北京理工大学
BEIJING INSTITUTE OF TECHNOLOGY

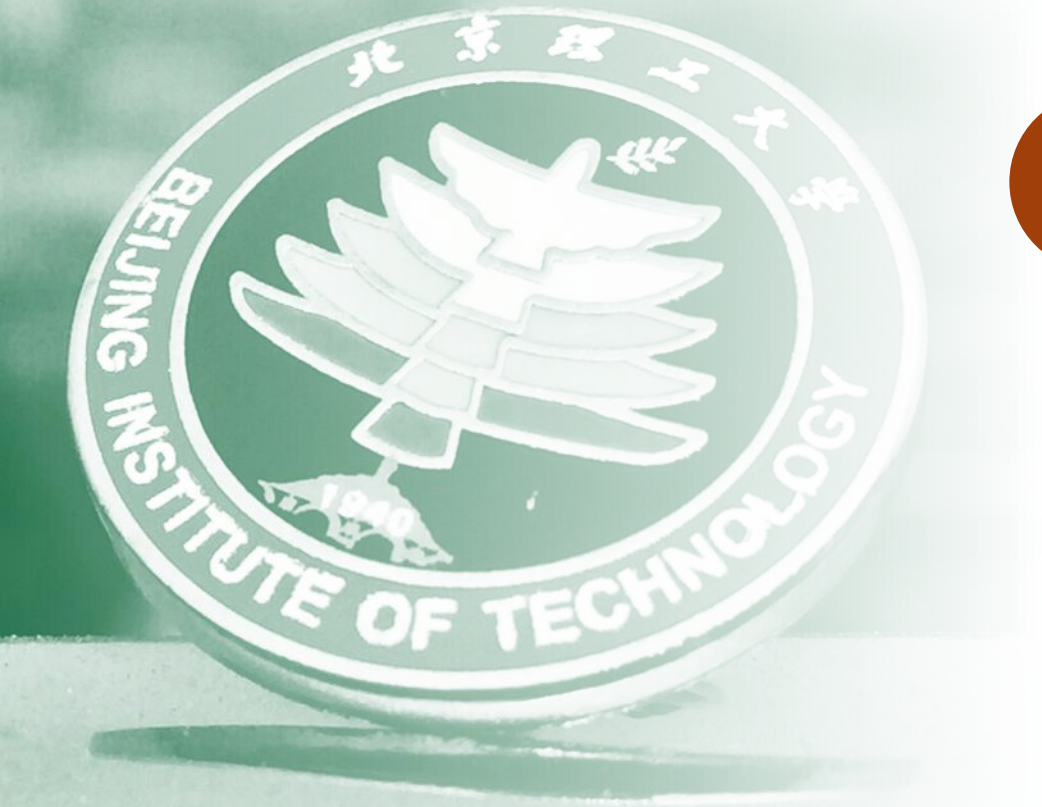
并行编程原理与实践

9. 异构计算概述

王一拙、计卫星

北京理工大学计算机学院

目录 | CONTENTS



1 什么是异构计算机系统

应用中的“异构”

- 现实世界中的大多数应用都有复杂的组成：
  - 某些部分容易并行化，某些部分很难并行化
  - 某些部分的数据访问模式可预测，某些部分不可预测
  - 某些部分是数据密集型的，某些部分是计算密集型的
  - 某些部分需要占用大量通信带宽，某些部分没有通信需求
  - ……
- 针对不同的应用特性配置不同的系统资源（用合适的工具做合适的事）

1 什么是异构计算机系统

- **计算平台是一个包含多种共享资源的系统**
  - Cores, caches, interconnects, memory, disks, power……
- **管理这些共享资源是十分困难的事**
  - What should be executed where with what resources?
- **通过异构系统并行编程模型（框架）来实现：**
  - 自动管理系统资源，给应用（应用的不同部分）配置适当的资源，最大化系统的运行效率
  - 降低程序员的负担

