# 计算机系统 lab2 -- CUDA矩阵乘法 实验报告

22级JOHN班 涂文良

# 实验过程简介

在本次实验中，我学习了CUDA，完成了朴素并行矩阵乘法和利用 shared memory 优化的矩阵乘法的实现，并通过了正确性测试，具体代码在 simple_gemm.cu 与 share_gemm.cu中。 随后， 我编写了一个脚本， 用于自动化输出所有参数情况的实验结果，该脚本即为服务器文件夹 tuwenliang 中的 script 文件，利用了 sed 进行宏的文本替换、grep 进行信息过滤。 最后，我利用 python 对实验数据进行了分析并绘制了曲线图。

# 实验思路

## 优化前实验思路

首先，在朴素思路实现矩阵乘法中，我使用每个 thread 独立地算出一个元素。我的 grid - block - thread 布局大致为：

```
Grid (M/THREAD_PRE_BLOCK, N/THREAD_PRE_BLOCK)
+-----------------------------------------+
| Block (0,0) | Block (1,0) | ... | Block (M/THREAD_PRE_BLOCK - 1, 0) |
+-----------------------------------------+
| Block (0,1) | Block (1,1) | ... | Block (M/THREAD_PRE_BLOCK - 1, 1) |
+-----------------------------------------+
| ...         | ...         | ... | ...                           |
+-----------------------------------------+
| Block (0,N/THREAD_PRE_BLOCK - 1) | ...    | Block (M/THREAD_PRE_BLOCK - 1,
N/THREAD_PRE_BLOCK - 1) |
+-----------------------------------------+

Each Block (THREAD_PRE_BLOCK, THREAD_PRE_BLOCK)
+--------------------+
| Thread (0,0) | ...  |
```

```
+--------------------+
| ...          | ... |
+--------------------+
| Thread (THREAD_PRE_BLOCK - 1, 0) | ... |
+--------------------+
| ...          | ... |
+--------------------+
| Thread (0, THREAD_PRE_BLOCK - 1) | ... |
+--------------------+
| ...          | ... |
+--------------------+
| Thread (THREAD_PRE_BLOCK - 1, THREAD_PRE_BLOCK - 1) |
+--------------------+
```

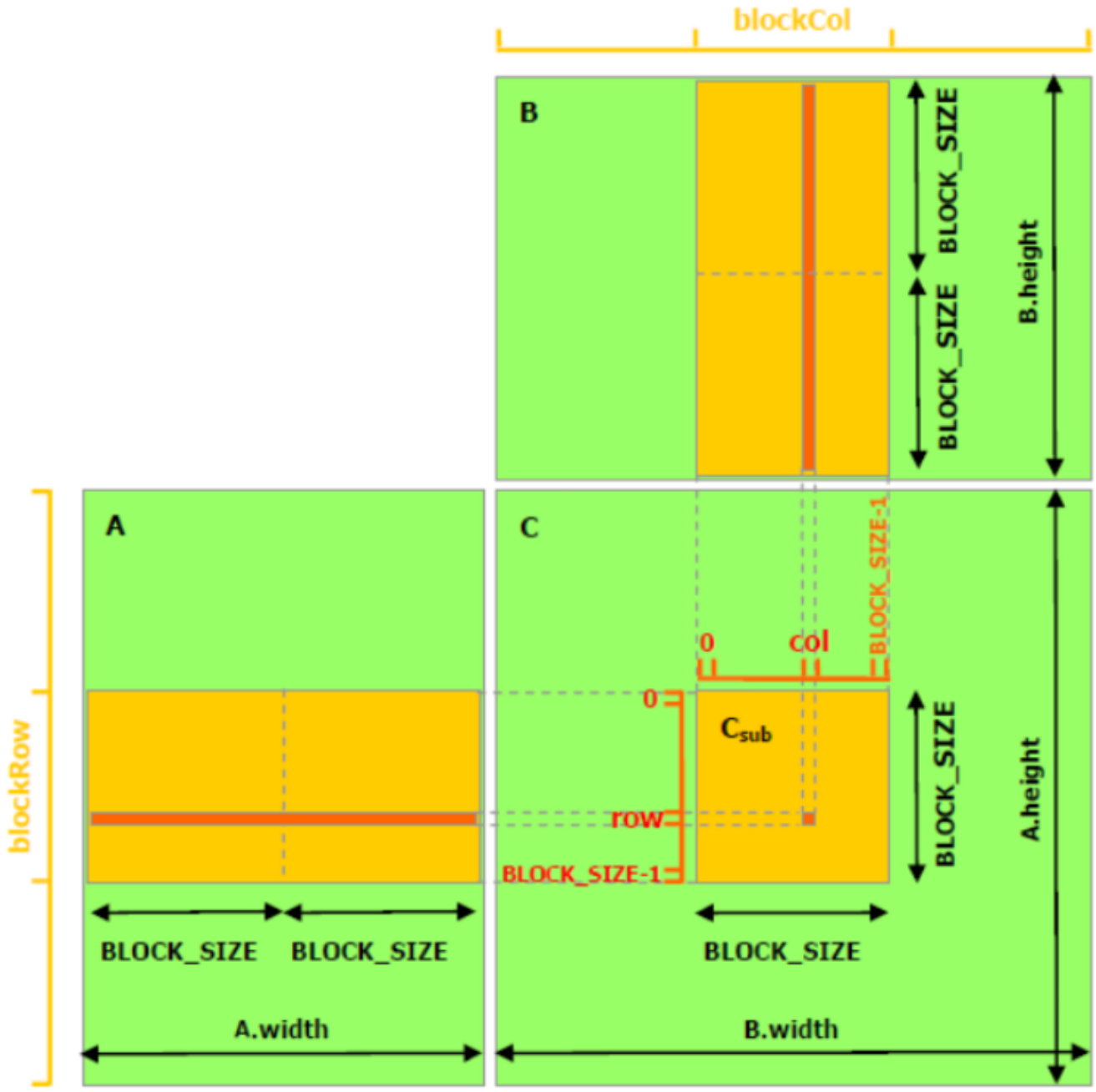
我的核心函数 gemm 即采取朴素思路， 用矩阵乘法的定义进行计算，代码如下：

```
__global__ void gemm(int* a, int* b, int* c) {
  // Calculate global thread indices
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  if (row < M && col < N) {
      int result = 0;
      // Perform matrix multiplication for the given element (row, col)
      for (int k = 0; k < K; ++k) {
          result += a[row * K + k] * b[k * N + col];
      }
      c[row * N + col] = result;
  }
}
```

## 性能瓶颈与优化思路

上述朴素并行算法， 虽然利用了 GPU 的并行计算特点，但是其中冗余地进行了大量访存。比如， 在计算 C 矩阵的同一行时，会用到 A 矩阵的同一行， 但上述朴素思路每次都需要从 GPU 的内存中读取 A 矩阵这一行的元素， 产生了大量的资源和时间浪

费。 实际上， 我们可以将这一行的元素存入 shared memory 中， 供一个 block 的 thread 共同使用， 避免了访存带来的性能削弱。




如图所示的示例中， 计算 C 矩阵的这个 block 需要 A 矩阵的两个 block 和 B 矩阵的两个 block。 实际上这也是使用了分块矩阵的思想。 那我们可以这样做：负责计算这部分 block 的那些 thread, 仍然每个 thread 负责计算一个元素。但是这里我们用一个循环去计算， 循环内部， 我们将 A 矩阵的一个 block 和 B 矩阵的一个 block 存入 shared memory 中， 然后每个 thread 将自己负责的那一行和那一列的元素乘积计算出， 每次累加到一个临时变量上。 那么经过这个循环之后， 每个 thread 都成功计算了自己负责的元素， 存入了一个临时变量中。 此时， 我再将该元素写入即可。

