Advanced CUDA Webinar
Memory Optimizations
Outline

- Overview
- Hardware
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
    - Measuring performance – effective bandwidth
    - Coalescing
    - Shared Memory
    - Textures
- Summary
Optimize Algorithms for the GPU

- Maximize independent parallelism
- Maximize arithmetic intensity (math/bandwidth)
- Sometimes it’s better to recompute than to cache
  - GPU spends its transistors on ALUs, not memory
- Do more computation on the GPU to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transferring back and forth to host
Optimize Memory Access

- **Coalesced vs. Non-coalesced** = order of magnitude
  - Global/Local device memory

- Optimize for spatial locality in cached texture memory

- In shared memory, avoid high-degree bank conflicts
Take Advantage of Shared Memory

- Hundreds of times faster than global memory
- Threads can cooperate via shared memory
- Use one / a few threads to load / compute data shared by all threads
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing
Use Parallelism Efficiently

- Partition your computation to keep the GPU multiprocessors equally busy
  - Many threads, many thread blocks

- Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - Registers, shared memory
Outline

- Overview
- **Hardware**
- **Memory Optimizations**
  - Data transfers between host and device
  - Device memory optimizations
    - Measuring performance – effective bandwidth
    - Coalescing
    - Shared Memory
    - Textures

- **Summary**
10-Series Architecture

- 240 Scalar Processor (SP) cores execute kernel threads
- 30 Streaming Multiprocessors (SMs) each contain
  - 8 scalar processors
  - 2 Special Function Units (SFUs)
  - 1 double precision unit
- Shared memory enables thread cooperation
### Execution Model

**Software**

- **Thread**

**Hardware**

- **Thread**
- **Thread Block**
- **Multiprocessor**
- **Grid**
- **Device**

#### Software

- Threads are executed by scalar processors.

#### Hardware

- Thread blocks are executed on multiprocessors.
- Thread blocks do not migrate.
- Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file).
- A kernel is launched as a grid of thread blocks.
- Only one kernel can execute on a device at one time.

© NVIDIA Corporation 2009
Warps and Half Warps

A thread block consists of 32-thread warps.

A warp is executed physically in parallel (SIMD) on a multiprocessor.

A half-warp of 16 threads can coordinate global memory accesses into a single transaction.
Memory Architecture

Host

CPU
Chipset
DRAM

Device

DRAM
{ Local
  
  Global

  Constant

  Texture

GPU

Multiprocessor
  Registers
  Shared Memory

Constant and Texture Caches
## Memory Architecture

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>On-chip</td>
<td>N/A</td>
<td>R/W</td>
<td>One thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>R/W</td>
<td>One thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A</td>
<td>R/W</td>
<td>All threads in a block</td>
<td>Block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>No</td>
<td>R/W</td>
<td>All threads + host</td>
<td>Application</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Application</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Application</td>
</tr>
</tbody>
</table>
Outline

- Overview
- Hardware
- **Memory Optimizations**
  - Data transfers between host and device
  - Device memory optimizations
    - Measuring performance – effective bandwidth
    - Coalescing
    - Shared Memory
    - Textures
- Summary
Host-Device Data Transfers

- **Device to host memory bandwidth much lower than device to device bandwidth**
  - 8 GB/s peak (PCI-e x16 Gen 2) vs. 141 GB/s peak (GTX 280)

- **Minimize transfers**
  - Intermediate data can be allocated, operated on, and deallocated without ever copying them to host memory

- **Group transfers**
  - One large transfer much better than many small ones

© NVIDIA Corporation 2009
cudaMallocHost() allows allocation of page-locked ("pinned") host memory

Enables highest cudaMemcpy performance
- 3.2 GB/s on PCI-e x16 Gen1
- 5.2 GB/s on PCI-e x16 Gen2

See the "bandwidthTest" CUDA SDK sample

Use with caution!!
- Allocating too much page-locked memory can reduce overall system performance
- Test your systems and apps to learn their limits
Overlapping Data Transfers and Computation

- Async and Stream APIs allow overlap of H2D or D2H data transfers with computation
  - CPU computation can overlap data transfers on all CUDA capable devices
  - Kernel computation can overlap data transfers on devices with “Concurrent copy and execution” (roughly compute capability >= 1.1)

