CSC367 Parallel computing

Lecture 17: General-purpose computing with Graphics Processing Units (GPUs)

(Continued)

Up next...

- CUDA programming basics, examples
- Grids and blocks and threads
- Parallel execution model
- Limitations

Useful tools

- Later versions of CUDA include Nsight, and IDE on Eclipse (includes a profiler and debugger too!)
- Visual Studio edition of Nsight for Windows
- The command-line profiler (nvprof), or the visual profiler (nvvp) – useful to analyze your programs
  - nvprof's "Profiling result" section – useful to know where is time being spent
  - nvprof's API trace can be turned off (--profile-api-trace none) to reduce some profiling overhead for short kernels
  - nvvp's timeline – visually see how the execution looks like
  - For more info, consult NVIDIA's documentation or man pages (nvprof --help)
  - Recall that profiling your code can save you a lot of trouble

Debugging and memory checks

- cuda-memcheck (runtime error checker tool for memory accesses, similar to valgrind to some extent) – some instability issues in the past
- Good ole' printing will only get you so far ...
  - Printing on the device involves data transfers (even in CUDA's cuPrintf…)
  - You *can* still do it, but it will be painful
- cuda-gdb: must compile code with: -g (host code), -G (device code)
- Example:
  - `$ nvcc -g -G program.cu -o program`
  - Similar to gdb (run, continue, bt, info, kill, break, print, next, step, quit, etc.)
  - cuda-gdb has cuda-memcheck integrated (Use: `set cuda memcheck on`)
- Quick guide: http://developer.download.nvidia.com/GTC/PDF/1062_Satoor.pdf

CUDA C (Typical) program

allocate memory on CPU (on the "host")

allocate memory on GPU (on the "device")

transfer data to device memory

launch kernel/s

wait to finish

transfer data back to host memory (if necessary)

The basics…

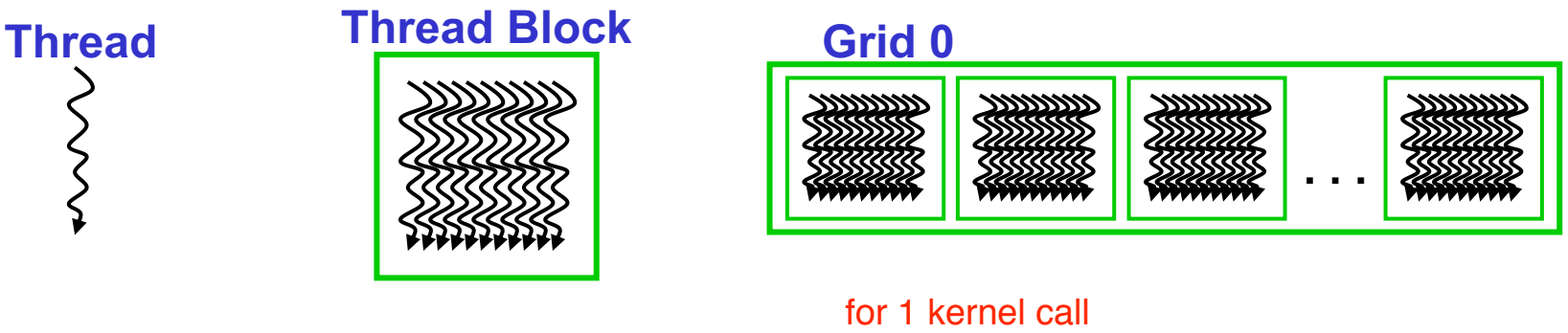
- Remember terminology
  - Host = CPU
  - Device = GPU
- Kernel launched with: <<<>>> - blocks, threads (more on this later)

Qualifier	Executed on the:	Callable from:	Conditions
`__device__`	Device	Device	
`__global__`	Device	Host*	
`__host__`	Host	Host	Can remove the extension word

* Callable from the device for devices of compute capability 3.2 or higher (see CUDA Dynamic Parallelism for more details).

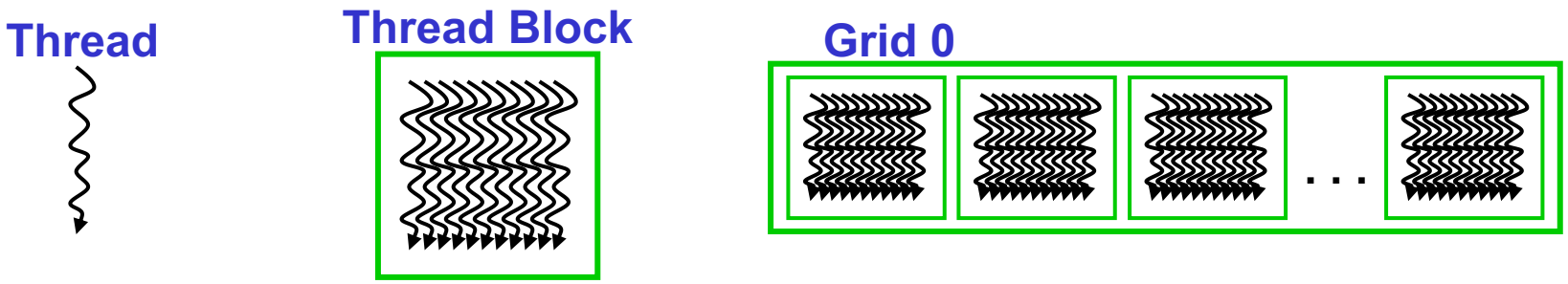
Grids and blocks and threads, oh my...

- Thread: basic unit of execution/parallelism
  - Can be organized in 1D, 2D, 3D layout within a block (for easier indexing)
  - Scheduled in warps (batches of 32 threads)



Grids and blocks and threads, oh my...

