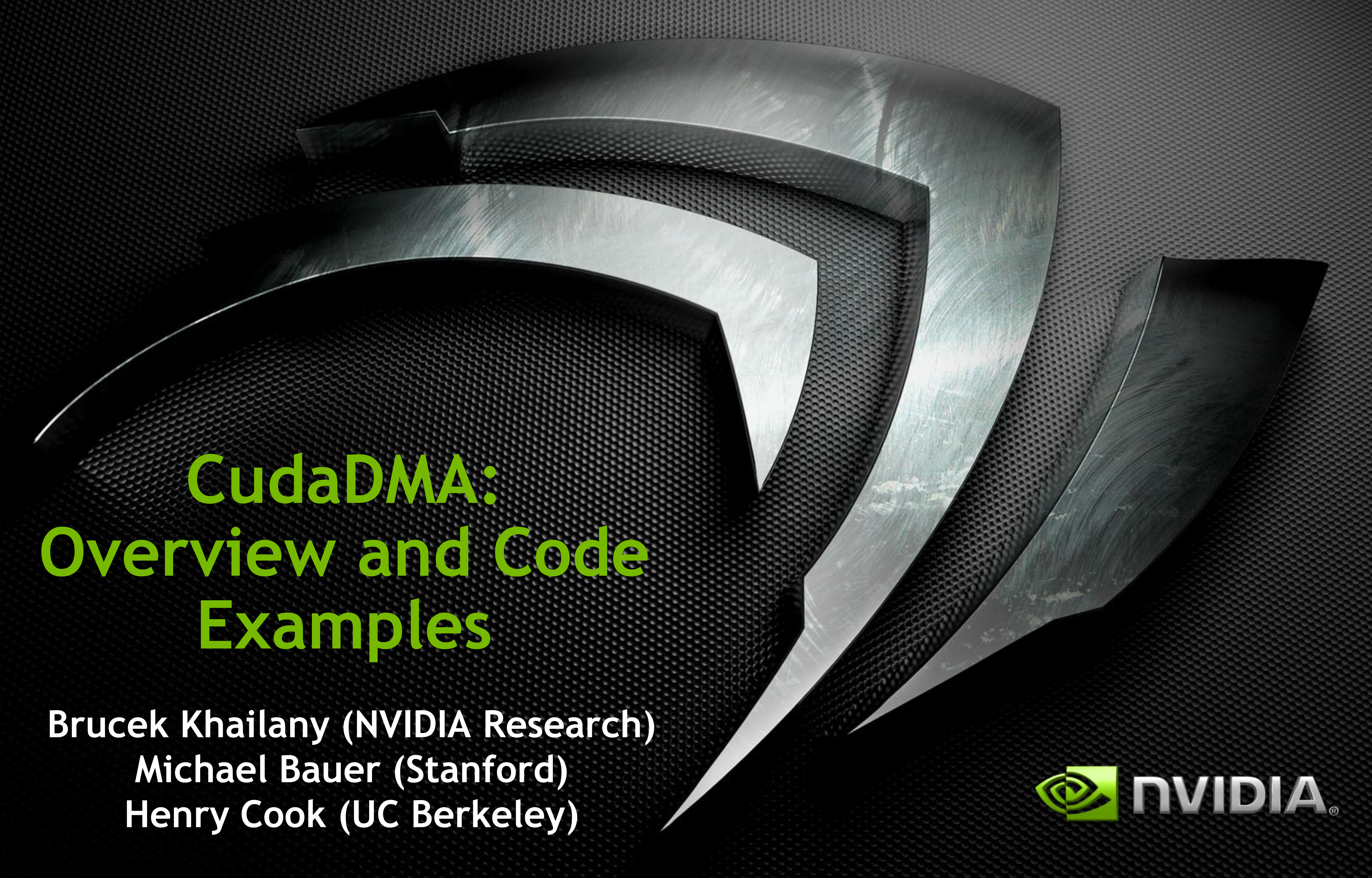


CudaDMA: Overview and Code Examples

Brucek Khailany (NVIDIA Research)
Michael Bauer (Stanford)
Henry Cook (UC Berkeley)

What is cudaDMA?

