GPU

朱虎明
西安电子科技大学
zhuhum@mail.xidian.edu.cn
网安大楼CII1008

高性能计算—软硬件协同计算

- DSP
- FPGA
- GPU
- TPU
- 多核处理器 （多任务） Pthread
- 多核处理器 OpenMP
- 共享存储的SMP OpenMP
- 大规模并行处理系统(MPP) MPI
- 计算机集群系统(Cluster) **MPI**
- 异构计算集群系统 MPI+X

并行计算框架比较

编程框架	出现时间	开放性	主要支持语言	支持硬件	编程难度	开源深度学习软件支持
CUDA	2007	企业私有	C，C++	NVIDIA GPU	容易	Caffe，TensorFlow，MxNet，CNTK，Torch，Theano
OpenCL	2008	API 标准	C，C++	GPU，CPU FPGA，DSP MIC	难	Caffe，Theano
OpenMP	1997	API 标准	C，C++，Fortran	CPU，MIC	容易	Theano
MPI	1992	API 标准	C，C++，Fortran	CPU，MIC	难	CNTK，S-Caffe
spark	2010	开源软件	Java，Scala，python	CPU	容易	Intel BigDL，SparkNet

GPU（图形处理器）

- 1999年的GeForce 256是第一块GPU芯片
- NVIDIA、AMD、Intel、高通、ARM
- 斯坦福大学计算机图形学实验室教授**Patrick Hanrahan**
- 计算机科学家、皮克斯动画工作室联合创始人**Edwin Catmull**
- **表彰他们对 3D 计算机图形学的贡献，以及这些技术对电影制作和计算机生成图像（CGI）等应用的革命性影响。**

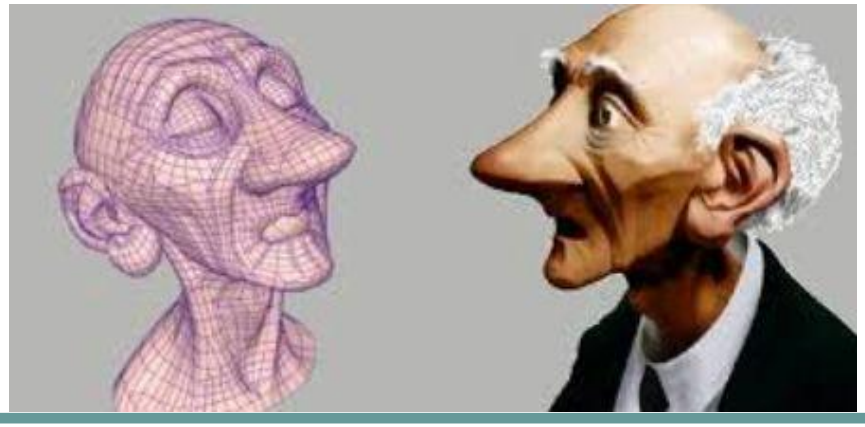
2019图灵奖

GPU

- **1986 年，史蒂夫·乔布斯（Steve Jobs）收购了卢卡斯电影公司（Lucas Pixar），并让 C**
- **Pat H nrahan 作为首**
- **Rend 其他电影公司。**
- **Hanrahan 促进了 OpenGL 等商业版本的发展**

GPU

- 47 部奥斯卡视觉效果提名影片中有 44 部使用了该软件，包括《阿凡达》、《泰坦尼克号》、《美女与野兽》、《魔戒电影三部曲》、《星球大战前传》
- **《阿凡达》中40%的画面由真实场景拍摄，60%完全由电脑动画生成，**拍摄立体画面使用的全新3D Fusion Camera系统也耗费了大量的成本。每帧画面平均耗费4万个人工小时



GPU通用计算发展历史

- 2000年，GPU上实现小波变换
- 2001年，矩阵运算
- 2002年，用细胞自动机(CA)仿真各种物理现象，加速光线跟踪算法
- 2003年，线性代数操作；Lattice Boltzmann的流体仿真；Level Set
- 2004年，数据库领域GPU加速；商业领域，支持GPU的视频工具
- 2006年，GeForce 8800诞生
- 2007年，GPU支持硬解码
- 2008年，CUDA架构，开始利用GPU的并行计算能力来加速视频编码
- 2009年，CUDA、OpenCL、DX11 Compute Shader百花齐放

AMD（ATI）——NVIDIA

- 黄仁勋1963/2/17在台湾台北出生
- 1984年毕业于俄勒冈州大学
- 1992 Nvidia

- ATI创办人**何国源**
  祖籍广东新会，
- 1950年出生
- 1974年毕台湾的成功大学
- 1985年多伦多市区ATI公司

DirectX 、 OpenGL 、 Vulkan

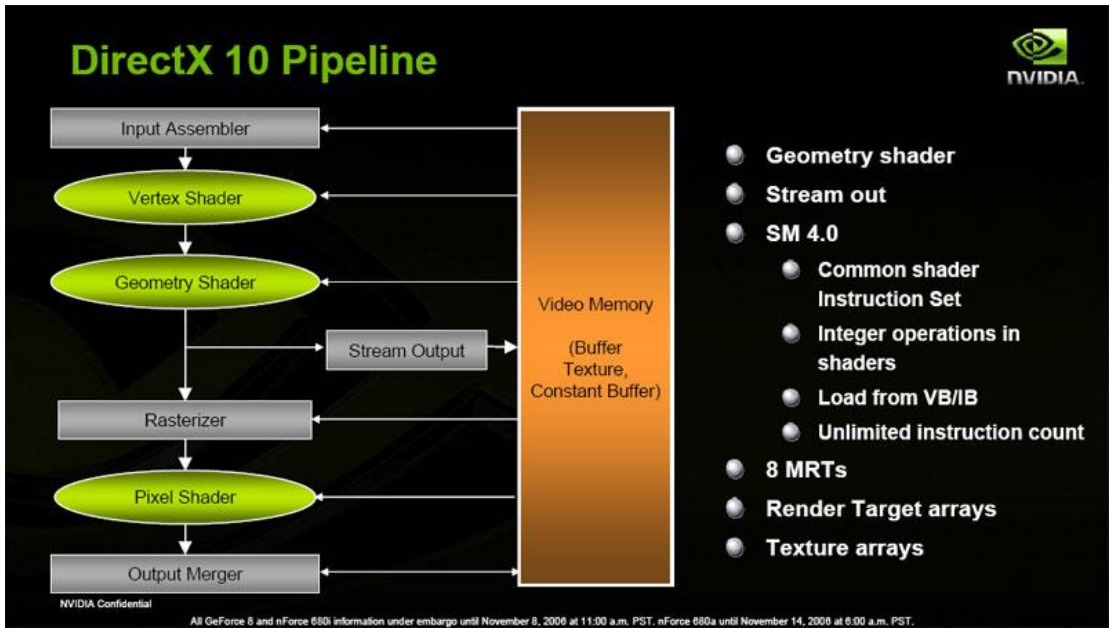
- 1.0 September 30, 1995
- 2.0 1996
- 3.0 September 15, 1996
- 4.0 Never launched
- 5.0 August 4, 1997
- 6.0 August 7, 1998
- 7.0 September 22, 1999
- 8.0 November 12, 2000
- 9.0 December 19, 2002
- 10.0 November 30, 2006
- 11 october 22, 2009
- 12 July 29, 2015Windows 10

OpenGL January 1992.
September 7, 2004
July 11, 2008
March 11, 2010
4.4 2013年7月23日

- **Vulkan：** 显式API， 2018年3月发布Vulkan 1.1； 2020年1月 V1.2

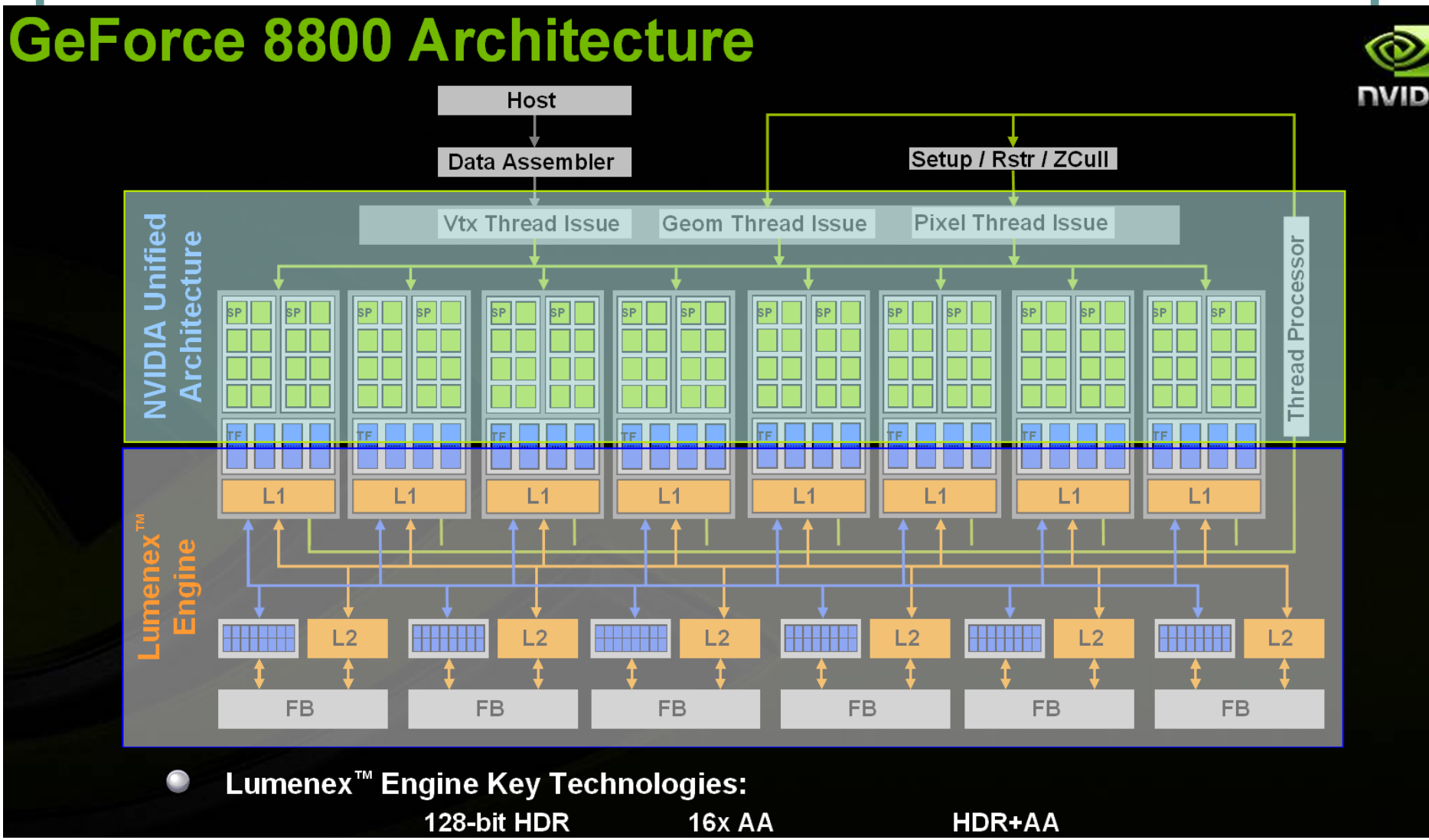
DirectX

- DirectX 7.0确定权威：核心技术T&L
- **DirectX 8.0：引入像素和顶点两大渲染管线 ,2001**
- **DirectX 10：统一渲染架构和几何着色**
- **DirectX 11.0：DirectCompute（通用计算）**



