CUDA Performance Considerations (2 of 2)

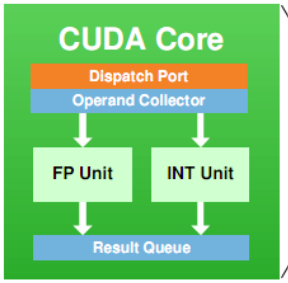
Varun Sampath
Original Slides by Patrick Cozzi
University of Pennsylvania
CIS 565 - Spring 2012

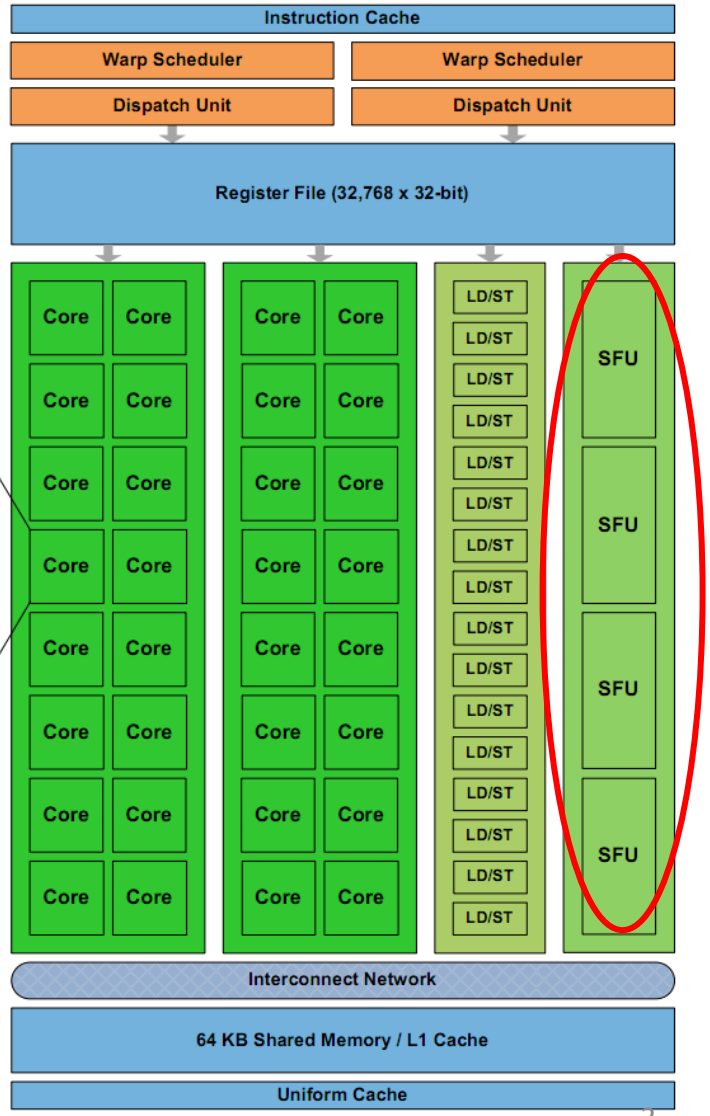
Agenda

- Instruction Optimizations
  - Mixed Instruction Types
  - Loop Unrolling
  - Thread Granularity
- Memory Optimizations
  - Shared Memory Bank Conflicts
  - Partition Camping
  - Pinned Memory
- Streams

Mixed Instructions

- Special Function Units (SFUs)
  - Use to compute `__sinf()`, `__expf()`
  - Only 4, each can execute 1 instruction per clock





Fermi Streaming Multiprocessor (SM)

Image: NVIDIA Fermi Whitepaper

Loop Unrolling

```
for (int k = 0; k < BLOCK_SIZE; ++k)
{
   Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

- Instructions per iteration
  - One floating-point multiply
  - One floating-point add
  - What else?

Loop Unrolling

```
for (int k = 0; k < BLOCK_SIZE; ++k)
{
   Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

- Other instructions per iteration
  - Update loop counter

Loop Unrolling

```
for (int k = 0; k < BLOCK_SIZE; ++k)
{
   Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

- Other instructions per iteration
  - Update loop counter
  - Branch

Loop Unrolling

```
for (int k = 0; k < BLOCK_SIZE; ++k)
{
   Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

- Other instructions per iteration
  - Update loop counter
  - Branch
  - Address arithmetic

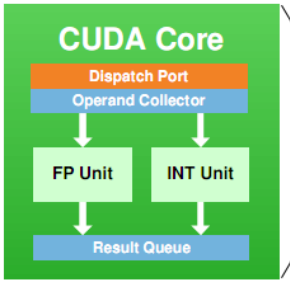
Loop Unrolling

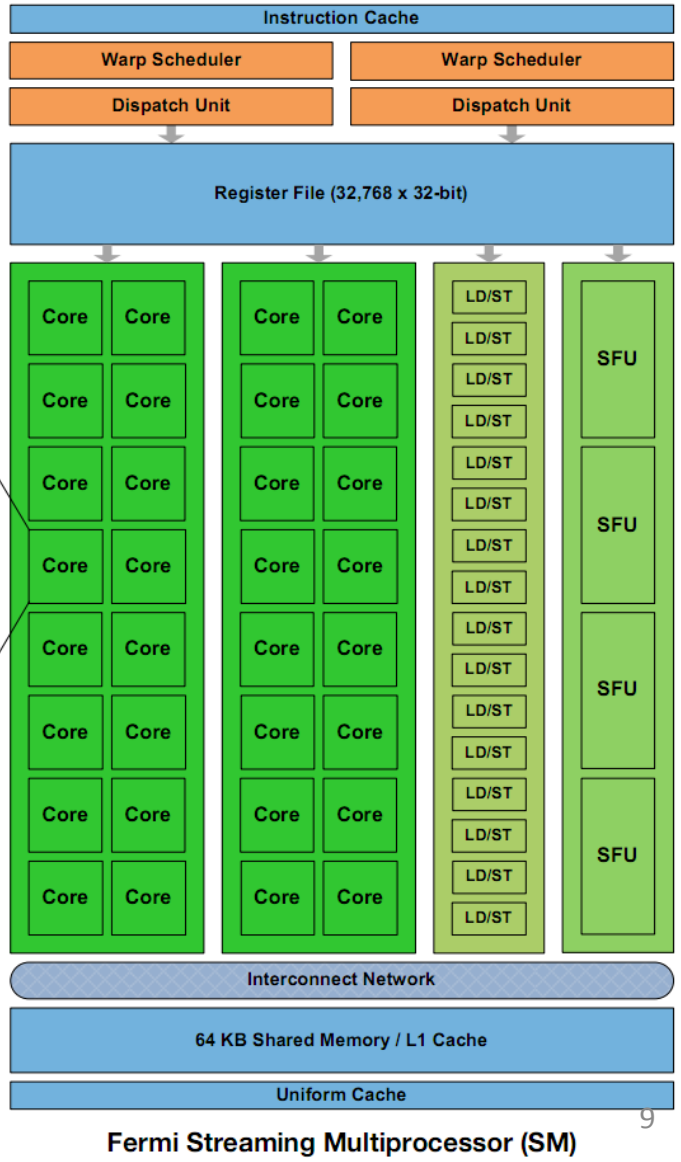
```
for (int k = 0; k < BLOCK_SIZE; ++k)
{
   Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

- Instruction Mix
  - 2 floating-point arithmetic instructions
  - 1 loop branch instruction
  - 2 address arithmetic instructions
  - 1 loop counter increment instruction

Loop Unrolling

- Only 1/3 are floating-point calculations
  - But I want my full theoretical 1 TFLOP (Fermi)
  - Consider *loop unrolling*





Fermi Streaming Multiprocessor (SM)

Image: NVIDIA Fermi Whitepaper

Loop Unrolling

```
Pvalue +=
  Ms[ty][0] * Ns[0][tx] +
  Ms[ty][1] * Ns[1][tx] +
  ...
  Ms[ty][15] * Ns[15][tx]; // BLOCK_SIZE = 16
```

- No more loop
  - No loop count update
  - No branch
  - Constant indices – no address arithmetic instructions

Loop Unrolling

- Automatically:

```
#pragma unroll BLOCK_SIZE
for (int k = 0; k < BLOCK_SIZE; ++k)
{
   Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

- Under the hood: Predication
- Disadvantages to unrolling?

Aside: Loop Counters

```
for (i = 0; i < n; ++i)
{
 out[i] = in[offset + stride*i];
}
```

- Should `i` be signed or unsigned?

