Question #1 (15 points). For the results above, did the compute times for the different matrix sizes vary in a reasonable manner as the problem size increased? How do they compare to the reference results from lab #3 (matrix multiply on a multi-core CPU)?

(1)The running time for matrices of different sizes varies in a reasonable manner. When the matrix size is relatively small, all the cores of the SM cannot be fully utilized. When the matrix size gradually increases from 1k*1k to 4k*4k, as the amount of data increases, the computing power consumed also increases by 8 times (N3fmac), and the amount of data of 1k*1k can contain 1k (32*32) blocks, these data volumes can be evenly distributed to all cores of the SM, and these cores can be fully utilized. Therefore, in the process from 1k*1k to 4k*4k, the core of sm is fully utilized, and the calculation time is proportional to the calculation amount, which is about 8 times, which is basically consistent with our running time。

(2) When comparing the reference results with the results of Experiment 3: the running time of dumb and the total time of GPU running have changed a lot, both of which have been reduced a lot.
Probably because the way the B matrix is accessed in the GPU version makes it suitable for spatial locality. This reduces the time for the GPU version.
For other reference results using GPU, we can see that the total runtime is much shorter than the CPU version. That's because a GPU has more cores than a CPU, which allows more threads to run on the GPU, which increases speed.

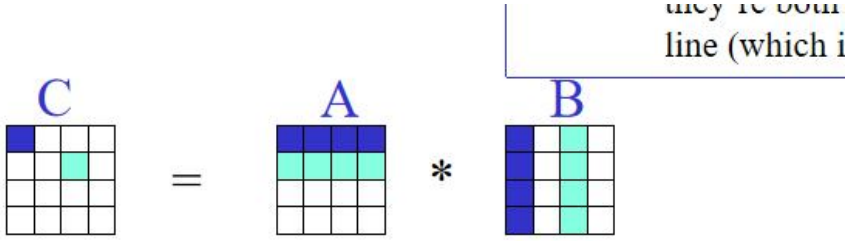
Question #2. The results above are for code that has both good shared-memory bank usage and good memory-coalescing access. Show a small snippet of your code and explain why its shared memory bank usage is either good or bad (10 points), and ditto for its memory-coalescing access 10 points).

```
__global__ void mat_mult (float *d_A, float *d_B, float *d_C, int N) {
    int rb = blockIdx.x;
    int cb = blockIdx.y;
    int ri = threadIdx.x;
    int ci = threadIdx.y;
```

```
// Copy the data to shared memory
for (int kb = 0; kb <gridDim.x; kb++) {
    SA[ri][ci] = d_A[N*(rb*BS+ri)+kb*BS+ci];
    SB[ri][ci] = d_B[N*(kb*BS+ri)+cb*BS+ci];
    __syncthreads();
```

Here is a small piece of my code, I think the following modifications can be done。

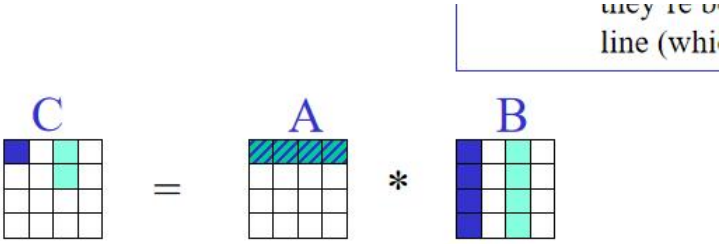
int ri = threadIdx.y;
int ci =threadIdx.x;




- Assume:

Before changing the threadid, when different threads access matrixA, they may access different rows of A, but since the bank is stored according to the column, this may cause collide.

After applying this: int ri = threadIdx.y;  int ci =threadIdx.x; we can avoid that question. Then we can access matrix like this:




- Are there some threads that don't collide when they

Now, when we access matrix A , we different threads can access same line of matrix A,which can avoid collision.( we changes the access sequence of  matrix A).

(2)  for its memory-coalescing access 10 points).

All threads in a wrap have the same threadIdx.y and consecutive threadIdx.x. Each thread needs to load one element of SA, SB from DRAM, so my code is not suitable for spatial locality. This is because each thread accesses SB from DRAM in the order of columns, and our data is stored in rows in DRAM. Now accessing DRAM in columns will cause us to jump continuously in memory. So we should change the code to change the order of accessing DRAM. We can achieve this by changing the code:

int ri = threadIdx.y;
int ci =threadIdx.x;
With this modification, we can achieve our goal.

```
Compute capability = 6.0

Working on 32x32 matrices with BS=32
Dumb mpy took 7.6e-05sec
mpy1 averaged 0.0756585sec

Working on 64x64 matrices with BS=32
Dumb mpy took 0.000534sec
mpy1 averaged 9.3e-05sec

Working on 256x256 matrices with BS=32
Dumb mpy took 0.031151sec
mpy1 averaged 0.0005775sec

Working on 1024x1024 matrices with BS=32
Dumb mpy took 2.32063sec
mpy1 averaged 0.011322sec

Working on 2048x2048 matrices with BS=32
Dumb mpy took 18.5098sec
mpy1 averaged 0.0621652sec

Working on 4096x4096 matrices with BS=32
Dumb mpy took 156.449sec
mpy1 averaged 0.405205sec
```