矩阵乘法加速

1 实验目的

本实验旨在通过优化矩阵乘法算法提高计算效率，主要采用缓存命中率改进和多线程计算加速。

2 实验环境

- 编程语言：C++
- 编译器：G++ 9.4.0
- 实验平台：Ubuntu 20.04

参数名	size	assoc	linesize
LEVEL1_ICACHE	32768	8	64
LEVEL1_DCACHE	49152	12	64
LEVEL2_CACHE	1310720	20	64
LEVEL3_CACHE	50331648	12	64
LEVEL4_CACHE	0	0	0

参数名	number
Physical CPU	2
CPU Cores	32
Processor	128

3 实验方法

1. 多线程计算：采用C++ `std::thread` 库实现。
2. 缓存优化：通过分块计算提高缓存命中率。
3. **SIMD**指令集：使用Intel SSE3指令集的 `_mm512_fmadd_pd` 函数进行向量化计算。

4 代码实现

4.1 核心函数描述

1. `multiply_block` 函数
   - 功能: 此函数负责计算矩阵乘法中的一个子块。它使用SIMD（Single Instruction, Multiple Data）指令集进行向量化计算，显著提高计算效率。
   - 参数:
     - `matrix1` 和 `matrix2`: 参与乘法的两个矩阵。
     - `result_matrix`: 存储结果的矩阵。
     - `row`, `col`, `inner`: 分别表示子块在大矩阵中的起始行、列和内部维度。
   - 逻辑:
     - 使用嵌套循环遍历子块的每个元素。
     - 加载结果矩阵当前位置的值到寄存器 `c`。
     - 内层循环使用 `_mm512_fmadd_pd` 指令执行乘加操作，将 `matrix1` 的元素与 `matrix2` 的一行进行点乘，累加到 `c`。
     - 将计算结果存储回结果矩阵。
   - **SIMD**

- `__m512d a, b, c;`: 声明三个变量 `a`, `b`, and `c`， 每个变量可以保存8个双精度浮点数。
- `c = _mm512_loadu_pd(&result_matrix[i][j]);`: 这一行将八个双精度浮点数从 `result_matrix` 加载到 `c` 变量中。 内部名称的 `loadu` 部分代表“加载不对齐”，这意味着数据不需要在64字节边界上对齐。
- `a = _mm512_set1_pd(matrix1[i][k]);`: 这一行将 `a` 变量中的所有八个值设置为 `matrix1[i][k]` 的值。它用于在整个512位寄存器中广播单个双精度值。
- `b = _mm512_loadu_pd(&matrix2[k][j]);`: 与之前的加载操作类似，这将从 `matrix2` 加载8个值到 `b`.
- `c = _mm512_fmadd_pd(a, b, c);`: 执行FMA(融合乘法加)操作。它从 `a` 和 `b`, 中获取每个对应的双精度浮点数，将它们相乘，并将结果与 `c` 中的相应数字相加。这是矩阵乘法的关键部分。
- `_mm512_storeu_pd(&result_matrix[i][j], c);`: 计算结果后，这一行将 `c` 中的8个值存储回 `result_matrix` 中。

2. `matrix_multiplication` 函数

- 功能: 控制整个矩阵乘法过程，并管理多线程。
- 参数:
  - `matrix1` 和 `matrix2`: 参与乘法的两个矩阵。
  - `result_matrix`: 存储结果的矩阵。
- 逻辑:
  - 如果矩阵维度小于块大小，使用朴素方法计算。
  - 初始化结果矩阵为0。
  - 使用多线程，为每个子块分配一个线程。
  - 外层循环遍历结果矩阵的子块。
  - 内层循环调用 `multiply_block` 函数处理每个子块。
  - 使用线程池模式管理线程，以防止创建过多线程。
  - 等待所有线程完成。

5 实验结果[N=M=P=1024]

实验配置	指标项 / GFlops
朴素版本	0.21
朴素多线程	11.56
循环重排	0.53
朴素向量化	0.49
block_size=32 & 向量化	1.16
block_size=64 & 向量化	1.16
block_size=128 & 向量化	1.03
block_size=64 & 向量化 & 16线程	4.39
block_size=64 & 向量化 & 32线程	7.89
block_size=64 & 向量化 & 64线程	26.50
block_size=64 & 向量化 & 128线程	26.14
block_size=64 & 向量化 & 256线程	25.03
CUBLAS	7082.73

6 分析与讨论

1. 在Intel的SSE3指令集中，包含了一个神奇的指令：可以在一个时钟周期内完成两次浮点数的乘法和两次浮点数的加法。在提升了cache命中率和采用多线程计算后，这个指令可以很好的帮助我们进一步地提升矩阵乘法的性能，并且使用非常简单，可以参考[1]、[2]、[3]。请给出使用SSE3指令集前后的性能对比和代码。

