NVIDIA®

Optimizing CUDA

Execution Model

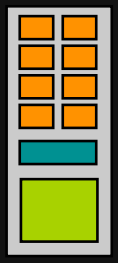
Software **Hardware**

Thread

Thread Processor

Threads are executed by thread processors

Thread Block

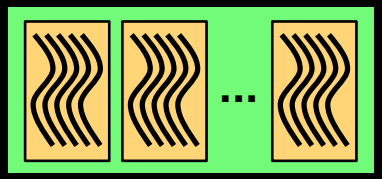


Multiprocessor

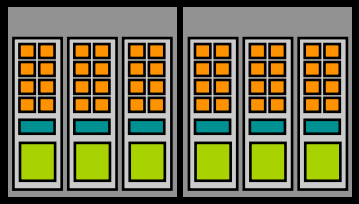
Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)



Grid

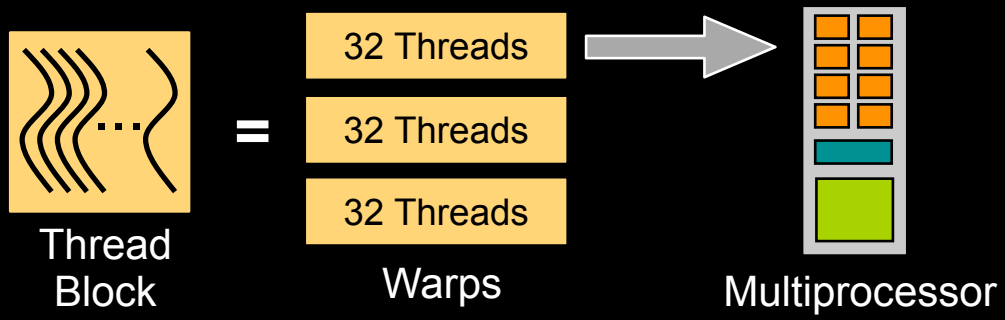


Device

A kernel is launched as a grid of thread blocks

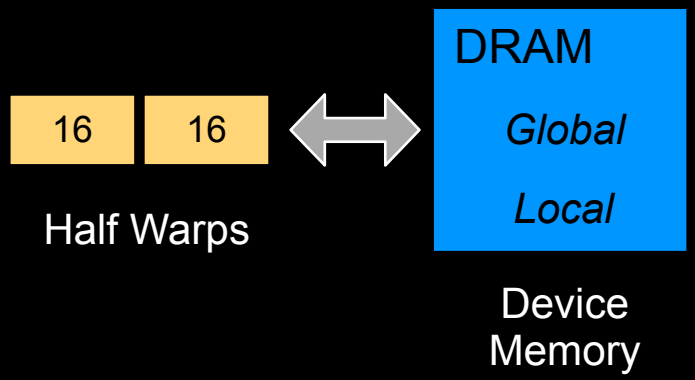
Only one kernel can execute on a device at one time

Warps and Half Warps



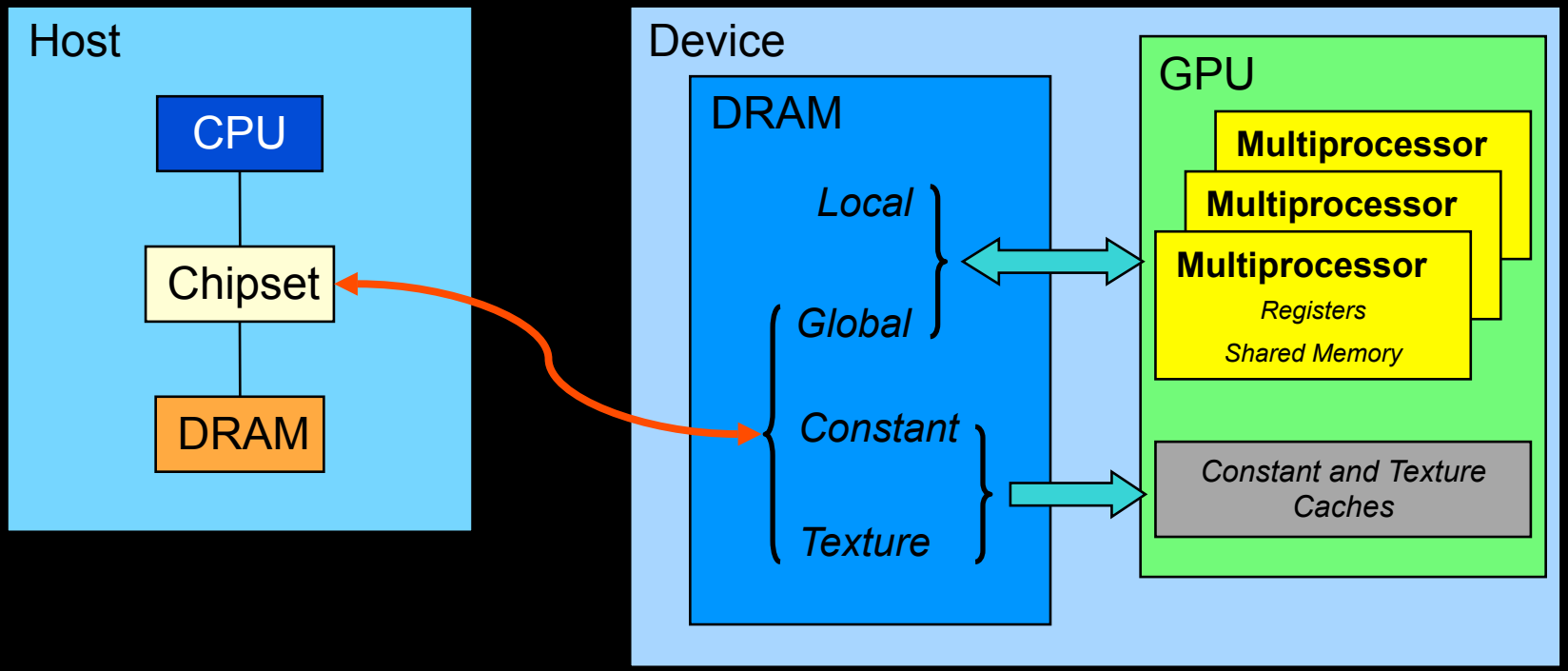
A thread block consists of 32-thread warps

