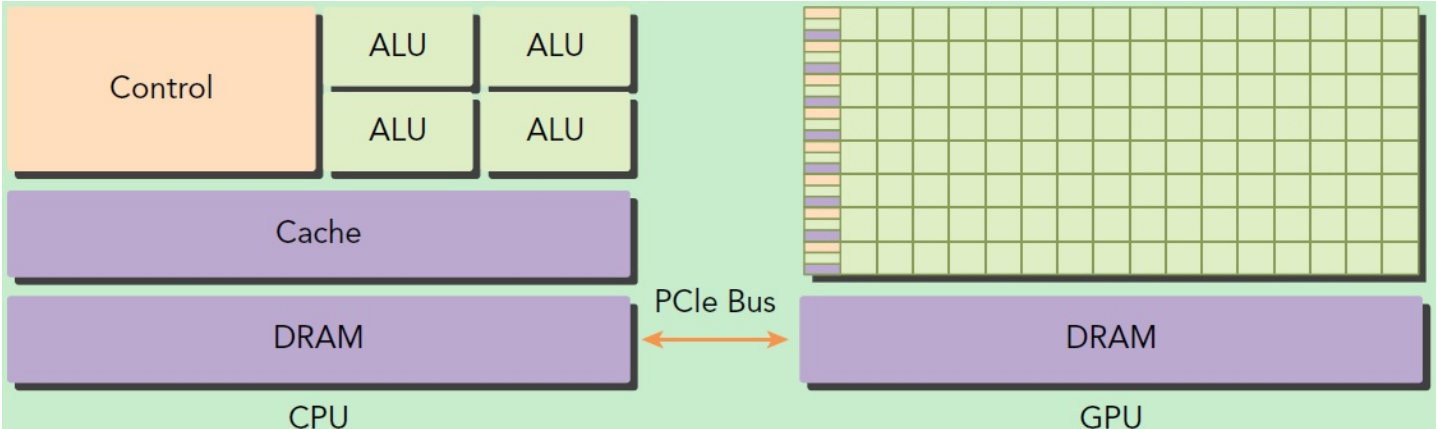


我们检测到你可能使用了 Adblock 或 Adblock Plus，它的部分策略可能会影响到正常功能的使用（如关注）。
你可以设定特殊规则或将知乎加入白名单，以便我们更好地提供服务。（为什么？）



CUDA编程入门极简教程

小小将
为人民日益增长的美好生活需要而读书

1,160 人赞同了该文章

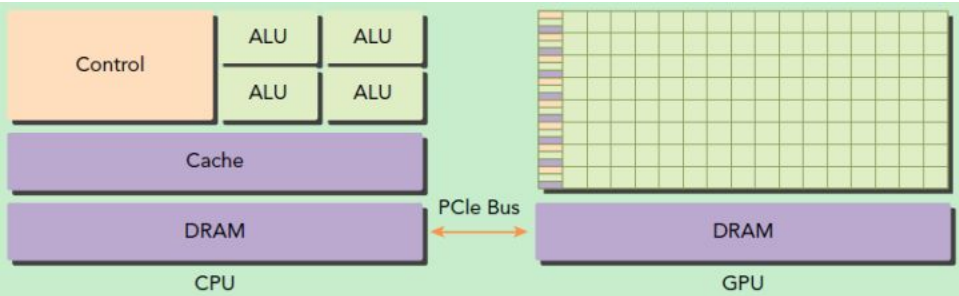
码字不易，欢迎给个赞！

欢迎交流与转载，文章会同步发布在公众号：机器学习算法全栈工程师(Jeemy110)

前言

2006年，NVIDIA公司发布了CUDA，CUDA是建立在NVIDIA的CPUs上的一个通用并行计算平台和编程模型，基于CUDA编程可以利用GPU的并行计算引擎来更加高效地解决比较复杂的计算难题。近年来，GPU最成功的一个应用就是深度学习领域，基于GPU的并行计算已经成为训练深度学习模型的标配。目前，最新的CUDA版本为CUDA 9。

GPU并不是一个独立运行的计算平台，而需要与CPU协同工作，可以看成是CPU的协处理器，因此当我们在说GPU并行计算时，其实是指的基于CPU+GPU的异构计算架构。在异构计算架构中，GPU与CPU通过PCIe总线连接在一起协同工作，CPU所在位置称为为主机端（host），而GPU所在位置称为设备端（device），如下图所示。



