Group 1: Florian Schrittwieser, Vladislav Válek, Maximilian Burr, Leandro Borzyk, Rohit Beer

Exercise 8.1:

Primary Contribution: The primary contribution of the paper is the development of an automated code generation approach for stencil computations on GPU architectures, addressing the challenges posed by shared scratch-pad memory and bank conflicts. The paper presents a Domain-Specific Language (DSL) for describing stencil computations and formalizes the class of stencil computations considered, enabling efficient code generation for overlapped tiling on GPUs. Additionally, the paper provides experimental evaluation of the proposed approach on four GPU platforms, demonstrating its effectiveness in optimizing stencil computations.

Key Insight of the Contribution: The key insight of the contribution lies in the development of compiler algorithms for automated GPU code generation, specifically targeting stencil computations. By enforcing constraints on the DSL to facilitate transformation to efficient code and addressing the challenges of time-tiling and bank conflicts, the paper provides a comprehensive solution for achieving high performance with stencil computations on GPU accelerators. The emphasis on overlapped tiling and the experimental validation further highlight the significance of the proposed approach.

Opinion on the Paper: The paper offers a valuable contribution to the field of high-performance computing by addressing the challenges of optimizing stencil computations on GPU architectures. The automated code generation approach and the formalization of stencil computations using a DSL demonstrate a thorough and systematic methodology for achieving efficient code generation. The experimental evaluation on multiple GPU platforms adds credibility to the proposed approach. Overall, I find the paper to be a significant and well-executed contribution to the scientific literature on GPU optimization for stencil computations.

Rating: Accept

Exercise 8.2:

Conway's Game of Life is a cellular automaton devised by mathematician John Conway in 1970. Despite its simple rules, the Game of Life exhibits complex and unpredictable behavior. It is played on a grid of cells, each of which can be in one of two states—alive or dead. The game progresses through generations, with the state of each cell determined by its neighbors in the previous generation.

The rules are straightforward: a living cell with fewer than two neighbors dies (loneliness), a living cell with two or three neighbors survives, and a living cell with more than three neighbors dies (overcrowding). Dead cells with exactly three living neighbors come to life. These rules give rise to various patterns, including oscillators, gliders, and stable structures.

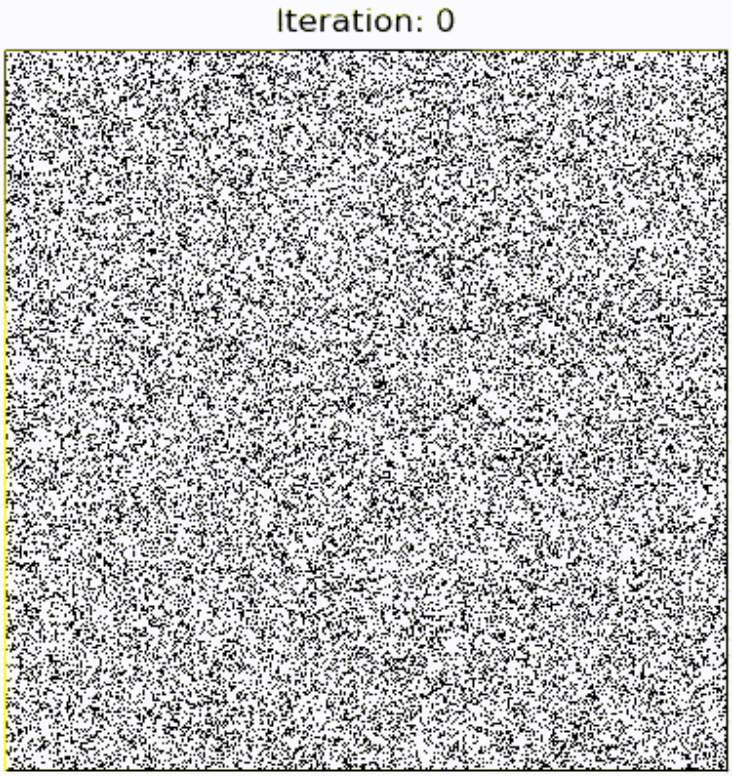
The primary problem for Conway's Game of Life is to explore and understand the emergent behaviors that arise from its simple rules. Given an initial configuration of living and dead cells on a grid, the challenge is to observe how the system evolves over generations and discover interesting patterns, oscillators, gliders, or stable structures that emerge during the simulation.