   在这个问题中，我使用的是AVX2指令集。性能对比与代码如下：

实验配置	指标项 / **GFlops**
w/ AVX2	21.35
w/o AVX2	11.45

```
// w/ AVX2
void multiply_block(double matrix1[N][M], double matrix2[M][P], double result_matrix[N]
[P], int row, int col, int inner) {
    __m512d a, b, c;
    for (int i = row; i < row + BLOCK_SIZE; ++i) {
        for (int j = col; j < col + BLOCK_SIZE; j += 8) {
            c = _mm512_loadu_pd(&result_matrix[i][j]);
            for (int k = inner; k < inner + BLOCK_SIZE; ++k) {
                a = _mm512_set1_pd(matrix1[i][k]);
                b = _mm512_loadu_pd(&matrix2[k][j]);
                c = _mm512_fmadd_pd(a, b, c);
            }
            _mm512_storeu_pd(&result_matrix[i][j], c);
        }
    }
}

// w/o AVX2
void multiply_block(double matrix1[N][M], double matrix2[M][P], double result_matrix[N]
[P], int row, int col, int inner) {
    for (int i = row; i < row + BLOCK_SIZE; ++i) {
        for (int j = col; j < col + BLOCK_SIZE; ++j) {
            for (int k = inner; k < inner + BLOCK_SIZE; ++k) {
                result_matrix[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}
```

2. 为什么有时候多线程性能反而不如单线程？在什么情况下会导致这样的情况？

   多线程性能不如单线程的情况通常是由于以下几个因素导致的：

   a. 线程间的竞争：当多个线程访问共享资源时，会涉及到竞争条件。如果没有正确地管理竞争条件，会导致性能下降，因为线程会频繁地争夺资源，导致等待和上下文切换的开销增加。

b. 资源限制：多线程可能会消耗更多的系统资源，如内存和CPU。如果系统资源不足以支持多个线程同时运行，性能可能会下降，因为线程之间会争夺有限的资源。

c. 负载不均衡：在某些情况下，多线程可能导致负载不均衡，其中一些线程可能会比其他线程更快完成任务，导致一些线程处于空闲状态，浪费了系统资源。

d. 同步开销：使用多线程时，通常需要进行线程同步操作，如锁定和信号量等。这些同步操作会引入额外的开销，并且可能会导致性能下降。

e. 数据局部性：多线程程序可能会导致缓存争用，因为多个线程试图同时访问相同的缓存行。这会导致缓存未命中，从而增加了内存访问延迟。

3. 矩阵乘法是否会出现频繁的内存缺页？如何解决这样的问题?

   矩阵乘法可能会导致频繁的内存缺页，特别是当矩阵的大小非常大时。这是因为矩阵乘法涉及到大量的数据访问，如果数据不能完全放入内存中，就会导致内存缺页，从而降低性能。

   为了解决矩阵乘法中的内存缺页问题，可以考虑以下几种方法：

   a. 数据局部性优化：通过重排矩阵的数据布局，使得矩阵乘法的数据访问模式更加局部化，减少缺页的发生。这包括使用缓存友好的数据结构和算法。

   b. 分块矩阵乘法：将大矩阵分成较小的块，并使用分块矩阵乘法算法，以减少内存访问的跨度。这可以减少缺页的发生，并提高性能。

   c. 矩阵压缩：对于稀疏矩阵，可以使用矩阵压缩技术来减少内存占用，从而减少缺页的发生。这包括使用稀疏矩阵存储格式和算法。

   d. 内存优化：增加系统的物理内存或使用更高速的内存设备可以减少内存缺页的问题。

   e. 并行计算：使用多线程或分布式计算来加速矩阵乘法，可以减少单个线程的内存占用，从而减少内存缺页的发生。

4. 尝试使用GPU进行矩阵运算，CPU和GPU运算各有什么特点？为什么GPU矩阵运算远远快于CPU?

