CS3210 Parallel Computing

Project 2 Subtask 3 – 3D Game of Life

Wong Yong Jie <a0088826@nus.edu.sg> Lewis Haris Nata <a0099727@nus.edu.sg>

1. Introduction

Conway's Game of Life is a cellular automaton devised by British mathematician John Horton Conway. It is a zero-player game, where the progress of the game is determined by the initial state of the universe and some parameters. The Game's universe is traditionally 2D, but has been extended by Prof Carter Bays' to the 3rd dimension (http://www.cse.sc.edu/~bays/d4d4d4/).

The implementation above uses a Java applet. There is an alternative implementation by Samuel Levy (http://gameoflife.samuellevy.com/) which uses WebGL and does not require a plug-in. Both implementations perform their computation on the CPU.

In this project, a CUDA-based version of 3D Game of Life is designed and implemented.

2. Problem Description

In 3D Game of Life, the universe is a regular cube with sz^3 discrete slots. Each slot can have two states: either a cell is present or not. The number of neighbours for each cell, along with the current state of the slot, determine its next state.

The rules are as follows: If the slot has a cell, it will die if the number of neighbours is more than $r3$ or less than $r4$. Otherwise, the cell remains alive. If the slot does not have a cell, a new cell will grow in it if the number of neighbours is equal or more than $r1$ and equal or less than $r2$.

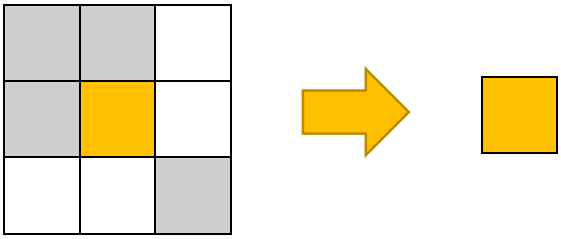
Using simple analysis, it can be seen that the maximum number of neighbours per cell is 26, and the sum of neighbours for each cell can be computed independently.

3. Solution Description

Task Decomposition

Four decomposition techniques were considered: recursive, data, exploratory and speculative. Exploratory decomposition can be ruled out since it is applicable only when the decomposition of the task cannot be pre-computed. Similarity, speculative decomposition is also not applicable as independent sub-tasks (sum of neighbours computation) can be easily identified. Recursive decomposition relies on the presence of sub-problems that can be recursively decomposed and solved. However, this problem does not exhibit such sub-problems.

From the problem, it can be seen that each element of the output is simply a function of the input.



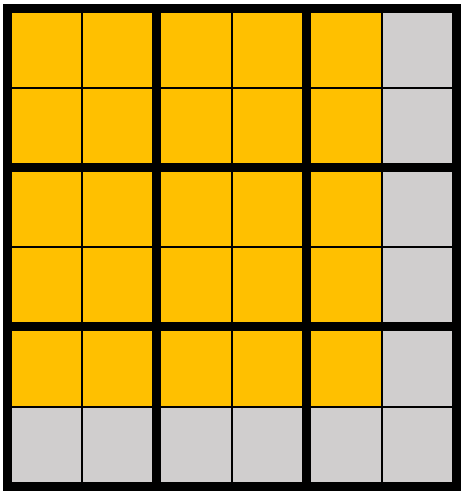
The $3 \times 3 \times 3$ grid of slots surrounding a cell used to determine the output for that particular slot. Grey slots are slots occupied with cells that are live.

Each slot in the output field creates a task. As data decomposition using input data partitioning is the most natural decomposition for this problem, it has been chosen as our task decomposition method.

For a universe with dimension sz, sz^3 subtasks are generated per generation.

Mapping of Tasks to Processors

The CUDA programming model allows processes to be organized into grids, blocks and threads. Each of the sz^3 subtasks are mapped into a thread. These threads can be executed in parallel. The field is divided into smaller cubes of dimensions $\text{GOL\_CUDA\_BLOCK\_X} \times \text{GOL\_CUDA\_BLOCK\_Y} \times \text{GOL\_CUDA\_BLOCK\_Z}$. If the field cannot be evenly divisible by these dimensions, no-op tasks are appended to the remaining slots.



The diagram above shows the decomposition of a $5 \times 5 \times 5$ field into $2 \times 2 \times 2$ blocks. The slots are actual tasks, and the grey slots are no-op tasks. While there are $5^3 = 125$ actual tasks, $6^3 = 216$ threads are actually spawned. The black lines indicate the boundaries of the block.

Each of these sub-cubes are mapped onto a CUDA block. In our implementation, all 3 constants are defined to be 8. This gives a total number of 512 threads per block, which is the block size limit imposed by processors with CUDA Compute Capability 1.x.