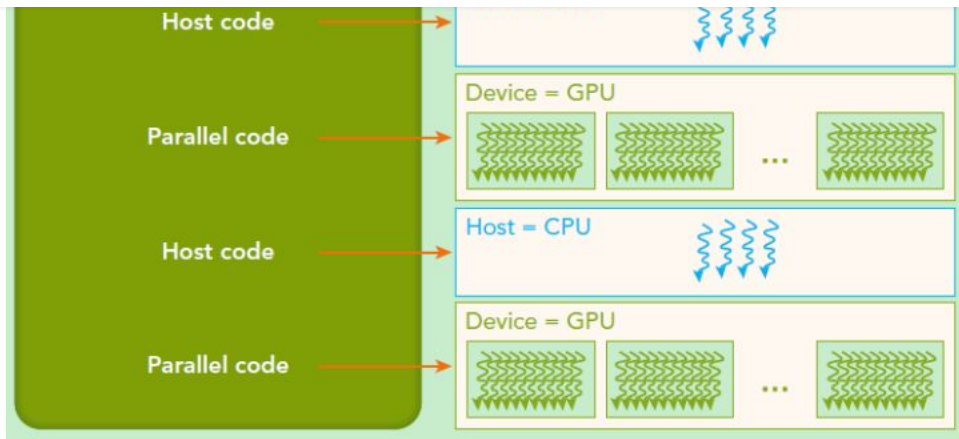
基于CPU+GPU的异构计算. 来源：Preofessional CUDA® C Programming

可以看到GPU包括更多的运算核心，其特别适合数据并行的计算密集型任务，如大型矩阵运算，而CPU的运算核心较少，但是其可以实现复杂的逻辑运算，因此其适合控制密集型任务。另外，CPU上的线程是重量级的，上下文切换开销大，但是GPU由于存在很多核心，其线程是轻量级的。因此，基于CPU+GPU的异构计算平台可以优势互补，CPU负责处理逻辑复杂的串行程序，而GPU重点处理数据密集型的并行计算程序，从而发挥最大功效。

▲
赞同 1.2K
▼
分享

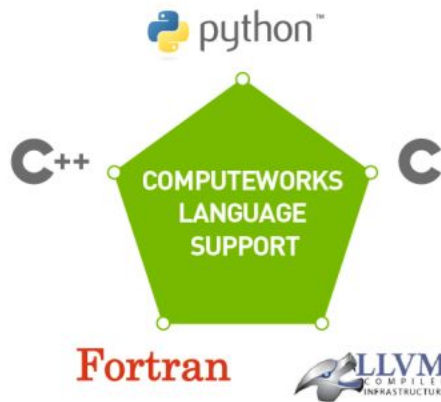
▲ 赞同 1.2K ▼ 50 条评论 分享 收藏 ...





基于CPU+GPU的异构计算应用执行逻辑. 来源: Professional CUDA® C Programming

CUDA是NVIDIA公司所开发的GPU编程模型, 它提供了GPU编程的简易接口, 基于CUDA编程可以构建基于GPU计算的应用程序。CUDA提供了对其它编程语言的支持, 如C/C++, Python, Fortran等语言, 这里我们选择CUDA C/C++接口对CUDA编程进行讲解。开发平台为Windows 10 + VS 2013, Windows系统下的CUDA安装教程可以参考[这里](#)。



CUDA编程模型支持的编程语言

CUDA编程模型基础

在给出CUDA的编程实例之前, 这里先对CUDA编程模型中的一些概念及基础知识做个简单介绍。CUDA编程模型是一个异构模型, 需要CPU和GPU协同工作。在CUDA中, **host**和**device**是两个重要的概念, 我们用host指代CPU及其内存, 而用device指代GPU及其内存。CUDA程序中既包含host程序, 又包含device程序, 它们分别在CPU和GPU上运行。同时, host与device之间可以进行通信, 这样它们之间可以进行数据拷贝。典型的CUDA程序的执行流程如下:

1. 分配host内存, 并进行数据初始化;
2. 分配device内存, 并从host将数据拷贝到device上;
3. 调用CUDA的核函数在device上完成指定的运算;
4. 将device上的运算结果拷贝到host上;
5. 释放device和host上分配的内存。

上面流程中最最重要的一个过程是调用CUDA的核函数来执行并行计算, **kernel**是CUDA中一个重要的概念, kernel是在device上线程中并行执行的函数, 核函数用 `__global__` 符号声明, 在调用时需要用 `<<<grid, block>>>` 来指定kernel要执行的线程数量, 在CUDA中, 每一个线程都要执行核函数, 并且每个线程会分配一个唯一的线程号thread ID, 这个ID值可以通过核函数的内置变量

`threadId`;

