使用 CUDA C/C++ 加速应用程序

加速计算正在取代 CPU 计算，成为最佳计算做法。加速计算带来的层出不穷的突破性进展、对加速应用程序日益增长的需求、轻松编写加速计算的编程规范以及支持加速计算的硬件的不断改进，所有这一切都在推动计算方式必然会过渡到加速计算。

无论是从出色的性能还是易用性来看，CUDA 计算平台均是加速计算的制胜法宝。CUDA 提供一种可扩展 C、C++、Python 和 Fortran 等语言的编码范式，能够在世界上性能超强劲的并行处理器 NVIDIA GPU 上运行大量经加速的并行代码。CUDA 可以毫不费力地大幅加速应用程序，具有适用于 DNN、BLAS、图形分析 和 FFT 等的高度优化库生态系统，并且还附带功能强大的 命令行 和 可视化分析器。

CUDA 支持以下领域的许多（即便不是大多数）世界上性能超强劲的应用程序：计算流体动力学、分子动力学、量子化学、物理学 和高性能计算 (HPC)。

学习 CUDA 将能助您加速自己的应用程序。加速应用程序的执行速度远远超过 CPU 应用程序，并且可以执行 CPU 应用程序受限于其性能而无法执行的计算。在本实验中, 您将学习使用 CUDA C/C++ 为加速应用程序编程的入门知识，这些入门知识足以让您开始加速自己的 CPU 应用程序以获得性能提升并助您迈入全新的计算领域。

如要充分利用本实验,您应已能胜任如下任务：

To get the most out of this lab you should already be able to:

- 在 C 中声明变量、编写循环并使用 if/else 语句。
- 在 C 中定义和调用函数。
- 在 C 中分配数组。

无需 CUDA 预备知识。

目标

当您在本实验完成学习后，您将能够：

- 编写、编译及运行既可调用 CPU 函数也可**启动** GPU **核函数** 的 C/C++ 程序。
- 使用**执行配置**控制并行**线程层次结构**。
- 重构串行循环以在 GPU 上并行执行其迭代。
- 分配和释放可用于 CPU 和 GPU 的内存。
- 处理 CUDA 代码生成的错误。
- 加速 CPU 应用程序。

加速系统

*加速系统*又称*异构系统*，由 CPU 和 GPU 组成。加速系统会运行 CPU 程序，这些程序也会转而启动将受益于 GPU 大规模并行计算能力的函数。本实验环境是一个包含 NVIDIA GPU 的加速系统。可以使用 `nvidia-smi` (*Systems Management Interface*) 命令行命令查询有关此 GPU 的信息。现在，可以在下方的代码执行单元上使用 `CTRL` + `ENTER` 发出 `nvidia-smi` 命令。无论您何时需要执行代码，均可在整个实验中找到这些单元。代码运行后，运行该命令的输出将打印在代码执行单元的正下方。在运行下方的代码执行块后，请注意在输出中找到并记录 GPU 的名称。

In [ ]:
```
!nvidia-smi
```

由GPU加速的还是纯CPU的应用程序

以下幻灯片以可视化的方式概要地介绍了后面将学习的内容。请点击浏览一遍这些幻灯片，然后再继续深入了解以下章节中的主题。

In [ ]:
```
%%HTML

<div align="center"><iframe src="https://view.officeapps.live.com/op/view
```

为GPU编写应用程序代码

CUDA 为许多常用编程语言提供扩展，而在本实验中，我们将会为 C/C++ 提供扩展。这些语言扩展可让开发人员在 GPU 上轻松运行其源代码中的函数。

以下是一个 `.cu` 文件（`.cu` 是 CUDA 加速程序的文件扩展名）。其中包含两个函数，第

一个函数将在 CPU 上运行，第二个将在 GPU 上运行。请抽点时间找出这两个函数在定义方式和调用方式上的差异。

```
void CPUFunction()
{
  printf("This function is defined to run on the CPU.\n");
}

__global__ void GPUFunction()
{
  printf("This function is defined to run on the GPU.\n");
}

int main()
{
  CPUFunction();

  GPUFunction<<<1, 1>>>();
  cudaDeviceSynchronize();
}
```

