GEneral Matrix Multiplication：实现与并行优化

袁少贤
BA21221027

日期：2021/11/20

摘　　要

通用矩阵乘法是深度学习框架的核心计算单元之一，也是科学计算中最为重要的基石。通用矩阵乘法算法特征具有代表性，很多其他类似的算法可以通过相同的优化技巧实现性能提升。

关键词： 通用矩阵乘法、并行计算、共享内存

1　机器环境

系统软件环境为 Windows 10 专业版，版本号 21H1。CPU 全部算法在 VS code 上实现，使用 MinGW 软件，g++ 编译器。GPU 全部算法使用 Visual Studio 2017 进行编程，Cuda nvcc 编译器。

```
PS C:\Users\yuans\source\repos\bin\win64\Debug> g++ --version
g++.exe (x86_64-win32-seh-rev0, Built by MinGW-W64 project) 8.1.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

PS C:\Users\yuans\source\repos\bin\win64\Debug> nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2021 NVIDIA Corporation
Built on Mon_Sep_13_20:11:50_Pacific_Daylight_Time_2021
Cuda compilation tools, release 11.5, V11.5.50
Build cuda_11.5.r11.5/compiler.30411180_0
```

图 1: Compiler Versions

1.1　CPU

CPU 为 Intel 6700HQ，四核心八线程，32KB 一级缓存，256KB 二级缓存以及 6MB 三级缓存，并配有内存大小为 16GB。具体参数如图所示2

1.2　GPU

GPU 为 Nvidia GTX 1060，配备 6G 显存，每个 BLOCK 支持最大 1024 线程。具体参数如图3所示

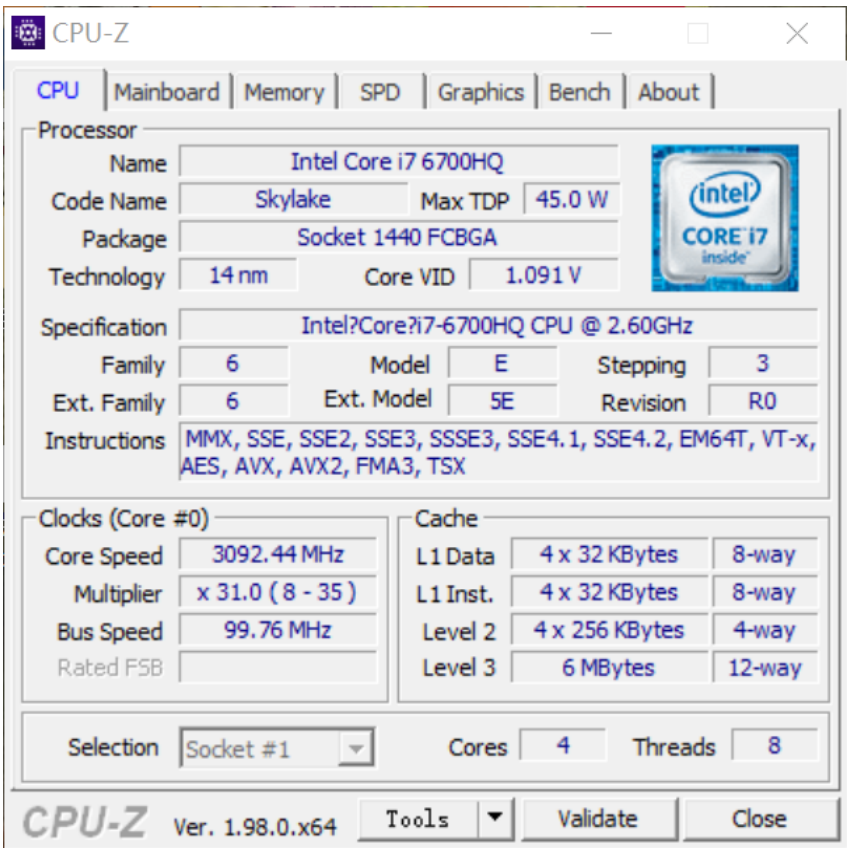

图 2: CPU parameters

```
Device Name : NVIDIA GeForce GTX 1060.
totalGlobalMem : 6144 MBytes----2147287040 Bytes.
sharedMemPerBlock : 49152.
regsPerBlock : 65536.
warpSize : 32.
memPitch : 2147483647.
maxThreadsPerBlock : 1024.
maxThreadsDim[0 - 2] : 1024 1024 64.
maxGridSize[0 - 2] : 2147483647 65535 65535.
totalConstMem : 65536.
major.minor : 6.1.
clockRate : 1670500.
textureAlignment : 512.
deviceOverlap : 1.
multiProcessorCount : 10.
```

