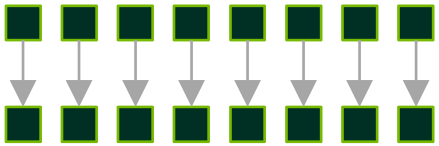
CUDA ON ARM PLATFORM—线程层次

NVIDIA企业级开发者社区 何琨

CUDA并行计算基础

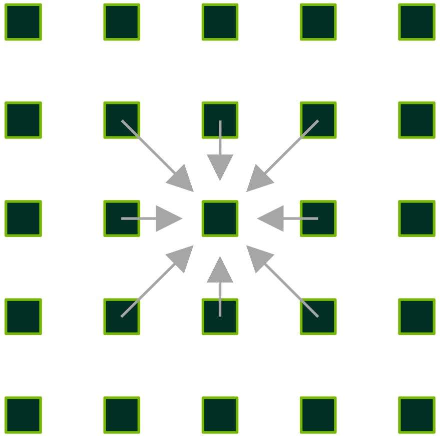
- CUDA显存分配
- CUDA数据传输
- CUDA线程索引
- CUDA线程分配

我们要处理的问题



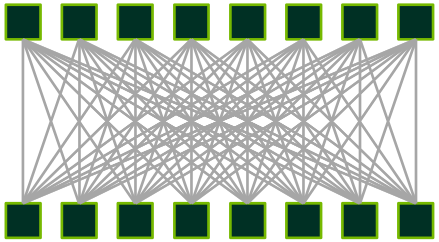
Element-wise

DAXPY



Local

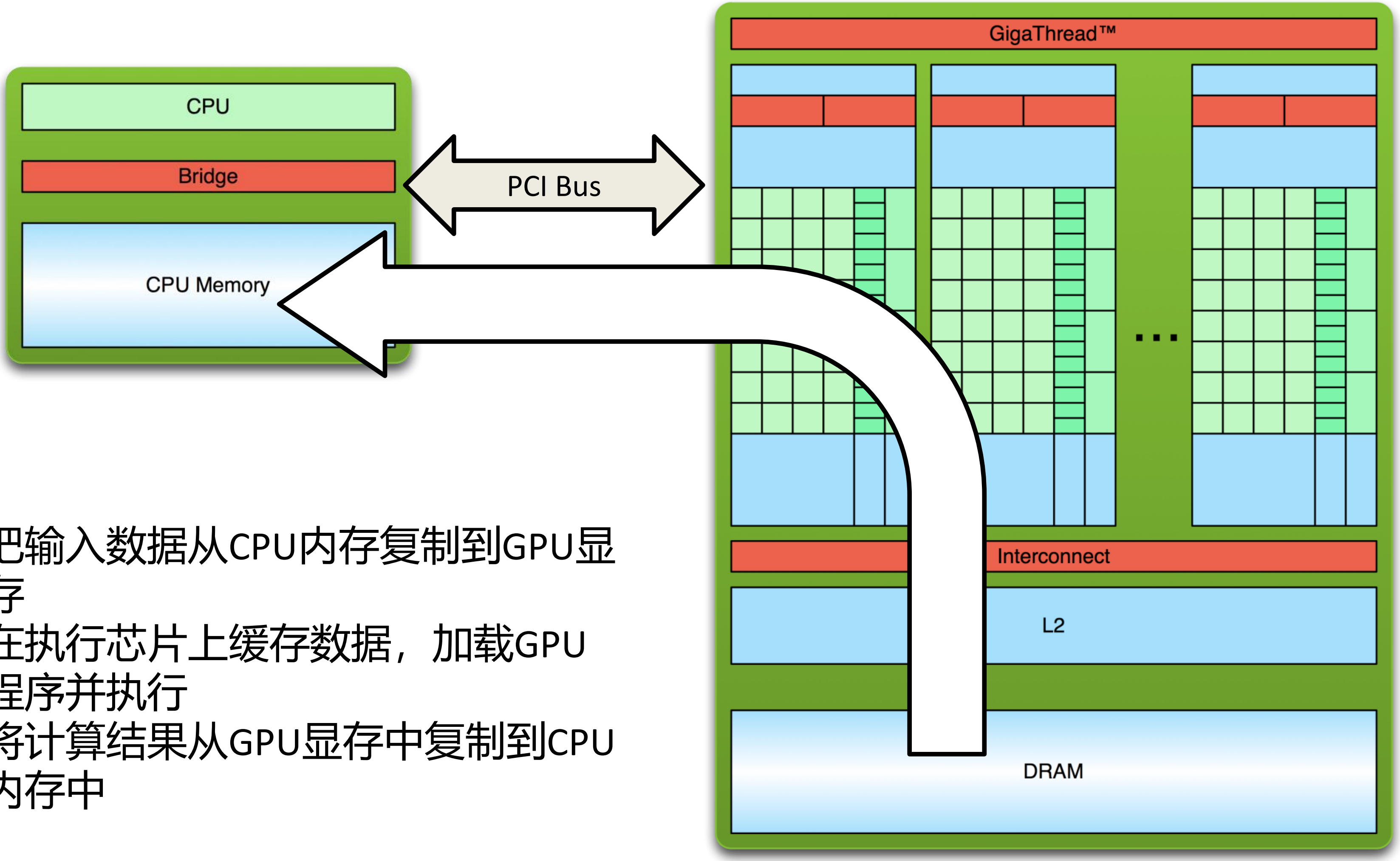
Convolution



All-to-All

Fourier Transform

CUDA程序的编写



1. 把输入数据从CPU内存复制到GPU显存
2. 在执行芯片上缓存数据，加载GPU程序并执行
3. 将计算结果从GPU显存中复制到CPU内存中

MEMORY ALLOCATION

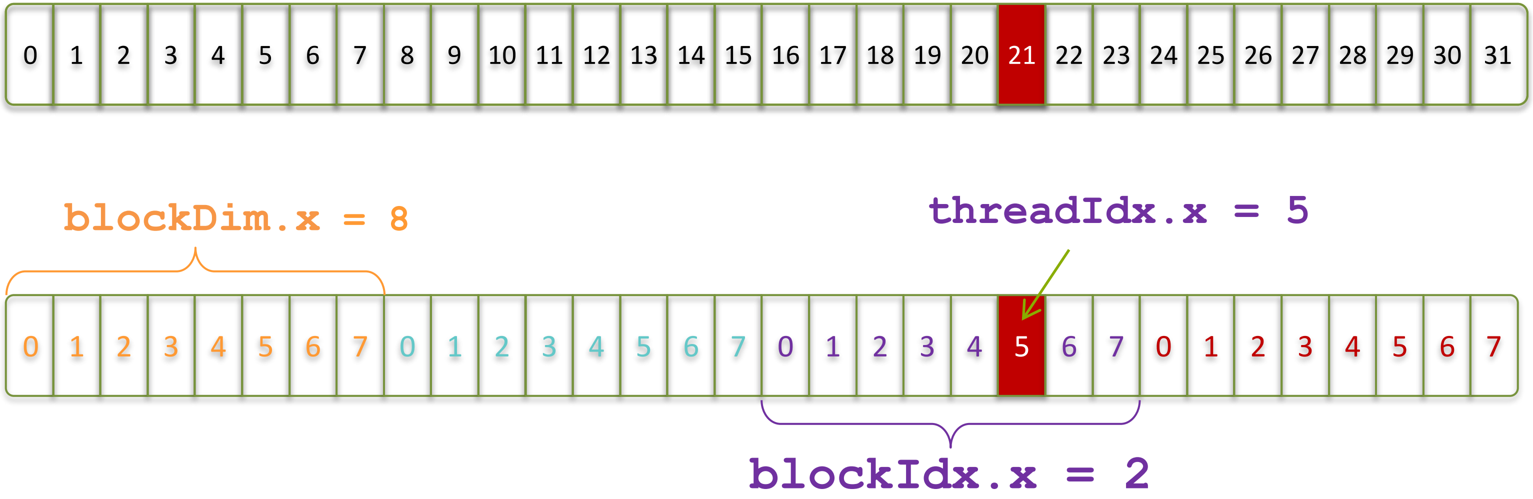
- __host__ __device__ cudaError_t cudaMalloc(void** devPtr, size_t size)
  - **devPtr:**
    - **- Pointer to allocated device memory**
  - **Size:**
    - **- Requested allocation size in bytes**

MEMORY COPY BETWEEN CPU AND GPU

- cudaMemcpy(void ***dst,** const void ***src,** size_t **count,** cudaMemcpyKind **kind)**
  - **dst:** destination memory address
  - **src:** source memory address
  - **count:** size in bytes to copy
  - **kind:** direction of the copy
- cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
  - cudaMemcpyHostToHost

