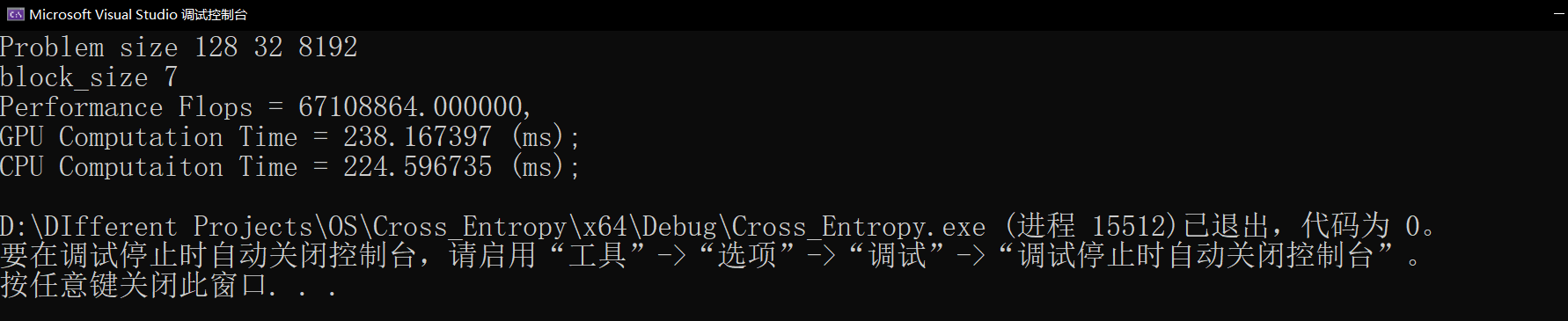
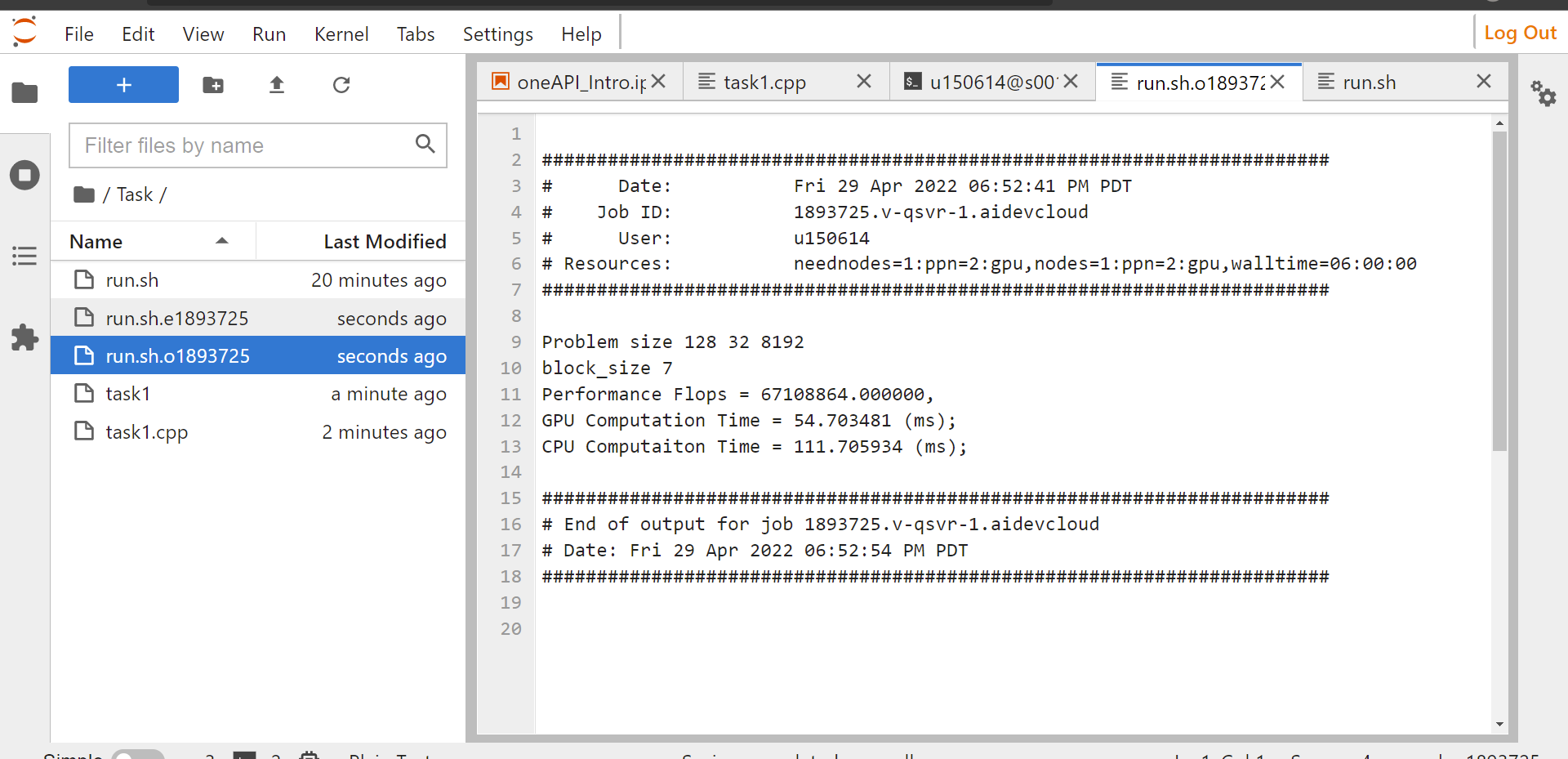
首先是确立cross\_entropy的目标：建立一个X[K][M][N]的立方体，作为基础数据。mask[K][N]，表示每列选取一个元素a，该元素在该列中的行坐标。weight[K][N]作为第(k,n)个的y在计算loss时的权重。其中X,mask,weight每个元素都是(0,1)区间中的随机数。其中的softmax是对每一个N列做M维向量的归一化处理。

