Advanced Computer Architecture
高 级 计 算 机 体 系 结 构

第6讲： DLP and GPU (1)

张献伟

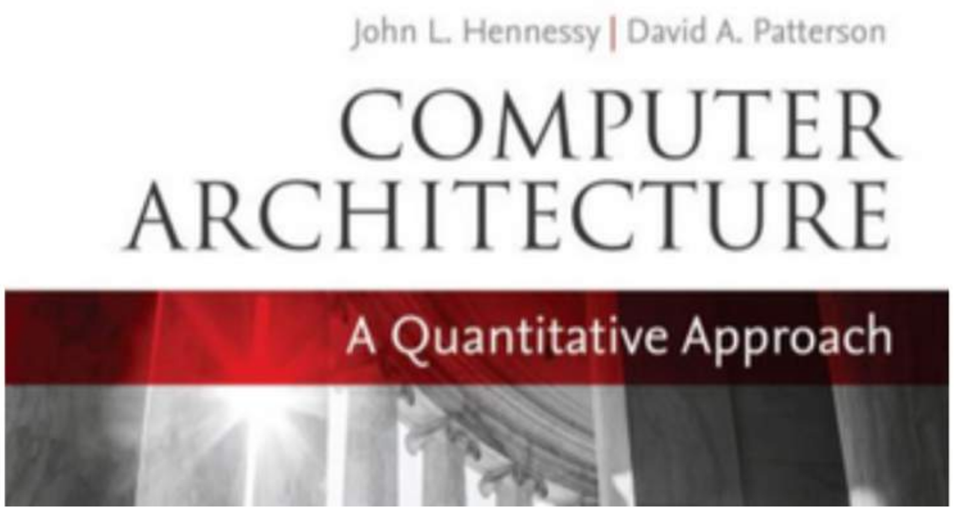
xianweiz.github.io

DCS5367, 11/9/2021

作业 – HW2

- https://xianweiz.github.io/teach/dcs5637/f2021.html
- 截止时间：11.21, 23:59
- 提交方式：超算习堂
  - 注册（ https://easyhpc.net/ ）
  - 加入课程（https://easyhpc.net/course/133）
  - 作业列表：HW2

HW-2, review-form.txt



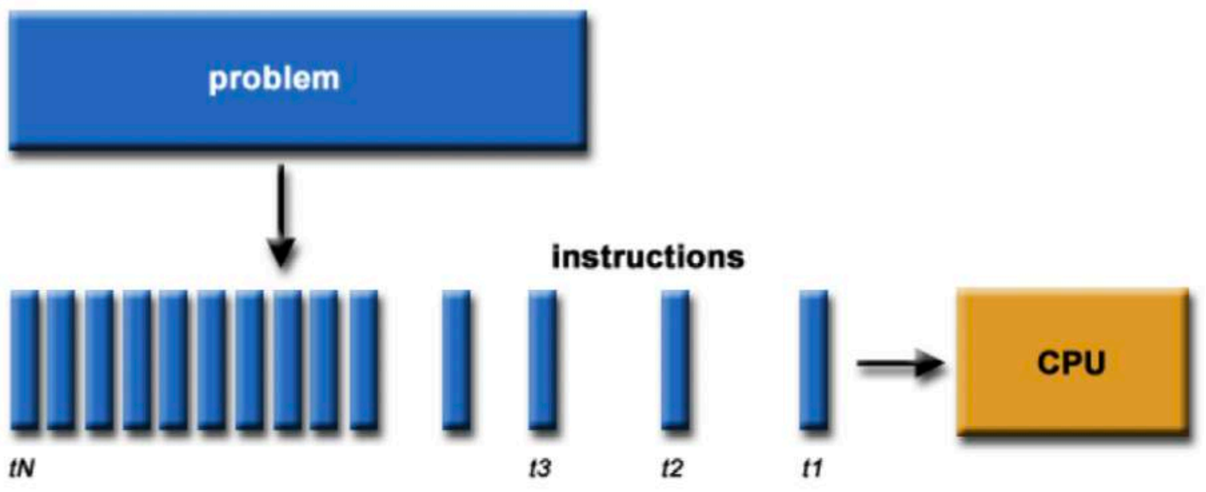
DCS5637 - 高级计算机体系结构（周二）

0人 其他

Advanced Computer Architecture，Fall 2021 课程主页：https://xianw

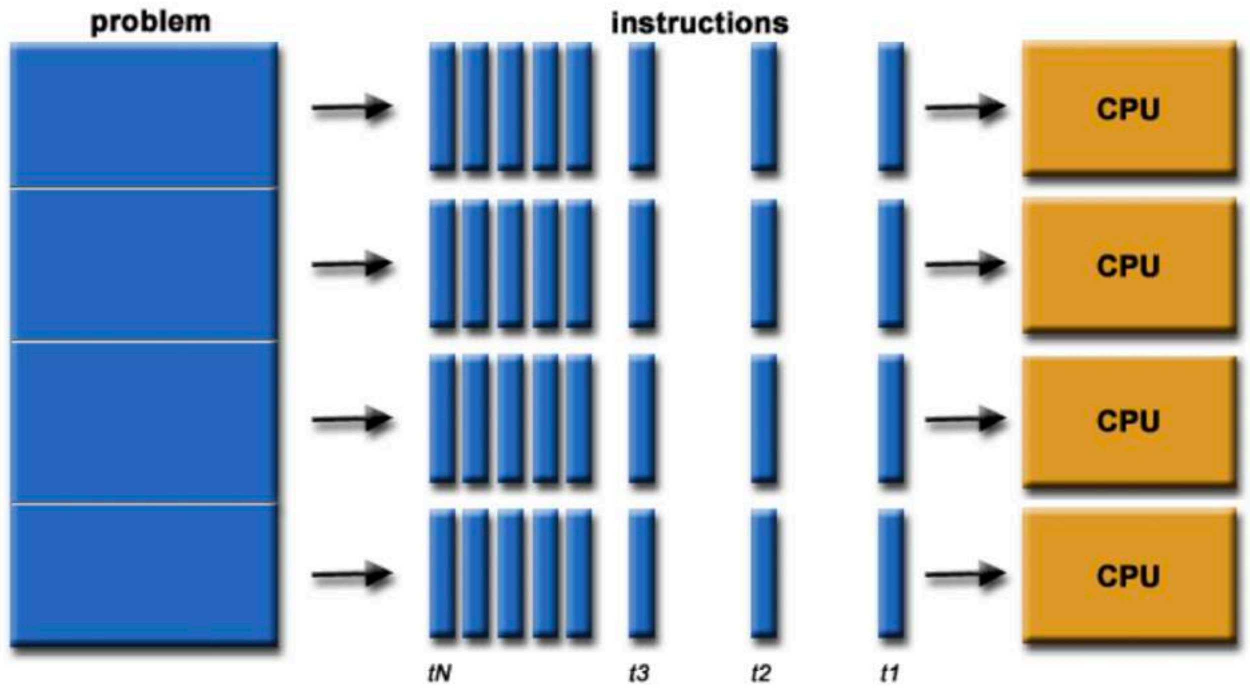
Serial Computing[串行计算]

- Traditionally, software has been written for serial computation
  - To be run on a single computer having a single CPU
  - A problem is broken into a discrete series of instructions
  - Instructions are executed one after another
  - Only one instruction may execute at any moment



Parallel Computing[并行计算]

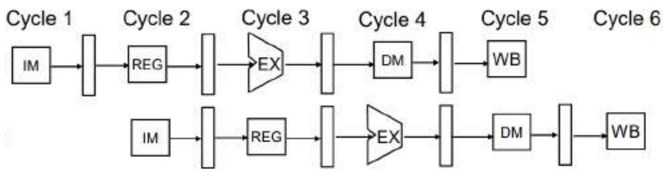
- Simultaneously use multiple compute resources to solve a computational problem
  - Typically in high-performance computing (HPC)
- HPC focuses on performance
  - To solve biggest possible problems in the least possible time



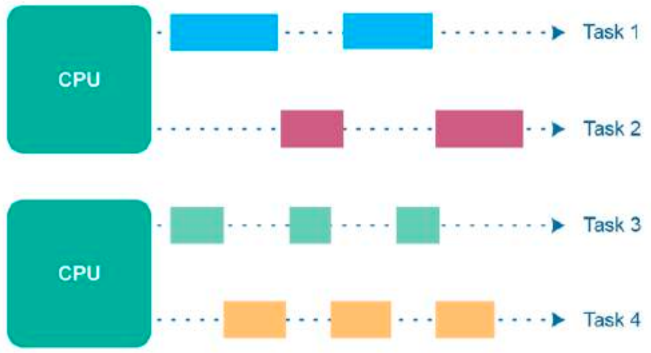
https://www.ima.umn.edu/materials/2010-2011/T11.28-29.10/10287/IMA-PPtTutorial.pdf

Types of Parallel Computing[并行类型]

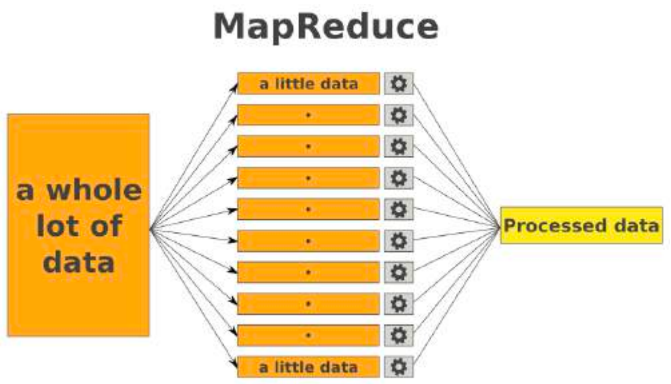
- Instruction level parallelism[指令级并行]
  - Classic RISC pipeline (fetch, …, write back)



- Task parallelism[任务级并行]
  - Different operations are performed concurrently
  - Task parallelism is achieved when the processors execute on the same or different data



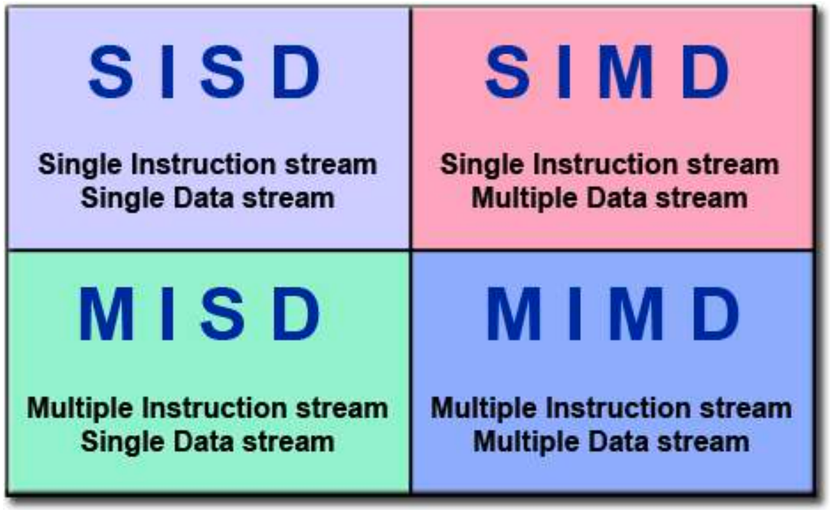
- Data parallelism[数据级并行]
  - Distribution of data across different parallel computing nodes
  - Data parallelism is achieved when each processor performs the same task on different pieces of the data



https://www.uio.no/studier/emner/matnat/ifi/IN5050/v20/undervisningsmaterialet/in5050-simd.pdf