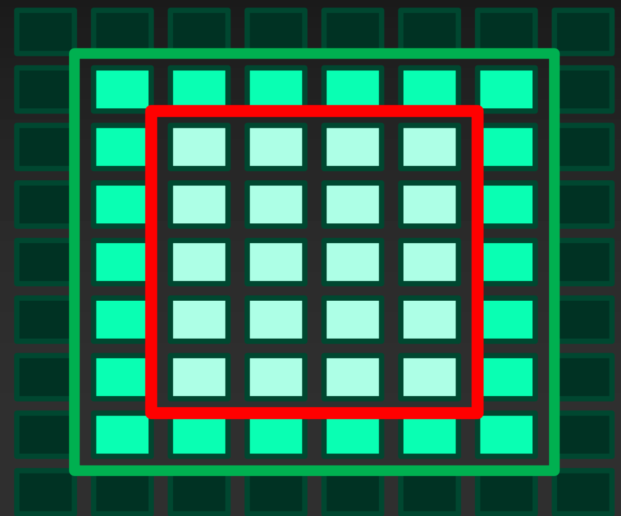
- An API for efficiently copying data from global to shared memory
  - Allows experts to implement optimized tuned code, accessible through a simple API
  - Decouples size & shape of thread block from size & shape of data
  - Works with or without *warp specialization*
- Why did we write this?
  - To research productive high-performance GPU programming techniques
- Where can I find it?
  - http://code.google.com/p/cudadma/

Outline

- Motivation
- CudaDMA API
- Code Examples
- A Few Experimental Results

Motivation 1: Productivity

- **Mismatch thread block size/shape and shared data size/shape**
  - Leads to lots of ‘if’ statements (and headaches)



Goal: decouple thread block size/shape from data size/shape

Example of size/shape mismatch

If/else serialization on data transfers in 3D stencil

```
/////////////////////////////////////////
// update the data slice in smem

s_data[ty][tx]        = local_input1[radius];
s_data[ty][tx+BDIMX] = local_input2[radius];
if( threadIdx.y<radius )   // halo above/below
{
     s_data[threadIdx.y][tx]                  = g_curr[c_offset - radius*dimx];
     s_data[threadIdx.y][tx+BDIMX]            = g_curr[c_offset - radius*dimx + BDIMX];
}


if( threadIdx.y >= radius && threadIdx.y < 2*radius )
{
     s_data[threadIdx.y+BDIMY][tx]            = g_curr[c_offset + (BDIMY-radius)*dimx];
     s_data[threadIdx.y+BDIMY][tx+BDIMX]      = g_curr[c_offset + (BDIMY-radius)*dimx + BDIMX];
}


if(threadIdx.x<radius)     // halo left/right
{
     s_data[ty][threadIdx.x]                  = g_curr[c_offset - radius];
     s_data[ty][threadIdx.x+2*BDIMX+radius] = g_curr[c_offset + 2*BDIMX];
}
__syncthreads();
```

Motivation 2: Performance Bottlenecks

Memory System Bottlenecks

- Instruction Issue
  - Memory Level Parallelism (MLP)
- Data Access Patterns
  - Coalescing

Computational Bottlenecks

- Long-latency memory accesses
- Synchronization overheads
- Data Access Patterns
  - Control Divergence

Goal: remove bottlenecks and entanglement

Warp Specialization

- CudaDMA enables warp specialization:
  - DMA warps
    - Maximize MLP
  - Compute warps
    - No stalls due to memory
- CudaDMA objects manage warp specialization
  - Describe data transfer patterns
  - Independent of warp count

