CS4803DGC Design Game Consoles

Spring 2010
Prof. Hyesoon Kim
CUDA Optimization Strategies

• Optimize Algorithms for the GPU
  – Reduce communications between the CPU and GPU
• Increase occupancy
• Optimize Memory Access Coherence
• Take Advantage of On-Chip Shared Memory
• Use Parallelism Efficiently
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 Coherence

- 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 noncoalesceable 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
Global Memory Reads/Writes

• Highest latency instructions: 400-600 clock cycles
• Likely to be performance bottleneck
• Optimizations can greatly increase performance
  – Coalescing: up to 10x speedup
Coalesced/Uncoalesced

One warp generates a memory request

Coalesced memory access type

Uncoalesced memory access type

- More processing cycles for the uncoalesced case

One memory transaction

Multiple memory transactions

- More processing cycles for the uncoalesced case
Coalesced Access: Reading floats

All threads participate

Some threads do not participate
Uncoalesced Access: Reading floats (Computing Capability <1.2)

- Computing capability =1.2 (GTX280, T10C). Those two cases are treated as coalesced memory.

**Permuted Access by Threads**

- Misaligned Starting Address (not a multiple of 64)
Coalescing: Timing Results

• Experiment:
  – Kernel: read a float, increment, write back
  – 3M floats (12MB)
  – Times averaged over 10K runs

• 12K blocks x 256 threads:
  – 356μs – coalesced
  – 357μs – coalesced, some threads don’t participate
  – 3,494μs – permuted/misaligned thread access
__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;
}

Uncoalesced float3 Code
**Uncoalesced Access:**

**float3 Case**

- 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
Coalescing float3 Access

Similarly, Step3 starting at offset 512
Coalesced Access: float3 Case

```c
__global__ void accessInt3Shared(float *g_in, float *g_out)
{
    int index = 3 * 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, or 16 Bytes

• Use a structure of arrays instead of AoS
• If SoA is not viable:
  – Force structure alignment: __align(X), where X = 4, 8, or 16
  – Use SMEM to achieve coalescing
SOA & AOS (Review)

- Array of structures (AOS)
  - \{x_1, y_1, z_1, w_1\}, \{x_2, y_2, z_2, w_2\}, \{x_3, y_3, z_3, w_3\}, \{x_4, y_4, z_4, w_4\} …
  - Intuitive but less efficient
  - What if we want to perform only x axis?

- Structure of array (SOA)
  - \{x_1, x_2, x_3, x_4\}, …, \{y_1, y_2, y_3, y_4\}, …, \{z_1, z_2, z_3, z_4\}, … \{w_1, w_2, w_3, w_4\} …
Coalescing: summary

- Coalescing greatly improves throughput
- Critical to small or memory-bound kernels
- Reading structures of size other than 4, 8, or 16 bytes will break coalescing:
  - Prefer Structures of Arrays over AoS
  - If SoA is not viable, read/write through SMEM
- Future proof code: coalesce over whole warps
- Additional resources:
  - Aligned Types CUDA SDK Sample
Occupancy

- Thread instructions executed sequentially, executing other warps is the only way to hide latencies and keep the hardware busy.
- **Occupancy** = Number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently.
- Minimize occupancy requirements by minimizing latency.
- Maximize occupancy by optimizing threads per multiprocessor.
Increasing occupancy does not necessarily increase performance

- *BUT*...

Low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels
  - (It all comes down to arithmetic intensity and available parallelism)
Use Occupancy calculator

- Part of the SDK
Prefetching

• One could double buffer the computation, getting better instruction mix within each thread
  – This is classic software pipelining in ILP compilers

Loop {
  Load current tile to shared memory
  syncthreads()
  Compute current tile
  syncthreads()
}

Loop {
  Load next tile from global memory
  Deposit current tile to shared memory
  syncthreads()
  Load next tile from global memory
  Compute current tile
  syncthreads()
}
Prefetch

- Deposit blue tile from register into shared memory
- Syncthreads
- Load orange tile into register
- Compute Blue tile
- Deposit orange tile into shared memory
- ....
Each thread block must load into shared memory the pixels to be filtered and the apron pixels.
Optimization I: Avoid idle thread

• When the kernel size is relatively too big compared to image size
• Use threads to load multiple image blocks
• Use 1/3 threads

[Diagram showing multiple blocks loaded by threads]
Optimization II

- Memory Coalescing:

   - Threads Inactive During Load Stage

   - KERNEL_RADIUS
   - ROW_TILE_W
   - KERNEL_RADIUS

   - KERNEL_RADIUS_ALIGNED
   - blockDim.x
Optimization-III

• Unrolling the kernel

```c
for(int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++)
    sum += data[smemPos + k] * d_Kernel[KERNEL_RADIUS - k];
```

• `#pragma unroll`
  – By default, the compiler unrolls small loops with a known trip count.
  – The `#pragma unroll` directive however can be used to control unrolling of any given loop.