



南開大學
Nankai University

计算机学院
并行程序设计实验报告

GPU 学习报告

姓名：刘宇轩

学号：2012677

专业：计算机科学与技术

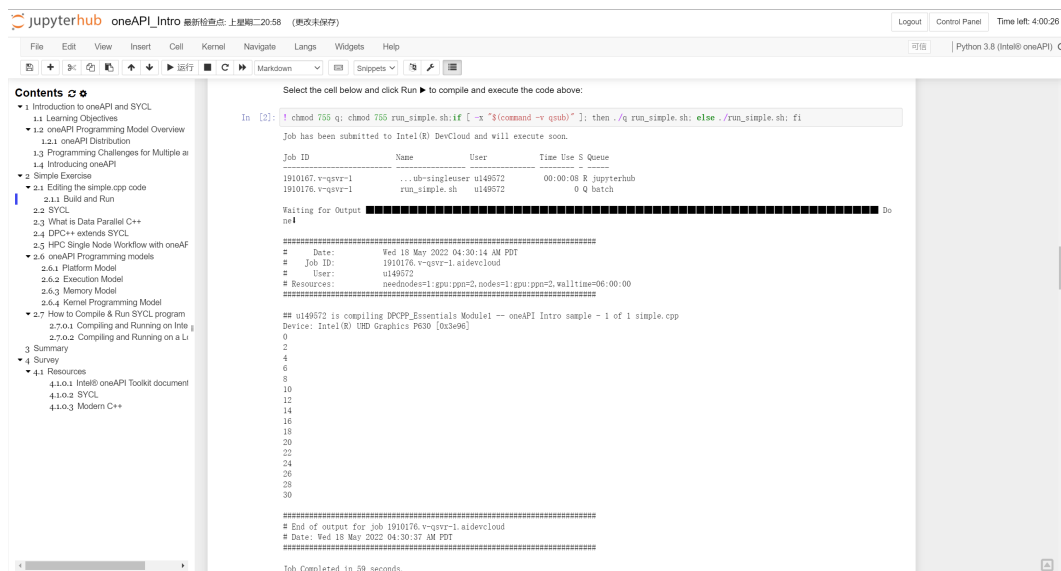
2022 年 6 月 15 日

目录

1 DevCloud 平台学习实践	2
2 CUDA 学习实践	3
2.1 使用 CUDA C/C++ 加速应用程序	3
2.1.1 学习目标	3
2.1.2 学习及实验	3
2.2 CUDA C/C++ 统一内存	5
2.2.1 学习目标	5
2.2.2 学习和实验	6
2.3 CUDA 加速应用程序的异步流和可视化分析	7
2.3.1 学习目标	7
2.3.2 学习和实验	7
2.4 学习证明	8
3 基于 CUDA 加速 KMeans 的尝试	9
4 总结	11

1 DevCloud 平台学习实践

在 Intel 的 DevCloud 平台上, 学习了解了 OneAPI 的基本知识, 并进行了 Introduction to oneAPI and SYCL 这个模块的实验, 成功使用 GPU 进行循环输出, 实验结果如图1.1所示。



```
1 chmod 755 q; chmod 755 run_sample.sh; if [ -x "$(command -v qsub)" ]; then ./q run_sample.sh; else ./run_sample.sh; fi
2
3 Job has been submitted to Intel(R) DevCloud and will execute soon.
4
5 Job ID      Name      User      Time Use S Queue
6 -----
7 1910167.v-qsvr-1    ...ub=singleuser u149572    00:00:08 8 Jupyterhub
8 1910176.v-qsvr-1    run_sample.sh  u149572    0 0 batch
9
10 Waiting for Output
11
12 #####
13 # Date:      Wed 18 May 2022 04:30:14 AM PDT
14 # Job ID:    1910176.v-qsvr-1.aidevcloud
15 # User:      u149572
16 # Resources: neednodes=1,gnu.ppn=2,nodes=1,gnu.ppn=2,walltime=06:00:00
17 #####
18 ## u149572 is compiling DPCPP_Essentials Module1 -- oneAPI Intro sample - 1 of 1 sample.cpp
19 Device: Intel(R) UHD Graphics P630 [0a3e96]
20 0
21 2
22 4
23 6
24 8
25 10
26 12
27 14
28 16
29 18
30 20
31 22
32 24
33 26
34 28
35 30
36
37 #####
38 # End of output for Job 1910176.v-qsvr-1.aidevcloud
39 # Date: Wed 18 May 2022 04:30:37 AM PDT
40 #####
41
42 Job Completed in 59 seconds.
```