CudaDMA API

CudaDMA API

- Declare CudaDMA object to manage shared buffer
- Separate DMA and compute warps
- Provide synchronization primitives
- Perform repeated transfer operations

```
class cudaDMA
{
public:
  // Base constructor
  __device__ cudaDMA (
    const int dmaID,
    const int num_dma_threads,
    const int num_comp_threads,

  __device__ void execute_dma(
      void *src_ptr, void *dst_ptr);
};
```

CudaDMA App Structure: No Warp Specialization

- Declare shared buffer at kernel scope
- Declare CudaDMA object to manage buffer
- Load buffer using CudaDMA execute_dma_no_sync member function
- Synchronize
- Process buffer
- Synchronize
- Iterate (optional)

```
__global__
void cuda_dma_kernel(float *data)
{
    __shared__ float buffer[NUM_ELMTS];
}
```

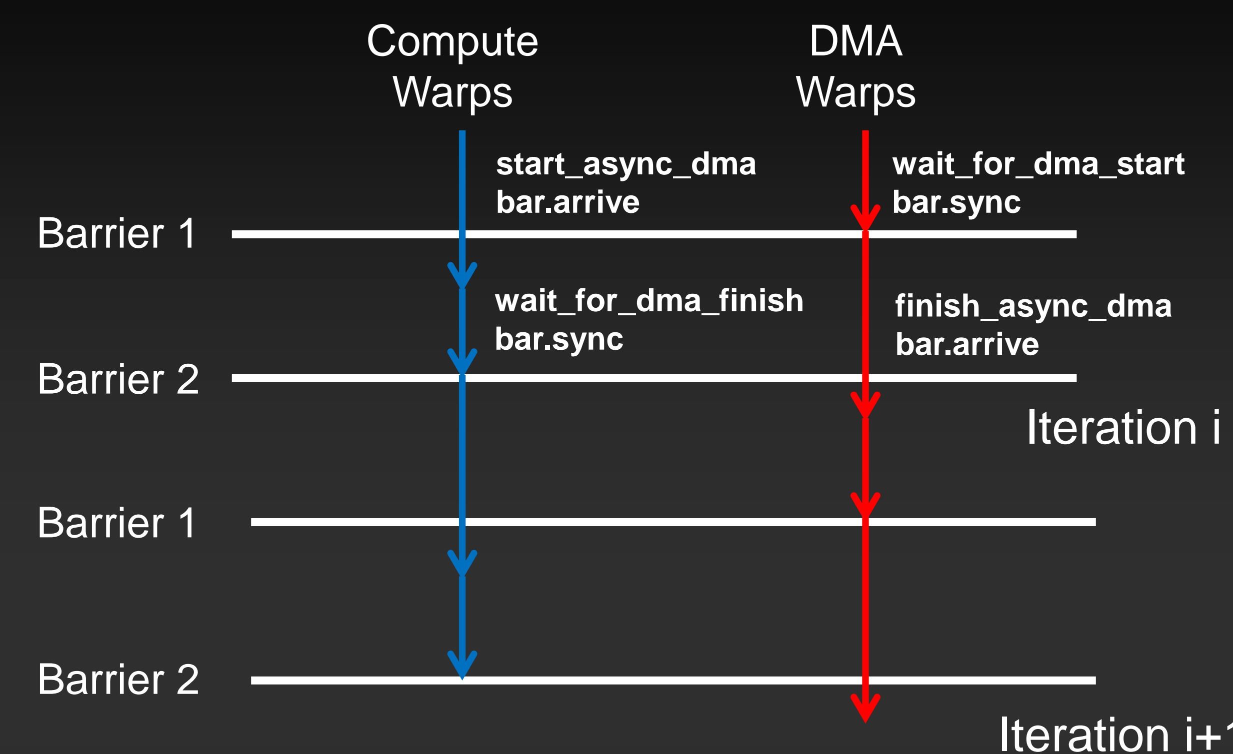
CudaDMA App Structure: Warp Specialization

