基于 CUDA 的矩阵乘法优化

贠龙飞　软件 91　2018012381　yuanlf18@mails.tsinghua.edu.cn

摘要

随着新科技和处理方法的普及，高性能计算（HPC）领域也在不断变化，而 HPC 的定义也随之产生了相应的变化。一般来说，它涉及多个处理器或计算机的使用，以高吞吐量和高效率来完成一个复杂的任务。HPC 不仅可以认为是一个计算架构，还可以认为是包括硬件系统、软件工具、编程平台及并行编程范例的一组元素列表。在过去的十几年中，高性能计算取得了极大的发展，尤其是 GPU-CPU 异构架构的出现，直接导致了在并行程序设计中一个基本的范例转变。本文首先简要介绍了作者学习 CUDA 的整体的认识和理解，然后编写了使用 CUDA 的矩阵乘法并对其进行增量优化。作者分析了各矩阵乘法方法的优势并用代码实现，最后对比了各矩阵乘法方法的运行时间，最好的实现和 CUDA 库函数的 cublas 相差不大。

1. CUDA 简介

CUDA **硬件基础**:GPU.　GPU 最初是专门用来处理并行图形计算问题的。随着时间的推移，GPU 的功能越来越强大，在执行大规模并行计算中具有优越的性能和极高的效率。GPU 不是一个独立运行的平台而是作为 CPU 的协处理器，通过 PCIe 总线与基于 CPU 的主机相连进行操作。因此 CPU 称为主机端，GPU 也称作设备端。

从右图可以看出 GPU 包含更多的计算单元 (ALU)，特别适合数据并行的计算密集型任务，如大型矩阵的运算；而 CPU 的运算核心比较少，但是可以实现复杂的逻辑运算，因此适合控制密集型任务。

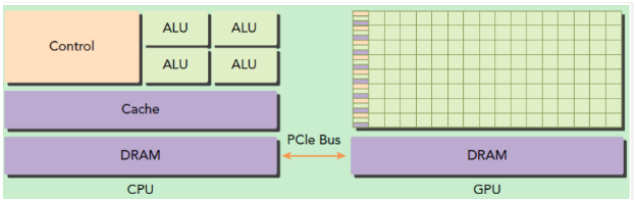

图 1. 异构架构

CUDA **编程模型**.　host 指代 CPU 及其内存，device 指代 GPU 及其内存。CUDA 程序中既包含了 host 程序又包含了 device 程序，它们分别在 CPU 和 GPU 上运行。host 和 device 之间可以进行数据拷贝。典型的 CUDA 程序的执行流程如下：

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

内核 (kernel) 是 CUDA 编程模型中的一个重要组成部分，其代码在 GPU 上运行。CUDA 编程模型是异步执行的：内核一旦被启动，管理权立刻返回给主机，继续执行 CPU 的主机代码。

CUDA **线程管理**.　由一个内核启动所产生的所有线程统称为一个网格 (grid)，同一网格中的所有线程共享相同的全局内存。一个网格由多个线程块 (block) 构成，一个线程块包含一组线程 (thread)，同一线程块内的线程协作可以通过同步和共享内存实现。线程可以通过 blockIdx(线程块在网格中的索引）和 threadIdx(线程块内的线程索引）来访问。它们都是

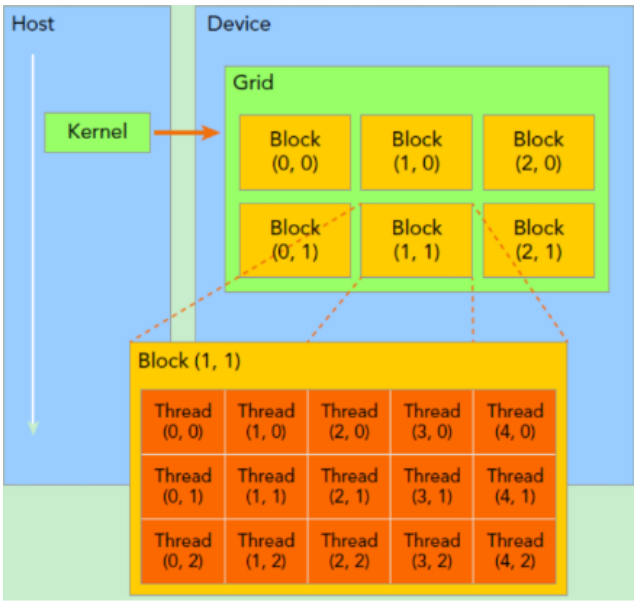

图 2. CUDA 的两层线程抽象层次

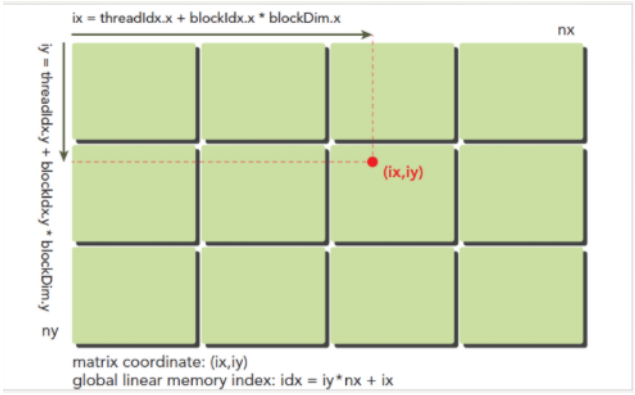

图 3. 利用块和线程建立矩阵索引

dim3 类型的变量即存在 x, y, z 三个字段。根据线程的坐标，可以将运算分配给不同的线程。

CUDA **执行模型**. 我的理解是线程管理是较高层次的、面向程序员的抽象，执行模型则更为底层，属于硬件层级的架构。GPU 的架构是围绕流式多处理器 (SM) 的可扩展阵列搭建的。Fermi SM 的关键组件包括 CUDA 核心、共享内存、寄存器文件、加载/存储单元、线程束调度器等。

