Concurrent and Distributed Prog. – 236370

Assignment 1 – Intruduction to parallel deep neural networks

Due Date: 04/12/22

1. I used naïve simple 2D loop for the CPU & Numba implementations.
   The loops go through the matrices elements and compare each element in both matrices to get the max.
2. For the GPU implementation, we have the assumption of enough threads in the system so each thread can work on a single cell of the output. So, the kernel become very simple.

```
43    @cuda.jit
44    def max_gpu_single_thread(A, B, C):
45        x, y = cuda.grid(2)
46        if x < A.shape[0] and y < A.shape[1]:
47            C[x, y] = max(A[x, y], B[x,y])
48
```

```
57        threads_per_block = (1, A.shape[1])
58        blocks_per_grid = (A.shape[0], 1)
```

3. The results on the server:

```
    [*] CPU: 13.368919022381306
    [*] Numba: 0.024377077817916 87
    [*] CUDA: 0.1456045601516962
(tf23-gpu) manor.zvi@lambda:~/236370/HW1$
```

4. Speedups:
   a. Numba/CPU: 556X
   b. CUDA/CPU: 92X
5. While meeting the assignment criteria, (above 40X), we see that the Numba implementation is better than the GPU. I think this is because if the grid dimension we chosen: each thread block run on a different SM, and 1024 threads per SM might be too much for the 2080ti we've got. Also, this task ha no data reuse, and very few compute compared to the memory requirement. Hence it is memory limited without option for improvement by using caching.
6. For the CPU implementation for the Matmul task, I used the fact that $X^T X$ is **symmetric**, and therefore we can calculate the result only for the lower(higher) triangular matrix and write the same data to the other half.
   It looks like this:

```
24    def matmul_transpose_trivial(X):
25        M = X.shape[0]
26        K = X.shape[1]
27        Y = np.zeros((M, M))
28        for m in range(M):
29            for n in range(m+1):
30                # for k in range(K):
31                for k in range(K):
32                    Y[m, n] += X[m, k] * X[n, k]
33                Y[n, m] = Y[m, n]
34        return Y
```

7. I used the same trick for the Numba implementation.
8. The GPU implementation is a bit trickier.

a. I didn't want to spread our 1024 thread uniformly over the output matrix, because it means that half of them will be idle, for the same reason explained in the previous section (symmetricity of $X^T X$).

b. On the other hand, they rather be idle than doing redundant work. This way at least they are not wasting register space, power, and other resources. For example: $C[0,1]$ is not calculated directly but copied from $C[1,0]$ by the thread which worked on $C[1,0]$.
I could use another thread just for the copy, but I thought that the overhead of using locks on the result is higher than the benefits.

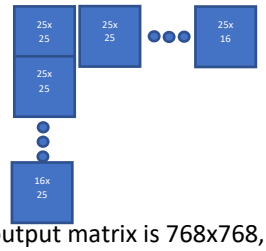
c. So I used 1D grid, and for each thread used non-linear transformation to transform each thread ID to X,Y coordinates in the output matrix, but **for the lower triangular part only**. Then, each thread writes the result he calculated into the equivalent cell in the upper triangular matrix.
The ThreadID translation to X, Y coordinates on the lower triangle is as follows:

```
x = int((math.sqrt(8 * i + 1) - 1) / 2)
y = int(i - x * (x + 1) / 2)
```

Where i is task index, and each thread gets an interleaved set of tasks:

```
for i in range(tid, n_elements, bdim):
```

d. Note: I did implement a 'blocked' version of the algorithm, where every thread gets a 25x25 section of the output matrix, and all threads above the main diagonal are idle. The last row and last column of output blocks get 16x25 or 25x16 block:



e. Out output matrix is 768x768, so we have: $\frac{784(784+1)}{2} = 307{,}720$ elements to calculate (size of the lower triangular matrix).

f. we have 1024 threads, and only 1 thread block, so threads 0-519 gets 301 elements each, and threads 520-1023 gets 300 elements.