A warp is executed physically in parallel (SIMD) on a multiprocessor



A half-warp of 16 threads can coordinate global memory accesses into a single transaction

Memory Architecture



Memory Architecture

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

Outline

- Overview
- Hardware
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

Host-Device Data Transfers

- Device to host memory bandwidth much lower than device to device bandwidth
  - 8 GB/s peak (PCI-e x16 Gen 2) vs. 141 GB/s peak (GTX 280)
- Minimize transfers
  - Intermediate data can be allocated, operated on, and deallocated without ever copying them to host memory
- Group transfers
  - One large transfer much better than many small ones

Page-Locked Data Transfers

- `cudaMallocHost()` allows allocation of page-locked ("pinned") host memory
- Enables highest cudaMemcpy performance
  - 3.2 GB/s on PCI-e x16 Gen1
  - 5.2 GB/s on PCI-e x16 Gen2
- See the "bandwidthTest" CUDA SDK sample
- Use with caution!!
  - Allocating too much page-locked memory can reduce overall system performance
  - Test your systems and apps to learn their limits

Overlapping Data Transfers and Computation

- Async and Stream APIs allow overlap of H2D or D2H data transfers with computation
  - CPU computation can overlap data transfers on all CUDA capable devices
  - Kernel computation can overlap data transfers on devices with “Concurrent copy and execution” (roughly compute capability >= 1.1)
- Stream = sequence of operations that execute in order on GPU
  - Operations from different streams can be interleaved
  - Stream ID used as argument to async calls and kernel launches

Asynchronous Data Transfers

- Asynchronous host-device memory copy returns control immediately to CPU
  - `cudaMemcpyAsync(dst, src, size, dir, stream);`
  - requires *pinned* host memory (allocated with "`cudaMallocHost`")

- Overlap CPU computation with data transfer
  - `0` = default stream

```
cudaMemcpyAsync(a_d, a_h, size,
                cudaMemcpyHostToDevice, 0);
kernel<<<grid, block>>>(a_d);
cpuFunction();
```

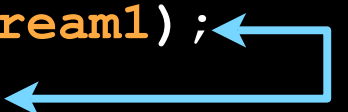
overlapped

Overlapping kernel and data transfer

- **Requires:**
  - "Concurrent copy and execute"
    - `deviceOverlap` field of a `cudaDeviceProp` variable
  - Kernel and transfer use different, *non-zero* streams
    - A CUDA call to stream-0 blocks until all previous calls complete and cannot be overlapped

