使用 CUDA C/C++ 统一内存和 Nsight Systems (nsys) 管理加速应用程序内存

对于本实验和其他 CUDA 基础实验，我们强烈建议您遵循 *CUDA 最佳实践指南*，其中推荐一种称为 **APOD** 的设计周期：**评估**、**并行化**、**优化**和**部署**。简言之，APOD 规定一个迭代设计过程，开发人员能够在该过程中对其加速应用程序性能施以渐进式改进，并发布代码。随着开发人员的 CUDA 编程能力愈渐增强，他们已能在加速代码库中应用更先进的优化技术。

本实验将支持这种迭代开发风格。您将使用 **Nsight Systems命令行分析器**定性衡量应用程序性能及确定优化机会，之后您将应用渐进式改进，最后您会学习新技术并重复该周期。需重点关注的是，您将在本实验中学习及应用的众多技术均会涉及 CUDA **统一内存**工作原理的具体细节。理解统一内存行为是 CUDA 开发人员的一项基本技能，同时也可作为多项更先进内存管理技术的先决条件。

预备知识

如要充分利用本实验，您应已能胜任如下任务：

- 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。
- 分配和释放统一内存。

学习目标

当您在本实验完成学习后，您将能够：

- 使用 **Nsight Systems命令行分析器** (**nsys**) 分析被加速的应用程序的性能。
- 利用对**流多处理器**的理解优化执行配置。
- 理解**统一内存**在页错误和数据迁移方面的行为。
- 使用**异步内存预取**减少页错误和数据迁移以提高性能。
- 采用循环式的迭代开发加快应用程序的优化加速和部署。

使用nsys性能分析器帮助应用程序迭代地进行优化

如要确保优化加速代码库的尝试真正取得成功，唯一方法便是分析应用程序以获取有关其性能的定量信息。`nsys` 是指 NVIDIA 的Nsight System命令行分析器。该分析器附带于CUDA工具包中，提供分析被加速的应用程序性能的强大功能。

`nsys` 使用起来十分简单，最基本用法是向其传递使用 `nvcc` 编译的可执行文件的路径。随后 `nsys` 会继续执行应用程序，并在此之后打印应用程序 GPU 活动的摘要输出、CUDA API 调用以及**统一内存**活动的相关信息。我们稍后会在本实验中详细介绍这一主题。

在加速应用程序或优化已经加速的应用程序时，我们应该采用科学的迭代方法。作出更改后需分析应用程序、做好记录并记录任何重构可能会对性能造成何种影响。尽早且经常进行此类观察通常会让您轻松获得足够的性能提升，以助您发布加速应用程序。此外，经常分析应用程序将使您了解到对 CUDA 代码库作出的特定更改会对其实际性能造成何种影响：而当只在代码库中进行多种更改后再分析应用程序时，将很难得知这一点。

练习：使用nsys分析应用程序

01-vector-add.cu（<------您可点击打开此文件链接和本实验中的任何源文件链接并进行编辑）是一个简单易用的加速向量加法程序。使用下方两个代码执行单元（按住 `CTRL` 并点击即可）。第一个代码执行单元将编译（及运行）向量加法程序。第二个代码执行单元将运用 `nsys profile` 分析刚编译好的可执行文件。

`nsys profile` 将生成一个 `qdrep` 报告文件，该文件可以以多种方式使用。 我们在这里使用 `--stats = true` 标志表示我们希望打印输出摘要统计信息。 输出的信息有很多，包括：

- 配置文件配置详细信息
- 报告文件的生成详细信息
- **CUDA API统计信息**
- **CUDA核函数的统计信息**
- **CUDA内存操作统计信息（时间和大小）**
- 操作系统内核调用接口的统计信息

