

CS205: Project 5: GPU Acceleration with CUDA

杨彦卓 (南方科技大学, 通识与学科基础部)

2024.06.02

摘 要

该文章从尝试用 CUDA 实现并优化矩阵的数乘 $B = aA + b$, 并分析线程块大小对效率的影响。最后引入 cuBLAS 和 CPU 层面的写法进行分析对比。

目录

1	环境及声明	1
1.1	简称声明	1
1.2	关于生成式人工智能的声明	1
1.3	开发环境	1
1.3.1	本地 NVIDIA 硬件信息	1
1.3.2	本地 nvcc 编译器信息	1
1.3.3	Linux Server 硬件信息	1
1.3.4	Linux Server nvcc 编译器信息	2
2	引言	2
3	CUDA 实现	2
3.1	矩阵实现	2
3.2	矩阵操作实现	3
3.3	时间测量实验	5
3.4	实验结果分析	6
4	优化思路	6
5	cuBLAS 对比与分析	7
6	CPU 与 GPU 对比	9
7	其他实验	10
7.1	本地测试	10
7.2	局限性分析与改进实验	10
8	结语	11
8.1	学习收获	11
8.2	心得感悟	12

1 环境及声明

1.1 简称声明




本次 Project 最关心的矩阵运算为：对于某一给定矩阵先执行数乘再进行加法操作，然后赋值给其他矩阵，即 $B = aA + b$ 。该操作在后文没有特殊说明的情况下简称**矩阵操作**。

1.2 关于生成式人工智能的声明

本次 Project 代码中一些较为简单的部分由 ChatGPT 4o (OpenAI, 2024-06) 编写 (对于已有的**矩阵操作**函数在 main.cu 中进行时间测试)。但对于测试代码、撰写报告、图表设计等流程均不涉及任何生成式人工智能。

1.3 开发环境

1.3.1 本地 NVIDIA 硬件信息

文件名	文件版本	产品名称
3D 设置		
 nvGameS.dll	31.0.15.5161	NVIDIA 3D Settings Server
 nvGameSR.dll	31.0.15.5161	NVIDIA 3D Settings Server
 NVCUDA64.DLL	31.0.15.5161	NVIDIA CUDA 12.4.89 driver
 PhysX	09.10.0513	NVIDIA PhysX

1.3.2 本地 nvcc 编译器信息

```
yangyang@MyY9000X:~$ nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Mon_Oct_24_19:12:58_PDT_2022
Cuda compilation tools, release 12.0, V12.0.76
Build cuda_12.0.r12.0/compiler.31968024_0
```

1.3.3 Linux Server 硬件信息

- CPU: Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz, 24 Cores
- Memory: 128GB
- GPU: NVIDIA GeForce RTX 2080 Ti x 4
- GCC: 11.4.0

1.3.4 Linux Server nvcc 编译器信息

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2024 NVIDIA Corporation
Built on Thu_Mar_28_02:18:24_PDT_2024
Cuda compilation tools, release 12.4, V12.4.131
Build cuda_12.4.r12.4/compiler.34097967_0
*
```

2 引言

在本次 Project 开始之前，我了解了 CUDA 并尝试分析 CUDA 执行矩阵操作的优势：矩阵运算中的每个元素的计算是独立的，可以同时进行。它可以利用 GPU 的并行计算能力，让同一时间的每个线程处理一个或多个矩阵元素。这使得运算速度极大加快。

本次实验将用 CUDA 实现矩阵的初始化以及矩阵操作，然后对其进行优化，计算各个优化思路下的运行时间，将其与 cuBLAS 库内实现的矩阵操作运行时间进行对比并分析。最后，本次实验还会尝试在 CPU 的层面运算矩阵操作，同样进行对比分析。

3 CUDA 实现

3.1 矩阵实现

```
void initMat(float* matrix, int rows, int cols) {
    for (int i = 0; i < rows * cols; i++) {
        matrix[i] = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
    }
}
```

使用 malloc() 函数分配内存后调用 initMat() 函数初始化矩阵: 随机生成 0-1 之间的值赋值给该矩阵中的元素。