Taxonomy[分类]

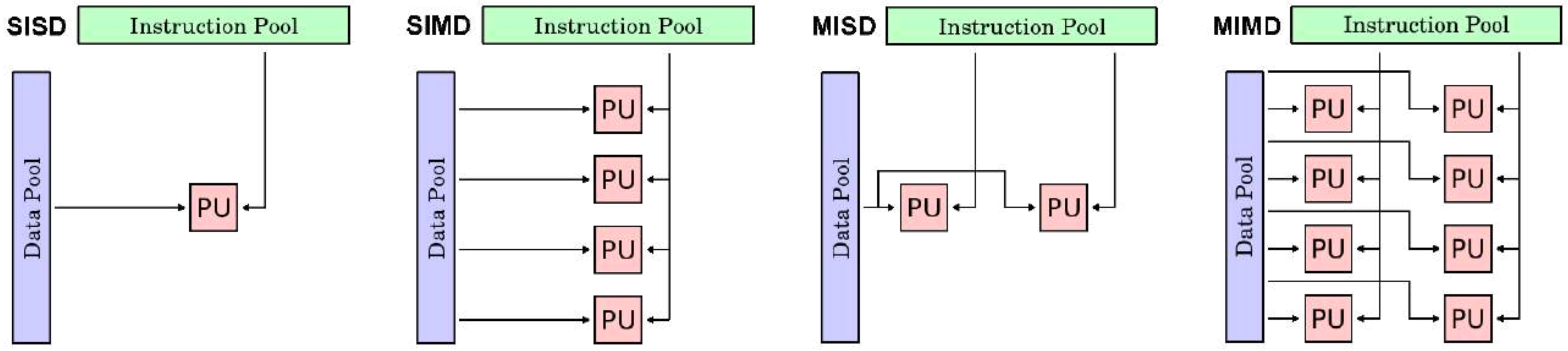
- **Flynn's Taxonomy** (1966) is widely used to classify parallel computers
  - Distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction Stream*** and ***Data Stream***
  - Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***
- 4 possible classifications according to Flynn


S I S D
Single Instruction stream
Single Data stream
S I M D
Single Instruction stream
Multiple Data stream
M I S D
Multiple Instruction stream
Single Data stream
M I M D
Multiple Instruction stream
Multiple Data stream

https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial

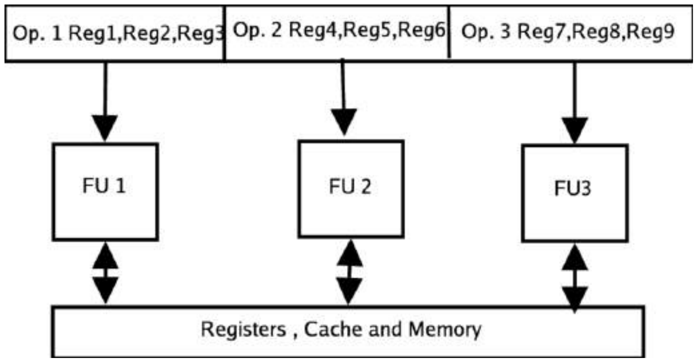
Taxonomy (cont.)

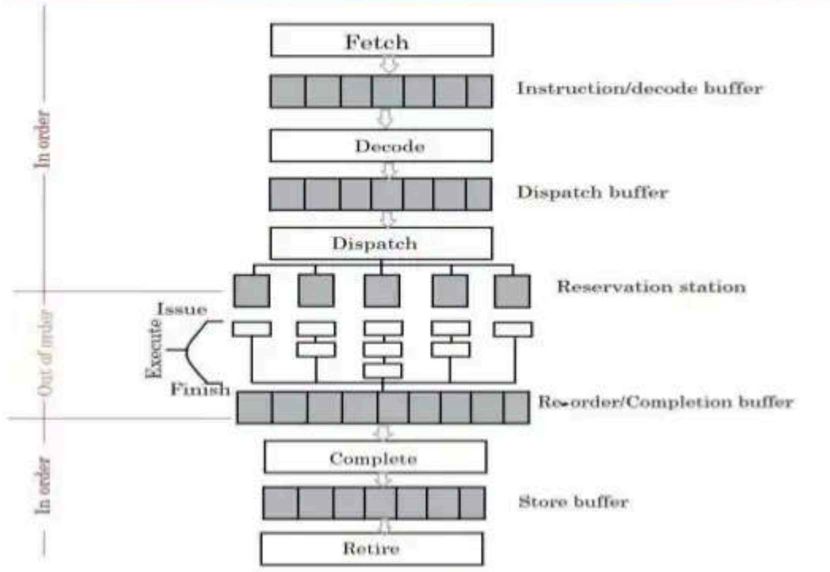
- SISD: single instruction, single data
  - A serial (non-parallel) computer
- **SIMD**: single instruction, multiple data
  - Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing
- MISD: multiple instruction, single data
  - Few (if any) actual examples of this class have ever existed
- MIMD: multiple instruction, multiple data
  - Examples: supercomputers, multi-core PCs, VLIW



SIMD: vs. superscalar and VLIW[对比]

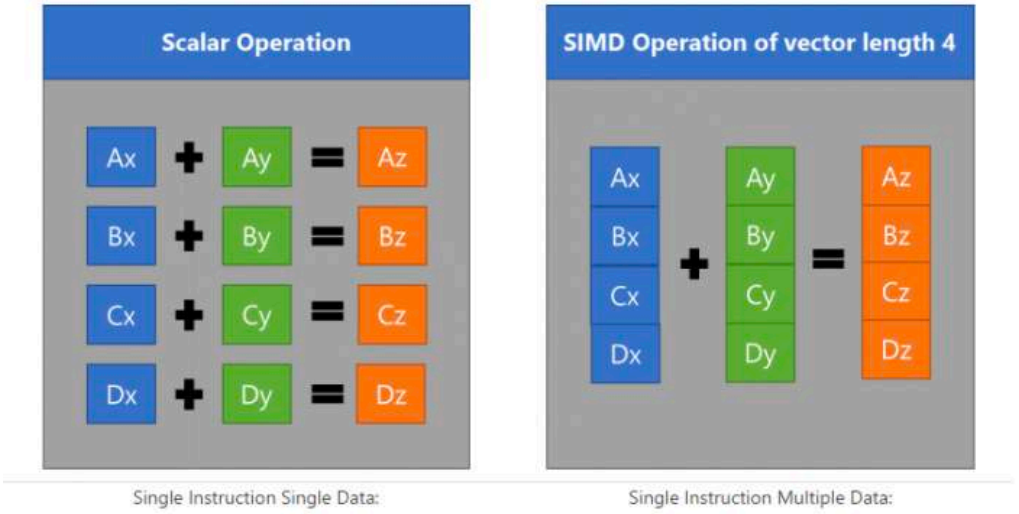
- SIMD performs the same operation on multiple data elements with one single instruction
  - Data-level parallelism
- Superscalar dynamically issues multi insts per clock[超标量]
  - Instruction level parallelism (ILP)
- VLIW receives long instruction words, each comprising a field (or opcode) for each execution unit[超长指令字]
  - Instruction level parallelism (ILP)





SIMD: Vector Processors[向量处理器]

- Vector processor (or array processor)[处理器]
  - CPU that implements an instruction set containing instructions that operate on one-dimensional arrays (vectors)
- People use vector processing in many areas[应用]
  - Scientific computing
  - Multimedia processing (compression, graphics, image processing, …)
- Instruction sets[指令集]
  - MMX
  - SSE
  - AVX
  - NEON
  - …

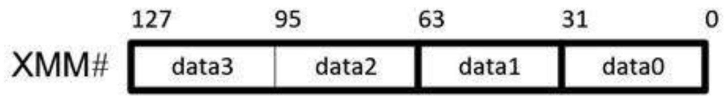

Scalar Operation
Ax + Ay = Az
Bx + By = Bz
Cx + Cy = Cz
Dx + Dy = Dz
SIMD Operation of vector length 4
Ax Bx Cx Dx + Ay By Cy Dy = Az Bz Cz Dz
Single Instruction Single Data:
Single Instruction Multiple Data:

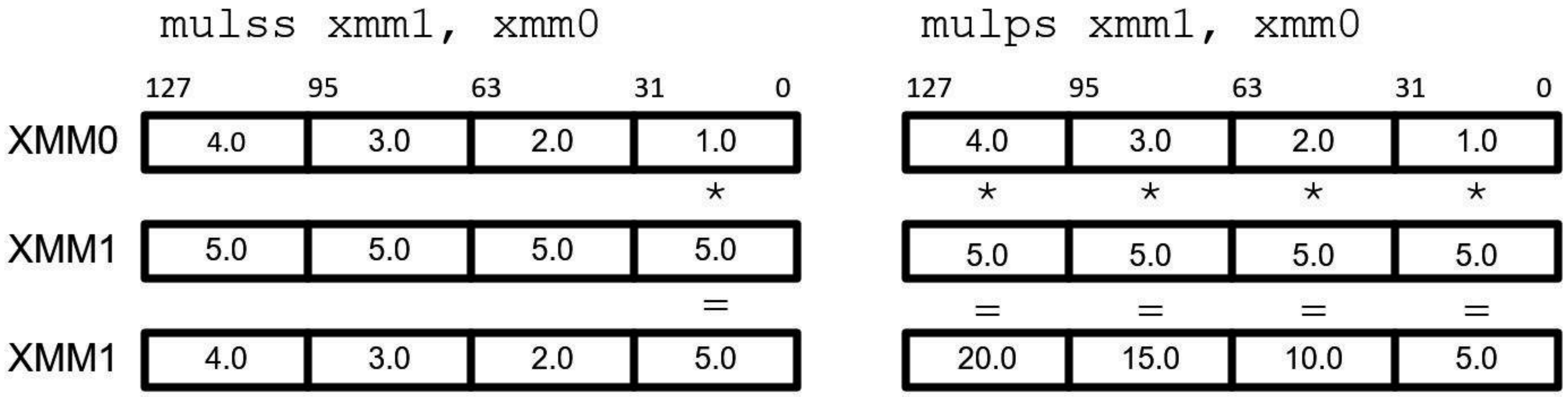
SIMD: MMX

- MMX is officially a meaningless initialism trademarked by Intel; unofficially,
  - MultiMedia eXtension
  - Multiple Math eXtension
  - Matrix Math eXtension
- Introduced on the "Pentium with MMX Technology" in 1998
- SIMD computation processes multiple data in parallel with a single instruction
  - MMX gives 2 x 32-bit computations at once
  - MMX defined 8 "new" 64-bit integer registers (mm0 ~ mm7)
  - 3DNow! was the AMD extension of MMX

