`iris`: First-Class Multi-GPU Programming Experience in Triton

Muhammad Awad, Muhammad Osama & Brandon Potter
muhaawad@amd.com, muhosama@amd.com & bpotter@amd.com
Research and Advanced Development (RAD), AMD

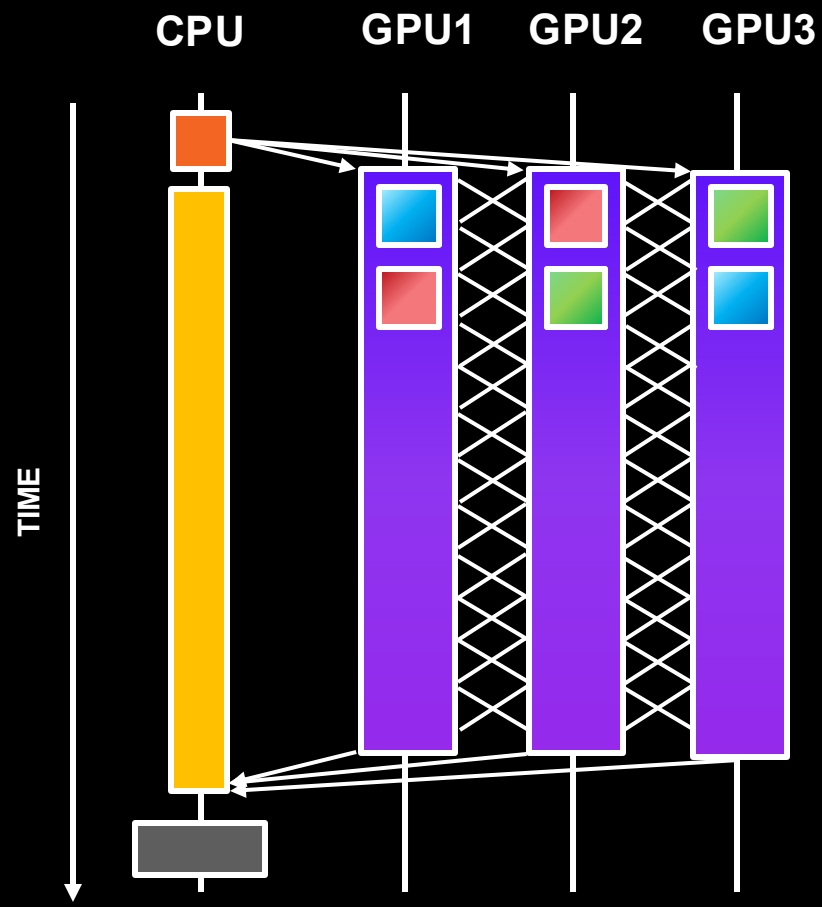
AMD
together we advance_

```
import iris
Github.com/ROCm/iris
```

Open-sourced triton-based framework for Remote Memory Access (RMA†) operations written in only 370 lines of code. Iris provides SHMEM-like APIs within Triton for Multi-GPU programming

[ ] Make Multi-GPU programming a first-class citizen in Triton while retaining Triton's programmability and performance

[ ] Familiar PyTorch- and Triton-Like APIs for host- and device-side abstractions



†*RDMA support is WIP*

Contemporary Approach

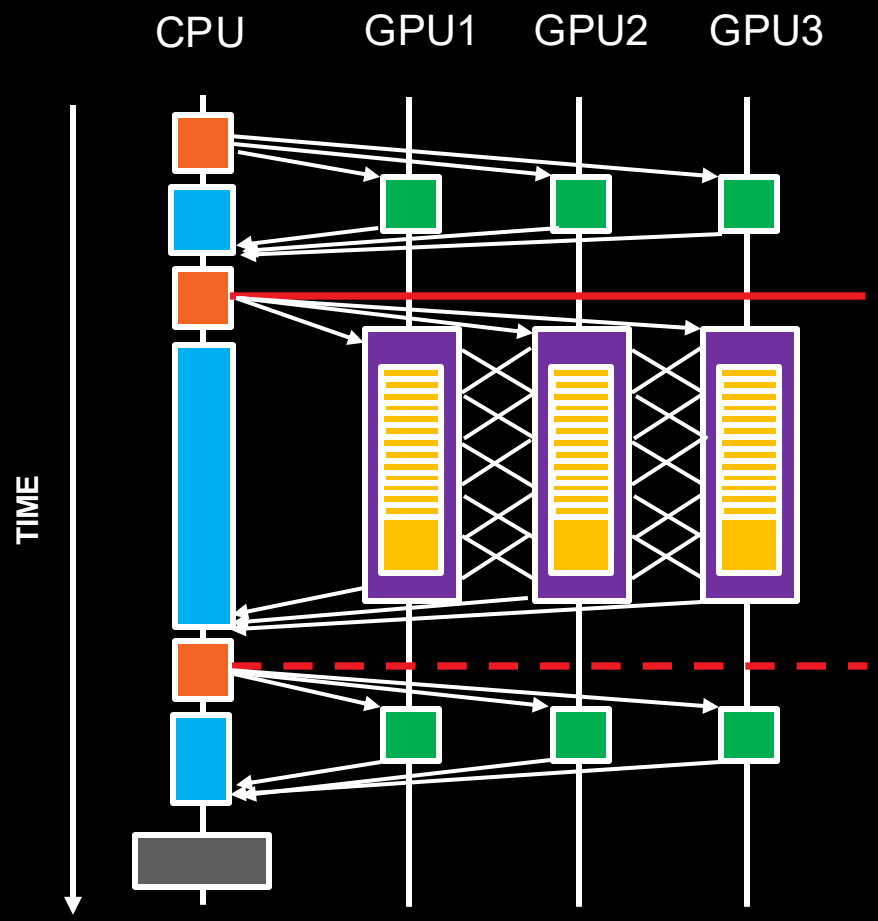
MAIN CPU THREAD

```
torch.cuda.set_device(rank)
A_KERNEL<<<…, stream>>>(buf);
RCCL<<<…>>>(buf);
B_KERNEL<<<…, stream>>>(buf);
```

RCCL COMM KERNEL

```
// Per workgroup
for (step in collective_schedule) {
    post_peer();
    wait_peer();
}
```

- CPU Execution
- GPU Kernel Execution
- RCCL Kernel Execution
- Stream Synchronize
- GPU-CPU Channel Operations



ATTRIBUTES

- **CPU initiates communication (control path)**
  - Host-device synchronization achieved with channels
  - Bulk-synchronous communication phases
- Remote device communication may not begin early due to stream synchronization between GPU kernels

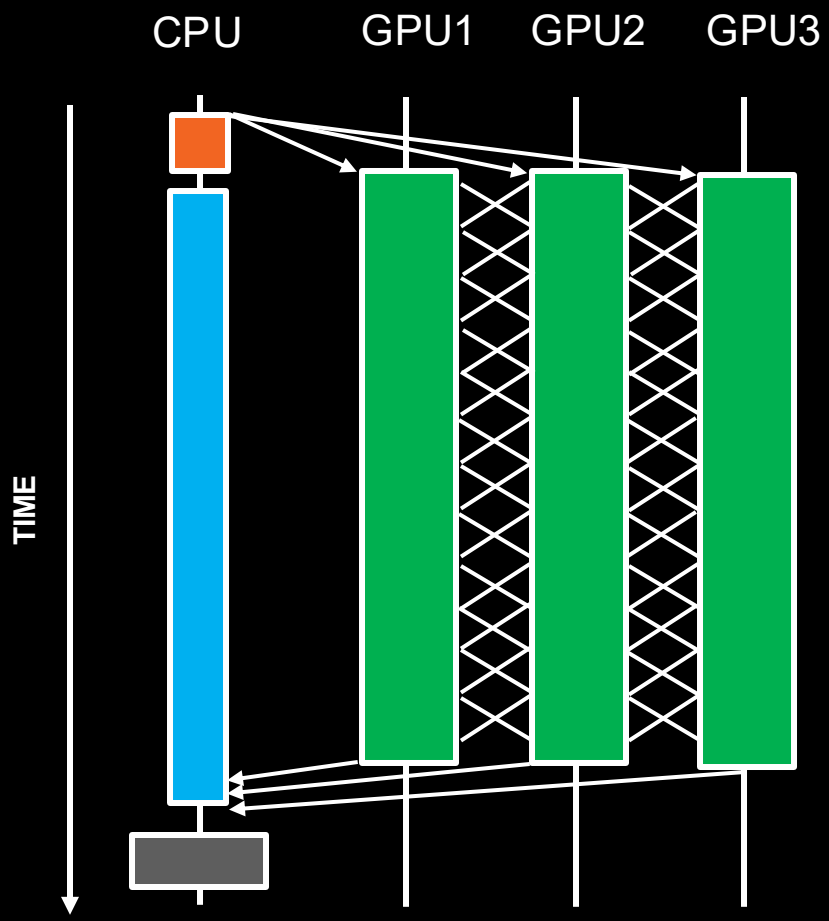
World of iris

MAIN CPU THREAD

```
torch.cuda.set_device(rank)
FUSED_KERNEL<<<…,stream>>>(buf);
```

IRIS COMM + COMP KERNEL

```
// Per workgroup
compute();
if (need_to_put_to_peer) {
  store(dst, data, peer);
}
compute();
if (need_to_get_from_peer) {
  data = load(src, peer);
}
```



CPU Execution
GPU Kernel Execution
Stream Synchronize

ATTRIBUTES

- **GPU initiates communication control path**
  - Dynamic communication phases
- **No intermediate buffers**
  - Data written directly to application
- Remote device communication may begin **as soon as data** is produced
  - Suitable for tiled algorithms
  - Suitable for partitioned communication / computation schemes with specialized roles

`iris`: Multi-GPU Library for Triton

Designed by Experts, Built for Scale

- Written from scratch by GPU and distributed computing experts
- **Minimal dependencies:** only Triton, PyTorch, HIP runtime
- No external frameworks or heavyweight runtimes beyond core stack

Clean Abstractions

- Full **Symmetric Heap** implementation in Python
- **Pythonic PyTorch-like host APIs** for tensor allocation and construction
- **Pythonic Triton-style device APIs** for load, store, and atomic ops

Communication + Computation

- Examples for device-side collective ops: **broadcast, scatter, reduce**, etc.
- Lock variants for **communication and computation overlap**
- **Fine-grained GEMM + communication overlap** via **workgroup specialization**

Scalable by Design

- Full **scale-up** (multi-GPU node) support
- **Scale-out** (multi-node) in progress

`iris`: Multi-GPU Library for Triton

Designed by Experts, Built for Scale

- Written from scratch by GPU and distributed computing experts
- **Minimal dependencies:** only Triton, PyTorch, HIP runtime
- No external frameworks or heavyweight runtimes beyond core stack

Clean Abstractions

- Full **Symmetric Heap** implementation in Python
- **Pythonic PyTorch-like host APIs** for tensor allocation and construction
- **Pythonic Triton-style device APIs** for load, store, and atomic ops

Communication + Computation

- Examples for device-side collective ops: **broadcast, scatter, reduce**, etc.
- Lock variants for **communication and computation overlap**
- **Fine-grained GEMM + communication overlap** via **workgroup specialization**

Scalable by Design

- Full **scale-up** (multi-GPU node) support
- **Scale-out** (multi-node) in progress

`iris`: Multi-GPU Library for Triton

Designed by Experts, Built for Scale

- Written from scratch by GPU and distributed computing experts
- **Minimal dependencies:** only Triton, PyTorch, HIP runtime
- No external frameworks or heavyweight runtimes beyond core stack

Clean Abstractions

- Full **Symmetric Heap** implementation in Python
- **Pythonic PyTorch-like host APIs** for tensor allocation and construction
- **Pythonic Triton-style device APIs** for load, store, and atomic ops

Communication + Computation

- Examples for device-side collective ops: **broadcast, scatter, reduce**, etc.
- Lock variants for **communication and computation overlap**
- **Fine-grained GEMM + communication overlap** via **workgroup specialization**

Scalable by Design

- Full **scale-up** (multi-GPU node) support
- **Scale-out** (multi-node) in progress

`iris`: Multi-GPU Library for Triton

Designed by Experts, Built for Scale

- Written from scratch by GPU and distributed computing experts
- **Minimal dependencies:** only Triton, PyTorch, HIP runtime
- No external frameworks or heavyweight runtimes beyond core stack

Clean Abstractions

- Full **Symmetric Heap** implementation in Python
- **Pythonic PyTorch-like host APIs** for tensor allocation and construction
- **Pythonic Triton-style device APIs** for load, store, and atomic ops

Communication + Computation

- Examples for device-side collective ops: **broadcast, scatter, reduce**, etc.
- Lock variants for **communication and computation overlap**
- **Fine-grained GEMM + communication overlap** via **workgroup specialization**

Scalable by Design

- Full **scale-up** (multi-GPU node) support
- **Scale-out** (multi-node) in progress

`iris`: Multi-GPU Library for Triton

Designed by Experts, Built for Scale

- Written from scratch by GPU and distributed computing experts
- **Minimal dependencies:** only Triton, PyTorch, HIP runtime
- No external frameworks or heavyweight runtimes beyond core stack

Clean Abstractions

- Full **Symmetric Heap** implementation in Python
- **Pythonic PyTorch-like host APIs** for tensor allocation and construction
- **Pythonic Triton-style device APIs** for load, store, and atomic ops

Communication + Computation

- Examples for device-side collective ops: **broadcast, scatter, reduce**, etc.
- Lock variants for **communication and computation overlap**
- **Fine-grained GEMM + communication overlap** via **workgroup specialization**

Scalable by Design

- Full **scale-up** (multi-GPU node) support
- **Scale-out** (multi-node) in progress

`iris`: Multi-GPU Library for Triton

Designed by Experts, Built for Scale

- Written from scratch by GPU and distributed computing experts
- **Minimal dependencies:** only Triton, PyTorch, HIP runtime
- No external frameworks or heavyweight runtimes beyond core stack

Clean Abstractions

- Full **Symmetric Heap** implementation in Python
- **Pythonic PyTorch-like host APIs** for tensor allocation and construction
- **Pythonic Triton-style device APIs** for load, store, and atomic ops

Communication + Computation

- Examples for device-side collective ops: **broadcast, scatter, reduce**, etc.
- Lock variants for **communication and computation overlap**
- **Fine-grained GEMM + communication overlap** via **workgroup specialization**

Scalable by Design

- Full **scale-up** (multi-GPU node) support
- **Scale-out** (multi-node) in progress

The `iris` API surface — clean, simple, familiar

PyTorch-Like Tensor Creation

```
def full(self, size, fill_value, *, out=None, dtype=None,
         layout=torch.strided, device=None, requires_grad=False):
    """
    Creates a tensor of size size filled with fill_value. The tensor's dtype is
    inferred from fill_value.
    The tensor is allocated on the Iris symmetric heap.


    Args:
    size (int...): a list, tuple, or torch.Size of integers defining the shape of
    the output tensor.
    fill_value (Scalar): the value to fill the output tensor with.


    Keyword Arguments:
    out (Tensor, optional): the output tensor.
    dtype (torch.dtype, optional): the desired data type of returned tensor.
    Default: if None, uses a global default (see torch.set_default_dtype()).
    layout (torch.layout, optional): the desired layout of returned Tensor.
    Default: torch.strided. Note: Iris tensors always use `torch.strided`
    regardless of this parameter.
    device (torch.device, optional): the desired device of returned tensor.
    Default: if None, uses the current device for the default tensor type.
    requires_grad (bool, optional): If autograd should record operations on the
    returned tensor. Default: False.
    """
```

PyTorch-Like Tensor Creation

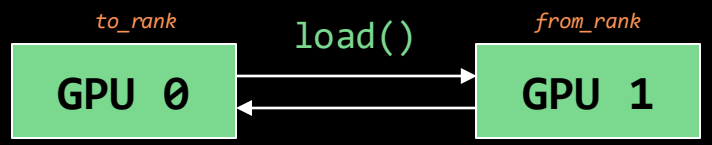
```
def full(self, size, fill_value, *, out=None, dtype=None,
         layout=torch.strided, device=None, requires_grad=False):
    """
    Creates a tensor of size size filled with fill_value. The tensor's dtype is
    inferred from fill_value.
    The tensor is allocated on the Iris symmetric heap.


    Args:
    size (int...): a list, tuple, or torch.Size of integers defining the shape of
    the output tensor.
    fill_value (Scalar): the value to fill the output tensor with.


    Keyword Arguments:
    out (Tensor, optional): the output tensor.
    dtype (torch.dtype, optional): the desired data type of returned tensor.
    Default: if None, uses a global default (see torch.set_default_dtype()).
    layout (torch.layout, optional): the desired layout of returned Tensor.
    Default: torch.strided. Note: Iris tensors always use `torch.strided`
    regardless of this parameter.
    device (torch.device, optional): the desired device of returned tensor.
    Default: if None, uses the current device for the default tensor type.
    requires_grad (bool, optional): If autograd should record operations on the
    returned tensor. Default: False.
    """
```

And more!

