第二章 数据级并行计算

我们将用很多的代码例子来说明编写并行代码中的关键概念，这就需要一种简单的编程语言，这种语言能够支持大规模并行编程和异构计算。最终我们选择了CUDA C作为编程例子和练习。CUDA C继承自C语言并且只增加了少量的语法与接口就让编程者能够在CPU核心和大型并行GPU上进行异构编程。从CUDA C的名字不难看出，CUDA C是建立在英伟达的CUDA平台上的。CUDA是目前大规模并行编程的最成熟的框架，在最常见的操作系统下被广泛应用与高性能计算领域，并且有着各种成熟的工具（例如：编译器、debugger、profiler）

有个很重要的点：虽然我们为了简单和通用，大多数例子都是用的CUDA C，但是CUDA平台还支持很多其他的编程语言API，例如C++，Python，Fortran，OpenCL，OpenACC，OpenMP等等。CUDA是一种支持将一系列概念组织起来，来达到大规模并行计算的架构，我们教的就是这些概念。为了其他语言的开发者方便（C++, Fortran, Python, OpenCL等等），在附录中可以查看这些概念在其他语言中的应用。

2.1数据级并行

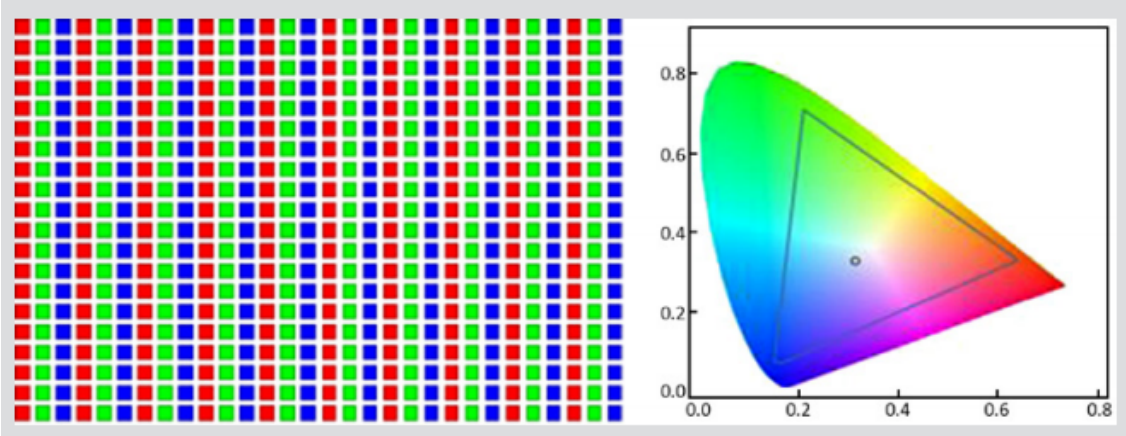
当现代的软件应用跑的慢时，很大肯能是软件要处理的数据太庞大了。用户的软件处理有成百上千万个像素点的图片或者视频，科学家的软件用上亿的网格来模拟流体力学，分子动力学必须模拟成千至成百万个原子之间的相互作用，航空时刻表需要处理上千的飞机，工作人员，登机口等数据。更重要的是，这些像素、分子、细胞、相互作用、飞机等等元素都能被单独处理。将一个颜色的像素转换成黑白需要的数据仅仅是那个像素点的数据，将图片中每个像素的颜色与附近像素的颜色想融合（虚化）需要的仅仅是一小部分像素的数据。即使是一个看上去的全局操作（global operation），比如寻找所有的像素的平均亮度，也可以被分解成更斯奥的计算单元，然后每个计算单元单独执行。像这样的单独执行操作是数据级并行的基础——（重新）组织数据的计算，使得组织后的数据单元能够执行独立的并行计算，来更快地完成整体工作。

> **任务级并行 VS 数据级并行**
>
> 数据级并行不是并行编程中的唯一一种并行方法，任务级并行也被广泛地应用。任务级并行常用在任务被拆解时，例如：一个简单的应用需要做一个向量加法和矩阵乘法，这两个任务如果能够独立完成，那么就可以用任务级并行。再比如I/O和数据传输就是常见的并行任务。
>
> 在大型应用中，经常有大量的独立完成的任务，也就有大量的任务级并行运算。比如：在模拟分子运动时，任务列表中有震动力，旋转力，用来计算非结合力的邻近分子识别，速度和位置的计算，还有其他基于速度和位置的物理性质等等。
>
> 总的来说，数据级并行是并行程序的可扩展性来源。有了大量的数据级后，人们总是能找到大量的数据级并行，并将之用在大规模并行处理器上，使得软件的性能随着硬件的发展（更新换代带来更多的计算资源）而发展。任务级并行同样在提升性能上扮演者重要角色。我们会在提及“流”时去讲任务级并行。

