FORMAL SPECIFICATION AND VERIFICATION OF OPENCL KERNEL OPTIMIZATION

Bachelor’s Thesis
of
Jeroen Vonk

UNIVERSITY OF TWENTE.

Formal Specification and Verification of OpenCL Kernel Optimization

Bachelor's Thesis

of

Jeroen Vonk

born on the 10th of June 1988 in Uitgeest

June 10, 2013

Supervisors:
Dr. M. Huisman
M. Mihelcic MSc
M. Zaharieva-Stojanovski MSc
Dr. W.K. Den Otter

Twente University
Faculty of Science and Technology
Advanced Technology

Contents

Chapter 1

Abstract

Computing general problems using the graphical processing unit (GPU) of a device is an emerging field. The parallel structure of the GPU allows for massive concurrency, when executing a program. Therefore, by executing (a part of) the code on the GPU, a previously unused resource can be used, to achieve a speed-up of an application. Previously, programming on GPUs was a tedious job, and the implementation was depending on the manufacturer - or even on the model of the GPU. With the arrival of OpenCL, an open and broad platform was offered, focussed to deliver General Purpose computing on the GPU, or GPGPU to a broader audience. Despite the sometimes simple appearance of OpenCL code, it is important to keep in mind that there can be thousands of threads running the code concurrently. All these concurrently executing threads that are potentially accessing the same memory locations, can easily lead to implementation errors. This research is focussed on verifying OpenCL code, using permission-based separation logic, to prevent those errors in an early stage. Moreover, we have investigated what are the consequences of optimizations of a OpenCL-program for the verification of that program. It is common practice to use optimization in GPGPU, since the code executed on the GPU is often "resource-hungry", either for memory, processing power, or both. Therefore, optimizing the GPGPU part of a program will often result in a significant speed-up.
As a verification use-case, we have developed a simple implementation of Conway's Game of Life, a well-known zero player game, based on a cellular automaton. We have verified this implementation using permission-based separation logic, enriched with some rules specifically for OpenCL. Therefore, we had to annotate the code in a similar way when using the VerCors tool-set. Furthermore, we developed three optimizations of this code using common optimization techniques. To verify each of the optimizations we have looked at the changes needed in the verification, in relation to the original verification. Our optimized versions, upon execution, are indeed faster than the original implementation. Moreover, we can show several patterns for changing our verification to fit our optimization. Using these patterns, one could possibly automatically optimize OpenCL code, whilst still guaranteeing the correctness of the program, given that the previous implementation was correct.

Chapter 2

Introduction

A lot of everyday puzzles and in nature occurring phenomena look rather complex - but do actually oblige to a simple set of basic rules. Some examples are problems of finding the shortest path (e.g. in train planning or behaviour of an ant colony). A way to solve such problems by use of a computer is by dividing the problem in smaller (simple) sub-problems. In this way, a very complex looking problem can be split into more manageable sub-problems. Each sub-problem can then be solved with relative ease and will often have more lenient requirements regarding memory or processing power.

All these sub-problems together still require a tedious amount of calculations. Luckily, the processing power of computers still increases; however, single chips are approaching their frequency limits and the focus of fast computing shifts towards parallel processing on multiple processor-cores. Another possibility to speed up computing is by making use of the Graphical Processing Unit (GPU) when computing parallel tasks. The GPU is highly optimized for processing a huge number of identical tasks running in lockstep, up to thousands at a time. This approach is also known as SIMD or SIMT (single instruction; multiple data or multiple thread).

2.1 GPU Programming

This field of General-purpose computing on GPU, also known as GPGPU or GP2U, is often fitted for computing earlier stated problems [32]. This is because these problems can be split up in a set of smaller identical sub-problems, each assigned to a thread, processing a different data set - the basic premise for using SIMD. The strength of GPGPU, being able to run thousands of threads in parallel, also comes with its drawbacks. Thousands of interleaving threads all potentially accessing the same data and waiting on each other can lead to unexpected problems [16]. Errors caused by data races are known to actually occur on seemingly random moments. By the time these errors occur the software may be already in a production environment. Bug hunting and solving errors in a production environment is very expensive [31]. As a result, it is important to prove that a GPGPU-program is correct i.e. that the program satisfies its pre-defined behaviour.

2.2 Correctness of a Program

The field of formal verification allows us to prove the correctness of a program. By analysing the code or by running it with specific tools a program can be proved to be correct. In this project we describe a manual way of verifying GPGPU-programs, and which extra operations are needed to make the code verifiable. Therefore we will introduce a sample code.

In addition, we looked at optimizations that we can apply to our sample code. These optimizations can range from using smarter algorithms to memory optimizations [13]. Optimizations, however, have the nature to make the code less transparent which increases the possibility for errors. Those errors might lead to incorrect results, for example by floating point errors. Harder to catch are racing conditions caused by badly distributed access to memory locations, e.g. a thread having read access to a location to which another thread has write access, can lead to unexpected results depending on the execution order of both threads.

The error-proneness of optimizations gives us additional motivation for developing a formal verification tool for GPGPU-programs [20].

2.3 Conway's Game of Life

For this project we have implemented, on the GPU, a game called *Conway's Game of Life*. This game consists of a two-dimensional, infinitely large grid. Each square, or "cell", in the grid can either be alive or dead. The state of this cell depends on the current state of the eight cells surrounding it.

The game of life is a well-known example of a so called cellular automaton. Cellular automatons are broadly applicable and can be used for simulations in the field of biology, physics and or computer science [33]. This automaton, Conway's Game of Life, is rather simple but still allows for a high degree of parallelization, because each cell can be calculated by a parallel thread and only requires access to the last state of its surrounding cells.

This parallel nature and the fact that after the initial state the game does not require any user input makes it a suitable problem for this project.

Another advantage is that this problem allows for multiple optimizations. The concrete optimizations we have implemented and their effect will be explained in detail in Chapter 5 and further.

2.4 The Project

We have verified our Game of Life implementation. After proving the correctness of this code, we applied a specific optimization and verified this new implementation, we iterated this process for several optimizations. With these multiple verifications we are also interested in the effect of our optimizations on the verification of our code in respect to the verification of the original implementation.

These can be trivial with simple GPGPU-programs. However, when we deduce some rules about certain optimizations, those optimizations can be directly applied in the future, without the necessity to prove the correctness of the complete program again.

Being able to automatically add such optimization without introducing errors in the code will decrease verification and programming time of GPGPU programs. Additionally, these optimizations can even be used in automatically generated GPGPU-code. This however is out of the scope of this project.

In summary, the focus of this project lays on the verification of GPGPU-code; therefore, we have written an implementation of the Game of Life and verified it to be correct. In Chapter 3 and 4 we will explain our verification method and its background. We are also interested in which optimizations we can apply and their effect on the verification of our code in respect to the verification of the original implementation (Chapter 5 - 8). Chapter 9 and 10 contain related work, the results, and the conclusions, where we hope to sufficiently answer the following research questions:

- How can we formally specify and reason about programs implemented on GPU devices?
- Which common optimizations can we use for such programs?
- What are the effects of code optimization on its verification and vice versa?

Happy reading,
- Jeroen

Chapter 3

Background

3.1 OpenCL

We verified GPGPU-programs written in OpenCL. OpenCL is an open standard developed by the Khronos group [24]. To be exact, OpenCL is a royalty-free standard for general purpose parallel programming across CPUs, GPUs and similar devices. The way the OpenCL platform is set up makes it possible to execute a single implementation on a broad range of GPUs and even on some CPUs [22][1][6]. This property makes OpenCL very suitable for our project. The fact that OpenCL is designed for a broad range of devices, makes it both practical to use for us and possibly a common standard for years to come. Therefore, focusing research on OpenCL is potentially more valuable than research of some other GPGPU-language e.g. CUDA, a language specifically targeted at the NVidia hardware [23][10].

