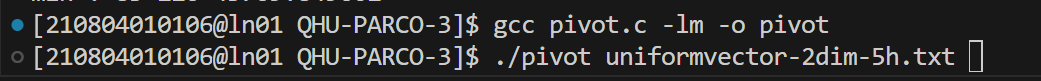
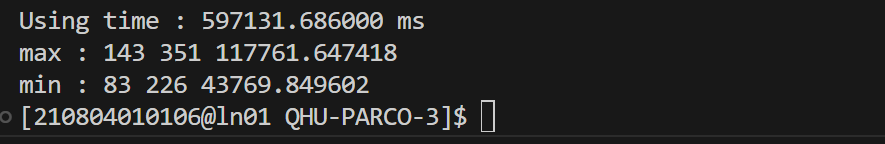
1. **实验准备**

在当前实验中，为了充分利用CUDA并行性能优势并最大化GPU计算资源，首要任务是熟悉CUDA的核函数调用、runtime API和共享内存等关键概念。这将为实现双调排序算法提供基础。通过性能分析，我们可以根据不同的数据特性采用相应的优化手段，以降低程序的运行时间。在实验过程中，深入理解组合数的取数规律和计算公式是必要的。这将有助于更好地设计算法，使其适应GPU并行计算的特性。同时，对双调排序的原理和特点有深入的了解，有助于在GPU上实现高效的双调排序算法。程序的性能分析应该全面而深入，从多个角度对其进行分析。这可能包括GPU核函数的执行时间、数据传输时间、以及共享内存的使用情况等。通过仔细的分析，可以找到潜在的性能瓶颈，并有针对性地采取措施进行优化。

1. **实验步骤**

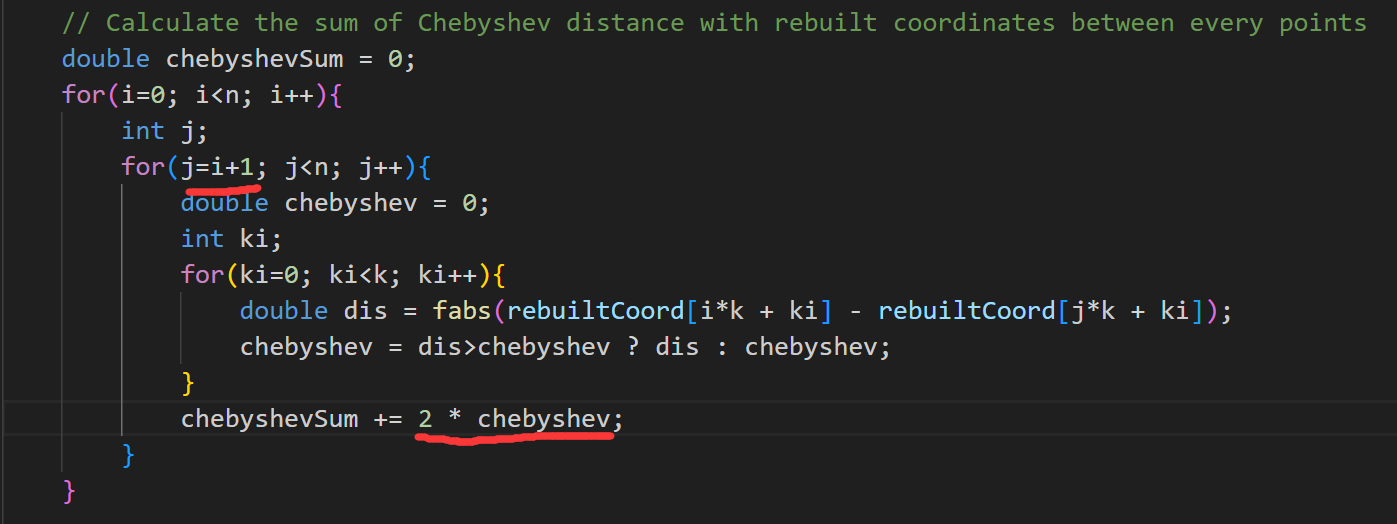
(1). 串行源程序执行时间为：597131.686000 ms



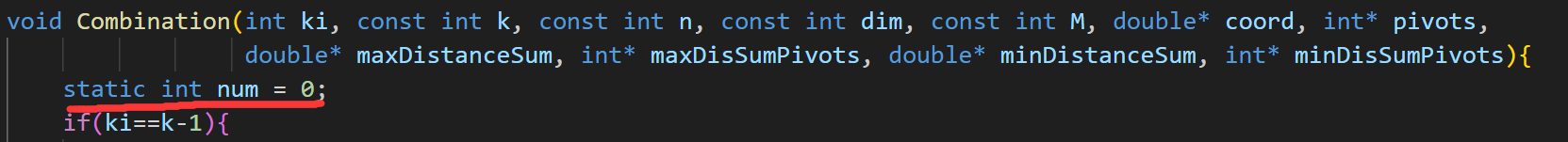


(2). 重复计算消除

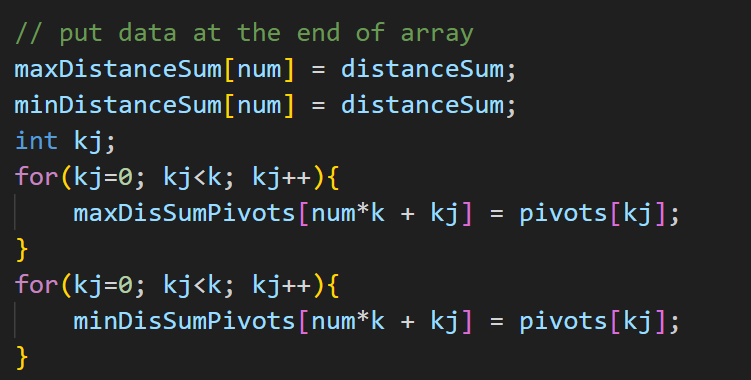
程序中该部分是计算每个点到其余点的切比雪夫距离之和的计算，其中存在着重复数据。举个例子，这个切比雪夫距离之和包含了：（x1,x4）(x4,x1) 的切比雪夫距离，所以我们只需要计算该矩阵的上三角部分，将结果乘以二即可。

