并行与分布式计算基础：第十五讲

杨超
chao_yang@pku.edu.cn

2019 秋

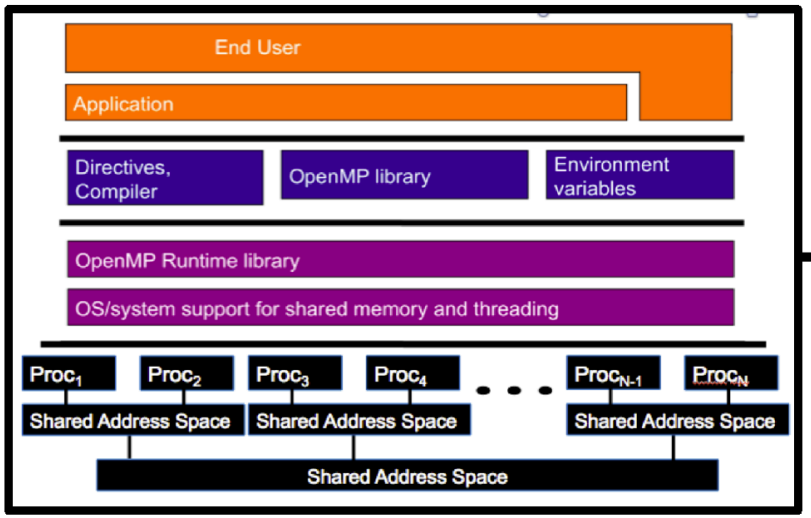
内容提纲

1. OpenMP 编程-6
   - 新特性
2. CUDA 编程-1
   - GPU 简介
   - 初识 CUDA

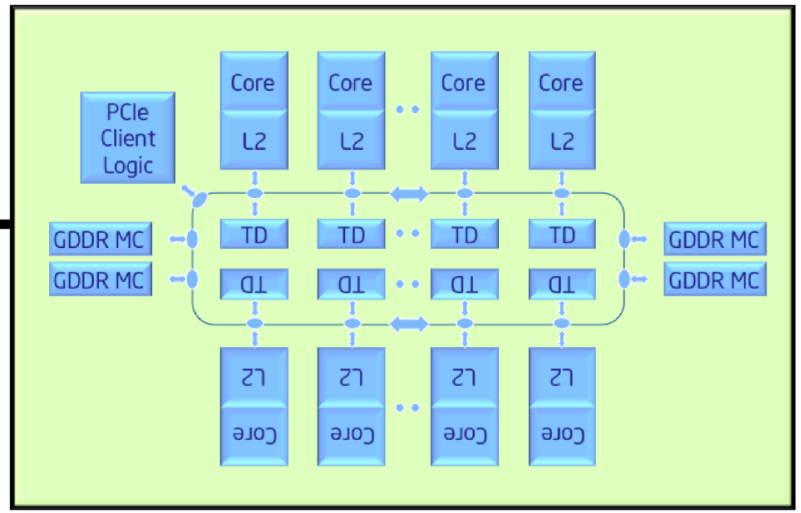
新特性

OpenMP 对加速卡 (accelerator) 的支持

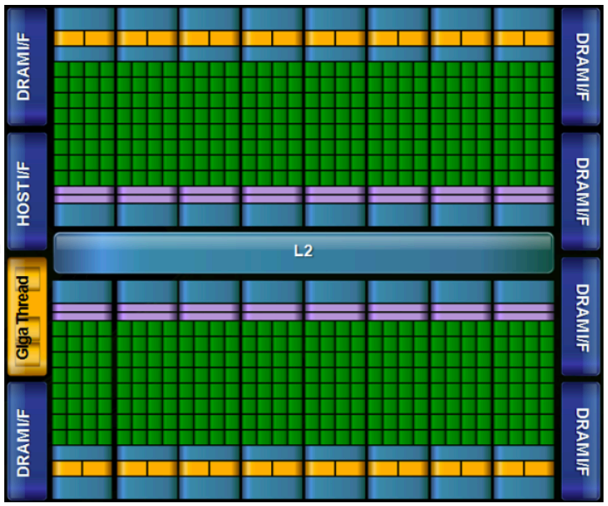
Supported (since OpenMP 4.0) with target, teams, distribute, and other constructs



Host



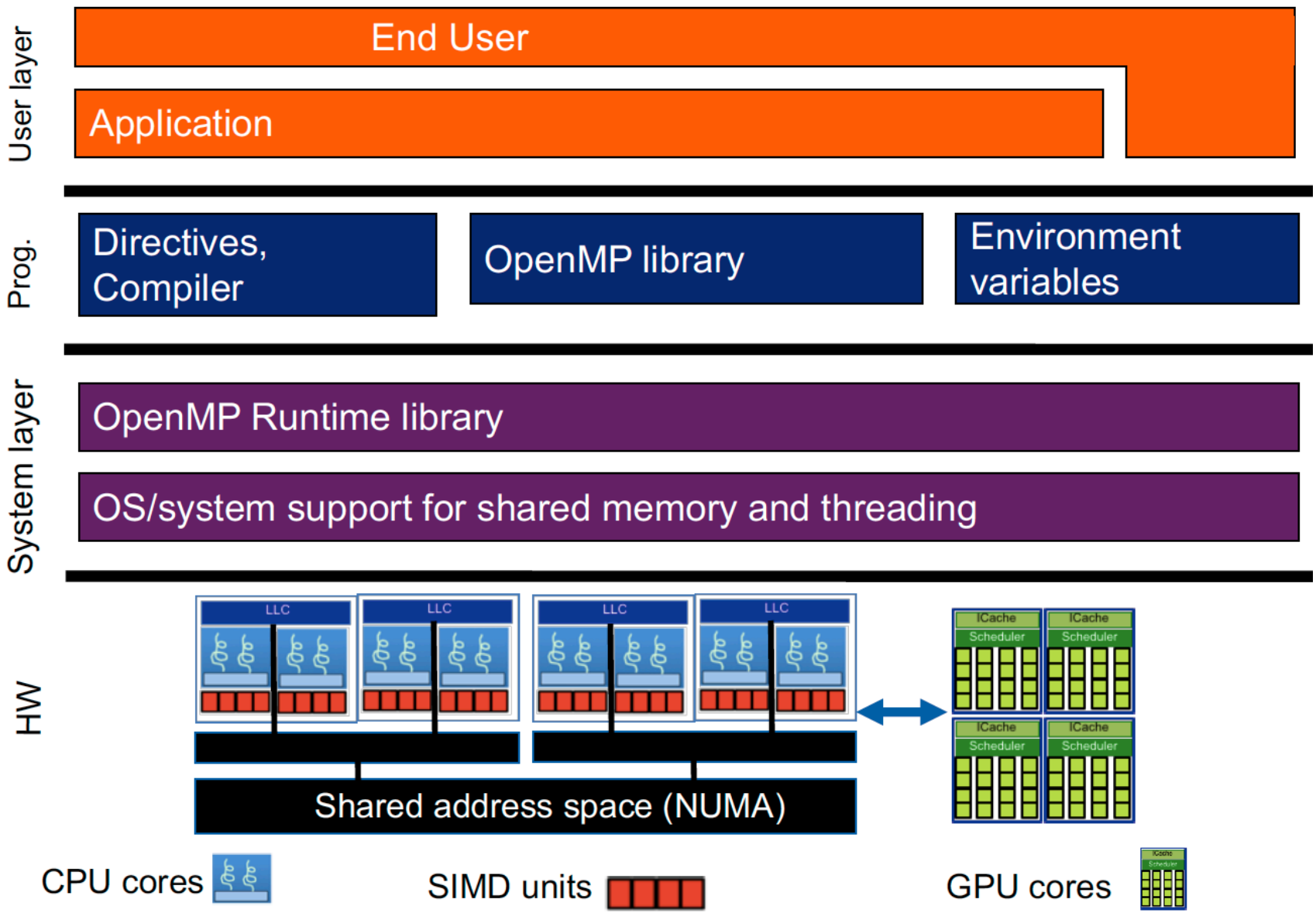
Target Device: Intel® Xeon Phi™ coprocessor



Target Device: GPU

OpenMP 4.5

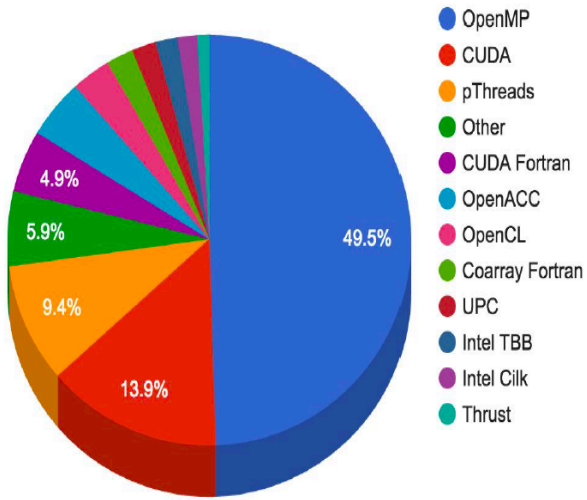
更新后的 OpenMP 基础解决方案



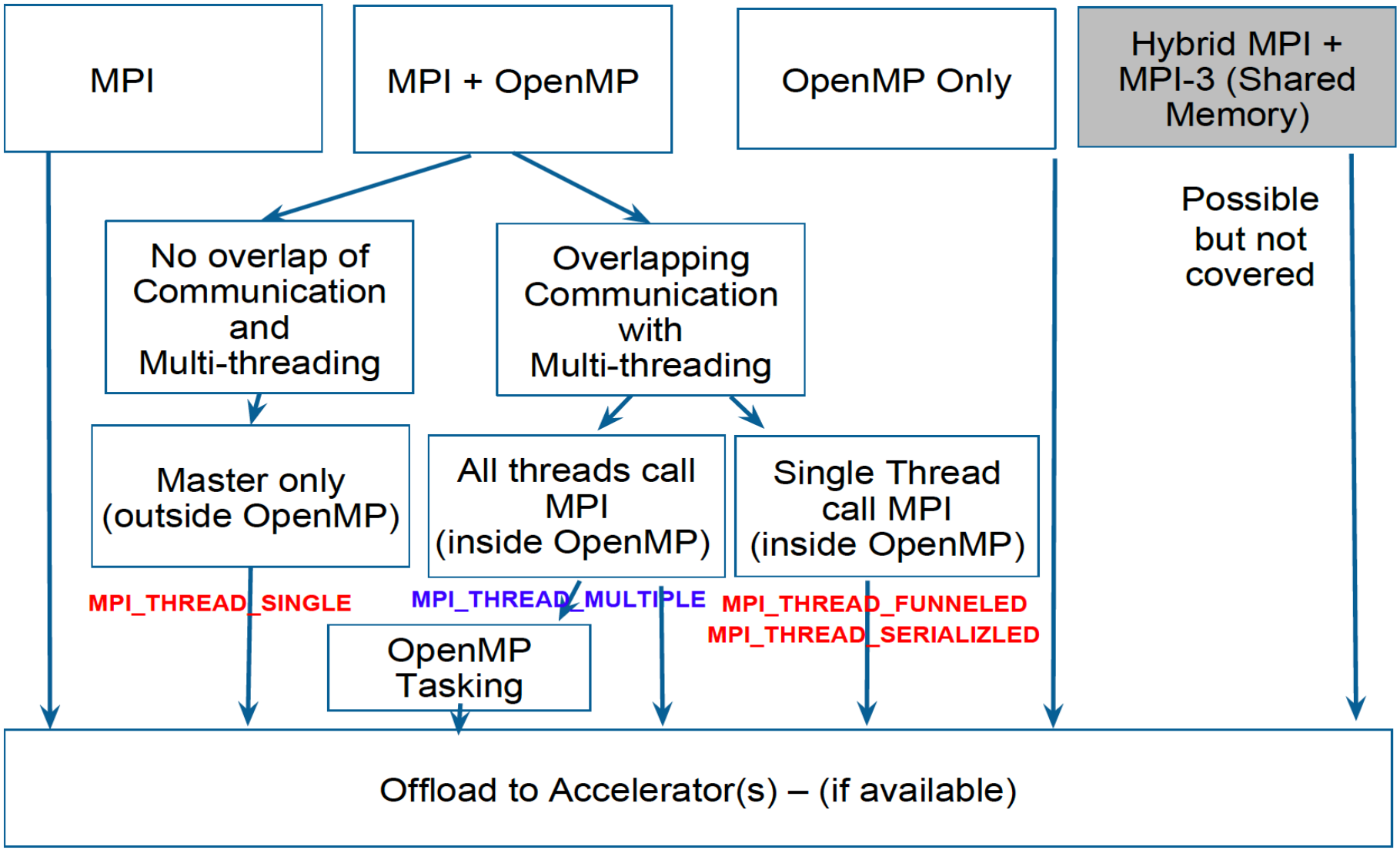
各种并行编程模型汇总

- MPI was developed primarily for inter-address space (inter means between or among)
- OpenMP was developed for shared memory or intra-node, and now supports accelerators as well (intra means within)
- Hybrid Programming (MPI+X) is when we use a solution with different programming models for inter *vs.* intra-node parallelism
- Several solutions including
  - Pure MPI
  - MPI + Shared Memory (OpenMP)
  - MPI + Accelerator programming
    - OpenMP 4.5 shared memory + offload, OpenACC, CUDA, etc
  - MPI message passing + MPI shared memory
  - PGAS: UPC/UPC++, Fortran 2008 coarrays, GA, OpenSHMEM, etc
  - Runtime tasks (Legion, HPX, HiHat (draft), etc)
  - Other hybrid based on Kokkos, Raja, SYCL, C++17 (C++20 draft)

NERSC data from 2015:
When asked: If you use MPI + X, what is X ?



