中山大學 SUN YAT-SEN UNIVERSITY

国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

# Advanced Computer Architecture

# 高级计算机体系结构

## 第13讲：DLP & GPU (2)

张献伟

xianweiz.github.io

DCS5637, 11/23/2022

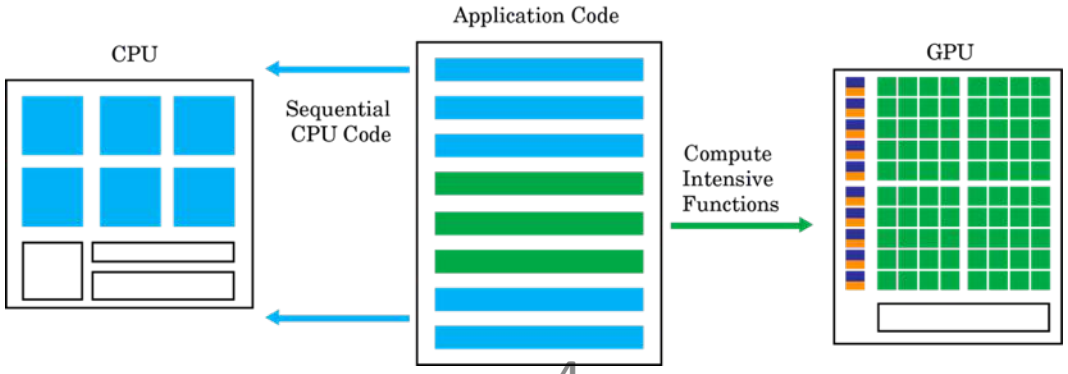
# Review Questions

- Suppose a pipeline of 5 stage. What's the ideal speedup?
  5. serial: inst retires every 5 cycles; pipeline: inst retires every cycle
- Why impossible to reach ideal speedup for pipelining?
  No perfect stage splitting; pipe overhead; hazards and deps …
- Techniques to improve ILP?
  Branch prediction; loop unrolling; dynamic scheduling; VLIW …
- SIMD?
  Single instruction, multiple data
- SIMD vs. SIMT?
  Threads to execute scalar operations.
- CPU vs. GPU?
  ILP vs. DLP; Few vs. lots of cores; O3 vs. IO; complex vs. simple

# Part-III: GPU Arch

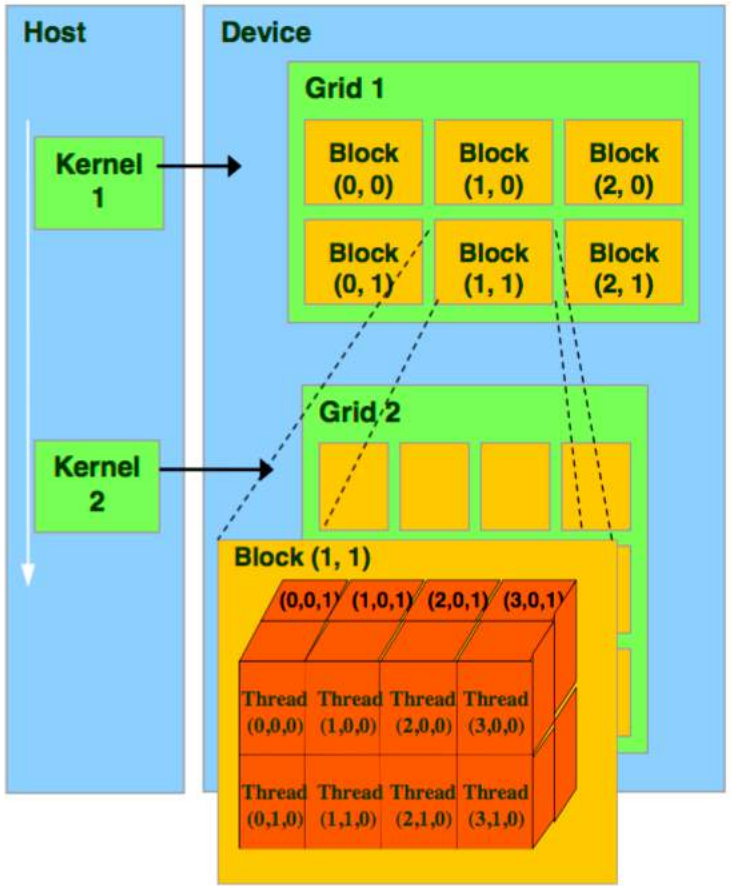
# GPU Programming Model[编程模型]

- GPU is viewed as a compute device that
  - Is a coprocessor to CPU (host)
  - Has its own main memory called device memory
  - Runs many threads in parallel
- Data-parallel parts of an application are executed on the device as **kernels**, which run in parallel on many threads
- CPU thread vs. GPU thread
  - GPU threads are very lightweight
  - A few vs. thousands for full efficiency



# Thread Organization[线程组织]

- A kernel is executed as a grid of thread blocks
- A thread block is a batch of threads that can cooperate with each other by
  - Synchronizing their execution
  - Efficiently sharing data through low-latency shared memory
- The grid and its associated blocks are just organizational constructs
  - The threads are the things that do the work



# GPU Programming Choices[编程选择]

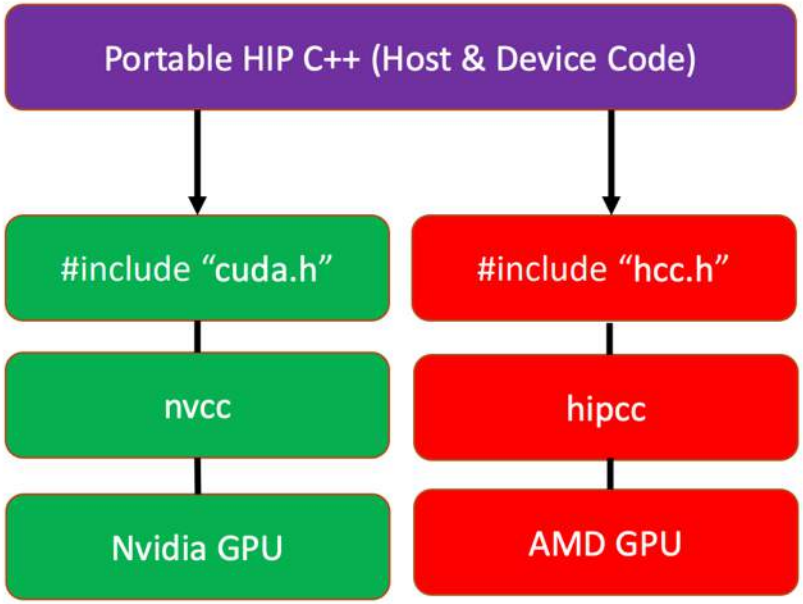
- **CUDA** – Compute Unified Device Architecture
  - Developed by Nvidia – proprietary
  - First serious GPGPU language/environment


- **OpenCL** – Open Computing Language
  - From makers of OpenGL
  - Wide industry support: AMD, Apple, Qualcomm, Nvidia (begrudgingly), etc


- **HIP** - Heterogeneous-compute Interface for Portability
  - Owned by AMD
  - A C++ runtime API and kernel language that allows developers to create portable applications that can run on AMD's accelerators as well as CUDA devices

# HIP

- Is open-source
- Provides an API for an application to leverage GPU acceleration for both AMD and Nvidia devices
- Syntactically similar to CUDA. Most CUDA API calls can be converted in place: cuda --hipify--> hip
- Supports a strong subset of CUDA runtime functionality



