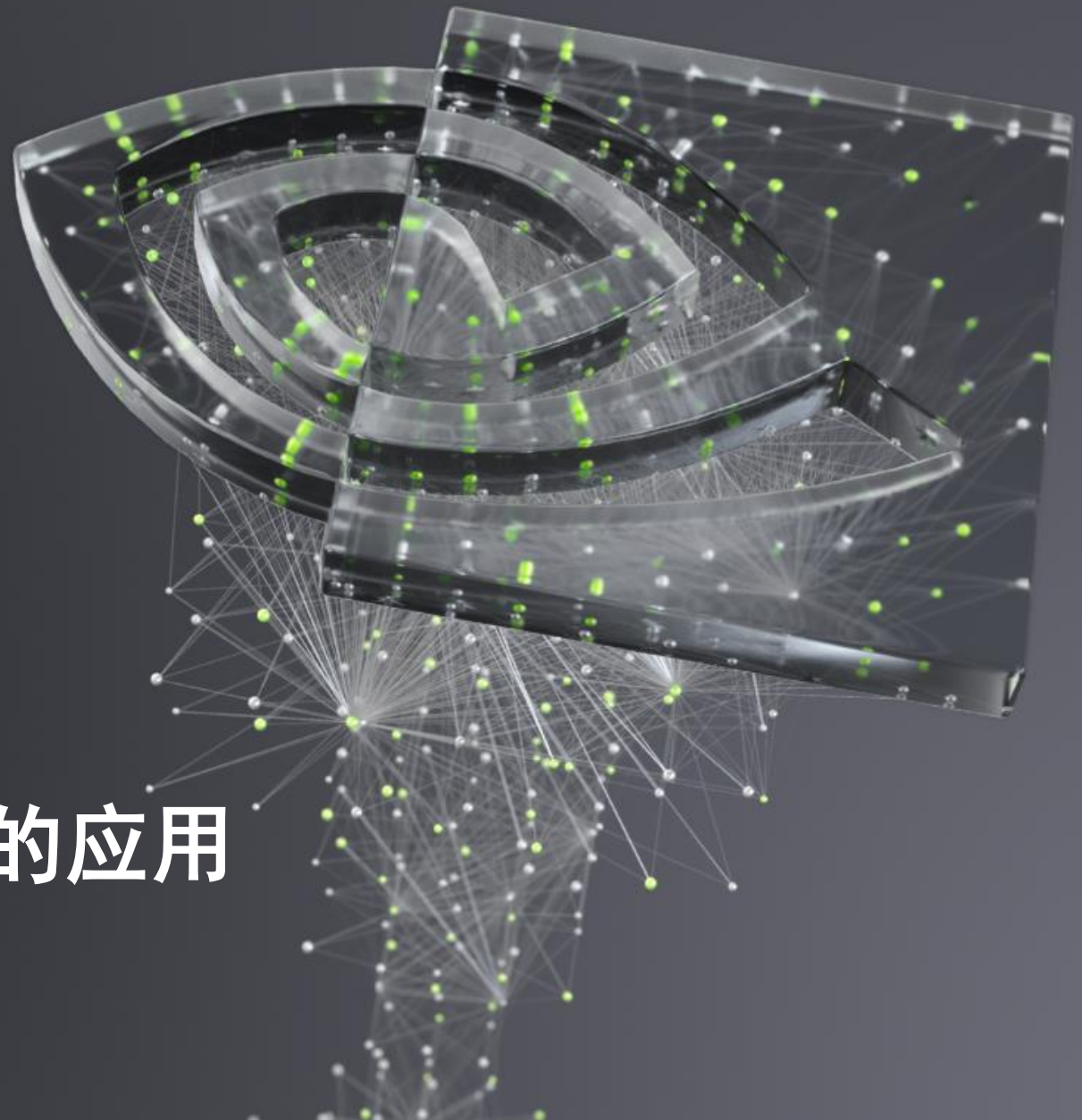
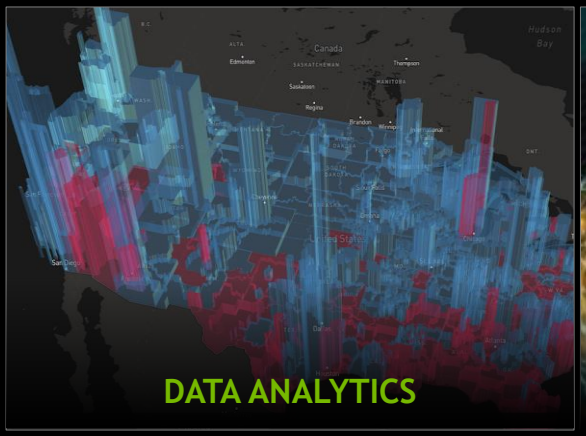


GPU在高性能计算中的应用

易成 | 方案架构师

GPU科学计算平台

Accelerating End-to-End Scientific Workflows



SPARK3.0 | RAPIDS | more

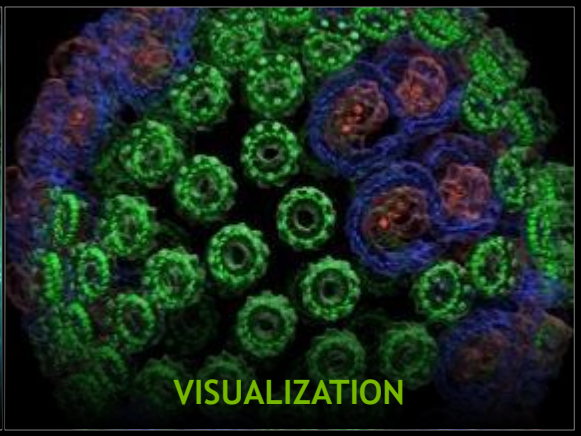
cuDF | cuML | cuGRAPH

NAMD | GROMACS | +700 More

cuBLAS| cuFFT | cuSOLVER

TensorFlow | PyTorch | more

cuDNN| TensorRT | NCCL



ParaView | IndeX | more

NvENC | Thrust| VisRTX

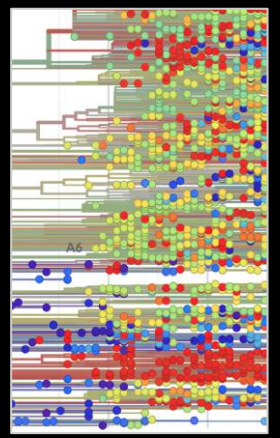
Desktop Development

Data Center Solutions

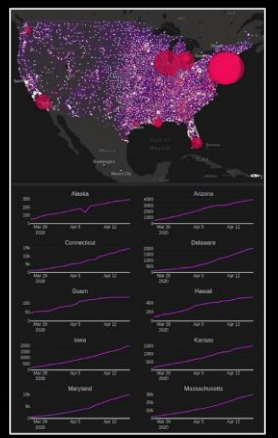
Supercomputers

GPU-Accelerated Cloud

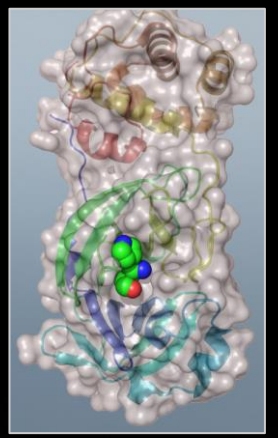
GPU在抗击新冠病毒COVID-19中的应用



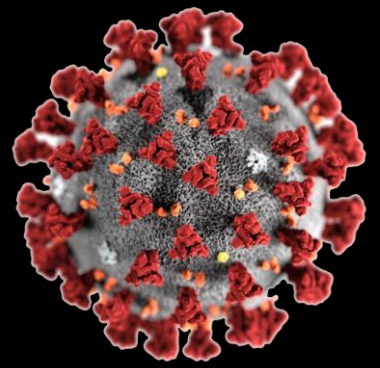
Oxford Nanopore
Sequence Viral Genome
in 7Hrs



