1 Results and Discussion

Following are the plots on execution time against block size for various world sizes.

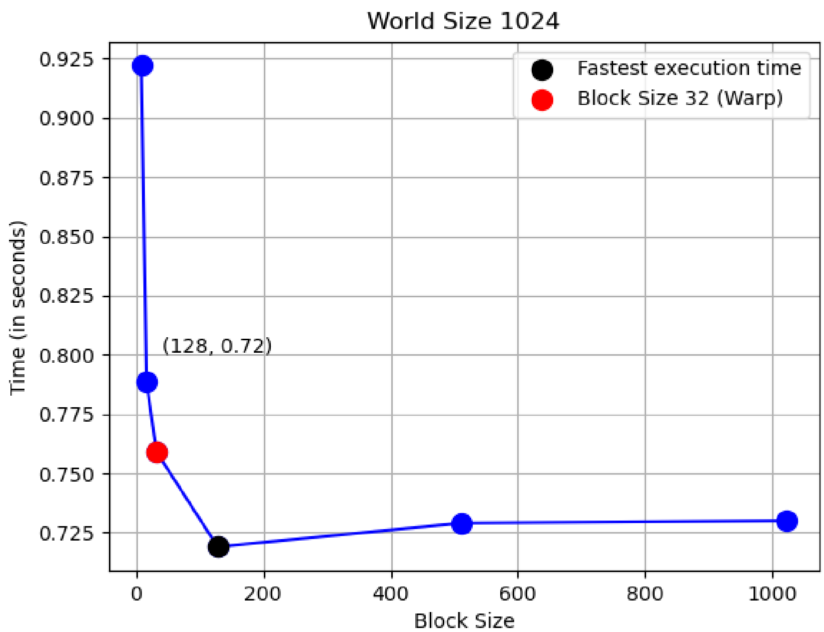

Figure 1: Execution time vs thread block size for world height/width 1024. **The fastest execution time is 0.72 second when the block size of 128 is used**.

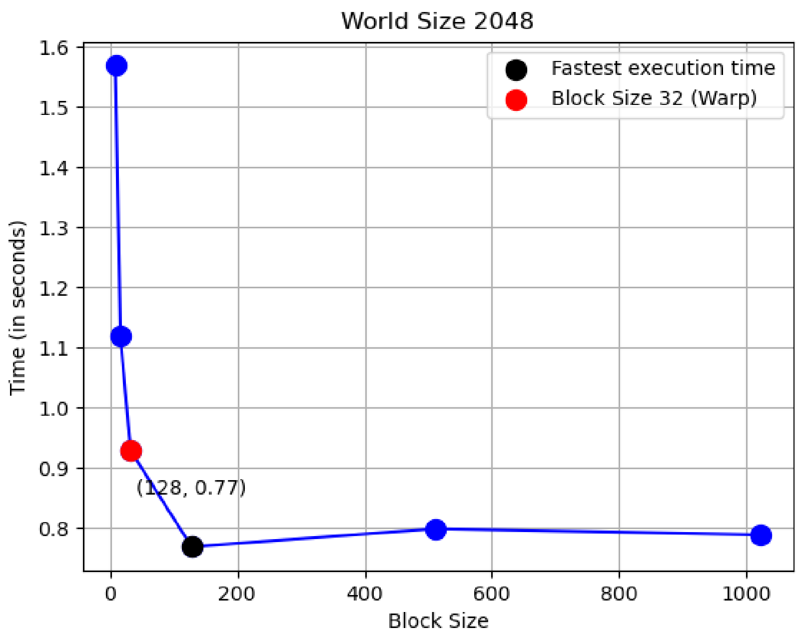

Figure 2: Execution time vs thread block size for world height/width 2048. **The fastest execution time is 0.77 second when the block size of 128 is used**.

