Parallel & Distributed Processing II: *parallel processing on manycore chips*

Nvidia GPUs and CUDA: Achieving High Performance

Eric Aubanel
Winter 2010, UNB Fredericton

Ideal CUDA programs

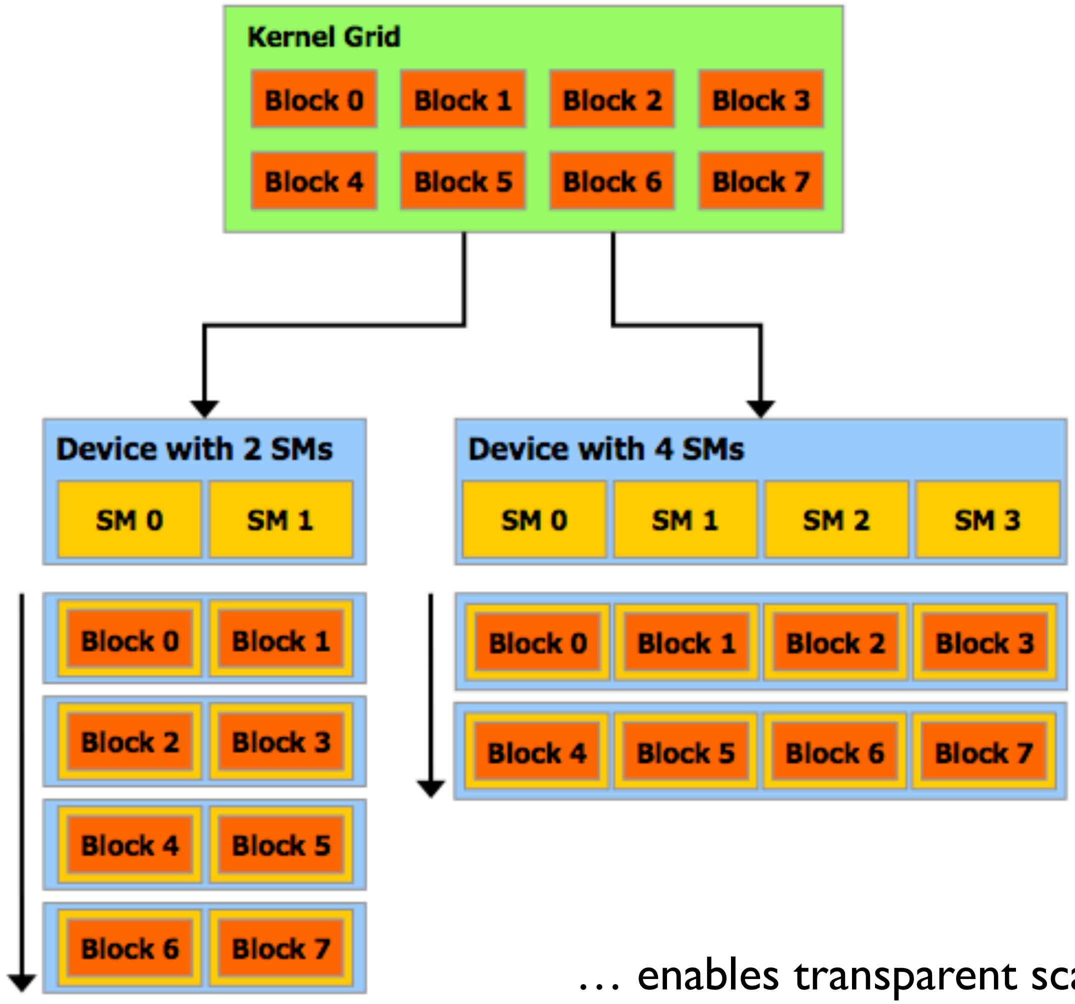
- High intrinsic parallelism
  - e.g. per-element operations
- Minimal communication (if any) between threads
  - limited synchronization
- High ratio of arithmetic to memory operations
- Few control flow statements
  - SIMD execution
    - divergent paths among threads in a block may be serialized (costly)

Source: John Mellor Crummey, Rice University

Some Factors Affecting Performance

- Ratio of arithmetic to memory operations (arithmetic intensity)
- Conditional execution
- Maximum number of threads
  - Large enough to fully occupy all SMs
- Block size
  - Occupancy
- Efficient use of global memory
  - Enabling coalescing of memory accesses
- Threads sharing data
  - Use fast shared memory
- Efficient use of shared memory
  - Bank conflicts
- Host-device data transfer
- …

