Computer Architecture
计 算 机 体 系 结 构

第10讲：DLP & GPU（4）

张献伟

xianweiz.github.io

DCS3013, 11/7/2022

Review Questions

- host and device in GPU programming?

  host = CPU, device = GPU. Host offloads kernels onto devices.
- relationship of kernel and grid?

  Threads to execute the kernel are organized as a grid.
- grid and block in GPU?

  Grid: all threads to run the kernel; Grid is further divided into blocks.
- GPU core?

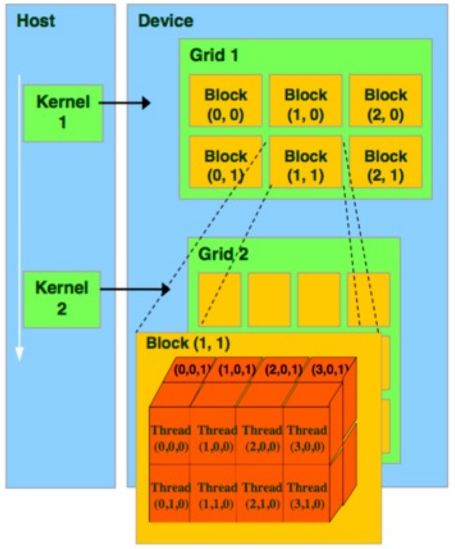
  CUDA or streaming core, the basic unit to execute a thread.
- CUDA: myKernel<<<blocks, threads, 0, 0>>>(...)?.

  Grid dimensions, i.e., how many blocks to run the kernel grid
- steps of kernel launch?

  User-mode queue, command processor, blocks to SM, warps

Concurrency[并发]

- GPU is mainly known for its data-level parallelism[数据级并行]
  - Thousands of cores, with thousands of outstanding threads
  - Simultaneously computing the same function on lots of data elements
- Still need task-level parallelism[任务级并行]
  - GPU is underutilized by a single application process
  - Doing two or more completely different tasks in parallel
  - Similar to the task parallelism that is found in multithreaded CPU applications
- Techniques
  - Multi-process service (MPS)
  - Streams



http://www.mat.unimi.it/users/sansotte/cuda/CUDA_by_Example.pdf

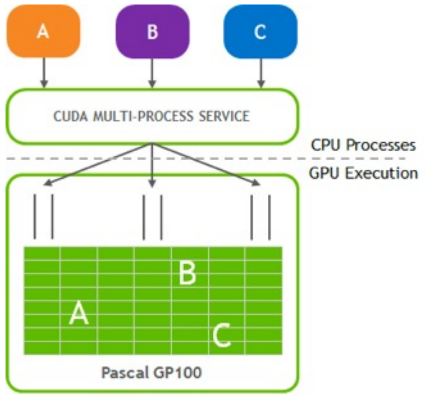
GPU Context[上下文]

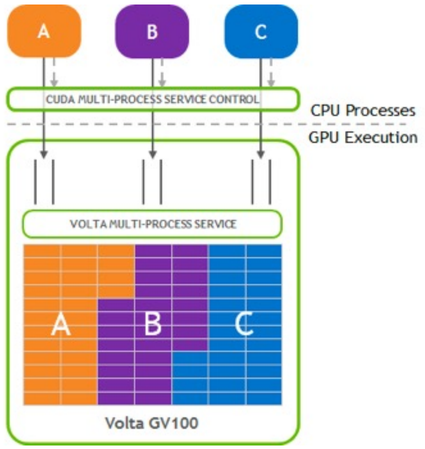
- A GPU program starts by creating a **context**
  - Either explicitly using the driver API or implicitly using the runtime API, for a specific GPU
- The **context** encapsulates all the hardware resources necessary for the program to be able to manage memory and launch work on that GPU
- Each process has a unique context[唯一]
  - Only a single context can be active on a device at a time
  - Multiple processes (e.g. MPI) on a single GPU could not operate concurrently

https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

MPS[多进程服务]

- **MPS**: multiple-process service, a software layer that sits between the driver and your application
  - Routes all CUDA calls through a single context
  - Multiple processes can execute concurrently
- Allows multiple processes to share a single GPU context, to utilize **Hyper-Q** capabilities
  - Hardware feature to construct multiple connections to GPU
  - Hyper-Q allows kernels to be processed concurrently on the same GPU

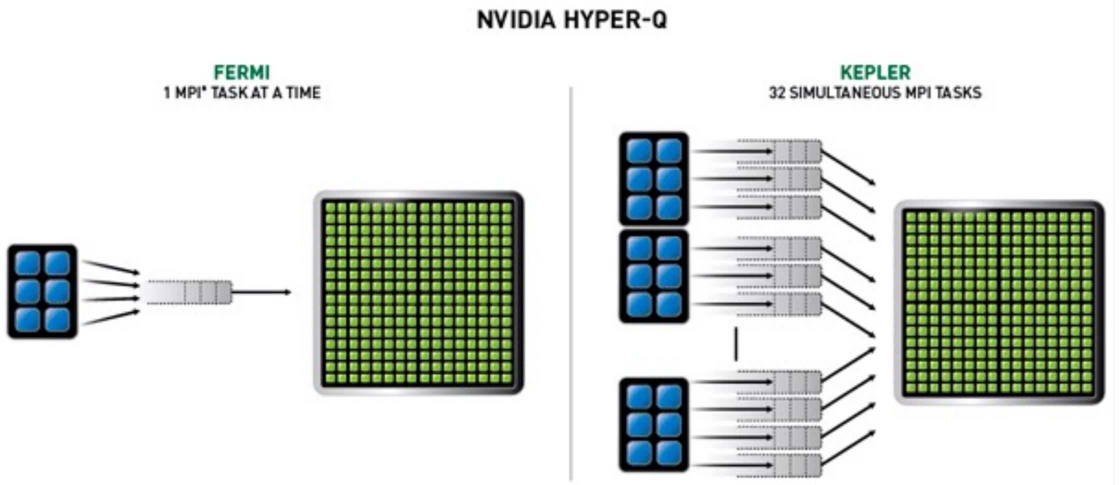



A
B
C
CUDA MULTI-PROCESS SERVICE CONTROL
CPU Processes
GPU Execution
VOLTA MULTI-PROCESS SERVICE
A
B
C
Volta GV100

