Advanced Performance Optimization in CUDA [S62192]

Igor Terentyev*, NVIDIA DevTech Compute

GPU Technology Conference / March 18th, 2024

* With Guillaume Thomas-Collignon & Athena Elafrou

Agenda

- Cooperative Grid Arrays
- Memory Model
- Asynchronous Barriers
- Asynchronous Data Copies
- Compressible Memory
- CUDA Graphs

Nomenclature

CTA (Cooperative Thread Array) == Thread Block

CGA (Cooperative Grid Array) == Thread Block Cluster

Code snippets:

```
namespace cg = cooperative_groups;
```

Cooperative Grid Arrays

Cooperative Grid Arrays

Teaser

Increased SMEM saves GMEM trips in many algorithms...

48 KB
Kepler
64 KB
P100
96 KB
V100
164 KB
A100
224 KB
H100
Cluster size 16
Distributed SMEM
3648 KB
H100

Cooperative Grid Arrays

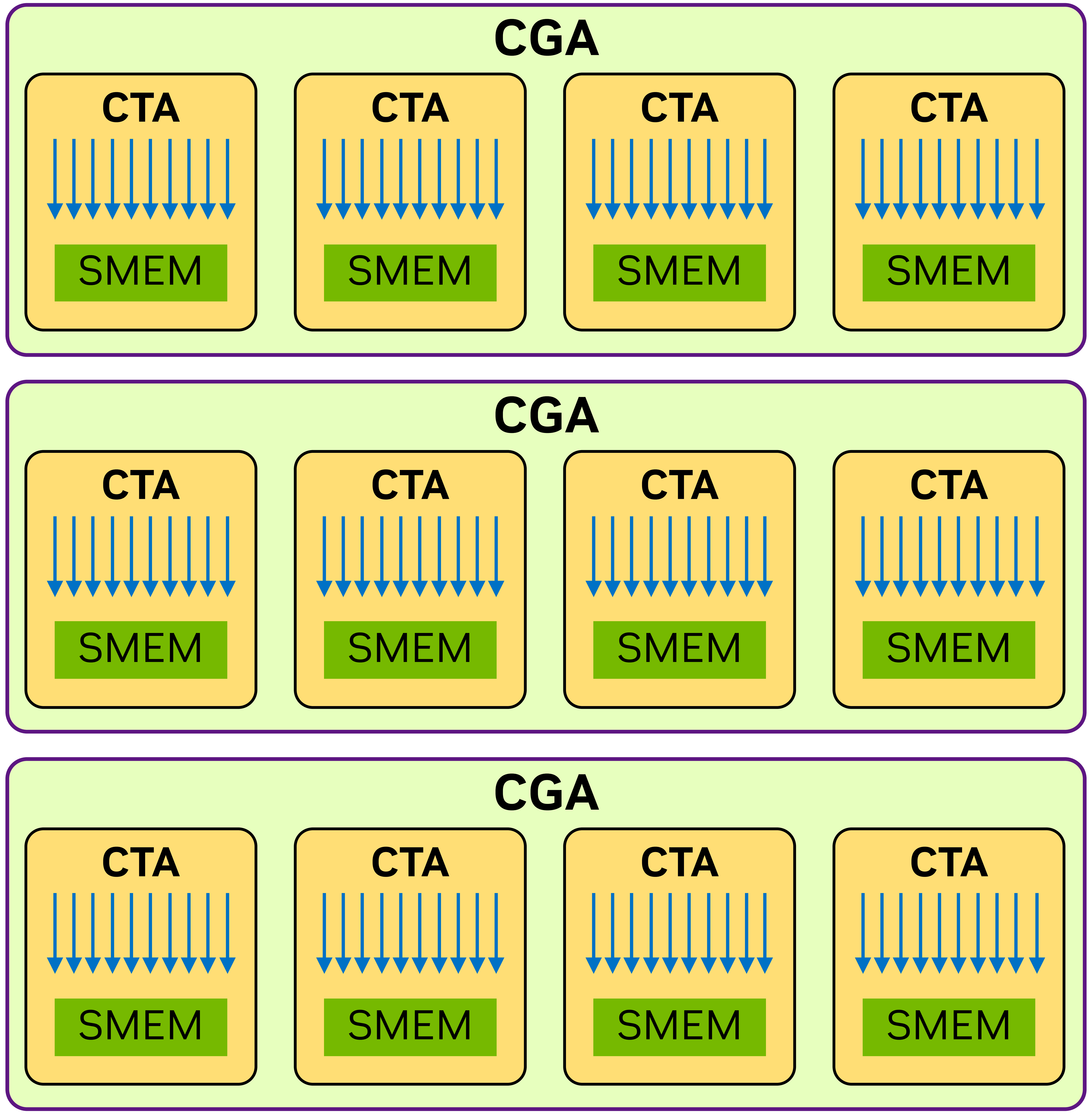
Properties

New level in the hierarchy:

T < CTA < **CGA** < GRID

Clusters:

- Support guaranteed for up to 8 CTAs / CGA
  need to opt-in for more than 8 (max 16 on Hopper)
- CTAs can be logically organized in 3D blocks:
  - linear rank: `cg.this_cluster().block_rank()`
  - 3D rank : `cg.this_cluster().block_index()`
- CTAs in cluster are co-scheduled
- Thread synchronization:
  - CTA → `cg.this_block().sync()`
  - CGA → `cg.this_cluster().sync()`
- Possible resource under-utilization:
  cluster size > 2 may lead to unused SMs



Cooperative Grid Arrays

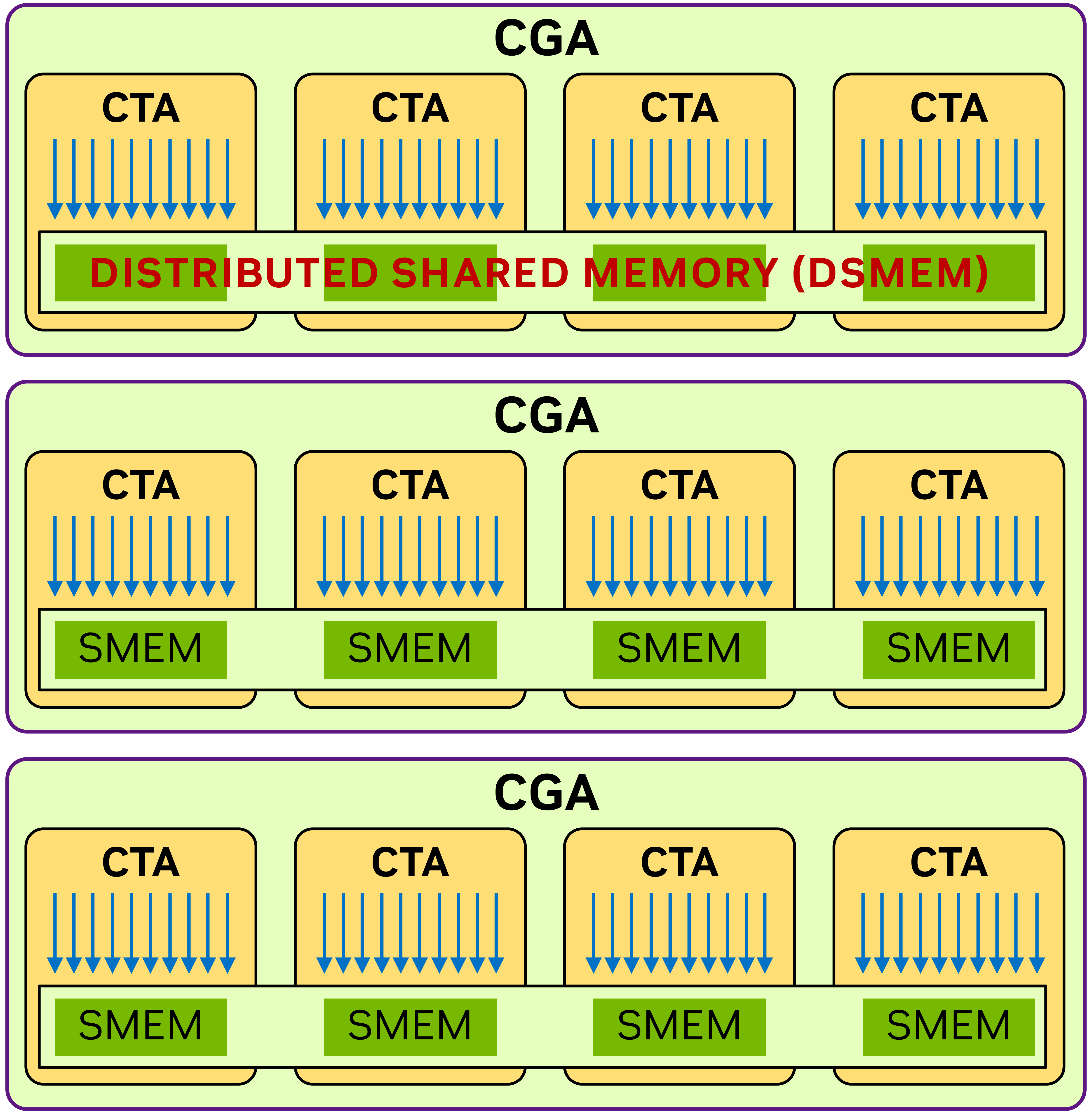
DSMEM properties

Distributed SMEM (DSMEM):

- SMEM within cluster is accessible by all CTAs → DSMEM
  E.g., H100 total DSMEM size ↗ 3.6 MB
- DSMEM allows:
  - Access: LD / ST / ATOM
  - Async copies into remote DSMEM
  - Multicasting
- Note: remote DSMEM is slower than local:
  low latency, but relatively low bandwidth
- Remote DSMEM has to be mapped:

```
__shared__ T smem;
auto dsmem_ptr =
    cg::this_cluster().map_shared_rank(&smem, rank);
```

- **Important:**
  make sure remote CTAs are alive while accessing DSMEM !
  (use `cluster.sync()` accordingly)



Cooperative Grid Arrays

Launching kernels with CGA support

Compile-time CGA:

```
constexpr int X = 2;
constexpr int Y = 2;
constexpr int Z = 2;
__global__ __cluster_dims__(X, Y, Z) void kernel ();
dim3 grid = {...};
assert(grid.x % X == 0 && grid.y % Y == 0 && grid.z % Z == 0); // Launch failure otherwise
kernel<<<grid_size,...>>>();
```

Run-time CGA:

```
__global__ void kernel (); // as usual
cudaLaunchAttribute attribute[1];
attribute[0].id = cudaLaunchAttributeClusterDimension;
attribute[0].val.clusterDim.x = X;
attribute[0].val.clusterDim.y = Y;
attribute[0].val.clusterDim.z = Z;
assert(grid.x % attribute[0].val.clusterDim.x == 0 && ...); // Launch failure otherwise
cudaLaunchConfig_t config = {0};
config.attrs = attribute;
config.numAttrs = 1;
cudaLaunchKernelEx(&config, kernel, ...);
```

Memory Model

Memory Model

Why memory model?

Why do we need Memory (Consistency) Model?

Memory operations behave “strangely” in a multithreaded world…

Memory Model

Weak memory model

Thread 0:

```
  data = 42;
  flag = 1;
```

Thread 1:

```
  while (!flag) {}
  assert(data == 42);   // not guaranteed !!
```

CUDA uses **Weak Memory Model**:

- Writes by one thread may not be observed in the same order by other threads

Memory Model

Weak memory model

```
Thread 0:                      Thread 1:
 data = 42;                       while (!flag) {}
 flag = 1;                        assert(data == 42);   //  not guaranteed !!
```

Moreover, Undefined Behavior (UB)!

CUDA uses **Weak Memory Model**:

- Writes by one thread may not be observed in the same order by other threads
- Concurrent access of same memory with at least one ‘`write`’ is a **CONFLICT (UNDEFINED BEHAIVIOR)**
- Compiler assumes no-conflict code, **it’s a programmer’s responsibility to synchronize**

Memory Model

Weak memory model

Thread 0:   **Thread 1:**

```
data = 42;                       while (!flag) {}
flag = 1;                        assert(data == 42);  // not guaranteed !!
```

Moreover, Undefined Behavior (UB)!

CUDA uses **Weak Memory Model**:

- Writes by one thread may not be observed in the same order by other threads
- Concurrent access of same memory with at least one ‘write’ is a **CONFLICT (UNDEFINED BEHAIVIOR)**
- Compiler assumes no-conflict code, **it’s a programmer’s responsibility to synchronize**

Synchronization between threads requires:

1. **Strong ordering: “happens before” relation**
2. **Reaching correct Point of Coherency**

Memory Model

Weak memory model

Thread 0: **Thread 1:**

```
data = 42;                while (!flag) {}
flag = 1;                 assert(data == 42);   // not guaranteed !!
```

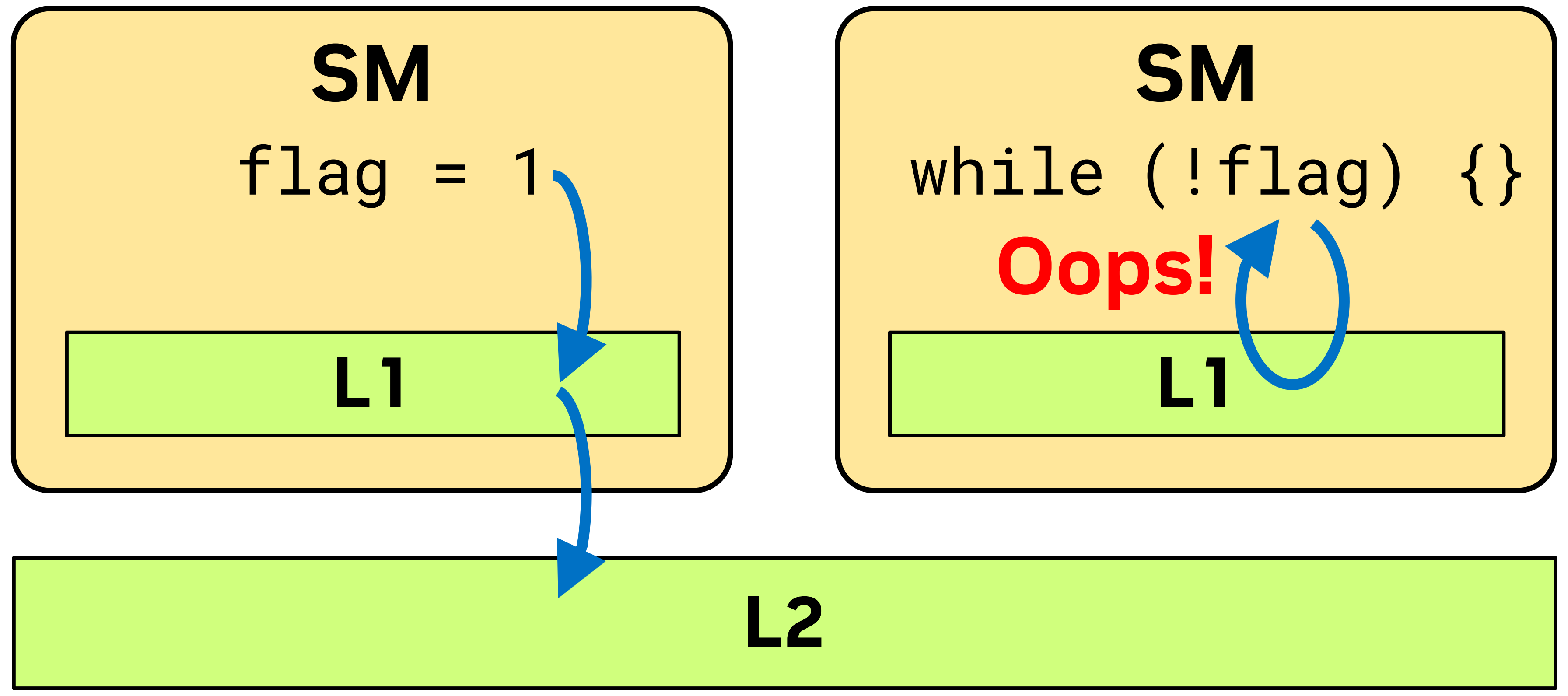
Moreover, Undefined Behavior (UB)!

CUDA uses **Weak Memory Model**:

- Writes by one thread may not be observed in the same order by other threads
- Concurrent access of same memory with at least one `‘write’` is a **CONFLICT (UNDEFINED BEHAIVIOR)**
- Compiler assumes no-conflict code, **it’s a programmer’s responsibility to synchronize**

Synchronization between threads requires:

1. **Strong ordering: “happens before” relation**
2. **Reaching correct Point of Coherency**



Memory Model

Common synchronization using shared memory

A very common communication pattern:

```
__shared__ T buf[1024];

buf[ind1] = value1;        // Threads writing to shared memory

__syncthreads();           // Barrier

value2 = buf[ind2];        // Threads reading values written by other threads
```

Memory Model

Common synchronization using shared memory

A very common communication pattern:

```
__shared__ T buf[1024];

buf[ind1] = value1;       // Threads writing to shared memory

__syncthreads();          // Barrier + makes all previous writes visible to all threads in the block

value2 = buf[ind2];       // Threads reading values written by other threads
```

✓ 1. Strong ordering: “happens before” relation
✓ 2. Reaching correct Point of Coherency

Memory Model

Common synchronization using global memory

A less common communication pattern:

```
__global__ void kernel (T* buf)
{

  buf[ind1] = value1;       // Threads writing to global memory


  __syncthreads();          // Barrier + makes all previous writes visible to all threads in block
                                                                              Including global memory !

  value2 = buf[ind2];       // Threads reading values written by other threads in the same block

}
```

✓ 1. Strong ordering: “happens before” relation
✓ 2. Reaching correct Point of Coherency

Memory Model

Common synchronization using global memory

A less common communication pattern:

```
__global__ void kernel (T* buf)
{

  buf[ind1] = value1;      // Threads writing to global memory


  cg::this_grid().sync();  // Barrier + makes all previous writes visible to all threads in ~~block~~ grid
                                                                   Including global memory !

  value2 = buf[ind2];      // Threads reading values written by other threads ~~in the same block!~~

}
```

requires cooperative launch

✓ 1. Strong ordering: “happens before” relation

✓ 2. Reaching correct Point of Coherency

Memory Model

Common synchronization using global memory

A less common communication pattern:

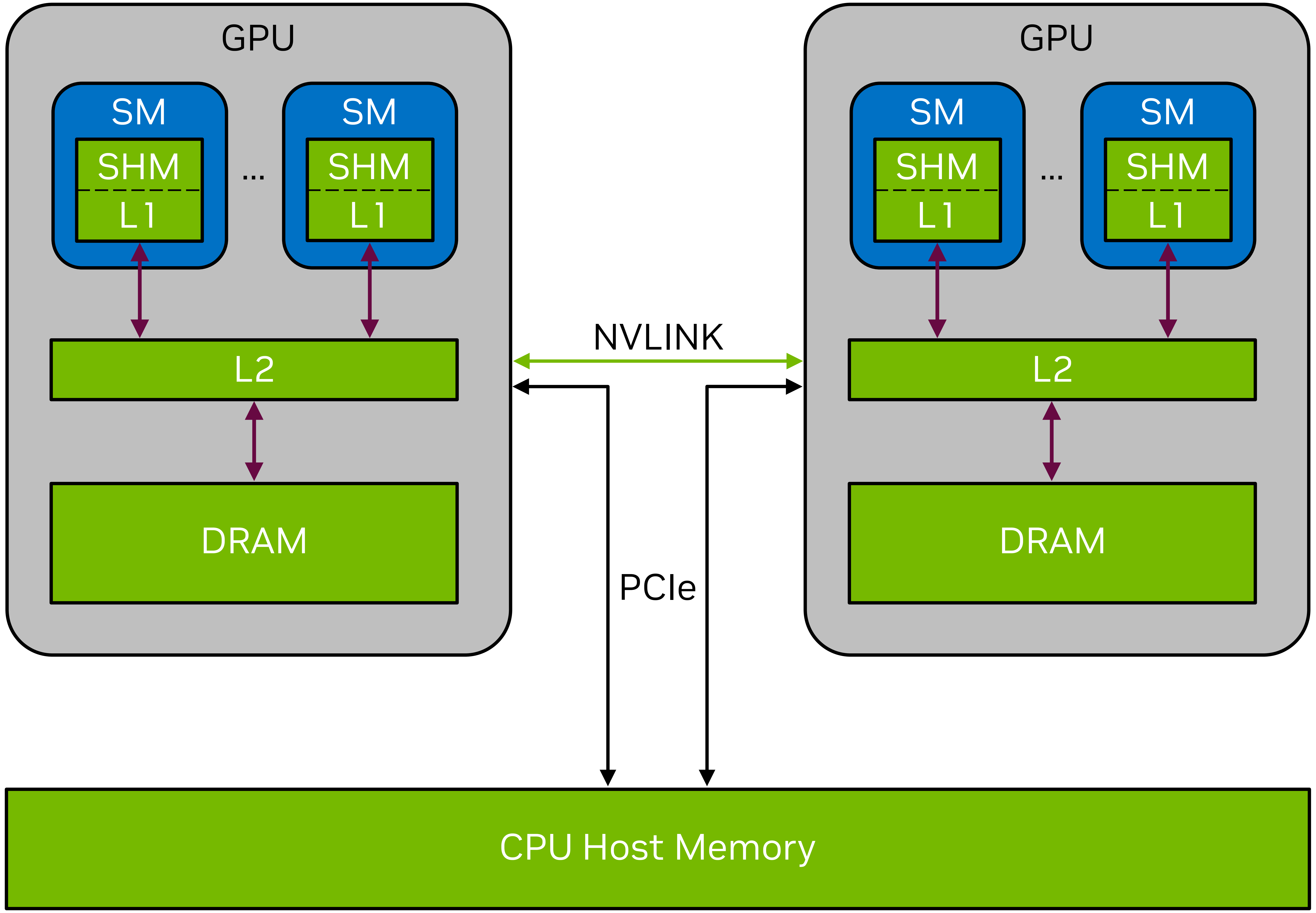
```
__global__ void kernel (T* buf)
{

  buf[ind1] = value1;       // Threads writing to global memory


  cg::this_grid().sync();   // Barrier + makes all previous writes visible to all threads in ~~block~~ grid
                                                                   Including global memory !

  value2 = buf[ind2];       // Threads reading values written by other threads ~~in the same block!~~

}
```

requires cooperative launch

What’s really happening here?

Memory Model

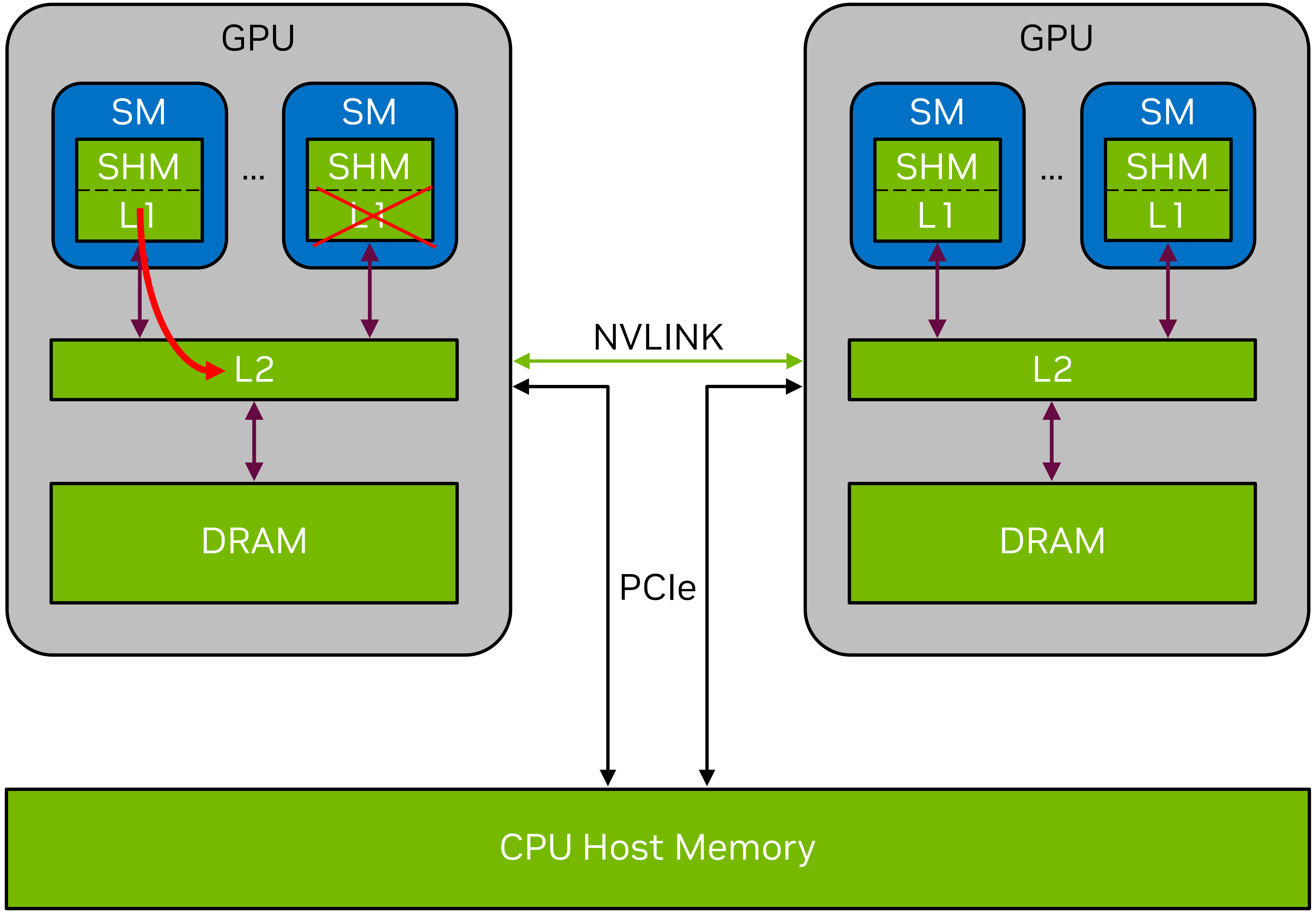
GPU memory hierarchy



Point of Coherency
depends on the level at which
threads are communicating

Memory Model

GPU memory hierarchy



Point of Coherency
depends on the level at which threads are communicating

Memory Model

Common synchronization using global memory

A less common communication pattern:

```
__global__ void kernel (T* buf)
{

  buf[ind1] = value1;       // Threads writing to global memory


  cg::this_grid().sync();   // Barrier + makes all previous writes visible to all threads in block
                                                                         Including global memory !

  value2 = buf[ind2];       // Threads reading values written by other threads ~~in the same block!~~

}
```

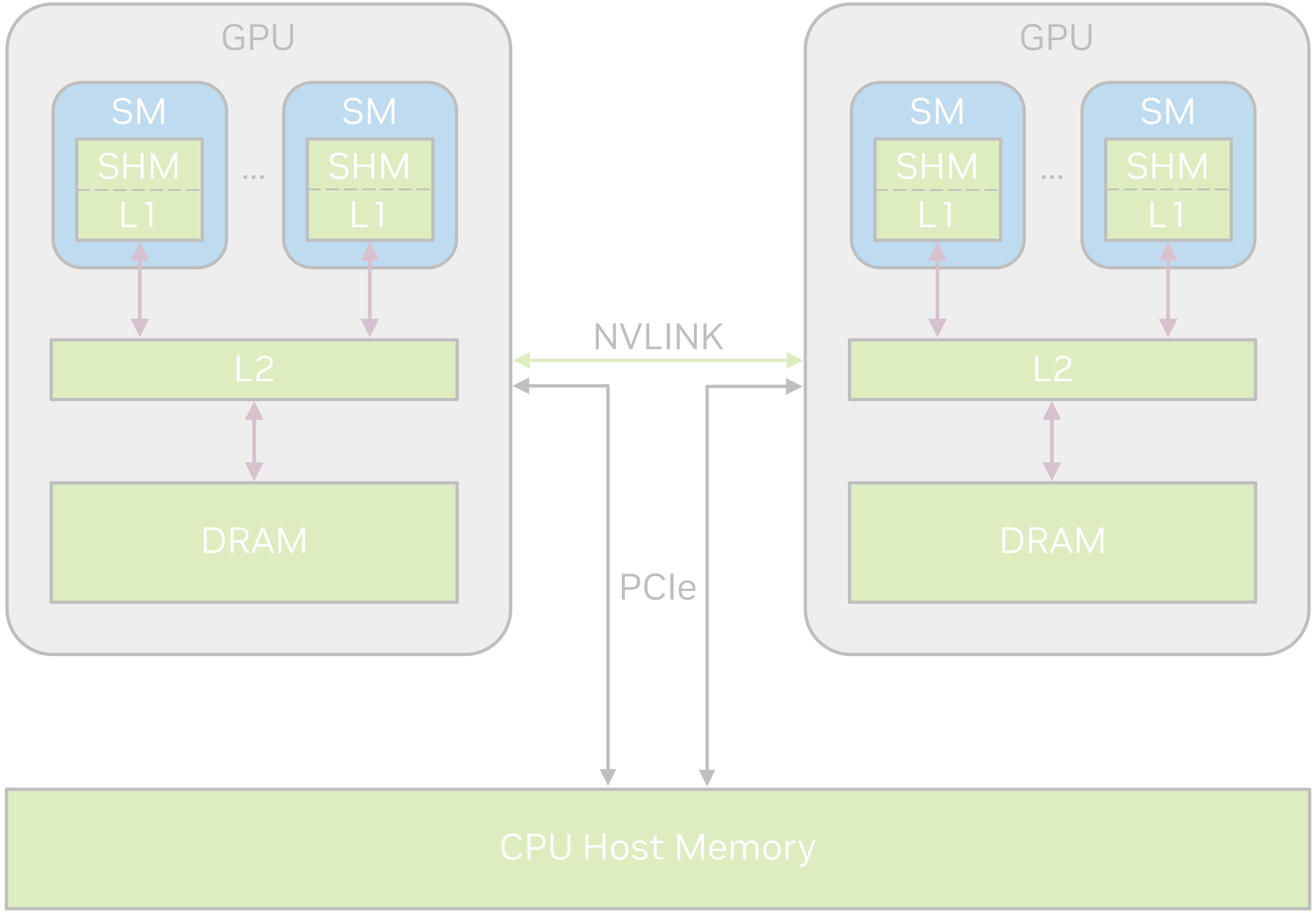
requires cooperative launch

What’s really happening here?

Memory fence + barrier (device level) + L1 cache invalidate

Memory Model

GPU memory hierarchy



Point of Coherency
depends on the level at which threads are communicating:

Self (thread)
Block
Cluster
Device
System

All you really need to know about POC

Memory Model

Fine-grained communication

Producer/consumer, different threads running on different GPUs

```
__global__ void producer (
    T* data, int* flag)


*data = value;          // Write data



*flag = 1;              // Then flag
```

```
__global__ void consumer (
    T* data, int* flag)


while (*flag == 0)      // Wait for flag
{}


value = *data;          // Then read data
```

Does this work? **NO**

❌ 1. Strong ordering: “happens before” relation

❌ 2. Reaching correct Point of Coherency

Memory Model

Fine-grained communication

Producer/consumer, different threads running on different GPUs

```
__global__ void producer (
    T* data, int* flag)


*data = value;          // Write data

__threadfence_system(); // Memory fence

*flag = 1;              // Then flag
```

```
__global__ void consumer (
    T* data, int* flag)


while (*flag == 0)      // Wait for flag
{}
__threadfence_system(); // Memory fence

value = *data;          // Then read data
```

Does this work? **NO**

✓ 1. Strong ordering: “happens before” relation

✗ 2. Reaching correct Point of Coherency

Memory Model

Fine-grained communication

Producer/consumer, different threads running on different GPUs

```
__global__ void producer (
    T* data, int* volatile flag)


*data = value;          // Write data

__threadfence_system(); // Memory fence

*flag = 1;              // Then flag
```

```
__global__ void consumer (
    T* data, int* volatile flag)


while (*flag == 0)      // Wait for flag
{}
__threadfence_system(); // Memory fence

value = *data;          // Then read data
```

Does this work? **YES (in practice)**

✓ 1. Strong ordering: “happens before” relation

✓ 2. Reaching correct Point of Coherency **– forced by `volatile`**

Is it right? **NO**

- **`volatile` does not imply atomicity**
  Concurrent access of same memory with at least one ‘write’ is a **CONFLICT (UNDEFINED BEHAIVIOR)** – remember?!