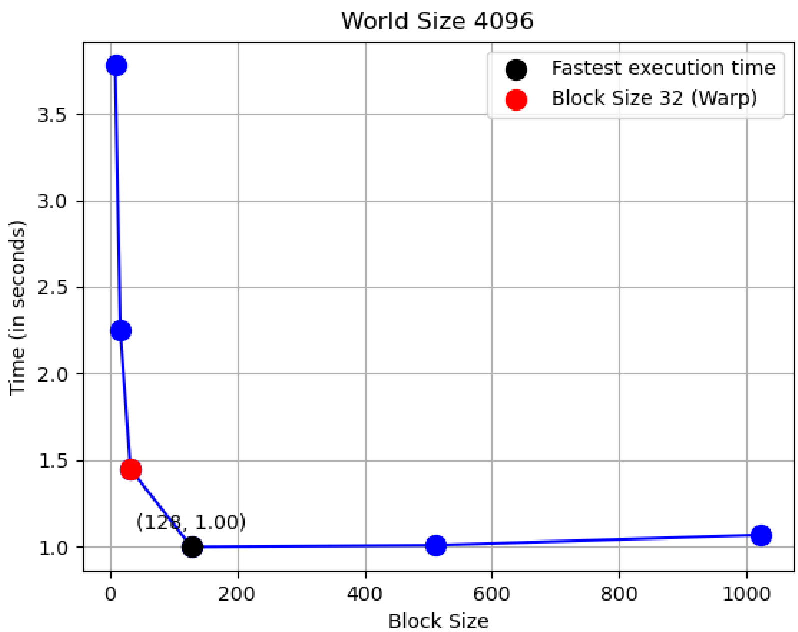

Figure 3: Execution time vs thread block size for world height/width 4096. **The fastest execution time is 0.99 second when the block size of 128 is used**.

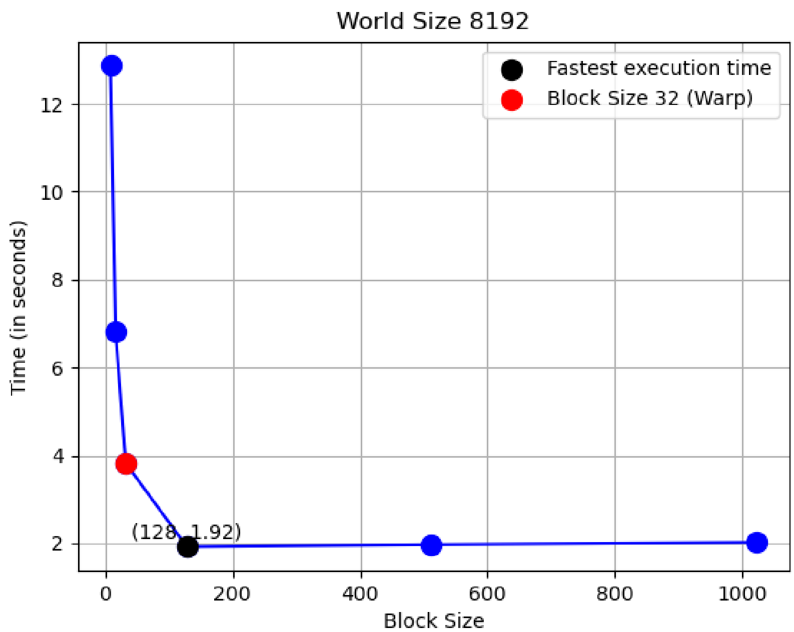

Figure 4: Execution time vs thread block size for world height/width 8192. **The fastest execution time is 1.92 second when the block size of 128 is used**.