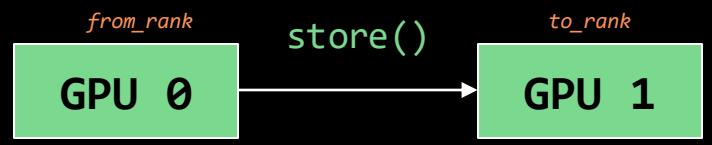
```
iris.ones(…)
iris.zeros(…)
iris.zero_like(…)

iris.arange(…)
iris.linespace(…)

iris.empty(…)
iris.randint(…)
iris.rand(…)
iris.randn(…)
```

Simple load & store APIs

```python
@triton.jit
def load(pointer, to_rank, from_rank, heap_bases, mask=None):
    """
    Loads a value from the specified memory location and rank.

       Args:
        pointer (int): The source pointer.
        to_rank (int): The current rank.
        from_rank (int): The rank to load data from.
        heap_bases (int): The heap bases.
        mask (Optional[tl.tensor], optional): A boolean tensor
           used to guard memory accesses.

       Returns:
        Any: The loaded value.
    """
```

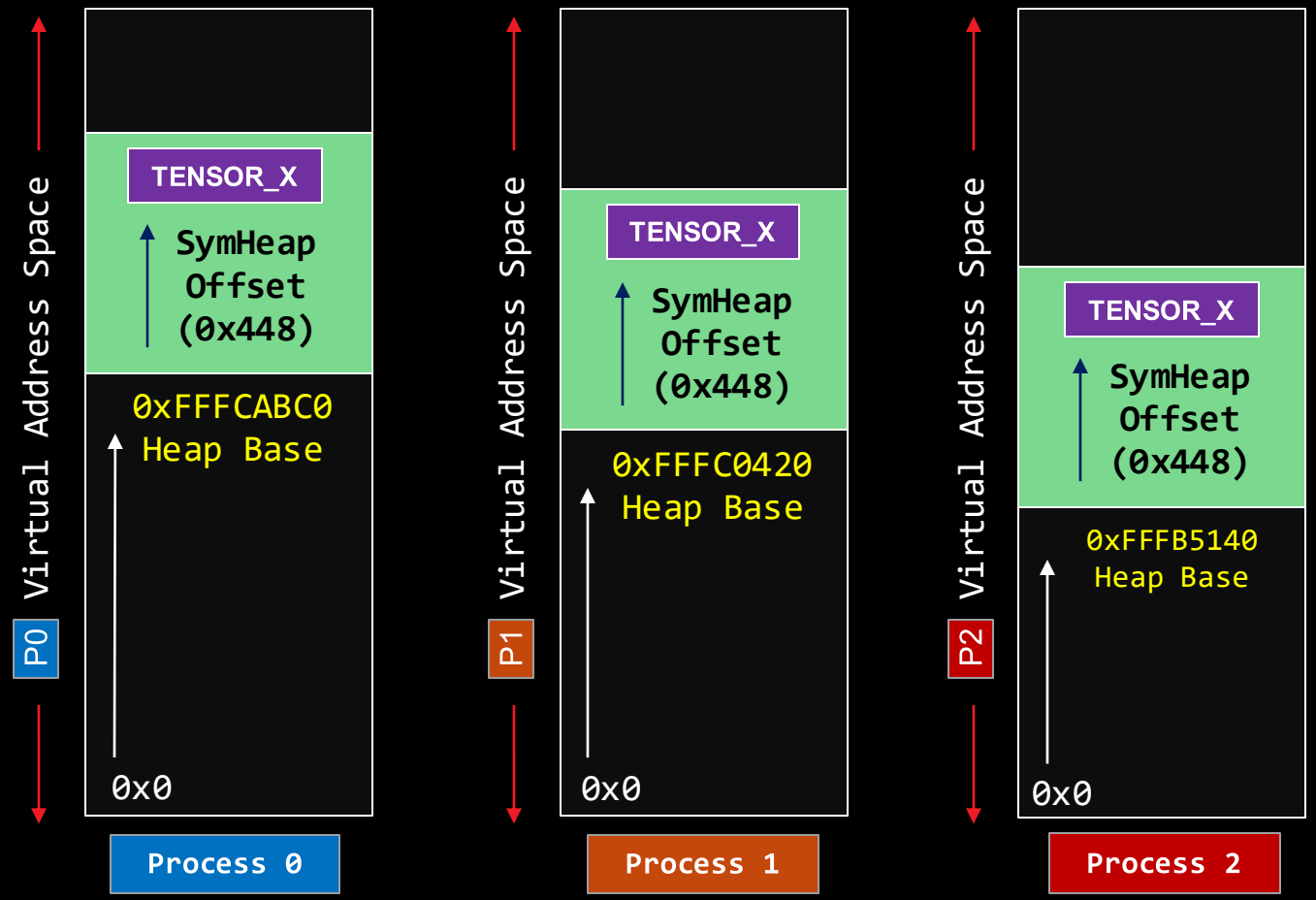



```python
@triton.jit
def store(pointer, value, from_rank, to_rank, heap_bases,
   mask=None):
    """
    Writes data to the specified memory location and rank.

       Args:
        pointer (int): The destination pointer.
        value (Any): The value to be written.
        from_rank (int): The current rank.
        to_rank (int): The rank to store into.
        heap_bases (int): The heap bases.
        mask (Optional[tl.tensor], optional): A boolean tensor
           used to guard memory accesses. Defaults to None.

       Returns:
        None
    """
```



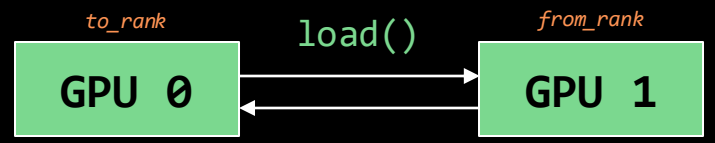
iris Symmetric Heap

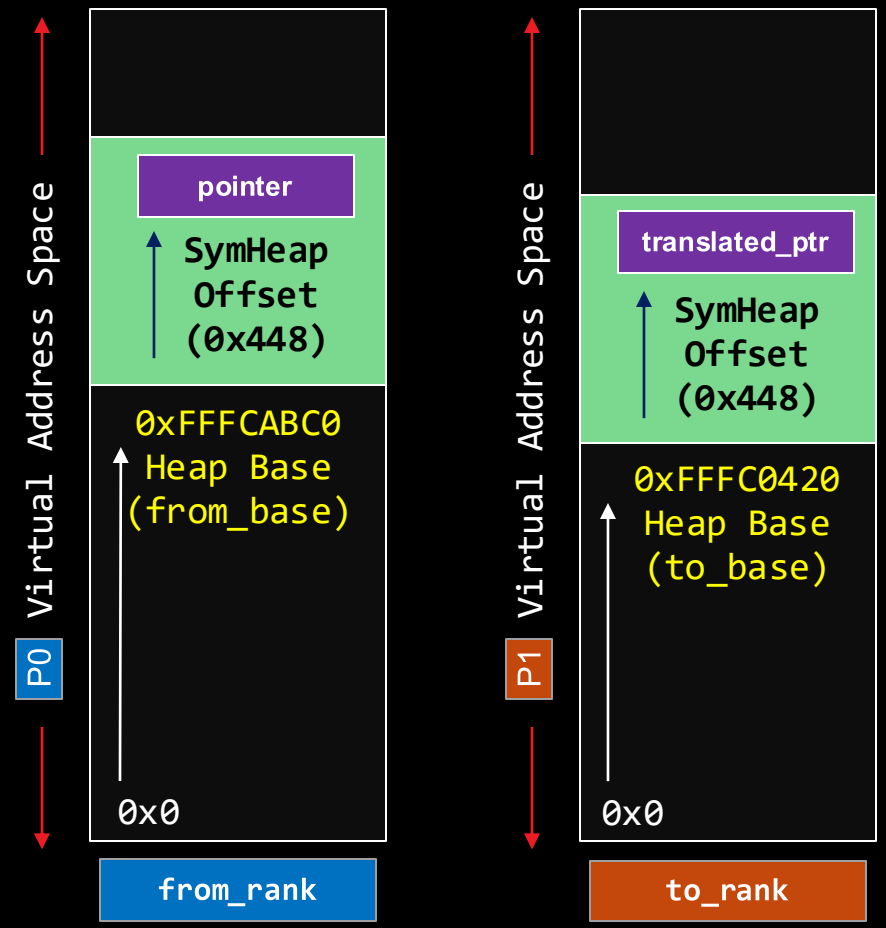
- Symmetric Heap is a Partitioned Global Address Space (PGAS) abstraction
- Key idea is that you can know the remote address of any symmetric variable with two offsets:
  1. Offset of target Process' heap base in its virtual address space
  2. Offset of the variable within the symmetric heap
- Allocation routine for symmetric variables must be collective or offset must be known
- Must `all_gather` the base heap addresses across all processes



Implementation of load API

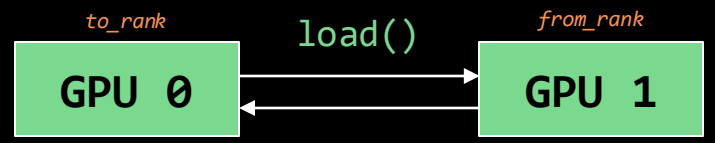
```python
@triton.jit
def load(pointer, to_rank, from_rank, heap_bases,
         mask=None):
    translated_ptr = __translate(pointer, to_rank,
                                 from_rank, heap_bases)
    result = tl.load(translated_ptr, mask=mask)
    return result

@triton.jit
def __translate(ptr, from_rank, to_rank, heap_bases):
    from_base = tl.load(heap_bases + from_rank)
    to_base = tl.load(heap_bases + to_rank)
    offset = tl.cast(ptr, tl.uint64) - from_base
    to_base_byte = tl.cast(to_base,
                   tl.pointer_type(tl.int8))
    translated_ptr_byte = to_base_byte + offset
    translated_ptr = tl.cast(translated_ptr_byte,
                     ptr.dtype)
    return translated_ptr
```

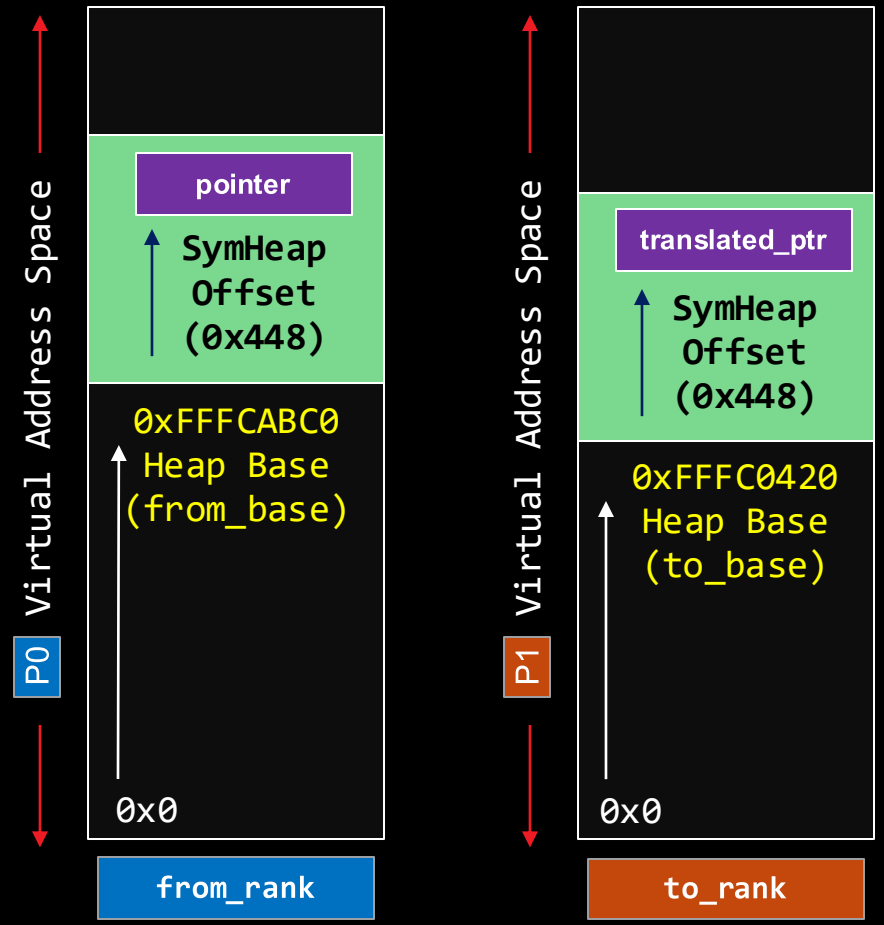




Implementation of load API

```python
@triton.jit
def load(pointer, to_rank, from_rank, heap_bases,
         mask=None):
    translated_ptr = __translate(pointer, to_rank,
                                 from_rank, heap_bases)
    result = tl.load(translated_ptr, mask=mask)
    return result

@triton.jit
def __translate(ptr, from_rank, to_rank, heap_bases):
    from_base = tl.load(heap_bases + from_rank)
    to_base = tl.load(heap_bases + to_rank)
    offset = tl.cast(ptr, tl.uint64) - from_base
    to_base_byte = tl.cast(to_base,
                    tl.pointer_type(tl.int8))
    translated_ptr_byte = to_base_byte + offset
    translated_ptr = tl.cast(translated_ptr_byte,
                      ptr.dtype)
    return translated_ptr
```

