CS3210 Assignment 2

Aaron Sng (A0242334N)

25 October 2021

Machines Used

- xgpc1 - Tesla V100
- xgpd3 - Titan V

Program Design

Memory Usage on the GPU

Global memory on the GPU is generally used throughout the program. There will be three main matrices that will be stored on to the global memory.

- World → the current state of the world
- Temporary World → a intermediate world used to store the subsequent generation
- Invasion Plans → any invasion plans if it happens at a particular generation

These matrices may take up a lot of memory in the GPU, depending on the world size. Hence these matrices will be prioritised in its use in the global memory.

Parallel Programming Pattern

The program submitted was similar to the master-worker parallel programming pattern featured in assignment 1. In essence, upon creation of the simulation, one master thread and n "worker" threads are spawned. In the case of implementing in CUDA, the "worker" threads correspond to thread blocks. These thread blocks represent individual Streaming Multiprocessors (SMs). Within these thread blocks, these sub-tasks are decomposed further to finer granularity to suit the GPU architecture.

Task Decomposition

The master thread first divides the program for each SM to work with on a row basis. Row basis was used to increase spatial locality of the program. Within each SMs, the program divides the tasks further according to the number of columns present in the system. This generates specific subgrids which each GPU thread will operate on. Hence, for K rows with n thread blocks, each thread block will perform computation for approximately K/n rows. Suppose the world has L columns and m threads are spawned within each thread block, that means each thread will work on a subgrid of K/n x L/m. After all the threads are done with their allocated subgrids, the master thread will then allow the next generation of the simulation to be computed. It also means that a total of $m{\cdot}n$ threads will be spawned.

Work Distribution - Blockwise Implementation

Similar to assignment 1, this implementation of assignment 2 uses a blockwise work distribution technique, operating on a checkerboard. It will attempt to distribute the work to all threads such that each of threads will receive roughly equal workload. There is one caveat however, which is that the program will distribute on two dimensions - row and column. In other words, while the program strives for equal distribution of tasks, there will be some form of variation of work load for each thread. Figure 1 illustrates this particular point.

Typical Workload	**Minimal Workload**
L/m $\begin{pmatrix} 1 & 1 & . & . & 1 \\ 1 & 1 & . & . & 1 \\ . & . & . & . & . \\ . & . & . & . & . \\ 1 & 1 & . & . & 1 \end{pmatrix}$ K/n	L/m - 1 $\begin{pmatrix} 1 & 1 & . & . & 1 \\ 1 & 1 & . & . & 1 \\ . & . & . & . & . \\ . & . & . & . & . \\ 1 & 1 & . & . & 1 \end{pmatrix}$ K/n - 1

Figure 1: Variation of Workloads between Threads

Eliminating the use of Mutual Exclusions

As in assignment 1, it is generally thought that mutual exclusions (MUTEX) incur losses in parallelism, specifically when updating the current world and as well as other variables such as death toll. To reduce the use of MUTEXes, a buffer that spans the total number of threads spawned would be used to store the death toll. After each thread is done working, the death toll as determined by the thread will be written to the array stored in the global memory.

Coalescing Memory Accesses

In the case of GPUs, coalescing memory access is ideal to achieve parallelism. With each thread tasked to K/n x L/m of data, this implies that the each thread might not be able to make efficient memory accesses across rows as they are not contiguous memory locations. To coalesce memory transactions, it is hence ideal to have $K/n \rightarrow 1$. Having such a condition been met, memory accesses can be minimised, increasing speed of the program. However, such a condition might not be met as input sizes become greater than the total number of SMs available on the GPU.

Optimising for Warps

Within each thread blocks, only a specified set of threads can run simultaneously at once. These set of threads and its corresponding size largely depends on Compute Capability (CC). However, for both GPUs available for testing, the maximum size is 32 threads. Going beyond this number, and the GPUs will have to run multiple warps at once, thereby reducing the overall parallelism of the program. This implies that there is a limit to the granularity of each individual thread before parallelism gains are lost.

Work Distribution - Block-Cyclic Implementation

The purpose of this implementation is to investigate the effect of different work distribution techniques as discussed in the lectures. One in particular that appears to be rather promising is the block-cyclic work distribution. In this implementation, an arbitrary number of n thread blocks are created. Within each thread block, m threads are spawned within each thread block. Supposing the world has L columns, each thread will operate on L/m columns. In this implementation, a block size of 1 x L/m is defined. Each thread operates on 1 row as it attempts to increase spatial locality and increase the likelihood of unified memory transactions, thereby speeding up the program. Figure 2 illustrate the description of the workload.