- Block: logical organization of a collection of threads
  - Each block could have, e.g., 64, 256, 512, 768, 1024, etc., threads
  - Not all blocks run in parallel, but more than 1 can run on a SM concurrently
  - Each block of threads is assigned to a SM (no control which goes where!)
  - If way more blocks than SMs, blocks are queued and context-switched
  - Each SM can maintain the context for multiple blocks!



Grids and blocks and threads, oh my...

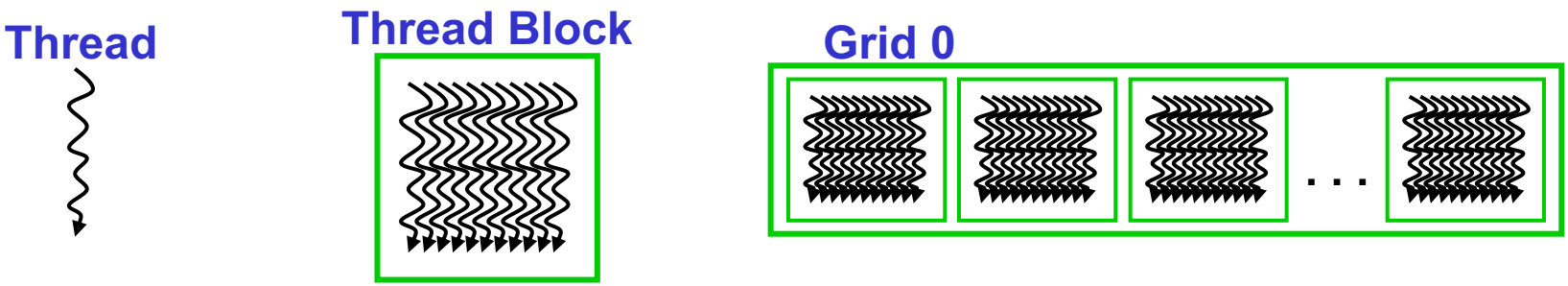
- **Block:** logical organization of a collection of threads

➢ Threads in the same block:

- Share data and synchronize while doing their share of the work
- Communicate via shared memory

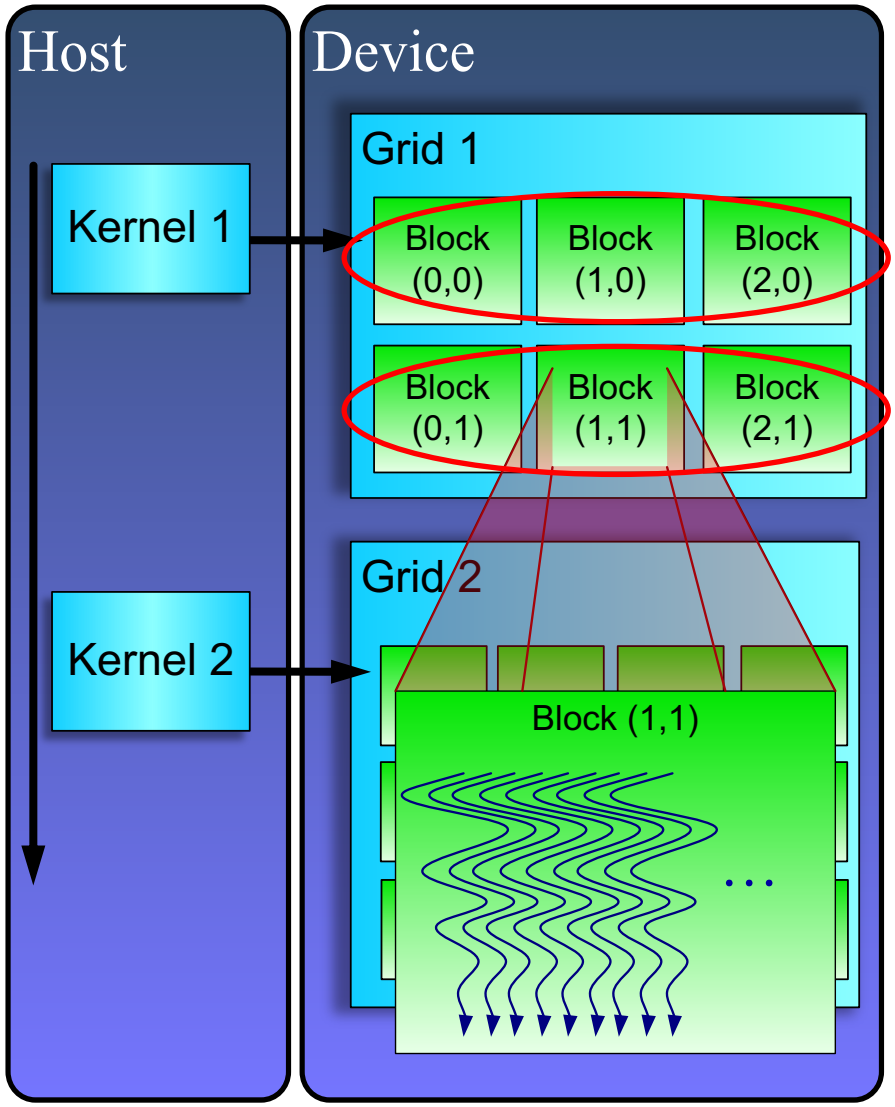
➢ Threads in different blocks cannot cooperate

- Blocks execute in any order!



Grids and blocks and threads, oh my...

- Grid: logical organization of a collection of blocks
  - e.g., 1D, 2D, 3D logical layout of the blocks (easier indexing)



Grids and blocks and threads, oh my...