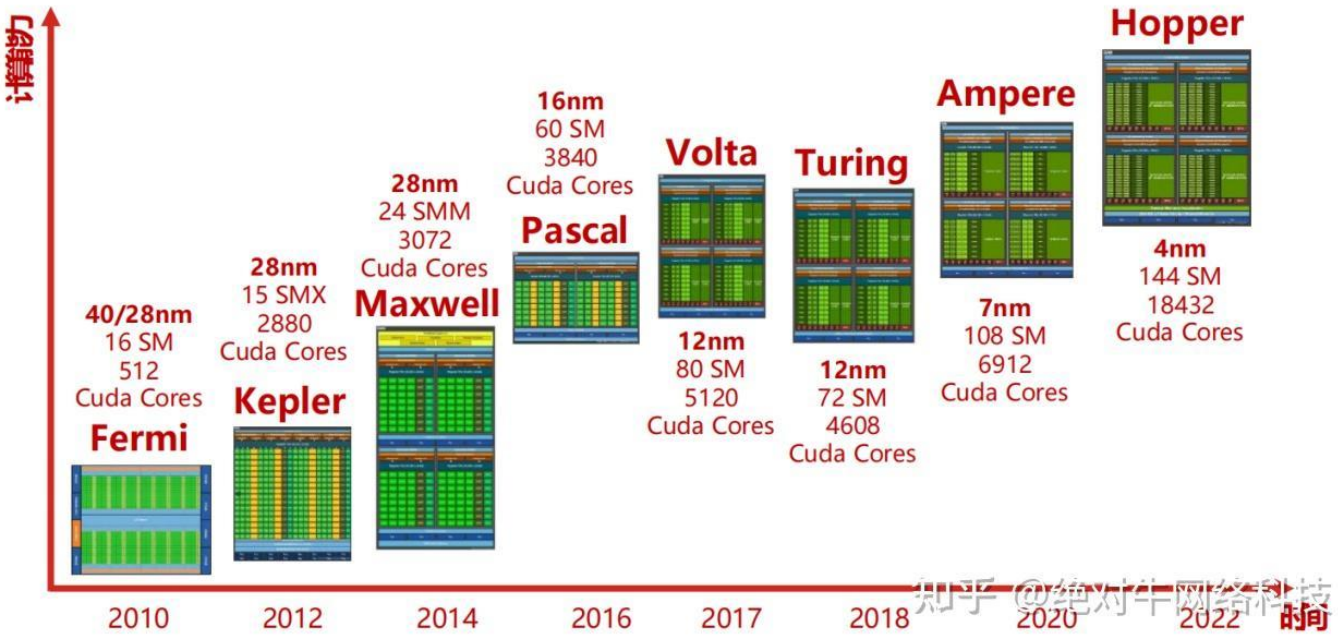
GeForce 8800

8个SM，每个有16个流数据处理器（SP，Stream Processing）



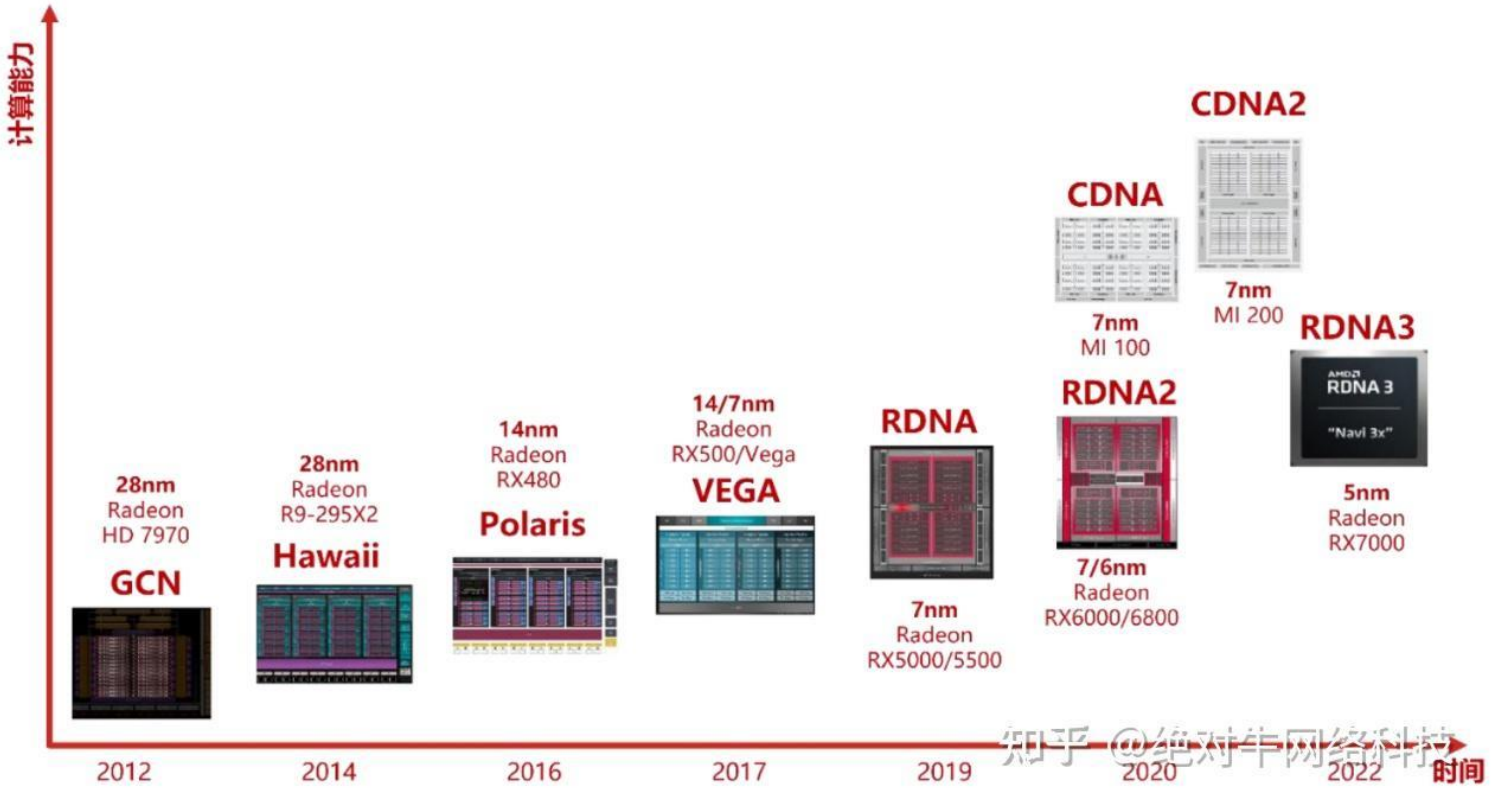
硬件架构（NVIDIA）

- 2008 G80
- 2010 Fermi
- 2012 Kepler
- 2014 Maxwell
- 2016 Pascal
- 2018 Volta
- 2020 Amper
- 2022 Hopper



硬件架构（AMD）

- AMD GPU硬件架构:
  - Southern Islands、Sea Islands、Volcanic Islands、Arctic Islands、**Vega**

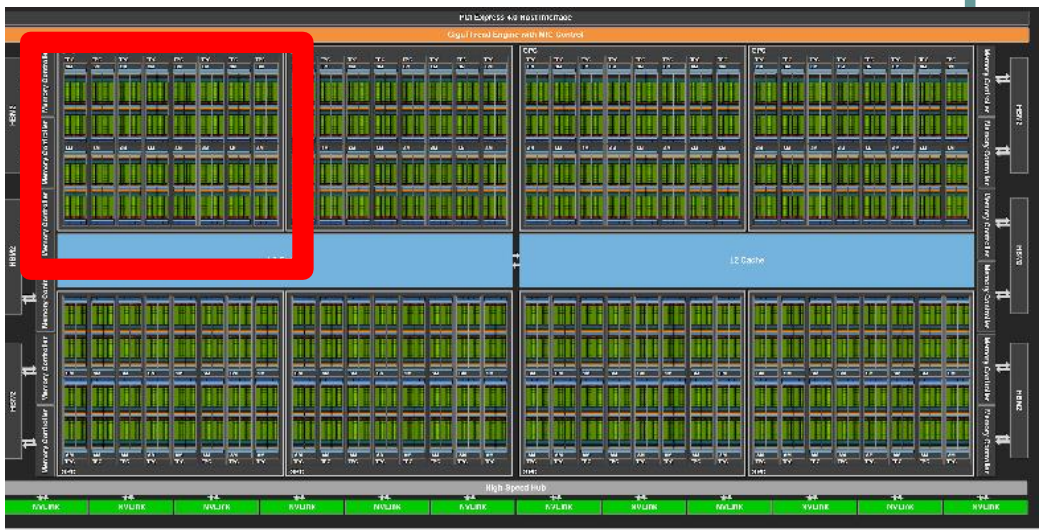


GPGPU并行编程模式(1/3)

