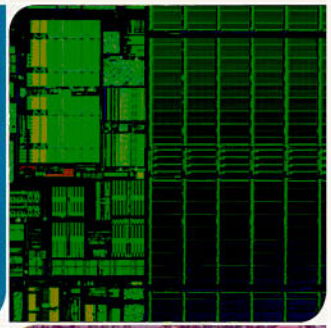
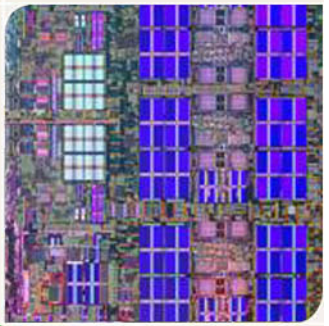
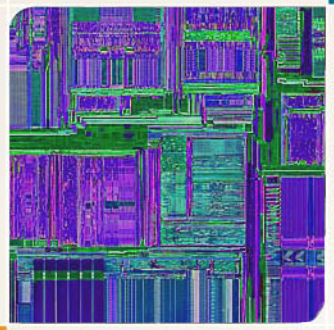
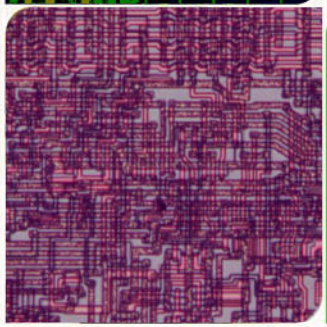
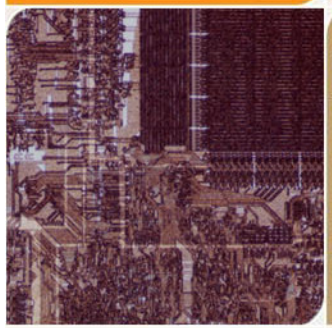





