Project V: : GPU Acceleration with CUDA

2024 春季学期 CS205 Project3 **GPU Acceleration with CUDA**项目文档

姓名：陈峙安 **学号：12211829**

1.项目概述

在之前的项目中，我们探讨了 C 语言的**基本特性**，并探索了在**CPU**上的矩阵乘法性能。当然，执着于让CPU进行大规模浮点数运算还是有些强人所难，因此在本项目中，我们着重于学习如何通过用**CUDA**利用**NVIDIA GPU**的特性加速矩阵乘法。事实上，GPU上的浮点运算一直是很重要的课题，尤其在深度学习、科学计算等领域。在本项目中，我们将以矩阵乘法为媒介，学习CUDA编程的**特性和优化方法**。

鉴于 **Dear.Yu** 提供了 "cannout great enough" 的Linux服务器，本项目的**大部分**实验和数据都通过该服务器得到：`NVIDIA GeForce RTX 2080 Ti x 4` 。虽然是公开服务器，但因为显存有限且GPU计算能力**确实强大**，并未出现如Project III中的资源争夺问题。

计时方面，由于涉及**不同设备**上的任务传递，我使用`cudaEventRecord()`、`cudaEventSynchronize()`和`cudaEventElapsedTime()`来计时。它们可以精确地获取GPU执行核心代码所需的时间，避免例如CPU的负载、操作系统的调度等造成的影响，可以更加准确地评估GPU计算的性能。

代码部分采用"宏断言"`(Macro Assertion)`的形式检查**矩阵合法性**，并针对不同情况打印错误信息。特别地，我们在**向CUDA分配内存**时，也需检查是否分配成功等问题。

2.探索过程与实验

在project3的探索过程中，我事实上已经尝试了CUDA编程，实现了**共享内存**和**瓦片优化 (Tiling)**，针对warp特性进行了优化，并大言不惭地说出：

> 可以看到，经过了一些优化后，我的CUDA版本矩阵乘法对于大矩阵的表现性能相当好，**优于OpenBLAS**！

事实上，对于float矩阵乘法而言，讨论GPU与CPU的性能差距并无意义。当前算法和cuBLAS相比依旧**差距巨大**，我们需要从头开始探索GPU的魅力。

2.1 CUDA基础

CUDA（Compute Unified Device Architecture）是由NVIDIA开发的一种并行计算平台和编程模型。在**CUDA编程模型**中，**线程**、**线程块**、**网格**是核心概念，它们构成了CUDA并行计算的基本

单位。

2.1.1 线程 (Thread)

定义

线程是CUDA程序中最小的计算单元。每个线程独立执行**相同的**程序代码（内核函数），但可以处理不同的数据。

线程索引

每个线程都有一个**唯一的索引**，这些索引用于标识线程在整个计算中的位置。索引可以是一维、二维或三维的，通常通过以下内置变量来访问：

- `threadIdx.x`，`threadIdx.y`，`threadIdx.z`

2.1.2 线程块 (Thread Block)

定义

线程块是一组线程的**集合**。所有线程块中的线程并行执行，线程块内的线程可以**共享数据**，并可以通过同步机制进行协作。

线程块大小

线程块的大小是可配置的，并且必须在硬件限制范围内。每个线程块可以是一维、二维或三维的。线程块的大小通过以下内置变量来访问：

- `blockDim.x`，`blockDim.y`，`blockDim.z`

线程块索引

每个线程块在网格中的位置由其索引确定，索引可以是一维、二维或三维的。线程块索引通过以下内置变量来访问：

- `blockIdx.x`，`blockIdx.y`，`blockIdx.z`

2.1.3 网格 (Grid)

定义

网格是所有线程块的集合。一个网格中包含多个线程块，所有线程块**并行执行**。

网格大小

网格的大小同样是可配置的，并且可以是一维、二维或三维的。网格的大小通过以下内置变量来访问：

- `gridDim.x`，`gridDim.y`，`gridDim.z`

2.1.4 内核函数 (Kernel Function)

定义

内核函数是在GPU上执行的函数，由 `__global__` 修饰符标记。它是CUDA程序的**入口点**，一般针对**单个线程**进行操作，也是我们着重优化的位置。

2.2 初探CUDA

- 知晓了CUDA编程的基本流程后，我们便可以写出初版的**朴素乘法代码**了。

```
__global__ void matrixMulNaive(float *A, float *B, float *C, size_t N) {

    size_t bx = blockIdx.x, by = blockIdx.y;
    size_t tx = threadIdx.x, ty = threadIdx.y;
    size_t Row = by * BLOCK_SIZE + ty;
    size_t Col = bx * BLOCK_SIZE + tx;

    float temp = 0;
    for (size_t j = 0; j < N; ++j) {
        temp += A[Row*N + j] * B[Col * N + j];
    }
    C[Row*N + Col] = temp;
}
```