Loop Unrolling Performance

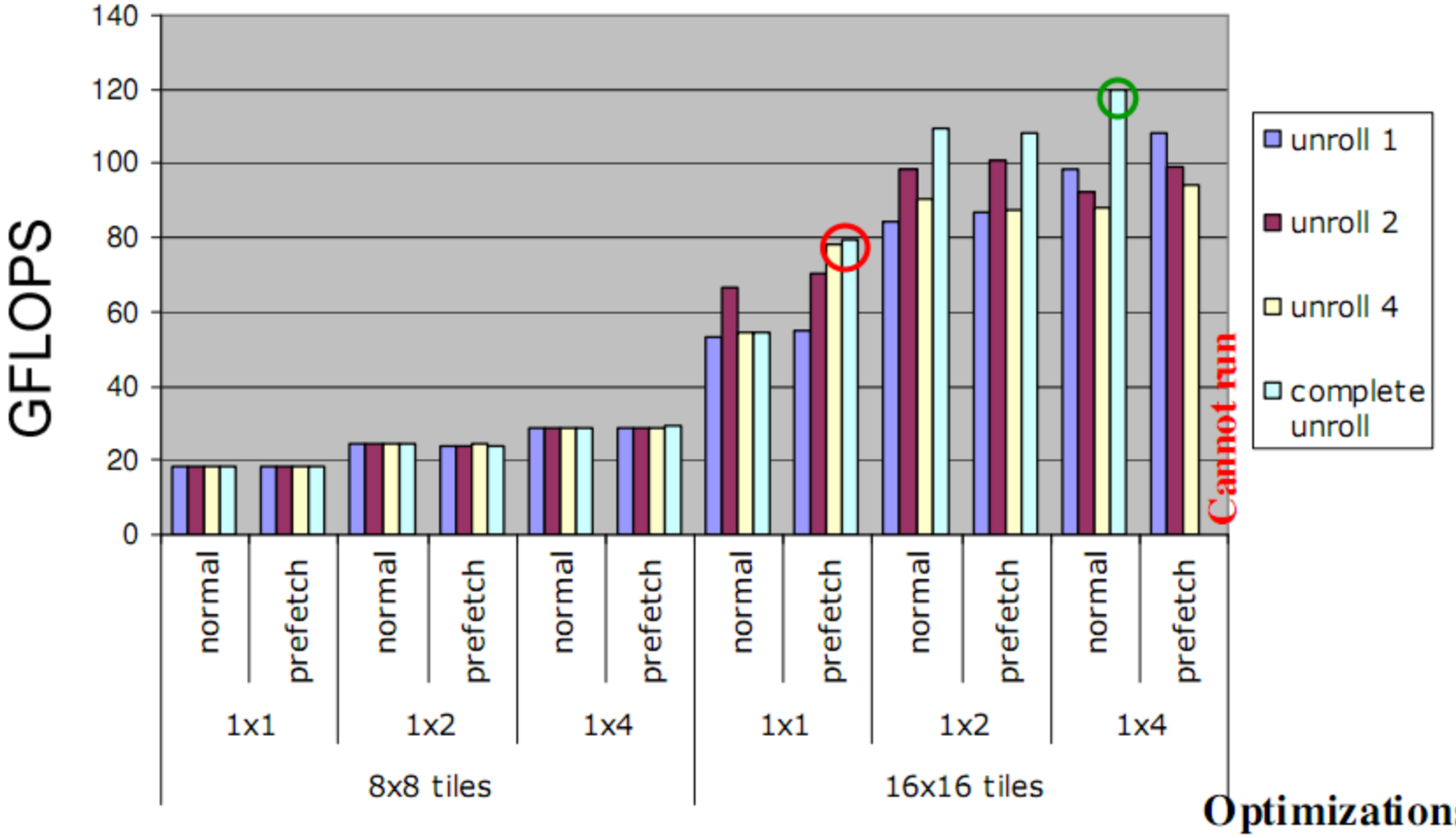


Image from http://courses.engr.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf

Thread Granularity

- How much work should one thread do?
  - Parallel Reduction
    - Reduce two elements?
  - Matrix multiply
    - Compute one element of `Pd`?

Unroll the parallel loop

- Use half as many threads
- But twice as much work per thread
- This amounts to replicating lines of code

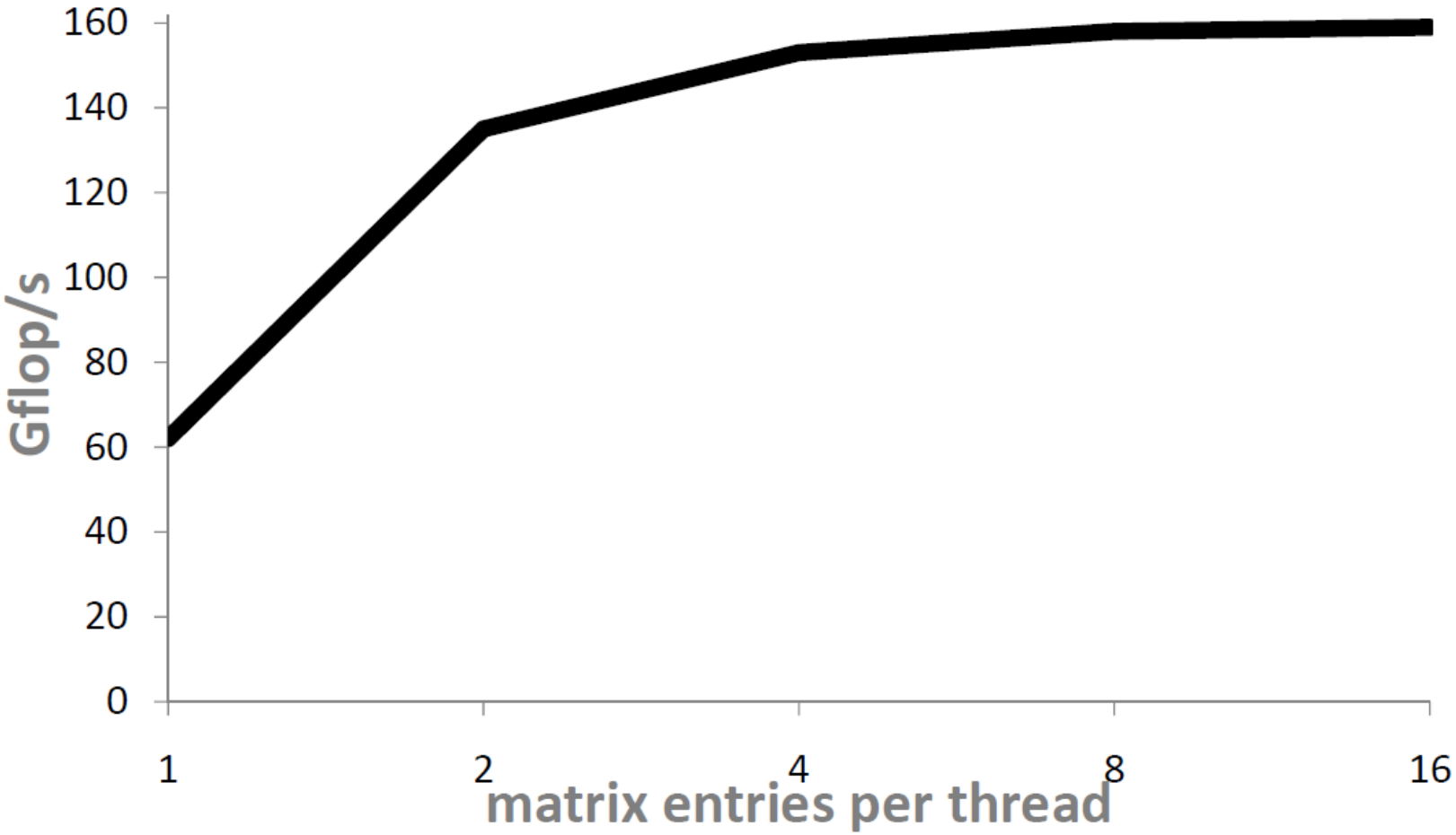
Slide from Vasily Volkov's talk at SC11

Unrolling 2x (red is new)

```
__global__ void eliminate( float *in, float *out ) {
    int x = threadIdx.x, y = threadIdx.y, problem = blockIdx.x;

    //copy matrix to shared memory
    A[2*y+0][x] = in[32*32*problem+32*(2*y+0)+x];
    A[2*y+1][x] = in[32*32*problem+32*(2*y+1)+x];

    //Gauss-Jordan in shared memory
    #pragma unroll
    for( int i = 0; i < 32; i++ )
    {
        if( y == i/2 ) A[i][x] /= A[i][i];
        __syncthreads( );
        if( 2*y+0 != i ) A[2*y+0][x] -= A[i][x]*A[2*y+0][i];
        if( 2*y+1 != i ) A[2*y+1][x] -= A[i][x]*A[2*y+1][i];
    }

    //store the result in global memory
    out[32*32*problem+32*(2*y+0)+x] = A[2*y+0][x];
    out[32*32*problem+32*(2*y+1)+x] = A[2*y+1][x];
}
```


