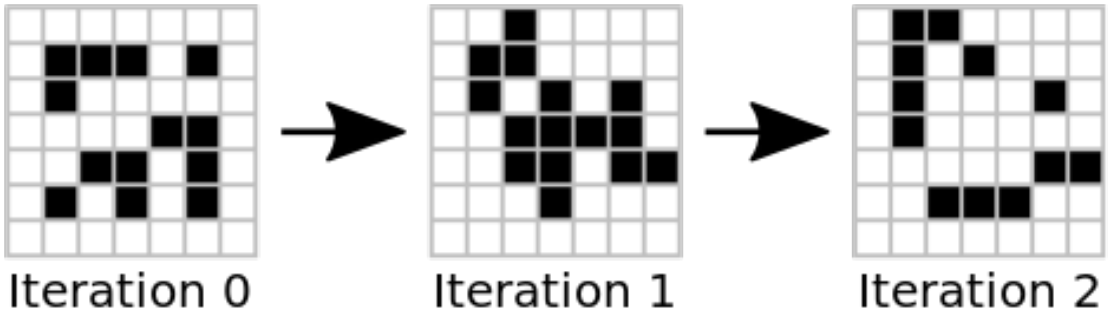
GameOfLife

March 20, 2020

1 Game of Life on GPU - Interactive & Extensible

- The example shows an interactive workflow of simulation and analysis.
- The simulation runs Conway's Game of Life on a GPU.



2 Include and Link

[1]:
```
// set the include path of PNGwriter
//   https://github.com/pngwriter/pngwriter/tree/dev
// (like -Ipngwriter/include for a compiler)
#pragma cling(add_include_path "pngwriter/include")

#include <fstream>
#include <vector>
#include <sstream>
#include <chrono>
#include <thread>

// include PNGwriter
#define NO_FREETYPE
#include <pngwriter.h>

// helper functions for displaying images
#include "xtl/xbase64.hpp"
#include "xeus/xjson.hpp"
```

```
// self-defined helper functions
#include "color_maps.hpp"
#include "input_reader.hpp"
#include "png_generator.hpp"
#include "helper.hpp"

// link PNGwriter (like -lPNGwriter for a compiler)
#pragma cling(load "pngwriter/lib/libPNGwriter.so")
```

3 Game of Life Setup

- setup world size
- allocate memory on CPU and GPU
- load initial world
- copy initial world to the GPU
- generate image of the initial world

[2]:
```
// size of the world
const unsigned int dim = 10u;
// two extra columns and rows for ghostcells
const unsigned int world_size = dim + 2u;
unsigned int iterations = 5;
unsigned int current_png = 0;

// pointers for host and device memory
int * sim_world;
int * d_sim_world;
int * d_new_sim_world;
int * d_swap;
// allocate memory on CPU and GPU
sim_world = new int[ world_size * world_size ];
cuCheck(cudaMalloc( (void **) &d_sim_world, sizeof(int)*world_size*world_size));
cuCheck(cudaMalloc( (void **) &d_new_sim_world,␣
 ↪sizeof(int)*world_size*world_size));

// read initial world from a file
if (int error = read_input("input.txt", sim_world, dim, dim, true))
    std::cout << "read input world failed - error code: " << error << std::endl;

// copy initial world to GPU
cuCheck(cudaMemcpy(d_sim_world, sim_world, sizeof(int)*world_size*world_size,␣
 ↪cudaMemcpyHostToDevice));

// allocate memory for the simulation images
std::vector< std::vector< unsigned char > > sim_pngs;
```

```
// create an image of the initial world
BlackWhiteMap<int> bw_map;
sim_pngs.push_back(generate_png<int>(sim_world, world_size, world_size,␣
 ↪&bw_map, true, 20));
```

4 CUDA Kernels

[3]:
```
// periodic boundary conditions: copy the first/last row/column
__global__ void update_boundaries(int dim, int *world) {
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;

  // ignore the first two threads: only needed to copy ghost cells for columns
  if(col > 1) {
    if(row == 0) {
      // Copy first real row to bottom ghost row
      world[col-1 + (dim+2)*(dim+1)] = world[(dim+2)     + col-1];
    }else{
      // Copy last real row to top ghost row
      world[col-1]                   = world[(dim+2)*dim + col-1];
    }
  }
  __syncthreads();

  if(row == 0) {
    // Copy first real column to right most ghost column
    world[col*(dim+2)+dim+1] = world[col*(dim+2) + 1];
  } else {
    // Copy last real column to left most ghost column
    world[col*(dim+2)      ] = world[col*(dim+2) + dim];
  }

}
```

