Assignment 3

Konstantin Fritzsch, Christopher Knapp

Problem 1 - Approximating Pi

Description

To parallelize the code using the GPU, we use the package `cupy`. We can easily calculate the single sums with the `bailey_borwein_plouf` function. To compare it to a not GPU-parallelized version, we change `cp.sum` / `cp.arange` to `np.sum` / `np.arange` and use `numpy` to calculate the results.

Run Code

The code is provided in `problem_1.ipynb`.

To run the code simply install the necessary packages and run the specific cell.

The first cell within the jupyter notebook with the parallel version can only be run when the computer has a NVDIA GPU and the cuda driver installed. The second cell with the sequential version only needs python and has no other dependency.

If there is no NVIDIA GPU available, you can always use Google Colab to run the notebook.

Results

Like in the last examples, we test our methods with different amount of terms. As we are going to see in the outputs, the algorithm is quite fast which is why we increased the maximum number of terms to 100.000.000 to get a better understanding of how well the algorithm scales using larger problem sizes.

The two different code cells provide us the following output:

numpy

```
Terms:  100000 , Execution time:  7.173  ms, Pi approximation:  3.141592653589793
Terms:  1000000 , Execution time:  52.927  ms, Pi approximation:  3.141592653589793
Terms:  10000000 , Execution time:  689.888  ms, Pi approximation:  3.141592653589793
Terms:  100000000 , Execution time:  5837.467  ms, Pi approximation:  3.141592653589793
```

cupy

```
Terms:  100000 , Execution time:  2.204  ms, Pi approximation:  3.1415926535897936
Terms:  1000000 , Execution time:  5.128  ms, Pi approximation:  3.1415926535897936
Terms:  10000000 , Execution time:  41.811  ms, Pi approximation:  3.1415926535897936
Terms:  100000000 , Execution time:  291.806  ms, Pi approximation:  3.1415926535897936
```

The parallelization works great. The numpy calculation is already very fast and is, even for the new large maximum term size, significantly faster than all the other versions before. Nevertheless, the cupy version is able to improve the runtime results up to 20 times and increases the improvement compared to the numpy version for increasing term amounts.

Apparently GPUs are pretty good in doing simple calculations like the bailey-borwein-plouf-formula. The many kernels can efficiently be used for calculating the single summands which is what was expected by us. The calculation for each number of terms can also be done independently to each other so there is no need for extra communication or writing back values to the cpu. That's why this problem is perfect for GPU parallelization.

Problem 2 - Applying Filters on an Image

Description

To execute and test the written algorithms we again use Google Colab with an NVIDIA GPU. The T4 GPU was mainly used in development, but the code was also tested on an A100 and V100.

The codes for the gaussian blur and the sobel filter are very similar. The only difference is that for the guassian blur we need a kernel in addition to the calculation.
We read the image, create a zero matrix like the image matrix for the resulting image. We

copy the read image matrix, the kernel and the resulting image matrix to the device.
After that we setup the grid and block sizes. This part is specific to the used GPU. Executing the code we used for the NVIDIA T4 lead to a crash on the A100 and V100 GPUs. Therefore the total amount of threads had to be lowered.
The function to apply the kernel in parallel with `@cuda.jit` annotation is executed and we can achieve great parallelization with that, what can be seen results.

Run code

To run the code, an NVIDIA GPU needs to be available and the dependencies must be installed. Then the single code block inside the jupyter notebook can just be executed with the wanted method.
Using Google Colab to run the code is recommended as we used it during development and it was very easy to use.
Just make sure to create an assets folder and upload an image and change the image's name in the python code to match the file name.

Results

Gaussian Blur

First we simply ran the serial version of the code to see the CPU performance on those machines. As expected, it's not great.