# Example: Putting Together

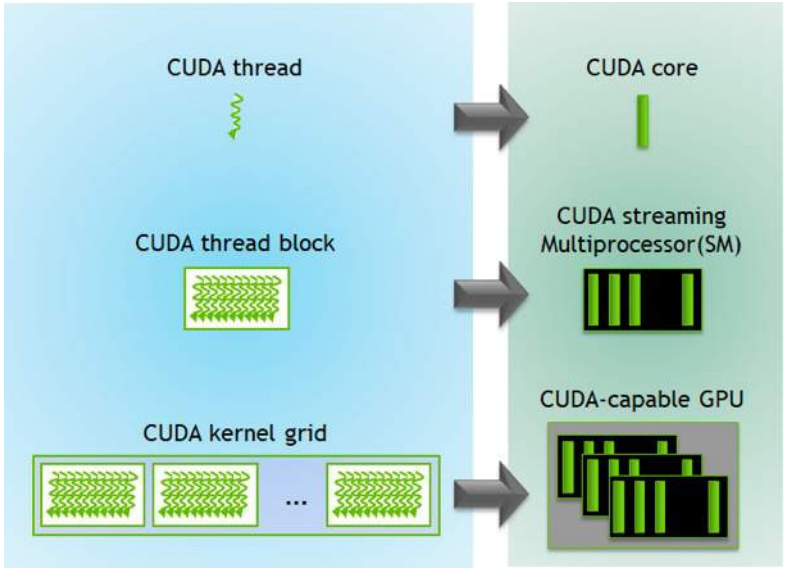
```
#include “hip/hip_runtime.h”
int main() {
  int N = 1000;
  size_t Nbytes = N*sizeof(double);
  double *h_a = (double*) malloc(Nbytes);  //host memory
  double *d_a = NULL;
  HIP_CHECK(hipMalloc(&d_a, Nbytes));
  …
  HIP_CHECK(hipMemcpy(d_a, h_a, Nbytes, hipMemcpyHostToDevice));   //copy data to device

  hipLaunchKernelGGL(myKernel, dim3((N+256-1)/256,1,1), dim3(256,1,1), 0, 0, N, d_a); //Launch kernel
  HIP_CHECK(hipGetLastError());

  HIP_CHECK(hipMemcpy(h_a, d_a, Nbytes, hipMemcpyDeviceToHost));
  …
  free(h_a);                  //free host memory
  HIP_CHECK(hipFree(d_a));    //free device memory
}
```

```
__global__ void myKernel(int N, double *d_a) {
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  if (i<N) {
    d_a[i] *= 2.0;
  }
}
```

```
#define HIP_CHECK(command) {                \
  hipError_t status = command;              \
  if (status!=hipSuccess) {                 \
    std::cerr << “Error: HIP reports ”      \
              << hipGetErrorString(status)  \
              << std::endl;                 \
    std::abort(); } }
```

# Map Kernel to Hardware[映射]

- Blocks are dynamically scheduled onto compute units (CUs) **SM for Nvidia**
  - All threads in a block execute on the same CU
  - Threads in block share LDS memory and L1 cache
- Blocks are further divided into wavefronts
  - A group of 32 or 64 threads **warp for Nvidia**
  - Wavefronts execute on SIMD units



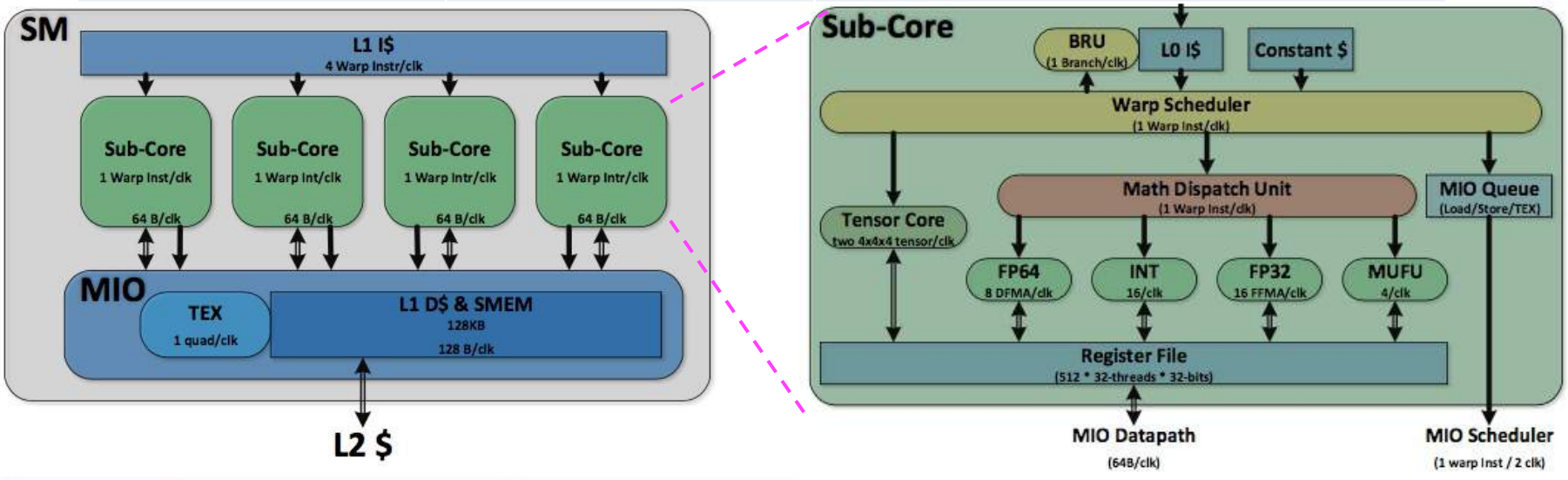

SIMD lane (streaming processor)

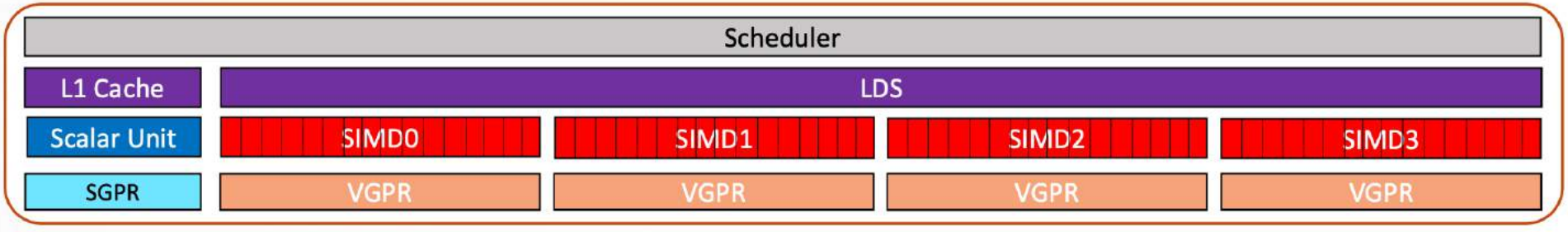
Compute unit

GPU

# Nvidia SM

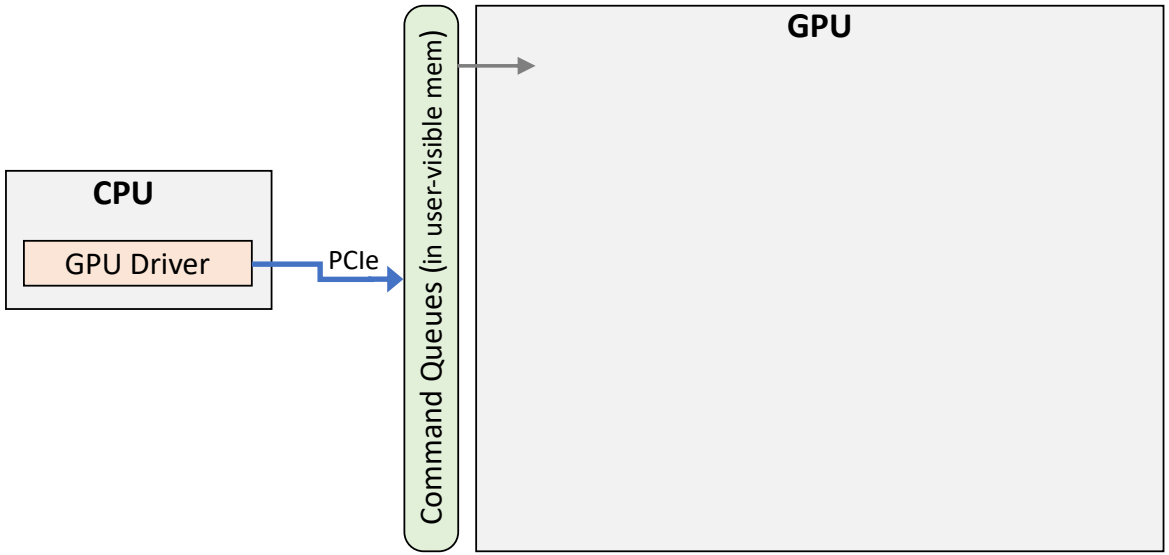
| Level | Nvidia | AMD |
|---|---|---|
| Thread | CUDA core | Streaming processor / SIMD lane |
| Warp/wavefront | SM sub-partition | SIMD unit |
| Block/workgroup | SM | Compute unit |
| All threads | GPU device | GPU device |