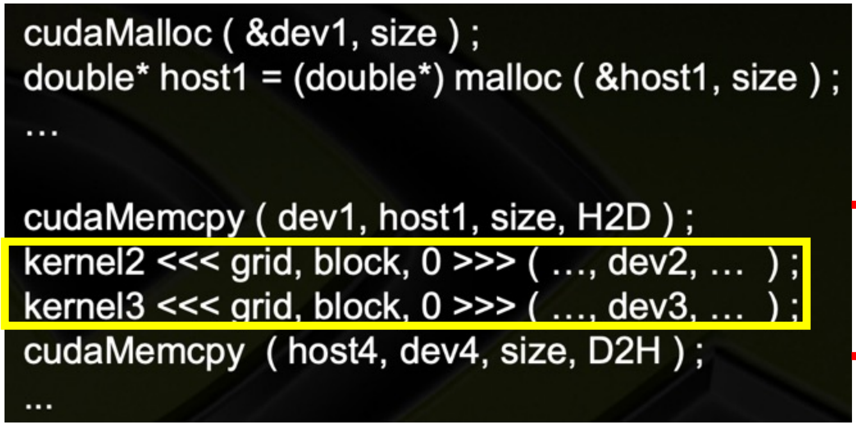
https://on-demand.gputechconf.com/gtc/2015/presentation/S5584-Priyanka-Sah.pdf

Hyper-Q[超队列]

- GPU's with **Hyper-Q** have a concurrent scheduler to schedule work from work queues belonging to a single CUDA context
- Work launched to the compute engine from work queues belonging to the same CUDA context can execute concurrently on the GPU

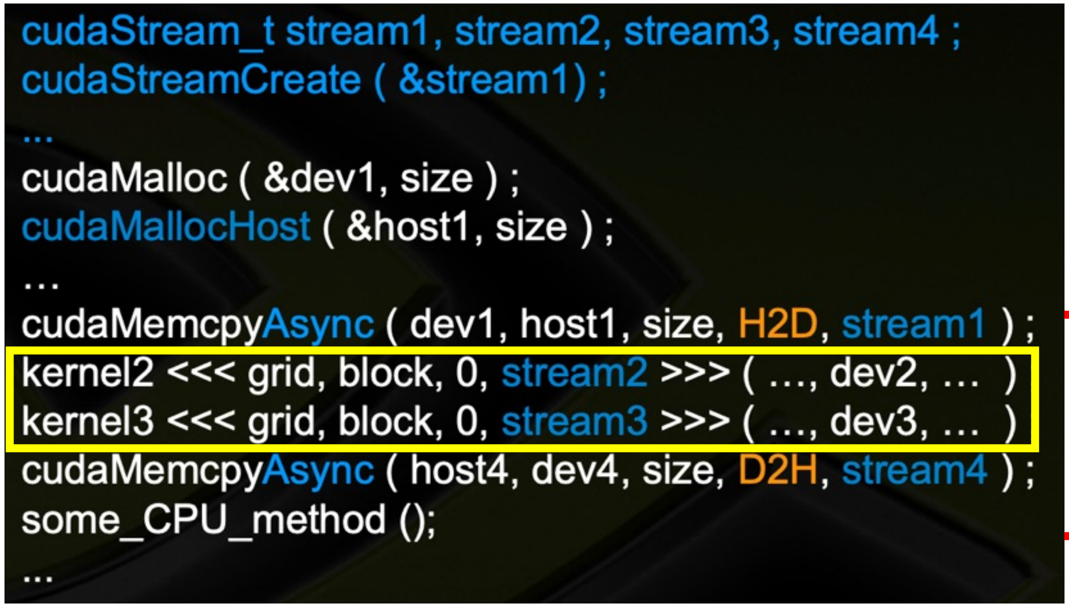


Code Example



```
cudaMalloc ( &dev1, size ) ;
double* host1 = (double*) malloc ( &host1, size ) ;
…

cudaMemcpy ( dev1, host1, size, H2D ) ;
kernel2 <<< grid, block, 0 >>> ( …, dev2, …  ) ;
kernel3 <<< grid, block, 0 >>> ( …, dev3, …  ) ;
cudaMemcpy  ( host4, dev4, size, D2H ) ;
…
```

Completely synchronous



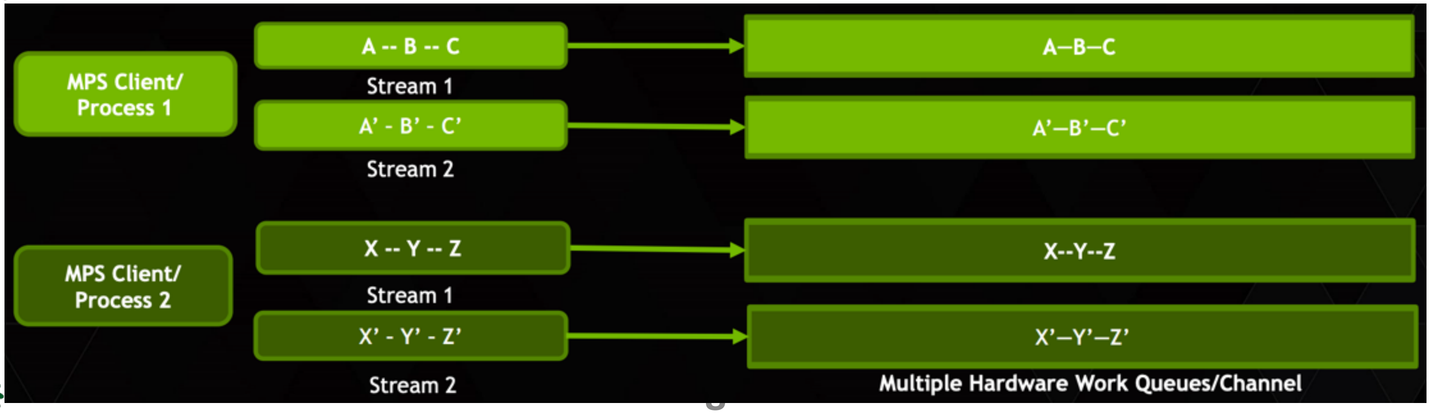
```
cudaStream_t stream1, stream2, stream3, stream4 ;
cudaStreamCreate ( &stream1) ;
…
cudaMalloc ( &dev1, size ) ;
cudaMallocHost ( &host1, size ) ;
…
cudaMemcpyAsync ( dev1, host1, size, H2D, stream1 ) ;
kernel2 <<< grid, block, 0, stream2 >>> ( …, dev2, …  )
kernel3 <<< grid, block, 0, stream3 >>> ( …, dev3, …  )
cudaMemcpyAsync ( host4, dev4, size, D2H, stream4 ) ;
some_CPU_method ();
…
```

Potentially overlapped

https://developer.download.nvidia.cn/CUDA/training/StreamsAndConcurrencyWebinar.pdf