- 图形API： OpenGL，Direct3D
- oneAPI:intel
- 通用计算编程接口
  - OpenCL:针对异构系统并行编程计算的API
    - 由Apple提出，支持(IBM, HP, NVIDIA)
    - 由Khronos组织维护
- NVIDIA CUDA架构
  - CUDA——Compute Unified Device Architecture

NVIDIA Tesla A100

- 128 SMs
- 64 FP32 CUDA Cores/SM,
- 6912 FP32 CUDA Cores per GPU
- 单精度19.5TFlops
- 双精度9.7TFlops
- INT8推理1248Tops
- INT8张量624Tops
- FP16张量312TFlops
- FP32张量156TFlops
- FP32训练312TFlops
- FP64高性能计算19.5TFlops。



AI 三要素算力——NVIDIA H100 GPU

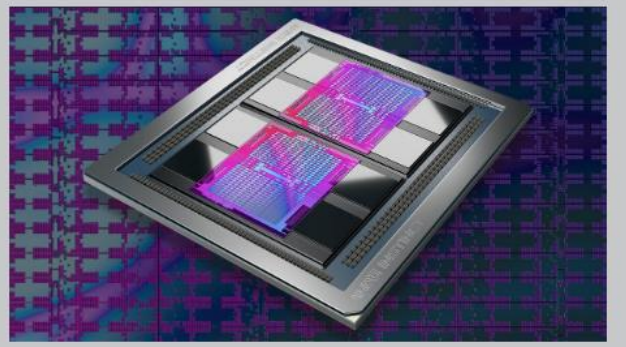
	H100	A100 (80GB)	V100
FP32 CUDA Cores	16896	6912	5120
Tensor Cores	528	432	640
Boost Clock	~1.78GHz (Not Finalized)	1.41GHz	1.53GHz
Memory Clock	4.8Gbps HBM3	3.2Gbps HBM2e	1.75Gbps HBM2
Memory Bus Width	5120-bit	5120-bit	4096-bit
Memory Bandwidth	3TB/sec	2TB/sec	900GB/sec
VRAM	80GB	80GB	16GB/32GB
FP32 Vector	60 TFLOPS	19.5 TFLOPS	15.7 TFLOPS
FP64 Vector	30 TFLOPS	9.7 TFLOPS (1/2 FP32 rate)	7.8 TFLOPS (1/2 FP32 rate)
INT8 Tensor	2000 TOPS	624 TOPS	N/A
FP16 Tensor	1000 TFLOPS	312 TFLOPS	125 TFLOPS
TF32 Tensor	500 TFLOPS	156 TFLOPS	N/A
FP64 Tensor	60 TFLOPS	19.5 TFLOPS	N/A
Interconnect	NVLink 4 18 Links (900GB/sec)	NVLink 3 12 Links (600GB/sec)	NVLink 2 6 Links (300GB/sec)

AI 三要素算力——NVIDIA H100 GPU

Transistor Count	80B	54.2B	21.1B
TDP	700W	400W	300W/350W
Manufacturing Process	TSMC 4N	TSMC 7N	TSMC 12nm FF
Interface	SXM5	SXM4	SXM2/SXM3
Architecture	Hopper	Ampere	Volta

AMD Instinct MI250X

- GPU Architecture： CDNA2
- Lithography： TSMC 6nm FinFET
- Stream Processors： 14,080
- Peak Half Precision (FP16) ： 83 TFLOPs
- Peak Single Precision (FP32) ： 47.9 TFLOPs
- Peak Double Precision (FP64) ： 7.9 TFLOPs
- Peak INT4 Performance： 383 TOPs
- Peak INT8 Performance： 383 TOPs
- Peak bfloat16： 383 TFLOPs



AMD CDNA™ 2 Architecture

Purpose-built for HPC and AI workloads.

AMD Infinity Architecture

Experience unified intelligence.

AMD ROCm™ - Ecosystem without Borders

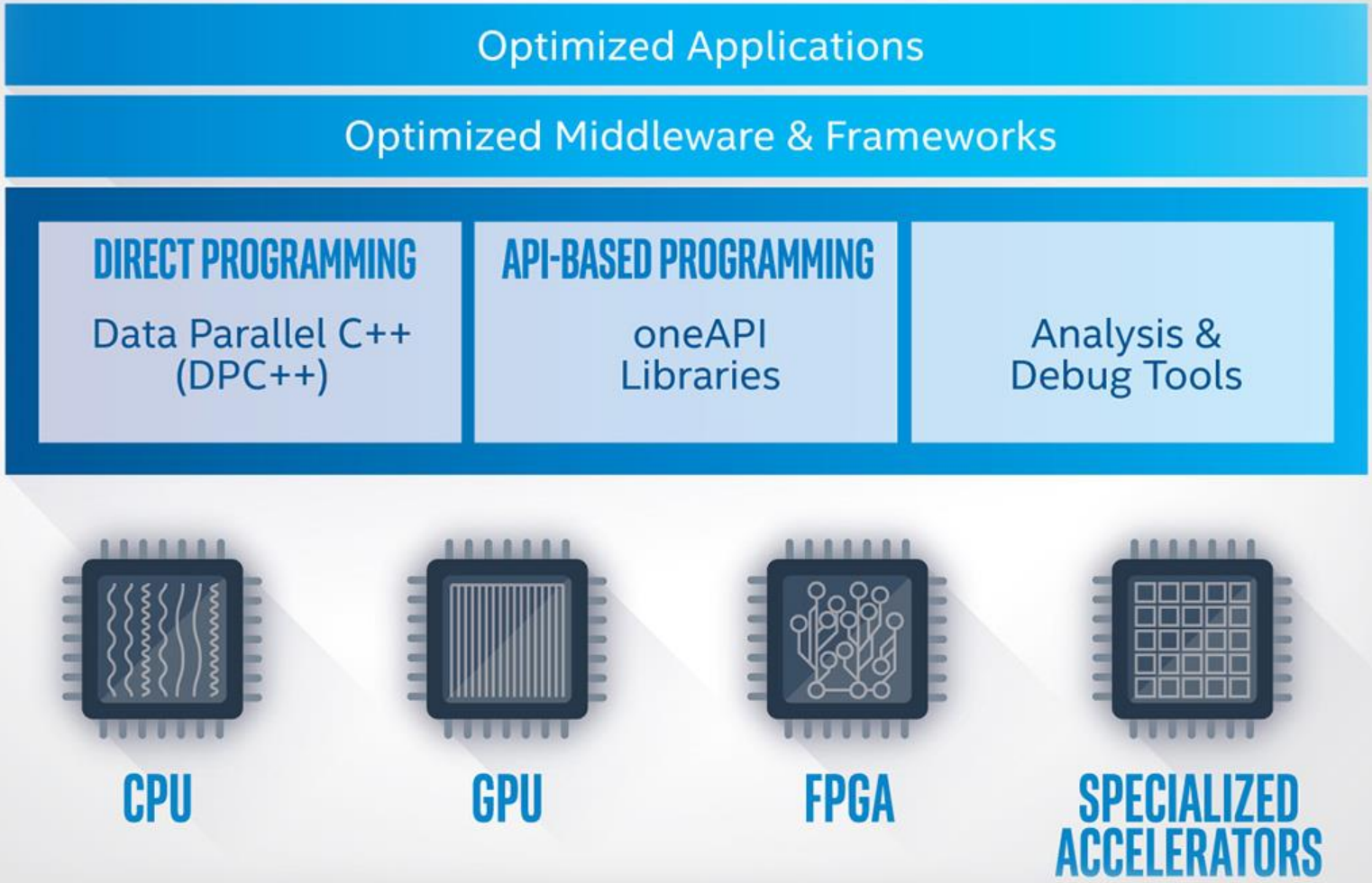
AMD ROCm™ open software platform.

intel ARC™ A770 Limited Edition

GPU Architecture	Alchemist
Xᵉ-cores / XMX Engines	32 / 512
Render Slices	8
Ray Tracing Units	32
Graphics Clock	2100 MHz
Memory Config	16GB GDDR6 @ 17.5Gbps
Memory Interface	256-bit
Memory Bandwidth	560 GB/s
System Interface	PCIe Gen 4.0 x 16
Power (TBP)	225W
Power Connector	1x 8-Pin 1x 6-Pin
HW Accelerated Media	AV1 (E&D), HEVC (E&D), H.264 (E&D), VP9 Bitstream & Decoding
Display Outputs	3x DP 2.0, 1x HDMI 2.1
Form Factor	10.5" Length, Dual Slot, Full Height
API Support	DirectX 12 Ultimate, OpenGL 4.6, OpenCL 3.0, Vulkan 1.3
OS Support	Win 10/11, Ubuntu
Intel Deep Link Technologies	Yes
Warranty	3-years

intel ARC™ A750 Limited Edition

GPU Architecture	Alchemist
Xᵉ-cores / XMX Engines	28 / 448
Render Slices	7
Ray Tracing Units	28
Graphics Clock	2050 MHz
Memory Config	8GB GDDR6 @ 16Gbps
Memory Interface	256-bit
Memory Bandwidth	512 GB/s
System Interface	PCIe Gen 4.0 x 16
Power (TBP)	225W
Power Connector	1x 8-Pin 1x 6-Pin
HW Accelerated Media	AV1 (E&D), HEVC (E&D), H.264 (E&D), VP9 Bitstream & Decoding
Display Outputs	3x DP 2.0, 1x HDMI 2.1
Form Factor	10.5" Length, Dual Slot, Full Height
API Support	DirectX 12 Ultimate, OpenGL 4.6, OpenCL 3.0, Vulkan 1.3
OS Support	Win 10/11, Ubuntu
Intel Deep Link Technologies	Yes
Warranty	3-years


Optimized Applications
Optimized Middleware & Frameworks
DIRECT PROGRAMMING
Data Parallel C++ (DPC++)
API-BASED PROGRAMMING
oneAPI Libraries
Analysis & Debug Tools
CPU
GPU
FPGA
SPECIALIZED ACCELERATORS

