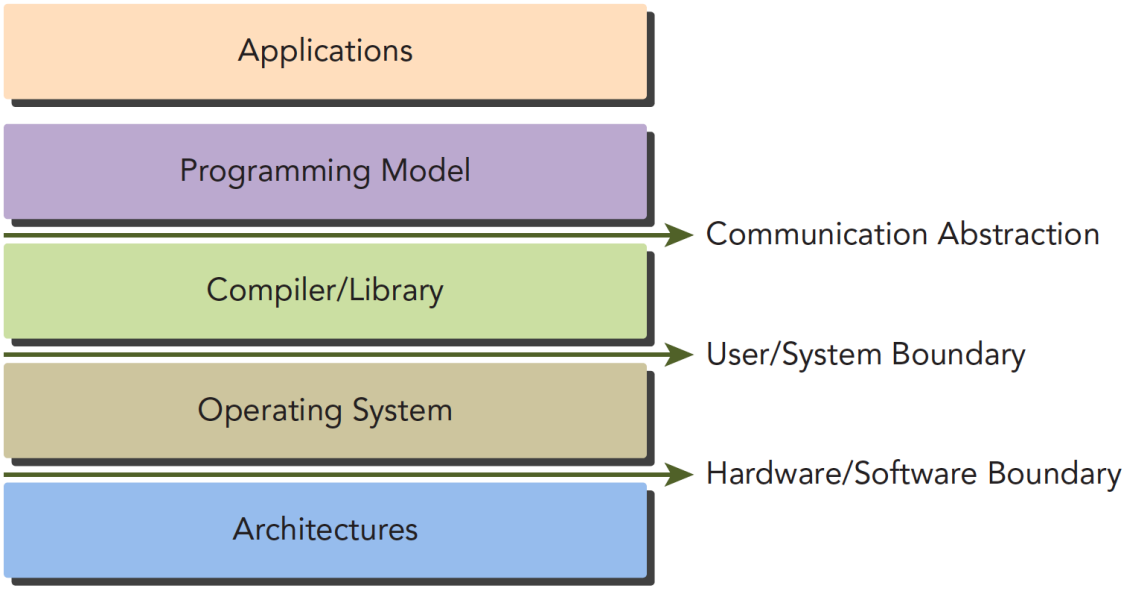
阅读

2022.04 chapter02

这次是2022.04月份，我觉得每次回头看知识都会获得新的内容

重头开始看blog内容：这个图对于编程模型和编译器具有一个Communication Abstraction



编程模型可以理解为，我们要用到的语法，内存结构，线程结构等这些我们写程序时我们自己控制的部分，这些部分控制了异构计算设备的工作模式，都是属于编程模型。（参考谭升blog）

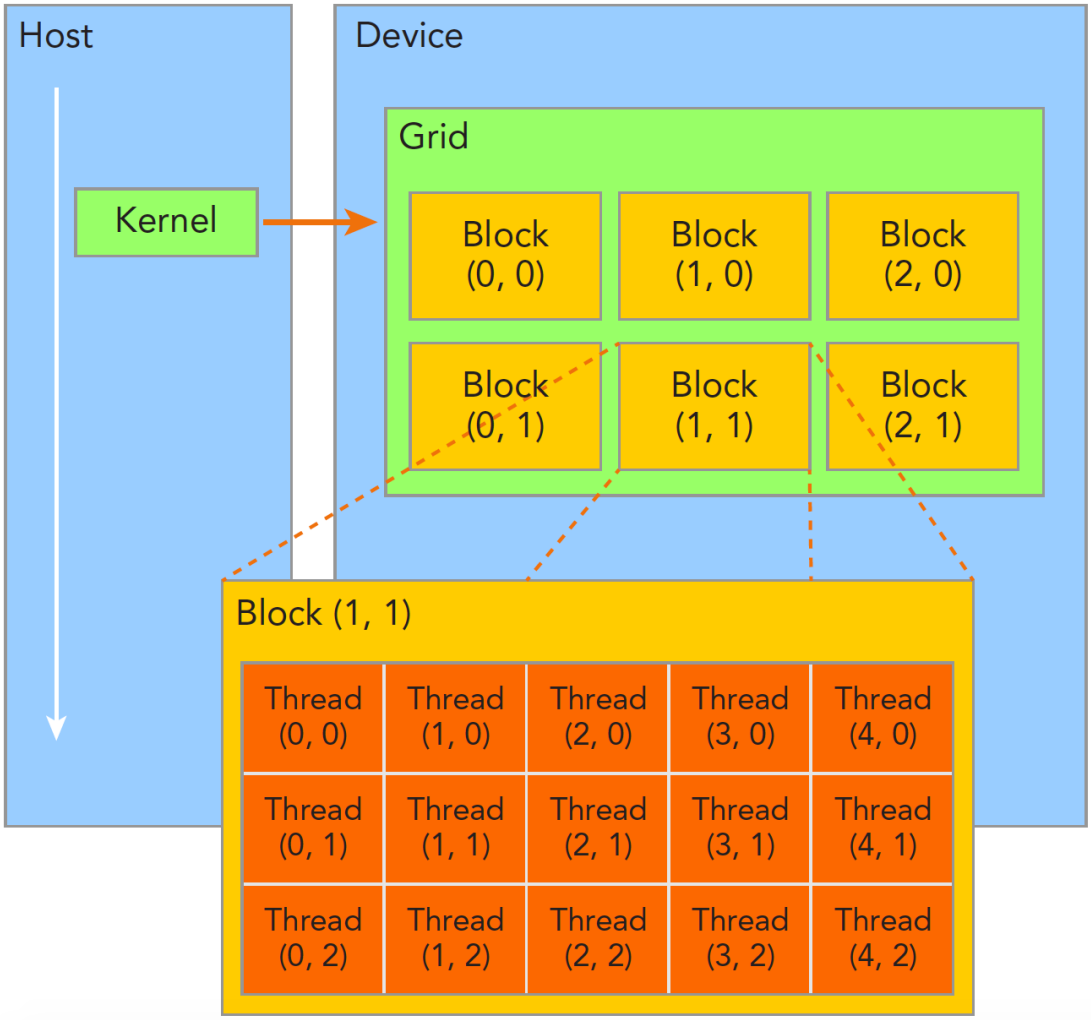
两个内存从硬件到软件都是隔离的（CUDA6.0 以后支持统一寻址）

- cuda和c接口

标准C函数	CUDA C 函数	说明
malloc	cudaMalloc	内存分配
memcpy	cudaMemcpy	内存复制
memset	cudaMemset	内存设置
free	cudaFree	释放内存

- cudaMemcpyHostToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

看了第二章较为关键的核函数的控制流，对于grid，block，thread的控制，这张图真好



其中在初始化核函数进去的时候grid已经定义了

在核函数运行的时候gridDim为一开始

对应dim3 grid 可通过在核函数内部GridDim.x GridDim.y GridDim.z

dim3 block也同理

```
<<<grid,block>>>
```

参考谭升的代码1_check_dimension

```
#include <cuda_runtime.h>
#include <stdio.h>
__global__ void checkIndex(void)
{
  printf("threadIdx:(%d,%d,%d) blockIdx:(%d,%d,%d) blockDim:(%d,%d,%d)\
  gridDim(%d,%d,%d)\n",threadIdx.x,threadIdx.y,threadIdx.z,
  blockIdx.x,blockIdx.y,blockIdx.z,blockDim.x,blockDim.y,blockDim.z,
  gridDim.x,gridDim.y,gridDim.z);
}
int main(int argc,char **argv)
{
  int nElem=6;
  dim3 block(3);
  dim3 grid((nElem+block.x-1)/block.x);
  printf("grid.x %d grid.y %d grid.z %d\n",grid.x,grid.y,grid.z);
  printf("block.x %d block.y %d block.z %d\n",block.x,block.y,block.z);
```

```
  checkIndex<<<grid,block>>>();
  cudaDeviceReset();
  return 0;
}
```

output

```
grid.x 2 grid.y 1 grid.z 1
//-> dim3 grid(2) same as dim3 grid(2,1,1)
block.x 3 block.y 1 block.z 1
//-> dim3 block(3) same as dim3 block(3,1,1)
// 这边后续blockDim girdDim比较好理解
// thread block 该部分就好从上chapter中得到，一个核函数为一个grid 所以grid设置好了进行划分
了
// checkIndex<<<grid,block>>>();
// 在核函数创建的过程中，会开辟一个网格grid空间，该空间物理隔离，该grid将会被拆分为block和
thread，对应参数<<<grid,block>>>,第一个参数制定了block的维度网格，第二个参数再被细化为
thread
// grid分割的过程可粗略的理解为先分割block再分割出thread，在网格块中block为gridDim维度对
应，thread对应blockDim
// threadIdx 应该是block参数对应
threadIdx:(0,0,0) blockIdx:(1,0,0) blockDim:(3,1,1)  gridDim(2,1,1)
threadIdx:(1,0,0) blockIdx:(1,0,0) blockDim:(3,1,1)  gridDim(2,1,1)
threadIdx:(2,0,0) blockIdx:(1,0,0) blockDim:(3,1,1)  gridDim(2,1,1)
threadIdx:(0,0,0) blockIdx:(0,0,0) blockDim:(3,1,1)  gridDim(2,1,1)
threadIdx:(1,0,0) blockIdx:(0,0,0) blockDim:(3,1,1)  gridDim(2,1,1)
threadIdx:(2,0,0) blockIdx:(0,0,0) blockDim:(3,1,1)  gridDim(2,1,1)
```

由于该sumArraysGPU中已经计算好该一维向量为1<<14 =1<<4 x 1<<10 = 16*1024

接下来去理解sumArraysGPU中的位置偏移量，由于a和b为一个一维向量，但是被grid切分为block的形式，

原本是在核函数传参<<<grid,block>>> ------- <<<16,1024>>> 划分为16个block 对应gridDim(16,1,1)

blockIdx:(0-15,0,0)，每个block被划分为1024个thread，然后就开始解析如下语句，需要有一定的**数据结构和地址理解**

那么再看16*1024个数据刚刚每个数据一个地址不需要重复计算，首先会将数据划分为16块

blockIdx.x该参数为block的第一维0-15变化，blockDim.x为1024其实就是地址偏移，thread.Idx为线程块，

由于一个块只计算一个值，所以底下仅仅跟一个，如果把改成一个线程两个相对应的block 要缩小一半，

如果不信可以进行测试

```
__global__ void sumArraysGPU(float*a,float*b,float*res)
{
  //int i=threadIdx.x;
// if(threadIdx.x==0 || threadIdx.x==1023){
 // printf("blockIdx.x,blockDim.x,threadIdx.x:
(%d,%d,%d)\n",blockIdx.x,blockDim.x,threadIdx.x);
  //}
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  res[i]=a[i]+b[i];
}
```

output中为会有，一维的角度来看更好理解了

```
blockIdx.x,blockDim.x,threadIdx.x:(15,1024,0)
blockIdx.x,blockDim.x,threadIdx.x:(9,1024,0)
blockIdx.x,blockDim.x,threadIdx.x:(3,1024,0)
blockIdx.x,blockDim.x,threadIdx.x:(13,1024,0)
blockIdx.x,blockDim.x,threadIdx.x:(1,1024,0)
```