Memory Model

Fine-grained communication

Producer/consumer, different threads running on different GPUs

```
__global__ void producer (
    T* data, int* flag)


*data = value;          // Write data

__threadfence_system(); // Memory fence

atomicExch_system((int*)flag, 1); // Then flag
```

```
__global__ void consumer (
    T* data, int* flag)


while (atomicAdd_system(flag, 0) == 0) // Wait for flag
{}
__threadfence_system(); // Memory fence

value = *data;          // Then read data
```

Does this work? **YES**

✓ 1. Strong ordering: “happens before” relation

✓ 2. Reaching correct Point of Coherency

Is it right? **YES**

But…

- Not the most elegant/efficient
- Cannot communicate with a CPU – no compatible atomic

Memory Model

Fine-grained communication

Producer/consumer, different threads running on different GPUs

```
__global__ void producer (
    T* data,
    cuda::atomic<int, thread_scope_system>* flag)

*data = value;          // Write data



*flag = 1;              // Then flag
flag->notify_all();
```

```
__global__ void consumer (
    T* data,
    cuda::atomic<int, thread_scope_system>* flag)

flag->wait(0);          // Wait for flag



value = *data;          // Then read data
```

C++ atomics have a built-in memory fence

Does this work? **YES**

✓ 1. Strong ordering: “happens before” relation

✓ 2. Reaching correct Point of Coherency

Is it right? **YES, elegant, C++ atomics can communicate with CPU**

But…

- Still not the most efficient

Memory Model

Memory order

“Memory order specifies how memory accesses, including regular (non-atomic) accesses, are to be ordered around an atomic operation”

Default behavior of `std::atomic` is **sequentially-consistent**:

- Enforces strict ordering not only between atomics but between all memory ops
  Think about it as **“slow for HW to do”**

Memory Model

Fine-grained communication

Producer/consumer, different threads running on different GPUs

```
__global__ void producer (
    T* data,
    cuda::atomic<int, thread_scope_system>* flag)

*data = value;          // Write data



*flag = 1;              // Then flag
flag->notify_all();
```

```
__global__ void consumer (
    T* data,
    cuda::atomic<int, thread_scope_system>* flag)

flag->wait(0);          // Wait for flag



value = *data;          // Then read data
```

Only need preceding writes not reordered after flag-write

Memory Model

Fine-grained communication

Producer/consumer, different threads running on different GPUs

```
__global__ void producer (
    T* data,
    cuda::atomic<int, thread_scope_system>* flag)

*data = value;          // Write data



*flag = 1;              // Then flag
flag->notify_all();
```

Only need preceding writes not reordered after flag-write

```
__global__ void consumer (
    T* data,
    cuda::atomic<int, thread_scope_system>* flag)

flag->wait(0);          // Wait for flag



value = *data;          // Then read data
```

Only need subsequent reads not reordered before flag-read

Memory Model

Acquire / release

“Memory order specifies how memory accesses, including regular (non-atomic) accesses, are to be ordered around an atomic operation”

Default behavior of `std::atomic` is **sequentially consistent**:

- Enforces strict ordering not only between atomics but between all memory ops
  Think about it as **“slow for HW to do”**

Fine-grained ordering around atomic ops:

- `memory_model::acquire`
  No reads or writes in the current thread can be moved before the acquire
  Makes writes from other threads (that use same atomic) visible to the current thread
- `memory_model::release`
  No reads or writes in the current thread can be moved after the release
  Makes writes from the current thread visible to the other threads (that use same atomic)
- Other orders

Memory Model

Fine-grained communication

Producer/consumer, different threads running on different GPUs

```
__global__ void producer (
    T* data,
    cuda::atomic<int, thread_scope_system>* flag)

*data = value;          // Write data




flag->write(1, memory_order_release); // Then flag
flag->notify_all();
```

```
__global__ void consumer (
    T* data,
    cuda::atomic<int, thread_scope_system>* flag)

flag->wait(0, memory_order_acquire); // Wait for flag




value = *data;          // Then read data
```

CORRECT, ELEGANT, and EFFICIENT !!!

Memory Model

Cumulativity

DEVICE 1

Thread 1

```
std::atomic<int, thread_scope_device>* flag_dev;
*data = 42;
flag_dev->write(1, memory_order_release);
```

Thread 2

```
std::atomic<int, thread_scope_system>* flag_sys;
flag_dev->wait(0, memory_order_acquire);
assert(*data == 42);
flag_sys->write(1, memory_order_release);
```

DEVICE 2

Thread 3

```
flag_sys->wait(0, memory_order_acquire);
assert(*data == 42);
```

Asynchronous Barriers

Asynchronous Barriers

Barrier intro

CUDA barriers provide (within scope!):

- Synchronization points (including async mem transactions)
- Memory ordering

block till all threads arrived
writes **before** `__syncthreads`
visible **after** `__syncthreads`

write SMEM
`__syncthreads()`
read SMEM

write SMEM
`barrier.arrive()`

mark arrived (non-blocking)
writes **before** `arrive`
visible **after** `wait`

Independent ops (e.g., on other MEM)

`barrier.wait()`

block till all threads marked arrived
writes **before** `arrive`
visible **after** `wait`

read SMEM

Asynchronous Barriers

Barrier details

Barrier

Tracked quantities:

- expected_arrival_count (>=0)
- transaction_count (>=0) **– optional**

State:

- arrival_count
- phase (0/1)

Methods:

- arrive
- wait

Initialization:

- expected_arrival_count := user-provided value
- arrival_count := expected_arrival_count
- phase := 0

- transaction_count – incremented by a user-provided value; **(Hopper++)** decremented (by HW) on completion of memory transactions (allows to synchronize async copies)

Asynchronous Barriers

Barrier details

Barrier

Tracked quantities:

- expected_arrival_count (>=0)
- transaction_count (>=0) **– optional**

State:

- arrival_count
- phase (0/1)

Methods:

- arrive
- wait

Arrive (single atomic non-blocking operation)**:**

```
1. arrival_count -= 1 (or -= n)
2. IF arrival_count == 0
       phase ^= 1  WHEN TRANSACTION COUNT REACHES 0
       arrival_count := expected_arrival_count (barrier reset)
```

Wait (two ways to wait for phase flip):

- Token-based. Typical code:

```
auto token = bar.arrive();
bar.wait(std::move(token)); // more expensive than explicit tracking
```

- Explicitly tracking the phase. Typical code:

```
while (!mbarrier_try_wait_parity(bar, curr_phase, 1000)) {}
curr_phase ^= 1; // need to explicitly keep track of phase
```

Initialization:

- expected_arrival_count := user-provided value
- arrival_count := expected_arrival_count
- phase := 0

- transaction_count – incremented by a user-provided value;
  (Hopper++) decremented (by HW) on completion of memory transactions
  (allows to synchronize async copies)

Asynchronous Barriers

Summary

HW-accelerated on Hopper

Scope	Synchronizes	Typical Memory Location	Initialization	Arrive	Wait
Block	CTA	Local SMEM	Single-thread initialization (followed by CTA sync)	ALLOWED	ALLOWED
Cluster	CGA	Local SMEM (owner CTA)	Single-thread initialization (followed by CGA sync)	ALLOWED	ALLOWED
		Remote DSMEM (after mapping)	NOT ALLOWED	ALLOWED	NOT ALLOWED
Device	GPU	GMEM	Single-thread initialization (followed by device sync#)	ALLOWED	ALLOWED
System	NODE	Accessible by all actors: • GMEM* • Uniform (managed)	Single-thread initialization	ALLOWED*	ALLOWED*

# – cooperative launch required

* – to access from non-owning GPU, peer access must be enabled

Asynchronous Barriers

Barrier location

Producer:

```
Loop {
    Produce data for Consumer
    Arrive at Data barrier
    (notifies Consumer that data is ready)
    Wait at Copy barrier
    (waits for next iteration notification)
}
```

Consumer:

```
Loop {
    Wait at Data barrier
    (waits for data ready notification)
    Use produced data
    Arrive at Copy barrier
    (notifies Consumer that ready for next iteration)
}
```

GPU-GPU Timings

		Data barrier location	
		Producer	Consumer
Copy barrier location	Consumer	0.035	0.065
	Producer	0.065	0.032

**Recommended barrier location:
where waiting**

CPU-GPU Timings

		Data barrier location	
		Producer	Consumer
Copy barrier location	Consumer	0.27	0.27
	Producer	3.39	3.39

**Recommended barrier location:
local to most threads (GPU)**

Tested on 4-way Grace-Hopper

Asynchronous Data Copies

Asynchronous Data Copies

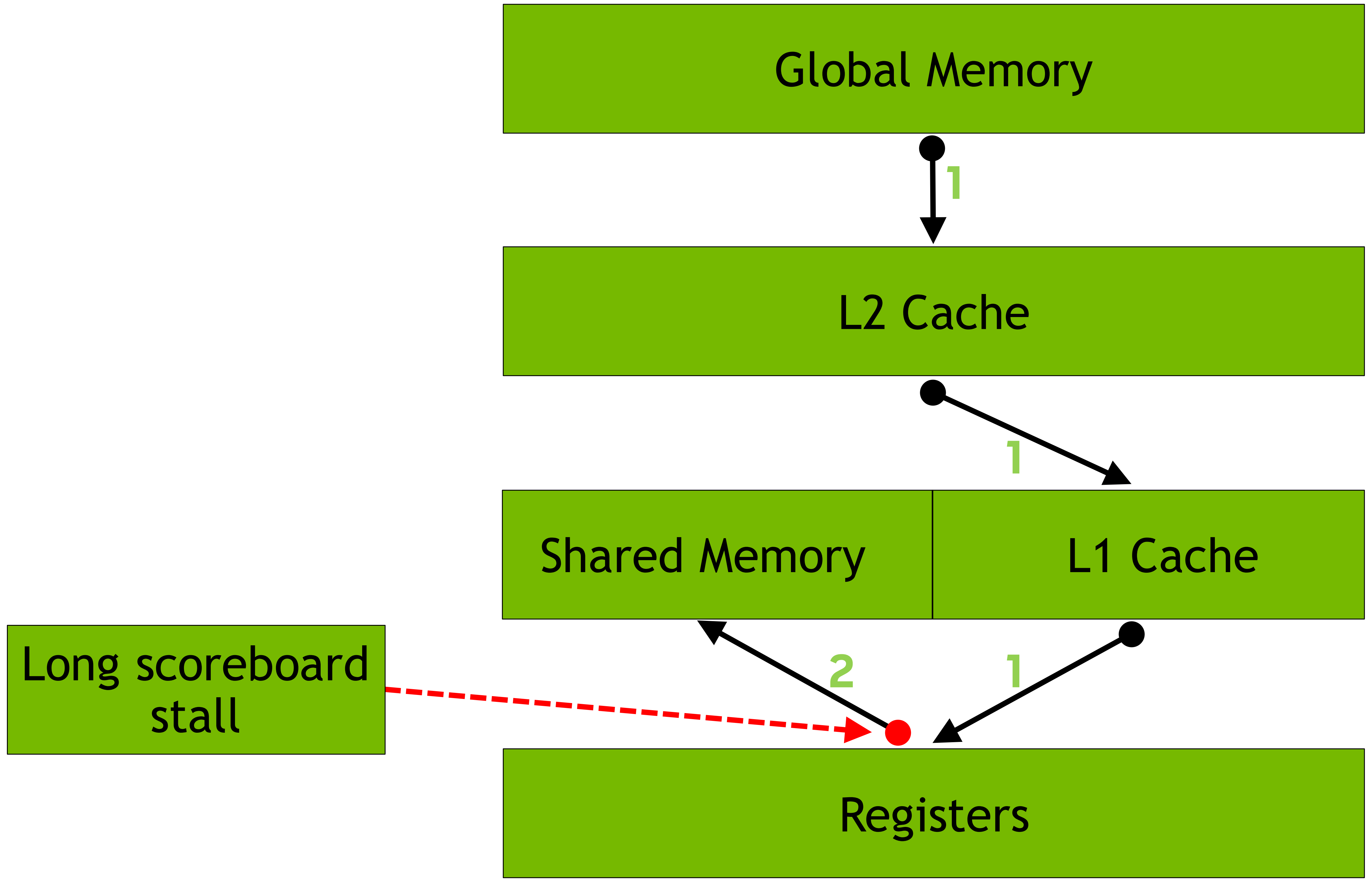
Synchronous copies dissected

```
T* gmem;
__shared__ T smem[];
smem[sind] = gmem[gind];
```

Involves:

1. Copying data from global memory to register
2. Copying data from register to shared memory

- STALL [Scoreboard wait]
- Wasting registers and L1 bandwidth



Asynchronous Data Copies

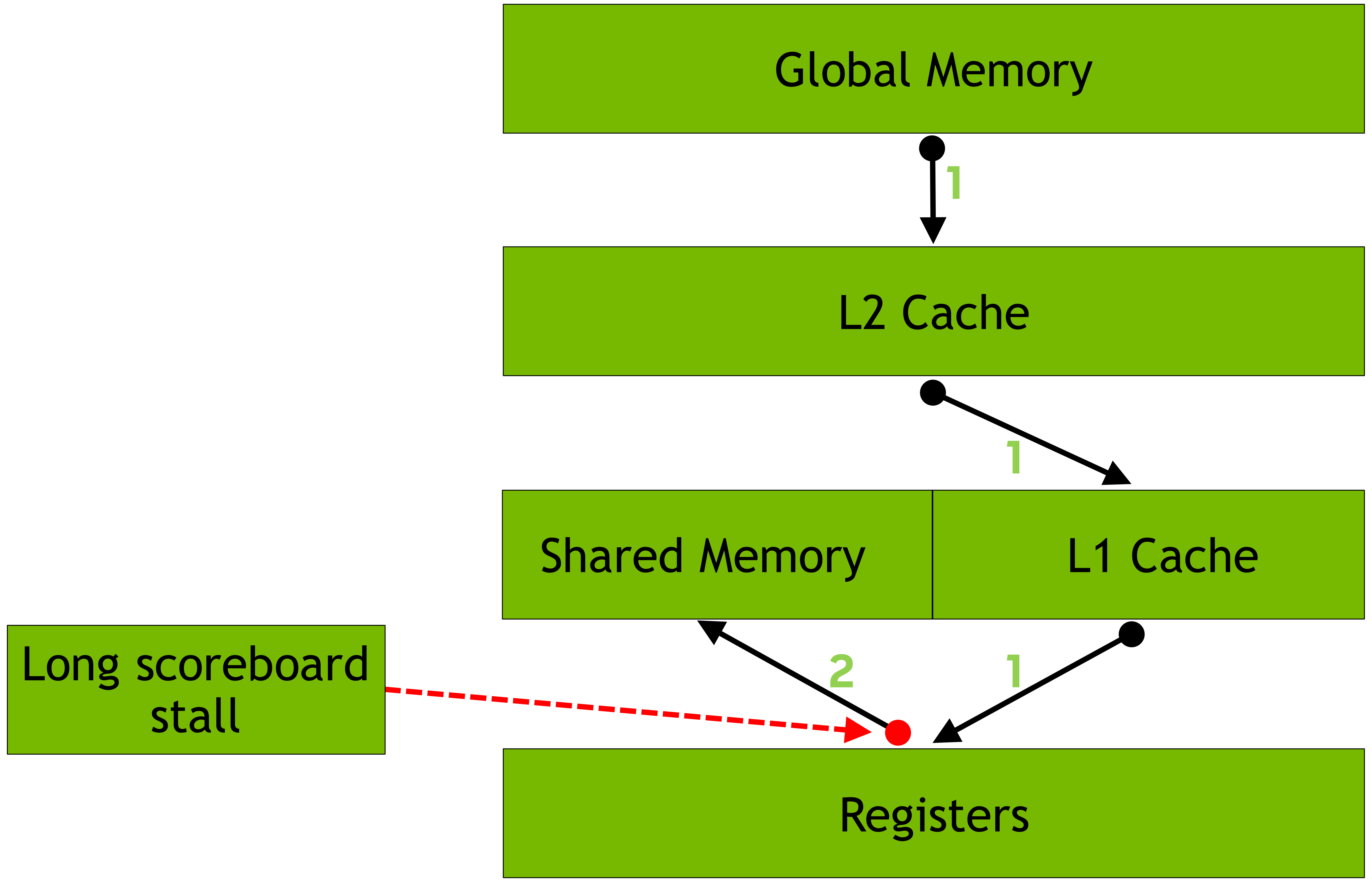
Synchronous copies dissected

```
T* gmem;
__shared__ T smem[];
smem[sind] = gmem[gind];
```

Involves:

1. Copying data from global memory to register
2. Copying data from register to shared memory

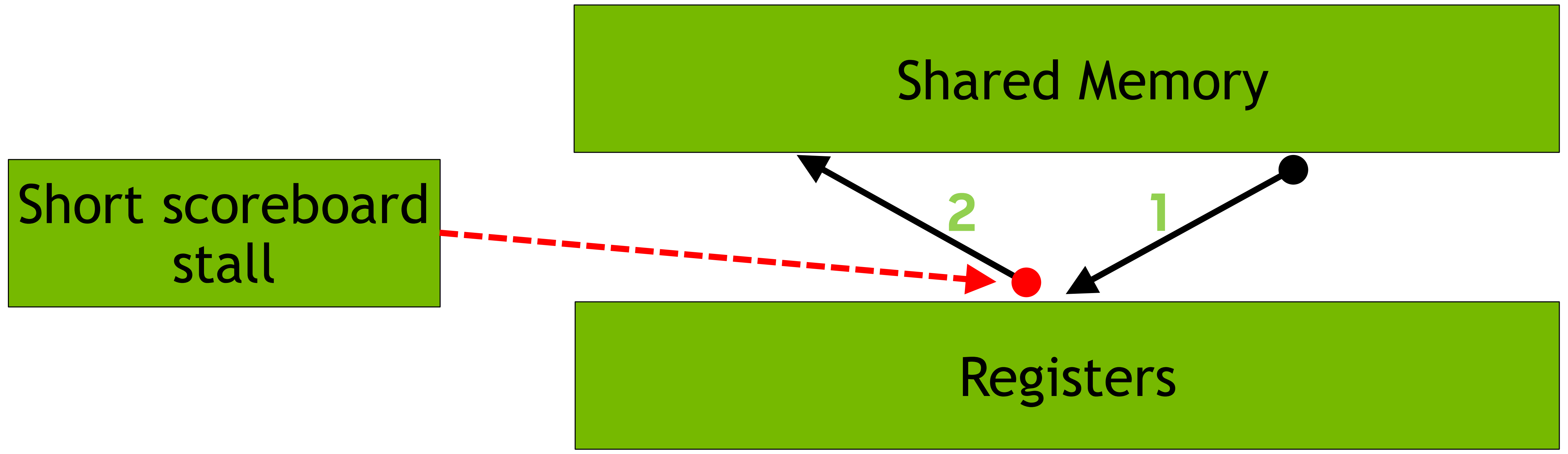
- STALL [Scoreboard wait]
- Wasting registers and L1 bandwidth



```
smem[sind1] = smem[sind2];
```

Involves:

1. Copying data from shared memory to register
2. Copying data from register to shared memory

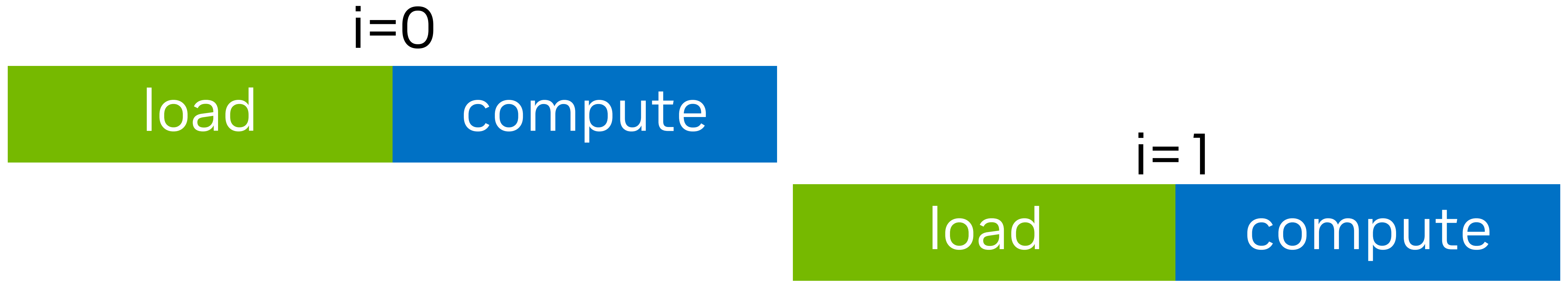


Asynchronous Data Copies

Synchronous copies – less bytes in flight ☹

Representative code:

```
 T* gmem;
 __shared__ T smem[..];
 for (int i = 0; i < n; ++i) {
     smem[sind] = g_mem[gind];
     gind += stride;
     __syncthreads();
     // compute with smem[]
     __syncthreads();
}
```



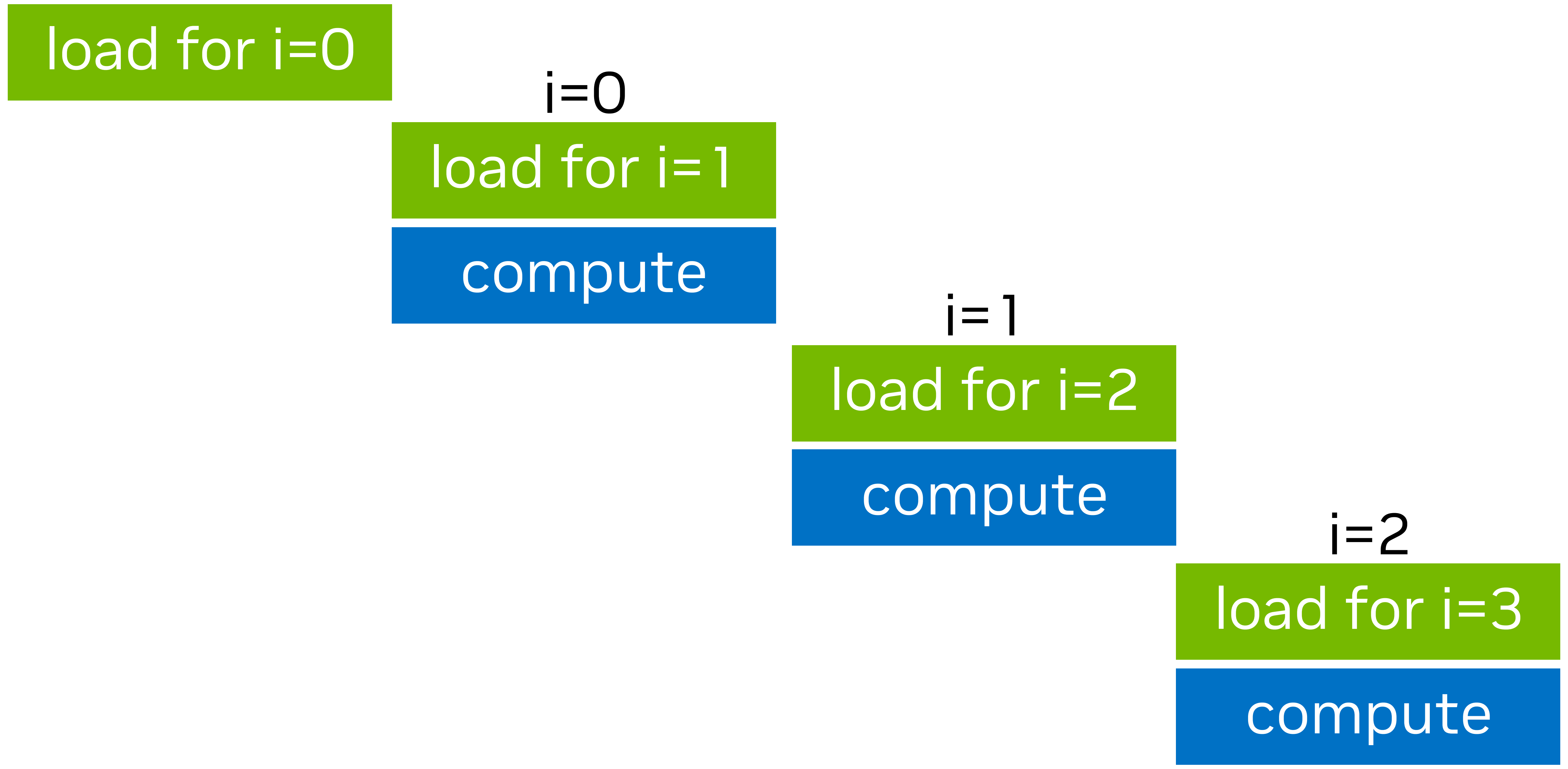
Copy and compute are sequenced.
Cannot fire SMEM prefetch during compute...

Asynchronous Data Copies

Asynchronous copies – more bytes in flight ☺

Asynchronous copy benefits:

- **Increases bytes in flight.**
- Other benefits:
  - Register bypass
  - Less L1 traffic
  - Less MIO pressure (fewer instructions)
  - Better dependency management

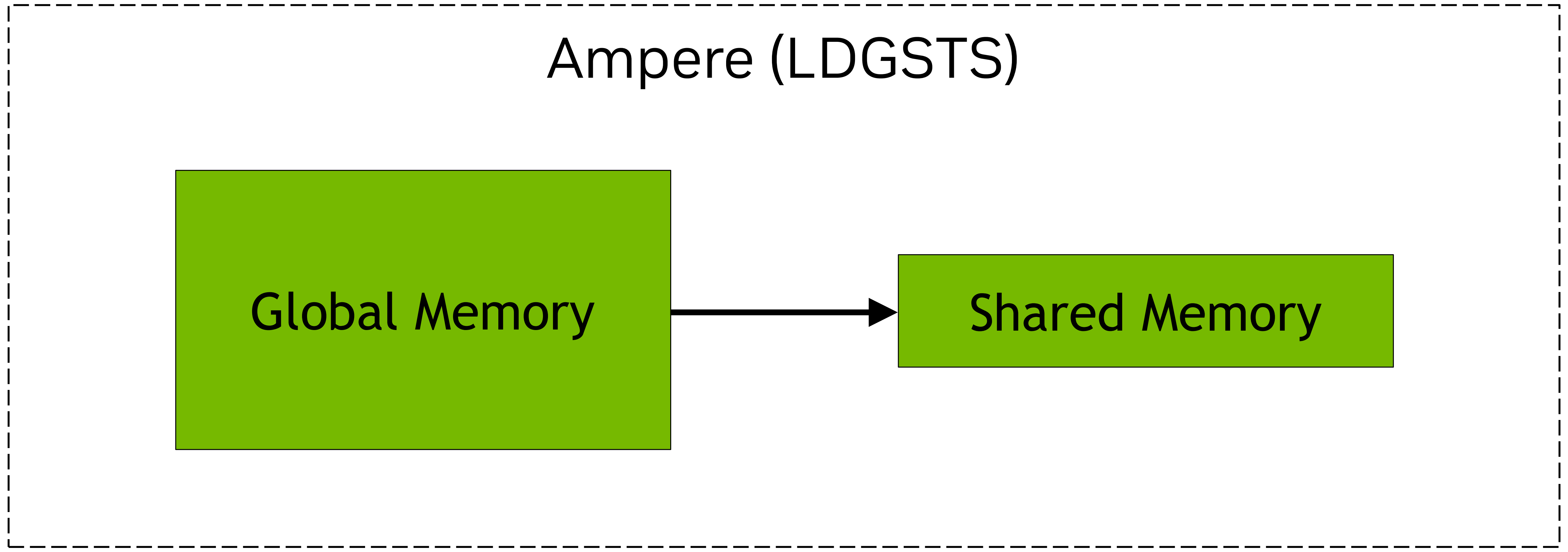


Asynchronous Data Copies

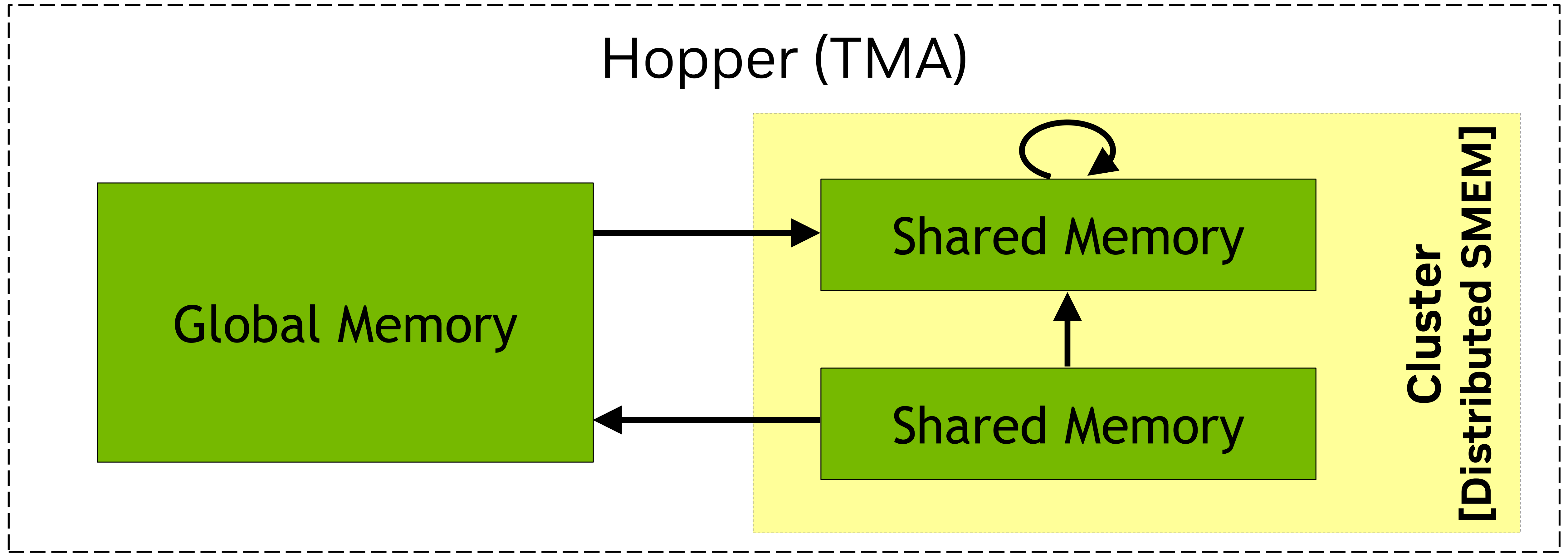
Asynchronous copies summary

HW	Direction	Operation	Synchronization	C++	SASS
Ampere	GMEM ➡ SMEM	Asynchronous version of smem[sind] := gmem[gind]	Thread-local SMEM barrier	__pipeline_memcpy_async	LDGSTS
Hopper	REG ➡ DSMEM	Asynchronous version of dsmem[sind] := var	DSMEM barrier	cuda::ptx::st_async	STAS
TMA	GMEM ➡ (D)SMEM	1D – 5D block copy Uniform	(D)SMEM barrier	cuda::device::experimental:: cp_async_bulk_tensor_Nd_global_to_shared	UTMALDG
	SMEM ➡ GMEM		Thread-local	cp_async_bulk_tensor_Nd_shared_to_global	UTMASTG
	GMEM ➡ SMEM	1D block copy Uniform	SMEM barrier	cuda::device::experimental:: cp_async_bulk_global_to_shared	UBLKCP
	SMEM ➡ GMEM		Thread-local	cp_async_bulk_shared_to_global	
	SMEM ➡ (D)SMEM		(D)SMEM barrier	cuda::ptx::cp_async_bulk*	

* Available in latest github CCCL



Ampere (LDGSTS)

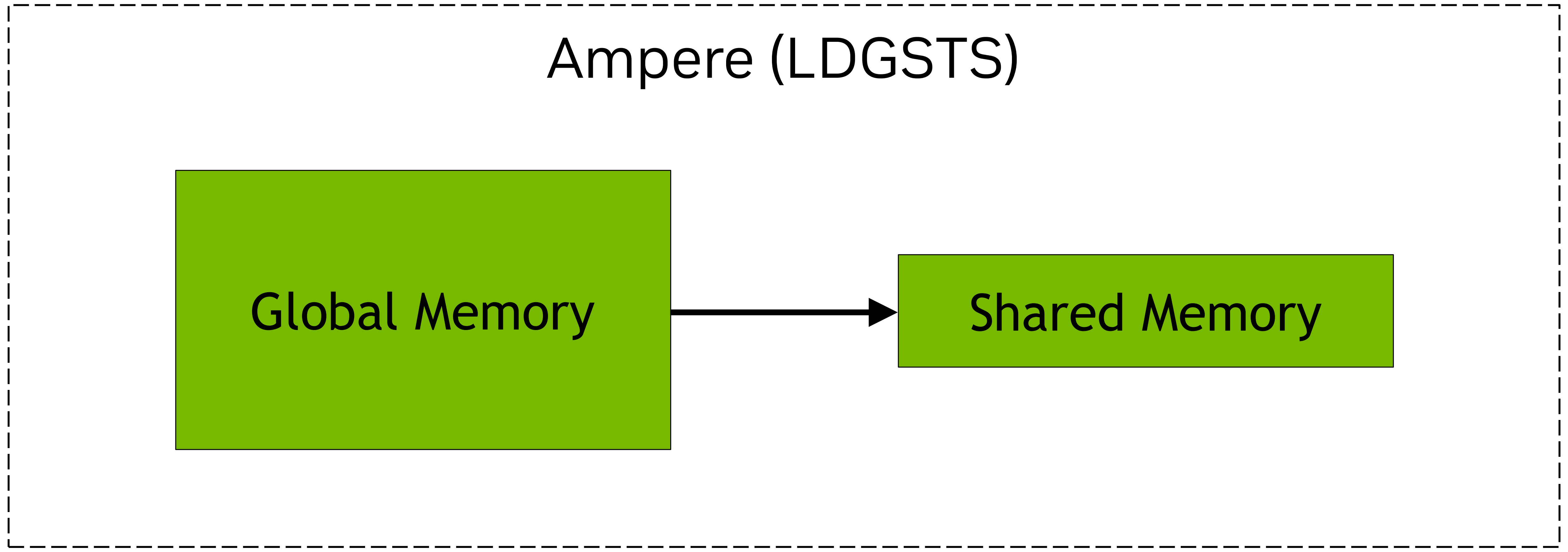


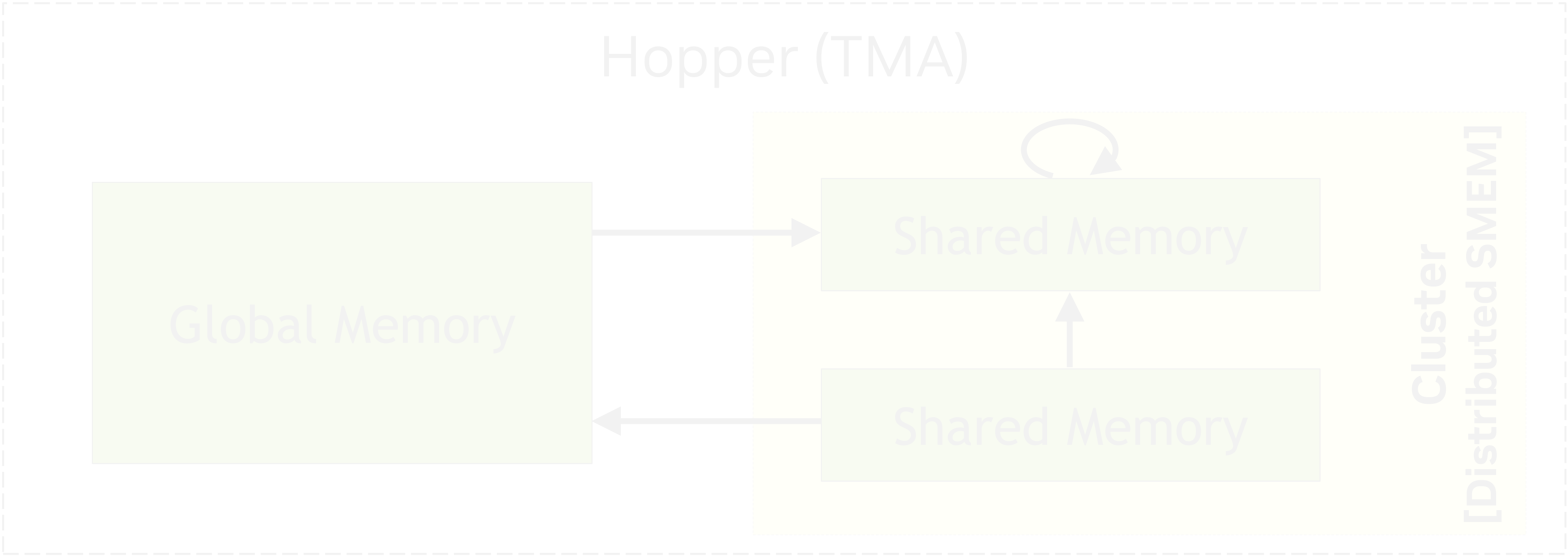
Hopper (TMA)

