高性能计算导论：hw6

刘泓尊　2018011446　计 84

2020 年 5 月 13 日

目录

本次作业已上传至集群**./2018011446/hw6** 文件夹下。

1 GEMM

本次实验我实现了基于矩阵分块 (tile) 的 GEMM 程序，包括 SGEMM 和 DGEMM，并利用 Shared Memory 进行加速。功能为计算**矩阵-矩阵乘法**:

$$C_{m\times n} = \alpha A_{m\times k} \cdot B_{k\times n} + \beta C_{m\times n}$$

其中 $\alpha = 1.5, \beta = 0.5$.

1.1 代码实现

1.1.1 循环展开的版本

在 baseline 版本的矩阵乘法中，每个线程都要读矩阵 A 的一行和矩阵 B 的一列，计算与访存比约为 1:1，我先对 baseline 版本进行了循环展开，作为一个小测试。代码如下:

```
int i = 0;
for(; i < k; i += stride) {
    tmp += A[row_off + i] * B[col + i*n];
    tmp += A[row_off + i + 1] * B[col + (i+1)*n];
    tmp += A[row_off + i + 2] * B[col + (i+2)*n];
    tmp += A[row_off + i + 3] * B[col + (i+3)*n];
}
    i -= stride;
for(; i < k; i++){
    tmp += A[row_off + i] * B[col + i*n];
}
```

实际测试发现，循环展开之后性能没有明显提升，GFLOPS 仅仅提升了 5 个点左右，因此我没有再使用此版本。

1.1.2 矩阵分块的版本

观察可以发现，矩阵乘法中每个元素都被多个线程重复使用，将元素存入 Shared Memory 供线程使用有望获得更好的性能，因此我使用矩阵分块策略，每一个 block 计算一小块方阵，每个 thread 计算这一小块方阵的一个元素，分块大小为 TILE_WIDTH. 当矩阵不能被方阵完全覆盖时，方阵补零做 padding 即可，每个线程将矩阵 A 和 B 的元素读入 Shared Memory. 代码如下:

gemm_kernel()

```
int block_col = threadIdx.x;
int block_row = threadIdx.y;
int col = blockIdx.x * blockDim.x + block_col;
int row = blockIdx.y * blockDim.y + block_row;
int i, j;
T v = 0;
__shared__ T shareM[TILE_WIDTH][TILE_WIDTH];
__shared__ T shareN[TILE_WIDTH][TILE_WIDTH];
for(i = 0; i < block_num; i++){
```

```
    shareM[block_row][block_col] = ((i * TILE_WIDTH+block_col < K) && (row < M))
        ? A[row * K + (i * TILE_WIDTH+block_col)] : 0.;
    shareN[block_row][block_col] = ((i * TILE_WIDTH+block_row < K) && (col < N))
        ? B[(i * TILE_WIDTH+block_row) * N + col] : 0.;
    __syncthreads();
    if( (col < N) && (row < M) ){
        for(j = 0; j < TILE_WIDTH; j ++){ //can unroll loop
            v += shareM[block_row][j] * shareN[j][block_col];
        }
    }
    __syncthreads();
}
if( (col < N) && (row < M) ) {
    C[row * N + col] = alpha*v + beta * C[row * N + col];
}
```

在第二部分将对性能进行测试和分析。

1.1.3 循环展开

为了更大限度地激发指令级并行，我将上述代码中的循环部分展开，将步数为 TILE_WIDTH 的循环改为了 TILE_WIDTH 条顺序执行的指令，但实测发现没有明显的加速，GFLOPS 仅仅提升了 1. 左右。因为循环展开的部分十分简单，这里就不罗列代码了。

1.1.4 内存对齐

线程访问内存以段对齐的方式读取，遇到非 2 的幂次的数组，实际上也是要读取完整的 2 的幂次，因此我对矩阵 A, B, C 做了内存对齐，方法是使用 cudaMallocPitch() 函数。预处理部分如下:

MemoryPitch

```
T *tmp_A, *tmp_B, *tmp_C;
size_t pitch_a_device, pitch_b_device, pitch_c_device;
cudaMallocPitch((void**)&tmp_A, &pitch_a_device, sizeof(T)*K, M);
cudaMallocPitch((void**)&tmp_B, &pitch_b_device, sizeof(T)*N, K);
cudaMallocPitch((void**)&tmp_C, &pitch_c_device, sizeof(T)*N, M);
cudaMemcpy2D(tmp_A, pitch_a_device, A, sizeof(T)*K, sizeof(T)*K, M,
    cudaMemcpyDeviceToDevice);
cudaMemcpy2D(tmp_B, pitch_b_device, B, sizeof(T)*N, sizeof(T)*N, K,
    cudaMemcpyDeviceToDevice);
cudaMemcpy2D(tmp_C, pitch_c_device, C, sizeof(T)*N, sizeof(T)*N, M,
    cudaMemcpyDeviceToDevice);
checkCudaErrors(cudaDeviceSynchronize());
```