我们将在下一章利用图片处理来作为例子，这里先用彩色变黑白来讲一下数据级并行的概念。图2.1左边的图中的每个像素都包含红绿蓝的值（rgb），红绿蓝每个的值都是从0（纯黑）到1（满色）。

RGB颜色展现的原理

在RGB表示法中，图片中的每个像素都被存储为一个（r,g,b）的元组。如下图所示，一个图片的行的格式为 (r g b) (r g b) … (r g b)。每个元组代表着红（R）绿（G）蓝（B）的混合，也就是说每个像素的三个从0道1的值分别表示着红绿蓝从无色到满色。



这三种颜色所能组合成的所有色彩组成的空间在工业上被称为色域。在右图中的三角形展示了AdobeRGB三种色彩所能组合成的色域。垂直坐标（y值）与水平坐标（x值）表示了G和R部分的值，剩下的B部分的值为（1-y-x）。在渲染图片时，每个像素的rgb值都用来计算色彩强度（亮度）和混合系数（mixture coefficients）（x,y,1-y-x）

将左侧的彩色图片变成右侧的黑白图片，我们需要通过这个公式计算亮度L

$$L = r * 0.21 + g * 0.72 + b * 0.07$$

如果我们认为图片是按照一个RGB值数组I输入的，并且要求输出是对应的亮度值组成的数组O，那么计算的结构就像图2.2所示的那样。例如：O[0]是通过I[0]的值根据公式，乘以对应的权重加和而得到的，O[1]通过同样的方法得到I[1]，O[2]通过同样的方法得到I[2]，以此类推。没有任何像素点的计算是依赖其他像素点的，所有像素点都单独计算。很明显，彩色变黑白的过程显现了大量的数据级并行。当然，数据级并行在完整的应用中会非常复杂，本书知识想教大家这种并行思想并实际应用书记级并行。

2.2CUDA C程序结构

我们现在已经准备好开始学习CUDA C编程去试试数据级并行来加速软件性能了。CUDA C的程序需要你的电脑上有一个host（CPU）和一个或多个devices（GPU），每个CUDA源文件都能有host和devices的代码。默认情况下，任何传统C语言只包含host代码。你可以在任何原文件中增加device的代码函数据声明。device的函数据声明都用CUDA C的关键字清晰地标注出来。这些函数分地展示了数据级并行。

当device函数和数据声明加到原文件中后，传统的C编译器不再能编译这个文件了，代码需要被一个能够识别并理解新增加的代码表达的编译器编译。我们会使用一个叫做NVCC（NVIDIA C Compiler）的CUDA C编译器。正如图2.3所示，NVCC用CUDA关键字来分开host代码与device代码，从而来处理CUDA C程序。host代码是标准的ANSI C代码，是被host的标准C/C++编译器编译并作为一个传统CPU进程运行。device代码被CUDA关键字标记作为数据级并行函数部分被为kernels和与kernels相关的帮助函数（helper function）与数据结构。device代码被一个NVCC的运行时组件（run-time component）进一步编译并执行在一个GPU设备上。在没有device硬件设备或者kernel能够在CPU上合理运行时，我们还可以选择利用像MCUDA这样的工具让kernel在CPU上执行。

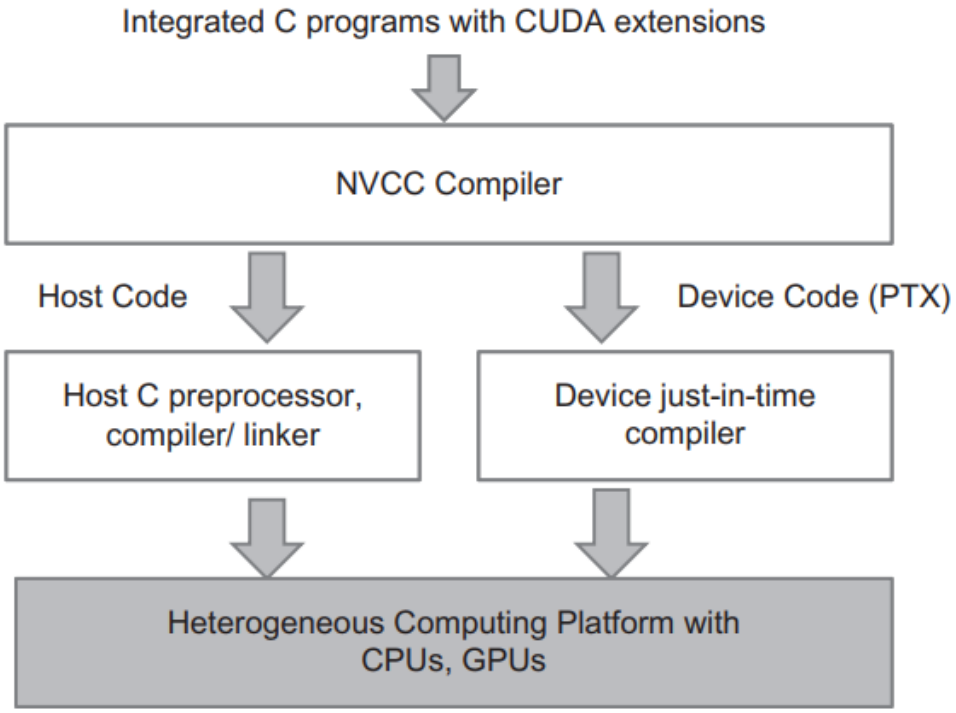


FIGURE 2.3

Overview of the compilation process of a CUDA C Program.