具体到代码上， 我的 gemm 函数核心组成部分如下， 具体思路写在代码注释中：（代码有部分省略）

```
__global__ void gemm(Matrix a, Matrix b, Matrix c) {
  //compute the bank index
  int B_Row = blockIdx.y;
  int B_Col = blockIdx.x;

  //define the target this thread is currently on
  Matrix Sub_of_C;
  ...//initialization for this matrix

  //result is a temporary variable which represent the result for the element
  //at (row, col)
  int result = 0;
  int row = threadIdx.y;
  int col = threadIdx.x;

  //Use a for loop to do the computing
  //looping over the blocks
  for(int k = 0; k < N / THREAD_PER_BLOCK; k++)
  {
    //define the sub matrix of A and B
    Matrix Sub_of_A, Sub_of_B;
    ...//initialization for these two matrices

    //Pushing elements into the shared memory
    __shared__ int Shared_sub_matrix_A[THREAD_PER_BLOCK][THREAD_PER_BLOCK];
    __shared__ int Shared_sub_matrix_B[THREAD_PER_BLOCK][THREAD_PER_BLOCK];

    //In every cycle, each thread is responsible for pushing one element
    Shared_sub_matrix_A[row][col] = Sub_of_A.Mat[row * a.stride + col];
    Shared_sub_matrix_B[row][col] = Sub_of_B.Mat[row * b.stride + col];
    //Syncranize, waiting for them to complete filling the shared memory
    __syncthreads();

    //Compute the product and add them to the temporary variable
    for(int r = 0; r < THREAD_PER_BLOCK; r++)
    {
      result += Shared_sub_matrix_A[row][r] * Shared_sub_matrix_B[r][col];
```

```
        }
        //syncranize, waiting for them to complete computing
        __syncthreads();

    }
    //Now assign the temporary element to Matrix C
    Sub_of_C.Mat[row * Sub_of_C.stride + col] = result;
}
```

因此， 通过利用 shared memory， 我实现了减少访存、 提高代码性能的任务。

# 实验数据与曲线图

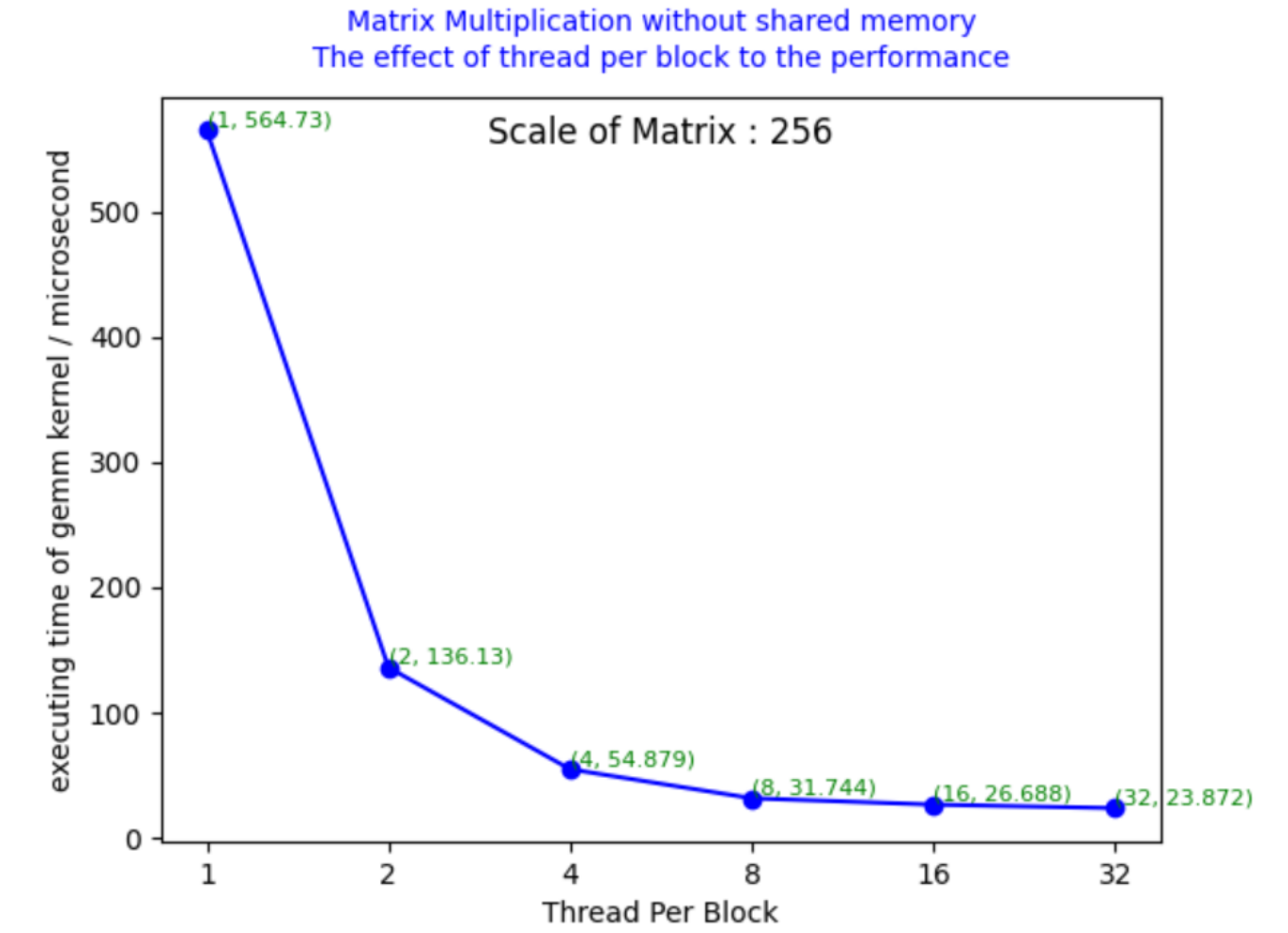
## 优化前实验数据与实验数据与曲线图

没有使用 shared memory 进行优化时， 矩阵规模分别为 256， 512， 1024，4096 时， thread per block 与性能的关系表格如下所示：

