分布式统计计算

4. GPU 和 CUDA

李舰

华东师范大学

2018-12-10

目 录

目 录

GPU 计算

- GPU 与并行
  - GPU 是图形处理器（Graphics Processing Unit）的简称，又称显示芯片，是一种专门用来进行图形计算的微处理器。
  - 图像处理的很多方法（比如图像渲染、滤镜、卷积）都需要大量的浮点运算。
  - 由于每个像素点都可以独立处理，因此 GPU 对并行计算的能力要求很高。
- GPU 编程框架
  - NVIDIA 公司于 2007 年推出的 CUDA 是目前最流行的 GPU 并行计算框架。
  - AMD 于 2010 年在收购 ATI 之后，将其并行计算框架 ATI Stream 更名为 AMD APP。
  - 2008 年，苹果提出了 OpenCL 规范，旨在提供一个通用的开放 API，在此基础上开发 GPU 通用计算软件，可以适应多种 GPU 硬件。

浮点和精度

- 单精度与双精度
  - 浮点运算指的是实数运算，由于计算机只能存储整数，所以实数都是约数，需要指定精度。
  - 单精度的实数（Single Real）在内存中占 4 个字节，可以保存 8 位有效数字（1 位整数加 7 位小数），在一些编程语言中常被称为浮点型（float）数据。
  - 双精度的实数（Double Real）在内存中占 8 个字节，可以保存 16 位有效数字，在一些编程语言中常被称为双精度型（double）数据。
- 显卡的精度
  - 不同的显卡可能具有不同的精度，有些显卡可以在两种模式间切换。
  - 双精度的显卡通常比较耗电。
  - 一般来说科学计算对双精度要求比较高，游戏对单精度要求比较高。

NVIDIA 与 CUDA

- NVIDIA 简介
  - 也称“英伟达”，创立于 1993 年 1 月，是一家以设计图形处理器为主的半导体公司。
  - 作为一家无芯片 IC 半导体设计公司，NVIDIA 于自己的实验室研发芯片，但将芯片制造工序分包给其他厂商，例如台积电、IBM、意法半导体和联华电子。
  - NVIDIA 最出名的产品线是为个人与游戏玩家所设计的 GeForce 系列，为专业工作站而设计的 Quadro 系列，以及为服务器和高效运算而设计的 Tesla 系列。
- CUDA 简介
  - 2007 年，NVIDIA 推出 GPU 编程接口 CUDA，极大地降低了计算成本。
  - 基于 C/C++，提供了一套编程接口和模板库。
  - 利用 CUDA 技术，可以将显卡所有的内处理器串通起来，成为线程处理器去解决数据密集的计算。
  - 深度学习利用 CUDA 技术产生了突破性的进展。

目 录

CUDA 简介

- CUDA
  - CUDA（Compute Unified Device Architecture，统一计算架构）是由 NVIDIA 所推出的一种集成技术，是该公司对于 GPGPU 的正式名称。
  - NVIDIA 的硬件和 CUDA 编程是基于共享线程和共享内存的（此处共享的内存是显卡内存而不是 CPU 可以存取的内存，称为全局内存）。
- 相关术语
  - 主机（Host）指代 CPU 和内存，设备（Device）指代 GPU 和显存。
  - GPU 编程里一般把共享的显卡内存称为全局内存，而“共享内存”通常指的是一种缓存。
  - GPU（设备）上运行的函数是 CPU（主机）通过调用程序员编写的 kernel 函数来启动的。
  - CUDA 函数库包括了在 device 上为数据对象分配空间，以及在 host 与 device 之间互相传递数据的程序。