MPI+OpenMP 混合编程

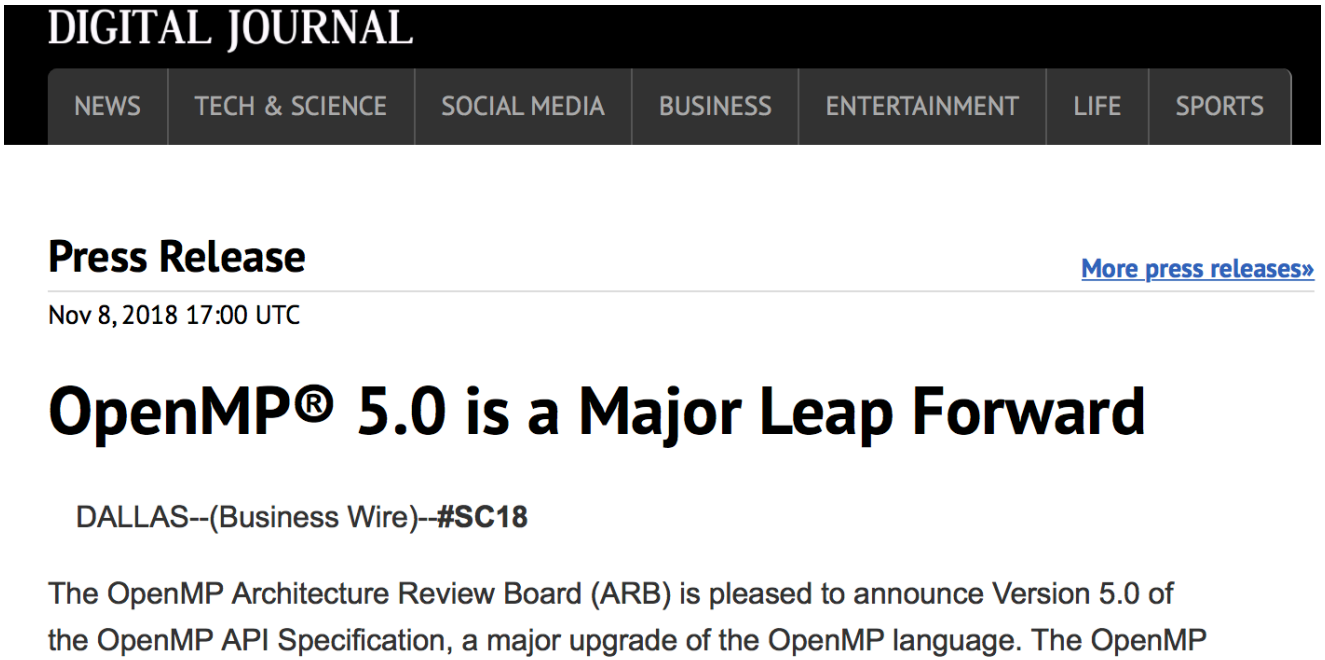


其他未讨论的话题

- 假共享 (false sharing);
- 线程锁;
- 自定义规约;
- 高级任务并行 (OpenMP4.0);
- ...

OpenMP 5.0 发布

2018 年 11 月 8 日，在美国 Dallas 举办的 SC18 大会上，OpenMP 5.0 正式发布，在功能方面的主要增强/增加包括：全面支持众核加速设备 (Intel Xeon Phi, GPU 等)、增强的调试和性能分析接口、支持最新版 C/C++/Fortran、增强的可移植性等.



DIGITAL JOURNAL

NEWS | TECH & SCIENCE | SOCIAL MEDIA | BUSINESS | ENTERTAINMENT | LIFE | SPORTS

Press Release More press releases»

Nov 8, 2018 17:00 UTC

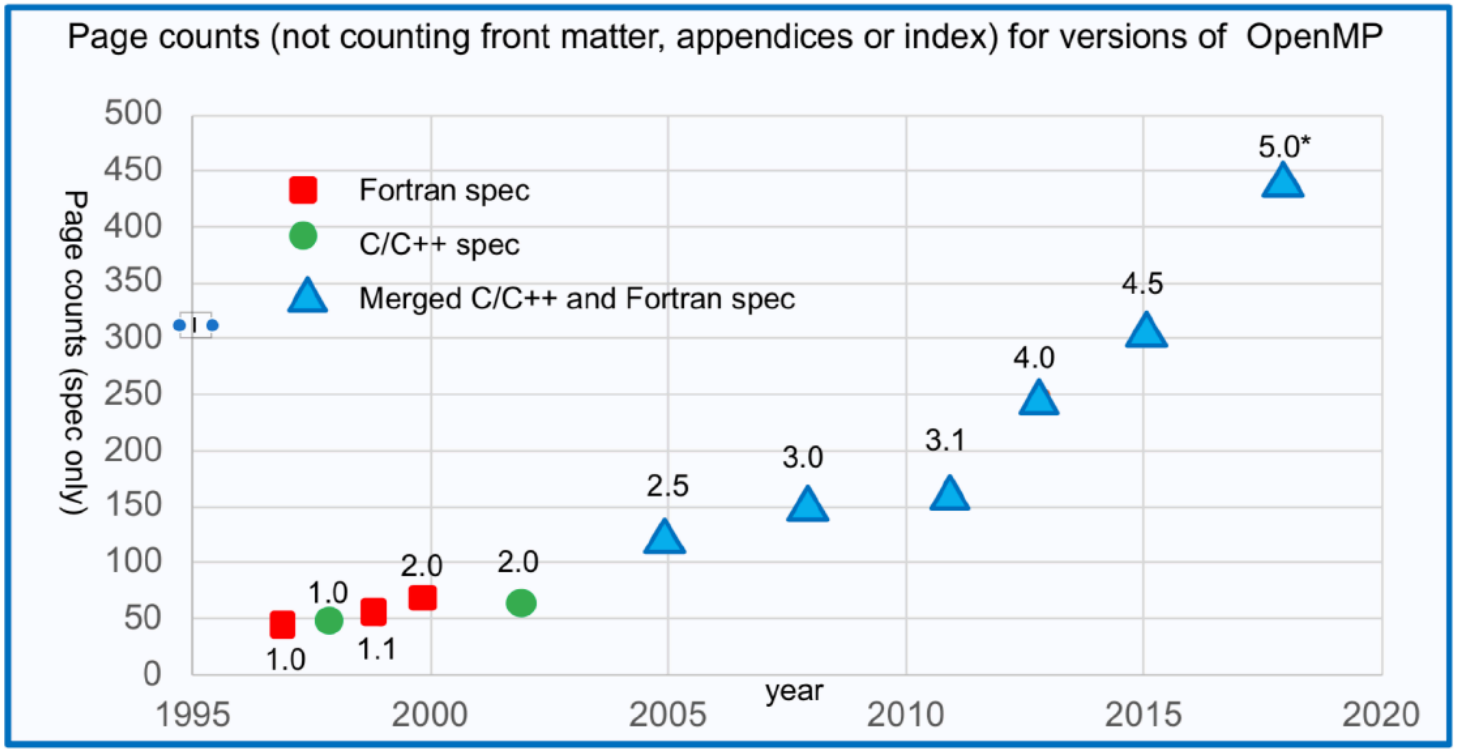
OpenMP® 5.0 is a Major Leap Forward

DALLAS--(Business Wire)--**#SC18**

The OpenMP Architecture Review Board (ARB) is pleased to announce Version 5.0 of the OpenMP API Specification, a major upgrade of the OpenMP language. The OpenMP

OpenMP 发展历史一览

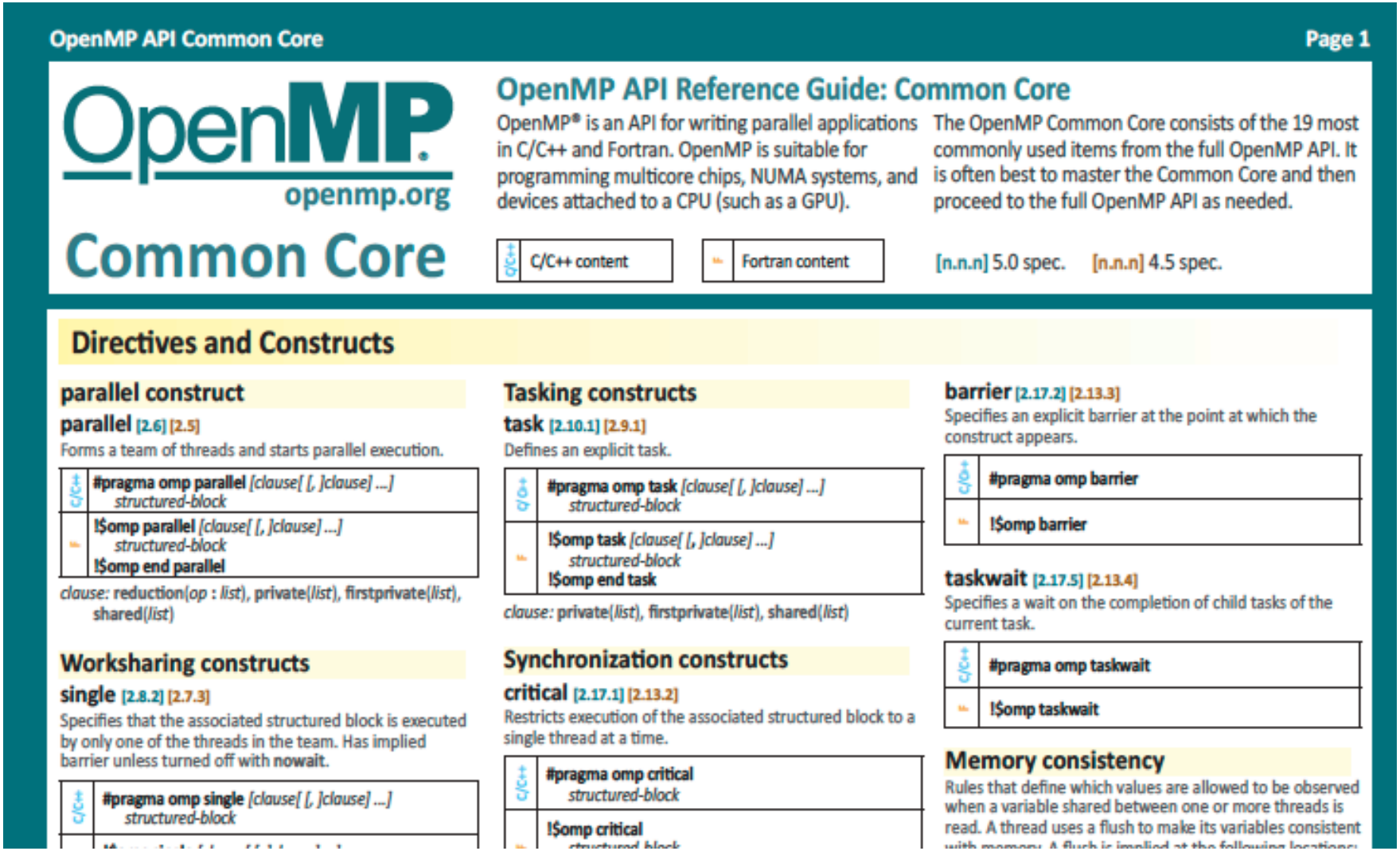
- 从 1997 年诞生开始，OpenMP 经历了从仅适用于科学计算的简单接口到更为通用的、复杂的 API 的发展历程。



* Does not include the tools interface added with OpneMP 5.0 which pushes the page count to 618

- 为了降低 OpenMP 的学习复杂性，可先学习主要的 OpenMP 常用核心 (Common Cores)，在此基础上学习向量化等高级技能。

OpenMP Common Core Reference Guide



OpenMP API Common Core Page 1

OpenMP
openmp.org

Common Core

OpenMP API Reference Guide: Common Core

OpenMP® is an API for writing parallel applications in C/C++ and Fortran. OpenMP is suitable for programming multicore chips, NUMA systems, and devices attached to a CPU (such as a GPU).