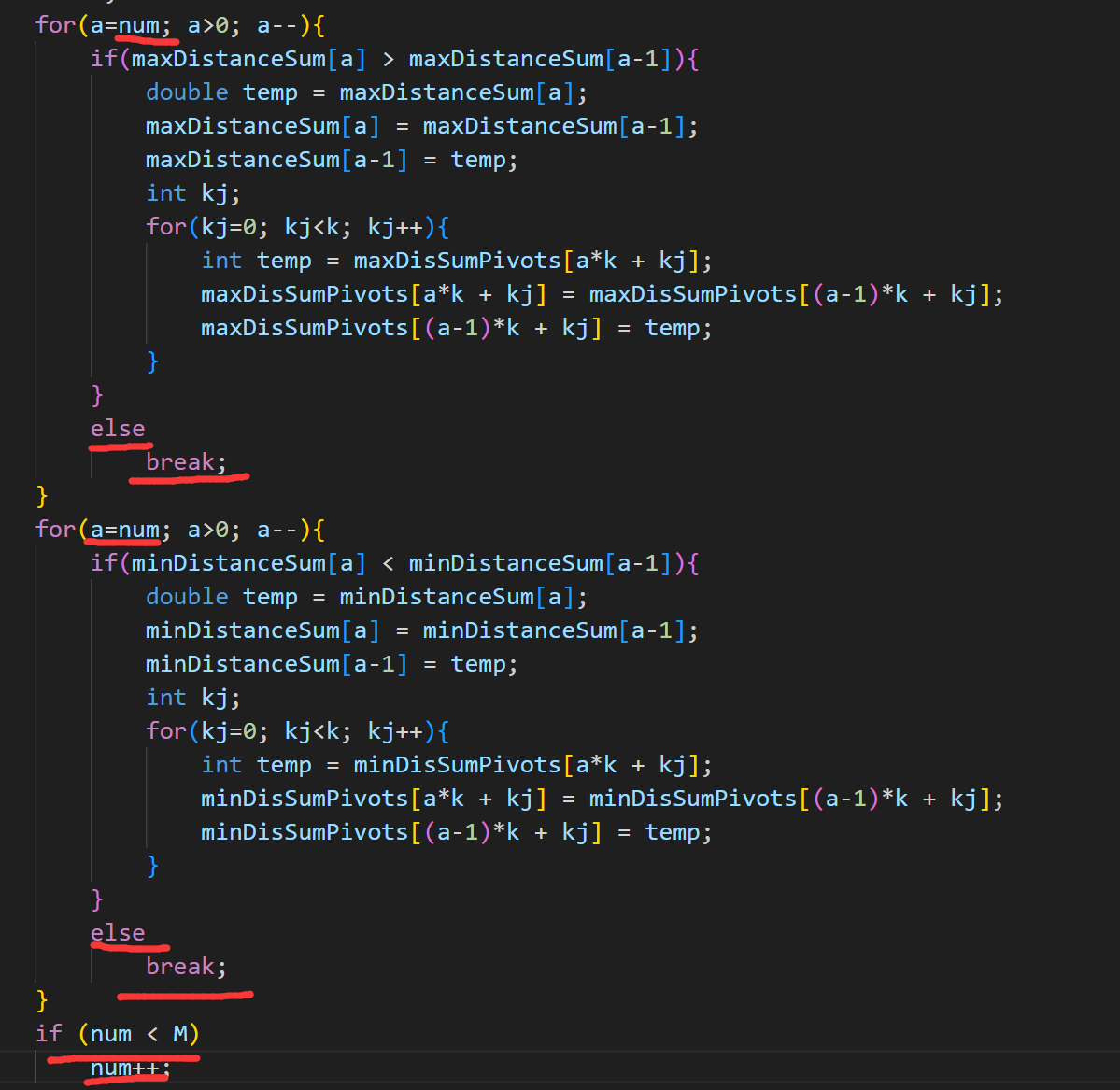


(3). 排序优化

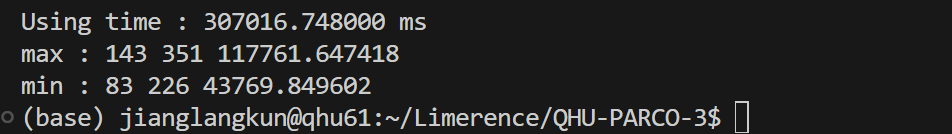


num用于记录已有多少个pivot被存储下来





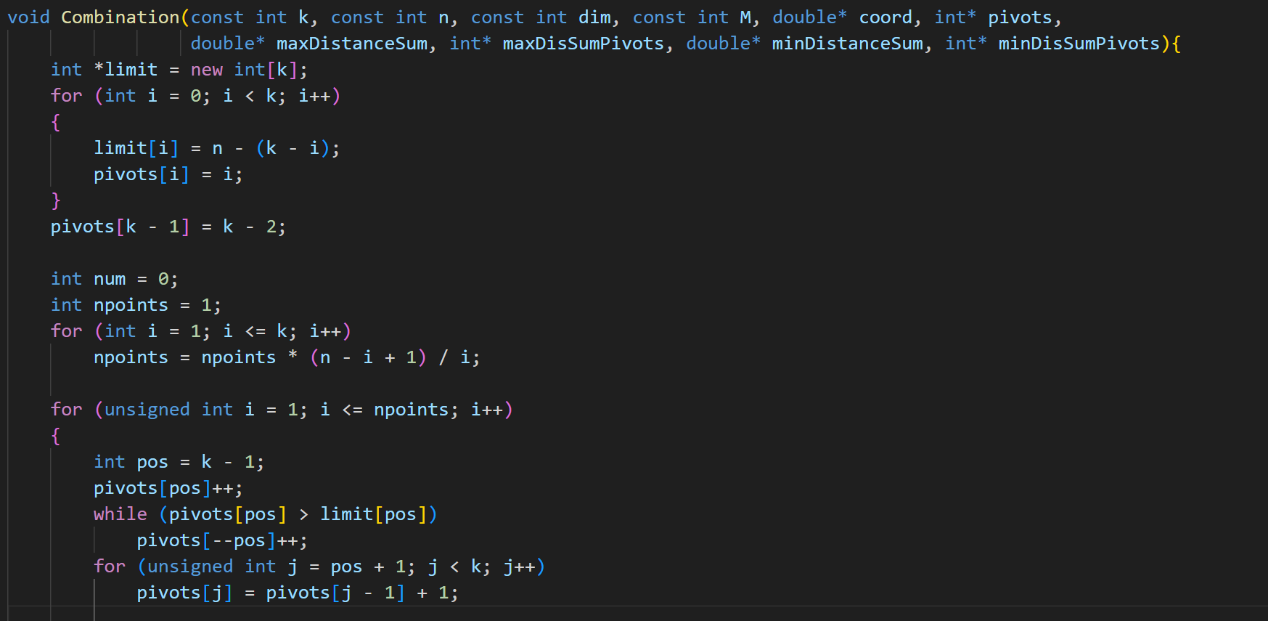
不需要把数据存放到M下标了，因为从num+1到M都是没有数据存放的，直接从num到1进行if条件判断，swap sum值和pivot数组，这样子可以减少if的判断次数，减少无用的数据处理时间。