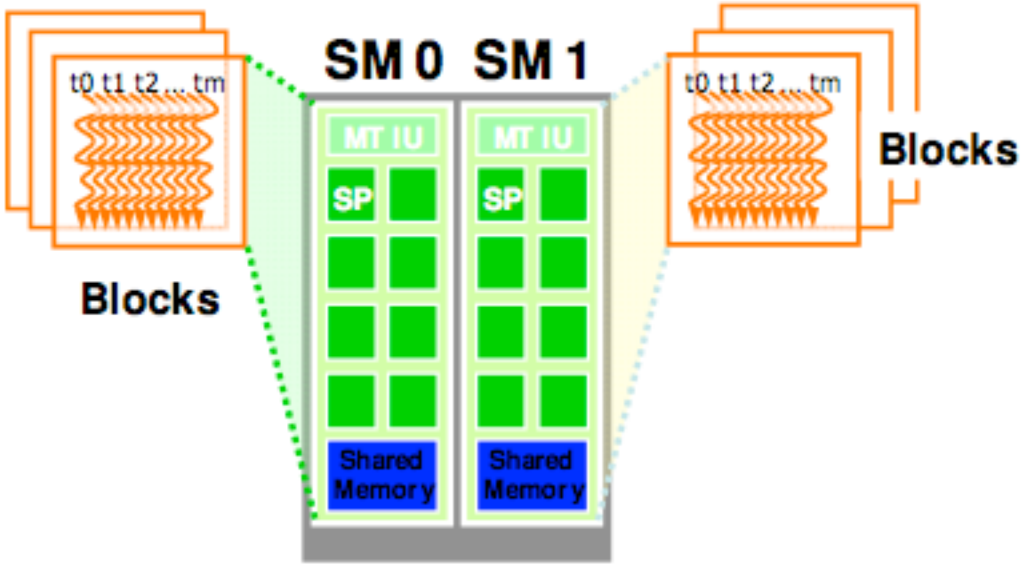
Block Scheduling



… enables transparent scalability

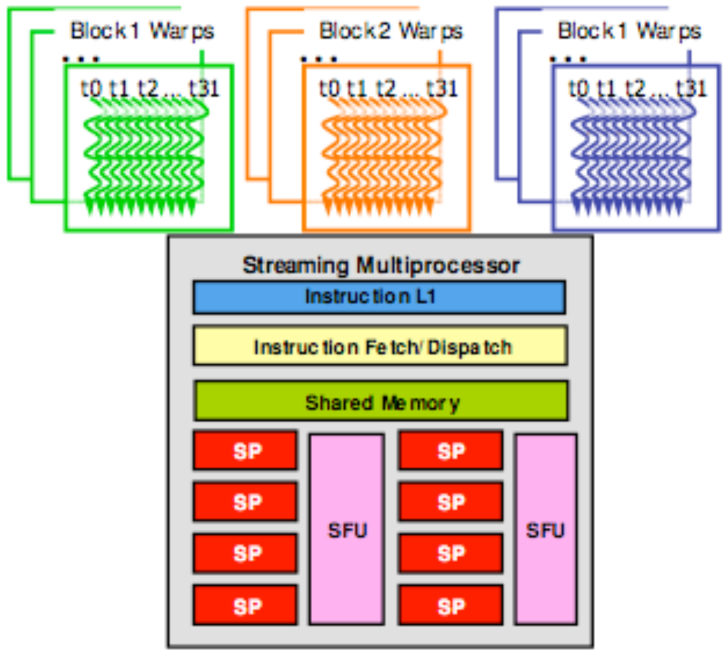
Block Scheduling

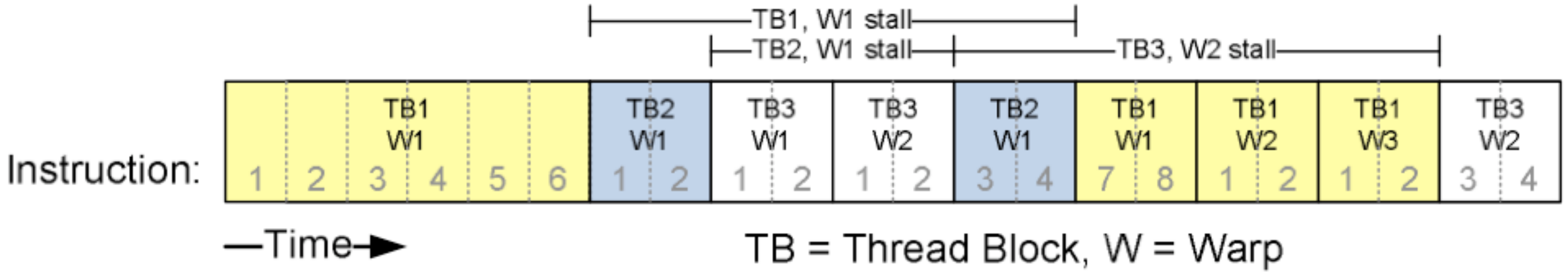
- Maximum number of active blocks per SM: 8
  - As long as sufficient resources
    - Max of 768 threads
    - Registers, shared memory



