CUDA: CONWAY'S GAME OF LIFE

GREG LINKLATER, G12L4025[1]

April 5, 2016

INTRODUCTION

Conway's Game of Life is a computational experiment where a world – divided into a 2 dimensional grid of a particular width and height – constantly changes on a cell-by-cell basis according to four rules:

1. Any live cell with fewer than two live neighbours dies.
2. Any live cell with two or three live neighbours lives.
3. Any cell with more than three neighbours dies.
4. Any dead cell with exactly three live neighbours becomes a live cell.

In this case a cell is defined as a distinct set of x and y coordinates and a cell's neighbourhood is defined as being the eight other cells that share a border with it (Moore neighbourhood).
(Fiser, 2013)

Through the use of a Nvidia GPU and CUDA C, it is possible to achieve a computation speed that is orders of magnitude larger than what is possible on a CPU.

[1] *Department of Computer Science, Rhodes University, Grahamstown, South Africa*

METHOD

Conway's Game of Life is executed by completing an iteration before the next can take place. As the next iteration depends on the current, it is not possible to parallelise this process; instead the resources of an extremely parallelised device can be best used to achieve a speedup within a single iteration.

The process for computing a single iteration of Conway's Game of Life, regardless of whether or not it is being evaluated sequentially or in parallel, is as follows:

1. Select a cell.
2. Calculate the number of alive neighbours.
3. Apply the above rules.
4. Write the cell result to the correct location in the result buffer.

Once all of the cells have been evaluated for that iteration the result buffer then becomes the data buffer for the next iteration and the process repeats. For the purpose of analysing the relative performance of different GPU and sequential operations, a number of types were defined in an attempt to standardise the operation.

Listing 1: Type definitions used across all operations

```
// use single byte to store cell state in order to increase memory efficiency.
typedef unsigned char ubyte;

// struct to store world parameters
typedef struct World {
  size_t worldWidth;
  size_t worldHeight;
  size_t dataLength;  // worldWidth * worldHeight
} world;

// struct to ease the passing of data to operations
typedef struct Board {
  world *_world;
  ubyte *data;
  ubyte *resultData;
} board;
```

Sequential CPU Implementation

The sequential implementation itself contains some minor optimisations such as the use of one byte of memory per cell so as to conserve memory usage. When considering the overall memory usage of a board with millions of cells as well as the fact that we need to allocate memory for twice the board size for the result buffer, it means that by not simply using an integer to store the state of each cell we use a quarter of the memory that we would use. However, fundamentally due to the sequential nature of operations on the CPU, there is still very little expectation of the CPU being able to perform in this regard in any way that is comparable to the GPU.

The code below was taken from an online article written by Marek Fiser (Fiser, 2013).

Listing 2: Sequential CPU Code for Conway's Game of Life (Fiser, 2013)

```
/*
 * countAliveCells
 * Evaluate the number of cells in the Moore Neighbourhood that are alive.
 */
inline ubyte countAliveCells(
    size_t x0,
    size_t x1,
    size_t x2,
    size_t y0,
    size_t y1,
    size_t y2
  ) {
  return gameBoard->data[x0 + y0] + gameBoard->data[x1 + y0]
    + gameBoard->data[x2 + y0] + gameBoard->data[x0 + y1]
    + gameBoard->data[x2 + y1] + gameBoard->data[x0 + y2]
    + gameBoard->data[x1 + y2] + gameBoard->data[x2 + y2];
}

/*
 * computeIterationSerial
 * For each cell in the data buffer evaluate if the cell should live or die
 * according to the rules and write to results buffer.
 */
void computeIterationSerial(ubyte *data, world *gameWorld, ubyte *resultData) {
  for (size_t y = 0; y < gameWorld->worldHeight; ++y) {
    // Compute cell coordinates from 1D space to 2D space.
    size_t y0 = ((y + gameWorld->worldHeight - 1) % gameWorld->dataLength);
    size_t y1 = y * gameWorld->worldWidth;
    size_t y2 = ((y + 1) % gameWorld->dataLength);

    for (size_t x = 0; x < gameWorld->worldWidth; ++x) {
      size_t x0 = (x + gameWorld->worldWidth - 1) % gameWorld->worldWidth;
      size_t x2 = (x + 1) % gameWorld->worldWidth;

      ubyte aliveCells = countAliveCells(x0, x, x2, y0, y1, y2);
      // Write evaluated cell state.
      resultData[y1 + x] =
        aliveCells == 3 || (aliveCells == 2 && gameBoard->data[x + y1]) ? 1 : 0;
    }
  }
  // Switch result buffer and data buffer for next iteration.
  std::swap(gameBoard->data, gameBoard->resultData);
}
```

Naive GPU Implementation

A direct translation of the algorithm from the CPU implementation (Listing 2) into CUDA C results in the code in Listing 3 on the next page.