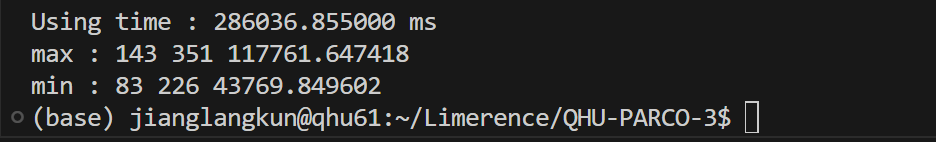
步骤 (2) 和 (3) 累计加速比为：

597131.686000 / 307016.74800 = **1.944948**

(4). 将Combination函数由递归形式，改为非递归形式

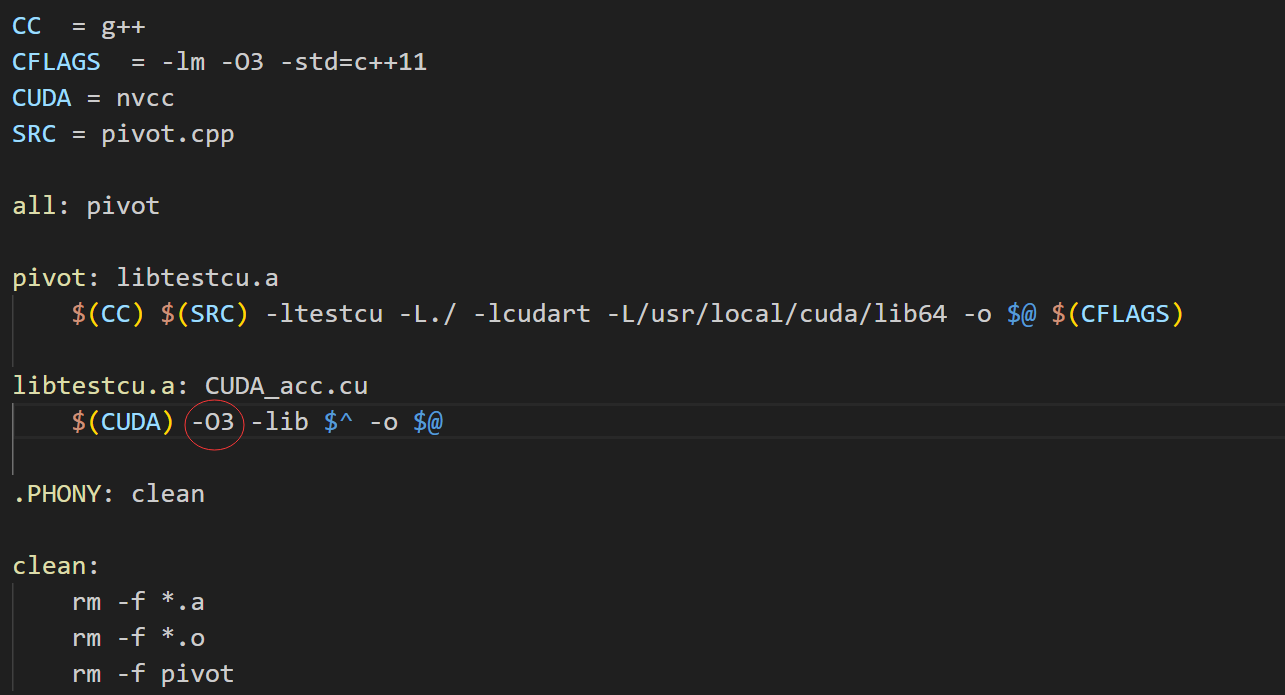


npoints为从n个数中无顺序选择k个数的组合总数，Combination由递归最终改为从1~npoints的大循环。limit数组存放的是k位pivots数组中，每一位最大上限数字（exa: 当n=10，k=5时，第三位不能超过7，如果超过7，后面几位由于必须升序，将没有数字可选）。在每次循环开始前将末位数字加一，往前while遍历，若有超过该位最大限度的，则前一位数字加一，待while结束后，从当前位开始，将后面每一位逐渐递增一，实现pivots数组的更新。

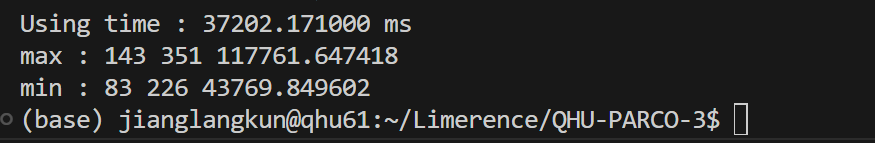


结果只优化了几个百分点，猜想可能是因为总的计算次数没有减少，而且编译器已经做好了递归的优化，因此优化效果不明显。

(5). 编译器选项优化



如g++一样，nvcc也能够进行-O3指定编译器进行程度最高、最激进的优化策略。



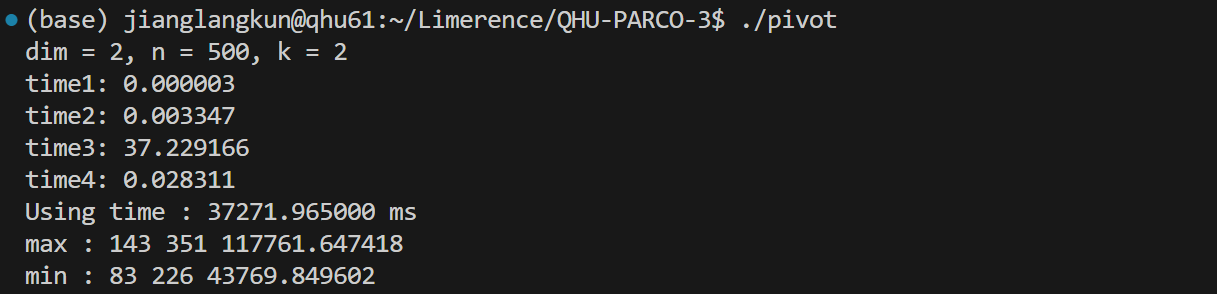
从实验结果来看，优化的程度非常惊人，-O3通过内联函数、循环展开、指令调度等优化手段，使得生成的机器代码更有效率。

综合(4)和(5)的加速比为：

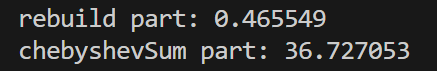
307016.74800 / 37202.171000 = **8.252657**