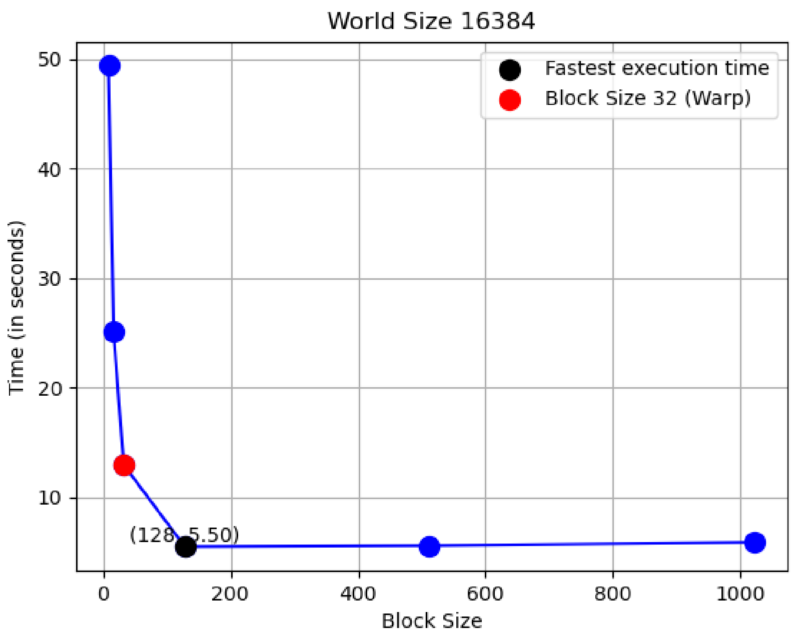

Figure 5: Execution time vs thread block size for world height/width 16384. **The fastest execution time is 5.50 second when the block size of 128 is used**.

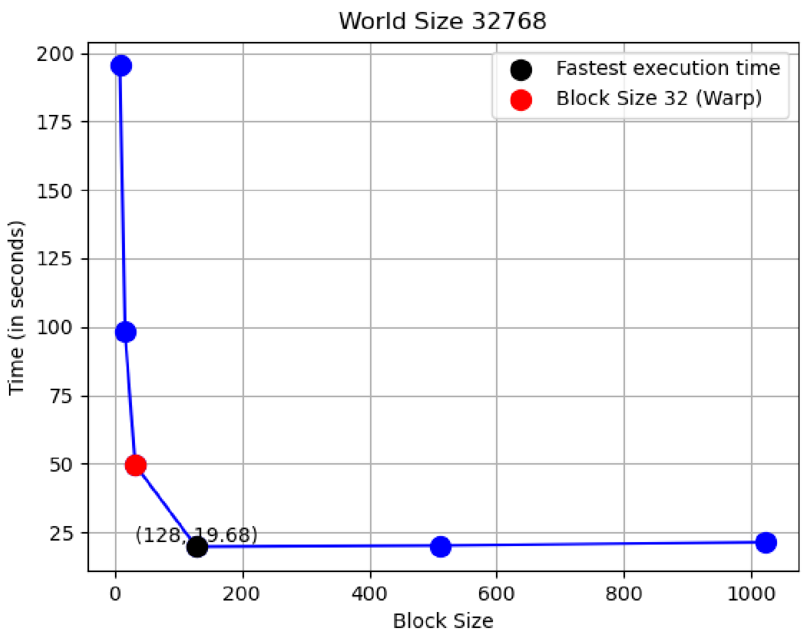

Figure 6: Execution time vs thread block size for world height/width 32768. **The fastest execution time is 19.68 second when the block size of 128 is used**.

