NVIDIA

LIBCU++ OVERVIEW

Gonzalo Brito Gadeschi

Developer Technology Engineer – Compute Performance

CSCS’21

Some concepts we will talk about:

Hardware Features

- Atomics
- Shared Memory Barrier in Ampere
- Asynchronous memory copies in Ampere

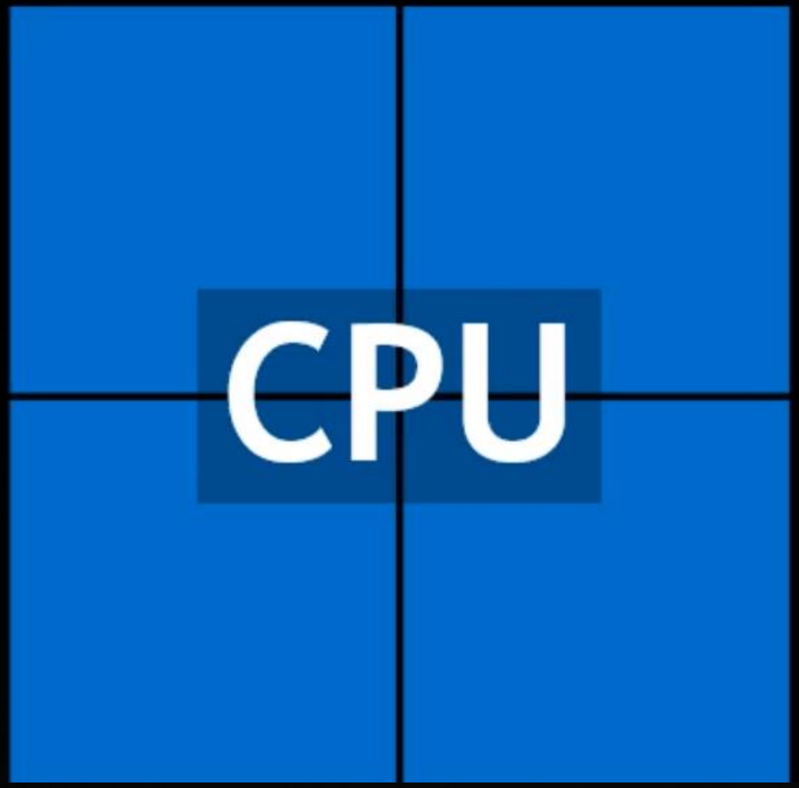
Async Programming Model

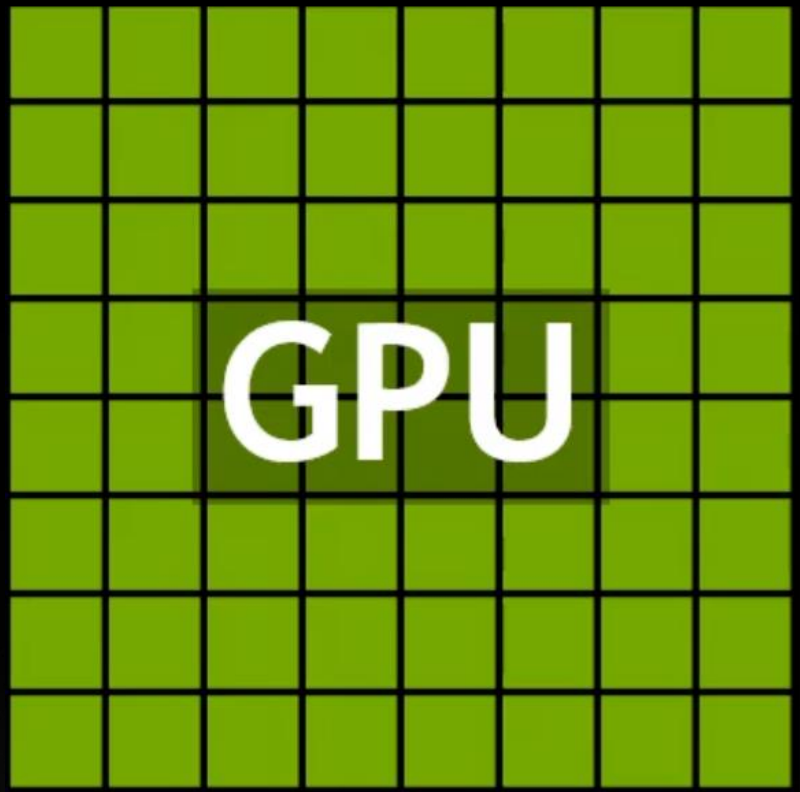
- cuda:: thread scopes
- cuda::[std::] atomics
- cuda::memcpy_async
- cuda::pipeline

LIBCU++

LIBCU++

The C++ Standard Library for Your Entire System





Open source: https://github.com/NVIDIA/libcudacxx

LIBCU++

Is not a complete C++ standard library today

1.0.0 (CUDA 10.2)
`atomic<T>` (SM60+)
Type Traits

1.1.0 (CUDA 11.0)
`atomic<T>::wait/notify` (SM70+)
`barrier` (SM70+)
`latch` (SM70+)
`*_semaphore` (SM70+)
`cuda::memcpy_async` (SM70+)
`chrono::` Clocks & Durations
`ratio<Num, Denom>`

1.2.0 (CUDA 11.1)
`cuda::pipeline` (SM80+)

1.3.0 (CUDA 11.2)
`tuple<T0, T1, ...>`

1.4.1 (CUDA 11.3)
`complex`
`byte`
`chrono::` Dates & Calendars

2.0.0
`atomic_ref<T>` (SM60+)
Memory Resources & Allocators

Open source: https://github.com/NVIDIA/libcudacxx

LIBCU++

Does not interfere or replace your host standard library

Host Compiler's Standard Library (GCC, MSVC, etc)

#include <…> std::	ISO C++, __host__ only. Complete, strictly conforming to Standard C++.
#include <cuda/std/…> cuda::std::	CUDA C++, __host__ __device__. Subset, strictly conforming to Standard C++.
#include <cuda/…> cuda::	CUDA C++, __host__ __device__. Conforming extensions to Standard C++.

libcu++ (NVCC)

OUTLOOK: LIBNV++

Complete C++ standard library: everything works on Host, most features work on Device

`#include <…>` `std::`	ISO C++, `__host__ __device__`. Complete, strictly conforming to Standard C++.
`#include <cuda/std/…>` `cuda::std::`	CUDA C++, `__host__ __device__`. Subset, strictly conforming to Standard C++.
`#include <cuda/…>` `cuda::`	CUDA C++, `__host__ __device__`. Conforming extensions to Standard C++.

libnv++ (NVC++)

Two standard library choices for nvc++

nvc++ -stdlib=libstdc++	std:: is host-only – ABI compatible with GCC
nvc++ -stdlib=libnv++	std:: is heterogeneous – not ABI compatible with GCC

GTC'21 talks: The NVIDIA C++ Standard Library S3135 and Inside NVC++ and NVFORTRAN S31358

CUDA::[STD::]ATOMIC

MESSAGE PASSING BETWEEN TWO THREADS

Global memory

```
__managed__ int flag;
__managed__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    int& flag, int& data, int value
) {
    data = value;
    flag = 1;
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    int& flag, int& data
) {
    while (flag != 1) ;
    return data;
}
```

MESSAGE PASSING BETWEEN TWO THREADS

Incorrect CUDA C++

Global memory

```
__managed__ int flag;
__managed__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    int& flag, int& data, int value
) {
    data = value;
    flag = 1; // data-race
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    int& flag, int& data
) {
    while (flag != 1) ; // data-race
    return data;
}
```

MESSAGE PASSING BETWEEN TWO THREADS

Global memory

```
__managed__ int flag;
__managed__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    volatile int& flag, int& data, int value
) {
    data = value;
    flag = 1;
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    volatile int& flag, int& data
) {
    while (flag != 1) ;
    return data;
}
```

MESSAGE PASSING BETWEEN TWO THREADS

Still incorrect CUDA C++

Global memory

```
__managed__ int flag;
__managed__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    volatile int& flag, int& data, int value
) {
    data = value;
    flag = 1; // data-race
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    volatile int& flag, int& data
) {
    while (flag != 1) ; // data-race
    return data;
}
```

volatile **does not** synchronize

MESSAGE PASSING BETWEEN TWO THREADS

Global memory

```
__managed__ int flag;
__managed__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    int& flag, int& data, int value
) {
    data = value;
    atomicExch(&flag, 1);
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    int& flag, int& data
) {
    while (atomicAdd(&flag,0) != 1) ;
    return data;
}
```

MESSAGE PASSING BETWEEN TWO THREADS

Still incorrect CUDA C++

Global memory

```
__managed__ int flag;
__managed__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    int& flag, int& data, int value
) {
    data = value;
    atomicExch(&flag, 1);  // relaxed
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    int& flag, int& data
) {
    while (atomicAdd(&flag,0) != 1) ; // relaxed
    return data;
}
```

- No synchronization: Thread 1 load of *data* does not read *value* written by Thread 0
- Another problem

MESSAGE PASSING BETWEEN TWO THREADS

Almost correct CUDA C++

Global memory

```
__managed__ int flag;
__managed__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    int& flag, int& data, int value
) {
    data = value;
    __threadfence_system();
    atomicExch(&flag, 1);
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    int& flag, int& data
) {
    while (atomicAdd(&flag,0) != 1) ;
    __threadfence_system();
    return data;
}
```

MESSAGE PASSING BETWEEN TWO THREADS

Almost correct CUDA C++

Global memory

```
__managed__ int flag;
__managed__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    int& flag, int& data, int value
) {
    data = value;
    __threadfence_system();
    atomicExch(&flag, 1);
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    int& flag, int& data
) {
    while (atomicAdd(&flag,0) != 1) ;
    __threadfence_system();
    return data;
}
```

Does not compile: atomicExch, atomicAdd, __threadfence_system are __device__ only

MESSAGE PASSING BETWEEN TWO THREADS

Almost correct CUDA C++

Global memory

```
__managed__ int flag;
__managed__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    int& flag, int& data, int value
) {
    data = value;
#ifdef __CUDA_ARCH__
    __threadfence_system();
    atomicExch(&flag, 1);
#else
    // ...host code...
#endif
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    int& flag, int& data
) {
#ifdef __CUDA_ARCH__
    while (atomicAdd(&flag,0) != 1) ;
    __threadfence_system();
#else
    // ...host code...
#endif
    return data;
}
```

MESSAGE PASSING BETWEEN TWO THREADS

Correct CUDA C++ and host code using libcu++

Global memory

```
__managed__ atomic<bool> flag;
__managed__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    atomic<bool>& flag, int& data, int value
) {
    data = value;
    flag = true;
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    atomic<bool>& flag, int& data
) {
    while (!flag.load()) ;
    return data;
}
```

MESSAGE PASSING BETWEEN TWO THREADS

Correct CUDA C++ and host code using libcu++

Global memory

```
__managed__ atomic<bool> flag;
__managed__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    atomic<bool>& flag, int& data, int value
) {
    data = value;
    flag = true;
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    atomic<bool>& flag, int& data
) {
    while (!flag.load()) ;
    return data;
}
```

Performance: sequential-consistent loads & stores, but acquire-release suffices.

MESSAGE PASSING BETWEEN TWO THREADS

Better CUDA C++ and host code using libcu++

Global memory

```
__managed__ atomic<bool> flag;
__managed__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    atomic<bool>& flag, int& data, int value
) {
    data = value;
    flag.store(true, memory_order_release);
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    atomic<bool>& flag, int& data
) {
    while (!flag.load(memory_order_acquire)) ;
    return data;
}
```

MESSAGE PASSING BETWEEN TWO THREADS

Better CUDA C++ and host code using libcu++

Global memory

```
__managed__ atomic<bool> flag;
__managed__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
   atomic<bool>& flag, int& data, int value
) {
   data = value;
   flag.store(true, memory_order_release);
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
   atomic<bool>& flag, int& data
) {
   while (!flag.load(memory_order_acquire)) ;
   return data;
}
```

Performance: explicit pooling prevents optimizations (back-off, using a Futex on host,…).

MESSAGE PASSING BETWEEN TWO THREADS

Excellent CUDA C++ and host code using libcu++

Global memory

```
__managed__ atomic<bool> flag;
__managed__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    atomic<bool>& flag, int& data, int value
) {
    data = value;
    flag.store(true, memory_order_release);
    flag.notify_all();
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    atomic<bool>& flag, int& data
) {
    flag.wait(false, memory_order_acquire);
    return data;
}
```

MESSAGE PASSING BETWEEN TWO THREADS

Excellent CUDA C++ and host code using libcu++

Global memory

```
__managed__ atomic<bool> flag;
__managed__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    atomic<bool>& flag, int& data, int value
) {
    data = value;
    flag.store(true, memory_order_release);
    flag.notify_all();
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    atomic<bool>& flag, int& data
) {
    flag.wait(false, memory_order_acquire);
    return data;
}
```

MESSAGE PASSING BETWEEN TWO THREADS

Excellent CUDA C++ and host code using libcu++

Global memory

```
__managed__ atomic<bool> flag;
__managed__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    atomic<bool>& flag, int& data, int value
) {
    data = value;
    flag.store(true, memory_order_release);
    flag.notify_all();
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    atomic<bool>& flag, int& data
) {
    flag.wait(false, memory_order_acquire);
    return data;
}
```

What if we just want to synchronize two threads in the device?
Performance: we don’t want to pay for system-wide atomics!

