GPU TECHNOLOGY CONFERENCE

Convolution Soup:
A case study in CUDA optimization

The Fairmont San Jose | Joe Stam

NVIDIA.

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)

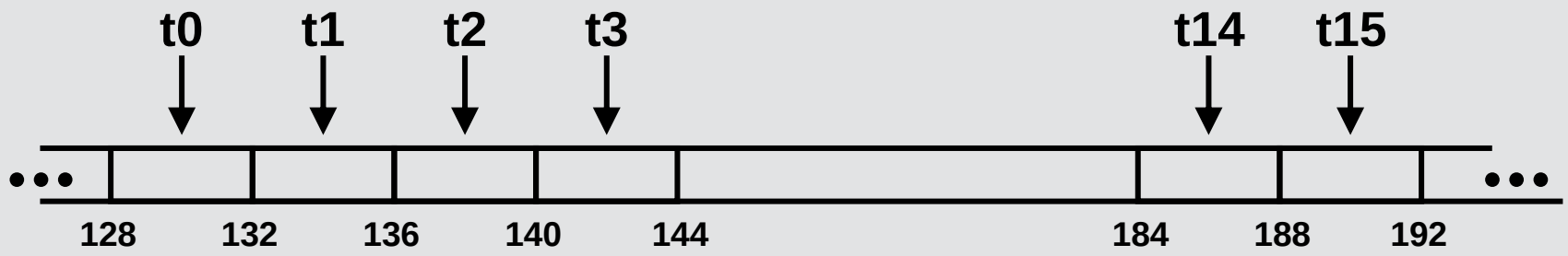
What to Optimize?

- 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

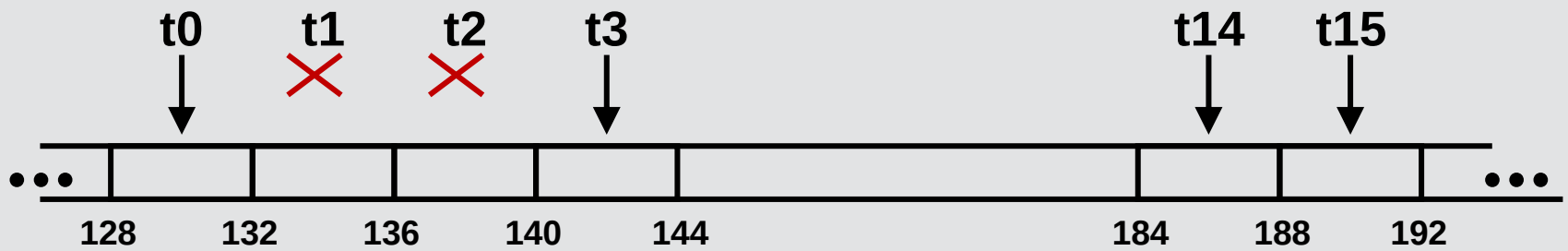
Coalescing GMEM:
Often the most important optimization

- 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 kth thread in a half-warp must access the kth 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)

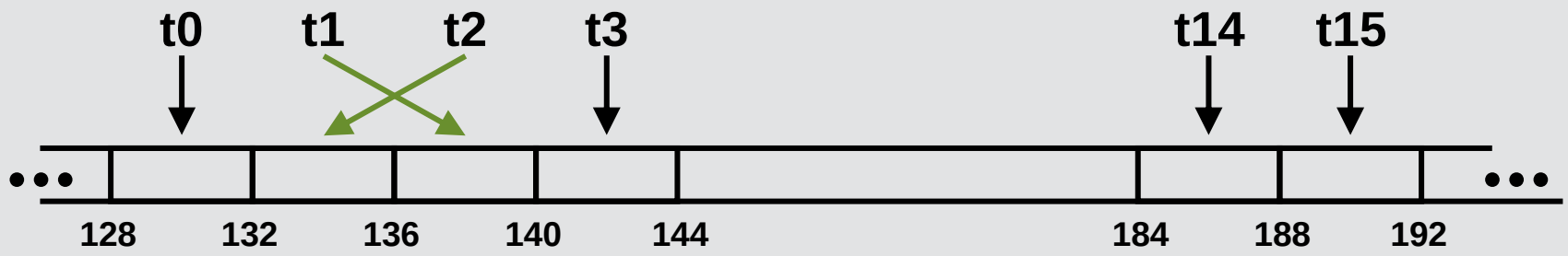


All Threads Participate

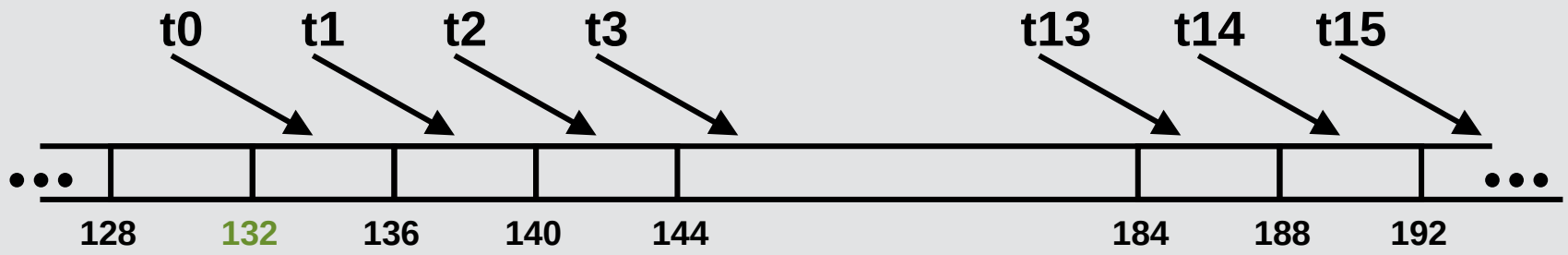


Some Threads Do Not Participate

Uncoalesced Access:
Reading floats (32-bit)