赞同 1.2K



分享

赞同 1.2K

50 条评论

分享

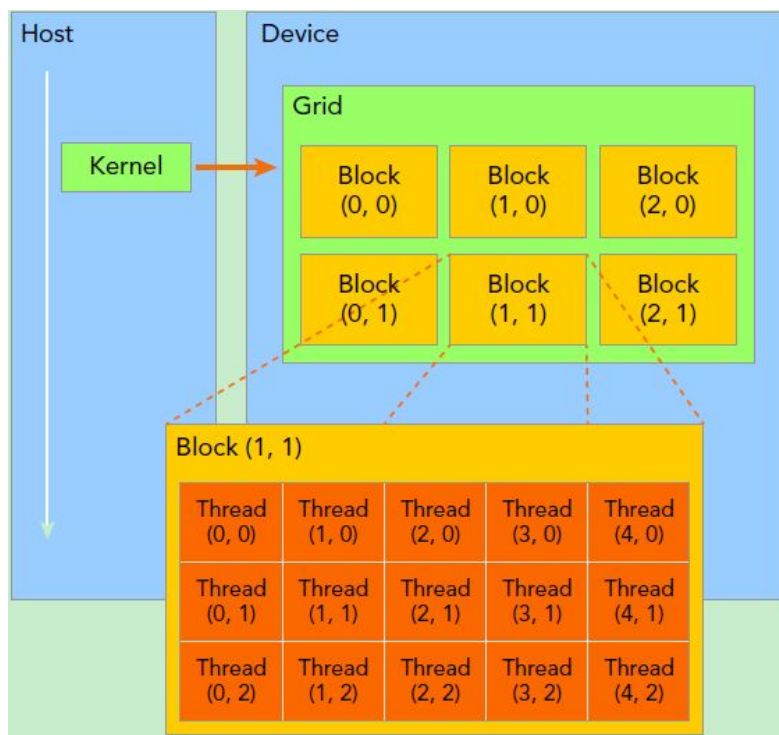
收藏

...

- `__global__`：在device上执行，从host中调用（一些特定的GPU也可以从device上调用），返回类型必须是 `void`，不支持可变参数参数，不能成为类成员函数。注意用 `__global__` 定义的kernel是异步的，这意味着host不会等待kernel执行完就执行下一步。
- `__device__`：在device上执行，单仅可以从device中调用，不可以和 `__global__` 同时用。
- `__host__`：在host上执行，仅可以从host上调用，一般省略不写，不可以和 `__global__` 同时用，但可和 `__device__`，此时函数会在device和host都编译。

要深刻理解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。线程两层组织结构如下图所示，这是一个grid和block均为2-dim的线程组织。grid和block都是定义为 `dim3` 类型的变量，`dim3` 可以看成是包含三个无符号整数（x, y, z）成员的结构体变量，在定义时，缺省值初始化为1。因此grid和block可以灵活地定义为1-dim, 2-dim以及3-dim结构，对于图中结构（主要水平方向为x轴），定义的grid和block如下所示，kernel在调用时也必须通过执行配置 `<<<grid, block>>>` 来指定kernel所使用的线程数及结构。

```
dim3 grid(3, 2);
dim3 block(5, 3);
kernel_fun<<< grid, block >>>(prams...);
```



Kernel上的两层线程组织结构（2-dim）

所以，一个线程需要两个内置的坐标变量（`blockIdx`, `threadIdx`）来唯一标识，它们都是 `dim3` 类型变量，其中`blockIdx`指明线程所在grid中的位置，而`threadIdx`指明线程所在block中的位置，如图中的Thread (1,1)满足：

```
threadIdx.x = 1
threadIdx.y = 1
blockIdx.x = 1
blockIdx.y = 1
```

赞同 1.2K

分享

一个线程块上的线程是放在同一个流式多处理器（SM）上的，但是单个SM的资源有限，这导致线程块中的线程一个线程

赞同 1.2K

50 条评论

分享

收藏

...



的ID值为 $(x + y * D_x + z * D_x * D_y)$ 。另外线程还有内置变量gridDim，用于获得网格块各个维度的大小。

kernel的这种线程组织结构天然适合vector,matrix等运算，如我们将利用上图2-dim结构实现两个矩阵的加法，每个线程负责处理每个位置的两个元素相加，代码如下所示。线程块大小为(16, 16)，然后将N*N大小的矩阵均分为不同的线程块来执行加法运算。

```
// Kernel 定义
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel 线程配置
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    // kernel 调用
    MatAdd<<<numBlocks, threadsPerBlock>>>>(A, B, C);
    ...
}
```

此外这里简单介绍一下CUDA的内存模型，如下图所示。可以看到，每个线程有自己的私有本地内存（Local Memory），而每个线程块有包含共享内存（Shared Memory），可以被线程块中所有线程共享，其生命周期与线程块一致。此外，所有的线程都可以访问全局内存（Global Memory）。还可以访问一些只读内存块：常量内存（Constant Memory）和纹理内存（Texture Memory）。内存结构涉及到程序优化，这里不深入探讨它们。