Slide from Vasily Volkov's talk at SC11

Aggregate speedup: 2.6x

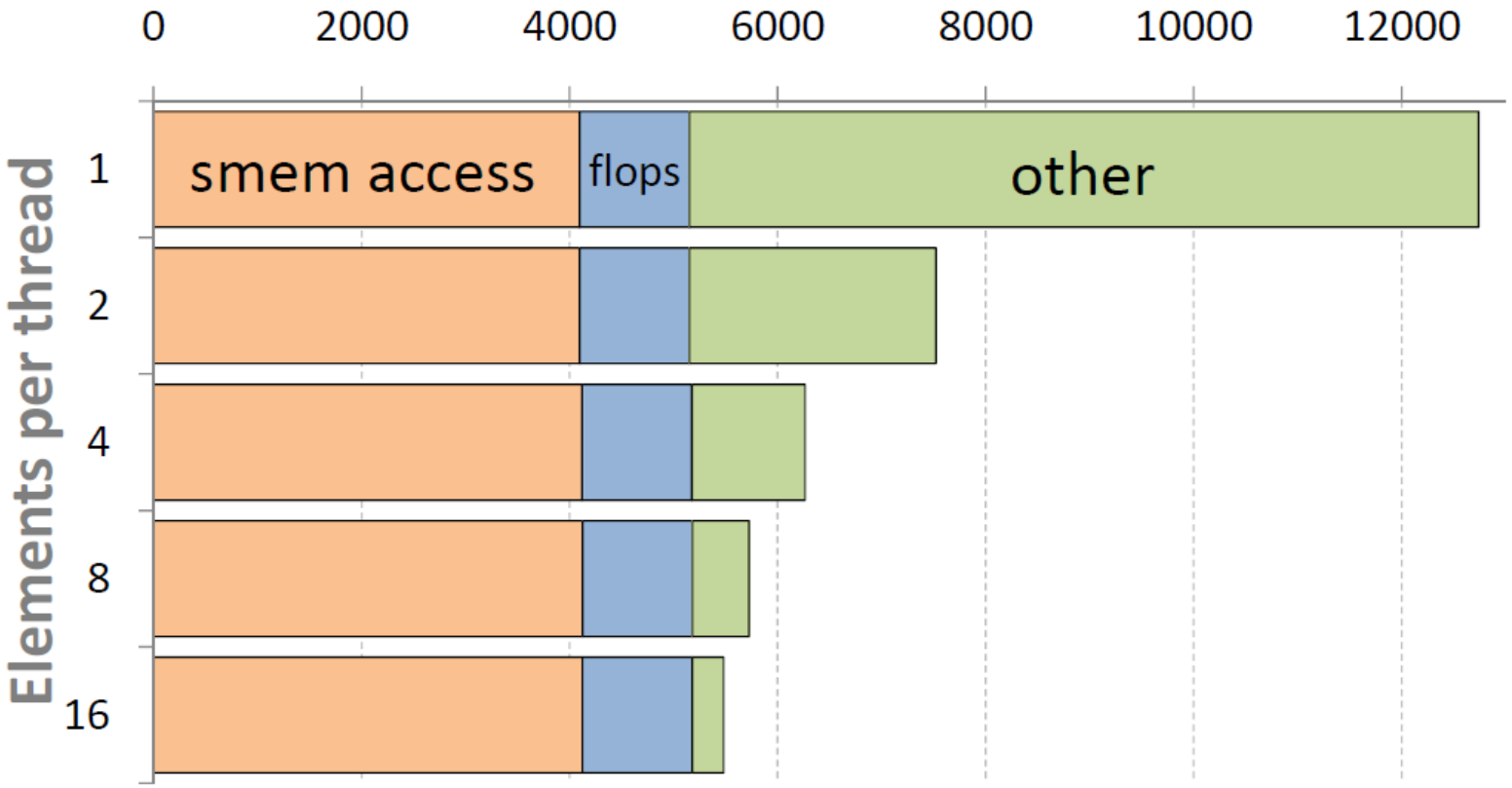


Let's use profiler to figure out what happened

30

Slide from Vasily Volkov's talk at SC11

Instructions executed per thread block



Dramatically fewer auxiliary instructions (control, barriers, etc.)

- Similar effect as with classical loop unrolling

Most instructions are shared memory access?!

Slide from Vasily Volkov's talk at SC11

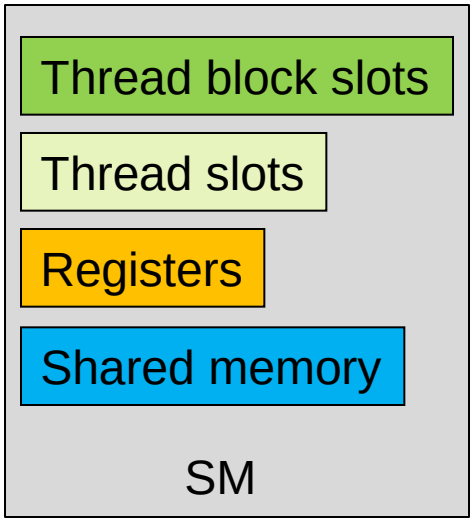
All about Tradeoffs

- Thread count
- Block count
- Register count
- Shared memory size
- Instruction size

Shared Memory

- Fast pool of on-chip memory
- Allocated per block
- Note: runs at base clock instead of shader clock

- Shared memory access patterns can affect performance. Why?



G80 Limits
8
768
8K registers
16K

Bank Conflicts

- Shared Memory
  - Sometimes called a *parallel data cache*
    - Multiple threads can access shared memory at the same time
  - Memory is divided into *banks* (Why?)

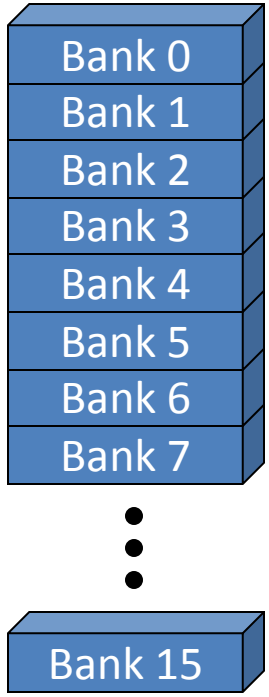


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- Banks
  - Each bank can service one address per two cycles
  - Per-bank bandwidth: 32-bits per two (shader clock) cycles
  - Successive 32-bit words are assigned to successive banks

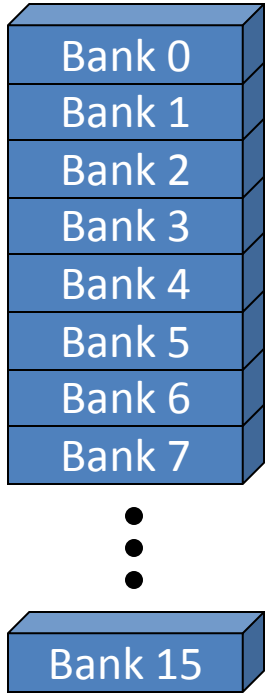


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- *Bank Conflict*: Two simultaneous accesses to the same bank, but not the same address
  - Serialized
- G80-GT200: 16 banks, with 8 SPs concurrently executing
- Fermi: 32 banks, with 16 SPs concurrently executing
  - What does this mean for conflicts?

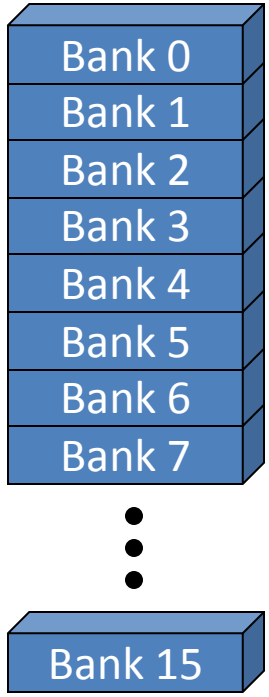
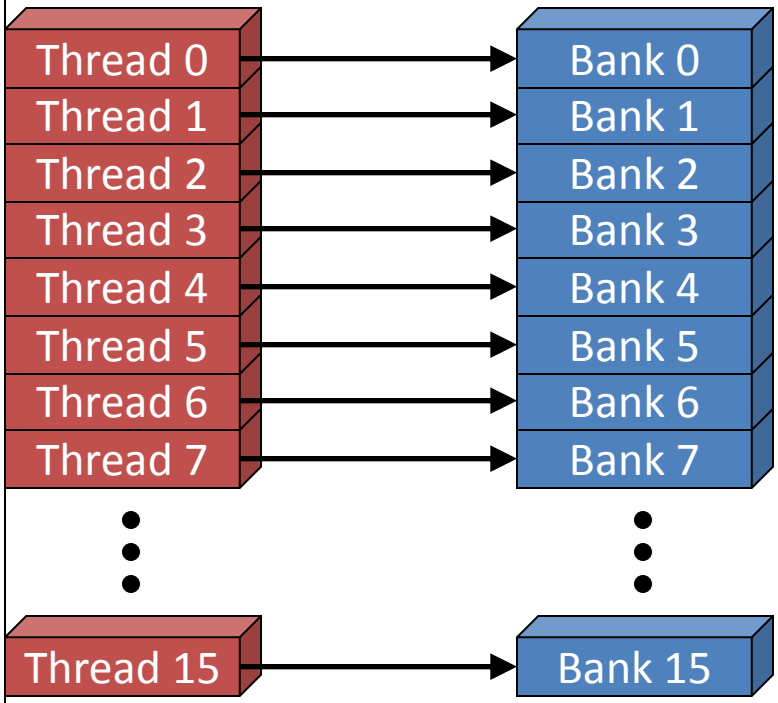


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- Bank Conflicts?
  - Linear addressing stride == 1



