PROGRAMAÇÃO PARALELA E CONCORRENTE

Evaluating expressions

REPORT

Mestrado em Engenharia Informática
2023

Ana Teixeira - fc56336

Project's limitations

Unfortunately, the results of each implementation of the project could not be obtained. Consequently, this project lacks benchmarking data, limiting the ability to draw real conclusions and insights. As a result, this report will mostly focus on the theoretical aspects.

How did you parallelize your program?

To parallelize the program two strategies were implemented: one passing to the kernel a multidimensional array in which the rows correspond to each line of the functions.txt evaluated and the columns correspond to each line of the CSV. That means this multidimensional array has 1000 rows and 100,000 columns. The other strategy launches 1000 kernels, each one responsible for processing and evaluating a line of functions.txt, and calculates the mean of its subtraction with the y column for all the CSV. The minimum value is calculated in the CPU. The advantages and disadvantages of using each strategy is going to be shown across the report.

How many kernels and memory copies did you use?

The multiple kernels strategy implements a kernel for each line in functions.txt, so there were 1000 kernels used to parallelize the program. Each kernel receives 3 arguments, `double* evaluatedLineArray, double* y, double* mean,` so there was the need of allocating memory and transferring the data to the GPU. The single kernel strategy implements one single kernel. The kernel receives 3 arguments, `double** evaluatedLineArray, double* y, double* mean,` so there was the need of allocating memory and transferring the data to the GPU.

How did you minimize the number of kernels called? / How did you minimize the number of data transfers required?

Minimizing the number of kernels launched was initially an empirical challenge, considering that launching 1000 kernels could introduce potential communication overhead within the CPU and GPU. This is due to the fact that each kernel launch involves transferring data from the host to the device, and that requires 1000 transfers in this strategy, which can lead to a considerable cost, especially with large data arrays (two arrays of 100,000 positions).

The single kernel strategy overcomes these problems by launching only one kernel with all the data. Despite the multidimensional array being large (1000 rows and 100,000 columns), it is transferred only once, avoiding the costs associated with multiple data transfers. This makes the single-kernel strategy preferable in this context.

How did you choose the number of threads and blocks (for each kernel)?

To optimize the parallelization on GPU, the number of threads and blocks for each kernel were carefully selected, giving priority to crucial factors such as the occupancy. As direct observation of the results of the execution for both strategies was not possible, the selected strategy was the one that maximized GPU occupancy percentage, a key aspect for achieving high parallelism.

The occupancy was measured for both strategies using 256 and 1024 threads per block, common choices as these numbers are multiple of 32 which is the number of threads in a warp. To calculate the number of blocks, the formula `(NUM_ROWS + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK` was used to ensure an even distribution of threads per block.

Executing the *deviceQuery.cu* file in Google Colab provided essential information as the maximum number of 1024 threads per multiprocessor and the number of 40 multiprocessors. The occupancy was measured by the formula `NUM_BLOCKS * THREADS_PER_BLOCK / MAX_THREADS_PER_MULTIPROCESSOR`. The results are presented in the table below:

Strategy	**Threads per block**	**Number of blocks**	**Occupancy (%)**
Single kernel	256	4	1
Multiple kernels	256	391	97,75
Single kernel	1024	1	1
Multiple kernels	1024	98	98

The table indicates that the single kernel strategy is not making use of the resources on GPU, making the multiple kernel strategy the preferred choice for maximizing parallelism. Additionally, both threads per block numbers used in this chosen strategy leverage the GPU's resources, so both of them are great choices in this specific problem and we will use the 256 threads per block.

How did you use shared_memory?

Shared memory was used for both strategies in order to accumulate the results of each thread's calculations, and to apply a reduction mechanism to get/calculate the value to be returned. Each block has its own shared memory variable with a `BLOCK_SIZE` size.

In the single kernel strategy, shared memory was used to store the result of the calculation for the entire row that each thread in a block is responsible for. Subsequently, a reduction was performed to get the minimum value in the array. It is important to note that in order for the reduction to work properly, it was needed to use the *__syncthread* to ensure that the threads only pass to the next reduction phase when all the threads within the same block finished their current task.

In the multiple kernel strategy, shared memory was used to store the result of the calculation for the value each thread is responsible for. The reduction process involved calculating the sum of all values in each block, and dividing the result by the number of rows in the data.csv file to obtain the mean.

How did you take into account branch conflicts in your project?

In order to avoid having multiple threads in the same block accessing the same memory bank simultaneously, both strategies use *int idx = threadIdx.x + blockIdx.x * blockDim.x*. This expression ensures that each thread is assigned to an unique index, calculated based on its thread index within the block (*threadIdx.x*), the block index (*blockIdx.x*), and the block size (*blockDim.x*). By employing this indexing strategy, threads are directed to different memory zones, preventing conflicts.

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?

Due to the unavailability of project results, the exact number of rows and functions cannot be specified. In the event that such information were provided, the *generate_input.py* file would be adjusted to determine the numbers where it starts to make sense using the GPU. Besides that, it is known that for a small number of functions the CPU might be sufficient to handle the tasks efficiently. As the number of functions increases, the parallel processing capabilities of the GPU become more advantageous. The same way, for a small dataset with a limited number of rows, the overhead associated with data transfers between the CPU and GPU might outweigh the benefits of parallelization. In such cases, the CPU could be more suitable. As the dataset size grows, the parallel processing power of the GPU becomes increasingly valuable. Based on these considerations, it is possible to estimate 100 as the number of functions and 10,000 as the number of rows to serve as a threshold for transitioning to GPU resources effectively.

If you were to use this program within a Genetic Programming loop several iterations, how would you adapt your code to minimize unnecessary overheads?

Only one thread would be launched and each block would be responsible for doing one task. Each coding block would be responsible for performing each task and would be surrounded by an if statement, to guarantee only the threads responsible for that task execute that part of the code. After each task, a *__syncthread* is needed.

Strategy chosen

While the number of kernels launched and the number of data transferred initially suggested a preference for the single kernel approach, other crucial factors such as occupancy are very relevant, so it is important to evaluate these factors before making definitive conclusions on which strategy to use. This way, despite the minimum value of the array being calculated in the CPU, the multiple kernel strategy utilizes the GPU resources much more than the single kernel strategy, improving performance and making this the chosen strategy to parallelize this problem.

Experiment setup

In this project the Google Colab platform was used to have access to NVIDIA's T4 GPUs.