CUDA的线程索引

- 如何确定线程执行地数据

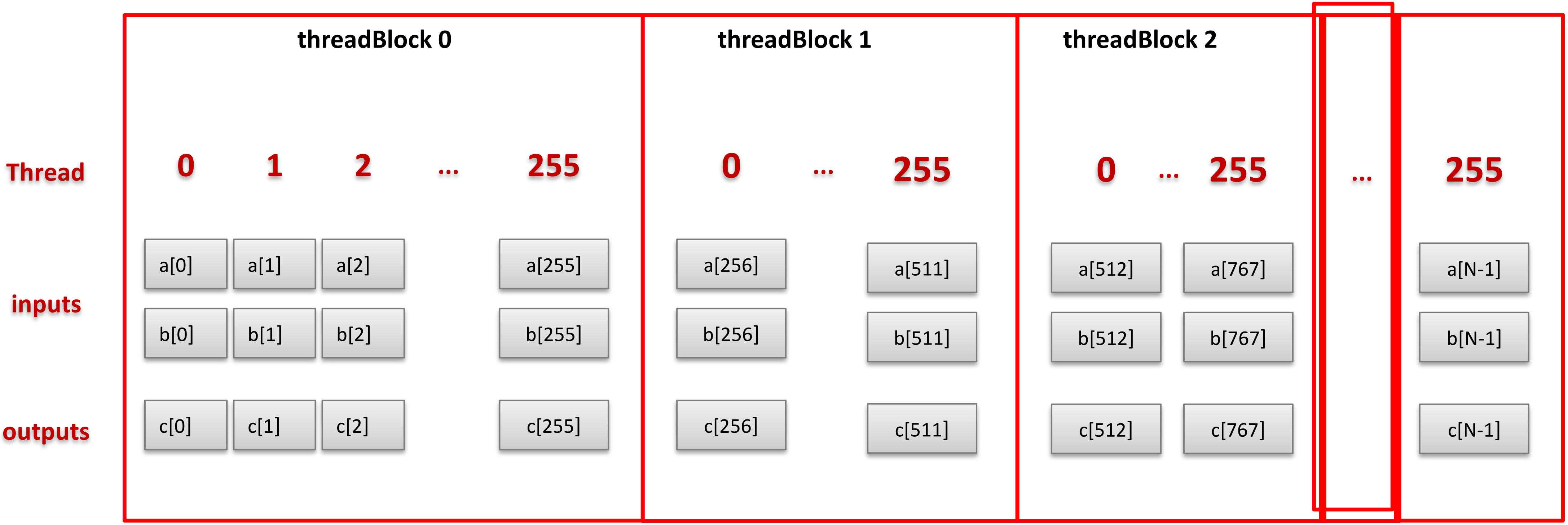



```
int index = threadIdx.x + blockIdx.x * blockDim.x;
          =         5   +        2   * 8;
          = 21;
```

PARALLELIZATION OF VECTORADD

Thread	0	1	2	...	255	256	...	511	512	...	767	...	N-1
inputs	a[0]	a[1]	a[2]		a[255]	a[256]		a[511]	a[512]		a[767]		a[N-1]
	b[0]	b[1]	b[2]		b[255]	b[256]		b[511]	b[512]		b[767]		b[N-1]
outputs	c[0]	c[1]	c[2]		c[255]	c[256]		c[511]	c[512]		c[767]		c[N-1]

PARALLELIZATION OF VECTORADD



work index i = threadIdx.x + blockIdx.x * blockDim.x;