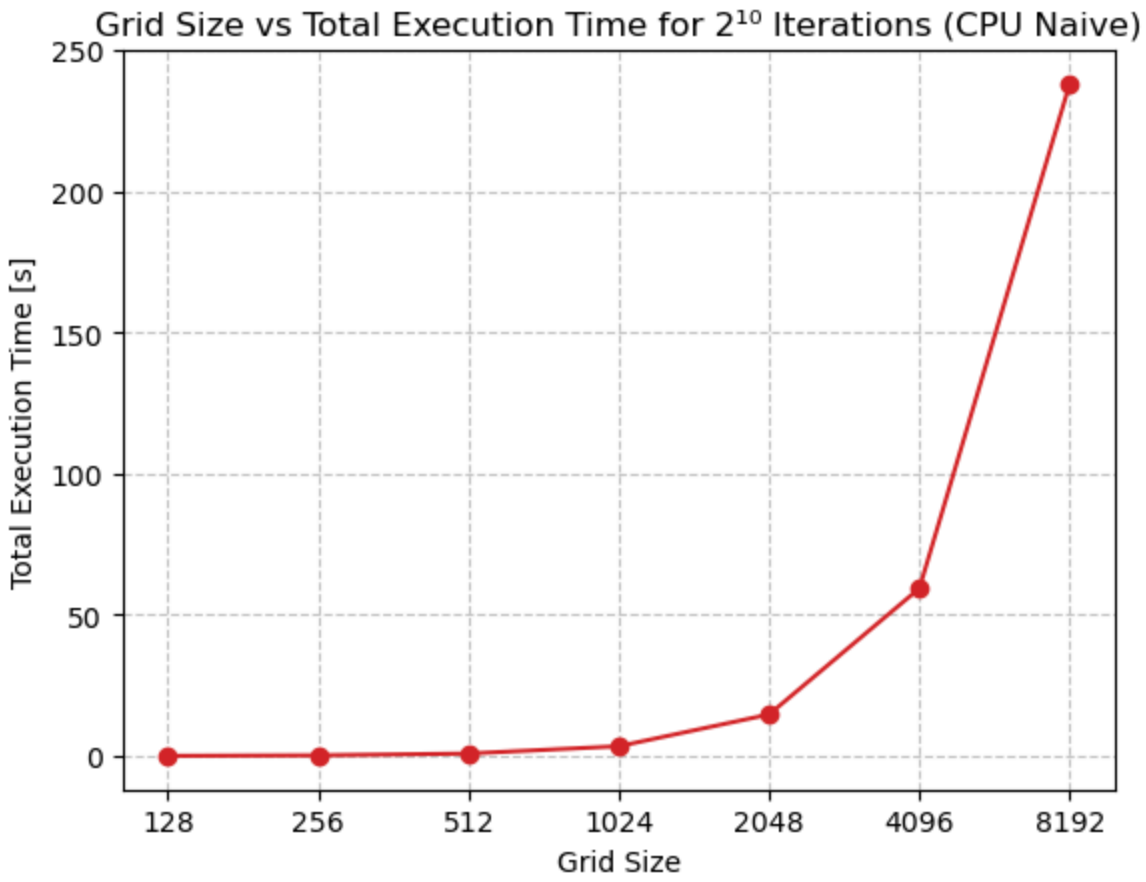
For a simply implementation of Game of Life we iterated through a grid and check the neighbors of the grid for rule evaluation. Regarding to the rules given the state of the cell we are currently

updating we set the state of the current cell and advance to the next cell. We store 2 copies of the grid. The new updated state of the grid and the actual one.