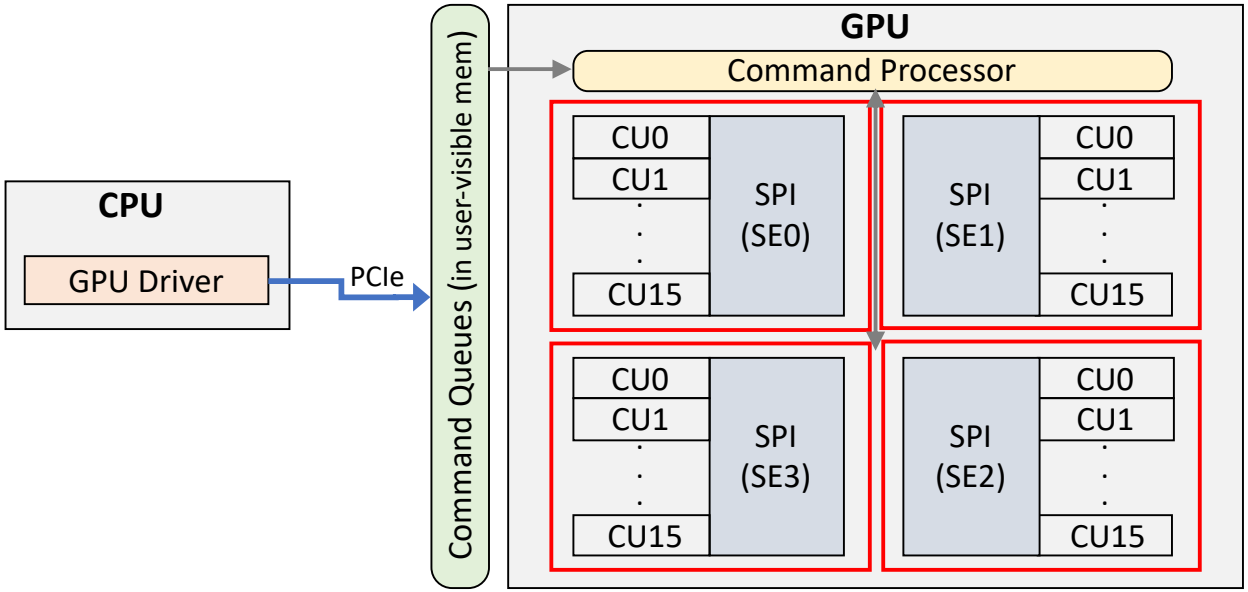
# CPU-GPU

- CPU communicates kernels to GPUs via PCIe
  - Kernel code object is filled into a dispatch *packet*
  - Next, the packet is placed into a *queue*, which is allocated by runtime and associated with a GPU
  - The *GPU* is then signaled to process packets from the queue
  - When kernel is finished, *CPU* is notified with an interrupt



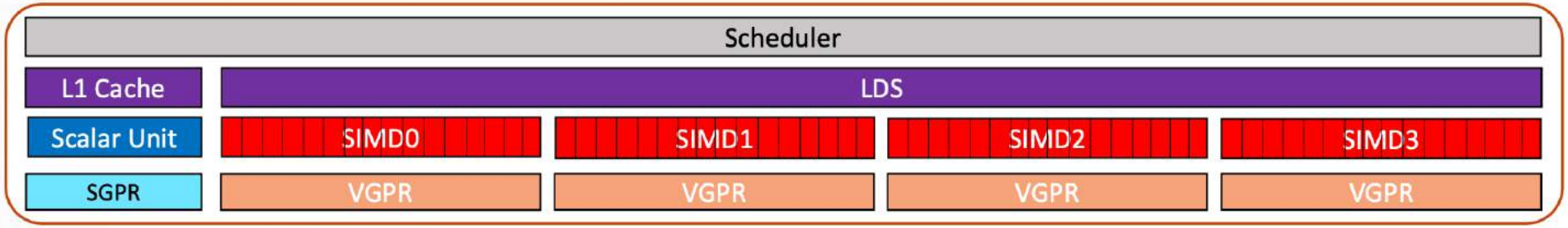
# GPU Structure

- Command processor (CP)
  - Forefront hardware component of a GPU to receive kernels
- Shader processor inputs (SPI)
  - Receives WGs from the CP **Blocks/CTAs for Nvidia**
- Compute unit (CU) **SM for Nvidia**
  - Fundamental compute component



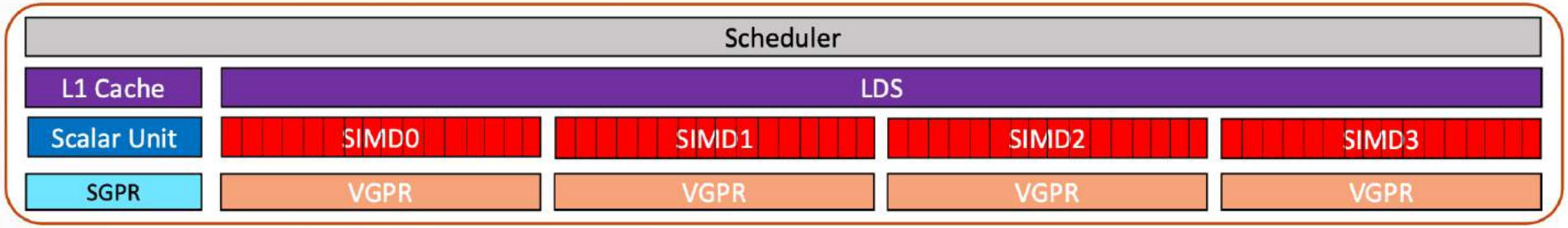
# Compute Unit

- Scheduler[调度]
  - Manage the wavefronts execution among the SIMDs
- Compute[计算]
  - SIMD: for vector processing (a.k.a., vector units, VALUs)[向量单元]
    - Is of 16 lanes in GCN, thus simultaneously executing a single operation among 16 threads
    - Has its own PC and instruction buffer (IB) for 10 WFs
  - Scalar unit[标量单元]
    - Shared by all threads in each WF, accessed on a per-WF level
    - Used for control flow, pointer arithmetic, loading a common value, etc.



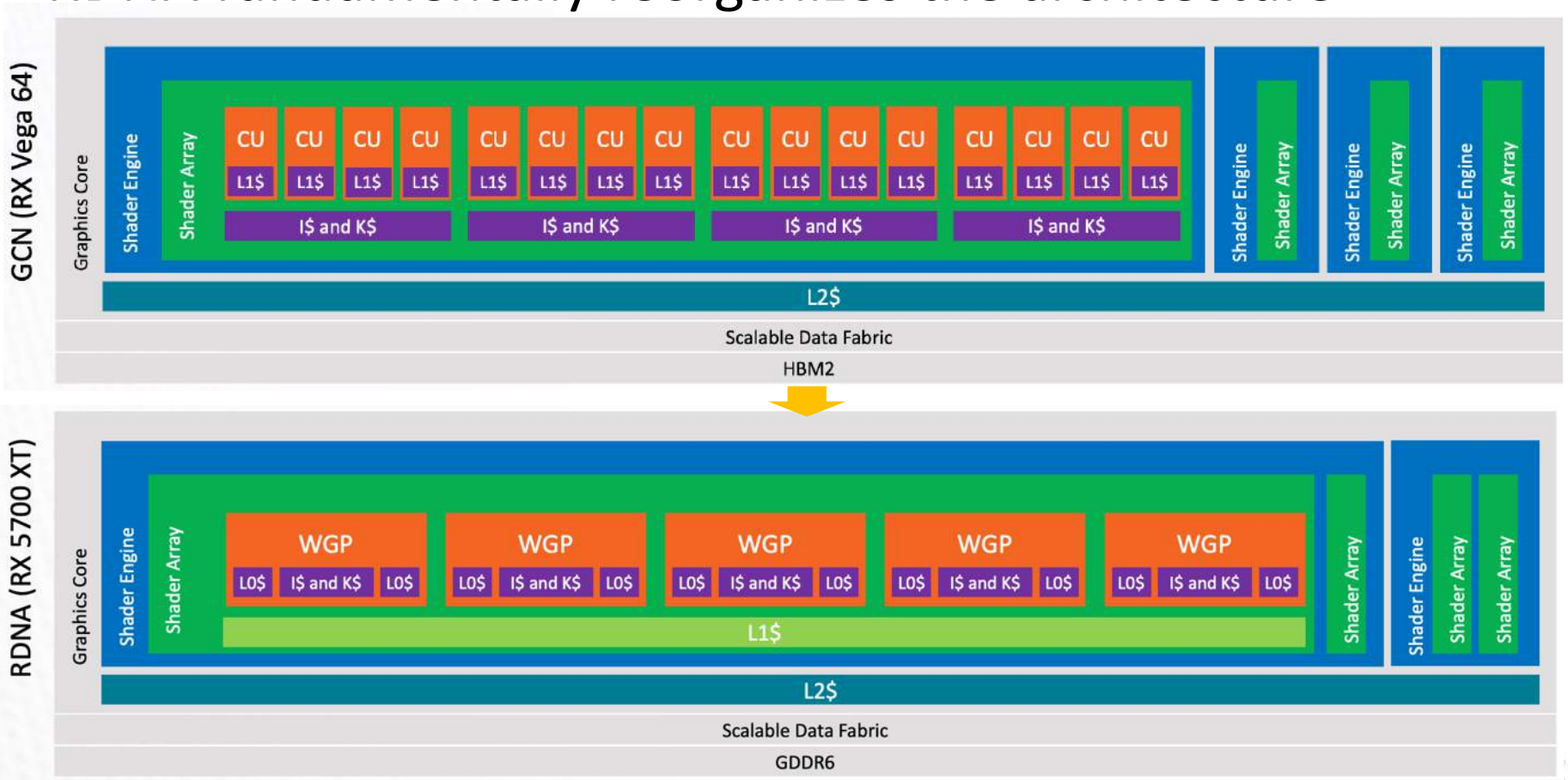
# Compute Unit (cont.)

