Programming GPUs in Python

Pedro Leite - 201906697

PyCuda Notebook

What is PyCuda?

- PyCuda is a Python package that provides an interface to Nvidia's Cuda parallel computing platform. It allows developers to maximize the power of Nvidia GPUs for general-purpose computing. PyCuda allows the user to write GPU kernels in Cuda C/C++ and then run them in Python, without having to worry about low level details.

Advantages:

- Maximizes memory usage an speeds up intensive tasks.
- Its easy to implement, since the user doesn't need to worry about the low level memory management.
- Integrates with other Python packages.

Disadvantages:

- It can only be used with Nvidia GPUs.
- May introduce some Python related overhead.
- More difficult to debug, when compared to regular CPUs.

1. Kernel Definition:

- In this code a Cuda Kernel defines a new vector, that is the result of the sum of 2 vectors.
- Each thread will handle a specific index "i" of the vector. This index is retrieved, and a vector addition is performed for each one. This operations are performed in parallel.

2. GPU Version 1:

- For starters, 3 arrays with size "N" (16) are initialized. "a" and "b" with random float numbers, and c with only 0s.

- The necessary memory in the GPU is allocated, with the size in bytes for each array.
- The data is copied from the host to the GPU.
- The kernel Cuda is defined just like “1. Kernel Definition” and compiled. The kernel as size (N, 1, 1).
- The function form the kernel is executed and the results are copied from the GPU back to the host.
- It took 0.9858 seconds.

Q1: Have you noticed that the array dimension, the grid shape and the offset calculated by each thread are all related? Increase the dimension of the arrays in both programs and play with the offset and with the dim3. Report what you understood about the distribution of threads and blocks and how the threads execute the kernel operations for each program.

3. Sequential Version (16x16):

- A random 2-dimensional array is created with RxC dimensions (16x16).
- A element-wise multiplication by 2, is performed with the previous array.
- It took 0.0013 seconds.

4. GPU Version 2 (16x16):

- Three 2-dimensional arrays with size RxC (16x16) are defined, “a” has random float numbers and “b” and “c” are filled with 0s.
- The necessary memory in the GPU is allocated, with the size in bytes for each array.
- The data is copied from the host to the GPU.
- A CUDA kernel is defined to perform element-wise multiplication of a 2D array by 2 and to store thread and block IDs. Each thread processes a specific element of the array, identified by a unique index calculated using thread and block indices. The kernel multiplies the array element at the index by 2 and records the thread ID and the block ID. These operations are executed in parallel across the GPU, leveraging its parallel processing capabilities. The kernel is launched with a block size of (C,R ,1), where “C” and “R" correspond to the dimensions of the 2D array.
- The function form the kernel is executed and the results are copied from the GPU back to the host.
- It took 0.6459 seconds.

3. Sequential Version (10240x10240):

- The same as before but there are new values for R and C.
- It took 3.2837 seconds.

4. GPU Version 2 (10240x10240):

- The values for R, C and size of the allocation in memory were updated.
- The kernel function was modified to correctly handle global indices within the bounds of the larger 2D array dimensions. This ensures that each thread operates on a unique element of the matrix without exceeding its bounds. Additionally, the block dimensions were set to (16, 16, 1), maintaining the original configuration, which divides the matrix into smaller blocks of threads.

```
# define the kernel that will run the multiplication in the GPU
mod = SourceModule("""
  __global__ void doublify(float *a, uint *a_tid, uint *a_bid, int R, int C)
  {
    int idx_x = blockIdx.x * blockDim.x + threadIdx.x;
    int idx_y = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = idx_y * C + idx_x;

    if (idx_x < C && idx_y < R) {
        a[idx] *= 2;
        a_tid[idx] = idx_x + blockDim.x * idx_y;
        a_bid[idx] = blockDim.x * blockDim.y;
    }
  }
  """)

func = mod.get_function("doublify")

# define the block and grid dimensions
block = (16, 16, 1)
grid = ((C + block[0] - 1) // block[0], (R + block[1] - 1) // block[1], 1)
```

- It took 5.0325 seconds.

Conclusions:

- Threads are the smallest unit of execution on the GPU. Threads within the same block can cooperate through shared memory.
- Blocks are groups of threads that execute concurrently on a multiprocessor of the GPU. The CUDA kernel launches multiple blocks to utilize the parallel processing capability of the GPU.

- The dimensions of the array (R rows and C columns) define the total number of elements that need to be processed. For instance, if we have an array of size 10240x10240, there are 10240*10240 = 104857600 elements in total.

- The grid in CUDA consists of multiple blocks organized in 1D, 2D, or 3D configurations. In this case its 2D, so RxCx1, when we have bigger dimensions the blocks might process a portion of the array, for example if you choose a block size of 16x16 (blockDim.x = 16, blockDim.y = 16), then you would need gridDim.x = ceil(10240 / 16) and gridDim.y = ceil(10240 / 16) blocks to cover the entire array.
- In the previous example of the 1D array was just Nx1x1.
- The offset calculation ensures that each thread accesses a unique element of the array.

