南 开 大 学

NanKai University

CUDA 编程练习

姓名：文静静

学号：1811507

年级：2018级

专业：计算机科学与技术

2021年6月14日

摘要

本次实验主要对英伟达网站上的课程加速计算基础—— CUDA C/C++ 的内容进行了实验，完成了所有练习，学习了 GPU 加速、CUDA 的内存管理、CUDA 流以及通过可视化分析工具来进行检查。

关键字：CUDA　GPU　内存管理　流　可视化工具

目录

1 绪论

本文详细描述了加速计算基础—— CUDA C/C++ 课程内容以及练习实验，学习编译 GPU 核函数、配置线程块和线程数、分配和释放 GPU 的内存、CUDA 错误处理，并学习使用 Nsight Systems 命令行分析器分析被加速的应用程序，针对流处理器优化执行配置，使用异步内存预取减少页错误和数据迁移以提高性能，最后通过可视化分析工具对异步流进行分析。

2 实验选题

CUDA 计算平台能够加速原仅适用于 CPU 的应用程序，从而能在世界上最快的大规模并行 GPU 上运行。通过英伟达网站上的课程：**加速计算基础—— CUDA C/C++** 可以完成以下任务：

1. 利用可在 GPU 上实现的潜在的并行性来加速原仅用于 CPU 的应用程序；
2. 利用基本的 CUDA 内存管理技术来进一步优化被加速的应用程序；
3. 挖掘被加速的应用程序的并发潜力，并利用 CUDA 流加以利用；
4. 利用命令行及可视化的分析工具来指导和检查以上工作。

3 实验内容

3.1 使用 CUDA C/C++ 加速应用程序

1. 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。

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 的信息如下。

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.33.01    Driver Version: 440.33.01    CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla T4            On   | 00000000:00:1E.0 Off |                    0 |
| N/A   29C    P8     9W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

练习：编写一个Hello GPU核函数

```
#include <stdio.h>
void helloCPU()
{
    printf("Hello from the CPU.\n");
}

/*
* The addition of __global__ signifies that this function
* should be launced on the GPU.
*/
__global__ void helloGPU()
{
    printf("Hello from the GPU.\n");
}

int main()
{
    helloCPU();

    /*
    * Add an execution configuration with the <<<...>>> syntax
    * will launch this function as a kernel on the GPU.
    */
    helloGPU<<<1, 1>>>();

    /*
    * cudaDeviceSynchronize  will block the CPU stream until
    * all GPU kernels have completed.
    */
    cudaDeviceSynchronize();
}
```

```
In [2]: !nvcc -arch=sm_70 -o hello-gpu 01-hello/01-hello-gpu.cu -run

Hello from the CPU.
Hello from the GPU!
```

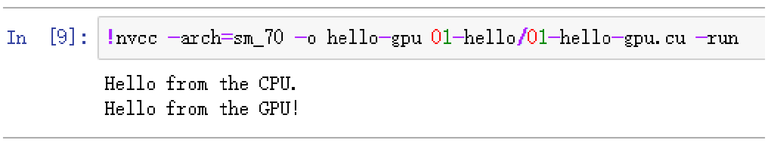
__global__ 关键字表明以下函数将在 GPU 上运行并可全局调用，而在此种情况下，则指由 CPU 或 GPU 调用。通常，我们将在 CPU 上执行的代码称为主机代码，而将在 GPU 上运行的代码称为设备代码。注意返回类型为 void。使用 __global__ 关键字定义的函数需要返回 void 类型。

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。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。

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。

重构以便 Hello from the GPU 在 Hello from the CPU 之前打印：

```
int main()
{
    helloGPU<<<1, 1>>>();
    cudaDeviceSynchronize();
    helloCPU();
}
```



重构以便 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

```
int main()
{
    helloGPU<<<1, 1>>>();
    cudaDeviceSynchronize();
```

```
    helloCPU();
    helloGPU<<<1, 1>>>();
    cudaDeviceSynchronize();
}
```

```
In [13]: !nvcc -arch=sm_70 -o hello-gpu 01-hello/01-hello-gpu.cu -run
Hello from the GPU!
Hello from the CPU.
Hello from the GPU!
```

2. 使用执行配置控制并行线程层次结构。

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。

练习：启动并行运行的核函数

重构 firstParallel 函数以便在 GPU 上作为 CUDA 核函数启动。

```
#include <stdio.h>

__global__ void firstParallel()
{
    printf("This is running in parallel.\n");
}

int main()
{
    firstParallel<<<1, 1>>>();
    cudaDeviceSynchronize();
}
```

```
In [14]: !nvcc -arch=sm_70 -o basic-parallel 02-first-parallel/01-basic-parallel.cu -run
This is running in parallel.
```

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。

```
#include <stdio.h>

__global__ void firstParallel()
{
    printf("This is running in parallel.\n");
}

int main()
{
    firstParallel<<<1, 5>>>();
    cudaDeviceSynchronize();
}
```

```
In [15]: !nvcc -arch=sm_70 -o basic-parallel 02-first-parallel/01-basic-parallel.cu -run
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
```

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）。

```
#include <stdio.h>

__global__ void firstParallel()
{
    printf("This is running in parallel.\n");
}

int main()
{
    firstParallel<<<5, 5>>>();
    cudaDeviceSynchronize();
}
```

```
In [16]: !nvcc -arch=sm_70 -o basic-parallel 02-first-parallel/01-basic-parallel.cu -run
```

```
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
This is running in parallel.
```

练习：使用特定的线程和块索引

根据核函数内的输出可知，核函数的线程数最少为1024，线程块最少为256，因此更新执行配置。

```
#include <stdio.h>
__global__ void printSuccessForCorrectExecutionConfiguration()
{
    if(threadIdx.x == 1023 && blockIdx.x == 255)
    {
        printf("Success!\n");
    }
}
int main()
{
    printSuccessForCorrectExecutionConfiguration<<<256, 1024>>>();
    cudaDeviceSynchronize();
}
```

```
In [32]: !nvcc -arch=sm_70 -o thread-and-block-idx 03-indices/01-thread-and-block-idx.cu -run
```

```
Success!
```

3. 重构串行循环以在 GPU 上并行执行迭代。

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。为此我们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

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。

```
#include <stdio.h>

/*
* Refactor loop to be a CUDA Kernel. The new kernel should
* only do the work of 1 iteration of the original loop.
*/

void loop(int N)
{
    for (int i = 0; i < N; ++i)
    {
        printf("This is iteration number %d\n", i);
    }
}

__global__ void loop_gpu()
{
    /*
    * This kernel does the work of only 1 iteration
    * of the original for loop. Indication of which
    * "iteration" is being executed by this kernel is
    * still available via threadIdx.x.
    */

    printf("This is gpu iteration number %d\n", threadIdx.x);
```

```
}

int main()
{
    /*
    * When refactoring loop to launch as a kernel, be sure
    * to use the execution configuration to control how many
    * "iterations" to perform.
    *
    * For this exercise, only use 1 block of threads.
    */

    int N = 10;
    loop(N);
    loop_gpu<<<1, N>>>();
    cudaDeviceSynchronize();
}
```

对比如下：

```
In [33]: !nvcc -arch=sm_70 -o single-block-loop 04-loops/01-single-block-loop.cu -run

This is iteration number 0
This is iteration number 1
This is iteration number 2
This is iteration number 3
This is iteration number 4
This is iteration number 5
This is iteration number 6
This is iteration number 7
This is iteration number 8
This is iteration number 9
This is gpu iteration number 0
This is gpu iteration number 1
This is gpu iteration number 2
This is gpu iteration number 3
This is gpu iteration number 4
This is gpu iteration number 5
This is gpu iteration number 6
This is gpu iteration number 7
This is gpu iteration number 8
This is gpu iteration number 9
```

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 + block-

Idx.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。

练习：加速具有多个线程块的For循环

使用10个线程块，每个线程块包含1个线程：

```
#include <stdio.h>

/*
 * Refactor loop to be a CUDA Kernel. The new kernel should
 * only do the work of 1 iteration of the original loop.
 */

void loop(int N)
{
    for (int i = 0; i < N; ++i)
    {
        printf("This is iteration number %d\n", i);
    }
}

__global__ void loop_gpu(){

    int time_loop = blockIdx.x;

    printf("This is gpu iteration number %d\n", time_loop);
}

int main()
{
    /*
     * When refactoring loop to launch as a kernel, be sure
     * to use the execution configuration to control how many
     * "iterations" to perform.
     *
     * For this exercise, be sure to use more than 1 block in
     * the execution configuration.
     */

```

```
    int N = 10;
    loop(N);
    loop_gpu<<<N, 1>>>();
    cudaDeviceSynchronize();

}
```

