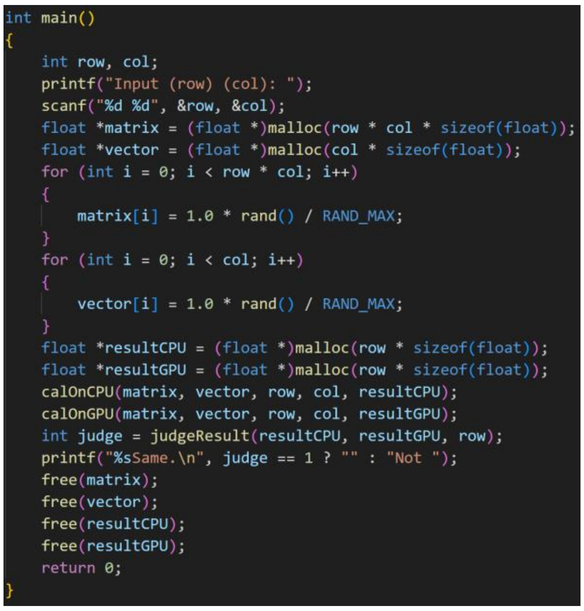
问题 1

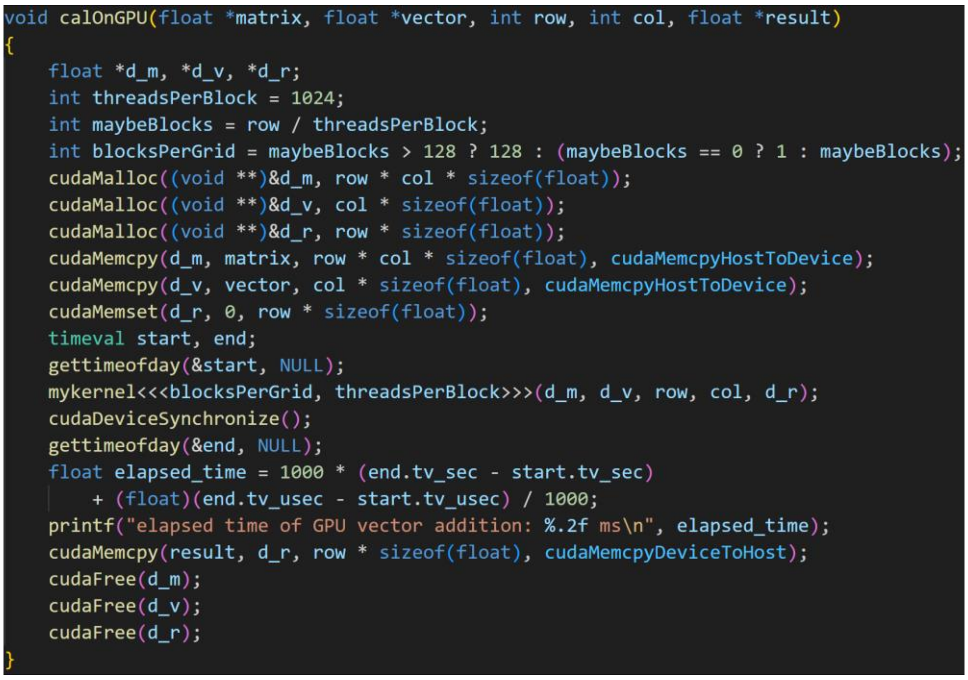
实验思路

整体上，程序读入用户输入的矩阵行和列，然后生成随机的矩阵和向量。分别在 CPU 和 GPU 上执行矩阵与向量相乘，打印计算时间并对比计算结果。



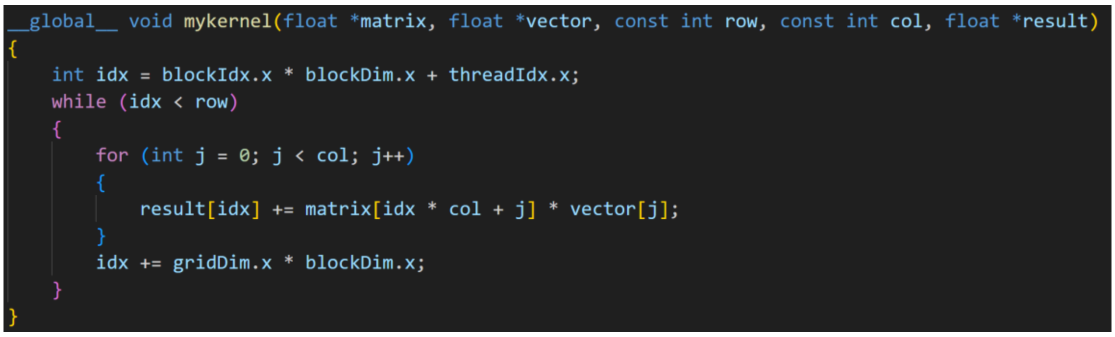
```
int main()
{
    int row, col;
    printf("Input (row) (col): ");
    scanf("%d %d", &row, &col);
    float *matrix = (float *)malloc(row * col * sizeof(float));
    float *vector = (float *)malloc(col * sizeof(float));
    for (int i = 0; i < row * col; i++)
    {
        matrix[i] = 1.0 * rand() / RAND_MAX;
    }
    for (int i = 0; i < col; i++)
    {
        vector[i] = 1.0 * rand() / RAND_MAX;
    }
    float *resultCPU = (float *)malloc(row * sizeof(float));
    float *resultGPU = (float *)malloc(row * sizeof(float));
    calOnCPU(matrix, vector, row, col, resultCPU);
    calOnGPU(matrix, vector, row, col, resultGPU);
    int judge = judgeResult(resultCPU, resultGPU, row);
    printf("%sSame.\n", judge == 1 ? "" : "Not ");
    free(matrix);
    free(vector);
    free(resultCPU);
    free(resultGPU);
    return 0;
}
```

对于 GPU 计算，每个 Block 的 Thread 固定为 1024 个。每个 Grid 的 Block 数目根据输入的数据来动态计算，最小为 1，最大为 128。另外，只计算了在 GPU 上运行的时间。



```
void calOnGPU(float *matrix, float *vector, int row, int col, float *result)
{
    float *d_m, *d_v, *d_r;
    int threadsPerBlock = 1024;
    int maybeBlocks = row / threadsPerBlock;