Asynchronous Data Copies

Asynchronous copies – LDGSTS

HW	Direction	Operation	Synchronization	C++	SASS
Ampere	GMEM ➡ SMEM	Asynchronous version of smem[sind] := gmem[gind]	Thread-local SMEM barrier	__pipeline_memcpy_async	LDGSTS
Hopper	REG ➡ DSMEM	Asynchronous version of dsmem[sind] := var	DSMEM barrier	cuda::ptx::st_async	STAS
TMA	GMEM ➡ (D)SMEM	1D – 5D block copy Uniform	(D)SMEM barrier	cuda::device::experimental:: cp_async_bulk_tensor_Nd_global_to_shared cp_async_bulk_tensor_Nd_shared_to_global	UTMALDG
	SMEM ➡ GMEM		Thread-local		UTMASTG
	GMEM ➡ SMEM	1D block copy Uniform	SMEM barrier	cuda::device::experimental:: cp_async_bulk_global_to_shared cp_async_bulk_shared_to_global	UBLKCP
	SMEM ➡ GMEM		Thread-local		
	SMEM ➡ (D)SMEM		(D)SMEM barrier	cuda::ptx::cp_async_bulk*	



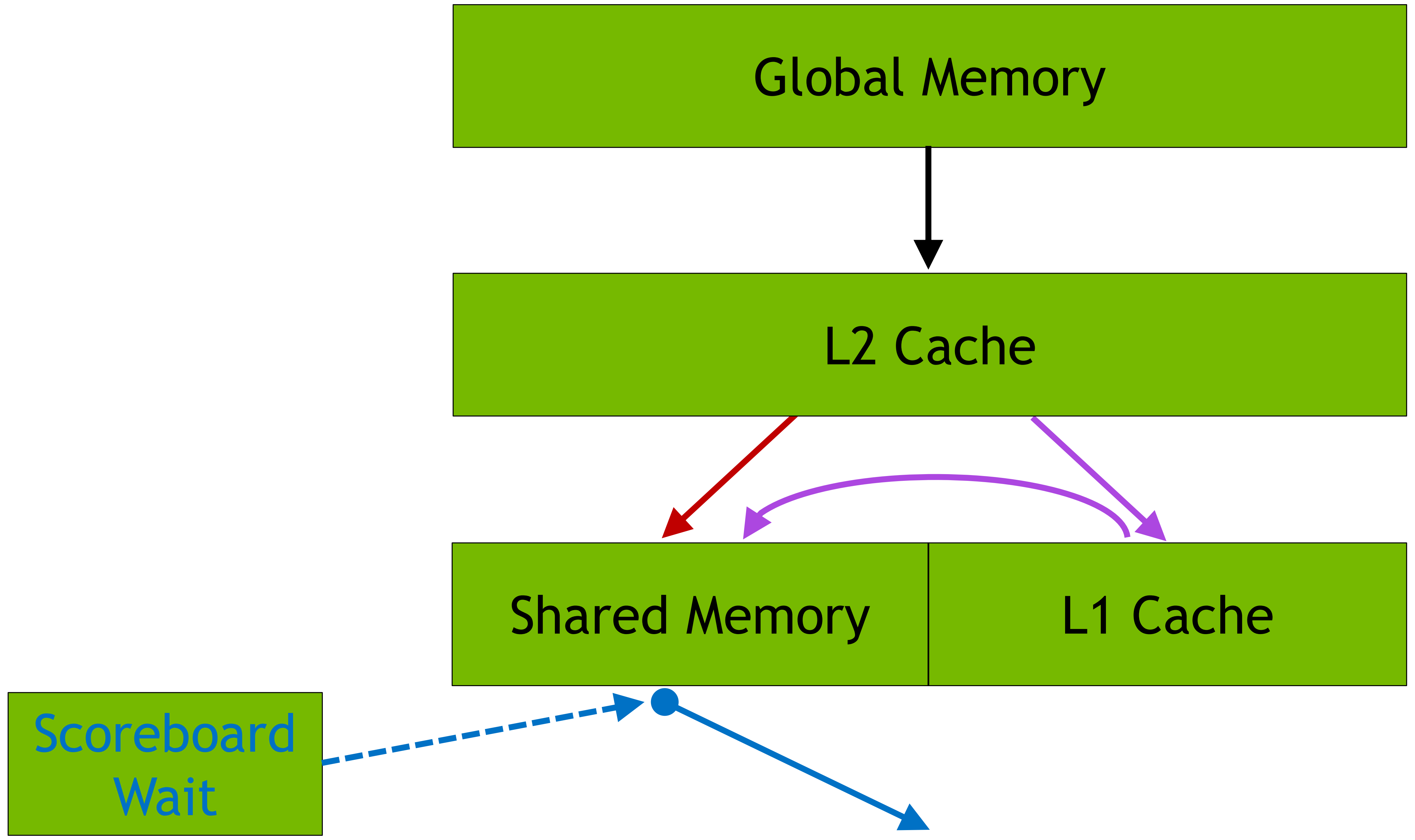


Asynchronous Data Copies

LDGSTS dissected

Asynchronous version of `smem[sind] = gmem[gind];`

```
__pipeline_memcpy_async(&smem[sind], &gmem[gind], sizeof(T));
```



Features:

1. Bypassing registers
   (less registers may lead to better occupancy / reduced spillage)
2. Two modes:
   - L1 bypass mode: kicks in when `sizeof(T)` and alignment are 16 B (benefit: no L1 pollution)
   - L1 access mode: when `sizeof(T)` and alignment are 4 or 8 B

Hopper note: shared memory must be local SMEM when CTA is part of CGA.

Asynchronous Data Copies

LDGSTS copies and synchronization

```
T* gmem;

__shared__ T smem[..];

__pipeline_commit();

__pipeline_memcpy_async(...);
...
__pipeline_memcpy_async(...);

__pipeline_commit();

__pipeline_wait_prior(N);
```

– issues async load instructions

– (non-blocking) batch-counting dependency barrier for previous loads released when data have written to SHMEM (for the executing thread)

– blocks until all previous commits except last N are “ready”

Asynchronous Data Copies

LDGSTS copies and synchronization

```
T* gmem;
__shared__ T smem[..];
```

```
__pipeline_memcpy_async(...);
__pipeline_memcpy_async(...);
__pipeline_commit();

__pipeline_memcpy_async(...);
__pipeline_memcpy_async(...);
__pipeline_memcpy_async(...);
__pipeline_commit();

__pipeline_memcpy_async(...);
__pipeline_commit();

__pipeline_wait_prior(2);
```

Async copies ready after wait

```
__pipeline_memcpy_async(...);
__pipeline_memcpy_async(...);
__pipeline_commit();

__pipeline_memcpy_async(...);
__pipeline_memcpy_async(...);
__pipeline_memcpy_async(...);
__pipeline_commit();

__pipeline_memcpy_async(...);
__pipeline_commit();

__pipeline_wait_prior(0);
```

Async copies ready after wait

Asynchronous Data Copies

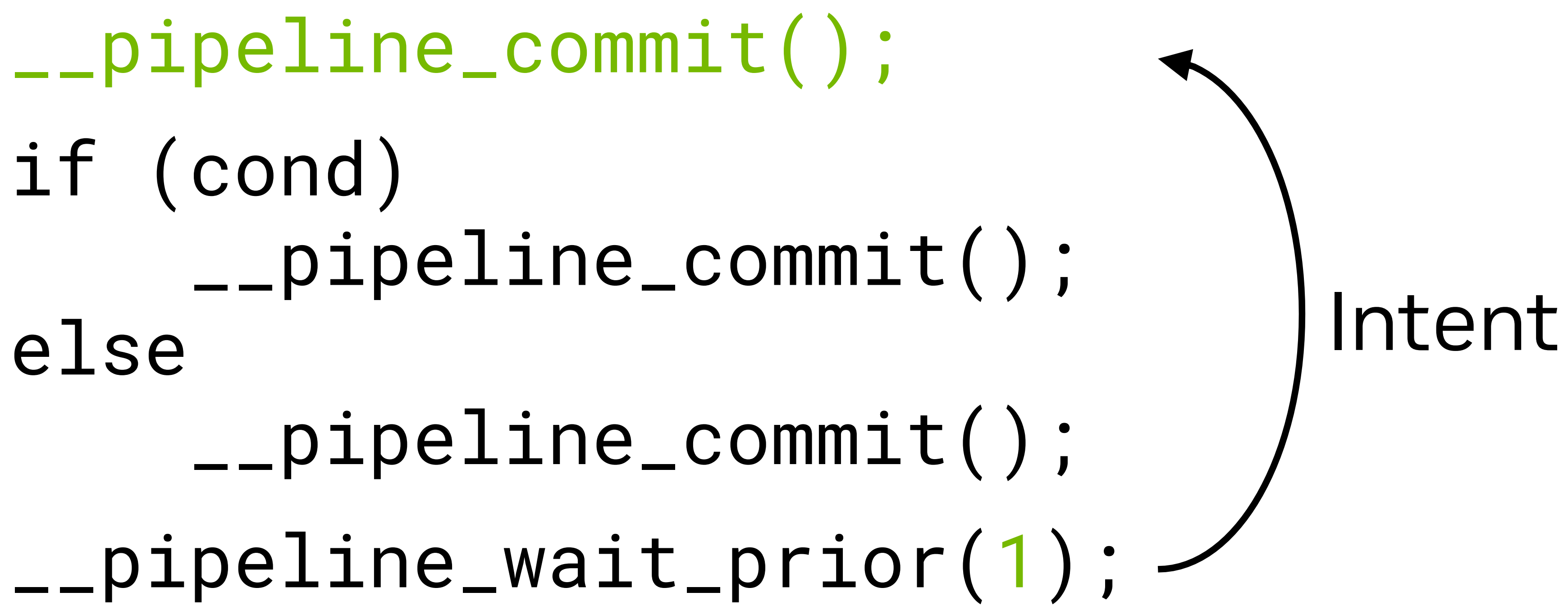
LDGSTS copies and synchronization

`__pipeline_commit` behaves as a full-warp level instruction regardless of warp divergence!

The code below is equivalent to 2 commits in case of divergence (and up to 32 for fully diverged warp):

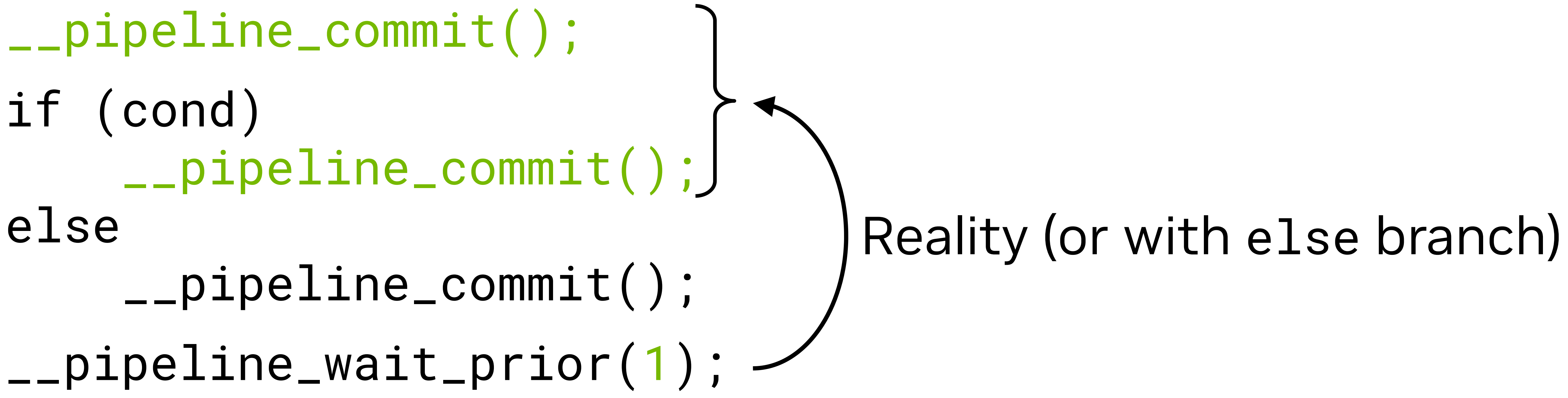
```
if (cond)
    __pipeline_commit();
else
    __pipeline_commit();
```

Consequence: overwait for 1st batch in case of warp divergence:



```
__pipeline_commit();
if (cond)
    __pipeline_commit();
else
    __pipeline_commit();
__pipeline_wait_prior(1);
```

Intent



```
__pipeline_commit();
if (cond)
    __pipeline_commit();
else
    __pipeline_commit();
__pipeline_wait_prior(1);
```

Reality (or with `else` branch)

Recommendation: do not commit in divergent code!

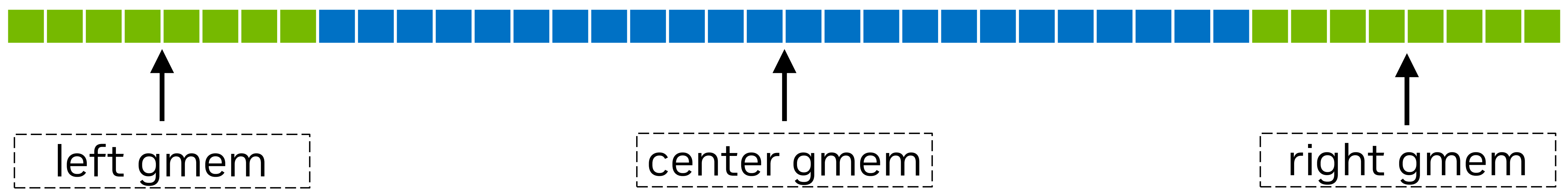
Asynchronous Data Copies

~~LDGSTS~~ use case: batching loads in conditional code

```
T const* __restrict__ left;
T const* __restrict__ right;
T const* __restrict__ center;

// Halo-Center-Halo.
__shared__ T smem[8 + 32 + 8];
auto const tx = threadIdx.x;

if (tx < 8)
    smem[tx] = left[tx];

else if (tx >= 32 - 8)
    smem[tx + 16] = right[tx];

smem[tx + 8] = center[tx];

__syncthreads();

// COMPUTE STENCIL
```

Compiler generates (pseudocode):

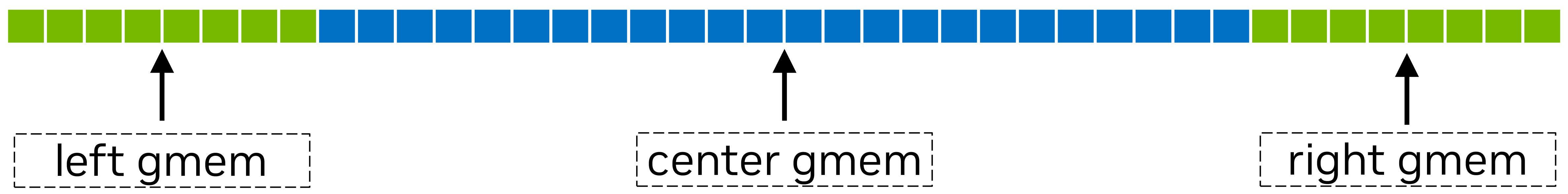
```
        BRA LABEL1
        LDG R1, [left]
        STS [..] R1        ←  STALL
        GOTO LABEL2

LABEL1: BRA LABEL2
        LDG R2, [right]
        STS [..] R2        ←  STALL

LABEL2: LDG R3, [center]
        STS [..] R3        ←  STALL
```



Asynchronous Data Copies

~~LDGSTS~~ use case: batching loads in conditional code

```
T const* __restrict__ left;
T const* __restrict__ right;
T const* __restrict__ center;

// Halo-Center-Halo.
__shared__ T smem[8 + 32 + 8];
auto const tx = threadIdx.x;

if (tx < 8)
    smem[tx] = left[tx];

else if (tx >= 32 - 8)
    smem[tx + 16] = right[tx];

smem[tx + 8] = center[tx];

__syncthreads();

// COMPUTE STENCIL
```



Compiler generates (pseudocode):

```
        BRA LABEL1
        LDG R1, [left]
        STS [..] R1      ← STALL
        GOTO LABEL2

LABEL1: BRA LABEL2
        LDG R2, [right]
        STS [..] R2      ← STALL

LABEL2: LDG R3, [center]
        STS [..] R3      ← STALL
```

Ideally, compiler could use predicates (but it may not):

```
PREDICATE1 LDG R1, [left]    }
PREDICATE2 LDG R2, [right]   }  ALL IN FLIGHT
           LDG R3, [center]  }

PREDICATE1 STS [..] R1
PREDICATE2 STS [..] R2
           STS [..] R3
```

Asynchronous Data Copies

LDGSTS use case: batching loads in conditional code

```
T const* __restrict__ left;
T const* __restrict__ right;
T const* __restrict__ center;

// Halo-Center-Halo.
__shared__ T smem[8 + 32 + 8];
auto const tx = threadIdx.x;

if (tx < 8)
    __pipeline_memcpy_async(
        &smem[tx], &left[tx], sizeof(T));

else if (tx >= 32 - 8)
    __pipeline_memcpy_async(
        &smem[tx + 16], &right[tx], sizeof(T));

__pipeline_memcpy_async(
&smem[tx + 8], &center[tx], sizeof(T));

__pipeline_commit();  ←  single commit, all in flight!

__pipeline_wait_prior<0> ();
__syncthreads();

// COMPUTE STENCIL
```

Compiler generates (pseudocode):

```
        BRA LABEL1
        LDGSTS [..], [left]    ←  NO STALL
        GOTO LABEL2

LABEL1: BRA LABEL2
        LDGSTS [..], [right]   ←  NO STALL

LABEL2: LDGSTS [..], [center]  ←  NO STALL

        LDGDEPBAR

        DEPBAR  ←  SINGLE STALL
```

Asynchronous Data Copies

LDGSTS use case: batching loads in conditional code

```
__shared__ float smem[8 + 32 + 8][32];

int const tx = threadIdx.x;
int const ty = threadIdx.y;
int gind = (ty * stride_y) + (blockIdx.x * 32 + tx);

for (int i = 0; i < nz; ++i) {
    if (ty < 8)
        __pipeline_memcpy_async(&smem[ty][tx], &gl[gind], sizeof(float));
    else if (ty >= 24)
        __pipeline_memcpy_async(&smem[ty + 16][tx], &gr[gind], sizeof(float));
    __pipeline_memcpy_async(&smem[ty + 8][tx], &gc[gind], sizeof(float));

    __pipeline_commit();
    __pipeline_wait_prior(0);
    __syncthreads();

    // “STENCIL”:
    float d = smem[ty + 8][tx];
    #pragma unroll
    for (int r = 1; r <= 8; ++r)
        d += (smem[ty + 8 + r][tx] - smem[ty + 8 - r][tx]) * float(r + 0.1);

    out[gind] = d;
    gind += stride_z;
    __syncthreads();
}
```

Extra dimension added to code from previous slide
Note: condition is on ty => no divergence within warp

Asynchronous Data Copies

LDGSTS use case: batching loads in conditional code

```
__shared__ float smem[8 + 32 + 8][32];

int const tx = threadIdx.x;
int const ty = threadIdx.y;
int gind = (ty * stride_y) + (blockIdx.x * 32 + tx);

for (int i = 0; i < nz; ++i) {
    if (ty < 8)
        __pipeline_memcpy_async(&smem[ty][tx], &gl[gind], sizeof(float));
    else if (ty >= 24)
        __pipeline_memcpy_async(&smem[ty + 16][tx], &gr[gind], sizeof(float));
    __pipeline_memcpy_async(&smem[ty + 8][tx], &gc[gind], sizeof(float));

    __pipeline_commit();
    __pipeline_wait_prior(0);
    __syncthreads();

    // “STENCIL”:
    float d = smem[ty + 8][tx];
    #pragma unroll
    for (int r = 1; r <= 8; ++r)
        d += (smem[ty + 8 + r][tx] - smem[ty + 8 - r][tx]) * float(r + 0.1);

    out[gind] = d;
    gind += stride_z;
    __syncthreads();
}
```

H200 experiment.

Speedup over sync-copy:

- Un-staged: 1.3X

Memory throughput:

- Sync copy : 46%
- Un-staged: 67%

Extra dimension added to code from previous slide
Note: condition is on ty => no divergence within warp

Asynchronous Data Copies

LDGSTS use case: batching loads in conditional code

```
__shared__ float smem[8 + 32 + 8][32];

int const tx = threadIdx.x;
int const ty = threadIdx.y;
int gind = (ty * stride_y) + (blockIdx.x * 32 + tx);

for (int i = 0; i < nz; ++i) {
    if (ty < 8)
        __pipeline_memcpy_async(&smem[ty][tx], &gl[gind], sizeof(float));
    else if (ty >= 24)
        __pipeline_memcpy_async(&smem[ty + 16][tx], &gr[gind], sizeof(float));
    __pipeline_memcpy_async(&smem[ty + 8][tx], &gc[gind], sizeof(float));

    __pipeline_commit();
    __pipeline_wait_prior(0);
    __syncthreads();

    // “STENCIL”:
    float d = smem[ty + 8][tx];
    #pragma unroll
    for (int r = 1; r <= 8; ++r)
        d += (smem[ty + 8 + r][tx] - smem[ty + 8 - r][tx]) * float(r + 0.1);

    out[gind] = d;
    gind += stride_z;
    __syncthreads();
}
```

H200 experiment.

Speedup over sync-copy:

- Un-staged: 1.3X
- Two-stage: **1.7X**

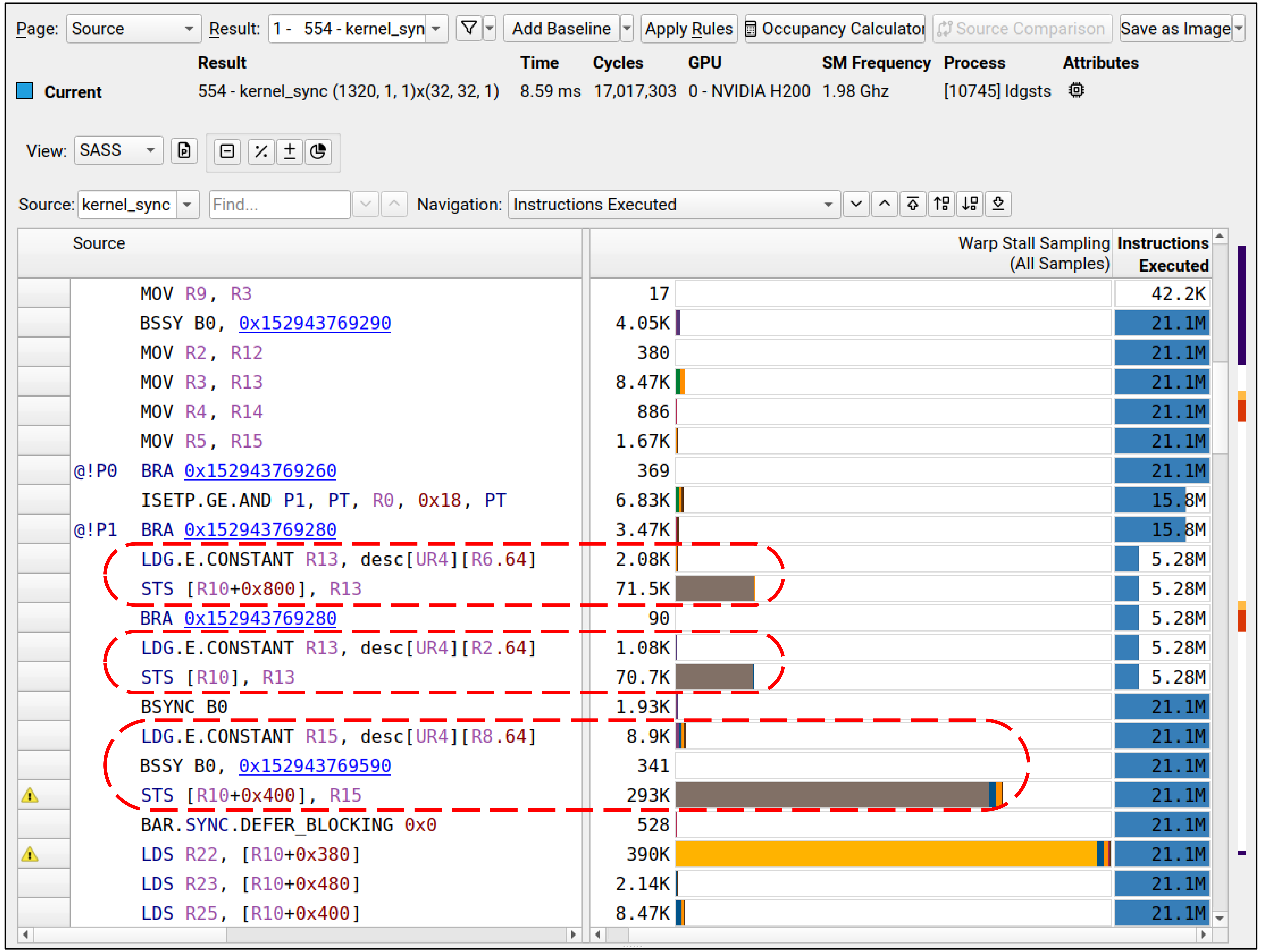
Memory throughput:

- Sync copy : 46%
- Un-staged: 67%
- Two-stage: **84%**

Extra dimension added to code from previous slide
Note: condition is on ty => no divergence within warp

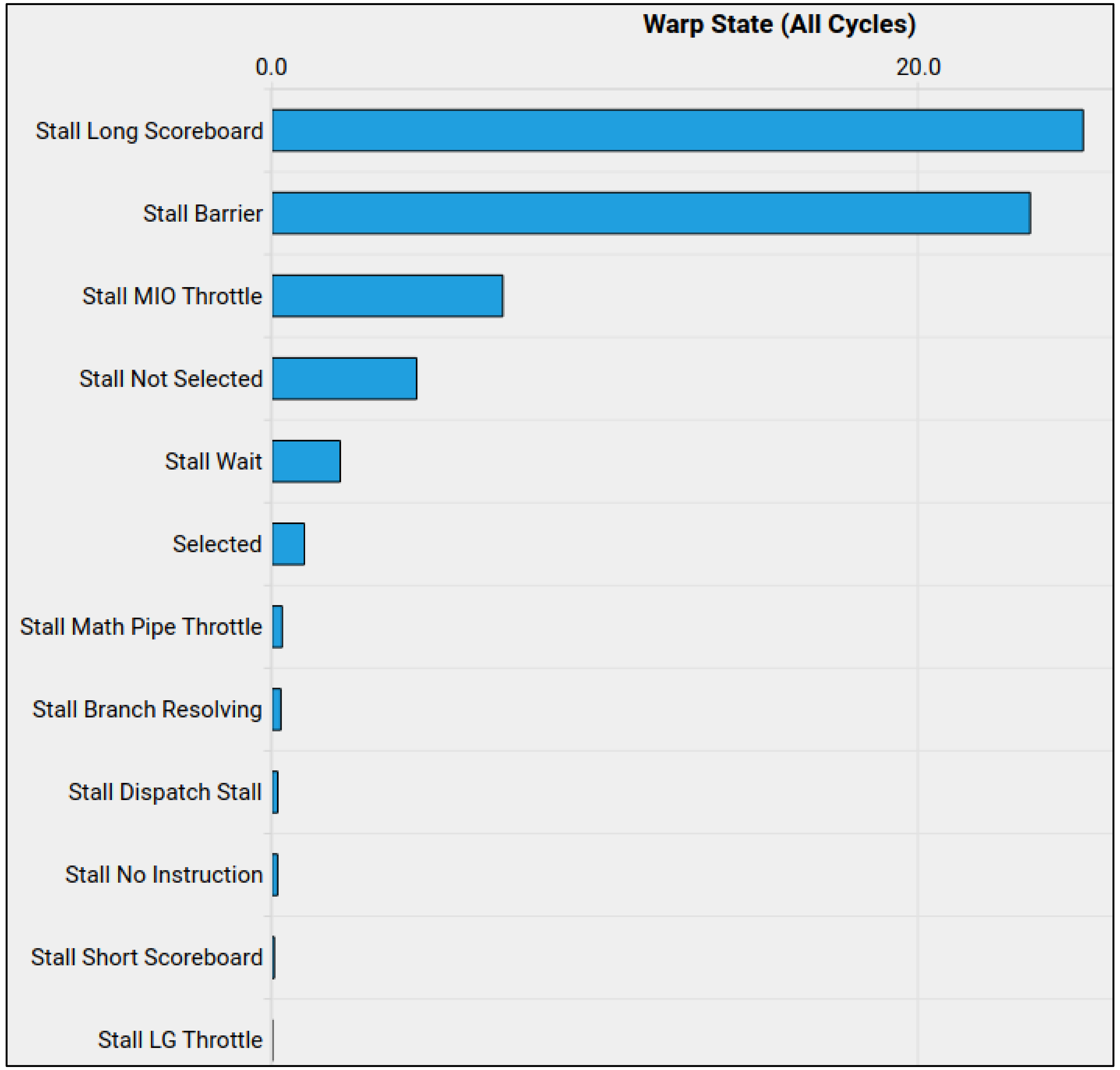
Asynchronous Data Copies

~~LDGSTS~~ use case: batching loads in conditional code – NCU



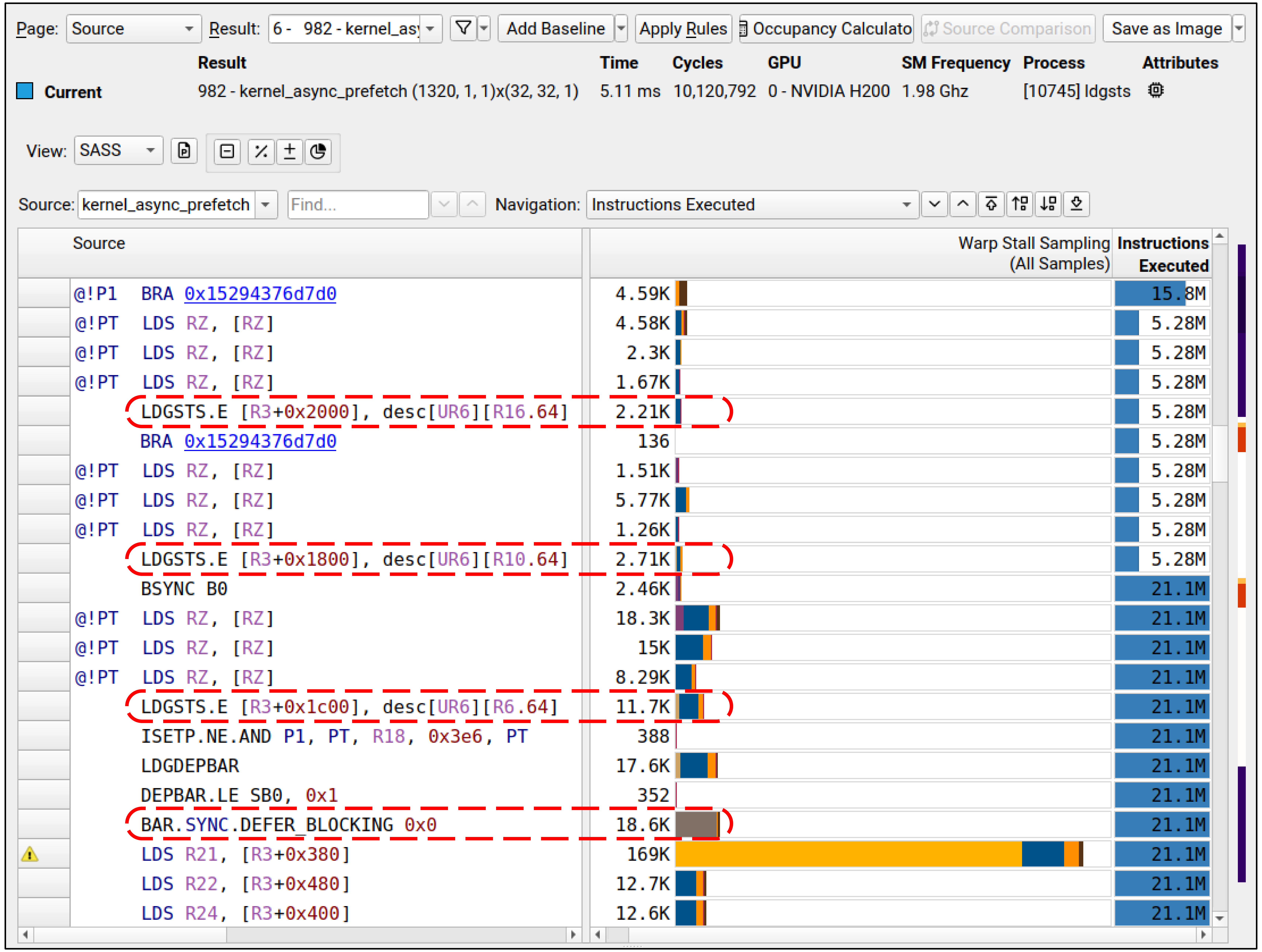
	Result	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	554 - kernel_sync (1320, 1, 1)x(32, 32, 1)	8.59 ms	17,017,303	0 - NVIDIA H200	1.98 Ghz	[10745] ldgsts	

