

CUDA 编程学习



分类: [学习笔记](#) 发布时间 2023.01.31 阅读数 170 评论数 0

0 简单的CUDA简介

1.简单教程

CUDA C++只是使用CUDA创建大规模并行应用程序的方法之一。它允许您使用功能强大的C++编程语言来开发由GPU上运行的数千个并行线程加速的高性能算法。许多开发人员以这种方式加速了他们的计算和带宽需求的应用程序，包括支持正在进行的人工智能革命(称为深度学习)的库和框架。

1.1.案例

C++程序开始, 该程序添加两个数组的元素, 每个数组包含一百万个元素

```
1  #include <iostream>
2  #include <math.h>
3
4  // function to add the elements of two arrays
5  void add(int n, float *x, float *y)
6  {
7      for (int i = 0; i < n; i++)
8          y[i] = x[i] + y[i];
9  }
10
11 int main(void)
12 {
13     int N = 1<<20; // 1M elements
14
15     float *x = new float[N];
16     float *y = new float[N];
17
18     // initialize x and y arrays on the host
19     for (int i = 0; i < N; i++) {
20         x[i] = 1.0f;
21         y[i] = 2.0f;
22     }
23
24     // Run kernel on 1M elements on the CPU
25     add(N, x, y);
26
27     // Check for errors (all values should be 3.0f)
28     float maxError = 0.0f;
29     for (int i = 0; i < N; i++)
30         maxError = fmax(maxError, fabs(y[i]-3.0f));
31     std::cout << "Max error: " << maxError << std::endl;
32
33     // Free memory
34     delete [] x;
35     delete [] y;
36
37     return 0;
38 }
39
```

1.2 GPU 实现

第一步 cuda内核函数

首先, 我只需将我们的add函数转换为GPU可以运行的函数, 称为CUDA中的内核。要做到这一点, 我所要做的就是函数中添加说明符__global__, 它告诉CUDA C++编译器这是一个在GPU上运行并可以从CPU代码调用的函数。

```
1  // CUDA Kernel function to add the elements of two arrays on the GPU
2  __global__
3  void add(int n, float *x, float *y)
4  {
5      for (int i = 0; i < n; i++)
6          y[i] = x[i] + y[i];
7  }
```

这儿 global 函数称为内核, 在GPU上运行的代码通常称为设备代码, 而在CPU上运行的代码则是主机代码。

第二步 cuda内存分配

要在GPU上进行计算, 我需要分配GPU可访问的内存。CUDA中的统一内存通过提供系统中所有GPU和CPU可访问的单个内存空间, 使这一过程变得简单。要在统一内存中分配数据, 请调用cudaMallocManaged(), 它返回一个可以从主机(CPU)代码或设备(GPU)代码访问的指针。要释放数据, 只需将指针传递给cudaFree()即可。

我只需要通过调用cudaMallocManaged()替换上面代码中对new的调用, 并用对cudaFree的调用替换对delete []的调用。

```
1  // Allocate Unified Memory -- accessible from CPU or GPU
2  float *x, *y;
3  cudaMallocManaged(&x, N*sizeof(float));
4  cudaMallocManaged(&y, N*sizeof(float));
5
6  ...
7
8  // Free memory
9  cudaFree(x);
10 cudaFree(y);
```

第三步 启动内核

最后, 我需要启动add()内核, 它在GPU上调用它。使用三角括号语法<<<>>>指定CUDA内核启动。我只需将它添加到参数列表之前添加的调用中。

```
1  add<<<1, 1>>>(N, x, y);
```

整体代码演示

还有一件事:我需耍CPU等到内核完成才能访问结果(因为CUDA内核启动不会阻塞调用CPU线程)。为此,我只需在对CPU进行最终错误检查之前调用cudaDeviceSynchronize()。

这是完整的代码

```
1 | #include <iostream>
2 | #include <math.h>
3 | // Kernel function to add the elements of two arrays
4 | __global__
5 | void add(int n, float *x, float *y)
6 | {
7 |     for (int i = 0; i < n; i++)
8 |         y[i] = x[i] + y[i];
9 | }
10 |
11 | int main(void)
12 | {
13 |     int N = 1<<20;
14 |     float *x, *y;
15 |
16 |     // Allocate Unified Memory - accessible from CPU or GPU
17 |     cudaMallocManaged(&x, N*sizeof(float));
18 |     cudaMallocManaged(&y, N*sizeof(float));
19 |
20 |     // initialize x and y arrays on the host
21 |     for (int i = 0; i < N; i++) {
22 |         x[i] = 1.0f;
23 |         y[i] = 2.0f;
24 |     }
25 |
26 |     // Run kernel on 1M elements on the GPU
27 |     add<<<1, 1>>>(N, x, y);
28 |
29 |     // Wait for GPU to finish before accessing on host
30 |     cudaDeviceSynchronize();
31 |
32 |     // Check for errors (all values should be 3.0f)
33 |     float maxError = 0.0f;
34 |     for (int i = 0; i < N; i++)
35 |         maxError = fmax(maxError, fabs(y[i]-3.0f));
36 |     std::cout << "Max error: " << maxError << std::endl;
37 |
38 |     // Free memory
39 |     cudaFree(x);
40 |     cudaFree(y);
41 |
42 |     return 0;
43 | }
```

CUDA文件的文件扩展名为.cu。因此,将此代码保存在名为add.cu的文件中,并使用CUDA C ++编译器nvcc进行编译。

```
1 | nvcc add.cu -o add_cuda
2 | ./add_cuda
```

这只是第一步,因为编写时,这个内核只对单个线程是正确的,因为运行它的每个线程都将在整个数组上执行添加。此外,存在竞争条件,因为多个并行线程将读取和写入相同的位置。

耗时计算

我认为找出内核运行多长时间的最简单方法是使用nvprof运行它,这是CUDA Toolkit附带的命令行GPU分析器。只需在命令行上键入

```
1 | $ nvprof ./add_cuda
2 | ==3355== NVPROF is profiling process 3355, command: ./add_cuda
3 | Max error: 0
4 | ==3355== Profiling application: ./add_cuda
5 | ==3355== Profiling result:
6 | Time(%)    Time    Calls    Avg      Min      Max   Name
7 | 100.00%   463.25ms      1  463.25ms  463.25ms  463.25ms  add(int, float*,
8 | float*)
9 | ...
```

以上是nvprof的截断输出,显示了一个要添加的调用。NVIDIA Tesla K80加速器需要大约半秒钟,而在我3年前的Macbook ok Pro中,NVIDIA GeForce GT 740M大约需要半秒钟。

设置线程

现在您已经运行了一个内核,其中一个线程可以进行一些计算,那么如何使它并行?关键在于CUDA的<<<1,1>>>语法。这称为执行配置,它告诉CUDA运行时有多少并行线程用于GPU上的启动。这里有两个参数,但让我们从改变第二个参数开始:线程块中的线程数。CUDA GPU使用大小为32的线程块运行内核,因此256个线程是合理的大小。

```
1 | add<<<1, 256>>>(N, x, y);
```

如果我仅使用此更改运行代码,它将为每个线程执行一次计算,而不是跨并行线程传播计算。要正确地做,我需要修改内核。CUDA C ++提供的关键字让内核获得正在运行的线程的索引。具体来说,threadIdx.x包含其块中当前线程的索引,blockDim.x包含块中的线程数。我只是修改循环以使用并行线程跨越数组。

```
1 | __global__
2 | void add(int n, float *x, float *y)
3 | {
4 |     int index = threadIdx.x;
5 |     int stride = blockDim.x;
6 |     for (int i = index; i < n; i += stride)
7 |         y[i] = x[i] + y[i];
8 | }
```

- 1.添加功能没有那么大改变。实际上,将index设置为0并且stride为1使其在语义上与第一个版本相同。
- 2.将文件另存为add_block.cu并再次在nvprof中编译并运行它。对于帖子的剩余部分,我将只显示输出中的相关行。

```
1 | Time(%)    Time    Calls    Avg      Min      Max   Name
2 | 100.00%   2.7107ms      1  2.7107ms  2.7107ms  2.7107ms  add(int, float*,
3 | float*)
```

这是一个很大的加速(463ms到2.7ms),但是从我从1个线程到256个线程就不足为奇了。K80比我的Macbook Pro G PU(3.2ms)更快。让我们继续获得更多的表现。

多余一个线程块

CUDA GPU有许多并行处理器, 分为流式多处理器或SM。每个SM可以运行多个并发线程块。

到目前为止, 您可能已经猜到执行配置的第一个参数指定了线程块的数量。并行线程块共同构成了所谓的网格。由于我要处理N个元素, 每个块有256个线程, 所以我只需要计算得到至少N个线程的块数。我只是将N除以块大小(如果N不是blockSize的倍数, 则要小心舍入)。

```
1 |     int blockSize = 256;
2 |     int numBlocks = (N + blockSize - 1) / blockSize;
3 |     add<<<numBlocks, blockSize>>>(N, x, y);
```

我还需要更新内核代码以考虑整个线程块网格。CUDA提供gridDim.x, 它包含网格中的块数, blockIdx.x包含网格中当前线程块的索引。图1说明了使用blockDim.x, gridDim.x和threadIdx.x在CUDA中索引到一个数组(一维)的方法。我们的想法是每个线程通过计算其块开头的偏移量(块索引乘以块大小:blockIdx.x * blockDim.x)并在块(threadIdx.x)中添加线程索引来获取其索引。代码blockIdx.x * blockDim.x + threadIdx.x是惯用的CUDA。

```
1 |     __global__
2 |     void add(int n, float *x, float *y)
3 |     {
4 |         int index = blockIdx.x * blockDim.x + threadIdx.x;
5 |         int stride = blockDim.x * gridDim.x;
6 |         for (int i = index; i < n; i += stride)
7 |             y[i] = x[i] + y[i];
8 |     }
```

更新的内核还设置了网格中线程总数(blockDim.x * gridDim.x)的步幅。CUDA内核中的这种类型的循环通常称为网格跨步循环。

- 将文件另存为add_grid.cu并再次在nvprof中编译并运行它。

```
1 |      Time(%)      Time      Calls      Avg      Min      Max  Name
2 | 100.00%   94.015us         1   94.015us   94.015us   94.015us  add(int, float*,
   float*)
```

2. 如何在CUDA C / C ++中实现性能指标

在本系列的第一篇文章中, 我们通过检查SAXPY的CUDA C / C ++实现来查看CUDA C / C ++的基本元素。在第二篇文章中, 我们将讨论如何分析这个和其他CUDA C / C ++代码的性能。我们将在未来的帖子中依赖这些性能测量技术, 其中性能优化将变得越来越重要。

CUDA性能测量通常从主机代码完成, 可以使用CPU计时器或CUDA特定的计时器来实现。在我们进入这些性能测量技术之前, 我们需要讨论如何在主机和设备之间同步执行。

