南開大學
Nankai University

计算机学院

并行程序设计实验报告

GPU 研究报告

姓名：刘心源

学号：2112614

专业：计算机科学与技术

2023 年 6 月 27 日

目录

1 讲座学习截图

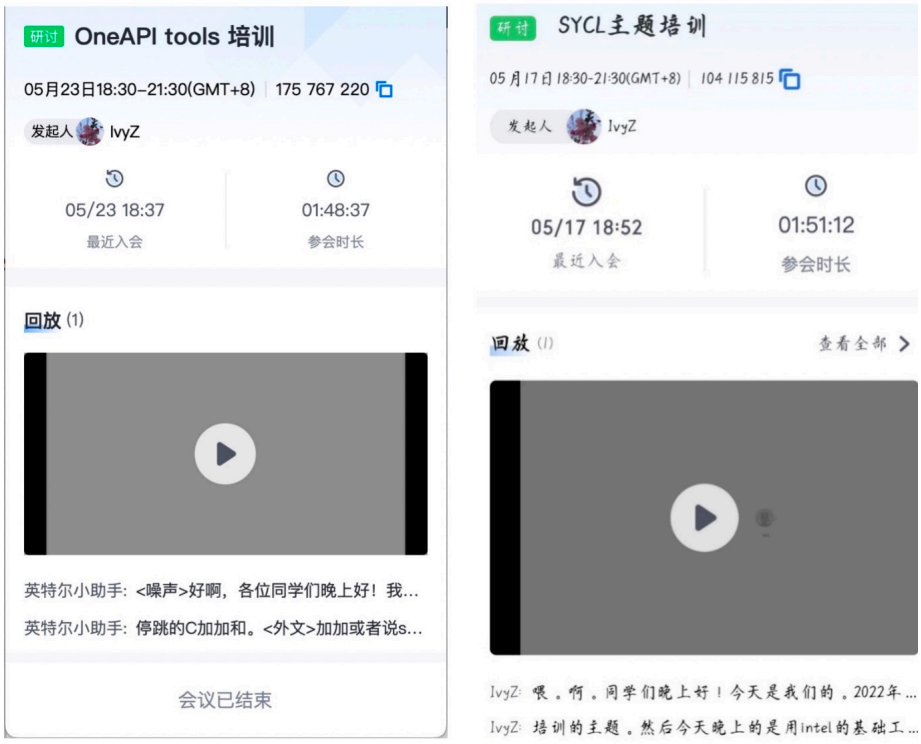

图 1.1: 两次讲座截图

2 Devcloud 平台的两道题

2.1 exer 3

根据 github 上所给代码，对于 tile_X 和 tile_Y 进行修改，并使用 oneAPI 测量矩阵计算性能。首先对于 tile_X 和 tile_Y 大小相等的情况进行测量，并记录结果如下：

tile_X	tile_Y	GPU_Time
2	2	8.012398
4	4	4.219492
8	8	2.3091943
16	16	19.48172
32	32	97.283992
64	64	434.29499
128	128	1512.14

表 1: tileX 和 Y 相等情况下不同 tile 测试结果

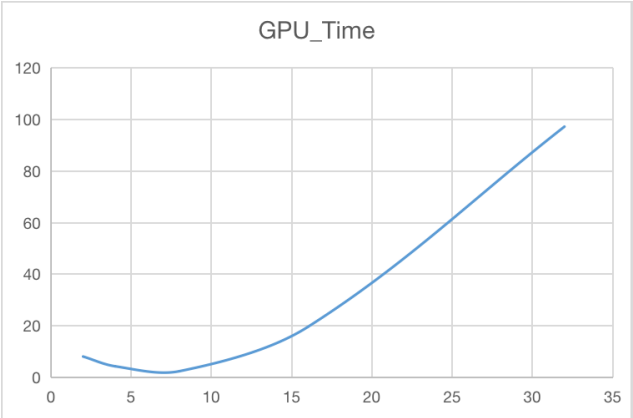

图 2.2: 测试得到 GPU Time 随 X/Y 的趋势

由图1.1可得出当 tile_X 和 tile_Y 的大小从 2 增加到 8 时，GPU 计算时间显著减少。这可能是因为较小的 tile_X 和 tile_Y 导致每个线程负责的计算量增加，从而降低了计算性能。当 tile_X 和 tile_Y 增加时，每个线程负责的计算量减少，从而提高了计算性能。

但是当 tile_X 和 tile_Y 继续增加到 16、32、64 和 128 时，GPU 计算时间却显著增加。这可能是因为较大的 tile_X 和 tile_Y 导致每个工作组中的线程数增加，从而导致硬件资源（如寄存器）不足，降低了计算性能。