- **Example:**

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst, src, size, dir, stream1);
kernel<<<grid, block, 0, stream2>>>(…);
```



overlapped

GPU/CPU Synchronization

- Context based
  - `cudaThreadSynchronize()`
    - Blocks until all previously issued CUDA calls from a CPU thread complete
- Stream based
  - `cudaStreamSynchronize(stream)`
    - Blocks until all CUDA calls issued to given stream complete
  - `cudaStreamQuery(stream)`
    - Indicates whether stream is idle
    - Returns `cudaSuccess`, `cudaErrorNotReady, ...`
    - Does not block CPU thread

GPU/CPU Synchronization

- Stream based using events
  - Events can be inserted into streams:

    ```
    cudaEventRecord(event, stream)
    ```

  - Event is recorded then GPU reaches it in a stream
    - Recorded = assigned a timestamp (GPU clocktick)
    - Useful for timing

  - `cudaEventSynchronize(event)`
    - Blocks until given event is recorded

  - `cudaEventQuery(event)`
    - Indicates whether event has recorded
    - Returns `cudaSuccess`, `cudaErrorNotReady, ...`
    - Does not block CPU thread

Outline

- Overview
- Hardware
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
    - Measuring performance - effective bandwidth
    - Coalescing
    - Shared memory
    - Textures
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

Theoretical Bandwidth

- Device Bandwidth of GTX 280
  - 1107 * 10^6 * (512 / 8) * 2 / 1024^3 = 131.9 GB/s

DDR ↓

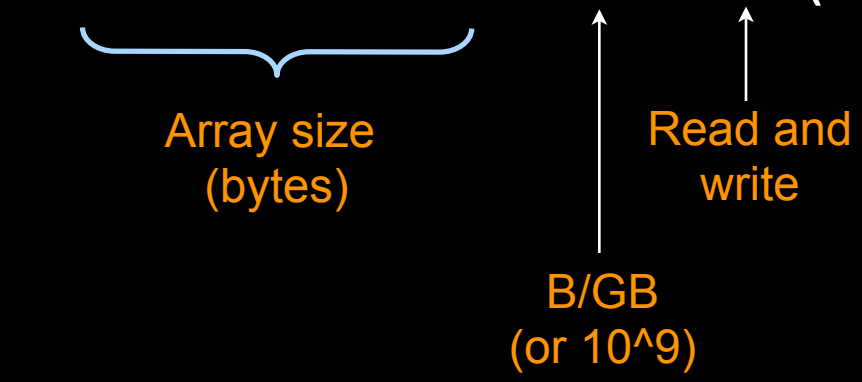
Memory clock (Hz)

Memory interface (bytes)

  - Specs report 141 GB/s
    - Use 10^9 B/GB conversion rather than 1024^3
    - Whichever you use, be consistent

Effective Bandwidth

- Effective Bandwidth (for copying array of N floats)
  - N * 4 B/element / 1024^3 * 2 / (time in secs) = GB/s