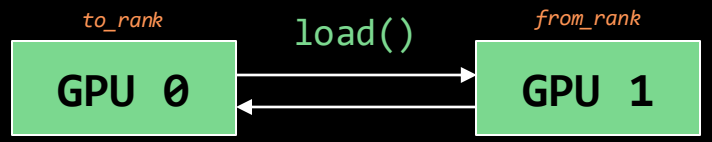


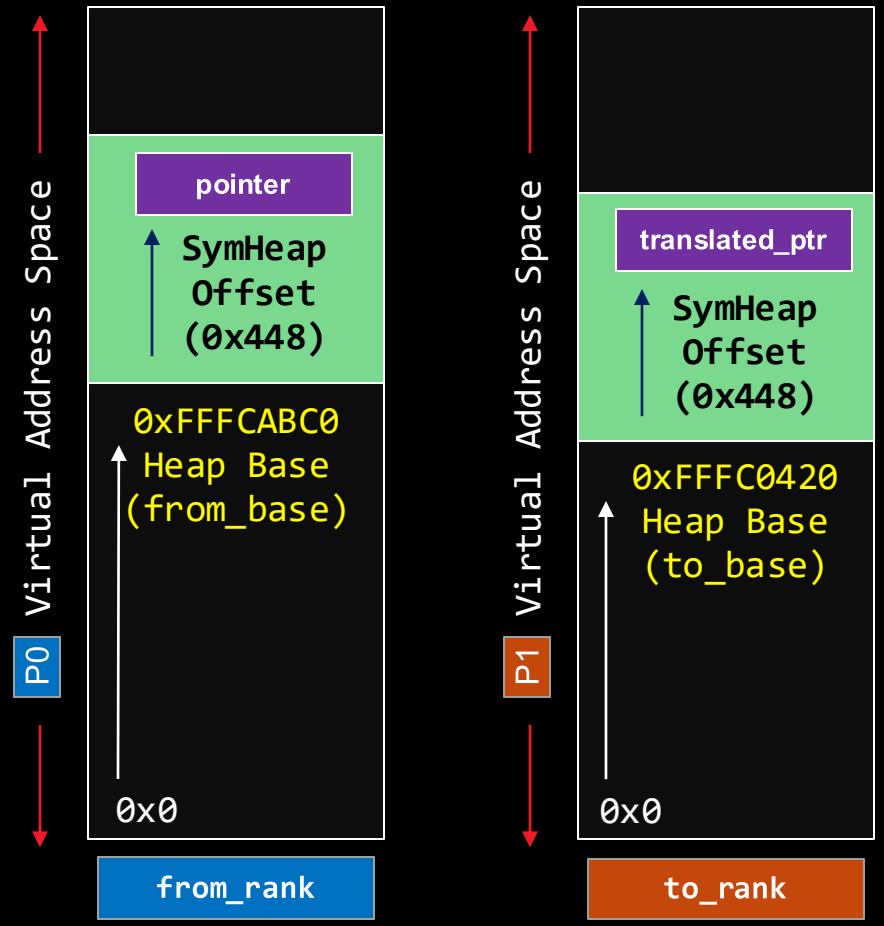


Implementation of load API

```python
@triton.jit
def load(pointer, to_rank, from_rank, heap_bases,
         mask=None):
    translated_ptr = __translate(pointer, to_rank,
                                 from_rank, heap_bases)
    result = tl.load(translated_ptr, mask=mask)
    return result

@triton.jit
def __translate(ptr, from_rank, to_rank, heap_bases):
    from_base = tl.load(heap_bases + from_rank)
    to_base = tl.load(heap_bases + to_rank)
    offset = tl.cast(ptr, tl.uint64) - from_base
    to_base_byte = tl.cast(to_base,
                           tl.pointer_type(tl.int8))
    translated_ptr_byte = to_base_byte + offset
    translated_ptr = tl.cast(translated_ptr_byte,
                             ptr.dtype)
    return translated_ptr
```

Load heap bases

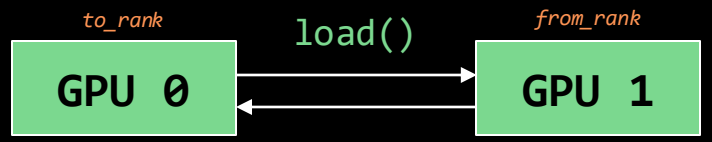


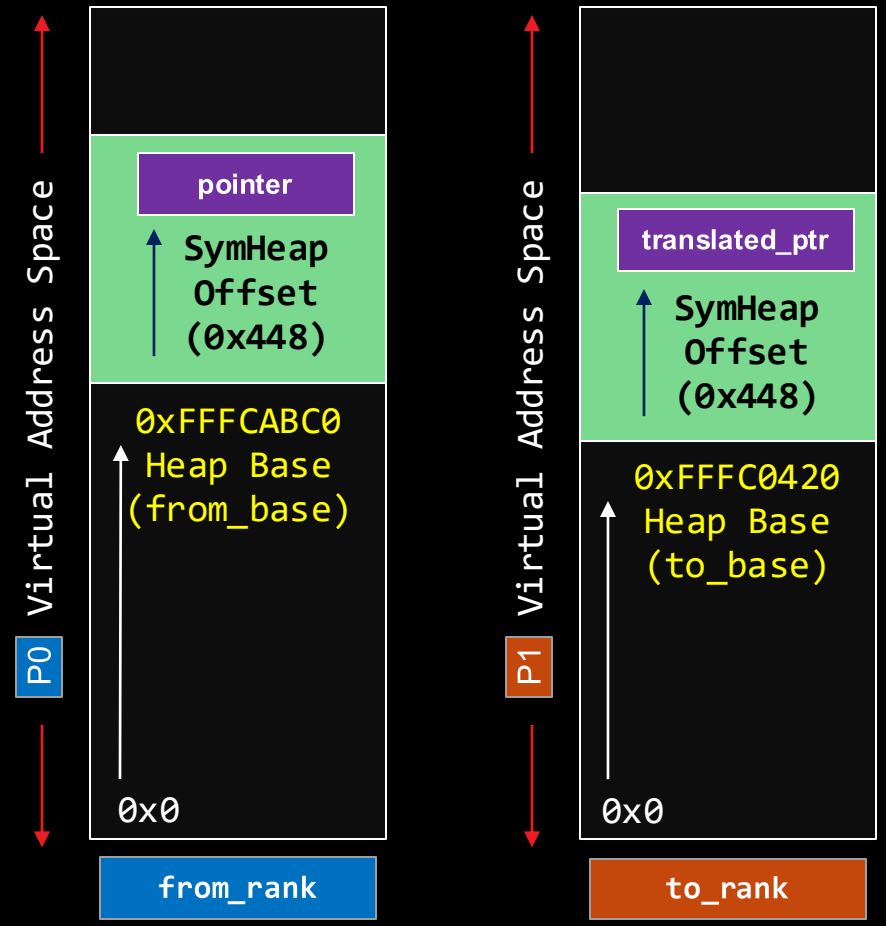


Implementation of load API

```python
@triton.jit
def load(pointer, to_rank, from_rank, heap_bases,
         mask=None):
    translated_ptr = __translate(pointer, to_rank,
                                 from_rank, heap_bases)
    result = tl.load(translated_ptr, mask=mask)
    return result

@triton.jit
def __translate(ptr, from_rank, to_rank, heap_bases):
    from_base = tl.load(heap_bases + from_rank)
    to_base = tl.load(heap_bases + to_rank)
    offset = tl.cast(ptr, tl.uint64) - from_base
    to_base_byte = tl.cast(to_base,
                           tl.pointer_type(tl.int8))
    translated_ptr_byte = to_base_byte + offset
    translated_ptr = tl.cast(translated_ptr_byte,
                             ptr.dtype)
    return translated_ptr
```

Compute offset on current rank

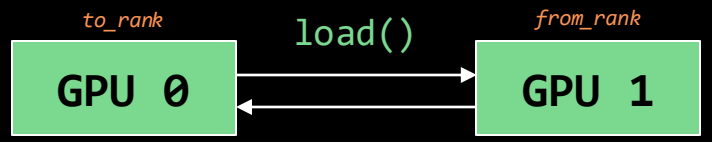


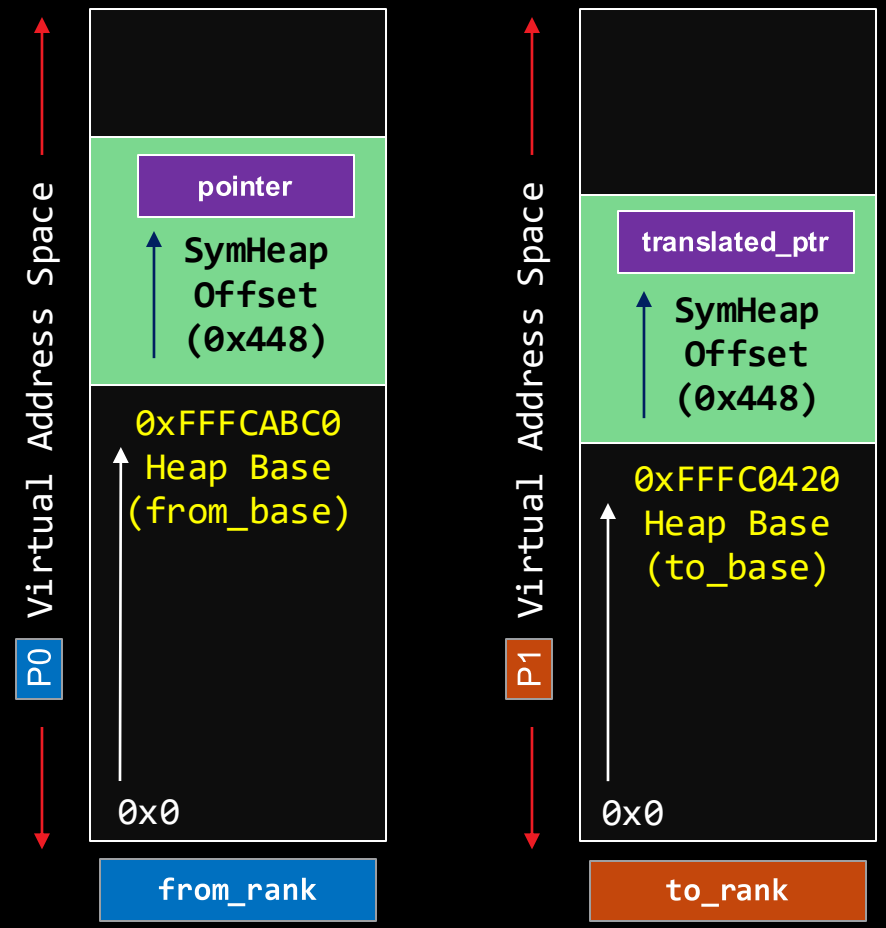


Implementation of load API

```python
@triton.jit
def load(pointer, to_rank, from_rank, heap_bases,
         mask=None):
    translated_ptr = __translate(pointer, to_rank,
                                 from_rank, heap_bases)
    result = tl.load(translated_ptr, mask=mask)
    return result

@triton.jit
def __translate(ptr, from_rank, to_rank, heap_bases):
    from_base = tl.load(heap_bases + from_rank)
    to_base = tl.load(heap_bases + to_rank)
    offset = tl.cast(ptr, tl.uint64) - from_base
    to_base_byte = tl.cast(to_base,
                           tl.pointer_type(tl.int8))
    translated_ptr_byte = to_base_byte + offset
    translated_ptr = tl.cast(translated_ptr_byte,
                             ptr.dtype)
    return translated_ptr
```