[4]:
```
// main kernel that calculates an iteration of the game of life
__global__ void GOL_GPU(int dim, int *world, int *newWorld) {
   int row = blockIdx.y * blockDim.y + threadIdx.y + 1;
   int col = blockIdx.x * blockDim.x + threadIdx.x + 1;
   int id = row*(dim+2) + col;

   int numNeighbors;
   int cell = world[id];

   numNeighbors =   world[id+(dim+2)]   // lower
      + world[id-(dim+2)]               // upper
```

```
      + world[id+1]                     // right
      + world[id-1]                     // left

      + world[id+(dim+3)]   // diagonal lower right
      + world[id-(dim+3)]   // diagonal upper left
      + world[id-(dim+1)]   // diagonal upper right
      + world[id+(dim+1)];  // diagonal lower left

   if (cell == 1 && numNeighbors < 2)
      newWorld[id] = 0;

    // 2) Any living cell with two or three living neighbors lives
    else if (cell == 1 && (numNeighbors == 2 || numNeighbors == 3))
      newWorld[id] = 1;

    // 3) Any living cell with more than three living neighbors dies
    else if (cell == 1 && numNeighbors > 3)
      newWorld[id] = 0;

    // 4) Any dead cell with exactly three living neighbors becomes alive
    else if (cell == 0 && numNeighbors == 3)
      newWorld[id] = 1;

    else
      newWorld[id] = cell;
}
```

5 Interactive Simulation: Main Loop

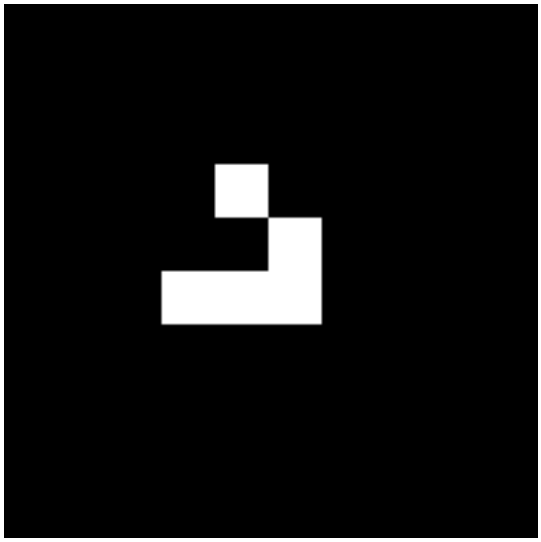
- calculate new iterations
- swap new world with the old one
- generate an image of the current iteration

[8]:
```
// main loop
for(unsigned int i = 0; i < iterations; ++i) {

    update_boundaries<<<1, dim3(dim+2, 2, 1)>>>(dim, d_sim_world);
    GOL_GPU<<<1, dim3(dim, dim, 1)>>>(dim, d_sim_world, d_new_sim_world);
    cuCheck(cudaDeviceSynchronize());

    d_swap = d_new_sim_world;
    d_new_sim_world = d_sim_world;
    d_sim_world = d_swap;
```

```
    cuCheck(cudaMemcpy(sim_world, d_sim_world,␣
 ↪sizeof(int)*world_size*world_size, cudaMemcpyDeviceToHost));
    sim_pngs.push_back(generate_png<int>(sim_world, world_size, world_size,␣
 ↪&bw_map, true, 20));
}
```

6 Display Simulation Images

[9]:
```
for(; current_png < sim_pngs.size(); ++current_png) {
    display_image(sim_pngs[current_png], true);
    std::cout << "iteration = " << current_png << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(800));
}
```



```
iteration = 8
```

6.1 Nonlinear Program Flow

[7]:
```
iterations = 3;
```

7 In-Situ Data Analysis

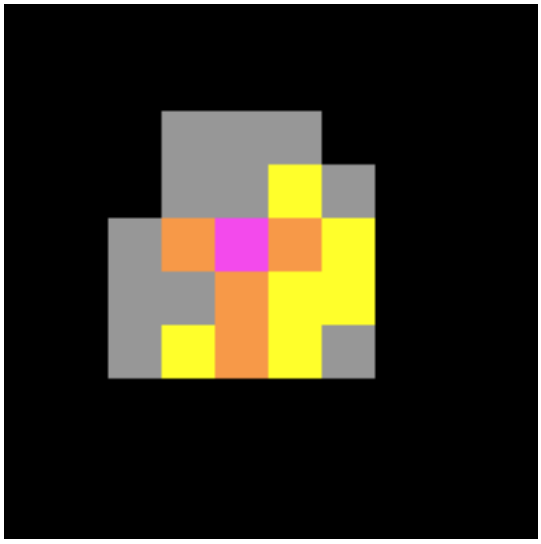
- heatmap of the living neighbors for each cell
- kernel uses the simulation result as its input and writes output to an extra buffer

