ACCELERATOR BASED PROGRAMMING
UPPSALA UNIVERSITY
FALL 2022

EXERCISE 1: A REDUCTION IN CUDA

This is a preparation exercise for the second assignment.

1. Exercise Goal. The goal of this exercise is to implement a reduction with CUDA. The corresponding C code we aim to reproduce is

```
void sum(int N, float *vector, float *result)
{
        *result = 0.f;
        for (int i = 0; i < N; ++i)
                *result += vector[i];
}
```

Due to the massively parallel nature of the GPU, we need to be careful to coordinate the work. To make this work, we use the advance CUDA feature of shared memory. Shared memory is shared by a thread block and can be used as a user-managed cache or scratchpad for a cooperative algorithm like in our case. It is set up as
`__shared__ float buffer[block_size];`
Furthermore, we use barriers for threads within a block to avoid race conditions when using shared memory, using the code
`__syncthreads();`
These concepts are also discussed in the lecture on cuda programming part 2.

In terms of the reduction algorithm, we follow the description by Mark Harris,
http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.
The tasks for this lab are as follows:

- Familiarize yourself with the concept of thread blocks and grids from the lecture.
- Read the report by Mark Harris and follow the reduction variants.
- Implement your code e.g. using `stream_triad_cuda.cu` as a starting point and then modifying the vectors used in the code and replacing the vector update by the reduction.
- We deviate in one place from what Mark Harris suggests: Rather than writing into a separate element for each array block, we want to use an atomic addition into the final single float field for the result. This means that we replace the code
  `g_odata[blockIdx.x] = sdata[0];`
  by the operation
  `atomicAdd(result, sdata[0]);`
  We need to ensure that result is a device pointer, so we need to allocate a single float variable with cudaMalloc, and use a `cudaMempy` to copy the result from the device to the host. Furthermore, since we repeatedly add into this variable when repeating the summations, we need to also include a call to
  `cudaMemset(result_device, 0, sizeof(float));`
  before we launch the CUDA kernel.
- Ensure to check the final result.