(6). 简单分析一下程序的运行时间：

通过手写的cpuSecond()函数(调用系统的gettimeofday函数)来分析程序各部分运行的时间，如下：

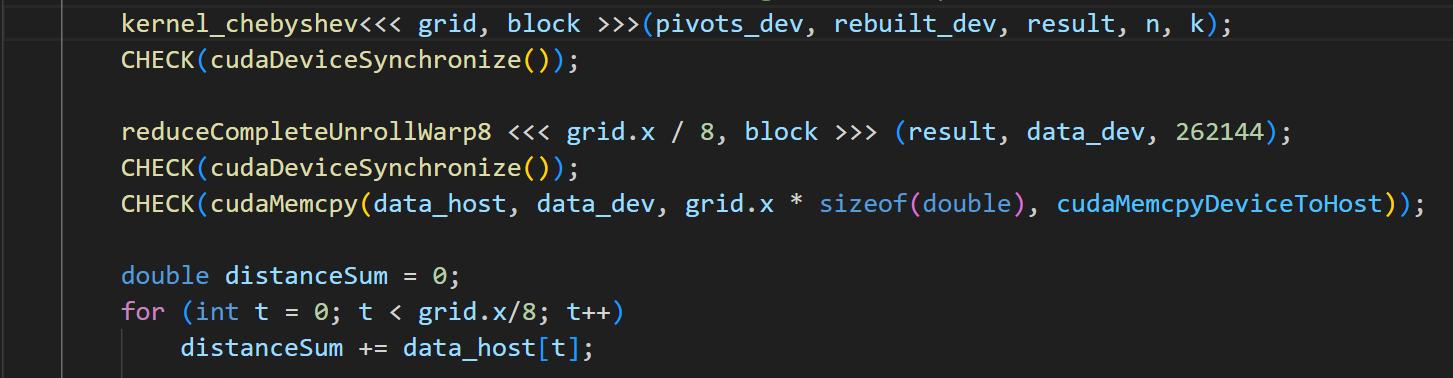


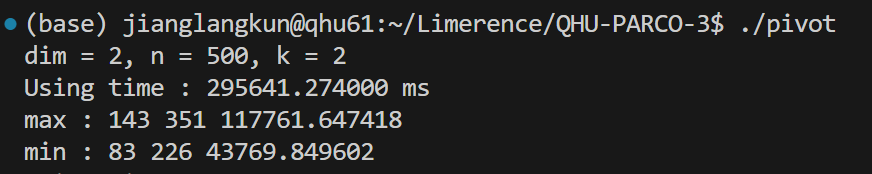
time1~4分别对应大循环开始前的预处理阶段、计算pivots数组、调用SumDistance和排序。由此可见，程序的最大瓶颈在于SumDistance函数，优化程序的最优方式就是降低该部分的运行时间。接下来分析该函数的两个部分。



可以看出，第二部分O(n2 \* k)复杂的计算切比雪夫之和耗时最多，是需要最先解决的瓶颈。

于是，修改SumDistance为两个模块，一个是计算切比雪夫之和(一共需要n \* n个线程，把每一组计算得出的结果存放到result数组中)，接着利用循环展开的归约核函数，将result数组累加到最后的chebyshevSum。

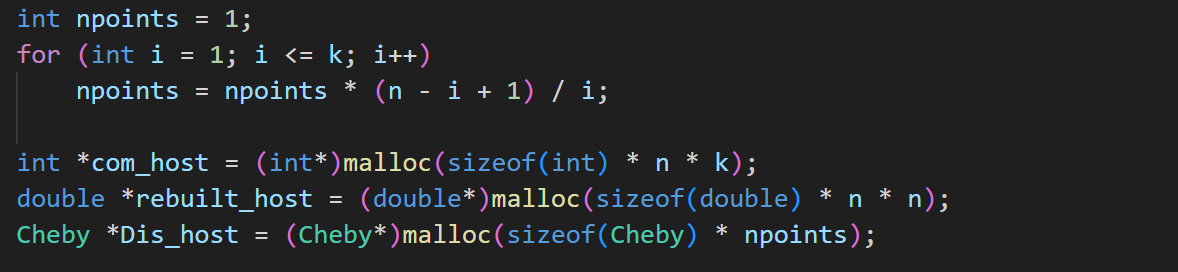


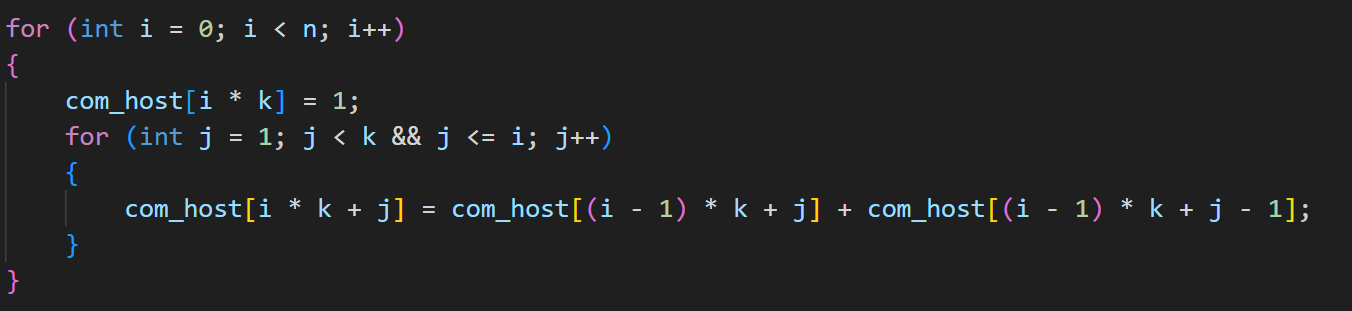


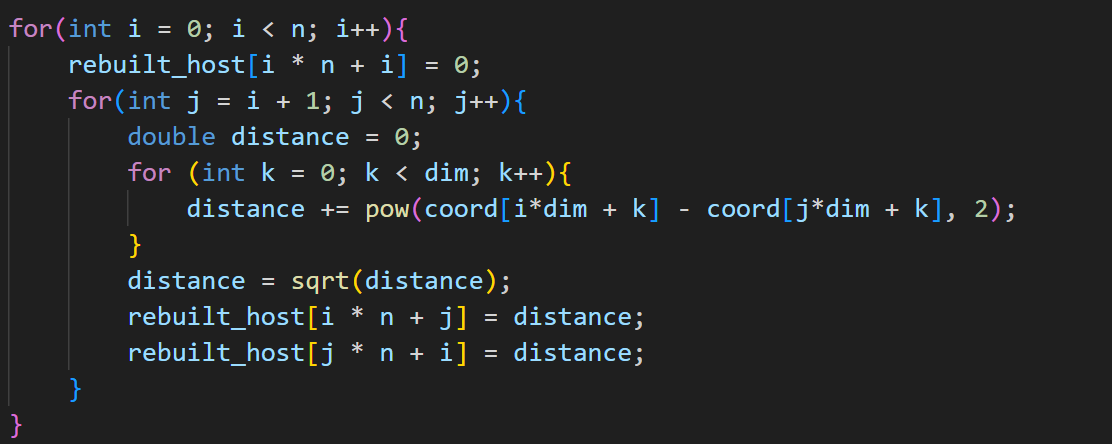
可以看出，cuda并行的加入甚至让程序的运行时间变为原来的十倍。仔细比较原来的代码，每个SumDistance的时间复杂度都为O(n2 \* k)，但是n和k的数量级都特别小，因此串行源程序的运行时间只有0.2966ms，但是利用核函数计算后，数据在gpu和cpu间复制以及启动核函数的花费让程序多了许多额外时间耗费，最终导致时间不增反减，因此需要换一种并行方式。