Workload for one thread
L/m
(1 1 . . 1)

Figure 2: Description of Workload for one Thread

Distribution of Blocks to Threads

For this implementation, each block is fixed at strictly sized at one row. After one thread is done with one row, the thread will proceed to perform on the next block. The order which this is decided is static, and it depends on the index of the thread. If the thread happens to take a row that's index is large, it will not proceed and terminate. A dynamic scheduling is thought to require global communication. It was thus not desired as the goal of the program is to minimise the total number of global memory transactions. The below code block briefly describes the block-cyclic implementation.

```
while blockIndex < nRows
{
    for each col in work_load // work_load is determined by the thread index
    {
        goi game logic, operating on world[blockIndex][col]
    }

    blockIndex += blockCount; // blockCount refers to the number of SMs used
}
```

Figure 3: Pseudo Code Describing the Cyclic Distribution of Blocks

Eliminating the use of Mutual Exclusions

Similar to the blockwise implementation, a buffer that spans the total number of threads spawned would be used to store the death toll. The final death toll will be computed sequentially upon completion.

Optimising for Warps and Reducing Memory Accesses

Within each thread blocks, 32 threads can run simultaneously at once. To minimise the memory transactions, it is hence ideal to reduce the block size 1 x L/m, where $L/m \rightarrow 32$ (128 bytes). Depending on the world size, it might not be possible to reduce the total number of warps and coalesce memory transactions. This will be investigated further in the discussion section.

Code Changes made to Block-Cyclic Implementation

Other than the static scheduling introduced with the block-cyclic implementation, there were subsequent changes made to the code over the blockwise implementation.

- Removal of `row_start_collection` and `row_end_collection`. These arrays dictated the general size of each block size during the blockwise implementation. As each thread block operates on a singular row, the threadIndex computed by each thread will be used to determine which row to compute.
- Increase in the size of `death_toll_collection` buffer. The size of the `death_toll_collection` is no longer `blockCount` x `threadCount` but rather `nRows` x `threadCount`.

Special Considerations during Runtime

Virtual Arrangement

In both implementations (blockwise and block-cyclic), the program does not take into account the virtual arrangement of threads within the block or blocks within the grid. In other words, a grid of 2x2x2 arrangement spawns the same configuration of a grid of 8x1x1. In both cases, the programs will run similarly.

Maximum number of Thread Blocks and Threads

As with CUDA Compute Capability 7.0, it is not possible to spawn more than 1024 threads within a thread block. Another thing is that spawning more thread blocks than rows incurs unnecessary redundancies as the program will work on the some rows more than once.

Methods

Running Experiments

After compiling the program with `make build`, a bash script known as `profile.sh` was used to understand the performance of the program.

```
# Profile the performance of the program.
chmod +x profile.sh
./profile.sh
```

The files `goi_cuda.cu` and `goi_cuda_baseline.cu` consist of lines that will produce the wall clock time. The values obtained from running the wall clock time will be used to profile the program. The program was ran on two different types of GPUs - Tesla V100 and Titan V. Take note that due to submission requirements, these lines to measure wall clock have been commented out.

Results

Unless otherwise specified, the program input and its corresponding execution time would be a 3000x3000 size matrix ran on 10,000 generations.

Assumptions

- No other programs in the background were running
- Background processes do not have significant impact on the execution of the simulation

Fastest OpenMP Execution Timings

The fastest OpenMP execution timings were recorded. This would be measured against the recorded timings from running the Game of Invasions on the GPU. It maybe observed that both CPUs on these devices are largely similar in terms of technical specifications.

- xgpc1 - 937.70 seconds
- xgpd3 - 945.64 seconds

Number of Registers in a Kernel

To empirically determine the optimal number of row-wise processors, the following script was ran.

```
# Obtain the number of registers used in a kernel.
nvcc -m64 -c -Xptxas="-v" goi_cuda.cu
```

It was determined that the kernel used in the program used a total of 32 registers. Given 65536 registers reside on the GPU, and assuming each block takes up one warp (32 threads), the number of blocks before more than one block occupying one SM was determined.

$$N_{blocks} = \frac{N_{registers}}{N_{warp} \cdot 32} = 64$$

Figure 4: Number of Blocks before more than one Block occupy one SM

In general, it is desired to have one block occupying one SM. Having such a condition being met would allow all the thread blocks terminate together in one step, thereby improving the general speedup of the program. Hence, it is desired to have a program that runs within this constraint. However, it is not as practical since most of the world sizes vary greatly in terms of size.