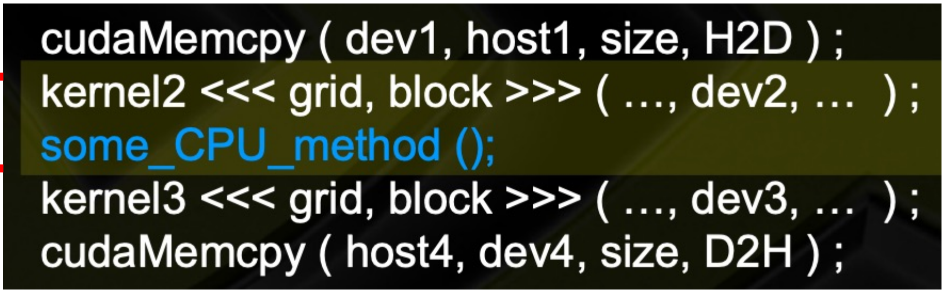
Stream[流]

- All work on the GPU is launched either explicitly into a CUDA **stream**, or implicitly using a default stream
- A **stream** is a software abstraction that represents a sequence of commands to be executed in order
  - May be a mix of kernels, copies, and other commands
- CUDA streams are aliased onto one or more '**work queues**' on the GPU by the driver
  - Work queues are hardware resources that represent an in-order sequence of the subset of commands in a stream



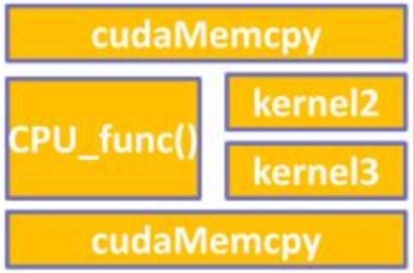
Synchronous/Asynchronous[同步/异步]

- All GPU API calls are either **synchronous** or **asynchronous** w.r.t the host
  - Synchronous: enqueue work and wait for completion
  - Asynchronous: enqueue work and return immediately
  - a.k.a., **blocking** vs. **non-blocking**[阻塞/非阻塞]
- The kernel launch function, *hipLaunchKernelGGL*, is **non-blocking** for the host
  - After sending instructions/data, the host continues immediately while the device executes the kernel
  - If you know the kernel will take some time, this is a good area to do some work on the host



Potentially overlap

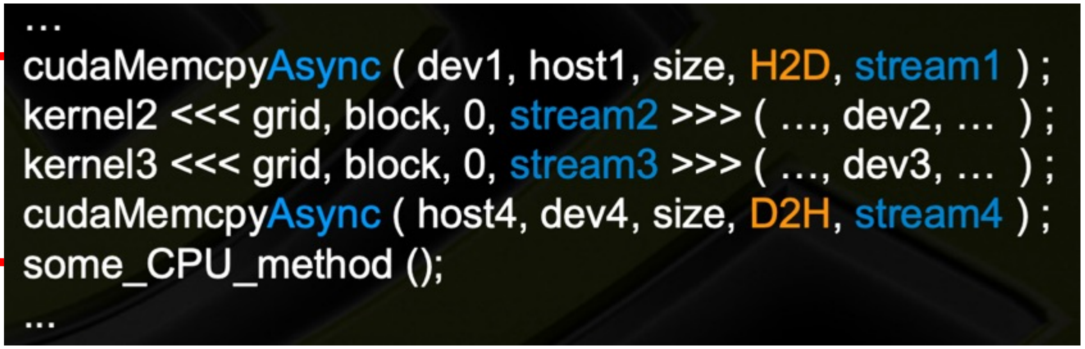
```
cudaMemcpy ( dev1, host1, size, H2D ) ;
kernel2 <<< grid, block >>> ( …, dev2, … ) ;
some_CPU_method ();
kernel3 <<< grid, block >>> ( …, dev3, … ) ;
cudaMemcpy ( host4, dev4, size, D2H ) ;
```



Synchronous/Asynchronous(cont.)

- However, *hipMemcpy* is **blocking**
  - The data pointed to in the arguments can be accessed/modified after the function returns
- The non-blocking version is *hipMemcpyAsync*
  - *hipMemcpyAsync(d_a, h_a, Nbytes, hipMemcpyHostToDevice, stream)*;
  - Like *hipLaunchKernelGGL*, this function takes an argument of type hipStream_t
  - It is not safe to access/modify the arguments of *hipMemcpyAsync* without some sort of synchronization.

Potentially overlap

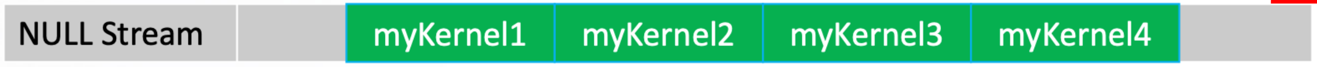


```
...
cudaMemcpyAsync ( dev1, host1, size, H2D, stream1 ) ;
kernel2 <<< grid, block, 0, stream2 >>> ( …, dev2, … ) ;
kernel3 <<< grid, block, 0, stream3 >>> ( …, dev3, … ) ;
cudaMemcpyAsync ( host4, dev4, size, D2H, stream4 ) ;
some_CPU_method ();
...
```

Streams[多流]

- A stream is a queue of device work
  - Host places work in the queue and continues on immediately
  - Device schedules work from streams when resources are free
- Operations are placed within a stream
  - e.g. Kernel launches, memory copies
- Default stream
  - Unless otherwise specified all calls are placed into a default stream (“Stream 0” or “NULL stream”)
    - Stream 0 has special sync rules: synchronous with all streams
    - Operations in stream 0 cannot overlap other streams

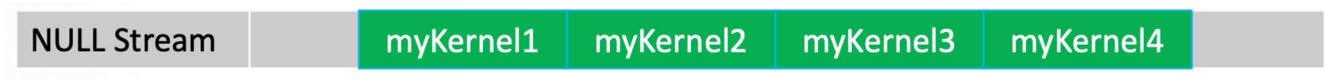
```
hipLaunchKernelGGL(myKernel1, dim3(1), dim3(256), 0, 0, 256, d_a1);
hipLaunchKernelGGL(myKernel2, dim3(1), dim3(256), 0, 0, 256, d_a2);
hipLaunchKernelGGL(myKernel3, dim3(1), dim3(256), 0, 0, 256, d_a3);
hipLaunchKernelGGL(myKernel4, dim3(1), dim3(256), 0, 0, 256, d_a4);
```



