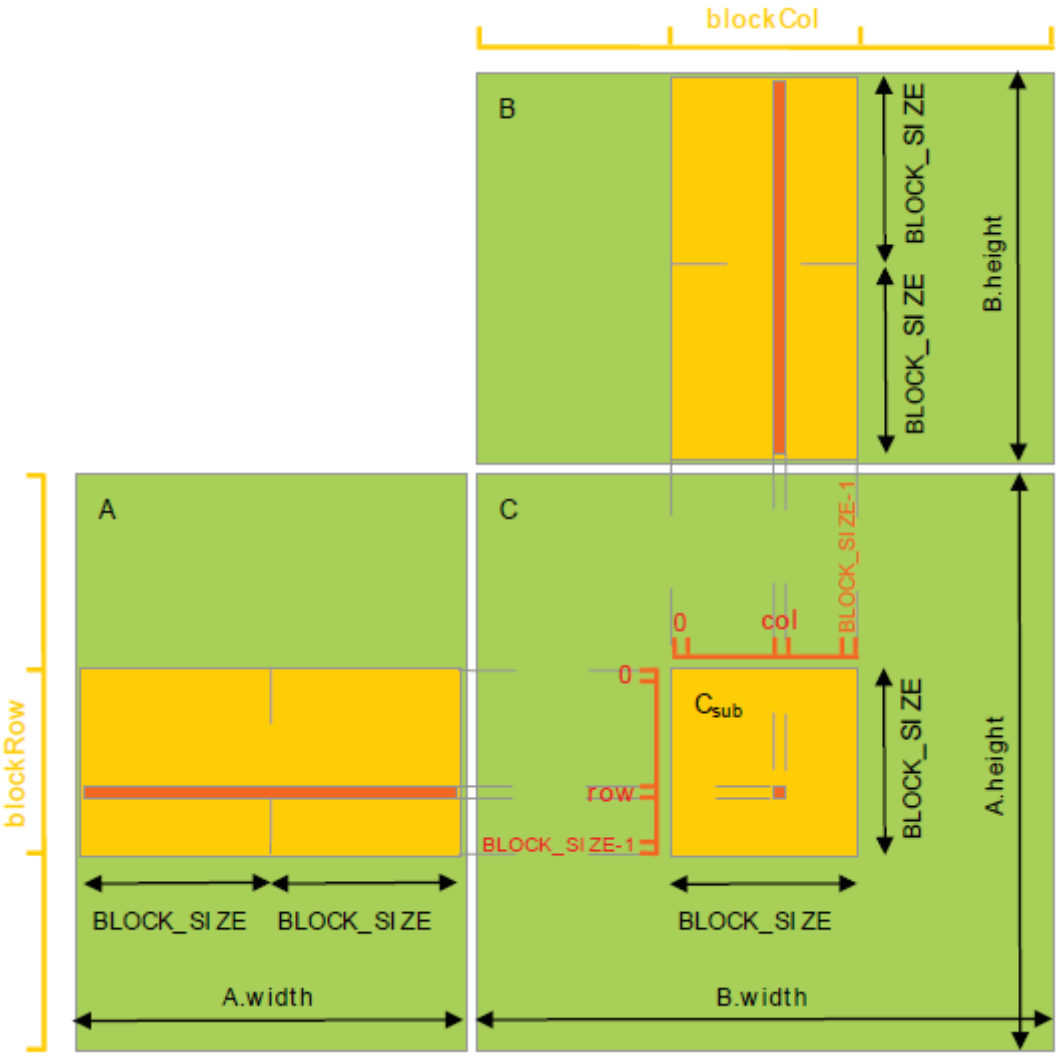
Homework 6

2018011365 张鹤潇

算法简述

Tilling



在baseline版本的基础上，通过共享内存进行优化。如图所示，将矩阵A和矩阵B分成多个小方阵，每个小方阵中的数据由一个block内的线程进行处理；整块数据都被读入共享内存后，每个线程调用其所需要的数据进行计算。实现代码如下：

```
// BLOCK DIM: (TS, TS)
    const size_t row = TS * blockIdx.y + threadIdx.y,
        col = TS * blockIdx.x + threadIdx.x;
    __shared__ T Asub[TS][TS], Bsub[TS][TS];

    T c = 0.;
```

```
    int k0 = 0, tiles = (K + TS - 1) / TS;

    for (int i = 0; i < tiles; i++, k0 += TS) {  // 遍历行和列
        if (row < M && (k0 + threadIdx.x < K))
            Asub[threadIdx.y][threadIdx.x] = A[row * K + (k0 + threadIdx.x)];
        else
            Asub[threadIdx.y][threadIdx.x] = 0.;

        if (col < N && (k0 + threadIdx.y < K))
            Bsub[threadIdx.y][threadIdx.x] = B[(k0 + threadIdx.y) * N + col];
        else
            Bsub[threadIdx.y][threadIdx.x] = 0.;
        __syncthreads();

        for (int k = 0; k < TS; k++)  // 执行一部分计算
            c += Asub[threadIdx.y][k] * Bsub[k][threadIdx.x];
        __syncthreads();
    }

    if (row < M && col < N) {
        C[row * N + col] = alpha * c + beta * C[row * N + col];
    }
```

Further Optimization

参考Matrix multiplication in CUDA，以另一种思路对程序进行优化。将

```
c += Asub[threadIdx.y][k] * Bsub[k][threadIdx.x];
```

由两次shared memory访问减少到一次。为了做到这一点，需要改变矩阵乘法的方式，由原来矩阵A的行向量与矩阵B的列向量做内积，变为矩阵A的列向量与矩阵B的行向量做外积。

将矩阵A的一小块按column-major读入共享内存，每个线程负责将块中元素与B中相应元素逐个相乘，将结果存入寄存器中。因B中每个元素的访问有极强的局部性，会使编译器将单个元素读入寄存器，这就大大提高了计算效率。代码实现如下，

```