SIMD: SSE

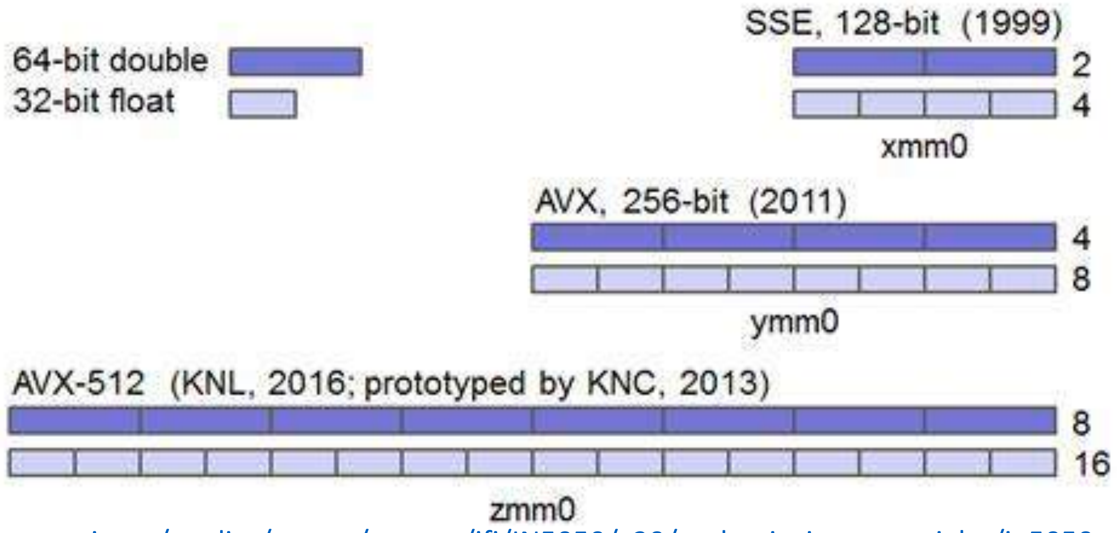
- Streaming SIMD Extensions
  - SSE defines 8 new 128-bit registers (xmm0 ~ xmm7) for FP32 computations
    - Since each register is 128-bit long, we can store total 4 FP32 numbers
  - 4 simultaneous 32-bit computations





SIMD: AVX

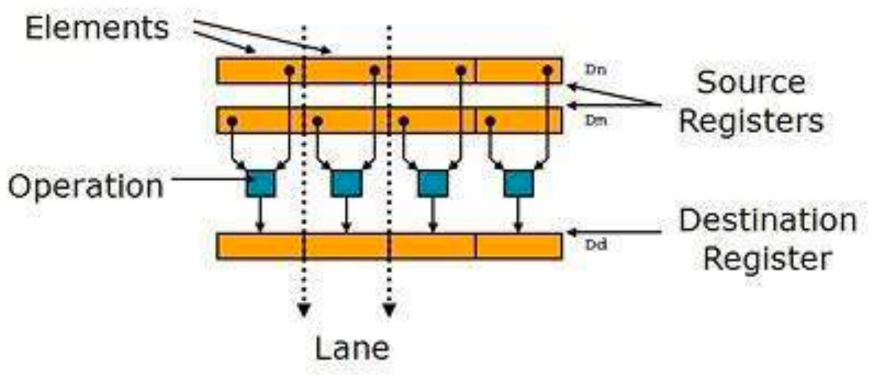
- Advanced Vector Extensions (AVX)
  - A new-256 bit instruction set extension to SSE
    - 16-registers available in x86-64
    - Registers renamed from XMMi to YMMi
  - Yet a proposed extension is AXV-512
    - A 512-bit extension to the 256-bit XMM
    - Supported in from Intel's Xeon Phi x200 (Knights Landing) and Skylake-SP, and onwards



https://www.uio.no/studier/emner/matnat/ifi/IN5050/v20/undervisningsmaterialet/in5050-simd.pdf

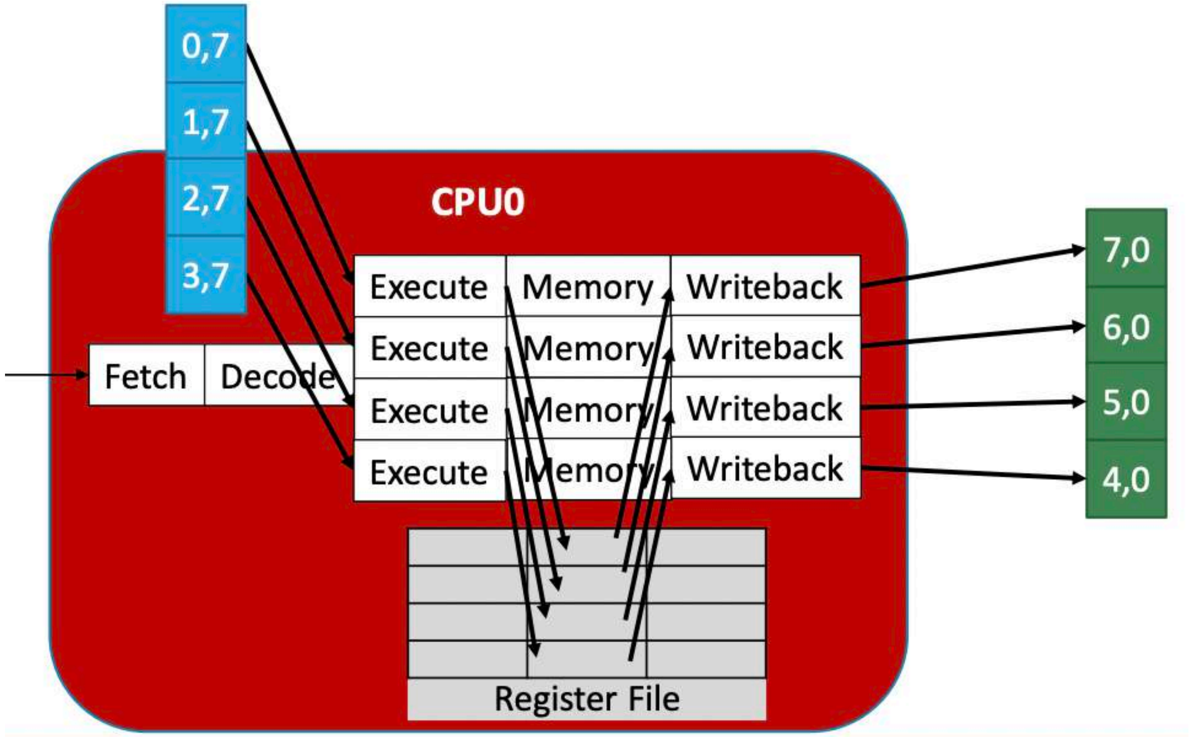
SIMD: NEON

- ARM Advanced SIMD Extensions
  - Introduced by ARM in 2004 to accelerate media and signal processing
    - NEON can for example execute MP3 decoding on CPUs running at 10 MHz
  - 128-bit SIMD Extension for the ARMv7 & ARMv8
    - Data types can be: signed/unsigned 8-bit, 16-bit, 32-bit or 64-bit



Data Parallelism: SIMD

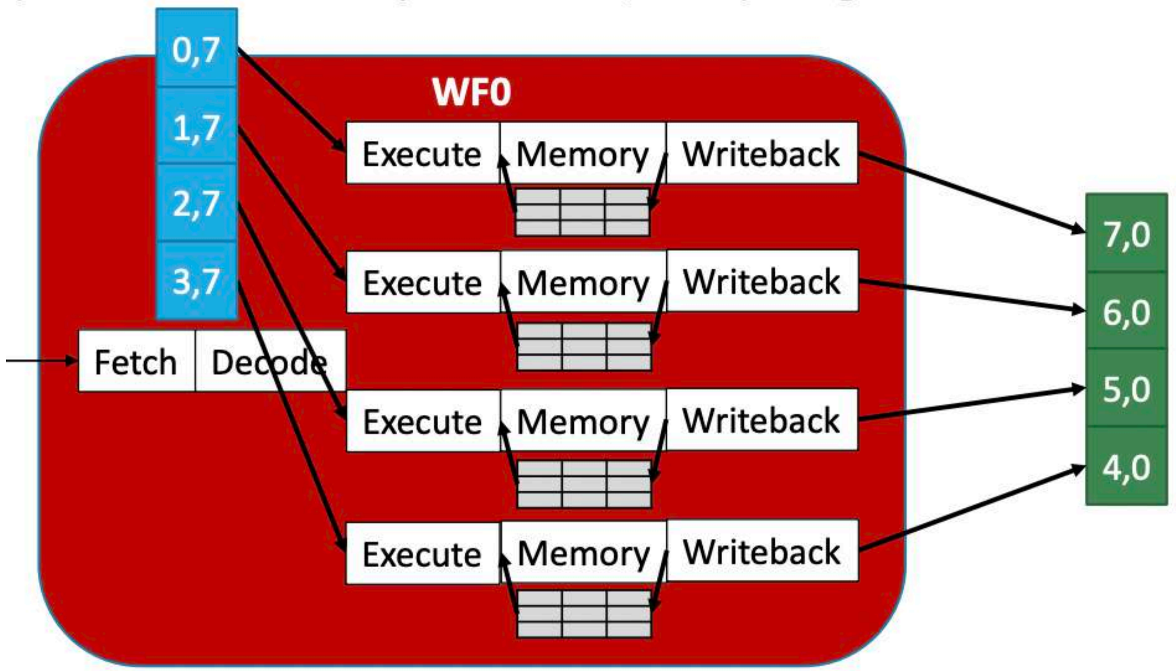
- Single Instruction Multiple Data
  - Split identical, independent work over multiple execution units (lanes)
  - More efficient: eliminate redundant fetch/decode
  - One Thread + Data Parallel Ops → Single PC, single register file



https://courses.cs.washington.edu/courses/cse471/13sp/lectures/GPUsStudents.pdf

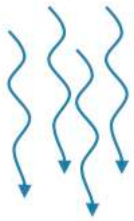
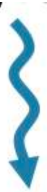
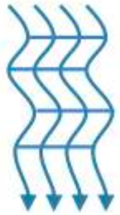
Data Parallelism: SIMT

- Single Instruction Multiple Thread
  - Split identical, independent work over multiple threads
  - Multiple Threads + Scalar Ops → One PC, multiple register files
  - ≈ SIMD + multithreading
  - Each thread has its own registers



https://courses.cs.washington.edu/courses/cse471/13sp/lectures/GPUsStudents.pdf

Execution Model[执行模型]

MIMD	SIMD	SIMT
		
Multiple independent threads	One thread with wide execution datapath	Multiple lockstep threads
Multicore CPUs	x86 SSE/AVX	GPUs

- SI(MD/MT)
  - Broadcasting the same instruction to multiple execution units
  - Replicate the execution units, but they all share the same fetch/decode hardware

SIMD and SIMT are used interchangeably

SIMD: GPU vs. Traditional

- Traditional SIMD contains a single thread
  - Programming model is SIMD (no threads)
  - SW needs to know vector length
  - ISA contains vector/SIMD instructions
- GPU SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
    - SW does not need to know vector length
    - Enables memory and branch latency tolerance
  - ISA is scalar → vector instructions formed dynamically
- Essentially, it is SPMD programming model implemented on SIMD hardware

Example: add two vectors

C:

```
for(i=0;i<n;++i) a[i]=b[i]+c[i];
```

Matlab:

```
a=b+c;
```

SIMD:

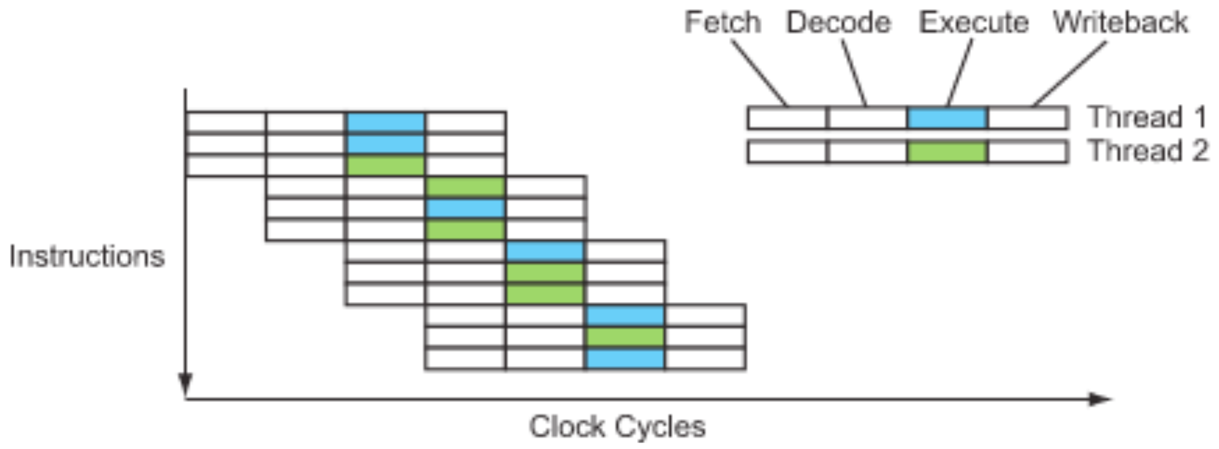
```
void add(uint32_t *a, uint32_t *b, uint32_t *c, int n) {
    for(int i=0; i<n; i+=4) {
        //compute c[i], c[i+1], c[i+2], c[i+3]
        uint32x4_t a4 = vld1q_u32(a+i);
        uint32x4_t b4 = vld1q_u32(b+i);
        uint32x4_t c4 = vaddq_u32(a4,b4);
        vst1q_u32(c+i,c4);
    }
}
```

SIMT:

```
__global__ void add(float *a, float *b, float *c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    a[i]=b[i]+c[i]; //no loop!
}
```

SMT[多线程]

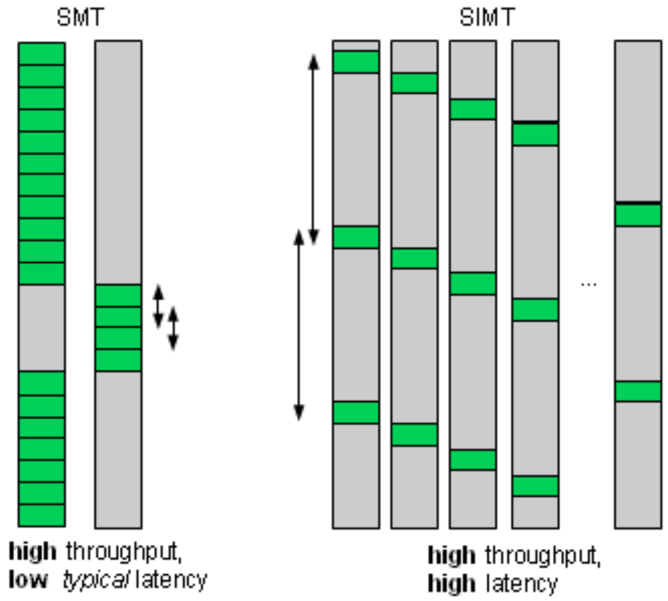
- SMT: simultaneous multithreading
  - Instructions from multiple threads issued on the same cycle
    - Use register renaming and dynamic scheduling facility of multi-issue architecture
  - Needs more hardware support
    - Register files, PC's for each thread
    - Support to sort out which threads to get results from which instructions
    - Thread scheduling, context switching
  - Maximize utilization of execution units



http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html

SMT vs. SIMT[比较]

- SMT: maximize the chances of an instruction to be issued without having to switch to another thread
  - superscalar execution
  - out-of-order execution
  - register renaming
  - branch prediction
  - speculative execution
  - cache hierarchy
  - speculative prefetching



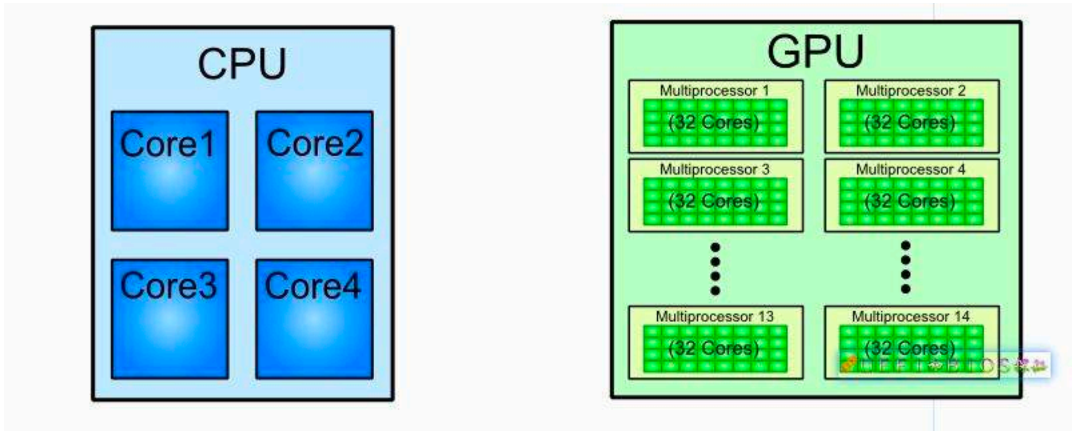
- SIMT: keep massive threads to achieve high throughput
  - Hardware becomes simpler and cheaper
  - No OoO, no prefetching, …

http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html

CPU vs. GPU

- CPU
  - Low compute density
  - Complex control logic
  - Fewer cores optimized for serial operations
    - Fewer execution units (ALUs)
    - Higher clock speeds
  - Low latency tolerance

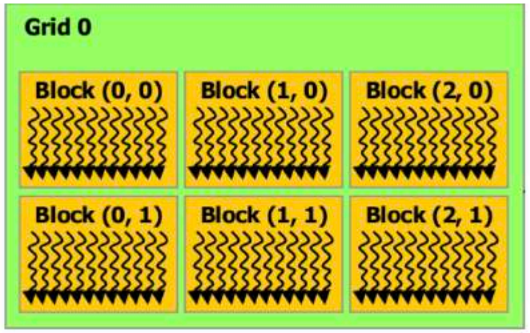
- GPU
  - High compute density
  - Simple control logic
  - 1000s cores optimized for parallel operations
    - Many parallel execution units (ALUs)
    - Lower clock speeds
  - High latency tolerance



GPU Overview

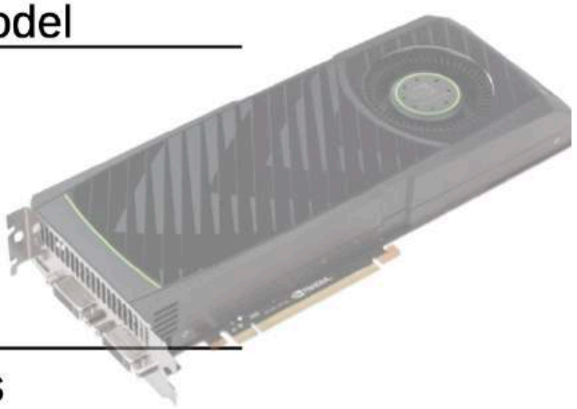
Software

```
__global__ void scale(float a, float * X)
{
    unsigned int tid;
    tid = blockIdx.x * blockDim.x
          + threadIdx.x;
    X[tid] = a * X[tid];
}
```

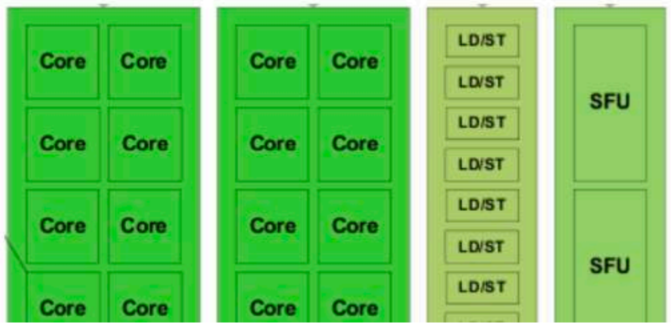


Architecture: **multi-thread** programming model

SIMT microarchitecture



Hardware datapaths: **SIMD** execution units



Hardware

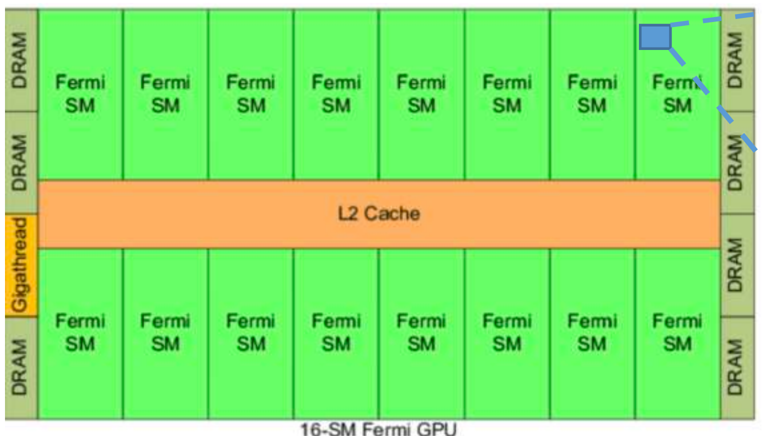
3

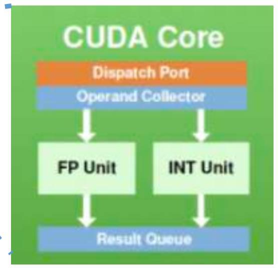
GPU Overview(cont.)