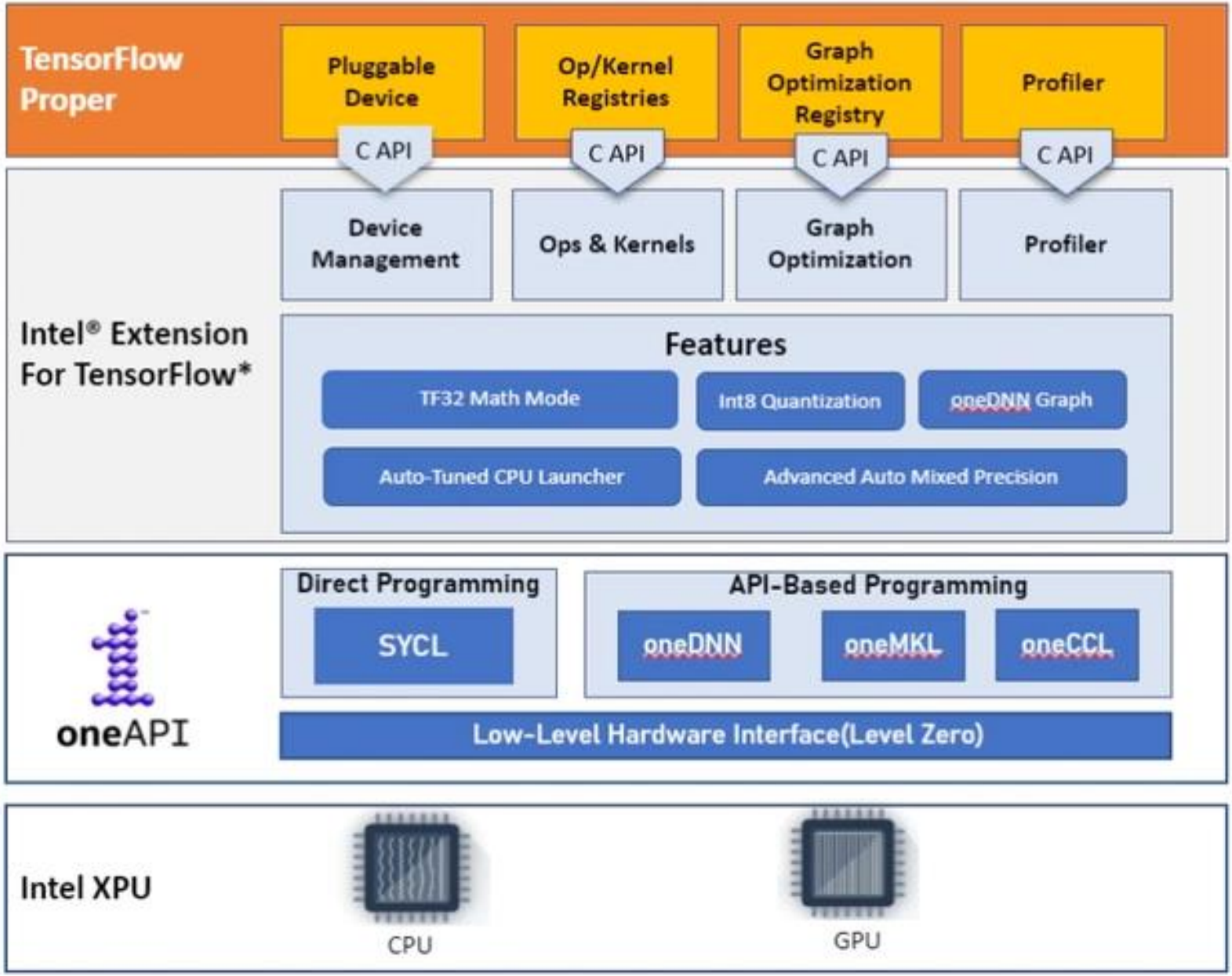
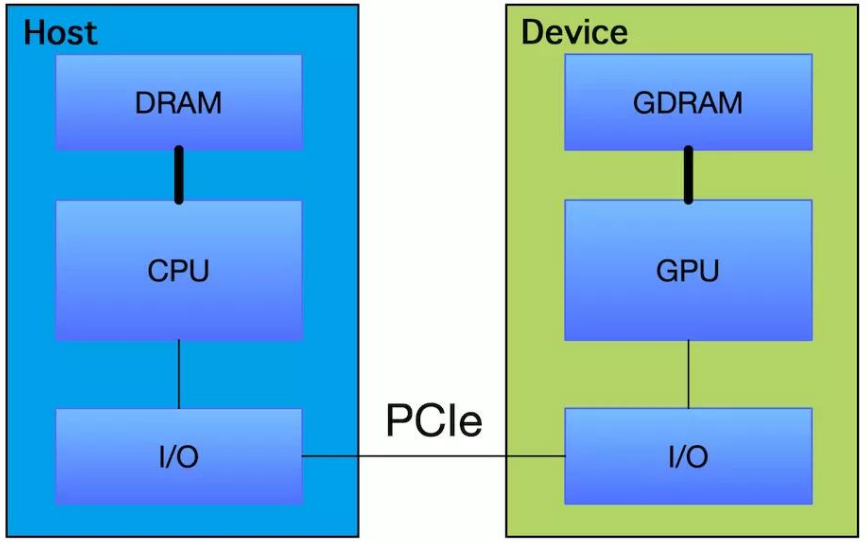
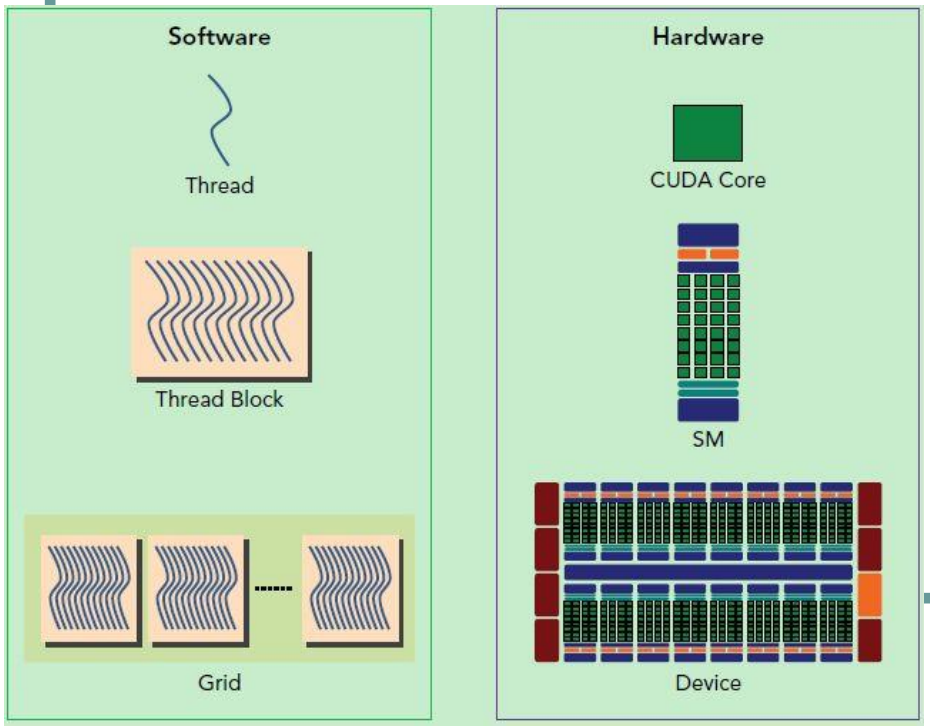
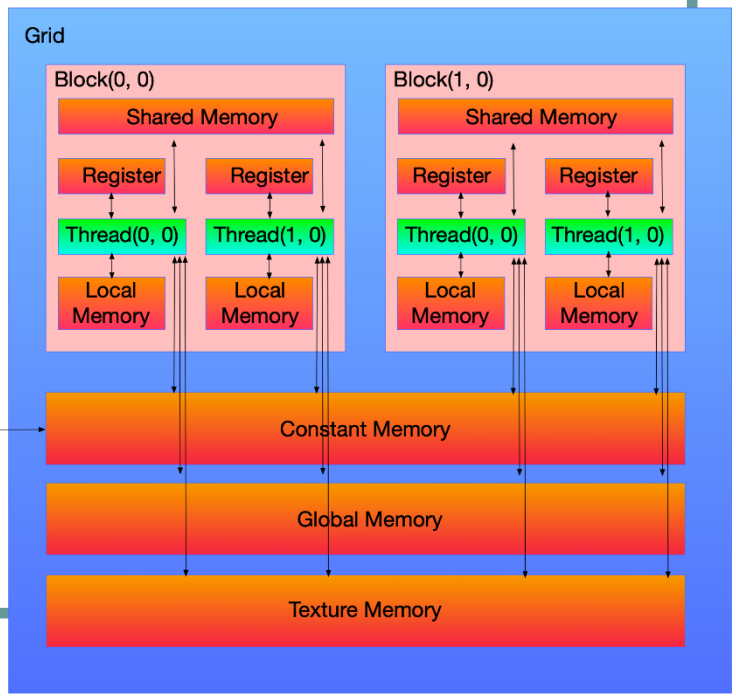