Radius	Execution Time (seconds)
1	41.35
3	42.38
5	43.47
7	47.54
9	49.49

Then we ran the parallel implementation on the NVIDIA T4 and got outstanding execution times for different threads per block and different radii. The best results were achieved with a quadratic distribution of threads per block creating a total of 256 threads with (16, 16). As it handled radii <20 so fast, we highered the numbers up to 80 and achieved a very fast execution time of only 3.42 seconds. On the CPU the same calculation would have taken significantly longer

Threads per Block	Radius	Execution Time (seconds)
(16, 16)	5	0.75
(16, 16)	10	0.18
(16, 16)	20	0.27
(16, 16)	40	0.87
(16, 16)	80	3.42 (best config)
(32, 8)	5	0.04
(32, 8)	10	0.07
(32, 8)	20	0.24
(32, 8)	40	0.92
(32, 8)	80	3.73
(8, 32)	5	0.04
(8, 32)	10	0.07
(8, 32)	20	0.25
(8, 32)	40	0.88
(8, 32)	80	3.50
(32, 32)	5	0.04
(32, 32)	10	0.08
(32, 32)	20	0.27
(32, 32)	40	1.02
(32, 32)	80	4.34

Threads per Block	Radius	Execution Time (seconds)
(10, 10)	5	0.04
(10, 10)	10	0.09
(10, 10)	20	0.29
(10, 10)	40	1.09
(10, 10)	80	4.27

The NVIDIA A100 and V100 show even lower execution times than the T4 GPU with an ideal configuration of, this time a lower amount of total threads being 64 and (8, 8). The same good execution time can also be seen with a total of 265 threads and the configuration (16, 16) and 128 threads and (32, 4) and (4, 32). It takes even less than a second for the GPUs to apply the gaussian blur on an image with a radius of 80, what is quite impressive.

Threads per Block	Radius	A100 Time (s)	V100 Time (s)
(8, 8)	5	0.25	1.45
(8, 8)	10	0.04	0.04
(8, 8)	20	0.08	0.08
(8, 8)	40	0.22	0.26
(8, 8)	80	0.71 (best)	0.89
(16, 16)	5	0.02	0.02
(16, 16)	10	0.03	0.03
(16, 16)	20	0.07	0.08
(16, 16)	40	0.20	0.24
(16, 16)	80	0.72	0.91
(32, 4)	5	0.02	0.02
(32, 4)	10	0.03	0.03

Threads per Block	Radius	A100 Time (s)	V100 Time (s)
(32, 4)	20	0.06	0.08
(32, 4)	40	0.20	0.24
(32, 4)	80	0.71 (best)	0.93
(4, 32)	5	0.02	0.03
(4, 32)	10	0.03	0.04
(4, 32)	20	0.06	0.09
(4, 32)	40	0.20	0.25
(4, 32)	80	0.71 (best)	0.91
(14, 14)	5	0.02	0.03
(14, 14)	10	0.03	0.05
(14, 14)	20	0.07	0.09
(14, 14)	40	0.22	0.31
(14, 14)	80	0.84	1.16

Sobel Filter

The sobel filter has less variability through different radii whatsoever. As we have seen in the results of the gaussian blur, the Google Colab GPU machines do not have very well performing CPUs. So the run times of the sequential sobel edge filter is between 28 and 30 seconds while on a MacBook Pro with M1 Pro chip it takes less than 10 seconds.

Run Number	Execution Time (seconds)
1	28.46
2	29.42
3	28.43

Run Number	Execution Time (seconds)
4	28.38
5	29.55

Running the filter in parallel on the GPU also gives us quite a performance improvement over running it sequentially on the CPU. For all of the tried configurations we get an impressive execution time of only 0.02 seconds. That is even faster than the numba implementation for the CPU on a MacBook Pro with M1 Pro chip which took 0.04 seconds.

Threads per Block	Execution Time (seconds)
(16, 16)	0.02
(32, 8)	0.02
(8, 32)	0.02
(32, 32)	0.02
(10, 10)	0.02

Problem 3 - Iterative Solver

Description

To parallelize this code, we use a combination of the recent strategies. First, we use the chessboard strategy like in assignment 1 and 2 so we are able to calculate the results of the "red" and "black" grid-cells independently of each other.

The main part of GPU parallelization is done by cuda. We first select the amount of threads per block (scalable) and calculate the number of blocks per grid based on the number of threads and the grid size.

This parallelization was able to be significantly improved by using cupy arrays instead of numpy arrays. To synchronize the program after each calculation (either red or black), we use the stream synchronization by cupy (`cp.cuda.stream.get_current_stream().synchronize()`).

For getting optimal results, we are testing different grid sizes together with different threads per block.

Run Code

The code is provided in the notebook `problem_3.ipynb`.

To run the code locally, the program needs the packages numpy, cuda and cupy installed. Note that cupy respectively cuda can only be installed when the computer has a NVIDIA GPU.