// BLOCK DIM: (TS, VECTOR_SIZE) = (16, 4)
    const size_t bx = blockIdx.x,
        by = blockIdx.y,
        tx = threadIdx.x,
        ty = threadIdx.y;

    __shared__ T Asub[TS * TS];
    T Csub[TS] = { 0. };

    const size_t aBegin = K * TS * by,
        aEnd = aBegin + K - 1,
        aStep = TS,
        bBegin = TS * VECTOR_SIZE * bx,
        bStep = TS * N;
    const size_t vec_col = TS * ty + tx;

    for (int a = aBegin, b = bBegin, phase = 0; a <= aEnd; a += aStep, b += bStep, phase +=
TS) {
        for (int i = 0; i < TS; i += VECTOR_SIZE) {  // read in Asub in column major
```

```
            if (phase + tx < K && TS * by + (i + ty) < M)
                Asub[(i + ty) + TS * tx] = A[a + K * (i + ty) + tx];
            else
                Asub[(i + ty) + TS * tx] = 0.;
        }
        __syncthreads();

        if (bBegin + vec_col < N) {
            T* ap = Asub, * bp = &B[b + vec_col];
            for (int i = 0; i < TS && phase + i < K; ++i) {
                T bv = *bp;
                for (int k = 0; k < TS; k++)  // ap[k] = Asub[i][k]
                    Csub[k] += ap[k] * bv;
                ap += TS;
                bp += N;
            }
        }
        __syncthreads();
    }

    if (bBegin + vec_col < N) {
        int row = TS * by, col = bBegin + vec_col;
        int idx = N * row + col;
        for (int i = 0; i < TS && row < M; ++i) {
            C[idx] = alpha * Csub[i] + beta * C[idx];
            row++; idx += N;
        }
    }
```

