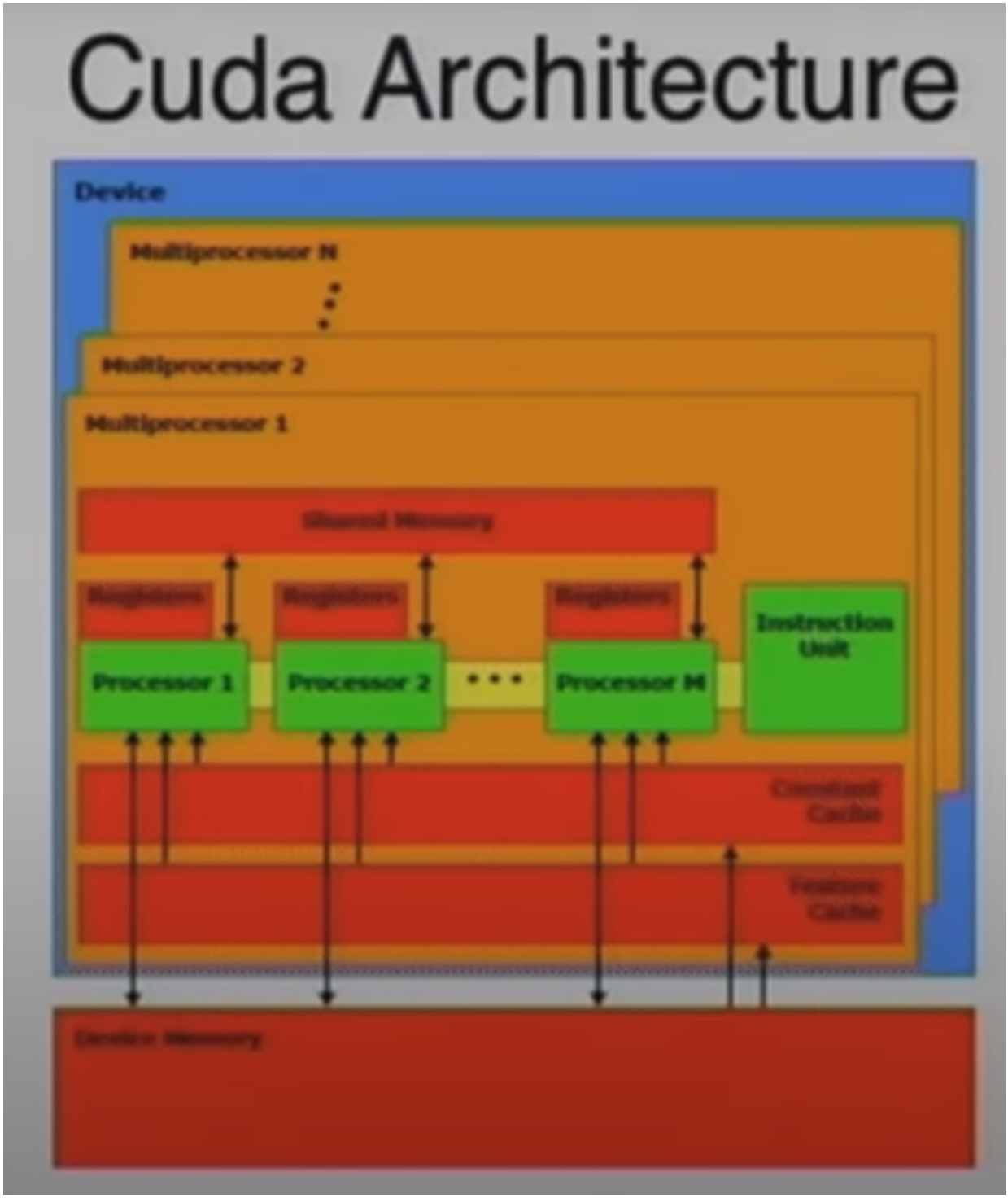
Unit 3

CUDA

GPU has a lot more computation but a LOT LESS CACHE.

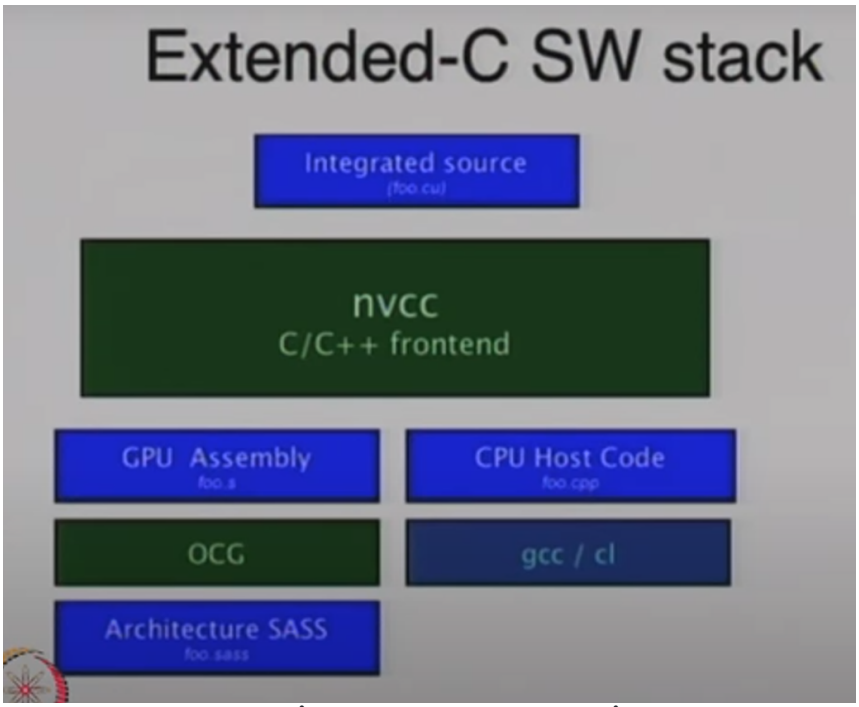


This is the best quality I found 😭.

CUDA is Compute Unified Device Architecture. User does batch execution.

⟸ this is how we launch a batch. The GPU had its own device memory.

nvcc, also called cuda compiler, hijacks cuda specific code and converts/compiles it. It seperates device code and host code.



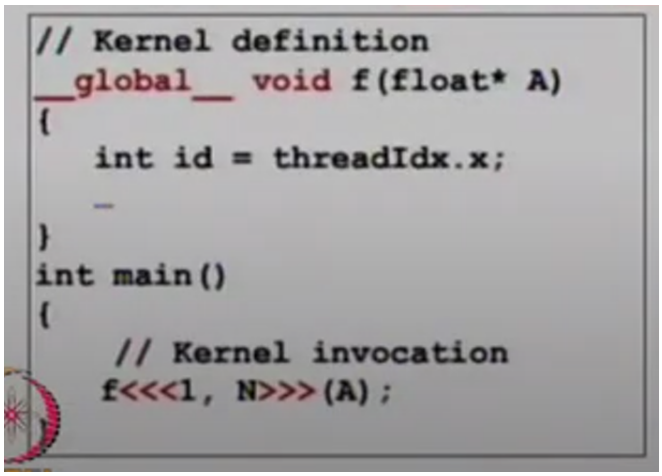
CUDA library ⟹ CUDA Runtime ⟹ CUDA Driver ⟹ Device. The application can speak to all 3 of these CUDA parts.