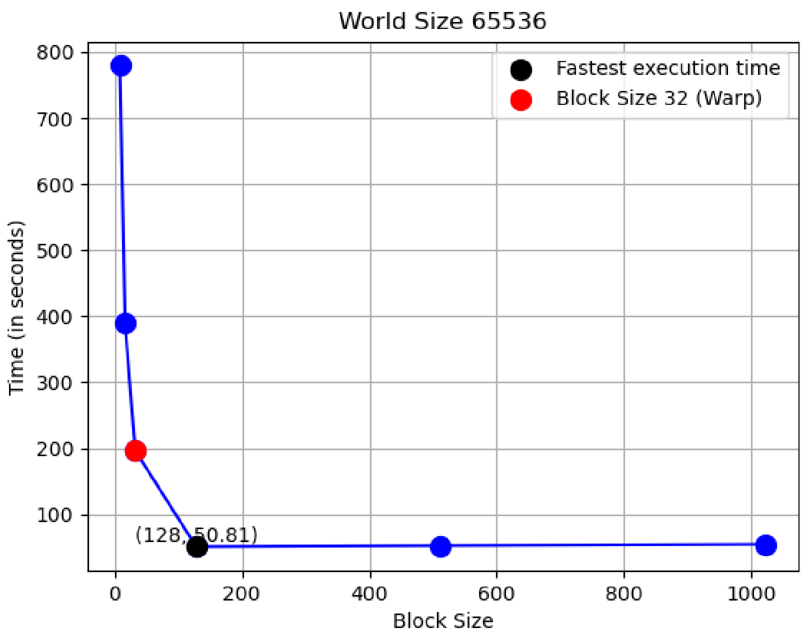

Figure 7: Execution time vs thread block size for world height/width 65536. **The fastest execution time is 50.81 second when the block size of 128 is used**.

The following observations can be made from the above plots:

1. Across all world sizes, a consistent trend emerges where execution times tend to be higher when block size is below 32.
2. Conversely, execution times generally decrease notably once block size exceeds 32.
3. For all the world sizes, the execution time is almost the same for block sizes 128, 512, and 1024.
4. Although CUDA hardware permits block sizes down to a single thread per block, the use of sizes smaller than 32 often leads to heightened memory dependency, increased latency, suboptimal occupancy, and inefficient utilization of hardware resources. Moreover, smaller block sizes result in concurrent warps and diminished parallelism, ultimately diminishing the kernel performance.
5. The performance appears to be better for block sizes that are multiples of 32 which can be seen from figures 1, 2, 3, 4, 5, 6 and 7. This phenomenon can be attributed to their alignment with the hardware's warp size, which serves as the fundamental unit of execution on an NVIDIA GPU.
6. From the depicted figures, it can be noted that the optimal block size is within the range of 128 to 1024, where the execution times are minimum and close.

World Size	Block Size					
	8	16	32	128	512	1024
1024	1.164579e+09	1.360890e+09	1.414680e+09	1.493382e+09	1.472897e+09	1.470879e+09
2048	2.737392e+09	3.838219e+09	4.623216e+09	5.585133e+09	5.375428e+09	5.443558e+09
4096	4.546142e+09	7.638892e+09	1.185636e+10	1.719707e+10	1.702663e+10	1.607097e+10
8192	5.339924e+09	1.007765e+10	1.798939e+10	3.581005e+10	3.490070e+10	3.403639e+10
16384	5.565569e+09	1.094739e+10	2.119500e+10	4.998689e+10	4.909411e+10	4.651852e+10
32768	5.628160e+09	1.119791e+10	2.216355e+10	5.587233e+10	5.459614e+10	5.143192e+10
65536	5.642869e+09	1.125888e+10	2.237634e+10	8.656038e+10	8.361305e+10	8.034282e+10

Table 1: Cell update per second for various world and block size configuration

Table 1 gives the cell update per second for various configurations of block sizes and world sizes. The blue color on the table highlights the fastest cell update per second which is 8.65×10^{10}. The fastest cell update per second is obtained for the world size of 65536 and block size of 128. It implies that with a block size of 128, the CUDA implementation is able to efficiently utilize the available GPU resources to process a significant number of cell updates per second, resulting in optimal performance. For the given world size, serial cell update per second comes to be around 9×10^7. The speedup of CUDA code relative to serial code is around 900. That means the parallel code is **900X** faster than the serial code. This underscores the significant performance advantage offered by parallel computing. This substantial speedup demonstrates the power of leveraging GPU parallelism to handle the computational workload highlighting the effectiveness of CUDA. This study also emphasizes the importance of block size in leveraging the potential of GPU. The performance is better if we choose the block size of the multiple of the warp size (32) as it can help maximize efficiency because it ensures that each warp is fully utilized.

