CUDA编程基础 – 利用CUDA加速图像处理

NVIDIA企业级开发者社区 何琨

CUDA并行计算基础

AGENDA

- GPU硬件平台
- CUDA 安装
- CUDA线程层次
- CUDA线程索引
- CUDA线程分配
- 图像处理
- 编程实例：Sobel边缘检测

What is CUDA?

- CUDA
  - Compute Unified Device Architecture

- CUDA C/C++
  - 基于C/C++的编程方法
  - 支持异构编程的扩展方法
  - 简单明了的APIs，能够轻松的管理存储系统

- CUDA支持的编程语言：
  - C/C++/Python/Fortran/Java/.......

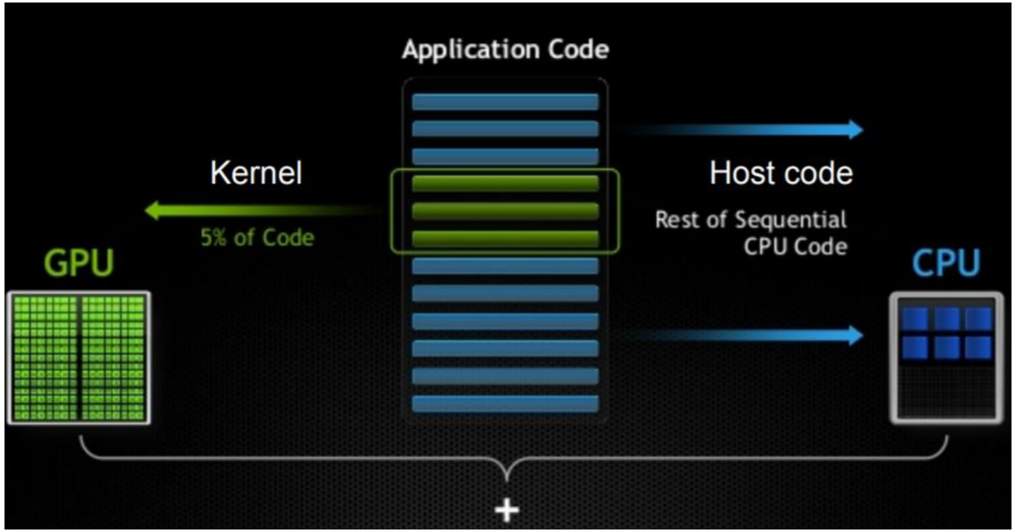
NVIDIA 开发者

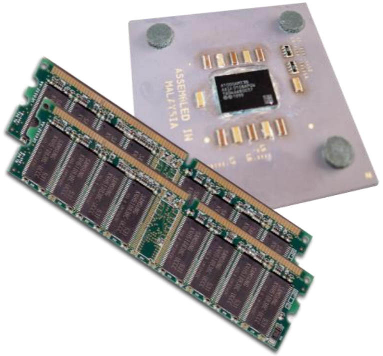
15

500k

异构计算

- 术语:
  - *Host*　CPU和内存(host memory)
  - *Device*　GPU和显存(device memory)