图 1.1: OneAPI 实验结果

2 CUDA 学习实践

加速计算正在取代 CPU 计算，成为目前最佳的计算方法，其中通过 GPU 进行加速又是在目前一种主流的加速计算方式。在本次实验中，就基于 NVIDIA 的 GPU 学习平台，学习了有关 CUDA 的一些基本的原理和使用技巧。

CUDA 计算平台，提供了一种可以扩展 C、C++、Python 和 Fortran 等语言的编码范式，该凡是能够在世界上性能超强的并行处理器 NVIDIA GPU 上运行经过加速的大规模并行代码。CUDA 可以毫不费力地大幅度加速应用程序，具有适用于 DNN、BLAS、图形分析和 FFT 等更多运算的高度优化库生态系统，并且还附带了强大的命令行和可视化性能分析工具。

2.1 使用 CUDA C/C++ 加速应用程序

2.1.1 学习目标

1. 编写编译及运行既可以调用 CPU 函数又可以启动 GPU 核函数的 C/C++ 程序
2. 使用执行配置控制并行线程层次结构
3. 重构串行循环以在 GPU 上并行执行迭代
4. 分配和释放可用于 CPU 和 GPU 的内存
5. 处理 CUDA 代码生成的错误
6. 加速 CPU 应用程序

2.1.2 学习及实验

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，这些程序会调用能够利用 GPU 进行并行优化加速的核函数。一般的，并不会把程序的所有过程全部部署到 GPU 上进行运算，原因是 GPU 上更适合一些大规模的算术运算，而对于逻辑判断等运算并没有显著优势，并且还可能性能差于 CPU。因此需要在合适的地方将计算过程通过核函数的方式分发给 GPU，进行加速优化。

1. 核函数

如果想要让某个函数能够分配到 GPU 上执行，需要首先将这个函数声明为核函数，即使用 `__global__` 关键字，这个关键字表明以下函数将在 GPU 上运行，并且可以又 CPU 或 GPU 调用。通常的，将 CPU 上执行的代码成为主机代码，而将 GPU 上执行的代码成为设备代码。

而在调用和函数的时候，需要使用一种特殊的语法标记

```
GPUFunction<<< number_of_blocks, threads_per_block >>>()>
```

这表明，启用了 `number_of_blocks` 个线程块，而每个线程块内包含 `threads_per_block`，这个函数将会被分配到所有的线程上去并行执行。CUDA 的线程层次结构如图所示。

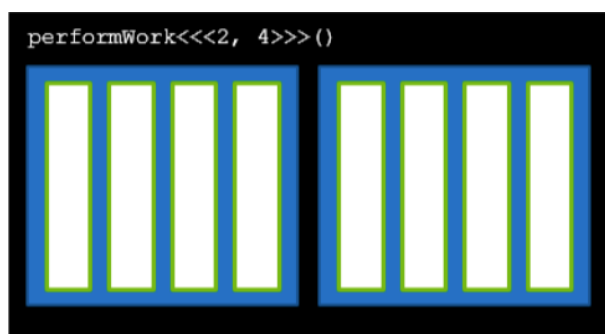


图 2.2: CUDA 线程层次结构

同时，由于核函数的启动方式是异步的，CPU 代码将不会等待 GPU 上的程序完全执行完成后再开始。因此在调用了核函数之后，一般需要进行一次同步，即需要等待所有的在 GPU 上执行的函数完成后，才能够继续串行去执行后续的代码。这就需要使用到 `cudaDeviceSynchronize()` 函数。

2. CUDA 线程层次结构变量

在上一小节中曾经介绍了 CUDA 的线程层次，即是通过线程块来进行组织的。每一个核函数可以使用多个线程块，并且可以指定线程块内线程的数量。因此我们就需要有一些变量，能够在函数内部确定当前的函数是在哪一个线程块中的哪一个线程上执行的，于是就提出了线程层次结构变量。

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`。

3. 加速 for 循环

对于一个循环而言，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。因此必须首先编写循环的单次迭代工作的核函数，由于核函数于其他正在运行的核函数无关，因此执行配置必须使用核函数执行正确的次数。

当然由于每一个线程块内的线程数量是有一个上限的，因此很有必要在调用核函数的时候，同时启用多个线程块，并且在每个线程块内启用多个线程。这时，就可以通过 `threadIdx.x` 和 `blockIdx.x` 来定位当前线程在整个任务中的一个为止，并根据此计算出当前线程所负责的任务。