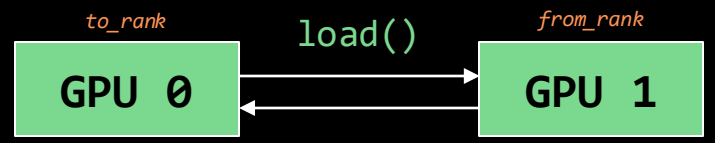
Add offset to destination's heap base

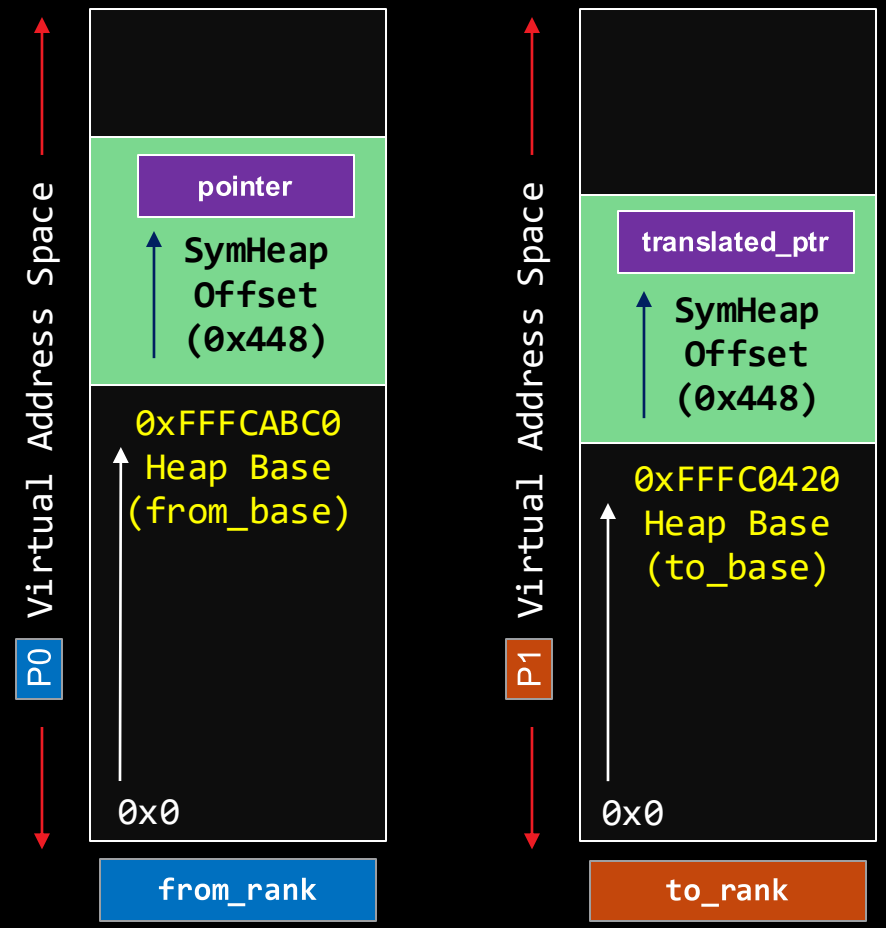




Implementation of load API

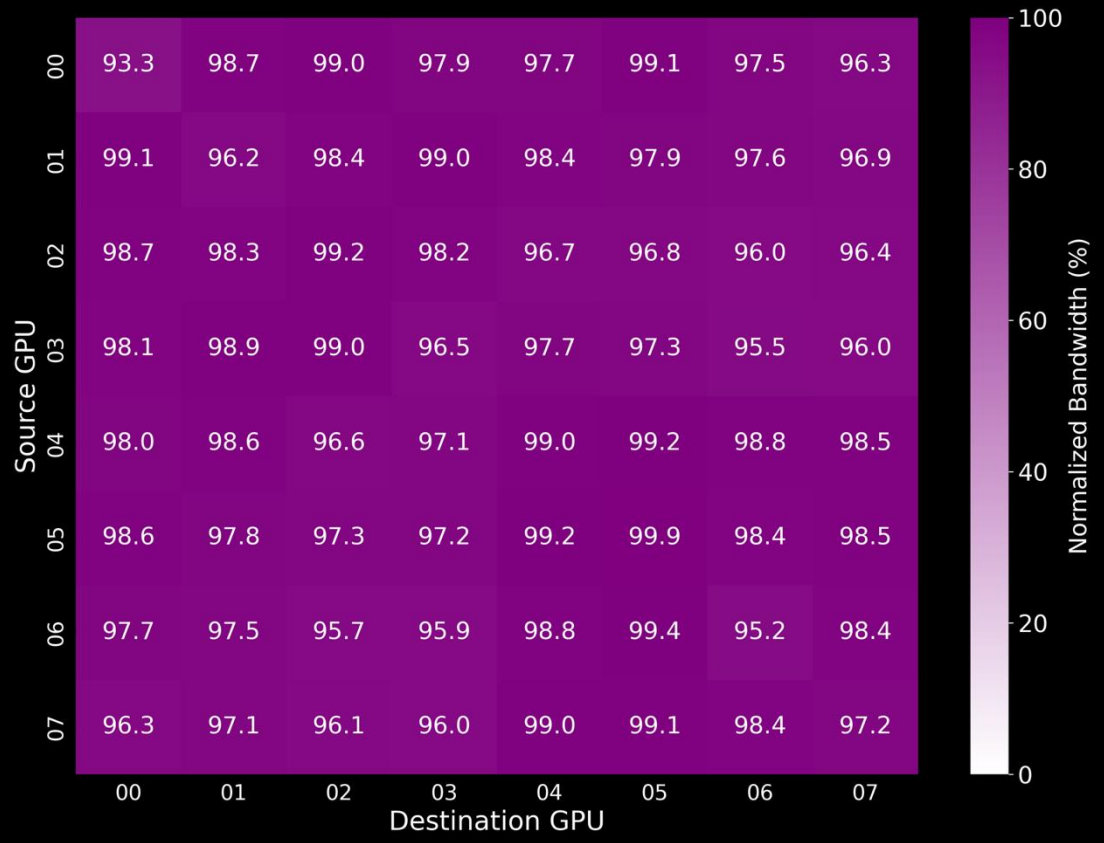
```python
@triton.jit
def load(pointer, to_rank, from_rank, heap_bases,
         mask=None):
    translated_ptr = __translate(pointer, to_rank,
                                 from_rank, heap_bases)
    result = tl.load(translated_ptr, mask=mask)
    return result

@triton.jit
def __translate(ptr, from_rank, to_rank, heap_bases):
    from_base = tl.load(heap_bases + from_rank)
    to_base = tl.load(heap_bases + to_rank)
    offset = tl.cast(ptr, tl.uint64) - from_base
    to_base_byte = tl.cast(to_base,
                   tl.pointer_type(tl.int8))
    translated_ptr_byte = to_base_byte + offset
    translated_ptr = tl.cast(translated_ptr_byte,
                     ptr.dtype)
    return translated_ptr
```



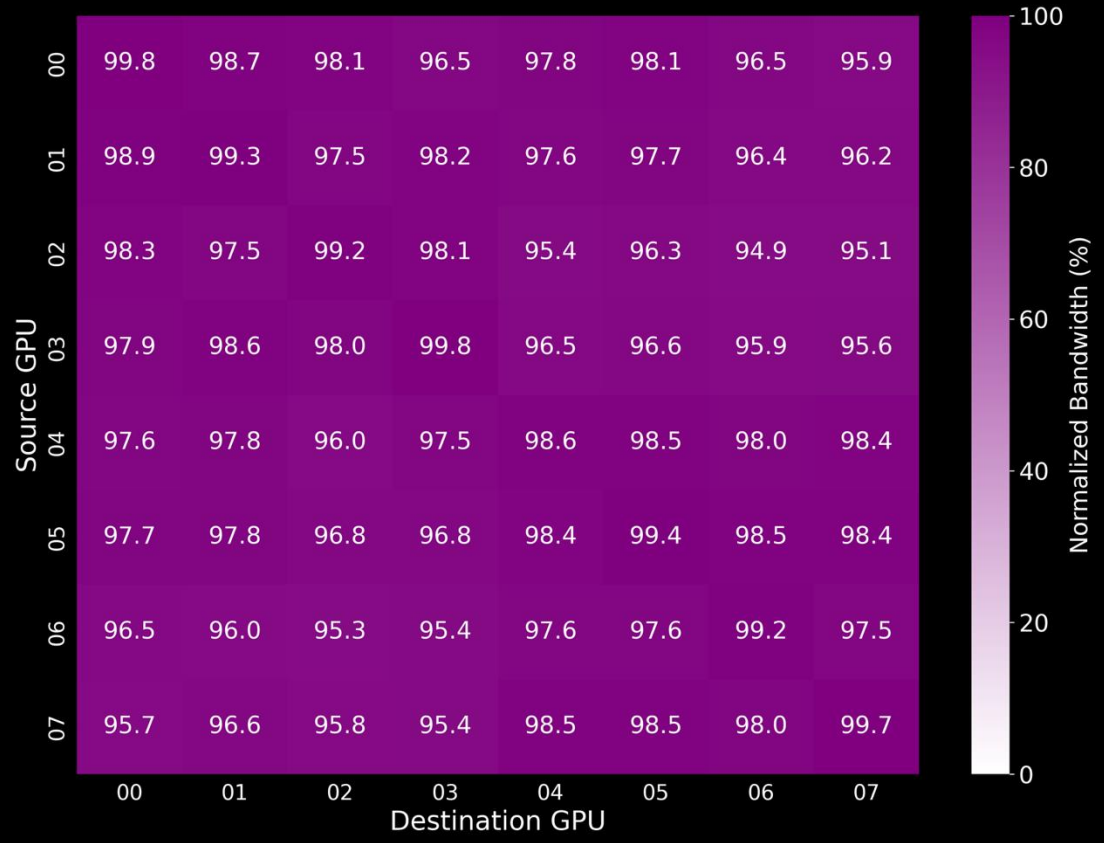


Performance: load & store

load



store



Relative to achievable HBM and xGMI bandwidth.

atomic Operations

```python
@triton.jit
def atomic_add(pointer, val, from_rank, to_rank,
               heap_bases, mask=None, sem=None, scope=None)

@triton.jit
def atomic_cas(pointer, cmp, val, from_rank, to_rank,
               heap_bases, sem=None, scope=None)

@triton.jit
def atomic_xchg(pointer, val, from_rank, to_rank,
               heap_bases, mask=None, sem=None, scope=None)

@triton.jit
def atomic_xor(pointer, val, from_rank, to_rank,
               heap_bases, mask=None, sem=None, scope=None)
```

And more!

```
atomic_and, atomic_or, atomic_min, atomic_max
```

```
"""
    Atomically <op> the memory location at pointer.
    Args:
        pointer (int): The source pointer.
        from_rank (int): The current rank.
        to_rank (int): The remote rank.
        heap_bases (int): The heap bases.
        mask (Optional[tl.tensor], optional): A boolean tensor
        used to guard memory accesses. Defaults to None.
        sem (str, optional): Specifies the memory semantics for
        the operation. Acceptable values are “acquire”,
        “release”, “acq_rel” (stands for “ACQUIRE_RELEASE”),
        and “relaxed”. Defaults to “acq_rel”.

        scope (str, optional): Defines the scope of threads
        that observe the synchronizing effect of the atomic
        operation. Acceptable values are “gpu” (default), “cta”
        (cooperative thread array, thread block), or “sys”
        (stands for “SYSTEM”). Defaults to “gpu”.
"""
```

Hello, iris

iris: Simple Producer-Consumer Example (import iris)

```python
def main():
    dist.init_process_group(backend="nccl")
    iris_instance = iris.iris(heap_size=1 << 30)
    input = iris.randint(...)
    flags = iris.zeros(...)
    if iris_instance.local_rank() == 0:
        with torch.cuda.stream(producer_stream):
            producer_kernel[grid](input, flags,...)
    else:
        with torch.cuda.stream(consumer_stream):
            consumer_kernel[grid](input, flags,...)
@triton.jit
def producer_kernel(input_ptr, flag_ptr, ...):
    # Logic to produce data:
    iris.store(input_ptr + offsets, input_data
        local_rank, to_rank, heap_bases, mask=mask)
    # Signal consumer kernel:
    compare = 0
    value = 1
    iris.atomic_cas(flag_ptr + pid, compare, value,
       local_rank, to_rank, heap_bases, sem="release", scope="sys")
```

Initialize an Iris Instance

Allocate and initialize tensor across ranks

Launch producer-consumer kernels on different ranks (GPUs)

```python
@triton.jit
def consumer_kernel(input_ptr, output_ptr, ...):
# Wait for producer's signal:
result = 0
while result == 0:
    compare = 1
    value = 0
    result = iris.atomic_cas(flag_ptr + pid, compare, value,
                local_rank, local_rank, heap_bases,
                sem="acquire", scope="sys")

# Fetch data:
data = iris.load(input_ptr + offsets, local_rank,
                local_rank, heap_bases, mask)

# Consume data:
# ..
```

iris: Simple Producer-Consumer Example (import iris)

```python
def main():
    dist.init_process_group(backend="nccl")
    iris_instance = iris.iris(heap_size=1 << 30)
    input = iris.randint(...)
    flags = iris.zeros(...)
    if iris_instance.local_rank() == 0:
        with torch.cuda.stream(producer_stream):
            producer_kernel[grid](input, flags,...)
    else:
        with torch.cuda.stream(consumer_stream):
            consumer_kernel[grid](input, flags,...)
@triton.jit
def producer_kernel(input_ptr, flag_ptr, ...):
    # Logic to produce data:
    iris.store(input_ptr + offsets, input_data
        local_rank, to_rank, heap_bases, mask=mask)
    # Signal consumer kernel:
    compare = 0
    value = 1
    iris.atomic_cas(flag_ptr + pid, compare, value,
       local_rank, to_rank, heap_bases, sem="release", scope="sys")
```

Producer kernel places the data in remote ranks using `iris.store()`

```python
@triton.jit
def consumer_kernel(input_ptr, output_ptr, ...):
    # Wait for producer's signal:
    result = 0
    while result == 0:
        compare = 1
        value = 0
        result = iris.atomic_cas(flag_ptr + pid, compare, value,
                    local_rank, local_rank, heap_bases,
                    sem="acquire", scope="sys")

    # Fetch data:
    data = iris.load(input_ptr + offsets, local_rank,
                    local_rank, heap_bases, mask)

    # Consume data:
    # ..
```

Using an atomic compare-and-swap, signal that the data is ready to be consumed

iris: Simple Producer-Consumer Example (import iris)

```python
def main():
    dist.init_process_group(backend="nccl")
    iris_instance = iris.iris(heap_size=1 << 30)
    input = iris.randint(...)
    flags = iris.zeros(...)
    if iris_instance.local_rank() == 0:
        with torch.cuda.stream(producer_stream):
            producer_kernel[grid](input, flags,...)
    else:
        with torch.cuda.stream(consumer_stream):
            consumer_kernel[grid](input, flags,...)
@triton.jit
def producer_kernel(input_ptr, flag_ptr, ...):
    # Logic to produce data:
    iris.store(input_ptr + offsets, input_data
        local_rank, to_rank, heap_bases, mask=mask)
    # Signal consumer kernel:
    compare = 0
    value = 1
    iris.atomic_cas(flag_ptr + pid, compare, value,
       local_rank, to_rank, heap_bases, sem="release", scope="sys")
```

Using an atomic compare-and-swap, wait for the data ready to be ready

Fetch using `iris.load()` and consume the produced data

```python
@triton.jit
def consumer_kernel(input_ptr, output_ptr, ...):
    # Wait for producer's signal:
    result = 0
    while result == 0:
        compare = 1
        value = 0
        result = iris.atomic_cas(flag_ptr + pid, compare, value,
                    local_rank, local_rank, heap_bases,
                    sem="acquire", scope="sys")

    # Fetch data:
    data = iris.load(input_ptr + offsets, local_rank,
                    local_rank, heap_bases, mask)

    # Consume data:
    # ..
```

iris: Simple Producer-Consumer Example (import iris)

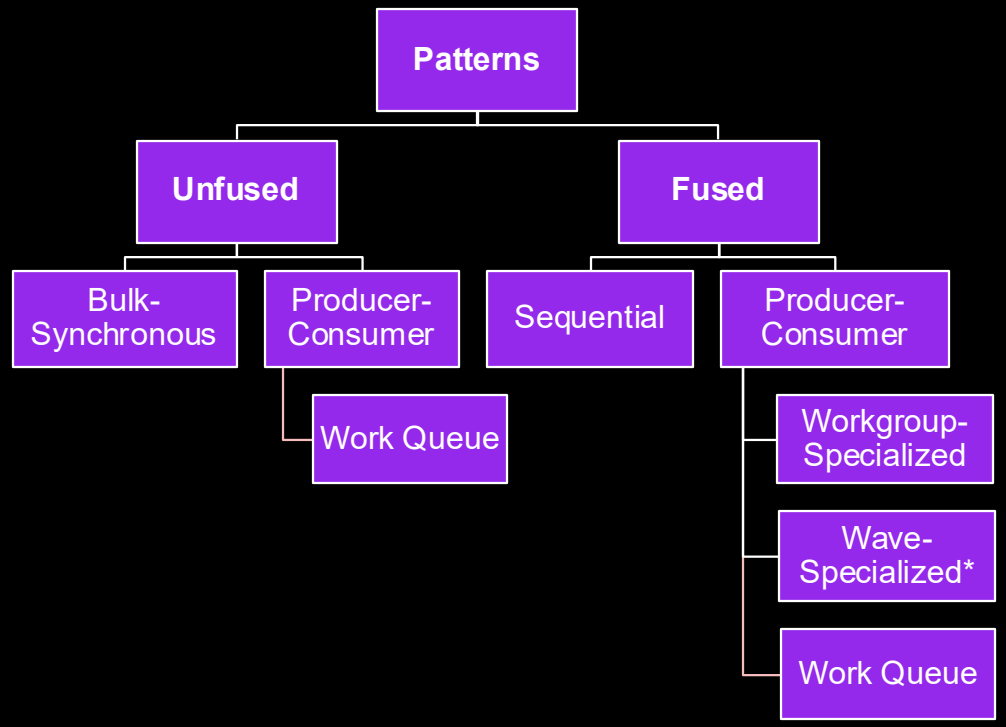
```python
def main():
    dist.init_process_group(backend="nccl")
    iris_instance = iris.iris(heap_size=1 << 30)
    input = iris.randint(...)
    flags = iris.zeros(...)
    if iris_instance.local_rank() == 0:
        with torch.cuda.stream(producer_stream):
            producer_kernel[grid](input, flags,...)
    else:
        with torch.cuda.stream(consumer_stream):
            consumer_kernel[grid](input, flags,...)
@triton.jit
def producer_kernel(input_ptr, flag_ptr, ...):
    # Logic to produce data:
    iris.store(input_ptr + offsets, input_data
        local_rank, to_rank, heap_bases, mask=mask)
    # Signal consumer kernel:
    compare = 0
    value = 1
    iris.atomic_cas(flag_ptr + pid, compare, value,
       local_rank, to_rank, heap_bases, sem="release", scope="sys")
```

