南開大學
Nankai University

计算机学院

并行程序设计实验报告

GPU 编程

2023 年 6 月 2 日

目录

1 oneAPI 和 DPC++ 基础知识学习

Intel DevCloud 平台上的 DPC++ 学习较为简单，只需按照教程登录 JupyterLab，在网页上完成例程的运行，并修改矢量相加程序使其成功运行即可。这里给出截图作为完成的证明，不多赘述。

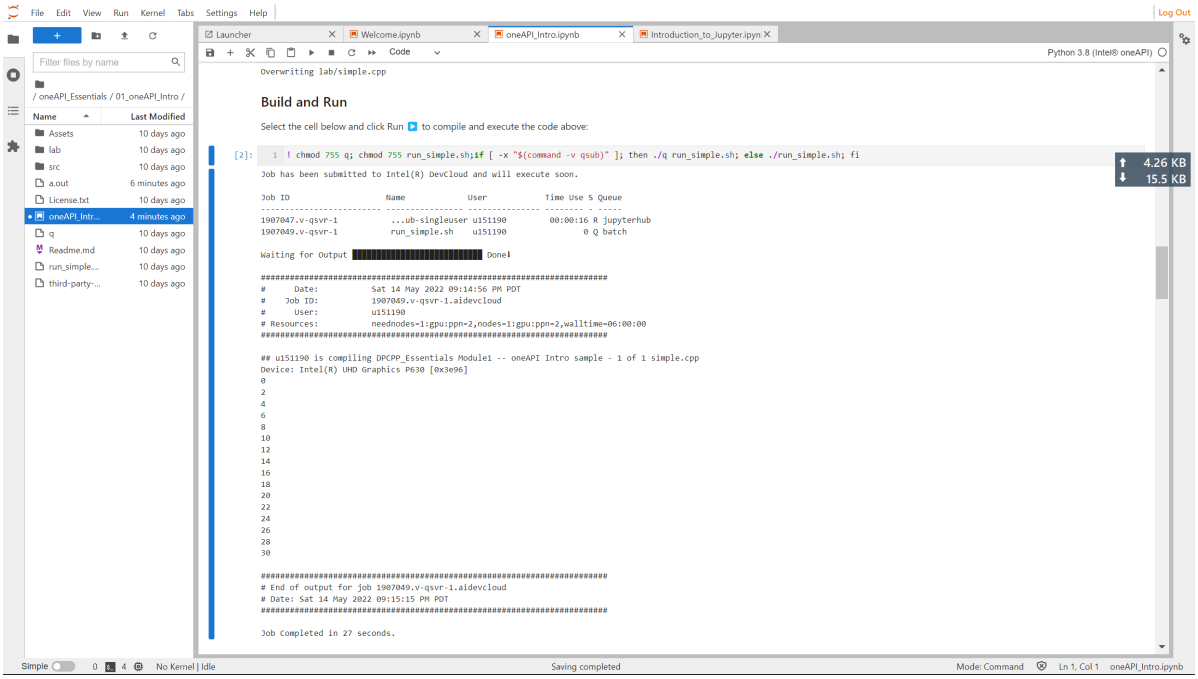

图 1.1: oneAPI 快速闯关中的 HelloWorld 程序（并行输出数字）

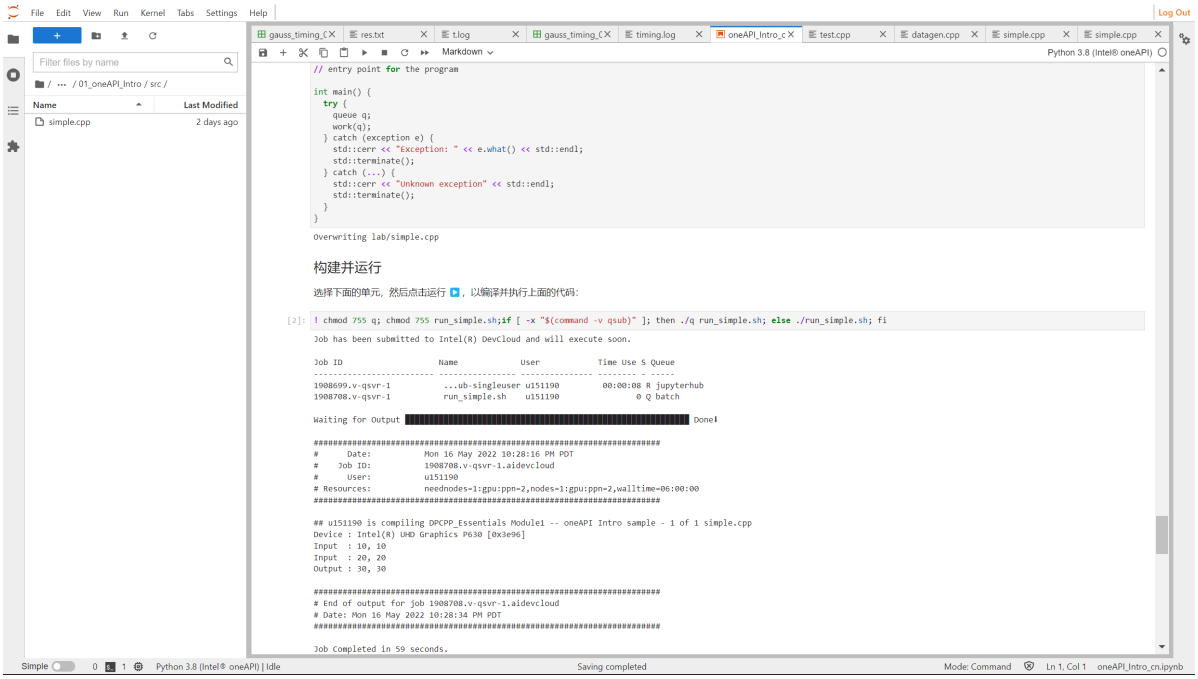

图 1.2: oneAPI 快速闯关中的矢量相加程序

图 1.3: oneAPI 问卷截图

2 DPC++ 实现普通高斯消去算法

2.1 学习过程

在完成闯关指南中的 Hello World 部分后，作者还通过 DevCloud JupyterLab 中的内容进行了进一步学习。同时还参考了oneAPI 的 Specs等学习资源。

在 DPC++ 的学习中，作者还参加了英特尔举办的 oneAPI 征文大赛，以期实现教学相长的效果。在比赛征文中，作者通过 oneAPI 和 DPC++ 这些工具，实现了高斯消去的 CPU 和 GPU 并行化，获得了不错的加速比。比赛征文的链接是英特尔 oneAPI - 高斯消去的并行化。并且征文最终也在比赛中拿到了二等奖的成绩。

图 2.4: oneAPI 征文比赛获奖截图

2.2 核心代码

在 DPC++ 下，并行化高斯消去算法的实现如下。并行化算法在 DPC++ 中的实现相比串行算法只需要替换极少量的代码，因此串行算法不再给出。

```
#ifndef WIDTH
#define WIDTH 1024
#endif

static const int N = WIDTH;

typedef float ele_t;
ele_t mat[N][N];

void LU_gpu(ele_t mat[N][N], int n)
{
    queue q{ cpu_selector{} };
    // queue q{ gpu_selector{} };

    ele_t(*new_mat)[N] = (ele_t(*)[N])malloc_shared<ele_t>(N * N, q);
    memcpy(new_mat, mat, sizeof(ele_t) * N * N);

    timespec start, end;
    double time_used = 0;
    clock_gettime(CLOCK_REALTIME, &start);

    for (int i = 0; i < n; i++)
        q.parallel_for(range{(unsigned long)(n - (i + 1))}, [=](id<1> idx)
        {
        // 等同于 for(int j=i+1; j<n; j++)
        int j = idx[0] + i + 1;
        ele_t div = new_mat[j][i] / new_mat[i][i];
        for (int k = i; k < n; k++)
            new_mat[j][k] -= new_mat[i][k] * div;
        }).wait();

    clock_gettime(CLOCK_REALTIME, &end);
    time_used += end.tv_sec - start.tv_sec;
    time_used += double(end.tv_nsec - start.tv_nsec) / 1000000000;
    std::cout << " 并行算法用时: " << time_used << std::endl;
    std::cout << " 并行算法使用设备: " << q.get_device().get_info<info::device::name>();
    std::cout << std::endl;
```

```
38  }
```