接下来的练习就是使用多个线程块来加速 for 循环，由于各个线程之间是并行的，因此数字 0-9 的输出很有可能是乱序的。

4. CPU 和 GPU 内存分配

CUDA 的最新版本已经能够轻松的分配可用于 CPU 主机和任意数量 GPU 设备的内存。尽管现在有许多适用于内存管理并且可支持加速应用程序中最优性能的，但是现在要介绍的基础 CUDA 内存管理技术还是不能支持远超 CPU 应用程序的卓越性能。

其主要区别就是在申请内存的时候，使用 `cudaMallocManaged()` 函数，并在释放内存的时候，使用 `cudaFree()` 函数。

在实验中，要求尝试能够分配一种既能在 CPU 上访问又能在 GPU 上访问的内存，并且对于这个数组进行一个翻倍的操作。因此只需要使用 `cudaMallocManaged()` 和 `cudaFree()` 函数进行内存的申请和释放，并且将 `loop` 函数重构为核函数即可。

5. 数据与网格不匹配

当然，由于数据的规模不同，很有可能会出现数据规模与网格大小不匹配的现象，包括各个网格分配的任务数量不同，以及数据比网格大的时候需要采用循环划分的方式。

那么首先就是要注意，需要在每个线程执行核函数的时候判断当前的任务标号是否是合法的，并且如果数据集比网格的大小的时候，还会涉及到跨网格的数据访问，这就类似于之前在进行任务划分时的循环任务划分的方式。这时只需要注意修改一下循环的步幅即可，将之前的步幅修改为 $\text{gridDim.x} * \text{blockDim.x}$ ，而在每个线程内仍采用之前的寻址方式。如图2.3所示。

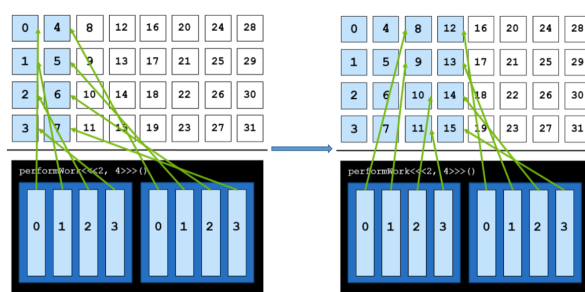


图 2.3: 任务划分

6. 错误处理

对于任何应用程序而言，错误的检查和处理都是至关重要的，加速 CUDA 代码中同样有错误处理的相关操作，并且 CUDA 的很多内置函数都以 `cudaError_t` 为返回值类型，该返回值可以用于检查调用函数的时候是否发生了错误。但是由于核函数必须声明为 `'void'` 类型，因此就不能够通过返回值来判断是否发生了错误，可以使用 CUDA 提供的方法 `cudaGetLastError` 方法获取最近的一次错误信息。对于内存申请，以及核函数执行之后的同步操作都需要使用 `cudaError_t` 的返回值来检查是否发生了错误。

2.2 CUDA C/C++ 统一内存

2.2.1 学习目标

1. 使用 Nsight Systems 命令行分析被加速的应用程序的性能
2. 利用对流多处理器的理解优化执行配置
3. 理解统一内存存在也错误和数据迁移方面的行为
4. 使用异步内存预取减少页错误和数据迁移以提高性能
5. 采用循环式的迭代开发加速应用恒旭的优化加速和部署

2.2.2 学习和实验

1. 使用 nsys 性能分析工具

nsys 是 NVIDIA 的命令行分析器，提供分析被加速的应用程序性能的强大功能。nsys 会执行使用 nvcc 编译的可执行程序，并打印应用程序的 GPU 活动的摘要输出、CUDA API 调用情况以及同意内存活动的相关信息。

使用 nsys profile 将会生成一个 qdrep 报告文件，可以增加指令选项 `-stats=true` 打印输出摘要信息。

2. 流多处理器

运行 CUDA 应用程序的 GPU 具有成为流多处理器（SM）的处理单元，在执行核函数的期间，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，通常可以选择线程块的数量数倍于指定的 GPU 上 SM 数量的网格大小来提高性能。并且 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，因此可以选择线程数量数倍于 32 的线程块大小来提升性能。如图2.4所示。

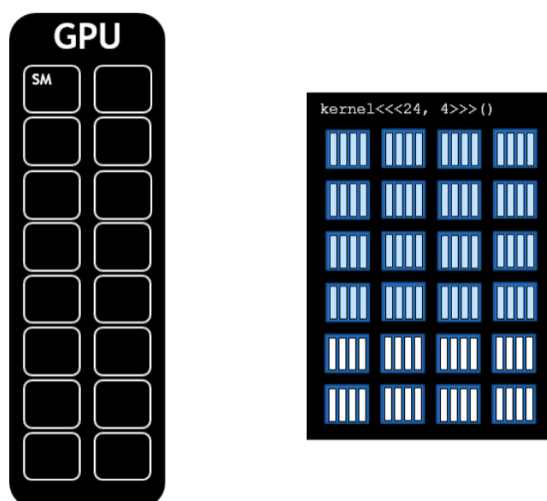


图 2.4: 流处理器

在 CUDA 中提供了 API 能够获取 SM 的数量，即可以调用函数 `cudaDeviceGetAttribute`，指定获取 `cudaDevAttrMultiProcessorCount` 属性的值，便可以得到流多处理器的数量。之后在声明线程的数量时，就可以将其数量指定为 SM 的数倍。

