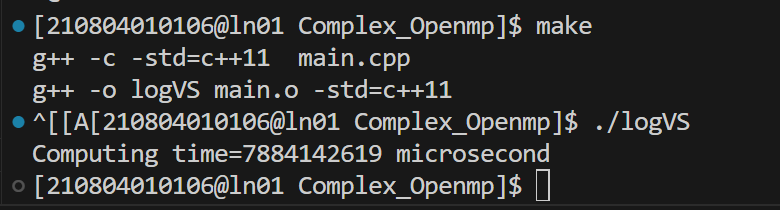
1. **实验准备**

在进行CUDA优化实验之前，首先需要建立对CUDA的基础知识的理解。CUDA是由NVIDIA推出的用于并行计算的编程模型，通常应用于GPU加速。在深入实验之前，对CUDA编程模型、核心概念和相关术语有清晰的认识是至关重要的。随后，深入了解如何编写和组织CUDA内核函数，以及在主机和设备之间高效传递数据也是必要的。在这个阶段，选择适应GPU并行性质的算法是关键，需要考虑数据并行性和任务并行性的平衡。在优化过程中，对CUDA内核函数进行精心的优化是提高性能的关键。以下是三个重要方面需要特别关注：①减少全局内存访问次数，充分利用共享内存。②优化线程束（warp）的利用，最大化GPU并行性。③选择合适的数据类型和进行向量化操作，提高算术运算效率。此外，在优化过程中，应该充分利用CUDA提供的错误检查机制，以确保代码在运行时没有错误。使用CUDA工具进行性能分析是分析程序性能瓶颈的有效手段，有助于明确哪些部分需要进一步优化。