图 3: GPU parameters

2 总体设计

针对矩阵乘法的优化策略大致分为两类，一类基于算法优化，一类基于软件工程优化。基于算法的优化，例如 Strassen 在 1969 年提出复杂度为 $\mathcal{O}(N^{2^{\log_2 7}})$ 的分块算法。这类算法一般不好实现且具有一定的使用场景限制。基于软件工程的优化，目标是充分发挥处理器的计算能力，例如提高处理器的缓存命中率来提升效率。本项目主要的内容包含通用矩阵乘法在 CPU 与 GPU 两部分的优化措施，核心思想都是提高高速缓存的利用率。

2.1 CPU

为了简化测试代码，我实现了一个简单的矩阵类 Matrix，实现了一些简单的操作，例如转置，随机初始化，单位化等。并且我实现了一些基本运算，例如加法、赋值、索引等。其中我参考实现了多个矩阵乘法算法。从 Gold 算法到最后的分块算法，在计算两个 1024×1024 矩阵的结果时，效率提升达到 14 倍左右。

Gold 算法是无 Open MP 优化的三层嵌套算法，我实现它用于对比误差与效率。我的初始方案是按照矩阵乘法定义实现三层 for 循环嵌套。然后添加 Open MP 编译指令实现并行化。

```
Matrix Gold_with_omp(Matrix a,Matrix b)
{
    if(a.columns()!=b.rows())
    {
        printf("The dimension does not match, nothing happened.");
        return a;
    }
    Matrix c(a.rows(),b.columns(),0);

    #pragma omp parallel for schedule(dynamic)
    for(int i=0;i<a.rows();i++)
        for(int j=0;j<b.columns();j++)
          for (int k=0;k<a.columns();k++)
                c[i][j]+=a(i,k)*b(k,j);
    return c;
}
```

在初始方案中，每个线程执行属于一个 k 循环，即，不断从 A 的一行与 B 的一列中读取数据并计算结果加到 C 的单个元素上。每个线程对 A,B 都访问了 K 次，对 C 访问且写入了 K 次。在这之中，对 C 的不断访问与写入是可以避免的，即在寄存器中新建临时变量盛放临时结果，最后通过一次写入即可。除此之外，初始方案还存在两个缺陷：

1. 程序在读取 A 的一行时是连续的，但在读取 B 的一列时是非连续的，因此每个线程需要从内存中不断重新读取数据，降低了缓存的命中率；
2. 不同的线程可能会共享同一行或同一列的数据，多个线程之间如果不做耦合，则会导致某一行数据或某一列数据被反复加载到缓存中，既降低了缓存命中率，也做了许多无效的加载；

针对这两个缺陷我们分别使用两种优化措施。针对第一个缺陷，可以通过更改矩阵 B 的内存分布来解决，具体来说，是将 B 进行一次转置后，将列改变为行在进行具体计算：

```
Matrix Gold_with_transpose(Matrix a, Matrix b)
{
    if(a.columns()!=b.columns())
    {
        printf("The dimension does not match, nothing happened.");
        return a;
    }
```

```
    b.Transpose();
    Matrix c(a.rows(),b.rows(),0);

    #pragma omp parallel for schedule(dynamic)
    for(int i=0;i<a.rows();i++)
        for(int j=0;j<b.rows();j++)
        {
            double temp_c=0.0;
            for (int k=0;k<a.columns();k++)
            {
                temp_c+=a(i,k)*b(j,k);
            }
            c(i,j)=temp_c;
        }

    return c;
}
```