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

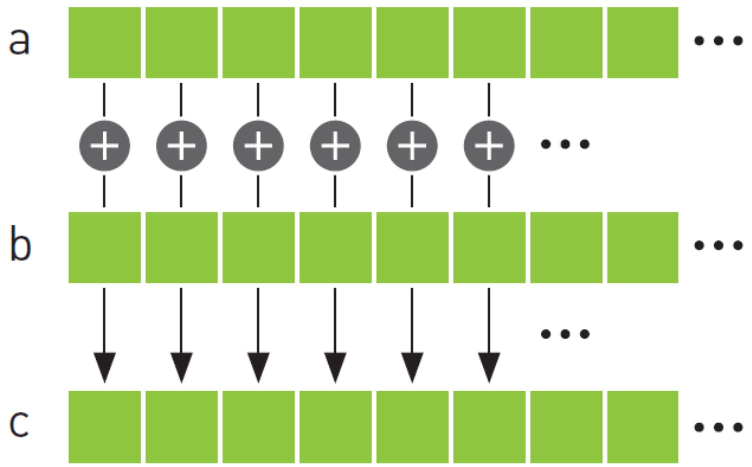
CUDA PROGRAMMING BY EXAMPLE

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];
   }
}
```

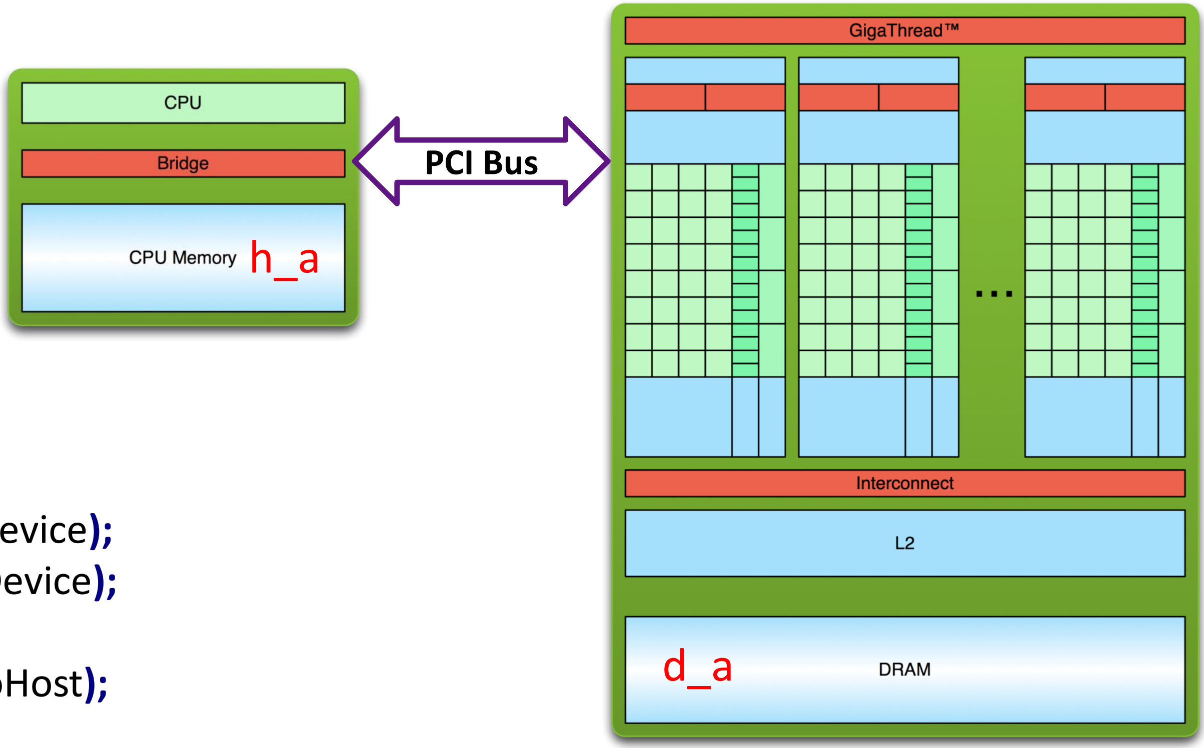
GPU CODE WORKFLOW

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;}
```