Source: Wen-Mei Hwu, U of Illinois and David Kirk, Nvidia

Blocks Divided into Warps



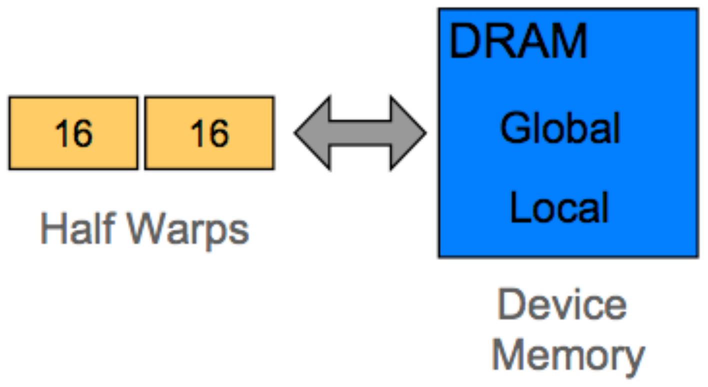


Source: Wen-Mei Hwu, U of Illinois and David Kirk, Nvidia

Occupancy

- (# active warps/SM)/(max # active warps = 24)
  - Blocks of 512 threads: only 1 block can be active at a time (since max 768 concurrent threads), so occupancy = 16/24 = 2/3.
  - Blocks of 128 threads: 6 blocks can be active giving 768 threads and full (24/24) occupancy
- Full occupancy also limited by:
  - Registers
    - 32KB or 64KB per multiprocessor, partitioned among concurrent threads
  - Shared memory
    - 16KB per multiprocessor, partitioned among concurrent thread blocks
- See occupancy calculator
  - http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

Coalescing Global Memory Accesses

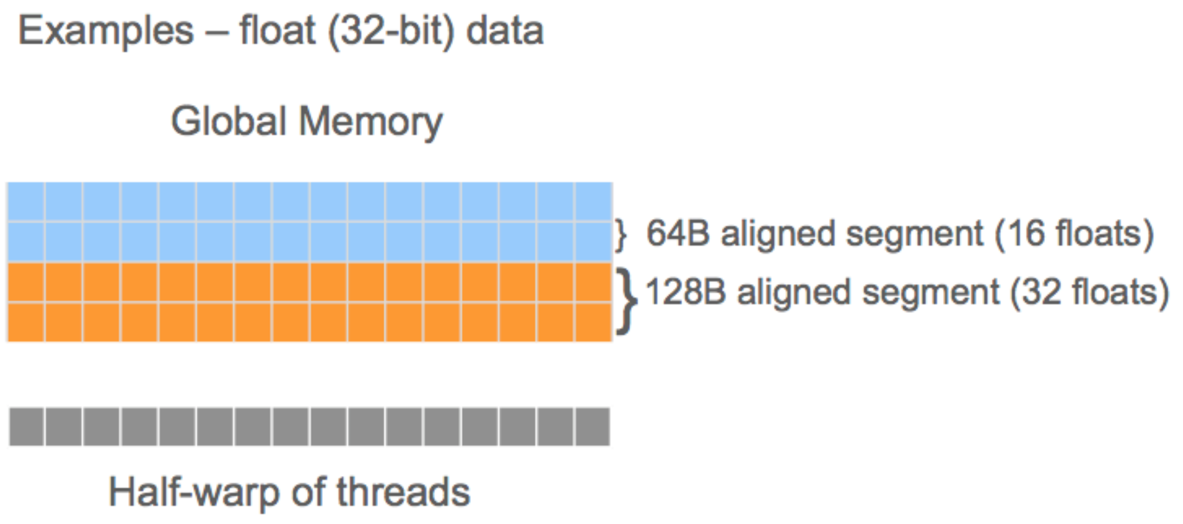


A half-warp of 16 threads can coordinate global memory accesses into a single transaction called coalescing

Dominik Göddeke, TU Dortmund

Coalescing

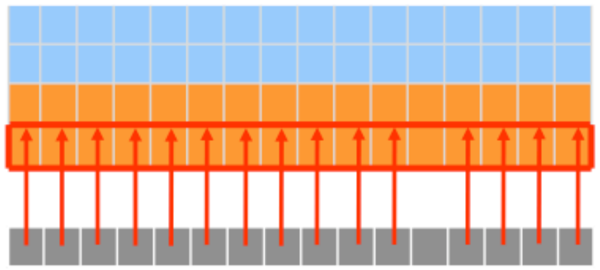
- Global memory bandwidth used most efficiently when the simultaneous memory accesses by threads in a half-warp can be coalesced into a single memory transaction of 32, 64, or 128 bytes. Certain access requirements must be met
  - Depends on compute capability (see programming guide)
  - 1.0 and 1.1 have stricter access requirements


Examples – float (32-bit) data
Global Memory
64B aligned segment (16 floats)
128B aligned segment (32 floats)
Half-warp of threads