Notable features of the naive CUDA implementation is that, like the CPU implementation, memory is allocated in a byte-per-cell manner and that there is no branching in the kernel itself. Both minor optimisations that remove the immediate problems that may impact efficiency if the kernel was written without those optimisations in mind. The problem with the naive kernel is that it is memory locked. Due to having to make three separate accesses of three bytes apiece and making significant use of the registers on every thread, we are using the entirity of our memory pipeline as well as limiting the amount of threads that can be running at any given point in time.

Listing 3: Naive GPU Code for Conway's Game of Life (Fiser, 2013)

```
/*
 * simpleLifeKernel
 * Compute offsets required to access all cells in the Moore Neighbourhood and
 * use to evaluate the life of a cell in the next iteration.
 *
 * This kernel is executed once per iteration by as many threads and blocks as
 * required to complete the iteration.
 */
__global__ void simpleLifeKernel(
    const ubyte *lifeData,
    uint worldWidth,
    uint worldSize,
    ubyte *resultLifeData
  ) {
  for (uint cellId = __mul24(blockIdx.x, blockDim.x) + threadIdx.x;
      cellId < worldSize;
      cellId += blockDim.x * gridDim.x
    ) {
    // Calculate offset values
    uint x = cellId % worldWidth;
    uint yAbs = cellId - x;
    uint xLeft = (x + worldWidth - 1) % worldWidth;
    uint xRight = (x + 1) % worldWidth;
    uint yAbsUp = (yAbs + worldSize - worldWidth) % worldSize;
    uint yAbsDown = (yAbs + worldWidth) % worldSize;

    // Calculate number of cells alive in the Moore Neighbourhood
    uint aliveCells = lifeData[xLeft + yAbsUp] + lifeData[x + yAbsUp]
      + lifeData[xRight + yAbsUp] + lifeData[xLeft + yAbs]
      + lifeData[xRight + yAbs] + lifeData[xLeft + yAbsDown]
      + lifeData[x + yAbsDown] + lifeData[xRight + yAbsDown];

    // Evaluate the life of the cell in question.
    resultLifeData[x + yAbs] =
      aliveCells == 3 || (aliveCells == 2 && lifeData[x + yAbs]) ? 1 : 0;
  }
}
```

Optimised GPU Implementation

In order to better performance from the naive implementation, the idea was to modify the code to use a single bit of memory per cell as opposed to a whole byte. Unfortunately this introduced some computational overhead as neither CUDA nor C support allocating memory in a bitwise manner. Aside from the optimised Game of Life kernel, two further kernels were created for the purpose of encoding byte-per-cell data into bit-per-cell data and the decoding thereof respectively (Listing 4 on the following page).

The resulting implementation of Conway's Game of Life (Listing 5 on the next page) makes three separate memory accesses of one byte apiece which make more effective use of the on-chip caches of the GPU. The optimised kernel also computes the result states of a variable amount of bytes worth of cells, resulting in fewer net memory accesses. This is done at the cost of making even more use of the registers, however this should result in a net efficiency gain.

Listing 4: Bit-per-cell encoding and decoding logic (Fiser, 2013)

```
/*
 * bitLifeEncodeKernel
 * Encode the life data of 8 cells into a single byte
 */
__global__ void bitLifeEncodeKernel(
  const ubyte* lifeData,
  size_t encWorldSize,
  ubyte* resultEncodedLifeData
) {
  for (size_t outputBucketId = blockIdx.x * blockDim.x + threadIdx.x;
      outputBucketId < encWorldSize;
      outputBucketId += blockDim.x * gridDim.x) {

    size_t cellId = outputBucketId << 3;

    ubyte result = lifeData[cellId] << 7 | lifeData[cellId + 1] << 6
      | lifeData[cellId + 2] << 5 | lifeData[cellId + 3] << 4
      | lifeData[cellId + 4] << 3 | lifeData[cellId + 5] << 2
      | lifeData[cellId + 6] << 1 | lifeData[cellId + 7];

    resultEncodedLifeData[outputBucketId] = result;
  }
}

/*
 * bitLifeDecodeKernel
 * Decode the life data of 8 cells contained in a single byte into a eight
 * separate bytes.
 */
__global__ void bitLifeDecodeKernel(
    const ubyte* encodedLifeData,
    uint encWorldSize,
    ubyte* resultDecodedlifeData
  ) {

  for (uint outputBucketId = blockIdx.x * blockDim.x + threadIdx.x;
      outputBucketId < encWorldSize;
      outputBucketId += blockDim.x * gridDim.x) {

    uint cellId = outputBucketId << 3;
    ubyte dataBucket = encodedLifeData[outputBucketId];

    resultDecodedlifeData[cellId] = dataBucket >> 7;
    resultDecodedlifeData[cellId + 1] = (dataBucket >> 6) & 0x01;
    resultDecodedlifeData[cellId + 2] = (dataBucket >> 5) & 0x01;
    resultDecodedlifeData[cellId + 3] = (dataBucket >> 4) & 0x01;
    resultDecodedlifeData[cellId + 4] = (dataBucket >> 3) & 0x01;
    resultDecodedlifeData[cellId + 5] = (dataBucket >> 2) & 0x01;
    resultDecodedlifeData[cellId + 6] = (dataBucket >> 1) & 0x01;
    resultDecodedlifeData[cellId + 7] = dataBucket & 0x01;
  }
}
```