In [35]:
```
!nvcc -arch=sm_70 -o multi-block-loop 04-loops/02-multi-block-loop.cu -run
```

```
This is iteration number 0
This is iteration number 1
This is iteration number 2
This is iteration number 3
This is iteration number 4
This is iteration number 5
This is iteration number 6
This is iteration number 7
This is iteration number 8
This is iteration number 9
This is gpu iteration number 2
This is gpu iteration number 7
This is gpu iteration number 0
This is gpu iteration number 5
This is gpu iteration number 3
This is gpu iteration number 8
This is gpu iteration number 1
This is gpu iteration number 6
This is gpu iteration number 4
This is gpu iteration number 9
```

4. 分配和释放可用于 CPU 和 GPU 的内存。

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。

练习：主机和设备上的数组操作

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。请重构应用程序以使得 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。

```
#include <stdio.h>

void init(int *a, int N)
{
    int i;
```

```
    for (i = 0; i < N; ++i)
    {
        a[i] = i;
    }
}

__global__
void doubleElements(int *a, int N)
{
    int i;
    i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N)
    {
        a[i] *= 2;
    }
}

bool checkElementsAreDoubled(int *a, int N)
{
    int i;
    for (i = 0; i < N; ++i)
    {
        if (a[i] != i*2) return false;
    }
    return true;
}

int main()
{
    int N = 1000;
    int *a;

    size_t size = N * sizeof(int);

    /*
    * Use cudaMallocManaged to allocate pointer a available
    * on both the host and the device.
    */
```

```

    cudaMallocManaged(&a, size);

    init(a, N);

    size_t threads_per_block = 256;
    size_t number_of_blocks = (N + threads_per_block - 1) /
        threads_per_block;

    doubleElements<<<number_of_blocks, threads_per_block>>>(a, N);
    cudaDeviceSynchronize();

    bool areDoubled = checkElementsAreDoubled(a, N);
    printf("All elements were doubled? %s\n", areDoubled ? "TRUE" : "
        FALSE");

    /*
    * Use cudaFree to free memory allocated
    * with cudaMallocManaged.
    */

    cudaFree(a);
}
```

```
In [43]: !nvcc -arch=sm_70 -o double-elements 05-allocate/01-double-elements.cu -run
         All elements were doubled? TRUE
```

当块配置与所需线程数不匹配时，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。

练习：使用不匹配的执行配置来加速For循环

```
#include <stdio.h>

```

```
__global__ void initializeElementsTo(int initialValue, int *a, int N)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < N)
    {
        a[i] = initialValue;
    }
}

int main()
{

    int N = 1000;

    int *a;
    size_t size = N * sizeof(int);

    cudaMallocManaged(&a, size);

    size_t threads_per_block = 256;

    /*
    * The following is idiomatic CUDA to make sure there are at
    * least as many threads in the grid as there are N elements.
    */

    size_t number_of_blocks = (N + threads_per_block - 1) /
        threads_per_block;

    int initialValue = 6;

    initializeElementsTo<<<number_of_blocks, threads_per_block>>>(
        initialValue, a, N);
    cudaDeviceSynchronize();

    /*
    * Check to make sure all values in a, were initialized.
    */
```

```
    for (int i = 0; i < N; ++i)
    {
        if(a[i] != initialValue)
        {
            printf("FAILURE: target value: %d\t a[%d]: %d\n",
               initialValue, i, a[i]);
            exit(1);
        }
    }
    printf("SUCCESS!\n");

    cudaFree(a);
}
```

```
In [46]: !nvcc -arch=sm_70 -o mismatched-config-loop 05-allocate/02-mismatched-config-loop.cu -run
SUCCESS!
```

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。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。

练习：使用跨网格循环来处理比网格更大的数组

```
#include <stdio.h>

void init(int *a, int N)
{
    int i;
    for (i = 0; i < N; ++i)
    {
        a[i] = i;
    }
}

__global__
void doubleElements(int *a, int N)
```

```
{

    /*
    * Use a grid-stride loop so each thread does work
    * on more than one element in the array.
    */

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = gridDim.x * blockDim.x;

    for (int i = idx; i < N; i += stride)
    {
        a[i] *= 2;
    }
}

bool checkElementsAreDoubled(int *a, int N)
{
    int i;
    for (i = 0; i < N; ++i)
    {
        if (a[i] != i*2) return false;
    }
    return true;
}

int main()
{
    int N = 10000;
    int *a;

    size_t size = N * sizeof(int);
    cudaMallocManaged(&a, size);

    init(a, N);

    size_t threads_per_block = 256;
    size_t number_of_blocks = 32;
```

```

    doubleElements<<<number_of_blocks, threads_per_block>>>(a, N);
    cudaDeviceSynchronize();

    bool areDoubled = checkElementsAreDoubled(a, N);
    printf("All elements were doubled? %s\n", areDoubled ? "TRUE" : "
      FALSE");

    cudaFree(a);
}
```

```
In [51]: !nvcc -arch=sm_70 -o grid-stride-double 05-allocate/03-grid-stride-double.cu -run
All elements were doubled? TRUE
```

5. 处理 CUDA 代码生成的错误。

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。

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。

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

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。

```
#include <stdio.h>

void init(int *a, int N)
{
    int i;
```

```
    for (i = 0; i < N; ++i)
    {
        a[i] = i;
    }
}

__global__
void doubleElements(int *a, int N)
{

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = gridDim.x * blockDim.x;

    /*
    * The previous code (now commented out) attempted
    * to access an element outside the range of a.
    */

    // for (int i = idx; i < N + stride; i += stride)
    for (int i = idx; i < N; i += stride)
    {
        a[i] *= 2;
    }
}

bool checkElementsAreDoubled(int *a, int N)
{
    int i;
    for (i = 0; i < N; ++i)
    {
        if (a[i] != i*2) return false;
    }
    return true;
}

int main()
{
    int N = 10000;
```

```
  int *a;

  size_t size = N * sizeof(int);
  cudaMallocManaged(&a, size);

  init(a, N);

  /*
  * The previous code (now commented out) attempted to launch
  * the kernel with more than the maximum number of threads per
  * block, which is 1024.
  */

  size_t threads_per_block = 1024;
  /* size_t threads_per_block = 2048; */
  size_t number_of_blocks = 32;

  cudaError_t syncErr, asyncErr;

  doubleElements<<<number_of_blocks, threads_per_block>>>(a, N);

  /*
  * Catch errors for both the kernel launch above and any
  * errors that occur during the asynchronous doubleElements
  * kernel execution.
  */

  syncErr = cudaGetLastError();
  asyncErr = cudaDeviceSynchronize();

  /*
  * Print errors should they exist.
  */

  if (syncErr != cudaSuccess) printf("Error: %s\n", cudaGetErrorString(syncErr));
  if (asyncErr != cudaSuccess) printf("Error: %s\n", cudaGetErrorString(asyncErr));
```

```
    
    bool areDoubled = checkElementsAreDoubled(a, N);
    printf("All elements were doubled? %s\n", areDoubled ? "TRUE" : "
      FALSE");

    cudaFree(a);
}
```

可以通过之前提到了三种方法得到错误提示，来改正程序中的错误：

```
In [59]: !nvcc -arch=sm_70 -o add-error-handling 06-errors/01-add-error-handling.cu -run

Error: invalid configuration argument
All elements were doubled? FALSE
```

这是因为一个线程块最多只能有1024个线程，改正错误，得到如下运行结果：

```
In [60]: !nvcc -arch=sm_70 -o add-error-handling 06-errors/01-add-error-handling.cu -run

All elements were doubled? TRUE
```