2.1 主机 - 设备同步

让我们看一下上一篇文章中SAXPY主机代码的数据传输和内核启动:

```
1 |     cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
2 |     cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
3 |
4 |     saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
5 |
6 |     cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

使用cudaMemcpy()在主机和设备之间传输数据是同步(或阻塞)传输。在完成所有先前发出的CUDA调用之前, 同步数据传输不会开始, 并且在同步传输完成之前, 后续CUDA调用无法开始。因此, 在第二行上My到d_y的传输完成之前, 第三行上的saxpy内核启动不会发出。另一方面, 内核启动是异步的。在第三行启动内核后, 控制会立即返回到CPU, 而不是等待内核完成。虽然这似乎为最后一行中的设备到主机数据传输设置了竞争条件, 但数据传输的阻塞特性可确保内核在传输开始之前完成。

2.2 使用CPU计时器执行定时内核执行

现在让我们来看看如何使用CPU计时器对内核执行进行计时。

```
1 |     cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
2 |     cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
3 |
4 |     t1 = myCPUTimer();
5 |     saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
6 |     cudaDeviceSynchronize();
7 |     t2 = myCPUTimer();
8 |
9 |     cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

注意: 除了对通用主机时间戳函数myCPUTimer()的两次调用之外, 我们还使用显式同步屏障cudaDeviceSynchronize()来阻止CPU执行, 直到设备上所有先前发出的命令都完成为止。如果没有此障碍, 此代码将测量内核启动时间而不是内核执行时间。

2.3 使用CUDA事件的时间计时

使用主机设备同步点(例如cudaDeviceSynchronize())的一个问题是它们阻止了GPU管道。因此, CUDA通过CUDA事件API为CPU计时器提供了相对轻量级的替代方案。CUDA事件API包括调用以创建和销毁事件, 记录事件以及计算两个记录事件之间经过的时间(以毫秒为单位)。

CUDA事件利用了CUDA流的概念。CUDA流只是在设备上按顺序执行的一系列操作。不同流中的操作可以交错并且在某些情况下可以重叠, 可以用于隐藏主机和设备之间的数据传输的属性(稍后我们将详细讨论)。到目前为止, GPU上的所有操作都发生在默认流或流0(也称为“空流”)中。在下面的列表中, 我们将CUDA事件应用于我们的SAXPY代码。

```
1 |     cudaEvent_t start, stop;
2 |     cudaEventCreate(&start);
3 |     cudaEventCreate(&stop);
4 |
5 |     cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
6 |     cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
7 |
8 |     cudaEventRecord(start);
9 |     saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
10 |    cudaEventRecord(stop);
11 |
12 |    cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
13 |
14 |    cudaEventSynchronize(stop);
15 |    float milliseconds = 0;
16 |    cudaEventElapsedTime(&milliseconds, start, stop);
```

CUDA事件的类型为cudaEvent_t, 使用cudaEventCreate()和cudaEventDestroy()创建和销毁。在上面的代码中, cudaEventRecord()将start和stop事件放入默认流, 即流0, 设备将在事件到达流中的事件时记录事件的时间戳。函数cudaEv

entSynchronize()阻止CPU执行，直到记录指定的事件。cudaEventElapsedTime()函数在第一个参数中返回记录start和stop之间经过的毫秒数。该值的分辨率约为半微秒。

2.4 内存带宽

现在我们有精确计时内核执行的方法，我们将使用它来计算带宽。在评估带宽效率时，我们同时使用理论峰值带宽和观察到的或有效的内存带宽。

2.4.1 理论带宽

可以使用产品文献中提供的硬件规格计算理论带宽。例如，NVIDIA Tesla M2050 GPU使用DDR(双倍数据速率)RAM，内存时钟速率为1,546 MHz，内存接口为384位。使用这些数据项，NVIDIA Tesla M2050的峰值理论内存带宽为148 GB /秒，如下所示。

$$BW_{Theoretical} = 1546 * 106 * (384/8) * 2 / 109 = 148 \text{ GB/s}$$

在此计算中，我们将内存时钟速率转换为Hz，将其乘以接口宽度(除以8，将位转换为字节)，并由于双倍数据速率乘以2。最后，我们除以109将结果转换为GB / s。

2.4.2 有效带宽

我们通过计时特定的程序活动以及了解程序如何访问数据来计算有效带宽。我们使用以下等式。

$$BWE_{Effective} = (RB + WB) / (t * 109)$$

这里，BWEffective是以GB / s为单位的有效带宽，RB是每个内核读取的字节数，WB是每个内核写入的字节数，t是以秒为单位的经过时间。我们可以修改SAXPY示例来计算有效带宽。完整的代码如下。

```
1      #include
2
3      __global__
4      void saxpy(int n, float a, float *x, float *y)
5      {
6          int i = blockIdx.x*blockDim.x + threadIdx.x;
7          if (i < n) y[i] = a*x[i] + y[i];
8      }
9
10     int main(void)
11     {
12         int N = 20 * (1 << 20);
13         float *x, *y, *d_x, *d_y;
14         x = (float*)malloc(N*sizeof(float));
15         y = (float*)malloc(N*sizeof(float));
16
17         cudaMalloc(&d_x, N*sizeof(float));
18         cudaMalloc(&d_y, N*sizeof(float));
19
20         for (int i = 0; i < N; i++) {
21             x[i] = 1.0f;
22             y[i] = 2.0f;
23         }
24
25         cudaEvent_t start, stop;
26         cudaEventCreate(&start);
27         cudaEventCreate(&stop);
28
29         cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
30         cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
31
32         cudaEventRecord(start);
33
34         // Perform SAXPY on 1M elements
35         saxpy<<<(N+511)/512, 512>>>(N, 2.0f, d_x, d_y);
36
37         cudaEventRecord(stop);
38
39         cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
40
41         cudaEventSynchronize(stop);
42         float milliseconds = 0;
43         cudaEventElapsedTime(&milliseconds, start, stop);
44
45         float maxError = 0.0f;
46         for (int i = 0; i < N; i++) {
47             maxError = max(maxError, abs(y[i]-4.0f));
48         }
49
50         printf("Max error: %fn", maxError);
51         printf("Effective Bandwidth (GB/s): %fn", N*4*3/milliseconds/1e6);
52     }
```

在带宽计算中，N * 4是每个阵列读或写传输的字节数，因子3表示x的读数和y的读写。经过的时间以变化的毫秒数存储，以使单位清晰。请注意，除了添加带宽计算所需的功能外，我们还更改了数组大小和线程块大小。

2.5 测量计算吞吐量

我们刚刚演示了如何测量带宽，这是衡量数据吞吐量的指标。另一个对性能非常重要的指标是计算吞吐量。计算吞吐量的常用度量是GFLOP / s，它代表“每秒Giga-FLoating-point OPerations”，其中Giga是109的前缀。对于我们的SAXPY计算，测量有效吞吐量很简单：每个SAXPY元素都相乘-add操作，通常测量为两个FLOP，所以我们有

$$GFLOP/s \text{ Effective} = 2N / (t * 109)$$

N是SAXPY操作中的元素数，t是以秒为单位的经过时间。与理论峰值带宽一样，理论峰值GFLOP / s可以从产品文献中收集(但计算它可能有点棘手，因为它非常依赖于架构)。例如，Tesla M2050 GPU的理论峰值单精度浮点吞吐量为1030 GFLOP / s，理论峰值双精度吞吐量为515 GFLOP / s。

SAXPY读取每个元素计算的12个字节，但只执行一个乘法 - 加法指令(2个FLOP)，因此很明显它将受带宽限制，因此在这种情况下(实际上在很多情况下)，带宽是最多的衡量和优化的重要指标。在更复杂的计算中，在FLOP级别上测量性能可能非常困难。因此，使用分析工具来了解计算吞吐量是否是瓶颈更为常见。应用程序通常提供特定于问题(而不是特定于体系结构)的吞吐量度量标准，因此对用户更有用。例如，天文n体问题的“每秒十亿次相互作用”，或分子动态模拟的“每秒纳秒”。

3.如何在CUDA C / C ++中优化数据传输

在本CUDA C & C ++系列的前三篇文章中，我们为该系列的主要内容奠定了基础：如何优化CUDA C / C ++代码。在本文和后面的文章中，我们将开始讨论代码优化，以及如何在主机和设备之间有效地传输数据。设备内存和GPU之间的峰值带宽要高得多(例如，NVIDIA Tesla C2050上为144 GB / s)，而不是主机内存和设备内存之间的峰值带宽(PCIe x16 Gen2上为8 GB / s)。这种差异意味着您在主机和GPU设备之间实现数据传输可能会影响您的整体应用程序性能。让我们从主机设备数据传输的一些通用指南开始。

尽可能减少主机和设备之间传输的数据量，即使这意味着在GPU上运行内核，与在主机CPU上运行它们相比，加速很少或没有加速。

当使用页锁定(或“固定”)存储器时,主机和设备之间可以有更高的带宽。

将许多小型传输批处理为一个较大的传输会更好,因为它消除了大部分的每次传输开销。

主机和设备之间的数据传输有时可能与内核执行和其他数据传输重叠。

我们在本文中调查了上面的前三条指南,并将下一篇文章专门用于重叠数据传输。首先,我想谈谈如何在不修改源代码的情况下测量数据传输所花费的时间。

3.1 使用nvprof测量数据传输时间

为了测量每次数据传输所花费的时间,我们可以在每次传输之前和之后记录一个CUDA事件,并使用cudaEventElapsedTime(),正如我们在上一篇文章中所描述的那样。但是,我们可以通过使用nvprof(CUDA工具包中包含的命令行CUDA分析器)(从CUDA 5开始)来获取已经过去的传输时间,而无需使用CUDA事件检测源代码。让我们尝试使用以下代码示例,您可以在Github存储库中找到此帖子。

NVIDIA CUDA Runtime API
https://docs.nvidia.com/cuda/cuda-runtime-api/index.html#group__CUDART__MEMORY_1g48efa06b81cc031b2aa6fdc2e9930741

1. CUDA 综述

CUDA®是NVIDIA发明的并行计算平台和编程模型。通过利用图形处理单元(GPU)的强大功能,它可以显着提高计算性能。

为标准编程语言(如C)提供一小组扩展,以实现并行算法的直接实现。使用CUDA C / C ++,程序员可以专注于算法并行化的任务,而不是花时间在他们的实现上。

支持异构计算,其中应用程序同时使用CPU和GPU。应用程序的串行部分在CPU上运行,并行部分卸载到GPU。因此,CUDA可以逐步应用于现有应用程序。CPU和GPU被视为具有自己的存储空间独立设备。此配置还允许在CPU和GPU上同时进行计算,而不会争用内存资源。