什么是异构计算机系统

- **计算平台应是一个异构（heterogeneous）和可配置（configurable）的系统**

C	C	C	C
C	C	C	C
C	C	C	C
C	C	C	C

同构

C1		C2	
		C3	
C4	C4	C4	C4
C5	C5	C5	C5

异构

- 同构：性能和能耗都不是最优的
- 异构：代码在最适合它的硬件资源上执行，使得性能和能耗达到最优

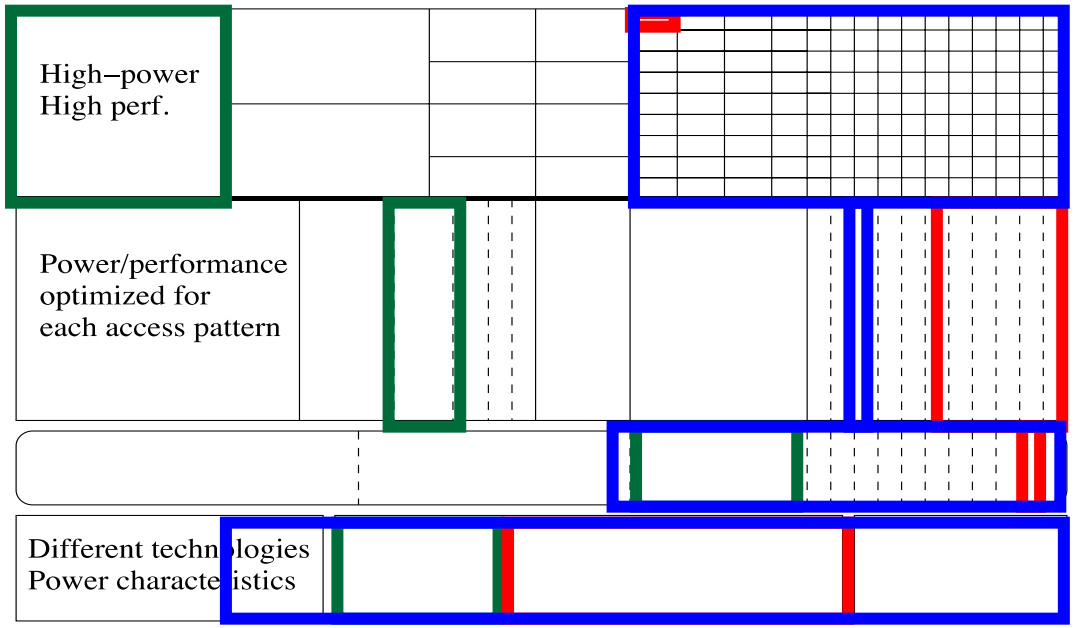
硬件资源异构、可配置

- 不同的计算/访存/通信资源配置来满足不同的能耗/性能/可靠性需求

Phase 1

Phase 2

Phase 3



Asymmetric / configurable cores and accelerators

Asymmetric / partitionable memory hierarchies

Asymmetric / partitionable interconnect

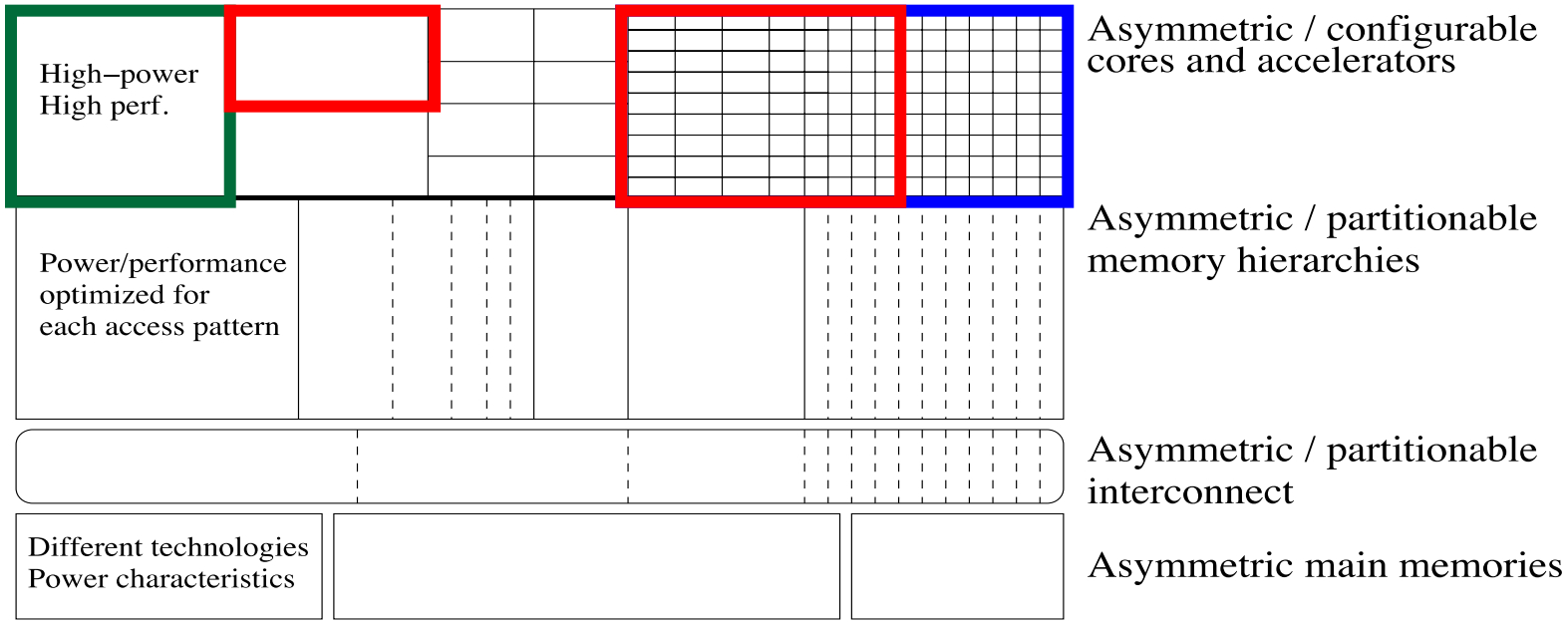
Asymmetric main memories

异构无处不在

Prof. Onur Mutlu, Carnegie Mellon University, 740: Computer Architecture

- **软件如何适应异构、可配置的硬件？**
  - 对不同硬件/ISA开发不同的软件/模块版本

Version 1
Version 2
Version 3

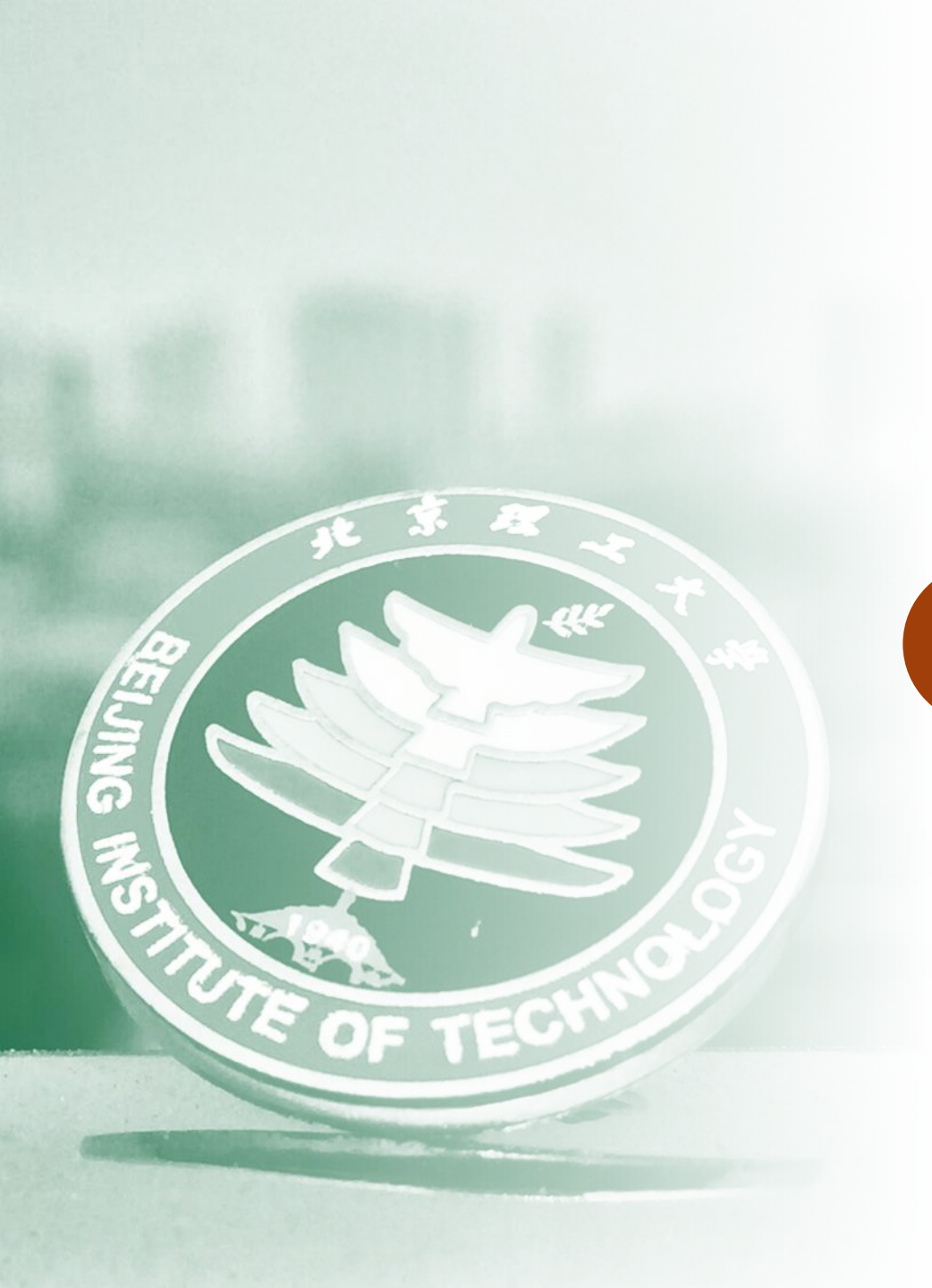


Prof. Onur Mutlu, Carnegie Mellon University, 740: Computer Architecture

从并行编程的角度看，异构存在于系统中的各个层次

Hardware level	Software (programming) level
Core	• Assembler • SIMD, AVX • Compiler • Libraries, Frameworks
Socket: Multicore	• Threads – Pthreads, OpenMP, TBB, Cilk Plus, … • Distributed memory model like MPI or GAS Languages • Libraries, Frameworks
Node	• Threads – Pthreads, OpenMP, TBB, Cilk Plus, … • Distributed memory model like MPI or GAS Languages • *Memory-Thread affinity becomes much more important* • Libraries, Frameworks
System	• Distributed memory model like MPI or GAS Languages • Libraries, Frameworks

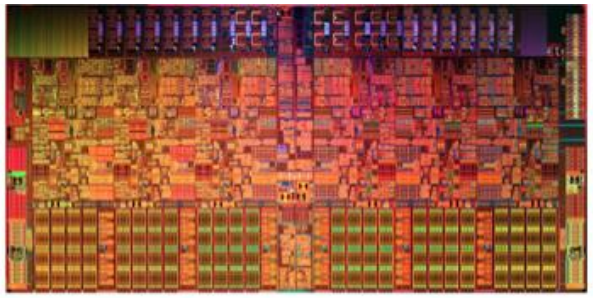
Introduction to Parallel Computing, University of Oregon, IPCC



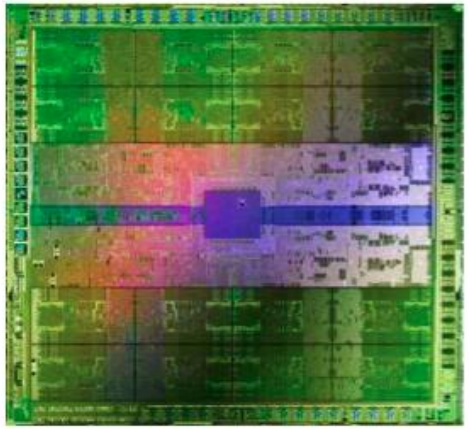
2 CPU+GPU异构系统

CPU vs. GPU

Specifications	Westmere-EP	Fermi (Tesla C2050)
Processing Elements	6 cores, 2 issue, 4 way SIMD @3.46 GHz	14 SMs, 2 issue, 16 way SIMD @1.15 GHz
Resident Strands/ Threads (max)	6 cores, 2 threads, 4 way SIMD: 48 strands	14 SMs, 48 SIMD vectors, 32 way SIMD: 21504 threads
SP GFLOP/s	166	1030
Memory Bandwidth	32 GB/s	144 GB/s
Register File	6 kB (?)	1.75 MB
Local Store/L1 Cache	192 kB	896 kB
L2 Cache	1536 kB	0.75 MB
L3 Cache	12 MB	-
# transistors & area:	12 亿, 240 mm^2	30亿, 520 mm^2
thermal design power:	130 Watts	160+ Watts? (240 W/card)



