计算机系统结构
实验报告

姓　　名	严昕宇
学　　号	20121802
实验名称	Linux/Windows 下 CUDA 安装及矩阵乘法实现
日　　期	2023 年 4 月 24 日

目录

一　实验环境

表 1: 实验环境

实验设备	Intel Xeon Platinum 8255C
	NVIDIA Tesla T4
操作系统	Ubuntu 22.04 LTS
开发语言	C、CUDA
IDE	CLion 2023.1
编译器	GCC、NVCC

二　CUDA 环境配置

1　检查显卡驱动

使用**代码 1**查看硬件设备，即确认是否有 NVIDIA 显卡。

代码 1: 检查显卡

```
1  lspci | grep -i nvidia
```

如果存在，则使用**代码 2**查看显卡的相关信息。

代码 2: 查看显卡信息

```
1  nvidia-smi
```

```
yanxinyu@VM-16-16-ubuntu:~$ lspci | grep -i nvidia
00:08.0 3D controller: NVIDIA Corporation TU104GL [Tesla T4] (rev a1)
yanxinyu@VM-16-16-ubuntu:~$ nvidia-smi
Mon Apr 24 19:15:42 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.142.00   Driver Version: 450.142.00   CUDA Version: 11.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  GRID T4-8C          On   | 00000000:00:08.0 Off |                    0 |
| N/A   N/A    P0    N/A /  N/A |   1104MiB /  8192MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

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

图 1: 显卡信息

如果出现类似于上图的效果，说明系统中已经安装显卡驱动了。通常 Ubuntu 在系统安装完成后会自动安装显卡驱动，但是如果没有安装，则需要按照以下步骤操作，手动安装显卡驱动。

2 安装显卡驱动

2.1 方法一：使用标准 Ubuntu 仓库自动化安装

Ubuntu-drivers 是 Ubuntu 系统的标准驱动管理软件，可以自动检测当前机器上的显卡型号及可安装的驱动型号。使用下面的命令安装：

代码 3: 安装 Ubuntu-drivers

```
sudo apt-get install ubuntu-drivers-common
```

安装完毕后，终端输入 **ubuntu-drivers** 查看是否安装成功，正常会显示如下内容：

```
yanxinyu@VM-16-16-ubuntu:~$ ubuntu-drivers
Usage: ubuntu-drivers [OPTIONS] COMMAND [ARGS]...

Options:
  --gpgpu              gpgpu drivers
  --free-only          Only consider free packages
  --package-list PATH  Create file with list of installed packages (in install
                       mode)
  --no-oem             Do not include OEM enablement packages (these enable an
                       external archive)  [default: False]
  -h, --help           Show this message and exit.

Commands:
  autoinstall  Deprecated, please use "install" instead
  debug        Print all available information and debug data about drivers.
  devices      Show all devices which need drivers, and which packages
               apply...
  install      Install a driver [driver[:version][,driver[:version]]]
  list         Show all driver packages which apply to the current system.
  list-oem     Show all OEM enablement packages which apply to this system
```

图 2: ubuntu-drivers 的返回结果

在终端中输入 **ubuntu-drivers devices**，驱动管理软件会自动检测本机的显卡，并给出可以安装的驱动型号，如下图所示：

```
== /sys/devices/pci0000:00/0000:00:08.0 ==
modalias : pci:v000010DEd00001EB8sv000010DEsd0000139Bbc03sc02i00
vendor   : NVIDIA Corporation
model    : TU104GL [Tesla T4]
manual_install: True
driver   : nvidia-driver-510 - distro non-free
driver   : nvidia-driver-418-server - distro non-free
driver   : nvidia-driver-450-server - distro non-free
driver   : nvidia-driver-470-server - distro non-free
driver   : nvidia-driver-530 - distro non-free recommended
driver   : nvidia-driver-470 - distro non-free
driver   : nvidia-driver-515 - distro non-free
driver   : nvidia-driver-525 - distro non-free
driver   : nvidia-driver-515-server - distro non-free
driver   : nvidia-driver-525-server - distro non-free
driver   : xserver-xorg-video-nouveau - distro free builtin
```

图 3: 可以安装的驱动型号

其中会有一个版本的驱动被标注成 **recommended**，即驱动管理软件建议安装的版本。该例中系统已连接 Tesla T4 显卡，推荐驱动版本为 nvidia-530。

接着使用**代码 4**自动安装

代码 4: 安装 Ubuntu-drivers

