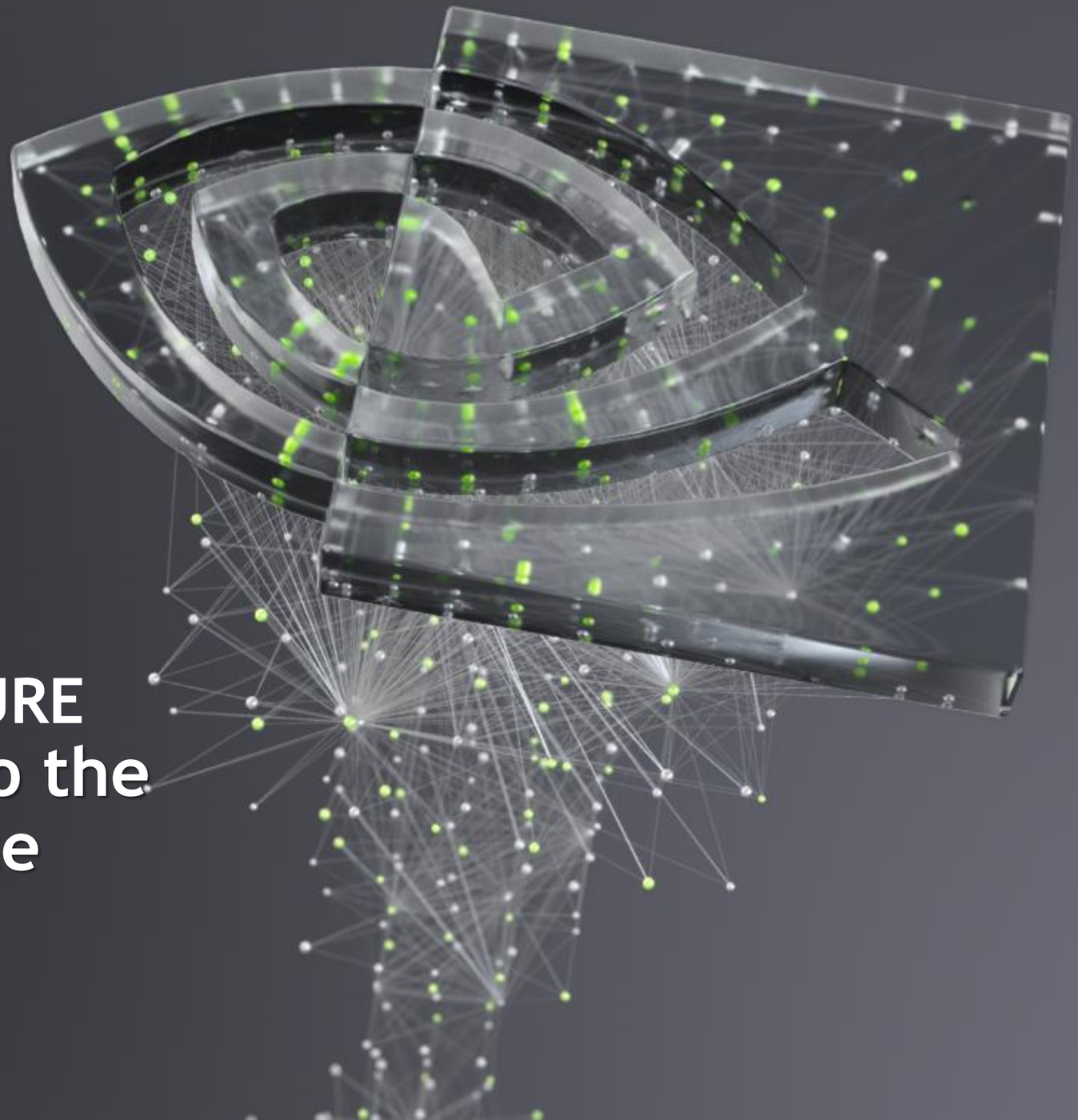


CUDA on NVIDIA GPU
AMPERE MICROARCHITECTURE
Taking your algorithms to the next level of performance

Carter Edwards, May 19, 2020

CUDA 11.0 Enhancements

Leveraging NVIDIA Ampere GPU Microarchitecture

Asynchronously Copy Global → Shared Memory

Flexible Synchronization for Producer → Consumer and other Algorithms

Influence Residency of Data in L2 Cache

Warp Synchronous Reduction


Asynchronously
Copy Global → Shared Memory

__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[] */
}
```

the __syncthreads() sandwich

Global → Shared Memory Journey

The journey may be longer than it appears

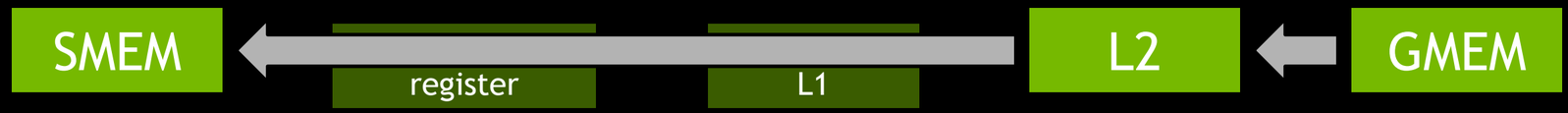
anatomy of copy shared ← global

```
shbuf[i] = gldata[i];
```

Journey through memory before GA100

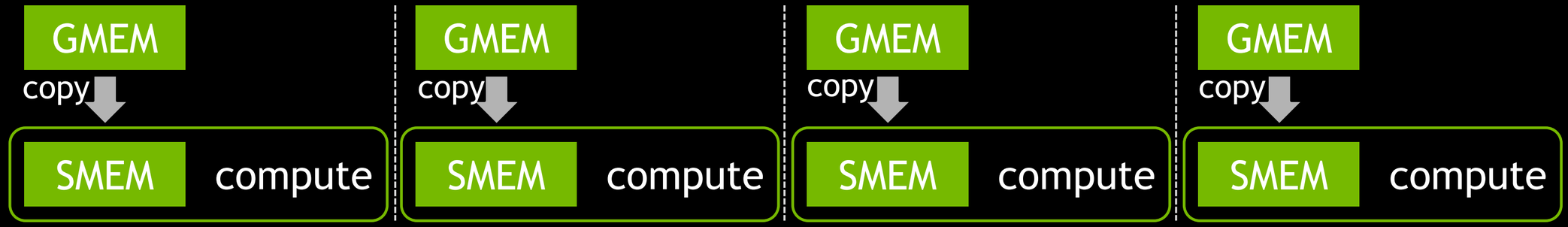
Better: Don’t pass through registers along the way; *using fewer registers can improve occupancy*

Even better: Don’t pass through L1 cache along the way; *let other data persist longer*

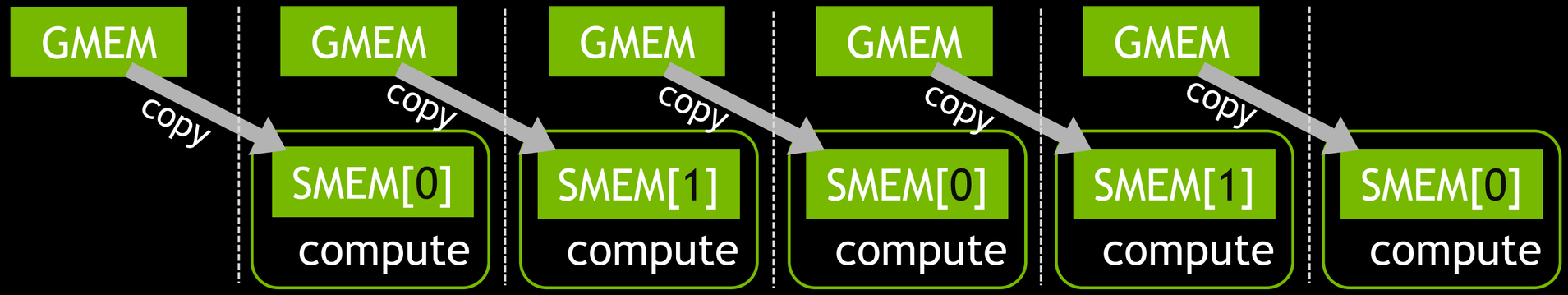


Copy and Compute Sequencing

Each iteration: First copy GMEM ➡ SMEM; then Compute on SMEM



Better to Compute while Copying for later iteration(s)



Example two stage pipelining of copy and compute

Async-Copy Pipeline

Begin with Simple One-Stage Pipeline

```
pipeline pipe;

memcpy_async(dst, src, pipe);

pipe.commit_and_wait();
```

Submit asynchronous copy via the better journey

- To **`dst`** in shared memory
- From **`src`** in global memory
- Data type is trivially copyable

Thread submits as many async-copy as needed

Thread waits for all submitted asynchronous copy operations to complete

Update Implementation with Async-Copy

Begin with Simple One-Stage Pipeline

Before GA100 GPU

```
__shared__ extern int shbuf[];

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

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

Now

```
__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[] */
}
```

Still have the __syncthreads() sandwich

Async-Copy Pipeline

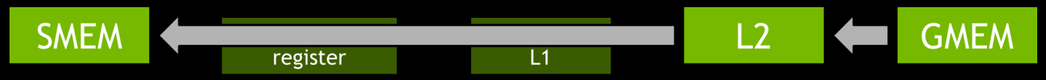
Then Improve with Even Better Journey through Memory

```
memcpy_async(dst, src, pipe);
```

