Neural Network

Asignatura:
Procesadores de Alto Rendimiento
Especialidad:
Ingeniería de Computadores
Autor:
Ian Fernández
Fecha:
16 de junio de 2023

Índice

Paralelización

En este punto trataré de explicar como he orientado la paralelización de la red neuronal, la estrategía que he seguido junto con los problemas que han ido surgiendo y las soluciones que yo he encontrado a estos.

Train

Primeramente, para la parte de entrenamiento de la red, estaba tratando de llevar a cabo una estrategía que ha resultado ser errónea, que consistía en convertir las funciones de *matrix.c* en funciones de kernel y que estas pudiesen ser invocadas desde el archivo *train.c*.

Tras desechar esta estrategía, empecé la práctica nuevamente con una nueva orientación. La idea principal, es crear un solo kernel global que trate las funciones *forward_pass* y *back_prop* y se encargue de realizar todo el entrenamiento. Para esto, me he tenido que centrar en el fichero *nn.c,* que he convertido en *nn.cu* para poder usar las funciones de CUDA. Dentro de este archivo he creado una nueva función *train_GPU,* basándome en la ya existente función *train,* para intentar paralelizar la red neuronal.

```
for (x = 0; x < n_batches; x++) {
    for (min_batch = (x * size_batch); min_batch < ((x + 1) * size_batch); min_batch++) {
        i = order[min_batch];
        forward_pass(nn, &ds->inputs[i * ds->n_inputs], A, Z);
        loss += back_prop(nn, &ds->outputs[i * ds->n_outputs], A, Z, D, d);
    }

    update(nn, D, d, lr, size_batch);
}
```

Figura 1. Bucle dentro de la función train.

```
clock_gettime(clk_id, &t1);
for (x = 0; x < n_batches; x++) {
    batch_process<<<blk_in_grid, thr_per_blk>>>(d_nn, d_ds, x, size_batch, d_order, &loss, A, Z, D, d, nn->BH, nn->WH, D_aux, E);
    cudaDeviceSynchronize();
    // update(nn, D, d, lr, size_batch);
}
clock_gettime(clk_id, &t2);
```

Figura 2. Nuevo bucle dentro de la función train_GPU.

La idea de lanzar un solo kernel de esta forma es que cada hilo trate una iteración de *min_batch*. Para esto, he analizado tanto el bucle de la [Figura 1], como las funciones que procesa para saber cuales son los datos necesarios para el kernel. Una vez conociendo estos datos, he pasado a intentar almacenarlos correctamente en la GPU para que el kernel pueda operar con ellos. En mi caso he almacenado todos los datos directamente en la función *train_GPU,* excepto WH y BH que por comodidad los he almacenado directamente en la función que los inicializa *init_nn.*

En este punto he tenido que afrontar diversos problemas. Existen varios tipos de datos que se deben almacenar en la GPU y cada uno necesita de un tratamiento distinto para lograr almacenarlo de manera correcta.

Empezando por la funciones *alloc_matrix_1v* y *alloc_matrix_2v*. Resulta que estas funciones reciben un vector como argumento y reservan un tamaño de memoria distinta para cada elemento de ese vector. El no tener un tamaño determinado, ha dificultado su alojamiento hasta tal punto que he tenido que usar memoria "pinned", lo que me permite tener los datos accesibles tanto en la CPU como en la GPU. Para esto, he creado dos nuevas funciones en *matrix.cu*, *cuda_alloc_matrix_1v* y *cuda_alloc_matrix_2v* que se encargan de realizar esta tarea. De todas formas, he dejado comentadas las dos versiones del código, que he tratado de usar sin éxito para realizar esta tarea, sin hacer uso de memoria anclada[2].

