# CS3210 Assignment 2

Joshua Tan Yin Feng (A0199502Y), Kishen Ashok Kumar (A0204777X)

# Introduction

In this report, we outline our CUDA implementation for the Game of Invasion programme. This report is broken into these sections:



# Parallelisation with different data distributions

Across all implementations, the majority of our code is similar to that of the original sequential Game of Invasion programme since the parsing of data and game logic does not need to be changed. In this section, we will discuss our design considerations, code implementation, results and follow up discussions based on our results.

## Design considerations

When designing the program, we mainly wanted to change the way data is being distributed and computed in the GPU. The data distribution is mainly handled in the `getNextWorld` kernel code. But before we go into detail, we will first start off with a brief overview of the non-implementation specific design details so that the reader has a better view of how we implemented our programs.

### Non-implementation specific design considerations and improvements

#### GPU dimensions

From assignment 1, we knew that the world was indexed in a 1D fashion after it was parsed from the input files. For simplicity and ease of design, we collapsed the grid and block dimensions into a single dimension even though the sizes of each dimension are provided as the program parameters.

```
int numOfBlocks = GRID_X * GRID_Y * GRID_Z;
int numThreadsPerBlock = BLOCK_X * BLOCK_Y * BLOCK_Z;
dim3 gridDim(numOfBlocks);
dim3 blockDim(numThreadsPerBlock);
```

Figure 1. Collapsing of dimensions into a single dimension

With this change, mapping the threads or blocks to the respective cells in the world grid becomes very simple.

### Removing (cuda)malloc

In assignment 1, we were allocating memory for a new world with every generation. We realised in assignment 2 that this was a poor design choice. We could easily just swap the pointers for the current and new world states. With this, we can reuse the available memory since the world size remains constant anyways. Though not shown in the final set of data presented in this report, we were able to achieve a 100% speedup across all our implementations when running on sample7 after we made this change.

```
//Execute
getNextWorld<<<gridDim, blockDim>>>(d_world, d_newWorld, d_invasion, worldSize, N_COLS);
cudaDeviceSynchronize();
check_cuda_errors();
deathToll += genDeathToll;
genDeathToll = 0;

int *temp = d_world;
d_world = d_newWorld;
d_newWorld = temp;
```

Figure 2. Replacing cudaMalloc of newWorld and freeing of the current world with a simple swapping technique

## Updating of deathToll counter

We created a global variable known as `genDeathToll` to keep track of the number of deaths per generation. Every thread would perform an atomicAdd to this counter. When the next world state has been fully computed, this counter would be added to the overall deathToll counter.

```
if (diedDueToFighting)
{
    atomicAdd(&genDeathToll, 1);
}
```

Figure 3. Atomic add of `genDeathToll`

```
//Execute
getNextWorld<<<gridDim, blockDim>>>(d_world, d_newWorld, d_invasion, N_ROWS, N_COLS);
cudaDeviceSynchronize();
deathToll += genDeathToll;
genDeathToll = 0;
```

Figure 4. Adding `genDeathToll` to the overall `deathToll`

## Design 1: Block-wise data distribution

The first question that we needed to answer was how we were going to distribute the data. We decided to start with a block-wise distribution. Each block would take an (almost[1]) equal number of rows and would compute the next world state for its assigned rows. Suppose a block takes $row_i$ to $row_{i+j}$, then it will require knowledge of $row_{i-1}$ to $row_{i+j+1}$. To keep our baseline implementation simple, we decided to load the current world into global memory for all blocks to be able to access. This would allow us to simply focus on the allocation of rows to the blocks. Threads in the block would be reused if the number of cells allocated to the block is greater than the total number of threads available.

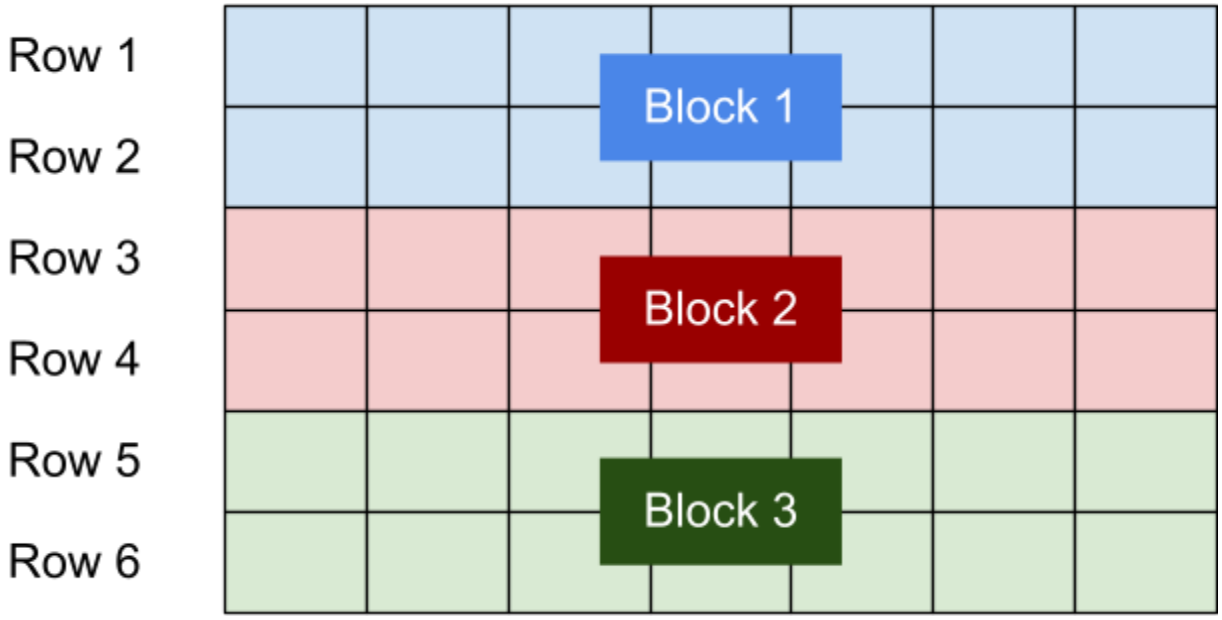


Figure 5. Block wise distribution

## Design 2: Block cyclic data distribution

The next alternative would be a block cyclic data distribution. Each GPU block would be assigned a fixed number of cells to work in a cyclical manner. The view of block cyclic data distribution at the GPU block level is equivalent to the view of cyclic data distribution at the GPU

[1] Blocks do not take the equal number of rows if the number of rows are not a multiple of the total number of blocks

thread level. Taking on the latter view makes it easier for us to calculate the indexing of each thread. If each thread takes the cell $Cell_{i, j}$, then the thread would require knowledge of the neighbouring 8 cells surrounding it. Once again, we store the current world state in global memory so that every thread will have access to its neighbouring cells' data.

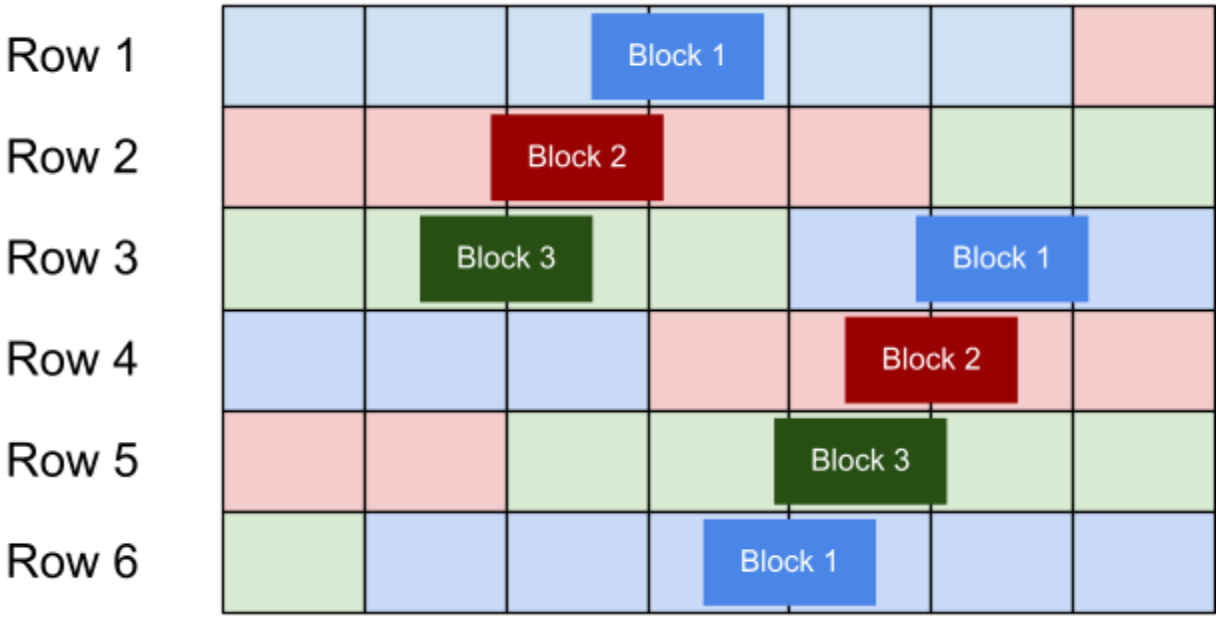


Figure 6. Block cyclic distribution when viewed at a GPU block level

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Row 1 | B1T1 | B1T2 | B1T3 | B1T4 | B1T5 | B1T6 | B2T1 |
| Row 2 | B2T2 | B2T3 | B2T4 | B2T5 | B2T6 | B3T1 | B3T2 |
| Row 3 | B3T3 | B3T4 | B3T5 | B3T6 | B1T1 | B1T2 | B1T3 |
| Row 4 | B1T4 | B1T5 | B1T6 | B2T1 | B2T2 | B2T3 | B2T4 |
| Row 5 | B2T5 | B2T6 | B3T1 | B3T2 | B3T3 | B3T4 | B3T5 |
| Row 6 | B3T6 | B1T1 | B1T2 | B1T3 | B1T4 | B1T5 | B1T6 |

Figure 7. Cyclic distribution when viewed at a GPU thread level

## Design 3: Shared memory data distribution

Our implementations so far have relied on global memory. However, access to global memory is slow and we hypothesise that if we used a shared memory implementation, our programme would be able to run faster. At this juncture, we decided to use block-cyclic but with each block taking up one full row instead of only a partial part of the row like in design 2. If each block took one full row, we would only need to include $row_i$ and $row_{i+1}$ into our shared memory per block. If we followed design 2 and each block only took up a partial part of the row, we would need to include the row above and below as well as the side columns into our shared memory per block. This poses 2 problems compared to the former: firstly, each block on average would need to transfer an additional 6 cells (the left and right columns) from the global memory to the shared memory; secondly, indexing would be complicated since each block starts and ends at a different location and may not have valid left or right columns if they are aligned at the edges of the grid.