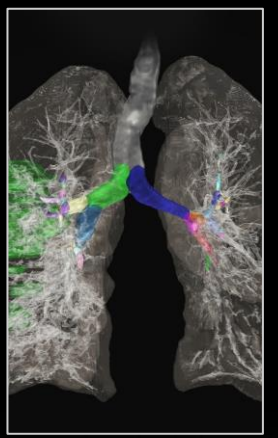
Plotly, NVIDIA
Real-Time
Infection Rate Analysis



ORNL, Scripps
Screen
2B Drug Compounds in
1 Day vs 1 Year

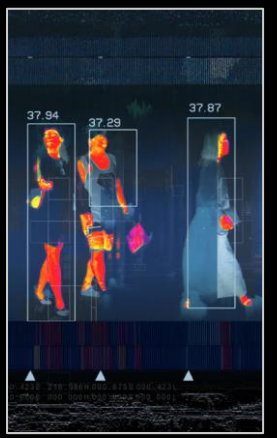


Structura, NIH, UT Austin
CryoSPARC
1st 3D Structure of Virus Spike Protein



NIH, NVIDIA
AI COVID-19
Classification

Kiwibot
Robot Medical Supply
Delivery



Whiteboard Coordinator
AI Elevated Body Temp
Screening System

Data Analytics — Simulation & Visualization — AI — Edge

GPU加速HPC应用

500+ APPLICATIONS

LIFE SCIENCES

50+ app

Including:
- Gaussian
- VASP
- AMBER
- HOOMD-Blue
- GAMESS

MFG, CAD, & CAE

111 apps

Including:
- Ansys Fluent
- Abaqus SIMULIA
- AutoCAD
- CST Studio Suite

PHYSICS

20 apps

Including:
- QUDA
- MILC
- GTC-P

OIL & GAS

17 apps

Including:
- RTM
- SPECFEM 3D

CLIMATE & WEATHER

4 apps

Including:
- Cosmos
- Gales
- WRF

DEEP LEARNING

32 apps

Including:
- Caffe2
- MXNet
- Tensorflow

MEDIA & ENT.

142 apps

Including:
- DaVinci Resolve
- Premiere Pro CC
- Redshift Renderer

FEDERAL & DEFENSE

13 apps

Including:
- ArcGIS Pro
- EVNI
- SocetGXP

DATA SCI. & ANALYTICS

23 apps

Including:
- MapD
- Kinetica
- Graphistry

SAFETY & SECURITY

15 apps

Including:
- Cyllance
- FaceControl
- Syndex Pro

COMP. FINANCE

16 apps

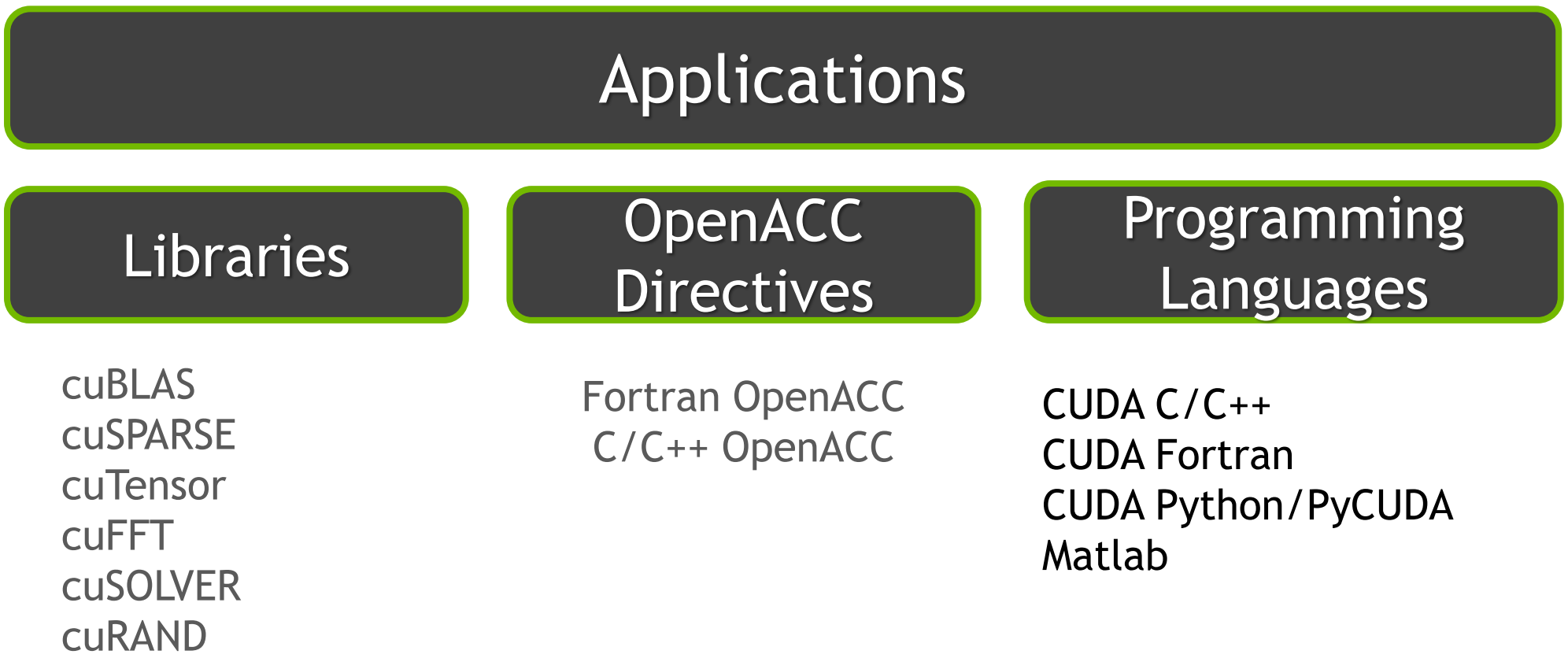
Including:
- O-Quant Options Pricing
- MUREX
- MISYS

TOOLS & MGMT.