Para el alojamiento de ambas estructuras de datos *nn* y *ds*, he realizado lo que he considerado más sencillo, replicar la estructura en la GPU (en la función train_GPU) y copiar los datos que se inicializan en la CPU. Estas estructuras almacenan en parte punteros a funciones, que ha sido otro de los grandes problemas que he tenido impidiéndome avanzar con la práctica. Buscando información, conseguí dar con una función de CUDA para poder realizar esto, *cudaMemcpyFromSymbol*[1]. Quiero anotar también, que solamente he inicializado los datos que se necesitan para el funcionamiento del kernel, hay datos que no he necesitado inicializar.

Aparte de estos, he almacenado también algunos vectores y matrices que necesitaba, así como matrices auxiliares que se reservaban con *malloc* dentro de *back_prop*.

Una vez realizada toda la reserva de memoria, lo siguiente que he hecho ha sido elegir el tamaño de bloque y grid que va a usar el kernel, en lo que no me he complicado demasiado. Para el tamaño del bloque he asignado todos los hilos que permite esta gráfica por bloque que son 1024. La grid no afecta mucho a la implementación específica que yo he realizado así que le he asignado el siguiente tamaño "ceil((float)n_batches / thr_per_blk)" siguiendo las directrices de prácticas anteriores.

Implementación del kernel

La implementación del kernel, se ha hecho un poco más sencilla por la estrategia de paralelización seguida, pero aún así, también he tenido varios problemas que no me han permitido terminar de paralelizar el código.

Dentro del kernel, primero se calcula el "tid" para cada hilo y se evalúa que cada hilo solamente participe en una iteración del bucle interno a la vez [Figura 3].

```
__global__ void batch_process(nn_t *nn, ds_t *ds, int x, int size_batch, int *order, double *loss, double **A, double **Z, double **D, do
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int i, j, o;

    if (tid >= (x * size_batch) && tid < ((x + 1) * size_batch)) {
        o = order[(x * size_batch) + tid];
        double *input = &ds->inputs[o * ds->n_inputs];
```

Figura 3. Cálculo del "tid" y asignación de tareas al mismo.

Para la implementación del código interno, no he tenido que cambiar mucho. Solamente tener en cuenta que ahora las matrices que se tratan son del tamaño anterior multiplicado por el tamaño del “batch” y para que cada hilo procese la parte de la matriz que le corresponde, en los índices de estos hago uso de *size_batch* y *tid*.

Dentro del kernel se hallan las funciones *forward_pass* y *back_prop* que hacen uso de las funciones *__device__* para realizar sus operaciones internas, que tampoco varían debido a que los argumentos ya se les pasan modificados. Aunque hay dos funciones específicas, que he creado yo para evitar reservar memoria dentro del kernel. Estas son *cuda_matrix_transpose_mul1* y *cuda_matrix_transpose_mul2* que como su nombre indica, realizan ambas operaciones de forma simultánea, la traspuesta y la multiplicación.

Por último, he tratado de obtener la suma acumulada del loss mediante una operación atómica sumandola primeramente de forma local para cada hilo, pero no lo he conseguido. Para este paso he tenido otro problema con *atomicAdd*[4] puesto que no era posible en un principio realizar una operación atómica con “doubles”. Modificando el makefile y agregando un flag he logrado realizarlo pero aún así no he tenido tiempo para conseguir ver por qué razón no obtengo la suma total del *loss*.

Test

Para la sección de pruebas, solamente he tenido tiempo para implementar las funciones dadas. Para cada iteración compruebo de que los *outputs* obtenidos en función a los esperados y en base a eso determino de que tipo son, si falso negativo, falso positivo, verdadero positivo o verdadero negativo.

Finalmente he implementado las funciones de precision, recall y f1 con los datos obtenidos.

Referencias

1. cudaMemcpyFromSymbol:
https://stackoverflow.com/questions/26075972/cudamemcpyfromsymbol-on-a-device-variable
2. cudaHostAlloc:
https://gist.github.com/gautambak/2836941
3. Malloc pointer **:
https://gist.github.com/nafe/733323
https://stackoverflow.com/questions/14284964/cuda-how-to-allocate-memory-for-data-member-of-a-class
4. Atomic add:
https://stackoverflow.com/questions/37566987/cuda-atomicadd-for-doubles-definition-error