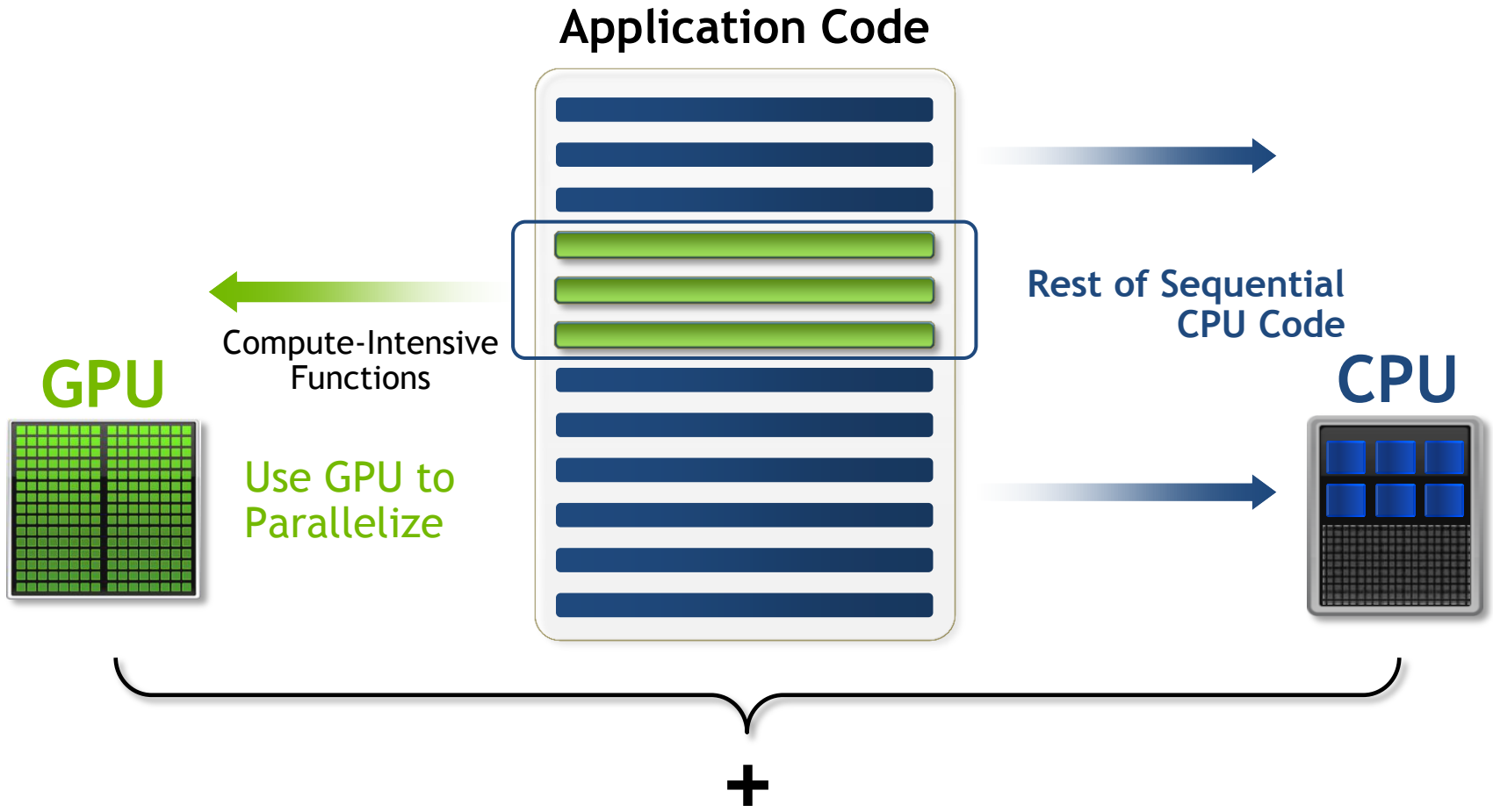
Westmere-EP (32nm)



Fermi (40nm)

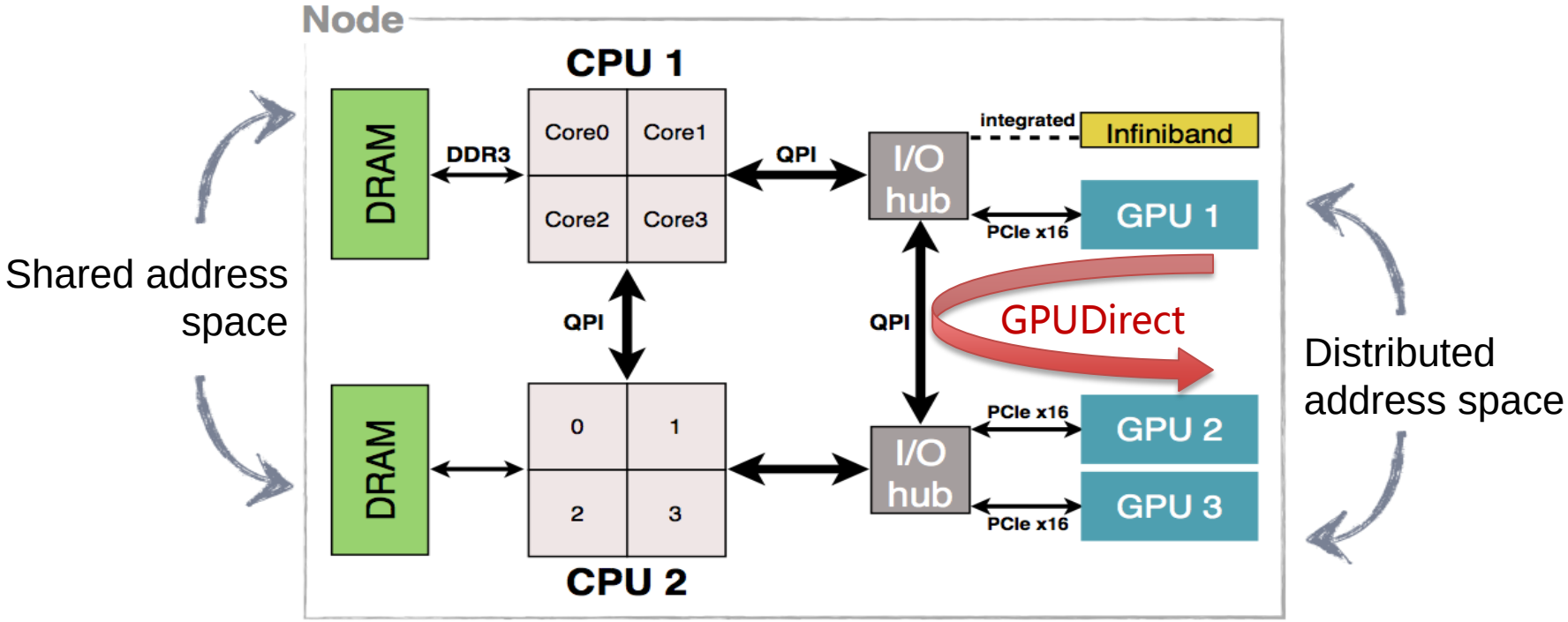
CPU+GPU异构系统

CPU+GPU：加速应用



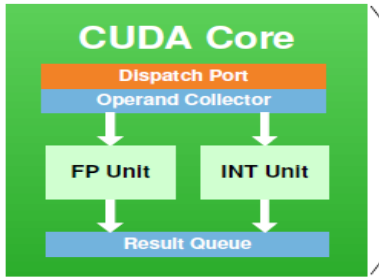
CPU+GPU：互连

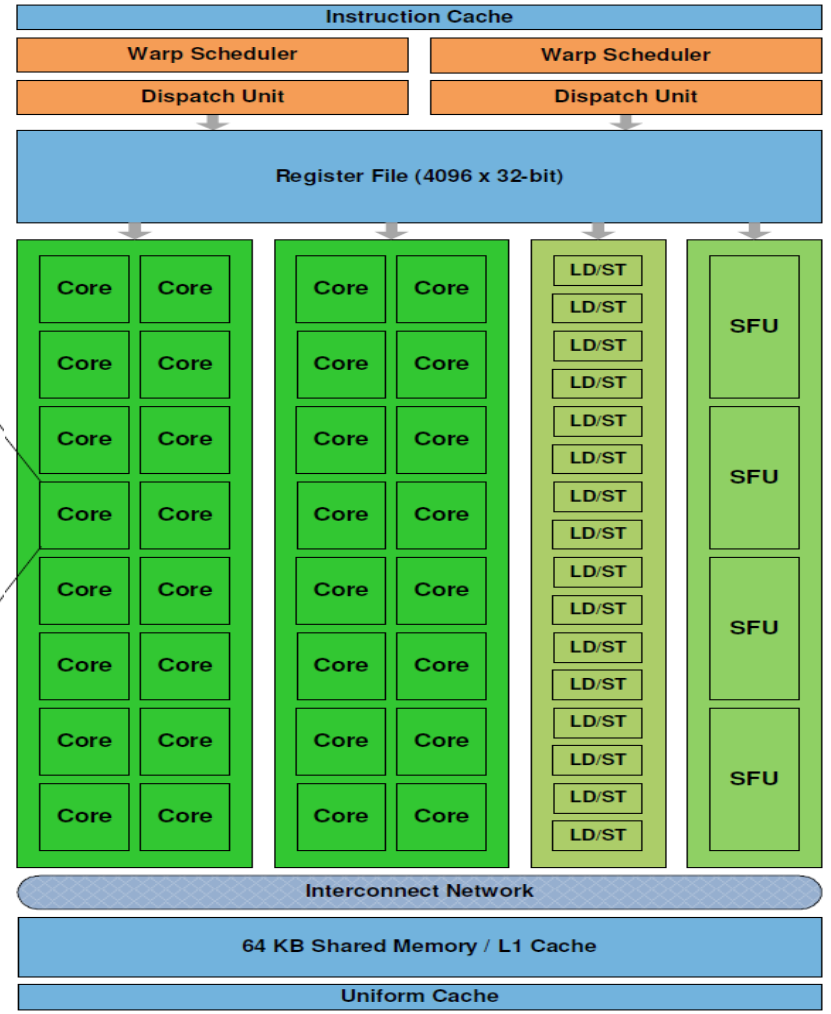
- CPU与独立GPU之间通过PCIe连接
- GPUDirect: GPU之间的P2P数据访问



