第3次作业说明

1 作业背景

单核矩阵乘法：内存对齐 Alignment，数据预取 Prefetch，循环变化 Loop refactorization，分块 Cache blocking，向量化 Vectorization，循环展开 Loop unrolling 这些优化。熟悉这些方法的原理，可以让我们掌握程序的优化。

2 作业描述

2.1 作业目标

1. 了解一些基本的性能分析工具使用方法。
2. 掌握基础的优化手段。

2.2 作业要求

1. 独立完成代码实现与优化。

2. 提交文件夹命名格式为学号 + 作业编号 + 姓名, 如第一次作业: 2019000000_h1_name，其中包含文件夹 dgemm，和报告 report.pdf。

3. 推荐采用 jusb 来提交任务

4. 注意 DDL，需要在 2022 年 10 月 19 号之前提交

5. 不能将 BLAS 的实现作为作业成果

6. 每位同学独立构建所评测平台上矩阵乘法的 roofline 模型，并用该模型评估代码优化过程。

2.3 作业任务

2.3.1 建立 Roofline 模型

我们以 Intel 官方给出的Roofline Sample程序作为演示示例，流程如下：

1. 首先导入 Intel Advisor 环境
2. 完成待分析程序的编译，生成可执行文件
3. 在后端节点运行应用程序并进行 Roofline 模型分析，生成报告
4. 通过 Intel Advisor Gui 查看报告

2.3.2 单核矩阵乘法

实现矩阵乘法 $C = C + A * B$, 其中, A, B, C 是 $N * N$ 的双精度稠密矩阵

Input: A: 输入稠密矩阵
B: 输入稠密矩阵

Output: C: A * B 的结果

```
1 receive matrix A B C;
2 for i ← 1 to N do
3     for j ← 1 to N do
4         for k ← 1 to N do
5             C(i,j) ⟵ C(i,j) + A(i,k) * B(k,j);
6         end
7     end
8 end
```

Algorithm 1: DGEMM

本次作业，通过教育在线进行下载，需要在集群（111.115.201.25）中运行，需要拷贝到用户对应的目录下，解压之后作业中包含 dgemm-naive.c，dgemm-blas.c，dgemm-blocked.c，三个样例，naive作为最简单的实现，blas 作为本次作业性能的上限，blocked 作为本次作业要改进和优化的基础代码。

编译 dgemm-blocked，可以通过提供的 Makefile 执行：

```
1 make dgemm − blocked
```

测试 dgemm-blocked 的性能，可以通过执行：

```
1 jsub -q normal -n 1 -e error.%J -o output.%J -J benchmark ./test
```

或者：

```
1 jsub -q para -n 4 -e error.%J -o output.%J test
```

更多提交作业命令请参见《青海大学智算管理平台使用手册》

2.4 作业要求

1. 采用双精度运算，运算结果通过以下的正确性验证 (eps 为机器精度)：

$$square\_dgemm(n, A, B, 0) - A * B\| < 3 * n * eps * \|A\| * \|B\|$$

2. 矩阵矩阵乘法的复杂度为 $O(N^3)$，在计算性能指标的时候采用 ($2N^3$ 计算，如果采用了一些非 $O(N^3)$ 而导致通过不了正确性测试，这种情况可以放宽精度的要求，但是需要在作业报告中指出。

3. 开展必要的性能分析，比如某些矩阵规模性能出现明显的降低，可以采用性能分析的工具进行性能分析 (比如: Intel Vtune Amplifier)

4. 测试 N 取 {127,128,129,255,256,257,383,384,385,511,512,513,639,640,641,767,768,769,895,896,897,1023,1024,1025,1151,1152,1153,1279,1280,1281} 的情况，作为矩阵性能的评价标准，会多次排他测试取平均值作为最后的性能成绩。

2.5 作业评分

2.6 Roofline 20%

1. 给出一张 Roofline 模型的结果图，标注自己的模型 (算法) 处在哪个位置 (10%)

2. 分析自己的模型没有达到 Roofline 可能的原因 (10%)

2.7 dgemm 80%

3. 评测 dgemm 的性能结果，按照提交后的性能排序结果，以及代码质量进行打分 (60%)