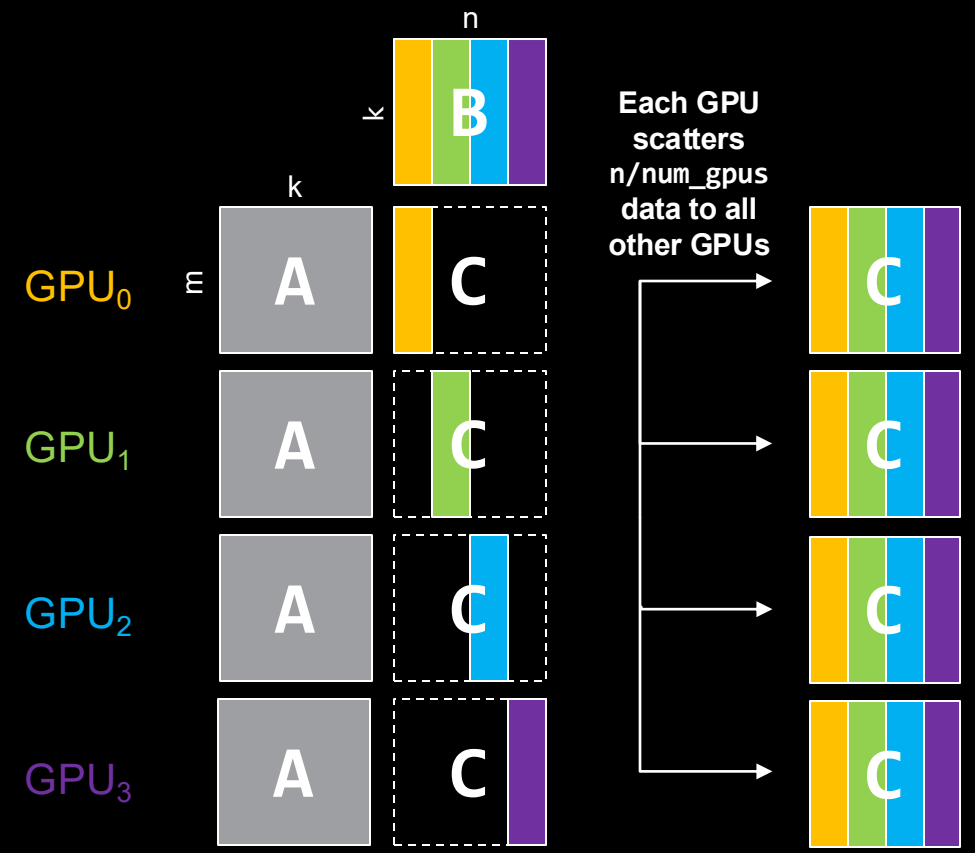
```python
@triton.jit
def consumer_kernel(input_ptr, output_ptr, ...):
    # Wait for producer's signal:
    result = 0
    while result == 0:
        compare = 1
        value = 0
        result = iris.atomic_cas(flag_ptr + pid, compare, value,
                    local_rank, local_rank, heap_bases,
                    sem="acquire", scope="sys")

    # Fetch data:
    data = iris.load(input_ptr + offsets, local_rank,
                    local_rank, heap_bases, mask)

    # Consume data:
    # ..
```

Multi-GPU Producer-Consumer code in a single slide

Many Patterns, One `iris`

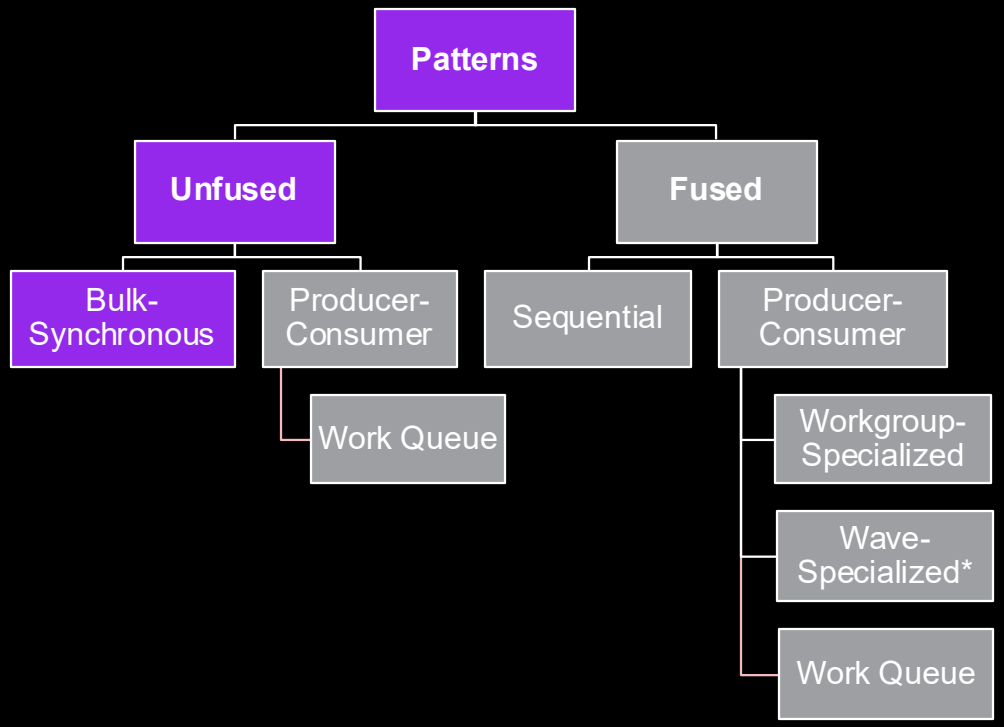
`iris` Taxonomy of Fused & Unfused Patterns (e.g., All-Scatter)

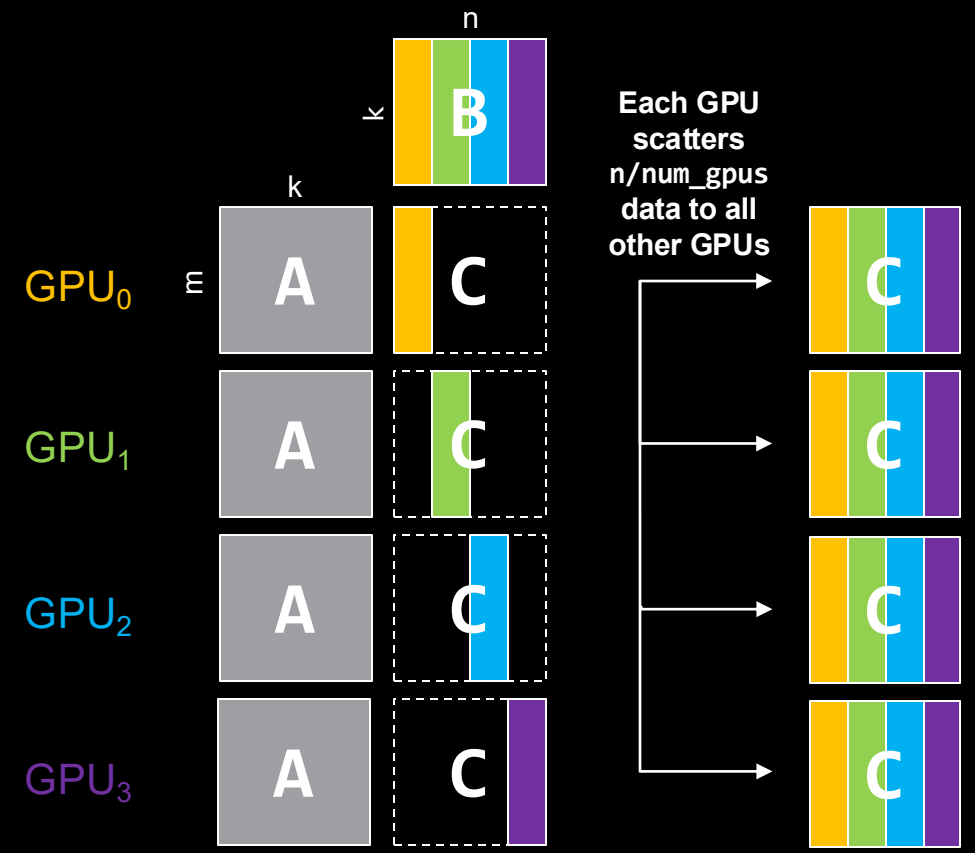




* Not possible without in Triton lang (needs Gluon)

`iris` Taxonomy of Fused & Unfused Patterns (e.g., All-Scatter)

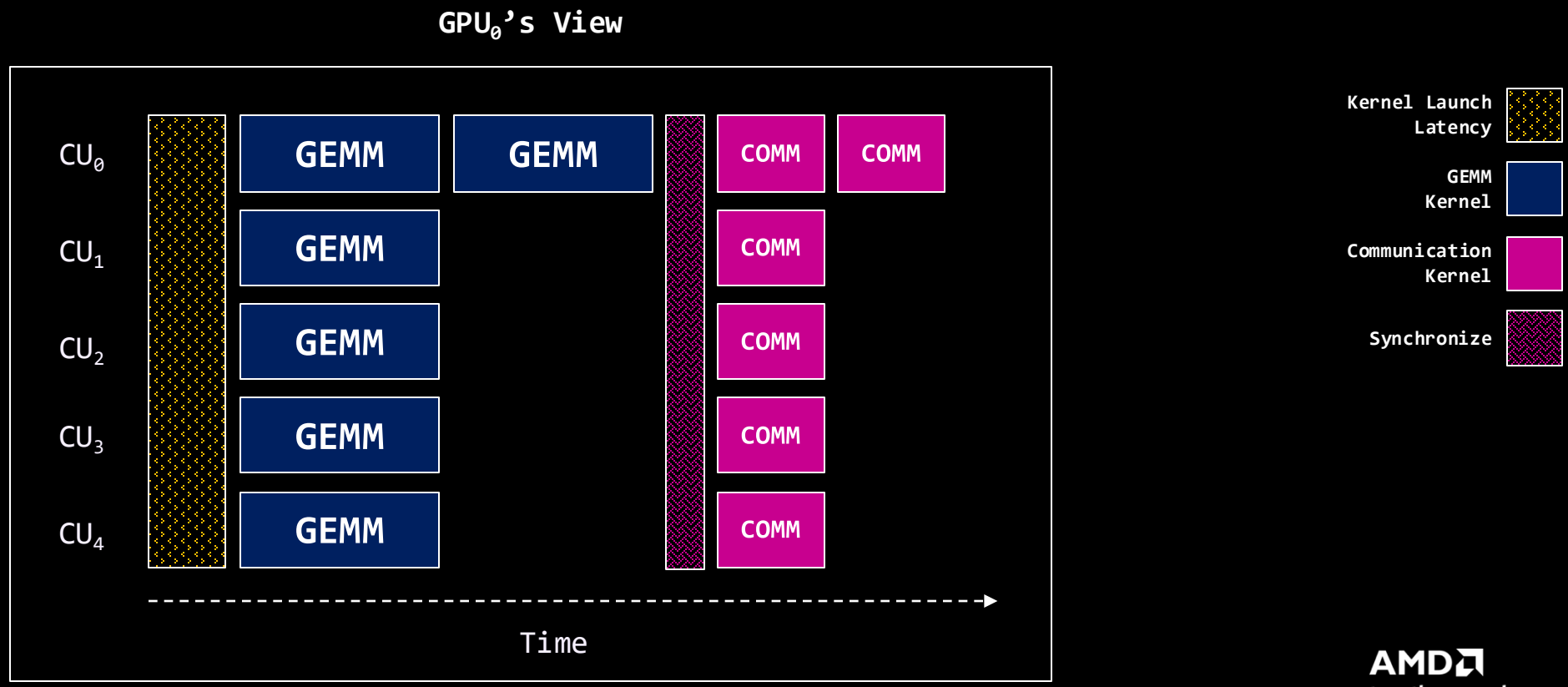




* Not possible without in Triton lang (needs Gluon)

`iris` Unfused, Bulk-Synchronous

- Launch **GEMM**, wait for the kernel to finish, launch **All-Scatter**



iris Unfused, Bulk-Synchronous

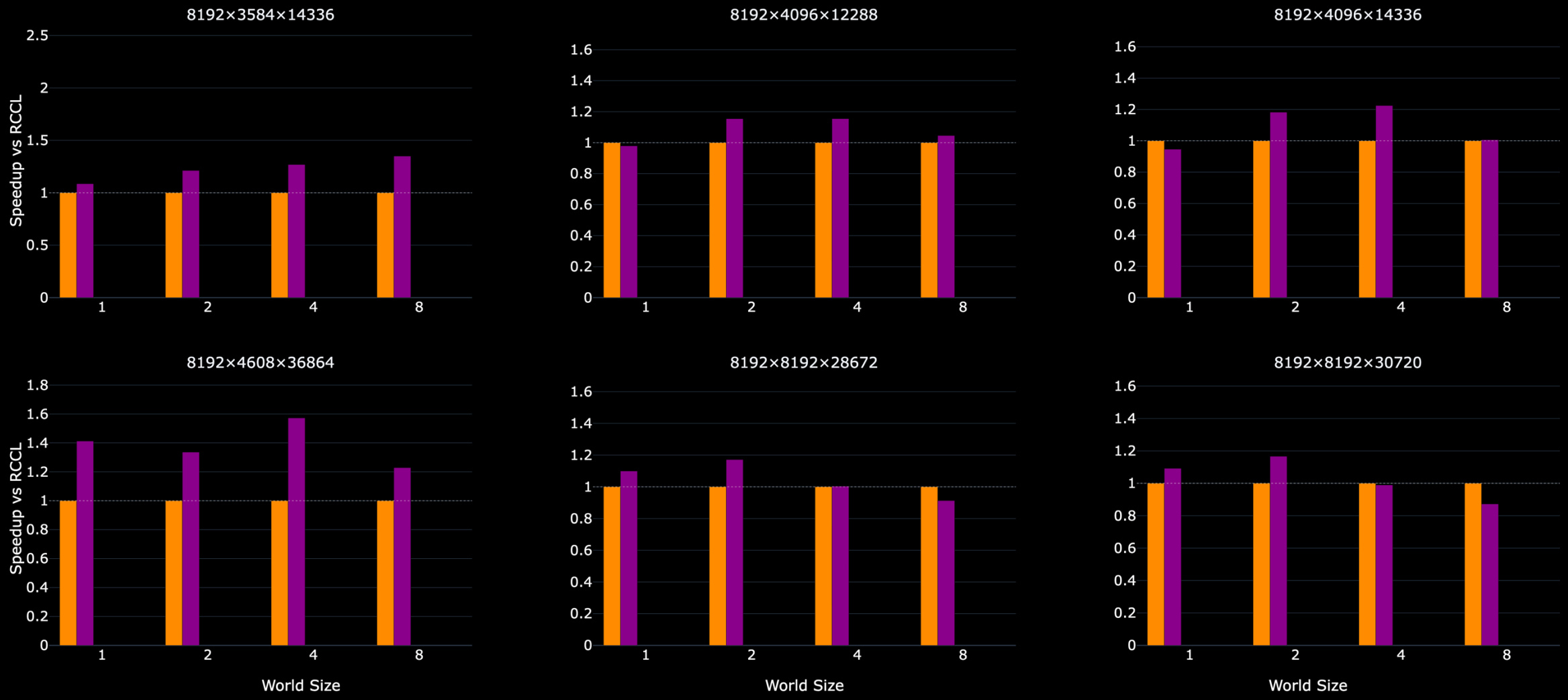
```python
@triton.jit()
def persistent_gemm(
    A,
    B,
    C,
):
    pid = tl.program_id(0)

    for tile_id in range(pid, total_tiles, GEMM_SMS):

        acc = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N),
              dtype=acc_dtype)
        ...

        for k in range(0, loop_k):
            a = tl.load(A_BASE)
            b = tl.load(B_BASE)
            acc += tl.dot(a, b)
            A_BASE += BLOCK_SIZE_K * stride_ak
            B_BASE += BLOCK_SIZE_K * stride_bk

        # Accumulator registers with C results
        c = acc.to(C.type.element_ty)

        tl.store(C + global_offset, c, mask=sub_mask)
```

```python
@triton.jit()
def persistent_all_scatter(
    C,
):

    pid = tl.program_id(0)


    for tile_id in range(pid, total_tiles, COMM_SMS):


        for remote_rank in range(world_size):
            if remote_rank != cur_rank:
                iris.put(C + global_offset, C +
                global_offset, cur_rank, remote_rank,
                heap_bases, mask=sub_mask)
```