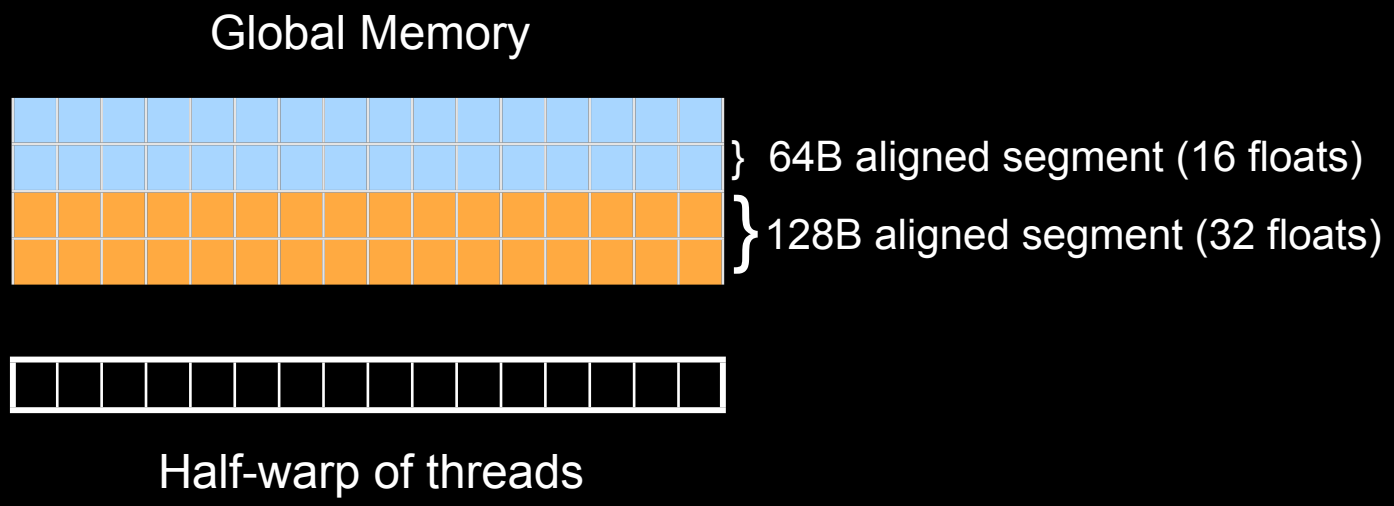
Outline

- Overview
- Hardware
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
    - Measuring performance - effective bandwidth
    - Coalescing
    - Shared memory
    - Textures
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

Coalescing

- Global memory access of 32, 64, or 128-bit words by a half-warp of threads can result in as few as one (or two) transaction(s) if certain access requirements are met
- Depends on compute capability
  - 1.0 and 1.1 have stricter access requirements

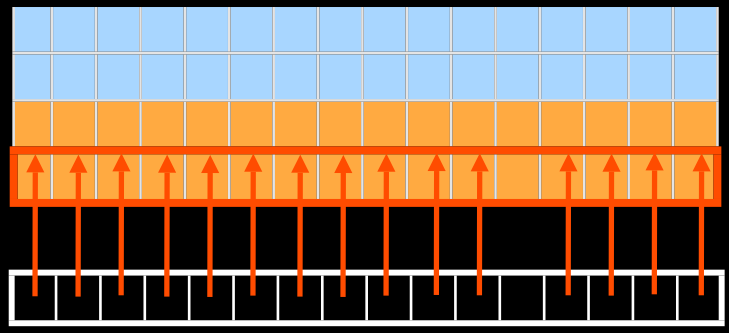
Examples – float (32-bit) data



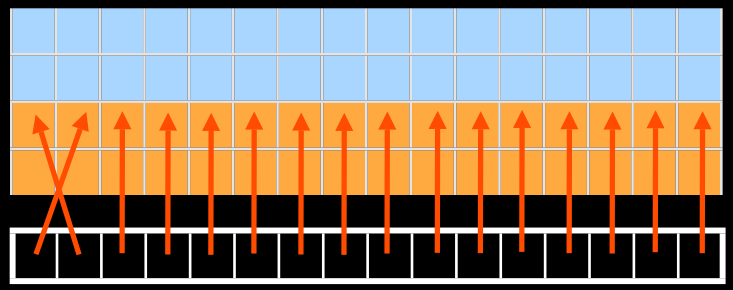
Coalescing
Compute capability 1.0 and 1.1

- **K-th thread must access k-th word in the segment (or k-th word in 2 contiguous 128B segments for 128-bit words), not all threads need to participate**

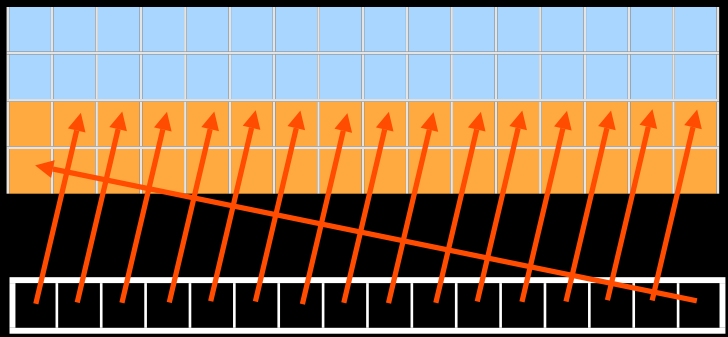
Coalesces – 1 transaction



Out of sequence – 16 transactions

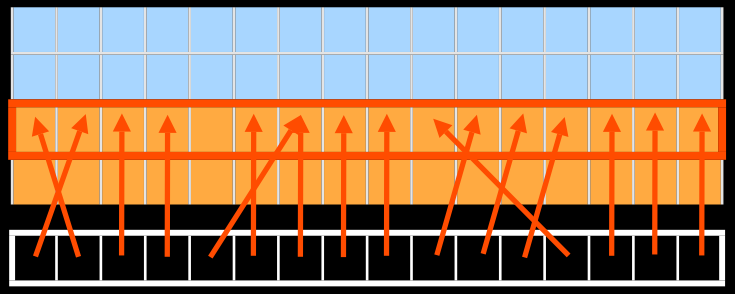


Misaligned – 16 transactions



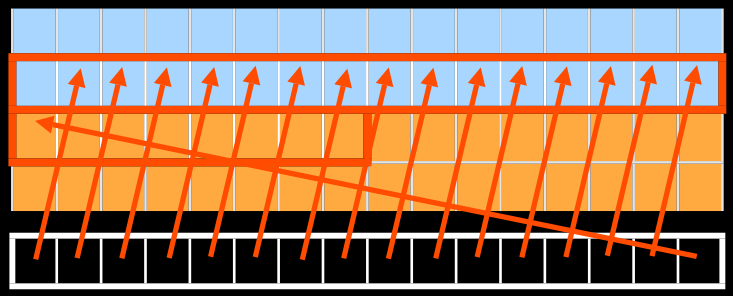
Coalescing
Compute capability 1.2 and higher

- 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
- Smaller transactions may be issued to avoid wasted bandwidth due to unused words

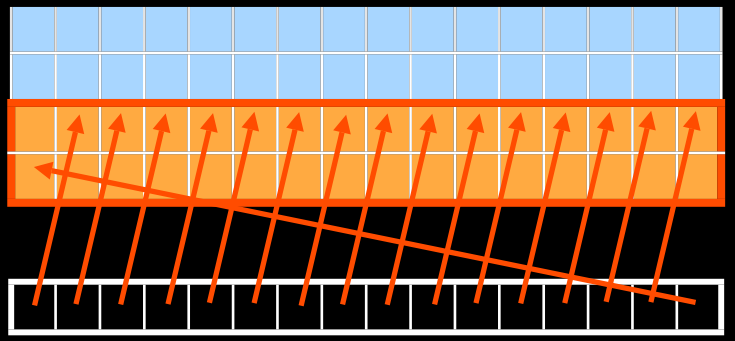


1 transaction - 64B segment

2 transactions - 64B and 32B segments



1 transaction - 128B segment



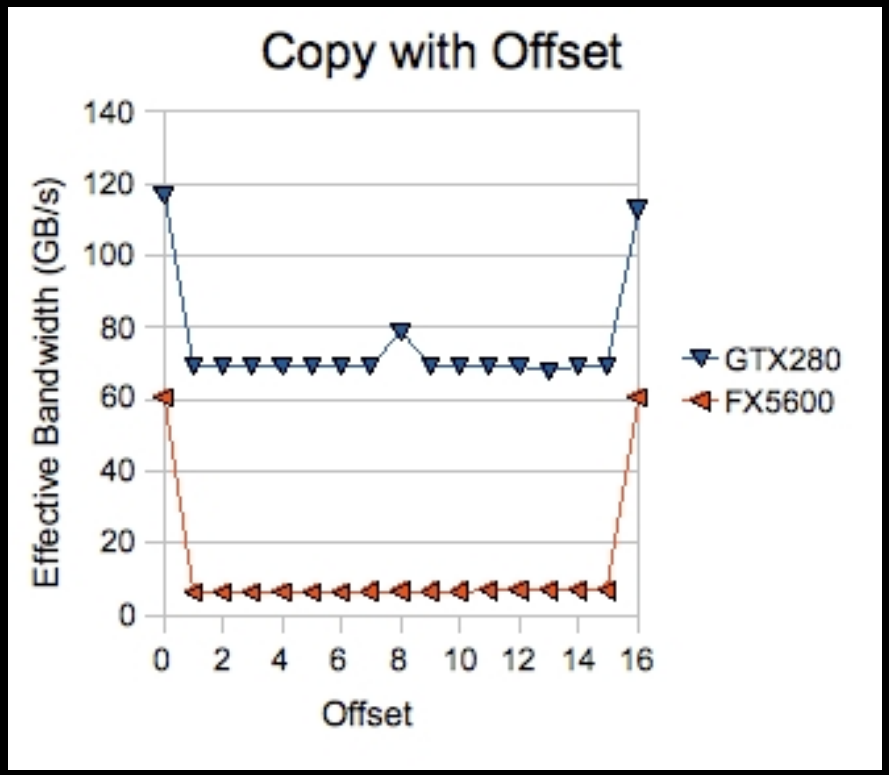
Coalescing Examples

- Effective bandwidth of small kernels that copy data
  - Effects of offset and stride on performance

- Two GPUs
  - GTX 280
    - Compute capability 1.3
    - Peak bandwidth of 141 GB/s
  - FX 5600
    - Compute capability 1.0
    - Peak bandwidth of 77 GB/s

Coalescing Examples