Better async-copy journey

- To shared memory
- From global memory
- Data type is trivially copyable
- Data size is multiple of 4 bytes
- Data is aligned to 4 bytes

Even better async-copy journey



- To shared memory
- From global memory
- Data type is trivially copyable
- Data size is multiple of 16 bytes
- Data is aligned to 16 bytes

Multi-Stage Pipeline

Then Improve by Overlapping Copy and Compute

```
pipeline pipe;

memcpy_async( dst, src, pipe);

pipe.commit();

pipe.wait_prior<N>();
```

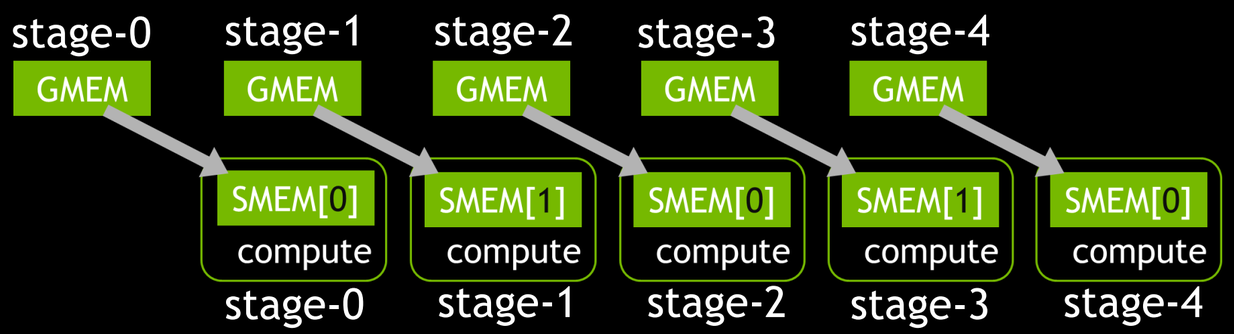
Submit async-copy for **`dst = src;`**

- Submit as many as needed
- Submit into new stage of pipeline

Commit new stage K, but do not wait for it now

- A sequence of stages { 0, 1, ..., K }

Wait for prior stage K-N in the pipeline of sync-copy operations to complete



Update Algorithm with Multi-Stage Pipeline

(1 of 2) Similar Pattern: Async-Copy, Commit, Wait

One Stage

```
for (stage=0; stage < end; ++stage){
  __syncthreads();


  for ( i = ... ) {
    memcpy_async(sh[i],gl[i],pipe);
  }
  pipe.commit_and_wait();


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

Multi-Stage

```
for (stage=next=0; stage < end; ++stage){
   /* __syncthreads(); */
   for (; next < stage + nStage ; ++next){
      s = next % nStage ;
      for ( i = ... ) {
        memcpy_async(sh[s][i],gl[i],pipe);
      }
      pipe.commit();
   }
   pipe.wait_prior< nStage-1 >();
   __syncthreads();
   /* compute on sh[stage % nStage][] */
}
```

Update Algorithm with Multi-Stage Pipeline

(2 of 2) Declare, Fill, and Wait with N-Stages of Buffers

Removed leading **`__syncthreads()`**

Submit async-copy for later stages

- Recycle among **`nStage`** buffers

Commit **`next`** stage but do not wait

Wait for current stage of copies to complete

- **`stage = next - (nStage-1);`**
- Prior stage relative to most recent commit

Wait for all threads to complete copies

Multi-Stage

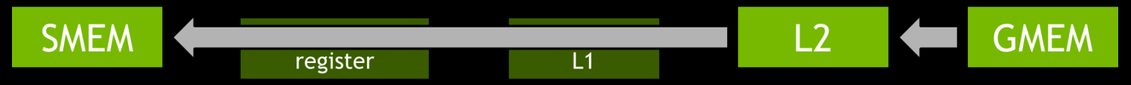
```
for (stage=next=0; stage < end; ++stage){
  /* __syncthreads(); */
  for (; next < stage + nStage ; ++next){
    s = next % nStage ;
    for ( i = ... ) {
      memcpy_async(sh[s][i],gl[i],pipe);
    }
    pipe.commit();
  }
  pipe.wait_prior< nStage-1 >();
  __syncthreads();
  /* compute on sh[stage % nStage][] */
}
```

Pairs Well with Cooperative Groups

Collective Async-Copy of a whole Array

```
template<class GroupType, class T>
size_t memcpy_async( GroupType & group, T * dstPtr, size_t dstCount
                                        const T * srcPtr, size_t srcCount
                                        pipeline & pipe );
```

- `GroupType` is an intra-block Cooperative Group
- Partitions array range [0..dstCount-1] among threads
- Submits async-copy for: `dstPtr[0..srcCount-1] = srcPtr[0..srcCount-1];`
- Zero fill left-overs: `dstPtr[srcCount..dstCount-1] = 0;`
- Given aligned arrays will use



Async-Copy Microbenchmark

Your Algorithm's Mileage May Vary

Microbenchmark Comparing Synchronous vs. Asynchronous Copy

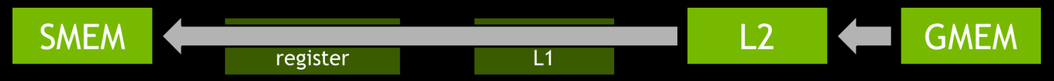
`shbuf[i]=gldata[i];` vs. `memcpy_async(shbuf[i],gldata[i],pipe);`

Simple one-stage pipeline

- Without computations
- Ample registers available

Results: Consistently better performance; best when

- Data type is 16 bytes to get



- Modest thread block size

Except, a corner case where traditional synchronous copy can perform better

Microbenchmark Performance Experiment

Microbenchmark Comparing Synchronous vs. Asynchronous Copy

```
/* sync: Conventional synchronous memory copy */
for (size_t i = 0; i < copy_count; ++i) {
    shared[blockDim.x * i + threadIdx.x] = global[blockDim.x * i + threadIdx.x];
}
```

