计算机体系结构实验五实验报告

PB21111681 朱炜荣

一、CPU上的矩阵乘法

验证优化后矩阵乘法的正确性，我采用的方法如下代码所示：

```
void gemm_verify(float *A, float *B, float *C, int N){
    float *temp = new float[N*N];
    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            float sum = 0;
            for(int k = 0; k < N; k++){
                sum += A[i*N + k] * B[k*N + j];
            }
            temp[i*N + j] = sum;
        }
    }

    for(int i = 0; i < N*N; i++){
        if(fabs(temp[i] - C[i]) > 1e-3){
            cout << "Error at " << i << endl;
            cout << "temp: " << temp[i] << " C: " << C[i] << endl;
            cout << "Error: " << fabs(temp[i] - C[i]) << endl;
            delete[] temp;
            return;
        }
    }

    cout << "Correct" << endl;
    delete[] temp;
    return;
}
```

即先用最基础100%正确的方法计算出A*B的结果，再将其与我们优化之后计算得到的矩阵进行比较。由于是浮点数之间的比较所以，精度可能存在一定的误差，比较的方法为相减之后的绝对值小于一个定值即认为合格。

由于测试的代码存在体量较小的情况，所以我选择使用库下面的chrono::high_resolution_clock::now()函数来进行计时，使用的编译环境具体见Makefile。

下表各个程序的用时单位均为ms，所用随机数的种子均为1958。

我们发现avx相较于直接进行baseline矩阵乘法，加速效果十分明显，几乎可以说在同一个N的大小下运算时间降低了一个数量级。而矩阵的分块乘法相较于avx的性能更加快速了，但是性能增加的幅度相较于avx之于baseline的增加幅度小了很多。性能增加的原因可能是由于其一次只取宽度为8的数据并集中进行大量运算，使得程序局部性较好，高速缓存缺失率大有降低，从而性能得到进一步的增加。

题目中曾提到你可能需要用到矩阵的转置，于是我在表中的最后一行进行了转置之后的矩阵相乘，可以看到相较于直接的分块矩阵相乘，其性能低了不少。我分析得到的可能原因如下：虽然矩阵分块乘法转置之后可以得到行主序的矩阵，更加方便我们地实际操作，但是这时对于C[i] [j]的元素存储时，需要对c[i]进行求和处理，可能相较于直接放置效率就大打折扣了。

只针对着avx_block_base做block大小的讨论分析，下述测试的矩阵规模N为1024：

block_size	baseline用时/ms	up用时/ms
8	770.901	1495.78
16	683.325	1024.52
32	669.813	857.887
64	703.962	694.464
128	917.005	686.985
256	903.116	639.845
512	882.846	594.09
1024	878.127	584.367

我们可以看到对于基础的矩阵分块乘法来说，运行时间随着块大小的增加先减少后上升，分析原因可能有以下两点：

- 数据局部性更好了，当块大小刚刚开始增加时，在每个分块内的操作逐渐增加，由于数据是连续存取的，所以程序可以充分地利用空间局部性和时间局部性。
- 缓存溢出，当块的大小过大时，我们可以遇到缓存大小不够的情况。也就是整个块内的所有数据没有办法直接装进缓存，此时在块内访问数据时，会涉及到大量的Miss事件，程序必须去内存中寻找对应的数据，此时程序的性能就所有下降了。

  而对于矩阵转置之后的情况，我们发现当块大小与矩阵大小相同时，性能最好。这可能是因为，本身转置后的矩阵分块乘法数据局部性就不体现在分块的部分，而是关于转置矩阵的存取。所以相比分块而言，一起处理反而能提高程序的运行效率。