Listing 5: Optimised GPU Code for Conway's Game of Life (Fiser, 2013)

```
/*
 * bitLife Kernel
 * Compute array and bit offsets required to access all cells in the Moore
 * Neighbourhood and determine the result state of the cell under evaluation.
 *
 * This kernel is executed once per iteration by as many threads and blocks as
 * required to complete the iteration.
 *
 * The number of bytes worth of cell data that each thread processes is
 * variable.
 */
```

```
__global__ void bitLifeKernel(
    const ubyte* lifeData,
    uint worldDataWidth,
    uint worldHeight,
    uint bytesPerThread,
    ubyte* resultLifeData) {

  uint worldSize = (worldDataWidth * worldHeight);

  for (uint cellId = (__mul24(blockIdx.x, blockDim.x) + threadIdx.x)
      * bytesPerThread;
      cellId < worldSize;
      cellId += blockDim.x * gridDim.x * bytesPerThread) {

    // Calculate data offsets
    // Start at block x - 1.
    uint x = (cellId + worldDataWidth - 1) % worldDataWidth;
    uint yAbs = (cellId / worldDataWidth) * worldDataWidth;
    uint yAbsUp = (yAbs + worldSize - worldDataWidth) % worldSize;
    uint yAbsDown = (yAbs + worldDataWidth) % worldSize;

    // Initialize data with previous byte and current byte.
    uint data0 = (uint)lifeData[x + yAbsUp] << 16;
    uint data1 = (uint)lifeData[x + yAbs] << 16;
    uint data2 = (uint)lifeData[x + yAbsDown] << 16;

    x = (x + 1) % worldDataWidth;
    data0 |= (uint)lifeData[x + yAbsUp] << 8;
    data1 |= (uint)lifeData[x + yAbs] << 8;
    data2 |= (uint)lifeData[x + yAbsDown] << 8;

    for (uint i = 0; i < bytesPerThread; ++i) {
      // get the bit coordinate of the cell under evaluation.
      uint oldX = x;  // old x is referring to current center cell
      x = (x + 1) % worldDataWidth;

      // extract state of the cell under evaluation.
      data0 |= (uint)lifeData[x + yAbsUp];
      data1 |= (uint)lifeData[x + yAbs];
      data2 |= (uint)lifeData[x + yAbsDown];

      // evaluate cell iteratively.
      uint result = 0;
      for (uint j = 0; j < 8; ++j) {
        uint aliveCells = (data0 & 0x14000) + (data1 & 0x14000)
          + (data2 & 0x14000);
        aliveCells >>= 14;
        aliveCells = (aliveCells & 0x3) + (aliveCells >> 2)
          + ((data0 >> 15) & 0x1u) + ((data2 >> 15) & 0x1u);

        result = result << 1
          | (aliveCells == 3 || (aliveCells == 2 && (data1 & 0x8000u))
          ? 1u
          : 0u);

        data0 <<= 1;
        data1 <<= 1;
        data2 <<= 1;
      }

      // write result
      resultLifeData[oldX + yAbs] = result;
    }
  }
}
```