- A GPU contains several largely independent processors called "**Streaming Multiprocessors**" (SMs)
  - Each SM hosts multiple "cores", and each "core" runs a thread
  - For instance, Fermi(2010) has up to 16 SMs w/ 32 cores per SM
    - So up to 512 threads can run in parallel

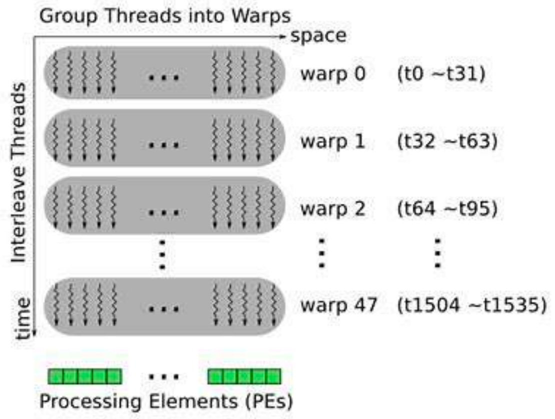
A100: 128 SMs w/ 64 cores per SM

- Some SIMT threads are grouped to execute in lockstep
  - One warp contains 32 threads
- Multiple 'groups' can be executed simultaneously
  - For Fermi, up to 48 warps per SM



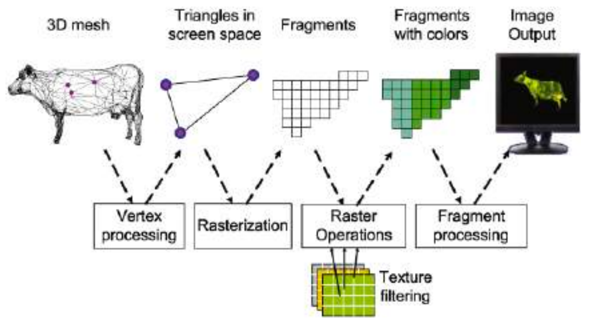


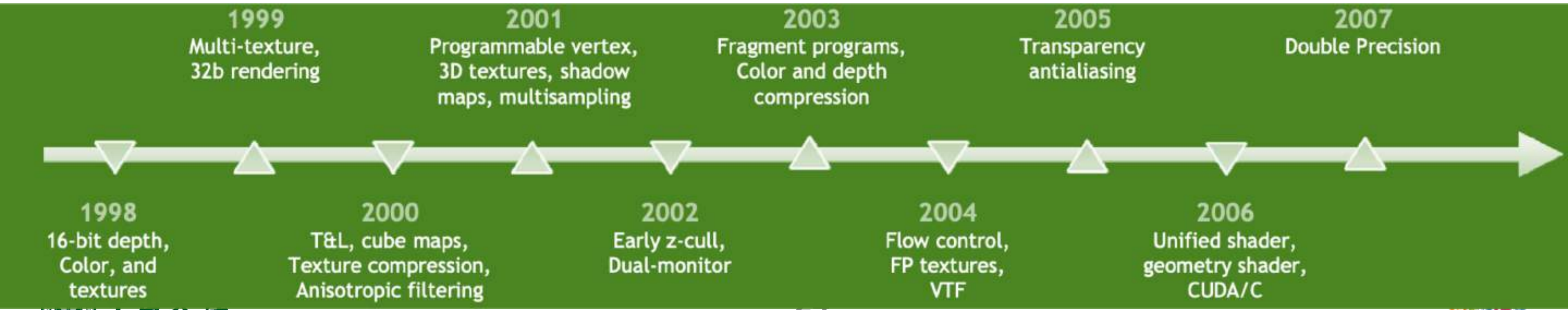
Example of SIMT Execution
Assume 32 threads are grouped into one warp.



GPU Evolution[演进]

- Arcade boards and display adapters (1951 - 1995)
  - ATI: founded in 1985
  - Nvidia: founded in 1993
- 3D revolution (1995 – 2006)
  - Term “graphics processing unit”: 1999
    - Nvidia GeForce 256
  - Rivalry between ATI and Nvidia
- General purpose GPU (2006 - present)
  - AI , data analytics, scientific computing, graphics rendering, etc




1999
Multi-texture, 32b rendering
2001
Programmable vertex, 3D textures, shadow maps, multisampling
2003
Fragment programs, Color and depth compression
2005
Transparency antialiasing
2007
Double Precision
1998
16-bit depth, Color, and textures
2000
T&L, cube maps, Texture compression, Anisotropic filtering
2002
Early z-cull, Dual-monitor
2004
Flow control, FP textures, VTF
2006
Unified shader, geometry shader, CUDA/C

GPGPU History[简史]

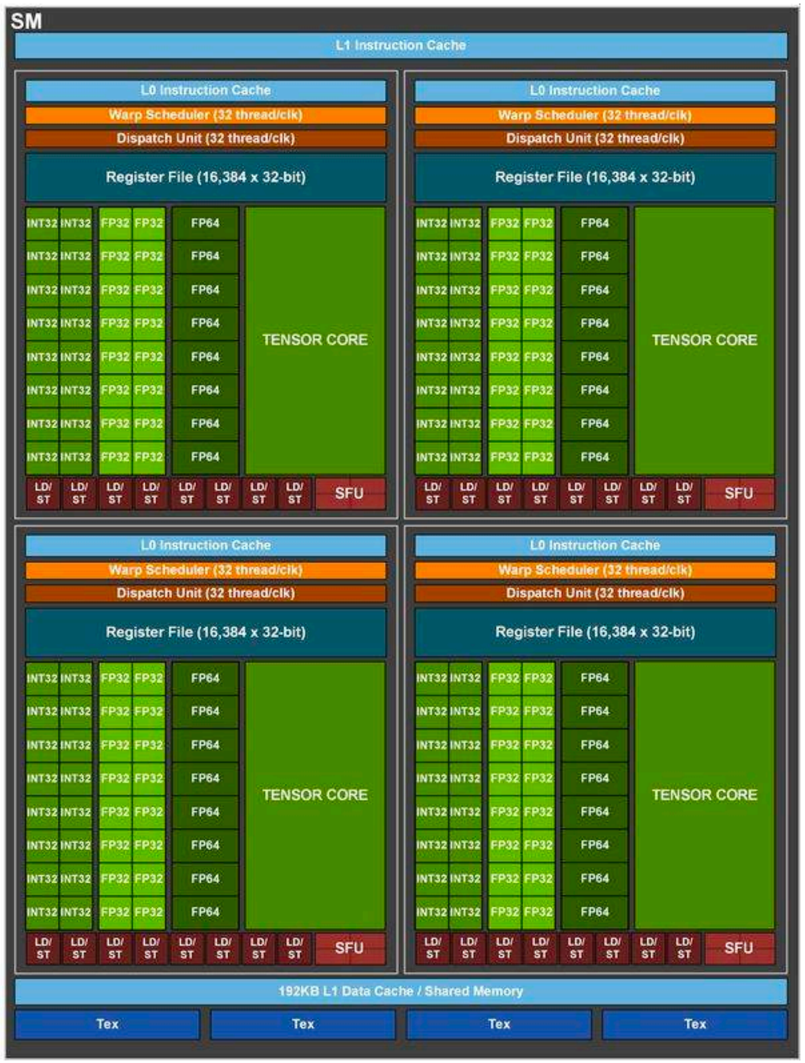
Year	AMD	Nvidia	Note
2006	AMD acquired ATI	Tesla (CUDA Launch)	Unified shader model
2007	TeraScale		Unified shader uarch
2009	TeraScale 2		
2010	TeraScale 3	Fermi / GTX580	First compute GPU
2011	GCN 1.0 / gfx6		VLIW → SIMD
2012		Kepler / GTX680	CUDA cores: 512 → 1536
2013	GCN 2.0 / gfx7		
2014	GCN 3.0 / gfx8	Maxwell / GTX980	Energy efficiency
2016	GCN 4.0 / gfx8	Pascal / GTX1080	
2017	GCN 5.0 / gfx9	Volta / GV100	First chip with Tensor cores
2018	GCN 5.1 / gfx9	Turing / RTX2080	
2019	RDNA 1.0 / gfx10		
2020	RDNA 2.0 / gfx10 CDNA 1.0 / gfx9	Ampere / RTX3090	First chip with Matrix cores

中山大學 SUN YAT-SEN UNIVERSITY

TFLOPS[衡量算力]

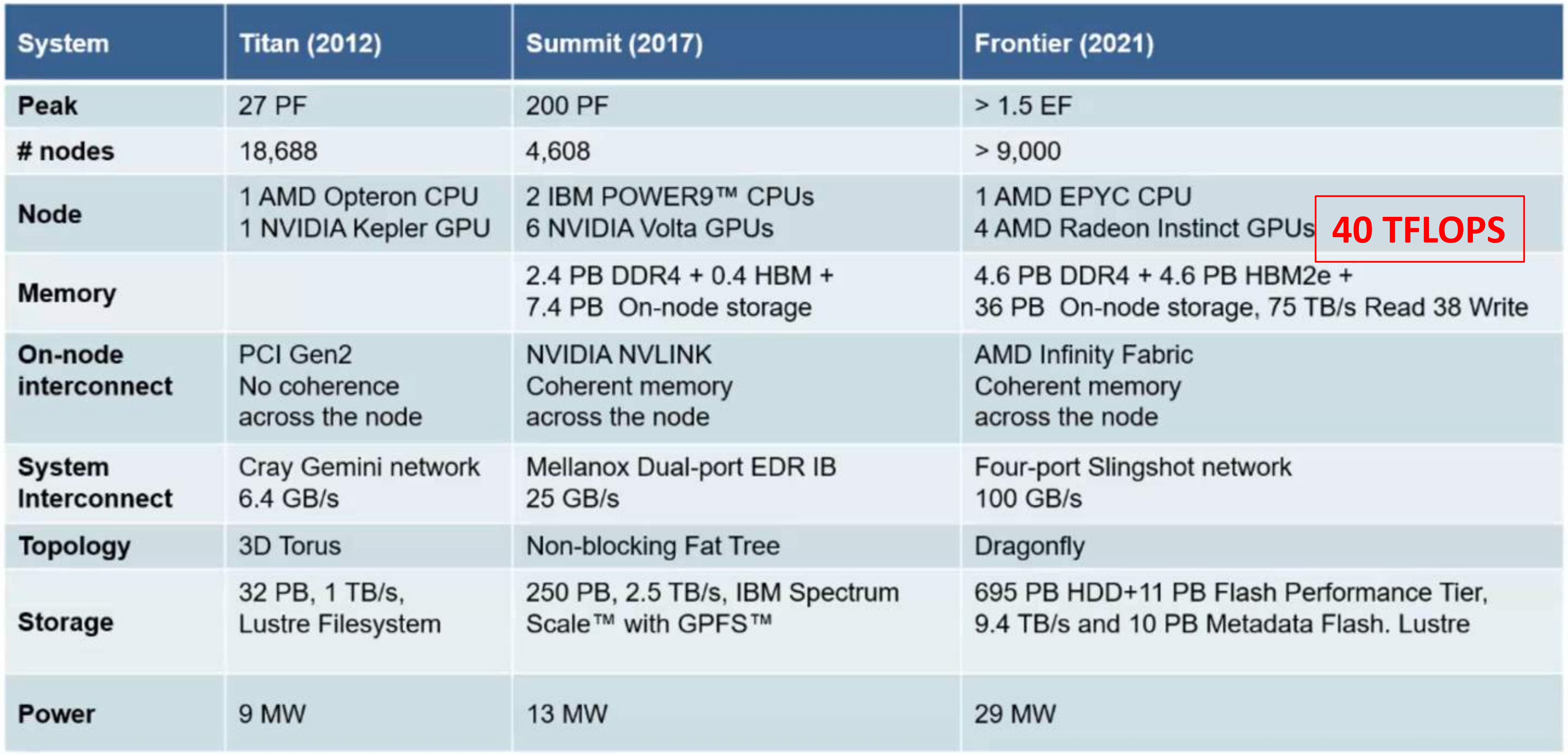
- A100 Tensor Core GPU
  - 108 SMs
    - GA100 Full GPU with 128 SMs
  - Base clock: 1065 MHz
  - Boost clock: 1410 MHz
  - Performance
    - FP64: 9.7 TFLOPS
    - FP32: 19.5 TFLOPS

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


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

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

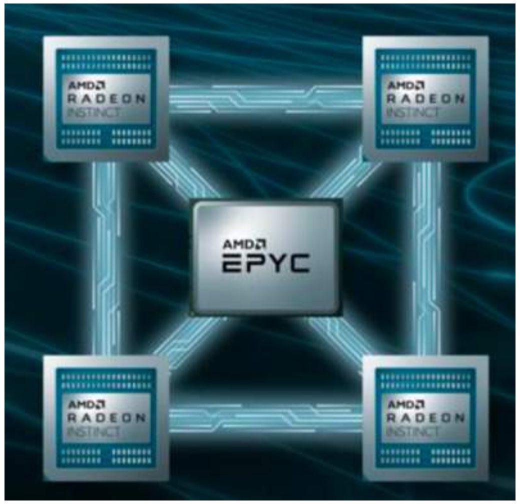
40 TFLOPS

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/

中山大學 SUN YAT-SEN UNIVERSITY

Frontier: 1.5 EFLOPS, How???