CS4803DGC Design Game Consoles

Spring 2009

Prof. Hyesoon Kim

Georgia Tech College of Computing



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

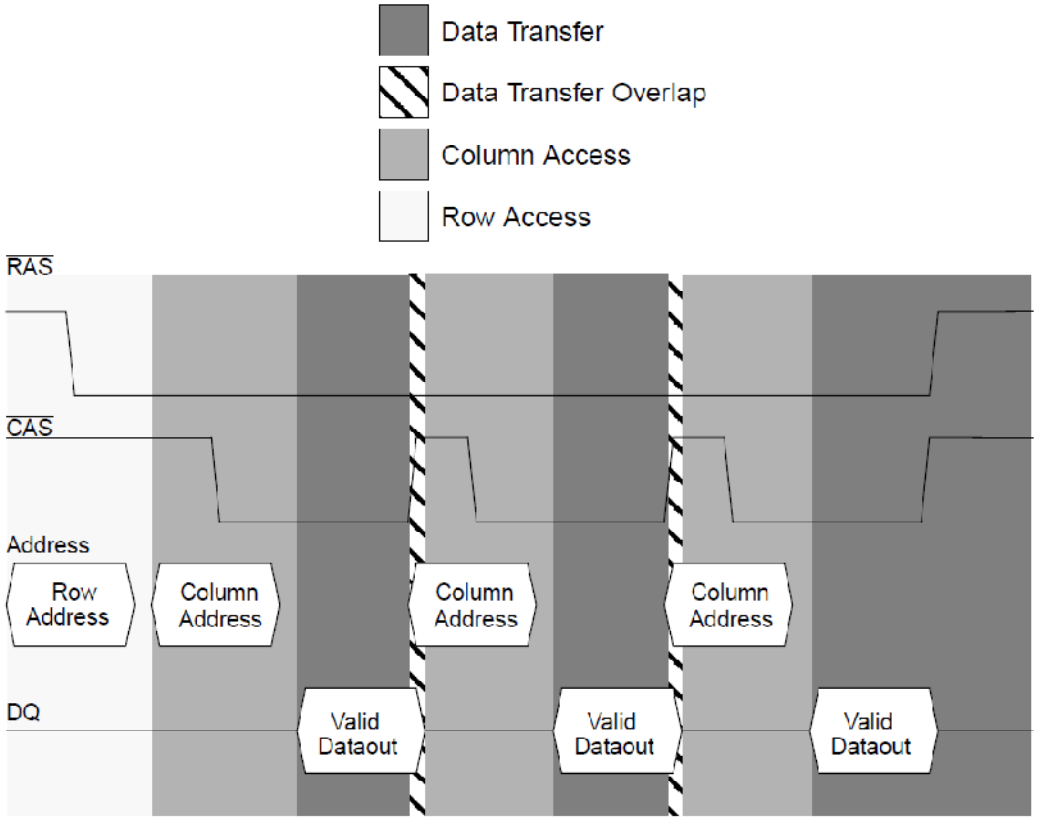




Coalescing

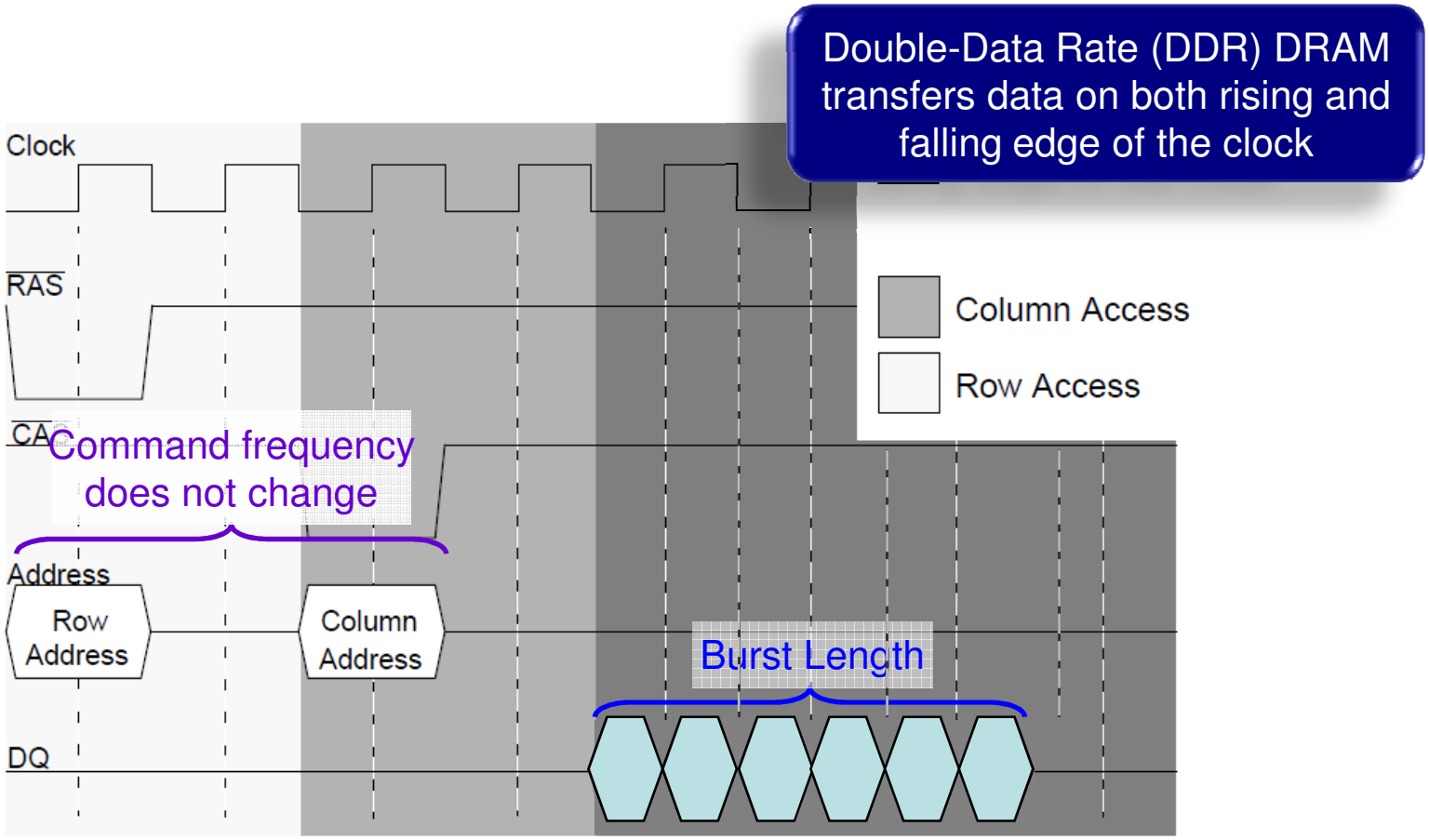
- A coordinated read by a warp
- A contiguous region of global memory:
  - 128 bytes - each thread reads a word: int, float, …
  - 256 bytes - each thread reads a double-word: int2, float2, …
  - 512 bytes – each thread reads a quad-word: int4, float4, …
- Additional restrictions:
  - Starting address for a region must be a multiple of region size
  - The *k*th thread in a warp must access the *k*th element in a block being read
- Exception: not all threads must be participating
  - Predicated access, divergence within a warp