```
float *A;
int size = matrixSize * matrixSize;
A = (float*)malloc(size * sizeof(float));
initMat(A, matrixSize, matrixSize);
```

3.2 矩阵操作实现

首先说明一些 CUDA 中将要应用的一些概念：

- **线程 (Thread)**: 线程是 CUDA 编程的基本执行单元。在这个 Project 中，每个线程处理矩阵中的一个元素的计算。CUDA 线程是在 GPU 上并行执行的。
- **线程块 (Block)**: 线程块是一个线程的集合，这些线程可以共享一些资源，如共享内存。
- **网格 (Grid)**: 网格是由多个线程块组成的集合。网格用于覆盖整个数据集。在这个示例中，网格中的每个线程块处理矩阵的一部分。

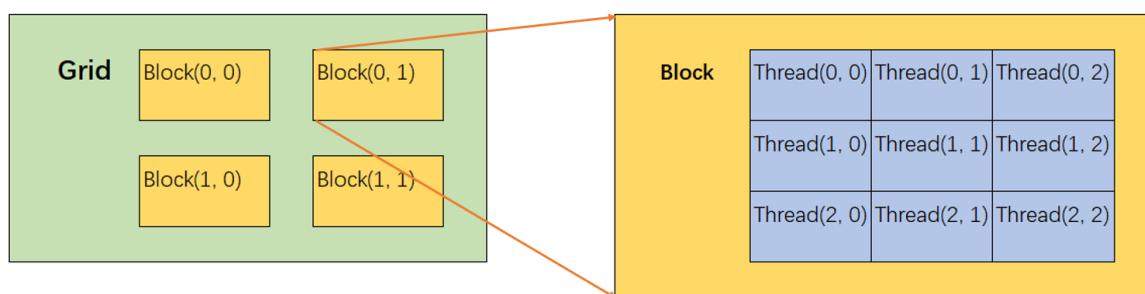


Figure 1: 自制 CUDA 概念示意图

理论上来说，同一个网格内的所有线程将会并行操作。由此我遇到了第一个问题：为什么需要 Block 这个中间层？为什么不直接让 Grid 接管所有 Thread 并让这些 Thread 并行？

分析后发现问题的本质是：Block 的尺寸如何调整？既然矩阵内的所有数据相互独立，能否让 Block 的尺寸大到接管所有数据？

查了一些资料，主流的观点一般认为：块内的线程可以共享使用一块高速的共享内存，这对一些需要频繁访问和更新的数据能显著提升性能。网格级别的全局内存访问相比之下较慢。

而分块能让 CUDA 访问到多个共享内存。但我认为矩阵似乎本身就可以用一个共享内存来指向所有数据。所以能否让 Block 的尺寸大到接管所有数据，进而只使用一个 block 块？我很难单从理论上来分析答案，于是有了后续的实验过程来证实结论。

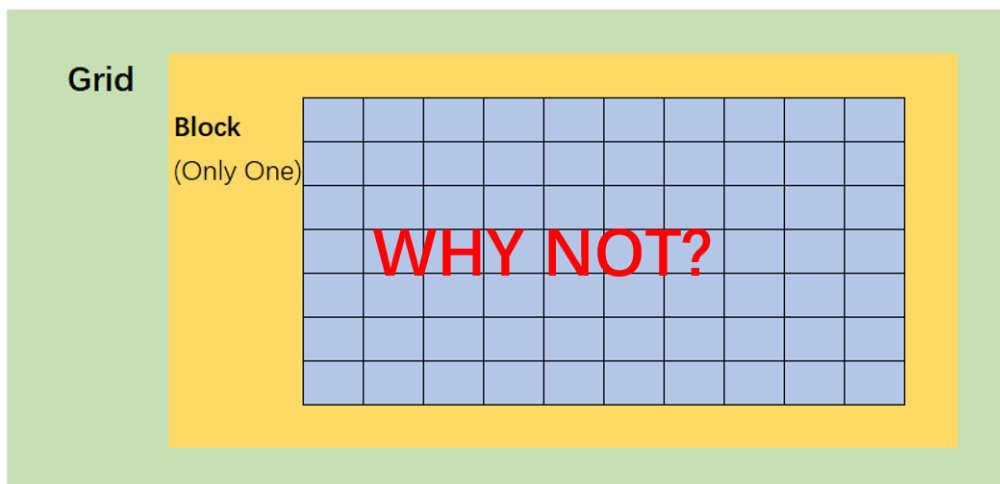


Figure 2: 单 Block 执行示意图

在实验前，首先实现一个相对简单的函数，用于计算矩阵操作。矩阵的初始化在 CPU 层面执行，我们需要将其转移至 GPU 层，再进行后续处理。