CUDA is SPMD Single program multiple data. A CUDA kernel is the code that is being run. A block is a group of threads running the same kernel. A group of blocks is a grid.

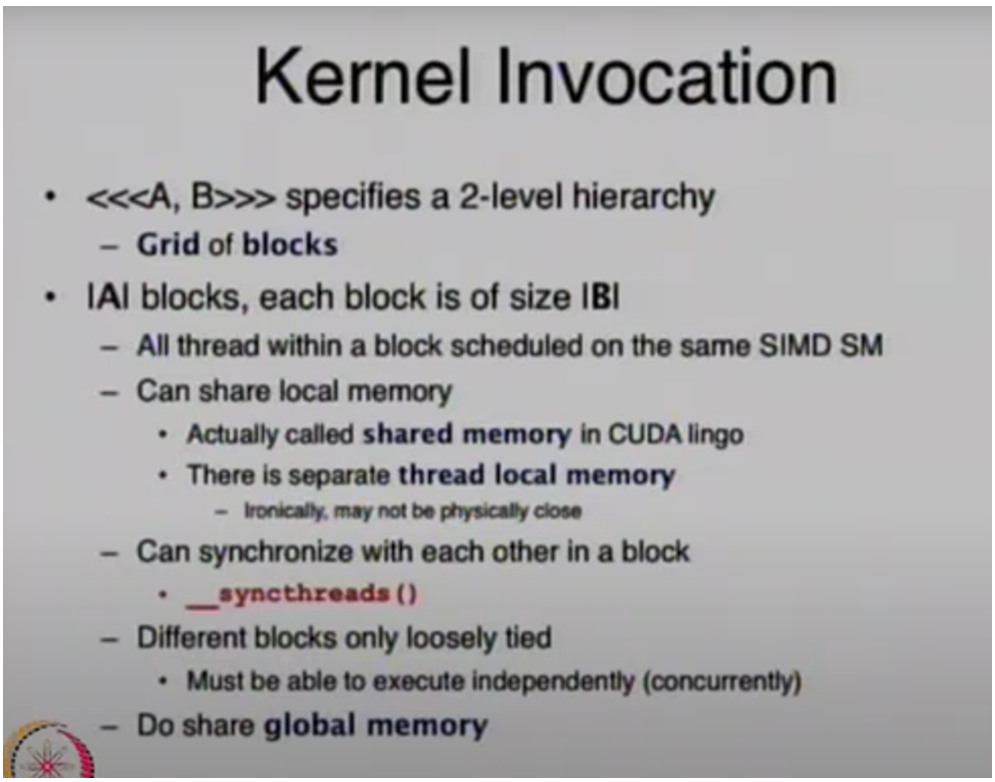
Threads in a block communicate with shared memory, global memory, atomic operations and barriers. Threads in different blocks communicate ONLY with global memory.

I don't know what an "SM" is but we fit blocks into it. Probably ~~shared memory~~? It is Streaming Multiprocessor. Mini CPU core for CUDA/GPU. One optimization implemented is when an SM is not used fully we give the rest to another block.

Basic cuda code:



```
// Kernel definition
__global__ void f(float* A)
{
    int id = threadIdx.x;
    _
}
int main()
{
    // Kernel invocation
    f<<<1, N>>>(A);
```



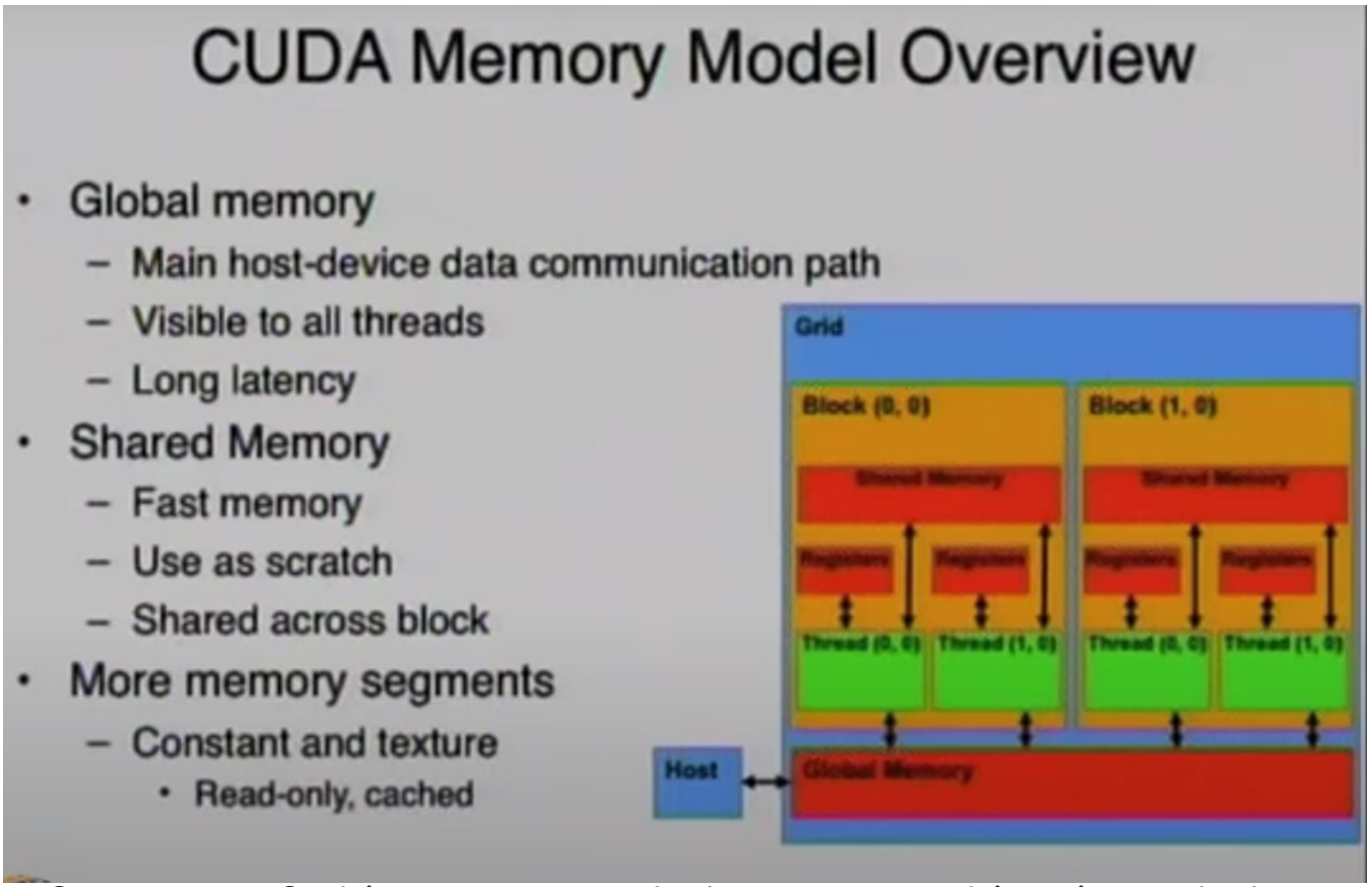
Kernel Invocation

- <<<A, B>>> specifies a 2-level hierarchy
  - Grid of blocks
- |A| blocks, each block is of size |B|
  - All thread within a block scheduled on the same SIMD SM
  - Can share local memory
    - Actually called **shared memory** in CUDA lingo
    - There is separate **thread local memory**
      - Ironically, may not be physically close
  - Can synchronize with each other in a block
    - `__syncthreads()`
  - Different blocks only loosely tied
    - Must be able to execute independently (concurrently)
  - Do share **global memory**