```
sudo ubuntu-drivers autoinstall
```

安装完成后，使用 **reboot** 命令重启系统，再使用 **nvidia-smi** 命令查看显卡相关信息，此时已经成功安装 nvidia-530 版本的驱动。

2.2 方法二：使用 PPA 仓库自动化安装

使用图形驱动程序 PPA 存储库，允许我们安装 NVIDIA 驱动程序，包括 Beta 版的驱动程序，但这种方法存在**不稳定**的风险，此处**仍建议第一种方法**。

首先，使用**代码 5**将 ppa:graphics-drivers/ppa 存储库添加到系统中，并更新软件包。

代码 5: 添加镜像源

```
sudo add-apt-repository ppa:graphics-drivers/ppa
sudo apt-get update
```

接下来，按照使用标准库时的操作，使用 **ubuntu-drivers devices** 命令识别显卡和推荐的驱动程序。

最后，直接使用 apt 命令安装希望安装的驱动版本，此处以 nvidia-530 版本为例，代码如下：

代码 6: 检查可安装的驱动

```
sudo apt install nvidia-driver-530
```

安装完毕后，同理使用 **nvidia-smi** 命令查看显卡相关信息。

3 安装 CUDA Toolkit

需要注意，Ubuntu 20.04 中默认 GCC 和 G++ 版本太高，会导致 CUDA 安装和运行过程中存在问题，因此需要先降低其版本，方法如下：

代码 7: 安装并配置低版本 GCC 和 G++

```
sudo apt-get install gcc-7 g++-7

sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-7 9
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-9 1

sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-7 9
sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-9 1

sudo update-alternatives --display gcc
sudo update-alternatives --display g++
```

结果如**图 4** 所示。接着通过 Ubuntu 软件源，安装 CUDA Toolkit 软件，命令如下：

代码 8: 安装 CUDA Toolkit

```
sudo apt install nvidia-cuda-toolkit
```

```
yanxinyu@VM-16-16-ubuntu:~$ sudo update-alternatives --display gcc
gcc - auto mode
  link best version is /usr/bin/gcc-7
  link currently points to /usr/bin/gcc-7
  link gcc is /usr/bin/gcc
/usr/bin/gcc-7 - priority 9
/usr/bin/gcc-9 - priority 1
yanxinyu@VM-16-16-ubuntu:~$ sudo update-alternatives --display g++
g++ - auto mode
  link best version is /usr/bin/g++-7
  link currently points to /usr/bin/g++-7
  link g++ is /usr/bin/g++
/usr/bin/g++-7 - priority 9
/usr/bin/g++-9 - priority 1
```

图 4: 系统中存在的 GCC、G++

这种方式安装最为简单，是首选方案。当然其安装的 CUDA Toolkit 版本往往不是最新版本。其他安装方法的具体实现，可参考网页：https://www.cnblogs.com/klchang/p/14353384.html

三 CUDA 下的矩阵乘法实现

在矩阵乘法中，进行了不同数据的大量相同计算操作（相乘并累加），这种计算是特别适合使用 GPU 来计算, 因为 GPU 拥有大量简单重复的计算单元, 通过并行就能极大的提高计算效率。在 CUDA 中常规实现使用 Global Memory，思路如下：

- 在 Global Memory 中分别为矩阵 A、B、C 分配存储空间（cudaMalloc）；
- 由于矩阵 C 中每个元素的计算均相互独立, NVIDIA GPU 采用的 SIMT 的体系结构来实现并行计算, 因此让每个 Thread 对应矩阵 C 中 1 个元素的计算；
- 配置 gridSize 和 blockSize，其均有 x(列向)、y(行向)

每个 CUDA 核心线程需要执行的操作为：从矩阵 A 中读取一行向量 (长度为 n), 从矩阵 B 中读取一列向量 (长度为 n), 对这两个向量做点积运算 (单层 n 次循环的相乘累加), 最后将结果写回矩阵 C.

具体代码实现见**代码 9**, 为了测试 CUDA 下的矩阵乘法效果, 对 1000×1000、2000×2000、3000×3000、4000×4000、5000×5000 矩阵乘法分别进行了测试，测试结果如下：