```
float *data_A, *data_B;
int size = rows * cols * sizeof(float);

cudaMalloc((void**)&data_A, size);
cudaMalloc((void**)&data_B, size);

cudaMemcpy(data_A, A, size, cudaMemcpyHostToDevice);
```

根据本学期另一课程：CS202: Computer Organization 对于缓存块与缓存数据的访问关系，转换下思路后我们很容易得出矩阵中数据的访问索引：blockDim.x 和 blockDim.y 分别表示线程块在 x 和 y 方向上的维度，blockIdx.x 和 blockIdx.y 分别表示当前线程块在 x 和 y 方向上的索引，blockDim.x 和 blockDim.y 分别表示线程块在 x 和 y 方向上的宽度，threadIdx.x 和 threadIdx.y 分别表示当前线程在线程块内 x 和 y 方向上的索引。由此计算出 idx 与 idy：当前线程在整个网格中的 x 和 y 方向全局索引。

```
__global__ void matTransKernel(const float* A, float a, float b, float* B, int rows, int cols) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    int idy = threadIdx.y + blockIdx.y * blockDim.y;
    int index = idy * cols + idx;
    if (idx < cols && idy < rows) {
        B[index] = a * A[index] + b;
    }
}
```

将数据转移至 GPU 层后，类似的思想逆向后可以算出方形线程块的数量，然后开始运行，最后再把数据传回 CPU 层。

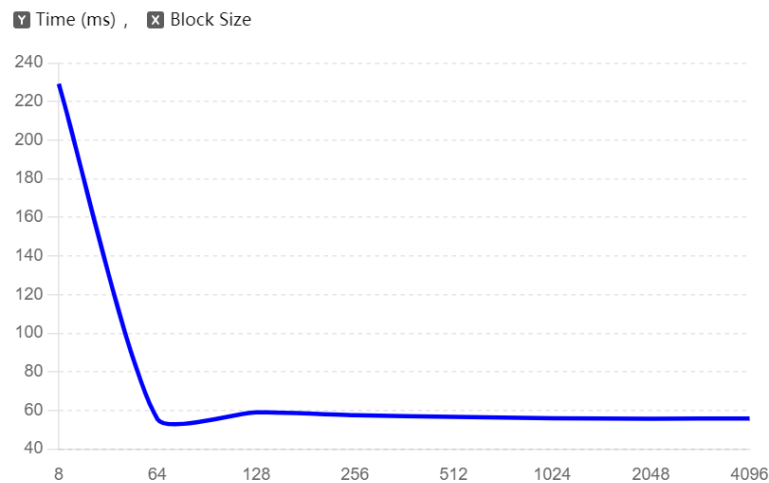
```
cudaMemcpy(data_A, A, size, cudaMemcpyHostToDevice);

dim3 blocksPerGrid((cols + threadsPerBlock.x - 1) / threadsPerBlock.x,
                  (rows + threadsPerBlock.y - 1) / threadsPerBlock.y);

