Author: Ryan Knowles
Date: 4-16-2014
Course: MTH 5050 - Parallel Processing
Assignment: Final Project - Cellular Automata Parallelization on CUDA

Problem Statement:
I choose to study procedural generation techniques because they have a lot of flexibility and it is an area I want to explore. From among the many procedural generation techniques I found, I selected a cavern generation algorithm the uses cellular automata. Since cellular automata have an algorithm similar to the Jacobi Method, I knew that it would be easy to parallelize. Furthermore, the base algorithm was presented in 2D but I was easily able to extend it to 3D.

Test Setup:

System:

Software	Hardware
OS: Windows 8.1 Pro 64-bit	CPU: Intel i5-4670k Quad-Core @ 3.4 GHz
GPU Driver: Nvidia GeForce 335.23	CPU RAM: 32768 MB
CUDA Runtime: CUDA 5.5	GPU: EVGA Nvidia GeForce 660
	GPU Memory: 3072 MB

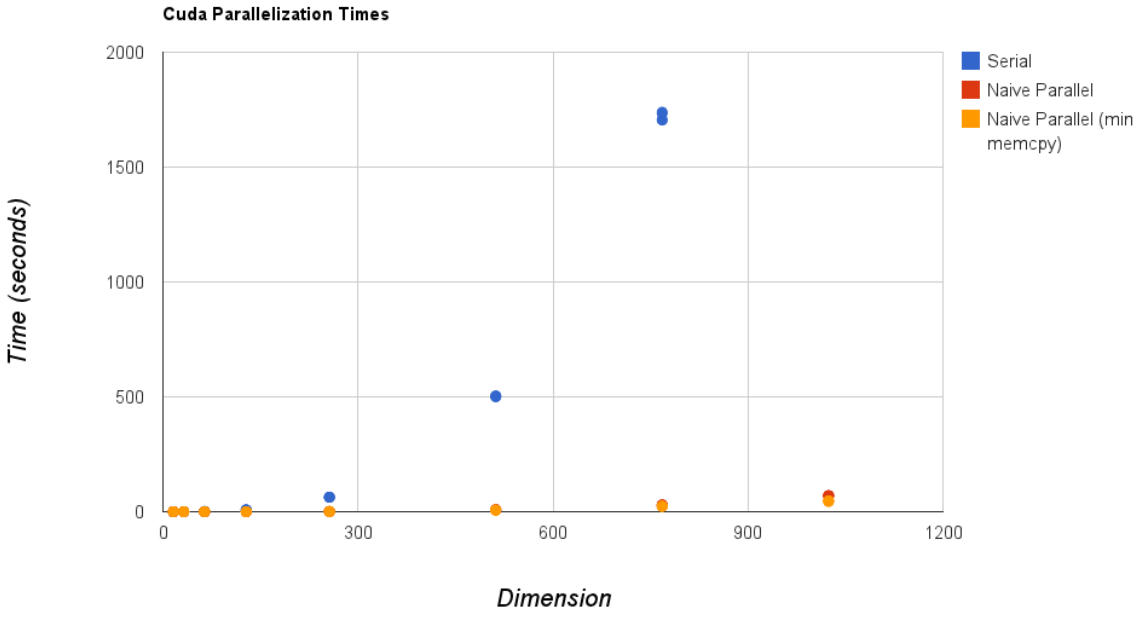
Algorithms:
1. Serial implementation to use as a basis.
2. Naive Parallel algorithm that implements no additional optimizations except for running all calculations on the GPU.
3. Second Naive Parallel algorithm that minimizes the amount of memory copies between the Host (CPU) and Device (GPU).