Figure 1: Intel® Extension for TensorFlow* Architecture

GPU 计算基础


Host
DRAM
CPU
I/O
PCIe
Device
GDRAM
GPU
I/O

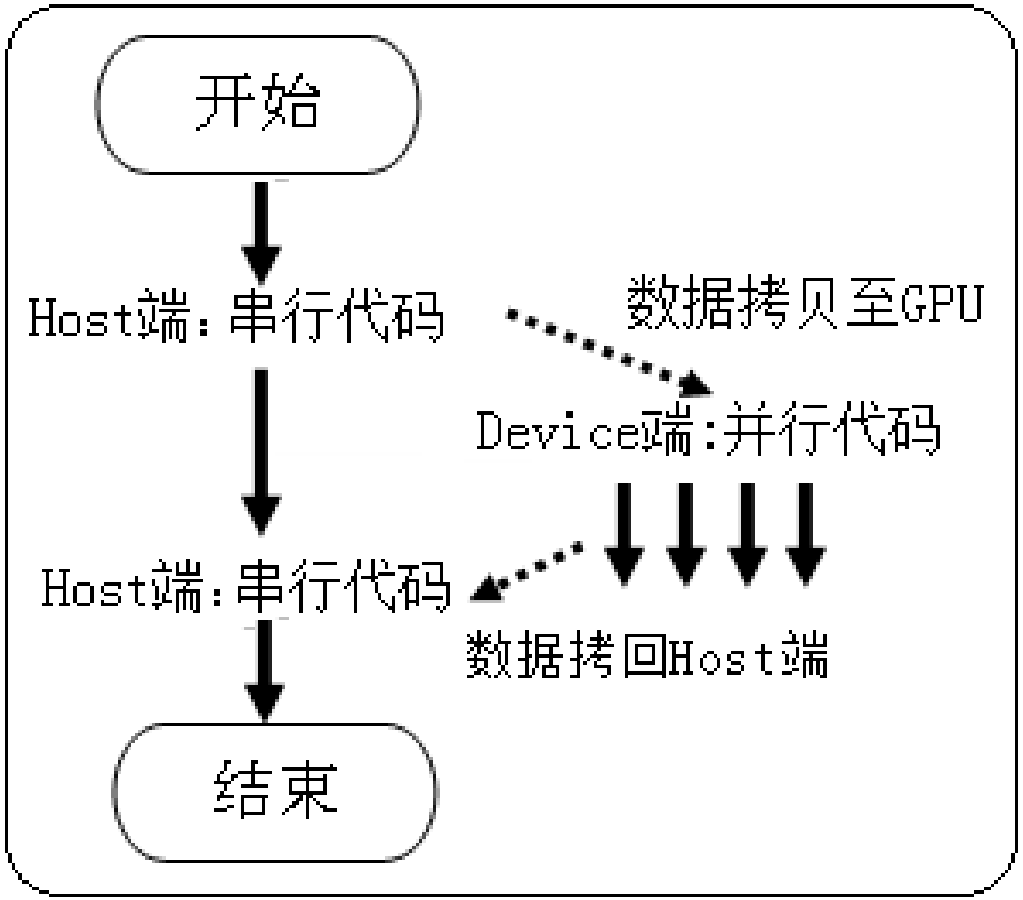

Software
Thread
Thread Block
Grid
Hardware
CUDA Core
SM
Device


Grid
Block(0, 0)
Block(1, 0)
Shared Memory
Register
Thread(0, 0)
Thread(1, 0)
Local Memory
Constant Memory
Global Memory
Texture Memory
Host