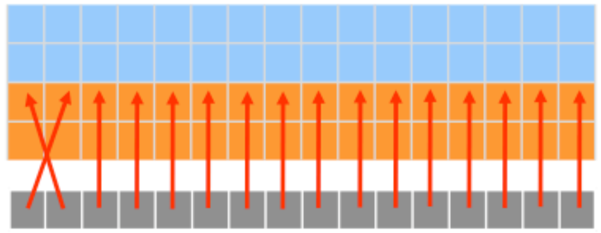
Dominik Göddeke, TU Dortmund

Coalescing

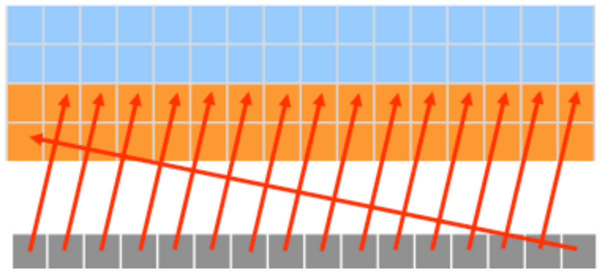
- Compute capability 1.0 and 1.1
  - K-th thread must access k-th word in the segment (or k-th word in two contiguous 128B segments for 128-bit words)
- Not all threads need to participate