一个CUDA程序的执行过程如图2.4所示。执行过程从host代码（CPU执行的代码）开始，当kernel函数（并行device代码）被调用或者加载时，这部分会被device上大量的线程执行。kernel加载时产生的所有的线程被称为网格（grid）。这些线程是CUDA平台上并行运行的主要载体。图2.4展示了两个线程网格的执行。我们一会儿将会讨论这些网格是如何组织架构的。当一个kernel的所有线程都完成了他们的执行操作后，相应的网格会终结掉，host的代码会继续执行，知道下一个kernel被加载。注意，图2.4展示的是CPU执行与GPU执行不相互重合的建议情况，很多异构计算软件会使CPU与GPU的执行相互重合，进而发挥出CPU和GPU的所有性能。

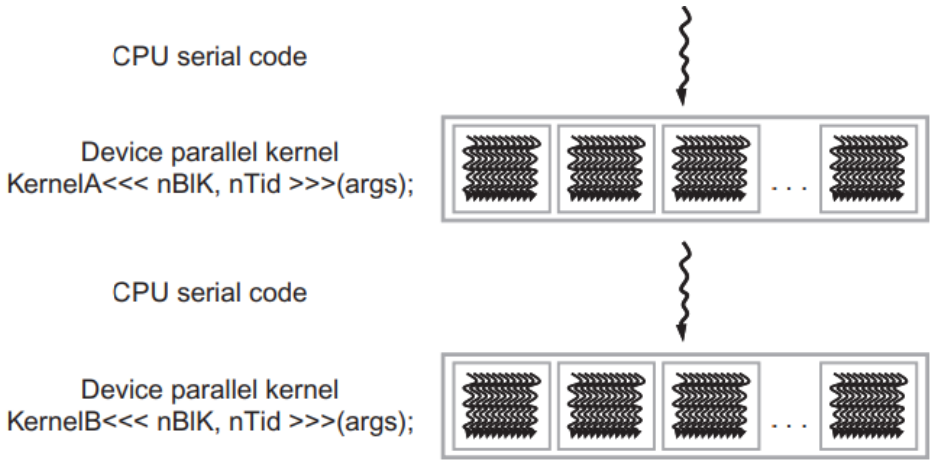


FIGURE 2.4

Execution of a CUDA program.

加载一个kernel需要产生大量的线程来实现数据级并行。在彩色变黑白的例子中，每个线程都能计算一个像素的输出值L，这种情况下，产生的线程数会等于图片的像素数，更大的图片自然会产生更多的线程数。实际情况中，一个线程通常会处理多个像素点来提高效率。CUDA编程者可以认为这些线程的生成和排布由于硬件的优良支持只消耗非常短的时钟周期。这点与传统CPU线程产生时消耗上千个时钟周期完全不同。

> **线程**
>
> 线程是现代计算机处理器执行一个代码序列的简化描述。一个线程是由程序代码、正在执行的特定代码以及其变量和数据结构的值组成。对于用户而言，线程代码的执行是顺序进行的。我们可以使用源码级debugger，通过一次执行一个语句，查看下一条将要执行的语句，并在执行过程中检查变量和数据结构的值，进而来监视线程的进度。
>
> 多线程已经被用在在编程中好多年了。如果一个编程者在一个应用中想要开始并行执行，他会用线程库或者特殊的编程语言来创建和管理线程。在CUDA中，每个线程也是顺序执行的，一个CUDA程序通过加载kernel函数开始并行执行，这会导致底层机制开始创建许多线程，这些线程并行处理不同数据。