- 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/

天河超算

- 2009，天河-1
  - CPU + ATI GPU
    - 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

240 GFLOPS

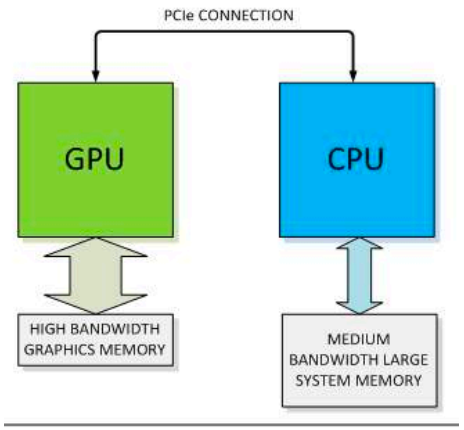
- 2010，天河-1A
  - CPU + Nvidia GPU
    - 2 * Intel Xeon X5670, 1 Nvidia Tesla M2050 (Fermi)
    - 2048 Galaxy "FT-1000" 1 GHz 8-core processors
  - 实测/峰值2.566P/4.7P FLOPS
  - 2010.11 TOP500第一

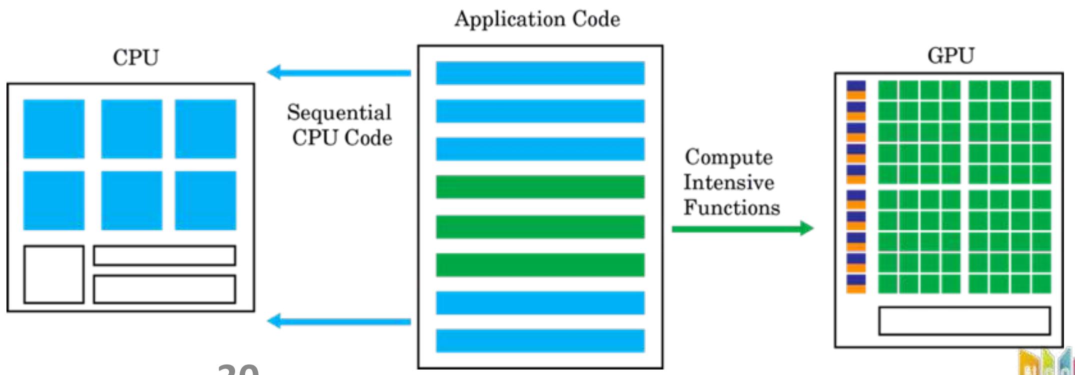
515 GFLOPS

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]

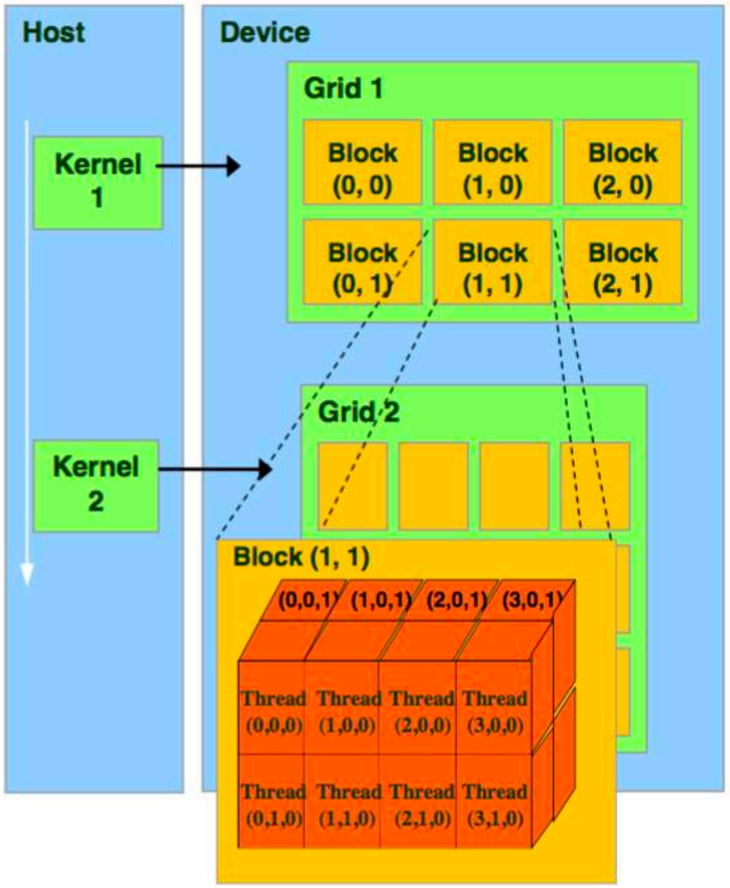
- GPU is viewed as a compute device that
  - Is a coprocessor to CPU (host)
  - Has its own main memory called device memory
  - Runs many threads in parallel
- Data-parallel parts of an application are executed on the device as **kernels**, which run in parallel on many threads
- CPU thread vs. GPU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


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
- 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

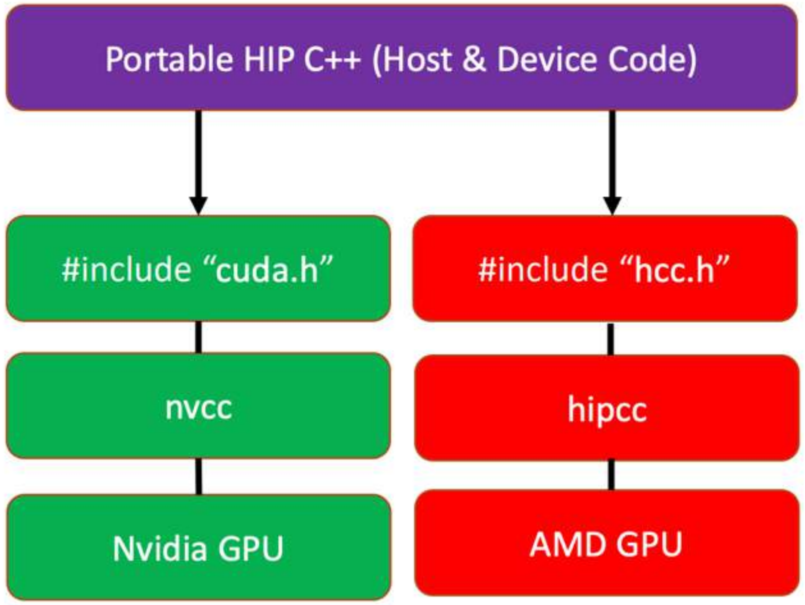


GPU Programming Choices[编程选择]

- **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



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();
```

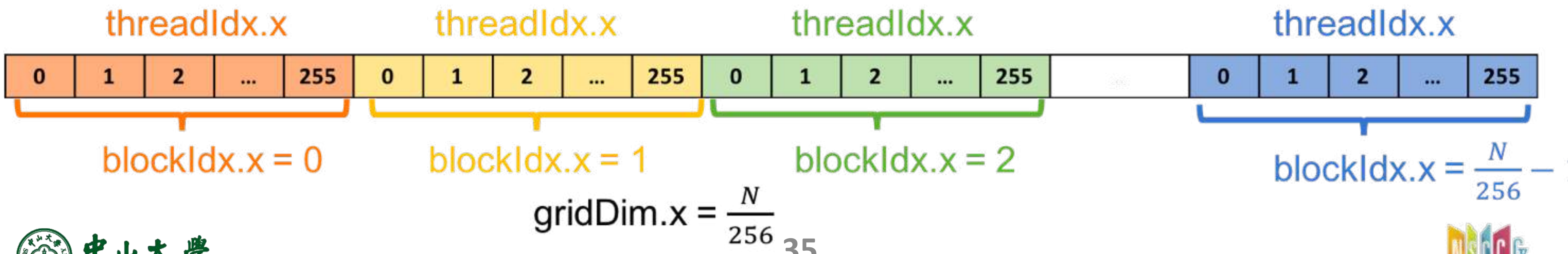
- Kernel launch

```
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
  - blockDimx.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);
```

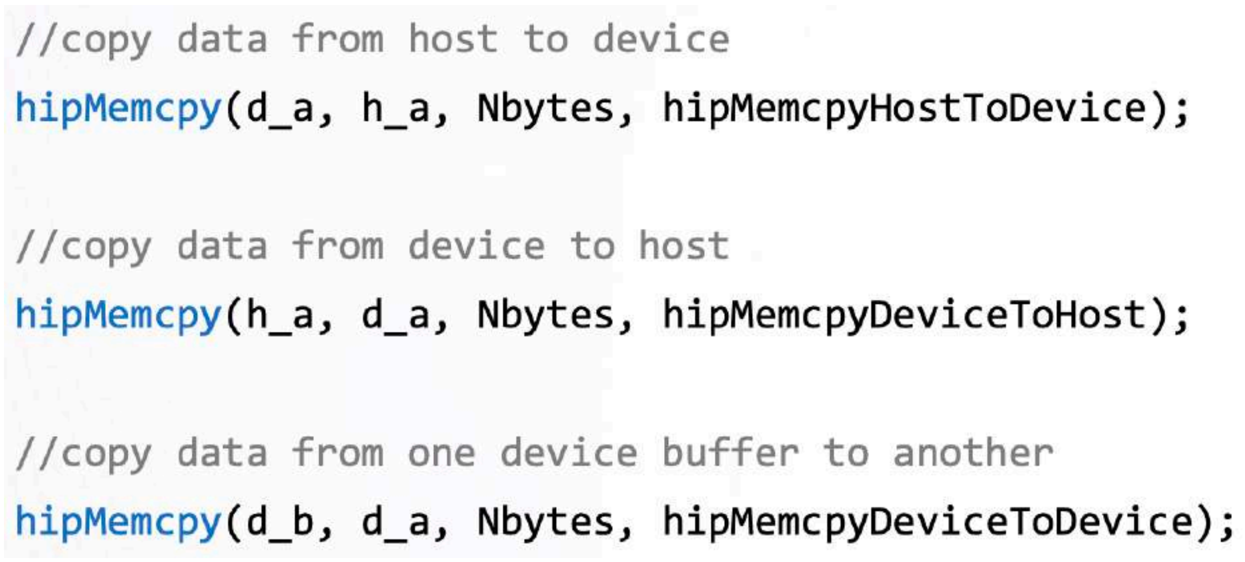
Example: Memory Allocation[内存分配]

- 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

- 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);
```