matTransKernel<<<blocksPerGrid, threadsPerBlock>>>(data_A, a, b, data_B, rows, cols);
cudaMemcpy(B, data_B, size, cudaMemcpyDeviceToHost);
```

3.3 时间测量实验

在 main 函数里，我选用了尽量大的方阵尺寸——4096*4096 以便于观察程序运行的时间差异，选择了若干个不同的方形线程块大小来进行比较，结果如下：



可以发现：当块大小达到 64*64 时，运行速度就几乎不优化了。在块大小达到最大——块数量仅为 1 时，运行速度也没有达到预期的结果。

后续我还测试了更大的尺寸，结果都大差不差——都在块大小为 64*64 时趋于平缓，与块最大时几乎没有时间差。

我查阅了更多 CUDA 的资料，发现最开始的理解过于简单：线程块内的并行原理是：同一个线程块内有若干线程束 (warp)，每个线程束内的 32 个线程完全严格并行 (Single Instruction Multiple Thread, SIMT)。多个线程束之间也支持并行从而达到加速效果，但具体的并行取决于流式多处理器 (Streaming Multiprocessor, SM)，SM 可能只能调度执行部分线程束，其他线程束则等待——具体取决于硬件资源和调度策略。

3.4 实验结果分析

当线程块尺寸较小、数量较多时，SM 的数量很可能饱和，不够分配到每一个线程块，从而导致其他线程块等待。

而一个 SM 只能处理有限大小的内存，当线程块数量较少时，有些 SM 可能空闲，分配不到线程块来进行工作。

在开篇提到的——仅一个线程块的想法——一个 SM 必然无法处理如此大的内存，CUDA 运行时系统会自动进行资源管理和调度 (在有限范围内)，它会尝试将线程块分解，最终还是会调整为更小、更多的线程块并分配给其他空闲 SM。

综述：当线程块很大时，运行时间也几乎和中等线程块一致。这是因为大线程块最终还是被分解为小线程块处理了。

4 优化思路

我在网上查找了一些有关 CUDA 加速矩阵乘法的例子，然后利用其中的优化思想重写了该矩阵操作：可以利用共享内存加速数据访问：共享内存是位于每个线程块内的高速缓存，每个线程块内的线程都可以快速访问共享内存。相比之下，一开始采用的全局内存的访问速度较慢，延迟较高。矩阵内的数据需要被频繁访问，把它们加载到共享内存可以大幅减少全局内存的访问次数，从而加快数据访问速度。但结果似乎并没有想象中的明显。

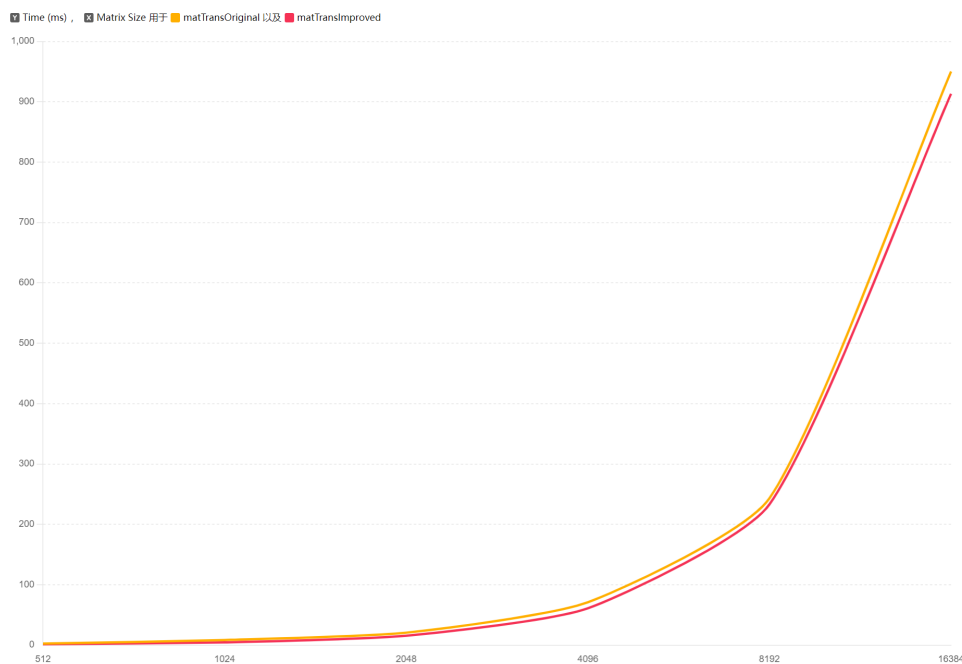
```
void matTransKernelAdv(const float* __restrict__ A, float a, float b, float* B, int rows, int cols) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    int idy = threadIdx.y + blockIdx.y * blockDim.y;
    int threadIdxInBlock = threadIdx.x + threadIdx.y * blockDim.x;

    extern __shared__ float shared_A[];
    shared_A[threadIdxInBlock] = A[idy * cols + idx];

    __syncthreads();

    B[idy * cols + idx] = a * shared_A[threadIdxInBlock] + b;
}
```

对于很多网上提供的思路，大部分就是利用共享内存来优化，我也尝试过很多中改法，但效果都很有限。而且此处只有一块需要关心的共享内存，所以也很难看到很大提升。



5 cuBLAS 对比与分析

在开始测量 cuBLAS 之前，我完全没有想到 cuBLAS 的效率与最终我写的优化结果接近。我在 Project3 中曾引入了 OpenBLAS 对比我实现的矩阵乘法，OpenBLAS 的效率要快数倍。所以我认为可能是我没有引入最合适的 cuBLAS。

过程中我测量了两种不同的 cuBLAS 的方法：

cublasSgeam: $C = \alpha * A + \beta * B$

