南開大學
Nankai University

计算机学院

并行程序实验报告

oneAPI 编程练习

姓名：唐明昊

学号：2113927

专业：计算机科学与技术

2023 年 6 月 26 日

目录

1 问题描述

1.1 问题引入

矩阵乘法是线性代数中的一个重要运算，两个矩阵相乘得到一个新的矩阵。矩阵乘法要求，第一个矩阵 A 的列数必须等于第二个矩阵 B 的行数。

矩阵乘法在许多领域都有广泛的应用，尤其在科学计算、工程领域和计算机图形学等方面发挥着重要作用。它被用于解决线性方程组、图像处理、数据压缩、神经网络和物理模拟等问题。

为了提高矩阵乘法的计算效率和性能，尤其是针对大规模矩阵的乘法运算，考虑采用并行化手段。

一般情况下，矩阵乘法可以通过 **GPU 进行任务划分**，由一个线程计算结果矩阵一个位置的元素，从而实现并行化。

为了优化计算性能，常常需要进行**子矩阵划分**：将一个大的矩阵划分成更小的子矩阵，对子矩阵进行并行计算。例如，将矩阵 A 划分成大小为 m × k 的子矩阵，将矩阵 B 划分成大小为 k × p 的子矩阵，然后使用并行计算的方式对这些子矩阵进行乘法运算。最后，将子矩阵的结果合并得到最终的矩阵乘积。

需要注意的是，在并行化矩阵乘法时还需要考虑一些问题：

负载均衡：确保每个处理单元或线程的工作负载均衡。要求将子矩阵划分均匀，每个处理单元耗时基本一致，保证整体性能。

数据依赖：合理安排矩阵乘法，保证结果正确性以及并行执行效率。

1.2 实验数据

实验中采用的数据由 **random_float()**（rand() / double(RAND_MAX)）随机生成。通过循环遍历的方式，随机生成浮点数赋值给数组 A 和 B 的各个元素。

实验数据生成

```
// init the A/B/C
for (int i = 0; i < M * K; i++) {// 随机生成矩阵A
    A[i] = random_float();
}

for (int i = 0; i < K * N; i++) {// 随机生成矩阵B
    B[i] = random_float();
}

for (int i = 0; i < M * N; i++) {// 初始化矩阵C
    C[i] = 0.0f;
    C_host[i] = 0.0f;
}
```

矩阵 A 的大小为 M × K，矩阵 B 的大小为 K × N。使用一维数组保存矩阵元素。

循环从 0 到 m * k - 1 进行迭代，依次访问 A 的每一个元素。在循环体内部，调用 random_float() 函数生成一个随机的浮点数，并将其赋值给 A[i]。矩阵 B 同理。矩阵 C 均初始化为 0.0f。

代码运行过程生成的矩阵 A、B 的规模均为 512*512。

1.3 实验环境

实验采用 DevCloud 云端环境，测试使用 oneAPI 进行 GPU 优化的加速效果。

1.4 实验概述

本实验从矩阵乘法的平凡算法出发，逐步探究使用 oneAPI 对普通算法进行 GPU 并行化、矩阵分块优化所带来的性能提升。

探究了任务不同划分粒度对性能的影响及其原因（问题三）。思考并测试输入寄存器的规模对于 GPU 资源利用率、算法性能的影响（问题四）。最后，对整个实验进行总结，指出在期末报告中运用 oneAPI 的可能性。

2 算法实现

2.1 oneAPI 并行化

首先将 CPU 单线程矩阵乘法并行化。使用 oneAPI 进行任务划分，将矩阵 C 划分为 M*N 个任务，每个线程读取一行以及一列，独立计算累计乘积，最后将结果放回矩阵。

