Convolution Soup:
A case study in CUDA optimization

The Fairmont San Jose | 10:30 AM  Friday October 2, 2009  | Joe Stam
Optimization

GPUs are very fast

BUT...

• Naïve programming can result in disappointing performance

• Applying a basic understanding of the architectural can yield significant performance improvements
A Convolution Case Study

We’ll use the simple, ubiquitous example of a 5x5 image convolution to illustrate optimization strategies and their effects.

- Basic 5x5 convolution
- 8-bit data, monochrome
- Generalized non-separable case
- No special border handling
- Benchmarks on 2048 X 2048 image GeForce 8800 GT (G92)
What to Optimize?

- GMEM Coalescing
- GMEM Bandwidth
- Occupancy
  - # of threads running on an SM
  - Limited by Registers, SMEM, 8-blocks maximum, 768 threads maximum (1024 on GT200)
  - More threads running allows more latency hiding!
- SMEM Bank Conflicts
- LMEM usage
- Compute instructions
  - inlining, __mul24() intrinsics, fast math
Coalescing GMEM: Often the most important optimization

- 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:
  - 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 (32-bit)

---

All Threads Participate

---

Some Threads Do Not Participate
Uncoalesced Access:
Reading floats (32-bit)

Permuted Access by Threads

Misaligned Starting Address (not a multiple of 64)
Coalescing
SM 1.2 and higher
add coalescing buffers

• Coalescing is achieved for any pattern of addresses that fits into a segment of size: 32B for 8-bit words, 64B for 16-bit words, 128B for 32- and 64-bit words

• Alignment within a segment is no longer a concern, but heavily scattered reads and writes are still slow
Tools

- Look at the .cubin to find register, smem, lmem usage (-keep compiler option)
- Verbose PTXAS output (--ptxas-options=-v)

```c
26 code {
27     name = __Z25NaiveGlobalConvolutionKernelPh3_jjjf
28     lmem = 0
29     smem = 48
30     reg = 15
31     bar = 0
32     const {
33         segname = const
34         segnum = 1
35         offset = 0
36         bytes = 16
37         mem {
38             0x00000001 0x0000003f 0x0000003c 0x0000000f
39         }
40     }
41     } code {  
42         0xa0004c05 0x04200780 0xa0004209 0x04200780
43         0x00202005 0x00001070 0xa0000009 0x04000760
44         0x20000219 0x04008780 0x30800dfd 0x6440c7c9
45         0x0000000d 0x000000280 0x213eeec0d 0x0fffffff
```
Visual Profiler
## CUDA GPU Occupancy Calculator

### Instructions
- Click here for detailed instructions on how to use this occupancy calculator.

### Resources
- For more information on NVIDIA CUDA, visit [developer.nvidia.com/cuda](http://developer.nvidia.com/cuda)

### Calculator Features
- Occupancy Calculator
- Spreadsheet

### Calculator Components
- **Occupancy Calculator**
  - Input fields for GPU occupancy calculation
- **Spreadsheet**
  - Data table for occupancy calculation

### Graphs
- **Varying Block Count**
  - Graph showing occupancy for varying block counts
- **Varying Register Count**
  - Graph showing occupancy for varying register counts
- **Varying Shared Memory Usage**
  - Graph showing occupancy for varying shared memory usage

### Physical Limits
- **CUDA**
  - Thread Limit
  - Warp Limit
  - Block Limit

### Memory Usage
- **CUDA**
  - Shared Memory
  - Global Memory

### CUDA Occupancy Calculator
- **CUDA Occupancy Calculator**
  - Calculation details

### License
- **Copyright and License**
  - License information
Memory vs. Compute

• Kernel time is frequently dominated by memory bandwidth
• With enough compute, memory latencies can be hidden by thread swapping

• Write kernels with compute commented out to determine the compute-to-memory ratio.
Example 1: Naïve GMEM

```c
__global__ void NaiveGlobalConvolutionKernel(unsigned char * img_in, unsigned char * img_out,
                                              unsigned int width, unsigned int height,
                                              unsigned int pitch, float scale)
{
    unsigned int X = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    unsigned int Y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;

    if(X > 1 && X < width-2 && Y > 1 && Y < height-2)
    {
        int sum = 0;
        int kidx = 0;
        for(int i = -2;i<= 2;i++)
        {
            for(int j= -2;j<= 2;j++)
            {
                sum += gpu_kernel[kidx++] * img_in[__umul24((Y+i),pitch) + X+j];
            }
        }
        sum = (int)((float)sum * scale);
        img_out[__umul24(Y,pitch) + X] = CLAMP(sum,0,255);
    }
}
```

Warning: Do not try this at home!
Results

148 ms!

