CSC367 Parallel computing

Lecture 19: General-purpose computing with Graphics Processing Units (GPUs)

(Continued)

Memory and access techniques

- Memory types
- Memory coalescing
- Shared memory and bank conflicts

Memory types

- Global memory
- Local memory
- Shared memory
- Constant memory
- Texture memory

Local and Global memory

- Global memory
  - Most data resides here
  - Host communication (this is where data gets transferred from/to CPU memory)
  - Shared by all threads
  - Large size (a few GB typically), but slower than shared memory
  - L1 and L2 cache helps hide the latency for global (and local) memory accesses
  - Good bandwidth via memory coalescing

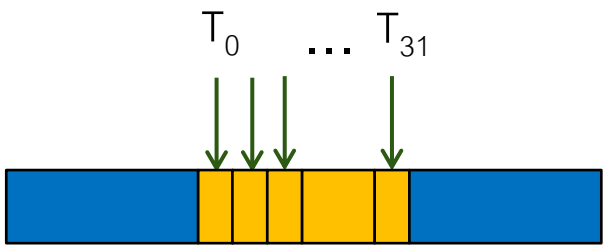
Local and Global memory

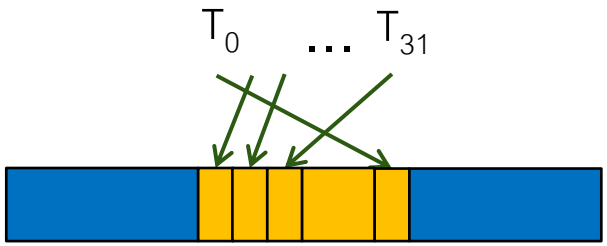
- Local memory is just an abstraction of global memory and is not a physical type of memory (keep in mind: terms used in CUDA)
  - Scope is local to a thread
  - aka Private per thread global memory
  - Automatic variables, register spill
  - Same speed as global memory but accesses are coalesced! (see next slide!)

Memory coalescing

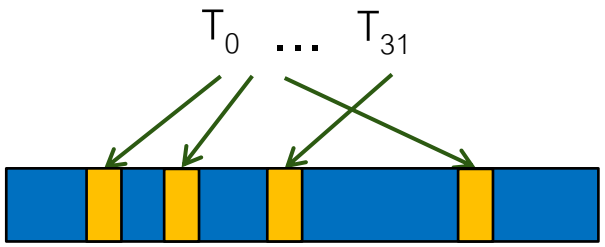
- Warp accesses should reference sequential memory locations for best performance => these accesses get coalesced into a single access