Permuted Access by Threads

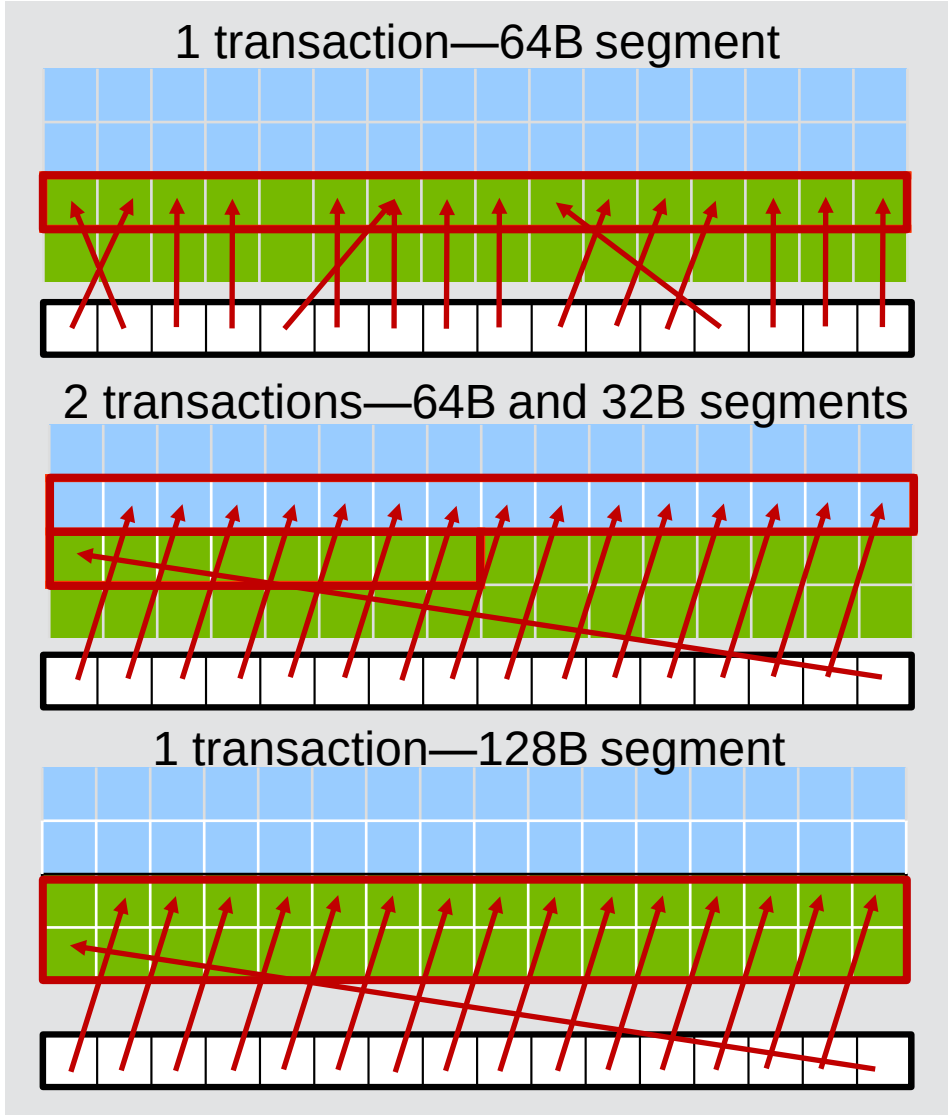


Misaligned Starting Address (not a multiple of 64)

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` complier option)
- Verbose PTXAS output (`--ptxas-options=-v`)

```
code {
    name = _Z28NaiveGlobalConvolutionKernelPhS_jjjf
    lmem = 0
    smem = 48
    reg  = 15
    bar  = 0
    const {
            segname = const
            segnum  = 1
            offset  = 0
            bytes   = 16
        mem {
            0x00000001 0x000003ff 0x0000003c 0x000000ff
        }
    }
    bincode {
        0xa0004c05 0x04200780 0xa0004209 0x04200780
        0x40020205 0x00010780 0xa0000009 0x04000780
        0x20000219 0x04008780 0x30800dfd 0x6440c7c8
        0x30000003 0x00000280 0x213eec0d 0xffffffff
```