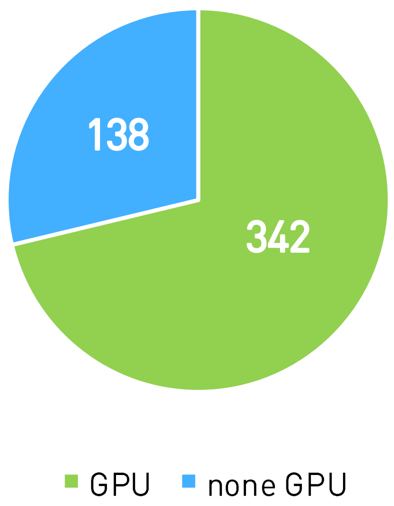


Host

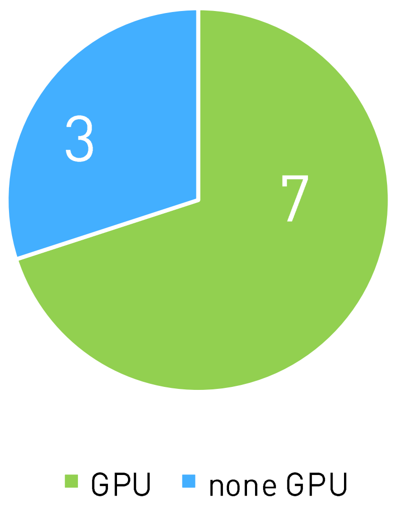
Device

异构计算

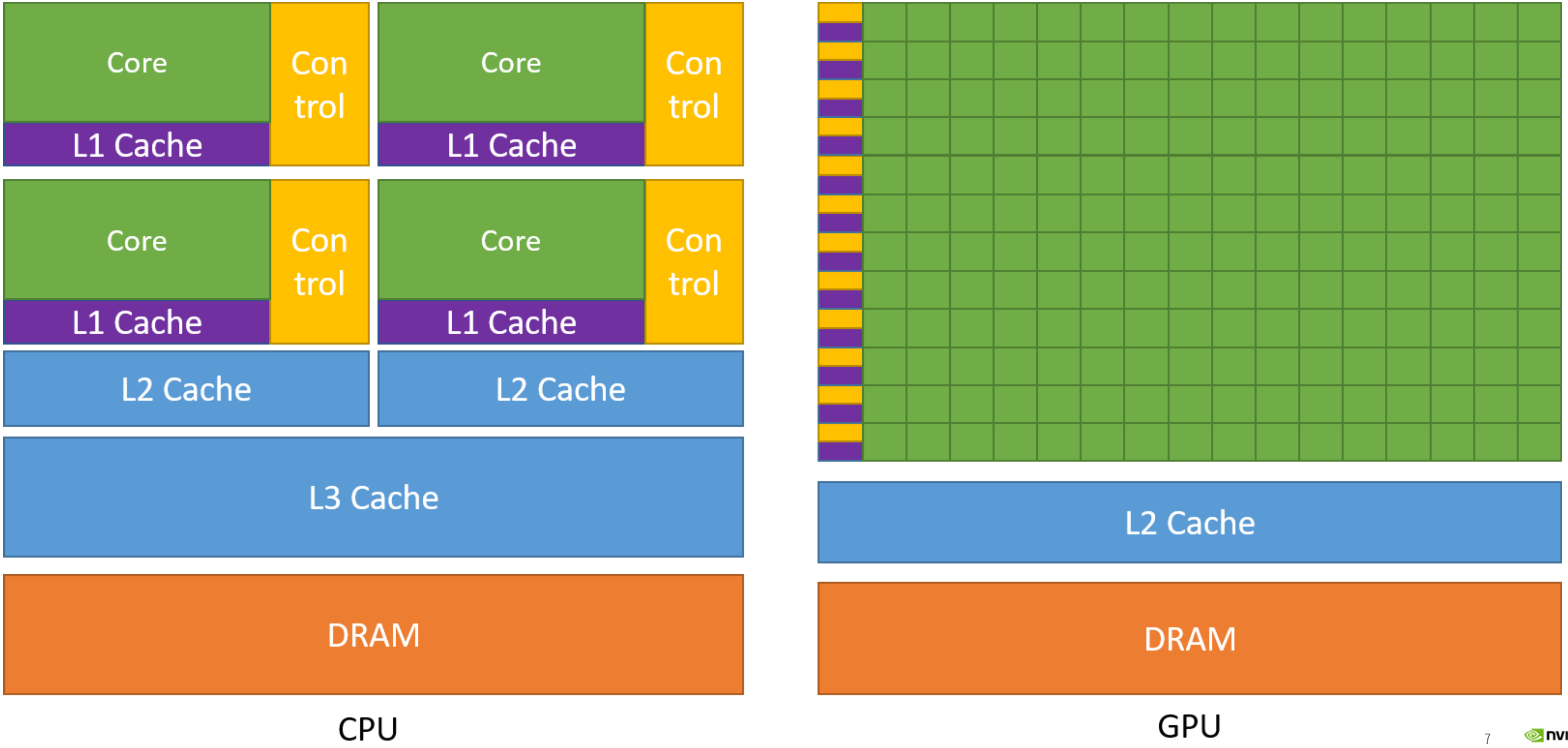
高性能计算大会ISCTOP 500



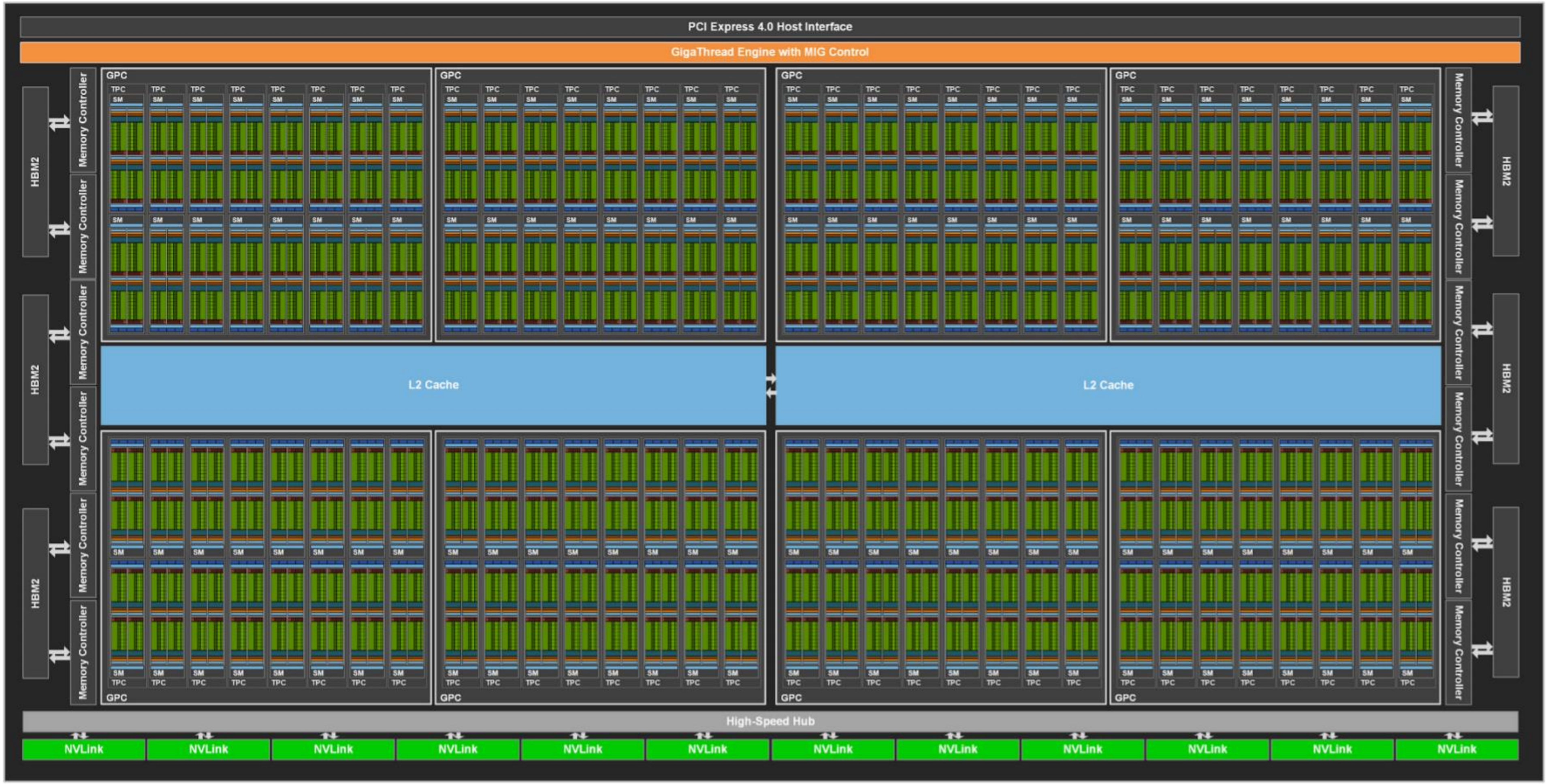
高性能计算大会ISC TOP10



芯片结构


Core
Con
trol
L1 Cache
Core
Con
trol
L1 Cache
Core
Con
trol
L1 Cache
Core
Con
trol
L1 Cache
L2 Cache
L2 Cache
L3 Cache
DRAM
CPU
L2 Cache
DRAM
GPU