Out of sequence – 16 transactions



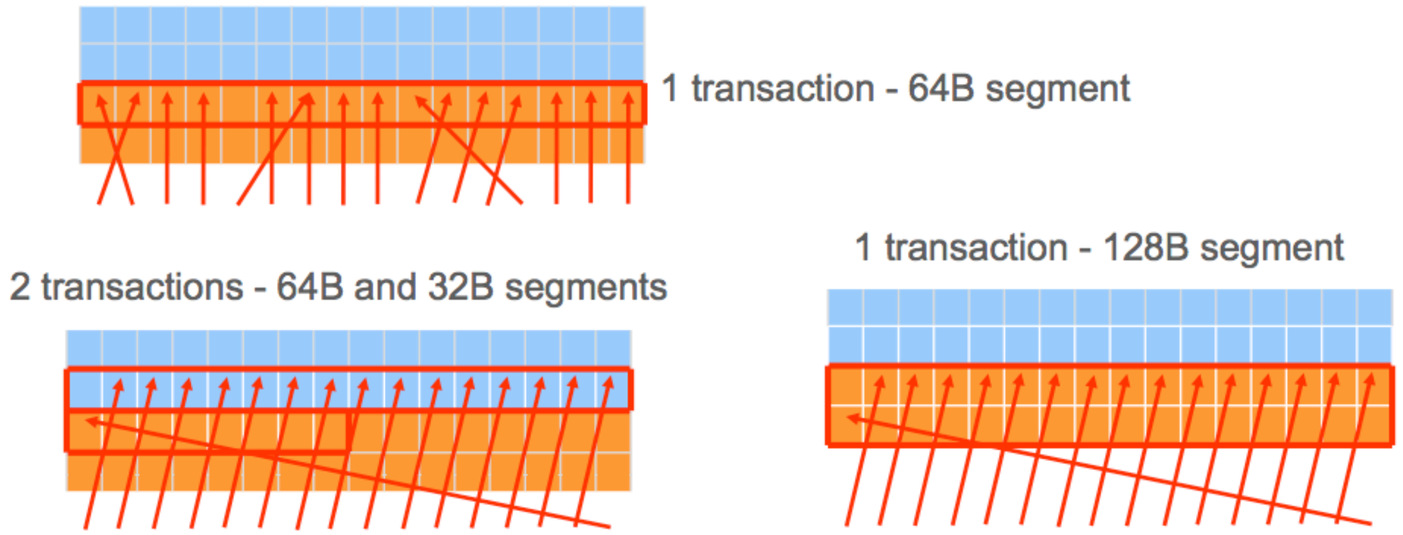
Misaligned – 16 transactions



Dominik Göddeke, TU Dortmund

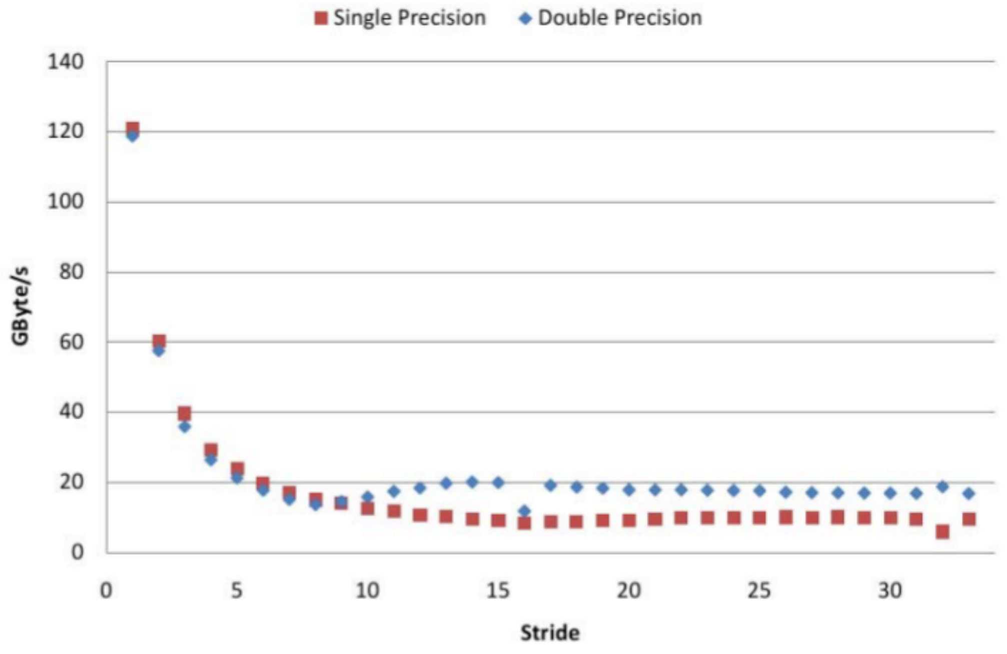
Coalescing

- 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

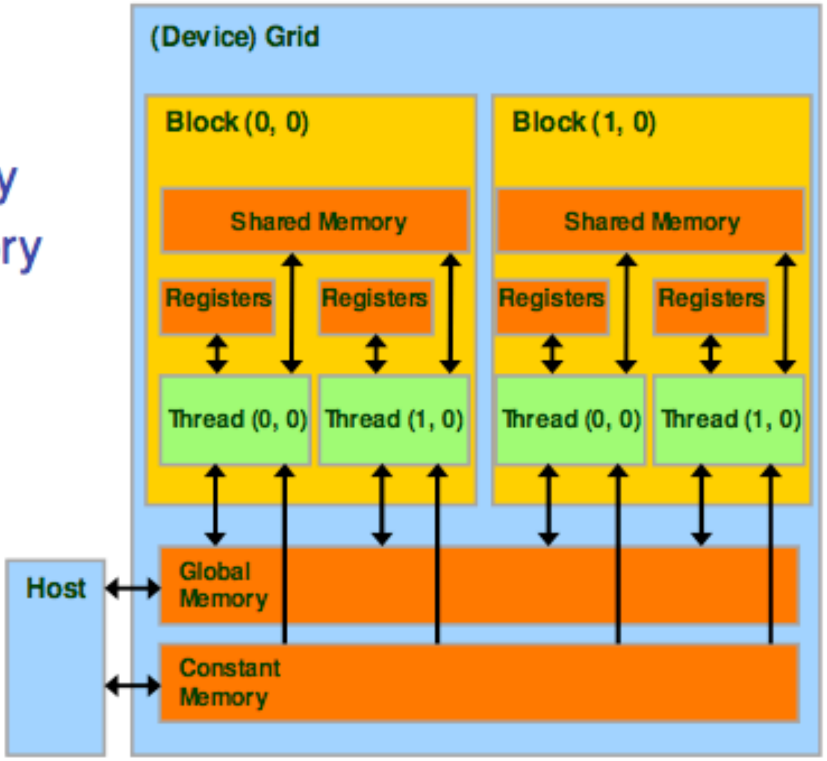
Importance of Coalescing

```
__global__
void saxpy_with_stride(int n, float a, float * x, float * y, int stride)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n)
        y[i * stride] = a * x[i * stride] + y[i * stride];
}
```