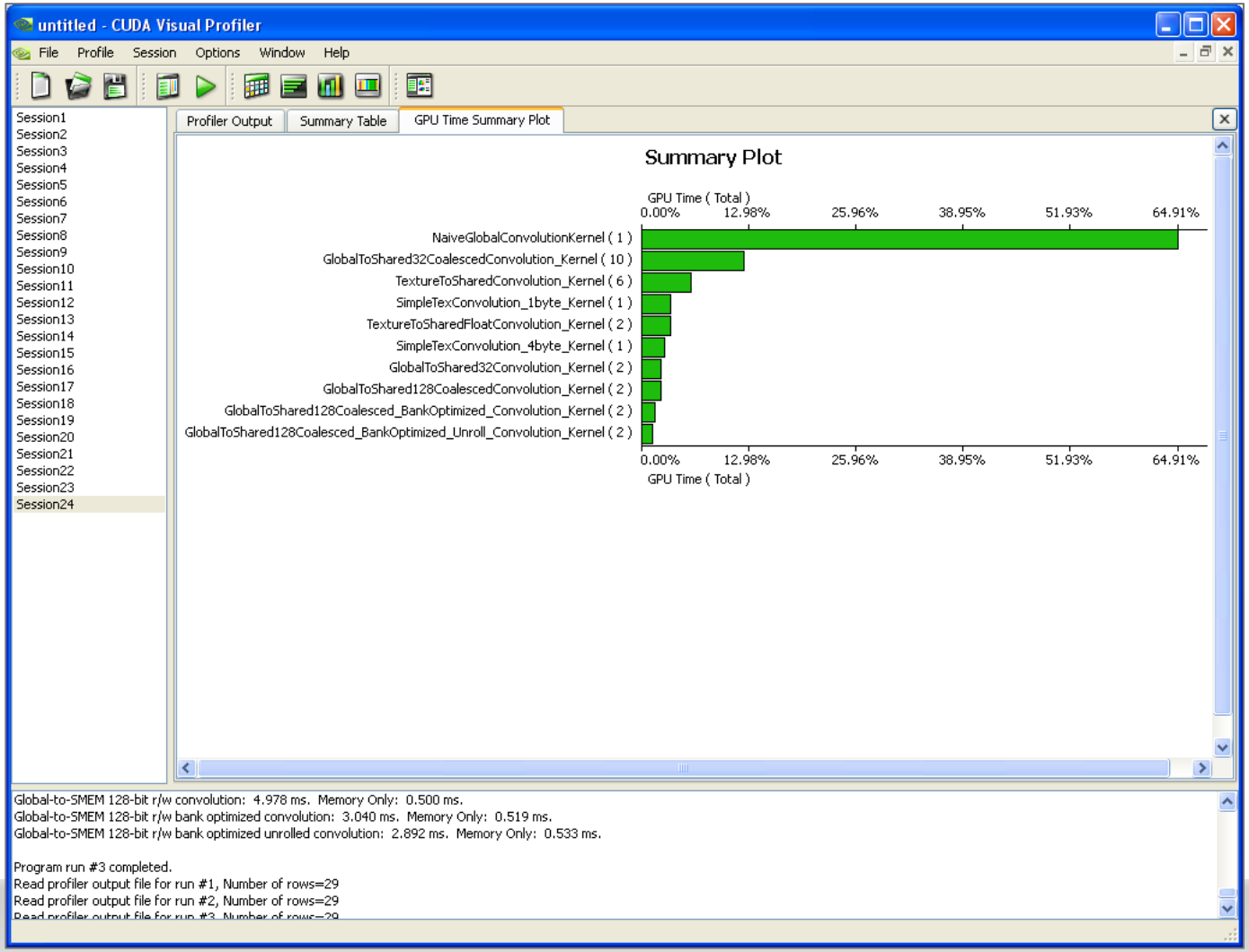
PTX—GPU intermediate assembly

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

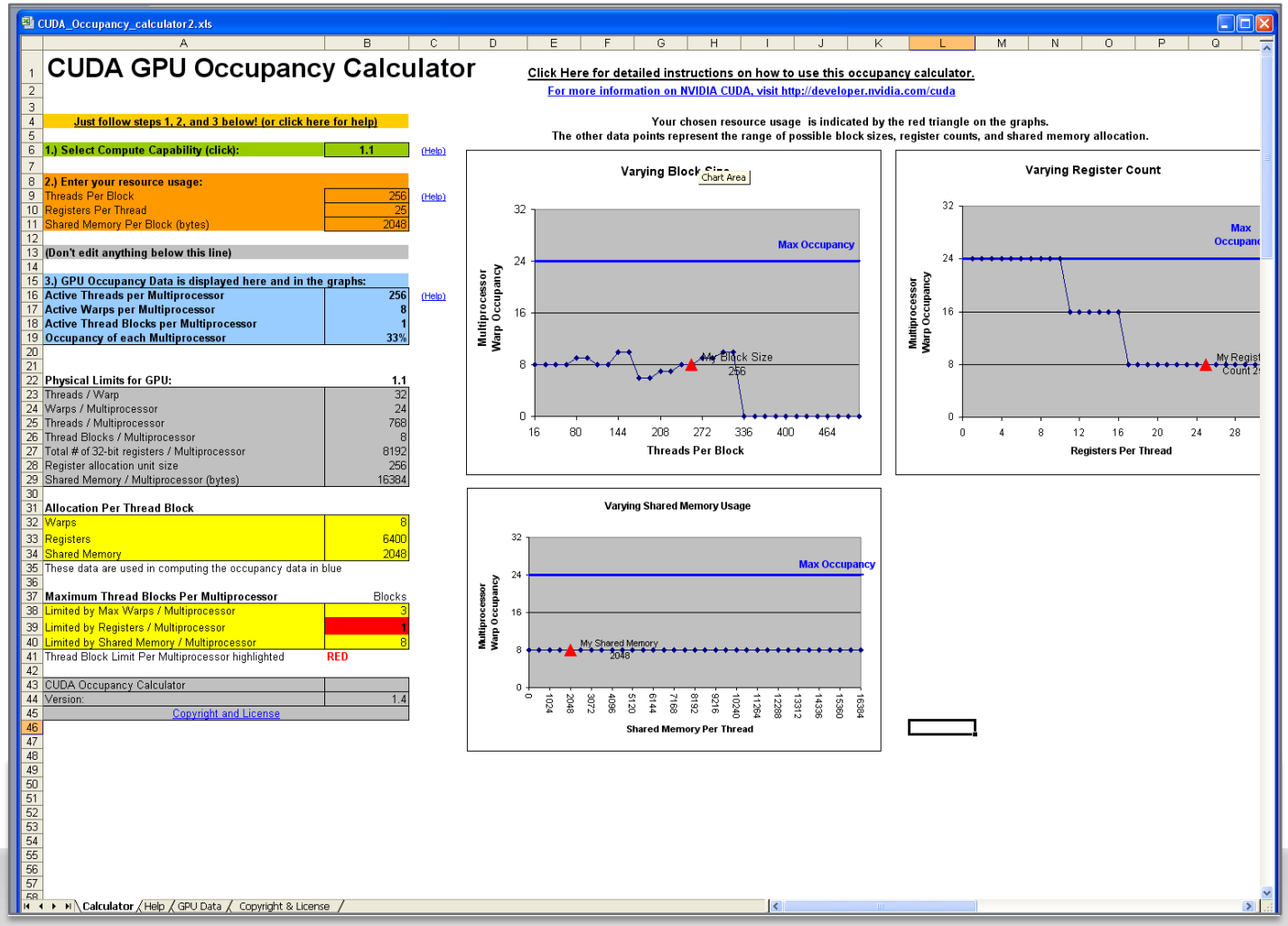
```
--opencc-options -LIST:source=on
```

```
1264      st.shared.u32    [%r63+12], %r78;     // id:493 smemf+0x0
1265      @!%p1 bra    $Lt_4_77;            //
1266      add.u32      %r79, %r2, 1;            //
1267      mul24.lo.u32     %r80, %r18, %r79;    //
1268      add.u32      %r81, %r8, %r80;         //
1269      cvt.rn.f32.u32   %f53, %r81;      //
1270      mov.f32      %f54, 0fc0000000;        // -2
1271      add.f32      %f55, %f53, %f54;        //
1272      mov.f32      %f56, %f36;              //
1273      mov.f32      %f57, 0f00000000;        // 0
1274      mov.f32      %f58, 0f00000000;        // 0
1275      tex.2d.v4.u32.f32 {%r82,%r83,%r84,%r85},[normFloatTex,{%f55,%f56,%f57,%f58}];
1276      .loc     2    556 0
1277   // 552          if(threadIdx.x < 4)
1278   // 553          {
1279   // 554
1280   // 555              sidx = __umul24(blockDim.y+threadIdx.y,smem_pitch) + bDimX4 + threadIdx.x;
1281   // 556               smemf[sidx] = tex2D(normFloatTex,(float)(__umul24(bDimX4,blockIdx.x+1)+threadIdx.x)-2.0f,  Ytex);
1282      mov.s32      %r86, %r82;              //
1283      add.u32      %r87, %r18, %r60;        //
1284      add.u32      %r88, %r8, %r87;         //
1285      mul.lo.u32   %r89, %r88, 4;       //
1286      add.u32      %r90, %r1, %r89;         //
1287      st.shared.u32    [%r90+0], %r86; // id:494 smemf+0x0
1288   $Lt_4_77:
```

Visual Profiler



GPU TECHNOLOGY CONFERENCE



Occupancy Calculator Spreadsheet

NVIDIA.