GPU结构---GA100



PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

GPU结构---GA100

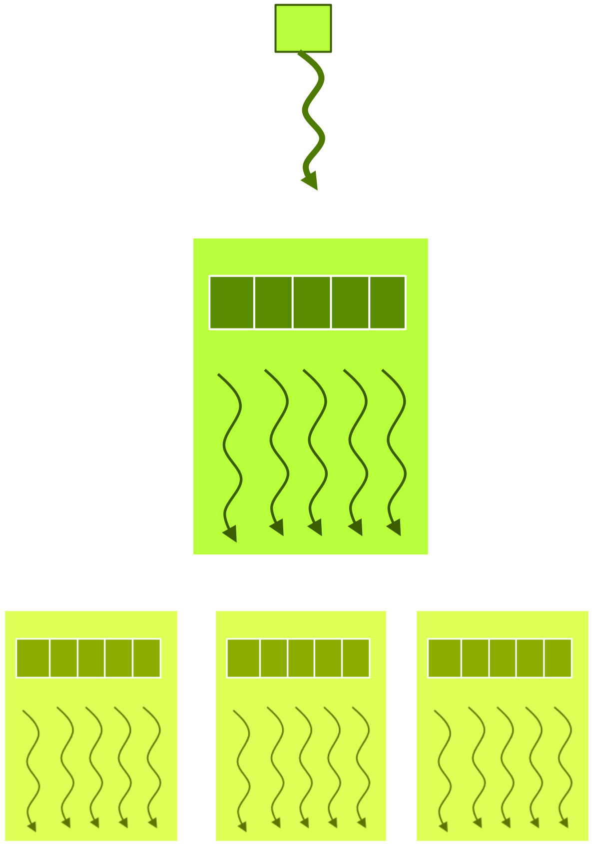
- 8 GPCs, 8 TPCs/GPC, 2 SMs/TPC, 16 SMs/GPC, 128 SMs per full GPU
- 64 FP32 CUDA Cores/SM, 8192 FP32 CUDA Cores per full GPU
- 4 third-generation Tensor Cores/SM, 512 third-generation Tensor Cores per full GPU
- 6 HBM2 stacks, 12 512-bit memory controllers

CUDA安装

- 适用设备:
  - *所有包含NVIDIA GPU的服务器，工作站，个人电脑，嵌入式设备等电子设备*
- 软件安装:
  - Windows：https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html
    只需安装一个.exe的可执行程序
  - Linux：https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html
    按照上面的教程，需要6 / 7 个步骤即可
  - Jetson： https://developer.nvidia.com/embedded/jetpack
    直接利用NVIDIA SDK Manager 或者 SD image进行刷机即可

CUDA线程层次

```
HelloFromGPU <<<?, ?>>>();
```



- Thread: sequential execution unit
  - 所有线程执行相同的核函数
  - 并行执行
- Thread Block: a group of threads
  - 执行在一个Streaming Multiprocessor (SM)
  - 同一个Block中的线程可以协作
- Thread Grid: a collection of thread blocks
  - 一个Grid当中的Block可以在多个SM中执行

线程层次

- 执行设置:

dim3 grid(3,2,1), block(5,3,1)

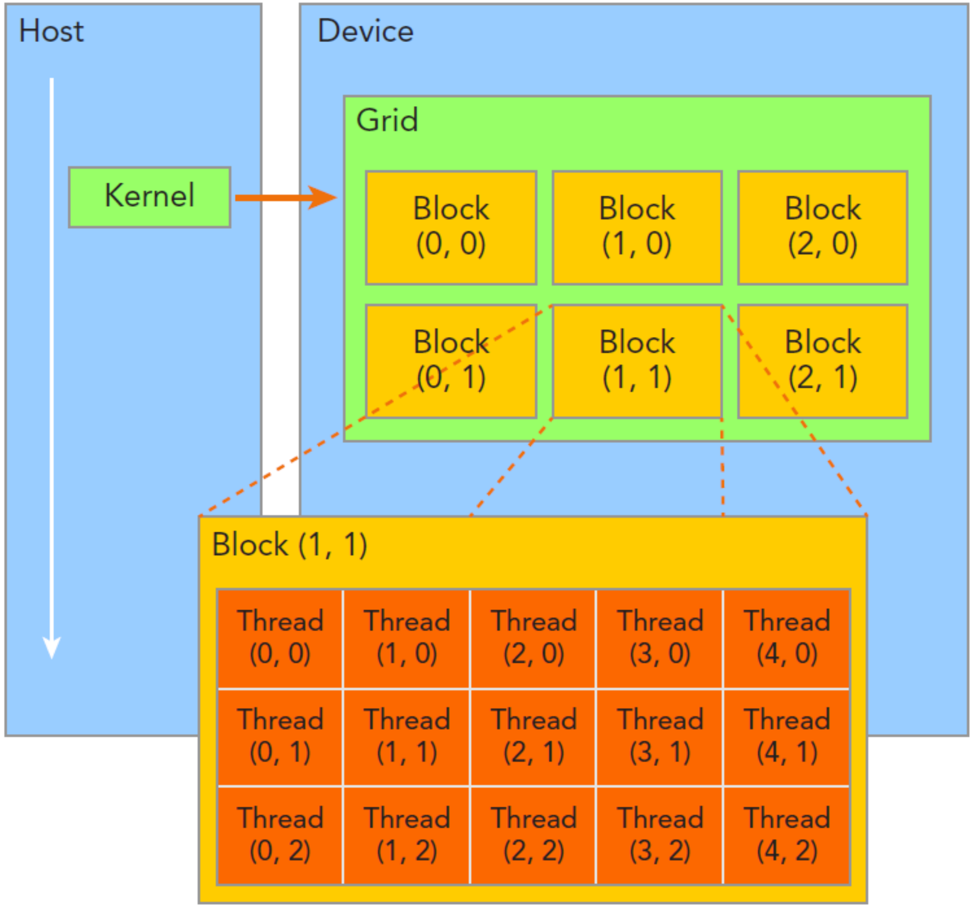
- Built-in variables:
  - threadIdx.[x y z]