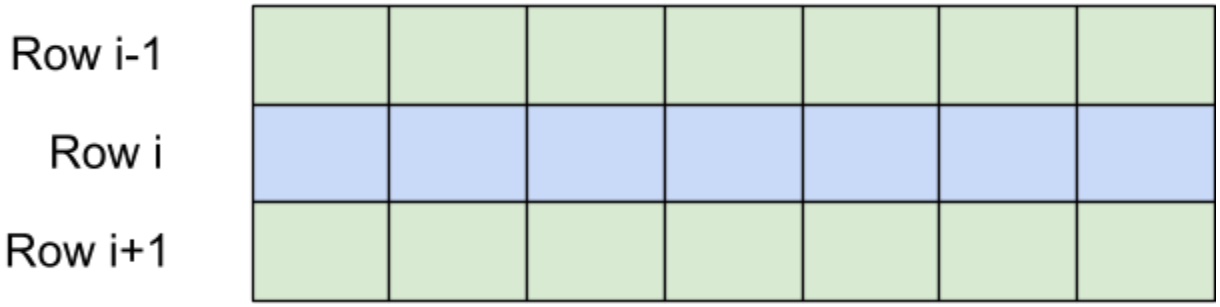


Figure 8. Shared memory of a block operating on $row_i$. Green area represents the extra data needed to compute the next state

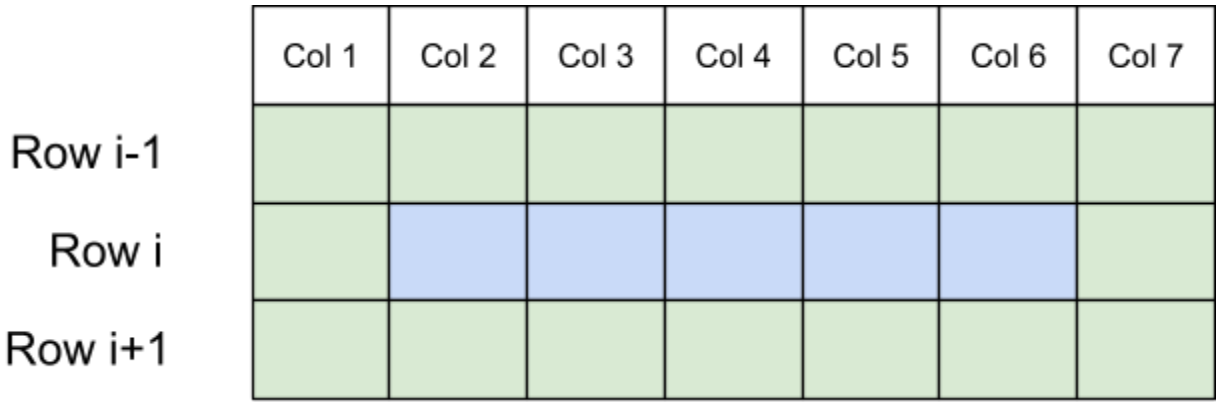


Figure 9. Shared memory of a block operating on $row_i$ from columns 2 to 6. Green area represents the extra data needed to compute the next state. Columns 1 and 7 are the extra 6 cells that need to be loaded into shared memory.

## Design 4: Removal of branches

When using the GPU, we knew that we should avoid as much branching as possible in order for the threads in a warp to proceed in a lock-step manner. This would provide for maximal

parallelisation within a block. If we were to remove the branches, then we would only have one `return` statement at the end and it must return the correct value. With a bit of brainstorming, we came up with a nifty solution.

Suppose there were originally 2 branches and each returned a value a and b respectively. If we were to remove the branching, we would have both values and we needed a way to choose between the two values. If we had a conditional variable set to either 0 or 1 depending on whether the first branch was supposed to be executed, then the return statement would simply be

```
return conditional * a + !conditional * b;
```

This solution works as long as a and b are both integer values. Since our kernel function returns integers, it is an ideal candidate for our nifty programming paradigm.

### Additional design considerations that we did not include

One potential optimization would be loop unrolling. Throughout our code, we utilised small loops. We could have removed the loops and hard coded each iteration in. This would remove any potential stalls from missed branch predictions and allow the threads in a warp to proceed in a lock-step manner. This is further justified by the small size of the loop which would not increase the memory space needed to store the code instructions by too much. However, we felt that this was a trivial optimization and so we did not include it.

```
for (int faction = DEAD_FACTION + 1; faction < MAX_FACTIONS; faction++)
{
    if (faction == cellFaction)
    {
        continue;
    }
    hostileCount += neighborCounts[faction];
}
```

Figure 10. Example of a loop that could be unrolled

## Implementation details

In this section, we will briefly describe the code we have written for each implementation.

### Block-wise implementation

As illustrated in figure 5, each block would take a set number of rows. We used some simple arithmetic to calculate the start and end rows index assigned to each block.

```
__global__ void getNextWorld(int* d_world,int* d_newWorld,int* d_invasion, int N_ROWS, int N_COLS) {
    int blockID = blockIdx.x;

    int rowsPerBlock = N_ROWS / gridDim.x;
    int leftoverRows = N_ROWS % gridDim.x;

    int startRow = blockID * rowsPerBlock + (blockID < leftoverRows ? blockID * 1 : leftoverRows);
    int endRow = startRow + rowsPerBlock + (blockID < leftoverRows ? 1 : 0);
    int row = startRow;
    while (row < endRow && startRow < N_ROWS) {
        int threadID = threadIdx.x; // use threadID to represent column
        while (threadID < N_COLS) {
            bool diedDueToFighting;
            int nextState = getNextState(d_world, d_invasion, N_ROWS, N_COLS, row, threadID, &diedDueToFighting);
            setValueAtD(d_newWorld, N_ROWS, N_COLS, row, threadID, nextState);
            if (diedDueToFighting)
            {
                atomicAdd(&genDeathToll, 1);
            }
            threadID += blockDim.x;
        }
        row++;
    }
}
```

Figure 11. Code implementation of block-wise distribution

## Block cyclic implementation

As mentioned previously, the block cyclic view of data when examined at the GPU block level is equivalent to the cyclic view of data when examined at the GPU thread level.
The position of the cell based on the thread's thread id and block id is stored in a variable called globalPos.

```
__global__ void getNextWorld(int* d_world,int* d_newWorld,int* d_invasion, int worldSize, int N_COLS) {
    int globalPos = blockIdx.x * blockDim.x + threadIdx.x;

    while (globalPos < worldSize) {
        int diedDueToFighting = 0;
        int nextState = getNextState(d_world, d_invasion, worldSize, globalPos, N_COLS, &diedDueToFighting);
        *(d_newWorld + globalPos) = nextState;
        atomicAdd(&genDeathToll, diedDueToFighting);
        globalPos += gridDim.x * blockDim.x;
    }
}
```

Figure 12. Code implementation of block-cyclic distribution

The only issue with this implementation is that our world is represented in a single dimension, which means that the edges of the grid are not represented. To solve this, we stored the indices of neighbours in an array called positions. If the current cell is at the boundary, then it should have some invalid neighbours; we use the value -1 to represent invalid indices.

```
int positions[TOTALPOS];
for (int i = 0; i < TOTALPOS; i++) {
    // let -1 indicate an invalid index
    positions[i] = -1;
}
// if not at left column, then it should have valid left neighbours
if (globalPos % N_COLS != 0) {
    positions[TOPLEFT] = globalPos - N_COLS - 1;
    positions[LEFT] = globalPos - 1;
    positions[BOTLEFT] = globalPos + N_COLS - 1;
}

// no need to concern ourselves with the top and bottom edge cases because they will be
// checked in getValueAt
positions[TOP] = globalPos - N_COLS;
positions[CENTER] = globalPos;
positions[BOT] = globalPos + N_COLS;

// if not at right column, then it should have valid right neighbours
if ((globalPos + 1) % N_COLS != 0) {
    positions[TOPRIGHT] = globalPos - N_COLS + 1;
    positions[RIGHT] = globalPos + 1;
    positions[BOTRIGHT] = globalPos + N_COLS + 1;
}
```

Figure 13. Storing the indices of neighbours in an array called positions

## Shared Memory

To initialise the shared memory, we create a 1D array called sharedRows. Each thread within the block would help to bring in data from the global memory into shared memory. Once that is done, we can start computing the next cell states similar to before. The only thing to take note is that we needed to synchronise the threads before and after computations so that there wouldn't be any race conditions.

```
__global__ void getNextWorld(int* d_world, int* d_newWorld, int* d_invasion, int sharedRowsSize, int worldSize, int N_ROWS, int N_COLS)
{
    extern __shared__ int sharedRows[];
    int row;
    int col;
    int globalPos;

    // let every block take one whole row
    row = blockIdx.x;

    while (row < N_ROWS) {
        col = threadIdx.x;

        // initialise particular top, curr and bottom rows
        while (col < N_COLS) {
            globalPos = row * N_COLS + col;
            // top row
            // (if curr row is already the top most row, then we set the value in shared rows to be -1 to indicate invalid)
            sharedRows[col] = globalPos - N_COLS < 0 ? -1 : d_world[globalPos - N_COLS];
            // curr row
            sharedRows[N_COLS + col] = d_world[globalPos];
            // bot row
            // (if curr row is already the bottom most row, then we set the value in shared rows to be -1 to indicate invalid)
            sharedRows[N_COLS * 2 + col] = globalPos + N_COLS >= worldSize ? -1 : d_world[globalPos + N_COLS];
            col += blockDim.x;
        }

        // make sure that sharedRows has been fully initialised before we move on
        __syncthreads();

        col = threadIdx.x;
        while (col < N_COLS) {
            globalPos = row * N_COLS + col;
            int diedDueToFighting = 0;
            int nextState = getNextState(sharedRows, d_invasion, sharedRowsSize, worldSize, globalPos, col, N_COLS, &diedDueToFighting);
            *(d_newWorld + globalPos) = nextState;
            atomicAdd(&genDeathToll, diedDueToFighting);
            col += blockDim.x;
        }

        // need to ensure that the current row has finished before moving on
        __syncthreads();

        row += gridDim.x;
    }
}
```

Figure 14. The highlighted portion is the code for bringing in data from global memory into shared memory

## No branching shared memory

As mentioned previously, we used simple arithmetic to represent boolean values and branch conditionals. Here is an example of how we removed the `if else` branches from calculating overpopulation and fighting with neighbours.

```
int hostileCount = 0;
for (int faction = DEAD_FACTION + 1; faction < MAX_FACTIONS; faction++)
{
    hostileCount += (faction != cellFaction) * neighborCounts[faction];
}

*diedDueToFighting = !originallyDead & ((*diedDueToFighting) | willFight(hostileCount));
isDead = !originallyDead & willFight(hostileCount);

int friendlyCount = neighborCounts[cellFaction];
// if was originally dead, then it can't die again
isDead = !originallyDead & (isDead | !isSurvivable(friendlyCount));

/*
No Invasion:
if no invasion and not dead from overpop/fighting,
then return original cell faction (assuming greater than 0)
or new faction (assuming orignal was dead). Note that new faction defaults to dead as well

Invasion:
return the new invader faction
*/
return !invasion * !isDead * (!originallyDead * cellFaction + originallyDead * newFaction)
       +
       invasion * invaderFaction;
```

Figure 15. Replacing `if else` branches with integer arithmetic to decide outcomes

# Methodology

Before we discuss the results of our various implementations, we will outline our data collection methods.

## Tools used

We used the perf linux utility to help us measure the execution runtime of our programs. Similar to assignment 1, we repeated the execution of our programs 5 times and got the average runtime.
We further used NVIDIA's CUDA profiler nvprof to gather various metrics about our kernel code. We only included metrics that would be relevant to our points of discussion.
The breakdown of the metrics can be found in Appendix A.

## Commands used to run the programs

### Compiling

To compile the block-wise implementation, we used `nvcc -o goi_cuda goi_cuda.cu`.

For the other implementations, we used `nvcc -o goi_cuda_<implementation> goi_cuda_<implementation>.cu`

Note that the makefile only contains the make recipe for the block implementation.

### Executing the code

To execute the program, we simply used `./goi_cuda <INPUT FILE> <OUTPUT FILE> <GRID X DIM> <GRID Y DIM> <GRID Z DIM> <BLOCK X DIM> <BLOCK Y DIM> <BLOCK Z DIM>`

## Analysis of the performance of our different implementations

In this section we will examine the impact that varying the grid and block size have on the performance of our different implementations for a fixed input size. In order to do this, we repeated the entire experiment twice on two different nodes, xgpc and xgpd.

Each experiment comprises execution of our block-wise, block cyclic, shared memory and no branch shared memory implementations with varying grid and block sizes on a particular sample.

For each sample, the 5 configurations we ran the experiment on are:

1. 80x1024
2. 160x1024
3. 320x1024
4. 512x1024
5. 1024x1024

The above 5 configurations were run on sample 1, 4, 5 and 7 with 10,000 iterations.

We will discuss the results of the executions at the end of the section. The full raw results can be viewed in Appendix B.

## Execution on Sample 1, varying grid and block size

### Results

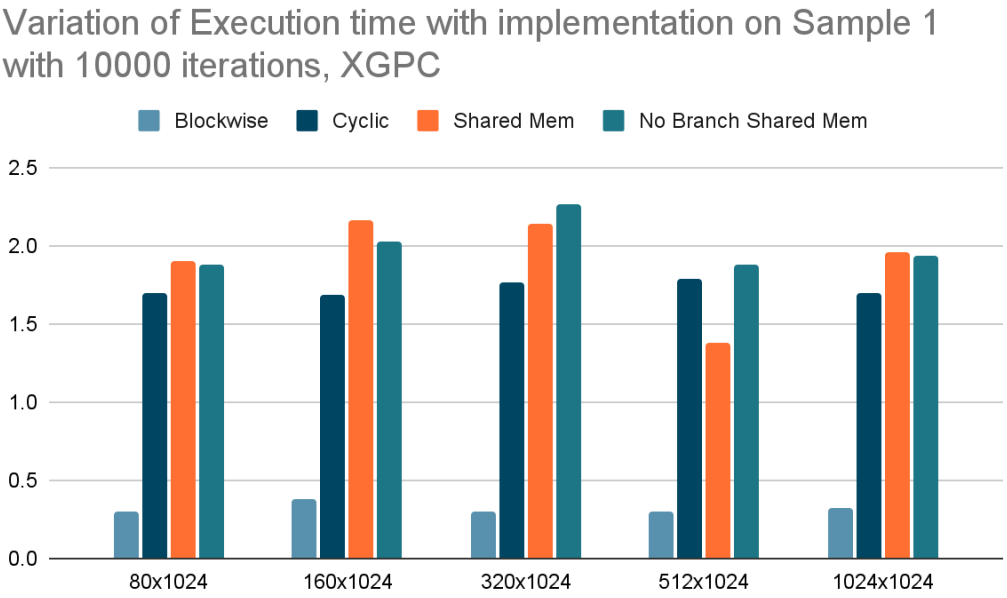


Figure 16. Variation of Execution time on Sample 1 on XGPC

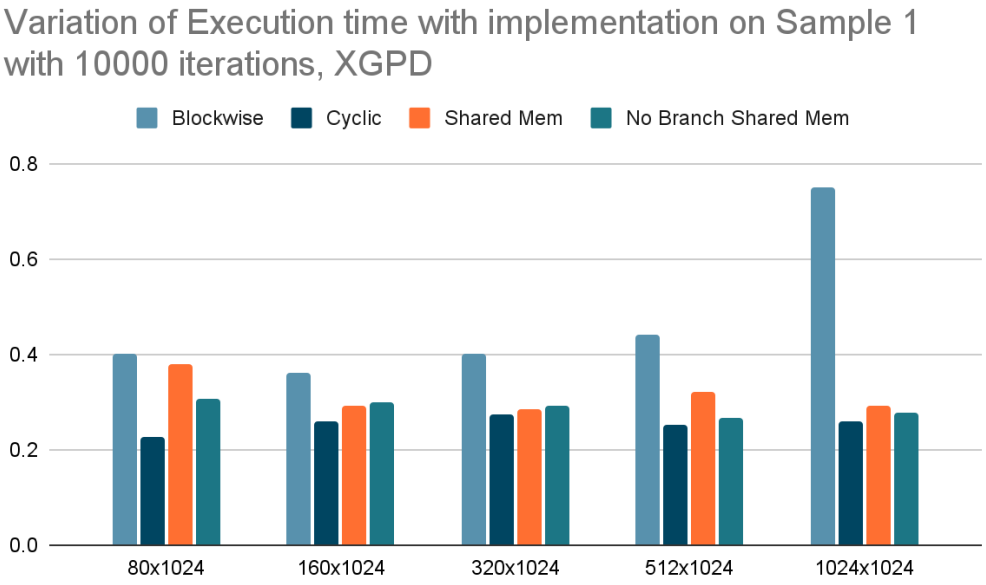


Figure 17. Variation of Execution time on Sample 1 on XGPD

## Execution on Sample 4, varying grid and block size

### Results

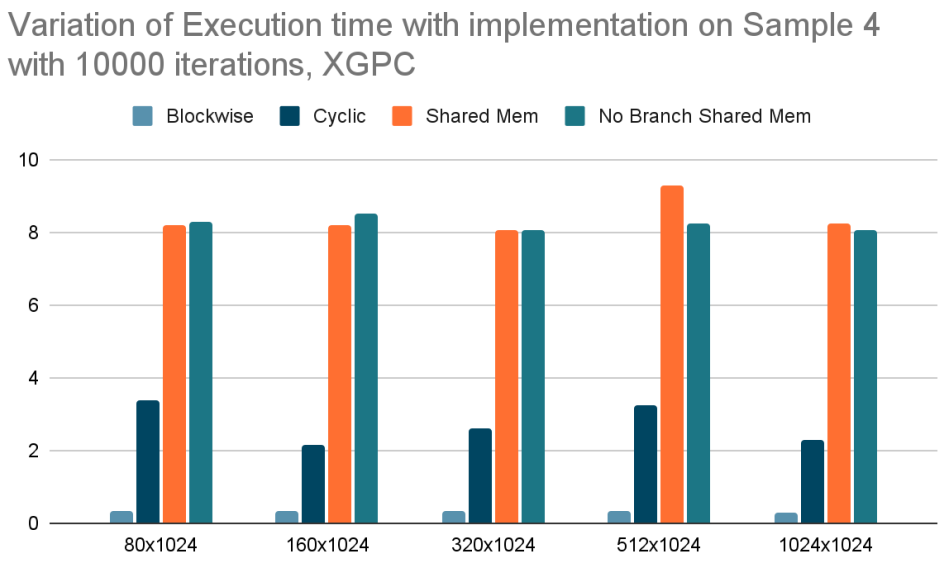


Figure 18. Variation of Execution time on Sample 4 on XGPC

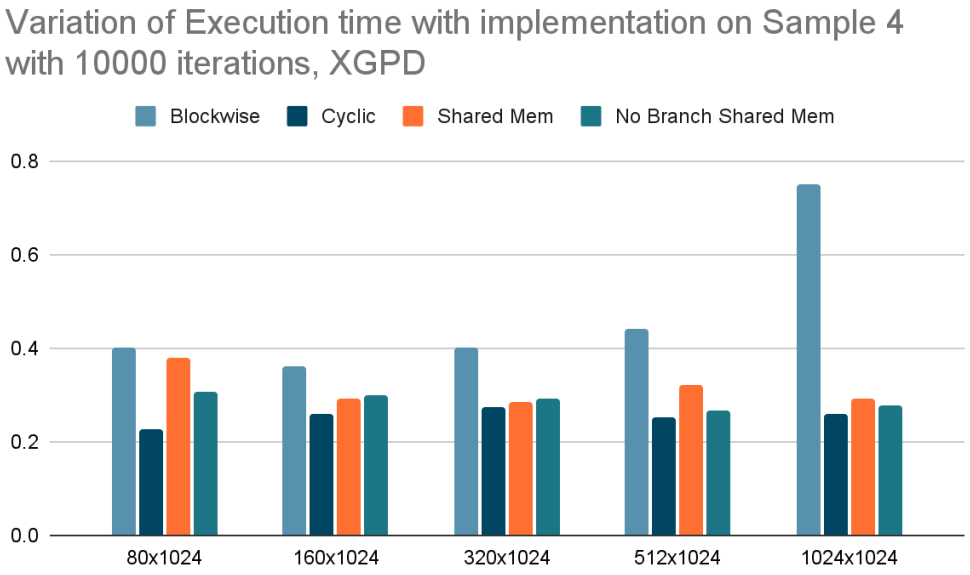


Figure 19. Variation of Execution time on Sample 4 on XGPD

## Execution on Sample 5, varying grid and block size

### Results

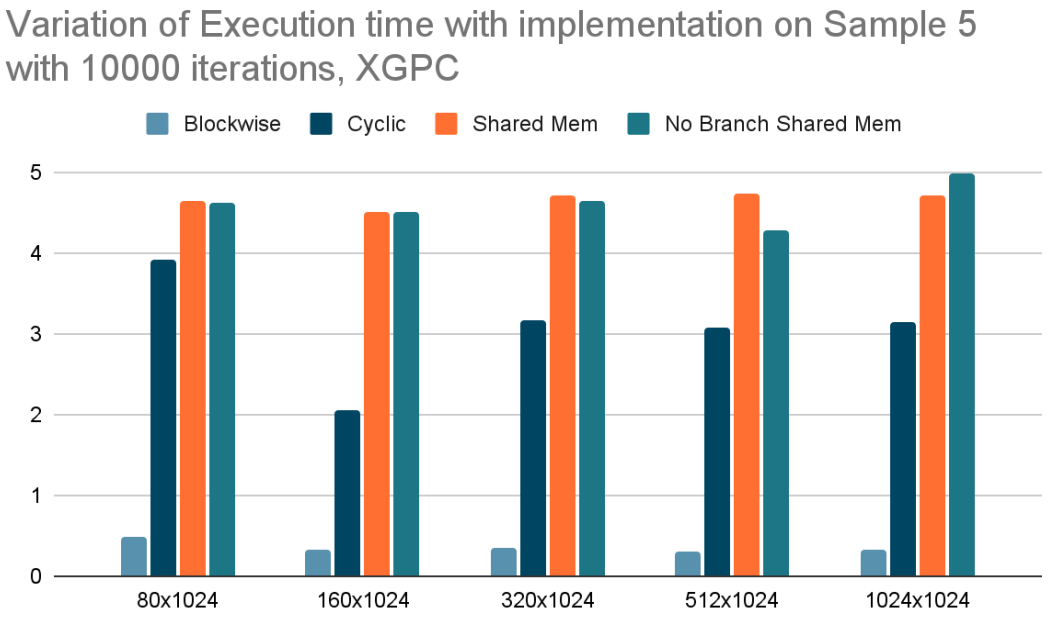


Figure 20. Variation of Execution time on Sample 5 on XGPC

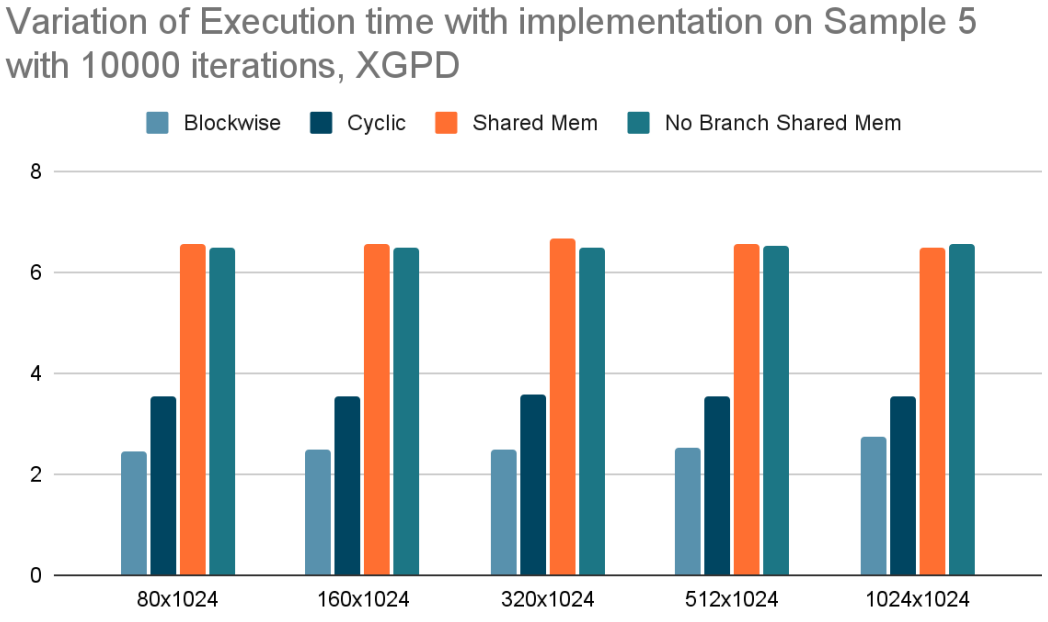


Figure 21. Variation of Execution time on Sample 5 on XGPD

## Execution on Sample 7, varying grid and block size

### Results

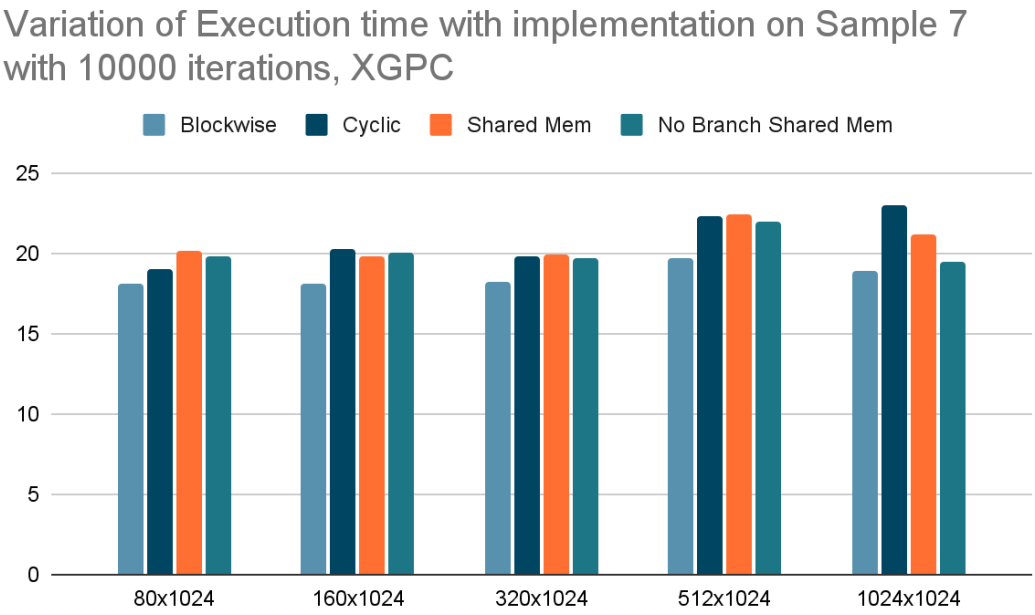


Figure 22. Variation of Execution time on Sample 7 on XGPC

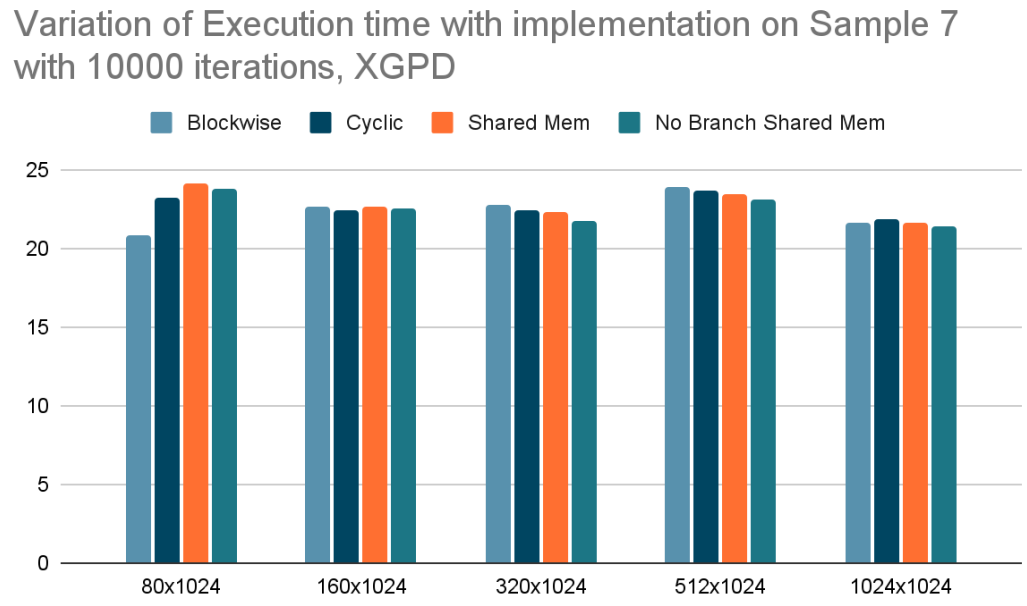


Figure 23. Variation of Execution time on Sample 7 on XGPD

## Discussion of Results

Throughout the different executions of our implementations with varying grid and block size on different world sizes, a common trend can be seen from the results. We see that regardless of the implementation, for a particular world size, varying the grid and block size does little to affect the performance of the implementation. Such a result is counterintuitive to our expectations as we would expect to see a clear improvement in performance as we increase the grid and block size.

However, from the results we see a repeated trend of the performances between the different implementations. Typically, we see that the block wise implementation performs the best, followed by the cyclic implementation and lastly the shared memory and no branch shared memory implementations who have similar performances. To aid in trying to understand these results, we can refer to the following graph depicting the metrics we gathered from using `nvprof`. The full results can be found in Appendix C.