```
__global__ void offsetCopy(float *odata, float* idata,
                           int offset)
{
  int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
  odata[xid] = idata[xid];
}
```



Outline

- Overview
- Hardware
- Memory Optimizations
  - Data Transfers between host and device
  - Device memory optimizations
    - Measuring performance - effective bandwidth
    - Coalescing
    - Shared memory
    - Textures
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

Shared Memory

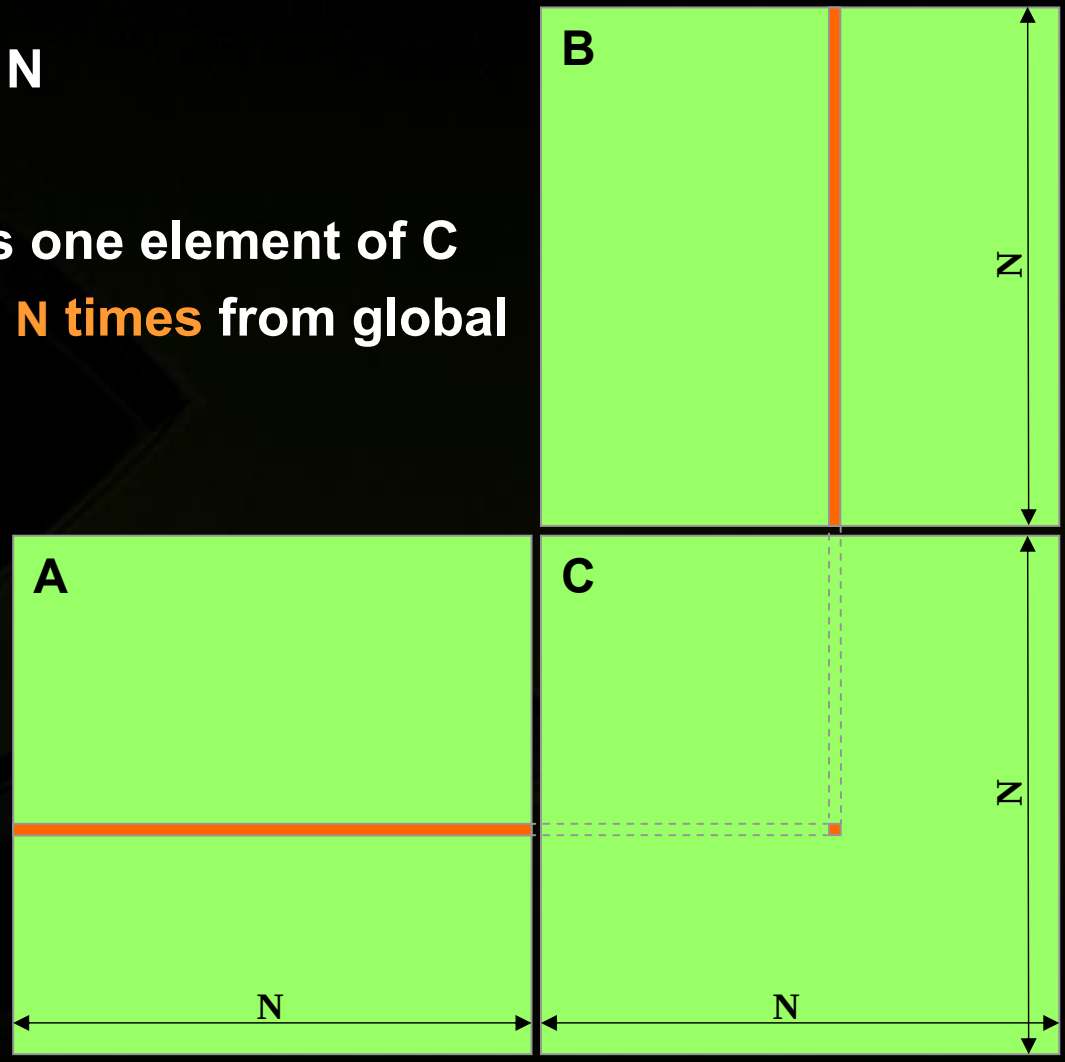
- ~Hundred times faster than global memory
- Cache data to reduce global memory accesses
- Threads can cooperate via shared memory
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing

Maximize Use of Shared Memory

- Shared memory is hundreds of times faster than global memory
- Threads can cooperate via shared memory
  - Not so via global memory
- A common way of scheduling some computation on the device is to block it up to take advantage of shared memory:
  - Partition the data set into data subsets that fit into shared memory
  - Handle each data subset with one thread block:
    - Load the subset from global memory to shared memory
    - __syncthreads()
    - Perform the computation on the subset from shared memory
      - each thread can efficiently multi-pass over any data element
    - __syncthreads() (if needed)
