Programming for Performance

General-purpose Programming of Massively Parallel Graphics Processors
Shiraz University, Spring 2010
Instructor: Reza Azimi

Some materials/slides are adapted from:
Andreas Moshovos’ Course at the University of Toronto
UIUC course by Wen-Mei Hwu and David Kirk

Optimization Steps

- Algorithmic Optimizations
  - Maximize Independent Parallelism
  - Maximize Arithmetic Intensity
- Remove Control Flow Divergence
- Memory Access Coalescing
- Exploit On-chip Shared Memory
  - Remove bank conflicts
- Use Parallelism Efficiently
Maximize Independent Parallelism

- Decompose the problem into many tasks
  - Not a few, but thousands or millions

- Avoid synchronization as much as possible
  - Minimize dependence among threads
    - Obvious case: One step feeds result to the next steps

- We’ll see more of this with examples

Maximize Arithmetic Intensity

- 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
- Sometimes it’s better to re-compute than to cache
  - GPU spends its transistors on ALUs, not memory
Control Flow Divergence

```c
if (in[i] == 0)
    out[i] = sqrt(x);
else
    out[i] = 10;
```

Source: A. Moshovos

Control Flow Divergence Contd.

```c
if (in[i] == 0)
    WARP #1
else
    WARP #2
```

Bad

Good

Source: A. Moshovos
Memory Access Coalescing

- A coordinated memory access by a half-warp (16 threads) becomes a single wide memory read
  - All accesses must fall into a continuous region

- Region Size
  - 16 bytes: each thread reads a byte (char)
  - 32 bytes: each thread reads two bytes (short)
  - 64 bytes: each thread reads a word (int, float, …)
  - 128 bytes: each thread reads a double-word (double, int2, float2, …)
  - 256 bytes: each thread reads a quad-word (int4, float4, …)
Coalesced Read

Coalescing Experiment

- Kernel: arrmul_kernel
  - Each thread multiplies an entry of an array by a number

- Array Size: 8 million Entries

- 16384 blocks x 512 threads/block

- Execution Time averaged over 1000 runs

Source: A. Moshovos

GPU Programming, Shiraz University, Winter 88/Spring 89, Reza Azimi
Kernel 1: Coalesced Access

```c
__global__ arrMulKernel (int *g_idata, int x) {
    threadId = blockIdx.x * blockDim.x + threadIdx.x;
    g_idata[threadId] *= x;
}
```

- Execution Time: 630 microseconds

Kernel 2: Not All Threads Participate

```c
__global__ arrMulKernel (int *g_idata, int x) {
    threadId = blockIdx.x * blockDim.x + threadIdx.x;
    if (threadId % 3 != 0)
        g_idata[threadId] *= x;
}
```

- Execution Time: 780 microseconds
Kernel 3: Two Threads Access One

```c
__global__ arrMulKernel (int *g_idata, int x)
{
    threadId = blockIdx.x * blockDim.x + threadIdx.x;
    g_idata[threadId & ~1] *= x;
}
```

- Execution Time: 630 microseconds

Kernel 4: Un-coalesced Access

```c
__global__ arrMulKernel (int *g_idata, int x)
{
    threadId = blockIdx.x * blockDim.x + threadIdx.x;

    if (threadId % 3 != 0)
        g_idata[threadId] *= x;
    else
        g_idata[0] *= x;
}
```

- Execution Time: 9780 microseconds
Kernel 4: Coalesced Access

```c
__global__ arrMulKernel (int *g_idata, int x)
{
    threadId = blockIdx.x * blockDim.x + threadIdx.x;

    if (threadId % 3 != 0)
        g_idata[threadId] *= x;
    else
        g_idata[threadId & 0xFFFFFFFFC0] *= x;
}
```

- The 3rd thread accesses the start of the 64-byte region
- Execution Time: 1175 microseconds

---

Un-coalesced `float3` access code

```c
__global__ void
accessFloat3(float3 *g_idata, float X)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    g_idata[index].x *= X;
    g_idata[index].y *= X;
    g_idata[index].z *= X;
}
```

Execution time: 5050 µs
8M float3 = 96MBytes
32768 thread blocks, 256 threads per block
Averaged over 1000 runs
Un-coalesced float3 Access Sequence

- Each thread ends up executing three 32bit reads
- sizeof(float3) = 12
- Offsets read for each half-warp:
  - First Read: 0, 12, 24, ..., 180
  - Second Read: 4, 16, 28, ..., 184
  - Third Read: 8, 20, 32, ..., 188
- Each half-warp reads three 64B non-contiguous regions