The OpenMP Common Core consists of the 19 most commonly used items from the full OpenMP API. It is often best to master the Common Core and then proceed to the full OpenMP API as needed.

C/C++ C/C++ content | Fortran Fortran content

[n.n.n] 5.0 spec. [n.n.n] 4.5 spec.

Directives and Constructs

parallel construct

parallel [2.6] [2.5]

Forms a team of threads and starts parallel execution.

C/C++
```
#pragma omp parallel [clause[ [, ]clause] ...]
    structured-block
```

Fortran
```
!$omp parallel [clause[ [, ]clause] ...]
    structured-block
!$omp end parallel
```

clause: reduction(*op* : *list*), private(*list*), firstprivate(*list*), shared(*list*)

Worksharing constructs

single [2.8.2] [2.7.3]

Specifies that the associated structured block is executed by only one of the threads in the team. Has implied barrier unless turned off with **nowait**.

C/C++
```
#pragma omp single [clause[ [, ]clause] ...]
    structured-block
```

Tasking constructs

task [2.10.1] [2.9.1]

Defines an explicit task.

C/C++
```
#pragma omp task [clause[ [, ]clause] ...]
    structured-block
```

Fortran
```
!$omp task [clause[ [, ]clause] ...]
    structured-block
!$omp end task
```

clause: private(*list*), firstprivate(*list*), shared(*list*)

Synchronization constructs

critical [2.17.1] [2.13.2]

Restricts execution of the associated structured block to a single thread at a time.

C/C++
```
#pragma omp critical
    structured-block
```

Fortran
```
!$omp critical
```

barrier [2.17.2] [2.13.3]

Specifies an explicit barrier at the point at which the construct appears.

C/C++
```
#pragma omp barrier
```

Fortran
```
!$omp barrier
```

taskwait [2.17.5] [2.13.4]

Specifies a wait on the completion of child tasks of the current task.

C/C++
```
#pragma omp taskwait
```

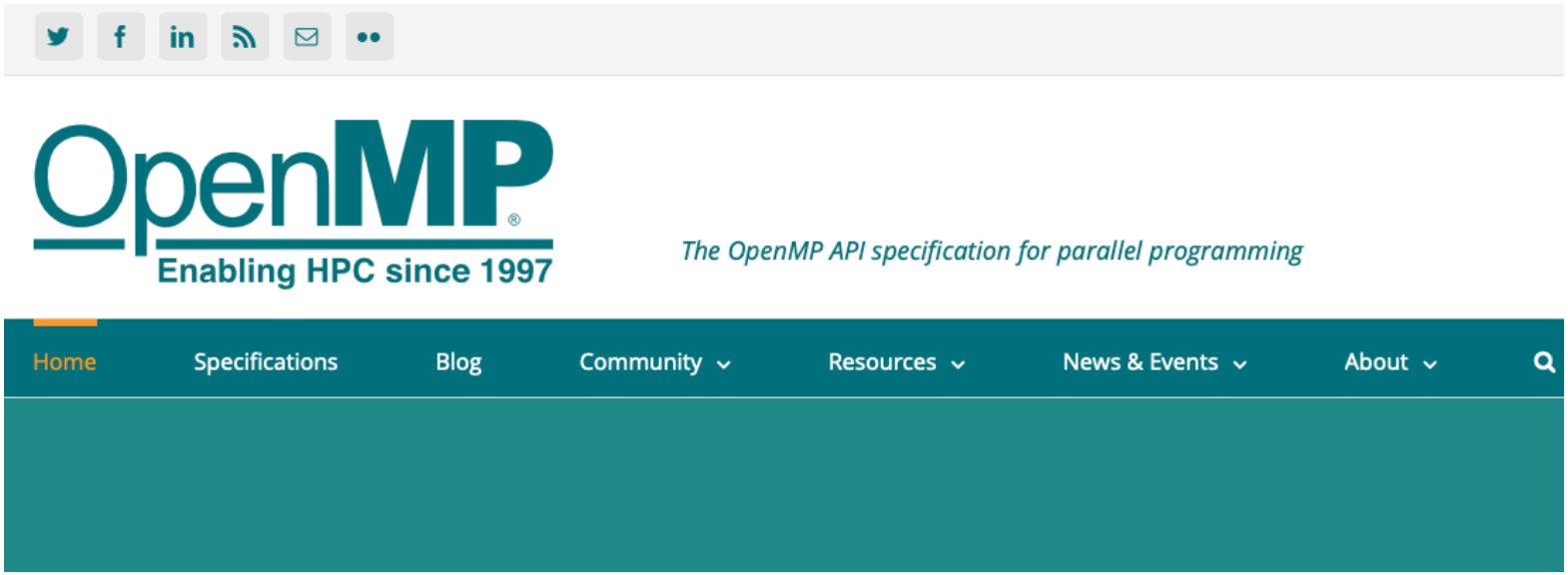
Fortran
```
!$omp taskwait
```

Memory consistency

Rules that define which values are allowed to be observed when a variable shared between one or more threads is read. A thread uses a flush to make its variables consistent

OpenMP 官方网址

- 为了更好的了解 OpenMP 的技术动态和进展，可关注官网：
  `https://www.openmp.org`



GPU 简介

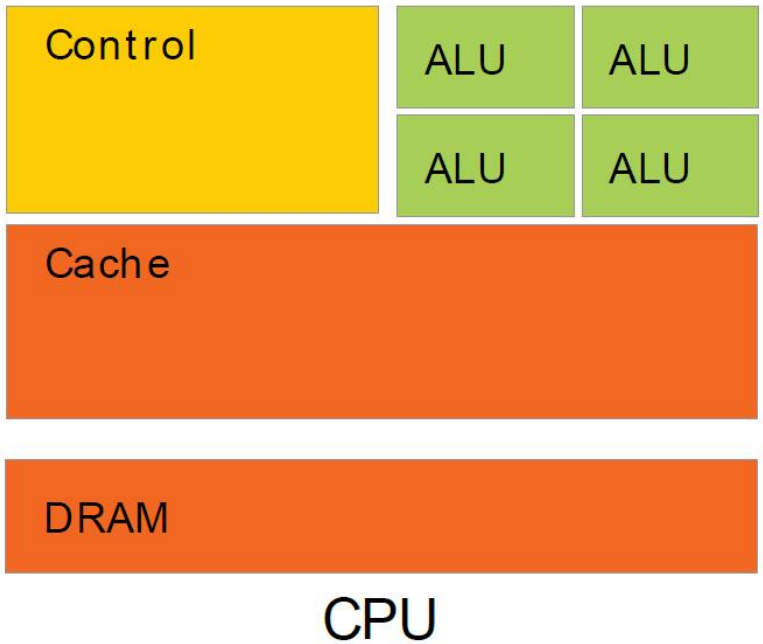
CPU 和 GPU 对比

CPU = Central Processing Unit：

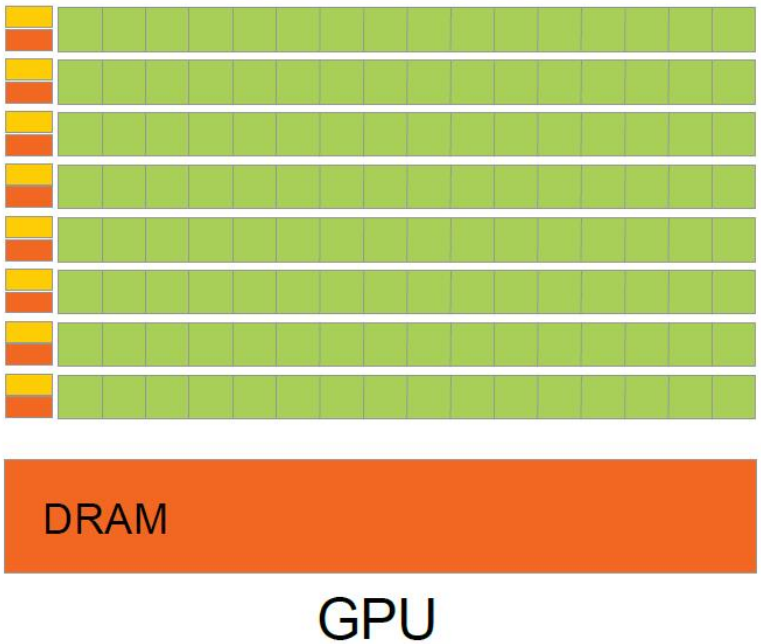
- 面向延迟的设计；
- 非常大的 cache；
- 复杂的控制逻辑；
- 强大的 ALU；
- 不多的线程。

GPU = Graphic Processing Unit：

- 面向通量的设计；
- 小的 cache；
- 简单的控制逻辑；
- 高能效的 ALU；
- 大量的线程。



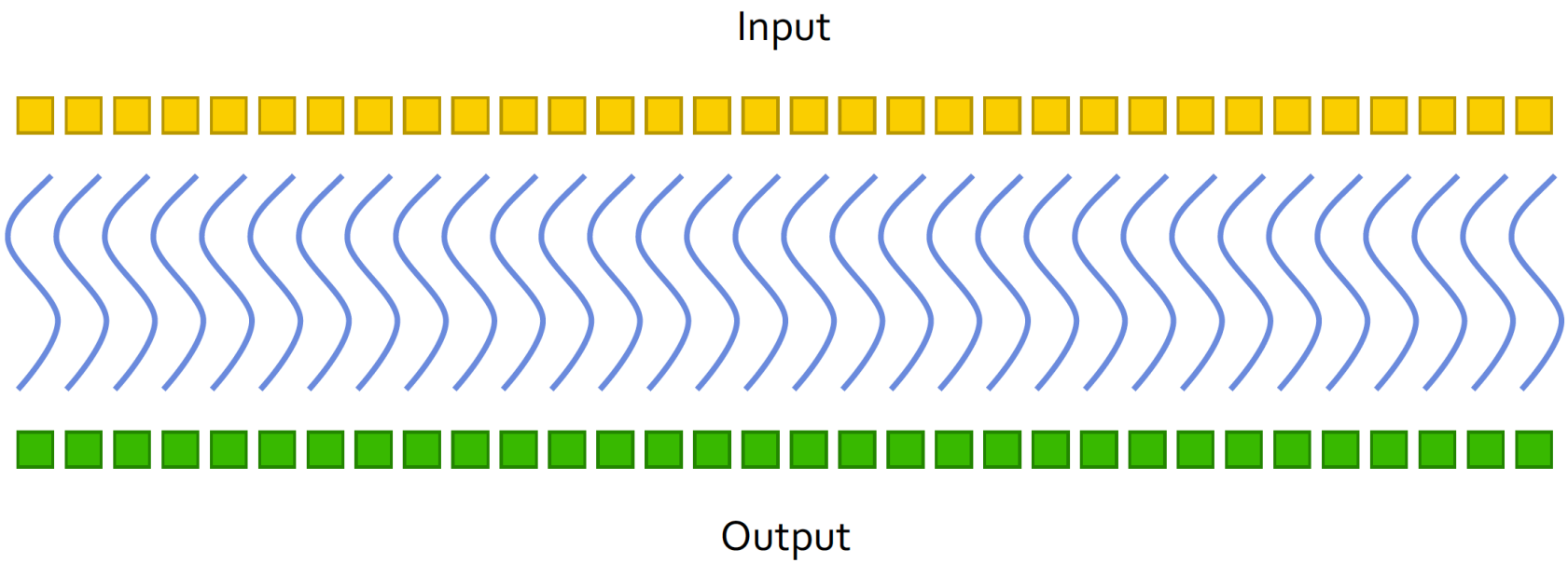
CPU



GPU

GPU 的执行模式

- **Calculations on a GPU always follow the same model.**
- **This is very constrained.**
- **GPUs run kernels that are designed to execute calculations in the following way.**



等一下！GPU 不是打游戏的吗?