为了达到更高的性能，我将外积计算循环展开。

```
        // ...
        if (bBegin + vec_col < N) {
            T* ap = Asub, * bp = &B[b + vec_col];
#pragma unroll
            for (int i = 0; i < TS && phase + i < K; ++i) {
                T bv = *bp;
                Csub[0] += ap[0] * bv;
                Csub[1] += ap[1] * bv;
                Csub[2] += ap[2] * bv;
                Csub[3] += ap[3] * bv;
                Csub[4] += ap[4] * bv;
                Csub[5] += ap[5] * bv;
                Csub[6] += ap[6] * bv;
                Csub[7] += ap[7] * bv;
                Csub[8] += ap[8] * bv;
                Csub[9] += ap[9] * bv;
                Csub[10] += ap[10] * bv;
                Csub[11] += ap[11] * bv;
                Csub[12] += ap[12] * bv;
                Csub[13] += ap[13] * bv;
                Csub[14] += ap[14] * bv;
                Csub[15] += ap[15] * bv;
                ap += TS;  // TS = 16
                bp += N;
```

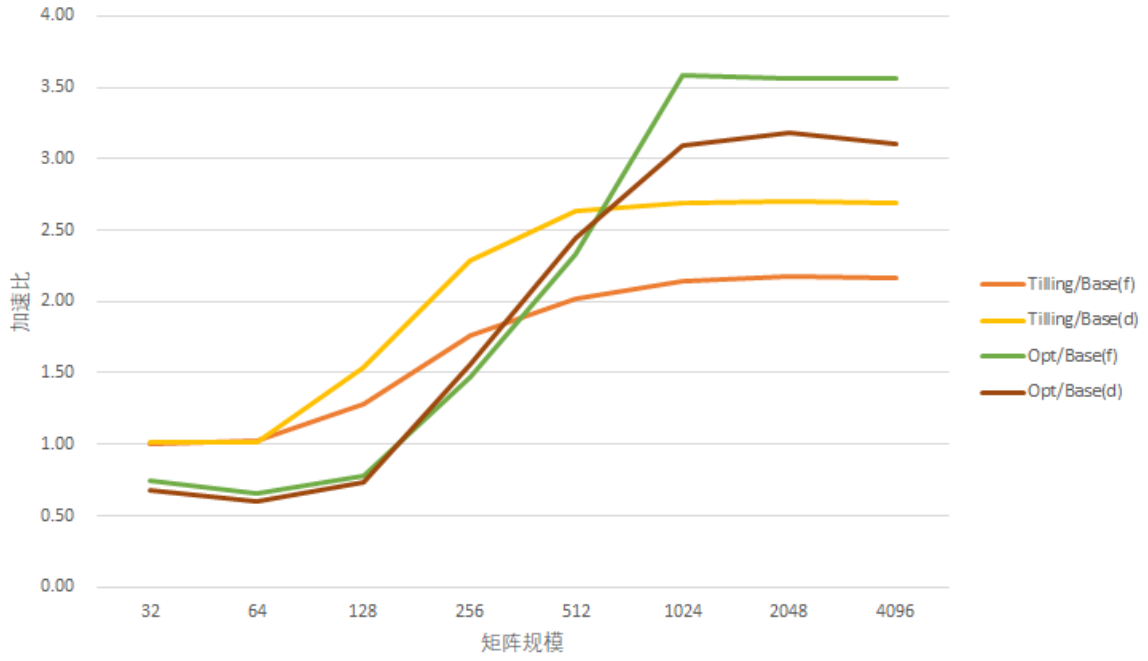
```
            }
        }
        //...
```