- 如上，我们让一个线程负责计算一个矩阵元素，这样的实现是最简单的，但是效率并不高。

具体来说，假设我们计算 `8192*8192` 大小的矩阵乘法，由于线程之间**完全独立**，那么理论上每个线程都需要从全局内存加载 `2*8192+1` 个浮点数. 由于我们总共有 8192^2 个线程, 这将导致 1096GB 的内存流量。

这是**不可接受的**，通过先前Project 3中的探索，内存访问是制约GFLOPS的主要瓶颈。具体来说，相比于CPU的几个到几十个核心，GPU有成千上万的计算核心，且对于“矩阵乘法”这种高度并行化的任务尤为擅长，它理应有更好的性能表现。然而，矩阵乘法需要**频繁地**访问和操作内存中的数据，这就导致了内存访问成为GPU计算的瓶颈，也是我们需要优化的重点，这与我们在CPU上的优化类似。

2.3 共享内存(shared memory)

在project3中，我们知晓了分块运算的优越性，它秉持着将载入缓存的数据**彻底利用**、“吃干抹净”之后再丢出内存，我们可以将这种思路用于GPU优化上，这便是共享内存。共享内存是CUDA中的一种特殊内存，它是线程块中的线程共享的内存，可以用于线程之间的通信和协作。共享内存的访问速度比全局内存**快很多**，因此可以用来优化内存访问。

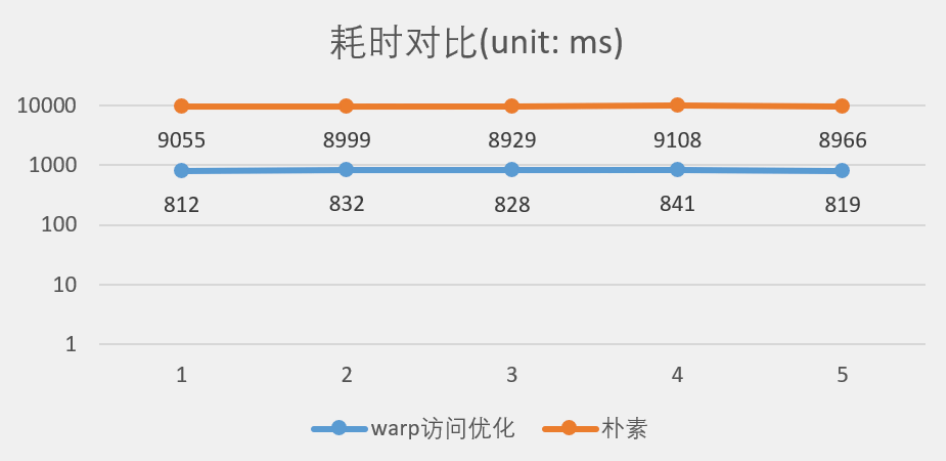
- 具体来说，虽然线程之间的计算是相对独立的，但是它们使用的数据却**非常相似**，通过对shared memory的设计，我们可以大大避免数据的**重复载入**。

下面是利用了共享内存的一种设计：

```cpp
__global__ void matrixMul(float *A, float *B, float *C, size_t Width) {
    __shared__ float share_A[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float share_B[BLOCK_SIZE][BLOCK_SIZE];

    size_t bx = blockIdx.x, by = blockIdx.y;
    size_t tx = threadIdx.x, ty = threadIdx.y;
    size_t Row = by * BLOCK_SIZE + ty;
    size_t Col = bx * BLOCK_SIZE + tx;

    float temp = 0;
    for (size_t j = 0; j < Width/BLOCK_SIZE; ++j) {
        share_A[ty][tx] = A[Row*Width + j*BLOCK_SIZE + tx];
        share_B[ty][tx] = B[Col*Width + j*BLOCK_SIZE + ty];
        __syncthreads();

        #pragma unroll
        for (size_t k = 0; k < BLOCK_SIZE; ++k) {
            temp += share_A[ty][k] * share_B[k][tx];
        }
        __syncthreads();
    }
    C[Row*Width + Col] = temp;
}
```

- 更有趣的是，由于多个线程可以同时向 `shared memory` 的不同位置载入数据，那么某个线程计算时不仅可以使用自己写入的数据，还可使用其他线程“**帮忙**”写入内存的数据，这种并行化的优势也可在共享内存的使用中得到体现。
- 使用了两次 `__syncthreads()`，第一次用于确保所有线程都已经将对应数据载入共享内存，第二次用于确保所有线程都已经计算完毕自己的部分，这是并行化载入数据所必须的操作。
- 在计算某一分块时还启用了 `#pragma unroll` 编译指令用于循环展开，用以减少循环开销、提高并行效率，这里不过多赘述。

