IPCC 答辩 PPT

各位评委老师大家好，我们是来自中国科学技术大学的鸿雁超算队，很高兴可以进入本次国际并行计算挑战赛的决赛环节

接下来我们将从参赛队伍、应用程序运行的软硬件环境、程序的代码结构，优化方法以及程序最终的运行结果五个方面来介绍我们的参赛方案

我们的参赛队伍名是鸿雁超算队，包括四位参赛队员，分别是付佳伟，朱子琦，沈沛祺和谭邵杰，我们都来自中国科学技术大学计算机学院，并在安虹教授的指导下完成了本次比赛。

在比赛的硬件资源配置上，决赛和初赛一样，都使用北京超算云计算中心的计算资源，每个计算节点均包括一个 64 核 AMD Epyc 7452 处理器，每个队伍最多同时使用 2 个节点，节点间通过 56GB 带宽的 IB 网连接

软件方面，平台提供了 gcc icpc 等编译环境，此外我们还自行安装了 aocc 编译器和性能分析软件 vtune 用于调试和分析。

本次决赛赛题内容是优化一个应用在格点量子色动力学 LQCD 的稀疏线性系统求解程序。程序的源码整体结构如左边的表格所示，主程序调用 load_gauge 和 CGInvert 通过共轭梯度算法求解线性方程

左边这张图是程序 Main 函数执行过程。在主函数里，首先调用 LoadGauge 函数根据进程的 Rank 值并行读取计算所需的 4 维矩阵数据 U，之后运用共轭梯度的数值迭代方法求解大规模的稀疏矩阵问题。在共轭梯度法的使用过程中，调用 Dslash 函数来实现矩阵 M 与输入向量的乘积操作。由于费米子矩阵 M 的特殊性，可以依赖外部输入的组态数据$U_\mu(x)$将其拆分,变成四部分分别计算。当算法迭代满足精度要求后，程序结束。

在拆解完代码结构以后，我们对程序进行了热点分析。由热点分析可知，程序运行过程中主要耗时为 Dslash 函数的计算。这个 Dslash 函数做的实际上就是实现一个矩阵乘向量的操作，在原始代码中已经做了 MPI 并行。进一步测试，发现主要的耗时来自 Dslashoffd 的复数乘法计算，通信在其中被掩盖的较好，因此先对于计算时间进行优化。

根据赛题要求，我们不能改变求解 Mx=b 的算法本质。在这个前提下能做的优化有两个角度：一是减少 Dslash 计算过程用时，二是减少 CGInvert 的迭代次数。因此我们的优化过程也如右图所示，在编译优化的基础上，先使用 CheckerBoarding 预处理减少 CGInvert 迭代次数，然后通过手动向量化、调整任务划分粒度等方法进一步减少计算时间。

我们以 case3 作为测试样例，按照默认方式编译运行得到的 baseline 用时 547s，然后我们分别测试使用 gcc 开启-O3 和使用 intel icpc 编译器，并加入 o3 和 ipo 等优化参数，发现 icpc 优化效果更好，运行时间降低到了 96s，相比基准已经实现了 5~6 倍的加速。因此我们之后的优化也基于 icpc 编译。

编译优化完成后，我们针对格点计算的特性，使用 Chckerboarding 对原问题进行优化。根据计算过程使用矩阵 M 的特性，我们对 M 进行了分解，并将原问题中关于 M 的线性方程求解转化为关于 Mtilde 的线性方程求解上，预处理后只需要原来一半左右的迭代次数就可以达到收敛，相比上一步得到了一倍的性能提升。

之后我们发现计算过程中复数乘法内层循环不存在数据依赖，因此我们考虑使用 avx256 进行向量化，一次同时操作内层循环的 4 个 double 数据。对于间隔分布的数据，使用 _mm256_i32gather_pd 实现非连续地址的读取，并通过将复数的实部与虚部分离的方式减少了浮点乘法的次数。但我们发现这样做了以后程序反而变慢了。

分析运算过程，我们发现我们做的向量化是按原始代码对 T 方向进行的，而在程序执行过程中，T 方向相当于存储的最外层，对访存极不友好，因此我们交换了外层循环 xyzt 的顺序，提高了读写数据时的 Cache 命中率。另一方面，因为复数运算中数据存储本身有间隔，因此我们设置了 buffer，使得运算过程中原本距离较远的数据可以在运算时存放在一起，能够用 avx 指令连续读取 buffer，解决读写数据不连续的问题，提升了访存性能和数据的局部性。

测试过程中我们发现，格点计算任务分配方式同样会对性能产生较大影响，因此采用穷举法来寻找任务划分四个参数的最佳取值。由于之前的手动向量化是按照 x 方向，故对于 subgrid[0]理应取最大值。我们得到的最佳划分方式如下表所示，在最佳排列下，只需要 36.8s 就可以完成 case3 的计算。

最后我们又针对循环中高频出现的重复计算，将其提取出并设为 const int，并将一些重复使用的数据初始化过程提到循环最外层，减少冗余计算。这一步又有 2s 左右的提升。

这是我们最终的优化结果，在 baseline 的基础上我们达到了 15.73 倍的加速。

以上就是我们团队的优化方案介绍，感谢各位老师的观看。