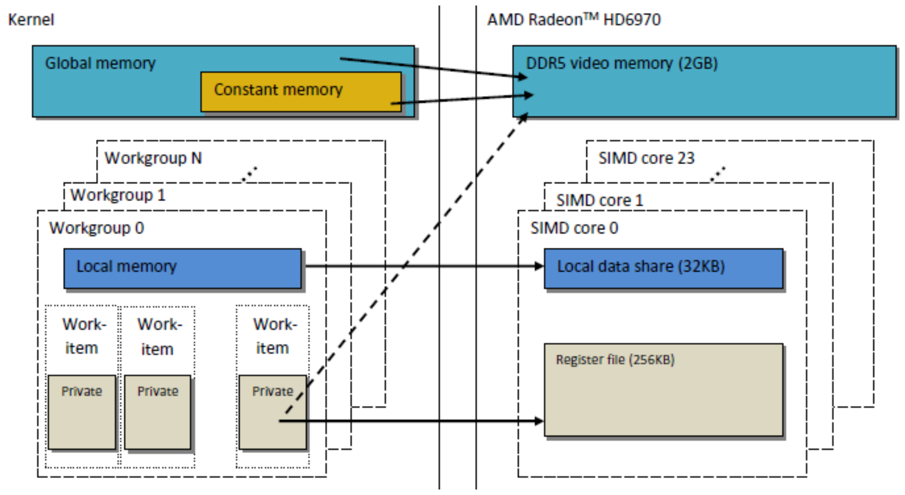


Figure 3.1: OpenCL structure (in relation to a graphics card)

OpenCL-code consists of the *host code* and the *kernel code*, the host code is a regular program that sets up some parameters and executes the kernel code. After execution the result of the calculations by the kernels can be retrieved by the host [2]. This kernel code is executed on the GPU multiple times in parallel in separate threads, each instance having a unique identifier, *thread id* (*tid*) ranging from 0 to *gid_size*-1, with *gid_size* being the total amount of threads. The threads can access these values in run time by using the *get_global_id(0)* and *get_global_size(0)* functions, respectively. Multiple instances of the kernel are grouped in so called *work groups.* This structure, as shown on the left side in figure 3.1, is chosen by the Khronos group because it is very similar to the memory structure of the GPU. A GPU consists of multiple streaming multiprocessors, each multiprocessor is able to run several threads, similar to the mapping of threads in work groups as seen in OpenCL. Each streaming multiprocessor is supplied with local memory, that can be shared by the threads ran by that multiprocessor. Figure 3.1, shows a schematic view of a GPU in relation to the OpenCL structure.

3.2 Permission-based Separation Logic

To verify our sample program we have used a version of permission-based separation logic. To understand the basis of permission-based separation logic a short history is needed.

The formal specification of programs basically started with David Hilbert, the founder of mathematical field called 'logic'. Robert Floyd pioneered using this logic to reason about programs. He did this with so called assertions; statements that are true at specific program locations.

In 1969 Sir Tony Hoare proposed the Hoare-Floyd logic, or now commonly known as Hoare Logic [18]. The Hoare Logic consists of the Hoare Triple and a set of rules. A Hoare Triple consists of a piece of code, or a command (S), and two assertions: the pre-condition (P), and the post-condition (Q), written: $\{P\}S\{Q\}$. The rules specify the relations between several Hoare Triples e.g.

$$\frac{\{P\}S\{Q\}, \{Q\}T\{R\}}{\{P\}S;T\{R\}}$$

This rule states that we can combine the two commands S and T, when the pre-condition of T is the same as the post-condition of S.

With the coming of pointers and heaps Hoare Logic needed an addition. John Reynolds *et al.* provided this in the form of Separation Logic [29]. Separation logic provides a way to reason about pointers that may point to the same location in the memory. Two main operators are `*`(separating conjunction) and `-*`(separating implication). These operators are used to prove that two pointers point to distinctive locations.

Separation Logic is also useful for reasoning about parallel programs. Logically, when two threads never access the same variables, they do not interfere with each other.

This is rather restrictive, for example, two threads will never enter a data race when they access the same variable only for reading. This is solved by the introduction of *permissions*, known as: permission-based separation logic. A pointer x to a location v is assigned a permission π in the domain $(0, 1]$, or: `PointsTo`(x, π, v) [17]. The sum of all the permissions to one location never exceed 1. A thread needs permission to be able to read a variable, and iff a thread has a permission equal to 1 it is allowed to write to the variable.

3.3 VerCors

Hurlin and Huisman used permission-based separation logic to verify a Java like language including the support for parallelism [21]. In relation to this work is VerCors, an ongoing project on the University of Twente with the goal to develop a specification language, program logic for concurrent programs and concurrent data structures[3]. Subsequently they want to make the program logic applicable by building a tool set implementing this program logic. The VerCors project uses permission-based separation logic and in order to verify a program, it requires the user to add annotations in the code. These annotations are written in-line in a JML-like style. VerCors is currently suited for use with concurrent Java programs.

In parallel with this project, a formal method for OpenCL verification is devised. Mihelcic, Huisman and Blom are currently working on extending the VerCors approach and the tool set to include verification of OpenCL code. As stated above, verification of GPGPU-programs is different from concurrent programs; GPGPU code is usually constrained to a few thousand lines of code and forks and joins are not common practice. However, GPGPU-programs can easily be executed concurrently with more than a thousand threads; moreover each of these threads can potentially read and write to the same memory locations. Complexity of the verification can thus potentially be $O(k) = n^{k*1000}$ in comparison to two concurrent similar threads, when one would be using dynamic verification. The static verification is realized with the permission-based separation logic provided by the VerCors project. However, the memory model and executional structure of OpenCL-threads require additional properties to be proven such as the behavior of work groups, barriers and the access to memory locations [20][19].

3.4 Game of Life

The Game of Life is a well known and studied cellular automaton (CA) [5], however, although the basic premises for the game are simple, the implementation can get rather complicated, due to its possibly infinite playing field. Algorithms to circumvent this are developed[11] with the best known one being HashLife described in the 1980s by Gosper[15].

The general rules of the Game of Life are as follows [33]. The Game of Life happens on a two-dimensional infinite grid. Each square on the grid represents a cell, the state of a cell can be either *dead* or *alive*. A cell on the grid has 8 (direct) neighbours. The game is a zero player game. The goal of the game, once an initial field is loaded, is to calculate the next generation of the field based on its previous state. The state of an individual cell can be determined based on: a) the current state of the cell, and b) the amount of neighbours that are currently alive. Those rules are:

- A live cell with two or three live neighbours stays alive
- A cell with exactly three live neighbours becomes/stays alive.
- In all other cases, < 2 or > 3 live neighbours, the cell dies.

An example of a succession of iterations can be seen in figure 3.2.

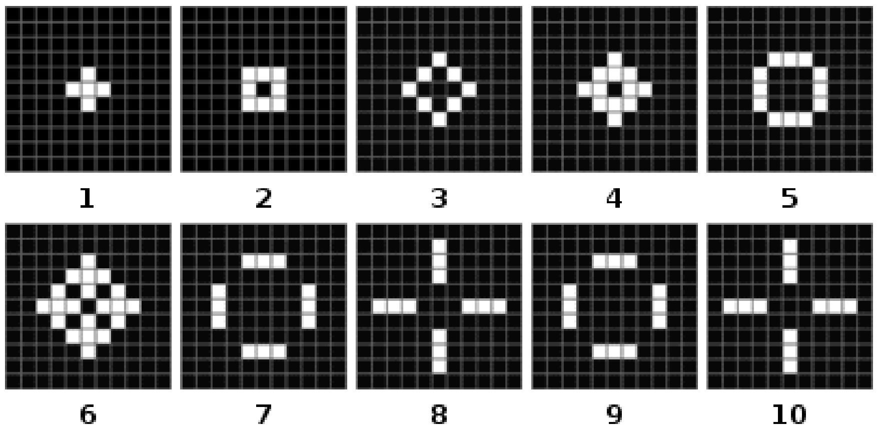


Figure 3.2: Example of a succession of iterations

The Game of Life has also been implemented on multiple occasions to run on GPU's[30][28][32] - some even implementing versions of HashLife[25].