- Stream = sequence of operations that execute in order on GPU
  - Operations from different streams can be interleaved
  - Stream ID used as argument to async calls and kernel launches
Asynchronous Data Transfers

- Asynchronous host-device memory copy returns control immediately to CPU
  - `cudaMemcpyAsync(dst, src, size, dir, stream);`
  - requires *pinned* host memory (allocated with “cudaMallocHost”)

- Overlap CPU computation with data transfer
  - `0 = default stream`

```c
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
kernel<<<grid, block>>>(a_d);
cpuFunction();
```

© NVIDIA Corporation 2009
Overlapping kernel and data transfer

**Requires:**
- “Concurrent copy and execute”
- deviceOverlap field of a cudaDeviceProp variable
- Kernel and transfer use different, non-zero streams
  - A CUDA call to stream-0 blocks until all previous calls complete and cannot be overlapped

**Example:**
```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst, src, size, dir, stream1);
kernel<<<grid, block, 0, stream2>>>(...);
```
GPU/CPU Synchronization

- **Context based**
  - `cudaThreadSynchronize()`: Blocks until all previously issued CUDA calls from a CPU thread complete

- **Stream based**
  - `cudaStreamSynchronize(stream)`: Blocks until all CUDA calls issued to given stream complete
  - `cudaStreamQuery(stream)`: Indicates whether stream is idle. Returns `cudaSuccess, cudaErrorNotReady, ...`. Does not block CPU thread
Stream based using events

- Events can be inserted into streams:
  - `cudaEventRecord(event, stream)`
  - Event is recorded when GPU reaches it in a stream
  - Recorded = assigned a timestamp (GPU clocktick)
  - Useful for timing

- `cudaEventSynchronize(event)`
  - Blocks until given event is recorded

- `cudaEventQuery(event)`
  - Indicates whether event has recorded
  - Returns `cudaSuccess`, `cudaErrorNotReady`, ...
  - Does not block CPU thread
Zero copy

- Access host memory directly from device code
  - Transfers implicitly performed as needed by device code
  - Introduced in CUDA 2.2
  - Check `canMapHostMemory` field of `cudaDeviceProp` variable
- All set-up is done on host using mapped memory

```c
cudaSetDeviceFlags(cudaDeviceMapHost);
...

cudaHostAlloc((void **)&a_h, nBytes,
cudaHostAllocMapped);
cudaHostGetDevicePointer((void **)&a_d, (void *)a_h, 0);
for (i=0; i<N; i++) a_h[i] = i;
increment<<<grid, block>>>(a_d, N);
```
Zero copy considerations

- Zero copy will always be a win for integrated devices that utilize CPU memory (you can check this using the `integrated` field in `cudaDeviceProp`)
- Zero copy will be faster if data is only read/written from/to global memory once, for example:
  - Copy input data to GPU memory
  - Run one kernel
  - Copy output data back to CPU memory
- Potentially easier and faster alternative to using `cudaMemcpyAsync`
  - For example, can both read and write CPU memory from within one kernel
- Note that current devices use pointers that are 32-bit so there is a limit of 4GB per context
Theoretical Bandwidth

Device Bandwidth of GTX 280

\[ \text{Memory clock (Hz)} \times \left( \frac{512}{8} \right) \times \frac{2}{1024^3} = 131.9 \text{ GB/s} \]

Specs report 141 GB/s
- Use \(10^9\) B/GB conversion rather than \(1024^3\)
- Whichever you use, be consistent
Effective Bandwidth

Effective Bandwidth (for copying array of N floats)

\[ \frac{N \times 4 \text{ B/element}}{1024^3 \times 2} \times \frac{1}{\text{time in secs}} = \text{GB/s} \]

Array size (bytes) \[ \text{Read and write} \]

B/GB (or \(10^9\))
Outline

- Overview
- Hardware
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
    - Measuring performance – effective bandwidth
    - Coalescing
    - Shared Memory
    - Textures
- Summary
Coalescing

Global memory access of 32, 64, or 128-bit words by a half-warp of threads can result in as few as one (or two) transaction(s) if certain access requirements are met.

