Assignment 2

Part 1 (Evaluating RBF Kernels)

In the module unit on CUDA we discussed the following code to evaluate radial basis functions.

```
from numba import cuda

@cuda.jit
def rbf_evaluation_cuda(sources, targets, weights, result):
    local_result = cuda.shared.array((SX, nsources), numba.float32)
    local_targets = cuda.shared.array((SX, 3), numba.float32)
    local_sources = cuda.shared.array((SY, 3), numba.float32)
    local_weights = cuda.shared.array(SY, numba.float32)

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y

    px, py = cuda.grid(2)

    if px >= targets.shape[0]:
        return

    # At first we are loading all the targets into the shared memory
    # We use only the first column of threads to do this.

    if ty == 0:
        for index in range(3):
            local_targets[tx, index] = targets[px, index]

    # We are now loading all the sources and weights.
    # We only require the first row of threads to do this.

    if tx == 0:
        for index in range(3):
            local_sources[ty, index] = sources[py, index]
        local_weights[ty] = weights[ty]

    # Let us now sync all threads

    cuda.syncthreads()

    # Now compute the interactions

    squared_diff = numba.float32(0)

    for index in range(3):
        squared_diff += (local_targets[tx, index] - local_sources[ty, index])**2
    local_result[tx, ty] = math.exp(-squared_diff / ( numba.float32(2) *
numba.float32(sigma)**2)) * local_weights[ty]

    cuda.syncthreads()

    # Now sum up all the local results

    if ty == 0:
        res = numba.float32(0)
        for index in range(nsources):
            res += local_result[tx, index]
        result[px] = res
```

We created a threadblock layout so that each threadblock evaluated all sources with a given number of target points. This works well for small numbers of source points, but not for larger numbers of sources. Your task is to extend this code to deal with larger numbers of sources.

The idea is the following. Create threadblocks of size 16 x 32 which are responsible for 16 targets and 32 sources. Each threadblock now evaluates its own local result, which is an array of size 16 and writes the result back into global memory. You then create a second kernel that runs over the intermediate results and sums up the intermediate results into the final sum.

Let's make this precise. We assume that we have m targets and n sources. We therefore need to create a grid of (ℓ, p) thread blocks with $\ell = (m + 15)/16$ and $p = (n + 31)/32$, where the division is to be understood as integer division.

- Use the CUDA memory transfer functions to copy the sources and targets to the compute device.

- On the compute device create an array for the intermediate results that is of size (m, p).
- Launch threadblocks that evaluate the partial sum for 16 targets and 32 sources at a time and store the results in the corresponding part of the (m, p) array.
- Then launch a summation kernel of m threads, where each thread sums up the p numbers in its row of the intermediate array (m, p) and store into a result array of dimension m.
- Finally, copy the end result back to the CPU and return to the user.

Be careful with memory transfers. We do not want to transfer the intermediate array back and forth between CPU and GPU. So you need to manually create the device arrays and launch the CUDA kernels with the device arrays.

Demonstrate that your code is correct by validating it with the Python Numba implementation that we have written and show that up to single precision accuracy your result agrees with this implementation. **You will lose significant marks if you do not validate your code as without validation the correctness is not demonstrated.**

Show benchmark results and experiment with how many sources and targets you can evaluate in a reasonable time (around a few seconds of runtime). Also, separately demonstrate benchmark times for the memory transfers from the CPU to the GPU and back, and benchmarks for the actual computation on the device.

Part 2 (Evaluating the discrete Laplacian on GPUs)

In the Lectures we have discussed the function `discretise_poisson` that generates a sparse matrix which is the discretisation of the Laplace operator with zero boundary conditions. In this exercise you are asked to create a GPU kernel, which given a one-dimensional array of values $u_{i,j}$ in the unit square grid (see the corresponding notebook on the discrete Laplace problem), evaluates this discrete Laplace operator without explicity creatiing a matrix.

The idea is as follows.

Create N^2 threads. Each thread checks if its associated with a boundary value or an interior value. If it is associated with a boundary value, just write the corresponding $u_{i,j}$ value in the result array (because our sparse matrix was just the identity for boundary values). If it is associated with an interior point, write the value of evaluating the 5 point stencil for the corresponding interior point into the result array.

Validate your code against the matrix vector multiplication with the matrix created in the `discrete_poisson` function. Your code must return the same result as it evaluates the same computation. Again, lack of validation will lead to substantial loss of marks.

Benchmark your code for growing matrix sizes n. What is the ratio of data movements to computations in your code?

Describe how you could use shared memory to improve the memory behavior and reduce global memory accesses. You need not provide a code implementation of your idea. But try to be precise.

Assessment of the coursework

- The submission deadline for the coursework is **Monday 9 November, 10am**.
- Each part counts 50% of the total mark.
- The assignment does not require much code writing. But a strong emphasis is put on good explanations. Putting a few comment lines in your code is not sufficient as explanation. Use the Jupyter notebook capabilities and write good explanations about what you are doing and why you are doing it as markdown cells.
- Your code must be executable without any errors from scratch in Jupyter by choosing "Restart kernel and run all cells." If this produces any errors, any code and explanations after the error occurs will be ignored for the marking. It is not the task of the markers to debug your code.
- The code should not run for much longer than 2 minutes on a typical laptop/desktop. This is a soft limit. If your notebook runs too long we reserve the right to reject it.
- In addition to core Python packages you are allowed to use Numpy, Scipy. Numba, matplotlib and the timing functions from Jupyter/IPython. No other packages are allowed and any such request will be rejected.
- You must submit your solution as a single Jupyter Notebook file with ending `*.ipynb`. **Any other submission will lead to 0 marks automatically. Make sure you submit a correct Notebook file with the right ending.**.