以下是一些需要特别注意的重要代码行，以及加速计算中使用的一些其他常用术语：

`__global__ void GPUFunction()`

- `__global__` 关键字表明以下函数将在 GPU 上运行并可**全局**调用，而在此种情况下，则指由 CPU 或 GPU 调用。
- 通常，我们将在 CPU 上执行的代码称为**主机**代码，而将在 GPU 上运行的代码称为**设备**代码。
- 注意返回类型为 `void`。使用 `__global__` 关键字定义的函数需要返回 `void` 类型。

`GPUFunction<<<1, 1>>>();`

- 通常，当调用要在 GPU 上运行的函数时，我们将此种函数称为**已启动**的**核函数**。
- 启动核函数时，我们必须提供**执行配置**，即在向核函数传递任何预期参数之前使用 `<<< ... >>>` 语法完成的配置。
- 在宏观层面，程序员可通过执行配置为核函数启动指定**线程层次结构**，从而定义线程组（称为**线程块**）的数量，以及要在每个线程块中执行的**线程**数量。稍后将在本实验深入探讨执行配置，但现在请注意正在使用包含 `1` 线程（第二个配置参数）的 `1` 线程块（第一个执行配置参数）启动核函数。

`cudaDeviceSynchronize();`

- 与许多 C/C++ 代码不同，核函数启动方式为**异步**：CPU 代码将继续执行*而无需等待核函数完成启动*。
- 调用 CUDA 运行时提供的函数 `cudaDeviceSynchronize` 将导致主机 (CPU) 代码暂作等待，直至设备 (GPU) 代码执行完成，才能在 CPU 上恢复执行。

练习：编写一个Hello GPU核函数

`01-hello-gpu.cu`（*<---- 点击源文件链接，以在另一个标签中打开并进行编辑*）包含已在运行的程序。其中包含两个函数，都有打印 \"Hello from the CPU\" 消息。您的目标是重构源文件中的 `helloGPU` 函数，以便该函数实际上在 GPU 上运行，并打印指示执行此操作的消息。

- 请先重构应用程序，然后使用下方的 `nvcc` 命令编译和运行该应用程序（请记住，您可以通过使用 `CTRL + ENTER` 启动该应用程序以执行代码执行单元的内容）。`01-hello-gpu.cu` 中的注释将有助您完成操作。如您遇到问题或要检查自己的操作，请参阅 解决方案。在使用下方命令进行编译和运行之前，请不要忘记保存对文件所作的更改。

In [ ]:
```
!nvcc -arch=sm_70 -o hello-gpu 01-hello/01-hello-gpu.cu -run
```

成功重构 `01-hello-gpu.cu` 后，请进行以下修改，并尝试在每次更改后编译并运行该应用程序（通过使用 `CTRL + ENTER` 点击上方的代码执行单元）。若出现错误，请花时间仔细阅读错误内容：熟知错误内容会在您开始编写自己的加速代码时，给与很大的帮助。

- 从核函数定义中删除关键字 `__global__`。注意错误中的行号：您认为错误中的 \"configured\" 是什么意思？完成后，请替换 `__global__`。
- 移除执行配置：您对 \"configured\" 的理解是否仍旧合理？完成后，请替换执行配置。
- 移除对 `cudaDeviceSynchronize` 的调用。在编译和运行代码之前，猜猜会发生什么情况，可以回顾一下核函数采取的是异步启动，且 `cudaDeviceSynchronize` 会使主机执行暂作等待，直至核函数执行完成后才会继续。完成后，请替换对 `cudaDeviceSynchronize` 的调用。
- 重构 `01-hello-gpu.cu`，以便 `Hello from the GPU` 在 `Hello from the CPU` **之前**打印。
- 重构 `01-hello-gpu.cu`，以便 `Hello from the GPU` 打印**两次**，一次是在 `Hello from the CPU` **之前**，另一次是在 `Hello from the CPU` **之后**。

