GPU matrix-vector multiplication

Exercise 4:

Write a CUDA C program for matrix-vector multiplication on the GPU - you can use the code from last week as a starting point.

1. Make a naive version (each thread computes one element of the output vector) that does NOT use shared memory.
2. Add timing of the kernel execution and of the CPU $\leftrightarrow$ GPU data transfers separately. Experiment with different thread block sizes to get the best runtime. *Consider the kernel runtime from here on.* What is the most serious problem with this kernel?
3. Write a new version that uses 2D thread blocks and atomicAdd. For simplicity you may assume that the matrix size is an integer multiple of the thread block size. Hints:
   - The most efficient is not to have one thread per element of the matrix but instead to have every thread do more elements of a row (i.e. keep the for loop but make it jump with the number of threads in the y dimension).
   - Remember to clear the output before using atomic add operation on it. This can be done with `cudaMemset()` before the kernel launch.
   - ONLY SMALL CHANGES at a time! A correct result is your lifeline.
4. (Optional) Write a new version that uses shared memory for reading the **A** matrix in order to improve the runtime.
   - Block the loop (as in Assignment1) into blocks equal to the thread block size.
   - Use static allocation, e.g., `__shared__ double A_smem[BS][BS]`, where `BS` is a constant at compile time (`#define BS ??`).
   - Threads within a block should collaborate on loading data into shared memory. Make sure to remember `__syncthreads()`.
5. Compare the kernel runtime to your fastest OpenMP version. Did you get any speed-up? Is it as you would expect?
6. Make a version that can run simultaneously on two GPUs by splitting the task equally between them (you may assume that the matrix size is an equal number).
7. Compare with the `DGEMV` function for GPUs provided by Nvidia's CUBLAS library.