    int blocksPerGrid = maybeBlocks > 128 ? 128 : (maybeBlocks == 0 ? 1 : maybeBlocks);
    cudaMalloc((void **)&d_m, row * col * sizeof(float));
    cudaMalloc((void **)&d_v, col * sizeof(float));
    cudaMalloc((void **)&d_r, row * sizeof(float));
    cudaMemcpy(d_m, matrix, row * col * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_v, vector, col * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemset(d_r, 0, row * sizeof(float));
    timeval start, end;
    gettimeofday(&start, NULL);
    mykernel<<<blocksPerGrid, threadsPerBlock>>>(d_m, d_v, row, col, d_r);
    cudaDeviceSynchronize();
    gettimeofday(&end, NULL);
    float elapsed_time = 1000 * (end.tv_sec - start.tv_sec)
        + (float)(end.tv_usec - start.tv_usec) / 1000;
    printf("elapsed time of GPU vector addition: %.2f ms\n", elapsed_time);
    cudaMemcpy(result, d_r, row * sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(d_m);
    cudaFree(d_v);
    cudaFree(d_r);
}
```

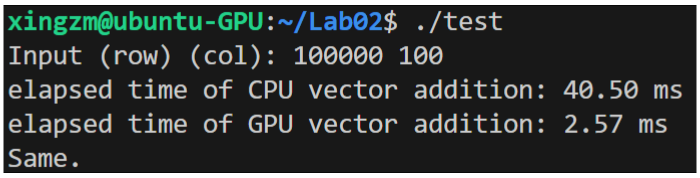
内核的代码如下所示。对 row 使用了 1 维的编号，使用 while 循环来防止计算不足或者越界。



```
__global__ void mykernel(float *matrix, float *vector, const int row, const int col, float *result)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    while (idx < row)
    {
        for (int j = 0; j < col; j++)
        {
            result[idx] += matrix[idx * col + j] * vector[j];
        }
        idx += gridDim.x * blockDim.x;
    }
}
```