CUDA 基础知识

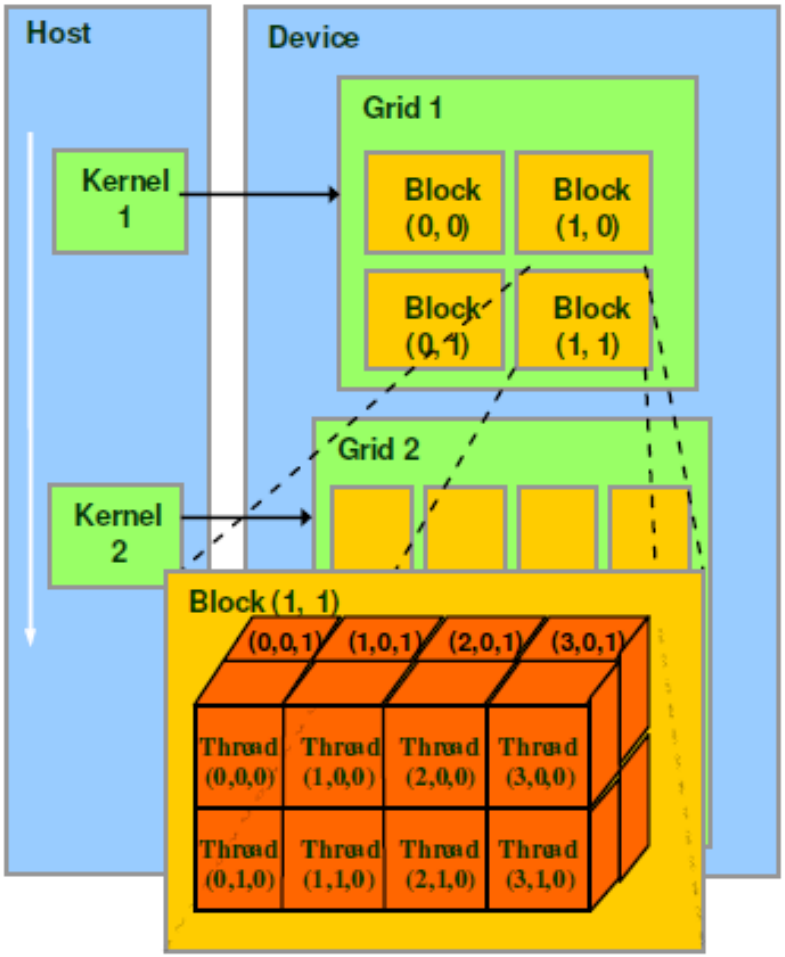
- 基础计算单元
  - CUDA 应用了单指令多线程（SIMT）的并行模式。
  - 最基础的计算单元称为核（Core），也称为流式处理器（Streaming Processor，简称 SP），每一个核包含了一个逻辑计算单元（ALU）和一个浮点计算单元（FPU）。
  - 多个核集在一起被称为多流处理器（Streaming Multiprocessor，简称 SM）。
  - 一个 GPU 由多个 SM 构成，每个 SM 独立运行，但共享 GPU 全局内存。
- 相关术语
  - GPU（称为 device）上运行的函数是 CPU（称为 host）通过调用程序员编写的 kernel 函数来启动的。
  - GPU 编程里一般把共享的显卡内存称为全局内存，而“共享内存”通常指的是一种缓存。
  - CUDA 函数库包括了在 device 上为数据对象分配空间，以及在 host 与 device 之间互相传递数据的程序。

CUDA 的并行架构

- 基础结构
  - 逻辑计算单元（ALU，绿色表示）是进行整数运算和逻辑运算的模块。
  - 存储单元（橙色表示），包括内存和缓存。
  - 控制单元（黄色表示），处理控制逻辑。

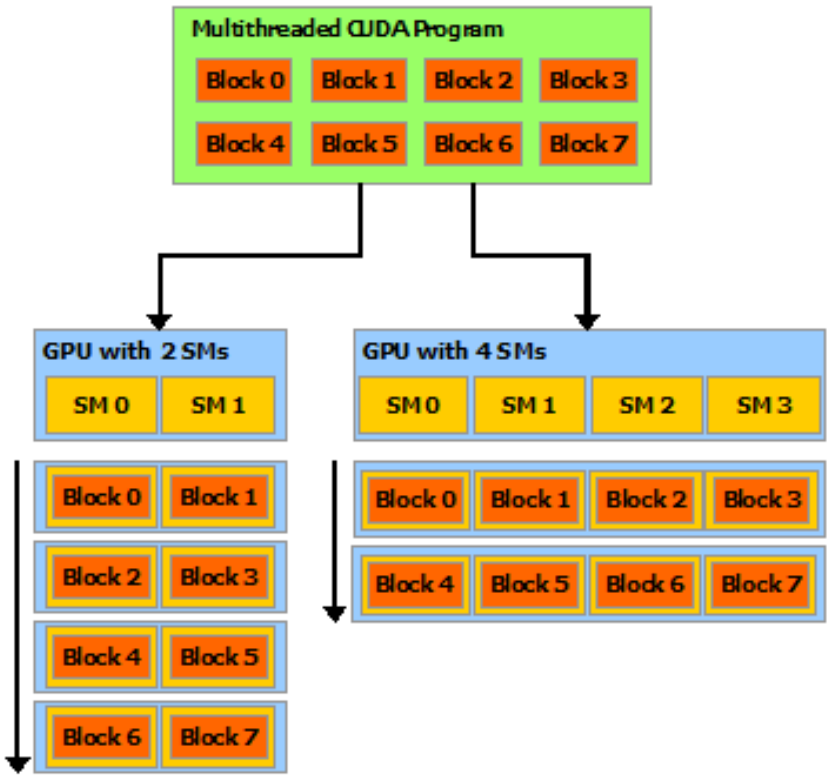
- CPU 和 GPU 的差异
  - CPU 拥有较少的核，占据芯片的小部分，而更多的区域被用来加速那些少量核的控制器和缓存。大量的缓存可以缩小因核和内存距离增加而带来的延迟。
  - GPU 的空间分配完全不同，芯片上的大多数地方被分配给了大量组织成 SM 的运算单元和共享的控制单元。

