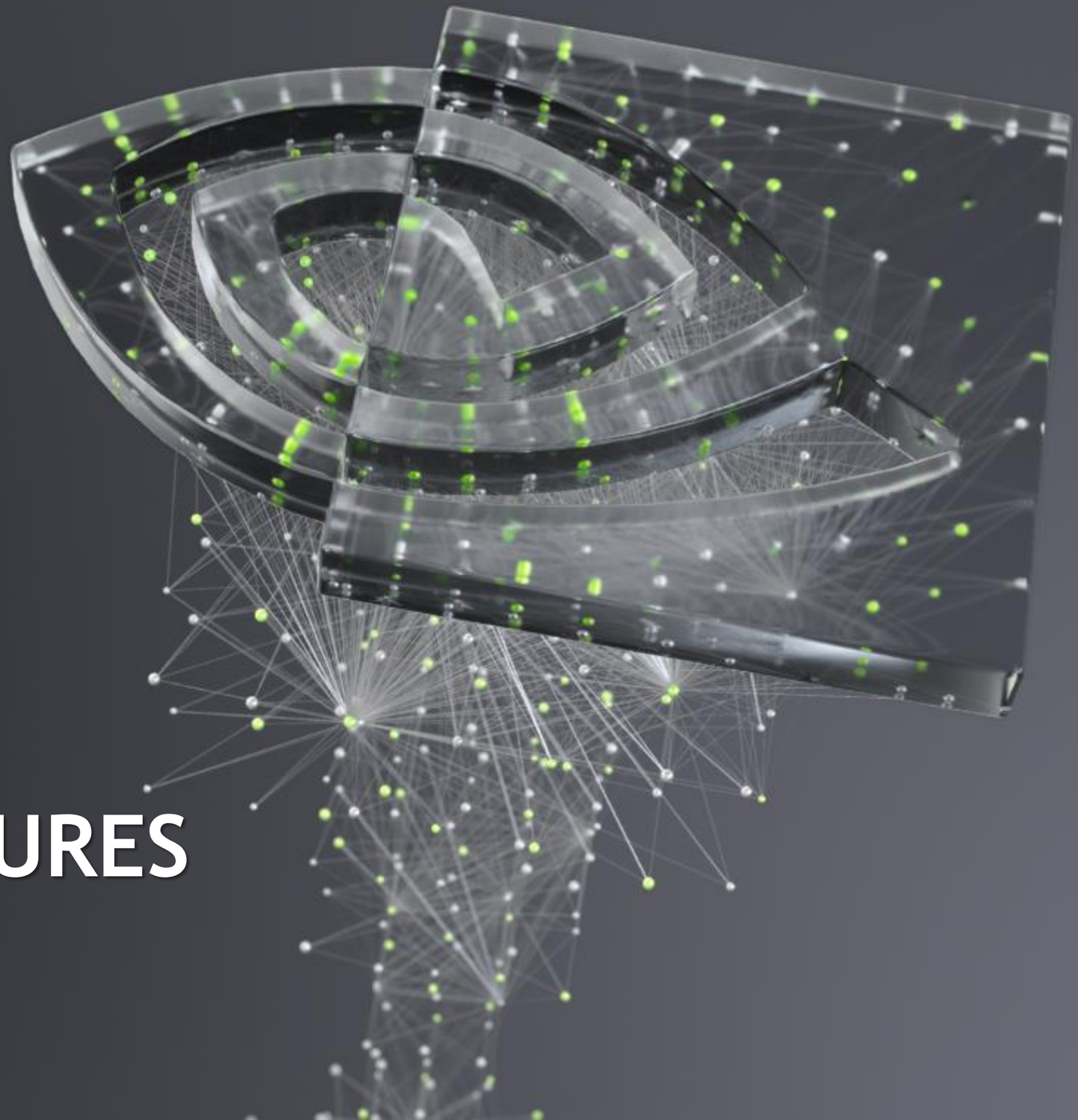


# CUDA 11 NEW FEATURES

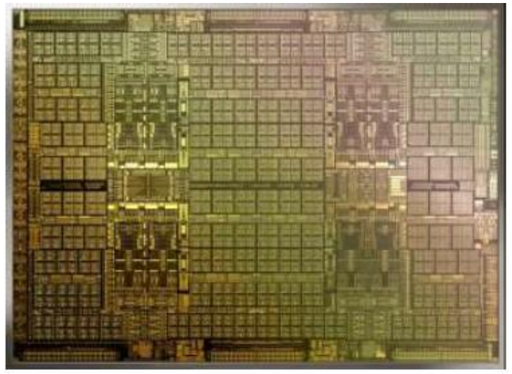
Jingrong Zhang, GTC CHINA

# AGENDA

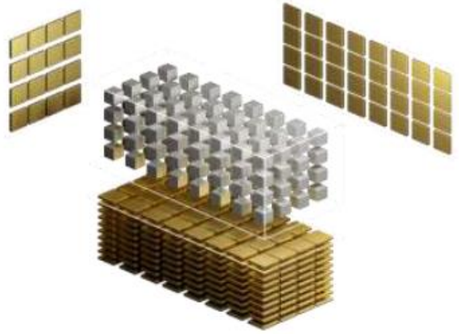
- NVIDIA A100 Highlights
- Programing with CUDA 11
  - Warp Synchronous Reduction
  - L2 Cache Residency Control
  - Asynchronous copy
  - Asynchronous barrier

# NVIDIA A100 HIGHLIGHTS

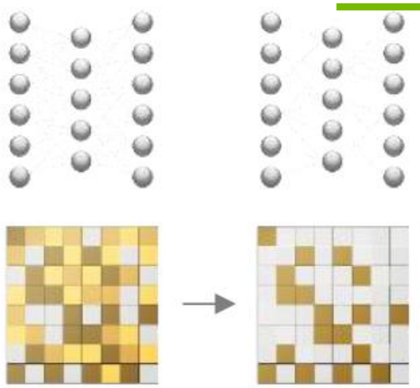
## 5 Miracles of NVIDIA A100



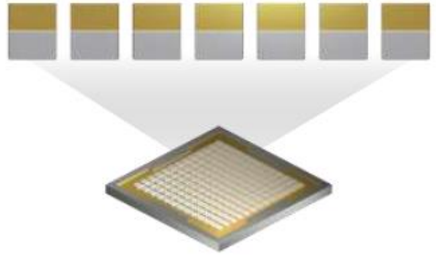
Ampere
World's Largest 7nm chip
54B XTORS, HBM2



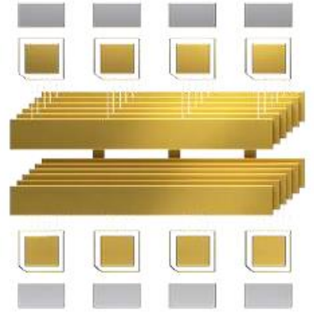
3rd Gen Tensor Cores
Faster, Flexible, Easier to use
20x AI Perf with TF32



New Sparsity Acceleration
Harness Sparsity in AI Models
2x AI Performance



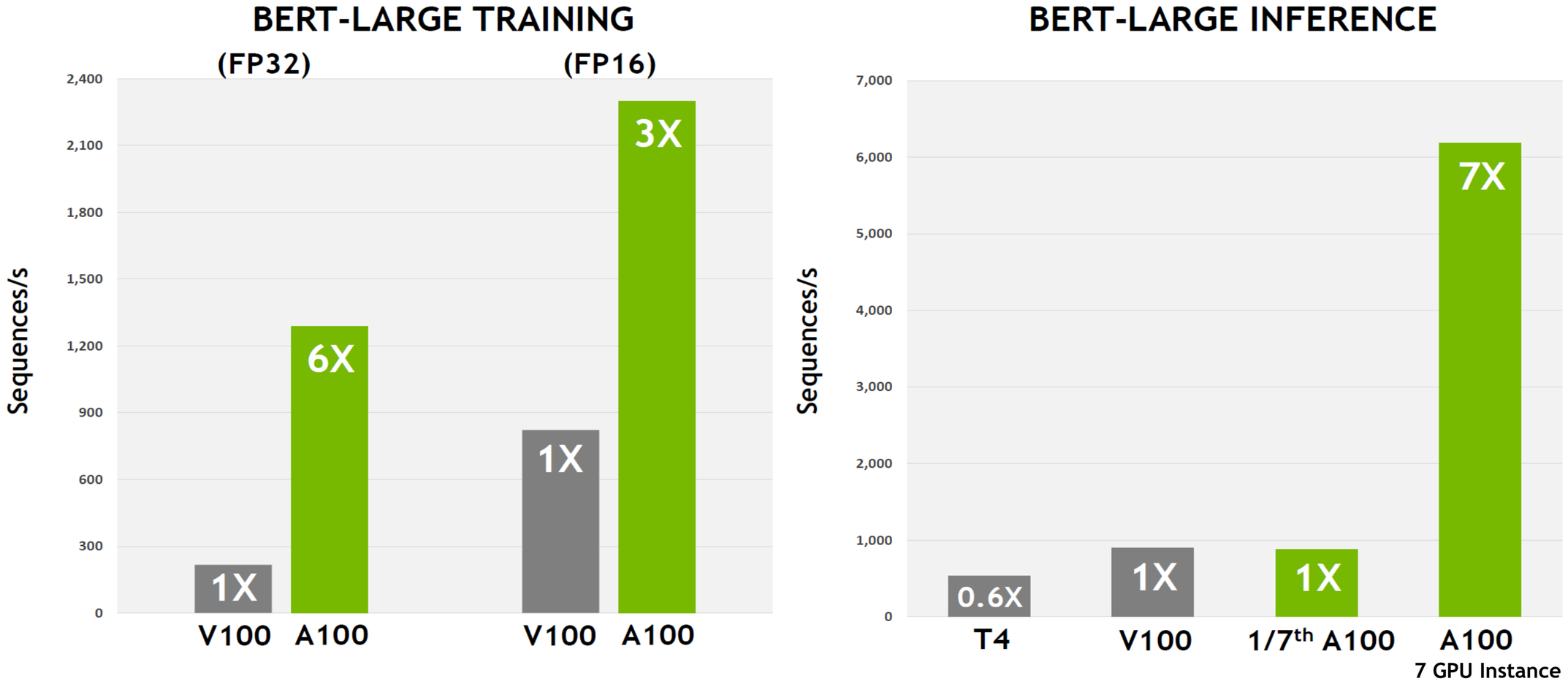
New Multi-Instance GPU
Optimal utilization with right sized GPU
7x Simultaneous Instances per GPU



3rd Gen NVLINK and NVSWITCH
Efficient Scaling to Enable Super GPU
2X More Bandwidth

https://www.nvidia.com/en-us/data-center/a100/

# UNIFIED AI ACCELERATION

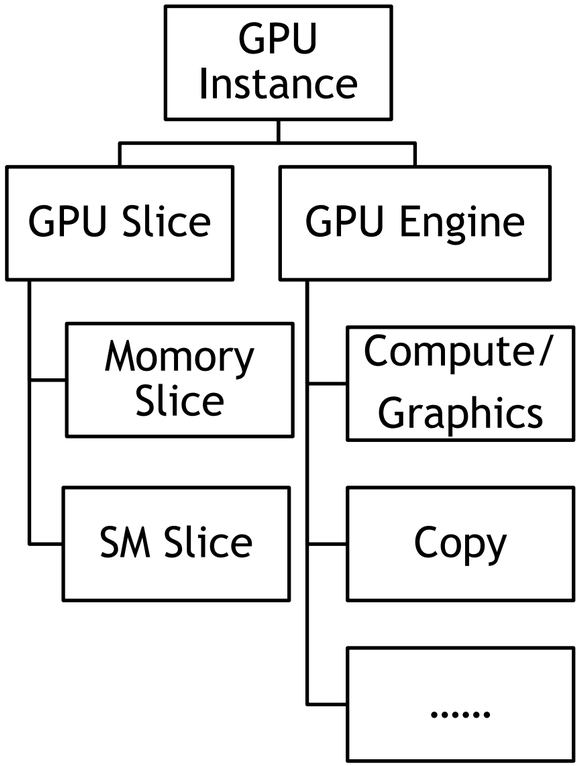



All results are measured
BERT Large Training (FP32 & FP16) measures Pre-Training phase, uses PyTorch including (2/3) Phase1 with Seq Len 128 and (1/3) Phase 2 with Seq Len 512,
V100 is DGX1 Server with 8xV100, A100 is DGX A100 Server with 8xA100, A100 uses TF32 Tensor Core for FP32 training
BERT Large Inference uses TRT 7.1 for T4/V100, with INT8/FP16 at batch size 256. Pre-production TRT for A100, uses batch size 94 and INT8 with sparsity

# MULTI-INSTANCE GPU (MIG)

## GPU Instance

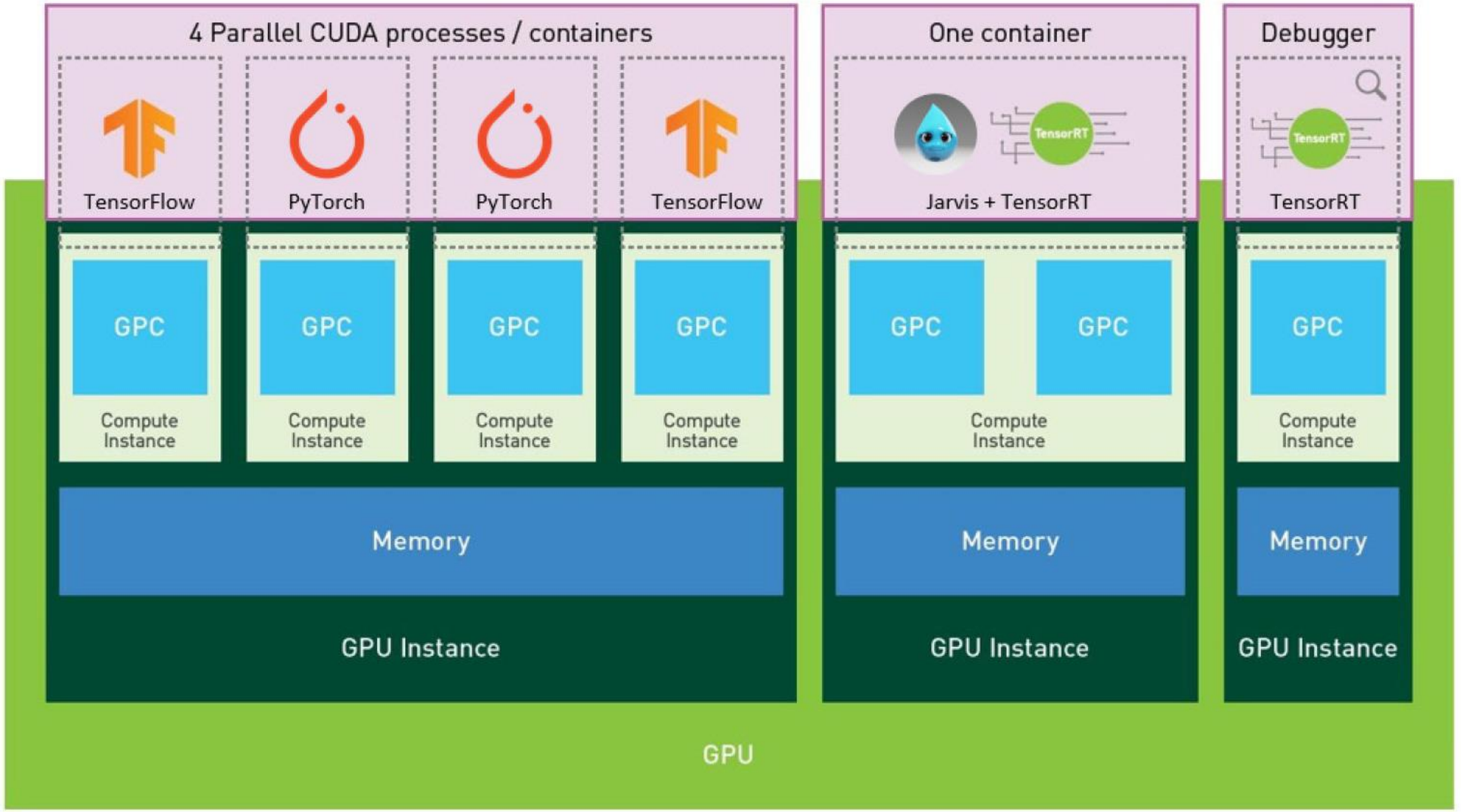
- Simultaneous Workload Execution With Guaranteed **Quality Of Service**
- Different sized MIG instances based on target workloads

# MULTI-INSTANCE GPU (MIG)

## Compute Instance

- A GPU instance can be subdivided into multiple **compute instances.**
- A Compute Instance (CI) contains **a subset of the parent GPU instance's SM slices and other GPU engines.**
- The CIs share **memory and engines.**




https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html#concepts

# MULTI-INSTANCE GPU (MIG)

## Partition

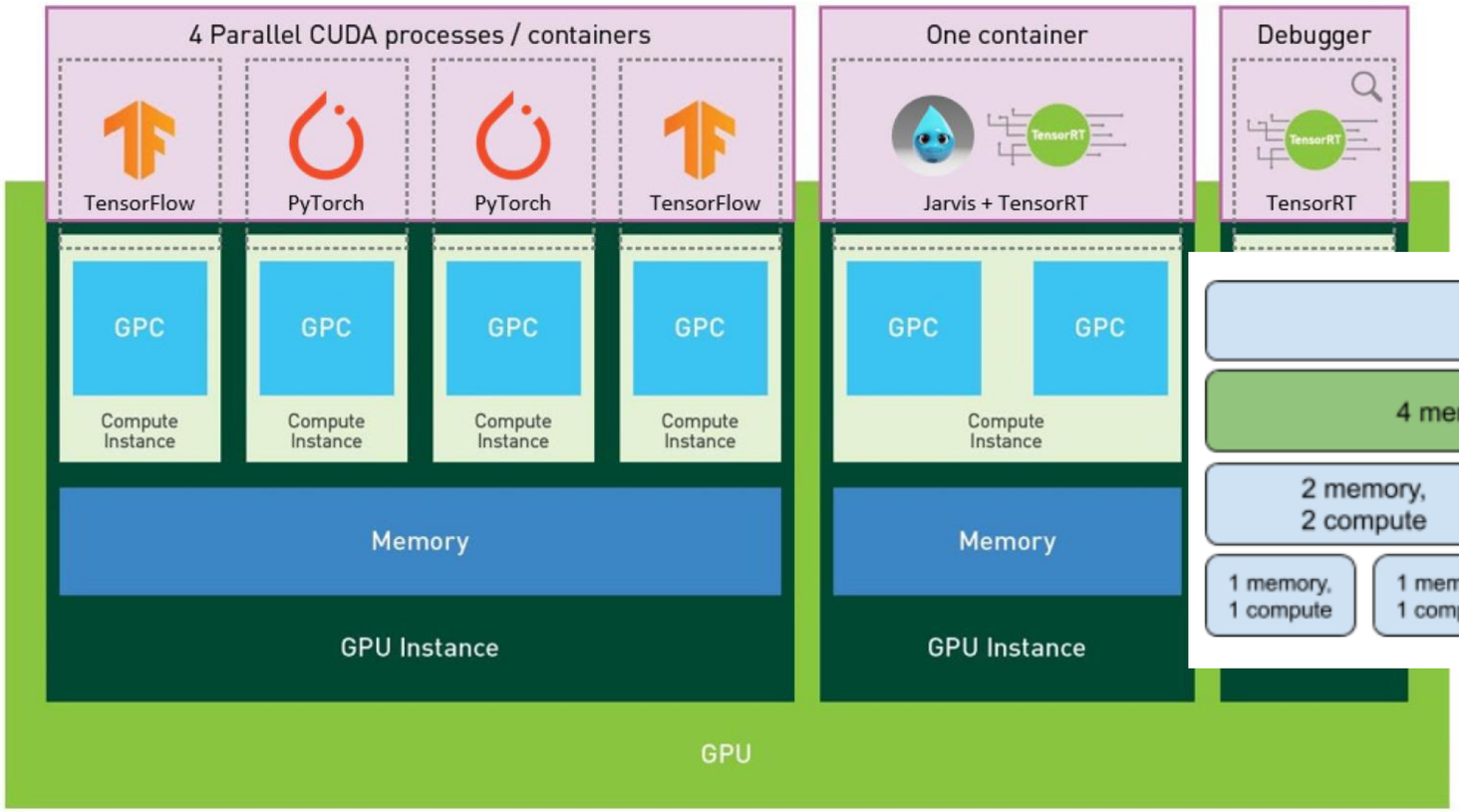
- Up To 7 GPU Instances
- The number of slices that a GI can be created with is **not arbitrary**.
- Users can create GIs by specifying **one of these profiles**