本次实验的整体思路都建立在github中提供的程序框架上来进行。首先是建立gemm(Corss\_Entropy)的核心函数，然后分别在CPU和GPU中计算，通过两者结果的误差比较小于1e-3来保证计算结果的正确性，同时统计CPU和GPU的计算时间，进行性能对比。最后是通过warmup和iterations来预热计算和多次计算，得到更为可靠的计算时间。在CPU的计算过程中，都是采用最基本的单线程计算，提高了数据的可靠性。

对于矩阵计算的并行思路是将C拆分成许多的work\_group，每个work\_group同时进行。每个work\_group中的各个元素同时进行并行计算。对于cross\_entropy的并行计算思路是将X的K和N维进行拆分成许多work\_group然后同样在每个work\_group中进行在M维上的遍历元素的并行计算。主要有三个地方的并行。第一个是在K，N上并行寻找每个M上的max元素，第二次上计算M上的sum，第三次计算loss。为了进一步加快GPU的计算，没有必要将所有的y都计算出来，只需计算mask所选中的元素进行计算即可。最后特殊处理了block\_size不能整除K和N的情况。CPU和GPU比较大的不同还在于GPU需要将max和sum都存储下来，而CPU的程序不用，GPU的速度虽然更快，但是相应也会占更大的空间。

在本次实验中，我将oneAPI base toolkit下载到了本地，连接了vs，在vs上编写代码，并且同时在本地和在线测试。对于cross\_entropy的性能结果运行如下：





由此可见，GPU的加速性能还是很明显的，达到了实验的目的。

因为本地的显卡性能比较弱，所以加速效果有限，在jupter lab中的加速效果就很明显了。

在实验中遇到的问题：

1. 在线编辑的时候比较卡，所偶想下载到本地。之前电脑是独显直连一直找不到Intell显卡，安装oneAPI base toolkit的时候因为找不到显卡卡了很久。
2. Vs中对get\_global\_id()解释与jupter lab的不同，在刚开始矩阵计算的时候总是出错，GPU计算会有很多0，找错误原因找了很久。后来发现使用get\_local\_id()和get\_group()后发现计算错误消失，发现了问题所在。
3. Cross\_entropy的作业中不知道如何并行实现softmax。一开始想用range<3>来针对X的每个元素并行，后面发现答案是二维的，所以改用对答案的range<2>来并行，在每个并行中额外进行对M维的遍历。就像矩阵计算对于C单个元素也是采用一次遍历，而不是再对每个向量的分量的乘积进行并行计算。这样能大大降低实现难度。
4. Jupter lab的在线编译中的脚本文件最后没有回车，导致没有运行编译的程序，最终一直没有输出。我在这个地方卡了很久，最后参考了jupter lab上的示例的脚本，在脚本后加上一行输出但是没有回车发现了程序有输出，但是脚本的输出不显示，知道了sh脚本一行的结尾是要有回车的。
5. 不清楚cross\_entorpy的每个面上的交叉熵的softmaxs是对每个M维的向量做归一化处理，还是对整个面做归一化处理，最终按照默认多元分类选择的认识对向量做归一化处理。

实验总结：

通过本次实验，我初步了解了并行异构算法，初步掌握了基本的并行程序的写法，以及如何用命令行去编译运行程序。通过这个实验，我极大锻炼了自己的自学能力。还记得刚开始什么都看不懂，连程序都跑不起来，现在已经能初步了解buffer和parallel的使用等。遇到一个新的事物，要善于自学，也许第一眼看到会觉得很难，但是静下心来好好学习几天，就会发现其实没有想得那么复杂。

这次实验让我感受到了异构并行计算的重要性，即使是intel的核显，在大量的并行计算下也能超过强力的CPU的运算能力。因此，未来的编程不再是以往的CPU的单线程运算，CPU多线程，GPU的并行运算日益重要，因此提前适应并行程序是很重要，提前重塑自己编程的定向思维。