Source	Warp Stall Sampling (All Samples)	Instructions Executed
MOV R9, R3	17	42.2K
BSSY B0, 0x152943769290	4.05K	21.1M
MOV R2, R12	380	21.1M
MOV R3, R13	8.47K	21.1M
MOV R4, R14	886	21.1M
MOV R5, R15	1.67K	21.1M
@!P0 BRA 0x152943769260	369	21.1M
ISETP.GE.AND P1, PT, R0, 0x18, PT	6.83K	15.8M
@!P1 BRA 0x152943769280	3.47K	15.8M
LDG.E.CONSTANT R13, desc[UR4][R6.64]	2.08K	5.28M
STS [R10+0x800], R13	71.5K	5.28M
BRA 0x152943769280	90	5.28M
LDG.E.CONSTANT R13, desc[UR4][R2.64]	1.08K	5.28M
STS [R10], R13	70.7K	5.28M
BSYNC B0	1.93K	21.1M
LDG.E.CONSTANT R15, desc[UR4][R8.64]	8.9K	21.1M
BSSY B0, 0x152943769590	341	21.1M
⚠ STS [R10+0x400], R15	293K	21.1M
BAR.SYNC.DEFER_BLOCKING 0x0	528	21.1M
⚠ LDS R22, [R10+0x380]	390K	21.1M
LDS R23, [R10+0x480]	2.14K	21.1M
LDS R25, [R10+0x400]	8.47K	21.1M



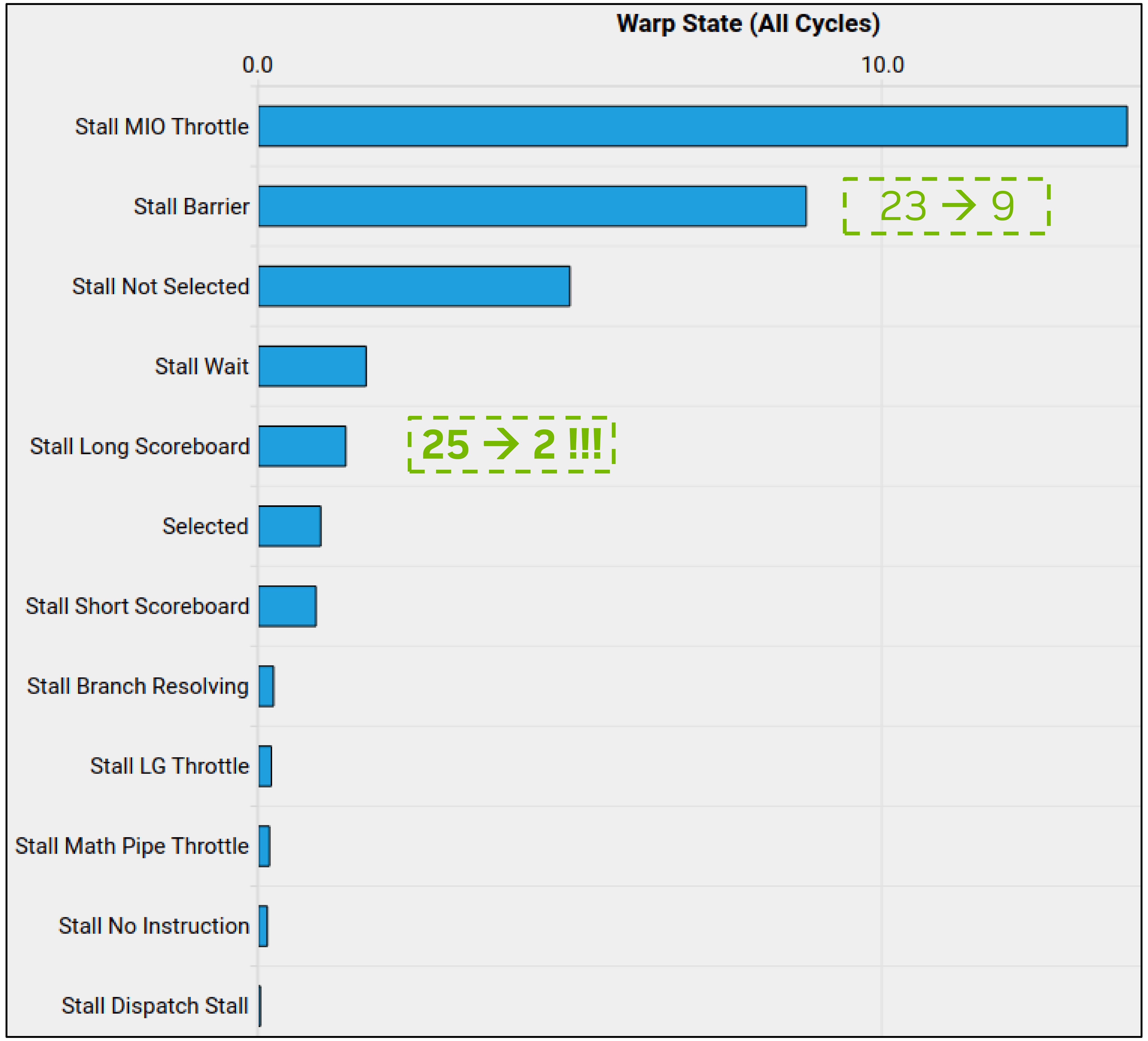
Asynchronous Data Copies

LDGSTS use case: batching loads in conditional code – NCU



	Result	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	982 - kernel_async_prefetch (1320, 1, 1)x(32, 32, 1)	5.11 ms	10,120,792	0 - NVIDIA H200	1.98 Ghz	[10745] ldgsts	

Source	Warp Stall Sampling (All Samples)	Instructions Executed
`@!P1 BRA 0x15294376d7d0`	4.59K	15.8M
`@!PT LDS RZ, [RZ]`	4.58K	5.28M
`@!PT LDS RZ, [RZ]`	2.3K	5.28M
`@!PT LDS RZ, [RZ]`	1.67K	5.28M
`LDGSTS.E [R3+0x2000], desc[UR6][R16.64]`	2.21K	5.28M
`BRA 0x15294376d7d0`	136	5.28M
`@!PT LDS RZ, [RZ]`	1.51K	5.28M
`@!PT LDS RZ, [RZ]`	5.77K	5.28M
`@!PT LDS RZ, [RZ]`	1.26K	5.28M
`LDGSTS.E [R3+0x1800], desc[UR6][R10.64]`	2.71K	5.28M
`BSYNC B0`	2.46K	21.1M
`@!PT LDS RZ, [RZ]`	18.3K	21.1M
`@!PT LDS RZ, [RZ]`	15K	21.1M
`@!PT LDS RZ, [RZ]`	8.29K	21.1M
`LDGSTS.E [R3+0x1c00], desc[UR6][R6.64]`	11.7K	21.1M
`ISETP.NE.AND P1, PT, R18, 0x3e6, PT`	388	21.1M
`LDGDEPBAR`	17.6K	21.1M
`DEPBAR.LE SB0, 0x1`	352	21.1M
`BAR.SYNC.DEFER_BLOCKING 0x0`	18.6K	21.1M
⚠ `LDS R21, [R3+0x380]`	169K	21.1M
`LDS R22, [R3+0x480]`	12.7K	21.1M
`LDS R24, [R3+0x400]`	12.6K	21.1M

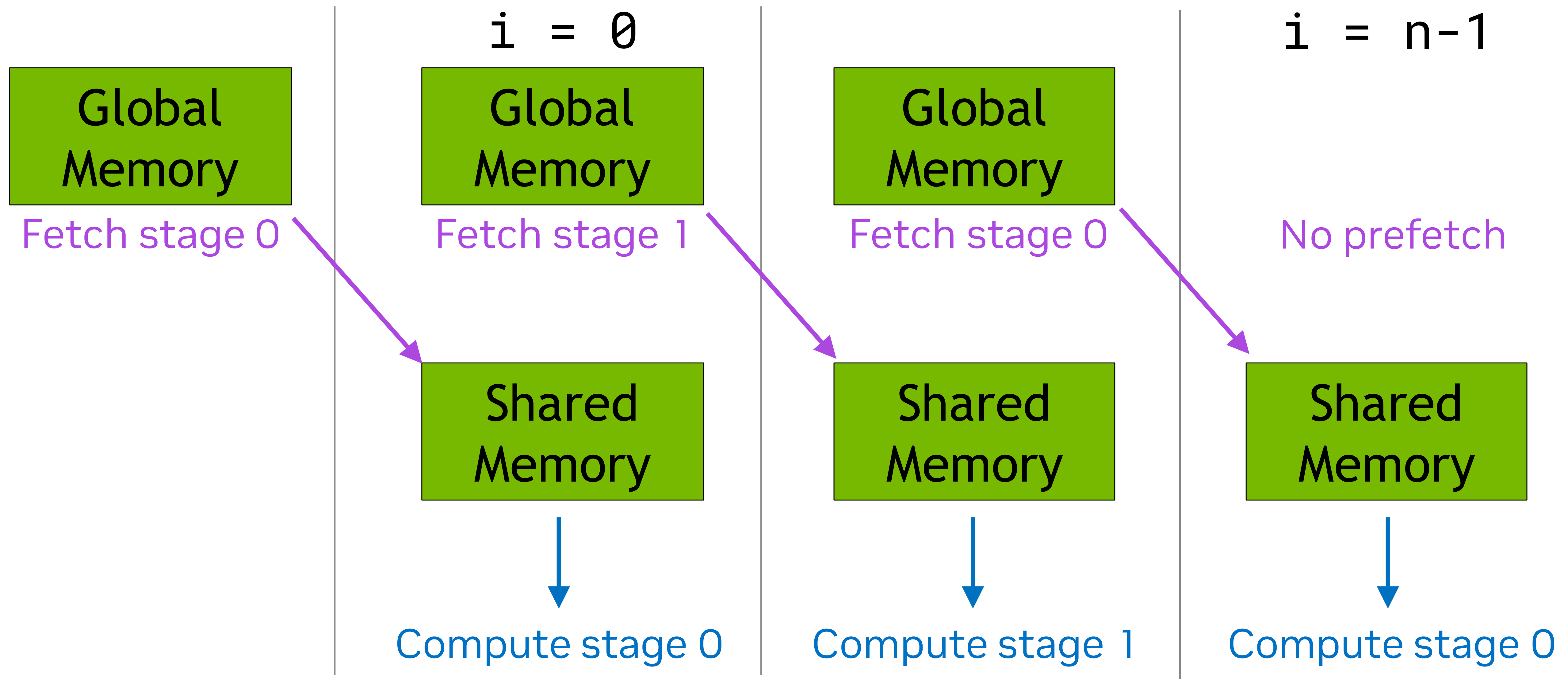


Asynchronous Data Copies

LDGSTS use case: staging/prefetching

```
T* gmem;

__shared__ T smem[2][..];

int is = 0;
__pipeline_memcpy_async(&smem[is][sind], &gmem[gind], sizeof(T));
__pipeline_commit();
is ^= 1;            gind += stride;

for (int i = 0; i < n - 1; ++i) {
    __pipeline_memcpy_async(&smem[is][sind], &gmem[gind], sizeof(T));
    __pipeline_commit();
    is ^= 1;            gind += stride;

    __pipeline_wait_prior(1);
    __syncthreads();

    // compute with smem[is][]

    __syncthreads();
}

__pipeline_wait_prior(0);
__syncthreads();

// compute with smem[is][]
```

```
// Synchronous copy code:
for (int i = 0; i < n; ++i) {
    smem[sind] = g_mem[gind];
    gind += stride;
    __syncthreads();
    // compute with smem[]
    __syncthreads();
}
```



i = 0

i = n-1

Global Memory

Fetch stage 0

Global Memory

Fetch stage 1

Global Memory

Fetch stage 0

No prefetch

Shared Memory

Compute stage 0

Shared Memory

Compute stage 1

Shared Memory

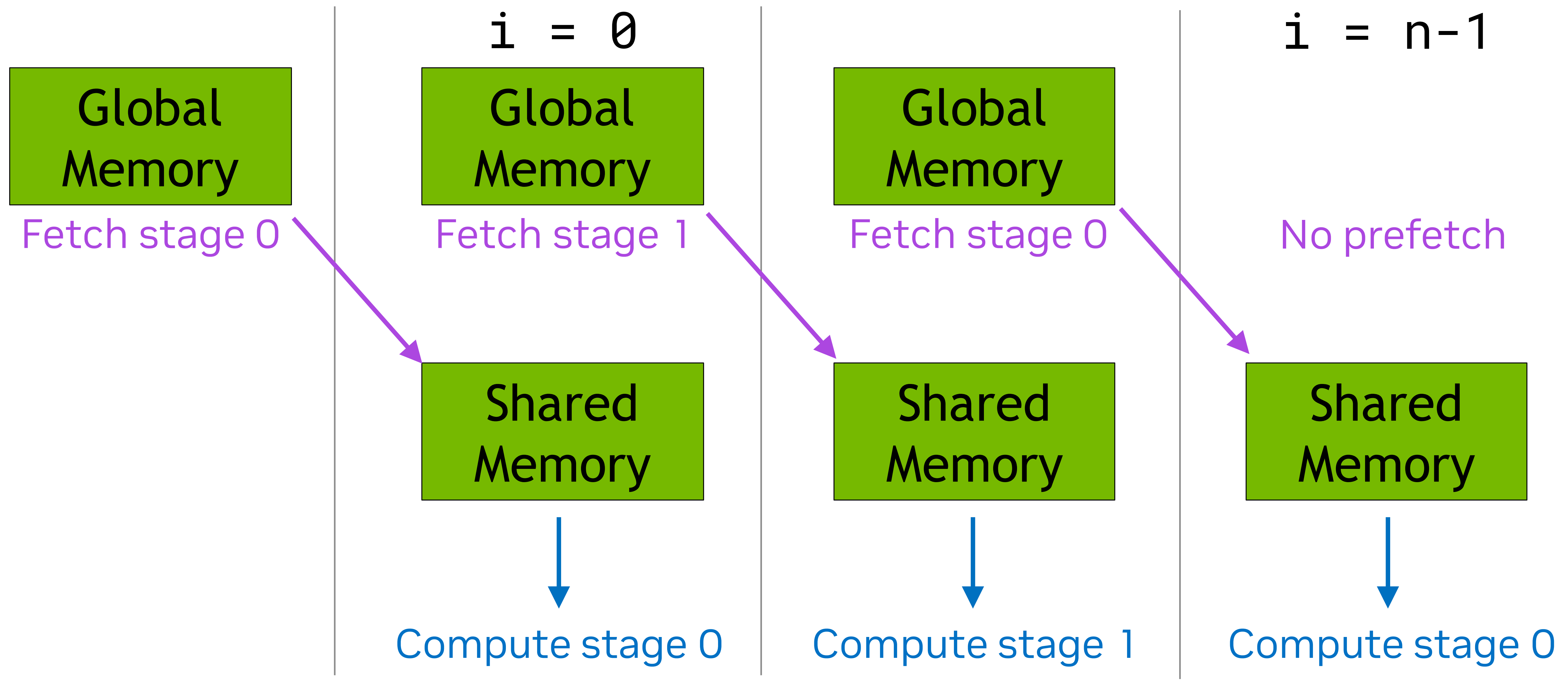
Compute stage 0

Asynchronous Data Copies

LDGSTS use case: staging/prefetching

```
T* gmem;

__shared__ T smem[2][..];

int is = 0;
__pipeline_memcpy_async(&smem[is][sind], &gmem[gind], sizeof(T));
__pipeline_commit();
is ^= 1;            gind += stride;

for (int i = 0; i < n - 1; ++i) {
    __pipeline_memcpy_async(&smem[is][sind], &gmem[gind], sizeof(T));
    __pipeline_commit();
    is ^= 1;            gind += stride;

    __pipeline_wait_prior(1);
    __syncthreads();

    // compute with smem[is][]

    __syncthreads();
}

__pipeline_wait_prior(0);
__syncthreads();

// compute with smem[is][]
```

```
// Synchronous copy code:
for (int i = 0; i < n; ++i) {
    smem[sind] = g_mem[gind];
    gind += stride;
    __syncthreads();
    // compute with smem[]
    __syncthreads();
}
```



i = 0

i = n-1

Global Memory

Fetch stage 0

Global Memory

Fetch stage 1

Global Memory

Fetch stage 0

No prefetch

Shared Memory

Compute stage 0

Shared Memory

Compute stage 1

Shared Memory

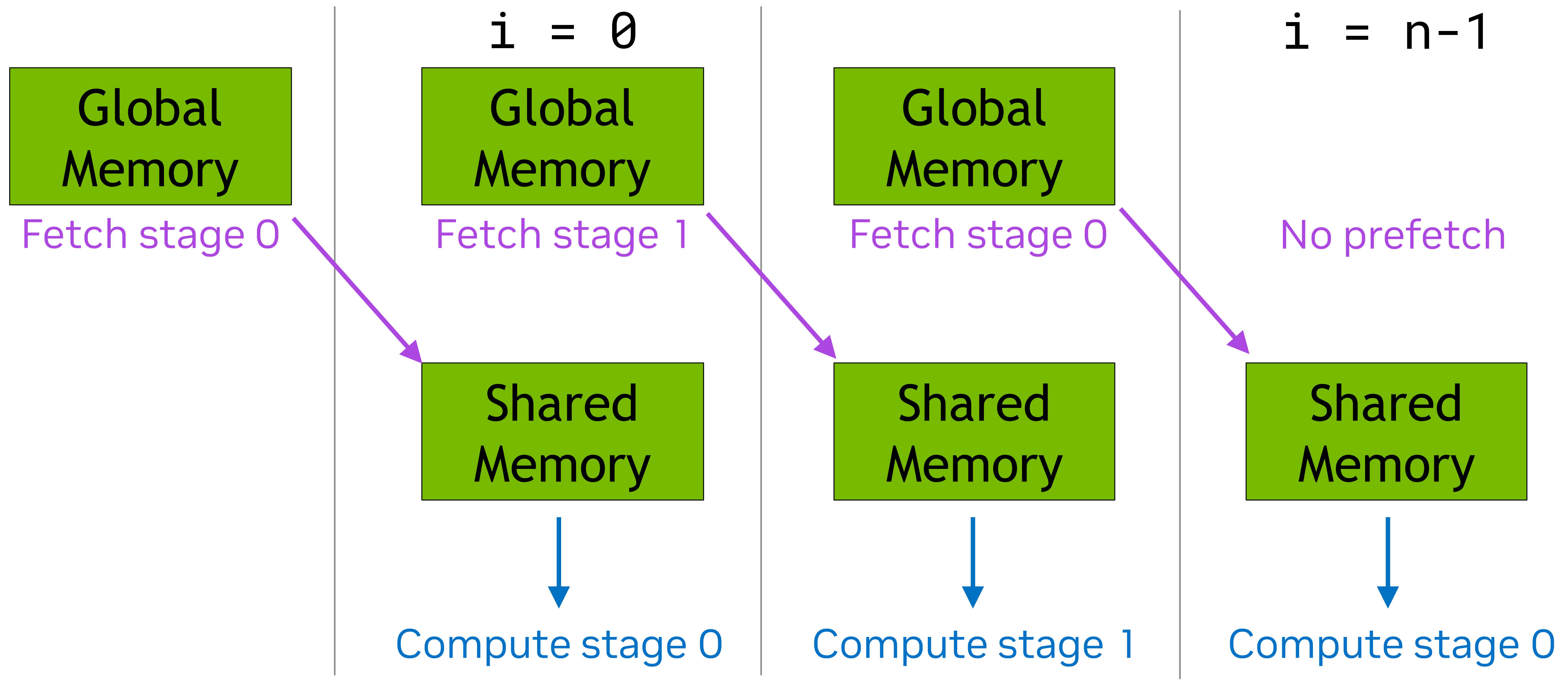
Compute stage 0

Asynchronous Data Copies

LDGSTS use case: staging/prefetching

```
T* gmem;

__shared__ T smem[3][..];

int is = 0;
__pipeline_memcpy_async(&smem[is][sind], &gmem[gind], sizeof(T));
__pipeline_commit();
is = (is + 1) % 3;  gind += stride;

for (int i = 0; i < n - 1; ++i) {
    __pipeline_memcpy_async(&smem[is][sind], &gmem[gind], sizeof(T));
    __pipeline_commit();
    is = (is + 1) % 3;  gind += stride;

    __pipeline_wait_prior(1);
    __syncthreads();

    // compute with smem[is][]

    ~~__syncthreads();~~
}

__pipeline_wait_prior(0);
__syncthreads();

// compute with smem[is][]
```

```
// Synchronous copy code:
for (int i = 0; i < n; ++i) {
    smem[sind] = g_mem[gind];
    gind += stride;
    __syncthreads();
    // compute with smem[]
    __syncthreads();
}
```



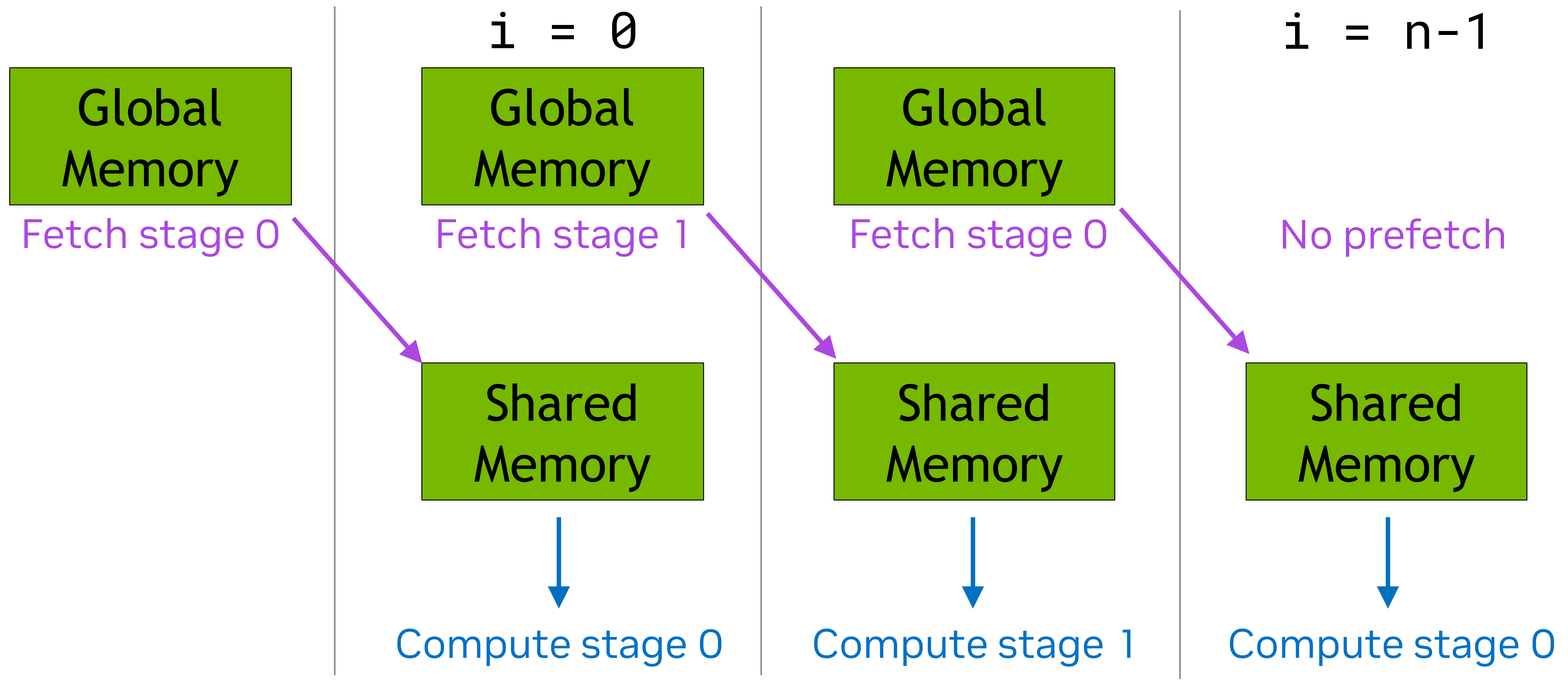
Asynchronous Data Copies

LDGSTS use case: staging/prefetching

More stages
(prefetch multiple iterations)
=> more Bytes in flight!

```
T* gmem;

__shared__ T smem[3][..];

int is = 0;
__pipeline_memcpy_async(&smem[is][sind], &gmem[gind], sizeof(T));
__pipeline_commit();
is = (is + 1) % 3;  gind += stride;

for (int i = 0; i < n - 1; ++i) {
    __pipeline_memcpy_async(&smem[is][sind], &gmem[gind], sizeof(T));
    __pipeline_commit();
    is = (is + 1) % 3;  gind += stride;

    __pipeline_wait_prior(1);
    __syncthreads();

    // compute with smem[is][]

    ~~__syncthreads();~~
}

__pipeline_wait_prior(0);
__syncthreads();

// compute with smem[is][]
```

```
// Synchronous copy code:
for (int i = 0; i < n; ++i) {
    smem[sind] = g_mem[gind];
    gind += stride;
    __syncthreads();
    // compute with smem[]
    __syncthreads();
}
```



Asynchronous Data Copies

Ways to code LDGSTS

```
// primitives  <cuda_pipeline.h>:
__pipeline_memcpy_async(&smem[sind], &gmem[gind], sizeof(T)); // LDGSTS
__pipeline_commit();       // LDGDEPBAR
__pipeline_wait_prior(N);  // DEPBAR.LE SB,N

// libcu++  <cuda/pipeline.h>:
auto pipeline = cuda::make_pipeline();
pipeline.producer_acquire();
cuda::memcpy_async(&smem[sind], &gmem[gind], sizeof(T), pipeline); // LDGSTS
pipeline.producer_commit();                       // LDGDEPBAR
cuda::pipeline_consumer_wait_prior<N>(pipeline);  // DEPBAR.LE SB,N
pipeline.consumer_release();

// cooperative_groups  <cooperative_groups/memcpy_async.h>:
auto block = cg:: this_thread_block();
cg::memcpy_async(block, smem, gmem, sizeof(T) * block.size());  // LDGSTS + LDGDEPBAR
cg::wait_prior<N>(block);                                       // DEPBAR.LE SB,N
```

Asynchronous Data Copies

LDGSTS with barrier

Producer/consumer example:

Loop, at each iteration:

1. Producer threads copy area of GMEM to SMEM
2. Consumer threads access SMEM after copy completion

iter 0
GMEM
producers
SMEM
consumers

Asynchronous Data Copies

LDGSTS with barrier

Producer/consumer example:

Loop, at each iteration:

1. Producer threads copy area of GMEM to SMEM
2. Consumer threads access SMEM after copy completion

iter 1
GMEM
producers
SMEM
consumers

Asynchronous Data Copies

LDGSTS with barrier

```
__global__ void kernel (T const* __restrict__ gmem, int n)
{
    __shared__ T smem[SMEM_SIZE];
    __shared__ __mbarrier_t bar;

    bool is_producer = (threadIdx.x < N_PRODUCERS);

    if (threadIdx.x == 0)
        __mbarrier_init(&bar, N_PRODUCERS); // count := N_PRODUCERS (reset)
    __syncthreads();

    for (int it = 0, phase = 0; it < n; ++it) {

        if (is_producer) {

            for (int i = threadIdx.x; i < SMEM_SIZE; i += N_PRODUCERS)
                __pipeline_memcpy_async(&smem[i], &gmem[it * SMEM_SIZE + i], sizeof(T));

In this order!
            __pipeline_arrive_on(&bar);     // count += N_PRODUCERS and count -= N_PRODUCERS by HW on copy completion
            __mbarrier_arrive(&bar);        // count -= N_PRODUCERS
        }






    }
}
```