GPU VS CPU

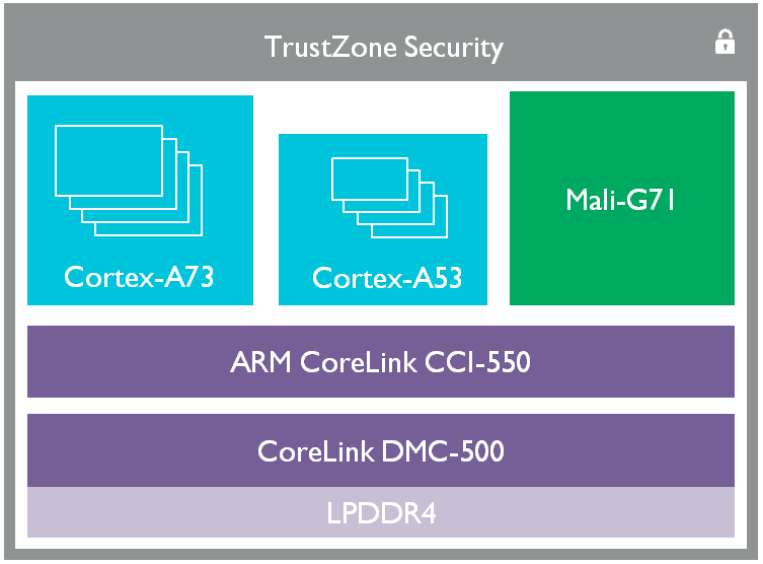
- PCI 16GB/s
- 内存
- 显存

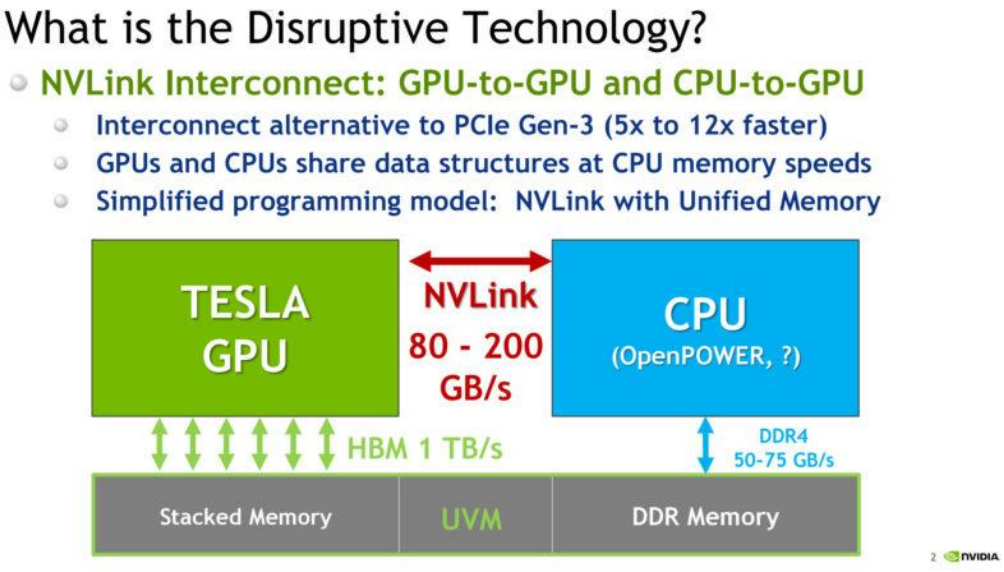
- CPU GPU



GPU +CPU

- CPU +GPU共享memory

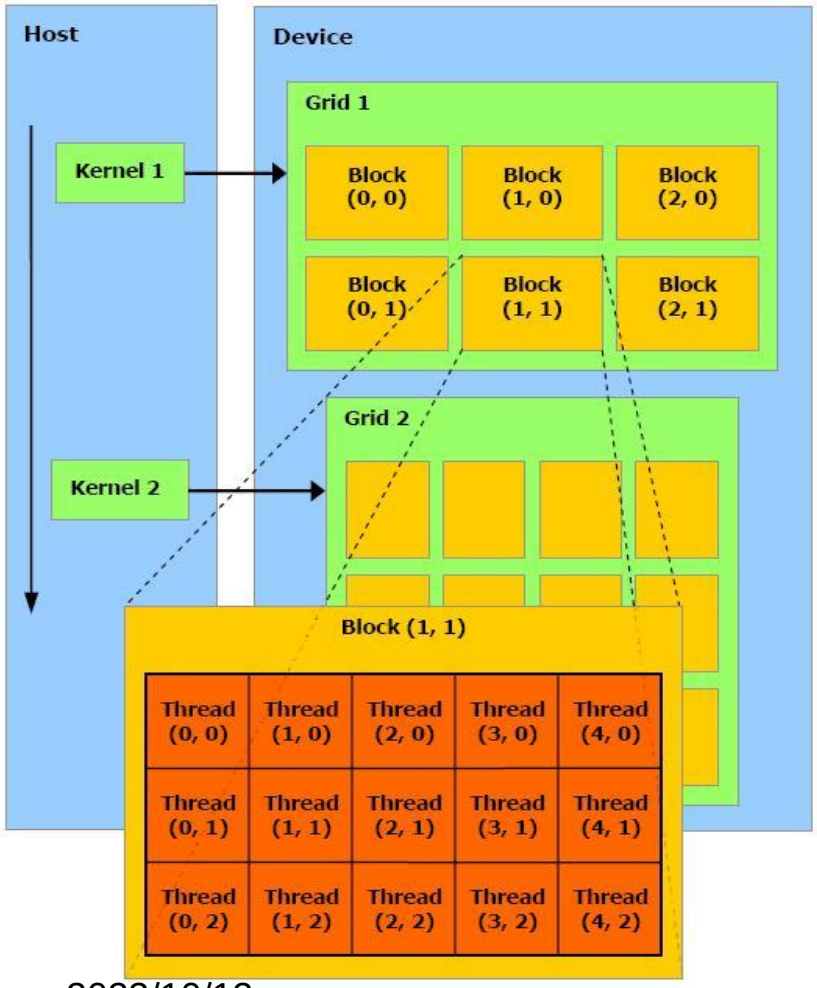




数据并行与任务并行

- 数据并行是所有线程中执行的内核(kernel)是相同的，只是处理的数据会根据内核代码分支等原因而不同。
- 任务并行则在每个线程中执行不同的代码

CUDA逻辑结构及内存结构

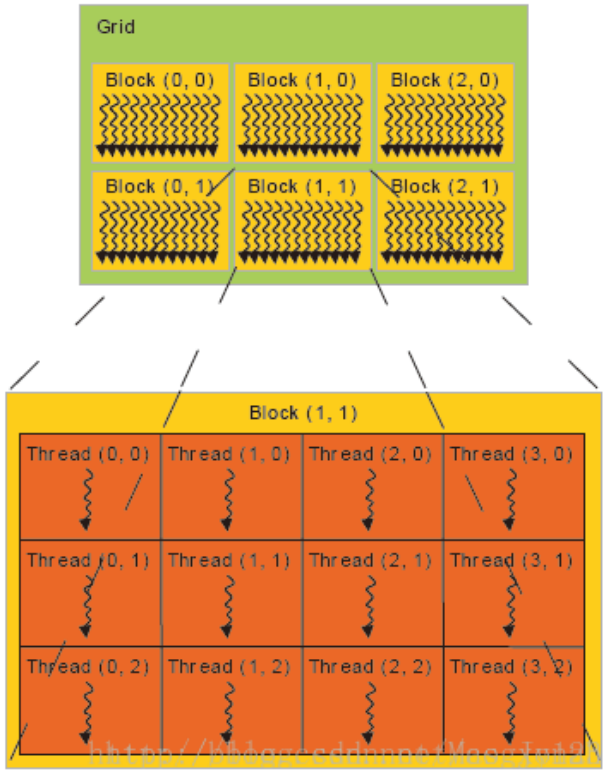


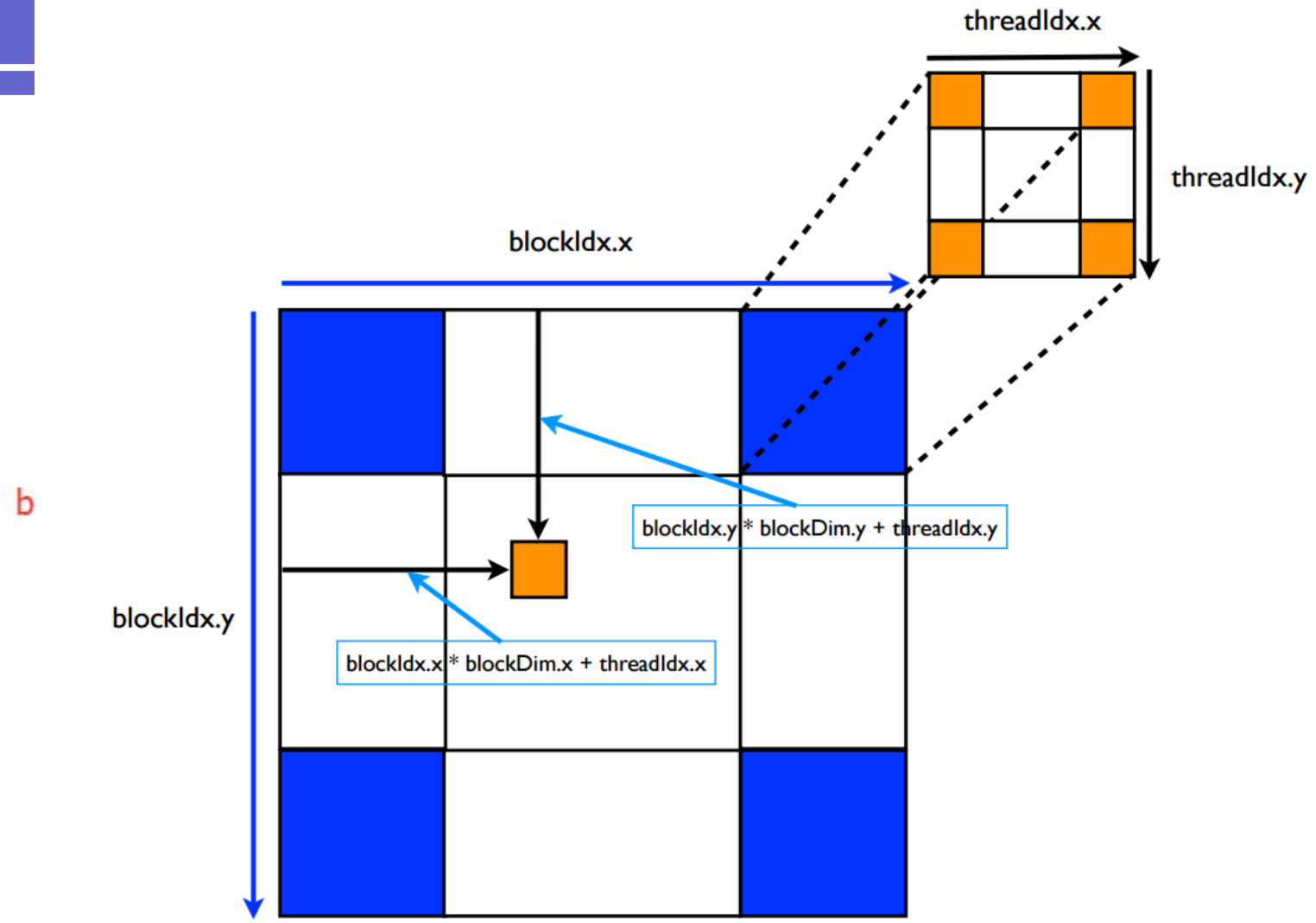
- 在CUDA的程序架构中，程序执行区域分为两个部分：Host 与Device。
  - Host 指在CPU上执行的部分
  - Device指在GPU 上执行的部分，这部分又称为“kernel”
- 通常，Host 程序会将需要并行计算的数据复制到 GPU 的显存，再由 GPU 执行 Device程序，完成后再由Host 程序将结果从GPU 显存中取回

CUDA code DEMO

1. blockDim.x * blockIdx.x + threadIdx.x

2. blockDim.x
3. blockIdx.x
4. threadIdx.x




threadIdx.x
threadIdx.y
blockIdx.x
blockIdx.y
blockIdx.y * blockDim.y + threadIdx.y
blockIdx.x * blockDim.x + threadIdx.x

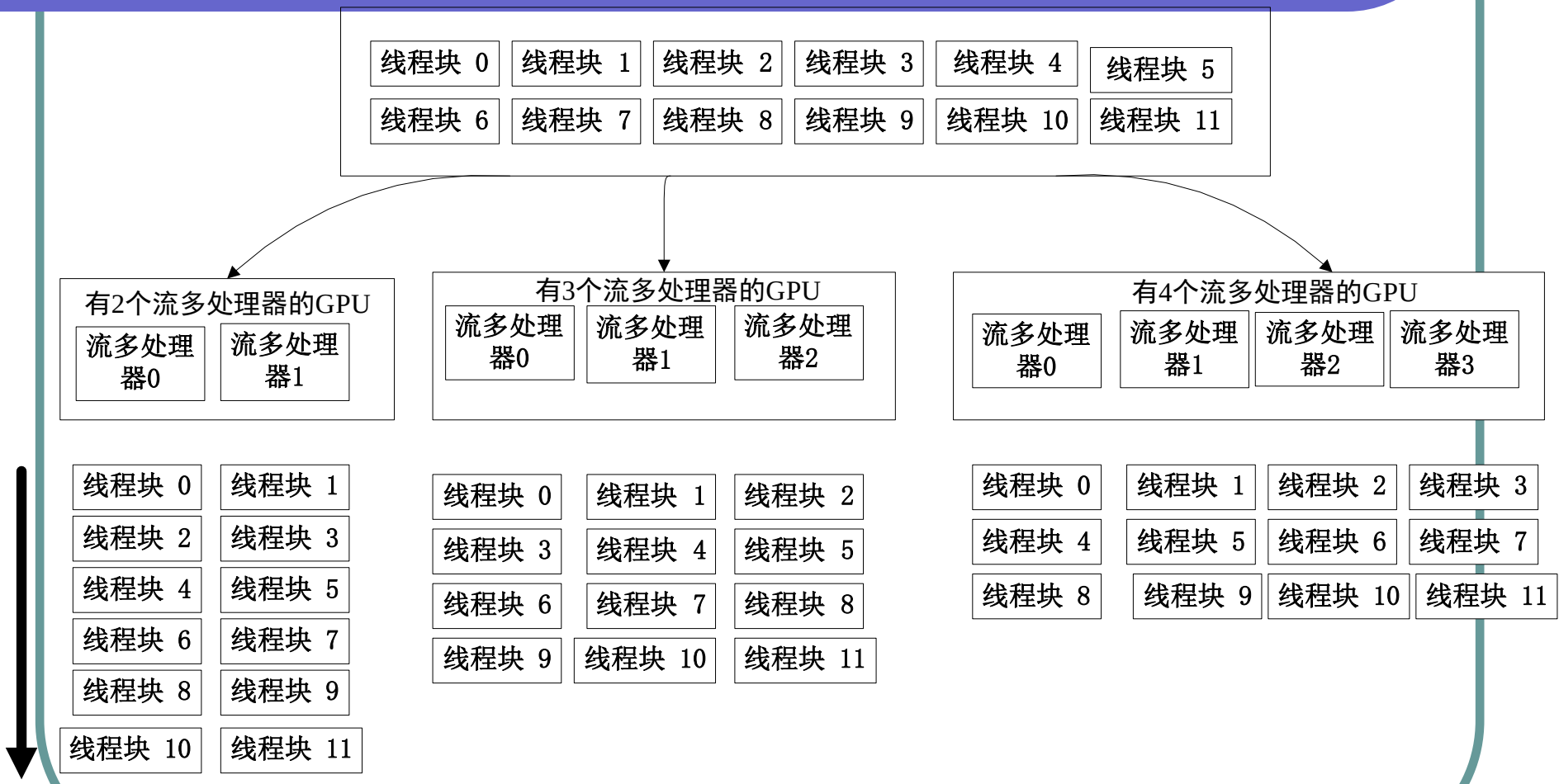
CUDA code DEMO

```
1. __global__ void
2. vectorAdd(const float *A, const float *B, float *C, int numElements)
3. {
4.     int i = blockDim.x * blockIdx.x + threadIdx.x;
5.     if (i < numElements)
6.     {
7.         C[i] = A[i] + B[i];
8.     }
9. }
```

CUDA code DEMO

- // Allocate the device input vector A
- float *d_A = NULL;
- err = cudaMalloc((void **)&d_A, size);
- // Copy the host input vectors A and B in host memory to the device input vectors in device memory
- printf("Copy input data from the host memory to the CUDA device\n");
- err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