    是执行当前kernel函数的线程在block中的索引值
  - blockIdx.[x y z]

    是指执行当前kernel函数的线程所在block，在grid中的索引值
  - blockDim.[x y z]

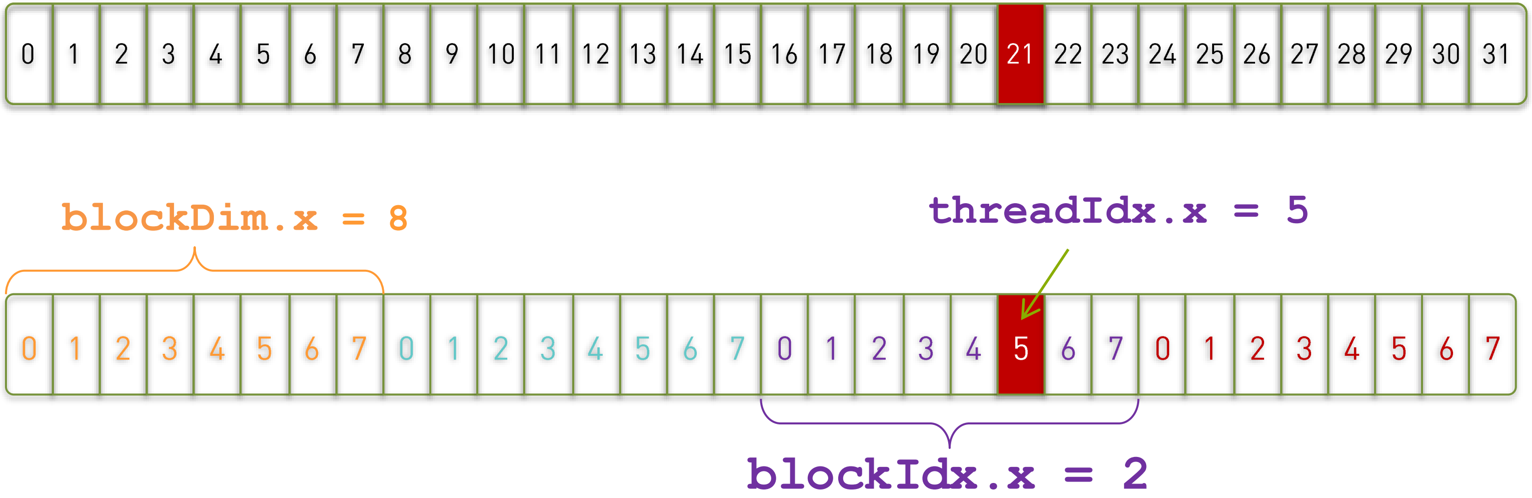
    表示一个grid中包含多少个block
  - gridDim.[x y z]

    表示一个block中包含多少个线程



CUDA的线程索引

- 如何确定线程执行地数据




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

线程层次

- 我们写的程序:

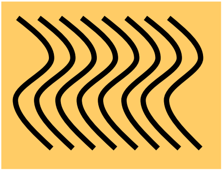
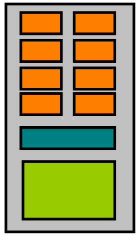
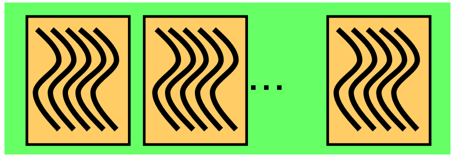
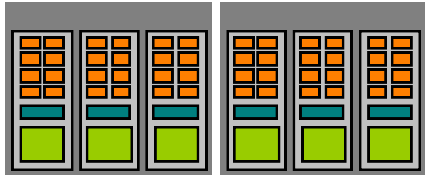
```
__global__ void add( int *a, int *b, int *c ) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
add<<<1,4>>>( a, b, c);
```

- 实际上在设备上运行的样子:

Thread 0	Thread 1
c[0] = a[0] + b[0];	c[1] = a[1] + b[1];

Thread 2	Thread 3
c[2] = a[2] + b[2];	c[3] = a[3] + b[3];

线程层次

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

线程层次

- 硬件调度：
  - Grid： GPU(GPC)级别的调度单位
  - Block(CTA)：SM级别的调度单位
  - Threads/Warp：CUDA core级别的调度单位
- 资源和通信：
  - Grid：共享同样的kernel 和 Context
  - Block(CTA)：同一个SM(Streaming Multiprocessor)，同一个SM(Shared Memory)
  - Threads/Warp: 允许同一个warp中的thread读取其他thread的值

线程层次

- 硬件调度：
  - Grid： GPU(GPC)级别的调度单位
  - Block(CTA)：SM级别的调度单位
  - Threads/Warp： CUDA core级别的调度单位
- 资源和通信：
  - Grid：共享同样的kernel 和 Context
  - Block(CTA)：同一个SM(Streaming Multiprocessor)，同一个SM(Shared Memory)
  - Threads/Warp: 允许同一个warp中的thread读取其他thread的值

NVIDIA

CUDA的线程索引

```
__global__ void add(const double *x, const double
*y, double *z)
{
    const int n = blockDim.x * blockIdx.x + threadIdx.x;
    z[n] = x[n] + y[n];
}
```

每个线程都执行相同的命令

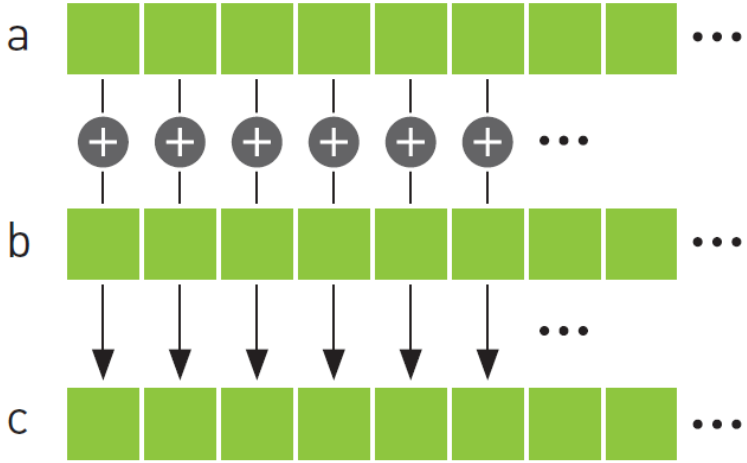
CUDA PROGRAMMING BY EXAMPLE

Case: Vector Add

- Parallelizable problem:
  - c = a + b
  - a, b, c are vectors of length N
- CPU implementation:

```
void main(){
   int size = N * sizeof(int);
   int *a, *b, *c;
   a = (int *)malloc(size);
   b = (int *)malloc(size);
   c = (int *)malloc(size);
   memset(c, 0, size);
   init_rand_f(a, N);
   init_rand_f(b, N);

   vecAdd(N, a, b, c);
}
```




```
void vecAdd (int n, int *a,
             int *b, int *c)
{
   for(int i = 0; i < n; i++)
   {
      c[i] = a[i] + b[i];
   }
}
```