For the scope of this project, being mainly on verification of possible optimizations, a simple implementation of the Game of Life will be used.

Chapter 4

Research Method

For the verification of the Game of Life implementation, we have chosen for a straightforward implementation. This first implementation is rather naive, but allows us to apply various optimizations in the later stages of the research. The exact implementation and the design choices are explained in the section below.

4.1 Implementation

The main problem in any implementation of the Game of Life is the infinite size of the playing field. There are several ways to deal with this. One option is storing only the cells that are "alive". Storing only the live cells allows for a huge improvement in memory usage, however, this will require a data structure different from a two-dimensional grid, resulting in the need for a fast search algorithm to find neighbouring cells. Another option is to use a $X * Y$-sized array of a finite size. To cope with the case where live cells reach the edge of this array there are two common options: either when live cells reach the outer borders of this grid they can "wrap around" or they can "die" on those edges. An easy way to visualize this is that the field is a plane where on the outer edges the cells die or that the field is a torus, allowing for the wrapping around. The scope of our research made us decide to use the $X * Y$-sized array approach with "dying cells" at the edges.

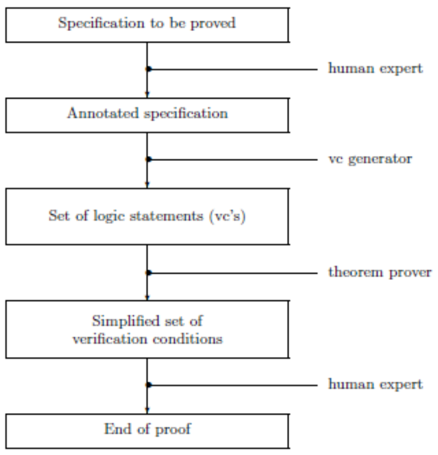
4.1.1 Host and Client Side

Implementing this in OpenCL requires an additional step in the design process. OpenCL requires us to split up our problem in two parts; the part running on the host side and the part of the algorithm running on the GPU (or client side). The host side of our implementation is written in C++, a general version can be found in appendix A. The host code is designed for setting up and launching the OpenCL kernel, and measuring the timing of several actions of this kernel. The host side code is based on the samples by AMD, the changes we have applied are related to doing measurements, the used data structure, loading and saving the data. The only way of communication between the host side and kernels on the OpenCL-devices is by using this data structure. The host side initializes a buffer containing the field with the initial state of the game and provides an empty buffer for the resulting game after the iteration. After the execution of the kernels these buffers are passed back to the host side. The kernel code is executed by the OpenCL-device. This is done many times in parallel. Based on the threadid a kernel thread is given, it calculates the next generation of a specific cell by looking at the current state of its neighbours. This result is then written to the memory allocated by the host side. After the execution of all the kernels the host can read the result, the next generation of the initialized board. In a nutshell this is our implementation. We have made several optimizations, each can be seen in their corresponding chapters. Each optimized version is based on the previous implementation and we tried to apply our optimizations with as few changes as possible.

4.2 Manual Verification

Each version of our Game of Life implementation is manually checked for correctness. For this manual checking we have devised a procedure based on two main sources. For the permission-based separation logic we have used the work in progress by Mihelcic and Huisman [20], which we briefly described in Chapter 3. The manual checking process is based on the lecture notes by Gordon [14].

Gordon's work describes mechanizing program verification. Mechanized verification may seem contradictory with the goal to manually verify our code. Having a strict, or mechanized, procedure to verify our code may be beneficial for future implementation, and reduces human errors in the, currently, manual verification. In the future use, people can follow the same procedure and yield the same result or even implement the steps in a program to automate the verification. The machine verification described by Gordon allows us to structurally analyze the code with the help of annotations written in line. Additional verification rules are introduced by Mihelcic and Huisman [20].


Specification to be proved
human expert
Annotated specification
vc generator
Set of logic statements (vc's)
theorem prover
Simplified set of verification conditions
human expert
End of proof
Figure 4.1: Verification steps in machine verification

The machine verification consists of several steps. First, the user has to annotate the code defining the pre- and post-conditions of the commands at predefined points. These annotations can be translated to a set of logic statements, or verification conditions, see figure 4.1. These verification conditions (or VCs) can be condensed and simplified to a form that can be inspected for its correctness. The verification conditions are generated by standard rules posed in Gordon's material. These rules for translation of code and annotations to VCs are based on Floyd Hoare Logic. Permission-based separation logic is based on Hoare Logic, this makes it possible to introduce some additional statements from the work of Mihelcic and Huisman. Additional statements are related to the use of barriers and memory structures in OpenCL; Also, the pre- and post-conditions for the kernel code need to be done separately since the verification is done on both the Thread, and Kernel level. The double pre- and post-conditions allow us to detect possible data races on the kernel level. When working with multiple work groups an additional pre- and post-condition should be introduced for the verification on the work group level.

Chapter 5 shows in detail how this is done in practice.

4.3 Optimizations

The optimizations that we made are common when optimizing programs to run on a GPU [13]. We have used the following optimizations:

- Our main implementation uses loop unrolling, a common opportunity for optimization [13], we have used it for reading the neighbouring values in the grid. Unrolling the commands in a loop leads to an increase of the size of the code, but can lead to significant speed-ups at execution time.
- In our first optimization we have limited the amount of threads that need to be initialized, thus avoiding the need to reinitialize the buffers needed for every iteration of the Game of Life. This is realized by giving each thread more cells to calculate, skewing the ratio of initialization time vs. execution time in our favour.
- In our second optimization we have included barriers and a second iterator. This results in a situation where we only have to start our kernel once.
- The third optimization uses the local memory of work groups; the local memory can be factors faster than the global memory. Therefore, the data is copied from the global memory to the local memory, where the data can be manipulated, and then the resulting values are copied back to the global memory. Even though the data are copied two times more, this overhead is often quickly compensated by the speed-up achieved using the local memory.

4.4 Platform

The first implementation was done on a Dell Inspirion 6400 running 32-bit Windows 7, a rather outdated laptop with 2GB of memory, a 2GHz processor, with no hardware support for multi-threading; and an unsupported graphics card. In our first implementation, this was not a real problem and it illustrated the versatility of OpenCL, it was clearly possible to compile and execute the OpenCL code on this machine. However, not using a dedicated GPU made it impossible to use the classical optimizations and trade-offs that come with a GPU. For example, the use of local memory vs global memory clearly revealed that. Therefore, after this initial acquaintance with OpenCL, we moved to a more mature platform. A 64-bit machine running Scientific Linux with two quad-core Intel Xeon Processors at 2.40GHz and a Tesla S2050. The Tesla S2050 consists of 4 NVIDIA Tesla M2050-cards each with up to 1TFLOP of peak performance. This was suitable for our research, even forcing us to run the Game of Life for a considerable amount of iterations, to allow for a reliable timing of our program execution.

Chapter 5

Main Implementation

```c
__kernel void kernel(___global  unsigned int * nextgen,
                     ___global  unsigned int * board,
                     const      unsigned int height,
                     const      unsigned int width)
  {
  int pos, up, down, outofbounds, neighbours;

  pos   = get_global_id(0);
  up    = pos - width;
  down  = pos + width;

  outofbounds   = (pos < width);          //upper edge
  outofbounds  |= (pos > (width * (height-1)));//lower edge
  outofbounds  |= (pos % width == 0);       //left edge
  outofbounds  |= (pos % width == width-1);  //right edge

  if (outofbounds)
    {
    nextgen[pos] = 0;
    }
  else
    {
    neighbours    = board[up-1]   +board[up]   +board[up+1];
    neighbours   += board[pos-1]               +board[pos+1];
    neighbours   += board[down-1] +board[down] +board[down+1];

    nextgen[pos] = (board[pos] && neighbours == 2) || (neighbours == 3);
  }
}
```

Listing 5.1: Main Implementation[1]

[1]The fully annotated version can be found in appendix A.1

5.1 Implementation

Our basis implementation, see above or in appendix A.1, is a fully functional implementation of the Game of Life. This code consist of a host side program compiling and initializing the OpenCL implementation of the Game of Life. The kernel consists of OpenCL-code that implements a straightforward implementation of one iteration of the Game of Life. The only aspect of this code that can be considered an optimization is the checking for the neighbouring cells on lines 21-23, this could be done with a loop. Technically, we have used loop unrolling in this implementation. This first implementation will henceforth be referred to as the Single Iteration kernel, or in shorthand 01_SI. An uncommented and un-annotated version of this kernel can be seen in the snippet at the beginning of this chapter.