- Declare shared buffer at kernel scope
- Declare CudaDMA object to manage buffer
- Split DMA warps from compute warps
- Load buffer using DMA warps
- Process buffer using compute warps
- Iterate (optional)

```
__global__
void cuda_dma_kernel(float *data)
{
    __shared__ float buffer[NUM_ELMTS];









}
```

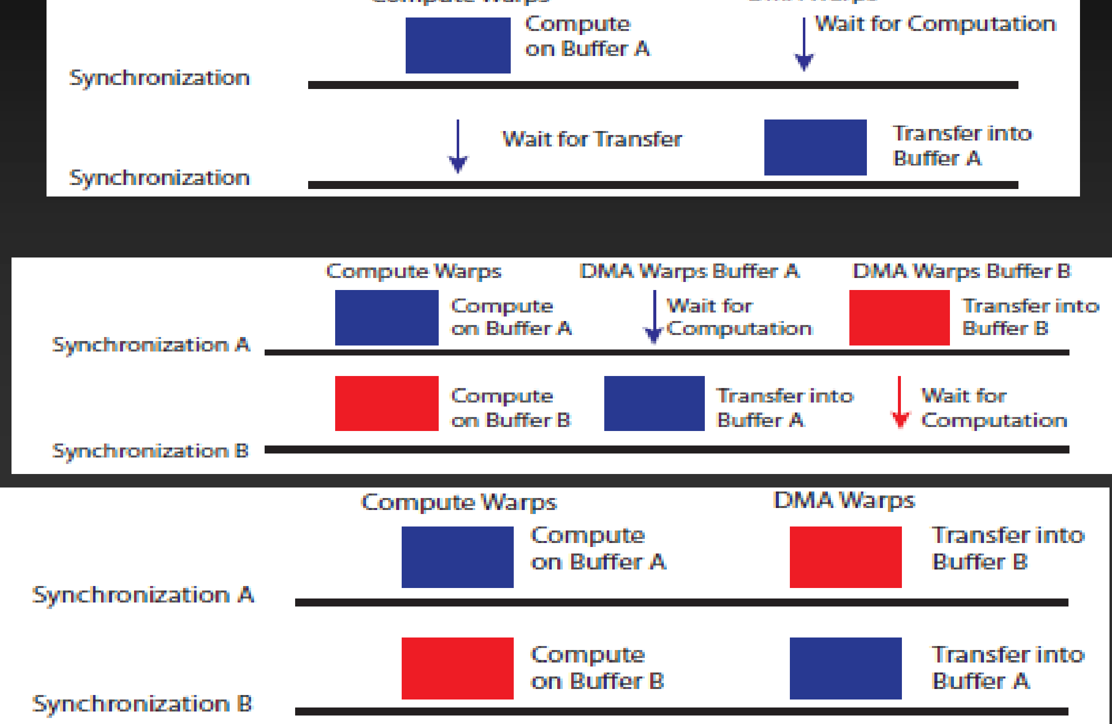
Execution Model With Warp Specialization

- Use PTX named barriers
  - bar.sync
  - bar.arrive
- Fine-grained synchronization



Buffering Techniques

- Usually one set of DMA warps per buffer
- Single-Buffering
  - One buffer, one warp group
- Double-Buffering
  - Two buffers, two warp groups
- Manual Double-Buffering
  - Two buffers, one warp group



Access Patterns

- Explicitly state data loading pattern in code
- Decouple data loaded from number of DMA warps
- State compile-time constants as template variables
- Common patterns implemented by experts
  - Used by application programmers

CudaDMA Instances

- CudaDMASequential
- CudaDMAStrided
- CudaDMAIndirect
  - Arbitrary accesses
- CudaDMAHalo
  - 2D halo regions
- CudaDMACustom