6. 加速 CPU 应用程序。

最后练习：加速向量加法

(1)扩充 addVectorsInto 定义，使之成为 CUDA 核函数。

(2)选择并使用有效的执行配置，以使 addVectorsInto 作为 CUDA 核函数启动。

(3)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。

(4)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。

(5)在 CUDA 代码可能以其他方式静默失败的位置添加错误处理。

```
#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;
}

void initWith(float num, float *a, int N)
{
    for(int i = 0; i < N; ++i)
    {
        a[i] = num;
    }
}

__global__
void addVectorsInto(float *result, float *a, float *b, int N)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for(int i = index; i < N; i += stride)
    {
        result[i] = a[i] + b[i];
    }
}

void checkElementsAre(float target, float *array, int N)
{
    for(int i = 0; i < N; i++)
    {
        if(array[i] != target)
        {
            printf("FAIL: array[%d] - %0.0f does not equal %0.0f\n", i,
               array[i], target);
```

```
            exit(1);
        }
    }
    printf("SUCCESS! All values added correctly.\n");
}

int main()
{
    const int N = 2<<20;
    size_t size = N * sizeof(float);

    float *a;
    float *b;
    float *c;

    checkCuda( cudaMallocManaged(&a, size) );
    checkCuda( cudaMallocManaged(&b, size) );
    checkCuda( cudaMallocManaged(&c, size) );

    initWith(3, a, N);
    initWith(4, b, N);
    initWith(0, c, N);

    size_t threadsPerBlock;
    size_t numberOfBlocks;

    threadsPerBlock = 256;
    numberOfBlocks = (N + threadsPerBlock - 1) / threadsPerBlock;

    addVectorsInto<<<numberOfBlocks, threadsPerBlock>>>(c, a, b, N);

    checkCuda( cudaGetLastError() );
    checkCuda( cudaDeviceSynchronize() );

    checkElementsAre(7, c, N);

    checkCuda( cudaFree(a) );
    checkCuda( cudaFree(b) );
```

```
    checkCuda( cudaFree(c) );
}
```

```
In [65]: !nvcc -arch=sm_70 -o vector-add 07-vector-add/01-vector-add.cu -run
```

```
SUCCESS! All values added correctly.
```

进阶内容

练习：加速2D矩阵乘法应用

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。我们需要创建执行配置，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，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。

```
#include <stdio.h>

#define N  64

__global__ void matrixMulGPU(int * a, int * b, int * c)
{
    int val = 0;

    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < N && col < N)
    {
        for (int k = 0; k < N; ++k)
            val += a[row * N + k] * b[k * N + col];
        c[row * N + col] = val;
    }
}

void matrixMulCPU(int * a, int * b, int * c)
{
    int val = 0;
```

```

    for(int row = 0; row < N; ++row)
    for(int col = 0; col < N; ++col)
    {
        val = 0;
        for (int k = 0; k < N; ++k)
            val += a[row * N + k] * b[k * N + col];
        c[row * N + col] = val;
    }
}

int main()
{
    int *a, *b, *c_cpu, *c_gpu;

    int size = N * N * sizeof (int); // Number of bytes of an N x N
       matrix

    // Allocate memory
    cudaMallocManaged (&a, size);
    cudaMallocManaged (&b, size);
    cudaMallocManaged (&c_cpu, size);
    cudaMallocManaged (&c_gpu, size);

    // Initialize memory
    for(int row = 0; row < N; ++row)
    for(int col = 0; col < N; ++col)
    {
        a[row*N + col] = row;
        b[row*N + col] = col+2;
        c_cpu[row*N + col] = 0;
        c_gpu[row*N + col] = 0;
    }

    dim3 threads_per_block (16, 16, 1); // A 16 x 16 block threads
    dim3 number_of_blocks ((N / threads_per_block.x) + 1, (N /
       threads_per_block.y) + 1, 1);

```

```
    matrixMulGPU <<< number_of_blocks, threads_per_block >>> ( a, b,
       c_gpu );

    cudaDeviceSynchronize(); // Wait for the GPU to finish before
       proceeding

    // Call the CPU version to check our work
    matrixMulCPU( a, b, c_cpu );

    // Compare the two answers to make sure they are equal
    bool error = false;
    for(int row = 0; row < N && !error; ++row)
    for(int col = 0; col < N && !error; ++col)
    if (c_cpu[row * N + col] != c_gpu[row * N + col])
    {
        printf("FOUND ERROR at c[%d][%d]\n", row, col);
        error = true;
        break;
    }
    if (!error)
    printf("Success!\n");

    // Free all our allocated memory
    cudaFree(a); cudaFree(b);
    cudaFree(c_cpu); cudaFree(c_gpu);
}
```

```
In [66]: !nvcc -arch=sm_70 -o matrix-multiply-2d 08-matrix-multiply/01-matrix-multiply-2d.cu -run
         Success!
```

练习：给热传导应用程序加速

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。

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

// Simple define to index into a 1D array from 2D space
#define I2D(num, c, r) ((r)*(num)+(c))

__global__
void step_kernel_mod(int ni, int nj, float fact, float* temp_in, float*
    temp_out)
{
    int i00, im10, ip10, i0m1, i0p1;
    float d2tdx2, d2tdy2;

    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;

    // loop over all points in domain (except boundary)
    if (j > 0 && i > 0 && j < nj-1 && i < ni-1) {
        // find indices into linear memory
        // for central point and neighbours
        i00 = I2D(ni, i, j);
        im10 = I2D(ni, i-1, j);
        ip10 = I2D(ni, i+1, j);
        i0m1 = I2D(ni, i, j-1);
        i0p1 = I2D(ni, i, j+1);

        // evaluate derivatives
        d2tdx2 = temp_in[im10]-2*temp_in[i00]+temp_in[ip10];
        d2tdy2 = temp_in[i0m1]-2*temp_in[i00]+temp_in[i0p1];

        // update temperatures
        temp_out[i00] = temp_in[i00]+fact*(d2tdx2 + d2tdy2);
    }
}

void step_kernel_ref(int ni, int nj, float fact, float* temp_in, float*
    temp_out)
```

```
{
    int i00, im10, ip10, i0m1, i0p1;
    float d2tdx2, d2tdy2;

    // loop over all points in domain (except boundary)
    for (int j=1; j < nj-1; j++) {
        for (int i=1; i < ni-1; i++) {
            // find indices into linear memory
            // for central point and neighbours
            i00 = I2D(ni, i, j);
            im10 = I2D(ni, i-1, j);
            ip10 = I2D(ni, i+1, j);
            i0m1 = I2D(ni, i, j-1);
            i0p1 = I2D(ni, i, j+1);

            // evaluate derivatives
            d2tdx2 = temp_in[im10]-2*temp_in[i00]+temp_in[ip10];
            d2tdy2 = temp_in[i0m1]-2*temp_in[i00]+temp_in[i0p1];

            // update temperatures
            temp_out[i00] = temp_in[i00]+fact*(d2tdx2 + d2tdy2);
        }
    }
}

int main()
{
    int istep;
    int nstep = 200; // number of time steps

    // Specify our 2D dimensions
    const int ni = 200;
    const int nj = 100;
    float tfac = 8.418e-5; // thermal diffusivity of silver

    float *temp1_ref, *temp2_ref, *temp1, *temp2, *temp_tmp;

    const int size = ni * nj * sizeof(float);
```

```

    temp1_ref = (float*)malloc(size);
    temp2_ref = (float*)malloc(size);
    cudaMallocManaged(&temp1, size);
    cudaMallocManaged(&temp2, size);

    // Initialize with random data
    for(int i = 0; i < ni*nj; ++i) {
        temp1_ref[i] = temp2_ref[i] = temp1[i] = temp2[i] = (float)rand()/(float)(RAND_MAX/100.0f);
    }

    // Execute the CPU-only reference version
    for (istep=0; istep < nstep; istep++) {
        step_kernel_ref(ni, nj, tfac, temp1_ref, temp2_ref);

        // swap the temperature pointers
        temp_tmp = temp1_ref;
        temp1_ref = temp2_ref;
        temp2_ref= temp_tmp;
    }

    dim3 tblocks(32, 16, 1);
    dim3 grid((nj/tblocks.x)+1, (ni/tblocks.y)+1, 1);
    cudaError_t ierrSync, ierrAsync;

    // Execute the modified version using same data
    for (istep=0; istep < nstep; istep++) {
        step_kernel_mod<<< grid, tblocks >>>(ni, nj, tfac, temp1, temp2);

        ierrSync = cudaGetLastError();
        ierrAsync = cudaDeviceSynchronize(); // Wait for the GPU to finish
        if (ierrSync != cudaSuccess) { printf("Sync error: %s\n", cudaGetErrorString(ierrSync)); }
        if (ierrAsync != cudaSuccess) { printf("Async error: %s\n", cudaGetErrorString(ierrAsync)); }
```

```
        // swap the temperature pointers
        temp_tmp = temp1;
        temp1 = temp2;
        temp2= temp_tmp;
    }

    float maxError = 0;
    // Output should always be stored in the temp1 and temp1_ref at this
        point
    for(int i = 0; i < ni*nj; ++i) {
        if (abs(temp1[i]-temp1_ref[i]) > maxError) { maxError = abs(
            temp1[i]-temp1_ref[i]); }
    }

    // Check and see if our maxError is greater than an error bound
    if (maxError > 0.0005f)
        printf("Problem! The Max Error of %.5f is NOT within acceptable
            bounds.\n", maxError);
    else
        printf("The Max Error of %.5f is within acceptable bounds.\n",
            maxError);

    free(temp1_ref);
    free(temp2_ref);
    cudaFree(temp1);
    cudaFree(temp2);

    return 0;
}
```

In [74]:
```
!nvcc -arch=sm_70 -o heat-conduction 09-heat/01-heat-conduction.cu -run
```