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

```
__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

Results: 7.8 ms

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

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!

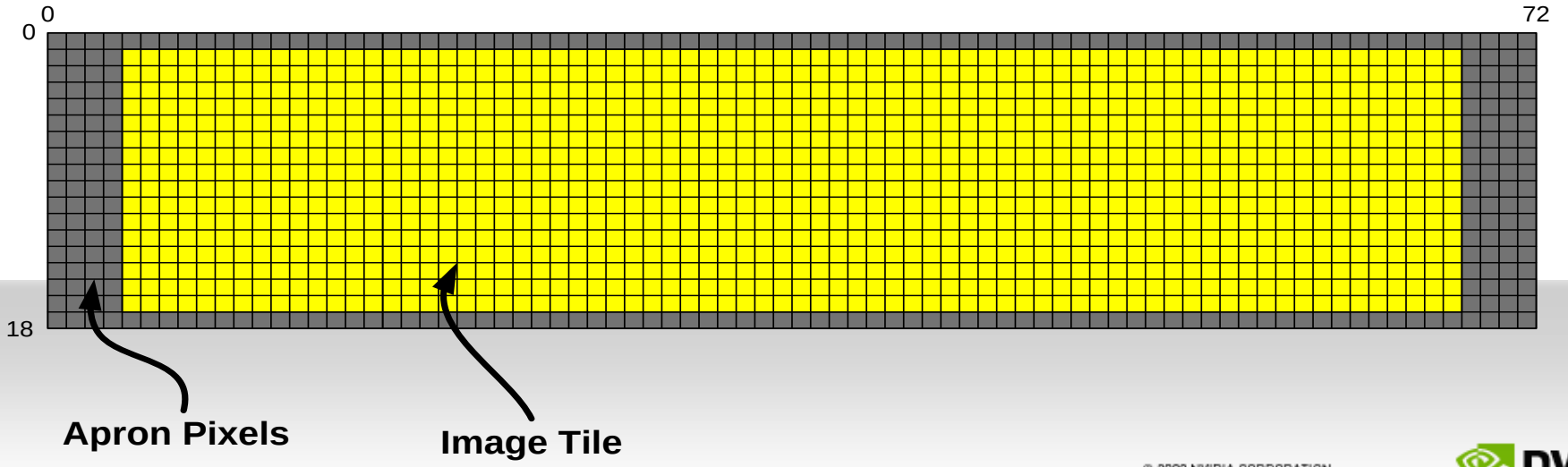
Results: 3.6 ms

Memory only: 1.6 ms

Slightly worse than using textures

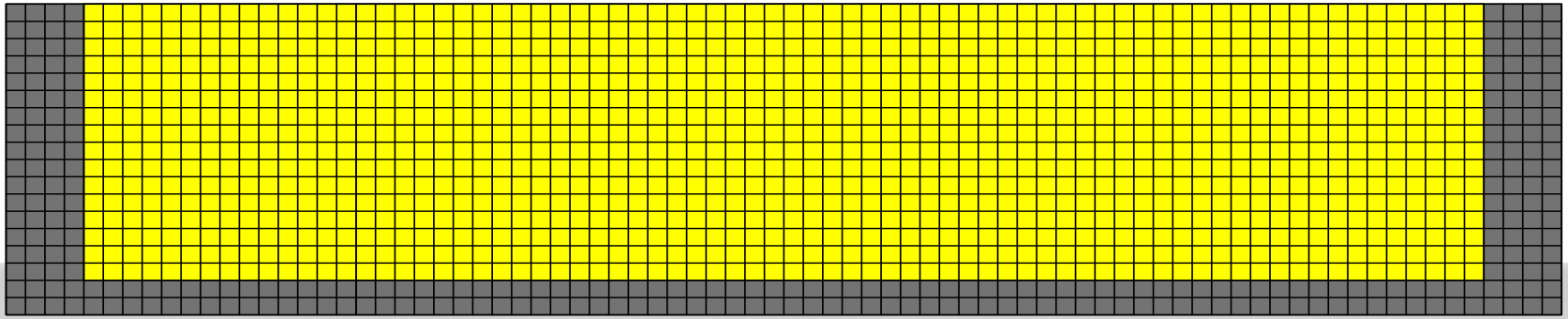
Example 6: GMEM to SMEM Strict Coalescing

- 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



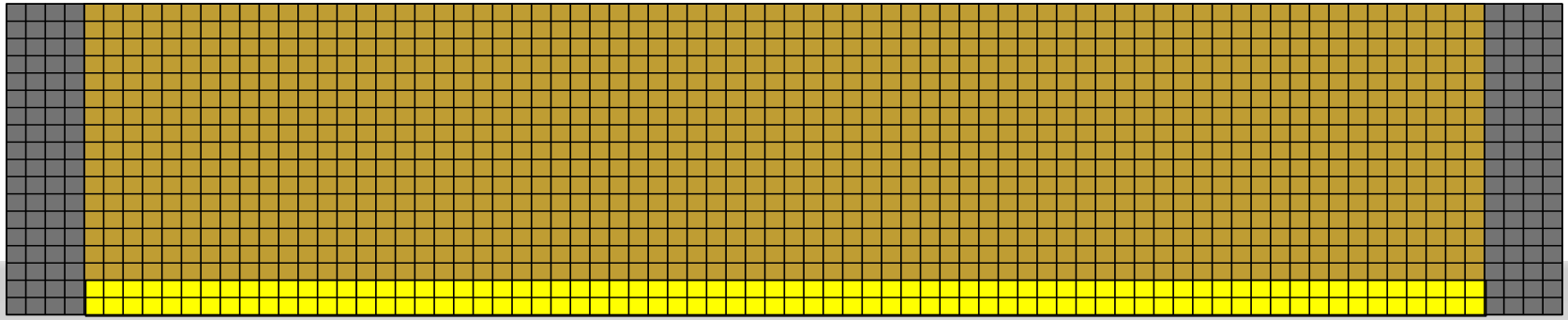
Convolution SMEM Reads

Step 1: All threads read center top portion into memory



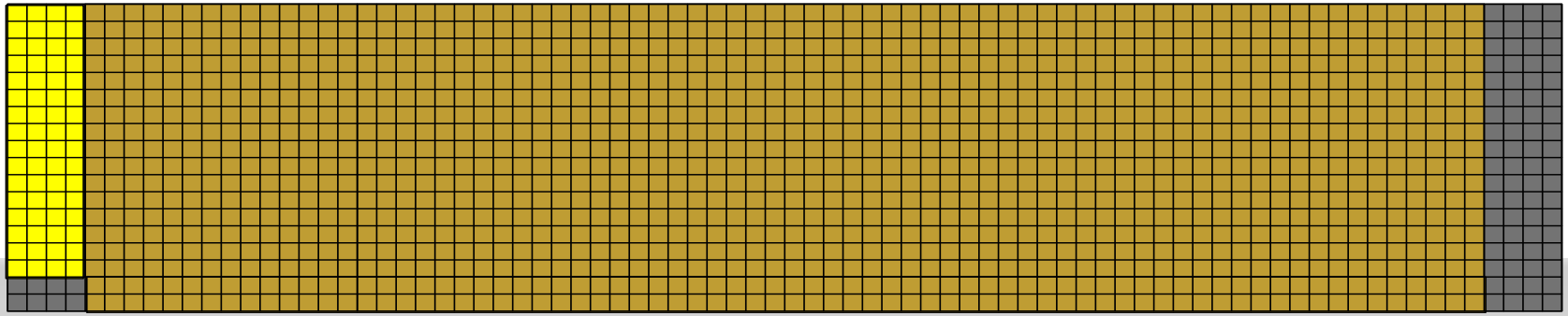
Convolution SMEM Reads

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



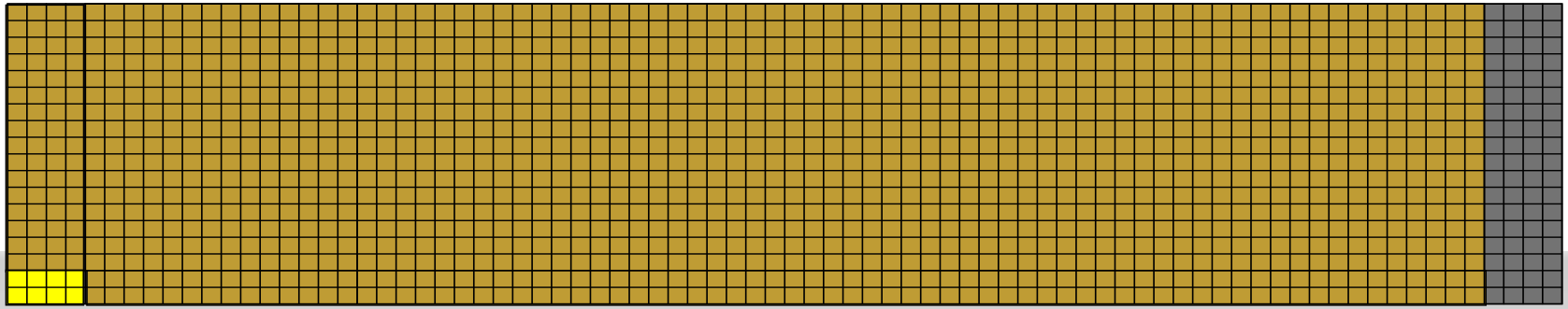