- Example: increment all elements in an N * N matrix
  - Say we want to assign one element per thread, and N is 1024
- Let's pick block size = 256 threads per 1 block => 1024*1024/256 = 4096 blocks
  - Organize threads in 2D blocks => 16 * 16 threads per block
  - Organize blocks in a 2D grid => 64 * 64 blocks
- Declare them in CUDA C:

```
dim3 threadsPerBlock(16, 16); // 256 threads in total

dim3 blocks(N / threadsPerBlock.x, N / threadsPerBlock.y); // 4096 blocks total
```

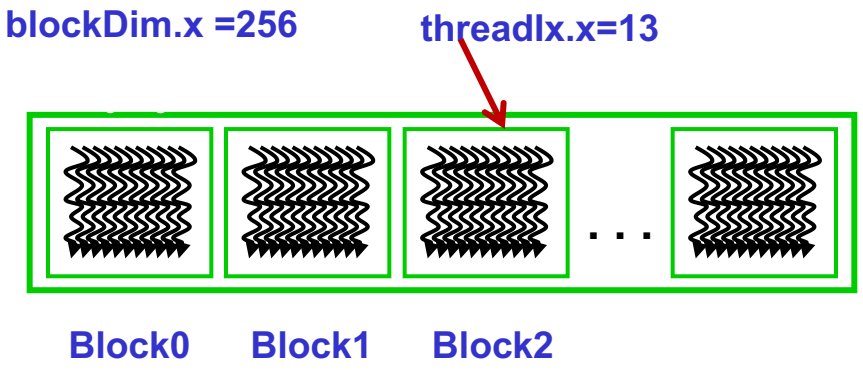
- Launch kernel using the declared dimensions:

```
compute<<<blocks, threadsPerBlock>>>( /* kernel parameters */ );
```

Identifying/indexing a thread

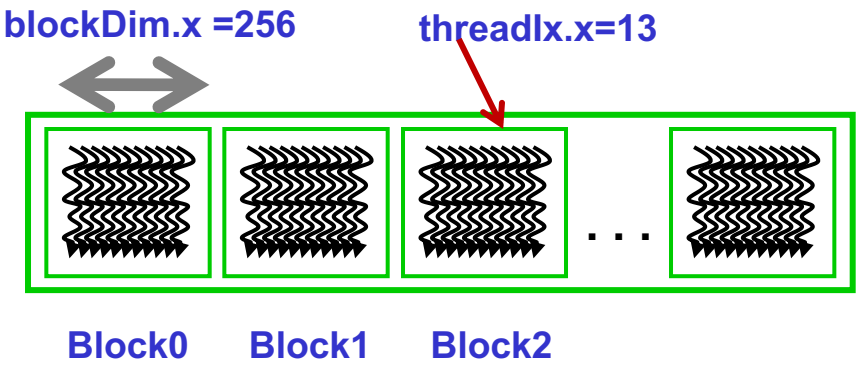
- These are builtin notations which a thread can use to identify its index in the grids and blocks.
  - threadIdx = thread index within its block
  - blockDim = size of a block (how many threads in each dimension)
  - blockIdx = block index in the grid
  - gridDim = size of a grid (how many blocks in each dimension)

This figure shows an example where each block has one dimension with 256 thread per block. The grid is also 1D.



Identifying/indexing a thread

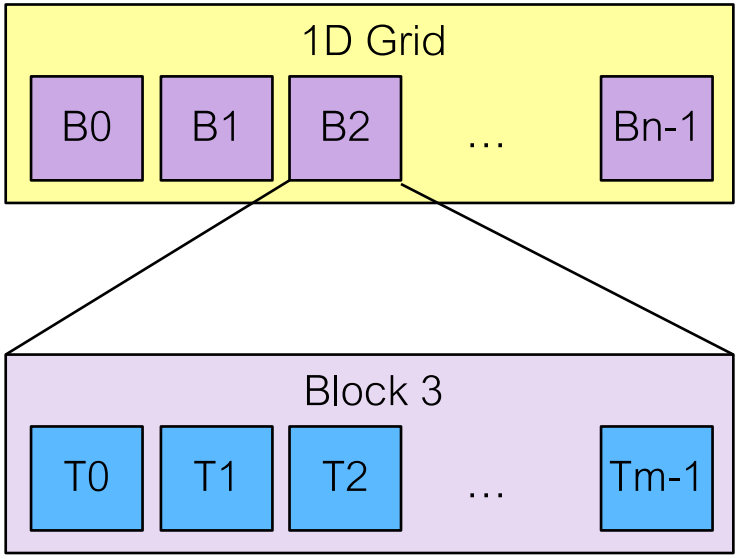
- Each thread can identify its assigned element, as follows:
  - int i = (blockIdx.x * blockDim.x) + threadIdx.x;
  - int j = (blockIdx.y * blockDim.y) + threadIdx.y;