支持CUDA的GPU具有数百个核心,可以共同运行数千个计算线程。这些内核具有共享资源,包括寄存器文件和共享内存。片上共享存储器允许在这些内核上运行的并行任务共享数据,而无需通过系统内存总线发送数据。

CUDA(Compute Unified Device Architecture)的中文全称为计算统一设备架构。CUDA编程真的是入门容易精通难,具有计算机体系结构和C语言编程知识储备的同学上手CUDA编程应该难度不会很大。本文将通过以下五个方面帮助大家比较全面地了解CUDA编程最重要的知识点,做到快速入门:

GPU 架构特点
CUDA 线程模型
CUDA 内存模型
CUDA 编程模型
CUDA 应用小例子

1.1 GPU 架构特点

串行并行。高性能计算的关键利用多核处理器进行并行计算。

一个程序可不可以进行并行计算,关键就在于我们要分析出该程序可以拆分出哪几个执行模块,这些执行模块哪些是独立的,哪些又是强依赖强耦合的,独立的模块我们可以试着设计并行计算,充分利用多核处理器的优势进一步加速我们的计算任务,强耦合模块我们就使用串行编程,利用串行+并行的编程思路完成一次高性能计算。

首先CPU是专为顺序串行处理而优化的几个核心组成。而GPU则由数以千计的更小、更高效的核心组成,这些核心专门为同时处理多任务而设计,可高效地处理并行任务。也就是,CPU虽然每个核心自身能力极强,处理任务上非常强悍,无奈他核心少,在并行计算上表现不佳;反观GPU,虽然他的每个核心的计算能力不算强,但他胜在核心非常多,可以同时处理多个计算任务,在并行计算的支持上做得很好。

GPU和CPU的不同硬件特点决定了他们应用场景,CPU是计算机的运算和控制的核心,GPU主要用作图形图像处理。图像在计算机呈现的形式就是矩阵,我们对图像的处理其实就是操作各种矩阵进行计算,而很多矩阵的运算其实可以做并行化,这使得图像处理可以做得很快,因此GPU在图形图像领域也有了大展拳脚的机会。下图表示的就是一个多GPU计算机硬件系统,可以看出,一个GPU内存就有很多个SP和各类内存,这些硬件都是GPU进行高效并行计算的基础。

。现在再从数据处理的角度来对比CPU和GPU的特点。CPU需要很强的通用性来处理各种不同的数据类型,比如整型、浮点数等,同时它又必须擅长处理逻辑判断所导致的大量分支跳转和中断处理,所以CPU其实就是一个能力很强的伙计,他能办很多事处理得妥妥当当,当然啦我们需要给他很多资源供他使用(各种硬件),这也导致了CPU不可能有太多核心(核心总数不超过16)。而GPU面对的则是类型高度统一的、相互无依赖的大规模数据和不需要被打断的纯净的计算环境,GPU有非常多核心(费米架构就有512核),虽然其核心的能力远没有CPU的核心强,但是胜在多,在处理简单计算任务时呈现出“人多力量大”的优势,这就是并行计算的魅力。

整理一下两者特点就是:

- CPU:擅长流程控制和逻辑处理,不规则数据结构,不可预测存储结构,单线程程序,分支密集型算法
- GPU:擅长数据并行计算,规则数据结构,可预测存储模式

1.2 CUDA 的线程模型

CUDA的线程模型从小往大来总结就是:

1. Thread:线程,并行的基本单位
2. Thread Block:线程块,互相合作的线程组,线程块有如下几个特点:
 - 允许彼此同步
 - 可以通过共享内存快速交换数据
 - 以1维、2维或3维组织
3. Grid:一组线程块
 - 以1维、2维组织
 - 共享全局内存
 - Kernel:在GPU上执行的核心程序,这个kernel函数是运行在某个Grid上的。

One kernel <=> One Grid
每一个block和每个thread都有自己的ID,我们通过相应的索引找到相应的线程和线程块。

threadIdx, blockIdx
Block ID: 1D or 2D
Thread ID: 1D, 2D or 3D

理解kernel,必须要对kernel的线程层次结构有一个清晰的认识。

首先GPU上很多并行化的轻量级线程,kernel在device上执行时实际上是启动很多线程,一个kernel所启动的所有线程称为一个网格(grid),同一个网格上的线程共享相同的全局内存空间,grid是线程结构的第一层次,而网格又可以分为很多线程块(block),一个线程块里面包含很多线程,这是第二个层次。

kernel调用时也必须通过执行配置<<<grid,block>>>来指定kernel所使用的网格维度和线程块维度。CUDA的这种<<<grid,block>>>其实就是一个多级索引的方法,第一级索引是(grid.xldx,grid.yldy),然后我们启动二级索引(block.xldx,block.yldx,block.zldx)来定位到指定的线程。这就是我们CUDA的线程组织结构。

这里想谈谈SP和SM(流处理器)

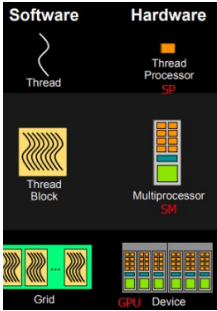
SP:最基本的处理单元,streaming processor,也称为CUDA core,最后具体的指令和任务都是在SP上处理的。GPU进行并行计算,也就是很多个SP同时做处理。

SM:多个SP加上其他的一些资源组成一个streaming multiprocessor,也叫GPU大核,其他资源如:warp scheduler,register,shared memory等。SM可以看做GPU的心脏(对比CPU核心),register和shared memory是SM的稀缺资源。CUDA将这些资源分配给所有驻留在SM中的threads。因此,这些有限的资源就使每个SM中active warps有非常严格的限制,也就限制了并行能力。

注意:每个SM包含的SP数量依据GPU架构而不同,Fermi架构GF100是32个,GF10X是48个,Kepler架构都是192个,Maxwell都是128个。

SP是线程执行的硬件单位，SM中包含多个SP，一个GPU可以有多个SM(比如16个)，最终一个GPU可能包含有上千个SP。这么多核心“同时运行”，速度可想而知，这个引号只是想表明实际上，软件逻辑上是所有SP是并行的，但是物理上并不是所有SP都能同时执行计算(比如我们只有8个SM却有1024个线程块需要调度处理)，因为有些会处于挂起，就绪等其他状态，这有关GPU的线程调度。

下面这个图得从硬件角度和软件角度解释CUDA的线程模型。



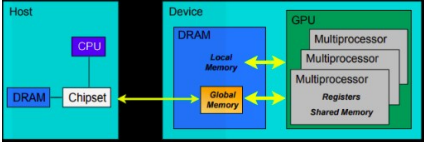
- 每个线程由每个线程处理器 (SP) 执行
- 线程块由多核处理器 (SM) 执行
- 一个kernel其实由一个grid来执行，一个kernel一次只能在一个GPU上执行

1.3 CUDA 内存模型

CUDA中的内存模型分为以下几个层次：

每个线程都用自己的registers(寄存器)
每个线程都有自己的local memory(局部内存)
每个线程块内都有自己的shared memory(共享内存)，所有线程块内的所有线程共享这段内存资源
每个grid都有自己的global memory(全局内存)，不同线程块的线程都可使用
每个grid都有自己的constant memory(常量内存)和texture memory(纹理内存)，不同线程块的线程都可使用
线程访问这几类存储器的速度是 register > local memory > shared memory > global memory

下面这幅图表示就是这些内存存在计算机架构中的所在层次。



1.4 CUDA 编程模型

上面讲了这么多硬件相关的知识点，现在终于可以开始说说CUDA是怎么写程序的了。

CUDA术语

Device = GPU
Host =CPU
Kernel = function that runs on the device
通过关键字就可以表示某个程序在CPU上跑还是在GPU上跑！如下表所示，比如我们用通过关键字就可以表示某个程序在CPU上跑还是在GPU上跑！如下表所示，比如我们用 global 定义一个kernel函数，就是CPU上调用，GPU上执行，注意 global 函数的返回值必须设置为void。

在...上执行	只能从以下方面调用：	
device float DeviceFunc()	device	device
global void KernelFunc()	device	host
host float HostFunc()	host	host

首先介绍在GPU内存分配回收内存的函数接口：

- cudaMalloc(): 在设备端分配global memory
- cudaFree(): 释放存储空间
- CPU的数据和GPU端数据做数据传输的函数接口是一样的，他们通过传递的函数实参(枚举类型)来表示传输方向：
cudaMemcpy(void dst, void src, size_t nbytes,enum cudaMemcpyKind direction)
其中:enum cudaMemcpyKind:
- cudaMemcpyHostToDevice (CPU到GPU)
- cudaMemcpyDeviceToHost (GPU到CPU)
- cudaMemcpyDeviceToDevice (GPU到GPU)

第三个编程要点是:怎么用代码表示线程组织模型？

我们可以用dim3类来表示网格和线程块的组织方式，网格grid可以表示为一维和二维格式，线程块block可以表示为一维、二维和三维的数据格式。

```
1 | dim3 DimGrid(100, 50); //5000个线程块，维度是100*50
2 | dim3 DimBlock(4, 8, 8); //每个线程块内包含256个线程，线程块内的维度是4*8*8
```

1.使用N个线程块，每一个线程块只有一个线程，即

```
1 | dim3 dimGrid(N);
2 | dim3 dimBlock(1);
```

此时的线程号的计算方式就是: threadIdx = blockIdx.x;

2.使用M×N个线程块，每个线程块1个线程

由于线程块是2维的，故可以看做是一个M*N的2维矩阵，其线程号有两个维度，即：

```
1 | dim3 dimGrid(M,N);
2 | dim3 dimBlock(1);
```

其中: blockIdx.x 取值0到M-1 blockIdx.y 取值0到N-1

这种情况一般用于处理2维数据结构，比如2维图像。每一个像素用一个线程来处理，此时需要线程号来映射图像像素的对应位置，如

pos = blockIdx.y * blockDim.x + blockIdx.x; //其中gridDim.x等于M

3.使用一个线程块，该线程具有N个线程，即

```
1 | dim3 dimGrid(1);
2 | dim3 dimBlock(N);
```

此时线程号的计算方式为:threadId = threadIdx.x;
其中threadIdx的范围是0到N-1, 对于这种情况, 可以看做是一个行向量, 行向量中的每一个元素的每一个元素对应着一个线程。

4.使用M个线程块, 每个线程块内含有N个线程, 即

```
1 |         dim3 dimGrid(M);
2 |         dim3 dimBlock(N);
```

这种情况, 可以把它想象成二维矩阵, 矩阵的行与线程块对应, 矩阵的列与线程编号对应, 那线程号的计算方式为: threadIdx = threadIdx.x + blockIdx*blockDim.x;

5.使用M*N的二维线程块, 每一个线程块具有P*Q个线程, 即

```
1 |         dim3 dimGrid(M, N);
2 |         dim3 dimBlock(P, Q);
```