针对第二个缺陷，我可以一次性读取矩阵 A 的多行或者矩阵 B 的多列数据进行计算，让每个线程计算矩阵 C 的多个元素，从而提高缓存的命中率。

```
Matrix Gold_double_packing(Matrix a, Matrix b)
{
    if(a.columns()!=b.rows())
    {
        printf("The dimension does not match, nothing happened.");
        return a;
    }
    Matrix c(a.rows(),b.columns(),0);

    int i=0,j=0,k=0;
    #pragma omp parallel for schedule(dynamic)
    for(int i=0;i<a.rows();i+=4)
        for(int j=0;j<b.columns();j+=4)
        {
            double temp_c[4][4]={{0}};
            for (int k=0;k<a.columns();k++)
            {
                temp_c[0][0..3]+=a(i+0,k)*b(k,j+0..3);
                temp_c[1][0..3]+=a(i+1,k)*b(k,j+0..3);
                temp_c[2][0..3]+=a(i+2,k)*b(k,j+0..3);
                temp_c[3][0..3]+=a(i+3,k)*b(k,j+0..3);
            }
            c(i+0,j+0..3)=temp_c[0][0..3];
            c(i+1,j+0..3)=temp_c[1][0..3];
            c(i+2,j+0..3)=temp_c[2][0..3];
            c(i+3,j+0..3)=temp_c[3][0..3];
        }
```

```
    return c;
}
```

尽管读取多行或多列已经能够起到加速作用，但对于大型矩阵，例如 1024×1024 矩阵，一次加载多行与多列可能导致矩阵数据大小超出缓存区大小，降低了缓存命中率。更好的做法是使用分块操作，将每个矩阵分为多个固定大小的区块加载到缓存中，从而提高缓存命中率。

```
Matrix Tiling(Matrix a, Matrix b)
{
    if(a.columns()!=b.rows())
    {
        printf("doublehe dimension does not match, nothing happened.");
        return a;
    }
    Matrix c(a.rows(),b.columns(),0);

    int tile_size=4;
    // int i=0,j=0,k=0;
    #pragma omp parallel for
    for(int i=0;i<a.rows();i+=tile_size)
        for(int j=0;j<b.columns();j+=tile_size)
        {
            double na[4][4]={{0}};
            double nb[4][4]={{0}};
            double nc[4][4]={{0}};
            for(int k=0;k<a.columns();k+=tile_size)
            {
                //load submatrix of a into caches
                na[0][0..3]=a(i+0,k+0..3);
                na[1][0..3]=a(i+1,k+0..3);
                na[2][0..3]=a(i+2,k+0..3);
                na[3][0..3]=a(i+3,k+0..3);

                //load submatrix of b into caches
                nb[0][0..3]=b(k+0,j+0..3);
                nb[1][0..3]=b(k+1,j+0..3);
                nb[2][0..3]=b(k+2,j+0..3);
                nb[3][0..3]=b(k+3,j+0..3);

                //matrix multiplication of sub-matrices
                nc[0][0..3]+=na[0][0]*nb[0..3][0]+na[0][1]*nb[1][0..3]+na
                    [0][2]*nb[2][0..3]+na[0][3]*nb[3][0..3];
                nc[1][0..3]+=na[1][0]*nb[0][0..3]+na[1][1]*nb[1][0..3]+na
                    [1][2]*nb[2][0..3]+na[1][3]*nb[3][0..3];
                nc[2][0..3]+=na[2][0]*nb[0][0..3]+na[2][1]*nb[1][0..3]+na
                    [2][2]*nb[2][0..3]+na[2][3]*nb[3][0..3];
                nc[3][0..3]+=na[3][0]*nb[0][0..3]+na[3][1]*nb[1][0..3]+na
```

```
                    [3][2]*nb[2][0..3]+na[3][3]*nb[3][0..3];

            }
            c(i+0,j+0..3)=nc[0][0..3];
            c(i+1,j+0..3)=nc[1][0..3];
            c(i+2,j+0..3)=nc[2][0..3];
            c(i+3,j+0..3)=nc[3][0..3];
        }
        return c;
}
```

