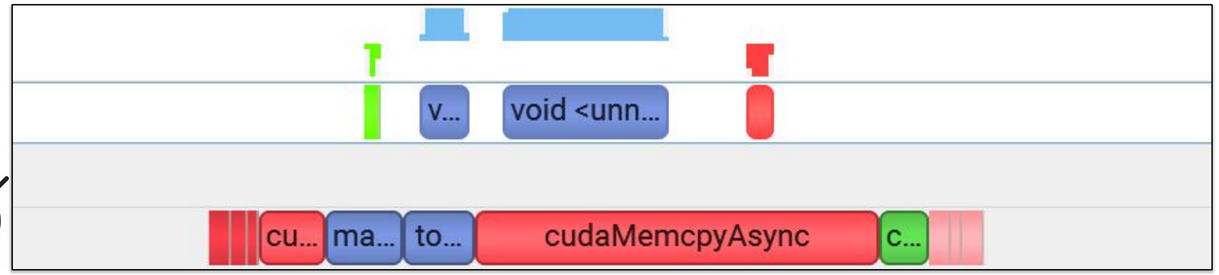


Stream Ordered Memory Allocator

Comparison



VS



Before SOMA

After SOMA

Call to cudaMalloc
- Memory allocation

Begins: 26.6619s
Ends: 26.662s (+48.639 µs)
Return value: 0
Correlation ID: 890682

Call to cudaFree
- Memory deallocation

Begins: 26.662s
Ends: 26.6621s (+44.459 µs)
Return value: 0
Correlation ID: 890692

Call to cudaMallocAsync
- Memory allocation

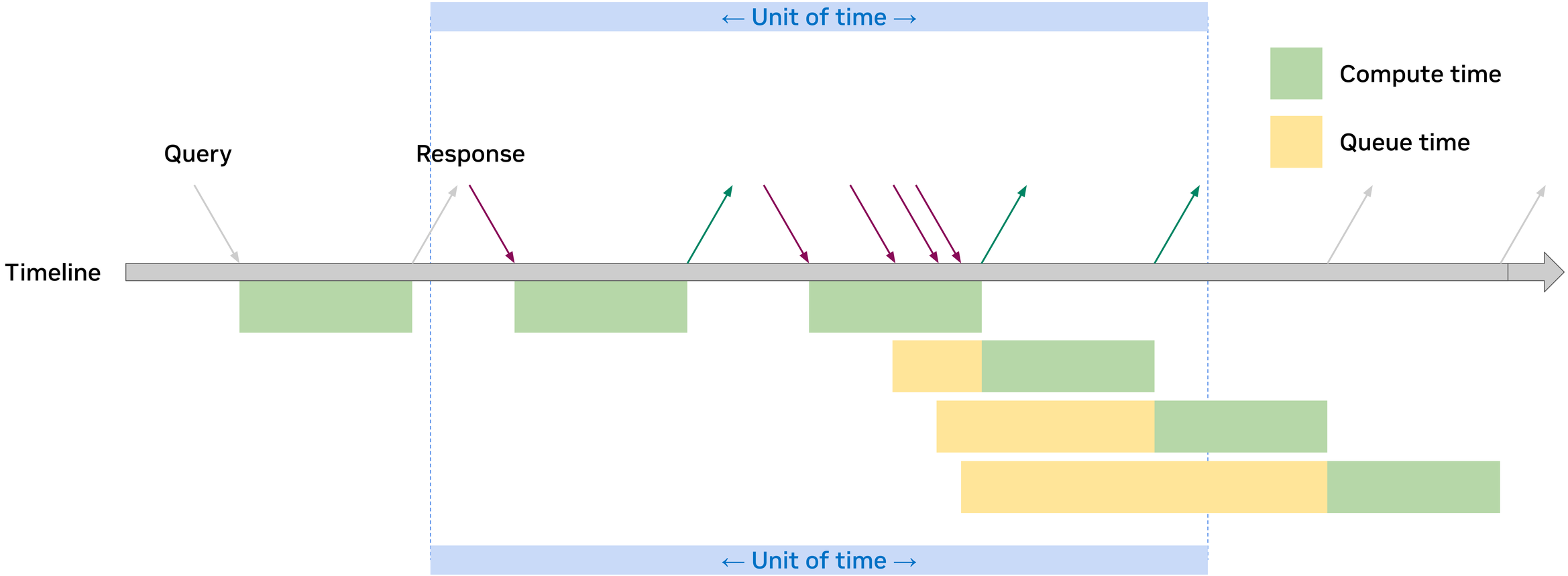
Begins: 7.88915s
Ends: 7.88915s (+810 ns)
Return value: 0
Correlation ID: 703738

Call to cudaFreeAsync
- Memory deallocation

Begins: 7.88918s
Ends: 7.88918s (+660 ns)
Return value: 0
Correlation ID: 703748

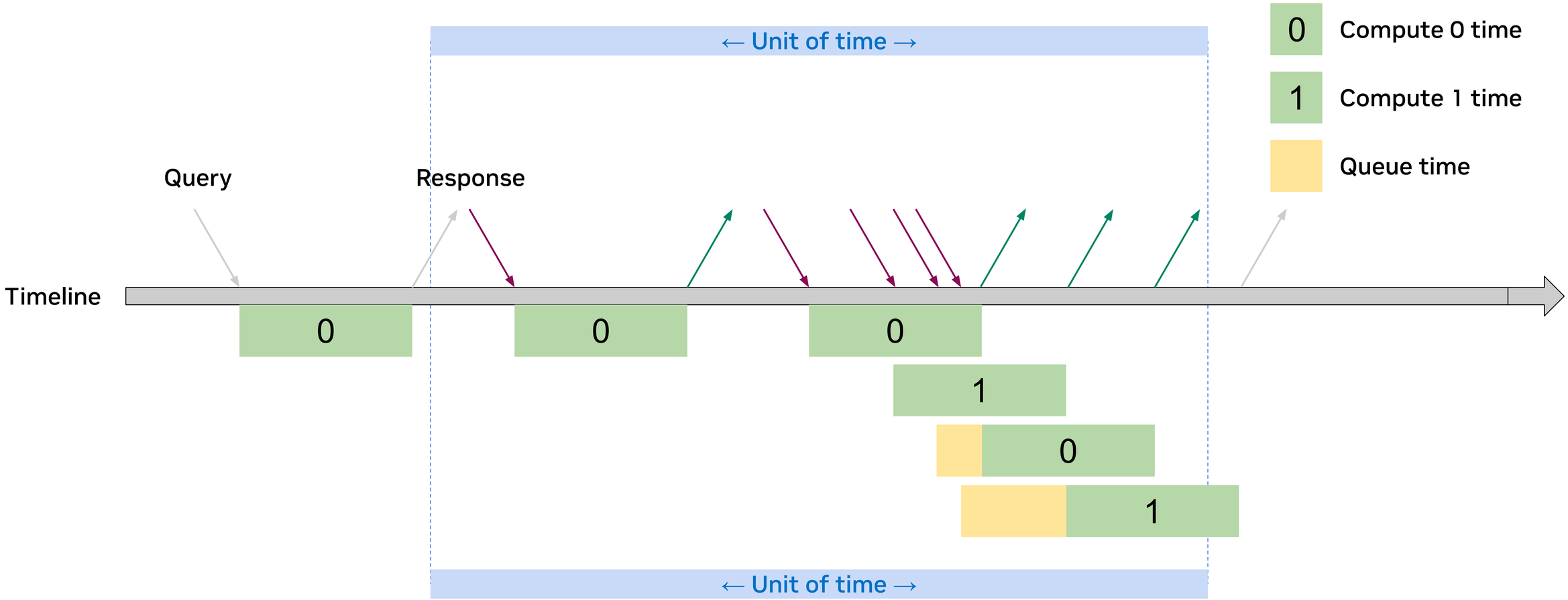
Test Model and Evaluation Metrics

Understanding the indicators on the timeline



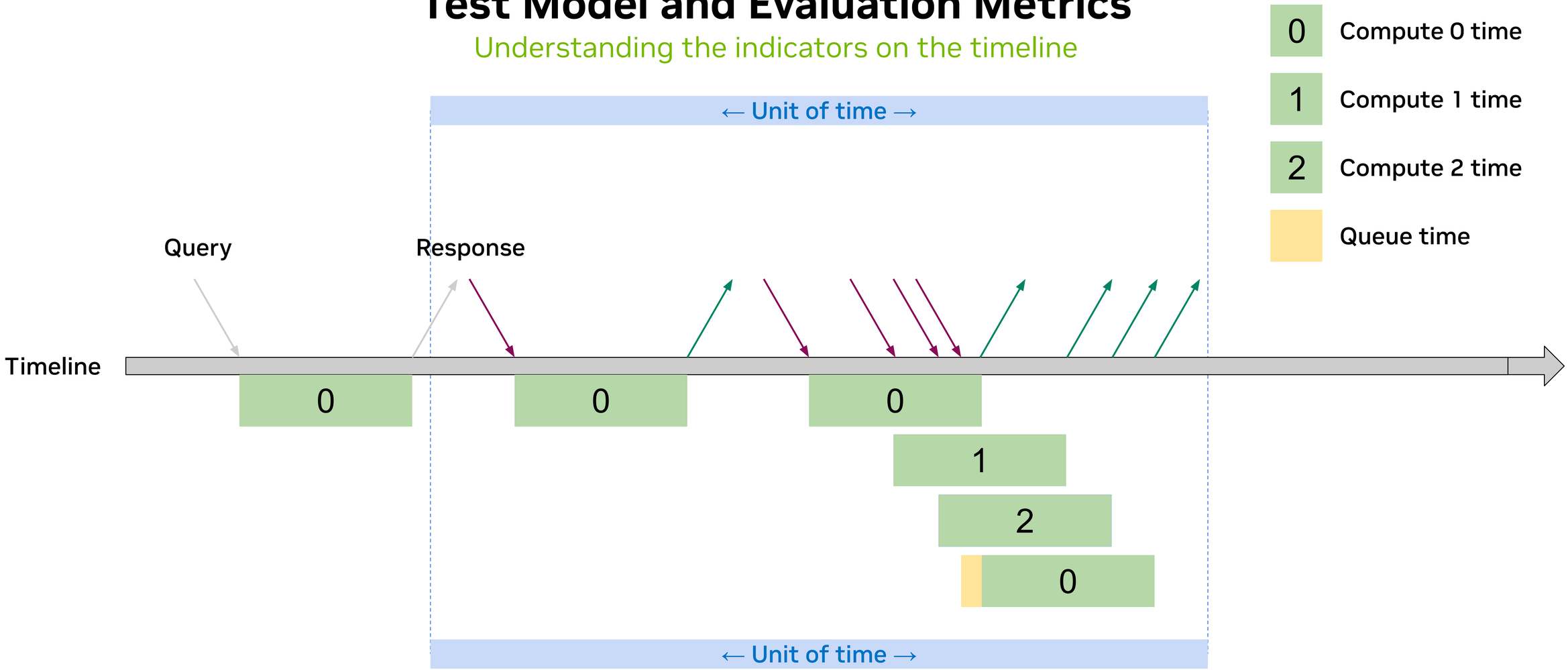
Test Model and Evaluation Metrics

Understanding the indicators on the timeline



Test Model and Evaluation Metrics

Understanding the indicators on the timeline



Test Model and Evaluation Metrics

Find the maximum possible parallelism

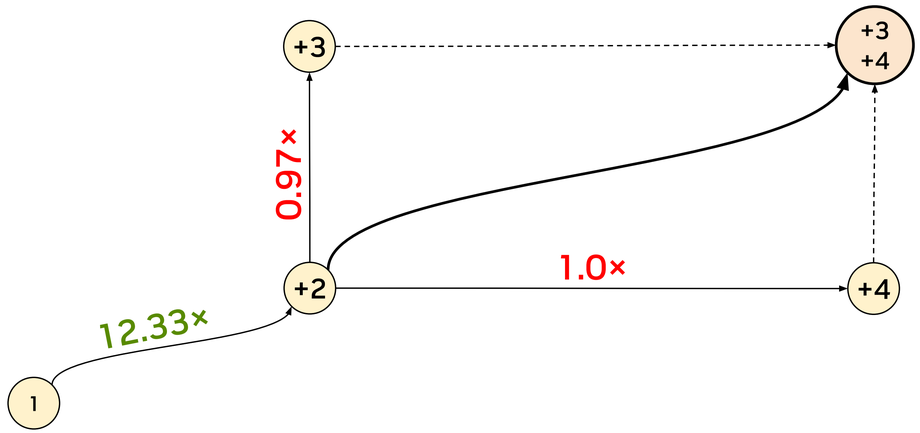
We can compute the SM utilization for the kernel.

The compiler output and the occupancy calculator in Nsight Compute show that the maximum number of allocatable blocks per SM is **10**.

L40 has **142 SMs**, so the total number of allocatable blocks is 142×10=**1420**, and one kernel launches **32** blocks, so the total number of kernels that can reside simultaneously is 1420÷32=44.375, rounded down to **44**.

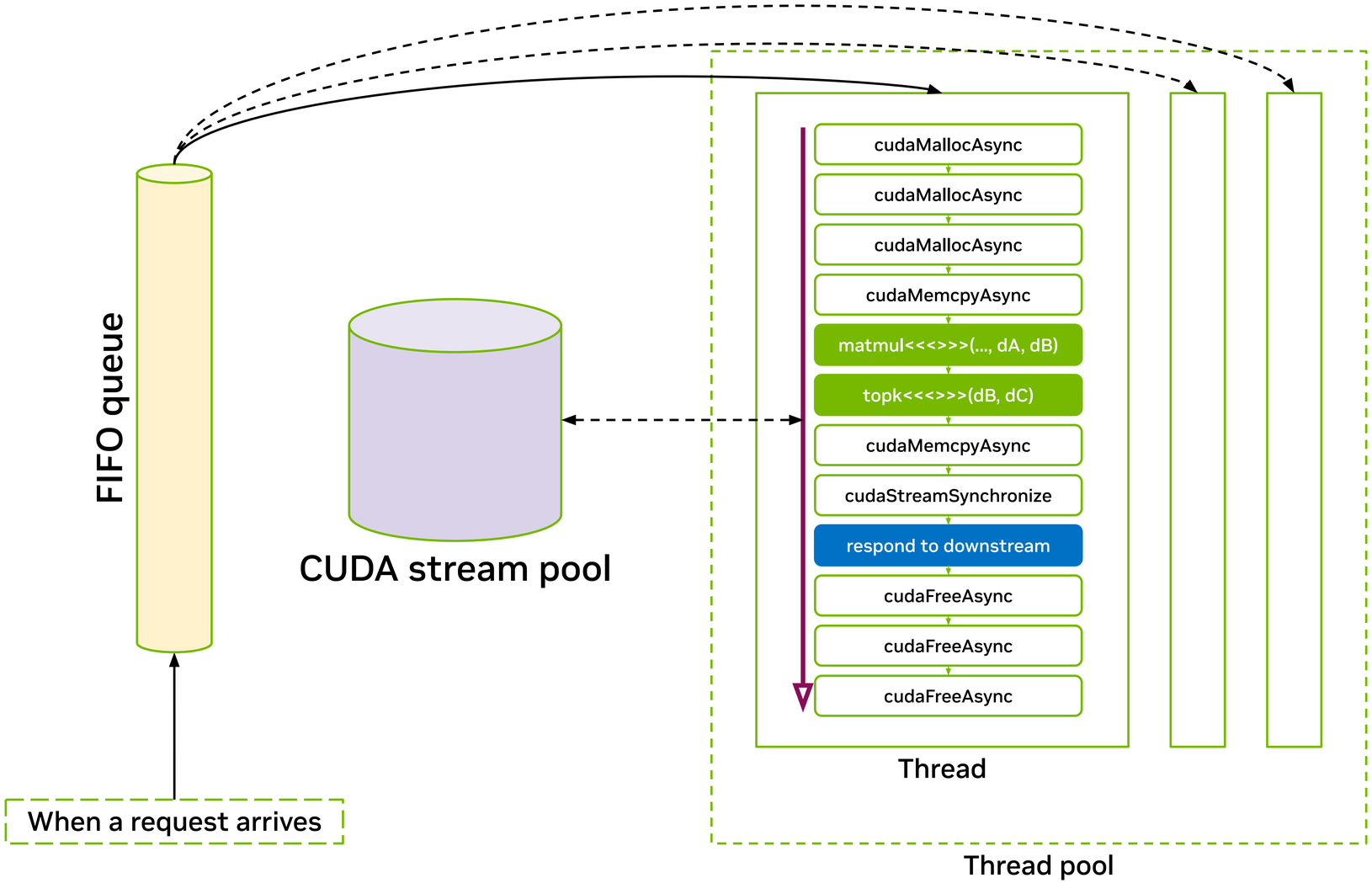
From this, we can infer that the maximum parallelism achievable is 44.

① Single CPU Thread & Single CUDA Stream
② Stream Ordered Memory Allocator
③ **Limited CPU Threads**
④ **Limited CUDA Streams**



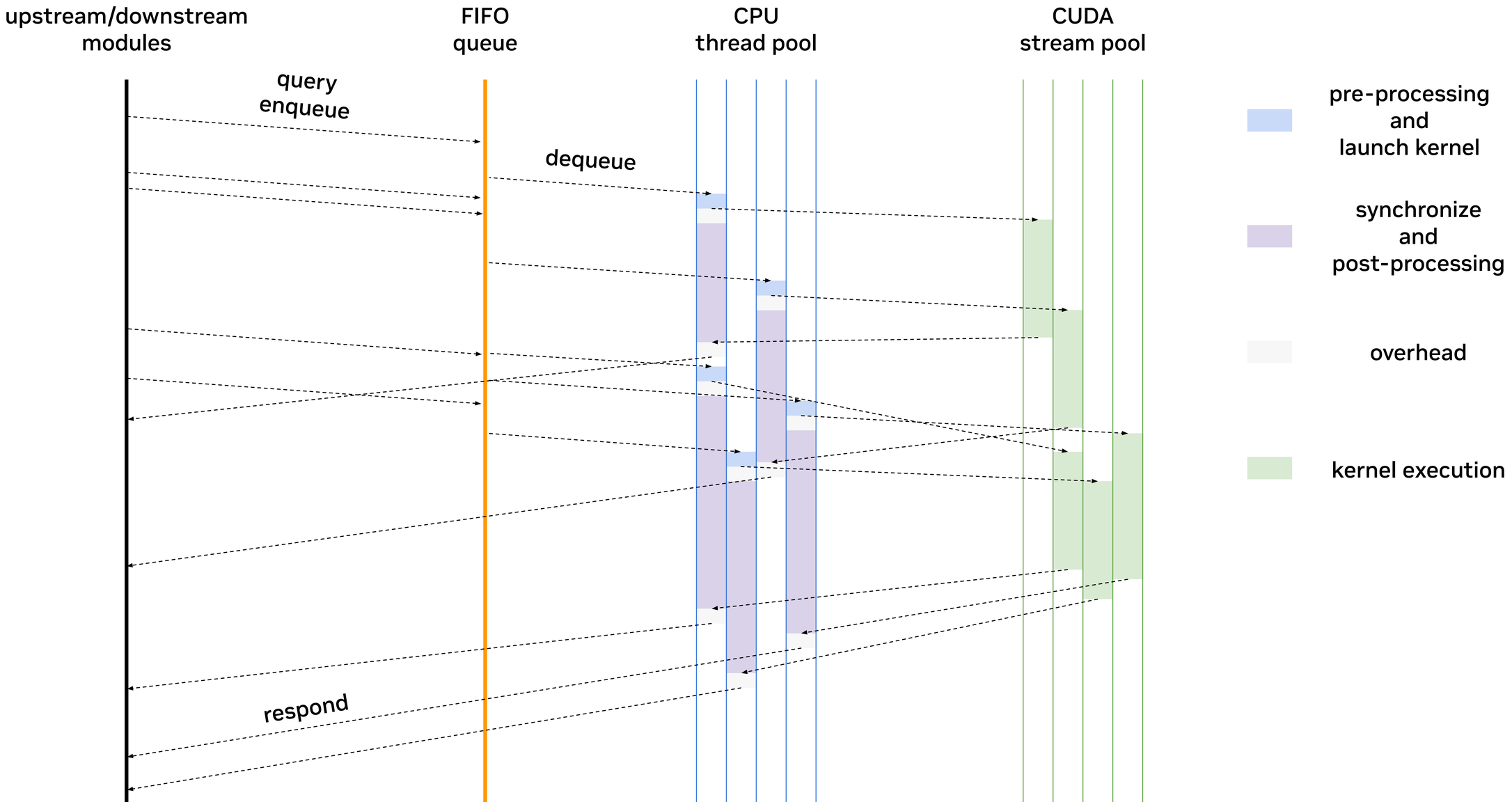
Limited CPU Threads / Limited CUDA Streams

Implementation

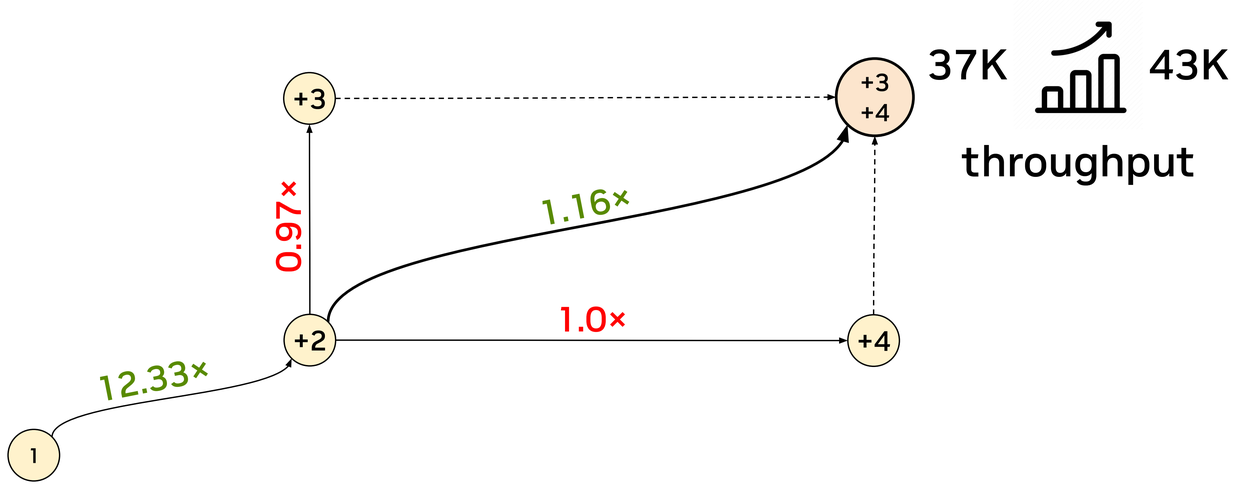


Limited CPU Threads / Limited CUDA Streams

Workflow



① Single CPU Thread & Single CUDA Stream
② Stream Ordered Memory Allocator
③ **Limited CPU Threads**
④ **Limited CUDA Streams**

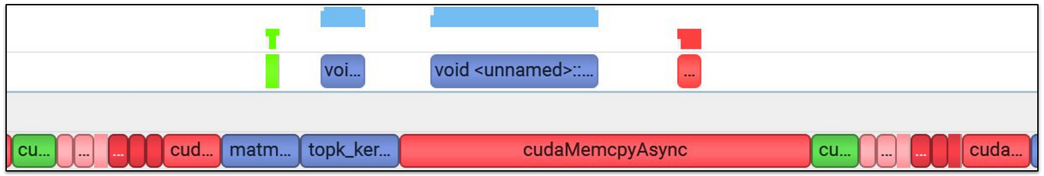


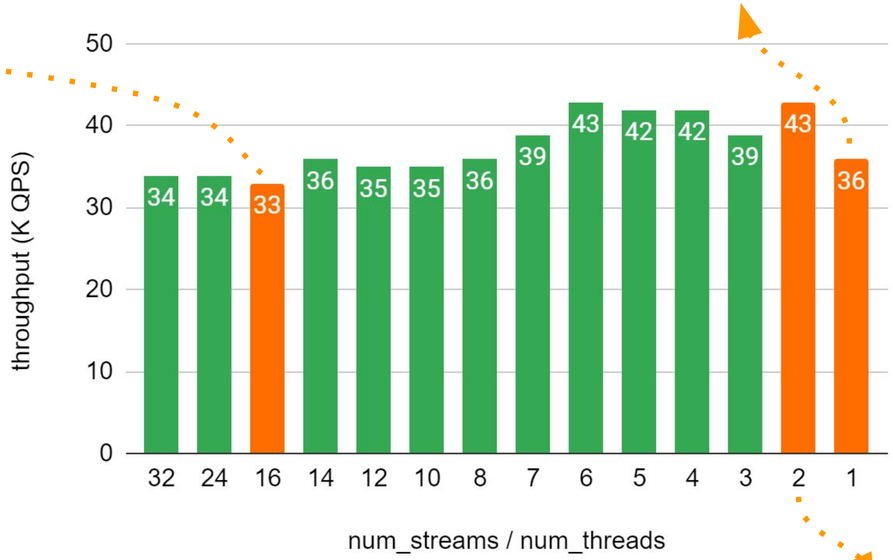
37K → 43K

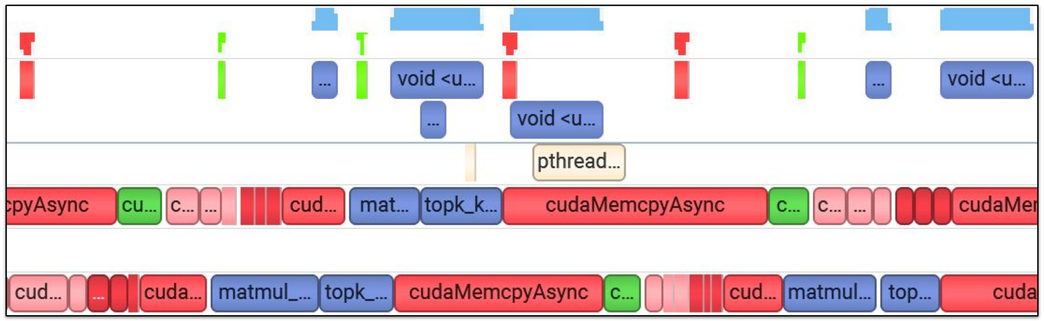
throughput

Limited CPU Threads / Limited CUDA Streams

Comparison

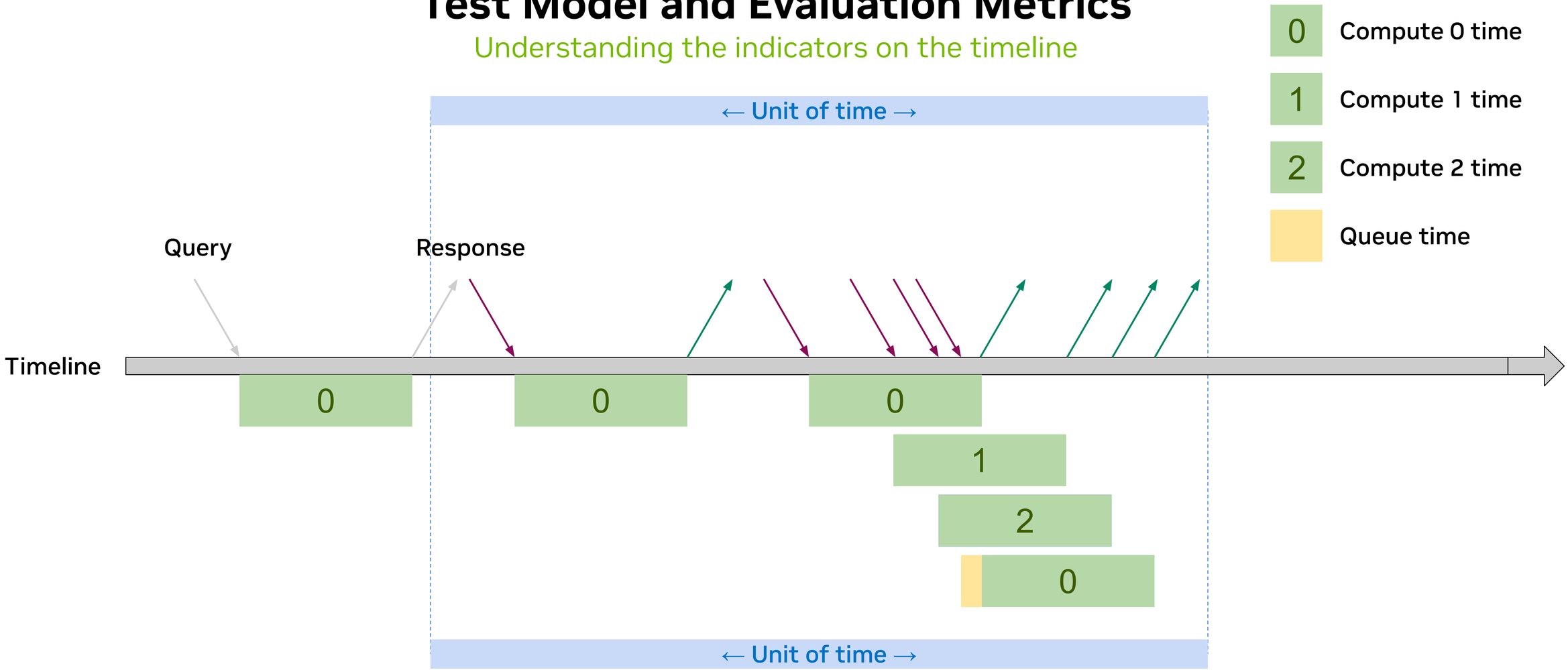






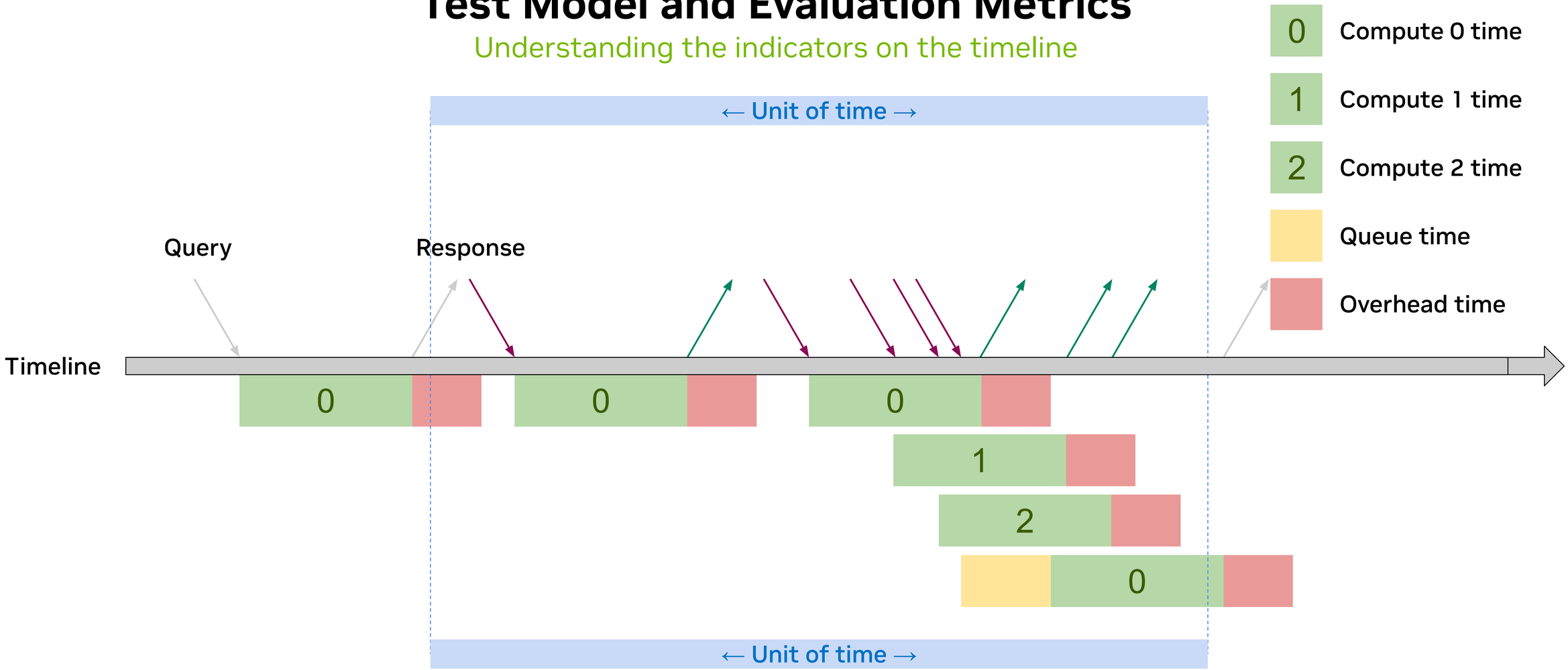
Test Model and Evaluation Metrics

Understanding the indicators on the timeline



Test Model and Evaluation Metrics

Understanding the indicators on the timeline

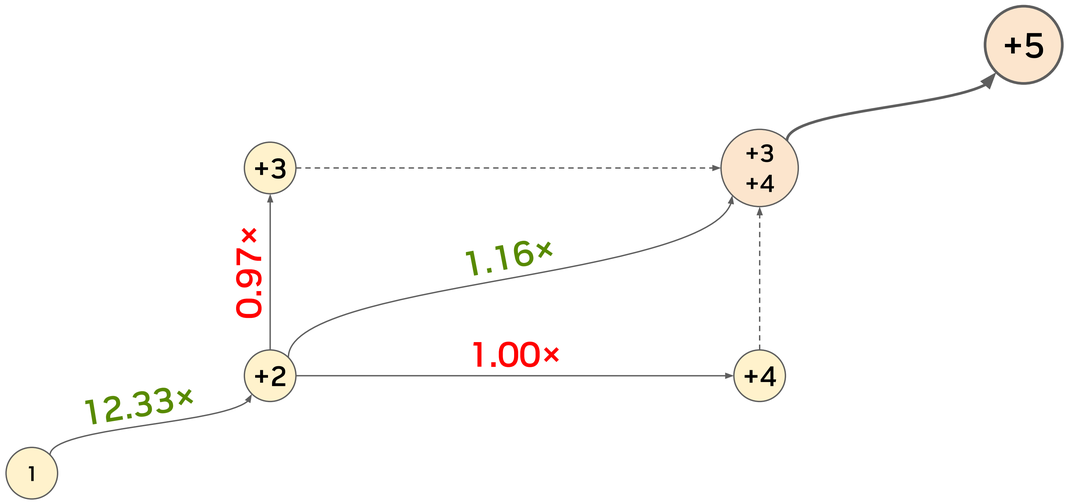


Test Model and Evaluation Metrics

Where might these overheads come from?