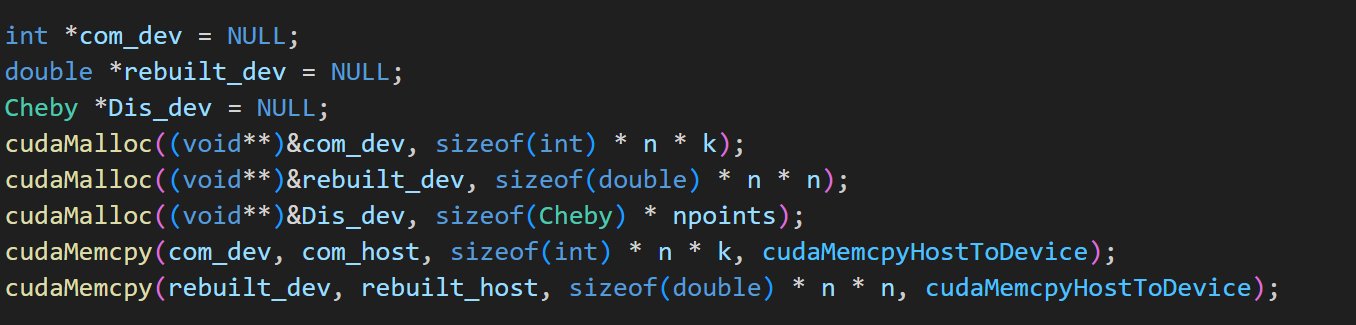
(7). 并行npoints：

从n个数 中选出k个数的组合，每一个pivots序列都对应一个序列号，而根据这个序列号，就可以以O(n)的复杂度加上预先处理好的组合数计算，算出该序列号对应的序列。