DRAM Read Timing



Accesses are asynchronous: triggered by RAS and CAS signals, which can in theory occur at arbitrary times (subject to DRAM timing constraints)

SDRAM Read Timing



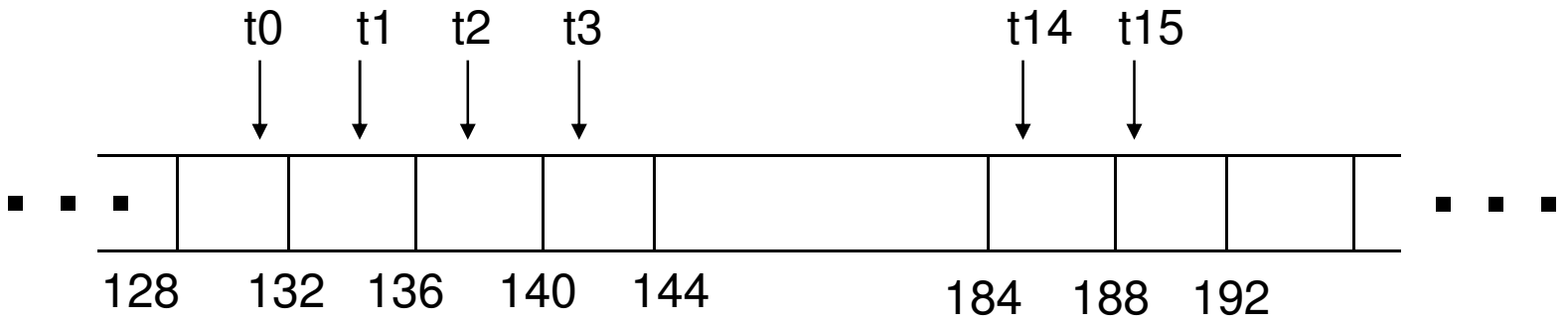
Timing figures taken from "A Performance Comparison of Contemporary DRAM Architectures" by Cuppu, Jacob, Davis and Mudge