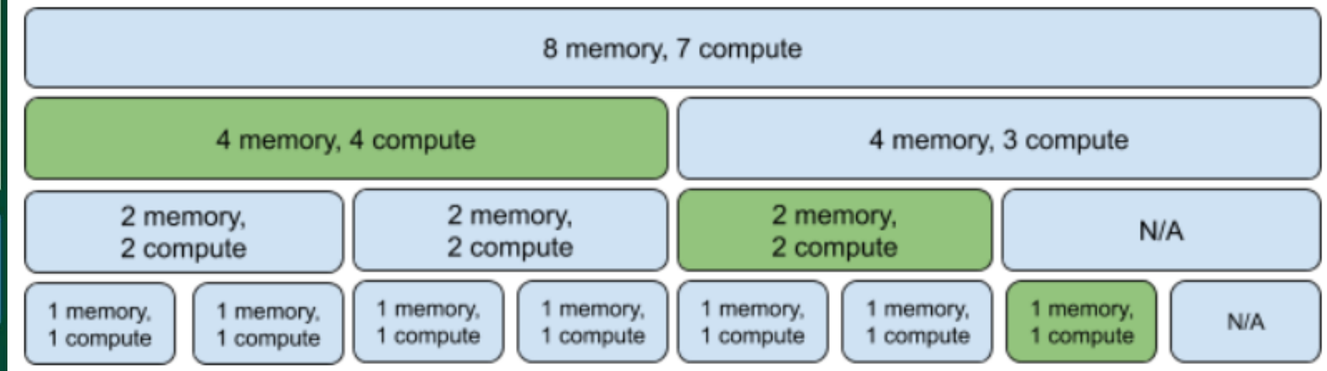
| Profile Name | Number of Instances available | Fraction of SMs | Fraction of Memory | Memory bandwidth | Number of NVDECs | NVJPG and NVOFA |
|---|---|---|---|---|---|---|
| MIG 1g.5gb | 7 | 1/7 | 1/8 | 1/8 | None | None |
| MIG 2g.10gb | 3 | 2/7 | 2/8 | 1/4 | 1 | None |
| MIG 3g.20gb | 2 | 3/7 | 4/8 | 1/2 | 2 | None |
| MIG 4g.20gb | 1 | 4/7 | 4/8 | 1/2 | 2 | None |
| MIG 7g.40gb | 1 | 7/7 | Full | Full | 5 | 1 |

https://mp.weixin.qq.com/s/ASkkgfcrdQNahfQnRWbHdw
https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html#concepts

# MULTI-INSTANCE GPU (MIG)

## Partition

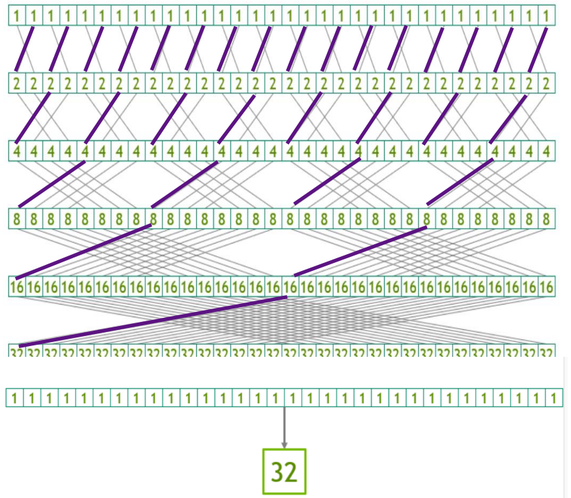


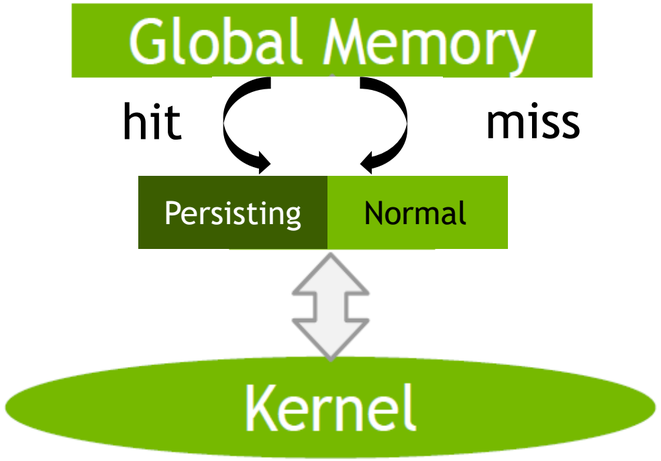

https://mp.weixin.qq.com/s/ASkkgfcrdQNahfQnRWbHdw
https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html#concepts

# AGENDA

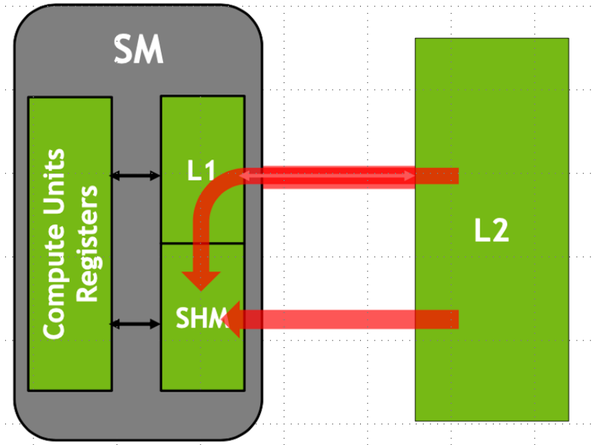
- NVIDIA A100 Highlights
- Programing with CUDA 11
  - Warp Synchronous Reduction
  - L2 Cache Residency Control
  - Asynchronous copy
  - Asynchronous barrier



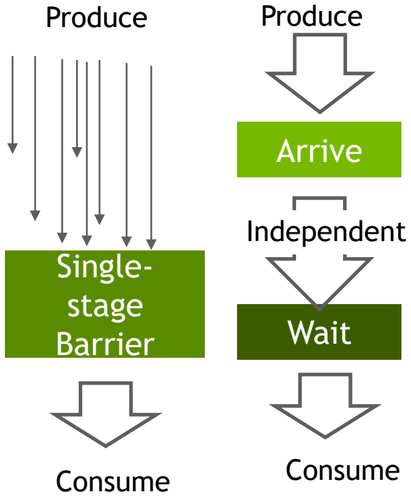

Warp Synchronous Reduction



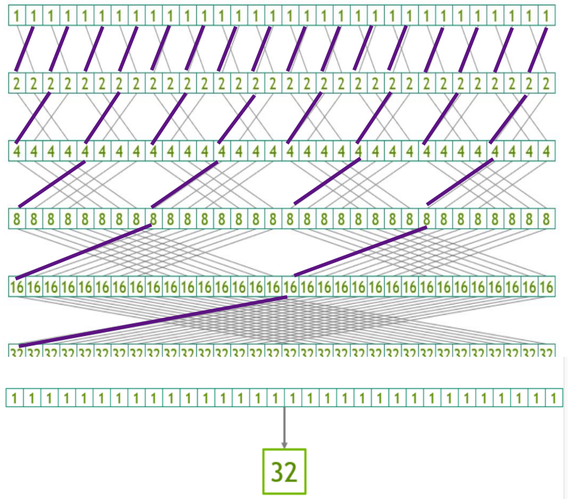

L2 cache residency controls



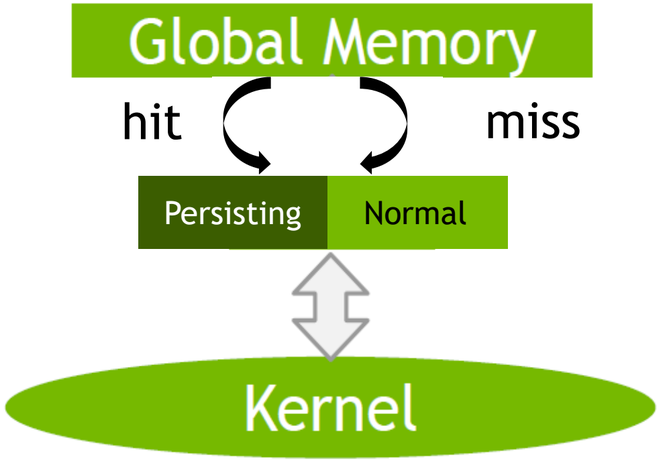

Asynchronous copy



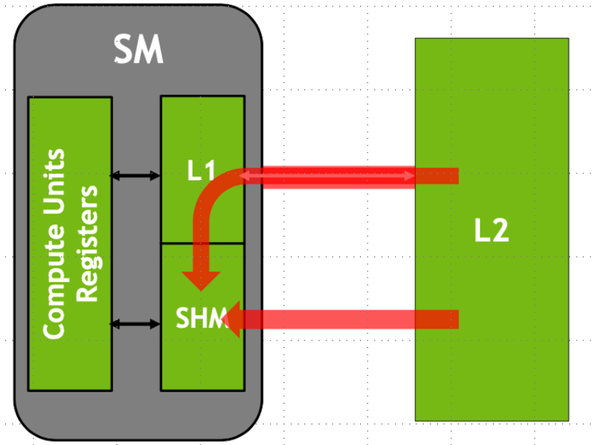

Asynchronous barrier



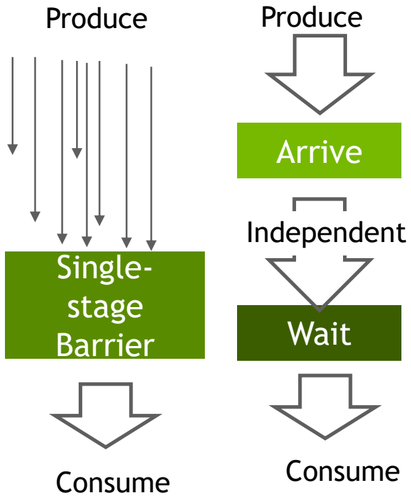

**Warp Synchronous Reduction**




L2 cache residency controls



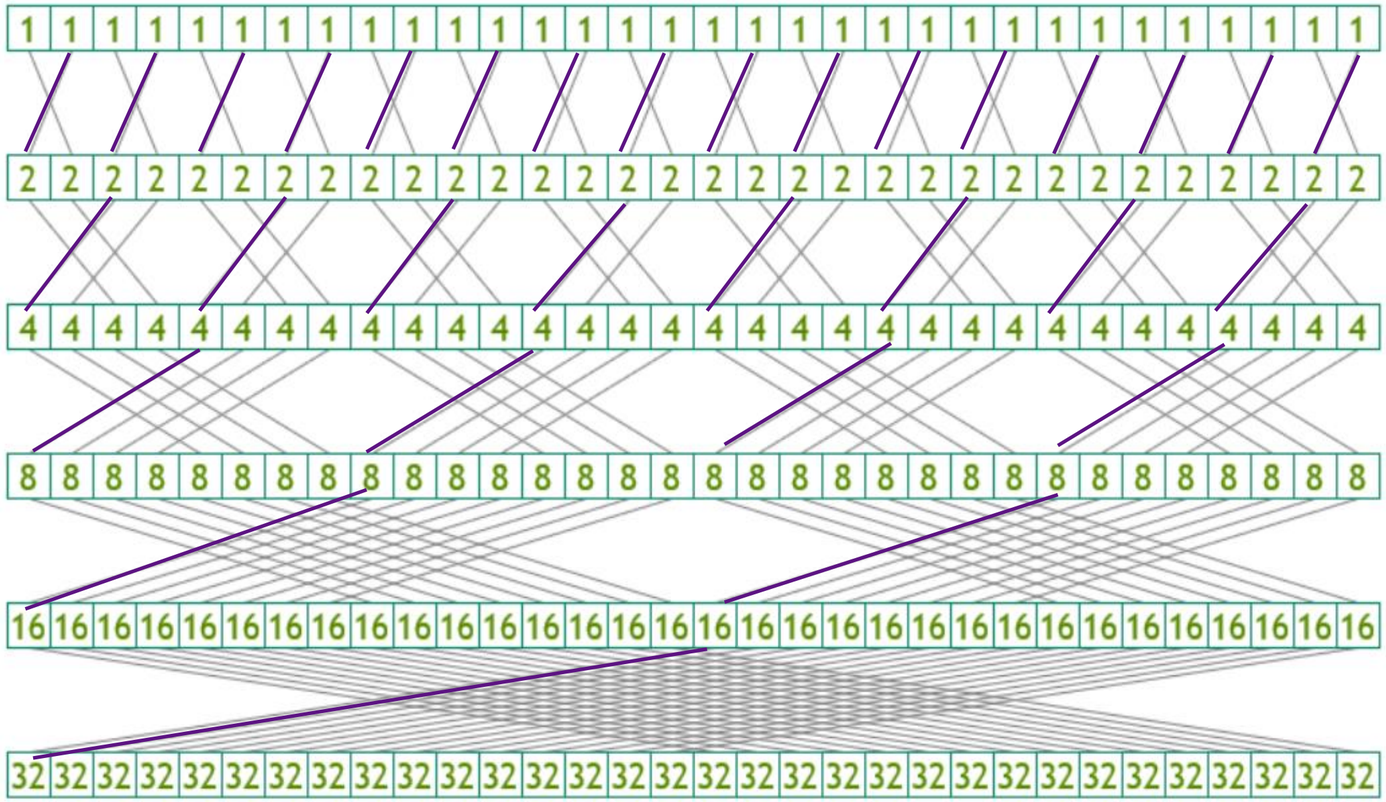

Asynchronous copy




Asynchronous barrier

# WARP-WIDE REDUCTION IN A SINGLE STEP

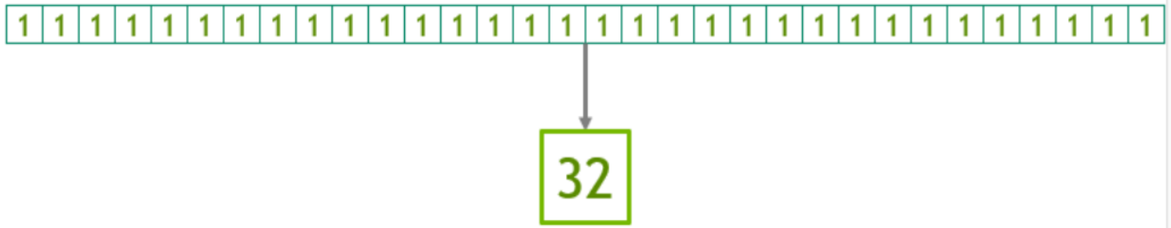
Pre-A100 warp-scope reductions are based on the SHFL operation and require 5 steps to complete




```
__device__ int reduce(int value) {
    value += __shfl_xor_sync(0xFFFFFFFF, value, 1);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 2);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 4);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 8);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 16);

    return value;
```

# WARP-WIDE REDUCTION IN A SINGLE STEP

The A100 GPU has **hardware-accelerated** reductions which produce a result in a single step




```
int total = __reduce_add_sync(0xFFFFFFFF, value);
```

```
thread_block_tile<32> tile32 =
            tiled_partition<32>(this_thread_block());

// Works on all GPUs back to Kepler
cg::reduce(value, tile32, cg::plus<int>());
```

# CUDA COOPERATIVE GROUPS

## Data manipulation

- For the **arithmetic** add, min, or max operations and the **logical** AND, OR, or XOR

```
template <typename TyVal, typename TyArg, template <class> class TyOp, typename
 TyGroup>
TyVal reduce(const TyGroup& group, TyArg&& val, TyOp<TyVal>&& op);
```

| | |
|---|---|
| **group** | **coalesced_group / thread_block_tile** |
| **val** | Only **4B types** are accelerated by hardware.<br>Only **unsigned int** for logical operators |
| **op** | **plus(), less(), greater(), bit_and(), bit_xor(), bit_or()** |

Additional types and operations are implemented in software as well as fallbacks on older generation hardware

# CUDA COOPERATIVE GROUPS

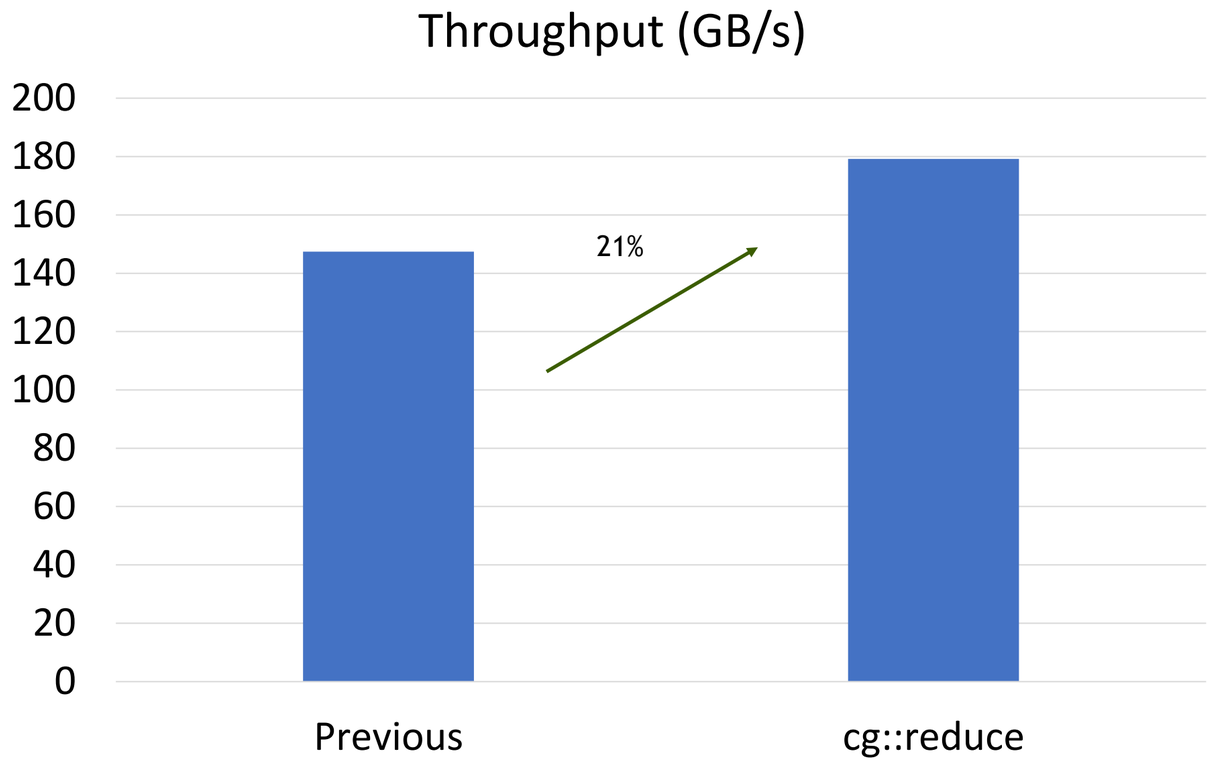
## Data manipulation

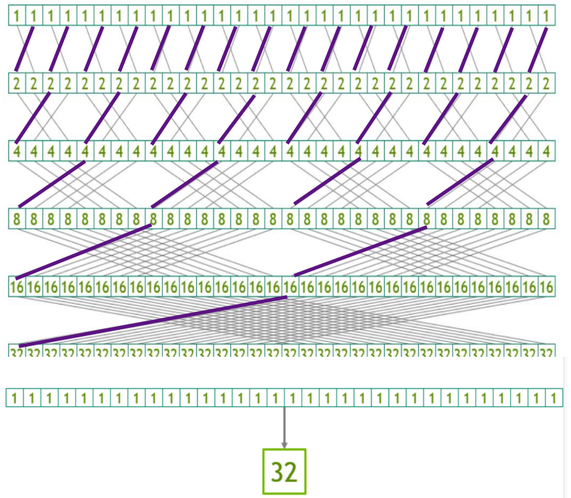
```
/// The following example accepts input in *A and outputs a result into *sum
/// It spreads the data within the block, one element per thread
#define blocksz 256
__global__ void block_reduce(const int *A, int *sum) {
    __shared__ int reduction_s[blocksz];

    cg::thread_block cta = cg::this_thread_block();
    cg::thread_block_tile<32> tile = cg::tiled_partition<32>(cta);

    const int tid = cta.thread_rank();
    int beta = A[tid];
    // reduce across the tile
    reduction_s[tid] = cg::reduce(tile, beta, cg::plus<int>());
    // synchronize the block so all data is ready
    cg::sync(cta);
    // single leader accumulates the result
    if (cta.thread_rank() == 0) {
        beta = 0;
        for (int i = 0; i < blocksz; i += tile.size()) {
            beta += reduction_s[i];
        }
    }
    sum[blockIdx.x] = beta;
}
```