2.3向量加法kernel

我们现在用向量加法来战术CUDA C的程序结构。向量加法可以说是最简单的数据级并行计算了，就像序列化编程中的“Hello World”一样。在我们展示向量加法的kernel code之前，先来看一下常规向量加法在host code上是怎么执行的。图2.5展示了一个简单的传统C语言程序，它由main函数和一个向量加法函数组成。在我们所有的例子中，每当我们要区分host数据和device数据是，我们给在变量的名字加上前缀。“h_"表示host的变量，“d_”表示device的变量，因为在图2.5中只有host代码，所以你只能看到“h_”开头的变量。

```
// Compute vector sum h_C = h_A+h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  for (int i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements each
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

FIGURE 2.5

A simple traditional vector addition C code example.

假设被加的向量被存储为数组A和数组B，且二者在主程序中已经分配了内存并初始化。输出的向量存储在数组C中，同样在主程序中分配了内存。为了简便，我们并没有展示ABC在主函数中分配内存与初始化的细节。这些数组的指针和向量长度N被传到vecAdd函数。注意，vecAdd函数的形参使用了“h_”前缀来表示这些变量是被host处理的。这个命名习惯在我们一会儿介绍device代码时很有帮助。

图2.5中的vecAdd韩式使用了for循环来便利所有的向量元素。在第i个循环中，h_C[i]是h_A[i]和h_B[i]的和。相连长度n用来控制循环，使循环的系数等于向量长度。形参h_A，h_B和h_C通过引用传递，使得该函数通过参数指针A，B和C读取h_A，h_B的元素并写入h_C的元素。当vecAdd函数返回时，main函数中的后续语句可以访问c的新内容。

一个并行执行向量加法直接的方法是更改vecAdd函数并将之改到用device计算。这种改动的方法如图2.6所示，在文件的开头，我们需要加一个C预处理器知识本代码包含cuda.h头文件。这个文件定义了CUDA API函数和内置变量，我们一会儿讲。函数的第一部分在GPU内存中分配了内存用来存储ABC向量，并将这些向量的值从host内存复制到device内存。第二部分在device上加载了向量加法的并行计算kernel。第三部分从device内存复制向量C的值给到host内存，并释放device上三个向量的内存。

> **C语言的指针**
>
> 图2.4中的函数变量A、B、C是指针。在C语言中，指针能够用来获取变量和数据结构。当一个浮点数变量v被声明为：
>
> ```
> float v;
> ```
>
> 一个指针变量p会被声明为：
>
> ```
> float *p;
> ```
>
> 通过`p = &v`把v的地址存储到指针变量p中，就可以说指针p指向v。`*p` 也就编程了v的代名词。例如 `u = *p` 把v的值给到u，再比如说 `*p = 3` 把v的值变成3。
>
> 一个C程序的数组能够通过指向其第0个元素的指针来获取。比如：`p = &(A[0])` 使p指向数组A的第0个元素，p[i]变成了A[i]的代名词。实际上数组的名字A本身也是指向其第0个元素的指针。
>
> 在图2.5中，把数组A的名字传给函数vecAdd作为第一个参数，使得函数的第一个形参h_A指向A的第0个元素，我们称之为A通过引用传值传给了vecAdd。

注意，修改后的vecAdd函数本质上是一个外包代理，它将输入数据运送到设备，激活设备上的计算，并从设备中收集结果。这个代理使主程序根本不需要知道向量加法在device上的完成情况。实际上，这个透明的外包模型会被来回的复制数据而导致效率低下。我们应当在device上保留一批重要数据并在host代码中调用device函数来操作这些数据。现阶段为了介绍CUDA C的程序结构，我们仍保留使用这一简单的透明外包模型。在本章剩余章节中将介绍修改后的函数的具体细节和如何构建kernel函数。

2.4device全局内存和数据传输

在现在的CUDA系统中，devices通常是有着自己动态内存（DRAM）的硬件设备。比如NVIDIA GTX1080就有一个8GB的DRAM，我们称之为全局内存。全局内存与device内存这两个术语可以相互替换。为了在device上执行kernel，编程者必须在device上分配全局内存并把相关数据从host内存传输到device内存中。这个过程对应图2.6中的第一步。同样，在device执行完毕后，编程者还需要从device内存中将数据传输回host内存并释放掉不再使用的device内存，这个过程对应图2.6中的第三步。CUDA运行时系统（runtime system）为程序员提供展示这些行为的API。从这一点出发，我们将数据从host内存复制到device内存简称为数据从host到device的传输。反向依然如此。

图2.7为编程者展示了CUDA的host内存和device内存的抽象图，变成自可借此在device上分配内存并在二者间传输数据。device的全局内存能够被host访问并传入或传出数据。device还有不同于图2.7的内存种类，常量内存只能被device函数以只读方式获取，这个在第7章介绍。在第4章，我们要讲寄存器的使用和共享内存。感兴趣的朋友可以查看CUDA programming guide来查看不同内存的功能。现阶段，我们只关注全局内存的使用。

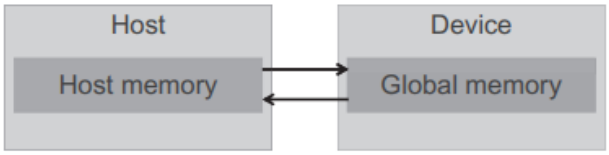


FIGURE 2.7

Host memory and device global memory.

> **内置变量**
>
> 很多编程语言都有内置变量。这些变量有着特殊的意思和用途。这些变量的值通常被运行时系统预先初始化了并且在程序中只读。编程者应当避免将这些变量用于其他用途。

在图2.6中，vecAdd函数的第一部分和第三部分需要使用CUDA API函数来分配A、B、C的内存，将A、B从host内存传输到device内存，在向量加法完成后再将C从device内存传输到host内存，然后释放A、B、C在device上的内存。我们先解释内存分配和释放。

在图2.8中展示了两个API函数，分别是用来分配device内存和释放device内存的。cudaMalloc函数能够在host代码中调用，功能是为一个对象在device中分配一段全局内存。读者应当发现了cudaMalloc和C语言库中的malloc函数非常相似，这是故意这么设计的，CUDA是小幅扩展版的C。CUDA使用标准C语言库中malloc函数来管理host内存并增加cudaMalloc最为C的库的扩展。CUDA通过保持新接口与原来的C接口尽可能的相似，来减少C编程者学习的时间。

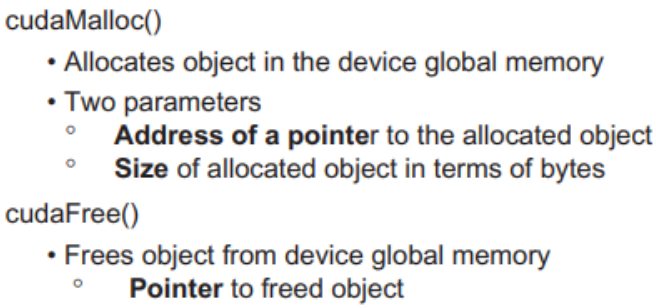


FIGURE 2.8

CUDA API functions for managing device global memory.

cudaMalloc的第一个参数是一个指向对象将被分配的位置的指针的**地址**。指针变量的地址应当用 (void **) 进行强制类型转换，因为这个函数的期待一个指针作为参数，内存分配函数是一个普通函数，它并没有限制分配对象的具体类型，这个参数允许cudaMalloc函数将要分配的内存地址写到指针变量中，加载kernel的host代码将这个指针变量的值传递给需要获取这个内存的对象。cudaMalloc的第二个参数要表明分配的内存的大小，用数字表示，单位是字节byte。第二个参数的用法与C的malloc函数一样。

我们现在用一个简单的代码例子来展示cudaMalloc的用法。这个例子是图2.6的续集，为了表示清楚，我们将创建一个以“d_”开头的指针变量来表示它指向device内存中的一个对象。这个程序把“d_A”的地址强制转换成void指针，然后当做第一个参数传给了函数，也就是说，“d_A”将指向device内存中分配给A向量的区域，分配的区域将会是一个单精度浮点数大小（大多数电脑上是4字节）的n倍。在计算完成之后，cudaFree函数将被调用，并传递"d_A"来传递，用来释放A向量在device内存上的存储空间。要注意的是cudaFree并不需要转换指针变量“d_A”的值，它只需要使用“d_A”的值来使分配的内存回到可用池（available pool）中。所以，只传“d_A”这个指针就行，不用把“d_A”的地址传进去作为参数。