Asynchronous Data Copies

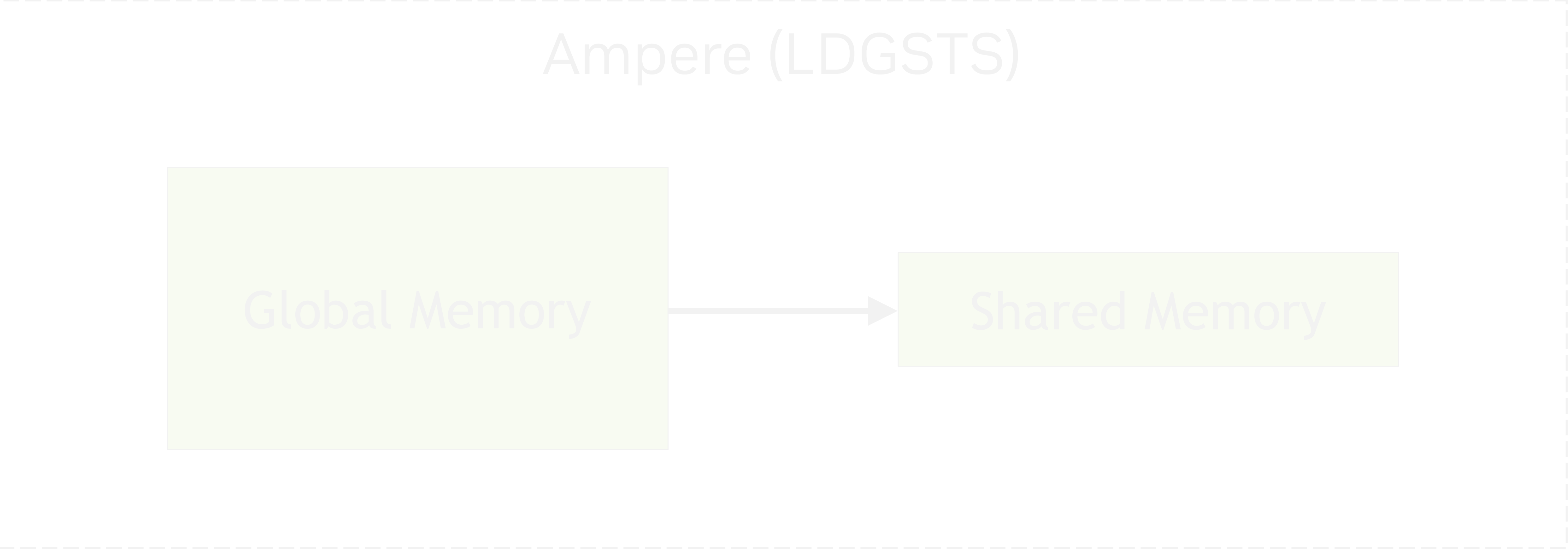
LDGSTS with barrier

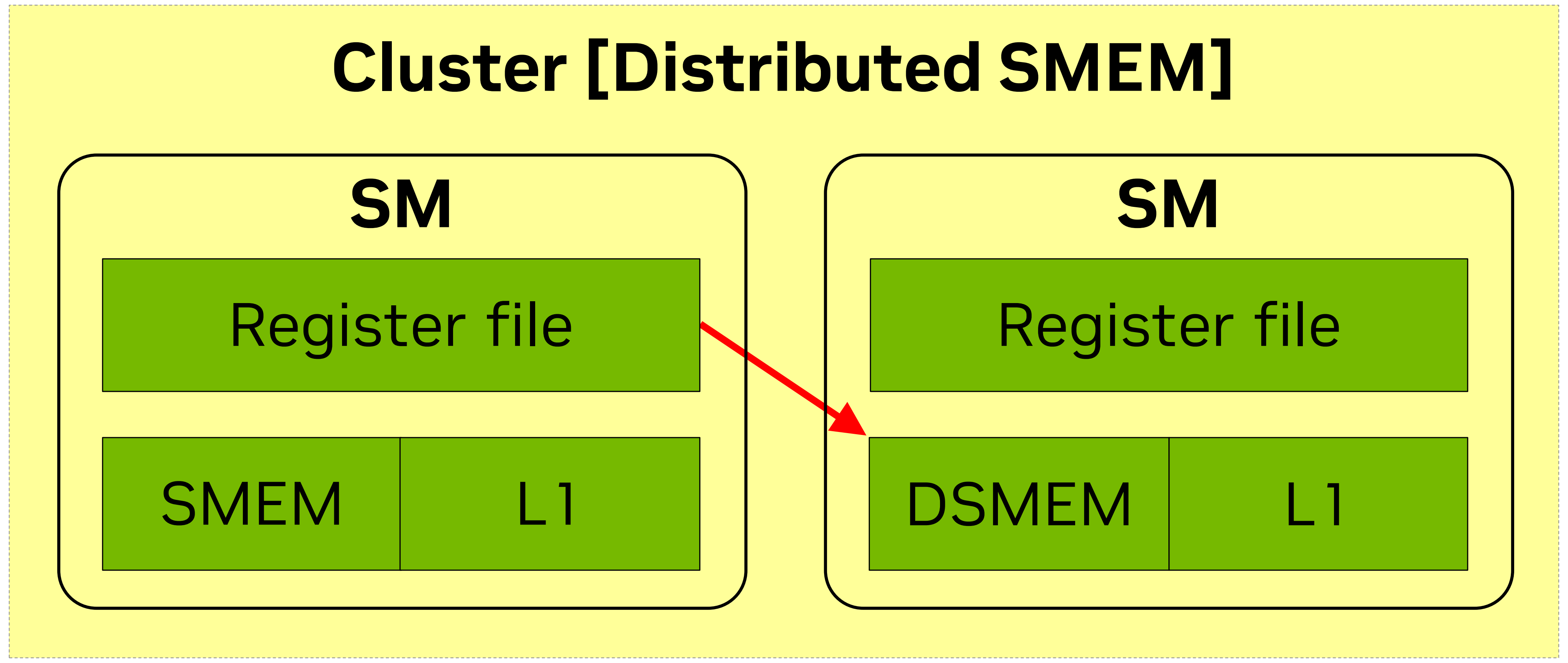
```
        __global__ void kernel (T const* __restrict__ gmem, int n)
        {
            __shared__ T smem[SMEM_SIZE];
            __shared__ __mbarrier_t bar;

            bool is_producer = (threadIdx.x < N_PRODUCERS);

            if (threadIdx.x == 0)
                __mbarrier_init(&bar, N_PRODUCERS); // count := N_PRODUCERS (reset)
            __syncthreads();

            for (int it = 0, phase = 0; it < n; ++it) {

                if (is_producer) {

                    for (int i = threadIdx.x; i < SMEM_SIZE; i += N_PRODUCERS)
                        __pipeline_memcpy_async(&smem[i], &gmem[it * SMEM_SIZE + i], sizeof(T));

In this order! {    __pipeline_arrive_on(&bar);     // count += N_PRODUCERS and count -= N_PRODUCERS by HW on copy completion
                    __mbarrier_arrive(&bar);        // count -= N_PRODUCERS
                } else {
                    while (!__mbarrier_try_wait_parity(&bar, phase, 1000))
                    {}

                    // Use loaded smem here
                }

                phase ^= 1;
                __syncthreads();
            }
        }
```

Asynchronous Data Copies

Asynchronous copies – STAS

HW	Direction	Operation	Synchronization	C++	SASS
Ampere	GMEM ➡ SMEM	Asynchronous version of smem[sind] := gmem[gind]	Thread-local SMEM barrier	__pipeline_memcpy_async	LDGSTS
Hopper	REG ➡ DSMEM	Asynchronous version of dsmem[sind] := var	DSMEM barrier	cuda::ptx::st_async	STAS
TMA	GMEM ➡ (D)SMEM	1D – 5D block copy Uniform	(D)SMEM barrier	cuda::device::experimental:: cp_async_bulk_tensor_Nd_global_to_shared cp_async_bulk_tensor_Nd_shared_to_global	UTMALDG
	SMEM ➡ GMEM		Thread-local		UTMASTG
	GMEM ➡ SMEM	1D block copy Uniform	SMEM barrier	cuda::device::experimental:: cp_async_bulk_global_to_shared cp_async_bulk_shared_to_global	UBLKCP
	SMEM ➡ GMEM		Thread-local		
	SMEM ➡ (D)SMEM		(D)SMEM barrier	cuda::ptx::cp_async_bulk*	





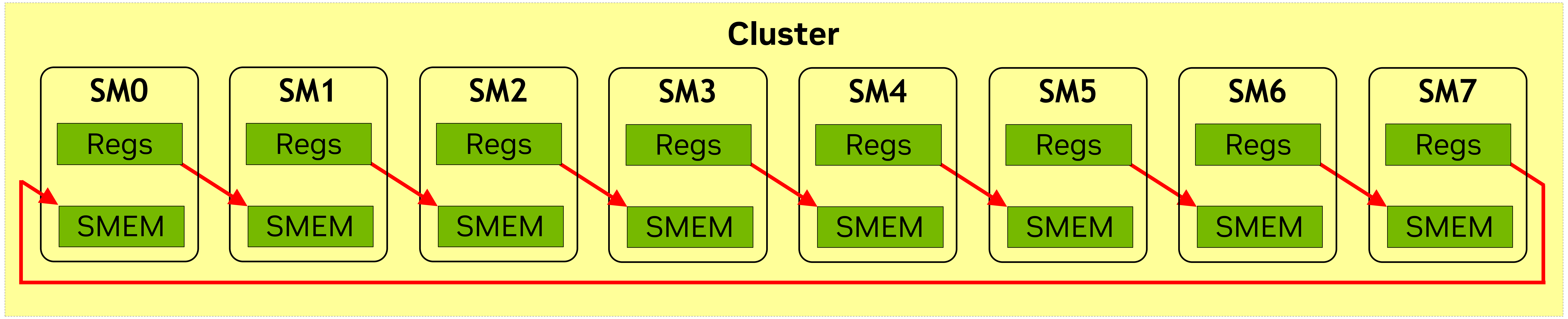
Asynchronous Data Copies

Hopper: cluster asynchronous copies (REG -> DSMEM)

Example (code in next slide):

In a loop:

CTA `#i` copies data to SMEM of CTA `#i+1`, last CTA – to SMEM of CTA `#0`



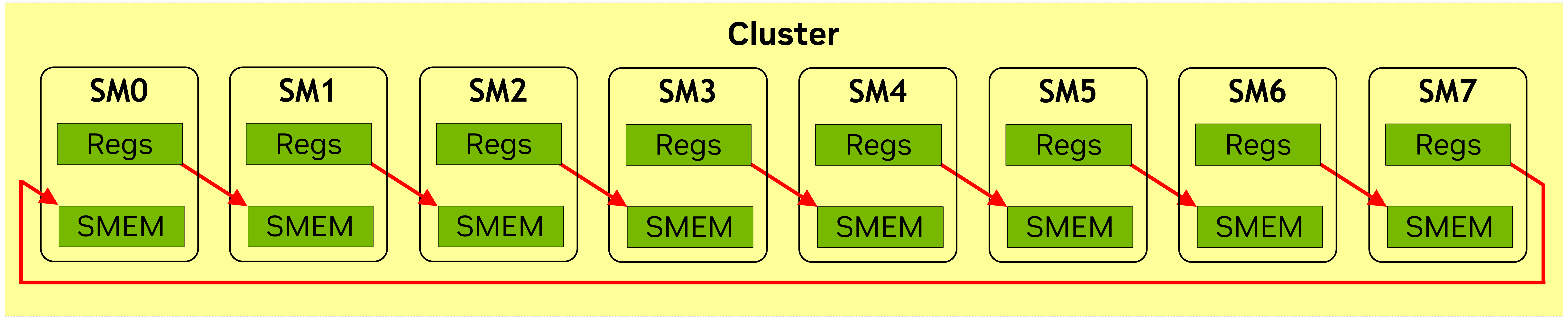
Asynchronous Data Copies

Hopper: cluster asynchronous copies (REG -> DSMEM)

Example (code in next slide):

In a loop:

CTA #i copies data to SMEM of CTA #i+1, last CTA – to SMEM of CTA #0



Two barriers on each CTA:

- `bar` – notifies consumer CTA that data copy from producer CTA is complete
- `bar_done` – notifies producer CTA that consumer CTA is ready to receive data

Asynchronous Data Copies

Hopper: cluster asynchronous copies (REG -> DSMEM)

```
__global__ __cluster_dims__(8, 1, 1)
void kernel ()
{
    using namespace cooperative_groups;
    using namespace cuda::device;
    using namespace cuda::ptx;

    __shared__ int4 smem[1024];

    __shared__ cuda::barrier<thread_scope_block> bar;
    __shared__ cuda::barrier<thread_scope_block> bar_done;

    if (threadIdx.x == 0) {
        init(&bar, 1);
        init(&bar_done, 1024);
    }

    auto cluster = this_cluster();
    cluster.sync();

    int rk = cluster.block_rank();
    int rk_next = (rk + 1) % 8;
    int rk_prev = (rk + 7) % 8; // Same as ‘-1’.

    auto smem_next = cluster.map_shared_rank(smem, rk_next);

    auto bar_next = cluster.map_shared_rank(
        barrier_native_handle(bar), rk_next);

    auto bar_prev = cluster.map_shared_rank(
        barrier_native_handle(bar_done), rk_prev);
```

Asynchronous Data Copies

Hopper: cluster asynchronous copies (REG -> DSMEM)

```
__global__ __cluster_dims__(8, 1, 1)
void kernel ()
{
    using namespace cooperative_groups;
    using namespace cuda::device;
    using namespace cuda::ptx;

    __shared__ int4 smem[1024];

    __shared__ cuda::barrier<thread_scope_block> bar;
    __shared__ cuda::barrier<thread_scope_block> bar_done;

    if (threadIdx.x == 0) {
        init(&bar, 1);
        init(&bar_done, 1024);
    }

    auto cluster = this_cluster();
    cluster.sync();

    int rk = cluster.block_rank();
    int rk_next = (rk + 1) % 8;
    int rk_prev = (rk + 7) % 8; // Same as ‘-1’.

    auto smem_next = cluster.map_shared_rank(smem, rk_next);

    auto bar_next = cluster.map_shared_rank(
        barrier_native_handle(bar), rk_next);

    auto bar_prev = cluster.map_shared_rank(
        barrier_native_handle(bar_done), rk_prev);
```

```
// kernel cont-d:
    for (int it = 0, phase = 0; it < 1000; ++it) {
        int4 r = {rk, rk_next, (int)threadIdx.x, it};

        st_async(&smem_next[1023 - threadIdx.x].x,
            {r.x, r.y, r.z, r.w}, bar_next);

        if (threadIdx.x == 0)
            mbarrier_arrive_expect_tx(
                sem_release, scope_cluster, space_shared,
                barrier_native_handle(bar), sizeof(smem));

        while (!mbarrier_try_wait_parity(
            barrier_native_handle(bar), phase, 1000))
        {}

        r = smem[threadIdx.x]; // done reading

        mbarrier_arrive(sem_release, scope_cluster, space_cluster,
            bar_prev);

        while (!mbarrier_try_wait_parity(
            barrier_native_handle(bar_done), phase, 1000))
        {}

        phase ^= 1;
    }
} // void kernel ()
```

Any order

tx_count -= 16 by each of 1024 threads **on completion**

tx_count += 16384

phase flip when tx_count gets to 0

Asynchronous Data Copies

Hopper: cluster asynchronous copies (REG -> DSMEM)

```
__global__ __cluster_dims__(8, 1, 1)
void kernel ()
{
  using namespace cooperative_groups;
  using namespace cuda::device;
  using namespace cuda::ptx;

  __shared__ int4 smem[1024];

  __shared__ cuda::barrier<thread_scope_block> bar;
  __shared__ cuda::barrier<thread_scope_block> bar_done;

  if (threadIdx.x == 0) {
    init(&bar, 1);
    init(&bar_done, 1024);
  }

  auto cluster = this_cluster();
  cluster.sync();

  int rk = cluster.block_rank();
  int rk_next = (rk + 1) % 8;
  int rk_prev = (rk + 7) % 8; // Same as ‘-1’.

  auto smem_next = cluster.map_shared_rank(smem, rk_next);

  auto bar_next = cluster.map_shared_rank(
    barrier_native_handle(bar), rk_next);

  auto bar_prev = cluster.map_shared_rank(
    barrier_native_handle(bar_done), rk_prev);
```

```
// kernel cont-d:
  for (int it = 0, phase = 0; it < 1000; ++it) {
    int4 r = {rk, rk_next, (int)threadIdx.x, it};

    st_async(&smem_next[1023 - threadIdx.x].x,
      {r.x, r.y, r.z, r.w}, bar_next);

    if (threadIdx.x == 0)
      mbarrier_arrive_expect_tx(
        sem_release, scope_cluster, space_shared,
        barrier_native_handle(bar), sizeof(smem));

    while (!mbarrier_try_wait_parity(
      barrier_native_handle(bar), phase, 1000))
    {}

    r = smem[threadIdx.x]; // done reading

    mbarrier_arrive(sem_release, scope_cluster, space_cluster,
      bar_prev);

    while (!mbarrier_try_wait_parity(
      barrier_native_handle(bar_done), phase, 1000))
    {}

    phase ^= 1;
  }
} // void kernel ()
```

Any order

tx_count -= 16 by each of 1024 threads **on completion**

tx_count += 16384

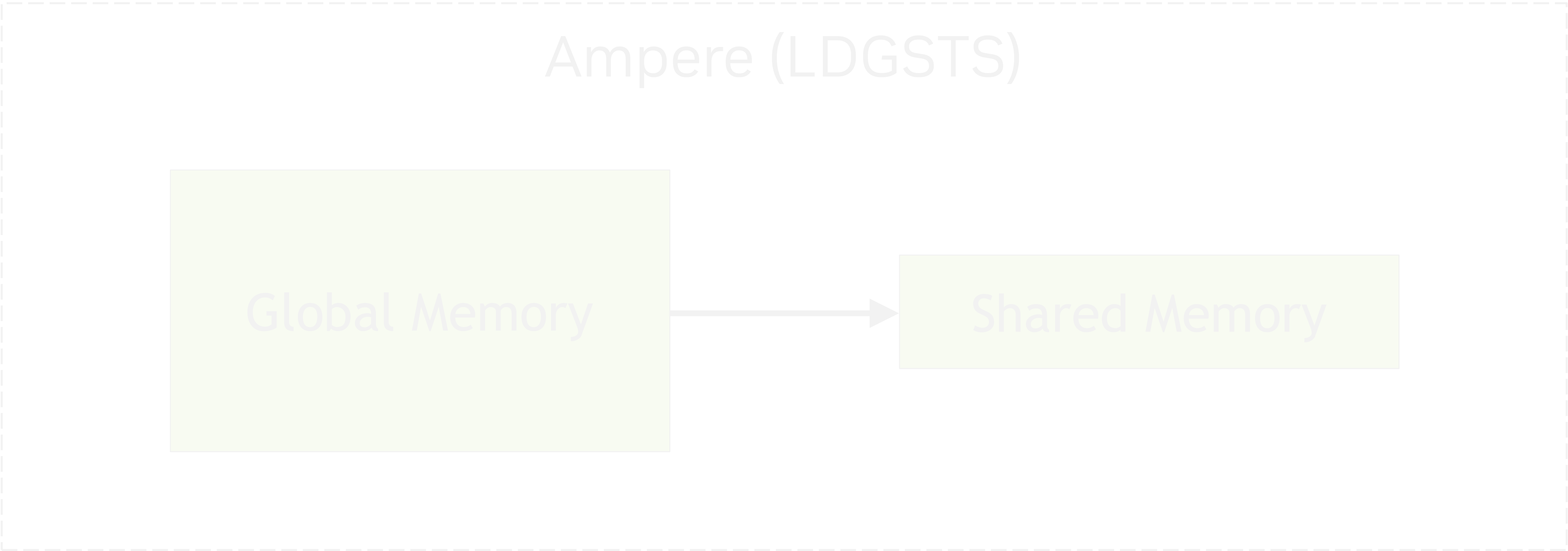
phase flip when tx_count gets to 0

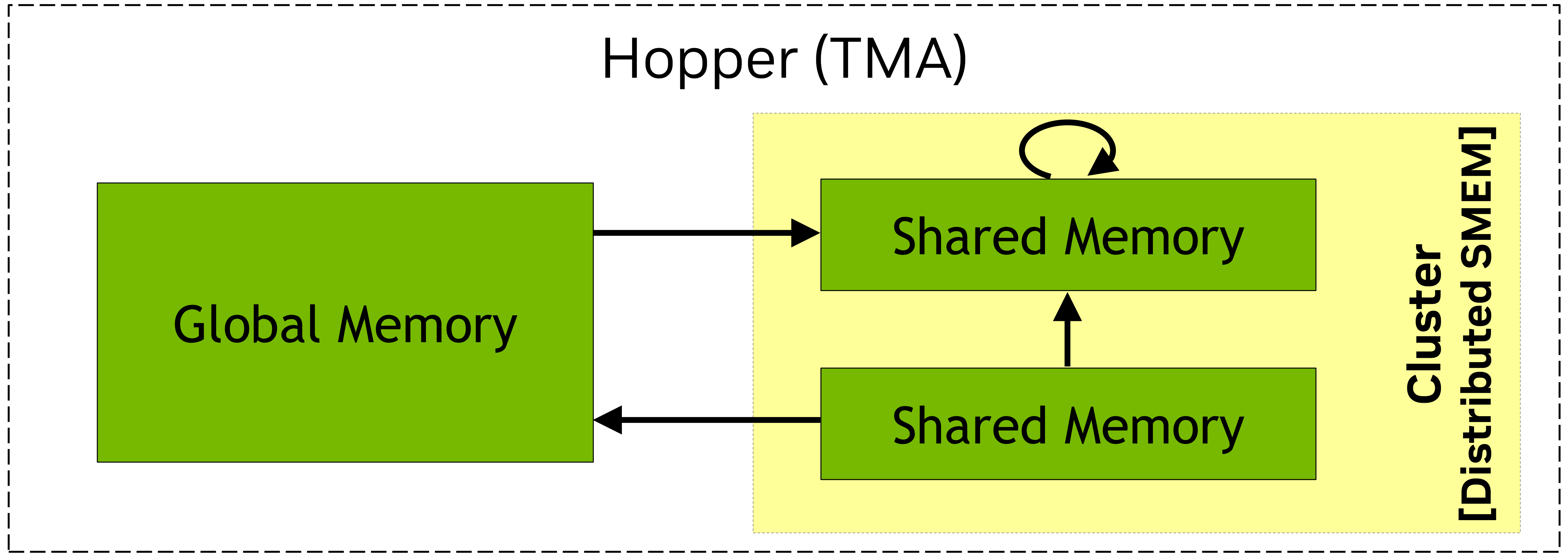
Space matches barrier location!

Asynchronous Data Copies

Asynchronous copies – UTMA*

HW	Direction	Operation	Synchronization	C++	SASS
Ampere	GMEM ➡ SMEM	Asynchronous version of smem[sind] := gmem[gind]	Thread-local SMEM barrier	__pipeline_memcpy_async	LDGSTS
Hopper	REG ➡ DSMEM	Asynchronous version of dsmem[sind] := var	DSMEM barrier	cuda::ptx::st_async	STAS
TMA	GMEM ➡ (D)SMEM	1D – 5D block copy Uniform	(D)SMEM barrier	cuda::device::experimental:: cp_async_bulk_tensor_**N**d_global_to_shared	UTMALDG
	SMEM ➡ GMEM		Thread-local	cp_async_bulk_tensor_**N**d_shared_to_global	UTMASTG
	GMEM ➡ SMEM	1D block copy Uniform	SMEM barrier	cuda::device::experimental:: cp_async_bulk_global_to_shared	UBLKCP
	SMEM ➡ GMEM		Thread-local	cp_async_bulk_shared_to_global	
	SMEM ➡ (D)SMEM		(D)SMEM barrier	cuda::ptx::cp_async_bulk*	


Ampere (LDGSTS)


Hopper (TMA)

Asynchronous Data Copies

TMA nD – properties

Tensor copies functionality (not comprehensive list)**:**

- Single thread can fire 1D-5D block copy
- Destination modes:
  - Tile – source layout preserved
  - Im2col – source copy-box elements rearranged into columns
- Out-of-bounds fill modes:
  - Zeros
  - NaNs
- SMEM swizzling

Constraints (not comprehensive list)**:**

- SMEM address must be 128B aligned
- GMEM address must be 16B aligned
- Strides must be multiples of 16B
- Copy-box fast-dimension size must be multiple of 16B
- Copy-box start address must be 16B aligned

Tensor copy geometry is described by a `CUtensorMap` descriptor:

- Can be a `__grid_constant__` kernel parameter or `__constant__` object (64B aligned)

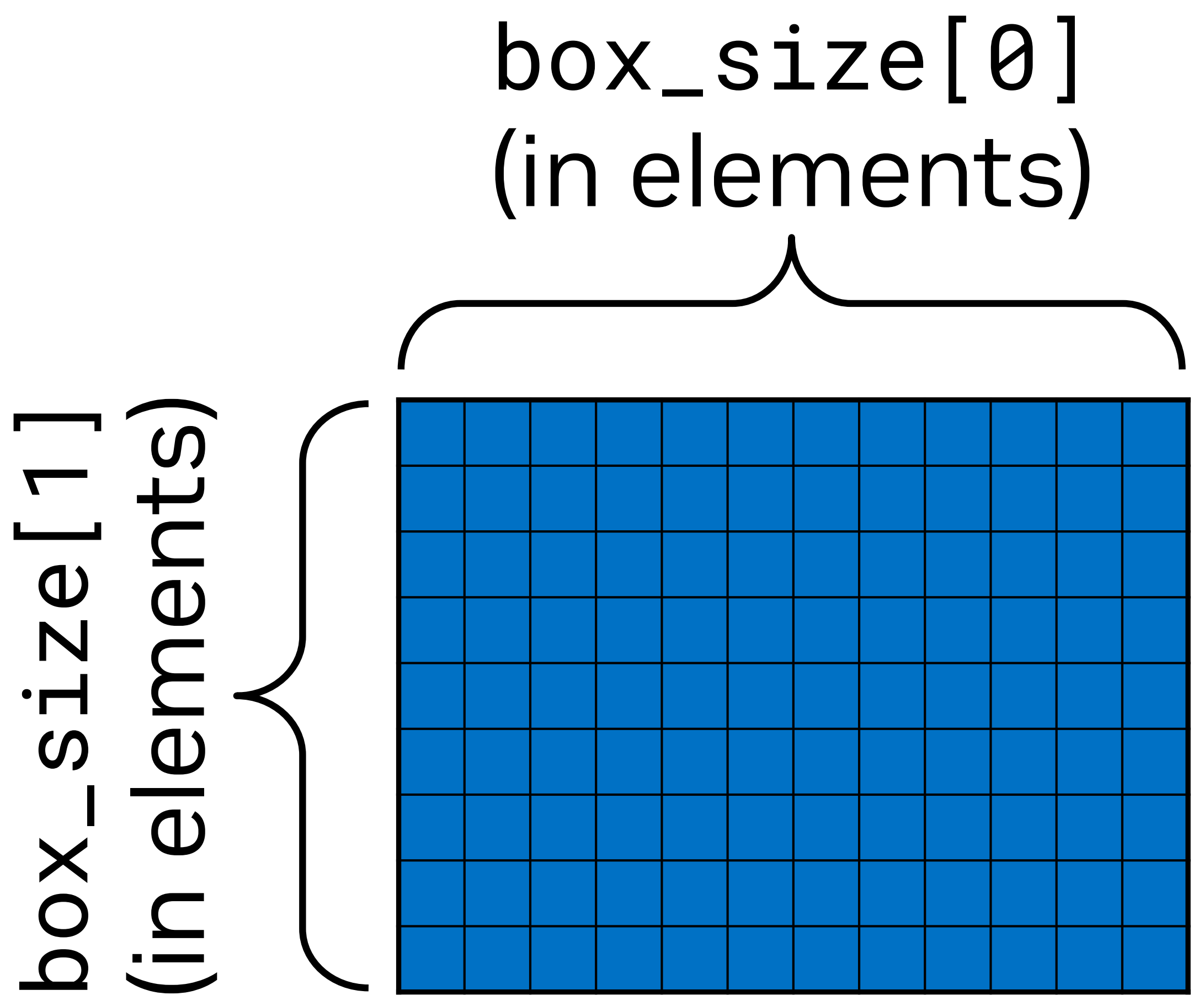
Asynchronous Data Copies

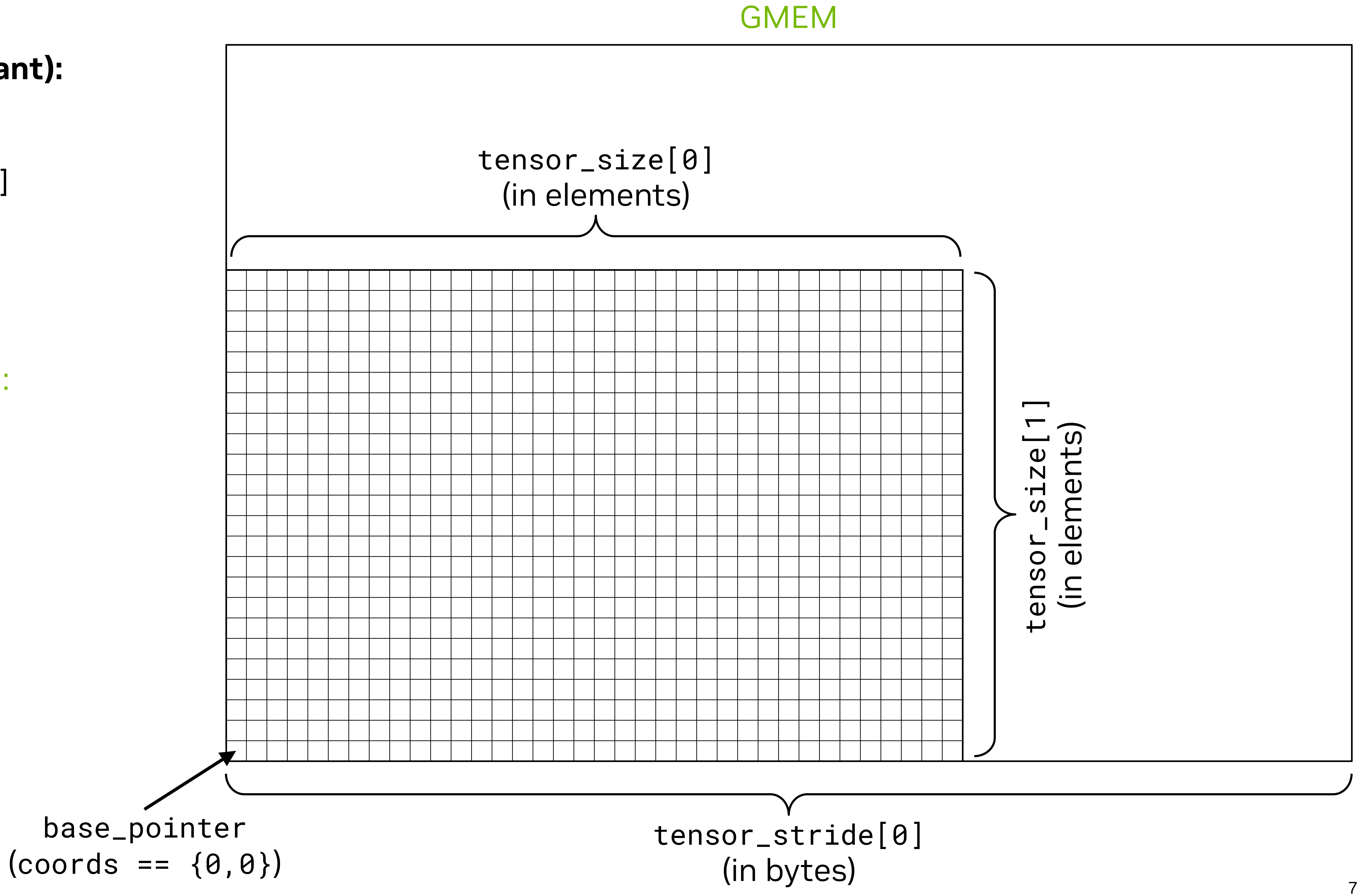
TMA nD – tensor descriptor

Tensor descriptor (constant):

- nD (1, 2, 3, 4, or 5)
- base_pointer (GMEM)
- tensor_stride[nD – 1]
- tensor_size[nD]
- box_size[nD]
- element_stride[nD]

Box copied to SMEM/DSMEM:





Asynchronous Data Copies

TMA nD (**GMEM->SMEM**) – tensor descriptor

Instruction args (variable):

- `dst_pointer` (SMEM/DSMEM)
- `coords[nD]`

- Tensor descriptor
- Synchronization barrier

Box copied to SMEM/DSMEM:

GMEM
tensor_size[0]
(in elements)
tensor_size[1]
(in elements)
tensor_stride[0]
(in bytes)
base_pointer
(coords == {0,0})
coords[nD]
(in elements)
copy
box_size[0]
(in elements)
box_size[1]
(in elements)
dst_pointer

Asynchronous Data Copies

TMA nD (**GMEM->SMEM**) – tensor descriptor

Instruction args (variable):

- `dst_pointer` (SMEM/DSMEM)
- `coords[nD]`

- Tensor descriptor
- Synchronization barrier

Box copied to SMEM/DSMEM:

GMEM
tensor_size[0]
(in elements)
tensor_size[1]
(in elements)
tensor_stride[0]
(in bytes)
Zero/Nan fill
copy
box_size[0]
(in elements)
box_size[1]
(in elements)
dst_pointer
coords[nD]
(in elements)
base_pointer
(coords == {0,0})

Asynchronous Data Copies

TMA nD (**GMEM->SMEM**) – tensor descriptor

Tensor descriptor (constant):

- nD
- base_pointer (GMEM)
- tensor_stride[nD – 1]
- tensor_size[nD]
- box_size[nD]
- element_stride[nD]:={**1**,2}

must be 1

Box copied to SMEM/DSMEM:

GMEM
tensor_size[0]
(in elements)
tensor_size[1]
(in elements)
tensor_stride[0]
(in bytes)
base_pointer
(coords == {0,0})
coords[nD]
(in elements)
copy
box_size[0]
(in elements)
box_size[1]
(in elements)
dst_pointer

Asynchronous Data Copies

TMA nD (**SMEM->GMEM**) – tensor descriptor

Instruction args (variable):

- `src_pointer` (SMEM/~~DSMEM~~)
- `coords[nD]`

- Tensor descriptor
- ~~Synchronization barrier~~
  -> thread-local sync

GMEM
Box copied from SMEM:
box_size[0]
(in elements)
box_size[0]
(in elements)
src_pointer
copy
tensor_size[0]
(in elements)
tensor_size[1]
(in elements)
coords[nD]
(in elements)
base_pointer
(coords == {0,0})
tensor_stride[0]
(in bytes)

Asynchronous Data Copies

TMA nD (**SMEM->GMEM**) – tensor descriptor

Instruction args (variable):

- `src_pointer` (SMEM/~~DSMEM~~)
- `coords[nD]`

- Tensor descriptor
- ~~Synchronization barrier~~
  -> thread-local sync

Box copied from SMEM:
box_size[0]
(in elements)
box_size[0]
(in elements)
src_pointer
copy
GMEM
tensor_size[0]
(in elements)
Unlike GMEM->SMEM,
coords must be >= 0
tensor_size[1]
(in elements)
base_pointer
(coords == {0,0})
tensor_stride[0]
(in bytes)

Asynchronous Data Copies

TMA nD vs LDGSTS

Example:

×-shaped 16-th order FD stencil.
Thread block size: 32×8, shared memory size: 48×24 (32×8 center + 8-point halo on each side)

LDGSTS (pseudocode):

```
int tx = threadIdx.x;
int ty = threadIdx.y;

int ix = blockIdx.x * 32 + tx;
int iy = blockIdx.y * 32 + ty;

if (tx < 8 && ty < 8)
    async_load_0(offset0);

if (tx < 8 && ty < 8 && iy < ny+8)
    async_load_1(offset1);
...

if (ix < nx && iy < ny)
    async_load_4(offset2);
...

commit_async_loads();
```