1.矩阵规模为 256：

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
THREAD_PER_BLOCK is 1
 GPU activities:   88.77%  564.73us         1  564.73us  564.73us  564.73us  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 2
 GPU activities:   65.50%  136.13us         1  136.13us  136.13us  136.13us  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 4
 GPU activities:   43.74%  54.879us         1  54.879us  54.879us  54.879us  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 8
                   31.03%  31.744us         1  31.744us  31.744us  31.744us  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 16
                   27.33%  26.688us         1  26.688us  26.688us  26.688us  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 32
                   25.15%  23.872us         1  23.872us  23.872us  23.872us  gemm(int*, int*, int*)
```

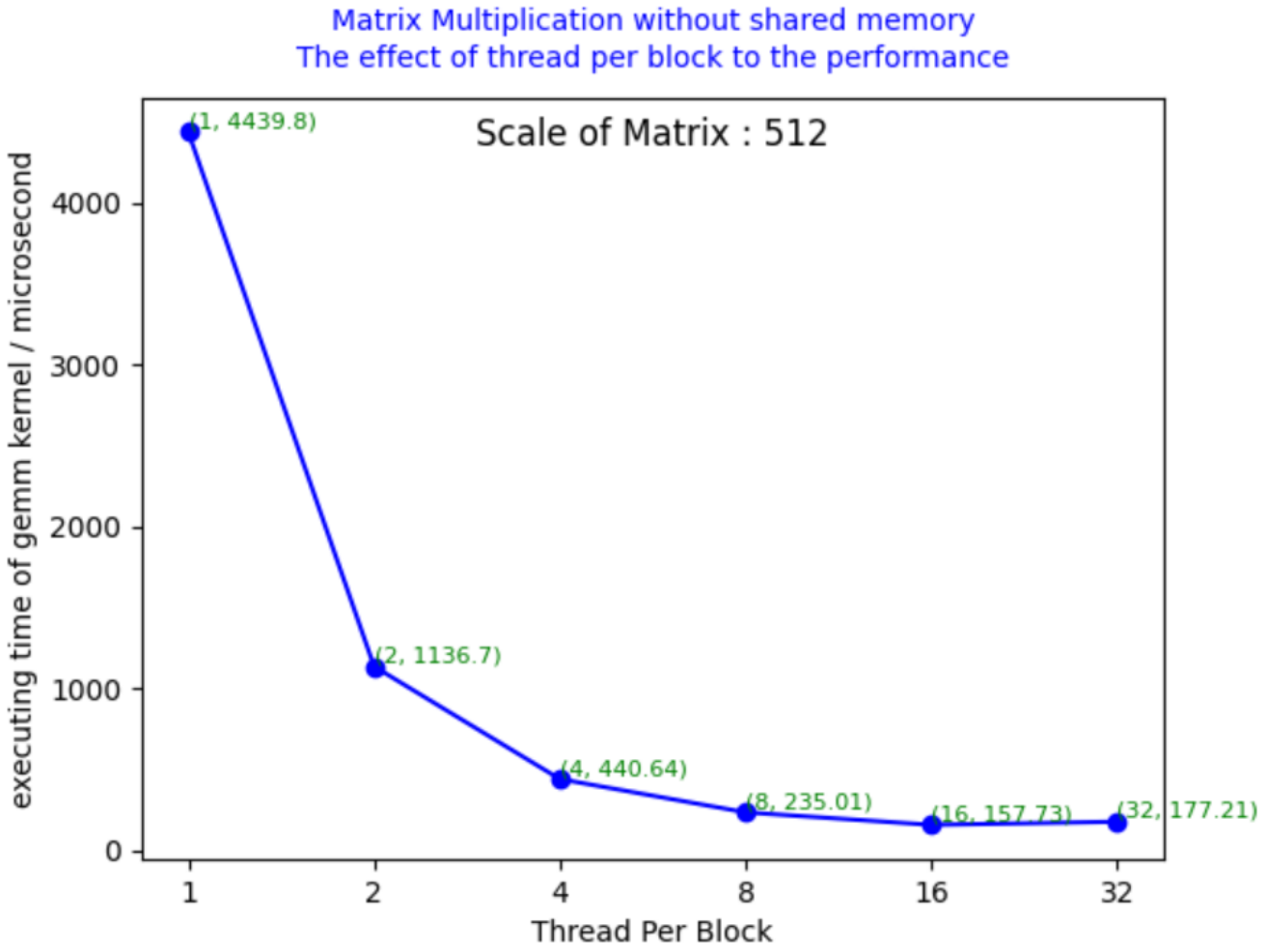
根据实验数据使用python作图如下：(python作图所用代码也在服务器文件夹tuwenliang中)




2.矩阵规模为 512：

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
THREAD_PER_BLOCK is 1
 GPU activities:   94.37%  4.4398ms         1  4.4398ms  4.4398ms  4.4398ms  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 2
 GPU activities:   81.19%  1.1367ms         1  1.1367ms  1.1367ms  1.1367ms  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 4
 GPU activities:   62.38%  440.64us         1  440.64us  440.64us  440.64us  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 8
 GPU activities:   47.12%  235.01us         1  235.01us  235.01us  235.01us  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 16
                   37.16%  157.73us         1  157.73us  157.73us  157.73us  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 32
 GPU activities:   40.01%  177.21us         1  177.21us  177.21us  177.21us  gemm(int*, int*, int*)
```