# CUDA COOPERATIVE GROUPS

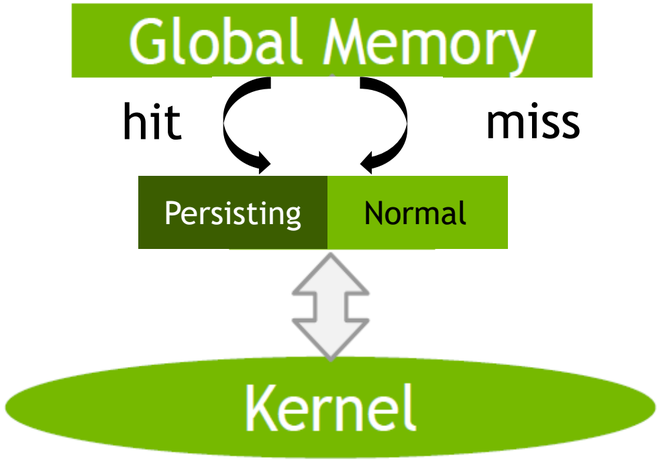
## Data manipulation

Given: Input array size 16MB  Output: Sum of array

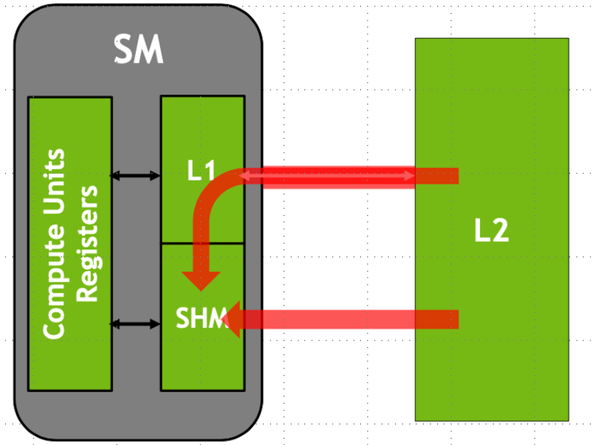




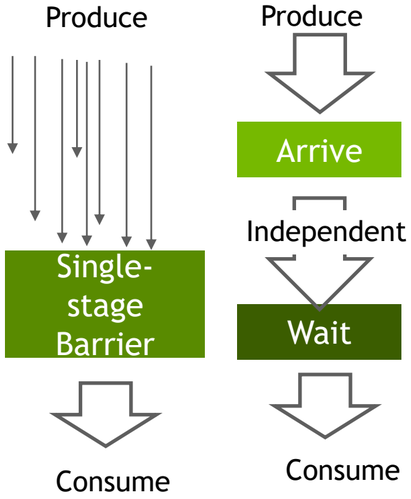

Warp Synchronous Reduction




**L2 cache residency controls**




Asynchronous copy

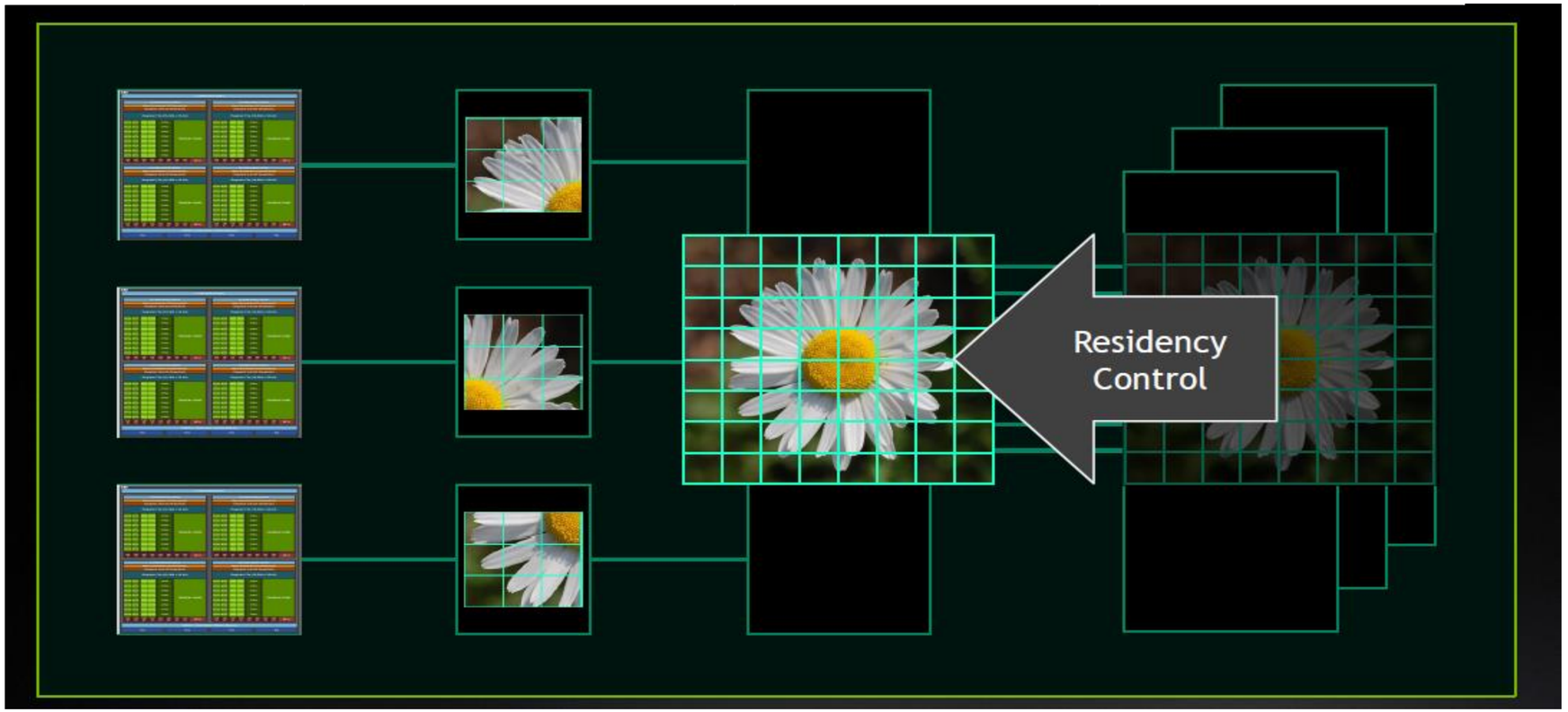



Asynchronous barrier

# L2 RESIDENCY CONTROL

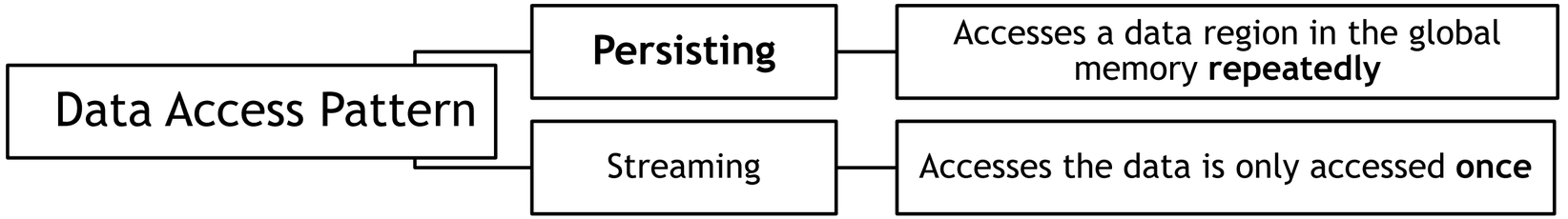
## What's L2 residency control

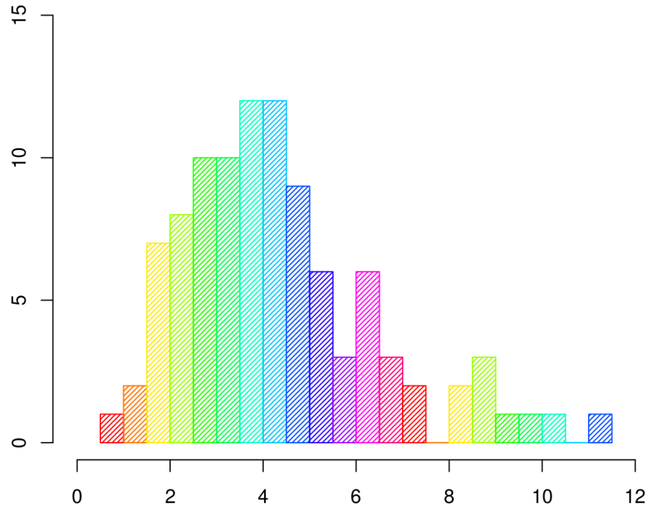
| | Shared Memory | L2 Cache | GPU Memory |
|---|---|---|---|
| Latency | 1x | 5x | 15x |
| Bandwidth | 12x | 3x | 1x |



# L2 RESIDENCY CONTROL

## What's L2 residency control

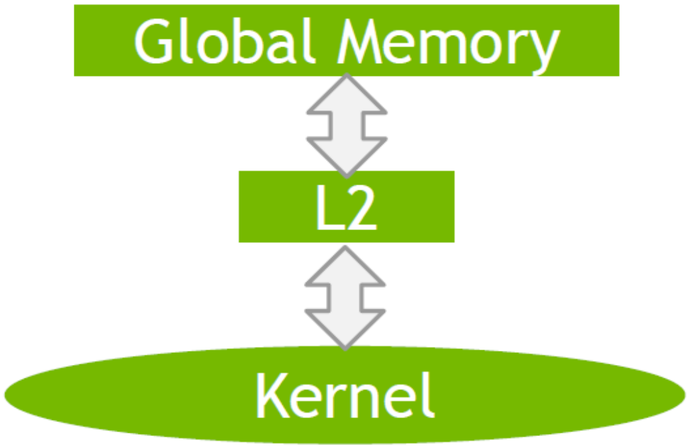


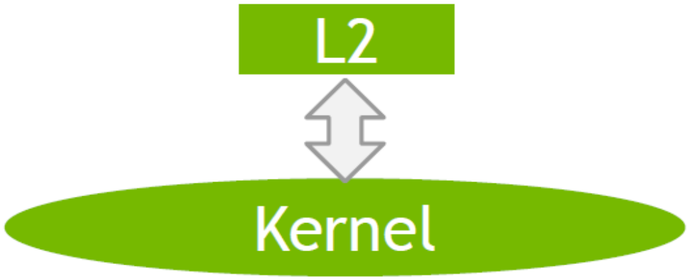

Dataset Size = 1024 MB ( 256 Million integers)
Size of Histogram bins = 20 MB (5 Million integer bins)

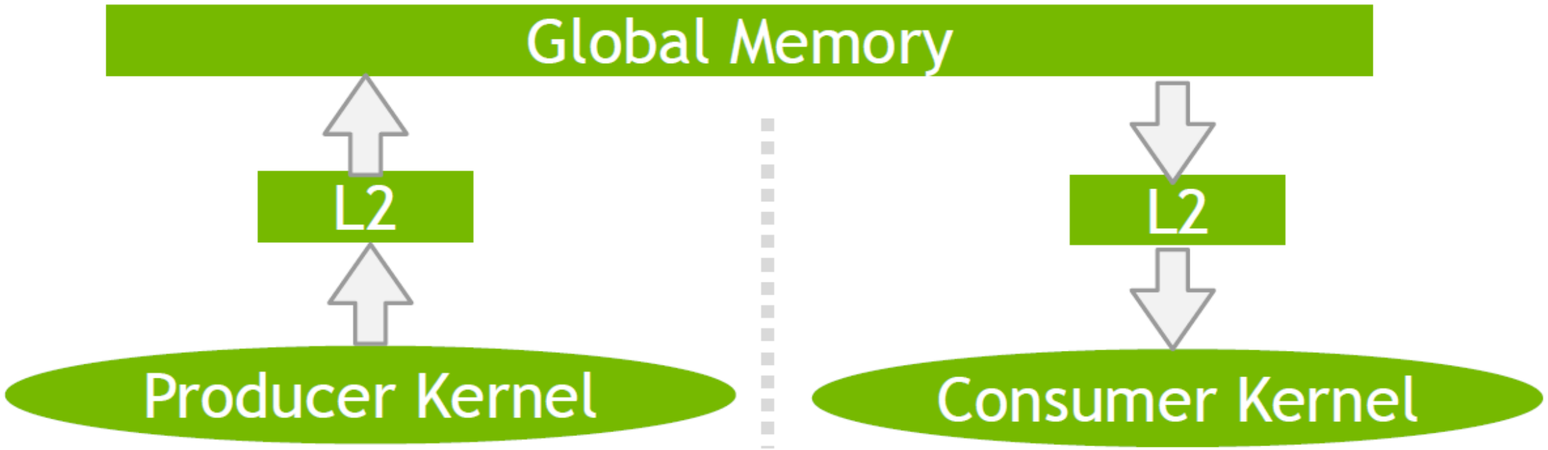
# L2 RESIDENCY CONTROL

Benefits

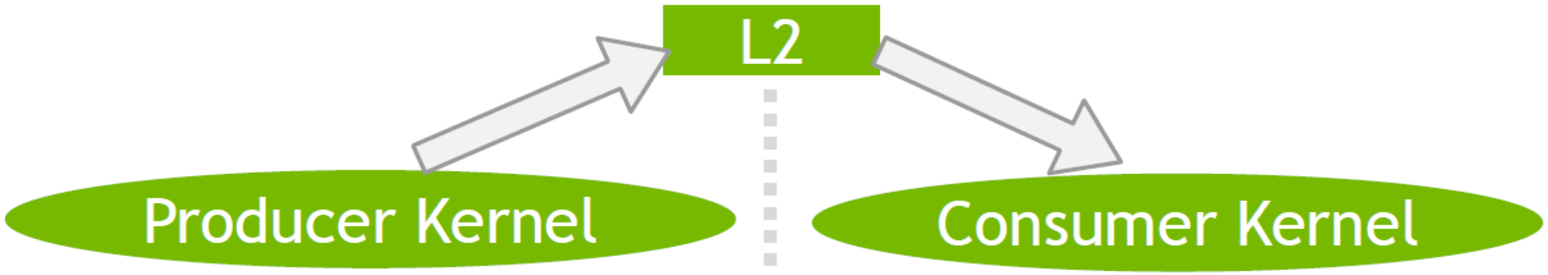



Reduce Intra-Kernel trips to global memory




Reduce Producer-Consumer Inter-Kernel trips to global memory



# RESIDENCY CONTROLS

## Require

- ❑ CUDA 11.0
- ❑ Compute Capability 8.0 and above
- ❑ In MIG mode, the L2 cache set-aside functionality is **disabled**. (deviceProp.persistingL2CacheMaxSize==0)

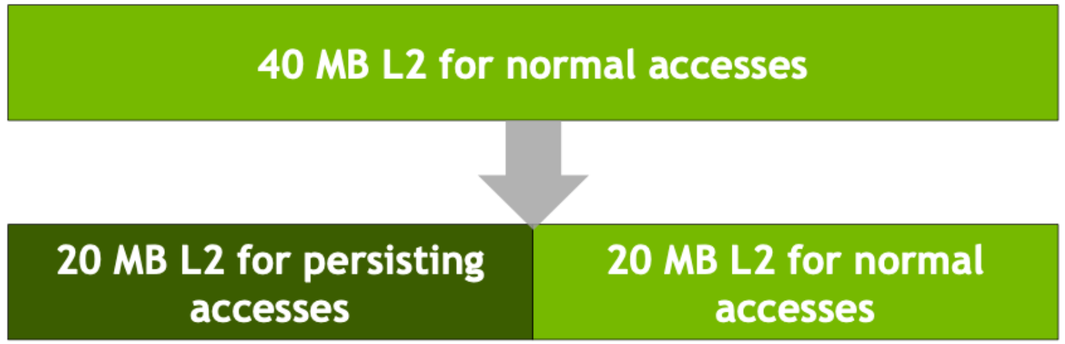
```
// Get device properties
CHECK(cudaGetDeviceProperties(&deviceProp, devID));


// Make sure device the l2 optimization
if (deviceProp.persistingL2CacheMaxSize == 0) {
  printf("Waiving execution as device %d does not support persisting L2 "
         "Caching\n",
         devID);
  exit(0);
}
```

# RESIDENCY CONTROLS

## L2 cache set-aside functionality

- A part of L2 cache to be set-aside for **persistent data accesses.**
- For A100, deviceProp.persistingL2CacheMaxSize=30MB
- Using MPS, set CUDA_DEVICE_DEFAULT_PERSISTING_L2_CACHE_PERCENTAGE_LIMIT at start up of MPS server

```
cudaDeviceSetLimit(cudaLimitPersistingL2CacheSize, size);
```



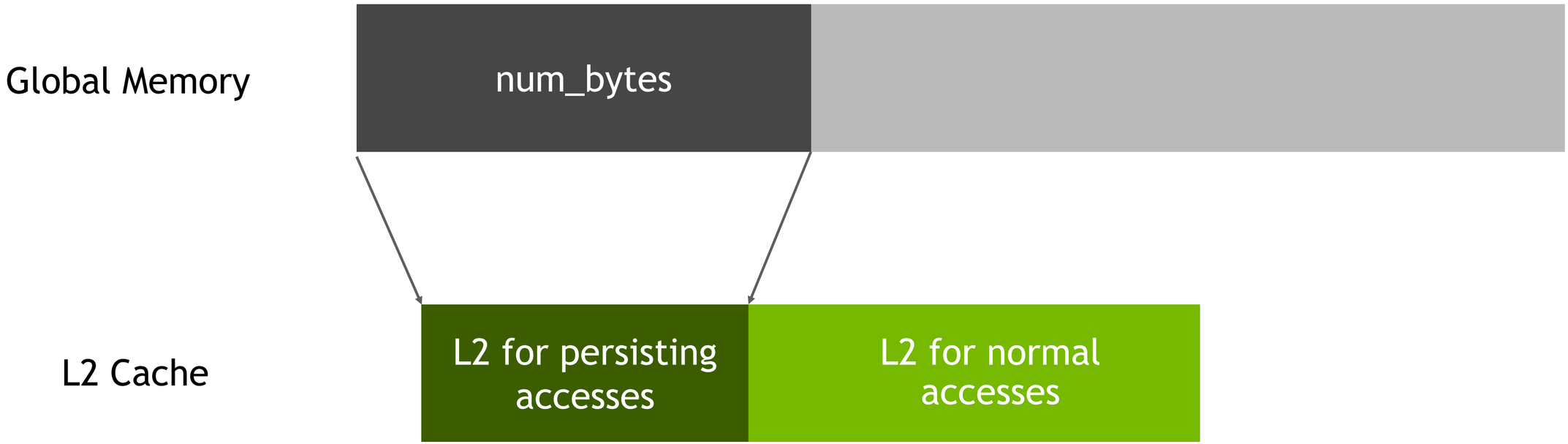

Persistent accesses has **higher residence priority** in L2 cache over other data accesses.

Normal accesses **can use** the set-aside region of L2 when persisting accesses are not using it.