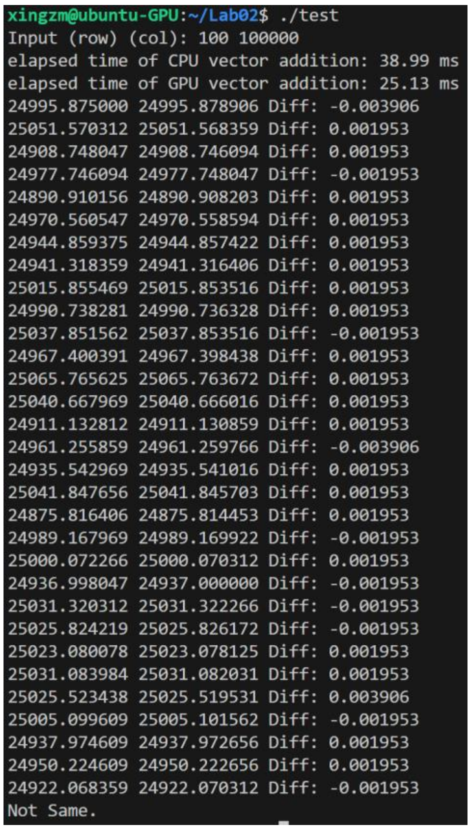
实验结果

在矩阵行数较大时，GPU 计算速度明显快于 CPU，提升了约 15 倍。



```
xingzm@ubuntu-GPU:~/Lab02$ ./test
Input (row) (col): 100000 100
elapsed time of CPU vector addition: 40.50 ms
elapsed time of GPU vector addition: 2.57 ms
Same.
```

对比精度为10^{-6}。当向量列数较多时，GPU 上运行的结果与 CPU 出现了一些不同。打印异常位置的元素，可以看到误差为10^{-3}量级。

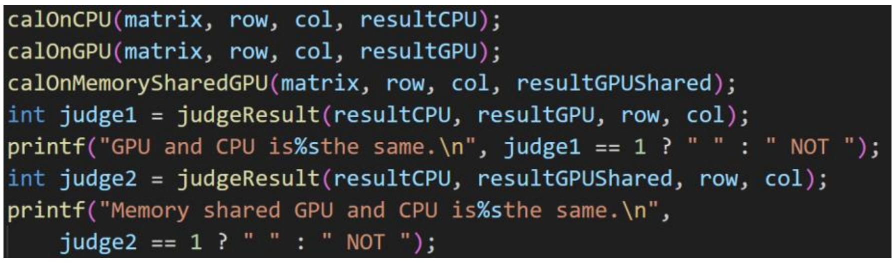


```
xingzm@ubuntu-GPU:~/Lab02$ ./test
Input (row) (col): 100 100000
elapsed time of CPU vector addition: 38.99 ms
elapsed time of GPU vector addition: 25.13 ms
24995.875000 24995.878906 Diff: -0.003906
25051.570312 25051.568359 Diff: 0.001953
24908.748047 24908.746094 Diff: 0.001953
24977.746094 24977.748047 Diff: -0.001953
24890.910156 24890.908203 Diff: 0.001953
24970.560547 24970.558594 Diff: 0.001953
24944.859375 24944.857422 Diff: 0.001953
24941.318359 24941.316406 Diff: 0.001953
25015.855469 25015.853516 Diff: 0.001953
24990.738281 24990.736328 Diff: 0.001953
25037.851562 25037.853516 Diff: -0.001953
24967.400391 24967.398438 Diff: 0.001953
25065.765625 25065.763672 Diff: 0.001953
25040.667969 25040.666016 Diff: 0.001953
24911.132812 24911.130859 Diff: 0.001953
24961.255859 24961.259766 Diff: -0.003906
24935.542969 24935.541016 Diff: 0.001953
25041.847656 25041.845703 Diff: 0.001953
24875.816406 24875.814453 Diff: 0.001953
24989.167969 24989.169922 Diff: -0.001953
25000.072266 25000.070312 Diff: 0.001953
24936.998047 24937.000000 Diff: -0.001953
25031.320312 25031.322266 Diff: -0.001953
25025.824219 25025.826172 Diff: -0.001953
25023.080078 25023.078125 Diff: 0.001953
25031.083984 25031.082031 Diff: 0.001953
25025.523438 25025.519531 Diff: 0.003906
25005.099609 25005.101562 Diff: -0.001953
24937.974609 24937.972656 Diff: 0.001953
24950.224609 24950.222656 Diff: 0.001953
24922.068359 24922.070312 Diff: -0.001953
Not Same.
```