这里简单解释下 `__syncthreads()`，这是针对线程的**同步化**命令，它用于对一个线程块中的所有线程的运行同步至某点，其实现背后涉及硬件级别的支持，以确保高效的同步操作。每个线程块在GPU的流多处理器（Streaming Multiprocessor, SM）上运行，SM包含的硬件调度器负责管理线程块内的线程并提供**同步机制**。硬件会确保在调用 `__syncthreads()` 时，所有线程在同步点等待，直到所有线程都到达该点。



运行结果如上图所示，减少了内存访问次数后，运行耗时缩短了十几倍，也从侧边证明了共享内存的访问速度比全局内存快很多。

2.4 探索warp

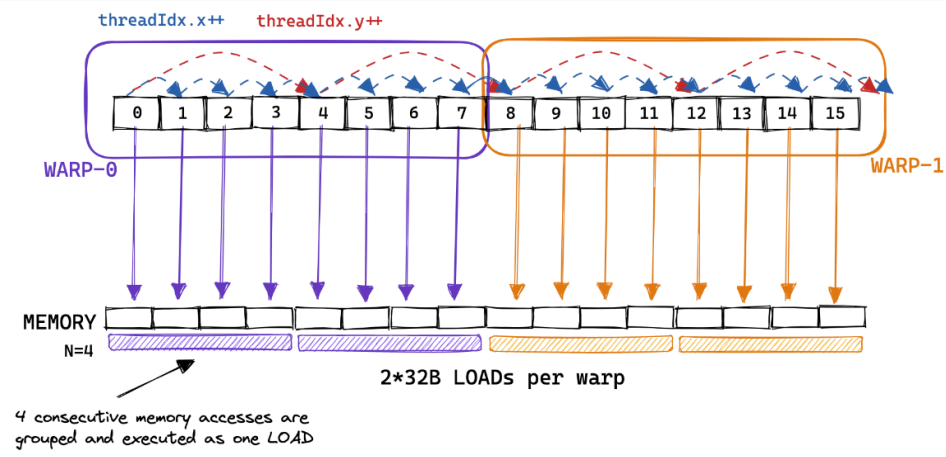
- 在CUDA中，线程是以**warp**的形式组织的，一个warp一般包含32个线程，这些线程在执行相同的指令，但是处理不同的数据。在CUDA中，warp是**最小的调度单位**，一个warp中的线程是**同时执行**的。因此，上文提到的 `__syncthreads()` 实际上是同步线程块中各个 `warp` 的操作。
  .
- 同时，warp是基于前文提到的 **SM** 包含的调度器控制的，其核心便是GPU的SIMT模型了。SIMT（Single Instruction, Multiple Threads）是一种并行计算模型，它允许**多个线程**同时执行相同的指令，但是处理不同的数据。这种模型类似于SIMD（Single Instruction, Multiple Data）模型，但无需开发者费力把数据凑成合适的矢量长度，并且SIMT允许每个线程有不同的分支———纯粹使用SIMD不能并行地执行有条件跳转的函数，而SIMT基于线程控制可以轻松做到。

名称	值
计算力 (Compute Capability)	7.5
每线程块最大线程数 (max threads per block)	1024
每 SM 最大线程数 (max threads per multiprocessor)	1024
每 warp 线程数 (threads per warp)	32
warp 分片粒度 (warp allocation granularity)	4
每线程块最多寄存器数 (max regs per block)	65536
每 SM 最多寄存器数 (max regs per multiprocessor)	65536
寄存器分配单元大小 (reg allocation unit size)	256
总全局内存 (total global mem)	11352866816 bytes
每线程块最大共享内存 (max shared mem per block)	49152 bytes
CUDA 运行时每线程块共享内存开销 (CUDA runtime shared mem overhead per block)	1024 B
每 SM 共享内存 (shared mem per multiprocessor)	49152 bytes
SM 数 (multiprocessor count)	68
每 SM 最大 warp 数 (max warps per multiprocessor)	48
最大线程维度 (Max threads dim)	[1024, 1024, 64]
最大网格大小 (Max grid size)	[2147483647, 65535, 65535]