2010 - NVIDIA Fermi

- Fermi是第一个完整的GPU计算架构，首款可支持与共享存储结合纯cache层次的GPU架构
- 30亿晶体管，40nm
- 16个SM，共512个CUDA核
- 384b GDDR5, 6 GB, 178 GB/s带宽
- 寄存器文件、一级缓存、二级缓存、DRAM全部支持ECC错误校验

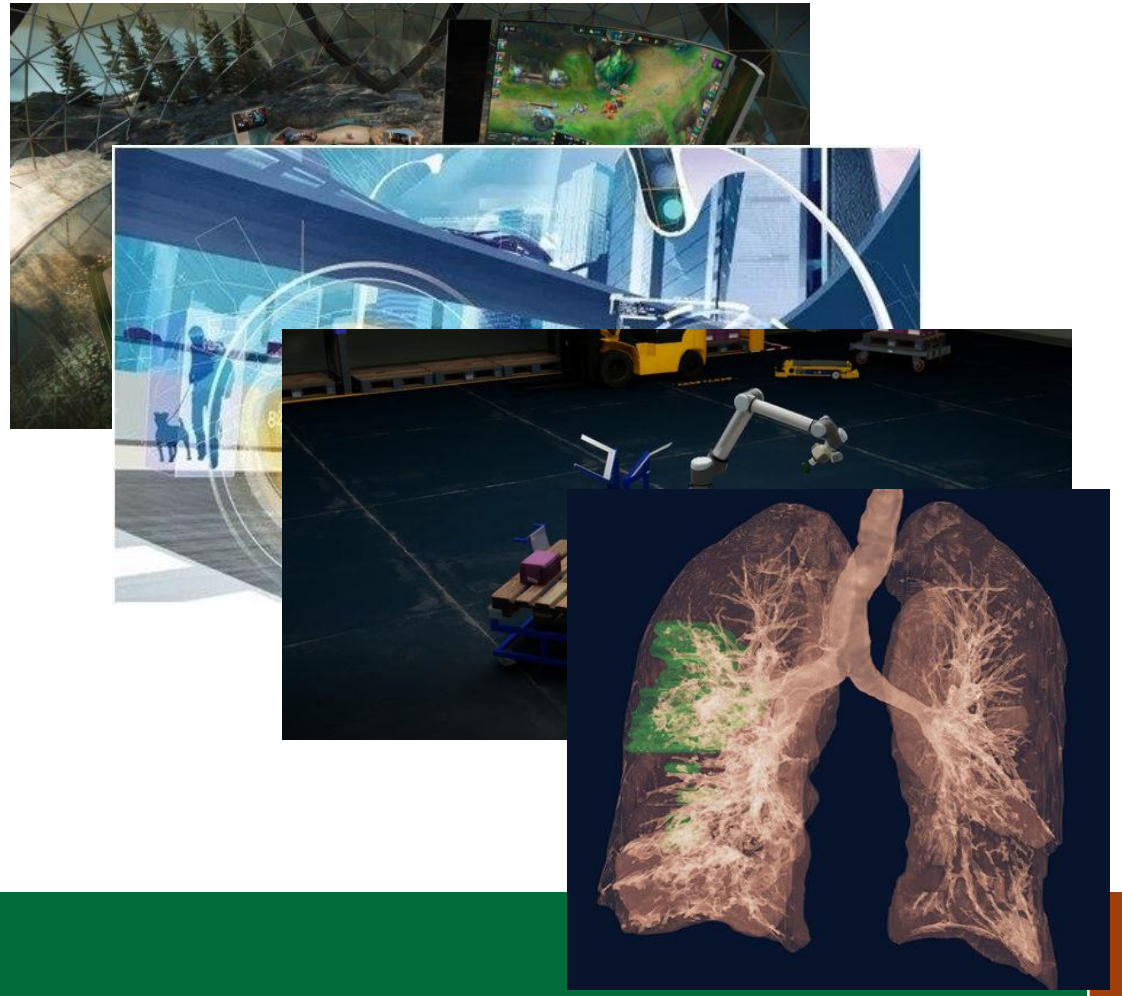




2 CPU+GPU异构系统

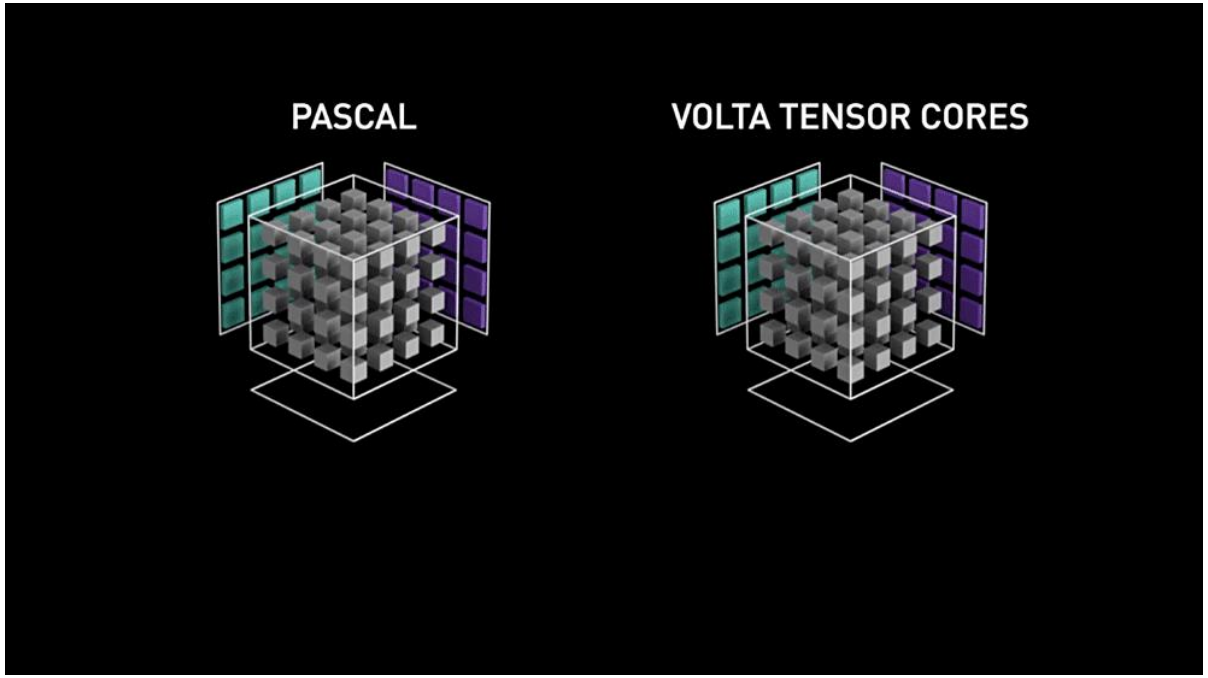
2020 - NVIDIA Tesla V100

- 加速当今时代的重要工作：NVIDIA A100 Tensor Core GPU 可针对 AI、数据分析和 HPC 应用场景，在不同规模下实现出色的加速，有效助力全球高性能弹性数据中心。
- 第一个基于NVIDIA第8代GPU架构Ampere架构的GPU
- 540亿晶体管，7nm
- 128个SM，8192个FP32 CUDA核心，4096个FP64 CUDA核心、512个Tensor核心
- 40GB, 1555GB/s带宽
- 全球最先进的AI系统—NVIDIA DGX A100系统，单节点AI算力达到创纪录的5 PFLOPS