    - Copy results from shared memory to global memory

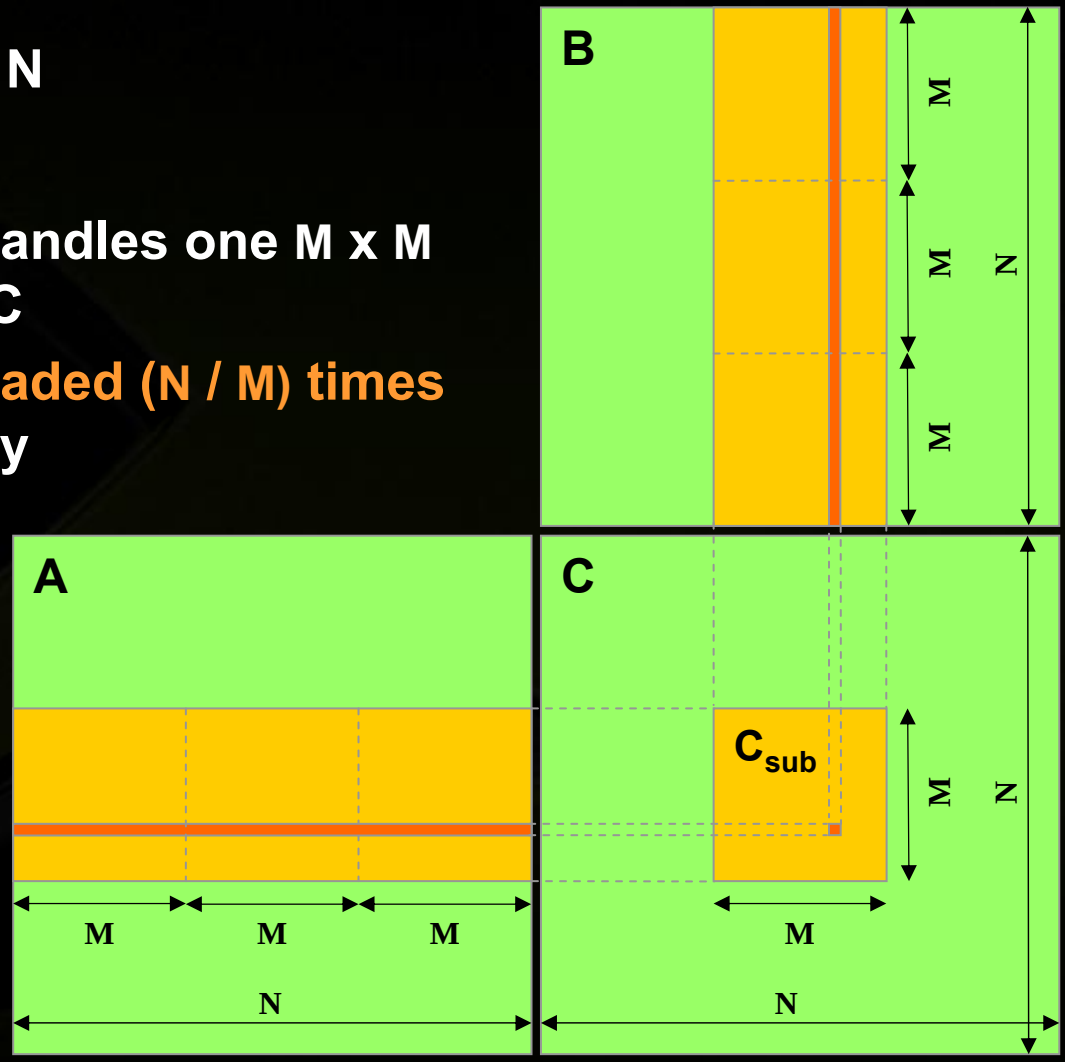
Example: Square Matrix Multiplication

- C = A · B of size N x N
- Without blocking:
  - One thread handles one element of C
  - A and B are loaded N times from global memory
- Wastes bandwidth
- Poor balance of work to bandwidth



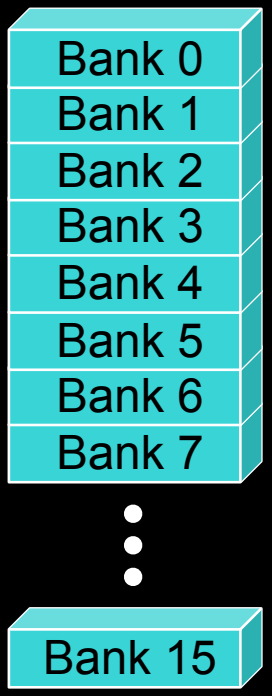
Example:
Square Matrix Multiplication Example

- C = A · B of size N x N
- With blocking:
  - One thread block handles one M x M sub-matrix C_{sub} of C
  - A and B are only loaded (N / M) times from global memory
- Much less bandwidth
- Much better balance of work to bandwidth

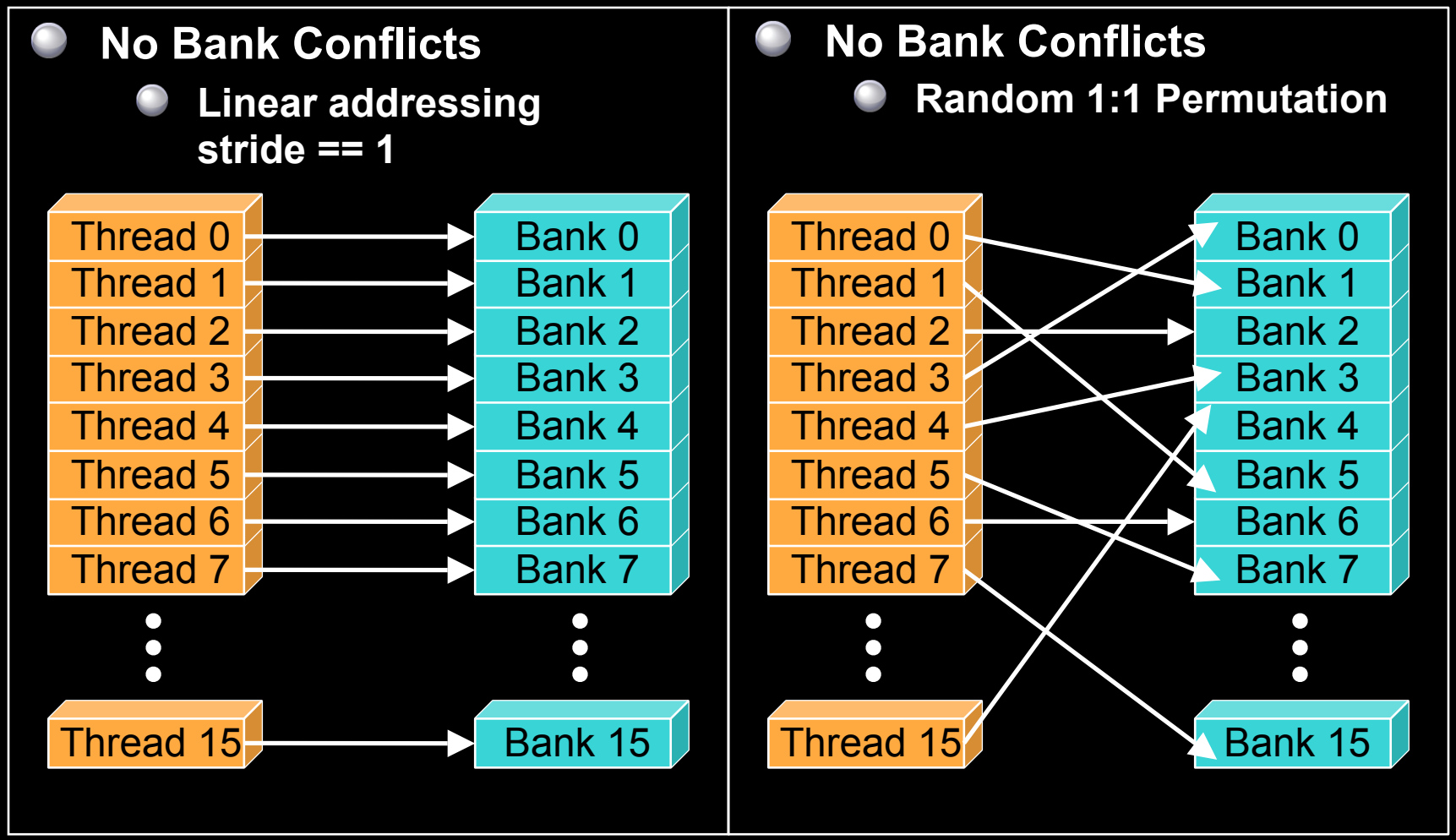


Shared Memory Architecture

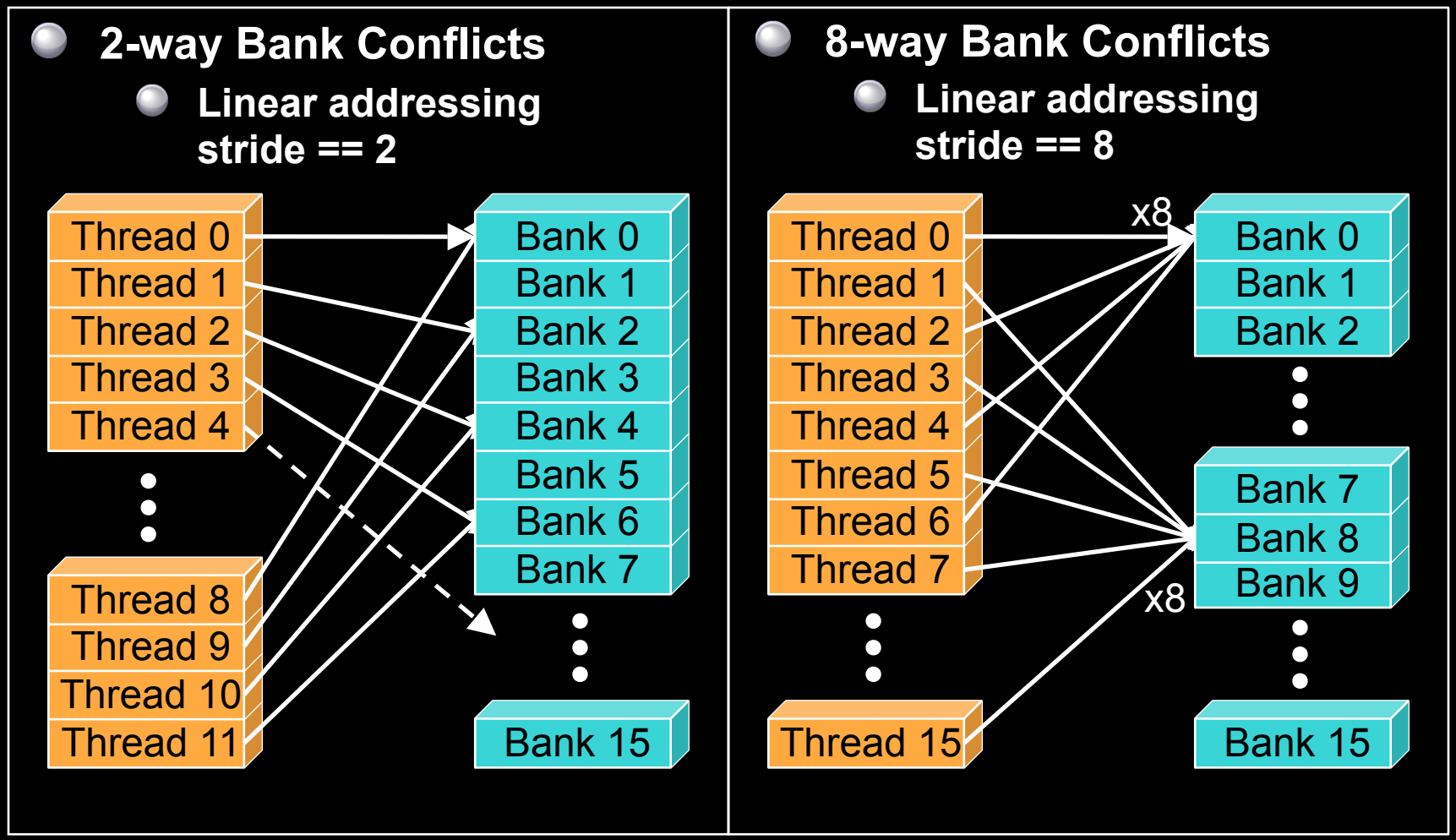
- Many threads accessing memory
  - Therefore, memory is divided into banks
  - Successive 32-bit words assigned to successive banks
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized



Bank Addressing Examples



Bank Addressing Examples

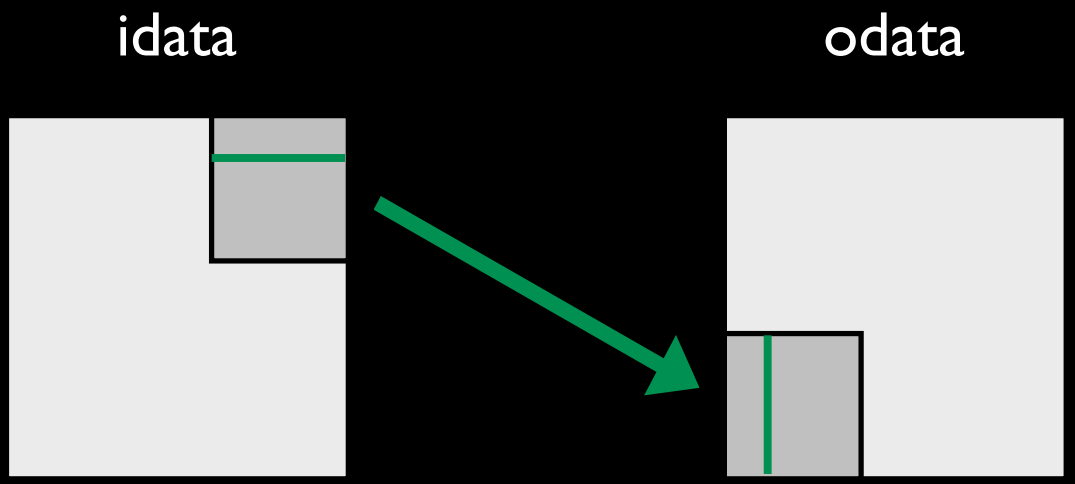


Shared memory bank conflicts

- Shared memory is ~ as fast as registers if there are no bank conflicts
- warp_serialize profiler signal reflects conflicts
- The fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp read the identical address, there is no bank conflict (broadcast)
- The slow case:
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

Shared Memory Example: Transpose