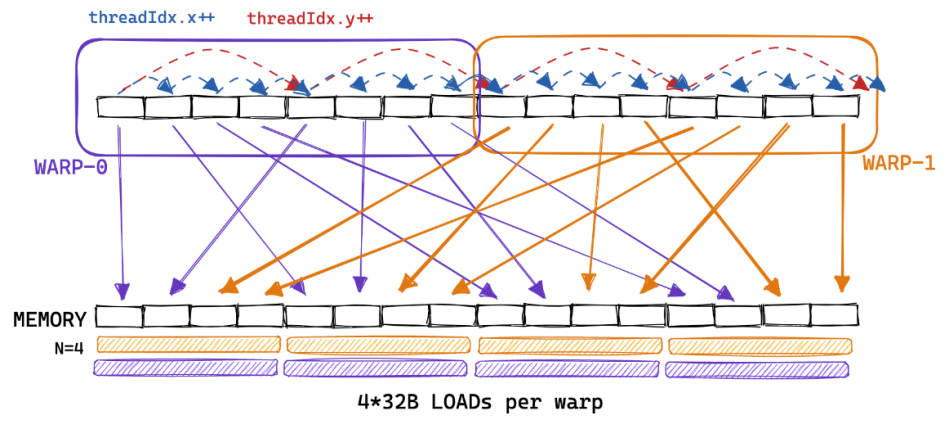
以上是从`cudaGetDeviceProperties` API 获得的我的 GPU 的相关硬件统计数据。可以看到warp的值是32，这意味着无论我们如何分组，真正的warp**总是包含32个线程**，并被统一分配给warp调度器。

值得注意的是，warp并非代表简单的线程集合，因为属于同一 warp 的线程的顺序内存访问可以被组**合并**作为一个整体执行。这被称为**全局内存合并**，是在优化内核的全局内存(GMEM)访问以达到峰值带宽时最重要的事情。

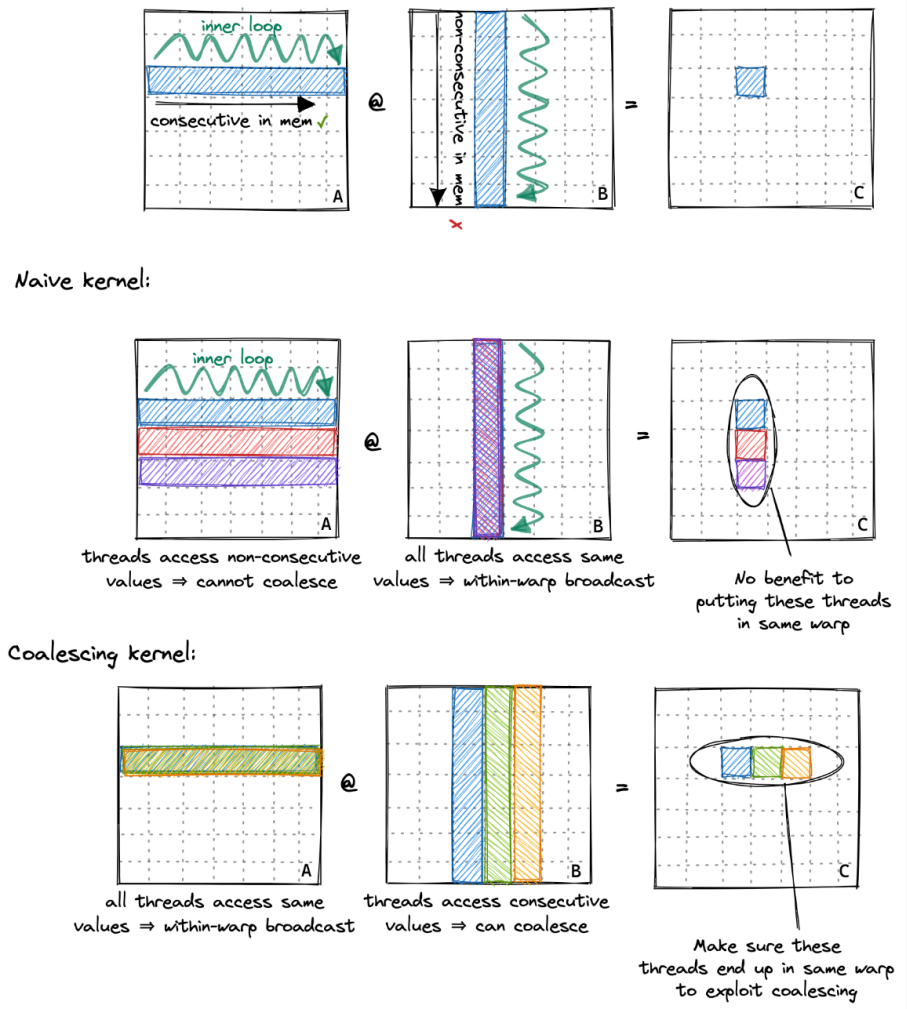
具体来说，正如CSDN中的一篇文章所画：



当加载的浮点数在内存中是**连续**的，并且访问是对齐的时候，warp调度器会把这一系列的加载**合并为一个事务**，从而大大增加内存吞吐速率。
然而，在我们之前的实现
中，`size_t Row = by * BLOCK_SIZE + ty;`和`size_t Col = bx * BLOCK_SIZE + tx;`实际上打乱了这一顺序(如下图所示)：



- 这将导致内存访问无法合并，GPU将每个内存读取视作独立的操作，致使大量带宽浪费。
  流程示意图如下所示(那篇文章真的把这方面讲得很清楚)：



为了实现图中的效果，我们需要对下标的分配做一些调整。首先，warp的划分是根据thread.x的值来的，故我们的目标是让具有**连续threadIdx.x**的线程从内存中连续地加载A的行。

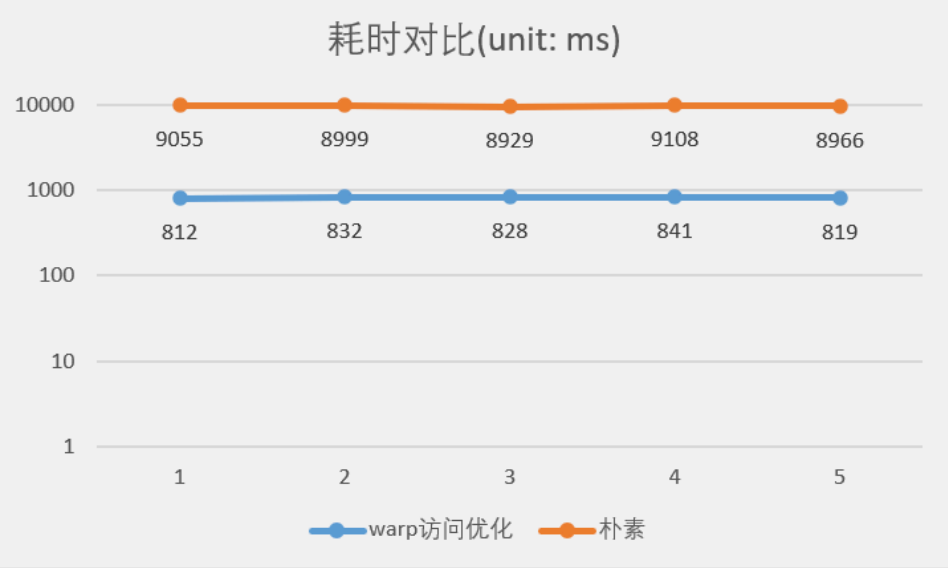
首先，我们可以先观察载入数据时我们使用的代码 `temp += A[x * K + i] * B[i * N + y];`，考虑到一个warp中的32个线程其实是同步的，我们可以假设不同线程的i值相同，故我们实际需要让**代表Col的y值连续**。

为此，我们可以这样修改先前的代码：

```
size_t Row = by * BLOCK_SIZE + ty;
size_t Col = bx * BLOCK_SIZE + tx;
↓↓↓
size_t Row = bx * BLOCK_SIZE + tx / BLOCK_SIZE;
size_t Col = by * BLOCK_SIZE + tx % BLOCK_SIZE;
```

同时将 `dim3 threadsPerBlock(BLOCK_SIZE,BLOCK_SIZE);` 由二维改为一维的 `dim3 threadsPerBlock(BLOCK_SIZE*BLOCK_SIZE);`，保证线程的分配是连续的。其余代码均保持不变。

运行结果如下：



可以看到仅仅通过改变下标的分配，我们就能有**数十倍**的性能提升，这也是GPU优化中的另一个重要技巧。

这也解释了我在**project3 report**中无法理解的、只能用GPT回答的问题：为什么先`tx`再`ty`的访问顺序会有数倍的性能差距问题，也算是“圆梦”了。

2.5 算法结构探索

然而，**8k** 大小矩阵乘法耗时800ms与cuBLAS相比依旧**太慢了**，也与2080Ti的理论算力不符，问题出在什么地方呢?

