中山大學 SUN YAT-SEN UNIVERSITY

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

Computer Architecture
计 算 机 体 系 结 构

第9讲：DLP & GPU（3）

张献伟

xianweiz.github.io

DCS3013, 11/2/2022

Review Questions

- 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

- GPU is of high latency tolerance?

  Massive threads to schedule and work on

- Explain SM or CU.

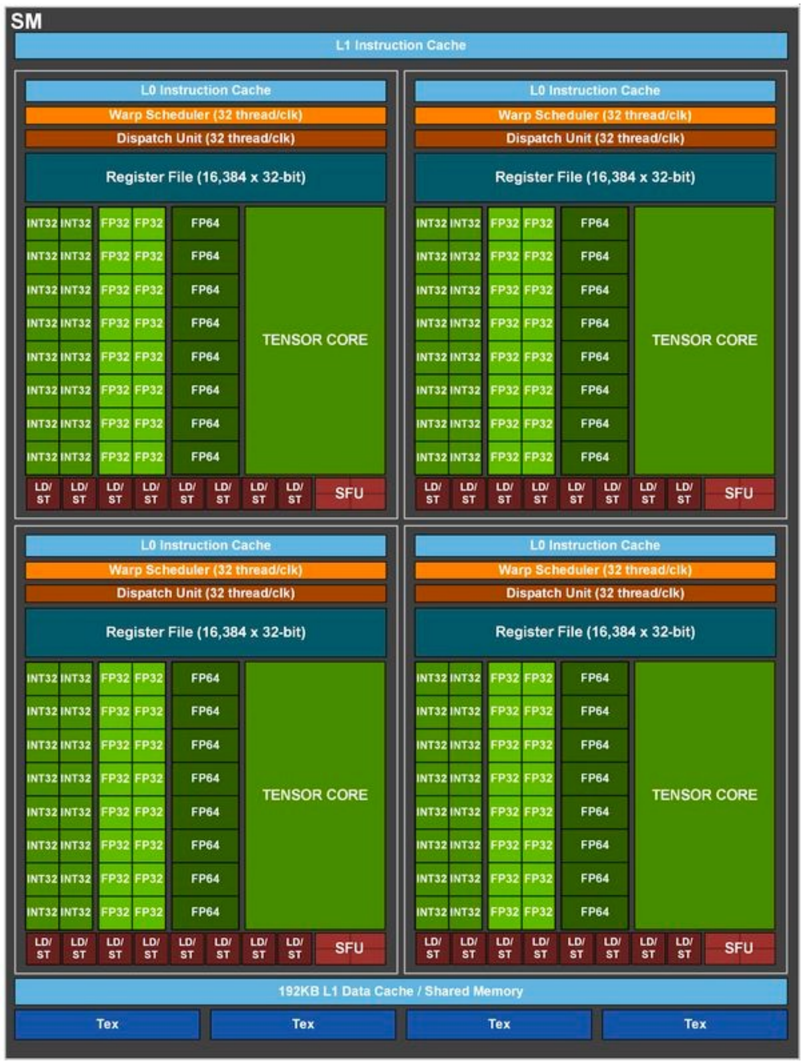
  The fundamental compute unit to execute GPU tasks, hosting multiple simple cores to run the threads

- Warp

  A group of 32 threads executing in lockstep.

TFLOPS[衡量算力]

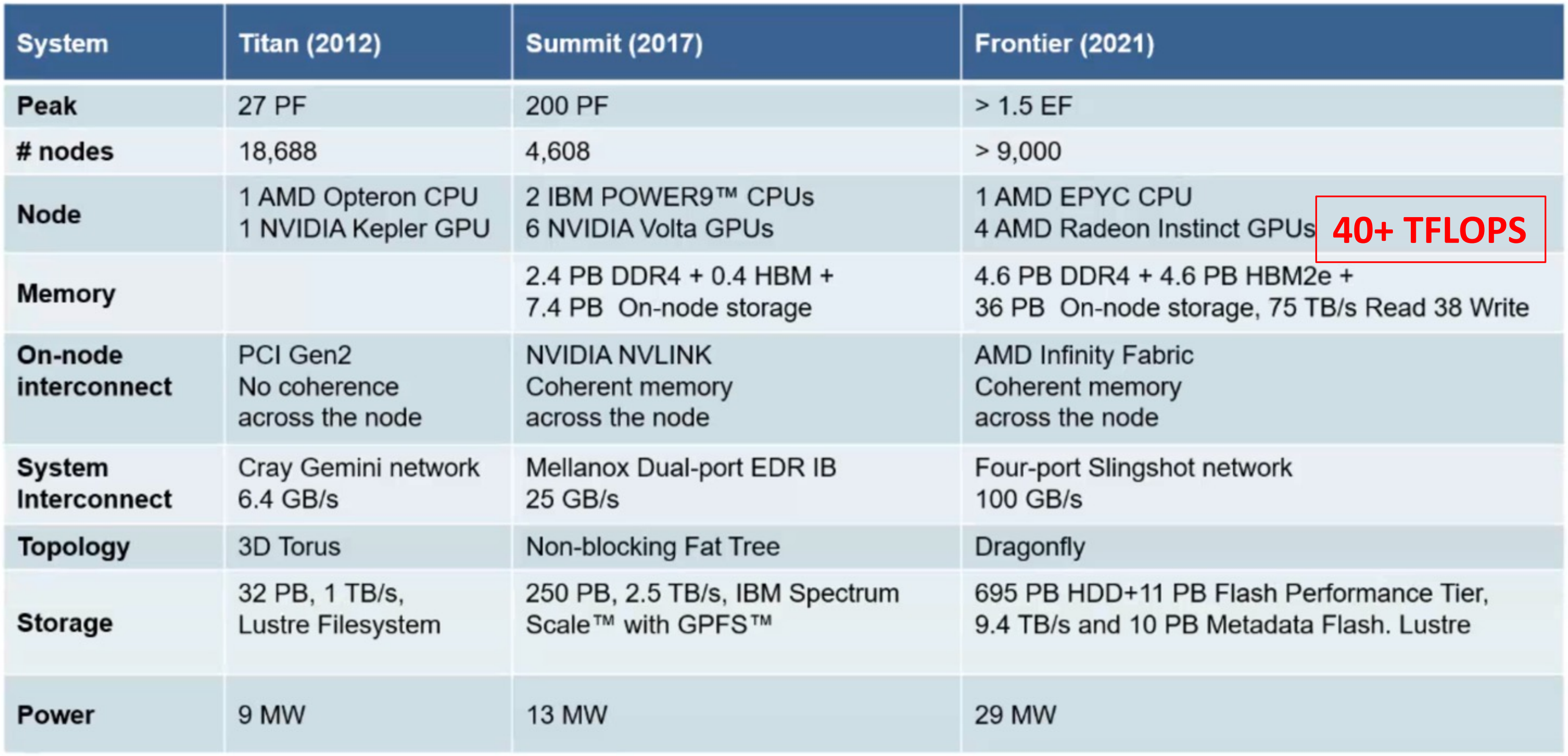
- A100 Tensor Core GPU
  - 108 SMs
    - GA100 Full GPU with 128 SMs
  - Base clock: 1065 MHz
  - Boost clock: 1410 MHz
  - Performance
    - FP64: 9.7 TFLOPS
    - FP32: 19.5 TFLOPS


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

- Calculate TFLOPS
  - FP64: 1410 MHz x (32 x 2) ops/clock x 108 SMs

GPUs in Supercomputer[超算中的GPU]

- Exascale: 50 GFLOPS/Watt (goal) → **51.7** GFLOPS/Watt



System	Titan (2012)	Summit (2017)	Frontier (2021)
Peak	27 PF	200 PF	> 1.5 EF
# nodes	18,688	4,608	> 9,000
Node	1 AMD Opteron CPU 1 NVIDIA Kepler GPU	2 IBM POWER9™ CPUs 6 NVIDIA Volta GPUs	1 AMD EPYC CPU 4 AMD Radeon Instinct GPUs
Memory		2.4 PB DDR4 + 0.4 HBM + 7.4 PB On-node storage	4.6 PB DDR4 + 4.6 PB HBM2e + 36 PB On-node storage, 75 TB/s Read 38 Write
On-node interconnect	PCI Gen2 No coherence across the node	NVIDIA NVLINK Coherent memory across the node	AMD Infinity Fabric Coherent memory across the node
System Interconnect	Cray Gemini network 6.4 GB/s	Mellanox Dual-port EDR IB 25 GB/s	Four-port Slingshot network 100 GB/s
Topology	3D Torus	Non-blocking Fat Tree	Dragonfly
Storage	32 PB, 1 TB/s, Lustre Filesystem	250 PB, 2.5 TB/s, IBM Spectrum Scale™ with GPFS™	695 PB HDD+11 PB Flash Performance Tier, 9.4 TB/s and 10 PB Metadata Flash. Lustre
Power	9 MW	13 MW	29 MW