```
float *d_A;
int size=n * sizeof(float);
cudaMalloc((void**)&d_A, size);
...
cudaFree(d_A);
```

d_A, d_B, 和d_C中的地址是device内存中的地址，这些地址不应当在host代码中接引计算，他们大多数情况应当被用来调用API函数和kernel函数，在host代码中接引会导致在执行时发生异常或者其他种类的报错。

读者应当用相似的声明语句完成图2.6中第一部分中对d_B和d_C的声明和对cudaMalloc函数的调用，还有第三部分中对二者cudaFree函数的调用。

当host代码已经为数据对象分配device内存后，就能够将数据从host传到device，同样是通过调用CUDA的API函数。图2.9展示了这个API函数，叫cudaMemcpy，这个函数需要4个参数。第一个参数是一个指向被复制数据对象目标地址的指针，第二个参数是指向原位置的指针，第三个参数复制的内容大小，单位是字节，第四个参数表示复制的类型：是从host内存到host内存，还是从host内存到device内存，还是从device内存到host内存，还是从device内存到device内存。举个例子，这个函数能够从device一个内存地址复制到device的另一个内存地址上去。

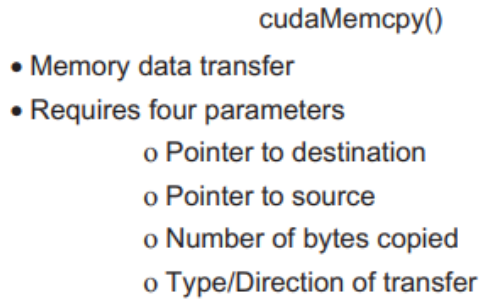


FIGURE 2.9

CUDA API function for data transfer between host and device.

vecAdd函数调用cudaMemcpy函数将h_A和h_B在向量加法之前从host复制到device，并在完成向量加法之后，将h_C从device复制到host。假设h_A, h_B, d_A, d_B的值和size大小已经确定，那么这三个cudaMemcpy函数的调用如下图所示。CUDA的编译环境会识别cudaMemcpyHostToDevice和cudaMemcpyDeviceToHost这两个预定义好的常量关键字。注意相同的函数能够被用来双向传递数据，只要妥善处理好数据源地址和目标地址并使用相应的常量表示传递类型。