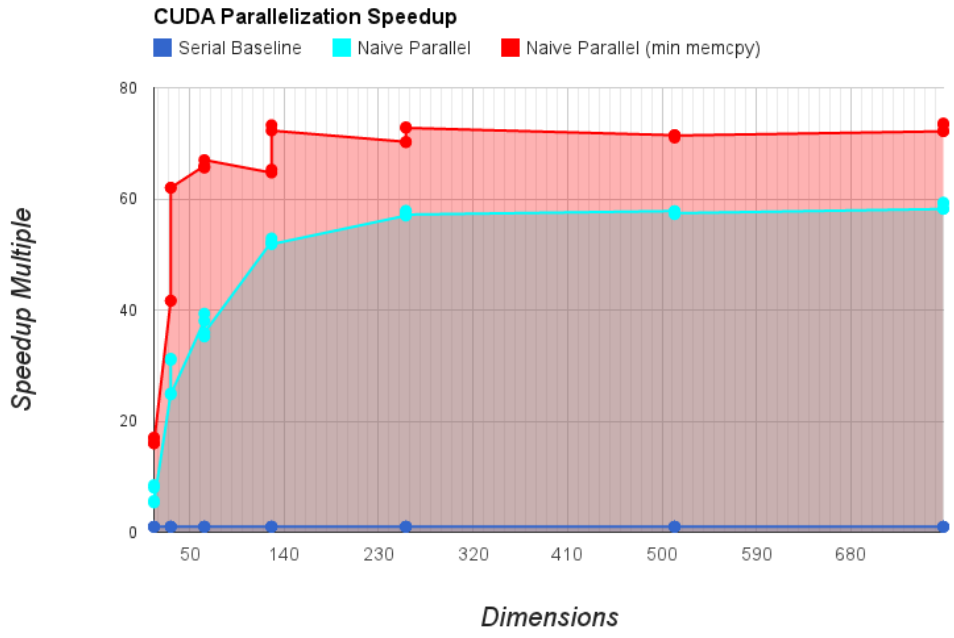
Test Cases:
Tests were run on 3D matrices of sizes 16^3, 32^3, 64^3, 128^3, 256^3, 512^3, 768^3, and 1024^3. I was not able to run the serial algorithm on the 1024^3 test case as it would crash with a runtime error. I was not able to find the reason for this, although I think it may have to do with taking up too much processing time. I performed some optimizations to the reference algorithm by:
- limiting array creation, deletion, and copying, and
- applying the principle of data coalescence to reduce page faults and memory reads in both the serial and parallel implementations.

Results:

Dimensions	Serial	Naive Parallel	Naive Parallel (min memcpy)
16	0.017	0.002	0.001
16	0.017	0.003	0.001
16	0.016	0.002	0.001
16	0.016	0.003	0.001
32	0.125	0.005	0.003
32	0.124	0.004	0.002
32	0.125	0.004	0.003
32	0.124	0.005	0.002
64	0.989	0.026	0.015
64	0.986	0.028	0.015
64	0.984	0.025	0.015
64	1.005	0.028	0.015
128	7.894	0.151	0.122
128	7.988	0.151	0.109
128	7.899	0.152	0.121
128	7.879	0.152	0.109
256	62.958	1.105	0.896
256	62.734	1.084	0.894
256	63.569	1.102	0.872
256	63.455	1.11	0.872
512	504.249	8.723	7.057
512	503.567	8.715	7.045
512	500.282	8.742	7.046
512	501.498	8.734	7.024
768	1704.958	29.308	23.629
768	1705.751	29.295	23.619
768	1733.845	29.278	23.628
768	1739.36	29.314	23.636
1024		69.677	44.049
1024		70.25	47.765
1024		70.224	45.935
1024		69.784	47.76

Dimensions	Serial Baseline	Naive Parallel	Naive Parallel (min memcpy)
16	1	8.5	17
16	1	5.66666666666667	17
16	1	8	16
16	1	5.33333333333333	16
32	1	25	41.6666666666667
32	1	31	62
32	1	31.25	41.6666666666667
32	1	24.8	62
64	1	38.0384615384615	65.9333333333333
64	1	35.2142857142857	65.7333333333333
64	1	39.36	65.6
64	1	35.8928571428571	67
128	1	52.2781456953642	64.7049180327869
128	1	52.9006622516556	73.2844036697248
128	1	51.9671052631579	65.2809917355372
128	1	51.8355263157895	72.2844036697248
256	1	56.9755656108597	70.265625
256	1	57.8726937269373	70.17225950783
256	1	57.6851179673321	72.9002293577982
256	1	57.1666666666667	72.769495412844
512	1	57.8068325117505	71.4537338812527
512	1	57.7816408491107	71.4786373314407
512	1	57.2274079158087	71.0022707919387
512	1	57.4190519807648	71.3977790432802
768	1	58.1738091988536	72.155317618181
768	1	58.2266939750811	72.2194419746814
768	1	59.2200628458228	73.3809463348569
768	1	59.3354711059562	73.589439837536