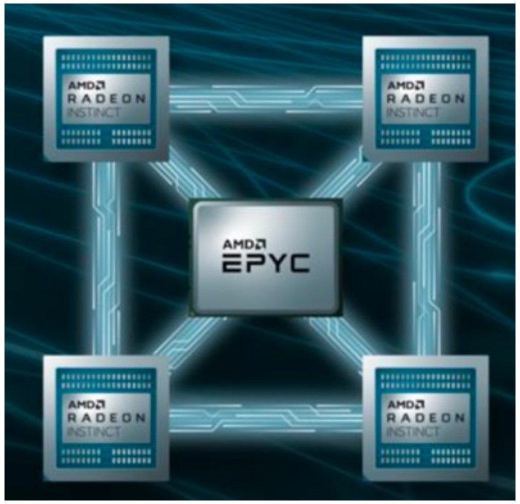
40+ TFLOPS

OAK RIDGE National Laboratory | OAK RIDGE LEADERSHIP COMPUTING FACILITY

https://www.hpcwire.com/2021/07/14/frontier-to-meet-20mw-exascale-power-target-set-by-darpa-in-2008/

Frontier: 1.5 EFLOPS, How???[E级超算]

- Per node[单节点]
  - Custom EPYC HPC-optimized CPU
    - ”zen 3” milan w/ 64-core
  - Four Instinct GPUs
    - CDNA MI200 w/ *256* CUs
      - Full-rate FP64 (128 ops/clock/CU)



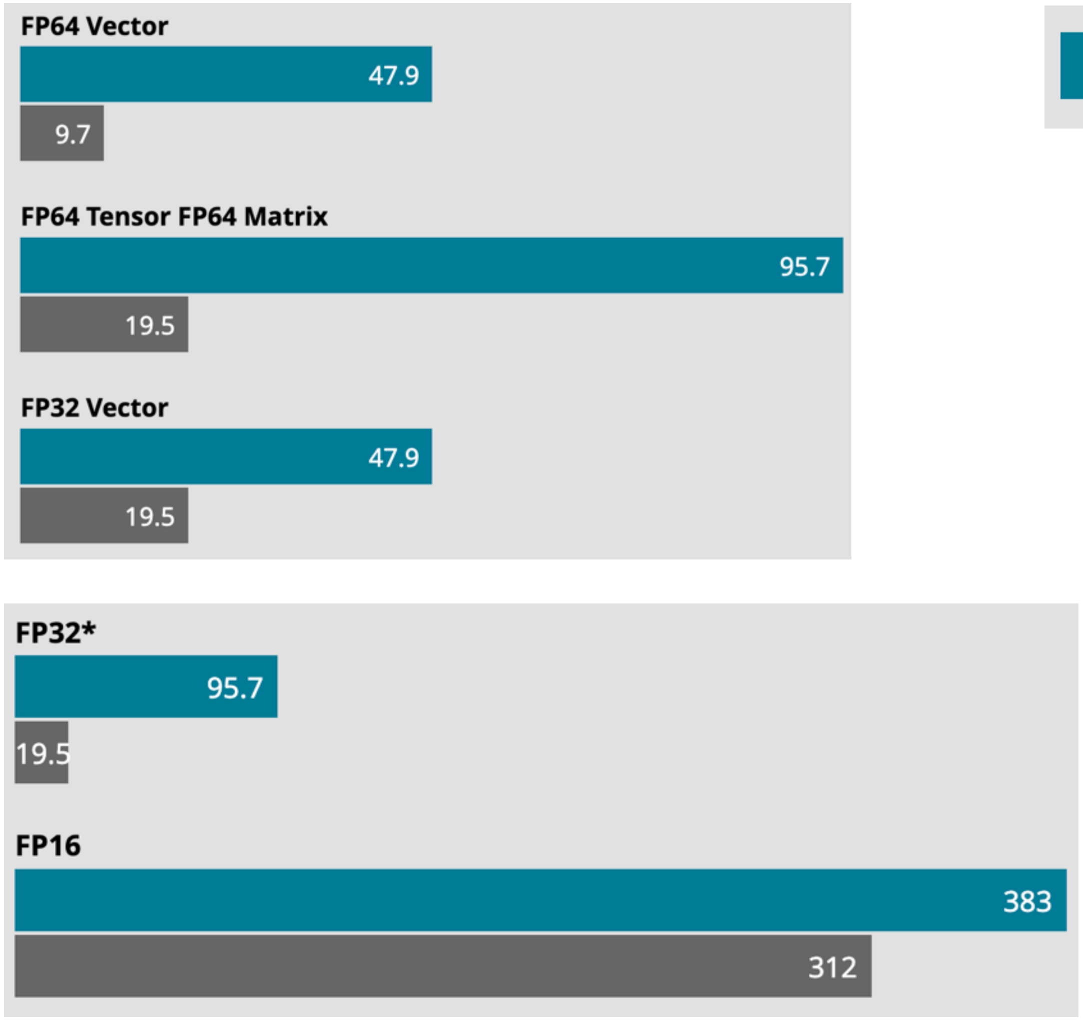
- 9000+ nodes[整体系统]
  - CPU: 9000 x 4 TFLOPS/CPU = 36 PFLOPS
  - GPU: 9000 x 4 x 42.2 TFLOPS/GPU = 1519 PFLOPS
    - Per GPU: 128 ops/clock x 1.5G x 220 = 42.2 TFLOPS
  - GPU provides **97.7%** computation power
    - 1519/(1519+36)

A100: 9.75 TFLOPS
MI100: 11.54 TFLOPS

OLCF spock training: AMD hardware and software, 05/2021, https://www.olcf.ornl.gov/wp-content/uploads/2021/04/Spock-MI100-Update-5.20.21.pdf

https://www.hpcwire.com/2021/03/15/amd-launches-epyc-milan-with-19-skus-for-hpc-enterprise-and-hyperscale/

MI200 GPU



MI200
A100 80GB

https://www.amd.com/en/press-releases/2022-05-26-amd-instinct-mi200-adopted-for-large-scale-ai-training-microsoft-azure

中山大學 SUN YAT-SEN UNIVERSITY

天河超算

- 2009，天河-1
  - CPU + ATI GPU

    240 GFLOPS

    - 2 * Xeon E5540/E5450, 1 ATI Radeon HD 4870 X2 (TeraScale)
  - 实测/峰值563.1T/1206.2T FLOPS
  - 2009.11 TOP500第五
- 2010，天河-1A
  - CPU + Nvidia GPU

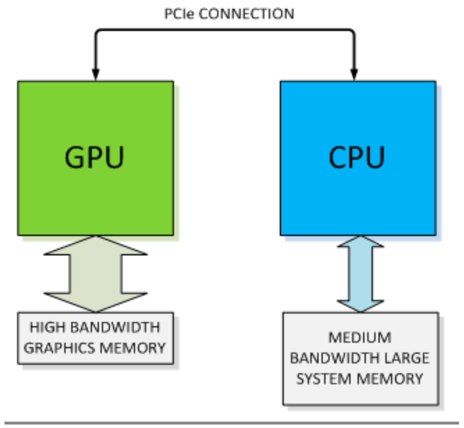
    515 GFLOPS

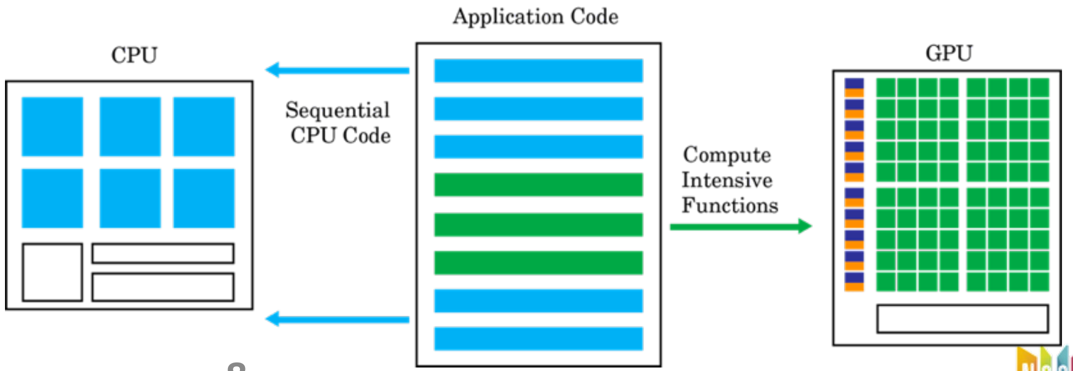
    - 2 * Intel Xeon X5670, 1 Nvidia Tesla M2050 (Fermi)
    - 2048 Galaxy "FT-1000" 1 GHz 8-core processors
  - 实测/峰值2.566P/4.7P FLOPS
  - 2010.11 TOP500第一