```
cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
```

总结来说，图2.5中的vecAdd函数是执行在host上的。图2.6中的vecAdd函数要分配device内存，请求数据传输，加载kernel完成向量加法。我们通常称这部分host代码为加载kernel的根函数（stub function），在kernel执行完成后，vecAdd还会将计算的结果从device复制到host上。完整的代码如图2.10所示

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code - to be shown later
    ...

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

FIGURE 2.10

A more complete version of `vecAdd()`.

> **CUDA的错误检查和处理**
>
> 总的来说，检查和处理错误对一个程序来说非常重要。在CUDA的API函数中，会返回一个代表着是否发生了错误的标志。大多数错误都是来源于给函数传参错误。
>
> 为了简便，我们在例子中不会展示错误检测代码。比如图2.10调用了cudaMalloc函数：

```
cudaMalloc((void **) &d_A, size);
```

实际上我们应该我们应当将这段代码包裹在测试错误的代码中，并在发生错误是展示错误信息。一个简单地检查报错的代码如下：

```
cudaError_t err = cudaMalloc((void //) &d_A, size);
if (error != cudaSuccess) {
    printf("%s in %s at line %d\n", cudaGetErrorString(err), __FILE__, __LINE__);
    exit(EXIT_FAILURE);
}
```

这样的话，如果系统的device内存不足，用户会知道这个信息，这回节省太多的debug时间。

我们还可以使用C宏来让源代码中检查报错更精准

相比较于图2.6，图2.10中的第一部分和第三部分更加完整。第一部分为d_A, d_B, 和d_C分配内存并把h_A传给d_A，把h_B 传给d_B，这是通过调用cudaMalloc函数和cudaMemcpy函数来完成的。 我们鼓励读者自己去写函数的调用并自己去传参，然后与图2.10中的代码进行比较。第二部分调用了kernel并将会在下一小节进行介绍。第三部分将加和后的数据从device内存复制到host内存，然后这个结果才能被主函数所访问，这一步是通过cudaMemcpy函数完成的。然后在释放d_A, d_B, 和d_C 的内存，这一步是通过cudaFree函数完成的。

2.5kernel函数和线程

我们现在已经要开始讲CUDA的kernel函数还有加载这些kernel函数的影响了。在CUDA中，一个kernel函数标志着代码要被所有的线程并行执行。因为所有这些线程执行相同的代码，所以CUDA程序是很具有代表性的SPMD（single-program multiple data）并行风格，这种风格在大型并行计算系统中十分流行。

当一个程序的host代码加载kernel时，CUDA的运行时系统生成一网格的线程并将这些线程分成两个等级，每个网格都是由多个线程组成的区块构成，一个网格的所有区块都是相同大小，每个区块最多容纳1024个线程。图2.11展示了每个区块由256个线程组成，每个线程用一个数组方格表示，并作为曲线箭头的起点。每个区块的线程总数是由host代码加载kernel时确定的，在host代码的不同部分，即使加载相同的kernel也可以用不同的线程数。对于给定的网格，区块中的线程数能够通过内置变量blockDim访问。

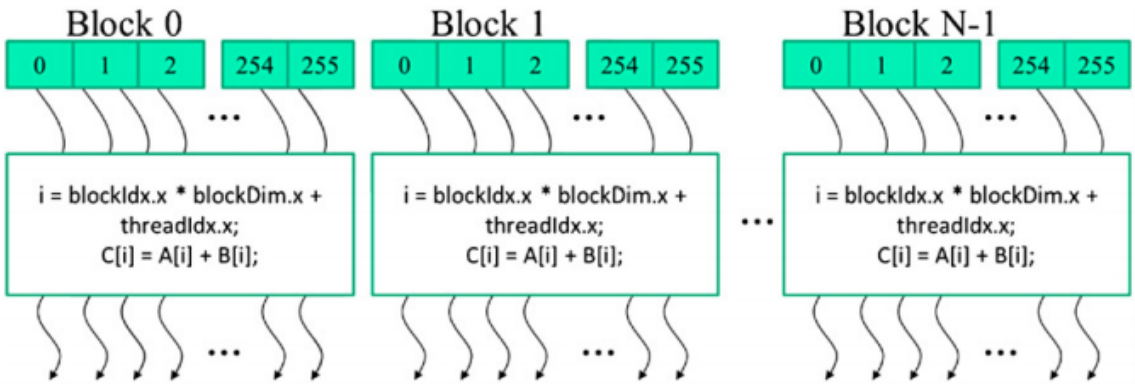


FIGURE 2.11

All threads in a grid execute the same kernel code.

blockDim变量是一个struct类型的变量，拥有三个非负整形的属性（field）x, y和z，这三个值能够帮助编程者将这些线程组织称一维、二维或者三维的数组。对于一维的情况而言，只会用到x属性，二维的话会用到x和y属性，三维的话，三个属性都会用到。如何安排线程的维度通常反映了数据的维度，这是因为线程被创造出来就是为了并行处理数据的。在图2.11中，因为数据是一维的，所以线程区块也被安排

成一维的。blockDim.x表示每个区块中线程的数量，在图2.11中是256个。总的来说，由于硬件方面的效率原因，线程区块中每个维度的线程数应当是32的倍数，这点一会再谈。

CUDA的kernel还能够访问另外两个内置变量：threadIdx和blockIdx。这两个变量能够让线程之间能够相互区分并决定每个线程在数据的哪部分工作。threadIdx变量给区块中的每个线程一个单独的名称，例如图2.11中，因为我们只使用了一维分布，所以只使用了threadIdx的x属性，因此图2.11中的第一个线程的threadIdx.x的值是0，下一个线程threadIdx.x的值是1，在下一个是2，以此类推。

blockIdx变量给所有的线程一个区块名称。在图2.11中，第一个区块的所有的线程的blockIdx.x的值是0，第二个区块的blockIdx.x 的值为1，以此类推。用电话号码做个类比，你可以把threadIdx想象成地区电话号，把blockIdx想象成区域号码，两个号码共同作用才能在一个国家中确定一个唯一的电话号码。与之类似，用threadIdx和blockIdx共同决定在一个网格中的唯一线程。

在图2.11中，一个唯一的全局索引用这个公式计算：`i = blockIdx.x * blockDim.x + threadIdx.x`。在我们这个例子中blockDim是256，区块0中的i值范围是0至255，区块1中的i值范围是256到511，区块2中的i值范围是512到767，可见这三个区块的i值范围是从0至767。每个线程用索引值i来访问A、B和C，这些线程第一次循环迭代768次。注意我们在这里不会再用“h_”和“d_”表达，因为这里不会涉及host代码，因此不会用到误导性名称。通过用大量的区块加载kernel，我们可以处理大型向量。通过用n个或者更多个线程加载kernel，我们能够出来长度为n的向量。

图2.12展示了一个向量加法的kernel函数。语法是ANSI C外加一些扩展。首先是一个CUDA C的关键字“__global__”，位于vecAddKernel函数之前，这个关键字表示：这个函数是一个kernel函数并且它能够被host函数调用，从而在device上生成一网格的线程。

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
   int i = blockDim.x*blockIdx.x + threadIdx.x;
   if(i<n) C[i] = A[i] + B[i];
}
```