Blockwise Work Distribution Results

For figure 5, the number of threads in thread blocks were fixed at 32. For figure 6, the number of thread blocks were fixed at 360. For figure 7, the number of threads in thread blocks were fixed at 32. For figure 8, the number of thread blocks were fixed at 360.

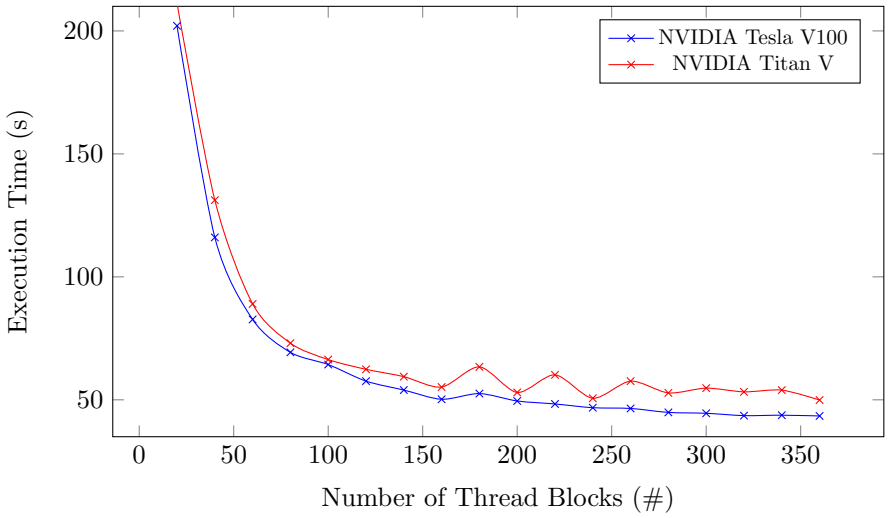

Figure 5: Blockwise Execution Time against Thread Blocks

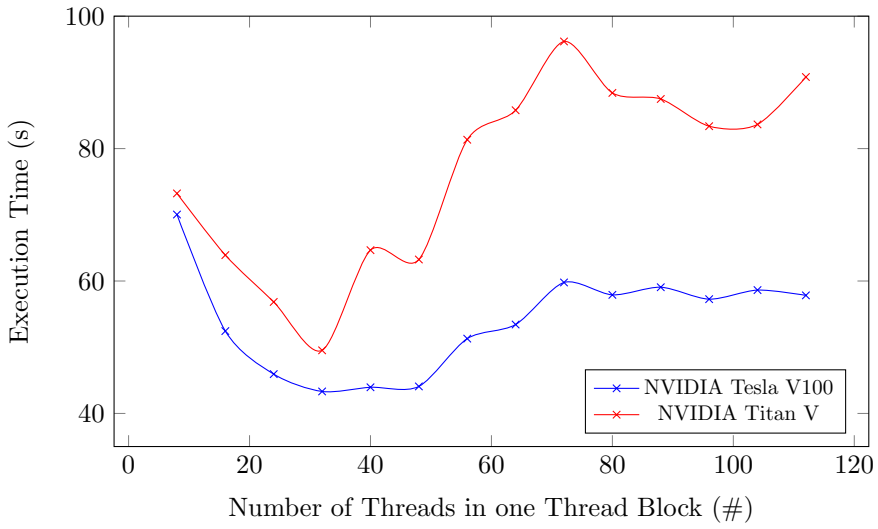

Figure 6: Blockwise Execution Time against Size of Thread Blocks