Tianhe-1, https://www.top500.org/system/176546/
Tianhe-1A, https://top500.org/system/176929/
Tianhe-1A, http://blog.zorinaq.com/introducing-tianhe-1a-4702-tflops-of-gpu-power-made-in-china-and/

GPU Programming Model[编程模型]

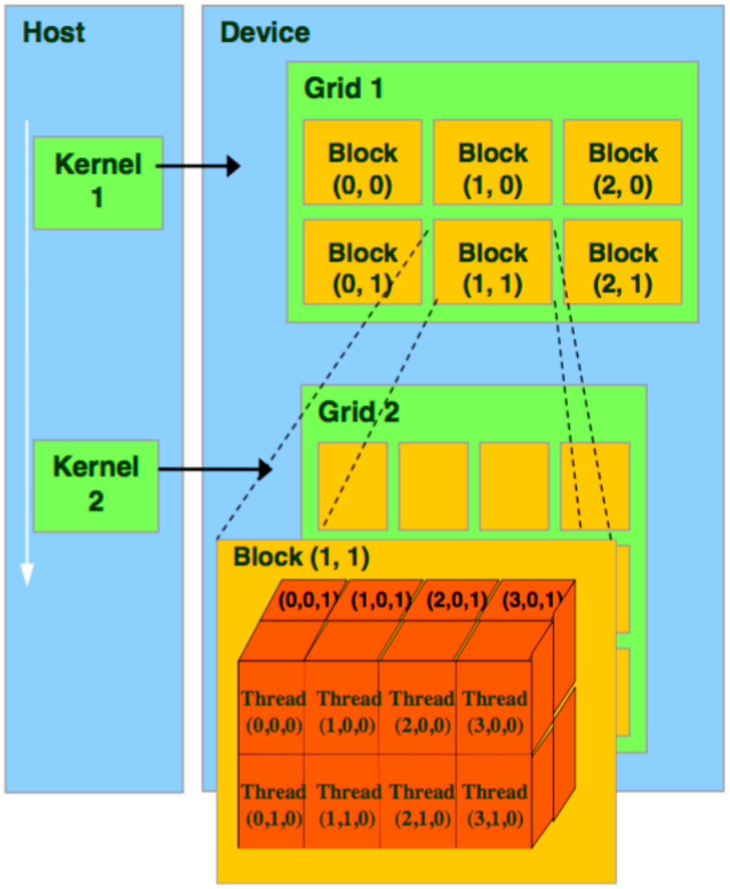
- GPU is viewed as a compute device that[设备]
  - Is a coprocessor to CPU (host)[协处理器]
  - Has its own main memory called device memory[显存]
  - Runs many threads in parallel[线程并发]
- Data-parallel parts of an application are executed on the device as **kernels**, which run in parallel on many threads
-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


PCIe CONNECTION
GPU
CPU
HIGH BANDWIDTH GRAPHICS MEMORY
MEDIUM BANDWIDTH LARGE SYSTEM MEMORY



Thread Organization[线程组织]

- A kernel is executed as a grid of thread blocks[网格]
- A thread block is a batch of threads that can cooperate with each other by[块]
  - 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

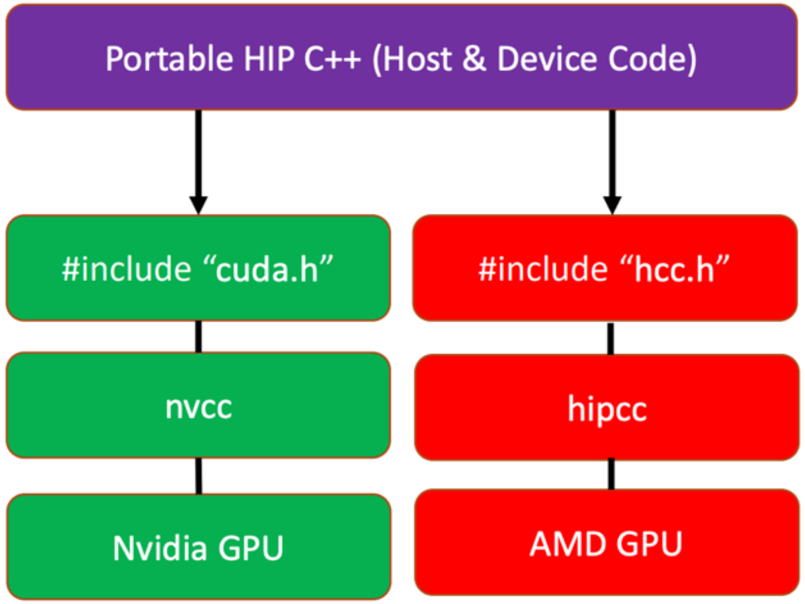


GPU Programming Choices[编程选择]

- **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



HIP vs. CUDA

- Kernel declare
  - Syntactically the same
- APIs

```
cudaMalloc(&d_x, N*sizeof(double));

cudaMemcpy(d_x, x, N*sizeof(double),
        cudaMemcpyHostToDevice);

cudaDeviceSynchronize();
```

```
hipMalloc(&d_x, N*sizeof(double));

hipMemcpy(d_x, x, N*sizeof(double),
        hipMemcpyHostToDevice);

hipDeviceSynchronize();
```

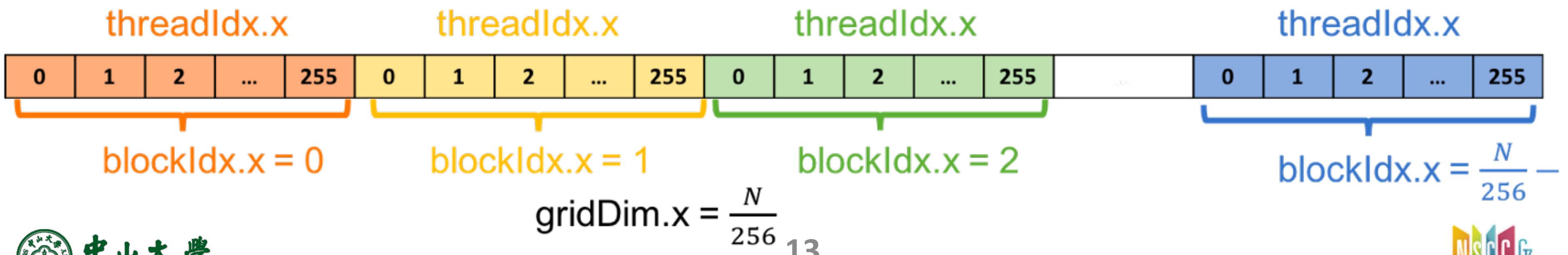
- Kernel launch

```
some_kernel<<<gridsize, blocksize,
        shared_mem_size, stream>>>
        (arg0, arg1, ...);
```

```
hipLaunchKernelGGL(some_kernel,
        gridsize, blocksize,
        shared_mem_size, stream,
        arg0, arg1, ...);
```

Kernel Dimensions[维度]