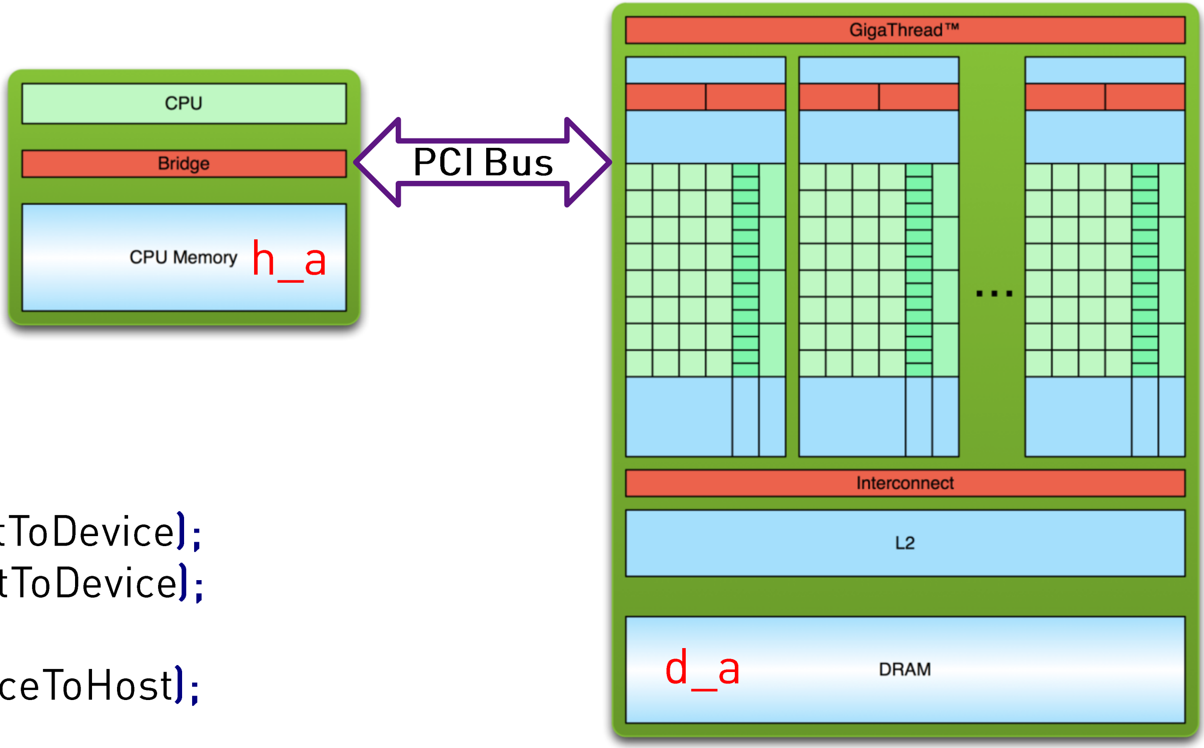
GPU CODE WORKFLOW

Allocate GPU Memories

```
int main(void) {
    size_t size = N * sizeof(int);
    int *h_a, *h_b; int *d_a, *d_b, *d_c;
    h_a = (int *)malloc(size);
    h_b = (int *)malloc(size);
    ...
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
    vectorAdd<<<grid, block>>>(d_a, d_b, d_c, N);
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    free(h_a); free(h_b);
    return 0;}
```