从 2005 年左右说起：GPGPU (General Purpose GPU)

Stream-based programming model

Express algorithms in terms of graphics operations

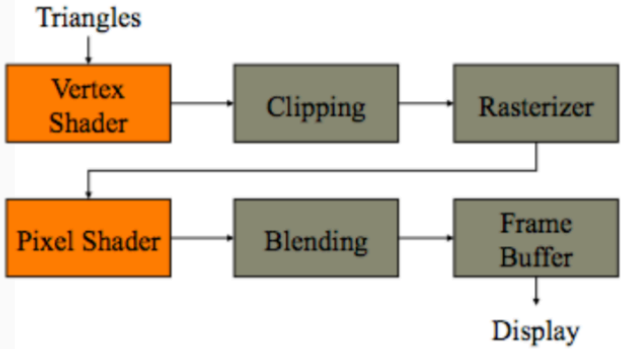
—use GPU pixel shaders as general-purpose SP floating point units

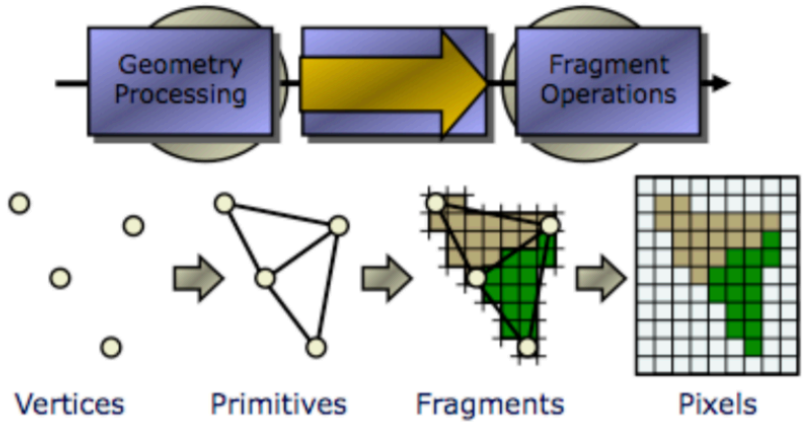
Directly exploit

—pixel shaders

—vertex shaders

—video memory

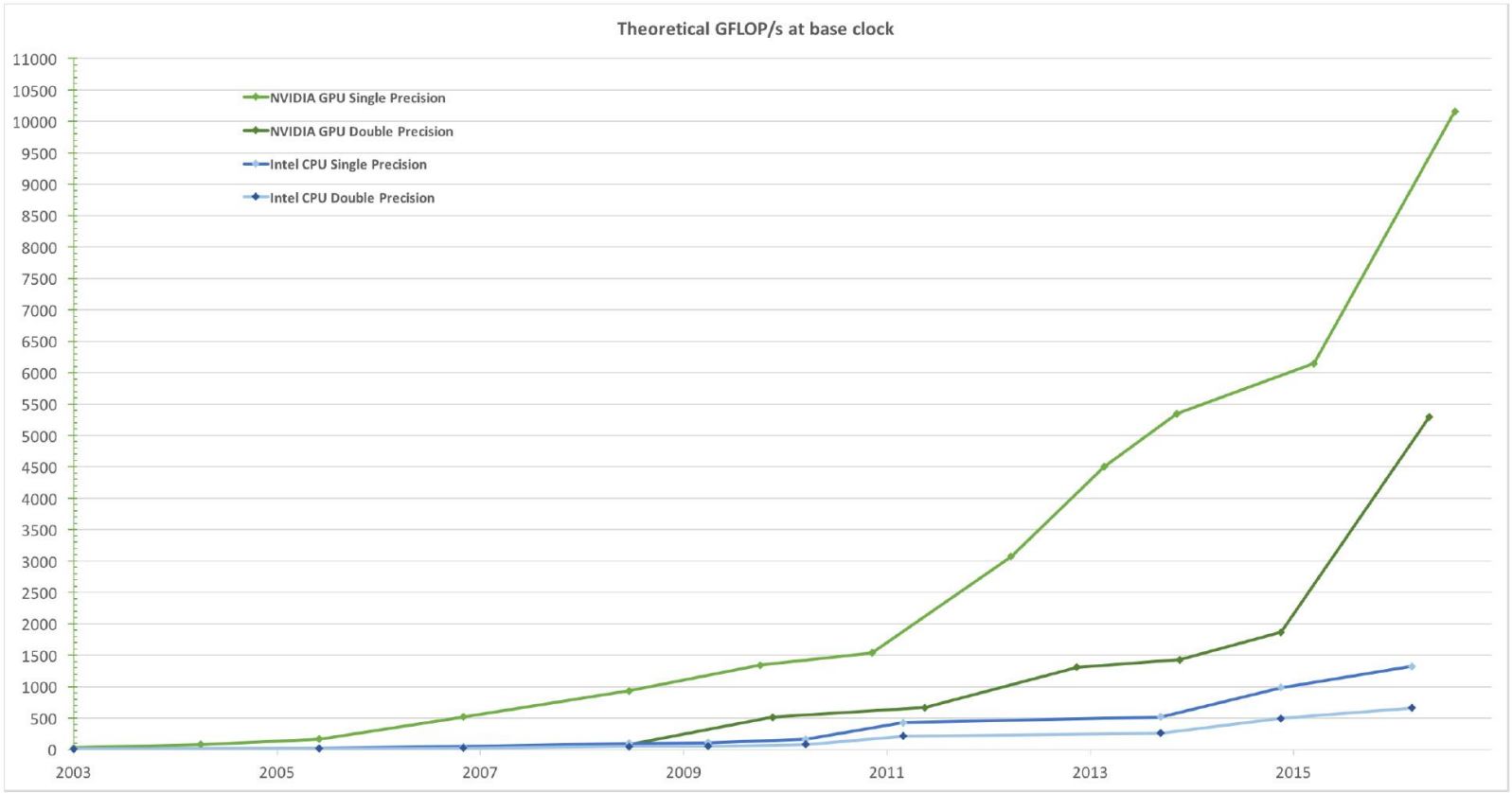




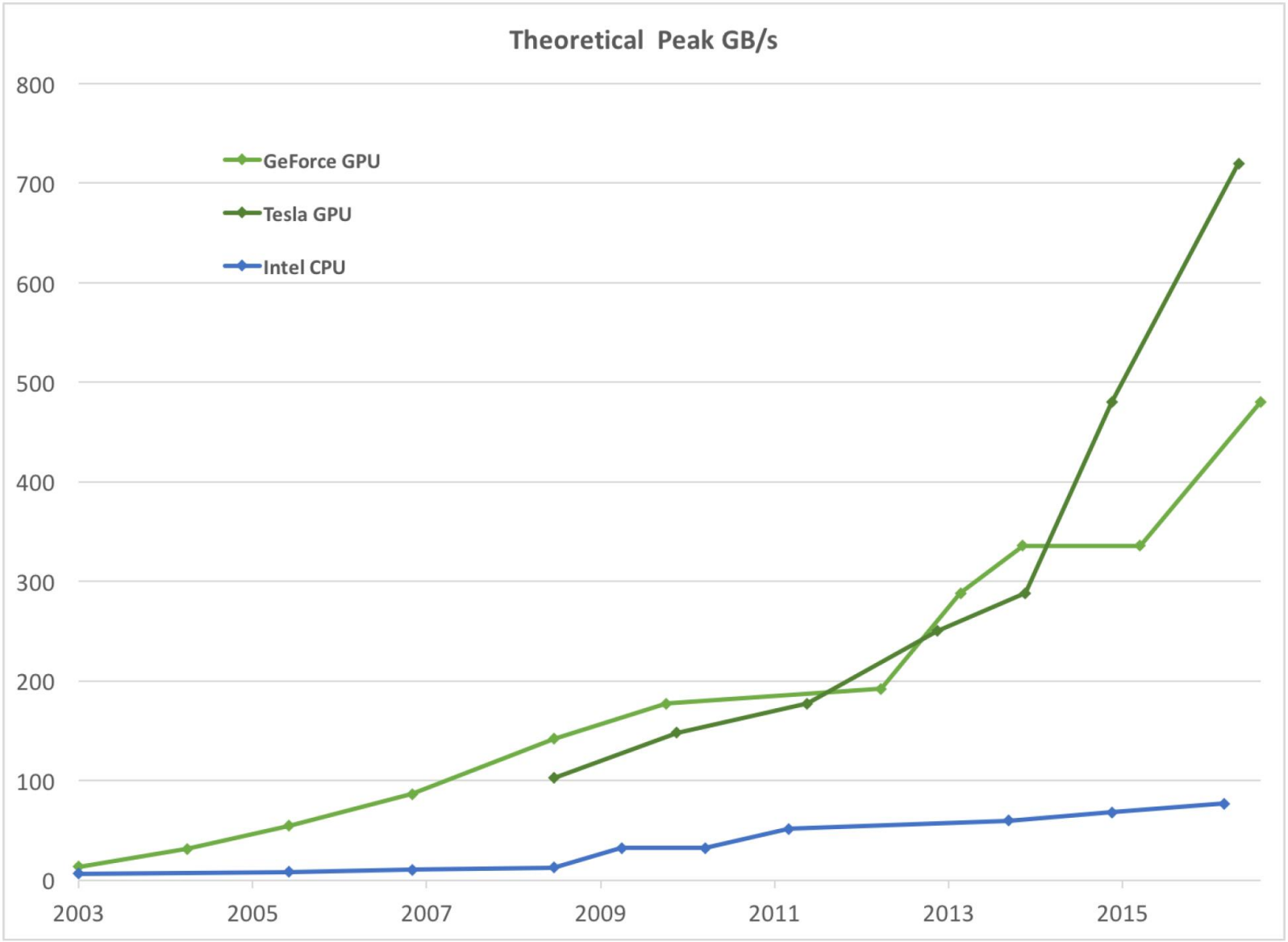
threads interact through off-chip video memory

Example: GPUSort (Govindaraju, Manocha; 2005)

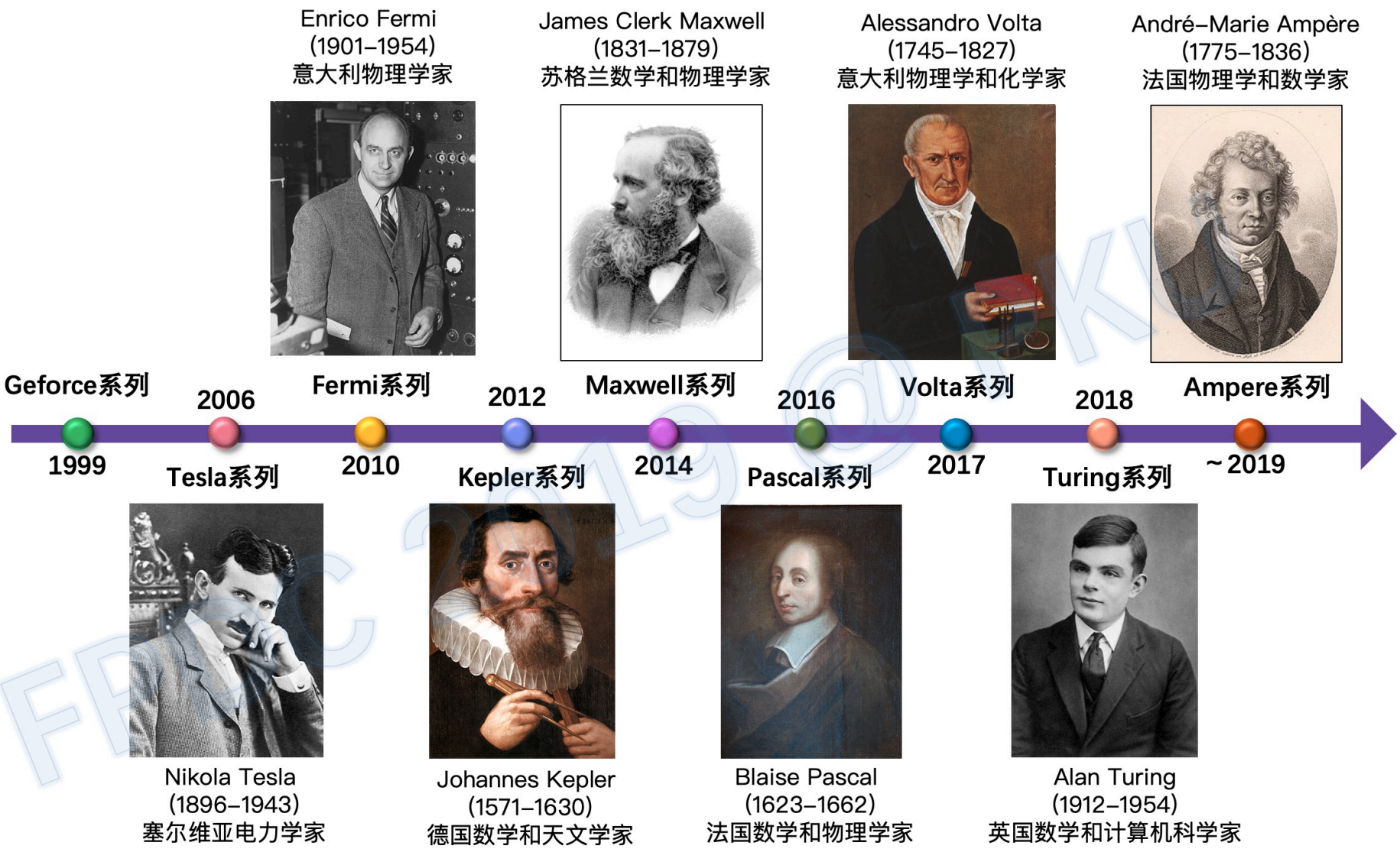
GPU 浮点计算能力的优势



GPU 访存能力的优势


Theoretical Peak GB/s
800
700
600
500
400
300
200
100
0
GeForce GPU
Tesla GPU
Intel CPU
2003
2005
2007
2009
2011
2013
2015