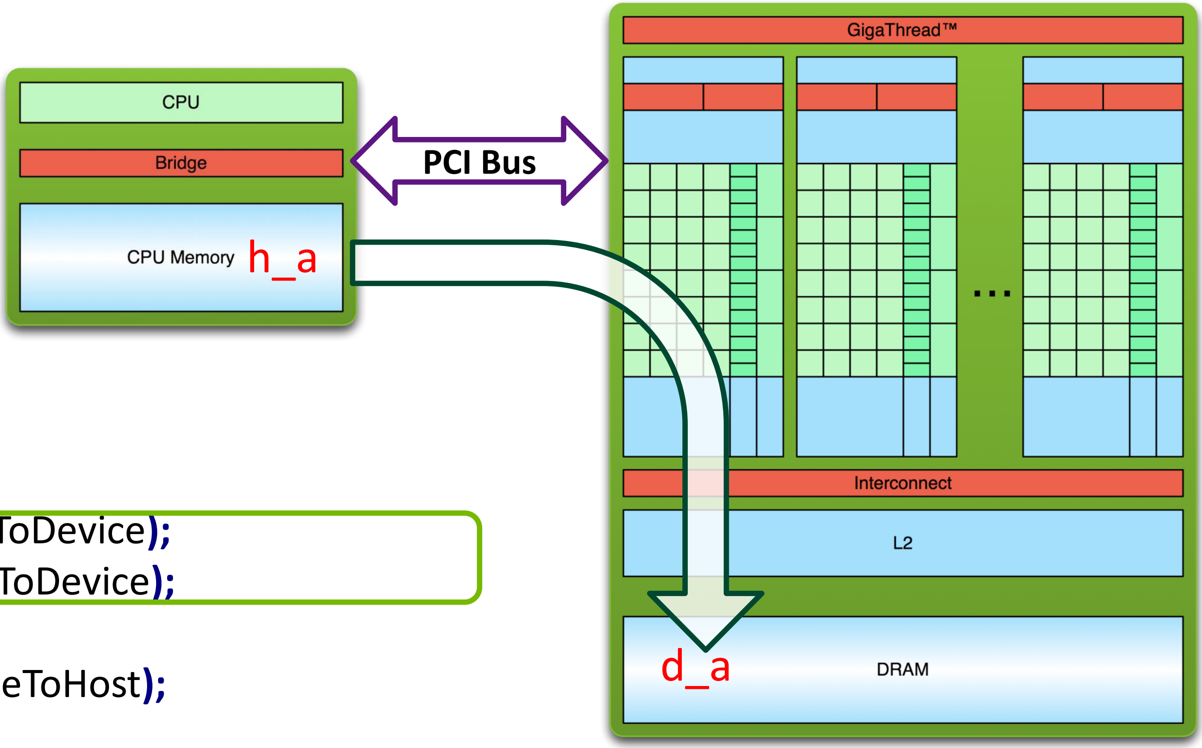
GPU CODE WORKFLOW

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

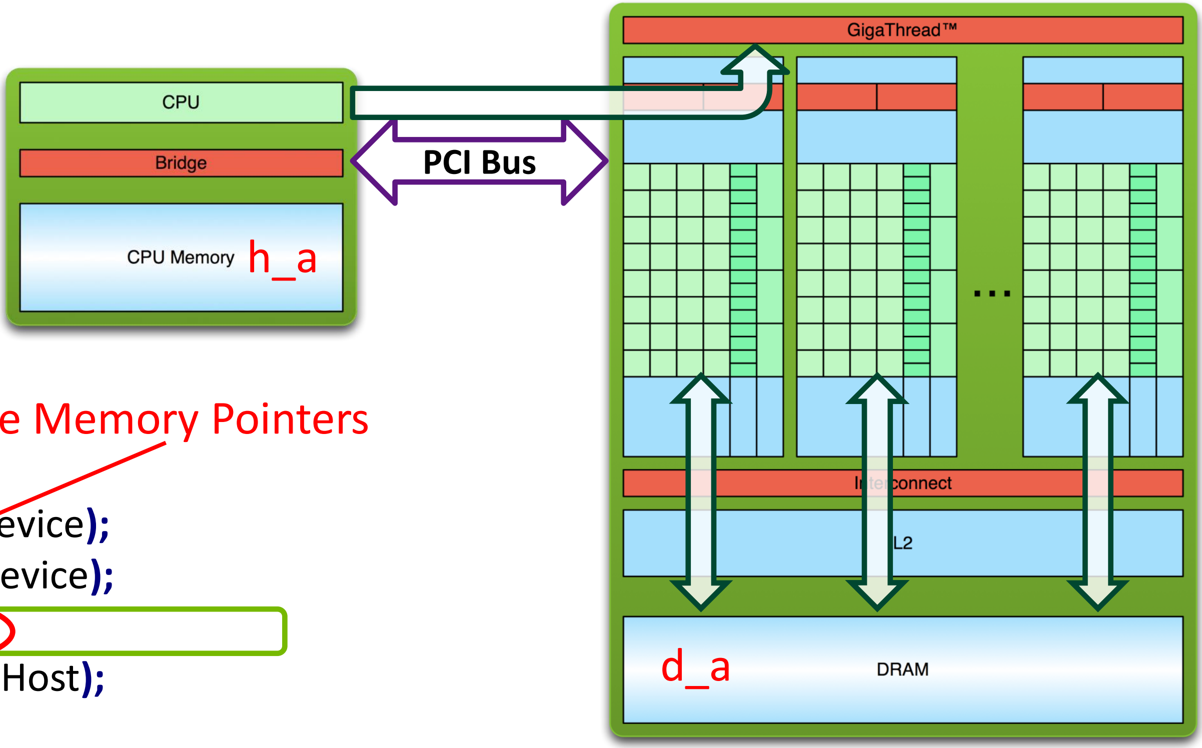
Invoke the CUDA Kernel

```