- Bank Conflicts?
  - Random 1:1 Permutation

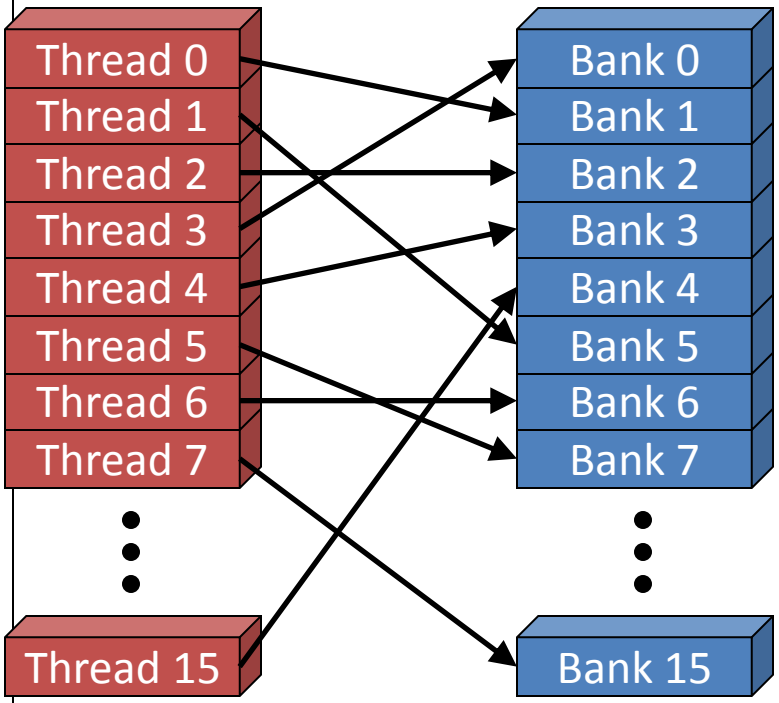


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

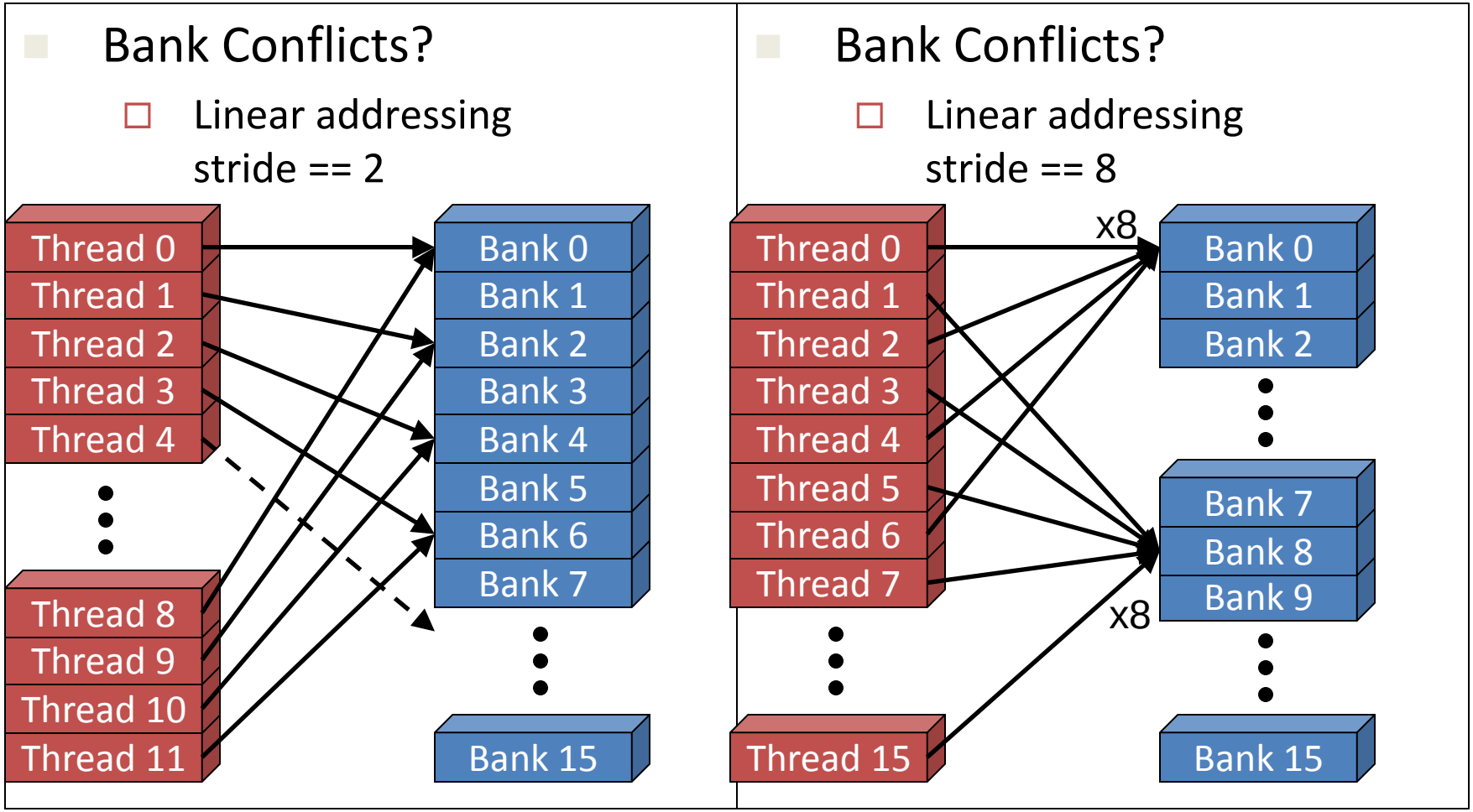


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- Fast Path 1 (G80)
  - All threads in a half-warp access different banks

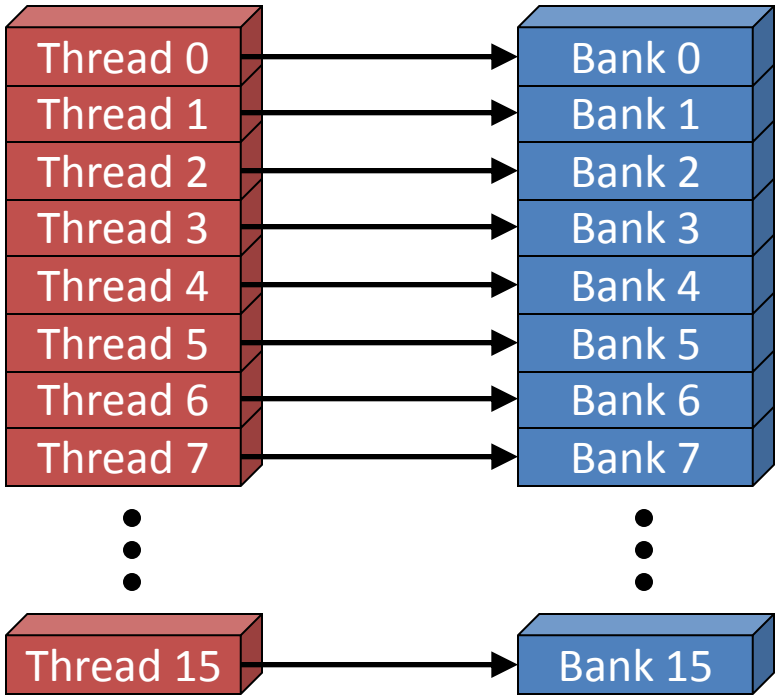


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- Fast Path 2 (G80)
  - All threads in a half-warp access the same address

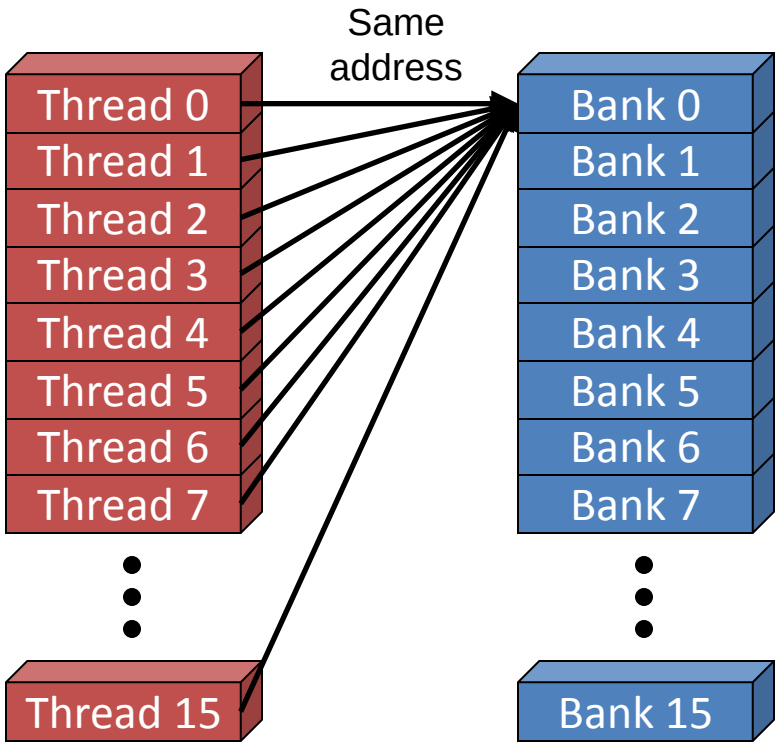


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- Slow Path (G80)
  - Multiple threads in a half-warp access the same bank
  - Access is serialized
  - What is the cost?

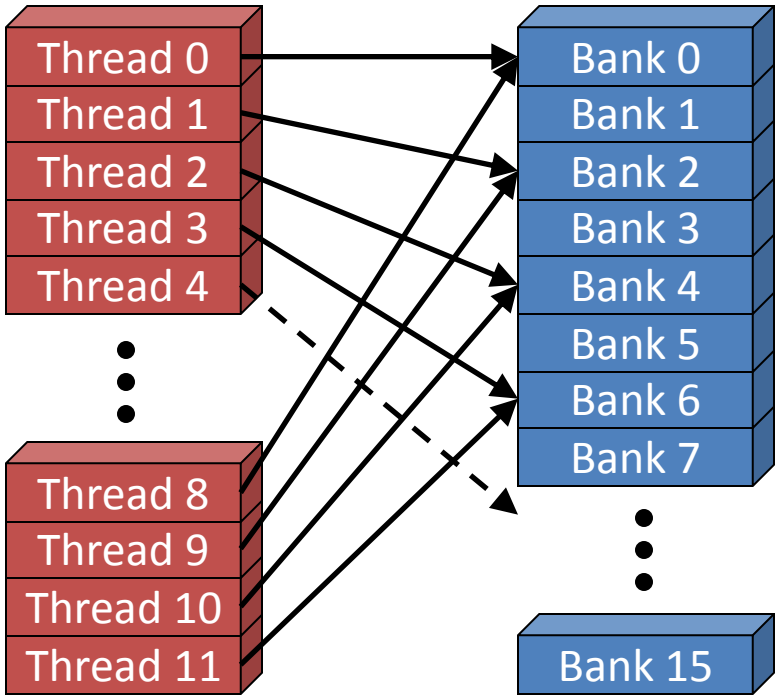


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

```
__shared__ float shared[256];
// ...
float f = shared[index + s * threadIdx.x];
```

- For what values of `s` is this conflict free?
  - Hint: The G80 has 16 banks

Bank Conflicts

```
__shared__ float shared[256];
// ...
float f = shared[index + s * threadIdx.x];
```

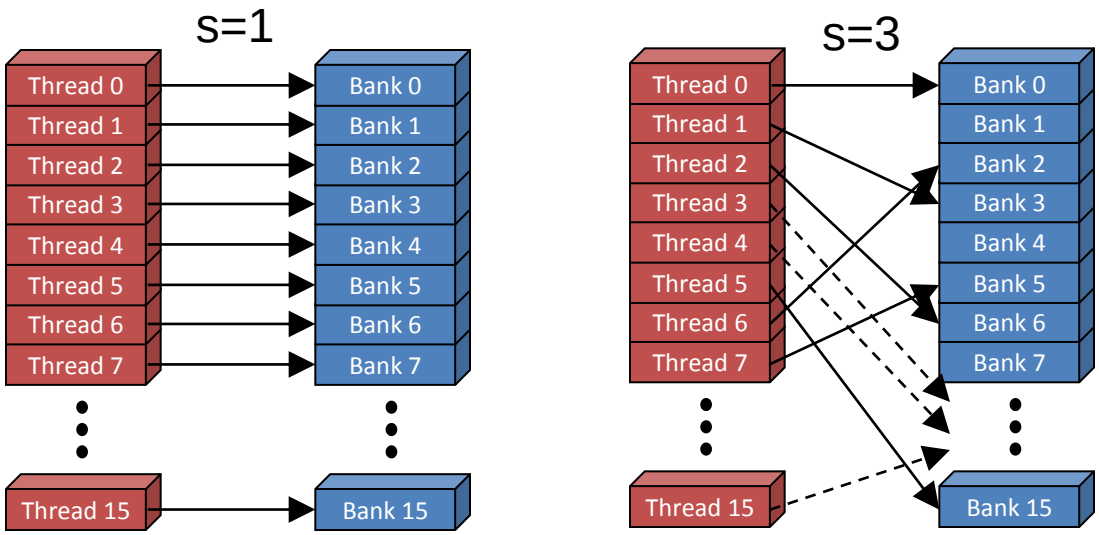


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- Without using a profiler, how can we tell what kind of speedup we can expect by removing bank conflicts?
- What happens if more than one thread in a warp writes to the same shared memory address (non-atomic instruction)?

