Lab 1: First application: GPUs, CUDA, and Python + Numba

Lab advanced computer architecture

Frederik Callens & Jonas De Schoenmacker
MELICTes 07/03/2021

I. Implement your first CUDA kernel in Python

1. Design a kernel function for calculating the DFT of a signal. Start off with the sequential code and accelerate it by omitting one of the *for loops*. Rely on parallel execution instead.

   The sequential code that runs on the CPU will calculate the DFT using 2 for loops. Each value will be calculated separately and added to the previous calculations. To accelerate the sequential code we can omit one of the for loops by executing it in parallel. We achieve this by running the code on the GPU and adapting it for parallel operation. We can achieve this in 2 ways:

   First of all we can eliminate the for loop that iterates over the frequencies. Now the GPU calculates all the frequency components at the same time. Figure 1 shows where the data is fetched and where it is going after computation. We see that for the first sample (for loop is in the first iteration, N= 0) we read with multiple threads of the same sample. Each thread is assigned to a frequency component and all threads will calculate at the same time. Each thread will calculate the shares of the frequency components of that sample. The function that uses this technique is DFT_parallel_easy.

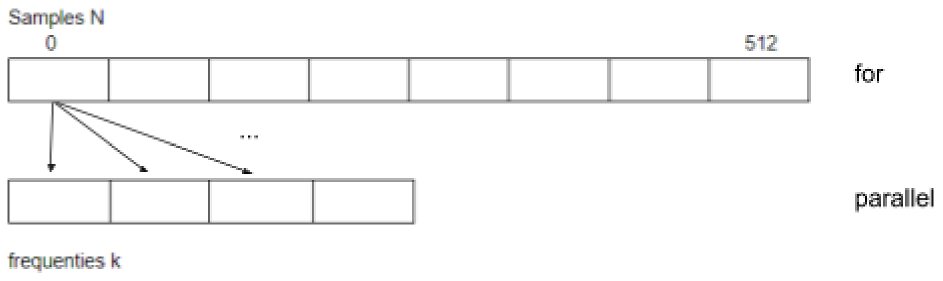

   Figure 1: How to eliminate the for loop that iterates over the frequencies.

   Secondly we can eliminate the other for loop that iterates over the samples. Now we calculate each frequency component at a time by iterating over the frequencies. As we can see in figure 2 this option is more difficult because we can read all the samples at once but we can't write each share of a specific frequency of the samples to the array at the same time. This would result in a

collision of summations of the different shares. We can solve this by using the cuda.atomic.add function which will prevent the memory being accessed when a frequency share is added. Of course this will delay the processing time, the GPU is powerful enough to calculate each share at once but is stalled because it can not write to the array.

Since the cuda.atomic.add function can't add complex numbers, we need to seperate the real and imaginary parts and store them in 2 arrays. These arrays are used as parameters in the function "DFT_parallel(samples, frequenciesReal, frequenciesIm)". If we want to calculate the absolute value of a frequency we need to combine the real and imaginary parts. We can achieve this by using "frequenciesReal - frequenciesIm*1j". This complex value is exactly the same as if we didn't use 2 arrays to save the frequency components.

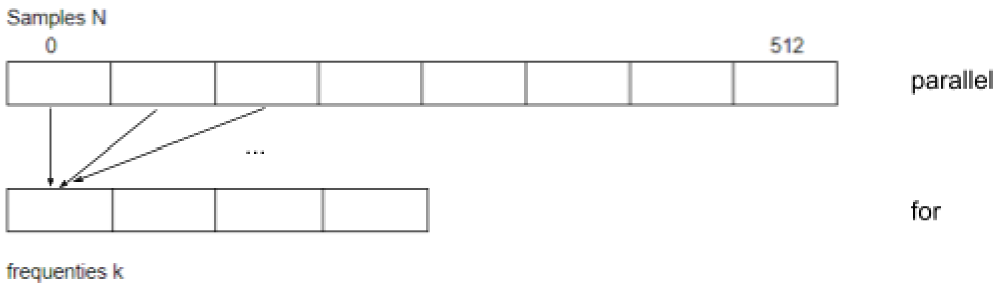

Figure 2: How to eliminate the for loop that iterates over the samples.

2. Call your kernel function and verify its output. Use the same amount of threads as is the number of output frequency components. Start off with a few hundred signal samples. Keep in mind the limit of 1024 threads / block.

   In figure 3 is the input signal and the DFT of this signal visible. If the amount of samples is less than the amount of threads available in a block and all the samples or frequencies are assigned to a thread, the output is exactly the same for DFT_parallel, DFT_parllel_easy as for the sequential CPU DFT.