CPU+GPU异构系统

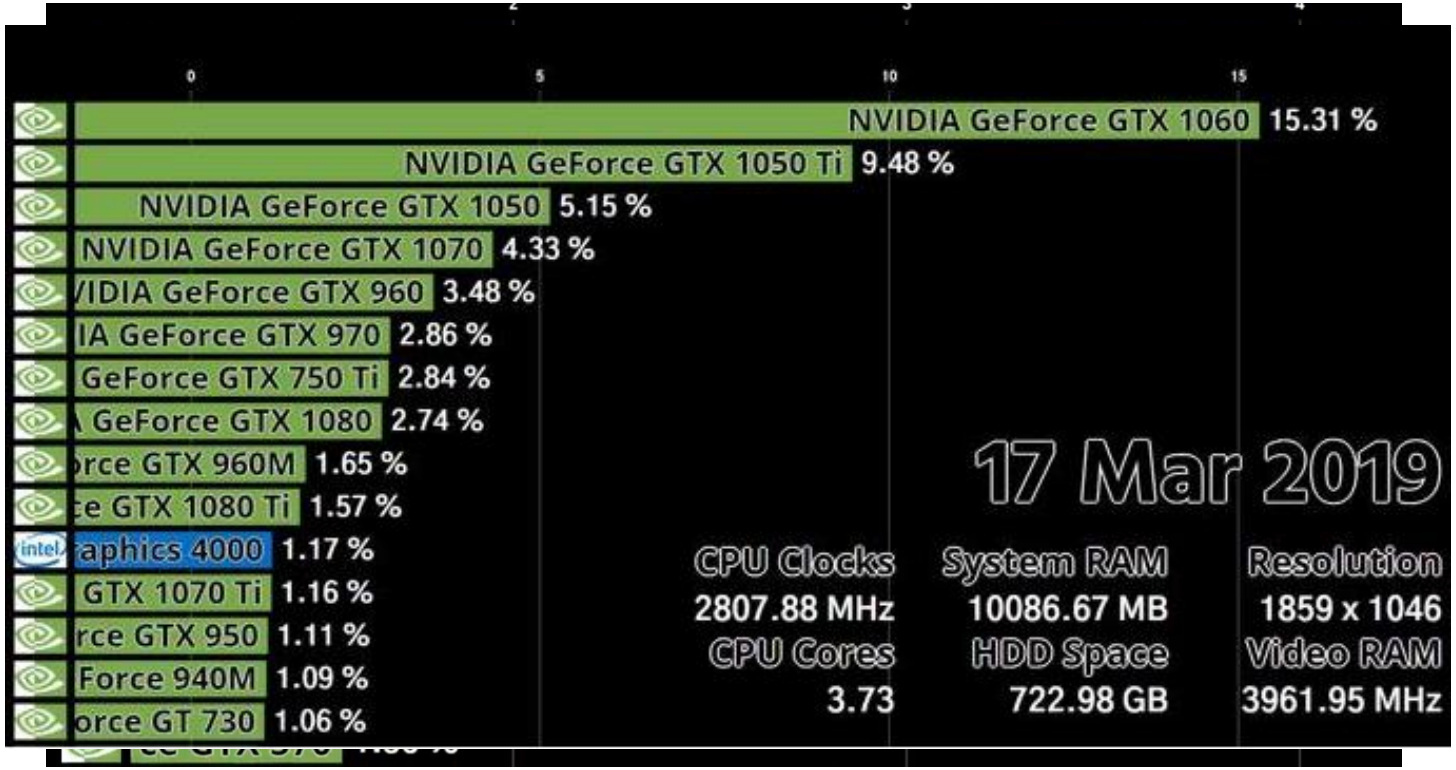
- 2020 - NVIDIA Tesla V100

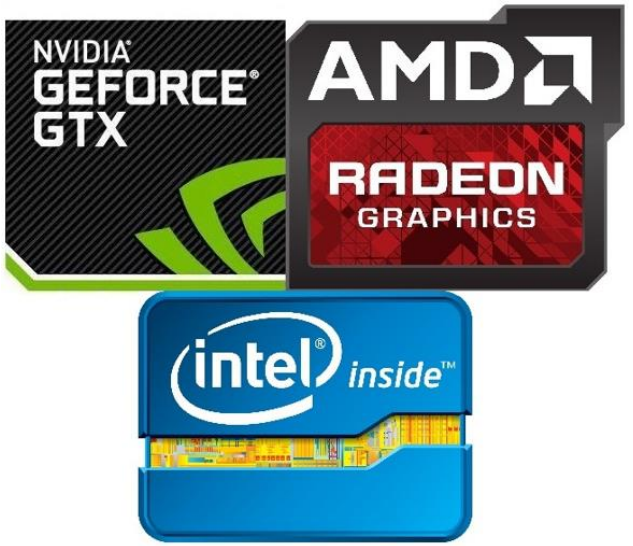


NVIDIA Tesla系列GPU发展历程

	K40	M40	P100	V100	A100
发布时间	2013.11	2015.11	2016.4	2017.05	2020.05
架构	Kepler	Maxwell	Pascal	Volta	Ampere
制程	28 nm	28 nm	16 nm	12 nm	7 nm
晶体管数量	71亿	80亿	153亿	211亿	510亿
Die Size	551 mm^2	601 mm^2	610 mm^2	815 mm^2	826 mm^2
最大功耗	235W	250W	300W	300W	400W
SM数量	15	24	56	80	108
Tensor核数量	NA	NA	NA	640	432
FP64 CUDA核数量	960	960	1792	2560	3456
FP32 CUDA核数量	2880	3072	3584	5120	6912
FP64峰值算力	1.68 TFLOPS	2.1 TFLOPS	5.3 TFLOPS	7.8 TFLOPS	9.7 TFLOPS
FP32峰值算力	5.04 TFLOPS	6.08 TFLOPS	10.6 TFLOPS	15.7 TFLOPS	19.5 TFLOPS