Convolution SMEM Reads

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



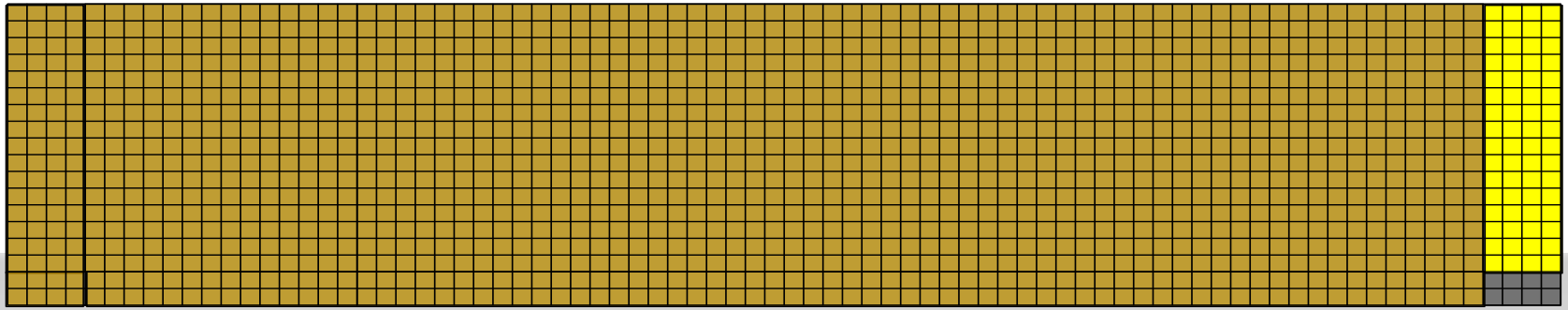
Convolution SMEM Reads

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



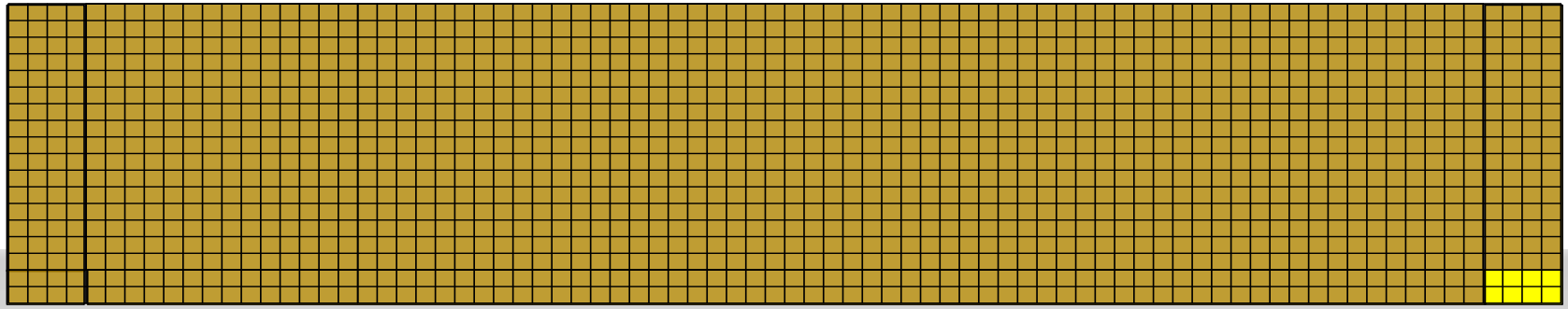
Convolution SMEM Reads

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



Convolution SMEM Reads

Step 6: Threads with `threadIdx.x == 0 && 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
- 16x16 = 2 blocks / SM
- 16x8 = 5 blocks / SM benefit

16x8 Results: 3.5 ms

No benefit (probably because of increased overlap)

16x20 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…

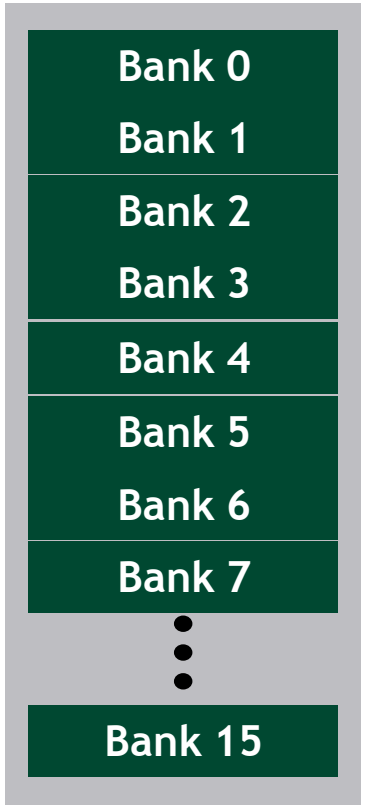
The Answer

e	branch	divergent branch	instructions	warp serialize	cta launched