Partition Camping

- “Bank conflicts” of global memory
- Global memory divided into 6 (G80) or 8 (GT200) 256-byte partitions
- The 1 million KBps question: How do active half-warps in your kernel access memory?

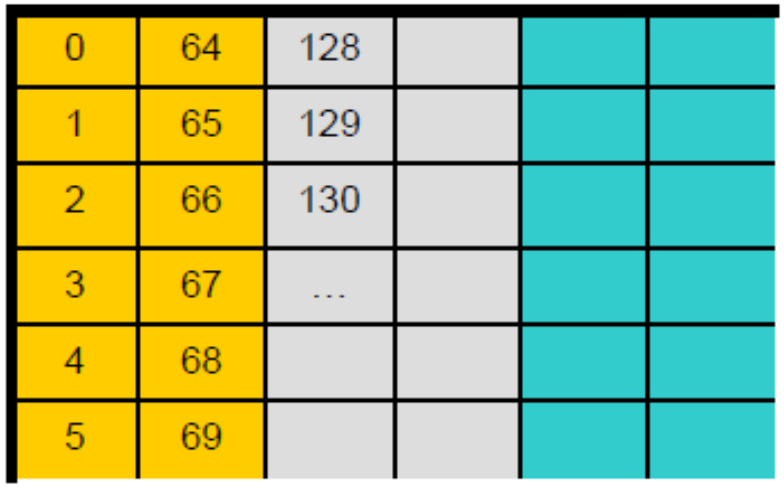


Image: [Reutsch & Micikevicius, 2010]

Fixing Partition Camping

- Diagonalize block indices

```
blockIdx_y=blockIdx.x;
blockIdx_x=
 (blockIdx.x+blockIdx.y)
%gridDim.x;
```

- Output:

0					
64	1				
128	65	2			
	129	66	3		
		130	67	4	
			...	68	5

- Not a problem in Fermi (How?)

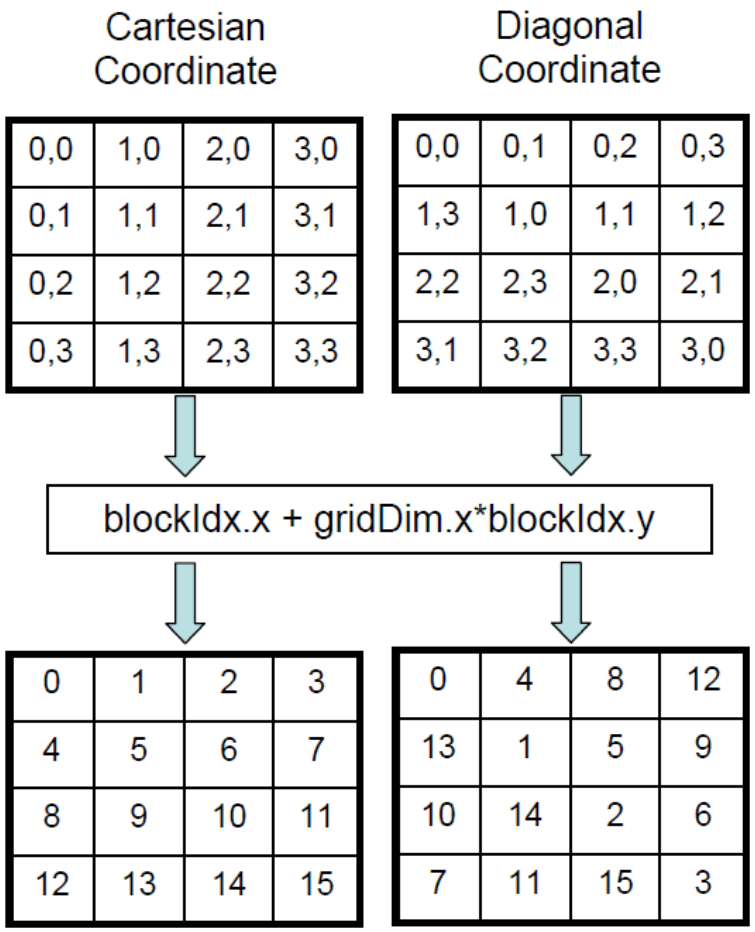


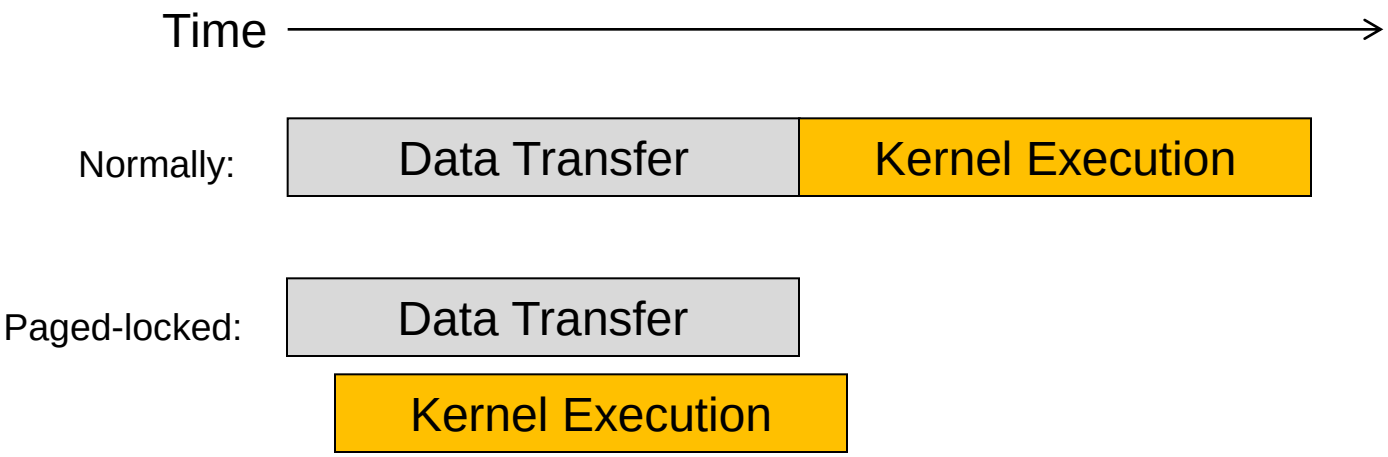
Image: [Reutsch & Micikevicius, 2010]

Page-Locked Host Memory

- *Page-locked Memory*
  - Host memory that is essentially removed from virtual memory
  - Also called *Pinned Memory*

Page-Locked Host Memory

- Benefits
  - Overlap kernel execution and data transfers



See G.1 in the NVIDIA CUDA C Programming Guide for full compute capability requirements

Page-Locked Host Memory

- Benefits
  - Increased memory bandwidth for systems with a front-side bus
    - Up to ~2x throughput

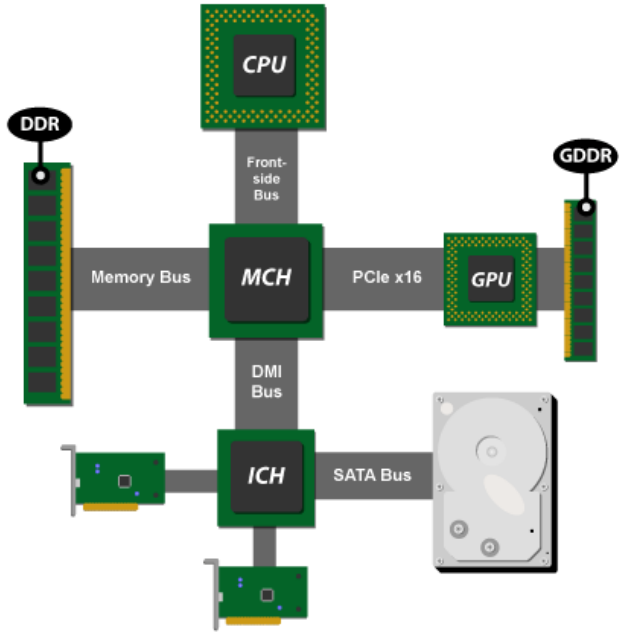


Image from http://arstechnica.com/hardware/news/2009/10/day-of-nvidia-chipset-reckoning-arrives.ars

Page-Locked Host Memory

- Benefits
  - Option: *Write-Combining* Memory
    - Disables page-locked memory's default caching
    - Allocate with `cudaHostAllocWriteCombined` to
      - Avoid polluting L1 and L2 caches
      - Avoid snooping transfers across PCIe
        - » Improve transfer performance up to 40% - in theory
    - Reading from write-combining memory is *slow*!
      - Only write to it from the host