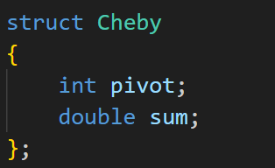


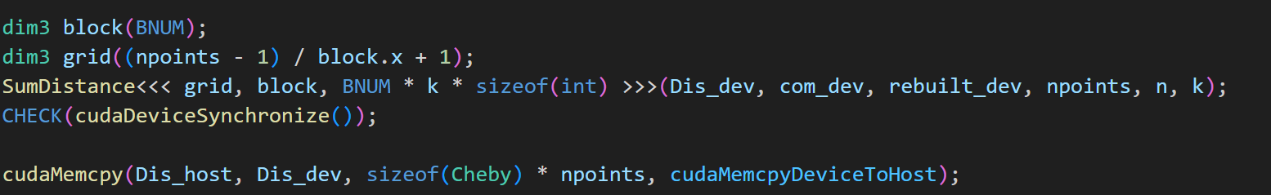


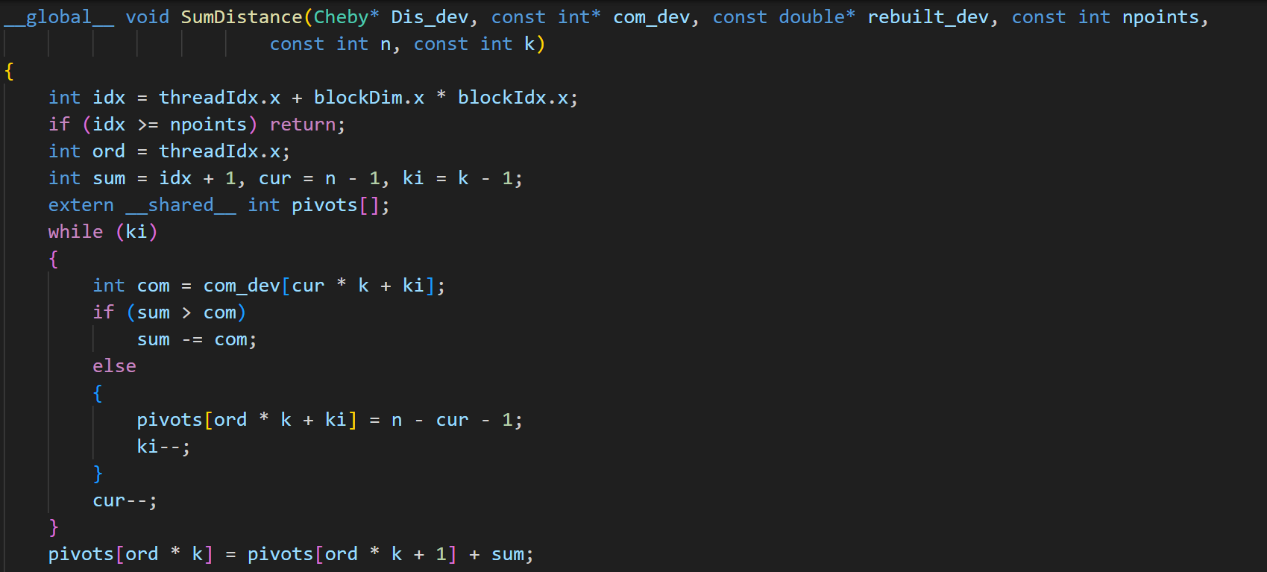




首先先计算总的npoints数(即要处理的线程数)，然后提前计算好组合数和点的距离数组，传入到设备中，以供接下来核函数使用。



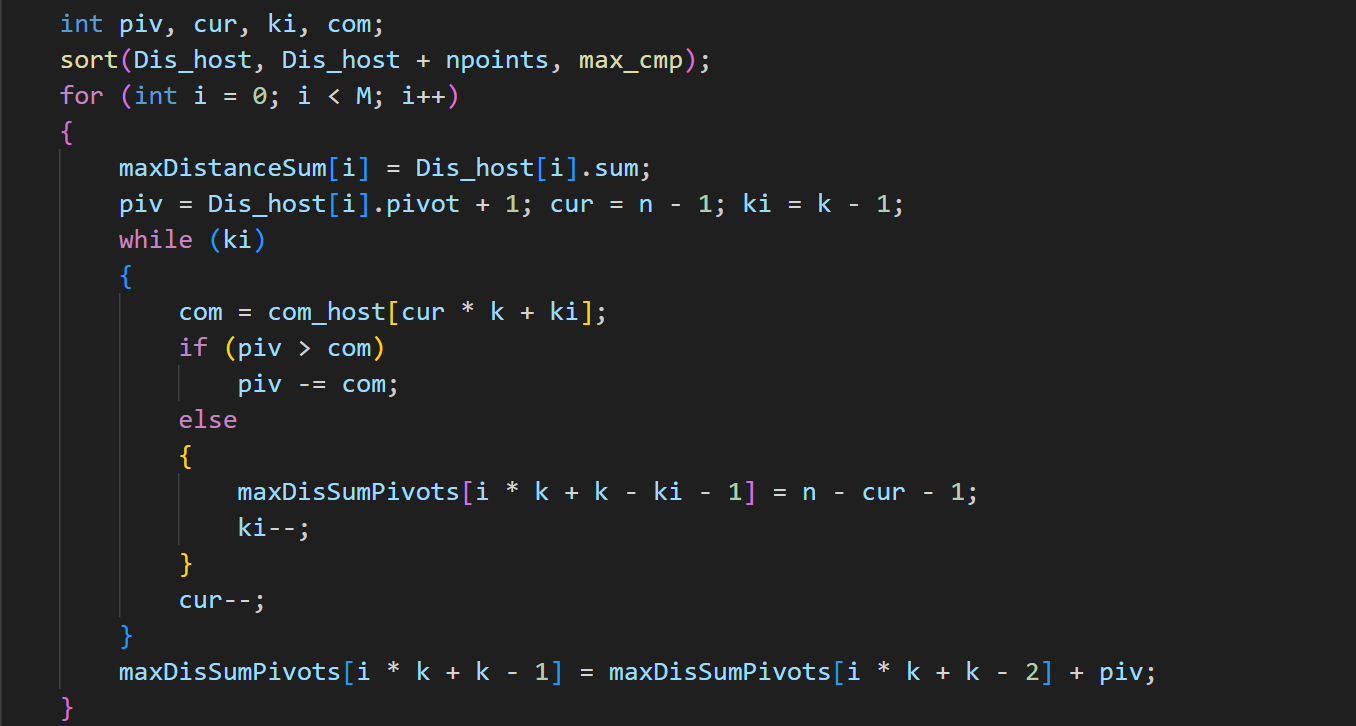




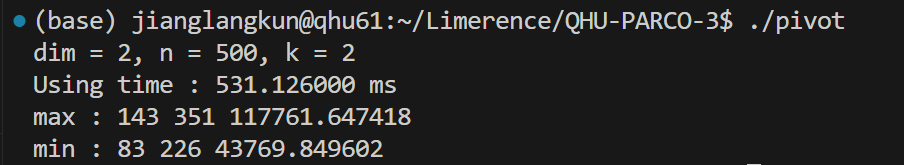


在核函数中，先计算需要用到的序列的k个元素(利用\_\_shared\_\_ memory存储)，然后计算得到切比雪夫之和，存放到Dis\_dev结构体中，idx表示序列号数(后面排序号需要利用序列号还原序列，并输出到文件)，sum表示切比雪夫和。

Tips: 最开始我和核函数内部新建一个pivots数组指针，发现当rebuilt\_dev访问较大内存地址时，就会出现访存错误，后来改为\_\_shared\_\_ memory就不再出现错误。

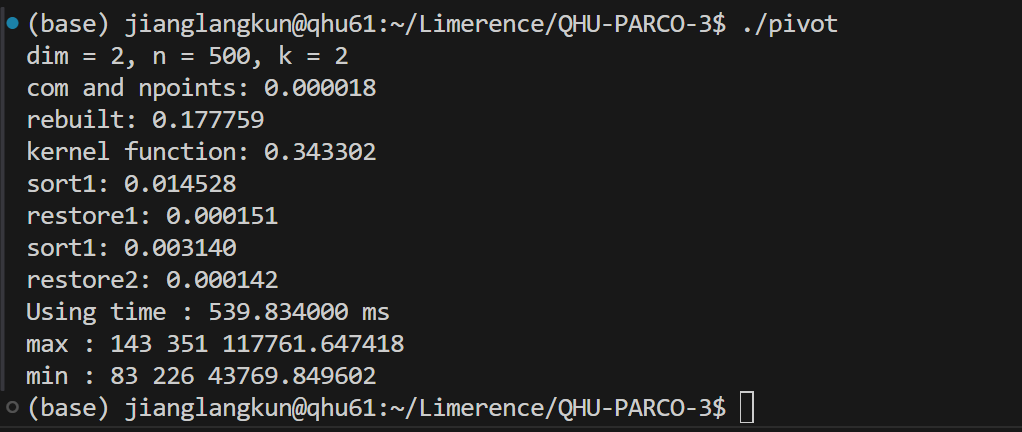


利用计算得来的Dis\_dev结构体数组，先排序然后再还原，将序列存在传入的结构数组中。



加速比为：37202.171000 / 531.126000 = **70.044**

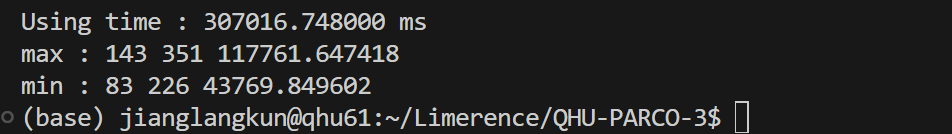
可以看出cuda的加速比还是依然十分显著的，只是程序还有很多可优化的点等待发掘。

(8). 重新分析一下uniformvector-2dim-5h.txt的运行时间

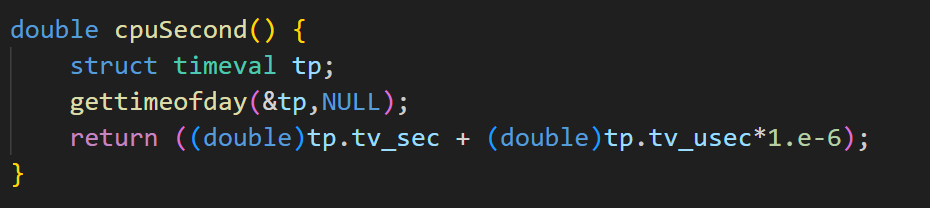
可以看出预处理部分（尤其是点的距离数组耗费了核函数一半的时间），因此需要用cuda对预处理部分进行并行优化。后经过详细的划分查看，发现是由于第一次cudaMalloc需要对gpu进行预热，所以需要远超正常申请内存的时间。