2.2 GPU

GPU 部分的实现与优化思路与 CPU 部分其实非常相似，只不过把相同的优化技巧在 GPU 上实现一边并对比整体的效率与误差。

首先我实现了一个初始方案，该方案利用一个 Grid 计算矩阵乘法，其中每个 Block 大小为 16×16，每个线程从 Global memory 中加载数据且计算矩阵的一个元素。

```
__global__ void matrixMul_naive(float* c, float* a, float* b, int m, int n, int
     k)
{
  // cal the index
  int tx=blockIdx.x*blockDim.x+threadIdx.x;
  int ty=blockIdx.y*blockDim.y+threadIdx.y;

  float temp_c=0.0;
  if(tx<n && ty<m)
  {
    for(int i=0;i<k;i++)
    {
      temp_c+=a[ty*k+i]*b[i*k+tx];
    }
  }
    c[ty*n+tx]=temp_c;
}
```

该方案在计算两个 1024×1024 矩阵相乘时，只消耗大约 160ms，比 CPU 分块优化还要快，由此可见 GPU 在大规模科学计算中的优势。

依据 CPU 的优化思路，接下来应该利用共享内存，存储多行或者多列进行计算，每个线程计算多个数据。但这样只会导致重复数据不断被读入读出共享内存，所以我直接实现对应 CPU 分块的 GPU 分块算法。

GPU 分块算法主要思路是：将 A、B 切割成多个子矩阵，线程将子矩阵加载到共享内存中，计算子矩阵的乘法，经过多次循环，求得 C。算法执行过程如图4所示。

每个 Block 对应矩阵 A 的若干行，矩阵 B 的若干列。每次执行中，每个线程从矩阵 A 与 B 中取子矩阵的一个对应元素放到共享内存中。然后所有线程同步一次，确保取值完成。接着所

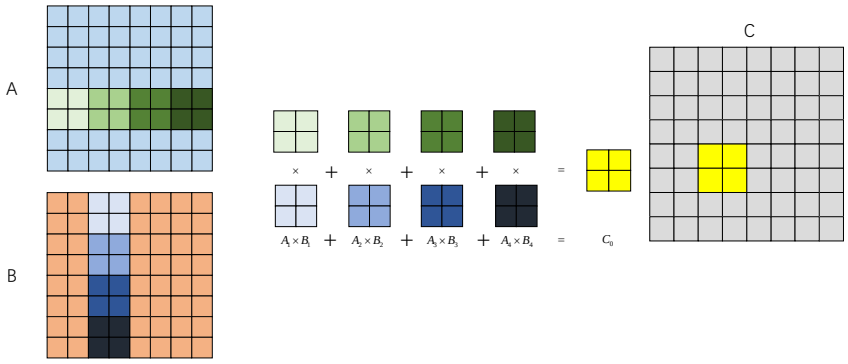

图 4: 分块

有线程计算子矩阵之间的乘法并加到中间变量上。通过循环，完成所有子矩阵的乘法得到最终结果，写到 C 的全局内存中。具体代码如下：

```
__global__ void matrixMul_tiling(float* c, float* a, float* b, int m, int n,int
    k)
{
  // Block index and thread index
  int by=blockIdx.y;
  int bx=blockIdx.x;
  int tx=threadIdx.x;
  int ty=threadIdx.y;
  int iy=BLOCK_SIZE*by+ty;
  int ix=BLOCK_SIZE*bx+tx;

  // shared memory to hold sub-matrix of A and B
  __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
  __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
  // index and numbers
  int aBegin = k * BLOCK_SIZE * by;
  int aEnd = aBegin + k - 1;
  int aStep = BLOCK_SIZE;
  int bBegin = BLOCK_SIZE * bx;
  int bStep = BLOCK_SIZE * n;

  // temp sum
  float Csub = 0;
  if(ix < n && iy < m)
  {
    for (int i = aBegin, j = bBegin;
      i <= aEnd;
      i += aStep, j += bStep)
    {
      // Load sub-matrix into shared memory
      As[ty][tx] = a[i + k * ty + tx];
      Bs[ty][tx] = b[j + n * ty + tx];
```

```
      // Synchronize to make sure the matrices are loaded
      __syncthreads();

      // multiply two sub-matrices
      // Note that here exists bank conflicts and we should try to avoid it
      for (int l = 0; l < BLOCK_SIZE; l++)
        Csub += As[ty][l] * Bs[l][tx];

      // sync to make sure all temp results are correctly computed
      __syncthreads();
    }
    // Write results back to matrix C
    int cBegin = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    c[cBegin + n * ty + tx] = Csub;
  }
}
```