Depends on compute capability:
- 1.0 and 1.1 have stricter access requirements.

Float (32-bit) data example:

Global Memory

32-byte segments
64-byte segments
128-byte segments

Half-warp of threads
Coalescing
Compute capability 1.0 and 1.1

K-th thread must access k-th word in the segment (or k-th word in 2 contiguous 128B segments for 128-bit words), not all threads need to participate

Coalesces – 1 transaction

Out of sequence – 16 transactions

Misaligned – 16 transactions
Coalescing
Compute capability 1.2 and higher

- Issues transactions for segments of 32B, 64B, and 128B
- Smaller transactions used to avoid wasted bandwidth

1 transaction - 64B segment

2 transactions - 64B and 32B segments

1 transaction - 128B segment
Coalescing Examples

- Effective bandwidth of small kernels that copy data
  - Effects of offset and stride on performance

- Two GPUs
  - GTX 280
    - Compute capability 1.3
    - Peak bandwidth of 141 GB/s
  - FX 5600
    - Compute capability 1.0
    - Peak bandwidth of 77 GB/s
Coalescing Examples

```c
__global__ void offsetCopy(float *odata, float *idata, int offset)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```

![Copy with Offset](image.png)
Coalescing Examples

```c
__global__ void strideCopy(float *odata, float *idata, int stride)
{
    int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
    odata[xid] = idata[xid];
}
```
Coalescing Examples

- Strided memory access is inherent in many multidimensional problems
  - Stride is generally large (>> 18)

However ...

- Strided access to global memory can be avoided using *shared memory*
Outline

- Overview
- Hardware
- **Memory Optimizations**
  - Data transfers between host and device
  - **Device memory optimizations**
    - Measuring performance – effective bandwidth
    - Coalescing
    - Shared Memory
    - Textures

- Summary
Shared Memory

- Hundred times faster than global memory
- Cache data to reduce global memory accesses
- Threads can cooperate via shared memory
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing
Shared Memory Architecture

- Many threads accessing memory
  - Therefore, memory is divided into **banks**
  - Successive 32-bit words assigned to successive banks

- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks

- Multiple simultaneous accesses to a bank result in a **bank conflict**
  - Conflicting accesses are serialized
Bank Addressing Examples

No Bank Conflicts
- Linear addressing
  - stride == 1

No Bank Conflicts
- Random 1:1 Permutation

© NVIDIA Corporation 2009
Bank Addressing Examples

- **2-way Bank Conflicts**
  - Linear addressing stride == 2
  - Threads: Thread 0, Thread 1, Thread 2, Thread 3, Thread 4, Thread 5, Thread 6, Thread 7, Thread 8, Thread 9, Thread 10, Thread 11
  - Banks: Bank 0, Bank 1, Bank 2, Bank 3, Bank 4, Bank 5, Bank 6, Bank 7

- **8-way Bank Conflicts**
  - Linear addressing stride == 8
  - Threads: Thread 0, Thread 1, Thread 2, Thread 3, Thread 4, Thread 5, Thread 6, Thread 7, Thread 8, Thread 9, Thread 10, Thread 11
  - Banks: Bank 0, Bank 1, Bank 2, Bank 3, Bank 4, Bank 5, Bank 6, Bank 7, Bank 8, Bank 9, Bank 10, Bank 11
Shared memory bank conflicts

- Shared memory is ~ as fast as registers if there are no bank conflicts

- `warp_serialize` profiler signal reflects conflicts

The fast case:
- If all threads of a half-warp access **different banks**, there is no bank conflict
- If all threads of a half-warp read the **identical address**, there is no bank conflict (broadcast)

The slow case:
- Bank Conflict: multiple threads in the same half-warp access the same bank
- Must serialize the accesses
- **Cost** = max # of simultaneous accesses to a single bank
Shared Memory Example: Transpose

- Each thread block works on a tile of the matrix
- Naïve implementation exhibits strided access to global memory

Elements transposed by a half-warp of threads
Naïve Transpose

Loads are coalesced, stores are not (strided by height)