```
The Max Error of 0.00001 is within acceptable bounds.
```

3.2 利用基本的 CUDA 内存管理技术来优化加速应用程序

1. 使用 Nsight Systems命令行分析器 (nsys) 分析被加速的应用程序的性能。

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。

练习：使用nsys分析应用程序

CUDA API统计信息:

CUDA API Statistics:

Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
90.8	2332321968	1	2332321968.0	2332321968	2332321968	cudaDeviceSynchronize
8.4	216948530	3	72316176.7	19024	216885828	cudaMallocManaged
0.7	19122430	3	6374143.3	5654514	7508666	cudaFree
0.0	56725	1	56725.0	56725	56725	cudaLaunchKernel

CUDA核函数的统计信息:

CUDA Kernel Statistics:

Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
100.0	2332309553	1	2332309553.0	2332309553	2332309553	addVectorsInto(float*, float*, float*, int)

CUDA内存操作统计信息（时间和大小）:

CUDA Memory Operation Statistics (by time):

Time(%)	Total Time (ns)	Operations	Average	Minimum	Maximum	Operation
76.6	68491008	2304	29727.0	1695	181728	[CUDA Unified Memory memcpy HtoD]
23.4	20882304	768	27190.5	1119	168192	[CUDA Unified Memory memcpy DtoH]

CUDA Memory Operation Statistics (by size in KiB):

Total	Operations	Average	Minimum	Maximum	Operation
393216.000	2304	170.667	4.000	1020.000	[CUDA Unified Memory memcpy HtoD]
131072.000	768	170.667	4.000	1020.000	[CUDA Unified Memory memcpy DtoH]

由上可发现此应用程序中唯一调用的 CUDA 核函数的名称是 addVectorsInto，此核函数运行了1次，运行时间为 2332309553 ns。

练习：优化并分析性能

更新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。

更新设置，单线程块中两个线程，得到核函数的统计信息如下：可以发现速度提升了1.5倍。

```
CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances    Average      Minimum     Maximum                      Name
 -------  ---------------  ---------  ------------  ----------  ----------  ----------------------------------------
   100.0       1554224198          1  1554224198.0  1554224198  1554224198  addVectorsInto(float*, float*, float*, int)
```

更新设置，单线程块中四个线程，得到核函数的统计信息如下：可以发现速度提升了1.98倍。

```
CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances    Average      Minimum     Maximum                      Name
 -------  ---------------  ---------  ------------  ----------  ----------  ----------------------------------------
   100.0       1176566816          1  1176566816.0  1176566816  1176566816  addVectorsInto(float*, float*, float*, int)
```

练习：迭代优化

多轮周期式的迭代优化，记录核函数运行时间，判断在哪个配置下优化最好。

配置	时间	加速比
block=2, thread=2	1158242058	2.014
block=2, thread=4	955268808	2.442
block=4, thread=2	922295027	2.529
block=4, thread=4	605991713	3.849
block=4, thread=8	414827559	5.622
block=8, thread=8	280536396	8.314
block=16, thread=8	212570593	10.972
block=16, thread=16	166507011	14.007
block=32, thread=16	166977858	13.968

各个配置的核函数运行时间如上表，可知不一定block和thread的数量越大越好的！

2. 利用对流多处理器的理解优化执行配置。

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。

练习：查询设备信息

```
#include <stdio.h>

int main()
{
    /*
    * Device ID is required first to query the device.
    */

    int deviceId;
    cudaGetDevice(&deviceId);

    cudaDeviceProp props;
    cudaGetDeviceProperties(&props, deviceId);

    /*
    * props now contains several properties about the current device.
    */

    int computeCapabilityMajor = props.major;
    int computeCapabilityMinor = props.minor;
    int multiProcessorCount = props.multiProcessorCount;
    int warpSize = props.warpSize;

    printf("Device ID: %d\nNumber of SMs: %d\nCompute Capability Major:
       %d\nCompute Capability Minor: %d\nWarp Size: %d\n", deviceId,
       multiProcessorCount, computeCapabilityMajor,
       computeCapabilityMinor, warpSize);
}
```

```
In [36]: !nvcc -o get-device-properties 04-device-properties/01-get-device-properties.cu -run

Device ID: 0
Number of SMs: 40
Compute Capability Major: 7
Compute Capability Minor: 5
Warp Size: 32
```

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 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。

根据之前的练习可知 SM 的数量为 40，因此设置线程块数为 80，每个线程块的线程数为2，得到如下的运行时间：优化近十倍!

CUDA Kernel Statistics:

Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
100.0	236243908	1	236243908.0	236243908	236243908	addVectorsInto(float*, float*, float*, int)

3. 理解统一内存在页错误和数据迁移方面的行为。

分配统一内存(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。

练习：探索统一内存（UM）的页错误

当仅通过CPU访问统一内存时:

```
__global__
void deviceKernel(int *a, int N)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = idx; i < N; i += stride)
    {
        a[i] = 1;
    }
}

void hostFunction(int *a, int N)
```

```
{
    for (int i = 0; i < N; ++i)
    {
        a[i] = 1;
    }
}

int main()
{

    int N = 2<<24;
    size_t size = N * sizeof(int);
    int *a;
    cudaMallocManaged(&a, size);
    hostFunction(a, N);
    cudaFree(a);
}
```

没有内存迁移和/或页面错误的证据。

```
==1535== Unified Memory profiling result:
Total CPU Page faults: 384
```

当仅通过GPU访问统一内存时:

```
__global__
void deviceKernel(int *a, int N)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = idx; i < N; i += stride)
    {
        a[i] = 1;
    }
}

void hostFunction(int *a, int N)
{
    for (int i = 0; i < N; ++i)
```

```
    {
        a[i] = 1;
    }
}

int main()
{

    int N = 2<<24;
    size_t size = N * sizeof(int);
    int *a;
    cudaMallocManaged(&a, size);
    deviceKernel<<<256, 256>>>(a, N);
    cudaDeviceSynchronize();
    cudaFree(a);
}
```

没有内存迁移和/或页面错误的证据。

```
==1589== Unified Memory profiling result:
Device "Tesla T4 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
     416         -         -         -           -  22.34403ms  Gpu page fault groups
```

当先由GPU然后由CPU访问统一内存时:

```
__global__
void deviceKernel(int *a, int N)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = idx; i < N; i += stride)
    {
        a[i] = 1;
    }
}

void hostFunction(int *a, int N)
{
    for (int i = 0; i < N; ++i)
```

```
    {
        a[i] = 1;
    }
}

int main()
{

    int N = 2<<24;
    size_t size = N * sizeof(int);
    int *a;
    cudaMallocManaged(&a, size);
    deviceKernel<<<256, 256>>>(a, N);
    cudaDeviceSynchronize();
    cudaFree(a);
}
```

有内存迁移和/或页面错误的证据。

```
==1697== Unified Memory profiling result:
Device "Tesla T4 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
     768  170.67KB  4.0000KB  0.9961MB  128.0000MB  21.19584ms  Device To Host
     418         -         -         -           -  22.41690ms  Gpu page fault groups
Total CPU Page faults: 384
```

当先由CPU然后由GPU访问统一内存时:

```
__global__
void deviceKernel(int *a, int N)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = idx; i < N; i += stride)
    {
        a[i] = 1;
    }
}

void hostFunction(int *a, int N)
{
```

```
    for (int i = 0; i < N; ++i)
    {
        a[i] = 1;
    }
}

int main()
{

    int N = 2<<24;
    size_t size = N * sizeof(int);
    int *a;
    cudaMallocManaged(&a, size);
    deviceKernel<<<256, 256>>>(a, N);
    cudaDeviceSynchronize();
    cudaFree(a);
}
```

有内存迁移和/或页面错误的证据。

```
==1643== Unified Memory profiling result:
Device "Tesla T4 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    4520  28.998KB  4.0000KB  724.00KB  128.0000MB  29.47654ms  Host To Device
     416         -         -         -           -  72.16406ms  Gpu page fault groups
Total CPU Page faults: 384
```

练习：重新审视矢量加法程序的UM行为

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。

内存迁移信息如下：

CUDA Memory Operation Statistics (by time):

Time(%)	Total Time (ns)	Operations	Average	Minimum	Maximum	Operation
76.7	69092832	2304	29988.2	2207	182176	[CUDA Unified Memory memcpy HtoD]
23.3	20948512	768	27276.7	1183	173440	[CUDA Unified Memory memcpy DtoH]

CUDA Memory Operation Statistics (by size in KiB):

Total	Operations	Average	Minimum	Maximum	Operation
393216.000	2304	170.667	4.000	1020.000	[CUDA Unified Memory memcpy HtoD]
131072.000	768	170.667	4.000	1020.000	[CUDA Unified Memory memcpy DtoH]

练习：在核函数中初始化向量

将程序中的 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。