```
yanxinyu@VM-16-6-ubuntu:~$ ./Matrix.out 1000
Device Name : GRID T4-8C.
totalGlobalMem : 0.
sharedMemPerBlock : 49152.
regsPerBlock : 65536.
warpSize : 32.
memPitch : 2147483647.
maxThreadsPerBlock : 1024.
maxThreadsDim[0 - 2] : 1024 1024 64.
maxGridSize[0 - 2] : 2147483647 65535 65535.
totalConstMem : 65536.
major.minor : 7.5.
clockRate : 1590000.
textureAlignment : 512.
deviceOverlap : 1.
multiProcessorCount : 40.
GPU memory: 7.629395e+00 MB
GPU time: 5.441000 ms
CPU time: 4167.170000 ms
Max error: 1.19207e-07 Average error: 8.52059e-10
```

图 5: 1000×1000 矩阵乘法

```
yanxinyu@VM-16-6-ubuntu:~$ ./Matrix.out 2000
Device Name : GRID T4-8C.
totalGlobalMem : 0.
sharedMemPerBlock : 49152.
regsPerBlock : 65536.
warpSize : 32.
memPitch : 2147483647.
maxThreadsPerBlock : 1024.
maxThreadsDim[0 - 2] : 1024 1024 64.
maxGridSize[0 - 2] : 2147483647 65535 65535.
totalConstMem : 65536.
major.minor : 7.5.
clockRate : 1590000.
textureAlignment : 512.
deviceOverlap : 1.
multiProcessorCount : 40.
GPU memory: 3.051758e+01 MB
GPU time: 55.427000 ms
CPU time: 52113.124000 ms
Max error: 1.19209e-07 Average error: 5.95903e-10
```

图 6: 2000×2000 矩阵乘法

```
yanxinyu@VM-16-6-ubuntu:~$ ./Matrix.out 3000
Device Name : GRID T4-8C.
totalGlobalMem : 0.
sharedMemPerBlock : 49152.
regsPerBlock : 65536.
warpSize : 32.
memPitch : 2147483647.
maxThreadsPerBlock : 1024.
maxThreadsDim[0 - 2] : 1024 1024 64.
maxGridSize[0 - 2] : 2147483647 65535 65535.
totalConstMem : 65536.
major.minor : 7.5.
clockRate : 1590000.
textureAlignment : 512.
deviceOverlap : 1.
multiProcessorCount : 40.
GPU memory: 6.866455e+01 MB
GPU time: 192.598000 ms
CPU time: 217559.350000 ms
Max error: 8.68492e-08 Average error: 4.84092e-10
```

图 7: 3000×3000 矩阵乘法

```
yanxinyu@VM-16-6-ubuntu:~$ ./Matrix.out 4000
Device Name : GRID T4-8C.
totalGlobalMem : 0.
sharedMemPerBlock : 49152.
regsPerBlock : 65536.
warpSize : 32.
memPitch : 2147483647.
maxThreadsPerBlock : 1024.
maxThreadsDim[0 - 2] : 1024 1024 64.
maxGridSize[0 - 2] : 2147483647 65535 65535.
totalConstMem : 65536.
major.minor : 7.5.
clockRate : 1590000.
textureAlignment : 512.
deviceOverlap : 1.
multiProcessorCount : 40.
GPU memory: 1.220703e+02 MB
GPU time: 543.256000 ms
CPU time: 559722.052000 ms
Max error: 1.19209e-07 Average error: 4.19135e-10
```

图 8: 4000×4000 矩阵乘法