在CPU平台上，矩阵乘法的优化是一个复杂而多层次的过程，涉及硬件特性、算法设计、编译器优化等多方面。以下是一些常见的矩阵乘法优化手段：

1. 使用向量化指令集利用CPU的向量化指令集（如AVX、AVX2、AVX-512等），可以在单个指令周期内处理多个数据元素，从而显著提高运算效率。常见的向量化指令包括加载（`_mm256_load_ps`）、存储（`_mm256_store_ps`）、乘加（`_mm256_fmadd_ps`）等。
2. 分块算法：分块算法通过将大矩阵分解为较小的子块进行计算，充分利用缓存的局部性，减少缓存失效次数。典型的分块方法包括对矩阵A、B、C进行分块，并在块内进行矩阵乘法。
3. 多线程并行化：利用CPU的多核特性，通过多线程并行计算矩阵乘法。常见的多线程库包括OpenMP和Pthreads。每个线程处理矩阵的一个子块，从而实现并行计算。
4. 缓存优化通过优化内存访问模式，提高缓存命中率。具体策略包括：缓存对齐——确保数据结构在内存中的对齐方式匹配CPU缓存行大小，减少缓存冲突。以及预取——提前加载即将使用的数据到缓存中，减少内存访问延迟。
5. 利用现成的高性能计算库，如Intel Math Kernel Library (MKL)、OpenBLAS、Eigen等，这些库对矩阵运算进行了高度优化，能够自动利用向量化指令、多线程和缓存优化等技术。
6. 自适应算法根据矩阵的大小和结构，选择最合适的算法进行矩阵乘法。例如，对于小矩阵，可以使用简单的三重循环方法；对于大矩阵，可以使用分块和多线程方法。
7. 软件流水线通过重排指令顺序，使得指令之间的依赖关系最小化，CPU能够并行执行更多指令。这样可以提高指令的吞吐量。

8. 手动内存管理通过手动管理内存分配和释放，避免不必要的内存分配开销和碎片化。例如，可以预分配矩阵所需的内存空间，避免在计算过程中频繁进行动态内存分配。

二、GPU上的矩阵乘法

以下GPU运行时间的单位均为毫秒，默认block_size大小为16：

N	MM_base	MM_blocks
128	0.028768	0.028416
256	0.05232	0.039584
512	0.310656	0.213312
1024	2.36413	1.51741
2048	19.4053	11.911

通过分析数据我们可以得到，不管是简单的矩阵乘法还是矩阵分块乘法，都大大提高了运算的性能，接近三个数量级。相较于不分块的乘法，分块乘法实现使用了共享内存来缓存 A 和 B 矩阵的一部分，而这部分内存比全局内存访问更快。这种方式可以减少全局内存访问次数，并且增加了访问共享内存的次数，所以可以加速矩阵乘法操作。

以下GPU运行时间的单位均为毫秒。

N	block_size	MM_base	MM_blocks
128	4	0.028448	0.043008
128	8	0.041024	0.027296
128	16	0.028256	0.03328
128	32	0.028512	0.030624
256	4	0.051328	0.16816
256	8	0.062304	0.061408
256	16	0.057408	0.048064
256	32	0.04896	0.044192
512	4	0.309696	0.997376
512	8	0.310656	0.31552
512	16	0.311488	0.212416
512	32	0.315104	0.187776
1024	4	2.35584	7.78115
1024	8	2.35638	2.35184
1024	16	2.35821	1.51395
1024	32	2.36147	1.40102
2048	4	19.4195	64.3838
2048	8	19.4172	18.7015
2048	16	19.4295	11.9367
2048	32	19.4071	11.009

不分块性能提升的原因：

1. **更多的线程并行执行**：
   - 增加块大小（block size）意味着每个线程块中包含更多的线程。这样，GPU中更多的计算单元（CUDA cores）可以同时被利用，提升了整体的并行计算能力。
2. **共享内存的利用**：
   - 共享内存是比全局内存速度更快的存储器，线程块中的所有线程可以访问和共享数据。块大小增加使得更多的线程可以使用共享内存进行数据交换，从而减少了对全局内存的访问，提升了内存访问效率。

分块程序性能提升的原因：

1. **减少全局内存访问**：
   - 分块程序使用共享内存来存储当前块的数据，减少了对全局内存的访问次数。由于共享内存的访问速度比全局内存快很多，这大大提高了性能。
2. **更多有效计算**：

- 随着块大小增加，更多的线程可以协同工作在同一个数据块上，这减少了需要的循环次数，从而减少了每次矩阵乘法所需的总计算时间。

但是性能最终也不能一直优化效果很好，每个SM（Streaming Multiprocessor）有固定数量的寄存器和共享内存。块大小增加，意味着每个块需要更多的资源。如果超过了SM的资源限制，GPU将无法充分利用并行计算能力，导致性能下降。

同样的增加块大小意味着更多的线程需要同步。这会增加同步开销，从而影响性能。随着块大小的增加，这种同步开销的影响会逐渐显现，导致性能提升逐渐减缓甚至下降。