编译并运行加速后的CUDA代码

本节包含上述为编译和运行 `.cu` 程序而调用的`nvcc` 命令的详细信息。

CUDA 平台附带 **NVIDIA CUDA 编译器** `nvcc`，可以编译 CUDA 加速应用程序，其中包含主机和设备代码。就本实验而言， `nvcc` 的讨论范围将根据我们的迫切需求据实确定。完成本实验学习后，有意深究 `nvcc` 的所有用户均可从 文档 开始入手。

曾使用过 `gcc` 的用户会对 `nvcc` 感到非常熟悉。例如，编译 `some-CUDA.cu` 文件就很简单：

```
nvcc -arch=sm_70 -o out some-CUDA.cu -run
```

- `nvcc` 是使用 `nvcc` 编译器的命令行命令。
- 将 `some-CUDA.cu` 作为文件传递以进行编译。
- `o` 标志用于指定编译程序的输出文件。
- `arch` 标志表示该文件必须编译为哪个**架构**类型。本示例中， `sm_70` 将用于专门针对本实验运行的 Volta GPU 进行编译，但有意深究的用户可以参阅有关 `arch` 标志、虚拟架构特性 和 GPU特性 的文档。
- 为方便起见，提供 `run` 标志将执行已成功编译的二进制文件。

CUDA的线程层次结构

以下幻灯片以可视化的方式概要地介绍了后面将学习的内容。请点击浏览一遍这些幻灯片，然后再继续深入了解以下章节中的主题。

In [ ]:
```
%%HTML

<div align="center"><iframe src="https://view.officeapps.live.com/op/view
```

启动并行运行的核函数

程序员可通过执行配置指定有关如何启动核函数以在多个 GPU **线程**中并行运行的详细信息。更准确地说，程序员可通过执行配置指定线程组（称为**线程块**或简称为**块**）数量以及其希望每个线程块所包含的线程数量。执行配置的语法如下：

```
<<< NUMBER_OF_BLOCKS, NUMBER_OF_THREADS_PER_BLOCK>>>
```

启动核函数时，核函数代码由每个已配置的线程块中的每个线程执行。

因此，如果假设已定义一个名为 `someKernel` 的核函数，则下列情况为真：

- `someKernel<<<1, 1>>()` 配置为在具有单线程的单个线程块中运行后，将只运行一次。
- `someKernel<<<1, 10>>()` 配置为在具有 10 线程的单个线程块中运行后，将运行 10 次。
- `someKernel<<<10, 1>>()` 配置为在 10 个线程块（每个均具有单线程）中运行后，将运行 10 次。
- `someKernel<<<10, 10>>()` 配置为在 10 个线程块（每个均具有 10 线程）中运行后，将运行 100 次。

练习： 启动并行运行的核函数

`01-first-parallel.cu` 目前已实现十分基本的函数调用，即打印消息 `This should be running in parallel`。请按下列步骤首先进行重构使之在 GPU 上运行，然后在单个线程块中并行运行，最后则在多个线程块中运行。如您遇到问题，请参阅 解决方案。

- 重构 `firstParallel` 函数以便在 GPU 上作为 CUDA 核函数启动。在使用下方 `nvcc` 命令编译和运行 `01-basic-parallel.cu` 后，您应仍能看到函数的输出。
- 重构 `firstParallel` 核函数以便在 5 个线程中并行执行，且均在同一个线程块中执行。在编译和运行代码后，您应能看到输出消息已打印 5 次。
- 再次重构 `firstParallel` 核函数，并使其在 5 个线程块内并行执行（每个线程块均包含 5 个线程）。编译和运行之后，您应能看到输出消息现已打印 25 次。

In [ ]:
```
!nvcc -arch=sm_70 -o basic-parallel 02-first-parallel/01-basic-parallel.c
```

CUDA提供的线程层次结构变量

以下幻灯片以可视化的方式概要地介绍了后面将学习的内容。请点击浏览一遍这些幻灯片，然后再继续深入了解以下章节中的主题。

In [ ]:
```
%%HTML

<div align="center"><iframe src="https://view.officeapps.live.com/op/view
```

线程和块的索引

每个线程在其线程块内部均会被分配一个索引，从 `0` 开始。此外，每个线程块也会被分配一个索引，并从 `0` 开始。正如线程组成线程块，线程块又会组成**网格**，而网格是 CUDA 线程层次结构中级别最高的实体。简言之，CUDA 核函数在由一个或多个线程块组成的网格中执行，且每个线程块中均包含相同数量的一个或多个线程。