这种情况其实是我们遇到的最多情况, 特别适用于处理具有二维数据结构的算法, 比如图像处理领域。
其索引有两个维度:

```
1 |         threadId.x = blockIdx.x*blockDim.x+threadIdx.x;
2 |         threadId.y = blockIdx.y*blockDim.y+threadIdx.y;
```

上述公式就是把线程和线程块的索引映射为图像像素坐标的计算方法。

1.5 CUDA 案例

1. 首先我们编写一个程序, 查看我们GPU的一些硬件配置情况

```
1 | #include "device_launch_parameters.h"
2 | #include "cuda_runtime.h"
3 | #include <iostream>
4 |
5 | int main()
6 | {
7 |     int deviceCount;
8 |     cudaGetDeviceCount(&deviceCount);
9 |     for(int i=0;i<deviceCount;i++)
10 |     {
11 |         cudaDeviceProp devProp;
12 |         cudaGetDeviceProperties(&devProp, i);
13 |         std::cout << "使用GPU device " << i << ": " << devProp.name << std::endl;
14 |         std::cout << "设备全局内存总量: " << devProp.totalGlobalMem / 1024 / 1024 <<
15 |         "MB" << std::endl;
16 |         std::cout << "SM的数量:" << devProp.multiProcessorCount << std::endl;
17 |         std::cout << "每个线程块的共享内存大小:" << devProp.sharedMemPerBlock /
18 |         1024.0 << " KB" << std::endl;
19 |         std::cout << "每个线程块的最大线程数:" << devProp.maxThreadsPerBlock <<
20 |         std::endl;
21 |         std::cout << "设备上每个线程块(Block)种可用的32位寄存器数量: " <<
22 |         devProp.regsPerBlock << std::endl;
23 |         std::cout << "每个EM的最大线程数:" << devProp.maxThreadsPerMultiProcessor <<
24 |         std::endl;
25 |         std::cout << "每个EM的最大线程束数:" << devProp.maxThreadsPerMultiProcessor
26 |         / 32 << std::endl;
27 |         std::cout << "设备上多处理器的数量: " << devProp.multiProcessorCount <<
28 |         std::endl;
29 |         std::cout << "===== " <<
30 |         std::endl;
31 |     }
32 |     return 0;
33 | }
```

我们利用nvcc来编译程序。

```
1 | nvcc test1.cu -o test1
```

输出结果:因为我的服务器是8个TITAN GPU

2. 第一个计算任务:将两个元素数目为1024*1024的float数组相加。

CPU

```
1 | #include <iostream>
2 | #include <stdlib.h>
3 | #include <sys/time.h>
4 | #include <math.h>
5 |
6 | using namespace std;
7 |
8 | int main()
9 | {
10 |     struct timeval start, end;
11 |     gettimeofday( &start, NULL );
12 |     float*A, *B, *C;
13 |     int n = 1024 * 1024;
14 |     int size = n * sizeof(float);
15 |     A = (float*)malloc(size);
16 |     B = (float*)malloc(size);
17 |     C = (float*)malloc(size);
18 |
19 |     for(int i=0;i<n;i++)
20 |     {
21 |         A[i] = 90.0;
22 |         B[i] = 10.0;
23 |     }
24 |
25 |     for(int i=0;i<n;i++)
26 |     {
27 |         C[i] = A[i] + B[i];
28 |     }
29 |
30 |     float max_error = 0.0;
31 |     for(int i=0;i<n;i++)
32 |     {
33 |         max_error += fabs(100.0-C[i]);
```

```

34     }
35     cout << "max_error is " << max_error << endl;
36     gettimeofday( &end, NULL );
37     int timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec -
38     start.tv_usec;
39     cout << "total time is " << timeuse/1000 << "ms" <<endl;
40     return 0;
    }
}

```

CPU方式输出结果

```

1 | max_error is 0
2 | total time is 22ms

```

GPU 编程要点：

1. 每个Block中的Thread数最大不超过512；
2. 为了充分利用SM, Block数尽可能多, >100。

```

1 | #include "cuda_runtime.h"
2 | #include <stdlib.h>
3 | #include <iostream>
4 | #include <sys/time.h>
5 |
6 | using namespace std;
7 |
8 | __global__ void Plus(float A[], float B[], float C[], int n)
9 | {
10 |     int i = blockDim.x * blockIdx.x + threadIdx.x;
11 |     C[i] = A[i] + B[i];
12 | }
13 |
14 | int main()
15 | {
16 |     struct timeval start, end;
17 |     gettimeofday( &start, NULL );
18 |     float*A, *Ad, *B, *Bd, *C, *Cd;
19 |     int n = 1024 * 1024;
20 |     int size = n * sizeof(float);
21 |
22 |     // CPU端分配内存
23 |     A = (float*)malloc(size);
24 |     B = (float*)malloc(size);
25 |     C = (float*)malloc(size);
26 |
27 |     // 初始化数组
28 |     for(int i=0;i<n;i++)
29 |     {
30 |         A[i] = 90.0;
31 |         B[i] = 10.0;
32 |     }
33 |
34 |     // GPU端分配内存
35 |     cudaMalloc((void**)&Ad, size);
36 |     cudaMalloc((void**)&Bd, size);
37 |     cudaMalloc((void**)&Cd, size);
38 |
39 |     // CPU的数据拷贝到GPU端
40 |     cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
41 |     cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
42 |     cudaMemcpy(Cd, C, size, cudaMemcpyHostToDevice);
43 |
44 |     // 定义kernel执行配置, (1024*1024/512)个block, 每个block里面有512个线程
45 |     dim3 dimBlock(512);
46 |     dim3 dimGrid(n/512);
47 |
48 |     // 执行kernel
49 |     Plus<<<dimGrid, dimBlock>>>(Ad, Bd, Cd, n);
50 |
51 |     // 将在GPU端计算好的结果拷贝回CPU端
52 |     cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
53 |
54 |     // 校验误差
55 |     float max_error = 0.0;
56 |     for(int i=0;i<n;i++)
57 |     {
58 |         max_error += fabs(100.0 - C[i]);
59 |     }
60 |
61 |     cout << "max error is " << max_error << endl;
62 |
63 |     // 释放CPU端、GPU端的内存
64 |     free(A);
65 |     free(B);
66 |     free(C);
67 |     cudaFree(Ad);
68 |     cudaFree(Bd);
69 |     cudaFree(Cd);
70 |     gettimeofday( &end, NULL );
71 |     int timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec -
72 |     start.tv_usec;
73 |     cout << "total time is " << timeuse/1000 << "ms" <<endl;
74 |     return 0;
    }
}

```

GPU方式输出结果

```

1 | max_error is 0
2 | total time is 29ms

```

使用CUDA编程时我们看不到for循环了，因为CPU编程的循环已经被分散到各个thread上做了，所以我们也看到不到for一类的语句。从结果上看，CPU的循环计算的速度比GPU计算快多了，原因就在于CUDA中有大量的内存拷贝操作（数据传输花费了大量时间，而计算时间却非常少），如果计算量比较小的话，CPU计算会更合适一些。

3. 下面计算一个稍微复杂的例子，矩阵加法，即对两个矩阵对应坐标的元素相加后的结果存储在第三个的对应位置的元素上。

值得注意的是, 这个计算任务我采用了二维数组的计算方式, 注意一下二维数组在CUDA编程中的写法。

CPU版本


```

1  #include <stdlib.h>
2  #include <iostream>
3  #include <sys/time.h>
4  #include <math.h>
5
6  #define ROWS 1024
7  #define COLS 1024
8
9  using namespace std;
10
11 int main()
12 {
13     struct timeval start, end;
14     gettimeofday( &start, NULL );
15     int *A, **A_ptr, *B, **B_ptr, *C, **C_ptr;
16     int total_size = ROWS*COLS*sizeof(int);
17     A = (int*)malloc(total_size);
18     B = (int*)malloc(total_size);
19     C = (int*)malloc(total_size);
20     A_ptr = (int**)malloc(ROWS*sizeof(int*));
21     B_ptr = (int**)malloc(ROWS*sizeof(int*));
22     C_ptr = (int**)malloc(ROWS*sizeof(int*));
23
24     //CPU一维数组初始化
25     for(int i=0;i<ROWS*COLS;i++)
26     {
27         A[i] = 80;
28         B[i] = 20;
29     }
30
31     for(int i=0;i<ROWS;i++)
32     {
33         A_ptr[i] = A + COLS*i;
34         B_ptr[i] = B + COLS*i;
35         C_ptr[i] = C + COLS*i;
36     }
37
38     for(int i=0;i<ROWS;i++)
39         for(int j=0;j<COLS;j++)
40         {
41             C_ptr[i][j] = A_ptr[i][j] + B_ptr[i][j];
42         }
43
44     //检查结果
45     int max_error = 0;
46     for(int i=0;i<ROWS*COLS;i++)
47     {
48         //cout << C[i] << endl;
49         max_error += abs(100-C[i]);
50     }
51
52     cout << "max_error is " << max_error << endl;
53     gettimeofday( &end, NULL );
54     int timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec -
55     start.tv_usec;
56     cout << "total time is " << timeuse/1000 << "ms" << endl;
57
58     return 0;
59 }

```

CPU方式输出

```

1  | max_error is 0
2  | total time is 29ms

```

GPU版本

```

#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <sys/time.h>
#include <stdio.h>
#include <math.h>
#define Row 1024
#define Col 1024

__global__ void addKernel(int **C, int **A, int ** B)
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    int idy = threadIdx.y + blockDim.y * blockIdx.y;
    if (idx < Col && idy < Row) {
        C[idy][idx] = A[idy][idx] + B[idy][idx];
    }
}

int main()
{
    struct timeval start, end;
    gettimeofday( &start, NULL );

    int **A = (int **)malloc(sizeof(int*) * Row);
    int **B = (int **)malloc(sizeof(int*) * Row);
    int **C = (int **)malloc(sizeof(int*) * Row);
    int *dataA = (int *)malloc(sizeof(int) * Row * Col);
    int *dataB = (int *)malloc(sizeof(int) * Row * Col);
    int *dataC = (int *)malloc(sizeof(int) * Row * Col);

    int **d_A;
    int **d_B;
    int **d_C;
    int *d_dataA;
    int *d_dataB;
    int *d_dataC;

    //malloc device memory
    cudaMalloc((void**)&d_A, sizeof(int **) * Row);
    cudaMalloc((void**)&d_B, sizeof(int **) * Row);
    cudaMalloc((void**)&d_C, sizeof(int **) * Row);