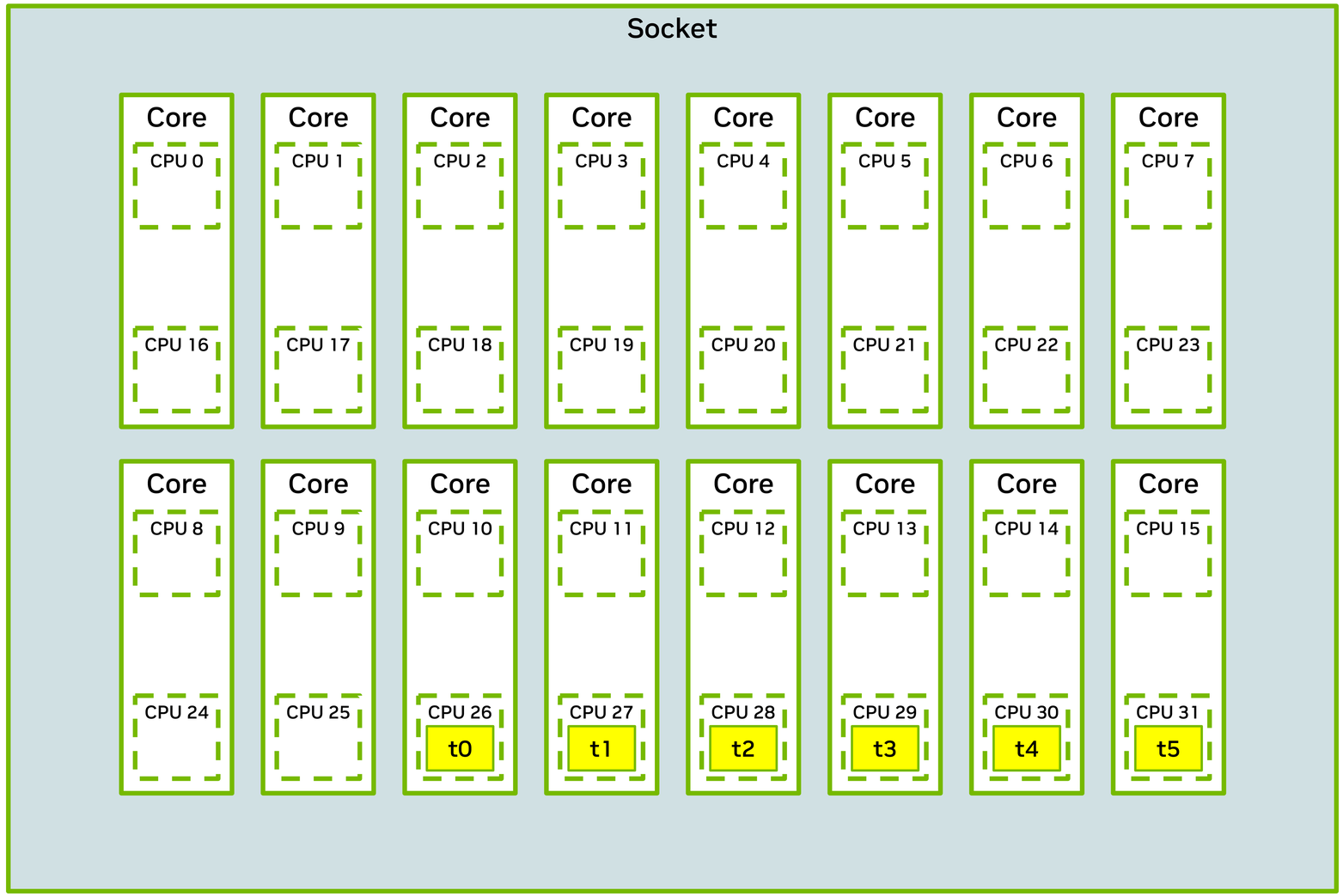
- Memory copy between host and device is queued at the DMA, which causes the copy action in one direction to be serialized on GPU
- Using pageable host memory results in very low PCIe bandwidth utilization, and it also serializes the copy API on CPU
- The use of multiple CPU threads causes the internal serialization of CUDA APIs operating on the same CUDA context
- Using multiple CPU cores can increase the latency and decrease the bandwidth for inter-core communication
- Frequent switching between user mode and kernel mode can be costly in terms of time and resources, and can affect system performance

① Single CPU Thread & Single CUDA Stream
② Stream Ordered Memory Allocator
③ Limited CPU Threads
④ Limited CUDA Streams
⑤ **Thread-Core Binding and Core Reordering**



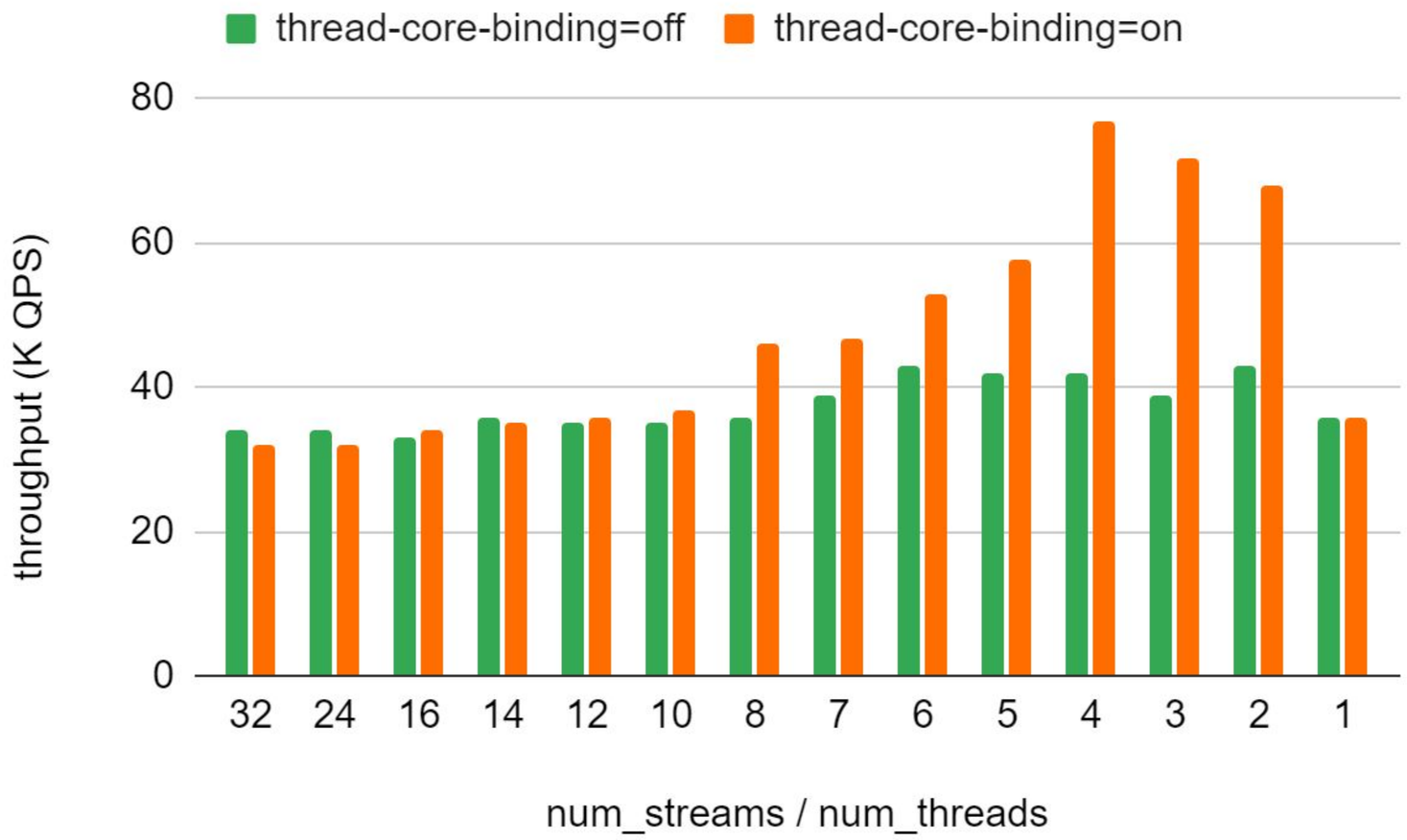
Thread-Core Binding and Core Reordering

Interpretation



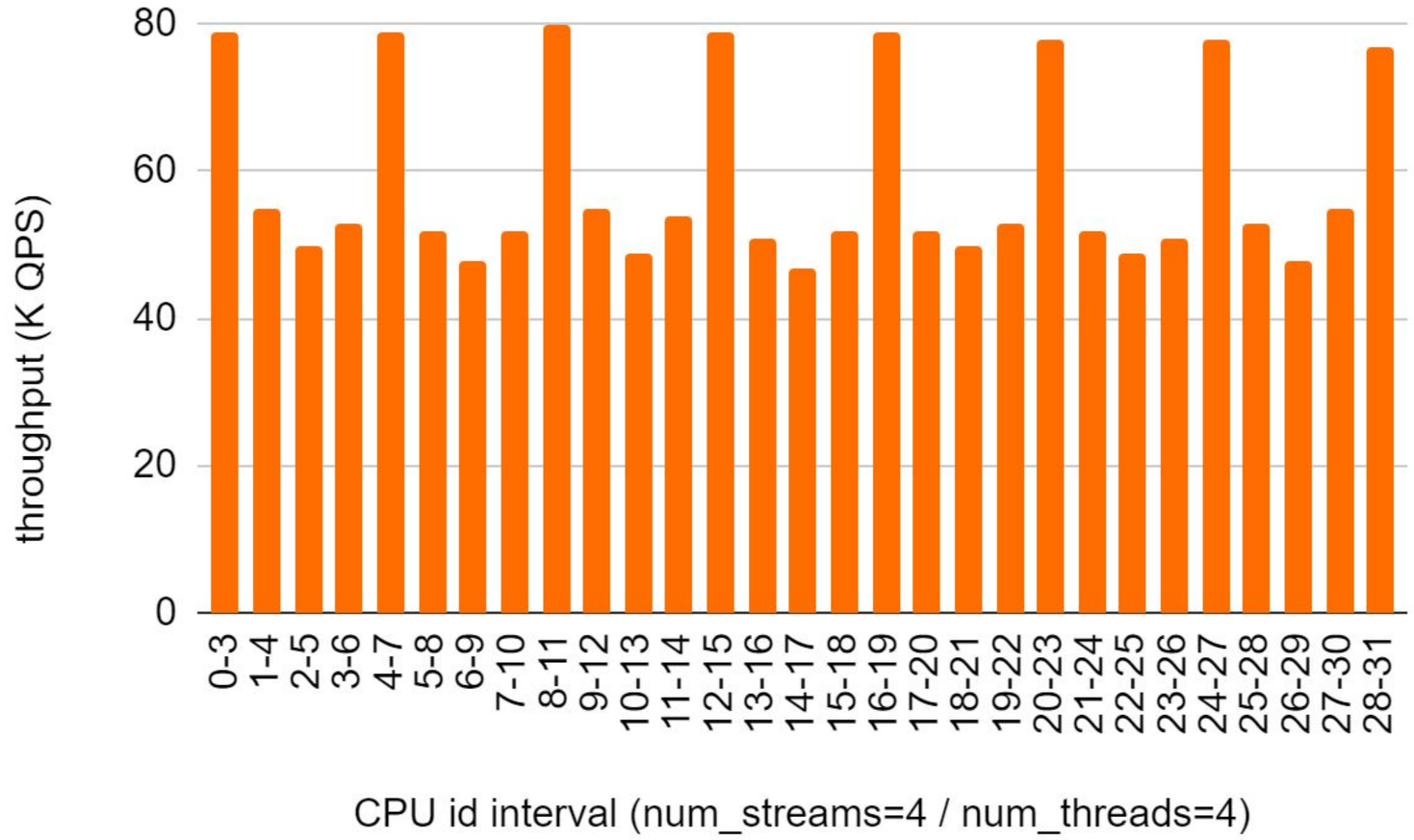
Thread-Core Binding and Core Reordering

Interpretation



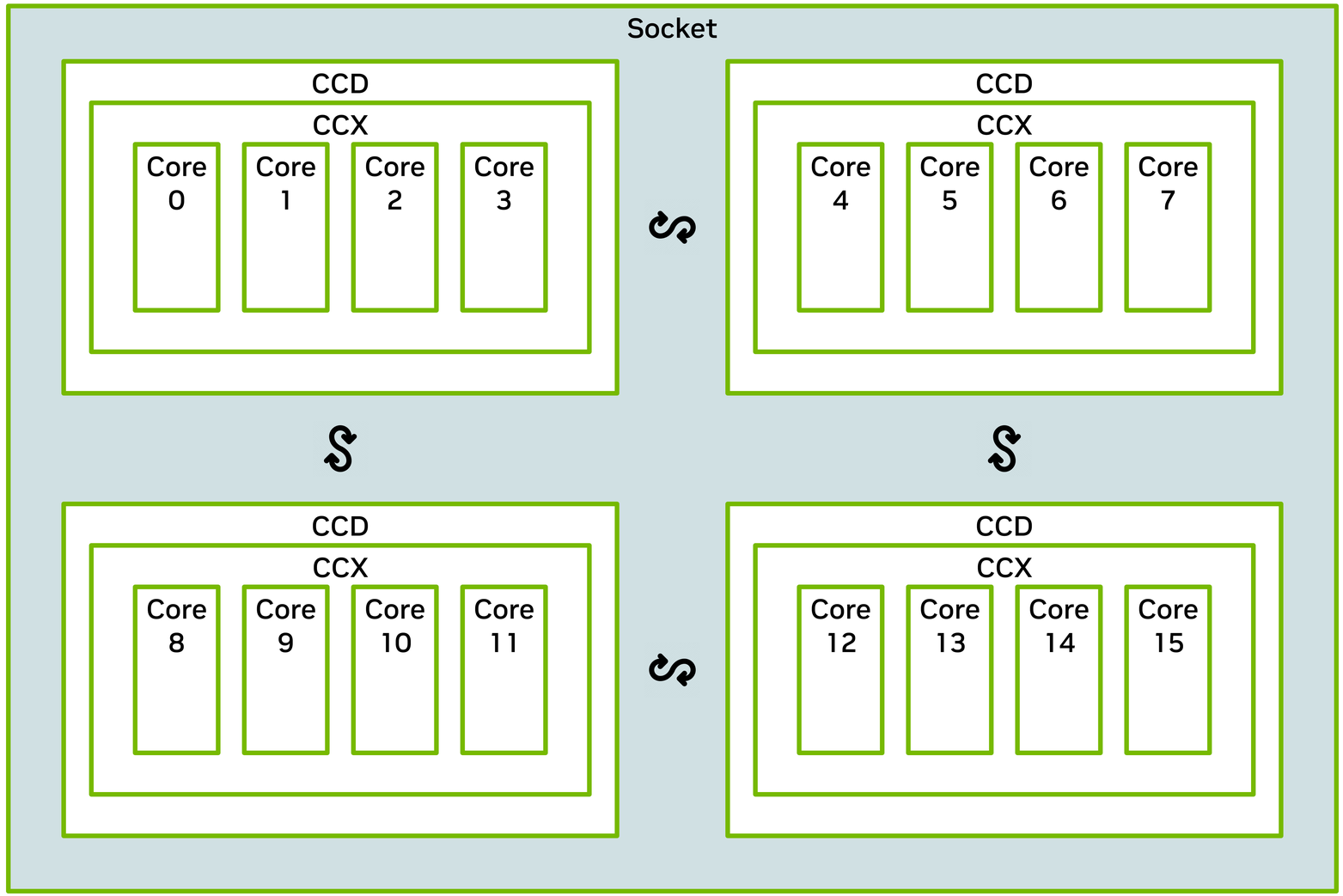
Thread-Core Binding and Core Reordering

Interpretation



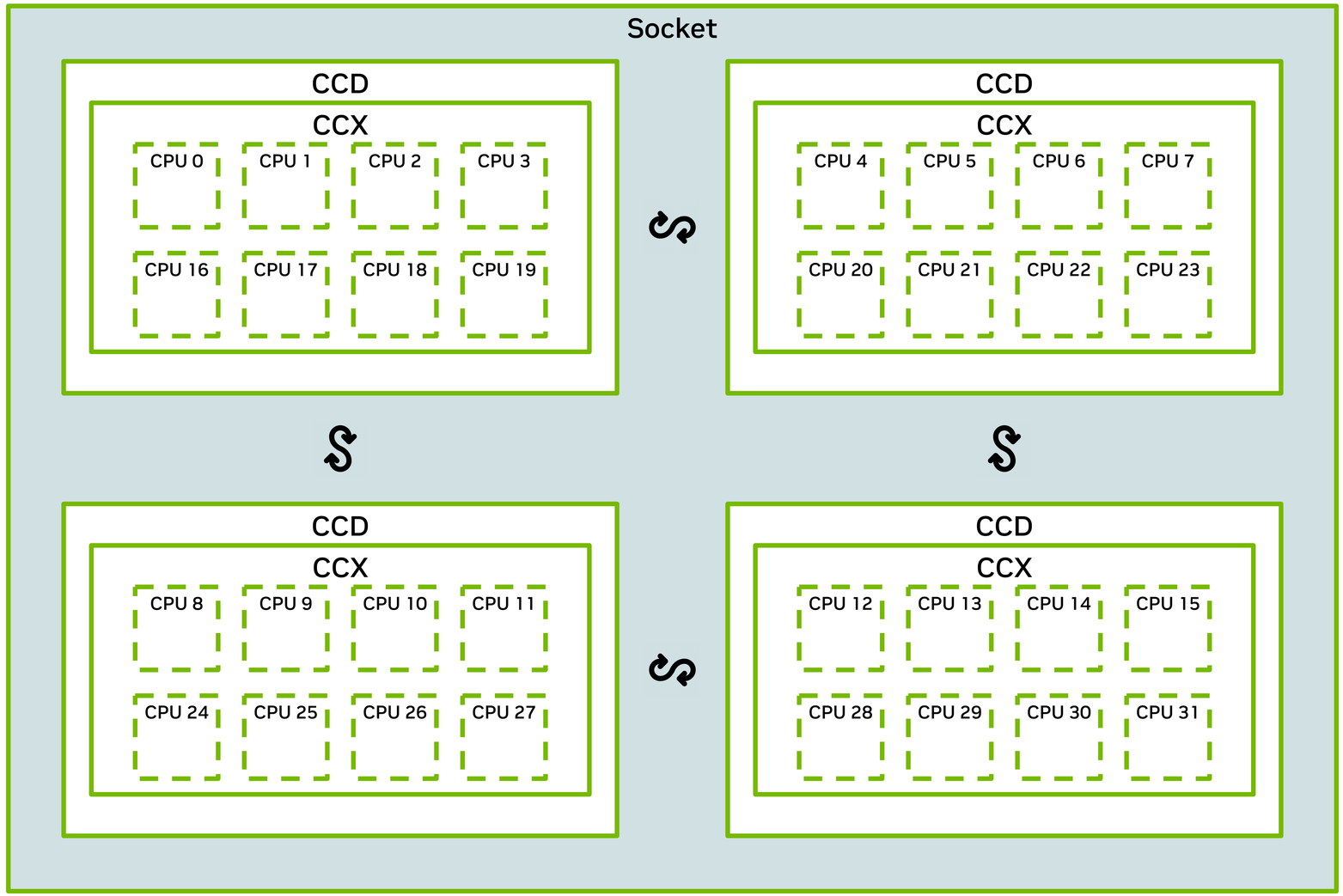
Thread-Core Binding and Core Reordering

Interpretation



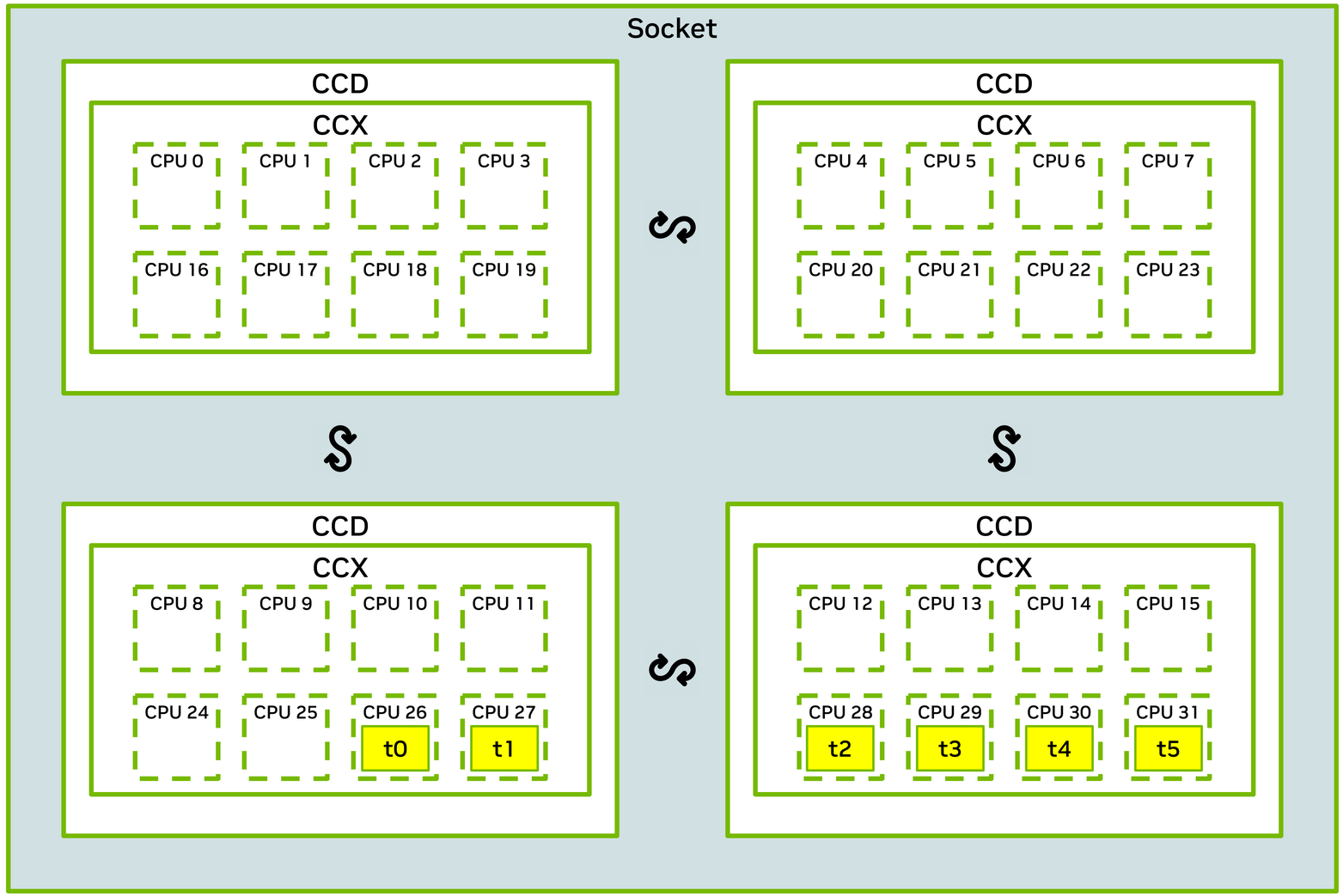
Thread-Core Binding and Core Reordering

Interpretation



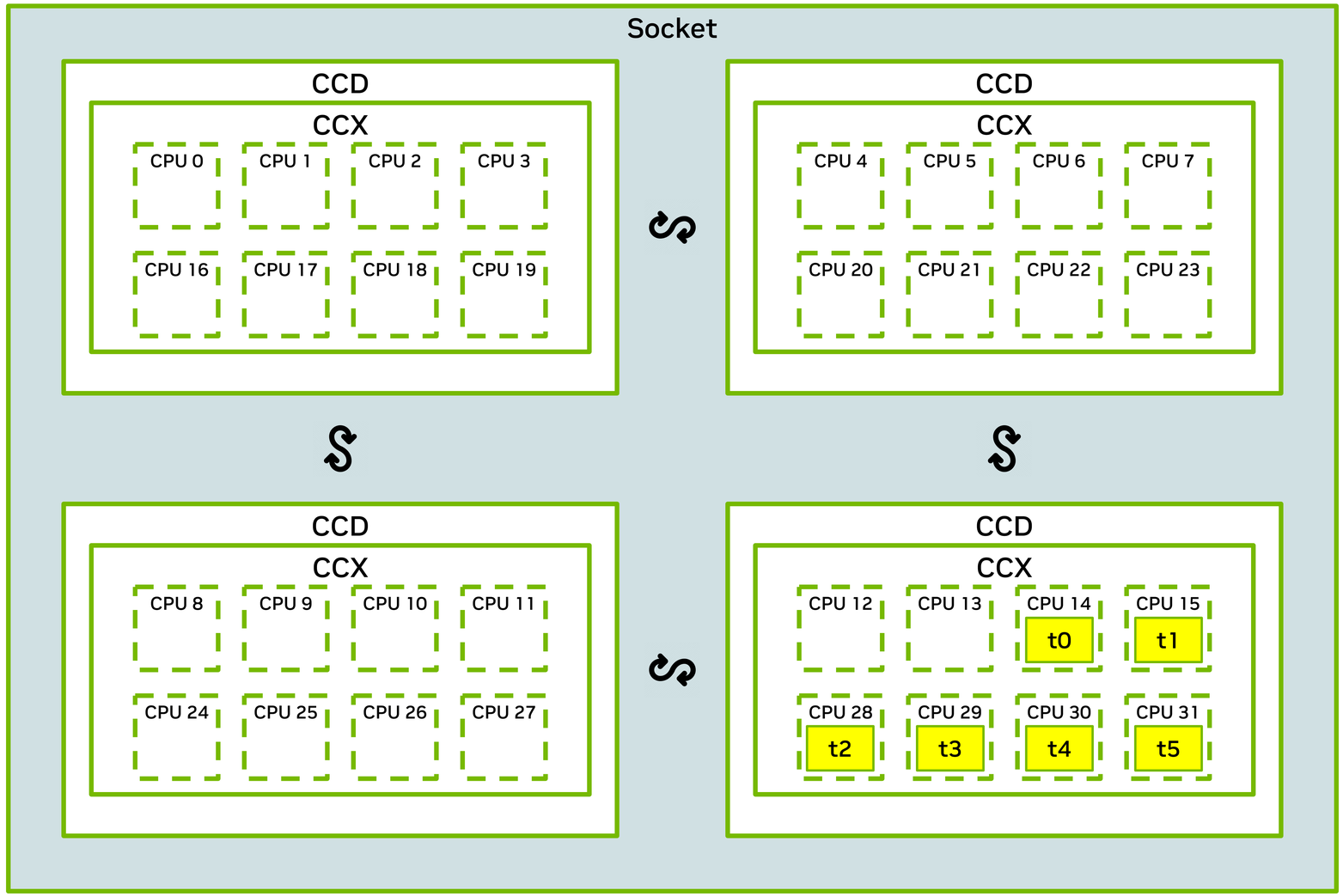
Thread-Core Binding and Core Reordering

Interpretation



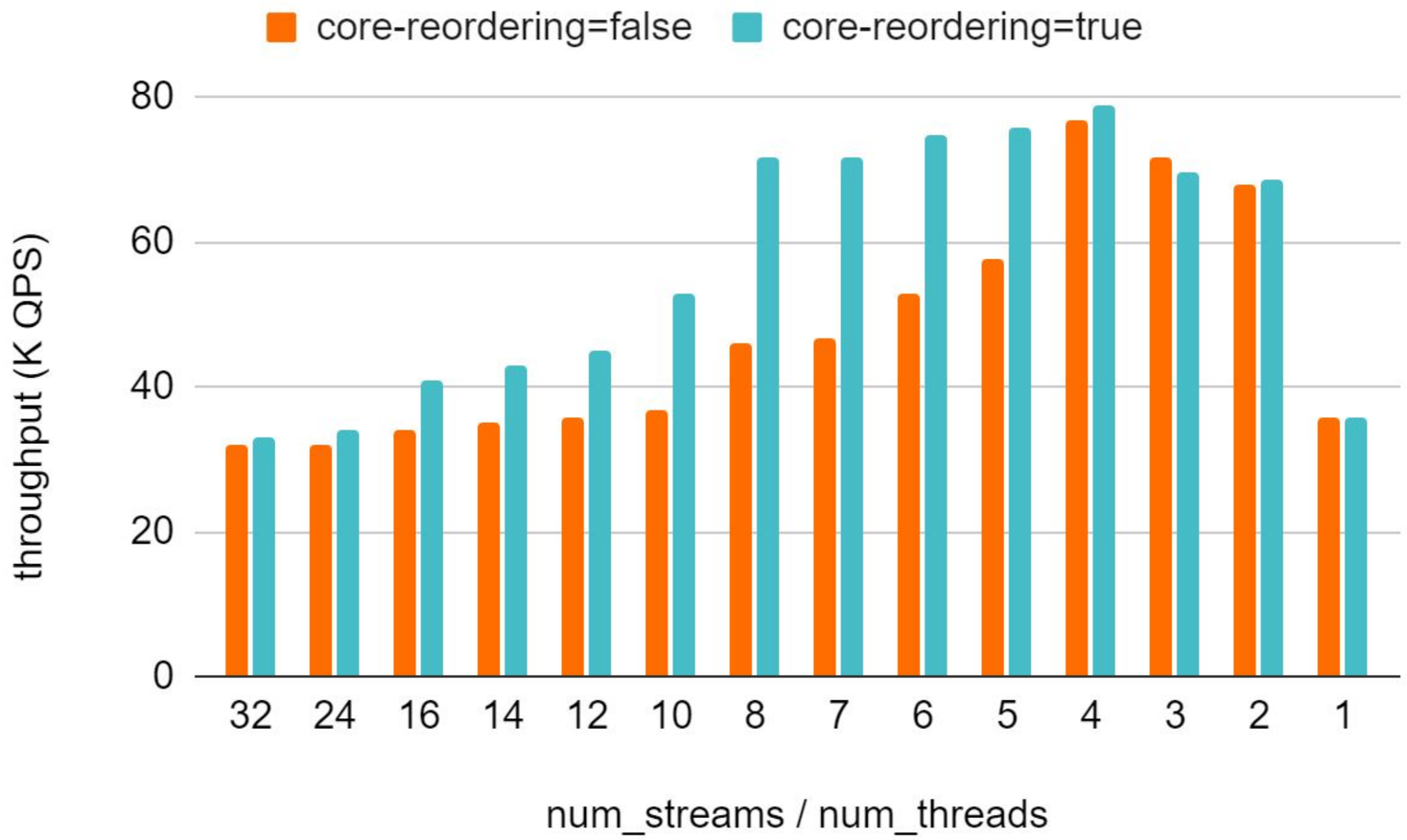
Thread-Core Binding and Core Reordering

Interpretation

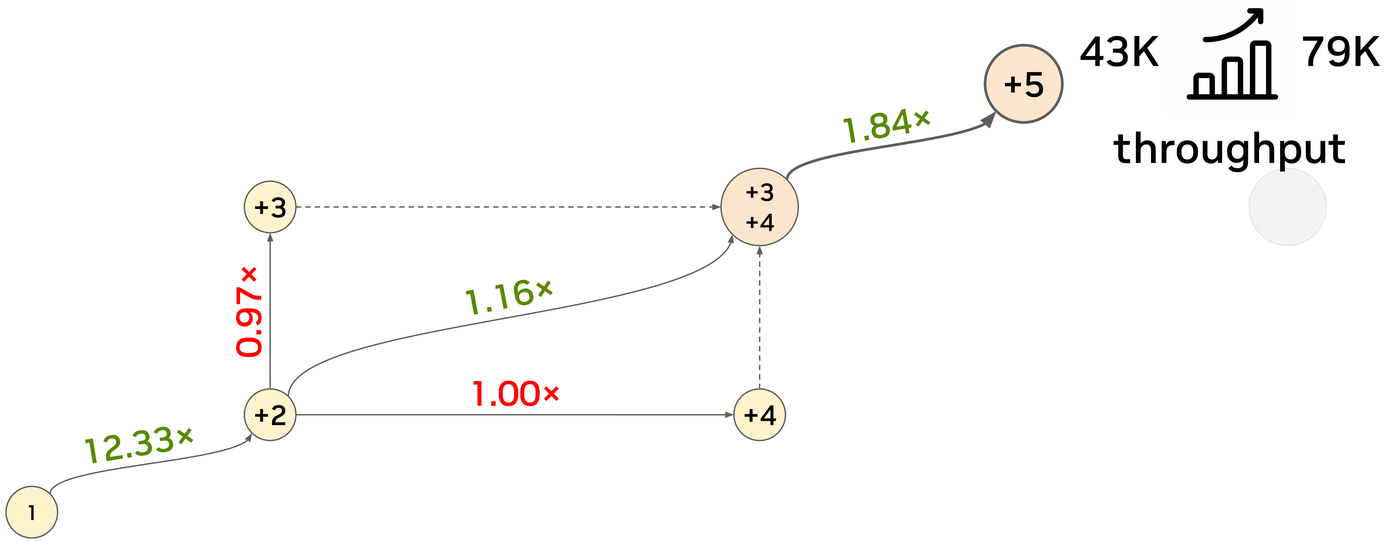


Thread-Core Binding and Core Reordering

Interpretation



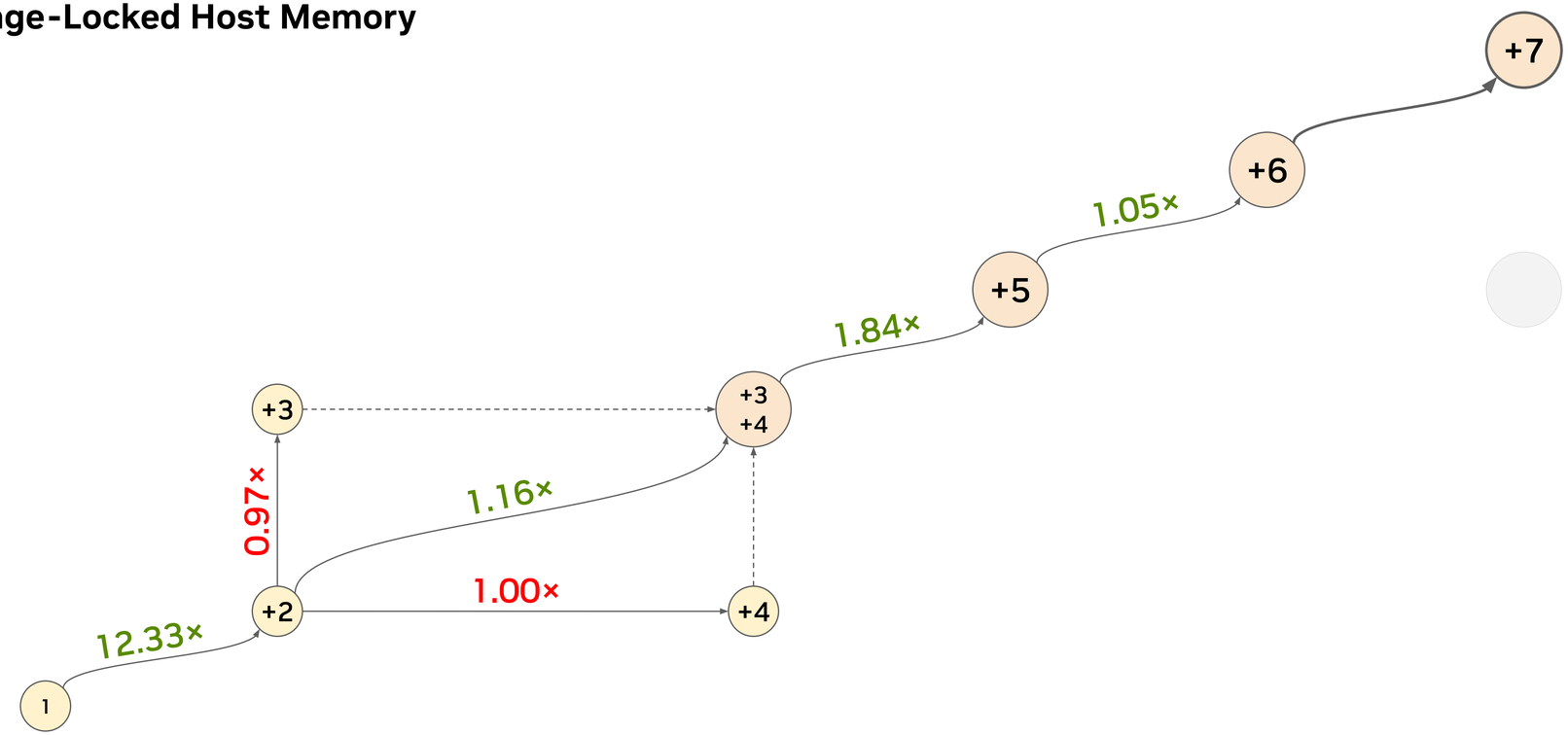
① Single CPU Thread & Single CUDA Stream
② Stream Ordered Memory Allocator
③ Limited CPU Threads
④ Limited CUDA Streams
⑤ **Thread-Core Binding and Core Reordering**