CUDA 核函数可以访问能够识别如下两种索引的特殊变量：正在执行核函数的线程（位于线程块内）索引和线程所在的线程块（位于网格内）索引。这两种变量分别为 `threadIdx.x` 和 `blockIdx.x`。

练习：使用特定的线程和块索引

目前，`01-thread-and-block-idx.cu` 文件包含一个正在打印失败消息的执行中的核函数。打开文件以了解如何更新执行配置，以便打印成功消息。重构后，请使用下方代码执行单元编译并运行代码以确认您的工作。如您遇到问题，请参阅 解决方案。

In [ ]:
```
!nvcc -arch=sm_70 -o thread-and-block-idx 03-indices/01-thread-and-block-
```

加速for循环

对 CPU 应用程序中的循环进行加速的时机已经成熟：我们并非要顺次运行循环的每次迭代，而是让每次迭代都在自身线程中并行运行。考虑以下“for 循环”，尽管很明显，但还是请注意它控制着循环将执行的次数，并会界定循环的每次迭代将会发生的情况：

```
int N = 2<<20;
for (int i = 0; i < N; ++i)
{
  printf("%d\n", i);
}
```

如要并行此循环，必须执行以下 2 个步骤：

- 必须编写完成**循环的单次迭代**工作的核函数。
- 由于核函数与其他正在运行的核函数无关，因此执行配置必须使核函数执行正确的次数，例如循环迭代的次数。

练习：使用单个线程块加速for循环

目前，`01-single-block-loop.cu` 内的 `loop` 函数运行着一个“for 循环”并将连续打印 `0` 至 `9` 之间的所有数字。将 `loop` 函数重构为 CUDA 核函数，使其在启动后并行执行 `N` 次迭代。重构成功后，应仍能打印 `0` 至 `9` 之间的所有数字。如您遇到问题，请参阅解决方案。

In [ ]:
```
!nvcc -arch=sm_70 -o single-block-loop 04-loops/01-single-block-loop.cu -
```

协调并行线程

以下幻灯片以可视化的方式概要地介绍了后面将学习的内容。请点击浏览一遍这些幻灯片，然后再继续深入了解以下章节中的主题。

In [ ]:
```
%%HTML

<div align="center"><iframe src="https://view.officeapps.live.com/op/view
```

调整线程块的大小以实现更多的并行化

线程块包含的线程具有数量限制：确切地说是 1024 个。为增加加速应用程序中的并行量，我们必须要能在多个线程块之间进行协调。

CUDA 核函数可以访问给出块中线程数的特殊变量：`blockDim.x`。通过将此变量与 `blockIdx.x` 和 `threadIdx.x` 变量结合使用，并借助惯用表达式 `threadIdx.x + blockIdx.x * blockDim.x` 在包含多个线程的多个线程块之间组织并行执行，并行性将得以提升。以下是详细示例。

执行配置 `<<<10, 10>>>` 将启动共计拥有 100 个线程的网格，这些线程均包含在由 10 个线程组成的 10 个线程块中。因此，我们希望每个线程（`0` 至 `99` 之间）都能计算该线程的某个唯一索引。

- 如果线程块 `blockIdx.x` 等于 `0`，则 `blockIdx.x * blockDim.x` 为 `0`。向 `0` 添加可能的 `threadIdx.x` 值（`0` 至 `9`），之后便可在包含 100 个线程的网格内生成索引 `0` 至 `9`。
- 如果线程块 `blockIdx.x` 等于 `1`，则 `blockIdx.x * blockDim.x` 为 `10`。向 `10` 添加可能的 `threadIdx.x` 值（`0` 至 `9`），之后便可在包含 100 个线程的网格内生成索引 `10` 至 `19`。
- 如果线程块 `blockIdx.x` 等于 `5`，则 `blockIdx.x * blockDim.x` 为 `50`。向 `50` 添加可能的 `threadIdx.x` 值（`0` 至 `9`），之后便可在包含 100 个线程的网格内生成索引 `50` 至 `59`。
- 如果线程块 `blockIdx.x` 等于 `9`，则 `blockIdx.x * blockDim.x` 为 `90`。向 `90` 添加可能的 `threadIdx.x` 值（`0` 至 `9`），之后便可在包含 100 个线程的网格内生成索引 `90` 至 `99`。

