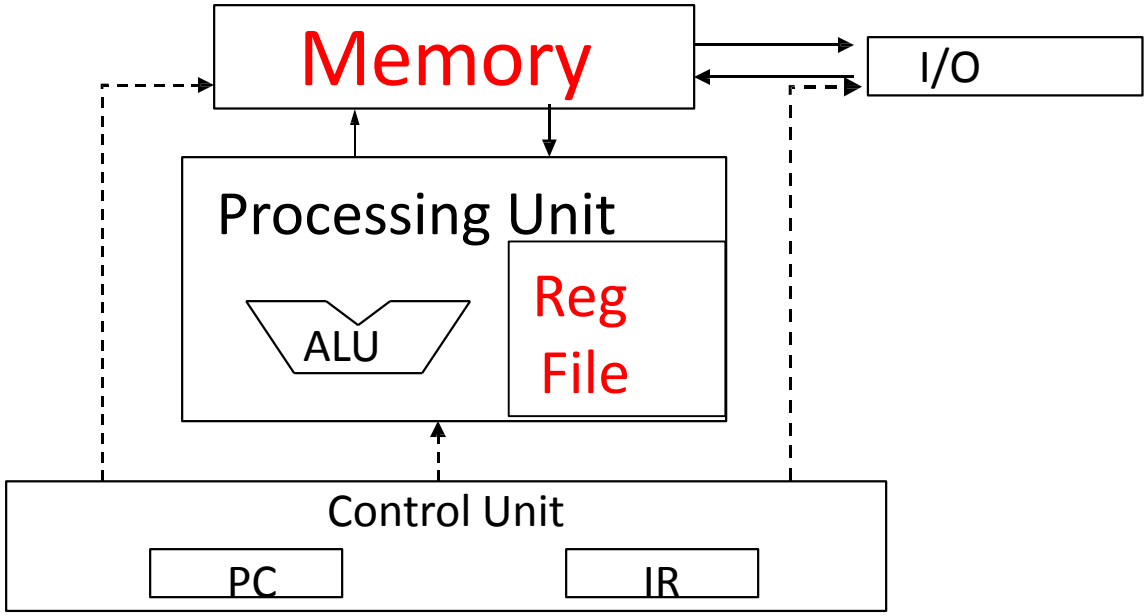
Parallel Programming

CUDA Memories and Optimizations

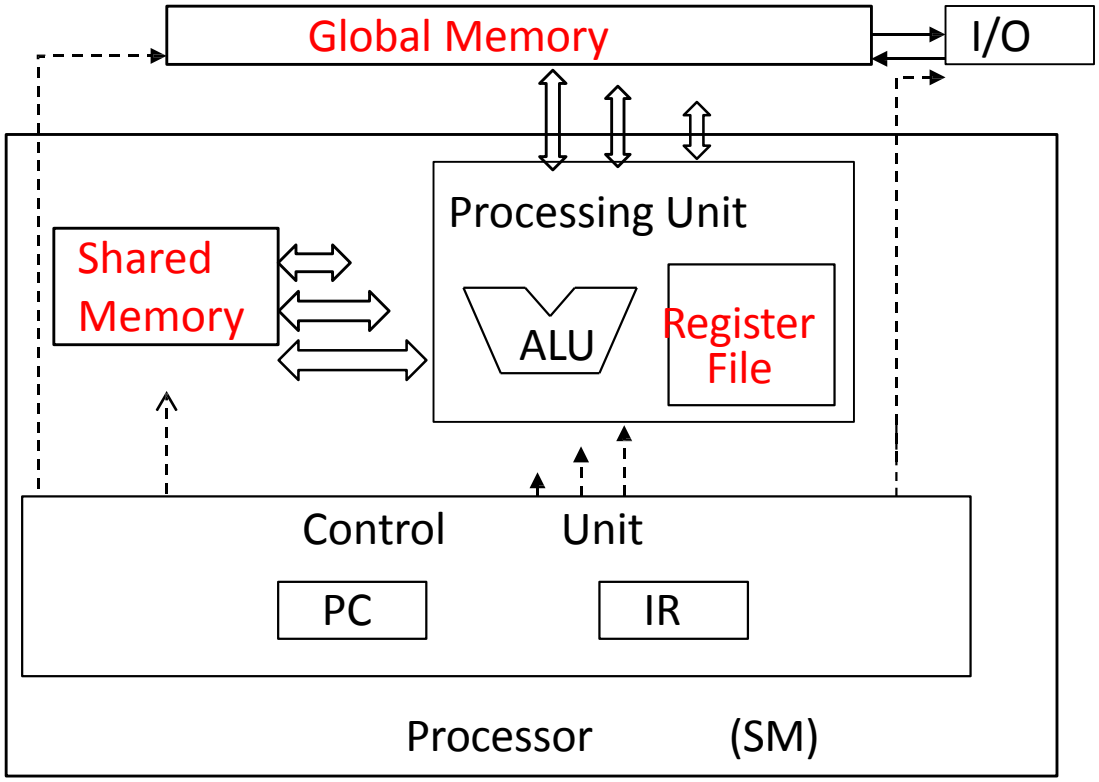
Overview

- CUDA Memories
  - Registers, shared memory, global memory