**二、实验步骤**

(1). 串行源程序执行时间为：7884142619 us

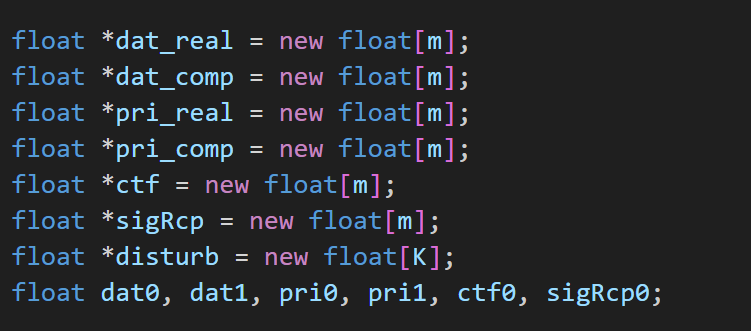


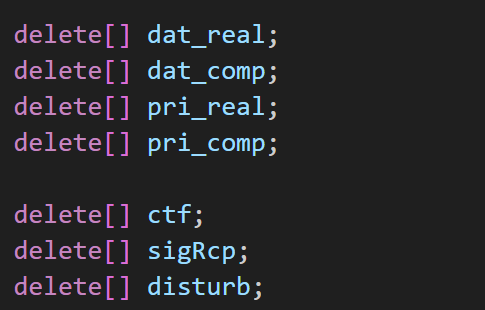
(2). 数据精度优化

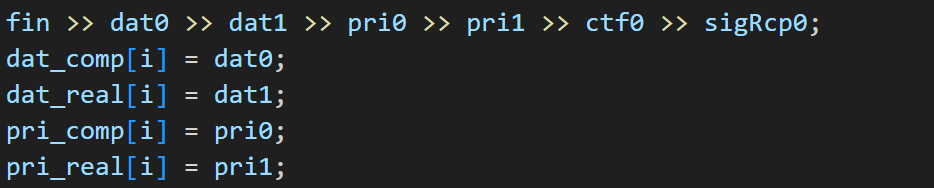
由于题目要求

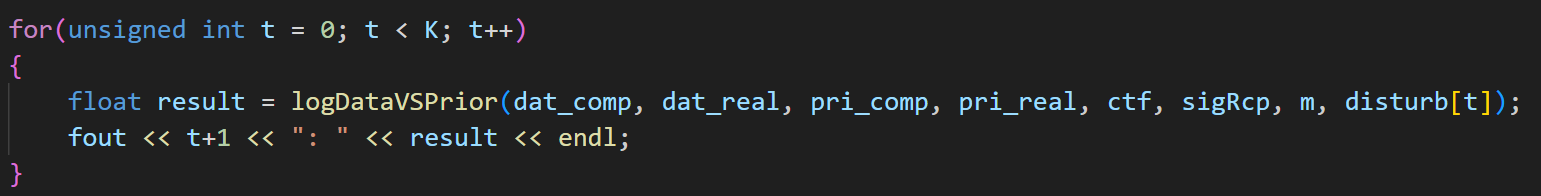
①．因此可以将程序中所有的double类型数据，无论是数组指针还是普通变量，都可以转用float储存，降低数据的精度来换取更小的存储空间和运行速度。

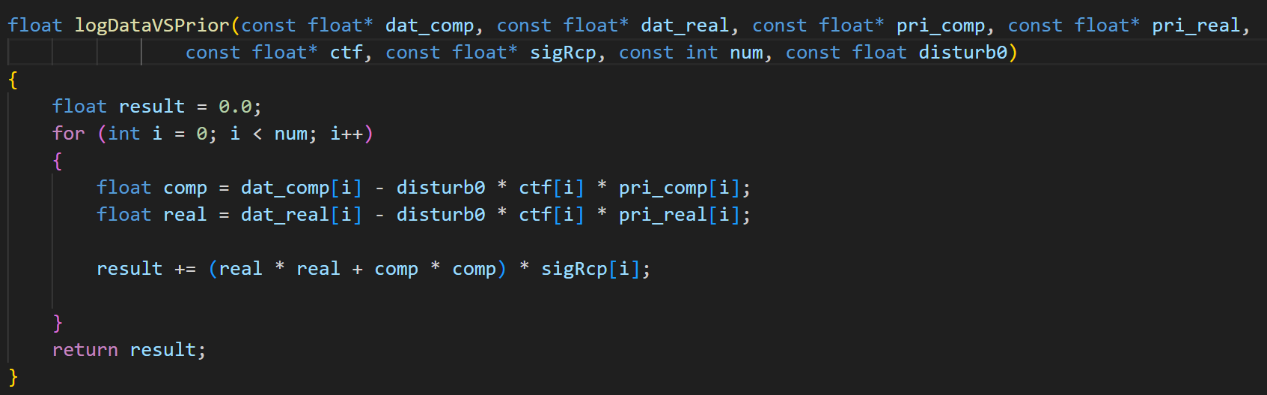
②．同时由于复数的操作会带来更长的运行时间和复杂度，我们可以把dat和pri两个复数数组根据虚部和实部进行拆分，这也是为了后续的AVX512优化铺路（向量化只能操作基本数据类型）。



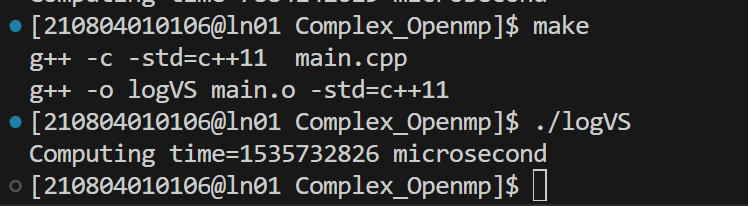








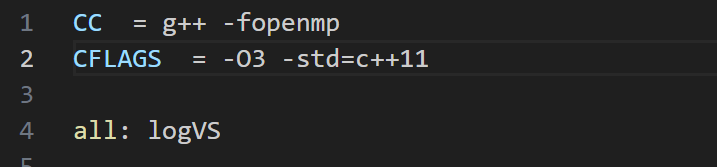
最后运行时间为：



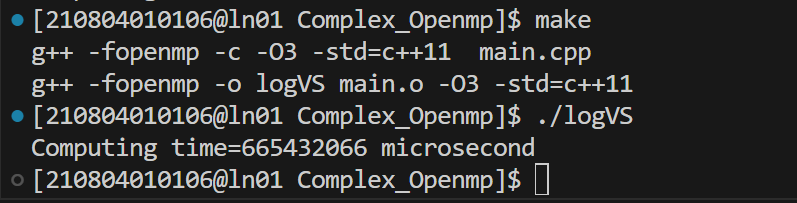
可以得出，加速比为7884142619 / 1535732826 = **5.13**   
64位的double转为32位的float，加速比还是十分可观的。