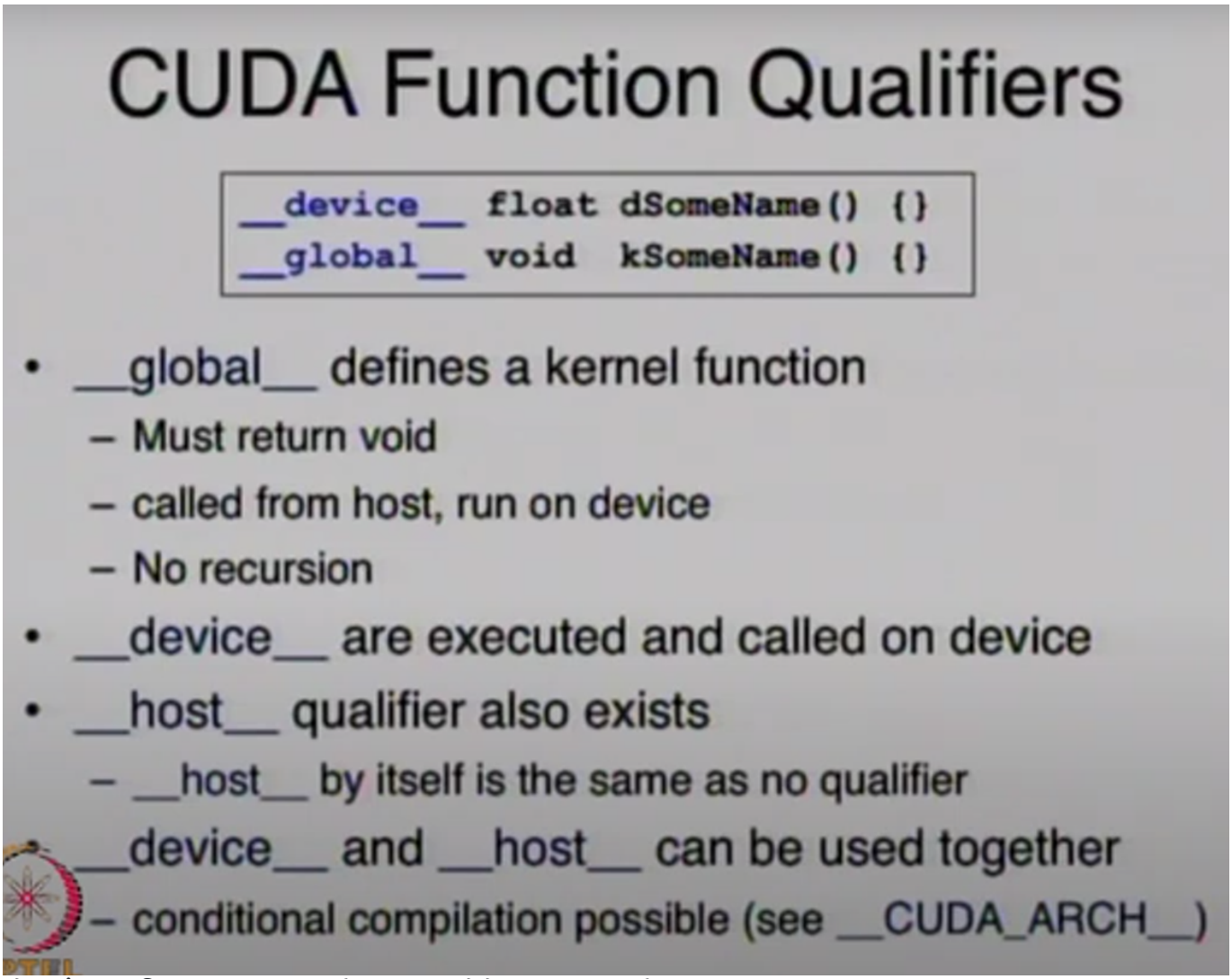
32 groups of parallel threads are executed. These are called "wraps". All threads need not executed identically. Some threads could potentially branch out. But eventually all of them sync up.

In kernel invocation, A can be a 2D vector. B can be a 3D vector. A will be referencing grid based.

CUDA memory:

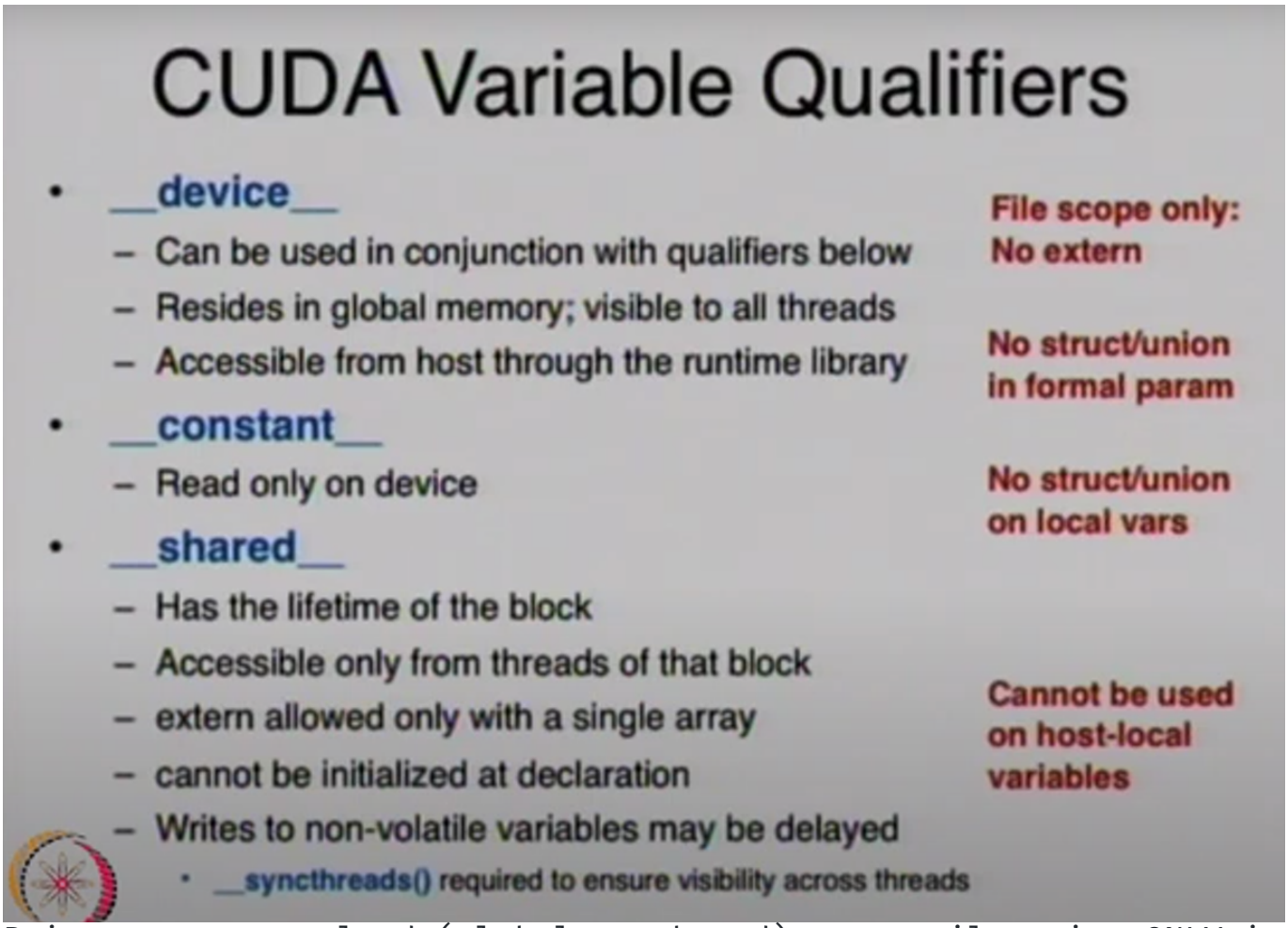


Before most of this was not cached, now everything is cached. `cudaMalloc((void**)&Md,size)` allocates in the global memory. We can also get "page-locked" memory, i.e. host memory. Shared memory is saved into "banks" $\Longleftarrow$ not explained. Texture memory is not discussed. It is different from the rest. It is more focused on graphics applications.



device func can;t have address on host.

Functions executed on the device cannot have static variables or dynamic number or arguments. OLD ass versions also does not support recursion.



Pointers are resolved (global or shared) at compiler time ONLY in 1.x versions.

Matrix multiplication example :)

```
__global__ void MatrixMulKernel(float* Md, float* Nd,
        float* Pd, int Width)
{
    // Pvalue stores the matrix element computed by the thread
    float Pvalue = 0;

    for (int k = 0; k < Width; ++k) {
        float Melement = Md[threadIdx.y*Width+k];
        float Nelement = Nd[k*Width+threadIdx.x];
        Pvalue += Melement * Nelement;
    }

    Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```

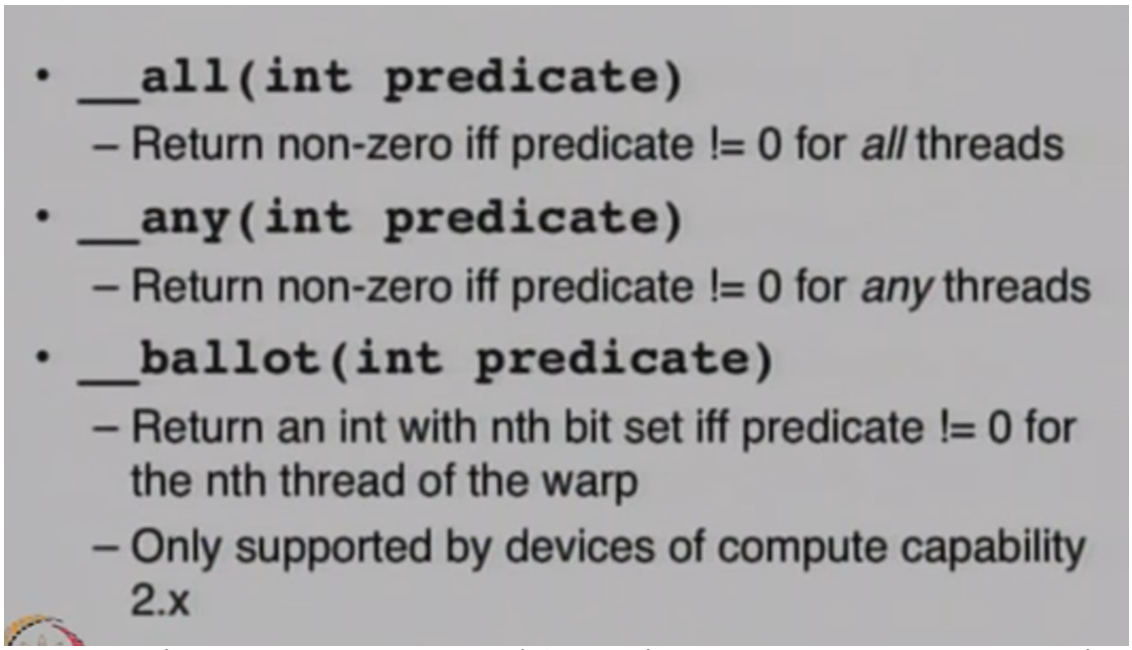
```
// Setup the execution configuration
dim3 dimGrid(1, 1);
dim3 dimBlock(Width, Width);


// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

Ways to sync threads in a block:

- **__syncthreads()**
  - block barrier AND
  - ensure all global/shared memory accesses by all threads are visible in the block
- **__syncthreads(int predicate)**
  - returns the count of threads where predicate != 0
- **__syncthreads_and(int predicate)**
  - returns non-zero *iff* predicate != 0 for *all* threads
- **__syncthreads_or(int predicate)**
  - returns non-zero *iff* predicate != 0 for *any* threads

Last 3 are 2.x and above ONLY. We can also do sync in a wrap.