To make the execution of the code easier, we can use 4 arguments instead of only 3. The 4th argument is block size. The implementation is given in the following code snippet. Since there were 42 cases to run, I wrote a bash script for it. Similarly, for the post-processing of data, I used Python. All the codes along with the output are listed below.

```
if( argc != 5 )    {
        printf("HighLife requires 4 arguments, 1st is pattern number, 2nd the sq size of the world, 3rd is the number of iterations and 4th is the thread blocksize, e.g. ./highlife 0 32 2 128 \n");
        exit(-1);
    }

    pattern = atoi(argv[1]);
    worldLength = atoi(argv[2]);
    iterations = atoi(argv[3]);
    threadBlockSize = atoi(argv[4]);
```

Listing 1: Instruction to run highlife code using 4 arguments

```bash
#!/bin/bash

module load xl_r spectrum-mpi cuda

# Number of blockSize
blockSize=( "8" "16" "32" "128" "512" "1024")

# World size
worldSize=("1024" "2048" "4096" "8192" "16384" "32768" "65536")

pattern="5"

iterations="1024"

# Output file

output_file="output.txt"

# Loop through each block Size
for threads in "${blockSize[@]}"; do
    echo "Running cases for blockSize: $threads"

    # Loop through World size
    for world in "${worldSize[@]}"; do
        echo "Running cases for worldSize: $world"

        # Print information about the thread and case
        echo "Block_Size: $threads : World_Size: $world" >> $output_file

        # execution time using the 'time' command and redirect output to file
       { time ./highlife $pattern $world $iterations $threads; } 2>&1 | tee -a $output_file

        echo "---------------------------------------"
    done

    echo "---------------------------------------"
done
```

Listing 2: script for job submission

```
Current running configuration - Thread Block Size: 8 ,
     World Size: 1024

real  0m0.922s
user  0m0.206s
sys 0m0.514s
Current running configuration - Thread Block Size: 8 ,
     World Size: 2048

real  0m1.569s
user  0m0.615s
sys 0m0.645s
Current running configuration - Thread Block Size: 8 ,
     World Size: 4096

real  0m3.779s
user  0m2.338s
sys 0m1.204s
Current running configuration - Thread Block Size: 8 ,
     World Size: 8192

real  0m12.869s
user  0m9.159s
sys 0m3.503s
Current running configuration - Thread Block Size: 8 ,
     World Size: 16384

real  0m49.389s
user  0m35.116s
sys 0m14.052s
Current running configuration - Thread Block Size: 8 ,
     World Size: 32768

real  3m15.359s
user  2m22.094s
sys 0m53.065s
Current running configuration - Thread Block Size: 8 ,
     World Size: 65536

real  12m59.399s
user  9m31.138s
sys 3m28.139s
Current running configuration - Thread Block Size: 16 ,
      World Size: 1024

real  0m0.789s
user  0m0.092s
sys 0m0.499s
Current running configuration - Thread Block Size: 16 ,
```

```
      World Size: 2048

real  0m1.119s
user  0m0.357s
sys 0m0.520s
Current running configuration - Thread Block Size: 16 ,
      World Size: 4096

real  0m2.249s
user  0m1.110s
sys 0m0.908s
Current running configuration - Thread Block Size: 16 ,
      World Size: 8192

real  0m6.819s
user  0m4.547s
sys 0m2.043s
Current running configuration - Thread Block Size: 16 ,
      World Size: 16384

real  0m25.109s
user  0m17.662s
sys 0m7.205s
Current running configuration - Thread Block Size: 16 ,
      World Size: 32768

real  1m38.189s
user  1m9.230s
sys 0m28.748s
Current running configuration - Thread Block Size: 16 ,
      World Size: 65536

real  6m30.629s
user  4m46.379s
sys 1m44.097s
Current running configuration - Thread Block Size: 32 ,
      World Size: 1024

real  0m0.759s
user  0m0.061s
sys 0m0.484s
Current running configuration - Thread Block Size: 32 ,
      World Size: 2048

real  0m0.929s
user  0m0.172s
sys 0m0.516s
Current running configuration - Thread Block Size: 32 ,
      World Size: 4096