15 apps

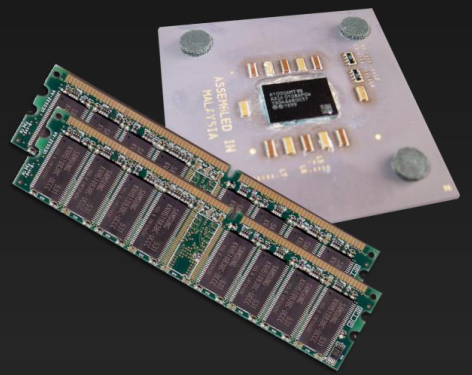
Including:
- Bright Cluster Manager
- HPCtoolkit
- Vampir

3种GPU加速方法


Applications
Libraries
OpenACC Directives
Programming Languages
cuBLAS
cuSPARSE
cuTensor
cuFFT
cuSOLVER
cuRAND
Fortran OpenACC
C/C++ OpenACC
CUDA C/C++
CUDA Fortran
CUDA Python/PyCUDA
Matlab

Heterogeneous Computing (异构计算)

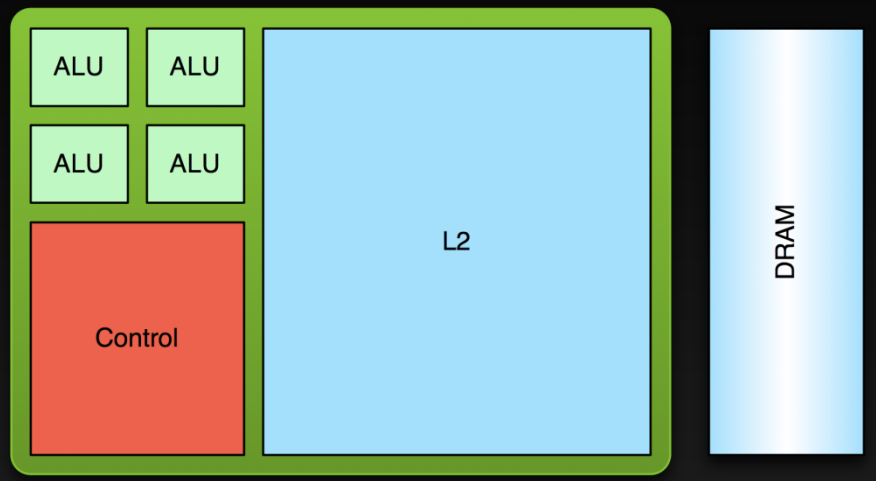
- 术语:
  - *Host* (主机) CPU 和它的内存 (host memory)
  - *Device* (设备) GPU 和它的内存 (device memory)