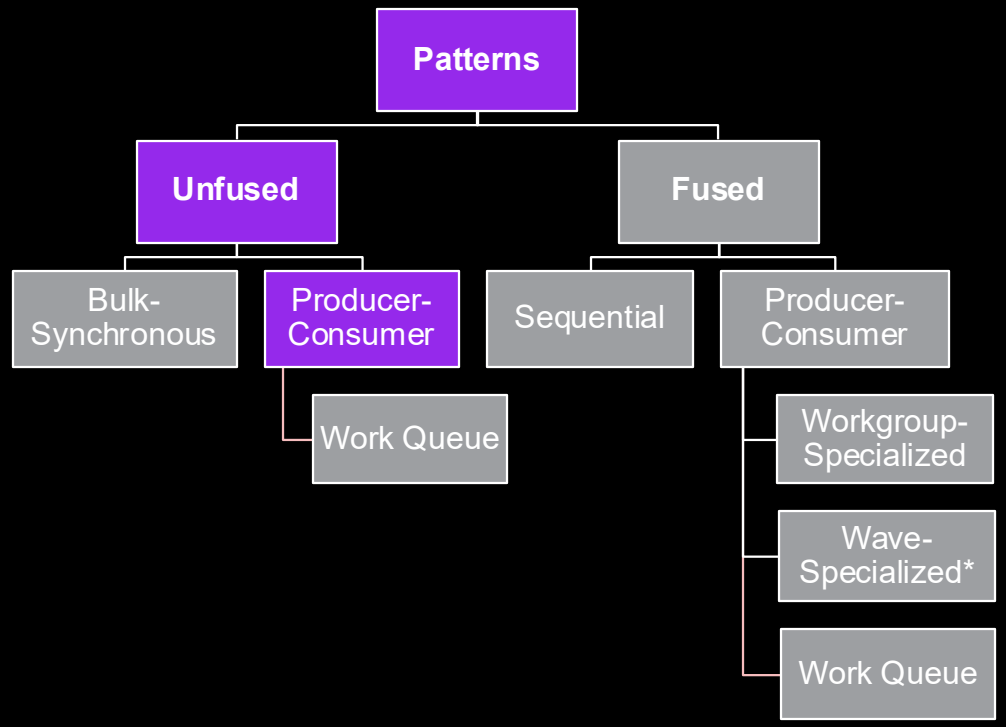
Launch Code

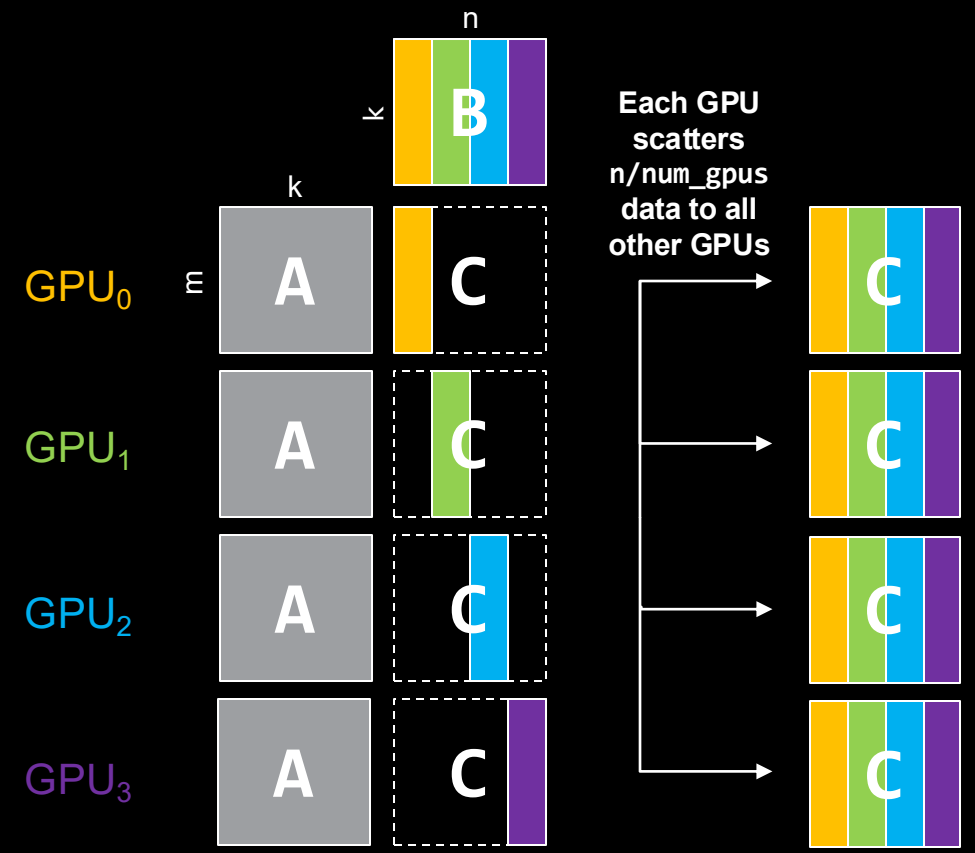
```python
with torch.cuda.stream(main_stream):
    C = persistent_gemm[(gemm_sms,)](
        A, B, C,
    )
with torch.cuda.stream(main_stream):
    persistent_all_scatter[(comm_sms,)](
        C,
    )
```

iris Unfused, Bulk-Synchronous



`iris` Taxonomy of Fused & Unfused Patterns (e.g., All-Scatter)

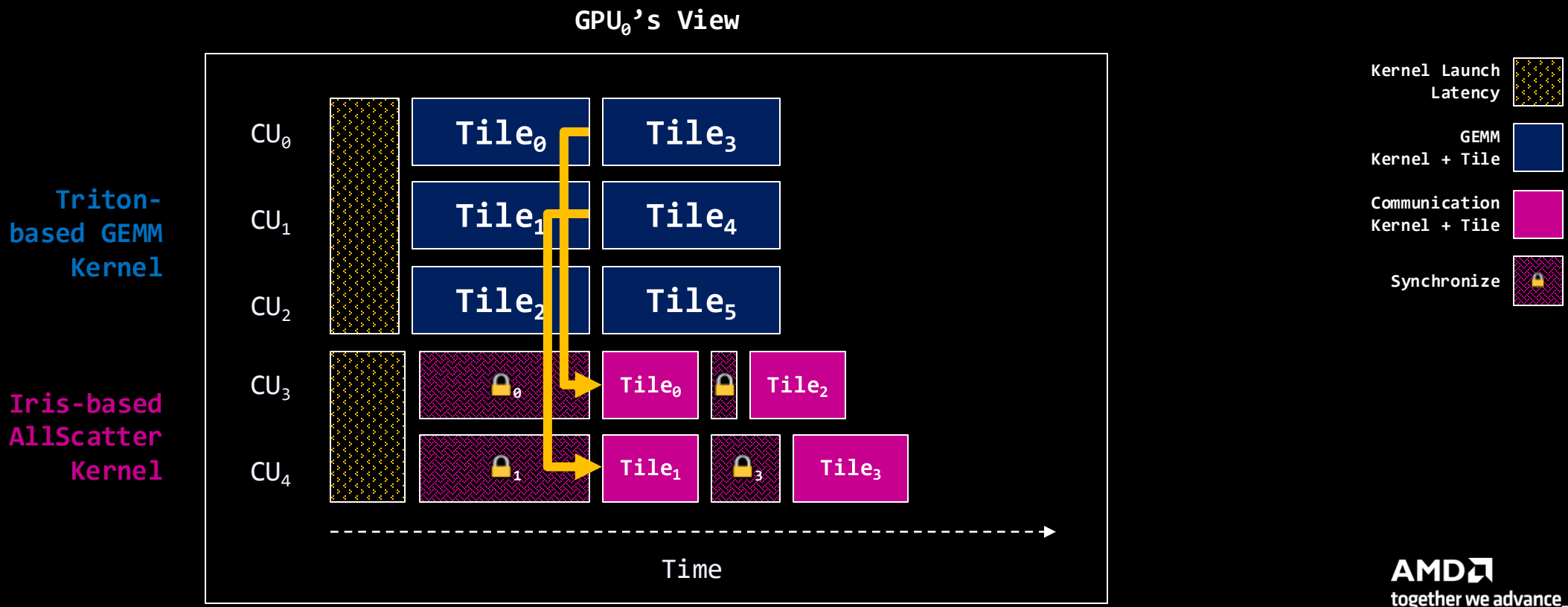




* Not possible without in Triton lang (needs Gluon)

iris Unfused, Producer-Consumer

- Asynchronously launch GEMM and All-Scatter in separate kernels (on different HIP streams)
- **GEMM,** Produce a tile, unlock a lock
- **All-Scatter,** Spin on the lock until unlocked and scatter the tile



iris Unfused, Producer-Consumer

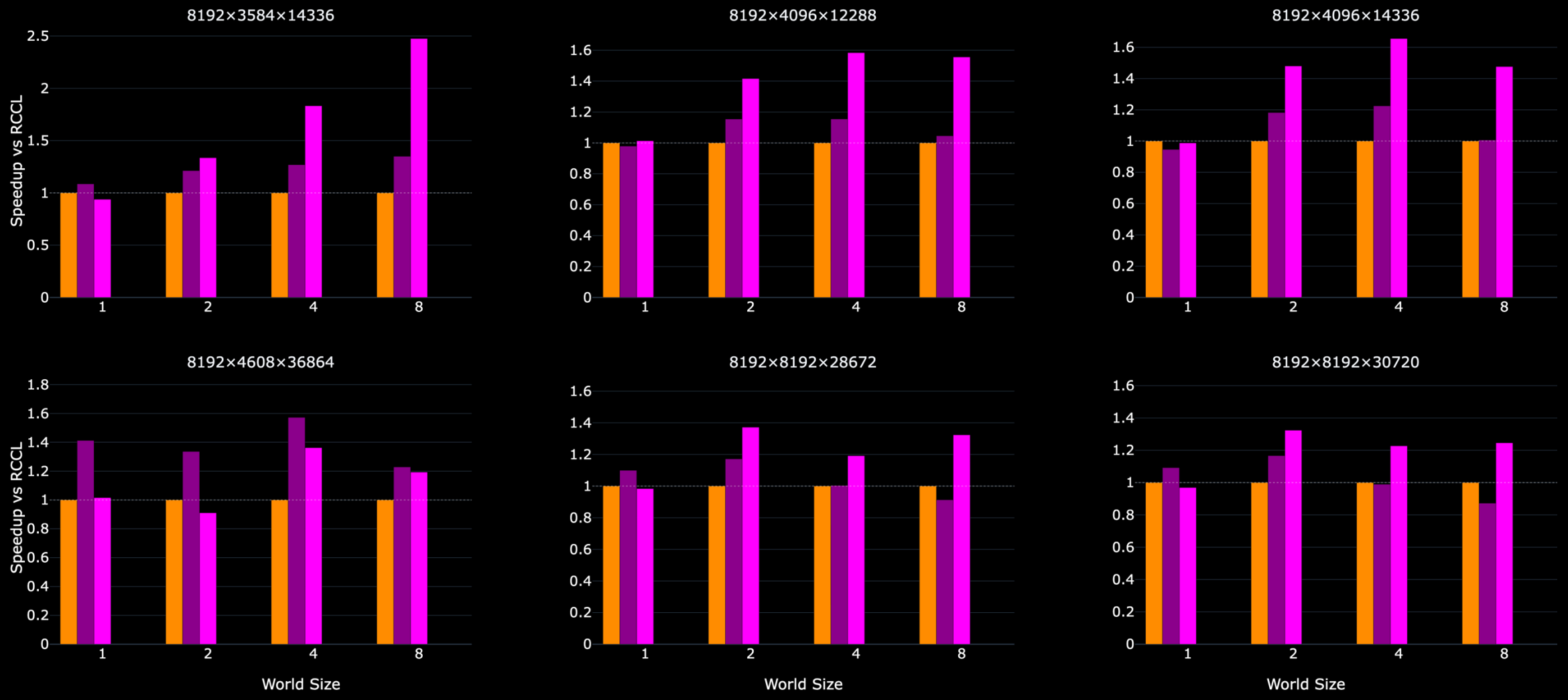
```python
@triton.jit()
def persistent_gemm(
    A,
    B,
    C, locks,
):
    pid = tl.program_id(0)

    for tile_id in range(pid, total_tiles, GEMM_SMS):

        acc = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N),
              dtype=acc_dtype)
        ...

        for k in range(0, loop_k):
            a = tl.load(A_BASE)
            b = tl.load(B_BASE)
            acc += tl.dot(a, b)
            A_BASE += BLOCK_SIZE_K * stride_ak
            B_BASE += BLOCK_SIZE_K * stride_bk

        # Accumulator registers with C results
        c = acc.to(C.type.element_ty)

        tl.store(C + global_offset, c, mask=sub_mask,
          cache_modifier=".wt")
        tl.atomic_cas(locks + tile_id, 0, 1,
           sem="release", scope="gpu")
```

```python
@triton.jit()
def persistent_all_scatter(
    C, locks,
):

    pid = tl.program_id(0)


    for tile_id in range(pid, total_tiles, COMM_SMS):

        while tl.atomic_cas(locks + tile_id, 1, 0,
           sem="acquire", scope="gpu") == 0:
            pass

        for remote_rank in range(world_size):
            if remote_rank != cur_rank:
                iris.put(C + global_offset, C +
                global_offset, cur_rank, remote_rank,
                heap_bases, mask=sub_mask)
```