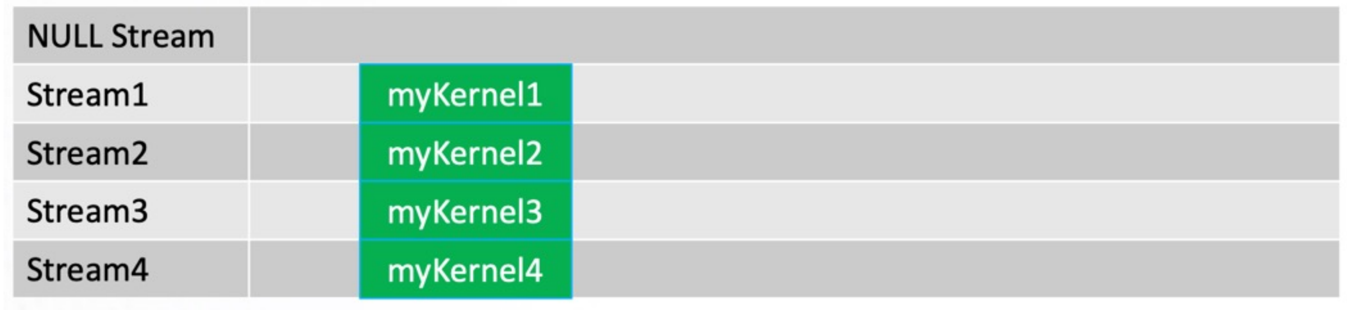
https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf

Streams (cont.)

- Operations within the same stream are ordered (FIFO) and cannot overlap
- Operations in different streams are unordered and can overlap



```
hipLaunchKernelGGL(myKernel1, dim3(1), dim3(256), 0, stream1, 256, d_a1);
hipLaunchKernelGGL(myKernel2, dim3(1), dim3(256), 0, stream2, 256, d_a2);
hipLaunchKernelGGL(myKernel3, dim3(1), dim3(256), 0, stream3, 256, d_a3);
hipLaunchKernelGGL(myKernel4, dim3(1), dim3(256), 0, stream4, 256, d_a4);
```



Synchronization[同步]

- How do we coordinate execution on device streams with host execution?
  - Need some synchronization points.
- *hipDeviceSynchronize()*; / *cudaDeviceSynchronize()*
  - Heavy-duty sync point
  - Blocks host until **all work** in **all device streams** has reported complete
- *hipStreamSynchronize(stream)*; / *cudaStreamSynchronize(stream)*
  - Blocks host until **all work** in **stream** has reported complete
- Can a stream synchronize with another stream?
  - For that we need 'Events'

Events[事件]

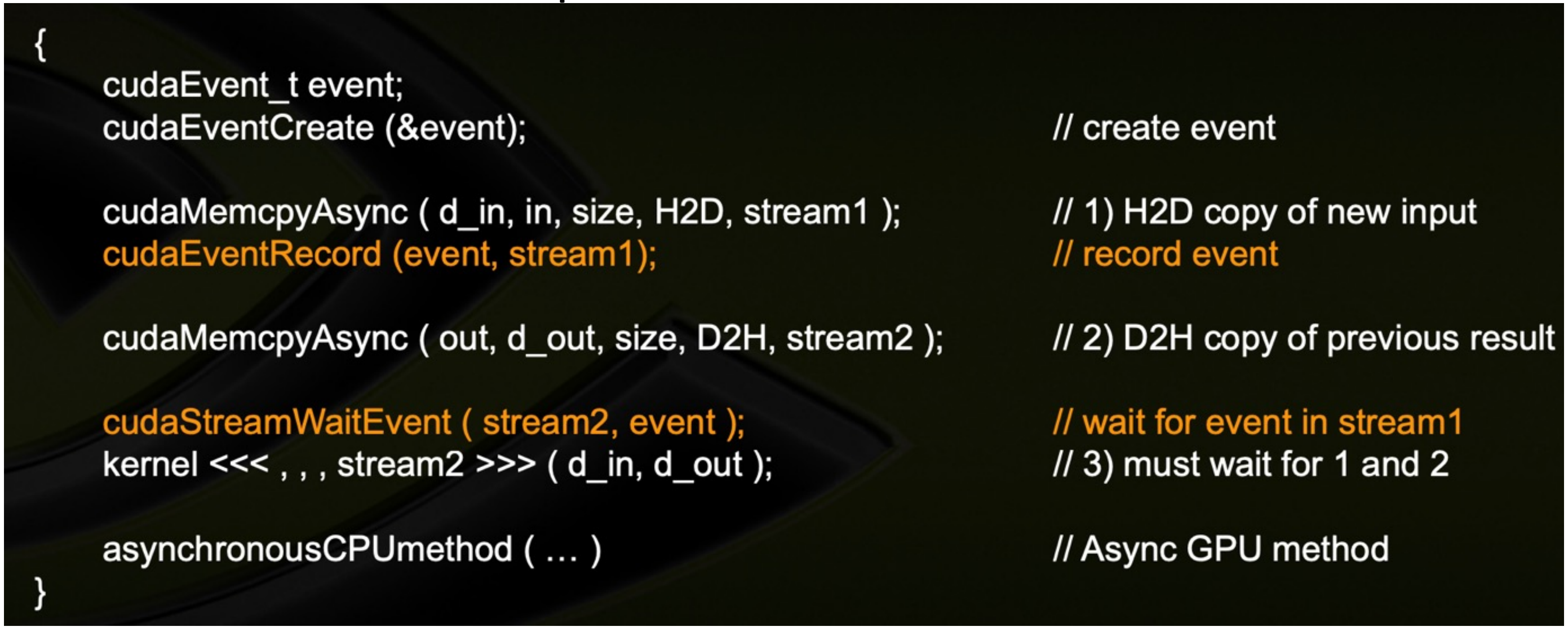
- Provide a mechanism to signal when operations have occurred in a stream
  - Useful for profiling and synchronization
  - Events have a boolean state: Occurred (default), Not Occurred
- A *hipEvent_t* object is created on a device via:
  - *hipEvent_t event;*
  - *hipEventCreate(&event);*
- We queue an event into a stream:
  - *hipEventRecord(event, stream)*;
  - The event records what work is currently enqueued in the stream
  - When the stream's execution reaches the event, the event is considered 'complete'
- At the end of the app, event objects should be destroyed:
  - *hipEventDestroy(event)*;

Events (cont.)