   我编写了如下的代码，使用nvcc multiply.cpp -std=c++14 -Xcompiler -pthread -lcublas -o main编译，测试了在N=M=P=1024下GPU计算矩阵运算的性能（见实验结果）。
   从实验结果可以明显看出在矩阵运算上，GPU运算性能远超CPU。在本次实验的CPU代码中，访存优化是一个重要的提升性能的方法，对于GPU而言在显存的带宽参数上是远远超过CPU的，而且GPU能够进行大规模的并行计算，这也是比CPU更加适合矩阵乘法的原因。

   GPU在进行大尺寸的矩阵运算时，由于上述原因，速度是远快于CPU的，但是也有一些情况，使用GPU运算的性能可能比用CPU还要慢，因为调用GPU运算需要先将内存中的数据搬运到显存中，当矩阵尺寸比较小时，转移数据的操作可能成为整个程序的瓶颈，带来的影响可能超过GPU并行计算带来的增益。

Parameter	**RTX 3090**	**i9-12900K**
Architecture	Ampere	Alder Lake
Process Size (nm)	8	10
Cores (Shading Units for GPU)	10,496	8 Performance + 8 Efficiency (16 total)
Memory Type	GDDR6X	DDR4/DDR5
Memory Size (GB)	24	Depends on motherboard
Memory Bus Width (bit)	384	64/128 (DDR4/DDR5)
Bandwidth (GB/s)	936 (GDDR6X 24GB at 19.5Gbps)	51.2 (DDR4-3200) / 76.8 (DDR5-4800)
Release Date	September 1, 2020	November 4, 2021

```
#include <cublas_v2.h>
```

```
#include <cuda_runtime_api.h>
#include <cuda.h>

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>

#define N 1024
#define M 1024
#define P 1024

#define IDX2C(i,j,ld) (((j)*(ld))+(i))

static void init_matrix(double *matrix, int n, int m)
{
    int length = n * m;
    for(int index = 0; index < length; ++index, ++matrix) {
        *matrix = rand() % 10;
    }
}

void matrix_multiplication(double *matrix1, double *matrix2, double *result_matrix) {
    cublasHandle_t handle;
    cublasCreate(&handle);

    double *d_matrix1, *d_matrix2, *d_result_matrix;
    cudaMalloc((void**)&d_matrix1, N * M * sizeof(double));
    cudaMalloc((void**)&d_matrix2, M * P * sizeof(double));
    cudaMalloc((void**)&d_result_matrix, N * P * sizeof(double));

    cudaMemcpy(d_matrix1, matrix1, N * M * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_matrix2, matrix2, M * P * sizeof(double), cudaMemcpyHostToDevice);

    double alpha = 1.0;
    double beta = 0.0;

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

    cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, P, M, &alpha, d_matrix1, N,
d_matrix2, M, &beta, d_result_matrix, N);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
```

```
    float elapsed_time;
    cudaEventElapsedTime(&elapsed_time, start, stop);
    elapsed_time /= 1000.0f;

    double gflops = 2.0 * pow(10.0, -9) * N * M * P / elapsed_time;
    printf("Performance: %f GFLOPS\n", gflops);

    cudaMemcpy(result_matrix, d_result_matrix, N * P * sizeof(double),
cudaMemcpyDeviceToHost);

    cudaFree(d_matrix1);
    cudaFree(d_matrix2);
    cudaFree(d_result_matrix);
    cublasDestroy(handle);
}

int main() {
    double *matrix1 = (double*)malloc(N * M * sizeof(double));
    double *matrix2 = (double*)malloc(M * P * sizeof(double));
    double *result_matrix = (double*)malloc(N * P * sizeof(double));

    init_matrix(matrix1, N, M);
    init_matrix(matrix2, M, P);

    matrix_multiplication(matrix1, matrix2, result_matrix);

    free(matrix1);
    free(matrix2);
    free(result_matrix);

    return 0;
}
```

7 遇到的困难与解决方案

- 问题描述：在使用AVX指令集进行优化计算时，我遇到了很多问题，首先是不知道应该使用什么指令实现自己的想法，因为对于这样底层的代码缺乏实践经验，这些问题解决起来是比较复杂的。然后就是在调试代码时遇到的各种错误，比如编译时没有加上`-mavx512f`选项，或者变量类型没有对上，这些问题主要是由于对AVX指令比较陌生，解决起来并不复杂。
- 解决方案：对于上面提到的第一类问题，我主要通过查阅文档、学习教程以及与ChatGPT交流获取解决方法。对于上面提到的第二类问题，主要通过观察报错信息以及在stackoverflow等bug交流网站查询解决。

8 总结

- 主要发现：
  - 通过分块的方式可以同时利用缓存的时间局部性与空间局部性。

    - 时间局部性：在矩阵乘法中，分块意味着一旦一个数据块被加载到缓存，它会在计算该块内所有元素的乘积时被多次使用。
    - 空间局部性：分块操作通常是连续的内存访问，这意味着当一个数据块被加载到缓存时，其相邻数据也可能很快被加载和使用。
- 通过多线程并行的方式可以充分利用机器的物理资源。
    - 在矩阵乘法中，很多计算之间没有先后依赖关系，因此可以通过多线程并行计算的方式来加速矩阵运算。

通过这次实验，我深刻理解了两种加速矩阵乘法的原理与方法，理解了cache在操作系统中的重要意义。