预处理之后调用内存对齐版本的核函数，寻址时不再使用 A[row * M + col] 而是 A[row * A_pitch + col]，可以更好的利用访存性能。在第二部分将对此优化的性能进行测试和分析。

1.2 性能测试与分析

分析部分的统计图均为针对 float 的统计数据，double 的数据放在报告最后。

1.2.1 对不同规模方阵和不同分块大小的测试

使用上述矩阵分块版本 (无内存对齐) 针对不同规模的方阵进行测试，得到的结果如下:

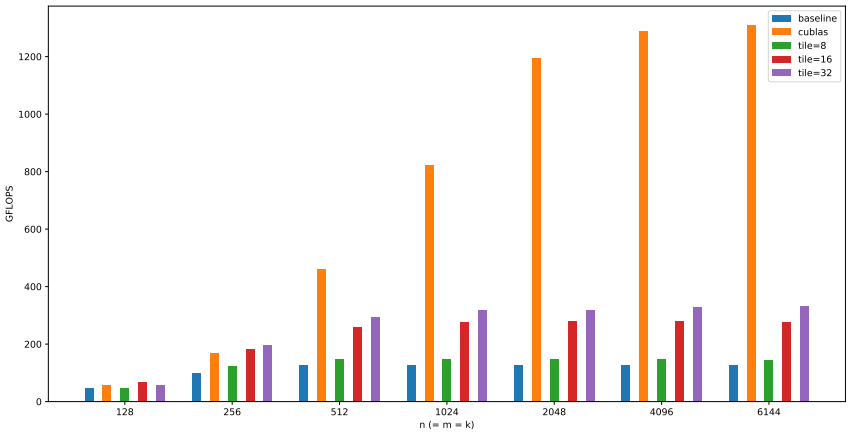

图 1: 矩阵分块版本 GEMM(with cublas)(float)

可以看到，cublas 非常快，我的实现仅仅达到了 cublas 的 25% 左右。为了便于展示我的数据，接下来的统计将不计入 cublas 的程序，但我十分希望助教老师讲解作业的时候能介绍一下 cublas 的优化技巧！

我对不同矩阵规模 (N) 和不同分块大小 (TILE) 进行统计，得到 GFLOPS 如下:

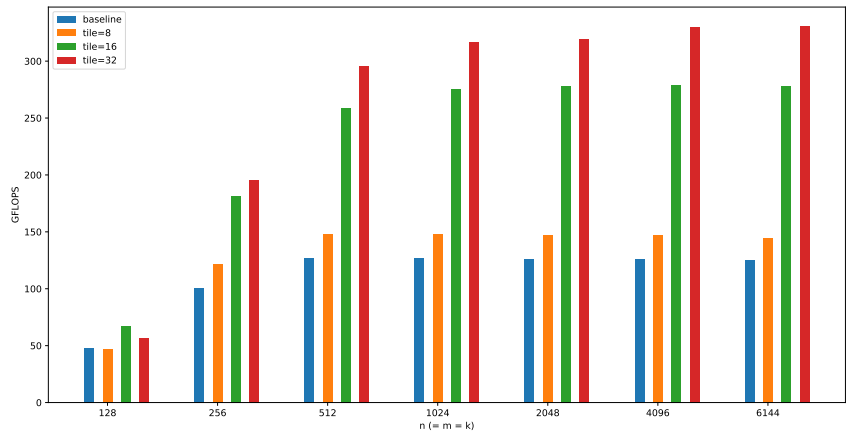

图 2: GEMM 性能对比 (不同矩阵规模和分块大小)(float)

从图中可以看到，baseline 版本和我的版本均在 N=1024 左右达到性能峰值。不同分块大小对性能的影响很大，TILE=8 的版本仅仅比 baseline 好一点；TILE=16 的版本可以达到 baseline 的两倍以上；TILE=32 的版本可以达到 baseline 的三倍左右。如果 Shared Memory 足够大，进一步增大 TILE_SIZE 可以取得更好的性能。