# RESIDENCY CONTROLS

## Access Policy Window

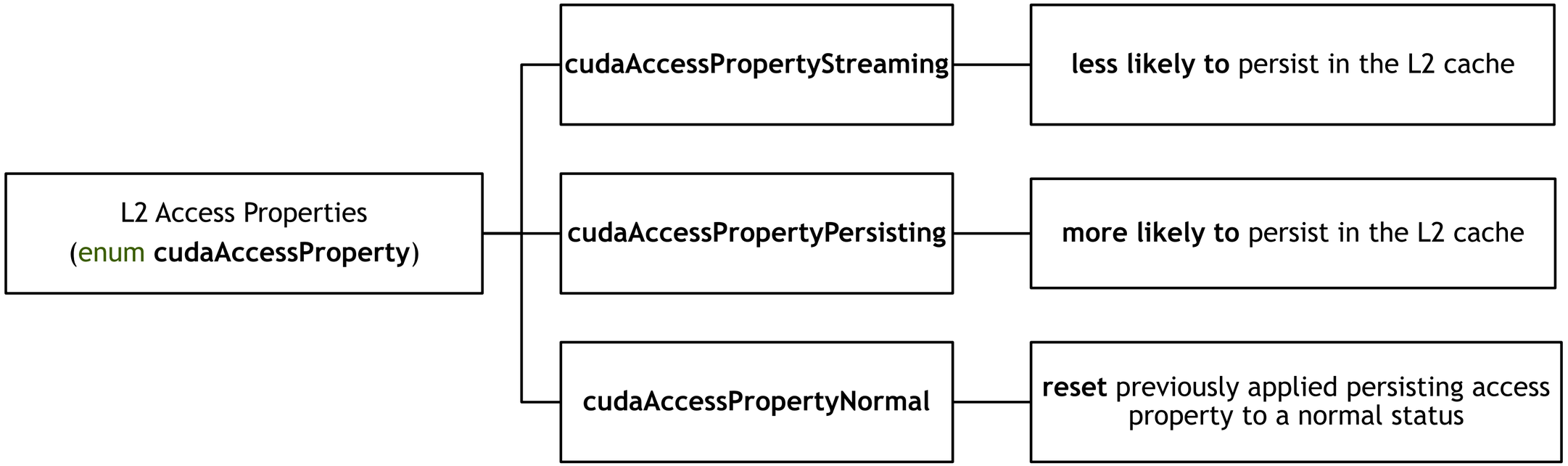
❑ Global memory region can be marked for persistence access using **accessPolicyWindow**



# RESIDENCY CONTROLS

## L2 Policy for Persisting Accesses

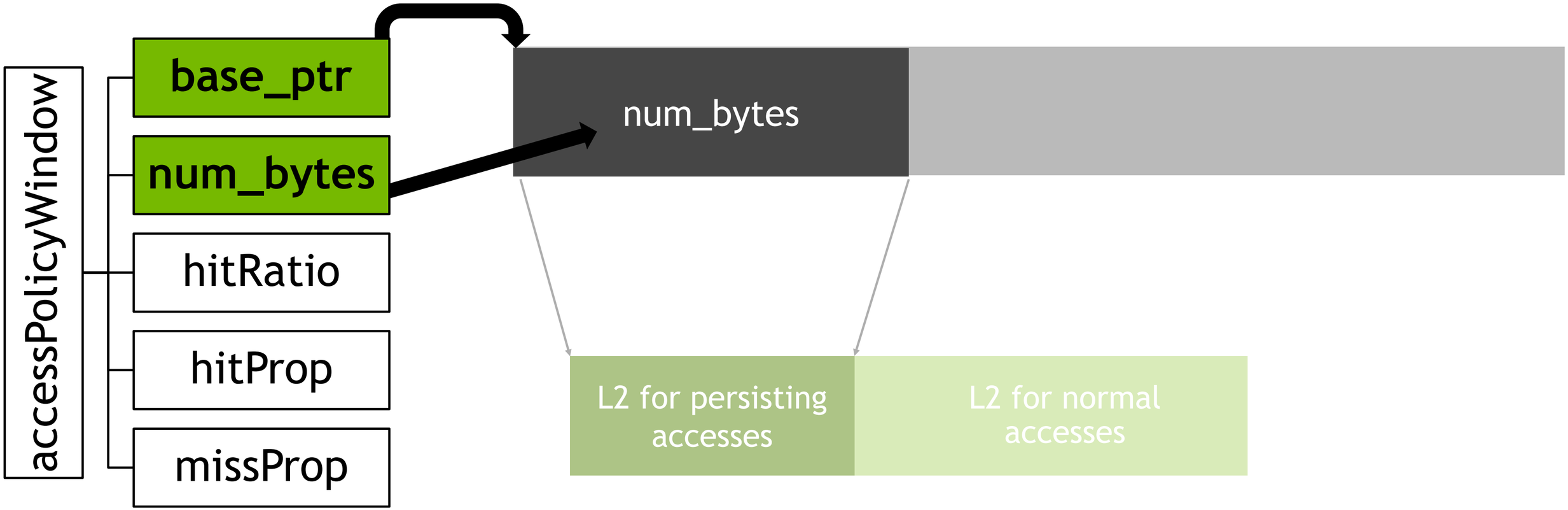
❑ An access policy window specifies **a contiguous region of global memory** and **a persistence property in the L2 cache** for accesses within that region.



# RESIDENCY CONTROLS

## Access Policy Window

The maximum value of ***num_bytes*** is **deviceProp.accessPolicyMaxWindowSize=128MB (A100)**

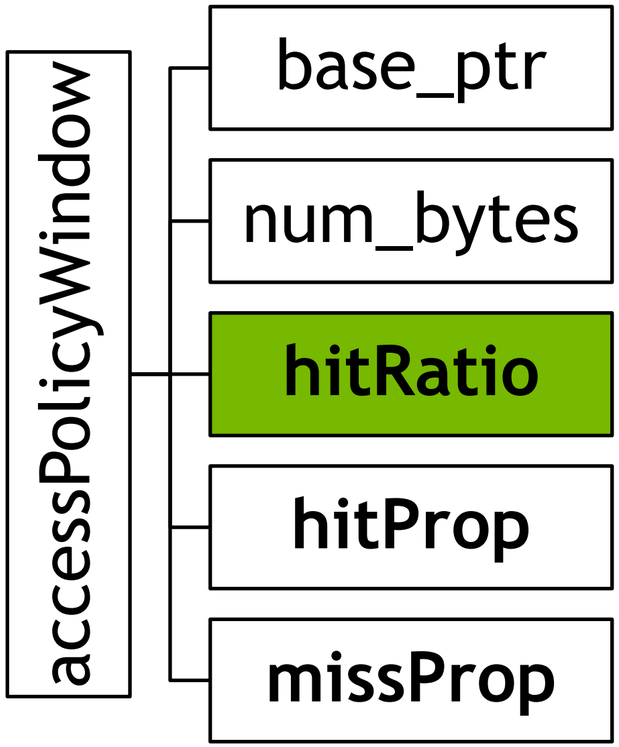


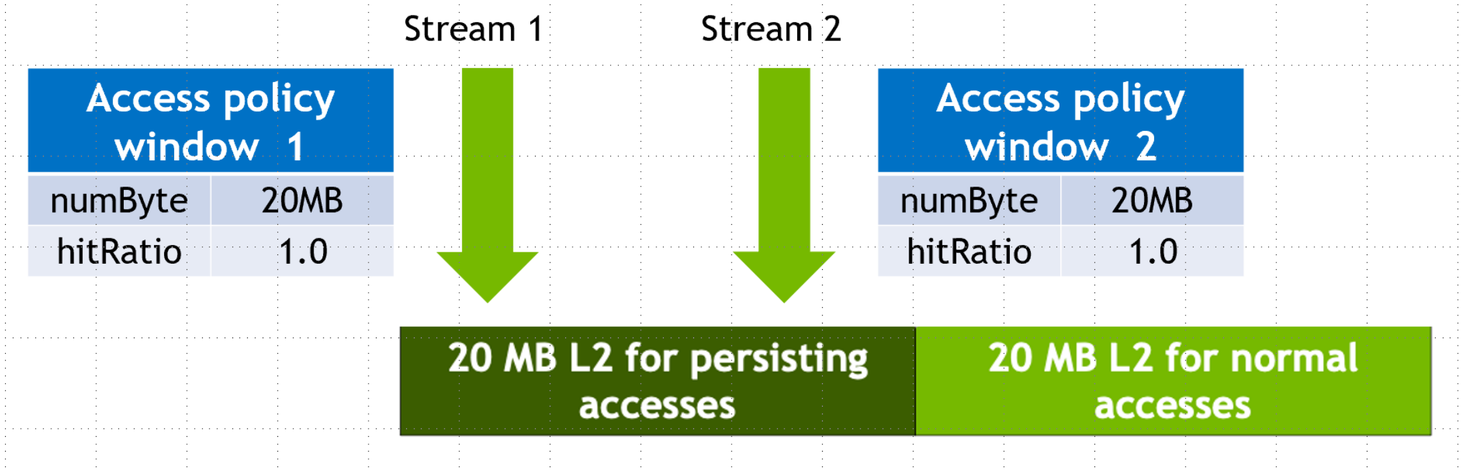

https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaAccessPolicyWindow.html#structcudaAccessPolicyWindow_1bbd68021cb5c3cb1928bee0a2941eb24

# RESIDENCY CONTROLS

## Access Policy Window

- Note: num_bytes can be **larger** than the size of L2 set-aside cache
- Cache lines will be **evicted** to keep the **most recently** used data in the set-aside portion of the L2 cache.



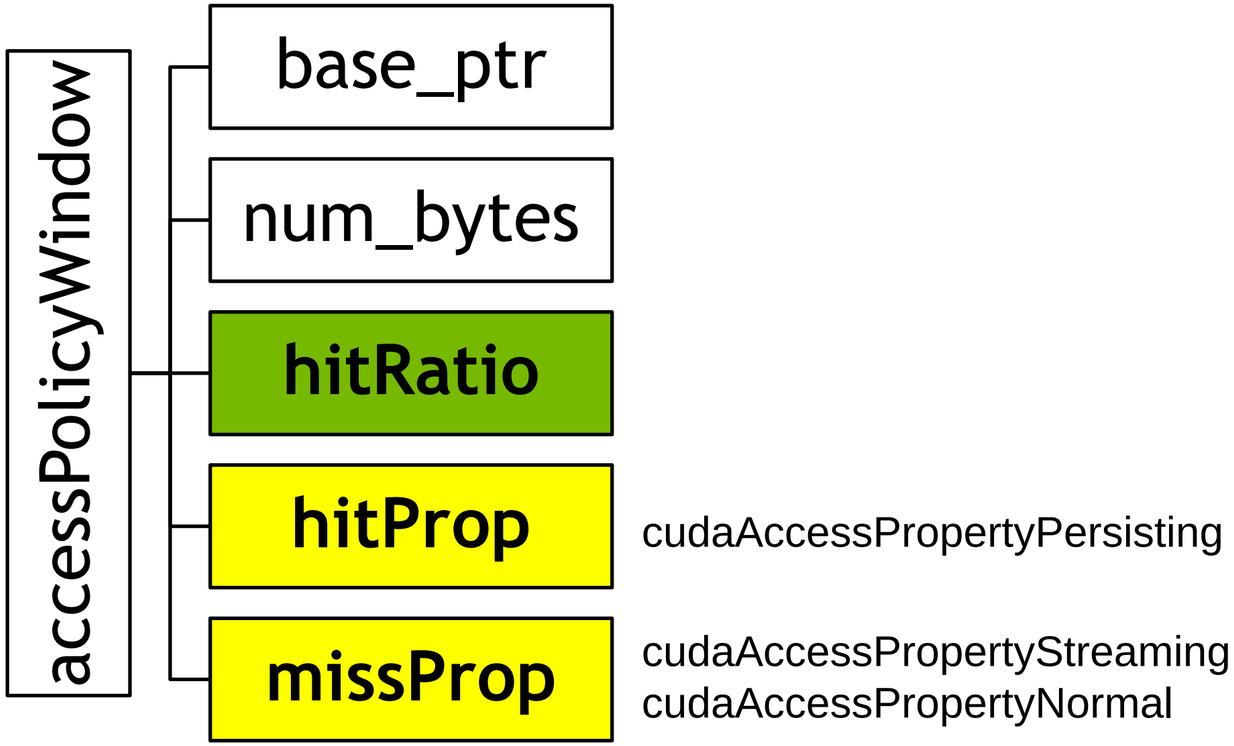

Thrashing of cache lines

https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaAccessPolicyWindow.html#structcudaAccessPolicyWindow_1bbd68021cb5c3cb1928bee0a2941eb24

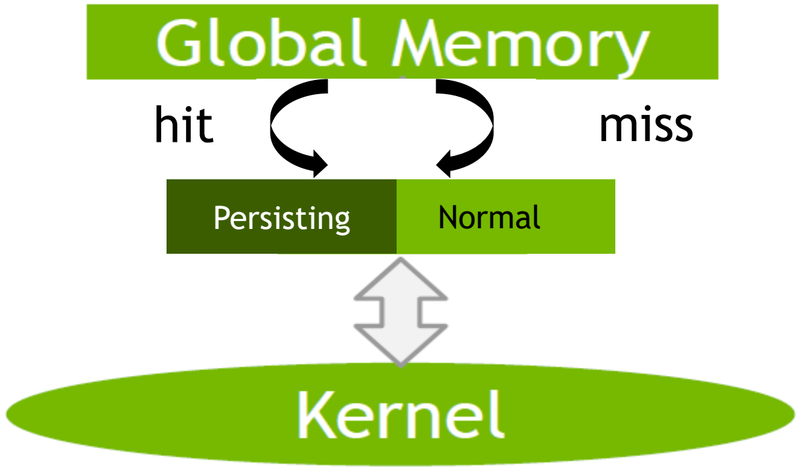
# RESIDENCY CONTROLS

## Access Policy Window

- ***hitRatio*** specifies **percentage of lines** assigned ***hitProp***, rest are assigned ***missProp***.
  Specific memory accesses are classified as persisting (the **hitProp**) is **random**



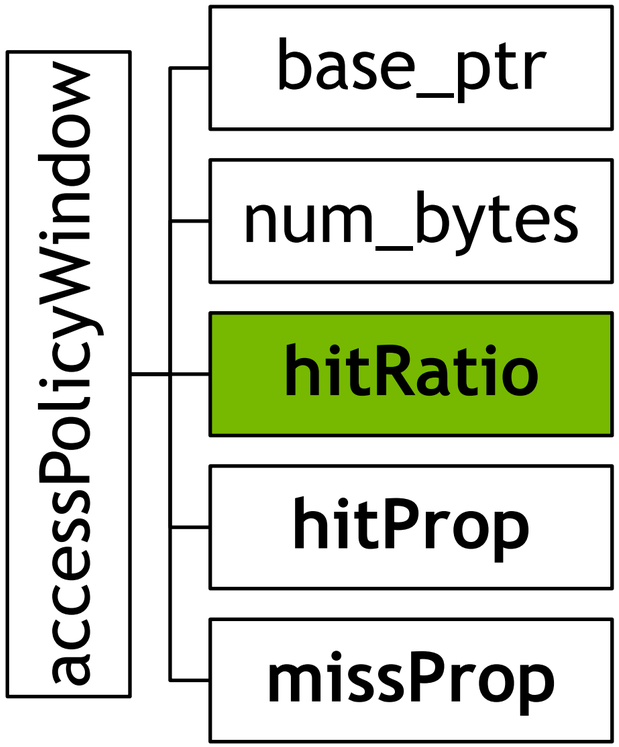

**Example:**
num_bytes = 100MB,hitRatio = 0.6
Memory size with hitProp =100*0.6 = 60MB
Memory size using missProp100*0.4 = 40MB

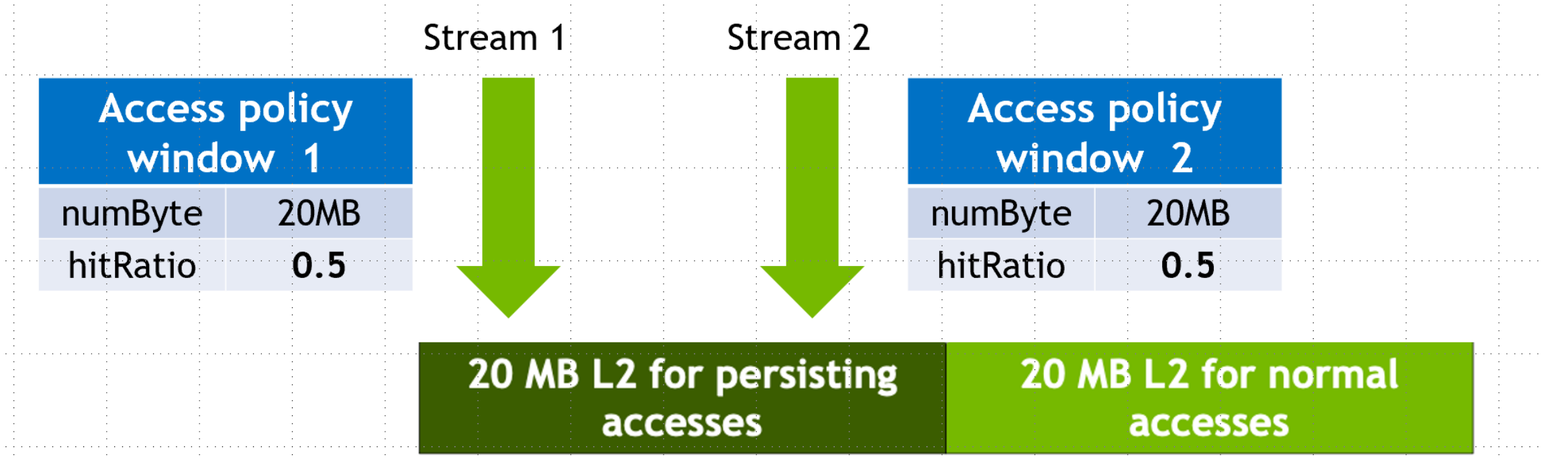


# RESIDENCY CONTROLS

## Access Policy Window

- ***hitRatio*** specifies **percentage of lines** assigned ***hitProp***, rest are assigned ***missProp***.
  Specific memory accesses are classified as persisting (the **hitProp**) is **random**



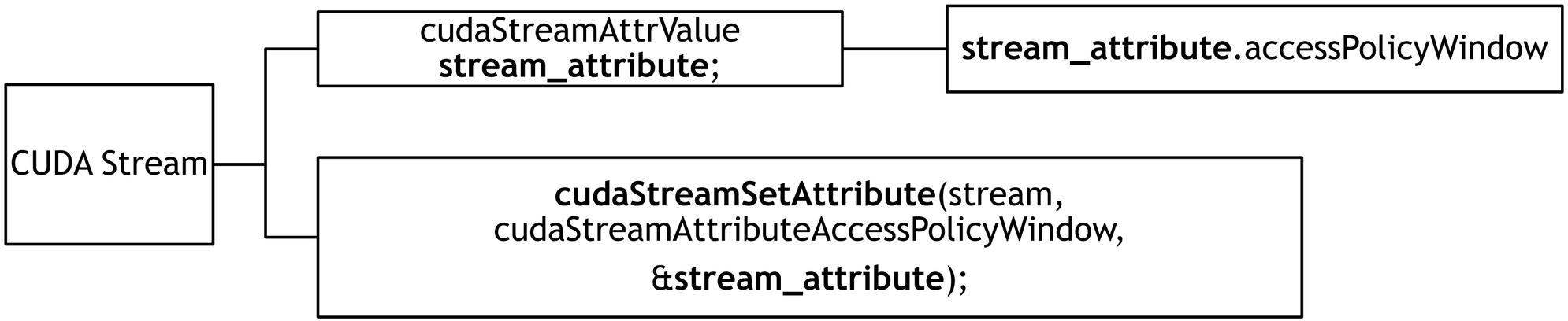

**Less likely** to evict their own or each others' persisting cache lines

https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaAccessPolicyWindow.html#structcudaAccessPolicyWindow_1bbd68021cb5c3cb1928bee0a2941eb24

# RESIDENCY CONTROLS

## Set an L2 persisting access window

- Set an L2 persisting access window using a CUDA Stream.



