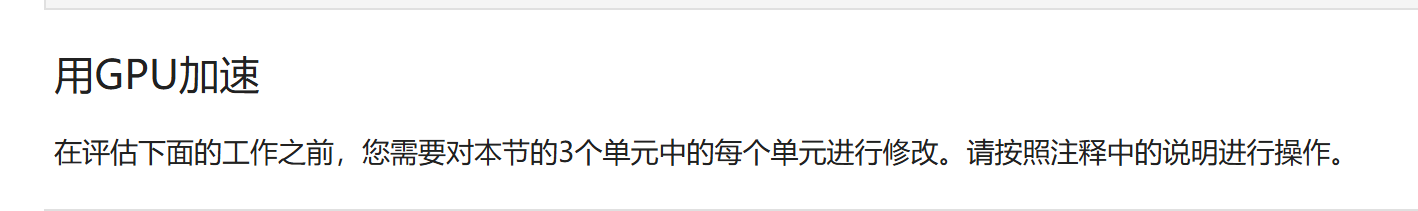
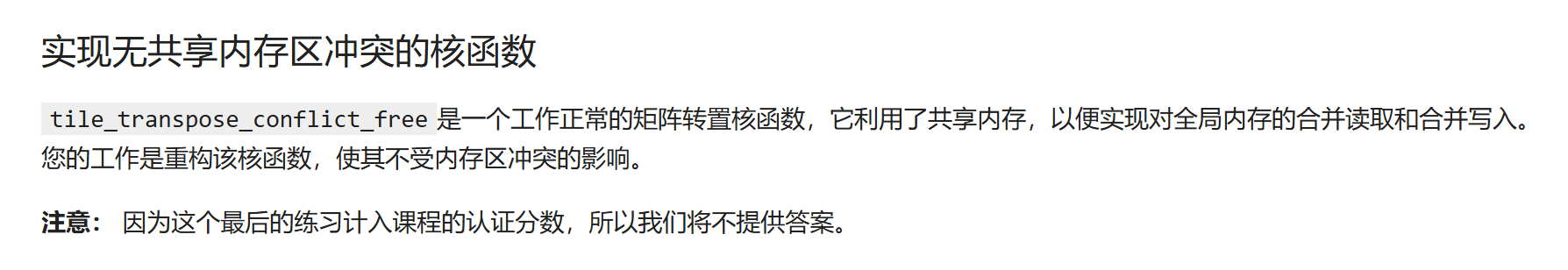
第一个从这儿开始



|  |
| --- |
| # As you will recall, `numpy.exp` works on the CPU, but, cannot be used in GPU implmentations.  # This import will work for the CPU-only boilerplate code provided below, but  # you will need to modify this import before your GPU implementation will work.  # from numpy import exp  import math |
| # Modify these 3 function calls to run on the GPU.  @vectorize(['float32(float32)'], target='cuda')  def normalize(grayscales):  return grayscales / 255  @vectorize(['float32(float32,float32)'], target='cuda')  def weigh(values, weights):  return values \* weights  @vectorize(['float32(float32)'], target='cuda')  def activate(values):  return ( math.exp(values) - math.exp(-values) ) / ( math.exp(values) + math.exp(-values) ) |
| # Modify the body of this function to optimize data transfers and therefore speed up performance.  # As a constraint, even after you move work to the GPU, make this function return a host array.  from numba import cuda  def create\_hidden\_layer(n, greyscales, weights, exp, normalize, weigh, activate):  # 输入  d\_greyscales = cuda.to\_device(greyscales)  d\_weights = cuda.to\_device(weights)    # 输出  d\_normalized = cuda.device\_array(shape=(n,), dtype=np.float32)  d\_weighted = cuda.device\_array(shape=(n,), dtype=np.float32)  d\_activated = cuda.device\_array(shape=(n,), dtype=np.float32)      normalize(d\_greyscales,out=d\_normalized)  weigh(d\_normalized, d\_weights,out=d\_weighted)  activate(d\_weighted,out=d\_activated)    # The assessment mechanism will expect `activated` to be a host array, so,  # even after you refactor this code to run on the GPU, make sure to explicitly copy  # `activated` back to the host.    return d\_activated.copy\_to\_host() |

第二个

第三个：只需要修改这个模块代码：



|  |
| --- |
| @cuda.jit  def tile\_transpose\_conflict\_free(a, transposed):  # `tile\_transpose` assumes it is launched with a 32x32 block dimension,  # and that `a` is a multiple of these dimensions.    # 1) Create 32x32 shared memory array.  tile = cuda.shared.array((32, 33), numba\_types.int32)  # Compute offsets into global input array.  x = cuda.blockIdx.x \* cuda.blockDim.x + cuda.threadIdx.x  y = cuda.blockIdx.y \* cuda.blockDim.y + cuda.threadIdx.y    # 2) Make coalesced read from global memory into shared memory array.  # Note the use of local thread indices for the shared memory write,  # and global offsets for global memory read.  tile[cuda.threadIdx.y, cuda.threadIdx.x] = a[y, x]  # 3) Wait for all threads in the block to finish updating shared memory.  cuda.syncthreads()    # 4) Calculate transposed location for the shared memory array tile  # to be written back to global memory.  t\_x = cuda.blockIdx.y \* cuda.blockDim.y + cuda.threadIdx.x  t\_y = cuda.blockIdx.x \* cuda.blockDim.x + cuda.threadIdx.y  # 5) Write back to global memory,  # transposing each element within the shared memory array.  transposed[t\_y, t\_x] = tile[cuda.threadIdx.x, cuda.threadIdx.y] |