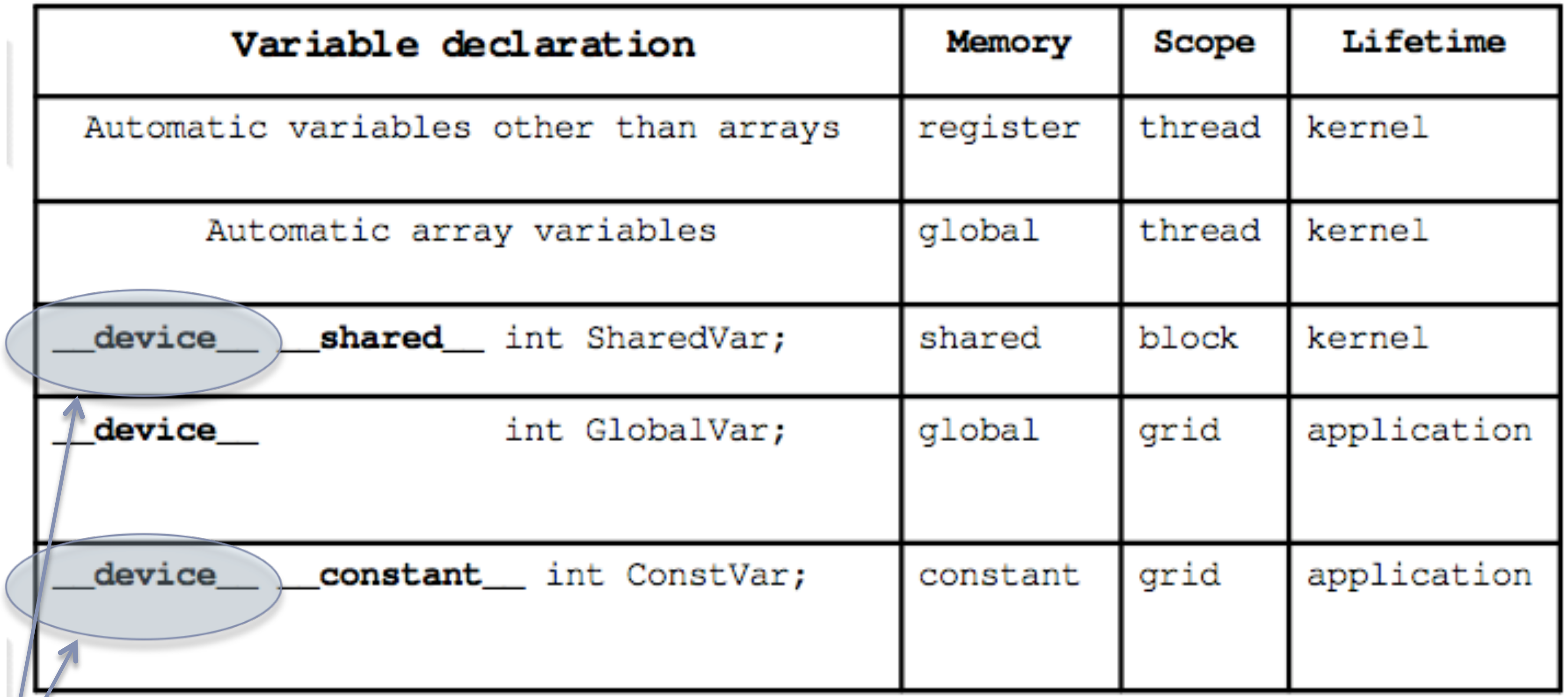
Recall: CUDA Memory Model

- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - R/W per grid global and constant memories



Source: Wen-Mei Hwu, U of Illinois and David Kirk, Nvidia

Accessing CUDA memories



Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	register	thread	kernel
Automatic array variables	global	thread	kernel
`__device__ __shared__ int SharedVar;`	shared	block	kernel
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstVar;`	constant	grid	application

optional

Source: Wen-Mei Hwu, U of Illinois and David Kirk, Nvidia

Shared Memory

- Each SM has 16 KB of Shared Memory
  - 16 banks of 32bit words
- CUDA uses Shared Memory as shared storage
  - visible to all threads in a thread block
- Useful when more than one thread in a block read same memory location
  - Reduce reads from global memory
- Can be used to take advantage of coalescing of global memory accesses
  - Coalesced copy to shared memory, followed by any pattern of access within shared memory
    - Except watch for bank conflicts