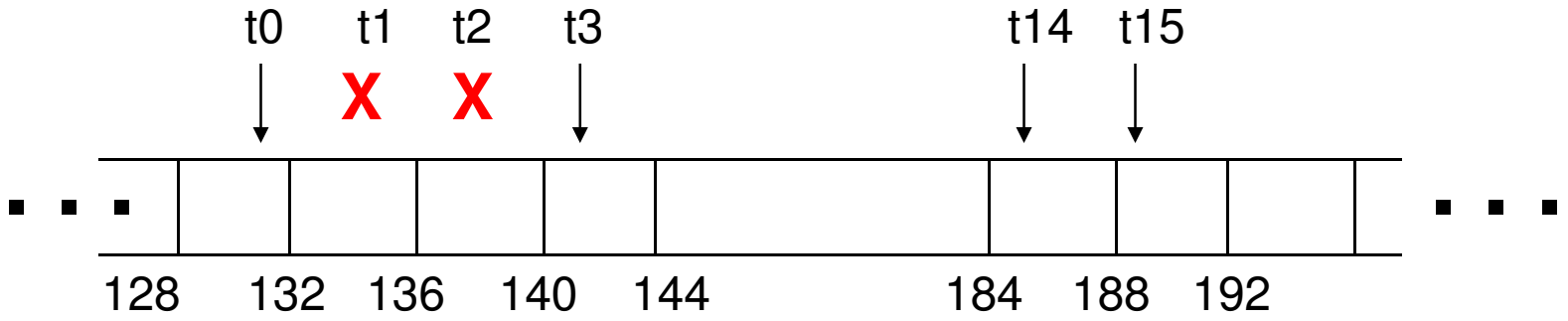
Burst Mode

- Coalesced Access is taking advantage of burst mode in the GPU

Coalesced Access: Reading floats

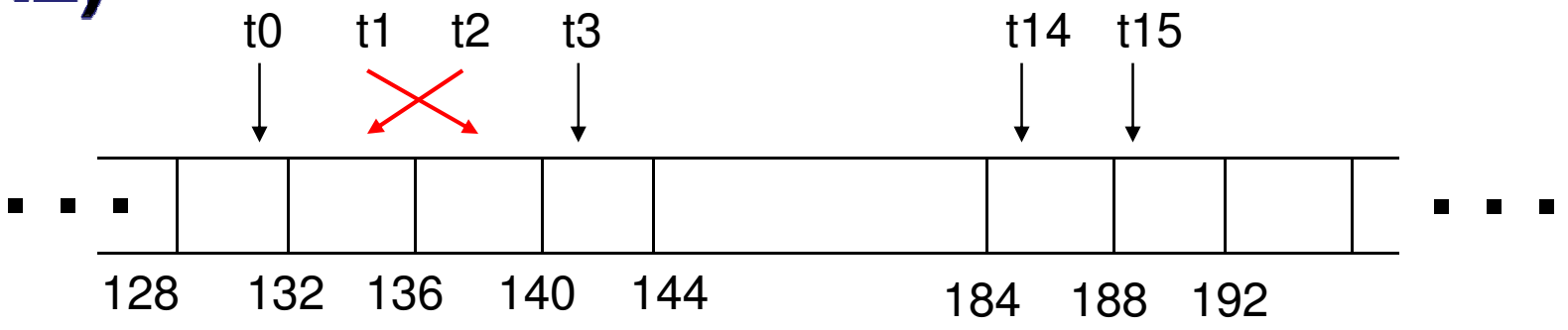


All threads participate

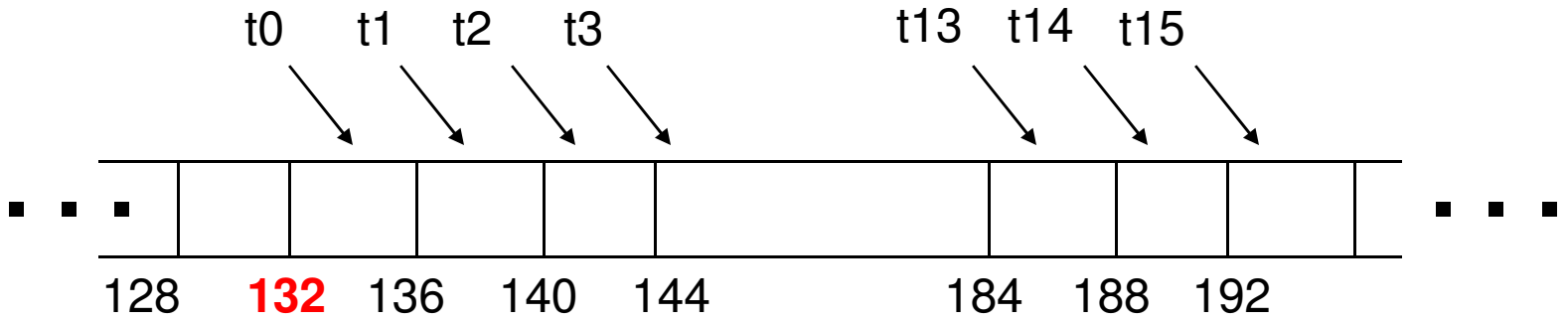


Some threads do not participate

Uncoalesced Access: Reading floats (Computing Capability <1.2)