分块策略是矩阵乘法常用的优化策略。因为矩阵乘法中每个元素都被多个线程重复使用，分块乘法可以更好地利用局部性。此外，将元素存入 Shared Memory 供线程使用可以获得更好的性能，因为 Shared Memory 能够用来隐藏由于 latency 和 bandwidth 对性能的影响。

我进一步探究了 row-major 和 column-major 对性能的影响，我的程序涉及向 SM 读数据和存数据，因为在同一个 warp 中的 thread 的使用相邻的 threadIdx.x 来访问 SM，如果使用 column-major 来存取数据，每个 SM 的读写都会导致 bank-conflict，导致程序的访存性能降低。因此在读写 SM 的时候，应该尽量使用 row-major.

1.2.2 对 MemPitch 版本的测试

我对加了 MemPitch 的矩阵乘法做了测试，矩阵规模均选取不是 2 的幂次的数据，这样更方便展现 MemPitch 的效果。

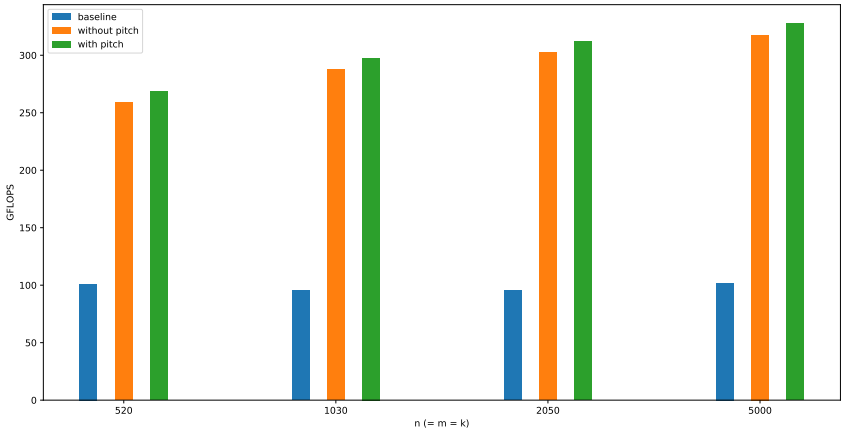

图 3: 使用 MemPitch 的 GEMM(float)

可以看到，使用 MemPitch 的版本性能有了略微的提升 (实际上因为 GFLOPS 单位很大，运算次数已经提升很多了)。对于规模不是 2 的 n 次幂的矩阵，使用内存对齐预处理可以达到更好的性能。

1.2.3 对狭长形矩阵的测试

为了更全面地展示性能，我对长方形矩阵做了性能测试，使用 TILE_SIZE=32 的分块策略。性能统计如下。

baseline 对于矩阵形状不太敏感，但是使用分块策略的矩阵乘法对矩阵形状较为敏感，虽然性能一直优于 baseline，但是对于特别“狭长”的矩阵，性能有了明显的下降。这是因为在固定的 TIILE_SIZE 下，矩阵宽度的减小将减少 block 的数量 (分块策略的不灵活性也由此体现)，导致有效线程数减少，性能降低。

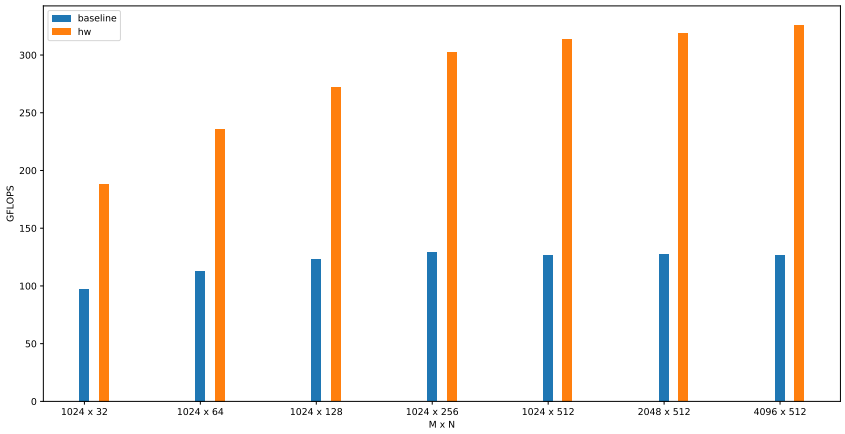

图 4: 长方形矩阵的性能测试 (float)

1.2.4 对 double 的统计数据展示

double 类型矩阵的结果与 float 类似，因此我放在最后统一展示。cublas 的性能有了明显下降，我的实现相对于 baseline 的改进基本和 float 类型类似。