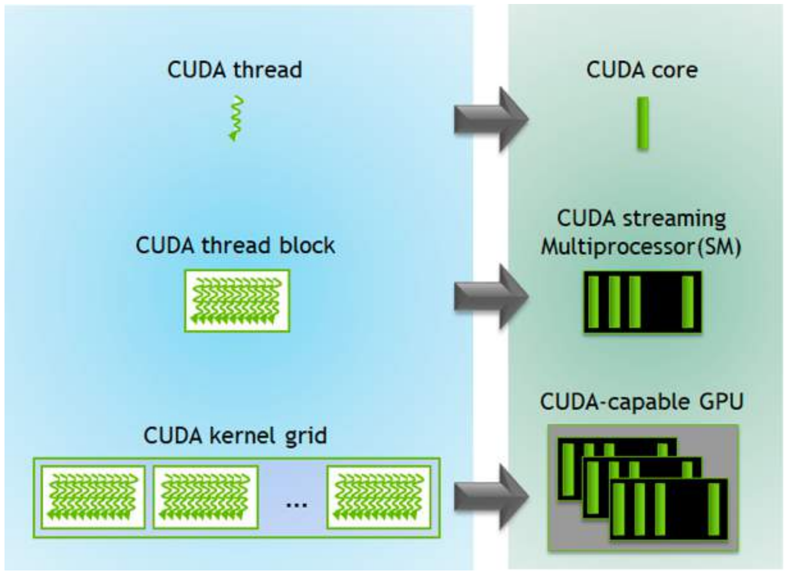
- The host can also query a device's *properties*:

```
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.

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



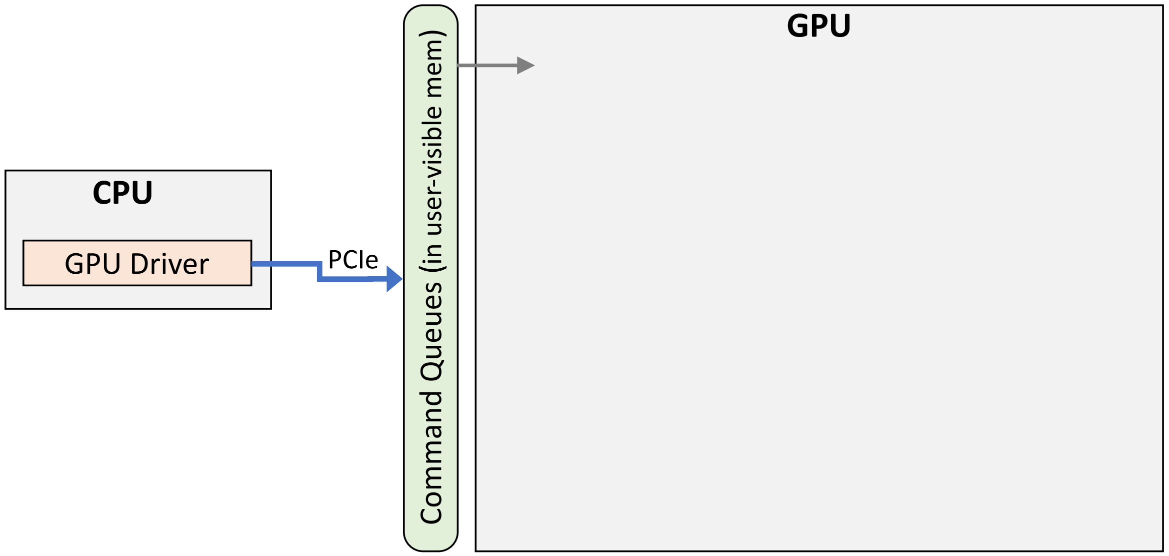
SIMD lane
(streaming processor)

Compute unit

GPU

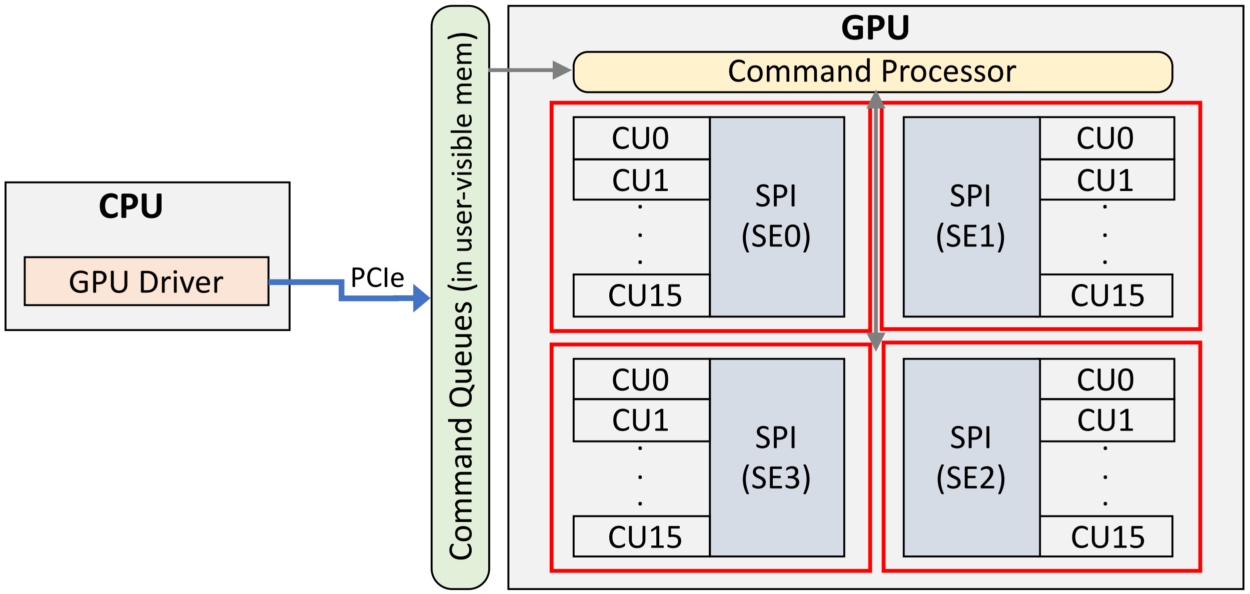
CPU-GPU

- 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



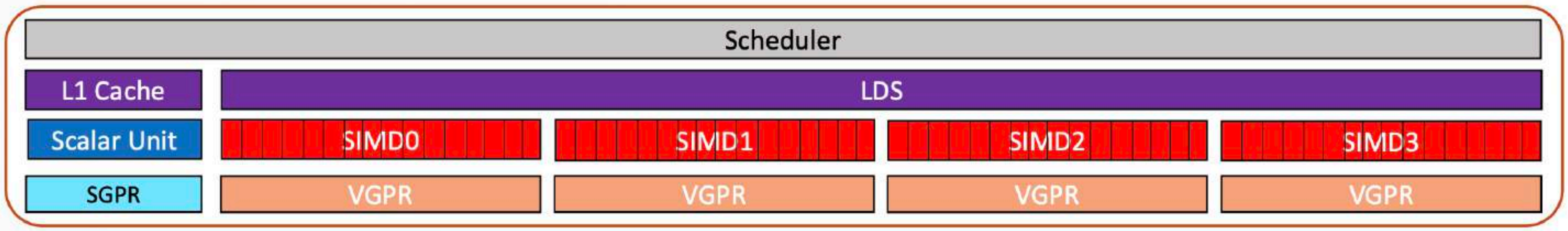
GPU Structure

- 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



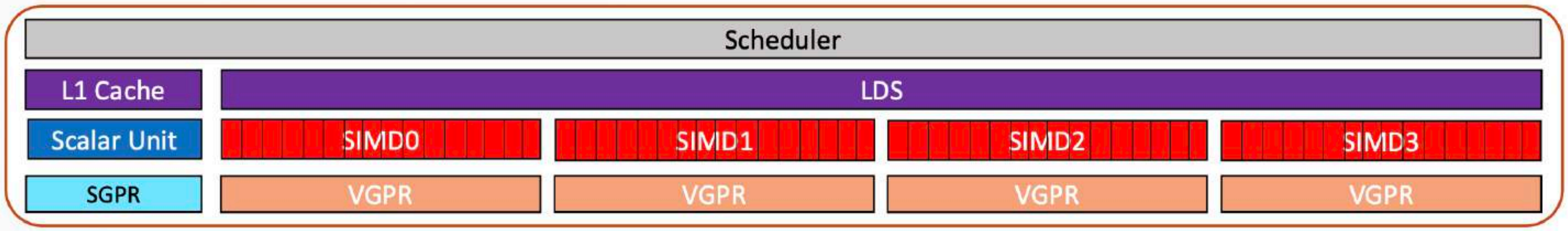
Compute Unit

- 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.



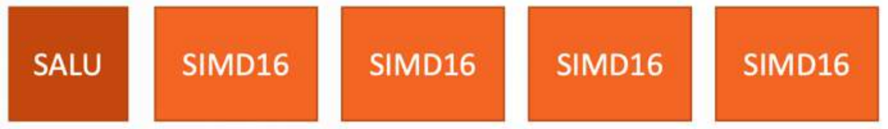
Compute Unit (cont.)

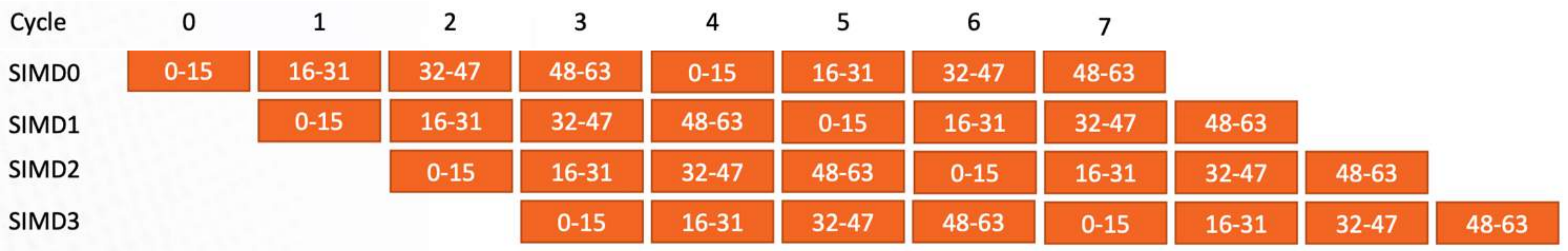
- GPRs[通用寄存器]
  - VGPR: vector general purpose register file
    - 4x 64KB (256KB total)
    - A maximum of 256 total registers per SIMD lane – each register is 64x 4-byte entries
  - SGPR: scalar general purpose register file
    - 12.5KB per CU
- L1 cache: 16KB[一级缓存]
- LDS: local data share (or, shared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



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