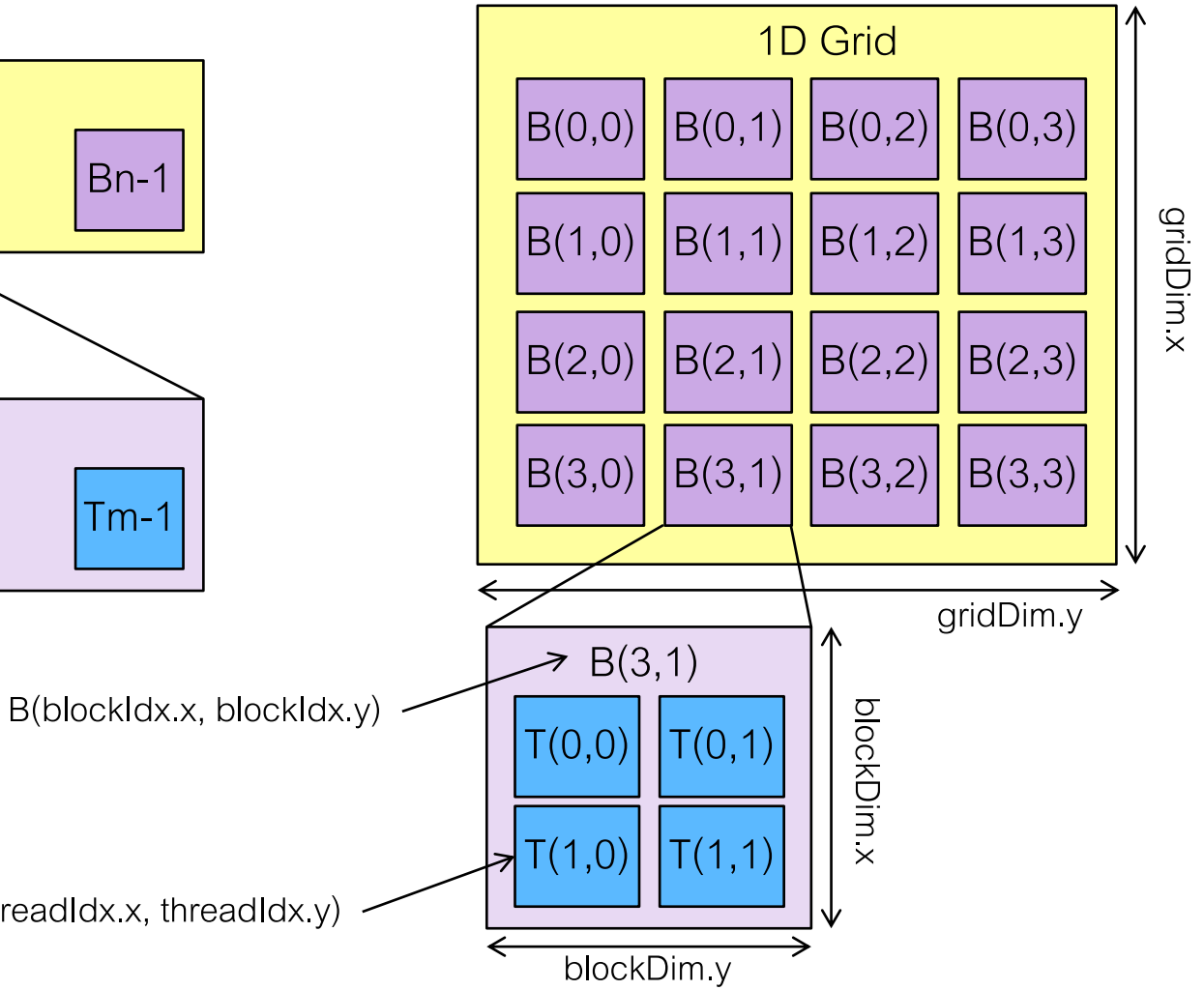
i =2*256+13= 525 is the ID of the thirteenth thread in block 2 in the entire grid!

Examples

- 1D Grid of blocks, each one is a 1D block of threads



- 2D Grid of blocks, each one is a 2D block of threads



Examples

Add the constant b to every element of the array a (a has K elements)

CPU-Only

```
Void Increment_cpu (float *a, float b, int K)
{

      for (int idx = 0; idx<K; idx++)
        a [idx] = a [idx] + b;
}



Void main()
{
…
    Increment_cpu (a, b, K);
}
```

GPU

```
__global__ Void Increment_gpu (float *a, float b, int K)
{
   int idx = blockIdx.x*blockDim.x + threadIdx.x;
   if (idx<K)
         a [idx] = a [idx] + b;
}



Void main()
{
…
dim3 blockDim (blocksize);
dim3 gridDim (K / (float)blocksize) );
Increment_gpu <<< gridDim, blockDim>>>(da, b, K);
}
```

Examples

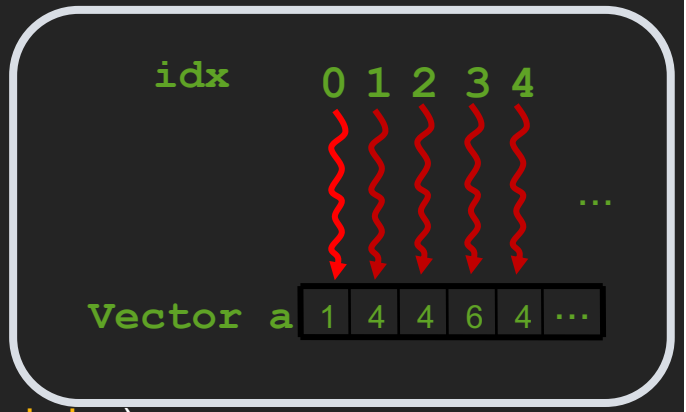
Add the constant b to every element of the array a (a has K elements)

CPU-Only

```
Void Increment_cpu (float *a, float b, int K)
{

      for (int idx = 0; idx<K; idx++)
        a [idx] = a [idx] + b;
}



Void main()
{
…
    Increment_cpu (a, b, K);
}
```

GPU

```
__global__ Void Increment_gpu (float *a, float b, int K)
{
   int idx = blockIdx.x*blockDim.x + threadIdx.x;
   if (idx<K)
        a [idx] = a [idx] + b;
}



Void main()
{
…
dim3 blockDim (blocksize);
dim3 gridDim (K / (float)blocksize) );
Increment_gpu <<< gridDim, blockDim>>>(da, b, K);
}
```