练习：加速具有多个线程块的For循环

目前，`02-multi-block-loop.cu` 内的 `loop` 函数运行着一个“for 循环”并将连续打印 `0` 至 `9` 之间的所有数字。将 `loop` 函数重构为 CUDA 核函数，使其在启动后并行执行 `N` 次迭代。重构成功后，应仍能打印 `0` 至 `9` 之间的所有数字。对于本练习，作为附加限制，请使用启动*至少 2 个线程块*的执行配置。如您遇到问题，请参阅 解决方案。

In [ ]:
```
!nvcc -arch=sm_70 -o multi-block-loop 04-loops/02-multi-block-loop.cu -ru
```

分配将要在GPU和CPU上访问的内存

CUDA 的最新版本（版本 6 和更高版本）已能轻松分配可用于 CPU 主机和任意数量 GPU 设备的内存。尽管现今有许多适用于内存管理并可支持加速应用程序中最优性能的 中高级技术，但我们现在要介绍的基础 CUDA 内存管理技术不但能够支持远超 CPU 应用程序的卓越性能，而且几乎不会产生任何开发人员成本。

如要分配和释放内存，并获取可在主机和设备代码中引用的指针，请使用 `cudaMallocManaged` 和 `cudaFree` 取代对 `malloc` 和 `free` 的调用，如下例所示：

```
// CPU-only

int N = 2<<20;
size_t size = N * sizeof(int);

int *a;
a = (int *)malloc(size);

// Use `a` in CPU-only program.

free(a);
// Accelerated

int N = 2<<20;
size_t size = N * sizeof(int);

int *a;
// Note the address of `a` is passed as first argument.
cudaMallocManaged(&a, size);

// Use `a` on the CPU and/or on any GPU in the accelerated
system.

cudaFree(a);
```

练习：主机和设备上的数组操作

`01-double-elements.cu` 程序分配一个数组，在主机上使用整数值对其进行初始化，并尝试在GPU上并行执行将每个数组值加倍，然后在主机上确认该加倍操作是否成功。目前该程序无法正常工作：它正在尝试在主机和设备上使用指针 `a` 处的数组进行交互，但分配的该数组（使用 `malloc`）只能在主机上访问。 请重构应用程序以满足下面的条件。如果遇到问题，请参考解决方案：

- `a` 应该对主机和设备代码均可用。
- 应该正确释放a处的内存。

In [ ]:
```
!nvcc -arch=sm_70 -o double-elements 05-allocate/01-double-elements.cu -r
```

网格大小与工作量不匹配

以下幻灯片以可视化的方式概要地介绍了后面将学习的内容。请点击浏览一遍这些幻灯片，然后再继续深入了解以下章节中的主题。

In [ ]:
```
%%HTML

<div align="center"><iframe src="https://view.officeapps.live.com/op/view
```

如何处理块配置与所需线程数不匹配

可能会出现这样的情况，执行配置所创建的线程数无法匹配为实现并行循环所需的线程数。

一个常见的例子与希望选择的最佳线程块大小有关。例如，鉴于 GPU 的硬件特性，所含线程的数量为 32 的倍数的线程块是最理想的选择，因其具备性能上的优势。假设我们要启动一些线程块且每个线程块中均包含 256 个线程（32 的倍数），并需运行 1000 个并行任务（此处使用极小的数量以便于说明），则任何数量的线程块均无法在网格中精确生成 1000 个总线程，因为没有任何整数值在乘以 32 后可以恰好等于 1000。

这个问题可以通过以下方式轻松地解决：

- 编写执行配置，使其创建的线程数**超过**执行分配工作所需的线程数。
- 将一个值作为参数传递到核函数（`N`）中，该值表示要处理的数据集总大小或完成工作所需的总线程数。
- 计算网格内的线程索引后（使用 `threadIdx + blockIdx*blockDim`），请检查该索引是否超过 `N`，并且只在不超过的情况下执行与核函数相关的工作。