43K → 79K

throughput

① Single CPU Thread & Single CUDA Stream
② Stream Ordered Memory Allocator
③ Limited CPU Threads
④ Limited CUDA Streams
⑤ Thread-Core Binding and Core Reordering
⑥ Resource Pool
⑦ **Page-Locked Host Memory**



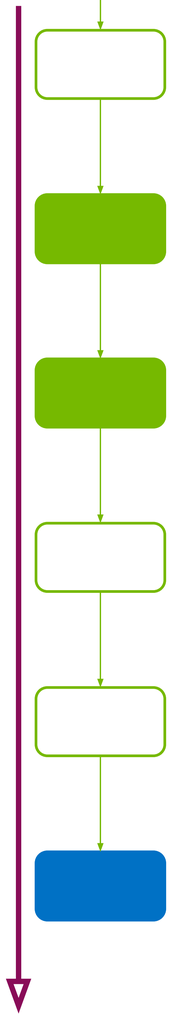
Page-Locked Host Memory

Implementation

- CUDA stream
- Device Buffer ***dA***, ***dB*** and ***dC***
- Page-Locked Host Buffer ***hA***, ***hB*** and ***hC***

Resource pool

When a query arrives



When a query arrives

memcpy

cudaMemcpyAsync

matmul<<<>>>(..., dA, dB)

topk<<<>>>(dB, dC)

cudaMemcpyAsync

cudaStreamSynchronize

memcpy

notify downstream module

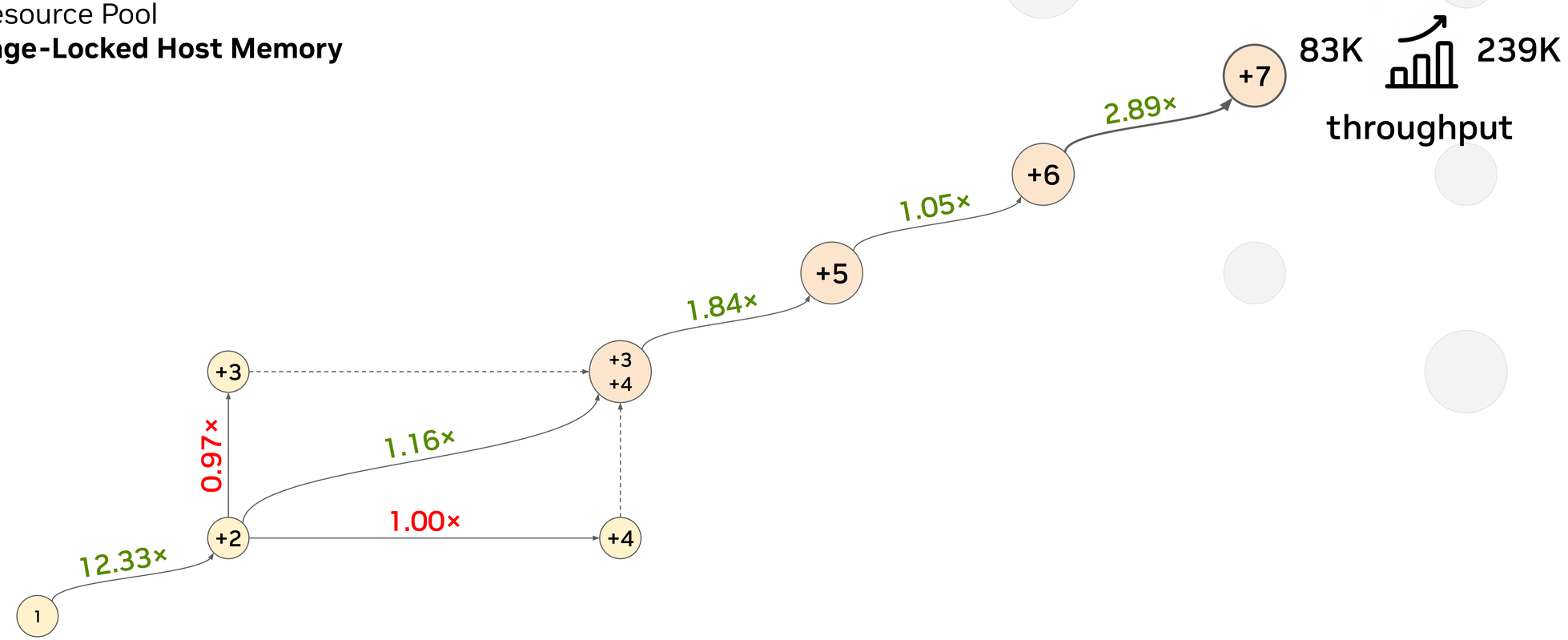
Copy the input data from query associated host buffer to ***hA***

Copy the input data from ***hA*** to ***dA***

Copy the output data from ***dC*** to ***hC***

Copy the output data from ***hC*** to response associated host buffer

① Single CPU Thread & Single CUDA Stream
② Stream Ordered Memory Allocator
③ Limited CPU Threads
④ Limited CUDA Streams
⑤ Thread-Core Binding and Core Reordering
⑥ Resource Pool
⑦ **Page-Locked Host Memory**



Page-Locked Host Memory

Interpretation

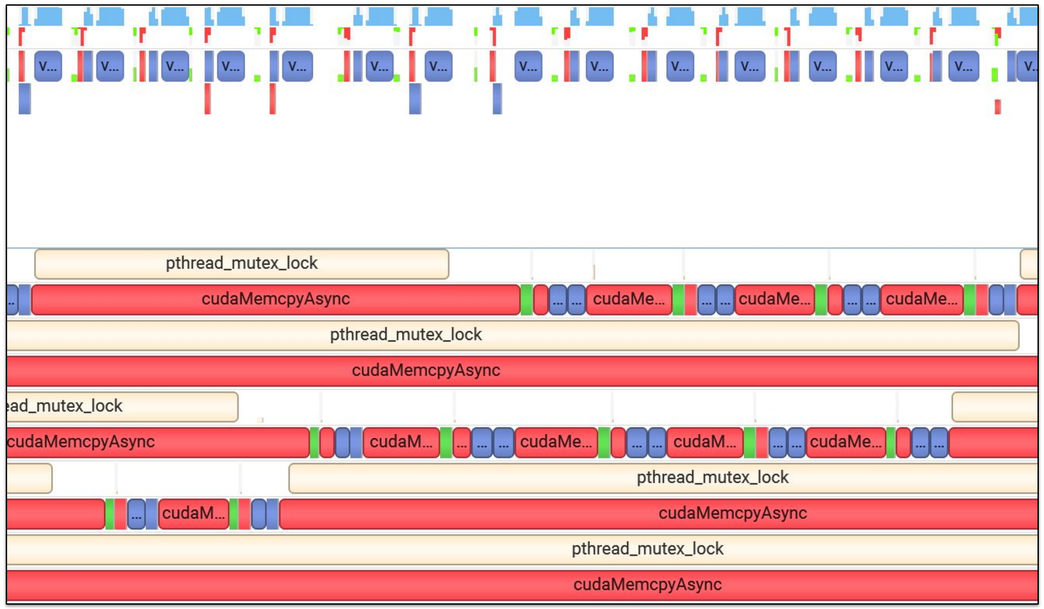
Using pre-allocated page-locked host memory as a transfer buffer can have the following benefits:

- It can make the ***cudaMemcpyAsync*** API fully asynchronous

- It can increase the bandwidth between host memory and device memory on a system with a front-side bus, if host memory is allocated as page-locked

Page-Locked Host Memory

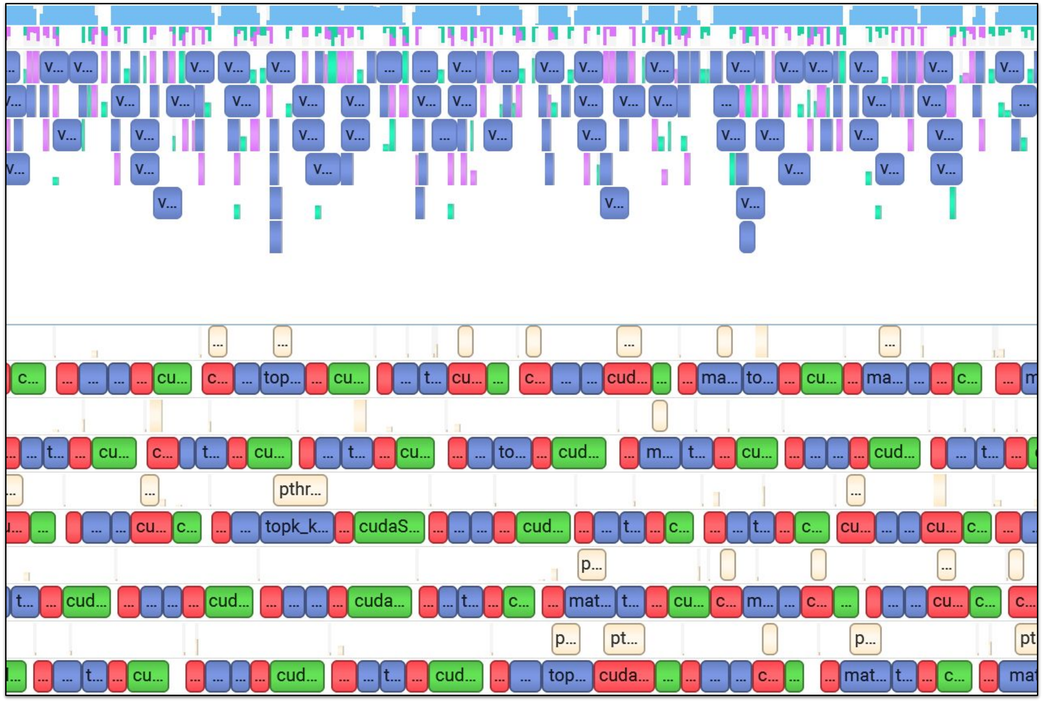
Comparison



Before Page-Locked Host Memory

num_threads=8

VS

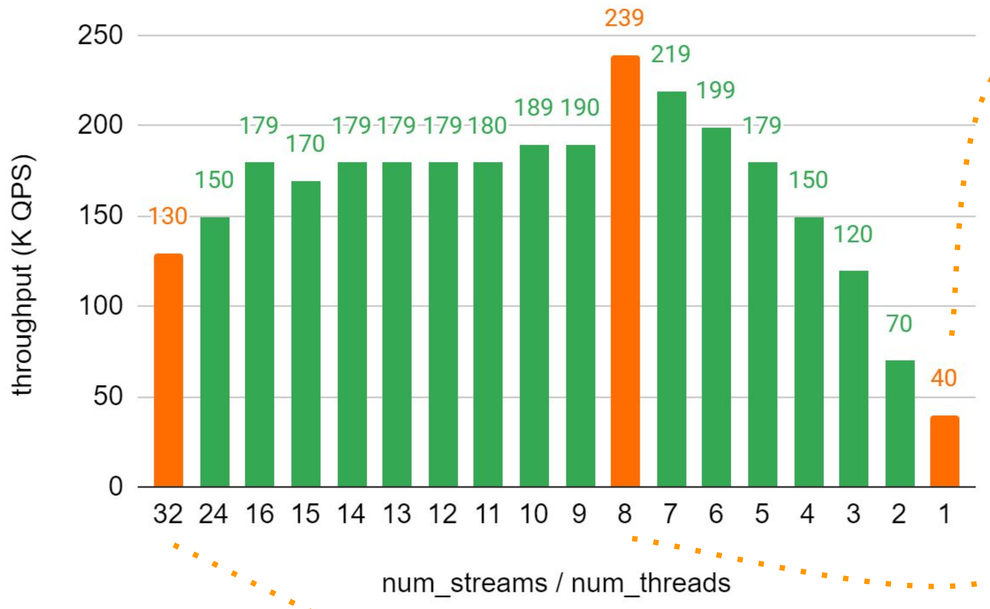


After Page-Locked Host Memory

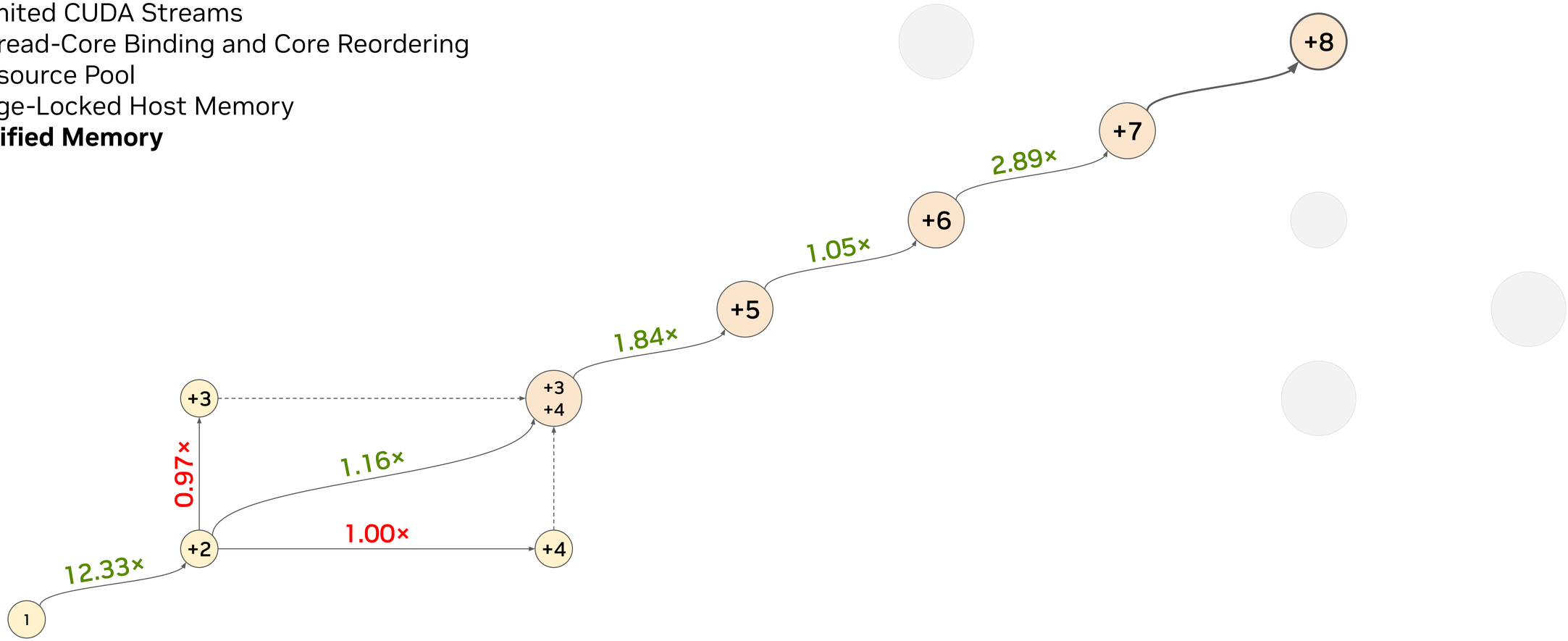
num_threads=8

Page-Locked Host Memory

Comparison

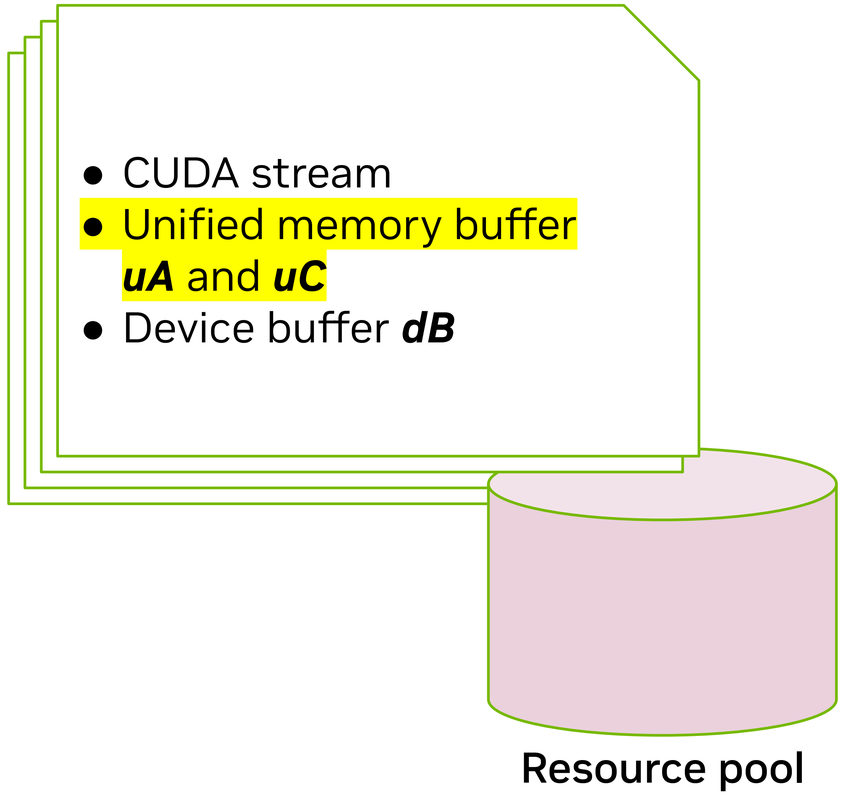


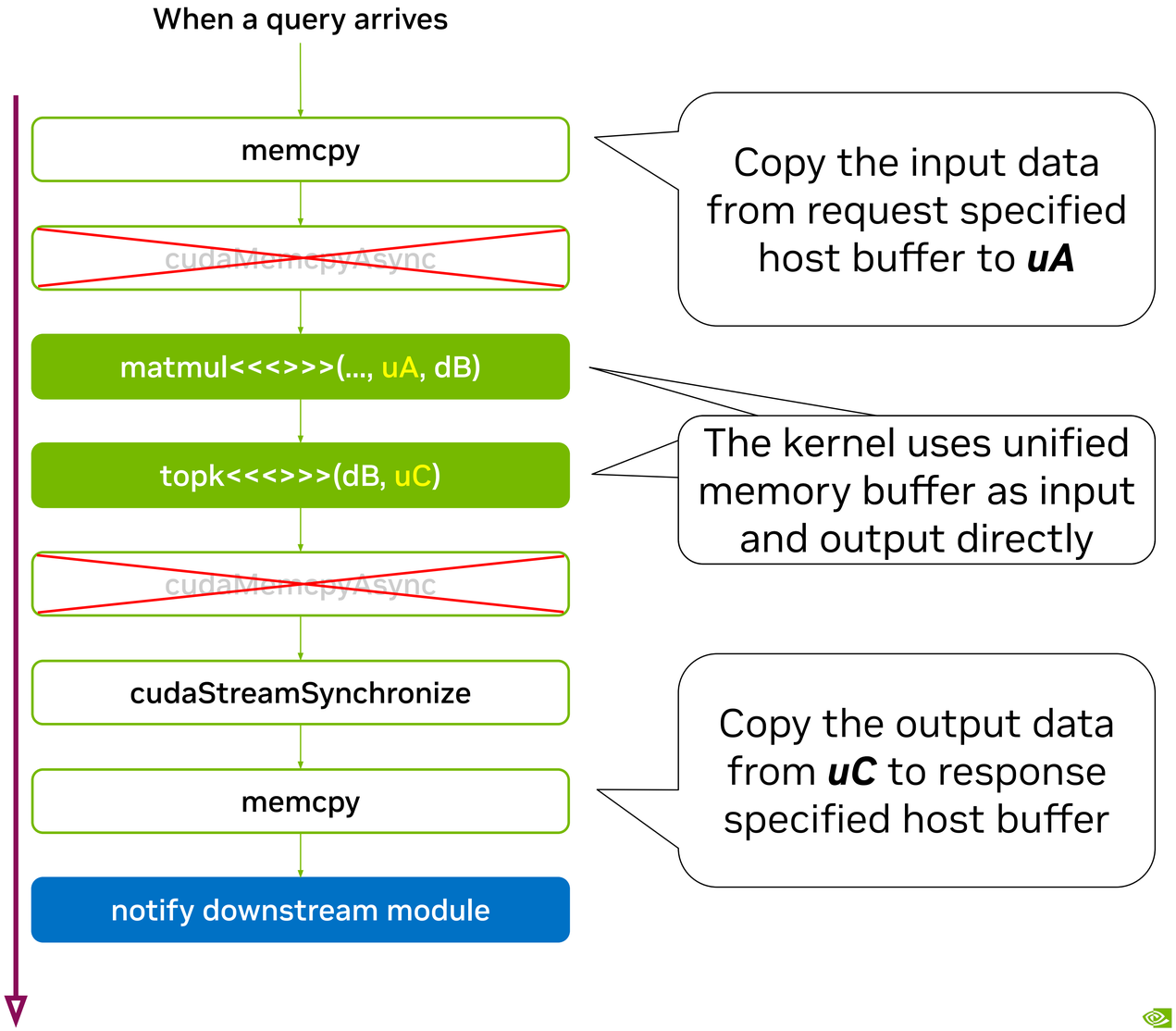
① Single CPU Thread & Single CUDA Stream
② Stream Ordered Memory Allocator
③ Limited CPU Threads
④ Limited CUDA Streams
⑤ Thread-Core Binding and Core Reordering
⑥ Resource Pool
⑦ Page-Locked Host Memory
⑧ **Unified Memory**



Unified Memory

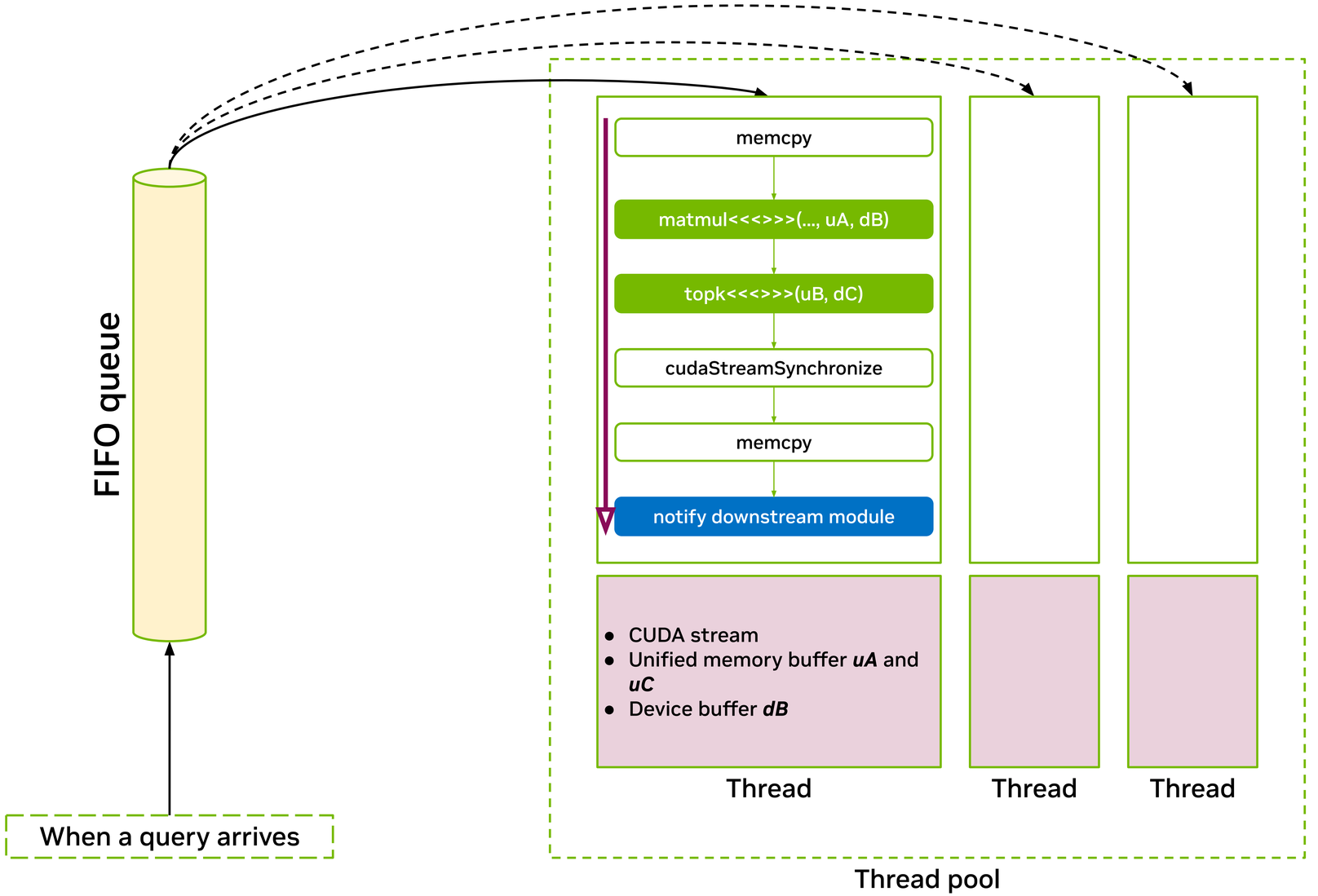
Implementation



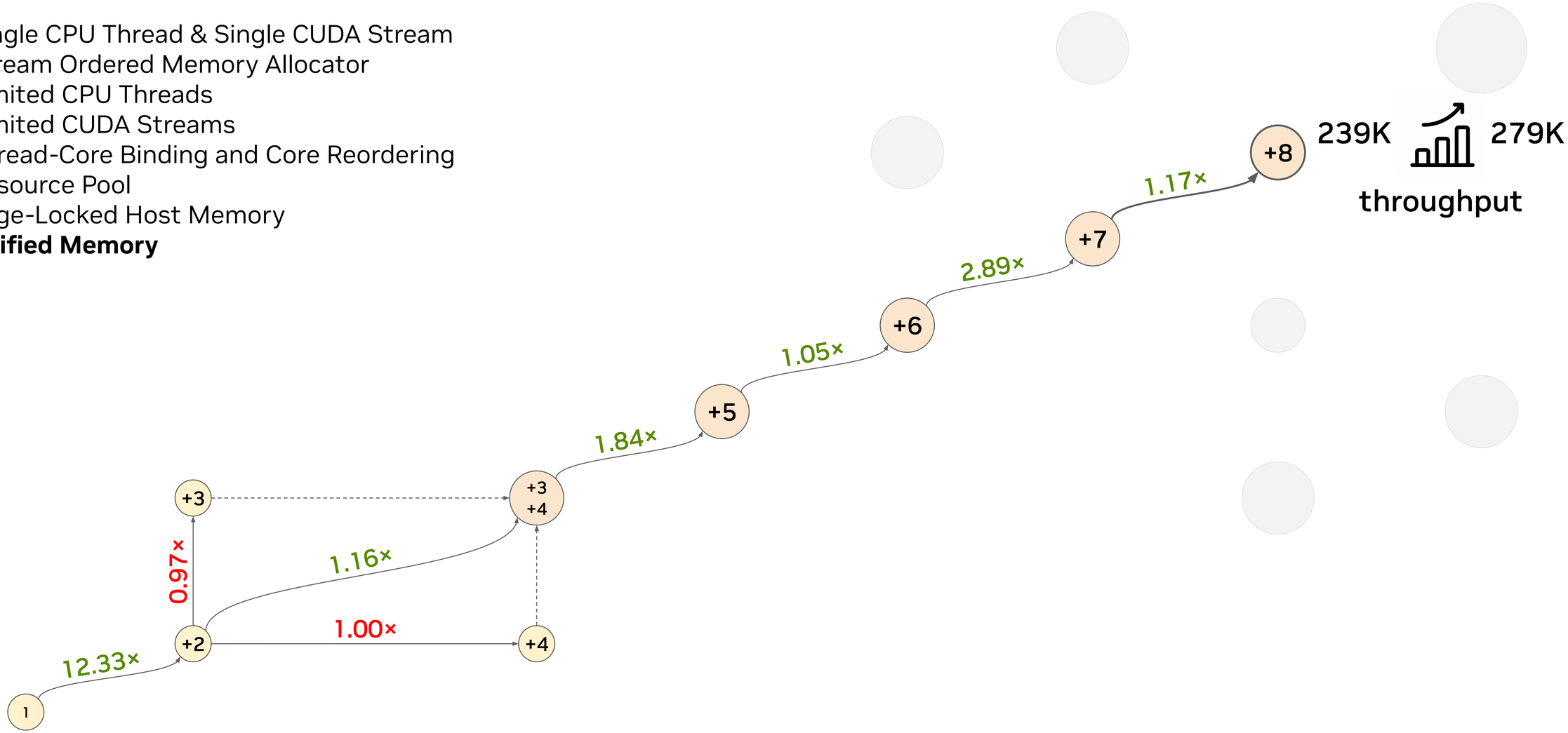


Unified Memory

Implementation

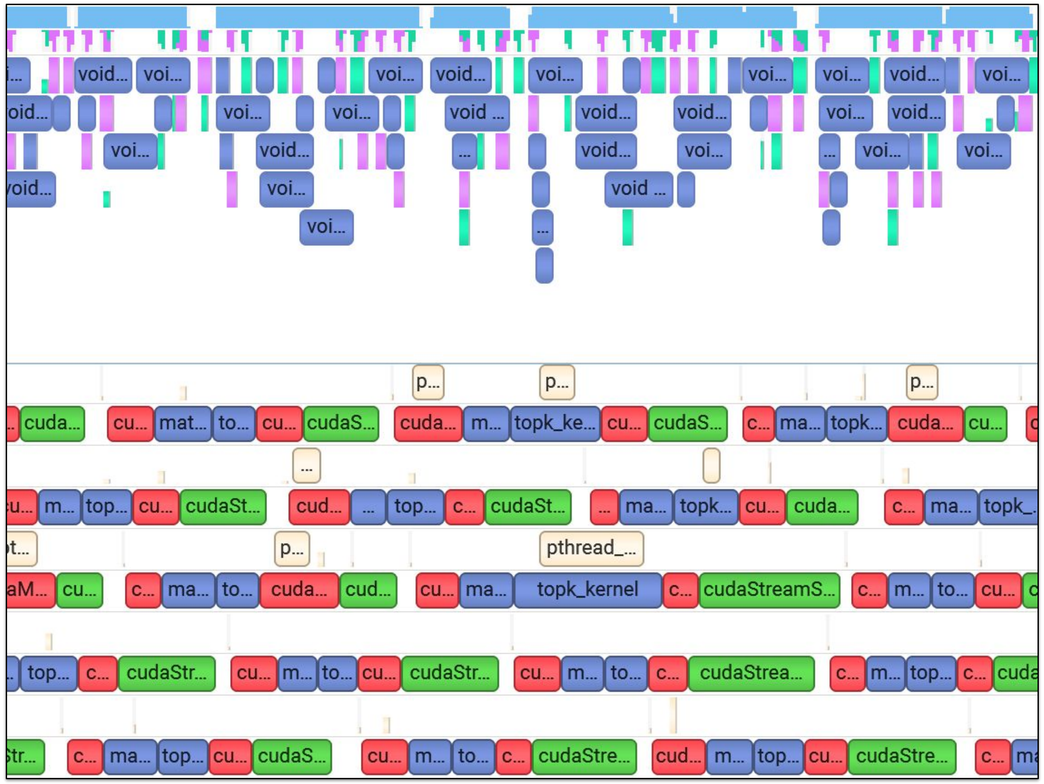


① Single CPU Thread & Single CUDA Stream
② Stream Ordered Memory Allocator
③ Limited CPU Threads
④ Limited CUDA Streams
⑤ Thread-Core Binding and Core Reordering
⑥ Resource Pool
⑦ Page-Locked Host Memory
⑧ **Unified Memory**