LIBCU++ THREAD SCOPES

CUDA C++ extension over standard C++

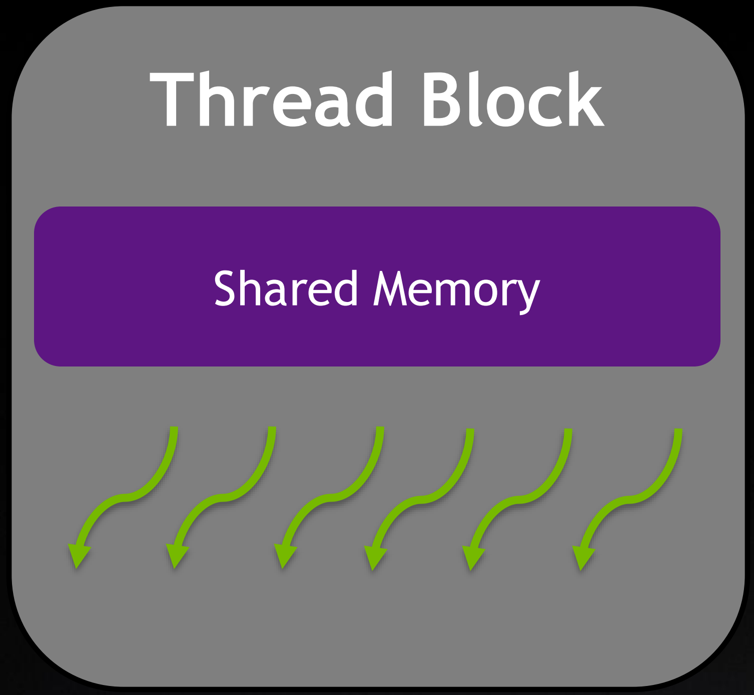
Standard C++ assumes similar synchronization overhead for all threads in system.
Does not hold for heterogeneous systems: __syncthreads vs atomic<thread_scope_system>

CUDA C++ thread scopes specify which threads primitives like atomic synchronize with each other.

Thread

thread_scope_thread

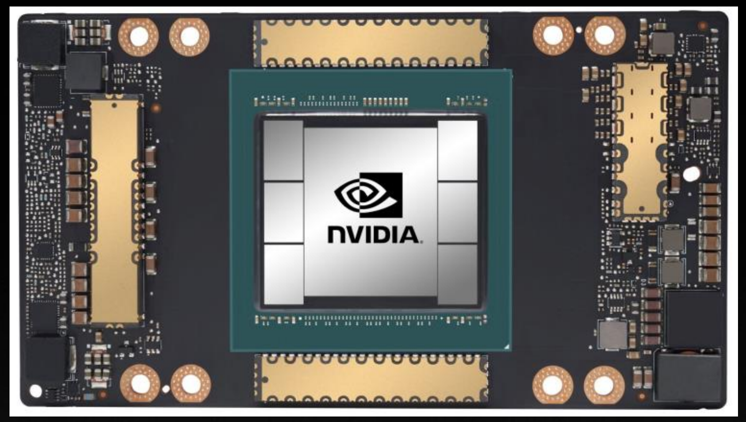
Thread with itself and its async ops (more later)



thread_scope_block

Threads within thread block

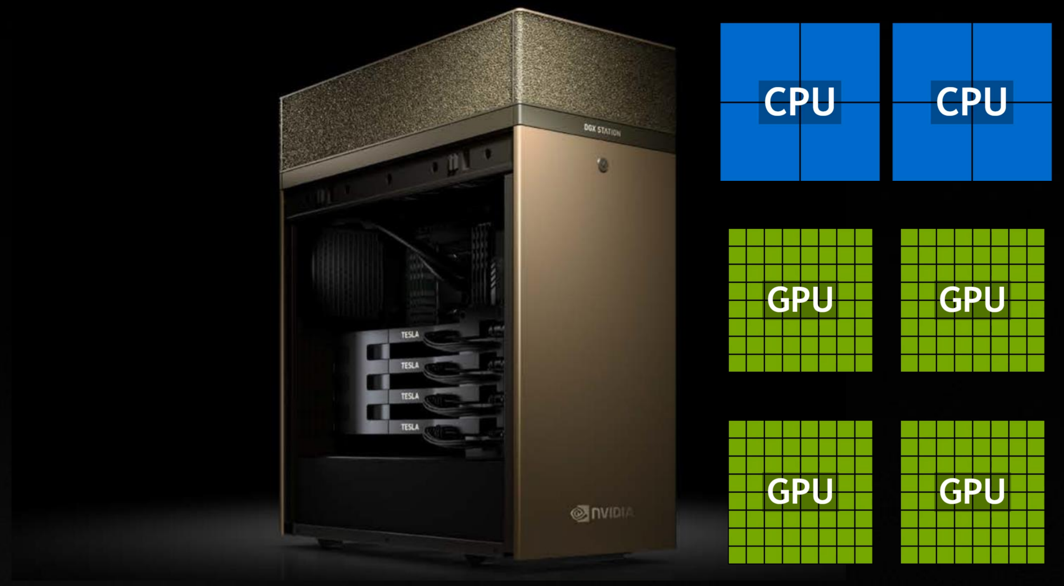
GPU Device



thread_scope_device

Threads within a single GPU

System



thread_scope_system

Threads within a system with multiple CPUs and GPUs

https://nvidia.github.io/libcudacxx/extended_api/thread_scopes.html

MESSAGE PASSING BETWEEN TWO THREADS

Excellent CUDA C++ device-only code

Global memory

```
__device__ cuda::atomic<bool, cuda::thread_scope_device> flag;
__device__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    atomic<bool>& flag, int& data, int value
) {
    data = value;
    flag.store(true, memory_order_release);
    flag.notify_all();
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    atomic<bool>& flag, int& data
) {
    flag.wait(false, memory_order_acquire);
    return data;
}
```

MESSAGE PASSING BETWEEN TWO THREADS

Excellent CUDA C++ device-only code

Global memory

```
__device__ cuda::atomic<bool, cuda::thread_scope_device> flag;
__device__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
   atomic<bool>& flag, int& data, int value
) {
   data = value;
   flag.store(true, memory_order_release);
   flag.notify_all();
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
   atomic<bool>& flag, int& data
) {
   flag.wait(false, memory_order_acquire);
   return data;
}
```

Does not compile: atomic<bool> has thread_scope system != thread_scope_device

MESSAGE PASSING BETWEEN TWO THREADS

Excellent CUDA C++ device-only code

Global memory

```
__device__ cuda::atomic<bool, cuda::thread_scope_device> flag;
__device__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    cuda::atomic<bool,
        cuda::thread_scope_device>& flag,
    int& data, int value
) {
    data = value;
    flag.store(true, memory_order_release);
    flag.notify_all();
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    cuda::atomic<bool,
        cuda::thread_scope_device>& flag,
    int& data
) {
    flag.wait(false, memory_order_acquire);
    return data;
}
```

MESSAGE PASSING BETWEEN TWO THREADS

Excellent CUDA C++ device-only code

Global memory

```
__device__ cuda::atomic<bool, cuda::thread_scope_device> flag;
__device__ int data;
```

Thread 0: writer

```
__host__ __device__
void write_then_signal(
    cuda::atomic<bool,
        cuda::thread_scope_device>& flag,
    int& data, int value
) {
    data = value;
    flag.store(true, memory_order_release);
    flag.notify_all();
}
```

Thread 1: reader

```
__host__ __device__
int poll_then_read(
    cuda::atomic<bool,
        cuda::thread_scope_device>& flag,
    int& data
) {
    flag.wait(false, memory_order_acquire);
    return data;
}
```


NVIDIA

ATOMICS RECAP

DON'T USE VOLATILE FOR SYNCHRONIZATION
is NOT atomic
does NOT synchronize

DON'T USE LEGACY CUDA ATOMICS: atomicAdd, atomicExch, …
relaxed ordering, require appropriate fencing
non-portable to CPU threads

USE STANDARD ATOMICS INSTEAD: std::atomic, cuda::atomic, cuda::std::atomic

You can now use CUDA C++ cuda::[std::]atomics to implement complex synchronization!

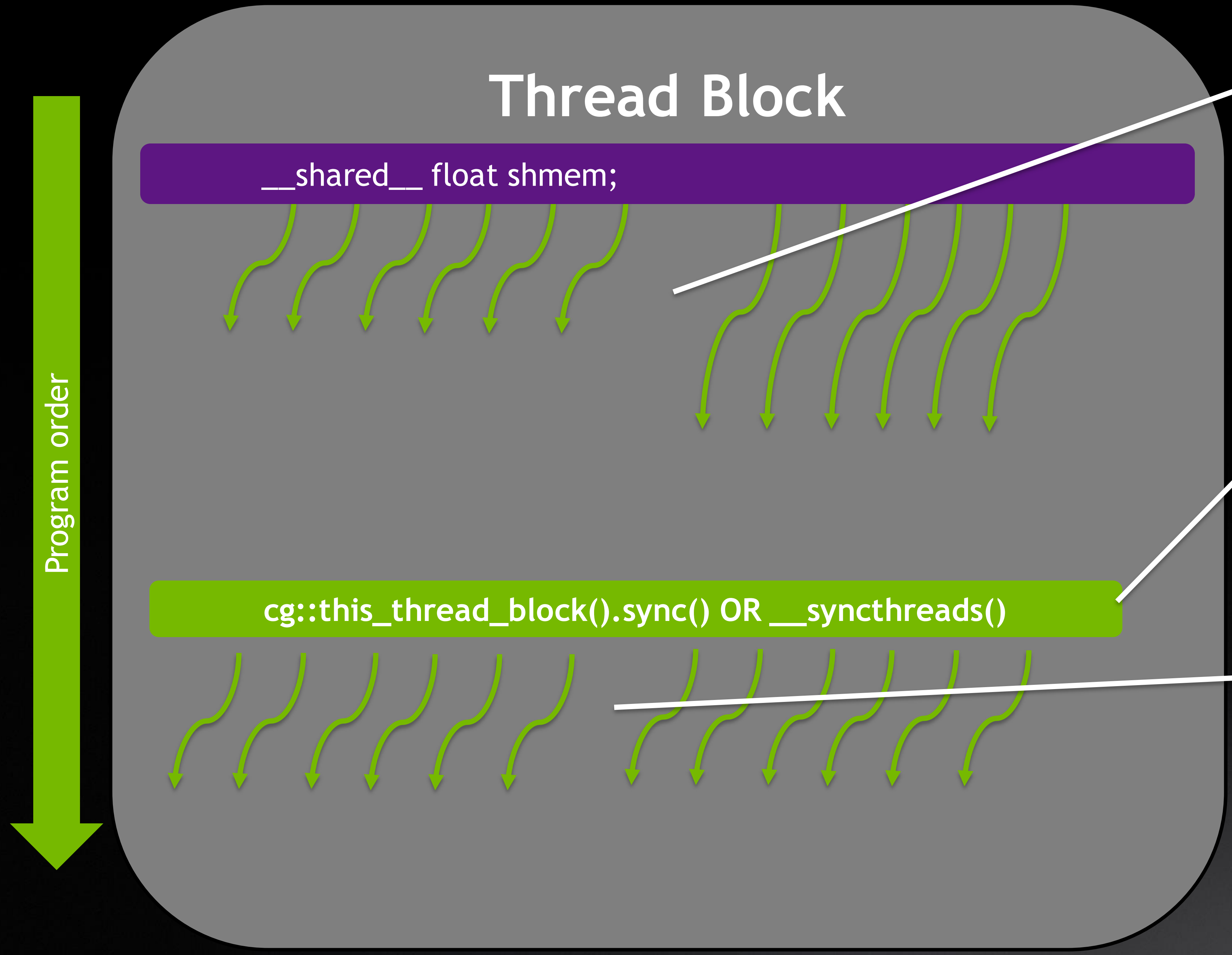
To learn more:

- check out: https://nvidia.github.io/libcudacxx/
- GTC'21: Develop Fast and Safe Concurrent Algorithms with CUDA Memory Model Understanding S31815

ASYNCHRONOUS MEMORY COPY (MEMCPY_ASYNC)

GLOBAL TO SHARED COPY PRE-AMPERE

Common way of copying global memory to shared memory



These threads copy global to shared memory:

`shmem[threadIdx.x] = input[threadIdx.x];`

Synchronize to make shmem writes visible:

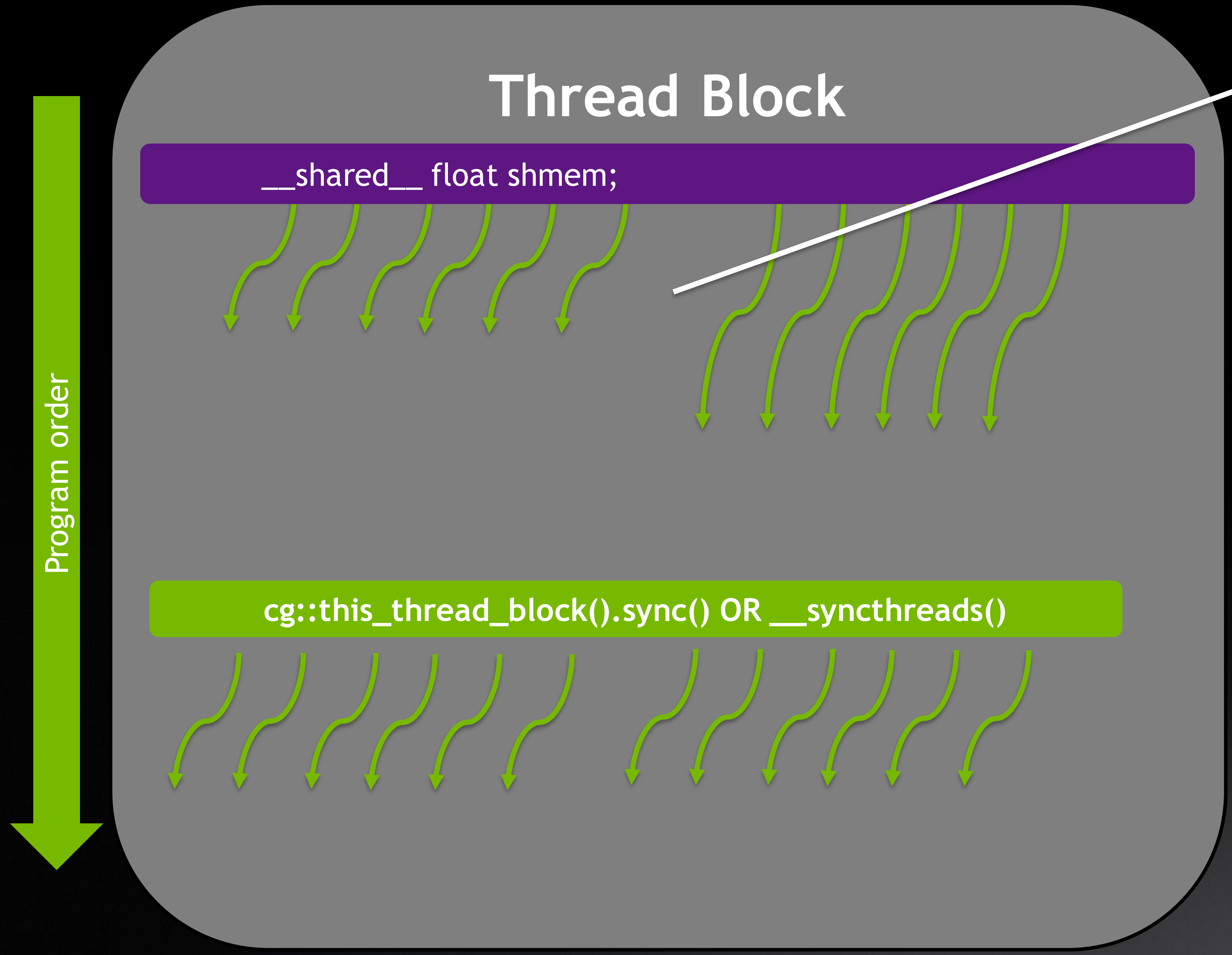
`cg::this_thread_block().sync();`

All threads access shmem to compute:

`compute(shmem);`

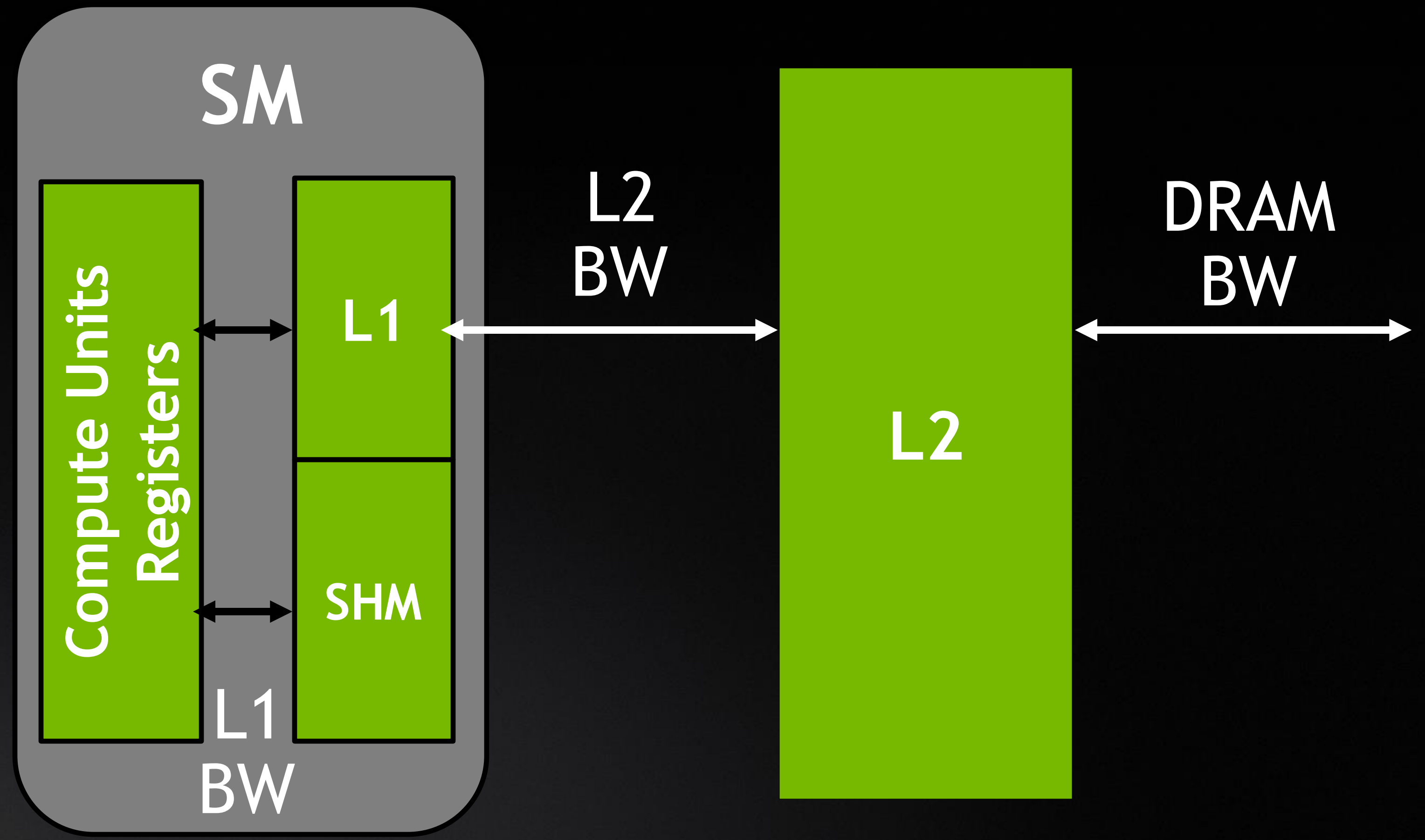
GLOBAL TO SHARED COPY PRE-AMPERE

Common way of copying global memory to shared memory



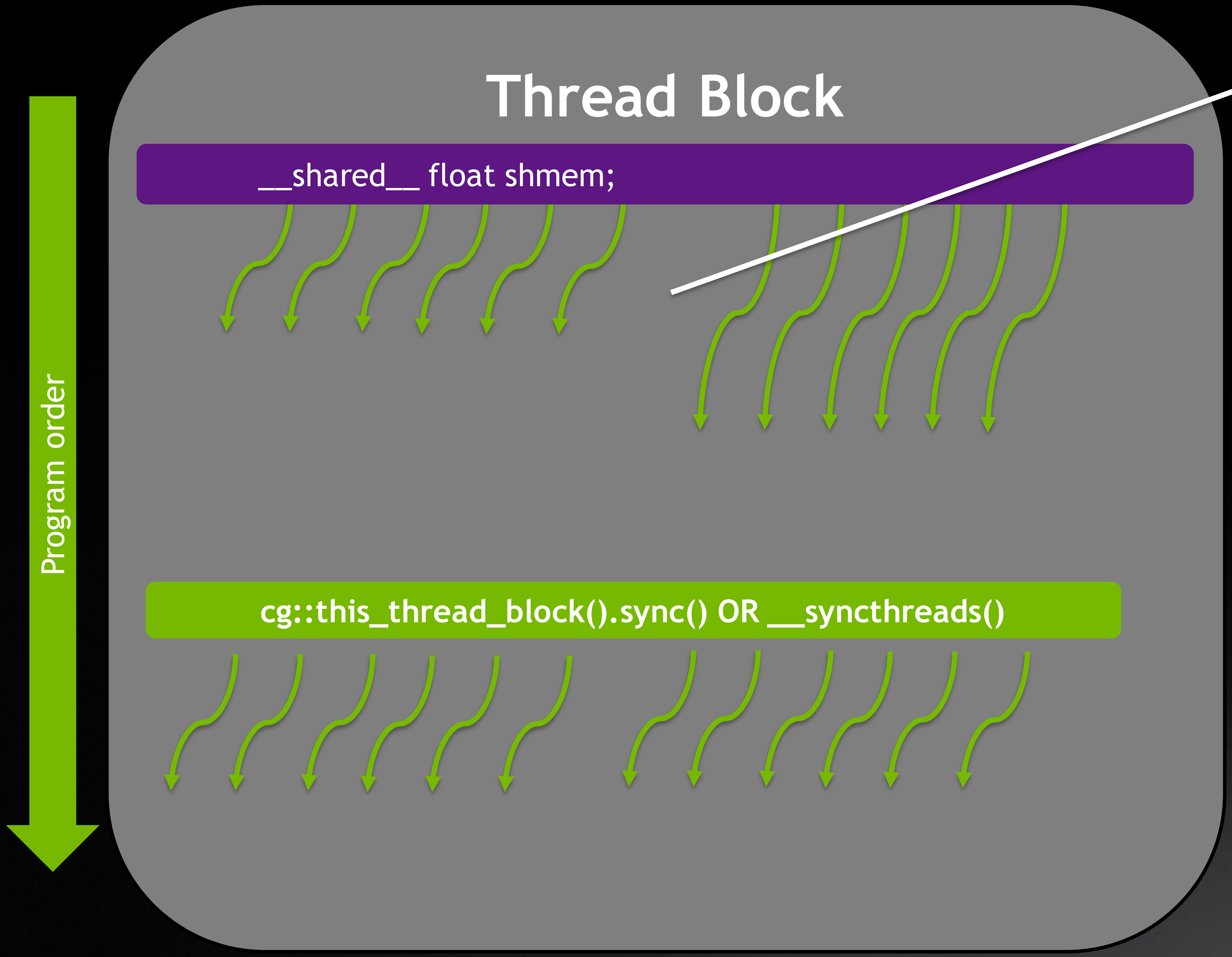
These threads copy global to shared memory:

shmem[threadIdx.x] = input[threadIdx.x];