Host

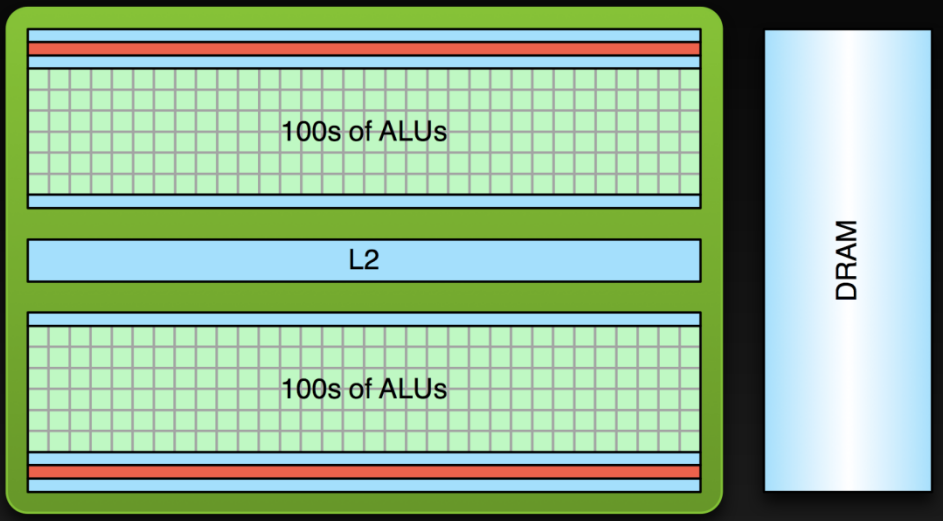
Device

低延迟 VS 高吞吐量



CPU

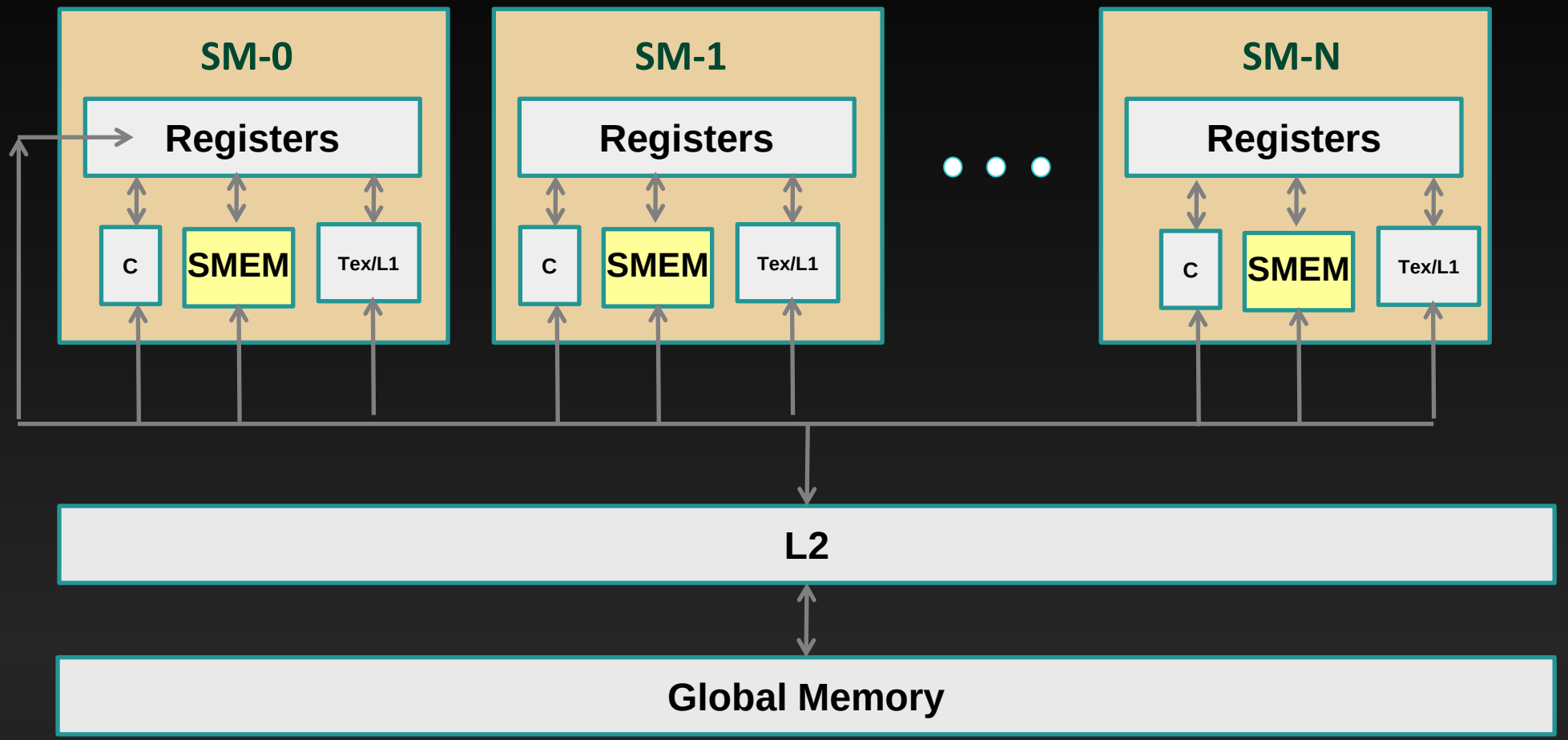
- 通过复杂的缓存体系结构，减小指令延迟
- 大量的晶体管处理逻辑任务



GPU

- 集成大量计算单元，获得高吞吐量
- 更多的晶体管用于数学计算

GPU 内存组织

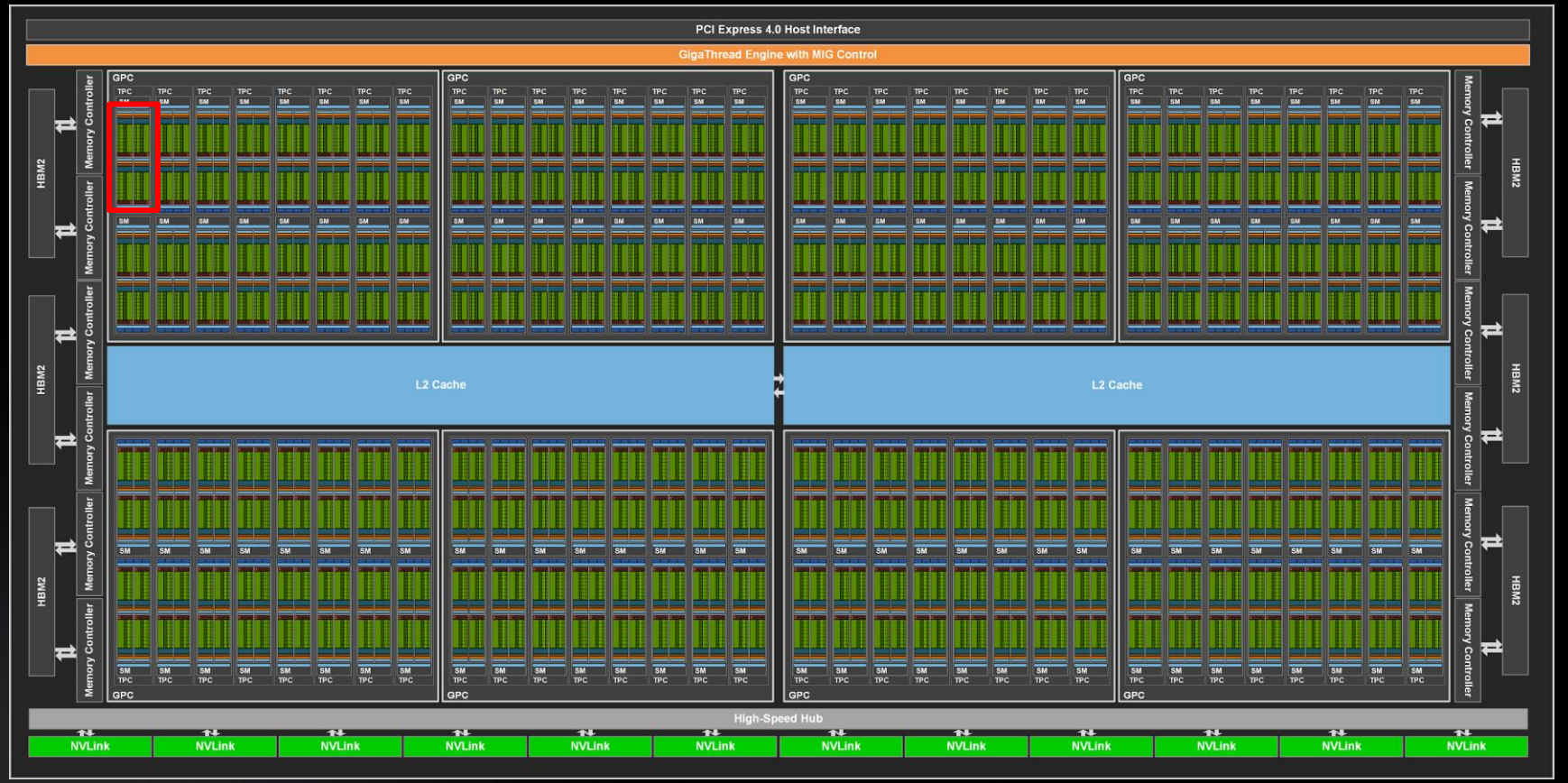

SM-0
Registers
C
SMEM
Tex/L1
SM-1
Registers
C
SMEM
Tex/L1
SM-N
Registers
C
SMEM
Tex/L1
L2
Global Memory

GPU 芯片架构

54B transistors
826 mm^2

108 SM
6912 FP32 CUDA Cores
432 Tensor Cores

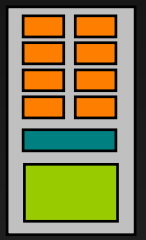
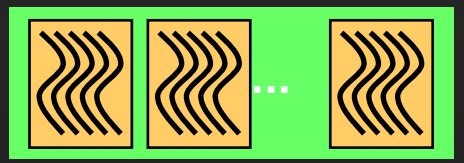
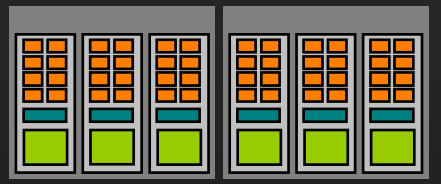
40 GB HBM2
1600 GB/s HBM2
600 GB/s NVLink



GPU A100 SM (Streaming Multiprocessor)

	GA100
SM/GPU	108
FP32 units	64
FP64 units	32
TensorCore	4
Register/SM	256 KB
Unified Shared Memory/ L1 Cache	164 KB

GPU CUDA 编程模型

Software	GPU	
 Thread	 CUDA Core	Threads are executed by cuda core
 Thread Block	 SM	Thread blocks are executed on SM
 Grid	 Device	A kernel is launched as a grid of thread blocks

Heterogeneous Computing

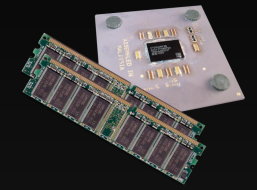
```
#include <iostream>
#include <algorithm>

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;
}
```