Cuda Parallelization Times
2000
1500
1000
500
0
0
300
600
900
1200
Time (seconds)
Dimension
Serial
Naive Parallel
Naive Parallel (min memcpy)


CUDA Parallelization Speedup
Serial Baseline
Naive Parallel
Naive Parallel (min memcpy)
80
60
40
20
0
50
140
230
320
410
500
590
680
Speedup Multiple
Dimensions

Discussion:

Result Analysis:
I expected the algorithm to have a decent speedup because it is innately parallel. The speedup is compared to the time the serial implementation took, which I refer to as the basis. I think the speedup demonstrated by the naive algorithm, approximately 60 times the basis, is very good for the time investment needed. By minimizing the usage of the cudaMemcpy() function the speedup increased to approximately 70 times the basis.

The graphs show that there is a limit to the usefulness of running a parallelized algorithm on the GPU. As documented by Nvidia, the GPU's computing cores are designed for throughput not latency. This means that any single computation is slower compared to a CPU (e.g. 1ms on CPU compared to 10 ms on GPU) performing the same computation. The benefit of the GPU is that it can run 10,000+ of these computations in parallel, so while the CPU may need 1ms * N time to compute on an array of numbers, the GPU may still only need 10ms to do the same computation. However, if the size of the array (N) is small enough, the CPU may still be faster, as seen in the smaller numbers on the charts.

Issues:
I encountered a variety of issues during this project and learned a great deal as a result. The most problematic issue I encountered was the TDR issue, where a watchdog process in Windows 8 was killing my program by rebooting the GPU if it took too long to respond. After many hours of trying to figure out the root cause, I found that editing a registry value was a possible fix for this. I think there are also some issues with driver compatibility between the CUDA 5.5 devtoolkit and my system, as I still have issues with playing audio and video from a web browser.

Another related issue to the one above was the GPU overheating. After research online, I found that my GPU had temperature issues due to an inadequate fan not providing the cooling needed. Under normal usage conditions, the GPU doesn't have an issue, even when playing some graphically complex 3D games, however when I ran my larger test case algorithms, I pushed my GPU to the max and saw the temperature spike to over 80^{o}C. I believe this could also have limited the maximum speedup seen on my GPU because as it got hotter it performed worse.

Finally, an interesting issue I came across was a race condition in my parallel algorithm code. I was attempting to synchronize all the threads and before another iteration inside the kernel code itself that was running on the GPU, however this introduced a race condition and data corruption. Because of how CUDA works, the threads running on the GPU are broken up into blocks and the syncthreads() function only waits until all threads on that block have finished before continuing. In order to synchronize all the threads across multiple blocks, I had to take the iteration loop out of the kernel code and put it back into the host (CPU) code (i.e. the main function).

Output Verification:

In order to ensure I was getting the correct output and my algorithm was running correctly, I performed a variety of different tests to ensure the validity of the test cases:

1. First I created a visualization to ensure that the serial algorithm was creating the desired result.
2. Next, for each test case in serial I analyzed the state of the matrix after each iteration, count the number of births, deaths, total alive, and total dead cells.
3. When running in parallel, I would run the program a few times and print out the same state information between iterations. After I was sure the algorithm was correct, I turned off the debug messages to ensure no interference with the run times.
    a. I also found that running the test with some time in between runs would allow me to easily see if I was reading from memory, as consequent runs tend to produce the same results, even when reading from memory. Letting some time pass between runs would generally mean that the same memory space was not being read or that the memory space's values had changed, resulting in different printed results.
4. After noticing a discrepancy in test times and comparing it to the temperature of the GPU, I tried to let the GPU cooldown to between 36°-40°C after each test.

Conclusions:

What I learned:

I learned a variety of different things while doing this project:

- Technology
    - Learned how to program for CUDA enabled devices.
    - Was very impressed by the Udacity course and will be using Udacity again to learn other subjects.
    - When running computations on large datasets, the hardware can generate a lot of heat and this can affect the runtime result or even cause system instability.
- Testing
    - After stumbling across some array index out of bounds and synchronization errors, I was forced to come up with some interesting ways to test my results and verify the correctness of my algorithms.
- Visualization & HTML5/Javascript
    - I spent more time than I should have creating the javascript code to visualize the serial output, however I think the results were worth it. When I attempted to optimize the visualization to allow me to display larger data sets, I was surprised at the result of trying to perform a memory space vs computation trade-off, resulting in my visualization eating up gigabytes of ram.