- Built-in variables
  - *blockDim.x*: the size of the block (#threads in the block)
  - *gridDim.x*: the size of the grid (#blocks)
  - *blockIdx.x*: the index of the block within the grid
  - *threadIdx.x*: the index of the thread within the block
- Example: *N* threads in total, 256 threads per block
  - blockDim.x = 256
  - #blocks = N / 256 → gridDim.x
  - blockIdx.x = [0, 1, …, N/256-1]
  - threadIdx.x = [0, 1, …, 255]



Example: Kernel Declare[声明]

- A kernel is declared with the __global__ attribute
  - Kernels should be declared void
  - *All pointers passed to kernels must point to device memory*
- All threads execute the kernel’s body “simultaneously”
  - Each thread uses its unique thread and block IDs to compute a global ID

```
for (int i=0;i<N;i++) {
  h_a[i] *= 2.0;
}
```

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

Example: Kernel Launch[启动]

- Kernels are launched from host

```
dim3 threads(256,1,1);              //3D dimensions of a block of threads
dim3 blocks((N+256-1)/256,1,1);     //3D dimensions the grid of blocks

hipLaunchKernelGGL(myKernel,        //Kernel name (__global__ void function)
                   blocks,          //Grid dimensions
                   threads,         //Block dimensions
                   0,               //Bytes of dynamic LDS space (see extra slides)
                   0,               //Stream (0=NULL stream)
                   N, a);           //Kernel arguments
```

- Analogous to CUDA kernel launch syntax:

```
myKernel<<<blocks, threads, 0, 0>>>(N,a);
```

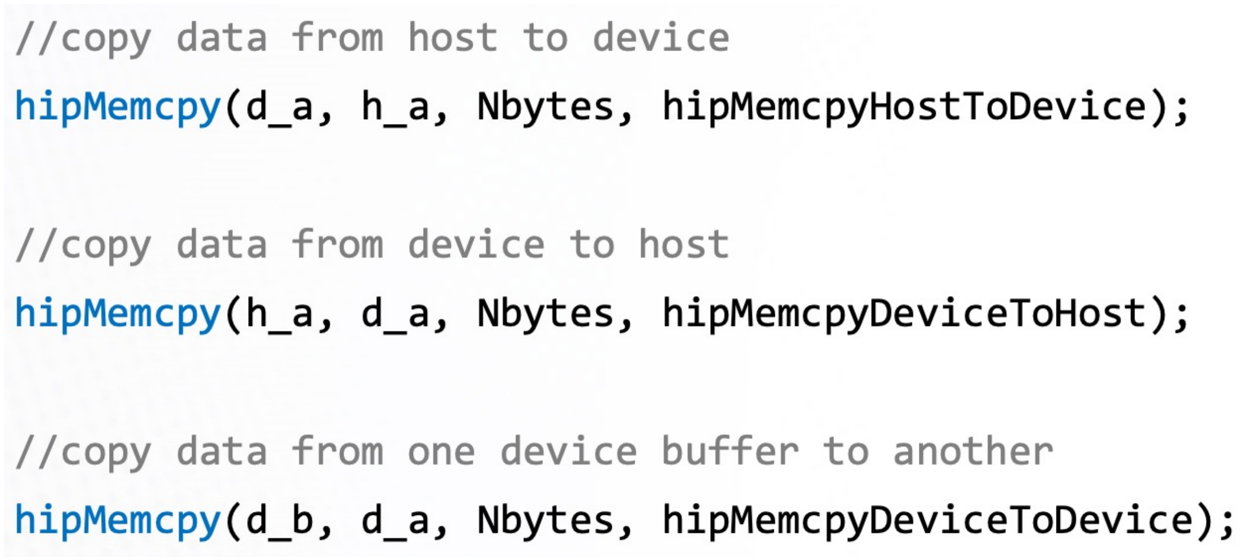
Example: Memory Allocation[内存分配]

- The host instructs the device to allocate memory and records a pointer to device memory

```
int main() {
  …
  int N = 1000;
  size_t Nbytes = N*sizeof(double);
  double *h_a = (double*) malloc(Nbytes);              //Host memory

  double *d_a = NULL;
  hipMalloc(&d_a, Nbytes);                              //Allocate Nbytes on device


  …


  free(h_a);                                            //free host memory
  hipFree(d_a);                                         //free device memory
}
```

Example: Memory Copy[数据传输]

- The host queues memory transfers
  - hipMemcpyHostToDevice
  - hipMemcpyDeviceToHost
  - hipMemcpyDeviceToDevice



```
//copy data from host to device
hipMemcpy(d_a, h_a, Nbytes, hipMemcpyHostToDevice);

//copy data from device to host
hipMemcpy(h_a, d_a, Nbytes, hipMemcpyDeviceToHost);

//copy data from one device buffer to another
hipMemcpy(d_b, d_a, Nbytes, hipMemcpyDeviceToDevice);
```

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(); } }
```

Device Management[管理]

- Host can query *number* of devices visible to system:

```
int numDevices = 0;
hipGetDeviceCount(&numDevices);
```

- Host tells the runtime to issue instructions to a *particular* device:

```
int deviceID = 0;
hipSetDevice(deviceID);
```

- Host can query what device is currently *selected*:

```
hipGetDevice(&deviceID);
```

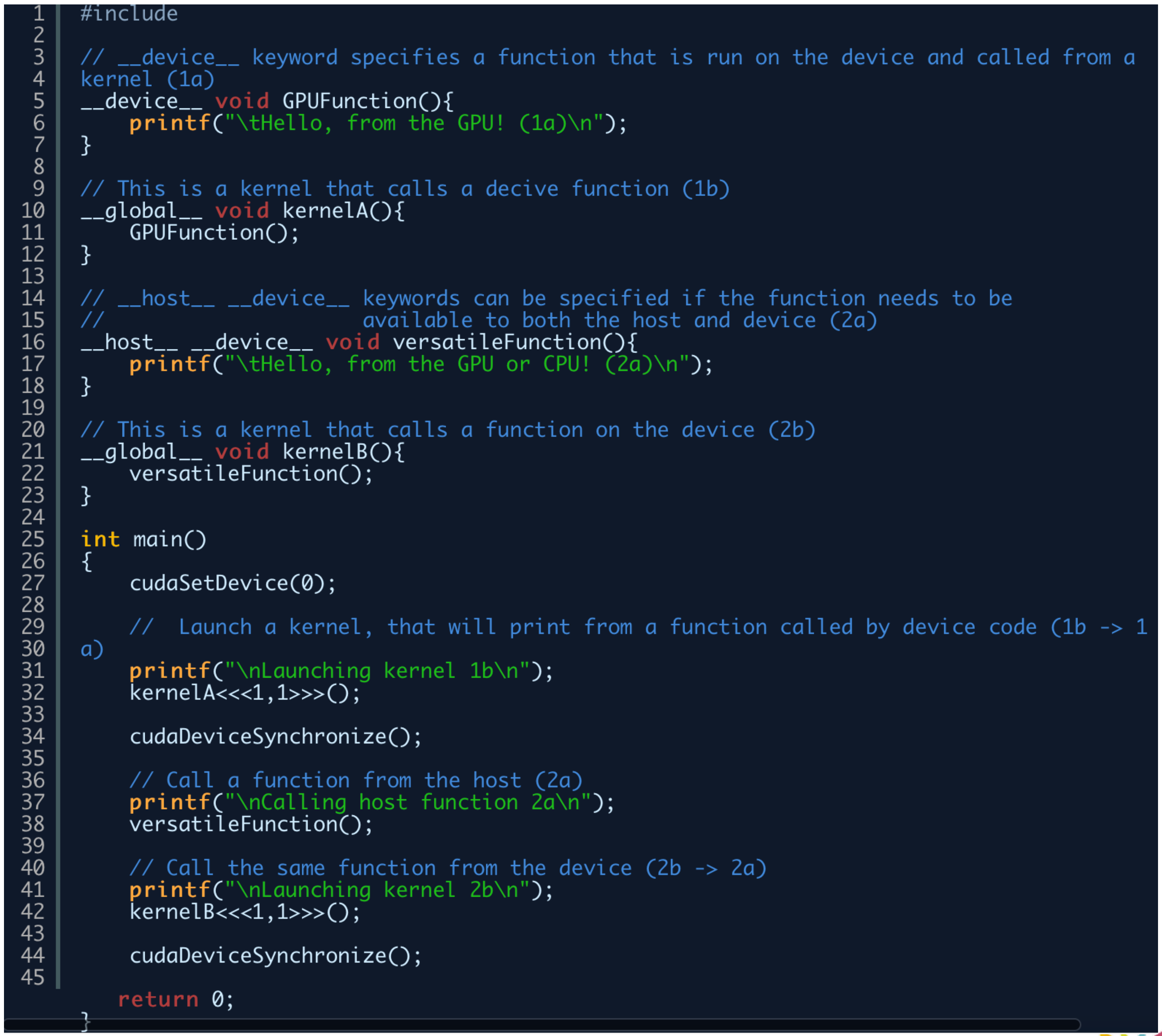
- The host can also query a device's *properties*:

```
hipDeviceProp_t props;
hipGetDeviceProperties(&props, deviceID);
```

hipDeviceProp_t is a struct that contains useful fields like the device's name, total VRAM, clock speed, and GCN architecture.

Specifiers

- __global__
- __device__

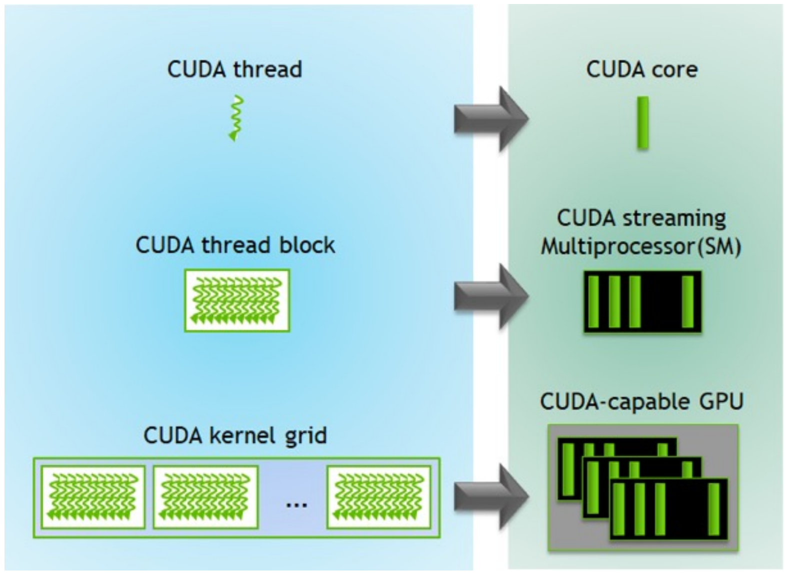


```
#include

// __device__ keyword specifies a function that is run on the device and called from a
kernel (1a)
__device__ void GPUFunction(){
    printf("\tHello, from the GPU! (1a)\n");
}

// This is a kernel that calls a decive function (1b)
__global__ void kernelA(){
    GPUFunction();
}

// __host__ __device__ keywords can be specified if the function needs to be
//                      available to both the host and device (2a)
__host__ __device__ void versatileFunction(){
    printf("\tHello, from the GPU or CPU! (2a)\n");
}

// This is a kernel that calls a function on the device (2b)
__global__ void kernelB(){
    versatileFunction();
}

int main()
{
    cudaSetDevice(0);

    //  Launch a kernel, that will print from a function called by device code (1b -> 1
a)
    printf("\nLaunching kernel 1b\n");
    kernelA<<<1,1>>>();

    cudaDeviceSynchronize();

    // Call a function from the host (2a)
    printf("\nCalling host function 2a\n");
    versatileFunction();

    // Call the same function from the device (2b -> 2a)
    printf("\nLaunching kernel 2b\n");
    kernelB<<<1,1>>>();

    cudaDeviceSynchronize();