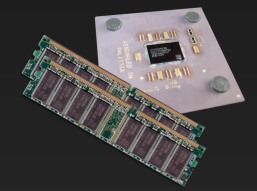
parallel fn

serial code

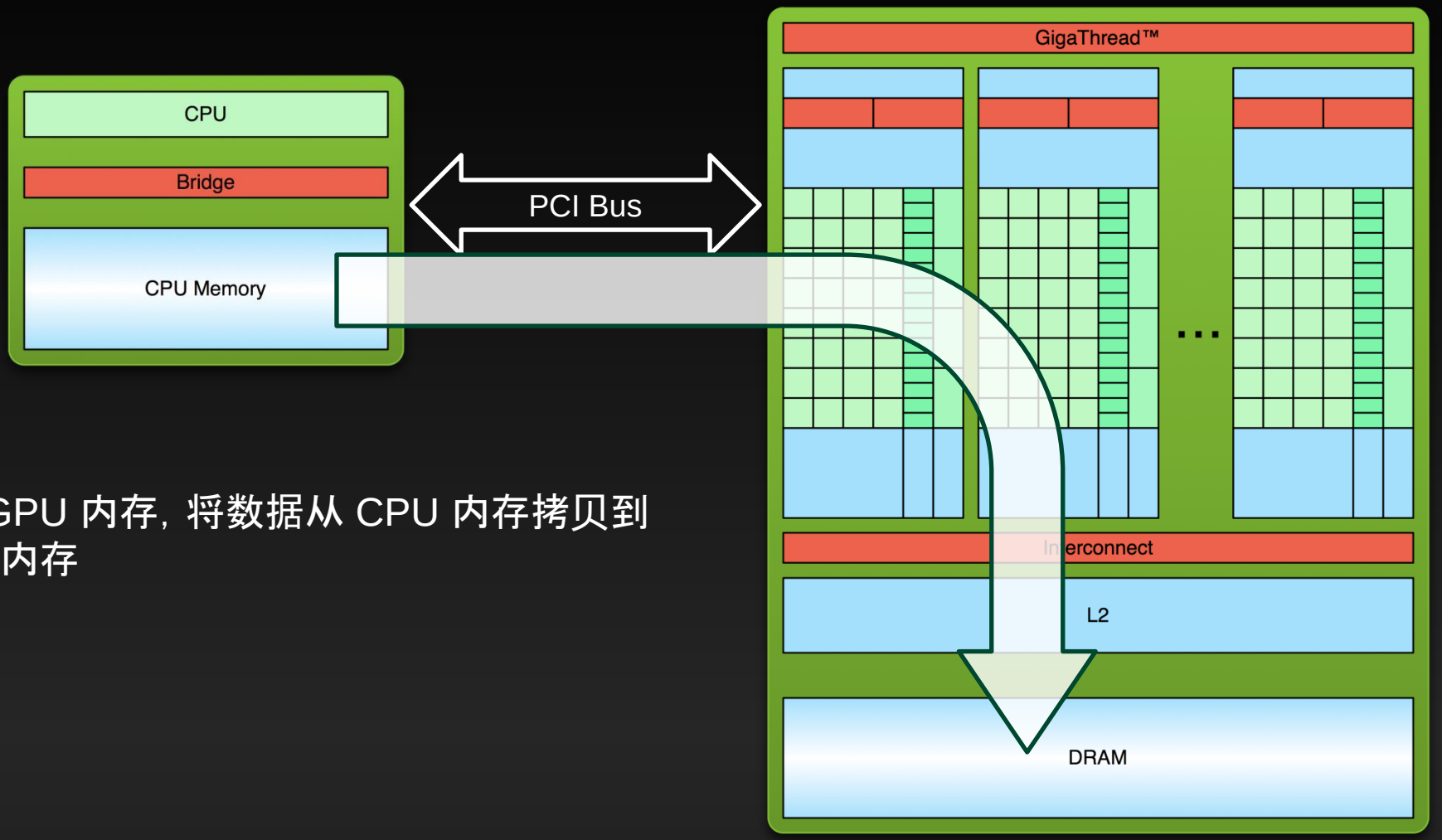
parallel code

serial code



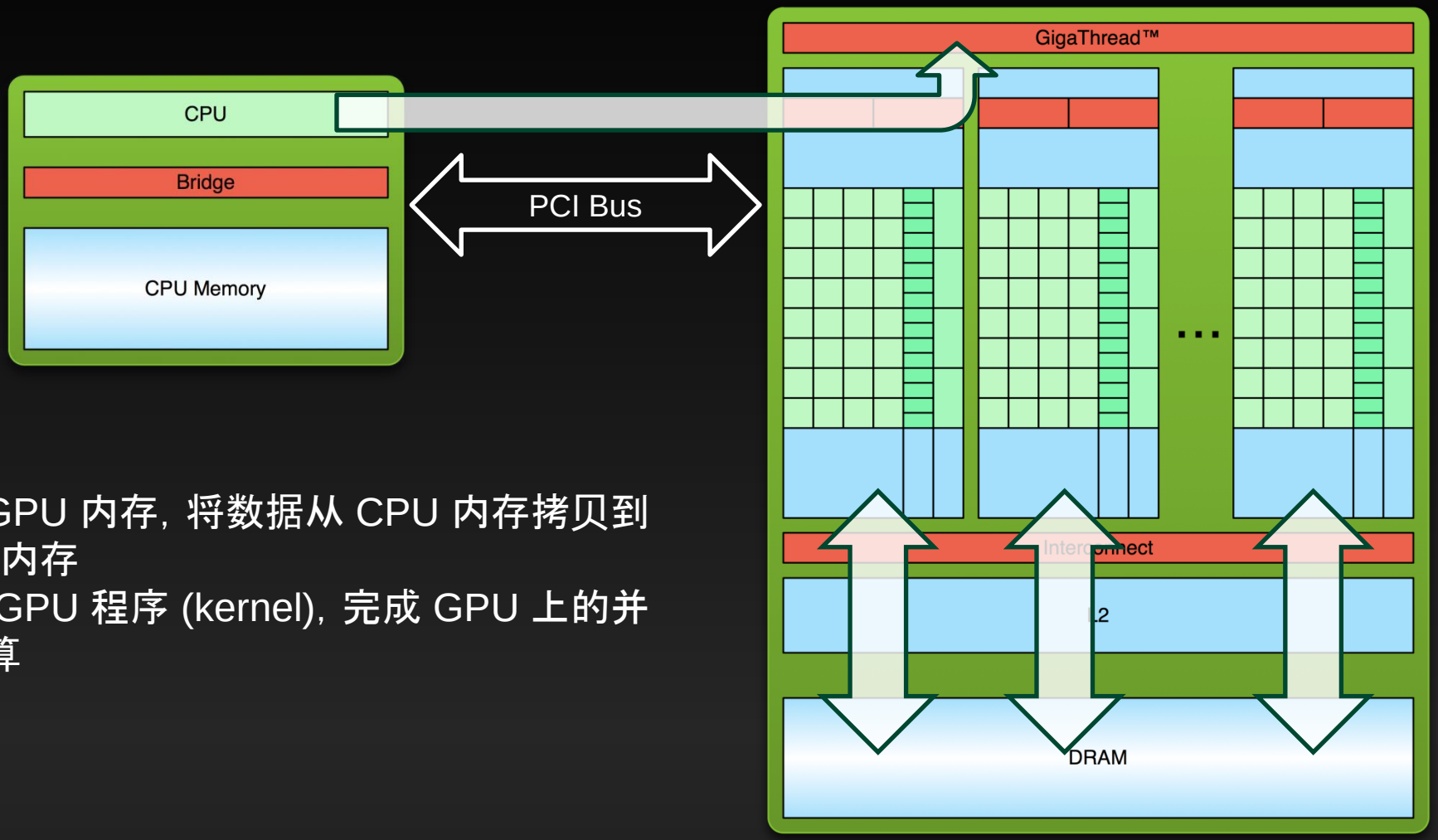


GPU计算的“三部曲”