Code Examples

Code Example: SAXPY

Baseline: Typical staging through shared

```
__global__ void saxpy ( float* y, float* x, float a)  {

  volatile __shared__ float sdata_x0 [COMPUTE_THREADS_PER_CTA];
  volatile __shared__ float sdata_y0 [COMPUTE_THREADS_PER_CTA];
  int tid = threadIdx.x ;

  for (int i=0; i < NUM_ITERS; ++i) {
     unsigned int idx = i * COMPUTE_THREADS_PER_CTA * CTA_COUNT +
                blockIdx.x * COMPUTE_THREADS_PER_CTA + tid;
     __syncthreads();
     sdata_x0[tid] = x[idx];
     sdata_y0[tid] = y[idx];
     __syncthreads();
     y[idx] = a * sdata_x0[tid] + sdata_y0[tid];
  }
}
```

Code Example: SAXPY

Example using CudaDMA (no warp specialization)

```
__global__ void saxpy ( float* y, float* x, float a)  {

  volatile __shared__ float sdata_x0 [COMPUTE_THREADS_PER_CTA];
  volatile __shared__ float sdata_y0 [COMPUTE_THREADS_PER_CTA];
  int tid = threadIdx.x ;

  cudaDMASequential<16, 4*COMPUTE_THREADS_PER_CTA, COMPUTE_THREADS_PER_CTA>
    dma_ld_x (0, COMPUTE_THREADS_PER_CTA, 0 );
  cudaDMASequential<16, 4*COMPUTE_THREADS_PER_CTA, COMPUTE_THREADS_PER_CTA>
    dma_ld_y (0, COMPUTE_THREADS_PER_CTA, 0 );

  for (int i=0; i < NUM_ITERS; ++i) {
     unsigned int idx = i * COMPUTE_THREADS_PER_CTA * CTA_COUNT +
                  blockIdx.x * COMPUTE_THREADS_PER_CTA;
     __syncthreads();
     dma_ld_x.execute_dma_no_sync(x[idx],sdata_x0);
     dma_ld_y.execute_dma_no_sync(y[idx],sdata_y0);
     __syncthreads();
     y[idx] = a * sdata_x0[tid] + sdata_y0[tid];
  }
}
```

Code Example: SGEMV (with warp specialization)

- BLAS2: matrix-vector multiplication
- Two Instances of CudaDMA objects
- Compute Warps
- Vector DMA Warps
- Matrix DMA Warps

```
__global__ void sgemv_cuda_dma(int n, int m, int n1, float alpha,
                               float *A, float *x, float *y)  {

  __shared__ float buff[VEC_ELMTS];
  __shared__ float mat[VEC_ELMTS][COMPUTE_THREADS]

  cudaDMASequential<16,4*VEC_ELMTS,DMA_THREADS>
    dma_ld_0(1,COMPUTE_THREADS,COMPUTE_THREADS);
  cudaDMAStrided<16,4*COMPUTE_THREADS,4*DMA_THREADS,VEC_ELMTS>
    dma_ld_1(2,COMPUTE_THREADS,COMPUTE_THREADS+1*DMA_THREADS,4*n);
  if (threadIdx.x < COMPUTE_THREADS_PER_CTA) {
    dma_ld_0.start_async_dma();
    dma_ld_1.start_async_dma();
    float res = 0.f;
    for(int i=0; i<n1; i += VEC_ELMTS) {
      dma_ld_0.wait_for_dma_finish();
      dma_ld_1.wait_for_dma_finish();
      for(int j=0; j < VEC_ELMTS; j++) {
        res+=mat[j][threadIdx.x]*buff[j];
      }
      dma_ld_0.start_async_dma();
      dma_ld_1.start_async_dma();
    }
    int ind = blockIdx.x*COMPUTE_THREADS_PER_CTA + threadIdx.x;
    if (ind<n) y[ind] = alpha * res;
  } else if (dma_ld_0.owns_this_thread()) {
    for (int idx=0; idx<n1; idx += VEC_ELMTS){
      dma_ld_0.execute_dma(x,buff);
      x += VEC_ELMTS;
    }
    dma_ld_0.wait_for_dma_start();
  } else if (dma_ld_1.owns_this_thread()) {
    for (int idx=0; idx<n1; idx += VEC_ELMTS) {
      dma_ld_1.execute_dma(A+blockIdx.x*num_threads+n*idx, mat);
    }
    dma_ld_1.wait_for_dma_start();
  }
}
```