矩阵乘法并行化

```
// 任务划分
auto grid_rows = (M + block_size - 1) / block_size * block_size;
auto grid_cols = (N + block_size - 1) / block_size * block_size;
auto local_ndrange  = range<2>(block_size, block_size);
auto global_ndrange = range<2>(grid_rows, grid_cols);
// GPU并行计算
auto e = q.submit([&](sycl::handler &h) {
  h.parallel_for<class k_name_t>(
      sycl::nd_range<2>(global_ndrange, local_ndrange),[=](sycl::nd_item<2> index) {
          int row = index.get_global_id(0);
          int col = index.get_global_id(1);
          float sum = 0.0f;
          for (int i = 0; i < K; i++) {
            sum += A[row * K + i] * B[i * N  + col];
          }
          C[row * N + col] = sum;
      });
```

2.2 分块优化

2.2.1 算法描述

为了加速矩阵乘法，采用分块进行算法优化。

分块算法的原理是**对行与列进行复用**，因为在矩阵相乘的过程中，矩阵 A 中的一行将与矩阵 B 中的 n 个列都相乘分别累加，然后得到 C 中的 n 个元素；对称的，矩阵 B 中的一列将与矩阵 A 中的

m 个行都相乘。由此可见，A 中的一行在矩阵乘法的过程中将被利用 n 次，而 B 中的一列将在乘法过程中被利用 m 次。平凡的循环迭代算法里，每个子任务只是读出 A 的一行和 B 的一列，只使用一次，利用效率较低。

由以上的分析，可以得到矩阵乘法任务划分的新思路：**矩阵分块任务划分**。每个子任务中不在只包含结果矩阵 C 中的 1 个位置，而是包含 C 中的 tileX*tileY 个位置。对于 A 中的一行，会进行 tileY 次利用；相应的，对于 B 中的一列会进行 tileX 次利用。

2.2.2 代码实现

核心计算部分，使用每个工作线程的私有变量来保存计算的中间结果，即相当于利用 **Input Registers** 的方法，避免反复访问全局变量，从而减少访问同步开销。而输入寄存器的值，在计算过程中会被反复利用，以提高算法效率。

分块任务划分

```
// 任务划分
auto grid_rows = M / tileY;
auto grid_cols = N / tileX;
auto local_ndrange = range<2>(BLOCK, BLOCK);
auto global_ndrange = range<2>(grid_rows, grid_cols);
// GPU并行计算
auto e = q.submit([&](sycl::handler& h) {
    h.parallel_for<class k_name_t>(
        sycl::nd_range<2>(global_ndrange, local_ndrange), [=](sycl::nd_item<2> index)
            {
            int row = tileY * index.get_global_id(0);// 每个子矩阵有tileY行
            int col = tileX * index.get_global_id(1);// 每个子矩阵有tileX列
            .......
            // core computation
            for (int k = 0; k < N; k++) {
                // 读入输入寄存器subA, subB
                for (int m = 0; m < tileY; m++)
                    subA[m] = A[(row + m) * K + k];
                for (int p = 0; p < tileX; p++)
                    subB[p] = B[k * N + p + col];
                // 计算中间结果
                for (int m = 0; m < tileY; m++)
                    for (int p = 0; p < tileX; p++)
                        sum[m][p] += subA[m] * subB[p];
            }
            // 计算结果并写回矩阵
            for (int m = 0; m < tileY; m++)
                for (int p = 0; p < tileX; p++)
                  C[(row + m) * N + col + p] = sum[m][p];
        });
    });
e.wait();
```

通过上面的调整，仅需要在计算之前从全局内存中读取 (tileY+tileX)*N 次数据，最后再使用 tileX*tileY 次操作写回全局内存。

如果不利用输入寄存器策略，只是进行三**重循环的一般策略**，每次从全局读取数据并计算，则计算过程的代码如下：

一般策略

```
// 计算结果并写回矩阵
for (int m = 0; m < tileY; m++)
    for (int p = 0; p < tileX; p++)
        for(int k = 0; k < N; k++)
            C[(row + m) * N + col + p] += A[(row + m) * K + k] * B[k * N + p +
                col];
```