```
#include <stdio.h>

/*
* Refactor host function to run as CUDA kernel
*/

__global__
void initWith(float num, float *a, int N)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for(int i = index; i < N; i += stride)
    {
        a[i] = num;
    }
```

```
}

__global__
void addArraysInto(float *result, float *a, float *b, int N)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for(int i = index; i < N; i += stride)
    {
        result[i] = a[i] + b[i];
    }
}

void checkElementsAre(float target, float *array, int N)
{
    for(int i = 0; i < N; i++)
    {
        if(array[i] != target)
        {
            printf("FAIL: array[%d] - %0.0f does not equal %0.0f\n", i,
              array[i], target);
            exit(1);
        }
    }
    printf("Success! All values calculated correctly.\n");
}

int main()
{
    int deviceId;
    int numberOfSMs;

    cudaGetDevice(&deviceId);
    cudaDeviceGetAttribute(&numberOfSMs, cudaDevAttrMultiProcessorCount,
        deviceId);
    printf("Device ID: %d\tNumber of SMs: %d\n", deviceId, numberOfSMs);

```

```
    const int N = 2<<24;
    size_t size = N * sizeof(float);

    float *a;
    float *b;
    float *c;

    cudaMallocManaged(&a, size);
    cudaMallocManaged(&b, size);
    cudaMallocManaged(&c, size);

    size_t threadsPerBlock;
    size_t numberOfBlocks;

    threadsPerBlock = 256;
    numberOfBlocks = 32 * numberOfSMs;

    cudaError_t addArraysErr;
    cudaError_t asyncErr;

    /*
    * Launch kernels.
    */

    initWith<<<numberOfBlocks, threadsPerBlock>>>(3, a, N);
    initWith<<<numberOfBlocks, threadsPerBlock>>>(4, b, N);
    initWith<<<numberOfBlocks, threadsPerBlock>>>(0, c, N);

    /*
    * Now that initialization is happening on a GPU, host code
    * must be synchronized to wait for its completion.
    */

    cudaDeviceSynchronize();

    addArraysInto<<<numberOfBlocks, threadsPerBlock>>>(c, a, b, N);

    addArraysErr = cudaGetLastError();
```

```
    if(addArraysErr != cudaSuccess) printf("Error: %s\n",
        cudaGetErrorString(addArraysErr));

    asyncErr = cudaDeviceSynchronize();
    if(asyncErr != cudaSuccess) printf("Error: %s\n", cudaGetErrorString
        (asyncErr));

    checkElementsAre(7, c, N);

    cudaFree(a);
    cudaFree(b);
    cudaFree(c);
}
```

如下可知，在核函数中初始化向量之后，没有 Host 到 Device 中的内存迁移了。

CUDA Memory Operation Statistics (by time):

Time(%)	Total Time (ns)	Operations	Average	Minimum	Maximum	Operation
100.0	21147808	768	27536.2	1631	165088	[CUDA Unified Memory memcpy DtoH]

CUDA Memory Operation Statistics (by size in KiB):

Total	Operations	Average	Minimum	Maximum	Operation
131072.000	768	170.667	4.000	1020.000	[CUDA Unified Memory memcpy DtoH]

4. 使用异步内存预取减少页错误和数据迁移以提高性能。

通过异步内存预取，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，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。

练习：异步内存预取

将其中一个初始化向量预取到主机时：

CUDA Memory Operation Statistics (by size in KiB):

Total	Operations	Average	Minimum	Maximum	Operation
262144.000	128	2048.000	2048.000	2048.000	[CUDA Unified Memory memcpy HtoD]
131072.000	768	170.667	4.000	1020.000	[CUDA Unified Memory memcpy DtoH]

将其中两个初始化向量预取到主机时：

CUDA Memory Operation Statistics (by size in KiB):

Total	Operations	Average	Minimum	Maximum	Operation
131072.000	768	170.667	4.000	1020.000	[CUDA Unified Memory memcpy DtoH]
131072.000	64	2048.000	2048.000	2048.000	[CUDA Unified Memory memcpy HtoD]

将三个初始化向量预取到主机时：

CUDA Memory Operation Statistics (by size in KiB):

Total	Operations	Average	Minimum	Maximum	Operation
131072.000	768	170.667	4.000	1020.000	[CUDA Unified Memory memcpy DtoH]

可以发现：在使用异步预取进行了一系列重构之后，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。

练习：将内存预取回CPU

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。

```
#include <stdio.h>

void initWith(float num, float *a, int N)
{
    for(int i = 0; i < N; ++i)
    {
        a[i] = num;
    }
}

__global__
void addVectorsInto(float *result, float *a, float *b, int N)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for(int i = index; i < N; i += stride)
    {
        result[i] = a[i] + b[i];
```

```
    }
}

void checkElementsAre(float target, float *vector, int N)
{
    for(int i = 0; i < N; i++)
    {
        if(vector[i] != target)
        {
            printf("FAIL: vector[%d] - %0.0f does not equal %0.0f\n", i,
                vector[i], target);
            exit(1);
        }
    }
    printf("Success! All values calculated correctly.\n");
}

int main()
{
    int deviceId;
    int numberOfSMs;

    cudaGetDevice(&deviceId);
    cudaDeviceGetAttribute(&numberOfSMs, cudaDevAttrMultiProcessorCount,
        deviceId);
    printf("Device ID: %d\tNumber of SMs: %d\n", deviceId, numberOfSMs);

    const int N = 2<<24;
    size_t size = N * sizeof(float);

    float *a;
    float *b;
    float *c;

    cudaMallocManaged(&a, size);
    cudaMallocManaged(&b, size);
    cudaMallocManaged(&c, size);

```

```
    /*
    * Prefetching can also be used to prevent CPU page faults.
    */

    cudaMemPrefetchAsync(a, size, cudaCpuDeviceId);
    cudaMemPrefetchAsync(b, size, cudaCpuDeviceId);
    cudaMemPrefetchAsync(c, size, cudaCpuDeviceId);
    initWith(3, a, N);
    initWith(4, b, N);
    initWith(0, c, N);

    cudaMemPrefetchAsync(a, size, deviceId);
    cudaMemPrefetchAsync(b, size, deviceId);
    cudaMemPrefetchAsync(c, size, deviceId);

    size_t threadsPerBlock;
    size_t numberOfBlocks;

    threadsPerBlock = 256;
    numberOfBlocks = 32 * numberOfSMs;

    cudaError_t addVectorsErr;
    cudaError_t asyncErr;

    addVectorsInto<<<numberOfBlocks, threadsPerBlock>>>(c, a, b, N);

    addVectorsErr = cudaGetLastError();
    if(addVectorsErr != cudaSuccess) printf("Error: %s\n",
       cudaGetErrorString(addVectorsErr));

    asyncErr = cudaDeviceSynchronize();
    if(asyncErr != cudaSuccess) printf("Error: %s\n", cudaGetErrorString
       (asyncErr));

    /*
    * Prefetching can also be used to prevent CPU page faults.
    */

```

```
    cudaMemPrefetchAsync(c, size, cudaCpuDeviceId);
    checkElementsAre(7, c, N);

    cudaFree(a);
    cudaFree(b);
    cudaFree(c);
}
```

```
In [60]: !nvcc -o prefetch-to-gpu 01-vector-add/01-vector-add.cu -run

Device ID: 0    Number of SMs: 40
Success! All values calculated correctly.
```

5. 采用循环式的迭代开发加快应用程序的优化加速和部署。

最后的练习：迭代优化加速的SAXPY应用程序

```
#include <stdio.h>

#define N 2048 * 2048 // Number of elements in each vector

__global__ void saxpy(int * a, int * b, int * c)
{
    // Determine our unique global thread ID, so we know which element
        to process
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = tid; i < N; i += stride)
    c[i] = 2 * a[i] + b[i];
}

int main()
{
    int *a, *b, *c;

    int size = N * sizeof (int); // The total number of bytes per vector

    int deviceId;
    int numberOfSMs;
```

```

  cudaGetDevice(&deviceId);
  cudaDeviceGetAttribute(&numberOfSMs, cudaDevAttrMultiProcessorCount,
      deviceId);

  // Allocate memory
  cudaMallocManaged(&a, size);
  cudaMallocManaged(&b, size);
  cudaMallocManaged(&c, size);