NVIDIA 的 GPU 微架构分类



NVIDIA 的 GPU 产品分类

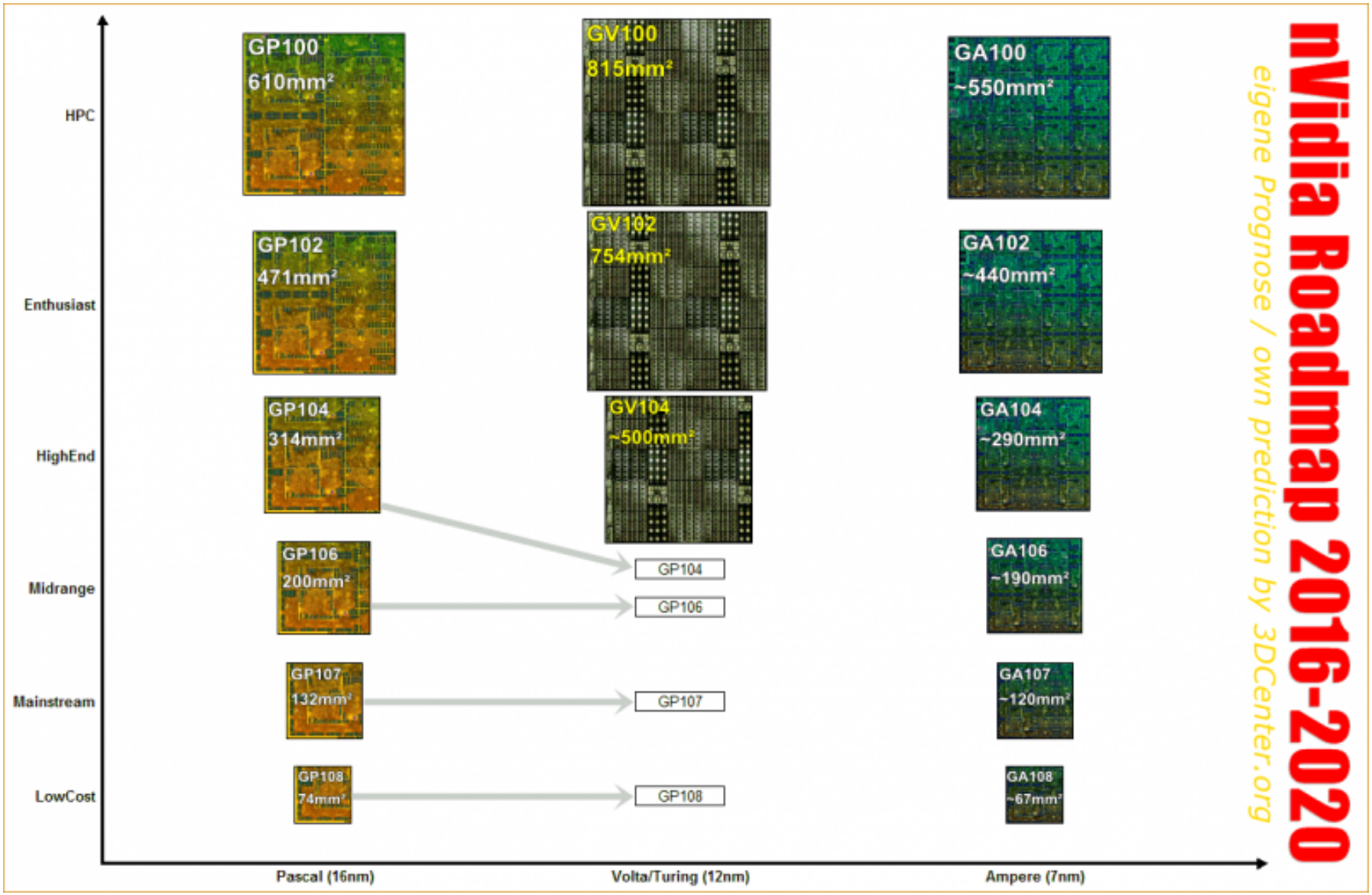
桌面 (desktop) GeForce系列	移动 (mobile) GeForce M系列	工作站 (workstation) Quadro系列	高性能计算 (HPC) Tesla系列
			
例：GeForce GTX 780 发布：2012.03 架构：Kepler GK110	例：GeForce GTX 970M 发布：2014.10 架构：Maxwell GM204	例：Quadro FX 5800 发布：2008.11 架构：Tesla GT200GL	例：Tesla V100 发布：2017.06 架构：Volta GV100

Nvidia早期产品：Riva, Vanta, TNT, Tegra, ...

此外，还有Titan等融合了不同特色的系列产品

更多信息： https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units

NVIDIA 的 2016-2020 路线图



几台典型的 GPU 超级计算机

- 中国天河 1 号 A (2010): 7,168 Tesla M2050 GPUs (架构：GF100)；
- 美国 Titan (2012): 18,688 Tesla K20x GPUs (架构：GK110)；
- 日本 TSUBAME 3.0 (2017): 2,160 Tesla P100 GPUs (架构：GP100)；
- 瑞士 Piz Daint (2017): 5,704 Tesla P100 GPUs (架构：GP100)；
- 美国 Summit (2018): 27,648 Tesla V100 GPUs (架构：GV100)；
- ...
- 数院机器 (2018): 40 Titan Xp (GeForce) GPUs (架构：GP102)。

Tesla P100 的计算性能

- 双精度峰值（FP64）5.3 TFLOPS
- 单精度峰值（FP32）10.6 TFLOPS
- 半精度峰值（FP16）21.2 TFLOPS
- 相对于 CPU，对于 DNN 的不同应用来说，加速 10 到 20 倍

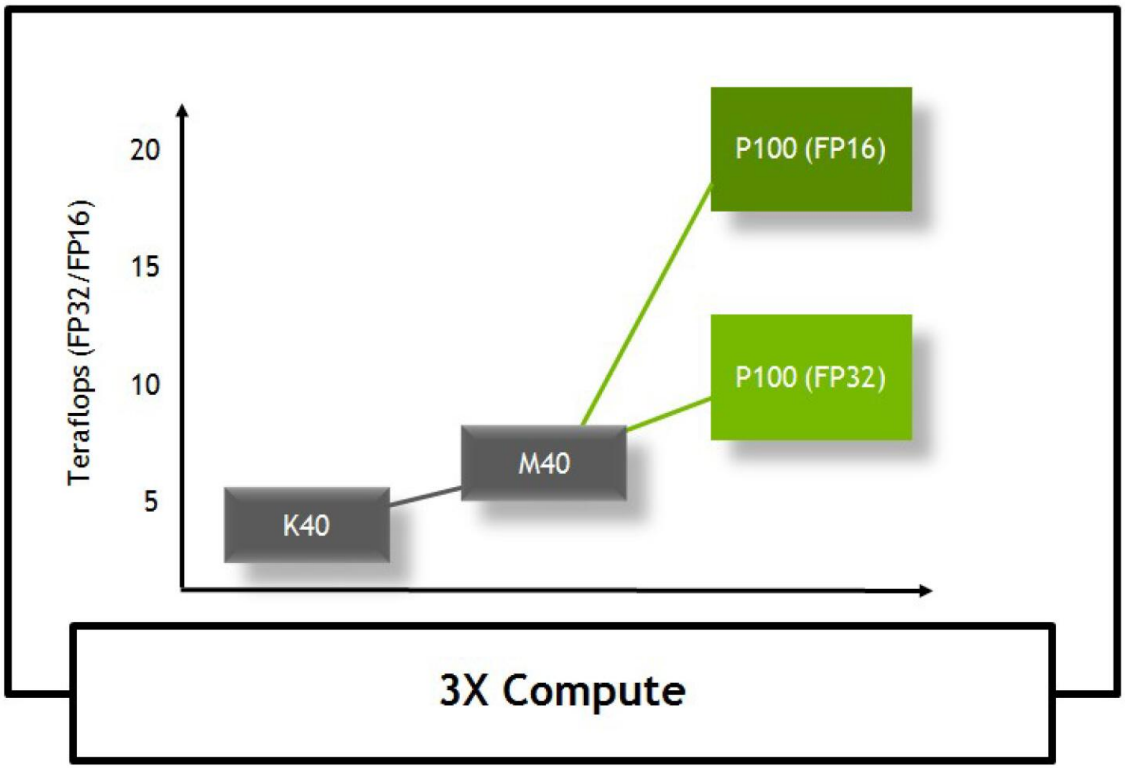
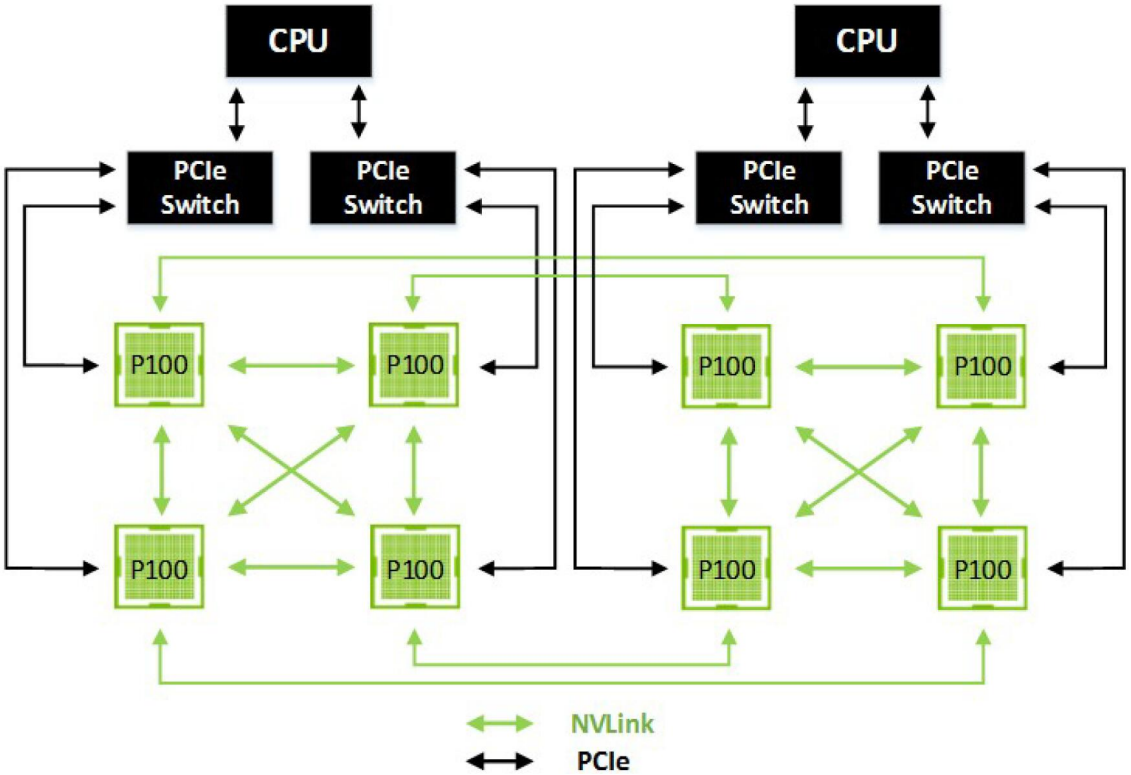