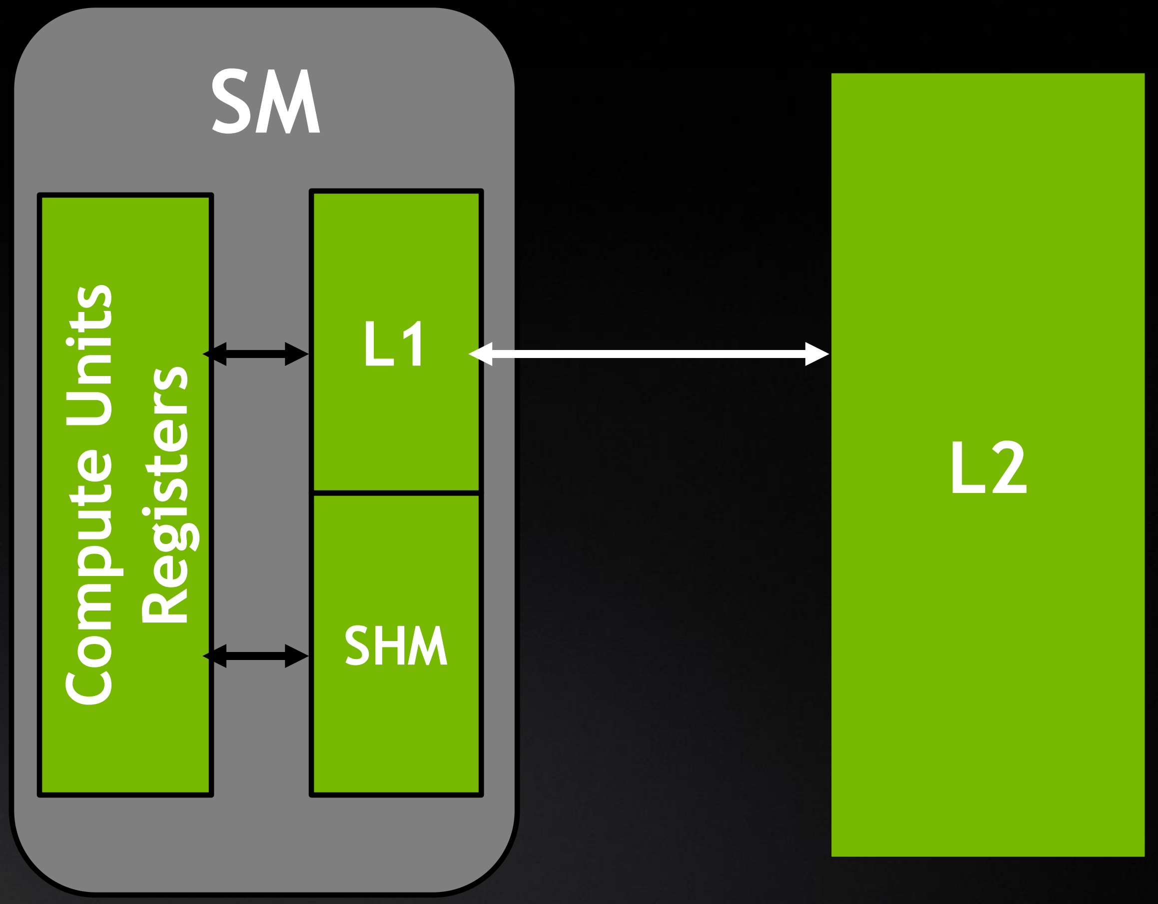
GLOBAL TO SHARED COPY PRE-AMPERE

Common way of copying global memory to shared memory



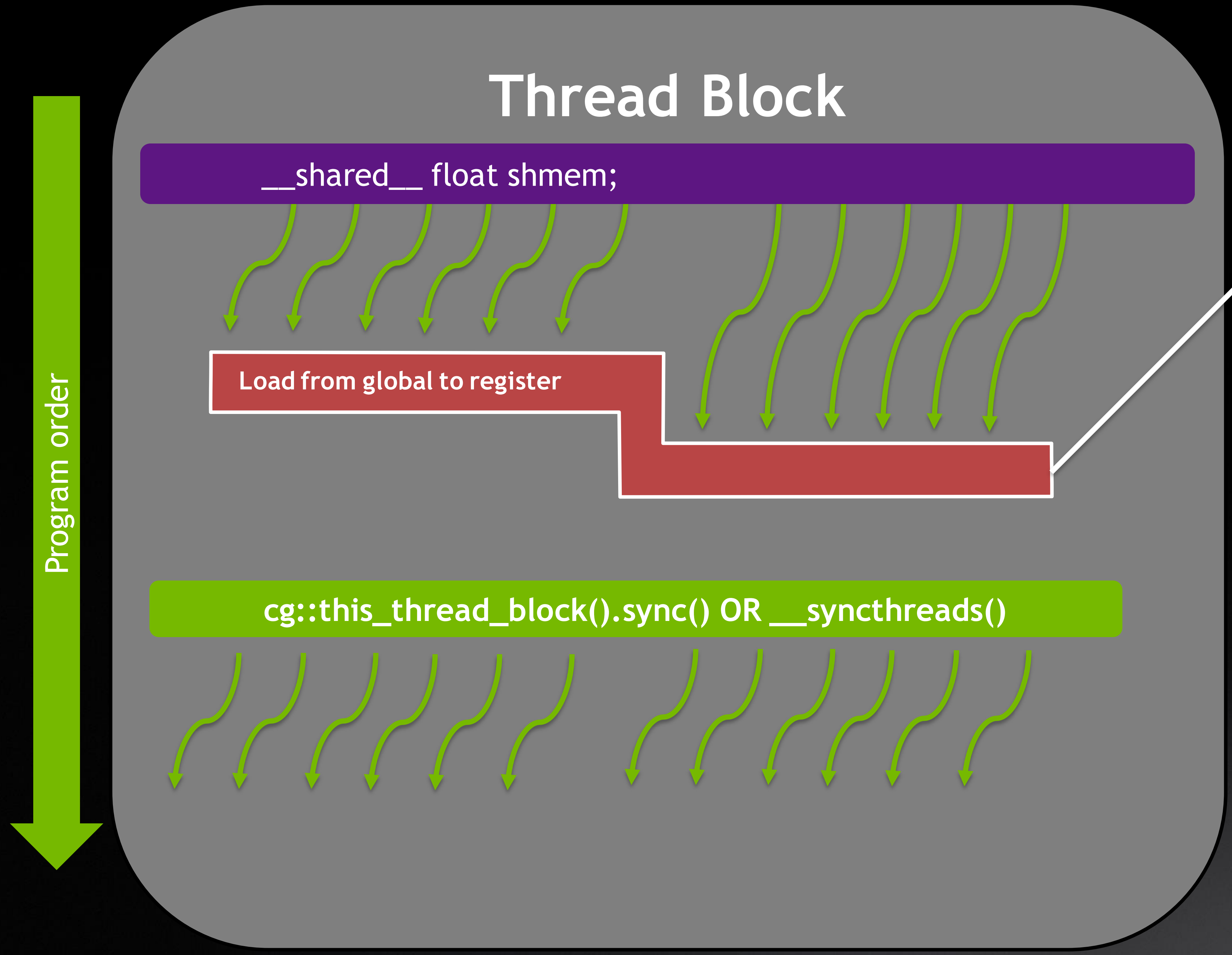
These threads copy global to shared memory:

```
__register = input[threadIdx.x];
shmem[threadIdx.x] = __register;
```



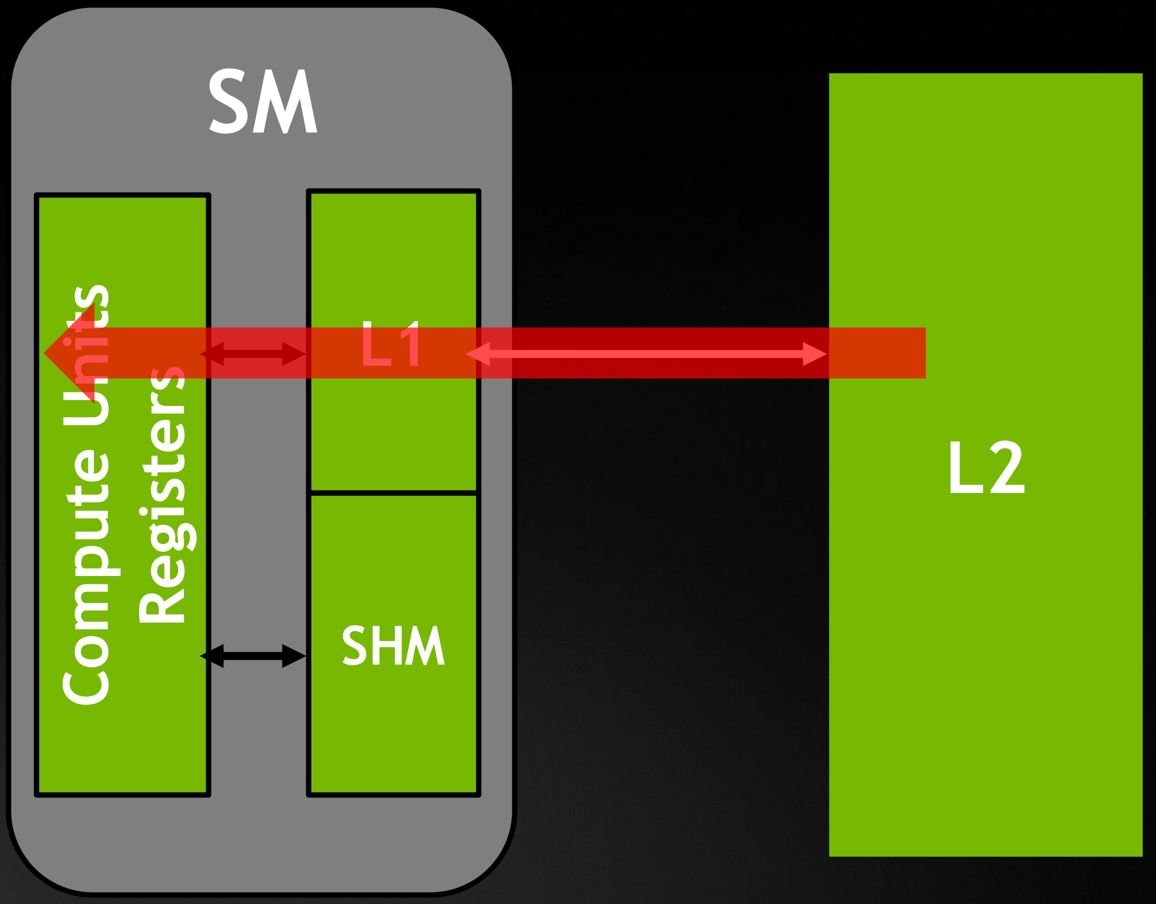
GLOBAL TO SHARED COPY PRE-AMPERE

Common way of copying global memory to shared memory



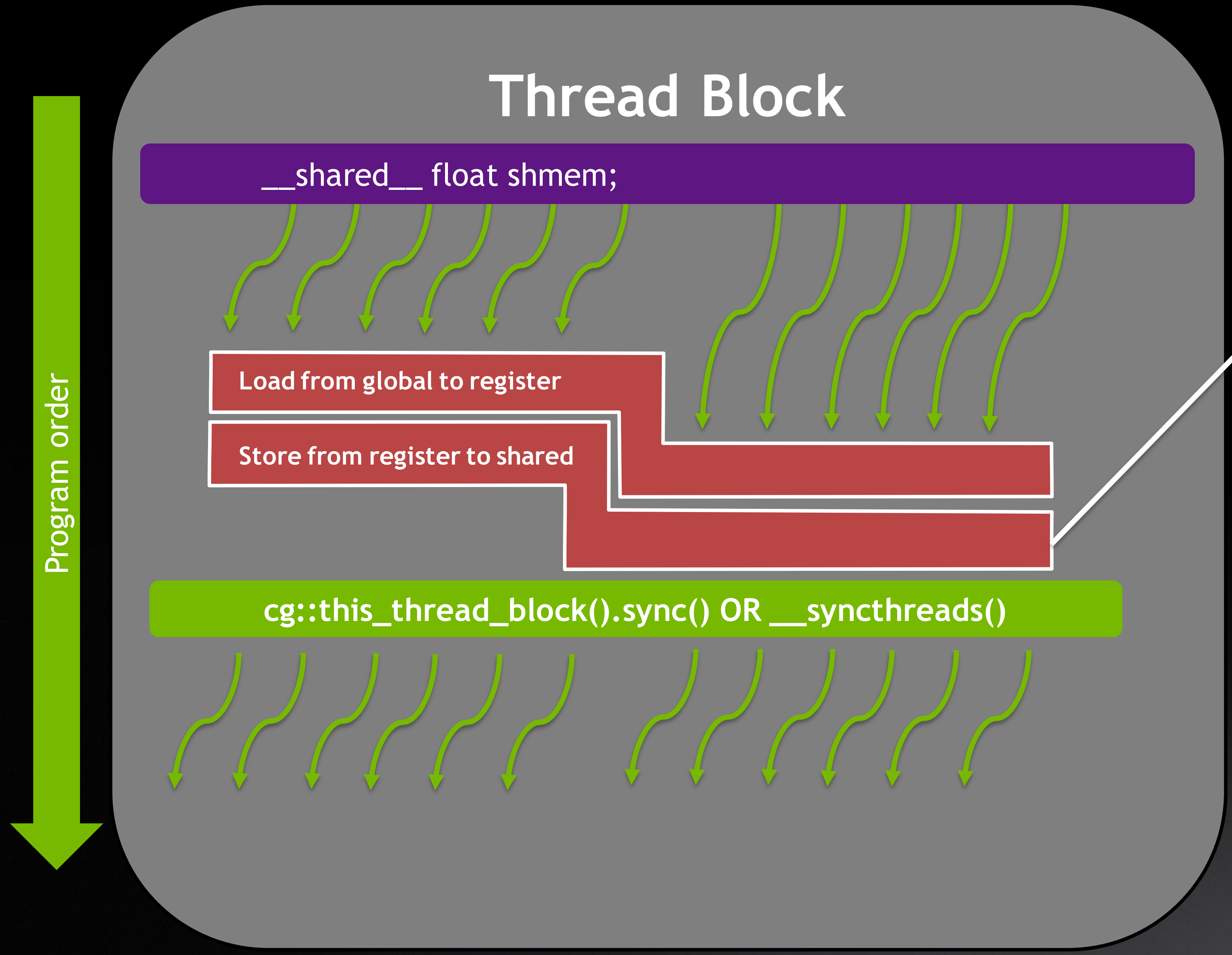
Load from global memory to register:

```
__register = input[threadIdx.x];
shmem[threadIdx.x] = __register;
```