Coalescing float3 Accesses
Coalescing float3 strategy

- Use shared memory to allow coalescing
  - Need sizeof(float3)*(threads/block) bytes of shared memory per block
  - Let’s assume we have 256 threads per block

- Three Phases:
  1. Fetch data in shared memory
     - Each thread reads 3 scalar floats
     - Offsets: 0, 256, 512
     - sync threads to make sure everyone got this done
  2. Processing
     - Each thread retrieves its float3 from SMEM array
     - Use thread ID within the block as index
     - Rest of the computation does not change
  3. Write results bank to global memory
     - Each thread writes 3 scalar floats
     - Offsets: 0, 256, 512

---

Coalesing float3 Accesses

```c
__global__ void memCoalKernelShared(float3 *g_idata, float X) {
    int threadIdx = (blockIdx.x * blockDim.x) + threadIdx.x;
    __shared__ float s_data[256 * 3];
    float *g_data = (float *)g_idata;
    s_data[threadIdx.x] = g_data[threadIdx.x];
    s_data[threadIdx.x+256] = g_data[threadIdx.x+256];
    s_data[threadIdx.x+512] = g_data[threadIdx.x+512];
    s_data[threadIdx.x] *= X;
    s_data[threadIdx.x+1] *= X;
    s_data[threadIdx.x+2] *= X;
    g_data[threadIdx] = s_data[threadIdx.x];
    g_data[threadIdx+256] = s_data[threadIdx.x+256];
    g_data[threadIdx+512] = s_data[threadIdx.x+512];
}
```
Performance Results

- Experiment:
  - 8M float3 (96MB)
  - 32768 thread blocks, 256 threads per block
  - Times averaged over 1000 runs

- Un-coalesced Access: **5050 µs**

- Coalesced Access through
  Shared Memory: **1912 µs**

Global Memory 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 Arrays of Structures
    - If structures of arrays is not viable, read/write through SMEM

- Planning for Future:
  - Coalesce over whole warps instead of half-warps
Shared Memory

- Shared memory is divided into banks
  - Essential to achieve high bandwidth
  - Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks (parallel access)

- 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
Bank Addressing Examples

How Addresses Map to Banks on GT200

- Each bank can fetch 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- G200 has 16 banks
  - \( \text{bank number} = (\text{address} / 4) \mod 16 \)
- Same as the size of a half-warp
  - No bank conflicts between different half-warps, only within a single half-warp
Shared Memory Access

- The Fast Case: **No Bank Conflict**
  - All threads of a half-warp (16 threads) access different banks
  - All threads of a half-warp access the identical address (broadcast)
  - Memory access is as fast as access to registers

- 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
    - An 8-way bank conflict costs 8 cycles

---

Linear Addressing

```c
__shared__ float shared[1024];
float foo = shared[s * threadIdx.x];
```

This is only bank-conflict-free if `s` shares no common factors with the number of banks
- 16 on G200, so `s` must be odd

---

GPU Programming, Shiraz University, Winter 88/Spring 89, Reza Azimi
Data types and Bank Conflicts

- No bank conflicts if the type of `array` is **32-bits**:

  
  ```
  foo = array[baseIndex + threadIdx.x]
  ```

- Smaller data types
  - **8-bit** (char) => 4-way bank conflicts:
    ```
    __shared__ char array[256];
    foo = array[baseIndex + threadIdx.x];
    ```
  - **16-bit** (short) => 2-way bank conflicts:
    ```
    __shared__ short array[256];
    foo = array[baseIndex + threadIdx.x];
    ```

Structs and Bank Conflicts

- Struct assignments compile into as many memory accesses as there are `struct` members:

  ```
  struct vector { float x, y, z; };
  struct myType { float f; int c; };
  __shared__ struct vector vectors[64];
  __shared__ struct myType myTypes[64];
  ```

- This has no bank conflicts for `vector`:
  - struct size is 3 words
  - 3 accesses per thread, contiguous banks (no common factor with 16)
  ```
  struct vector v = vectors[baseIndex + threadIdx.x].x;
  ```

- This has 2-way bank conflicts for `myType`:
  - 2 accesses per thread
  ```
  struct myType m = myTypes[baseIndex + threadIdx.x].c;
  ```
1D Array Bank Conflict Patterns

```c
int tid = threadIdx.x;
__shared__ int s_array[256];
s_array[2*tid] = g_array[2*tid];
s_array[2*tid+1] = g_array[2*tid+1];
```