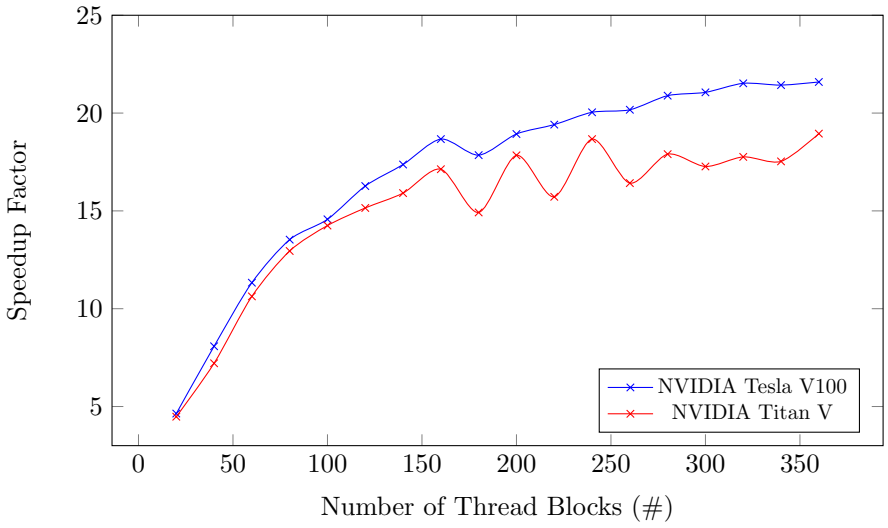

Figure 7: Blockwise Speedup Factor against Thread Blocks

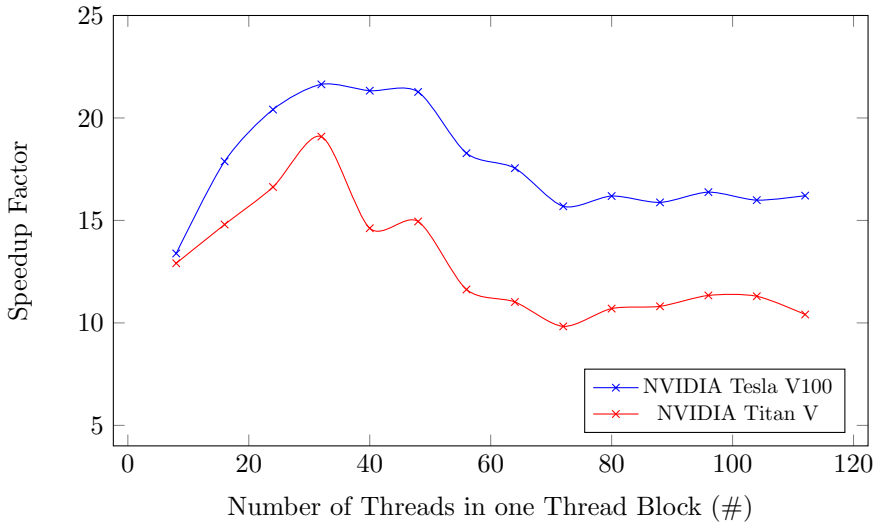

Figure 8: Blockwise Speedup Factor against Size of Thread Blocks

Block-Cyclic Work Distribution Results

For figure 9, the number of threads in thread blocks were fixed at 32. For figure 10, the number of thread blocks were fixed at 360. For figure 11, the number of threads in thread blocks were fixed at 32. For figure 12, the number of thread blocks were fixed at 360.