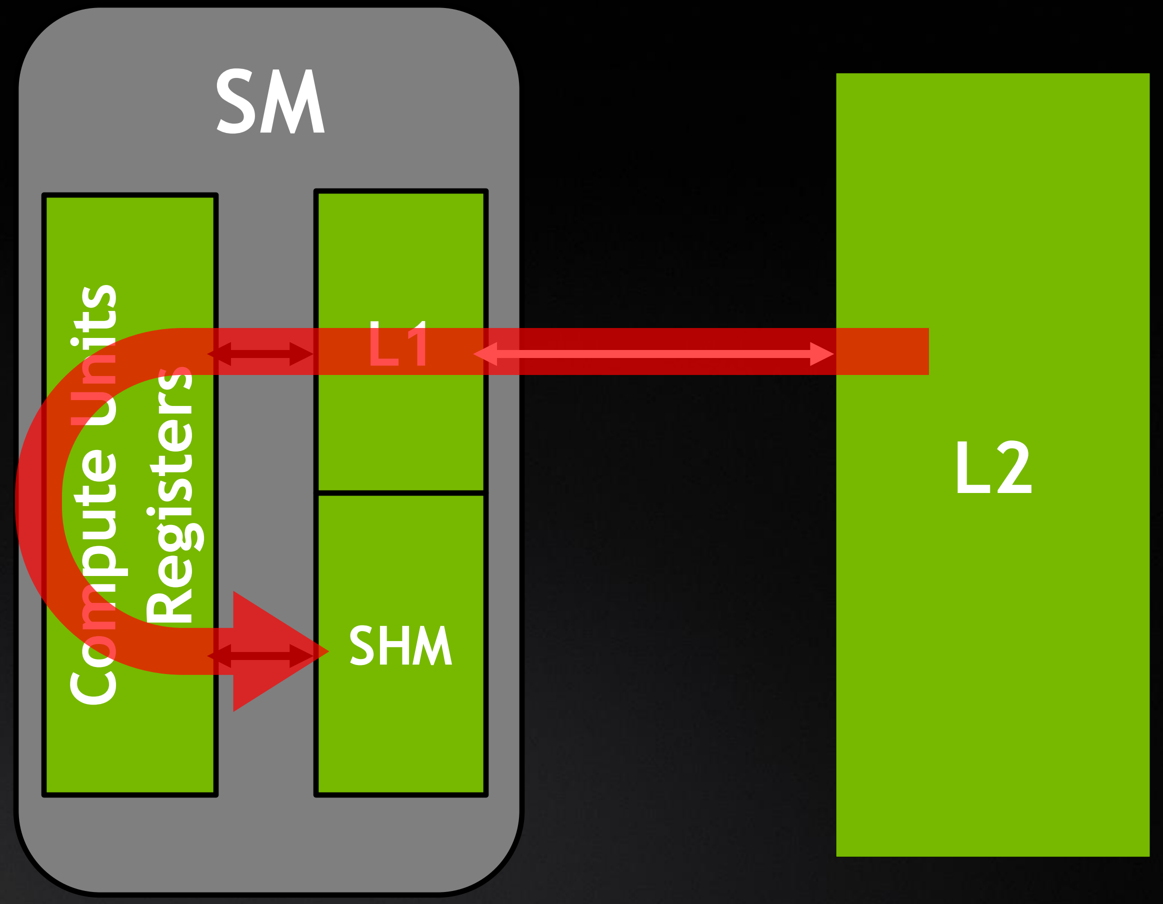
GLOBAL TO SHARED COPY PRE-AMPERE

Common way of copying global memory to shared memory



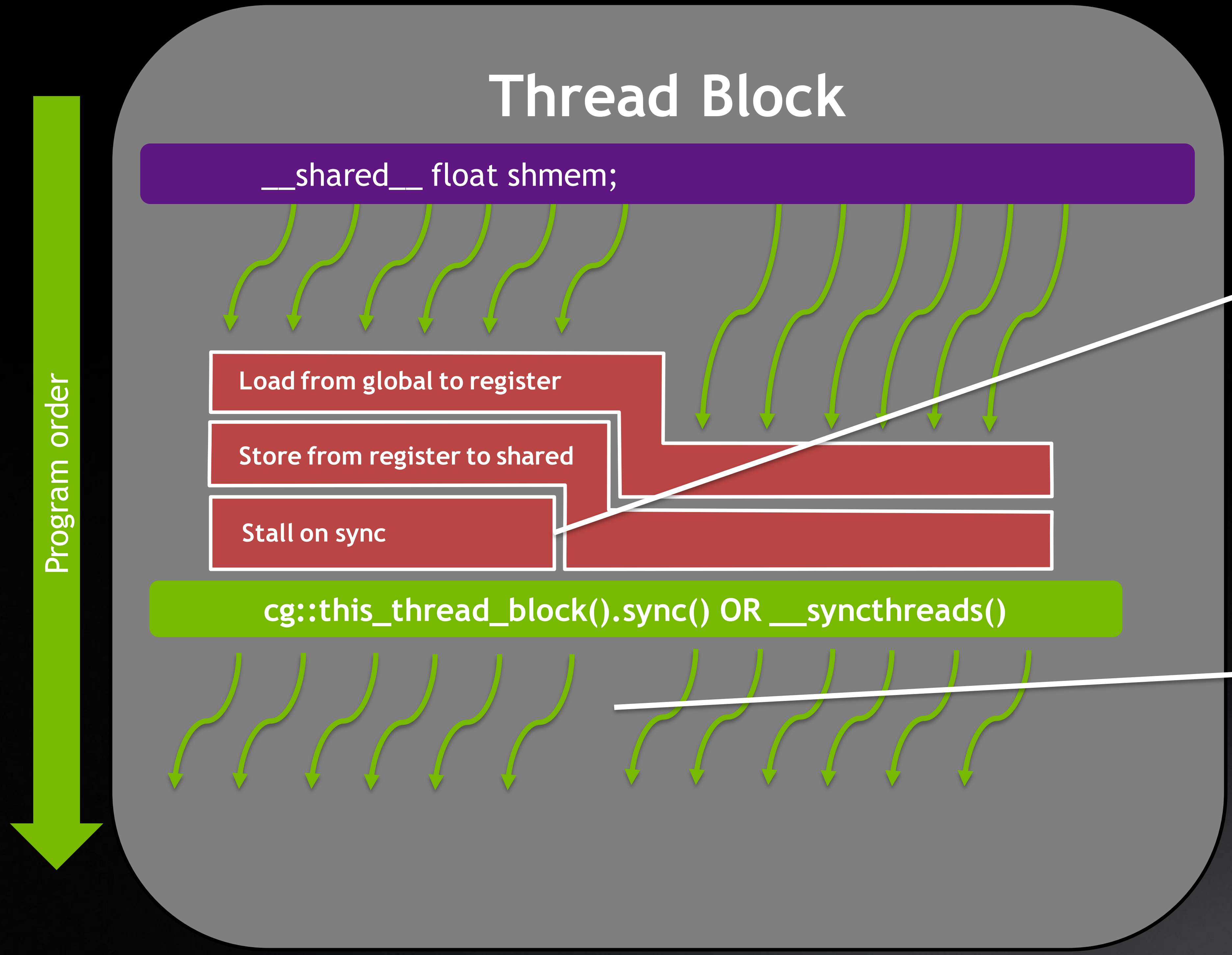
Store from register to shared memory:

```
__register = input[threadIdx.x];
shmem[threadIdx.x] = __register;
```



GLOBAL TO SHARED COPY PRE-AMPERE

Common way of copying global memory to shared memory



Divergent threads stall until stores to shmem become visible to all threads in the block:

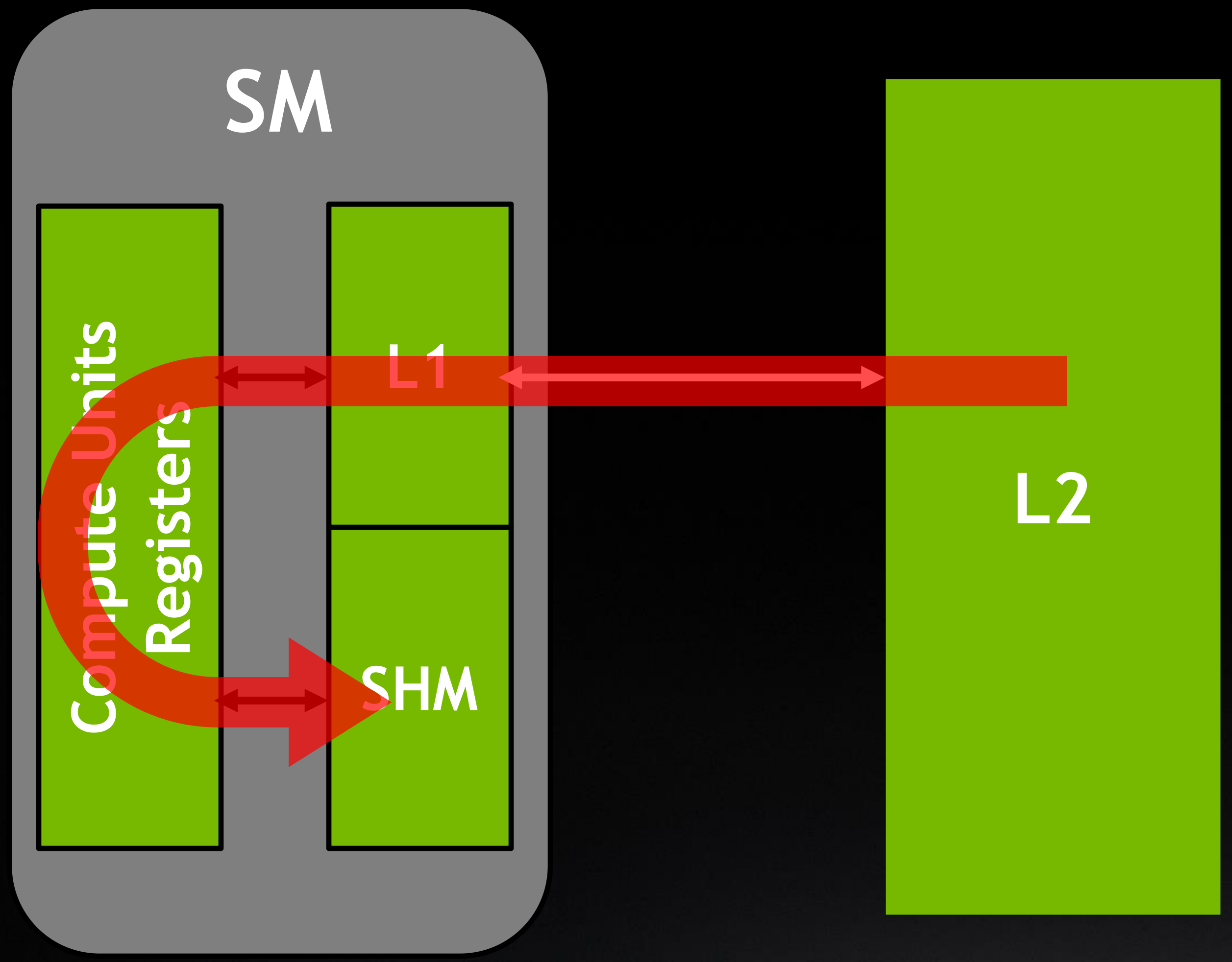
```
cg::this_thread_block().sync();
```

All threads access shmem to compute:

```
compute(shmem);
```

GLOBAL TO SHARED COPY PRE-AMPERE

Drawbacks of the common way of copying global memory to shared memory



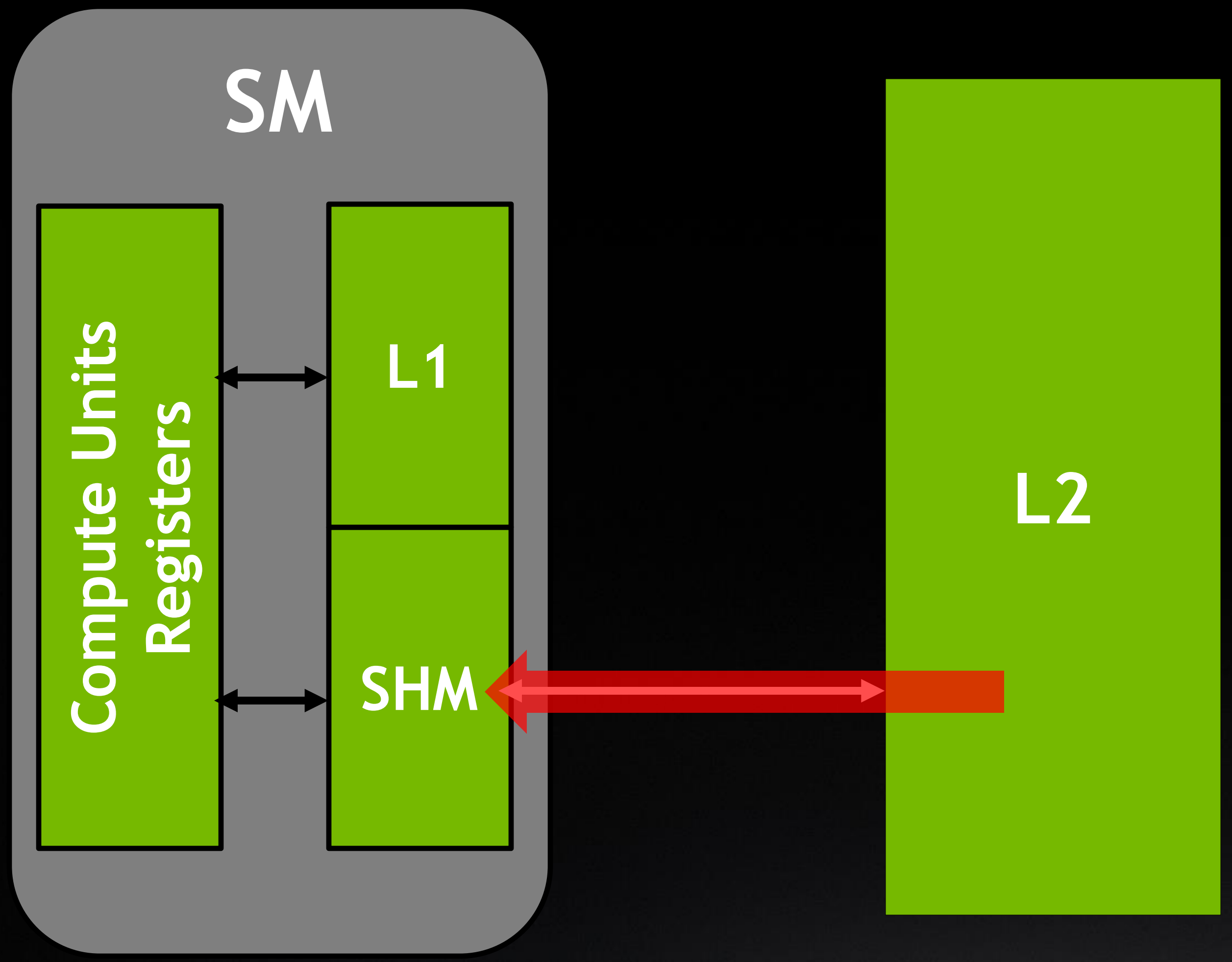
Global to shared copy pre-Ampere:

```
shmem[threadIdx.x] = input[threadIdx.x];
```

- Con: Consumes registers
- Con: Consumes L1/Shared Memory bandwidth

ASYNCHRONOUS MEMORY COPY

Ampere introduces hardware-accelerated asynchronous copy from global to shared memory



Copies directly and asynchronously to shmem:

- Pro: no registers
- Pro: no L1/shared memory bandwidth
- Pro: simplifies multi-stage pipelines
- Con: consumes shared memory for multi-stage

Shared memory has increased significantly on every GPU generation!

ASYNCHRONOUS MEMORY COPY

CUDA C++ memcpy_async collective APIs

Group: set of threads involved in collective operation, e.g., `cg::this_thread_block()`.

Shape: shape of the memory to copy, e.g., `size_t` as #bytes for 1D copy.

ASYNCHRONOUS MEMORY COPY

CUDA C++ memcpy_async collective APIs

Group: set of threads involved in collective operation, e.g., cg::this_thread_block().