Unified Memory

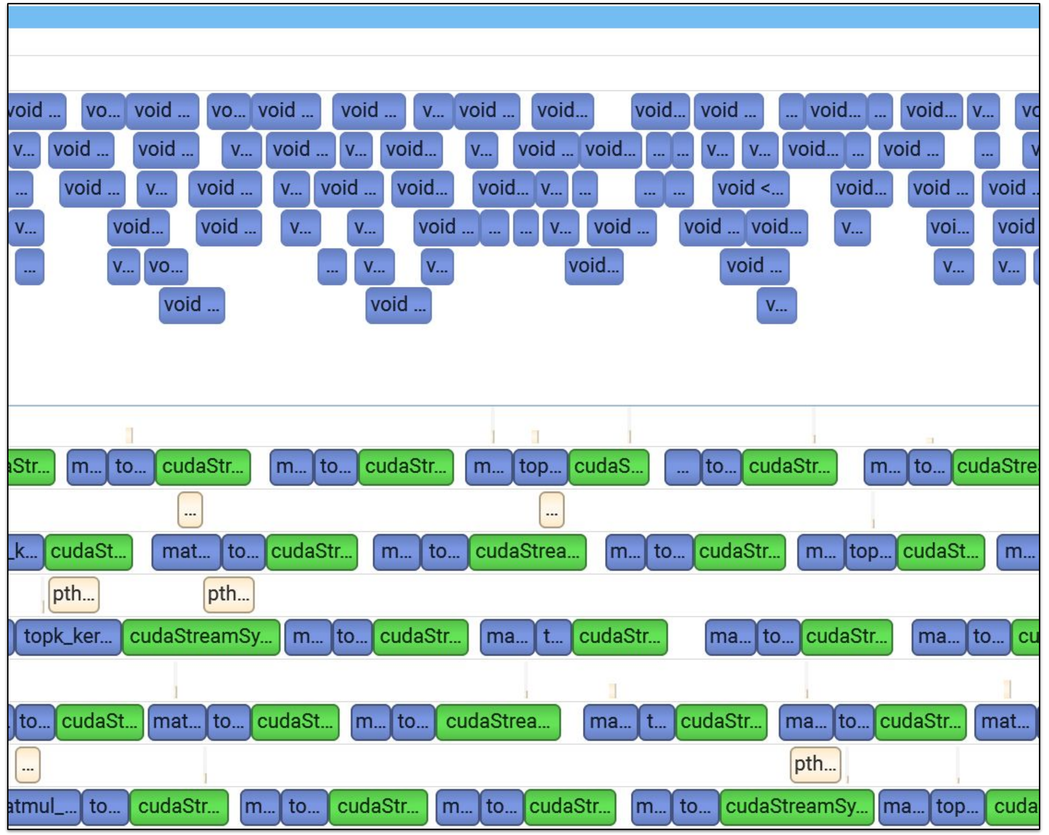
Comparison



Before Unified Memory

num_threads=8

VS

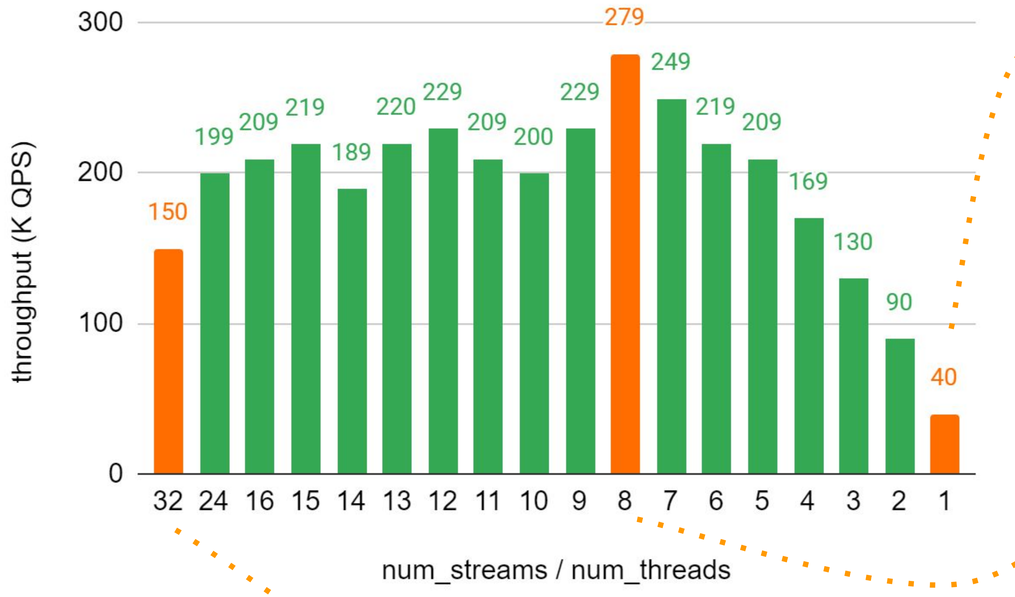


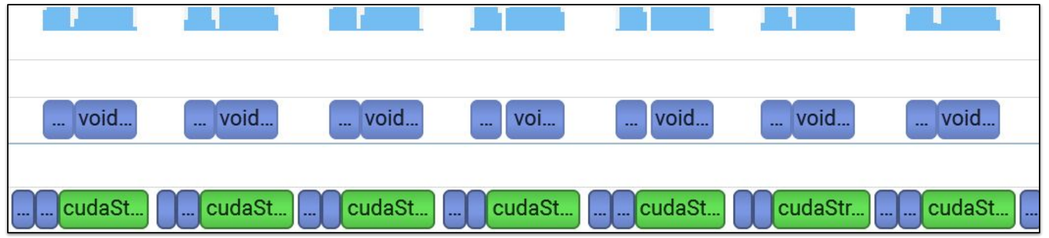
After Unified Memory

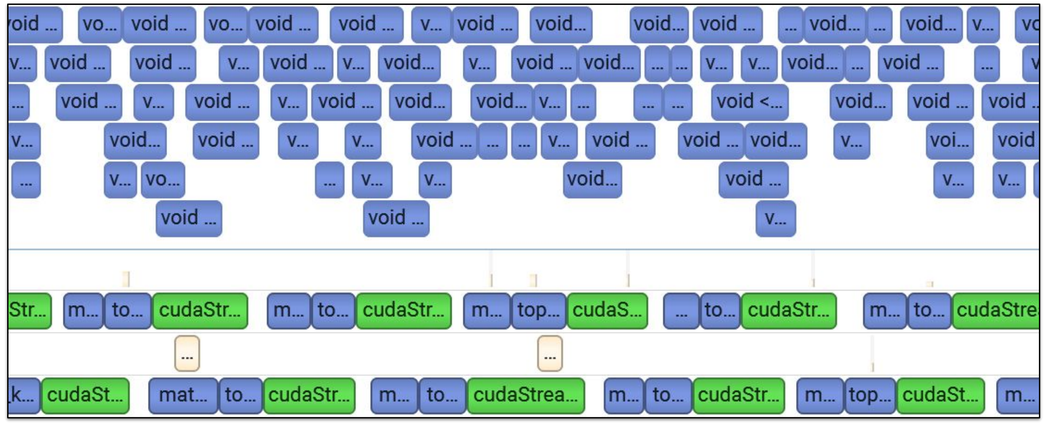
num_threads=8

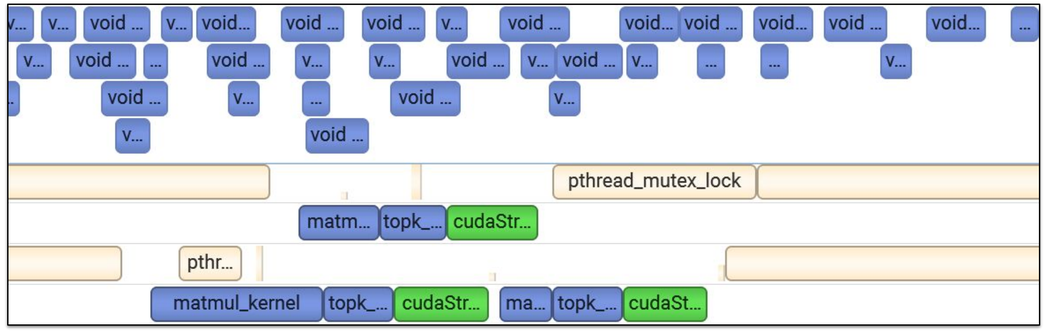
Unified Memory

Comparison



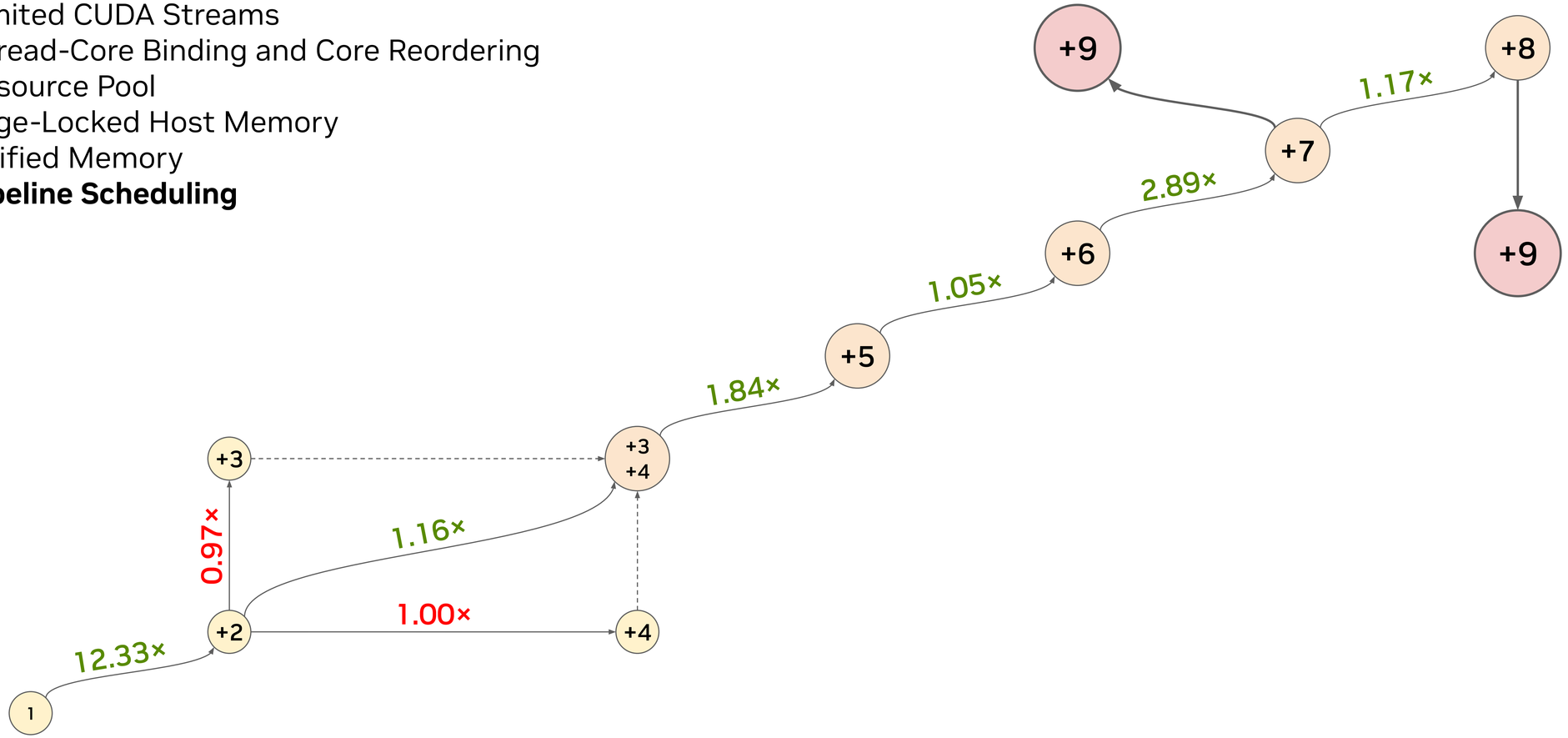






NVIDIA

① Single CPU Thread & Single CUDA Stream
② Stream Ordered Memory Allocator
③ Limited CPU Threads
④ Limited CUDA Streams
⑤ Thread-Core Binding and Core Reordering
⑥ Resource Pool
⑦ Page-Locked Host Memory
⑧ Unified Memory
⑨ **Pipeline Scheduling**



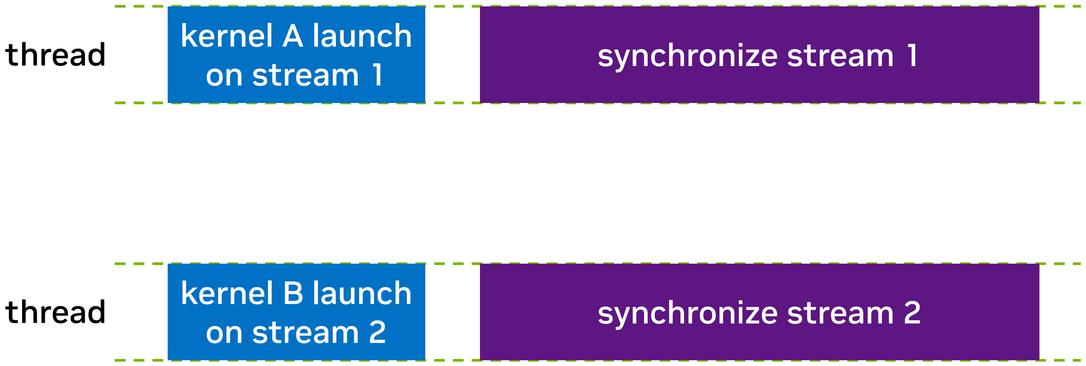
Pipeline Scheduling

Motivation

- For all the previous methods, the CPU thread and the CUDA stream have a 1:1 correspondence
- Using more CUDA streams with the previous methods requires more CPU threads
- However, increasing the number of CPU threads also increases the overhead
- The overhead can be caused by internal serialization of CUDA APIs operating on the same CUDA context
- A better way is to let one CPU thread manage multiple CUDA streams, which reduces the number of CPU threads and the overhead

Pipeline Scheduling

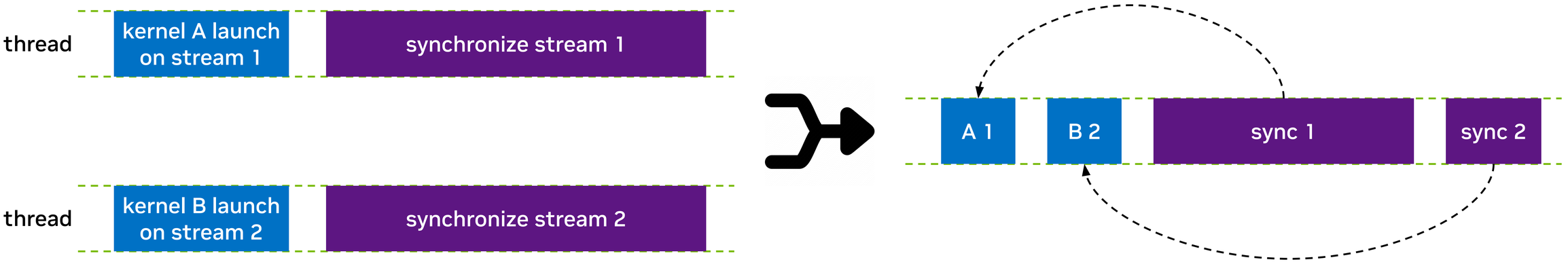
Interpretation



The API is a blocking, the CPU thread will be blocked until all tasks on the CUDA stream are completed.