在本实验中，您将主要使用上面**黑体字**的3个部分。 在下一个实验中，您将使用生成的报告文件将其提供给Nsight Systems 进行可视化分析。

应用程序分析完毕后，请使用分析输出中显示的信息回答下列问题：

- 此应用程序中唯一调用的 CUDA 核函数的名称是什么？
- 此核函数运行了多少次？
- 此核函数的运行时间为多少？请记录下此时间：您将继续优化此应用程序，再和此时间做对比。

In [ ]:
```
!nvcc -o single-thread-vector-add 01-vector-add/01-vector-add.cu -run
```

In [ ]:
```
!nsys profile --stats=true ./single-thread-vector-add
```

值得一提的是，默认情况下， `nsys profile` 不会覆盖现有的报告文件。 这样做是为了防止在进行概要分析时意外丢失工作。 如果出于某种原因，您宁愿覆盖现有的报告文件，例如在快速迭代期间，可以向 `nsys profile` 提供 `-f` 标志以允许覆盖现有的报告文件。

练习：优化并分析性能

请抽出一到两分钟时间，更新 01-vector-add.cu 的执行配置以对其进行简单的优化，以便其能在单个线程块中的多个线程上运行。请使用下方的代码执行单元重新编译并借助 `nsys profile` 进行分析。使用性能分析的输出检查核函数的运行时间。此次优化带来多大的速度提升？请务必在某处记录下您的结果。

In [ ]:
```
!nvcc -o multi-thread-vector-add 01-vector-add/01-vector-add.cu -run
```

In [ ]:
```
!nsys profile --stats=true ./multi-thread-vector-add
```

练习：迭代优化

在本练习中，您将进行多轮周期式的迭代优化，具体包括：编辑 01-vector-add.cu 的执行配置、开展性能分析及记录结果以查看影响。开展操作时请依循以下指南：

- 首先列出您将用于更新执行配置的 3 至 5 种不同方法，确保涵盖一系列不同的网格和线程块大小组合。
- 使用所列的其中一种方法编辑 01-vector-add.cu 程序。
- 使用下方的两个代码执行单元编译和分析更新后的代码。
- 记录核函数执行的运行时间，应与性能分析输出中给出的时间一样。
- 对以上列出的每个可能实现的优化重复执行编辑代码/性能分析/记录结果

在您尝试的执行配置中，哪个经证明是最快的?

In [ ]:
```
!nvcc -o iteratively-optimized-vector-add 01-vector-add/01-vector-add.cu
```

In [ ]:
```
!nsys profile --stats=true ./iteratively-optimized-vector-add
```

流多处理器（Streaming Multiprocessors）及查询GPU的设备配置

本节将探讨了解 GPU 硬件的特定功能，以进一步促进优化。学习完**流多处理器**后，您将尝试进一步优化自己一直执行的加速向量加法程序。

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。 在更详细地学习这些内容之前，请单击每一页幻灯片直至结束。

In [ ]:
```
%%HTML

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

流多处理器和Warps

运行 CUDA 应用程序的 GPU 具有称为**流多处理器**（或 **SM**）的处理单元。在核函数执行期间，将线程块提供给 SM 以供其执行。为支持 GPU 执行尽可能多的并行操作，您通常可以*选择线程块数量数倍于指定 GPU 上 SM 数量的网格大小*来提升性能。

此外，SM 会在一个名为**warp**的线程块内创建、管理、调度和执行包含 32 个线程的线程组。本课程将不会更深入探讨 SM 和warp，但值得注意的是，您也可*选择线程数量数倍于 32 的线程块大小*来提升性能。

以编程方式查询GPU设备属性

由于 GPU 上的 SM 数量会因所用的特定 GPU 而异，因此为支持可移植性，您不得将 SM 数量硬编码到代码库中。相反，应该以编程方式获取此信息。

以下所示为在 CUDA C/C++ 中获取 C 结构的方法，该结构包含当前处于活动状态的 GPU 设备的多个属性，其中包括设备的 SM 数量：

```
int deviceId;
cudaGetDevice(&deviceId);                  // `deviceId` now
points to the id of the currently active GPU.