图: P100 和过去两代 GPU 的计算性能对比

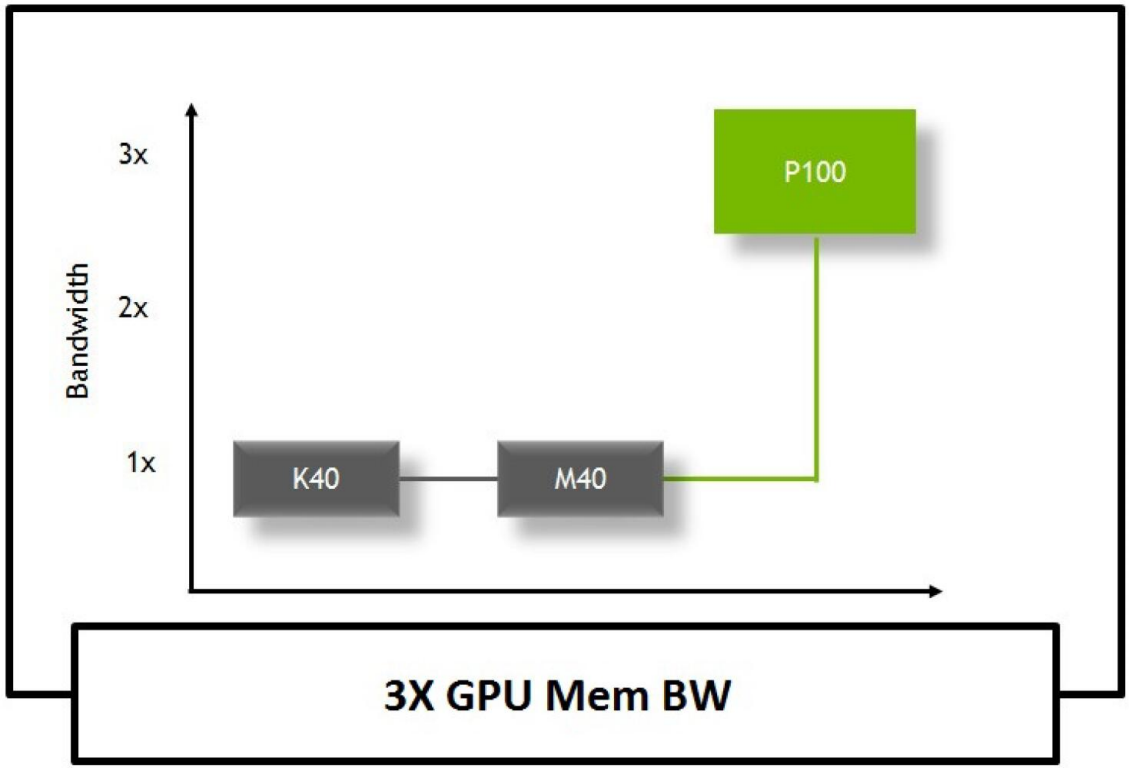
Tesla P100 的芯片互连

- NVLINK 能够在 GPU-to-GPU 间提供 160 Gigabytes/second 的双向带宽。相对于 PCIe Gen 3 x16 来说，数据传输能力提升 5 倍。



Tesla P100 的访存带宽

- 在 P100 中，首次采用了 High Bandwidth Memory 2 (HBM2) 堆叠式存储，带宽相对于 Maxwell GM200 GPU 提升了 3 倍。



Titan Xp (GP102) vs Tesla P100 (GP100)

Model	NVIDIA Tesla P100	NVIDIA Titan Xp
Core name	GP100	GP102
Transisitor Count	15.3B	12B
Die Size	610mm^2	471mm^2
Manufacturing Process	TSMC 16nm FF+	TSMC 16nm FF+
Stream Processors	3584	3840
SM or Cus	56 SM	60 SM
FP32 Cores	3584	3840
FP64 Cores	1792 (1: 2)	120 (1: 32)
Tensor Cores	N/A	N/A
TA/TF	224	240
ROP/RBE	?	96

Boost Clock	1480MHz	1582MHz
FP16 TFLOPS	21TFLOPS	24TFLOPS
FP32 TFLOPS	10.6TFLOPS	12.1TFLOPS
FP64 TFLOPS	5.3TFLOPS	0.38TFLOPS
Tensor TFLOPS	N/A	N/A
L2 Cache	4096KB	4096KB
Memory Size	16GB	12GB
Memory Clock	1.44Gbps	11.4Gbps
Memory Interface	4096bit HBM2	384bit GDDR5X
Memory Bandwidth	720GB/s	547.7GB/s
PCI-E/NVLink	NVLink/PCI-E	PCI-E
TDP	300W	250W

Tesla V100 的计算性能

- 双精度峰值（FP64）7.8 TFLOPS (P100: 5.3 TFLOPS)
- 单精度峰值（FP32）15.7 TFLOPS (P100: 10.6 TFLOPS)
- 半精度峰值（FP16*）125 TFLOPS (P100: 21.2 TFLOPS)

  *: 基于 Tensor Core 张量计算核心的混合精度实现

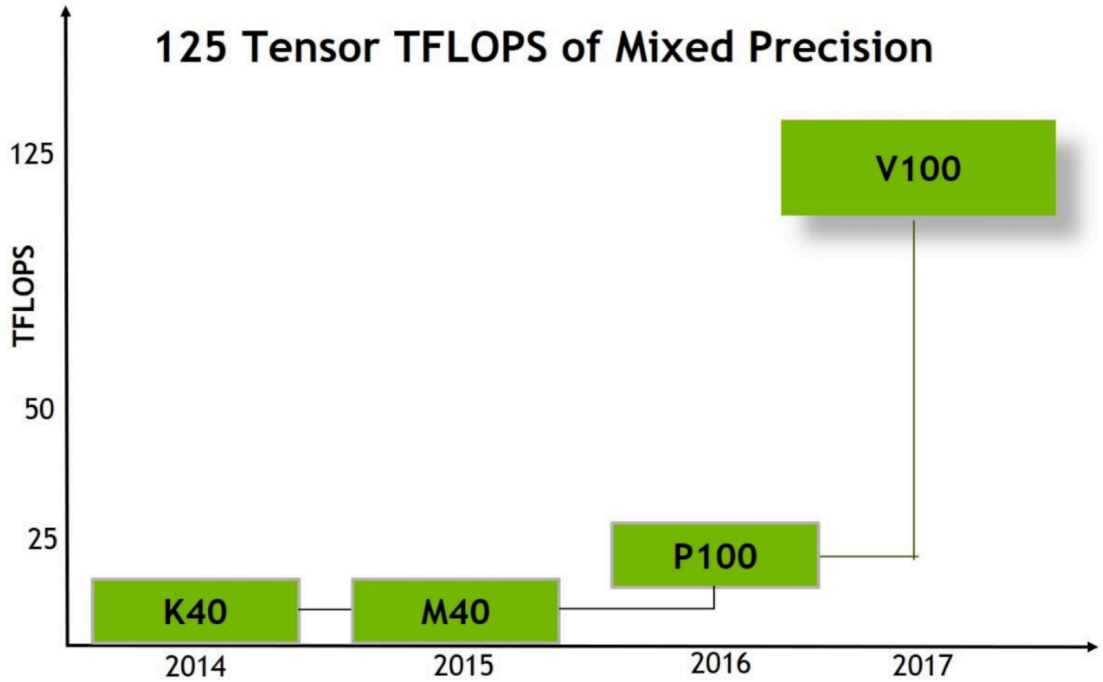
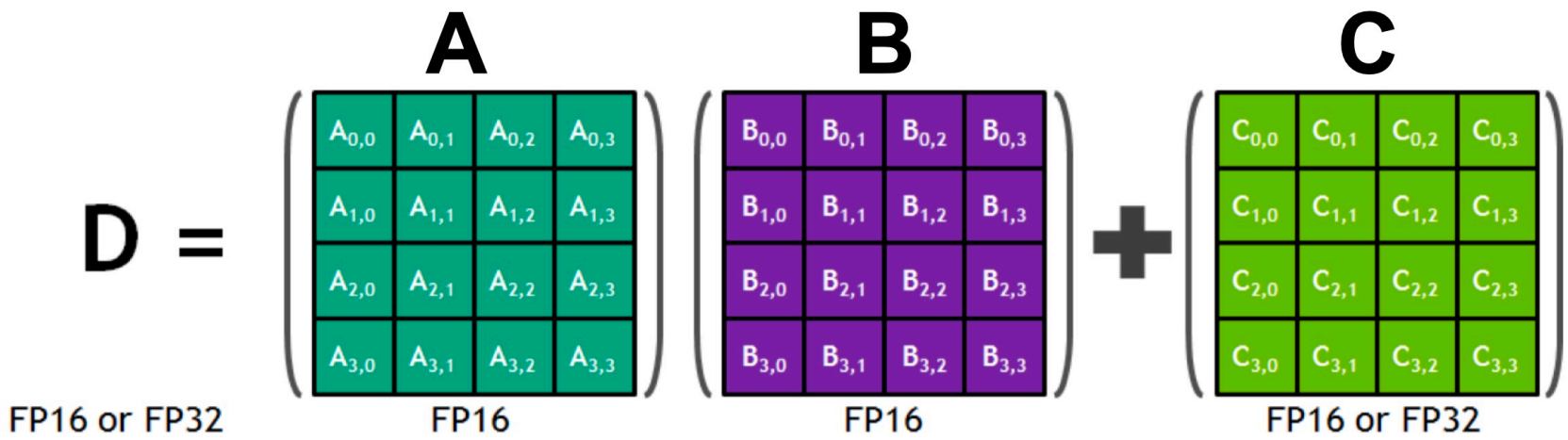
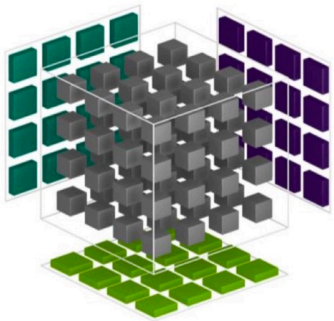


图: V100 和过去三代 GPU 的计算性能对比

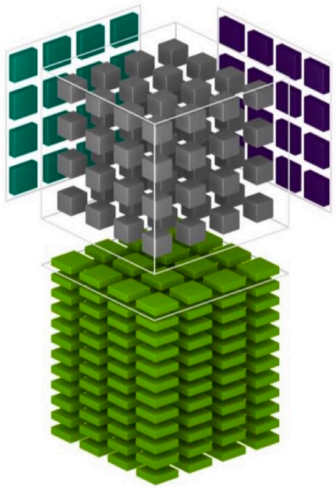
Tesla V100 的 Tensor Core



Pascal



Volta Tensor Core



Tensor Core GEMM

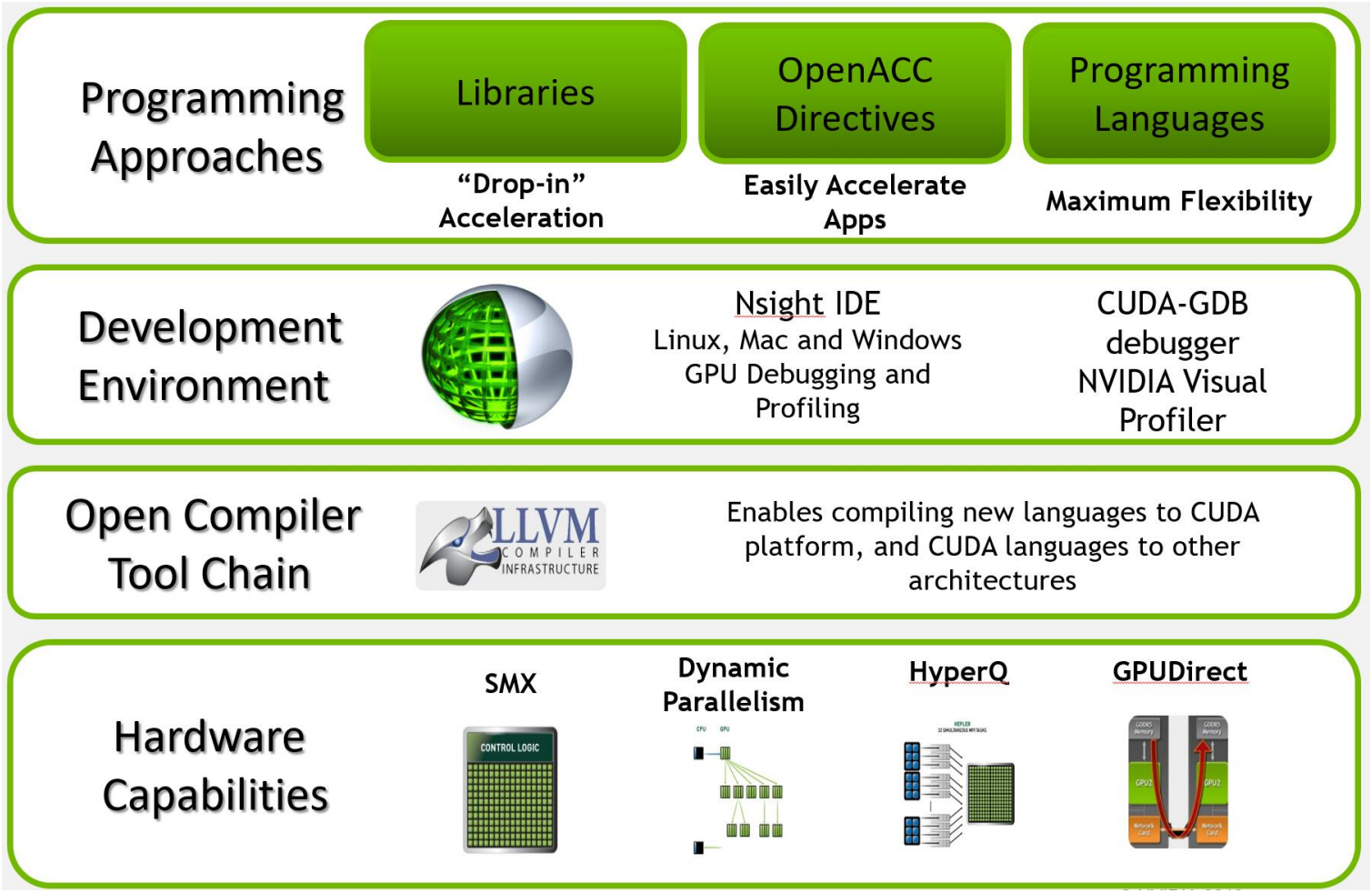
- A, B, C, D are 4x4 matrices
- A and B are FP16
- C and D are FP16 or FP32
- Products are FP64
- Additions are FP32

