ation-assignment-6-hcp-ui22cs03-2

October 13, 2024

Game of Life Simulation on CUDA is a popular cellular automaton that simulates the evolution of cells on a 2D grid based on simple rules. It's an ideal candidate for parallel processing because the state of each cell depends only on its neighbouring cells, making it a highly local computation that can be performed in parallel.Problem

Description: The grid consists of cells that can be either (1) or (0).

The state of each cell in the next generation is determined by its 8 neighbours using the following rules:: A live cell with fewer than two live neighbours dies.: A live cell with more than three live neighbours dies.: A dead cell with exactly three live neighbours becomes a live cell.: A live cell with two or three live neighbours stays alive.

Also, add a visualization of the grid in each generation or save the results in a file to create an animation.

[ ]:
```
!nvidia-smi
```

```
Thu Oct 10 10:50:46 2024
+-----------------------------------------------------------------------------
--------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version:
12.2     |
|-----------------------------------------+----------------------+--------------
--------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile
Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util
Compute M. |
|                                         |                      |
MIG M. |
|=========================================+======================+==============
========|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |
0 |
| N/A   40C    P8               9W /  70W |      0MiB / 15360MiB |      0%
Default |
|                                         |                      |
N/A |
+-----------------------------------------+----------------------+--------------
--------+
```

```
+-----------------------------------------------------------------------------
--------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name                            GPU
Memory |
|        ID   ID
Usage      |
|=============================================================================
========|
|  No running processes found
|
+-----------------------------------------------------------------------------
--------+
```

[1]:
```
# This cell can be skipped in Colab, as numba and other libraries are␣
↪pre-installed
!pip install numba numpy matplotlib
```

```
Requirement already satisfied: numba in /usr/local/lib/python3.10/dist-packages
(0.60.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages
(1.26.4)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-
packages (3.7.1)
Requirement already satisfied: llvmlite<0.44,>=0.43.0dev0 in
/usr/local/lib/python3.10/dist-packages (from numba) (0.43.0)
Requirement already satisfied: contourpy>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (1.3.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-
packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (4.54.1)
Requirement already satisfied: kiwisolver>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (1.4.7)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (24.1)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-
packages (from matplotlib) (10.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (3.1.4)
Requirement already satisfied: python-dateutil>=2.7 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-
packages (from python-dateutil>=2.7->matplotlib) (1.16.0)
```

[13]:

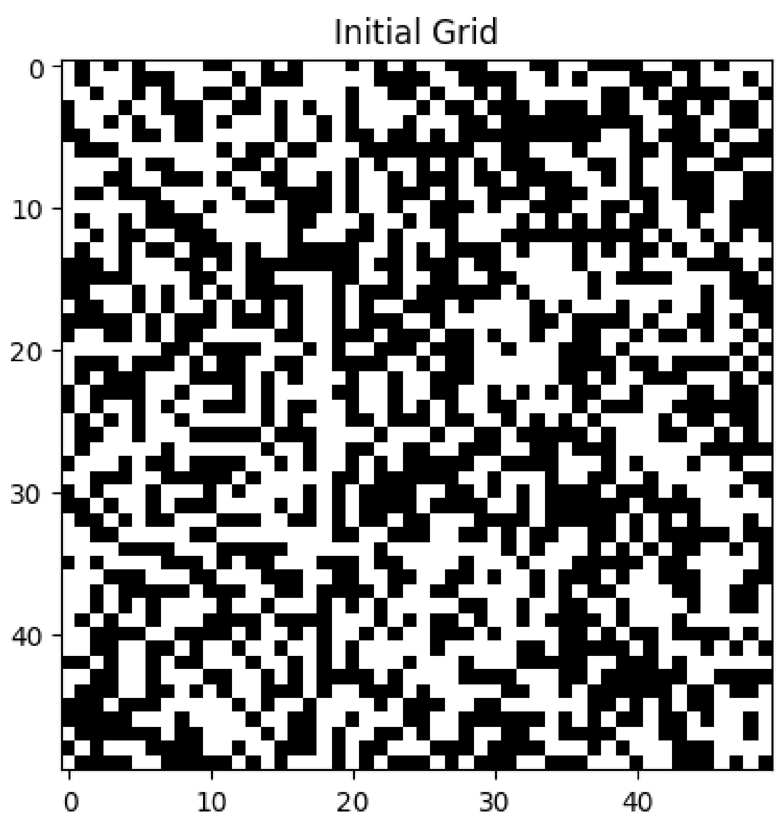
```
# Numba is pre-installed in Google Colab
import numpy as np
import matplotlib.pyplot as plt
from numba import cuda
from PIL import Image
```