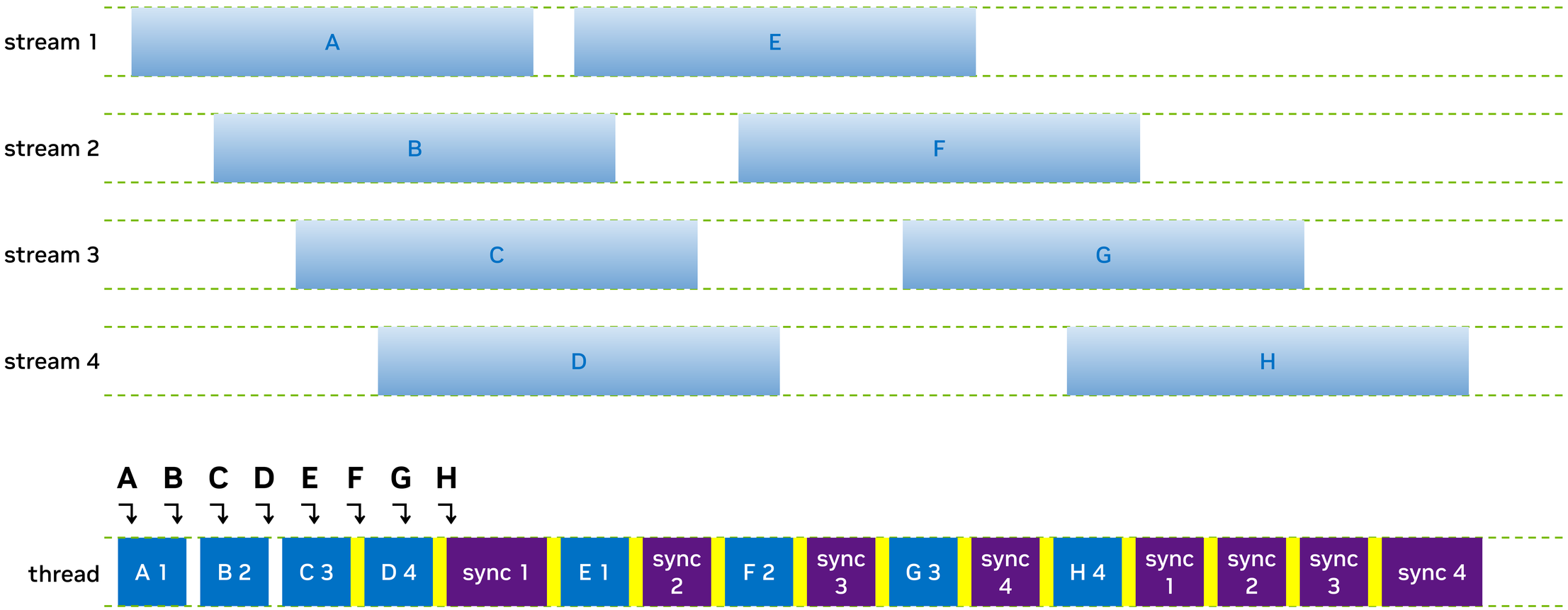
Pipeline Scheduling

Interpretation



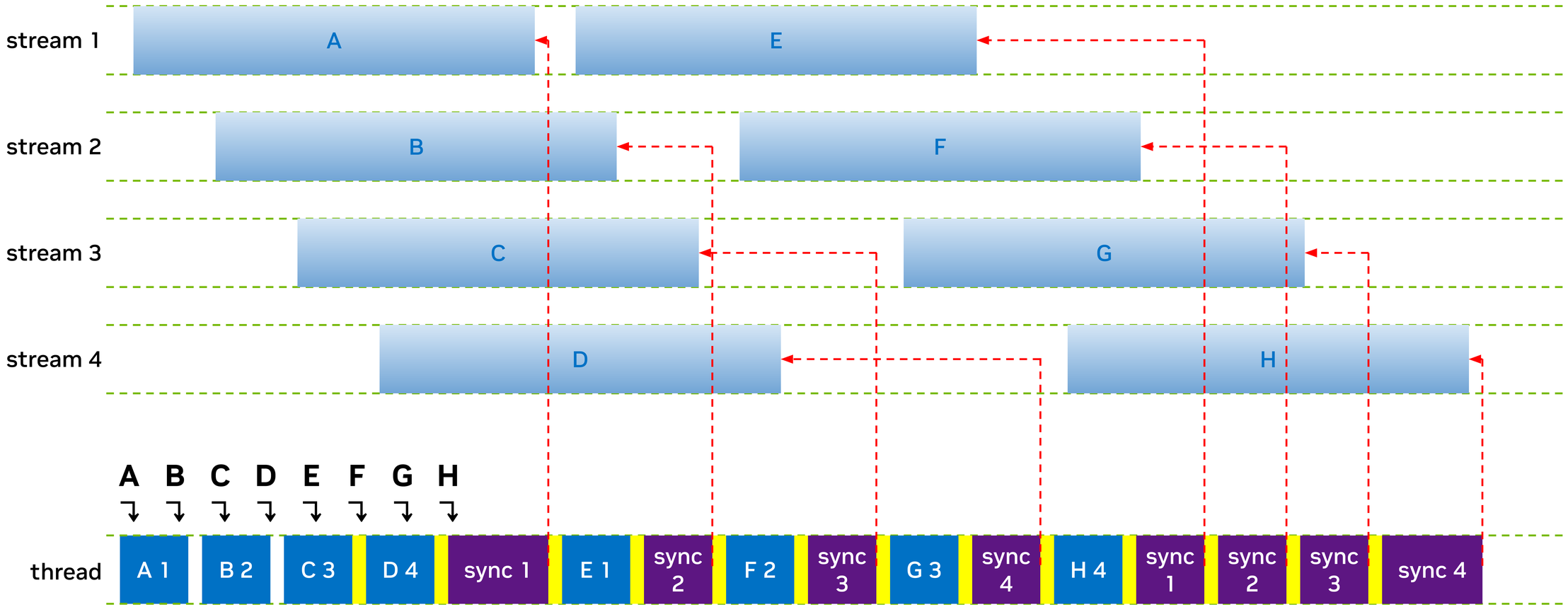
Pipeline Scheduling

Interpretation



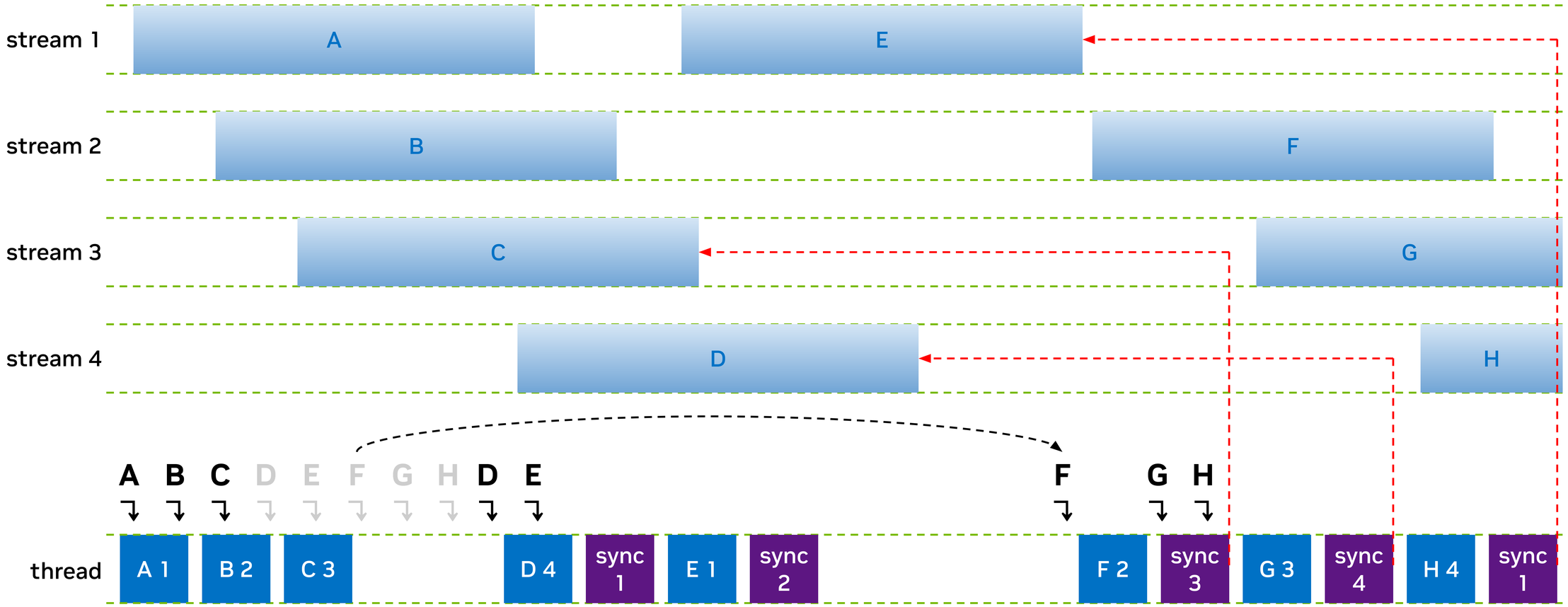
Pipeline Scheduling

Interpretation



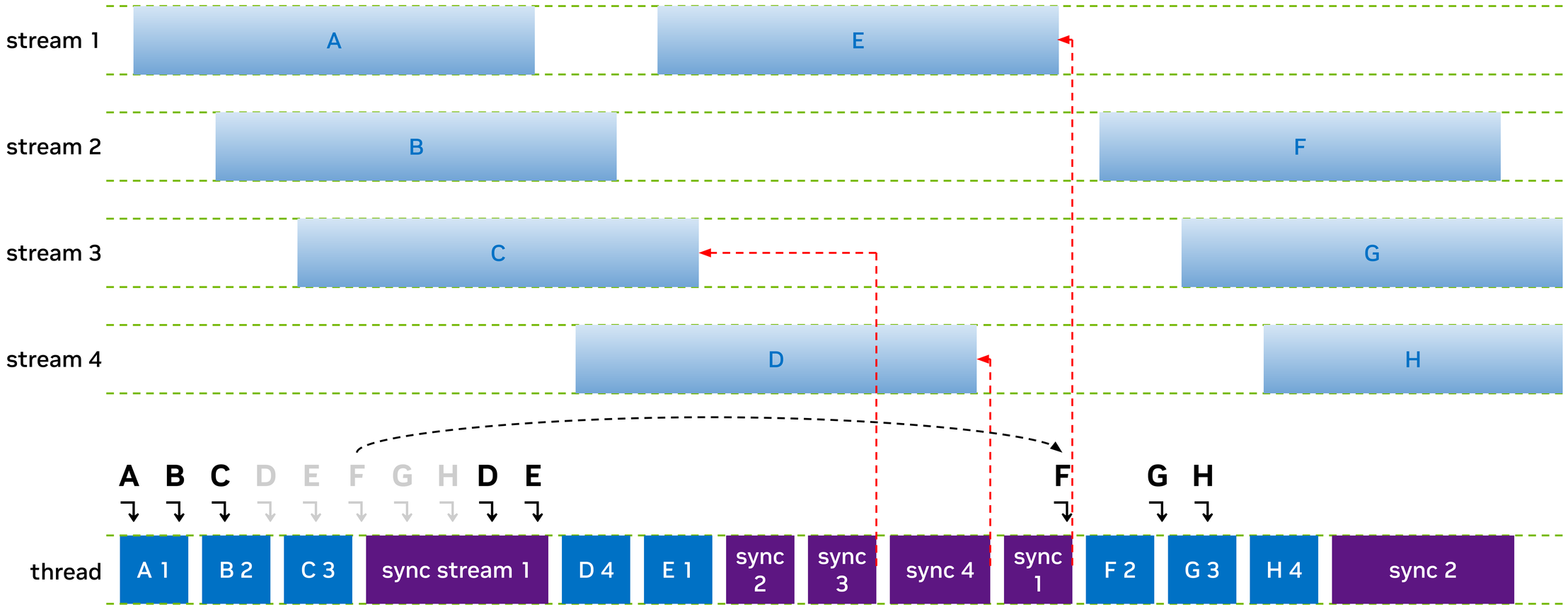
Pipeline Scheduling

Interpretation



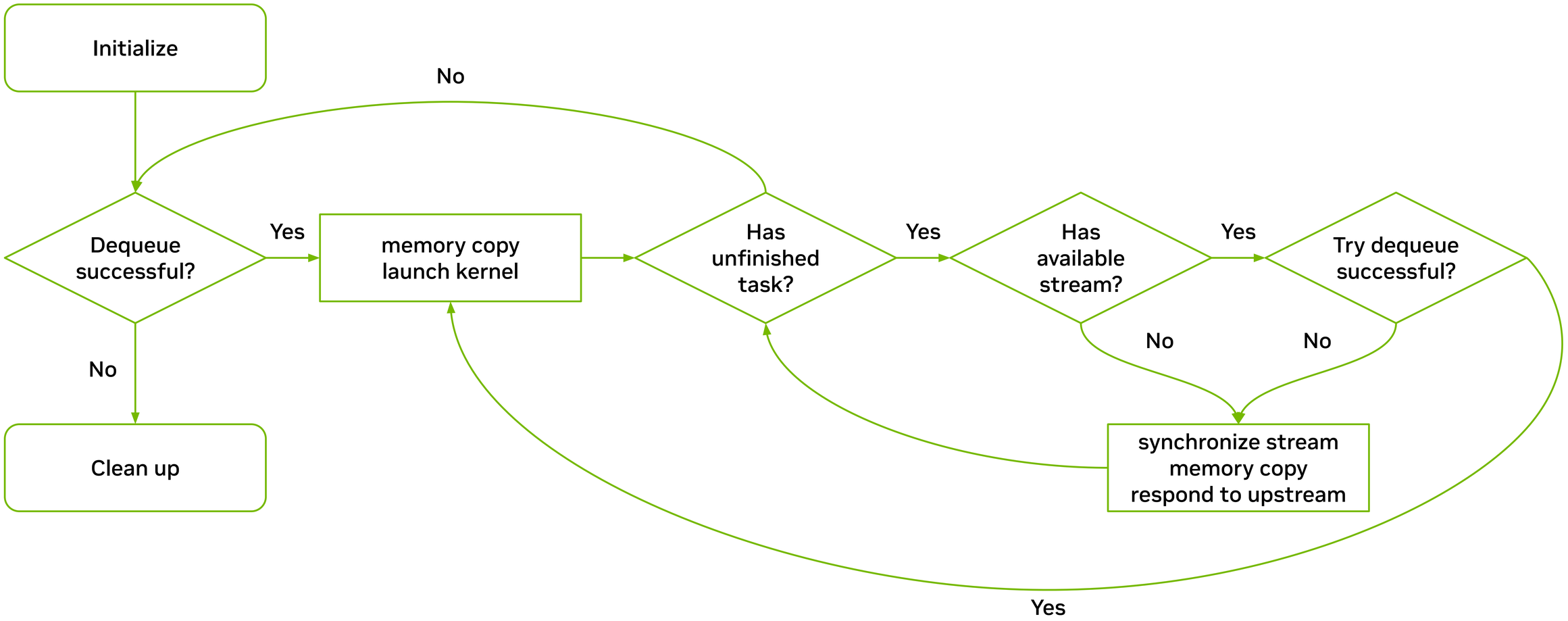
Pipeline Scheduling

Interpretation



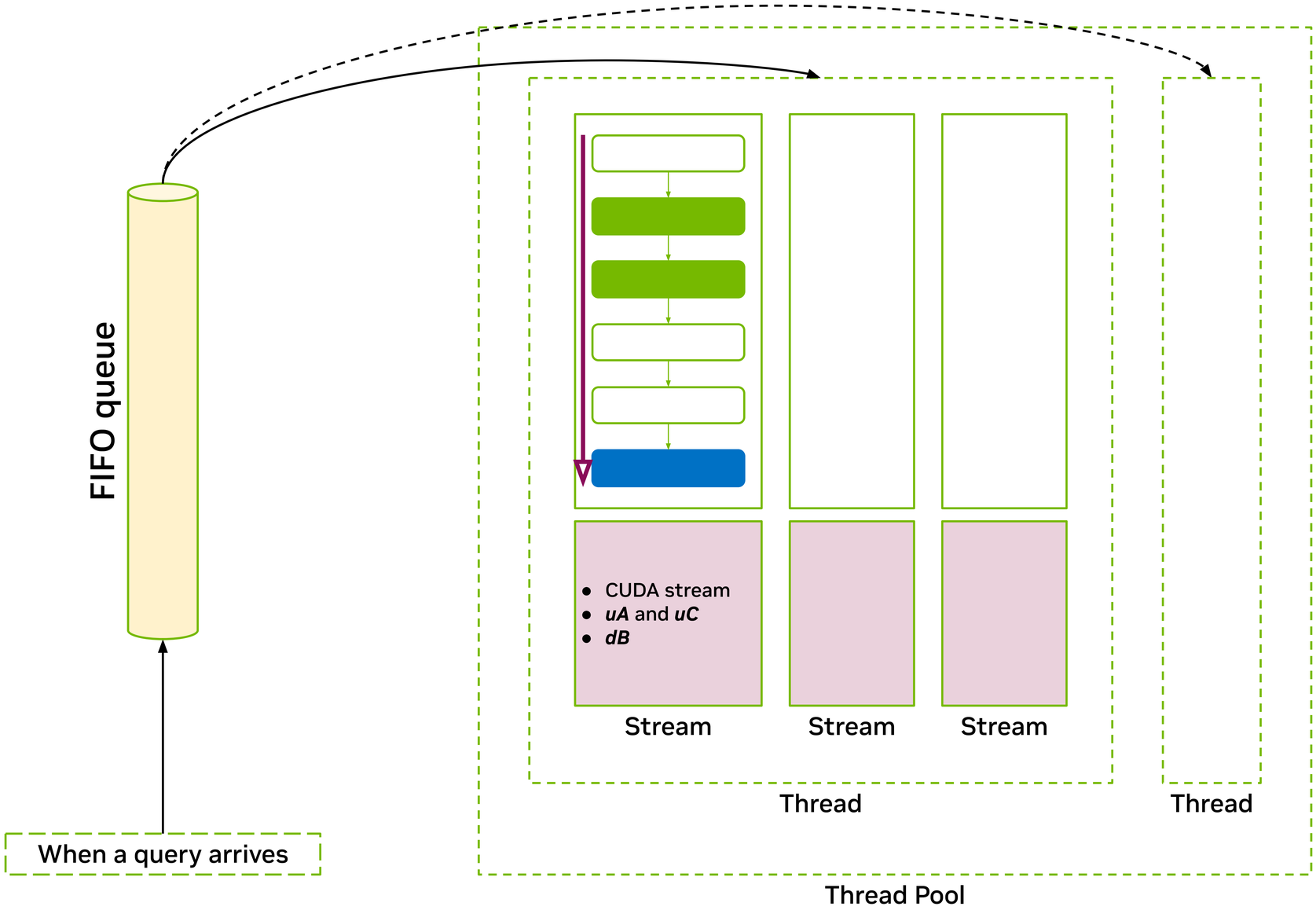
Pipeline Scheduling

Flowchart



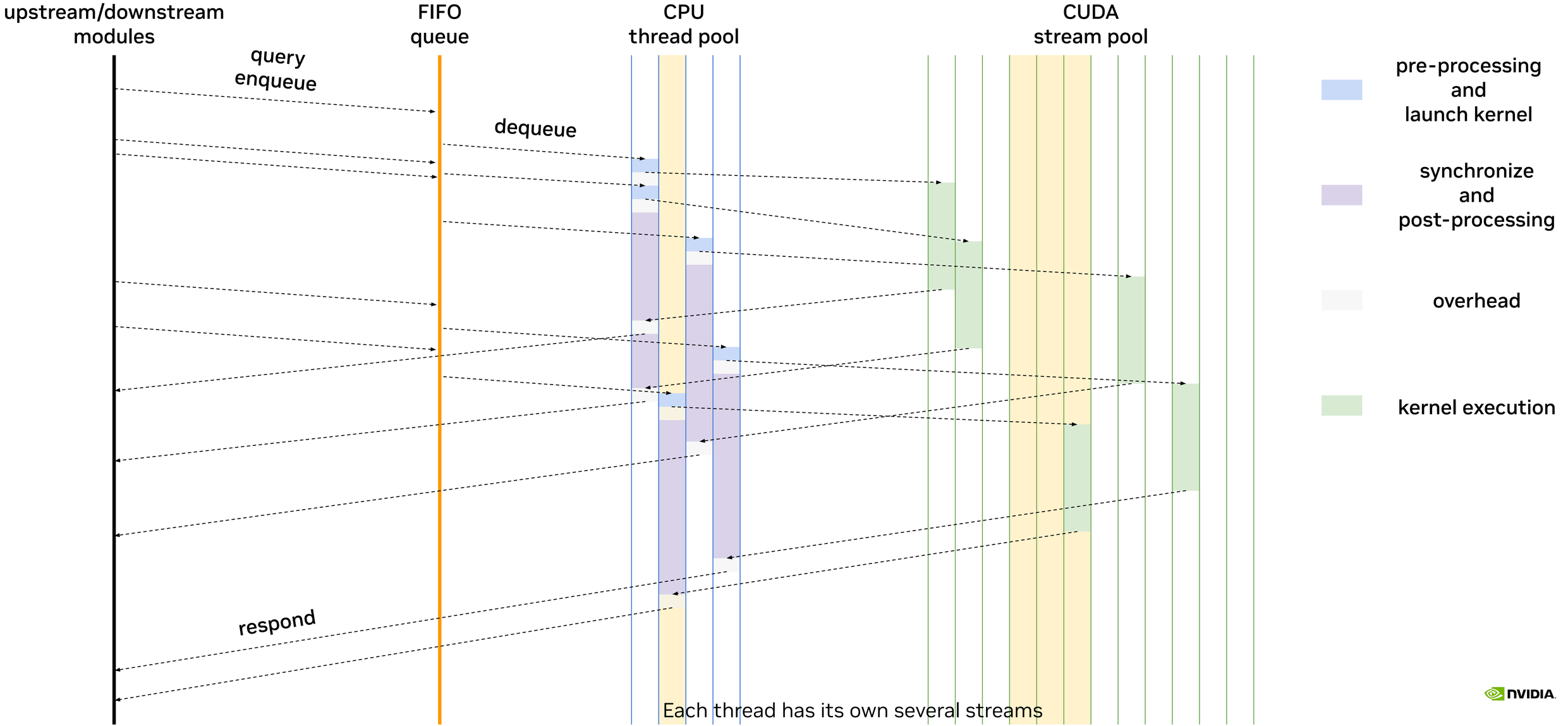
Pipeline Scheduling

Implementation



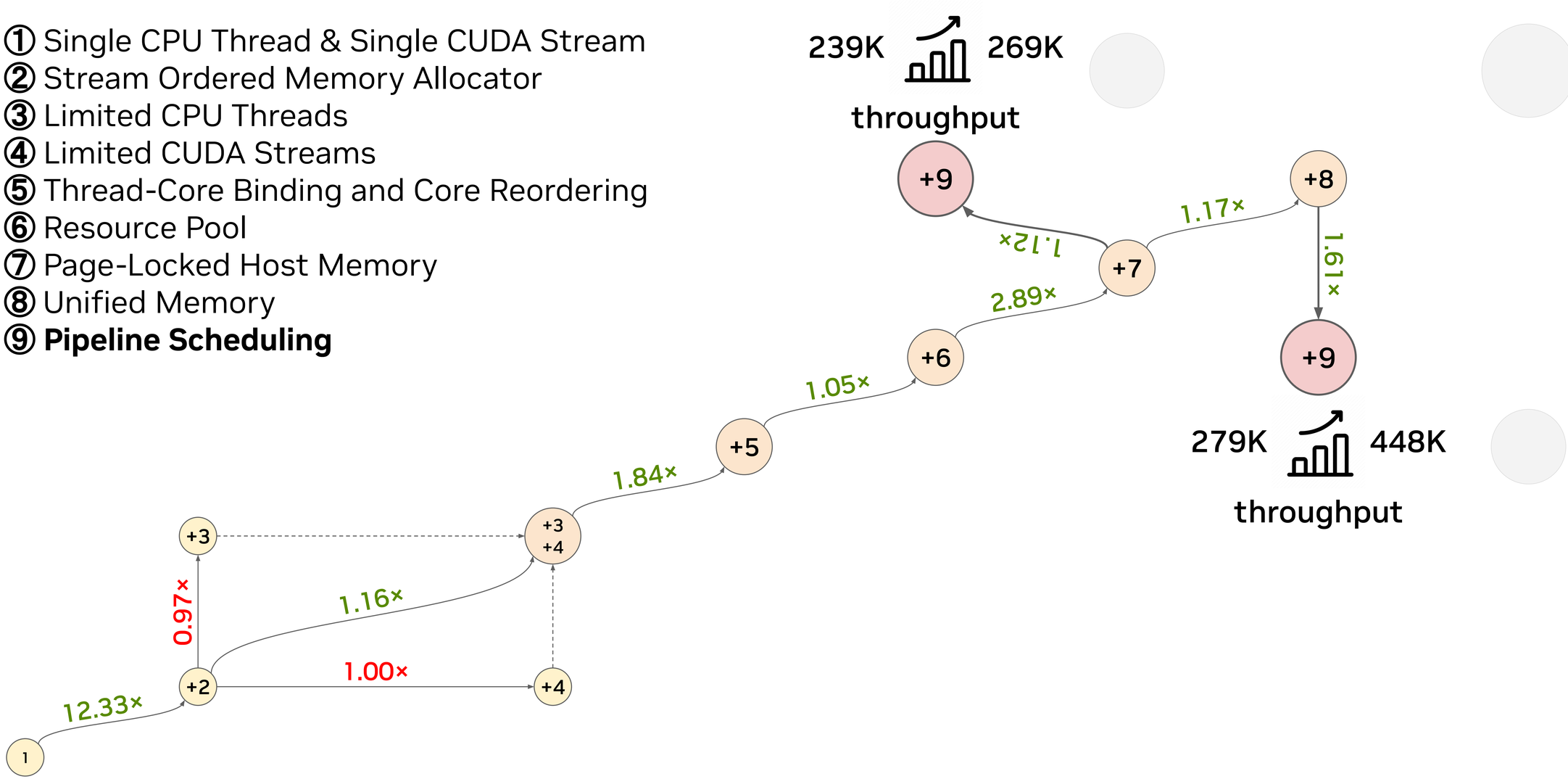
Pipeline Scheduling

Implementation



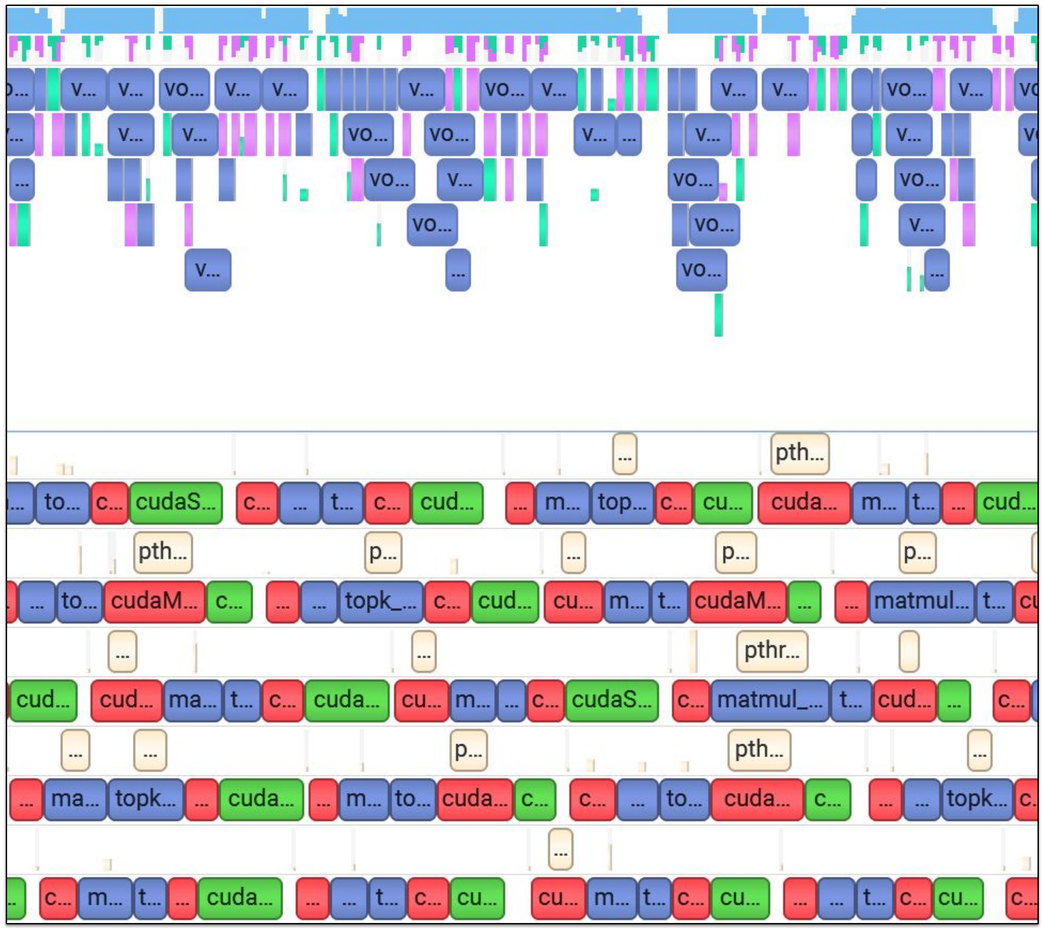
Each thread has its own several streams

① Single CPU Thread & Single CUDA Stream
② Stream Ordered Memory Allocator
③ Limited CPU Threads
④ Limited CUDA Streams
⑤ Thread-Core Binding and Core Reordering
⑥ Resource Pool
⑦ Page-Locked Host Memory
⑧ Unified Memory
⑨ **Pipeline Scheduling**



Pipeline Scheduling

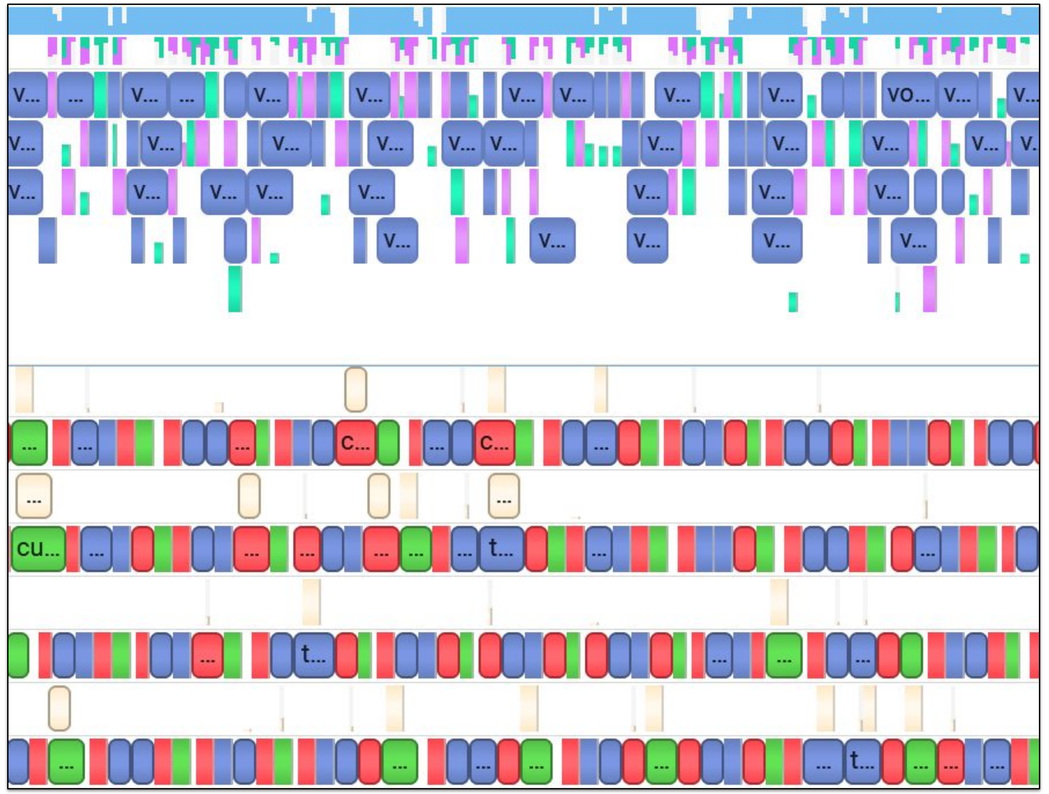
Comparison



Before Pipeline Scheduling

num_threads=8

VS

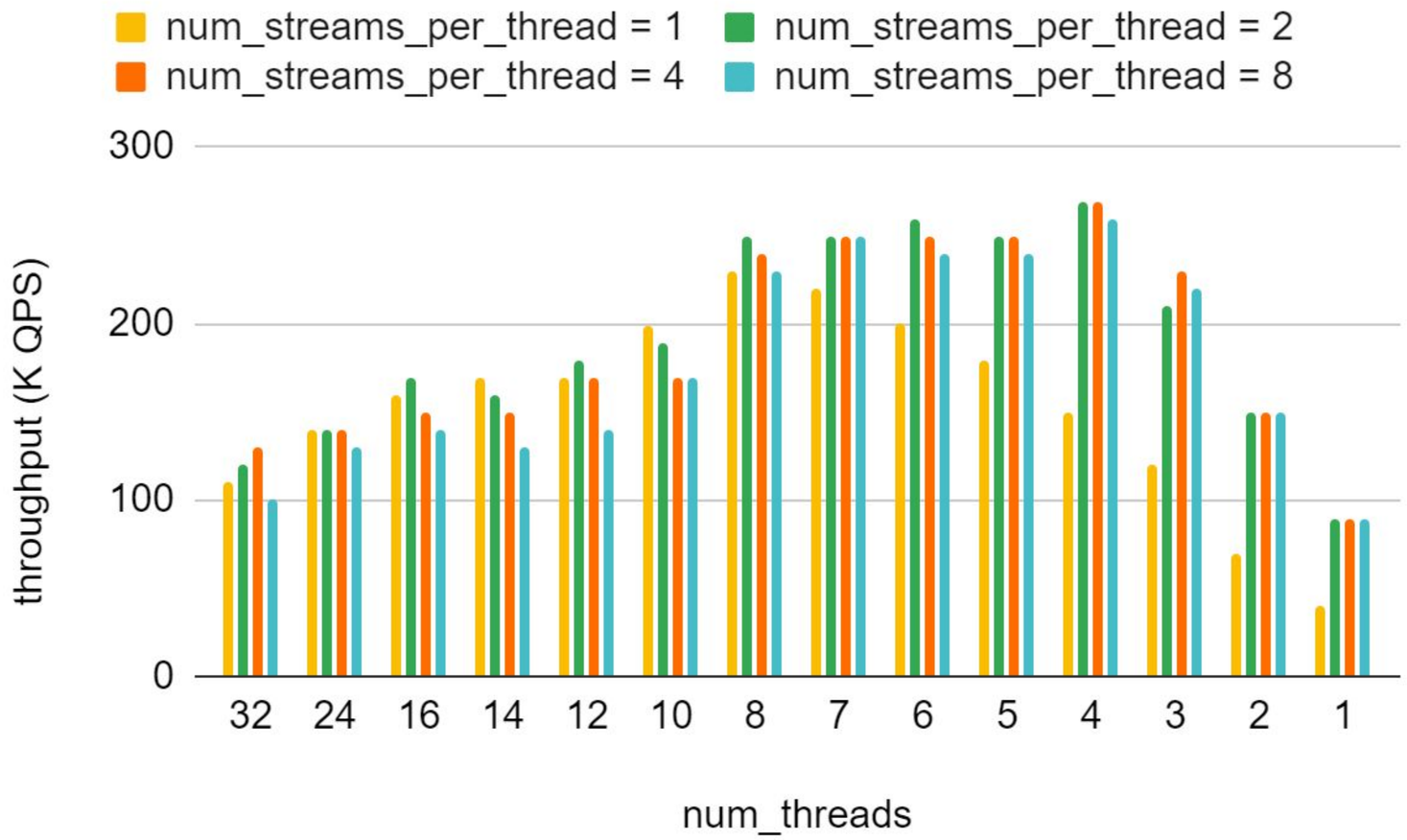


After Pipeline Scheduling

num_threads=4, num_streams_per_thread=2

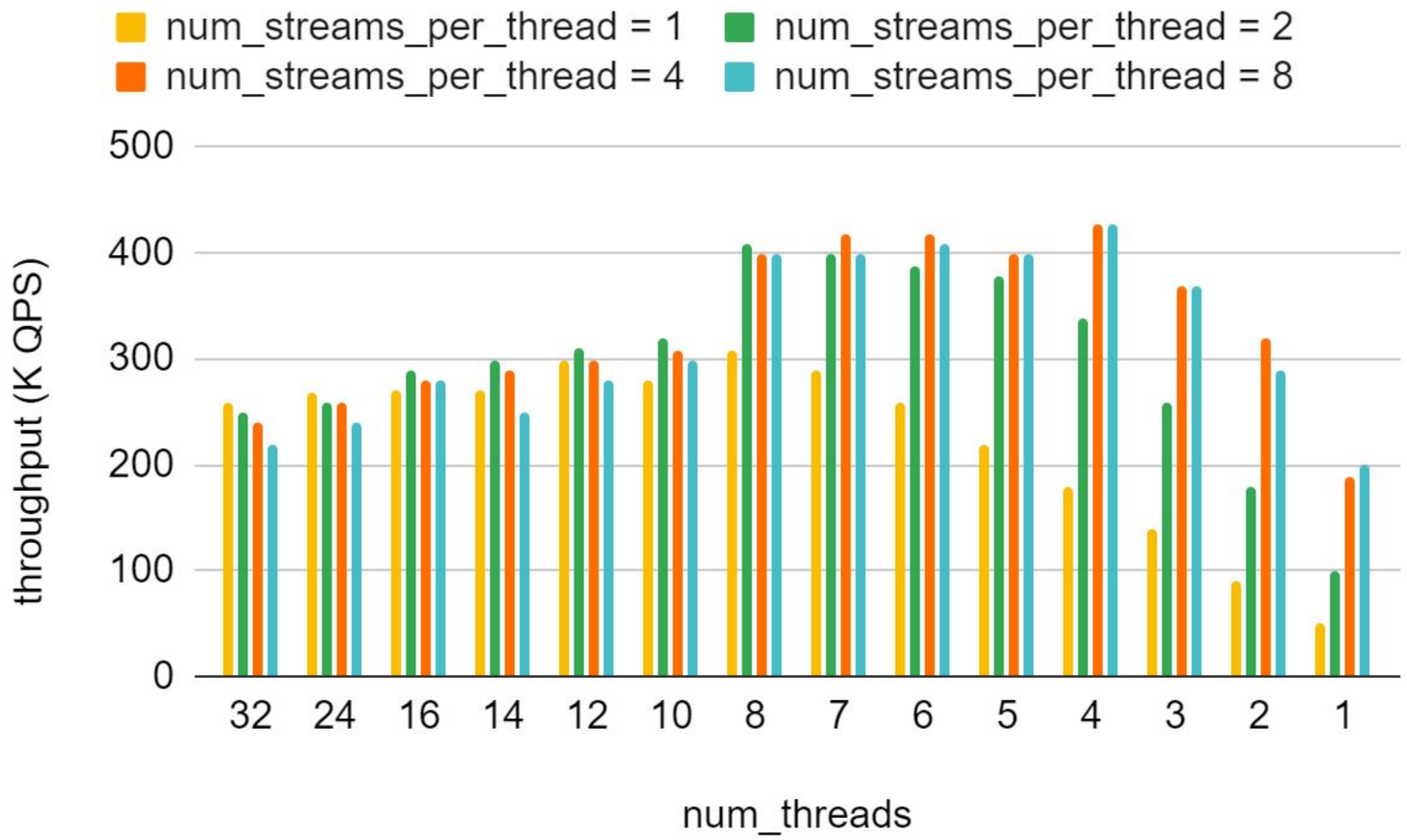
Pipeline Scheduling

Comparison

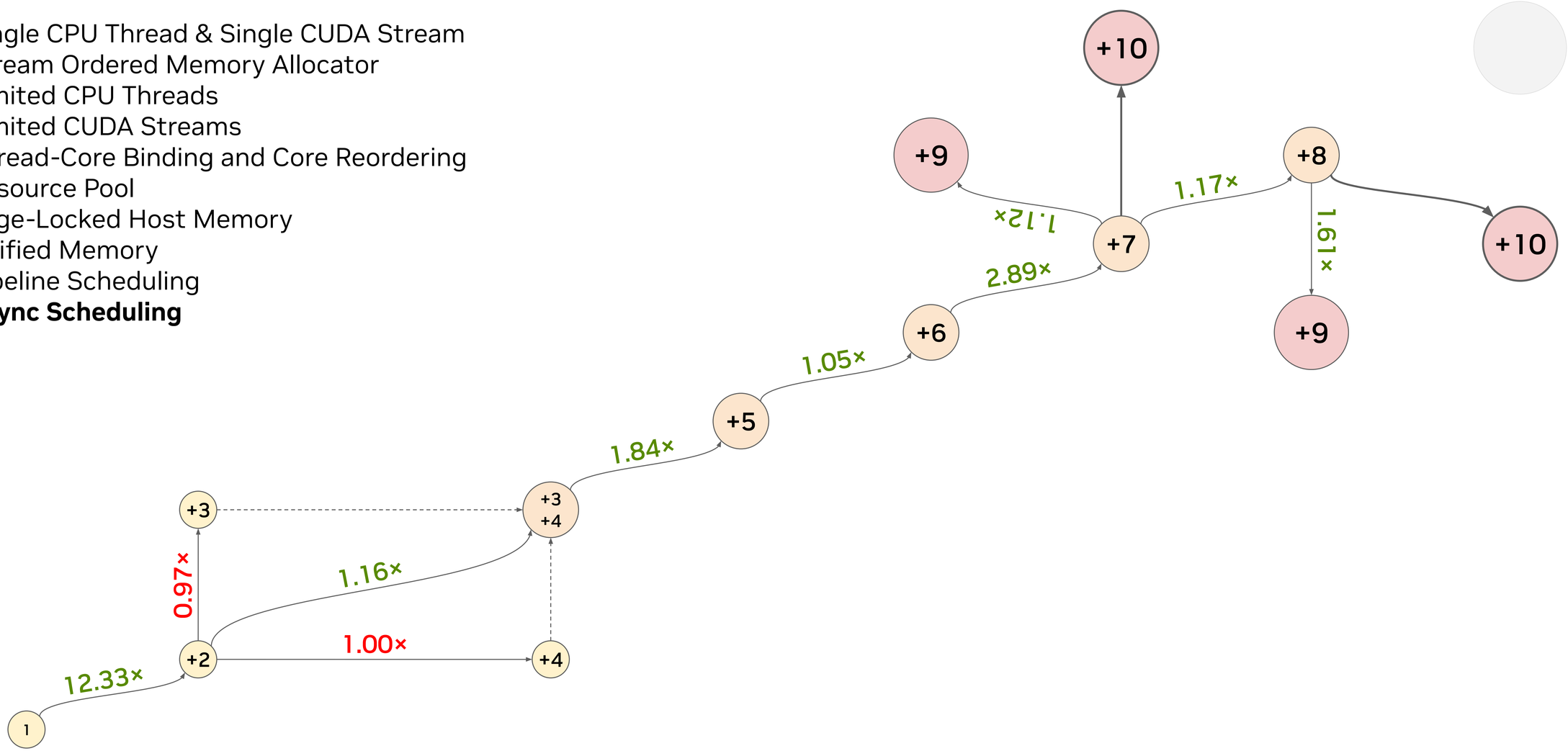


Pipeline Scheduling

Comparison



① Single CPU Thread & Single CUDA Stream
② Stream Ordered Memory Allocator
③ Limited CPU Threads
④ Limited CUDA Streams
⑤ Thread-Core Binding and Core Reordering
⑥ Resource Pool
⑦ Page-Locked Host Memory
⑧ Unified Memory
⑨ Pipeline Scheduling
⑩ **Async Scheduling**

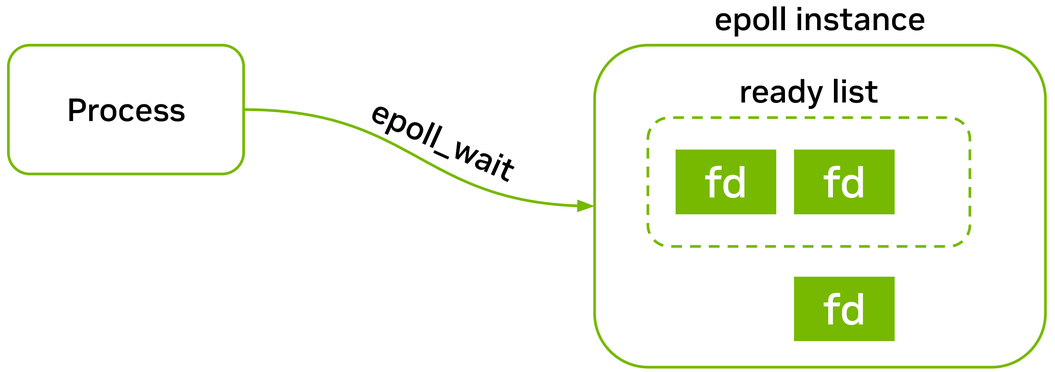


Async Scheduling

Epoll

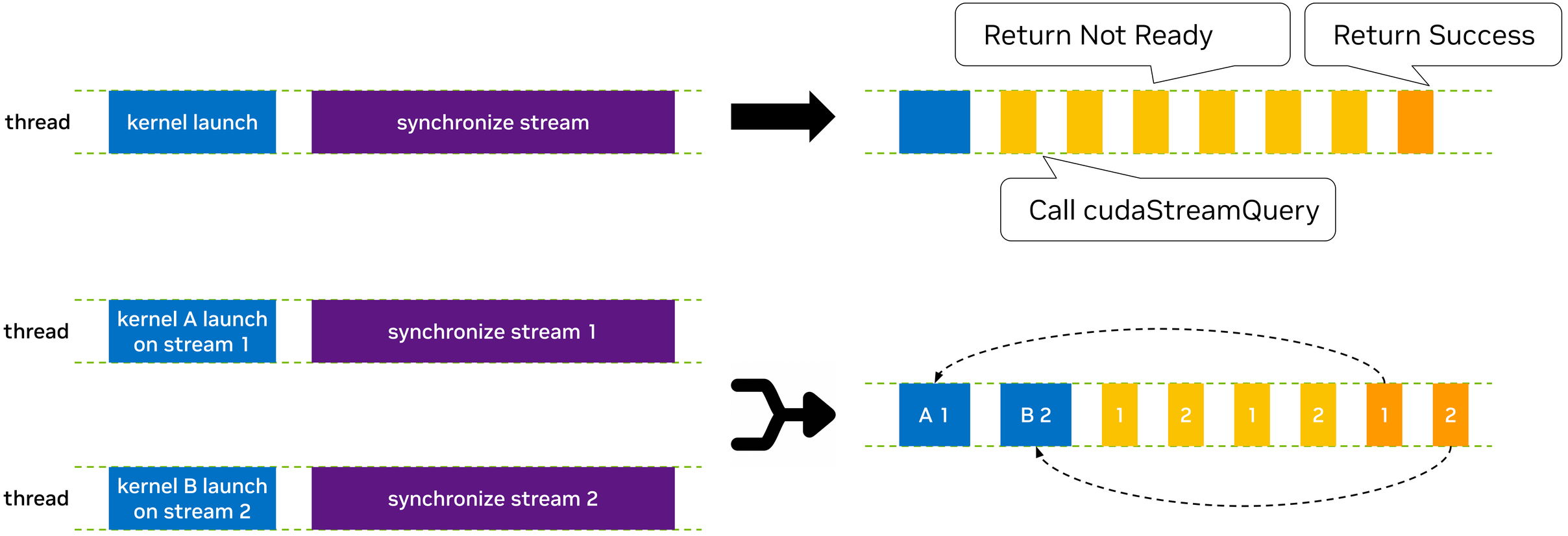
```
epfd = epoll_create(1);
epoll_ctl(epfd, EPOLL_CTL_ADD, listen_sock, …);

for (;;) {
    nfds = epoll_wait(epfd, events, MAX_EVENTS, -1);
    for (i = 0; i < nfds; i++) {
      if (events[i].data.fd == listen_sock) {
        /* handle new connection */
        conn_sock = accept(…);
        /* … */
        epoll_ctl(epfd, EPOLL_CTL_ADD, conn_sock, …);
      } else if (events[i].events & EPOLLIN) {
        /* handle EPOLLIN event */
        /* … */
      } else {
        /* handle unexpected */
      }
      /* check if the connection is closing */
      if (events[i].events & (EPOLLRDHUP | EPOLLHUP)) {
        /* … */
        epoll_ctl(epfd, EPOLL_CTL_DEL, events[i].data.fd,
        …);
        close(events[i].data.fd);
        continue;
      }
    }
  }
```