   return 0;
}
```

https://codingbyexample.com/2018/12/14/launching-cuda-functions/

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 a.k.a., workgroup
  - Threads in block share LDS memory and L1 cache SMEM for Nvidia
- 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)

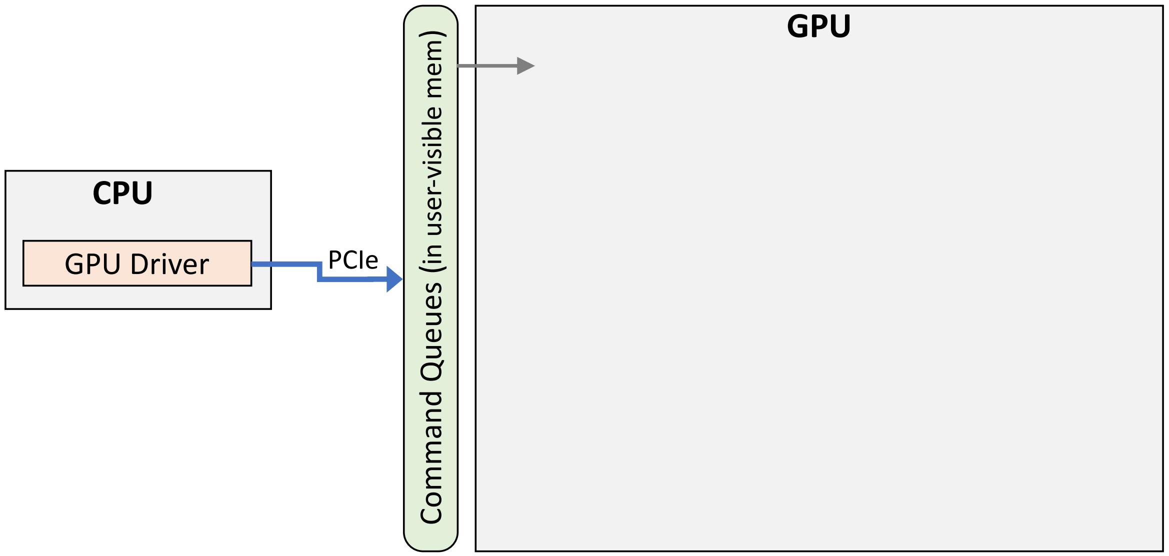
Compute unit

GPU

CPU-GPU

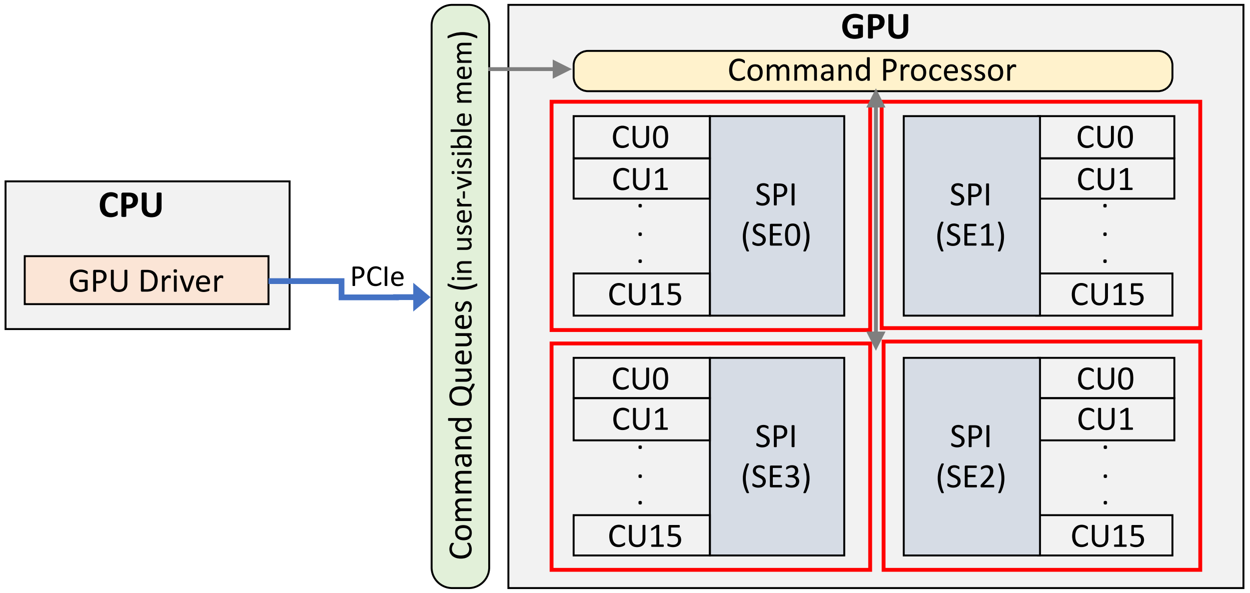
- CPU communicates kernels to GPUs via PCIe
  - Kernel code object is filled into a dispatch *packet*
  - Next, the packet is placed into a *queue*, which is allocated by runtime and associated with a GPU **stream for Nvidia**
  - The *GPU* is then signaled to process packets from the queue
  - When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

TIP:
Task
==
Command
==
Packet
==
Kernel



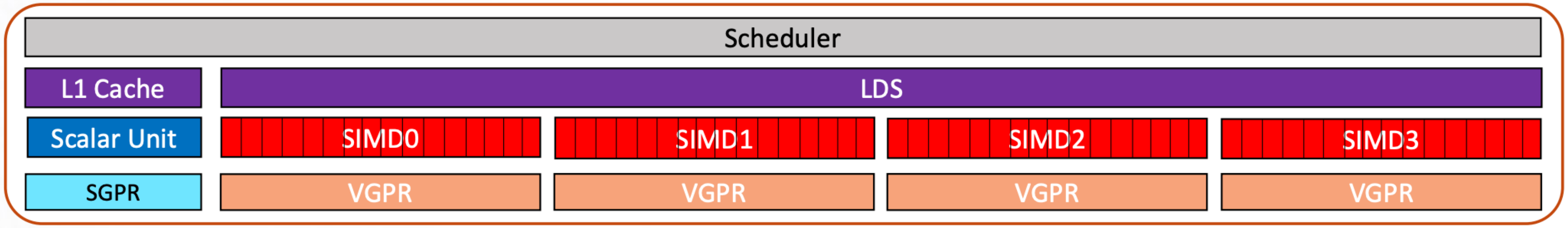
GPU Structure[内部架构]

- 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



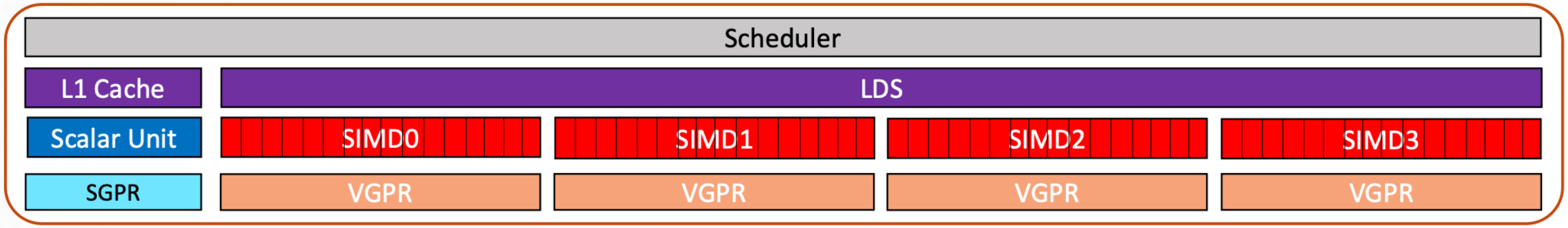
Compute Unit

- Scheduler[调度器]
  - 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.



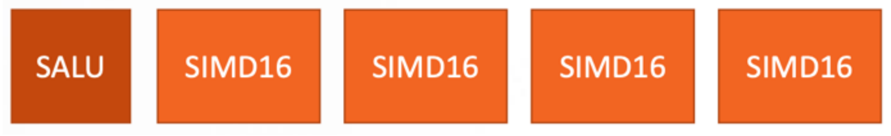
Compute Unit (cont.)

- 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

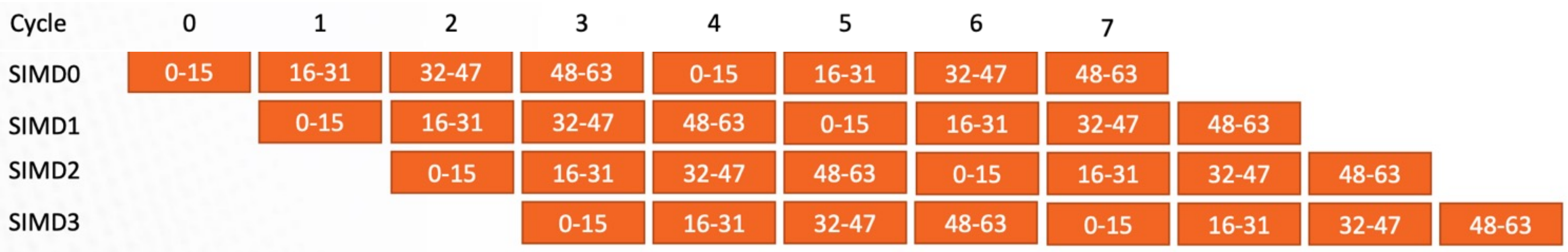


Compute Unit (cont.)

- At each clock, waves on *1 SIMD* unit are considered for execution (Round Robin scheduling among SIMDs)
- Each wave is assigned to one SIMD16, up to *10 waves* per SIMD16 (*math: 4 x 10 x 64 = 2560 threads*)
- Each SIMD16 issues 1 instruction every 4 cycles
- Vector instructions throughput is 1 every 4 cycles



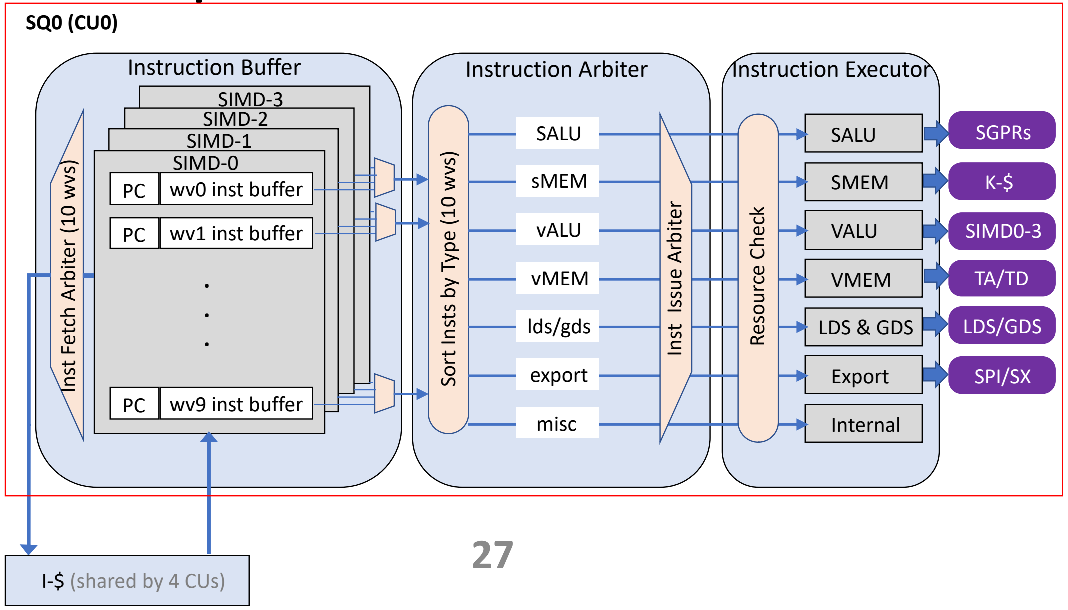
1 every cycle in Nvidia and AMD next generation



Cycle	0	1	2	3	4	5	6	7			
SIMD0	0-15	16-31	32-47	48-63	0-15	16-31	32-47	48-63			
SIMD1		0-15	16-31	32-47	48-63	0-15	16-31	32-47	48-63		
SIMD2			0-15	16-31	32-47	48-63	0-15	16-31	32-47	48-63	