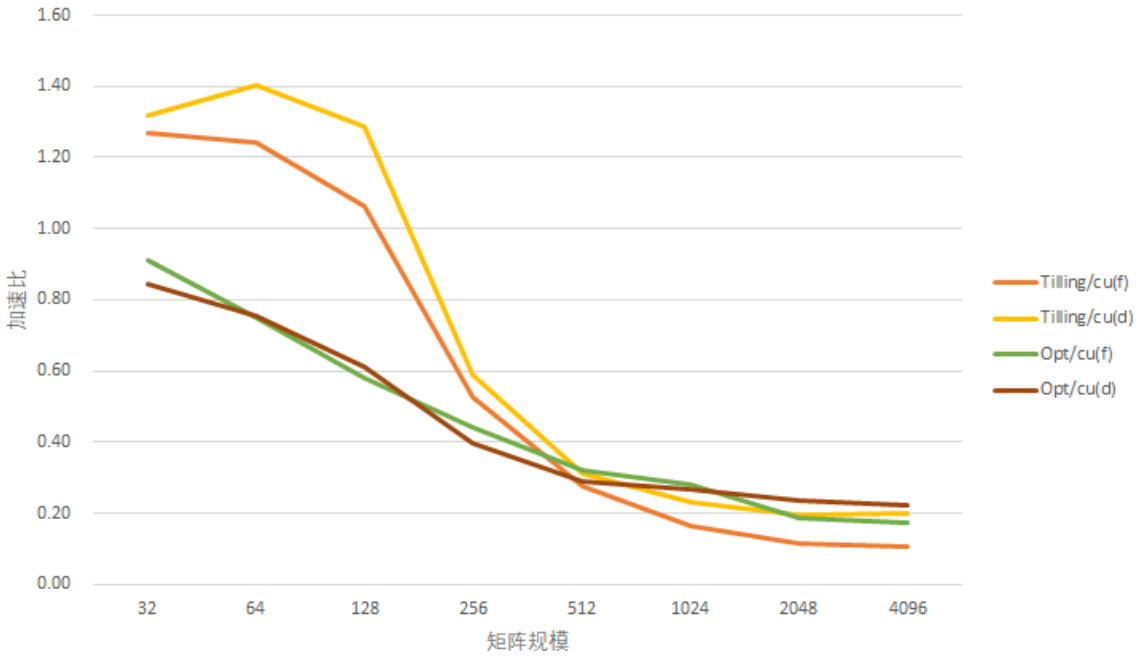
性能测试

程序的正确性在整个测试过程中得到验证。

设定$N = M = K$为2的方幂，测试结果如下，表中数据均为加速比。

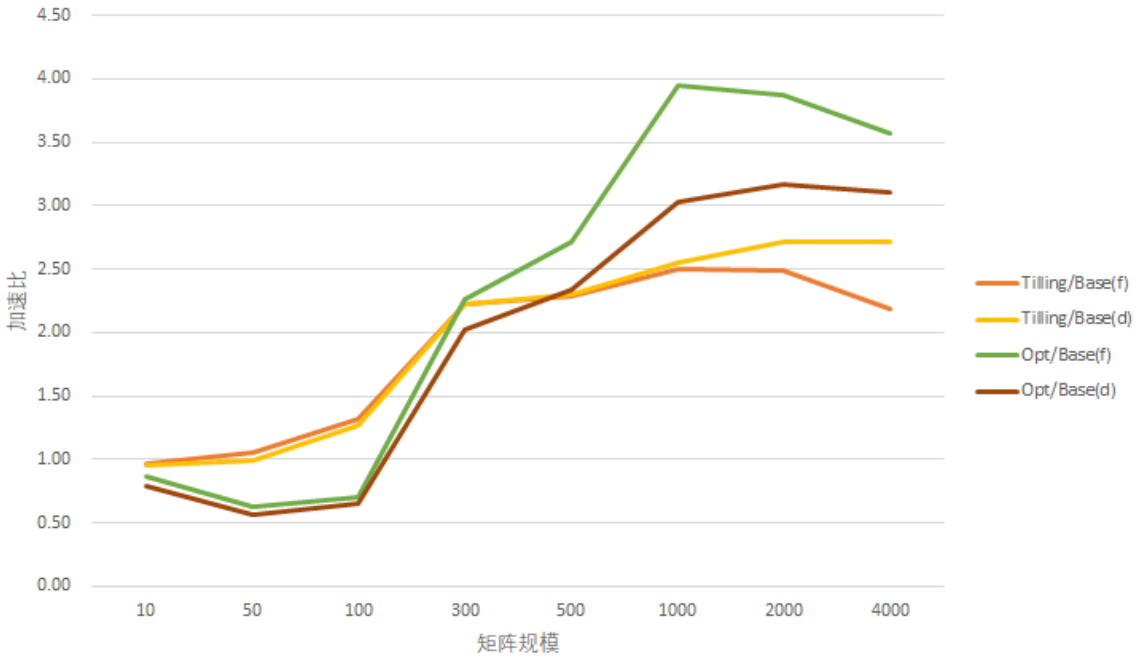
N	Tilling/Base(f)	Tilling/Base(d)	Opt/Base(f)	Opt/Base(d)	Tilling/cu(f)	Tilling/cu(d)	Opt/cu(f)	Opt/cu(d)
32	1.01	1.02	0.74	0.67	1.27	1.32	0.91	0.85
64	1.02	1.02	0.65	0.61	1.24	1.40	0.75	0.76
128	1.29	1.53	0.78	0.73	1.06	1.29	0.58	0.61
256	1.77	2.29	1.48	1.57	0.53	0.59	0.44	0.40
512	2.02	2.63	2.33	2.45	0.27	0.31	0.32	0.29
1024	2.14	2.69	**3.58**	3.10	0.16	0.23	0.28	0.27
2048	**2.17**	**2.70**	3.56	**3.18**	0.11	0.20	0.19	0.23
4096	2.17	2.69	3.56	3.11	0.11	0.20	0.18	0.22