4. **详细描述**在实现 dgemm 中采取的优化手段，代码对应的部分，以及对应的实验结果，来解释目前取得的性能结果 (10%)。

5. 给出一张完整的实验结果图，描述当前算法的性能，横坐标为矩阵规模，纵坐标为 $Gflop/s$(10%)。

2.8 额外的加分 20%

6. 实现其他的 dgemm 算法，比如Strassen 算法，实现之后请同时提供不同规模矩阵乘法计算的完成时间与标准算法进行对比，为此需针对性修改 benchmark.c；正确性测试上需要通过 benchmark.c 中提供的检验 (20%)。

2.9 作业提示

1. 优化本文已经基本给出。

2.input sensitivity 问题，算法性能的表现和矩阵规模的大小具有很强的相关性，分析 input sensitivity 将能更有效的提升性能。

2. 可以先利用 intel 强大的编译选项来辅助自己完成一部分优化从而减少代码量（能提升 10% 到 20% 的性能），再进行优化。

3. 优化中也会存在一些超参数，可以搜索最优参数，来提升性能。

4. 与助教和老师及时交流。

3 快速上手编程

为了方便大家快速上手编程 (本次作业无需要用到，但是后续的作业都需要)，为大家准备了 3 个 Helloworld 的小 demo 对应编译的 Makefile 在提供的 Helloworld 中:

3.1 omp

多线程，一般用来加速 for 循环，加个注释立即见效，或者最大化利用机器的线程。

```
#include<stdio.h>
#include<stdlib.h>
int main(){
   int nthreads = 0;
   #pragma omp parallel
   {
      if (omp_get_thread_num() == 0) nthreads = omp_get_num_threads();
   }
   printf("We have %d threads!\n" , nthreads);
   #pragma omp parallel
   {
      printf("Thread %d(core %d): Hello World\n",omp_get_thread_num() , sched_getcpu
          ());
```

```
    }
    return 0;
}
```

3.2 mpi

多进程，一般在集群中使用，存在多个计算节点的时候可以有效的加速程序，但是需要控制进程中收发数据，等待数据带来的通信开销，缺点是进程中的通讯开销往往比较大，如何优化通信是提升 MPI 效果的关键。

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc , char **argv){
    int nprocs , rank ;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD , &rank);
    MPI_Comm_size(MPI_COMM_WORLD , &nprocs);
    printf("Hello World! Thread num: %d , Total: %d\n" , rank , nprocs);
    return 0;
}
```

3.3 omp+mpi

多进程中嵌套多线程可以更大的利用计算机资源。

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
int main(int argc , char **argv){
    int nprocs , rank ;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD , &rank);
    MPI_Comm_size(MPI_COMM_WORLD , &nprocs);
    int nthreads = 0;
    omp_set_num_threads(5);
    #pragma omp parallel
    {
        if (omp_get_thread_num() == 0) nthreads = omp_get_num_threads();
    }
    printf("Process %d have %d threads!\n" , rank , nthreads);
    #pragma omp parallel
    {
```

```
        printf("Hello World! Process Num: %d, Thread Num: %d Total Process: %d Total
            Thread per Process: %d\n",rank , omp_get_thread_num() , nprocs , nthreads)
            ;
    }
    return 0;
}
```

4 参考资料

课程将会提供一些资料帮助大家更加深入的了解高性能计算，BLAS 是 一个高性能矩阵运算库，可以参考 GotoBLAS [2]，或者 Franchetti 的 [1] 中 会介绍 dgemm(即本次作业) 的一些思考角度，很适合新手阅读，Rothberg 在 [3] 中会详细介绍分块的策略，这里也给大家提醒一下，矩阵应该如何分 块才能更加有效的利用 cache，以及 cache 的行为（局部性），关于向量化推 荐大家先在课件上看看向量化指令的样例，再去按需查看 Intel AVX2 的向 量化实现文档，可参考的代码框架 1，代码框架 2，代码实现。关于Roofline Model可以参考[4]，或者通过阅读Intel对应的文档1，Intel对应的文档 2，Intel 对应的文档 3

参考文献

[1] Srinivas Chellappa, Franz Franchetti, and Markus Püschel. How to write fast numerical code: A small introduction. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 196–259. Springer, 2007.

[2] Kazushige Goto and Robert A van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):1–25, 2008.

[3] Monica D Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):63–74, 1991.

[4] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.