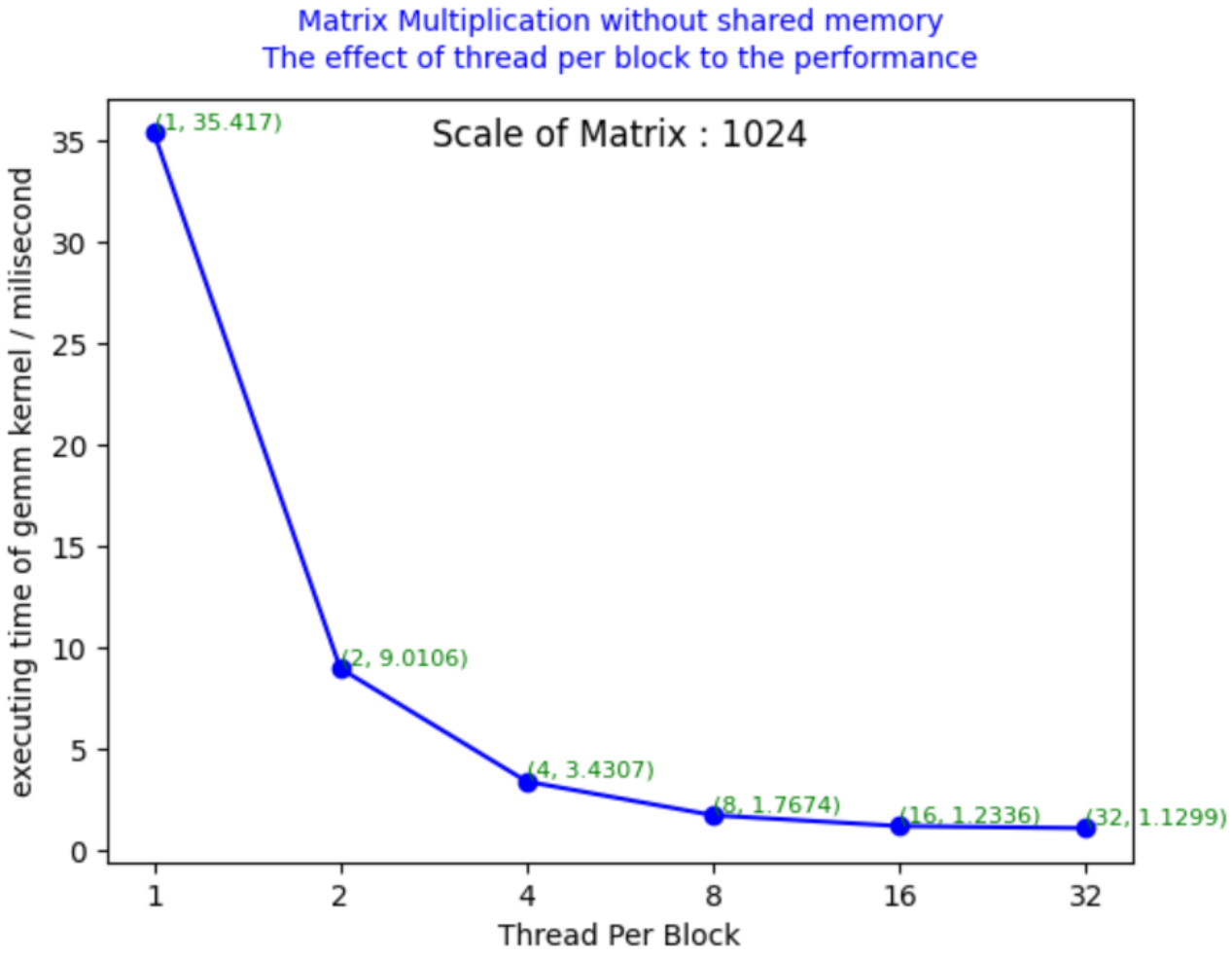
作图如下：




3.矩阵规模为 1024：

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
THREAD_PER_BLOCK is 1
 GPU activities:   91.18%  35.417ms         1  35.417ms  35.417ms  35.417ms  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 2
 GPU activities:   72.13%  9.0106ms         1  9.0106ms  9.0106ms  9.0106ms  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 4
 GPU activities:   50.56%  3.4307ms         1  3.4307ms  3.4307ms  3.4307ms  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 8
                   34.05%  1.7674ms         1  1.7674ms  1.7674ms  1.7674ms  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 16
                   26.61%  1.2336ms         1  1.2336ms  1.2336ms  1.2336ms  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 32
                   24.99%  1.1299ms         1  1.1299ms  1.1299ms  1.1299ms  gemm(int*, int*, int*)
```

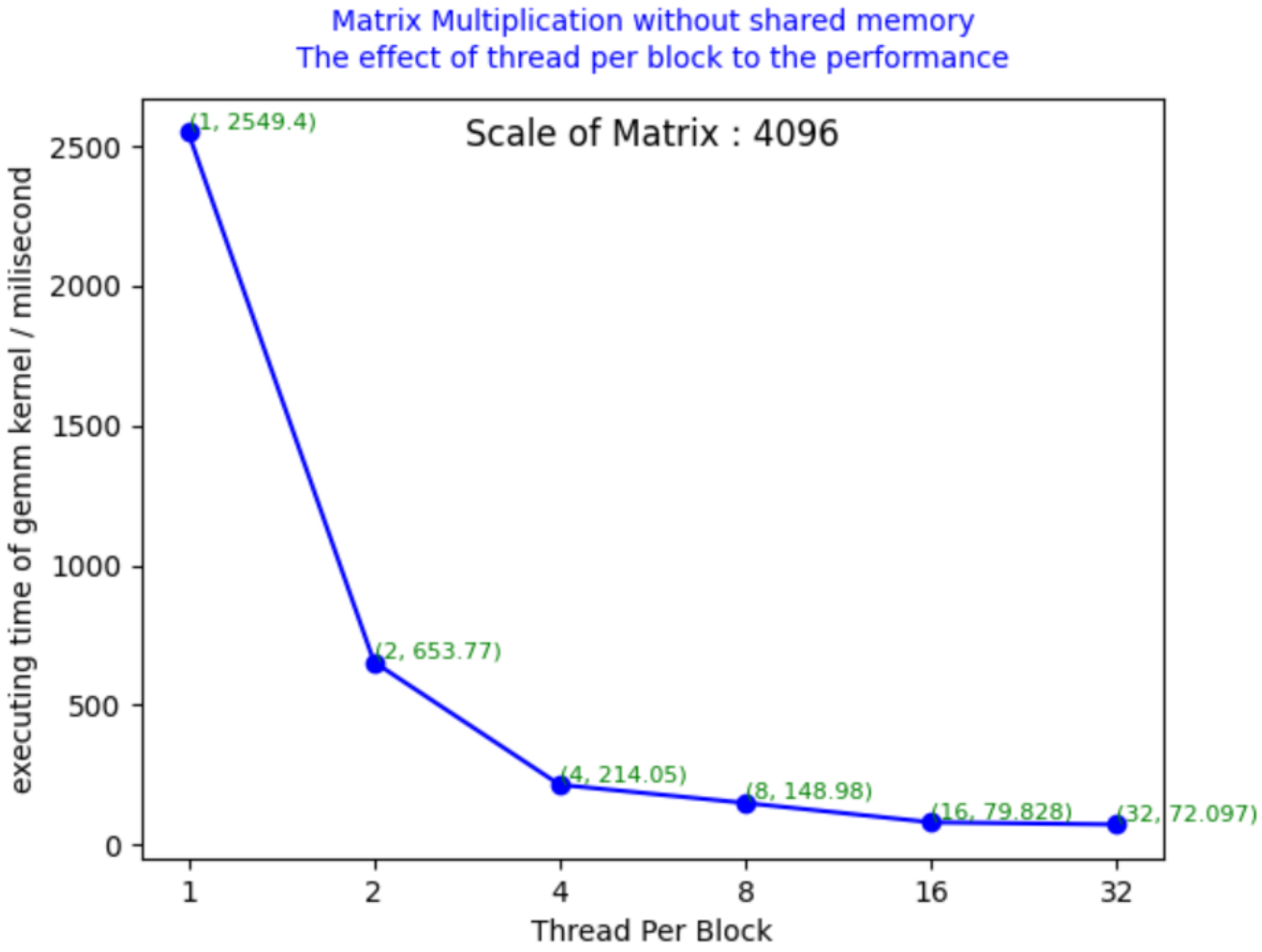
作图如下：



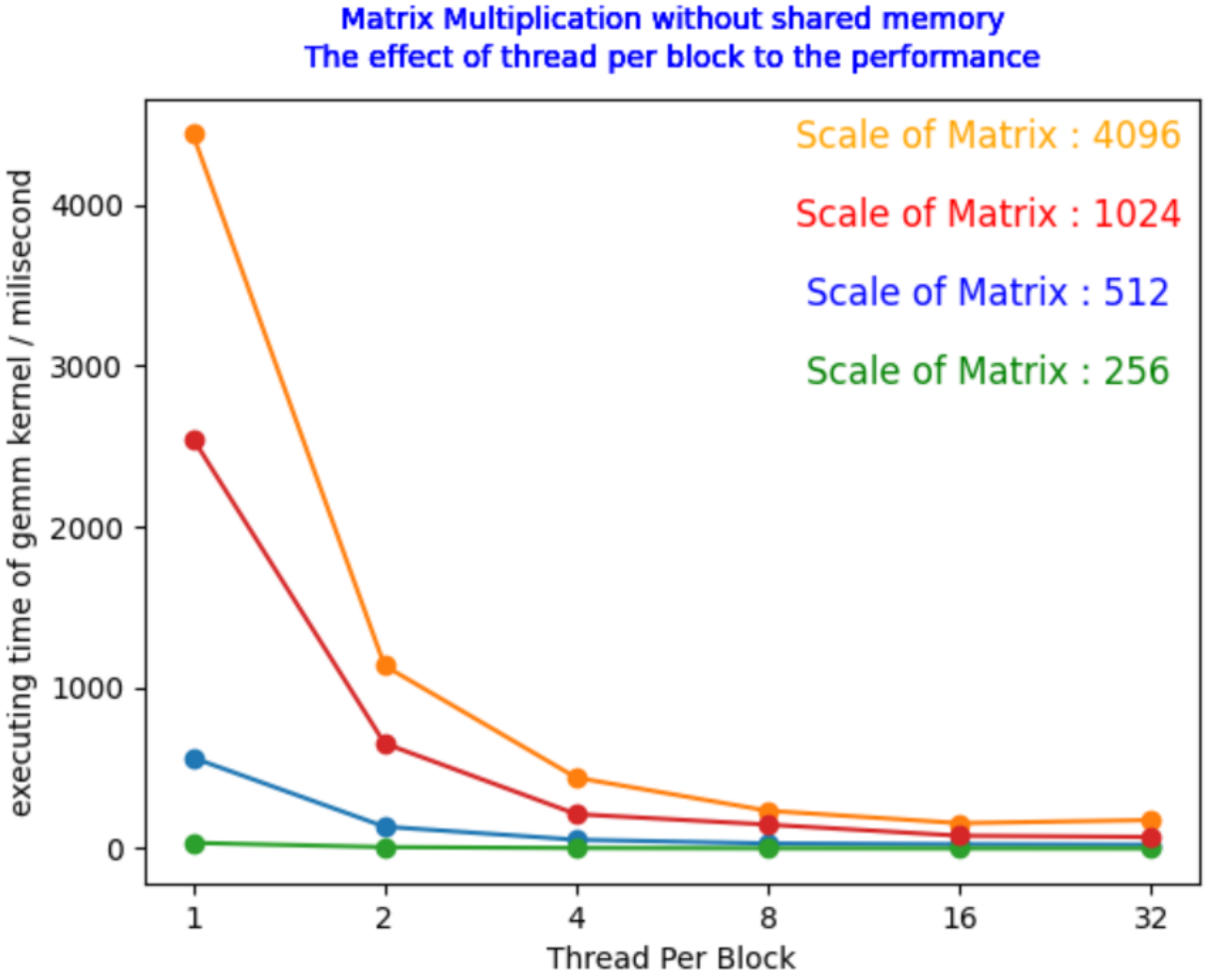

4.矩阵规模为 4096：