[10]:
```
// counts the living neighbors of a cell
__global__ void get_num_neighbors(int dim, int *world, int *newWorld) {
   int row = blockIdx.y * blockDim.y + threadIdx.y + 1;
   int col = blockIdx.x * blockDim.x + threadIdx.x + 1;
   int id = row*(dim+2) + col;

   newWorld[id] =   world[id+(dim+2)]   // lower
      + world[id-(dim+2)]               // upper
      + world[id+1]                     // right
      + world[id-1]                     // left

      + world[id+(dim+3)]   // diagonal lower right
      + world[id-(dim+3)]   // diagonal upper left
      + world[id-(dim+1)]   // diagonal upper right
      + world[id+(dim+1)];  // diagonal lower left
}
```

- allocate memory for analysis on CPU and GPU
- run the analysis
- generate an image from the analysis result

[11]:
```
// allocate extra memory on the GPU to output the analysis
int * d_ana_world;
cuCheck(cudaMalloc( (void **) &d_ana_world, sizeof(int)*world_size*world_size));
// allocate memory on CPU for the images
std::vector< std::vector< unsigned char > > ana_pngs;
int * ana_world = new int[world_size*world_size];

// run the analysis
// use the simulation data as input and write the result into extra memory
get_num_neighbors<<<1,dim3(dim, dim, 1)>>>(dim, d_sim_world, d_ana_world);

// copy analysis data to the CPU
cuCheck(cudaMemcpy(ana_world, d_ana_world, sizeof(int)*world_size*world_size,␣
 ↪cudaMemcpyDeviceToHost));
// generate a heat map image
ana_pngs.push_back(generate_png<int>(ana_world, world_size, world_size,␣
 ↪&ch_map, true, 20));
```

[12]:
```
display_image(ana_pngs.back(), true);
```

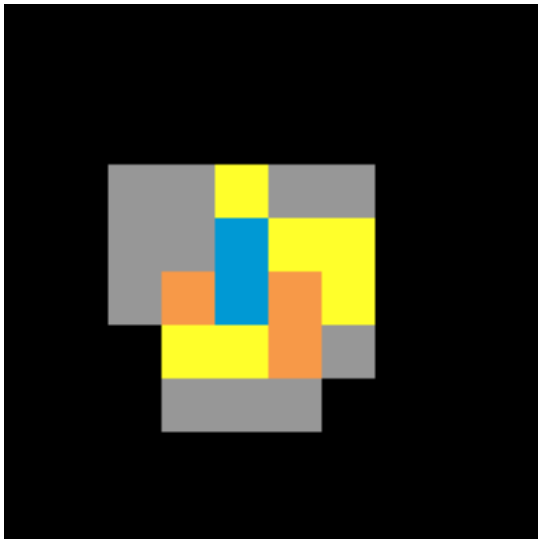


8 Run a single Simulation step with Analysis

[13]:
```
// one iteration of the simulation
update_boundaries<<<1, dim3(dim+2, 2, 1)>>>(dim, d_sim_world);
GOL_GPU<<<1, dim3(dim, dim, 1)>>>(dim, d_sim_world, d_new_sim_world);
cuCheck(cudaDeviceSynchronize());

// swap memory
d_swap = d_new_sim_world;
d_new_sim_world = d_sim_world;
d_sim_world = d_swap;
```

[14]:
```
// run analysis
get_num_neighbors<<<1,dim3(dim, dim, 1)>>>(dim, d_sim_world, d_ana_world);
cuCheck(cudaMemcpy(ana_world, d_ana_world, sizeof(int)*world_size*world_size,␣
 ↪cudaMemcpyDeviceToHost));
ana_pngs.push_back(generate_png<int>(ana_world, world_size, world_size,␣
 ↪&ch_map, true, 20));
```

[15]:
```
display_image(ana_pngs.back(), true);
```



9 Extras

9.1 Interactive Input

- Jupyter Notebook offers "magic" commands that provide language-independent functions
- magic commands starts with `%%`
- `%%file [name]` writes the contents of a cell to a file
  - the file is stored in the same folder as the notebook and can be loaded via C/C++ functions
- depends on the language kernel (xeus features)

Define the initial world for the Game of Life simulation. `X` are living cells and `0` are dead.

[ ]:
```
%%file input.txt
0 0 0 0 0 0 0 0 0 0
0 0 X 0 0 0 0 0 0 0
0 0 0 X 0 0 0 0 0 0
0 X X X 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

9.2 Reset the simulation without restarting the kernel

[ ]:
```
// load a new inital world into the host memory
read_input("input.txt", sim_world, dim, dim, true);
// copy the world to the device
cuCheck(cudaMemcpy(d_sim_world, sim_world, sizeof(int)*world_size*world_size,␣
 ↪cudaMemcpyHostToDevice));
// reset png print counter
current_png = 0;
// delete old images
sim_pngs.clear();
// create an image of the initial world
sim_pngs.push_back(generate_png<int>(sim_world, world_size, world_size,␣
 ↪&bw_map, true, 20));
```