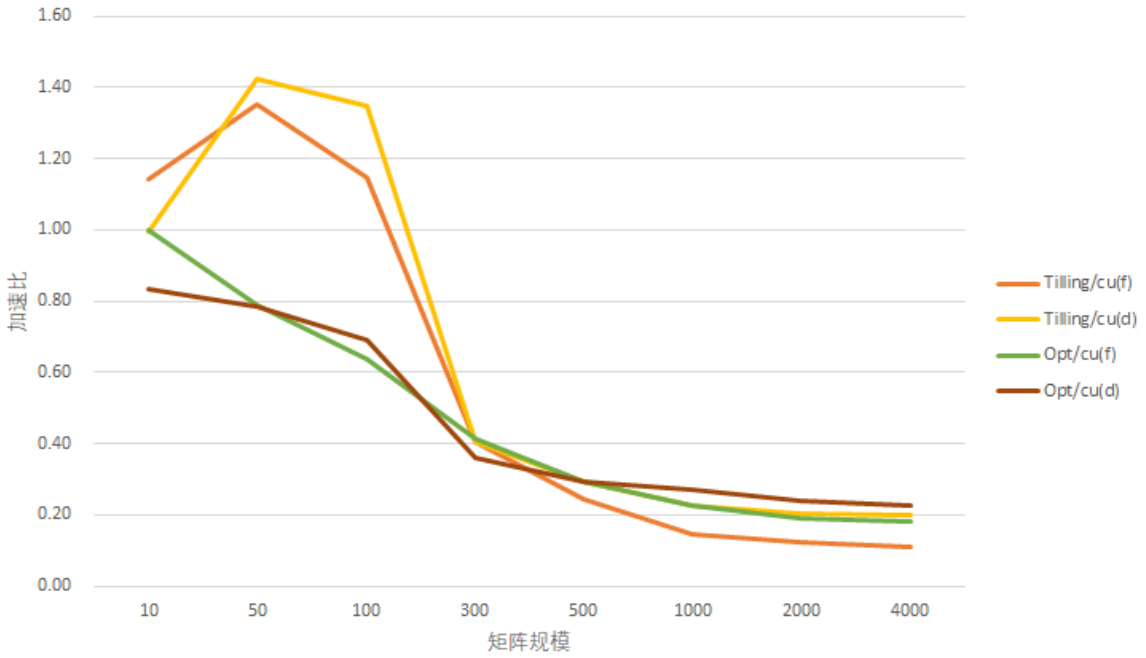




设定$N = M = K$不为2的方幂，测试结果如下。

N	Tilling/Base(f)	Tilling/Base(d)	Opt/Base(f)	Opt/Base(d)	Tilling/cu(f)	Tilling/cu(d)	Opt/cu(f)	Opt/cu(d)
10	0.97	0.96	0.86	0.79	1.14	1.00	1.00	0.83
50	1.06	1.00	0.63	0.56	1.35	1.42	0.79	0.79
100	1.31	1.27	0.70	0.66	1.15	1.35	0.64	0.69
300	2.22	2.23	2.26	2.02	0.40	0.40	0.41	0.36
500	2.29	2.30	2.72	2.34	0.24	0.29	0.29	0.29
1000	**2.51**	2.55	**3.95**	3.03	0.14	0.22	0.23	0.27
2000	2.49	**2.72**	3.87	**3.16**	0.12	0.21	0.19	0.24
4000	2.18	2.71	3.56	3.10	0.11	0.20	0.18	0.23