The host side code, which can be found in appendix B, is based on the framework given by the AMD OpenCL SDK. The host side compiles the OpenCL code and configures some parameters. Part of these parameters determine how the OpenCL kernel is executed: the amount of threads and how they are grouped. The other parameters are the variables that are provided to each kernel, in our case those parameters are: *board*, *nextgen*, *height*, and *width*. *Board* is an one-dimensional integer array containing the initial state of the Game of Life, 0 stands for "dead" and 1 for "alive". The *width* and *height* parameters are provided to translate the one-dimensional array to a two-dimensional field in each kernel. Next, the host side launches the OpenCL kernel with given parameters and the GPU calculates one step in the Game of Life. Using the get_global_id(0)-function each kernel gets its unique (global) threadid, from which it can calculate to which field on the board the id corresponds (line 6). When the field is on the edge of the board it will automatically die, this can be seen on lines 10-18. The last parameter provided to the kernel is *nextgen*, which contains the resulting board after one iteration of the Game of Life. The host uses an optimized command to swap the input buffer (*board*) containing the initial state of the automaton with the output (*nextgen*) and reissue the command for execution of the kernel. After a set of iterations the result will be loaded from the output buffer and be used on the host side.

Our first implementation used boolean arrays, however the implementation of booleans in OpenCL depends on the used platform, therefore, to make the code work on the Tesla we needed to resort to the more resource consuming integer arrays.

5.2 Verification

To verify our code we need to provide certain annotations at certain places. First of all, it needs to be annotated at the beginning and at the end of the kernel. Usually this needs to be done on the thread $(T_{res}, T_{pre}, T_{post})$, work group $(W_{res}, W_{pre}, W_{post})$ and kernel level $(K_{res}, K_{pre}, K_{post})$. Since we are using only one work group, $(W_{res}, W_{pre}, W_{post})$ will be identical to $(K_{res}, K_{pre}, K_{post})$. Therefore, we can omit our work group specification. The other places where the code needs te be annotated are explained by Gordon:

> *A command is said to be properly annotated if statements have been inserted at the following places:*

- *Before each command C_i (where $i > 1$) in a sequence "$C_1; C_2; :::$ $; C_n$" which is not an assignment command,*
- *After the word **DO** in **WHILE** and **FOR** commands.*

Next, we split up the code of the Single Iteration code to its separate components as can be seen below, according to Gordon's work.

```
{ANNOTATION_A}
C - BLOCK
  BEGIN
  VAR pos;
  VAR up;
  VAR down;
  VAR outofbounds;
  VAR neighbours;
  C - Sequence
    C - Assignment;
      pos = get_global_id(0);
    C - Assignment;
      up = pos - width;
    C - Assignment;
      down = pos + width;
    C - Assignment;
      outofbounds =               (pos < width);
    C - Assignment;
      outofbounds = outofbounds | (pos > (width * (height-1)));
    C - Assignment;
      outofbounds = outofbounds | (pos % width == 0);
    C - Assignment;
      outofbounds = outofbounds | (pos % width == width-1);
    {ANNOTATION_B}
    C - Two armed conditional;
      IF outofbounds
      THEN
      C - Assignment
        nextgen[pos] = 0;
      ELSE
      C - Sequence
        C - Assignment
          neighbours =            board[up-1]    +board[up]     +board[up
              +1];
        C - Assignment
          neighbours = neighbours +  board[pos-1]             +board[pos
              +1];
        C - Assignment
          neighbours = neighbours +  board[down-1] +board[down]  +board
              [down+1];
        C - Assignment
          nextgen[pos]= (board[pos] && neighbours == 2) || (neighbours
              == 3);

  END
{ANNOTATION_C}
```

Listing 5.2: 01_SI - split up

Annotation_A and Annotation_C will be respectivly T_{pre} and T_{post}. Annotation_B can be seen as both a pre-condition for our conditional and a post-condition of the assignments in the code.

Annotation B and $(T_{res}, T_{pre}, T_{post})$ can be found in the appendix A.1 and are:

```
// Tres - resources needed for the thread
  //@ requires perm(width,p) ** perm(height,p);
  //@ requires !oob(gtid) ==> perm(board[gtid-width-1],p) ** perm(board[
      gtid-width],p) **
  //@   perm(board[gtid-width+1],p) ** perm(board[gtid -1],p) ** perm(
      board[gtid+1],p) **
  //@    perm(board[gtid+width-1],p) ** perm(board[gtid+width],p) ** perm
       (board[gtid+width+1],p)
  //@ requires perm(nextgen[gtid],1)

// Tpre - preconditions for the thread
  // The amount of threads must equal the size of the boards
  //@ requires (width*height) == gtid_size;
  //@ requires sizeof(board)==sizeof(nextgen) && sizeof(board)/sizeof(
       int)==(width*height);
// Tpost
  // if we are out of bounds, the cell is always dead. Otherwise the
       result should be the result of the rules of the Game of Life
  //@ ensures oob(gtid) ==> nextgen[gtid]==0;
  //@ ensures !oob(gtid) ==> nextgen[gtid]==gol(gtid)
```

Listing 5.3: Thread specification

```
//@ assert outofbounds == oob(gtid)
//@ assert up = pos - width;
//@ assert down = pos + width;
//@ assert pos = gtid;
```

Listing 5.4: Annotation_B

As we can see from our thread specification T_{pre}, only contains a precondition about the size of the board, thus we can assume Annotation_A to be *true*. Therefore, there are no specific conditions for the population of the board. Annotation_B mostly says that all the variables used in our implementation need to have the correct value. The names *oob* and *gid* are model variables, as explained in listing 5.5. Annotation_C, or T_{post}, guarantees that the next generation for a cell is calculated, except for cells that lay on the border of the field: cells on the border of the field will always die, as previously specified.

Now that we have set up our annotations, we can construct our Verification Conditions or VC's. We will do this with the classical Floyd-Hoare logic used by Gordon; in parallel we will check for the permission-based separation part. By separating these aspects we can make the manual verification easier. To simplify the verification with respect to this project's scope, a part of the proof will be informal, mostly, the permissions redistribution at barriers and the verification of the kernel.

In our annotations we use JML-variables [26]. The use of these model variables gives a better insight during manual verification. However, these variables are simply shorthands and the use of these variables do not influence the verification.

Our model-variables are:

```
// gtid is used as shorthand for the unique (global) thread identifier.
//@ private model int gtid;
//@ private represents gtid <- get_global_id(0);
// gtid_size returns the total amount of threads.
//@ private model int gtid_size;
//@ private represents gtid_size == get_global_size(0);
// oob stands for Out Of Bounds, and tells us whether cell i, is on the
     border of the field.
//@ private model function oob(i);
//@ private represents oob(i) <- (i < width)||(i > (width * (height-1)))
     ||(i % width == 0)||(i % width == width-1);
// nb, gives the amount of direct, live neighbours of cell i.
//@ private model function nb(i);
//@ private represents nb(i) <- board[i-width-1]+board[i-width]+board[i-
     width+1]+board[i-1]+board[i+1]+board[i+width-1]+board[i+width]+
     board[i+width+1];
// gol returns the expected state of cell i, given the current board,
     respecting the rules of the game of life
```

```
//@ private model function gol(i);
//@ private represents gol(i) <- (board[i] && nb(i) == 2) || (nb(i) ==
    3)
```

Listing 5.5: 01_SI - Model variables

- *gid* is the global identifier or *threadid* of the thread
- *oob* stands for "out of bounds" and describes whether the cell is on the border of the field.
- *nb(i)* represents the count of live neighbours of cell *i*
- *gol(i)* calculates the state of a given cell *i* when the rules of the Game of Life are applied upon it.
- *gid_size* returns the total amount of threads executed.