```
void cublasSgeamTrans(const float* A, float a, float b, float* B, int rows, int cols) {
    float *d_A, *d_B;
    int size = rows * cols * sizeof(float);

    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);

    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);

    cublasHandle_t handle;
    cublasCreate(&handle);

    cudaMemset(d_B, 0, size);
    float alpha = a;
    float beta = b;

    cublasSgeam(handle, CUBLAS_OP_N, CUBLAS_OP_N, rows, cols, &alpha, d_A, rows, &beta, d_B, rows, d_B, rows);

    cudaMemcpy(B, d_B, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A);
    cudaFree(d_B);
    cublasDestroy(handle);
}
```

cublasSgemm: $C = \alpha * A * B + \beta * C$

```

void cublasSgemmTrans(const float* A, float a, float b, float* B, int rows, int cols) {
    float *d_A, *d_B;
    int size = rows * cols * sizeof(float);

    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);

    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);

    cublasHandle_t handle;
    cublasCreate(&handle);

    cudaMemcpy(d_B, 0, size);

    float alpha = a;
    float beta = b;

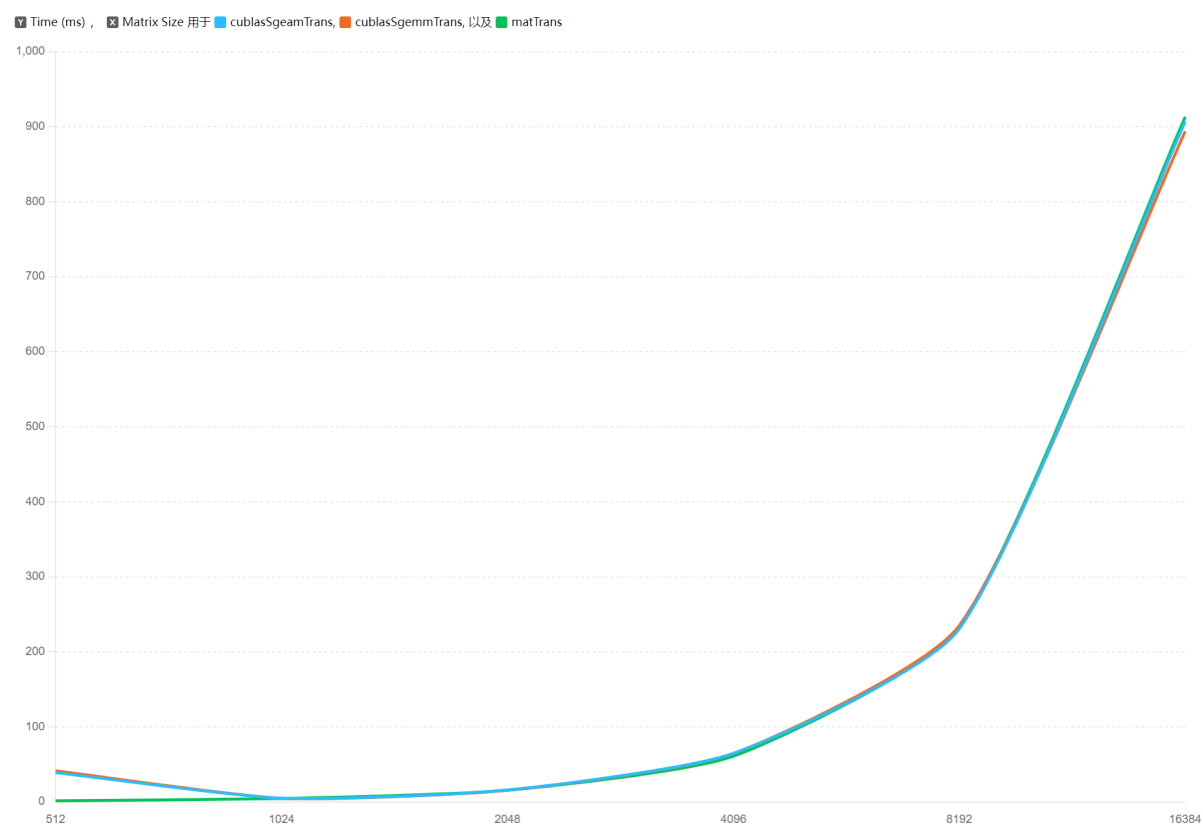
    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, rows, cols, 1, &alpha, d_A, rows, d_B, rows, &beta, d_B, rows);