GPU 中有多个 SM，每个 SM 都能够支持数百个线程并发执行。当启动一个 grid 时，它的线程块根据 SM 资源的可用性分布到一个可用的 SM 上，多个线程块可能被分配到同一个 SM 上。同一线程中

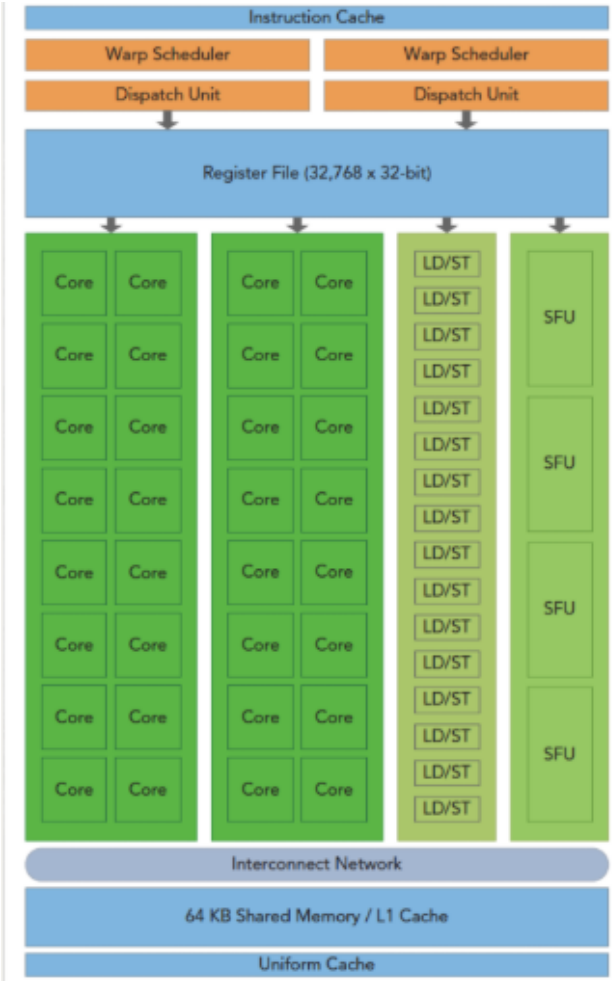

图 4. Fermi SM 的关键组件

的指令通过流水线化实现指令级的并行。

CUDA 采用单指令多线程 (SIMT) 架构来管理和执行线程，每 32 个线程为一组，称作线程束 (wrap). 每个 SM 将分配给它们的线程块划分到线程束中，在可用的硬件资源上调度执行。每个线程都有自己的指令地址计数器 (PC) 和寄存器状态，线程束中的所有线程同时执行相同过的指令。

CUDA **内存模型**. 现代计算机不断使用改进的低延迟、低容量的内存结构来优化性能。这种内存层次结构只在支持局部性原则的情况下有效：

- 时间局部性：如果一个数据被引用，则该数据在较短的时间周期内很可能再次被引用
- 空间局部性：如果某个位置的内存被引用，则该

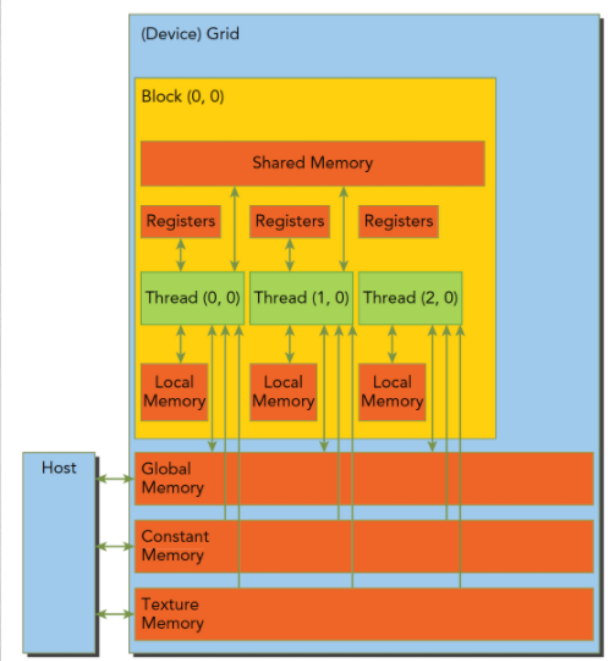

图 5. CUDA 内存模型

位置附近的内存也可能被引用。

一个内存层次结构由具有不同延迟、带宽和容量的多级内存组成。通常，随着处理器的内存延迟的增加，内存的容量也随之增加。

CUDA 的存储从编程的角度可以分为两类:

- 可编程的：需要显示控制数据放在可编程内存中
- 不可编程的：程序员不能决定数据的存放位置，程序将自动生成存放位置。

CPU 的一级缓存和二级缓存都是不可编程的存储器。CUDA 内存模型提出了多种可编程内存的类型:

- 寄存器
- 共享内存
- 本地内存
- 常量内存
- 纹理内存
- 全局内存

因为后面的优化会用到共享内存，所以这里着重介绍一下共享内存。共享内存是**片上内存** (容量较小因而延迟相对较低)，与本地内存或者全局内存相

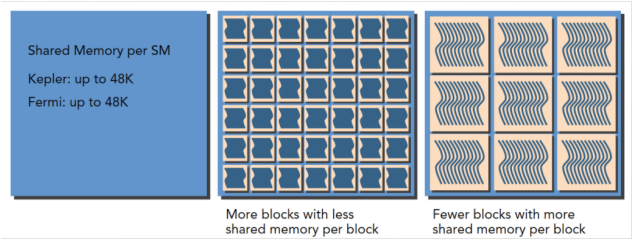

图 6. 资源可用性通常会限制 SM 中常驻线程块的数量

比，它具有更高的带宽和更低的延迟，每一个 SM 都有一定数量的由线程块分配的共享内存。共享内存的生命周期伴随着整个线程块，当一个线程块执行结束后，分配的共享内存被释放并重新分配给其它线程块。共享内存是线程之间互相通信的基本方式。一个块内的线程通过使用共享内存中的数据可以相互合作。

每个 SM 都有 32 位的寄存器组，存储在寄存器文件中，可以在线程中进行分配。同时固定数量的共享内存可以在线程块之间进行分配。线程束、线程块的数量和寄存器、共享内存的数量之间存在竞争关系。(见 图6)

程序性能指标. 不仅要关注运行时间，我们需要从多个维度来评价程序是否得到了优化，下面是一些比较常用的指标:

- 延迟：一个操作从开始到完成所需要的时间，常用微妙表示。
- 带宽：单位时间内可以处理的数据量，通常表示位 MB/s 或 GB/s。
- 吞吐量：单位时间内可以处理的运算数量，通常表示为 gflops(每秒十亿次的浮点运算数量。

2. 基于 CPU 的矩阵乘法

我使用了两种方法在 CPU 上运行矩阵乘法。第一种方法是蛮力算法，利用公式 $C_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$，三重循环计算两个矩阵的乘积,时间复杂度为 $O(n^3)$.