# RESIDENCY CONTROLS

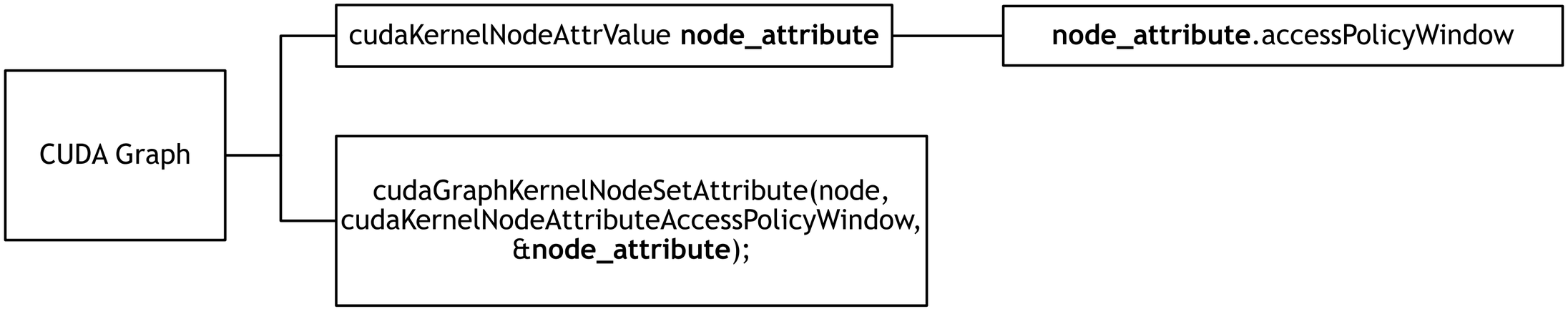
## Set an L2 persisting access window

```
// Stream level attributes data structure
cudaStreamAttrValue stream_attribute;

// Global Memory data pointer
stream_attribute.accessPolicyWindow.base_ptr = reinterpret_cast<void*>(ptr);

//Number of bytes for persistence access.
//(Must be less than cudaDeviceProp::accessPolicyMaxWindowSize)
stream_attribute.accessPolicyWindow.num_bytes = num_bytes;

//Hint for cache hit ratio
 stream_attribute.accessPolicyWindow.hitRatio = 0.6;

//Type of access property on cache hit
stream_attribute.accessPolicyWindow.hitProp = cudaAccessPropertyPersisting;
//Type of access property on cache miss.
stream_attribute.accessPolicyWindow.missProp = cudaAccessPropertyStreaming;

//Set the attributes to a CUDA stream of type cudaStream_t
cudaStreamSetAttribute(stream, cudaStreamAttributeAccessPolicyWindow, &stream_attribute);
```

# RESIDENCY CONTROLS

## Set an L2 persisting access window

- Set an L2 persisting access window using a CUDA Graph.



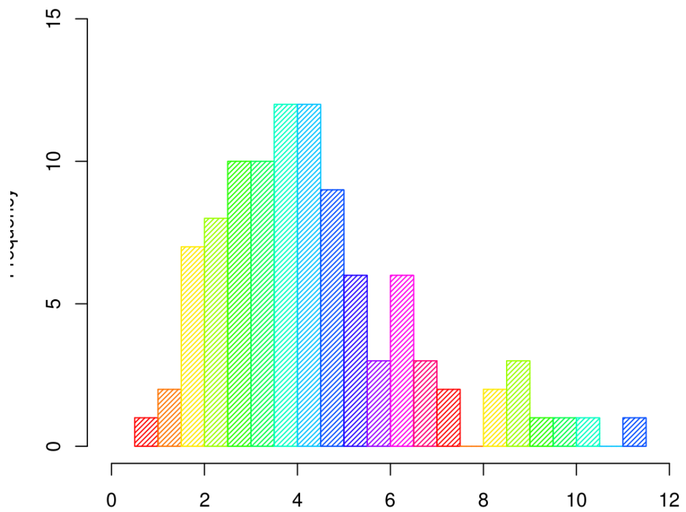
# RESIDENCY CONTROLS

## Set an L2 persisting access window

```
// Kernel level attributes data structure
cudaKernelNodeAttrValue node_attribute;

// Global Memory data pointer
node_attribute.accessPolicyWindow.base_ptr = reinterpret_cast<void*>(ptr);

//Number of bytes for persistence access.
//(Must be less than cudaDeviceProp::accessPolicyMaxWindowSize)
node_attribute.accessPolicyWindow.num_bytes = num_bytes;

//Hint for cache hit ratio
node_attribute.accessPolicyWindow.hitRatio = 0.6;

//Type of access property on cache hit
node_attribute.accessPolicyWindow.hitProp = cudaAccessPropertyPersisting;
//Type of access property on cache miss.
node_attribute.accessPolicyWindow.missProp = cudaAccessPropertyStreaming;

//Set the attributes to a CUDA Graph Kernel node of type cudaGraphNode_t
cudaGraphKernelNodeSetAttribute(node, cudaKernelNodeAttributeAccessPolicyWindow, &node_attribute);
```

# EXAMPLE: HISTOGRAM

Dataset Size = 1024 MB ( 256 Million integers)
Size of Histogram bins = 20 MB (5 Million integer bins)




```
static __global__ void
histogram(int* data, int dataSize, int *hist, int nbins)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int bin_id= data[idx]%nbins;
    atomicAdd(&hist[bin_id], 1);
}
```

# EXAMPLE: HISTOGRAM

**Steps**

**Set the site-aside L2 Cache**

Set an L2 persisting access window

Run kernels

Reset the access policy window

```
int devID=0;
checkCudaErrors(cudaSetDevice(devID));
cudaDeviceProp deviceProp;
checkCudaErrors(cudaGetDeviceProperties(&deviceProp, devID));
```

```
checkCudaErrors(cudaDeviceSetLimit(cudaLimitPersistingL2CacheSize, deviceProp.persistingL2CacheMaxSize));
```

# EXAMPLE: HISTOGRAM

**Steps**

**Set the site-aside L2 Cache**

Set an L2 persisting access window

Run kernels

Reset the access policy window

```
//Make a window for the buffer of interest
streamAttrValue.accessPolicyWindow      = initAccessPolicyWindow();
accessPolicyWindow.base_ptr = (void *)d_hist;
accessPolicyWindow.num_bytes = hist_size * sizeof(int);
accessPolicyWindow.hitRatio = 1.f;
accessPolicyWindow.hitProp = cudaAccessPropertyPersisting;
accessPolicyWindow.missProp = cudaAccessPropertyNormal;
streamAttrValue.accessPolicyWindow = accessPolicyWindow;
```

```
//Assign window to stream
checkCudaErrors(cudaStreamSetAttribute(stream, cudaStreamAttributeAccessPolicyWindow, &streamAttrValue));
```

# EXAMPLE: HISTOGRAM

**Steps**

**Set the site-aside L2 Cache**

Set an L2 persisting access window

Run kernels

Reset the access policy window

```
static __global__ void
histogram(int* data, int dataSize, int *hist, int nbins)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int bin_id= data[idx]%nbins;
    atomicAdd(&hist[bin_id], 1);
}
```

```
histogram<<<blocks, threads, 0, stream>>>(d_data, data_size, d_hist, hist_size);
```

# EXAMPLE: HISTOGRAM

**Steps**

**Set the site-aside L2 Cache**

Set an L2 persisting access window

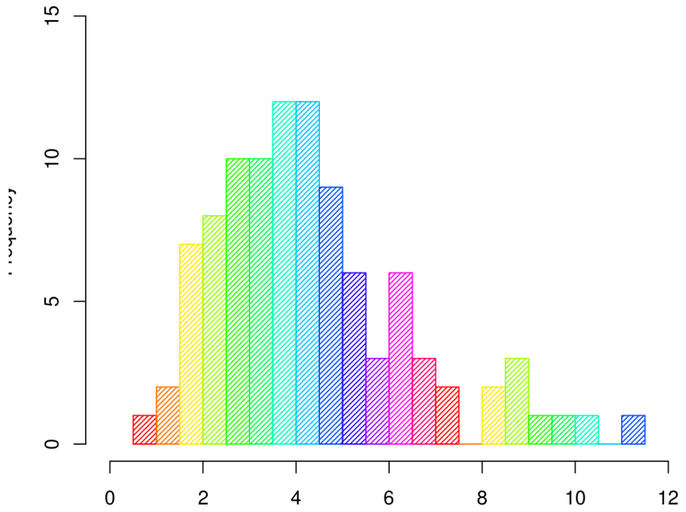
Run kernels

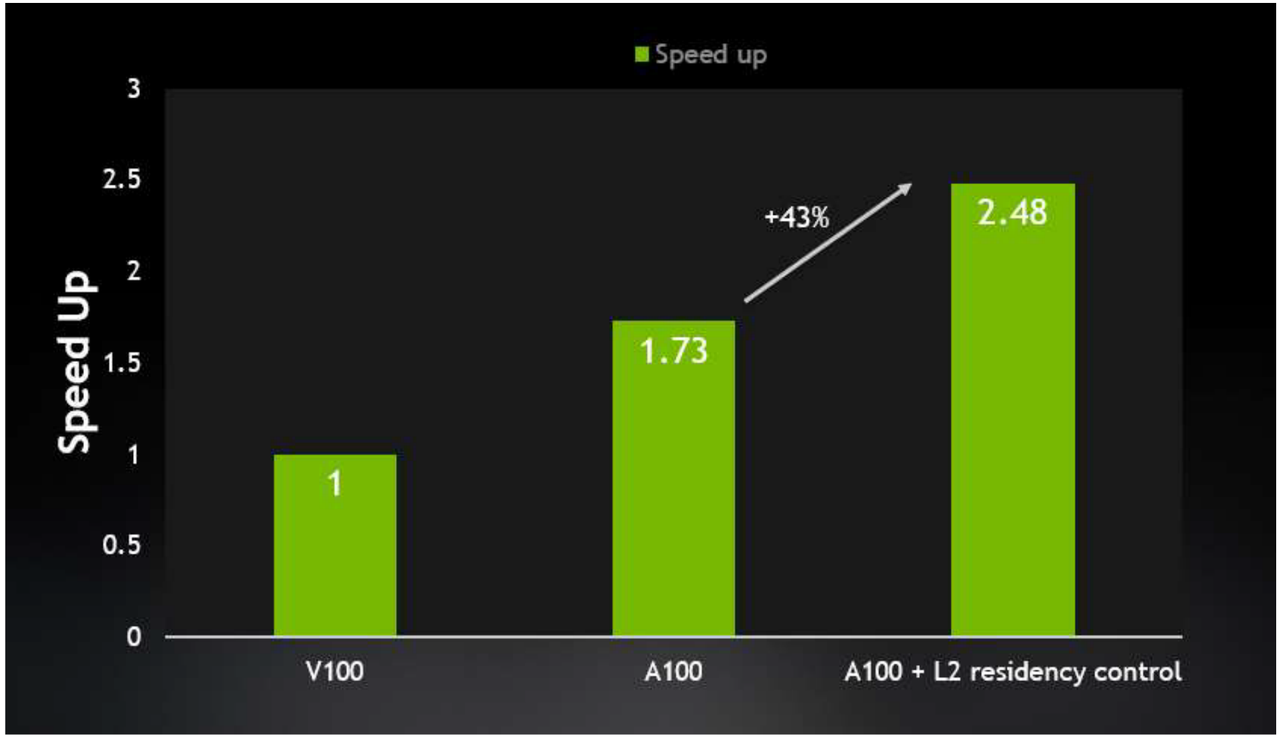
Reset the access policy window

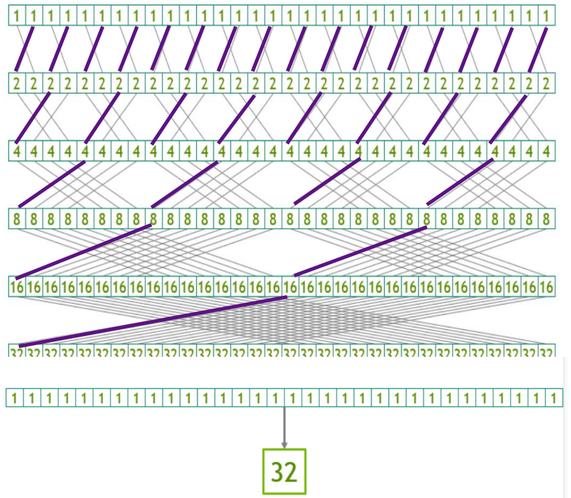
```
// Setting the window size to 0 disable it
stream_attribute.accessPolicyWindow.num_bytes = 0;
```

```
// Overwrite the access policy attribute to a CUDA Stream
cudaStreamSetAttribute(stream,
     cudaStreamAttributeAccessPolicyWindow, &stream_attribute);

// Remove any persistent lines in L2
cudaCtxResetPersistingL2Cache();
```

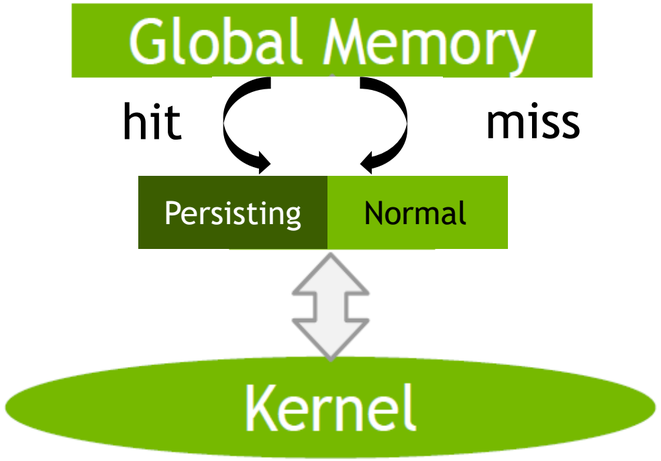
# EXAMPLE: HISTOGRAM



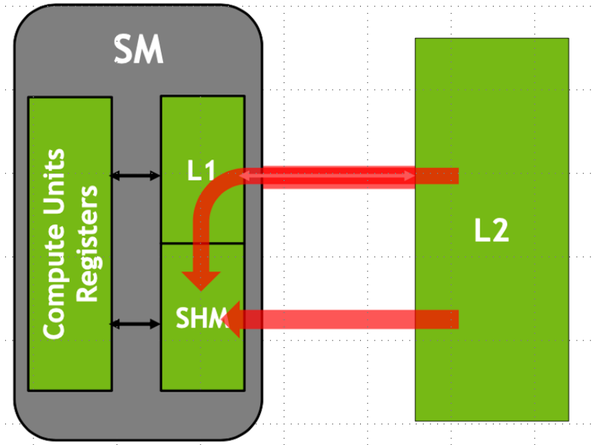





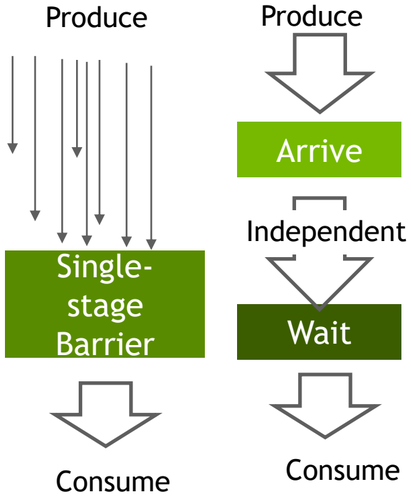

Warp Synchronous Reduction




L2 cache residency controls




**Asynchronous copy**

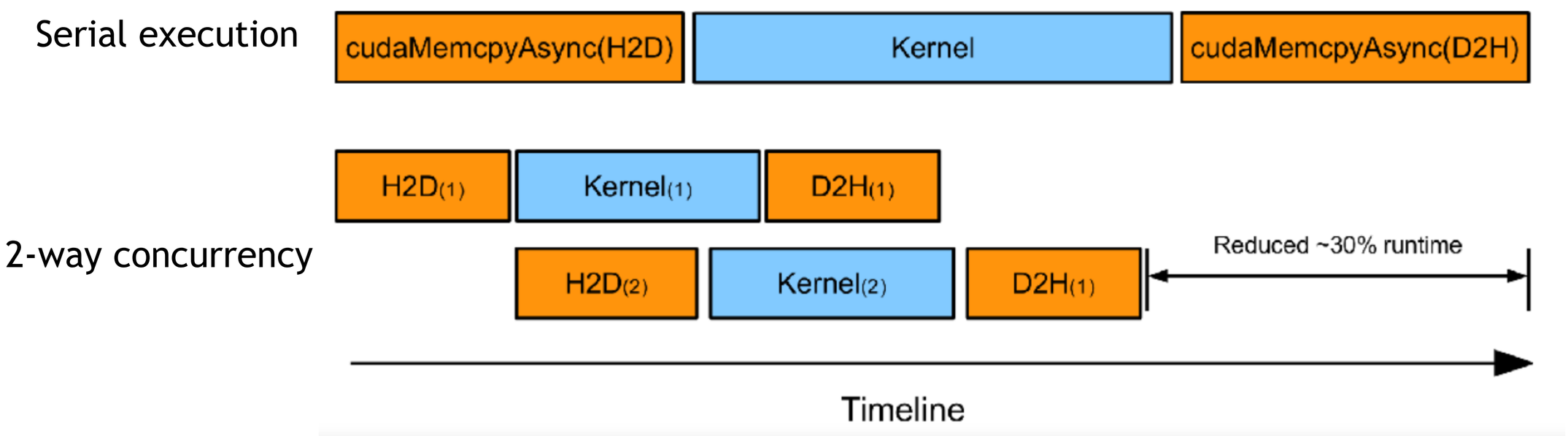



Asynchronous barrier

# ASYNCHRONOUS COPY

## What's async-copy

- Previously: data movement between **CPU and GPU global memory** overlapped with **kernel execution**
  Using **cudaMemcpyAsync**




Yu, Leiming. "Multilevel Interference-aware Scheduling On Modern Gpus." (2019).

# ASYNCHRONOUS COPY

## What's async-copy

- Overlap **copying** data **from global to shared memory** with **computation**
  Using **cuda::memcpy_async**

## __shared__ Memory

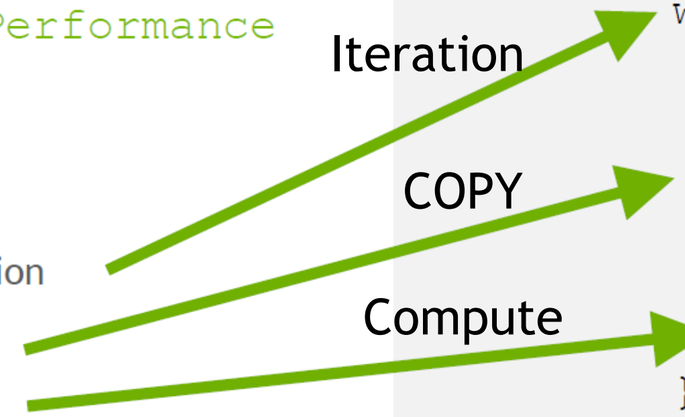
Many Algorithms' Key for Performance

Current use of shared memory

- Time-stepping and global data iteration
- Copy global data to shared memory
- Compute on shared memory

Copy and Compute Phases are Sequenced

Global → Shared Memory has a Journey

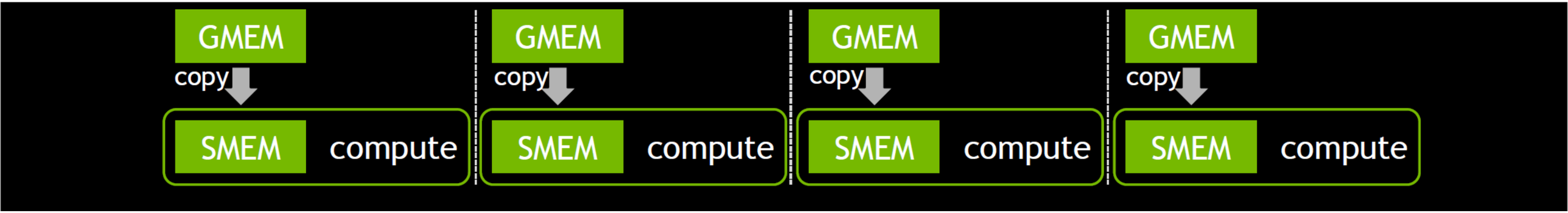



```
__shared__ extern int shbuf[];
while ( an_algorithm_iterates ) {
  __syncthreads();
  for ( i = ... ) {
    shbuf[i] = gldata[i]; /* copy */
  }
  __syncthreads();
  /* compute on shbuf[] */
}
```

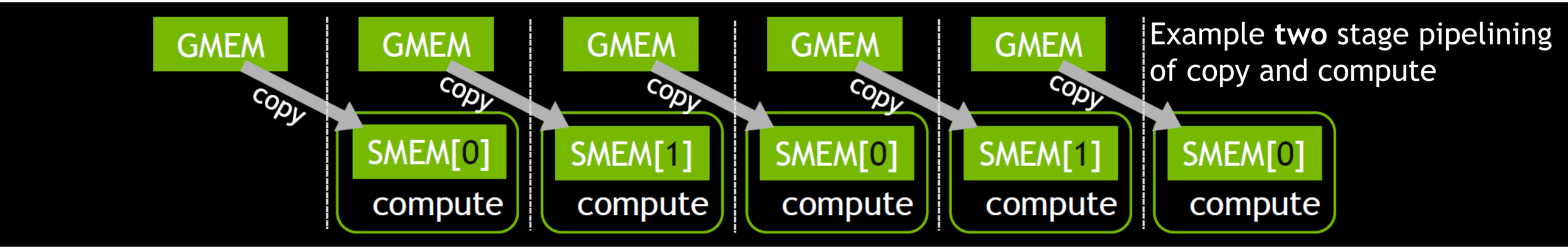
# ASYNCHRONOUS COPY

## What's async-copy

Each iteration: First copy GMEM -> SMEM; then Compute on SMEM




Better to Compute while **Copying** for later iteration(s)



# ASYNCHRONOUS COPY

## Before cuda::memcpy_aysnc

# THE JOURNEY

## Global Memory -> Share Memory

Typical way of using shared memory:

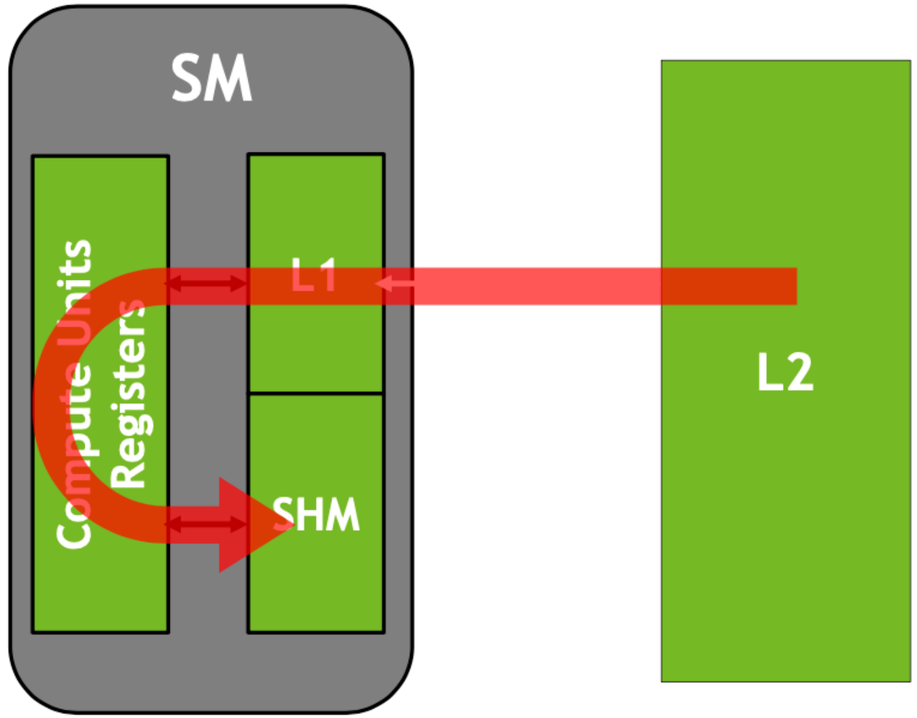
```
__shared__ int smem[1024];
smem[threadIdx.x] = input[index];
```

```
LDG.E.SYS R0, [R2] ;
   * STALL *
STS [R5], R0 ;
```

Wasting registers

Stalling while the data is loaded

Wasting L1/SHM bandwidth




# ASYNCHRONOUS COPY

With cuda::memcpy_async

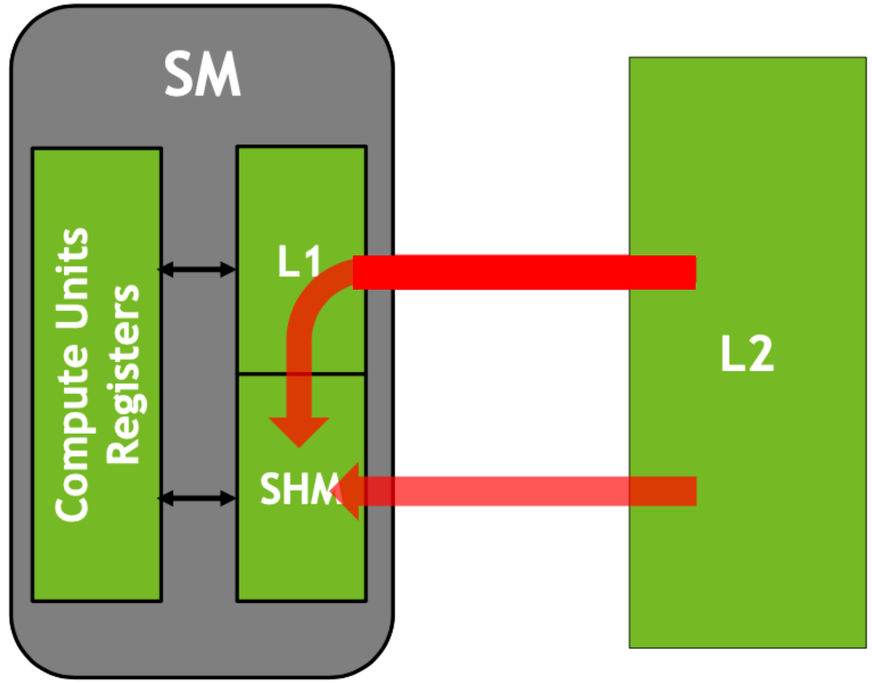
## ASYNC-COPY

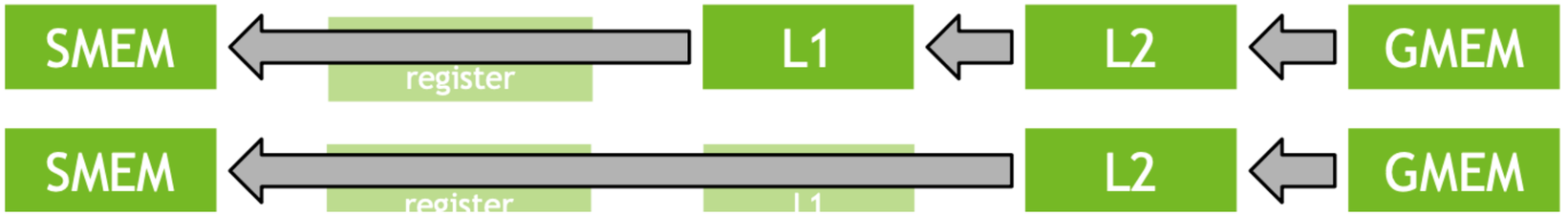
### Global Memory -> Share Memory

Copies the data straight to shared memory asynchronously with 2 possible paths:

- L1 Access (Data gets Cached in L1)
- L1 Bypass (No L1 Caching, 16-Byte vector LDGSTS)

Very flexible scheduling (e.g. multi-stage)




Data Size & alignment: 4B

Data Size & alignment: 16B

# ASYNCHRONOUS COPY

## Begin with One Stage Pipeline

- Dispatch: The **memcpy_async** dispatches an async-copy operation that will *eventually* read global data **gdata[i]** and write shared data **sdata[i]**.

| Description | Function | Note |
|---|---|---|
| Async-Copy | template<class T><br>void memcpy_async(T& dst, const T& src, pipeline& pipe); | sizeof(T) % 4 == 0<br>dst/src in share/global mem. |

Performance of memcpy_async is best when sizeof(T) == 16

# ASYNCHRONOUS COPY

## Begin with One Stage Pipeline

- Still have the __syncthreads() sandwitch

```
__shared__ extern int shbuf[];

while ( an_algorithm_iterates ) {
  __syncthreads();
  for ( i = ... ) {
    shbuf[i] = gldata[i];
  }

  __syncthreads();
  /* compute on shbuf[] */
}
```

```
__shared__ extern int shbuf[];
pipeline pipe;
while ( an_algorithm_iterates ) {
  __syncthreads();
  for ( i = ... ) {
    memcpy_async(shbuf[i],gldata[i],pipe);
  }
  pipe.commit_and_wait();
  __syncthreads();
  /* compute on shbuf[] */
}
```

# ASYNCHRONOUS COPY

## Begin with One Stage Pipeline

CUDA 11.0

```
#include <cuda_pipeline.h>
using namespace nvcuda::experimental;

    extern __shared__ T shared[];

    pipeline pipe;
    for (size_t i = 0; i < copy_count; ++i){
        memcpy_async(shared[blockDim.x * i + threadIdx.x],global[blockDim.x * i + threadIdx.x], pipe);
    }
    pipe.commit();
    pipe.wait_prior<0>();
```

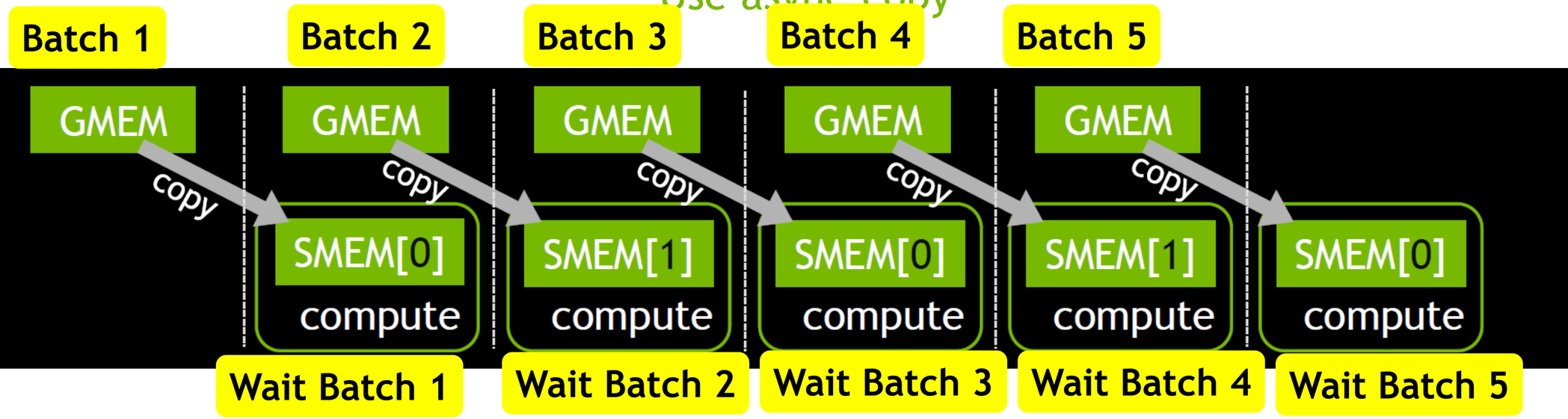
CUDA 11.1

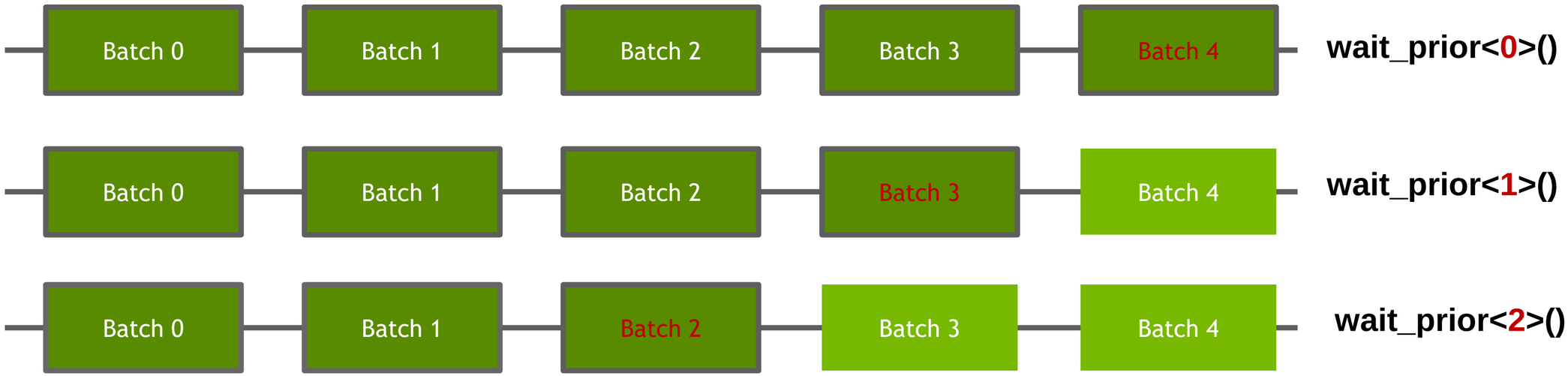
```
#include <cooperative_groups.h>
#include <cooperative_groups/memcpy_async.h>

   extern __shared__ T shared[];

   auto group = cooperative_groups::this_thread_block();
    for (size_t i = 0; i < copy_count; ++i){
        cooperative_groups::memcpy_async(group,s_shared,global,sizeof(float) * group.size());
    }
    cooperative_groups::wait(group); // Wait for all copies to complete
```

https://developer.nvidia.com/blog/controlling-data-movement-to-boost-performance-on-ampere-architecture/

# ASYNCHRONOUS COPY

Use async-copy



# ASYNCHRONOUS COPY

## Use async-copy

- A sequence of asynchronous copy batches and computations pipelined in two stages. (CUDA 11.1)

```
#include <cooperative_groups.h>
#include <cooperative_groups/memcpy_async.h>

   auto group = cooperative_groups::this_thread_block();
   cooperative_groups::memcpy_async(group,shared[stage], elem_shared, global,elem_global);

    for (size_t i = 0; i < copy_count; ++i){
        cooperative_groups::memcpy_async(group,shared[stage^1],elem_shared, global, elem_global);
        cg::wait_prior<1>(group);//we wait on the one before it
        /***Compute***/
        stage ^=1;
    }
    cooperative_groups::wait(group); // Wait for all copies to complete
```

# EXAMPLE: ASYNC-COPY

Use one block with 128 threads.
Use elements of size 4B, 8B and 16B per thread (int, int2 and int4)
each thread block copies from 512 bytes up to 48 KB

```
template <typename T>
__global__ void pipeline_kernel_sync(T *global, uint64_t *clock, size_t copy_count)
 {
  extern __shared__ char s[];
  T *shared = reinterpret_cast<T *>(s);

  uint64_t clock_start = clock64();

  for (size_t i = 0; i < copy_count; ++i) {
    shared[blockDim.x * i + threadIdx.x] = global[blockDim.x * i + threadIdx.x];
  }

  uint64_t clock_end = clock64();

  atomicAdd(reinterpret_cast<unsigned long long *>(clock),
            clock_end - clock_start);
}
```

T int/int2/int4

copy_count

1 to 49152 (int)
1 to 24576(int2)
1 to 12288 (int4)

# EXAMPLE: ASYNC-COPY

Use one block with 128 threads.
Use elements of size 4B, 8B and 16B per thread (int, int2 and int4)
each thread block copies from 512 bytes up to 48 KB

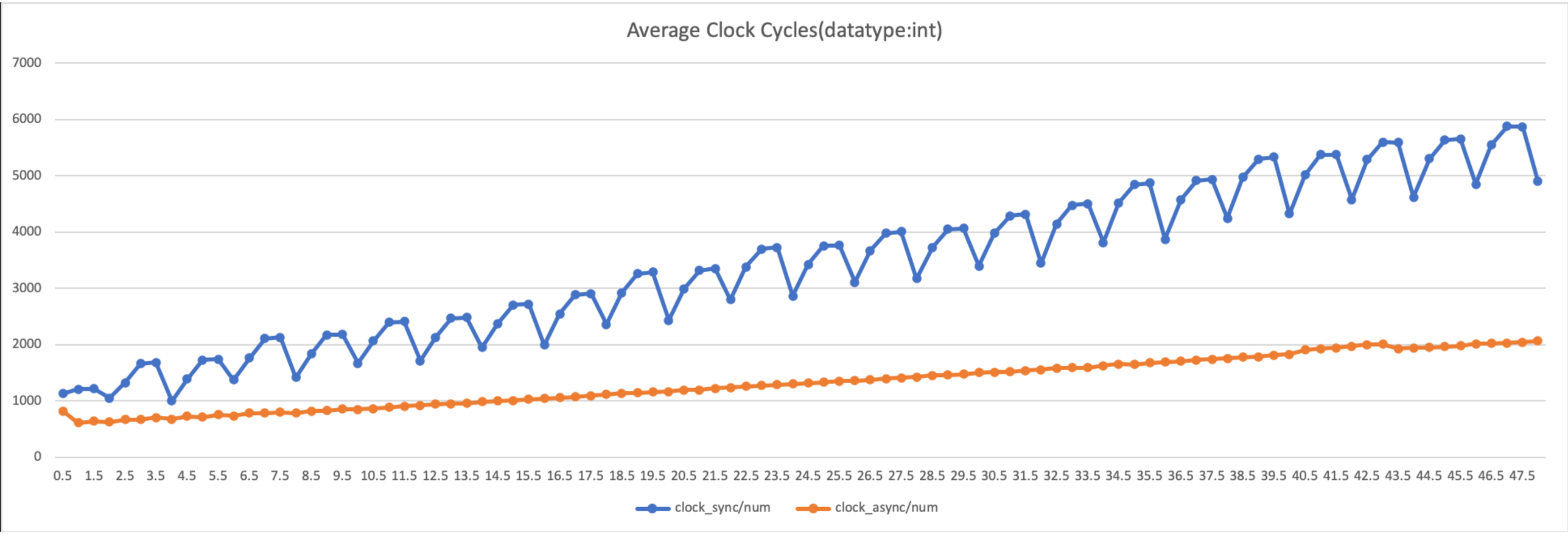
T int/int2/int4

```
template <typename T>
__global__ void pipeline_kernel_async(T *global, uint64_t *clock, size_t copy_count)
 {
  extern __shared__ char s[];
  T *shared = reinterpret_cast<T *>(s);

  uint64_t clock_start = clock64();

  pipeline pipe;
  for (size_t i = 0; i < copy_count; ++i) {
    memcpy_async(shared[blockDim.x * i + threadIdx.x],
                 global[blockDim.x * i + threadIdx.x], pipe);
  }
  pipe.commit();
  pipe.wait_prior<0>();

  uint64_t clock_end = clock64();

  atomicAdd(reinterpret_cast<unsigned long long *>(clock),
            clock_end - clock_start);
}
```