NVIDIA

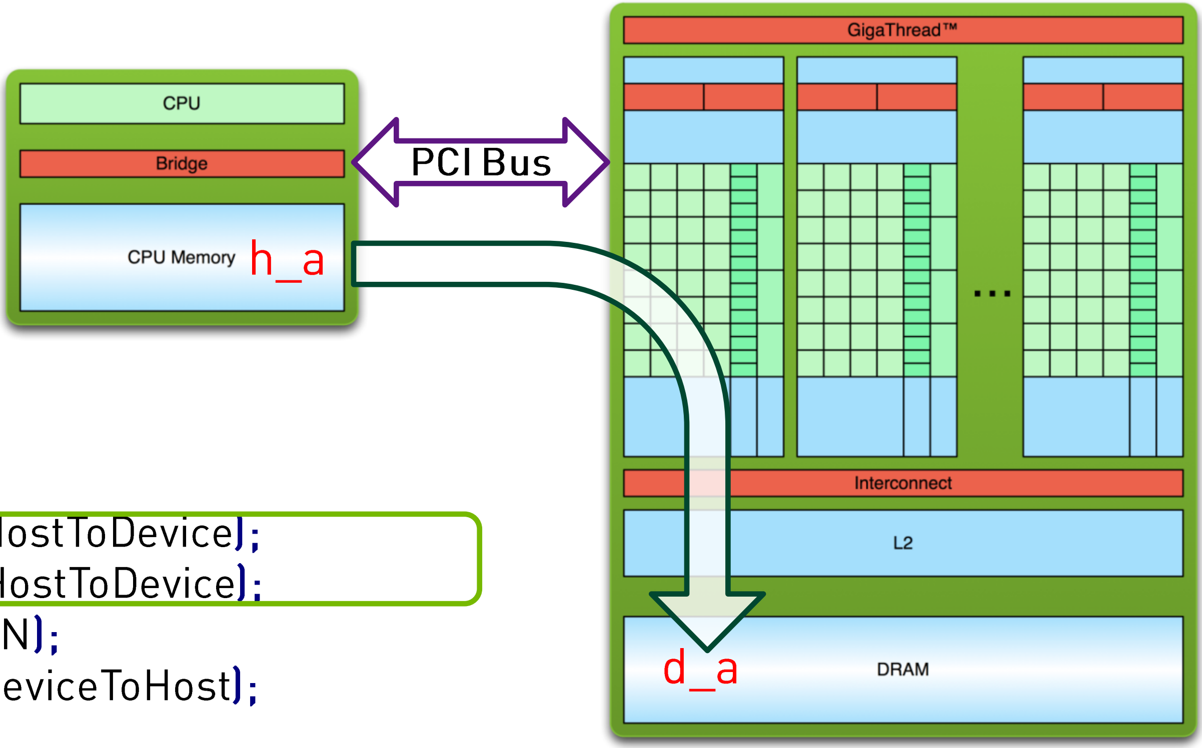
GPU CODE WORKFLOW

Copy data from CPU to GPU

```
int main(void) {
    size_t size = N * sizeof(int);
    int *h_a, *h_b; int *d_a, *d_b, *d_c;
    h_a = (int *)malloc(size);
    h_b = (int *)malloc(size);
    ...
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
    vectorAdd<<<grid, block>>>(d_a, d_b, d_c, N);
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    free(h_a); free(h_b);
    return 0;}
```



GPU CODE WORKFLOW

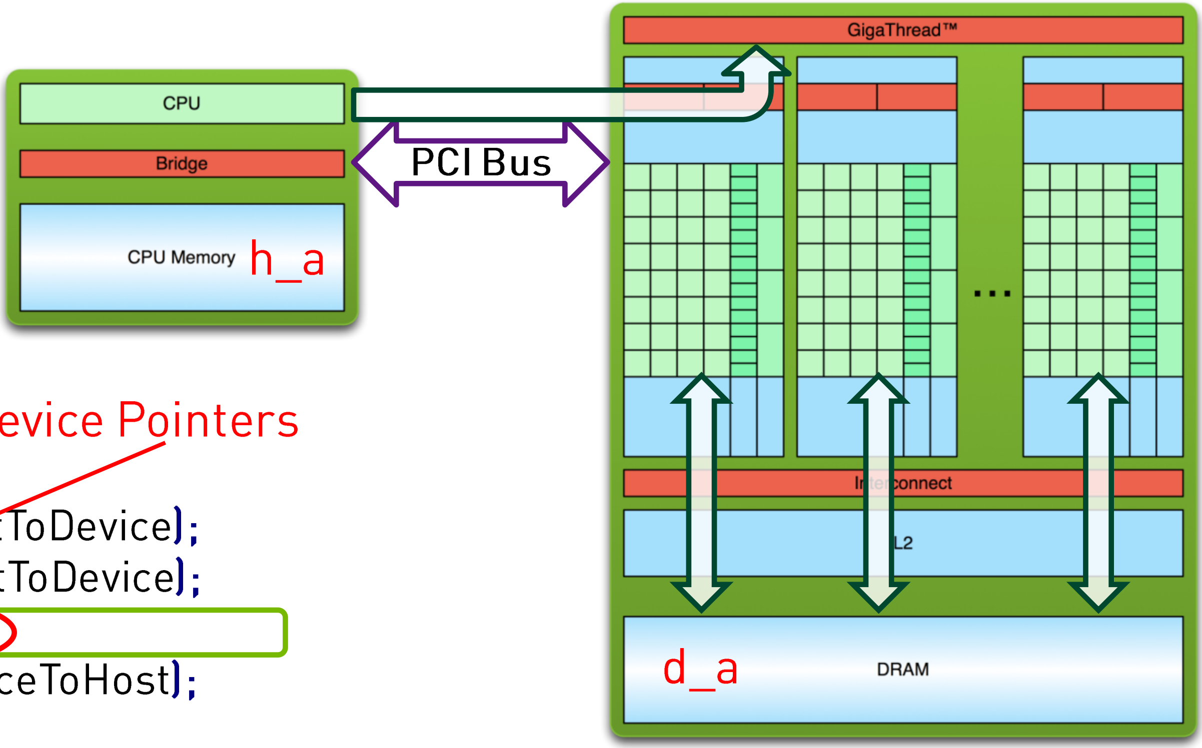
Invoke the CUDA Kernel

```
int main(void) {
    size_t size = N * sizeof(int);
    int *h_a, *h_b; int *d_a, *d_b, *d_c;
    h_a = (int *)malloc(size);
    h_b = (int *)malloc(size);
    ...
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
    vectorAdd<<<grid, block>>>(d_a, d_b, d_c, N);
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    free(h_a); free(h_b);
    return 0;}
```