Improvements:
I would like to have made better optimized algorithms and test the speedup of different optimizations individually and combined. One optimization I read about was copying data from the device's global memory into shared memory. Unfortunately, due to time constraints and my limited knowledge of CUDA programming this was not possible within my time constraints. Also, it would have been better to run the parallel algorithms on a dedicated video card that is not also being used by the operating system as a display device.

Summary:
I think using the GPU to perform calculations can be very beneficial. However, as I experienced in my case, this is not recommended to run on the display device. I expect that in the future, motherboards will come equipped with a GPU spot, much like the CPU, or even have a GPU processor built in. Combine this with compilers that can detect parallelizable code sections, we could continue programming as normal while benefiting from the speedup boost provided by GPUs. Regardless, I think learning how to program on a GPU is a very good time investment and I am glad I chose this for a project.

Code:

- Serial:

```
#include <cstdio>
#include <cstdlib>
#include <ctime>

#define DO_STATS
#define M_SIZE              768

typedef struct {
       int births;
       int deaths;
       int alive;
       int dead;
} MapStats;

template <size_t xSize, size_t ySize, size_t zSize>
void initMap(unsigned char *oldMap, int mapSeed);
template <size_t xSize, size_t ySize, size_t zSize>
void iterateCA(unsigned char *oldMap, unsigned char *newMap);
template <size_t xSize, size_t ySize, size_t zSize>
int countNeighbours(unsigned char *map, int x, int y, int z);
template <size_t xSize, size_t ySize, size_t zSize>
void printJSON(unsigned char *map, int iter);
template <size_t xSize, size_t ySize, size_t zSize>
void countStats(unsigned char *oldMap, unsigned char *newMap, MapStats &stats);

//Globals -- bad code
const int deathLimit = 15;
const int birthLimit = 17;
```

```
int main()
{
	const int mapSeed = 45000;
	const int xSize = M_SIZE;
	const int ySize = M_SIZE;
	const int zSize = M_SIZE;
	const int maxIters = 30;
	const bool PRINT = false;
	const bool TIME = true;

	clock_t start, total = 0;
	unsigned char *oldMap = new unsigned char[xSize*ySize*zSize];
	unsigned char *newMap = new unsigned char[xSize*ySize*zSize];
	initMap<xSize,ySize,zSize>(oldMap, mapSeed);
	MapStats stats;

	unsigned char *temp;
	if(PRINT)
	{
		printf("{\n");
		printf("\"mapSeed\":%d,\n", mapSeed);
		printf("\"deathLimit\":%d,\n", deathLimit);
		printf("\"birthLimit\":%d,\n", birthLimit);
		printf("\"xSize\":%d,\n", xSize);
		printf("\"ySize\":%d,\n", ySize);
		printf("\"zSize\":%d,\n", zSize);
		printf("\"maxIters\":%d,\n", maxIters);
		printf("\"mapData\" : [\n");
		printJSON<xSize,ySize,zSize>(oldMap, 0);	//Iteration 0 is starting
iteration
	}

	for(int iter=0; iter<maxIters; ++iter)
	{
		if(PRINT) printf(",\n");


		//Start iteration section
		if(TIME) start = clock();
		iterateCA<xSize,ySize,zSize>(oldMap, newMap);
		if(TIME) total += clock() - start;

		#ifdef DO_STATS
		if(iter == maxIters-1)
		{
			countStats<xSize,ySize,zSize>(oldMap, newMap, stats);
			printf("[%d] ",iter+1);
			printf("births: %d	\tdeaths: %d	\talive: %d	\tdead: %d
\ttotal: %d\n", stats.births, stats.deaths, stats.alive, stats.dead,
stats.alive+stats.dead);
		}
		#endif