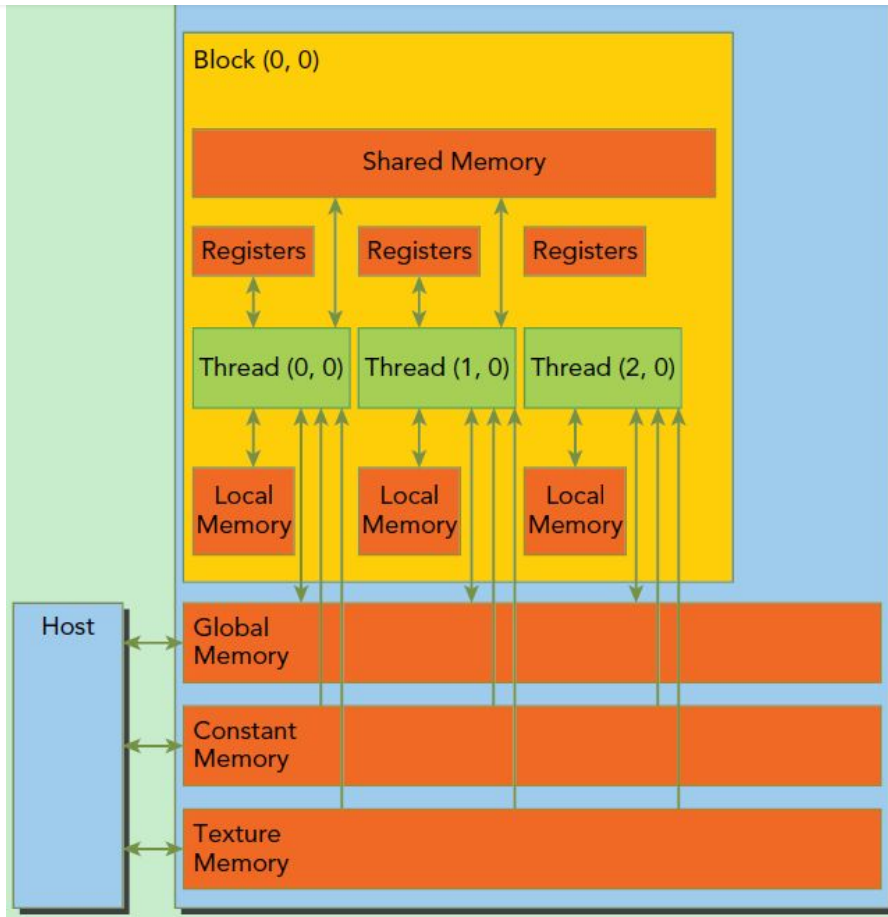
赞同 1.2K



分享

▲ 赞同 1.2K ▼ 50 条评论 分享 收藏 ...





CUDA内存模型

还有重要一点，你需要对GPU的硬件实现有一个基本的认识。上面说到了kernel的线程组织层次，那么一个kernel实际上会启动很多线程，这些线程是逻辑上并行的，但是在物理层却并不一定。这其实和CPU的多线程有类似之处，多线程如果没有多核支持，在物理层也是无法实现并行的。但是好在GPU存在很多CUDA核心，充分利用CUDA核心可以充分发挥GPU的并行计算能力。GPU硬件的一个核心组件是SM，前面已经说过，SM是英文名是 Streaming Multiprocessor，翻译过来就是流式多处理器。SM的核心组件包括CUDA核心，共享内存，寄存器等，SM可以并发地执行数百个线程，并发能力就取决于SM所拥有的资源数。当一个kernel被执行时，它的grid中的线程块被分配到SM上，一个线程块只能在一个SM上被调度。SM一般可以调度多个线程块，这要看SM本身的能力。那么有可能一个kernel的各个线程块被分配多个SM，所以grid只是逻辑层，而SM才是执行的物理层。SM采用的是SIMT (Single-Instruction, Multiple-Thread, 单指令多线程)架构，基本的执行单元是线程束 (warps)，线程束包含32个线程，这些线程同时执行相同的指令，但是每个线程都包含自己的指令地址计数器和寄存器状态，也有自己独立的执行路径。所以尽管线程束中的线程同时从同一程序地址执行，但是可能具有不同的行为，比如遇到了分支结构，一些线程可能进入这个分支，但是另外一些有可能不执行，它们只能死等，因为GPU规定线程束中所有线程在同一周期执行相同的指令，线程束分化会导致性能下降。当线程块被划分到某个SM上时，它将进一步划分为多个线程束，因为这才是SM的基本执行单元，但是一个SM同时并发的线程束数是有限的。这是因为资源限制，SM要为每个线程块分配共享内存，而也要为每个线程束中的线程分配独立的寄存器。所以SM的配置会影响其所支持的线程块和线程束并发数量。总之，就是网格和线程块只是逻辑划分，一个kernel的所有线程其实在物理层是不一定同时并发的。所以kernel的grid和block的配置不同，性能会出现差异，这点是要特别注意的。还有，由于SM的基本执行单元是包含32个线程的线程束，所以block大小一般要设置为32的倍数。