Increasing the grid size directly results in exponential growth of the execution time.

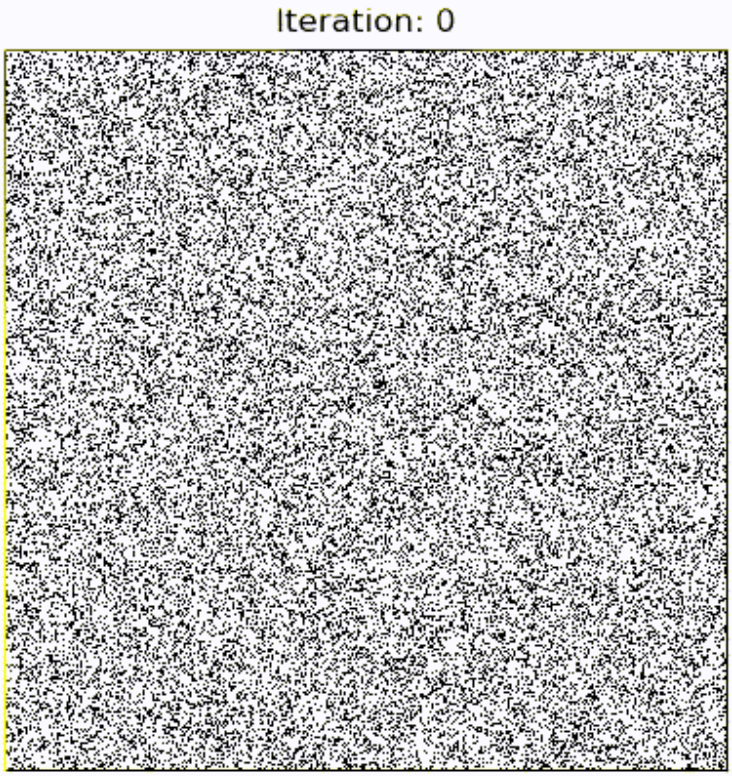
Correctness Check: We placed a animated GIF for correctness check of the outputs in the source code directories.