cudaDeviceProp props;
cudaGetDeviceProperties(&props, deviceId); // `props` now has
many useful properties about
                                           // the active GPU
device.
```

练习: 查询设备信息

目前，`01-get-device-properties.cu` 包含众多未分配的变量，并将打印一些无用信息，这些信息用于描述当前处于活动状态的 GPU 设备的详细信息。

构建 `01-get-device-properties.cu` 以打印源代码中指示的所需设备属性的实际值。为获取操作支持并查看相关介绍，请参阅 CUDA 运行时文档 以帮助识别设备属性结构中的相关属性。如您遇到问题，请参阅 解决方案。

In [ ]:
```
!nvcc -o get-device-properties 04-device-properties/01-get-device-propert
```

练习：将网格数调整为SM数，进一步优化矢量加法

通过查询设备的 SM 数量重构您一直在 01-vector-add.cu 内执行的 `addVectorsInto` 核函数，以便其启动时的网格包含数倍于设备上 SM 数量的线程块数。

根据您所编写代码中的其他特定详细信息，此重构可能会或不会提高或大幅改善核函数的性能。因此，请务必始终使用 `nsys profile`，以便定量评估性能变化。根据分析输出，记录目前所得结果和其他发现。

In [ ]:
```
!nvcc -o sm-optimized-vector-add 01-vector-add/01-vector-add.cu -run
```

In [ ]:
```
!nsys profile --stats=true ./sm-optimized-vector-add
```

获得统一内存的细节

您一直使用 `cudaMallocManaged` 分配旨在供主机或设备代码使用的内存，并且现在仍在享受这种方法的便利之处，即在实现自动内存迁移且简化编程的同时，而无需深入了解 `cudaMallocManaged` 所分配**统一内存** (**UM**) 实际工作原理的详细信息。 `nsys profile` 提供有关加速应用程序中 UM 管理的详细信息，并在利用这些信息的同时结合对 UM 工作原理的更深入理解，进而为优化加速应用程序创造更多机会。

以下幻灯片将直观呈现即将发布的材料的概要信息。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
```

统一内存(UM)的迁移

分配 UM 时，内存尚未驻留在主机或设备上。主机或设备尝试访问内存时会发生 页错误，此时主机或设备会批量迁移所需的数据。同理，当 CPU 或加速系统中的任何 GPU 尝试访问尚未驻留在其上的内存时，会发生页错误并触发迁移。

能够执行页错误并按需迁移内存对于在加速应用程序中简化开发流程大有助益。此外，在处理展示稀疏访问模式的数据时（例如，在应用程序实际运行之前无法得知需要处理的数据时），以及在具有多个 GPU 的加速系统中，数据可能由多个 GPU 设备访问时，按需迁移内存将会带来显著优势。

有些情况下（例如，在运行时之前需要得知数据，以及需要大量连续的内存块时），我们还能有效规避页错误和按需数据迁移所产生的开销。

本实验的后续内容将侧重于对按需迁移的理解，以及如何在分析器输出中识别按需迁移。这些知识可让您在享受按需迁移优势的同时，减少其产生的开销。

练习：探索统一内存（UM）的页错误

`nsys profile` 会提供描述所分析应用程序 UM 行为的输出。在本练习中，您将对一个简单的应用程序做出一些修改，并会在每次更改后利用 `nsys profile` 的统一内存输出部分，探讨 UM 数据迁移的行为方式。

`01-page-faults.cu` 包含 `hostFunction` 和 `gpuKernel` 函数，我们可以通过这两个函数并使用数字 `1` 初始化 `2<<24` 个单元向量的元素。主机函数和 GPU 核函数目前均未使用。

对于以下 4 个问题中的每一问题，请根据您对 UM 行为的理解，首先假设应会发生何种页错误，然后使用代码库中所提供 2 个函数中的其中一个或同时使用这两个函数编辑 `01-page-faults.cu` 以创建场景，以便您测试假设。