Note: GEMM=General Matrix Multiplication

初识 CUDA

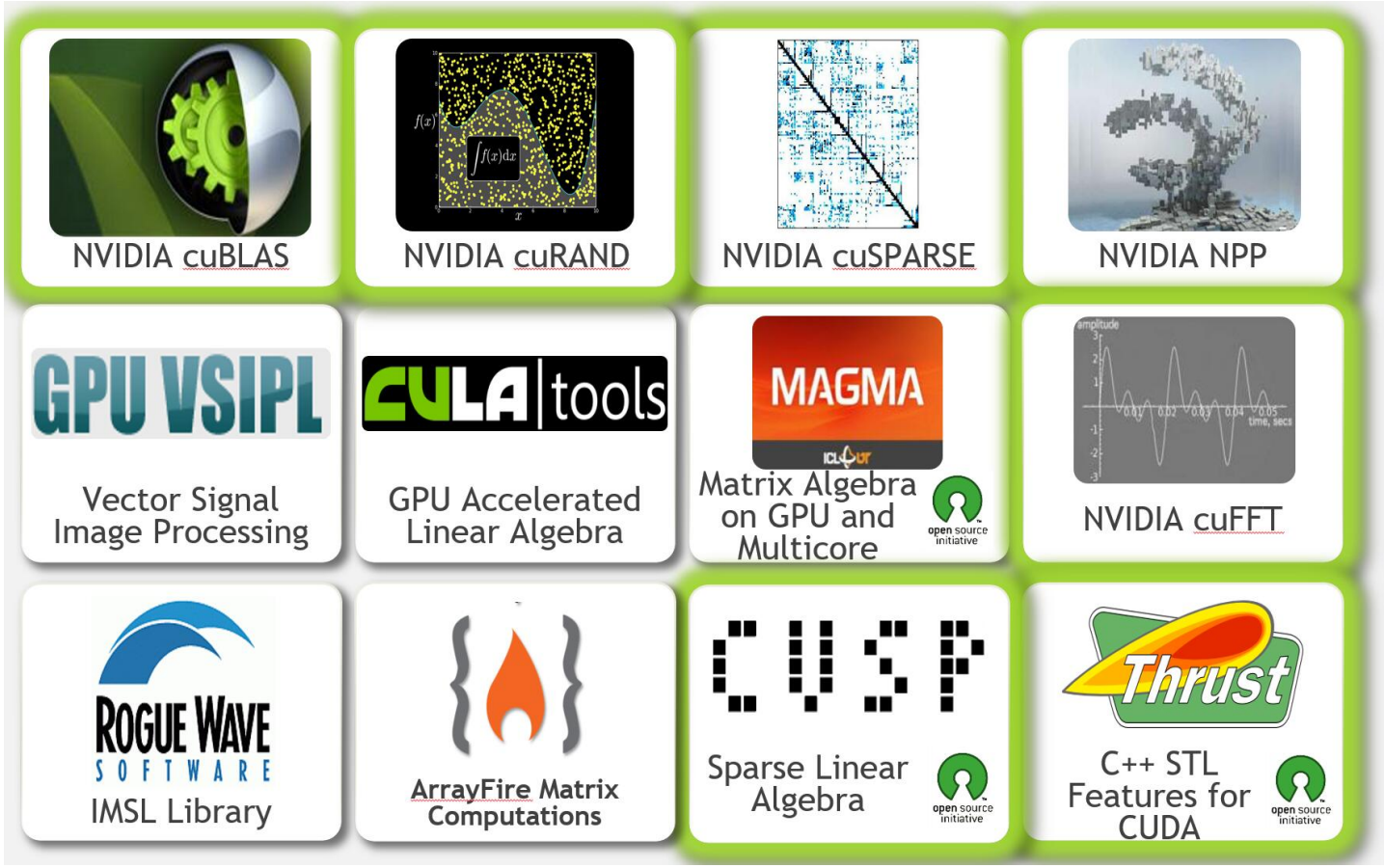
CUDA 并行编程平台

CUDA (Compute Unified Device Architecture)：由 NVIDIA 公司于 2007 年推出旨在进行通用 GPU 计算的并行计算平台和并行编程接口。



方式一：使用 CUDA 计算库

- 主要特点：
  - 最简单，基本不需要了解 GPU 并行编程知识；
  - 库具有很高的代码质量，性能经受过专业人士调优。



方式二：使用 OpenACC 编译指示

- 主要特点：
  - 较简单，加入编译制导语句能实现自动并行；
  - 较灵活，能够快速实现多 GPU 和 CPU 上并行执行。

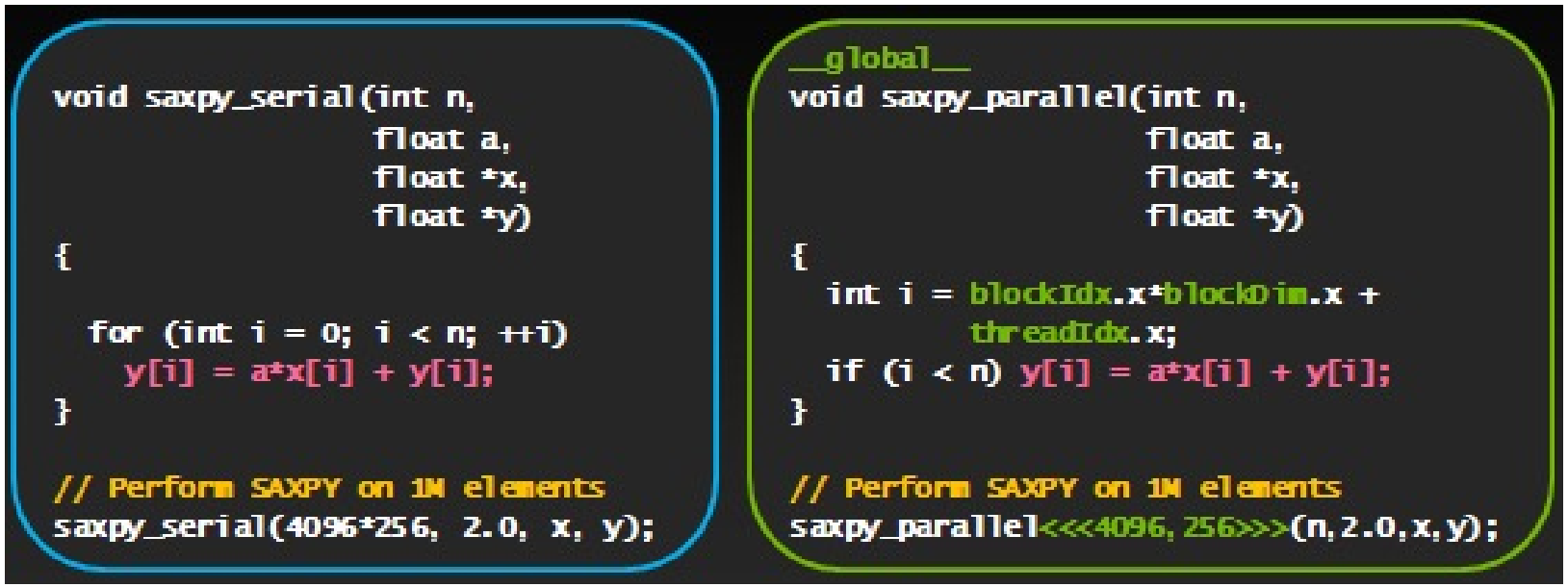
Matrix-vector multiplication

```
#pragma acc data copyin(a[0:n*m])
{
  ...
  #pragma acc data copyin(v[0:n]) \
                    copyout(x[0:n])
  {
    ...
    matvecmul( x, a, v, m, n );
    ...
  }
  ...
}
```

```
void matvecmul( float* x, float* a,
                float* v, int m, int n ){
#pragma acc parallel loop gang \
  pcopyin(a[0:n*m],v[0:n]) pcopyout(x[0:m])
  for( int i = 0; i < m; ++i ){
    float xx = 0.0;

#pragma acc loop worker reduction(+:xx)
    for( int j = 0; j < n; ++j )
      xx += a[i*n+j]*v[j];
    x[i] = xx;
  }
}
```

方式三：使用 CUDA 编程语言

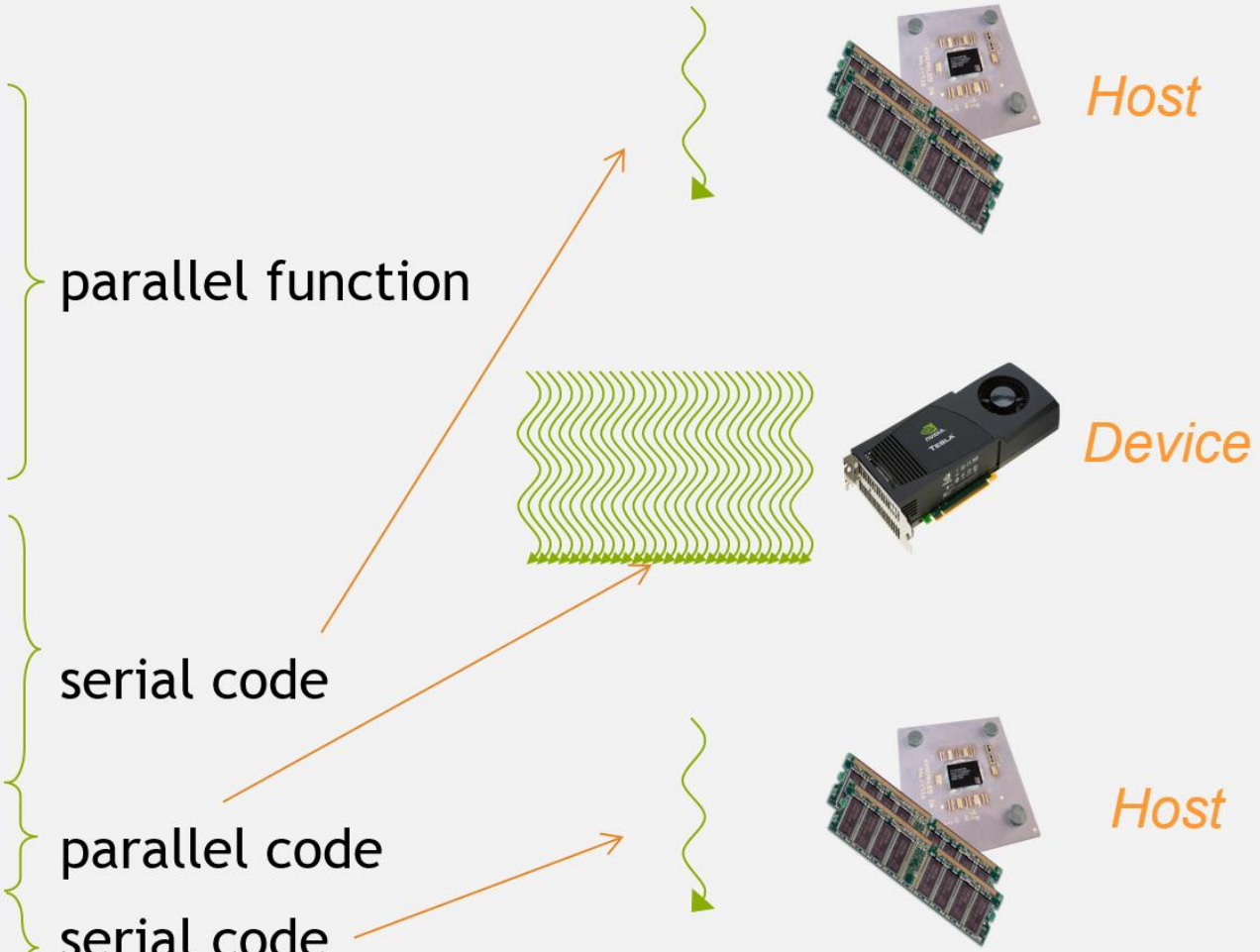
- 主要特点：
  - 较复杂，需要深入了解 GPU 并行编程知识；
  - 能提供最大的灵活性和性能需求。