Shape: shape of the memory to copy, e.g., `size_t` as #bytes for 1D copy.

The cooperative_groups API:

```
void cg::memcpy_async(Group, dst*, src*, Shape);
void cg::wait(Group);          // group sync, makes all memcpy visible
void cg::wait_prior<N>(Group) // group sync, makes Nth memcpy visible
```

ASYNCHRONOUS MEMORY COPY

CUDA C++ memcpy_async collective APIs

Group: set of threads involved in collective operation, e.g., cg::this_thread_block().

Shape: shape of the memory to copy, e.g., `size_t` as #bytes for 1D copy.

The cooperative_groups API:

```
void cg::memcpy_async(Group, dst*, src*, Shape);
void cg::wait(Group);          // group sync, makes all memcpy visible
void cg::wait_prior<N>(Group) // group sync, makes Nth memcpy visible
```

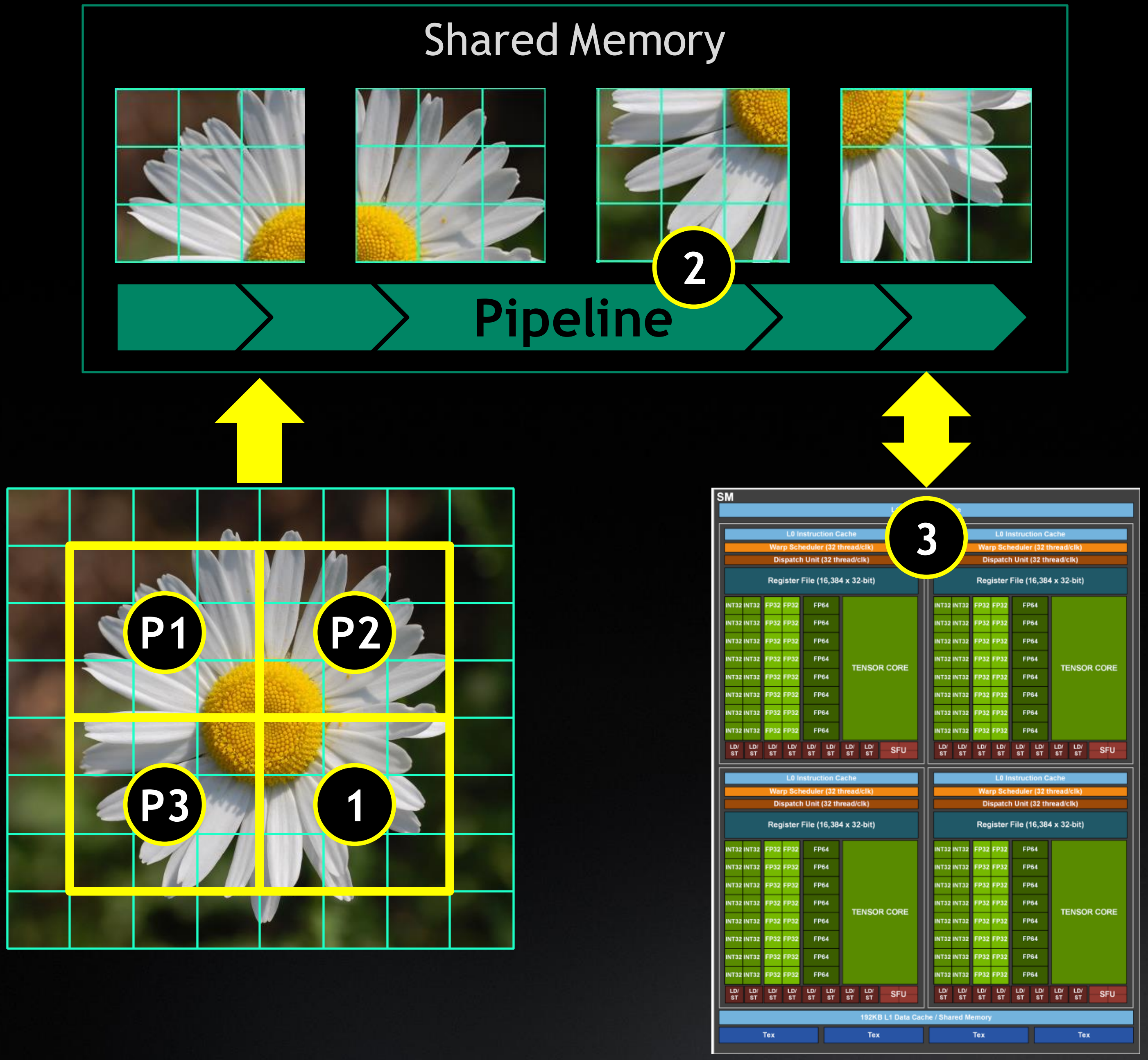
The CUDA APIs with synchronization primitives:

```
cuda::memcpy_async(Group, dst*, src*, Shape, cuda::barrier);

cuda::memcpy_async(Group, dst*, src*, Shape, cuda::pipeline);
```

OVERLAPPING DATA TRANSFER WITH COMPUTATION

ASYNCHRONOUS COPY PIPELINES

Prefetch multiple images in a continuous stream



P1 P2 P3 **Async copy** multiple elements into shared memory

1 **Async copy next** element into shared memory

2 Threads **synchronize** with **oldest** pipelined copy

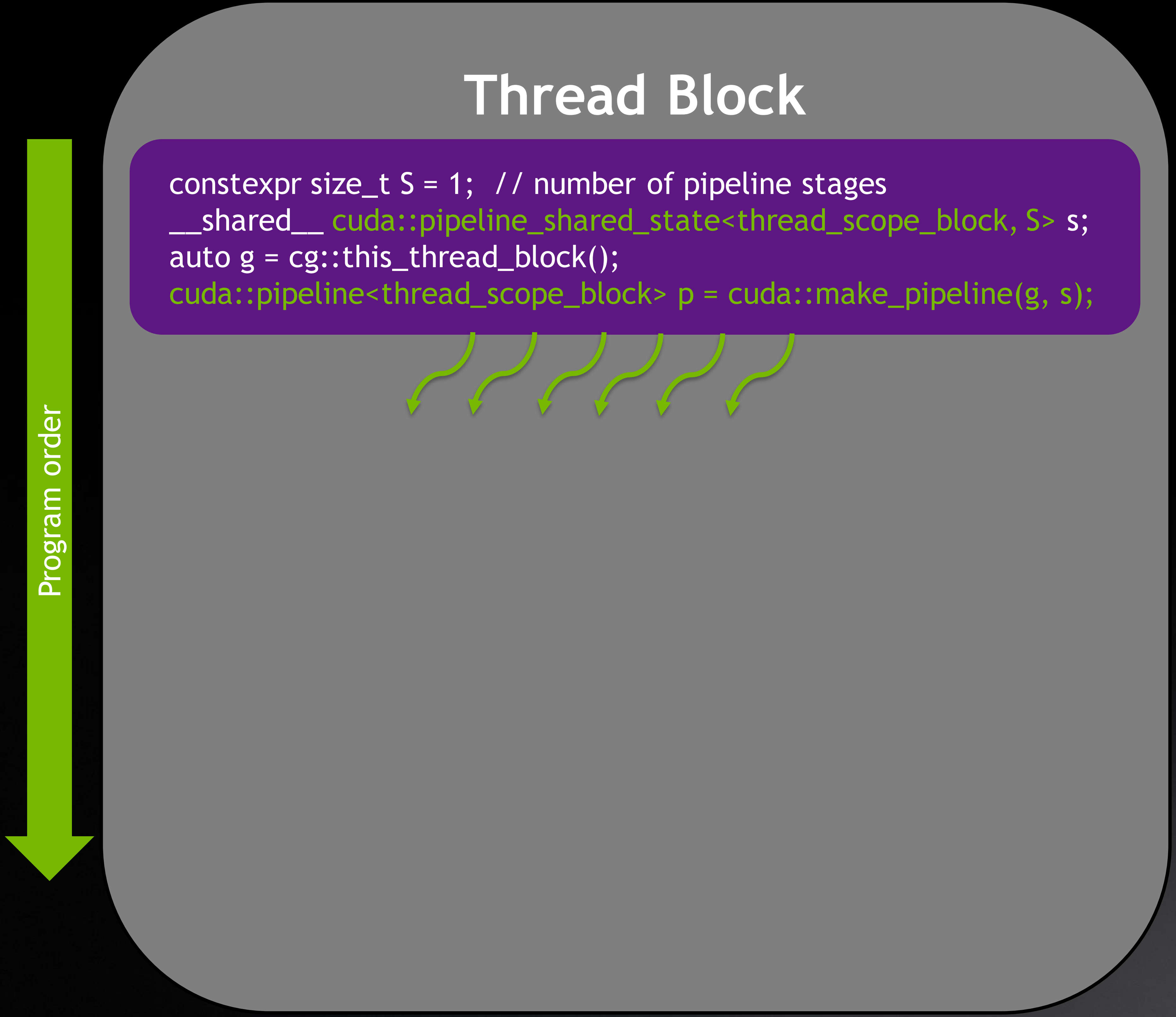
3 Compute using shared memory data

4 Repeat for next element

For more information see: S21170 - CUDA on NVIDIA GPU Ampere Architecture, Taking your algorithms to the next level of performance

MANAGING ASYNCHRONOUS COPIES

Issuing multiple asynchronous operations with cuda::pipeline



cuda::pipeline initialization:

pipeline_shared_state<Scope, StagesCount>
storage to coordinate threads participating in the pipeline.

cuda::make_pipeline:

- Creates a pipeline and initializes the shared state.
- Must be invoked by every thread participating in the pipeline.
- Synchronizes all participating threads.

MANAGING ASYNCHRONOUS COPIES

Issuing multiple asynchronous operations with cuda::pipeline

Program order

Thread Block

```
constexpr size_t S = 1;  // number of pipeline stages
__shared__ cuda::pipeline_shared_state<thread_scope_block, S> s;
auto g = cg::this_thread_block();
cuda::pipeline<thread_scope_block> p = cuda::make_pipeline(g, s);
```

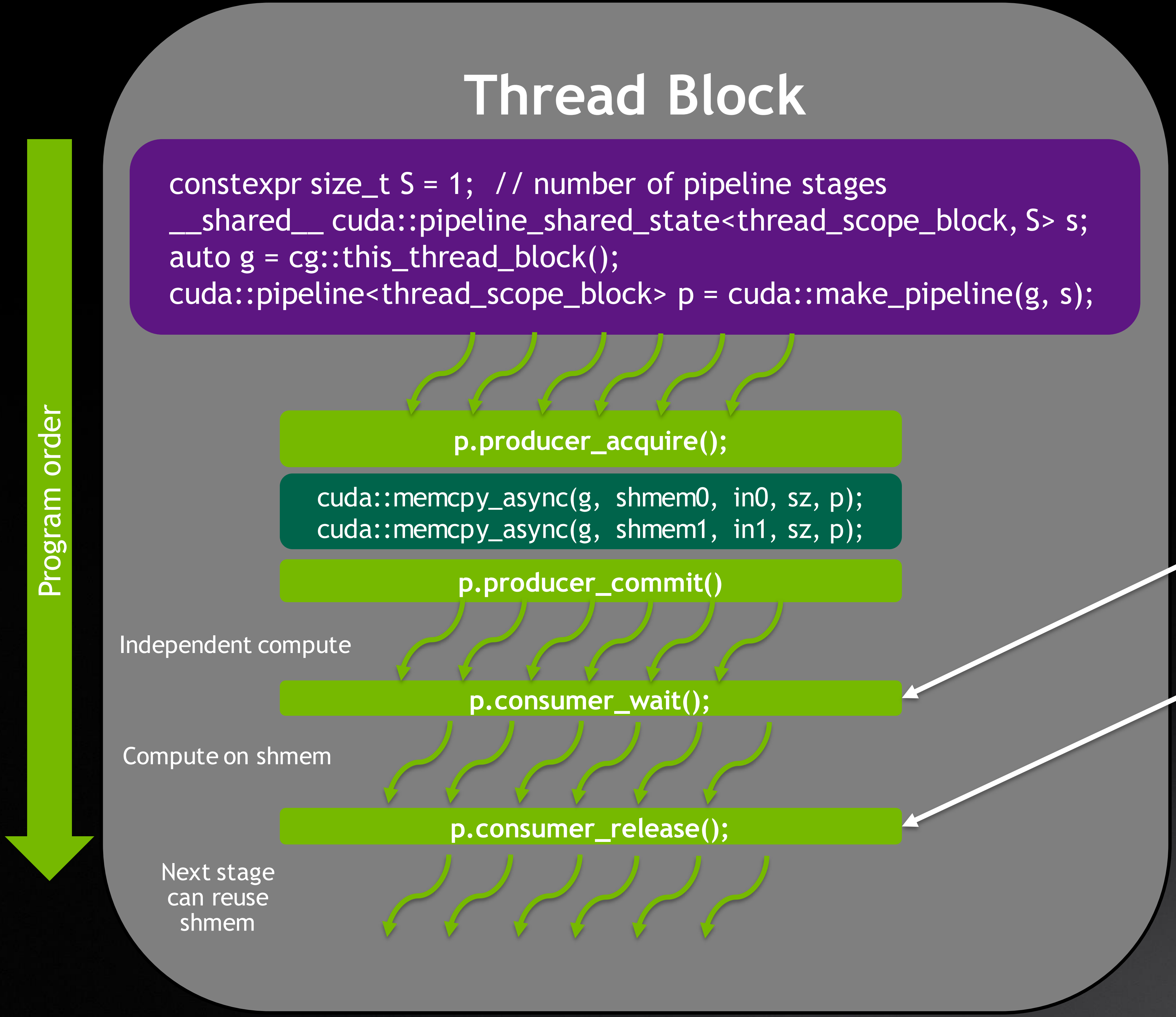
cuda::pipeline introduces:

1. Producer threads: produce async ops.
2. Consumer threads: consume async ops.

In this example all threads are both producer and consumers of asynchronous operations.