可以分析，该策略将对全局数据矩阵 A、B 读取 2*tileX*tileY*N 次，效率大幅落后于使用 Input Registers 策略。除了读取次数上的落后，还可以观察到，对于 B 的读取并不是顺序的，而是每次间隔 N 个元素读取。该方法空间局部性差于 Input Register 策略，会造成一定的性能损失。

当然，由于分块的优势，可以对该代码进行简单的改动，以重复利用 A 中一行元素 tileY 次，进而优化读取次数。但是对 C 的写入操作却无法优化，次数仍然远大于输入寄存器策略。

以上即是代码采用输入寄存器策略的原因。

3 问题探究

3.1 tileX、tileY 对性能的影响

改变算法中 tileX、tileY 的大小，探究其对性能的影响，得到实验结果如图3.1所示：

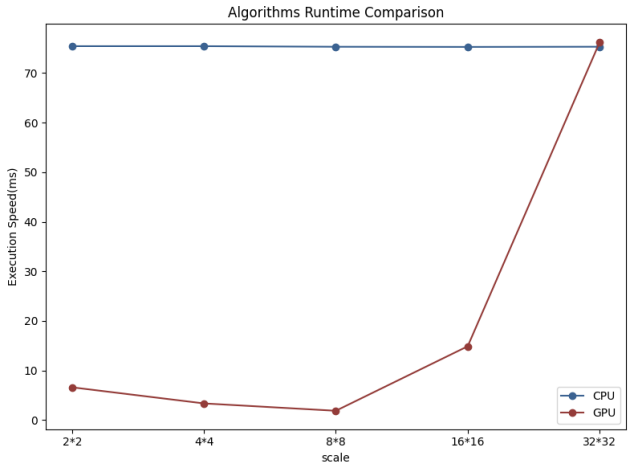

图 3.1: 并行算法不同规模执行时间以及与串行对比（单位：ms）

由图可知，tileX、tileY 的规模增大初期，GPU 并行化算法执行速度随之逐渐增大。直到规模达到 8*8 拐点出现，速度逐渐下降。继续观察图中数据可得，进一步增大规模，当 tileX*tileY 增长到 32*32 时，GPU 算法的性能已经低于 CPU 算法。

tileX、tileY 决定了任务划分粒度的大小：tileX、tileY 越大，任务划分的粒度越粗，每个工作线程得到的任务越多；反之任务越少。而经查阅资料得，任务量的大小将会影响开辟线程数的多少：tileX、

tileY 越大，子任务的任务量更大，故 GPU 将开辟较少的线程；相应的，如果一个子任务的任务量较小，GPU 将开辟较多的线程。

据此分析，造成图3.1现象的原因可能是：

- tileX、tileY 过小时，每个线程所分配的任务量较小，由此**需要开辟较多的线程**。而由于线程的创建、销毁和切换都需要一定的时间和资源。另外，**线程的调度**也造成额外的开销：GPU **硬件资源限制**，所以线程共享 GPU 上面有限的硬件资源，如寄存器、共享内存、总线等；当线程数量较多时，会出现资源竞争而被迫等待的情况，影响执行效率。
- tileX、tileY 过大时，每个线程都将被分配大量工作，且总线程数减少。**少数线程负载大量任务**，而其他非工作线程则处于空闲状态，GPU 利用率过低，是对 GPU 计算资源的浪费。

3.2 输入寄存器大小对性能影响

3.2.1 问题描述

2.2.1节中分析了在核心计算部分，使用 Input Registers 策略能够有效的减少从全局内存中读取数据的次数，从而提高运行效率。

而在2.2.1节中所使用的输入寄存器总数为 suB 的 tileX + subA 的 tileY 个。思考，如果 GPU 还有剩余的存储资源未被利用，能否通过提高使用的输入寄存器个数，进而充分利用剩余的存储资源，从而取得更好的性能。