```
      Type  Time(%)      Time     Calls       Avg       Min       Max  Name
THREAD_PER_BLOCK is 1
 GPU activities:   97.06%  2.54944s         1  2.54944s  2.54944s  2.54944s  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 2
 GPU activities:   89.48%  653.77ms         1  653.77ms  653.77ms  653.77ms  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 4
 GPU activities:   73.63%  214.05ms         1  214.05ms  214.05ms  214.05ms  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 8
 GPU activities:   65.99%  148.98ms         1  148.98ms  148.98ms  148.98ms  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 16
 GPU activities:   50.84%  79.828ms         1  79.828ms  79.828ms  79.828ms  gemm(int*, int*, int*)
THREAD_PER_BLOCK is 32
 GPU activities:   48.45%  72.097ms         1  72.097ms  72.097ms  72.097ms  gemm(int*, int*, int*)
```

作图如下：




将四条曲线画在一个图中，即为：



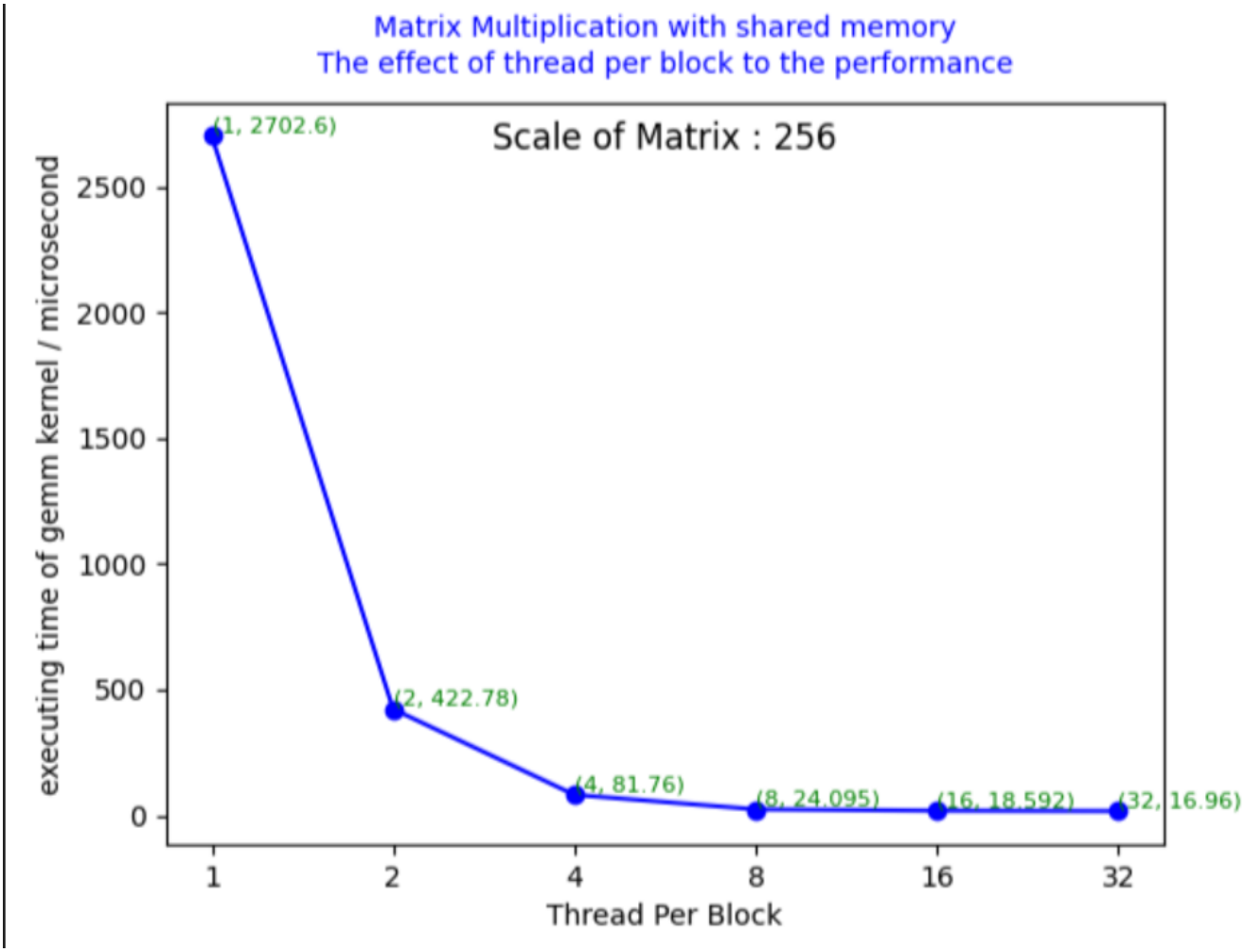

# 优化后实验数据与实验数据与曲线图

使用 shared memory 进行优化后， 矩阵规模分别为 256， 512， 1024，4096 时， thread per block 与性能的关系表格如下所示：

1.矩阵规模为 256：

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
THREAD_PER_BLOCK is 1
 GPU activities:   97.39%  2.7026ms         1  2.7026ms  2.7026ms  2.7026ms  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 2
 GPU activities:   85.07%  422.78us         1  422.78us  422.78us  422.78us  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 4
 GPU activities:   52.96%  81.760us         1  81.760us  81.760us  81.760us  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 8
                   25.42%  24.095us         1  24.095us  24.095us  24.095us  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 16
                   20.94%  18.592us         1  18.592us  18.592us  18.592us  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 32
                   19.15%  16.960us         1  16.960us  16.960us  16.960us  gemm(Matrix, Matrix, Matrix)
```

