Assignment 4: Evaluating Expressions

Beatriz Rosa 55313

17th December of 2023

1 Introduction

In this assignment, the implementation leveraged **Numba** and **PyCUDA** to accelerate computation on the GPU. Numba's JIT compilation optimized Python functions for CUDA kernels, while PyCUDA facilitated seamless GPU integration within the Python environment. The combined use of Numba and PyCUDA aimed to enhance parallel processing capabilities and achieve significant speedup in mathematical function evaluations on this large dataset. It is also important to mention that before running the **a4_55313.py** with the solution it is necessary to run the `pip install pycuda` command.

2 How was the program parallelized?

1. **Function Preprocessing**
   - The process initiated on the CPU, where the mathematical functions were preprocessed from the "functions.txt" file.
   - In function `convert_function_string` each mathematical expression underwent a conversion to a CUDA-compatible string. Single-letter function names for example "sinf," "cosf," "tanf," "sqrtf" were replaced with their full forms ("sin," "cos," "tan," "sqrt").
   - `gen_eval_block` function then generates a CUDA evaluation block for each function, incorporating these CUDA-compatible strings.
2. **Compiling the CUDA Kernel**
   - The `compile_cuda_kernel` function takes the generated CUDA evaluation block and DataFrame block as arguments, compiles the CUDA kernel, and returns the compiled module.
3. **Memory Management - GPU Data Transfer**
   - GPU memory management is handled by converting DataFrame columns into device arrays using `cuda.to_device`. This facilitates efficient data transfer from the CPU to the GPU.
4. **GPU Parallelization - CUDA Kernel**
   - The `eval` function represents the compiled CUDA kernel, its the core of the gpu parallellization. Its launched on the GPU with the specified grid and block dimensions, **processing each function in parallel**:
     - For each function, the kernel evaluates the corresponding CUDA-compatible string generated by the `gen_eval_block` function, the conditional check (`if(idx == {i})`) ensures that each thread processes a distinct function;
     - Within each function evaluation, a loop iterates over the DataFrame items (`num_items`). For each item, the mathematical function's value is calculated and compared against the corresponding `df_y` value ;
     - The squared difference between the calculated value and `df_y` is accumulated in the `results` array. This accumulation is performed in parallel by multiple threads, harnessing the GPU's parallel processing capabilities;
     - To handle potential NaN results, the code includes a check to exclude NaN values from the accumulation and a variable to count them(`nan_count`), so that it can be subtracted from the (`num_items`) variable before calculating the mean .

- The mean is then calculated by dividing the accumulated result by the total number of items (`num_items`) .

5. **Copying Data Back to CPU**

   - Once the GPU completes the parallel computation, the results are copied back to the CPU using `driver.memcpy_dtoh`, then the minimum value index, its corresponding value, and the associated mathematical function are determined and returned.

3 Processor and GPU Specifications

- Processor: 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz with: Total Cores: 14; Performance-Cores: 6; Efficient-cores: 8; Total threads: 20.
- GPU: NVIDIA Tesla T4 that is available on Google Colab.

4 How many kernels and memory copies were used?

- I used only one CUDA kernel (`eval`) with arguments:
  - `results` - Array to store the results of the evaluation;
  - `num_items` - Number of items in the dataset;
  - `num_functions` - Number of mathematical functions to evaluate;
  - `{df_block}` - comma-separated list of arguments, where each argument represents a pointer to a device array corresponding to a DataFrame column. For example if we have 2 columns (let's say `df_column_1` and `df_column_2`) the generated code would look like this:

    ```
    __global__ void eval(double *results, int num_items, int num_functions,
    double *df_column_1, double *df_column_2)
    ```

    The kernel is executed 1000 times, which is the number of lines with expressions in the file functions.txt, each time processing a subset of mathematical functions concurrently.
- Two memory copies:
  - First Memory Copy is of DataFrame columns from the host (CPU) to the device (GPU) using function `cuda.to_device`.
  - Second memory Copy is of the Results array from the device (GPU) to the host (CPU) using `memcpy_dtoh` function.

5 How was the number of kernel calls minimized?

The number of kernel calls is minimized by using a single kernel to evaluate all mathematical functions in parallel. This is achieved by organizing the computation in such a way that each thread within the GPU kernel handles the evaluation of a specific mathematical function. The loop inside the kernel iterates over the data points for that function, performing the necessary calculations. The results are accumulated within the `results` array.

6 How was the number of data transfers required minimized?

- Allocating Device Memory Once: Device memory for the output (`dev_output`) and DataFrame columns (`dev_arrays_for_df`) is allocated once before the kernel is launched. This reduces the overhead of repeated memory allocations;
- Asynchronous Data Transfer: The `cuda.to_device` function is used to transfer DataFrame columns to the GPU. This function supports asynchronous data transfer, allowing other CPU tasks to proceed while data is being transferred.

7 Choosing the number of threads and blocks (for each kernel)

The CUDA kernel is organized to use a total of 1024 threads (4 blocks per grid* 256 threads per block).

The number of blocks was calculated based on the total number of expressions (num_lines) and the threads per block to ensure all expressions are processed.

Each thread within a block is responsible for processing one function from the input set, and the grid of blocks covers all the functions. This was done to fully utilize the parallel processing capabilities of the GPU.