问题 2

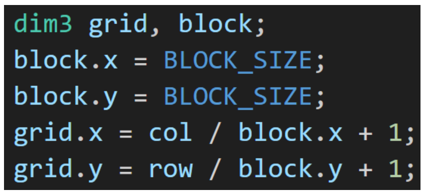
实验思路

使用模块化编程，分别运行 CPU、GPU 和使用了共享内存的 GPU 的计算函数，然后判断两种 GPU 方式与 CPU 执行的结果是否一致。



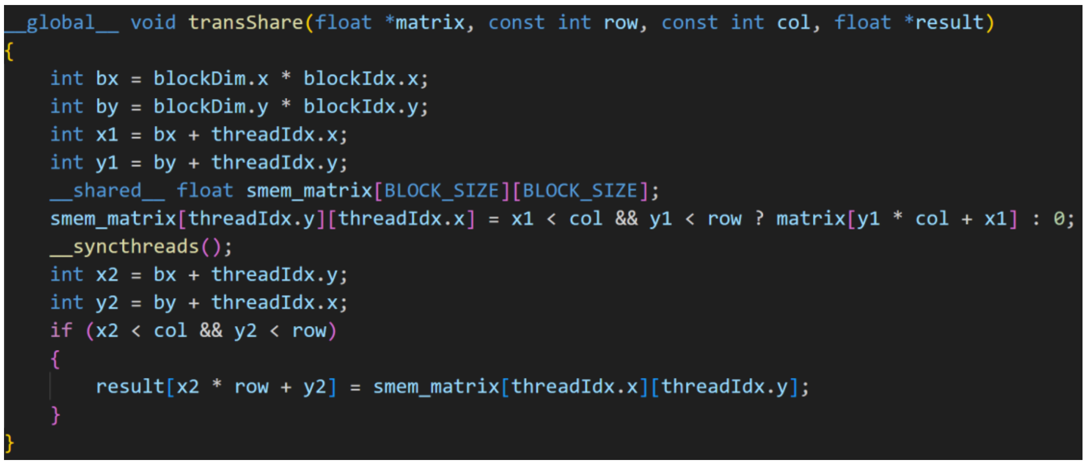
```
calOnCPU(matrix, row, col, resultCPU);
calOnGPU(matrix, row, col, resultGPU);
calOnMemorySharedGPU(matrix, row, col, resultGPUShared);
int judge1 = judgeResult(resultCPU, resultGPU, row, col);
printf("GPU and CPU is%sthe same.\n", judge1 == 1 ? " " : " NOT ");
int judge2 = judgeResult(resultCPU, resultGPUShared, row, col);
printf("Memory shared GPU and CPU is%sthe same.\n",
    judge2 == 1 ? " " : " NOT ");
```

两个维度的 block 大小都设置为了 32，grid 大小由矩阵的行列数目决定。+1 保证了线程的充足性，每个线程对一个元素进行转置。在使用了共享内存的 GPU 计算中，x 设置为列变量，y 设置为了行变量。



```
dim3 grid, block;
block.x = BLOCK_SIZE;
block.y = BLOCK_SIZE;
grid.x = col / block.x + 1;
grid.y = row / block.y + 1;
```

使用了共享内存的 GPU 计算内核代码如下图所示。在每个块中，声明块大小的共享内存区域。对于一个处理线程(y_1, x_1)，将其值存放到共享内存变量的$[t_y][t_x]$位置，其中t_y、t_x为线程号。然后等待线程同步，当这个块中所有线程都将值写入共享内存后，每个线程处理在块中对称的线程(x_2, y_2)，因此是从共享内存中读$[t_x][t_y]$位置的值。最后将这个值存放到 result 的转置位置，也就是$[y][x]$，但此时行列发生变化，y 是列。