为了检验您的假设，请使用下面的代码执行单元来编译和分析代码。 一定要记录从 `nsys profile --stats = true` 输出中获得的假设以及结果。 在 `nsys profile --stats = true` 的输出中，您应该查找以下内容：

- 输出中是否有 *CUDA内存操作统计信息* 部分？
- 如果是，这是否表示数据从主机到设备（HtoD）或从设备到主机（DtoH）的迁移？
- 进行迁移时，输出如何说明有多少个“操作”？ 如果看到许多小的内存迁移操作，则表明按需出现页面错误，并且每次在请求的位置出现页面错误时都会发生小内存迁移。

以下是供您探索的方案，以及遇到困难时的解决方案：

- 当仅通过CPU访问统一内存时，是否存在内存迁移和/或页面错误的证据？ （解决方案）
- 当仅通过GPU访问统一内存时，是否有证据表明内存迁移和/或页面错误？ （解决方案）
- 当先由CPU然后由GPU访问统一内存时，是否有证据表明存在内存迁移和/或页面错误？ （解决方案）
- 当先由GPU然后由CPU访问统一内存时，是否存在内存迁移和/或页面错误的证据？ （解决方案）

In [ ]:
```
!nvcc -o page-faults 06-unified-memory-page-faults/01-page-faults.cu -run
```

In [ ]:
```
!nsys profile --stats=true ./page-faults
```

练习：重新审视矢量加法程序的UM行为

返回您一直在本实验中执行的 01-vector-add.cu 程序，查看当前状态的代码库，并假设您预期会发生哪种类型的内存迁移和/或页面错误。 查看最后一次重构的概要分析输出（通过向上滚动查找输出或通过执行下面的代码执行单元），并观察性能分析器输出的 *CUDA内存操作统计信息* 部分。 您能否根据代码库的内容解释迁移的种类及其操作的数量？

In [ ]:
```
!nsys profile --stats=true ./sm-optimized-vector-add
```

练习：在核函数中初始化向量

当 `nsys profile` 给出核函数所需的执行时间时，则在此函数执行期间发生的主机到设备页错误和数据迁移都会包含在所显示的执行时间中。

带着这样的想法来将 01-vector-add.cu 程序中的 `initWith` 主机函数重构为 CUDA 核函数，以便在 GPU 上并行初始化所分配的向量。成功编译及运行重构的应用程序后，但在对其进行分析之前，请假设如下内容：

- 您期望重构会对 UM 页错误行为产生何种影响?
- 您期望重构会对所报告的 `addVectorsInto` 运行时产生何种影响?

请再次记录结果。如您遇到问题，请参阅 解决方案。

In [ ]:
```
!nvcc -o initialize-in-kernel 01-vector-add/01-vector-add.cu -run
```

In [ ]:
```
!nsys profile --stats=true ./initialize-in-kernel
```

异步内存预取

在主机到设备和设备到主机的内存传输过程中，我们使用一种技术来减少页错误和按需内存迁移成本，此强大技术称为**异步内存预取**。通过此技术，程序员可以在应用程序代码使用统一内存 (UM) 之前，在后台将其异步迁移至系统中的任何 CPU 或 GPU 设备。此举可以减少页错误和按需数据迁移所带来的成本，并进而提高 GPU 核函数和 CPU 函数的性能。

此外，预取往往会以更大的数据块来迁移数据，因此其迁移次数要低于按需迁移。此技术非常适用于以下情况：在运行时之前已知数据访问需求且数据访问并未采用稀疏模式。

CUDA 可通过 `cudaMemPrefetchAsync` 函数，轻松将托管内存异步预取到 GPU 设备或 CPU。以下所示为如何使用该函数将数据预取到当前处于活动状态的 GPU 设备，然后再预取到 CPU：

```
int deviceId;
cudaGetDevice(&deviceId);
// The ID of the currently active GPU device.

cudaMemPrefetchAsync(pointerToSomeUMData, size, deviceId);
// Prefetch to GPU device.
cudaMemPrefetchAsync(pointerToSomeUMData, size,
cudaCpuDeviceId); // Prefetch to host. `cudaCpuDeviceId` is a

// built-in CUDA variable.
```