```

```

    cudaMalloc((void**)&d_dataA, sizeof(int) * Row * Col);
    cudaMalloc((void**)&d_dataB, sizeof(int) * Row * Col);
    cudaMalloc((void**)&d_dataC, sizeof(int) * Row * Col);

    //set value
    for (int i = 0; i < Row * Col; i++) {
        dataA[i] = 90;
        dataB[i] = 10;
    }

    //将主机指针A指向设备数据位置, 目的是让设备二级指针能够指向设备数据一级指针
    //A 和 dataA 都传到了设备上, 但是二者还没有建立对应关系
    for (int i = 0; i < Row; i++) {
        A[i] = d_dataA + Col * i;
        B[i] = d_dataB + Col * i;
        C[i] = d_dataC + Col * i;
    }

    cudaMemcpy(d_A, A, sizeof(int) * Row, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, sizeof(int) * Row, cudaMemcpyHostToDevice);
    cudaMemcpy(d_C, C, sizeof(int) * Row, cudaMemcpyHostToDevice);
    cudaMemcpy(d_dataA, dataA, sizeof(int) * Row * Col, cudaMemcpyHostToDevice);
    cudaMemcpy(d_dataB, dataB, sizeof(int) * Row * Col, cudaMemcpyHostToDevice);
    dim3 threadPerBlock(16, 16);
    dim3 blockNumber( (Col + threadPerBlock.x - 1) / threadPerBlock.x, (Row + threadPerBlock.y - 1) / threadPerBlock.y );

    printf("Block(%d,%d) Grid(%d,%d).\n", threadPerBlock.x, threadPerBlock.y, blockNumber.x, blockNumber.y);
    addKernel << <blockNumber, threadPerBlock >> > (d_C, d_A, d_B);

    //拷贝计算数据-一级数据指针
    cudaMemcpy(dataC, d_dataC, sizeof(int) * Row * Col, cudaMemcpyDeviceToHost);

    int max_error = 0;
    for(int i=0;i<Row*Col;i++)
    {
        //printf("%d\n", dataC[i]);
        max_error += abs(100-dataC[i]);
    }

    //释放内存
    free(A);
    free(B);
    free(C);
    free(dataA);
    free(dataB);
    free(dataC);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    cudaFree(d_dataA);
    cudaFree(d_dataB);
    cudaFree(d_dataC);

    printf("max_error is %d\n", max_error);
    gettimeofday( &end, NULL );
    int timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec - start.tv_usec;
    printf("total time is %d ms\n", timeuse/1000);

    return 0;
}

```

```

1  #include "cuda_runtime.h"
2  #include "device_launch_parameters.h"
3  #include <sys/time.h>
4  #include <stdio.h>
5  #include <math.h>
6  #define Row 1024
7  #define Col 1024
8
9
10 __global__ void addKernel(int **C, int **A, int ** B)
11 {
12     int idx = threadIdx.x + blockDim.x * blockIdx.x;
13     int idy = threadIdx.y + blockDim.y * blockIdx.y;
14     if (idx < Col && idy < Row) {
15         C[idy][idx] = A[idy][idx] + B[idy][idx];
16     }
17 }
18
19 int main()
20 {
21     struct timeval start, end;
22     gettimeofday( &start, NULL );
23
24     int **A = (int **)malloc(sizeof(int*) * Row);
25     int **B = (int **)malloc(sizeof(int*) * Row);
26     int **C = (int **)malloc(sizeof(int*) * Row);
27     int *dataA = (int *)malloc(sizeof(int) * Row * Col);
28     int *dataB = (int *)malloc(sizeof(int) * Row * Col);
29     int *dataC = (int *)malloc(sizeof(int) * Row * Col);
30     int **d_A;
31     int **d_B;
32     int **d_C;
33     int *d_dataA;
34     int *d_dataB;
35     int *d_dataC;
36     //malloc device memory
37     cudaMalloc((void**)&d_A, sizeof(int) * Row);
38     cudaMalloc((void**)&d_B, sizeof(int) * Row);
39     cudaMalloc((void**)&d_C, sizeof(int) * Row);
40     cudaMalloc((void**)&d_dataA, sizeof(int) * Row * Col);
41     cudaMalloc((void**)&d_dataB, sizeof(int) * Row * Col);
42     cudaMalloc((void**)&d_dataC, sizeof(int) * Row * Col);
43     //set value
44     for (int i = 0; i < Row * Col; i++) {
45         dataA[i] = 90;
46         dataB[i] = 10;

```

```

47     }
48     //将主机指针A指向设备数据位置, 目的是让设备二级指针能够指向设备数据一级指针
49     //A 和 dataA 都传到了设备上, 但是二者还没有建立对应关系
50     for (int i = 0; i < Row; i++) {
51         A[i] = d_dataA + Col * i;
52         B[i] = d_dataB + Col * i;
53         C[i] = d_dataC + Col * i;
54     }
55
56     cudaMemcpy(d_A, A, sizeof(int*) * Row, cudaMemcpyHostToDevice);
57     cudaMemcpy(d_B, B, sizeof(int*) * Row, cudaMemcpyHostToDevice);
58     cudaMemcpy(d_C, C, sizeof(int*) * Row, cudaMemcpyHostToDevice);
59     cudaMemcpy(d_dataA, dataA, sizeof(int) * Row * Col, cudaMemcpyHostToDevice);
60     cudaMemcpy(d_dataB, dataB, sizeof(int) * Row * Col, cudaMemcpyHostToDevice);
61     dim3 threadPerBlock(16, 16);
62     dim3 blockNumber( (Col + threadPerBlock.x - 1)/ threadPerBlock.x, (Row +
threadPerBlock.y - 1) / threadPerBlock.y );
63     printf("Block(%d,%d)  Grid(%d,%d).\n", threadPerBlock.x, threadPerBlock.y,
blockNumber.x, blockNumber.y);
64     addKernel << <blockNumber, threadPerBlock >>> (d_C, d_A, d_B);
65     //拷贝计算数据-一级数据指针
66     cudaMemcpy(dataC, d_dataC, sizeof(int) * Row * Col, cudaMemcpyDeviceToHost);
67
68     int max_error = 0;
69     for(int i=0;i<Row*Col;i++)
70     {
71         //printf("%d\n", dataC[i]);
72         max_error += abs(100-dataC[i]);
73     }
74
75     //释放内存
76     free(A);
77     free(B);
78     free(C);
79     free(dataA);
80     free(dataB);
81     free(dataC);
82     cudaFree(d_A);
83     cudaFree(d_B);
84     cudaFree(d_C);
85     cudaFree(d_dataA);
86     cudaFree(d_dataB);
87     cudaFree(d_dataC);
88
89     printf("max_error is %d\n", max_error);
90     gettimeofday( &end, NULL );
91     int timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec -
start.tv_usec;
92     printf("total time is %d ms\n", timeuse/1000);
93
94     return 0;
95 }
96

```

GPU输出

```

1 | Block(16,16)  Grid(64,64).
2 | max_error is 0
3 | total time is 442 ms

```

从结果看出, CPU计算时间还是比GPU的计算时间短。这里需要指出的是, 这种二维数组的程序写法的效率并不高(虽然比较符合我们的思维方式), 因为我们做了两次访存操作。所以一般而言, 做高性能计算一般不会采取这种编程方式。

最后一个例子我们将计算一个更加复杂的任务, 矩阵乘法

4 回顾一下矩阵乘法

两矩阵相乘, 左矩阵第一行乘以右矩阵第一列(分别相乘, 第一个数乘第一个数), 乘完之后相加, 即为结果的第一行第一列的数, 依次往下算, 直到计算完所有矩阵元素。

$$C = AB = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} = \begin{pmatrix} 1 \times 1 + 2 \times 2 + 3 \times 3 & 1 \times 4 + 2 \times 5 + 3 \times 6 \\ 4 \times 1 + 5 \times 2 + 6 \times 3 & 4 \times 4 + 5 \times 5 + 6 \times 6 \end{pmatrix} = \begin{pmatrix} 14 & 32 \\ 32 & 77 \end{pmatrix}$$

CPU版本

```

1 | #include <iostream>
2 | #include <stdlib.h>
3 | #include <sys/time.h>
4 |
5 | #define ROWS 1024
6 | #define COLS 1024
7 |
8 | using namespace std;
9 |
10 | void matrix_mul_cpu(float* M, float* N, float* P, int width)
11 | {
12 |     for(int i=0;i<width;i++)
13 |         for(int j=0;j<width;j++)
14 |         {
15 |             float sum = 0.0;
16 |             for(int k=0;k<width;k++)
17 |             {
18 |                 float a = M[i*width+k];
19 |                 float b = N[k*width+j];
20 |                 sum += a*b;
21 |             }
22 |             P[i*width+j] = sum;
23 |         }
24 |     }
25 |
26 | int main()
27 | {
28 |     struct timeval start, end;
29 |     gettimeofday( &start, NULL );
30 |     float *A, *B, *C;
31 |     int total_size = ROWS*COLS*sizeof(float);
32 |     A = (float*)malloc(total_size);
33 |     B = (float*)malloc(total_size);
34 |     C = (float*)malloc(total_size);
35 |
36 |     //CPU一维数组初始化
37 |     for(int i=0;i<ROWS*COLS;i++)

```

```
38     A[i] = 80.0;
39     B[i] = 20.0;
40 }
41
42
43 matrix_mul_cpu(A, B, C, COLS);
44
45 gettimeofday( &end, NULL );
46 int timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec -
47 start.tv_usec;
48 cout << "total time is " << timeuse/1000 << "ms" <<endl;
49
50 return 0;
51 }
```