Permuted Access by Threads



Misaligned Starting Address (not a multiple of 64)

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



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



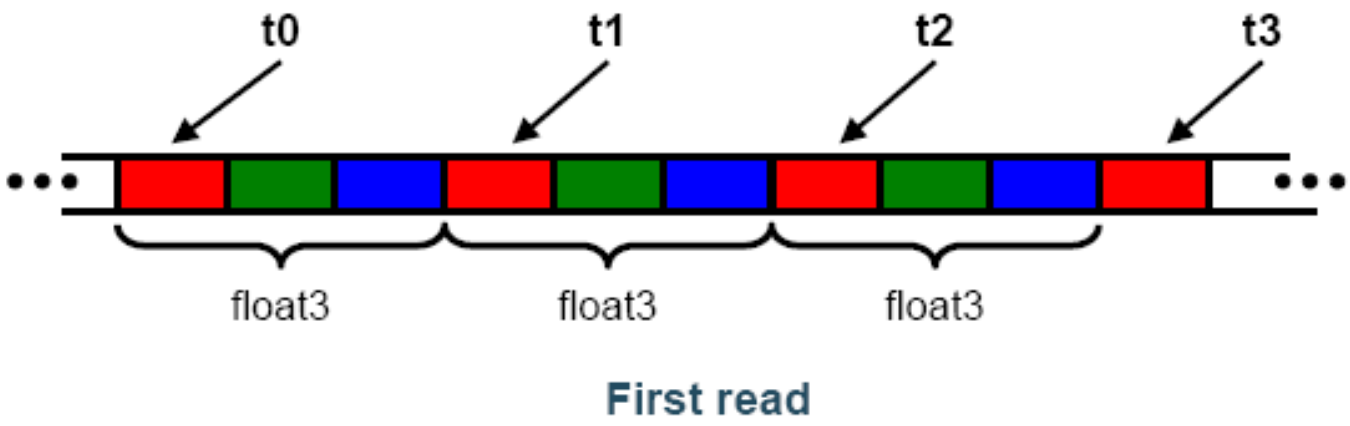
Uncoalesced float3 Code

```
__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 Access: float3 Case

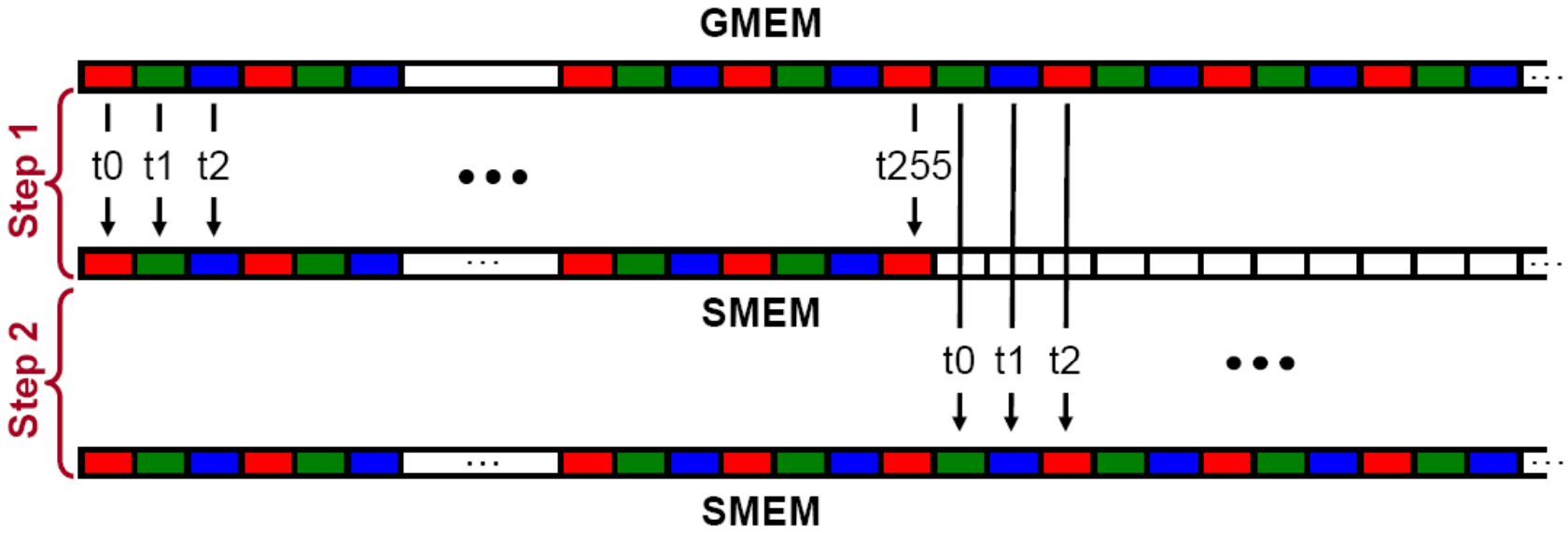


- float3 is 12 bytes
- Each thread ends up executing 3 reads
  - sizeof(float3) ≠ 4, 8, or 12
  - Half-warp reads three 64B non-contiguous regions



First read

Coalescing float3 Access



Similarly, Step3 starting at offset 512



Coalesced Access: float3 Case

Read the input through SMEM

Compute code is not changed

Write the result through SMEM