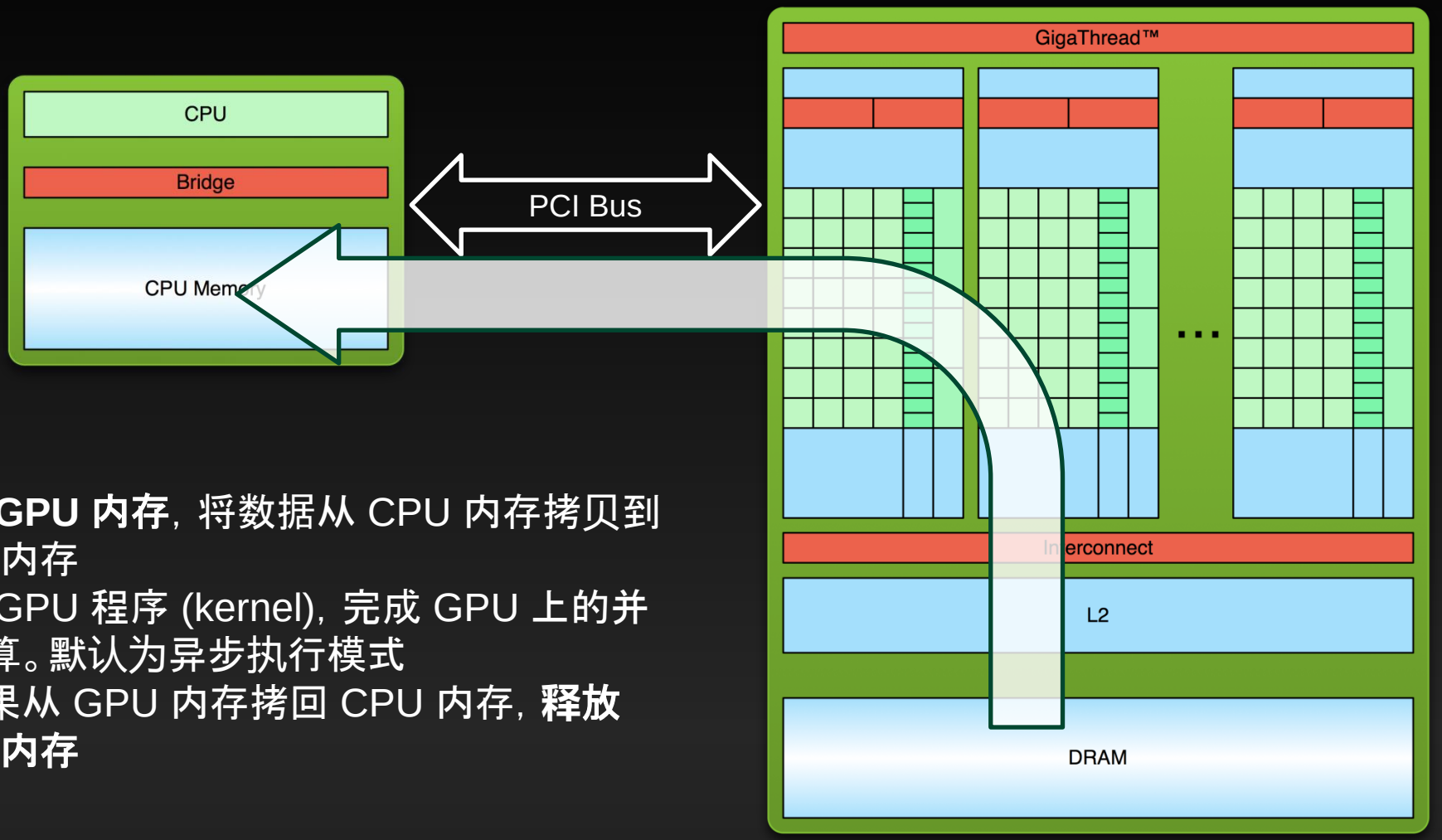
1. 申请GPU 内存，将数据从 CPU 内存拷贝到 GPU 内存

GPU计算的“三部曲”



1. 申请GPU 内存，将数据从 CPU 内存拷贝到 GPU 内存
2. 加载 GPU 程序 (kernel)，完成 GPU 上的并行计算

GPU计算的“三部曲”



1. **申请 GPU 内存**，将数据从 CPU 内存拷贝到 GPU 内存
2. 加载 GPU 程序 (kernel)，完成 GPU 上的并行计算。默认为异步执行模式
3. 将结果从 GPU 内存拷回 CPU 内存，**释放 GPU 内存**

Hello World! (CUDA C 代码)


```
__global__ void mykernel(void) {
      printf("Hello World!\n");
      printf("threadIdx= %d,BlockIDx.x=%d \n"
,threadIdx.x,blockIdx.x);
}

int main(void) {
      mykernel<<<1,2>>>();
      cudaDeviceSynchronize();
      return 0;
}
```

Output:

```
$ nvcc hello.cu
$ a.out
hello world !!!
hello world !!!
threadIdx= 0,
BlockIDx.x=0
threadIdx= 1,
BlockIDx.x=0
$
```

- mykernel 执行的任务很简单，“三部曲”只有“一部曲”

__global__

```
__global__ void mykernel(void) {
        printf("Hello World!\n");
}
```

- CUDA C/C++ 关键字 __global__ 修饰的函数
  - 从主机端调用 (CC3.x, 也可以从设备端调用)
  - 在设备端执行
  - Ampere GPU (CC8.X, A100: CC8.0, A40: CC8.1)
- 其它的函数修饰符
  - __device__：设备端调用，设备端执行
  - __host__ ：主机端调用，主机端执行
  - __device__ 和 __host__ 可同时使用 (同时编译主机端和设备端两个版本)

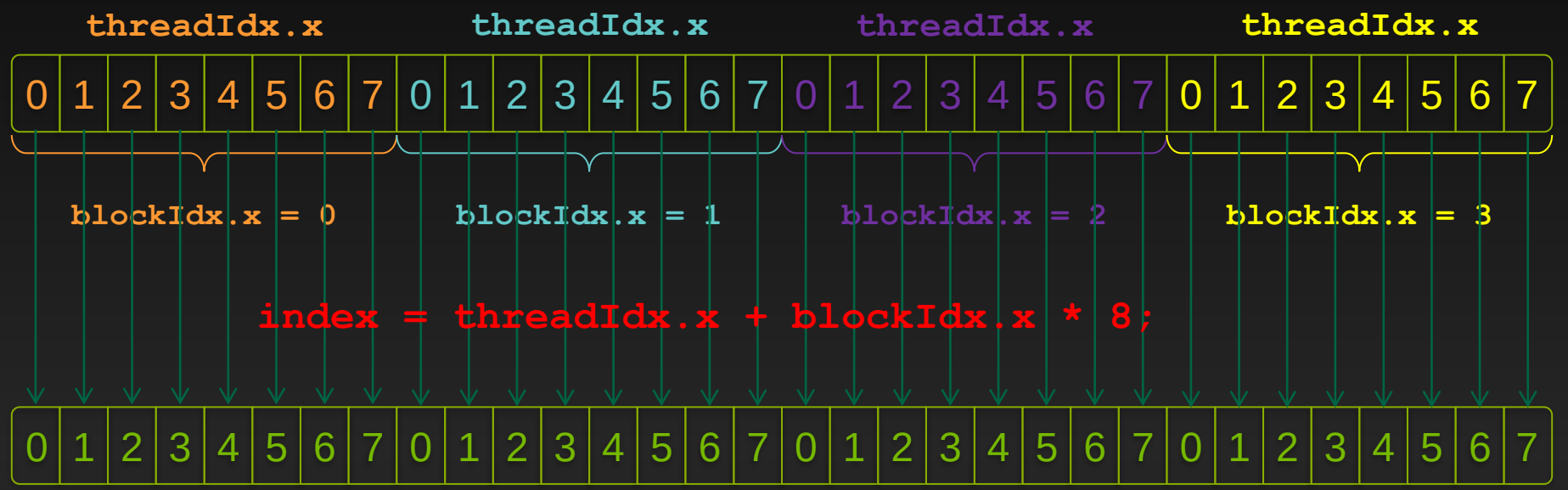
<<<...>>>

```
mykernel<<<1,2>>>();
```

- <<<X1, X2>>> 标记从主机端调用设备端函数以及相应的并行配置
  - 也叫做 “kernel launch” (kernel 启动)
  - X1 和 X2 分别为 kernel 的 grid (栅格) 和 block (线程块) 的设置
- 完成了在主机端调用、设备端执行一个 __global__ 函数过程