Examples

Add the constant b to every element of the array a (a has K elements)

CPU-Only

```
Void Increment_cpu (float *a, float b, int K)
{

      for (int idx = 0; idx<K; idx++)
        a [idx] = a [idx] + b;
}
```

Executed on Host (CPU)

```
Void main()
{
…
    Increment_cpu (a, b, K);
}
```

Called from Host

GPU

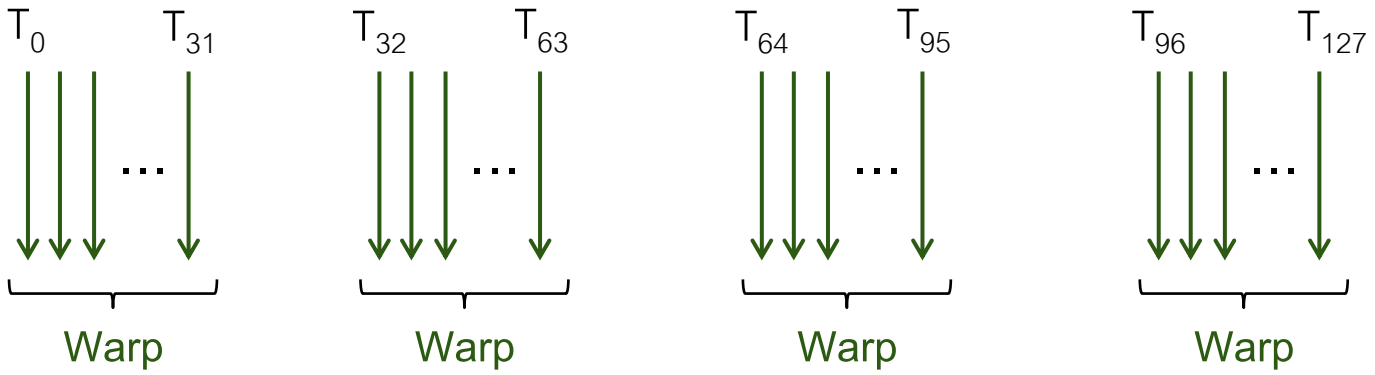
```
__global__ Void Increment_gpu (float *a, float b, int K)
{
   int idx = blockIdx.x*blockDim.x + threadIdx.x;
   if (idx<K)
        a [idx] = a [idx] + b;
}
```

Executed on Device (GPU)

```
Void main()
{
…
dim3 blockDim (blocksize);
dim3 gridDim (K / (float)blocksize) );
Increment_gpu <<< gridDim, blockDim>>>(da, b, K);
}
```

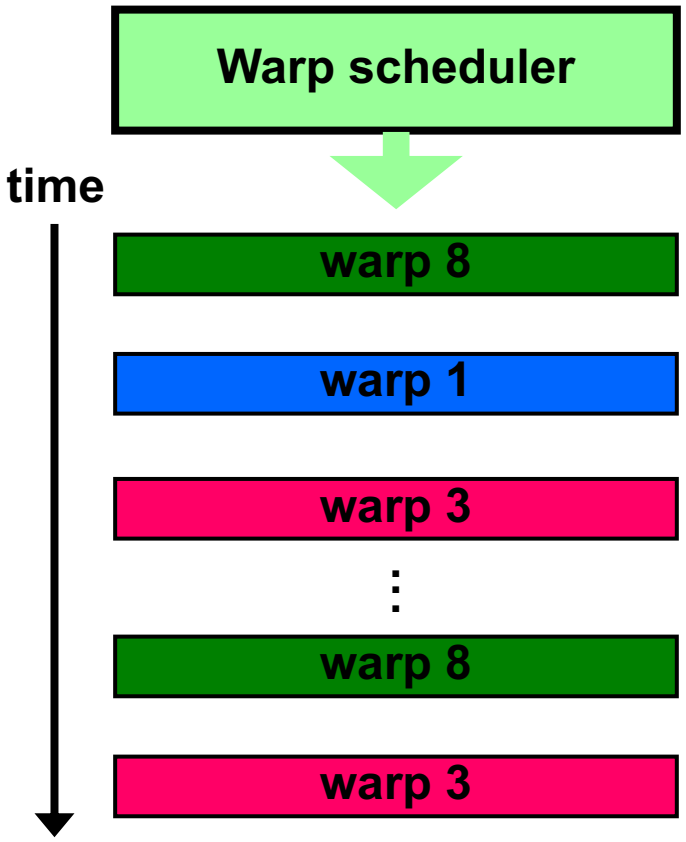
Parallel execution

- A single instruction is issued for a warp at a time (warp = 32 threads)
- Threads in a warp execute instructions in lock-step (same instruction for all)
- Warps can run ahead of other warps – use __syncthreads() to barrier all thread warps in a block (not all threads from all blocks!!)

