CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization

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

- GPUs are in many supercomputers today

- GPUs are great
  - High floating point performance
  - High memory bandwidth

- Why is programming them so challenging?
  - Explicit data movement through memory hierarchy
  - Difficult to overlap computation and memory accesses
Outline

- Overview of GPU Architecture
- Motivating Benchmark
- CudaDMA API
- Methodology
- Experiments
- Conclusions
GPU Architecture/Programming

On-Chip Memory

Off-Chip DRAM
Warp Definition

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

- SM 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
Motivating Benchmark
Motivating Benchmark

- Modified SAXPY kernel, staging data through shared
  - Variable amount of arithmetic
  - Fixed amount of data transferred and number of warps
## GPU Performance Challenges

<table>
<thead>
<tr>
<th>Memory System Bottlenecks</th>
<th>Computational Bottlenecks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Issue</td>
<td>Long-latency memory accesses</td>
</tr>
<tr>
<td>- Memory Level Parallelism (MLP)</td>
<td></td>
</tr>
<tr>
<td>Data Access Patterns</td>
<td>Synchronization overheads</td>
</tr>
<tr>
<td>- Coalescing</td>
<td>Data Access Patterns</td>
</tr>
<tr>
<td></td>
<td>- Control Divergence</td>
</tr>
</tbody>
</table>

**Goal:** remove entanglement between the bottlenecks
GPU Programmability Challenges

- Mismatch CTA size/shape and shared data size/shape
  - Leads to thread divergence (lots of ‘if’ statements)

Goal: decouple CTA size/shape from data size/shape
Warp Specialization

- Differentiate warps into compute and DMA*
  - DMA warps
    - Maximize MLP
  - Compute warps
    - No stalls due to memory

- Producer-consumer synchronization
  - Enable better overlapping of compute and memory accesses

- CudaDMA objects to 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

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

    __device__ bool owns_this_thread();

    // Compute thread sync functions
    __device__ void start_async_dma();
    __device__ void wait_for_dma_finish();

    // DMA thread sync functions
    __device__ void wait_for_dma_start();
    __device__ void finish_async_dma();

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

- 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)

```c
__global__
void cuda_dma_kernel(float *data)
{
    __shared__ float buffer[NUM_ELMTS];
    cudaDMA dma_ld(0, NUM_DMA_THRS, NUM_COMPUTE_THRS, NUM_COMPUTE_THRS);

    if (dma_ld.owns_this_thread()) {
        // DMA warps
        for (int i=0; i<NUM_ITERS; i++) {
            dma_ld.wait_for_dma_start();
            dma_ld.execute_dma(data, buffer);
            dma_ld.finish_async_dma();
        }
    }
    else { // Compute warps
        for (int i=0; i<NUM_ITERS; i++) {
            dma_ld.start_async_dma();
            dma_ld.wait_for_dma_finish();
            process_buffer(buffer);
        }
    }
}
```
Execution Model

- Use PTX named barriers
  - bar.sync
  - bar.arrive
- Available on Fermi

- Fine-grained synchronization

![Diagram showing the execution model with Compute Warps and DMA Warps with named barriers and synchronization points.]

Iteration i

Iteration i+1
CudaDMA Methodology
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
CudaDMA Instances

- CudaDMASequential

- CudaDMAStrided

- CudaDMAIndirect
  - Arbitrary accesses

- CudaDMAHalo
  - 2D halo regions

- CudaDMACustom
Access Patterns

- Explicitly state data loading pattern in code
- Decouple implementation from transfer pattern
- Common patterns implemented by experts
  - Used by application programmers
- Optimized for high memory bandwidth at low warp count
Experiments
Micro-Benchmarks

- Same modified SAXPY kernel shown earlier
- Fix compute intensity (6 B/FLOP), vary warp count
BLAS2: SGEMV

- Dense matrix-vector multiplication
- CudaDMASequential for loading vector elements
- CudaDMAStrided for loading matrix elements
- Varied buffering schemes
- Up to 3.2x speedup
3D Finite Difference Stencil

- 8th order in space, 1st order in time computation
- Load 2D slices into shared for each step in Z-dimension
- Loading halo cells uses uncoalesced accesses
  - Earlier version of cudaDMAHalo

Figures from: P. Micikevicius. 3D Finite Difference Computation on GPUs Using CUDA.
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.
Conclusions

- CudaDMA
  - Extensible API
  - Create specialized DMA Warps
  - Works best for moderate compute intensity applications
  - Decouple transfer pattern from implementation
- Optimized instances for common patterns
  - CudaDMAStrided, CudaDMASequential
  - CudaDMAHalo, CudaDMAIndirect
- Speedups on micro-benchmarks and applications
Download CudaDMA:
http://code.google.com/p/cudadma

Tech Talk at NVIDIA Booth on Thursday at 1pm

Questions?
Backup Slides
Asynchronous DMA Engines

- Decouple transfer implementation from specification
  - Asynchronous to overlap computation and memory access

- Ironman abstraction for ZPL (software)

- Sequoia runtime interface (software)

- Cell Broadband Engine (hardware)