以下是编写执行配置的惯用方法示例，适用于 `N` 和线程块中的线程数已知，但无法保证网格中的线程数和 `N` 之间完全匹配的情况。如此一来，便可确保网格中至少始终拥有 `N` 所需的线程数，且超出的线程数至多仅可相当于 1 个线程块的线程数量：

```
// Assume `N` is known
int N = 100000;

// Assume we have a desire to set `threads_per_block` exactly to
`256`
size_t threads_per_block = 256;

// Ensure there are at least `N` threads in the grid, but only 1
block's worth extra
size_t number_of_blocks = (N + threads_per_block - 1) /
threads_per_block;

some_kernel<<<number_of_blocks, threads_per_block>>>(N);
```

由于上述执行配置致使网格中的线程数超过 `N`，因此需要注意 `some_kernel` 定义中的内容，以确保 `some_kernel` 在由其中一个 "额外的" 线程执行时不会尝试访问超出范围的数据元素：

```
__global__ some_kernel(int N)
{
  int idx = threadIdx.x + blockIdx.x * blockDim.x;

  if (idx < N) // Check to make sure `idx` maps to some value
within `N`
  {
    // Only do work if it does
  }
}
```

练习：使用不匹配的执行配置来加速For循环

`02-mismatched-config-loop.cu` 中的程序使用 `cudaMallocManaged` 为包含 1000 个元素的整数数组分配内存，然后试图使用 CUDA 核函数以并行方式初始化数组中的所有值。此程序假设 `N` 和 `threads_per_block` 的数量均为已知。您的任务是完成以下两个目标，如您遇到问题，请参阅 解决方案：

- 为 `number_of_blocks` 分配一个值，以确保线程数至少与指针 `a` 中可供访问的元素数同样多。
- 更新 `initializeElementsTo` 核函数以确保不会尝试访问超出范围的数据元素。

In [ ]:
```
!nvcc -arch=sm_70 -o mismatched-config-loop 05-allocate/02-mismatched-con
```

跨网格的循环

以下幻灯片以可视化的方式概要地介绍了后面将学习的内容。请点击浏览一遍这些幻灯片，然后再继续深入了解以下章节中的主题。

In [ ]:
```
%%HTML

<div align="center"><iframe src="https://view.officeapps.live.com/op/view
```

数据集比网格大

或出于选择，为了要创建具有超高性能的执行配置，或出于需要，一个网格中的线程数量可能会小于数据集的大小。请思考一下包含 1000 个元素的数组和包含 250 个线程的网格（此处使用极小的规模以便于说明）。此网格中的每个线程将需使用 4 次。如要实现此操作，一种常用方法便是在核函数中使用**跨网格循环**。

在跨网格循环中，每个线程将在网格内使用 `threadIdx + blockIdx*blockDim` 计算自身唯一的索引，并对数组内该索引的元素执行相应运算，然后将网格中的线程数添加到索引并重复此操作，直至超出数组范围。例如，对于包含 500 个元素的数组和包含 250 个线程的网格，网格中索引为 20 的线程将执行如下操作：

- 对包含 500 个元素的数组的元素 20 执行相应运算
- 将其索引增加 250，使网格的大小达到 270
- 对包含 500 个元素的数组的元素 270 执行相应运算
- 将其索引增加 250，使网格的大小达到 520
- 由于 520 现已超出数组范围，因此线程将停止工作

CUDA 提供一个可给出网格中线程块数的特殊变量：`gridDim.x`。然后计算网格中的总线程数，即网格中的线程块数乘以每个线程块中的线程数：`gridDim.x * blockDim.x`。带着这样的想法来看看以下核函数中网格跨度循环的详细示例：

```
__global void kernel(int *a, int N)
{
  int indexWithinTheGrid = threadIdx.x + blockIdx.x *
blockDim.x;
  int gridStride = gridDim.x * blockDim.x;

  for (int i = indexWithinTheGrid; i < N; i += gridStride)
  {
    // do work on a[i];
  }
}
```