```
/* async: Asynchronous memory copy */
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>();
```

Performance Experiment

Varied Thread Block Size and Sizeof Data Type

Thread block sizes : 128, 256, 512

Data type sizes : 4 and 16 bytes

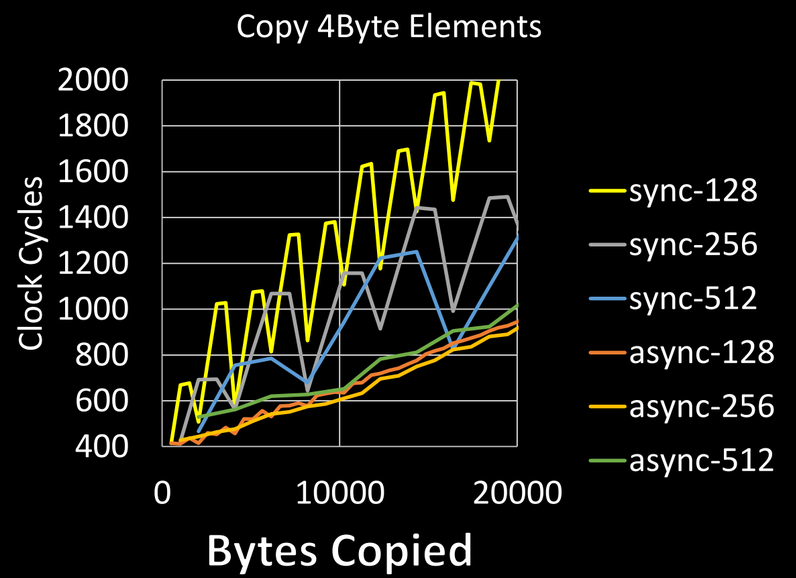
Measure clock-cycles required to copy an array of N bytes

- Y-axis = clock-cycles
- X-axis = bytes copied

Simple experiment with surprisingly complicated results

- Now step through results

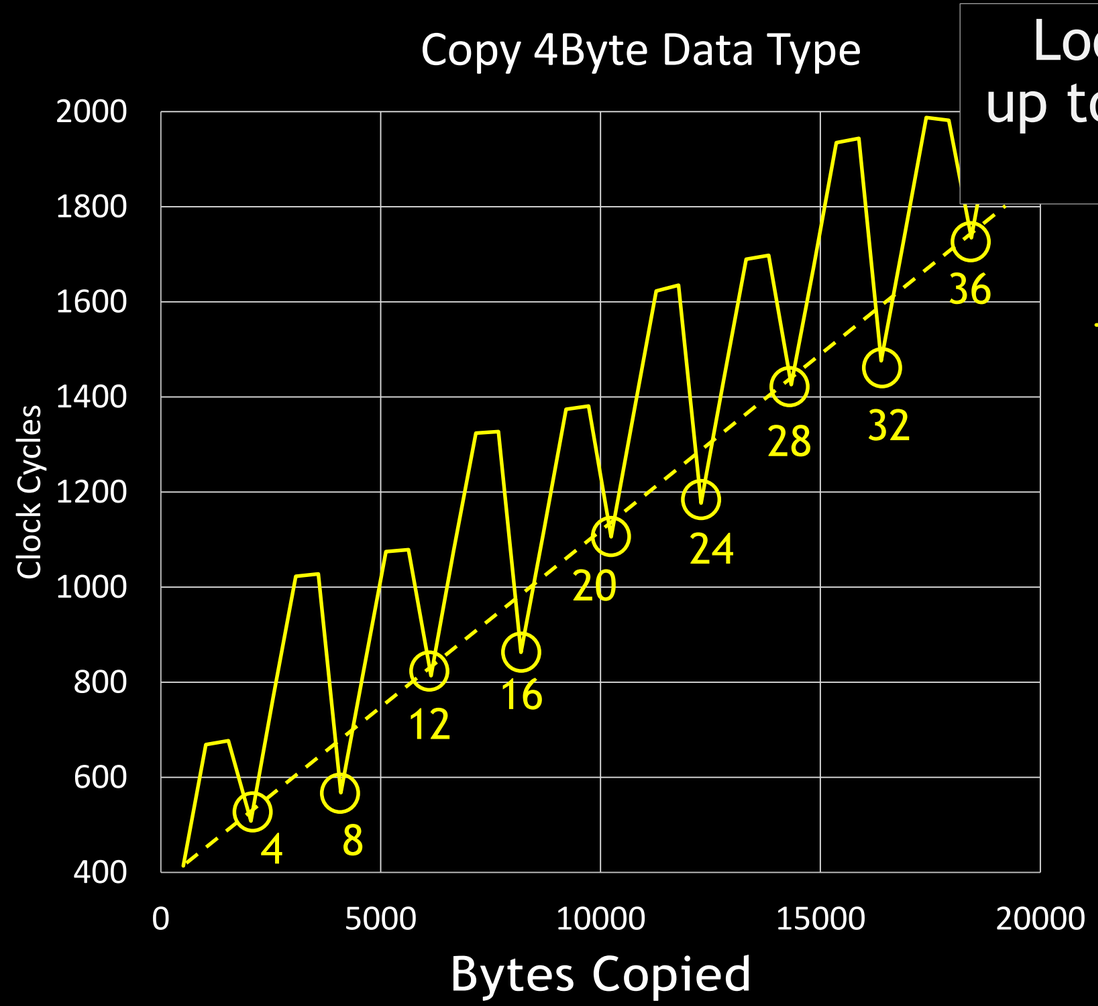
```
/* Conventional synchronous memory copy */
for (size_t i = 0; i < copy_count; ++i) {
    shared[blockDim.x*i + threadIdx.x] =
    global[blockDim.x*i + threadIdx.x];
}
```



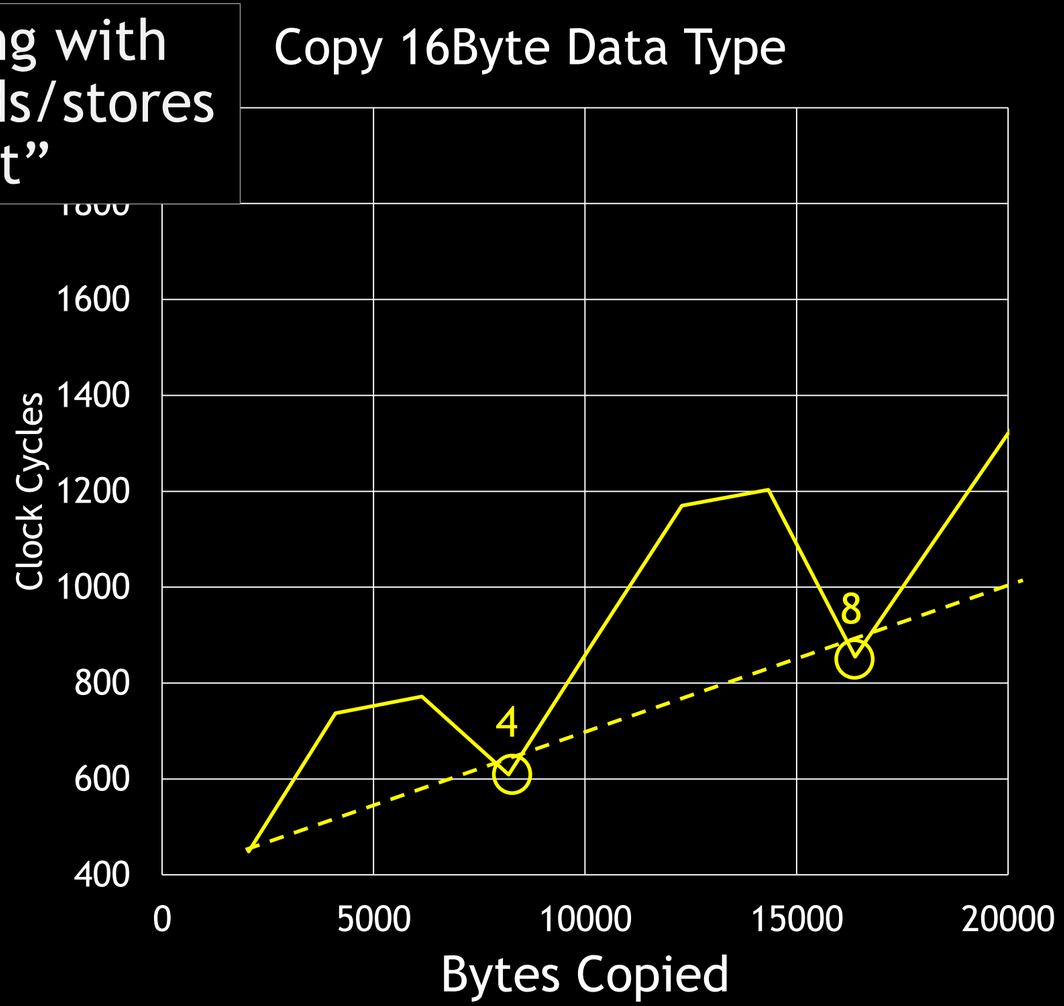
Performance Experiment

(1 of 4) Compiler Optimizes Traditional Synchronous Copy

Loop unrolling with up to four loads/stores "in flight"



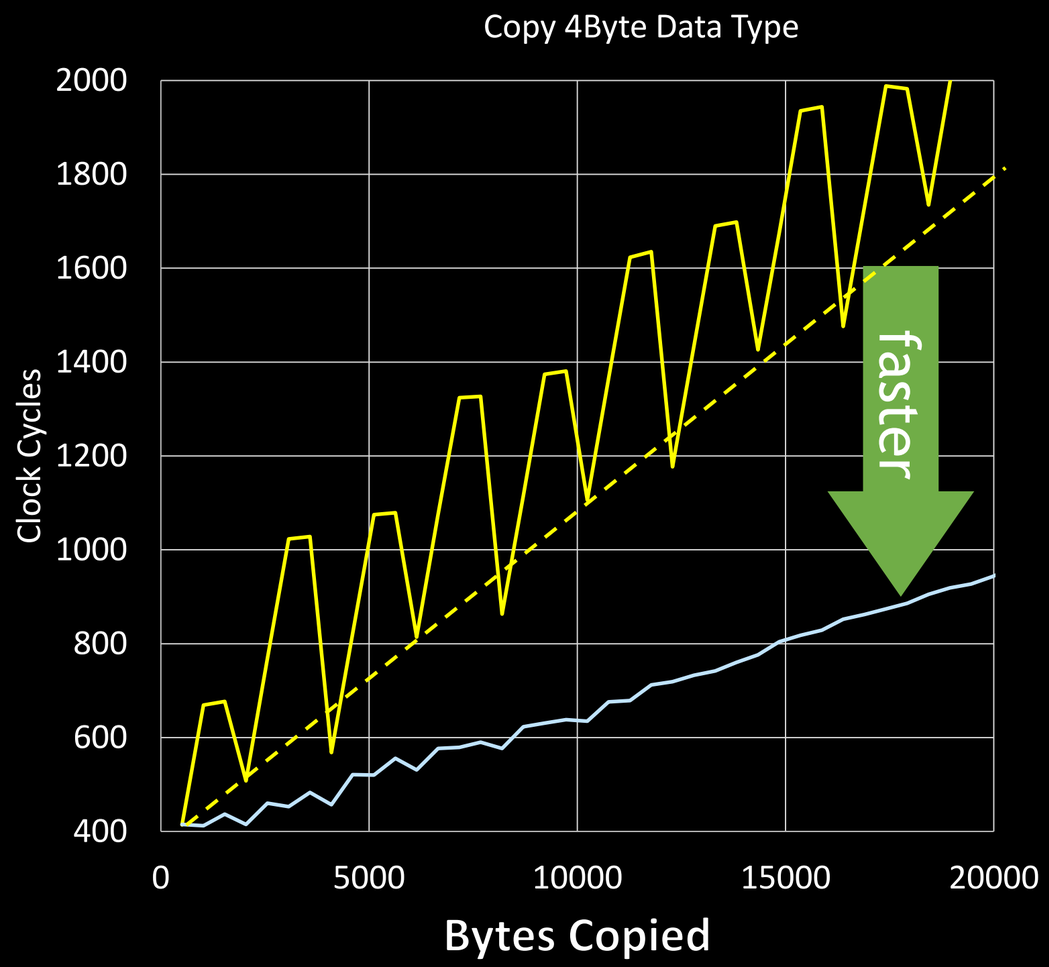
time to copy



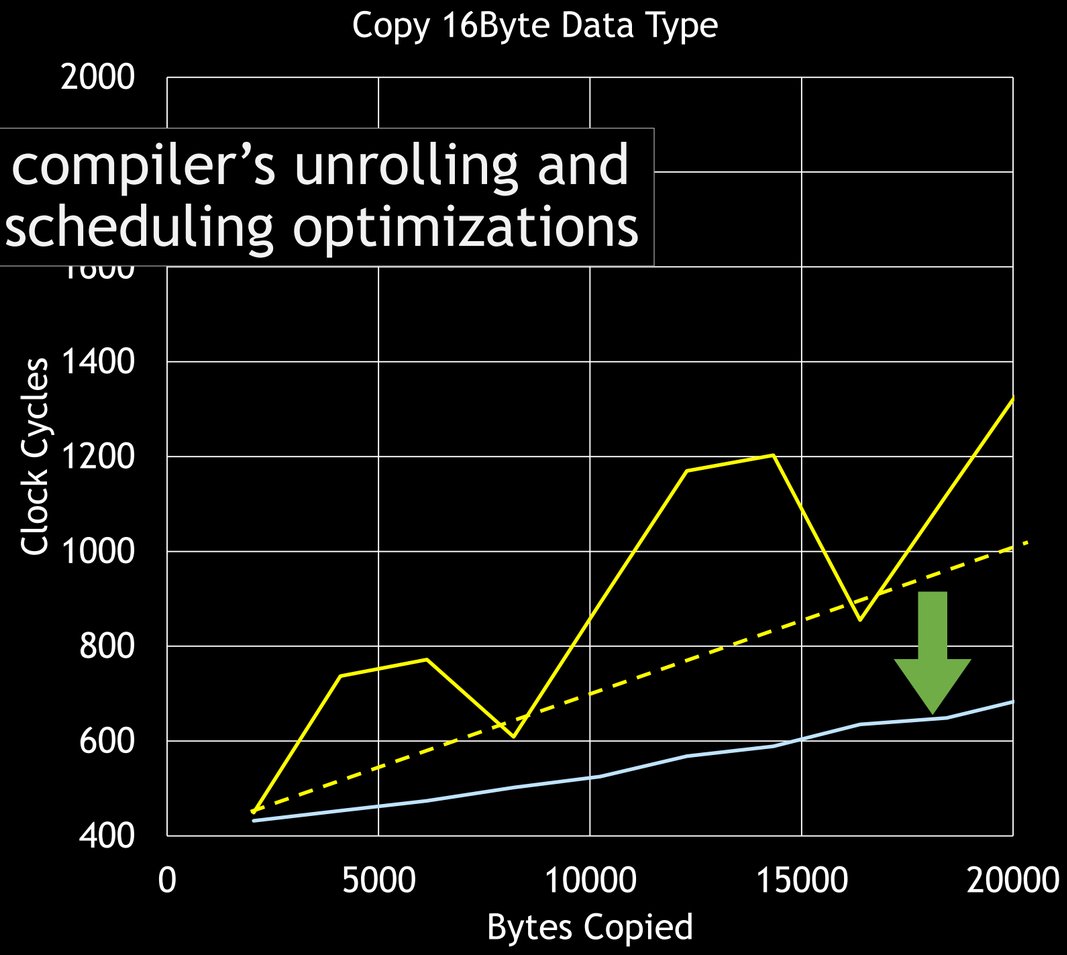
NVIDIA

Performance Experiment

(2 of 4) Async-Copy is Faster

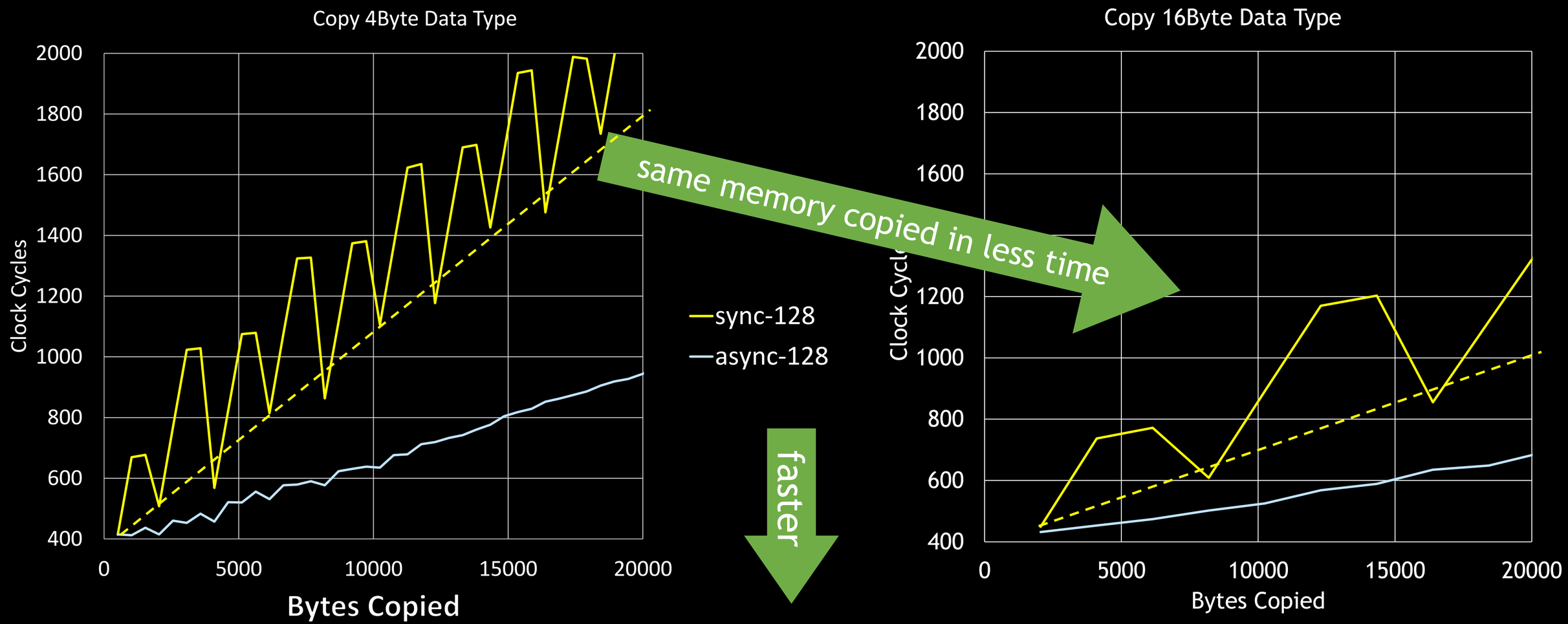