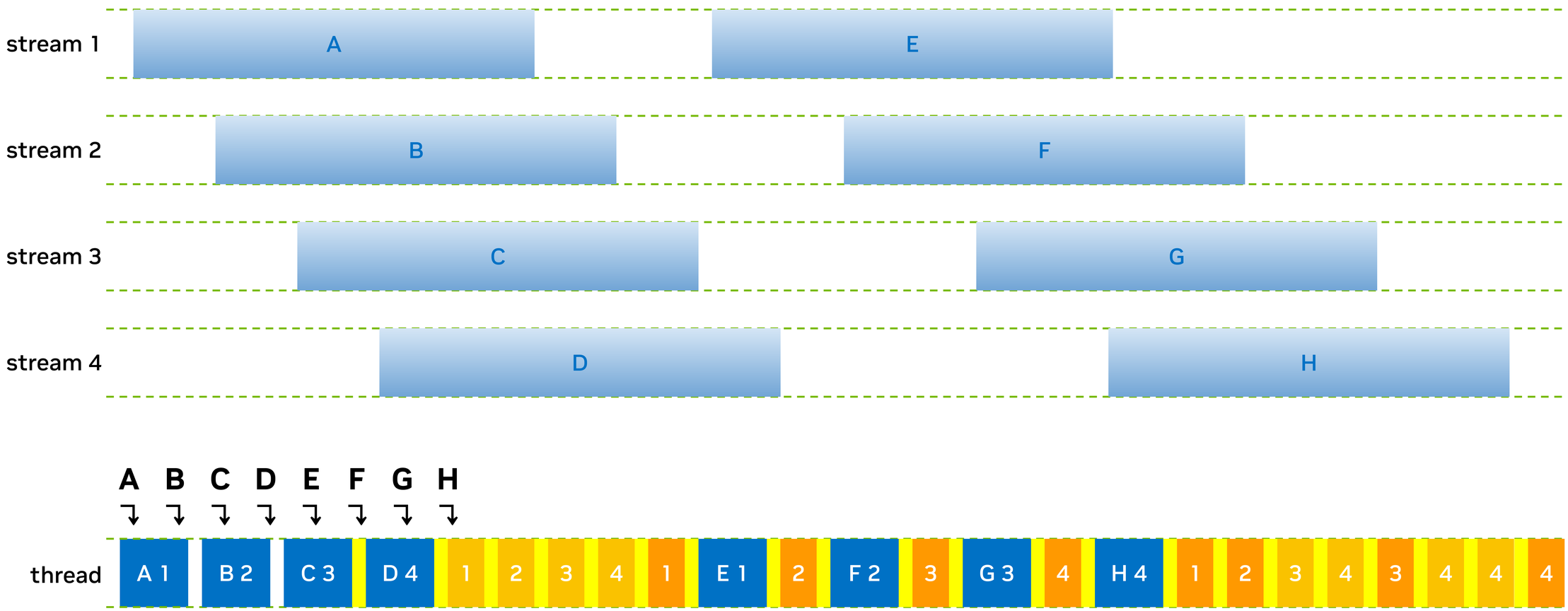
Async Scheduling

Interpretation



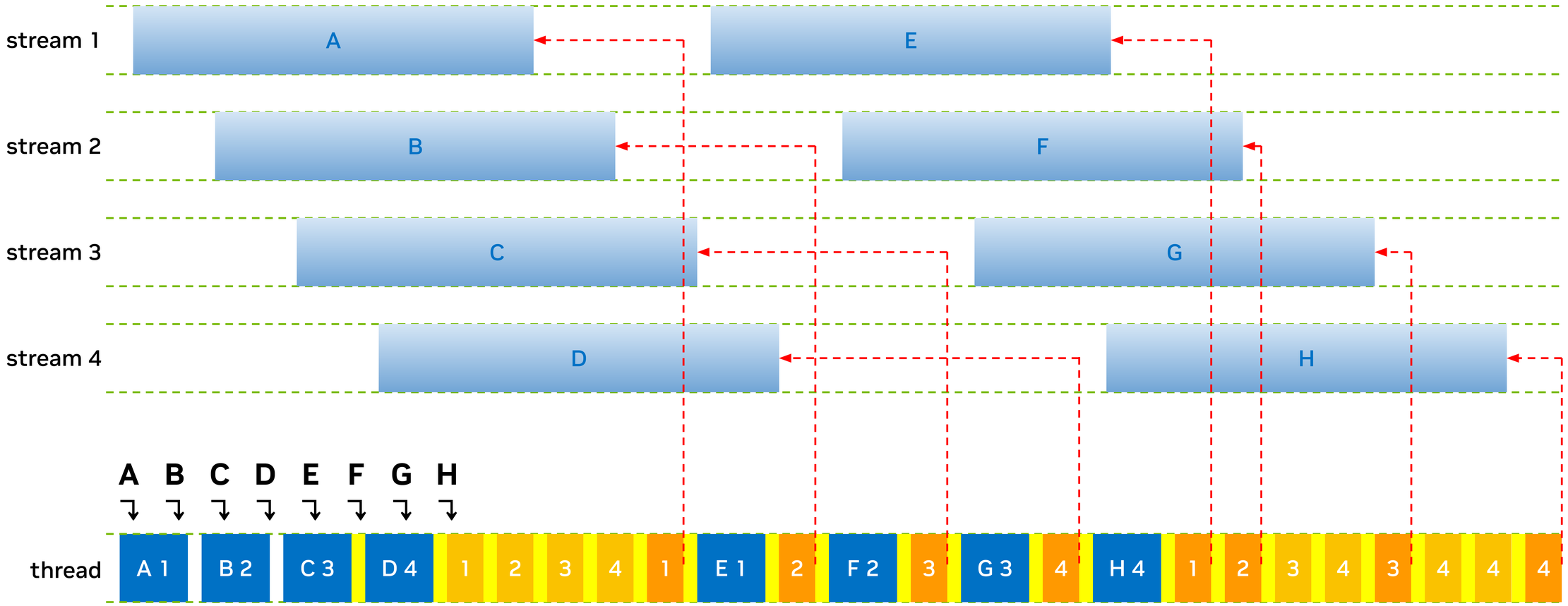
Async Scheduling

Interpretation



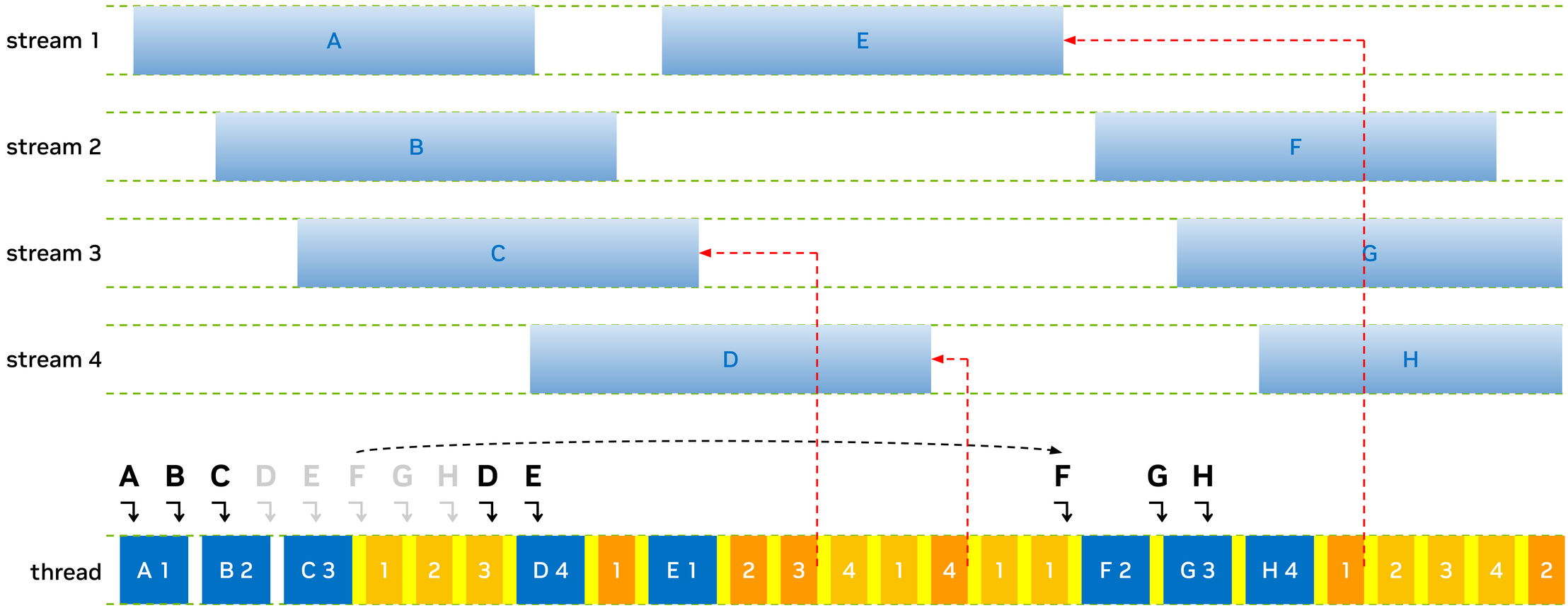
Async Scheduling

Interpretation



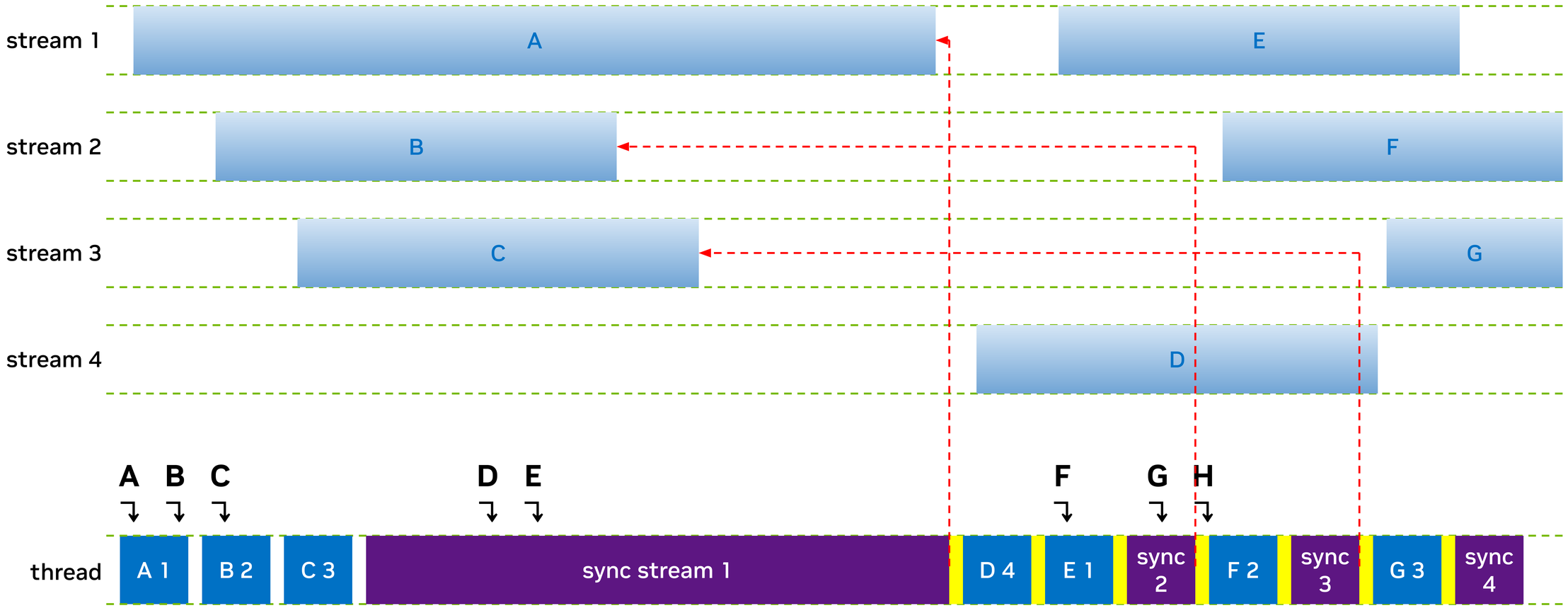
Async Scheduling

Interpretation



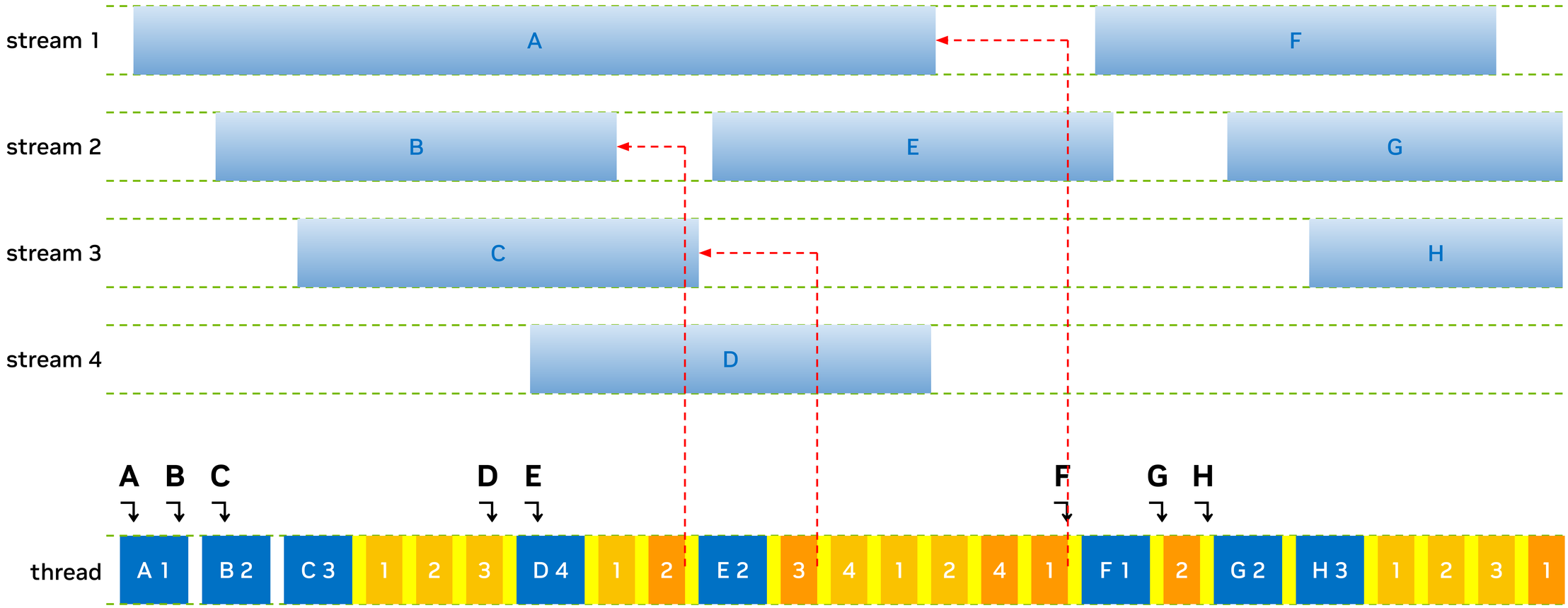
Async Scheduling

Interpretation



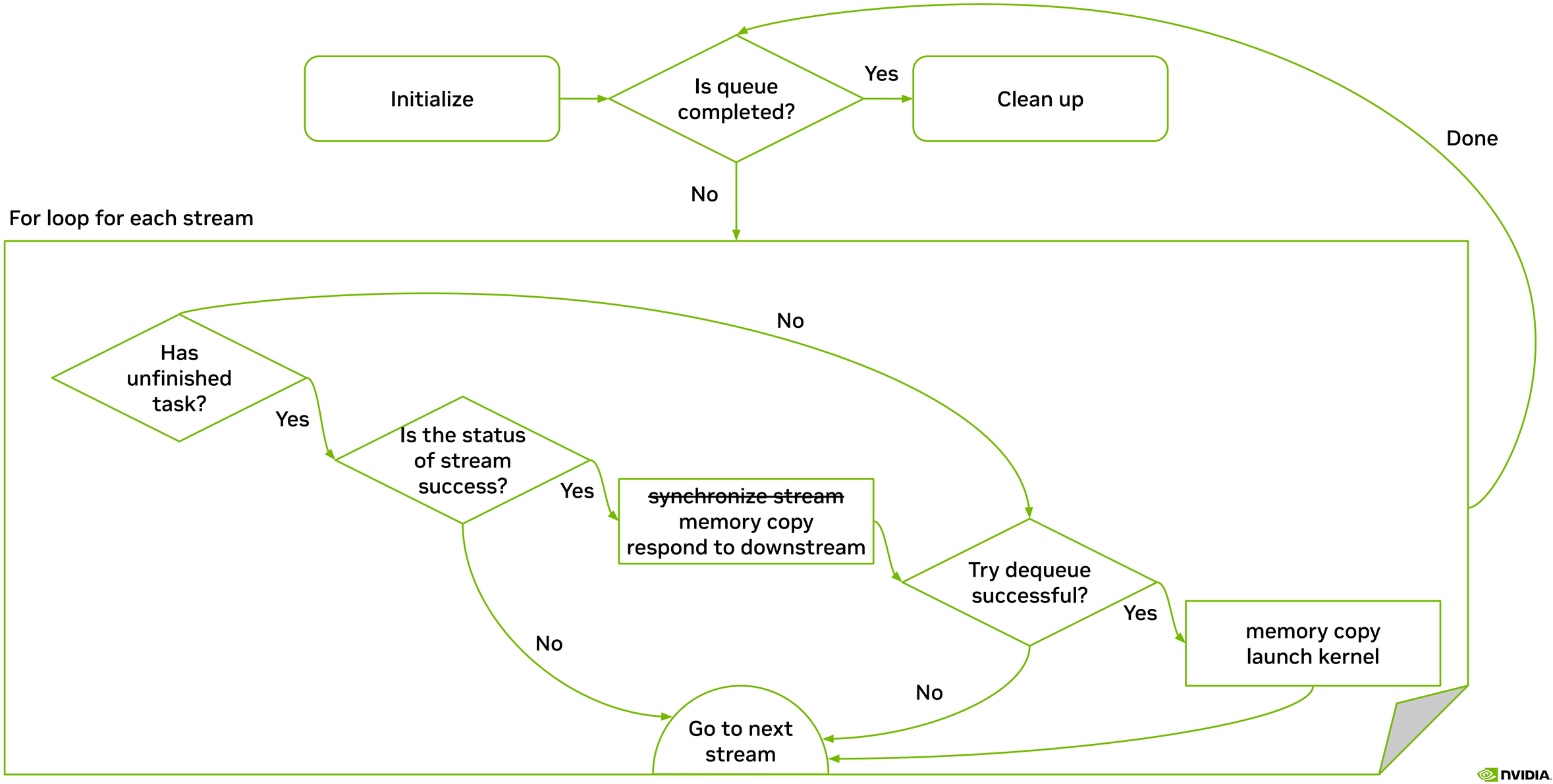
Async Scheduling

Interpretation

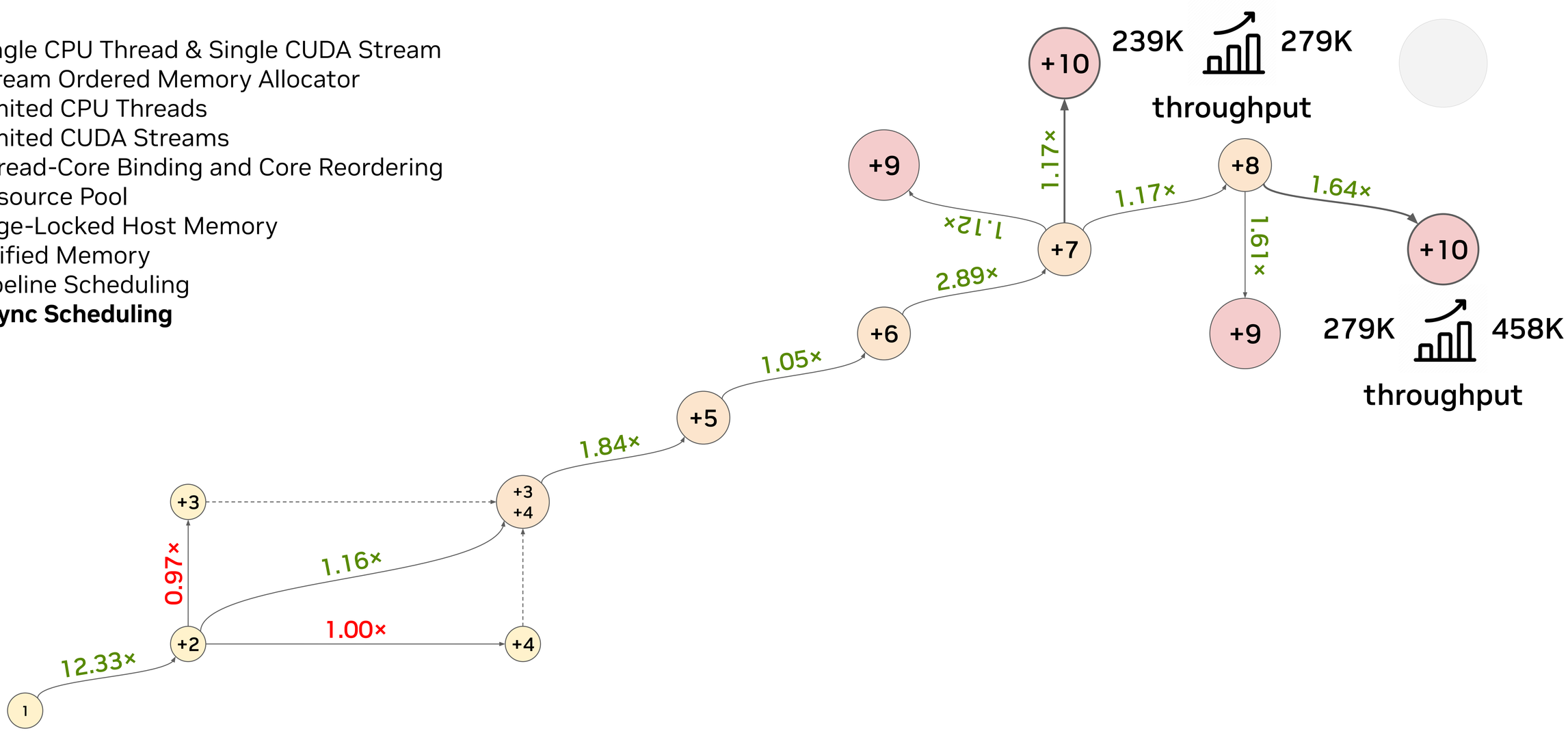


Async Scheduling

Flowchart

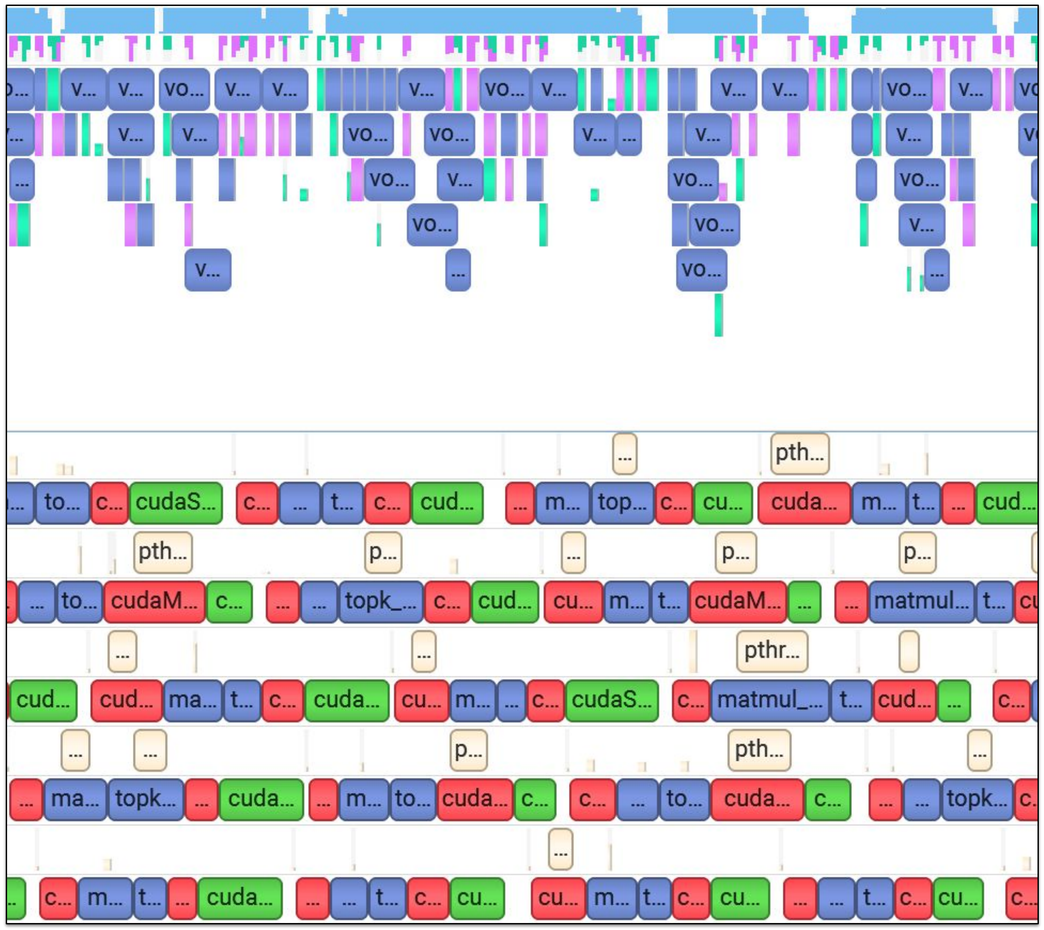


① Single CPU Thread & Single CUDA Stream
② Stream Ordered Memory Allocator
③ Limited CPU Threads
④ Limited CUDA Streams
⑤ Thread-Core Binding and Core Reordering
⑥ Resource Pool
⑦ Page-Locked Host Memory
⑧ Unified Memory
⑨ Pipeline Scheduling
⑩ **Async Scheduling**