Device Pointers



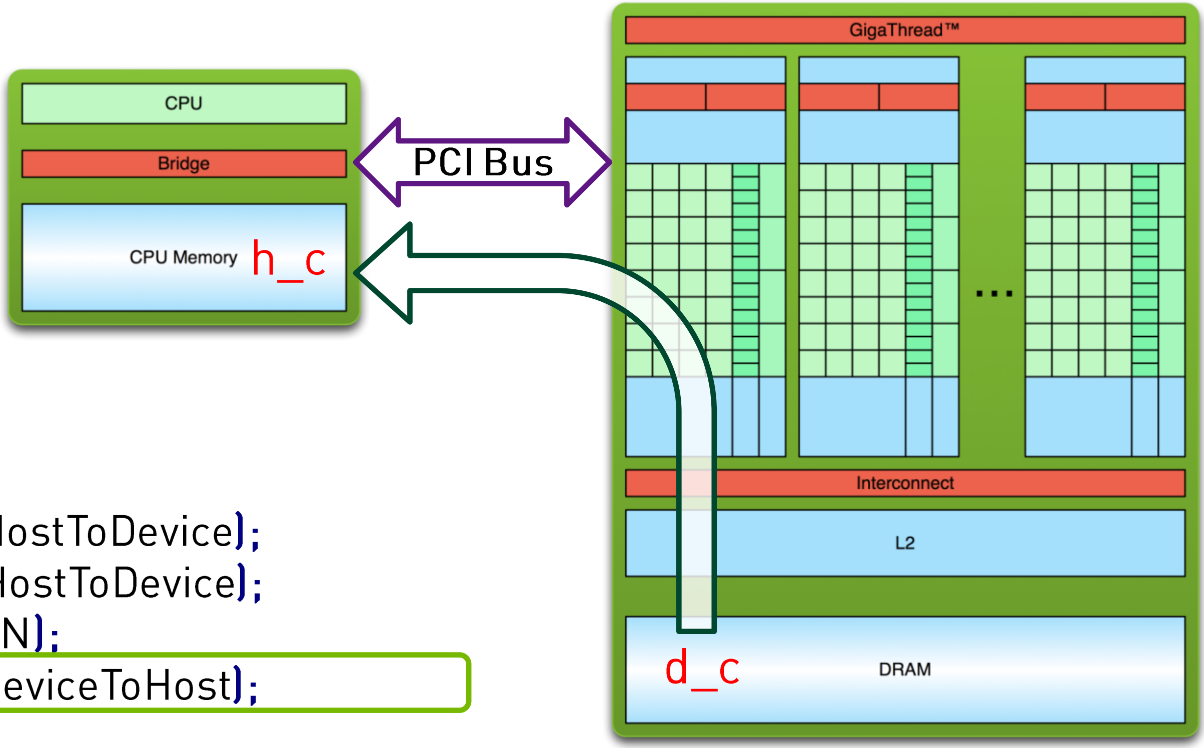
GPU CODE WORKFLOW

Copy result from GPU to CPU

```
int main(void) {
    size_t size = N * sizeof(int);
    int *h_a, *h_b; int *d_a, *d_b, *d_c;
    h_a = (int *)malloc(size);
    h_b = (int *)malloc(size);
    ...
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
    vectorAdd<<<grid, block>>>(d_a, d_b, d_c, N);
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    free(h_a); free(h_b);
    return 0;}
```