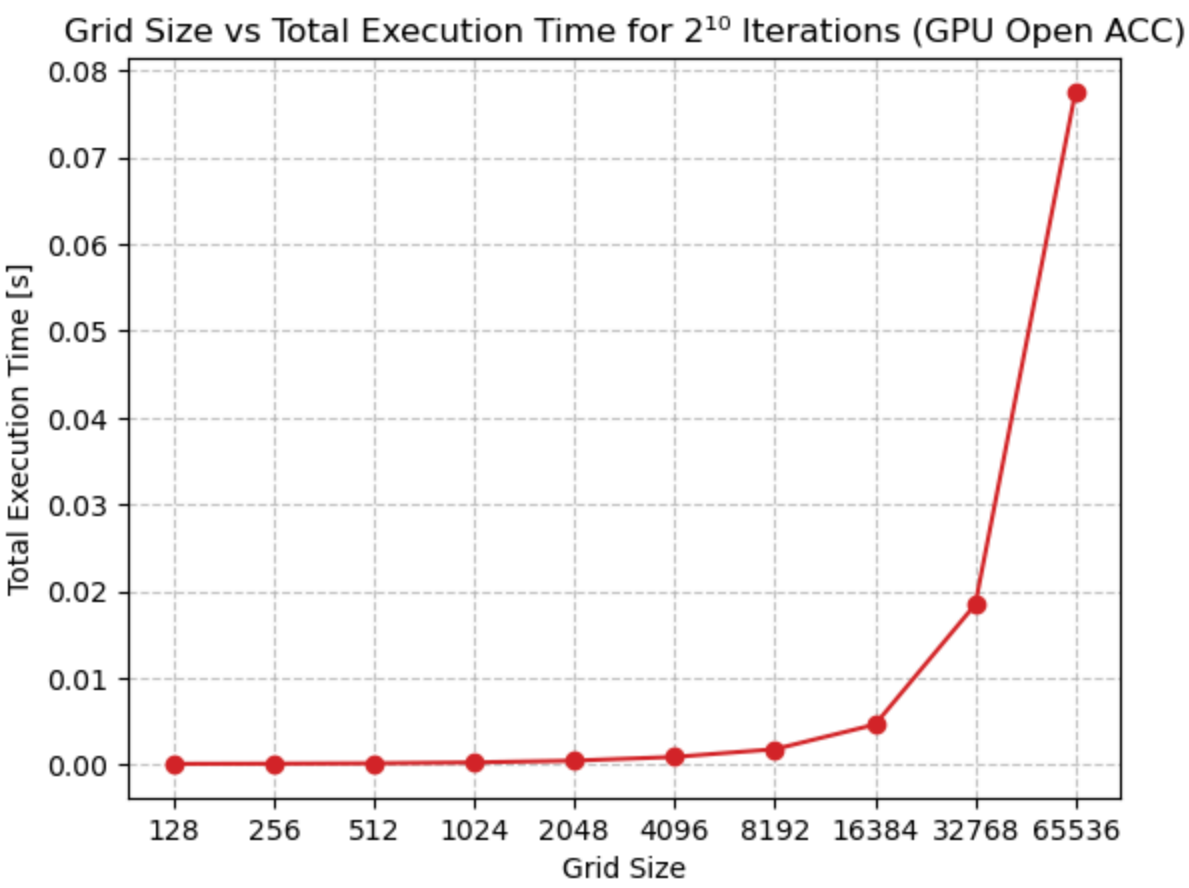


Exercise 8.3:

Correctness Check: We placed a animated GIF for correctness check of the outputs in the source code directories.