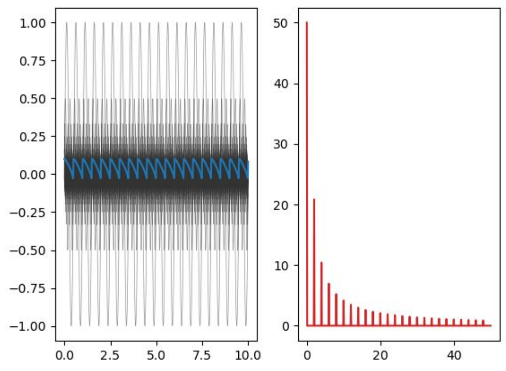

Figure 3: The input signal and output DFT of the sequential CPU and complete parallel GPU.

3. Increase the DFT resolution and with it the number of parallel threads. Does an upper limit exist? If there you can't reach it (long processing times, etc.) then reason about what the limit might be (take a look at the 3.5 compute capability specs on Wikipedia / Nvidia guide).

   An upper limit does exist. There is an upper limit for the number of threads in a block (=1024). And also a limit of blocks in a grid (=2^31 -1). If there are more threads used than are available in a block, there is an error generated (CUDA_ERROR_INVALID_VALUE). It is possible to reach the limit of blocks in a grid but the computation time will be long because there are a lot more blocks in a grid then there are threads in a block. Because of this the programmer will mostly be limited by the amount of threads.

II. Timing your code

1. Use the supplied timing function to see how fast your code executes and compare it to the performance of the sequential CPU code. Remember to run the kernel at least once before in order for it to compile.

Time for sequential [s]	**Time for parallel [s]**	**Time for parallel_easy [s]**
1.6565581560134888	0.049755644798278806	0.00603640079498291

   The time the CPU needs to calculate the DFT sequentially is a lot more than the GPU accelerated DFTs. The time needed by DFT_parallel is more than for DFT_parallel_easy because like we mentioned earlier the cuda.atomic.add function will add a delay because the GPU can not write the calculated value directly to the memory.

2. Increase the DFT resolution and observe the GPU and CPU code processing times. What is the time difference?

Number of samples N	**Time for sequential [s]**	**Time for parallel [s]**	**Time for parallel_easy [s]**
100	0.020245933532714845	0.003197669982910156	0.0011138200759887695
200	0.08240597248077393	0.005283141136169433	0.001382136344909668
300	0.1608484983444214	0.007907629013061523	0.0017423152923583985

400	0.27644355297088624	0.011020064353942871	0.002057337760925293
500	0.4341402769088745	0.014892172813415528	0.002370738983154297
600	0.6179552555084229	0.02016472816467285	0.003369903564453125
700	0.8370244979858399	0.02633323669433594	0.0037317752838134767
800	1.0978435754776001	0.034163498878479005	0.004891753196716309
900	1.4096882820129395	0.039416718482971194	0.0054713010787963865
1000	1.6565581560134888	0.049755644798278806	0.00603640079498291

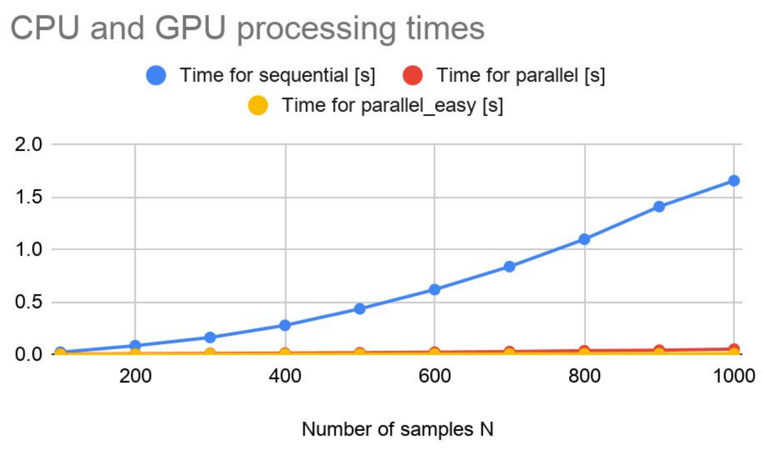

Figure 4: Time of computation CPU and GPU, increasing by the number of samples N.

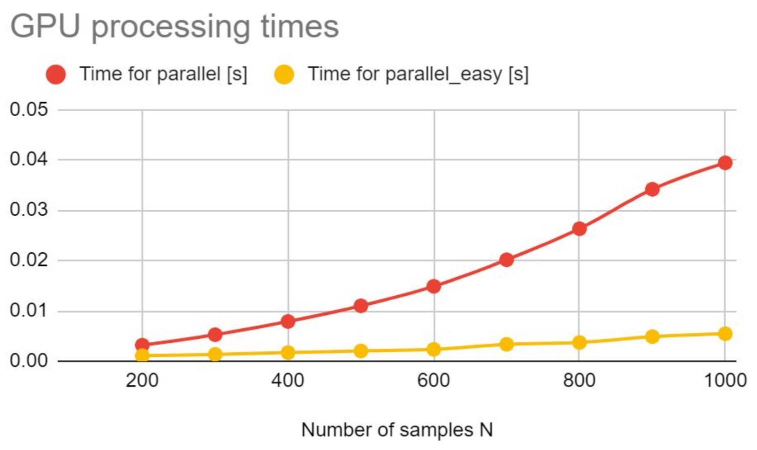

Figure 5: Time of computation GPU, increasing by the number of samples N.