Don’t need compiler’s unrolling and instruction scheduling optimizations



Performance Experiment

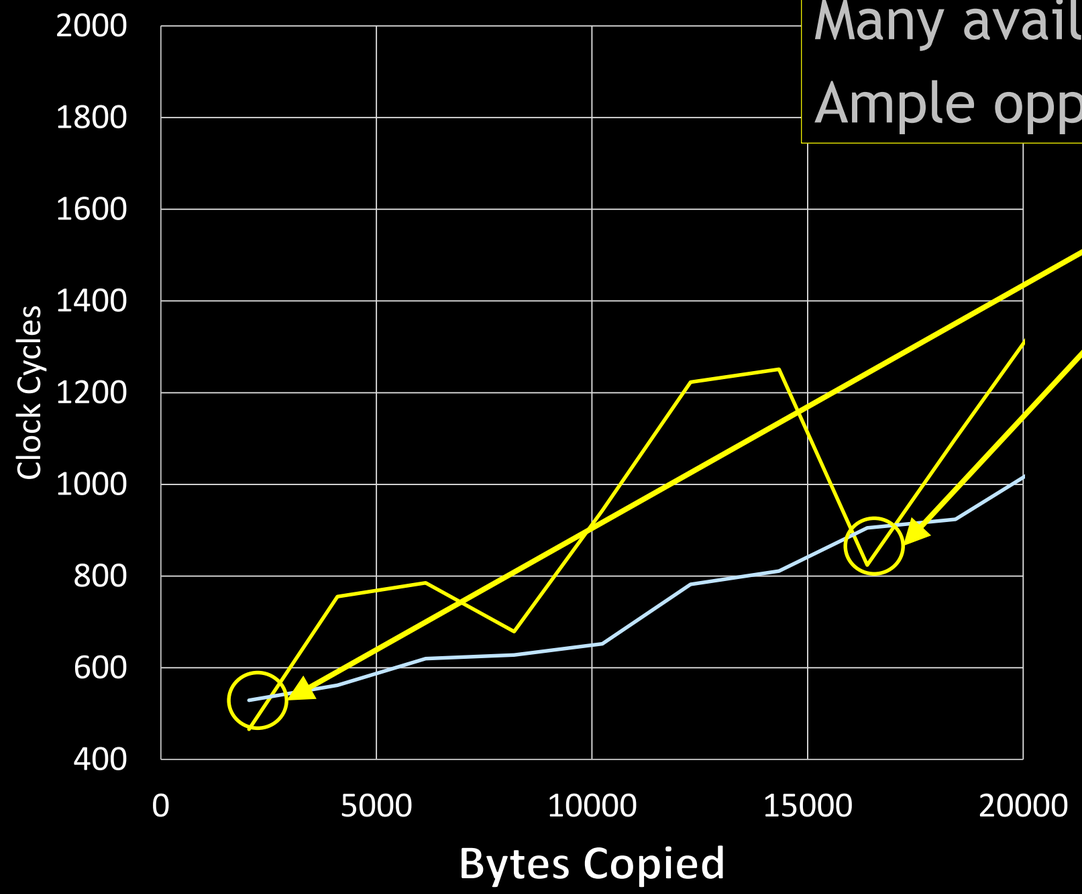
(3 of 4) Copying 16byte Data Type is Faster

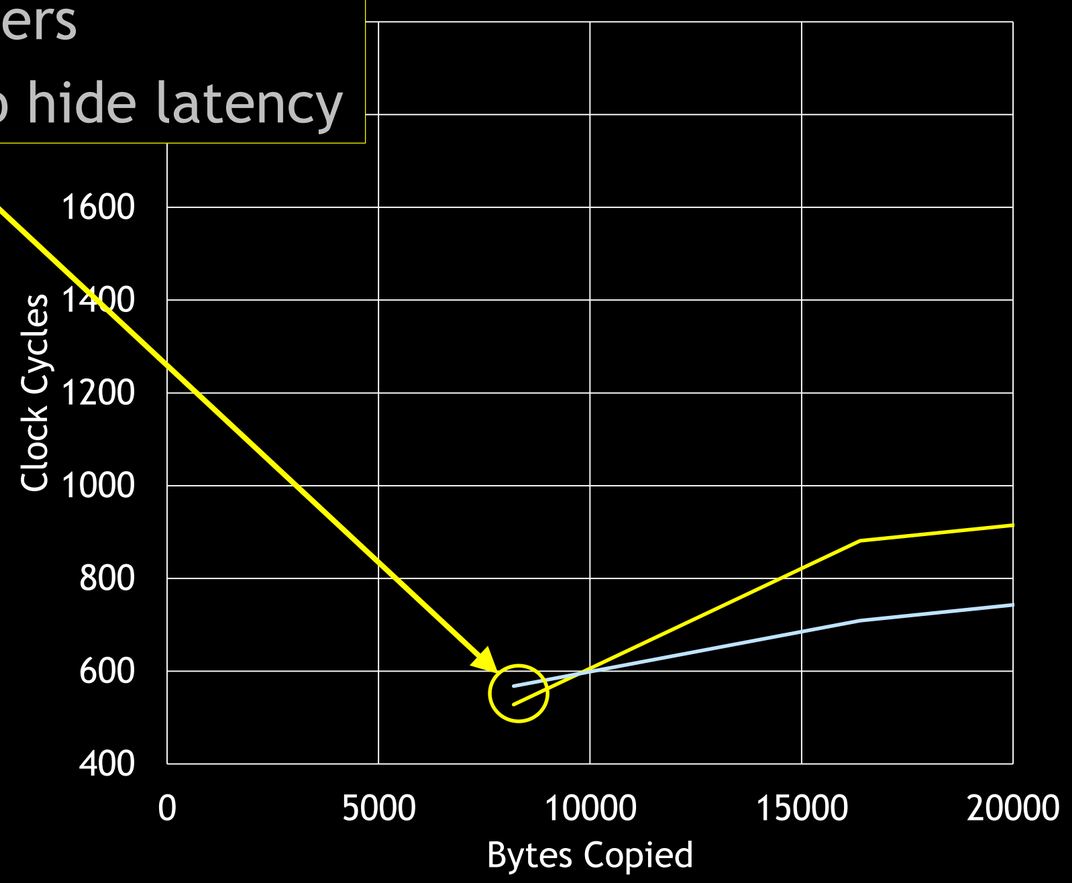


Performance Experiment

(4 of 4) Corner Case when Synchronous Copy might Win

Large number of threads (512+)
Many available registers
Ample opportunity to hide latency





for more Async-Copy examples and performance deep-dive see:

S21819: Optimizing Applications for NVIDIA Ampere GPU Architecture

Thursday May 21 at 10:15am Pacific


Synchronize
Producer → Consumer
and other Algorithms

Built-in Sync-Functions (barriers)

__syncthreads() , __syncwarp() , and siblings

Absolute Best Performing Barrier for

- Synchronizing whole thread block
- Synchronizing a warp, or subset of a warp

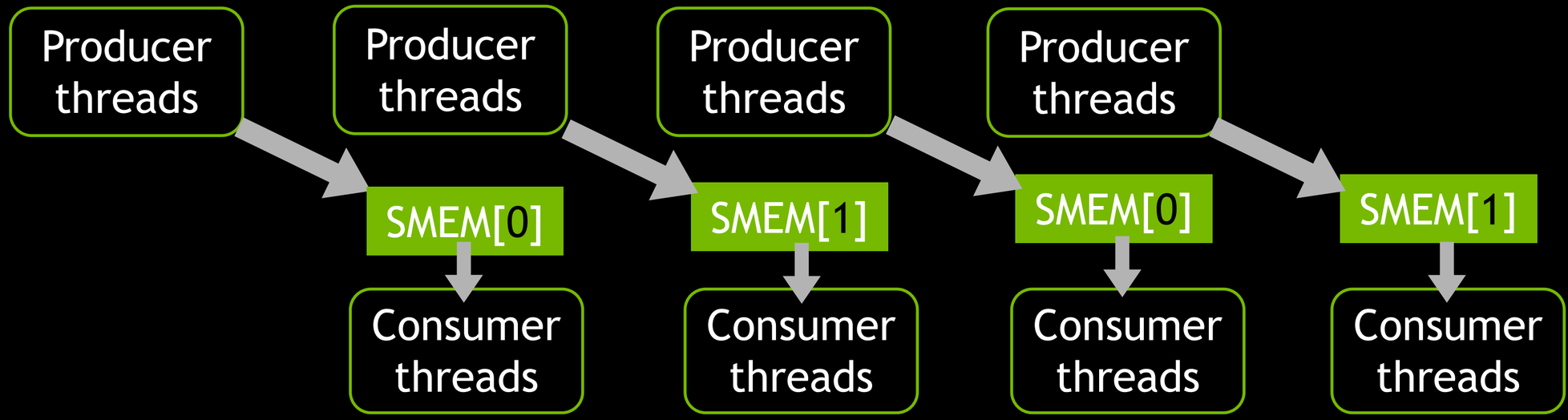
And now CUDA adds barriers to synchronize

- Multi-warp subset of thread block
- Producer → Consumer pattern
- Integrated synchronization of thread execution *and* asynchronous memory copy