Synchronization Points

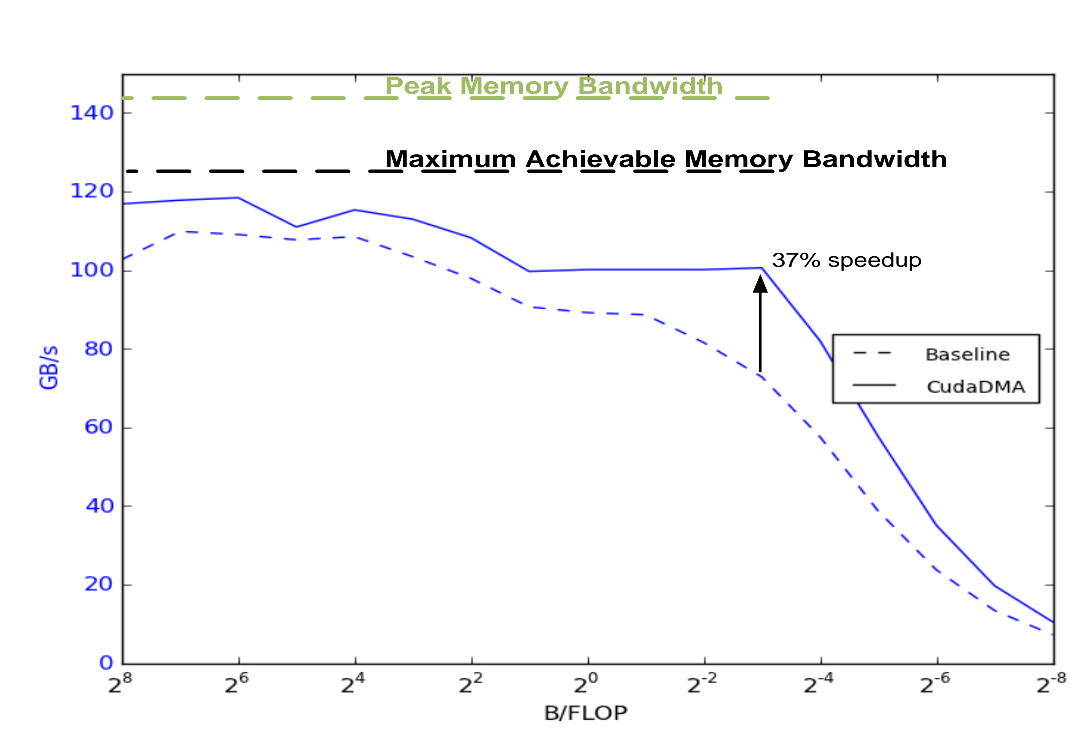
- Compute Warps
  - start_async_dma()
  - wait_for_dma_finish()
- DMA Warps
  - execute_dma()
    - Includes implicit wait_for_dma_start() and finish_async_dma()
  - wait_for_dma_start()