4. Implementation and Code Walkthrough

Our implementation comes in two forms: `3dgol` and `3dgol_gui`. The former is a command-line, batch mode version, and the latter is an interactive version with 3D display. We note that the latter has a slightly different set of required parameters, and they can be listed by running the executable with no arguments.

File	Description	File	Description
3dgol.cpp	Game of Life interface	util.cpp	Utility functions for CUDA
display.cpp	3D display logic	kernels.cu	Game of Life CUDA kernels
field.cpp	Abstract data type for a field	*.frag	Fragment shaders for GUI
gui.cpp	Entry point for 3dgol_gui	*.vert	Vertex shaders for GUI
main.cpp	Entry point for 3dgol	CMakeLists.txt	CMake build file

Field Representation (field.cpp)

The field is represented using a linear array of integers. The 3D field is mapped to a linear array using the following formula:

```
x * this->_size * this->_size + y * this->_size + z
```

Where `this->_size` is the size of the field in a single dimension, and `x`, `y` and `z` are the requested slot coordinates in the field.

This linear array of integers is allocated using `cudaMallocHost` rather than plain `malloc` for better performance. According to our informal benchmarks, this reduces the total execution time of `3dgol` by about 200 ms.

```
cudaError_t result = cudaMallocHost(&this->field, allocationSize);
if (result != cudaSuccess) {
    Util::cudaThrowError("…", result);
}
```

The `Field` class also provides some routines (`allocateOnDevice`, `transferToDevice`, `transferFromDevice`) for convenient data transfer between host and device. Their implementations are:

```
cudaMalloc(&this->fieldOnDevice, allocationSize);
cudaMemcpy(this->fieldOnDevice, this->field,
           allocationSize, cudaMemcpyHostToDevice);
cudaMemcpy(this->field, this->fieldOnDevice,
                 allocationSize, cudaMemcpyDeviceToHost);
```

GameOfLife Class (3dgol.cpp)

The `GameOfLife` class encapsulates the logic involved in the game. Upon initialization, CUDA capability is checked, and a new field with the supplied size is created. If a file is supplied, `initializeField` populates the field with data from the file. Otherwise, random data is generated.

```
this->cudaCheck();
this->field = new Field(sz);
this->initializeField();
```

Additionally, it provides one crucial method: `iterate`. When called, this method advances the state of the game by on generation. This method has two implementations.

```
void GameOfLife::iterate() {
    // Run the kernel.
    Field* outField = new Field(this->sz);
    golPerformSimulation(this->field, outField, this->r1, this->r2,
            this->r3, this->r4);
    delete this->field;
    this->field = outField;
}
```

Firstly, a field to contain the results of the next generation is allocated. Then, this field is passed to another method `golPerformSimulation`, which is the entry point to the CUDA kernel. When the method is complete, the old field is deleted, and the new field replaces the old.

However, this method is inefficient in cases where we want to execute many successive iterations without requiring the intermediate field states. This is because the results have to be copied from host to device and vice versa many times. The second implementation solves this problem.

```
void GameOfLife::iterate(int count) {
    Field* outField = new Field(this->sz);
    golPerformSimulationMulti(this->field, outField, this->r1, this->r2,
            this->r3, this->r4, count);
    delete this->field;
    this->field = outField;
}
```

While this second implementation looks similar to the first, the underlying kernel is different. This will be covered in the next section.

Kernel Entry Point (kernels.cu)

The kernel entry point `golPerformSimulation` prepares the CUDA kernel for execution. Firstly, the dimensions of the grid and block are specified:

```
dim3 grid = dim3((field->size() + GOL_CUDA_BLOCK_X - 1) /
GOL_CUDA_BLOCK_X,
            (field->size() + GOL_CUDA_BLOCK_Y - 1) / GOL_CUDA_BLOCK_Y,
            (field->size() + GOL_CUDA_BLOCK_Z - 1) / GOL_CUDA_BLOCK_Z);
dim3 block = dim3(GOL_CUDA_BLOCK_X, GOL_CUDA_BLOCK_Y, GOL_CUDA_BLOCK_Z);
```

Note that `(field->size() + GOL_CUDA_BLOCK_X - 1)` is equivalent to `std::ceil((double) field->size() / GOL_CUDA_BLOCK_COUNT)`, and this provides the additional no-op threads that are described previously. Then, the input and output buffers are prepared:

```
// Prepare the buffers.
int* data = field->allocateOnDevice();
field->transferToDevice();
int* outData = outField->allocateOnDevice();
```