Page-Locked Host Memory

- Benefits
  - Paged-locked *host* memory can be mapped into the address space of the *device* on some systems
    - What systems allow this?
    - What does this eliminate?
    - What applications does this enable?
  - Call `cudaGetDeviceProperties()` and check `canMapHostMemory`

Page-Locked Host Memory

- Usage:

```
cudaHostAlloc() / cudaMallocHost()
cudaHostFree()

cudaMemcpyAsync()
```

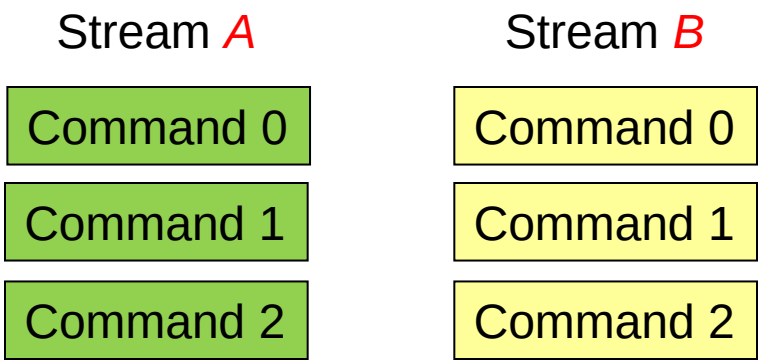

See 3.2.5 in the NVIDIA CUDA C Programming Guide

Page-Locked Host Memory

- What's the catch?
  - Page-locked memory is scarce
    - Allocations will start failing before allocation of in pageable memory
  - Reduces amount of physical memory available to the OS for paging
    - Allocating too much will hurt overall system performance

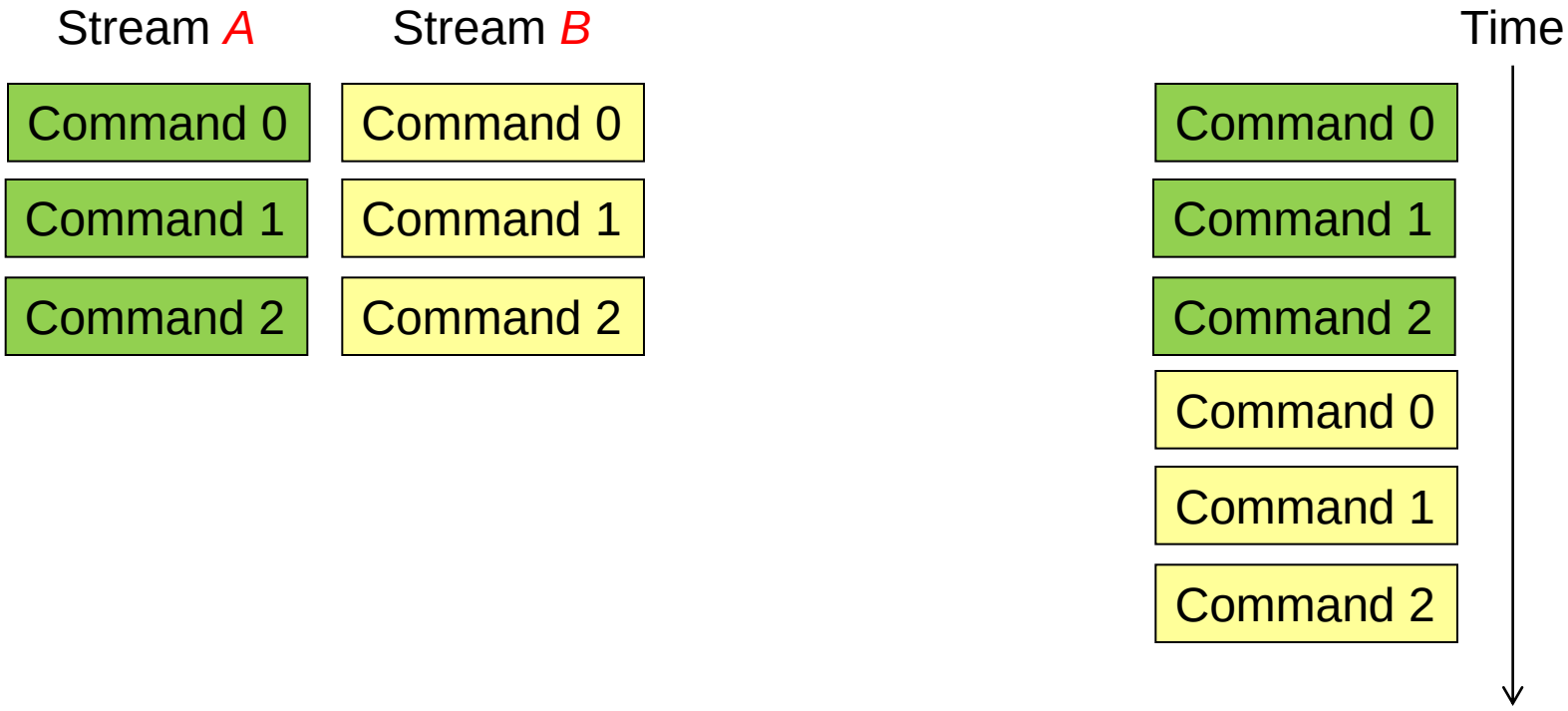
Streams

- *Stream*: Sequence of commands that execute in order
- Streams may execute their commands out-of-order or concurrently with respect to other streams



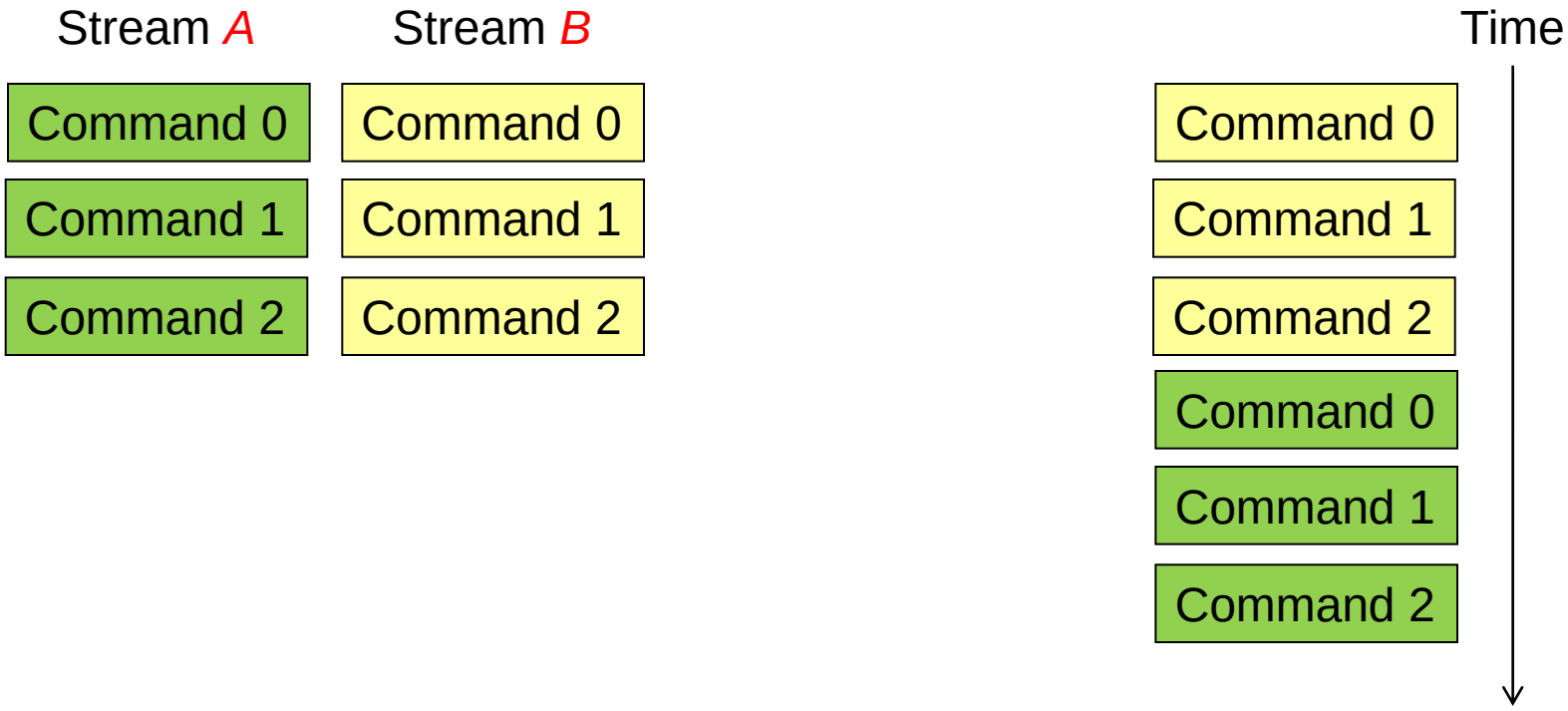
Streams

- Is this a possible order?

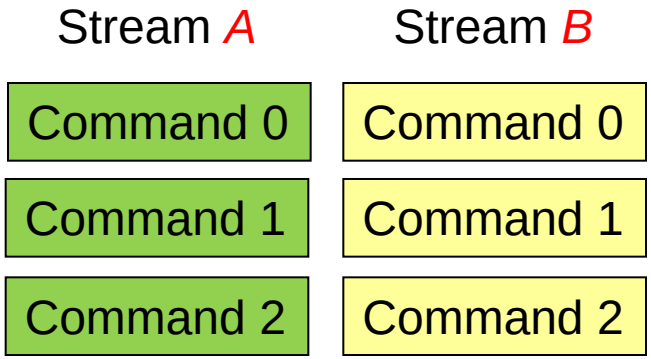


Streams

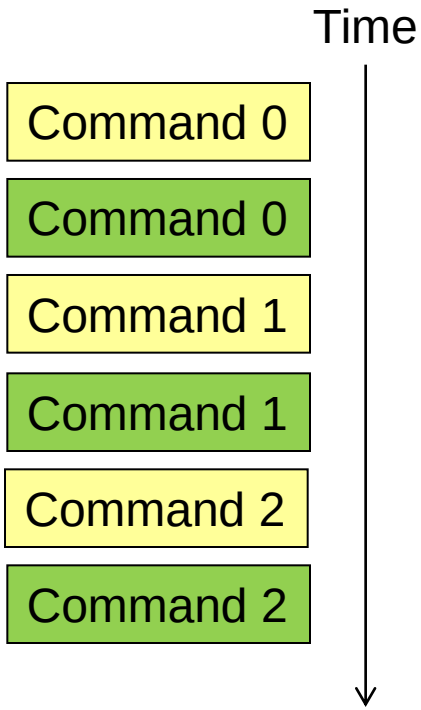
- Is this a possible order?