Next, we introduce several snippets where we show the construction of the VC's. Annotation_A only tells us that the amount of threads should be equal to the size of the board, so for the sake of this part we can say that Annotation_A is *true*. Our model variables is assumed to be part off the context in our verification, so it will not be explicitly mentioned in every pre- and post-condition.

```
{true}
  C - Assignment;
    pos = get_global_id(0);
  C - Assignment;
    up = pos - width;
  C - Assignment;
    down = pos + width;
  C - Assignment;
    outofbounds =                (pos < width);
  C - Assignment;
    outofbounds = outofbounds | (pos > (width * (height-1)));
  C - Assignment;
    outofbounds = outofbounds | (pos % width == 0);
  C - Assignment;
    outofbounds = outofbounds | (pos % width == width-1);
{(outofbounds == oob)*(up = pos - width)*(down = pos + width)*(pos = gid
    )}
  C - Two armed conditional;
{(oob*nextgen[gid]==0)||(!oob*nextgen[gid]==gol(gid))}
```

Listing 5.6: 01_SI - Condensed

We can now apply the assignments, for example:

$$\frac{\{\textit{true} * \text{gid} = \text{get\_global\_id}(0)\}\text{pos} = \text{get\_global\_id}(0)\{(\text{pos} = \text{gid}) * ....\}}{\textit{true} * \text{gid} = \text{get\_global\_id}(0) => ((\text{pos} = \text{gid}) * ....)[\text{get\_global\_id}(0)\backslash\text{pos}]}$$

which leaves us with the proof obligation:

$$\text{gid} = \text{get\_global\_id}(0) => ((\text{pos} = \text{gid}))[\text{get\_global\_id}(0)\backslash\text{pos}]$$

or simply:

$$\text{gid} = \text{get\_global\_id}(0) => \text{get\_global\_id}(0) = \text{gid}$$

The other assignments are proved in a similar manner. This leaves us with the verification of the conditional.

```
{ANNOTATION_A}
.
.
.
{ANNOTATION_B: (outofbounds == oob)*(up = pos - width)*(down = pos +
    width)*(pos = gid)}


{ANNOTATION_B && outofbounds}
  C - Assignment
{(oob*nextgen[gid]==0)||(!oob*nextgen[gid]==gol(gid))}

{ANNOTATION_B && !outofbounds}
  C - Sequence
    .
    .
    .
{(oob*nextgen[gid]==0)||(!oob*nextgen[gid]==gol(gid))}
```

Listing 5.7: 01_SI - conditional

The **outofbounds**=*true* arm of the conditional results in the following proof obligation:

$$\{\text{outofbounds}(\text{outofbounds} == \text{oob}) * (\text{pos} == \text{gid}) * ....\}$$
$$\text{nextgen[pos]} = 0$$
$$\{(\text{oob} * \text{nextgen[gid]} == 0)||....\}$$

The *false* arm is a similar procedure, except that it demands more substitutions.

5.2.1 Proving the Correctness of the VCs using Permission-based Separation Logic

Now we have constructed the two VCs as specified by Gordon [14]. With the Annotations inserted in these VCs, we can combine them with Huisman and Mihelcics work [20]. According to Huisman and Mihelcic we need two triples, one for the kernel and one for our thread specification. The kernel specification $(K_{res}, K_{pre}, K_{post})$ consists of the pre-condition (K_{pre}) and a postcondition (K_{post}) alongside of K_{res}, which represents all the resources provided by the host side to the kernel. The thread specification $(T_{res}, T_{pre}, T_{post})$ is quite similar to the kernel specification. Only the thread specification is in relation to every thread. So T_{pre} and T_{post} specify the pre- and postcondtions for each separate thread. And T_{res} expresses the global and local recources allocated for each thread. To prove the correctness of our code, we take the following steps:

1. Check the VC's against the Thread (T_{pre} & T_{post} used for Annotation_A and Annotation_C)
2. Check the used variables in the code against T_{res}
3. Check if the total of $(T_{res}, T_{pre}, T_{post})$ for all threads accumulates to $(K_{res}, K_{pre}, K_{post})$

1. Check the VC's against the Thread

Mihelcic and Huisman proposed

$$\{T_{res} * T_{pre}\}\ K_{body}\{T_{post}\}$$

to be proven correctly by using standard rules for permission-based logic. In our case we will prove, using Hoare logic, that $\{T_{pre}\}\ K_{body}\{T_{post}\}$ is true. Therefore, we still will have to prove that there are no resource conflicts, which we will do in step 2.
Above we have checked whether $\{T_{pre}\}\ K_{body}\{T_{post}\}$ holds, with help of the rules provided by Gordon. The next step is checking whether T_{res} is sufficient.

2. Check the used variables in the VCs against T_{res}

We will check for resource conflicts by analysing for each used variable whether the thread has sufficient permissions to access this variable (as stated in T_{res}). A check is needed to verify that two threads do not access the same local memory. The following formula states that if a thread has write permission for each variable in the local memory, than these accumulated rights will be sufficient to satisfy all the permissions, for local variables, needed by all the threads together.

$$\mathop{\ast}_{v\in Local} \texttt{Perm}(v,1) \mathrel{-\!\!\ast} \mathop{\ast}_{ltid\in LTid} T_{res|loc}$$

Additionally, we will check whether a thread does not has more permissions than needed in order to keep T_{res} as minimal as possible.
The code shows that the permissions for all the locations we used in the kernel are correctly allocated in T_{res}. Moreover, none of the threads share local memory (on the work group level). We can state that T_{res} is respected by the code and that we do not have any resource conflicts on the kernel level. Our additional check is to see whether T_{res} is as strict as possible, this is not an actual requirement for verification, but doing this is more likely to catch possible faults when a thread accidentally writes to an unintended location. In our specification T_{res} is strict. Since we only give permissions to locations used by the thread. Additionally, write permission is only given if a thread actually writes to a location.

3. Check if the total of $(T_{res}, T_{pre}, T_{post})$ for all threads accumulates to $(K_{res}, K_{pre}, K_{post})$

At the beginning we have to check if all resources allocated to the kernel are sufficient for all $T_{res|glob}$. Additionally, we have to check if all the preconditions (T_{pre}) summate to K_{pre}. Formally, this can be done by proving the following formula to be true. $K_{res}\&K_{pre} \mathrel{-\!\!\ast} \mathop{\ast}_{tid\in Tid}(T_{res|glob}\&T_{pre})$

(In our project, however, this is done in a more informal manner.)
The last check we need to do is checking T_{post} against K_{post}. We do this by proving that the disjoint set of al post-conditions for all the threads implicate K_{post}. $\mathop{\ast}_{tid\in Tid} T_{post} \mathrel{-\!\!\ast} K_{post}$
Our pre-conditions for both the Thread and the kernel are *true*, resulting in:

$$true\&K_{res} \mathrel{-\!\!\ast} \mathop{\ast}_{tid\in Tid}(true\&T_{res|glob})$$

Where *Tid* is a set of natural numbers in the range [0..get_global_size(0)).

$$\mathop{\ast}_{i\in\{0..global\_size\}} \texttt{Perm}(\text{next\_gen}[i],1) \mathrel{-\!\!\ast} \mathop{\ast}_{tid\in Tid} \texttt{Perm}(\text{next\_gen}[tid],1)$$

This, of course, is correct. The permissions for **board** look a bit more complicated. However, the specification on the thread level simply says:

- Iff the thread calculates a cell on the border of the field, it does not need access to **board**. This is because we already decided that all cells on the border die.
- Otherwise we need $\texttt{Perm(}board[n], \pi\texttt{)}$ with n being the id of all direct neighbours of the cell represented by this thread.