```c
__global__ void transposeNaive(float *odata, float *idata, int width, int height) {
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

    int index_in = xIndex + width * yIndex;
    int index_out = yIndex + height * xIndex;

    odata[index_out] = idata[index_in];
}
```
Coalescing through shared memory

- Access columns of a tile in shared memory to write contiguous data to global memory
- Requires `__syncthreads()` since threads access data in shared memory stored by other threads

Elements transposed by a half-warp of threads
Coalescing through shared memory

```c
__global__ void transposeCoalesced(float *odata, float *idata,
    int width, int height)
{
__shared__ float tile[TILE_DIM][TILE_DIM];

    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index_in = xIndex + (yIndex)*width;

    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = xIndex + (yIndex)*height;

    tile[threadIdx.y][threadIdx.x] = idata[index_in];

    __syncthreads();

    odata[index_out] = tile[threadIdx.x][threadIdx.y];
}
```
Bank Conflicts in Transpose

- **16x16 shared memory tile of floats**
  - Data in columns are in the same bank
  - 16-way bank conflict reading columns in tile

- **Solution - pad shared memory array**
  - `__shared__ float tile[TILE_DIM][TILE_DIM+1];`
  - Data in anti-diagonals are in same bank

Elements transposed by a half-warp of threads
Outline

- Overview
- Hardware
- **Memory Optimizations**
  - Data transfers between host and device
  - **Device memory optimizations**
    - Measuring performance – effective bandwidth
    - Coalescing
    - Shared Memory
    - **Textures**

- Summary
Textures in CUDA

Texture is an object for reading data

Benefits:

- Data is cached
  - Helpful when coalescing is a problem
- Filtering
  - Linear / bilinear / trilinear interpolation
  - Dedicated hardware
- Wrap modes (for “out-of-bounds” addresses)
  - Clamp to edge / repeat
- Addressable in 1D, 2D, or 3D
  - Using integer or normalized coordinates
Texture Addressing

**Wrap**
- Out-of-bounds coordinate is wrapped (modulo arithmetic)

**Clamp**
- Out-of-bounds coordinate is replaced with the closest boundary
CUDA Texture Types

**Bound to linear memory**
- Global memory address is bound to a texture
- Only 1D
- Integer addressing
- No filtering, no addressing modes

**Bound to CUDA arrays**
- Block linear CUDA array is bound to a texture
- 1D, 2D, or 3D
- Float addressing (size-based or normalized)
- Filtering
- Addressing modes (clamping, repeat)

**Bound to pitch linear (CUDA 2.2)**
- Global memory address is bound to a texture
- 2D
- Float/integer addressing, filtering, and clamp/repeat addressing modes similar to CUDA arrays
CUDA Texturing Steps

**Host (CPU) code:**
- Allocate/obtain memory (global linear/pitch linear, or CUDA array)
- Create a texture reference object
  - Currently must be at file-scope
- Bind the texture reference to memory/array
- When done:
  - Unbind the texture reference, free resources

**Device (kernel) code:**
- Fetch using texture reference
- Linear memory textures: `tex1Dfetch()`
- Array textures: `tex1D()` or `tex2D()` or `tex3D()`
- Pitch linear textures: `tex2D()`
Texture Example

```c
__global__ void
shiftCopy(float *odata, float *idata, int shift)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x;
    odata[xid] = idata[xid + shift];
}

texture <float> texRef;

__global__ void
textureShiftCopy(float *odata, float *idata, int shift)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x;
    odata[xid] = tex1Dfetch(texRef, xid + shift);
}
```

© NVIDIA Corporation 2009
GPU hardware can achieve great performance on data-parallel computations if you follow a few simple guidelines:

- Use parallelism efficiently
- Coalesce memory accesses if possible
- Take advantage of shared memory
- Explore other memory spaces
  - Texture
  - Constant
- Reduce bank conflicts
Special CUDA Developer Offer on Tesla GPUs

- 50% off MSRP on Tesla C1060 GPUs
- Up to four per developer
- Act now, limited time offer


*If you are outside of US or Canada, please contact an NVIDIA Tesla Preferred Provider in your country*