RESULTS

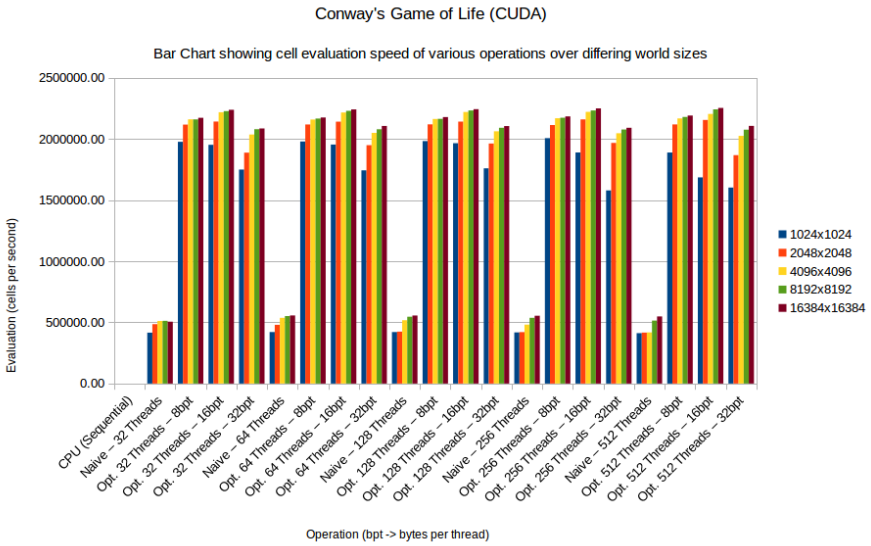

Figure 1: Results of benchmarking evalution of Conway's Game of Life

Figure 1 shows the final benchmarking results of the evaluation speed of the various operations with a variable number of threads per block, a variable number of bytes per thread for the optimised code and variable world sizes.

Each of the tests in the benchmark were conducted under the following conditions:

- Each test consisted of 10000 iterations of Conway's Game of Life.
- Each operation was executed over the following world sizes:
  - 1024 x 1024 cells
  - 2048 x 2048 cells
  - 4096 x 4096 cells
  - 8192 x 8192 cells (GPU operations only)
  - 16384 x 16384 cells (GPU operations only)
- A single board was randomly generated for each of the above board sizes and the same board was used for each test of that board size.
- Separate tests were conducted for GPU operations with 32, 64, 128, 256 and 512 threads per block for each different game board.
- Separate tests were conducted for the optimised GPU operations with 8, 16 and 32 bytes per thread for each distinct combination of thread size and game board.
- All timings obtained are purely for the computation of Conway's Game of Life and do not include any setup or encoding logic (specifically in the case of the Optimised kernel).
- Performance is measured in terms of evaluated cells per second.

As predicted, the evalution speed of the naive CUDA implementation was, at worst, **101898** times faster and **132564** times at best, than the sequential implementation; it is for this reason that the CPU does not visibly feature in Figure 1 on the previous page. The reason for this is purely that the GPU was created for and is suited to tasks that can be executed in parallel; in the event that the naive kernel was written to include significant branch divergence or needless memory waste it would still operate orders of magnitude above the sequential implementation.

When considering the naive kernel in isolation, it is seen that best evaluation speed is achieved for when using the largest of the board sizes and, with exception of 32 threads per block which performed poorly in comparison, the impact caused by varing the number of threads per block was negligible. This is due to the fact that using the largest amount of data means providing the GPU with enough work to operate at maximum possible occupancy. The reason why most of the different thread per block configurations performed similarly is likely that all of the different configurations were chosen because to be multiples of 32 which should result in the fewest wasted threads possible; the reason why 32 threads did not perform well compared to the other configurations is likely due to insufficient use of the cache resulting in additional lookups from global memory. The best results for the naive kernel were approximately **0.55M** cells evaluated per second.

Analysis of the optimised kernel can be done easily by comparing its performance to that of the naive kernel. The optimised kernel is, at worst, **3.77** times faster and **5.27** times at best, than the naive kernel; which is clearly visible from Figure 1 on the preceding page. As discussed in "Optimised GPU Implementation" on page 4, this is due to the the significant reduction in memory accesses and increased cache usage from using a single bit of memory per cell as opposed to using an entire byte to store a true of false value.

When considering the optimised kernel in isolation, the highest evaluation speed of **2.25M** cells per second was achieved with the combination of the largest board size, 512 threads per block and 16 bytes per thread. This once again is likely due to an occupancy issue where the configuration of the number of threads per block and the number of bytes per thread were simply optimal for that particular board size. It is highly likely that if the experiment had to be run with a larger board size, number of threads and bytes per thread, it may be possible to achieve an even higher occupancy rate and therefore a greater speedup.

REFERENCES

Fiser, M. (2013). Conway's Game of Life on GPU using CUDA.