By specifying K_{res} to contain $\mathop{\ast}_{tid \in Tid} \texttt{Perm(}\mathrm{board}[tid], \pi\texttt{)}$ does guarantee this in the strictest manner.

Checking whether $\mathop{\ast}_{tid \in Tid} T_{post} \texttt{-*} K_{post}$ is true requires us to prove the following:

$$\mathop{\ast}_{i \in \{0..gid\_size\}} oob?nextgen[i] == 0 : nextgen[i] == gol(i) \texttt{-*}$$
$$\mathop{\ast}_{tid \in Tid} ((\mathrm{oob} * \mathrm{nextgen}[tid] == 0)||(!\mathrm{oob} * \mathrm{nextgen}[tid] == \mathrm{gol}(tid)))$$

5.3 Conclusion

This concludes the verification of our main implementation. Manual verification of this seemingly simple kernel, even in an informal manner, still requires an awful lot of writing. Automatic verification and maybe even annotation of kernels would be preferable if one would want programmers to make verification of their code a common practice. Since OpenCL kernels usually contain a limited amount of code, it is natural that a programmer would not want to put a tedious amount of time in annotation and formally verifying its code. The catch with OpenCL-code is that it is very profitable to optimize one's code, resulting in a complicated code, and introducing possible errors. In the next chapters we analyse several optimizations of the current implementation; at each optimization we focus on what we have actually changed in our code, and how it changes our verification. In those chapters we will explore the possibility of a "blueprint" for similar optimizations, allowing to speed up manual verification, and in the future maybe even automatic verification of OpenCL code.

Chapter 6

Thread Count Optimization

```
__kernel void kernel(__global  unsigned int * nextgen,
                     __global  unsigned int * board,
                     const     unsigned int height,
                     const     unsigned int width)
  {
  int pos, up, down, outofbounds, neighbours;

  for (pos = get_global_id(0);pos<width*height;pos += get_global_size(0))
    {

    outofbounds     = (pos < width);
    outofbounds    |= (pos > (width * (height-1)));
    outofbounds    |= (pos % width == 0);
    outofbounds    |= (pos % width == width-1);

    if (outofbounds)
      {
      nextgen[pos] = 0;
      }
    else
      {
      int neighbours = board[up-1]   +board[up]   +board[up+1];
      neighbours    += board[pos-1]               +board[pos+1];
      neighbours    += board[down-1] +board[down] +board[down+1];

      nextgen[pos] = (board[pos] && neighbours==2) || (neighbours == 3);
      }
    }
  }
```

Listing 6.1: Thread Count Optimization[1]

[1]The fully annotated version can be found in appendix A.2

6.1 Implementation

In our main implementation we execute a separate thread for the calculation of the value in every cell. However, this is far from optimal. The overhead for creating a separate thread for the calculation of each cell is massive. Furthermore, the hardware limitations of a graphics card limit the physical amount of concurrent executing threads, creating significantly more threads than this limit, which does not make the application any faster. Another consideration is that, when using barriers, all the threads in a work group wait for each other to enter the same barrier. When one would create more threads than the GPU physically can execute concurrently, it will slow the application down.

Therefore, we have optimized our main implementation to let every thread process multiple cells. The kernel will automatically calculate which cells it has to evaluate based on the total amount of threads, the total amount of cells (width*height) and its (global) thread id. This is illustrated in figure 6.1 and can be seen in the kernel on line 8. In this illustration we have a field of 2 by 5 cells, a work group size (or global_size) of 5 and we look at the execution of the thread with the identifier (global_id) 1; First the thread will calculate the cell at position 1, the next position is at 6 (or global_id+1*global_size), the next position (global_id+2*global_size) will be out of the range (width*height). And it is easy to see that the accumulated result of all the threads results in the calculation of the complete field.

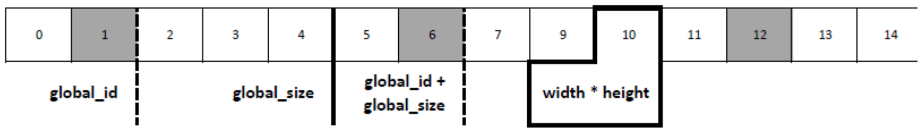


Figure 6.1: Loop for multiple cells per thread

This optimization is commonly used, and it has proven to be possible to automatically annotate such loops to include the needed loop invariants [12].

6.2 Verification

The optimization used in this kernel is common and as stated earlier it requires an additional loop invariant to hold true during execution. Bets *et al.* have deduced this invariant automatically for several common loops in their tool chain GPUverify[4]:

- Loops with a constant offset
- Loops with a constant offset and a strided offset (like our implementation)
- Loops where one thread accesses a continues range.
- Looping in powers of two

In our implementation, a loop with a constant offset and a strided offset is used, we access the following data:

> nextgen[global_id+(global_size*n)] where global_id+(global_size*n) < width*height

In general, the introduction of a loop needs an invariant. In our optimization case we use the loop to execute the same task multiple times for different memory locations and can be seen as the serialization of multiple parallel kernel executions. Therefore, when we have proven the original kernel to be correct we have to:

- Define an invariant for the loop and prove it to be correct.
- Adjust T_{res} to include the additional memory locations.
- Adjust T_{pre} and T_{post} to include the verification of the additional cells

The kernel specification $(K_{res}, K_{pre}, K_{post})$ would not need changes and the following property can be easily proven to be correct as long as the memory locations, for each cell, in each loop are disjoint.

$$K_{res} * K_{pre} -\!\!* \; \bigstar_{tid \in Tid} \, (T_{res|glob} * T_{pre})$$

The Loop Invariant

Since our loop has a set offset defined by the identifier of our kernel and a strided offset based on the work group size, we can define our loop invariant to be of this form:

```
( iterator − global_id ) % group_size == 0 AND
iterator < board_size+group_size
```

Proving our loop to be correct can then be done in the following fashion:

```
loop(statement,condition,update) + invariant:

{invariant&condition}body,update{invariant}
--------------------------------------------
{invariant}loop(body){invariant&!condition}
```

The $(T_{res}, T_{pre}, T_{post})$-Triple

To include the additional cells that the thread has to calculate in our thread-specification we can look at our loop invariant, since the loop invariant exactly describes which cells we access, we can adjust $(T_{res}, T_{pre}, T_{post})$ accordingly. For example, when a thread accesses an array in location *tid* to write some value it needs write permission for the location *array*[*tid*]. We can adjust the specification by replacing *tid* with something we can deduce from the iterator of the loop. For example, the write permissions for the cell in 01_SI was:

```
//@ requires perm(nextgen[gid],1)
```

With the introduced invariant, it becomes:

```
//@ requires \forall int i; i>=gid && i<width*height &&
(i-gid)%gid_size==0;perm(nextgen[i],1)
```

6.2.1 Verification in Relation to 01_SI

When we look at the changes in our kernel (verification) in relation to the Single Iteration kernel, we can determine a strategy to accomplish an informal verification with help of the, proven to be correct, Single Iteration kernel. What we have changed is the statements $(T_{res}, T_{pre}, T_{post})$ and the added invariant. First we look at our loop and its invariant. We have constructed an invariant by combining the previous T_{post} with the boundaries of our iterator. Therefore we can deduce that the code inside the loop is proven with respect to the T_{post}-part for locations i where $i \in (tid..\text{pos})\&(i\%gid\_size == 0)$ with *pos* being the iterator. The "iterator part" of our invariant determines the set for which this will be proven correct. Our iterator starts at pos $= tid$ and increments with *gid_size*. The loop condition is pos $<$ width $*$ height. Thus the set of *pos*(tid) will be:

$$\{\text{pos}|pos = tid + \text{gid\_size} * n,\\ n \in \{0..\mathbf{floor}((\text{width} * \text{height} - 1)/\text{gid\_size})\}\}$$

The new T_{post} is defined as:

```
// Tpost
  // We check all the cells that where reached (strided) by this thread.
  // If we are out of bounds, the cell is always dead. Otherwise the
       result should be the result of the rules of the Game of Life
  //@ ensures \forall int i; i>=gtid && i<width*height && (i-gtid)%
       gtid_size==0;oob(i) ==> nextgen[i]==0;
  //@ ensures \forall int i; i>=gtid && i<width*height && (i-gtid)%
       gtid_size==0;!oob(i) ==> nextgen[i]==gol(i)
```

Listing 6.2: Thread specification

Since the iterator follows exactly the same values (the set mentioned above) and we have argued that for all those values for *pos* (or *i*) the loop invariant holds, we can say that our invariant implies T_{post}.