[14]:

```
# Initialize the grid with random 1s (alive) and 0s (dead)
def initialize_grid(size):
    return np.random.randint(2, size=(size, size))

# Grid size (smaller to easily visualize multiple generations)
grid_size = 50
grid = initialize_grid(grid_size)

# Display the initial grid
plt.imshow(grid, cmap='binary')
plt.title("Initial Grid")
plt.show()
```



[15]:

```
# CUDA kernel to update the grid based on Game of Life rules
@cuda.jit
```

```
def update_grid_cuda(grid, new_grid, N):
    x, y = cuda.grid(2)

    if x < N and y < N:
        # Count live neighbors
        live_neighbors = 0
        for i in range(-1, 2):
            for j in range(-1, 2):
                if i != 0 or j != 0:
                    nx = (x + i + N) % N  # Handle wrap-around for boundaries
                    ny = (y + j + N) % N
                    live_neighbors += grid[nx, ny]

        # Apply Game of Life rules
        if grid[x, y] == 1:  # Cell is alive
            if live_neighbors < 2 or live_neighbors > 3:
                new_grid[x, y] = 0  # Cell dies
            else:
                new_grid[x, y] = 1  # Cell survives
        else:  # Cell is dead
            if live_neighbors == 3:
                new_grid[x, y] = 1  # Cell becomes alive
            else:
                new_grid[x, y] = 0  # Cell remains dead
```

[23]:

```
import time

# Function to run the simulation using CUDA
def run_simulation_cuda(grid, generations):
    # Start time recording
    start_time = time.time()

    # Copy the grid to the GPU
    grid_device = cuda.to_device(grid)
    new_grid_device = cuda.device_array_like(grid)

    # Thread and block configuration
    threads_per_block = (16, 16)
    blocks_per_grid = (grid_size // threads_per_block[0], grid_size //␣
 ↪threads_per_block[1])

    states = []

    for gen in range(generations):
        # Run the kernel
        update_grid_cuda[blocks_per_grid, threads_per_block](grid_device,␣
 ↪new_grid_device, grid_size)
```

```
        # Swap grids for the next generation
        grid_device, new_grid_device = new_grid_device, grid_device

        # Copy the updated grid to host and store it
        states.append(grid_device.copy_to_host())

    # End time recording
    end_time = time.time()

    # Calculate elapsed time
    elapsed_time = end_time - start_time
    print(f"Time taken for {generations} generations on GPU: {elapsed_time:.4f}␣
↪seconds")

    return states

# Set number of generations
generations = 10

# Run the simulation on the GPU for 5 generations
states = run_simulation_cuda(grid, generations)
```

```
Time taken for 10 generations on GPU: 0.0125 seconds
```

[24]:
```
# Function to combine multiple grids into a single image
def combine_images(states):
    images = []

    for i, state in enumerate(states):
        fig, ax = plt.subplots(figsize=(5, 5))
        ax.imshow(state, cmap='binary')
        ax.set_title(f'Generation {i+1}')
        ax.axis('off')

        # Save the image of each generation to a buffer
        plt.savefig(f'gen_{i+1}.png', bbox_inches='tight')
        images.append(Image.open(f'gen_{i+1}.png'))
        plt.close()

    # Combine all images vertically
    widths, heights = zip(*(i.size for i in images))
    total_height = sum(heights)
    max_width = max(widths)

    combined_image = Image.new('RGB', (max_width, total_height))
```

```python
    y_offset = 0
    for img in images:
        combined_image.paste(img, (0, y_offset))
        y_offset += img.height

    return combined_image

# Combine and visualize the images for each generation
combined_image = combine_images(states)
combined_image.show()
```

[25]:
```python
# Display the combined image directly in the notebook
plt.figure(figsize=(10, 10))
plt.imshow(combined_image)
plt.axis('off')  # Hide axes
plt.show()
```

[ ]: