Lab5-数据级并行实验

实验简介

数据级并行是在计算机体系结构课程中重点讨论的一种并行方式，本实验将在CPU与GPU平台上开展，以矩阵乘法这一经典的例子作为切入点，动手实现不同版本的矩阵乘法实现，探讨不同矩阵规模与划分参数的性能。

实验约定

- 为了避免复杂，本次试验涉及到的矩阵运算的规模$(2^n, 2^n) \times (2^n, 2^n) = (2^n, 2^n)$，每个程序的矩阵规模需要由参数传入程序，以便考察不同规模的矩阵乘法的性能。
- 为了消除编译器优化的影响，在CPU与GPU平台上的编译优化参数可以自行选择，你需要提供 `Makefile` 来辅助编译你的程序。
- 在CPU与GPU平台上的实验的数据类型为 `float32` 。
- 在CPU平台上计时请包含 `time.h` 文件，使用 `clock()` 函数计时；在GPU上请使用 `nvprof` 工具对你写的矩阵乘法kernel的时间进行profiling。
- 在CPU平台上请使用动态内存分配申请矩阵的空间（为一维数组形式），随机数初始化两个参与计算的矩阵A和B，随机初始化的目的是为了验证计算结果的正确性；在GPU上请先在Host端使用动态内存分配申请矩阵的空间（为一维数组形式），随机数初始化两个参与计算的矩阵A和B，随机初始化的目的是为了验证计算结果的正确性。
- 本实验无线下检查环节，请各位同学讲实验源代码与实验报告打包上传。

CPU

任务1-基础矩阵乘法

- 在本任务中，你将实现一个经典的三重嵌套循环的矩阵乘法，大致代码框架如下：

```
#include <xxx.h>

int N = (1 << 8);

void gemm_baseline(float *A, float *B, float *C); // you can use inline function

int main(void) {
  // malloc A, B, C
  // random initialize A, B
  // measure time
```

```
  gemm_baseline(A, B, C);
  return 0;
}
void gemm_baseline(float *A, float *B, float *C) {
  for (...) {
    for (...) {
      for (...) {

      }
    }
  }
}
```

任务2-AVX矩阵乘法

- 在本任务中，你将使用C语言，通过包含 `immintrin.h` 实现一个简单的AVX矩阵乘法。
- 你需要使用CPU-任务1中的基础矩阵乘法验证计算的正确性，验证结束后在性能测量阶段可以不进行正确性的验证。
- 这个简单的AVX矩阵乘法的程序框架仍以三重循环为主体，大致的代码框架如下：

```
#include <xxx.h>

int N = (1 << 8);

void gemm_verify(float *A, float *B, float *C); // you can use inline function
void gemm_avx(float *A, float *B, float *C); // you can use inline function

int main(void) {
  // malloc A, B, C
  // random initialize A, B
  // measure time
  gemm_avx(A, B, C);

  // use gemm_baseline verify gemm_avx
  gemm_verify(A, B, C);
  return 0;
}
void gemm_verify(float *A, float *B, float *C) {
  for (...) {
    for (...) {
      for (...) {

      }
    }
  }
}
void gemm_avx(float *A, float *B, float *C) {
```

```
    for (...) {
      for (...) {
        for (...) {

        }
      }
    }
  }
```

任务3-AVX分块矩阵乘法

- 先前的AVX实现由于仍然是三重循环为主体，访存跨度较大，并未充分利用cache的局部性。
- 你需要使用CPU-任务1中的基础矩阵乘法验证计算的正确性，验证结束后在性能测量阶段可以不进行正确性的验证。
- 在本任务中，你需要调研基于AVX指令集的分块矩阵乘法的实现，并完成代码，你可能需要对B矩阵进行转置。大致代码框架如下：

```
#include <xxx.h>

int N = (1 << 8);

void gemm_verify(float *A, float *B, float *C); // you can use inline function

// you may need to add some additional function parameters to adjust the blocking
strategy.
void gemm_avx_block(float *A, float *B, float *C, ...); // you can use inline function

int main(void) {
  // malloc A, B, C
  // random initialize A, B
  // measure time
  gemm_avx_block(A, B, C, ...);

  // use gemm_baseline verify gemm_avx_block
  gemm_verify(A, B, C);
  return 0;
}
void gemm_verify(float *A, float *B, float *C) {
  for (...) {
    for (...) {
      for (...) {

      }
    }
  }
}
void gemm_avx_block(float *A, float *B, float *C, ...) {
```

```
}
```

实验报告

在CPU部分的实验报告中，你需要体现以下几点：

- 对不同规模的输入（输入的范围由你自己确定，可以不考虑过大的矩阵规模，因为这可能导致性能测量很慢），考察三种实现的性能差异，并简要分析原因。
- 对CPU-任务3中的AVX分块矩阵乘法，探讨不同的分块参数对性能的影响，并简要分析原因。
- 调研并了解CPU平台上其它矩阵乘法的优化手段，在报告中简要总结。
- **（选做，不计入分数）**编写代码调用CPU平台上的BLAS库，将你的实现的性能与BLAS库的性能进行对比，并分析性能差异以及潜在的优化点。

GPU

任务1-基础矩阵乘法

- 在本任务中，你将初步了解GPU的SIMT的编程模型以及GPU中的层级结构，你需要用你了解的知识写一个简单的矩阵乘法kernel，大致代码如下：

```
#include <xxx.h>

#define N (1 << 10)

__global__ void gemm_baseline(float *A, float *B, float *C);
void gemm_verify(float *A, float *B, float *C);

int main()
{
  // malloc A, B, C
  // random initialize A, B
  // cumalloc A, B, C
  // define gridsize and blocksize
  // compute
  // gemm_verify(A, B, C);
  // free mem
}
__global__ void gemm_baseline(float* A, float * B, float* C) {
}
void gemm_verify(float *A, float *B, float *C) {
  for (...) {
    for (...) {
      for (...) {

      }
```

```
26         }
27       }
28     }
```

任务2-分块矩阵乘法

- GPU-任务1中，你完成了一个简单的kernel以实现矩阵乘法，但是其在访存的性能上是糟糕的，所有数据都在 `global memory` 中，加载数据非常耗时。在GPU的存储层次中，有访问相对更快的 `shared memory` ，但其通常较小，不能存储整个矩阵，因此你需要实现一个分块矩阵乘法，以达成更高的性能，代码框架与任务1相似，你需要额外定义一个分块因子 `BLOCK` 来控制矩阵分块的粒度。

实验报告

在GPU部分的实验报告中，你需要体现以下几点：

- 对不同规模的输入（输入的范围由你自己确定，可以不考虑过大的矩阵规模，因为这可能导致性能测量很慢），考察三种实现的性能差异，并简要分析原因。
- 对GPU-任务1中的基础矩阵乘法，探讨不同的 `gridsize` 和 `blocksize` 对性能的影响，并简要分析原因。
- 对GPU-任务2中的基础矩阵乘法，探讨不同的 `gridsize` 、 `blocksize` 以及 `BLOCK` 对性能的影响，并简要分析原因。
- **（选做，不计入分数）**编写代码调用GPU平台上的cuBLAS库，将你的实现的性能与cuBLAS库的性能进行对比，并分析性能差异以及潜在的优化点。