赞同 1.2K



分享

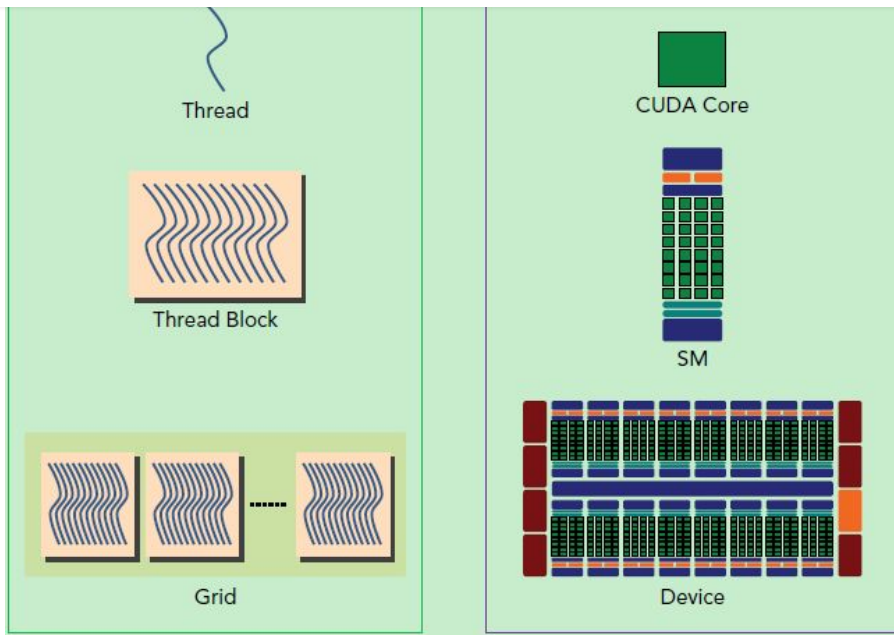
赞同 1.2K

50 条评论

分享

收藏

...



CUDA编程的逻辑层和物理层

在进行CUDA编程前，可以先检查一下自己的GPU的硬件配置，这样才可以有的放矢，可以通过下面的程序获得GPU的配置属性：

```
int dev = 0;
cudaDeviceProp devProp;
CHECK(cudaGetDeviceProperties(&devProp, dev));
std::cout << "使用GPU device " << dev << ": " << devProp.name << std::endl;
std::cout << "SM的数量: " << devProp.multiProcessorCount << std::endl;
std::cout << "每个线程块的共享内存大小: " << devProp.sharedMemPerBlock / 1024.0 << "
std::cout << "每个线程块的最大线程数: " << devProp.maxThreadsPerBlock << std::endl;
std::cout << "每个EM的最大线程数: " << devProp.maxThreadsPerMultiProcessor << std::e
std::cout << "每个EM的最大线程束数: " << devProp.maxThreadsPerMultiProcessor / 32 <<

// 输出如下
使用GPU device 0: GeForce GT 730
SM的数量: 2
每个线程块的共享内存大小: 48 KB
每个线程块的最大线程数: 1024
每个EM的最大线程数: 2048
每个EM的最大线程束数: 64
```

好吧，GT 730显卡确实有点渣，只有2个SM，呜呜.....

向量加法实例

知道了CUDA编程基础，我们就来个简单的实战，利用CUDA编程实现两个向量的加法，在实现之前，先简单介绍一下CUDA编程中内存管理API。首先是在device上分配内存的cudaMalloc函数：

```
cudaError_t cudaMalloc(void** devPtr, size_t size);
```

这个函数和C语言中的malloc类似，但是在device上申请一定字节大小的显存，其中devPtr是指向所分配内存的指针。同时要释放分配的内存使用cudaFree函数，这和C语言中的free函数对应。另外一个重要的函数是负责host和device之间数据通信的cudaMemcpy函数：

赞同 1.2K

分享

cudaErrc

赞同 1.2K

50 条评论

分享

收藏

...



cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost及 cudaMemcpyDeviceToDevice, 如cudaMemcpyHostToDevice将host上数据拷贝到device上。

现在我们来实现一个向量加法的实例, 这里grid和block都设计为1-dim, 首先定义kernel如下:

```
// 两个向量加法kernel, grid和block均为一维
__global__ void add(float* x, float* y, float* z, int n)
{
    // 获取全局索引
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    // 步长
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride)
    {
        z[i] = x[i] + y[i];
    }
}
```

其中stride是整个grid的线程数, 有时候向量的元素数很多, 这时候可以将在每个线程实现多个元素 (元素总数/线程总数) 的加法, 相当于使用了多个grid来处理, 这是一种grid-stride loop方式, 不过下面的例子一个线程只处理一个元素, 所以kernel里面的循环是不执行的。下面我们具体实现向量加法:

```
int main()
{
    int N = 1 << 20;
    int nBytes = N * sizeof(float);
    // 申请host内存
    float *x, *y, *z;
    x = (float*)malloc(nBytes);
    y = (float*)malloc(nBytes);
    z = (float*)malloc(nBytes);

    // 初始化数据
    for (int i = 0; i < N; ++i)
    {
        x[i] = 10.0;
        y[i] = 20.0;
    }

    // 申请device内存
    float *d_x, *d_y, *d_z;
    cudaMalloc((void**)&d_x, nBytes);
    cudaMalloc((void**)&d_y, nBytes);
    cudaMalloc((void**)&d_z, nBytes);

    // 将host数据拷贝到device
    cudaMemcpy((void*)d_x, (void*)x, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy((void*)d_y, (void*)y, nBytes, cudaMemcpyHostToDevice);
    // 定义kernel的执行配置
    dim3 blockSize(256);
    dim3 gridSize((N + blockSize.x - 1) / blockSize.x);
    // 执行kernel
    add << < gridSize, blockSize >> >(d_x, d_y, d_z, N);

    // 将device得到的结果拷贝到host
    cudaMemcpy((void*)z, (void*)d_z, nBytes, cudaMemcpyDeviceToHost);

    // 打印结果
    float
```

赞同 1.2K



分享

赞同 1.2K

50 条评论

分享

★ 收藏

...