完整代码可以在GitCode 平台找到。

这里需要注意的一点是，我们使用到了 $range(unsignedlong)(n-(i+1))$ 和 $intj = idx[0]+i+1;$ 来指定循环变量范围为 $[i+1,n)$，这个写法是相对麻烦的。

其实，笔者一开始也想要通过某种方法给 parallel_for 指定一个不从 0 开始的循环范围。但在查阅资料 [1] 后，发现 DPC++ 所使用的 SYCL2020 标准中，已经移除了可以使得 parallel_for 的循环变量不从 0 开始的特性。因此，我们需要手动的进行偏移的操作。

2.3 实验环境

在本次实验中，作者在安装了 oneAPI 开发套件的 Visual Studio 2022 中完成了代码的编写与调试，在笔者的笔记本电脑和 Intel Devcloud 中完成了性能测量。

笔者笔记本电脑使用的环境为 Visual Studio 2022 + oneAPI base kit 2022.2.0.166。电脑 CPU 型号为 12th Gen Intel(R) Core(TM) i7-12700H，GPU 为配套的核显。

Devcloud 上编译代码使用的编译器为 Intel oneAPI DPC++/C++ Compiler 2022.1.0 (2022.1.0.20220316)。Devcloud 平台所用的 CPU 为 Intel Xeon Gold 6128，具体参数如下所示。

```
1   Architecture:                    x86_64
2   CPU op-mode(s):                  32-bit, 64-bit
3   Byte Order:                      Little Endian
4   Address sizes:                   46 bits physical, 48 bits virtual
5   CPU(s):                          24
6   On-line CPU(s) list:             0-23
7   Thread(s) per core:              2
8   Core(s) per socket:              6
9   Socket(s):                       2
10  NUMA node(s):                    2
11  Vendor ID:                       GenuineIntel
12  CPU family:                      6
13  Model:                           85
14  Model name:                      Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
15  Stepping:                        4
16  CPU MHz:                         1200.236
17  CPU max MHz:                     3700.0000
18  CPU min MHz:                     1200.0000
19  BogoMIPS:                        6800.00
20  Virtualization:                  VT-x
21  L1d cache:                       384 KiB
22  L1i cache:                       384 KiB
23  L2 cache:                        12 MiB
24  L3 cache:                        38.5 MiB
```

```
NUMA node0 CPU(s):               0-5,12-17
NUMA node1 CPU(s):               6-11,18-23
```

2.4 实验过程

Devcloud 上首先编译数据生成器并生成数据，然后编译程序并运行。

```
g++ -DN=4096 ./datagen.cpp -o datagen
./datagen
dpcpp -DWIDTH=1024 ./test.cpp && ./a.out
dpcpp -DWIDTH=2048 ./test.cpp && ./a.out
dpcpp -DWIDTH=3072 ./test.cpp && ./a.out
dpcpp -DWIDTH=4096 ./test.cpp && ./a.out
```

Windows 下则将数据复制到项目目录下，之后多次修改宏定义值，并以 VS 的 Release 模式编译。

2.5 实验结果

运行计时性能测量的结果如下二表所示。

矩阵行数\并行方法	不加速	CPU 并行
1024	0.702722	0.272612
2048	5.61782	1.62869
3072	18.9493	5.23514
4096	45.2458	12.1328

表 1: Devcloud 平台上普通高斯消去算法运行时间（单位：s）

矩阵行数\并行方法	不加速	CPU 并行	GPU 并行
1024	0.0516336	0.539425	0.405941
2048	0.692901	0.807384	1.74969
3072	2.6067	1.88383	5.04222
4096	6.78803	4.56309	12.7793