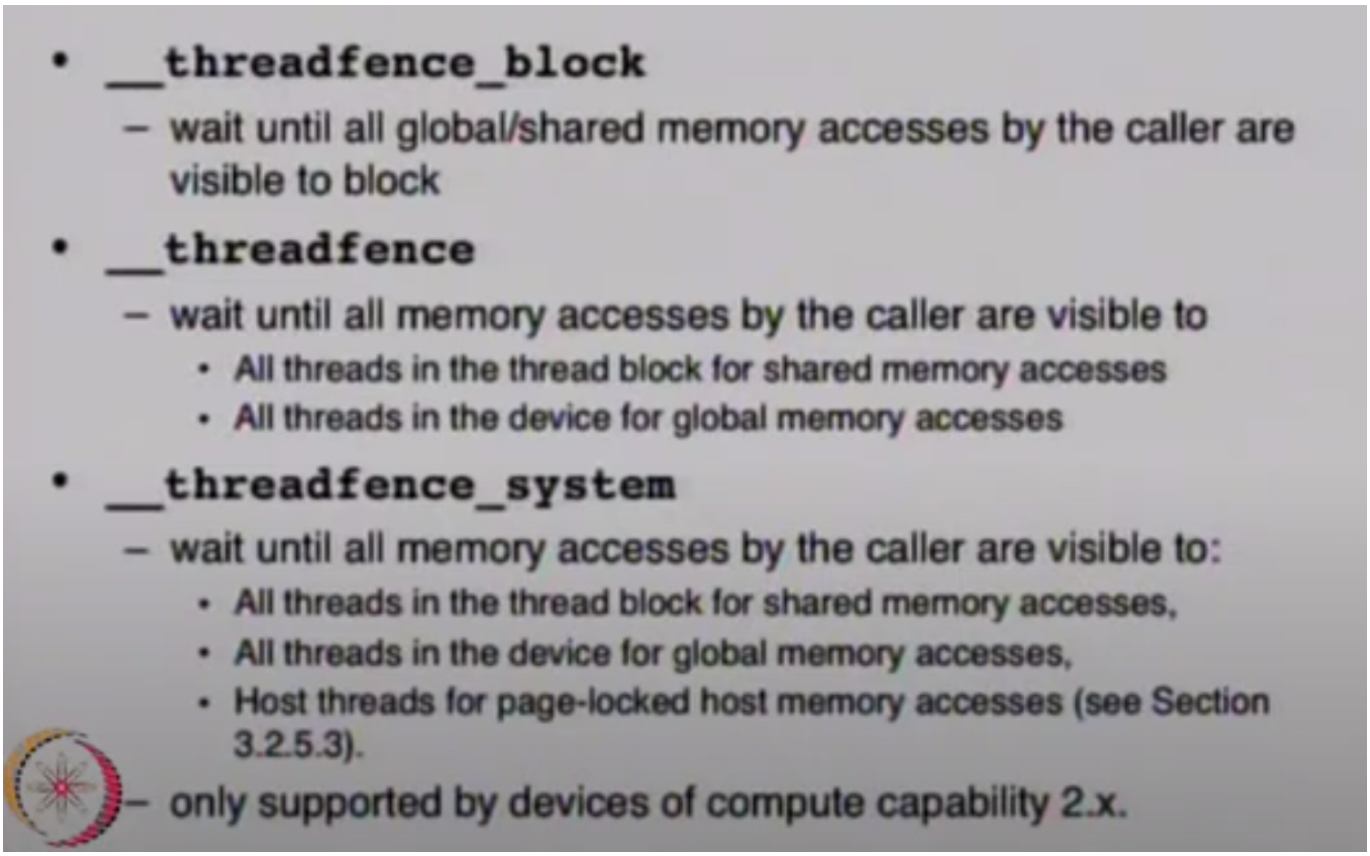


Atomic operations are read-modify-write on one 32 or 64 bit word in global or shared memory. If target is page locked memory, the operation is not atomic from the host's perspective. CAS is Compared and Swap/Set.

nvcc has a device emulation mode. Not very good but decent enough for debugging and general use.

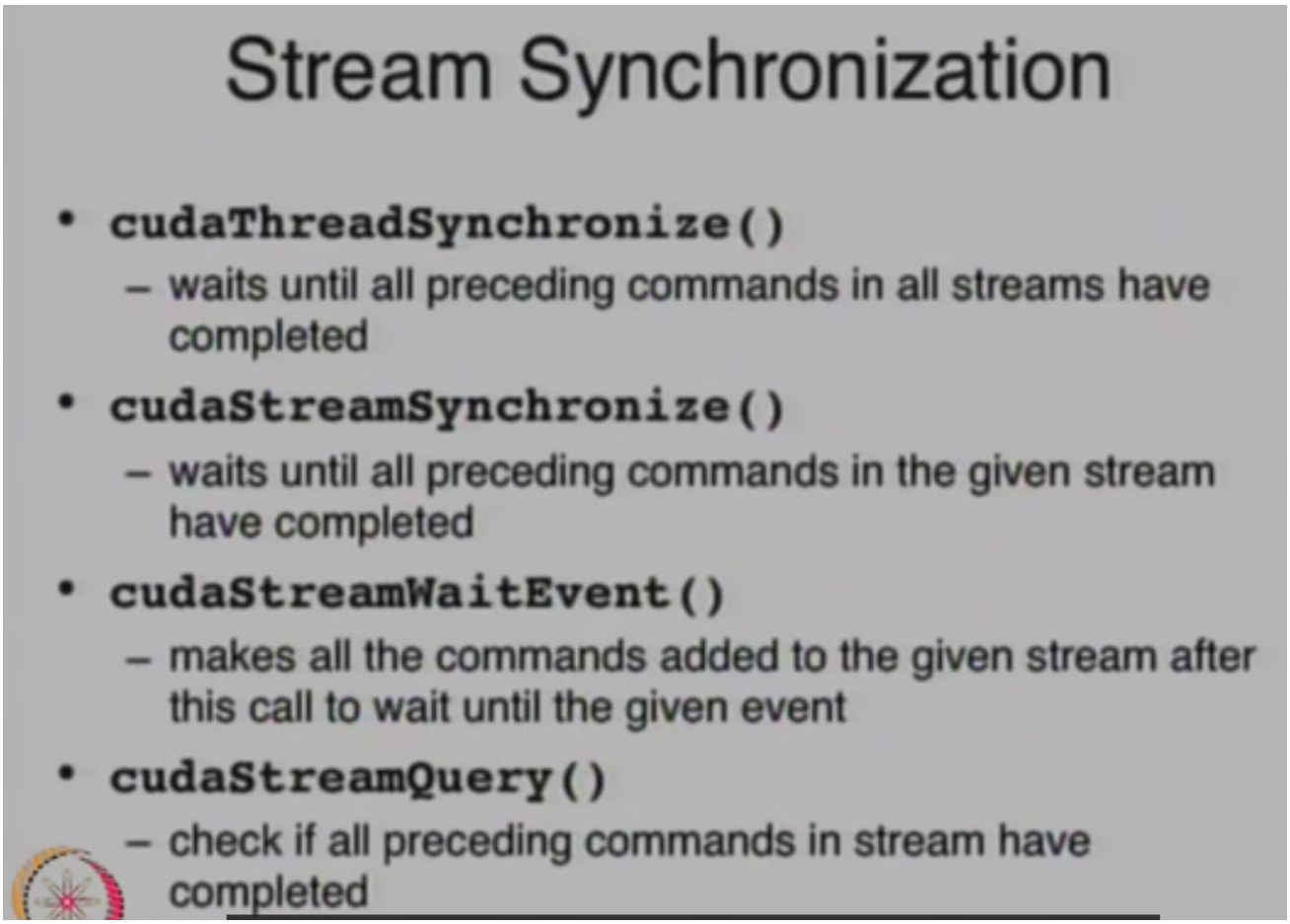
There are a few code examples shown. It confused me and the prof so im not putting it.