The CPU time of computation increases linearly with the number of samples N. On the other hand, the time that the GPU needs to run the DFT is a lot shorter and doesn't increase as fast. This can be seen in figure 4. If we zoom in on the

GPU processing times (Figure 5), it’s visible that the time for DFT_parallel increases more than DFT_parallel_easy. This is expected, the cuda.atomic.add function will affect the performance of DFT_parallel. The frequency component shares can’t be added to each other faster than the time needed to add one share. Hereby there will be a build up of shares that needs to be added. The DFT_parallel_easy doesn’t have this problem which results in a faster processing time.

3. Now adapt your kernel function to run the entire DFT using a single thread on the GPU and observe the difference in processing time. What happens to the execution time?

Number of samples N = 1000	**Time for sequential [s]**	**Time for parallel [s]**	**Time for parallel_easy [s]**
Multi thread	1.6565581560134888	0.049755644798278806	0.00603640079498291
Single thread	1.7020869970321655	0.07168829441070557	0.0034636735916137697

The time needed for computation on the CPU sequentially does not change. This is expected. The DFT_parallel function needs more time to finish even though it calculates the frequency components from only 1 sample. The DFT_parallel_easy function needs less time which is to be expected because only the first frequency component is calculated completely and is correct. The other components are all zero. The two results are incomplete. This can be seen in the figures 6 and 7.
If more threads are assigned to the samples for DFT_parallel the DFT will be more complete but not fully complete like can be seen in figure 8. The same for the frequencies for DFT_parallel_easy the threads will compute more total frequencies of all the samples. The output will result in 25 frequencies if 25 threads are assigned. This can be seen in figure 9. There are only two spikes because there are only 2 frequency components present in the input signal between 0 (DC) and 25.

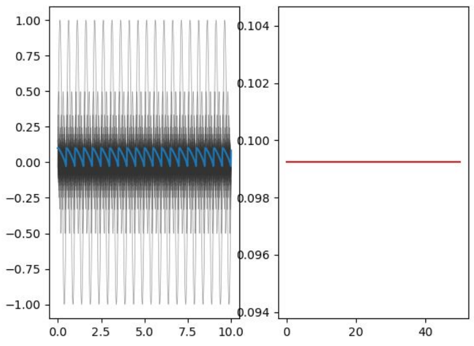
Figure 6: DFT_parallel 1 sample

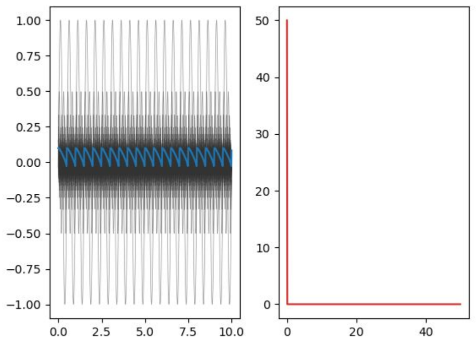
Figure 7: DFT_parallel_easy 1 frequency

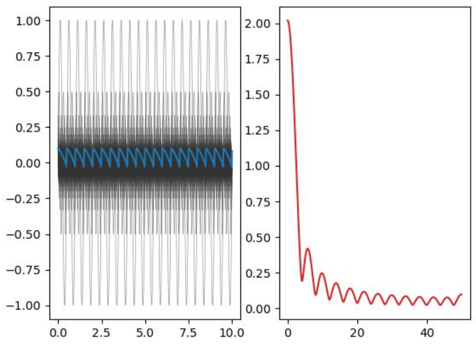
Figure 8: DFT_parallel 25 samples

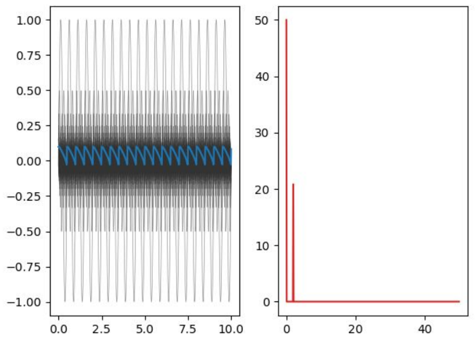
Figure 9: DFT_parallel_easy 25 frequencies

4. For an input sequence of several hundred samples, execute the DFT on the GPU in a fully sequential manner (single thread), entirely in parallel, and for a few values in between (a thread needs to loop over several but not all frequency components). Roughly how is the execution time dependent on the number of threads?