Shared Memory Allocation

- 2 modes
  - Static size within kernel

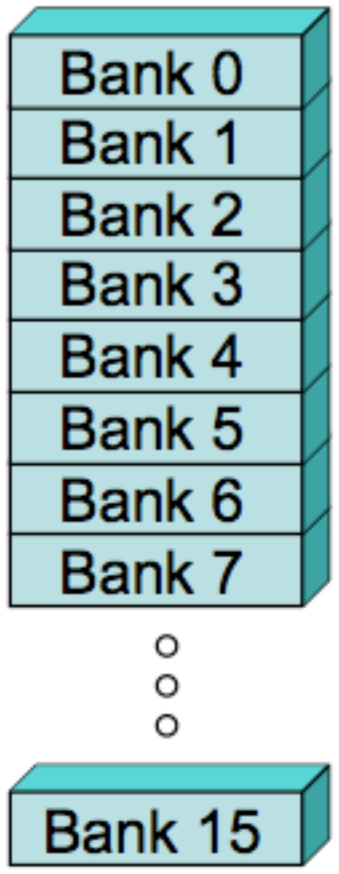
```
__shared__ float vec[256];
```

- Dynamic size when calling the kernel

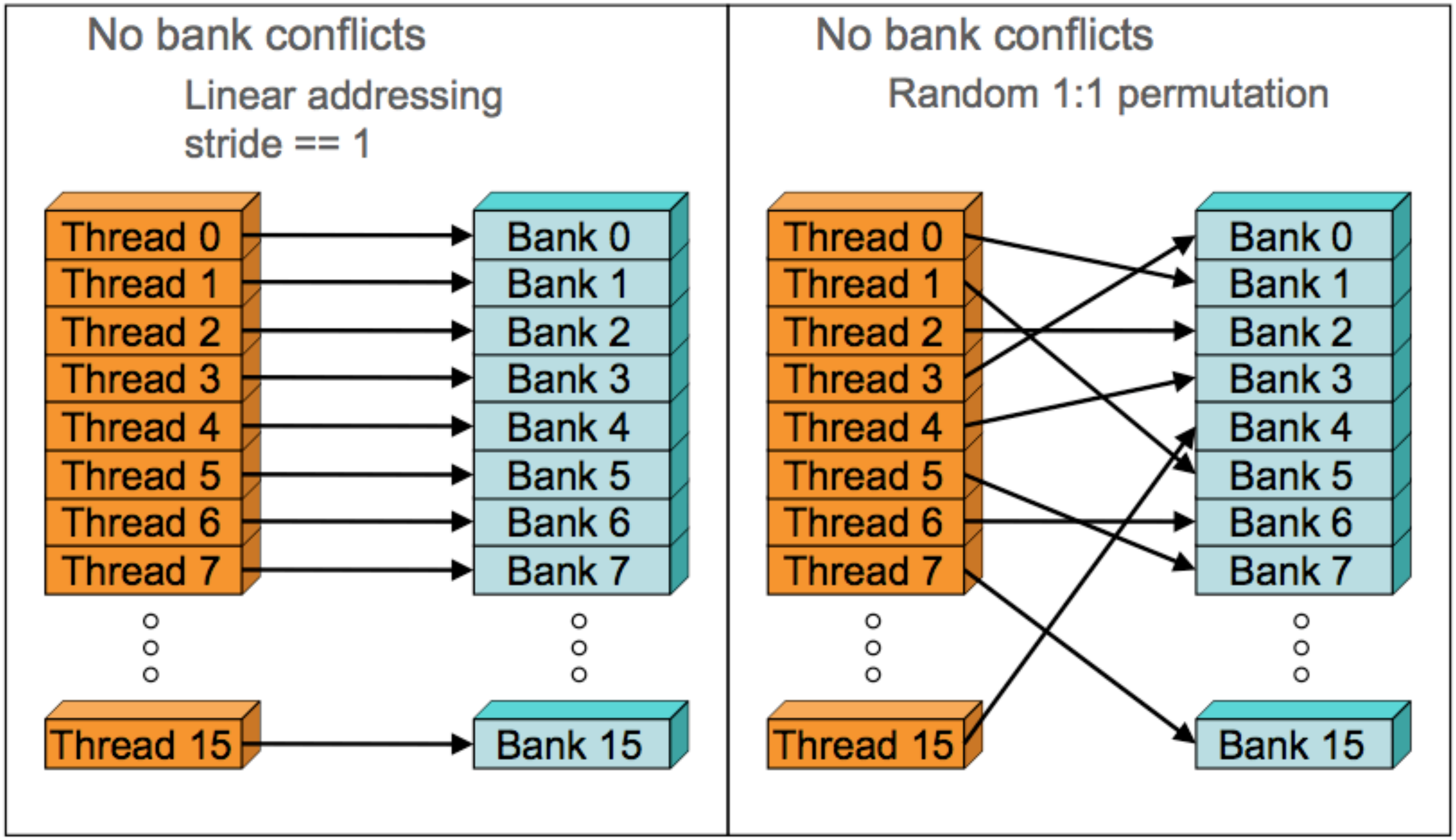
```
// in main
int VecSize= 256 * sizeof(float);
foo<<<nBlocks,blockSize, VecSize>>>(p1, p2, …);

// declare as extern within kernel
extern __shared__ float vec[];
```