这里使用`Nsight computing`进行性能分析。经过warp 状态的采样后得知：`Stall MIO Throttle`占据了绝对的主导地位，在NVIDIA官方手册的`ProfilingGuide`中提到其意义为

`Warp was stalled waiting for the MIO (memory input/output) instruction queue to be not full.`

这意味着GPU内核发出了大量的**内存访问指令**，尤其是访问共享内存(**SMEM**)。

那么，如何让内核发出更少的内存访问指令呢？答案有点反直觉——增加单个线程的**计算量**。在project之初，我们曾以GPU的庞大核心数量为傲，下意识将计算任务分成了单线程单结果矩阵值的最小颗粒度形式，然而这种方式导致了内存访问时间在我们的总耗时中占比巨大，现在我们需要提高单个线程的计算量，考虑到硬件配置，8 * 8 的大小正合适，即单个线程负责计算结果矩阵的**64个元素**。

同时，由于负责运算的值增多，单个线程又会更多地访问重复位置的共享内存。即使访问共享内存相较全局内存已经很快了，但有没有方法**进一步优化**呢?

答案是当然有！前文中我们曾提过`寄存器`的字眼，类似于CPU优化中尽可能把数据放入Cache，我们也应将重复多次利用的数据用**寄存器储存**，从根源上减少对共享内存的访问。

最后，在CPU上，我们曾使用了SIMD的优化方式大大优化了运行时间，那么在GPU上我们有没有类似的方式呢？这里我选择使用**向量数据类型**`float2`进行一定程度上的向量化优化。

3. 最终实现与比较

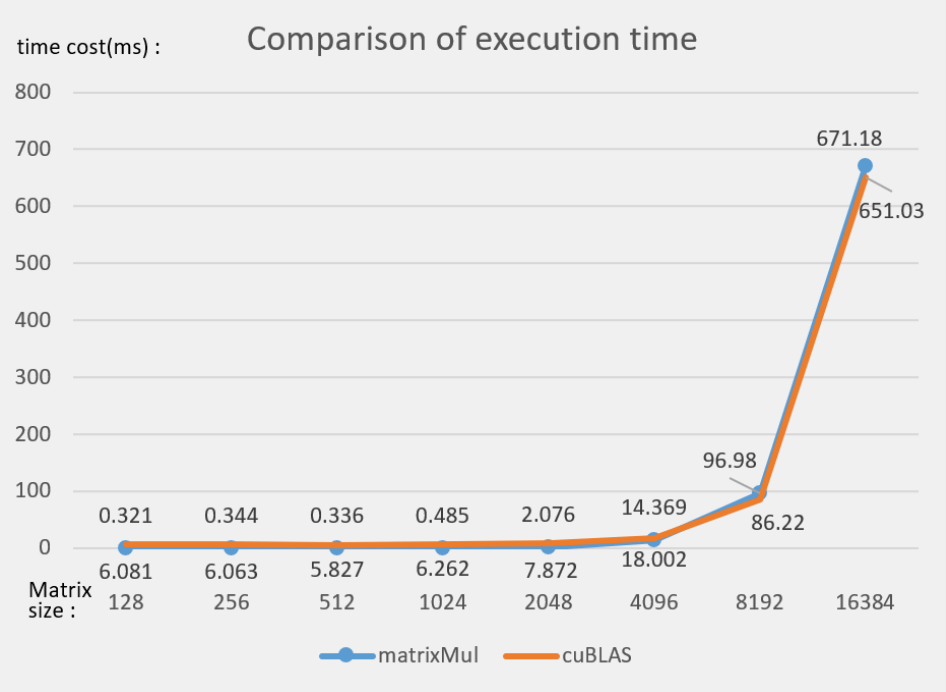
总结下目前我们得到的所有优化方向：共享内存、寄存器优化、warp访问连续化、分块与线程大小、向量化。在最终实现中，我们尽可能将这些优化方向**结合**起来，得到了以下代码：

