Conways Game of Life parrallised on the GPU with CUDA

CAB 401 - Highly Performant Computing

Anhad Ahuja

October 2020

Abstract

This assignment is majorly thanks to the workings of Mathematician John Horton Conway who recently, at the age of 82 passed away due to Covid-19.
Specs:

- CPU: AMD Ryzen 5 2600 Six-Core Processor (Base speed 3.4GHz)
- GPU: GALAX NVIDIA GeForce GTX 1050Ti
- SSD: SATA Samsung SSD 860 EVO 500GB
- RAM: DDR4 16GB 2666MHz

1 Introduction

Cellular automata are types of descriptions of how a grid which has state should behave based on a consistent set of rules that apply to each cell. Conway's Game of Life is a famous cellular automaton that has the following rules:

- Any live cell with fewer than two live neighbours dies, as if by underpopulation.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

(Sourced from https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life) See more at (read from Sci-Hub):

- https://mathworld.wolfram.com/CellularAutomaton.html#:~:text=A%20cellular%20automaton%20is%20a,many%20time%20steps%20as%20desired.
- doi:10.1103/RevModPhys.55.601
- doi:10.1038/scientificamerican1070-120

These rules make for a very simple sequential program that has the potential of making very pretty patterns that can be fun to observe.

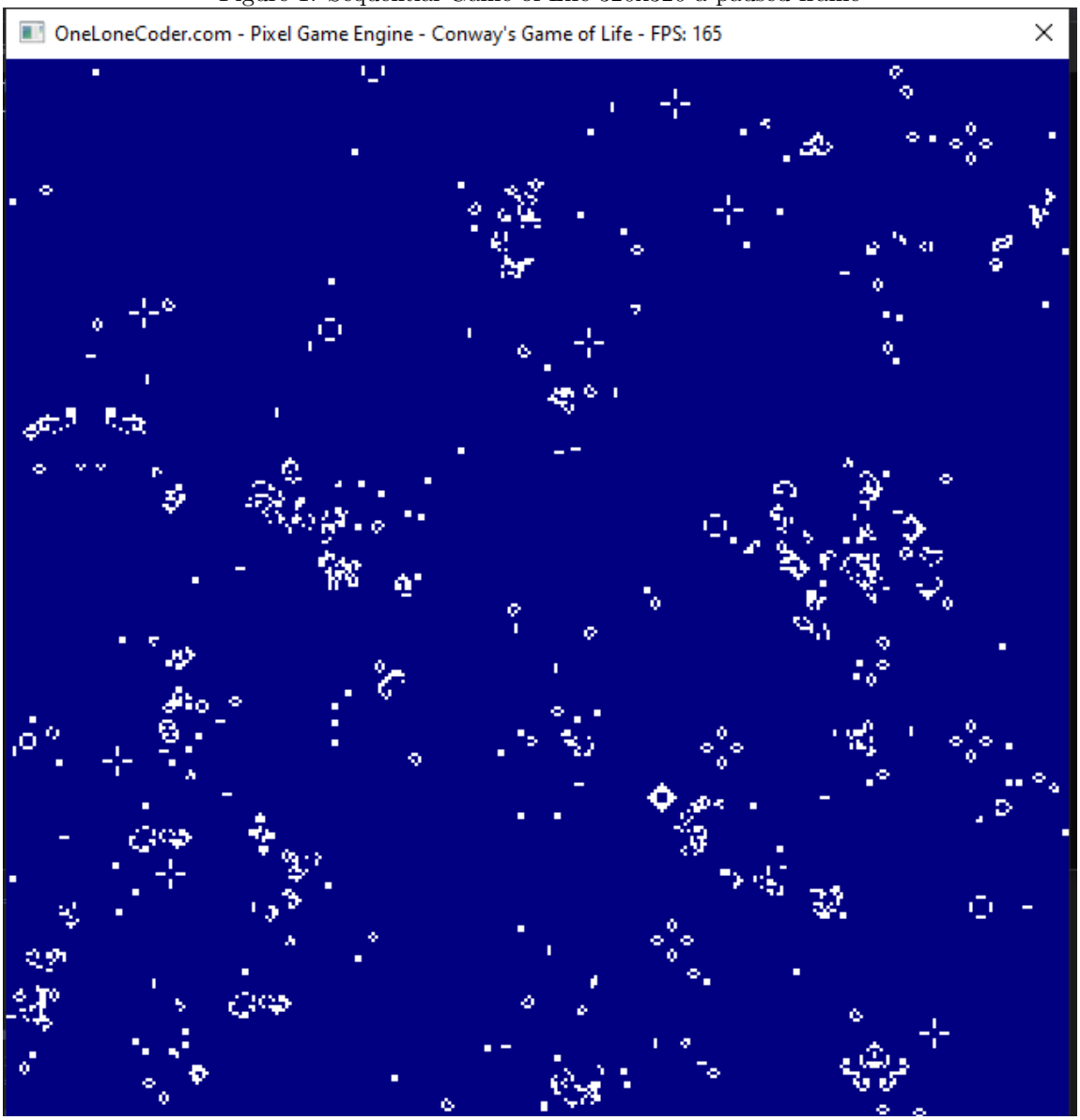
A very important distinction here is that each generation of every cell following those rules is entirely dependant on the history of the neighbours of the cell itself. Therefore the data dependancy is not across the grid but rather to the last generation. Optimizing this to be parallel on the GPU only seemed natural to do as a 2d grid based computation would be easy for CUDA to handle.

2 Sequential Implementation

```
void gameLoop(bool* data, int currentGen)
{
  for (int y = 0; y < N; y++)
  {
    for (int x = 0; x < N; x++)
    {
      //Treat generations (time) as the third dimension to array
      bool isAlive = data[((currentGen - 1) * N + y) * N + x];
      //An important note is to store repretetive access in local stack variable, compiler will place in register and avoids repeat memory access
      int totalNeighbours = 0;
      //Non modulous wraparound version of GOL due to parallel version
      for (int i = (y - 1 > 0) ? y - 1 : 0; i < ((y + 2 < N) ? y + 2 : N); i++)
        for (int j = (x - 1 > 0) ? x - 1 : 0; j < ((x + 2 < N) ? x + 2 : N); j++)
        {
          totalNeighbours += data[((currentGen - 1) * N + i) * N + j];
        }

      /*for (int i = y-1; i<y+2; i++)
        for (int j = x - 1; j < x + 2; j++) {
          if (i < 0 || i >= N || j < 0 || j >= N) continue;
          totalNeighbours += data[((currentGen - 1) * N + i) * N + j];
        }*/

        //Do not count self
      totalNeighbours -= isAlive;
      bool currentStatus = (isAlive && !(totalNeighbours < 2 || totalNeighbours > 3)) || (totalNeighbours == 3);
      data[(currentGen * N + y) * N + x] = currentStatus;
    }
  }
}


for (; gen < generations; gen++)
  {
    gameLoop(host_A, gen);
  }
```

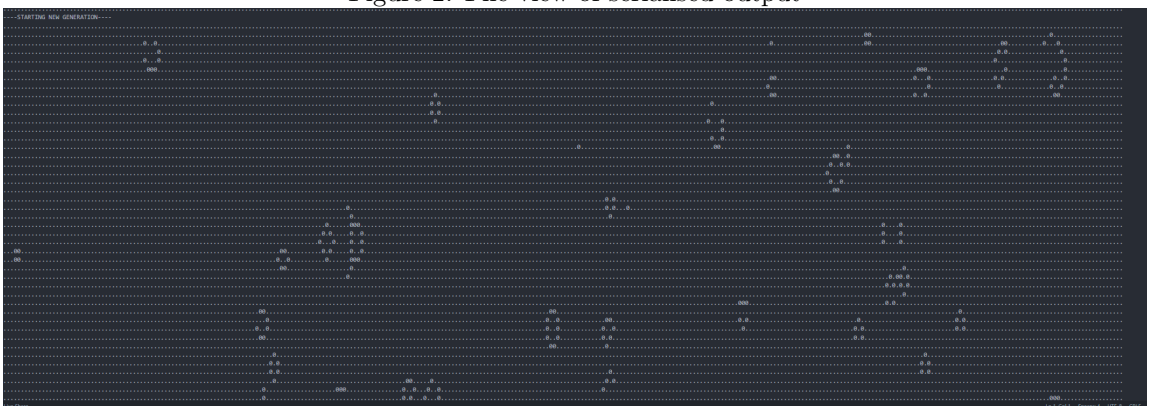
The methodology for approaching this basically was to treat each generation or iteration of computation as a 3D dimension. This is so that every generation could be stored in a large buffer to render at the very end of the computation. This was done so that the render pipeline does not disturb or affect the computation. For the time being an external single header library was used to draw and visualise the result called olcPixelGameEngine.
The result of this was:

Figure 1: Sequential Game of Life 320x320 a paused frame



I also wanted a way to analyse the result of the large Buffer and compare it, therefore I created a serealiser that places the contents of the buffer into a Human readable format file. The dots indicate an empty space and 0 indicates a live cell.

Figure 2: File view of serialised output



3 Parallel Implementation

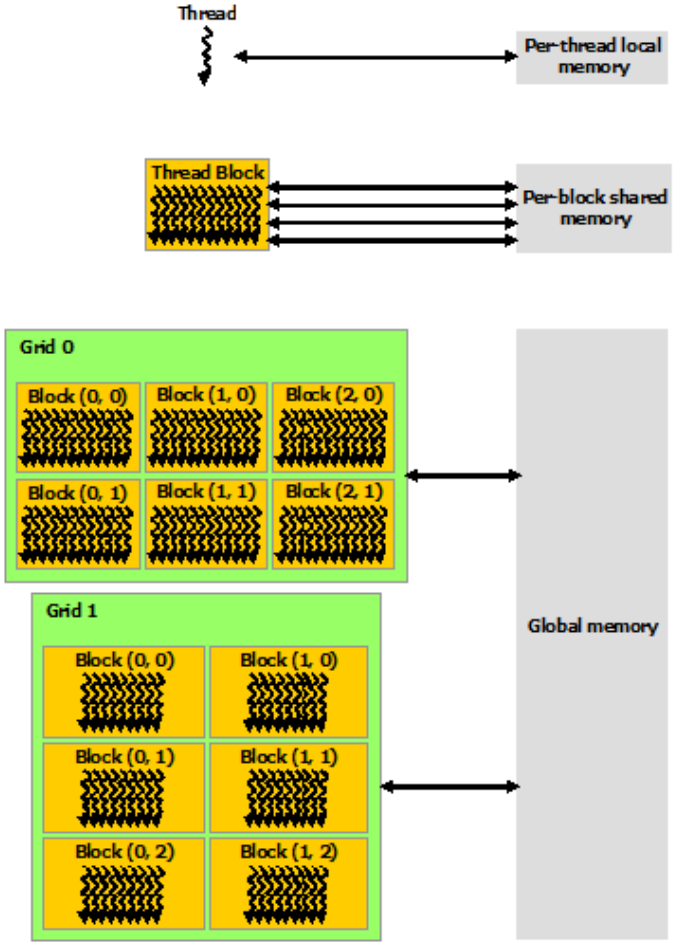
To start CUDA, the API dictates that you get a device, and create a buffer in the GPU space or the GPU global memory for the kernel to operate on. Then, In order to use CUDA, we had to look at where the for loops occurred and place the inner parts of those loops in a kernel call.

One brilliant tool that set apart CUDA from OpenCL was the ability to use printf() inside of the kernel function. This allowed premium debugging that allows us to see exaclty what the results were at which cell.

```
__global__ void CUDAgameLoop(bool* data, int currentGen)
{
  //Indexing similiar to 2d flattening
  //Treating blocks as 2nd dim (y), and thread as 1st dim (x)
  const int y = blockIdx.y * blockDim.y + threadIdx.y;
  const int x = blockIdx.x * blockDim.x + threadIdx.x;
  if (y < N && x < N) {
    bool isAlive = data[((currentGen - 1) * N + y) * N + x];
    unsigned int totalNeighbours = 0;
    for (int i = (y - 1 > 0) ? y - 1 : 0; i < ((y + 2 < N) ? y + 2 : N); i++)
      for (int j = (x - 1 > 0) ? x - 1 : 0; j < ((x + 2 < N) ? x + 2 : N); j++)
      {
        totalNeighbours += data[((currentGen - 1) * N + i) * N + j];
      }
    //Attempt manual looping in cardinal directions to test speed
    totalNeighbours -= isAlive;
    bool currentStatus = (isAlive && !(totalNeighbours < 2 || totalNeighbours > 3)) || (totalNeighbours == 3);

    data[(currentGen * N + y) * N + x] = currentStatus;
    //data[((currentGen * N + y) * N + x) / warpSize] = currentStatus; //For bit packing indexing
  }
}
```

Then the block location and the thread within the block could be used to determine the index of where that kernel was running and which cell that piece of code would be responsible for. As the following visual indicates, Each grid contains a lot of blocks and each block contains a set amount of threads. How many blocks is dictated by the size of data divided by how many threads there are in a block.

Figure 3: Visualisation of how a GPU scales




```
void runKernel(int gen, bool* device_A)
{
  dim3 blockSize(BLOCKSIZE, BLOCKSIZE);
  dim3 gridSize(N / BLOCKSIZE, N / BLOCKSIZE);
  //#pragma warning disable E0029
  CUDAgameLoop << <gridSize, blockSize >> > (device_A, gen);
}
```

4 Initial testing

NOTE: All tests were conducted in Release mode as to limit variance and compare optimal performance

4.1 BlockSize testing

The first parameter I had to attempt to configure was the best Blocksize. The results of my testing here were:

Figure 4: Testing BlockSize changes on 2048

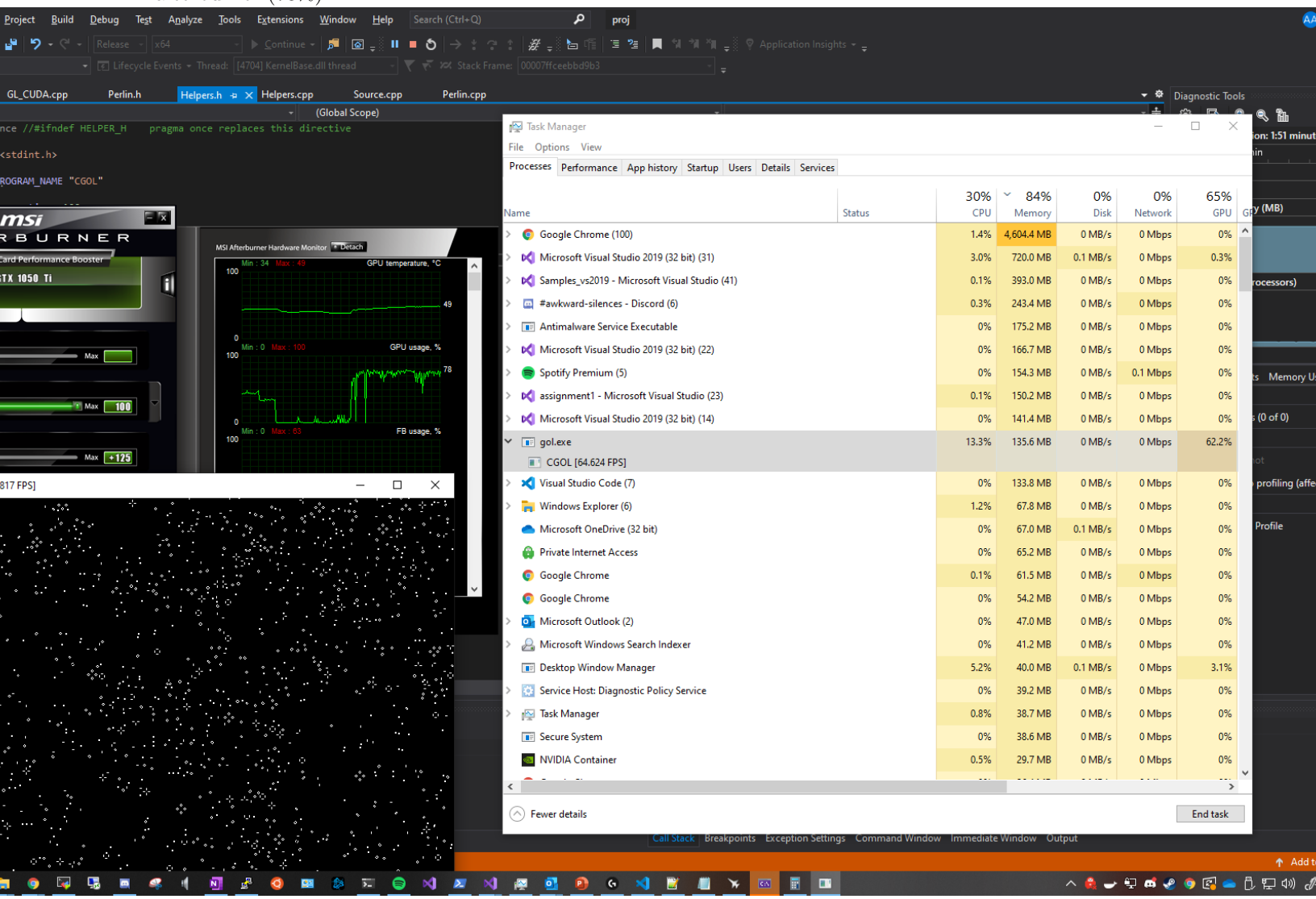
BLOCKSIZE	AVGTime
2	1.137762
4	0.361564
16	0.17543
32	0.173229

I found that it was not possible to go past 32 threads per block and this made sense as the limit to my GPU's archetecture would deem that so.

4.2 Overclocking GPU using MSI Afterburner

In order to see how much influence the GPU clock speed and memory clock speed had on the actual performance, MSI Afterburner https://www.msi.com/page/afterburner was used to perform such overclocks.It also provided a more reliable representation of the GPU usage than the windows task manager. Where windows would report a lower GPU usage than MSI often when playing heavy 3d games and this is both anecdotal and confirmed on multiple forums including https://linustechtips.com.It can also be seen in the following figure:

Figure 5: Running OpenGl variant (4096x4096) and comparing windows Task manager(65%) and MSI afterburner (78%)



It seemed that I was able to arrive at an Overclock of +300MHz Core Clock before my system went unstable. And it seemed that increasing the memory clock did not benefit in most cases, however, my sample size for testing this was very small. In the end I decided to go with a +150 MHz base clock increase for most of my tests as I did not want to tempt the stability of my system too much.

Figure 6: Testing OC changes on 4096

OC (+MHz)	Time
Stock	0.70186
100CORE	0.713576
150CORE	0.692741
200CORE	0.686869
250CORE	0.721297
300CORE	0.682287
50MEM	0.759627

5 Perlin Noise

5.1 Sequential

Perlin noise is a method of procedural generation that uses linear interpolation to make random noise more appealing and fluid between cells. I used this algorithm in attempt to seed the initial state of the cells so that the data was not totally random and so that it had some patterns to it.

I converted the Java Perlin static functions (https://mrl.nyu.edu/ perlin/noise/)written by Ken Perlin himself to c++ code.

There are 2 methods to seed Perlin Noise (through permutation mutation according to seed(preferred) or using Perlin's permutaion Matrix and placing seed in z value). I opted for the second and the seed here dictates the shuffle order of Ken Perlin's provided permutation array;

The p matrix declared on global state is an array of permutation that is based on the input seed value. This array dictates the gradient for the linear interpolation, in other words influences the random.

```
int p[512];

void initPerlin(int seed)
{
  int permutation[] = { 151, 160, 137, 91, 90, 15,
              131, 13, 201, 95, 96, 53, 194, 233, 7, 225, 140, 36, 103, 30, 69, 142, 8, 99, 37, 240, 21, 10, 23,
              190, 6, 148, 247, 120, 234, 75, 0, 26, 197, 62, 94, 252, 219, 203, 117, 35, 11, 32, 57, 177, 33,
              88, 237, 149, 56, 87, 174, 20, 125, 136, 171, 168, 68, 175, 74, 165, 71, 134, 139, 48, 27, 166,
              77, 146, 158, 231, 83, 111, 229, 122, 60, 211, 133, 230, 220, 105, 92, 41, 55, 46, 245, 40, 244,
              102, 143, 54, 65, 25, 63, 161, 1, 216, 80, 73, 209, 76, 132, 187, 208, 89, 18, 169, 200, 196,
              135, 130, 116, 188, 159, 86, 164, 100, 109, 198, 173, 186, 3, 64, 52, 217, 226, 250, 124, 123,
              5, 202, 38, 147, 118, 126, 255, 82, 85, 212, 207, 206, 59, 227, 47, 16, 58, 17, 182, 189, 28, 42,
              223, 183, 170, 213, 119, 248, 152, 2, 44, 154, 163, 70, 221, 153, 101, 155, 167, 43, 172, 9,
              129, 22, 39, 253, 19, 98, 108, 110, 79, 113, 224, 232, 178, 185, 112, 104, 218, 246, 97, 228,
              251, 34, 242, 193, 238, 210, 144, 12, 191, 179, 162, 241, 81, 51, 145, 235, 249, 14, 239, 107,
              49, 192, 214, 31, 181, 199, 106, 157, 184, 84, 204, 176, 115, 121, 50, 45, 127, 4, 150, 254,
              138, 236, 205, 93, 222, 114, 67, 29, 24, 72, 243, 141, 128, 195, 78, 66, 215, 61, 156, 180 };
  //shuffle Needs end +1
  if (seed != -1)
  {
    std::shuffle(&permutation[0], &permutation[256], std::default_random_engine(seed));
  }
  for (int i = 0; i < 256; i++)
    p[256 + i] = p[i] = permutation[i];
}

//In Source.cpp to call the noise function:

initPerlin(SEED);
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
    {
      host_A[(gen * N + i) * N + j] = noise((double)j * OFFSET_SCALE, (double)i * OFFSET_SCALE, 0) <= INCLINITATION;
    }
```

5.2 Parallel

It is beneficial to generate the Perlin noise in the CUDA buffer itself instead of creating a new CUDA buffer for the generation, computing, transfer the result to host and transferring back to device due to the heavy IO operations implied with memory transfer.

The special constant prefix in CUDA dictates that the following memory will remain constant for the duration of the kernel running, This allows the GPU to treat it effectively as if it were part of the kernel itself. Of course it can be dynamically set from the host however.

Another benefit that was found out of using CUDA was the ease of use of declaring whether a function would be run on device side or host side, a prefix of `__device__` or `__host__` made this very apparent to the compiler.

This also meant you could abstract the sequential logic to functions to make code more legible instead of writing inline

In the end the result of the parallel and the sequential implementation was the same as seen in the following figure:

Figure 7: Left(Sequential Generation) and Right(Parallel Generation) of Perlin Noise are identical

Since this code is essentially the most direct measurement of serial CPU vs GPU since there is only One kernel call being run to generate the Perlin Noise, It was only fair to run a single generation that can be used to determine how much faster the GPU is in its own domain.

For generating a piece of Data for Perlin Noise of the size: 8192, it takes the GPU 0.00027 seconds seconds whereas the CPU takes 3.05544 seconds which is a speedup of 11316 times.

6 Bit-Packing

Bit packing is the process of utilising the the bits inside of a byte to store a boolean instead of using 1 whole bit per boolean. eg:
`{00000000, 00000001, 00000001, 00000000, 00000001, 00000001, 00000001, 00000000}` $\rightarrow$ `01011110`

In an unsigned integer on 64 bit machines, an unsigned is 32 bits. One important note with bit packing however is that the size of the GOL matrix or the N value had to be a multiple of 32. Therefore to store 32 booleans the following code had to be employed:

6.1 Sequential

```
void BPgameLoop(uint32_t* data, int currentGen)
{
  for (int y = 0; y < N; y++)
  {
    for (int x = 0; x < N; x++)
    {
      uint32_t cell = data[(((currentGen - 1) * N + y) * N + x )/ BP];
      bool isAlive = (cell & (1 << (x % 32))) != 0;
      int totalNeighbours = 0;
      for (int i = (y - 1 > 0) ? y - 1 : 0; i < ((y + 2 < N) ? y + 2 : N); i++)
        for (int j = (x - 1 > 0) ? x - 1 : 0; j < ((x + 2 < N) ? x + 2 : N); j++)
```

```
        {
          uint32_t cell = data[(((currentGen - 1) * N + i) * N + j) / BP];
          totalNeighbours += (cell & (1 << (j % 32))) != 0;
        }
      totalNeighbours -= isAlive;
      bool currentStatus = (isAlive && !(totalNeighbours < 2 || totalNeighbours > 3)) || (totalNeighbours == 3);

      //This is fine to do because array is initialised to 0 (not reusing array)
      data[((currentGen * N + y) * N + x )/ BP] |= (currentStatus << (x % 32));
    }
  }
}
```

6.2 Parallel

```
__global__ void BPKernel(uint32_t* data, int currentGen)
{
  //Only works if BLOCKSIZE is 32
  __shared__ bool packing[32];
  const uint16_t tid = threadIdx.x;

  const int y = blockIdx.y * blockDim.y + threadIdx.y;
  const int x = blockIdx.x * blockDim.x + tid;

  uint32_t cell = data[(((currentGen - 1) * N + y) * N + x) / BP];
  bool isAlive = (cell & (1 << (x % 32))) != 0;
  int totalNeighbours = 0;
  for (int i = (y - 1 > 0) ? y - 1 : 0; i < ((y + 2 < N) ? y + 2 : N); i++)
    for (int j = (x - 1 > 0) ? x - 1 : 0; j < ((x + 2 < N) ? x + 2 : N); j++)
    {
      uint32_t cell = data[(((currentGen - 1) * N + i) * N + j) / BP];
      totalNeighbours += (cell & (1 << (j % 32))) != 0;
    }
  totalNeighbours -= isAlive;
  bool currentStatus = (isAlive && !(totalNeighbours < 2 || totalNeighbours > 3)) || (totalNeighbours == 3);

  packing[tid] = currentStatus;

  printf("x, y: %d, %d | status: %d\n", x, y, currentStatus);
  //HERE all threads in block sync to submit the mask to the final Buffer
  data[((currentGen * N + y) * N + x) / BP] = __ballot_sync(FULL_MASK, packing[tid]);
  //https://stackoverflow.com/questions/39488441/how-to-pack-bits-efficiently-in-cuda/39488714#39488714
  printf("afterwards: %d \n", data[((currentGen * N + y) * N + x) / BP]);
}
```

One major issue with directly parallelising the Bit packing program is that each byte or unsinged int in this case is going to be accessed across multiple treads. As an intrinsic data form, there is no implicit barrier such as what teh kernel has set up for each thread to access each bit individually. The issue arises here. A solution to this is the ballot warp intrinsic. This method basically synchronises all of the threads in a block and generates a mask based on a conditional or predicate. that predicate in this case can be the next state of the cell.

At the end of it all, using the normal way of bit-wise Or'ing resulted in distorted results however, the ballot method translated into exactly the same data as the sequential. This test was conducted with 320x320 matrix for 10 generations.

7 Final results

These tests were all ran for 100 generations so as to keep the overall size of the buffer small.

Figure 8: Testing OC changes on 4096

Size	Sequential Time	Parallel time
256	0.090785	0.0046808
512	0.358401	0.012533
1024	1.44107	0.0507667
2048	5.85612	0.192726
4096	23.1983	0.733965

Overall the piece of code that possibly contributed the most to the GPU version was actually allocated and deallocated the memory as well as copying the Memory back to host. This is considered an expensive operation.

8 OpenGL interop and realtime Rendering

One primary reason for the switch from OpenCL to CUDA was the support for the OpenGL interoperability. Instead of utilising a very large buffer that uses 3rd Dimension indexing for the generations.

The way that the interop works is that the cuda code takes takes control of the Pixel Buffer Object, does its operations and then releases it for the openGL code to use. Instead of transferring the Pixel data through the CPU memory or treating the CPU as an intermediary, the reason why this method is fast is because the memory always stays on the GPU, and never has to be transferred to the Host.

```
void render(cudaGraphicsResource* cuda_pbo_resource, bool* prevGen, bool* currGen)
{
  uchar4* d_out = 0;
  cudaGraphicsMapResources(1, &cuda_pbo_resource, 0);
  cudaGraphicsResourceGetMappedPointer((void**)&d_out, NULL, cuda_pbo_resource);
  dim3 blockSize(BLOCKSIZE, BLOCKSIZE);
  dim3 gridSize(N / BLOCKSIZE, N / BLOCKSIZE);
  CUDA_GL_gameLoop << <gridSize, blockSize >> > (prevGen, currGen, d_out);
  cudaDeviceSynchronize();
  cudaGraphicsUnmapResources(1, &cuda_pbo_resource, 0);
}
```

Due to the render fucntion being called many times a second, it would be unfeasible to utilise the long history buffer method to retrieve the last generation's information. Instead it is possible to use Alternating buffers to cycle through buffer B1 and B2 based on if the generation is odd or even. In true c++ fashion, it is better to create 1 large buffer instead of two and use pointer arithmatic to swap between the 2.

```
bool* A = (bool*)malloc(N * N * 2);
...

///* Loop until the user closes the window */
  while (!glfwWindowShouldClose(window))
  {
    if (!showFPS()) {
      //std::cout << "skipping";
      continue;
    }
    bool isOdd = !(gen % 2);
    //std::cout << "Is odd: " << sizeofSingle * (!isOdd) << "Generation: " << gen << '\n';

    /* Render here */
    glClear(GL_COLOR_BUFFER_BIT);
    //If even (0) then B1 will be previous and B2 will be next,
#ifdef PARALLEL
    render(cuda_pbo_resource, &device_A[sizeofSingle * isOdd], &device_A[sizeofSingle * (!isOdd)]);
#else
    seqRender();
#endif // PARALLEL
    drawTexture();
```

A very premature thought process would be to reset the last gen's buffer to 0 after the data is consumed and the next gen is generated already for that particular index. However it is important to note that the data dependancy exists for the neighbours as well which rely on the information from the last gen. Therefore carefully formulated code had to be made.

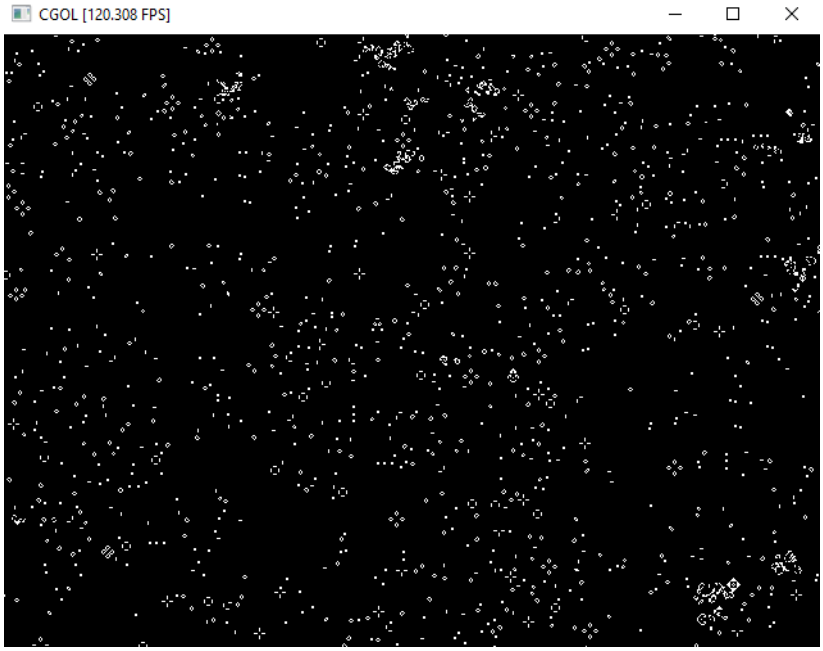
Since the kernel computes a value for EVERY x,y coordinate. the reassignment of currentStatus would occur regardless. Therefore leaking of info by reusing the buffer would not occur.

The results of the code was buttery smooth rendering and maxing out my GPU's utilisation in MSI afterburner:

8.1 Installation and boilerplate

The installation for opengl was the fairly standard unpack, make environment varibales and load linkers in visual studio process. We had to do this for glfw which is source for the opengl 1.1 code and glew which retrieves gpu driver code and links it for more recent opengl functions.

Figure 9: Rendering OpenGl and CUDA



8.2 Alternative methods

There are compute shaders as well in opengl which would be far more efficient and essecitnailly they would handle the computation that CUDA does and do so in a similiar manner that openCL does.

9 Parallel File operations

By Far the slowest part of my program is the part the serialises the array to file. Right now, my file format is incredibly bulky, storing as strings (so as to make visualisation and comparison between versions easy) instead of direct bytes. Nevertheless the slowest IOP for any computer is going to be accessing a system OS disk, HDD or SSD. Therefore I had the idea of creating multiple File handlers in the program and using threading to write to File. The ideal way of approaching this would be to write to chunks instead of one contiguous file so that there are little race conditions. With a hard dive this would be redundant as they are bottle necked by the head and platter.

To my disappointment I found that the story is very similiar with SSD as well. Due to ssd's having one controller, the bottleneck of one Bus to the ssd still stands. The only thing threading may have assisted with is getting priority to write to file from the OS. However a new interface for SSD's (NVME) does not require such a lock as seen by this table found on this article https://sata-io.org/sites/default/files/documents/NVMe%20and%20AHCI%20as%20SATA%20Express%20Interface%20Options%20-%20Whitepaper_.pdf

Figure 10: Running OpenGl variant (4096x4096) and comparing windows Task manager(65%) and MSI afterburner (78%)

	AHCI	NVMe
Maximum Queue Depth	1 command queue 32 commands per Q	64K queues 64K Commands per Q
Un-cacheable register accesses (2K cycles each)	6 per non-queued command 9 per queued command	2 per command
MXI-X and Interrupt Steering	Single interrupt; no steering	2K MSI-X interrupts
Parallelism & Multiple Threads	Requires synchronization lock to issue command	No locking
Efficiency for 4KB Commands	Command parameters require two serialized host DRAM fetches	Command parameters in one 64B fetch

As well as that a new form of SSD technology found on Sony's PS5 has a IO throughput of 5.5GB/s and in order to maximize value from this it may be impertenent to look into parallel serialisation for games (https://www.eurogamer.net/articles/digitalfoundry-2020-playstation-5-specs-and-tech-that-deliver
Unfortunately I do not have access to an NVME storage device, However given the opportunity, I would readily test out the File operations on an NVME drive. Given the current trajectory in faster hardware, it seems that more and more games/ programs will have to utilise efficient and parallel code to take full advantage of the write speeds.

10 Major setbacks

At first, for experimentation there was a cluttered sense of implementing everything at the same time. As the project progressed, it seemed ideal to segregate the OpenGL interop with all other projects due to the fact that testing others and recording that seemed difficult in the openGL event loop

Installing CUDA was not an easy task, especially with Visual studio: https://forums.developer.nvidia.com/t/cannot-run-samples-on-ms-visual-studi-2019/72472

Visual Studio was often an issue in this case as well. For example there was an occasion in error handling where the newest c++ 20 spec wouldve been very helpful. This was in the case of using the new std::format method to use template python like strings with the std::string class. However, Visual studio

10.1 Language Choice

C++ is a very difficult language to wrangle with, a lot of paradigms are very unorthodox for an OOP developer, and dealing with libraries is quite often the BIGGEST pain for any developer as c++ has many ways to approach a problem and so libraries' implementation of a method may not be as straightforward as they first seem. OpenGL was a particularly big one. Hence it was not really the parallelisation that I struggled with in this language, rather the creation of the GOL to begin with. However one thing c++ does well is manual control over pointers and the efficiency in cod you can squeeze out of this. This is what also allows CUDA to exist in this platform due to manual control over GPU pointers and arrays. This is opposed to a language like javascript where one cannot achieve parallelisation but is able to complete what they set out to do very quickly.

11 Aspirations and Stretch Goals

11.1 Golly Open source GOL

Another aspiration that was found in this journey of GOL parallelism, was in the open source project Golly which is a GUI/cmd program that emulates and contains all kinds of utilities for mostly Game of Life but also for other automata. The primary appeal of this prorgam is that it uses very efficient algorithms as the backend for generating the generations.

bGolly: bgolly -m 10 -a HashLife -v -o out.mc Patterns-lines.mc

By using the Profiling tool: we can see where the algorithm is most computationally intensive. These hotspots can be parallelised using the CPU. As they are memoized algorithms, it would be difficult to do so without running into data dependancies as memoized algorithms such as the HashLife implementation, generally require a mutex around a data structure that stores commonly seen shapes and patterns to predict future generations.

11.2 Better hardware

I had specced my PC primarily to play non Graphics intensive games therefore my CPU was a lot more powerful than my GPU. These tests would be very interesting on more Modern Hardware such as the recently released RTX 30XX series. As well as that, the tests I had done were limited in size of Matrix due to the

host memory running out for displaying the actual computed result. I didn't use the bitpacked version as I had just come up with the system and didn't have time to run tests.

11.3 Hexagonal GOL

This concept was brought up and explored through research however, as research progressed it was deemed more of an issue of actually implementing the sequential version and less of parallelisation issue. The data dependencies were very similiar and to perceive any interesting results, a more integer based cellular automata would have to be used. Nevertheless, an openGL parallel sped up version of the hexagonal celullar automata for wildfire spread would be an interesting simulation

A simple stretch goal of this project was to be able to create an Interface or an API to abstract the parralel logic away for any cellular automata so that all that would need to be provided is the homogeonous rules that every cell has to follow.

A very tricky and exhausting part of integrating the openGL library, especially for nvidea, was the setup. The first step was to install both the geforce experience, and the cuda toolkit.
This tutorial was very helpful in this progress.

Then setting the necessary environment variables.

12 Conclusion

It is my personal opinion that CUDA is one of the easiest ways to mass parallelise data generation ONCE the environment is there to do so. The overall esoteric nature of this library along with its proprietary code is what is limiting users to start coding parallel applications in it right now.

I realise this project is like lumping a bunch of algorithms at the c++ compiler and parralelising them with no real purpose however, I wanted to prove that there are a lot of components of video games (such as procedural generation in the form of Perlin Noise, and cellular automata in the form of Game of life) that can be very trivially parallelised.

It should be noted that the lack of using a profiler tool was intentional, as this was my own program, I could identify the obvious locations where the hotspots were located.

The primary resources for OpenGL and CUDA were the Samples that were included with the installlation of Cuda developer SDK As well as these resources:

- https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- https://en.cppreference.com/w/cpp/language/
- https://www.informit.com/articles/article.aspx?p=2455391seqNum=2

As a final note, this project was very entertaining but challenging to code from scratch.