  // Initialize memory
  for( int i = 0; i < N; ++i )
  {
      a[i] = 2;
      b[i] = 1;
      c[i] = 0;
  }

  cudaMemPrefetchAsync(a, size, deviceId);
  cudaMemPrefetchAsync(b, size, deviceId);
  cudaMemPrefetchAsync(c, size, deviceId);

  int threads_per_block = 256;
  int number_of_blocks = numberOfSMs * 32;

  saxpy<<<number_of_blocks, threads_per_block>>>( a, b, c );

  cudaDeviceSynchronize(); // Wait for the GPU to finish

  // Print out the first and last 5 values of c for a quality check
  for( int i = 0; i < 5; ++i )
  printf("c[%d] = %d, ", i, c[i]);
  printf ("\n");
  for( int i = N-5; i < N; ++i )
  printf("c[%d] = %d, ", i, c[i]);
  printf ("\n");

  // Free all our allocated memory
```

```
    cudaFree( a ); cudaFree( b ); cudaFree( c );
}
```

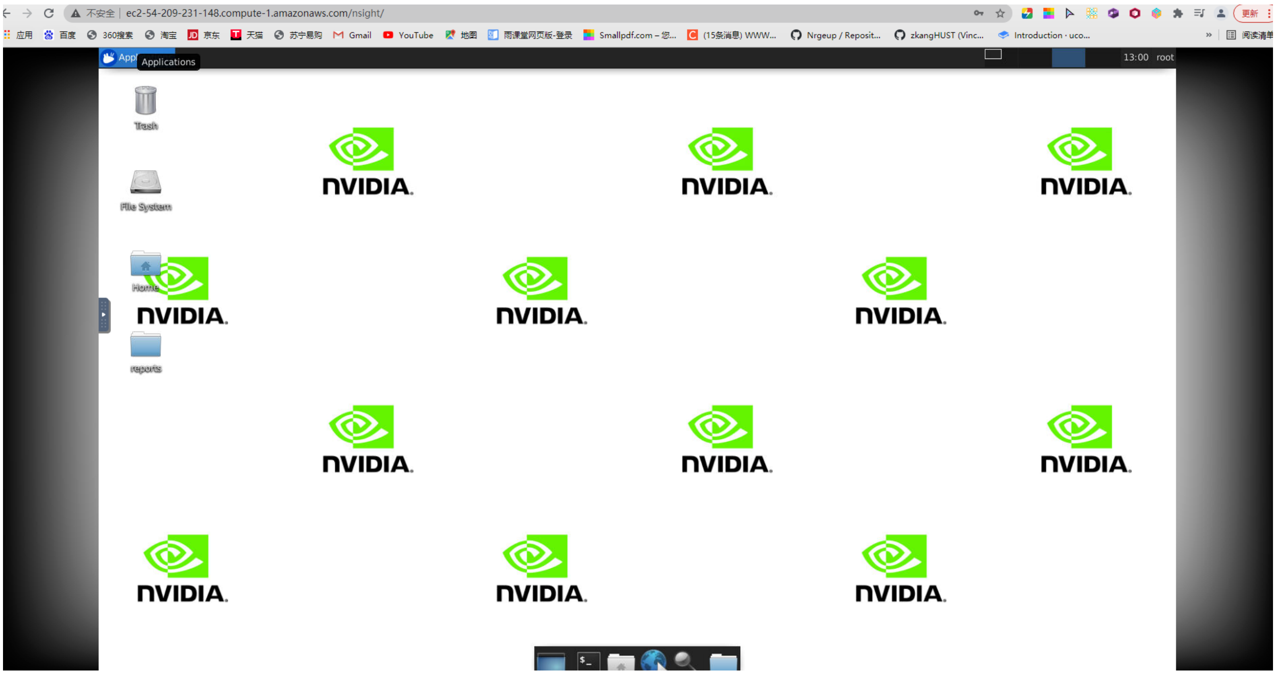
```
In [54]: !nvcc -o saxpy 09-saxpy/01-saxpy.cu -run