Coalesced accesses



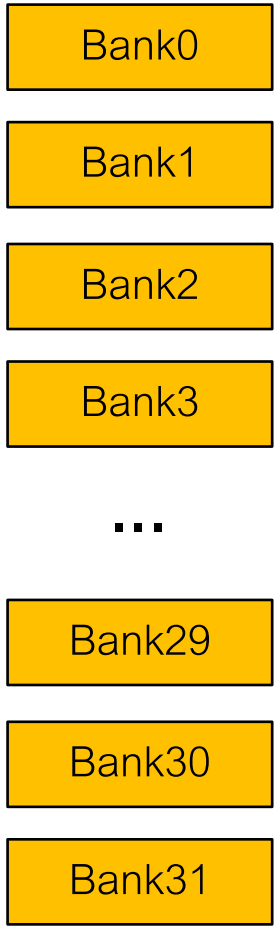


Scattered accesses



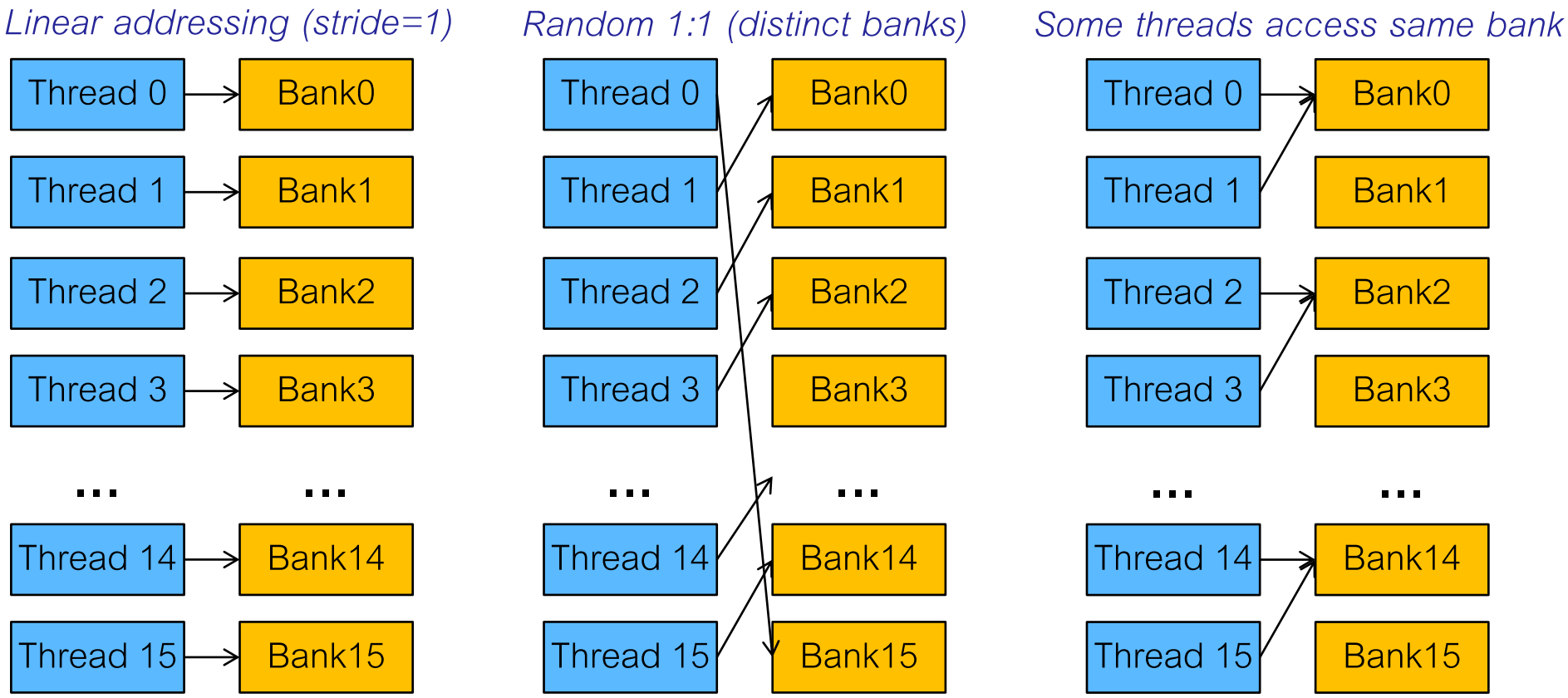
Shared memory

- Lower latency than global memory
- Acts as software programmable cache (it's a chunk of L1!)
  - Declare intention by using __shared__ keyword
- Organized in 32 banks
  - Successive 32-bit words are assigned to successive banks
  - Any memory load/store of N addresses spanning N distinct memory banks can be serviced simultaneously => N times the bandwidth of a single bank!
  - Bandwidth of shared memory: 32-bits per bank per cycle
- Bank conflicts: intuitively, it's the failure to distribute the threads' accesses across memory banks



Bank conflicts

- When threads in a warp access different 32-bit words from the same bank
- Must avoid bank conflicts! => Design code accordingly
- Threads accessing bytes within the same 32-bit word is ok though => no conflict



Shared memory – key observations

- Much faster than global memory, it's a "controllable" part of L1 cache
  - Configurable amount on some cards!
- Shared memory is shared by threads in a block => provides a mechanism for threads to cooperate!
  - When necessary, use __syncthreads() for block-level barriers
- Facilitates global memory coalescing in cases where it would not otherwise be possible
  - Does not have the sequential access restrictions of global memory, to achieve optimal performance
- Only need to avoid bank conflicts
  - Otherwise accesses get serialized => poor performance, potentially worse than global memory

So far ... Key takeaways!

- Must have enough parallelism
  - At least a few thousands of threads executing concurrently
  - Keep the cores busy and benefit from high memory bandwidth
- Coalesced memory access
  - Accesses to sequential memory locations by threads in a warp are very fast
  - Not as crucial on newer GPUs / compute capabilities, but still a big performance hit!
- Coherent execution
  - Threads in a warp are automatically synchronized (proceed in lock step)
  - Careful with warp divergence
- Shared memory
  - Fast but must avoid bank conflicts
- Rework your data access patterns when necessary!

Next up ...