线程的层次



- 层次结构
  - 线程（thread）：一个 CUDA 的并行程序会被以许多个 threads 来执行。
  - 线程块（block）：多个线程会组成一个 block，同一个 block 中的线程可以同步，也可以通过共享内存通信。
  - 网格（grid）：多个 blocks 则会再构成 grid。
- 关系介绍
  - 线程和线程块的组织方式都可以是一维的，二维或者三维的。

block 的分配机制



- 分配规则
  - 整个 block 都会在一个 SM 中，一个 SM 上可以运行多个 block。
  - 同一个 block 中的线程可以进行屏障同步，不同 SM 的线程之间不能相互同步。
  - 同一个 block 中的线程可以存取一个能被程序员控制的、名为“共享内存”的缓存。

SIMT 机制

- 线程束（Warp）
  - 在计算的过程中，block 会被分解成某个大小（warp size，一般是 32 个线程）的线程束（Warp）。
  - 每个 warp 由一个特定的 SM 执行，这些 SM 的控制单元指挥其所有核同时在一个 warp 的每个线程中执行同一个命令，因此称为单指令多线程（SIMT）。
- 基于 warp 的调度
  - 线程调度是基于 warp 的，一些核空闲时，会以 32 个为一组进入空闲，此时会运行线程调度，硬件会找到一个新的 warp 来运行到这 32 个核上。
  - 一个 warp 中的线程步调一致，会执行相同的指令（根据线程编号的不同执行不同的任务）。
  - 如果一个 warp 中的线程执行了 `if/then/else` 分支，那么一些线程必须等待其他线程结束，此时的代码更接近串行而不是并行，会降低性能。这称为“线程分支”，是性能杀手。同一个 block 的不同 warp 发生分支不会有问题。

CUDA 编程经验

- Grid 配置
  - 如果 block 设置得太大，可能有的 SM 空闲。而且屏障同步只能在 block 层面进行，block 越大，屏障延迟越多。因此我们通常需要设置较小的 block。
  - 如果使用“共享内存”，也只能在 block 层面，有效地使用“共享内存”又意味着使用更大的 block。
  - 如果同一个 block 内的两个线程没有联系或者存在大量 `if/then/else`，就会出现很多线程分支，这样的情况尽量分配到不同的 block。
  - 一般来说，每个 block 用 128 到 256 个线程。
- 共享内存
  - 这里的共享内存指同一个 block 里的线程共享的部分内存，相比存取芯片外的全局内存，存取共享内存有低延迟、高带宽的特点。由于共享内存很小，因此程序设计时需要考虑哪些数据会被重复使用。
  - 把全局内存中的某些数据复制到共享内存中，可以提升性能。
  - 本质上说，共享内存是一个程序员可控的缓存。

目 录

示例：计算行和

- 代码说明
  - 新建一个文本文件 RowSums.cu，写入 C 代码。通常把 CUDA 源码的文件后缀设为 .cu，也可以任意修改，比如 .c。
  - #include <cuda.h> 会导入 CUDA 中的函数定义。
  - 参数 m 表示待计算行和的矩阵，rs 表示行和结果向量，n 表示矩阵行列数。
  - main 函数运行在 CPU 上，find1elt 函数运行在 GPU 上。
- 代码示例（I）

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

__global__ void find1elt(int *m, int *rs, int n) {
   // 该线程会处理第 rownum 行
   int rownum = blockIdx.x;
   int sum = 0;
   for (int k = 0; k < n; k++)
      sum += m[rownum*n+k];
   rs[rownum] = sum;
}
```

示例：计算行和

- 代码示例（II）

```
int main(int argc, char **argv) {
    int n = atoi(argv[1]);  // 矩阵的行和列数
    int *hm, // host 矩阵
        *dm, // device 矩阵
        *hrs, // host 行和
        *drs; // device 列和
    // 矩阵占用了多少 bytes
    int msize = n * n * sizeof(int);
    // 为 host 矩阵分配空间
    hm = (int *) malloc(msize);
    // 作为测试，矩阵用连续整数填充
    int t = 0,i,j;
    for (i = 0; i < n; i++) {
       for (j = 0; j < n; j++) {
          hm[i*n+j] = t++;
       }
    }
```

示例：计算行和

- 代码示例（III）