(3). 编译器指令优化

"O3" 通常表示编译器将进行更高级别的优化，以提高生成的代码的性能。这些优化可以包括内联函数、循环展开、寄存器分配优化、死代码消除和其他一系列性能优化技术。编译器的不同实现可能会在"O3" 优化级别下执行不同的优化。"O3" 通常表示编译器将进行更高级别的优化，以提高生成的代码的性能。这些优化可以包括内联函数、循环展开、寄存器分配优化、死代码消除和其他一系列性能优化技术。编译器的不同实现可能会在"O3" 优化级别下执行不同的优化。



运行时间为：



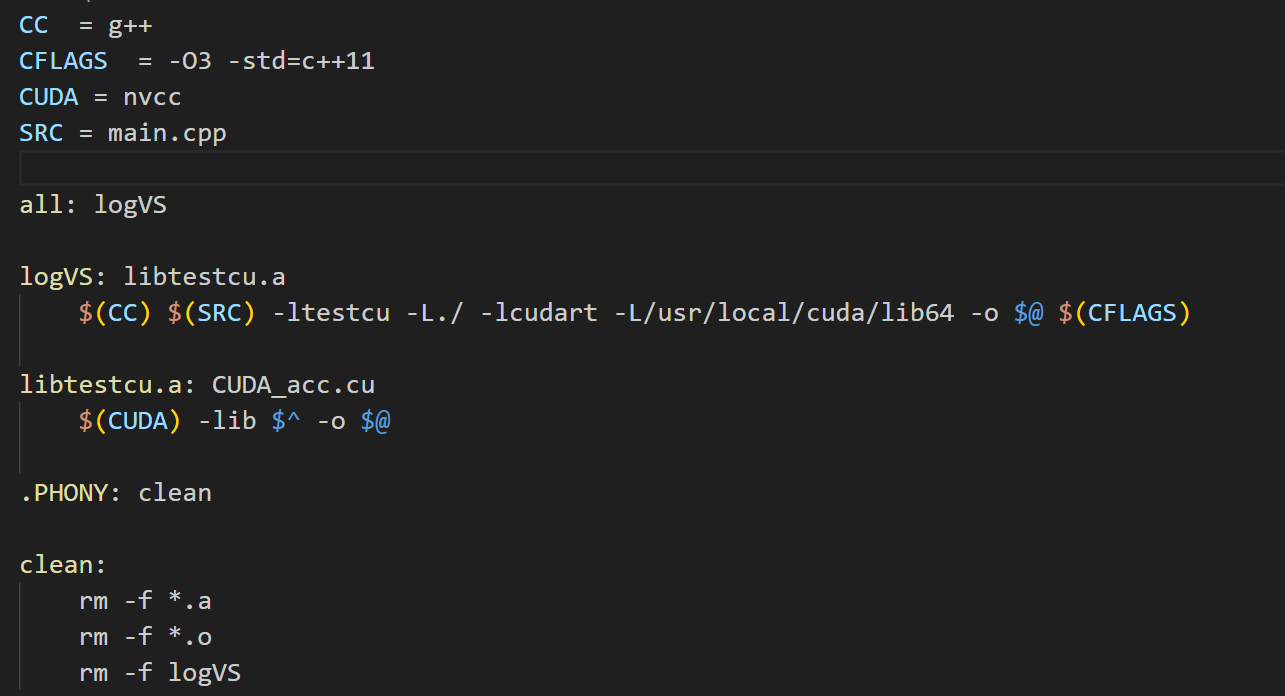
加速比为：1535732826 / 665432066 = **2.31**

可以看出，编译器自动帮我们进行了优化，耗费了更长的时间进行编译，但是在执行时大大加快了对数据的处理速度。

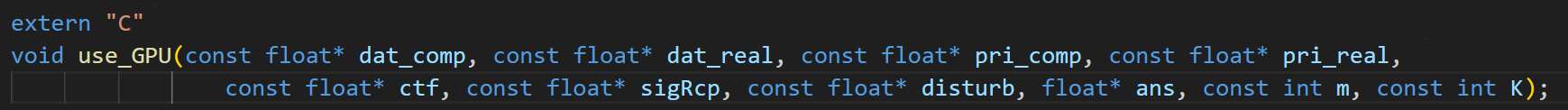
(4). CUDA并行优化

本次实验通过CUDA\_acc.cuh头文件把CUDA\_acc.cu文件main.cpp连接起来，将.cu文件编译为静态库，与main.cpp混合编译，生成可执行文件。