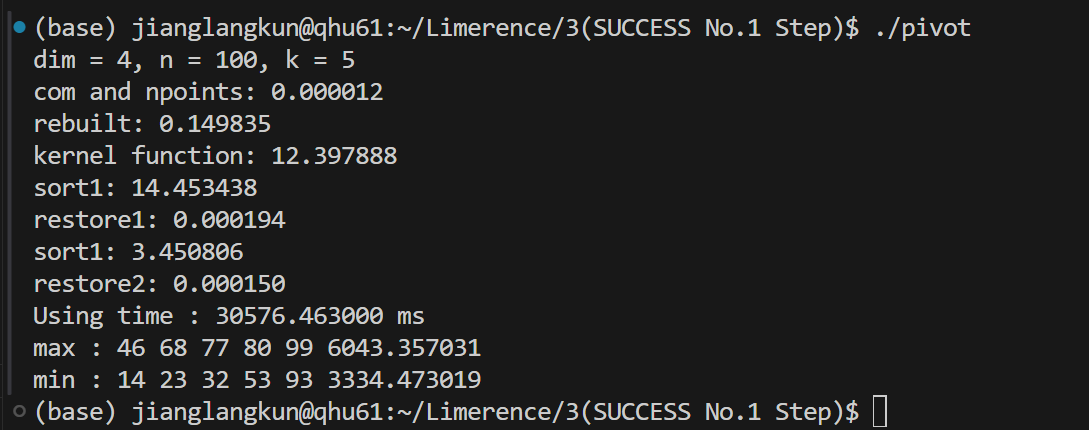
但由于这个是无法避免的时间，所以小样本的情况下无法优化。

(9). 排序优化

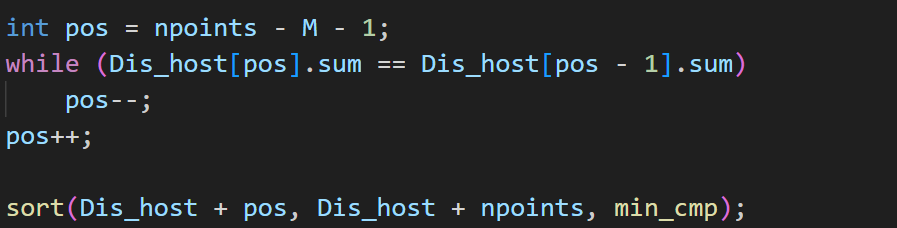


经过初步的cuda优化之后，uniformvector-4dim-1h.txt的时间成功优化到了30秒，通过cpuSecond函数我们可以跟踪各个部分的时间。

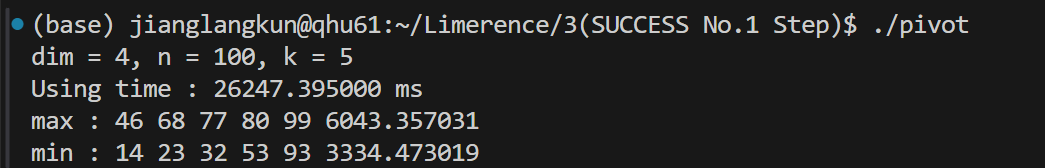




由此可以看出，两次排序由于采用了STL的标准sort排序，因此当数据量达到一定程度后，串行快速排序的时间耗费甚至高于核函数执行的时间，因此需要采取一定的措施，降低排序的时间，达到优化整体时间的效果。



由于第一次排序已经将所有的Sum值从大到小排序，因此只需要考虑倒数M个数据即可，但是由于Sum值相同的数据可能会重复出现，而且序列号也是从小到大排序，因此需要从倒数M个开始找，直到找到Sum值不相同的元素，再进行Sum从大到小的排序，这样子可以使得排序的元素个数大大降低，减少排序时间。

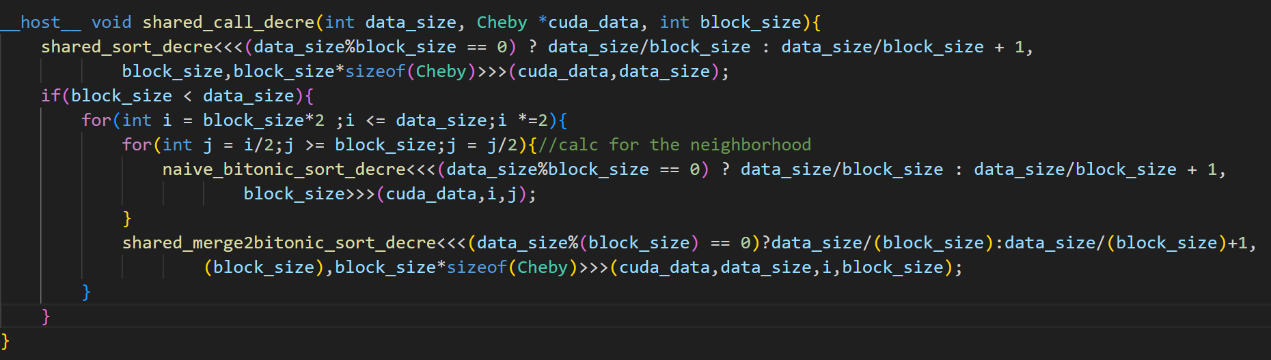


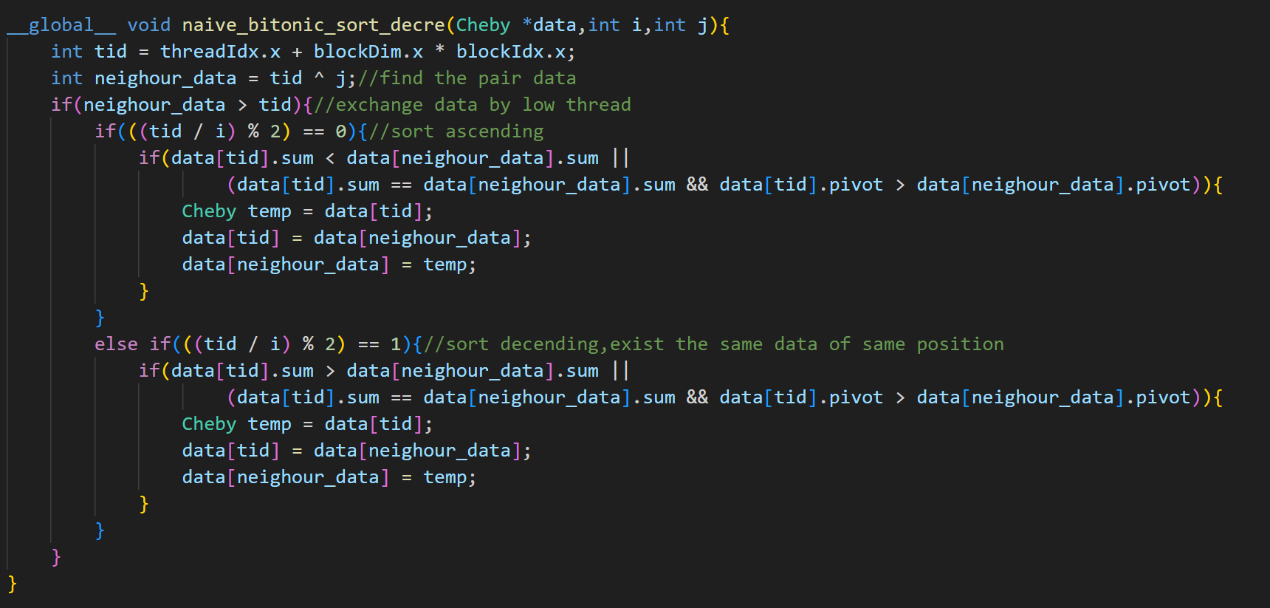
可以看出，这一小小的调整直接减少了3秒多的时间。但是由于第一次排序的时间仍然很久，甚至超过了核函数，需要排序的数据量很大，因此我们可以利用GPU的线程并行和双调排序来对第一个排序进行并行，降低排序时间。