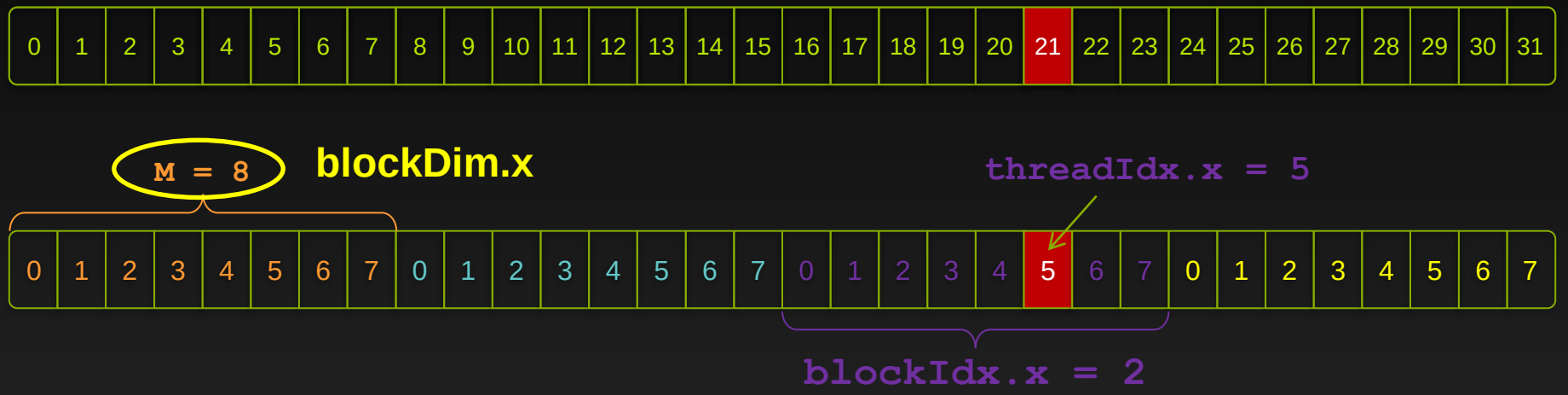
多个 blocks 和 threads 时数据的索引

- `blockIdx.x` 和 `threadIdx.x`
  - 考虑有4个 blocks、每个有 8 threads→线程与待处理数据的索引关系



数组索引: 实例

- 哪个线程块中的线程会计算这个红色的数据?




```
int index = threadIdx.x + blockIdx.x * 8;
          =       5     +      2       * 8;
          = 21;
```

矢量点乘：同时使用多线程块和多线程

- 除 `blockIdx.x` 和 `threadIdx.x` 外，还要采用 `blockDim.x` 确定线程与待处理数据之间的对应关系

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

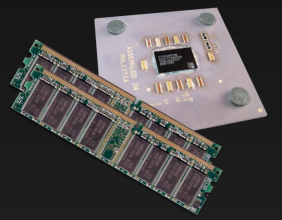
同时使用多线程块和多线程：完整 kernel


```
__global__ void mul(int *a, int *b, int *c) {
            int index = threadIdx.x + blockIdx.x * blockDim.x;
            c[index] = a[index] * b[index];
      }
```

- mul() 在设备端执行，因此 a，b 和 c 必须指向设备内存
- 因此，事先需要在设备端分配内存

内存管理

- 主机内存和设备内存位于不同的物理空间
  - ***设备*** 指针只能指向 ***设备*** 内存
    - 可以在设备与主机之间传递
    - 不能在***主机***端引用 (不能取地址里的内容)
  - ***主机*** 指针只能指向 ***主机*** 内存
    - 可以在主机与设备之间传递
    - 不能在***设备***端引用 (不能取地址里的内容)



- 内存管理的 CUDA API
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - 与主机端的 `malloc()`, `free()`, `memcpy()` 类似

同时使用多线程块和多线程：完整 main


```
#define N (256*256)
#define THREADS_PER_BLOCK 256
#define BLOCKS_NUM (N / THREADS_PER_BLOCK )
int main(void) {
    int *a, *b, *c, sum=0;    // host copies of a, b, sub_sum
    int *d_a, *d_b, *d_c;     // device copies of a, b, sub_sum
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, sub_sum
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc for host copies of a, b, sub_sum and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

同时使用多线程块和多线程：完整 main

Step 1: 申请 GPU 内存，CPU 到 GPU 数据拷贝

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
```

Step 2: 启动 kernel，完成设备端代码的计算

```
// Launch mul() kernel
mul<<< BLOCKS_NUM,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
```

Step 3: GPU 到 CPU 的数据拷贝，释放 GPU 内存

```
// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Reduce on Host
for(int i=0; i< N; i++) sum += c[i];

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c); return 0;}
```

小结 (1 of 3)

- 主机和设备
  - *主机 Host*　　CPU & 内存
  - *设备 Device*　　GPU & 内存

- `__global__ __device__ __host__`
  - `__global__`: 主机端调用、设备端执行
  - `__device__`: 设备端调用、设备端执行
  - `__host__`: 主机端调用、主机端执行
  - `__host__ __device__`：同时在设备端和主机端生成两个版本

小结 (2 of 3)

- 编写 CUDA 程序的三步曲
  - 在设备端申请内存，并将数据从主机内存拷贝到设备内存
  - 调用 kernel，完成设备端的并发计算
  - 将计算结果从设备内存拷贝到主机内存，并释放内存

- 线程的两层组织结构
  - grid：block 的集合，由 blockIdx 标识。不同 block 之间不能通信
  - block: thread 的集合，由 threadIdx 标识
  - 同一个 block 内的线程可通过 shared memory 通信
  - 由 __sycnthreads() 同步

小结 (3 of 3)

- 多个线程和多个线程块协同
  - 利用内建变量 blockDim.x，实现线程与待处理数据的索引关系
  - 归约成总和在主机端完成

- 内建变量 (built-in)
  - threadIdx : 线程 ID
  - blockIdx : 线程块 ID
  - blockDim : block的维度
  - gridDim : grid 的维度
  - 这四个内建变量都是 dim3 型变量 (threadIdx.x; threadIdx.y; threadIdx.z)

两步实现OPENACC加速

Step 1:

```
!$acc data copy(util1,util2,util3) copyin(ip,scp2,scp2i)
  !$acc parallel loop
  …
  !$acc end parallel
!$acc end data
```

Step 2:

```
nvfortran -fast -Minfo -acc file.f
nvc/nvc++ -fast -Minfo -acc file.c
```

注：nvfortran和nvc/nvc++均来自nvidia HPC SDK
如果是PGI编译器，请使用pgfortran和pgcc/pgc++

OpenACC DIRECTIVES EXAMPLE

```
!$acc data copy(A,Anew)
 iter=0
 do while ( err > tol .and. iter < iter_max )

  iter = iter +1
  err=0._fp_kind

!$acc kernels
   do j=1,m
    do i=1,n
     Anew(i,j) = .25_fp_kind *( A(i+1,j  ) + A(i-1,j  ) &
                               +A(i  ,j-1) + A(i  ,j+1))
     err = max( err, Anew(i,j)-A(i,j))
    end do
   end do
!$acc end kernels
   IF(mod(iter,100)==0 .or. iter == 1)    print *, iter, err
   A= Anew

 end do
!$acc end data
```

Copy arrays into GPU memory within data region

Parallelize code inside region

Close off parallel region

Close off data region, copy data back

OpenACC kernels Directive

The kernels directive identifies a region that may contain *loops* that the compiler can turn into parallel *kernels*.