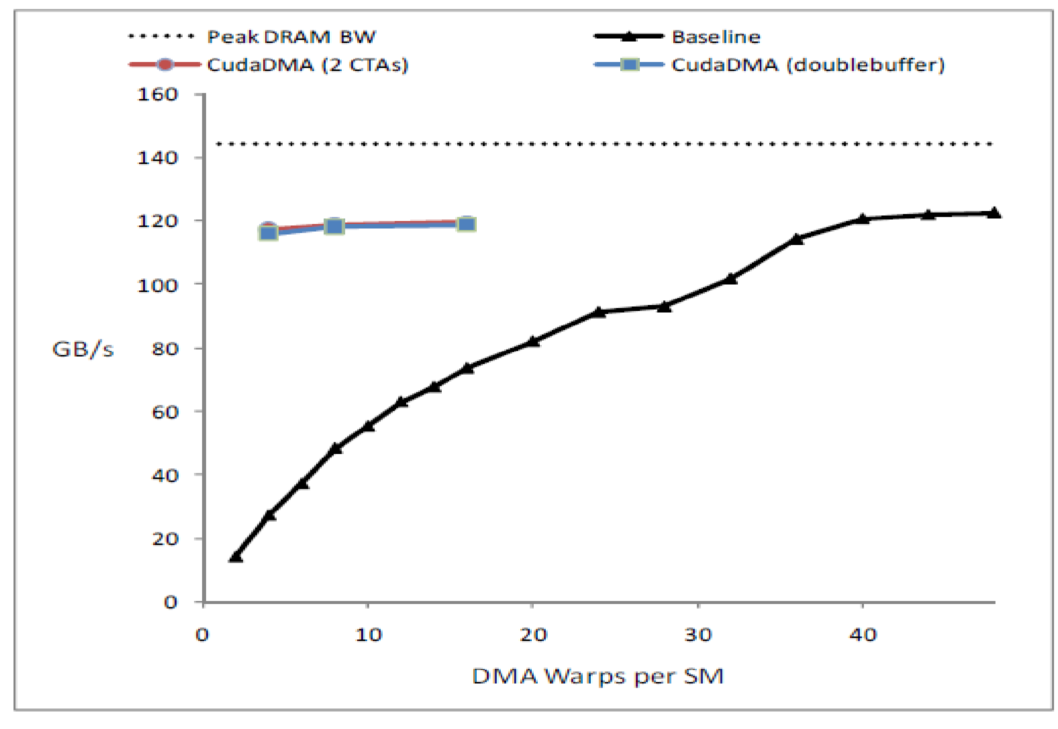
```
__global__ void sgemv_cuda_dma(int n, int m, int n1, float alpha,
                               float *A, float *x, float *y)  {

  __shared__ float buff[VEC_ELMTS];
  __shared__ float mat[VEC_ELMTS][COMPUTE_THREADS]

  cudaDMASequential<16,4*VEC_ELMTS,DMA_THREADS>
    dma_ld_0(1,COMPUTE_THREADS,COMPUTE_THREADS);
  cudaDMAStrided<16,4*COMPUTE_THREADS,4*DMA_THREADS,VEC_ELMTS>
    dma_ld_1(2,COMPUTE_THREADS,COMPUTE_THREADS+1*DMA_THREADS,4*n);
  if (threadIdx.x < COMPUTE_THREADS_PER_CTA) {
    dma_ld_0.start_async_dma();
    dma_ld_1.start_async_dma();
    float res = 0.f;
    for(int i=0; i<n1; i += VEC_ELMTS) {
      dma_ld_0.wait_for_dma_finish();
      dma_ld_1.wait_for_dma_finish();
      for(int j=0; j < VEC_ELMTS; j++) {
        res+=mat[j][threadIdx.x]*buff[j];
      }
      dma_ld_0.start_async_dma();
      dma_ld_1.start_async_dma();
    }
    int ind = blockIdx.x*COMPUTE_THREADS_PER_CTA + threadIdx.x;
    if (ind<n) y[ind] = alpha * res;

  } else if (dma_ld_0.owns_this_thread()) {
    for (int idx=0; idx<n1; idx += VEC_ELMTS){
      dma_ld_0.execute_dma(x,buff);
      x += VEC_ELMTS;
    }
    dma_ld_0.wait_for_dma_start();

  } else if (dma_ld_1.owns_this_thread()) {
    for (int idx=0; idx<n1; idx += VEC_ELMTS) {
      dma_ld_1.execute_dma(A+blockIdx.x*num_threads+n*idx, mat);
    }
    dma_ld_1.wait_for_dma_start();
  }
}
```

A few experimental results…

Micro-Benchmarks

- “SAXPY” using staging through shared memory
- Fix compute intensity (6 B/FLOP), vary warp count