- Memory optimizations
  - General memory optimizations
  - Use of shared memory

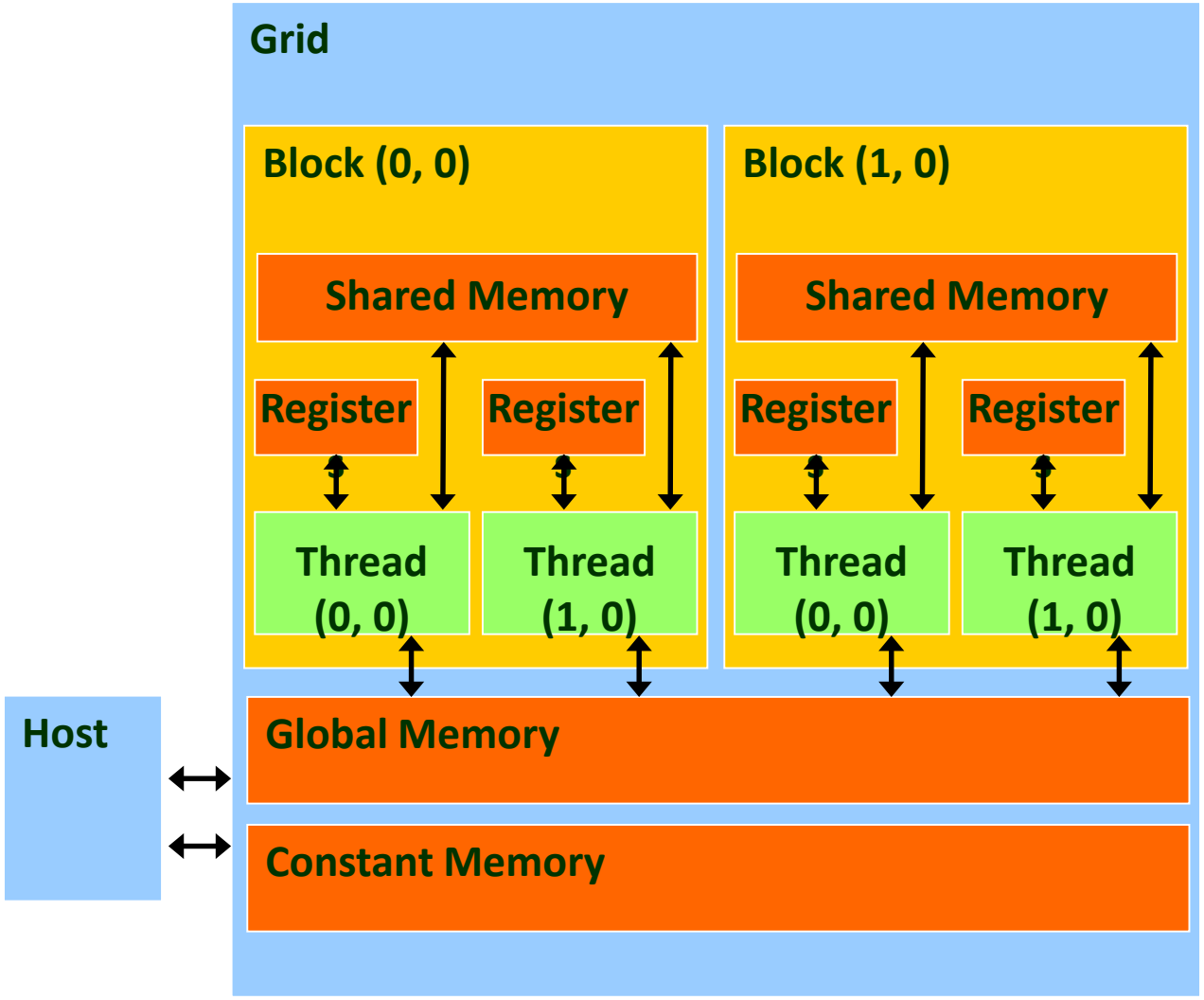
Memory and Registers in the Von-Neumann Model