CUDA Cooperative Groups also provides `this_grid().sync();`

Split Producer → Consumer Thread Block

a.k.a. Algorithms with Block Partitioning and Specialization



Producer threads and Consumer threads must coordinate

- Consumer must wait until buffer is ready for consumption
- Producer must wait until buffer is available for production

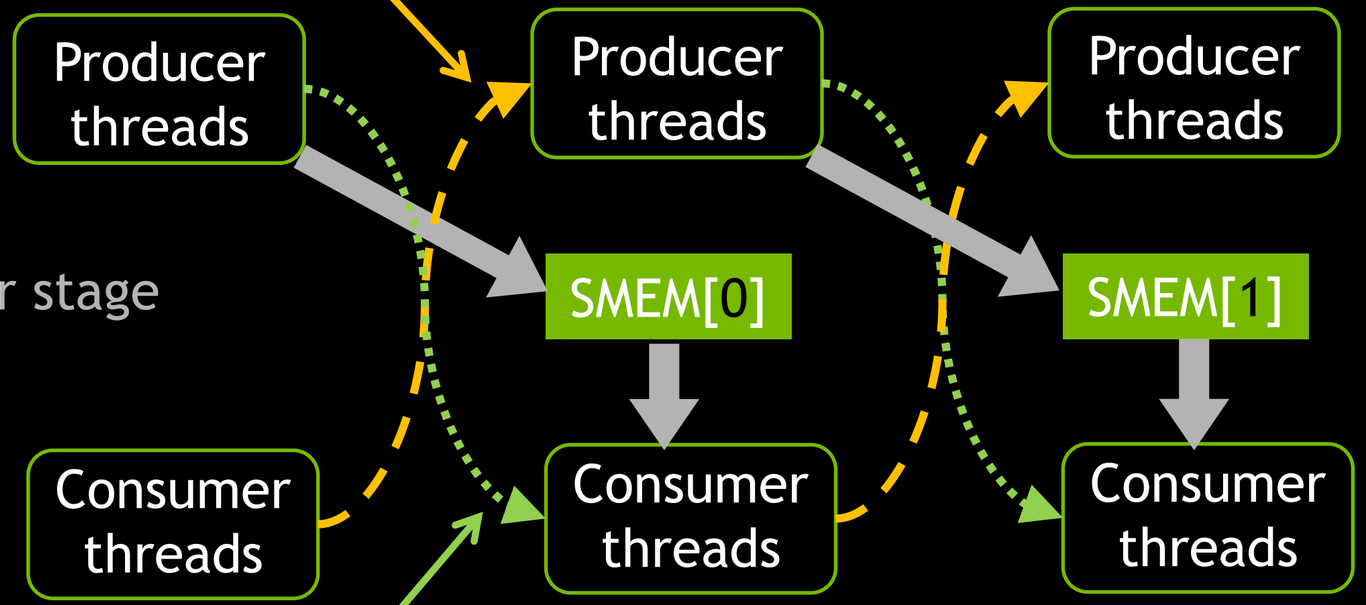
Recommendation: keep warps' threads together

Producer → Consumer Synchronization

Need a New Type of Barrier for “One Sided” Sync

Consumer → Producer Sync : OK to fill SMEM buffer[stage % nStage]

Using two barriers per stage



Producer → Consumer Sync : done filling SMEM buffer[stage % nStage]

Arrive/Wait Barrier

Enabling Producer ←→ Consumer Synchronization

cuda::barrier<...>

- Implementation of ISO/C++ arrive/wait barrier
- Flexibly synchronize arbitrary groups of threads
- This presentation is only considering threads within a CUDA thread block

First: Introduce you to arrive/wait barrier

Then: Show in producer → consumer pattern

Arrive/Wait Barrier

Introductory Example: Replacing __syncthreads()