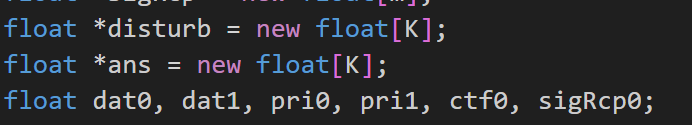
Makefile文件如下：



## TIPS: 在CUDA\_acc.cuh和CUDA\_acc.cu文件中声明use\_GPU函数时需要在void前面添加extern "C" 表示修饰的函数是按照 C/C++ 语言方式编译和连接的。

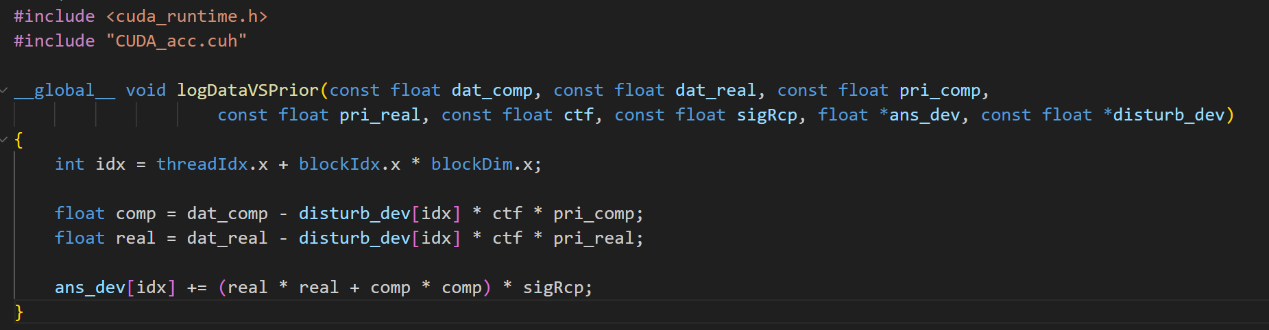


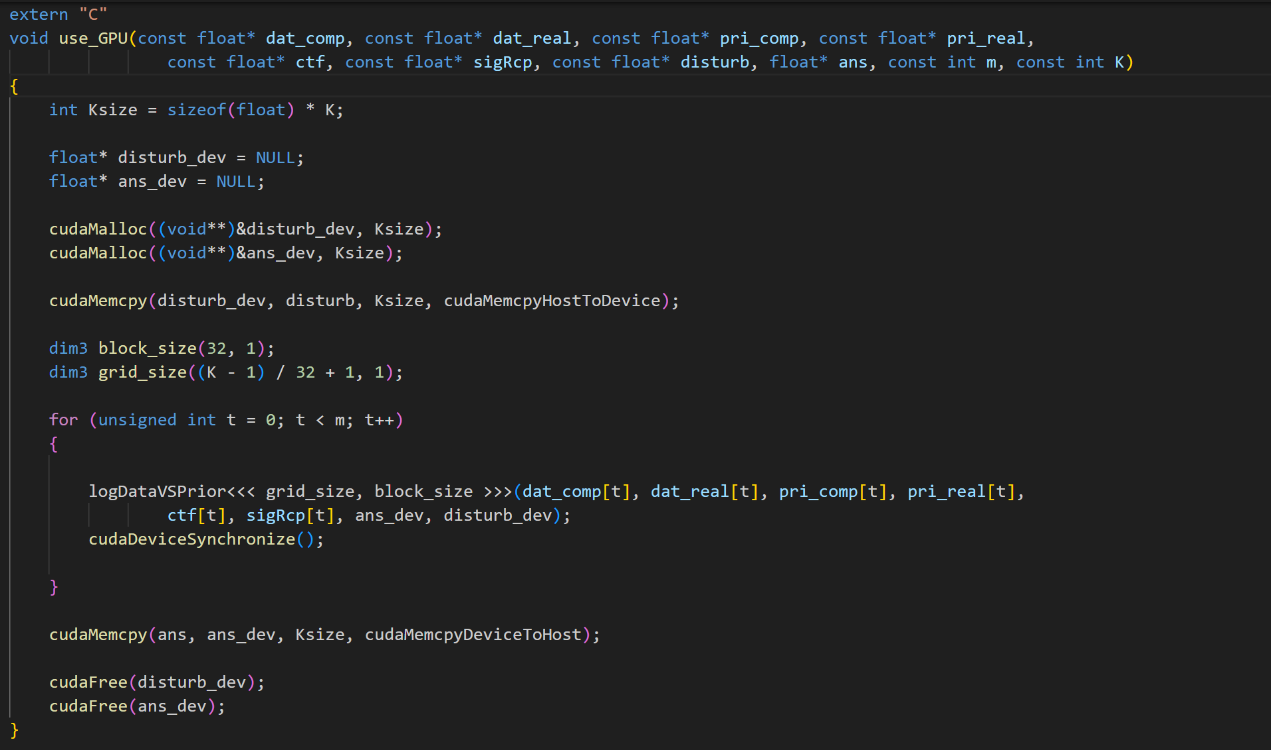
需先新建一个ans数组存放答案，最后计算结束再输出。



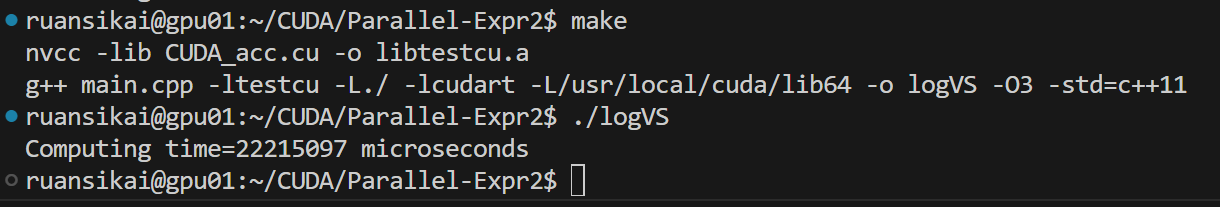
①本次实验的计算部分需要以m为次数遍历for循环，然后在for循环中每次都调用一次核函数（线程总数为K），传入的参数中dat\_comp、dat\_real、pri\_comp、pri\_real、ctf和sigRcp