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 Memory Pointers



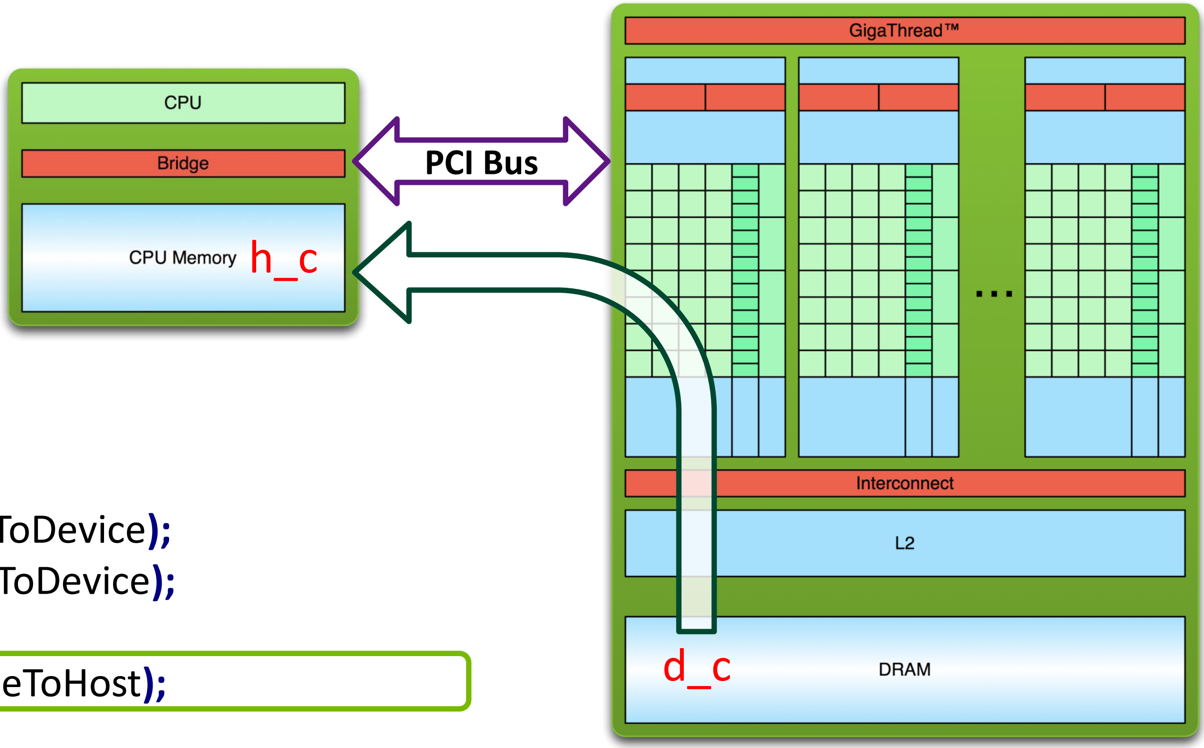
GPU CODE WORKFLOW

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;}
```