表 2: PC 上普通高斯消去算法运行时间（单位：s）

2.6 结果分析

2.6.1 GPU 的负优化

在表2中，不难注意到，GPU 加速的高斯消去算法在运行时间上明显长于 CPU 加速算法的运行时间，也逊色于普通串行算法。导致这样的负优化的原因有两点：首先，加速算法所使用的 GPU 为 CPU 内置的核显，不是专门为了计算而设计的，只能承载简单的图形处理任务。其次，我们的并行化是在算法的第二层循环完成的，而这一层循环中还有用于处理无需消元行的判断代码，这样的场景是 GPU 所不擅长处理的，因此 GPU 加速算法的效果可能才不理想。

2.6.2 Devcloud 平台的性能限制

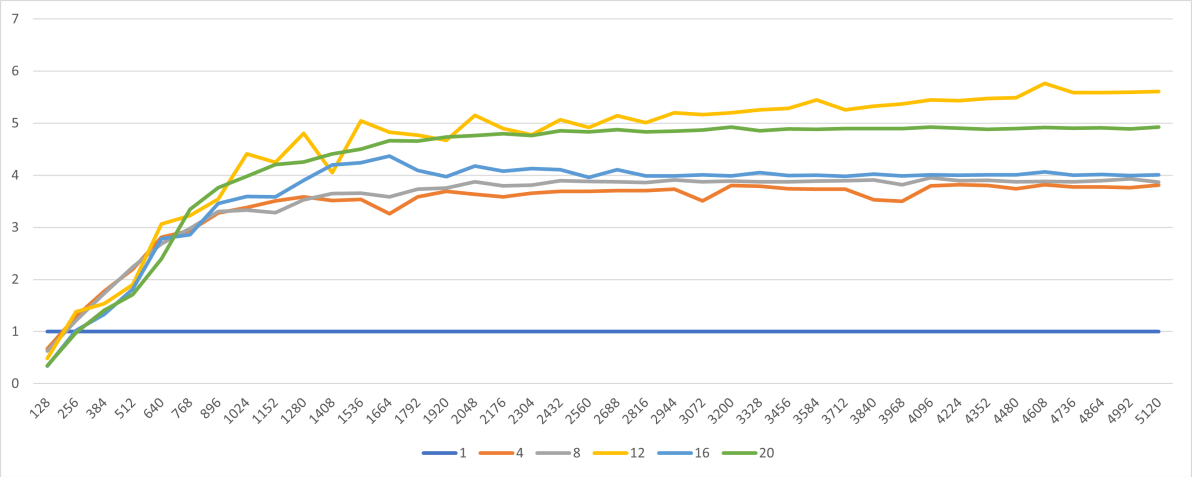

图 2.5: X86 平台下 Pthread 加速普通高斯消去算法加速比-输入数据规模（矩阵行数）

通过 Linux 下的性能检测工具 top 发现，在 DevCloud 平台上的实验过程中，并行算法运行时的 CPU 的占用率基本在 1200% 左右徘徊（top 中 CPU 占用率超过 100% 时表明程序正占用多个核心运行），而 DevCloud 平台使用的 CPU 是 24 核的 Xeon Gold 6128 处理器，并且作者在完成 Pthread 和 OpenMP 实验时发现了相似的限制（图2.5），因此作者怀疑 Devcloud 平台限制了单用户的性能。

2.6.3 服务器 CPU 和家用 CPU 的差距

对比表1和表2，不难发现，在作者笔记本上运行的高斯消去算法运行时间明显优于 Devcloud 平台，但同时并行加速比也更低了。

之所以会出现这样的性能差距，除了上文中提到的 DevCloud 平台的性能限制，主要是因为服务器 CPU 和家用 CPU 的用途不同：服务器 CPU 更注重高并发、低功耗的特性，因此并行化的特征更能在设计中体现出来，而目前许多客户端应用中的运算比较“碎片化”，对突发性能需求较高，因此设计上更注重单核性能，于是就导致了实验中的这样的现象。

参考文献

[1] aland. c++ - dpc++ start the do loop from 1 to n-2 using parallel_for range. https://stackoverflow.com/questions/72020782/dpc-start-the-do-loop-from-1-to-n-2-using-parallel-for-range Accessed April 24, 2022.