综合来看，使用本机 GPU 进行测试时，tile_X 和 tile_Y 等于 8 时可以获得最佳计算性能。同时也说明选择合适的 tile_X 和 tile_Y 大小对于获得最佳计算性能至关重要。

2.2 exer 4

根据 github 上所给代码，由题目得需要增加输入寄存器的大小，让其可以一次读取更多的行列，也就是一次从全局内存中读取多行 A 矩阵和多列 B 矩阵，以减少全局内存访问次数。故可以将读取行列的代码进行修改：

行列读取代码

```
//修改后
for(int m = 0; m < tileY; m++) {
    for(int n = 0; n < UNROLL_FACTOR; n++) {
        subA[m][n] = A[(row + m) * N + k + n];
    }
}

for(int p = 0; p < tileX; p++) {
    for(int n = 0; n < UNROLL_FACTOR; n++) {
        subB[p][n] = B[(k + n) * N + p + col];
    }
}
```

其中，each 是一个常数，表示一次读取的行/列数。同时也需要相应地调整 subA 和 subB 数组的大小，并修改计算 C 矩阵元素的代码。

修改代码后，调整 each 和 X、Y 大小，测量得到以下结果：

tile_X	tile_Y	UNROLL_FACTOR	GPU time
2	2	1	7.62343
		2	4.78629
		4	4.28123
		8	4.14837
		16	4.05643
		32	5.32928
		64	24.23489
		128	25.39452
		256	22.19839
16	16	1	16.83478
		2	14.37342
		4	37.44837
		8	36.29983
		16	36.23498
		32	8.29383
		64	14.39745
		128	39.28374
		256	48.76949
64	64	1	369.98654
		2	407.39589
		4	380.54983
		8	85.46574
		16	75.46527
		32	61.34875
		64	75.43458
		128	598.80834
		256	432.54837

表 2: 不同寄存器大小下测试 GPU Time

当 tile_X 和 tile_Y 等于 2 时，随着 UNROLL_FACTOR 的增加，GPU 计算时间开始保持在较低水平，随后增加。但是在 each=1 时较大，这可能是因为较小的 UNROLL_FACTOR 导致全局内存访问次数增加，从而降低了计算性能。当 UNROLL_FACTOR 增加时，全局内存访问次数减少，从而

提高了计算性能。然而，当 UNROLL_FACTOR 继续增加到 32、64 和 128 时，GPU 计算时间却显著增加，因为较大的 UNROLL_FACTOR 导致寄存器使用量增加，从而导致硬件资源不足，降低了计算性能。

当 tile_X 和 tile_Y 等于 16 时，随着 UNROLL_FACTOR 的增加，GPU 计算时间先增加后减少，因为较小的 UNROLL_FACTOR 导致全局内存访问次数增加，从而降低了计算性能。然而，当 UNROLL_FACTOR 增加到 4、8 和 16 时，GPU 计算时间并没有显著减少，由于较大的 UNROLL_FACTOR 导致寄存器使用量增加，从而抵消了全局内存访问次数减少带来的性能提升。当 UNROLL_FACTOR 继续增加到 32 时，GPU 计算时间显著减少，则可能是因为此时全局内存访问次数减少的效果大于寄存器使用量增加的影响。

当 tile_X 和 tile_Y 等于 64 时，随着 UNROLL_FACTOR 的增加，GPU 计算时间先增加后减少。这与 tile_X 和 tile_Y 等于 16 时的情况类似。

综上所述，不同的 tile_X 和 tile_Y 组合对应不同的最佳 UNROLL_FACTOR 值。例如，当 tile_X 和 tile_Y 等于 2 时，最佳 UNROLL_FACTOR 值为 16；当 tile_X 和 tile_Y 等于 16 时，最佳 UNROLL_FACTOR 值为 32；当 tile_X 和 tile_Y 等于 64 时，最佳 UNROLL_FACTOR 值为 32。这说明选择合适的 UNROLL_FACTOR 值对于获得最佳计算性能至关重要。

3 在 Cuba 上的 GPU 实验

近年来，加速计算正逐渐取代传统的 CPU 计算，成为当前最优的计算方法。其中，使用 GPU 进行加速已成为主流。在本次实验中，我们基于 NVIDIA 的 GPU 学习平台，学习了与 CUDA 相关的基本原理和使用技巧。

CUDA 是一种计算平台，它提供了一种编程范式，可扩展到 C、C++、Python 和 Fortran 等语言，使得经过加速的大规模并行代码可以在全球性能强大的并行处理器 NVIDIA GPU 上运行。通过 CUDA，我们可以轻松地大幅加速应用程序，并且还可以使用高度优化的库生态系统，适用于深度神经网络（DNN）、基础线性代数子程序（BLAS）、图形分析和快速傅里叶变换（FFT）等多种运算。此外，CUDA 还提供了强大的命令行和可视化性能分析工具，方便我们进行性能优化和调试。