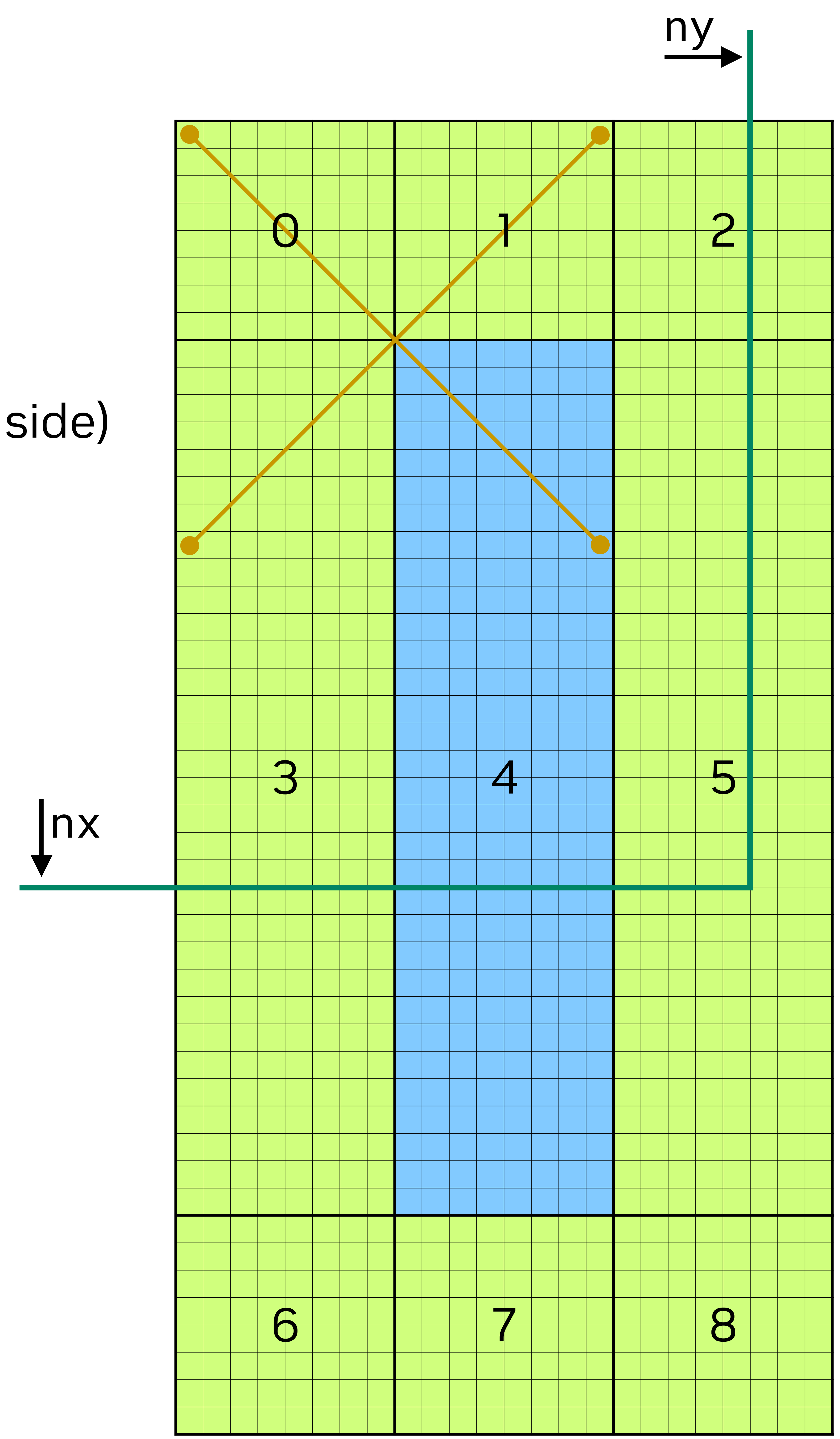
Multiple (up to 9) conditional loads by each thread with different offset indices:

- MIO pressure
- Register pressure
- Predicate pressure

TMA (pseudocode):

```
if (tid == 0)
    async_load_tma(...);
```

Single-thread, unconditional (wrt. domain bounds) load



Asynchronous Data Copies

TMA nD vs LDGSTS

Example:

×-shaped 16-th order FD stencil.
Thread block size: 32×8, shared memory size: 48×24 (32×8 center + 8-point halo on each side)

LDGSTS (pseudocode):

```
int tx = threadIdx.x;
int ty = threadIdx.y;

int ix = blockIdx.x * 32 + tx;
int iy = blockIdx.y * 32 + ty;

if (tx < 8 && ty < 8)
    async_load_0(offset0);

if (tx < 8 && ty < 8 && iy < ny+8)
    async_load_1(offset1);
...

if (ix < nx && iy < ny)
    async_load_4(offset2);
...

commit_async_loads();
```

Multiple (up to 9) conditional loads by each thread with different offset indices:

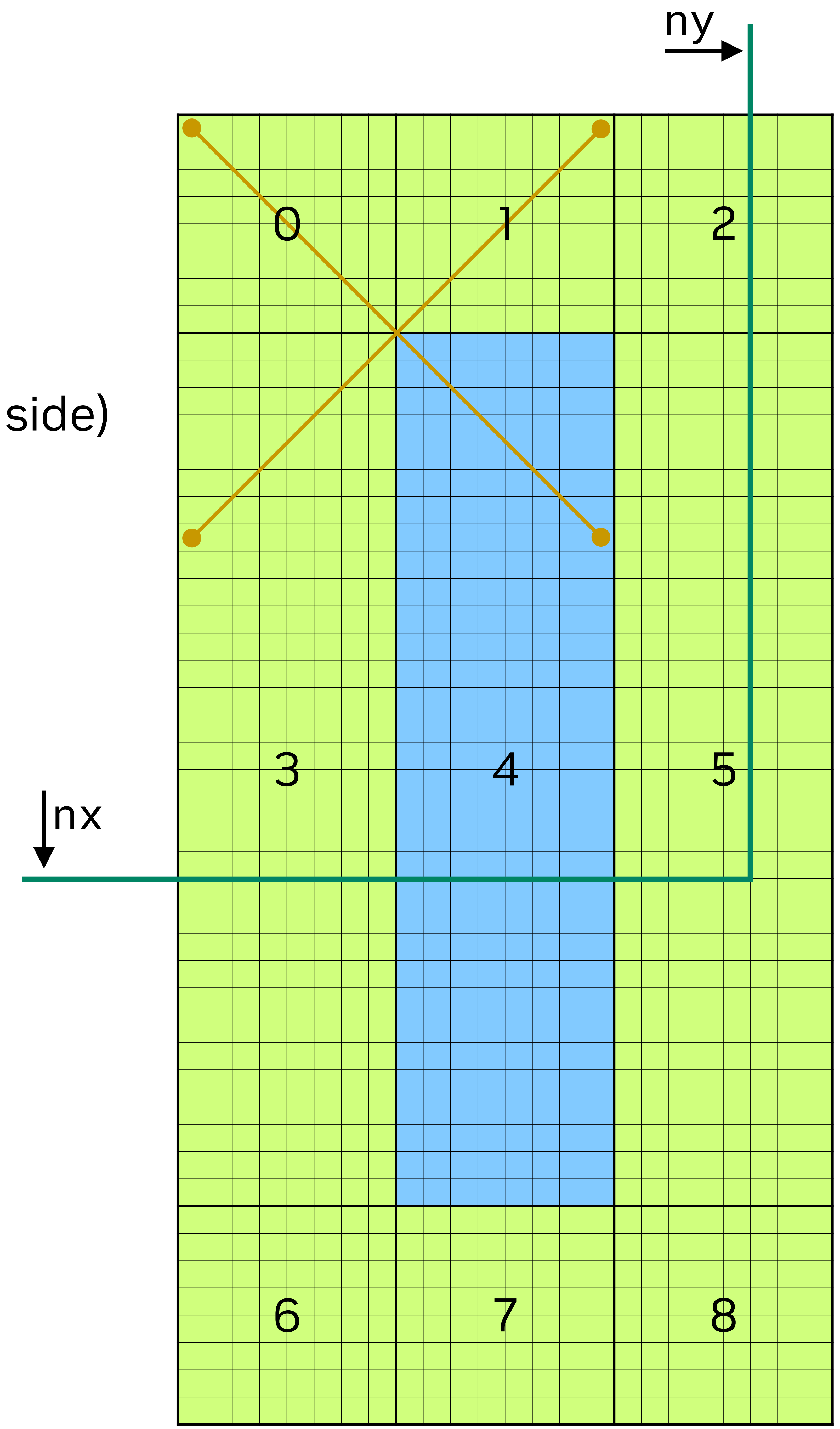
- MIO pressure
- Register pressure
- Predicate pressure

Even worse for higher order stencil.
E.g., 12-point halos: regions #3 and #5 are not covered by TB and require 2 loads per region!

TMA (pseudocode):

```
if (tid == 0)
    async_load_tma(...);
```

Single-thread, unconditional (wrt. to domain bounds) load



Asynchronous Data Copies

TMA details

TMA programming model is Uniform:

all active threads in a warp are executing TMA operation “with same arguments”.

Code:

```
async_load_tma(args_per_thread);

                ⬇

Loop over active thread i in {0,...,31}
{
    Issue_TMA_Instruction(args_of_thread #i);
}
```

Asynchronous Data Copies

TMA details

TMA programming model is Uniform:

all active threads in a warp are executing TMA operation “with same arguments”.

Code:

```
async_load_tma(args_per_thread);
```

⬇

```
Loop over active thread i in {0,...,31}
{
    Issue_TMA_Instruction(args_of_thread #i);
}
```

Erroneous code (race):

```
async_load_tma(single_thread_args);
```

Not optimal code:

```
if (tid == 0)
    async_load_tma(single_thread_args);
```

Compiler doesn’t know that `tid == 0` for only one thread!

Compiler still creates a peeling loop over the active threads:

```
Loop over active thread i in {0,...,31}
{
    if (tid(i) == 0) // uniform condition
        Issue_TMA_Instruction(single_thread_args);
}
```

Asynchronous Data Copies

TMA details

TMA programming model is Uniform:

all active threads in a warp are executing TMA operation “with same arguments”.

Code:

```
async_load_tma(args_per_thread);

                  ⬇

Loop over active thread i in {0,...,31}
{
    Issue_TMA_Instruction(args_of_thread #i);
}
```

Erroneous code (race):

```
async_load_tma(single_thread_args);
```

Not optimal code:

```
if (tid == 0)
    async_load_tma(single_thread_args);
```

Compiler doesn’t know that `tid == 0` for only one thread!

Compiler still creates a peeling loop over the active threads:

```
Loop over active thread i in {0,...,31}
{
    if (tid(i) == 0) // uniform condition
        Issue_TMA_Instruction(single_thread_args);
}
```

Better code:

```
if (tid == 0)
    cg::invoke_one(cg::coalesced_threads(),
        async_load_tma(single_thread_args));
```

Asynchronous Data Copies

TMA nD main steps summary

GMEM->SMEM:

INIT

```
if (threadIdx.x == 0) {
    __mbarrier_init(&bar, 1);
    cuda::device::experimental::fence_proxy_async_shared_cta();
}
```

```
__syncthreads(); // cluster.sync() if GMEM->DSMEM
```

FIRE

```
if (threadIdx.x == 0) {
    invoke_one(coalesced_threads(), cp_async_bulk_tensor_2d_global_to_shared, smem, &map,
                coord[0], coord[1], bar); // mapped_remote_bar if GMEM->DSMEM
    mbarrier_arrive_expect_tx(sem_release, scope_cta, space_shared, &bar, SMEM_SIZE);
}
```

WAIT FOR COMPLETION

```
while (!mbarrier_try_wait_parity(&bar, phase, 1000))
{}
```

Asynchronous Data Copies

TMA nD main steps summary

SMEM->GMEM:

```
                 __syncthreads(); //  After all threads use SMEM

                 if (threadIdx.x == 0) {
                     invoke_one(coalesced_threads(),
          FIRE {         [] () { cp_async_bulk_tensor_2d_shared_to_global(&map, coordA[0], coordA[1], smemA);
                                 cp_async_bulk_tensor_2d_shared_to_global(&map, coordB[0], coordB[1], smemB); }
                     );
  COMMIT GROUP {     cp_async_bulk_commit_group();

          FIRE {     invoke_one(coalesced_threads(), cp_async_bulk_tensor_2d_shared_to_global,
                                &map, coordD[0], coordD[1], smemD);
  COMMIT GROUP {     cp_async_bulk_commit_group();

                     cp_async_bulk_wait_group_read<N>(); } WAIT FOR COMPLETION EXCEPT LAST  N  GROUPS
                 }

                 __syncthreads(); //  For all threads to use SMEM again
```

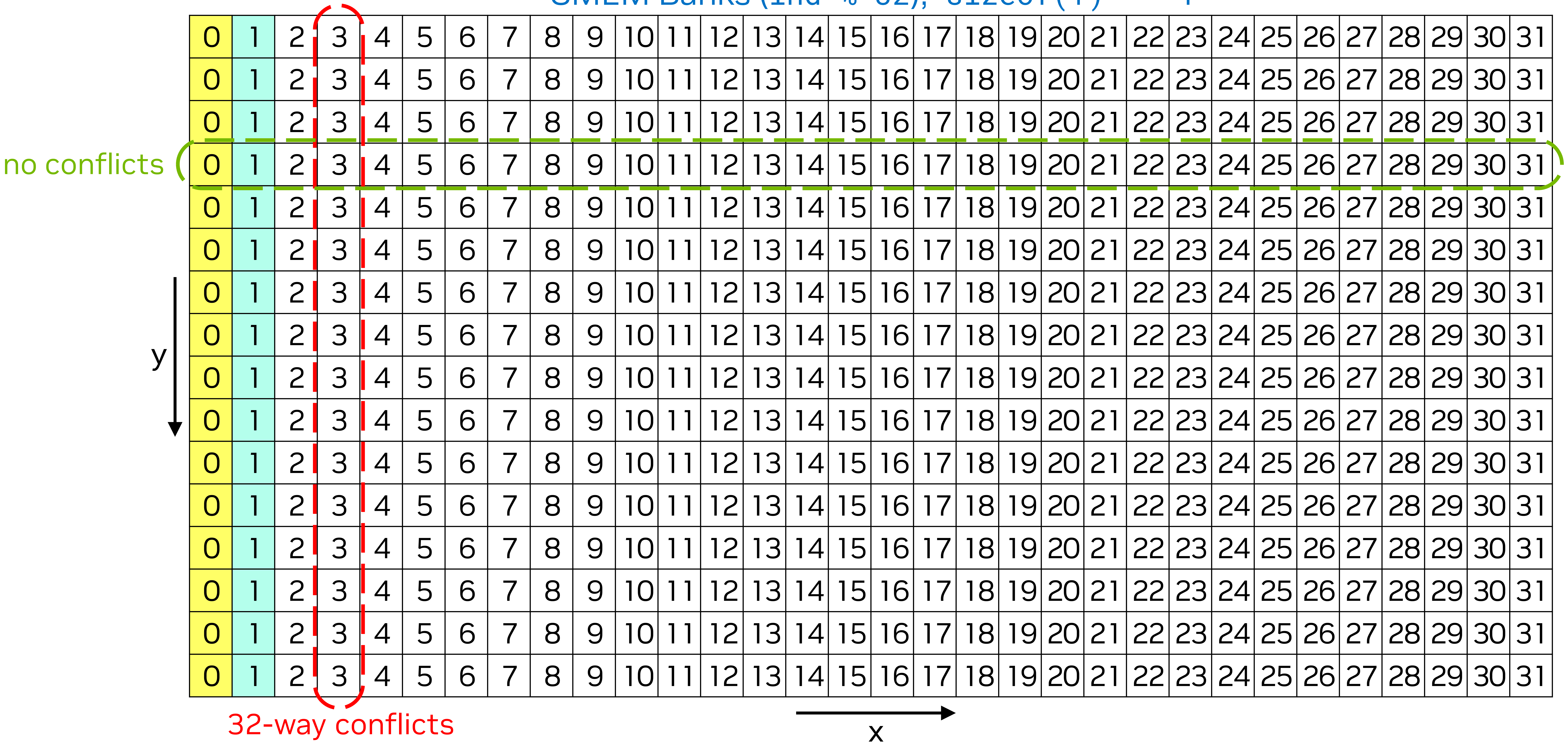
Asynchronous Data Copies

SMEM bank conflicts

Given 2D `array[][NX]`, index `[y][x]` maps to linear index `ind` via:

- **Conventional:** `ind := y * NX + x`

SMEM Banks (`ind % 32`), `sizeof(T) == 4`



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

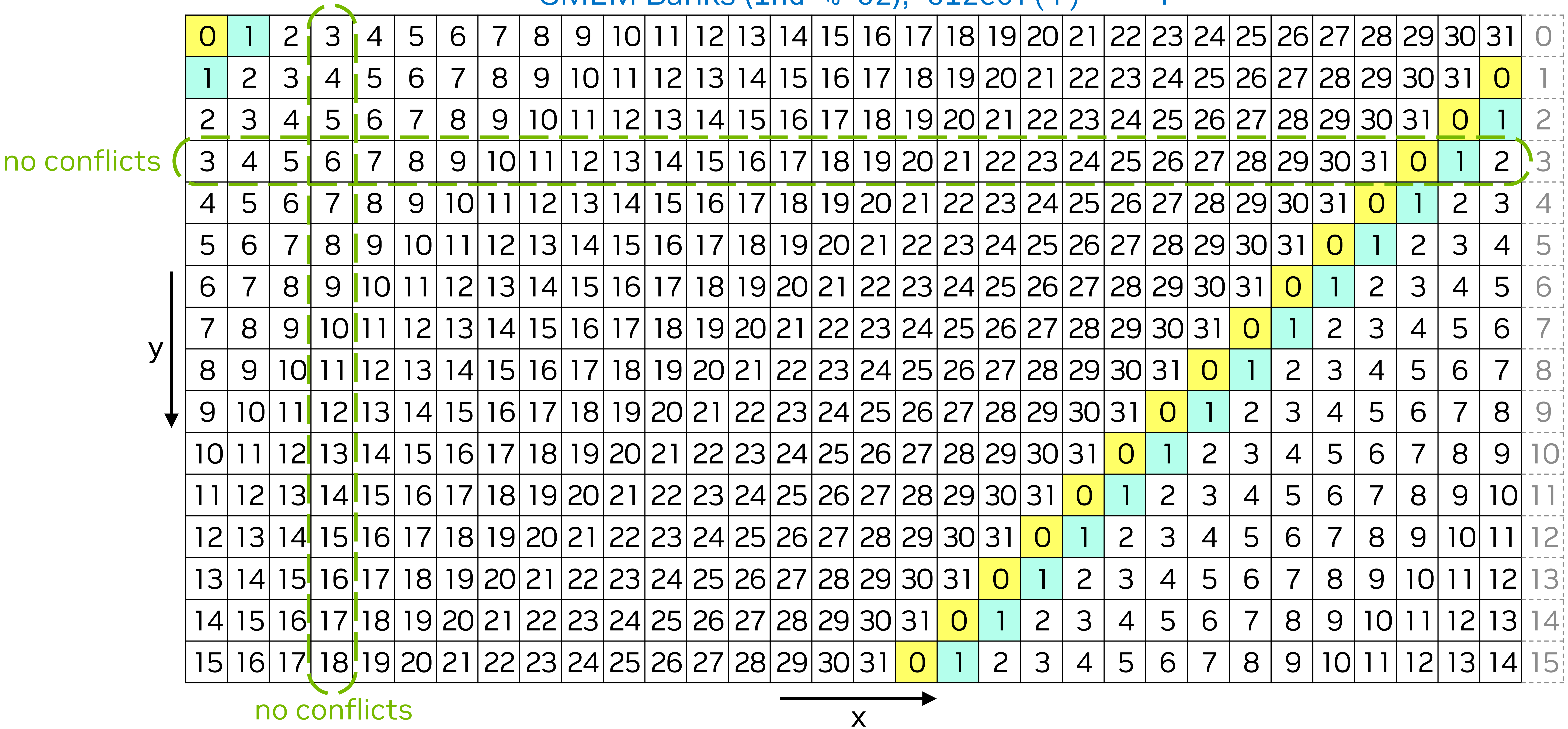
Asynchronous Data Copies

SMEM bank ~~conflicts~~ – padding

Given 2D `array[][~~NX~~ NX+1]`, index `[y][x]` maps to linear index `ind` via:

- **Conventional:** `ind := y * ~~NX~~ (NX+1) + x`

SMEM Banks (`ind % 32`), `sizeof(T) == 4`



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2
3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4
5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5
6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6
7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	13
14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Asynchronous Data Copies

SMEM bank ~~conflicts~~ – swizzling

Given 2D `array[][NX]`, index `[y][x]` maps to linear index `ind` via:

- **Conventional:** `ind := y * NX + x`
- **Swizzling:** `ind := y * NX + (y ^ x)`

SMEM Banks (`ind % 32`), `sizeof(T) == 4`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1	0	3	2	5	4	7	6	9	8	11	10	13	12	15	14	17	16	19	18	21	20	23	22	25	24	27	26	29	28	31	30
2	3	0	1	6	7	4	5	10	11	8	9	14	15	12	13	18	19	16	17	22	23	20	21	26	27	24	25	30	31	28	29
3	2	1	0	7	6	5	4	11	10	9	8	15	14	13	12	19	18	17	16	23	22	21	20	27	26	25	24	31	30	29	28
4	5	6	7	0	1	2	3	12	13	14	15	8	9	10	11	20	21	22	23	16	17	18	19	28	29	30	31	24	25	26	27
5	4	7	6	1	0	3	2	13	12	15	14	9	8	11	10	21	20	23	22	17	16	19	18	29	28	31	30	25	24	27	26
6	7	4	5	2	3	0	1	14	15	12	13	10	11	8	9	22	23	20	21	18	19	16	17	30	31	28	29	26	27	24	25
7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	23	22	21	20	19	18	17	16	31	30	29	28	27	26	25	24
8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	24	25	26	27	28	29	30	31	16	17	18	19	20	21	22	23
9	8	11	10	13	12	15	14	1	0	3	2	5	4	7	6	25	24	27	26	29	28	31	30	17	16	19	18	21	20	23	22
10	11	8	9	14	15	12	13	2	3	0	1	6	7	4	5	26	27	24	25	30	31	28	29	18	19	16	17	22	23	20	21
11	10	9	8	15	14	13	12	3	2	1	0	7	6	5	4	27	26	25	24	31	30	29	28	19	18	17	16	23	22	21	20
12	13	14	15	8	9	10	11	4	5	6	7	0	1	2	3	28	29	30	31	24	25	26	27	20	21	22	23	16	17	18	19
13	12	15	14	9	8	11	10	5	4	7	6	1	0	3	2	29	28	31	30	25	24	27	26	21	20	23	22	17	16	19	18
14	15	12	13	10	11	8	9	6	7	4	5	2	3	0	1	30	31	28	29	26	27	24	25	22	23	20	21	18	19	16	17
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

no conflicts

y

x

no conflicts

Asynchronous Data Copies

TMA nD – SMEM swizzling

TMA swizzles eight 16-byte chunks within 128-byte segments.

Constraints:

- `NX * sizeof(T) == SWIZZLE_SIZE` (where: `T array[][NX]`)
- Allowed values of `SWIZZLE_SIZE`: 32, 64, 128

Given `[y][x]` index in `T array[][NX]`:

1. Compute index of 16-byte chunk within 128-byte segment:

```
i16 := (y * NX + x) * sizeof(T) / 16
y16 := i16 / 8
x16 := i16 % 8
```

2. Compute swizzled index of 16-byte chunk:

```
x16_swz := y16 ^ x16
```

3. Compute swizzled index:

```
x_swz := x16_swz * 16 / sizeof(T) % NX + x % (16 / sizeof(T))
```

Swizzled indices of 16-byte chunks		
32-byte	64-byte	128-byte
0 1	0 1 2 3	0 1 2 3 4 5 6 7
0 1	0 1 2 3	1 0 3 2 5 4 7 6
0 1	1 0 3 2	2 3 0 1 6 7 4 5
0 1	1 0 3 2	3 2 1 0 7 6 5 4
1 0	2 3 0 1	4 5 6 7 0 1 2 3
1 0	2 3 0 1	5 4 7 6 1 0 3 2
1 0	3 2 1 0	6 7 4 5 2 3 0 1
1 0	3 2 1 0	7 6 5 4 3 2 1 0

Asynchronous Data Copies

Segmented sort performance

4194304 sequences of 128 integers (~ 2GB)
Pair-wise sorting network algorithm

H200 experiment

SMEM copies	Time (ms)	BW (GB/s)	% SOL
LDG+STS	1.88	2274	50
LDGSTS	1.66	2600	56
UBLKCP	1.18	3632	79
TMA + swizzle	1.07	4066	87

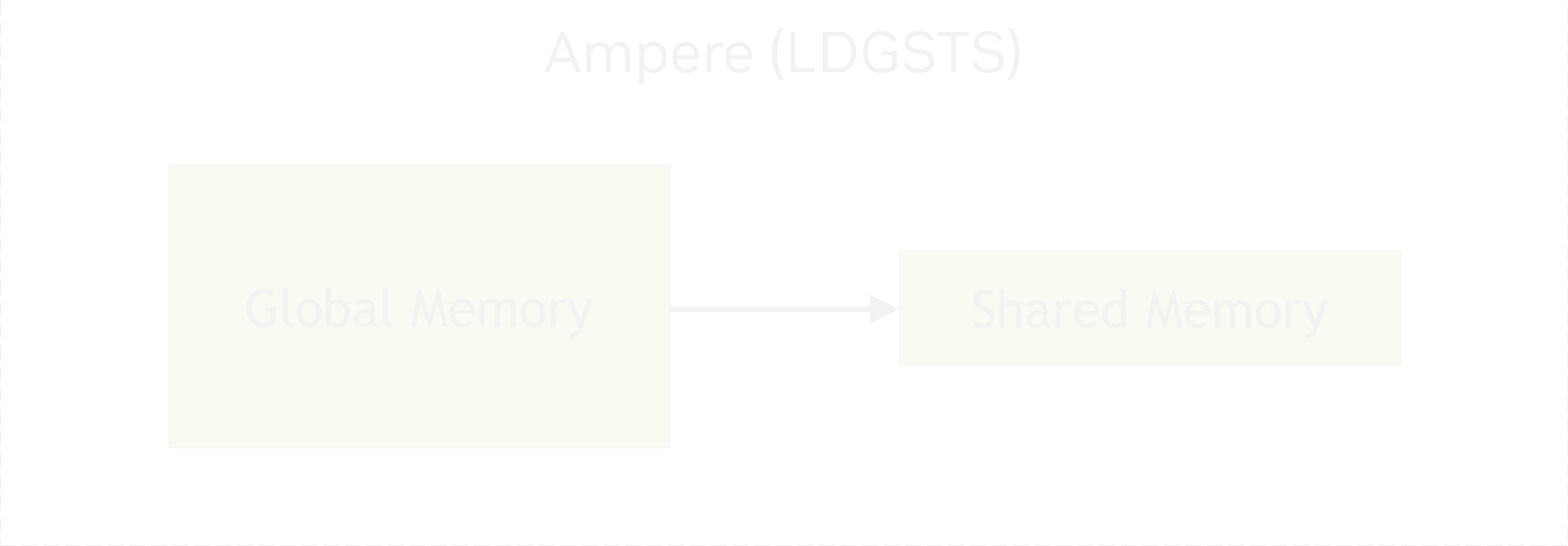
[Test provided by Allard Hendriksen]

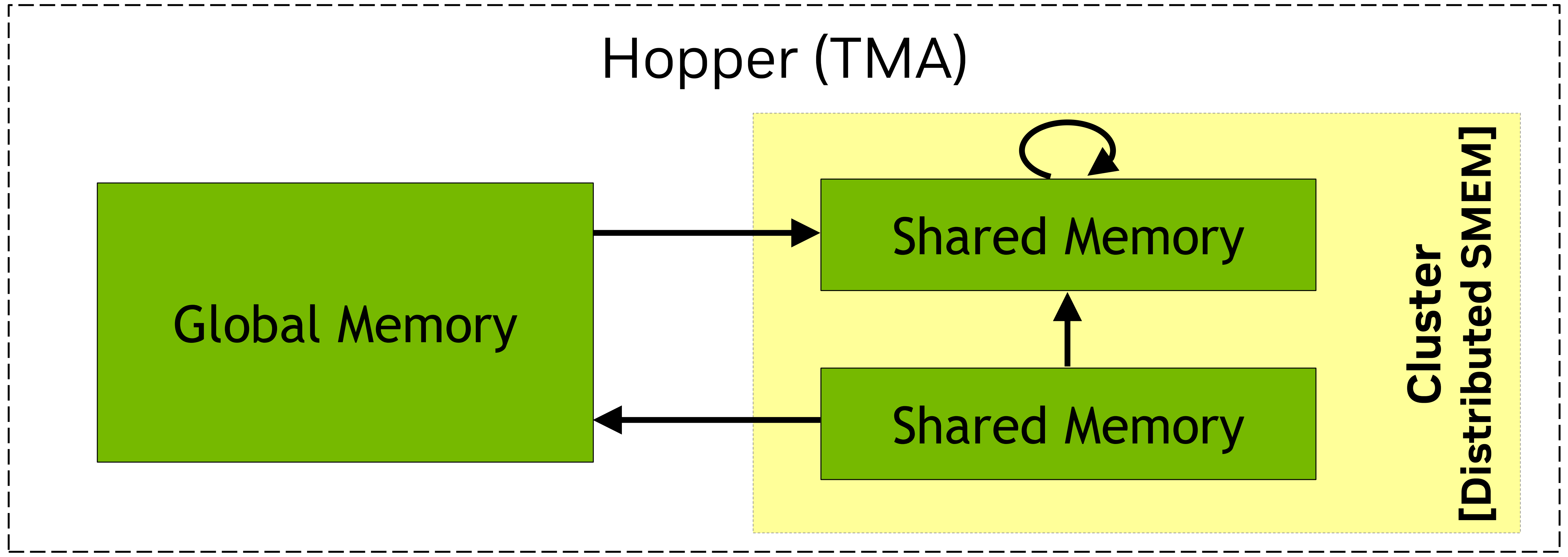
Asynchronous Data Copies

Asynchronous copies – UBLKCP

HW	Direction	Operation	Synchronization	C++	SASS
Ampere	GMEM ➡ SMEM	Asynchronous version of smem[sind] := gmem[gind]	Thread-local SMEM barrier	__pipeline_memcpy_async	LDGSTS
Hopper	REG ➡ DSMEM	Asynchronous version of dsmem[sind] := var	DSMEM barrier	cuda::ptx::st_async	STAS
TMA	GMEM ➡ (D)SMEM	1D – 5D block copy Uniform	(D)SMEM barrier	cuda::device::experimental:: cp_async_bulk_tensor_Nd_global_to_shared	UTMALDG
	SMEM ➡ GMEM		Thread-local	cp_async_bulk_tensor_Nd_shared_to_global	UTMASTG
	GMEM ➡ SMEM	1D block copy Uniform	SMEM barrier	cuda::device::experimental:: cp_async_bulk_global_to_shared	UBLKCP
	SMEM ➡ GMEM		Thread-local	cp_async_bulk_shared_to_global	
	SMEM ➡ (D)SMEM		(D)SMEM barrier	cuda::ptx::cp_async_bulk*	

* Available in latest github CCCL





Asynchronous Data Copies

TMA – properties

UBLKCP ~ async version of `memcpy`:

- Fired by a single thread
- Uses `src`/`dst` pointers and copy `size`
  (no Tensor descriptors => no bound-box, stride, fill, swizzle, etc. functionality)
- Alignment requirements:
  - 16B alignment of GMEM/SMEM
  - Copy size multiple of 16B
- Same synchronization (barrier or thread-local) as for TMA nD:
  - `mbarrier_arrive_expect_tx(sem_release, scope_cta, space_shared, &bar, SIZE);`
  - `cp_async_bulk_commit_group();`
    `cp_async_bulk_wait_group_read<N>();`