```
__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)
  - {x1,y1, z1,w1} , {x2,y2, z2,w2} , {x3,y3, z3,w3} , {x4,y4, z4,w4} ….
  - Intuitive but less efficient
  - What if we want to perform only x axis?
- Structure of array (SOA)
  - {x1,x2,x3,x4}, …,{y1,y2,y3,y4}, …{z1,z2,z3,z4}, … {w1,w2,w3,w4}…



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
- 4K blocks x 256 threads:
  - 3,302µs – float3 uncoalesced
  - 359µs – float3 coalesced through shared 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 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



Occupancy != Performance

- 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)



Grid/Block Size Heuristics

- # of blocks / # of multiprocessors > 1
  - So all multiprocessors have at least one block to execute
- Per-block resources at most half of total available
  - Shared memory and registers
  - Multiple blocks can run concurrently in a multiprocessor
  - If multiple blocks coexist that aren't all waiting at a __syncthreads(), machine can stay busy
- # of blocks / # of multiprocessors > 2
  - So multiple blocks run concurrently in a multiprocessor
- # of blocks > 100 to scale to future devices
  - Blocks stream through machine in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations



Use Occupancy calculator

- Part of the SDK



- Image Convolution with CUDA
  - White Documents from Nvidia website



Convolution?

Input

23	12	25	36	10
73	26	99	56	2
65	11	5	26	76
83	67	52	32	17
34	84	46	99	32

Kernel

1	0	1
0	1	0
1	0	1



Convolution

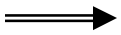
23	12	25	36	10
73	26	99	56	2
65	11	5	26	76
83	67	52	32	17
34	84	46	99	32

26	99	56
11	5	26
67	52	32

*

1	0	1
0	1	0
1	0	1

(26 * 1) +
(99 * 0) +
(56 * 1) +
(11 * 0) +
(5 * 1) +
(26 * 0)+
(67 * 1) +
(52 * 0) +
(32 * 1)



23	12	25	36	10
73	26	99	56	2
65	11	18	26	76
83	67	52	32	17
34	84	46	99	32

Boundary

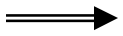
23	12	25	36	10
73	26	99	56	2
65	11	5	26	76
83	67	52	32	17
34	84	46	99	32