    cudaMemcpy(B, d_B, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A);
    cudaFree(d_B);
    cublasDestroy(handle);
}

```

然而最后算出来的时间相差无几甚至非常接近，接近到多次让我怀疑是否是某部分的代码出了问题。但我用了很多方法方法排查潜在的错误都没有发现有何疏漏，这些方法包括但不限于：



- 用其他函数测量时间：比如 `auto start = std::chrono::high_resolution_clock::now();`

-
- 更改时间测量点的位置：比如把测量时间的语句放在 for 循环内部的最外侧再进行测量，最后时间理所应当变长了，但三种方法依旧没有很大差别：

```
for (int matSize : sizes) {  
    // measure time  
    // code  
    // calculate Matrix B = aA + b  
    // code  
    // measure time  
    // output time  
}
```

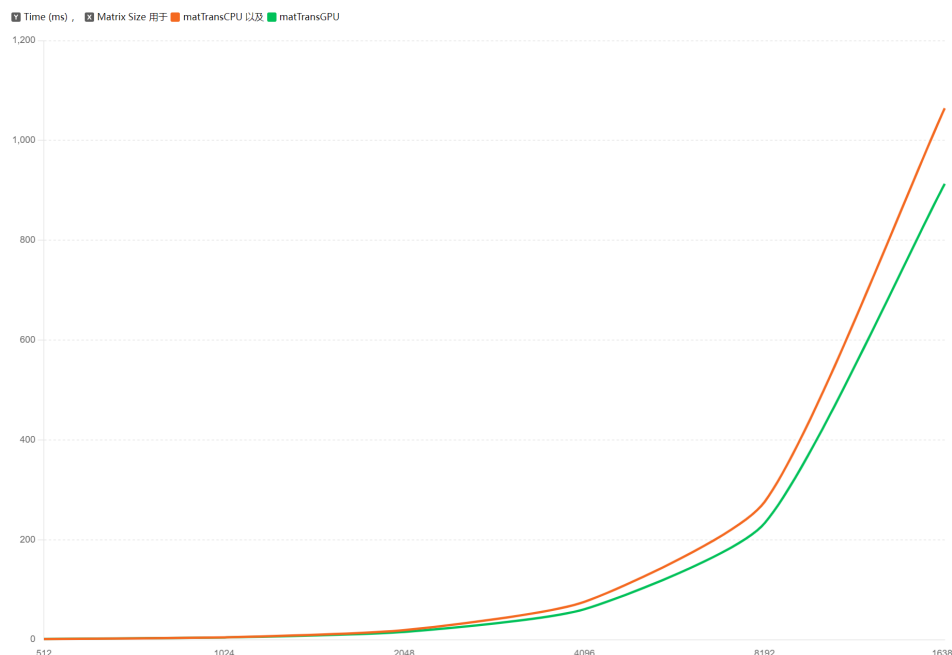
- 换用其他更小或者更大的矩阵尺寸；
- 病急乱投医：把三种方法分成三个 main 函数分别编译运行测量时间。

最后我认定，这是因为优化的结果较为理想。我只是优化到了接近 cuBLAS 库的程度，但在课程中我的同学无限骄傲地告诉我他写出来的结果比 cuBLAS 还要快。

6 CPU 与 GPU 对比

最后写了一个简单的 CPU 执行矩阵操作的程序，可以明显地看出两者之间的时间差。