```

```
real  0m1.449s
user  0m0.589s
sys 0m0.670s
Current running configuration - Thread Block Size: 32 ,
      World Size: 8192

real  0m3.820s
user  0m2.283s
sys 0m1.271s
Current running configuration - Thread Block Size: 32 ,
      World Size: 16384

real  0m12.969s
user  0m8.931s
sys 0m3.800s
Current running configuration - Thread Block Size: 32 ,
      World Size: 32768

real  0m49.609s
user  0m35.982s
sys 0m13.455s
Current running configuration - Thread Block Size: 32 ,
      World Size: 65536

real  3m16.549s
user  2m22.398s
sys 0m53.895s
Current running configuration - Thread Block Size: 128
    ,  World Size: 1024

real  0m0.719s
user  0m0.040s
sys 0m0.471s
Current running configuration - Thread Block Size: 128
    ,  World Size: 2048

real  0m0.769s
user  0m0.093s
sys 0m0.476s
Current running configuration - Thread Block Size: 128
    ,  World Size: 4096

real  0m0.999s
user  0m0.253s
sys 0m0.536s
Current running configuration - Thread Block Size: 128
    ,  World Size: 8192

real  0m1.919s
user  0m1.021s
```

```
sys 0m0.657s
Current running configuration - Thread Block Size: 128, World Size: 16384

real  0m5.499s
user  0m3.452s
sys 0m1.781s
Current running configuration - Thread Block Size: 128, World Size: 32768

real  0m19.679s
user  0m14.246s
sys 0m5.186s
Current running configuration - Thread Block Size: 128, World Size: 65536

real  0m50.809s
user  0m37.452s
sys 0m13.144s
Current running configuration - Thread Block Size: 512, World Size: 1024

real  0m0.729s
user  0m0.062s
sys 0m0.450s
Current running configuration - Thread Block Size: 512, World Size: 2048

real  0m0.799s
user  0m0.072s
sys 0m0.499s
Current running configuration - Thread Block Size: 512, World Size: 4096

real  0m1.009s
user  0m0.251s
sys 0m0.551s
Current running configuration - Thread Block Size: 512, World Size: 8192

real  0m1.969s
user  0m0.935s
sys 0m0.774s
Current running configuration - Thread Block Size: 512, World Size: 16384

real  0m5.599s
user  0m3.816s
sys 0m1.542s
Current running configuration - Thread Block Size: 512
```

```
    , World Size: 32768

real  0m20.139s
user  0m14.602s
sys 0m5.324s
Current running configuration - Thread Block Size: 512
    , World Size: 65536

real  0m52.600s
user  0m10.071s
sys 0m4.289s
Current running configuration - Thread Block Size: 1024
     , World Size: 1024

real  0m0.730s
user  0m0.051s
sys 0m0.465s
Current running configuration - Thread Block Size: 1024
     , World Size: 2048

real  0m0.789s
user  0m0.072s
sys 0m0.504s
Current running configuration - Thread Block Size: 1024
     , World Size: 4096

real  0m1.069s
user  0m0.250s
sys 0m0.570s
Current running configuration - Thread Block Size: 1024
     , World Size: 8192

real  0m2.019s
user  0m1.084s
sys 0m0.703s
Current running configuration - Thread Block Size: 1024
     , World Size: 16384

real  0m5.909s
user  0m4.022s
sys 0m1.635s
Current running configuration - Thread Block Size: 1024
     , World Size: 32768

real  0m21.378s
user  0m15.258s
sys 0m5.883s
Current running configuration - Thread Block Size: 1024
     , World Size: 65536