CPU输出

```
1 | total time is 7617ms
```

GPU版本

梳理一下CUDA求解矩阵乘法的思路:因为 $C=A \times B$, 我们利用每个线程求解C矩阵每个(x, y)的元素, 每个线程载入A的一行和B的一列, 遍历各自行列元素, 对A、B对应的元素做一次乘法和一次加法。

```
1  #include "cuda_runtime.h"
2  #include "device_launch_parameters.h"
3  #include <sys/time.h>
4  #include <stdio.h>
5  #include <math.h>
6  #define Row 1024
7  #define Col 1024
8
9
10 __global__ void matrix_mul_gpu(int *M, int* N, int* P, int width)
11 {
12     int i = threadIdx.x + blockDim.x * blockIdx.x;
13     int j = threadIdx.y + blockDim.y * blockIdx.y;
14
15     int sum = 0;
16     for(int k=0;k<width;k++)
17     {
18         int a = M[j*width+k];
19         int b = N[k*width+i];
20         sum += a*b;
21     }
22     P[j*width+i] = sum;
23 }
24
25 int main()
26 {
27     struct timeval start, end;
28     gettimeofday( &start, NULL );
29
30     int *A = (int *)malloc(sizeof(int) * Row * Col);
31     int *B = (int *)malloc(sizeof(int) * Row * Col);
32     int *C = (int *)malloc(sizeof(int) * Row * Col);
33     //malloc device memory
34     int *d_dataA, *d_dataB, *d_dataC;
35     cudaMalloc((void**)&d_dataA, sizeof(int) *Row*Col);
36     cudaMalloc((void**)&d_dataB, sizeof(int) *Row*Col);
37     cudaMalloc((void**)&d_dataC, sizeof(int) *Row*Col);
38     //set value
39     for (int i = 0; i < Row*Col; i++) {
40         A[i] = 90;
41         B[i] = 10;
42     }
43
44     cudaMemcpy(d_dataA, A, sizeof(int) * Row * Col, cudaMemcpyHostToDevice);
45     cudaMemcpy(d_dataB, B, sizeof(int) * Row * Col, cudaMemcpyHostToDevice);
46     dim3 threadPerBlock(16, 16);
47     dim3 blockNumber((Col+threadPerBlock.x-1)/ threadPerBlock.x,
48 (Row+threadPerBlock.y-1)/ threadPerBlock.y );
49     printf("Block(%d,%d)  Grid(%d,%d).\n", threadPerBlock.x, threadPerBlock.y,
50 blockNumber.x, blockNumber.y);
51     matrix_mul_gpu << <blockNumber, threadPerBlock >> > (d_dataA, d_dataB,
52 d_dataC, Col);
53     //拷贝计算数据-一级数据指针
54     cudaMemcpy(C, d_dataC, sizeof(int) * Row * Col, cudaMemcpyDeviceToHost);
55
56     //释放内存
57     free(A);
58     free(B);
59     free(C);
60     cudaFree(d_dataA);
61     cudaFree(d_dataB);
62     cudaFree(d_dataC);
63
64     gettimeofday( &end, NULL );
65     int timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec -
```

GPU输出

```
1 | Block(16,16)  Grid(64,64).
2 | total time is 506 ms
```

从这个矩阵乘法任务可以看出, 我们通过GPU进行并行计算的方式仅花费了0.5秒, 但是CPU串行计算方式却花费了7.6秒, 计算速度提升了十多倍, 可见并行计算的威力!

2. Nvidia GPU架构

2.1 硬件架构

■ SP:最基本的处理单元, streaming processor, 也称为CUDA core. 最后具体的指令和任务都是在SP上处理的。GPU进行并行计算, 也就是很多个SP同时做处理。

关于作者



关注者 2

关注

博客

泡泡

积分

勋章

cartographer 生成子图

阅读数 687 评论数 0

move_bae参数介绍

阅读数 70 评论数 0

cartographer 前端两种匹配

阅读数 932 评论数 0



相关推荐

git提交PR(pull requests)全过程记录

阅读数 1999 评论数 0

Ubuntu18下安装ROS1&2

阅读数 9693 评论数 0

zotero自动从自己的文献库条目中下载pdf文件

阅读数 9122 评论数 0

PID一点都不准! (非原创, 出自哪忘了, 内容很好, 分享给大家)

阅读数 3673 评论数 1

ROS答案(八)——进行多传感器融合的思路?(视觉SLAM和激光...

阅读数 4465 评论数 2

如何检索国外的博士论文

阅读数 4943 评论数 0

热门泡泡

30秒: 失眠, 聊聊自己搞ROS的心得体会吧

ros学习路线

30秒: TF_REPEATED_DATA ignoring data错误

各位大佬, 有什么ROS定位算法推荐吗

5秒: ros中启动gazebo时报错

5秒: 想买能用ROS2的开发套件. 或者开发板



首页

博客

泡泡

精品课程

项目

高校服务

企业服务

编队

发布



}

GPU输出

```
1 | Block(16,16)  Grid(64,64).
2 | total time is 506 ms
```

从这个矩阵乘法任务可以看出, 我们通过GPU进行并行计算的方式仅花费了0.5秒, 但是CPU串行计算方式却花费了7.6秒, 计算速度提升了十多倍, 可见并行计算的威力!

2. Nvidia GPU架构

2.1 硬件架构

■ SP:最基本的处理单元, streaming processor, 也称为CUDA core. 最后具体的指令和任务都是在SP上处理的。GPU进行并行计算, 也就是很多个SP同时做处理。

■ SM: 多个SP加上其他的一些资源组成一个streaming multiprocessor, 也叫GPU大核, 其他资源如: warp scheduler, register, shared memory等。SM可以看做GPU的心脏(对比CPU核心), register和shared memory是SM的稀缺资源。CUDA将这些资源分配给所有驻留在SM中的threads。因此, 这些有限的资源就使每个SM中active warps有非常严格的限制, 也就限制了并行能力。

■ 具有Tesla架构的GPU是具有芯片共享存储器的一组SIMT(单指令多线程)多处理器。它以一个可伸缩的多线程流处理器(Streaming Multiprocessors, SMs)阵列为中心实现了MIMD(多指令多数据)的异步并行机制, 其中每个多处理器包含多个标量处理器(Scalar Processor, SP)。为了管理运行各种不同程序的数百个线程, SIMT架构的多处理器会将各线程映射到一个标量处理器核心, 各标量线程使用自己的指令地址和寄存器状态独立执行。

每个多处理器(Multiprocessor)都有一个属于以下四种类型之一的芯片存储器:

每个处理器上有一组本地 32 位寄存器(Registers)并行数据缓存或共享存储器(Shared Memory)。由所有标量处理器核心共享, 共享存储空间就位于此处; 只读固定缓存(Constant Cache), 由所有标量处理器核心共享, 可加速从固定存储器空间进行的读取操作(这是设备存储器的一个只读区域)一个只读纹理缓存(Texture Cache), 由所有标量处理器核心共享, 加速从纹理存储空间进行的读取操作(这是设备存储器的一个只读区域), 每个多处理器都会通过实现不同寻址模型和数据过滤的纹理单元访问纹理缓存。多处理器 SIMT 单元以32个并行线程为一组来创建、管理、调度和执行线程, 这样的线程组称为 warp 块(束), 即以线程束为调度单位, 但只有所有32个线程都在诸如内存读取这样的操作时, 它们就会被挂起, 如下所示的状态变化。当主机CPU上的CUDA程序调用内核网格时, 网格的块将被枚举并分发到具有可用执行容量的多处理器;SIMT 单元会选择一个已准备好执行的 warp 块, 并将下一条指令发送到该 warp 块的活动线程。一个线程块的线程在一个多处理器上并发执行, 在线程块终止时, 将在空闲多处理器上启动新块。

2.2 软件架构

CUDA是一种新的操作GPU计算的硬件和软件架构, 它将GPU视作一个数据并行计算设备, 而且无需把这些计算映射到图形API。操作系统的多任务机制可以同时管理CUDA访问和图形程序的运行库, 其计算特性支持利用CUDA直观地编写GPU核心程序。目前Tesla架构具有在笔记本电脑、台式机、工作站和服务器上的广泛可用性。配以C/C++语言的编程环境和CUDA软件, 使这种架构得以成为最优秀的超级计算平台。

CUDA在软件方面组成有:一个CUDA库、一个应用程序编程接口(API)及其运行库(Runtime)、两个较高级别的通用数学库, 即CUFFT和CUBLAS。CUDA改进了DRAM的读写灵活性, 使得GPU与CPU的机制相吻合。另一方面, CUDA提供了片上(on-chip)共享内存, 使得线程之间可以共享数据。应用程序可以利用共享内存来减少DRAM的数据传送, 更多的依赖DRAM的内存带宽。

- thread: 一个CUDA的并行程序会被以许多个threads来执行。
- block: 数个threads会被群组成一个block, 同一个block中的threads可以同步, 也可以通过shared memory通信。
- grid: 多个blocks则会再构成grid。
- warp: GPU执行程序时的调度单位, 目前cuda的warp的大小为32, 同在一个warp的线程, 以不同数据资源执行相同的指令,这就是所谓 SIMT。

2.3 CUDA C编程入门

CUDA程序构架分为两部分: Host和Device。一般而言, Host指的是CPU, Device指的是GPU。在CUDA程序构架中, 主程序还是由 CPU 来执行, 而当遇到数据并行处理的部分, CUDA 就会将程序编译成 GPU 能执行的程序, 并传送到GPU, 而这个程序在CUDA里称做核(kernel)。CUDA允许程序员定义称为核的C语言函数, 从而扩展了 C 语言, 在调用此类函数时, 它将由N个不同的CUDA线程并行执行N次, 这与普通的C语言函数只执行一次的方式不同。执行核的每个线程都会被分配一个独特的线程ID, 可通过内置的threadIdx变量在内核中访问此ID。

在 CUDA 程序中, 主程序在调用任何 GPU 内核之前, 必须对核进行执行配置, 即确定线程块数和每个线程块中的线程数以及共享内存大小。

1.线程层次结构

在GPU中要执行的线程, 根据最有效的数据共享来创建块(Block), 其类型有一维、二维或三维。在同一个块内的线程可彼此协作, 通过一些共享存储器来共享数据, 并同步其执行来协调存储器访问。一个块中的所有线程都必须位于同一个处理器核心中。因而, 一个处理器核心的有限存储器资源制约了每个块的线程数量。在早起的 NVIDIA 架构中, 一个线程块最多可以包含 512 个线程, 而在后期出现的一些设备中则最多可支持1024个线程。一般 GPGPU 程序线程数目是很多的, 所以不能把所有的线程都塞到同一个块里。但一个内核可由多个大小相同的线程块同时执行, 因而线程总数应等于每个块的线程数乘以块的数量。这些同样维度和大小的块将组织为一个一维或二维线程块网格(Grid)

核函数只能在主机端调用, 其调用形式为: Kernel<<<Dg,Db, Ns,S>>>(param list)

Dg:用于定义整个grid的维度和尺寸, 即一个grid有多少个block。为dim3类型。Dim3 Dg(Dg.x, Dg.y, 1)表示grid中每行有Dg.x个block, 每列有Dg.y个block, 第三维恒为1(目前一个核函数只有一个grid)。整个grid中共有Dg.x*Dg.y个block, 其中Dg.x和Dg.y最大值为65535。

Db:用于定义一个block的维度和尺寸, 即一个block有多少个thread。为dim3类型。Dim3 Db(Db.x, Db.y, Db.z)表示整个block中每行有Db.x个thread, 每列有Db.y个thread, 高度为Db.z。Db.x和Db.y最大值为512, Db.z最大值为62。一个block中共有Db.xDb.yDb.z个thread。计算能力为1.0,1.1的硬件该乘积的最大值为768, 计算能力为1.2,1.3的硬件支持的最大值为1024。

Ns:是一个可选参数, 用于设置每个block除了静态分配的shared Memory以外, 最多能动态分配的shared memory大小, 单位为byte。不需要动态分配时该值为0或省略不写。