Blocks	**Threads**	**Time for sequential (s)**	**Time for parallel (s)**	**Time for parallel_easy (s)**
1	**1000**	1.6697031736373902	0.04949791431427002	0.010490727424621583
10	**100**	1.6625480651855469	0.02923274040222168	0.006981563568115234
20	**50**	1.67161226272583	0.03175914287567139	0.007020282745361328
25	**40**	1.7526648044586182	0.039806151390075685	0.00933685302734375
40	**25**	1.6812180995941162	0.033866167068481445	0.00787975788116455
50	**20**	0.00787975788116455	0.026555609703063966	0.00917513370513916

100	**10**	1.7058769464492798	0.022429299354553223	0.017113590240478517
1000	**1**	1.7054619789123535	0.07001450061798095	0.1444690704345703

The timings we have, conclude that with most ratios (blocks to threads), it takes the same time. Only when watching the extreme cases 1 block with a 1000 threads or a 1000 blocks with 1 thread show a deviation from the aforementioned pattern. Using 1 block with a 1000 threads takes a bit longer than in most other cases. But when we use a 1000 blocks with each 1 thread there is a bigger difference, especially with the DFT_parallel_easy function. This is visible in figure 10. We believe this is caused by the controller. We think it takes more time to write and read the results of the calculation than the calculation itself and there is an overhead time to switch between 2 blocks. This is why the fewer blocks that need to be iterated to get the results the faster the total calculation will be.

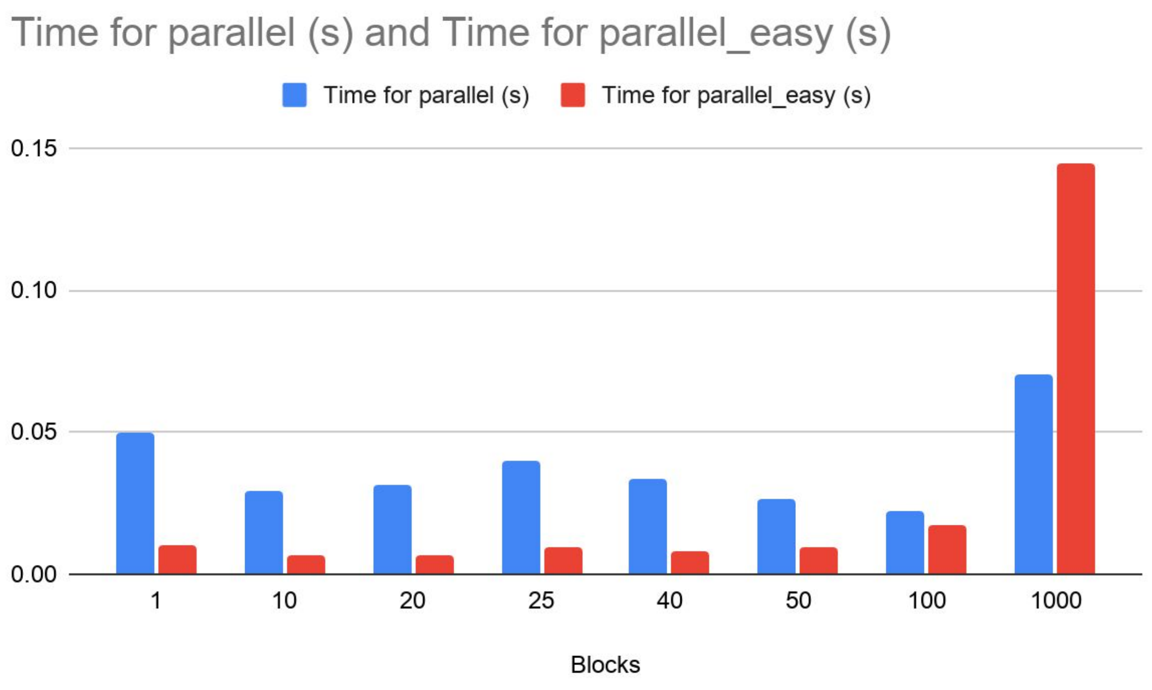

Figure 10: Processing time for a different number of threads and blocks.

This is the link to the spreadsheet file:
https://docs.google.com/spreadsheets/d/1HFt2GhMVucLzv3mI_TICflt6ditFJuZ2nO2A0G9unBQ/edit?usp=sharing
Link to code:
https://github.com/jonasdeschoenmacker/Lab-advanced-computerarchitecture/tree/main/Opdracht%201/versie%202