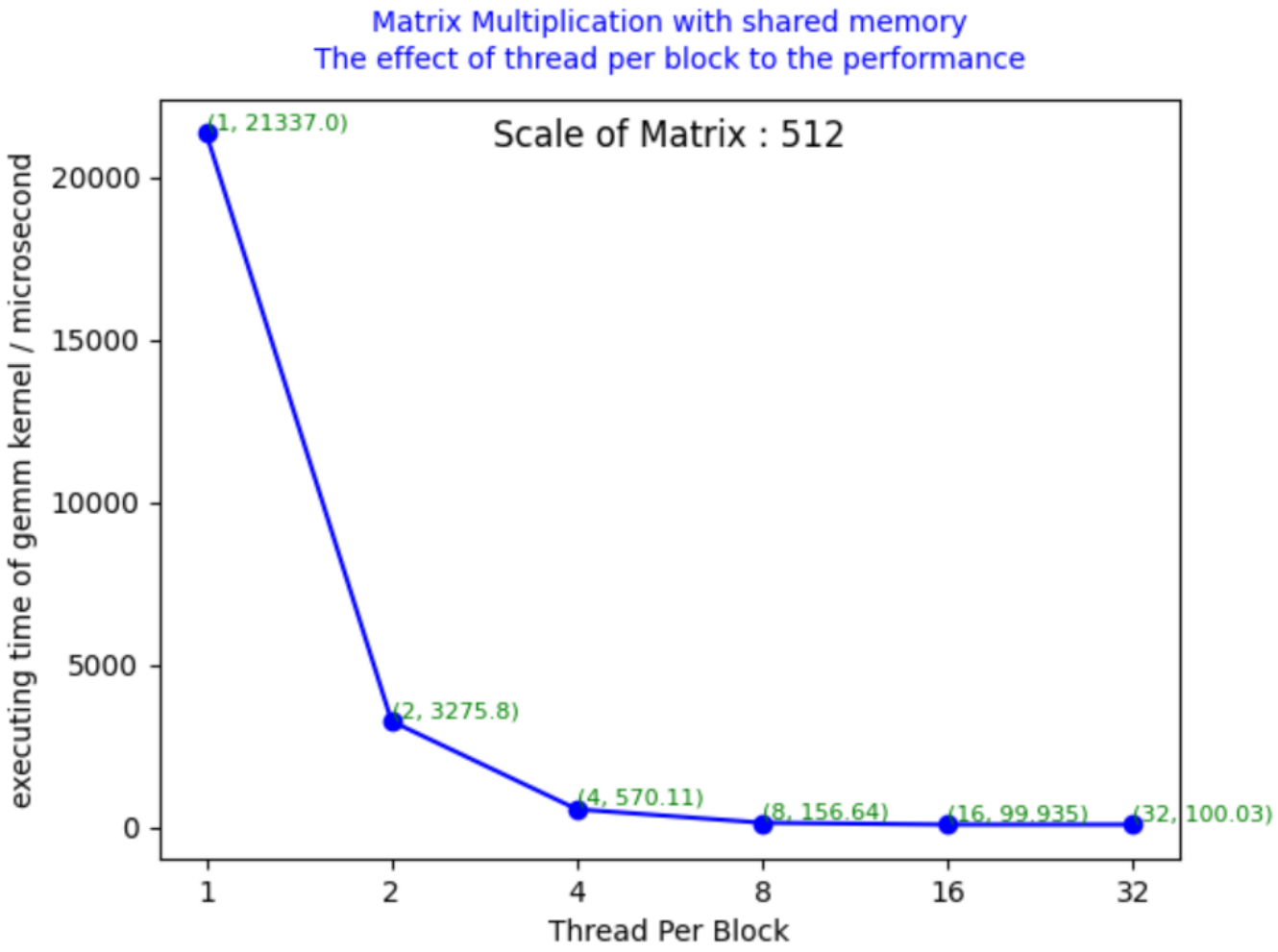
作图如下：




2.矩阵规模为 512：

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
THREAD_PER_BLOCK is 1
 GPU activities:   98.78%  21.337ms         1  21.337ms  21.337ms  21.337ms  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 2
 GPU activities:   92.56%  3.2758ms         1  3.2758ms  3.2758ms  3.2758ms  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 4
 GPU activities:   68.26%  570.11us         1  570.11us  570.11us  570.11us  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 8
                   37.07%  156.64us         1  156.64us  156.64us  156.64us  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 16
                   27.40%  99.935us         1  99.935us  99.935us  99.935us  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 32
                   27.46%  100.03us         1  100.03us  100.03us  100.03us  gemm(Matrix, Matrix, Matrix)
```

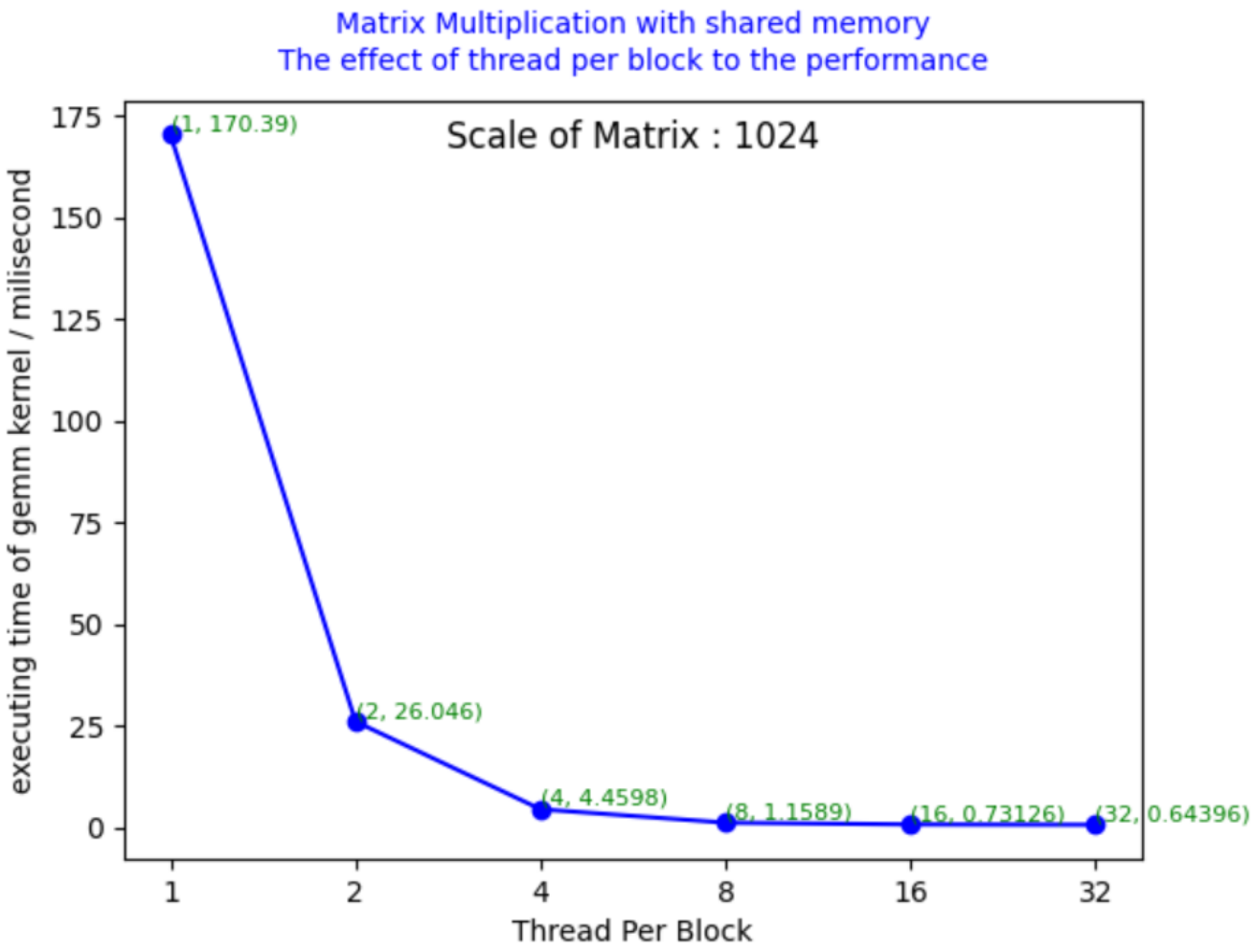
作图如下：




3.矩阵规模为 1024：

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
THREAD_PER_BLOCK is 1
 GPU activities:   98.05%  170.39ms         1  170.39ms  170.39ms  170.39ms  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 2
 GPU activities:   88.42%  26.046ms         1  26.046ms  26.046ms  26.046ms  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 4
 GPU activities:   56.94%  4.4598ms         1  4.4598ms  4.4598ms  4.4598ms  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 8
                   25.18%  1.1589ms         1  1.1589ms  1.1589ms  1.1589ms  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 16
                   17.85%  731.26us         1  731.26us  731.26us  731.26us  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 32
                   16.16%  643.96us         1  643.96us  643.96us  643.96us  gemm(Matrix, Matrix, Matrix)
```