类似的，分块算法没有处理矩阵 A 与 B 的内存分布，存在不连续取值的现象，解决方法在 kernel 外矩阵 B 进行一次转置。

```
__global__ void matrixMul_coalescing(float* C, float*A, float*B, int wA, int wC
    )
{
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int i = by * blockDim.y + ty;
  int j = bx * blockDim.x + tx;

  float sum = 0.0;
  for (int k = 0; k < wA; k++)
  {
    sum += A[i*wA + k] * B[j*wA + k];
  }
  C[i*wC + j] = sum;
}
```

3 主体安排

在这一节，我主要展示优化算法的结果，包括错误率与效率对比。

3.1 实验设置

对于不同的优化算法，我主要对比了不同维度大小下，矩阵相乘的计算效率与错误率。每个实验值重复十次取平均值。

3.2 CPU

首先，固定矩阵大小为两个 1024×1024 时，CPU 上不同优化算法的效率对比如图5

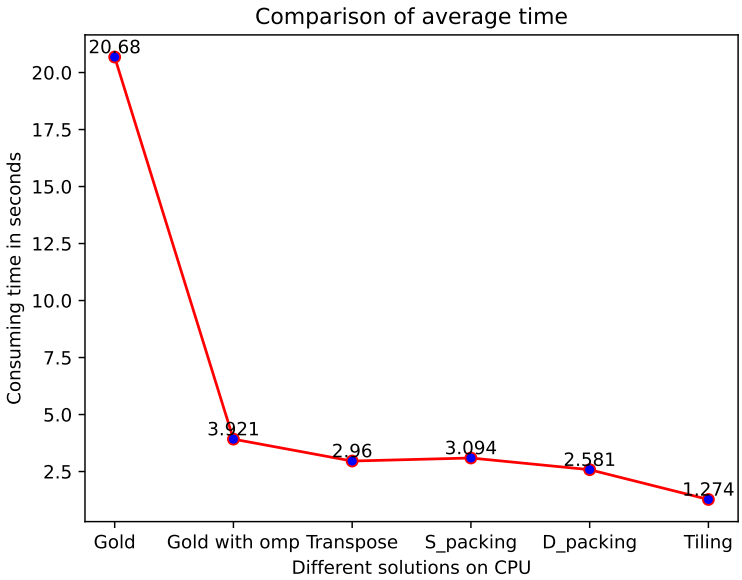

图 5: 平均时间对比

其中 Gold 算法是不经并行优化的三层嵌套循环，耗时在 20s 左右。在加上 Open MP 优化指令后，Gold with omp 算法能立即得到大约五倍的优化效果。在对矩阵进行内存分布的优化后，Transpose 效率有进一步的提升，但幅度不大。Single packing（S_packing）是一次性读取多列时的优化，没有内存优化时，其效率虽然相比 Gold with omp 有提升，但比不过 Transpose。当同时读取多行多列时，Double packing（D_packing）才有了进一步的效率提升。由此可知，在没有对内存进行优化时，多次读取多列的做法能带来的性能提升非常有限。

最后，当采取 4×4 大小的分块措施时，Tiling 算法能在 Double packing 的基础上获得大约两倍的效率提升。

接下来对比实验引起的误差，其中矩阵中的每个值是在 1 到 100 之间随机选取。错误的计算公式如下

$$error(solution, gold) = \sum_{i=1}^{M}\sum_{j=1}^{N} |c_{i,j}^{gold} - c_{i,j}^{solution}| \tag{1}$$

其中 $solution \in$ {Gold,Gold with omp, Transpose, S_packing,D_packing,Tiling}, 为了保证实验可复现，每次的随机种子均相同。错误率如图6所示。