If you use Google Colab, you can simply upload the file and then run it.

The different parameters can be adjusted in the main method (commented). If there is no need to run it with specific parameters, the main method will run different grid sizes and multiple thread amounts per block.

Results

We get the following results using five different grid sizes and five different amounts of threads per block. The maxiterations are kept at 25.000 and the residual tolerance at 0.00005 to ensure a more precise and also more complex calculation so we are able to do a better evaluation. The results are calculated using Google Colab with a T4 GPU.

```
Running with (16, 16) threads per block...

    Grid Size: 100x100, Execution time: 4.55 seconds, after 2810 iterations residual: 4.99
    Grid Size: 250x250, Execution time: 17.56 seconds, after 11986 iterations residual: 4
    Grid Size: 500x500, Execution time: 50.74 seconds, after 24999 iterations residual: 7
    Grid Size: 1000x1000, Execution time: 82.08 seconds, after 24999 iterations residual:
    Grid Size: 1500x1500, Execution time: 144.70 seconds, after 24999 iterations residual

Running with (32, 8) threads per block...

    Grid Size: 100x100, Execution time: 3.77 seconds, after 2810 iterations residual: 4.99
    Grid Size: 250x250, Execution time: 17.39 seconds, after 11985 iterations residual: 4
    Grid Size: 500x500, Execution time: 50.27 seconds, after 24999 iterations residual: 7
    Grid Size: 1000x1000, Execution time: 81.58 seconds, after 24999 iterations residual:
    Grid Size: 1500x1500, Execution time: 144.55 seconds, after 24999 iterations residual

Running with (8, 32) threads per block...

    Grid Size: 100x100, Execution time: 3.30 seconds, after 2810 iterations residual: 4.99
    Grid Size: 250x250, Execution time: 17.46 seconds, after 11985 iterations residual: 4
    Grid Size: 500x500, Execution time: 49.38 seconds, after 24999 iterations residual: 7
    Grid Size: 1000x1000, Execution time: 80.98 seconds, after 24999 iterations residual:
    Grid Size: 1500x1500, Execution time: 143.81 seconds, after 24999 iterations residual

Running with (32, 32) threads per block...

    Grid Size: 100x100, Execution time: 3.59 seconds, after 2810 iterations residual: 4.99
    Grid Size: 250x250, Execution time: 17.12 seconds, after 11986 iterations residual: 4
    Grid Size: 500x500, Execution time: 49.76 seconds, after 24999 iterations residual: 7
    Grid Size: 1000x1000, Execution time: 81.66 seconds, after 24999 iterations residual:
    Grid Size: 1500x1500, Execution time: 144.27 seconds, after 24999 iterations residual

Running with (10, 10) threads per block...

    Grid Size: 100x100, Execution time: 3.33 seconds, after 2810 iterations residual: 4.99
    Grid Size: 250x250, Execution time: 17.93 seconds, after 11985 iterations residual: 4
    Grid Size: 500x500, Execution time: 49.90 seconds, after 24999 iterations residual: 7
    Grid Size: 1000x1000, Execution time: 81.60 seconds, after 24999 iterations residual:
    Grid Size: 1500x1500, Execution time: 144.43 seconds, after 24999 iterations residual
```

All single executions using different amounts of threads per block are giving us pretty similar results while the selection (8, 32) has the best results. The execution time rises of course when the grid size rises but the ratio is interesting:

Compared to a 100x100 grid, the 500x500 grid has 25 times as many fields but the runtime is only 15 times as long. If we compare 100x100 to 1500x1500 the instance size is 225 times the size but the runtime is only around 43 as much. We can see that there are some improvements compared to a sequential version but the parallelization is of course not optimal.

The problem with the algorithm is the necessarity to synchronize the results. We have to synchronize the results two times in every loop iteration so especially for a high amount of iterations we have a lot of synchronizations. This synchronization is necessary for the correctness of the algorithm because the calculation of red results is depending on the (finished) calculation of the black results and the other way around. The synchronization is why we are not able to use the many kernels of the GPU efficiently and improve our results the way we did it for the other two problems of assignment 3.

Note: We also tried to measure the runtime using `%timeit`. Doing that, our algorithm had a runtime of about 1ms for a 100x100 grid but we were not able to process and print the results of the calculation. Somehow timeit measures only part of the calculation and doesn't measure correct results when using gpu parallelization.