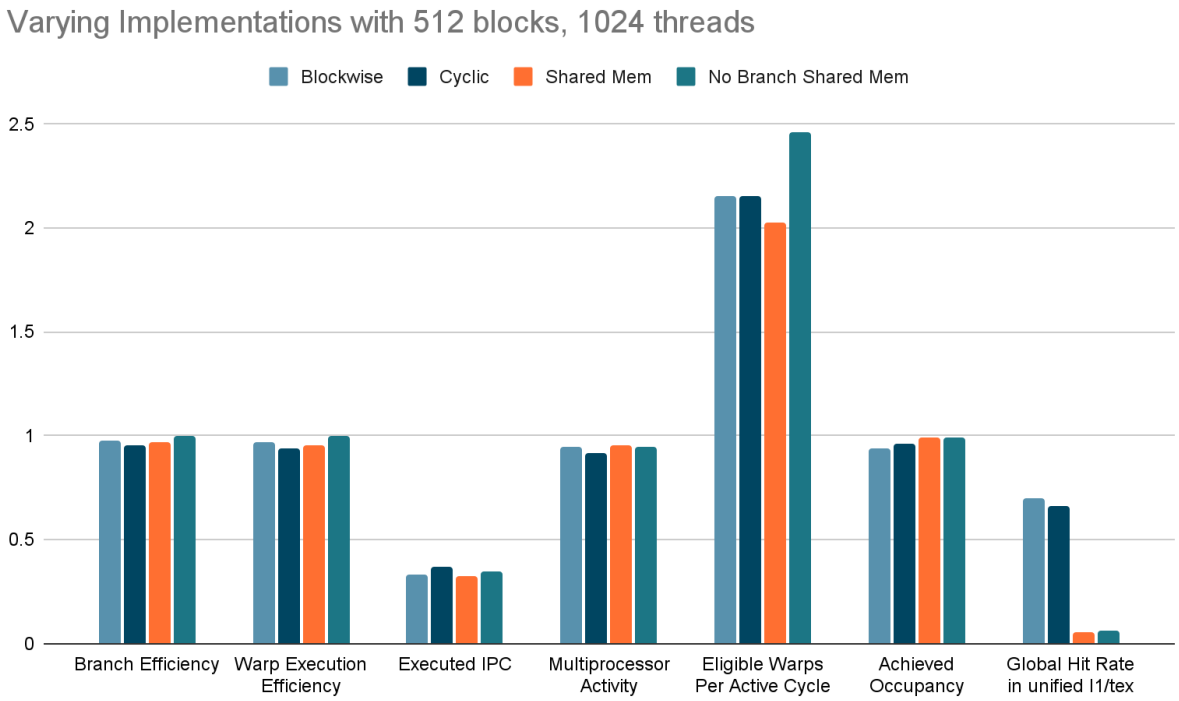


Figure 24. Key Performance Indicators across different implementations

From the above graph, we see that regardless of the implementation, the performance indicators remain largely the same. The only stark difference is that of global hit rate in unified l1/tex. This result is expected because our shared memory and no branch shared memory implementations were designed to avoid using global memory as much as possible and hence have a lower hit rate due to missing on the few times it accesses global memory. While counterintuitive to our understanding of shared memory being faster to access than global memory, our data seems to show an inverse correlation between a higher global hit rate and a

better performance - resulting in the block wise and cyclic implementations having a better performance than our shared memory and no branch implementations. We hypothesise that the size of the task is still too small for the speedup incurred due to using shared memory to outweigh the overhead in setting up and transferring data to and fro shared memory and hence we get a decrease in performance.

# Comparison with OpenMP

Running sample7.in on our OpenMP implementation from assignment 1, takes 1774.077s.

Comparing the different implementations with a 1024x1024 grid and block size, we get the following speedup information.

| | **Time Taken** | **Speedup** |
|---|---|---|
| **Block-wise** | 18.878 | 93.97x |
| **Block cyclic** | 22.96 | 77.26x |
| **Shared Memory** | 21.171 | 83.80x |
| **No Branch, Shared Memory** | 19.521 | 90.88x |

From this we see that we are able to achieve an incredible speed up as compared to OpenMP with the use of CUDA programming.

# Varying grid and block sizes

Having experimented with our different data distribution implementations, we wanted to also investigate the effects of varying the grid and block size and their impacts on performance. We conducted 3 experiments to observe the effects of the various factors:

1. Fixed block size (1024 threads), varying grid size
2. Fixed grid size (160 blocks), varying block size
3. Fixed total number of threads (512 threads), varying grid and block size

The full raw data can be found in Appendix D.

## Fixed block size (1024 threads), varying grid size

### Results

Our results show that with decreasing number of blocks, the total number of instructions executed per warp and executed IPC increases. Multiprocessor activity and achieved

occupancy also decreased starting with 80 blocks. Warp execution efficiency remained consistent throughout. Most interestingly, the number of eligible warps per cycle stayed consistent before it dipped at 80 blocks; it then rose back up at 40 blocks.

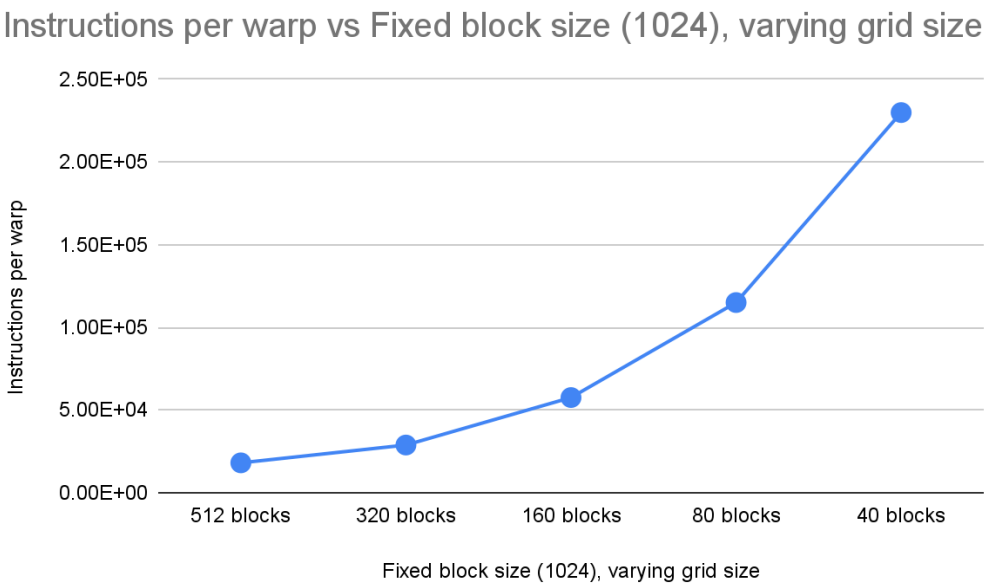


Figure 25. Instructions per warp vs Fixed block size

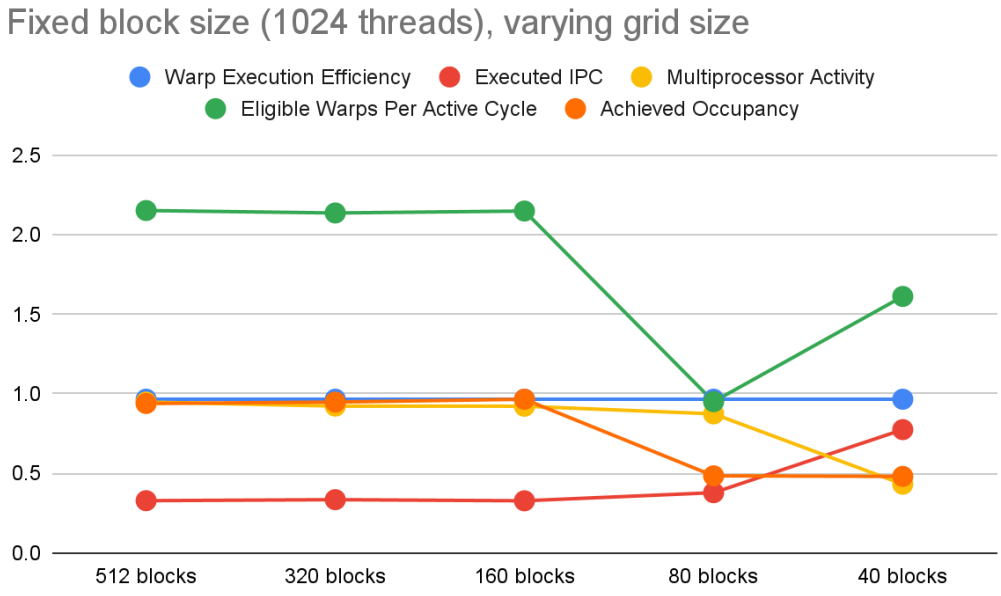


Figure 26. Fixed block size, varying grid size

### Discussion

The results were quite expected. With lesser blocks, it is expected that the number instructions executed per warp increased since each block is assigned more rows to compute.
The trend follows an exponential pattern which is what we expected.
The decrease in occupancy as well as multiprocessor activity was also expected since there were less blocks to occupy the SMs. Our GPU only has 80 SMs and so it is expected to see these values start decreasing at 80 blocks.
Warp execution efficiency being almost 1.0 is expected as well since we are using 1024 threads, meaning that there are plenty of warps in one block to continue execution in case one warp stalls.

The outstanding result was the rise in eligible warps per active cycle at 40 blocks. Going from 160 blocks (and more) to 80 blocks, the number of eligible warps dropped by half from around 2.2 to around 1.0. When there are more than 80 blocks, there should be an average of at least 2 blocks per SM. If one block stalls, the other block is eligible to proceed. When there are only 80 blocks, we suspect that the GPU distributed them equally amongst the SMs in order to balance the loads, hence the eligible warps dropped to around 1.0. At 40 blocks, we suspect that the GPU might have loaded some SMs with 2 blocks so as to achieve maximal utility if one of the blocks stalls. Without further evidence, we are unable to draw definitive conclusions.

## Fixed grid size (160 blocks), varying block size

### Results

As the number of threads decreases, the total number of instructions per warp increases in an exponential manner. The number of executed IPC follows a slight parabolic arc. Multiprocessor activity and warp execution efficiency stayed consistent throughout. All other results showed a decreasing trend as the number of threads decreases.