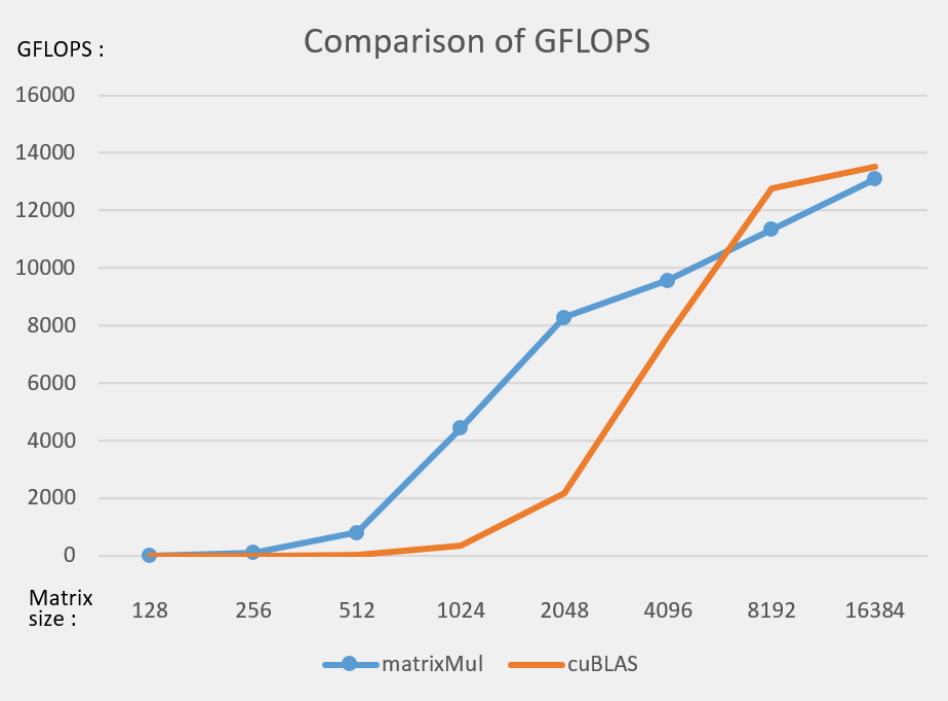
```
__global__ void matrixMul(float* __restrict__ d_A, float* __restrict__ d_B,
float* d_C, int n) {
    __shared__ float2 s_a[512];
    __shared__ float2 s_b[512];
    // Thread result matrix
    float c[64] = {0.f};
    // Thread registers
    float2 a[4];
    float2 b[4];
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int x = threadIdx.x;
    // Continuous warp access
    int tidx = x % 32;
    int tidy = x / 32;
    int v = x % 16;
    int u = x / 16;

    d_A += (by * 128 + tidx) + tidy * n;
    d_B += tidy * n + bx * 128 + tidx;
    d_C += (by * 128 + u * 2) * n + bx * 128 + v * 2;

    for (int i = 0; i < n; i += 8) {
        ((float*) s_a)[tidy * 128 + tidx] = d_A[0];
        ((float*) s_a)[tidy * 128 + tidx + 32] = d_A[32];
        ((float*) s_a)[tidy * 128 + tidx + 64] = d_A[64];
        ((float*) s_a)[tidy * 128 + tidx + 96] = d_A[96];

        ((float*) s_b)[tidy * 128 + tidx] = d_B[0];
        ((float*) s_b)[tidy * 128 + tidx + 32] = d_B[32];
        ((float*) s_b)[tidy * 128 + tidx + 64] = d_B[64];
        ((float*) s_b)[tidy * 128 + tidx + 96] = d_B[96];

        __syncthreads();

#pragma unroll
        for (int k = 0; k < 8; k++) {
            // load relational s_a,s_b data into a,b registers
            (notice that s_a,s_b,a,b are all float2 type arrays)
            // use a,b to calculate c
            (accumulate the results in the corresponding c array)
        }

        __syncthreads();
        d_A += 8 * n;
        d_B += 8 * n;
    }
```

```
#pragma unroll
    // write the final value to global memory
    for (int j = 0; j < 8; j += 2) {
        d_C[0] = c[j];
        d_C[1] = c[j + 8];
        d_C[32] = c[j + 16];
        d_C[33] = c[j + 24];
        d_C[64] = c[j + 32];
        d_C[65] = c[j + 40];
        d_C[96] = c[j + 48];
        d_C[97] = c[j + 56];
        d_C += n;
        d_C[0] = c[j + 1];
        d_C[1] = c[j + 9];
        d_C[32] = c[j + 17];
        d_C[33] = c[j + 25];
        d_C[64] = c[j + 33];
        d_C[65] = c[j + 41];
        d_C[96] = c[j + 49];
        d_C[97] = c[j + 57];
        d_C += 31 * n;
    }
}
```