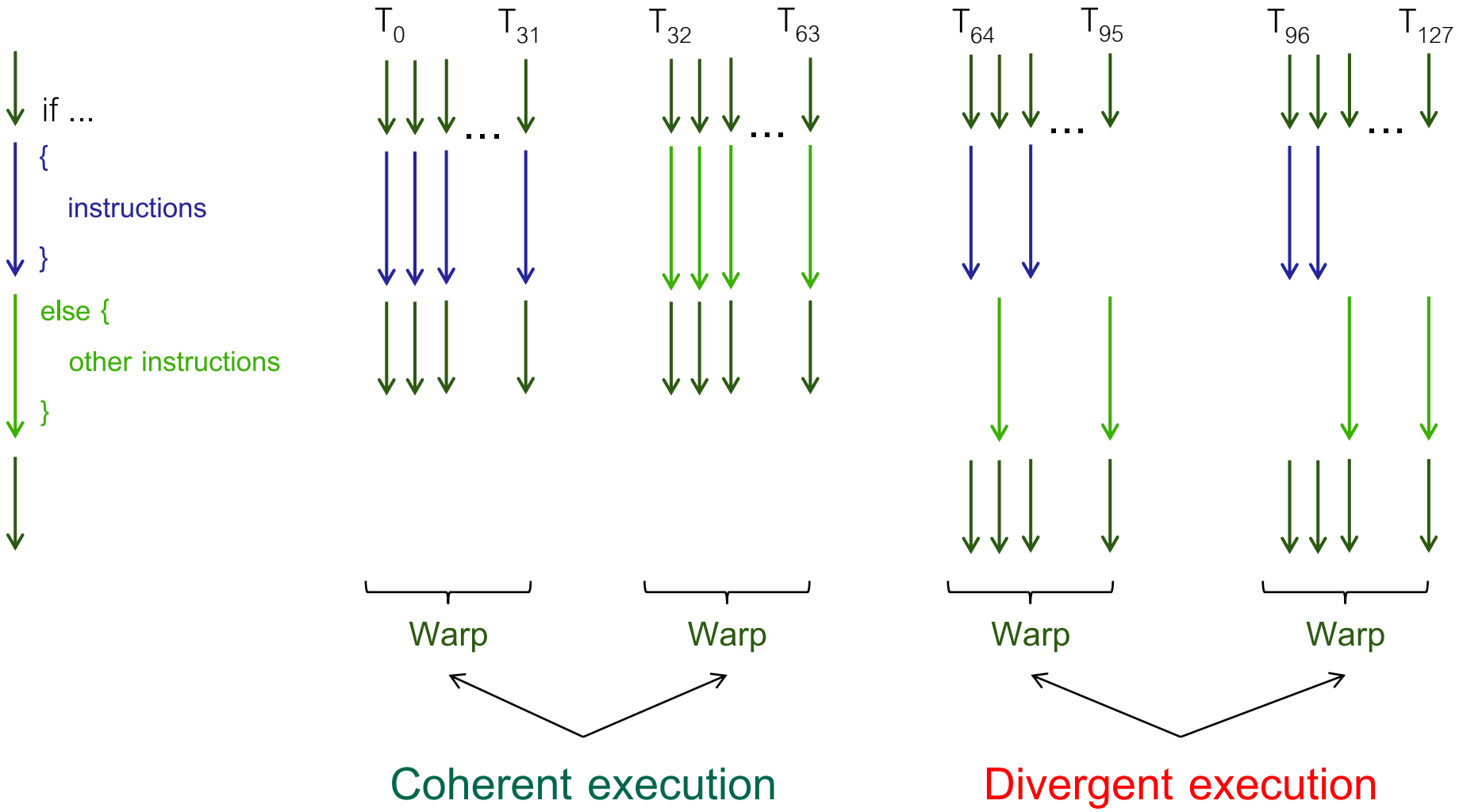


Warp Scheduler

- All threads in a warp execute the same instruction
- Need to have enough warps to hide memory access latency
- Thread divergence : When threads in a warp go through different execution paths (next slide)



Control Flow: Thread Divergence



Limitations

- Maximum number of threads per block
- Maximum number of blocks per SM
- Maximum number of threads per block
- Maximum number of blocks per grid
- These numbers depend on the GPU
- These are hardware limitations => if exceeded, kernel launch failure!

  => For huge data, cannot count on one item per thread

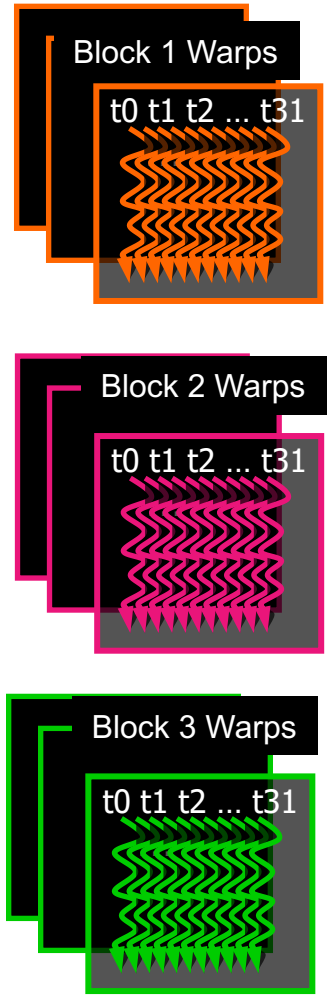
- Not always best to run with the max for each of these
  - Sometimes less is more
  - Understand your device and pick the parallelism parameters for the job

	Compute Capability											
Technical Specifications	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.5**
Maximum dimensionality of thread block	3											
Maximum x- or y-dimension of a block	1024											
Maximum z-dimension of a block	64											
Maximum number of threads per block	1024											
Warp size	32											
Maximum number of resident blocks per multiprocessor	16				32							16
Maximum number of resident warps per multiprocessor	64											32
Maximum number of resident threads per multiprocessor	2048											1024
Number of 32-bit registers per multiprocessor	64 K			128 K	64 K							
Maximum number of 32-bit registers per thread block	64 K	32 K	64 K				32 K	64 K		32 K	64 K	
Maximum number of 32-bit registers per thread	63	255										
Maximum amount of shared memory per multiprocessor	48 KB			112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB	64 KB

Limitations and Thread scheduling

- If the maximum number of threads per SM is 2048, maximum number of blocks per SM is 32, and maximum registers per SM is 64K.