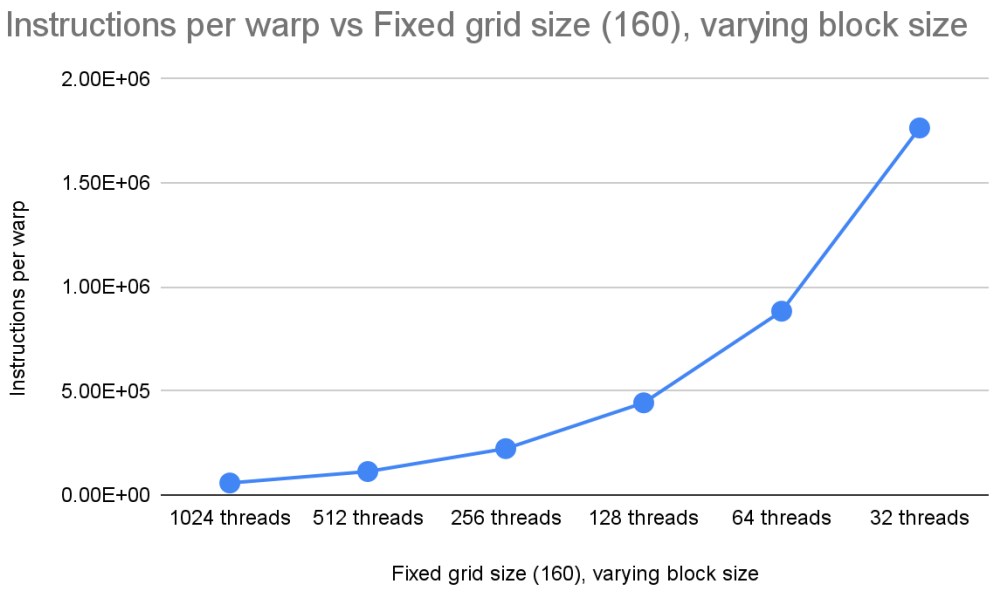

Figure 27. Instructions per warp vs Fixed grid size

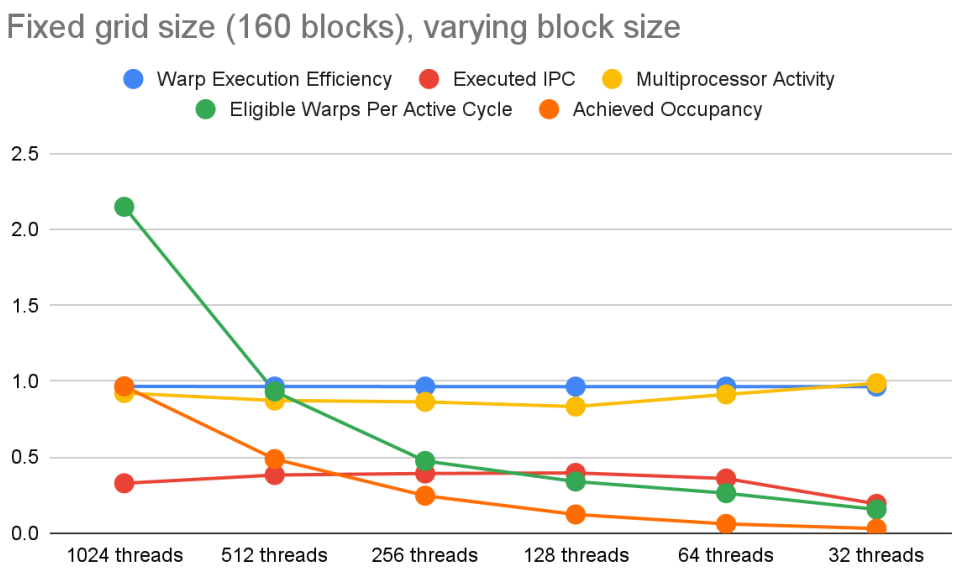


Figure 28.Fixed grid size, varying block size

## Discussion

Once again, it is expected that the number of instructions per warp increased with decreasing number of threads as each thread now needs to take more columns in the grid. Efficiency also stayed consistent at 1.0 but we expect that it would have dropped if we conducted the tests with less than 32 threads. This was later confirmed in our next set of tests.
Unlike the previous set of tests on varying grid sizes, multiprocessor activity remains consistent. This is because we fixed our grid size to be 160 blocks and hence at any one point in time, there should be an average of 2 blocks per SM.
The achieved occupancy trend was also expected. Each SM is only able to support up to a maximum of 32 warps (1024 threads). Hence as the number of threads decreases, the ratio of average active warps per active cycle to the maximum number of warps supported on a multiprocessor decreases proportionately as well. A similar argument holds for the discussion on the trend for eligible warps per active cycle as well.
Overall, the results we gathered here were not surprising and were consistent with what we expected.

# Fixed total number of threads (512), varying grid and block sizes

## Results

For this particular set of tests, we decided to also measure the runtime of our program to see the effects.
Our results show a decreasing trend in our execution time. Warp execution efficiency stayed consistent before decreasing from size 32x16 onwards. Executed IPC, eligible warps per active

cycle and achieved occupancy follows a parabolic pattern, with their lowest points hovering around sizes 16x32 and 32x16. Multiprocessor activity increased exponentially and peaked at around 128x4 and 256x2 before decreasing again.

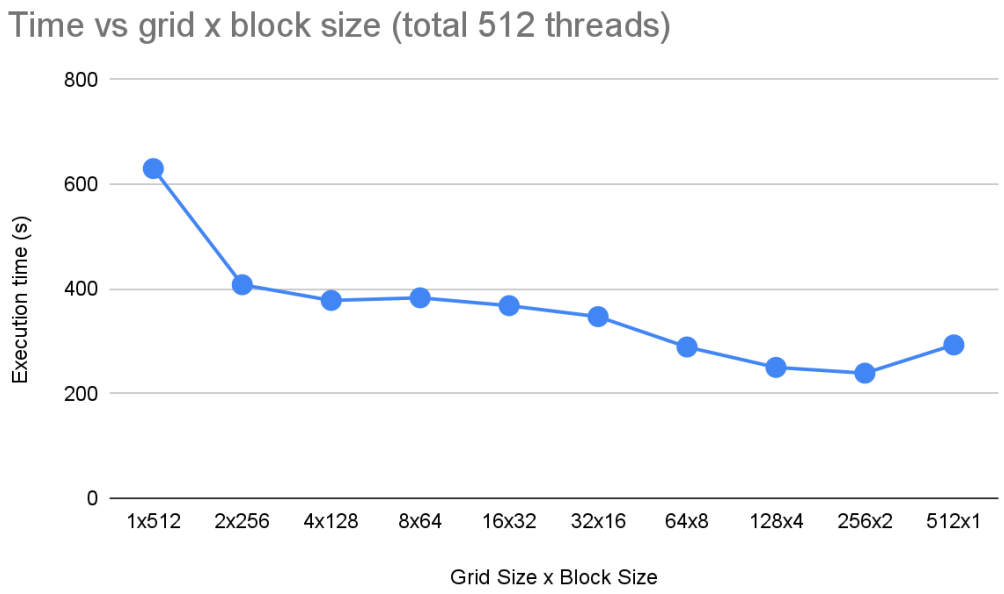


Figure 29. Time vs Grid x Block size

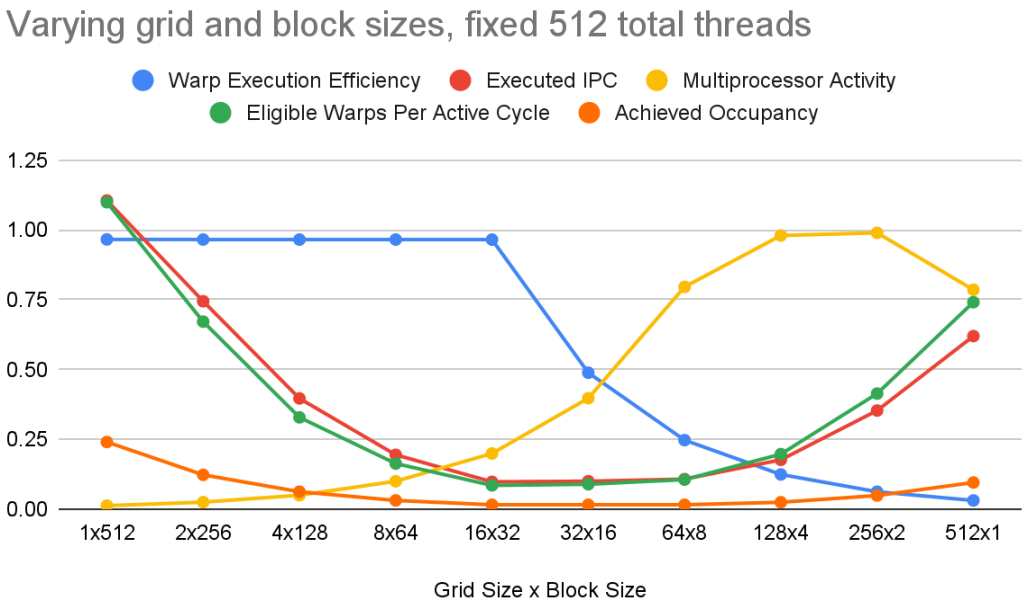


Figure 30. Varying grid and block sizes, fixed total threads

## Discussion

The trend in warp execution efficiency was expected and confirms our hypothesis proposed in the previous set of tests.

The parabolic trend for both eligible warps per active cycle and executed IPC is an interesting one. From the previous set of tests on varying block size, we expected the number of eligible warps per active cycle to decrease as the block size decreases. However, we did not expect it to increase again as the grid size increased. We suspect that the GPU loaded each SM with multiple blocks, similar to our argument for our first set of tests on varying grid size. But we are unable to conclude that this argument is valid since we do not know if the eligible warps per active cycle metric includes warps from different blocks on the same SM. We propose the same argument for the parabolic trend of executed IPC but are unable to draw any definitive conclusions.

The increase in multiprocessor activity was expected. However, we are unable to justify why it dropped at 512 grid size. Our first set of tests on varying grid size did not produce a similar drop. In fact, if there was a drop in multiprocessor activity, we would have expected to see a similar drop in achieved occupancy but this was not shown in our test results.

