Advanced computer architecture
Lab1: Getting started

by Steven IMPENS & Nino SEGERS

MELICT
2020-2021

Mentor: Alexander MARINSEK

Contents

1 Introduction

In this lab we get acquainted with the core concepts of computing on GPUs. We do this with some exercises in python, specifically the nNmba package that uses the Nvidia CUDA toolkit. We first look at an example that leverages the power of GPU computing and its performance is compared to that of the CPU. After we create some kernel functions and compare the time it takes to compute the result, we then compare those result.

The code repository can be found at:
https://github.com/imstevenxyz/geavanceerde-computerarch

2 DFT

2.1 Kernel functions

From the given CPU based code we create two kernel functions, one that is completely parallel and one that only uses one thread.

The complete parallel function is seen in listing 2.1, here we remove one for loop from the CPU based function. To make sure that each sample gets a single thread we need to map them. For example, if we have 500 samples we enter `[1,500]` in the function as seen on line 1. By this we mean 1 block with 500 threads, a total of $1 * 500$ threads and so mapping each sample to only one thread.

```python
1  kernel_parallel[1,500](sig_sum, frequencies_real, frequencies_img)
2
3  @cuda.jit
4  def kernel_parallel(samples, frequencies_real, frequencies_img):
5      '''Use the GPU for generating the dft. In parallel.'''
6
7      # Calculate the thread's absolute position within the grid
8      x = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
9
10     sample = samples[x]
11
12     for k in range(frequencies_img.shape[0]):
13         cuda.atomic.add(frequencies_real, k, ((sample * (cos(2 * pi * k * x / N)))))
14         cuda.atomic.add(frequencies_img, k, ((sample * (-1* sin(2 * pi * k * x / N) ))))
```

Codefragment 2.1: *DFT parallel kernel*

2.2 Timing results

In figure 2.1 we see the timing performance of the three functions using different sample timings. To make sure we have comparable times we use the supplied `synchronous_kernel_timeit` function and always compile the kernel function beforehand. It is clear that the parallel kernel function is the best performing of the three as it stays relatively flat. The CPU function as thought is the worst performing but if we use `@numba.jit` 2.2 we get similar timing results as the GPU parallel function.

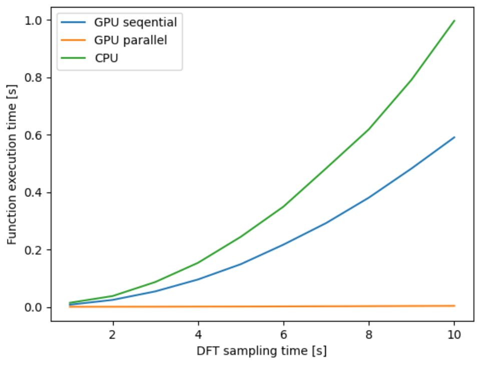

Figure 2.1: *Timing performance*

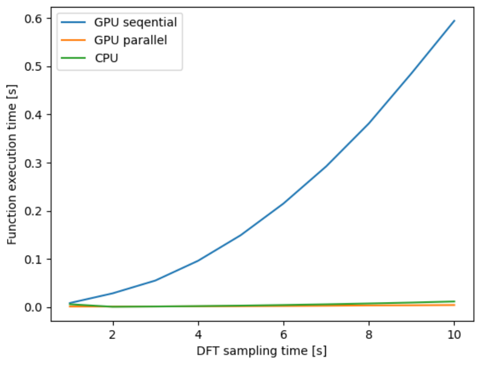

Figure 2.2: *With @numba.jit*

2.3 Semi parallel

3 Conclusion

References