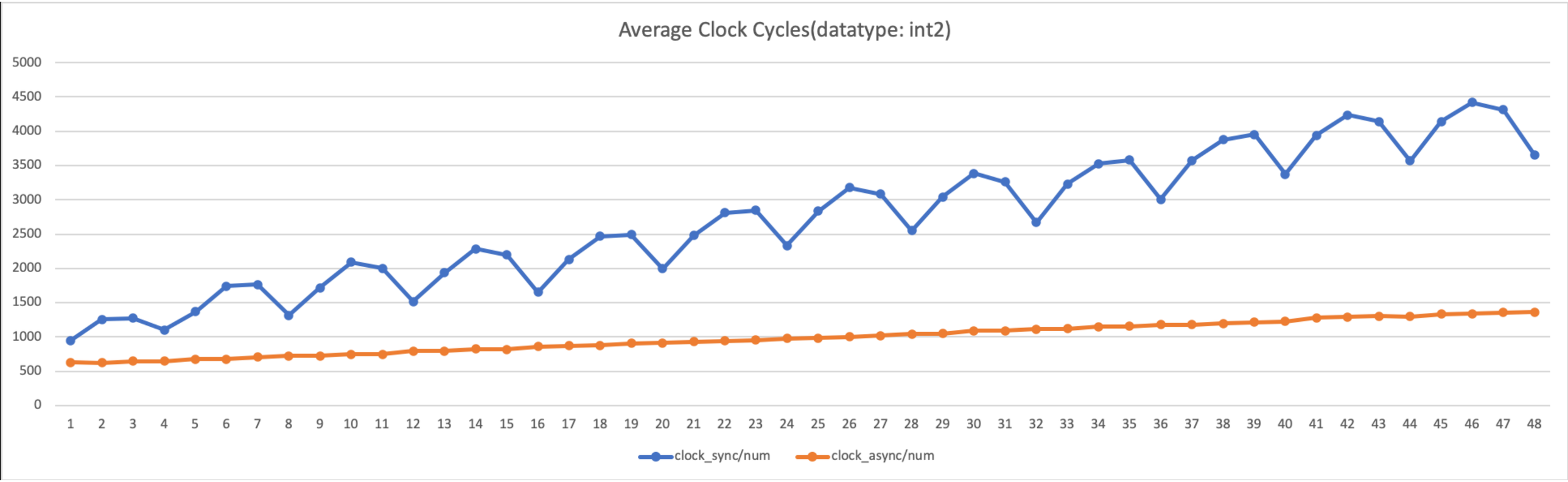
copy_count

1 to 49152 (int)
1 to 24576(int2)
1 to 12288 (int4)

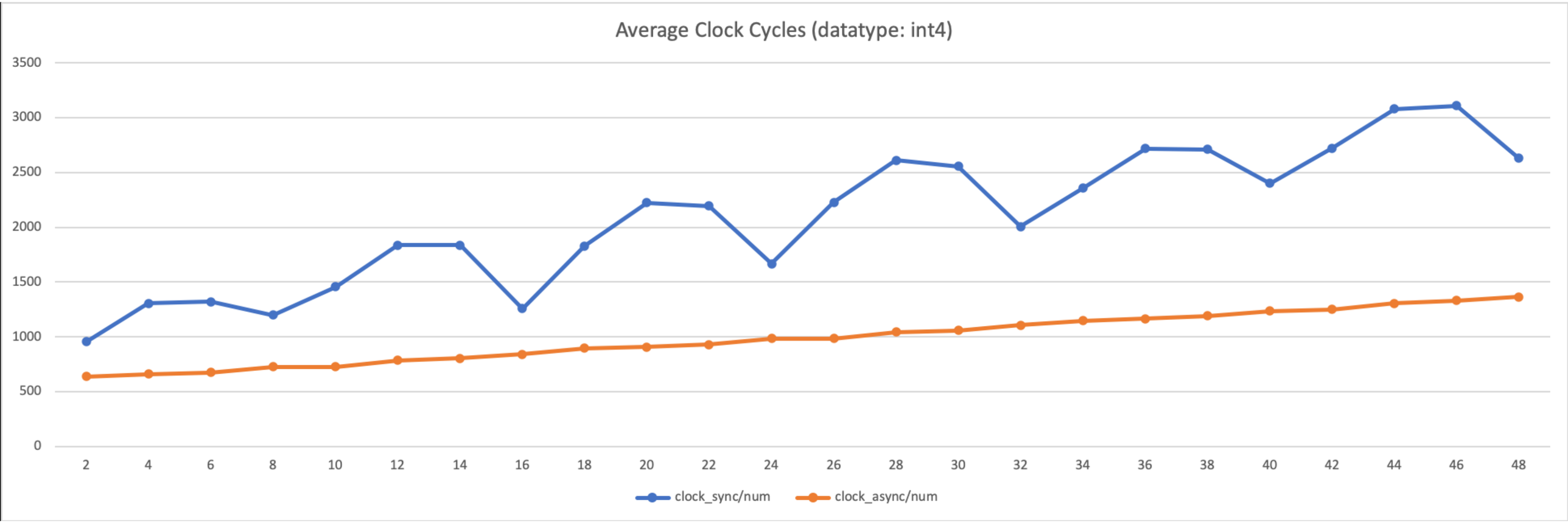
# EXAMPLE: ASYNC-COPY



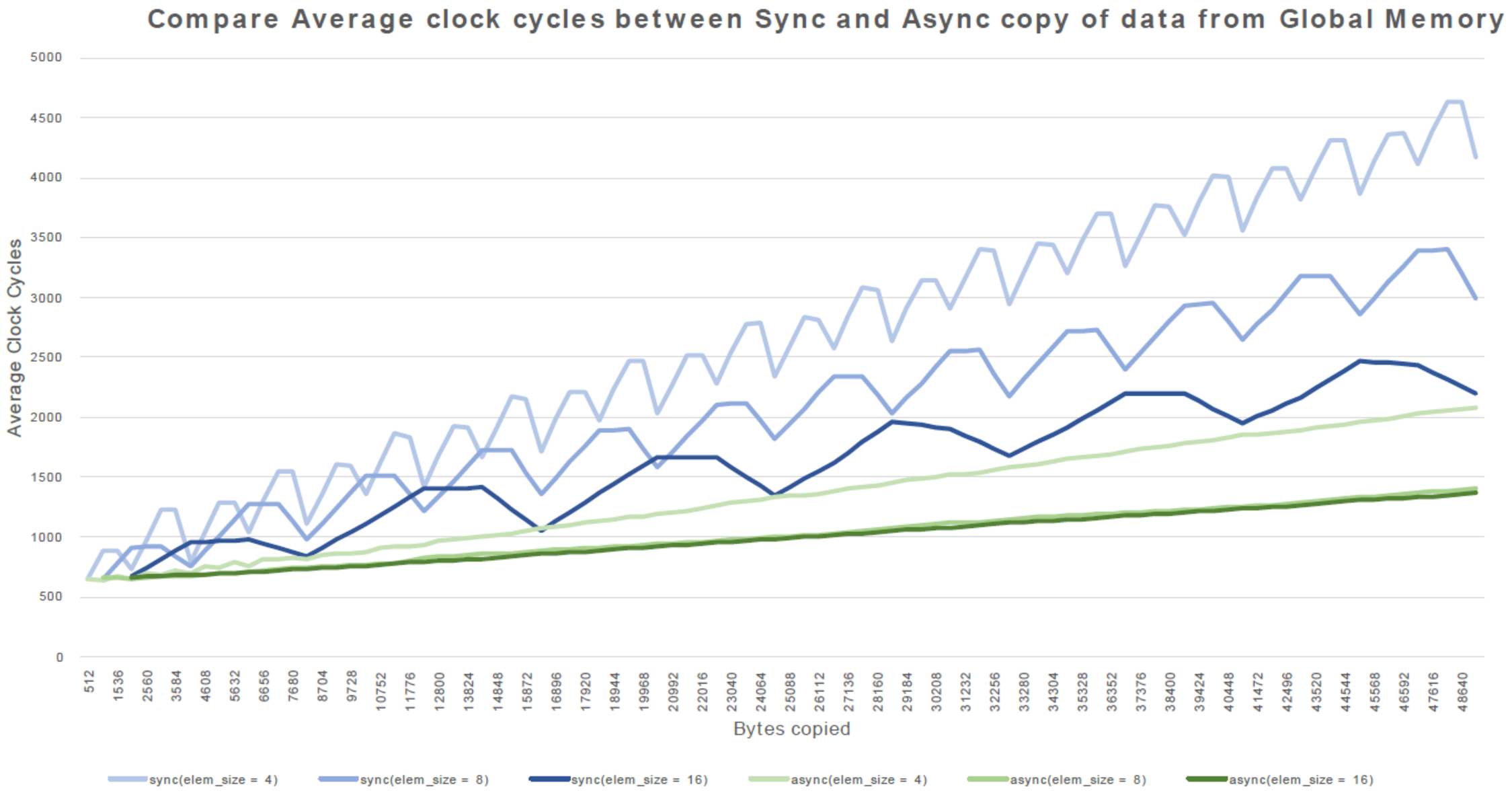
# EXAMPLE: ASYNC-COPY

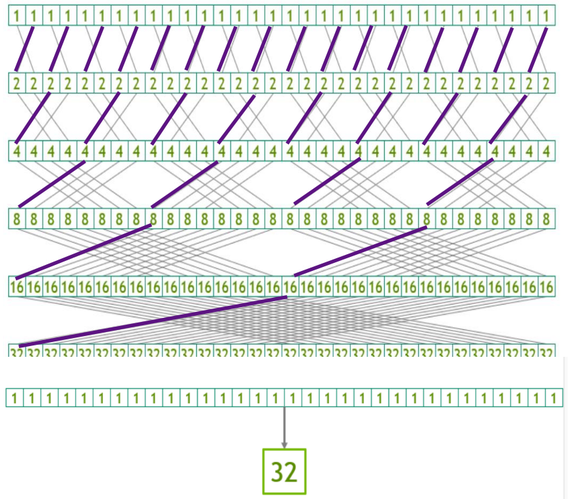


# EXAMPLE: ASYNC-COPY

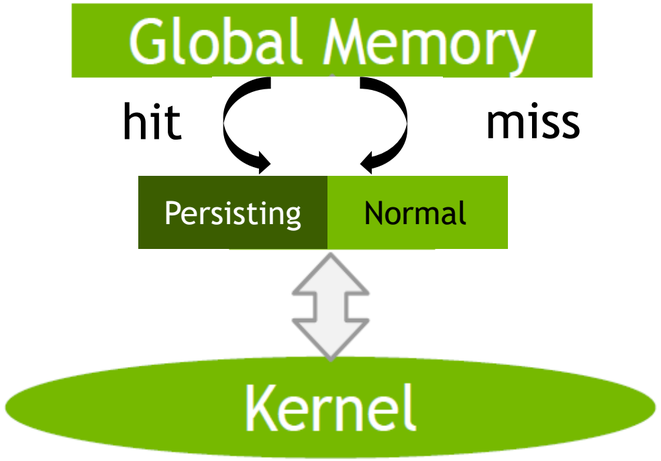


# EXAMPLE: ASYNC-COPY

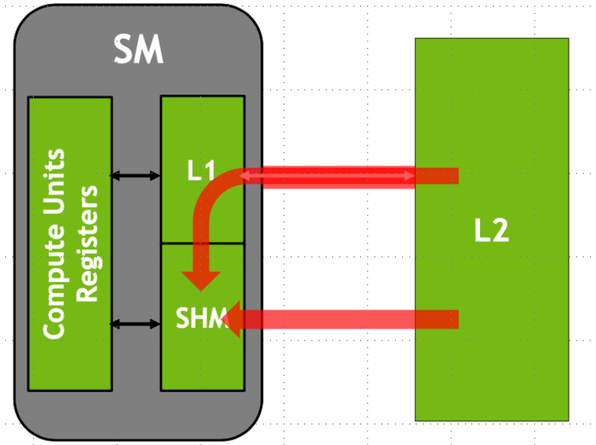




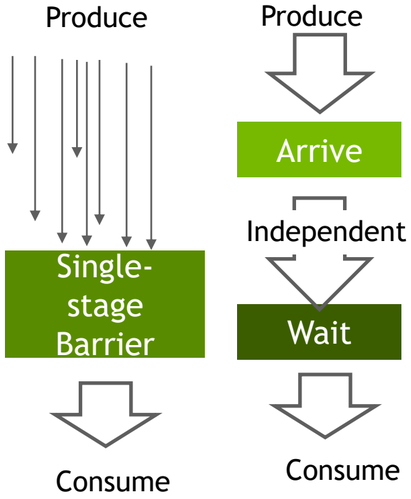

Warp Synchronous Reduction




L2 cache residency controls




Asynchronous copy




**Asynchronous barrier**

# ASYNCHRONOUS BARRIER

## What's Asynchronous barriers

CUDA 11 introduces a split arrive/wait barrier

Compute capability >=8.0 : **hardware acceleration** for barrier operations

7.0 <=Compute capability <8.0: barriers available, **without hardware** acceleration.

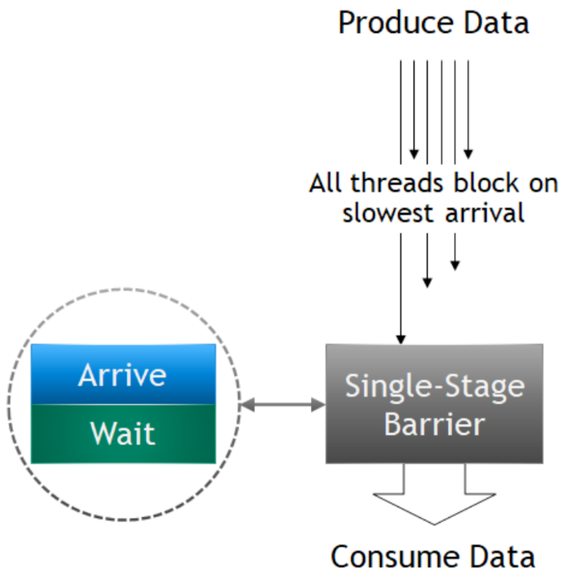
AWBarrier (CUDA 11.0 / -std=c++11 )

```
#include <cuda_awbarrier.h>
using namespace nvcuda::experimental;
```

# ASYNCHRONOUS BARRIER

## What's Asynchronous barriers

Synchronization is achieved using **__syncthreads()** (to synchronize all threads in a block) or **group.sync()** when using Cooperative Groups.

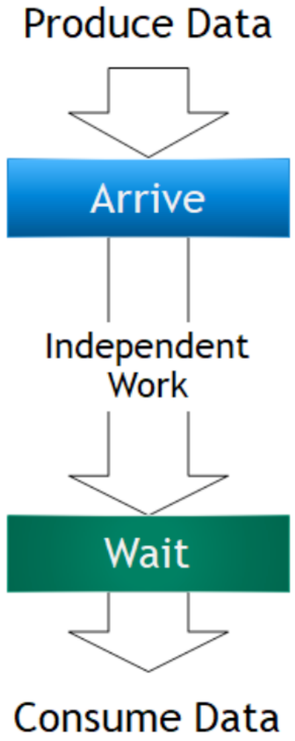



**Single-Stage barriers** combine back-to-back arrive & wait

```
__global__ void my_kernel() {
    auto block = cooperative_groups::this_thread_block();
    for (int i = 0; i < N; ++i) {
        /* code before arrive */
        block.sync(); /* wait for all threads to arrive here */
        /* code after wait */
    }
}
```

# ASYNCHRONOUS BARRIER

## What's Asynchronous barriers




**Arrival:** notify **a thread** that **it** has reached the barrier

Pipelined processing

**Wait**: wait for **others** to arrive at the barrier

Asynchronous barriers enable pipelined processing

# ASYNCHRONOUS BARRIER

## What's Asynchronous barriers

```
using namespace nvcuda::experimental; __global__ void my_kernel() {
__shared__ awbarrier bar;
auto block = cooperative_groups::this_thread_block();

if (block.thread_rank() == 0)
    init(&bar, block.size());
block.sync();

for (int i = 0; i < N; ++i) {
    /* code before arrive */

    awbarrier::arrival_token tok = bar.arrive(); /* this thread arrives */

    /* code between arrive and wait */

    bar.wait(tok); /* wait for all threads to bar.arrive() */

    /* code after wait */
    }
}
```

Arrive

Wait

# ASYNCHRONOUS BARRIER

## Init

- bar must be a pointer to __shared__ memory.
- **Expected arrive count :**
  The number of times **bar.arrive()** will be called before unblocked
- Initialize with number of threads that will participate
- Benefits: Enabling Arbitrary Subgroups of a Thread Block

```
if (block.thread_rank() == 0)
    init(&bar, block.size());
block.sync();
```

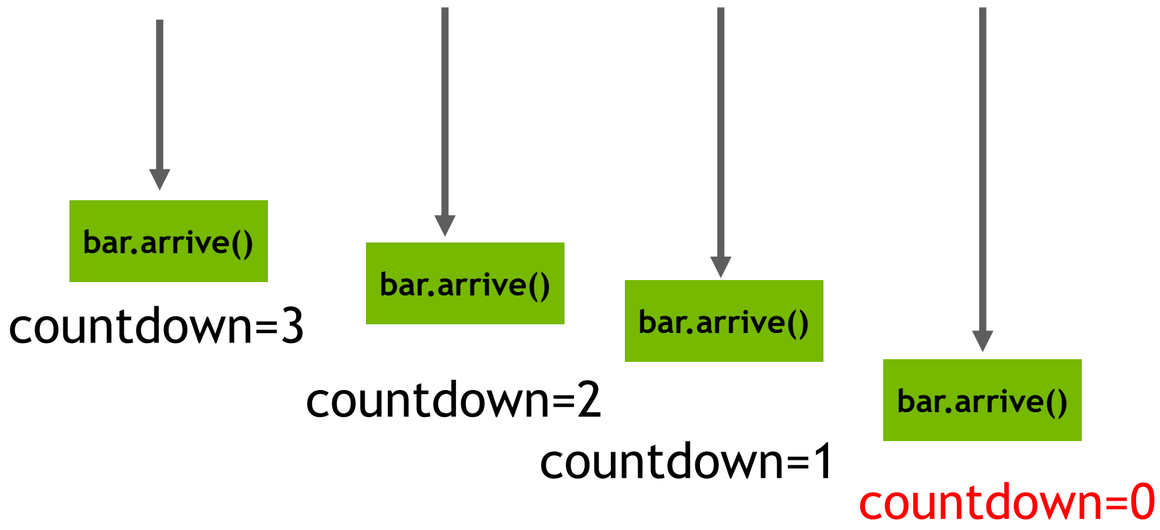
# ASYNCHRONOUS BARRIER

## Init

**Countdown:**

An **awbarrier** counts down from **the expected arrive count** to **zero** as participating threads call **bar.arrive()**.