g. Outer-Product fashion: I implemented two flavors of the algorithm: inner-product and outer product. Inner-product have the following structure:

$$
\begin{aligned}
&for\ m\\
&\quad for\ n\\
&\quad\quad for\ k\\
&\quad\quad\quad C[m,n] \mathrel{+}= A[m,k] \cdot A[n,k]
\end{aligned}
$$

This should affect negatively on the cache since we need to fetch the same k data again and again.
Outer product, on the other hand, looks loke that:

$$
\begin{aligned}
&for\ k\\
&\quad for\ m\\
&\quad\quad for\ n\\
&\quad\quad\quad C[m,n] \mathrel{+}= A[m,k] \cdot A[n,k]
\end{aligned}
$$

This should be better because after finishing one iteration of the k loop, we are never access the same k column of A again and improve out cache hit rate.

h. Quite weird, the outer product had worse performance than the inner product.
I think it is because the need to copy elements to the upper triangle, which balanced the benefits of the outer product, or the memory layout which is not visible using Numba (but surly does using raw CUDA C++).

i. Shared memory: I implemented a version called 'Vanilla Shared Memory' in which only the first k columns of A is prefetched into shared memory, and in that way all threads can enjoy it.
lambda GPU is 2080TI, with 48[KB] of shared memory per SM. Since A dimensions are 784x128, it is

enough for 7 full columns.
Prefetching was performed by all threads, interleaved between shared memory cells.
Its performance was also worse than the non-shared memory version, probably because the prefetching time.

j.

```
Numpy: 0.47112250328063965
Numba: 4.599685485474765
CUDA (Interleaving Threads, Inner Product): 3.569250321947038
CUDA (Interleaving Threads, Outer Product): 20.48097164928913
CUDA (Interleaving Threads, Inner Product, With Vannila Shared Memory): 4.131337059661746
CUDA (Blocked Thread, Inner Product): 7.075060237199068
CUDA (Blocked Thread, Outer Product): 9.70716774649918
```

k. I get that the GPU is faster than the Numba but slower than Numpy.
I think this is because there is only one thread block.
each thread block is running on a single SM, so effectively we get very low utilization of the GPU: only 1 SM is working, and the rest are idle. Each SM for itself is still quite strong vector processing machine, but not sure is better than a Xeon core.
moreover, there are a lot of overheads in GPU processing, much higher latency etc., so it makes sense that the GPU with 1 SM is slower than the Numba implementation.

l. Also, the shared memory version is not optimal. Ideally, I would have to iterate through the K dimension, prefetch each shared memory size of A columns, and use it for the next phase of outer-product computation.
nevertheless, given the poor performance of the vanilla version of shared memory, I don't think it would change the picture a lot.

m.

```python
def matmul_kernel_interleaving_threads_inner_prod_with_vannila_shmem(A, C):

    tid = cuda.threadIdx.x
    bdim = cuda.blockDim.x
    m = C.shape[0]
    k = A.shape[1]
    n = C.shape[1]
    n_elements = int(m * (m + 1) / 2)
    shmem_m = m
    shmem_k = int(SHMEM_SIZE / 8 / shmem_m)
    shmem_elements = shmem_m * shmem_k

    sA = cuda.shared.array(shape=(784, 7), dtype=numba.float64)  # TODO: Using shmem_m,shmem_k here is not working. why?

    for i in range(tid, shmem_elements, bdim):
        y = int(i % shmem_k)
        x = int(i / shmem_k)
        sA[x,y] = A[x,y]

    cuda.syncthreads()

    for i in range(tid, n_elements, bdim):
        x = int((math.sqrt(8 * i + 1) - 1) / 2)
        y = int(i - x * (x + 1) / 2)

        for kk in range(k):
            if kk < shmem_k:
                C[x, y] += sA[x, kk] * sA[y, kk]
            else:
                C[x, y] += A[x, kk] * A[y, kk]
            # C[x, y] += A[x, kk] * A[y, kk]
        C[y, x] = C[x, y]
```