第二种方法是 Strassen 方法，由分治算法优化而来。

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
1  n = A.rows
2  let C be a new n×n matrix
3  if n==1
4      c11=a11 • b11
5  else partition A, B, and C as in equations (4.9)
6      C11=SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
         + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
7      C12=SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
         + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
8      C21=SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
         + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
9      C22=SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
         + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10 return C
```

图 7. 矩阵乘法的 CPU 分治算法

分治算法. 如果将 A、B、C 简单分解为 4 个 $n/2 \times n/2$ 的子矩阵:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

矩阵乘法可以改写为:

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

等价于

$$\begin{aligned} c_{11} &= a_{11} \times b_{11} + a_{12} \times b_{21} \\ c_{12} &= a_{11} \times b_{12} + a_{12} \times b_{22} \\ c_{21} &= a_{21} \times b_{11} + a_{22} \times b_{21} \\ c_{22} &= a_{21} \times b_{12} + a_{22} \times b_{22} \end{aligned}$$

其中复制子矩阵和矩阵加法的总时间复杂度为 $O(n^2)$, 所以分治算法的总时间复杂度为:

$$T(n) = 8T(n/2) + \Theta(n^2) \tag{1}$$

由 master theory, $n^{log_b a} = n^{log_2 8} = n^3$, $O(n^{3-\epsilon} > O(n^2)$, 所以算法的时间复杂度为 $O(n^3)$. 分治算法没有优化矩阵乘法的时间复杂度。

Strassen **算法**. Strassen 算法的核心思想是减少递归树的分支，只进行了 7 次 $n/2 \times n/2$ 的矩阵乘法, 额外代价是产生额外几次 $n/2 \times n/2$ 的矩阵加法, 这部分被递归式中的 $\Theta(n^2)$ 吸收。

算法步骤如下:

1. 将矩阵 A、B 和 C 分解成 $n/2 \times n/2$ 的子矩阵.
2. 创建十个 $n/2 \times n/2$ 的矩阵 $S_1, ..., S_{10}$, 每个矩阵保存步骤一中子矩阵的和或差的运算结果。
3. 用步骤一、二的结果递归地计算 7 个矩阵乘积 $P_1, ..., P_7$. 每个矩阵的大小也是 $n/2 \times n/2$ 的。
4. 通过矩阵的加减运算，计算出结果子矩阵 $c_{11}, c_{12}, c_{21}, c_{22}$.

第二步中创建的矩阵如下:

$$\begin{aligned} S_1 &= b_{12} - b_{22} \\ S_2 &= a_{11} + a_{12} \\ S_3 &= a_{21} + a_{22} \\ S_4 &= b_{21} - b_{11} \\ S_5 &= a_{11} + a_{22} \\ S_6 &= b_{11} + b_{22} \\ S_7 &= a_{12} - a_{22} \\ S_8 &= b_{21} + b_{22} \\ S_9 &= a_{11} - a_{21} \\ S_{10} &= b_{11} + b_{12} \end{aligned}$$

步骤三中递归的 7 次 $n/2 \times n/2$ 矩阵乘法如下:

$$\begin{aligned} P_1 &= a_{11} \times S_1 \\ P_2 &= S_2 \times b_{22} \\ P_3 &= S_3 \times b_{11} \\ P_4 &= a_2 2 \times S_4 \\ P_5 &= S_5 \times S_6 \\ P_6 &= S_7 \times S_8 \\ P_7 &= S_9 \times S_1 0 \end{aligned}$$

n	2	4	8	16	32	64	128	256	512	1024	2048
Brute-Force	0	0	0	0	0	2	13	115	877	10596	132782
Strassen	0	0	0	0	0	3	15	147	1088	8019	67475

表 1. 蛮力算法和 Strassen 算法的时间比较

步骤四对步骤三计算的 P 矩阵进行加减操作:

$$\begin{aligned} c_{11} &= P_5 + P_4 - P_2 + P_6 \\ c_{12} &= P_1 + P_2 \\ c_{21} &= P_3 + P_4 \\ c_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

Strassen 算法的递归式为:

$$T(n) = 7T(n/2) + \Theta(n^2) \tag{2}$$

由 master theory, $f(n) = \Theta(n^2) = O(n^{log_b a - \epsilon} + \Theta(n^2) \approx O(n^{2.81-\epsilon})$, 所以算法的时间复杂度约为 $\Theta(n^{2.81})$.

运行时间比较. 从上面时间复杂度的分析我们可以看出，这两种方法的时间复杂度差别并不是特别大，而且 strassen 算法涉及到很多矩阵的拷贝、加减运算等操作，所以隐藏了很大的常数。这一部分开销在 n 比较小的时候甚至可能会超过降低时间复杂度本身带来的优化。

所以我们可以推测: 在一定范围内, 蛮力算法会略微优于 strassen 算法。但是随着 n 增大，strassen 算法性能会超过蛮力算法。

这张表格是两种运行方法的运行时间 (ms) 随着 n 增长的准确数值。从表中我们可以看出 Strassen 算法一开始的确慢于蛮力算法，但是随着 n 增大最终超过了蛮力算法。

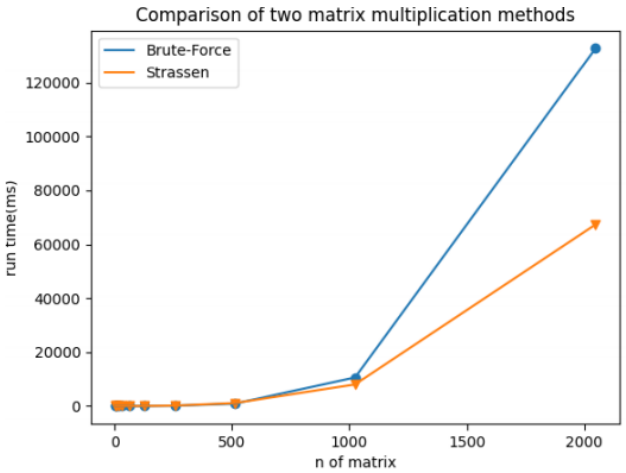

图 8. 蛮力算法和 Strassen 算法的时间比较

这张图更加直观地反应了二者的关系，虽然这里 Strassen 使用的递归方法只适用于 n 是 2 的幂的情形，但是我们可以大致推断出 n 在五百到六百之间会超过第一种方法，验证了我们之前的假设。

3. 基于 GPU 的矩阵乘法

3.1. Benchmark

为了找到一个性能比较的基准, 我将实现的矩阵乘法和标准库 CUBLAS 标准库中的 cublasSgemm 函数进行比较。追根溯源，CUBLAS 来自于线性代数子程序库 BLAS，最初使用 FORTRAN 语句编写的，所以非常重要的一点是矩阵的运算都是列主序的.

cublasSgemm 执行如下的矩阵乘法操作:

$$C = \alpha op(A) op(B) + \beta C \tag{3}$$

- A, B 是两个列主序的矩阵。
- α 和 β 是两个标量。
- op(A) 可以表示 A、A^T 和 A^H, op(B) 类似。

当取 $\alpha = 1, \beta = 0$ 时，由式3可以计算 A 和 B 的乘积。

Param.	Memory	In/Out	Meaning
handle		input	handle to the cuBLAS library context.
transa		input	operation op(A)
transb		input	operation op(B)
m		input	number of rows of op(A) and op(C)
n		input	number of columns of op(B) and op(C)
k		input	number of columns of op(A)
alpha	host/device	input	scalar used for multiplication
A	device	input	
lda		input	leading dimension of A
beta	host/device	input	scalar used for multiplication

表 2. cublasSgemm 各形式参数的含义

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float           *alpha,
                           const float           *A, int lda,
                           const float           *B, int ldb,
                           const float           *beta,
                           float           *C, int ldc)
```