- *hipEventSynchronize(event)*;
  - Block host until event reports complete
  - Only a synchronization point w.r.t. the stream where event was enqueued
- *hipEventElapsedTime(&time, startEvent, endEvent)*;
  - Returns the time in *ms* between when two events, *startEvent* and *endEvent*, completed
  - Can be very useful for timing kernels/memcpys
- *hipStreamWaitEvent(stream, event)*;
  - Non-blocking for host
  - Instructs all future work submitted to stream to wait until event reports complete
  - Primary way we enforce an 'ordering' between tasks in separate streams

Example

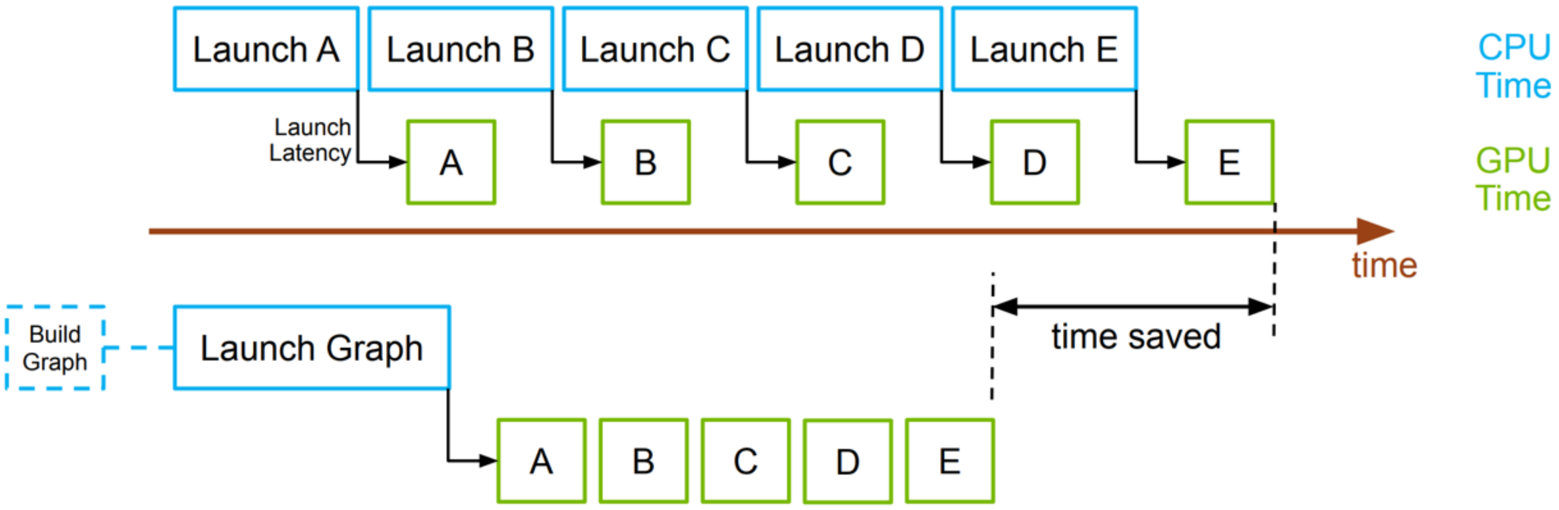
- *cudaEventRecord(&event, stream)*
  - Enqueue an event into stream, whose state is set to occurred when reaching the front of the stream
- *cudaStreamWaitEvent(stream, event)*
  - The stream cannot proceed until the event occurs



```
{
    cudaEvent_t event;
    cudaEventCreate (&event);                                  // create event

    cudaMemcpyAsync ( d_in, in, size, H2D, stream1 );          // 1) H2D copy of new input
    cudaEventRecord (event, stream1);                          // record event

    cudaMemcpyAsync ( out, d_out, size, D2H, stream2 );        // 2) D2H copy of previous result

    cudaStreamWaitEvent ( stream2, event );                    // wait for event in stream1
    kernel <<< , , , stream2 >>> ( d_in, d_out );              // 3) must wait for 1 and 2

    asynchronousCPUmethod ( … )                                // Async GPU method
}
```

Task Graph[任务图]

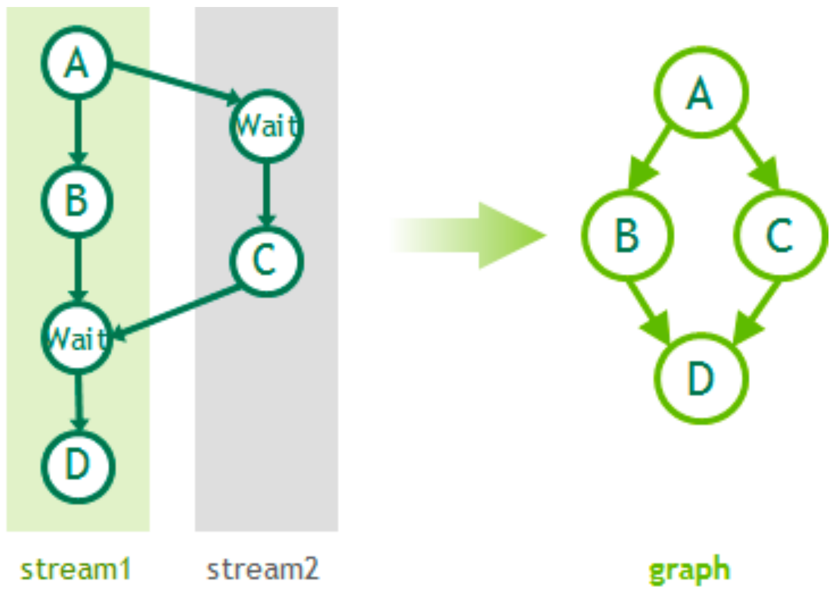
- CPU launches each kernel to GPU
  - When kernel runtime is short, execution time is dominated by CPU launch cost
- CUDA graph launch submits all work at once, reducing CPU cost
  - A sequence of operations, connected by dependencies



Example

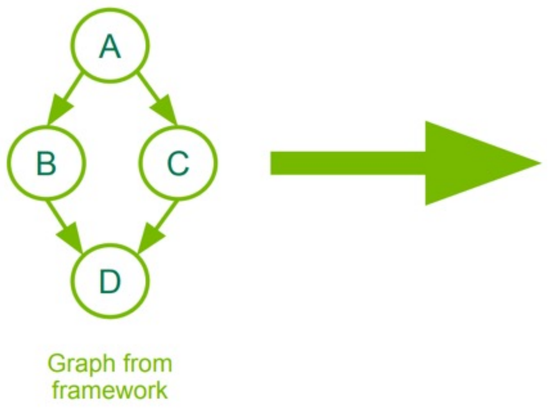
- Capture CUDA stream work into a graph[基于流构建]

```
// Start by initiating stream capture
cudaStreamBeginCapture(&stream1);

// Build stream work as usual
A<<< ..., stream1 >>>();
cudaEventRecord(e1, stream1);
B<<< ..., stream1 >>>();
cudaStreamWaitEvent(stream2, e1);
C<<< ..., stream2 >>>();
cudaEventRecord(e2, stream2);
cudaStreamWaitEvent(stream1, e2);
D<<< ..., stream1 >>>();

// Now convert the stream to a graph
cudaStreamEndCapture(stream1, &graph);
```



Example (cont.)