练习：使用跨网格循环来处理比网格更大的数组

重构 `03-grid-stride-double.cu` 以在 `doubleElements` 核函数中使用网格跨度循环，进而使小于 `N` 的网格可以重用线程以覆盖数组中的每个元素。程序会打印数组中的每个元素是否均已加倍，而当前该程序会准确打印出 `FALSE`。如您遇到问题，请参阅 解决方案。

In [ ]:
```
!nvcc -arch=sm_70 -o grid-stride-double 05-allocate/03-grid-stride-double
```

错误处理

与在任何应用程序中一样，加速 CUDA 代码中的错误处理同样至关重要。即便不是大多数，也有许多 CUDA 函数（例如，内存管理函数）会返回类型为 `cudaError_t` 的值，该值可用于检查调用函数时是否发生错误。以下是对调用 `cudaMallocManaged` 函数执行错误处理的示例：

```
cudaError_t err;
err = cudaMallocManaged(&a, N)                    // Assume the
existence of `a` and `N`.

if (err != cudaSuccess)                           //
`cudaSuccess` is provided by CUDA.
{
  printf("Error: %s\n", cudaGetErrorString(err)); //
`cudaGetErrorString` is provided by CUDA.
}
```

启动定义为返回 `void` 的核函数后，将不会返回类型为 `cudaError_t` 的值。为检查启动核函数时是否发生错误（例如，如果启动配置错误），CUDA 提供 `cudaGetLastError` 函数，该函数会返回类型为 `cudaError_t` 的值。

```
/*
 * This launch should cause an error, but the kernel itself
 * cannot return it.
 */

someKernel<<<1, -1>>>();  // -1 is not a valid number of
threads.

cudaError_t err;
err = cudaGetLastError(); // `cudaGetLastError` will return the
error from above.
if (err != cudaSuccess)
{
  printf("Error: %s\n", cudaGetErrorString(err));
}
```

最后，为捕捉异步错误（例如，在异步核函数执行期间），请务必检查后续同步 CUDA 运行时 API 调用所返回的状态（例如 `cudaDeviceSynchronize`）；如果之前启动的其中一个核函数失败，则将返回错误。

练习：添加错误处理

目前，`01-add-error-handling.cu` 会编译、运行并打印已加倍失败的数组元素。不过，该程序不会指明其中是否存在任何错误。重构应用程序以处理 CUDA 错误，以便您可以了解程序出现的问题并进行有效调试。您将需要调查在调用 CUDA 函数时可能出现的同步错误，以及在执行 CUDA 核函数时可能出现的异步错误。如您遇到问题，请参阅 解决方案。

In [ ]:
```
!nvcc -arch=sm_70 -o add-error-handling 06-errors/01-add-error-handling.c
```

CUDA错误处理功能

创建一个包装 CUDA 函数调用的宏对于检查错误十分有用。以下是一个宏示例，您可以在余下练习中随时使用：

```
#include <stdio.h>
#include <assert.h>

inline cudaError_t checkCuda(cudaError_t result)
{
  if (result != cudaSuccess) {
    fprintf(stderr, "CUDA Runtime Error: %s\n",
cudaGetErrorString(result));
    assert(result == cudaSuccess);
  }
  return result;
}

int main()
{

/*
 * The macro can be wrapped around any function returning
 * a value of type `cudaError_t`.
 */

  checkCuda( cudaDeviceSynchronize() )
}
```

总结

至此，您已经完成以下列出的所有实验学习目标：

- 编写、编译及运行既可调用 CPU 函数也可**启动**GPU**核函数**的 C/C++ 程序。
- 使用**执行配置**控制并行**线程层次结构**。
- 重构串行循环以在 GPU 上并行执行其迭代。
- 分配和释放可用于 CPU 和 GPU 的内存。
- 处理 CUDA 代码生成的错误。

现在，您将完成实验的最终目标：

- 加速 CPU 应用程序。

最后练习：加速向量加法

下面的挑战将使您有机会运用在实验中学到的知识。其中涉及加速 CPU 向量加法程序，尽管该程序不甚复杂，但仍能让您有机会重点运用所学的借助 CUDA 加速 GPU 应用程序的相关知识。完成此练习后，如果您有富余时间并有意深究，可继续学习*高阶内容*部分以了解涉及更复杂代码库的一些挑战。