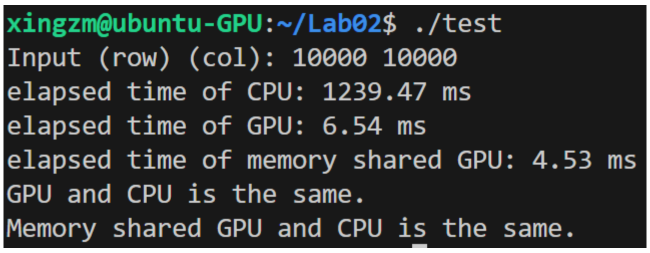


```
__global__ void transShare(float *matrix, const int row, const int col, float *result)
{
    int bx = blockDim.x * blockIdx.x;
    int by = blockDim.y * blockIdx.y;
    int x1 = bx + threadIdx.x;
    int y1 = by + threadIdx.y;
    __shared__ float smem_matrix[BLOCK_SIZE][BLOCK_SIZE];
    smem_matrix[threadIdx.y][threadIdx.x] = x1 < col && y1 < row ? matrix[y1 * col + x1] : 0;
    __syncthreads();
    int x2 = bx + threadIdx.y;
    int y2 = by + threadIdx.x;
    if (x2 < col && y2 < row)
    {
        result[x2 * row + y2] = smem_matrix[threadIdx.x][threadIdx.y];
    }
}
```

由于会申请到多余矩阵大小的线程，因此需要在读 matrix 数组和写 result 数组之前判断下标是否越界。在写共享内存时，越界了也要继续向后执行，因为对称的位置可能没有越界，需要在同步后读取并写回。

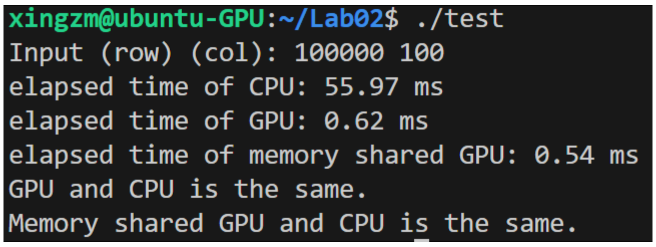
实验结果

在 1000*1000 的方阵上分别运行 3 种代码，可以看到单纯使用 GPU 比 CPU 提升了 189 倍，用了共享内存的 GPU 版本又进一步地提升了 1.4 倍。

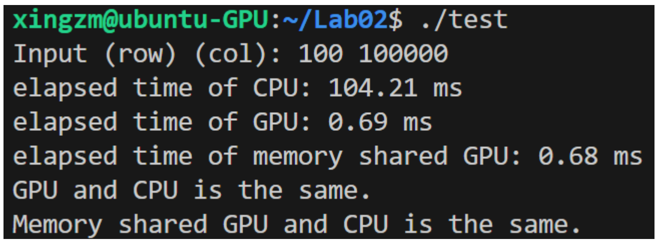


```
xingzm@ubuntu-GPU:~/Lab02$ ./test
Input (row) (col): 10000 10000
elapsed time of CPU: 1239.47 ms
elapsed time of GPU: 6.54 ms
elapsed time of memory shared GPU: 4.53 ms
GPU and CPU is the same.
Memory shared GPU and CPU is the same.
```

对于行和列相差较大的矩阵，共享内存的 GPU 计算依然是最快的且正确的，如下面两张图所示。



```
xingzm@ubuntu-GPU:~/Lab02$ ./test
Input (row) (col): 100000 100
elapsed time of CPU: 55.97 ms
elapsed time of GPU: 0.62 ms
elapsed time of memory shared GPU: 0.54 ms
GPU and CPU is the same.
Memory shared GPU and CPU is the same.
```