	15767	1168	45235	8591	146
	72415	1168	685146	348967	146
	18418	1176	47364	9465	196
	81674	1164	501579	11288	195
	18236	1164	47771	9411	195
	35414	1176	427735	12484	196

SMEM Bank Conflicts causes warp serialization

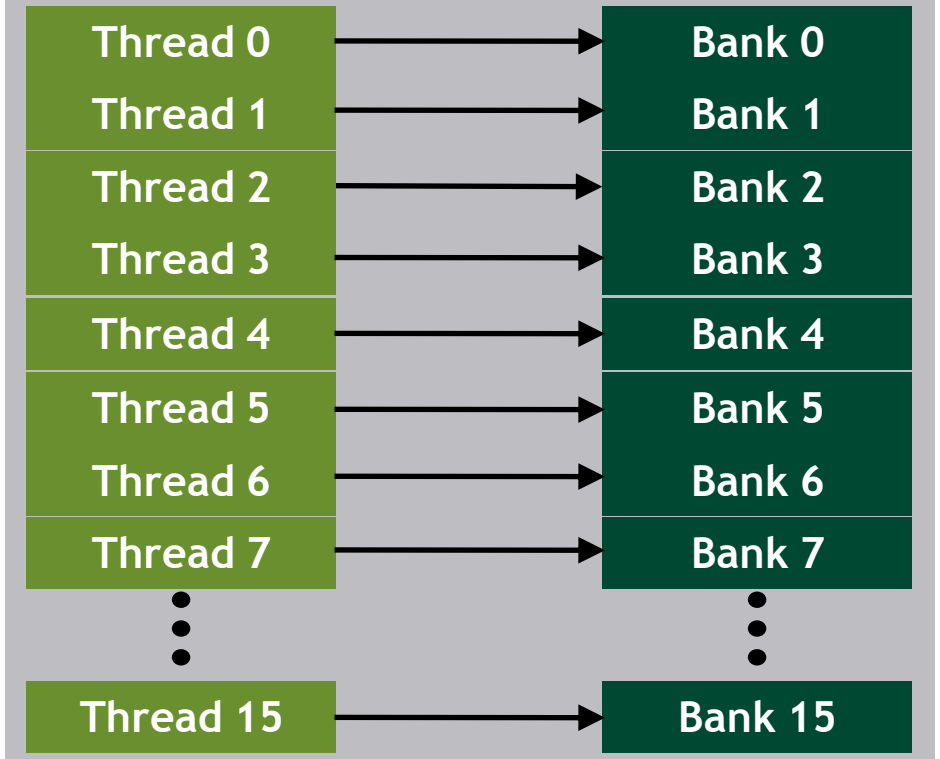
Banked SMEM Architecture

- 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

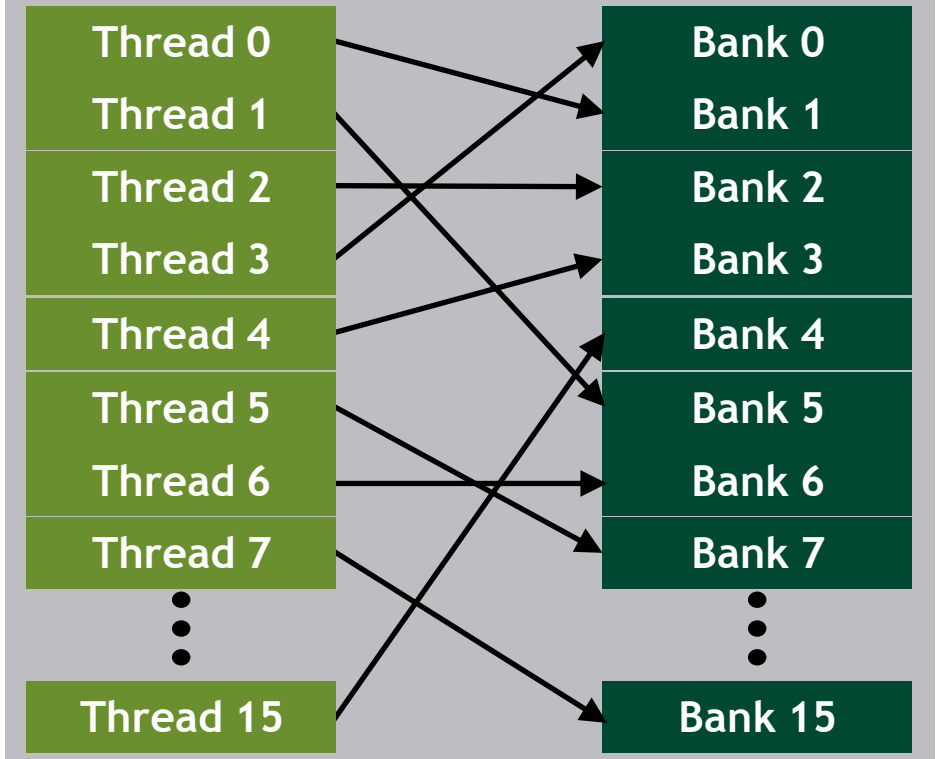


Bank Addressing Examples

No Bank Conflicts
- Linear addressing, stride == 1

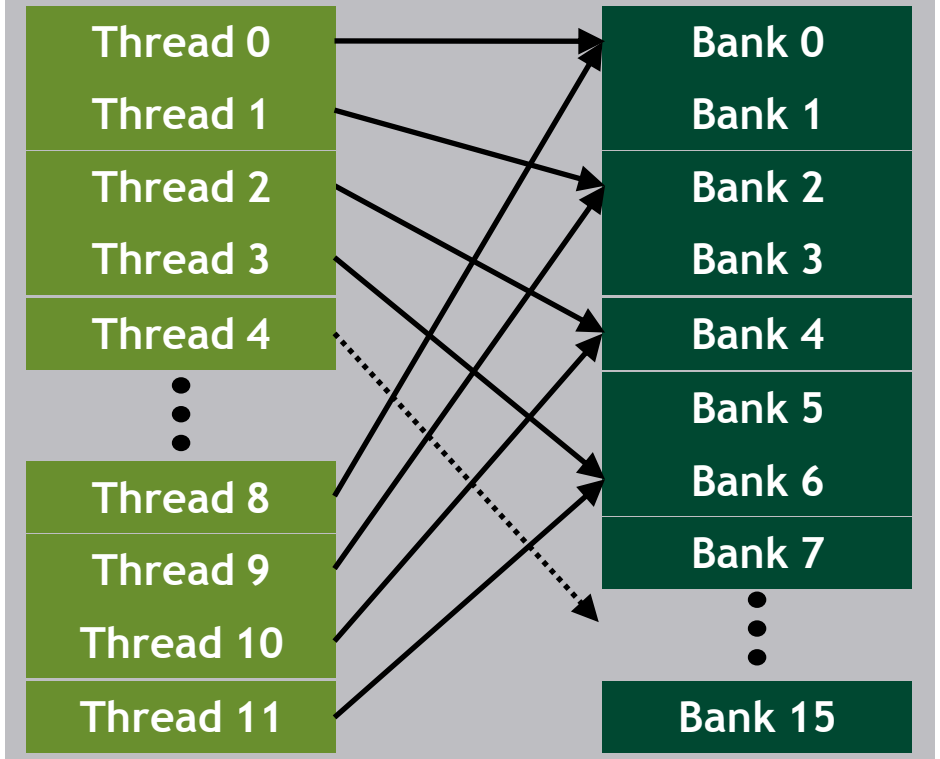


No Bank Conflicts
- Random 1:1 Permutation

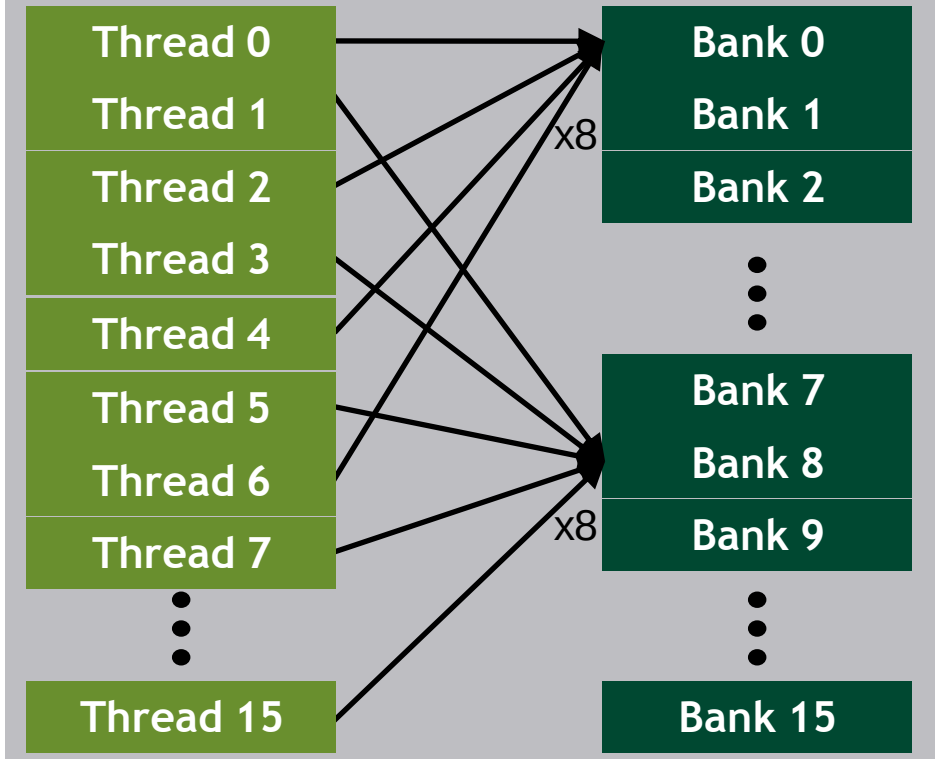


Bank Addressing Examples

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



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



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