图 9. cublasSgemm 函数定义

我们的目标就是不断改进 GPU 的矩阵乘法，最终使运行时间接近 cublasSgemm。

3.2. GPU 矩阵乘法朴素实现

GPU 计算矩阵乘法的朴素实现使用一个线程计算结果矩阵 C 中的一个元素。每个线程加载矩阵 A 的一行和矩阵 B 的一列，做内积后把结果存储到矩阵 C 的对应位置。

$$row = threadIdx.x + blockDim.x \times blockIdx.x \quad (4)$$

$$col = threadIdx.y + blockDim.y \times blockIdx.y \quad (5)$$

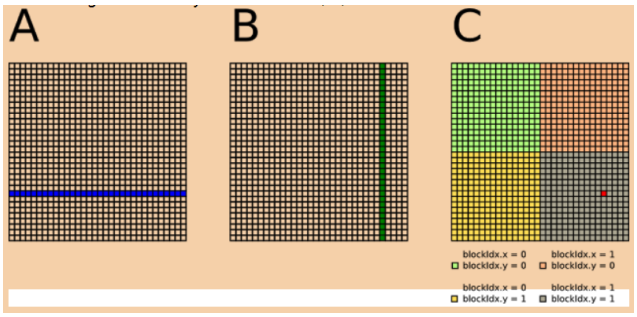

图 10. GPU 矩阵乘法的朴素实现。

我们可以看出，每计算 C 中的一个元素，都需要访问一次全局内存。GPU 的算术操作需要 10 ~ 20 个时钟周期，全局访存一次大约需要 400 ~ 800 个时钟周期，远远超过计算的时间开销成为程序性能的瓶颈，如何提高程序的带宽是我们接下来需要考虑的问题。

3.3. 通过 Tiling 提高计算/访存比

把矩阵 C 划分成不同的 tile，一个线程块计算矩阵 C 的一个 tile，线程块中的每一个线程计算 tile 中的一个元素。

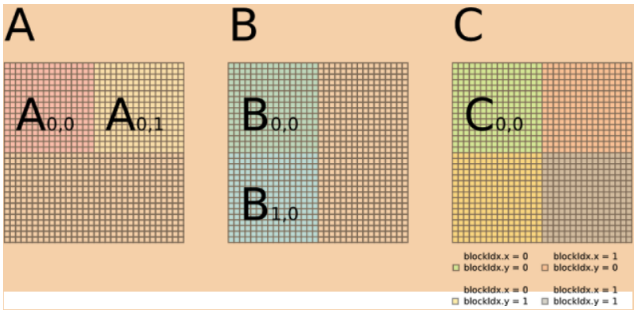

图 11. 使用 Tiling 划分矩阵 C

如上图所示，我们把 32×32 的矩阵 C 划分成四个 16×16 的 tile. 为了计算 C，需要四个有 16×16 个线程的线程块。

核函数使用多次迭代计算 C。在每一次迭代中线程块从全局内存中加载矩阵 A 和矩阵 B 的各一个 tile 到共享内存中，每个线程把计算结果累加到

一个寄存器 (变量) 中。在所有的迭代完成后，该寄存器写入 C 的对应位置。

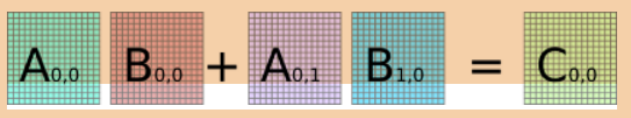

图 12. 迭代计算 $c_{0,0}$

以上图为例，线程块通过两次迭代计算 C 的一个 tile $c_{0,0}$:

$$c_{0,0} = a_{0,0} \times b_{0,0} + a_{0,1} \times b_{1,0} \tag{6}$$

在第一次迭代中，线程块加载 tile $a_{0,0}$ 和 $b_{0,0}$ 到共享内存中。每个线程计算 c 中对应元素的内积并存储在寄存器中，以便在下次迭代中继续累加。

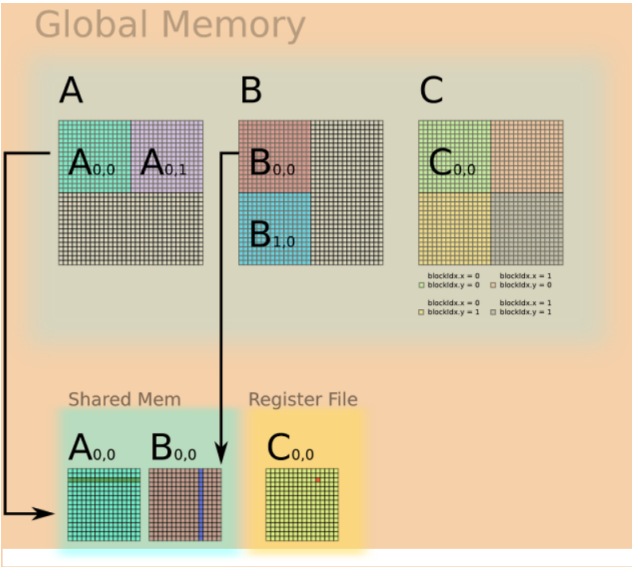

图 13. 第一次迭代过程

在第二次迭代中，线程块加载 tile $a_{0,1}$ 和 $b_{1,0}$ 到共享内存中，每个线程计算 c 中对应元素的内积并累加到之前的寄存器中，如果这是最后一次迭代，寄存器中的数值将会写回全局内存中矩阵 C 的对应位置。