从图中可以看出，四个优化算法与 Gold 的差距均为 0，只有在分块时才会引入一定的误差。但相对于矩阵规模与数值范围，这些误差完全是可以接受的。

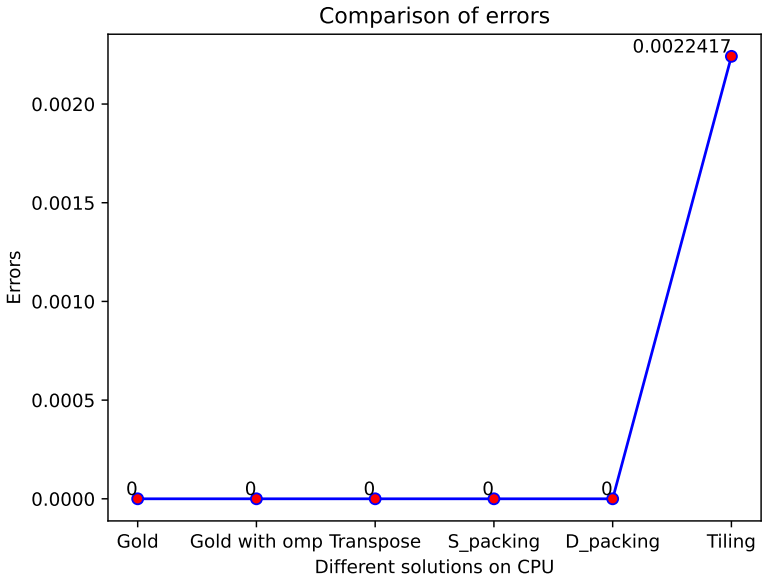

图 6: 平均错误对比

接下来继续分析不同维度大小下，各个算法的效率对比。我设置了五个不等的矩阵规模 $\{512, 1024, 1536, 2048, 2560\}$，得到各个算法的时间对比如图7

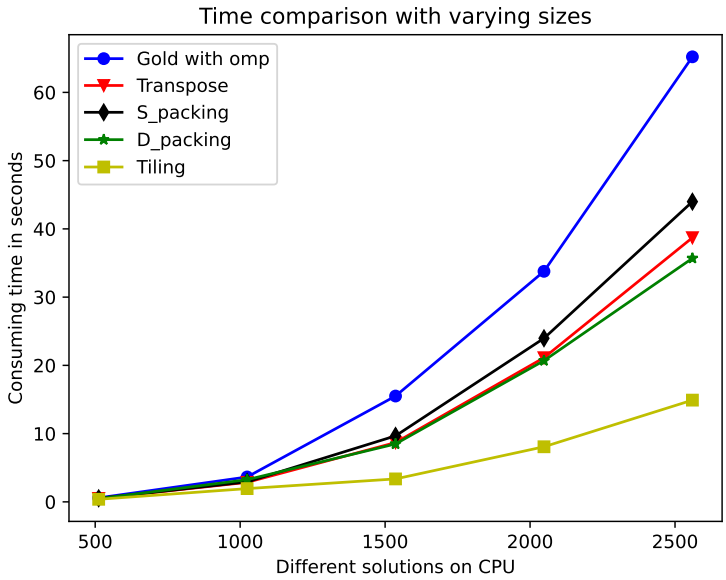

图 7: 不同规模平均时间对比

从图中可以看出，矩阵维度越高，分块的算法效率提升越明显。不仅如此，分块算法的增长速度也明显低于其他算法。因此针对 CPU 的通用矩阵乘法，分块算法相对于其他所有算法具有明显的优势。