The kernel is then executed with the correct configuration and parameters:

```
// Run the kernel.
golKernelSharedMemory<<<grid, block>>> (data, outData, field->size(), r1,
            r2, r3, r4);
cudaError_t result = cudaGetLastError();
if (result != cudaSuccess) {
    Util::cudaThrowError("Unable to run kernel", result);
}
```

`cudaGetLastError` is executed right after the kernel is run to determine if there were any problems executing the CUDA kernel. Lastly, the output is copied back to host memory and the memory allocated on the GPU is freed.

```
outField->transferFromDevice();

// Free memory.
cudaFree(data);
cudaFree(outData);
```

The alternate kernel, `golPerformSimulationMulti` is largely similar but with an important difference. This kernel goes through multiple generations without copying any output back to host memory.

```
// Run the kernel multiple times.
for (int i = 0; i < count; i++) {
    golKernelSharedMemory<<<grid, block>>> (data, outData, field->size(),
                r1, r2, r3, r4);
    …
    // Swap input and output.
    int* temp = outData;
    outData = data;
    data = temp;
}
```

At every iteration, the input and output pointers to the buffer in GPU memory are swapped. As no copying takes place, this is highly efficient.

The Kernel (kernels.cu)

A convenience function is defined to quickly locate the desired slot in the linear array.

```
__device__ int golKernelGetOffset(int x, int y, int z, int size) {
    return x * size * size + y * size + z;
}
```

We implemented a kernel that utilizes CUDA shared memory to improve performance. However, it was difficult to avoid un-coalesced accesses to global memory due to the cubic arrangement of required data. The signature of the kernel is:

```
__global__ void golKernelSharedMemory(int* data, int* outData, int size,
        int r1, int r2, int r3, int r4)
```

Firstly, the output coordinates of the thread is obtained. Threads that have output coordinates beyond the size of the field return immediately as there is no work to do.

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
int z = blockIdx.z * blockDim.z + threadIdx.z;

// Some threads would be left idle as a consequence.
if (x >= size || y >= size || z >= size) {
    return;
}
```

Then, data from global memory is copied into the shared memory.

```
// Use shared memory.
int threadLinearId = threadIdx.x * GOL_CUDA_BLOCK_X * GOL_CUDA_BLOCK_X +
        threadIdx.y * GOL_CUDA_BLOCK_Y + threadIdx.z;
__shared__ int dataS[GOL_CUDA_BLOCK_COUNT];
dataS[threadLinearId] = data[dataOffset];
__syncthreads();
```

The shared memory contains all the data that is required by a single block. Note that `__syncthreads` is called after the shared memory is copied to ensure that all threads have completed copying data, avoiding potential race conditions.

Then, the sum of neighbours is computed:

```
// Check neighbours.
int sum = 0;
for (int i = -1; i < 2; i++) {
    for (int j = -1; j < 2; j++) {
        for (int k = -1; k < 2; k++) {
            sum += golKernelHasNeighbour(x, y, z, i, j, k, size, data,
                    dataS);
        }
    }
}
```

The subroutine `golKernelHasNeighbour` checks if a neighbour exists at the specified coordinates, taking into account if the slot is located within shared memory. If a required value is not found in shared memory, then global memory is used. If a neighbour exists, then the subroutine returns 1. Otherwise, 0 is returned.

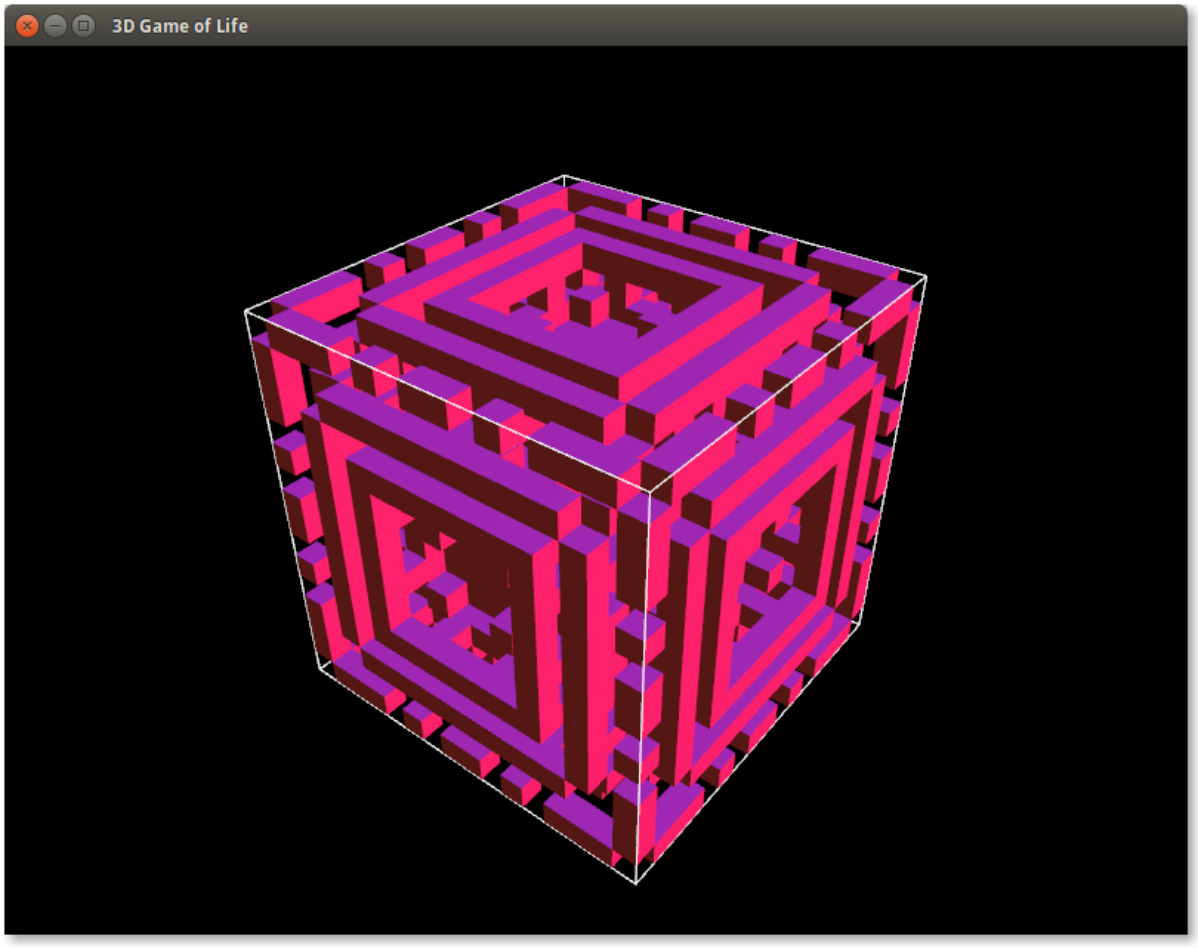
The rules of 3D Game of Life are then applied.

```
// Apply the rules.
int thisCell = dataS[sharedDataOffset];
if (thisCell == 0) {
    if (sum >= r1 && sum <= r2) {
        outData[dataOffset] = 1;
    } else {
        outData[dataOffset] = 0;
    }
} else {
    if (sum > r3 || sum < r4) {
        outData[dataOffset] = 0;
    } else {
        outData[dataOffset] = 1;
    }
}
```

The results of the computation are stored in `outData`.

3D Display (display.cpp)

As the implementation of the 3D display is not the focus of this report, only a brief overview is provided. It relies on OpenGL and GLM (OpenGL Math Library), and uses Phong shading and shaders to improve performance and presentation.



At high-level, the display is rendered using a render loop:

```
// Begin the event loop.
while (!glfwWindowShouldClose(window)) {
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Draw a wireframe cube with axes.
    this->drawWireCube();

    // Request new generation based on timer.
    this->time = glfwGetTime();
    if (sp != 0 && this->time > this->timeNextGen) {
        this->gameOfLife->iterate();
        this->timeNextGen = this->time + sp / 1000.0;
    }

    // Draw the cubes.
    this->drawCubes();

    // Display the output.
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

The display maintains its own instance of the Game of Life state. When a new state is requested, `iterate` is called, and `this->drawCubes` renders each cell as a cube based on the state.

5. Evaluation

The graphics hardware used was an nVidia GeForce GTX850M with Compute Capability 5.0. This GPU has 640 CUDA cores. The benchmark used was:

```
./3dgol 0 32 4 5 7 6 0 10000
```

This creates a $32 \times 32 \times 32$ cube, and 10,000 generations are computed.

The execution time was measured using two means: via the Unix `time` utility, and via events in the CUDA runtime API. The former was used to measure the total execution time of the program, and the latter was used to measure the total execution time of the kernel. Measurements were taken 3 times.

Two versions of the program were tested: one utilizing shared memory and one without. The results are summarized below.

Component	Shared Memory?	Time Taken (s)
Entire Program	No	2.176
Kernel Only	No	1.263
Entire Program	Yes	1.210
Kernel Only	Yes	0.952

As observed in the results above, the use of shared memory provided a speedup of 1.798x.

6. Conclusion

As it can be observed, the 3D Game of Life problem is suited for implementation on GPUs.