- Each thread loads 2 elements into shared mem:
  - 2-way-interleaved loads result in 2-way bank conflicts.
- This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.
- Not in shared memory usage where there is no cache line effects but banking effects

A Better Access Pattern

```c
int tid = threadIdx.x;
__shared__ int s_array[256];
s_array[tid] = g_array[tid];
s_array[tid + blockDim.x] = g_array[tid + blockDim.x];
```

Each thread loads one element in every consecutive group of `blockDim` elements.
2D Bank Conflict Patterns

- Problem (example: 16x16 block)
  - Each thread processes a row
  - Threads in a block access the elements in each **column** simultaneously
  - 16-way bank conflicts:
    - Rows all start at bank 0

- Solution
  - Pad the rows
    - Add one float to the end of each row
  - Transpose the matrix before processing
    - Suffer bank conflicts during transpose
    - May still be advantageous

Bank Conflict Experiment

```c
__global__ void bankConfKernel(float X)
{
    __shared__ float matrix[16][16];
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    // assigning each thread a row
    for (int k = 0; k < 16; k++)
        matrix[index][k] *= X;
}
```

**Execution time: 4256 µs**
8192 thread blocks, 256 threads per block
Averaged over 1000 runs
Bank Conflict Experiment: Padding

```c
__global__ void bankConfKernel(float X)
{
    __shared__ float matrix[16][17];
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    // assigning each thread a row
    for (int k = 0; k < 16; k++)
    {
        matrix[index][k] *= X;
    }
}
```

**Execution time: 309 µs**

A speed up of 16 over the unpadded version due to removing 16-way bank conflict!

Example: Matrix Transpose

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
**Uncoalesced Transpose**

```c
__global__ void transpose_naive(float* odata, float* idata, int width, int height) {
    int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
    int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
    if (xIndex < width && yIndex < height) {
        int index_in = xIndex + width * yIndex;
        int index_out = yIndex + height * xIndex;
        odata[index_out] = idata[index_in];
    }
}
```

---

**Uncoalesced Memory Access Pattern**

![Uncoalesced Memory Access Pattern](image)
Coalesced Transpose

- Conceptually partition the input matrix into square tiles

- **ThreadBlock** \((bx, by)\):
  - Read the \((bx,by)\) input tile, store into SMEM
  - Write the SMEM data to \((by,bx)\) output tile
    - Transpose the indexing into SMEM

- **Thread** \((tx,ty)\):
  - Reads element \((tx,ty)\) from input tile
  - Writes element \((tx,ty)\) into output tile

- **Coalescing is achieved if:**
  - Block/tile dimensions are multiples of 16 (why?)

---

Using Shared Memory
### Coalesced Transpose: Access Patterns

**Reads from GMEM**

![Access Patterns Diagram](image1)

**Writes to SMEM**

**Reads from SMEM**

**Writes to GMEM**

### Avoiding Bank Conflicts

- **Threads read SMEM with stride**
  - 16x16-way bank conflicts
  - 16x slower than no conflicts

- **Solution: Padding**
  - Read stride = 17
  - Threads read from consecutive banks
Coalesced, Conflict-Free Transpose

```c
__global__ void transpose(float *odata, float *idata, int width, int height) {
    __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];

    // read the matrix tile into shared memory
    int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    if ((xIndex < width) && (yIndex < height)) {
        int index_in = yIndex * width + xIndex;
        block[threadIdx.y][threadIdx.x] = idata[index_in];
    }
    __syncthreads();

    // write the transposed matrix tile to global memory
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    if ((xIndex < height) && (yIndex < width)) {
        int index_out = yIndex * height + xIndex;
        odata[index_out] = block[threadIdx.x][threadIdx.y];
    }
}
```

Transpose Measurements

- Average over 100 runs
- 16x16 thread blocks

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Naïve (ms)</th>
<th>Optimized (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024x1024</td>
<td>3</td>
<td>0.4</td>
<td>7.5</td>
</tr>
<tr>
<td>2048x2048</td>
<td>13.5</td>
<td>1.7</td>
<td>8</td>
</tr>
<tr>
<td>4096x4096</td>
<td>91</td>
<td>7</td>
<td>13</td>
</tr>
</tbody>
</table>
Summary of the Major Optimizations

- Maximizing GPU Occupancy
- Remove Control Flow Divergence
- Enable Memory Access Coalescing
- Exploit On-chip Shared Memory
- Remove Shared Memory Bank Conflicts