Convolution Soup:
A case study in CUDA optimization

The Fairmont San Jose  |  Joe Stam
Optimization

GPUs are very fast

BUT...

• Poor programming can lead to disappointing performance
• Squeaking out the most speed takes a bit of expertise
A Convolution Case Study

We’ll use the simple, ubiquitous example of a 5x5 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 = _Z22NaiveGlobalConvolutionKernelPhS_jjdf
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 0x0000003ff 0x0000000c 0x0000000f
39          }
40      }
41  }  // bincode
```
PTX—GPU intermediate assembly

• Use -keep to write it
• Not exactly the machine code—it’s useful but not final
• To show interleaved source code:

```
--opencc-options -LIST:source
```

```
1264  st.shared.u32 [v[e6f4]], r78; // id=493 memf+C00
1265  0x11 bne $r3.4, 77; //
1266  add.u32 $r79, $r2, 1; //
1267  mul24.lo.u32 $r30, $r18, $r78; //
1268  add.u32 $r91, $r9, $r80; //
1269  cvt.ru.f32.u32 $r51, $r81; //
1270  mov.f32 $r54, 0x00000000; // -2
1271  add.f32 $r55, $r53, $r34; //
1272  mov.f32 $r56, $r56; //
1273  mov.f32 $r57, 0x00000000; // 0
1274  mov.f32 $r58, 0x00000000; // 0
1275  tex.2d.v4.u32.f32 ($v32, $v33, $v84, $v65), [normFloatTex, $v255, $v256, $v257, $v258];
1276  .loc 2  556 0
1277  // 552 if(threadIdx.x < 4)
1278  // 553 {
1279  // 554 // 555
1280  // 556  sidx = __umul24(blockDim.y+threadIdx.y, smem_pitch) + blockDim.y + threadIdx.x;
1281  // 557  smem[sidx] = tex2d(normFloatTex, {float}{__umul24(blockDim.x+threadIdx.x)*-2.0f, Ytex});
1282  // 558  add.m32 $r36, $r32; //
1283  // 559  add.u32 $r37, $r18, $r60; //
1284  // 55a  add.u32 $r30, $r35, $r57; //
1285  // 55b  mul.lo.u32 $r38, $r80, $r88, 4; //
1286  // 55c  cvt.ru.f32.u32 [v50457], $v66; // id=496 memf+C00
1287  // 55d  [v32]
```

---
CUDA GPU Occupancy Calculator

- Click here for detailed instructions on how to use this occupancy calculator.
- For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graph. The other data points represent the range of possible block sizes, registers counts, and shared memory allocation.

### GPU Occupancy Calculator

<table>
<thead>
<tr>
<th>GPU Occupancy Data</th>
<th>Displayed here in the graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Threads per Multiprocessor</td>
<td>256</td>
</tr>
<tr>
<td>Active Warps per Multiprocessor</td>
<td>8</td>
</tr>
<tr>
<td>Active Thread Blocks per Multiprocessor</td>
<td>1</td>
</tr>
<tr>
<td>Occupancy of each Multiprocessor</td>
<td>33%</td>
</tr>
</tbody>
</table>

### Physical Limits for GPU

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024 WARPS/MULTIPROCESSOR</td>
<td>96</td>
</tr>
<tr>
<td>1024 THREAD BLOCKS/MULTIPROCESSOR</td>
<td>96</td>
</tr>
<tr>
<td>1024 REGISTERS/MULTIPROCESSOR</td>
<td>96</td>
</tr>
<tr>
<td>1024 SHARED MEMORY/MULTIPROCESSOR</td>
<td>96</td>
</tr>
</tbody>
</table>

### Error Checking

- Errors in computing the occupancy data are indicated in red.

- Maximum Thread Blocks Per Multiprocessor: 1024
- Maximum Registors Per Multiprocessor: 1024
- Maximum Shared Memory Per Multiprocessor: 1024
- Maximum Threads Per Multiprocessor: 1024

### CUDA Occupancy Calculator

- GPU Occupancy Calculator
- NVIDIA Corporation

© 2009 NVIDIA CORPORATION
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 examine the effects
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
• 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. CUDA 2.2 now allows binding GMEM directly to textures!
- 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: Texture with floats

• Use texture hardware to promote image to float
• Uses 4x the SMEM, but requires no data type conversion until the write

Results: 6.5 ms

Oops...bad idea!

May be useful for cases where f32 compute is needed
Example 5: 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 6: 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 72x16 pixels into SMEM
Convolution SMEM Reads

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

Step 2: Threads with `threadIdx.y < 2` read bottom two rows
Convolution SMEM Reads

Step 3: Threads with $\text{threadIdx.x} = 15$ read left-top apron pixels
Convolution SMEM Reads

Step 4: Threads with `threadIdx.x == 15` and `threadIdx.y < 2` read left-bottom apron pixels
Convolution SMEM Reads