Async Scheduling

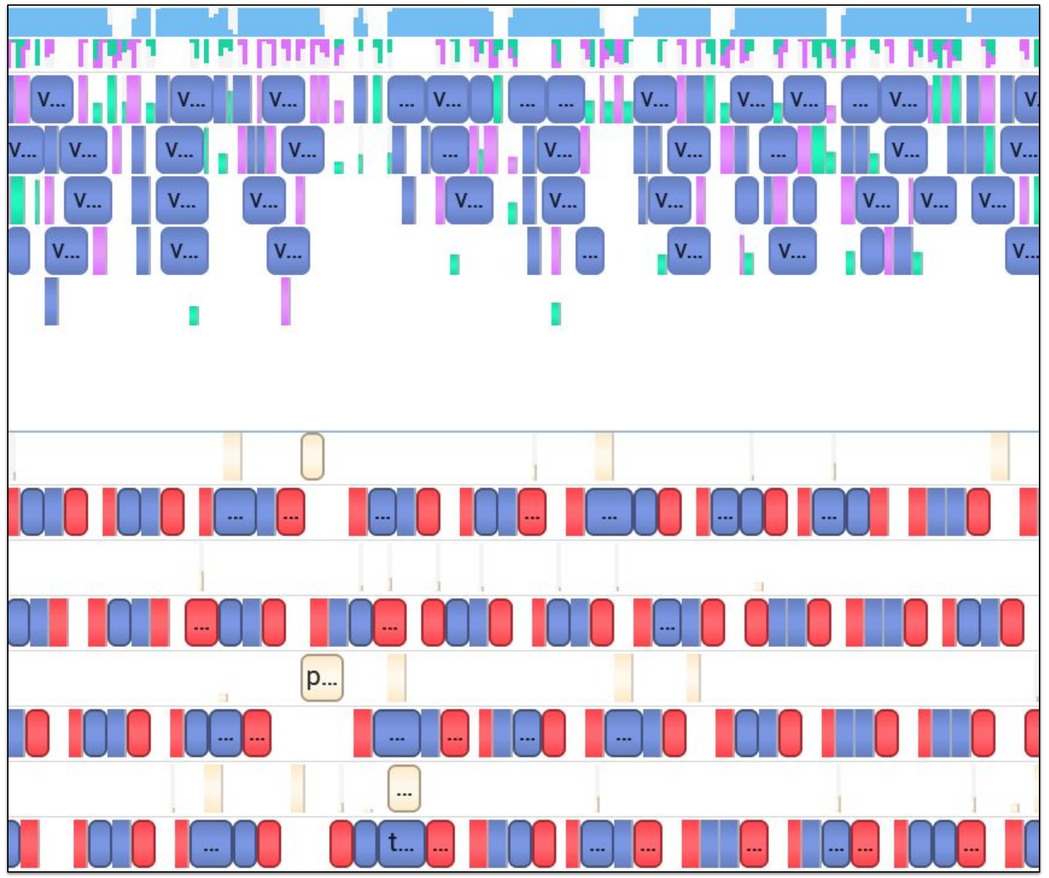
Comparison



Before Async Scheduling

num_threads=8

VS

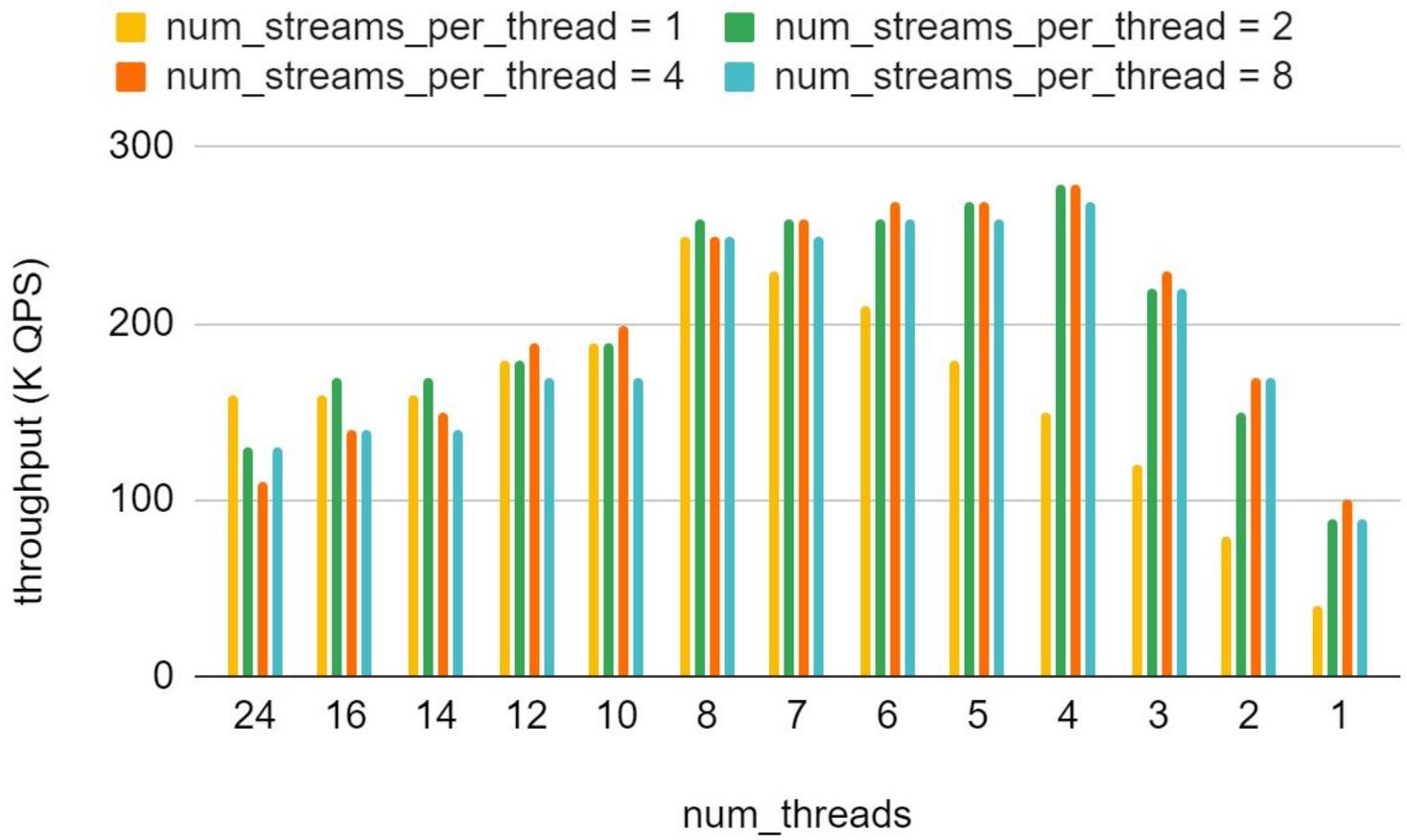


After Async Scheduling

num_threads=4, num_streams_per_thread=2

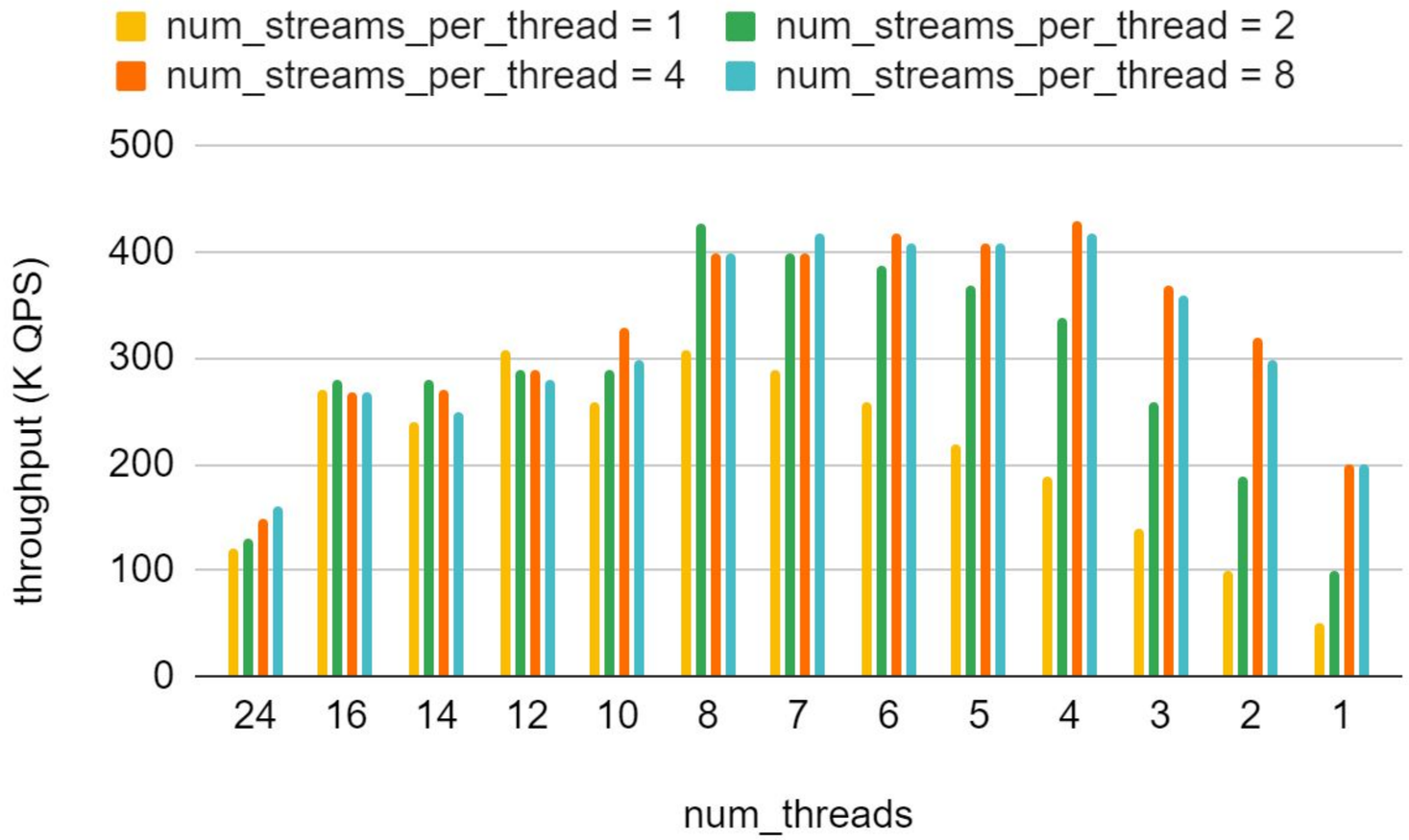
Async Scheduling

Comparison

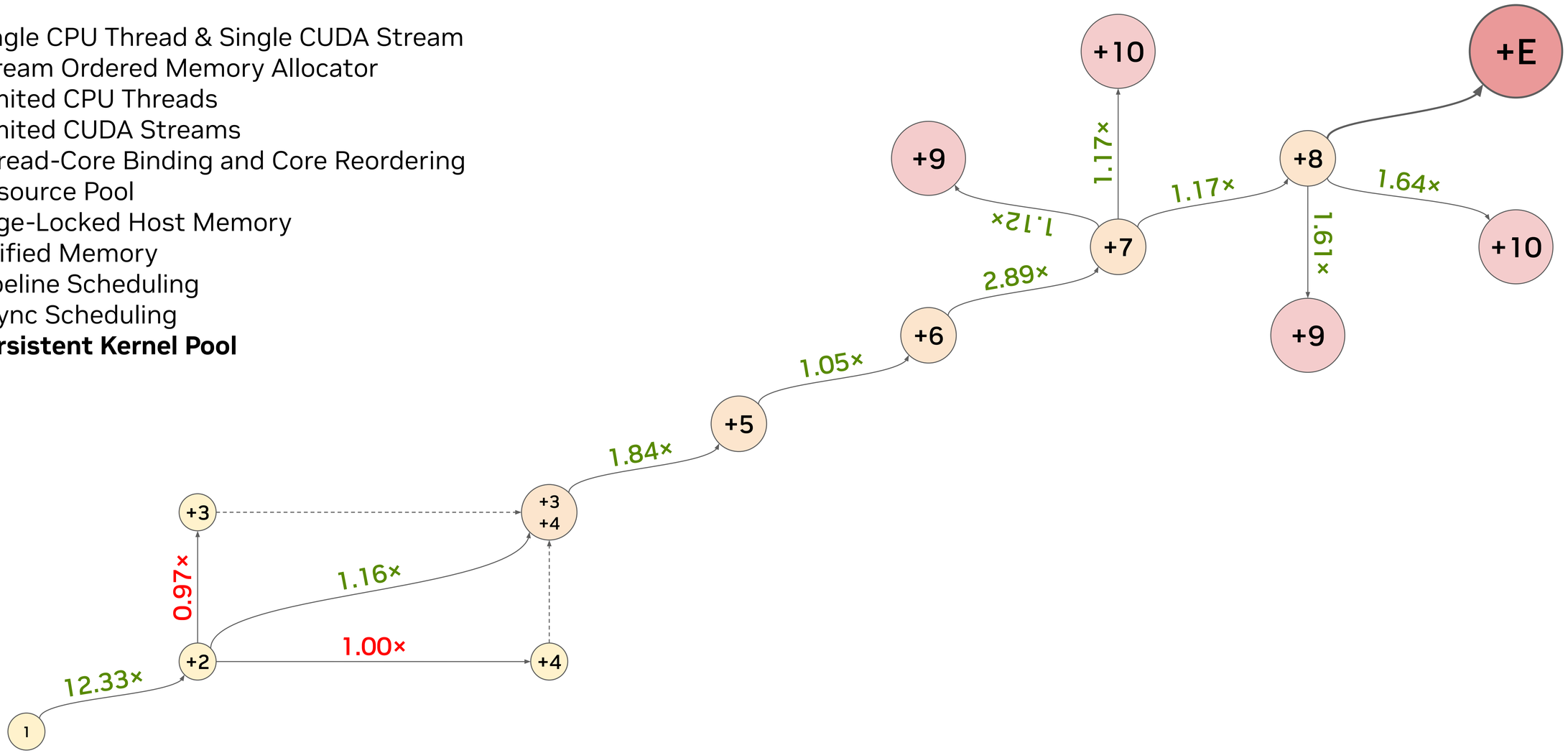


Async Scheduling

Comparison

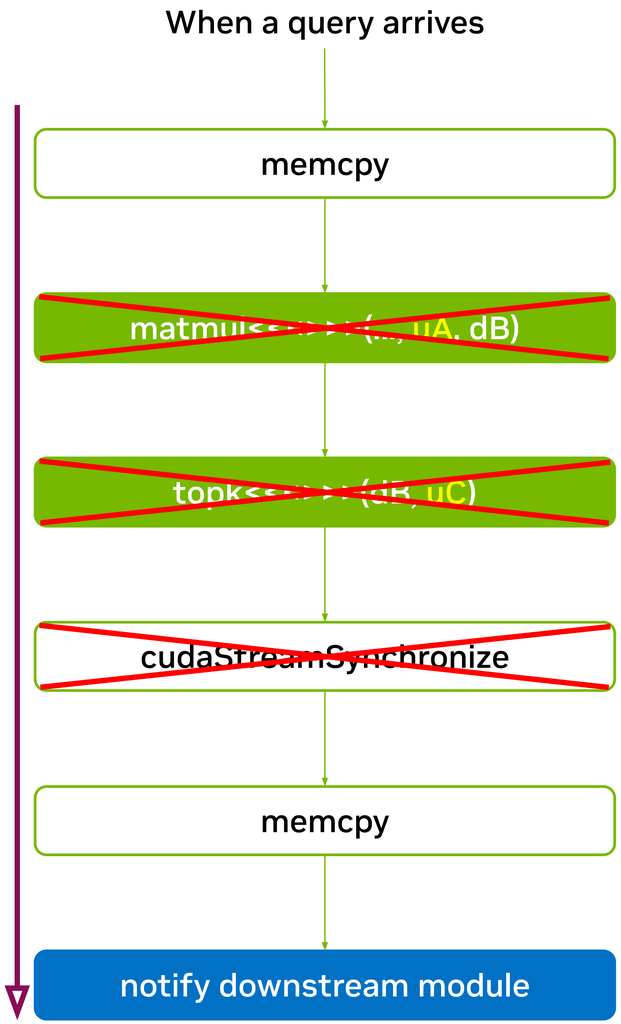


① Single CPU Thread & Single CUDA Stream
② Stream Ordered Memory Allocator
③ Limited CPU Threads
④ Limited CUDA Streams
⑤ Thread-Core Binding and Core Reordering
⑥ Resource Pool
⑦ Page-Locked Host Memory
⑧ Unified Memory
⑨ Pipeline Scheduling
⑩ Async Scheduling
Ⓔ **Persistent Kernel Pool**



Persistent Kernel Pool

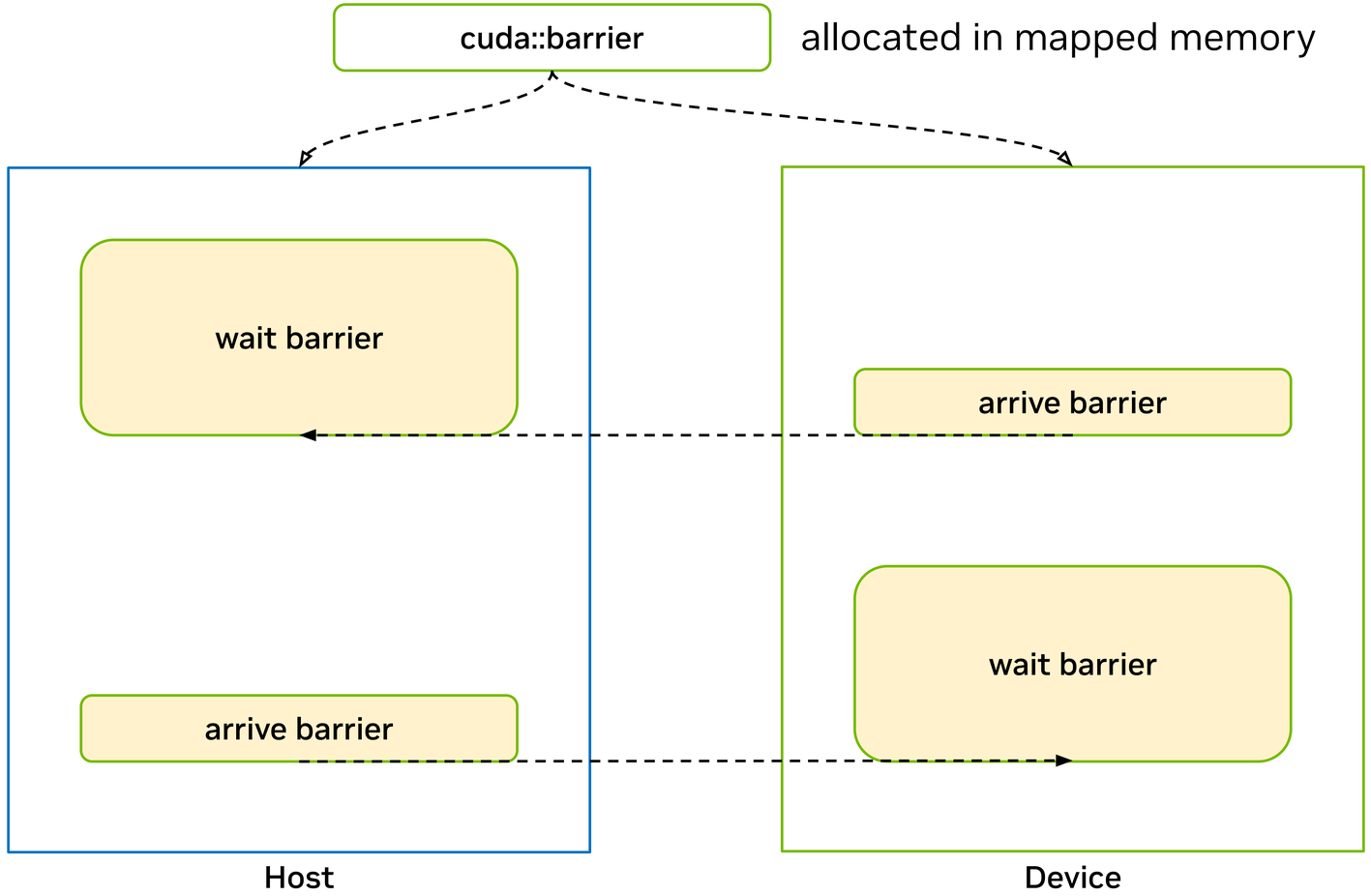
Implementation



Persistent Kernel Pool

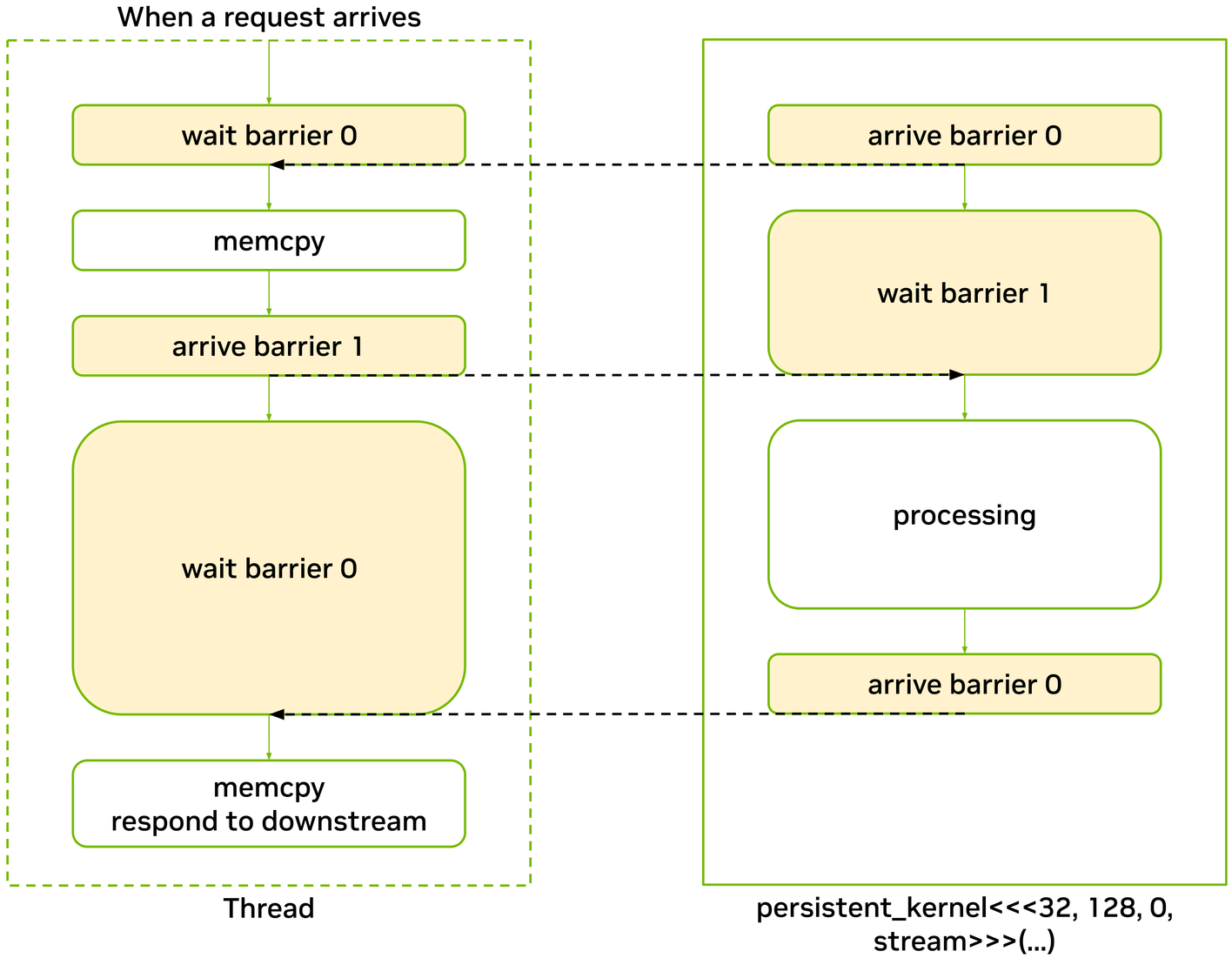
Interpretation

The class ***cuda::barrier<cuda::thread_scope_system>*** provides a synchronization mechanism between host and device.