SIMD3				0-15	16-31	32-47	48-63	0-15	16-31	32-47	48-63

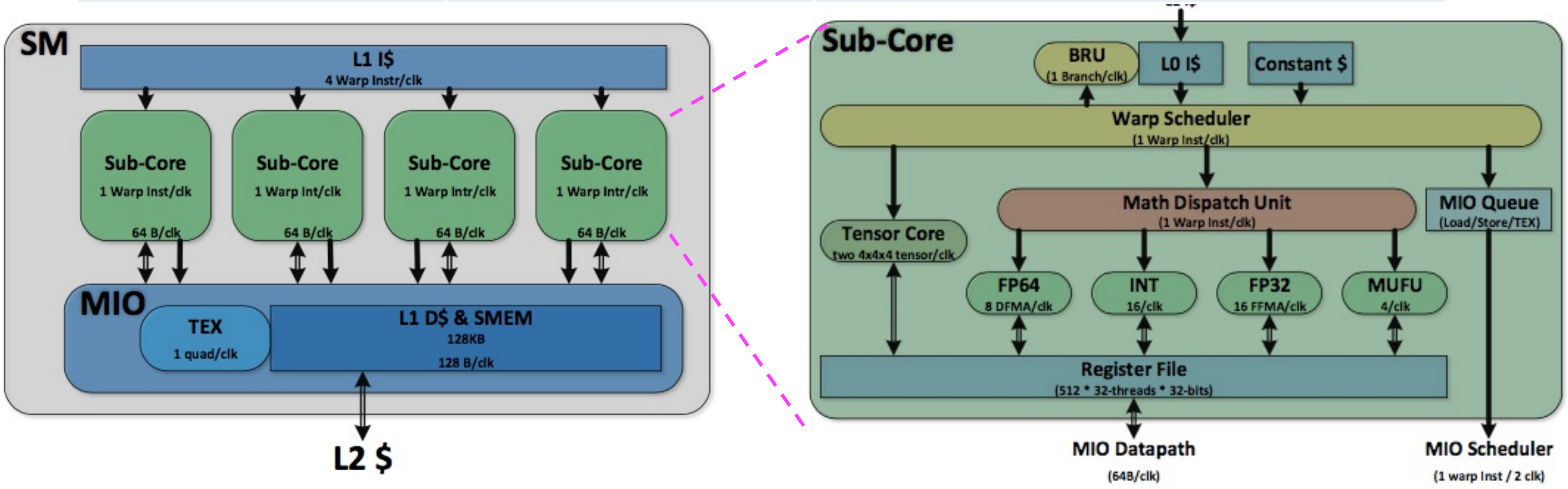
Instruction Execution[指令执行]

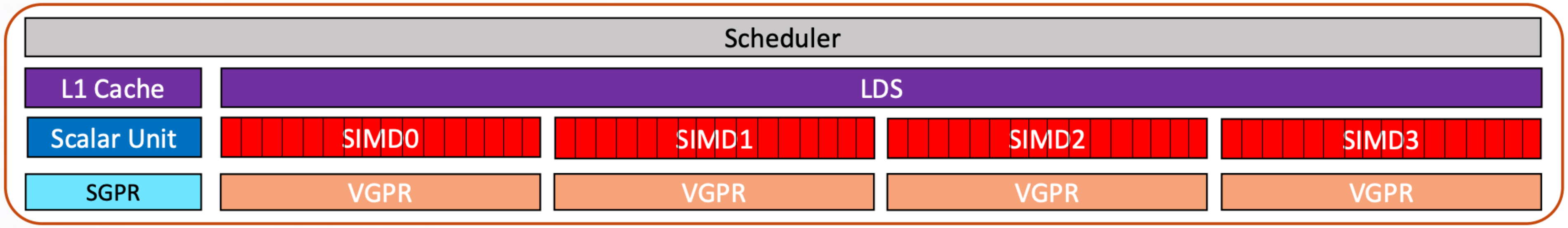
- **Instruction buffer** (IB): each cycle, the 10 wvs of the selected SIMD compete for instruction fetch (oldest wins)
- **Instruction arbiter** (IA): arbitrates multi wvs which want to execute the same type of instructions
- **Instruction executor** (IE): multiple execution units running in parallel; only one instruction of each type can be issued at a time per SIMD



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



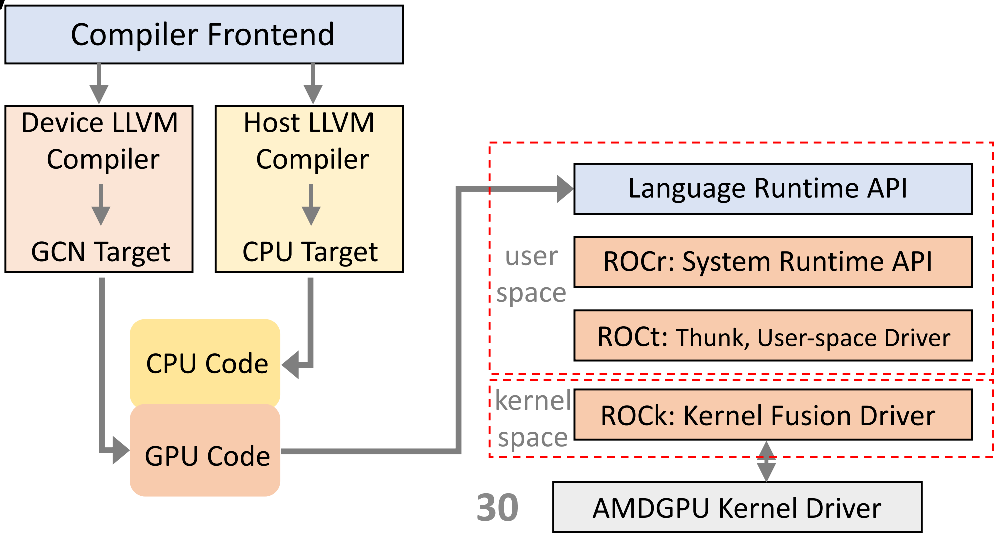


Terminology[术语]

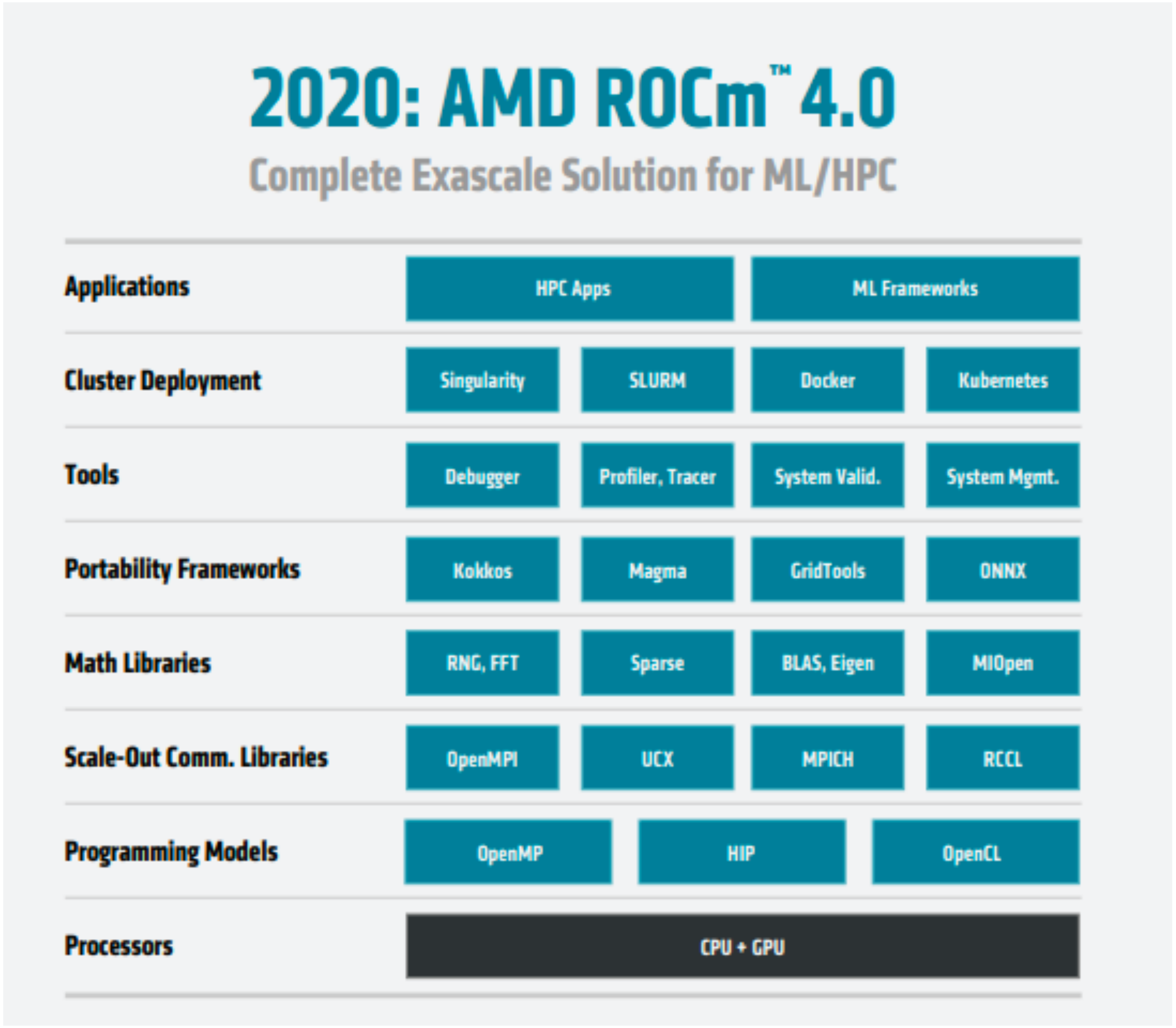
Nvidia	AMD	Note
Thread Block (TB) / Cooperative Thread Array (CTA)	Workgroup (WG)	Basic workload unit assigned to an SM or CU. Each kernel is split into multiple CTAs, and the #CTAs is controlled by the application. Typically, hw limits 1024 threads per block.
Warp	Wavefront (wave/WF/WV)	A group of threads (e.g., 32 for NV, 64 for AMD) executing in lockstep (i.e., run the same inst, follow the same control-flow path). #WFs/WG is chosen by developers.
Thread	Work-item(WI)/thread	A basic element to be processed.
GPU Processing Cluster (GPC)	Shader Engine (SE)	A collection of CUs organized into one or two SHs.
Texture Processing Cluster (TPC)	Shader Array (SH)	A group made up of several SMs or CUs.
Stream Multiprocessor (SM) / Multiprocessor	Compute Unit (CU)	Fundamental unit of computation, replicated multiple times on a GPU.
Sub-core/partition	SIMD	A group of cores to execute one warp/wave.
Stream Processor (SP) / CUDA Core / FPxx Core	Stream Processor / SIMD Lane / VALU Lane	A parallel execution lane comprising an SM or CU.

Software Stack[软件栈]

- Radeon Open Compute platform (ROCm)