```
// 在 device 上为矩阵分配空间
cudaMalloc((void **)&dm,msize);
// 将矩阵从 host 复制到 device 上
cudaMemcpy(dm,hm,msize,cudaMemcpyHostToDevice);
// 在 host 和 device 上为行和向量分配空间
int rssize = n * sizeof(int);
hrs = (int *) malloc(rssize);
cudaMalloc((void **)&drs,rssize);
// 设置线程结构的参数
dim3 dimGrid(n,1);  // grid 里共有 n 个 blocks
dim3 dimBlock(1,1,1); // 每个 block对应一个线程
// 启动 kernel 函数
find1elt<<<dimGrid,dimBlock>>>(dm,drs,n);
// 等待 kernel 函数完成
cudaThreadSynchronize();
// 把行和向量从 device 复制到 host
cudaMemcpy(hrs,drs,rssize,cudaMemcpyDeviceToHost);
```

示例：计算行和

- 代码示例（IV）

```
    // 检查结果
    if (n < 10)
       for(int i=0; i<n; i++) printf("%d ",hrs[i]);
    // 清理释放
    free(hm);
    cudaFree(dm);
    free(hrs);
    cudaFree(drs);
}
```

- 运行代码
  - 可以使用 CUDA 工具箱里的 nvcc 来编译这些代码：