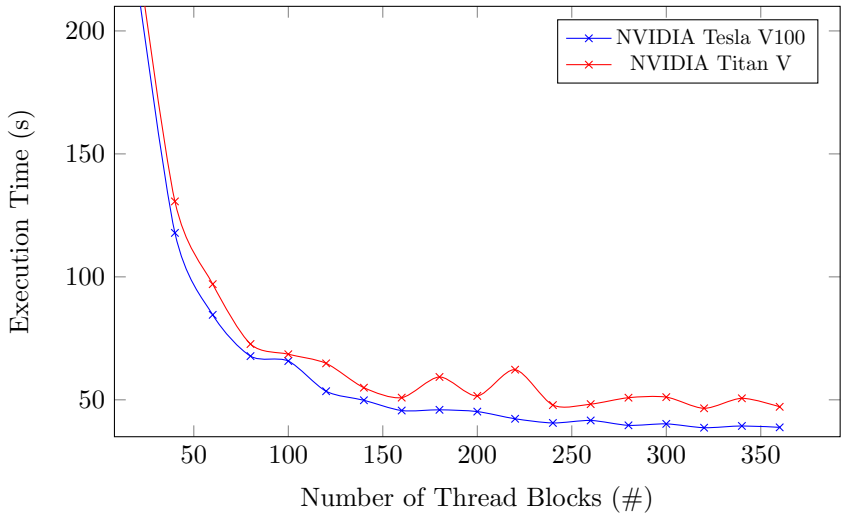

Figure 9: Block-Cylic Execution Time against Thread Blocks

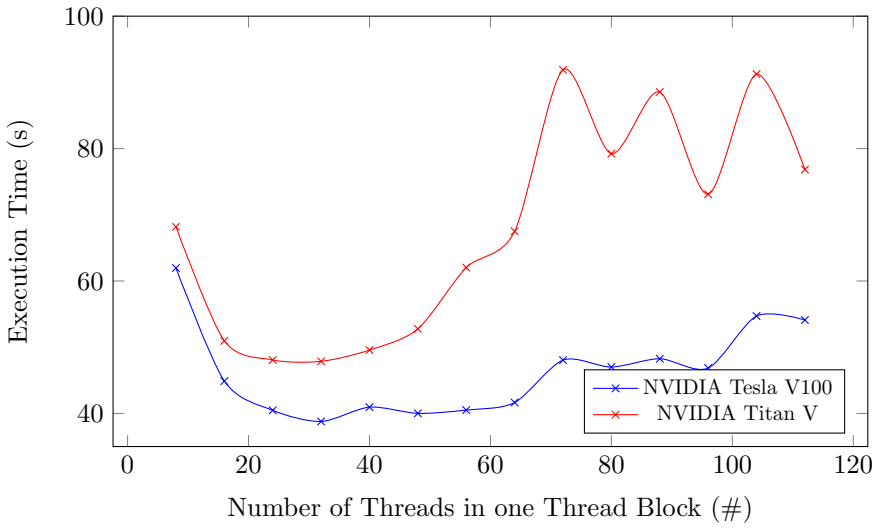

Figure 10: Block-Cyclic Execution Time against Size of Thread Blocks

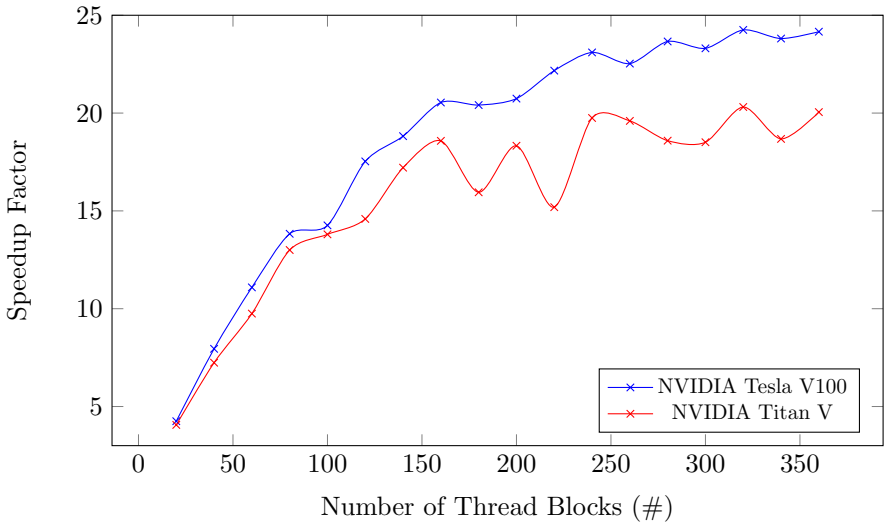

Figure 11: Block-Cyclic Speedup Factor against Thread Blocks

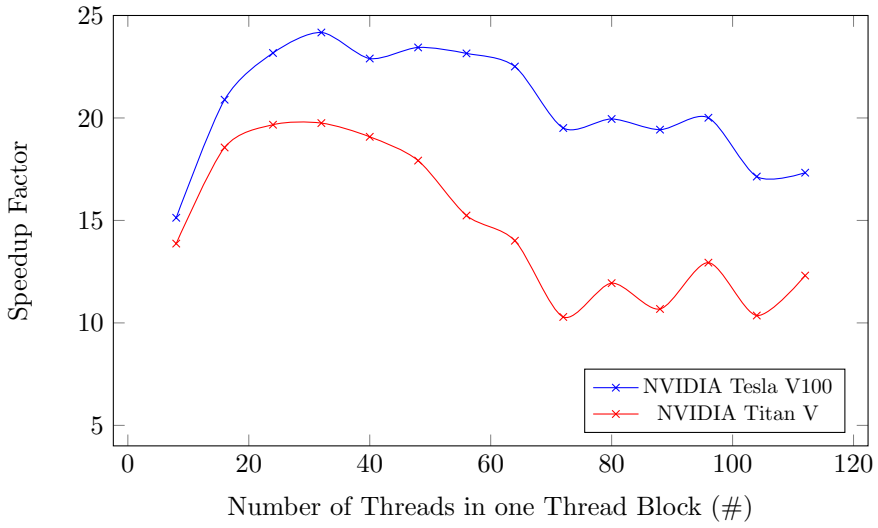

Figure 12: Block-Cyclic Speedup Factor against Size of Thread Blocks

Comparison of Block-Cyclic and Blockwise Work Distribution

For table 1, the maximum speedups were obtained when 360 blocks and 32 threads per block were used.