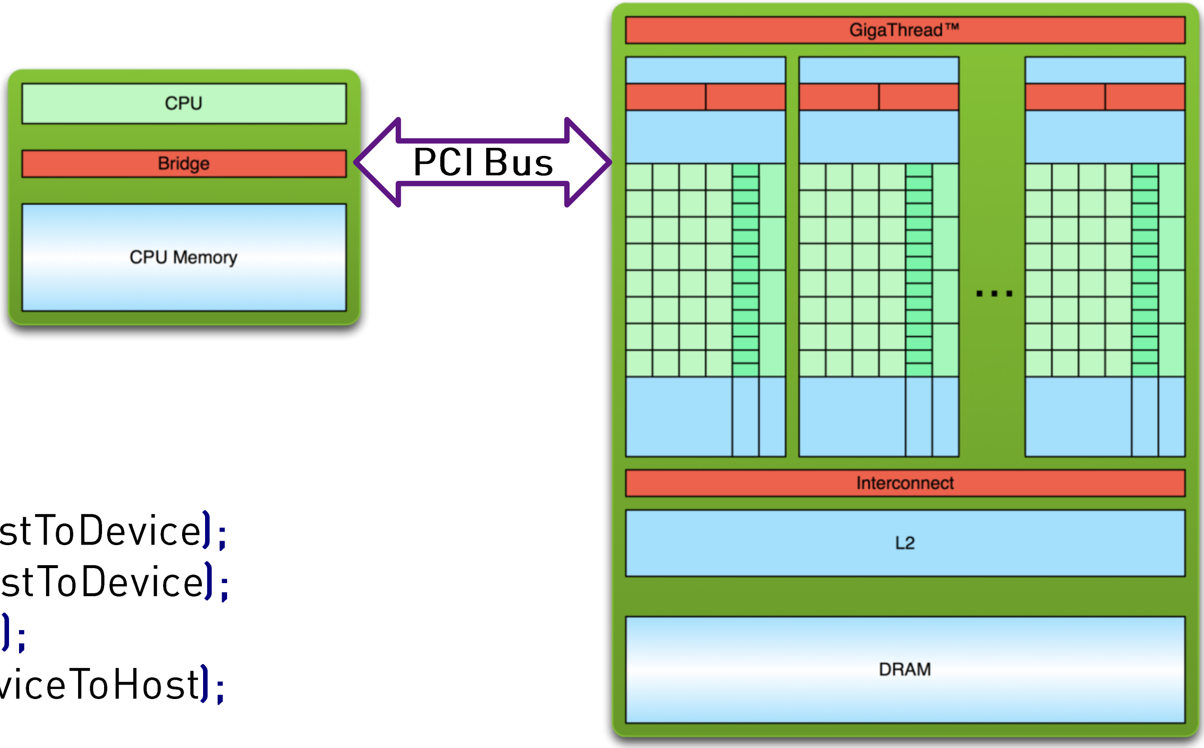
GPU CODE WORKFLOW

Release GPU Memories

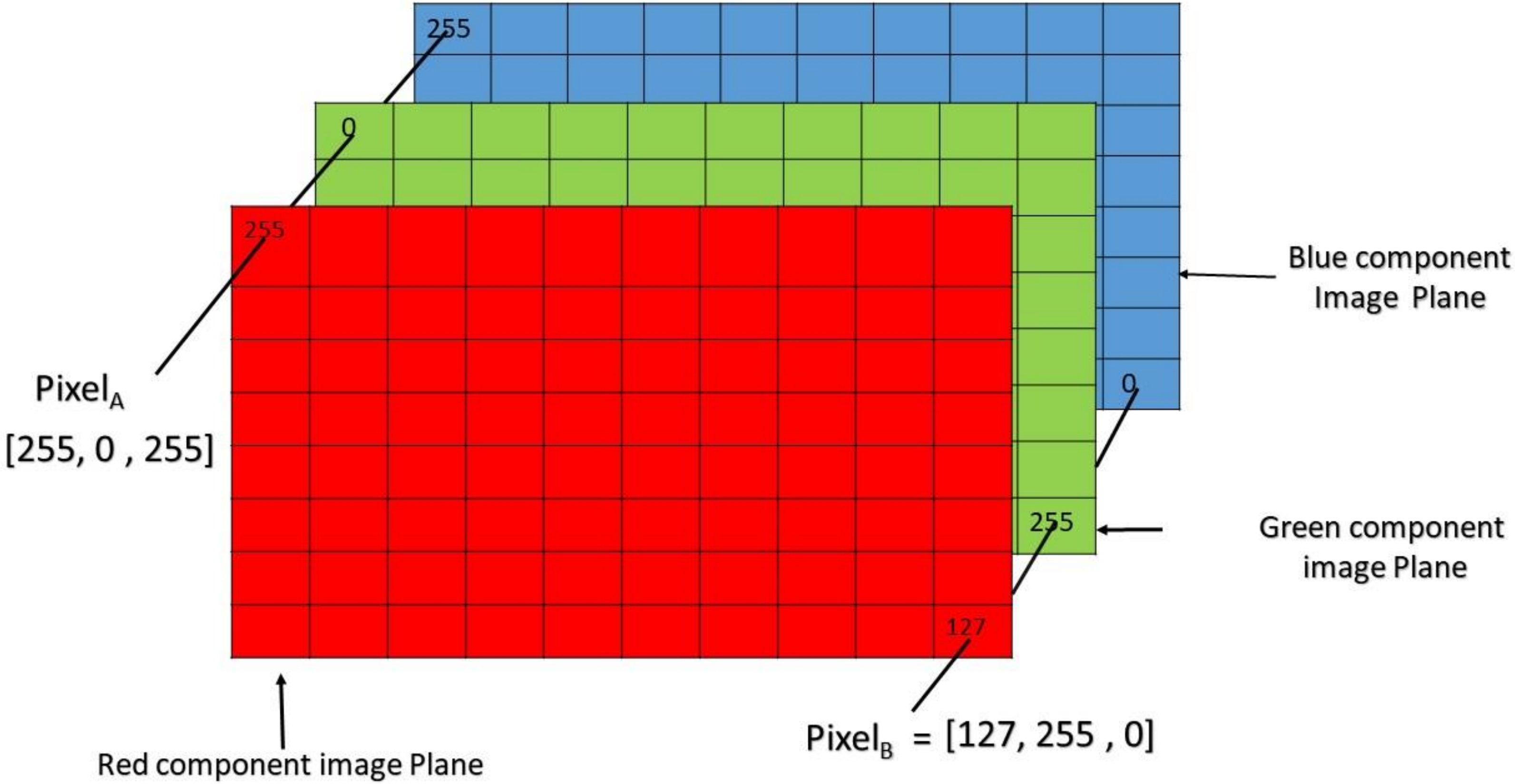
```
int main(void) {
    size_t size = N * sizeof(int);
    int *h_a, *h_b; int *d_a, *d_b, *d_c;
    h_a = (int *)malloc(size);
    h_b = (int *)malloc(size);
    ...
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
    vectorAdd<<<grid, block>>>(d_a, d_b, d_c, N);
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    free(h_a); free(h_b);
    return 0;}
```



图像处理

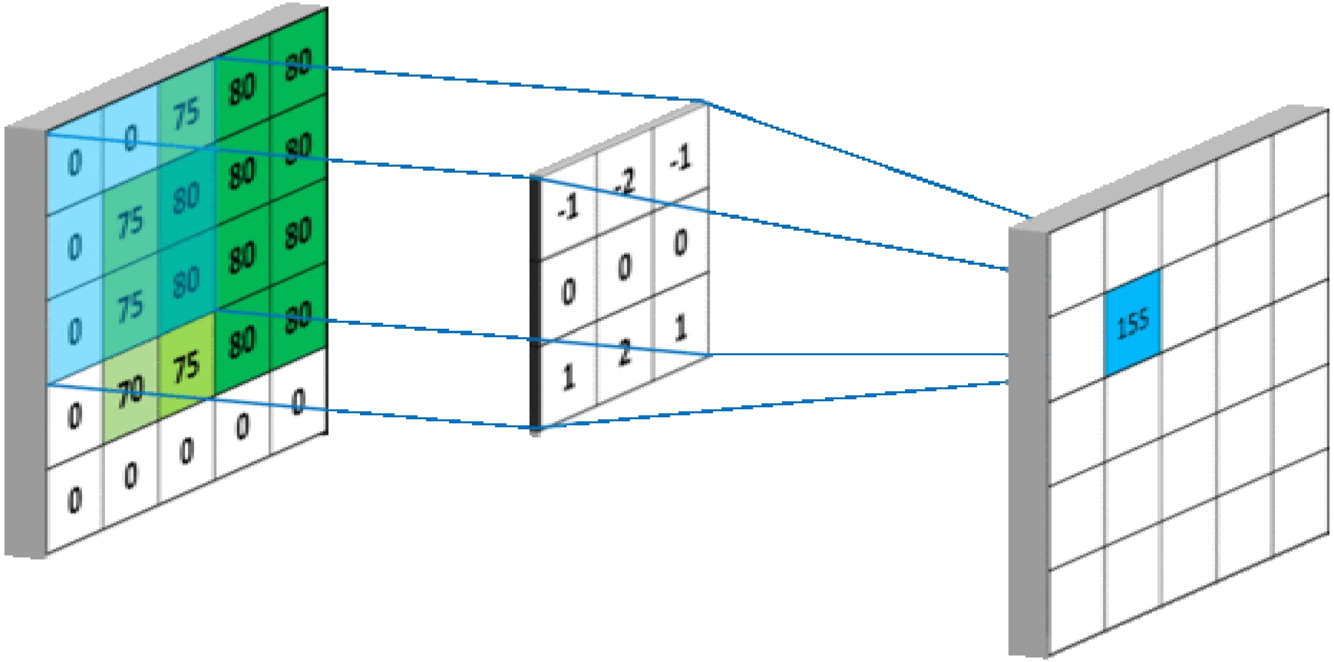


Pixel of an RGB image are formed from the corresponding pixel of the three component images

Sobel 边缘检测

$$\mathbf{G_x} = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G_y} = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Sobel 边缘检测



0
0
75
80
80
0
75
80
80
80
0
75
80
80
80
0
70
75
80
80
0
0
0
0
0
0
0
0
0
-1
-2
-1
0
0
0
1
2
1
155

更多资源：

https://developer.nvidia-china.com

何琨-Ken

北京 密云

https://www.nvidia.cn/developer/community-training/

扫一扫上面的二维码图案，加我微信

NVIDIA