0	0	0
0	23	12
0	73	26

*

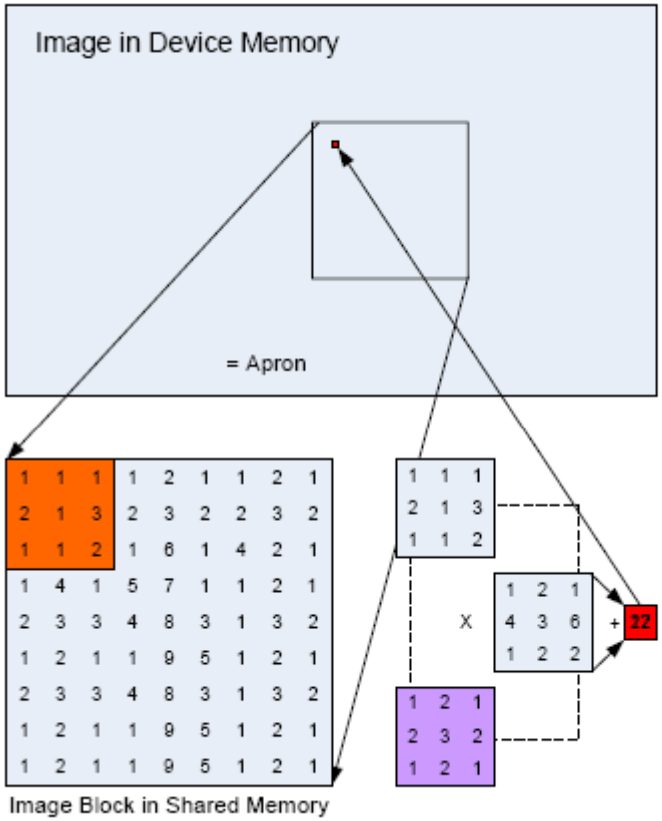
1	0	1
0	1	0
1	0	1

(0 * 1) +
(0 * 0) +
(0 * 1) +
(0 * 0) +
(23 * 1) +
(12 * 0)+
(0 * 1) +
(73 * 0) +
(26 * 1)



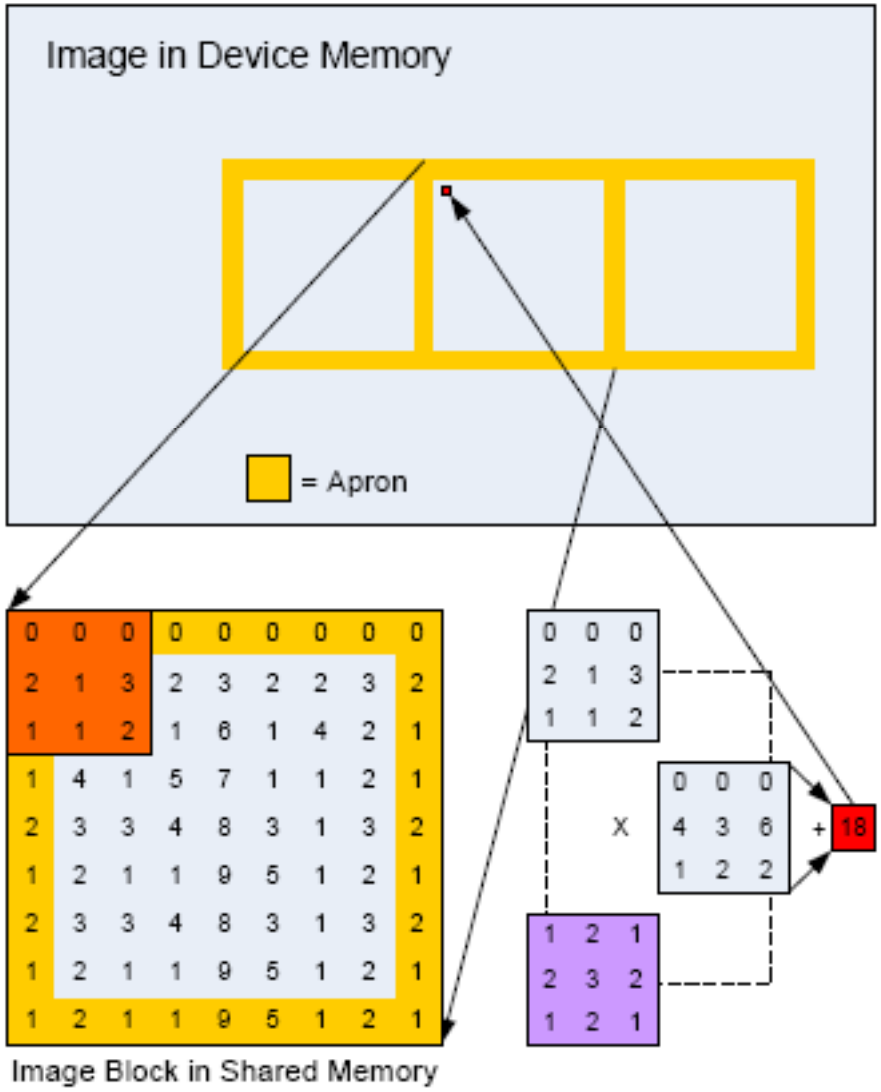
23	12	25	36	10
73	26	99	56	2
65	11	49	26	76
83	67	52	32	17
34	84	46	99	32