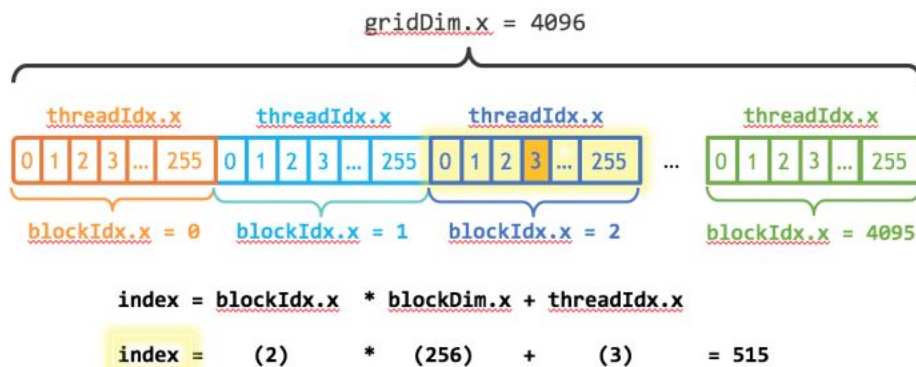


```
std::cout << "最大误差: " << maxError << std::endl;

// 释放device内存
cudaFree(d_x);
cudaFree(d_y);
cudaFree(d_z);
// 释放host内存
free(x);
free(y);
free(z);

return 0;
}
```

这里我们的向量大小为1<20，而block大小为256，那么grid大小是4096，kernel的线程层级结构如下图所示：



kernel的线程层次结构. 来源: <https://devblogs.nvidia.com/even-easier-introduction-cuda/>

使用nvprof工具可以分析kernel运行情况，结果如下所示，可以看到kernel函数费时约1.5ms。

```
nvprof cuda9.exe
==7244== NVPROF is profiling process 7244, command: cuda9.exe
最大误差: 4.31602e+008
==7244== Profiling application: cuda9.exe
==7244== Profiling result:
```

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	67.57%	3.2256ms	2	1.6128ms	1.6017ms	1.6239ms	[CUDA mem
	32.43%	1.5478ms	1	1.5478ms	1.5478ms	1.5478ms	add(float

你调整block的大小，对比不同配置下的kernel运行情况，我这里测试的是当block为128时，kernel费时约1.6ms，而block为512时kernel费时约1.7ms，当block为64时，kernel费时约2.3ms。看来不是block越大越好，而要适当选择。

在上面的实现中，我们需要单独在host和device上进行内存分配，并且要进行数据拷贝，这是很容易出错的。好在CUDA 6.0引入统一内存（Unified Memory）来避免这种麻烦，简单来说就是统一内存使用一个托管内存来共同管理host和device中的内存，并且自动在host和device中进行数据传输。CUDA中使用cudaMallocManaged函数分配托管内存：

```
cudaError_t cudaMallocManaged(void **devPtr, size_t size, unsigned int flag=0);
```

利用统一内存，可以将上面的程序简化如下：

```
int main
{
```

赞同 1.2K

50 条评论

分享

收藏

...



```

// 申请托管内存
float *x, *y, *z;
cudaMallocManaged((void**)&x, nBytes);
cudaMallocManaged((void**)&y, nBytes);
cudaMallocManaged((void**)&z, nBytes);

// 初始化数据
for (int i = 0; i < N; ++i)
{
    x[i] = 10.0;
    y[i] = 20.0;
}

// 定义kernel的执行配置
dim3 blockSize(256);
dim3 gridSize((N + blockSize.x - 1) / blockSize.x);
// 执行kernel
add << < gridSize, blockSize >> >(x, y, z, N);

// 同步device 保证结果能正确访问
cudaDeviceSynchronize();
// 检查执行结果
float maxError = 0.0;
for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(z[i] - 30.0));
std::cout << "最大误差: " << maxError << std::endl;

// 释放内存
cudaFree(x);
cudaFree(y);
cudaFree(z);

return 0;
}

```

相比之前的代码，使用统一内存更简洁了，值得注意的是kernel执行是与host异步的，由于托管内存自动进行数据传输，这里要用cudaDeviceSynchronize()函数保证device和host同步，这样后面才可以正确访问kernel计算的结果。

矩阵乘法实例

最后我们再实现一个稍微复杂一些的例子，就是两个矩阵的乘法，设输入矩阵为 A 和 B ，要得到 $C = A \times B$ 。实现思路是每个线程计算 C 的一个元素值 $C_{i,j}$ ，对于矩阵运算，应该选用grid和block为2-D的。首先定义矩阵的结构体：

```

// 矩阵类型，行优先，M(row, col) = *(M.elements + row * M.width + col)
struct Matrix
{
    int width;
    int height;
    float *elements;
};