S:是一个cudaStream_t类型的可选参数, 初始值为零, 表示该核函数处在哪个流之中。

cuda c基础

CUDA C是对C/C++语言进行拓展后形成的变种, 兼容C/C++语法, 文件类型为".cu"文件, 编译器为"nvcc", 相比传统的C/C++, 主要添加了以下几个方面:

函数类型限定符
执行配置运算符
五个内置变量
变量类型限定符
其他的还有数学函数、原子函数、纹理读取、绑定函数等
1)函数类型限定符
用来确定某个函数是在CPU还是GPU上运行, 以及这个函数是从CPU调用还是从GPU调用

device表示从GPU调用, 在GPU上执行
global表示从CPU调用, 在GPU上执行, 也称之为kernel函数
host表示在CPU上调用, 在CPU上执行

```
1 #include <stdio.h>
2
3 __device__ void device_func(void) {
4 }
5
6 __global__ void global_func(void) {
7     device_func();
8 }
9
10 int main() {
11     printf("%s\n", __FILE__);
12     global_func<<<1,1>>>();
```

```
13 |         return 0;
14 |     }
}

2) 执行配置运算符
执行配置运算符<<<<>>>, 用来传递内核函数的执行参数。格式如下:
kernel<<<gridDim, blockDim, memSize, stream>>>(para1, para2, ...);

gridDim表示网格的大小, 可以是1,2,3维
blockDim表示块的大小, 可以是1,2,3维
memSize表示动态分配的共享存储器大小, 默认为0
stream表示执行的流, 默认为0
para1, para2等为核函数参数

1 | #include <stdio.h>
2 |
3 | __global__ void func(int a, int b) {
4 | }
5 |
6 | int main() {
7 |     int a = 0, b = 0;
8 |     func<<<128, 128>>>(a, b);
9 |     func<<<dim3(128, 128), dim3(16, 16)>>>(a, b);
10 |    func<<<dim3(128, 128, 128), dim3(16, 16, 2)>>>(a, b);
11 |    return 0;
12 | }
```

3) 五个内置变量

这些内置变量用来在运行时获得Grid和Block的尺寸及线程索引等信息

gridDim: 包含三个元素x, y, z的结构体, 表示Grid在三个方向上的尺寸, 对应于执行配置中的第一个参数

blockDim: 包含三个元素x, y, z的结构体, 表示Block在三个方向上的尺寸, 对应于执行配置中的第二个参数

blockIdx: 包含三个元素x, y, z的结构体, 分别表示当前线程所在块在网格中x, y, z方向上的索引

threadIdx: 包含三个元素x, y, z的结构体, 分别表示当前线程在其所在块中x, y, z方向上的索引

warpSize: 表明warp的尺寸

4) 变量类型限定符

用来确定某个变量在设备上的内存位置

`_device_` 表示位于全局内存空间, 默认类型

`_share_` 表示位于共享内存空间

`_constant_` 表示位于常量内存空间

texture表示其绑定的变量可以被纹理缓存加速访问

3.cuda API

3.1 Runtime API函数

Runtime API 函数: cudaGetDeviceCount, cudaGetDeviceProperties, cudaSetDevice, 在cuda_runtime.h 头文件中

3.1.1.CUDA 初始化函数

由于是使用 Runtime API, 所以在文件开头要加入 cuda_runtime.h 头文件。

初始化函数包括一下几个步骤:

1. 获取 CUDA 设备数
- 可以通过 cudaGetDeviceCount 函数获取 CUDA 的设备数, 具体用法, 如下所示:

```
1 | int deviceCount;
2 | cudaGetDeviceCount(&deviceCount);
3 | if(deviceCount==0) {
4 |     fprintf(stderr, "There is no device.\n");
5 | }
```

获取 CUDA 设备属性

可以通过 cudaGetDeviceProperties 函数获取 CUDA 设备的属性, 具体用法, 如下所示:

```
1 | int i;
2 | for (i = 0; i < count; ++i) {
3 |     cudaDeviceProp prop;
4 |     if (cudaGetDeviceProperties(&prop, i) == cudaSuccess) {
5 |         if (prop.major >= 1) {
6 |             printDeviceProp(prop);
7 |             break;
8 |         }
9 |     }
10 | }
11 |
12 | // if can't find the device
13 | if (i == count) {
14 |     fprintf(stderr, "There is no device supporting CUDA 1.x.\n");
15 |     return false;
16 | }
```

函数通过引用传递 prop 关于属性的结构体, 并且列出主设备号大于 1 的设备属性, 其中设备属性通过函数 printDeviceProp 打印。打印函数如下所示:

```
1 | // function printDeviceProp
2 | void printDeviceProp(const cudaDeviceProp &prop)
3 | {
4 |     printf("Device Name : %s.\n", prop.name);
5 |     printf("totalGlobalMem : %d.\n", prop.totalGlobalMem);
6 |     printf("sharedMemPerBlock : %d.\n", prop.sharedMemPerBlock);
7 |     printf("regsPerBlock : %d.\n", prop.regsPerBlock);
8 |     printf("warpSize : %d.\n", prop.warpSize);
9 |     printf("memPitch : %d.\n", prop.memPitch);
10 |    printf("maxThreadsPerBlock : %d.\n", prop.maxThreadsPerBlock);
11 |    printf("maxThreadsDim[0 - 2] : %d %d %d.\n", prop.maxThreadsDim[0], prop.maxThreadsDim[1], prop.maxThreadsDim[2]);
12 |    printf("maxGridSize[0 - 2] : %d %d %d.\n", prop.maxGridSize[0], prop.maxGridSize[1], prop.maxGridSize[2]);
13 |    printf("totalConstMem : %d.\n", prop.totalConstMem);
14 |    printf("major.minor : %d.%d.\n", prop.major, prop.minor);
15 |    printf("clockRate : %d.\n", prop.clockRate);
16 |    printf("textureAlignment : %d.\n", prop.textureAlignment);
17 |    printf("deviceOverlap : %d.\n", prop.deviceOverlap);
18 |    printf("multiProcessorCount : %d.\n", prop.multiProcessorCount);
19 | }
```

设置 CUDA 设备
通过函数 cudaSetDevice 就可以设置 CUDA 设备了，具体用法，如下所示：

```
1 | cudaSetDevice(i);
```

CUDA 初始化完整代码

```
1 | #include <stdio.h>
2 | #include <cuda_runtime.h>
3 |
4 | void printDeviceProp(const cudaDeviceProp &prop)
5 | {
6 |     printf("Device Name : %s.\n", prop.name);
7 |     printf("totalGlobalMem : %d.\n", prop.totalGlobalMem);
8 |     printf("sharedMemPerBlock : %d.\n", prop.sharedMemPerBlock);
9 |     printf("regsPerBlock : %d.\n", prop.regsPerBlock);
10 |    printf("warpSize : %d.\n", prop.warpSize);
11 |    printf("memPitch : %d.\n", prop.memPitch);
12 |    printf("maxThreadsPerBlock : %d.\n", prop.maxThreadsPerBlock);
13 |    printf("maxThreadsDim[0 - 2] : %d %d %d.\n", prop.maxThreadsDim[0],
prop.maxThreadsDim[1], prop.maxThreadsDim[2]);
14 |    printf("maxGridSize[0 - 2] : %d %d %d.\n", prop.maxGridSize[0],
prop.maxGridSize[1], prop.maxGridSize[2]);
15 |    printf("totalConstMem : %d.\n", prop.totalConstMem);
16 |    printf("major.minor : %d.%d.\n", prop.major, prop.minor);
17 |    printf("clockRate : %d.\n", prop.clockRate);
18 |    printf("textureAlignment : %d.\n", prop.textureAlignment);
19 |    printf("deviceOverlap : %d.\n", prop.deviceOverlap);
20 |    printf("multiProcessorCount : %d.\n", prop.multiProcessorCount);
21 | }
22 |
23 | bool InitCUDA()
24 | {
25 |     //used to count the device numbers
26 |     int count;
27 |
28 |     // get the cuda device count
29 |     cudaGetDeviceCount(&count);
30 |     if (count == 0) {
31 |         fprintf(stderr, "There is no device.\n");
32 |         return false;
33 |     }
34 |
35 |     // find the device >= 1.X
36 |     int i;
37 |     for (i = 0; i < count; ++i) {
38 |         cudaDeviceProp prop;
39 |         if (cudaGetDeviceProperties(&prop, i) == cudaSuccess) {
40 |             if (prop.major >= 1) {
41 |                 printDeviceProp(prop);
42 |                 break;
43 |             }
44 |         }
45 |     }
46 |
47 |     // if can't find the device
48 |     if (i == count) {
49 |         fprintf(stderr, "There is no device supporting CUDA 1.x.\n");
50 |         return false;
51 |     }
52 |
53 |     // set cuda device
54 |     cudaSetDevice(i);
55 |
56 |     return true;
57 | }
58 |
59 | int main(int argc, char const *argv[])
60 | {
61 |     if (InitCUDA()) {
62 |         printf("CUDA initialized.\n");
63 |     }
64 |
65 |     return 0;
66 | }
```

3.1.3.Runtime API 函数解析

cudaGetDeviceCount

以 *count 形式返回可用于执行的计算能力大于等于 1.0 的设备数量。如果不存在此类设备，将返回 1

cudaGetDeviceProperties

返回关于计算设备的信息。返回值:cudaSuccess、cudaErrorInvalidDevice, 注，如果之前是异步启动，该函数可能返回错误码。

```
1 | struct cudaDeviceProp {
2 |     char name [256];
3 |     size_t totalGlobalMem;
4 |     size_t sharedMemPerBlock;
5 |     int regsPerBlock;
6 |     int warpSize;
7 |     size_t memPitch;
8 |     int maxThreadsPerBlock;
9 |     int maxThreadsDim [3];
10 |    int maxGridSize [3];
11 |    size_t totalConstMem;
12 |    int major;
13 |    int minor;
14 |    int clockRate;
15 |    size_t textureAlignment;
16 |    int deviceOverlap;
17 |    int multiProcessorCount;
```

