Conways game of life in CUDA

Introduction

This project implements **Conway's Game of Life** using **CUDA** to demonstrate **GPU** parallelism for fast and large-scale simulations on **Google collab**. The grid is initialized with a dense random pattern, filling 60% of the screen with live cells to create a dynamic evolution. By running on a **512×512 grid** over **500 iterations**.

This project essentially lets the user understand how to leverage **parallelism** for tasks that would otherwise consume a lot more time using only the CPU.

Conways game of life

Conways game of life is a **cellular automaton**; in simple words it's a game with 4 rules. Those rules are:

1. Any live cell with **fewer than two live neighbors dies**, as if by underpopulation.
2. Any live cell with **two or three live neighbors lives** on to the next generation.
3. Any live cell with **more than three live neighbors dies**, as if by overpopulation.
4. Any dead cell with **exactly three live neighbors becomes a live cell**, as if by reproduction.

It's easier as well as more intuitive to understand its working by giving it a try. Click here to try it online.

The reason why chose Conways game of life for CUDA is that there are **multiple calculations** that need to be done, however these are relatively simple calculations and don't need a lot of power. This makes it ideal to run it on an GPU.

CUDA

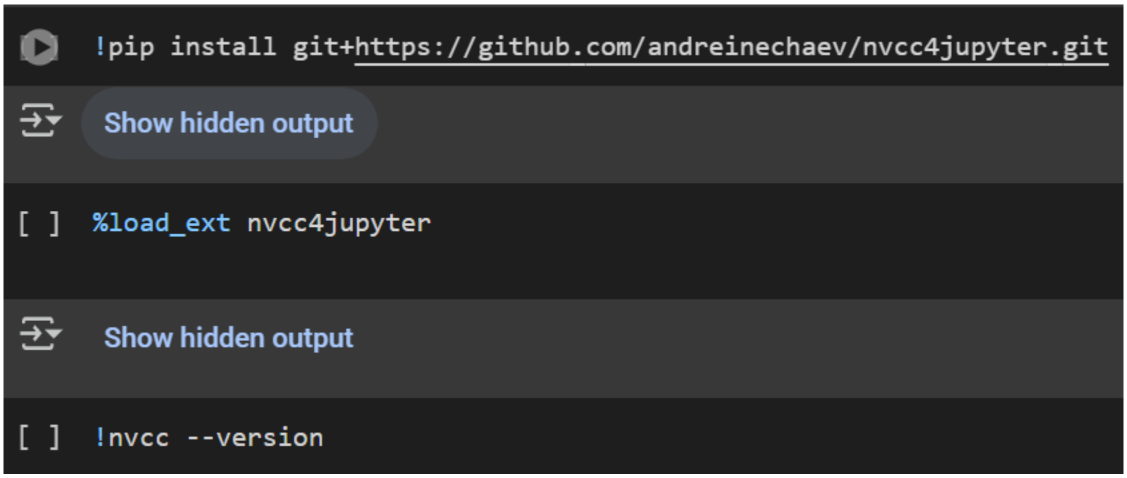
CUDA (Compute Unified Device Architecture) is the tool that lets us run our code on GPU's. Unlike traditional CPU programming, which executes instructions sequentially, CUDA enables thousands of **lightweight threads** to run in parallel. Click here to watch a 3-minute video explaining CUDA in detail.
The key components of CUDA that we will be using in our project are:

1. Kernels
2. Grids
3. Blocks
4. Shared memory

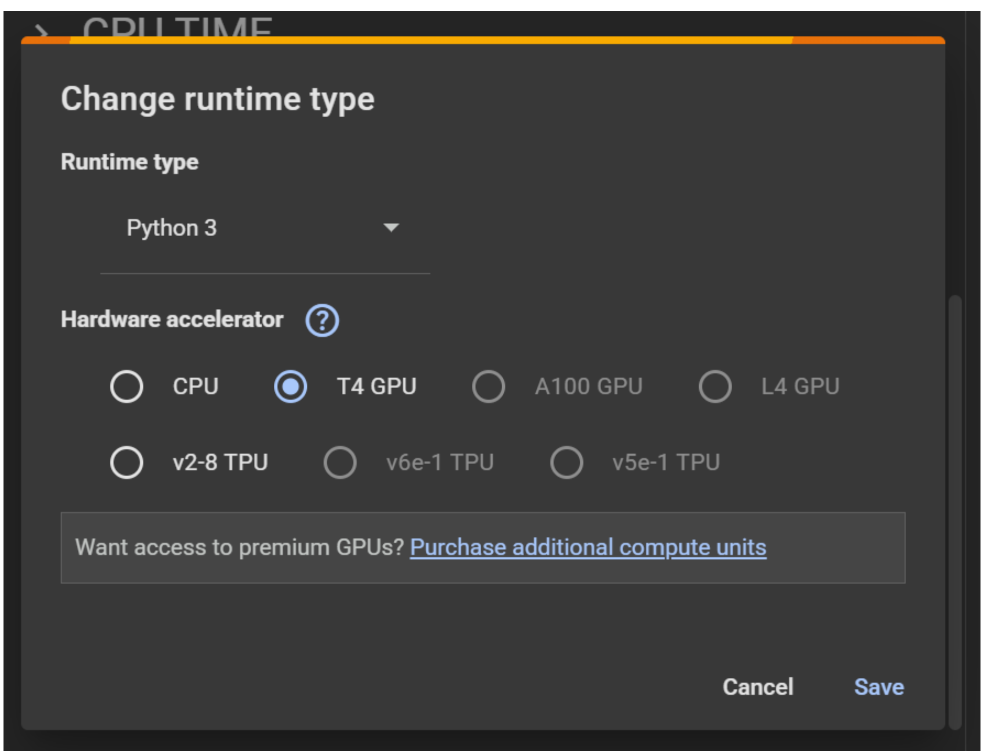
Code

We will be running the Entire code on Google Collab, making it accessible to everyone without requiring Actual GPU on your laptop!!

We install the prerequire library



Before, we run the code, we need to ensure that the we are connected to the T4 GPU. By default it connects to a CPU and won't work as intended.



The GPU code

```
%%writefile buu.cpp
#include <iostream>
#include <cuda_runtime.h>
#include <chrono>
#include <random>

#define WIDTH 512
#define HEIGHT 512
#define ITERATIONS 1000

__device__ int count_neighbors(int* grid, int x, int y, int width, int height) {
    int count = 0;
    for (int dx = -1; dx <= 1; ++dx) {
        for (int dy = -1; dy <= 1; ++dy) {
            if (dx == 0 && dy == 0) continue;
            int nx = (x + dx + width) % width;
            int ny = (y + dy + height) % height;
            count += grid[ny * width + nx];
        }
    }
    return count;
}

__global__ void game_of_life_step(int* current, int* next, int width, int height) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < width && y < height) {
        int idx = y * width + x;
        int neighbors = count_neighbors(current, x, y, width, height);
        int state = current[idx];
        if (state == 1 && (neighbors == 2 || neighbors == 3)) {
            next[idx] = 1;
        } else if (state == 0 && neighbors == 3) {
            next[idx] = 1;
        } else {
            next[idx] = 0;
        }
    }
}


void set_random_dense_pattern(int* grid, int width, int height, float fill_ratio = 0.6f) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::bernoulli_distribution d(fill_ratio);
```

```
int main() {
    int size = WIDTH * HEIGHT * sizeof(int);

    int* h_grid = new int[WIDTH * HEIGHT]();
    int* h_result = new int[WIDTH * HEIGHT]();

    // Fill about 60% of the screen with live cells
    set_random_dense_pattern(h_grid, WIDTH, HEIGHT, 0.6f);


    int* d_current;
    int* d_next;
    cudaMalloc(&d_current, size);
    cudaMalloc(&d_next, size);
    cudaMemcpy(d_current, h_grid, size, cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((WIDTH + 15) / 16, (HEIGHT + 15) / 16);

    // CUDA timing setup
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // gPU timing start
    auto gpu_start = std::chrono::high_resolution_clock::now();

    cudaEventRecord(start); // GPU timing start

    for (int i = 0; i < ITERATIONS; ++i) {
        game_of_life_step<<<numBlocks, threadsPerBlock>>>(d_current, d_next, WIDTH, HEIGHT);
        cudaDeviceSynchronize(); // Wait for kernel to finish

        std::swap(d_current, d_next);
        cudaMemcpy(h_result, d_current, size, cudaMemcpyDeviceToHost);
        //print_grid(h_result, WIDTH, HEIGHT);
    }

    cudaEventRecord(stop);            // GPU timing stop
    cudaEventSynchronize(stop);       // Ensure stop event completed

    // gPU timing end
    auto gpu_end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> gpu_duration = gpu_end - gpu_start;

    // Calculate GPU time
    // Output timing
    std::cout << "GPU kernel time: " << gpu_duration.count() << " ms\n";
```

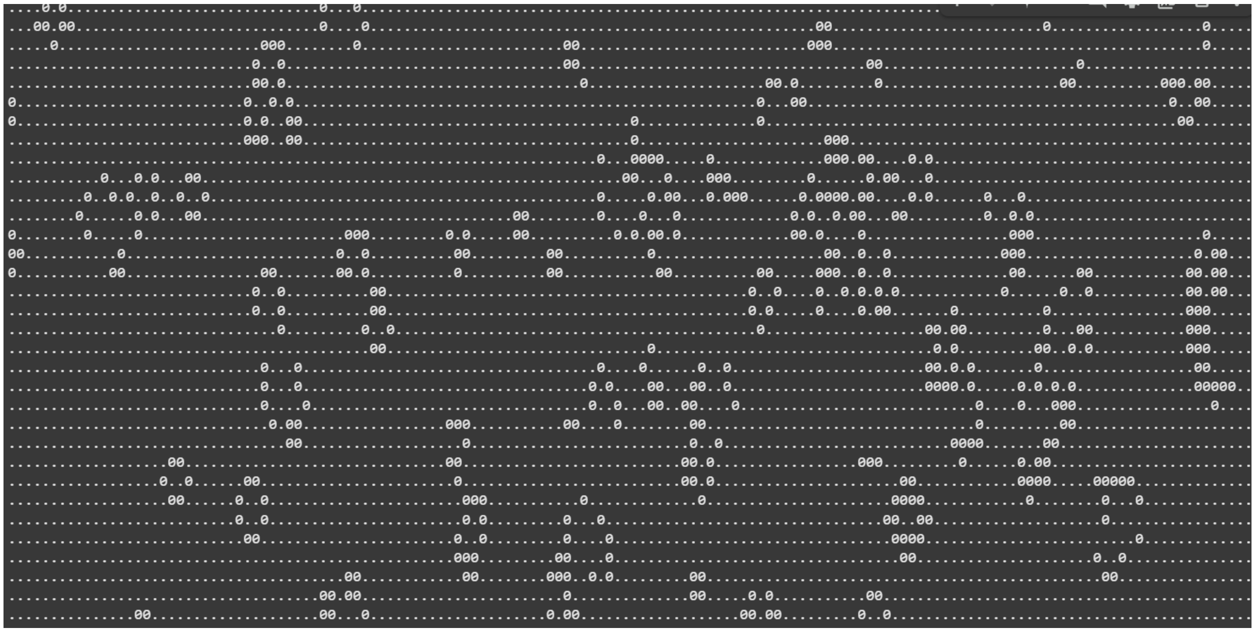
The CPU Code

```
[ ]  %%writefile buu_cpu.cpp
     #include <iostream>
     #include <chrono>
     #include <random>

     #define WIDTH 512
     #define HEIGHT 512
     #define ITERATIONS 1000



     void print_grid(int* grid) {
         for (int y = 0; y < HEIGHT; ++y) {
             for (int x = 0; x < WIDTH; ++x) {
                 std::cout << (grid[y * WIDTH + x] ? '0' : '.');
             }
             std::cout << '\n';
         }
         std::cout << std::string(WIDTH, '=') << '\n';
     }



     void set_random_dense_pattern(int* grid, float fill_ratio = 0.6f) {
         std::random_device rd;
         std::mt19937 gen(rd());
         std::bernoulli_distribution d(fill_ratio);

         for (int y = 0; y < HEIGHT; ++y) {
             for (int x = 0; x < WIDTH; ++x) {
                 grid[y * WIDTH + x] = d(gen) ? 1 : 0;
             }
         }
     }
```

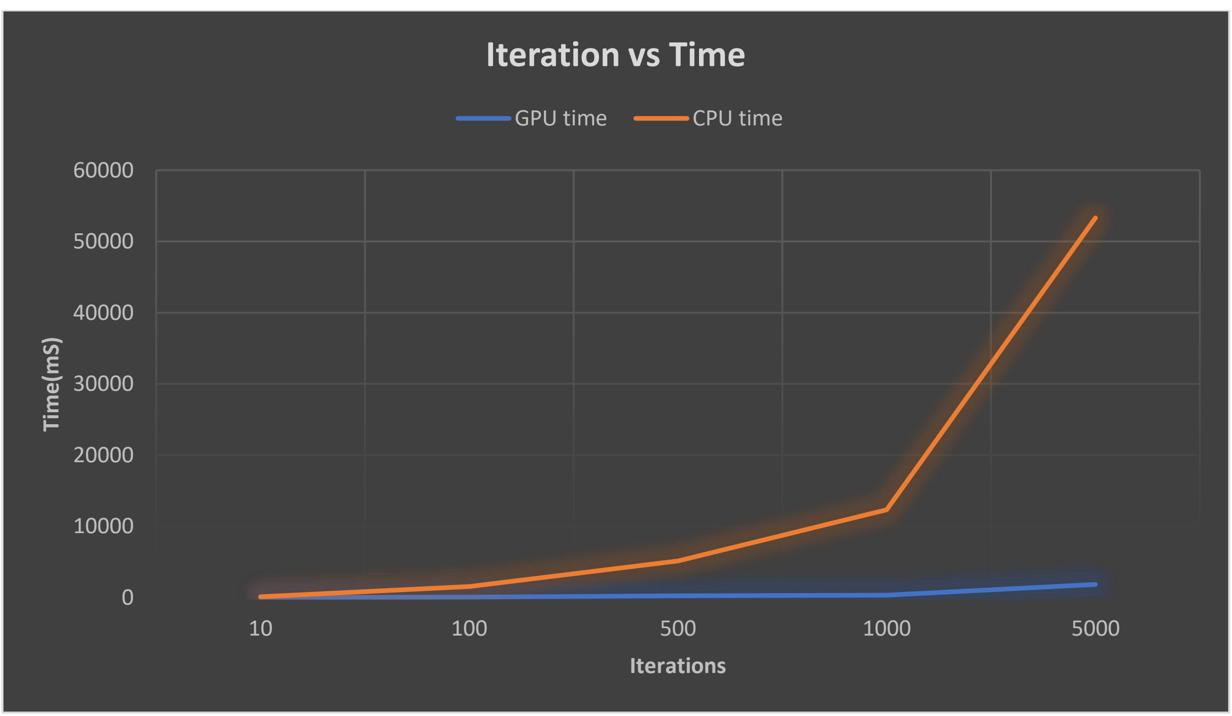
```
[ ]  int main() {
        int size = WIDTH * HEIGHT * sizeof(int);

        int* current = new int[WIDTH * HEIGHT]();
        int* next = new int[WIDTH * HEIGHT]();

        // Fill about 60% of the screen with live cells
        set_random_dense_pattern(current, 0.6f);



        auto cpu_start = std::chrono::high_resolution_clock::now();

        for (int i = 0; i < ITERATIONS; ++i) {
          for(int y = 0; y<HEIGHT; y++) {
            for(int x = 0; x<WIDTH; x++) {

              int idx = y * WIDTH + x;

              int neighbors = 0;
              for (int dx = -1; dx <= 1; ++dx) {
                  for (int dy = -1; dy <= 1; ++dy) {
                      if (dx == 0 && dy == 0) continue;
                      int nx = (x + dx + WIDTH) % WIDTH;
                      int ny = (y + dy + HEIGHT) % HEIGHT;
                      neighbors += current[ny * WIDTH + nx];
                  }
              }


            int state = current[idx];
            if (state == 1 && (neighbors == 2 || neighbors == 3)) {
                next[idx] = 1;
            } else if (state == 0 && neighbors == 3) {
                next[idx] = 1;
            } else {
                next[idx] = 0;
            }
            }
          }
```

```
        int * temp = current;
        current = next;
        next = temp;

      }


      // CPU timing end
      auto cpu_end = std::chrono::high_resolution_clock::now();
      std::chrono::duration<double, std::milli> cpu_duration = cpu_end - cpu_start;



      // Output timing
      std::cout << "CPU total time: " << cpu_duration.count() << " ms" << std::endl;


      // Cleanup
      // cudaEventDestroy(start);
      // cudaEventDestroy(stop);
      // cudaFree(current);
      // cudaFree(next);
      delete[] current;
      delete[] next;

      return 0;
  }
```

Animation snippet:



Results:



In the above test, we keep the grid size same at 512x512 and run the code for several iterations to compare the results.

We see that the GPU execution is **37** times faster than the CPU execution time (1000 iterations). In general for more iterations the GPU performs exponentially better when compared to the CPU.

References

1. Conways game of life-wikipedia
2. Numberphile-GOF
3. Computerphile-CUDA
4. Intro to CUDA
5. Nvidia-CUDA