FIGURE 2.12

A vector addition kernel function.

总的来说，CUDA C用三个关键字限定词扩展了C语言，这三个关键字能够用在函数的声明中。这三个关键字的总结如图2.13。“__global__”关键字表示这个函数是CUDA C的kernel函数，注意在global两侧分别由两个下划线。这样的kernel函数会在device上执行并且只能被支持动态并行的CUDA系统中的host代码所调用，动态并行我们将在第13章介绍。“__device__”关键字表示声明的函数是一个CUDA的device函数，device函数在CUDA上执行，并且只能被kernel函数或者其他的device函数所调用。

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

FIGURE 2.13

CUDA C keywords for function declaration.

“__host__”关键字表示声明的函数是CUDA的host函数，一个host函数是一个传统的C函数，这个函数在host上执行，并且只能被另一个host函数调用。默认情况下，如果函数在声明时没有使用关键字，那么这个函数就是host函数，毕竟CUDA应用是从CPU-only执行环境中移植过来的。编程者在移植时应当增加kernel函数和device函数，剩下的函数就当做host函数，这简化了程序员为每个函数增加声明的工作

量。

注意“__host__”和“__device__”能够用在同一个函数声明中使用，这两个一起使用的话，编译系统会生成同一个函数的两个版本的对象文件，一个用来在host上执行并且只能被host函数调用，另一个在device上执行且只能被device或者kernel函数执行。我们可以看出来，同一段源代码能够被再编译成device的版本，这一点很常用，很多用户的库函数就能够被在编译成device版本。

如图2.12，第二个基于ANSI C的扩展点是threadIdx、blockIdx和blockDim。回忆一下，所有的线程执行一段相同的kernel代码，这就需要有一种方法能够区分不同的线程并将它们导向数据的不同部分。这些内置的变量是用来让线程通过制定的坐标来访问硬件上的寄存器。不同的线程将会看到不同的threadIdx、blockIdx和blockDim值。此处为了简便，我们用（blockIdx.x， threadIdx.x）表示一个线程。注意.x表示的是这个变量还应该有.y和.z，我们一会儿在讨论这一点。