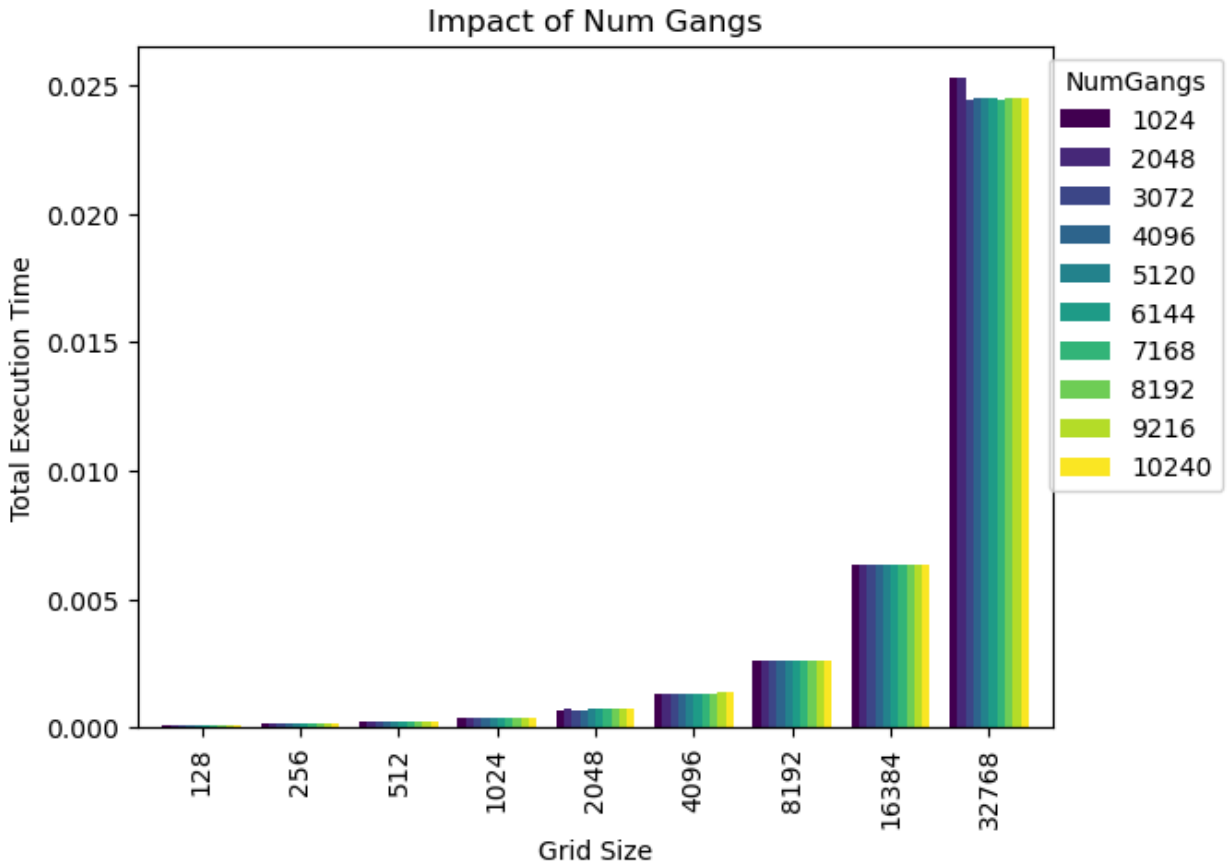


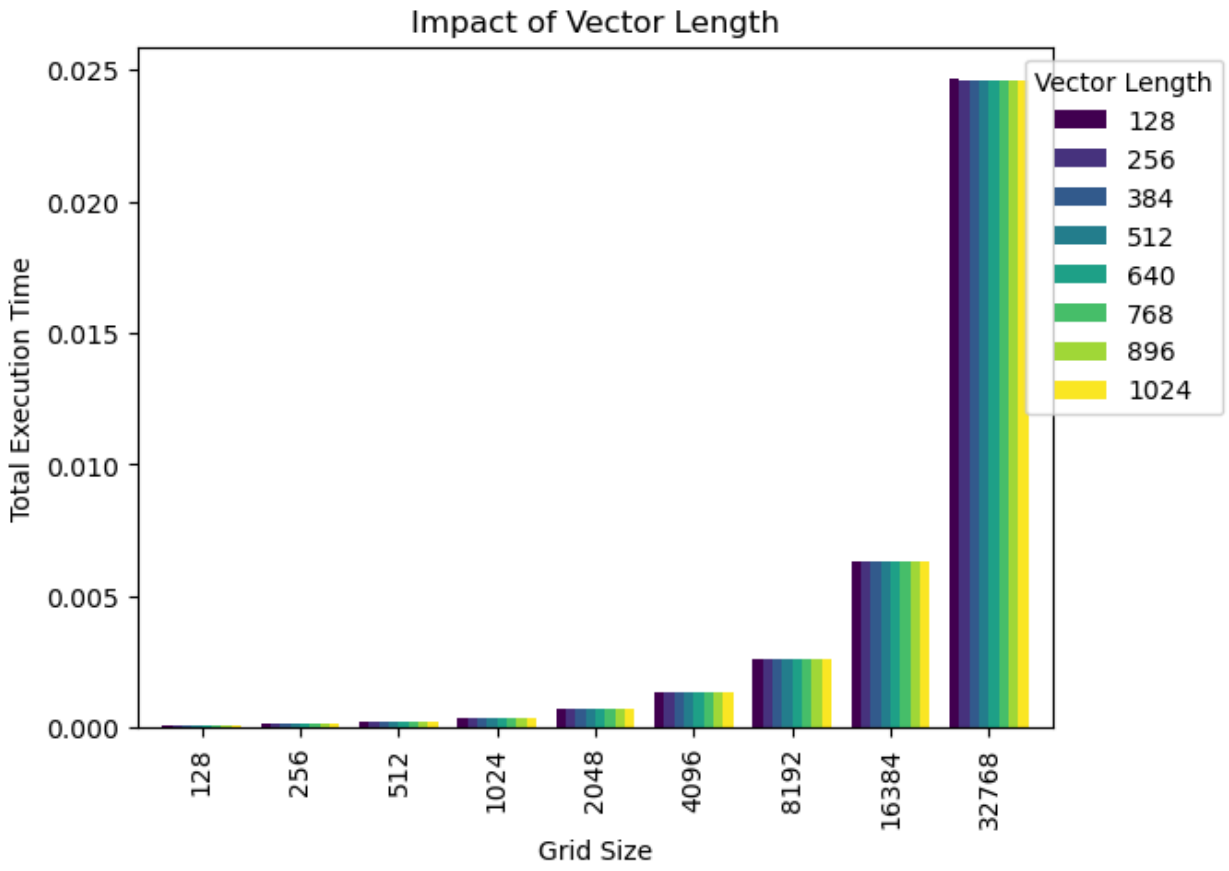
After accelerating the Code with openACC Directives we could improve the speed a lot.
The gameoflife rules results only in direct neighbors data connections to update the state of a cell. At the end of iteration the grid gets stored. And the new grid state gets copied over.
To address the borders we used a copy of the opposite row/column. This allows the programm to calculate the borders correctly.