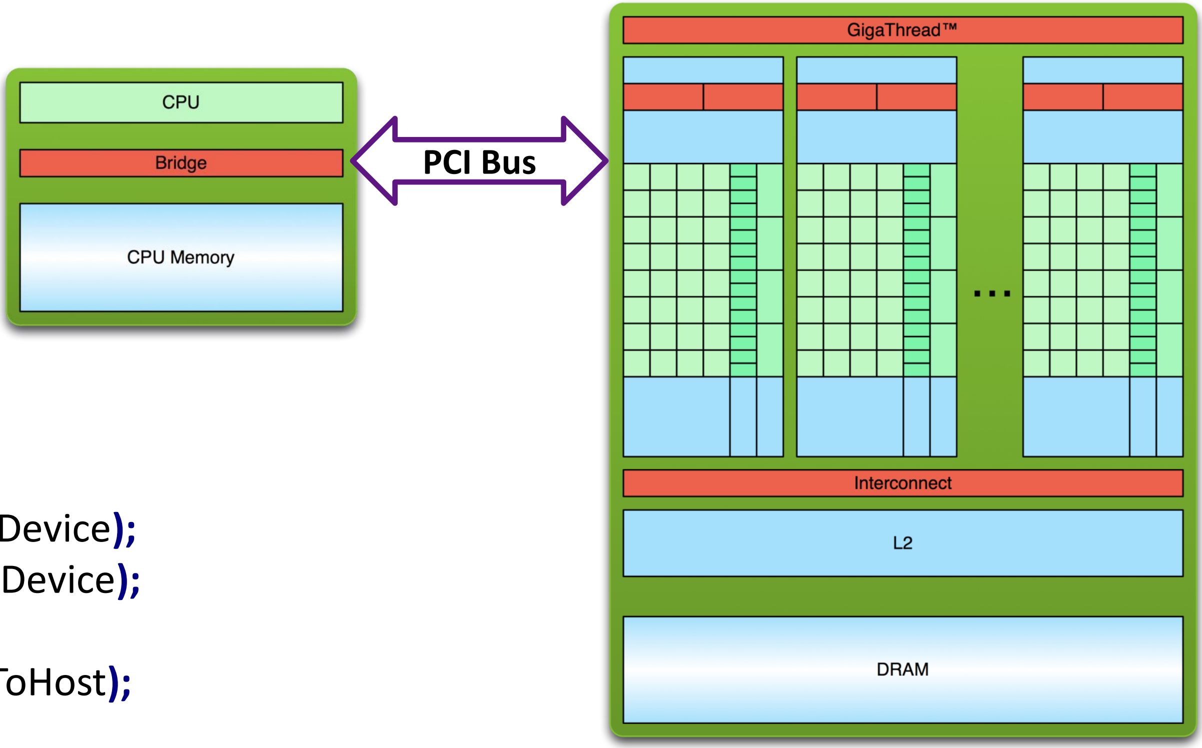
GPU CODE WORKFLOW

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;}
```



CUDA的线程索引

如何设置Gridsize & Blocksize：

```
block_size = 128;
grid_size = (N + block_size - 1) / block_size;
```

CUDA的线程分配

那么，我们的每个BLOCK可以申请多少个线程?