3D Finite-Difference Stencil

- Use DMA warps for loading halo cells as well as main block cells
- Speedups from 13-15%
- Improvement from more MLP and fewer load instructions

Kernel	Time (ms)	Throughput (Mpoints/s)	Bandwidth (GB/s)	Speedup
Reference	27.83	4746.6	76.85	1.00
halo-only-single	26.38	5007.6	81.08	1.055
halo-only-double	31.66	4173.8	67.58	0.879
halo-only-manual	26.12	5055.4	81.85	1.065
block-halo-single	24.16	5467.6	88.53	1.152

Table 1: 3D Stencil: 512x512x512

Kernel	Time (ms)	Throughput (Mpoints/s)	Bandwidth (GB/s)	Speedup
Reference	33.14	4845.0	78.41	1.00
halo-only-single	30.97	5185.1	83.92	1.058
halo-only-double	37.37	4296.2	69.53	0.887
halo-only-manual	31.33	5125.1	82.95	1.058
block-halo-single	29.10	5517.7	89.30	1.139

Table 2: 3D Stencil: 640x640x400

Kernel	Time (ms)	Throughput (Mpoints/s)	Bandwidth (GB/s)	Speedup
Reference	25.22	4872.2	79.18	1.000
halo-only-single	23.74	5176.5	84.12	1.062
halo-only-double	28.71	4280.0	69.55	0.878
halo-only-manual	24.20	5078.7	82.53	1.042
block-halo-single	22.30	5509.4	89.53	1.131

Table 3: 3D Stencil: 800x800x200

Summary

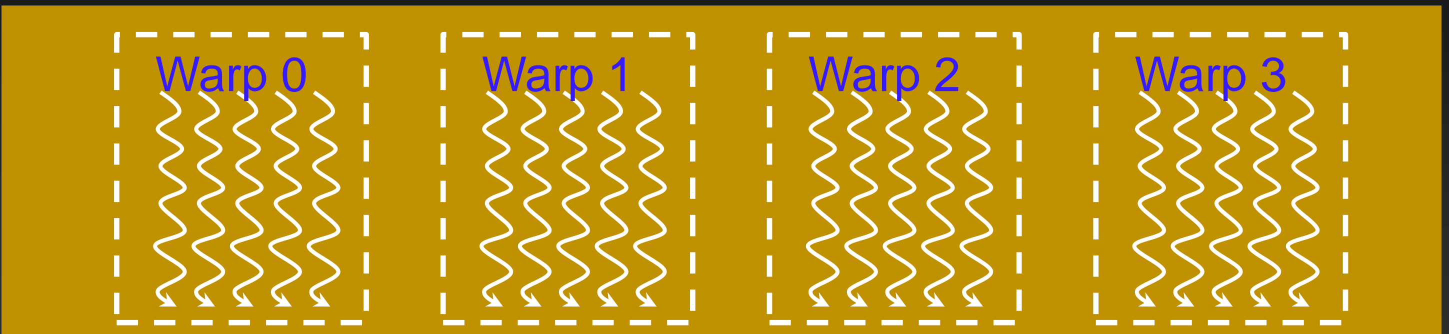
- CudaDMA
  - A library for efficiently copying data from off-chip DRAM to on-chip shared memory on GPUs
- CudaDMA Features
  - Optimized instances for common patterns
    - CudaDMASequential, CudaDMAStrided
    - CudaDMAIndirect, CudaDMAHalo
  - Extensible API
  - Provides framework for warp specialization
- Key Results
  - Easily handles size/shape mismatches without programmer pain
  - Speedups on micro-benchmarks and applications

Thank You!
http://code.google.com/p/cudadma/
NVIDIA

Backup Slides

Warp Defintion

- Each CTA is decomposed into warps
  - A warp is 32 contiguous threads in the same CTA



- SM scheduler performs scheduling at warp-granularity
  - Each warp has its own program counter
  - All threads in a warp execute in lock-step
  - Intra-warp divergence has performance penalty
  - Inter-warp divergence has no performance penalty