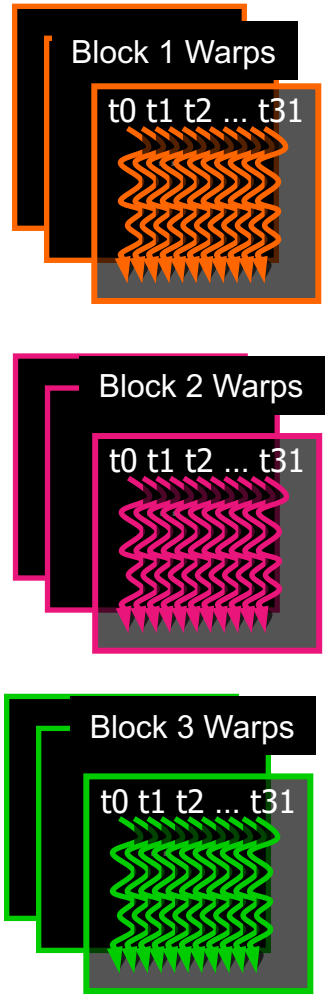
- With 256 threads per block. each SM can only run 8 thread blocks. Why?



Limitations and Thread scheduling

- If the maximum number of threads per SM is 2048, maximum number of blocks per SM is 32, and maximum registers per SM is 64K.

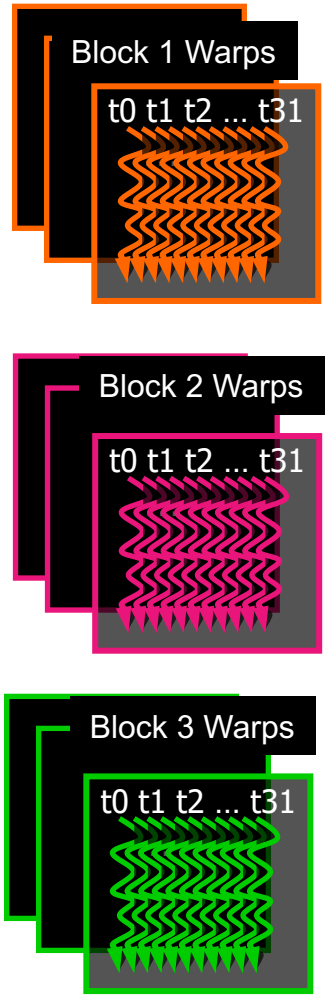
- With 256 threads per block, each SM can only run 8 thread blocks. Why? Because of the 2048 max threads per SM limit.



Limitations and Thread scheduling

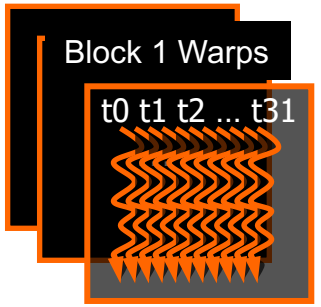
- If the maximum number of threads per SM is 2048, maximum number of blocks per SM is 32, and maximum registers per SM is 64K.

- If we assign 256 threads per block and in our code each thread uses 48 registers, how many thread blocks does an SM run?

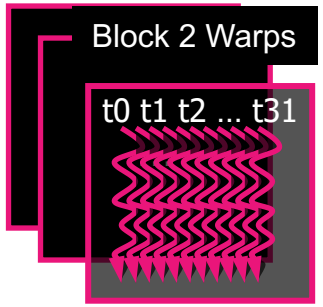


Limitations and Thread scheduling

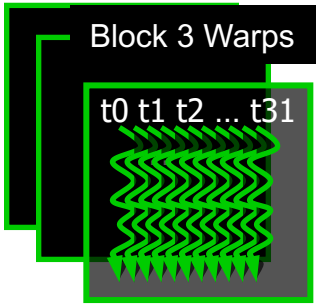
- If the maximum number of threads per SM is 2048, maximum number of blocks per SM is 32, and maximum registers per SM is 64K.



- If we assign 256 threads per block and in our code each thread uses 48 registers, how many thread blocks does an SM run?



- It depends but most probably around 5! The compiler will reduce the number of active threads per SM to not “register spill”, more later!



Memory and access techniques

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

Memory types

- Global memory
- Local memory — local memory that does not fit into register
- Shared memory — L1 cache
- Constant memory — cached in L1 cache
- Texture memory

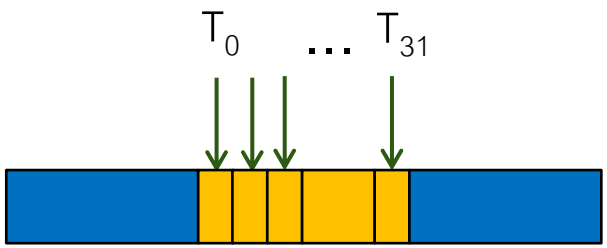
Local and Global memory

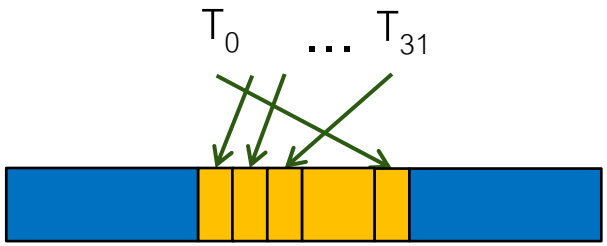
- Global memory similar to DRAM
  - Most data resides here
  - Host communication (this is where data gets transferred from/to CPU memory)
  - Shared by all threads
  - Large size (a few GB typically), but slower than shared memory
  - L1 cache helps hide the latency for global (and local) memory accesses
  - Good bandwidth via memory coalescing
- Local memory (keep in mind: terms used in CUDA)
  - aka Private per thread global memory
  - Auto variables, register spill
  - Same speed as global memory but accesses are coalesced!

