# Exercise 5

*360.252 - Computational Science on Many-Core Architectures*
WS 2020

November 18, 2020

The following tasks are due by 23:59pm on Tuesday, November 24, 2020. Please document your answers (please add code listings in the appendix) in a PDF document and email the PDF (including your student ID to get due credit) to karl.rupp@tuwien.ac.at.

You are free to discuss ideas with your peers. Keep in mind that you learn most if you come up with your own solutions. In any case, each student needs to write and hand in their own report. Please refrain from plagiarism!

> "I've been imitated so well I've heard people copy my mistakes." — Jimi Hendrix

There is a dedicated environment set up for this exercise:

https://gtx1080.360252.org/2020/ex5/.

To have a common reference, please run all benchmarks for the report on this machine.

## Inclusive and Exclusive Scan (4 Points)

Given a vector $\mathbf{x} = (x_0, x_1, \dots, x_{N-1})$ of size $N$, the *inclusive scan* of $\mathbf{x}$ is given by the vector $\mathbf{y} = (y_0, y_1, \dots, y_{N-1})$ with

$$y_k = \sum_{j=0}^{k} x_k \ .$$

Similarly, the *exclusive scan* operation applied to $\mathbf{x}$ results in a vector $\mathbf{z} = (z_0, z_1, \dots, z_{N-1})$ given by

$$z_k = \sum_{j=0}^{k-1} x_k \ .$$

Based on the exclusive scan code provided at the URL above, do the following:

1. Describe the workings of the kernel in your own words (provide figures as you see fit). (1 Point)
2. Provide an implementation of *inclusive scan* that fully reuses the *exclusive scan* implementation (without modification). (1 Point)

3. Modify the *exclusive scan* code to convert it to an *inclusive scan* implementation. No additional memory allocations permitted. (1 Point)

4. Compare the performance of the exclusive scan implementation and the two inclusive scan implementations above. Explain any differences you observe. (1 Point)

# Finite Differences on the GPU (5 Points)

After successfully implementing a conjugate gradient solver on the GPU, you find that the system matrix assembly takes a significant amount of time. Therefore, you decide to assemble the system matrix right on the GPU.

For a regular $N \times M$-grid in two dimensions and discrete indices $(i, j)$ to denote the respective grid point, the $k$-th grid node in the $i$-th row and the $j$-th column is given by $i * N + j$. With this, you deduce after some book-keeping that the $i * N + j$-th row of the system matrix $\mathbf{A} = (a_{i,j})$ is given by:

$$\begin{aligned}
a_{i*N+j,i*N+j} &= 4 \\
a_{i*N+j,(i-1)*N+j} &= -1, \text{if } i > 0 \\
a_{i*N+j,(i+1)*N+j} &= -1, \text{if } i < N-1 \\
a_{i,i*N+(j+1)} &= -1, \text{if } j > 0 \\
a_{i,i*N+(j-1)} &= -1, \text{if } j < M-1
\end{aligned}$$

Because you want to reuse your code in the future for other finite difference stencil, you decide to implement the following general assembly routine:

1. Write a kernel that counts and stores the number of nonzero entries for each row of $\mathbf{A}$. (1 Point)

2. Use the *exclusive scan* from above to form the row offset array of the CSR format. (1 Point)

3. Write a kernel to write the column indices as well as the nonzero matrix values to the remaining CSR arrays. (1 Point)

4. Write the remaining code to allocate the necessary arrays and call a conjugate gradient solver. Choose the right hand side vector for the linear system to be a vector of all $1$s. (1 Point)

5. Benchmark your code for different system sizes (for simplicity, choose $N = M$). Compare the time of a naive CPU-based matrix assembly (as provided) with the GPU-based assembly and the time spent in the CG solver. (1 Point)

# Bonus point: Visualize Results

Take the result vector from the finite difference solution above and plot it on the unit square $[0, 1] \times [0, 1]$.