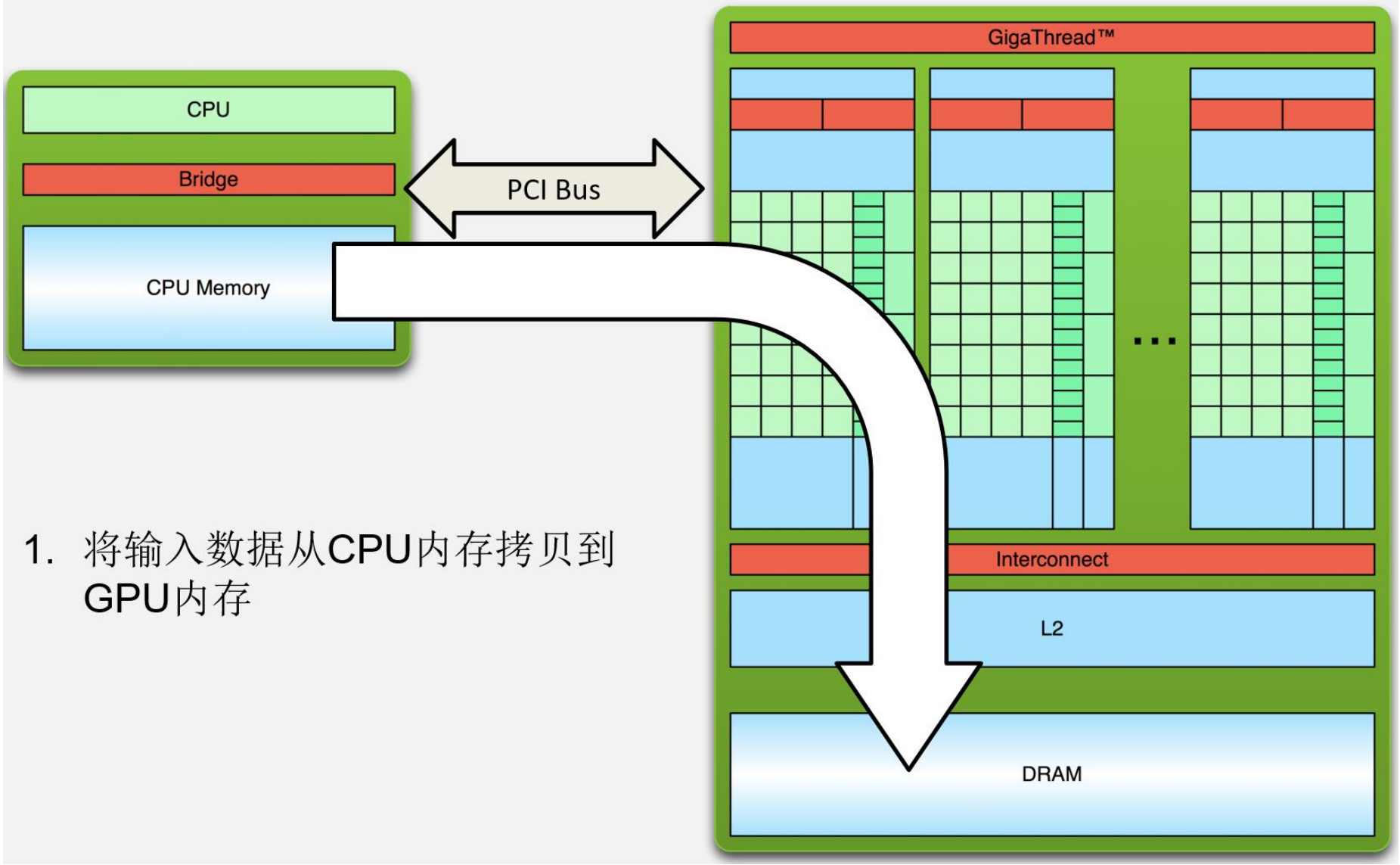
C　　CUDA

CUDA 异构编程方式

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

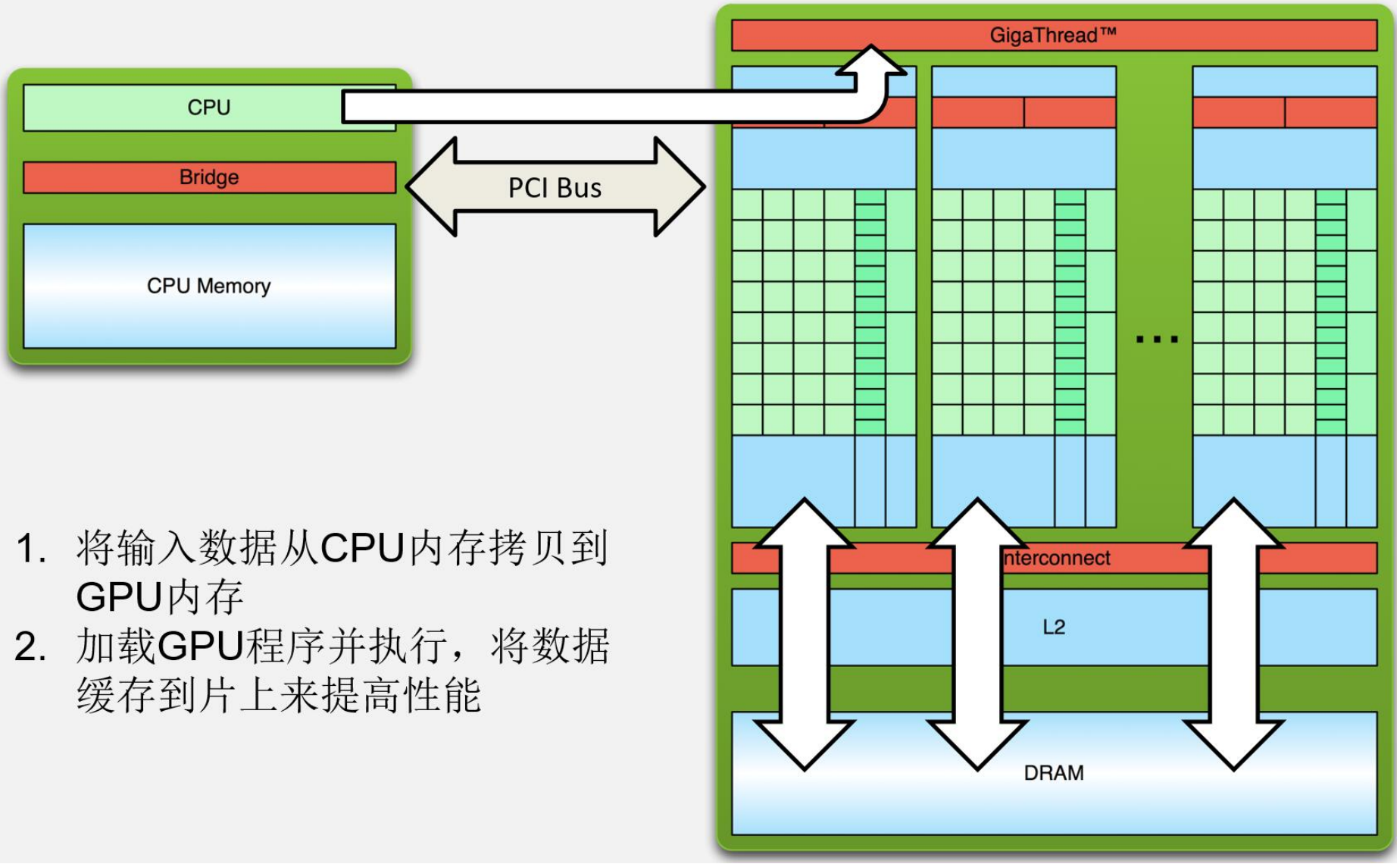


CUDA 异构并行计算流程



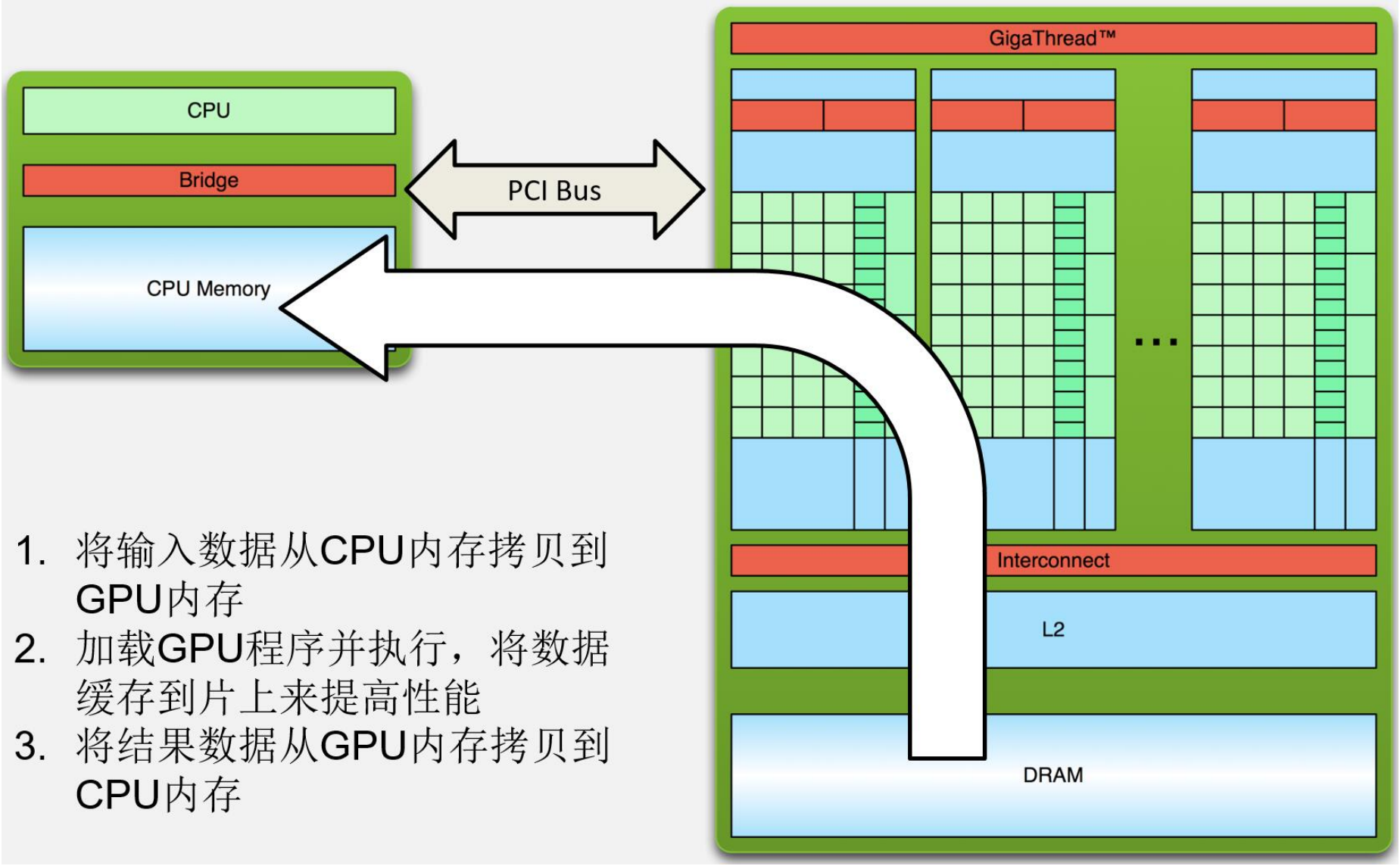
1. 将输入数据从CPU内存拷贝到GPU内存

CUDA 异构并行计算流程



1. 将输入数据从CPU内存拷贝到GPU内存
2. 加载GPU程序并执行，将数据缓存到片上来提高性能

CUDA 异构并行计算流程



1. 将输入数据从CPU内存拷贝到GPU内存
2. 加载GPU程序并执行，将数据缓存到片上来提高性能
3. 将结果数据从GPU内存拷贝到CPU内存

第一个 CUDA 程序：hello world!

cuda_hello.cu

```
1  __global__ void mykernel(void) {
2    printf("Hello World from GPU by thread %d!\n", threadIdx.x);
3  }
4  int main(void) {
5    // Use 1 thread
6    mykernel<<<1,1>>>();
7    // Flush the output
8    cudaDeviceSynchronize();
9    // Use 4 threads
10   mykernel<<<1,4>>>();
11   // Flush the output
12   cudaDeviceSynchronize();
13   return 0;
14 }
```

程序的编译与运行

- 加载编译器：

```
$ module load cuda
```

- 编译：

```
$ nvcc -o hello cuda_hello.cu
```

- 运行 (请正确使用 sbatch 或者 salloc)：

```
$ ./hello
```

运行结果

- 运行结果:

```
Hello World from GPU by thread 0!
Hello World from GPU by thread 0!
Hello World from GPU by thread 1!
Hello World from GPU by thread 2!
Hello World from GPU by thread 3!
```