网格硬件映射

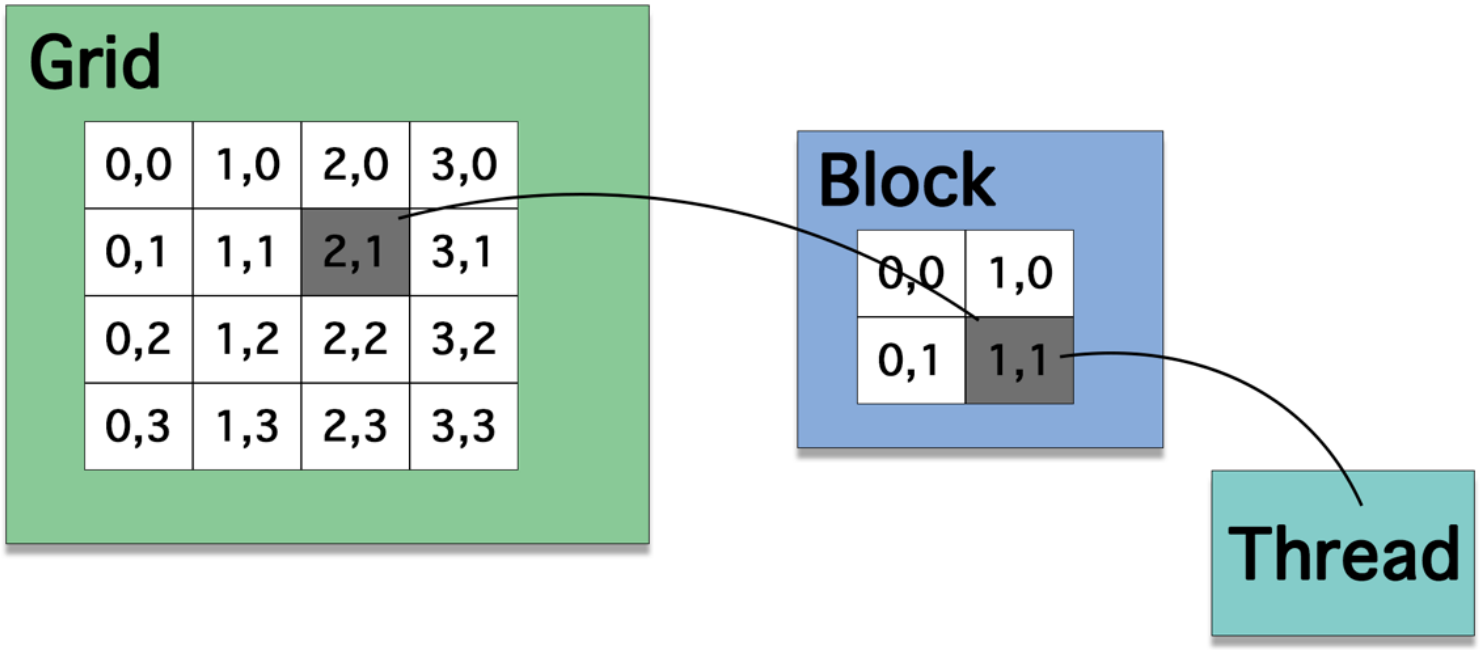

线程块 0
线程块 1
线程块 2
线程块 3
线程块 4
线程块 5
线程块 6
线程块 7
线程块 8
线程块 9
线程块 10
线程块 11
有2个流多处理器的GPU
流多处理器0
流多处理器1
有3个流多处理器的GPU
流多处理器0
流多处理器1
流多处理器2
有4个流多处理器的GPU
流多处理器0
流多处理器1
流多处理器2
流多处理器3
线程块 0
线程块 1
线程块 2
线程块 3
线程块 4
线程块 5
线程块 6
线程块 7
线程块 8
线程块 9
线程块 10
线程块 11
线程块 0
线程块 1
线程块 2
线程块 3
线程块 4
线程块 5
线程块 6
线程块 7
线程块 8
线程块 9
线程块 10
线程块 11
线程块 0
线程块 1
线程块 2
线程块 3
线程块 4
线程块 5
线程块 6
线程块 7
线程块 8
线程块 9
线程块 10
线程块 11

网格执行例子

- 1000个元素相加
- 每个block=128，启动几个block？
- 每个block=256，启动几个block？

- Block取值范围？？？

Grid


Grid
0,0
1,0
2,0
3,0
0,1
1,1
2,1
3,1
0,2
1,2
2,2
3,2
0,3
1,3
2,3
3,3
Block
0,0
1,0
0,1
1,1
Thread

saxpy

- SAXPY stands for “Single-Precision A·X Plus Y”. It is a function in the standard Basic Linear Algebra Subroutines (BLAS)library
- void saxpy(int n, float a, float * restrict x, float * restrict y)
- { 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);

Example Kernel

```
__global__ void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
```

CUDA程序

- int N = 1<<20;
- float *x, *y, *d_x, *d_y;
- x = (float*)malloc(N*sizeof(float));
- cudaMalloc(&d_x, N*sizeof(float));
- cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
- // Perform SAXPY on 1M elements
- saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
- cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
- cudaFree(d_x);
- free(x);

CUDA --SAXPY

- cudaMemcpy 同步传输(或者是阻塞传输)方式

- 核函数启动是异步的。一旦核函数被启动，控制权就立刻返回到CPU，并不会等待核函数执行完成。saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);

SAXPY性能度量

- cuda事件是cudaEvent_t类型，通过cudaEventCreate()和cudaEventDestroy()进行事件的创建和销毁。
- 函数cudaEventSynchronize()用来阻塞CPU执行直到指定的事件被记录。
- 函数 cudaEventElapsedTime()的第一个参数返回start和stop两个记录之间消逝的毫秒时间。这个值的精度大约是0.5*ms*。

SAXPY性能度量

```
1. cudaEvent_t start, stop;
2. cudaEventCreate(&start);
3. cudaEventCreate(&stop);
4. cudaMemcpy
5. cudaEventRecord(start);
6. saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
7.  cudaEventRecord(stop);
8. cudaMemcpy(…cudaMemcpyDeviceToHost);
9. cudaEventSynchronize(stop);
10.  float milliseconds = 0;
11. cudaEventElapsedTime(&millise
```

内存带宽

1. **理论带宽**

- GPU时钟频率1546MHz，显存位宽384-bitDDR(双倍数据速率)RAM。理论峰值带宽是148 GB/sec：
- $BW_{Theoretical} = 1546 * 10^6 * (384/8) * 2/10^9 = 148 GB/s$

2. **有效带宽**

- $BW_{Effective} = (R_B + W_B)/(t * 10^9)$
- $BW_{Effective}$ 是以GB/s的有效带宽， R_B 是每个核函数被读取的字节数， W_B是每个核函数被写入的字节数，t 是以秒为单位的运行时间。

内存带宽测试

公式：

N*4*3/milliseconds/1e6

分析：N*4是每次数组读或写的字节数，因子3的含义是对x的读以及y的读和写共3次读写操作。程序运行时间被存在变量milliseconds中，把它作为分母即可算出单位时间的带宽大小

```
__global__  void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
```

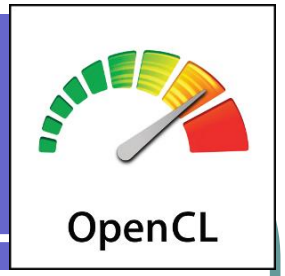
计算GFLOPS

公式：

SAXPY计算：每个SAXPY元素都会做一次乘法加法操作，因此是2FLOPS

```
__global__  void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
```

OpenCL



- 开放计算语言，Open Computing Language
- 苹果发起，Khronos开发
  - 2008年6月苹果公司提出
  - AMD, Intel, ARM, NVIDIA, Qualcomm, Nokia, IBM, Samsung, TI等
- 适合CPU、GPU、CELL、DSP架构
  - AMD ： OpenCL 2.0，ROCM
  - NVIDIA支持OpenCL

- 由一个并行计算API和一种针对此类计算的编程语言组成
  - C99编程语言并行扩展子集;
  - 适用于各种类型异构处理器的坐标数据和基于任务并行计算API;
  - 基于IEEE 754标准的数字条件;
  - 与OpenGL、OpenGL ES和其他图形类API高效互通
  - 提供了基于任务分割和数据分割的并行计算机制
- Khronos Group
  - 发布OpenGL
  - http://www.khronos.org/opencl/

OpenCL的现状

- 已由KHronos Groups发布了OpenCL 1.0
- OpenCL 1.1　2010.6
  - http://www.khronos.cn/
  - The OpenCL 1.0 specification and header files
- AMD
  - AMD发布业界第一款x86 CPU用OpenCL软件开发平台
  - AMD OpenCL并行处理技术在Siggraph Asia 2008演示
  - AMD FireStreamTM 9270 支持OpenCL
- Apple
  - 2009全球开发者大会（WWDC）发布了包含OpenCL技术的Snow Leopard OS

OpenCL四种模型

- OpenCL allows parallel computing on heterogeneous devices
  - CPUs, GPUs, other processors (Cell, DSPs, etc)
  - Provides portable accelerated code

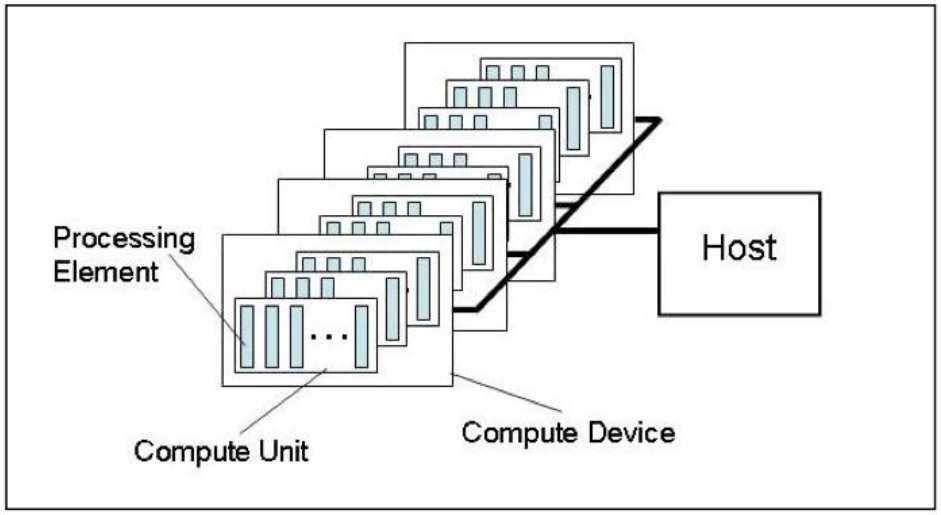
- 平台模型(Platform Model)
- 执行模型(Execution Model)
- 内存模型(Memory Model)
- 编程模型(Programming Model)