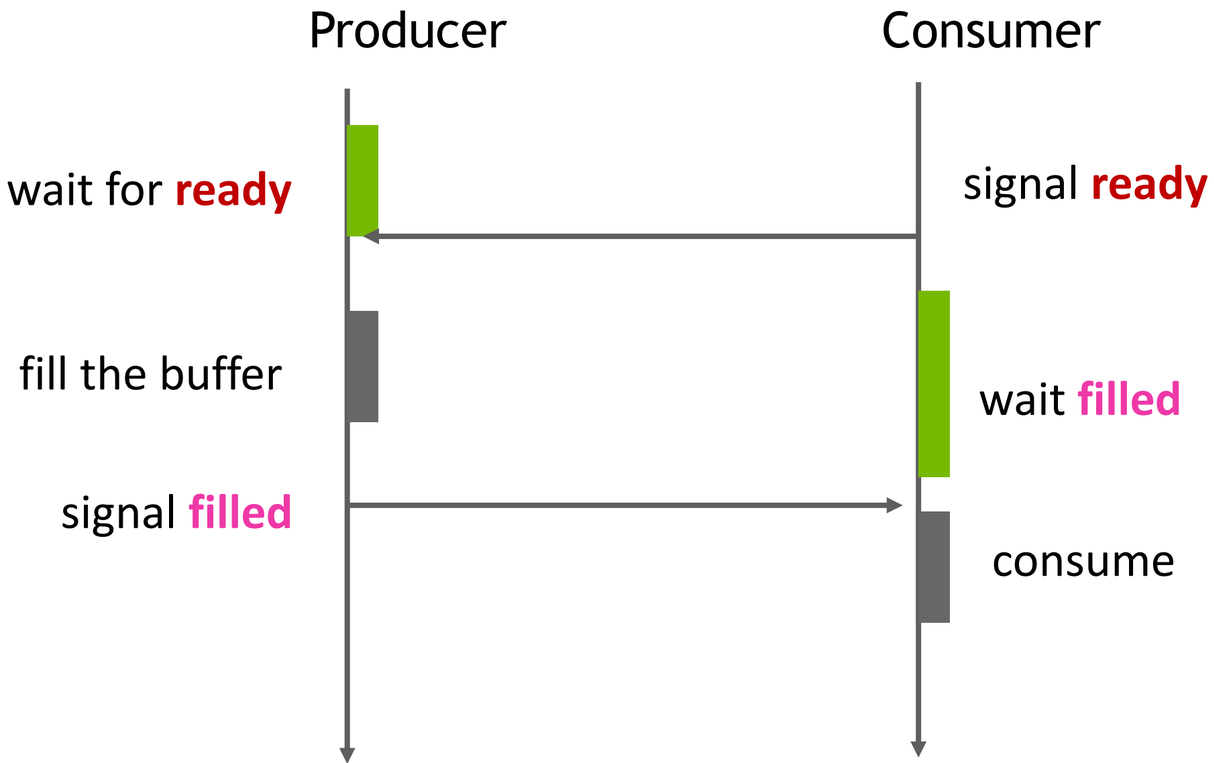
Expected arrive count=4



# ASYNCHRONOUS BARRIER

## Example: Warp Specialization

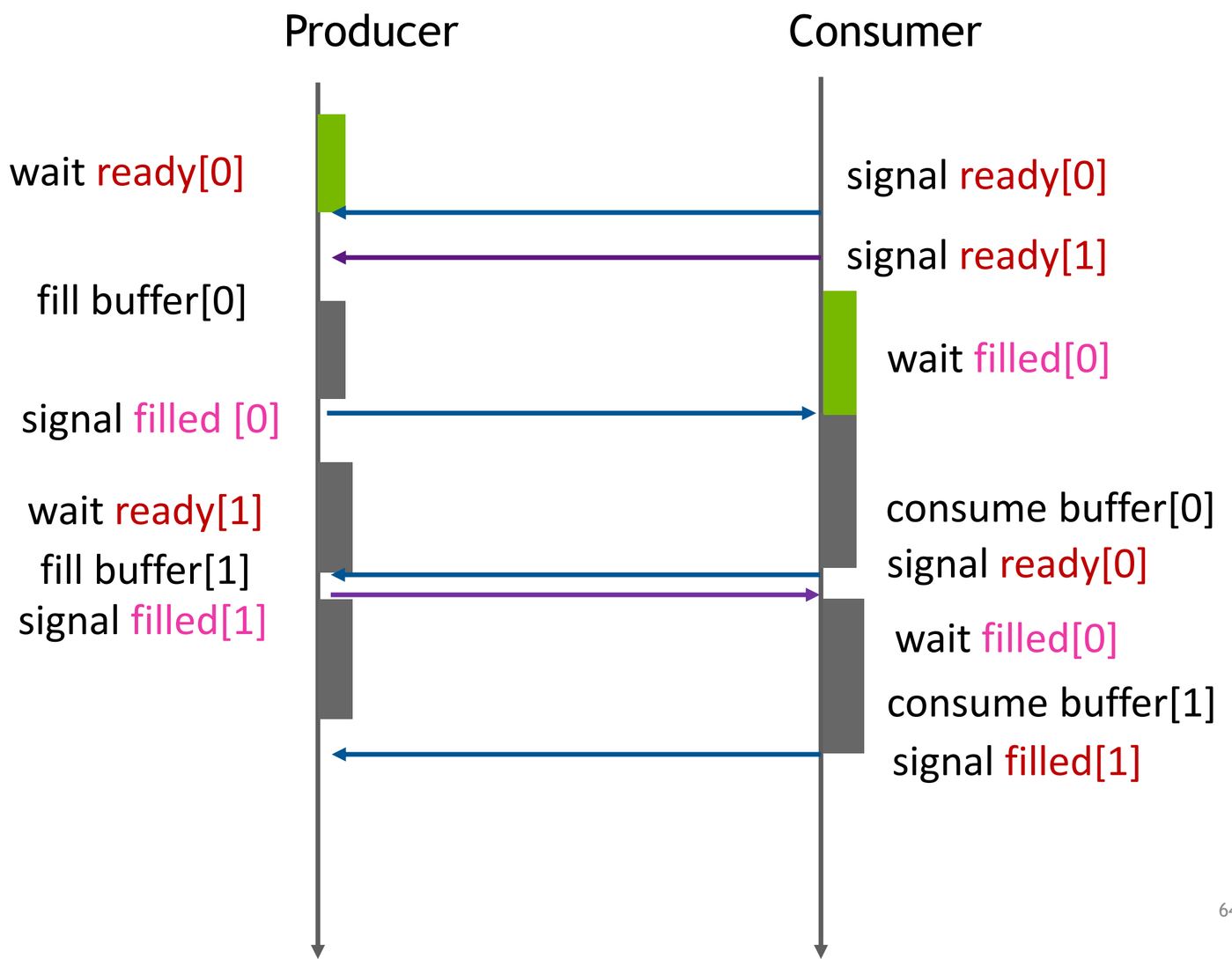
- Spatial partitioning is used in a **producer or consumer pattern**
  - Producer: One subset of threads produces data
  - Consumer: Concurrently consumed by the other (disjoint) subset of threads
- Each buffer requires two **awbarriers**



# ASYNCHRONOUS BARRIER

## Example: Warp Specialization

For full producer/consumer concurrency this pattern has (at least) **double** buffering



# ASYNCHRONOUS BARRIER

## Example: Warp Specialization

4 **awbarrier**s: First two: ready buffer (ready[2]) / Second two: filled buffer (filled[2] )

```
__shared__ awbarrier bar[4]
__shared__ extern int buffer[];
auto block =
cooperative_groups::this_thread_block();

if (block.thread_rank() < 4)
    init(bar + block.thread_rank(), block.size());
block.sync();
```

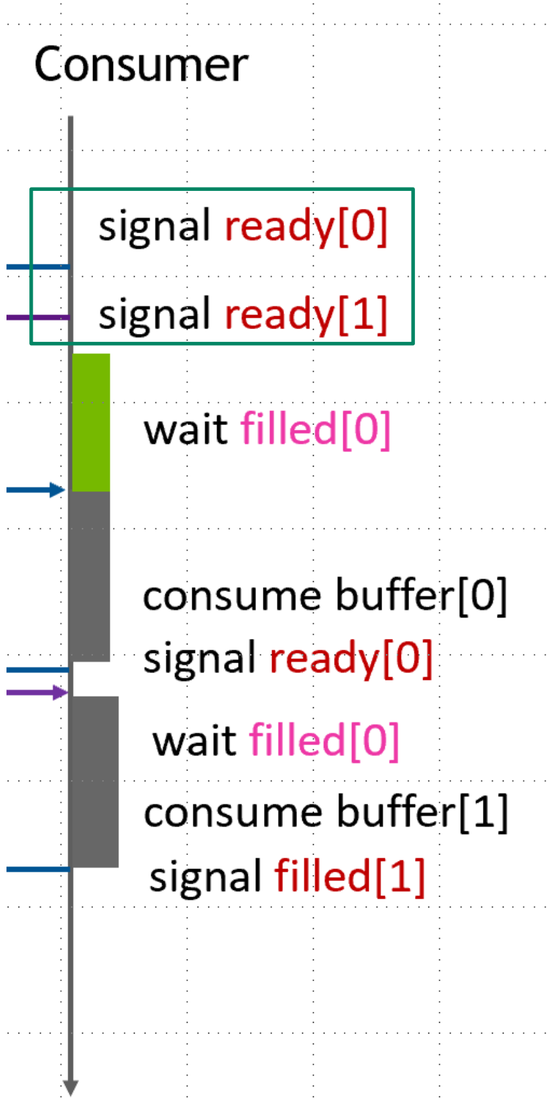
4 **awbarriers** initialization

```
if (block.thread_rank() < warpSize)
    producer(bar, bar+2, buffer);
else
    consumer(bar, bar+2, buffer);
}
```

First warp : the producer
Remaining warps : the consumer

# ASYNCHRONOUS BARRIER

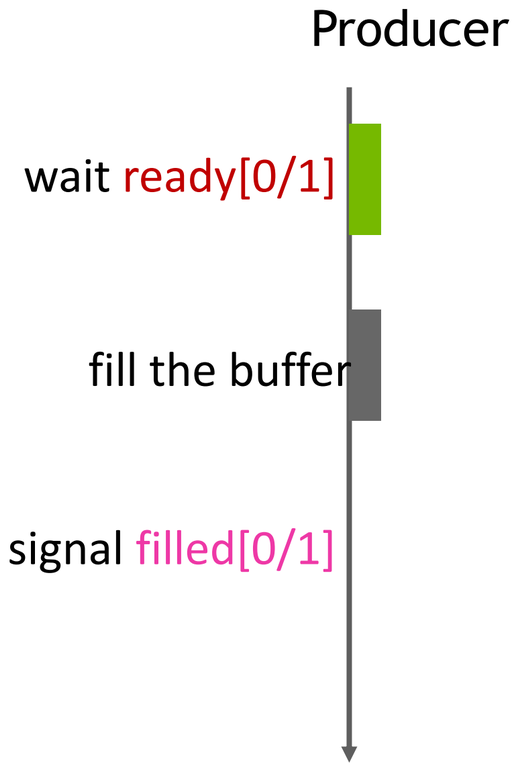
## Example: Warp Specialization




```
__device__ void consumer(awbarrier ready[],
                         awbarrier filled[], int* buffer)
{
    /* buffer_0 is ready for initial fill */
    ready[0].arrive();
    /* buffer_1 is ready for initial fill */
    ready[1].arrive();

    for (int i = 0; i < N; ++i) {

        /* wait for buffer_(i%2) to fill */
        filled[i%2].arrive_and_wait();

        /* consume buffer + length*(i%2) */

        /* buffer_(i%2) is ready to be re-filled */
        ready[i%2].arrive();
    }
}
```

# ASYNCHRONOUS BARRIER

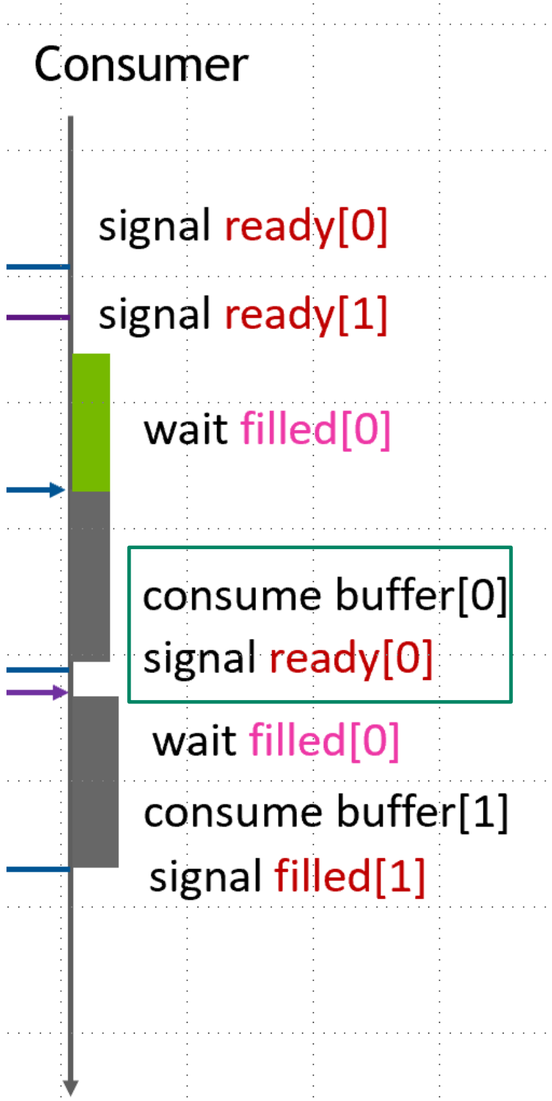
## Example: Warp Specialization



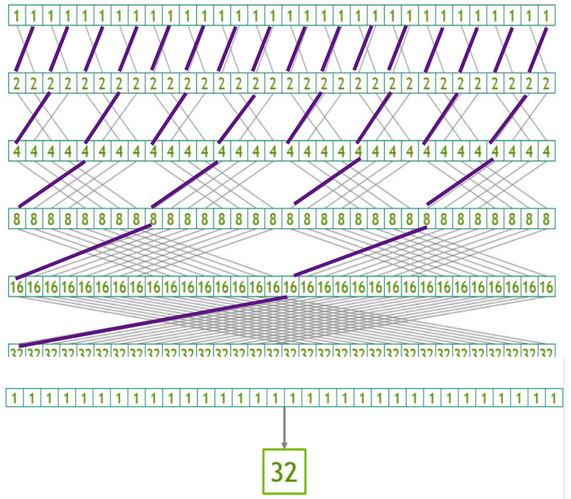

```
__device__ void producer(awbarrier ready[],
                         awbarrier filled[], int* buffer)
{
    for (int i = 0; i < N; ++i) {
        /* wait for buffer_(i%2) ready to filled */
        ready[i%2].arrive_and_wait();

        /* produce buffer + length*(i%2) */

        /* buffer_(i%2) is filled */
        filled[i%2].arrive();
    }
}
```

# ASYNCHRONOUS BARRIER

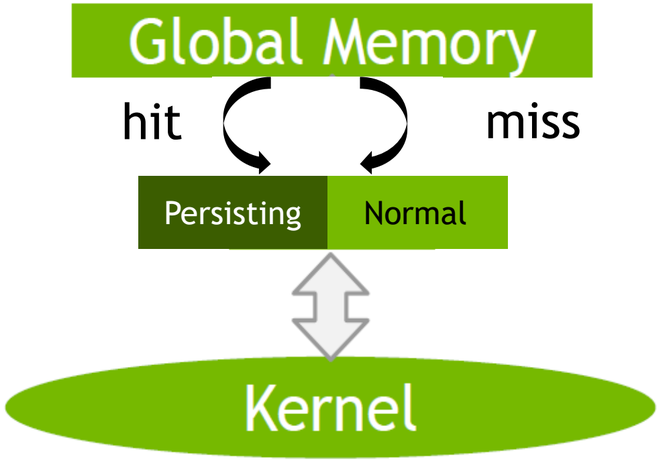
## Example: Warp Specialization



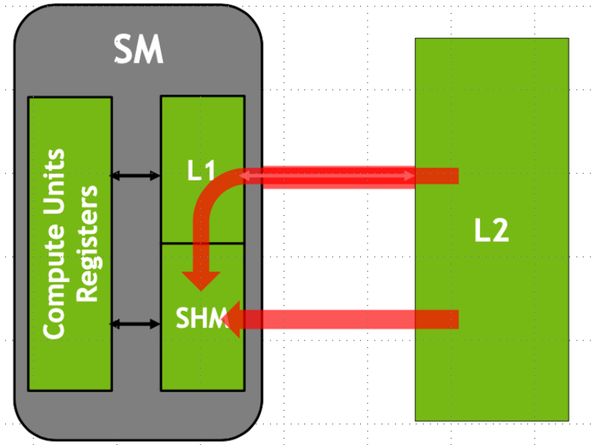

```
__device__ void consumer(awbarrier ready[],
                         awbarrier filled[], int* buffer)
{
    /* buffer_0 is ready for initial fill */
    ready[0].arrive();
    /* buffer_1 is ready for initial fill */
    ready[1].arrive();

    for (int i = 0; i < N; ++i) {

        /* wait for buffer_(i%2) to fill */
        filled[i%2].arrive_and_wait();

        /* consume buffer + length*(i%2) */

        /* buffer_(i%2) is ready to be re-filled */
        ready[i%2].arrive();
    }
}
```



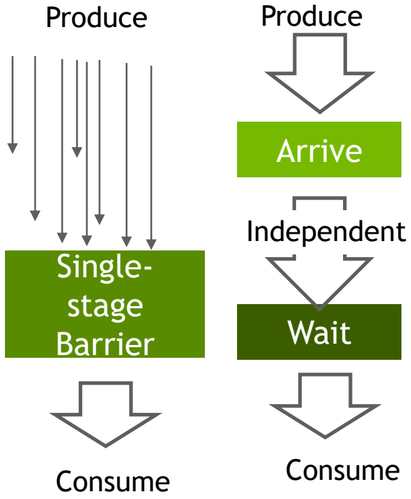

Warp Synchronous Reduction



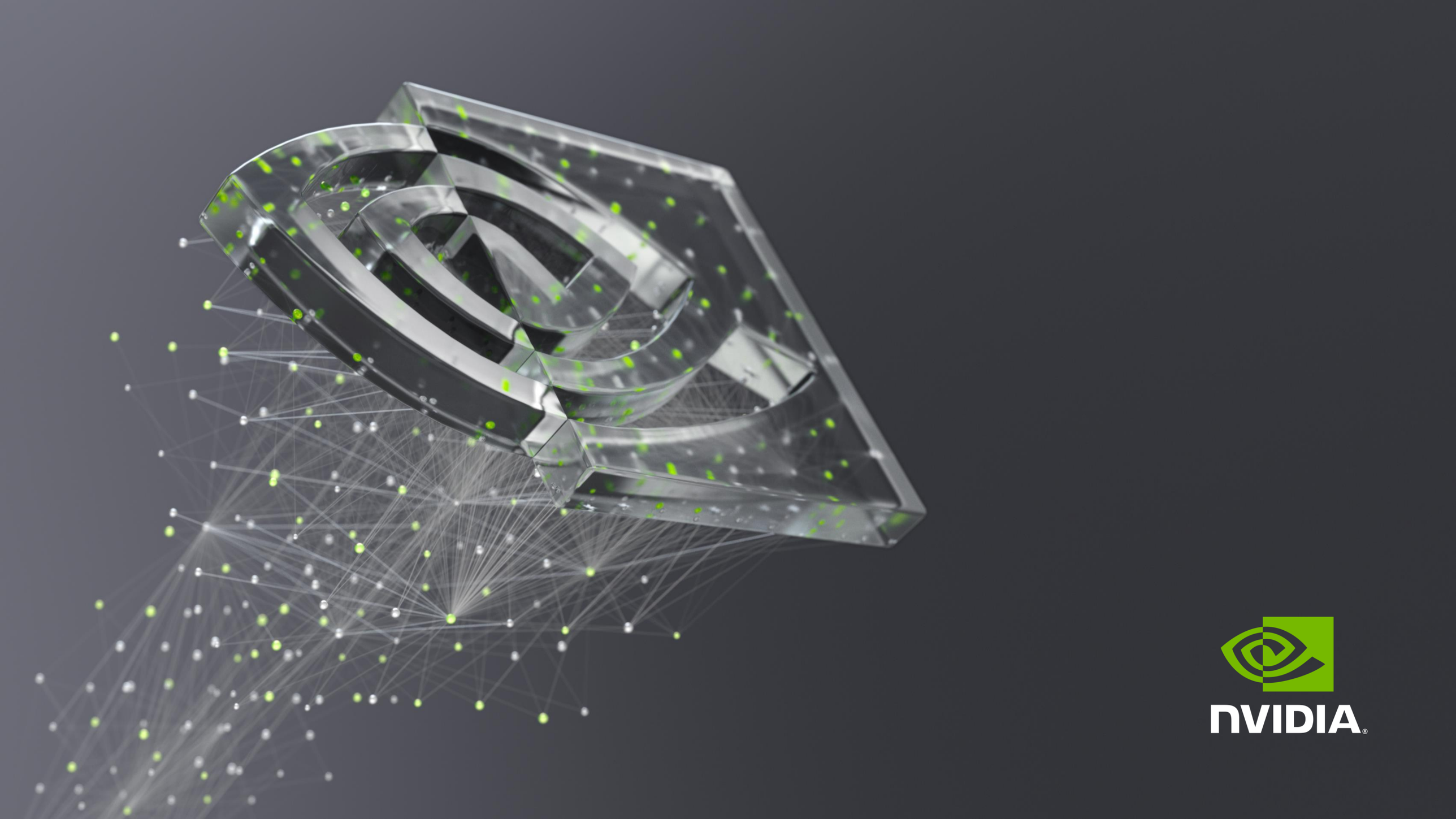

L2 cache residency controls




Asynchronous copy




Asynchronous barrier


NVIDIA