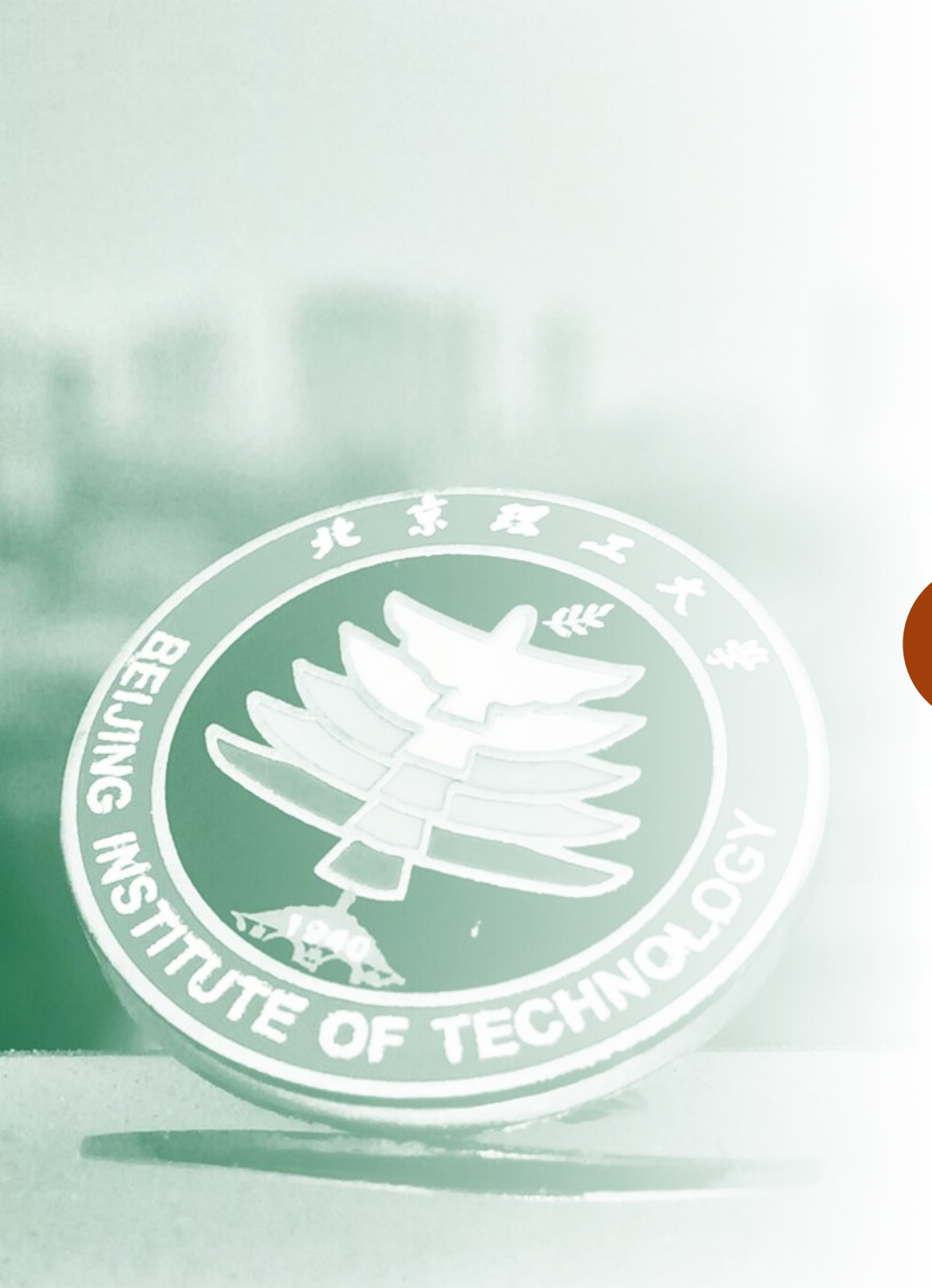
GPU市场状况


0
5
10
15
NVIDIA GeForce GTX 1060 15.31 %
NVIDIA GeForce GTX 1050 Ti 9.48 %
NVIDIA GeForce GTX 1050 5.15 %
NVIDIA GeForce GTX 1070 4.33 %
VIDIA GeForce GTX 960 3.48 %
IA GeForce GTX 970 2.86 %
GeForce GTX 750 Ti 2.84 %
GeForce GTX 1080 2.74 %
orce GTX 960M 1.65 %
e GTX 1080 Ti 1.57 %
raphics 4000 1.17 %
GTX 1070 Ti 1.16 %
rce GTX 950 1.11 %
Force 940M 1.09 %
orce GT 730 1.06 %
17 Mar 2019
CPU Clocks
2807.88 MHz
System RAM
10086.67 MB
Resolution
1859 x 1046
CPU Cores
3.73
HDD Space
722.98 GB
Video RAM
3961.95 MHz


NVIDIA
GEFORCE
GTX
AMD
RADEON
GRAPHICS
intel
inside

CPU片上集成GPU

- AMD Accelerated Processing Unit (APU)
  - AMD 锐龙 5000
  - 7nm“Zen 3”架构
  - 集成Radeon显卡
  - Zen 3代表着AMD在更高单核性能、更高能效和更低延迟方面的不懈努力。强大核心打造先进游戏处理器。

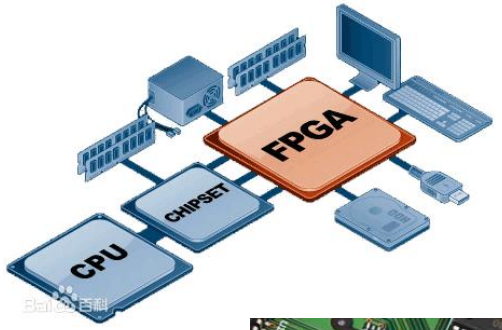


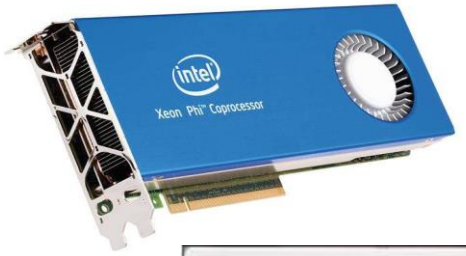
3 异构系统的选择和挑战

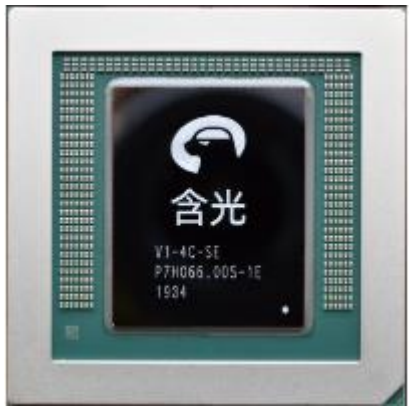
异构系统的选择和挑战

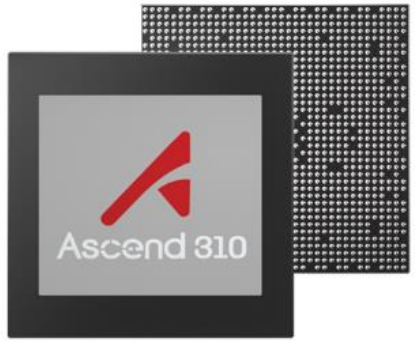
异构系统的种类

- CPU+协处理器
- CPU+众核处理器（MIC）
- CPU+GPU
- CPU+FPGA
- CPU+ASIC
- CPU+DSP
- CPU+AI芯片





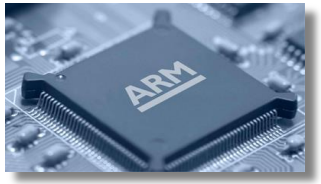




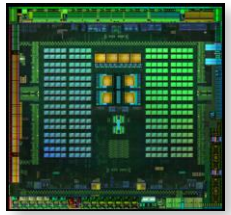
选择适合的芯片

- 考虑应用场景、性能、功耗、可靠性、成本等

CPU



GPU



DSP

HEXAGON™

Texas Instruments

FPGA等可编程逻辑电路

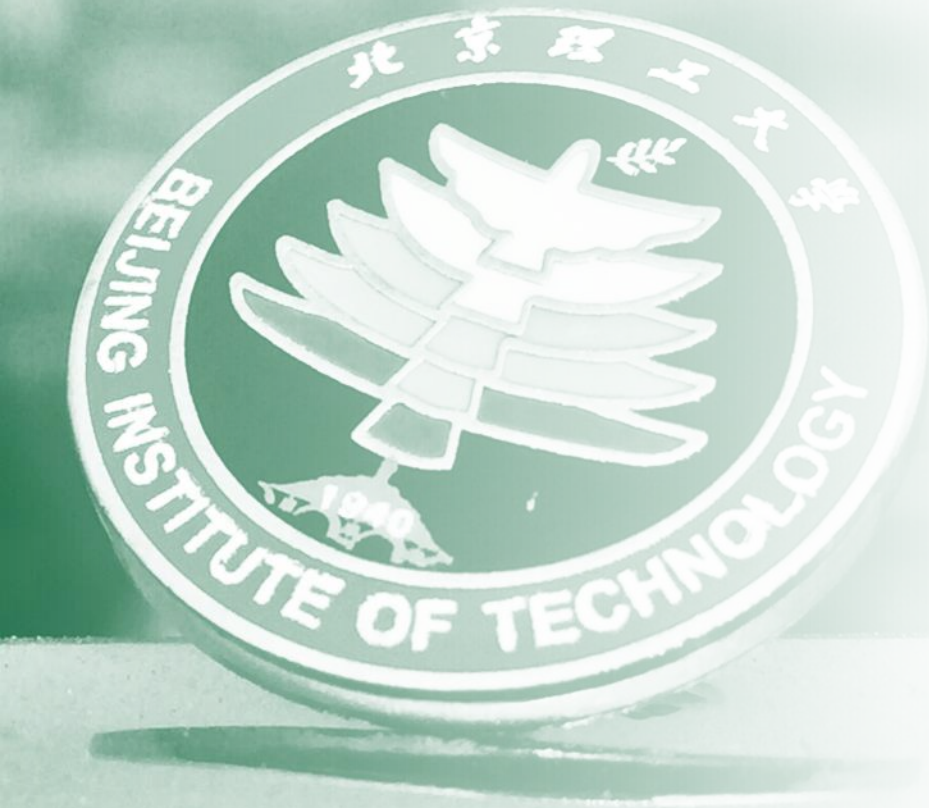
ASIC

视频编解码器等专用芯片

编程容易 → 性能高

异构系统的选择和挑战

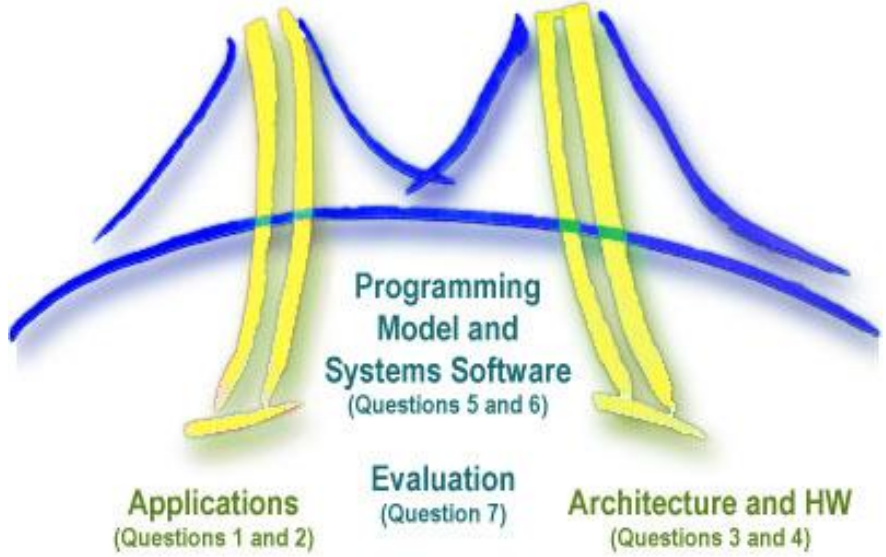
- **异构系统：让每个任务运行在最适合它的处理器上**
  - 对硬件设计的挑战：
    - 如何选择异构的硬件资源来构建系统？如何实现异构资源的衔接？如何进行资源的配置和管理？
  - 对软件设计的挑战：
    - 如何确定适合软件运行的硬件？任务的划分、调度、同步等相比同构系统更加复杂。软件的移植和维护更加困难。异构的硬件资源可能会限制算法选择。
  - 软硬件协同设计越来越重要



