236370 – Homework 1 Report

Submitters:

Yuval Nahon 206866832

Sahar Cohen 206824088

1) The hist kernel implementation is:

```
@cuda.jit
def hist_kernel(A, C):
    tx = cuda.threadIdx.x
    bx = cuda.blockIdx.x
    cuda.atomic.add(C, A[bx * cuda.blockDim.x + tx], 1)
```

Each Thread is responsible for one value from the input array.
By getting the thread's block number and thread id each thread is assigned to the element it needs to evaluate and add to the counter of the appropriate value. The atomic add command is **crucial** because 2 different threads can access the same data and update it according to its previous value.
Example: threads A, B read the value they were assigned to. Both read 20. Thread A reads the current counter in the result array and wants to increment it by 1. In the meantime, thread B reads the same counter and wants to increment it by 1. As a result, the counter is incremented by 1 instead of 2.

```
CPU: 9.026708356104791
Numba: 0.01613563159480691
CUDA: 0.07692737691104412
CPU speedup: 117.3406493054213
Numba speedup: 0.20975148565725227
```

The gpu_hist function was slower than the numba_hist function because the threads are dependent so the usage of the atomic command in each thread slowed down the process compared to the numba equivalent that is optimized for these cases and can still make the code parallel and efficient even with this kind of dependency (as specified in tutorial 5).

2) The matmul_kernel implementation:

```
@cuda.jit
def matmul_kernel(A, B, C):
    tx = cuda.threadIdx.x
    to_calculate = range(tx, C.size, 1024)
    for i in to_calculate:
        row_index = int(i / C.shape[1])
        col_index = i % C.shape[1]
        for k in range(A.shape[1]):
            C[row_index][col_index] += A[row_index][k] * B[k][col_index]
```

The output matrix is flattened, and each thread is assigned to handle the indices (i) that satisfy: $tx = i \ (mod\ 1024)$. This way the workload is spread (almost) evenly between the threads.
For each index the current thread is responsible, it calculates the position of the index in the output matrix, calculates the value for this position and inserts it to the output matrix.

3) Matmul_functions.py comparison:

```
Numpy: 0.5481185712851584
Numba: 1.7901637721806765
CUDA: 0.29814770305529237
```

The GPU matmul is a lot faster because each thread was assigned a computational job and no communication was needed at all. Additionally, there are no locks/atomic commands that may slow down the process since the threads are completely independent.

4) The main.py run comparison:

```
Epoch 1, accuracy 91.58 %.
Epoch 2, accuracy 94.93 %.
Epoch 3, accuracy 96.09 %.
Epoch 4, accuracy 96.39 %.
Epoch 5, accuracy 97.02 %.
Time matmul_np: 897.8704850673676
Epoch 1, accuracy 92.1 %.
Epoch 2, accuracy 95.24 %.
Epoch 3, accuracy 96.22 %.
Epoch 4, accuracy 96.66 %.
Epoch 5, accuracy 96.83 %.
Time matmul_numba: 165.20489263534546
Epoch 1, accuracy 92.43 %.
Epoch 2, accuracy 95.28 %.
Epoch 3, accuracy 96.18 %.
Epoch 4, accuracy 96.49 %.
Epoch 5, accuracy 96.91 %.
Time matmul_gpu: 209.35237765312195
Test Accuracy: 96.7%
```

As we can see the matmul_np was the slowest one of them all because it runs the code in serial manner on a **single** cpu.
The matmul_numba performed better than the matmul_gpu because the result matrix in the forward_prop multiplication is very small (it's an input vector multiplied by the weights matrix which results an output **vector**) so there are a lot of threads that have no work to do at all while other threads have a heavy calculation to process. Even in the transition between the first layer and the second layer (which are the biggest layers) only some of the threads, out of all the 1024 threads, have something to compute. And as we learned the computation power of a single gpu thread is weaker than a cpu thread's equivalent – so the numba implementation is faster overall for the forward_prop multiplication (which is performed a lot of times).

5) The main.py run comparison (with -c32 flag):

As we can see, the Numpy implementation performed much better than before, whereas the others didn't. By running the code with 32 cores, numpy libraries were able to spread the workload evenly between the cores. Numba, on the other hand, parallelizes the code only by using threads (on one core). The performance of the gpu implementation hasn't changed since the number of GPUs is the same as before.

```
Epoch 1, accuracy 92.0 %.
Epoch 2, accuracy 94.81 %.
Epoch 3, accuracy 95.68 %.
Epoch 4, accuracy 96.74 %.
Epoch 5, accuracy 96.41 %.
Time matmul_np: 79.30428409576416
Epoch 1, accuracy 90.66 %.
Epoch 2, accuracy 94.82 %.
Epoch 3, accuracy 95.69 %.
Epoch 4, accuracy 96.5 %.
Epoch 5, accuracy 96.79 %.
Time matmul_numba: 164.91183018684387
Epoch 1, accuracy 92.59 %.
Epoch 2, accuracy 94.5 %.
Epoch 3, accuracy 95.97 %.
Epoch 4, accuracy 96.29 %.
Epoch 5, accuracy 96.77 %.
Time matmul_gpu: 214.84611439704895
Test Accuracy: 96.56%
```