`kernels` Usage:

C: `#pragma acc kernels [clause]`

Fortran: `!$acc kernels [clause]`

```
#pragma acc kernels
{
    for(int i=0; i<N; i++)
    {
        x[i] = 1.0;
    }

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

kernel 1

kernel 2

The compiler identifies 2 parallel loops and generates 2 kernels.

OpenACC parallel loop Directive

`parallel` - Programmer identifies a block of code containing parallelism. Compiler generates a *kernel*.

`loop` - Programmer identifies a loop that can be parallelized within the kernel.

NOTE: parallel & loop are often placed together

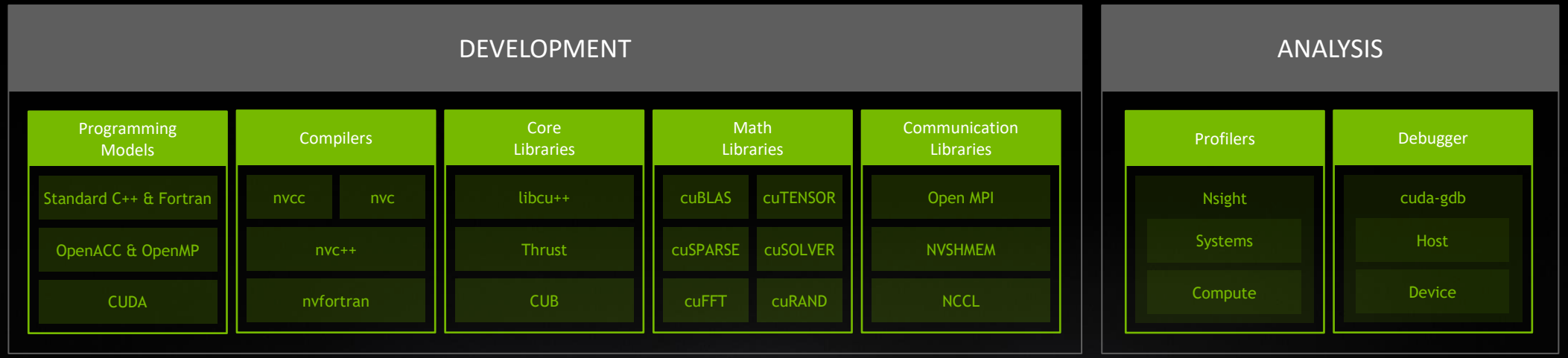
```
#pragma acc parallel loop
for(int i=0; i<N; i++)
{
  x[i] = 1;
  y[i] = 1;
}
```

Generates a Parallel Kernel

ANNOUNCING: THE NVIDIA HPC SDK

Available at developer.nvidia.com/hpc-sdk

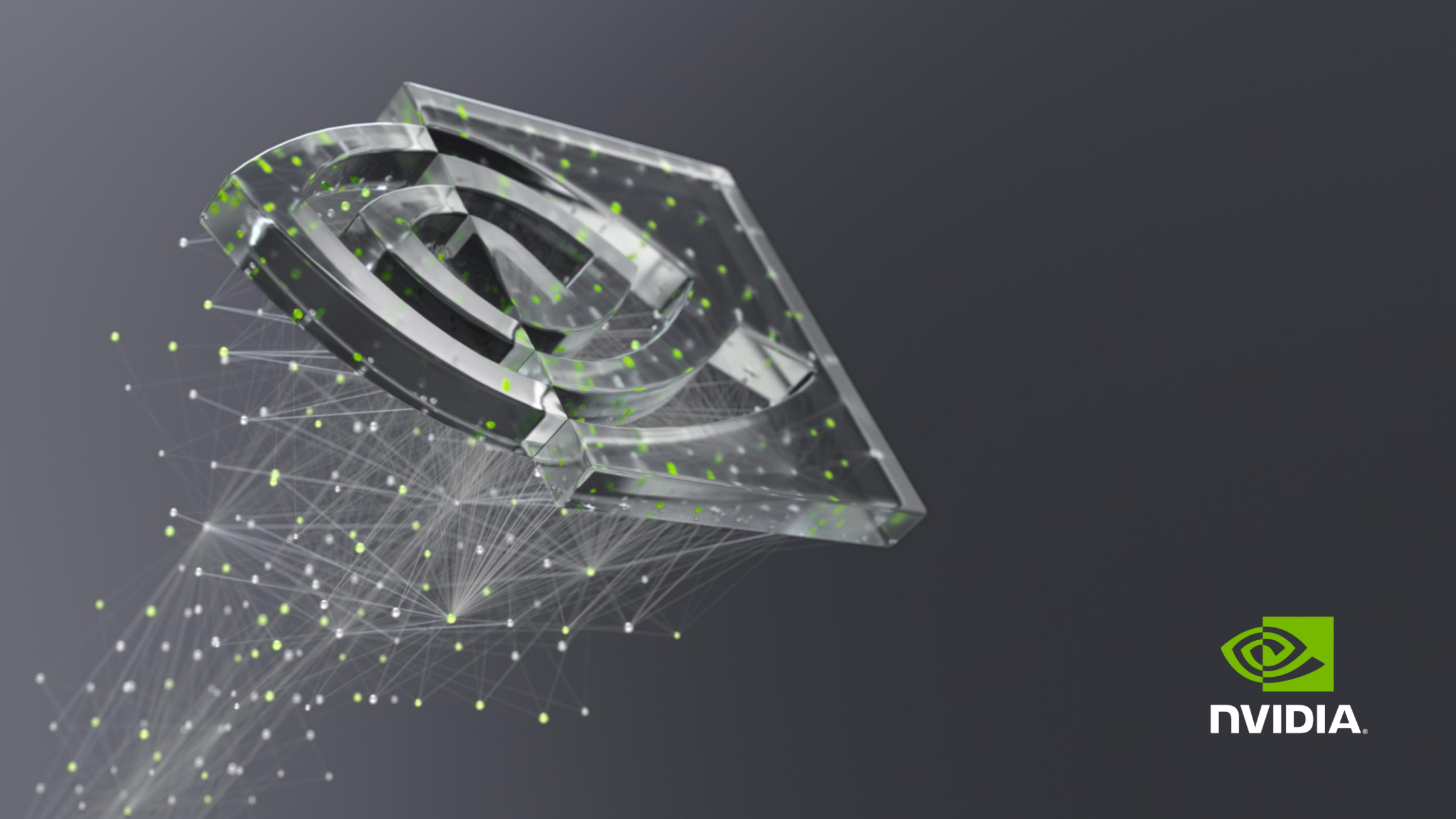
NVIDIA HPC SDK



Develop for the NVIDIA HPC Platform: GPU, CPU and Interconnect
HPC Libraries | GPU Accelerated C++ and Fortran | Directives | CUDA
7-8 Releases Per Year | Freely Available

参考资料

- An introduction to CUDA:
  - https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/
- Another introduction to CUDA:
  - https://devblogs.nvidia.com/even-easier-introduction-cuda/
- CUDA Programming Guide:
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- HPC SDK Documentation:
  - https://docs.nvidia.com/hpc-sdk/index.html


NVIDIA

Stream流

- cudaStream_t stream //定义流
- cudaStreamCreate(&stream) //
- cudaStreamDestroy(stream)//销毁流 ,不是必须，可省略

- cudaStream_t copy_stream; //数据流
- cudaStream_t compute_stream; //计算流
- cudaMemcpyAsync(x_d, x, n*sizeof(double), cudaMemcpyDefault, copy_stream);
- init_data_kernel<<<ceil(n/256),256,0,compute_stream>>>(n, y_d);
- cudaDeviceSynchronize(); //等待所有stream计算完成