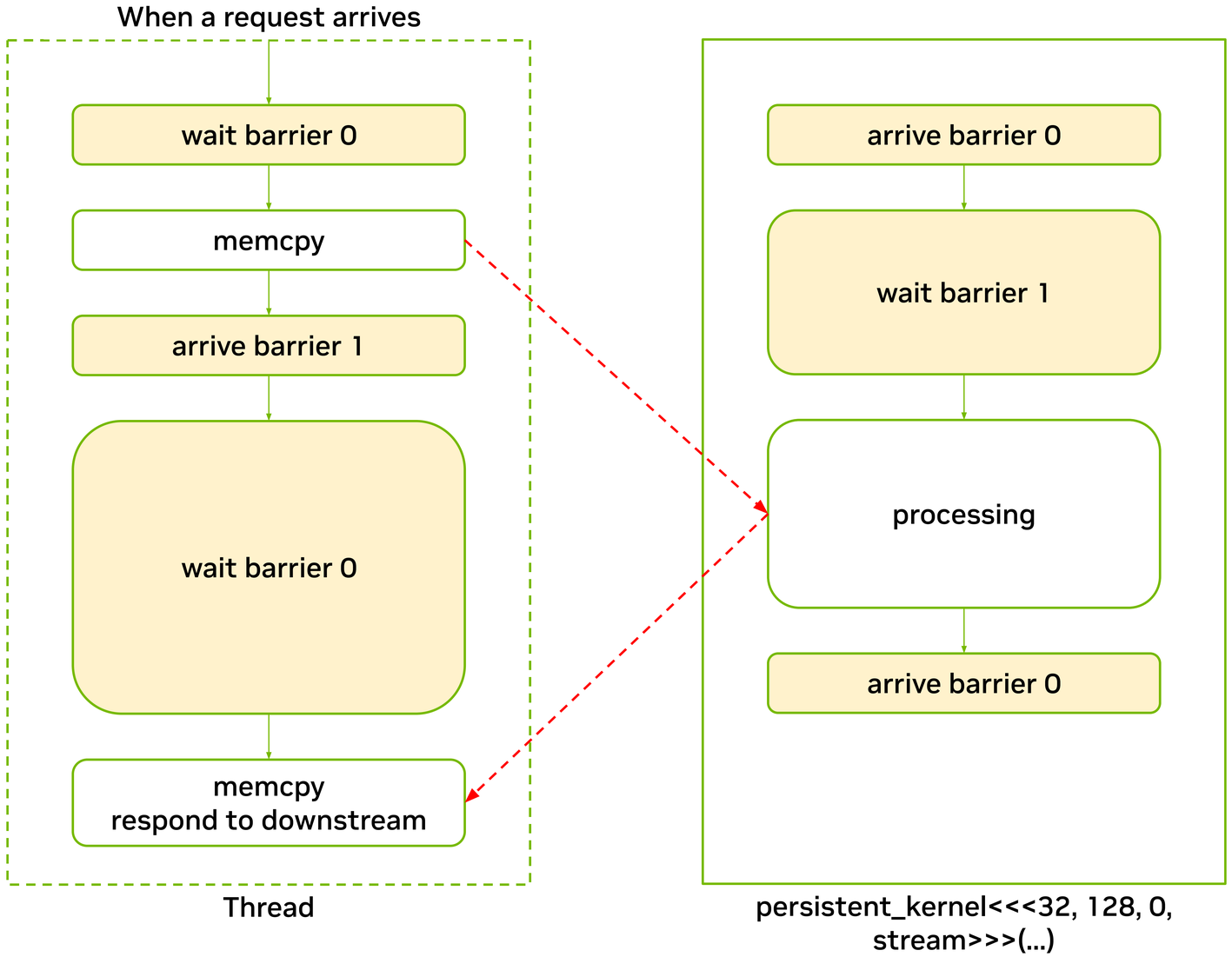
Persistent Kernel Pool

Interpretation



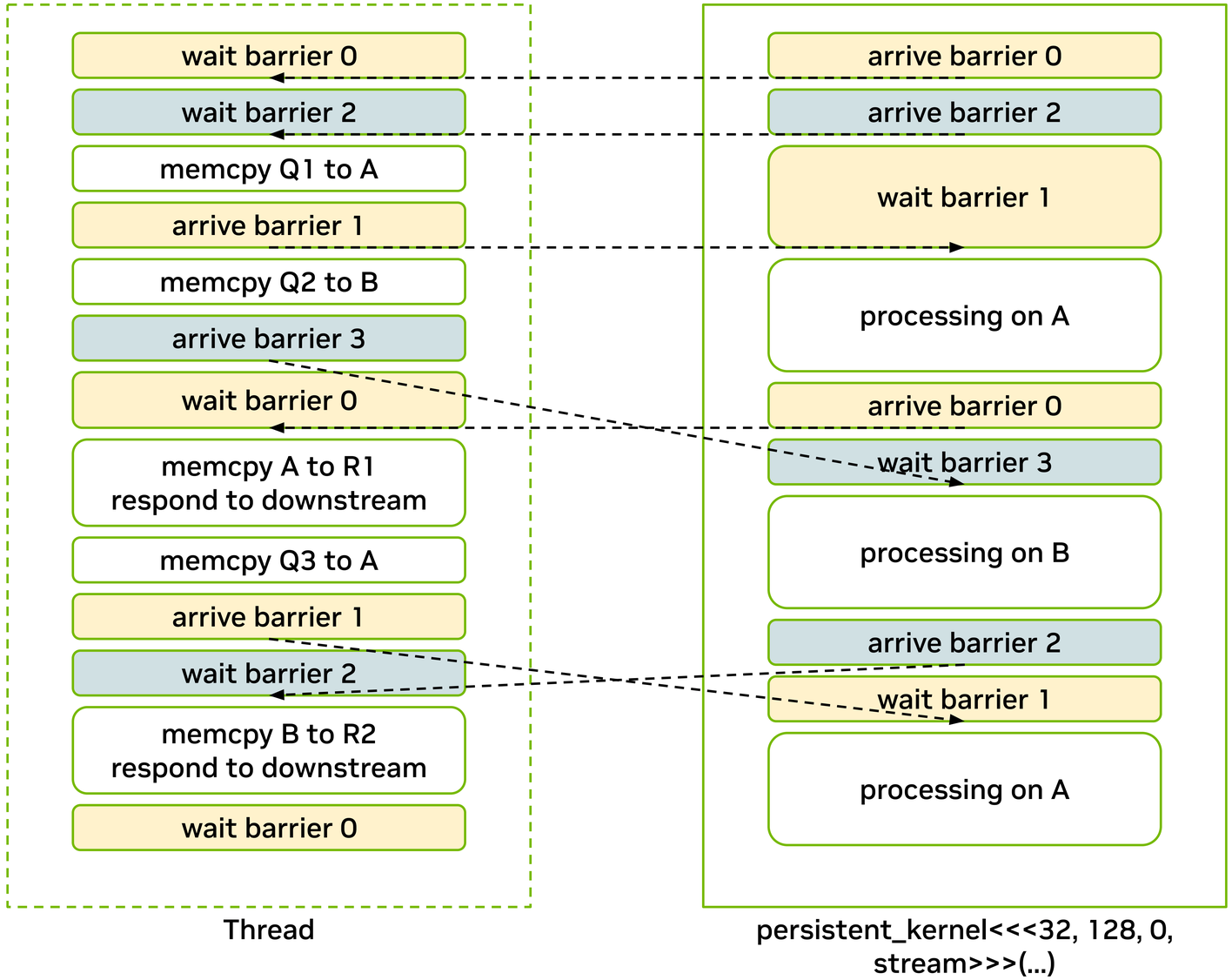
Persistent Kernel Pool

Interpretation



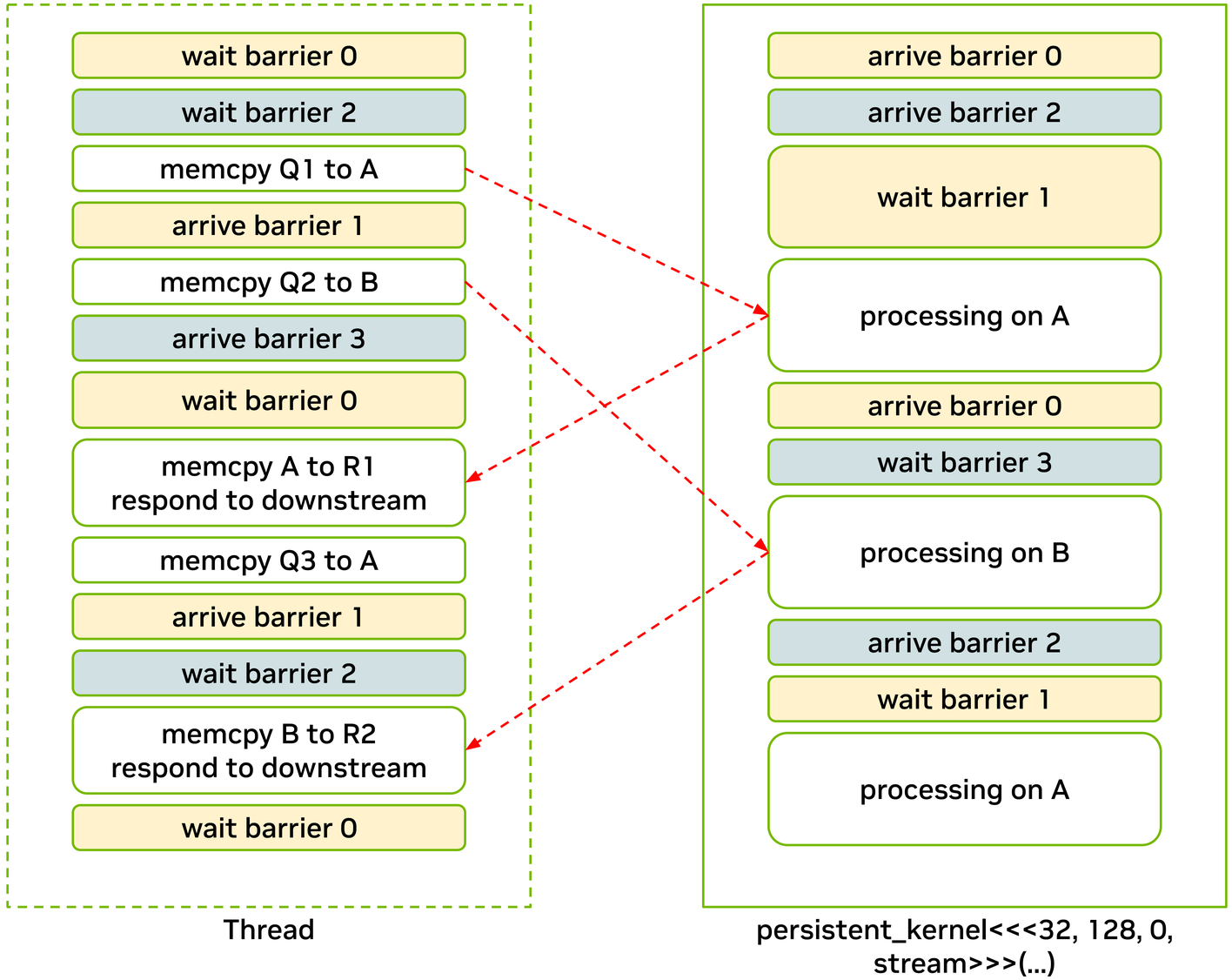
Persistent Kernel Pool

Interpretation



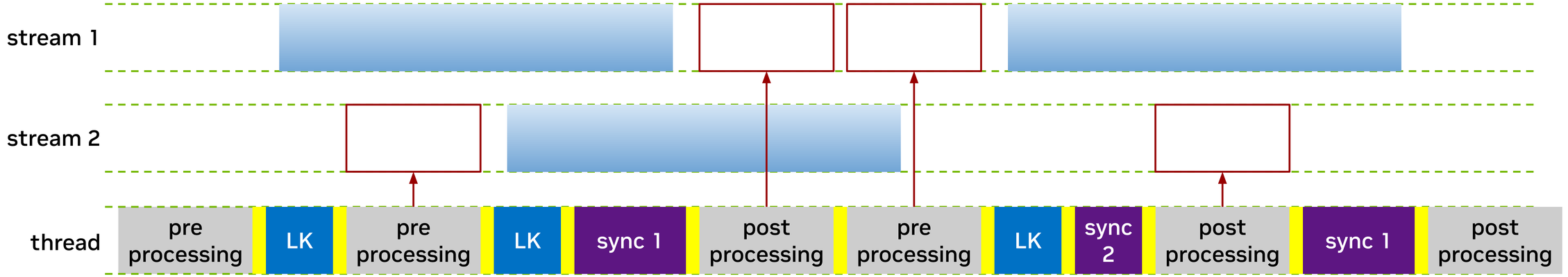
Persistent Kernel Pool

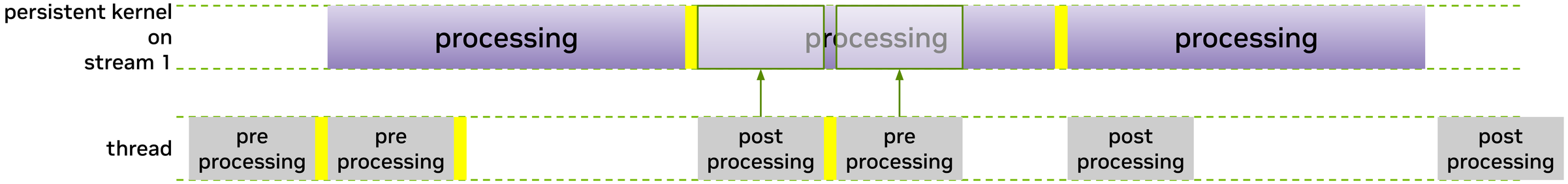
Interpretation



Persistent Kernel Pool

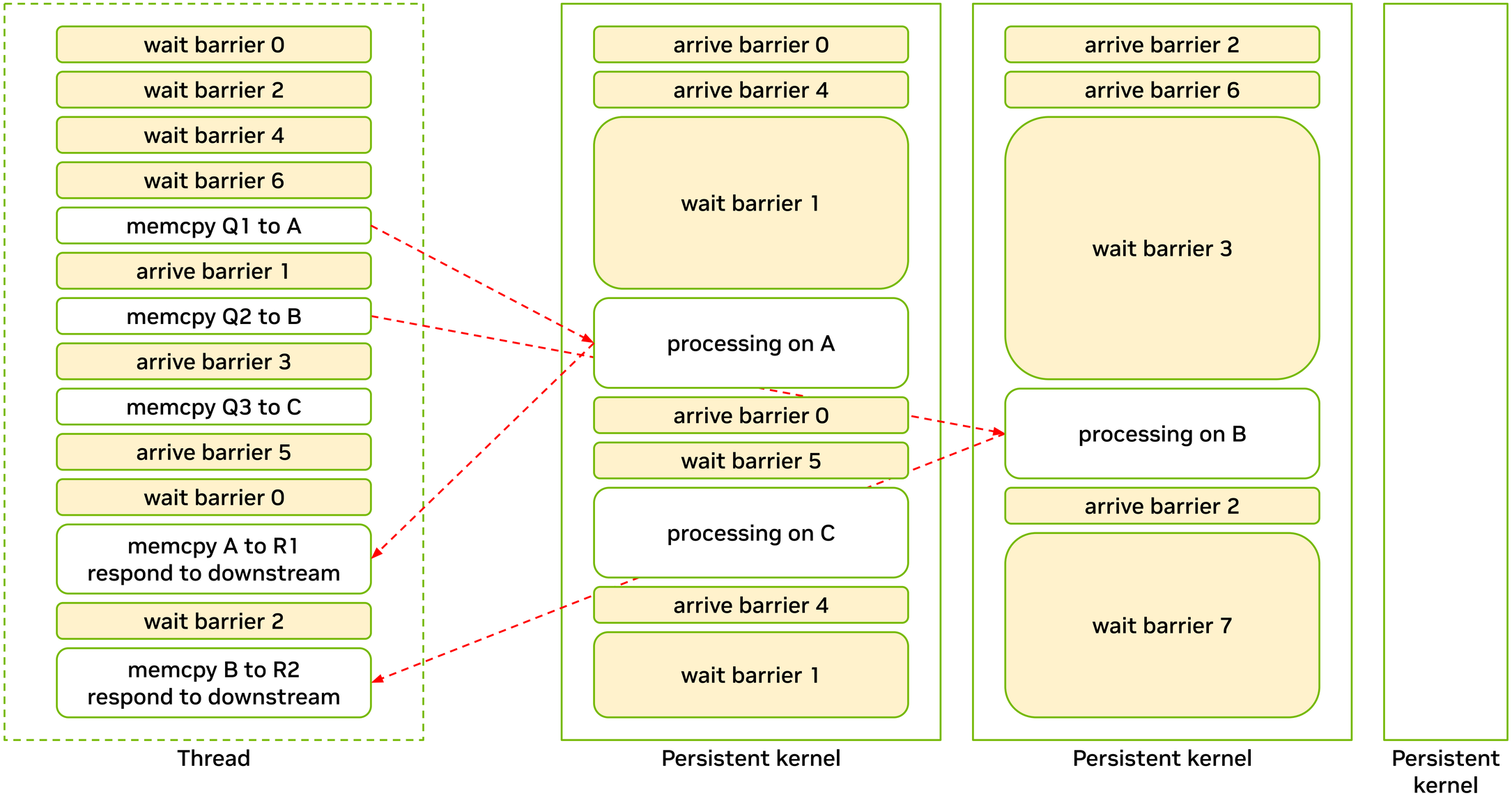
Interpretation





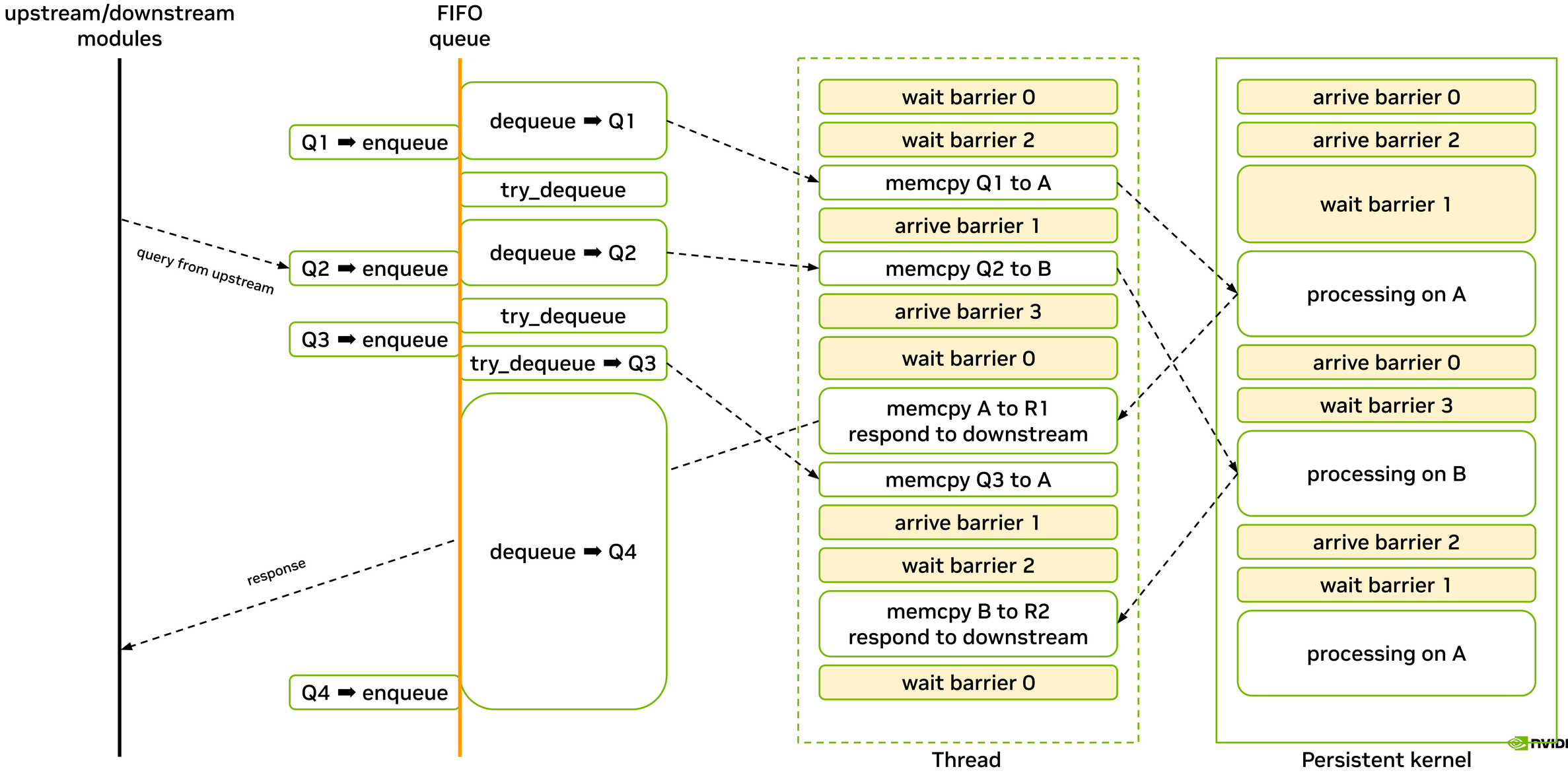
Persistent Kernel Pool

Interpretation



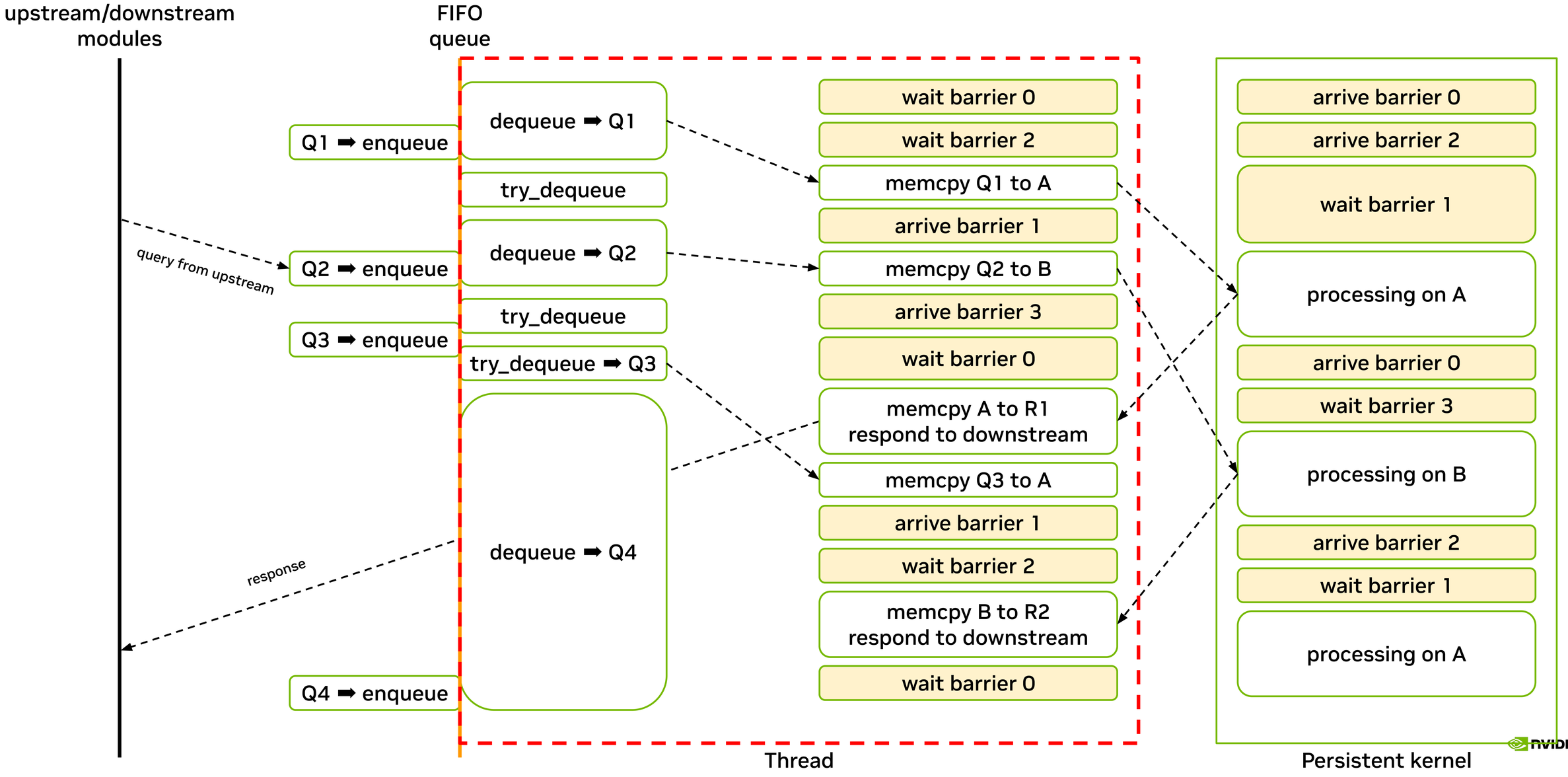
Persistent Kernel Pool

Interpretation



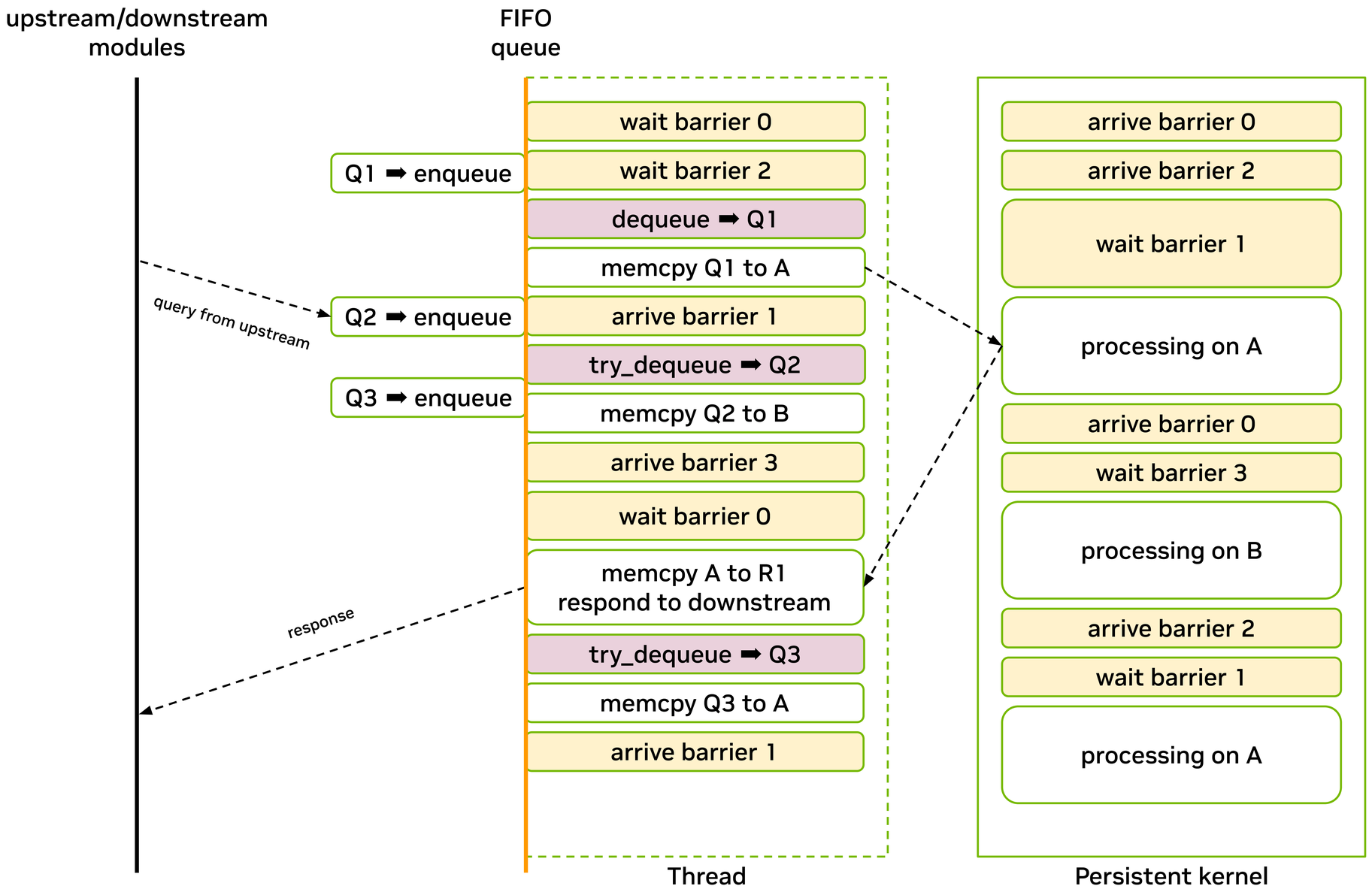
Persistent Kernel Pool

Interpretation



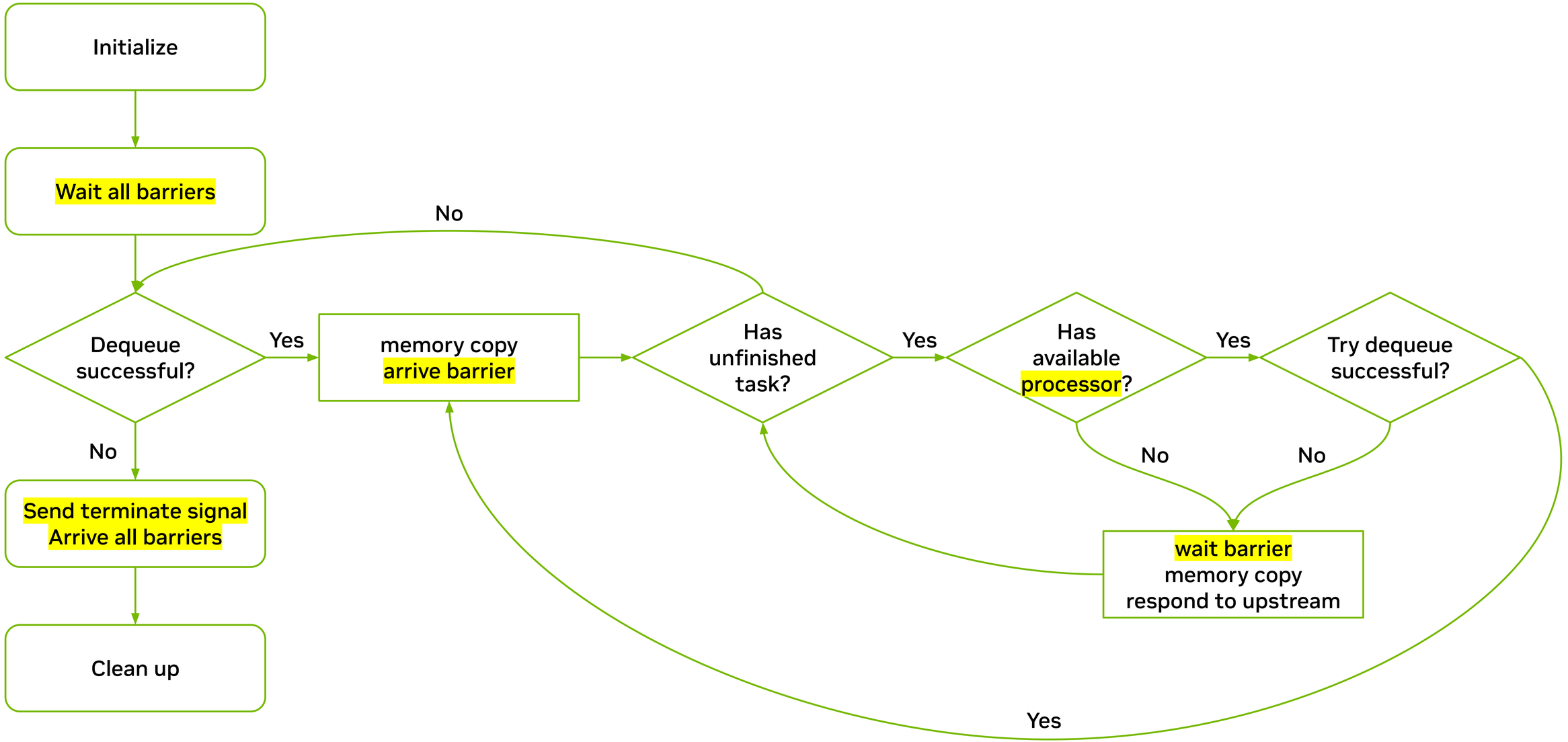
Persistent Kernel Pool

Interpretation



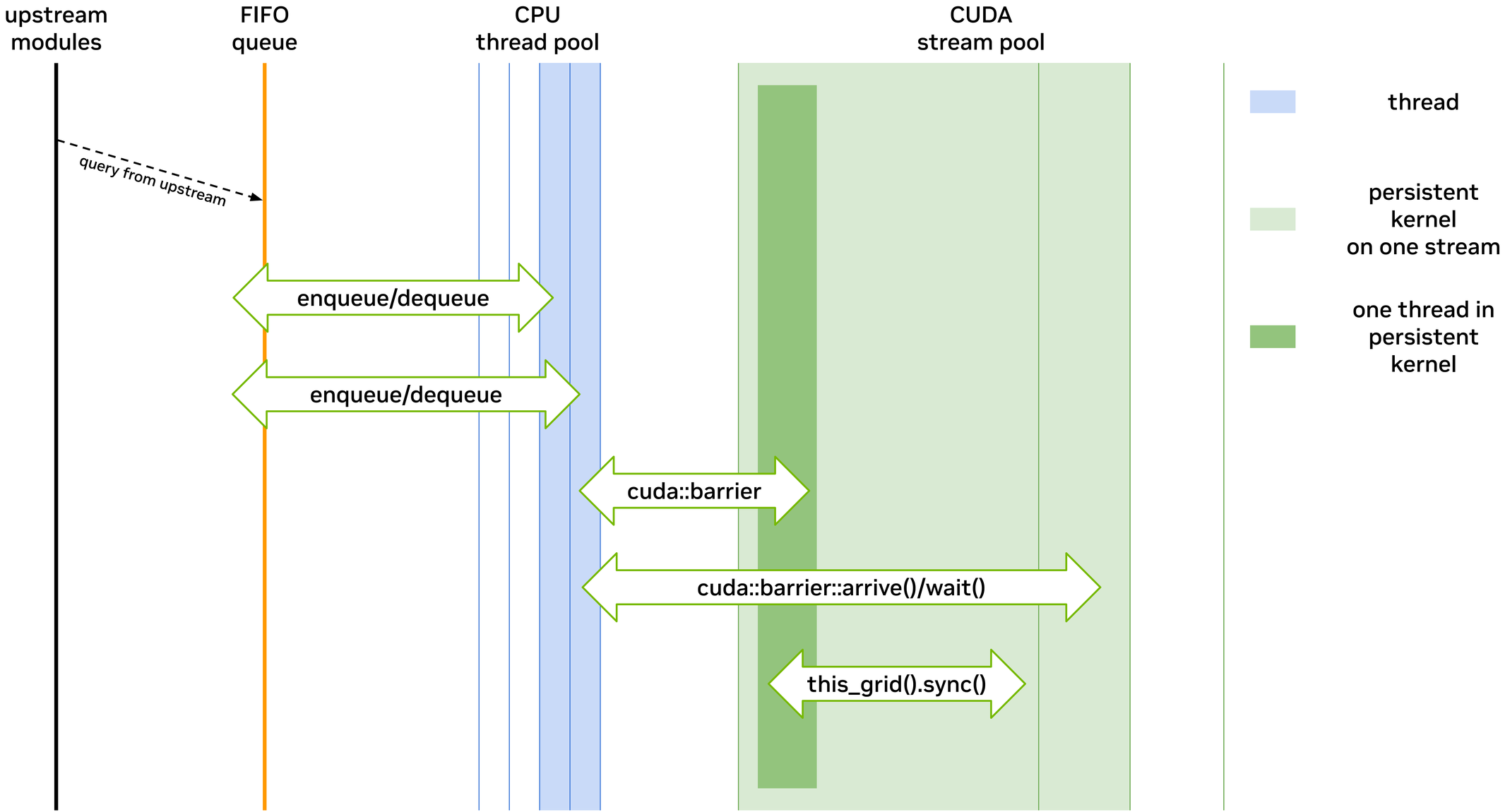
Persistent Kernel Pool

Flowchart



Persistent Kernel Pool

Workflow



Persistent Kernel Pool

Interpretation

Let's compute the SM utilization for a persistent kernel.

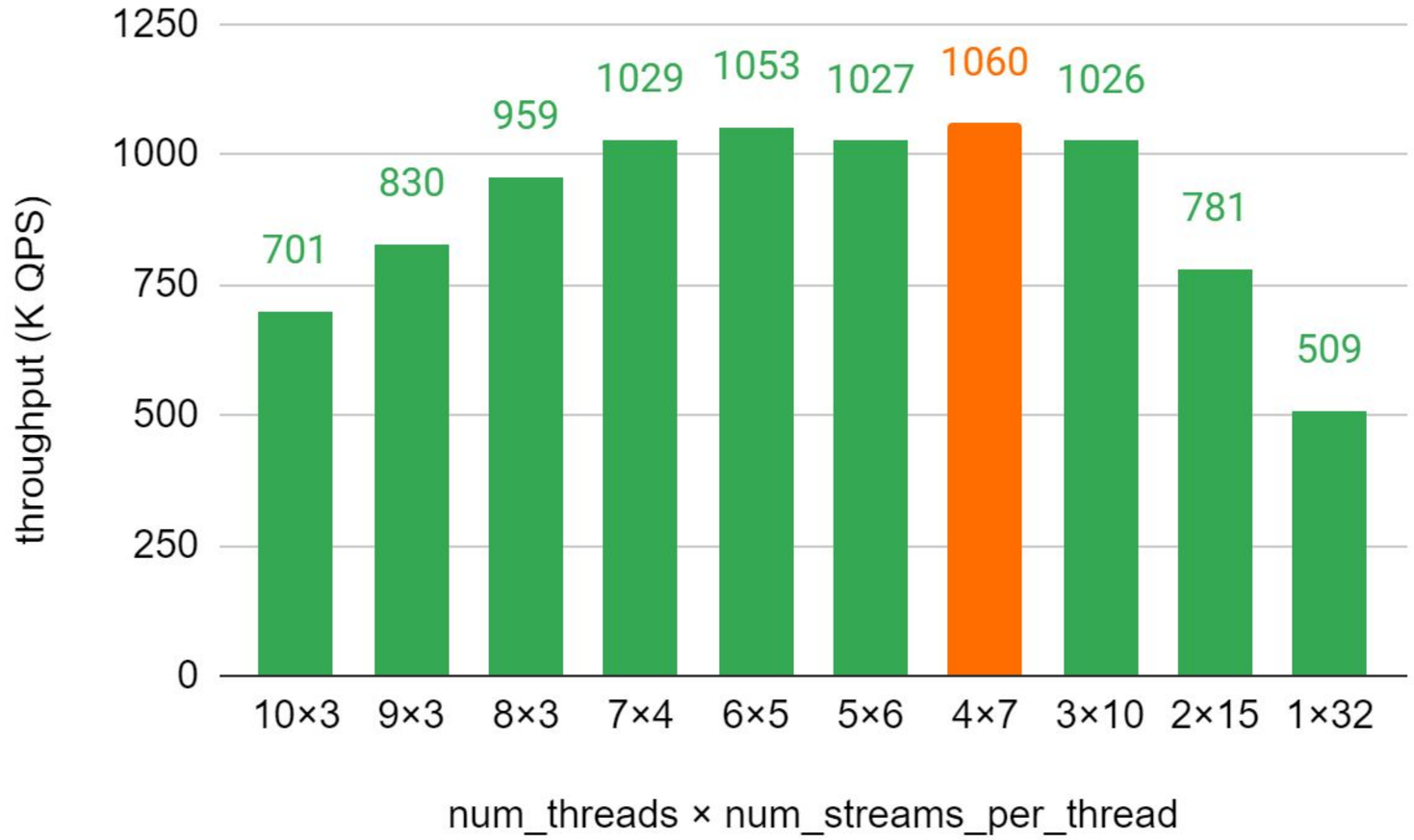
The compiler output and occupancy calculator in Nsight Compute show that the maximum number of allocatable blocks per SM is **7**.

L40 has **142 SMs**, so the total number of allocatable Blocks is 142×7=**994**, and one persistent kernel launches **32** blocks, so the total number of persistent kernels that can reside simultaneously is 994÷32=31.0625, rounded down to **31**, each persistent kernel uses a different stream, so the number of resident streams is also 31.

For different number of threads, the following combinations can achieve the best SM utilization: **1×31**, **2×15**, **3×10**, **4×7**, **5×6**, **6×5**, **7×4**, **8×3**, **9×3**, **10×3**.

Persistent Kernel Pool

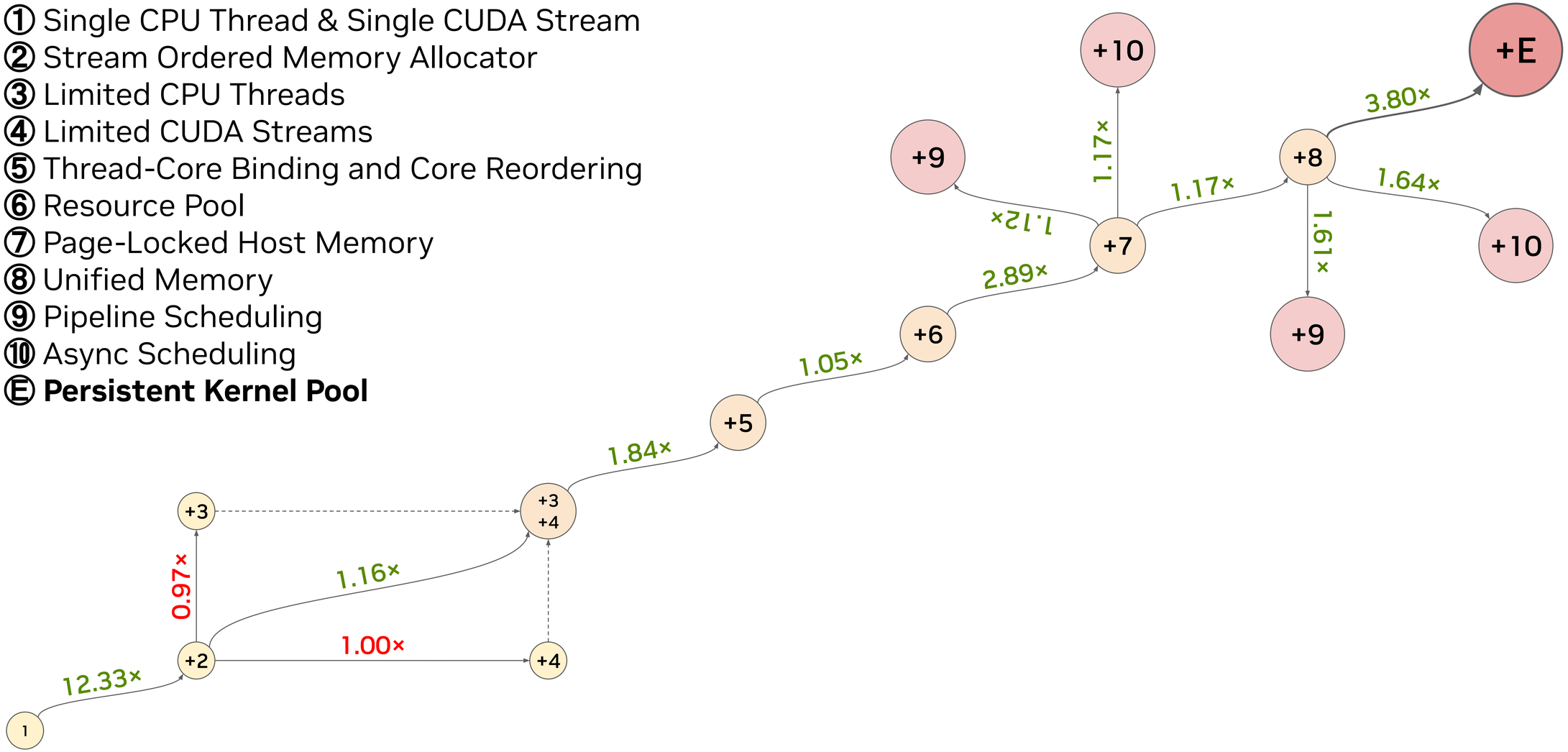
Interpretation



279K 1060K

throughput

① Single CPU Thread & Single CUDA Stream
② Stream Ordered Memory Allocator
③ Limited CPU Threads
④ Limited CUDA Streams
⑤ Thread-Core Binding and Core Reordering
⑥ Resource Pool
⑦ Page-Locked Host Memory
⑧ Unified Memory
⑨ Pipeline Scheduling
⑩ Async Scheduling
Ⓔ **Persistent Kernel Pool**

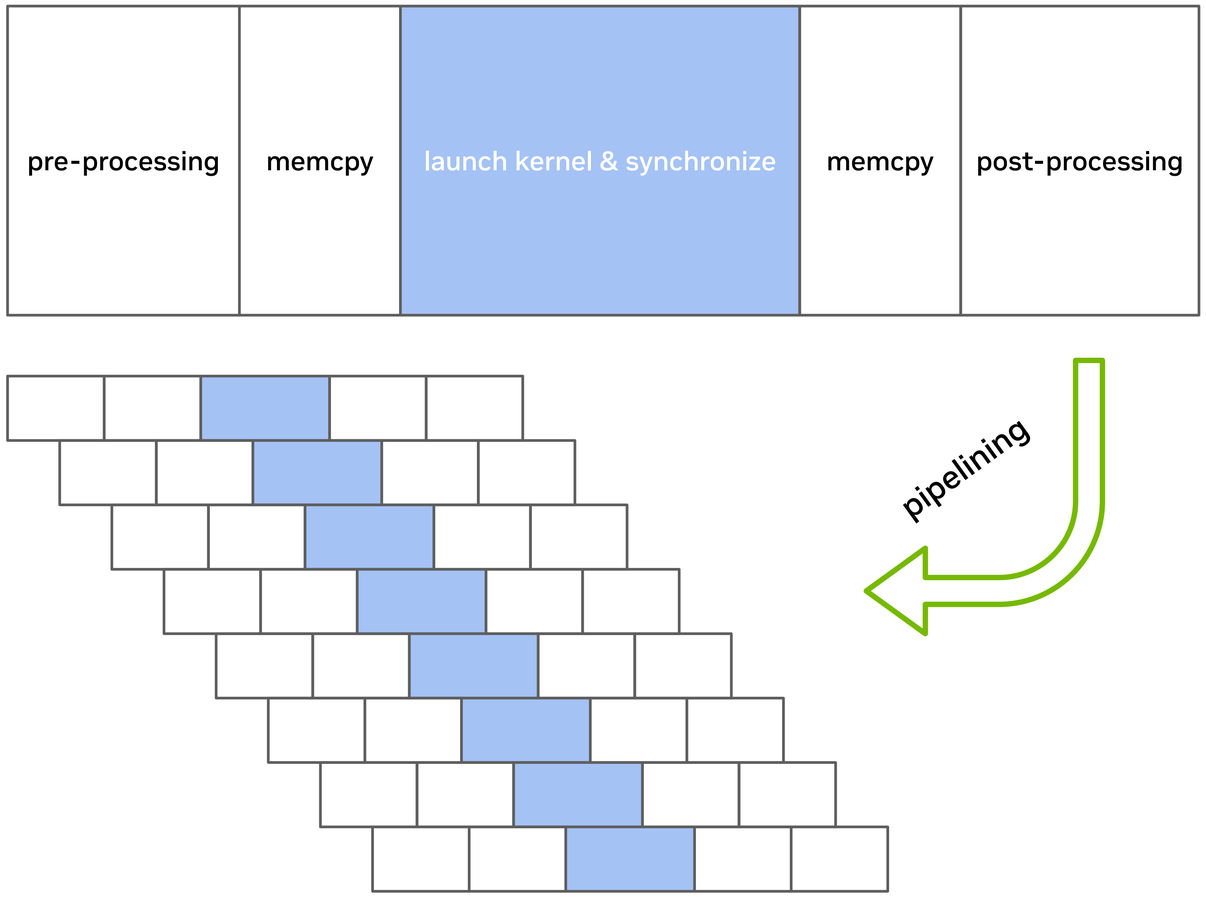


Optimization Techniques Summary

	Average latency (ms)	99%-tile latency (ms)	Throughput (K QPS)	Optimal number of CPU threads	Optimal number of CUDA stream per thread	Equivalent parallelism
Single Thread Single Stream	0.136	0.434	3	1	1	1
Stream Ordered Memory Allocator	0.120	0.487	37			
Limited Threads Limited Streams	0.136	0.473	43	2		2
Thread-core Binding and Core Reordering	0.115	0.494	79	4		4
Resource Pool	0.106	0.472	97			
Page-Locked Host Memory	0.074	0.247	239	8		8
Unified Memory	0.122	0.470	279			
Pipeline Scheduling with Unified Memory	0.118	0.357	448	4	4	16
Async Scheduling with Unified Memory	0.179	0.323	458			
Persistent Kernel Pool	0.249	0.343	1060	4	7	56

Technical Tips

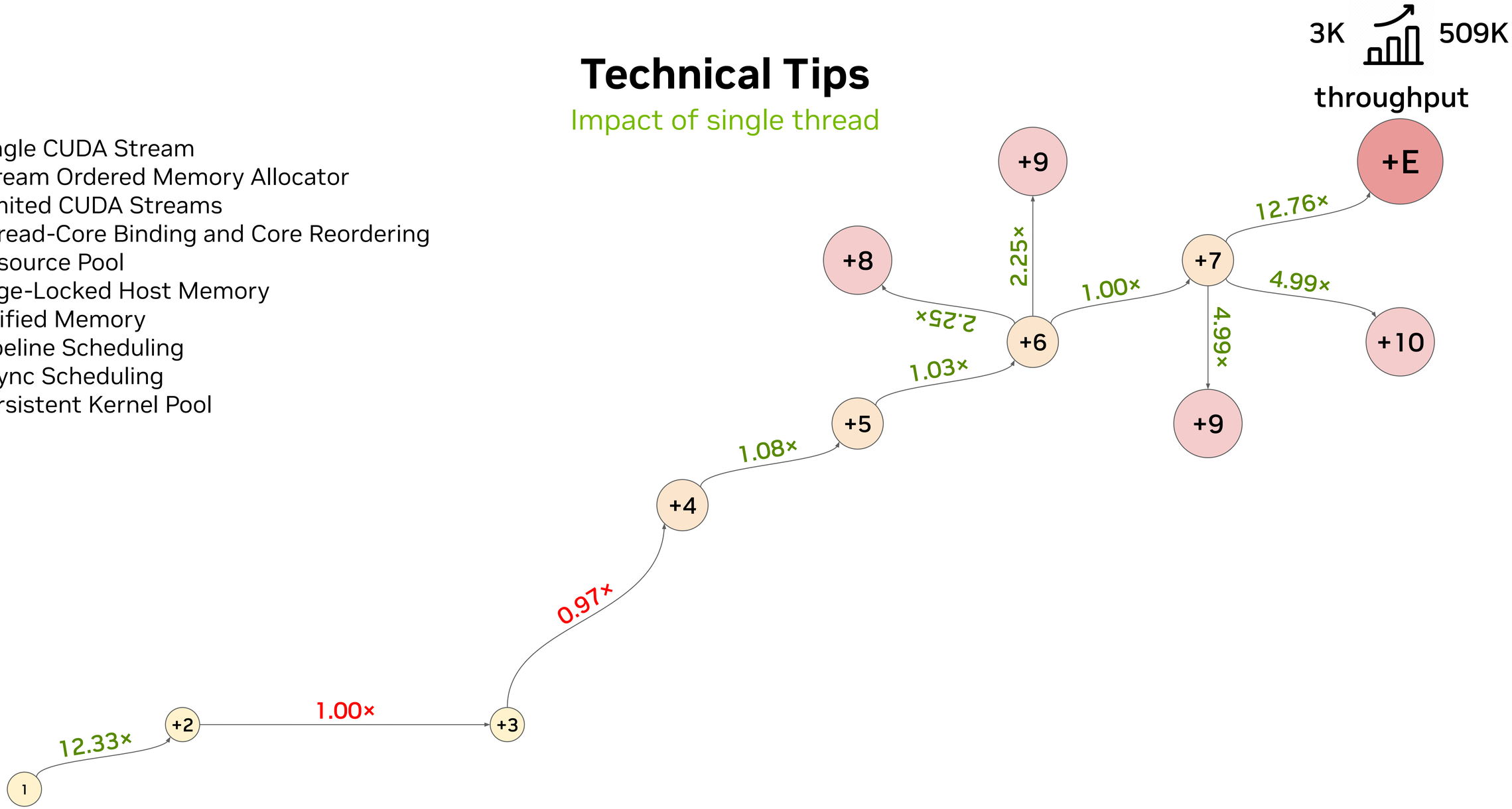
Impact of batch size



Technical Tips

Impact of single thread

① Single CUDA Stream
② Stream Ordered Memory Allocator
③ Limited CUDA Streams
④ Thread-Core Binding and Core Reordering
⑤ Resource Pool
⑥ Page-Locked Host Memory
⑦ Unified Memory
⑧ Pipeline Scheduling
⑨ Async Scheduling
Ⓔ Persistent Kernel Pool



Even if limited to single CPU thread from beginning to end, it still has a very good speedup ratio.

Technical Tips

Impact of single thread

	Average latency (ms)	99%-tile latency (ms)	Throughput (K QPS)	Optimal number of CPU threads	Optimal number of CUDA stream per thread	Equivalent parallelism
Single Thread Single Stream	0.136	0.434	3	1	1	1
Stream Ordered Memory Allocator	0.120	0.487	37			
Limited Threads Limited Streams	0.114	0.469	37			
Thread-core Binding and Core Reordering	0.126	0.491	36			
Resource Pool	0.114	0.485	39			
Page-Locked Host Memory	0.070	0.255	40			
Unified Memory	0.058	0.204	40			
Pipeline Scheduling with Unified Memory	0.180	0.474	199		8	8
Async Scheduling with Unified Memory	0.131	0.452	199			
Persistent Kernel Pool	0.199	0.448	509		31	62

Takeaways

- System level optimization is as important as kernel level optimization.
- GPU does not work alone, and good collaboration between CPU and GPU is also essential for efficiency.
- A GPU friendly architecture is a good start.
- The measurement of throughput should be guaranteed to be within acceptable latency.
- Good memory management is crucial to efficiency.
- Multithreading is a common optimization, but the launch overhead caused by multithreading is also one of the main reasons for low performance.
- Using fewer CPU threads and fewer CUDA API calls can greatly reduce this overhead.
- Leveraging the asynchronous nature of CUDA API can help use fewer CPU threads when maintaining the number of CUDA streams.