不再是数组，而是数组中下标为i的确切值， ans\_dev和disturb为传入数组，每个线程计算不同的K值产生的不同数据值，并累加到ans\_dev对应位置。

如果以K为次数遍历for循环，然后核函数线程总数为m的话，由于每次执行核函数时，所有的线程（m个）都对答案数组ans\_dev的同一个地址进行累加，由于GPU的高度并行，必须要加锁来确保答案是正确的，但是大量的锁会降低性能，因此，需要采用上一种方案。





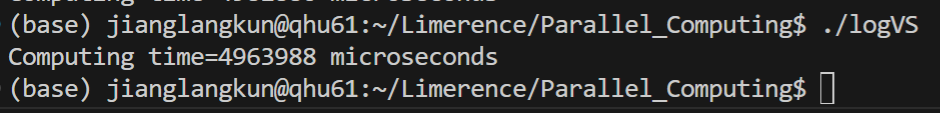
运行时间为：



加速比为：665432066 / 22215097 = **29.954047**

(5). CUDA核函数线程优化

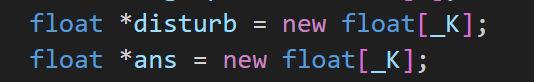
CUDA的核函数的线程块最大可容纳的线程数为1024个，目前只使用了32个，因此可以增加线程数来加快运行速度。