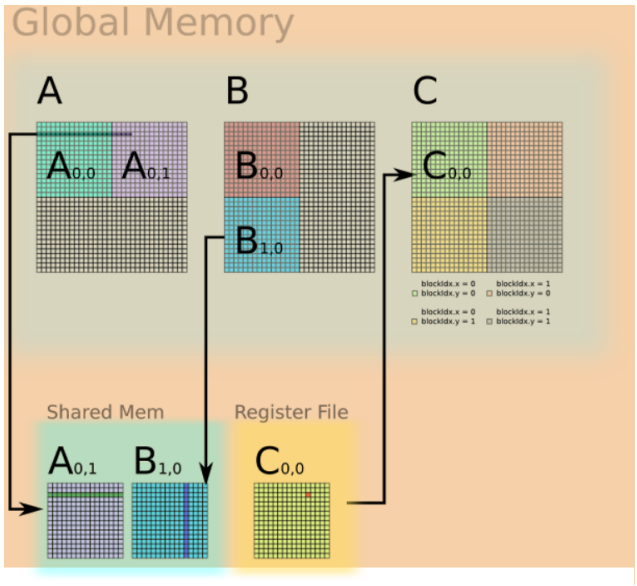

图 14. 第二次迭代过程

使用 tile 使得访问全局内存的次数降低为原来的 $\frac{1}{k}$, 其中 k 是一个 tile 的大小。那我们自然会想到是不是 tile 的大小越大越好呢？答案是否定的。首先一个线程块共享内存的大小有限，一般为 64KB，其次共享内存和线程块数、线程数之间存在对硬件资源的竞争关系，我们可以通过实验确定最佳的 tile 大小。

3.4. 利用矩阵转置实现矩阵 B 的 Coalescing

全局内存带宽. GPU 的全局内存是用 DRAMs 实现的，DRAM 利用电容内存储电量的多少来表示一个比特是 1 还是 0。因为涉及电容的充放电，所以 DRAM 读取数据很慢。现代的 DRAM 芯片读取数据需要约十纳秒，相对于其它计算设备的亚纳秒的时钟周期较慢，所以 DRAMs 通常会使用并行化来提高数据访问速度，即内存访问吞吐量。设备每次访问 DRAMs 中的一个位置，包含目标位置在内的连续位置都会被访问到。如果我们编写程序时能够注意连续访问，就可以以更高的速率获取数据了。

内存合并. 线程束中的线程在一个给定的时刻执行相同的指令，当线程束中所有线程访问内存的位置连续时，将实现最佳的内存访问模式。当线程束中的所有线程执行访存时，硬件会检测它们访问的全

局内存位置是否连续，如果连续，硬件将这些访问合并为对连续位置的访问。

以矩阵乘法为例，假设有一个 3×4 的矩阵如下:

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & a & b \end{bmatrix}$$

假设我们需要访问每一个元素一次且共有四个线程，可以有两种访问方式:

- thread 0: 0, 1, 2
- thread 1: 3, 4, 5
- thread 2: 6, 7, 8
- thread 3: 9, a, b

或者

- thread 0: 0, 4, 8
- thread 1: 1, 5, 9
- thread 2: 2, 6, a
- thread 3: 3, 7, b

需要注意 Coalescing 发生在不同线程之间，而不是线程内部的不同迭代之间，第二种方式第一次迭代各线程访问的数据为 0，1，2，3 是内存连续的，可以合并。

3.4.1 避免共享内存冲突

共享存储器被划分为多个大小相等的存储器模块，称为 bank，可以被同时访问。对不同 bank 的地址读取和写入的操作可以同时进行，从而可以获得较高的带宽。当一个线程束中的不同线程访问一个 bank 中不同的字的地址时就会发生 bank 冲突。最好的情况是当前线程束中的每个线程都访问一个不冲突的共享内存。

3.4.2 计算优化

现在制约核函数性能的主要因素是计算，我们需要增加计算在所有的指令中的比例。流式多处理器目前的架构只允许来自共享内存的一个源操作数。但是我们计算内积的时候需要两个操作数，一种解决方法是把一个矩阵存储到寄存器中，但是寄存器文件中的数据不能被不同的线程共享。

一个更好的解决方案是计算外积而不是内积。在这种情况下，矩阵 A 存储在共享内存中，而矩阵 B 和 C 存储在寄存器中。外积不需要矩阵 B 和矩阵 C 的共享，因此，每个线程在寄存器中只存储 B 的一个元素和 C 的 tile 的一列。外积的“计算/内存比”与内积相同。

下面是一个由 $a_{0,0}$ 和 $b_{0,0}$ 计算 $c_{0,0}$ 的示例。$a_{0,0}$ 的大小是 16×16, $b_{0,0}$ 的大小是 16×64, $c_{0,0}$ 的大小是 16×64.