这个实现看起来似乎很简单，但站在单个线程的角度**构思全局**实在不是一件容易事，更别说还要考虑一堆优化方式了。这是CUDA编程的最主要特点，不得不品。

以下是该代码与cuBLAS的对比。

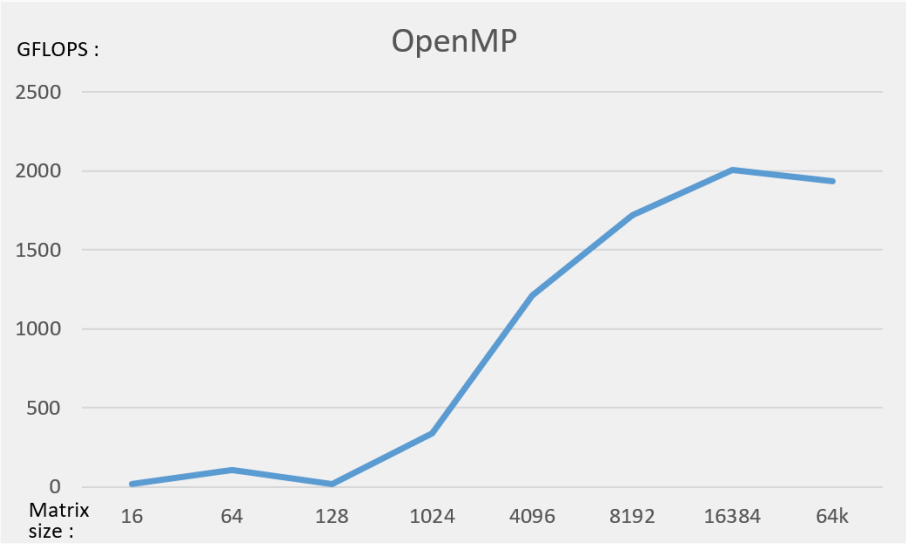




Matrix size	128	256	512	1024	2048	4096	8192	16384
matrixMul (ms)	0.321	0.344	0.336	0.485	2.076	14.369	96.98	671.18
cuBLAS (ms)	6.081	6.063	5.827	6.262	7.872	18.002	86.22	651.03
GFLOPS_matrixMul	13.066368	97.541953	798.915048	4427.801336	8275.466852	9564.963009	11337.509051	13105.415868
GFLOPS_cuBLAS	0.689739	5.534295	46.067523	342.938941	2182.402081	7634.649121	12752.396518	13511.041000

可以看到，总体效果还是**相当不错**的。其中，对于小型矩阵(小于4k)的计算时，我们的程序都**优于**cuBLAS,可能是因为cuBLAS的初始化开销较大。而对于大型矩阵，我们的程序则与cuBLAS**只有5%左右**的差距，考虑到cuBLAS对NVIDIA GPU的硬件特化程度，这是完全可以接受的。

- 附上**OpenBLAS**的运行速度以作比较 ~~(虽然我认为让 `CPU` 和 `GPU` 比 `FLOPS` 还是过于“超前”子)~~



这里再提一下project **要求1**里实现的:

$$\mathbf{B} = a\mathbf{A} + b$$

其主要为内存密集型任务，对计算的需求量小很多，8k矩阵的耗时也在**1ms以内**。

```
__global__ void scaleAddKernel(float* A, float* B, float a, float b, int n) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x % 32;
    int idy = blockIdx.y * blockDim.y + threadIdx.x / 32;
    int index = idy * n + idx;
    float temp = A[index];

    B[index] = a * temp + b;
}
```

在调取前需使用`CHECK_MATRIX_VALID(matrixA, matrixB, N, N, N, -1);`进行**合法性检查**。

4. 一些发现与思考

- cuBLAS 与 OpenBLAS `sgemm`函数，但前者是**列主序**的，后者是**行主序**的，这是一个很有意思的发现，也是我在实现中遇到的一个小坑。
- GPU 和 CPU上的矩阵乘法**精度差异**较大，且CPU更为准确。这是因为CPU端计算浮点数默认是double精度，而GPU端是float精度。解释是float对于很多图形和科学计算中已经足够精确(希望如此吧)。
- 与OpenBLAS的GFLOPS图相比，GPU上的GFLOPS随着矩阵大小的增加而迅速增加，且暂无停止趋势，这或许意味着我们对GPU的优化还有一定空间，依旧没有达到其**计算瓶颈**。

特别声明：本文在**warp**部分使用的一些图片和优化思路来自于CSDN上的一篇博客，特此感谢。

5. 感受与收获

该项目仍有很多不足之处，很多数据受时间原因并没有做大量测试，并且最终版参数的设置实际上还有一些优化空间，并且我对这些现象没有找到比较权威的解释。

项目的整个实践过程对我来说收获了很多。从对GPU硬件的探索过程中，我深刻认识到算法的重要性，硬件的性能特点和底层机制需借助完善的算法体现。这对我而言是一次宝贵的学习经历，尤其是具体编写CUDA代码时，从单个线程的角度思考问题的方式非常有趣。

在具体实现中，我深刻体会到理论与实践之间的差异。理论上可能看起来完美的优化，在实际操作中可能因为硬件的限制、编译器的差异、操作系统的底层实现甚至是自己的误会等因素，而无法达到预期的效果。(例如指令集版本、CUDA Toolkit的文件编码bug、nvcc的编译架构选项等)

这是本门课程的最后一个project，百感交集之下，只能感谢于老师的布置。这门课使我第一次接触类似“实现”、“优化”、“探索”类型的project，却也是我本学期收获最多的一门课。对于没上过计组的我而言，这是一次很好的补充，令我对计算机硬件本身有了更深的理解，推荐所有计系人都来上一遍Yu++!