		if(PRINT) printJSON<xSize,ySize,zSize>(newMap, iter+1);
		temp = oldMap;
		oldMap = newMap;
		newMap = temp;
	}
	if(PRINT) printf("\n]}\n");
	if(TIME)
```

```
	{
		double diff = (double(total))/CLOCKS_PER_SEC;
		printf("time: took %f seconds for %dx%dx%d matrix\n", diff, xSize, ySize,
ySize);
	}
	delete[] oldMap;
	delete[] newMap;

	return 0;
}

template <size_t xSize, size_t ySize, size_t zSize>
void initMap(unsigned char *oldMap, int mapSeed)
{
	srand(mapSeed);
	for(int k=0; k<zSize; ++k)
	{
		for(int j=0; j<ySize; ++j)
		{
			for(int i=0; i<xSize; ++i)
			{
				oldMap[k*(xSize*ySize)+j*xSize+i] = rand() % 2;
			}
		}
	}
}

template <size_t xSize, size_t ySize, size_t zSize>
void iterateCA(unsigned char *oldMap, unsigned char *newMap)
{
	for(int k=0; k<zSize; ++k)
	{
		for(int j=0; j<ySize; ++j)
		{
			for(int i=0; i<xSize; ++i)
			{
				int aliveCnt = countNeighbours<xSize,ySize,zSize>(oldMap,
i, j, k);
				//If cell is alive, check if it dies
				if(oldMap[k*(xSize*ySize)+j*xSize+i] == 1)
				{
					newMap[k*(xSize*ySize)+j*xSize+i] = (aliveCnt <
deathLimit) ? 0 : 1;
				}
				else
				{
					newMap[k*(xSize*ySize)+j*xSize+i] = (aliveCnt >
birthLimit) ? 1 : 0;
				}
				//printf("(%d,%d,%d): %d->%d
[%d]\n",i,j,k,oldMap[i+j*xSize+k*(xSize*ySize)],newMap[i+j*xSize+k*(xSize*ySize)],alive
Cnt);
			}
		}
	}

}

//Count neighbours that are alive
template <size_t xSize, size_t ySize, size_t zSize>
```

```
int countNeighbours(unsigned char *map, int x, int y, int z)
{
	const bool countBounds = true;
    int count = 0;
    for(int k=-1; k<2; ++k)
	{
	 for(int j=-1; j<2; ++j)
		{
			for(int i=-1; i<2; ++i)
			{
				//Count all except middle point
				if( i != 0 || j != 0 || k != 0)
				{
					int xPos = x + i;
					int yPos = y + j;
					int zPos = z + k;

					//Check boundaries
					if(xPos < 0 || yPos < 0 || zPos < 0 || xPos >= xSize
|| yPos >= ySize || zPos >= zSize)
					{
						//if(x==0 && y==0 && z==0)
printf("(%d,%d,%d):bounds\n",xPos,yPos,zPos);
						if(countBounds) count++;
					}
					else
					{
						//if(x==0 && y==0 && z==0)
printf("(%d,%d,%d):not bounds\n",xPos,yPos,zPos);
						count +=
map[zPos*(xSize*ySize)+yPos*xSize+xPos];
					}
				}
			}
	 }
    }
	return count;
}

template <size_t xSize, size_t ySize, size_t zSize>
void printJSON(unsigned char *map, int iter)
{
	printf("\t{\n");
	printf("\t\"iteration\" : %d,\n", iter);
	printf("\t\"map\" : [\n");
	for(int i=0; i<xSize; ++i)
	{
		for(int j=0; j<ySize; ++j)
		{
			for(int k=0; k<zSize; ++k)
			{
				if( !(i == 0 && j == 0 && k == 0) ) printf(",\n");
				char *val = ( map[i+j*xSize+k*(xSize*ySize)] ) ? "true" :
"false";
				printf("\t\t{\"x\":%d, \"y\":%d, \"z\":%d, \"value\":%s}",
i, j, k, val);
			}
		}
	}
	printf("\n\t\t]\n\t}");
```

```
}

template <size_t xSize, size_t ySize, size_t zSize>
void countStats(unsigned char *oldMap, unsigned char *newMap, MapStats &stats)
{
	int oldAlive = 0, oldDead = 0;
	int newAlive = 0, newDead = 0;

	for(int k=0; k<zSize; ++k)
	{
		for(int j=0; j<ySize; ++j)
		{
			for(int i=0; i<xSize; ++i)
			{
				oldAlive += oldMap[k*(xSize*ySize)+j*xSize+i];
				newAlive += newMap[k*(xSize*ySize)+j*xSize+i];
			}
		}
	}
	stats.alive = newAlive;
	newDead= (xSize*ySize*zSize) - newAlive;
	oldDead = (xSize*ySize*zSize) - oldAlive;
	stats.dead = newDead;
	stats.births = (newAlive > oldAlive) ? newAlive - oldAlive : 0;
	stats.deaths = (newDead > oldDead) ? newDead - oldDead : 0;
}
```