c[0] = 5, c[1] = 5, c[2] = 5, c[3] = 5, c[4] = 5,
c[4194299] = 5, c[4194300] = 5, c[4194301] = 5, c[4194302] = 5, c[4194303] = 5,
```

3.3 被加速的 C/C++ 应用程序的异步流和可视化分析

运行Nsight Systems:

生成报告文件，打开远程桌面，打开远程桌面终端应用程序：

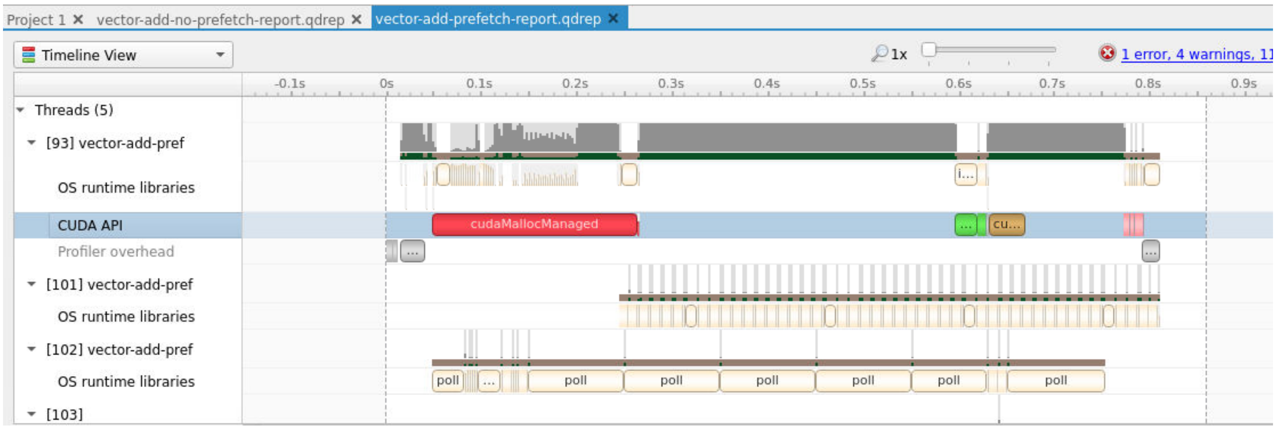


然后打开Nsight Systems，启用使用情况报告，打开报告文件，忽略警告/错误信息，然后可以查看对应时间表。

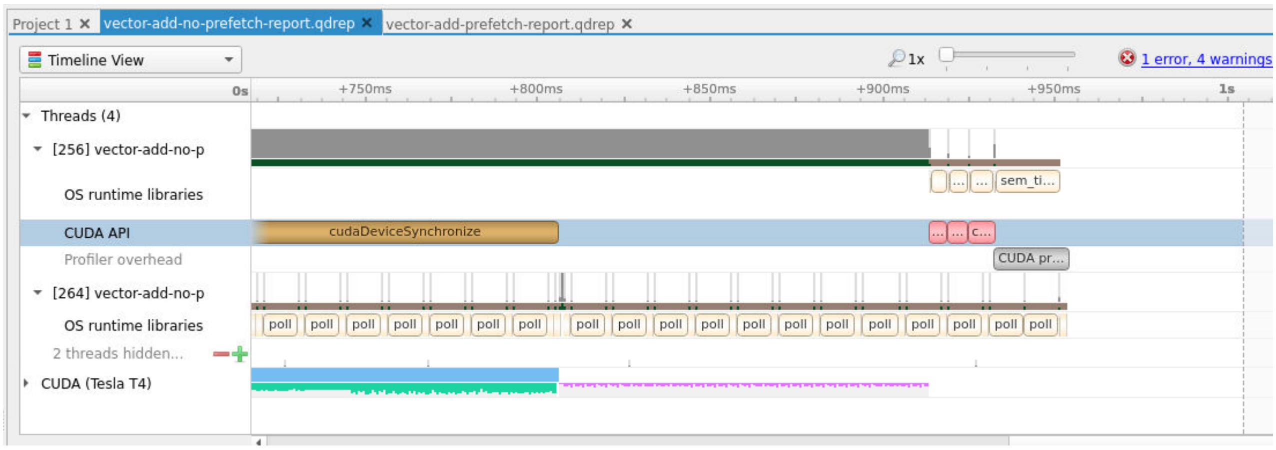
1. 使用Nsight Systems直观地描述由GPU加速的CUDA应用程序的时间表。

练习：比较预取与不预取的活动时间表

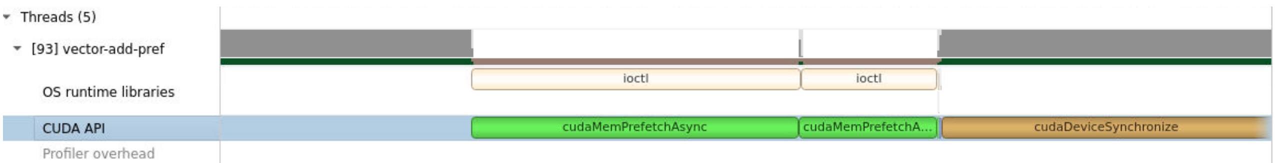
预取的活动时间表如下：



不预取的活动时间表如下：



如上可发现预取的时间比不预取的时间更短，如下可以看到 cudaMemPrefetchAsync 预取到设备的时间表：

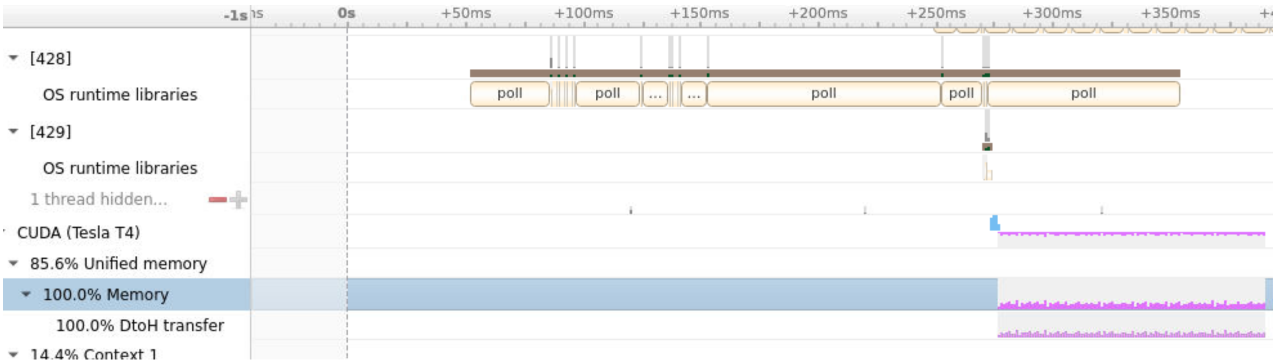


2. 使用Nsight Systems识别和利用CUDA应用程序中的优化机会。

练习：使用核函数进行向量初始化并分析其性能

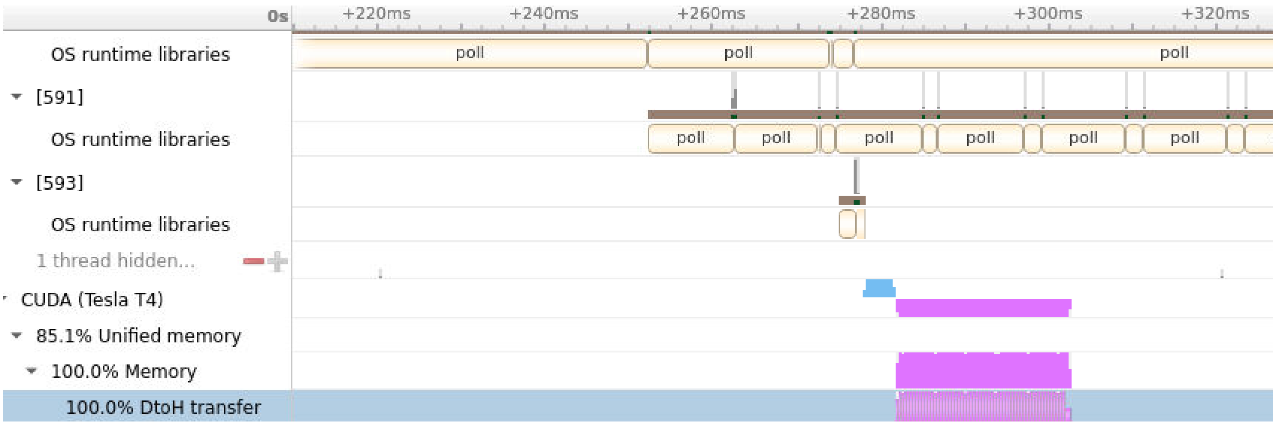
查看时间表如下：可以发现，两个核函数（addVectorsInto和初始化核函数）中的初始化核函数占用了GPU的大部分时间。

由于初始化是在核函数中进行的，因此可以发现没有 Host 到 Device 的数据迁移，与我们之前的实验是相符的。



练习：使用异步预将数据取回主机，并分析其性能

使用数据预取之后，可以发现数据迁移的时间显然减少了，只有20多毫秒，而之前有100多毫秒。



3. 利用CUDA流在被加速的应用程序中并发执行核函数。

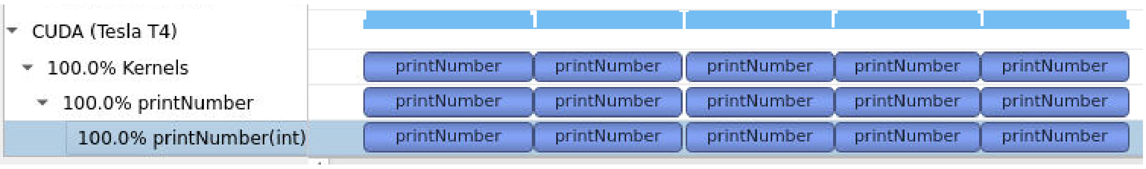
在 CUDA 编程中，流是由按顺序执行的一系列命令构成。在 CUDA 应用程序中，核函数的执行以及一些内存传输均在 CUDA 流中进行。但实际上 CUDA 代码已在名为默认流的流中执行了其核函数。除默认流以外，CUDA 程序员还可创建并使用非默认 CUDA 流，此举可支持执行多个操作，例如

在不同的流中并发执行多个核函数。多流的使用可以为加速应用程序带来另外一个层次的并行，并能提供更多应用程序的优化机会。

练习：预测默认流行为

程序带有一个非常简单的 printNumber 核函数，可用于接受及打印整数。仅在单个线程块内使用单线程执行该核函数，但使用“for 循环”可执行 5 次，并且每次启动时都会传递“for 循环”的迭代次数。

可以发现，5次运行是按序执行的：



练习：实现并发CUDA流

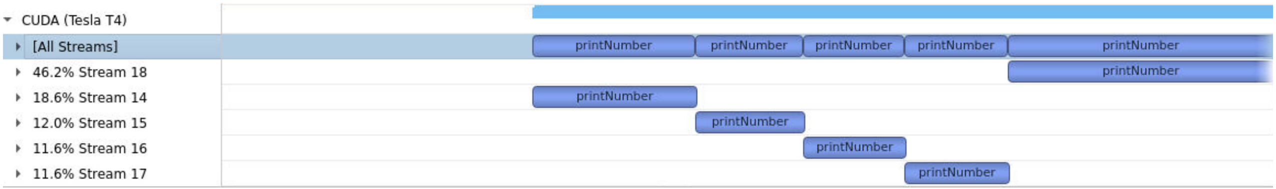
由于默认流具有阻断作用，所以核函数都会在完成本次启动之后才启动下一次，重构程序，以便核函数的每次启动都在自身非默认流中进行。若已不再需要所创建的流，请务必予以销毁。

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

__global__ void printNumber(int number)
{
    printf("%d\n", number);
}

int main()
{
    for (int i = 0; i < 5; ++i)
    {
        cudaStream_t stream;
        cudaStreamCreate(&stream);
        printNumber<<<1, 1, 0, stream>>>(i);
        cudaStreamDestroy(stream);
    }
    cudaDeviceSynchronize();
}
```