```
nvcc -o rowsums RowSums.cu
```

  - 会产生一个可执行文件 rowsums。
  - nvcc 只是底层编译器（比如 gcc）的一层封装，因此后者的所有编译选项都被保留了。

CUDA 的并行机制

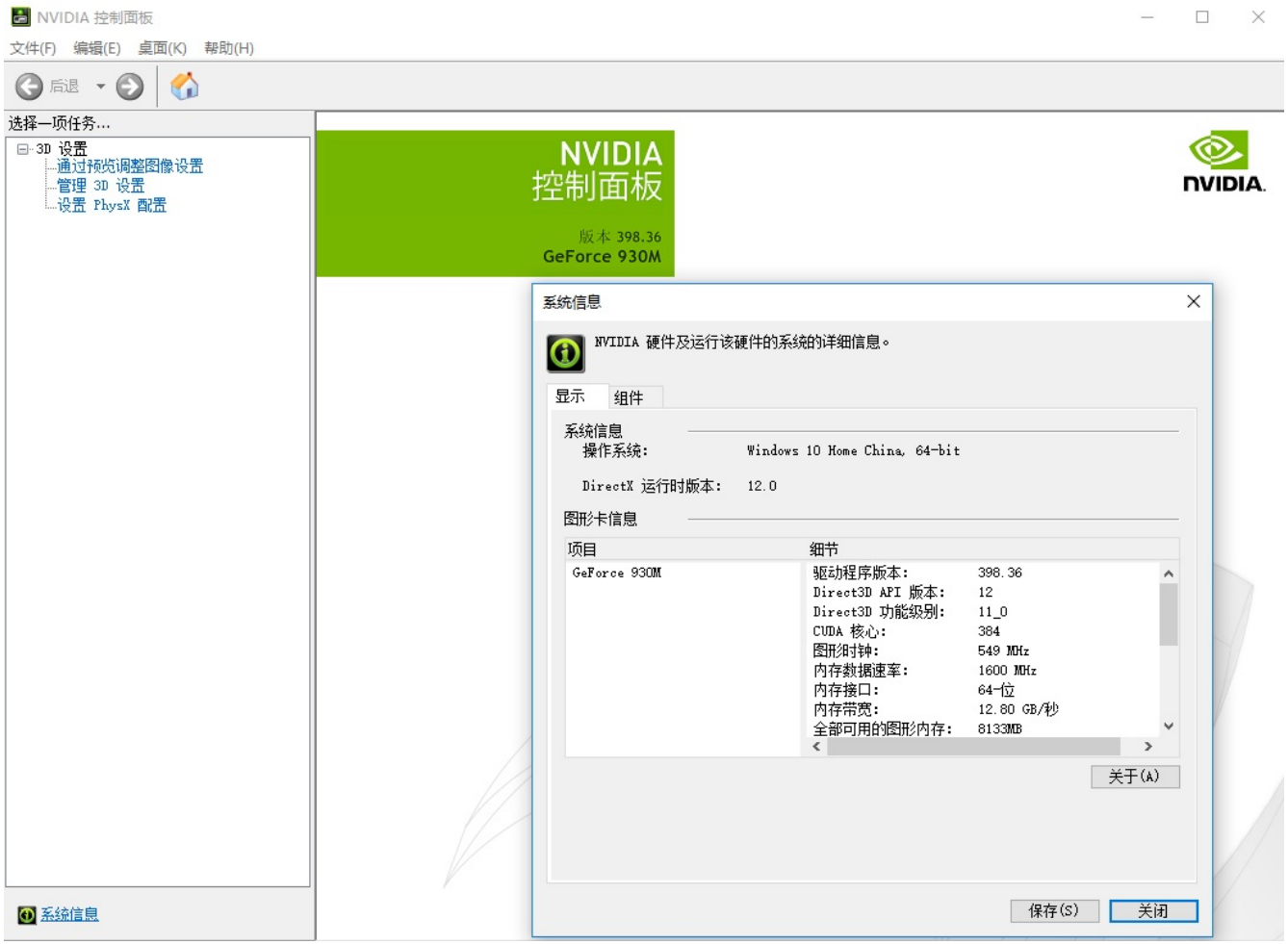
- 函数结构
  - `find1elt` 函数是 kernel 函数，运行在 GPU 上，由前缀 `__global__` 标记。
  - host 上通过调用 `cudaMalloc` 来在 device 上分配内存，之后通过 `cudaMemcpy` 在 host 和 device 之间相互传输数据，device 上的数据对于所有线程都是全局变量。
  - `find1elt<<<dimGrid,dimBlock>>>(dm,drs,n)` 用来启动 kernel 函数。
- 线程与分块
  - 启动 kernel 时需要告诉硬件 block 和线程的信息。
  - `dimGrid` 指定 grid 中有多少个 block（第一个参数），`dimBlock` 指定每个 block 中有多少个线程（第一个参数）。
  - `blockIdx` 表示 block 的 ID，`threadIdx` 表示线程 ID，每个线程内都可以获取到自己的值。`.x` 表示这两个数值的第一维度。在本例中每个 block 都只有一个线程，因此 `blockIdx.x` 就是线程 ID。

目 录

目 录

确认环境

- 确认是否具备 NVIDIAN 显卡
  - 桌面上右键，选择“NVIDIAN”控制面板。



确认计算能力

- 进官网查看显卡的计算能力
  - https://developer.nvidia.com/cuda-gpus

GeForce Desktop Products

GPU	Compute Capability
NVIDIA TITAN V	7.0
NVIDIA TITAN Xp	6.1
NVIDIA TITAN X	6.1
GeForce GTX 1080 Ti	6.1
GeForce GTX 1080	6.1
GeForce GTX 1070	6.1
GeForce GTX 1060	6.1
GeForce GTX 1050	6.1
GeForce GTX TITAN X	5.2
GeForce GTX TITAN Z	3.5
GeForce GTX TITAN Black	3.5
GeForce GTX TITAN	3.5
GeForce GTX 980 Ti	5.2

GeForce Notebook Products

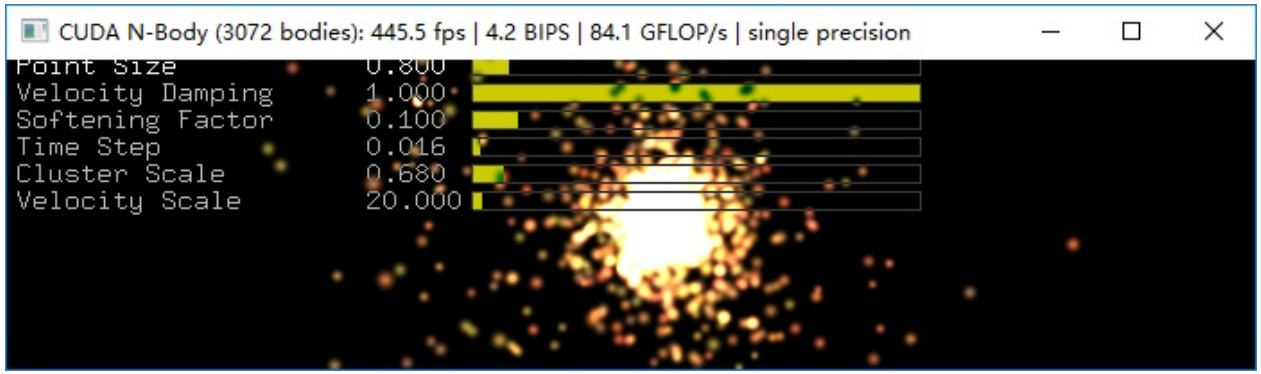
GPU	Compute Capability
GeForce GTX 1080	6.1
GeForce GTX 1070	6.1
GeForce GTX 1060	6.1
GeForce GTX 980	5.2
GeForce GTX 980M	5.2
GeForce GTX 970M	5.2
GeForce GTX 965M	5.2
GeForce GTX 960M	5.0
GeForce GTX 950M	5.0
GeForce 940M	5.0
GeForce 930M	5.0
GeForce 920M	3.5
GeForce 910M	5.2

安装开发环境

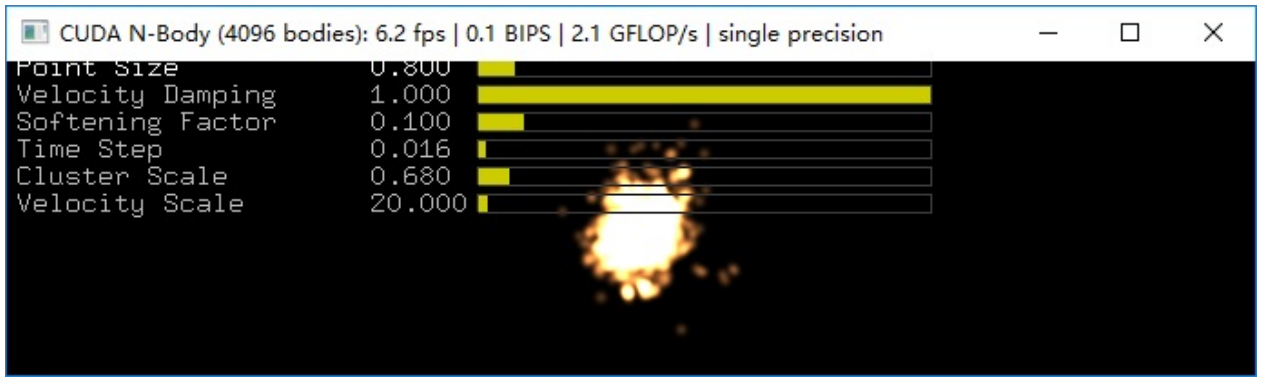
- 安装 CUDA
  - 到 NVIDIA CUDA 下载页面（https://developer.nvidia.com/cuda-downloads）选择 Windows 10 下 local 的安装文件。