Most interestingly was the runtime trend. We had expected the runtime to be increasing. This was because as the block size decreased, each warp was not maximally utilised as shown by the decrease in warp execution efficiency. Global load and store efficiencies were decreasing. Given these factors, we were surprised that the execution time was made shorter. We propose a few possible explanations:

1. With fewer blocks, we are not fully maximising the parallelisation across the SMs. Since blocks and warps within an SM run concurrently (switching only when another block stalls), they do not perform as well as when there are multiple blocks spread across the SMs. Blocks across SMs are able to run in parallel with each other.
2. A decreasing number of global load and store transactions per request makes each request execute faster. With faster request turnaround times, the overall program ran faster.

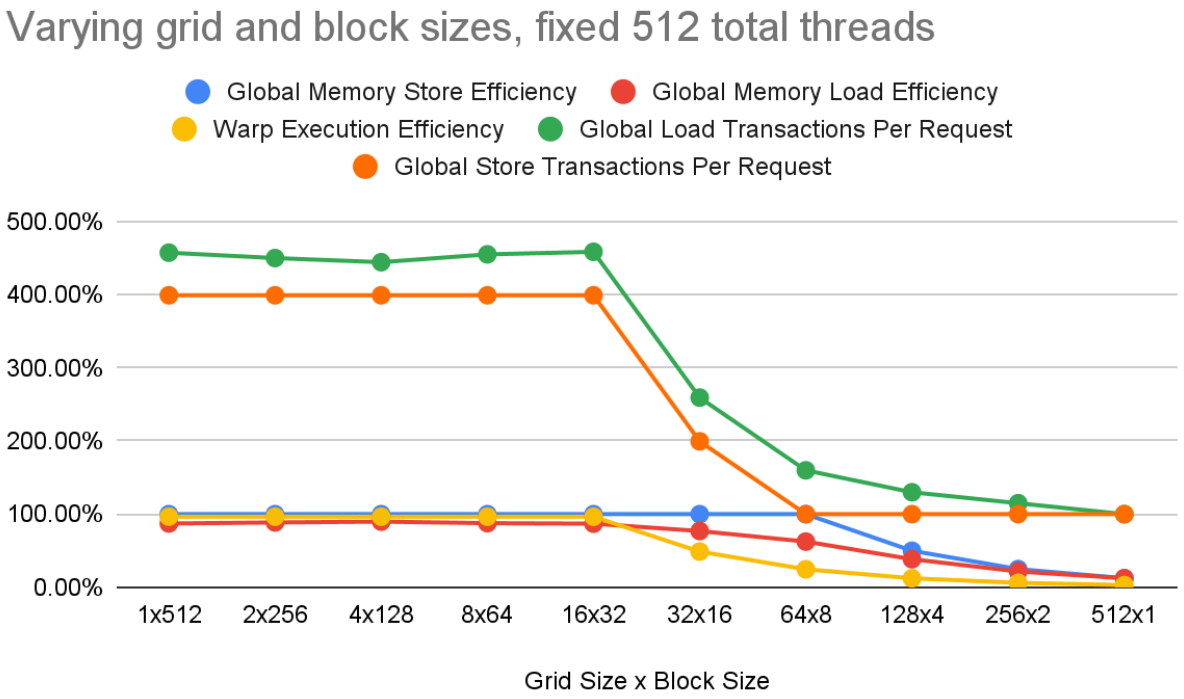


Figure 31. Varying grid and block sizes, fixed total threads

# Conclusion

Through this assignment, we have explored the different ways one task can be parallelised as well as the methods of implementation and the tradeoffs associated with each implementation. Additionally, we have explored their execution behaviour and how varying the multiple execution dimensions available to us as programmers affects the performance of our parallel implementations. Such knowledge is important, as having an arsenal of techniques will prove useful when encountering parallelizable problems in the future. Additionally, through our comparison with OpenMP, we have shown the shortcomings of such implementations and how we need both hardware and software support for effective parallelisation.

# Appendix A

## Metrics used for nvprof

| Metric Name | Metric Description |
|---|---|
| inst_per_warp | **Instructions per warp**: Average number of instructions executed by each warp |
| branch_efficiency | **Branch Efficiency**: Ratio of non-divergent branches to total branches |
| warp_execution_efficiency | **Warp Execution Efficiency**: Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor |
| shared_load_transactions_per_request | **Shared Memory Load Transactions Per Request**: Average number of shared memory load transactions performed for each shared memory load |
| shared_store_transactions_per_request | **Shared Memory Store Transactions Per Request**: Average number of shared memory store transactions performed for each shared memory store |
| gld_transactions_per_request | **Global Load Transactions Per Request**: Average number of global memory load transactions performed for each global memory load |
| gst_transactions_per_request | **Global Store Transactions Per Request**: Average number of global memory store transactions performed for each global memory store |
| ipc | **Executed IPC**: Instructions executed per cycle |
| sm_efficiency | **Multiprocessor Activity**: The percentage of time at least one warp is active on a specific multiprocessor |
| eligible_warps_per_cycle | **Eligible Warps Per Active Cycle**: Average number of warps that are eligible to issue per active cycle |
| achieved_occupancy | **Achieved Occupancy**: Ratio of the average |

| | |
|---|---|
| | active warps per active cycle to the maximum number of warps supported on a multiprocessor |
| gld_transactions | **Global Load Transactions**: Number of global memory load transactions |
| gst_transactions | **Global Store Transactions**: Number of global memory store transactions |
| gld_efficiency | **Global Memory Load Efficiency**: Ratio of requested global memory load throughput to required global memory load throughput |
| gst_efficiency | **Global Memory Load Efficiency**: Ratio of requested global memory store throughput to required global memory store throughput. tex_cache_transactions: Unified cache read transactions |
| global_hit_rate | **Global Hit Rate in unified l1/tex**: Hit rate for global loads in unified l1/tex cache |

# Appendix B

## Results of Varying grid and block sizes for a fixed world size

### XGPC Results

| Sample 1 | Grid Size x Block Size | | | | |
|---|---|---|---|---|---|
| | **80x1024** | **160x1024** | **320x1024** | **512x1024** | **1024x1024** |
| **Block-wise** | 0.31 | 0.383 | 0.3 | 0.304 | 0.319 |
| **Block cyclic** | 1.7 | 1.69 | 1.768 | 1.787 | 1.699 |
| **Shared Memory** | 1.9 | 2.168 | 2.142 | 1.384 | 1.9575 |
| **No Branch shared memory** | 1.879 | 2.029 | 2.269 | 1.877 | 1.932 |

| Sample 4 | Grid Size x Block Size | | | | |
|---|---|---|---|---|---|
| | **80x1024** | **160x1024** | **320x1024** | **512x1024** | **1024x1024** |
| **Block-wise** | 0.329 | 0.332 | 0.333 | 0.319 | 0.312 |
| **Block cyclic** | 3.396 | 2.151 | 2.625 | 3.235 | 2.283 |
| **Shared Memory** | 8.196 | 8.194 | 8.07 | 9.276 | 8.234 |
| **No Branch shared memory** | 8.285 | 8.495 | 8.062 | 8.258 | 8.064 |

| Sample 5 | Grid Size x Block Size | | | | |
|---|---|---|---|---|---|
| | **80x1024** | **160x1024** | **320x1024** | **512x1024** | **1024x1024** |
| **Block-wise** | 0.496 | 0.326 | 0.351 | 0.314 | 0.3266 |
| **Block cyclic** | 3.918 | 2.055 | 3.174 | 3.072 | 3.138 |
| **Shared Memory** | 4.64 | 4.509 | 4.72 | 4.723 | 4.7 |
| **No Branch shared memory** | 4.623 | 4.518 | 4.649 | 4.272 | 4.993 |

| Sample 7 | Grid Size x Block Size | | | | |
|---|---|---|---|---|---|
| | **80x1024** | **160x1024** | **320x1024** | **512x1024** | **1024x1024** |

| | | | | | |
|---|---|---|---|---|---|
| **Block-wise** | 18.079 | 18.15 | 18.21 | 19.719 | 18.878 |
| **Block cyclic** | 19.048 | 20.272 | 19.865 | 22.325 | 22.966 |
| **Shared Memory** | 20.12 | 19.822 | 19.941 | 22.432 | 21.171 |
| **No Branch shared memory** | 19.853 | 20.04 | 19.6425 | 21.945 | 19.521 |

## XGPD Results

| Sample 1 | Grid Size x Block Size | | | | |
|---|---|---|---|---|---|
| | **80x1024** | **160x1024** | **320x1024** | **512x1024** | **1024x1024** |
| **Block-wise** | 0.40188 | 0.3606 | 0.4005 | 0.4396 | 0.751 |
| **Block cyclic** | 0.22653 | 0.2602 | 0.2738 | 0.2529 | 0.259 |
| **Shared Memory** | 0.3801 | 0.2928 | 0.2837 | 0.3205 | 0.2928 |
| **No Branch shared memory** | 0.3053 | 0.2996 | 0.293 | 0.2665 | 0.2783 |

| Sample 4 | Grid Size x Block Size | | | | |
|---|---|---|---|---|---|
| | **80x1024** | **160x1024** | **320x1024** | **512x1024** | **1024x1024** |
| **Block-wise** | 2.712 | 2.528 | 2.617 | 2.44227 | 2.5614 |
| **Block cyclic** | 2.045 | 2.0055 | 2.0284 | 1.9954 | 2.0727 |
| **Shared Memory** | 7.1249 | 7.0367 | 7.0662 | 7.0269 | 7.0713 |
| **No Branch shared memory** | 6.9991 | 6.9024 | 7.0321 | 7.0037 | 6.9936 |

| Sample 5 | Grid Size x Block Size | | | | |
|---|---|---|---|---|---|
| | **80x1024** | **160x1024** | **320x1024** | **512x1024** | **1024x1024** |
| **Block-wise** | 2.461 | 2.479 | 2.487 | 2.5093 | 2.728 |
| **Block cyclic** | 3.5406 | 3.5303 | 3.5656 | 3.5263 | 3.5489 |
| **Shared Memory** | 6.5482 | 6.5526 | 6.6587 | 6.5478 | 6.4863 |
| **No Branch shared** | 6.4899 | 6.4882 | 6.4838 | 6.5075 | 6.5434 |

| | | | | | |
|---|---|---|---|---|---|
| **memory** | | | | | |

| Sample 7 | Grid Size x Block Size | | | | |
|---|---|---|---|---|---|
| | **80x1024** | **160x1024** | **320x1024** | **512x1024** | **1024x1024** |
| **Block-wise** | 20.842 | 22.611 | 22.722 | 23.91 | 21.5822 |
| **Block cyclic** | 23.1646 | 22.4244 | 22.4353 | 23.7099 | 21.7996 |
| **Shared Memory** | 24.126 | 22.623 | 22.3141 | 23.4296 | 21.6811 |
| **No Branch shared memory** | 23.825 | 22.489 | 21.768 | 23.051 | 21.4338 |