CUDA Memories in a Similar Model



Programmer's View of CUDA Memories



Type Qualifiers of Device Variables

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

- __device__ is optional when used with __shared__, or __constant__
- Automatic variables (variables declared without any of these qualifiers) reside in a register except per-thread arrays that reside in global memory

Global Memory

- Resides in device memory (high latency + high bandwidth)
- Accessed via 32-, 64-, or 128-byte memory transactions
  - Addresses in a transaction must be aligned to these sizes.
  - Memory accesses of the threads within a warp are coalesced into one or more of memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads.

Local Memory

- Resides in device memory
  - Same latency and bandwidth as global memory access
  - Same requirements for memory coalescing
  - Access cached same way as global memory access
- Organized such that consecutive 32-bit words are accessed by consecutive thread IDs
  - Accesses are therefore fully coalesced as long as all threads in a warp follow the access pattern

Constant Memory

- Resides in device memory
- Cached in the constant cache
- Accesses are split into separate memory requests depending on the addresses.
  - Each request is serviced at the throughput of the constant cache in case of a cache hit, or at the throughput of device memory otherwise.

Shared Memory

- On-chip
  - Much higher bandwidth and much lower latency than local or global memory
- Divided into equally-sized memory modules, called banks, which can be accessed simultaneously
  - If two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized.

Texture and Surface Memory

- Reside in device memory
- Cached in texture cache
  - A texture fetch or surface read costs one memory read from device memory only on a cache miss, otherwise it just costs one read from texture cache.
- The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture or surface addresses that are close together in 2D will achieve best performance.

Details of CUDA Memories

Memory	Location	Cached	Access	Who	Latency
Register	On-chip	Resident	Read/write	One thread	O(1 cycle)
Shared	On-chip	Resident	Read/write	Threads in block	O(1 cycle) w/o conflict
Global	Off-chip	No/Yes	Read/write	All threads + host	O(1)- O(100) cycles, depending on if cached
Local	Off-chip	No/Yes	Read/write	One thread	O(1)- O(100) cycles, depending on if cached
Constant	Off-chip	Yes	Read only	All threads + host (host may write)	O(1)-O(100) cycles, depending on if cached
Texture	Off-chip	Yes	Read only	All threads + host (host may write)	O(1)- O(100) cycles, depending on if cached
Surface	Off-chip	Yes	Read/write	All threads+host	O(1)-O(100) cycles, depending on if cached

Targets of Memory Optimizations

- Reduce ***memory latency***
  - The latency of a memory access is the time (usually in cycles) between a memory request and its completion
- Maximize ***memory bandwidth***
  - Bandwidth is the amount of useful data that can be retrieved over a time interval
- Manage overhead
  - Cost of performing optimization (e.g., copying) should be less than anticipated gain

Reuse and Locality

- Consider how data is accessed
  - ***Data reuse:***
    - Same data used multiple times
    - Intrinsic in computation
  - ***Data locality:***
    - Data is reused and is present in “fast memory”
    - Same data or same data transfer
  - If a computation has reuse, what can we do to get locality?
    - Appropriate data placement and layout
    - Code reordering transformations