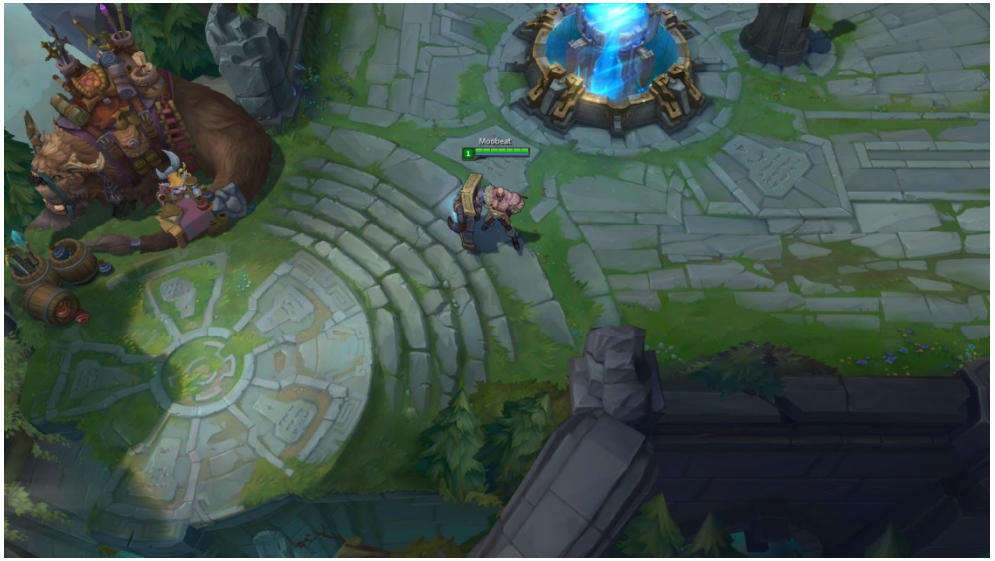
- Constant memory
- Texture memory
- Atomics

Constant memory

- Used for data which does not change during kernel execution
  - => Constrains usage of that data to be read-only
- Small amount: typically 64KB
- Advantages: handled differently than global memory
  - 1. A single read can be broadcast to a half-warp => saves up to 15 reads, helps reduce the required memory bandwidth by 94%
  - 2. Constant memory is cached => consecutive reads from same address will not incur any additional memory traffic
- Disadvantage: half-warp threads must read from same location
- Use __constant__ modifier to indicate data is stored there

Texture memory

- What is a texture?



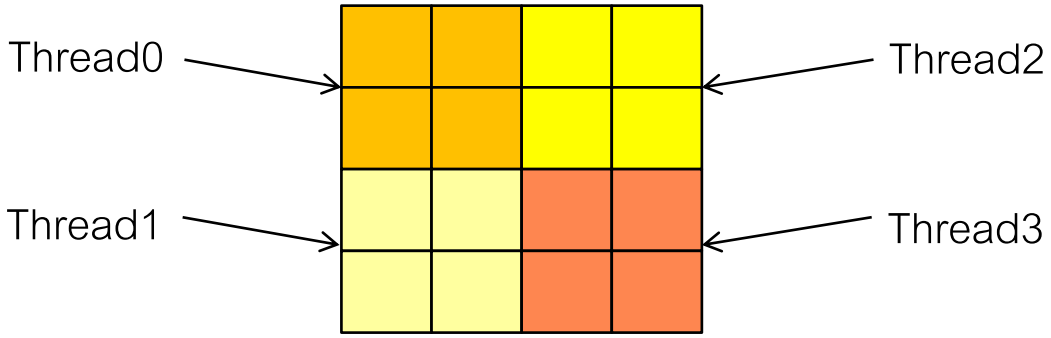
Img src: © Riot Games

Img src: © Blizzard Entertainment

- Texture memory normally were not that easy to use
  - Special steps to follow due to legacy use (binding, unbinding textures, etc.)
- As of CUDA5.0, bindless textures (cudaTextureObject_t class API)
- Useful to squeeze every bit of performance in the right kind of use case

Texture memory in a nutshell...

