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

```c
__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

\[
\text{anatomy of copy shared} \leftarrow \text{global} \\
\text{shbuf}[i] = \text{gldata}[i];
\]

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

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 \textit{waits} for all submitted asynchronous copy operations to complete

```c
pipeline pipe;
memcpy_async(dst, src, pipe);
pipe.commit_and_wait();
```
Update Implementation with Async-Copy

Begin with Simple One-Stage Pipeline

Before GA100 GPU

```c
__shared__ extern int shbuf[];

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

Now

```c
__shared__ extern int shbuf[];

pipeline pipe;
while ( an_algorithm_iterates ) {
    __syncthreads();
    for ( i = ... ) {
        Shbuf[i] = gldata[i];
    }
    __syncthreads();
    /* compute on shbuf[] */
}
```

Still have the __syncthreads() sandwich
Async-Copy Pipeline
Then Improve with Even Better Journey through Memory

```c
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 \( \text{dst} = \text{src} \);
- 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 \( \{ 0, 1, \ldots, K \} \)

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

```c
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

```c
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_and_wait();
    }
    pipe.commit();
    __syncthreads();
    /* compute on sh[stage % nStage][] */
    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
- stage = next - (nStage-1);
- Prior stage relative to most recent commit

Wait for current stage of copies to complete
- 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 memcmpy_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”
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

Copy 4Byte Data Type
- sync-128
- async-128

Copy 16Byte Data Type

same memory copied in less time

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

**Copy 4Byte Elements**

**Copy 16Byte Elements**
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 ]

Producer threads

SMEM[0]

SMEM[1]

Consumer threads

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

Using two barriers per stage
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()**

<table>
<thead>
<tr>
<th><strong>global</strong> void kernel()</th>
</tr>
</thead>
<tbody>
<tr>
<td>`{</td>
</tr>
<tr>
<td>while ( iterating ) {</td>
</tr>
<tr>
<td>__syncthreads();</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>global</strong> void kernel()</th>
</tr>
</thead>
<tbody>
<tr>
<td>`{</td>
</tr>
<tr>
<td><strong>shared</strong> cuda::barrier&lt;...&gt; bar;</td>
</tr>
<tr>
<td>/* ... initialize bar ... */</td>
</tr>
<tr>
<td>while ( iterating ) {</td>
</tr>
<tr>
<td>bar.arrive_and_wait();</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

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

```c
__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],g1[j],pipe);
    pipe.arrive_on(bar);
    auto token = bar.arrive();
    bar.wait(token);
    assert( sh[i] == g1[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

```c
__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
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 \(\rightarrow\) 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

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

\[
\text{array}[\text{threadIx.x}] \leftrightarrow \text{adjacent thread accesses adjacent memory}
\]

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

\[
\begin{align*}
\text{array} & [i] \\
\text{array} & [i + 1] \leftrightarrow \text{adjacent instruction accesses adjacent memory}
\end{align*}
\]

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

Residency of Data in Cache Affects Performance
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

Global Memory

array

L2 Cache

array

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:

```c
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

L2 Cache: Persisting Normal and Streaming

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

```c
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 ];
}
```

<table>
<thead>
<tr>
<th></th>
<th>Persisting global array</th>
<th>Regular global array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>0.25 x L2 cache size</td>
<td>4 x L2 cache size</td>
</tr>
<tr>
<td>Update</td>
<td>Even id threads update</td>
<td>Odd id threads update</td>
</tr>
</tbody>
</table>
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

\[
\text{value} = \text{reduce( } \text{group, value, op } );
\]

group is: thread_tile_block\langle N \rangle \text{ or coalesced_group}

op is C++: \text{plus\langle T \rangle, less\langle T \rangle, greater\langle T \rangle, bit\_and\langle T \rangle, bit\_or\langle T \rangle, bit\_xor\langle T \rangle}

When data type \( T \) is 32bit integer then use new CUDA warp intrinsics

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