- GPRs[通用寄存器]
  - VGPR: vector general purpose register file
    - 4x 64KB (256KB total)
    - A maximum of 256 total registers per SIMD lane – each register is 64x 4-byte entries
  - SGPR: scalar general purpose register file
    - 12.5KB per CU
- L1 cache: 16KB[一级缓存]
- LDS: local data share (or, shared memory)[片上共享存储]
  - Enables data share between threads of a block



# GPU Memory Hierarchy[存储层级]

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

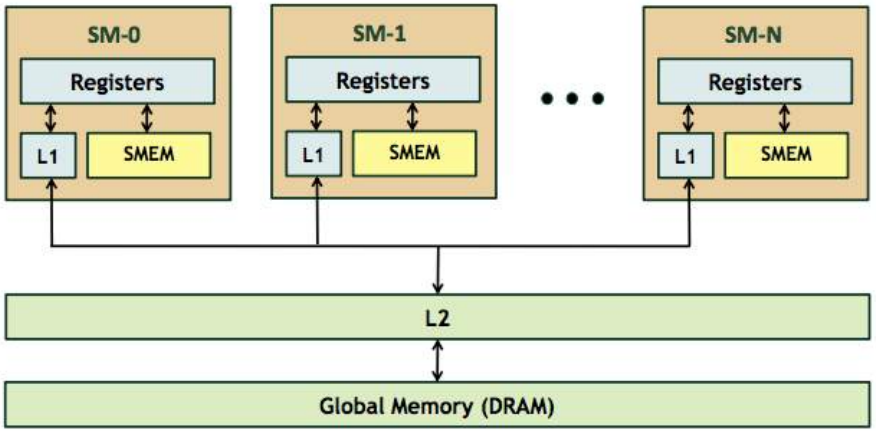


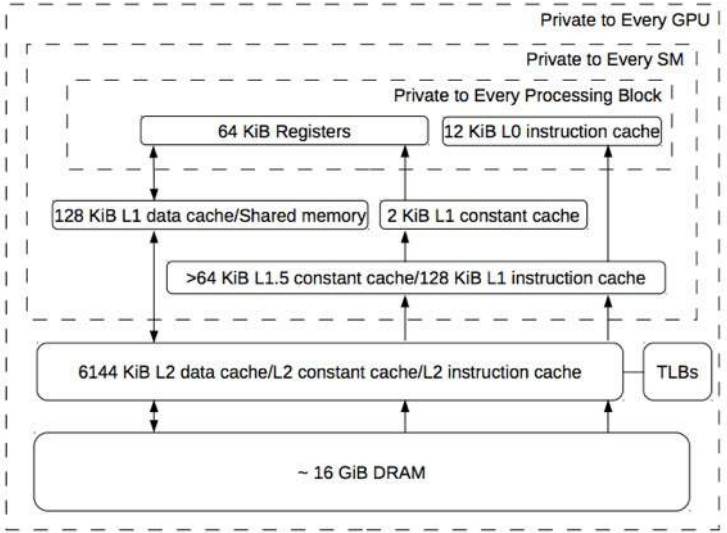



https://gpuopen.com/wp-content/uploads/2019/08/RDNA_Architecture_public.pdf

# Memory Hierarchy

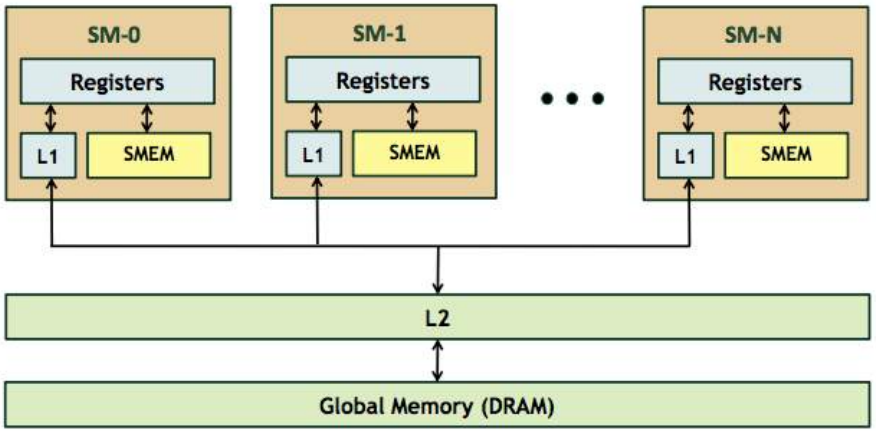
- **Register**: per-thread, deallocate when the thread done
- **Cache**: instruction, data, RO constant, RO texture
- **Global memory**: per-GPU, shared across kernels
- **Shared memory (SMEM)**: per-block, deallocate when the block done (and re-allocated to other blocks) **LDS for AMD GPU**
- **Constant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



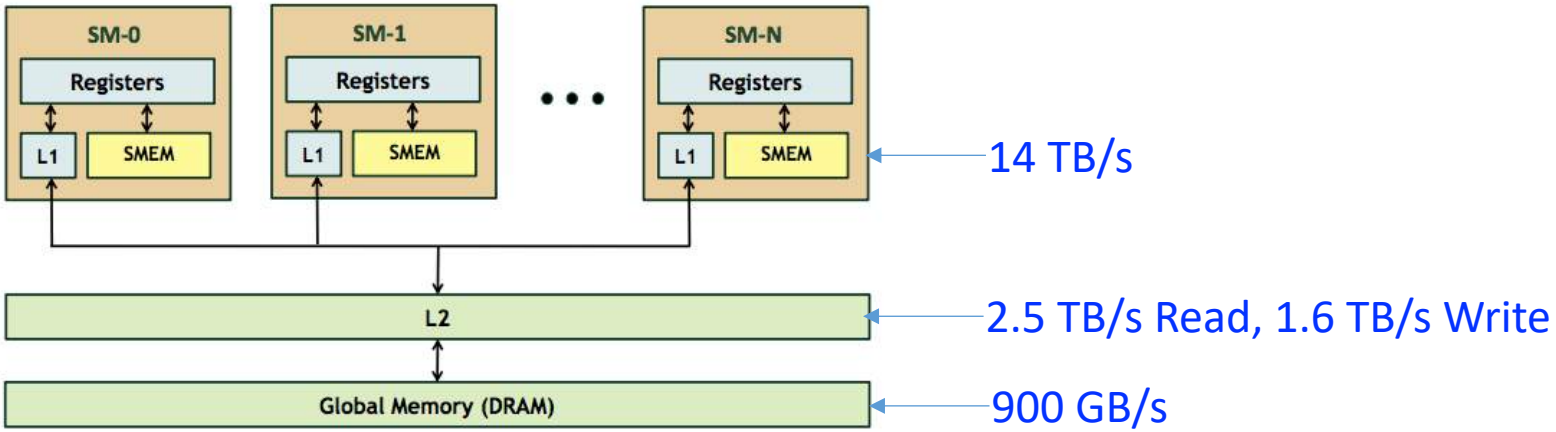
# V100 Memory Hierarchy[存储层级]

- 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



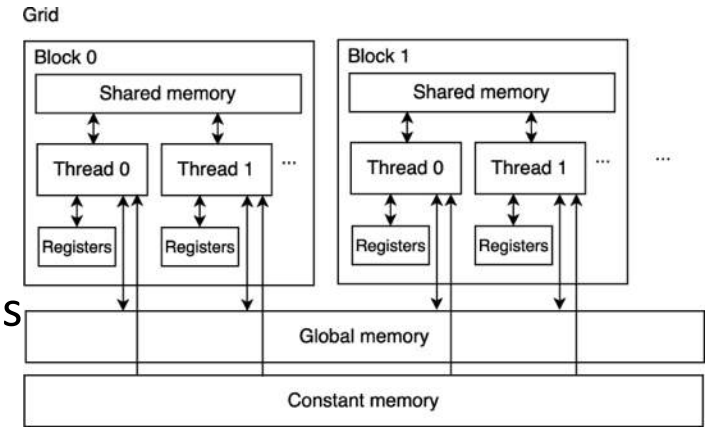
# SMEM & CMEM

- 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


https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#variable-memory-space-specifiers