```
Total amount of shared memory per block:       49152 bytes
Total number of registers available per block: 65536
Warp size:                                     32
Maximum number of threads per multiprocessor:  2048
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

CUDA的线程分配

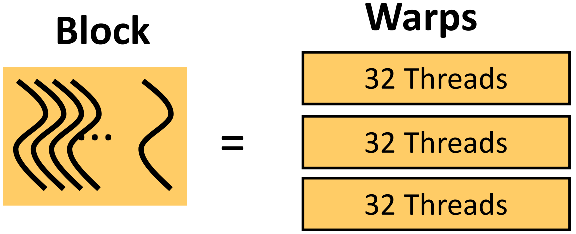
那么，我们的每个BLOCK可以申请多少个线程?

CUDA的线程

那么，我们的每个BLOCK应该申请多少个线程?

CUDA的线程分配

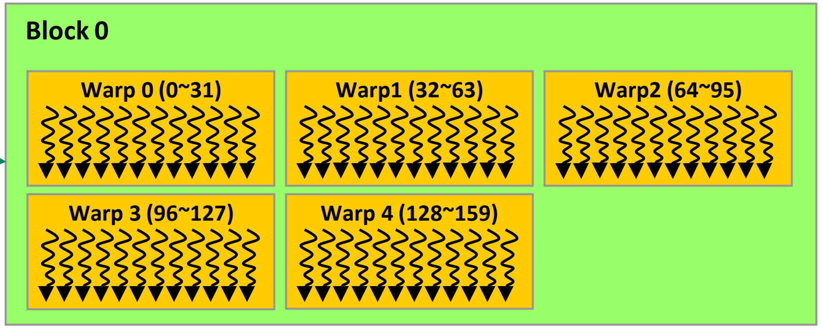
WARP



- Warp is successive 32 threads in a block

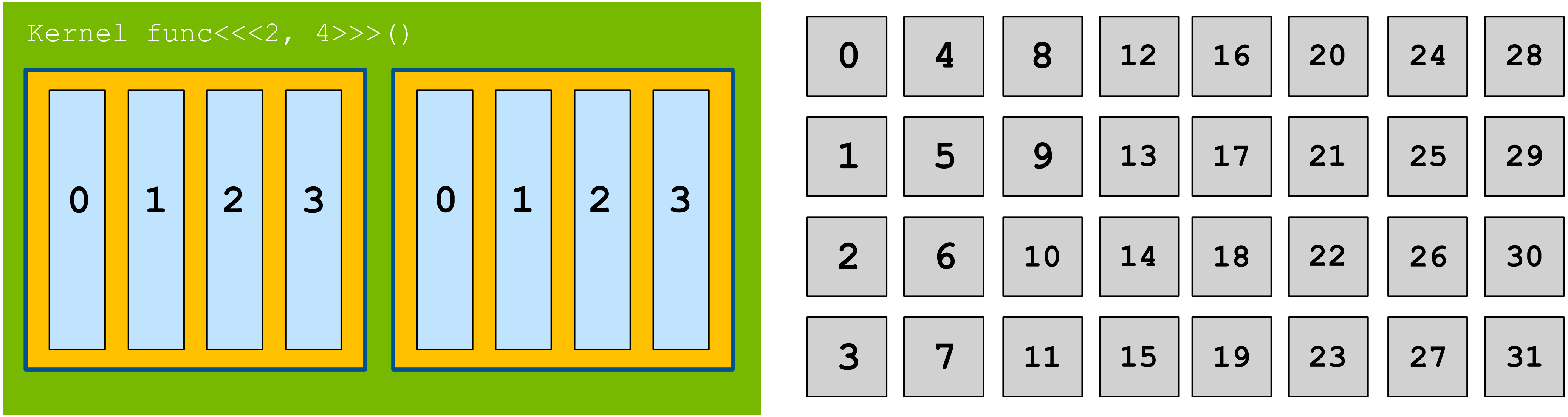
- E.g. blockDim = 160
  - Automatically divided to 5 warps by GPU

- E.g. blockDim = 161
  - If the blockDim is not the Multiple of 32
    The rest of thread will occupy one more warp