```
__global__ void kernel()
{


   while ( iterating ) {
      __syncthreads();
   }
}
```

```
__global__ void kernel()
{
   __shared__ cuda::barrier<...> bar;
   /* ... initialize bar ... */
   while ( iterating ) {
      bar.arrive_and_wait();
   }
}
```

Barrier object in shared memory

Note: __syncthreads() has *the* best performance to sync a *whole* thread block

Split Arrive and then Wait

Thread Group Memory Ordering and Visibility

```
bar.arrive_and_wait(){ bar.wait( bar.arrive() ); }
```

A thread's memory updates BEFORE arrive

are visible to thread group AFTER wait

Memory updates BETWEEN arrive and wait

should be local to this thread

Put the BETWEEN time to good use,

otherwise threads may just idle

```
__shared__ int x ;
while ( iterating ) {
    if ( tid == 0 ) x = 42 ;
    /* BEFORE */
    auto token = bar.arrive();
    /* BETWEEN */
    bar.wait(token);
    /* AFTER */
    assert( x == 42 );
}
```

Introduce Async-Copy Operations

Thread Group Memory Ordering & Visibility

Transfer pipeline's wait to the barrier

Threads submit async-copies BEFORE

Pipeline arrives on the barrier

Barrier wait combines pipeline wait and thread synchronization wait.

Copied data visible to thread group AFTER

```
while ( iterating ) {
    memcpy_async(sh[i],gl[j],pipe);
    /* BEFORE */
    pipe.arrive_on(bar);
    auto token = bar.arrive();
    /* BETWEEN */
    bar.wait(token);
    /* AFTER */
    assert( sh[i] == gl[j] );
}
```

Arrive/Wait Barrier Initialization

Enabling Arbitrary Subgroups of a Thread Block

“Bootstrap Initialization”

Barrier object is uninitialized

Choose a thread to initialize

Initialize with number of threads that will arrive and wait (participate)

Synchronize participating threads **before** they use the barrier

```
__global__ void kernel()
{
  __shared__ barrier<...> bar;
  /* ... to initialize bar: */
  if ( tid == 0 ) init( &bar, NumThreads );

  __syncthreads();
  /* barrier is ready for use */
}
```

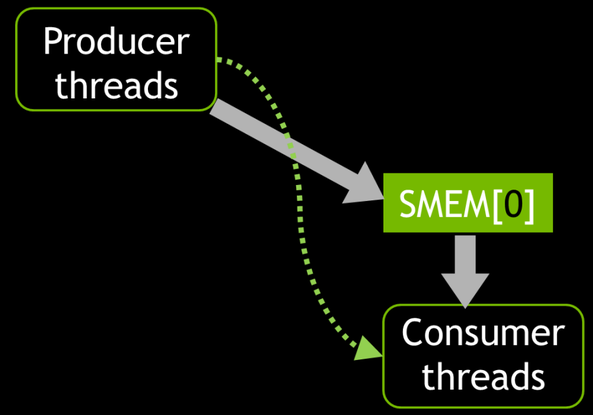
Barrier initialized to synchronize an arbitrary subset of a thread block

Keep threads in a warp together for best performance

Producer → Consumer Async-Copy

Partition Thread Block into Producer and Consumer Subsets

```
__global__ void kernel() {
    __shared__ barrier<...> bar;
    if ( 0 == threadIdx.x ) init(&bar,blockDim.x);
    __syncthreads();
    if ( threadIdx.x < NumProducer ) producer(bar);
    else                             consumer(bar);
```



Consumer Threads

```
/* wait for fill of shared memory buffer */
bar.arrive_and_wait();
/* compute using buffer */
```

Producer Threads

```
/* fill shared memory buffer */
memcpy_async(sh[i],gl[j],pipe);
pipe.arrive_on(bar);
bar.arrive();
```

one-sided synchronization: producer arrives and consumer waits

Producer → Consumer Pattern

We did a Deep-Dive into Some of the Details

Covered

Producer uses async-copy to fill buffer

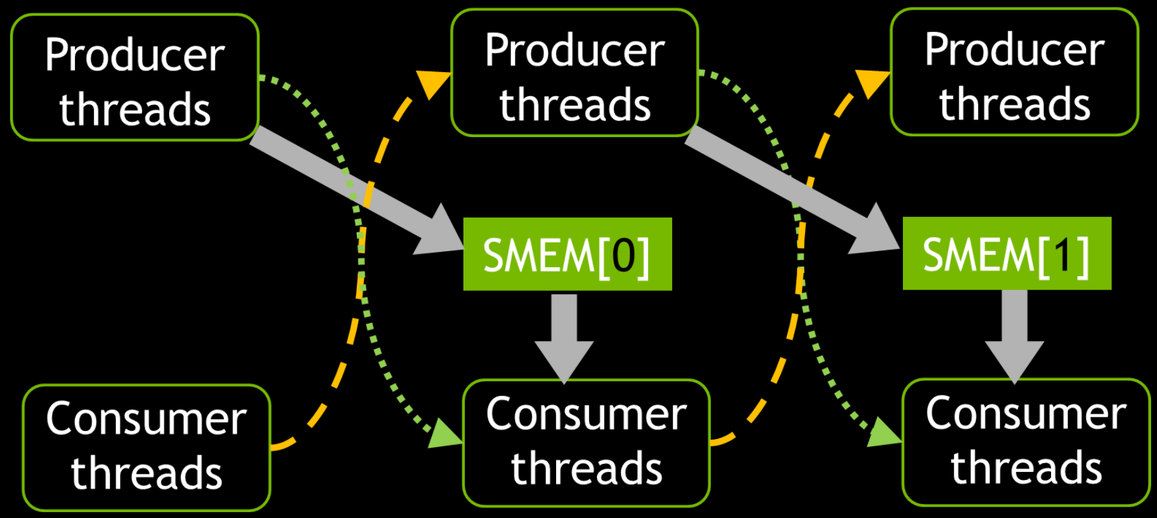
Producer uses barrier for “buffer is filled”

Consumer waits on barrier for “buffer is filled”

Exercise for the Student

Barrier for “OK to fill buffer”

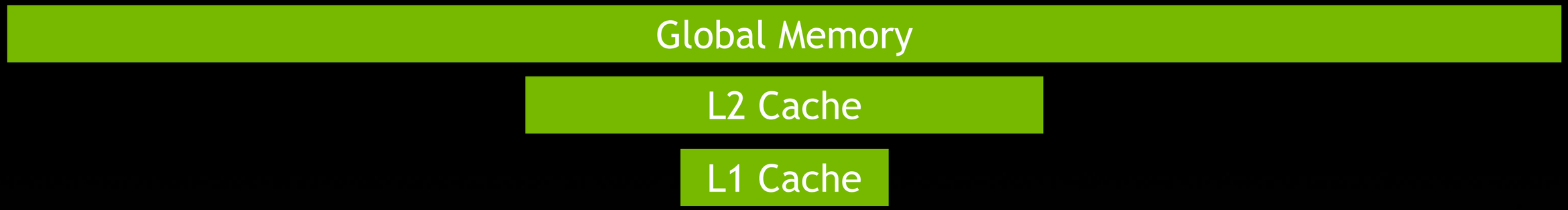
Producer-internal or consumer-internal barrier



Influence Residency of Data in L2 Cache

Cache Memory Hierarchy

Residency of Data in Cache Affects Performance



Algorithms tune for spatial-temporal locality to get cache residency and thus performance

Spatial Locality : Algorithm's nearby threads access nearby global memory

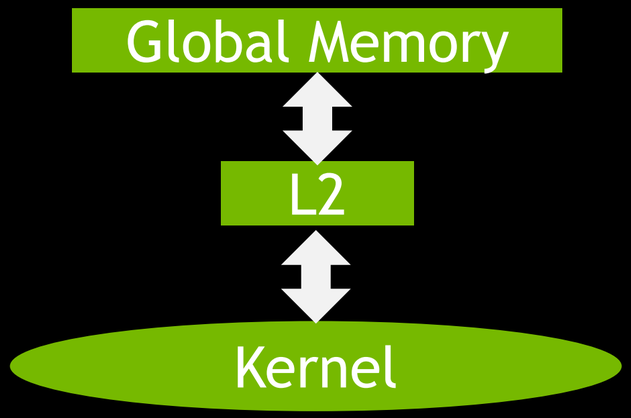
```
array[ threadIx.x ]
```
← adjacent thread accesses adjacent memory

Temporal Locality : Algorithm's nearby instructions access nearby global memory

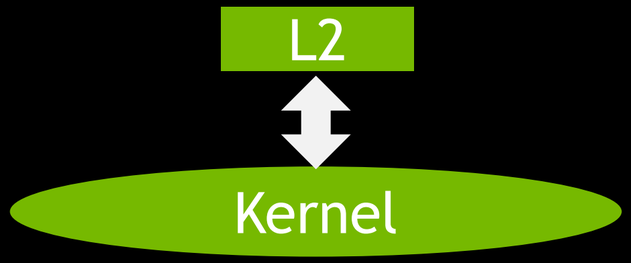
```
array[ i ]
array[ i + 1 ]
```
← **adjacent instruction accesses adjacent memory**

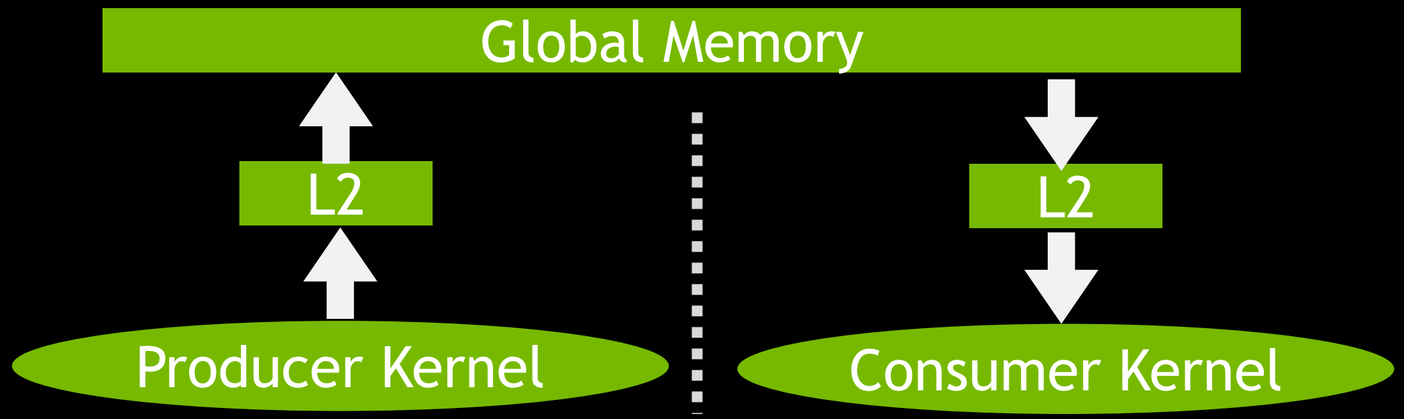
Influence Residency of Data in L2 Cache

Intra-Kernel and Inter-Kernel Performance Benefits

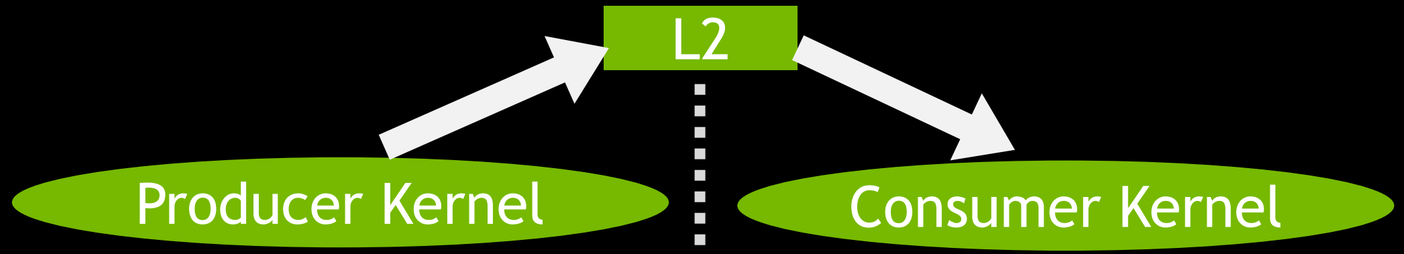


Reduce Intra-Kernel trips to global memory



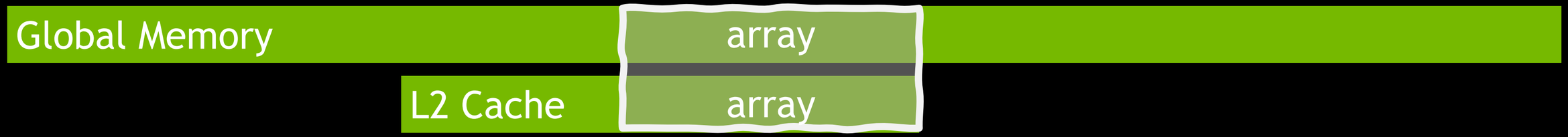


Reduce Producer-Consumer Inter-Kernel trips to global memory



Access Policy to Influence L2 Residency

Select an Array in Global Memory to Persist in L2 Cache

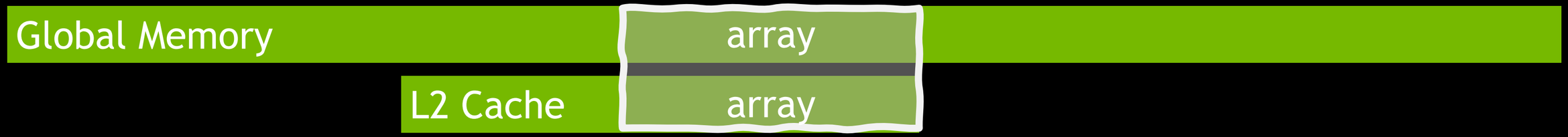