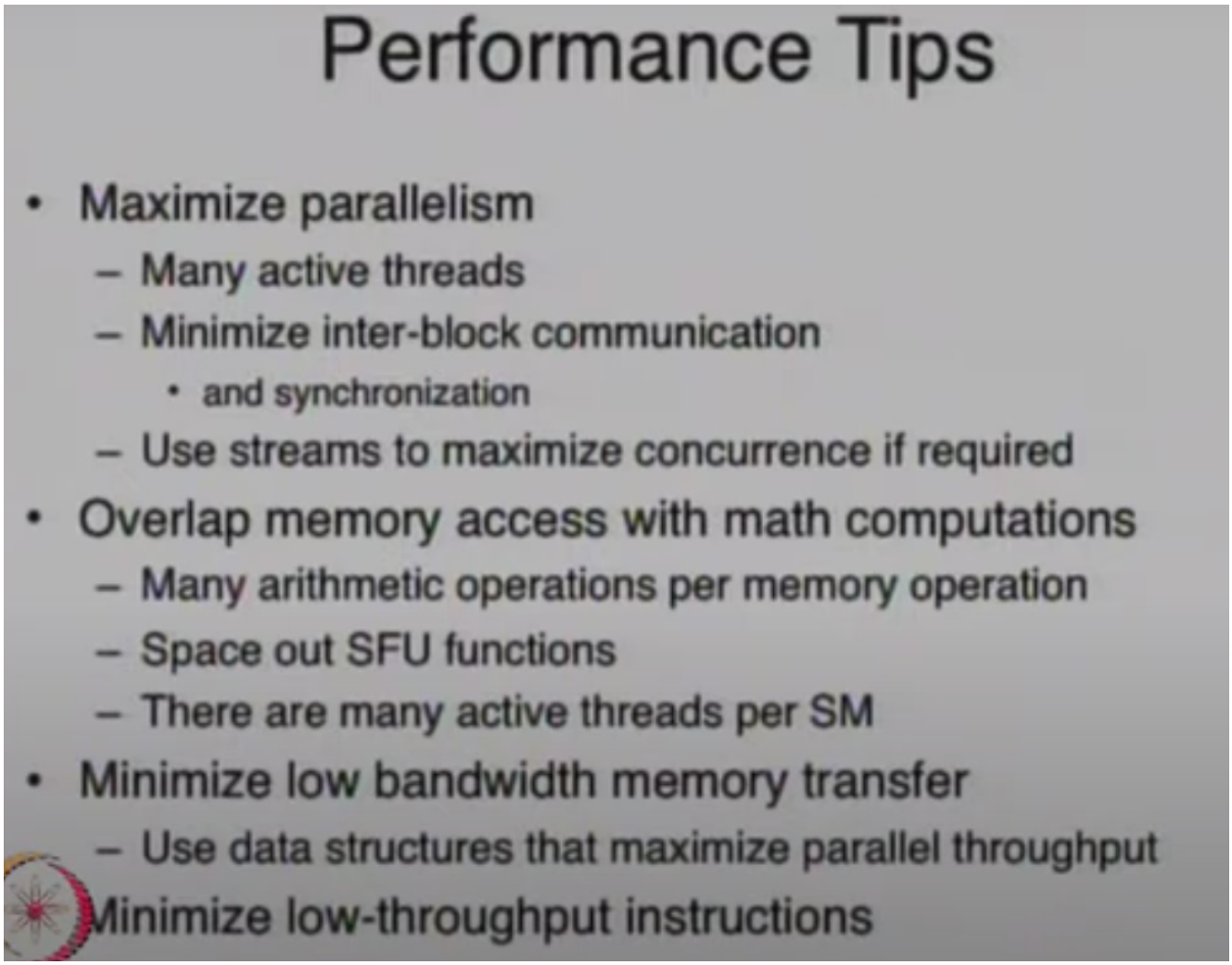
Memory fences:

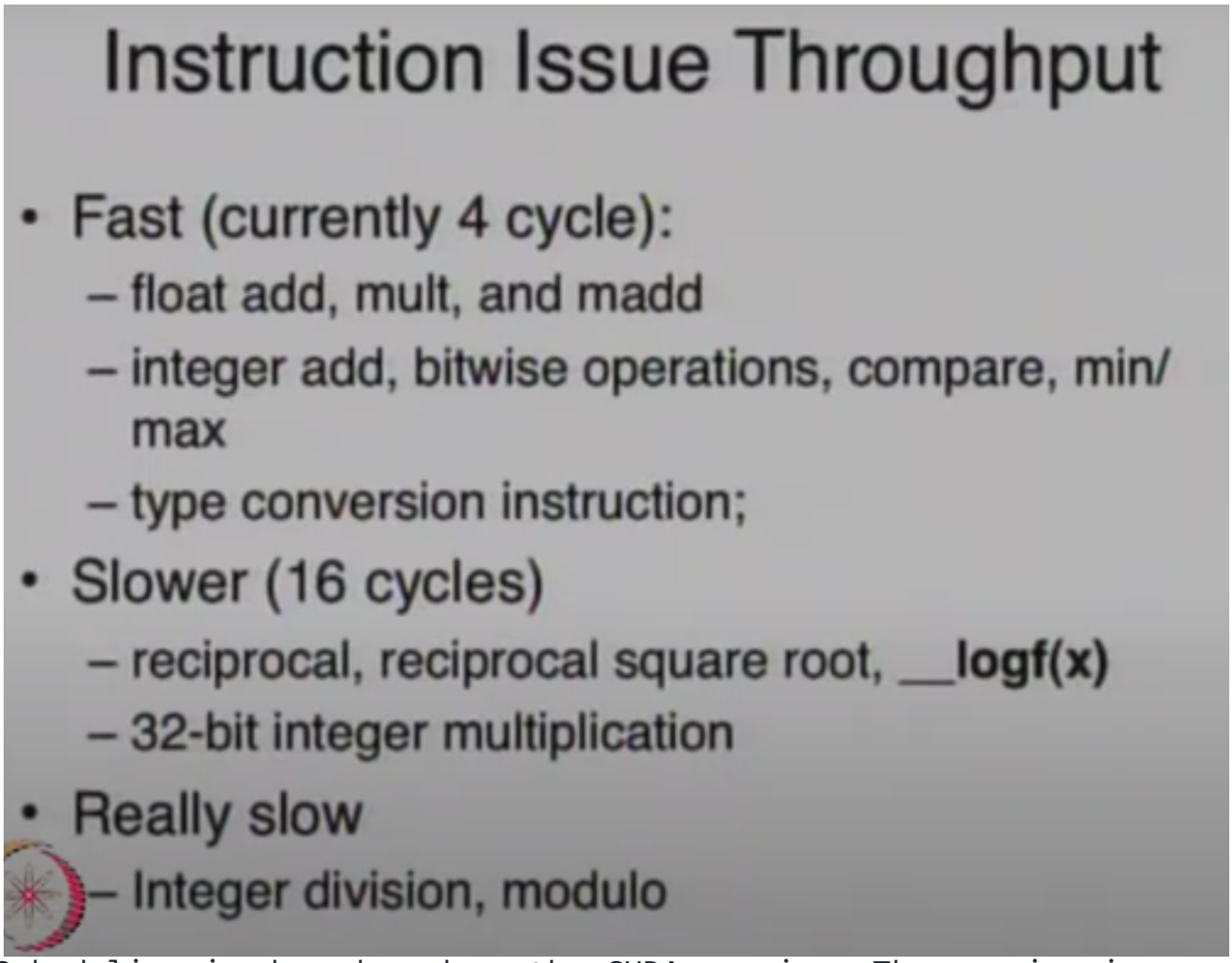


Streams

This is a "high" level thread. This is generally used to ensure concurrency. Multiple streams run at the same time, HOWEVER, stream 0 runs alone. Streams appear to be sequential to the system. If we do not specify a stream it is sent to stream 0