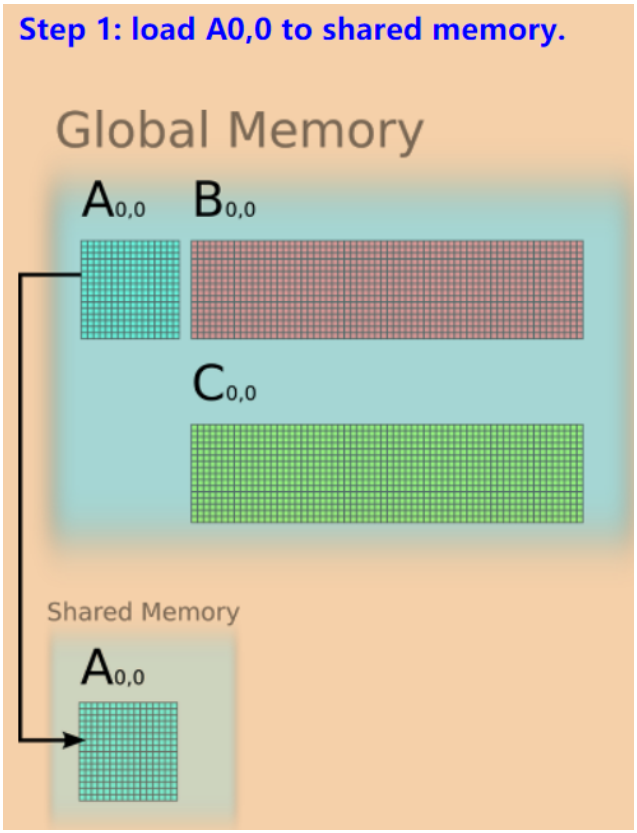

图 15. 将 $a_{0,0}$ 加载到共享内存

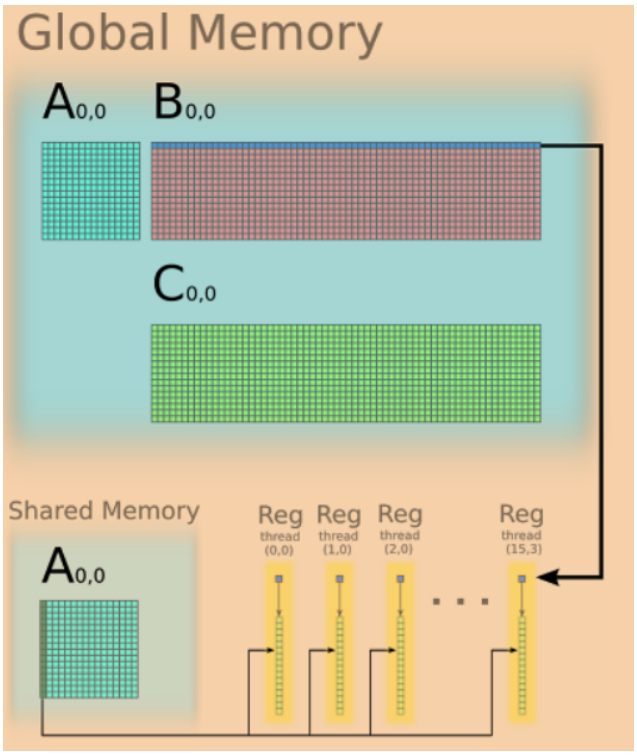

图 16. 通过 16 次迭代更新 $c_{0,0}$

- 第一步首先将 $a_{0,0}$ 加载到共享内存。
- 第二步通过 16 次迭代更新 $c_{0,0}$。每个线程存储 $b_{0,0}$ 的一个元素和 $c_{0,0}$ 到寄存器中。
  第一次迭代计算 $a_{0,0}$ 的第一列和 $b_{0,0}$ 的第一行的外积并更新 $c_{0,0}$。
  第二次迭代计算 $a_{0,0}$ 的第二列和 $b_{0,0}$ 的第二行的外积并更新 $c_{0,0}$。后面的迭代一次类推。实际上计算优化就是通过把少量 B、C 的元素放到每个线程的寄存器中，并改变了矩阵乘法的计算顺序。
- 最后每个线程把计算好的一列写回 C。

3.4.3 数据预取

数据预取就是当进行一次迭代时把下一次迭代要用到的数据放到共享内存中。prefetching 利用了 CUDA 的异步特性。回顾流水线包括取指、译码、执行、访存、写回五个阶段，如果当前访存的读入的数据不会被接下来时钟周期重叠的指令用到，就不会堵塞程序的执行过程。

在 (带有预取的核函数) 的循环中，设备首先为下一次迭代进行内存加载操作，同时并行地进行加法操作。加法的时间实际上与内存访问时间重叠，但是增加的寄存器使用可能会降低 SM 上的活动线程的数量。

4. 性能分析

4.1. 运行环境

1. 服务器：腾讯云 GPU 计算型 GN6S
2. 实例配置：4 核 20GB 1Mbps

程序运行方法 make + ./main 1024

- 通过 make 编译链接，make clean 删除目标文件.
- 通过带形参的 main 函数传入矩阵的大小。为了简化问题，假设两个输入矩阵 A，B 和 C 的矩阵大小一样。
- 因为代码实现中涉及矩阵的分块，所以矩阵大小应当是 32 的倍数。

4.2. 程序实现思路

本次大作业没有采用类进行封装，因为我一开始尝试用面向对象的思想写 cuda 程序时遇到一些难以解决的问题。在不用类写后问题就解决了，所以后面都没有用类进行封装。

在 util.cuh 文件中定义了如下函数:

- cpuSecond(): 计算 CPU 的时间，这个函数也用来计算各计算矩阵乘法的核函数的运行时间。
  由于核函数调用与主机端是异步的，需要使用 cudaDeviceSynchronize() 函数等待所有的 GPU 线程运行结束。
- initialData(): 对 host 的矩阵通过随机数进行初始化。
- printMatrix(): 打印矩阵，用于 debug。
- checkResult(): 逐个检验两个矩阵的元素。如果第 i 个元素不相等，输出第 i 个结果不匹配；否则返回检验成功。

method\n	32	64	128	256	512	1024	2048
cpu	0.114	0.947	8.534	69.183	790.727	8294.088	200934.2
strassen	0.268	2.027	16.686	136.927	1109.601	8958.394	75672.4
cublas	0.032	0.024	0.026	0.045	0.135	0.701	4.624
mul1	0.032	0.055	0.201	1.416	10.212	79.076	572.109
mul2	0.013	0.025	0.025	0.094	0.584	4.474	28.342
mul3	0.012	0.016	0.024	0.108	0.598	4.535	29.611
mul4	0.013	0.018	0.033	0.173	1.197	9.248	58.469
mul5	0.011	0.018	0.025	0.075	0.416	3.145	25.400
mul6	0	0.036	0.055	0.104	0.301	1.824	11.866
mul7	0	0.035	0.052	0.106	0.331	1.989	12.661

表 3. 程序运行结果：取矩阵大小 n 不同时的各方法运行时间，单位为 ms.

在主函数中首先对设置的矩阵大小 m 分配 host、device 内存空间，然后初始化两个输入矩阵。首先用 CPU 计算矩阵乘法，然后把各使用 GPU 的核函数和 CPU 计算的结果相比较并输出运行时间。

矩阵乘法正确性验证 每个核函数都用 util.cuh 中的 checkResult() 函数进行验证。经过测试各核函数结果正确。

矩阵乘法优化 我通过前面的原理和网络上的资料思考实现了七种方法对普通的 GPU 矩阵乘法进行了优化。但是由于硬件的发展，可能有些方法现在并不能实现性能上的优化，最后我只能做到缩小与 cublas 库函数的运行时间差距，但是 n 较大时还有一点差距。

各优化分别对应如下：

- gpu Matrix Benchmark: 使用 cublas 库函数.
- gpu Matrix multiplication: 朴素的 gpu 矩阵乘法，一个线程对应一个元素。
- gpu Matrix multiplication2: 通过 Tile 优化矩阵乘法。
- gpu Matrix multiplication3: 通过 Coalescing 优化矩阵乘法。
- gpu Matrix multiplication4: 避免共享内存冲突优化矩阵乘法。
- gpu Matrix multiplication5: 增加每次访存放到共享内存中的元素数量以达到减少访存次数的目的。本程序测试了每一次访存放到共享内存中的元素数量为情况 (WPT = 4)。
- gpu Matrix multiplication6: 计算优化，增加计算在所有指令中的比例，我的实现中每个线程块计算 B 的列数为原来的四倍。
- gpu Matrix multiplication7: 通过 Prefetch 优化矩阵乘法。

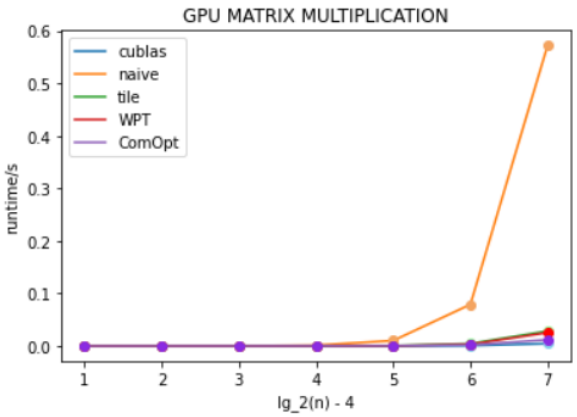

图 17. GPU 矩阵乘法运行时间比较

运行时间及分析

- 分别使用 CPU 和 GPU 计算矩阵乘法，运行时间差异非常大。通过我们对 GPU 矩阵乘法的增量改进，在矩阵大小为 2048×2048 时，mul6 比

朴素的 CPU 算法快了接近两万倍。

- 在朴素的 GPU 矩阵乘法的基础上，可以看到 1、2、5、6 四种优化的效果比较显著，对它们分别作图得到上图.
- 在经过优化后我们的实现已经非常接近库函数 cublas 的运行时间，但是在矩阵较大时会比 cublas 稍微差一些。

时间复杂度分析 以前学过的并行算法是通过分治实现的。CUDA 具有线程块和线程的概念，可以使用索引给线程分配具体的计算任务。所以实际的运行时间就是最大的线程运行时间。以第六种优化为例，每个线程计算矩阵 C 的一条竖着的向量，在我的代码实现中，向量的长度为 4。所以时间复杂度为 O(1)。从图中我们也可以看到，随着 n 的增大，运行时间的增加非常缓慢。

但是以朴素的 GPU 矩阵乘法为例，每个线程计算矩阵 C 中的一个元素，时间复杂度为 O(n)，随着 n 的增加，可以看到朴素的 GPU 矩阵乘法的运行时间也大幅度增加。

空间复杂度分析 需要在主机和设备之间拷贝输入矩阵和输出矩阵，所以空间复杂度为 $O(n^2)$.

5. 参考资料

电子版资料和参考的源代码已一并提交。