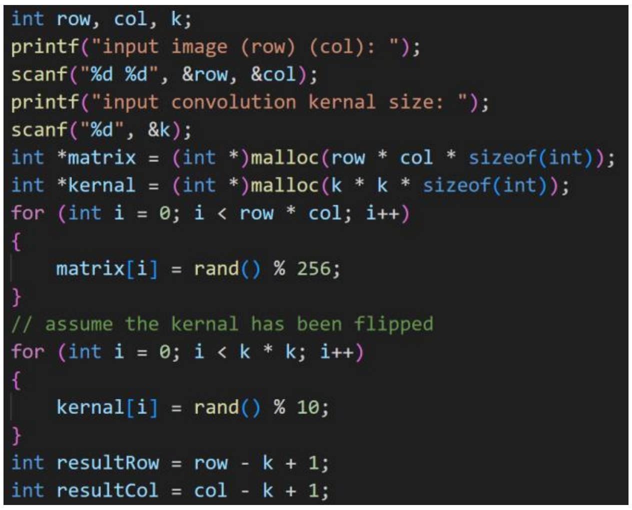


```
xingzm@ubuntu-GPU:~/Lab02$ ./test
Input (row) (col): 100 100000
elapsed time of CPU: 104.21 ms
elapsed time of GPU: 0.69 ms
elapsed time of memory shared GPU: 0.68 ms
GPU and CPU is the same.
Memory shared GPU and CPU is the same.
```

问题 3

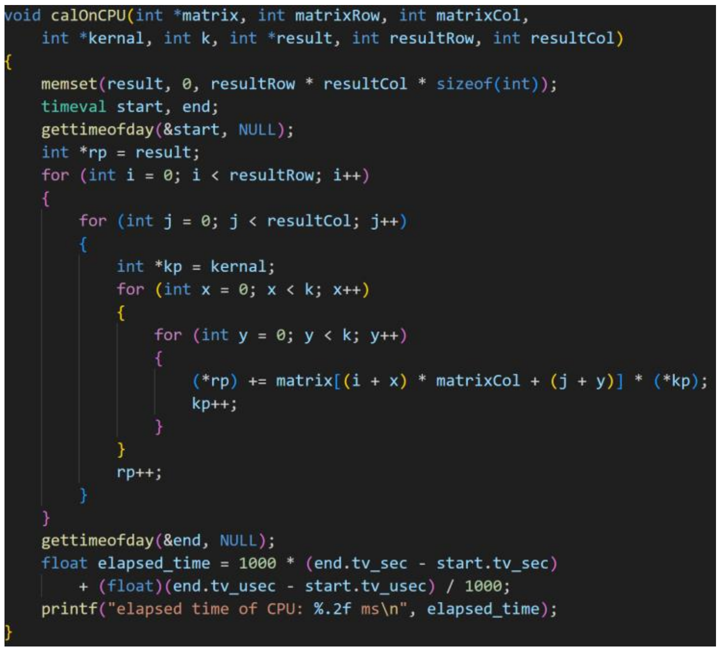
实验思路

让用户输入矩阵行列大小以及卷积核大小，然后随机生成矩阵和卷积核，这里假设生成的卷积核就是翻转完的。本实验使用 valid 卷积，即不对边界补零，因此卷积后矩阵长度可计算，如下面的代码所示。



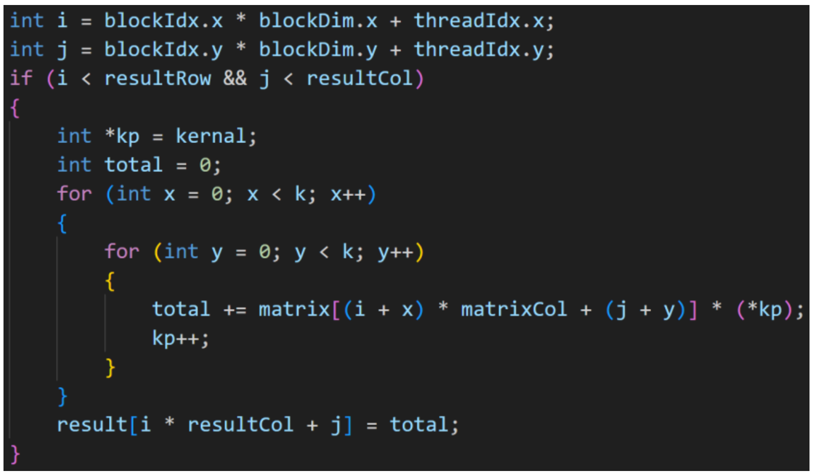
```
int row, col, k;
printf("input image (row) (col): ");
scanf("%d %d", &row, &col);
printf("input convolution kernal size: ");
scanf("%d", &k);
int *matrix = (int *)malloc(row * col * sizeof(int));
int *kernal = (int *)malloc(k * k * sizeof(int));
for (int i = 0; i < row * col; i++)
{
    matrix[i] = rand() % 256;
}
// assume the kernal has been flipped
for (int i = 0; i < k * k; i++)
{
    kernal[i] = rand() % 10;
}
int resultRow = row - k + 1;
int resultCol = col - k + 1;
```