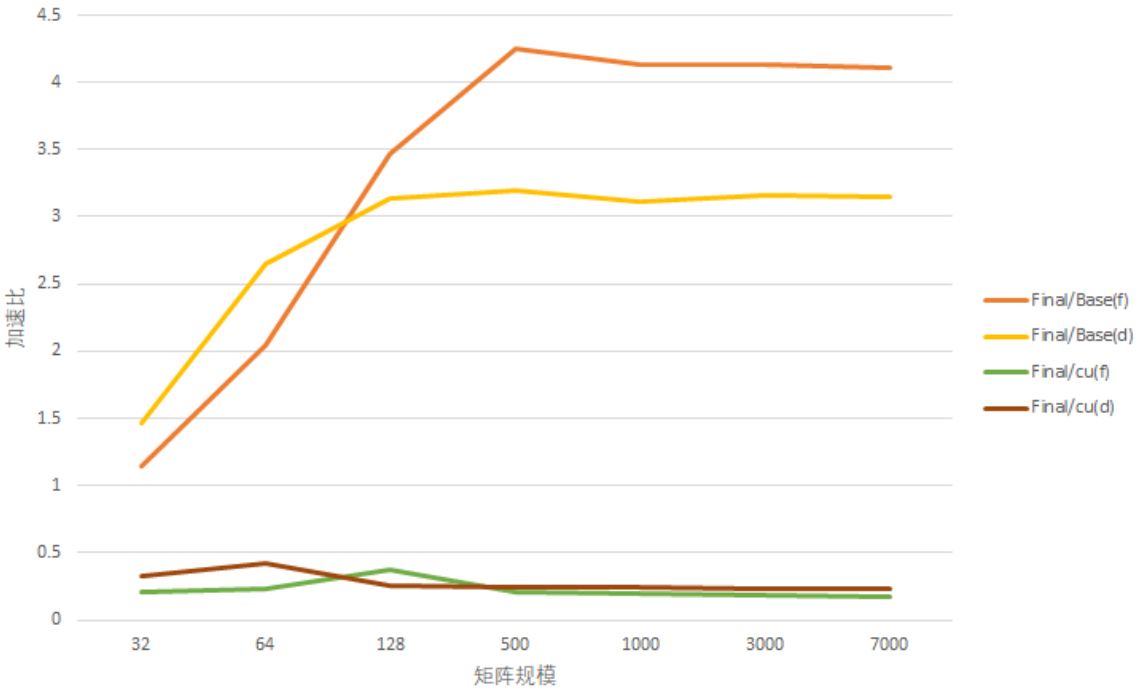
可见，当矩阵规模较小时，逻辑简单的tilling性能较好；当矩阵规模变大时，Opt开始占据优势，对baseline的加速比最大可以达到4倍。但当矩阵规模变大时，我的程序与cublas库函数的差距也越来越大，只能达到cublas两三成的水平，可见程序的提升空间还很大。

对于这两种实现来说，矩阵的维数是否是2的幂影响不大。

为了结合两种实现的优势，我在主调函数中加入了条件判断，如果矩阵的规模较小，就调用tilling，否则调用Opt.

对于非方阵，固定$M = K = 1760$，改变N的规模，以最终提交版本测试，结果如下。

N	Final/Base(f)	Final/Base(d)	Final/cu(f)	Final/cu(d)
32	1.1425	1.4647	0.2093	0.3240
64	2.0424	2.6517	0.2363	0.4191
128	3.4664	3.1406	0.3785	0.2552
500	4.2528	3.1948	0.2070	0.2433
1000	4.1345	3.1153	0.1946	0.2408
3000	4.1350	3.1563	0.1833	0.2339
7000	4.1055	3.1423	0.1747	0.2259


4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
加速比
32
64
128
500
1000
3000
7000
矩阵规模
Final/Base(f)
Final/Base(d)
Final/cu(f)
Final/cu(d)