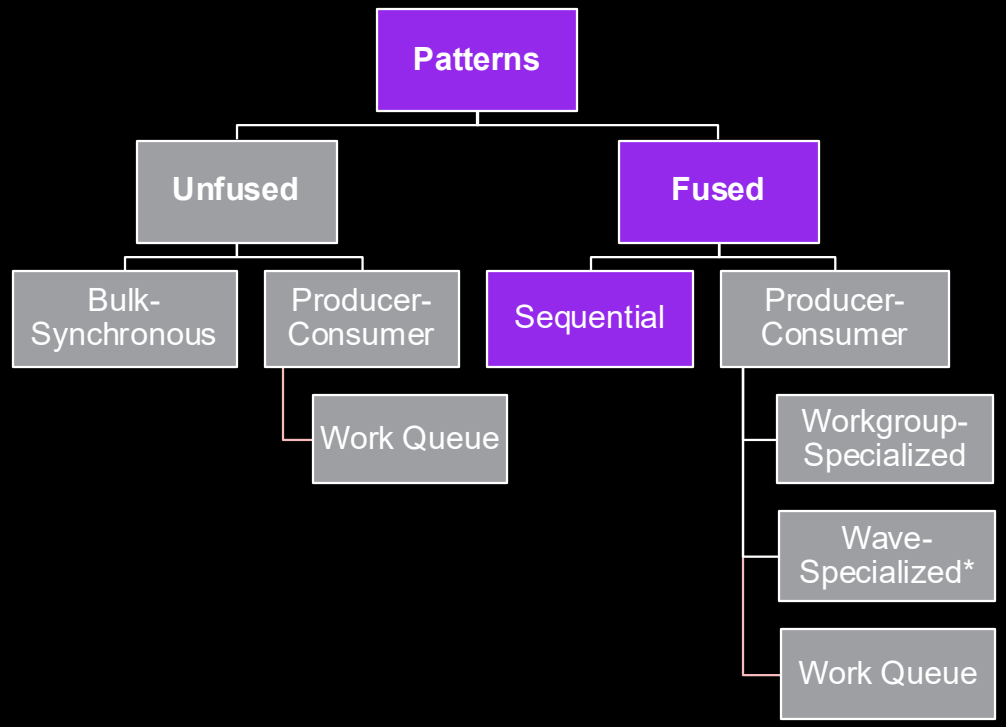
Launch Code

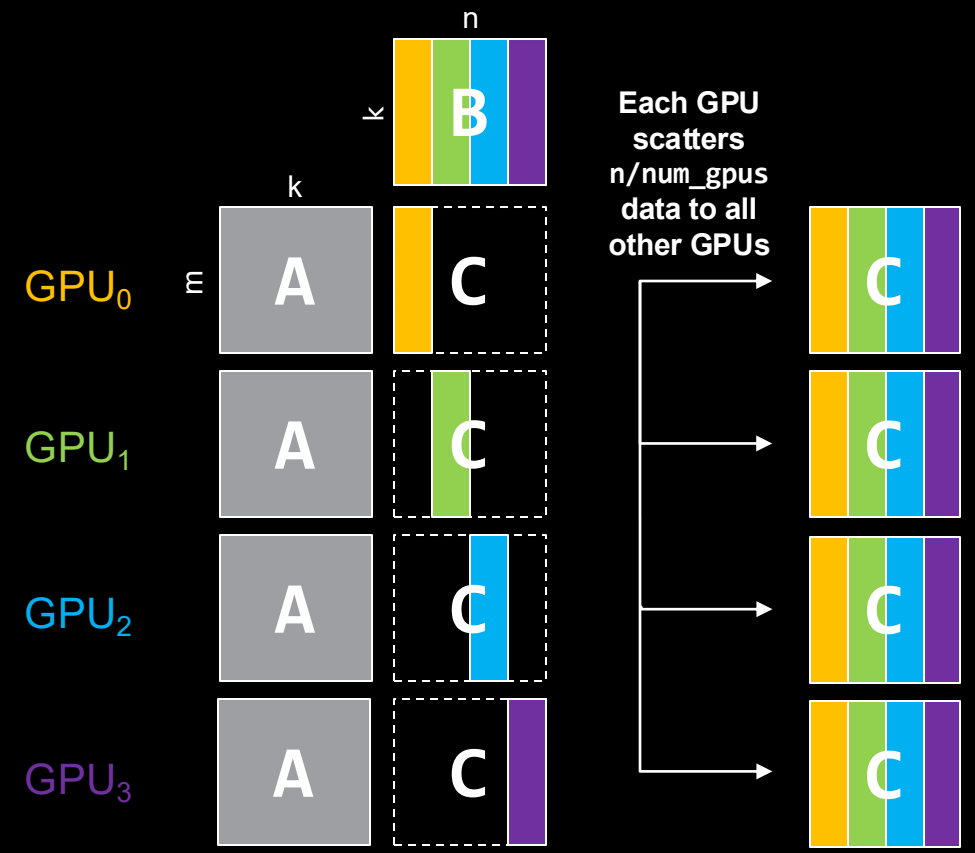
```python
with torch.cuda.stream(gemm_stream):
    C = persistent_gemm[(gemm_sms,)](
        A, B, C,
    )
with torch.cuda.stream(comm_stream):
    persistent_all_scatter[(comm_sms,)](
        C,
    )
```

iris Unfused, Producer-Consumer



`iris` Taxonomy of Fused & Unfused Patterns (e.g., All-Scatter)

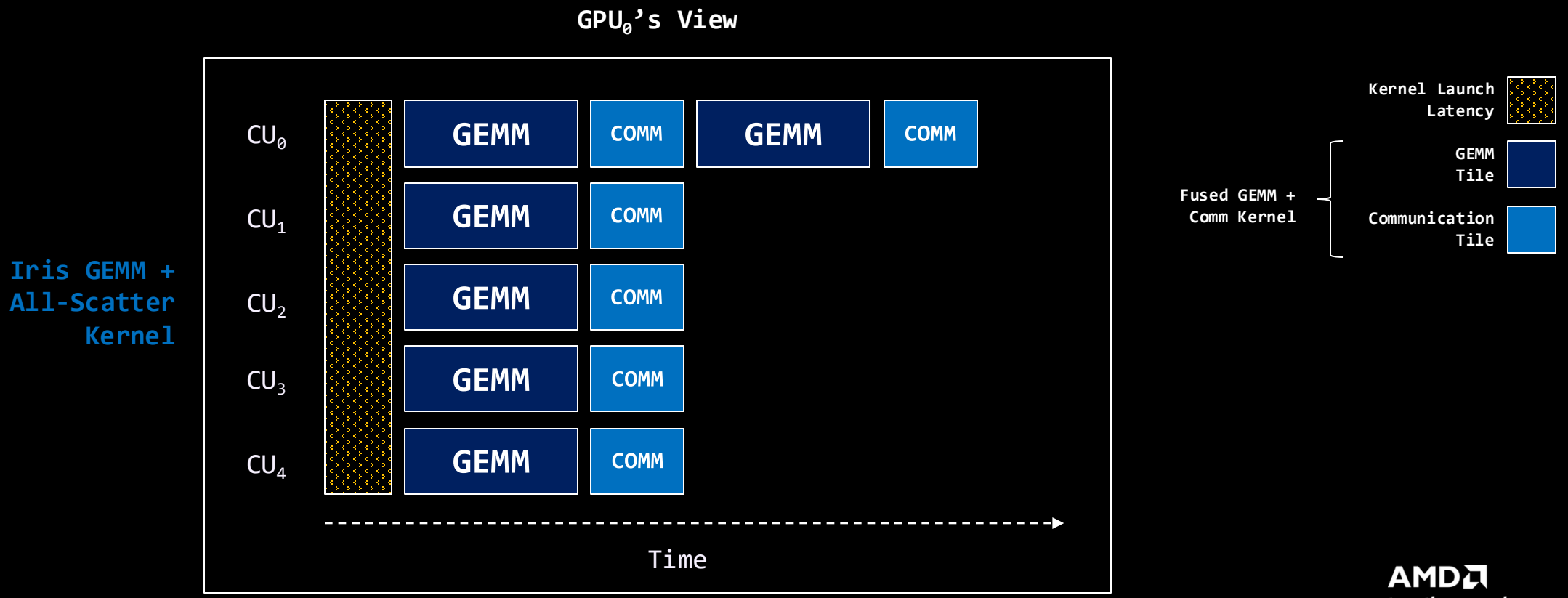




* Not possible without in Triton lang (needs Gluon)

`iris` Fused, Sequential

- Launch a single kernel with both GEMM and All-Scatter
- **GEMM,** Produce a tile, and immediately **scatter** it to other GPUs



iris Fused, Sequential

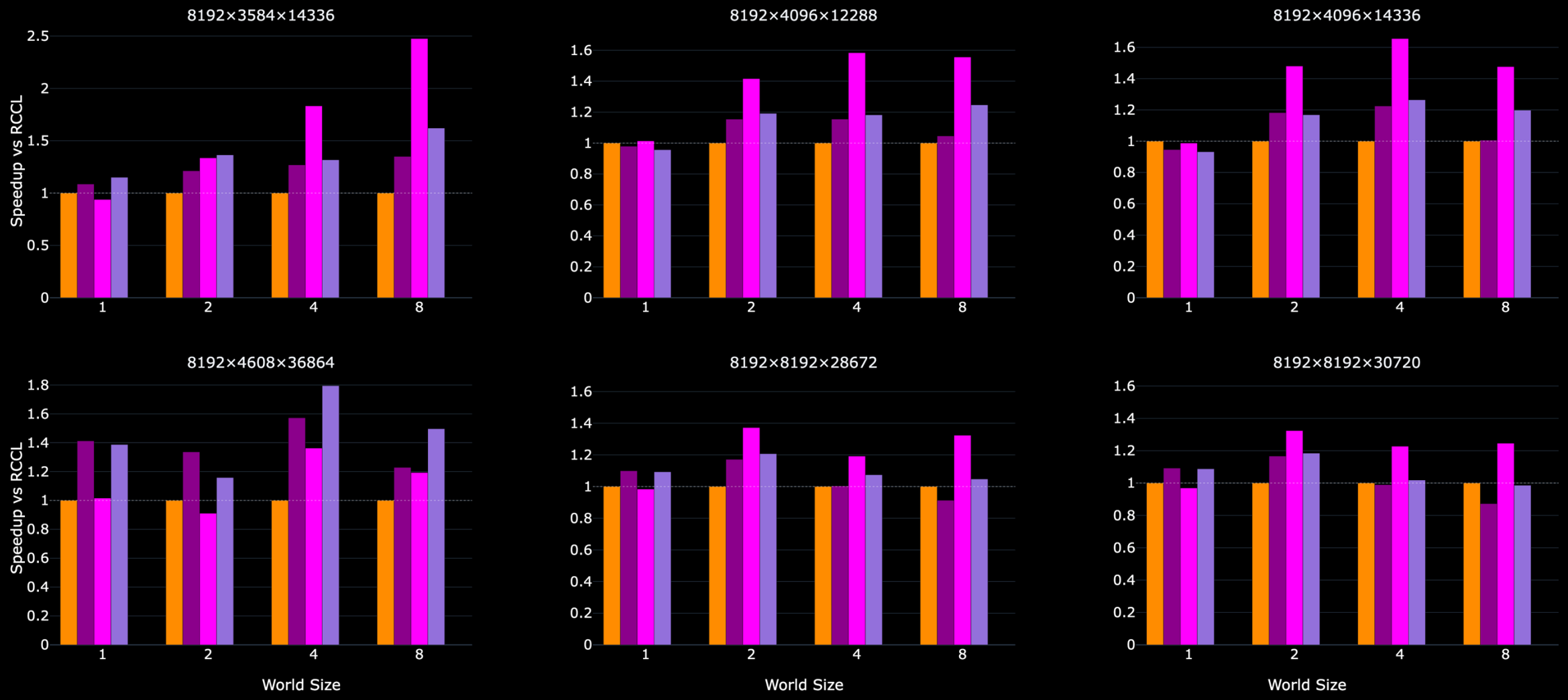
```python
@triton.jit()
def persistent_gemm_all_scatter(
    A,
    B,
    C,
):
    pid = tl.program_id(0)

    for tile_id in range(pid, total_tiles, GEMM_SMS):

        acc = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N),
              dtype=acc_dtype)
        ...

        for k in range(0, loop_k):
            a = tl.load(A_BASE)
            b = tl.load(B_BASE)
            acc += tl.dot(a, b)
            A_BASE += BLOCK_SIZE_K * stride_ak
            B_BASE += BLOCK_SIZE_K * stride_bk

        # Accumulator registers with C results
        c = acc.to(C.type.element_ty)
```

```python
        # Store data to the global result using puts
        for remote_rank in range(world_size):
            if remote_rank == cur_rank:
                # For the current rank, we can use store
                tl.store(c_global + global_offset, c,
                   mask=sub_mask)
            else:
                iris.store(
                    c_global + global_offset,
                    c,
                    cur_rank,
                    remote_rank,
                    heap_bases,
                    mask=sub_mask,
                )
```

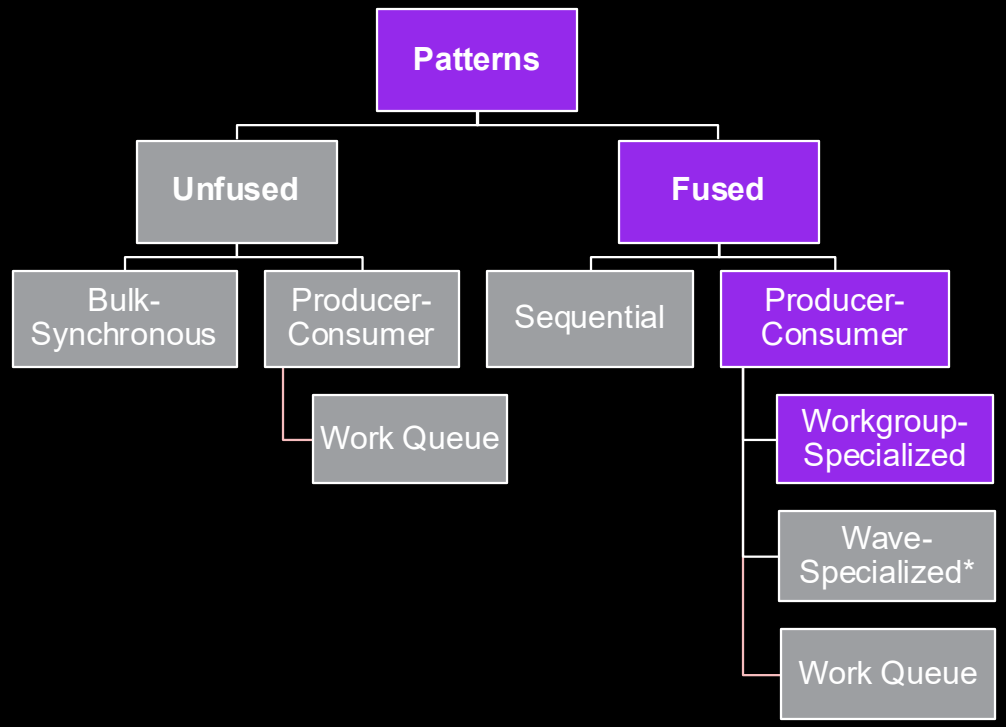
Launch Code

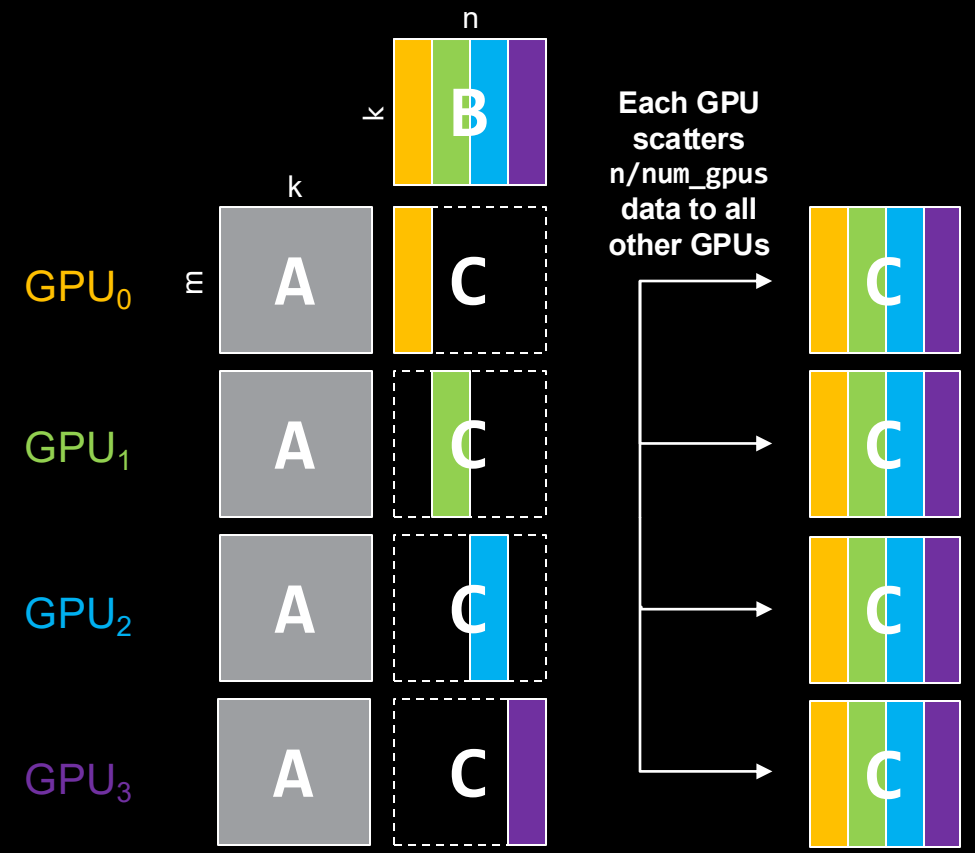
```python
with torch.cuda.stream(main_stream):
    C = persistent_gemm_all_scatter[(num_cus,)](
        A, B, C,
    )
```

iris Fused, Sequential



`iris` Taxonomy of Fused & Unfused Patterns (e.g., All-Scatter)