8 Branch Conflicts

The project doesn't explicitly show branch conflicts, as it doesn't involve complex branching conditions within the kernel. However, it's important to handle potential divergent branching efficiently, especially in cases where expressions vary significantly in their complexity.

9 Performance evaluation

9.1 On the GPU

In my first working solution of this problem that executed in 4 minutes and 30 seconds on the first time and then in the following executions took 19 seconds, in my `gen_eval_block` function that runs on the GPU, there were several nested conditions with prioritization of element-wise operations, instead of performing them on entire arrays. To optimize the code, i tried to **eliminate the maximum nested conditions** i could and simplify them by using a conditional operator **?:** and opted for using **NumPy** for optimized array operations, allowing to take advantage of **vectorization** which turned out more efficient than using explicit loops in Python.

As a consequence, both these performance improvement measures led to achieving a decrease by nearly half of the execution time from 4 minutes and 30 seconds to 2 minutes and 35 seconds (for first time executions) and from 19 seconds to 11 seconds (for executions after the first one).

Now focusing on the execution time evolution through many run iterations, the first time took 2 minutes and 35 seconds and the executions that followed next took approximately 11 seconds. This decrease of the execution time can be explained by:

- **GPU Warm-Up**: In the first run, the GPU and associated CUDA libraries might not have been fully initialized which involves various setup tasks that can take some time;
- **Compilation Overhead**: The first time the program is run, there may be additional overhead associated with compiling the CUDA kernel, which involves converting CUDA C code to machine code that can be executed on the GPU. Once compiled, the GPU retains the compiled code, and subsequent runs can skip this compilation step, leading to faster execution;
- **Data Transfer Overhead**: The first run involves transferring data between the CPU and GPU by copying data from the host (CPU) to the device (GPU) and vice versa. Subsequent runs might reuse already transferred data, reducing the overhead associated with the data transfer;
- **Caching**: the GPU's internal caches allow the performance improvement of repeated operations on the same data, although the first run may not benefit from caching, subsequent runs can leverage cached data and intermediate results.

9.2 Sequential VS GPU approaches

The sequential version of the problem took approximately 2 seconds to execute and the GPU implementation which took 11 seconds. This discrepancy in execution times can be attributed to the data transfer overhead and the GPU parallelization advantages not being superior to the time taken transferring data between GPU and the CPU, adding to that the already mentioned absence of shared memory usage and finding the minimum value on the GPU instead of the CPU. These aspects are missing in order to take more advantage of the GPU's parallelization capabilities and its components characteristics.

Despite the successful parallelization of the mathematical function evaluation on the GPU, the full potential of parallelization was not fully realized. One implementation decision that could have enhanced performance is the utilization of **shared memory** between threads within the same block which would have allowed for faster

communication and data sharing among threads, leading to more efficient parallel processing. Unfortunately, it was not straightforward to implement, therefore was omitted in this implementation.

Another opportunity for optimization lies in the computation of the minimum score and its corresponding function on the GPU's side. In the current approach, this task is carried out on the CPU after the GPU execution. If well implemented, a more efficient strategy would involve performing a **parallel reduction** on the GPU itself to find the minimum value and its index across all threads. This would result in sending back only the essential information to the CPU, namely the minimum value and the corresponding function, rather than the entire results array. Such optimization would have consequently **reduced the data transfer overhead** and enhanced overall execution speed.

10 After which combination of number of functions and number of rows in the CSV does it make sense to use the GPU vs the CPU?

The decision to use the GPU vs the CPU depends on the total workload, expression complexity, and GPU capabilities. The GPU becomes more advantageous as the number of functions and rows increases. In this problem the CSV has 100000 rows and the functions.txt has 1000 lines. The following experiment was made varying the number of rows in the dataset and maintaining a fixed number of 1000 lines in the functions.txt:

Dataset number of Lines	Seq. (seconds)	GPU version 1st time (seconds)	GPU version 2nd + times (seconds)
100000	2	156	11
2500000	37	368	242
5000000	86	600	441

Table 1: Execution times for Sequential and GPU version with varying Dataset size

Analyzing the evolution of execution times in Table 1 reveals that, consistent with the findings in the performance section, factors such as **data transfer overhead** and the **absence of shared memory** utilization outweigh the benefits of GPU parallelization. Consequently, determining a specific combination of the number of functions and dataset rows where GPU usage surpasses CPU efficiency may not be straightforward.

11 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?

- Moving GPU Initialization Outside the Loop: such as setting the device and allocating device memory outside the genetic programming loop which would only need to be done once;
- Reusing GPU Memory: Allocating GPU memory for data that doesn't change frequently (for example the constants, data that remains constant across iterations) outside the loop therefore reusing this memory in each iteration to avoid unnecessary allocations.
- Batching GPU Operations:Evaluating multiple individuals in a single GPU kernel call rather than launching a separate kernel for each individual which would reduces the overhead of launching kernels and transferring data.
- Minimizing Data Transfer: Only transferring data between the CPU and GPU that is necessary for the genetic operations. Data that remains constant across iterations is kept on the GPU and reused.
- Evaluation on GPU: Adapting the GPU program to evaluate multiple individuals simultaneously would significantly speed up the evaluation process.
- Asynchronous Transfers: Using asynchronous data transfers between the CPU and GPU. This allows the CPU to perform other tasks while data is being transferred.