MANAGING ASYNCHRONOUS COPIES

Issuing multiple asynchronous operations with cuda::pipeline



Threads interact with cuda::pipeline as follows:

1. Acquire the oldest available pipeline stage.
2. Commit currently staged operations to the pipeline head.
3. Wait for the previously committed operation to complete.
4. Release the pipeline stage for reuse.

cuda::pipeline allows committing multiple asynchronous operations at once!

MULTI-STAGE CUDA::PIPELINE

MULTI-STAGE PIPELINE

Overlapping data transfers with computation using cuda::pipeline

Program order

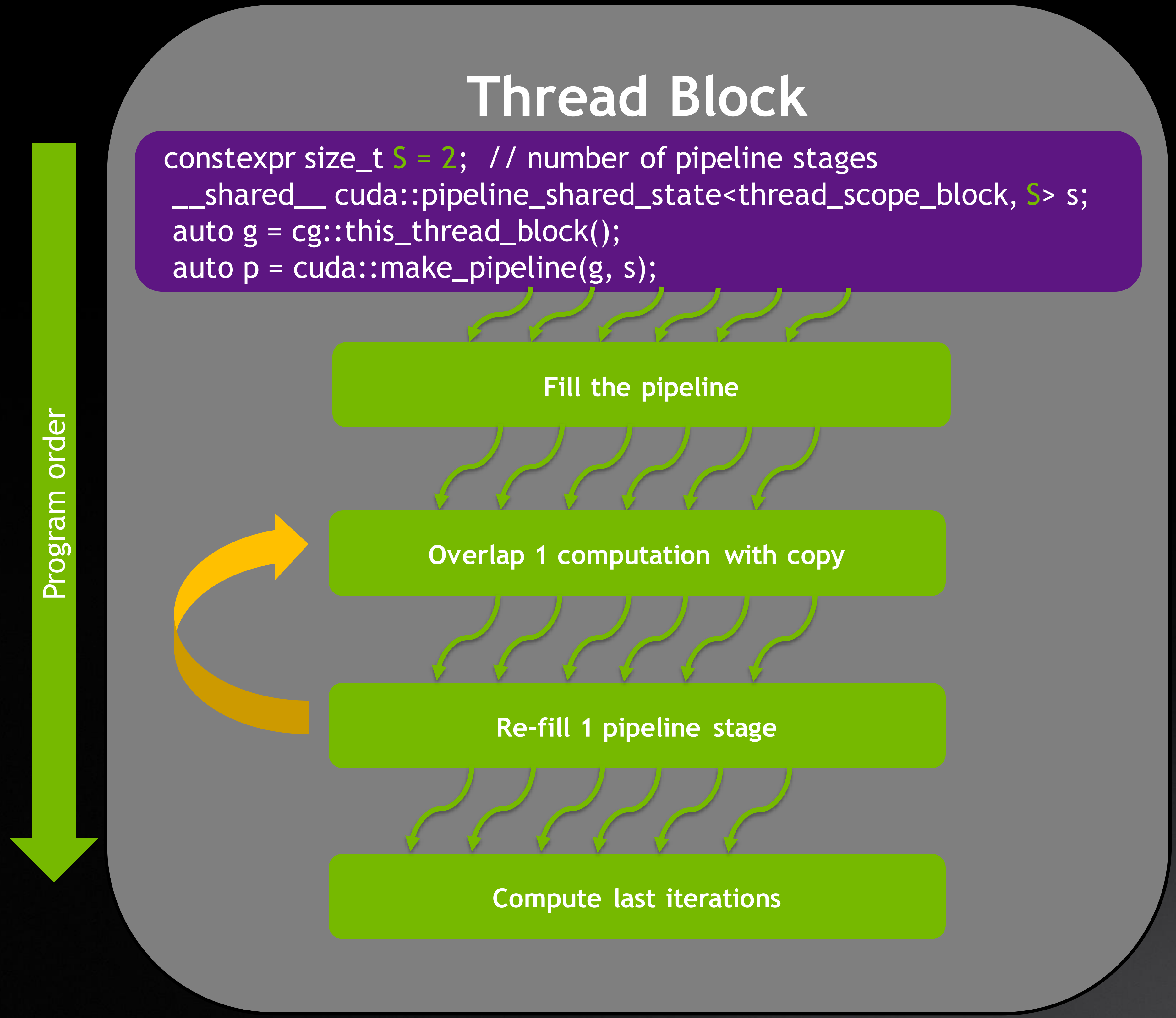
Thread Block

```
constexpr size_t S = 2;  // number of pipeline stages
 __shared__ cuda::pipeline_shared_state<thread_scope_block, S> s;
 auto g = cg::this_thread_block();
 auto p = cuda::make_pipeline(g, s);
```

Multi-stage cuda::pipeline: initialization.

MULTI-STAGE PIPELINE

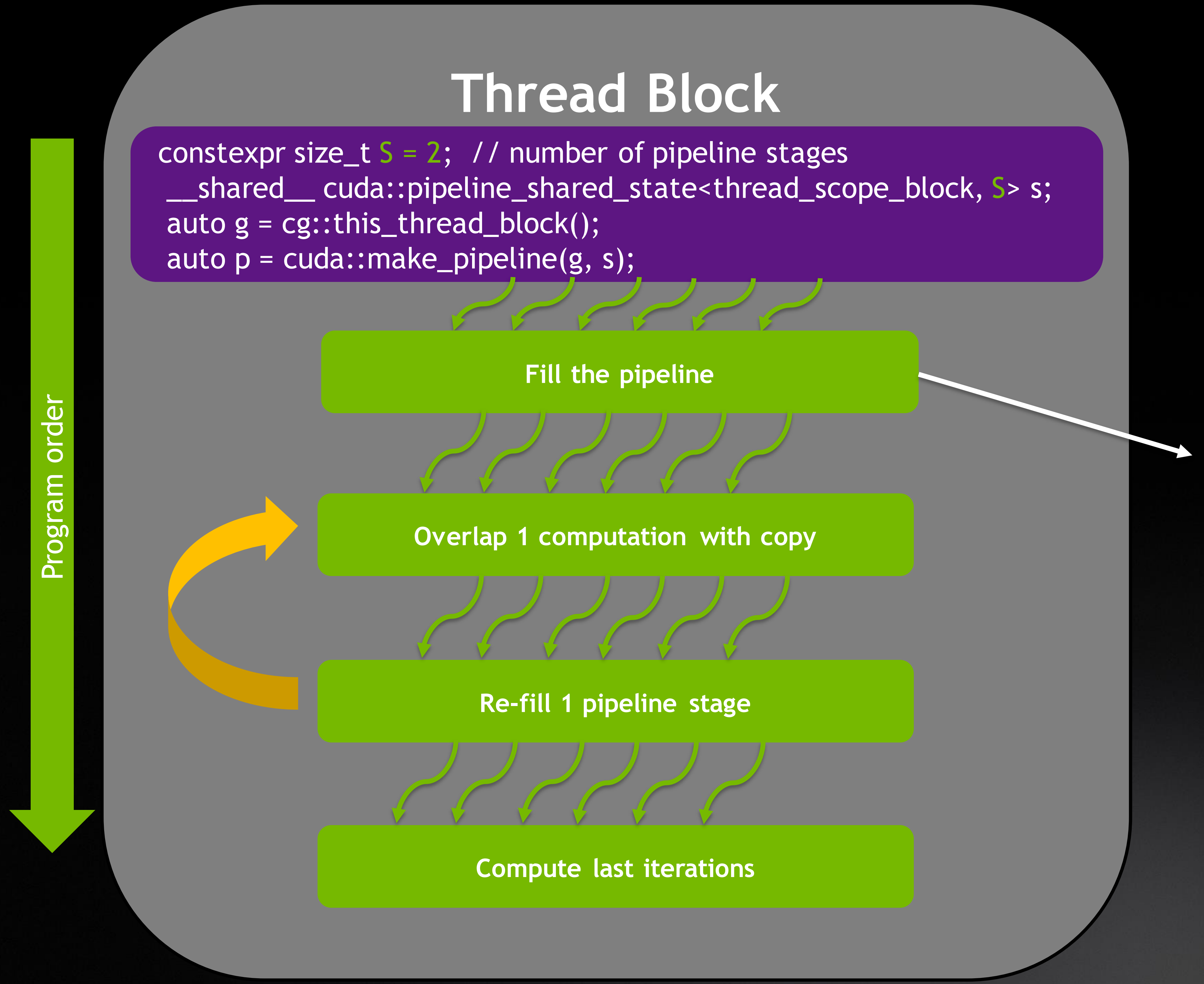
Overlapping data transfers with computation using cuda::pipeline



Multi-stage cuda::pipeline

MULTI-STAGE PIPELINE

Overlapping data transfers with computation using cuda::pipeline

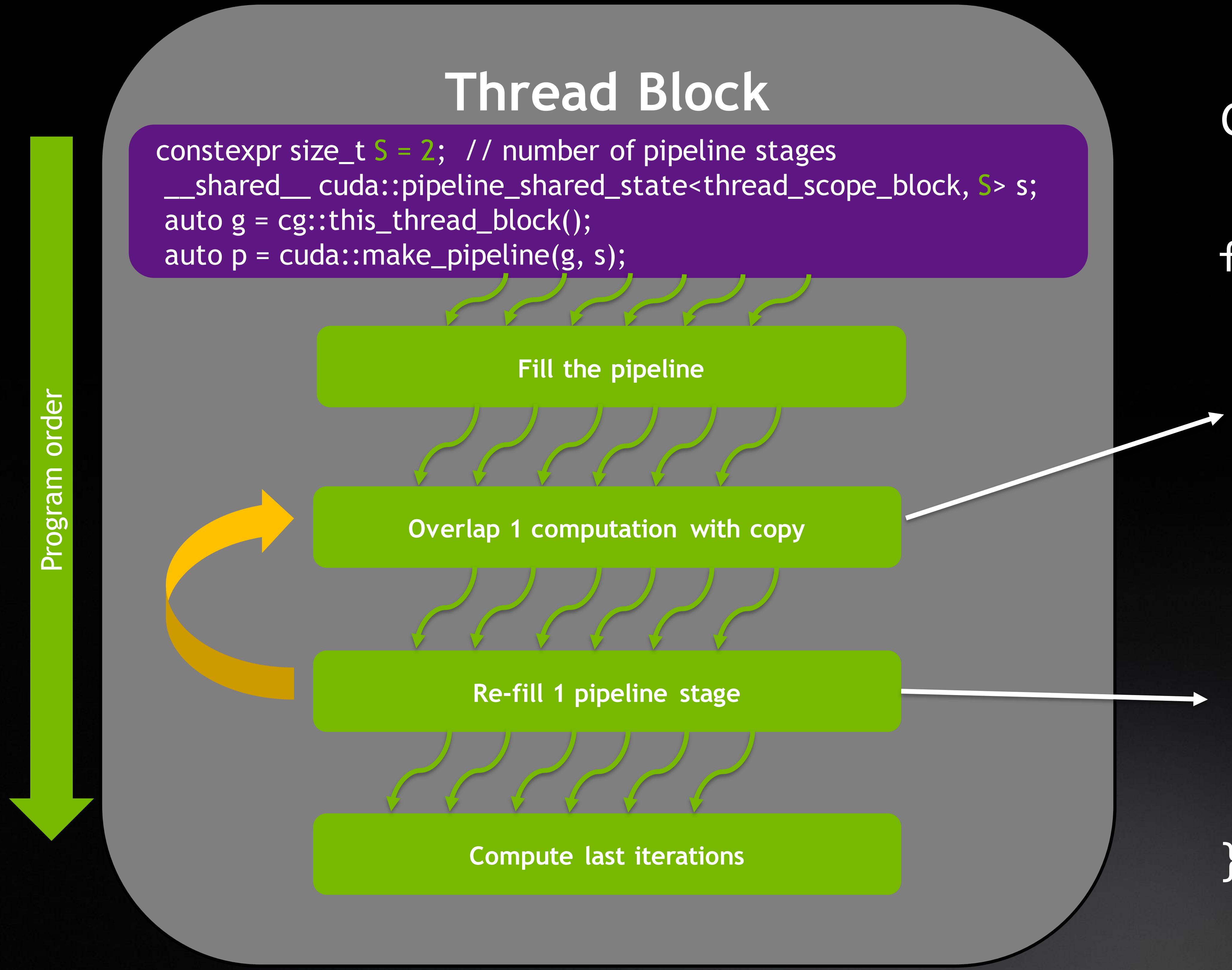


Fill the cuda::pipeline (prologue)

```
for (int s = 0; s < S; ++s) {
  pipeline.producer_acquire();
  // …issue memcpy_asyncs…
  pipeline.producer_commit();
}
```