练习：异步内存预取

此时，实验中的 01-vector-add.cu 程序不仅应启动 CUDA 核函数以将 2 个向量添加到第三个结果向量（所有向量均通过 `cudaMallocManaged` 函数进行分配），还应在 CUDA 核函数中并行初始化其中的每个向量。如果某种原因导致应用程序不执行上述任何操作，则请参阅以下 参考应用程序，并更新自己的代码库以反映其当前功能。

在 01-vector-add.cu 应用程序中使用 `cudaMemPrefetchAsync` 函数开展 3 个实验，以探究其会对页错误和内存迁移产生何种影响。

- 当您将其中一个初始化向量预取到主机时会出现什么情况?
- 当您将其中两个初始化向量预取到主机时会出现什么情况?
- 当您将三个初始化向量全部预取到主机时会出现什么情况?

在进行每个实验之前，请先假设 UM 的行为表现（尤其就页错误而言），以及其对所报告的初始化核函数运行时会产生何种影响，然后运行 `nsys profile` 进行验证。如您遇到问题，请参阅 解决方案。

In [ ]:
```
!nvcc -o prefetch-to-gpu 01-vector-add/01-vector-add.cu -run
```

In [ ]:
```
!nsys profile --stats=true ./prefetch-to-gpu
```

练习：将内存预取回CPU

请为该函数添加额外的内存预取回 CPU，以验证 `addVectorInto` 核函数的正确性。然后再次假设 UM 所受影响，并在 `nsys profile` 中进行分析确认。如您遇到问题，请参阅 解决方案。

In [ ]:
```
!nvcc -o prefetch-to-cpu 01-vector-add/01-vector-add.cu -run
```

In [ ]:
```
!nsys profile --stats=true ./prefetch-to-cpu
```

在使用异步预取进行了一系列重构之后，您应该看到内存传输次数减少了，但是每次传输的量增加了，并且内核执行时间大大减少了。

总结

此时，您在实验中能够执行以下操作：

- 使用 **NVIDIA 命令行分析器 (nsys)** 分析加速应用程序性能。
- 利用对**流多处理器**的理解优化执行配置。
- 理解**统一内存**在页错误和数据迁移方面的行为。
- 使用**异步内存预取**减少页错误和数据迁移以提高性能。
- 采用迭代开发周期快速加速和部署应用程序。

为巩固您的学习成果，并加强您通过迭代方式加速、优化及部署应用程序的能力，请继续完成本实验的最后一个练习。完成后，时间富余并有意深究的学习者可以继续学习*高阶内容*部分。

最后的练习：迭代优化加速的SAXPY应用程序

此处 为您提供一个基本的 SAXPY 加速应用程序。该程序目前包含一些您需要找到并修复的错误，在此之后您才能使用 `nsys profile` 成功对其进行编译、运行和分析。

在修复完错误并对应用程序进行分析后，您需记录 `saxpy` 核函数的运行时，然后采用*迭代方式*优化应用程序，并在每次迭代后使用 `nsys profile` 进行分析验证，以便了解代码更改对核函数性能和 UM 行为产生的影响。

请充分运用本实验提供的各项技术。为获得更好的学习效果，请尽可能利用 effortful retrieval 提供的方法回想已经学过的内容，而不要急于在本课程开始之初就查阅技术细节。

您的最终目标是在不修改 `N` 的情况下分析准确的 `saxpy` 核函数，以便在 *100us* 内运行。如您遇到问题，请参阅 解决方案，您亦可随时对其进行编译和分析。

In [ ]:
```
!nvcc -o saxpy 09-saxpy/01-saxpy.cu -run
```

In [ ]:
```
!nsys profile --stats=true ./saxpy
```