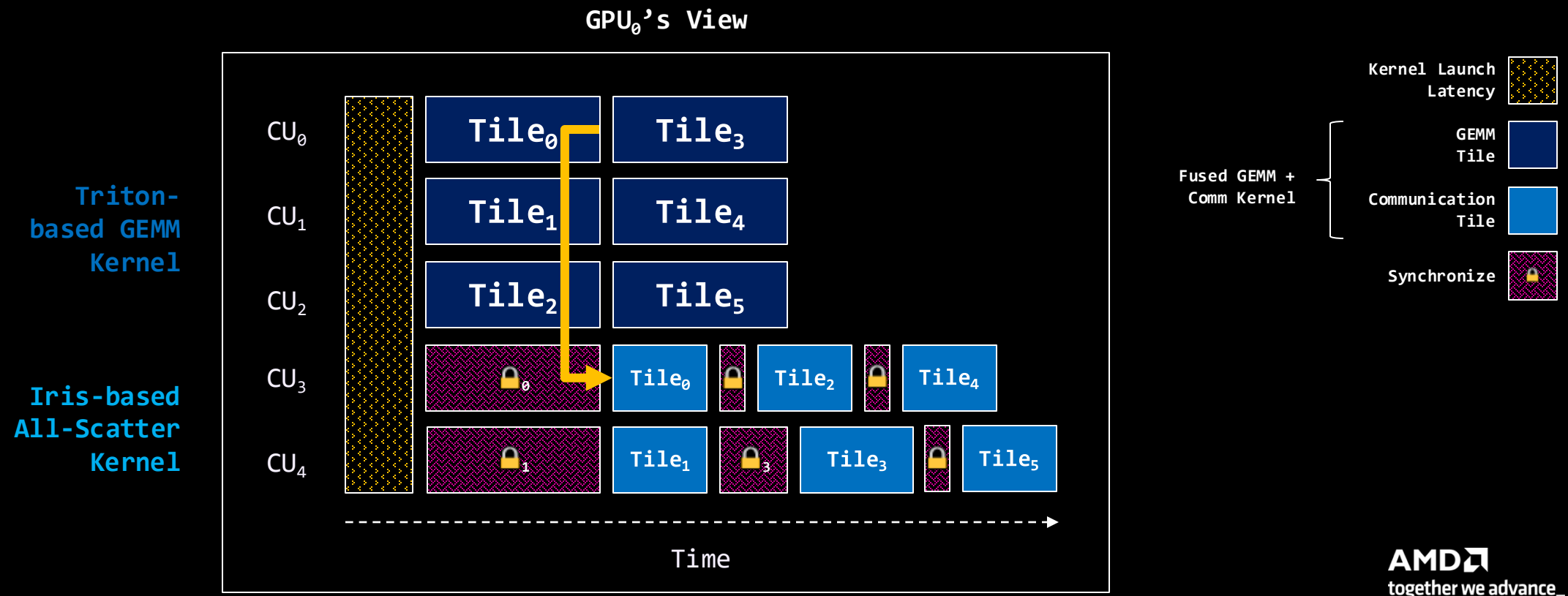




* Not possible without in Triton lang (needs Gluon)

`iris` Fused, Workgroup-Specialized

- Launch a single kernel with both GEMM and All-Scatter
- Use Workgroup ID (`pid`) to direct some workgroups to **GEMM** and some to **All-Scatter**
- Communication uses a **spinlock** to wait for the data to be ready, GEMM unlocks the lock when a tile is produced



iris Fused, Workgroup-Specialized

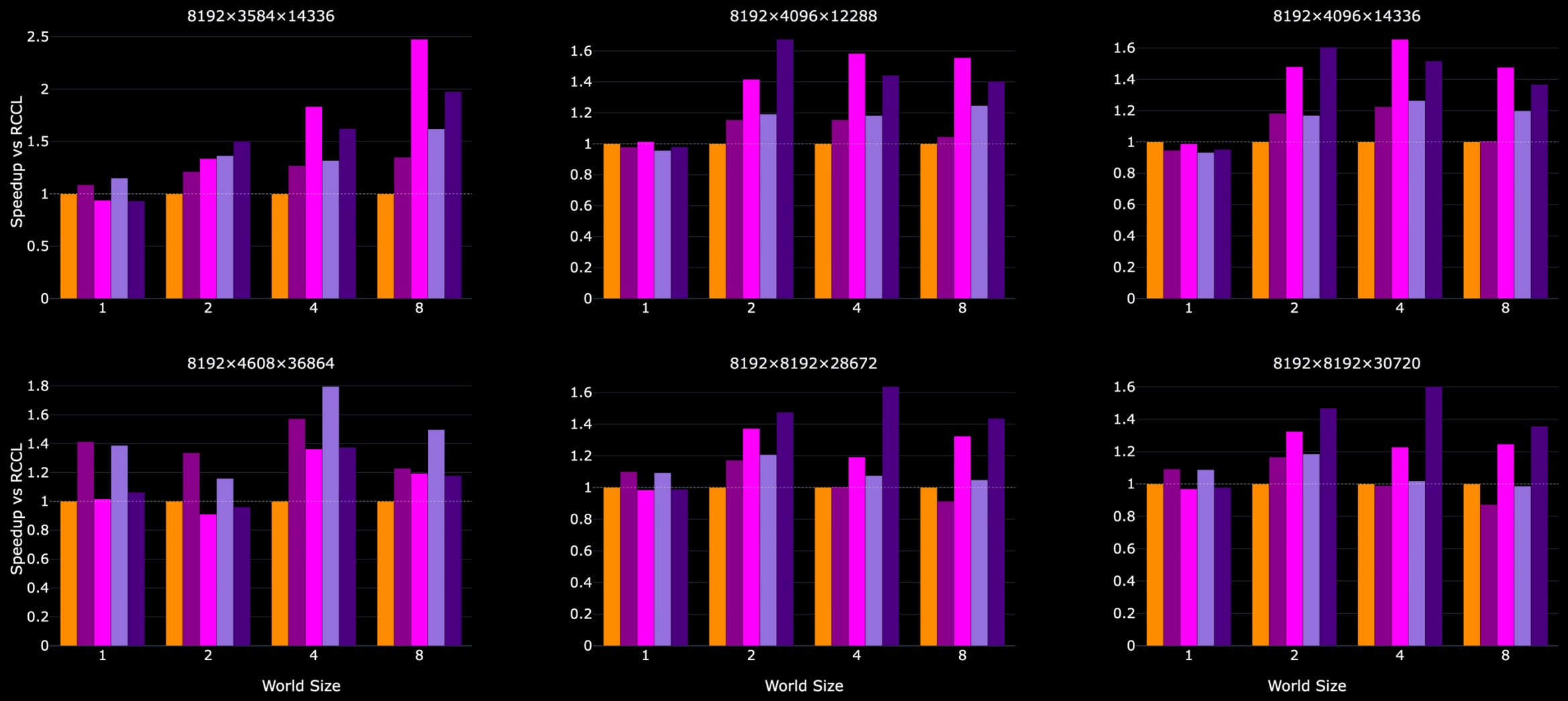
```python
@triton.jit()
def persistent_gemm_all_scatter(
    A,
    B,
    C,
):
    pid = tl.program_id(0)

    # Workgroup specialization:
    # Split the kernel into two paths, one that
    # performs the GEMM and another that performs the
    # communication. Uses persistent-kernel.
    if pid < GEMM_SMS:

        for tile_id in range(pid, total_tiles,
                             GEMM_SMS):

            ...

            for k in range(0, loop_k):
                a = tl.load(A_BASE)
                b = tl.load(B_BASE)
                acc += tl.dot(a, b)
                A_BASE += BLOCK_SIZE_K * stride_ak
                B_BASE += BLOCK_SIZE_K * stride_bk

            # Accumulator registers with C results
            c = acc.to(C.type.element_ty)
```

```python
            tl.store(c_global + global_offset, c, mask=sub_mask,
            cache_modifier=".wt")
            tl.atomic_cas(locks + tile_id, 0, 1, sem="release",
                   scope="gpu")

    else:  # pid >= GEMM_SMS
            COMM_SMS = NUM_SMS - GEMM_SMS
            pid = pid - GEMM_SMS
            for tile_id in range(pid, total_tiles, COMM_SMS):

                while tl.atomic_cas(locks + tile_id, 1, 0,
                            sem="acquire", scope="gpu") == 0:
                    pass

                for remote_rank in range(world_size):
                    if remote_rank != cur_rank:
                        iris.put(
                            c_global + global_offset,
                            c_global + global_offset,
                            cur_rank,
                            remote_rank,
                            heap_bases,
                            mask=sub_mask,
                        )
```

Launch Code

```python
with torch.cuda.stream(main_stream):
    C = persistent_gemm_all_scatter[(num_cus,)](
        A, B, C, GEMM_SMS, COMM_SMS
    )
```

iris Fused, Workgroup-Specialized

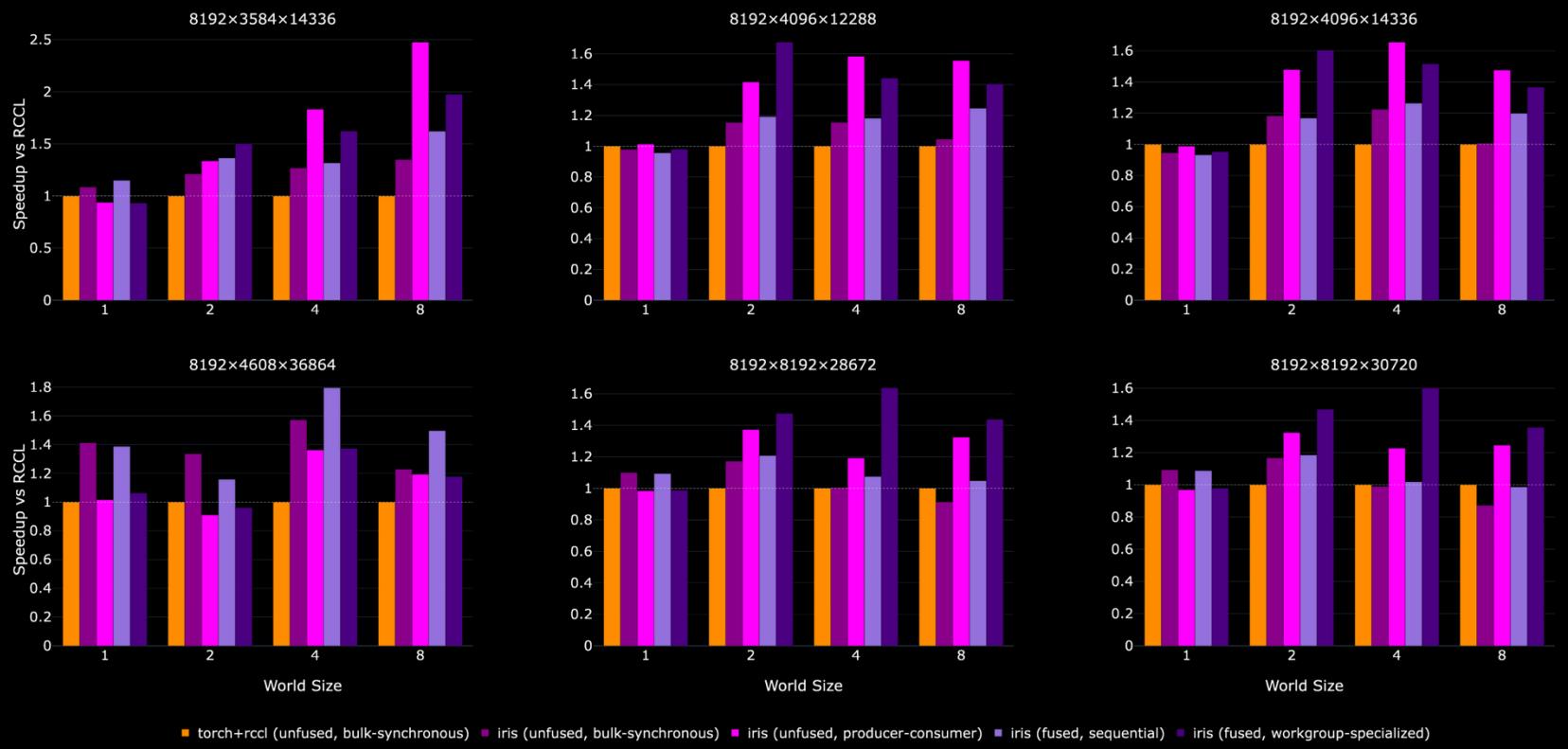


Which pattern should I use?

All of them!

Just with these 6 sizes, the answer is; it depends

`iris` argues that we should have an easy way to describe and experiment with any of these patterns, and our hardware should support all of them



`iris` + More Applications!

Application	Status
All-Load, All-Store	✓ Completed
FlashDecode	✓ Completed
Remote Atomics	✓ Completed
GEMM + All-Scatter (4)	✓ Completed
GEMM + One-Shot All-Reduce	✓ Completed
GEMM + Atomic All-Reduce	✓ Completed
GEMM + Ring-based All-Reduce	⏳ In Progress
All-Gather + GEMM	⏳ In Progress
GEMM + Reduce-Scatter	⏳ In Progress
Mixture of Experts (MoE)	⏳ In Progress
End-to-End Models (vLLM w/ Iris)	⏳ In Progress

https://github.com/ROCm/iris/tree/main/examples

The Scale-Out Plan

- Multi-GPU programming made easy with familiar, high-level abstractions
- Extend the well-defined HIP/CUDA memory models using locality-aware memory scopes

- Iris provides:
  - Low-level RDMA primitives
  - Optimized collectives through clear examples
- Iris enables fast prototyping of fine-grained communication/computation overlap

```
# Synchronization Scopes
block                   # Block scope
gpu                     # GPU scope
sys                     # System scope
world                   # World scope

# Memory orders
relaxed                 # Relaxed memory order
acquire                 # Acquire memory order
release                 # Release memory order
acq_rel                 # Consume memory order
```

Next Steps for Iris and On-going Work

Library Development

- Integration with inference frameworks
- Library improvements and productization
- Scale-out backend

Applications and Workloads

- GEMM + Ring-based All-Reduce
- All-Gather + GEMM
- GEMM + Reduce-Scatter
- Mixture of Experts (MoE)
- More examples, docs, and use cases — across all domains!

```
import iris
Github.com/ROCm/iris
```

We built Iris for you. Use it, break it, improve it — your feedback and PRs shape its future. And Iris will always remain open source.

Disclaimer and Attribution


AMD