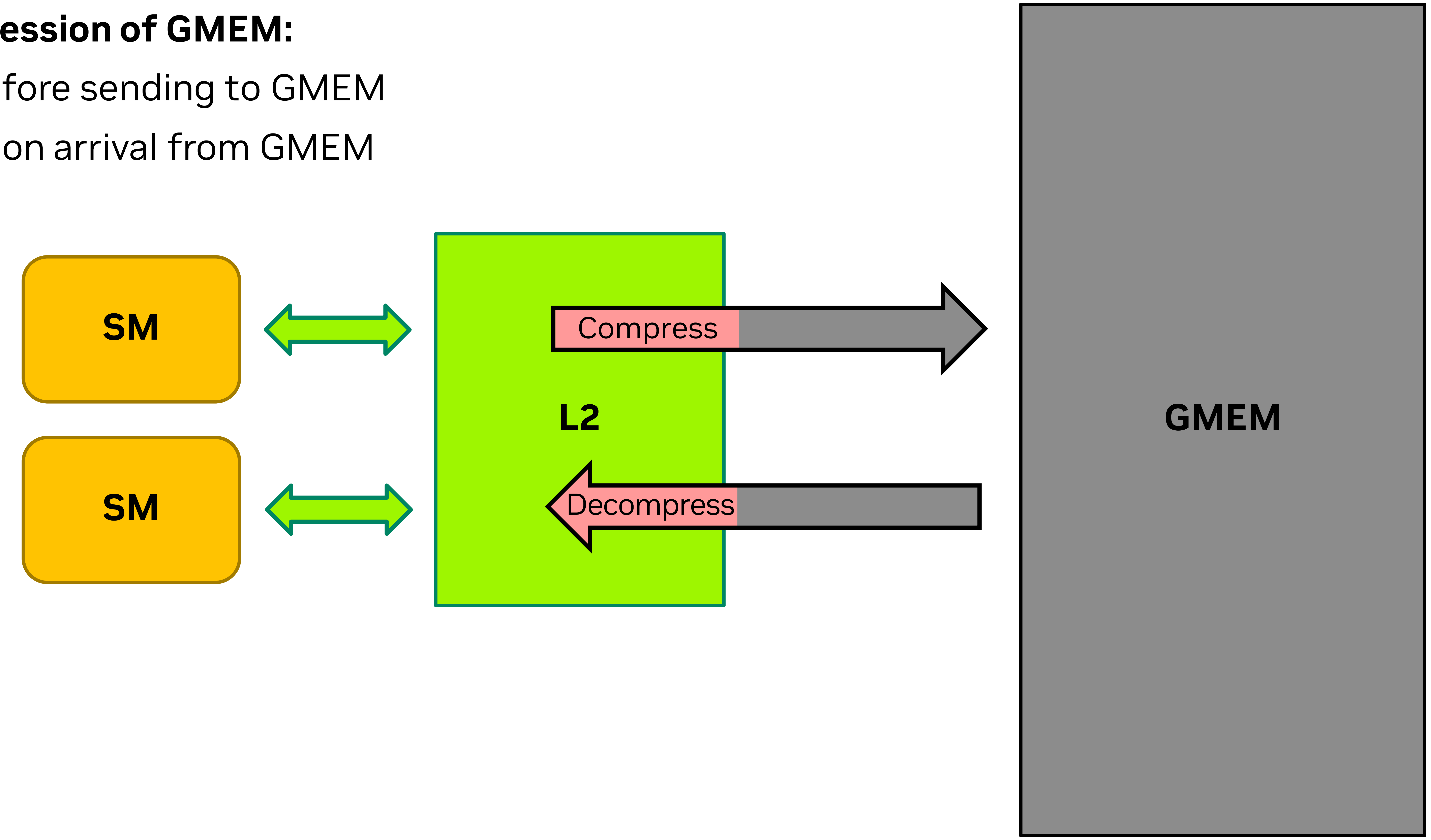
Compressible Memory

Compressible Memory

Introduction

Hardware compression of GMEM:

- Compresses before sending to GMEM
- Decompresses on arrival from GMEM



Compressible Memory

Quick facts

Compressible memory:

- **Fully transparent** (only change your GMEM allocation calls)
  - "Compresses bandwidth" (GMEM <–> L2)
  - Doesn't compress storage (GMEM or L2)
- 128B (cache line) granularity
- Lossless (2x, 4x, or 1x if cannot compress)
- Compression is automatic (no user control), not all data will compress
- Available on Ada and Hopper
- **May decrease performance** (compression cache misses => large penalties)

Compressible Memory

Saxpy example – kernels

“No-loop” kernel:

```
__global__ void saxpy_no_loop(float4       * __restrict__ z,
                              float4 const* __restrict__ x,
                              float4 const* __restrict__ y,
                              float a, int n)
{
    int const i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        z[i] = a * x[i] + y[i];
}
```

Wave 0
Wave 1
Wave N/blockDim.x/B-1
0
1
B-1
B
B+1
2B-1

B:= SM count

Compressible Memory

Saxpy example – kernels

“Grid-stride loop” kernel:

```
__global__ void saxpy_grid   (float4      * __restrict__ z,
                              float4 const* __restrict__ x,
                              float4 const* __restrict__ y,
                              float a, int n)
{
    int const i0 = blockIdx.x * blockDim.x + threadIdx.x;
    int const step = gridDim.x * blockDim.x;
    #pragma unroll(4)
    for (int i = i0; i < n; i += step)
        z[i] = a * x[i] + y[i];
}
```

Access locality worsens with more waves

Wave 0
0
1
B-1
0
1
B-1
0
1
B-1
0
1
B-1
B := SM count
0
1
B-1
B
B+1
2B-1
0
1
B-1
B
B+1
2B-1
Wave 0
Wave 1
Wave 0
Wave 1

Compressible Memory

Saxpy example – kernels

“Block-stride loop” kernel:

```
__global__ void saxpy_block  (float4      * __restrict__ z,
                              float4 const* __restrict__ x,
                              float4 const* __restrict__ y,
                              float a, int n)
{
    int const i0 = n / gridDim.x * blockIdx.x + threadIdx.x;
    int const i1 = n / gridDim.x * (blockIdx.x + 1) + threadIdx.x;
    #pragma unroll(4)
    for (int i = i0; i < i1; i += blockDim.x)
        z[i] = a * x[i] + y[i];
}
```

Access locality improves with more waves

Wave 0
0
0
1
1
B-2
B-2
B-1
B-1
B := SM count
0
0
1
1
2
2
3
3
2B-4
2B-4
2B-3
2B-3
2B-2
2B-2
2B-1
2B-1
Wave 0
Wave 1

Compressible Memory

Saxpy example – performance

Performance for `x[i] = y[i] = {i, i, i, i}` input data

H100 (1.3GB / array)

Kernel	Uncompressed		Compressed	
	Time (ms)	B-width (TB/s)	Time (ms)	B-width (TB/s)
Grid-loop SMx1	1.10	3.53	0.86	4.54
Grid-loop SMx2	1.11	3.50	1.17	3.31
Grid-loop SMx20	1.09	3.59	1.72	2.26
Block-loop SMx1	1.11	3.52	3.00	**1.30**
Block-loop SMx2	1.10	3.53	2.72	**1.43**
Block-loop SMx20	1.08	3.59	0.77	5.08
No-loop	1.05	3.71	0.67	**5.77**

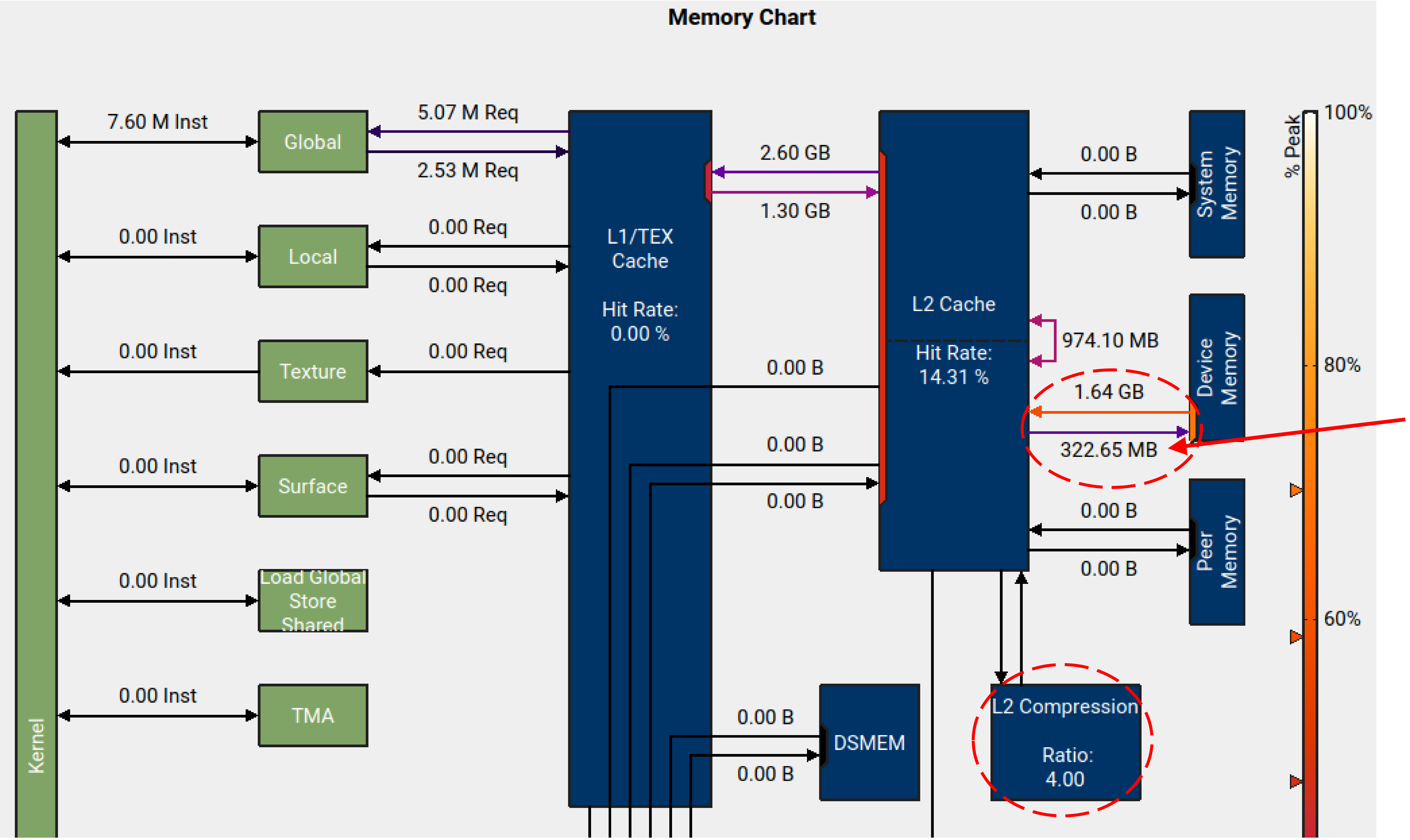
H100 (13GB / array)

Kernel	Uncompressed		Compressed	
	Time (ms)	B-width (TB/s)	Time (ms)	B-width (TB/s)
Grid-loop SMx1	11.0	3.53	26.6	1.46
Grid-loop SMx2	11.1	3.50	39.9	**0.98**
Grid-loop SMx20	10.9	3.57	44.7	**0.87**
Block-loop SMx1	10.9	3.57	44.2	**0.88**
Block-loop SMx2	10.9	3.57	46.5	**0.84**
Block-loop SMx20	10.9	3.58	30.4	1.28
No-loop	10.4	3.73	6.73	**5.78**

Locality is crucial!

Compressible Memory

NCU



Array:1.3GB / 4 => 322.65 MB

Compressible Memory

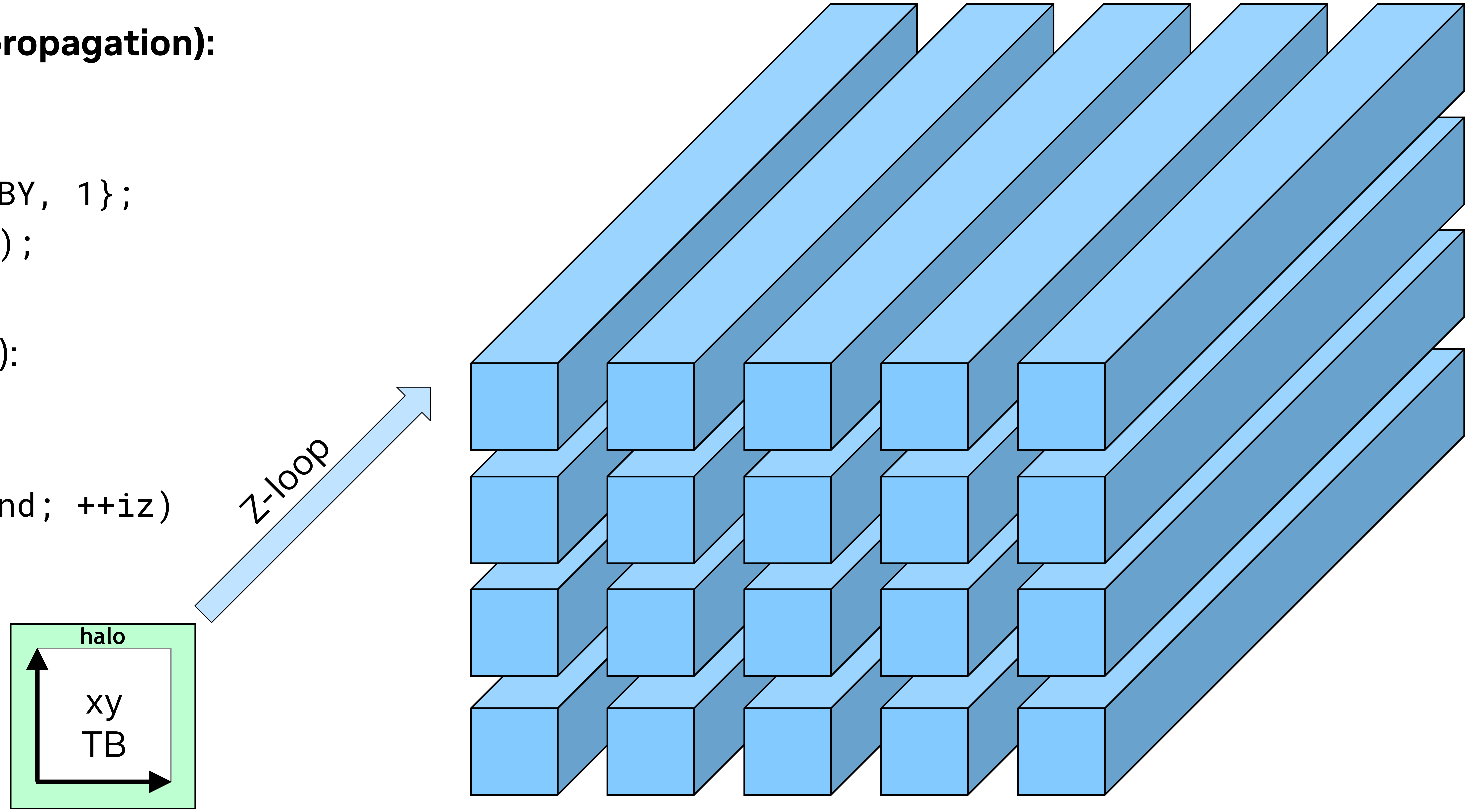
Seismic example

RTM (3D finite-difference wave propagation):

```
Dim3 block = {BX, BY, 1};
Dim3 grid = {nx / BX, ny / BY, 1};
kernel<<<block, grid>>>(...);
```

Kernel z-loop (z is the slowest axis):

```
int zbeg = 0;
int zend = nz;
for (int iz = zbeg; iz < zend; ++iz)
{
    ...
}
```



∂x,∂y via SMEM
∂z via REG queue

See algorithm details in “3D Finite Difference Computation on GPUs using CUDA” by Paulius Micikevicius

Compressible Memory

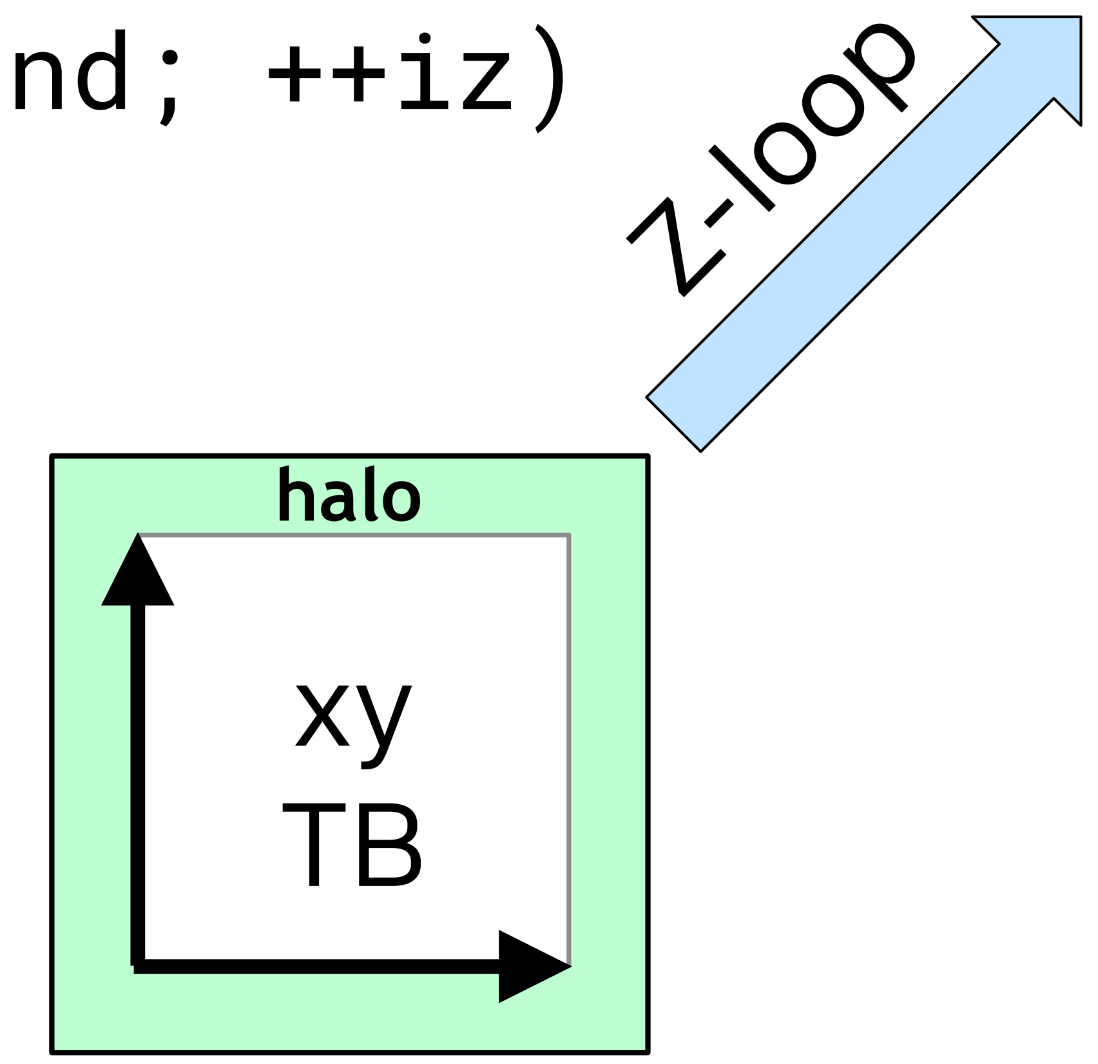
Seismic example – improving locality

RTM (3D finite-difference wave propagation):

```
int BZ = 64;
Dim3 block = {BX, BY, 1};
Dim3 grid = {nx / BX, ny / BY, nz / BZ};
kernel<<<block, grid>>>(...);
```

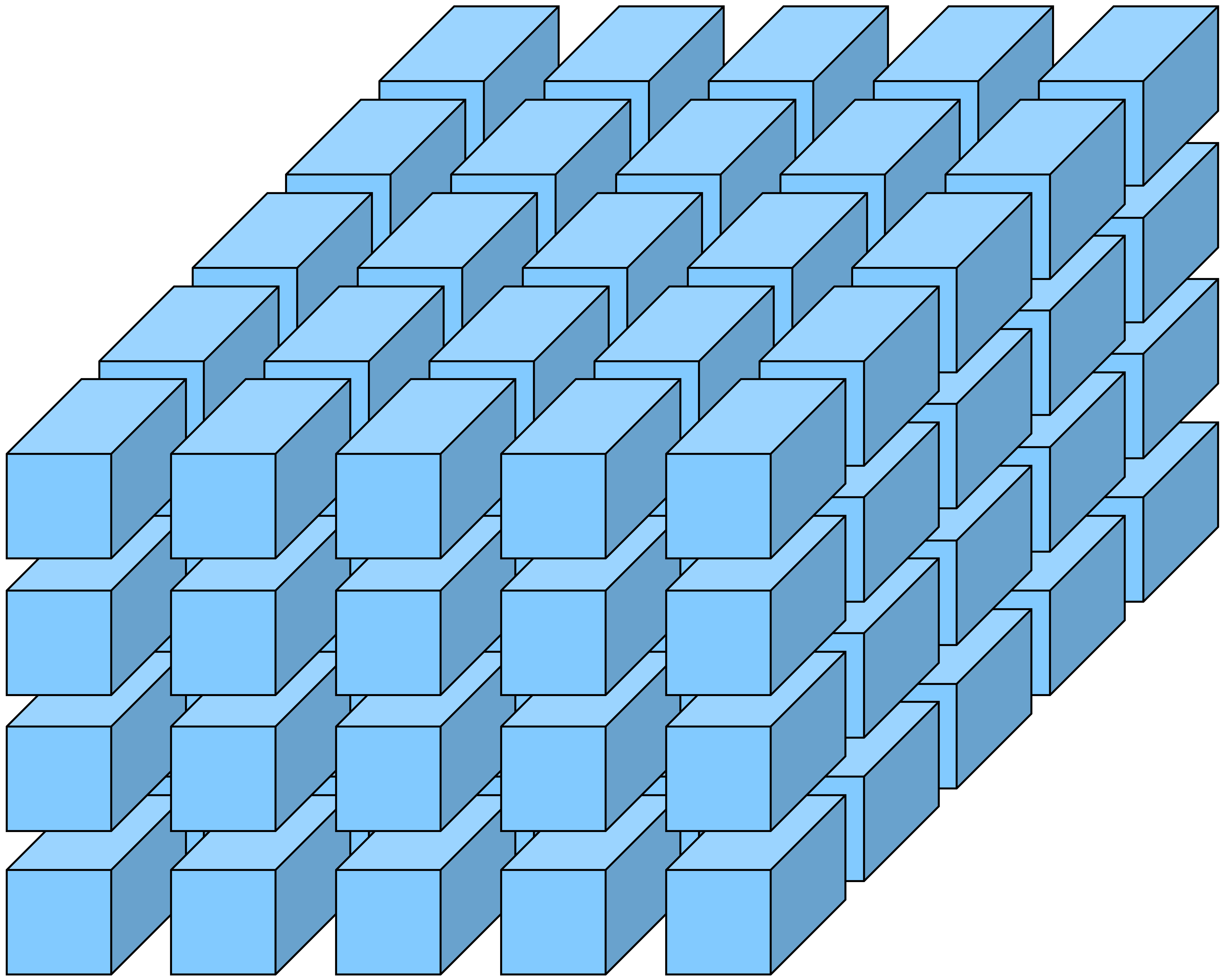
Kernel z-loop (z is the slowest axis):

```
int zbeg = blockIdx.z * BZ;
int zend = min(nz, zbeg + BZ);
for (int iz = zbeg; iz < zend; ++iz)
{
    ...
}
```



∂x,∂y via SMEM

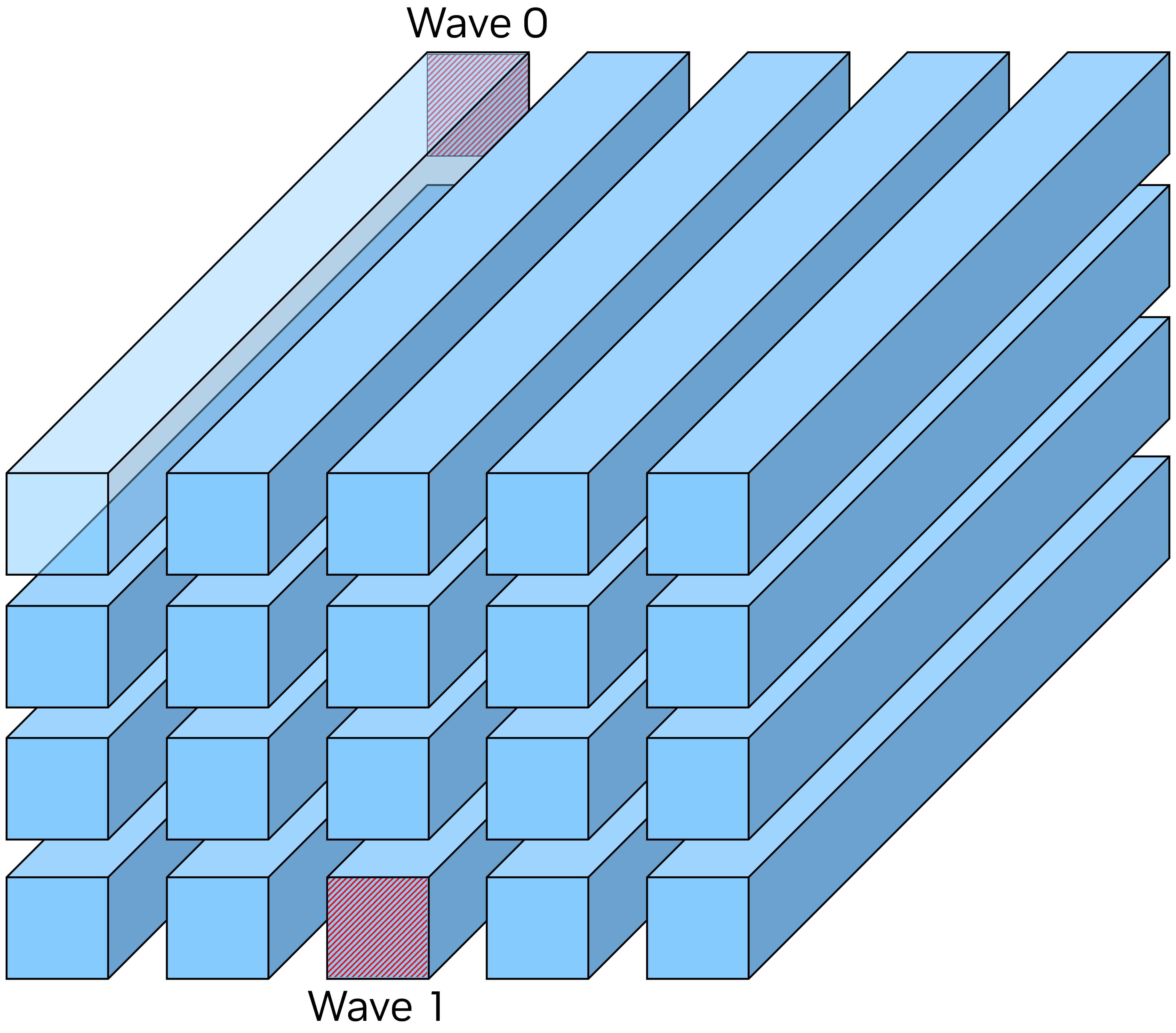
∂z via REG queue

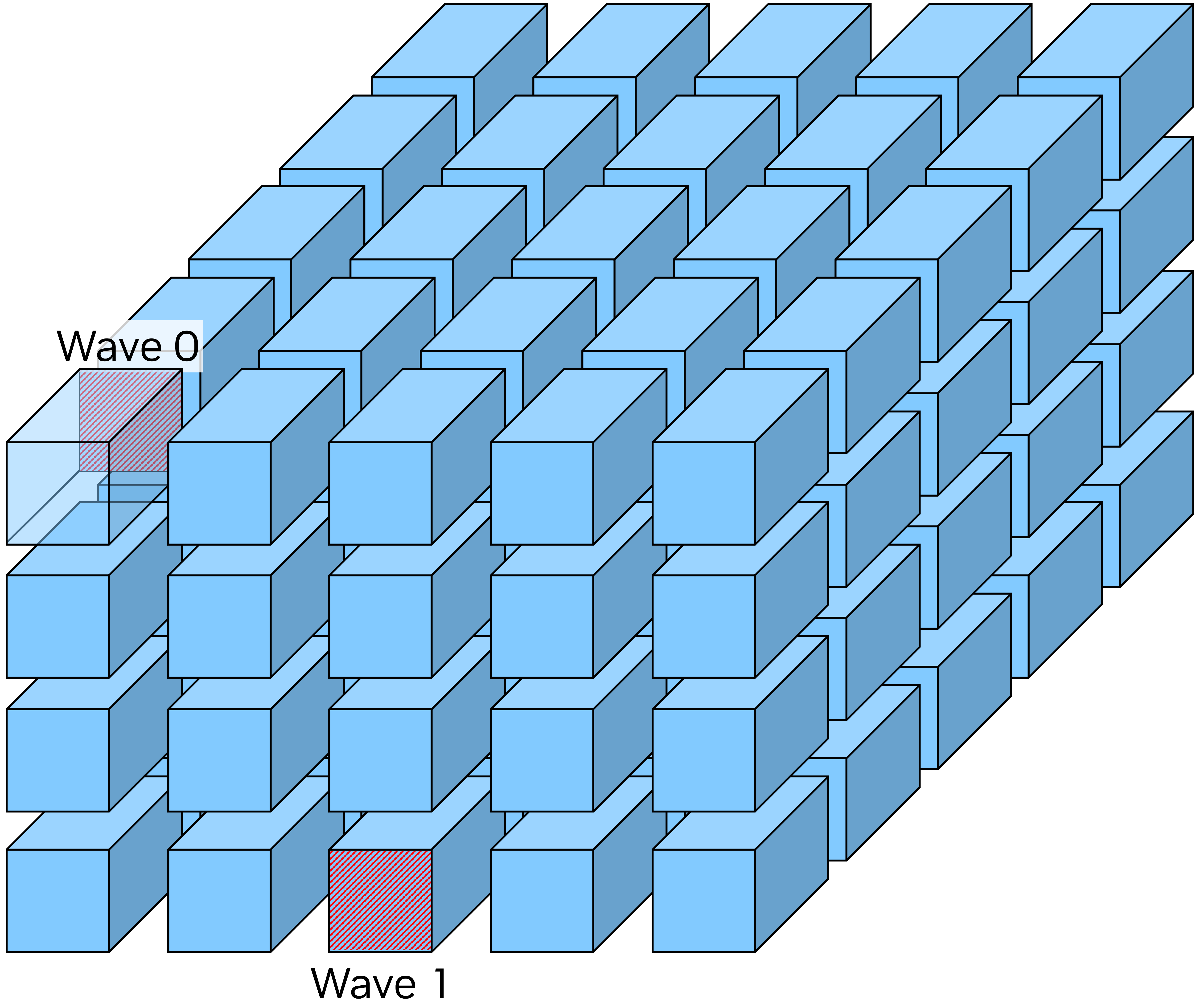


See algorithm details in “3D Finite Difference Computation on GPUs using CUDA” by Paulius Micikevicius

Compressible Memory

Seismic example – improving locality





Locality ≈ (z-distance between wave slices) * z-stride

Locality ≈ (z-distance between wave slices) * z-stride

Compressible Memory

Seismic example – L40s

Problem:

- 8th order TTI RTM, single pass kernel
- Domain: 800×800×800 (9 volumes @ 2.2 GB / volume)
- Thread block size: 32×16

L40s MCells/sec

	No z-blocking	128 z-blocking	64 z-blocking	48 z-blocking	32 z-blocking
No compression	13888	14145	14574	14610	14661
Compression	**1594**	**2081**	4220	5002	**51497**

Compressible Memory

Seismic example – L40s

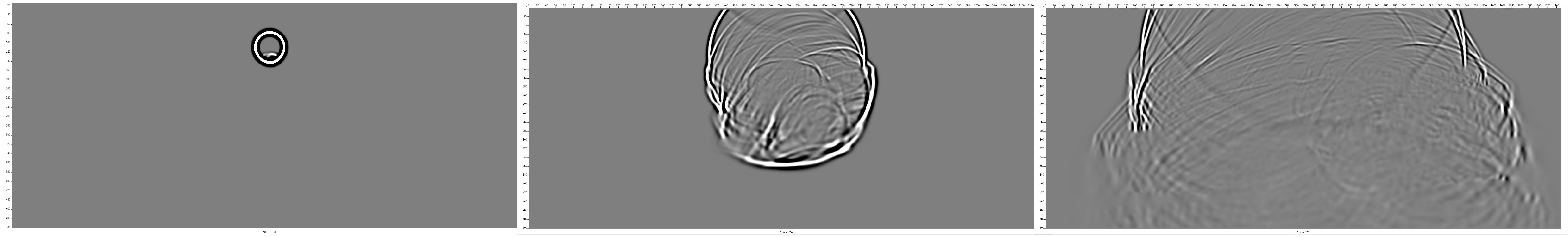
Problem:

- 8th order TTI RTM, single pass kernel
- Domain: 800×800×800 (9 volumes @ 2.2 GB / volume)
- Thread block size: 32×16

L40s MCells/sec

	No z-blocking	128 z-blocking	64 z-blocking	48 z-blocking	32 z-blocking	
No compression	13888	14145	14574	14610	14661	– independent of # time-steps
Compression	**1594**	**2081**	4220	5002	**51497**	100 time-steps
					43004	1000 time-steps
					31640	2000 time-steps
					23422	5000 time-steps
					21440	10000 time-steps

Starts with zero wavefields, which become less compressible as they propagate:



Compressible Memory

Allocating compressible memory

Via Virtual Memory Management API (CUDA driver):

```
CUmemAllocationProp prop = {};
prop.type = CU_MEM_ALLOCATION_TYPE_PINNED;
prop.location.type = CU_MEM_LOCATION_TYPE_DEVICE;
prop.location.id = device;
prop.allocFlags.compressionType = CU_MEM_ALLOCATION_COMP_GENERIC;
```

Round-up allocation size to granularity

```
size_t granularity = 0;
cuMemGetAllocationGranularity(&granularity, &prop,
                              CU_MEM_ALLOC_GRANULARITY_MINIMUM);
size = ((size + granularity - 1) / granularity) * granularity;
```

```
CUdeviceptr ptr;
CUmemGenericAllocationHandle handle;
```

Reserve a virtual address range

```
cuMemAddressReserve(&ptr, size, 0, 0, 0);
```

Allocate physical memory

```
cuMemCreate(&handle, size, &prop, 0);
```

Map allocated memory to VA range

```
cuMemMap(ptr, size, 0, handle, 0);
```

Make accessible

```
CUmemAccessDesc desc = {};
desc.location.type = CU_MEM_LOCATION_TYPE_DEVICE;
desc.location.id = device;
desc.flags = CU_MEM_ACCESS_FLAGS_PROT_READWRITE;
cuMemSetAccess(ptr, size, &desc, 1);
```