- Nothing is coalesced
- Each output requires 25 reads to GMEM
- 8-bit Memory accesses are very inefficient

You are the weakest link — Good Bye!
Example 2: Simple Textures

- Texture hardware provides cached access to GMEM
- No worries about coalescing reads

But:
- Original test required 0.6 ms additional time to copy to cudaArray. Since CUDA 2.2, textures can be bound directly to GMEM!
- Still using 8-bit writes back to GMEM

Results: 7.8 ms
Example 2a: Textures

• Process 4 pixels / thread
• Using 32-bit writes improves bandwidth and coalesces

Results: 6.3 ms
~25% faster
Example 3: Texture into SMEM

• Use textures to fetch memory to avoid coalescing issues
• Store tile in SMEM so all pixels are only read once

Results: 3.3 ms!
Memory read/write only: 1.1 ms

Stopping here would be quite respectable!
Example 3 (Cont.)

• Unfortunately, textures use a lot of registers (25 in this kernel)—This reduces occupancy
• 16x16 block size limits us to 1 block / SM (on G92), thus the entire block is stalled during the `__syncthreads()`
• 16x8 block allows 2 blocks / SM,
  – Surprisingly little performance improvement (3.2 ms)
• 16x20 block maximizes threads
  – Also little performance improvement (3.2 ms)
  – Memory bandwidth goes up slightly because of fewer reads from kernel apron overlap
Example 4: GMEM to SMEM

- 4 pixels / thread
- Try naïve implementation first. Shift reads left & up by apron amount, then read an additional 4-pixel strip to the right
- All loads are uncoalesced!

Results: 3.6 ms
Memory only: 1.6 ms

Slightly worse than using textures
Example 4: GMEM to SMEM
Strict Coalescing

• Process 4 pixels / thread for 32-bit reads
• Read an image tile plus the apron into SMEM
• For 16x16 block size, read 72x20 pixels into SMEM
Convolution SMEM Reads

Step 1: All threads read center top portion into memory
Convolution SMEM Reads

Step 2: Threads with $\text{threadIdx.y} < 4$ read bottom two rows
Convolution SMEM Reads

Step 3: Threads with threadIdx.x == 15 read left-top apron pixels
Convolution SMEM Reads

Step 4: Threads with $\text{threadIdx.x} = 15$ && $\text{threadIdx.y} < 4$ read left-bottom apron pixels
Convolution SMEM Reads

Step 5: Threads with `threadIdx.x == 0` read top-right apron pixels
Convolution SMEM Reads

Step 6: Threads with threadIdx.x == 0 && threadIdx.y < 4 read bottom-right apron pixels
Example 5: GMEM to SMEM
Strict Coalescing (Cont.)

- Process 4 pixels / thread for 32-bit reads
- Read an image tile plus the apron into SMEM
- For 16x16 block size, read 72x20 pixels into SMEM

Results: 3.4 ms
Memory only: 1.2 ms (8.4 GB/s)

Note: Texture is slightly better, even with all this work
Example 5: Effect of block Size

- 1200 bytes of SMEM per block
- 11 registers
- 16x16 = 2 blocks / SM
- 16x8 = 5 blocks / SM benefit

16x8 Results:
3.5 ms
No benefit (more apron pixel redundancy)

16x20 Results:
3.4 ms
Again, no real benefit
Example 5: A Couple Pathological Cases

- Non multiple-of-16 block width results in non-coalesced access and wasted threads

- Change image pitch to break coalescing

19x13 Results:
5.1 ms (50% decrease)
Memory only: 2.4 ms (4.2 GB/s)

Unaligned Pitch Results: 4.3 ms
Memory only: 2.4 ms
Example 6: 128-bit Read/Write to GMEM

• Reading data in 128 bit words is faster than 32-bit words (64-bit is also good)

• Same amount of data, but fewer *transactions* to the memory controller

• Read data as `int4`'s, cast to `char`, process 16-bytes / thread

Results: 4.8 ms
Memory only: 0.4 ms (25 GB/s)

What happened? Memory is way faster, but compute is SLOW...
The Answer

SMEM Bank Conflicts causes warp serialization

<table>
<thead>
<tr>
<th></th>
<th>branch</th>
<th>divergent branch</th>
<th>instructions</th>
<th>warp serialize</th>
<th>cta launched</th>
</tr>
</thead>
<tbody>
<tr>
<td>15767</td>
<td>1168</td>
<td>45235</td>
<td>8591</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>72415</td>
<td>1168</td>
<td>685146</td>
<td>348967</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>18418</td>
<td>1176</td>
<td>47364</td>
<td>9403</td>
<td>196</td>
<td></td>
</tr>
<tr>
<td>81674</td>
<td>1164</td>
<td>501579</td>
<td>11288</td>
<td>195</td>
<td></td>
</tr>
<tr>
<td>18236</td>
<td>1164</td>
<td>47771</td>
<td>9411</td>
<td>195</td>
<td></td>
</tr>
<tr>
<td>35414</td>
<td>1176</td>
<td>427735</td>
<td>12484</td>
<td>196</td>
<td></td>
</tr>
</tbody>
</table>
Banked SMEM Architecture