3. 统一内存

通常的使用函数 `cudaMallocManaged` 分配旨在供主机或设备代码使用的内存，这种方法的便利是能够自动实现内存的迁移并简化编程，但同样会带来内存迁移而产生的性能损失。因此有必要对 CUDA 的统一内存进行了解。

如果是在 CPU 上调用了 `cudaMallocManaged` 函数，则所申请的内存会出现在 CPU 上，而当 GPU 想要访问相关的数据的时候，就需要将这部分内存分配的 GPU 上。而如果在 GPU 上的核函数调用了 `cudaMallocManaged` 函数，则申请的内存会分配在 GPU 上，因此当 CPU 想要访问这部分数据的时候，又要将数据从 GPU 迁移的 CPU 上。

4. 异步内存预取

在分配 UM 的时候，最初可能并没有驻留在 CPU 或者 GPU 上，因此当某些线程执行工作的时候就可能发生页错误，因此就会触发内存的迁移，这会造成性能的损失。

在主机到设备和设备到主机的内存传输过程中，我们使用异步内存预取的技术来减少页错误和按需内存迁移成本。通过此技术，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。

但是由于预取往往会以更大的数据块来迁移数据，因此这种迁移的成本也是很高的，也要尽可能减少这种预取操作。一般的，当在运行之前已经知道数据访问需要且数据访问并未采用稀疏模式的时候，就可以考虑使用异步内存预取。

CUDA 也提供了相关的函数来实现异步内存预取，可以使用 `cudaMemPrefetchAsync` 来实现 CPU 和 GPU 之间的数据预取。

2.3 CUDA 加速应用程序的异步流和可视化分析

2.3.1 学习目标

1. 使用 Nsight Systems 直观描述由 GPU 加速的 CUDA 应用程序的时间表
2. 使用 Nsight Systems 识别和利用 CUDA 应用程序中的优化机会
3. 利用 CUDA 流在被加速的应用程序中并发执行核函数

2.3.2 学习和实验

1. 比较异步预取

在上一章的学习过程中我们了解了异步预取对于程序性能的重要影响，即如果已经提前明确了在程序的哪一部分会发生内存的迁移，则可以提前将这部分数据预取到 GPU 或者 CPU 上，来减少分页错误。在本次实验中，就可以通过使用 Nsight Systems 这个可视化的性能分析工具，来观察异步预取对于程序的性能影响。

这里对比了不进行异步预取和提前进行异步预取这两种方式，在进行向量加法时的访存差异。如图2.5所示。可以看到，由于在进行向量加法之前，体检将数据预取到了 GPU，因此在执行核函数的时候，并没有发生内存的迁移。

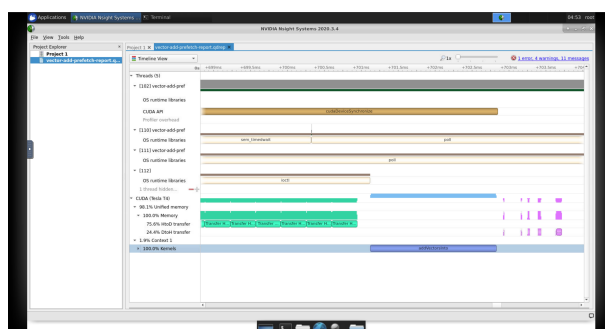


图 2.5: 比较异步预取

2. 并发 CUDA 流

在 CUDA 编程中,流是由按照顺序执行的一系列命令构成的。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。除了默认流意外,CUDA 还可以创建并使用非默认的 CUDA 流,此举可以支持执行多个操作。对于一个给定的流,其中的所有操作都会顺序执行,不同的非默认流之间是并行执行的,默认流拥有阻断能力,即他会等待所有其他的流执行完成后才开始执行自己的操作,并在此时阻塞其他的非默认流。

可以使用 `cudaStreamCreate` 来创建一个流,并且将得到的流作为一个参数传递给核函数 `someKernel <<< number_of_blocks, threads_per_block, 0, stream >>>()`。