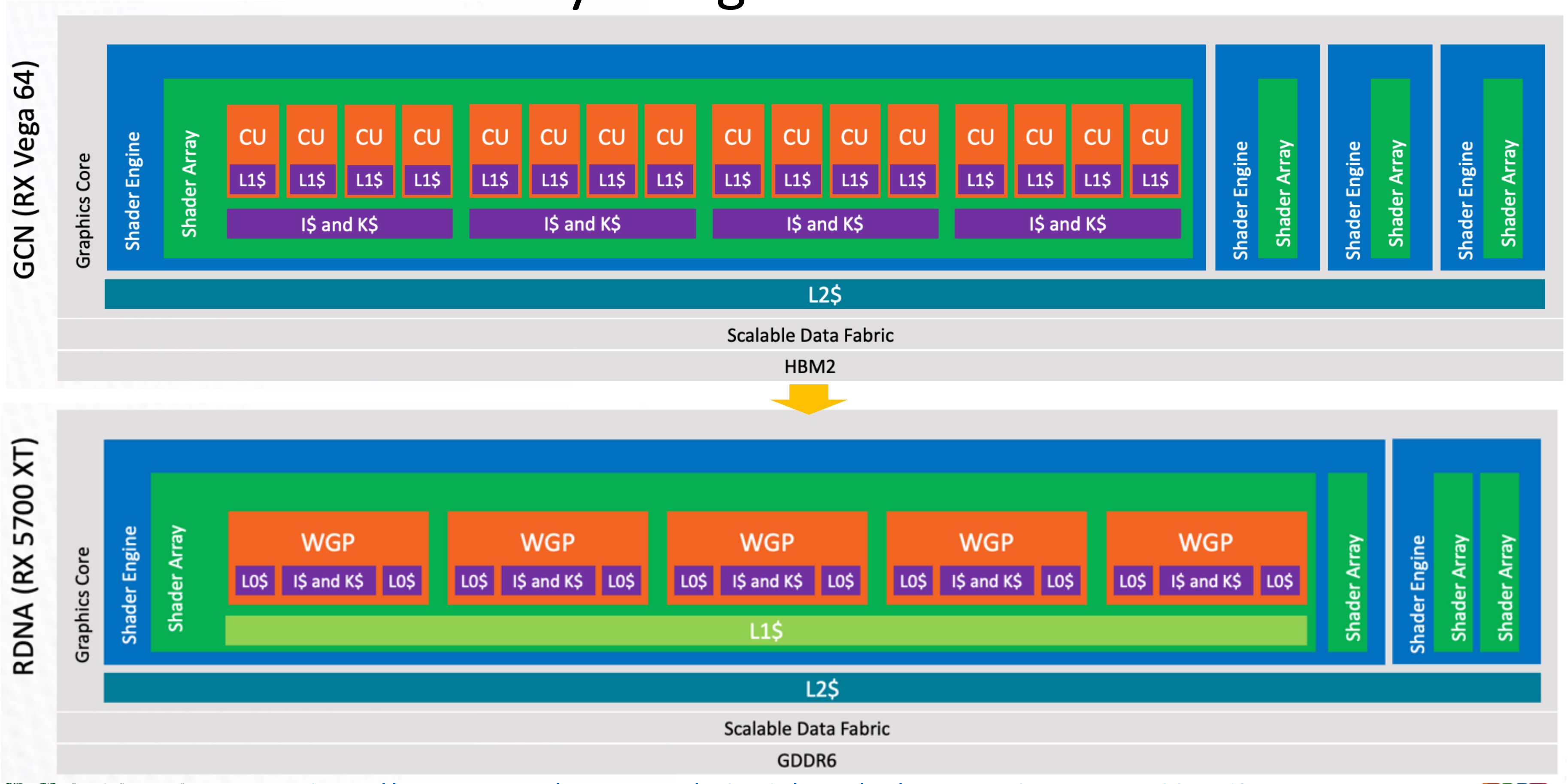
- Create graphs directly[直接构建]
  - Map graph-based workflows directly into CUDA



```
// Define graph of work + dependencies
cudaGraphCreate(&graph);

cudaGraphAddNode(graph, kernel_a, {}, ...);
cudaGraphAddNode(graph, kernel_b, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_c, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_d, { kernel_b, kernel_c }, ...);

// Instantiate graph and apply optimizations
cudaGraphInstantiate(&instance, graph);

// Launch executable graph 100 times
for(int i=0; i<100; i++)
    cudaGraphLaunch(instance, stream);
```

GPU Memory Hierarchy[存储层级]

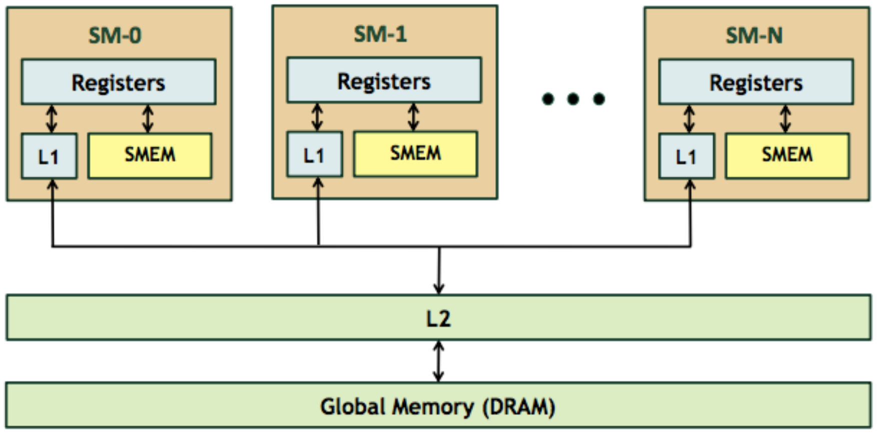
- CU internal memories: registers, caches, …
- Shared L2, off-chip HBM/GDDR
- RDNA fundamentally reorganizes the architecture