17 X N SMEM bank layout

SMEM Columns

SMEM Rows

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	1
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	1	2
3	4	5	6	7	8	9	10	11	12	13	14	15	16	1	2	3
4	5	6	7	8	9	10	11	12	13	14	15	16	1	2	3	4
5	6	7	8	9	10	11	12	13	14	15	16	1	2	3	4	5
6	7	8	9	10	11	12	13	14	15	16	1	2	3	4	5	6
7	8	9	10	11	12	13	14	15	16	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15	16	1	2	3	4	5	6	7	8

Dummy Column

Example 8: 128-bit, Resolve Bank Conflicts

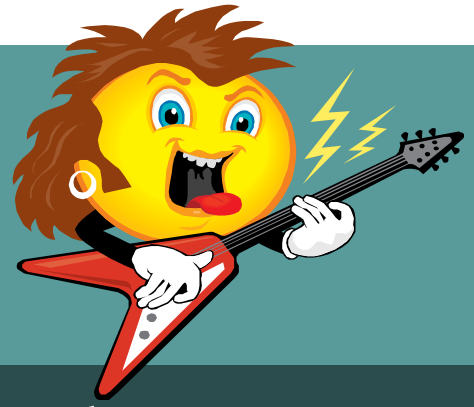
- Have each thread process every 4th 32-bit word
- Intermediate results are stored in SMEM
  - Need to shrink the block size since this uses more SMEM

Results: 2.9 ms!

Memory only: 0.4 ms

Example 10: 128-bit, Unroll Inner Loop

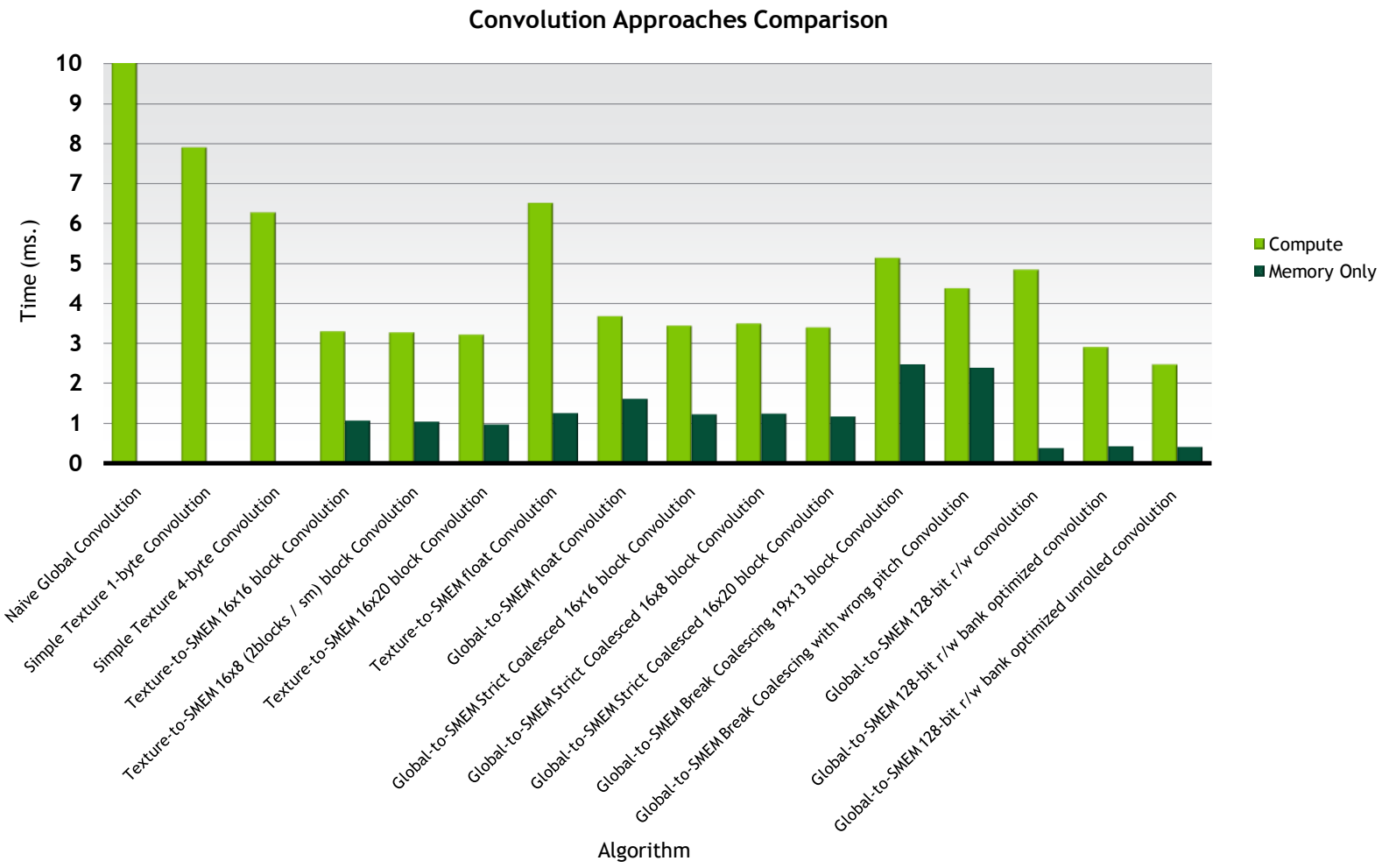