# 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

# 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



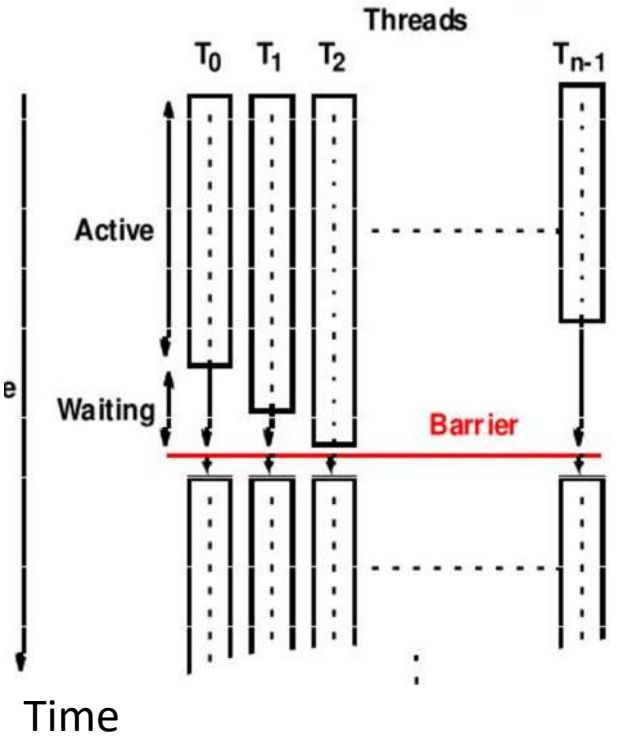
https://developer.download.nvidia.com/CUDA/training/register_spilling.pdf

# 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()





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

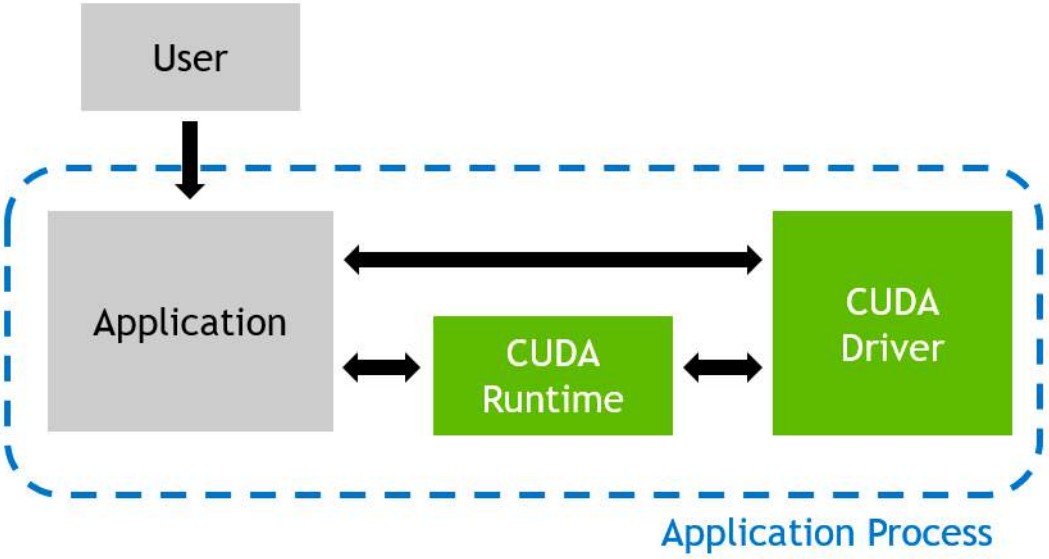
# Example

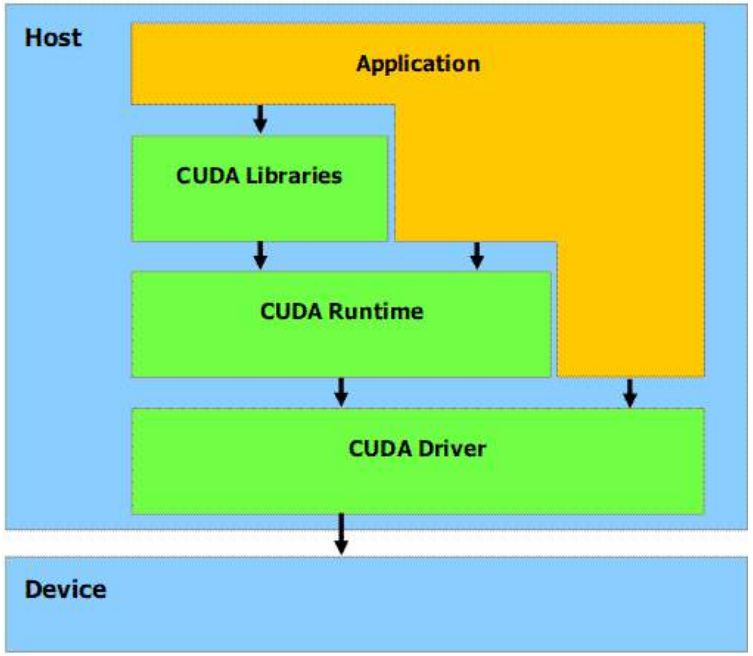
```
__global__ void reverse(double *d_a) {
  __shared__ double s_a[256]; //array of doubles, shared in this block

  int tid = threadIdx.x;
  s_a[tid] = d_a[tid];    //each thread fills one entry

  //all wavefronts must reach this point before any wavefront is allowed to continue.
                          //something is missing here…
__syncthreads();

  d_a[tid] = s_a[255-tid]; //write out array in reverse order
}

int main() {
  …
  hipLaunchKernelGGL(reverse, dim3(1), dim3(256), 0, 0, d_a); //Launch kernel
  …
}
```

# Part-IV: GPU System

# CUDA

- During regular execution, a CUDA application process will be launched by the user
- The application communicates directly with the CUDA user-mode driver, and potentially with the CUDA runtime library



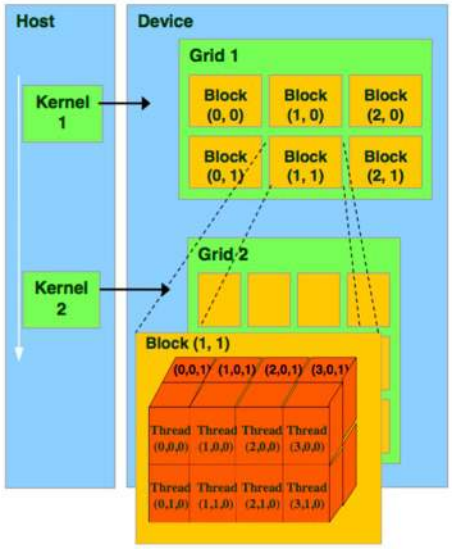






https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html

# 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]

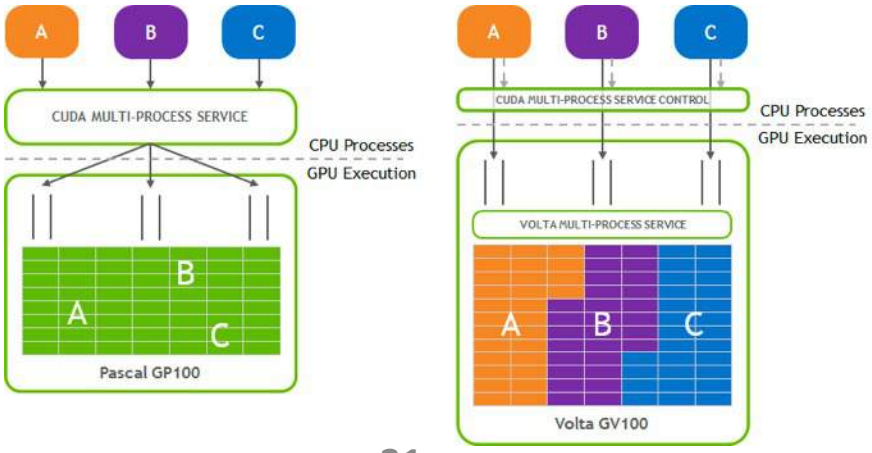
- A GPU program starts by creating a **context**
  - Either explicitly using the driver API or implicitly using the runtime API, for a specific GPU
- The **context** encapsulates all the hardware resources necessary for the program to be able to manage memory and launch work on that GPU