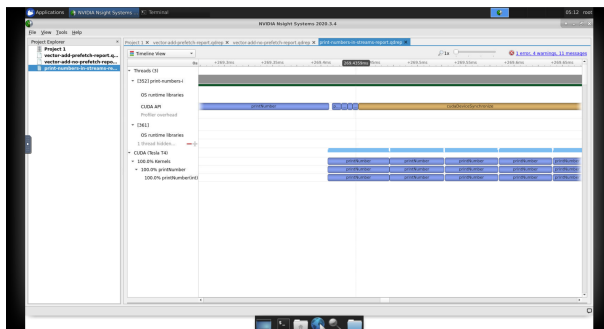


图 2.6: 并发 CUDA 流

2.4 学习证明



图 2.7: 学习证书

3 基于 CUDA 加速 KMeans 的尝试

在本次实验中,学习完 CUDA 的基本操作之后,尝试着对于期末选题 KMeans 使用 CUDA 进行优化。由于本次学习过程中了解的内容相对比较基础,因此只是对 KMeans 进行基础的 CUDA 优化,并没有增加其他的优化技巧。其中的核心函数包含以下两个,一个是寻找最近的质心 `find_nearest_cluster()`,一个是计算和质心之间的距离 `compute_delta()`。下面附上这两个核心函数的代码。

```

1  __global__ static
2  void find_nearest_cluster(int numCoords, int numObjs, int numClusters, float
   *objects, float *deviceClusters, int *membership, int *intermediates)
3  {
4      extern __shared__ char sharedMemory[];
5      unsigned char *membershipChanged = (unsigned char *)sharedMemory;
6      float *clusters = deviceClusters;
7      membershipChanged[threadIdx.x] = 0;
8      int objectId = blockDim.x * blockIdx.x + threadIdx.x;
9      if (objectId < numObjs) {
10         int index, i;
11         float dist, min_dist;
12         index = 0;
13         min_dist = euclid_dist_2(numCoords, numObjs, numClusters,
14                                 objects, clusters, objectId, 0);
15         for (i=1; i<numClusters; i++) {
16             dist = euclid_dist_2(numCoords, numObjs, numClusters,
17                                 objects, clusters, objectId, i);
18             if (dist < min_dist) {
19                 min_dist = dist;
20                 index = i;
21             }
22         }
23         if (membership[objectId] != index) {
24             membershipChanged[threadIdx.x] = 1;
25         }
26         membership[objectId] = index;
27         __syncthreads();
28         for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
29             if (threadIdx.x < s) {
30                 membershipChanged[threadIdx.x] +=
31                     membershipChanged[threadIdx.x + s];
32             }
33             __syncthreads();
34         }
35         if (threadIdx.x == 0) {
36             intermediates[blockIdx.x] = membershipChanged[0];
37         }
38     }
39 }

```

```
1  __global__ static
2  void compute_delta(int *deviceIntermediates, int numIntermediates, int
   numIntermediates2)
3  {
4      extern __shared__ unsigned int intermediates[];
5      intermediates[threadIdx.x] =
6          (threadIdx.x < numIntermediates) ? deviceIntermediates[threadIdx.x] : 0;
7      __syncthreads();
8      for (unsigned int s = numIntermediates2 / 2; s > 0; s >>= 1) {
9          if (threadIdx.x < s) {
10             intermediates[threadIdx.x] += intermediates[threadIdx.x + s];
11         }
12         __syncthreads();
13     }
14     if (threadIdx.x == 0) {
15         deviceIntermediates[0] = intermediates[0];
16     }
17 }
```

实验结果表明，使用了 CUDA 优化的 KMeans 能够比普通的串行算法取得超过 30 倍的性能提升，因此可以证明 CUDA 具有强大的并行计算能力。关于 CUDA 的其他性能优化方式，将在后续的实验继续探究。

4 总结

在本次实验中，首先在 Intel 的 DevCloud 平台上学习了有关 OneAPI 的相关知识。然后主要是学习了解 CUDA 的相关知识，学习了如何声明和调用核函数，如何理解 CUDA 的线程层次结构，并且基于此尝试优化 for 循环。更进一步了解了 CUDA 的统一内存管理，体会到了内存在 GPU 和 CPU 之间迁移的性能损失，并且可以通过异步预取的方式提高降低分页错误提高性能。最后还学习了解了如何使用 NVIDIA 开发的可视化性能分析工具，对于 CUDA 程序进行分析，并针对特定的指标进行相关的优化。最终利用本次实验学习到的知识，尝试了对于 KMeans 算法的基础优化，并取得了不错的性能提升。本次实验的相关代码和文档已经上传至[GitHub](#)。