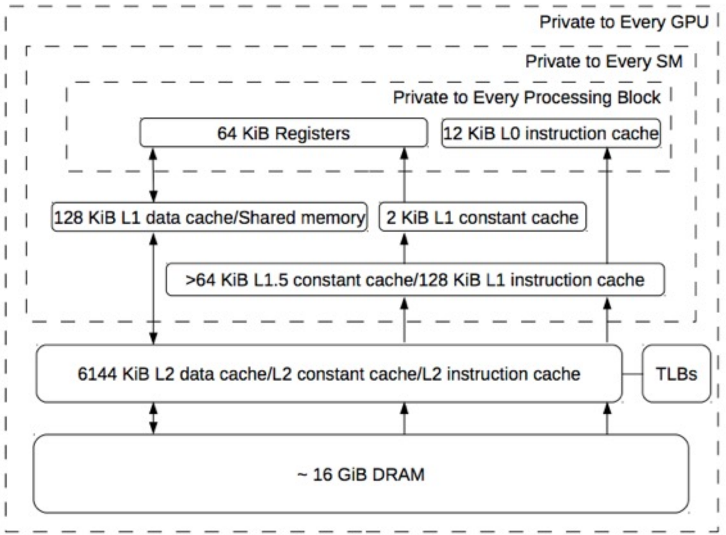


https://gpuopen.com/wp-content/uploads/2019/08/RDNA_Architecture_public.pdf

Memory Hierarchy

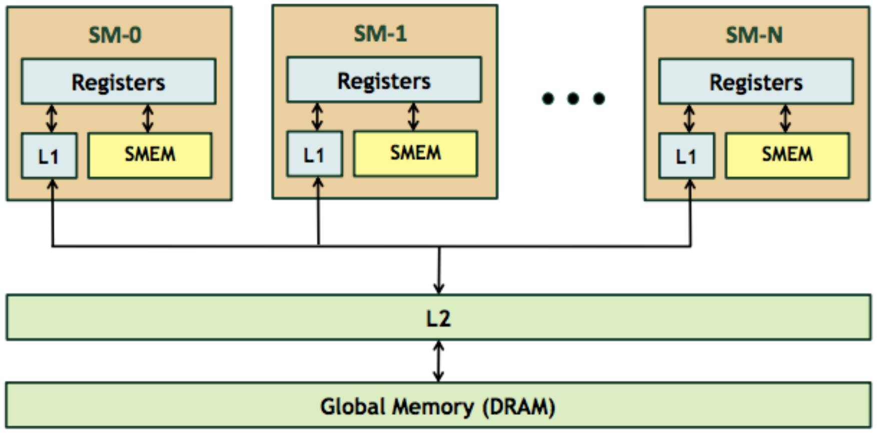
- **Register**: per-thread, deallocate when the thread done
- **Cache**: instruction, data, RO constant, RO texture
- **Global memory**: per-GPU, shared across kernels
- **Shared memory (SMEM)**: per-block, deallocate when the block done (and re-allocated to other blocks) LDS for AMD GPU
- **Constant memory (CMEM)**: part of device memory, use dedicated per-SM constant cache; shared across kernels




Private to Every GPU
Private to Every SM
Private to Every Processing Block
64 KiB Registers
12 KiB L0 instruction cache
128 KiB L1 data cache/Shared memory
2 KiB L1 constant cache
>64 KiB L1.5 constant cache/128 KiB L1 instruction cache
6144 KiB L2 data cache/L2 constant cache/L2 instruction cache
TLBs
~ 16 GiB DRAM

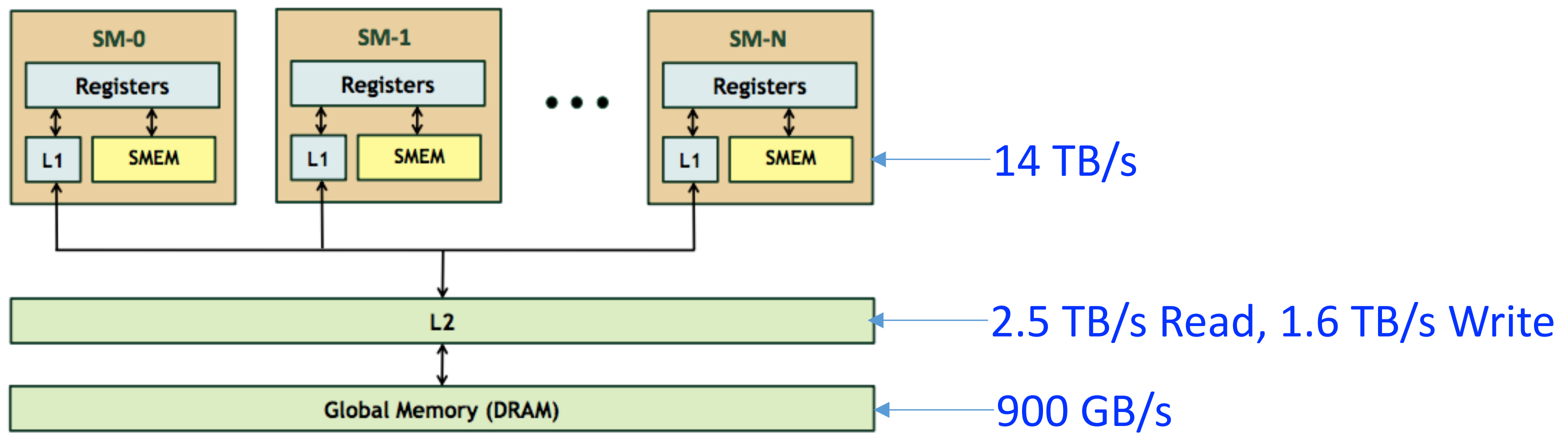
V100 Memory Hierarchy[存储层级]

- 80 SMs
  - Cores per SM: 64 INT32, 64 FP32, 32 FP64, 8 Tensor
  - Peak TFLOPS: 15.7 FP32, 7.8 FP64, 125 Tensor
  - Per SM: 64K 32-bit Register File, 128KB SMEM+L1
- 6MB L2 cache, 16GB 900GB/s HBM2
  - Shared by all SMs
  - For comparison: 20MB RF, 10MB SMEM+L1