  - 下载后得到一个 exe 文件，例如 cuda_8.0.61_win10.exe。
  - 双击默认安装，将会被安装到 C:\Program Files\NVIDIA GPU Computing Toolkit 文件夹。
  - 安装后将会自动更新环境变量。
- 安装 Visual Studio
  - 可以安装 Visual Studio 的社区版，比如 Visual Studio 2013，有些系统默认包含，也可以到微软官网下载。
  - 进入示例文件夹 C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0，可以通过双击后缀名为 .sln 的文件用 Visual Studio 打开示例中的工程。
  - 右击相应的例子，选择“build”，可以进行编译。注意设置 debug 或则 release 模式，如果是 release 模式，将会在 bin 目录下生成可执行的 exe 文件。

测试

- 比较 GPU 和 CPU 的计算能力
  - GFLOP/s 表示每秒 10 亿次浮点运算数。
- GPU 运行 nbody 示例
  - 在编译出来的 bin 文件夹中运行：nbody



- CPU 运行 nbody 示例
  - 在编译出来的 bin 文件夹中运行：nbody -cpu



目 录

Linux 简介

- 简介
  - Linux 是一种自由和开放源代码的类 UNIX 操作系统。只要遵循 GPL，任何个人和机构都可以自由地使用 Linux 的所有底层源代码，也可以自由地修改和再发布。
  - 严格来讲，术语 Linux 只表示操作系统内核本身，但通常采用 Linux 内核来表达该意思。Linux 则常用来指基于 Linux 内核的完整操作系统，包括 GUI 组件和许多其他实用工具。
- 发展历程
  - 1969 年，美国 AT&T 公司贝尔实验室完成 UNIX 操作系统，并于 1971 年首次发布。
  - 1983 年，RMS 创立 GNU 计划。该计划有一个目标，是为了发展一个完全自由的类 Unix 操作系统。
  - 1987 年，Andrew S. Tanenbaum 为教学而设计了 MINIX，这是一个轻量小型并采用微内核架构的类 Unix 操作系统。
  - 1991 年，Linus Torvalds 上大学时对 MINIX 只允许在教育上使用很不满，于是他便开始写他自己的操作系统，这就是后来的 Linux 内核。

Linux 版本

- Linux 发行版
  - 一个典型的 Linux 桌面发行版包括一个 Linux 内核，来自 GNU 的工具和库，和附加的软件、文档，还有一个窗口系统，窗口管理器，和一个桌面环境。
- 常见发行版
  - Ubuntu，一个非常流行的桌面发行版，由 Canonical 维护。
  - Red Hat Enterprise Linux，Fedora 的商业版，由 Red Hat 维护和提供技术支持。
  - Fedora，是 Red Hat 的社区版，会经常引入新特性进行测试。
  - CentOS，从 Red Hat 发展而来的发行版，由志愿者维护，旨在提供开源的，并与 Red Hat 100% 兼容的系统。
  - Debian，一个强烈信奉自由软件，并由志愿者维护的系统。
  - Slackware，最早的发行版之一，1993 年创建，由 Patrick J. Volkerding 维护。
  - openSUSE，最初由 Slackware 分离出来，现在由 Novell 维护。

显卡环境

- 确认显卡
  - 在命令行执行以下命令可以弹出显卡设置：