- Each thread within a CUDA kernel is identified by a unique thread index. The thread index is typically calculated using a combination of “blockIdx” (index of the block in the grid), “blockDim” (number of dimensions per block), and “threadIdx” (index of the thread in the block) variables.
- When a CUDA kernel is launched, the GPU scheduler assigns blocks to available multiprocessors. Each block executes independently, and threads within the same block can cooperate using shared memory.
- Threads within a block execute concurrently. They typically process elements of a large array or perform independent computations on different parts of the dataset.

- For the smaller 16x16 arrays, the sequential CPU approach outperforms the GPU version due to its simplicity and minimal overhead. In contrast, the GPU version incurs overhead from memory transfers, kernel launch times, and thread synchronization, which diminish the speed advantage for such a small dataset.
- However, as the array size increases to 10240x10240, the sequential CPU execution time escalates significantly due to the sheer volume of data processed sequentially, resulting in a slower runtime of 3.2837 seconds. The GPU version also experiences increased execution time (5.0325 seconds). Despite the longer runtime compared to the sequential approach for the larger array, the GPU version demonstrates scalability and potential for significant speedup as datasets grow even larger, leveraging its inherent parallel processing capabilities effectively.

Q2: Inspect the nvidia-smi command and check how you can obtain the same information or more detailed information using the command line.

- “nvidia-smi” providos aa summary of all Nvidia GPUs in the system.
- “nvidia-smi -i 0” provides more detailed about a specific GPU.
- “nvdia-smi dmon” allows to monitor the GPU status and update it every second.
- We can query specific attributes of the GPU using the “—query-gpu" option followed by attributes separated by commas. For example, to

query memory usage and temperature: “nvidia-smi --query-gpu=memory.used,memory.total,temperature.gpu —format=csv”.
- For more advanced querying, you can use the “—query-compute-apps" and “—query-compute-apps=pid,gpu_name,used_memory” options to get information about running compute processes on the GPU.

RAPIDS CuDF Notebook

What is RAPIDS CuDF?

- RAPIDS CuDF is a GPU-accelerated DataFrame library developed by NVIDIA as part of the RAPIDS suite. It provides a pandas-like API for Python, optimized to leverage NVIDIA GPUs for faster data manipulation and analytics tasks.

Advantages:

- Ability to leverage the parallel processing power of NVIDIA GPUs.
- CuDF offers a pandas-like API, making it easy for users familiar with pandas to transition to GPU-accelerated computing without a steep learning curve.
- CuDF is designed to handle large-scale datasets efficiently on GPUs.

Disadvantages:

- CuDF's performance advantages are dependent on the availability of compatible NVIDIA GPUs.
- While CuDF provides a significant subset of pandas functionalities, it may not yet support all pandas operations or advanced features.
- Setting up and configuring CuDF and related RAPIDS libraries may require specific hardware configurations, software dependencies, and CUDA environment setup.

Pandas:

- Query 1: Read + only 2 columns + value_counts + groupby + head + sort_index + reset_index. CPU times: user 6.97 s, sys: 2.58 s, total: 9.55 s. Wall time: 8.44 s.
- Query 2: groupby + agg + rename + sort_values. CPU times: user 768 ms, sys: 231 ms, total: 999 ms. Wall time: 997 ms.

- Query 3: astype+ weekday.map + groupby + count + sort_values. CPU times: user 3.88 s, sys: 657 ms, total: 4.54 s. Wall time: 4.72 s.

CuDF:

- Query 1: Read + only 2 columns + value_counts + groupby + head + sort_index + reset_index. CPU times: user 6.86 s, sys: 2.36 s, total: 9.22 s. Wall time: 9.92 s.
- Query 2: groupby + agg + rename + sort_values. CPU times: user 948 ms, sys: 281 ms, total: 1.23 s. Wall time: 1.26 s.
- Query 3: astype + weekday.map + groupby + count + sort_values. CPU times: user 3.64 s, sys: 601 ms, total: 4.24 s. Wall time: 4.56 s.

Conclusions:

Operations	Pandas (CPU)	Pandas (Wall)	CuDF (CPU)	CuDF (Wall)
Query 1	9.55	8.44	9.22	9.92
Query 2	0.999	0.997	1.23	1.26
Query 3	4.54	4.72	4.24	4.56

- Query 1 has better CPU time using CuDF, but better Wall time using Pandas.
- Query 2 is quicker in Pandas.
- Query 3 is quicker in CuDF.

Q1: The cell above (Query 1) executes a series of operations. What operation(s) is(are) actually taking less time to execute using CuDF when compared with Pandas?

Operations	Pandas (CPU)	Pandas (Wall)	CuDF (CPU)	CuDF (Wall)
read	5.43	4.3	4.73	3.72
value_counts	3.81	3.97	3.05	3.08
groupby	0.251	0.249	0.247	0.246
head	0.337	0.349	0.250	0.249
sort_index	0.611	0.629	0.457	0.459
reset_index	0.616	0.614	0.627	0.624

- CuDF shows better results, in compression to Pandas, in all operations tested but the “reset_index”. “group_by” shows equivalent scores.