  - AMD’s open-source software stack
- Multiple layers
  - **Language runtime**: language-specific runtime
  - **ROCr**: user-level language-agnostic runtime
  - **ROCt**: user-space driver talking to the lower-level ROCk
  - **ROCk**: kernel driver to initialize and register with CP the queues allocated by runtimes



ROCm



2020: AMD ROCm™ 4.0

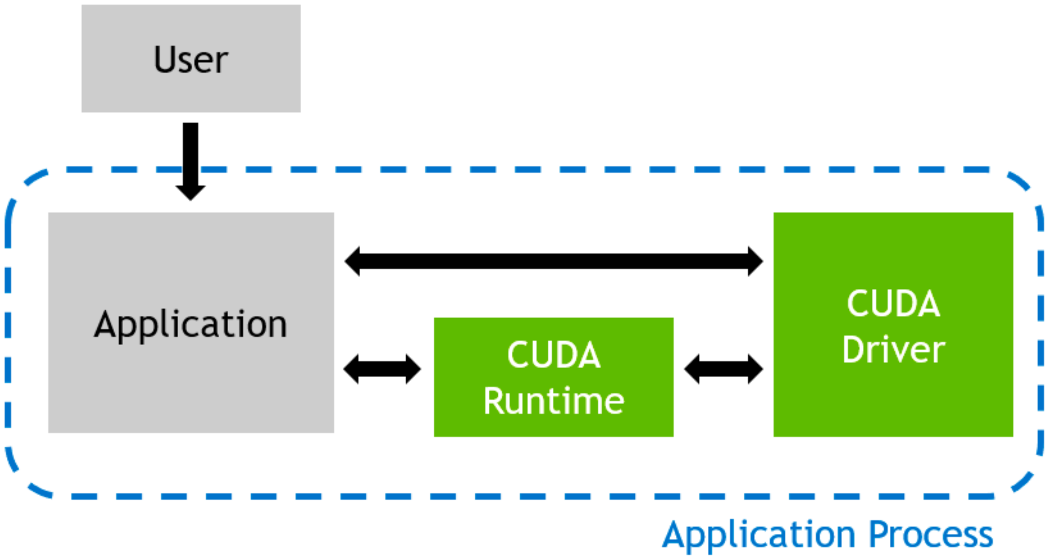
Complete Exascale Solution for ML/HPC

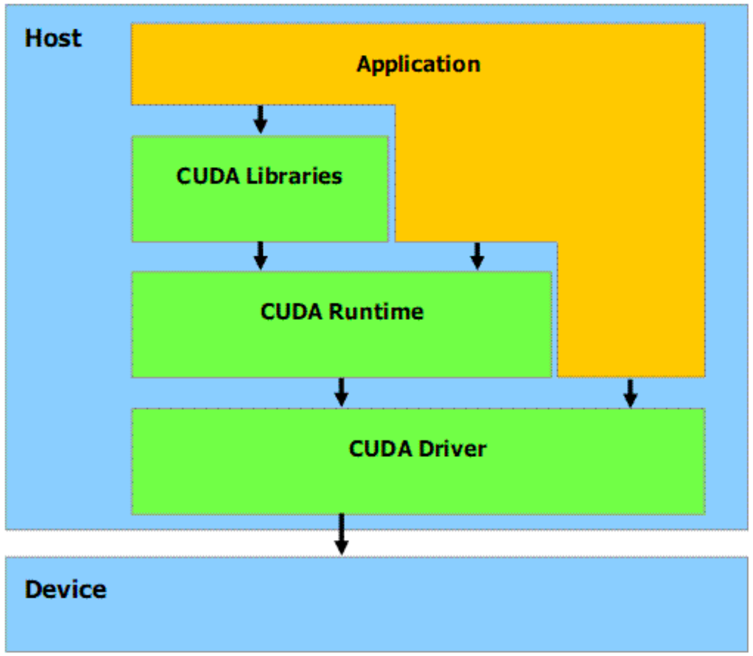
Applications	HPC Apps		ML Frameworks	
Cluster Deployment	Singularity	SLURM	Docker	Kubernetes
Tools	Debugger	Profiler, Tracer	System Valid.	System Mgmt.
Portability Frameworks	Kokkos	Magma	GridTools	ONNX
Math Libraries	RNG, FFT	Sparse	BLAS, Eigen	MIOpen
Scale-Out Comm. Libraries	OpenMPI	UCX	MPICH	RCCL
Programming Models	OpenMP	HIP	OpenCL	
Processors	CPU + GPU			

https://rocmdocs.amd.com/en/latest/

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





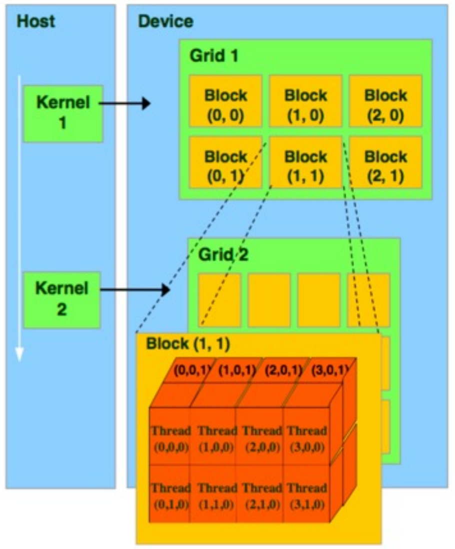
https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html

Detailed Kernel Launch[任务启动细节]

- S0: *application* creates user-mode queues (i.e., streams)
  - The queue is associated with a specific GPU
- S1: *application* places kernel dispatch packets into the queue
  - Done with user-level memory writes in ROCm (no kernel drivers)
  - Dependencies should be specified
- S2: *CPU* rings the doorbell to notify the CP of the GPU device
- S3: *CP* reads the packet, understands the kernel parameters
- S4: *CP* sends WGs to SPIs, which then launches WFs to CUs
- S5: when final WF is finished, *CP* sends a completion signal specified in the kernel dispatch packet
- S6: next, *CPU* receives an interrupt to pass the completion signal to runtime, which further completes the kernel in application code

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