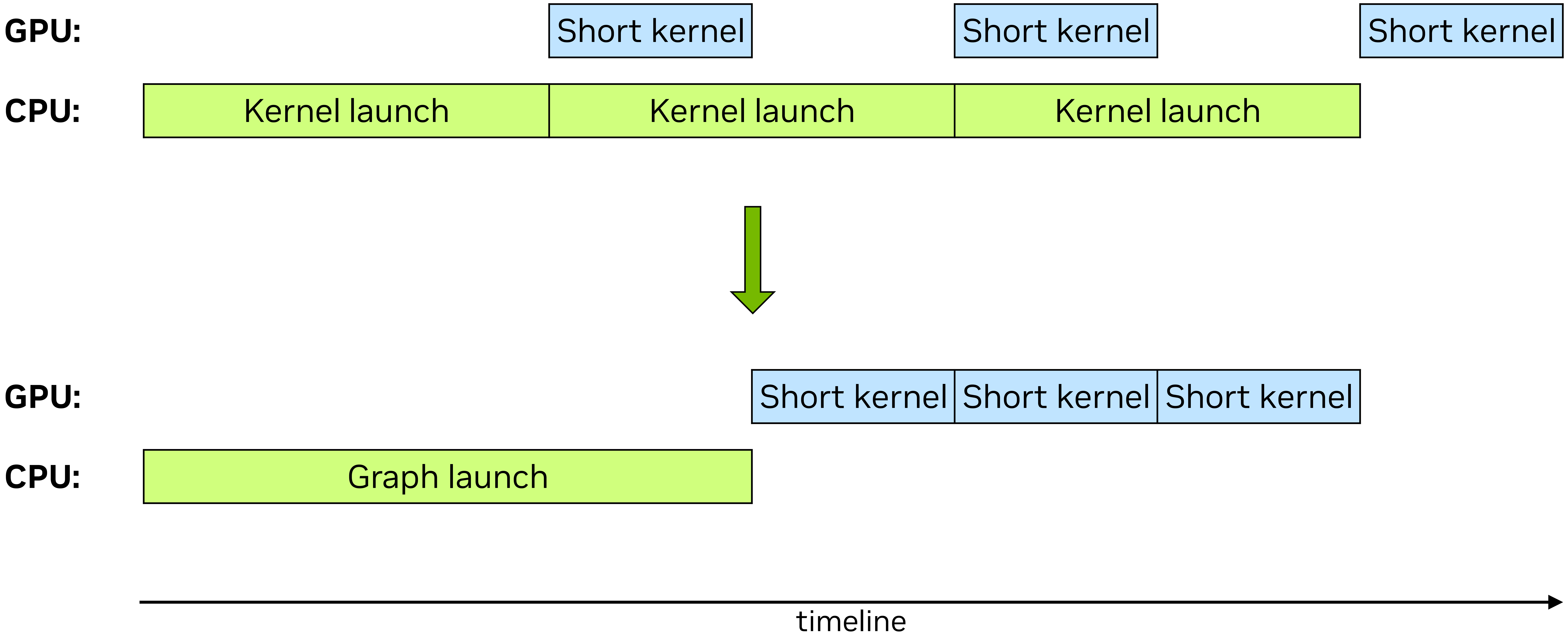
```
cuMemRelease(handle);
```

CUDA Graphs

CUDA Graphs

Reducing launch overhead

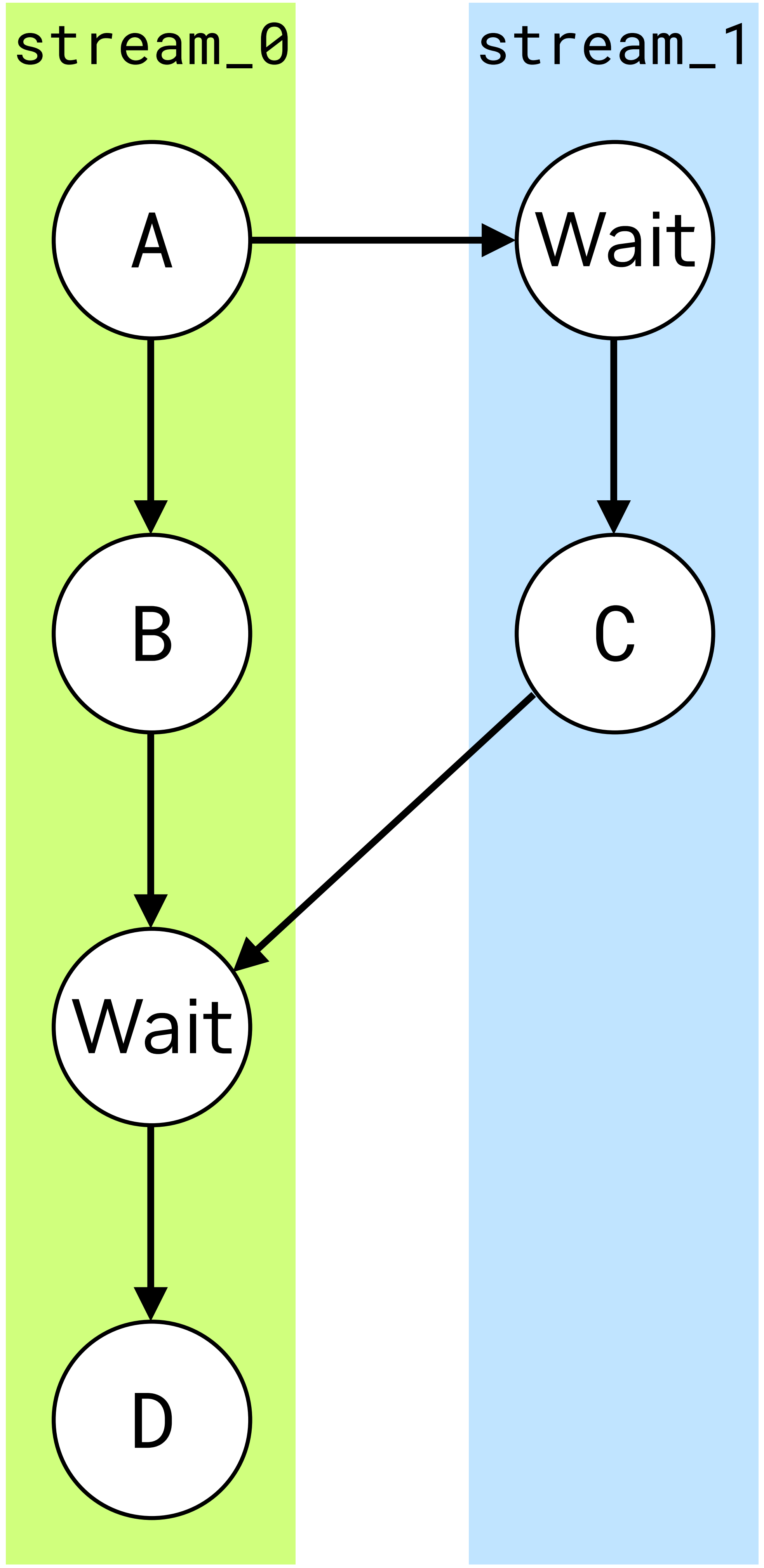
CUDA graphs reduce kernel launch overheads:



CUDA Graphs

Simple example – capturing

```
kernel_A<<<..., stream_0>>>();
cudaEventRecord(event_0, stream_0);
kernel_B<<<..., stream_0>>>();
cudaStreamWaitEvent(stream_1, event_0);
kernel_C<<<..., stream_1>>>();
cudaEventRecord(event_1, stream_1);
cudaStreamWaitEvent(stream_0, event_1);
kernel_D<<<..., stream_0>>>();
```

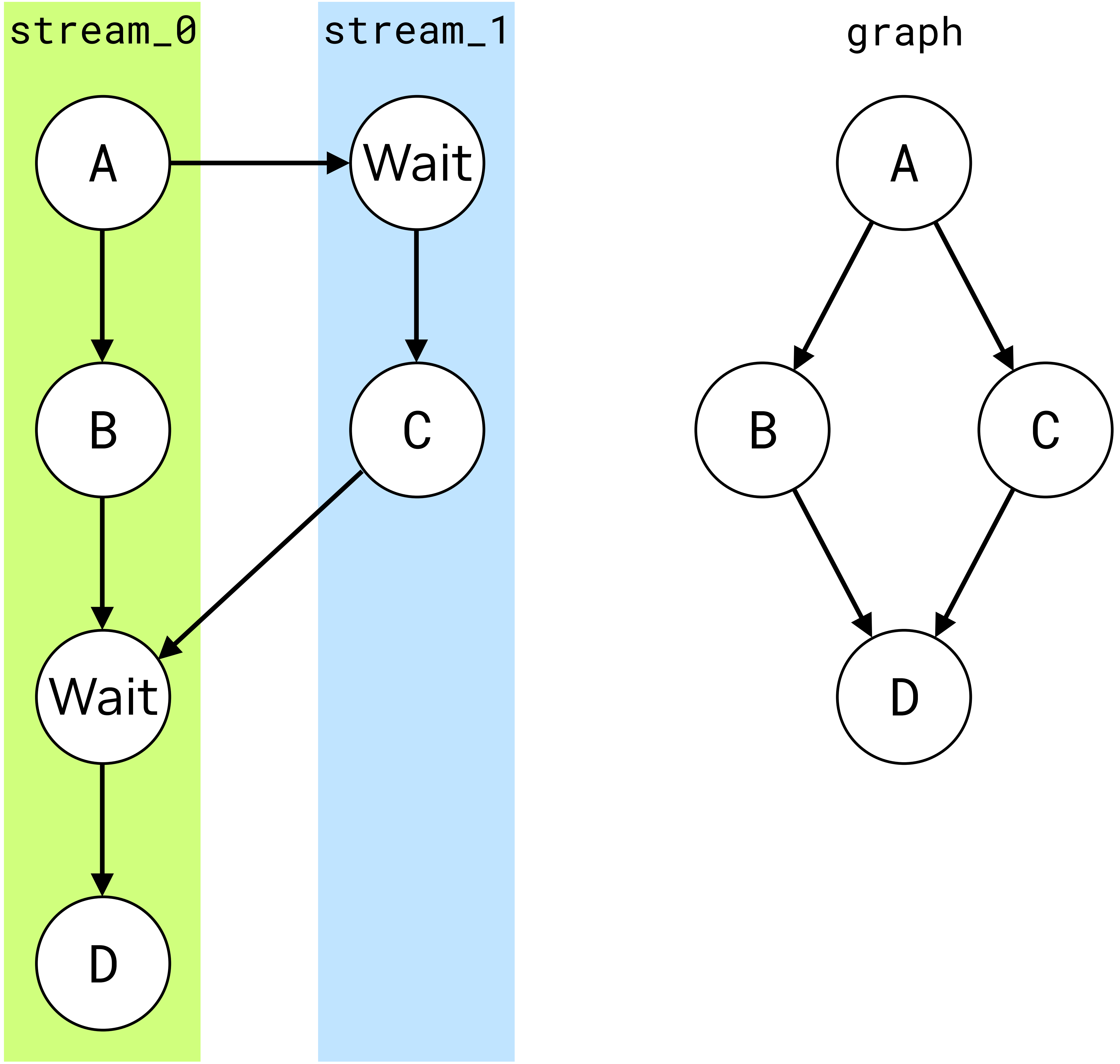


CUDA Graphs

Simple example – capturing

```
cudaStreamBeginCapture(stream_0, cudaStreamCaptureModeGlobal);
kernel_A<<<..., stream_0>>>();
cudaEventRecord(event_0, stream_0);
kernel_B<<<..., stream_0>>>();
cudaStreamWaitEvent(stream_1, event_0);
kernel_C<<<..., stream_1>>>();
cudaEventRecord(event_1, stream_1);
cudaStreamWaitEvent(stream_0, event_1);
kernel_D<<<..., stream_0>>>();
cudaGraph_t graph;
cudaStreamEndCapture(stream_0, &graph);
```

"dry-run"
during capture



CUDA Graphs

Simple example – steps

1. Capture graph (~ *write code*):

   ```
   cudaStreamBeginCapture(stream_0, cudaStreamCaptureModeGlobal);
   <CAPTURED ACTIONS>
   cudaGraph_t graph;
   cudaStreamEndCapture(stream_0, &graph);
   ```

2. Create executable graph (~ *compile code*):

   ```
   cudaGraphExec_t graph_exec;
   cudaGraphInstantiate(&graph_exec, graph);
   ```

3. Launch executable graph (~ *execute compiled code*):

   ```
   cudaGraphLaunch(graph_exec, stream);
   ```

Graph encapsulates:

- Stream dependencies and concurrency
  - **Branches execute concurrently even though `cudaGraphLaunch` into a single stream**
  - Stream priorities can be inherited by nodes (requires `cudaGraphInstantiateFlagUseNodePriority` flag)
  - Streams used while capturing can be destroyed after capturing
- Kernel parameters

  – how do you change them?

CUDA Graphs

Simple example – graph modifications

Common time-stepping pattern:

```
for (int it = 0; it < nit; ++it) {
    kernel<<<...>>>(ptr_prev, ptr_next);
    // more kernels, etc.
    std::swap(ptr_prev, ptr_next);
}
```

Several approaches:

I. Capture even- and odd-iteration graphs and `cudaGraphLaunch(graph_exec[it & 1], stream);`
(see examples in https://github.com/NVIDIA/multi-gpu-programming-models/tree/master/nccl_graphs)

II. Always capture and `cudaGraphExecUpdate`

III. Explicit graph construction API:
`cudaStreamGetCaptureInfo`
`cudaGraphAddKernelNode` – returns a node handle used by `cudaGraphExecKernelNodeSetParams` during iterations
`cudaStreamUpdateCaptureDependencies`

Topology changes typically require graph re-instantiation.

Minor topology changes may not, e.g., turn node on/off (`cudaGraphNodeSetEnabled`)

CUDA Graphs

Other capture restrictions

Restrictions (not comprehensive list):

- Default legacy stream cannot be captured: `libraryCall(cudaStreamDefault)` → `libraryCall(stream)`
- Synchronous calls cannot be captured: `cudaMemcpy` → `cudaMemcpyAsync`
- Cannot synchronize: `cudaStreamSynchronize`, `cudaDeviceSynchronize`
- Host logic: `host_logic` → `cudaLaunchHostFunc(stream, host_logic)`

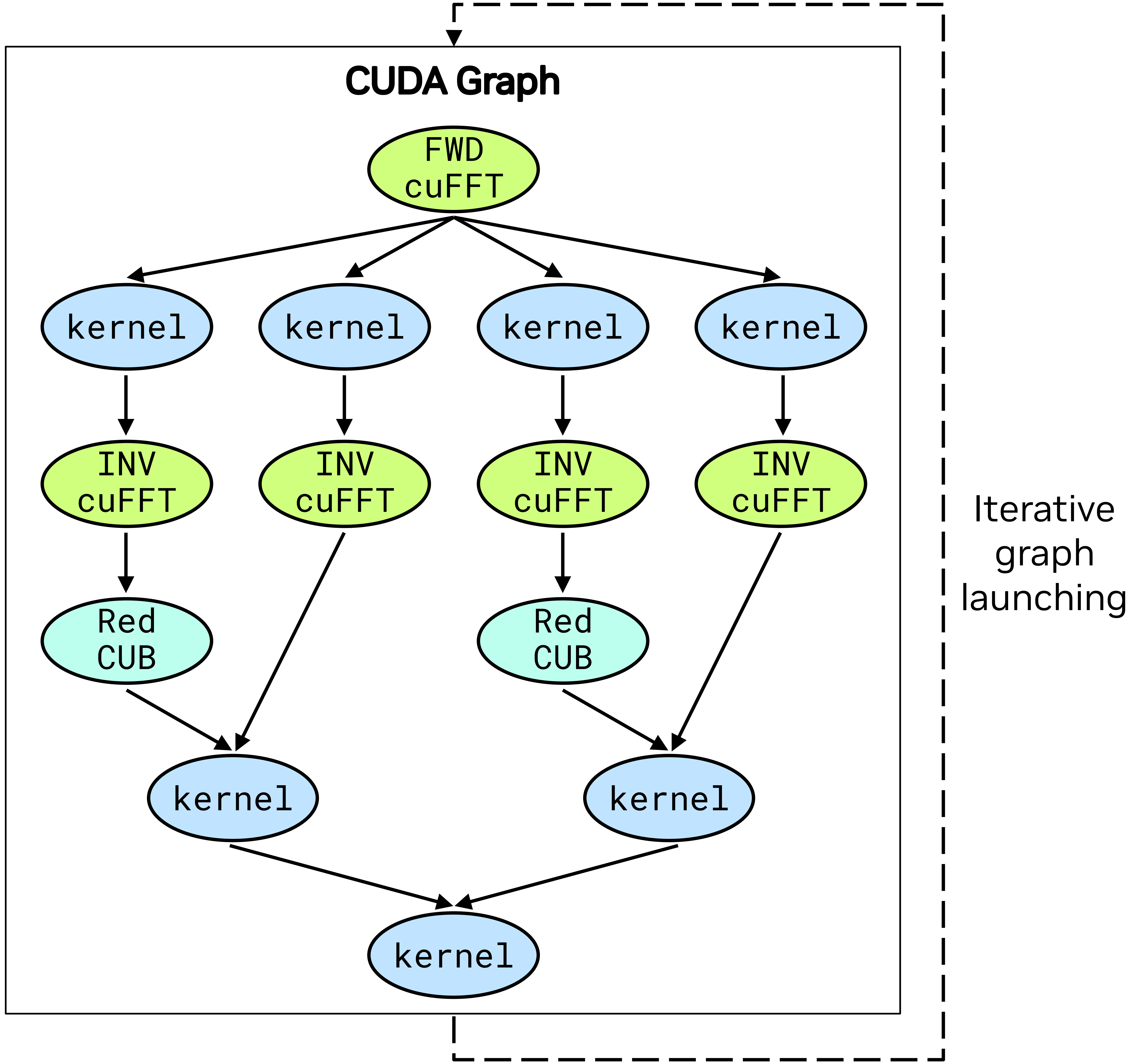
Can capture what can be put in a stream.

CUDA Graphs

Example – algo

Workflow based on seismic trace processing:

1. Forward cuFFT
2. Frequency continuations into multiple traces with different frequency bands
3. Inverse cuFFTs of multiple traces (cannot batch due to variable sizes)
4. Reductions (CUB) and combining even/odd
5. Final combining



CUDA Graphs

Example – code

```
cudaStreamBeginCapture(f_stream, cudaStreamCaptureModeGlobal);
cufftExecR2C(f_plan, f_data, of_data); // f_plan is associated with f_stream
cudaEventRecord(f_event, f_stream);
for (int i = 0; i < m; ++i) {
    cudaStreamWaitEvent(b_stream[i], f_event);
    kernel_0<<<1, BX, 0, b_stream[i]>>>(b_data[i], of_data, of_n, b_n[i]);
    cufftExecC2R(b_plan[i], b_data[i], ob_data[i]); // b_plan[i] is associated with b_stream[i]
}
for (int i = 0; i < m; i += 2) {
    cub::DeviceReduce::Sum(cub_storage[i].first, cub_storage[i].second, ob_data[i], ocub + i, ob_n[i], b_stream[i]);
    cudaEventRecord(b_event[i], b_stream[i]);
    cudaStreamWaitEvent(b_stream[i + 1], b_event[i]);
    kernel_1<<<1, BX, 0, b_stream[i + 1]>>>(ob_data[i + 1], ocub + i, ob_n[i + 1]);
    cudaEventRecord(b_event[i + 1], b_stream[i + 1]);
    cudaStreamWaitEvent(f_stream, b_event[i + 1]);
}
kernel_2<<<m, BX, 0, f_stream>>>(f_data, ob_data_gpu, ob_n_gpu, f_n);
cudaStreamEndCapture(f_stream, &graph);
```

No extra effort
to capture library calls !!
(into sub-graphs)

CUDA Graphs

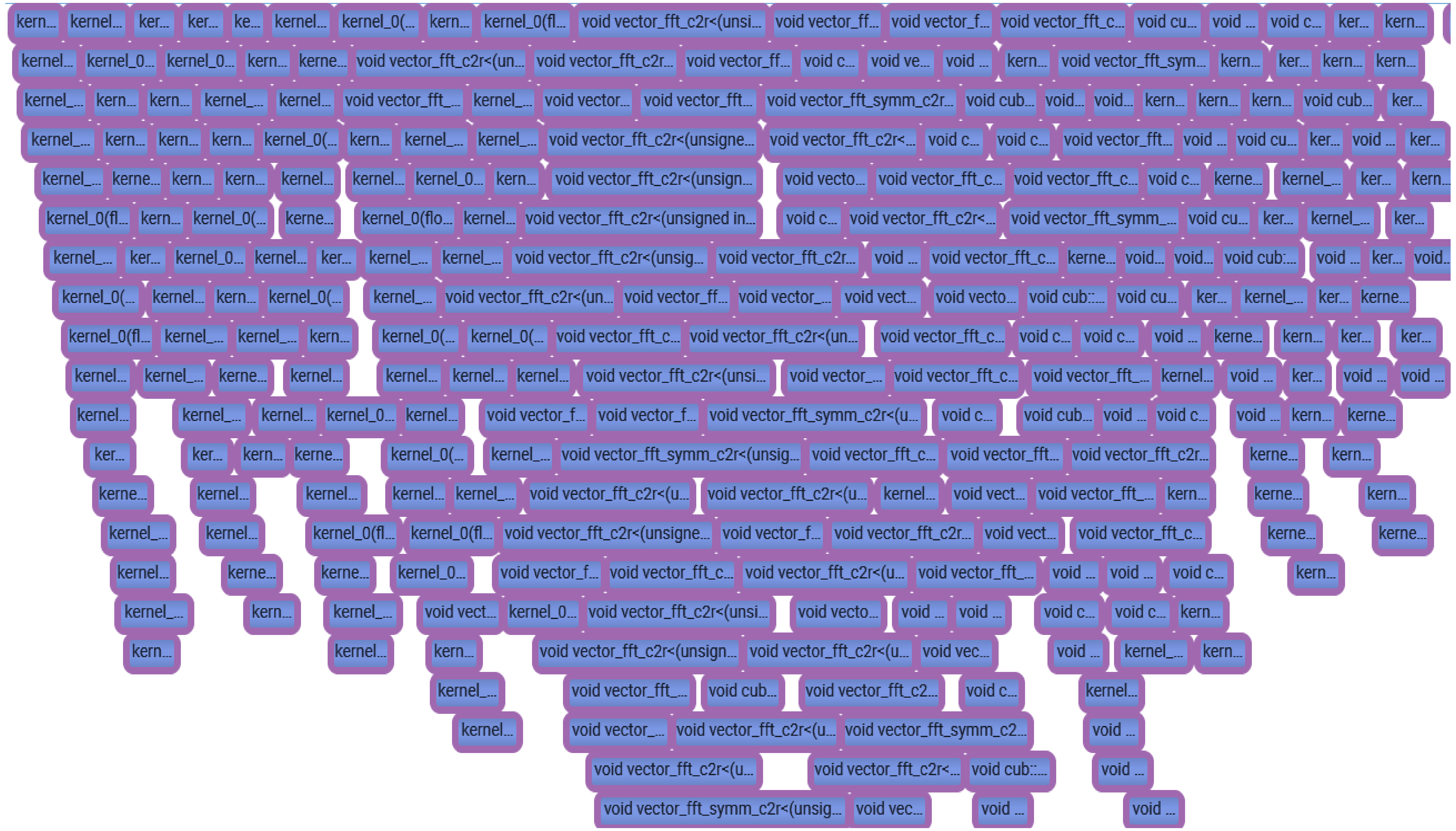
Example – performance

Performance:

- 1024 samples input trace.
- 100 output traces 128-8192 sample range.

Graphs provide **5X speedup** (H200 test).

Graphs: packed and compact


kern… kernel… ker… ker… ke… kernel… kernel_0(… kern… kernel_0(fl… void vector_fft_c2r<(unsi… void vector_ff… void vector_f… void vector_fft_c… void cu… void … void c… ker… kern…
kernel… kernel_0… kernel_0… kern… kerne… void vector_fft_c2r<(un… void vector_fft_c2r… void vector_ff… void c… void ve… void … kern… void vector_fft_sym… kern… ker… kern… kern…
kernel_… kern… kern… kernel_… kernel… void vector_fft… kernel_… void vector… void vector_fft… void vector_fft_symm_c2r… void cub… void… void… kern… kern… kern… void cub… ker…
kernel_… kern… kern… kern… kernel_0(… kern… kernel_… kernel_… void vector_fft_c2r<(unsigne… void vector_fft_c2r<… void c… void c… void vector_fft… void … void cu… ker… void … ker…
kernel_… kerne… kern… kern… kernel… kernel… kernel_0… kern… void vector_fft_c2r<(unsign… void vecto… void vector_fft_c… void vector_fft_c… void c… kerne… kernel_… ker… kern…
kernel_0(fl… kern… kernel_0(… kerne… kernel_0(flo… kernel… void vector_fft_c2r<(unsigned in… void c… void vector_fft_c2r<… void vector_fft_symm_… void cu… ker… kernel_… ker…
kernel_… ker… kernel_0… kernel… ker… kernel_… kernel_… void vector_fft_c2r<(unsig… void vector_fft_c2r… void … void vector_fft_c… kerne… void… void… void cub:… void … ker… void…
kernel_0(… kernel… kern… kernel_0(… kernel… void vector_fft_c2r<(un… void vector_ff… void vector_… void vect… void vecto… void cub::… void cu… ker… kernel_… ker… kerne…
kernel_0(fl… kernel_… kernel_… kern… kernel_0(… kernel_0(… void vector_fft_c… void vector_fft_c2r<(un… void vector_fft_c… void c… void c… void … kerne… kern… ker… ker…
kernel… kernel_… kerne… kernel… kernel… kernel… kernel… void vector_fft_c2r<(unsi… void vector_… void vector_fft_c… void vector_fft_… kernel… void … ker… void … void …
kernel… kernel_… kernel… kernel_0… kernel… void vector_f… void vector_f… void vector_fft_symm_c2r<(u… void c… void cub… void … void c… void … kern… kerne…
ker… ker… kern… kerne… kernel_0(… kernel_… void vector_fft_symm_c2r<(unsig… void vector_fft_c… void vector_fft… void vector_fft_c2r… kerne… kern…
kerne… kernel… kernel… kernel… kernel_… void vector_fft_c2r<(u… void vector_fft_c2r<(u… kernel… void vect… void vector_fft_… kern… kerne… kern…
kernel_… kernel… kernel_0(fl… kernel_0(fl… void vector_fft_c2r<(unsigne… void vector_f… void vector_fft_c2r… void vect… void vector_fft_c… kerne… kerne…
kernel… kerne… kerne… kernel_0… void vector_f… void vector_fft_c… void vector_fft_c2r<(u… void vector_fft_… void … void … void c… kern…
kernel_… kern… kernel_… void vect… kernel_0… void vector_fft_c2r<(unsi… void vecto… void … void … void c… void c… kern…
kern… kernel… kern… void vector_fft_c2r<(unsign… void vector_fft_c2r<(u… void vec… void … kernel_… kern…
kernel_… void vector_fft_… void cub… void vector_fft_c2… void c… kernel…
kernel… void vector_… void vector_fft_c2r<(u… void vector_fft_symm_c2… void …
void vector_fft_c2r<(u… void vector_fft_c2r<… void cub::… void …
void vector_fft_symm_c2r<(unsig… void vec… void … void …

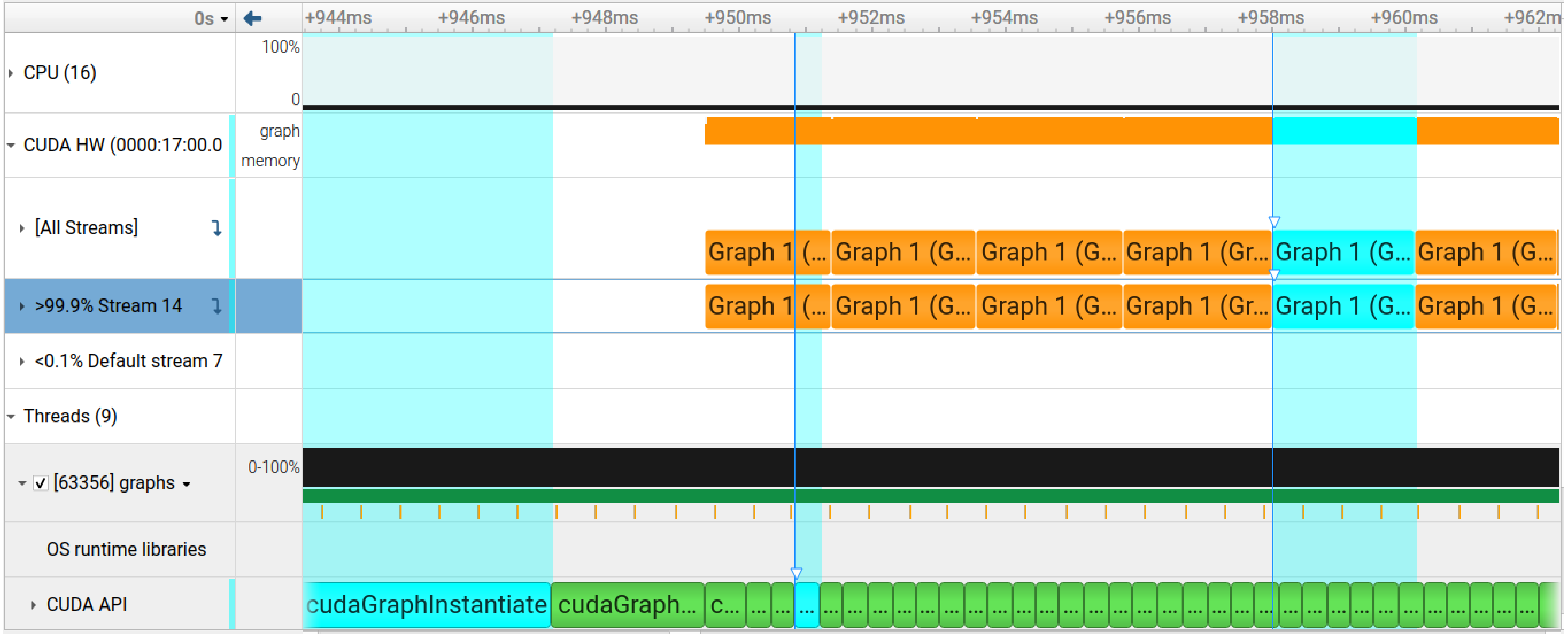
No-graphs: almost useless to employ multiple streams because launch overheads “serialize” kernels execution

kernel_2(float *, float **, int *, int)
v…

CUDA Graphs

Nsight systems

- To trace nodes inside graphs, use `--cuda-graph-trace node` nsys option
  Note: has higher profiling overhead than default.
- Default is `--cuda-graph-trace graph` option:



CUDA Graphs

Further reading

CUDA graphs API is rich, we just looked at tip of the iceberg.

See also at this GTC:

- “Multi GPU Programming Models for HPC and AI” by Jiri Kraus [S61339] and accompanying examples: https://github.com/NVIDIA/multi-gpu-programming-models/tree/master/nccl_graphs
- “Accelerating Drug Discovery: Optimizing Dynamic GPU Workflows with CUDA Graphs, Mapped Memory, C++ Coroutines, and More” by Jiqun Tu & Ellery Russell [S61156]

Related resources:

- GTS’23 [S51211]: “CUDA Graphs 101”
- GTC’21 [S32082]: “Effortless CUDA Graphs”
- Technical Blog: “Constructing CUDA Graphs with Dynamic Parameters”
- Technical Blog: “Getting Started with CUDA Graphs”
- CUDA Samples:
  - simpleCudaGraphs
  - jacobiCudaGraphs
  - graphMemoryNodes
  - graphMemoryFootprint

Summary

Summary

And take-aways

- **Memory model:**
  - What every CUDA programmer should know about memory…
- **Asynchronous memory copies & barriers:**
  - Increase bytes in flight + less pressure (reg, pred, MIO, etc.)
  - Require some programming effort
- **Compressible memory:**
  - Improves effective GMEM bandwidth (if data is compressible)
  - Can be a very low-hanging fruit: just replace `cudaMalloc`-s
  - Danger: sensitive to access patterns (**can decrease performance**); may require kernel changes to be effective
- **CUDA graphs:**
  - Way to go for short-timed kernels
  - Can be a low-hanging fruit: stream capturing is just a few lines

Core Performance Optimization Techniques

Core Performance Optimization Techniques at GTC'24

List of presentations

- Introduction to CUDA Programming and Performance Optimization [S62191]
- Advanced Performance Optimization in CUDA [S62192]
- Performance Optimization for Grace CPU Superchip [S62275]
- Grace Hopper Superchip Architecture and Performance Optimizations for Deep Learning Applications [S61159]
- Multi GPU Programming Models for HPC and AI [S61339]
- More Data, Faster: GPU Memory Management Best Practices in Python and C++ [S62550]
- Harnessing Grace Hopper's Capabilities to Accelerate Vector Database Search [S62339]
- From Scratch to Extreme: Boosting Service Throughput by Dozens of Times with Step-by-Step Optimization [S62410]