Optimization Strategies

— Global Memory Access Pattern and Control Flow
Objectives

- Optimization Strategies
  - Global Memory Access Pattern (Coalescing)
  - Control Flow (Divergent branch)
Global Memory Access

- **Highest latency instructions: 200-400 clock cycles**
  - Likely to be performance bottleneck

- **Optimizations can greatly increase performance**
  - Best access pattern: Coalescing
  - Up to 10x speedup
Coalesced Memory Access

- A coordinated read by a half-warp (16 threads)
- A contiguous region of global memory:
  - 64 bytes - each thread reads a word: int, float, …
  - 128 bytes - each thread reads a double-word: int2, float2, …
  - 256 bytes – each thread reads a quad-word: int4, float4, …
- Additional restrictions on G8X architecture:
  - Starting address for a region must be a multiple of region size
  - The $k^{th}$ thread in a half-warp must access the $k^{th}$ element in a block being read
- Exception: not all threads must be participating
  - Predicated access, divergence within a halfwarp
Coalesced Access: Reading floats

All threads participate

Some threads do not participate
Non-Coalesced Access: Reading floats

Permuted access by threads

Misaligned starting address (not a multiple of 64)
Example: Non-coalesced float3 read

```
__global__ void accessFloat3(float3 *d_in, float3 d_out)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    float3 a = d_in[index];
    a.x += 2;
    a.y += 2;
    a.z += 2;
    d_out[index] = a;
}
```
Example: Non-coalesced float3 read (Cont’)

- float3 is 12 bytes
- Each thread ends up executing 3 reads
  - `sizeof(float3) ≠ 4, 8, or 12`
  - Half-warp reads three 64B non-contiguous regions

First read
Example: Non-coalesced float3 read (2)

Similarly, step 3 start at offset 512
Example: Non-coalesced float3 read (3)

- Use shared memory to allow coalescing
  - Need `sizeof(float3)*(threads/block)` bytes of SMEM
  - Each thread reads 3 scalar floats:
    - Offsets: 0, (threads/block), 2*(threads/block)
    - These will likely be processed by other threads, so sync

- Processing
  - Each thread retrieves its float3 from SMEM array
    - Cast the SMEM pointer to (float3*)
    - Use thread ID as index
  - Rest of the compute code does not change!
Example: Final Coalesced Code

```c
__global__ void accessInt3Shared(float *g_in, float *g_out)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ float s_data[256*3];
    s_data[threadIdx.x] = g_in[index];
    s_data[threadIdx.x+256] = g_in[index+256];
    s_data[threadIdx.x+512] = g_in[index+512];
    __syncthreads();
    float3 a = ((float3*)s_data)[threadIdx.x];
    a.x += 2;
    a.y += 2;
    a.z += 2;
    ((float3*)s_data)[threadIdx.x] = a;
    __syncthreads();
    g_out[index] = s_data[threadIdx.x];
    g_out[index+256] = s_data[threadIdx.x+256];
    g_out[index+512] = s_data[threadIdx.x+512];
}
```
Coalescing: Structure of Size ≠ 4, 8, 16 Bytes

- Use a structure of arrays instead of Array of Structure
- If Array of Structure is not viable:
  - Force structure alignment: `__align(X)`, where \( X = 4, 8, \) or 16
  - Use SMEM to achieve coalescing
Main performance concern with branching is divergence

- Threads within a single warp take different paths
- Different execution paths are serialized
  - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
Divergent Branch

A common case: avoid divergence when branch condition is a function of thread ID

Example with divergence:

- If (threadIdx.x > 2) { }
- This creates two different control paths for threads in a block
- Branch granularity < warp size; threads 0 and 1 follow different path than the rest of the threads in the first warp

Example without divergence:

- If (threadIdx.x / WARP_SIZE > 2) { }
- Also creates two different control paths for threads in a block
- Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path
Parallel Reduction

- Given an array of values, “reduce” them to a single value in parallel.

Examples

- Sum reduction: sum of all values in the array.
- Max reduction: maximum of all values in the array.

Typically parallel implementation:

- Recursively halve the number of threads, add two values per thread.
- Takes log(n) steps for n elements, requires n/2 threads.
A Vector Reduction Example

- Assume an in-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory used to hold a partial sum vector
  - Each iteration brings the partial sum vector closer to the final sum
  - The final solution will be in element 0
Vector Reduction

Array elements

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0+1</td>
<td>2+3</td>
<td>4+5</td>
<td>6+7</td>
<td>8+9</td>
<td>10+11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 0...3 | 4..7 | 8..11 |

| 0..7 | 8..15 |

iterations
A simple implementation

```c
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```
Interleaved Reduction

Values (shared memory)

Step 1
Stride 1

Thread IDs

Values

Step 2
Stride 2

Thread IDs

Values

Step 3
Stride 4

Thread IDs

Values

Step 4
Stride 8

Thread IDs

Values
Some Observations

- In each iterations, two control flow paths will be sequentially traversed for each warp
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition may cost extra cycles depending on the implementation of divergence

```c
// do reduction in shared mem
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

Problem: highly divergent branching results in very poor performance!
Some Observations (Cont’)

- No more than half of threads will be executing at any time
  - All odd index threads are disabled right from the beginning!
  - On average, less than \( \frac{1}{4} \) of the threads will be activated for all warps over time.
  - After the 5\(^{th}\) iteration, entire warps in each block will be disabled, poor resource utilization but no divergence.
  - This can go on for a while, up to 4 more iterations (512/32=16= \(2^4\)), where each iteration only has one thread activated until all warps retire.

![Thread Diagram]

Copyright © 2013 by Yong Cao, Referencing UIUC ECE498AL Course Notes
Optimization 1:

- Replace divergent branch

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

- With strided index and non-divergent branch

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```
Optimization 1: (Cont’)

Values (shared memory)

- Step 1: Thread IDs
  - Stride 1
  - Values

- Step 2: Thread IDs
  - Stride 2
  - Values

- Step 3: Thread IDs
  - Stride 4
  - Values

- Step 4: Thread IDs
  - Stride 8
  - Values

No divergence until less than 16 sub sum.
Optimization 1: Bank Conflict Issue

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

Bank Conflict due to the Strided Addressing
Optimization 2: Sequential Addressing

Values (shared memory)

Step 1
Stride 8
Thread IDs

Values

Step 2
Stride 4
Thread IDs

Values

Step 3
Stride 2
Thread IDs

Values

Step 4
Stride 1
Thread IDs

Values

Copyright © 2013 by Yong Cao, Referencing UIUC ECE498AL Course Notes
Optimization 2: (Cont’)

- Replace strided indexing

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

- With reversed loop and threadIdx-based indexing

```c
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```
Some Observations About the New Implementation

- Only the last 5 iterations will have divergence
- Entire warps will be shut down as iterations progress
  - For a 512-thread block, 4 iterations to shut down all but one warps in each block
  - Better resource utilization, will likely retire warps and thus blocks faster
- Recall, no bank conflicts either