定义：

name

用于标识设备的ASCII字符串；

totalGlobalMem

设备上可用的全局存储器的总量,以字节为单位；

sharedMemPerBlock

线程块可以使用的共享存储器的最大值,以字节为单位;多处理器上的所有线程块可以同时共享这些存储器；

regsPerBlock
线程块可以使用的32位寄存器的最大值;多处理器上的所有线程块可以同时共享这些寄存器;
warpSize
按线程计算的warp块大小;
memPitch
允许通过cudaMallocPitch()为包含存储器区域的存储器复制函数分配的最大间距(pitch),以字节为单位;
maxThreadsPerBlock
每个块中的最大线程数
maxThreadsDim[3]
块各个维度的最大值:
maxGridSize[3]
网格各个维度的最大值;
totalConstMem
设备上可用的不变存储器总量,以字节为单位;
major,minor
定义设备计算能力的主要修订号和次要修订号;
clockRate
以千赫为单位的时钟频率;
textureAlignment
对齐要求;与textureAlignment字节对齐的纹理基址无需对纹理取样应用偏移;
deviceOverlap
如果设备可在主机和设备之间并发复制存储器,同时又能执行内核,则此值为 1;否则此值为 0;
multiProcessorCount
设备上多处理器的数量。

cudaGetDeviceCount
将devi记录为活动主线程将执行设备码的设备。

4. 官网教程

4.1 CUDA模型和界面的编程指南。

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

4.1.1 引言

CPU和GPU之间浮点能力差异背后的原因是GPU专门用于计算密集型, 高度并行计算 - 正是图形渲染的关键 - 因此设计使得更多晶体管用于数据处理 而不是数据缓存和流量控制。
GPU特别适合干解决可以表示为数据并行计算的问题 - 在许多数据元素上并行执行相同的程序 - 具有高算术强度 - 算术运算与存储器操作的比率。因为对每个数据元素执行相同的程序, 所以对复杂流控制的要求较低, 并且因为它在许多数据元素上执行并且具有高算术强度, 所以可以通过计算而不是大数据缓存来隐藏存储器访问延迟。
数据并行处理将数据元素映射到并行处理线程。许多处理大型数据集的应用程序可以使用数据并行编程模型来加速计算。

4.1.2 编程模型

本章通过概述它们如何在C中公开, 介绍了CUDA编程模型背后的主要概念。编程接口中给出了对CUDA C的详细描述。

2.1. Kernels

CUDA C通过允许程序员定义称为内核的C函数来扩展C, 这些函数在被调用时由N个不同的CUDA线程并行执行N次, 而不是像常规C函数那样只执行一次。
使用 `__global__` 声明说明符定义内核, 并使用新的`<<< ... >>>`执行配置语法指定为给定内核调用执行该内核的CUDA线程数(请参阅C语言扩展)。执行内核的每个线程都有一个唯一的线程ID, 可以通过内置的threadIdx变量在内核中访问。
作为说明, 以下示例代码添加了两个大小为N的向量A和B, 并将结果存储到向量C中:

```
1 // Kernel definition
2 __global__ void VecAdd(float* A, float* B, float* C)
3 {
4     int i = threadIdx.x;
5     C[i] = A[i] + B[i];
6 }
7
8 int main()
9 {
10     ...
11     // Kernel invocation with N threads
12     VecAdd<<<1, N>>>>(A, B, C);
13     ...
14 }
```

这里, 执行VecAdd()的N个线程中的每一个执行一对成对添加。

2.2. Thread Hierarchy 线程层次结构

为方便起见, threadIdx是一个3分量向量, 因此可以使用一维, 二维或三维线程索引来识别线程, 形成一维, 二维或三维块。线程, 称为线程块。这提供了一种自然的方式来调用域中元素(如向量, 矩阵或体积)的计算。
线程的索引及其线程ID以直接的方式相互关联: 对于一维块, 它们是相同的; 对于二维大小块(Dx, Dy), 索引(x, y)的线程的线程ID是(x + y Dx); 对于尺寸为三维的块(Dx, Dy, Dz), 索引(x, y, z)的线程的线程ID是(x + y Dx + z Dx Dy)。
作为示例, 以下代码添加两个大小为NxN的矩阵A和B, 并将结果存储到矩阵C中:

```
1 // Kernel definition
2 __global__ void MatAdd(float A[N][N], float B[N][N],
3                       float C[N][N])
4 {
5     int i = threadIdx.x;
6     int j = threadIdx.y;
7     C[i][j] = A[i][j] + B[i][j];
8 }
9
10 int main()
11 {
12     ...
13     // Kernel invocation with one block of N * N * 1 threads
14     int numBlocks = 1;
15     dim3 threadsPerBlock(N, N);
16     MatAdd<<<numBlocks, threadsPerBlock>>>>(A, B, C);
17     ...
18 }
```

每个块的线程数有限制, 因为预期块的所有线程都驻留在同一处理器核心上, 并且必须共享该核心的有限内存资源。
在当前的GPU上, 线程块最多可包含1024个线程。

但是, 内核可以由多个同形状的线程块执行, 因此线程总数等于每个块的线程数乘以块数。

每个线程块的线程数 和 每个网格的线程块 都通过<<<...>>>设置。

```
1 // Kernel definition
2 __global__ void MatAdd(float A[N][N], float B[N][N],
3 float C[N][N])
4 {
5     int i = blockIdx.x * blockDim.x + threadIdx.x;
6     int j = blockIdx.y * blockDim.y + threadIdx.y;
7     if (i < N && j < N)
8         C[i][j] = A[i][j] + B[i][j];
9 }
10
11 int main()
12 {
13     ...
14     // Kernel invocation
15     dim3 threadsPerBlock(16, 16);
16     dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
17     MatAdd<<numBlocks, threadsPerBlock>>>(A, B, C);
18     ...
19 }
```

线程块大小为16x16(256个线程), 虽然在这种情况下是任意的, 但却是常见的选择。使用足够的线程块创建网格, 以便像以前一样为每个矩阵元素创建一个线程。为简单起见, 此示例假定每个维度中每个网格的线程数可以被该维度中每个块的线程数整除, 但不一定是这种情况。

线程块需要独立执行: 必须能够以任何顺序独立执行, 不管是并行还是串行。这种独立性要求允许线程块以任意顺序在任意数量的内核上进行调度。

块内的线程可以通过一些共享内存共享数据并通过同步它们的执行来协调内存访问来协作。更确切地说, 可以通过调用__syncthreads()内部函数来指定内核中的同步点; __syncthreads()充当一个屏障, 在该屏障中, 块中的所有线程必须等待才能允许任何线程继续。共享内存提供了使用共享内存的示例。除__syncthreads()之外, 协作组API还提供了一组丰富的线程同步原语。

为了有效合作, 共享内存应该是每个处理器内核附近的低延迟内存(很像L1缓存), 而__syncthreads()应该是轻量级的。

2.3 Memory Hierarchy 内存层次结构

CUDA线程可以在执行期间从多个内存空间访问数据, 如图7所示。每个线程都有私有本地内存。每个线程块都具有对块的所有线程可见的共享内存, 并且具有与块相同的生存期。所有线程都可以访问相同的全局内存。

所有线程都可以访问两个额外的只读内存空间: 常量和纹理内存空间。全局, 常量和纹理内存空间针对不同的内存使用进行了优化(请参阅设备内存访问)。纹理存储器还为某些特定数据格式提供不同的寻址模式以及数据滤波(请参阅纹理和表面存储器)。

全局, 常量和纹理内存空间在同一应用程序的内核启动之间是持久的。

2.5 Compute Capability 计算能力

设备的计算能力由版本号表示, 有时也称为“SM版本”。此版本号标识GPU硬件支持的功能, 并由运行时的应用程序用于确定当前GPU上可用的硬件功能和/或指令。

4.1.3 编程接口

写程序以供设备执行。

它由对C语言的最小扩展集和运行时库组成。

核心语言扩展已在编程模型中引入。它们允许程序员将内核定义为C函数, 并在每次调用函数时使用一些新语法来指定网格和块维度。可以在C语言扩展中找到所有扩展的完整描述。必须使用nvcc编译包含其中某些扩展的任何源文件, 如使用NVCC编译中所述。

运行时在编译工作流程中引入。它提供在主机上执行的C函数, 用于分配和释放设备内存, 在主机内存和设备内存之间传输数据, 管理具有多个设备的系统等。可以在CUDA参考手册中找到运行时的完整描述。

运行时构建在较低级别的C API(CUDA驱动程序API)之上, 该API也可由应用程序访问。驱动程序API通过暴露较低级别的概念(例如CUDA上下文 - 设备的主机进程的模拟)和CUDA模块(设备的动态加载库的模拟)来提供额外的控制级别。大多数应用程序不使用驱动程序API, 因为它们不需要这种额外的控制级别, 并且在使用运行时时, 上下文和模块管理是隐式的, 从而产生更简洁的代码。驱动程序API在Driver API中引入, 并在参考手册中有详细描述。

4.1.3.1 Compilation with NVCC

- 编译流程
- 二进制兼容性
- PTX 兼容性
- 应用兼容性
- C++ 兼容性
- 64位兼容

4.1.3.2 CUDA C Runtime

运行时在cudart库中实现。该库通过cudart.lib或libcudart.a静态链接到应用程序, 或通过cudart.dll或libcudart.so动态链接。需要cudart.dll和/或cudart.so进行动态链接的应用程序通常将它们作为应用程序安装包的一部分包含在内。在链接到CUDA运行时的同一实例的组件之间传递CUDA运行时符号的地址是安全的。

所有入口点都以cuda为前缀。

如异构编程中所述, CUDA编程模型假定由主机和设备组成的系统, 每个系统都有自己独立的内存。设备存储器概述了用于管理设备存储器的运行时功能。

—待续


CUDA 经验分享 C++ GPU



打赏 0



点赞 0



收藏 0



分享

上一篇: gazebo 机器人仿真

关于作者



关注者 2

关注

博客 泡泡 积分 勋章

cartographer 生成子图

阅读量 687 评论数 0

move_bae参数介绍

阅读量 70 评论数 0

cartographer 前端两种匹配

阅读量 932 评论数 0



相关推荐

git提交PR(pull requests)全过程记录

阅读量 1999 评论数 0

Ubuntu18下安装ROS1&2

阅读量 9693 评论数 0

zotero自动从自己的文献库条目中下载pdf文件

阅读量 9122 评论数 0

PID一点都不难！（非原创，出自哪忘了，内容很好，分享给大家）

阅读量 3673 评论数 1

为你推荐

Jetson NANO上安装配置ROS

阅读数 345 评论数 0

一些常见算法总结

阅读数 2414 评论数 0

Jetson Nano 下使用 RealSense 摄像头

阅读数 2310 评论数 0

STM32复习笔记(五)外部中断

阅读数 701 评论数 0

英伟达jetson NANO开箱

阅读数 326 评论数 0

单足机器人技术分析——空间单足机器人

阅读数 1948 评论数 0

评论(0)

您还未登录, 请[登录](#)后发表或查看评论

ROS答疑(八)——进行多传感器融合的思路？(视觉SLAM和激光...

阅读数 4465 评论数 2

如何检索国外的博士论文

阅读数 4943 评论数 0

热门泡泡

30秒:分

失眠, 聊聊自己搞ROS的心得体会吧

ros学习路线

30秒:分

TF_REPEATED_DATA ignoring data错误

各位大佬, 有什么ROS定位算法推荐吗

5秒:分

ros中启动gazebo时报错

5秒:分

想买能用ROS2的开发套件。或者开发板

评论

收藏

分享

不感兴趣