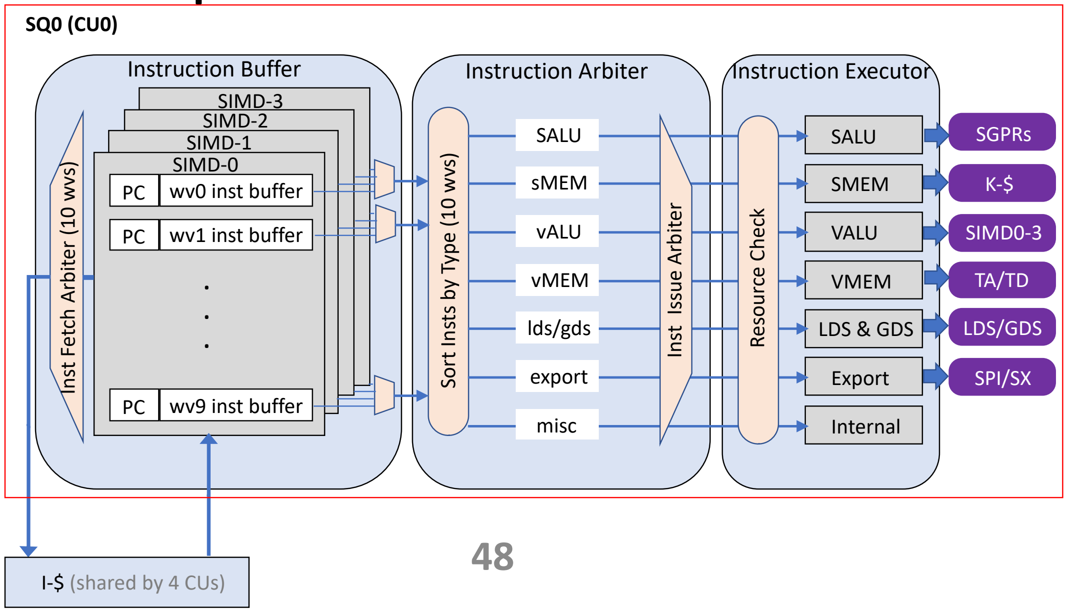




Cycle	0	1	2	3	4	5	6	7			
SIMD0	0-15	16-31	32-47	48-63	0-15	16-31	32-47	48-63			
SIMD1		0-15	16-31	32-47	48-63	0-15	16-31	32-47	48-63		
SIMD2			0-15	16-31	32-47	48-63	0-15	16-31	32-47	48-63	
SIMD3				0-15	16-31	32-47	48-63	0-15	16-31	32-47	48-63

Instruction Execution[指令执行]

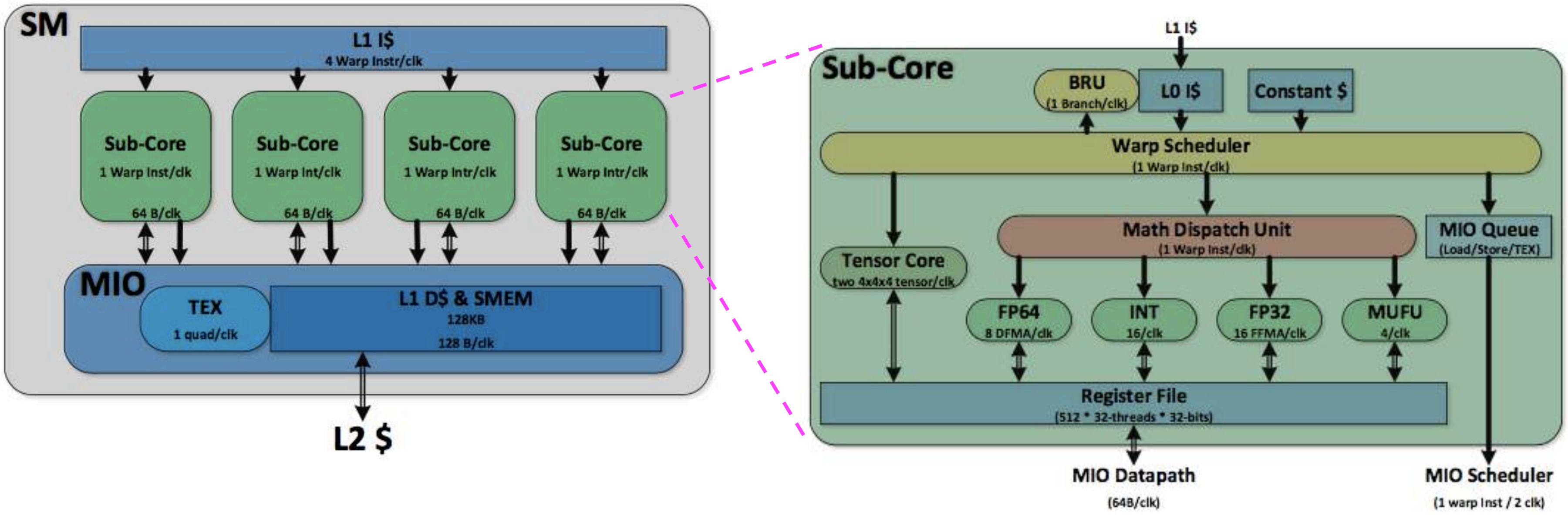
- **Instruction buffer** (IB): each cycle, the 10 wvs of the selected SIMD compete for instruction fetch (oldest wins)
- **Instruction arbiter** (IA): arbitrates multi wvs which want to execute the same type of instructions
- **Instruction executor** (IE): multiple execution units running in parallel;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

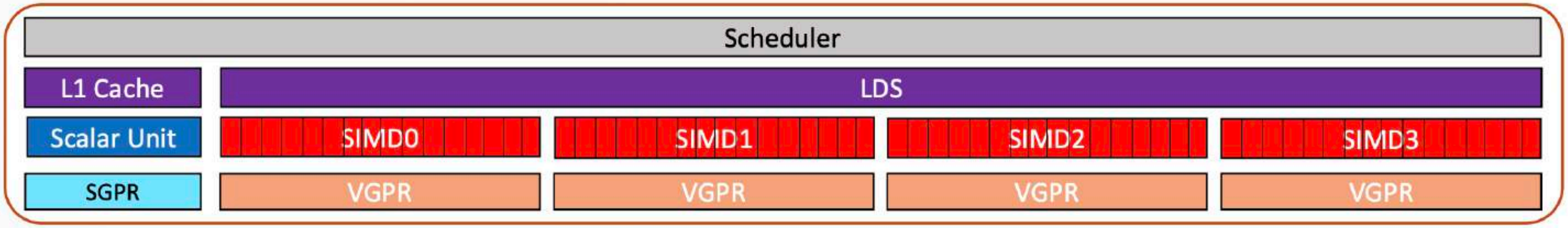


Terminology[术语]

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.

Nvidia SM

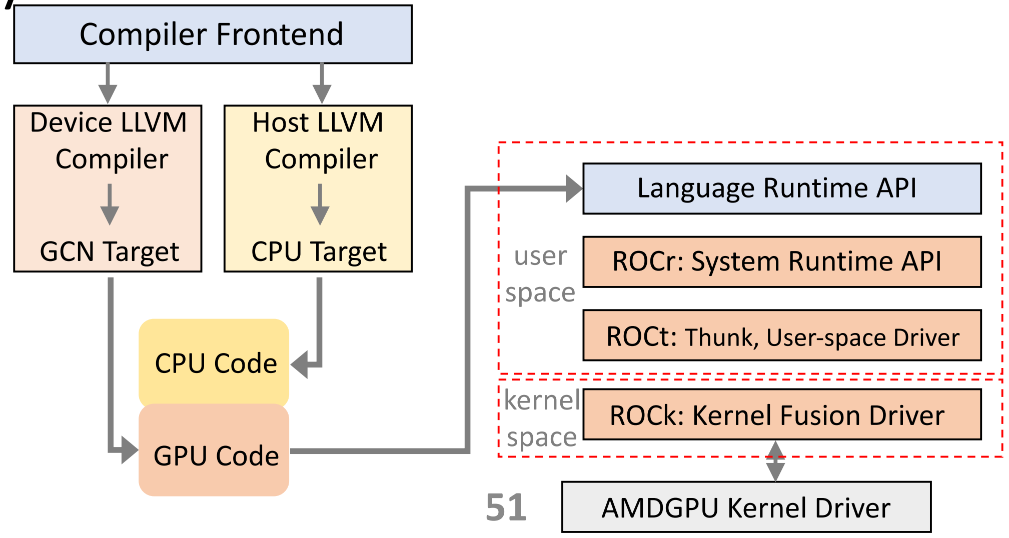




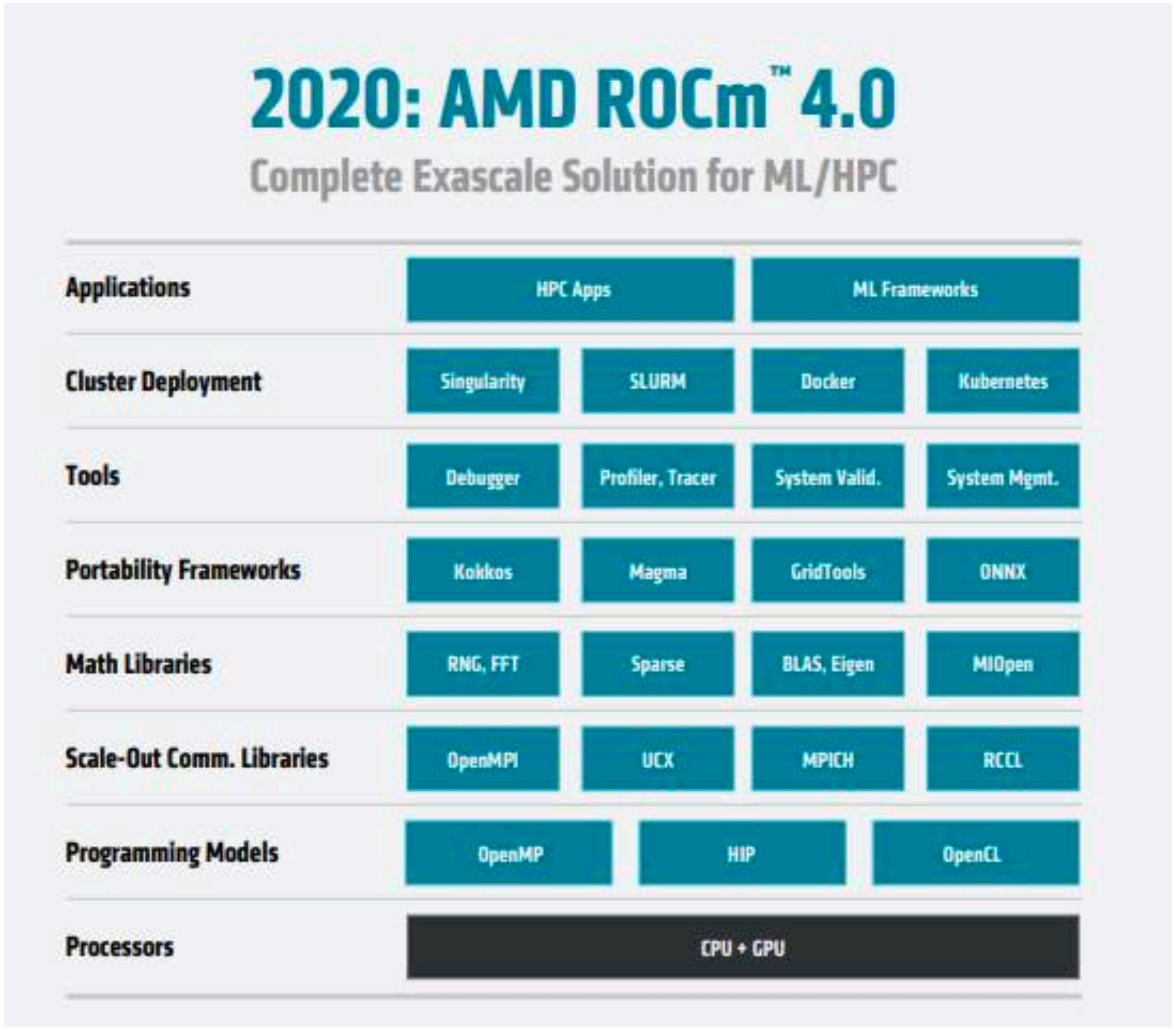
https://www.servethehome.com/nvidia-v100-volta-update-hot-chips-2017/

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/

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