To completly prove our optimization we have to prove that $K_{res}\&K_{pre} \mathrel{-\!\!*} \mathop{\LARGE *}_{tid\in Tid} (T_{res|glob}\&T_{pre})$ and $\mathop{\LARGE *}_{tid\in Tid} T_{post} \mathrel{-\!\!*} K_{post}$ still holds. This can be concluded, in an informal manner, from figure 6.1, where we can see that the optimized kernel still accesses each location once.

6.3 Conclusion

Introducing a loop to do the same operation multiple times at several locations in the memory is quite common practice. We have considered a case where the locations are:

1. Disjoint and thus, did not depend on each other
2. No barriers were used.
3. All functions dependent on *tid* were in the loop body.

To apply this optimization at a given kernel one has to define the following:

- Define **operation**(*tid*) as the operation inside the loop
- Define *pos* to be a set containing a range of values dependent on *tid*. With the combined set of *pos* for each new thread being the same as the set of all *tid* at the previous implementation.
- Split up $(T_{res}, T_{pre}, T_{post})$ (and all annotations) in $(T_{res}, T_{pre}, T_{post})$' and $(T_{res}, T_{pre}, T_{post})_{tid}$, with $(T_{res}, T_{pre}, T_{post})_{tid}$ being all the thread specifications that are not disjoint from *tid*, and $(T_{res}, T_{pre}, T_{post})$' with all thread specifications disjoint from *tid*.

Generally, the specified code will look like this:

$$\begin{aligned}
K_{pre} * K_{res} \\
T'_{pre} * (T_{pre})_{tid} \\
T'_{res} * (T_{res})_{tid} \\
. \\
\textbf{annotation\_pre}' * \textbf{annotation\_pre}_{tid} \\
\textbf{operation}(tid) \\
\textbf{annotation\_post}' * \textbf{annotation\_post}_{tid} \\
. \\
T'_{post} * (T_{post})_{tid} \\
K_{post}
\end{aligned}$$

We can now change $(T_{res}, T_{pre}, T_{post})_{tid}$ to $(T_{res}, T_{pre}, T_{post})_{pos}$, with $(T_{res}, T_{pre}, T_{post})_{pos}$ being the disjoint set with the original specifications but now for all values in *pos*.

Now we will have to create an invariant *inv*, based on *pos* and on $\textbf{annotation\_post}_{pos}$ and you can include that invariant in the loop around **operation**(*tid*). If this is done correctly, the following will hold:

$$\{inv * \text{loop\_condition} * \textbf{annotation\_pre}_{\text{pos}}\}\textbf{operation}(tid)\{inv\}$$

We know that $(T_{post})_{tid}$ holds for **operation**(*tid*), our invariant holds for all values of pos_{tid} and now we have changed the range of $(T_{post})_{tid}$ to $(T_{post})_{tid}$. Therefore our kernel is still correct.

Chapter 7

Barrier Optimization

```
__kernel void kernel(___global  unsigned int * B,
                     ___global  unsigned int * A,
                     const      unsigned int height,
                     const      unsigned int width,
                     const      unsigned int iterations)
  {
  int pos, up, down, outofbounds, neighbours;

  ___global unsigned int * board;
  ___global unsigned int * nextgen;

  for (int i=0;i<iterations;i++)
    {
    board = (i%2==0)?A:B;
    nextgen = (i%2==0)?B:A;


    for (pos = get_global_id(0);pos<width*height;pos += get_global_size(0))
      {

      outofbounds      = (pos < width);
      outofbounds     |= (pos > (width * (height-1)));
      outofbounds     |= (pos % width == 0);
      outofbounds     |= (pos % width == width-1);

      if (outofbounds)
        {
        nextgen[pos] = 0;
        }
      else
        {
        int neighbours = board[up-1]   +board[up]   +board[up+1];
        neighbours    += board[pos-1]               +board[pos+1];
        neighbours    += board[down-1] +board[down] +board[down+1];

        nextgen[pos] = (board[pos] && neighbours==2) || (neighbours == 3);
        }
      }
    barrier(CLK_GLOBAL_MEM_FENCE);
    }
  if (iterations%2==0)
    for (pos=get_global_id(0);pos<width*height;pos+=get_global_size(0))
      B[pos] = A[pos];
  }
```

Listing 7.1: Barrier Optimization[1]

[1]The fully annotated version can be found in appendix A.3

7.1 Implementation

In our current (optimized) implementation we still revoke the kernel from the host for each iteration. After an iteration *board* and *next_gen* get swapped by the host with the *clEnqueueCopyBuffer()*-command. Using the *clEnqueue-CopyBuffer()*-command is already a huge improvement, since it simply swaps the pointers instead of copying *board* and *next_gen* to the host, swap the values and sending it back. Still, the extra time needed to restart the kernel for each iteration will accumulate when we are doing millions of iterations. Therefore an additional speed-up can be achieved by implementing the iteration, and the swapping, at the kernel. We will use our implementation only on one work group. This in order to simplify the synchronization between multiple threads. Since the use of barriers will suffice when using only one work group. However, when using multiple work groups, synchronization on the host side or with atomics would be needed. To implement this we have three actions that we now have to handle on the kernel:

- Iteration.
- Swapping the input and output.
- Guaranteeing that each kernel is working on the same iteration, to prevent errors. With only one work group, a barrier will suffice.

7.2 Verification

As in the last implementation we have added a for-loop to calculate multiple cells per thread, this required us to do a similar operation as we have to do for our new iterator. However, our last implementation expected the operations to be disjoint, this iterator is swapping the buffers and thus the operations will write to the same location at several times in the loop. OpenCL offers a method to prevent data-races in such cases [19]: barriers. Barriers constitute two important properties. Foremost, when a thread reaches a barrier, the thread will halt until all the other threads in the work group also reach this point. Secondly, because of the property that all threads will wait for each other allows us to redistribute the permissions within the work group. Either on work group memory level, global level or both. Mihelcic and Huisman describe each barrier with a triple, similar to the kernel specification. A pre- and postcondition and a specification of the resources that will specify how the permissions will be redistributed within the work group. Mihelcic and Huisman proposed the following rules to be true for barrier verification [20]:

- The global memory from K_{res} is redistributed over B_{res} and B_{res} is not greater than K_{res}

  $K_{res} \mathbin{-\!\!*} \mathop{\Large *}_{tid \in Tid} B_{res|glob}$

- The same goes for the local memory within a work group.

  $\mathop{\Large *}_{v \in Local} \texttt{Perm}(v, 1) \mathbin{-\!\!*} \mathop{\Large *}_{ltid \in LTid} B_{res|loc}$

- Show that B_{post}, restricted by the locations that *tid* can read, is implied by B_{pre}

  $\mathop{\Large *}_{tid \in Tid} B_{pre} \mathbin{-\!\!*} \mathop{\Large *}_{tid \in Tid} B_{post}|_{RGPerm(tid)}$

For the verification of the barrier implementation, the following needs to be done:

1. Use the T_{post} of the previous implementation to construct the invariant for the outer loop, in a similar manner as the previous implementation.
2. Add $(B_{res}, B_{pre}, B_{post})$
3. Define new $(T_{res}, T_{pre}, T_{post})$ and $(K_{res}, K_{pre}, K_{post})$ in a similar manner as the previous implementation.