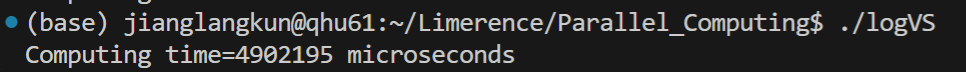
将线程数最大化后，可以最大化利用GPU的计算资源，因此很大程度上提升了运行速度。

(6). 细微的优化

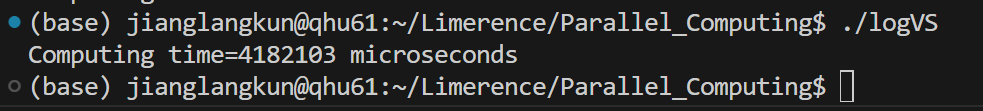


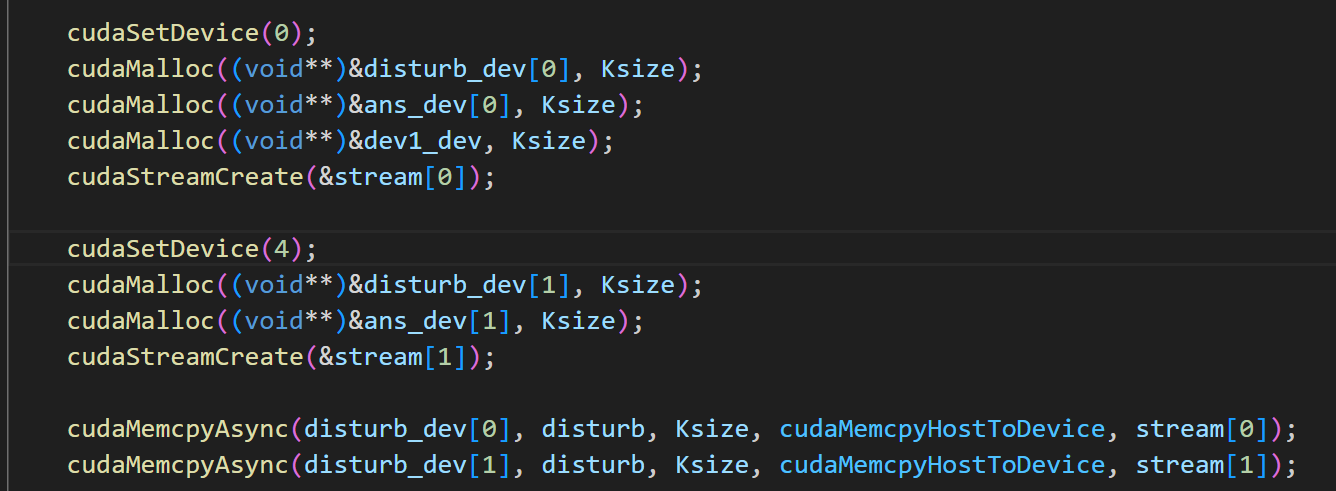


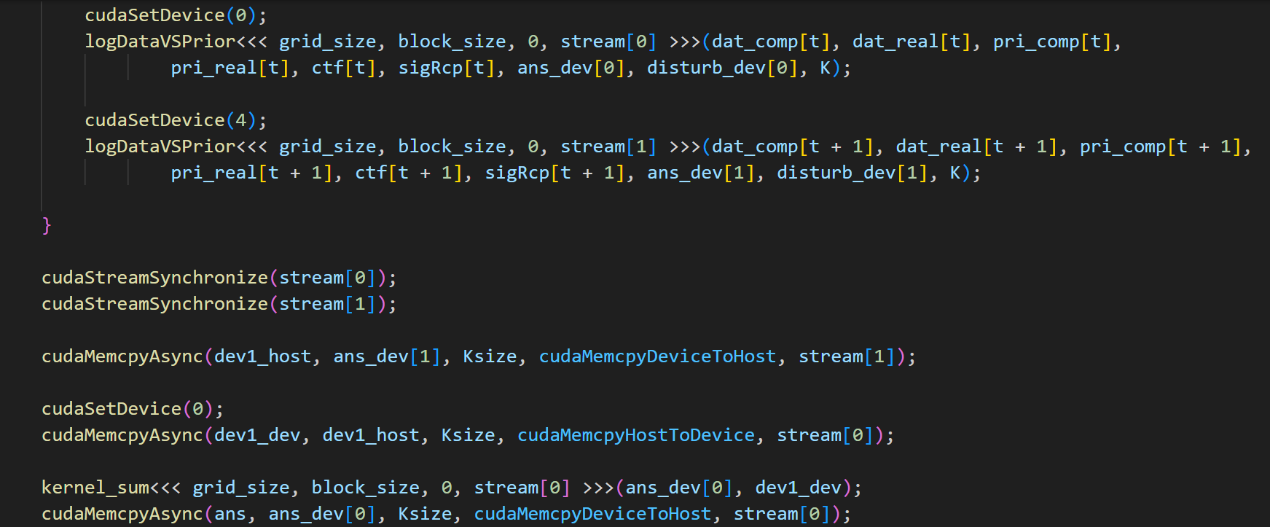
由于K不为1024的整数倍，block数和grid数相乘可能会大于K，因此在每次核函数时都需要额外进行判断，线程数是否超过整体的K数，如果扩充K的数量为\_K = 102400，则可以省去每次核函数的判断合法的分支条件，让活跃的warp数增多，加快程序运行时间。



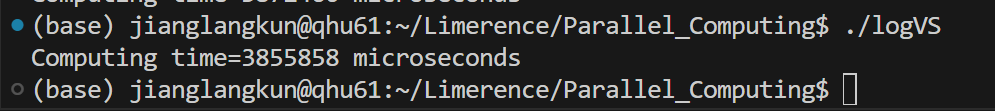
将fout改为fprintf也可以加快程序运行时间







利用CUDA中的流，来使用两张GPU卡设备来对程序进行多GPU并行，加快程序的运行时间，但是由于数据量过小，不能将数据对半分，而是两个不同的核函数执行不同的t值，最后全部执行完毕后，通过一个求和的核函数将两个gpu的数据累加，得到完整的数据。由于种种局限，并没有达到预想中的效果，但是仍旧有略微加速。



加速比为： 22215097 / 3555858 = **6.247465**

