# CALab5_Report

**PB19051183 吴承泽**

**实验报告文件树目录如下：**

```
CAlab5
   |  CAlab5_report.pdf
   |
   ├─cpu
   |       cpu1.c
   |       cpu1.exe
   |       cpu2.c
   |       cpu2.exe
   |       cpu3_16x16.c
   |       cpu3_16x16.exe
   |       cpu3_32x32.c
   |       cpu3_32x32.exe
   |       cpu3_4x4.c
   |       cpu3_4x4.exe
   |       cpu3_64x64.c
   |       cpu3_64x64.exe
   |       cpu3_8x8.c
   |       cpu3_8x8.exe
   |       Makefile
   |
   └─gpu
            gpu1.cu
            gpu2.cu
```

# CPU

## 1 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。

执行task1,task2,task3中编写的程序，所得到的运行时间如下所示：

| | (2^8,2^8) | (2^9,2^9) | (2^10，2^10) |
|---|---|---|---|
| task1 | 0.060000 | 0.513000 | 5.412000 |
| task2 | 0.045000 | 0.365000 | 3.827000 |
| task3（BLOCKSIZE=8） | 0.031000 | 0.224000 | 1.849000 |

任务一中使用的方法是直接对两个矩阵进行相乘，任务二中使用的方法为通过AVX对矩阵行与列的相乘相加，任务三使用的方法为对整个矩阵分为(2^n,2^n)的块，将矩阵分块相乘后相加，得到每个局部的解，最后通过循环得到整体的解。

性能对比（task3 > task2 > task1）原因如下：

- task1中对B数组每次寻找下一个数需跨越大小为N*sizeof(float)的内存距离，内存局部性较差，寻址中对Cache的利用率较低，从而性能较差。

- task2中使用了`AVX`的SIMD指令来处理矩阵的相乘与相加，通过`ACX`中的接口来提升数据执行的并行度，相较于task1有效的提升了并行程度。
- task3中进行了分块处理，内存局部性与task2相比更好，访存跨度降低，而且task3中使用了转置，将B矩阵中的列大跨度的寻址变为行整体的寻址，有效提升了Cache的利用率，提高了性能。

### 2 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。

执行task3中各种分块参数的程序，所得到的运行时间如下所示：

| | **(2^8,2^8)** | **(2^9,2^9)** | **(2^10，2^10)** |
|---|---|---|---|
| BLOCKSIZE=4 | 0.039000 | 0.280000 | 2.226000 |
| BLOCKSIZE=8 | 0.031000 | 0.224000 | 1.849000 |
| BLOCKSIZE=16 | 0.059000 | 0.451000 | 3.696000 |
| BLOCKSIZE=32 | 0.112000 | 0.953000 | 7.471000 |
| BLOCKSIZE=64 | 0.262000 | 2.132000 | 17.592000 |

影响规律大致为：随着`BLOCKSIZE`的增长，**性能先提升后下降**，主要原因如下：

- 初始阶段`BLOCKSIZE`从4至8时性能提升，在此时分块参数较小，Cache中仅需通过较少次换块可以将换块后的数据包含在内，且后续查找时都可以在Cache中查找到。在块增长时，查找的次数减少，因此性能提升。
- 之后`BLOCKSIZE`从8至64时性能下降，因为当块的大小逐渐增大时，分块的大小以平方的倍数增长，内存的局部性逐渐变差，换块的频率同时也会逐渐增大，因此性能下降。

### 3 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。

- 改变循环的嵌套展开顺序，使得合理的规划内存布局最大化利用Cache，尽可能的一行一行的进行访问。
- 利用流水线的性质进行循环展开，通过硬件的调度算法减少Stall的时钟数量。
- 利用Stressen算法降低时间复杂度，提高CPU的计算效率。

## GPU

### 1 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）， 考察两种实现的性能差异，并简要分析原因。

对于不同矩阵的输入大小来说，取**(blocksize,gridsize)=(16,(1 << (N-4)))**:

| **实现方式\N** | **64** | **128** | **256** | **512** | **1024** |
|---|---|---|---|---|---|
| gpu基础矩阵相乘 | 8.4480us | 28.289us | 211.68us | 1.5883ms | 12.365ms |
| gpu分块矩阵相乘 | 5.3120us | 9.8570us | 41.088us | 287.56us | 2.1596ms |

输入矩阵的增大，所需的运行时间增大，是因为对于更大的N，所需要的资源分配和时间是更多的，会消耗更多点时间去计算。

同时可以看出，gpu基础矩阵相乘总是慢于gpu分块矩阵相乘，大致原因如下：

- 分块矩阵中对每个Block中使用`Shared Memory`存储关键的Block数据供其中的线程使用，Block中的Thread使用`Shared Memory`中的数据的Latency远小于`Global Memory`中的Latency，对于合适将多次使用的数据存入`Shared Memory`中，可以有效提升性能。
- 对于普通的矩阵相乘，分块矩阵拥有更好的`Memory Locality`，对于Thread中的`Local Memory`有着更好的局部性。(类似Cache)

## 2 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。

在不同的`gridsize`与不同的`blocksize`下的执行时间如下**(MatrixSize = (2^10, 2^10))且BlockSize与GridSize满足**$BlockSize * GridSize = (1 << N)$:

| (blocksize,gridsize) | (4,256) | (8,128) | (16,64) | (32,32) | (64,16) |
|---|---|---|---|---|---|
| gpu time(ms) | 7.2911 | 6.5553 | 12.412 | 24.598 | 25.666 |

当`blocksize`从4增至8时，性能变化不大；当`blocksize`从8增至64时，所需时间增长，性能下降。**总体上blocksize增加性能下降**，原因如下:

- 当`blocksize`增加时，对于相同规模的情况下，一个块中的线程数越大，对数据访问的要求越大，对于物理上同一个Block同时对`Global Memory`的访问带宽大小并不是无限的，在`BlockSize`增大时，对访问带宽的压力会增大，当运算的速度超过转交数据的速度后，此时线程会产生空等，并发度下降。

## 3 对GPU-任务2中的分块矩阵乘法，探讨不同的 gridsize 、 blocksize 以及 BLOCK 对性能的影响，并简要分析原因。

在不同的`blocksize`与不同的`gridsize`下的执行时间如下**(MatrixSize = (2^10, 2^10))且BlockSize与GridSize满足**$BlockSize * GridSize = (1 << N)$:

| (blocksize,gridsize) | (4,256) | (8,128) | (16,64) | (32,32) |
|---|---|---|---|---|
| gpu time(ms) | 12.020 | 3.6312 | 2.1577 | 2.5959 |

在分块矩阵乘法中，`BLOCK` = `blocksize`，即分析`blocksize`即可，且$BlockSize * GridSize = (1 << N)$可以使负载均衡，性能较好。

可以看出，当`blocksize`增加时，**性能总体呈先提升后下降**趋势，原因如下:

- Block中存储的`Shared Memory`在`BlockSize`较少的时候利用率不高，因为Block中的线程过少以致于`Share dMemory`所带来的优化有限。当`BlockSize`增大时，`Shared memory`的利用率增大，此时性能提升；再之后`BlockSize`增大时，`Shared Memory`所需存储的数据增加，超过了`Share Memory`的大小，或是`Shared Memory`的带宽不足以支撑较多Block中愈多的线程数，使得线程取数据所需的平均Latency增加，从而导致性能下降。