A Naïve Implementation



A naïve convolution algorithm. A block of pixels from the image is loaded into an array in shared memory. To process and compute an output pixel (red), a region of the input image (orange) is multiplied element-wise with the filter kernel (purple) and then the results are summed. The resulting output pixel is then written back into the image.

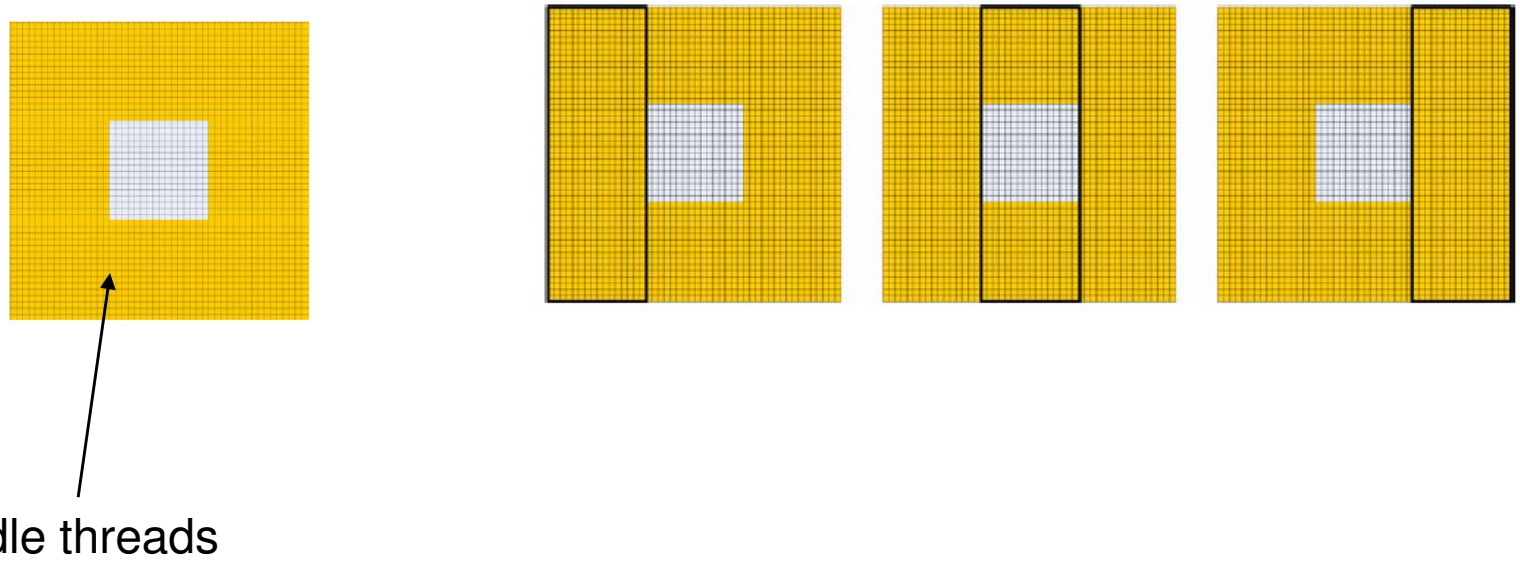
Naïve Implementation: Shared Memory and the Apron



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





Optimizations

- Memory Coalescing: If all threads within a warp (32 threads) simultaneously read consecutive words then single large read of the 32 values can be performed at optimum speed. If 32 random addresses are read, then only a fraction of the total DRAM bandwidth can be achieved, and performance will be much lower.
- Unrolling the kernel



Loop Unrolling

- #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.