`01-vector-add.cu` 包含一个可正常运作的 CPU 向量加法应用程序。加速其 `addVectorsInto` 函数，使之在 GPU 上以 CUDA 核函数运行并使其并行执行工作。鉴于需发生以下操作，如您遇到问题，请参阅 解决方案。

- 扩充 `addVectorsInto` 定义，使之成为 CUDA 核函数。
- 选择并使用有效的执行配置，以使 `addVectorsInto` 作为 CUDA 核函数启动。
- 更新内存分配，内存释放以反映主机和设备代码需要访问 3 个向量：`a`、`b` 和 `result`。
- 重构 `addVectorsInto` 的主体：将在单个线程内部启动，并且只需对输入向量执行单线程操作。确保线程从不尝试访问输入向量范围之外的元素，并注意线程是否需对输入向量的多个元素执行操作。
- 在 CUDA 代码中的可能会运行失败的位置添加错误处理（否则该失败就无法察觉）。

In [ ]:
```
!nvcc -arch=sm_70 -o vector-add 07-vector-add/01-vector-add.cu -run
```

进阶内容

以下练习为时间富余且有意深究的学习者提供额外挑战。这些挑战需要使用更先进的技术加以应对，并且我们没有提供很多的框架代码。因此，完成这些挑战着实不易，但您在此过程中亦会收获重大进步。

2维和3维的网格和块

可以将网格和线程块定义为最多具有 3 个维度。使用多个维度定义网格和线程块绝不会对其性能造成任何影响，但这在处理具有多个维度的数据时可能非常有用，例如 2D 矩阵。如要定义二维或三维网格或线程块，可以使用 CUDA 的 `dim3` 类型，即如下所示：

```
dim3 threads_per_block(16, 16, 1);
dim3 number_of_blocks(16, 16, 1);
someKernel<<<number_of_blocks, threads_per_block>>>();
```

鉴于以上示例，`someKernel` 内部的变量 `gridDim.x`、`gridDim.y`、`blockDim.x` 和 `blockDim.y` 均将等于 `16`。

练习：加速2D矩阵乘法应用

文件 `01-matrix-multiply-2d.cu` 包含一个功能齐全的主机函数 `matrixMulCPU`。您的任务是扩建 CUDA 核函数 `matrixMulGPU`。源代码将使用这两个函数执行矩阵乘法，并比较它们的答案以验证您编写的 CUDA 核函数是否正确。您使用以下指南获得编程支持，如您遇到问题，请参阅 解决方案：

- 您将需创建执行配置，其参数均为 `dim3` 值，且 `x` 和 `y` 维度均设为大于 `1`。
- 在核函数主体内部，您将需要按照惯例在网格内建立所运行线程的唯一索引，但应为线程建立两个索引：一个用于网格的 x 轴，另一个用于网格的 y 轴。

In [ ]:
```
!nvcc -arch=sm_70 -o matrix-multiply-2d 08-matrix-multiply/01-matrix-mult
```

练习：给热传导应用程序加速

在下面的练习中，您将为模拟金属银二维热传导的应用程序执行加速操作。

将 `01-heat-conduction.cu` 内的 `step_kernel_mod` 函数转换为在 GPU 上执行，并修改 `main` 函数以恰当分配在 CPU 和 GPU 上使用的数据。`step_kernel_ref` 函数在 CPU 上执行并用于检查错误。由于此代码涉及浮点计算，因此不同的处理器甚或同一处理器上的简单重排操作都可能导致结果略有出入。为此，错误检查代码会使用错误阈值，而非查找完全匹配。如您遇到问题，请参阅 解决方案。

In [ ]:
```
!nvcc -arch=sm_70 -o heat-conduction 09-heat/01-heat-conduction.cu -run
```

> 此任务中的原始热传导 CPU 源代码取自于休斯顿大学的文章 An OpenACC Example Code for a C-based heat conduction code（基于 C 的热传导代码的 OpenACC 示例代码）。