```
cudaStreamAttrValue attr ;

attr.accessPolicyWindow.base_ptr  = /* beginning of array */ ;

attr.accessPolicyWindow.num_bytes = /* number of bytes in array */ ;

attr.accessPolicyWindow.hitProp   = cudaAccessPropertyPersisting;

cudaStreamSetAttribute(stream,cudaStreamAttributeAccessPolicyWindow,&attr);
```

Set on a CUDA stream, applied to subsequent kernels in that stream

Access Policy to Influence L2 Residency

Select an Array in Global Memory to Persist in L2 Cache




```
attr.accessPolicyWindow.hitProp = cudaAccessPropertyPersisting ;
```

L2 Access Policies

- Persisting : accessed memory more likely to persist in L2 cache
- Streaming : accessed memory less likely to persist in L2 cache
- Force to Normal : remove Persisting policy*

*will come back to this

Persistence in L2 Cache

Data Can Persist “Long” After Kernel Exists

Power: Improve intra-kernel and inter-kernel producer→consumer performance

Responsibility: Avoid oversubscription of the persisting L2 cache capacity

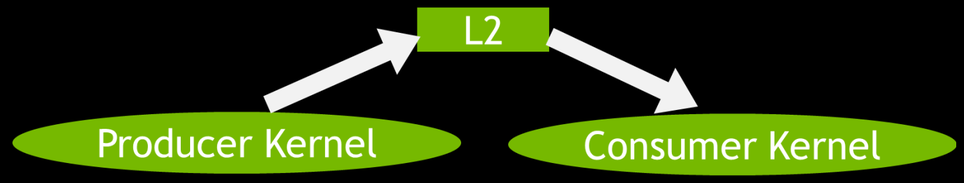
- Concurrently executing kernels only use their fare share of persisting L2 cache
- Clean up when done, don’t let unused data persist in L2 cache

Eventually HW will automatically clean up

Clean Up

Remove Persisting Property When No Longer Needed

1) Consumer kernel cleans up:




```
attr.accessPolicyWindow.base_ptr  = /* beginning of array */ ;
attr.accessPolicyWindow.num_bytes = /* number of bytes in array */ ;
attr.accessPolicyWindow.hitProp   = cudaAccessPropertyNormal ;
cudaStreamSetAttribute(stream,cudaStreamAttributeAccessPolicyWindow,&attr);
consumer_kernel<<<...,stream>>>(...);
```

OR

2) Host cleans up whole L2 cache: `cudaCtxResetPersistingL2Cache();`

Set Aside L2 Cache for Persisting Accesses

Setting persisting set-aside is a device-level operation

```
cudaGetDeviceProperties(&prop,device);

cudaDeviceSetLimit(cudaLimitPersistingL2CacheSize,prop.l2CacheSize*0.75);
```

Interoperability limitations

- Multi-Process Service (MPS), only set at process start via environment variable

```
CUDA_DEVICE_DEFAULT_PERSISTING_L2_CACHE_PERCENTAGE_LIMIT=75   (75%)
```

- Multi-Instance GPU (MIG), disables this feature

Influence L2 Cache Residency Microbenchmark

Microbenchmark Performance Experiment

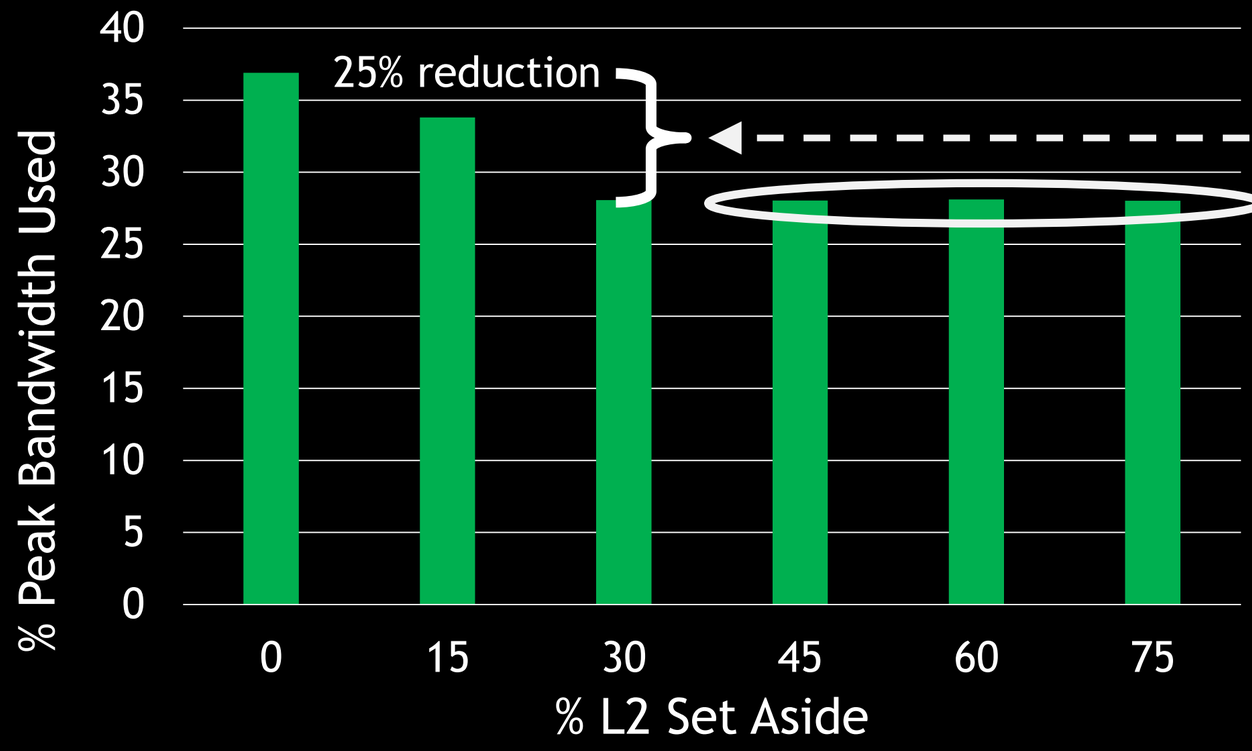
Update Elements of Array in a Random Order