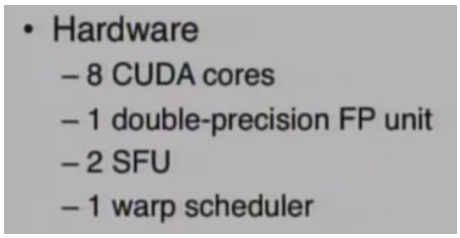

Stream Synchronization
• cudaThreadSynchronize()
– waits until all preceding commands in all streams have completed
• cudaStreamSynchronize()
– waits until all preceding commands in the given stream have completed
• cudaStreamWaitEvent()
– makes all the commands added to the given stream after this call to wait until the given event
• cudaStreamQuery()
– check if all preceding commands in stream have completed

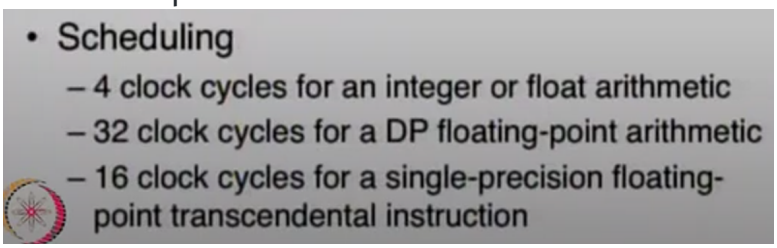

Performance Tips
Maximize parallelism
Many active threads
Minimize inter-block communication
and synchronization
Use streams to maximize concurrence if required
Overlap memory access with math computations
Many arithmetic operations per memory operation
Space out SFU functions
There are many active threads per SM
Minimize low bandwidth memory transfer
Use data structures that maximize parallel throughput
Minimize low-throughput instructions



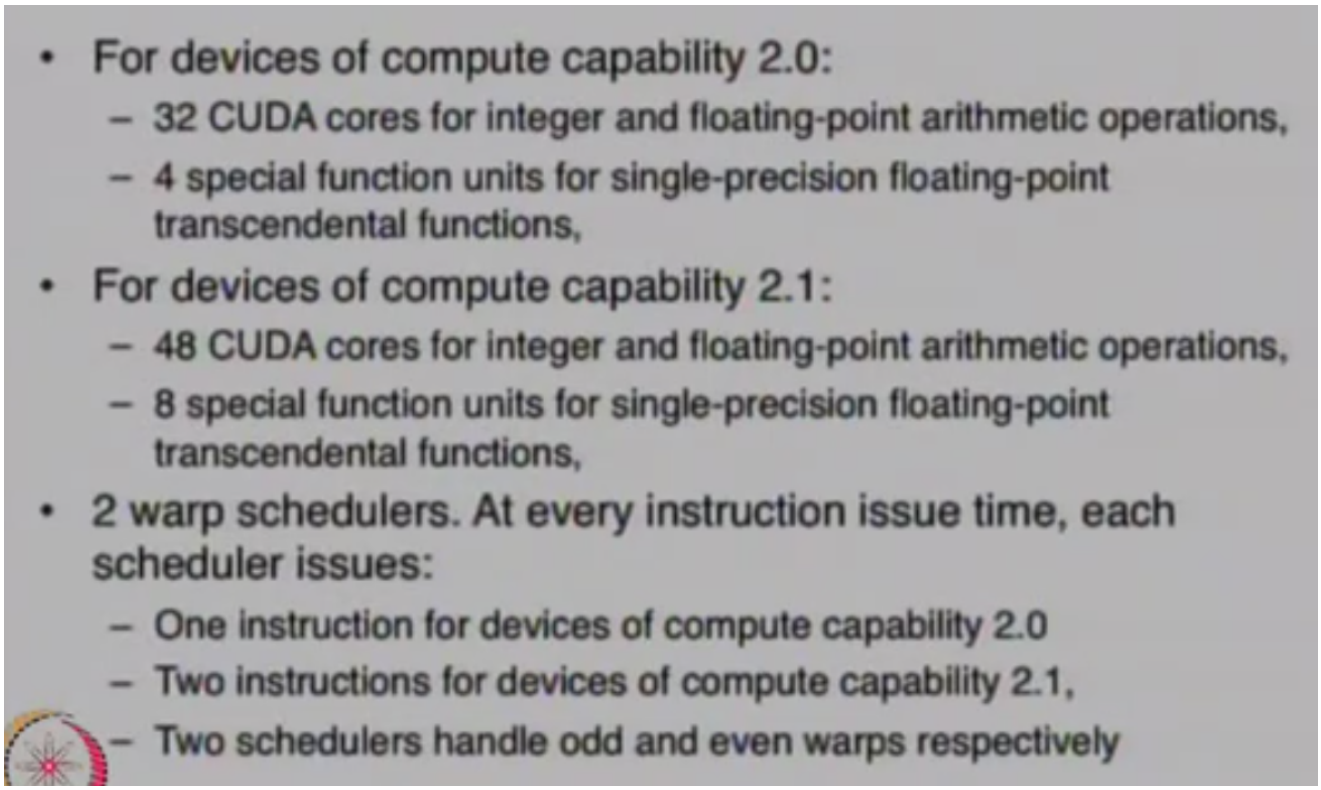
Scheduling is done based on the CUDA version. The version is often hardware dependent.