• Many threads accessing memory
  – Therefore, 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

• Multiple simultaneous accesses to a bank result in a **bank conflict**
  – Conflicting accesses are serialized

16 Banks on SM 1, 32 Banks on the new Fermi SM 2 architecture
No Bank Conflicts
– Linear addressing, stride == 1
Thread 0 → Bank 0
Thread 1 → Bank 1
Thread 2 → Bank 2
Thread 3 → Bank 3
Thread 4 → Bank 4
Thread 5 → Bank 5
Thread 6 → Bank 6
Thread 7 → Bank 7
Thread 15 → Bank 15

No Bank Conflicts
– Random 1:1 Permutation
Thread 0 → Bank 0
Thread 1 → Bank 1
Thread 2 → Bank 2
Thread 3 → Bank 3
Thread 4 → Bank 4
Thread 5 → Bank 5
Thread 6 → Bank 6
Thread 7 → Bank 7
Thread 15 → Bank 15
Bank Addressing Examples

2-Way Bank Conflicts
– Linear addressing, stride == 2

8-Way Bank Conflicts
– Linear addressing, stride == 8

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11

Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 8
Bank 9
Bank 10
Bank 11

Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 8
Bank 9
Bank 10
Bank 11
Sidebar: Tips for Avoiding Bank Conflicts

- When processing color RGBA images it’s best to store each color plane into a different region of SMEM.
- For image processing operations requiring thread/row access or transpose, allocate SMEM as 17xN for a 16xN tile.

<table>
<thead>
<tr>
<th>SMEM Columns</th>
<th>SMEM Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 1</td>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 1</td>
</tr>
<tr>
<td>2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 1 2</td>
<td>2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 1 2</td>
</tr>
<tr>
<td>3 4 5 6 7 8 9 10 11 12 13 14 15 16 1 2 3</td>
<td>3 4 5 6 7 8 9 10 11 12 13 14 15 16 1 2 3</td>
</tr>
<tr>
<td>4 5 6 7 8 9 10 11 12 13 14 15 16 1 2 3 4</td>
<td>4 5 6 7 8 9 10 11 12 13 14 15 16 1 2 3 4</td>
</tr>
<tr>
<td>5 6 7 8 9 10 11 12 13 14 15 16 1 2 3 4 5</td>
<td>5 6 7 8 9 10 11 12 13 14 15 16 1 2 3 4 5</td>
</tr>
<tr>
<td>6 7 8 9 10 11 12 13 14 15 16 1 2 3 4 5 6</td>
<td>6 7 8 9 10 11 12 13 14 15 16 1 2 3 4 5 6</td>
</tr>
<tr>
<td>7 8 9 10 11 12 13 14 15 16 1 2 3 4 5 6 7</td>
<td>7 8 9 10 11 12 13 14 15 16 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>8 9 10 11 12 13 14 15 16 1 2 3 4 5 6 7 8</td>
<td>8 9 10 11 12 13 14 15 16 1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>
Example 7: 128-bit, Resolve Bank Conflicts

- Have each thread process every 4th 32-bit word
- Intermediate results are stored in SMEM
  - Need to shrink the block size since this uses more SMEM

Results: 2.9 ms!
Memory only: 0.4 ms
Example 8: 128-bit, Unroll Inner Loop

- Mostly focused on memory optimizations until now
- All code used fast math where possible (e.g. \texttt{\_umul24})
- Unroll Loops

Results: 2.5 ms
Memory only: 0.4 ms (25 GB/s*)

*Maximum bandwidth on the NVIDIA GeForce 8800 GT is 57.6 GB/s. 25 GB/s is quite good considering the complexity of the tile apron reads.
Summary

Convolution Approaches Comparison

- **Time (ms.)**
- **Algorithm**

- Compute
- Memory Only
GT200 Optimization

- Coalescing buffers greatly improve the performance, especially in non-optimal situations.
Apron Reading: GT200
Coalescing buffers greatly simplify read patterns

Step 1: All threads read, shift up and left

Step 2: Threads with threadIdx.x < 2 read right columns
Apron Reading: GT200

Step 3: Threads with threadIdx.y < 4 read bottom rows

Step 4: Threads with threadIdx.x < 2 && threadIdx.y < 4 read bottom-right apron pixels
GT200 Observations

• Naïve case isn’t so bad: 2.5x slower than best case, vs. 60x slower on G92
• 128-bit is still best
• Complex Coalesced pattern to fill SMEM is actually slightly slower than just a simple shift
Some Conclusions

• Naïve code can be very bad
• Textures improve greatly upon Naïve code, but still only about 50% efficiency
• SMEM has huge benefits for data-sharing algorithms like convolutions
• Final optimizations include 128-bit GMEM read/write, loop unrolling, fast math
• GPU memory architecture is evolving rapidly and becoming easier to optimize