3.3 GPU

首先，固定矩阵大小为 1024×1024，Block 大小为 16×16 时，GPU 上不同优化算法的效率对比如图8

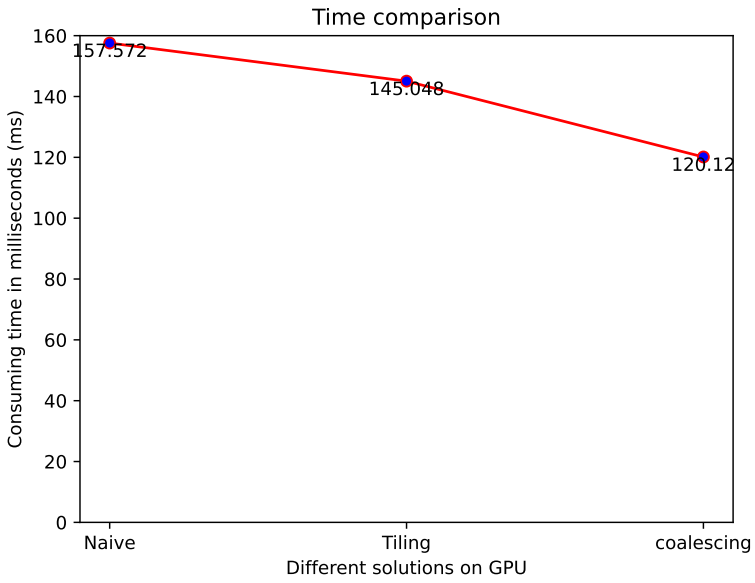

图 8: 平均时间对比

从图中可以看出，分块算法相对初始方案有一定的效率提升，内存布局优化比分块算法则更为高效。但是两者的优化幅度都有限，没有达到预期的效果。在查阅了很多资料后，确定不是代码问题，但优化的效果和已有的资料严重不符，我猜测可能是 bank_conflict 的原因，但是没有找到具体解决的方法。

接下来我将对比不同矩阵规模下，不同算法的计算效率，矩阵规模设置与 CPU 相同。BLOCK_SIZE 一直维持 16×16。实验结果如图9

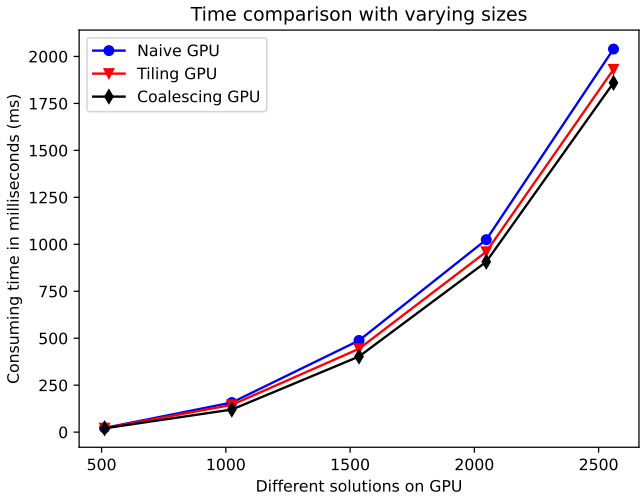

图 9: 不同规模平均时间对比

从图中可以看出，三个算法间的效率大致相同，增长模式近似，算法的优化效果不明显。

3.4 实验结论

通过本次大作业我获得了如下几个结论：

1. 针对 CPU 优化算法，打包读取的方式并不能带来明显的效率提升，对列的打包读取甚至会产生一些负面作用。但分块计算可以在忍受一定误差的情况下，实现明显的效率提升；
2. 针对 GPU 优化算法，针对如矩阵乘法的科学计算问题，GPU 具有明显的计算优势。GPU 在不做任何优化条件下，其计算性能都要高于 CPU 算法。本次大作业实现的两个 GPU 优化算法的效果都不明显，需要进一步的分析与观察。

4 实验困难

实验过程中，主要的困难在于 GPU 代码的编写与调试。我尝试过在 Cuda 5.1 版本之前曾使用过的 cutil.h 头文件尝试消除 bank_conflict 的影响，但是实际并没有区别。后来我尝试使用了不同的索引方式，让不同的线程索引不同的值，但是也没取得效果。查阅网上的资料，大都只笼统的谈如何消除 bank_conflict 的方法，没有具体结合实际代码的实现。因此针对这个问题，我暂时不知道如何处理。接下来的想法是查阅英伟达《CUDA C++ PROGRAMMING GUIDE》来寻找解决方案。