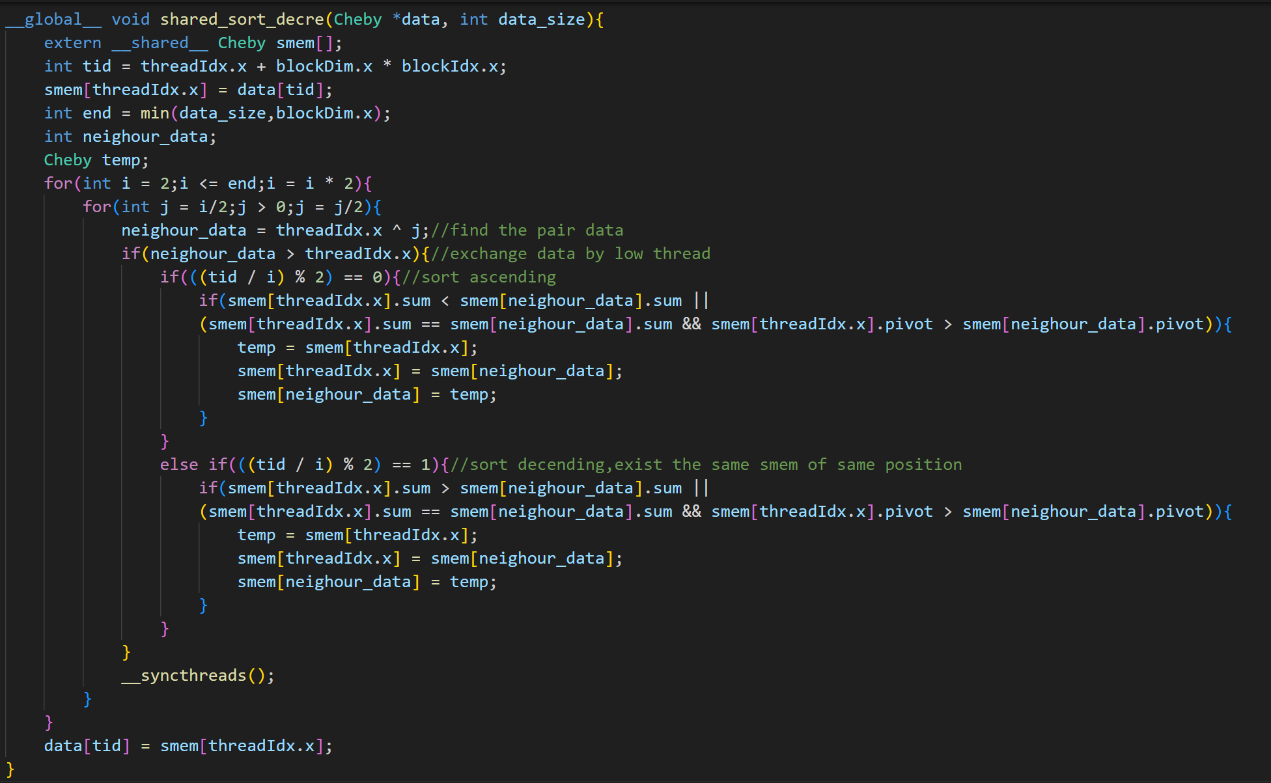
在比较排序算法中，经典的排序算法如快速排序（Quicksort）和归并排序（Mergesort）的平均时间复杂度都是 O(n log n)。这里的 n 是待排序元素的数量。双调排序（Bitonic Sort）是一种并行排序算法，特别适合在并行计算环境中实现。其时间复杂度为 O(log^2(n))。在串行程序中，双调排序的复杂度稍逊于前两者，但是在GPU并行中，由于双调排序可以确保每一个线程在每一步都是可以执行的，排序的总任务会均分到每一个线程上，实现负载的均衡，最大程度地利用计算资源，因此利用双调排序优化排序将大大降低程序的运行时间。



由于双调排序只适用于n为2的整数次幂，正常来说我们会利用padding给数组填充极小值或者极大值，但是这里由于n的数量太大，我们先将后面几个裁剪下来，用STL的标准库函数sort排序，只保留M个，确保序列号的正确性，然后将得到的n为2的整数次幂的数组传入cuda，进行双调排序。

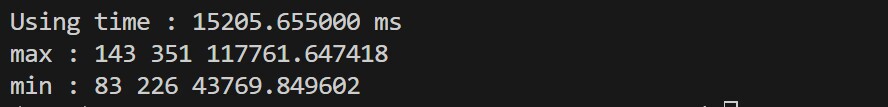






调用shared\_call\_decre后，1、先将无序序列转为步长为block\_size（这里为256）的有序序列这一个过程中的多次核函数调用进行合并，这样可以通过减少kernel调用，从而减少线程分配的代价。2、利用O(log n ^ 2)的循环调用核函数，进行序列的双调排序。3、然后用同样的操作，在第一层循环中，第二层循环后，将固定步长下的迭代同样进行合并，当在固定步长下迭代到j=block\_size时，我们可以将后面的迭代同样的进行合并；这样同样减少了kernel调用的次数，从而减少线程分配的代价。

在两次减少核函数调用次数而使用的函数中，使用共享内存，存下全局内存中序列的值。由于我们将无序序列处理为步长为block\_size的双调序列的过程内置到kernel函数中，因此会在核函数中多次进行不同位置的数据交换。因此，我们可以先将数据预取到共享内存中，在共享内存中对数据进行交换，从而减少访问全局内存的次数。



可以看出，双调排序大大降低了排序的时间，使程序除Sumdistance核函数之外，时间消耗都很低。

1. **实验总结**

**uniformvector-2dim-5h.txt整体的加速比为**：

597131.686000 / 531.126000 = **1124.275004**

**uniformvector-4dim-1h.txt整体的加速比为**：

4681692.461000 / 15205.655000 = **8814.655018**

在本次实验中，我们旨在充分利用CUDA并行性能，最大化GPU计算资源，通过对核函数调用、runtime API、共享内存等关键概念的熟悉，以实现双调排序算法以及优化计算切比雪夫和。通过性能分析，根据不同数据特性采用相应的优化手段，降低程序运行时间。深入理解组合数的规律和计算公式，以及双调排序的原理和特点，有助于在GPU上实现高效的双调排序，来降低大规模数据情况下的排序时间。

在实验步骤中，我们通过不断优化程序，从重复计算消除、排序优化、编译器选项优化、将递归形式改为非递归形式等方面入手，逐步提高程序性能。通过并行计算切比雪夫距离之和、采用CUDA并行的方式降低排序时间等手段，显著提高了程序的效率。通过细致的性能分析，我们定位了潜在的性能瓶颈，并有针对性地进行了优化。

在具体的优化步骤中，我们使用了CUDA的共享内存、线程并行、双调排序等特性，以及编译器选项进行程序性能优化。通过合理的算法设计和CUDA并行计算，成功提高了程序的运行速度。最终，实验结果表明，CUDA并行计算对于处理大规模数据集和复杂计算任务具有显著的优势，通过合理的优化手段，程序在GPU上的性能得到了显著提升。