1, brief explanation describing how my changes ensure use of shared memory

Explanation of function **mmul(float *A, float *B, float *C, int ds):**

part 1: allocate dynamic shared memory to shared[]

part 2: There are (ds+blockDim.x-1)/blockDim.x blocks, process width blocks for one shared memory shared with the first for loop. width is a number, I assigned it as 4.

part 3: In the width blocks unit, assign values of matrix A and B into shared separately in the second for loop.

part 4: Use __syncthreads(); Synchronizes all threads within a block. A thread's execution can only proceed past a __syncthreads() after all threads in its block have executed the __syncthreads(). In other words, it ensures that each thread waits until all values of matrix A and B are given to shared memory ident shared in the width blocks in all threads.

part 5: Then add the multiplication of one element of A and corresponding element B in shared memory ident shared to temp.

part 6: __syncthreads(); to make sure that all values in the width block are added to temp

part 7: After all elements of one row of A and col of B are processed in parallel, assign the multiplication result termed temp to C.

```
extern __shared__ float shared[];
for (int i =0; i < ((ds+blockDim.x-1)/blockDim.x); i = i+width) {
        for (int j = 0; j < width; j = j+1){
            if (idy < ds && ((i+j)*blockDim.x+threadIdx.x) < ds)  {shared[threadIdx.y*blockDim.x*width + threadIdx.x + j*blockDim.x] = A[idy*ds +  (i+j)*blockDim.x+threadIdx.x];}
            else  {shared[threadIdx.y*blockDim.x*width + threadIdx.x + j*blockDim.x] = 0;}
            if (idx < ds && ((i+j)*blockDim.x+threadIdx.y) < ds)  {shared[threadIdx.y*blockDim.x + threadIdx.x + (j+width)*blockDim.x*blockDim.x] = B[((i+j) * blockDim.x + threadIdx.y) * ds + idx];}
            else  {shared[threadIdx.y*blockDim.x + threadIdx.x + (j+width) * blockDim.x * blockDim.x] = 0;}
        }
        __syncthreads();
        for (int k = 0; k < blockDim.x * width; ++k) {
temp += shared[threadIdx.y*blockDim.x*width+k] * shared[k*blockDim.x+threadIdx.x+width*blockDim.x*blockDim.x];
       }
        __syncthreads();
    }
 if (idy < ds && idx < ds) {C[idy*ds + idx] = temp;}
```

Explanation of **Launch kernel part:**

part 1: launch kernel, 2D blocks, block_size indicate number of threads in x axis and y axis, (DSIZE+block.x-1)/block.x indicates number of blocks in x axis, and (DSIZE+block.y-1)/block.y indicates number of blocks in y axis.

part 2: perform mmul function on device, the memory size of dynamic shared memory is sizeof(float)*block_size*block_size*2 * width

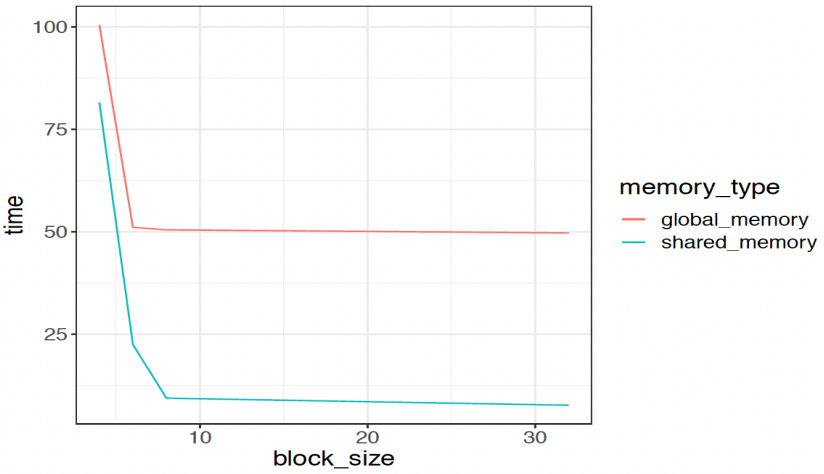
```
  dim3 block(block_size, block_size);
  dim3 grid((DSIZE+block.x-1)/block.x, (DSIZE+block.y-1)/block.y);
  mmul<<<grid, block, sizeof(float)*block_size*block_size*2 * width>>>(d_A, d_B, d_C, DSIZE);
```

Explanation of **cleaning up all sources it allocates part:**

```
//Free memory
  free(h_A);
  free(h_B);
  free(h_C);
```

```
// free(h_C1);
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

2, chart of graph which illustrates the scalability



Please choose one grid/block size which gave you the best performance.

Answer: I choose block size of 32 which gave me the best performance.

Explain why the grid/block size you chose gave the best performance

Answer: From the plot we can see that the bigger block_size, the less time it needs, the better the performance. Since the number of threads one block has is block_size*block_size, one block can have 1024(32*32) threads at most, so 32 is the biggest block size that can work. Thus, block size 32 gave the best performance.

Describe any unusual or unexpected results your found.

Answer: I found that if I put the block size higher than 32, the program will get wrong. I found that for global memory, after block size 8, the block size higher, the performance will not get much better.

For shared memory, the performance will get better with the increase of the block size but the scalability will decrease.