for example for 1.x



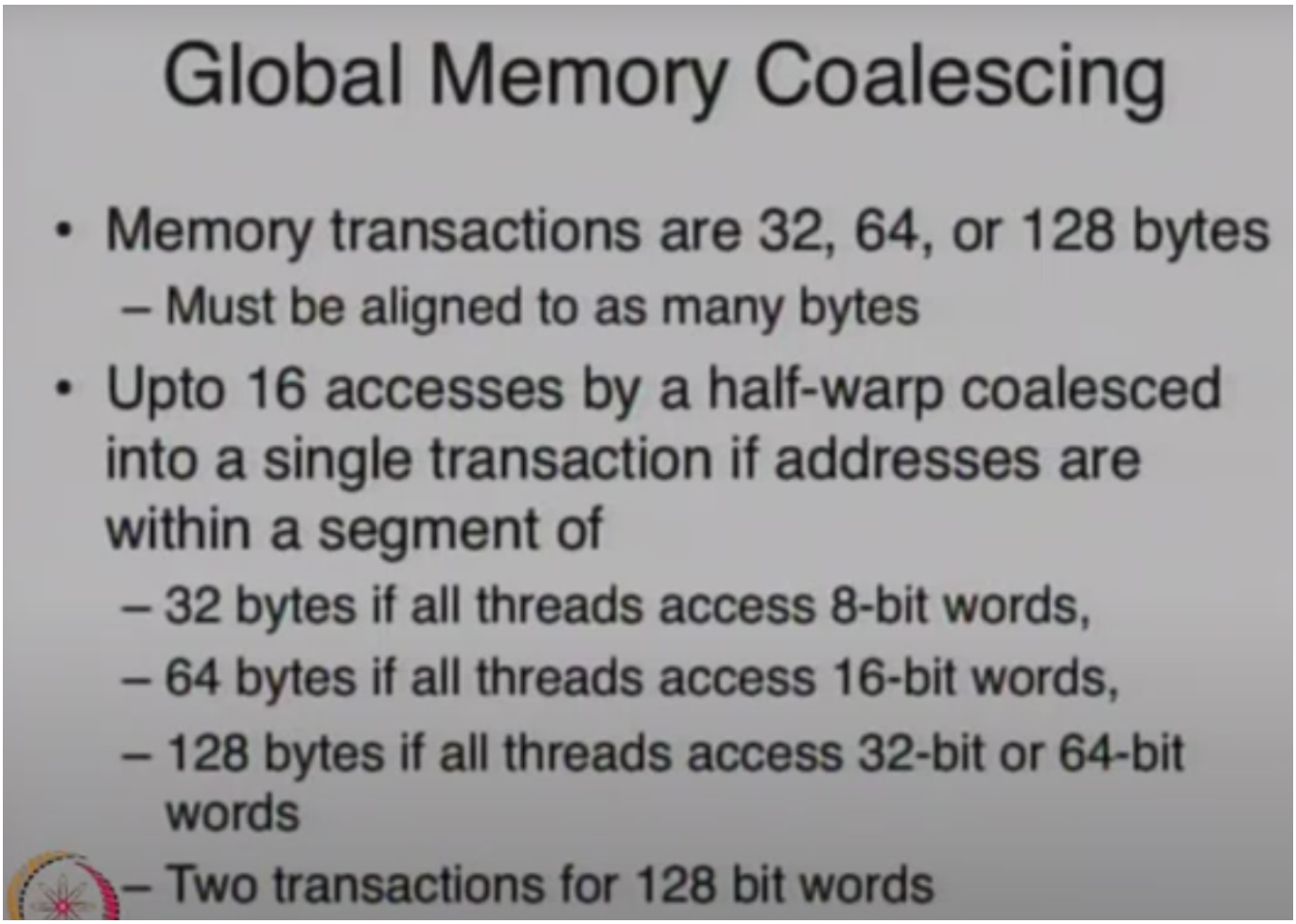


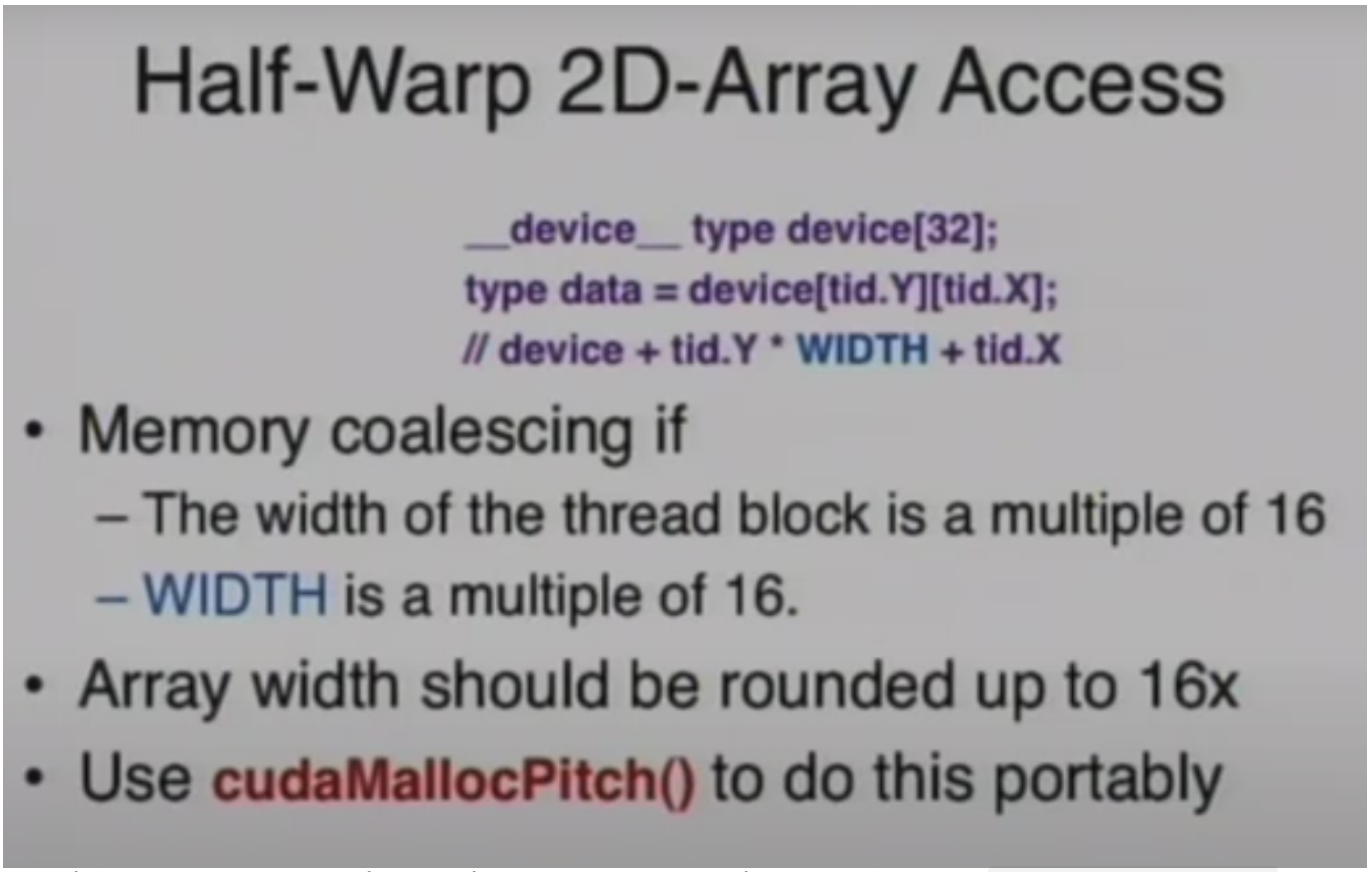
In 2.0:



Control flow

Conditions should be written in a way to minimize divergent wraps. So we can arrange wraps such that all threads in the wrap have the same condition flow. Avoid using syncthreads. Try to instead just write to shared memory or something in that nature. Getting data from global memory and things like that is A LOT slower, so optimize with shared/ local memory. ~~The cache line size if also LARGE in global memory so it is unoptimized in terms of fetching fragmented memory.~~


Global Memory Coalescing
• Memory transactions are 32, 64, or 128 bytes
– Must be aligned to as many bytes
• Upto 16 accesses by a half-warp coalesced into a single transaction if addresses are within a segment of
– 32 bytes if all threads access 8-bit words,
– 64 bytes if all threads access 16-bit words,
– 128 bytes if all threads access 32-bit or 64-bit words
– Two transactions for 128 bit words



So if the array width is not a multiple of 16, `cudaMallocPitch()` will pad the memory.

Check the other notes also pls.

END OF CUDA