- A special type of read-only memory
- Although typically used for OpenGL/DirectX rendering pipelines, these can be used for general-purpose computing too
- Advantages:
  - Like constant memory, it's cached on chip => more effective bandwidth by reducing requests to off-chip (global) memory
  - Useful when memory access patterns exhibit good spatial locality (i.e., a thread reads from memory locations "nearby" where other threads also read)



Texture memory in a nutshell…

- When to use?
  - If you rarely update your data, but often read it
  - If the read access pattern presents some spatial locality ("nearby" threads access (spatially) nearby locations in the texture)
- When not to use?
  - When data is read rarely (e.g., only once) after it's written
- See examples in textbook (old-style though, requires binding/unbinding, etc.) and recent CUDA documentation

CUDA Variables

Variable declaration	Memory	Scope	Lifetime
__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__**

Atomic Operations

- Reminder: necessary to avoid race conditions on shared data
- Example: x = x + a
  - 1. Read value of x
  - 2. Add a to the value read in step 1
  - 3. Write result back to x
- Multi-threaded environment: race conditions can lead to wrong results
  - Must have a way to perform this **read-modify-write** atomically
- CUDA supports several atomic operations:
  - atomicAdd, atomicMin, etc.

Atomics: take-aways

- Atomics are great for many use cases
  - Check out the documentation for more operations
- Do not rely on atomics blindly
  - Can lead to serious performance degradation
  - Must understand what is the impact of atomics in that specific context
  - Rethink the algorithm when problems arise!

Memory Allocation

- Shared and global memory allocation

Memory Allocation

- Data requires to be transferred from host to device
  - Data is transferred across PCIe bus

- Pointers are used to allocate addresses on GPU and CPU
  - Dereferencing CPU pointers on GPU and vise versa will likely crash

- Device memory managed from CPU
  1. cudaMalloc (void ** pointer, size_t nbytes)
  2. cudaMemcpy (void *dst, void *src, size_t nbytes, enum memcopykind)
  3. cudaMemset (void * pointer, int value, size_t count)
  4. cudaFree (void* pointer)

Memory Allocation

- cudaMalloc (void ** pointer, size_t nbytes)
  - Pointer points to an object on GPU Global memory
  - nbytes: Size of allocated object

- cudaMemcpy (void *dst, void *src, size_t nbytes, enum Type of transfer)
  - Host to Device : cudaMemcpyHostToDevice
  - Device to Host : cudaMemcpyDeviceToHost
  - Device to Device : cudaMemcpyDeviceToDevice

- cudaMemset (void * pointer, int value, size_t count)

- cudaFree (void* pointer)

Memory Allocation

```
Void main()
{  ...
dim3 blockDim (blocksize);
dim3 gridDim (K / (float)blocksize) );
increment_gpu <<< gridDim, blockDim>>>(da, b, K);

}
```

Kernel

Memory Allocation

```
Void main()
{  ...
int num_bytes= K * sizeof (float);
float  *da, *ha = 0;   //host and device pointers

dim3 blockDim (blocksize);
dim3 gridDim (K / (float)blocksize) );
increment_gpu <<< gridDim, blockDim>>>(da, b, K);

}
```

Kernel

Memory Allocation

```
Void main()
{  ...
int num_bytes= K * sizeof (float);
float  *da, *ha = 0;   //host and device pointers

ha= (float *) malloc (num_bytes) ;
cudaMalloc (&da, num_bytes) ;

dim3 blockDim (blocksize);
dim3 gridDim (K / (float)blocksize) );
increment_gpu <<< gridDim, blockDim>>>(da, b, K);

}
```

Kernel

Memory Allocation

```
Void main()
{  ...
int num_bytes= K * sizeof (float);
float  *da, *ha = 0;   //host and device pointers

ha= (float *) malloc (num_bytes) ;
cudaMalloc (&da, num_bytes) ;

cudaMemcpy (da, ha, num_bytes, cudaMemcpyHostToDevice);
dim3 blockDim (blocksize);
dim3 gridDim (K / (float)blocksize) );
increment_gpu <<< gridDim, blockDim>>>(da, b, K);
cudaMemcpy (ha, da, num_bytes, cudaMemcpyDeviceToHost);
}
```

Kernel

Memory Allocation

```
Void main()
{  ...
int num_bytes= K * sizeof (float);
float  *da, *ha = 0;   //host and device pointers

ha= (float *) malloc (num_bytes) ;
cudaMalloc (&da, num_bytes) ;

cudaMemcpy (da, ha, num_bytes, cudaMemcpyHostToDevice);
dim3 blockDim (blocksize);
dim3 gridDim (K / (float)blocksize) );
increment_gpu <<< gridDim, blockDim>>>(da, b, K);
cudaMemcpy (ha, da, num_bytes, cudaMemcpyDeviceToHost);
Free (ha);
cudaFree(da);
}
```

Kernel