3.2.2 实验测试

将原有代码进行改动，参考循环展开的思想，每次循环读入的数据量提升至 length* (tileX+tileY) 个。

改动输入寄存器

```
for (int k = 0; k < N/length; k++) {
    // 读入输入寄存器subA[tileY][length], subB[tileX][length]
    for (int m = 0; m < tileY; m++)
        for (int f = 0; f < length; f++)
            subA[m][l] = A[(row + m) * K + k * length + f];
    for (int f = 0; f <length; f++)
        for (int p = 0; p < tileX; p++)
            subB[p][l] = B[(k * length + f) * N + p + col];
    // 计算中间结果sum
    for (int f = 0; f<length; f++)
        for (int m = 0; m < tileY; m++)
            for (int p = 0; p < tileX; p++)
                sum[m][p] += subA[m][f] * subB[p][f];
}
// write results back
```

3.2.3 结果分析

当 tileX，tileY 均为 2 时，改变 length 的大小即可调整所使用的输入寄存器个数，进而得到使用不同数量输入寄存器的情况下，算法运行效率如图3.2所示。(64+64 表示 subA 的规模为 tileY*length

= 64，此时 subB 的规模也为 64。2+2 即表示为不进行改动，subA 和 subB 规模均为 2）

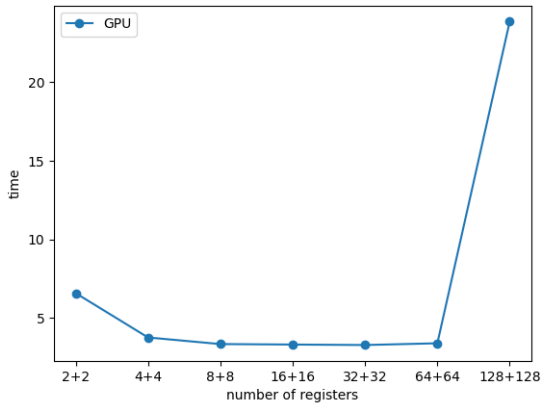

图 3.2: 不同输入寄存器个数下性能（单位：ms）

由图可知，随着使用的输入寄存器的个数逐渐增多，算法的运行速度先不断提升，达到拐点后，速度急剧降低。在 128+128 的规模下，算法的速度是不改变大小的约 1/4。

- 输入寄存器增多，充分利用了 GPU 的剩余资源，一次循环能够**读入更多的数据**。另外引入更多存储寄存器后，初始化 subA 能够连续读取多行，在一定程度上提高了程序的**空间局部性**。由图也可以看出，寄存器个数从 2+2 到 4+4 性能有明显提升。
- 继续增大使用的输入存储器的数量，GPU 资源已逐渐被消耗殆尽，但由于空间局部性平衡资源资源消耗的影响，性能仍然有略微提升。
- 输入寄存器规模从 64+64 提升到 128+128 后，**资源完全被消耗空**，过度利用存储器对算法造成巨大的性能损耗。

4 实验总结

在本次 oneAPI 编程练习中，使用分块矩阵的方法对矩阵相乘问题进行了并行化优化，并探究了两个问题：划分块大小对算法性能的影响；输入寄存器的数量多少对算法性能的影响。

通过实验测试具体的数值与查阅相关资料，探究分块优化与读入寄存器引起性能差异的具体原因。两个问题最终都指向了 GPU 资源的合理利用。

通过本次实验，我熟练掌握了 oneAPI 的运用，熟悉了 oneAPI 编程范式，任务划分方法等相关操作。深刻体悟到了 GPU 并行化的特点，以及对 GPU 资源使用应该注意的问题。在期末报告撰写中，我也会尝试引入 oneAPI，对自主选题进行并行优化，挖掘其更多价值。

5 线上学习记录

于 5 月 17 日与 5 月 23 日两次参加英特尔 oneAPI 校园黑客松系列培训，学习了“用 Intel@oneAPI 基础工具套件中的 C++/SYCL 直接编程语言实现异构编程” 和 “基于 Intel@oneAPI 的应用性能分析工具、优化方法”。过程中掌握了 oneAPI 的基本使用，了解了 oneAPI 工具套件的强大功能，为本次编程练习打下了坚实基础。

图 5.3: 云端课程学习截图