```
yanxinyu@VM-16-6-ubuntu:~$ ./Matrix.out 5000
Device Name : GRID T4-8C.
totalGlobalMem : 0.
sharedMemPerBlock : 49152.
regsPerBlock : 65536.
warpSize : 32.
memPitch : 2147483647.
maxThreadsPerBlock : 1024.
maxThreadsDim[0 - 2] : 1024 1024 64.
maxGridSize[0 - 2] : 2147483647 65535 65535.
totalConstMem : 65536.
major.minor : 7.5.
clockRate : 1590000.
textureAlignment : 512.
deviceOverlap : 1.
multiProcessorCount : 40.
GPU memory: 1.907349e+02 MB
GPU time: 1030.523000 ms
CPU time: 1132998.479000 ms
Max error: 1.02807e-07 Average error: 3.7491e-10
```

图 9: 5000×5000 矩阵乘法

表 2: 矩阵乘法

矩阵规模	CPU 时间 (ms)	GPU 时间 (ms)	加速比 (%)
1000×1000	4167.17	5.441	765.88
2000×2000	52113.124	55.427	940.21
3000×3000	217559.35	192.598	1129.60
4000×4000	559722.052	543.235	1030.35
5000×5000	1132998.479	1030.523	1099.44

从输出的设备信息可以看到，本次用来计算的显卡 Tesla T4，每个 Block 最大支持 1024 个线程，最大 GridSize 支持 $2147483647 \times 65535 \times 65535$，性能较强。

以计算 3000x3000 的矩阵乘法为例，CPU（Intel Xeon Platinum 8255C）拥有十颗核心，使用多线程计算需要 21.7s 左右。但如果使用 GPU 只需要 0.18s，获得了 1100 倍的加速，效果惊人。当然对于更高性能的 GPU 和优化的 CUDA 程序，加速比也能更高。

四　实验感想

在实验二中，我通过使用 OpenMP 实现了矩阵乘法的简单并行，获得 10 倍左右的加速。在本次实验中，我接着使用 CUDA 完成了矩阵乘法，获得 1000 倍左右的加速，两者的差距十分巨大，足以彰显 GPU 在并行计算中的优势地位。但是，GPU 也会存在很多问题，如 GPU 浮点数运算的精度很差，在查阅资料后我知道可以使用了 Kahan's Summation Formula，在一定程度上解决 CUDA 运算 float 精度不够的情况。

五　附录

代码 9: 程序代码

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <omp.h>
#include "cuda.h"
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#define MATRIX_SIZE 1000
#define BLOCK_SIZE 16
int DevicedChoosed = 0;

void printDeviceProp(const cudaDeviceProp &prop) {
    printf("Device Name : %s.\n", prop.name);
    printf("totalGlobalMem : %d.\n", prop.totalGlobalMem);
    printf("sharedMemPerBlock : %d.\n", prop.sharedMemPerBlock);
    printf("regsPerBlock : %d.\n", prop.regsPerBlock);
    printf("warpSize : %d.\n", prop.warpSize);
    printf("memPitch : %d.\n", prop.memPitch);
    printf("maxThreadsPerBlock : %d.\n", prop.maxThreadsPerBlock);
    printf("maxThreadsDim[0 - 2] : %d %d %d.\n", prop.maxThreadsDim[0], prop.
        maxThreadsDim[1], prop.maxThreadsDim[2]);
    printf("maxGridSize[0 - 2] : %d %d %d.\n", prop.maxGridSize[0], prop.
        maxGridSize[1], prop.maxGridSize[2]);
    printf("totalConstMem : %d.\n", prop.totalConstMem);
    printf("major.minor : %d.%d.\n", prop.major, prop.minor);
    printf("clockRate : %d.\n", prop.clockRate);
    printf("textureAlignment : %d.\n", prop.textureAlignment);
    printf("deviceOverlap : %d.\n", prop.deviceOverlap);
    printf("multiProcessorCount : %d.\n", prop.multiProcessorCount);
}

//CUDA 初始化
bool InitCUDA() {
    int count;
    //取得支持Cuda的装置的数目
    cudaGetDeviceCount(&count);
    if (count == 0) {
        fprintf(stderr, "There is no device.\n");
        return false;
    }
    int i;
```

```
42      for (i = 0; i < count; i++) {
43          cudaDeviceProp prop;
44          cudaGetDeviceProperties(&prop, i);
45          //打印设备信息
46          printDeviceProp(prop);
47          if (cudaGetDeviceProperties(&prop, i) == cudaSuccess) {
48              if (prop.major >= 1) {
49                  break;
50              }
51          }
52      }
53
54      if (i == count) {
55          fprintf(stderr, "There is no device supporting CUDA 1.x.\n");
56          return false;
57      }
58      cudaSetDevice(i);
59      DevicedChoosed = i;
60      return true;
61  }
62
63  void matMultCPU(const float *a, const float *b, float *c, int n) {
64  #pragma omp parallel for
65      for (int i = 0; i < n; i++) {
66          for (int j = 0; j < n; j++) {
67              double t = 0;
68              for (int k = 0; k < n; k++) {
69                  t += (double) a[i * n + k] * b[k * n + j];
70              }
71              c[i * n + j] = t;
72          }