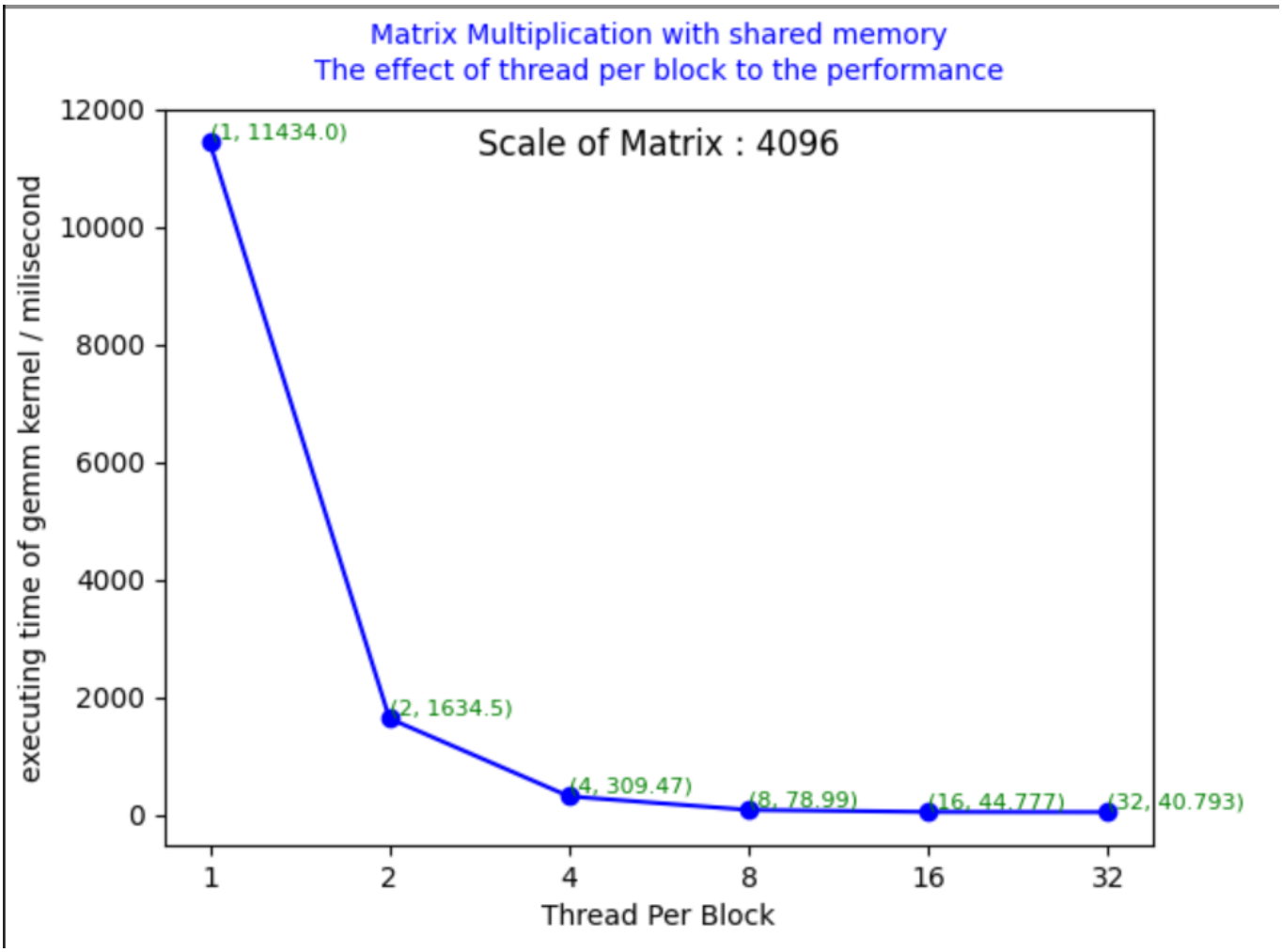
作图如下：



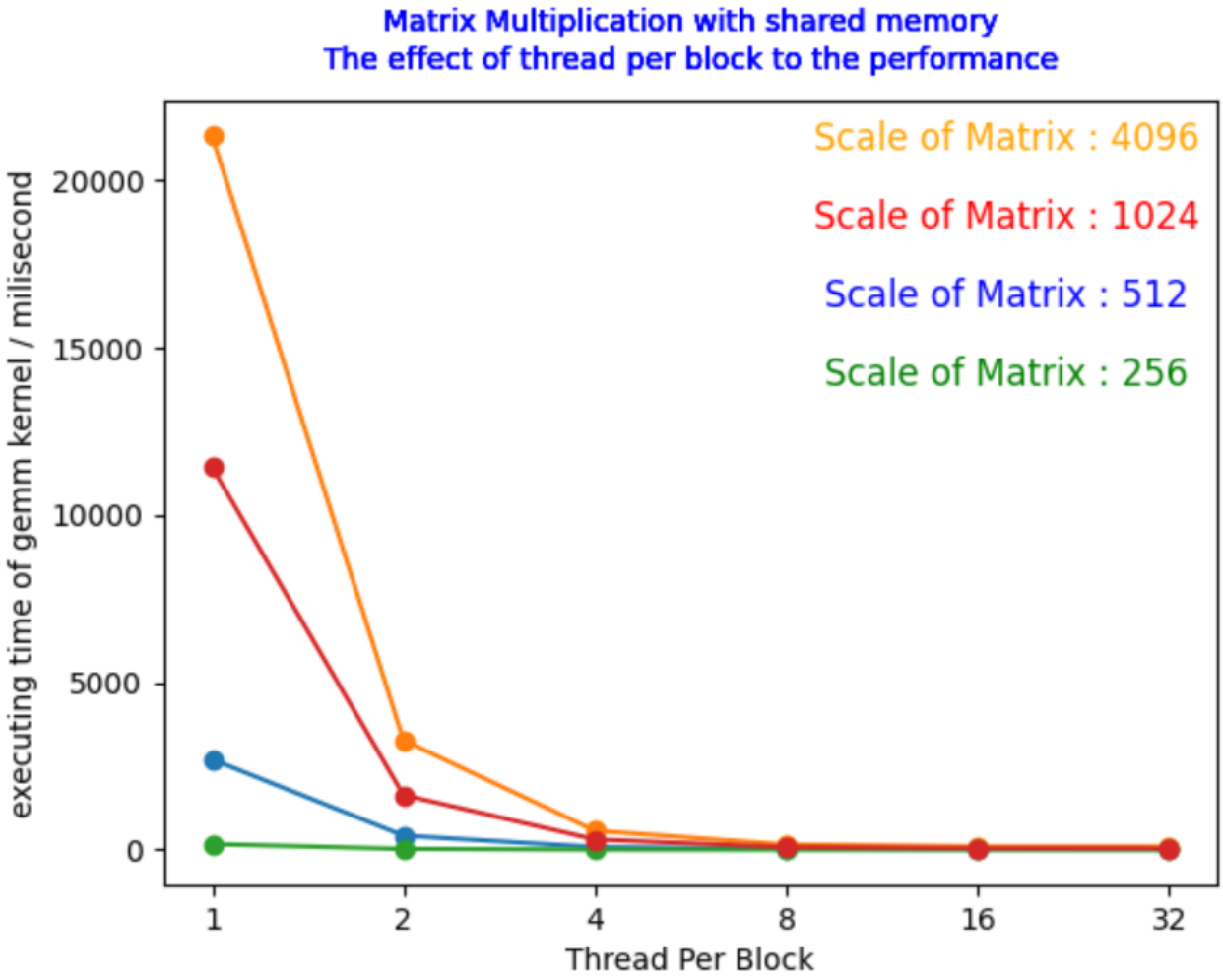

4.矩阵规模为 4096：

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
THREAD_PER_BLOCK is 1
 GPU activities:   99.33%  11.4347s         1  11.4347s  11.4347s  11.4347s  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 2
 GPU activities:   95.50%  1.63459s         1  1.63459s  1.63459s  1.63459s  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 4
 GPU activities:   79.99%  309.47ms         1  309.47ms  309.47ms  309.47ms  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 8
 GPU activities:   50.58%  78.990ms         1  78.990ms  78.990ms  78.990ms  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 16
 GPU activities:   36.62%  44.777ms         1  44.777ms  44.777ms  44.777ms  gemm(Matrix, Matrix, Matrix)