Shared Memory Architecture

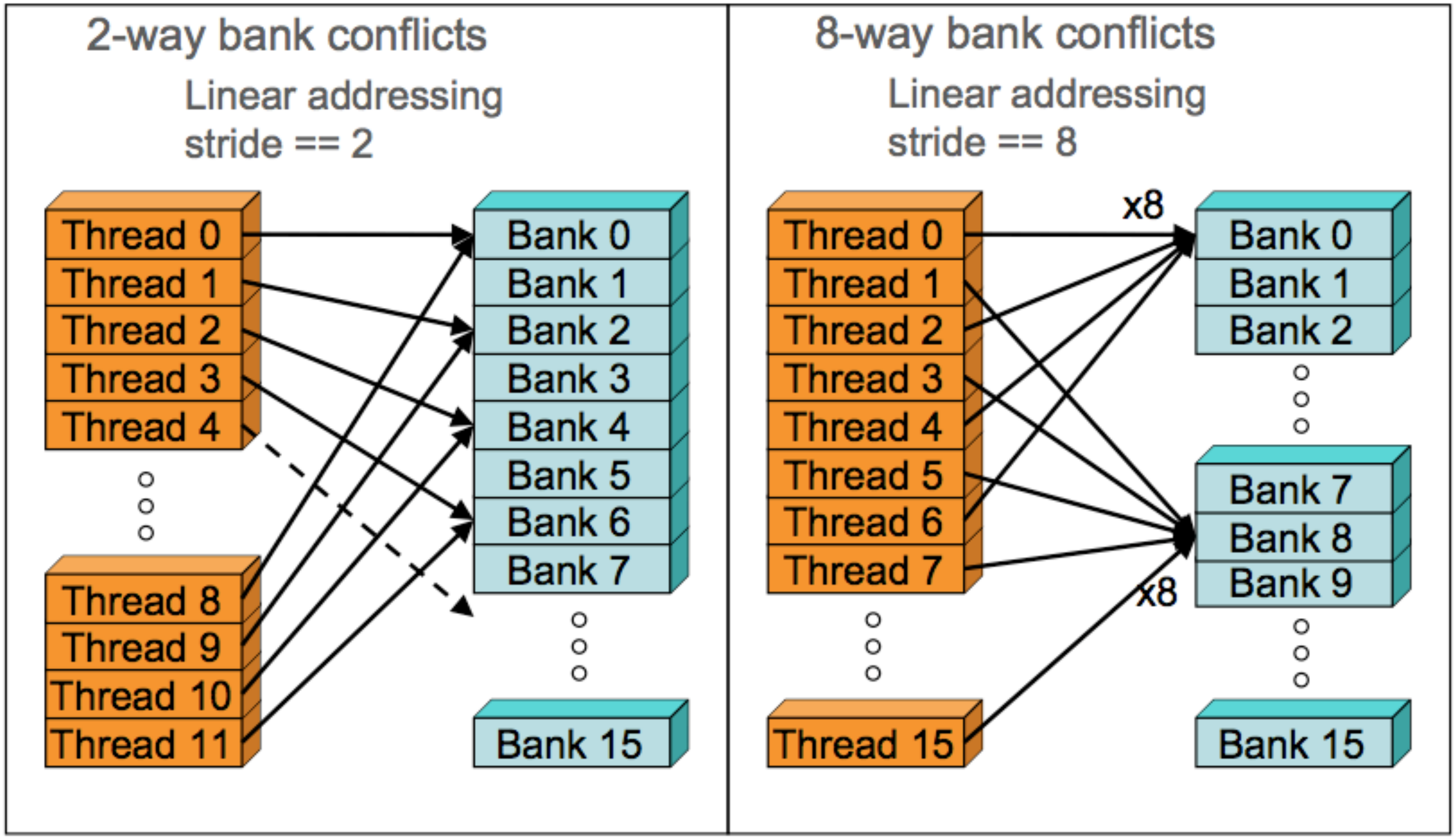
- Many threads accessing memory
  - Therefore, memory is divided in banks
  - Successive 32-bit words assigned to successive banks
- Each bank can serve one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a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



Bank Addressing Examples



Bank Addressing Examples



Bank Conflicts

- Shared memory is ~ as fast as registers
  - If there are no bank conflicts
  - warp_serialize profiler signal (see CUDA profiler)
- The fast case
  - If all threads of a half-warp access different banks, there are no bank conflicts
  - 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

CUDA Profiler

- Accessible via GUI (cudaprof) and text interface
- Four environment variables for textual profiler:

CUDA_PROFILE: Set to 1 or 0 to enable/disable the profiler

CUDA_PROFILE_LOG: Set to the name of the log file

(The default is ./cuda_profile.log)

CUDA_PROFILE_CSV: Set to 1 or 0 to enable or disable a comma separated version of the log

CUDA_PROFILE_CONFIG: Specify a configuration file with up to four signals

CUDA Profiler Signals

- **gld_incoherent:** Number of non-coalesced global memory loads
- **gld_coherent:** Number of coalesced global memory loads
- **gst_incoherent:** Number of non-coalesced global memory stores
- **gst_coherent:** Number of coalesced global memory stores
- **local_load:** Number of local memory loads
- **local_store:** Number of local memory stores
- **branch:** Number of branch events taken by threads
- **divergent_branch:** Number of divergent branches within a warp
- **instructions:** instruction count
- **warp_serialize:** Number of threads in a warp that serialize based on address conflicts to shared or constant memory
- **cta_launched:** executed thread blocks

Using the Profiler

- Profiler can only target one of the multiprocessors in the GPU
  - counter values will not correspond to the total number of warps launched for a particular kernel.
  - Best used to identify relative performance differences
- First set environment variables

```
export CUDA_PROFILE=1
export CUDA_PROFILE_CONFIG=
  $HOME/.cuda_profile_config
```

- Put up to four signals in `.cuda_profile_config`

gld_coherent

gld_incoherent

gst_coherent

gst_incoherent

- After running program, see profile data in log file.

Dr. Dobb's: CUDA, Supercomputing for the Masses: Part 6 (July 25, 2008)