```

赞同 1.2K



分享

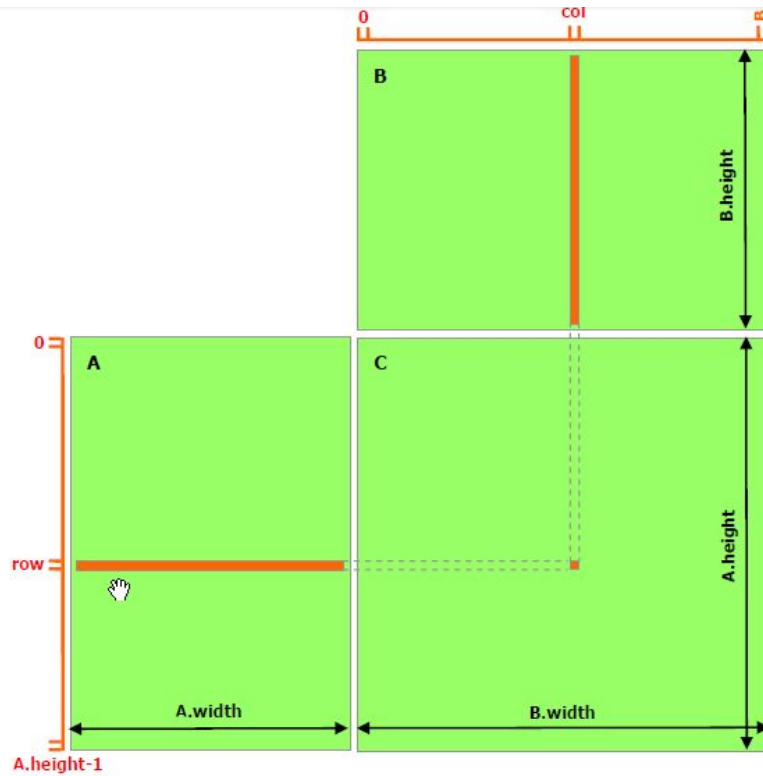
赞同 1.2K

50 条评论

分享

收藏

...



矩阵乘法实现模式

然后实现矩阵乘法的核函数，这里我们定义了两个辅助的 `__device__` 函数分别用于获取矩阵的元素值和为矩阵元素赋值，具体代码如下：

```
// 获取矩阵A的(row, col)元素
__device__ float getElement(Matrix *A, int row, int col)
{
    return A->elements[row * A->width + col];
}

// 为矩阵A的(row, col)元素赋值
__device__ void setElement(Matrix *A, int row, int col, float value)
{
    A->elements[row * A->width + col] = value;
}

// 矩阵相乘kernel, 2-D, 每个线程计算一个元素
__global__ void matMulKernel(Matrix *A, Matrix *B, Matrix *C)
{
    float Cvalue = 0.0;
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    for (int i = 0; i < A->width; ++i)
    {
        Cvalue += getElement(A, row, i) * getElement(B, i, col);
    }
    setElement(C, row, col, Cvalue);
}
```

最后我们采用统一内存编写矩阵相乘的测试实例：

```
int main()
{
    int width = 128, height = 128;
```

赞同 1.2K



分享

赞同 1.2K

50 条评论

分享

收藏

...



```

cudaMallocManaged((void**)&A, sizeof(Matrix));
cudaMallocManaged((void**)&B, sizeof(Matrix));
cudaMallocManaged((void**)&C, sizeof(Matrix));
int nBytes = width * height * sizeof(float);
cudaMallocManaged((void**)&A->elements, nBytes);
cudaMallocManaged((void**)&B->elements, nBytes);
cudaMallocManaged((void**)&C->elements, nBytes);

// 初始化数据
A->height = height;
A->width = width;
B->height = height;
B->width = width;
C->height = height;
C->width = width;
for (int i = 0; i < width * height; ++i)
{
    A->elements[i] = 1.0;
    B->elements[i] = 2.0;
}

// 定义kernel的执行配置
dim3 blockSize(32, 32);
dim3 gridSize((width + blockSize.x - 1) / blockSize.x,
               (height + blockSize.y - 1) / blockSize.y);
// 执行kernel
matMulKernel << < gridSize, blockSize >> >(A, B, C);

// 同步device 保证结果能正确访问
cudaDeviceSynchronize();
// 检查执行结果
float maxError = 0.0;
for (int i = 0; i < width * height; ++i)
    maxError = fmax(maxError, fabs(C->elements[i] - 2 * width));
std::cout << "最大误差: " << maxError << std::endl;

return 0;
}

```

这里矩阵大小为，设计的线程的block大小为(32, 32)，那么grid大小为(32, 32)，最终测试结果如下：

```

nvprof cuda9.exe
==16304== NVPROF is profiling process 16304, command: cuda9.exe
最大误差: 0
==16304== Profiling application: cuda9.exe
==16304== Profiling result:

```

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	100.00%	1.32752s	1	1.32752s	1.32752s	1.32752s	matMulKernel
API calls:	83.11%	1.32762s	1	1.32762s	1.32762s	1.32762s	cudaDeviceSynchronize
	13.99%	223.40ms	6	37.233ms	37.341us	217.66ms	cudaMallocManaged
	2.81%	44.810ms	1	44.810ms	44.810ms	44.810ms	cudaLaunchKernel
	0.08%	1.3300ms	94	14.149us	0ns	884.64us	cudaDeviceGet
	0.01%	199.03us	1	199.03us	199.03us	199.03us	cudaDeviceGet
	0.00%	10.009us	1	10.009us	10.009us	10.009us	cudaDeviceGet
	0.00%	6.5440us	1	6.5440us	6.5440us	6.5440us	cudaConfigureP
	0.00%	3.0800us	3	1.0260us	385ns	1.5400us	cudaSetupDevice
	0.00%	2.6940us	3	898ns	385ns	1.5390us	cudaDeviceGet
	0.00%	1.9250us	2	962ns	385ns	1.5400us	cudaDeviceGet

赞同 1.2K



分享

==16304==

赞同 1.2K

50 条评论

分享

收藏

...