	Tesla V100	Titan V
Blockwise	21.64x	19.09x
Block-Cyclic	24.25x	20.31x

Table 1: Comparison of Maximum Speedups against OpenMP Recorded Timings

Discussion

Unless otherwise specified, discussions made are when the program input is a 3000x3000 size matrix running on 10,000 generations.

Warps and Parallelism

Reducing blocksizes and parallelism

As seen in figure 8 and 12, the peak speedup obtained happened when the number of threads in thread block becomes 32. This number coincides with the size of each warp. With 32 threads in each thread block, this implies that each thread will work on a 1 x 94 size subgrid (for the block-cyclic implementation), or approximately 376 bytes. This indicates that multiple memory transactions will have to be made. Furthermore, as you increase the total number of threads in the thread block, the size of each block shrinks. The reduction in size of thread blocks will hence require fewer memory transactions, which potentially will reduce the execution timing for the program.

Increasing Total Number of Warps

Despite the potential cost savings brought about by the reduction in memory transactions, the execution timing of the program increased. The observation can be attributed by the increase in total number of warps required for each thread block to execute. As a result, this incurs parallelism lost.

Granularity of the Program against Warps

This observation accentuates the fact that too fine grained tasks that minimise the cost of memory may increase the number of warps required to run the program. While it is good to reduce the cost of memory accesses on a GPU, it appears to be more important to minimise the total number of warps required to run the program.

Thread Blocks within a SM

Reducing number of resident blocks in a SM

It was earlier stated that the number of thread blocks residing in a SM was desired to be kept at a minimal. As calculated earlier, having the number of blocks exceed 64 would cause more than 1 thread block residing in one SM. At this recommended value, this means that each thread will work on a 47 x 94 size subgrid. The subgrid size implies a high degree of data sparsity, which will require multiple data transactions thereby slowing the program. This is clearly shown in figures 5 and 7, where having just 1 resident thread block with a large subgrid shows that the program does not reach its maximum speedup.

Improving Spatial Locality of Data

Reducing the row dimension generally improves the spatial locality of the data. It also aides in helping with coalescing global memory transactions. Having this reduced data size greatly reduces the program execution timing, and negates the parallelism losses incurred with having more than one resident block in a SM. In the test case with the provided input `sample7.in`, the program reaches peak speedup when there were 5-6 resident blocks in the SM. This illustrates the importance of spatial locality of data, and the need for reducing memory accesses in obtaining the optimal speedup.

Block-Cyclic Work Distribution Effectiveness

It is clear from table 1 that the block-cyclic work distribution technique is effective in improving the overall speedup for the program. The increase in speedups can be observed on both the Titan V and Tesla V100 GPUs. In the block-cyclic work distribution scheme, each block is constrained to improve spatial locality, and to reduce memory accesses by unifying memory transactions. Despite each thread having to go through more iterations as a result of reduced block size, the compute savings from reduced memory transactions reduced the computation time.

Areas of Improvement

Thread-pool Implementation of Block-Cyclic Work Distribution Scheme

As stated earlier, the block-cyclic work distribution allocates work on a static schedule. A dynamic scheduling of work, such as using a thread-pool, was not considered due to concerns with memory accesses. The downside of having static scheduling of tasks to threads could increase the likelihood of starvation of other threads. Some of the other threads could idle unnecessarily long which is a waste of computational resources. Perhaps such an implementation could be explored further to speedup the program.

Increase Use of Shared Memory

Shared memory is much faster than the global memory. Given the decomposition of the tasks into smaller block sizes, it might be possible to use shared memory to substantially speedup the program.

Reducing number of branches in `getNextState`

The number of branches in `getNextState` increases the overall thread branching. It reduces the overall ability for each thread to execute in lockstep, thereby slowing the program. `getNextState` and it's branches can be further reduced to speedup the program.