THREAD_PER_BLOCK is 32
                   34.71%  40.793ms         1  40.793ms  40.793ms  40.793ms  gemm(Matrix, Matrix, Matrix)
```

作图如下：

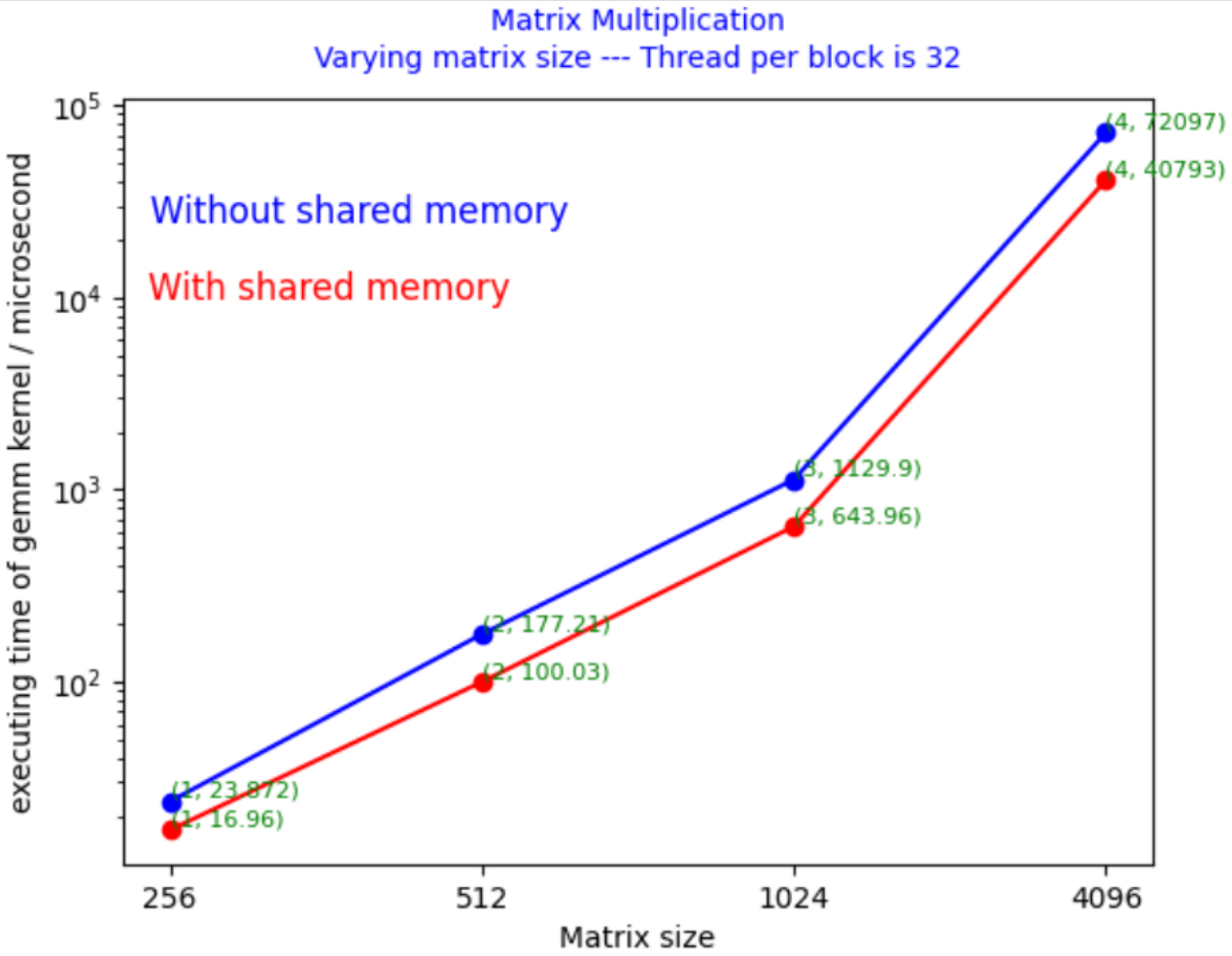



将四条曲线画在一个图中，即为：

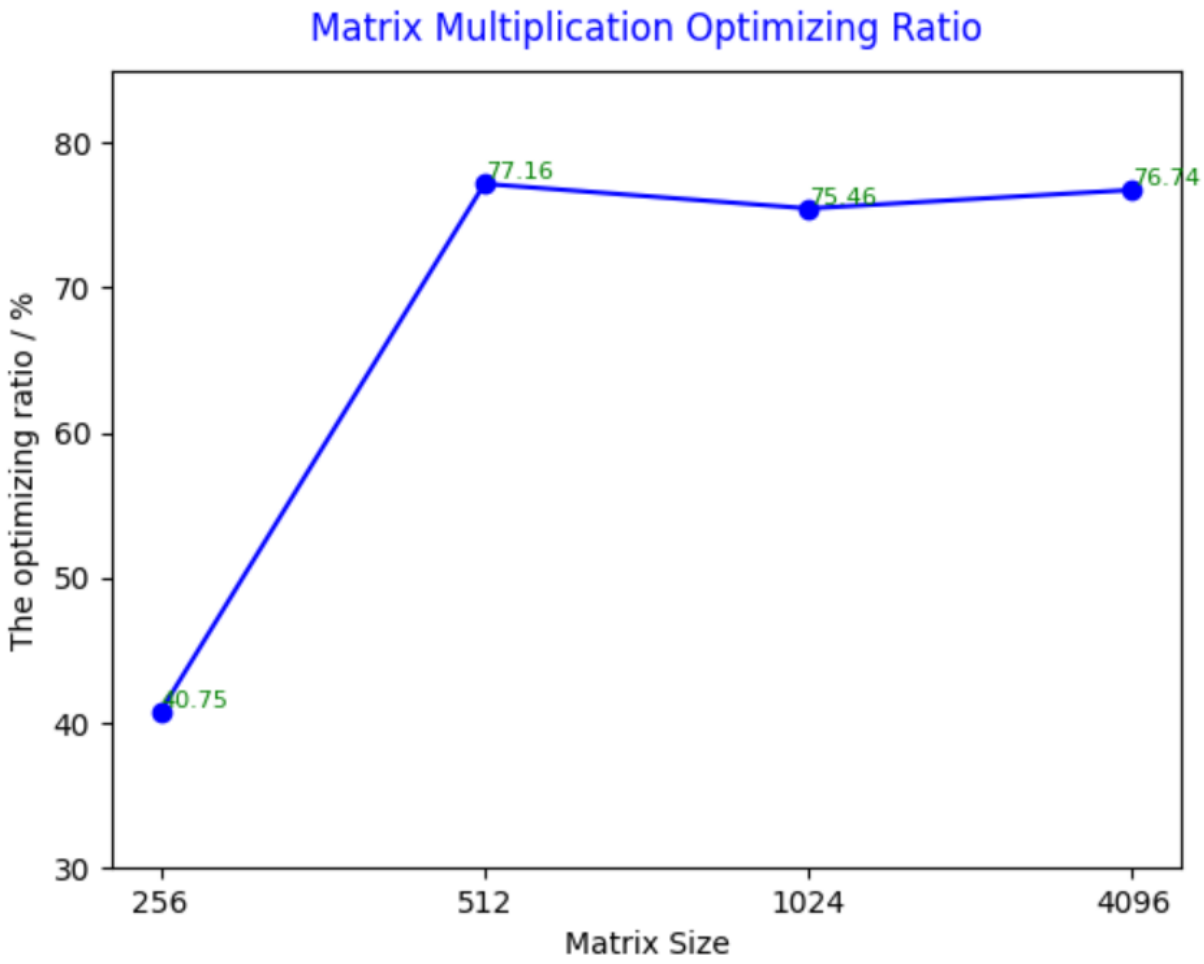



## 优化前后性能对比

经过观察，发现 thread per block 取 32 时， 运算性能最好。 所以，在该取值下，画图性能随矩阵规模变化图：




进而可以画出优化率曲线如下




# 结论与感悟

首先， 利用 GPU 进行矩阵乘法计算比 CPU 进行矩阵乘法计算要快上好几个数量级。 我们可以发现， 当矩阵规模为 4096 时， CPU 进行计算的矩阵乘法已经完全难以跑动了， 我最初尝试时计算了接近十分钟都没有结束。 而相同规模下 GPU 进行计算的矩阵乘法只需要十秒钟， 大大提升了运算效率。 这也可以看出 GPU 对于重复性独立计算的优势。

其次，通过观察实验数据，我们也可以论证发现利用 shared memory 进行优化可以大大提升性能。 这告诉我们，以后再有机会编写 GPU 编程的时候，一定要用好 shared memory， 把频繁被访问的数据存入 shared memory 中达到提速效果。

通过本次大作业，我终于有机会学习 CUDA 技术， 领悟到并行计算的思想， 一睹 GPU 的威力。 只听过课上讲解的 GPU 总会让我觉得一知半解，只有自己真正动手去

写 GPU 编程才让我真正领悟了其中的智慧。 感谢老师和助教设计的大作业和租的服务器!