```
nvidia-settings
```

  - 通常默认的显卡是 Intel 集成显卡，这样会比较省电。如果这个设置窗口中显示当前使用着 Intel 集成显卡，需要手动修改成 NVIDIA 的显卡，然后重启。
  - 查看测试环境中显卡型号，例如 GeForce 830M，具有 256 个核和 2G 内存。
- 更新驱动程序
  - 如果显卡驱动过旧的话，将不能正常安装 CUDA 7.5。可以到软件更新中心去将 NVIDIA 的驱动更新到最新（352.63 之后）
  - 执行如下命令可以看到更详细的显卡信息：

```
nvidia-smi
```

安装 CUDA

- 下载和安装
  - 到 https://developer.nvidia.com/cuda-downloads 下载页面选择 Ubuntu 14.04 下 deb 的安装文件，下载到某个文件夹，如 ~/Downloads，然后进入该文件夹运行以下命令：

```
cd ~/Downloads

sudo dpkg -i cuda-repo-ubuntu1404-7-5_amd64.deb
sudo apt-get update
sudo apt-get install cuda
```

- 设置环境变量
  - 用 vi 打开 /etc/profile 文件，并添加以下内容：

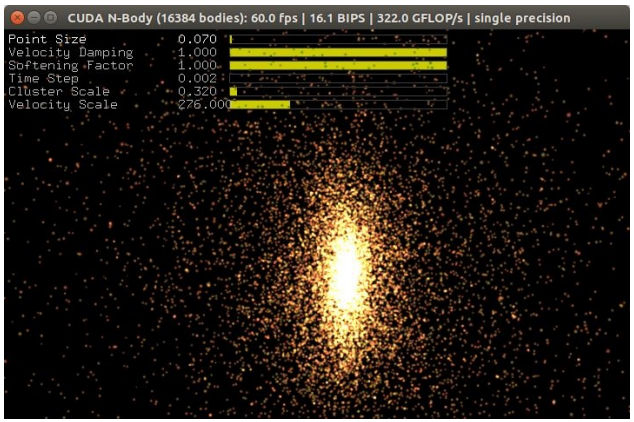
```
CUDA_HOME=/usr/local/cuda-7.5
export CUDA_HOME
PATH=$PATH:$CUDA_HOME/bin
export PATH
```

测试安装

- 测试
  - 进入到某个工作文件夹，比如 ~/tmp，运行如下命令：

```
cd ~/tmp
cuda-install-samples-7.5.sh .
cd NVIDIA_CUDA-Samples_7.5/5_Simulations/nbody
make
./nbody
```

  - 将会在当前目录下新建一个名为 NVIDIA_CUDA-Samples_7.5 的文件夹，成功运行后将会弹出一个现实显存计算的框。无误则说明安装成功



目 录

目 录

R 与 GPU

- R 中 GPU 运算的工具
  - gputools，最常用的 R 包，包含了一些基础的矩阵运算函数和常用分析函数。
  - gpuR，可以在 Windows 下使用，该包用 OpenCL 的接口调用 GPU 进行计算，理论上不需要 CUDA，但是 CUDA 也提供了 OpenCL 的库，并且操作起来非常简便。
  - 也可以直接从 R 中调用 CUDA。因为 CUDA 基于 C，用调用 C 接口的方式就可以了。
- Linux 中设置环境变量
  - 用 vi 打开 /etc/profile 文件，并添加以下内容：

```
R_HOME := /home/jian/R/R-3.3.4/lib/R
R_INC := $R_HOME/include
R_LIB := $R_HOME/lib
```

gpuR 示例

- 矩阵乘法的例子