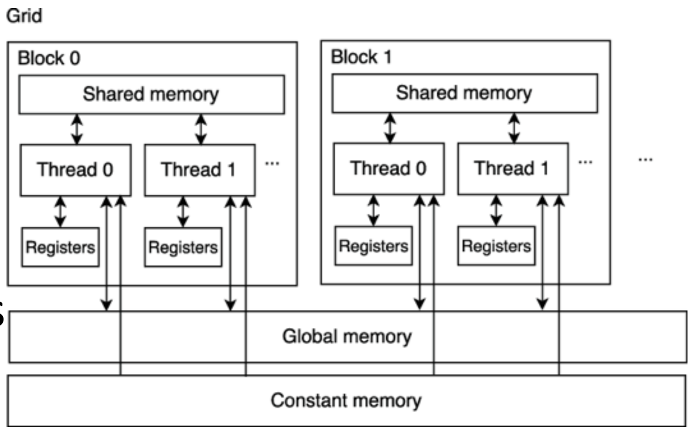
SMEM & CMEM

- SMEM benefits compared to DRAM:
  - 20-40x lower latency
  - ~15x higher bandwidth
  - Access granularity: 4B vs. 32B
- Constant memory (CMEM):
  - Total constant data size limited to 64KB
  - Throughput: 4B/clock per SM
  - Can be used directly in arithmetic insts (saving regs)



Resource Limits[资源限制]

- Threads[线程]
  - Max per SM: 32 TBs, 64 Warps (i.e., 2048 threads)
    - Up to 1024 threads/TB
    - TBs should be of at least 2 warps
- Registers[寄存器]
  - Max: 64K regs/TB, 255 regs/thread
    - Per SM: total 64K regs
    - If exceeding 255 regs, then spilling happens
- Memory[存储]
  - Max 96KB SMEM per SM (default 48KB)
- 100% occupancy[若满载]
  - 2048 threads/SM, 64K regs/SM → 32 regs/thread (128B)
  - 2048 threads/SM, 96KB smem/SM → 32B/thread



Memory Space Specifiers[存储空间指定]

- Variable memory space specifiers denote the memory location on the device of a variable
- **__device__**: declares a variable that resides on the device, by default
  - Resides in global memory space
  - Has the lifetime of the CUDA context in which it is created
  - Is accessible from all the threads within the grid and from the host through the runtime library
- **__constant__**: declares a variable that resides in constant memory space
  - Optionally used together with __device__
- **__shared__**: declares a variable that resides in shared memory space
  - Has the lifetime of the block,
  - Is only accessible from all the threads within the block

Memory Space Specifiers (cont.)

- **__managed__**: declares a variable that can be referenced from both device and host code
  - optionally used together with __device__
  - Has the lifetime of an application
- An automatic variable declared in device code without any of the __device__, __shared__ and __constant__ specifiers generally resides in a register
  - However in some cases the compiler might choose to place it in local memory, which can hurt performance

Variable declaration	Memory	Scope	Lifetime
__device__ int globalVar;	global	grid	application
__shared__ int sharedVar;	shared	block	block
__constant__ int constantVar;	constant	grid	application
int localVar;	register	thread	thread
int localArray[10];	local	thread	thread

Local Memory['本地'内存]

- Name refers to memory where registers and other thread-data is spilled
  - Usually when one runs out of SM resources
  - "Local" because each thread has its own private area
- Use **case 1**: register spilling[寄存器溢出]
  - Fermi hardware limit is 63 registers per thread (255 now)
  - Programmer can specify lower registers/thread limits:
    - To increase occupancy (number of concurrently running threads)
    - -maxrregcount option to nvcc, __launch_bounds__() qualifier in the code
  - LMEM is used if the source code exceeds register limit
- Use **case 2**: arrays declared inside kernels, if compiler can't resolve indexing[核函数内数组]
  - Registers aren't indexable, so have to be placed in LMEM

https://developer.download.nvidia.com/CUDA/training/register_spilling.pdf

Local Memory (cont.)

- LMEM is not really a memory
  - Bytes are actually stored in global memory
  - Differences from global memory:
    - Addressing is resolved by the compiler
    - Stores are cached in L1
- LMEM could hurt performance in two ways:
  - Increased memory traffic
  - Increased instruction count
- Spilling/LMEM usage isn't always bad
  - LMEM bytes can get contained within L1
    - Avoids memory traffic increase
  - Additional instructions don't matter much if code is not instruction-throughput limited

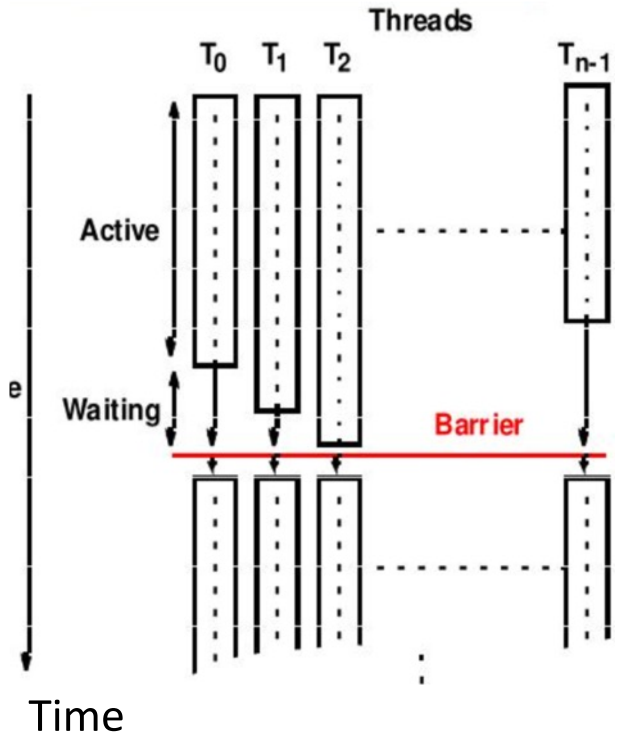
https://developer.download.nvidia.com/CUDA/training/register_spilling.pdf

Shared Memory[“共享”存储]

- A per-block, software managed cache or scratchpad
  - Programmer can modify variable declarations with __shared__ to make this variable resident in shared memory
  - Compiler creates a copy of the variable for each block
    - Every thread in that block shares the memory, but threads cannot see or modify the copy of this variable that is seen within other blocks
    - This provides an excellent means by which threads within a block can communicate and collaborate on computations
- CUDA L1 cache and SMEM are unified
  - cudaDeviceSetCacheConfig(enum cudaFuncCache)
- A mechanism is needed to **synchronize** between threads
  - *Thread A* writes a value to shared memory and we want *thread B* to do something with this value
  - We can’t have *thread B* start its work until we know the write from *thread A* is complete

Shared Memory (cont.)

- One can specify synchronization points in the kernel by calling __syncthreads()
- __syncthreads() acts as a barrier at which all threads in the block must wait before any is allowed to proceed
  - Guarantees that every thread in the block has completed instructions prior to the __syncthreads() before the hardware will execute the next inst on any thread
  - When the first thread executes the first instruction after __syncthreads(), every other thread in the block has also finished executing up to the __syncthreads()


Threads
T_0
T_1
T_2
T_{n-1}
Active
Waiting
Barrier
Time