- 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

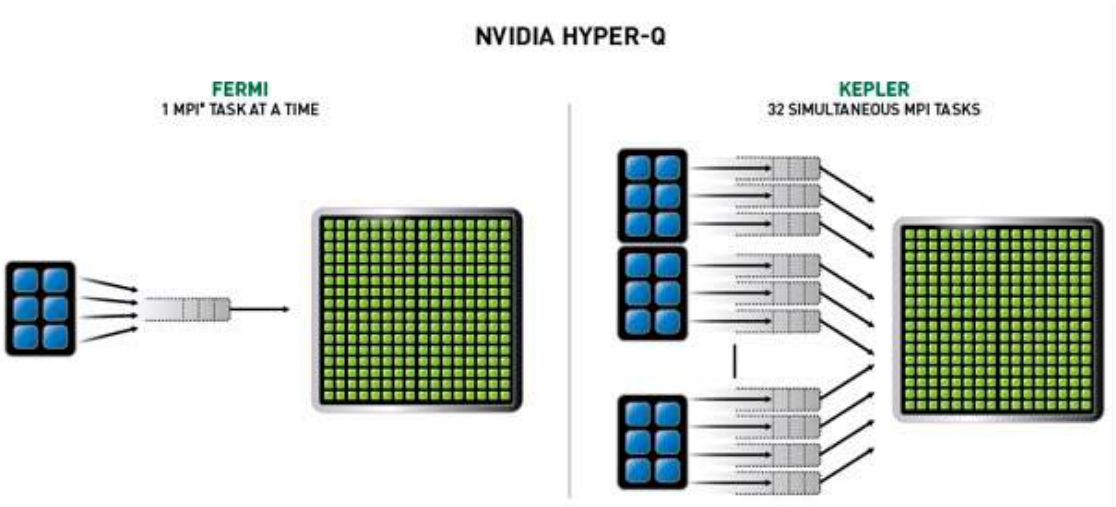





https://on-demand.gputechconf.com/gtc/2015/presentation/S5584-Priyanka-Sah.pdf

# 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 ) ;
…
```

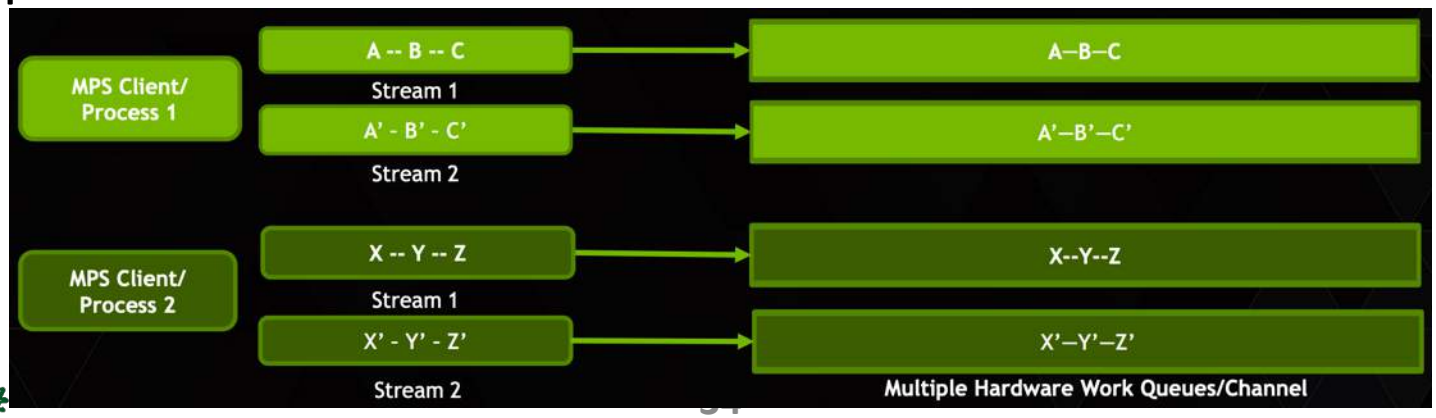
Completely synchronous

```
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

# 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

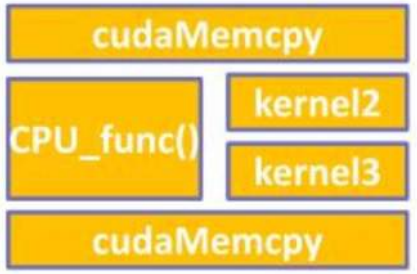


# Synchronous/Asynchronous[同步/异步]

- 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);
```

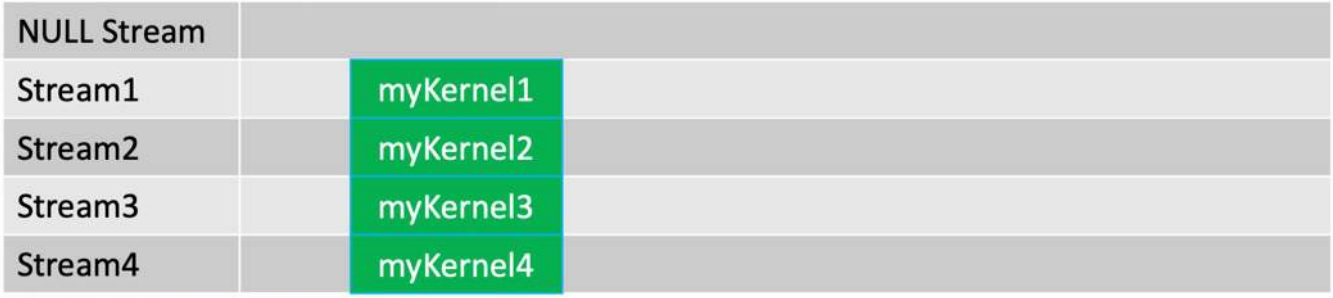



https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf

# 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);
```