Memory coalescing

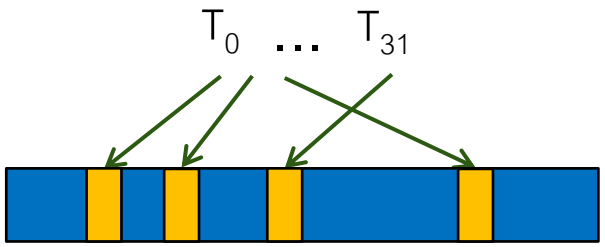
- Warp accesses should reference sequential memory locations for best performance => these accesses get coalesced into a single access

Coalesced accesses



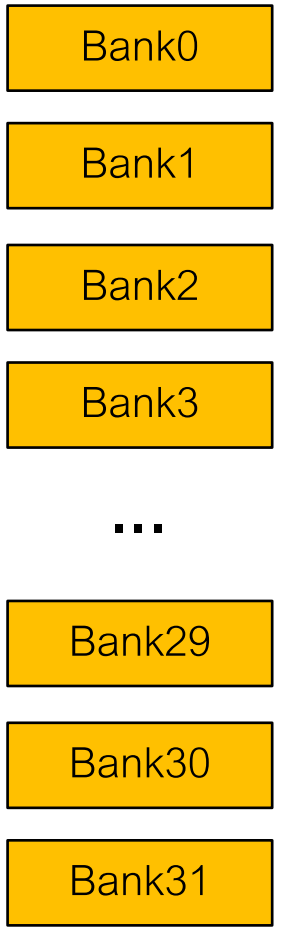


Scattered accesses



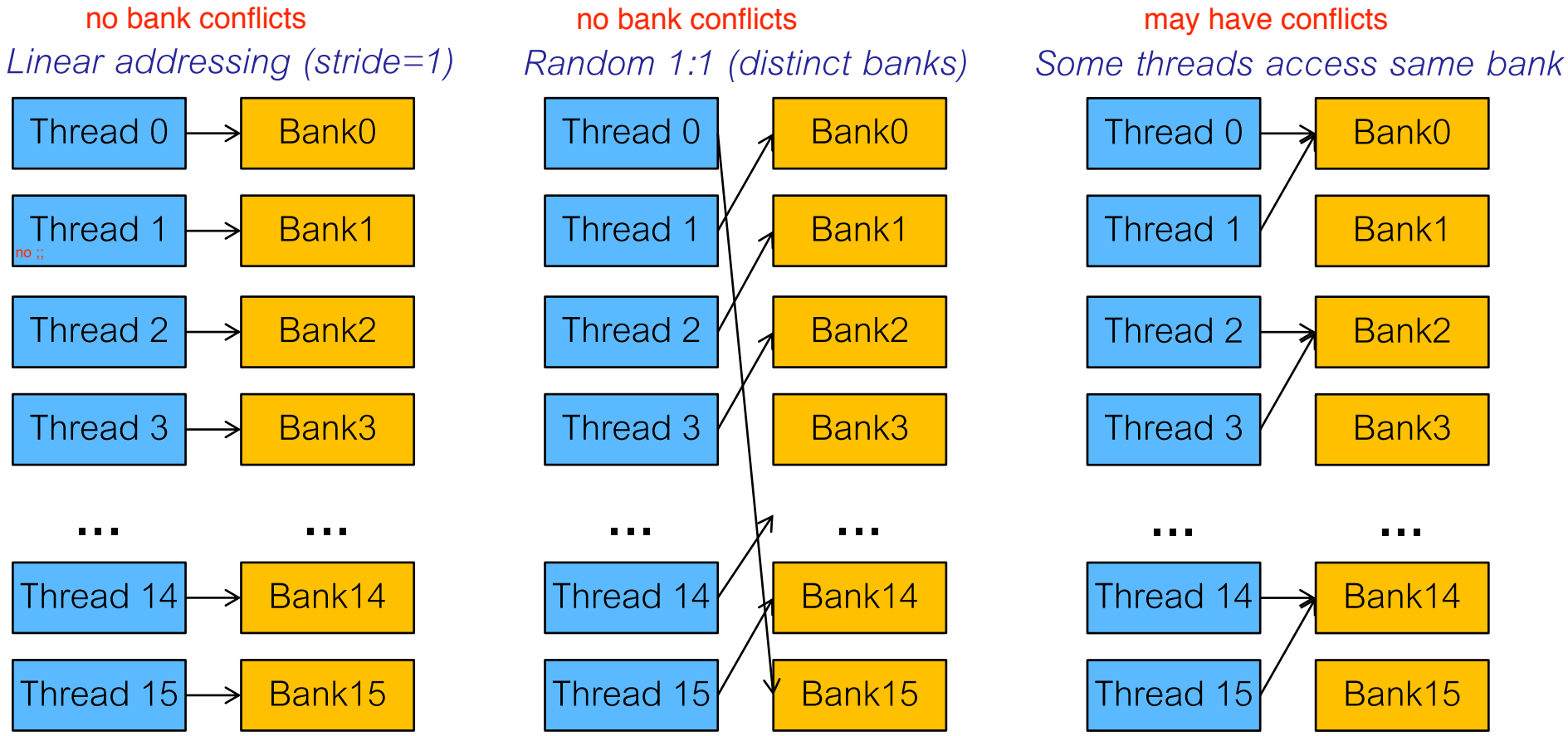
Shared memory

- 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 32 … same as number of warps
  - 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
    different threads access to same bank is serialized (not parallelised)
- 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!