- Mostly memory focused until now
- All code used fast math where possible (e.g. `__umul24`)
- Unroll Loops



Results: 2.5 ms

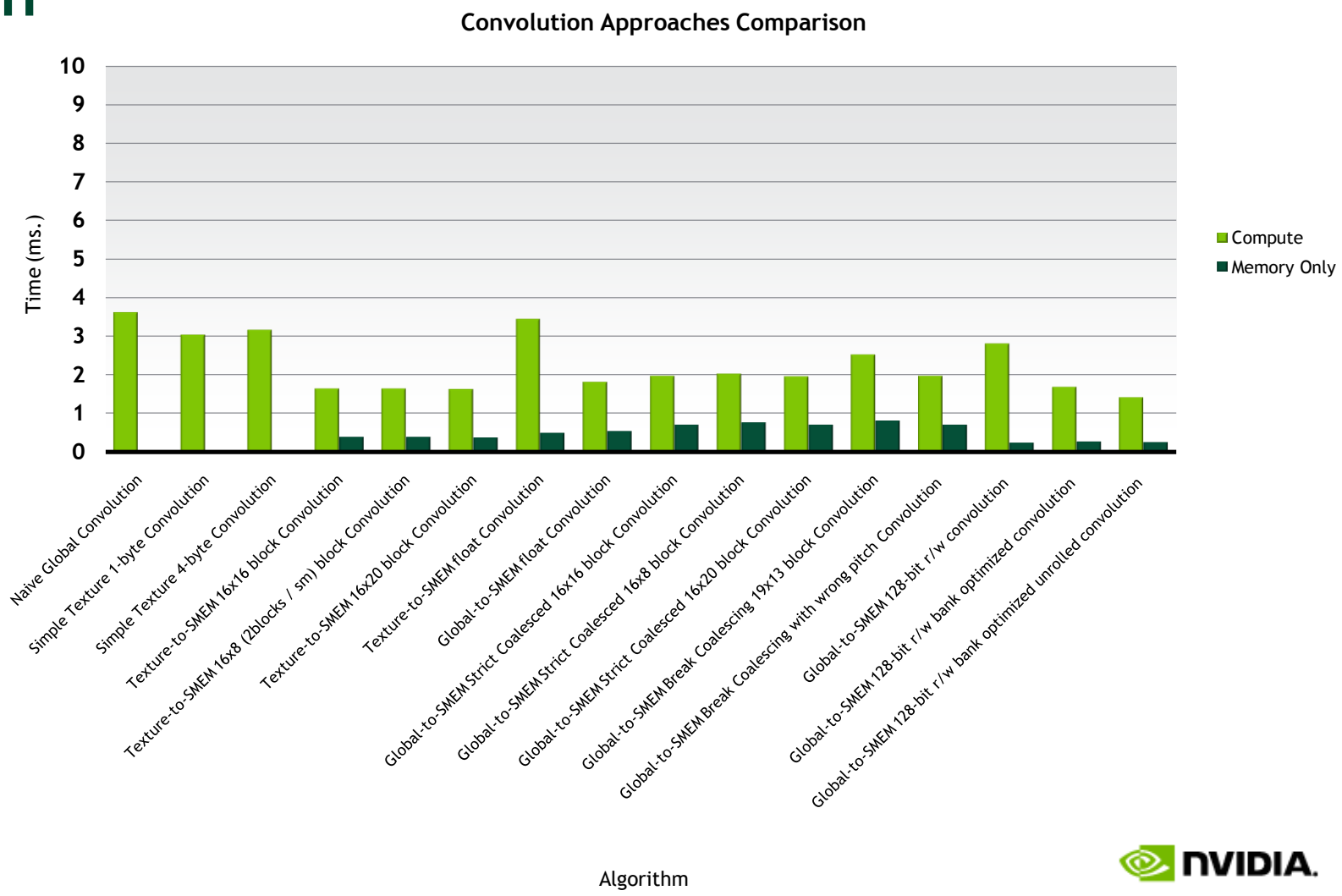
Memory only: 0.4 ms

Summary



GT200 Optimization

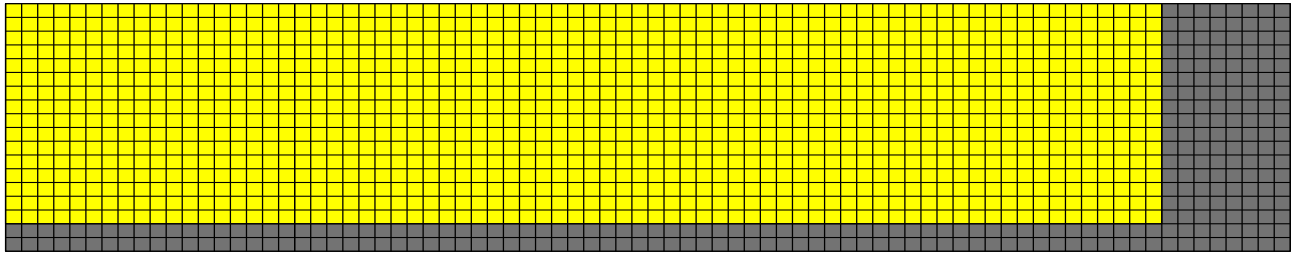
- Coalescing buffers greatly improve the performance, especially in non-optimal situations



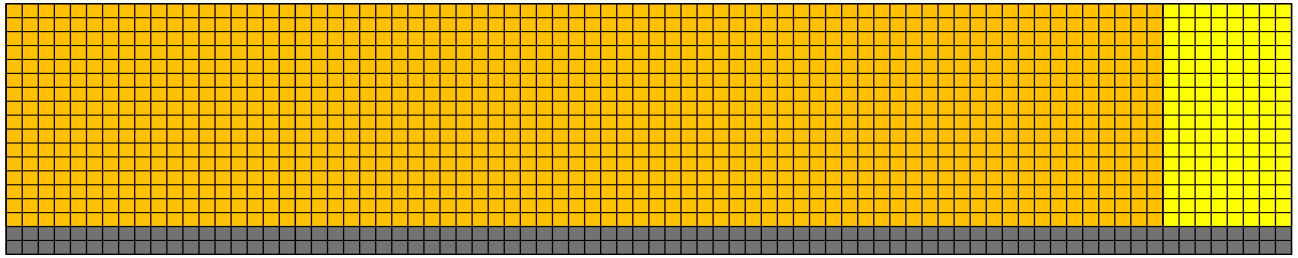
Apron Reading: GT200 Architecture

Coalescing buffers greatly simplify read patterns

Step 1: All threads read, shift up and left

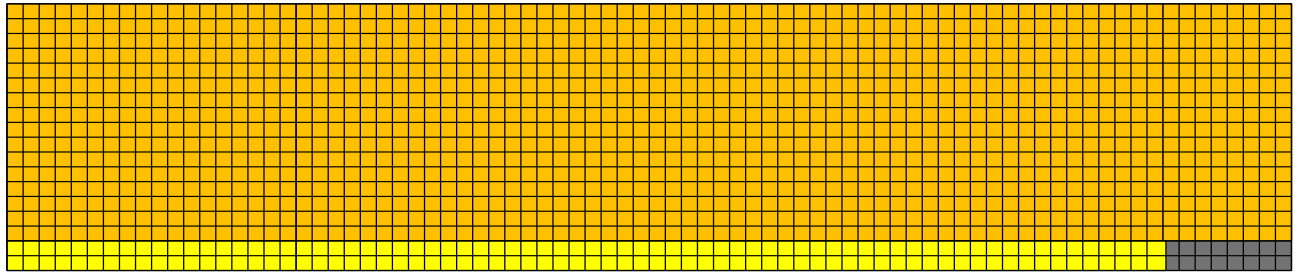


Step 2: Threads with `threadIdx.x < 2` read right columns

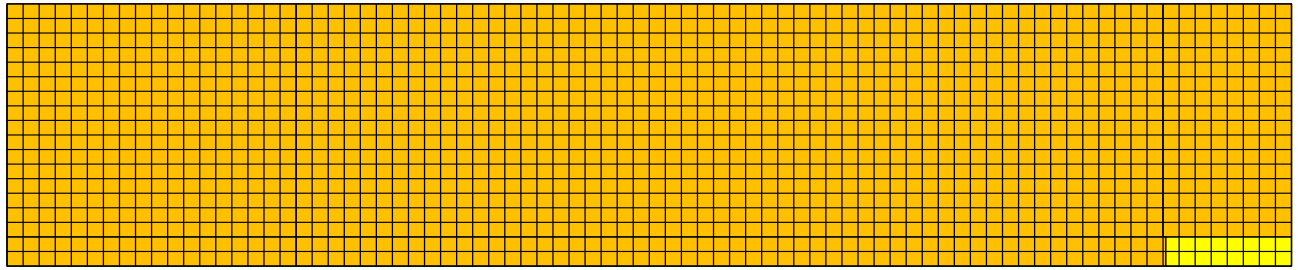


Apron Reading: GT200 Architecture

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