Streams

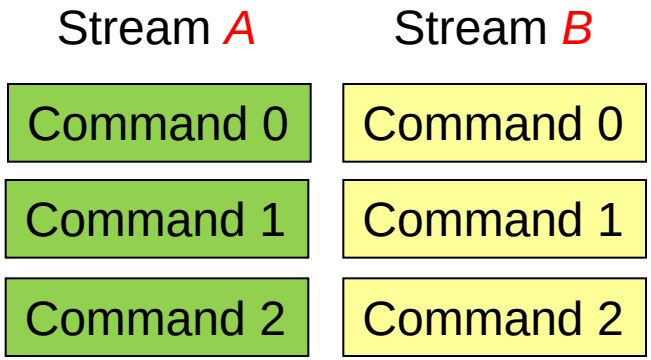


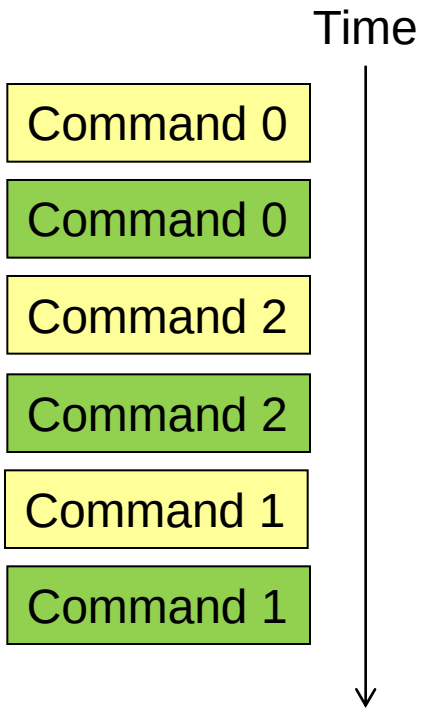
- Is this a possible order?



Streams

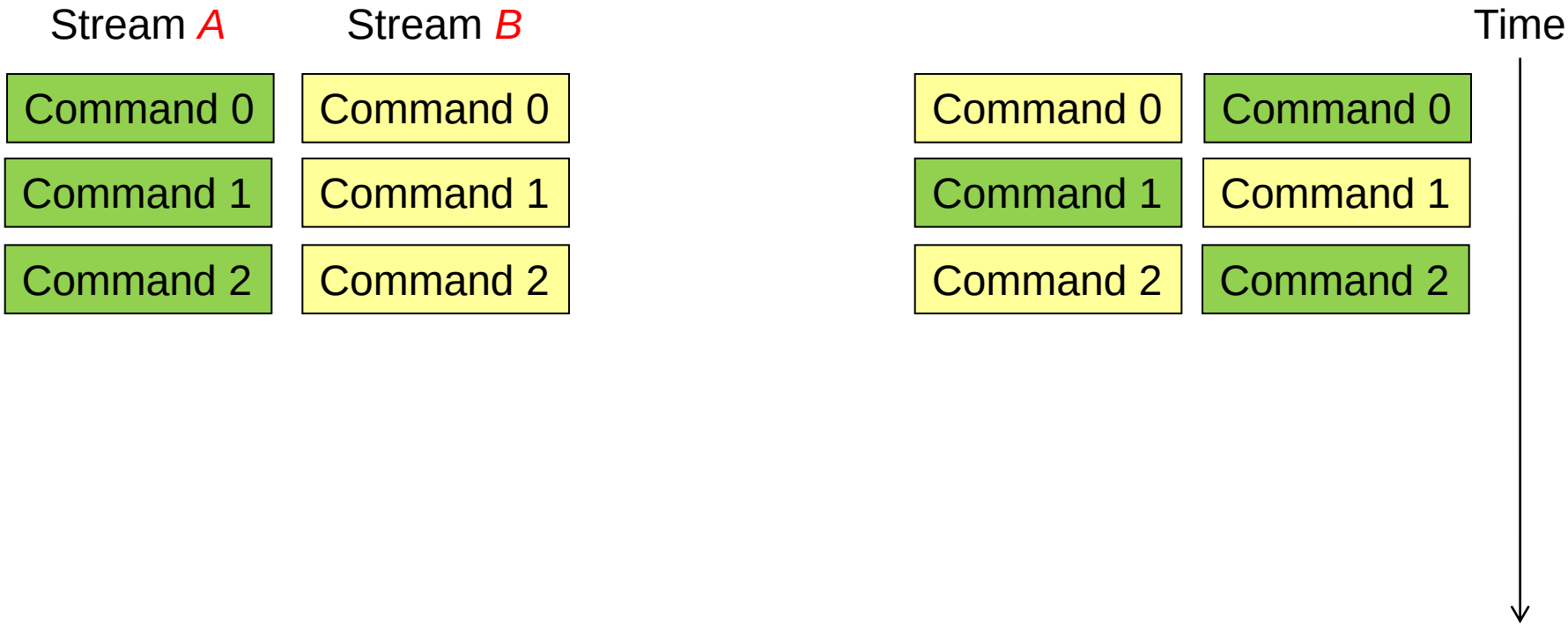
- Is this a possible order?



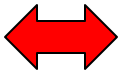


Streams

- Is this a possible order?



Streams

- In CUDA, what commands go in a stream?
  - Kernel launches
  - Host  device memory transfers

Streams

- Code Example
  1. Create two streams
  2. Each stream:
     1. Copy page-locked memory to device
     2. Launch kernel
     3. Copy memory back to host
  3. Destroy streams

Stream Example (Step 1 of 3)

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
{
    cudaStreamCreate(&stream[i]);
}

float *hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

Stream Example (Step 1 of 3)

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
{
    cudaStreamCreate(&stream[i]);
}

float *hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

Create two streams

Stream Example (Step 1 of 3)

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
{
    cudaStreamCreate(&stream[i]);
}

float *hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

Allocate two buffers in page-locked memory

Stream Example (Step 2 of 3)

```
for (int i = 0; i < 2; ++i)
{
  cudaMemcpyAsync(/* ... */,
    cudaMemcpyHostToDevice, stream[i]);
  kernel<<<100, 512, 0, stream[i]>>>
    (/* ... */);
  cudaMemcpyAsync(/* ... */,
    cudaMemcpyDeviceToHost, stream[i]);
}
```

Stream Example (Step 2 of 3)

```
for (int i = 0; i < 2; ++i)
{
  cudaMemcpyAsync(/* ... */,
    cudaMemcpyHostToDevice, stream[i]);
  kernel<<<100, 512, 0, stream[i]>>>
    (/* ... */);
  cudaMemcpyAsync(/* ... */,
    cudaMemcpyDeviceToHost, stream[i]);
}
```

Commands are assigned to, and executed by streams

Stream Example (Step 3 of 3)

```
for (int i = 0; i < 2; ++i)
{
  // Blocks until commands complete
  cudaStreamDestroy(stream[i]);
}
```

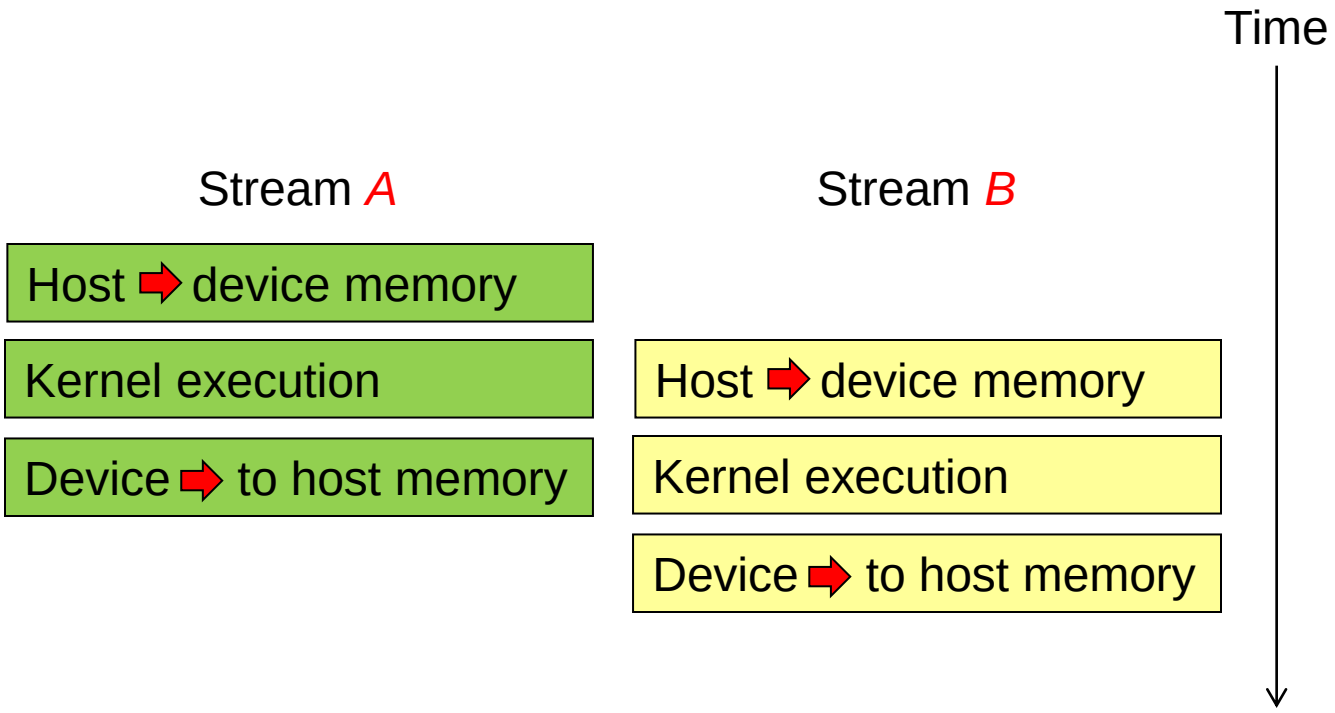
Streams

- Assume compute capabilities:
  - Overlap of data transfer and kernel execution
  - Concurrent kernel execution
  - Concurrent data transfer
- How can the streams overlap?