2051	4.0000KB	4.0000KB	4.0000KB	8.011719MB	21.20721ms	Host To Device
270	45.570KB	4.0000KB	1.0000MB	12.01563MB	7.032508ms	Device To Host



当然，这不是最高效的实现，后面可以继续优化...

小结

最后只有一句话：CUDA入门容易，但是深入难！希望不是从入门到放弃...

参考资料

- 1. John Cheng, Max Grossman, Ty McKercher. Professional CUDA C Programming, 2014.
- 2. CUDA docs.
- 3. An Even Easier Introduction to CUDA.
- 4. Unified Memory in CUDA 6.
- 5. Maximizing Unified Memory Performance in CUDA.

码字不易，欢迎给个赞！

欢迎交流与转载，文章会同步发布在公众号：机器学习算法全栈工程师(Jeemy110)

编辑于 2018-04-07

图形处理器 (GPU) CUDA 并行计算

文章被以下专栏收录

机器学习算法工程师

学习AI算法，请关注微信公众号：机器学习算法工程师.....

进入专栏

推荐阅读

揭秘GPU

摘要： GPU在机器学习中至关重要，但很少有人能解释清楚，本文对此进行了一番研究。如果你从事机器学习，或者碰巧是NVIDIA的投资者，你一定听说过GPU对这个领域至关重要。但对我来说，GPU...

阿里云云栖... 发表于我是程序员

Tencent 腾讯

腾讯GPU并行计算高级工程师 (25K ~ 60K/月)

姜军师

CUDA 基础知识博客整理

谭升 发表于CUDA从...

CUDA

我的CUDA学习之旅

Russell

赞同 1.2K

分享

50 条评论 切换为时间排序

▲ 赞同 1.2K ▼ 50 条评论 分享 ★ 收藏 ...

https://zhuanlan.zhihu.com/p/34587739

12/15

精选评论 (11)

 愤怒的胸毛

讲的很细，不错的文章

👍 3 💬 查看回复

 霍华德

这样的文章越多越好

👍 10 💬 查看回复

 xinnjie

block, grid, thread, warp, SM的关系讲得很好了，受教

👍 1 💬 查看回复

 扶余城里小老二

10多年前，为什么没有这篇文章？

👍 1 💬 查看回复

 知乎用户

入乎一年多，首次推送到cuda相关文章。。。

👍 1

 dongdong

666

👍 1

 Dionysus

佩服佩服，最近刚准备了解一下CUDA，刚好有这么一个教程， 十分感谢！

👍 1 💬 查看回复

 知乎用户

写的不错

👍 1

 西南皮皮王

写的太好了，非常感谢，第一次遇到这么棒的cuda相关的文章

👍 1

 lhrbu

模型讲解的很清楚，一下就能明白cuda的运行原理了

👍 1

 许煊汶

第一次看到对新手这么友好的文章

👍 1

评论 (50)

 愤怒的胸毛

讲的很细，不错的文章

👍 3

 小小将 (作者) 回复 愤怒的胸毛

谢谢支持

👍 1 1 年前

 霍:

赞同 1.2K

50 条评论

分享

收藏

...

▲

赞同 1.2K

🔗

分享





10

小小将 (作者) 回复 霍华德
😊😊好的
赞

1 年前

流川枫
感谢分享。加油
1

1 年前

孙明
很赞
1

1 年前

yx huang
SM那块的概念NV自己也讲得啰哩啰嗦，名字没取好。
1

1 年前

小小将 (作者) 回复 yx huang
这块我也只是理解个大概，GPU编程涉及的不只是软件，也有硬件
赞

1 年前

xinnjie
block, grid, thread, warp, SM的关系讲得很好了，受教
1

1 年前

小小将 (作者) 回复 xinnjie
谢谢！
赞

1 年前

冰力寒霜
赞一个，网上写CUDA的博客比较少见
赞

1 年前

小小将 (作者) 回复 冰力寒霜
😊😊😊
赞

1 年前

知乎用户 回复 冰力寒霜
百度过吗？一搜一大堆。。。
赞

1 年前

展开其他 3 条回复

扶余城里小老二
10多年前，为什么没有这篇文章？
1

1 年前

小小将 (作者) 回复 扶余城里小老二
什么意思😊😊
赞

1 年前

Find Hao 回复 扶余城里小老二
有的是这样的文章，只不过十年前没有知乎

12 个月前

赞同 1.2K 50 条评论 分享 收藏 ...

知乎



首发于
机器学习算法工程师

关注专栏

👍 1

1 2 3 下一页

