Assignment #4-Evaluating Expressions

Henrique Catarino-56278

I used PyCUDA to make this project and used google collab since I don't have a NVidia GPU. As such I created 2 kernel functions:

- **mse_kernel**: This function computes the MSE between 2 arrays of integers, it will compute the unique index for each thread across all blocks **(int idx = threadIdx.x + blockIdx.x * blockDim.x;)** and from there each thread calculates the squared difference for a unique pair of elements. Saving the value for each pair in an output array
- **reduce_min:** This function is a parallel reduction algorithm designed to find the minimum value in an array. It makes use of shared memory allocation with an array in the shared memory of the blocks since is faster than global memory and is shared across all blocks. Like the previous function it will calculate the unique index for each thread and its corresponding data point in the input array. Each thread loads one element from the input array into shared memory. If the thread's global index 'i' is outside the bounds of the array (i >= n), it initializes its position in shared memory with a very large number, effectively ignoring it in the min-reduction process. Afterward the for-loop iteratively halves the number of active threads, each time comparing two elements in the shared memory and stores the minimum of the two. After the loop, the minimum value within the block is stored to the output array at the position corresponding to the block. This means each block produces one minimum value, representing the minimum of all values processed by that block.

Note: In both functions branch conflicts may occur since threads may follow different execution paths but the impact it may have is minimal if it exists at all since in both cases the condition is based on global index (**n** and **i** respectively), as such the code works accordingly but one way to prevent it is to avoid the use of if-else if possible.

With these 2 kernel functions I've modified the code accordingly, the 'a' and 'b' arrays in python stay the same as in the PyCUDA version, 'funs' stays the same as well. I have a for-loop that for each line of funs it will do the exact same as it did in the original code until a = eval(line) because the MSE will be calculated in the gpu.

After calculating 'a' we will allocate memory for the variables used in the mse_kernel function (a_gpu and mse_gpu) since b_gpu is always the same the memory allocation is made outside the for-loop. We will copy the contents of 'a' and 'b' to these variables and run the kernel with the mse_kernel function and afterwards we copy the contents of the output array (mse_gpu) to a list in the CPU side, we make the mean of all the values and save it on another list that will have all the MSE for each line of functions.txt. We repeat this process for every line. Therefore, this kernel will be called once for each line. At the end of the cycle, we free the memory.

After the for-loop we have a list with the MSE for each line of functions.txt, with this we will allocate memory for the input array and the output array of the **reduce_min** function. Copy the values of the MSE list to the input list and call the kernel to execute the computation. Next, we copy the value of the output array to a local CPU side array and will use the min python function to get the minimum MSE from each block of the function and finally we get the smallest MSE and the formula from function.txt that produced it.

The block size I've used I've tried several multiples of 32 but the results didn't change as such I've decided to use block size of 256 like the examples showed by the professor. In the mse_kernel I used grid size **grid_size = (n + block_size - 1) / block_size** being n the size of array a, this formula ensures that you have enough blocks to cover all elements in the array, even if n is not a multiple of block_size. And for reduce_min I used a similar logic, **num_blocks = int(np.ceil(len(mse_list) / block_size))** divide the total size of your array by the block size and rounding up

In analyses I make **n+1** kernel codes in my code, '**n**' being the number of lines in functions.txt this is the best number of kernels call I could do since each line of function.txt must be calculated and the minimum must also be calculated. I decided not to make any extra kernel since we were given the chance of use PyCUDA and string manipulation in python is a lot easier and simpler then in CUDA as such I left the first part in python, besides making it in CUDA could result in a worst performance.

The same principle goes to the data transfers since data transfers between the CPU and GPU is crucial for maximizing performance in GPU and making another kernel for the string manipulation and/or for calculating eval would result in more data transfer and could also result in performance issues.

The way my code is build I believe sent the most amount of data at once and minimize the frequency of data transfers as much as possible in the for-loop. While the second part of calculating the minimum MSE we only need to make 2 data transfers (from host to device and then device to host) and only 1 kernel call which improves performance.

Now to determine the combination of number of functions and number of rows in the CSV does it make sense to use the GPU vs the CPU I made several tests and in all cases I got better results with the GPU here are some of the results:

N = 800000 FEATURES = 10	CPU: 8.499212503433228 seconds GPU: 7.867822647094727
N = 100000 FEATURES = 10	CPU: 1.5455589294433594 seconds GPU: time: 1.106525182723999 seconds
N = 400000 FEATURES = 10	CPU: 4.0611467361450195 seconds GPU: 3.8772971630096436 seconds
N = 800000 FEATURES = 20	CPU: 9.752091884613037 seconds GPU: 9.258446455001831 seconds
N = 100000	CPU: 2.180751085281372 seconds

FEATURES = 20	GPU: 1.2296631336212158 seconds
N = 400000 FEATURES = 20	CPU: 4.7220213413238525 seconds GPU: 4.685572862625122 seconds
N = 10000 FEATURES = 5	CPU: 0.5137960910797119 seconds GPU: 0.2912464141845703 seconds
N = 10000 FEATURES = 20	CPU: 0.6648027896881104 seconds GPU: 0.33099961280822754 seconds
N = 800000 FEATURES = 5	CPU: 1.181427240371704 seconds GPU: 0.8372437953948975 seconds
N = 1500000 FEATURES = 20	CPU: 18.461271286010742 seconds GPU: 17.00204062461853 seconds
N = 1500000 FEATURES = 10	CPU: 16.45693588256836 seconds GPU: 13.653202056884766 seconds

As we can see by the results the use of GPU results in an increase of performance in pretty much all cases independently of the number of columns and lines of the CSV, however it seems that for larger N and smaller number of columns the performance difference between CPU and GPU is better than a large N and a larger number of columns.

Finally answering the last question of the essay: “If you were to use this program within a Genetic Programming loop (with evaluation, tournament, crossover, mutation, etc...) several iterations, how would you adapt your code to minimize unnecessary overheads?”

In order to decrease overhead the functions would have to be parallelized in the GPU kernel function, for example, for tournament, multiple tournaments can be run simultaneously on the GPU. And to minimize the data transfers I would transfer the entire population to the GPU at the beginning and make all logical processes in the kernel and only after obtaining the best solution I would make another data transfer with the final solution from the GPU to the CPU.