- Naive Parallel:

```
#include <cstdio>
#include <cstdlib>
#include <ctime>

#define DO_STATS
#define M_SIZE		128
typedef struct {
	int births;
	int deaths;
	int alive;
	int dead;
} MapStats;

template <size_t xSize, size_t ySize, size_t zSize>
void initMap(unsigned char *oldMap, int mapSeed);
template <size_t xSize, size_t ySize, size_t zSize>
void printJSON(unsigned char *map, int iter);
template <size_t xSize, size_t ySize, size_t zSize>
void countStats(unsigned char *oldMap, unsigned char *newMap, MapStats &stats);

//On Device function
template <size_t xSize, size_t ySize, size_t zSize>
__device__ int countNeighbours(unsigned char *map, int x, int y, int z);
//Kernels
template<size_t xSize, size_t ySize, size_t zSize>
__global__ void unopIterate(unsigned char *d_oldMap, unsigned char *d_newMap, int
iters, int bLim, int dLim);
```

```
//Globals -- bad code
const int deathLimit = 15;
const int birthLimit = 17;

int main()
{
	const int mapSeed = 45000;
	const int xSize = M_SIZE;
	const int ySize = M_SIZE;
	const int zSize = M_SIZE;
	const int maxIters = 30;
	const bool PRINT = false;
	const bool TIME = true;

	const int mapSize = xSize*ySize*zSize;

	const int blockSize = 8;
	dim3 blockDim(blockSize, blockSize, blockSize);
	dim3 gridDim(xSize/blockSize, ySize/blockSize, zSize/blockSize);

	clock_t start, total = 0;
	unsigned char *oldMap = new unsigned char[xSize*ySize*zSize];
	unsigned char *newMap = new unsigned char[xSize*ySize*zSize];
	unsigned char *temp;
	MapStats stats;

	unsigned char *d_oldMap;
	unsigned char *d_newMap;


	cudaMalloc((void **) &d_oldMap, mapSize);
	cudaMalloc((void **) &d_newMap, mapSize);

	initMap<xSize,ySize,zSize>(oldMap, mapSeed);

	if(PRINT)
	{
		printf("{\n");
		printf("\"mapSeed\":%d,\n", mapSeed);
		printf("\"deathLimit\":%d,\n", deathLimit);
		printf("\"birthLimit\":%d,\n", birthLimit);
		printf("\"xSize\":%d,\n", xSize);
		printf("\"ySize\":%d,\n", ySize);
		printf("\"zSize\":%d,\n", zSize);
		printf("\"maxIters\":%d,\n", maxIters);
		printf("\"mapData\" : [\n");
		printJSON<xSize,ySize,zSize>(oldMap, 0);    //Iteration 0 is starting
iteration
	}

	for(int iter=0; iter<maxIters; ++iter)
	{
		if(PRINT) printf(",\n");


		//Main iteration section
		if(TIME) start = clock();
		cudaMemcpy(d_oldMap, oldMap, mapSize, cudaMemcpyHostToDevice);
```

```
                unopIterate<xSize,ySize,zSize><<<gridDim,blockDim>>>(d_oldMap, d_newMap,
1, birthLimit, deathLimit);
                cudaMemcpy(newMap, d_newMap, mapSize, cudaMemcpyDeviceToHost);
                if(TIME) total += clock() - start;

                #ifdef DO_STATS
                if(iter == maxIters-1)
                {
                        countStats<xSize,ySize,zSize>(oldMap, newMap, stats);
                        printf("[%d] ",iter+1);
                        printf("births: %d   \tdeaths: %d   \talive: %d   \tdead: %d
\ttotal: %d\n", stats.births, stats.deaths, stats.alive, stats.dead,
stats.alive+stats.dead);
                }
                #endif
                if(PRINT) printJSON<xSize,ySize,zSize>(newMap, iter+1);
                temp = oldMap;
                oldMap = newMap;
                newMap = temp;
        }
        if(PRINT) printf("\n]}\n");
        if(TIME)
        {
                double diff = (double(total))/CLOCKS_PER_SEC;
                printf("time: took %f seconds for %dx%dx%d matrix\n", diff, xSize, ySize,
ySize);
        }
        delete[] oldMap;
        delete[] newMap;

        return 0;
}

template <size_t xSize, size_t ySize, size_t zSize>
void initMap(unsigned char *oldMap, int mapSeed)
{
        srand(mapSeed);
        for(int k=0; k<zSize; ++k)
        {
                for(int j=0; j<ySize; ++j)
                {
                        for(int i=0; i<xSize; ++i)
                        {
                                oldMap[k*(xSize*ySize)+j*xSize+i] = rand() % 2;
                        }
                }
        }
}

//Count neighbours that are alive
template <size_t xSize, size_t ySize, size_t zSize>
__device__ int countNeighbours(unsigned char *map, int x, int y, int z)
{
        const bool countBounds = true;
    int count = 0;
    for(int k=-1; k<2; ++k)
        {
         for(int j=-1; j<2; ++j)
                {
                        for(int i=-1; i<2; ++i)
```

```
                        {
                                //Count all except middle point
                                if( i != 0 || j != 0 || k != 0)
                                {
                                        int xPos = x + i;
                                        int yPos = y + j;
                                        int zPos = z + k;

                                        //Check boundaries
                                        if(xPos < 0 || yPos < 0 || zPos < 0 || xPos >= xSize
|| yPos >= ySize || zPos >= zSize)
                                        {
                                                //if(x==0 && y==0 && z==0)
printf("(%d,%d,%d):bounds\n",xPos,yPos,zPos);
                                                if(countBounds) count++;
                                        }
                                        else
                                        {
                                                //if(x==0 && y==0 && z==0)
printf("(%d,%d,%d):not bounds\n",xPos,yPos,zPos);
                                                count +=
map[zPos*(xSize*ySize)+yPos*xSize+xPos];
                                        }
                                }
                        }
                }
        }
        return count;
}

template<size_t xSize, size_t ySize, size_t zSize>
__global__ void unopIterate(unsigned char *d_oldMap, unsigned char *d_newMap, int
iters, int bLim, int dLim)
{
        const int globalx = (blockIdx.x * blockDim.x) + threadIdx.x;
        const int globaly = (blockIdx.y * blockDim.y) + threadIdx.y;
        const int globalz = (blockIdx.z * blockDim.z) + threadIdx.z;

        //Only perform action if thread is inside the bounds of the grid
        if( !(globalx >= xSize || globaly >= ySize || globalz >= zSize) )
        {
                int globalIndex = globalz*(xSize * ySize) + globaly*(xSize)+globalx;
                int aliveCnt = countNeighbours<xSize,ySize,zSize>(d_oldMap, globalx,
globaly, globalz);
                if(d_oldMap[globalIndex] == 1)
                {
                        d_newMap[globalIndex] = (aliveCnt < dLim) ? 0 : 1;
                }
                else
                {
                        d_newMap[globalIndex] = (aliveCnt > bLim) ? 1 : 0;
                }
        }
}


template <size_t xSize, size_t ySize, size_t zSize>
void printJSON(unsigned char *map, int iter)
{
        printf("\t{\n");
```

```
        printf("\t\"iteration\" : %d,\n", iter);
        printf("\t\"map\" : [\n");
        for(int i=0; i<xSize; ++i)
        {
                for(int j=0; j<ySize; ++j)
                {
                        for(int k=0; k<zSize; ++k)
                        {
                                if( !(i == 0 && j == 0 && k == 0) ) printf(",\n");
                                char *val = ( map[k*(xSize*ySize)+j*xSize+i] ) ? "true" :
"false";
                                printf("\t\t{\"x\":%d, \"y\":%d, \"z\":%d, \"value\":%s}",
i, j, k, val);
                        }
                }
        }
        printf("\n\t\t]\n\t}");
}

template <size_t xSize, size_t ySize, size_t zSize>
void countStats(unsigned char *oldMap, unsigned char *newMap, MapStats &stats)
{
        int oldAlive = 0, oldDead = 0;
        int newAlive = 0, newDead = 0;

        for(int k=0; k<zSize; ++k)
        {
                for(int j=0; j<ySize; ++j)
                {
                        for(int i=0; i<xSize; ++i)
                        {
                                oldAlive += oldMap[k*(xSize*ySize)+j*xSize+i];
                                newAlive += newMap[k*(xSize*ySize)+j*xSize+i];
                        }
                }
        }
        stats.alive = newAlive;
        newDead= (xSize*ySize*zSize) - newAlive;
        oldDead = (xSize*ySize*zSize) - oldAlive;
        stats.dead = newDead;
        stats.births = (newAlive > oldAlive) ? newAlive - oldAlive : 0;
        stats.deaths = (newDead > oldDead) ? newDead - oldDead : 0;
}
```

- Naive Parallel (minimum use of memcpy):

Only Main function differs from above code.

```
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <Windows.h>

#define DO_STATS
#define M_SIZE                1024
typedef struct {
        int births;
        int deaths;
```

```
	int alive;
	int dead;
} MapStats;

template <size_t xSize, size_t ySize, size_t zSize>
void initMap(unsigned char *oldMap, int mapSeed);
template <size_t xSize, size_t ySize, size_t zSize>
void printJSON(unsigned char *map, int iter);
template <size_t xSize, size_t ySize, size_t zSize>
void countStats(unsigned char *oldMap, unsigned char *newMap, MapStats &stats);

//On Device function
template <size_t xSize, size_t ySize, size_t zSize>
__device__ int countNeighbours(unsigned char *map, int x, int y, int z);
//Kernels
template<size_t xSize, size_t ySize, size_t zSize>
__global__ void unopIterate(unsigned char *d_oldMap, unsigned char *d_newMap, int
iters, int bLim, int dLim);


//Globals -- bad code
const int deathLimit = 15;
const int birthLimit = 17;

int main()
{
	const int mapSeed = 45000;
	const int xSize = M_SIZE;
	const int ySize = M_SIZE;
	const int zSize = M_SIZE;
	const int maxIters = 30;
	const bool TIME = true;

	const int mapSize = xSize*ySize*zSize;

	const int blockSize = 8;
	dim3 blockDim(blockSize, blockSize, blockSize);
	dim3 gridDim(xSize/blockSize, ySize/blockSize, zSize/blockSize);

	clock_t start, total = 0;
	unsigned char *oldMap = new unsigned char[xSize*ySize*zSize];
	unsigned char *newMap = new unsigned char[xSize*ySize*zSize];
	unsigned char *temp;
	MapStats stats;

	unsigned char *d_oldMap;
	unsigned char *d_newMap;

	cudaMalloc((void **) &d_oldMap, mapSize);
	cudaMalloc((void **) &d_newMap, mapSize);

	initMap<xSize,ySize,zSize>(oldMap, mapSeed);

	#ifdef DO_STATS

	#endif

	//Main iteration section
	if(TIME) start = clock();
	cudaMemcpy(d_oldMap, oldMap, mapSize, cudaMemcpyHostToDevice);
```

```
	for(int iter=0; iter<maxIters; ++iter)
	{
		unopIterate<xSize,ySize,zSize><<<gridDim,blockDim>>>(d_oldMap, d_newMap,
1, birthLimit, deathLimit);
//		if(TIME) total += clock() - start;
//		Sleep(50);
//		printf("[%d] ", iter+1);
//		if(TIME) start = clock();
		if(iter != maxIters-1)
		{
			temp = d_oldMap;
			d_oldMap = d_newMap;
			d_newMap = temp;
		}
	}
	cudaMemcpy(newMap, d_newMap, mapSize, cudaMemcpyDeviceToHost);
	if(TIME) total += clock() - start;
	//printf("\n");
	cudaMemcpy(oldMap, d_oldMap, mapSize, cudaMemcpyDeviceToHost);
	#ifdef DO_STATS
	printf("[%d] ",maxIters);
	countStats<xSize,ySize,zSize>(oldMap, newMap, stats);
	printf("births: %d   \tdeaths: %d   \talive: %d   \tdead: %d   \ttotal: %d\n",
stats.births, stats.deaths, stats.alive, stats.dead, stats.alive+stats.dead);
	#endif

	if(TIME)
	{
		double diff = (double(total))/CLOCKS_PER_SEC;
		printf("time: took %f seconds for %dx%dx%d matrix\n", diff, xSize, ySize,
ySize);
	}

	delete[] oldMap;
	delete[] newMap;

	return 0;
}
```