CPU 版本的代码如下所示。对结果矩阵位置使用 2 重循环遍历，在对卷积核进行 2 重循环遍历。将卷积核元素与原矩阵对应元素相乘，然后加入到结果的对应位置。这里对 CPU 代码的遍历部分简单使用了指针优化，减少了计算下标的开销。



```
void calOnCPU(int *matrix, int matrixRow, int matrixCol,
    int *kernal, int k, int *result, int resultRow, int resultCol)
{
    memset(result, 0, resultRow * resultCol * sizeof(int));
    timeval start, end;
    gettimeofday(&start, NULL);
    int *rp = result;
    for (int i = 0; i < resultRow; i++)
    {
        for (int j = 0; j < resultCol; j++)
        {
            int *kp = kernal;
            for (int x = 0; x < k; x++)
            {
                for (int y = 0; y < k; y++)
                {
                    (*rp) += matrix[(i + x) * matrixCol + (j + y)] * (*kp);
                    kp++;
                }
            }
            rp++;
        }
    }
    gettimeofday(&end, NULL);
    float elapsed_time = 1000 * (end.tv_sec - start.tv_sec)
        + (float)(end.tv_usec - start.tv_usec) / 1000;
    printf("elapsed time of CPU: %.2f ms\n", elapsed_time);
}
```

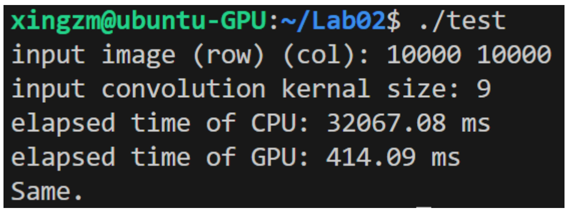
GPU 版本部分的代码与实验中前 2 个问题几乎一致，内核代码使用了二维的 block，其余部分与 CPU 版本类似。



```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
if (i < resultRow && j < resultCol)
{
    int *kp = kernal;
    int total = 0;
    for (int x = 0; x < k; x++)
    {
        for (int y = 0; y < k; y++)
        {
            total += matrix[(i + x) * matrixCol + (j + y)] * (*kp);
            kp++;
        }
    }
    result[i * resultCol + j] = total;
}
```

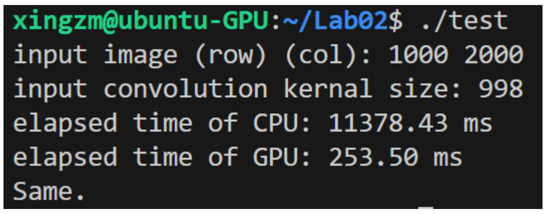
实验结果

对大矩阵(10000*10000)小卷积核(9*9)进行实验，可以看到执行后的结果相同，GPU 版本比 CPU 版本提升了约 77 倍。



```
xingzm@ubuntu-GPU:~/Lab02$ ./test
input image (row) (col): 10000 10000
input convolution kernal size: 9
elapsed time of CPU: 32067.08 ms
elapsed time of GPU: 414.09 ms
Same.
```

对小矩阵(1000*2000)大卷积核(998*998)进行实验，GPU 版本比 CPU 版本提升了约 44 倍。



```
xingzm@ubuntu-GPU:~/Lab02$ ./test
input image (row) (col): 1000 2000
input convolution kernal size: 998
elapsed time of CPU: 11378.43 ms
elapsed time of GPU: 253.50 ms
Same.
```