```
while( iter < count ) {
  index = random(...);
  if ( threadId % 2 )
    regular[ index % rLen ] = regular[ index % rLen ] + regular[ index % rLen ];
  else
    persist[ index % pLen ] = persist[ index % pLen ] + persist[ index % pLen ];
}
```

	Persisting global array	Regular global array
Size	0.25 x L2 cache size	4 x L2 cache size
Update	Even id threads update	Odd id threads update

Reducing Global Memory Traffic

Metric: Percentage of Peak Global Bandwidth Utilized



Persisting array is 25% L2 capacity

Fully persists with 30% L2 set-aside

Leftover set-aside is used normally

Your algorithm's mileage will vary

- Identify an array to persist that is more frequently used

for more L2 Residency examples and performance deep-dive see:

S21819: Optimizing Applications for NVIDIA Ampere GPU Architecture

Thursday May 21 at 10:15am Pacific

Warp Synchronous Reduction

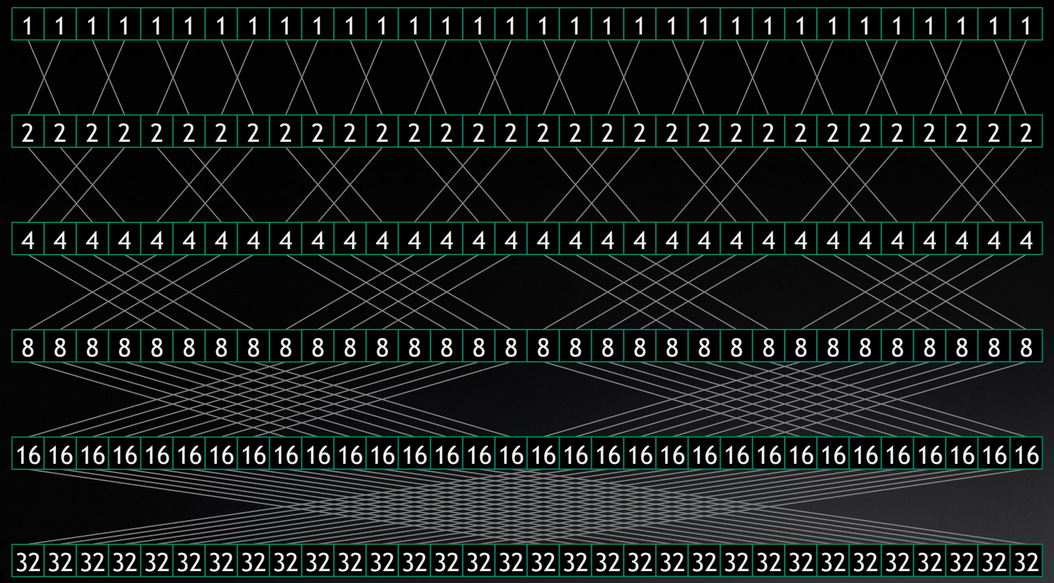
New CUDA Warp Intrinsics

```
int __reduce_op_sync(unsigned mask, int val);
```

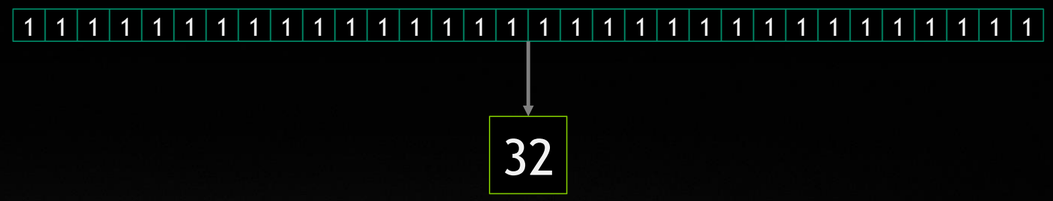
Integer reduce *op* { **add**, **min**, **max** } and bitwise reduce *op* { **and**, **or**, **xor** }

Approximately 10x faster than current best shuffle-based fan-in algorithm

Before: Five Steps of Warp-Shuffle



Now: One HW Accelerated Collective



CUDA Cooperative Group Collective

thread_tile_block and coalesced_group

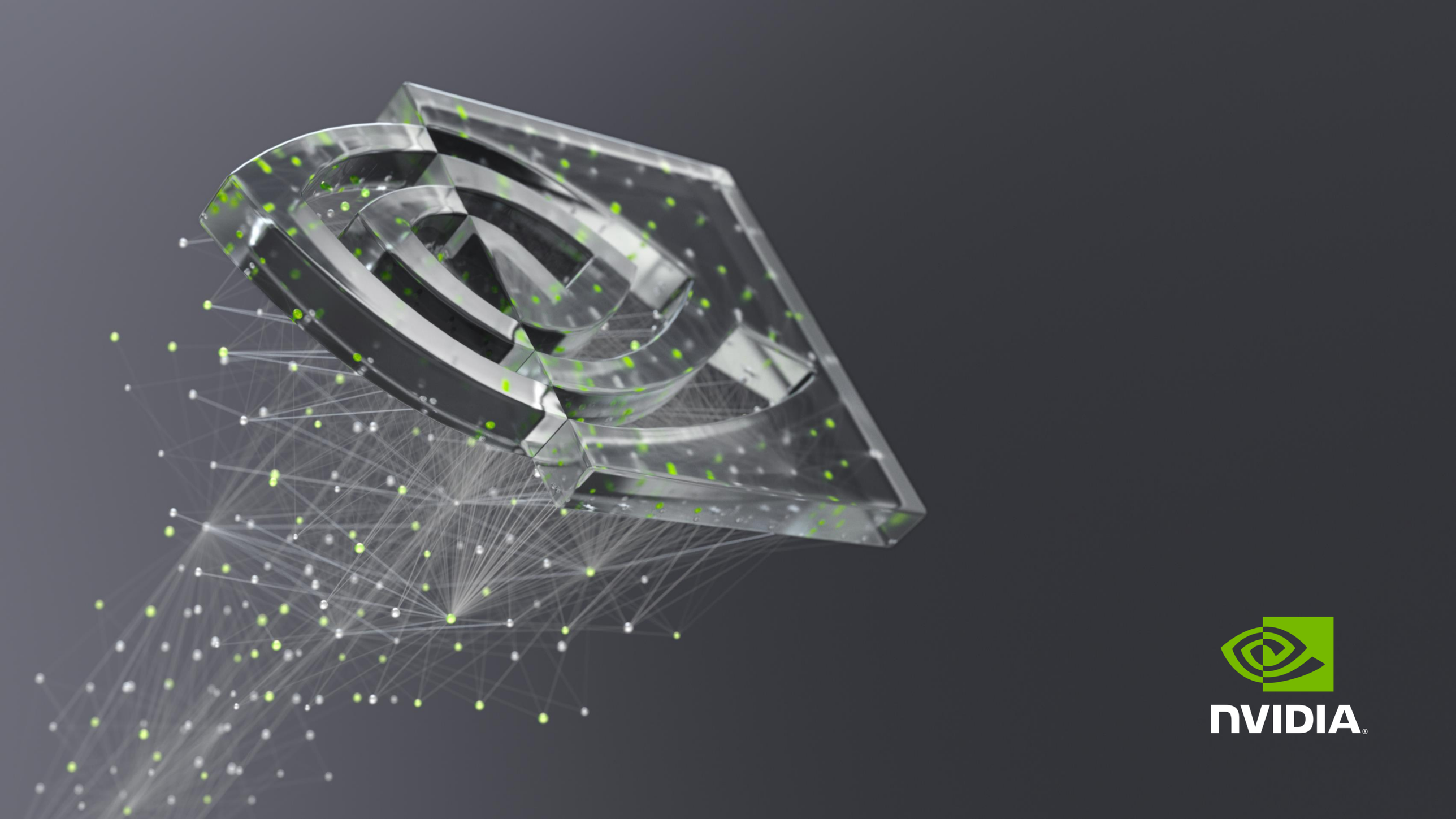
```
value = reduce( group, value, op );
```

group is : **thread_tile_block<N>** or **coalesced_group**

op is C++ : **plus<T>, less<T>, greater<T>, bit_and<T>, bit_or<T>, bit_xor<T>**

When data type **T** is 32bit integer then use new CUDA warp intrinsics

Otherwise use best five step warp-shuffle and apply the operator


NVIDIA