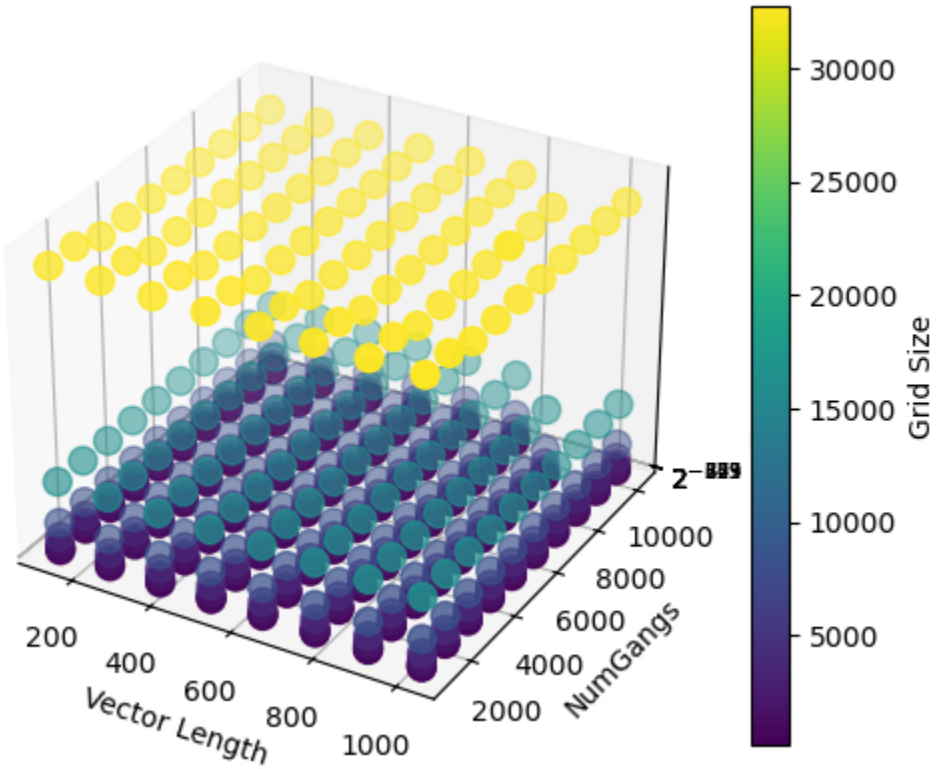

Grid Size vs Total Execution Time for 2^{10} Iterations (GPU Open ACC)
Total Execution Time [s]
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
128
256
512
1024
2048
4096
8192
16384
32768
65536
Grid Size

Exercise 8.4:

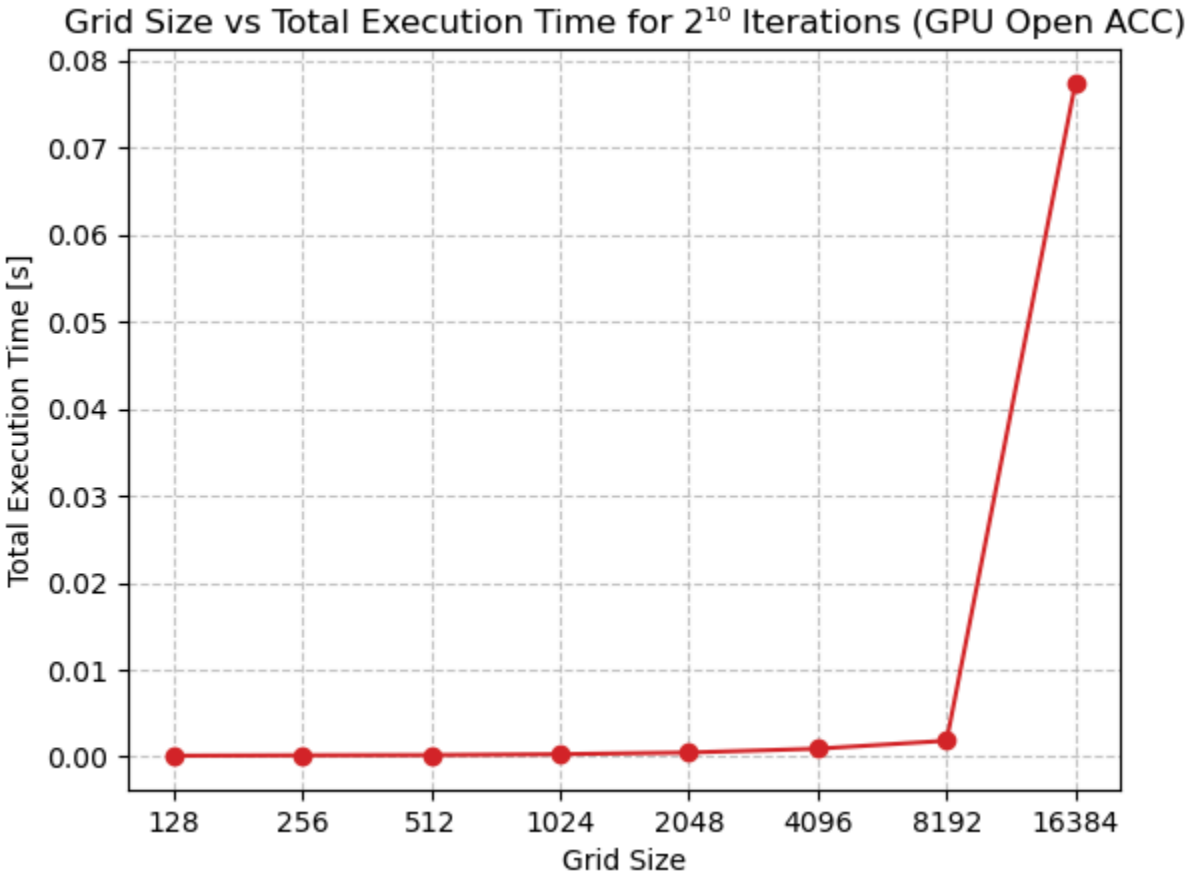
1. The vector Length and Num Gangs as shown below had no great impact on the calculation of the Grid of Game of Life. We think that we are memory bound because the grid has to distributed to all threads after each iteration.




Impact of Vector Length
Total Execution Time
0.025
0.020
0.015
0.010
0.005
0.000
128
256
512
1024
2048
4096
8192
16384
32768
Grid Size
Vector Length
128
256
384
512
640
768
896
1024


200
400
600
800
1000
Vector Length
2000
4000
6000
8000
10000
NumGangs
30000
25000
20000
15000
10000
5000
Grid Size

2. Setting the Cache Directive results in lower runtime. But with larger sizes the programm crashed. We had no further time to investigate the error.



3. At the time when we tested the correctness of the GameofLife Implementation we write the state after each game iteration. This resulted in a big increase of the Execution Time.

Willingness to present:

8.1 -> YES
8.2 -> YES
8.3 -> YES
8.4 -> YES