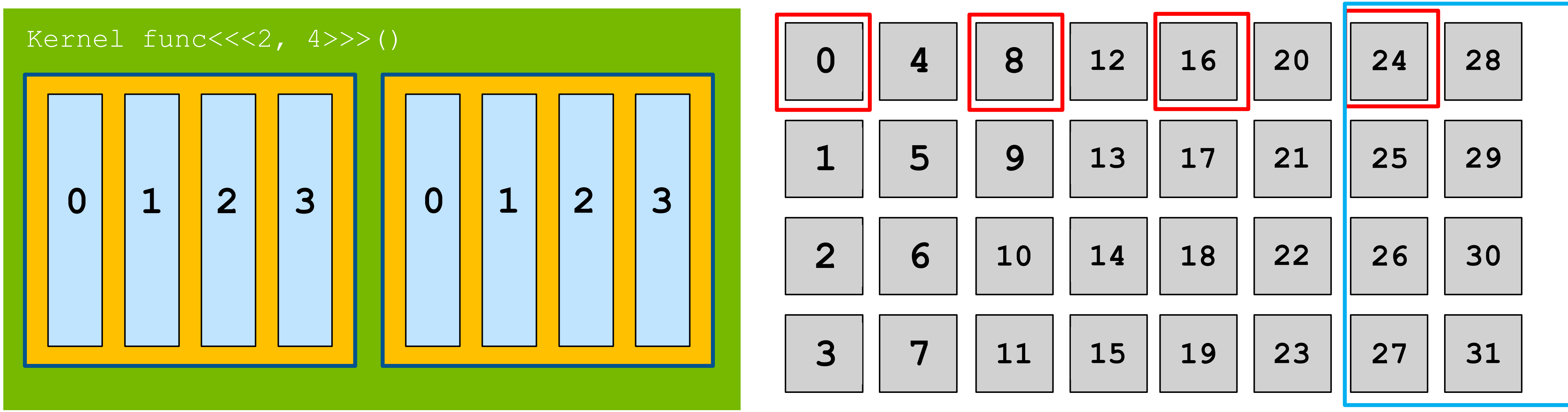
CUDA的线程

那么，如果我们的数据过大，线程不够用怎么办？



CUDA的线程

那么，如果我们的数据过大，线程不够用怎么办？



红色框代表索引值为0的线程处理的数据？

CUDA的线程

那么，如果我们的数据过大，线程不够用怎么办?

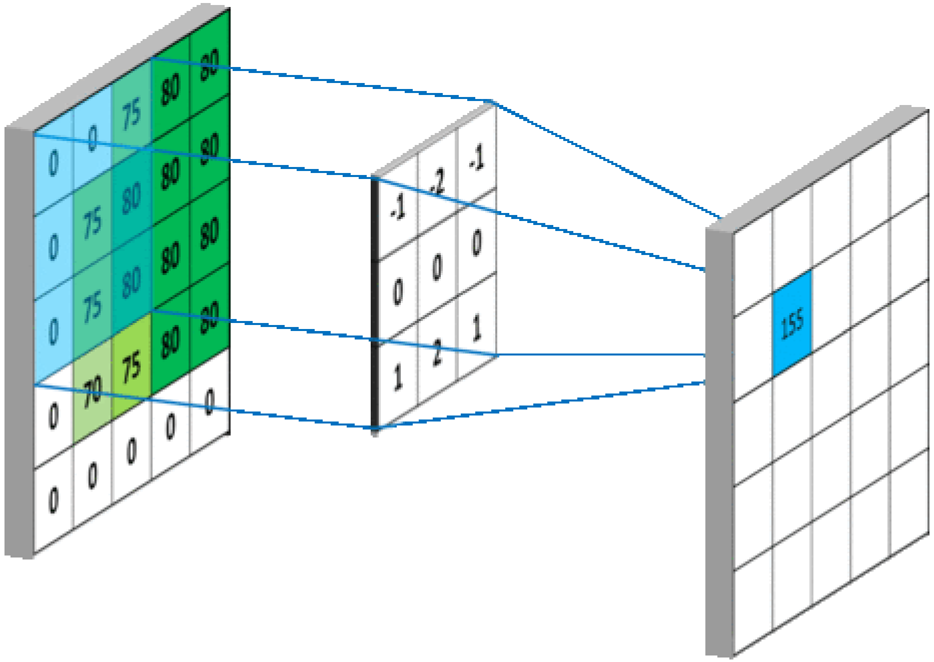
```
__global__ add(const double *x, const double *y, double *z, int n)
{
        int index = blockDim.x * blockIdx.x + threadIdx.x;
        int stride = blockDim.x * gridDim.x;
        for(; index <n; index +=stride)
                z[index] = x[index] + y[index];
}
```

CUDA的线程

那么，如果我们的数据过小，线程太多怎么办?

```
if( blockDim.x * blockIdx.x + threadIdx.x < count )
```

Sobel算子



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

更多资源：

何琨-Ken

北京 密云

扫一扫上面的二维码图案，加我微信

NVIDIA

加入 NVIDIA 开发者计划

获取最新版本软件、工具及开发信息

扫码了解详情

NVIDIA®