MULTI-STAGE PIPELINE

Overlapping data transfers with computation using cuda::pipeline

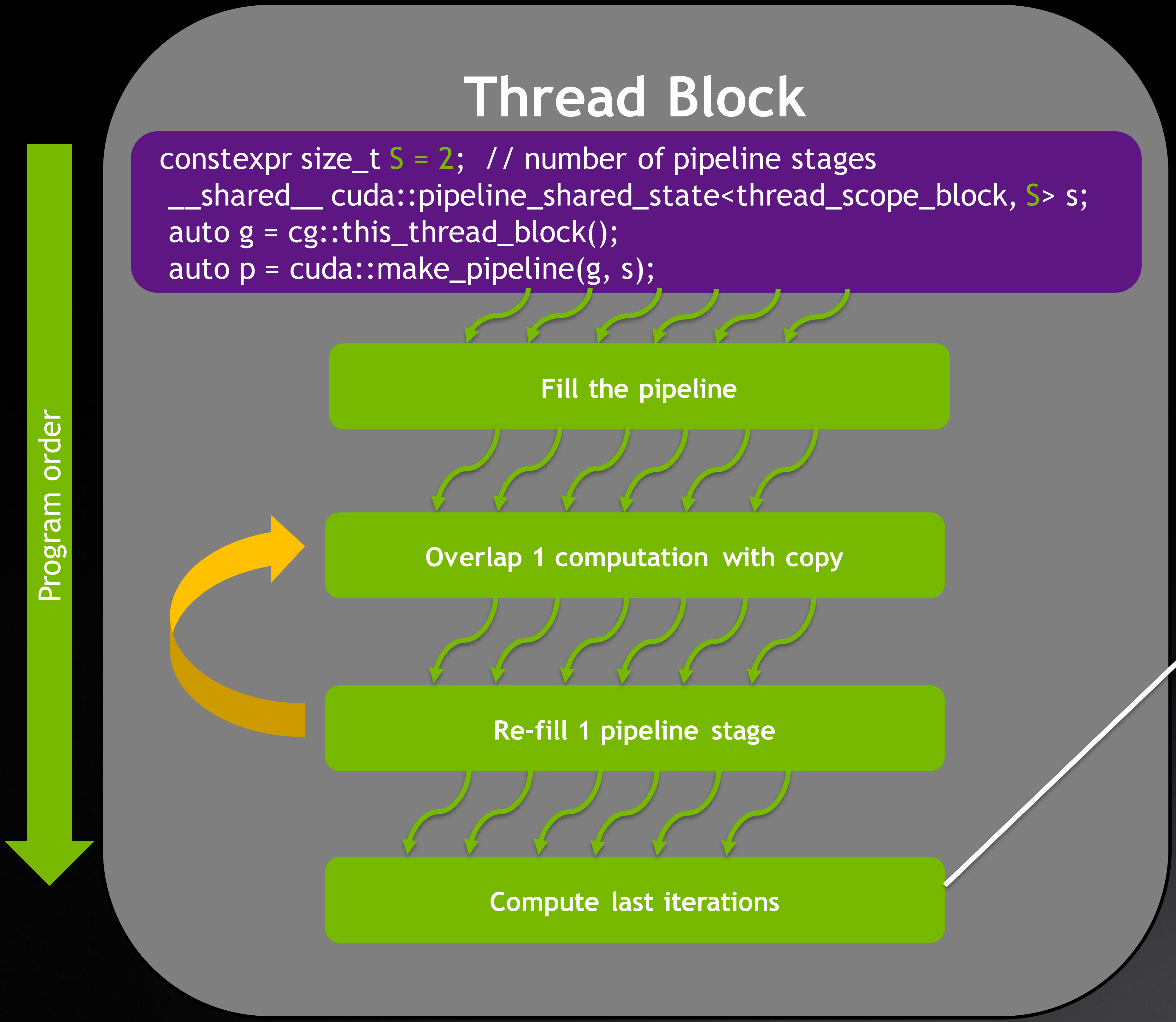


Compute and re-fill (body)

```
for (int i = 0; s < N-S; ++i) {
  pipeline.consumer_wait();
  // …compute this batch…
  pipeline.consumer_release();

  pipeline.producer_acquire();
  // …issue memcpy_asyncs…
  pipeline.producer_commit();
}
```

MULTI-STAGE PIPELINE

Overlapping data transfers with computation using cuda::pipeline



Compute last iterations (epilogue)

```
for (int i = 0; s < N-S; ++i) {
  pipeline.consumer_wait();
  // …compute this batch…
  pipeline.consumer_release();
}
```

MANAGING ASYNCHRONOUS COPIES

Multi-stage pipeline: overview

Initalization

```
constexpr size_t stages_count = 2;     // Pipeline with two stages
extern __shared__ int shared[];        // stages_count * block.size() * sizeof(int) bytes
size_t shared_offset[stages_count] = { 0, block.size() }; // Offsets to each batch

__shared__ cuda::pipeline_shared_state< // Allocate shared storage for a two-stage cuda::pipeline:
    cuda::thread_scope::thread_scope_block,
    stages_count
> shared_state;
auto pipeline = cuda::make_pipeline(block, &shared_state);
```

Prologue

```
pipeline.producer_acquire();  // Initialize pipeline by submitting copies for the first iteration
cuda::memcpy_async(block, shared + shared_offset[0], global_in + block_batch(0), sizeof(int) * block.size(), pipeline);
pipeline.producer_commit();
```

Body

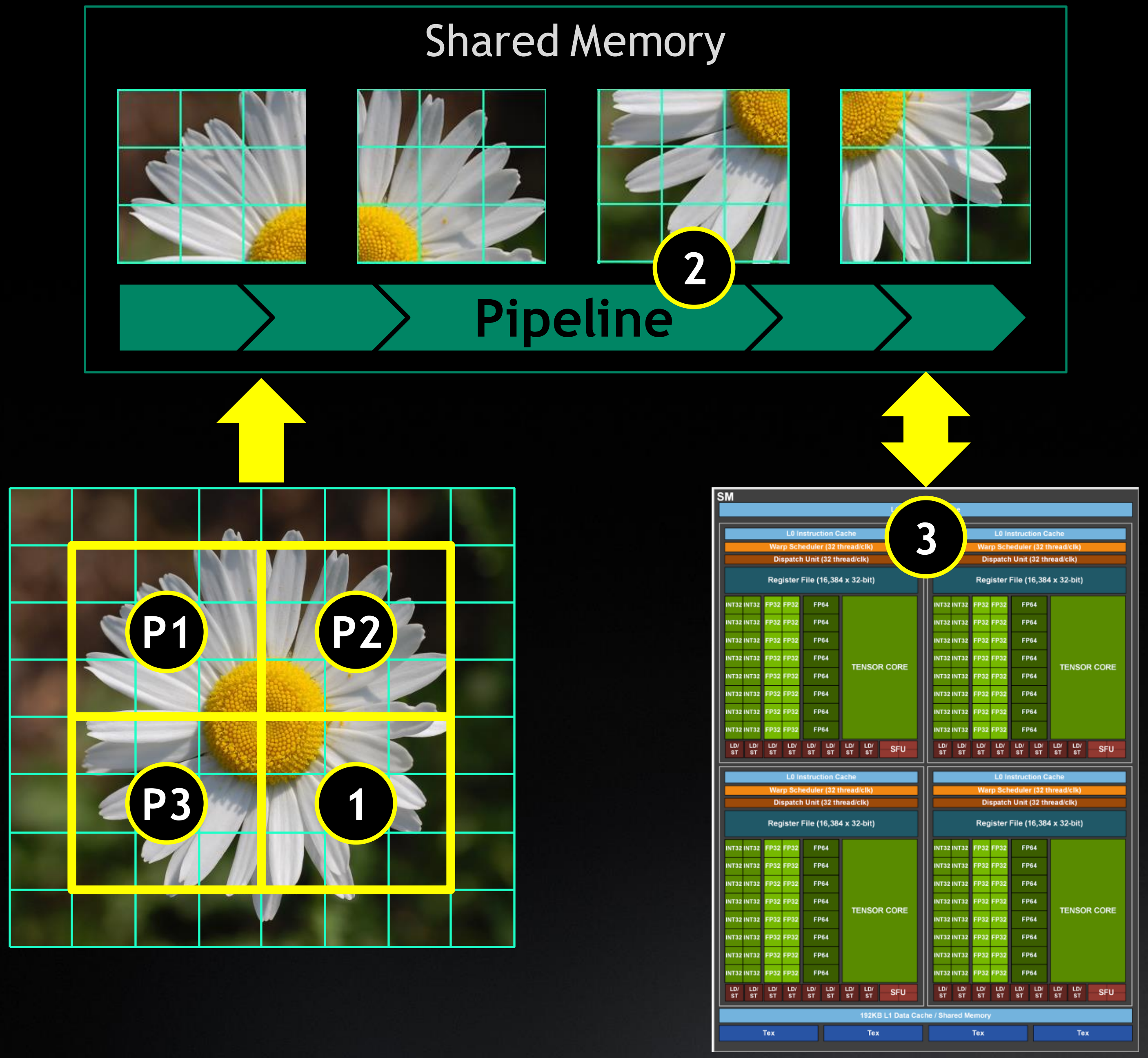
```
for (size_t batch = 1; batch < batch_sz; ++batch) { // Pipelined copy/compute:
    size_t compute_stage_idx = (batch - 1) % 2; // Stage indices for the compute and copy stages:
    size_t copy_stage_idx = batch % 2;
    size_t global_idx = block_batch(batch);

    pipeline.producer_acquire();
    cuda::memcpy_async(block, shared + shared_offset[copy_stage_idx], global_in + global_idx, sizeof(int) * block.size(), pipeline);
    pipeline.producer_commit();

    pipeline.consumer_wait();
    compute(global_out + global_idx, shared + shared_offset[compute_stage_idx]);
    pipeline.consumer_release();
}
```

Epilogue

```
pipeline.consumer_wait();  // Compute last iteration
compute(global_out + block_batch(batch_sz-1), shared + shared_offset[(batch_sz - 1) % 2]);
pipeline.consumer_release();
```

ASYNCHRONOUS COPY PIPELINES

Prefetch multiple images in a continuous stream



P1 P2 P3 **Async copy** multiple elements into shared memory

1 **Async copy next** element into shared memory

2 Threads **synchronize** with **oldest** pipelined copy

3 Compute using shared memory data

4 Repeat for next element

For more information see: S21170 - CUDA on NVIDIA GPU Ampere Architecture, Taking your algorithms to the next level of performance

MANAGING ASYNCHRONOUS COPIES

Multi-stage pipeline: initialization

Initalization

```
constexpr size_t stages_count = 2;    // Pipeline with two stages
extern __shared__ int shared[];       // stages_count * block.size() * sizeof(int) bytes
size_t shared_offset[stages_count] = { 0, block.size() }; // Offsets to each batch

__shared__ cuda::pipeline_shared_state< // Allocate shared storage for a two-stage pipeline:
    cuda::thread_scope::thread_scope_block,
    stages_count
> shared_state;
auto pipeline = cuda::make_pipeline(block, &shared_state);
```

MANAGING ASYNCHRONOUS COPIES

Multi-stage pipeline: prologue & epilogue

Prologue

```
// Initialize pipeline by submitting copies for the first iteration
pipeline.producer_acquire();
cuda::memcpy_async(block, shared + shared_offset[0], global_in + block_batch(0),
                   sizeof(int) * block.size(), pipeline);
pipeline.producer_commit();

… body …
```

Epilogue

```
// Compute remaining iterations
pipeline.consumer_wait();  // Compute last iteration
compute(global_out + block_batch(batch_sz-1), shared + shared_offset[(batch_sz - 1) % 2]);
pipeline.consumer_release();
```

MANAGING ASYNCHRONOUS COPIES

Multi-stage pipeline: body

Body

```
 for (size_t batch = 1; batch < batch_sz; ++batch) { // Pipelined copy/compute:
     size_t compute_stage_idx = (batch - 1) % 2; // Stage indices for the compute and copy
     size_t copy_stage_idx = batch % 2;
     size_t global_idx = block_batch(batch);

     pipeline.producer_acquire();
     cuda::memcpy_async(block, shared + shared_offset[copy_stage_idx], global_in + global_idx,
                        sizeof(int) * block.size(), pipeline);
     pipeline.producer_commit();

     pipeline.consumer_wait();
     compute(global_out + global_idx, shared + shared_offset[compute_stage_idx]);
     pipeline.consumer_release();
}
```

MANAGING ASYNCHRONOUS COPIES

Multi-stage pipeline with fused prologue and epilogue

```
for (size_t subset = 0, fetch = 0; subset < subset_count; ++subset) {



  pipeline.consumer_wait();
  compute(subset % stages_count, subset);
  pipeline.consumer_release();
}
```

See blog post [Controlling data movement to boost performance on Ampere architecture]
by Matthieu Tardy and Carter H. Edwards

MANAGING ASYNCHRONOUS COPIES

Multi-stage pipeline with fused prologue and epilogue

Fused prologue
Fills pipeline and keeps it filled

Fused epilogue

```
for (size_t subset = 0, fetch = 0; subset < subset_count; ++subset) {
  for (; fetch < subset_count && fetch < (subset + stages_count); ++fetch) {
     pipeline.producer_acquire();
     copy(fetch % stages_count, fetch);
     pipeline.producer_commit();
  }
  pipeline.consumer_wait();
  compute(subset % stages_count, subset);
  pipeline.consumer_release();
}
```

See blog post [Controlling data movement to boost performance on Ampere architecture]
by Matthieu Tardy and Carter H. Edwards

OTHER SYNCHRONIZATION PRIMITIVES IN LIBCU++

LIBCU++ SYNC PRIMITIVES

libcu++ synchronization primitives

CUDA 11.3	
cuda::atomic cuda::barrier cuda::latch	cuda::counting_semaphore cuda::binary_semaphore cuda::pipeline

Resources

- CUDA C++ Programming guide:
    - Asynchronous SIMT Programming Model
    - Asynchronous Barrier
    - Asynchronous Data Copies
    - Asynchronous Data Copies with cuda::pipeline
- libcu++ documentation and examples
- GTC'21 talks:
    - Optimizing Applications with asynchronous GPU programming in CUDA C++ E31888
    - Develop Fast and Safe Concurrent Algorithms with CUDA Memory Model Understanding S31815