**三、实验总结**

**整体的加速比为**：7884142619 / 3555858 = **2217.226509**

在本次实验中，我们对一个串行程序进行了多层次的优化，包括数据精度优化、编译器指令优化、CUDA并行优化和CUDA核函数线程优化等。以下是一些关键点和总结：

1. 数据精度优化： 通过将程序中的 `double` 类型数据转换为 `float` 类型，减小了数据存储空间，提高了运行速度。同时，对复数数组进行拆分，分别处理实部和虚部，为后续的向量化操作做了铺垫。

2. 编译器指令优化： 使用编译器选项 `-O3` 进行高级别的优化，充分发挥编译器在生成代码时的性能优势。这导致编译时间增加，但在运行时提高了数据处理速度。

3. CUDA并行优化：利用CUDA对GPU进行并行计算，将计算密集型任务转移到GPU上，显著提高了程序性能。在此过程中，使用CUDA提供的错误检查机制，确保程序在GPU上运行时没有错误。

4. CUDA核函数线程优化： 通过最大化核函数线程数，充分利用GPU的计算资源，加速了程序的运行。同时，合理设计线程和块的数量，确保GPU的高度并行性。

5. 其他细微的优化： 通过调整代码细节，如将 `fout` 改为 `fprintf`、合理设计数据分布等，进一步提高了程序的运行效率。

6. 多GPU并行： 尝试了多GPU并行计算，虽然由于数据量较小，效果不如预期，但仍然实现了一定的加速。

总的来说，通过多层次的优化，最终实现了在GPU上的高性能计算。实验结果表明，综合考虑数据存储、编译器优化和并行计算等方面的因素，可以显著提升程序的运行效率。然而，实际的优化过程中需综合考虑硬件特性、算法复杂度和数据规模等因素，以达到最佳的性能优化效果。