Platform Model

- Each OpenCL implementation (i.e. an OpenCL library from AMD, NVIDIA, etc.) defines *platforms* which enable the host system to interact with OpenCL-capable devices
  - Currently each vendor supplies only a single platform per implementation
- OpenCL uses an “Installable Client Driver” model
  - The goal is to allow platforms from different vendors to co-exist
  - Current systems’ device driver model will not allow different vendors’ GPUs to run at the same time

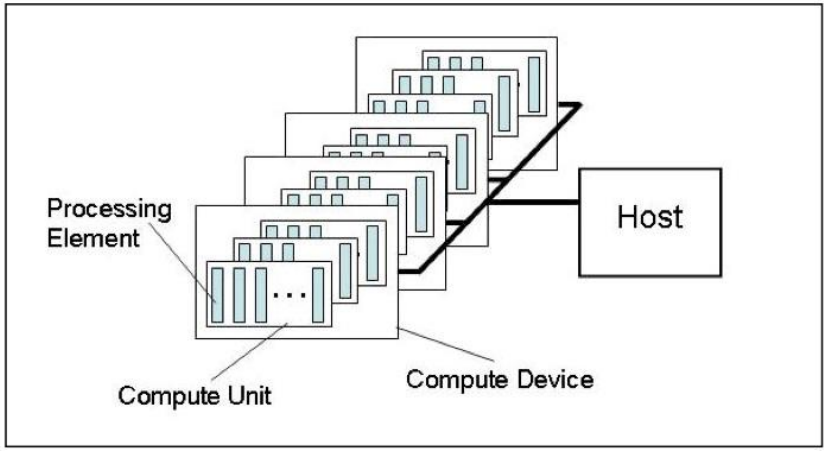
Platform Model

- The model consists of a host connected to one or more OpenCL devices
- A device is divided into one or more compute units
- Compute units are divided into one or more processing elements
  - Each processing element maintains its own program counter



Host/Devices

- The host is whatever the OpenCL library runs on
  - x86 CPUs for both NVIDIA and AMD
- Devices are processors that the library can talk to
  - CPUs, GPUs, and generic accelerators
- For AMD
  - All CPUs are combined into a single device (each core is a compute unit and processing element)
  - Each GPU is a separate device

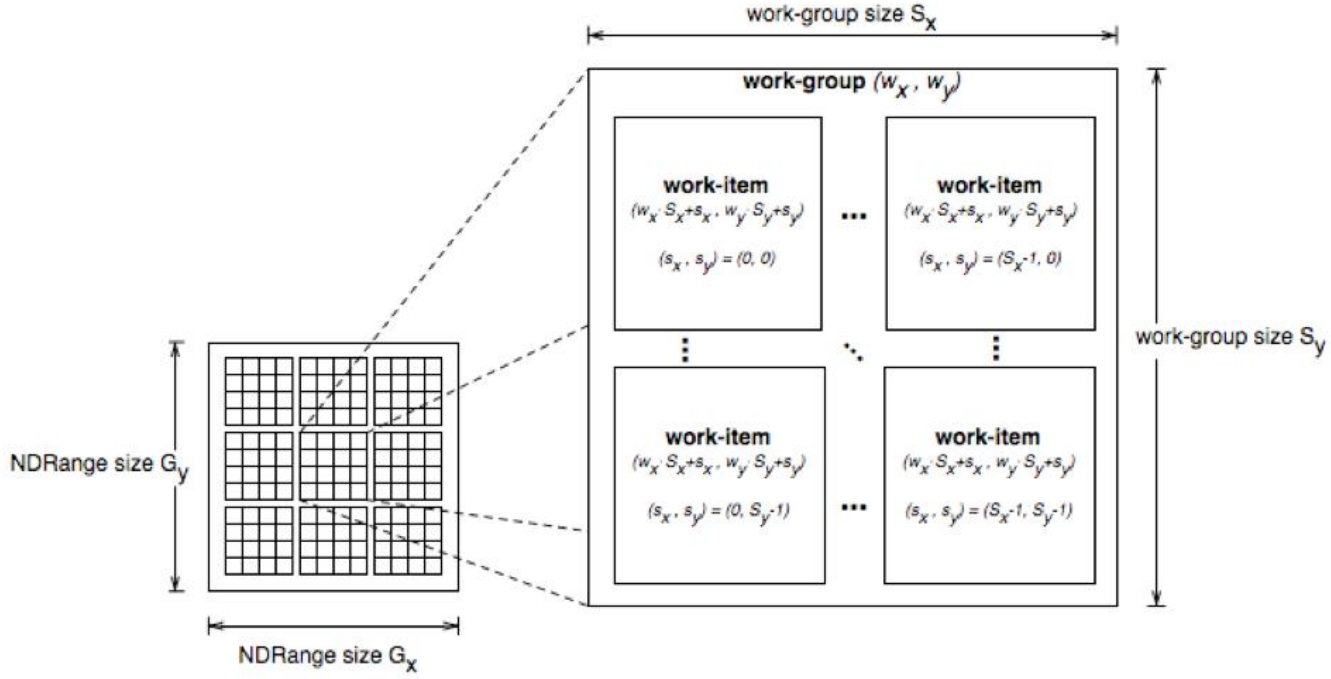


执行模型

- OpenCL的执行模型可以分为两部分，一部分是在 host上执行的主程序（host program），另一部分是在 OpenCL设备上执行的内核程序（kernels），OpenCL通过主程序来定义上下文并管理内核程序在OpenCL设备的执行。
- OpenCL执行模型的核心工作就是管理 kernel在OpenCL设备上的运行。在 Host创建一个kernel程序之前必须先为该 kernel创建一个标识了索引的工作空间，kernel会在工作空间每个节点 (workitem)上执行

Thread Structure

- Work-items can uniquely identify themselves based on:
  - A global id (unique within the index space)
  -

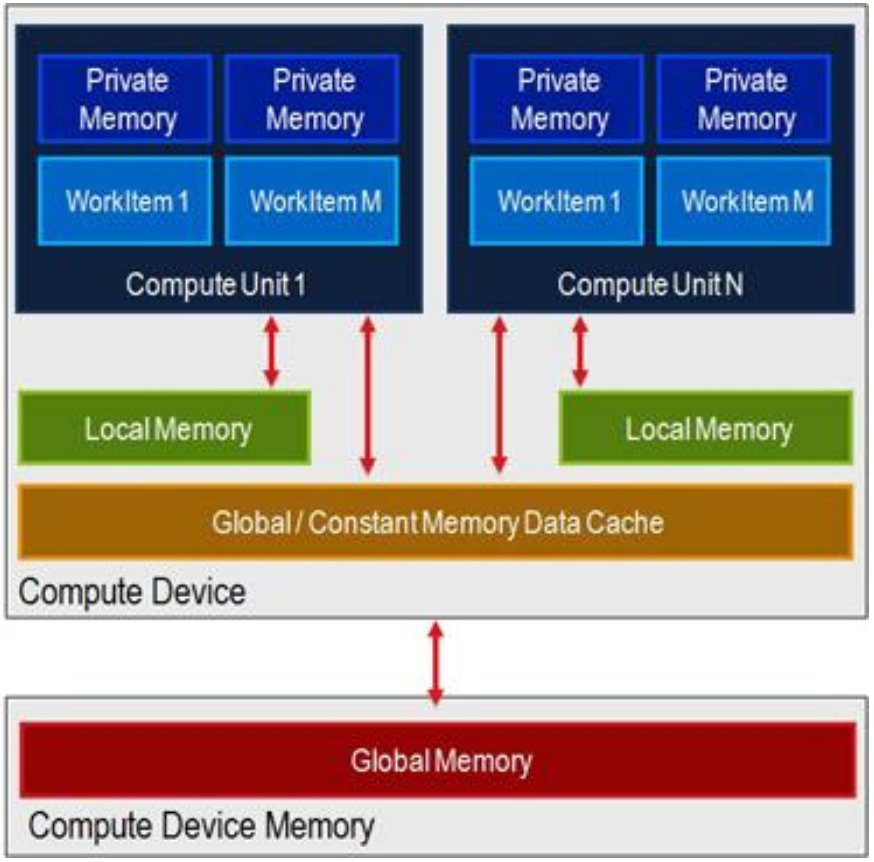


Thread Structure

- API calls allow threads to identify themselves and their data
- Threads can determine their global ID in each dimension
  - get_global_id(dim)
  - get_global_size(dim)
- Or they can determine their work-group ID and ID within the workgroup
  - get_group_id(dim)
  - get_num_groups(dim)
  - get_local_id(dim)
  - get_local_size(dim)
- get_global_id(0) = column, get_global_id(1) = row
- get_num_groups(0) * get_local_size(0) == get_global_size(0)

Memory Model

- private memory
- local memory
- constant memory
- global memory



使用OpenCL编程的6个步骤

- 查找支持OpenCL的硬件设备(Device)，并创建上下文(Context)
- 创建命令队列(Command Queue)及包含了内核的程序(Program)；如果该程序是源代码，则还需进行在线编译。
- 创建程序执行过程中需要的内存对象(Buffer)及图像对象(Image Object)，并初始化
- 创建内核对象（可理解为Kernel中的函数）并设置其所需参数
- 设置内核的索引空间(NDRange)并执行内核。其中NDRange通过全局尺寸(GlobalSize)和工作包尺寸(WorkGroup)来进行管理。
- 将运行的结果拷贝回主机(Host)内存。

主机处理器上运行的代码

- 打开OpenCL上下文，
- 获得并选择执行设备，
- 创建命令队列，以接受并执行内存请求，
- 分配OpenCL内存对象以掌握用于计算内核的输入和输出，
- 在线编译及构建计算内核代码，
- 建立参数和执行域，
- 开始执行计算内核，以及
- 收集结果。

example

```
__kernel   void vecadd(__global int *A,
           __global int *B,
           __global int *C) {

   int idx = get_global_id(0);
   C[idx] = A[idx] + B[idx];
   }
```

example

- // System includes
- #include <stdio.h>
- #include <stdlib.h>
- #include <CL/cl.h>

作业

- 搭建自己的GPU环境
- 测试第一个GPU程序