73      }
74  }
75
76  //GPU并行计算矩阵乘法
77  __global__ void matMultCUDAKernel1(const float *a, const float *b, float *c, int
        n) {
78      //计算这个 thread 应该计算的 row 和 col
79      const int col = blockIdx.x * blockDim.x + threadIdx.x;
80      const int row = blockIdx.y * blockDim.y + threadIdx.y;
81
82      int i;
83      //计算矩阵乘法 Kahan' s Summation Formula
84      if (row < n && col < n) {
85          float t = 0;
86          float y = 0;
87          for (i = 0; i < n; i++) {
```

```
            float r;

            y -= a[row * n + i] * b[i * n + col];
            r = t - y;
            y = (r - t) + y;
            t = r;
        }
        c[row * n + col] = t;
    }
}

void genMat(float *arr, int n) {
    int i, j;

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            arr[i * n + j] = (float) rand() / RAND_MAX + (float) rand() / (
                RAND_MAX * RAND_MAX);
        }
    }

}


typedef struct Error {
    float max;
    float average;
} Error;

Error accuracyCheck(const float *a, const float *b, int n) {
    Error err;
    err.max = 0;
    err.average = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (b[i * n + j] != 0) {
                //fabs求浮点数x的绝对值
                float delta = fabs((a[i * n + j] - b[i * n + j]) / b[i * n + j])
                    ;
                if (err.max < delta) err.max = delta;
                err.average += delta;
            }
        }
    }
    err.average = err.average / (n * n);
    return err;
}
```

```
133
134
135 int main(int argc, char **argv) {
136
137     //CUDA 初始化
138     if (!InitCUDA()) return 0;
139     cudaDeviceProp prop;
140     cudaGetDeviceProperties(&prop, DevicedChoosed);
141     //定义矩阵
142     float *a, *b, *c, *d;
143     int n = MATRIX_SIZE;
144     if (argc >= 2) n = atoi(argv[1]) > 0 ? atoi(argv[1]) : MATRIX_SIZE;
145
146     //分配host内存
147     cudaMallocHost((void **) &a, sizeof(float) * n * n);
148     cudaMallocHost((void **) &b, sizeof(float) * n * n);
149     cudaMallocHost((void **) &c, sizeof(float) * n * n);
150     d = (float *) malloc(sizeof(float) * n * n);
151
152     genMat(a, n);
153     genMat(b, n);
154
155     float *cuda_a, *cuda_b, *cuda_c;
156     clock_t start, stop;
157     //分配GPU上的内存
158     cudaMalloc((void **) &cuda_a, sizeof(float) * n * n);
159     cudaMalloc((void **) &cuda_b, sizeof(float) * n * n);
160     cudaMalloc((void **) &cuda_c, sizeof(float) * n * n);
161
162     //拷贝数据至GPU内存
163     cudaMemcpy(cuda_a, a, sizeof(float) * n * n, cudaMemcpyHostToDevice);
164     cudaMemcpy(cuda_b, b, sizeof(float) * n * n, cudaMemcpyHostToDevice);
165     start = clock();
166     //调用核函数计算
167     dim3 blockSize(BLOCK_SIZE, BLOCK_SIZE, 1);
168     dim3 gridSize((n + BLOCK_SIZE - 1) / BLOCK_SIZE, (n + BLOCK_SIZE - 1) /
        BLOCK_SIZE, 1);
169     matMultCUDAKernel1 << < gridSize, blockSize >> > (cuda_a, cuda_b, cuda_c, n)
        ;
170
171     //计算结果复制回主存，隐式调用同步函数
172     cudaMemcpy(c, cuda_c, sizeof(float) * n * n, cudaMemcpyDeviceToHost);
173     stop = clock();
174     //释放GPU上的内存
175     cudaFree(cuda_a);
176     cudaFree(cuda_b);
177     cudaFree(cuda_c);
```

```
178     //GPU memory
179     printf("GPU memory: %e MB\n", (double) (n * n * 8) / (1024. * 1024.));
180     //GPU time
181     printf("GPU time: %3f ms\n", (double) (stop - start) / CLOCKS_PER_SEC *
            1000.0);
182     //CPU time
183     start = clock();
184     matMultCPU(a, b, d, n);
185     stop = clock();
186     printf("CPU time: %3f ms\n", (double) (stop - start) / CLOCKS_PER_SEC *
            1000.0);
187     //精度检测
188     Error error;
189     error = accuracyCheck(c, d, n);
190     printf("Max error: %g Average error: %g\n", error.max, error.average);
191
192     return 0;
193 }
```