Step 5: Threads with $\text{threadIdx.x} == 0$
read top-right apron pixels
Convolution SMEM Reads

Step 6: Threads with $\text{threadIdx.x} \equiv 0$ \&\& $\text{threadIdx.y} < 2$ read bottom-right apron pixels
Example 6: 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 72x16 pixels into SMEM

Results: 3.4 ms
Memory only: 1.2 ms

Note: Texture is slightly better, even with all this work
Example 6: Effect of block Size

- 1200 bytes of SMEM per block
- 11 registers
- $16 \times 16 = 2 \text{ blocks} / \text{SM}$
- $16 \times 8 = 5 \text{ blocks} / \text{SM benefit}$

$16 \times 8$ Results:
3.5 ms
No benefit (probably because of increased overlap)

$16 \times 20$ Results:
3.4 ms
Again, no real benefit
Example 6: A Couple Pathological Cases

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

  **19x13 Results:**
  
  5.1 ms (50% decrease)
  
  Memory only: 2.4 ms

- Change image pitch to break coalescing

  **Unaligned Pitch Results:**
  
  4.3 ms
  
  Memory only: 2.4 ms
Example 7: 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

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

SMEM Bank Conflicts causes warp serialization

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>72415</td>
<td>1168</td>
<td>685146</td>
<td>348967</td>
<td>146</td>
</tr>
<tr>
<td>18418</td>
<td>1176</td>
<td>47364</td>
<td>9405</td>
<td>196</td>
</tr>
<tr>
<td>81674</td>
<td>1164</td>
<td>501579</td>
<td>11288</td>
<td>195</td>
</tr>
<tr>
<td>18236</td>
<td>1164</td>
<td>47771</td>
<td>9411</td>
<td>195</td>
</tr>
<tr>
<td>35414</td>
<td>1176</td>
<td>427735</td>
<td>12484</td>
<td>196</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
Bank Addressing Examples

No Bank Conflicts
- Linear addressing, stride == 1

No Bank Conflicts
- Random 1:1 Permutation
Bank Addressing Examples

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

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

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

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

x8
Other Notes About Banked SMEM

- When processing **color** images best to store images as RGBA and load each color plane into a different region of SMEM (tiled instead of interleaved)
- For image processing operations requiring vertical access or transpose, allocate SMEM as 17xN for a 16xN tile
Example 8: 128-bit, Resolve Bank Conflicts

- Have each thread process every 4\textsuperscript{th} 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 10: 128-bit, Unroll Inner Loop

• Mostly memory focused until now
• All code used fast math where possible (e.g. __umul24)
• Unroll Loops

Results: 2.5 ms
Memory only: 0.4 ms
Summary

Convolution Approaches Comparison

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Compute</th>
<th>Memory Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next Global Convolution</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Simple Texture 4-byte Convolution</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>Simple Texture 4-byte Convolution with wrong pitch Convolution</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Global-to-SMIM/32 bit i/w bank optimized convolution</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Global-to-SMIM/128 bit i/w bank optimized convolution</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Global-to-SMIM/128 bit i/w bank optimized convolution</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Global-to-SMIM/32 bit i/w bank optimized convolution</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Global-to-SMIM/128 bit i/w bank optimized convolution</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Global-to-SMIM/32 bit i/w bank optimized convolution</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Global-to-SMIM/128 bit i/w bank optimized convolution</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
GT200 Optimization

- Coalescing buffers greatly improve the performance, especially in non-optimal situations
Apron Reading: GT200 Architecture

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 Architecture

Step 3: Threads with threadIdx.y < 2 read bottom rows

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

- ‘hideous’ 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 to easy programmer pain