```

```
real	0m54.741s
user	0m5.845s
sys	0m2.687s
```

Listing 3: Output

```python
import numpy as np
import re
import pandas as pd

data = open('output.txt', 'r').read()

# Extract block sizes, world sizes and real times
block_sizes = re.findall(r"Block_Size: (\d+)", data)
world_sizes = re.findall(r"World_Size: (\d+)", data)
real_times = re.findall(r"real\t(\d+m\d+\.\d+s)", data)

# Convert real times to seconds
real_times_in_seconds = [int(m.split('m')[0])*60 + float(m.split('m')[1][:-1]) for m in real_times]

# Create a DataFrame
df = pd.DataFrame({
    'Block_Size': block_sizes,
    'World_Size': world_sizes,
    'Real_Time_in_Seconds': real_times_in_seconds
})

# Pivot the DataFrame to get each block size as a column
df_pivot = df.pivot(index='World_Size', columns='Block_Size', values='Real_Time_in_Seconds')
df_pivot.index = df_pivot.index.astype(int)
df_pivot.columns = df_pivot.columns.astype(int)

df_pivot_sorted = df_pivot.sort_index().sort_index(axis=1)
print(df_pivot_sorted)
print("\n")

# Find the location of the minimum value in the DataFrame
min_time_location = df_pivot.stack().idxmin()

# min_time_location is a tuple (World_Size, Block_Size)
world_size, block_size = min_time_location

# Get the minimum time
min_time = df_pivot.loc[world_size, block_size]

print(f"Minimum time is {min_time} at World_Size {world_size} and Block_Size {block_size}")
```

Listing 4: Python code used for obtaining the executable times for various configurations from raw data

```python
import matplotlib.pyplot as plt
import numpy as np

worldsize = df_pivot_sorted.index.values
blocksize = df_pivot_sorted.columns.values

for i in range(len(worldsize)):
    plt.figure()  # Create a new figure
    y_values = df_pivot_sorted.iloc[i,:].values
    plt.plot(blocksize, y_values, marker='o', color='b', markersize=10)
    plt.title(f'World Size {worldsize[i]}')  # Set the title
    plt.xlabel('Block Size')  # Set the x-axis label
    plt.ylabel('Time (in seconds)')  # Set the y-axis label
    plt.grid(True)  # Add a grid



    # Find the minimum value and its index
    min_index = np.argmin(y_values)
    min_value = y_values[min_index]
    blocksize_32 = y_values[2]

    # Plot a large dot at the minimum value
    plt.scatter(blocksize[min_index], min_value ,color='k',zorder=5,label='Fastest execution time',s=100)

    plt.scatter(blocksize[2], blocksize_32, color='r', zorder=5, label='Block Size 32 (Warp)',s=100)

    plt.legend()  # Add a legend

    # Write the coordinates above the point
    plt.text(blocksize[min_index], min_value + 0.08, f'({blocksize[min_index]}, {min_value:.2f})', ha='center', va='bottom')

   # plt.text(blocksize[2], blocksize_32 + 0.1, f'({blocksize[2]}, {blocksize_32:.2f})', ha='center', va='bottom')

    plt.savefig(f'plot_{worldsize[i]}.png')  # Save the figure
```

Listing 5: Python code for plots

```python
cell_update = worldsize * worldsize * 1024
time_grid = np.zeros((len(worldsize),len(blocksize)))
for i in range (len(worldsize)):
    time_grid[i,:]= cell_update[i]/df_pivot_sorted.iloc[i,:].values

# Find the index of the minimum value in the flattened array
index_flat = np.argmax(time_grid)

# Convert the index in the flattened array to a row and column index
row_index, col_index = np.unravel_index(index_flat, time_grid.shape)
df = pd.DataFrame(time_grid)

print(df)
print("\n")
print(f"Minimum cell update per second is for World Height {worldsize[row_index]} and block size {blocksize[col_index]} : {time_grid[row_index, col_index]}")
```

Listing 6: Python code for creating the table for cells update per second for each configuration