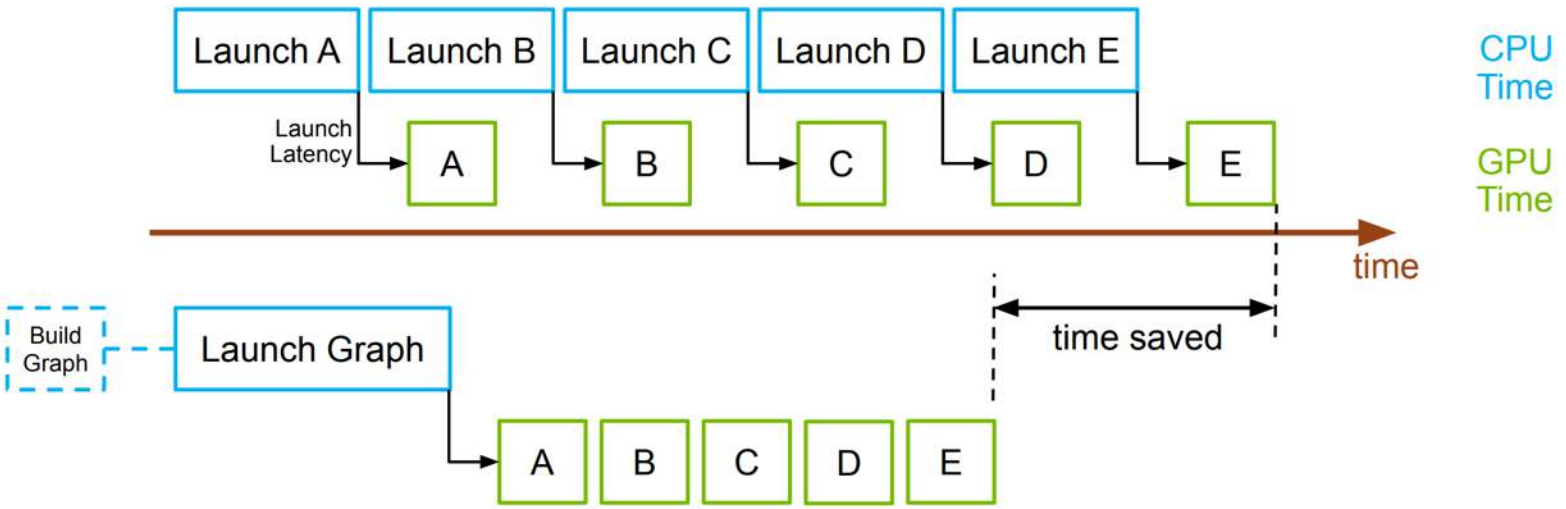
# Example

- *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



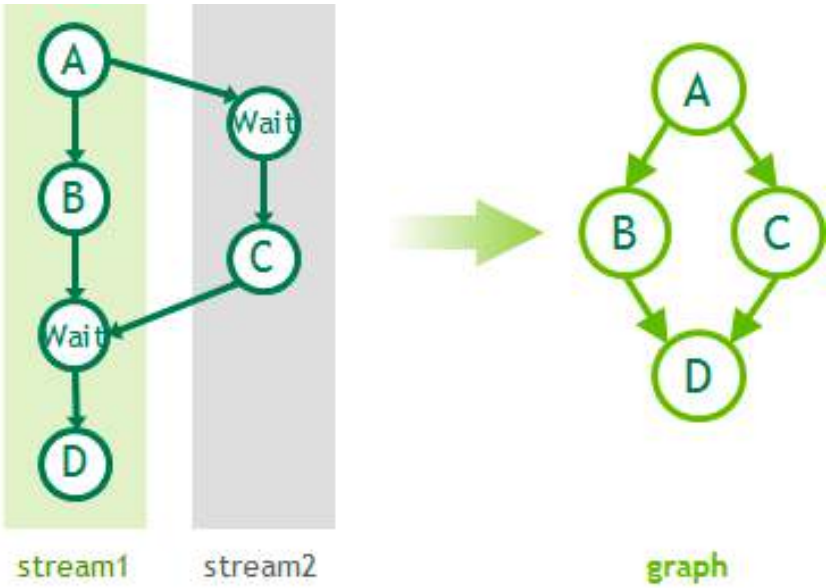



https://www.olcf.ornl.gov/wp-content/uploads/2021/10/013_CUDA_Graphs.pdf

# 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);
```

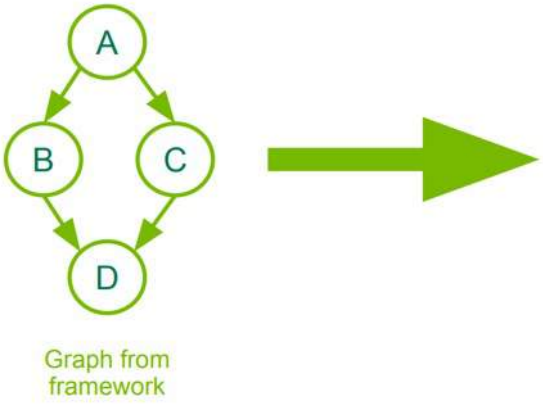





https://www.olcf.ornl.gov/wp-content/uploads/2021/10/013_CUDA_Graphs.pdf

# Example (cont.)

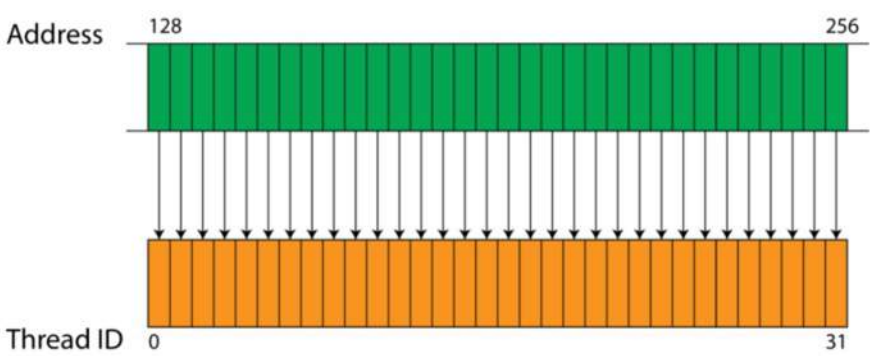
- Create graphs directly[直接构建]
  - Map graph-based workflows directly into CUDA

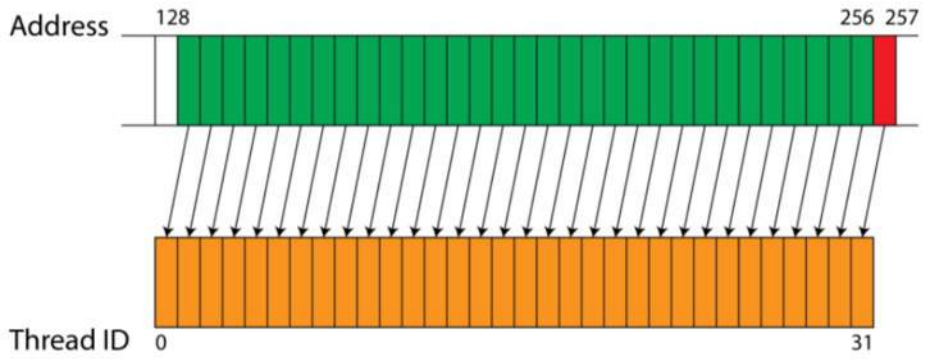



```
// Define graph of work + dependencies
cudaGraphCreate(&graph);

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);
```

# Address Coalescing[地址合并]

- Threads in a block are computed a warp at a time (32 threads)
- Global data is read or written in as few transactions as possible by combining memory access requests into a single transaction
  - This is referred to the device coalescing mem stores and reads
- Every successive 128 bytes can be accessed by a warp (or 32 single precision words)
- Not in successive 128 bytes; more data to read



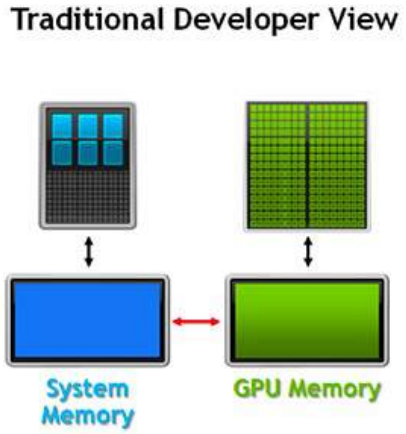



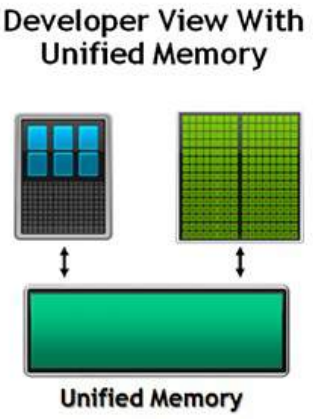



https://www.ce.jhu.edu/dalrymple/classes/602/Class13.pdf

# Unified Memory[统一内存]

- Classical model[经典模型]
  - Allocate memory on host
  - Allocate memory on device
  - Copy data from host to device
  - Operate on the GPU data
  - Copy data back to host
- Unified memory model[统一模型]
  - Allocate memory
  - Operate on data on GPU
- Unified Memory is a single memory address space accessible from any processor in a system
  - cudaMalloc() → cudaMallocManaged()
  - on-demand page migration





https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

# Example

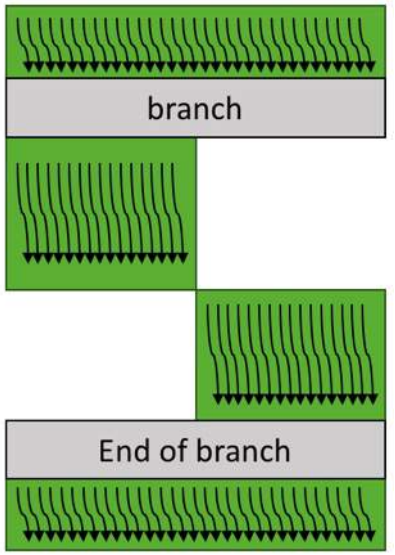
```
int N = 1<<20;
float *x, *y;

// Allocate Unified Memory -- accessible from CPU or GPU
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));

// initialize x and y arrays on the host
for (int i = 0; i < N; i++) {
  x[i] = 1.0f;
  y[i] = 2.0f;
}

// Launch kernel on 1M elements on the GPU
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

# Divergence[分支]

- Within a block of threads, the threads are executes in groups of 32 called a warp
  - All threads in a warp do the same thing at the same time
- What happens if different threads in a warp need to do different things?
  - A logical predicate and two predicated instructions → serialized
- Branch divergence is a major cause for performance degr adation in GPGPUs

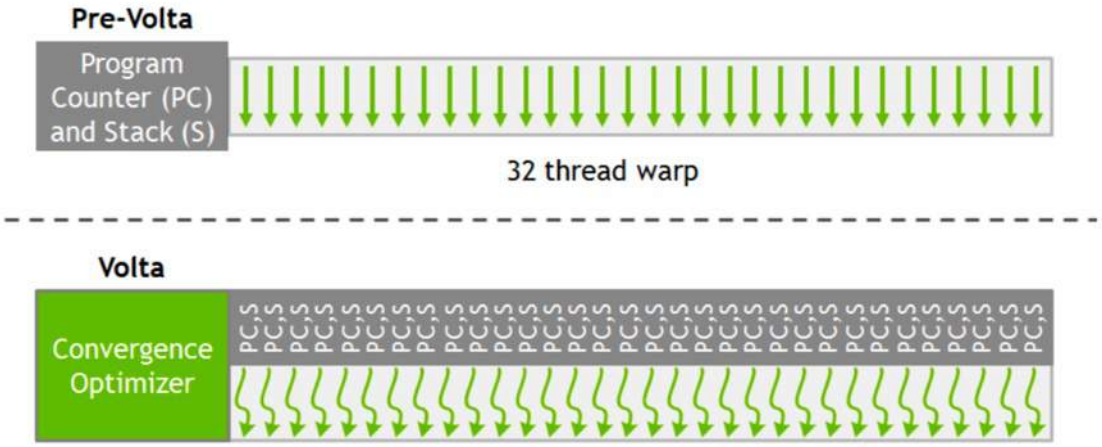
```
...
if ( threadIdx.x < 16 )
{
    ... A ...
}
else
{
    ... B ...
}
...
```

p = (threadIdx.x < 16);
if (p) … A …
if (!p) … B …



# Divergence (cont.)

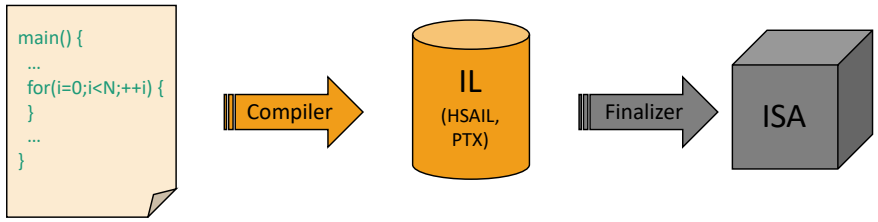
- Pre-Volta GPUs use a single PC shared amongst all 32 threads of a warp, combined with an active mask that specifies which threads of the warp are active at any given time
  - Leaves threads that are not executing a branch inactive
- Since Volta, each thread features its own PC, which allows threads of the same warp to execute different branches of a divergent section simultaneously