3.1 学习目标

1. 编写编译及运行既可以调用 CPU 函数又可以启动 GPU 核函数的 C/C++ 程序
2. 使用执行配置控制并行线程层次结构
3. 分配和释放可用于 CPU 和 GPU 的内存
4. 加速 CPU 应用程序

3.2 实验分析

对于本次期末选题 Gauss 消去实验，则根据实验指导书上步骤进行简单的 GPU 优化。分析如下：

如果想要让某个函数能够分配到 GPU 上执行，需要首先将这个函数声明为核函数，即使用 ___global___ 关键字，这个关键字表明以下函数将在 GPU 上运行，并且可以用 CPU 或 GPU 调用。通常的，将 CPU 上执行的代码成为主机代码，而将 GPU 上执行的代码成为设备代码。

定义 GPU 上的求解函数 solveOnGPU()（代码太长就不在此处展示），该函数通过 CUDA 在 GPU 上进行并行计算。首先在 GPU 上分配内存并将输入数据拷贝到 GPU 内存中，然后使用 gpuSolveBottom 核函数和 gpuSolveTop 核函数对矩阵进行处理，最后将结果拷贝回 CPU 内存，两个核函数的代码如下：

核函数

```
1  ___global___
2  void gpuSolveBottom(matrix d_A){
3    int j = (blockIdx.x * blockDim.x + threadIdx.x) + k;
4
5    ___shared___ double temp;
6    temp = d_A[k][k];
7    double selectedRow = d_A[k][j] / temp;
8    ___syncthreads();
9    for (int i = k + 1; i < MATRIX_SIZE; i++){
10     temp = d_A[i][k];
11     d_A[i][j] -= selectedRow * temp;
12     ___syncthreads();
13   }
14   d_A[j][k] = selectedRow;
```

```
15      }
16      __global__
17      void gpuSolveTop(matrix d_A){
18        int i = (blockIdx.x * blockDim.x + threadIdx.x);
19        for (int j = MATRIX_SIZE - 1; j > 0; j--){
20          if (i < j){
21            d_A[MATRIX_SIZE][i] -= d_A[MATRIX_SIZE][j] * d_A[j][i];
22            __syncthreads();
23          }
24        }
25      }
```

3.3 实验结果

为了更好地比较 GPU 的性能，我们同时对于 CPU 与 GPU 上的运行时间进行测量，其时间如下：

matrix_size	GPU_time	CPU_time
128	0.198342	0.00075
256	0.186347	0.005432
512	0.057236	0.121234
1024	0.218743	0.523478
2048	0.028373	4.238567
4096	0.19357	31.4395
8192	0.238578	312.773

表 3: GPU 和 CPU 求解时间比较

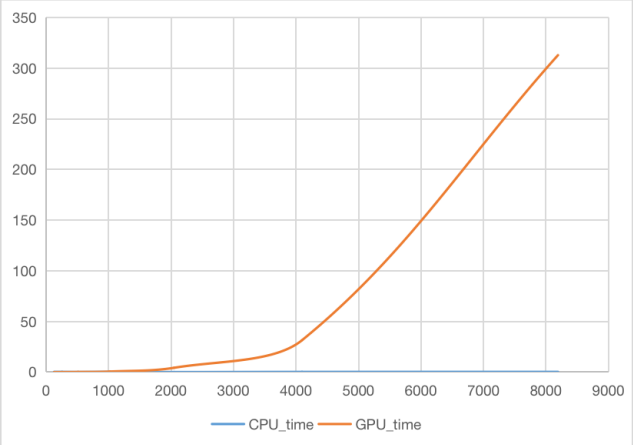

图 3.3: 高斯消元 CPU 与 GPU 耗时曲线

比较 CPU 和 GPU 的耗时。从图中可以看出，在所有的矩阵大小下，GPU 的耗时都明显低于 CPU 的耗时。这是因为 GPU 具有并行计算的能力，可以同时处理多个线程，从而加快计算速度。相比之下，CPU 是顺序执行的，无法充分利用并行性。在数据规模较大的时候，CPU 的耗时将会迅速增大，性能也将大大下降。

4 总结反思

4.1 实验总结

在本次实验中，我首先在 Intel 的 DevCloud 平台对于 exer3，4 完成并进行测量分析。我学习了 CUDA 的知识。我学习了如何声明和调用 CUDA 的核函数，深入理解了 CUDA 的线程层次结构，包括线程块和线程的概念，并学会如何利用这些概念来优化循环操作。最后，我应用所学知识尝试对 Gauss 消去算法进行了基础优化，并取得了显著的性能提升。这次实验让我详细了解了 CUDA 的相关技术，并且通过实际应用加深了对 GPU 并行计算的理解。

具体代码已经上传到刘大圆的 Github。