如下可发现，有5个流：



练习：将流用于并行进行数据初始化的核函数

重构该应用程序，以便在其各自的非默认流中启动全部 3 个初始化核函数。

```
#include <stdio.h>

__global__
void initWith(float num, float *a, int N)
{

    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for(int i = index; i < N; i += stride)
    {
        a[i] = num;
    }
}

__global__
void addVectorsInto(float *result, float *a, float *b, int N)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for(int i = index; i < N; i += stride)
    {
        result[i] = a[i] + b[i];
    }
}

void checkElementsAre(float target, float *vector, int N)
```

```
{
    for(int i = 0; i < N; i++)
    {
        if(vector[i] != target)
        {
            printf("FAIL: vector[%d] - %0.0f does not equal %0.0f\n", i,
                vector[i], target);
            exit(1);
        }
    }
    printf("Success! All values calculated correctly.\n");
}

int main()
{
    int deviceId;
    int numberOfSMs;

    cudaGetDevice(&deviceId);
    cudaDeviceGetAttribute(&numberOfSMs, cudaDevAttrMultiProcessorCount,
        deviceId);

    const int N = 2<<24;
    size_t size = N * sizeof(float);

    float *a;
    float *b;
    float *c;

    cudaMallocManaged(&a, size);
    cudaMallocManaged(&b, size);
    cudaMallocManaged(&c, size);

    cudaMemPrefetchAsync(a, size, deviceId);
    cudaMemPrefetchAsync(b, size, deviceId);
    cudaMemPrefetchAsync(c, size, deviceId);

    size_t threadsPerBlock;
```

```
    size_t numberOfBlocks;

    threadsPerBlock = 256;
    numberOfBlocks = 32 * numberOfSMs;

    cudaError_t addVectorsErr;
    cudaError_t asyncErr;

    /*
    * Create 3 streams to run initialize the 3 data vectors in parallel.
    */

    cudaStream_t stream1, stream2, stream3;
    cudaStreamCreate(&stream1);
    cudaStreamCreate(&stream2);
    cudaStreamCreate(&stream3);

    /*
    * Give each initWith launch its own non-standard stream.
    */

    initWith<<<numberOfBlocks, threadsPerBlock, 0, stream1>>>(3, a, N);
    initWith<<<numberOfBlocks, threadsPerBlock, 0, stream2>>>(4, b, N);
    initWith<<<numberOfBlocks, threadsPerBlock, 0, stream3>>>(0, c, N);

    addVectorsInto<<<numberOfBlocks, threadsPerBlock>>>(c, a, b, N);

    addVectorsErr = cudaGetLastError();
    if(addVectorsErr != cudaSuccess) printf("Error: %s\n",
        cudaGetErrorString(addVectorsErr));

    asyncErr = cudaDeviceSynchronize();
    if(asyncErr != cudaSuccess) printf("Error: %s\n", cudaGetErrorString
        (asyncErr));

    cudaMemPrefetchAsync(c, size, cudaCpuDeviceId);

    checkElementsAre(7, c, N);
```

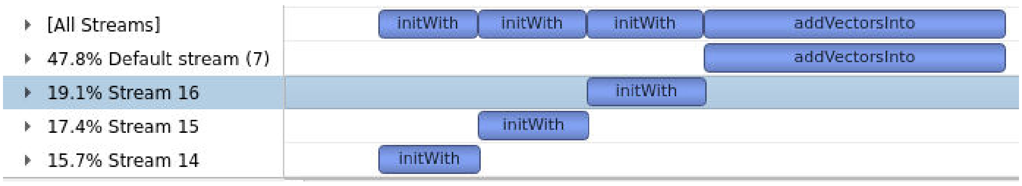
```

    /*
    * Destroy streams when they are no longer needed.
    */

    cudaStreamDestroy(stream1);
    cudaStreamDestroy(stream2);
    cudaStreamDestroy(stream3);

    cudaFree(a);
    cudaFree(b);
    cudaFree(c);
}
```

如下可发现，有3个非默认流：



最后的练习：加速和优化N体模拟器

代码思路是：将bodyForce函数改为核函数，在GPU上运行。因为多个epoch必须按序执行，所以无法使用并发的cuda流，默认的串行流行为可以完成任务。将bodyForce执行结束后的for循环改为核函数。其他技巧就是块数和线程数设置。

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "timer.h"
#include "files.h"

#define SOFTENING 1e-9f

/*
* Each body contains x, y, and z coordinate positions,
* as well as velocities in the x, y, and z directions.
```

```
*/

typedef struct
{
    float x, y, z, vx, vy, vz;
} Body;

/*
* This function calculates the gravitational impact of all bodies in the system
* on all others, but does not update their positions.
*/

__global__ void bodyForce(Body *p, float dt, int n)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = index; i < n; i += stride)
    {
        float Fx = 0.0f;
        float Fy = 0.0f;
        float Fz = 0.0f;
        for (int j = 0; j < n; j++)
        {
            float dx = p[j].x - p[i].x;
            float dy = p[j].y - p[i].y;
            float dz = p[j].z - p[i].z;
            float distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
            float invDist = rsqrtf(distSqr);
            float invDist3 = invDist * invDist * invDist;

            Fx += dx * invDist3;
            Fy += dy * invDist3;
            Fz += dz * invDist3;
        }

        p[i].vx += dt * Fx;
```

```
        p[i].vy += dt * Fy;
        p[i].vz += dt * Fz;
    }
}
__global__ void add(Body *p, float dt, int n)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride)
    {
        p[i].x += p[i].vx * dt;
        p[i].y += p[i].vy * dt;
        p[i].z += p[i].vz * dt;
    }
}

int main(const int argc, const char **argv)
{
    int deviceId;
    int numberOfSMs;

    cudaGetDevice(&deviceId);
    cudaDeviceGetAttribute(&numberOfSMs, cudaDevAttrMultiProcessorCount,
        deviceId);

    /*
    * Do not change the value for nBodies here. If you would like to
        modify it,
    * pass values into the command line.
    */

    int nBodies = 2 << 11;
    if (argc > 1)
    nBodies = 2 << atoi(argv[1]);

    const char *initialized_values;
    const char *solution_values;

```

```
    if (nBodies == 2 << 11)
    {
        initialized_values = "files/initialized_4096";
        solution_values = "files/solution_4096";
    }
    else
    { // nBodies == 2<<15
        initialized_values = "files/initialized_65536";
        solution_values = "files/solution_65536";
    }

    if (argc > 2)
    initialized_values = argv[2];
    if (argc > 3)
    solution_values = argv[3];

    const float dt = 0.01f; // time step
    const int nIters = 10;  // simulation iterations

    int bytes = nBodies * sizeof(Body);
    float *buf;
    buf = (float *)malloc(bytes);

    cudaMallocManaged(&buf, bytes);
    //cudaMemPrefetchAsync(buf, bytes, deviceId);

    Body *p = (Body *)buf;

    /*
    * As a constraint of this exercise, randomizeBodies must remain a host
        function.
    */
    read_values_from_file(initialized_values, buf, bytes);

    size_t threadsPerBlock = 256;
    size_t numberOfBlocks = 32 * numberOfSMs;

    double totalTime = 0.0;
```

```

/*
* This simulation will run for 10 cycles of time, calculating gravitational
* interaction amongst bodies, and adjusting their positions to reflect.
*/


for (int iter = 0; iter < nIters; iter++)
{
    StartTimer();
    /*
    * You will likely wish to refactor the work being done in bodyForce,
    * as well as the work to integrate the positions.
    */
    bodyForce<<<numberOfBlocks, threadsPerBlock>>>(p, dt, nBodies); // compute interbody forces
    cudaDeviceSynchronize();
    add<<<numberOfBlocks, threadsPerBlock>>>(p, dt, nBodies);

    const double tElapsed = GetTimer() / 1000.0;
    totalTime += tElapsed;
}
cudaDeviceSynchronize();

double avgTime = totalTime / (double)(nIters);
float billionsOfOpsPerSecond = 1e-9 * nBodies * nBodies / avgTime;
write_values_to_file(solution_values, buf, bytes);

printf("%0.3f Billion Interactions / second", billionsOfOpsPerSecond);

/*
* Feel free to modify code below.
*/

```

```
    cudaFree(buf);
}
```

得到最后的评估结果如下：

```
In [31]: run_assessment()
```

```
使用4096个物体运行n-体模拟器

此应用程序应该运行得快于0.9秒。
您的应用程序运行了：0.1657秒
您的应用程序运行速度是  18.314 Billion Interactions / second
您的结果是正确的。

使用65536个物体运行n-体模拟器

此应用程序应该运行得快于1.3秒。
您的应用程序运行了：0.5138秒
您的应用程序运行速度是  118.380 Billion Interactions / second
您的结果是正确的。

祝贺您！您通过了评估！
```

证书：

完成了所有的学习内容：

学生'1811507' (1491465778@qq.com)的学习进度

Congratulations, you qualified for a certificate!

You can keep working for a higher grade, or request your certificate now.

申请证书

拿到了证书：

NVIDIA DEEP LEARNING INSTITUTE
CERTIFICATE OF COMPETENCY

This certificate is awarded to
1811507

for demonstrating competence in the completion of
加速计算基础 —— CUDA C/C++
FUNDAMENTALS OF ACCELERATED COMPUTING WITH CUDA C/C++

Will Ramey
Senior Director, Developer Programs, NVIDIA

2021 June 10
Year issued

NVIDIA. DEEP LEARNING INSTITUTE

4 总结

通过本次实验，学习了编译 GPU 核函数、配置线程块和线程数、分配和释放 GPU 的内存、CUDA 错误处理，并学习了使用 Nsight Systems 命令行分析器分析被加速的应用程序，针对流处理器优化执行配置，使用异步内存预取减少页错误和数据迁移以提高性能，最后学习通过可视化分析工具对异步流进行分析。