See F.1 in the NVIDIA CUDA C Programming Guide for more on compute capabilities

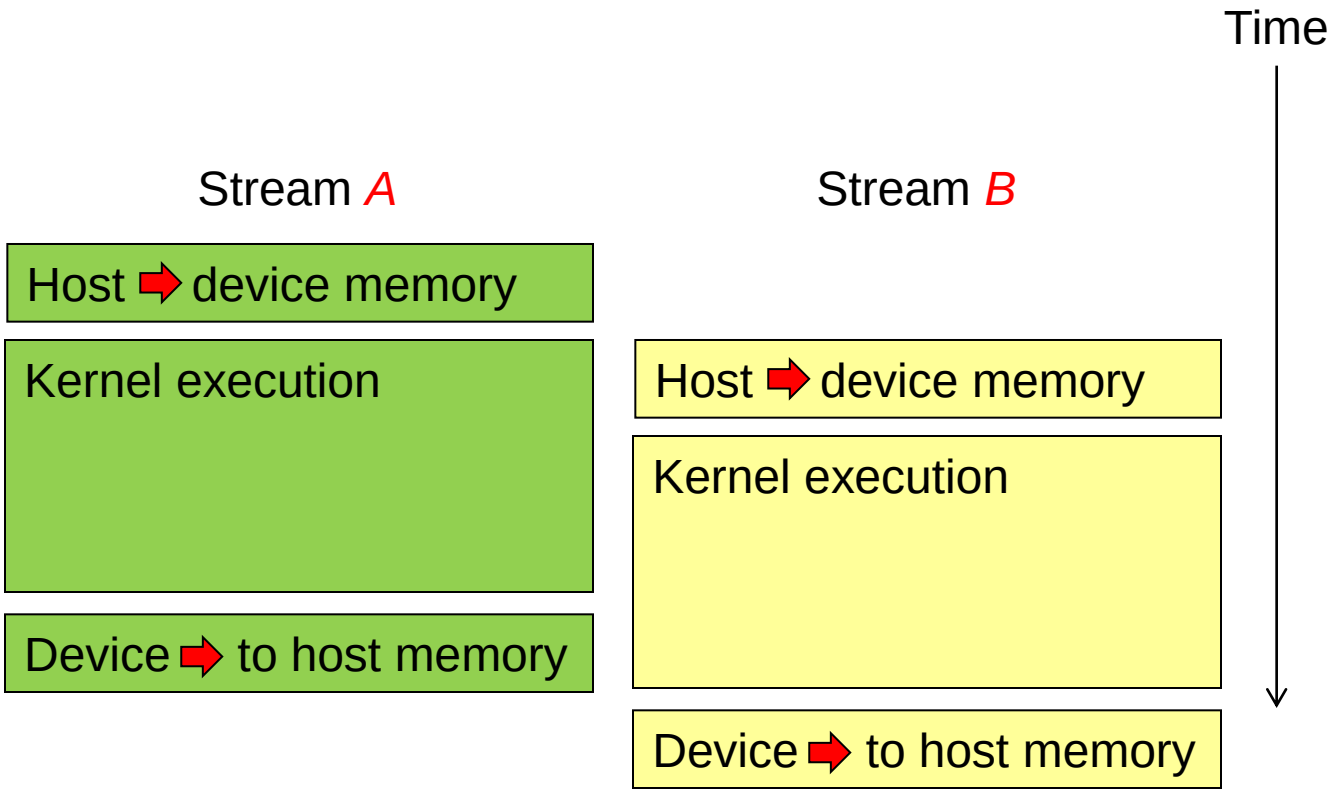
Streams

- Can we have more overlap than this?



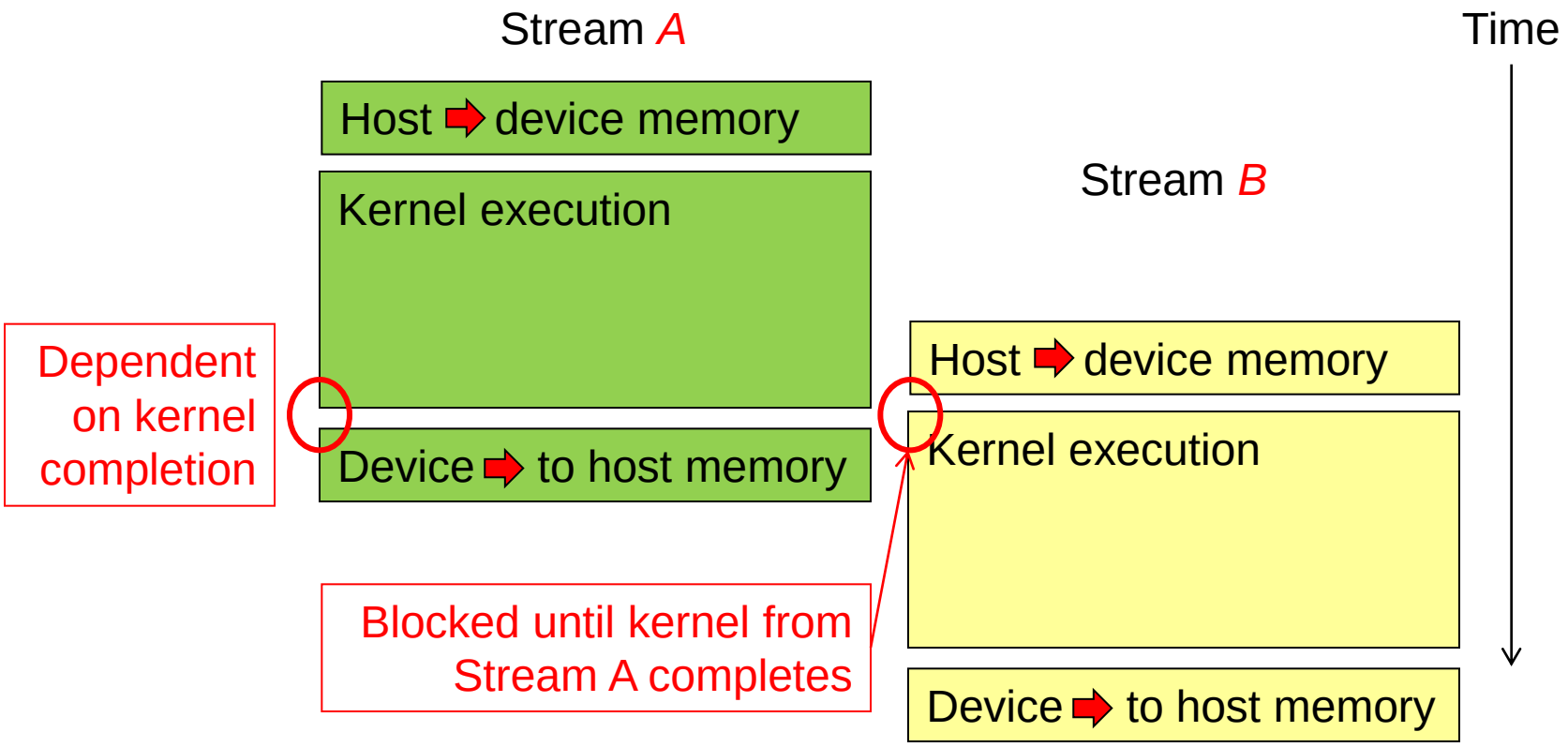
Streams

- Can we have this?



Streams

- Can we have this?



Streams

- Performance Advice
  - Issue all independent commands before dependent ones
  - Delay synchronization (implicit or explicit) as long as possible

Streams

- Rewrite this to allow concurrent kernel execution

```
for (int i = 0; i < 2; ++i)
{
  cudaMemcpyAsync(/* ... */, stream[i]);
  kernel<<< /*... */ stream[i]>>>();
  cudaMemcpyAsync(/* ... */, stream[i]);
}
```

Streams

```
for (int i = 0; i < 2; ++i) // to device
 cudaMemcpyAsync(/* ... */, stream[i]);

for (int i = 0; i < 2; ++i)
  kernel<<< /*... */ stream[i]>>>();

for (int i = 0; i < 2; ++i)  // to host
  cudaMemcpyAsync(/* ... */, stream[i]);
```

Streams

- *Explicit Synchronization*
  - `cudaThreadSynchronize()`
    - Blocks until commands in all streams finish
  - `cudaStreamSynchronize()`
    - Blocks until commands in a stream finish

See 3.2.5.5.3 in the NVIDIA CUDA C Programming Guide for more synchronization functions

References

- CUDA C Best Practices Guide, version 4.1
- CUDA C Programming Guide, version 4.1
- Reutsch, Greg and Micikevicius, Paulius. "Optimizing Matrix Transpose in CUDA." June 2010.
- Volkov, Vasily. "Unrolling parallel loops." November 14, 2011. Slides

Bibliography

- Optimal Parallel Reduction Proof with Brent's Theorem
- Vasily Volkov. "Better Performance at Lower Occupancy." Slides
- Mark Harris. "Optimizing Parallel Reduction in CUDA." Slides