```
void matTransCPU(const float* A, float a, float b, float* B, int rows, int cols) {  
    for (int i = 0; i < rows; ++i) {  
        for (int j = 0; j < cols; ++j) {  
            int index = i * cols + j;  
            B[index] = a * A[index] + b;  
        }  
    }  
}
```



结论：GPU 有明显的并行的效率优势

7 其他实验

7.1 本地测试

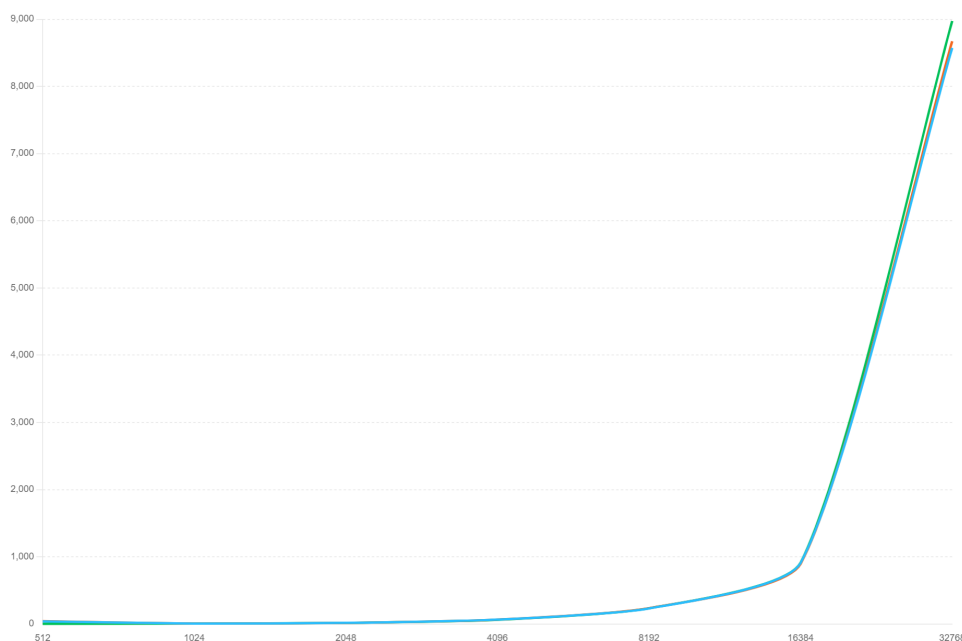
我在最后一天——大家都在赶ddl的时候，计算机系的设备曾短暂崩溃过，在等待之余我才意识到我的笔记本也搭载了英伟达产品。但目前的实验场景大部分位于图书馆等公共场合，我尝试用本地的GPU来测量时间，但无奈笔记本的风机声音实在太大了，周围的人都投来异样的目光。而且在配置cuda环境时已经发出了很久噪音了，重试了计系的服务后发现崩溃修复了，遂放弃。

7.2 局限性分析与改进实验

当时在实验过程中想了很多方法去验证时间测量的准确性，始终觉得自己写的应该达不到cuBLAS的运行效率。但都没想到最根本的原因——该算法的时间复杂度仅为 $O(n^2)$ ，难怪看不到矩阵乘法那样立竿见影的效果。但这一点我在分析了CPU和GPU的对比之后才意识到，二次方的增长速率不够快，所以对于所以的方法，在矩阵尺寸有限的情况下看不到很大的差异。

虽然在Project的文档里提示了实验过程中不要使用超过4096的尺寸，但在ddl的

最后半个小时内，我还是尝试了一下对于更大的矩阵尺寸的运行时间：最后还是相差无几：



8 结语

8.1 学习收获

我曾在 Project2、Project3 中的实验反思总结中都很明确地提到了一点——对于分块矩阵的优化思路，实验过程中没有斟酌块的大小，这导致优化效果不够。之前只是测量了不同块大小，发现块过大过小都无法达到最优，但并不知道原理。

1. Limitations

- The calculation of `BLOCK_SIZE` in `optimized_II` is not precise enough, which may leads to unsatisfactory result;

Figure 3: From Project 2 Report: Simple Matrix Multiplication

Project 3 Report: Improved Matrix Multiplication in C 反思总结节选：

分块大小选择：*OpenBLAS* 对于块大小的选择经过精心调整以匹配 *CPU* 的缓存系统。在手写的代码中，很难考虑到对于某一个具体的尺寸来说，如何调整块大小会使得效率达到最优。而不合适的块大小可能会导致缓存未命中率增高，从而影响性能。而经

过实验，简单地调整块大小在提高效率方面并不显著，并且目前选择的块大小已经是相对较好的值。块大小的选择需要大量测量数据分析；

虽然这一次依旧没能从该角度优化效率，但终于分析了其内在的原理。其实这些东西也完全没有想象中那么难。本次 Project 中，我对于线程、并行等待概念有了更加深刻的理解。

8.2 心得感悟