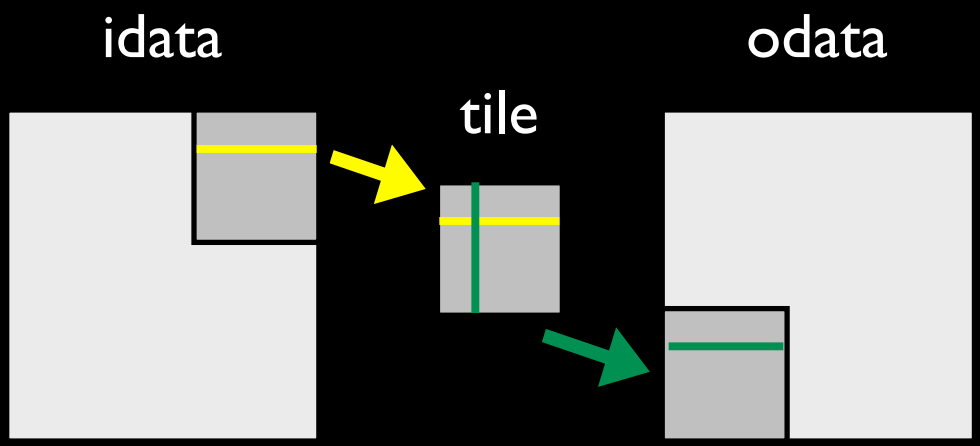
- Each thread block works on a tile of the matrix
- Naïve implementation exhibits strided access to global memory



Elements transposed by a half-warp of threads

Coalescing through shared memory

- Access columns of a tile in shared memory to write contiguous data to global memory
- Requires `__syncthreads()` since threads access data in shared memory stored by other threads



Elements transposed by a half-warp of threads

Outline

- Overview
- Hardware
- Memory Optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

Occupancy

- Thread instructions are executed sequentially, so 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
- Limited by resource usage:
  - Registers
  - Shared memory

Blocks per Grid Heuristics

- # of blocks > # of multiprocessors
  - So all multiprocessors have at least one block to execute
- # of blocks / # of multiprocessors > 2
  - Multiple blocks can run concurrently in a multiprocessor
  - Blocks that aren't waiting at a __syncthreads() keep the hardware busy
  - Subject to resource availability – registers, shared memory
- # of blocks > 100 to scale to future devices
  - Blocks executed in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations

Register Dependency

- **Read-after-write register dependency**
  - Instruction's result can be read ~24 cycles later
  - Scenarios:

CUDA:

```
x = y + 5;
z = x + 3;
```

PTX:

```
add.f32   $f3, $f1, $f2
add.f32   $f5, $f3, $f4
```

CUDA:

```
s_data[0] += 3;
```

PTX:

```
ld.shared.f32  $f3, [$r31+0]
add.f32        $f3, $f3, $f4
```

- **To completely hide the latency:**
  - Run at least **192** threads (6 warps) per multiprocessor
    - At least **25%** occupancy (1.0/1.1), **18.75%** (1.2/1.3)
  - Threads do not have to belong to the same thread block

Register Pressure

- Hide latency by using more threads per multiprocessor
- Limiting Factors:
  - Number of registers per kernel
    - 8K/16K per multiprocessor, partitioned among concurrent threads
  - Amount of shared memory
    - 16KB per multiprocessor, partitioned among concurrent threadblocks
- Compile with `-ptxas-options=-v` flag
- Use `-maxrregcount=N` flag to NVCC
  - `N` = desired maximum registers / kernel
  - At some point "spilling" into local memory may occur
    - Reduces performance – local memory is slow

Optimizing threads per block

- Choose threads per block as a multiple of warp size
  - Avoid wasting computation on under-populated warps
  - Facilitates coalescing
- More threads per block != higher occupancy
  - Granularity of allocation
  - Eg. compute capability 1.1 (max 768 threads/multiprocessor)
    - 512 threads/block => 66% occupancy
    - 256 threads/block can have 100% occupancy
- Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 192 or 256 threads a better choice
    - Usually still enough regs to compile and invoke successfully
  - This all depends on your computation, so experiment!