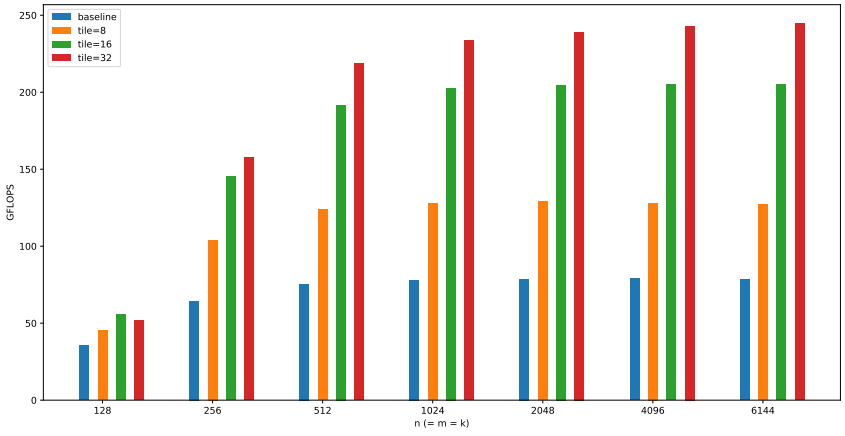

图 5: 不同规模矩阵和分块策略 (double)

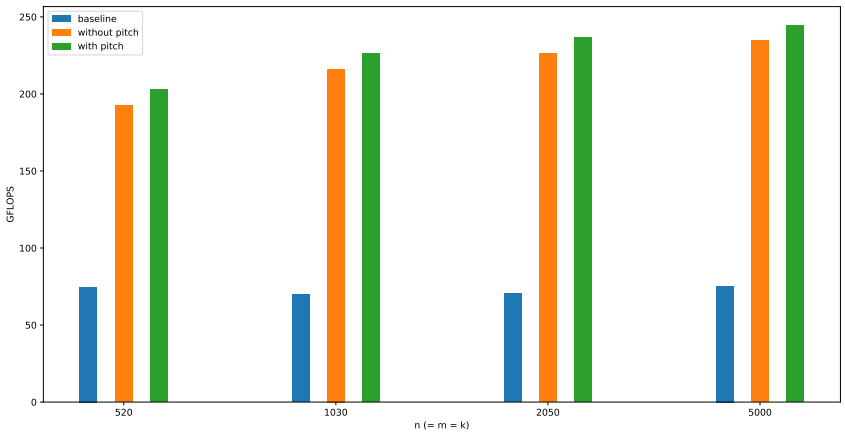

图 6: 使用 MemPitch 的 GEMM(double)

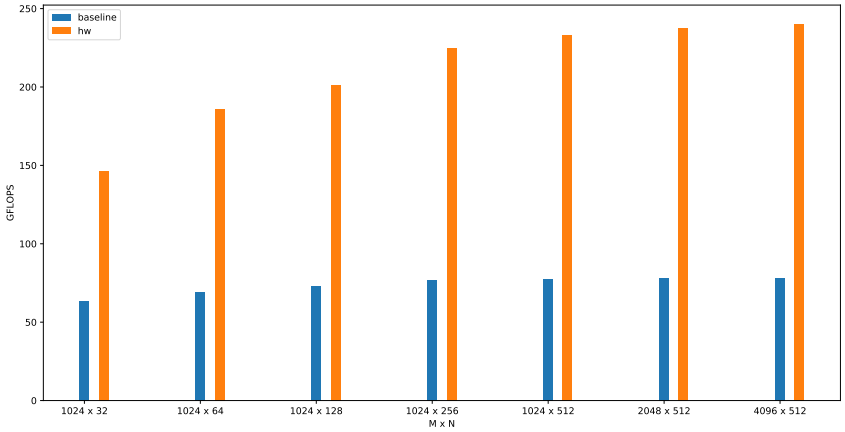

图 7: 长方形矩阵的性能测试 (double)

1.3 总结

本次作业我实现了 CUDA 版本的 GEMM，采用矩阵分块策略加速矩阵乘法，达到了 baseline 版本 3 倍的加速。使用了 Shared Memory 存取数据，并尽可能避免了 Bank Conflict. 但是和 cublas 的实现依然存在很大的差距，仅仅有其 1/5 - 1/4 的性能。接下来可以进一步研究使用数据预取和指令级并行来进一步优化。感谢助教老师的悉心指导！