Data Placement: Conceptual

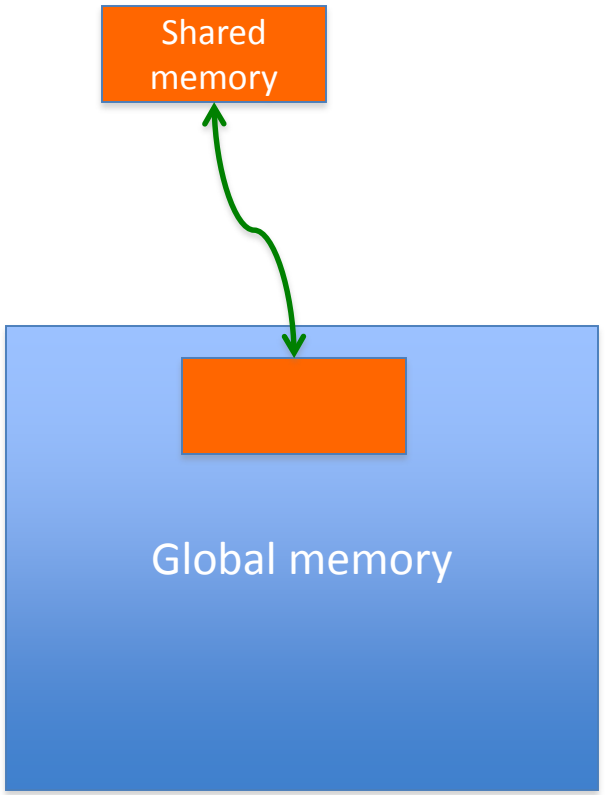
- Copies from host to device go to some part of global memory (possibly, constant or texture memory)
- How to use shared memory
  - Must construct or be copied from global memory by kernel program
- How to use constant or texture cache
  - Read-only “reused” data can be placed in constant & texture memory by host
- How to use registers
  - Most locally-allocated data is placed directly in registers
  - Even array variables can use registers if compiler understands access patterns
  - Can allocate vectors to registers, e.g., float4
  - Excessive use of registers will “spill” data to local memory

Data Placement: Syntax

- Through type qualifiers
  - __constant__, __shared__, __device__
- Through cudaMemcpy calls
  - Any directions between host and device memories
- Implicit default behavior
  - Device memory without other qualifier is global memory
  - Host by default copies to global memory
  - Thread-local variables go into registers unless capacity exceeded, then local memory

Common Programming Pattern of Using Shared Memory

- Load data into shared memory
- Synchronize (if necessary)
- Operate on data in shared memory
- Synchronize (if necessary)
- Write intermediate results to global memory
- Repeat until done



Mechanics of Using Shared Memory

- __shared__ type qualifier required
- Must be allocated from global/device function, or as "extern"
- Examples:

```
extern __shared__ float  d_s_array[];

__host__ void outerCompute() {
  compute<<<gs,bs>>>();
}
__global__ void compute() {
   d_s_array[i] = …;
}
```

```
__global__ void compute2() {
__shared__ float d_s_array[M];

  // create or copy from global memory
  d_s_array[j] = …;
  //synchronize threads before use
  __syncthreads();
  … = d_s_array[x]; // now can use any element

 // more synchronization needed if updated
  __syncthreads();
  // may write result back to global memory
  d_g_array[j] =  d_s_array[j];
}
```

Tiling for Limited Capacity Storage

- Tiling can be used hierarchically to compute partial results on a block of data wherever there are capacity limitations
  - Between grids if total data exceeds global memory capacity
  - Across thread blocks if shared data exceeds shared memory capacity (also to partition computation across blocks and threads)
  - Within threads if data in registers exceeds register capacity or data in shared memory for block still exceeds shared memory capacity

Summary

- Device variables reside in the global memory, the shared memory, or registers.
- CUDA memories have different latency and bandwidth characteristics.
- Memory optimizations can be done through data placement and reuse.
- Tiling for the shared memory is a common memory optimization in CUDA programming.