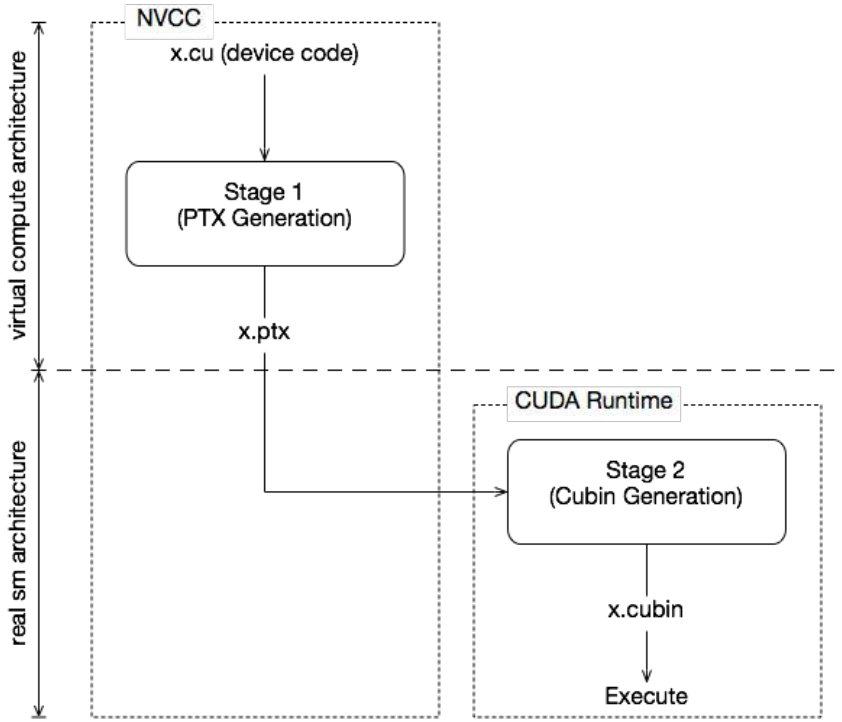



https://www.icl.utk.edu/files/publications/2018/icl-utk-1080-2018.pdf

# Two-phase Execution[两段式]

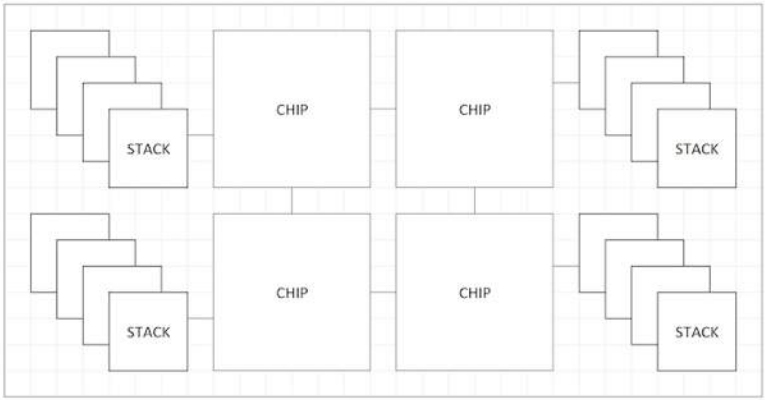
- Compilation workflow
  - Source code → virtual instruction (PTX or HSAIL)
  - Virtual inst → real inst (SASS or GCN)
- **.cu**: CUDA source file, containing host code and device functions
- **.ptx**: PTX intermediate assembly file
- **.cubin**: CUDA device code binary file (CUBIN) for a single GPU architecture

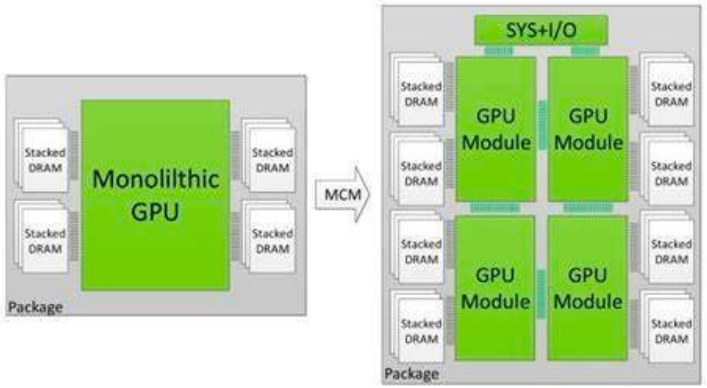


# Multi-chip Module

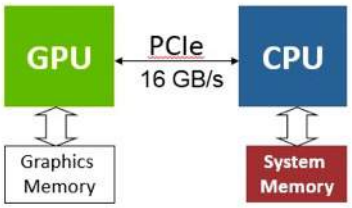
- Aggregating multiple GPU modules within a single package, as opposed to a single monolithic die.
- AMD: Chiplet GPUs
  - MI200: 220 compute units, 14K streaming cores
  - MI100: 120 compute units, 7680 streaming cores
- Nvidia: Multi-Chip-Module (MCM) GPUs
  - Hopper (Ampere -> Lovelace): 300+ SMs, 40K+ CUDA cores
  - A100: 128 SMs, 8192 CUDA cores



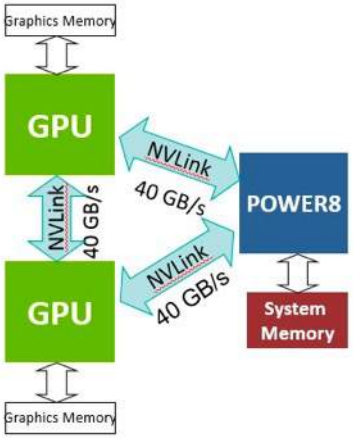



# High-speed Links[高速连接]

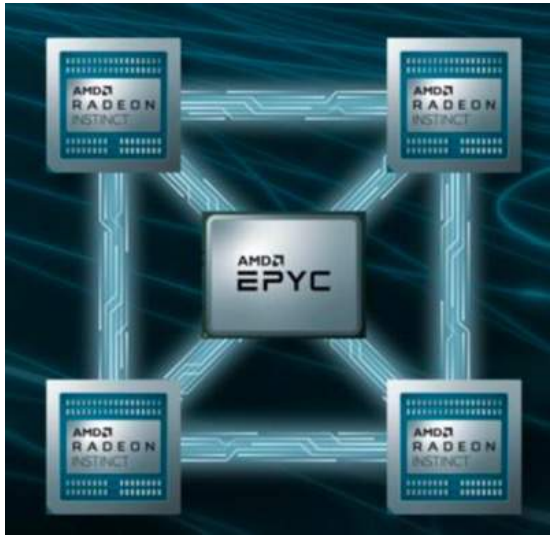
- GPUs are of high compute capability, being bottlenecked on data movement
- High-speed interconnect to achieve significantly higher data movement
  - Nvidia: NVLink
  - AMD: Infinity Fabric
  - Intel: Compute eXpress Link (CXL)




CPU-GPU Systems Connected via PCI-e

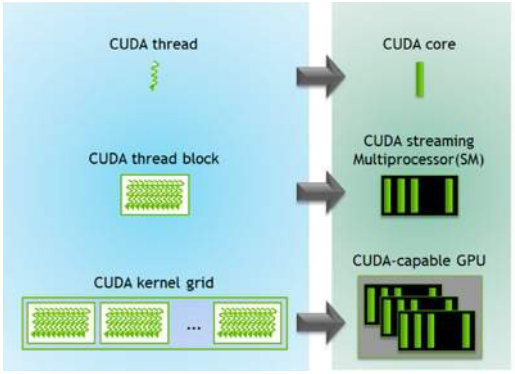



NVLink Enables Fast Unified Memory Access between CPU & GPU Memories



# Summary of DLP/GPU[总结]

- Data level parallelism
  - SIMD: operates on multiple data with on single instruction
    - AVX-512 on Intel CPU is the typical example
  - SIMT: consists of multiple scalar threads executing in a SIMD manner
    - GPU is the example with threads executing the same instruction
- GPU hardware and thread organization
  - Device → SM → SIMD/Partition → Core
  - Grid → Block → Warp → Thread




- GPU programming
  - Streams to support concurrency
  - Memory hierarchy and usage (thread, cache/smem, global)
  - Advanced topics: virtualization, profiling/tuning, divergence, etc