在图2.12中有一个本地变量（automatic variable / local variable）i。在kernel函数中，本地变量是属于每一个线程的私有变量，也就是说每一个线程都会产生一个变量i，如果kernel产生了10000个线程，那就有10000个不同的i，一个线程的i对其他线程不可见，我们将在第四章介绍本地变量。

比较一下图2.5与图2.12就能较为明显地理解CUDA的kernel以及kernel加载。图2.12中的kernel函数相较于图2.5并没有循环体。读者们可能会想：我草，循环跑他妈的哪里去了？答案是：循环被线程网格所取代了，整个的网格与循环的作用一样。网格中的每个线程对应着原始代码中的一次循环。像这种用线程的并行来处理原始代码中循环的数据级并行，通常情况下也被称之为循环级并行。

注意在图2.12中的addVecKernel函数中，有一个if(i < n)表达式，这是因为并不是所有的向量长度都是区块大小的整数倍。举个例子，假设向量的长度是100，最小的区块所拥有的的线程数是32，我们应当用4个区块来处理这100个元素的向量。但是，4个区块会产生128个线程，我们应当取消掉区块3（区块编号从0开始）的最后28个线程的操作。既然所有的线程都执行相同的操作，我们就应当在程序中先判断i是否大于100,。这样通过if条件语句，我们似的前100个线程执行相应的加法操作，而后28个线程并不会执行这个操作，这就能够让kernel去处理不同长度的向量。

当host代码加载kernel时，是通过执行配置参数（execution configuration parameters）来设置网格和线程区块维度的。如图2.14，配置参数在传统C函数参数前，在<<<和>>>之间给出。第一个配置参数提供网格中的线程区块数，第二个参数表明每个区块中的线程数。在这个例子中，我们每个区块有256个线程，为了保证我们有足够的线程来运算所有的向量元素，我们用C的ceil函数取n/256的整数部分+1的值。我们使用256.0是为了使得除法得到的数是浮点数，从而被ceil函数正确地处理。例如我们有1000个线程，我们就需要ceil(1000/256.0)=4个区块。因此，我们实际上加载了4*256=1024个线程，通过图2.12中的if表达式，使得前1000个线程正常执行家法操作，而剩下的24个线程不会执行。

```
int vectAdd(float* A, float* B, float* C, int n)
{
//  d_A, d_B, d_C allocations and copies omitted
// Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
}
```

FIGURE 2.14

A vector addition kernel launch statement.

2.6kernel加载

图2.15展示了vecAdd函数的最终host代码。这个源代码完全补全了图2.6、图2.12、图2.15所提供的代码框架，并展示了一个既有host代码又有device的kernel的一个完整且简单的CUDA程序。这个代码在硬件上，每个区块有256个线程，具体使用多少区块，取决于向量的长度n。如果n是750，那么久用三个区块，如果n是4000，那么就用16个区块，如果n是2000000，那就用7813个区块。注意所有的线程区块在向量的不同部位执行，他们能够以任意顺序执行，编程者不可以臆断执行的顺序，一个小型的、没怎么堆料的GPU可能用以时间会并行执行1个或者2个区块，一个大型GPU可能会同时并行执行64或者128个区块。相同的一段代码在不同的条件的硬件上有着不同的执行速度，也就意味着CUDA的kernel有着硬件方面的可扩展性。我们会在第三章讨论这一点。

```
void vecAdd(float* A, float* B, float* C, int n)
{

  int size = n * sizeof(float);
  float *d_A, *d_B, *d_C;

  cudaMalloc((void **) &d_A, size);
  cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
  cudaMalloc((void **) &d_B, size);
  cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

  cudaMalloc((void **) &d_C, size);

  vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);

  cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

        // Free device memory for A, B, C
  cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

FIGURE 2.15

A complete version of the host code in the vecAdd.function.

必须要说明的是，拿这个向量加法举例子只是为了方便和简单，真实情况下那些分配device内存，把输入从host传到device，输出从device传到host，还有释放device内存这些操作会使得我们代码的执行速度甚至不如图2.5所示的顺序执行，这是由于相比较于要处理的数据，kernel所做的计算太少了，知识对两个浮点型的操作数执行加法操作并输出一个浮点型操作数。真实应用中往往会让kernel做大量的操作，从而让之前那些操作所花费的时间是值得的。而且通常也会在device内存中保存这些数据，让统一份数据被多个kernel所使用从而减少这些操作的重复。我们以后将会举几个这样的应用的例子。

2.7总结

总结了一些本章提到的关键名词：function declarations, kernel launch, built-in(predefined) variables, run-time API

详细内容见原书，再次不赘述。

2.8习题

详细内容见原书