4 异构并行编程模型

异构并行编程模型（框架）的作用

- 异构并行编程模型是异构系统与上层应用之间的桥梁
- 为程序员提供一个合理的编程接口，使其在编程时既可以充分利用丰富的异构资源、又不必考虑复杂的硬件细节
- 异构并行编程模型依靠编程接口以及编译/运行时系统解决任务划分、调度、数据分布、同步与通信等并行编程的关键问题



常见异构并行编程模型

- CUDA — Nvidia
- OpenCL — Khronos Group
- C++ AMP — Microsoft
- OpenACC — Nvidia, PGI, Cray
- OpenMP4.0 — OpenMP Architecture Review Board
- ROCm — AMD

CUDA (Compute Unified Device Architecture)

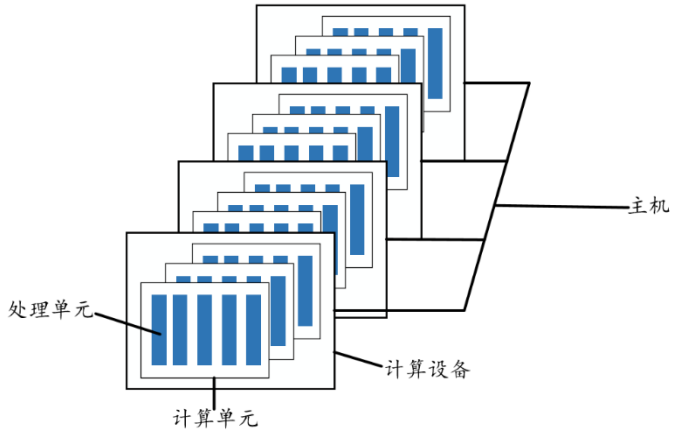
- NVIDA针对GPU推出的编程模型
- 是对流行编程语言的扩展（C/C++/Fortran）
- CUDA 架构由主机（host）和设备（device）组成，CPU （host）负责进行逻辑复杂的串行计算，而GPU进行数据并行的计算（device）
- CUDA编程模型提供了对GPU硬件资源的（流式处理器硬件线程、GPU内存）一个软件抽象层
- CUDA支持Linux和Windows操作系统，有完善的开发环境和文档

CUDA 示例

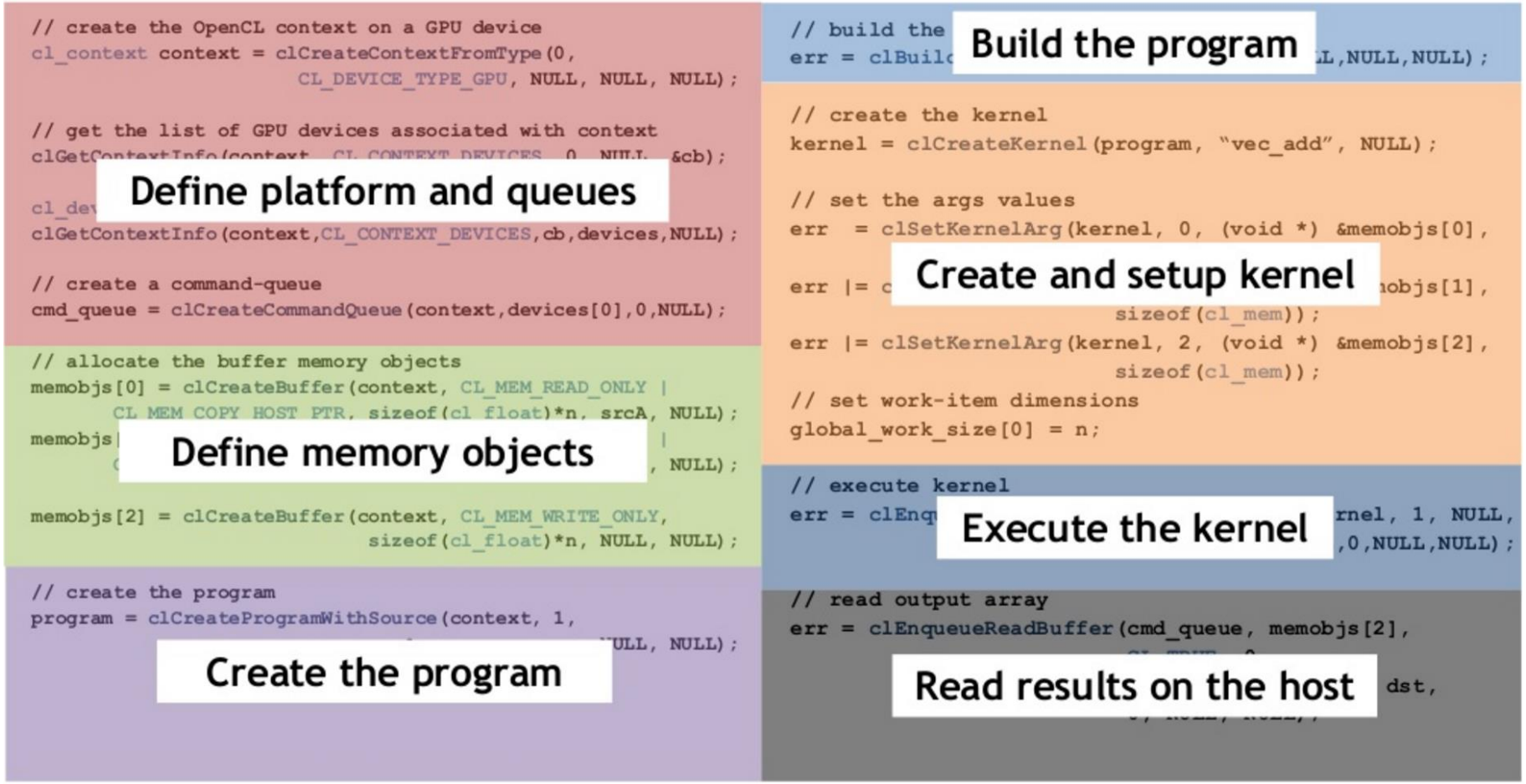
```
void main(){
    float *a, *b, *out;
    float *d_a;

    a = (float*)malloc(sizeof(float) * N);

    // Allocate device memory for a
    cudaMalloc((void**)&d_a, sizeof(float) * N);

    // Transfer data from host to device memory
    cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);

    …
    vector_add<<<1,1>>>(out, d_a, b, N);
    …

    // Cleanup after kernel execution
    cudaFree(d_a);
    free(a);
}
```

OpenCL (Open Computing Language)

- OpenCL是非营利性技术联盟Khronos Group维护的开放标准
- 是面向各种异构系统的（CPU, GPU, DSP, FPGA, ASIC）统一编程框架
- 2008年由苹果提出，随后AMD、IBM、Intel和NVIDIA等公司开始参与，发布OpenCL标准规范，目前OpenCL 3.0，各主流硬件厂商都有自己的OpenCL实现
- OpenCL由一门用于编写kernels （在OpenCL设备上运行的函数）的语言（基于C99）和一组用于定义并控制平台的API组成
- OpenCL提供了基于任务和基于数据两种并行计算机制
- OpenCL将异构系统抽象为主机连接一个或多个OpenCL设备，设备又可分成一个或多个计算单元（CU），每个计算单元又可进一步分成一个或多个处理单元 (PE)



OpenCL 示例


// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(0,
CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);
// get the list of GPU devices associated with context
clGetContextInfo(context,CL_CONTEXT_DEVICES,cb,devices,NULL);
// create a command-queue
cmd_queue = clCreateCommandQueue(context,devices[0],0,NULL);
Define platform and queues
// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA, NULL);
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
sizeof(cl_float)*n, NULL, NULL);
Define memory objects
// create the program
program = clCreateProgramWithSource(context, 1,
Create the program
// build the
err = clBuild
Build the program
// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);
// set the args values
err = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2],
sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;
Create and setup kernel
// execute kernel
err = clEnq
Execute the kernel
// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
Read results on the host
dst,