# Appendix C

## Analysis of key performance indicators on sample 7 with 100 iterations (retrieved with `nvprof`)

| | **Block-wise** | **Block cyclic** | **Shared Memory** | **No branch shared memory** |
|---|---|---|---|---|
| **Instructions per warp** | 1.81E+04 | 9.43E+03 | 1.18E+04 | 2.12E+04 |
| **Branch Efficiency** | 97.90% | 95.66% | 96.69% | 99.87% |
| **Warp Execution Efficiency** | 96.68% | 94.11% | 95.37% | 99.75% |
| **Shared Memory Load Transactions Per Request** | 0 | 0 | 1.301545 | 1.303405 |
| **Shared Memory Store Transactions Per Request** | 0 | 0 | 1.046517 | 1.048112 |
| **Global Load Transactions Per Request** | 4.483409 | 4.523879 | 3.657885 | 3.656575 |
| **Global Store Transactions Per Request** | 3.989362 | 4 | 3.989362 | 3.989362 |
| **Executed IPC** | 0.330145 | 0.370505 | 0.320736 | 0.344419 |
| **Multiprocessor Activity** | 94.73% | 91.73% | 95.44% | 94.98% |
| **Eligible Warps Per Active Cycle** | 2.151323 | 2.151199 | 2.02801 | 2.464219 |
| **Achieved Occupancy** | 0.940284 | 0.963414 | 0.993617 | 0.992674 |
| **Global Load Transactions** | 12640684 | 12720884 | 3093882 | 3092774 |
| **Global Store Transactions** | 1125000 | 1125000 | 1125000 | 1125000 |
| **Global Memory Load Efficiency** | 88.96% | 88.40% | 109.06% | 109.10% |
| **Global Memory Store Efficiency** | 100.00% | 100.00% | 100.00% | 100.00% |
| **Global Hit Rate in unified l1/tex** | 69.84% | 66.49% | 5.39% | 6.41% |
| **Control-Flow Function Unit Utilization** | Low (1) | Low (1) | Low (1) | Low (1) |

# Appendix D

## Fixed number of threads (1024), varying number of blocks

| | 512 blocks | 320 blocks | 160 blocks | 80 blocks | 40 blocks |
|---|---|---|---|---|---|
| **Instructions per warp** | 1.81E+04 | 2.88E+04 | 5.76E+04 | 1.15E+05 | 2.30E+05 |
| **Branch Efficiency** | 97.90% | 97.89% | 97.89% | 97.89% | 97.89% |
| **Warp Execution Efficiency** | 96.68% | 96.67% | 96.67% | 96.66% | 96.66% |
| **Shared Memory Load Transactions Per Request** | 0 | 0 | 0 | 0 | 0 |
| **Shared Memory Store Transactions Per Request** | 0 | 0 | 0 | 0 | 0 |
| **Global Load Transactions Per Request** | 4.483546 | 4.484121 | 4.484125 | 4.479202 | 4.472419 |
| **Global Store Transactions Per Request** | 3.989362 | 3.989362 | 3.989362 | 3.989362 | 3.989362 |
| **Executed IPC** | 0.329518 | 0.335832 | 0.32882 | 0.379732 | 0.775753 |
| **Multiprocessor Activity** | 94.89% | 92.35% | 92.27% | 87.47% | 43.30% |
| **Eligible Warps Per Active Cycle** | 2.151691 | 2.136321 | 2.148502 | 0.951389 | 1.612222 |
| **Achieved Occupancy** | 0.939813 | 0.948859 | 0.966529 | 0.485506 | 0.481887 |
| **Global Load Transactions** | 12641070 | 12642691 | 12642703 | 12628823 | 12609699 |
| **Global Store Transactions** | 1125000 | 1125000 | 1125000 | 1125000 | 1125000 |
| **Global Memory Load Efficiency** | 88.96% | 88.95% | 88.95% | 89.05% | 89.18% |

| | | | | | |
|---|---|---|---|---|---|
| **Global Memory Store Efficiency** | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| **Global Hit Rate in unified l1/tex** | 69.84% | 69.08% | 68.11% | 71.05% | 71.10% |
| **Control-Flow Function Unit Utilization** | Low (1) | Low (1) | Low (1) | Low (1) | Low (1) |

## Fixed grid size (160), varying block size

| | **1024 threads** | **512 threads** | **256 threads** | **128 threads** | **64 threads** | **32 threads** |
|---|---|---|---|---|---|---|
| **Instructions per warp** | 5.76E+04 | 1.13E+05 | 2.23E+05 | 4.43E+05 | 8.83E+05 | 1.76E+06 |
| **Branch Efficiency** | 97.89% | 97.84% | 97.81% | 97.80% | 97.79% | 97.79% |
| **Warp Execution Efficiency** | 96.67% | 96.59% | 96.55% | 96.54% | 96.53% | 96.52% |
| **Shared Memory Load Transactions Per Request** | 0 | 0 | 0 | 0 | 0 | 0 |
| **Shared Memory Store Transactions Per Request** | 0 | 0 | 0 | 0 | 0 | 0 |
| **Global Load Transactions Per Request** | 4.484044 | 4.482306 | 4.472739 | 4.448721 | 4.541886 | 4.582667 |
| **Global Store Transactions Per Request** | 3.989362 | 3.989362 | 3.989362 | 3.989362 | 3.989362 | 3.989362 |
| **Executed IPC** | 0.32905 | 0.383389 | 0.393457 | 0.397365 | 0.36022 | 0.193337 |
| **Multiproces** | 92.29% | 87.40% | 86.48% | 83.41% | 91.40% | 98.76% |

| sor Activity | | | | | | |
|---|---|---|---|---|---|---|
| **Eligible Warps Per Active Cycle** | 2.148964 | 0.93343 | 0.475767 | 0.340544 | 0.263881 | 0.156573 |
| **Achieved Occupancy** | 0.966629 | 0.488257 | 0.247173 | 0.123738 | 0.061646 | 0.030815 |
| **Global Load Transactions** | 12642474 | 12637573 | 12610600 | 12542885 | 12805556 | 12920537 |
| **Global Store Transactions** | 1125000 | 1125000 | 1125000 | 1125000 | 1125000 | 1125000 |
| **Global Memory Load Efficiency** | 88.95% | 88.98% | 89.17% | 89.66% | 87.82% | 87.04% |
| **Global Memory Store Efficiency** | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| **Global Hit Rate in unified l1/tex** | 68.09% | 70.45% | 72.53% | 72.91% | 71.54% | 70.95% |
| **Control-Flow Function Unit Utilization** | Low (1) | Low (1) | Low (1) | Low (1) | Low (1) | Low (1) |

# Fixed number of threads (512)

Column headers: Grid Size x Block Size

| | **1x512** | **2x256** | **4x128** | **8x64** | **16x32** | **32x16** | **64x8** | **128x4** | **256x2** | **512x1** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Instructions per warp** | 1.80E+07 | 1.78E+07 | 1.77E+07 | 1.77E+07 | 1.76E+07 | 1.74E+07 | 1.72E+07 | 1.71E+07 | 1.71E+07 | 1.70E+07 |
| **Branch Efficiency** | 97.84% | 97.81% | 97.80% | 97.79% | 97.79% | 98.63% | 99.21% | 99.53% | 99.77% | 100.00% |
| **Warp Execution Efficiency** | 96.59% | 96.55% | 96.53% | 96.53% | 96.52% | 48.86% | 24.72% | 12.43% | 6.23% | 3.12% |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Shared Memory Load Transactions Per Request** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Shared Memory Store Transactions Per Request** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Global Load Transactions Per Request** | 4.571027 | 4.497484 | 4.441133 | 4.548172 | 4.582718 | 2.591476 | 1.598387 | 1.299193 | 1.149597 | 1.000006 |
| **Global Store Transactions Per Request** | 3.989362 | 3.989362 | 3.989362 | 3.989362 | 3.989362 | 1.994681 | 1 | 1 | 1.000093 | 1.000132 |
| **Executed IPC** | 1.105238 | 0.744525 | 0.396521 | 0.194577 | 9.79% | 10.00% | 10.75% | 17.66% | 35.31% | 0.619685 |
| **Multiprocessor Activity** | 1.25% | 2.50% | 5.00% | 9.99% | 19.94% | 39.72% | 79.58% | 97.99% | 98.95% | 78.55% |
| **Eligible Warps Per Active Cycle** | 1.098997 | 0.671473 | 0.328683 | 0.163359 | 0.085112 | 0.089178 | 0.105881 | 0.197058 | 0.41389 | 0.741067 |
| **Achieved Occupancy** | 0.240487 | 0.122901 | 0.062499 | 0.03125 | 0.015625 | 0.015625 | 0.015625 | 0.024733 | 0.048793 | 0.095427 |
| **Global Load Transactions** | 12887718 | 12680367 | 12521491 | 12823279 | 12920678 | 14613002 | 17978254 | 29226004 | 51721504 | 89964921 |
| **Global Store Transactions** | 1125000 | 1125000 | 1125000 | 1125077 | 1125000 | 1125000 | 1125000 | 2250000 | 4500419 | 9001679 |
| **Global Memory Load Efficiency** | 87.26% | 88.68% | 89.81% | 87.70% | 87.03% | 76.96% | 62.55% | 38.48% | 21.74% | 12.50% |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Global Memory Store Efficiency** | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 50.00% | 25.00% | 12.50% |
| **Global Hit Rate in unified l1/tex** | 81.05% | 80.30% | 73.03% | 71.46% | 70.95% | 75.86% | 80.85% | 88.38% | 93.49% | 96.30% |
| **Control-Flow Function Unit Utilization** | Low (1) | Low (1) | Low (1) | Low (1) | Low (1) | Low (1) | Low (1) | Low (1) | Low (1) | Low (1) |
| **Timings** | 630s | 408s | 378s | 383s | 368s | 347s | 289s | 250s | 239s | 293s |