```
library(gpuR)
gpu.matmult <- function(n) {
      gpuA <- gpuMatrix(runif(n*n), nrow=n, ncol=n)
      gpuB <- gpuMatrix(runif(n*n), nrow=n, ncol=n)
        A <- as.matrix(gpuA)
        B <- as.matrix(gpuB)
    tic <- Sys.time()
    C <- A %*% B
    toc <- Sys.time()
    comp.time <- toc - tic
    cat("CPU: ", comp.time, "\n")
    tic <- Sys.time()
    C <- gpuA %*% gpuA
    toc <- Sys.time()
    comp.time <- toc - tic
    cat("GPU: ", comp.time, "\n")
}
```

gpuR 函数简介

- 设备管理
  - detectGPUs/detectCPUs，查看 GPU 和 CPU 的个数。
  - gpuInfo/cpuInfo，查看 GPU 和 CPU 的详细信息。
  - currentDevice，查看当前设备。
  - platformInfo，查看平台信息。
- 数据操作
  - gpuMatrix，创建一个基于 GPU 的矩阵，返回一个 S4 的 fgpuMatrix 对象，指向一个地址，但是可以使用 [] 对矩阵取值。注意需要传入浮点型数值，不能是双精度的。
  - gpuVector，创建 GPU 向量对象 igpuVector。
  - as.gpuMatrix/as.gpuVector，将普通矩阵和向量转化成基于 GPU 的对象。
  - t，矩阵转置。
  - deepcopy，复制矩阵对象。
  - Compare，比较两个向量。

gpuR 函数简介

- 数据计算
  - nrow/ncol，获取矩阵的行数和列数。
  - pmax/pmin，最大值和最小值。
  - rowMeans/rowSums，按行求和或者求平均。
  - distance，计算距离矩阵。
  - diag，求矩阵的对角线。
- 矩阵运算
  - %*%，矩阵乘法。
  - %o%，矩阵外积。
  - solve，求逆。
  - svd，奇异值分解。
  - qr，QR 分解
  - eigen，计算特征值和特征向量。
  - crossprod内积。
  - det，计算行列式。

目 录

Thrust 简介

- 直接基于 CUDA 编程
  - CUDA 本质上是 C++ 框架，直接使用比较麻烦，也很难达到真正的高效。
  - 如果有可能，可以使用 GPU 代码库，Thrust 就是一个很常用的库。
- Thrust
  - Thrust 由 NVIDIA 开发，由一系列类似 C++ 标准库（STL）的 C++ 模板构成。
  - 使用模板方法的一大优势在于不需要使用特别的编译器，Thrust 仅仅是由 `#include` 引入的一系列文件而已。
  - 某些意义上说，Thrust 可以认为是用于 GPU 开发的高级代码，以避免 GPU 开发中的种种细节。
  - Thrust 可以实现异构计算，能够产生运行在不同平台上的不同版本的代码。编译 Thrust 代码时，我们可以选择不同后台，包括 GPU 或者多核后台。如果是多核后台，用户可以选择在多核机器上使用 OpenMP 或者 TBB。

示例：计算分位数

- 代码说明
  - 从一个数组中每 k 个元素取一个元素，从而找到数据的 $i \cdot k/n$ 分位数，其中 $i = 1, 2, \cdots$。
  - 函数 ismultk 用来判断是否为 k 的倍数。通过 copy_if 将该函数作用到数组中的所有值 i 上。
- 代码示例（I）

```
#include <stdio.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <thrust/sequence.h>
#include <thrust/copy.h>

struct ismultk {
    const int increm;  // 通过调用得到的 k
    ismultk(int _increm): increm(_increm) {}
    __device__ bool operator()(const int i)
    {  return i != 0 && (i % increm) == 0;
    }
};
```

示例：计算分位数

- 代码示例（II）

```
int main(int argc, char **argv)
{
    int x[15] =
       6,12,5,13,3,5,4,5,8,88,1,11,9,22,168;
    int n=15;

    // 在设备上生成矩阵
    // 初始化 x[0], x[1], ..., x[n-1]
    thrust::device_vector<int> dx(x,x+n);

    // dx 就地排序
    thrust::sort(dx.begin(),dx.end());

    // 生成一个长度为 n 的向量 seq
    thrust::device_vector<int> seq(n);

    // 用 0,1,2,...n-1 填充 seq
    thrust::sequence(seq.begin(),seq.end(),0);
```

示例：计算分位数

- 代码示例（III）

```
    // 为存储分位数分配空间
    thrust::device_vector<int> out(n);

    // 从命令行读入 k
    int incr = atoi(argv[1]);

    // 对 seq 中的每个元素调用 ismultk()
    // 如果结果为真，将 dx[i] 放入 out
    // 最终返回指向输入数组结尾的指针
    thrust::device_vector<int>::iterator newend =
     thrust::copy_if(dx.begin(),dx.end(),seq.begin(),
          out.begin(),ismultk(incr));

    // 打印结果
    thrust::copy(out.begin(), newend,
       std::ostream_iterator<int>(std::cout, " "));
    std::cout << "\n";
}
```

Rth 简介

- Rth
  - 主页：https://github.com/Rth-org/Rth。Norman 基于 Thrust 封装的 R 包，可以利用 GPU 实现一些常见的算法。
  - 在 CUDA 环境下，完全傻瓜化操作。
- 代码示例

```
library(Rth)
n <- 200000000
tmp <- matrix(runif(2*n), ncol = 2)
x <- tmp[, 1]
y <- x + tmp[, 2]

system.time(c1 <- cor.test(x, y))

##    user  system elapsed
##    8.50    8.48   30.31

system.time(c2 <- rthpearson(x, y, 2))

##    user  system elapsed
##    2.47    2.53    4.53
```

Thank you!

李舰 Email: jian.li@188.com