C++ AMP (Accelerated Massive Parallelism)

- 由微软开发的加速C++程序的并行计算库，是C++的扩展，目前只支持Windows平台
- 是基于DirectX 11技术实现的一个并行计算库和开放的编程规范，使C++程序员可以很容易的编写运行在GPU上的并行程序
- 使用C++ AMP一般涉及三步
  - 创建array_view对象
  - 调用parallel_for_each函数
  - 通过array_view对象访问计算结果
- array_view模板类管理数据的移动，在kernel中用到的时候进行数据的拷贝
- parallel_for_each是启动kernel的入口，函数入口需要指定线程任务划分和执行的kernel
- C++ AMP会自动处理显存的分配和释放、GPU线程的分配和管理

OpenACC (for open accelerators)

- 由OpenACC组织于2011年推出的众核加速编程标准
- 编译指示(compiler directive)：插入特殊的编译指示实现并行加速
- 编译指示的基本语法：#pragma acc <directive> [clause[[,] clause…]
- 常用的directives：parallel, kernels, data, host_data, wait, update, loop……
- 常用的clauses：private, reduction, local, copyin, copyout, num_workers……
- 三级并行机制：gang、worker、vector
- 运行时库函数：acc_get_num_devices, acc_init, acc_malloc, acc_free, acc_init, acc_async_wait……
- PGI编译器支持多核CPU、Nvidia GPU、Intel MIC上的OpenACC编程

OpenACC 示例

SAXPY in C

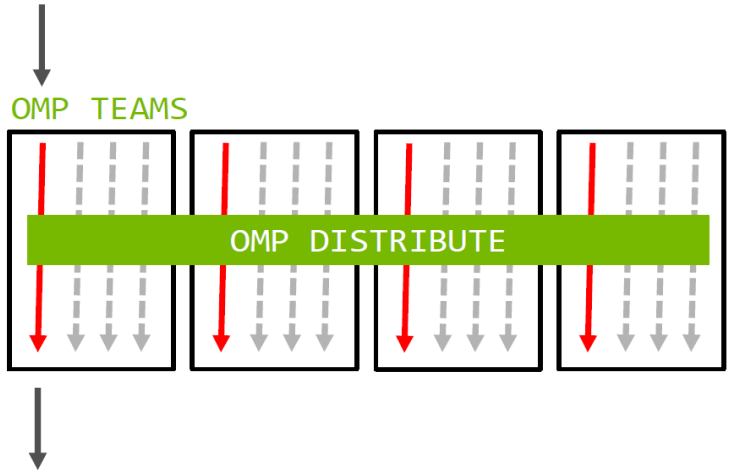
```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc parallel loop
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

SAXPY in Fortran

```
subroutine saxpy(n, a, x, y)
  real :: x(n), y(n), a
  integer :: n, i

  !$acc parallel loop
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
  !$acc end parallel loop
end subroutine saxpy

...
! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x, y)
...
```

OpenMP (Open Multi-Processing)

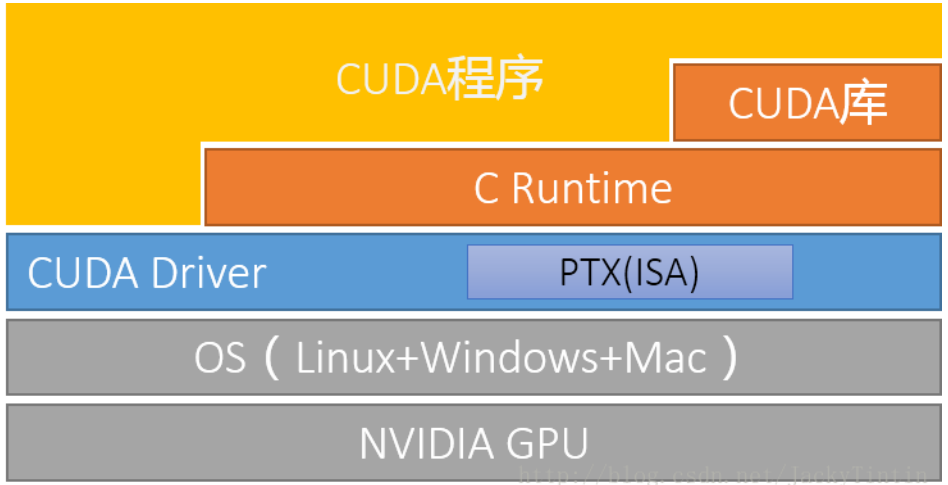
- OpenMP 4.0 开始支持加速器装载和向量化指令
- OpenMP 5.0 开始完全支持加速器设备，包含主机和设备的统一共享内存
- 为了处理设备之间指令集和编程模式的不同，OpenMP 添加了target指令以及相关的语句（teams、distribute、data map…）和函数来适配这些设备
- 支持采用OpenMP进行GPU编程的编译器：
  - Clang
  - XL
  - GCC
  - Cray Compiler Environment



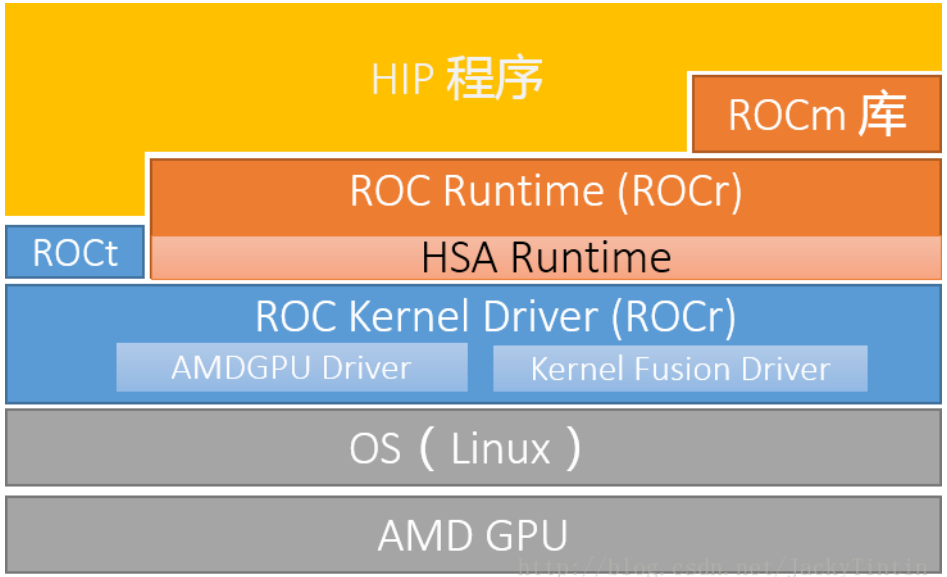
OpenMP 示例

```
error = 0.0;

#pragma omp target teams distribute \
            parallel for reduction(max:error)
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }
```

- **ROCm (Radeon Open Computing platforM)**
  - AMD建立的可替代 CUDA 的开源GPU计算生态
  - ROCm 复制了CUDA 的技术栈
  - 支持HIP, OpenCL, OpenMP编程



CUDA 技术栈

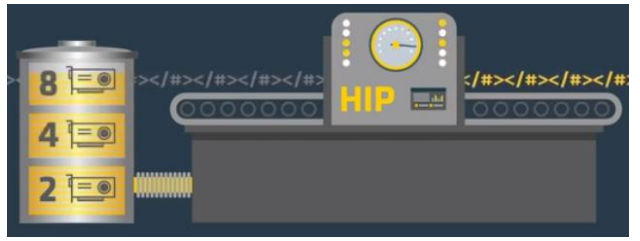


ROCm 技术栈

ROCm (Radeon Open Computing platforM)

- HIP（Heterogeous-compute Interface for Portability）C++运行时API和内核语言，类似于CUDA API
- 使用C++编程语言，支持模板，lambdas， 类，命名空间等
- 提供HIPFY工具，可以自动将CUDA代码转换为HIP代码，且性能几乎没有影响

```
hipMalloc(&A_d, Nbytes));
hipMalloc(&C_d, Nbytes));

hipMemcpy(A_d, A_h, Nbytes, hipMemcpyHostToDevice);

const unsigned blocks = 512;
const unsigned threadsPerBlock = 256;
hipLaunchKernel(vector_square,   /* compute kernel*/
                dim3(blocks), dim3(threadsPerBlock), 0/*dynamic shared*/, 0/*stream*/,
                C_d, A_d, N);  /* arguments to the compute kernel */

hipMemcpy(C_h, C_d, Nbytes, hipMemcpyDeviceToHost);
```




北京理工大学
BEIJING INSTITUTE OF TECHNOLOGY
谢谢!
德以明理 学以精工