- Imagine Stream Processor (hardware)
Code Example: SGEMV

- BLAS2: matrix-vector multiplication

- Two Instances of CudaDMA objects

- Compute Warps

- Vector DMA Warps

- Matrix DMA Warps

```c
__global__ void sgemv_cuda_dma(int n, int m, float alpha, float *A,
float *x, float *y) {
__shared__ float buff[VEC_ELMTS];
__shared__ float mat [VEC_ELMTS][COMPUTE_THREADS];

cudaDMASequential<sizeof(float)*VEC_ELMTS/DMAS_THREADS_SEQ>
dma ld_0(1, DMA_THREADS_SEQ, COMPUTE_THREADS,
COMPUTE_THREADS, sizeof(float)*VEC_ELMTS);

cudaDMAStrided<sizeof(float)*VEC_ELMTS*
COMPUTE_THREADS/DMAS_THREADS_STRD>
dma ld_1(2, DMA_THREADS_STRD, COMPUTE_THREADS,
COMPUTE_THREADS*DMAS_THREADS_SEQ,
sizeof(float)*COMPUTE_THREADS,
VEC_ELMTS, sizeof(float)*n,
sizeof(float)*COMPUTE_THREADS);

if (threadIdx.x < COMPUTE_THREADS) {
  dma ld_0.start_async_dma();
  dma ld_1.start_async_dma();
  float res = 0.f;
  for(int i=0; i<n; 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();
  }
  ind = blockIdx.x*COMPUTE_THREADS+threadIdx.x;
  if (ind < n) y[ind] = alpha * res;
}
else if (dma ld_0.owns_this_thread()) {
  dma ld_0.wait_for_dma_start();
  for (int idx=0; idx<n; idx += VEC_ELMTS) {
    dma ld_0.execute_dma(x, buff);
    dma ld_0.finish_async_dma();
    dma ld_0.wait_for_dma_start();
    x += VEC_ELMTS;
  }
}
else if (dma ld_1.owns_this_thread()) {
  dma ld_1.wait_for_dma_start();
  for (int idx=0; idx<n; idx += VEC_ELMTS) {
    dma ld_1.execute_dma(
        A+idx*m+blockIdx.x*COMPUTE_THREADS, mat);
    dma ld_1.finish_async_dma();
    dma ld_1.wait_for_dma_start();
  }
}
```
Synchronization Points

- **Compute Warps**
  - `start_async_dma()`
  - `wait_for_dma_finish()`

- **DMA Warps**
  - `wait_for_dma_start()`
  - `finish_async_dma()`

```c
__global__ void
gemv_cudama(int n, int m, float alpha, float *A,
  float *x, float *y) {
  __shared__ float buff[VEC_ELMTS];
  __shared__ float mat [VEC_ELMTS][COMPUTE_THREADS];

cudAMASequential<sizeof(float)*VEC_ELMTS/DMA_THREADS_SEQ>
dma_ld_0( 1, DMA_THREADS_SEQ, COMPUTE_THREADS, 
  DMA_THREADS_SEQ, COMPUTE_THREADS, sizeof(float)*n, 
  sizeof(float)*VEC_ELMTS);

cudAMAStrided<sizeof(float)*VEC_ELMTS*
  COMPUTE_THREADS/DMA_THREADS_STRD>
dma ld_1( 2, DMA_THREADS_STRD, COMPUTE_THREADS, 
  DMA_THREADS_SEQ, COMPUTE_THREADS, sizeof(float)*VEC_ELMTS, 
  sizeof(float)*n, 
  sizeof(float)*DMA_THREADS_SEQ);

if (threadIdx.x < COMPUTE_THREADS) {
  dma ld_0.start_async_dma();
  dma ld_1.start_async_dma();

  float res = 0.f;
  for(int i=0; i<n; 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();
}

ind = blockIdx.x*COMPUTE_THREADS+threadIdx.x;
if (ind < n) y[ind] = alpha * res;

else if (dma ld 0.owns_this_thread()) {
  dma ld 0.wait_for_dma_start();
  for (int idx=0; idx<n; idx += VEC_ELMTS) {
    dma ld 0.execute_dma(x, buff);
    dma ld 0.finish_async_dma();
    dma ld 0.wait_for_dma_start();
    x += VEC_ELMTS;
  }
}
else if (dma ld 1.owns_this_thread()) {
  dma ld 1.wait_for_dma_start();
  for (int idx=0; idx<n; idx += VEC_ELMTS) {
    dma ld 1.execute_dma(
        A+idx+m*blockIdx.x*COMPUTE_THREADS, mat);
    dma ld 1.finish_async_dma();
    dma ld 1.wait_for_dma_start();
  }
}

```

Future Work

- Additional CudaDMA Instances
  - Indirect memory accesses

- More applications
  - Sparse-Matrix operations

- Target for higher-level language/DSL compilers
  - Copperhead, Liszt

- Actual hardware DMA engines for GPUs

- Warp-specialization aware programming models
  - Compiler implementations
Fast Fourier Transforms

- 1D, Power of 2 FFTs

- Compared to optimized CUFFT library (version 4.0)
  - 32 warps per SM

- CudaDMA (custom loader)
  - 24 warps per SM
  - 16 compute, 8 DMA

- Same performance at lower warp count

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>CUFFT (ms)</th>
<th>CudaDMA (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>524288</td>
<td>0.231</td>
<td>0.227</td>
<td>1.017</td>
</tr>
<tr>
<td>1048576</td>
<td>0.457</td>
<td>0.489</td>
<td>0.935</td>
</tr>
<tr>
<td>2097152</td>
<td>0.916</td>
<td>0.982</td>
<td>0.933</td>
</tr>
<tr>
<td>4194304</td>
<td>1.909</td>
<td>1.939</td>
<td>0.985</td>
</tr>
<tr>
<td>8388608</td>
<td>3.894</td>
<td>3.827</td>
<td>1.017</td>
</tr>
<tr>
<td>16777216</td>
<td>8.040</td>
<td>7.995</td>
<td>1.006</td>
</tr>
<tr>
<td>33554432</td>
<td>18.309</td>
<td>18.154</td>
<td>1.008</td>
</tr>
<tr>
<td>67108864</td>
<td>37.191</td>
<td>36.978</td>
<td>1.006</td>
</tr>
</tbody>
</table>