7.2.1 Invariant

For our invariant we use a part associated with the iterator:

$$i >= 0 \ \& \ i <= iterations$$

Secondly, we want to prove that all the previously calculated generations are correct. In our last implementation this was done by checking it for all the values of our iterator. This is not possible since our operations are not disjoint. For example, we use the same memory for all our iterations, reading from location A, writing to B and vice versa. For this we have introduced ghost-variables. A ghost variable simply refers to the state of a variable at a given time, or in our case the state of the board at a given iteration. For example,
$\mathbf{ghost}\{A, i\}$, gives the value of the variable A representing the board at iteration i, while
$\mathbf{ghost}\{A, 0\}$ describes the state of the board at initialization. Now we have designed a model method that can calculate the state of a cell at a given generation. Therefore we have to keep a ghost variable for both A and B, and prove by induction that each calculated board is correct. We define the method: $\mathrm{gol}^i(n, A)$ with i being the iteration, A the initial state of the board at iteration i, and n the location of the cell. The result of the method is the calculated result for cell n, based on the given parameters, respecting the rules of the Game of Life.

The state of a cell at a given time can thus be calculated with:

$$\mathbf{gol}^i(n, A) = \begin{cases} i = 0 & \begin{cases} \mathrm{outofbounds}(A, n) & dead \\ else & \begin{cases} (\mathrm{neighbours}(A, n) == 2\&\& \\ A[n] == alive) || \\ (\mathrm{neighbours}(A, n) == 3) & alive \\ else & dead \end{cases} \end{cases} \\ i \neq 0 & \mathbf{gol}^{(i-1)}(n, \backslash\mathbf{ghost}\{A, i-1\}) \end{cases}$$

- With *neighbours*(A, n) denoting the amount of live neighbours of cell n given board A. And the function *outofbounds*(A, n) returning true when cell n is positioned on the boarder of board A
- We can use $\mathrm{nextgen[pos]} == \mathbf{gol}^0(\mathrm{pos}, \mathrm{board})$ as assurance at line 37 of the kernel shown at the beginning of this Chapter.

- The invariant at line 18 contains the following formula:

$$\begin{aligned}\backslash\textbf{forall}(\textbf{int}\ i;\\ i >= \textbf{gid}\ \&\ i < \text{pos}\&(i-\text{gid})\%\text{gid\_size} == 0;\\ \text{nextgen}[\text{pos}] == \textbf{gol}^{i}(\text{pos}, \text{board}))\end{aligned}$$

  The invariant of the outer loop (line 12) contains:

$$\begin{aligned}\backslash\textbf{forall}(\textbf{int}\ i;\\ i >= \textbf{gid}\&i < \text{pos}\&(i-\text{gid})\%\text{gid\_size} == 0;\\ \backslash\textbf{forall}(\textbf{int}\ j;\\ j >= 0\&j < width * height;\\ \text{nextgen}[j] == \textbf{gol}^{i}(j, \text{board}))\end{aligned}$$

Note that for the outer loop invariant we can check the correct outcome for the complete board at the thread level, this is because of the nature of the barrier. This shows that the creation of an invariant for a loop in a rather simple looking kernel can be quite a challenge.

7.2.2 Barrier

For the barrier we defined, we state:

- When a thread reaches the barrier, the next generation for the cells associated with the *tid* of that thread have been correctly calculated. This is B_{pre};
- When the rights are redistributed the two arrays are swapped. The next generation will become the current generation and vice versa. This is described in B_{res}
- B_{post} describes that after the barrier it is guaranteed that each cell is one generation "older".

This is shown in the code as follows:

```
//Bpre
//@ assert \forall int i; i>=gtid && i<width*height && (i-gtid)%
     gtid_size==0;oob(i) ==> nextgen[i]==0;
//@ assert \forall int i; i>=gtid && i<width*height && (i-gtid)%
     gtid_size==0;!oob(i) ==> nextgen[i]==gol(i)

barrier(CLK_GLOBAL_MEM_FENCE);

//Bres
//flip rights
//@ ensures \forall int i; i>=gtid && i<width*height && (i-gtid)%
     gtid_size==0;!oob(i) ==>
//@   perm(((i%2==0)?B:A)[i-width-1],p) ** perm(((i%2==0)?B:A)[i-width
     ],p) ** perm(((i%2==0)?B:A)[i-width+1],p) **
//@   perm(((i%2==0)?B:A)[i-1],p) ** perm(((i%2==0)?B:A)[i+1],p) **
     perm(((i%2==0)?B:A)[i+width-1],p) **
//@   perm(((i%2==0)?B:A)[i+width],p) ** perm(((i%2==0)?B:A)[i+width
     +1],p) ** perm(((i%2==0)?A:B)[i],1)

//Bpost
// make sure we calculated the right result
```

```
//@ ensures \forall int n; n>=0 && i<width*height;oob(n) ==> nextgen[n
    ]==0;
//@ ensures \forall int n; n>=0 && i<width*height;!oob(n) ==> nextgen[
    n]==gol(n,i)
```

Listing 7.2: Barrier and the barrier-specification

From our previous implementation, and/or B_{pre}, we know that when a thread reaches the barrier all the cells for which the thread is responsible are correctly calculated. When all the threads have passed the barrier, the relation between B_{pre} and B_{post} is quite similar to parts of T_{post} and K_{post} in our previous implementation. Thus, we can argue that after the barrier, we can guarantee that the complete board has evolved one generation. The third element in our barrier specification (B_{res}) guarantees that we have the correct right distribution for the next iteration. Moreover, $K_{res} \mathrel{-\!\!*} \mathop{\ast}_{tid \in Tid} B_{res|glob}$ still holds.

After the execution of the two nested loops, the correct field is copied to the array that the host expects the result to be (array **B**). The postcondition of the kernel states that that field is the field that was given as input, after *iteration*-generations of the Game of Life. For this we use the following statement in the code, seen in the appendix:

$$B[\text{pos}] == \mathbf{gol}^{i}(\text{pos}, \text{board})$$

We say that this statement is correct, if the invariant of the outer loop is correct (and when the copy to field B is done correct).

7.3 Conclusion

The use of a barrier has not led to a general "recipe" for the appliance of barriers, but it gives us an idea about what to do with loops containing a barrier. Let's consider a kernel, with an input and an output buffer, and a correct specification for that kernel. This kernel is executed multiple times, using one work group, and each time with the result of the previous execution as input. Then we can introduce a new kernel, with a loop containing the code from the original kernel combined with a barrier. When we make sure that this barrier guarantees the following:

- B_{pre} contains T_{post} of the old kernel.
- B_{post} contains K_{post} from the old kernel.
- B_{res} guarantees that the correct right distribution for that specific iteration.

For each iteration we make sure that we use the output from the previous iteration as input, for example, by using the first buffer for the odd iteration and the second one for the even iterations. Then by introducing an ghost-variable for both the input and output, and the old T_{post}, we can make an invariant for the loop. For K_{post} one could use the same function, based on T_{post} and the ghost-variables.

The nature of the barrier allows us to eliminate T_{post} or use K_{post} for T_{post}.

Chapter 8

Localization Optimization

```
__kernel void kernel(__global  unsigned int * nextgen,
                     __global  unsigned int * board,
                     const     unsigned int height,
                     const     unsigned int width,
                     const     unsigned int iterations)
  {
  int pos, up, down, outofbounds, neighbours;
  __local int cached[2][16*128];
  for (pos=get_global_id(0);pos<width*height;pos+=get_global_size(0))
    {
    cached[0][pos] = board[pos];
    cached[1][pos] = nextgen[pos];
    }
  barrier(CLK_LOCAL_MEM_FENCE);
  for (int i=0;i<iterations;i++)
    {
    for (pos=get_global_id(0);pos<width*height;pos+=get_global_size(0))
      {
      outofbounds      = (pos < width);
      outofbounds     |= (pos > (width * (height-1)));
      outofbounds     |= (pos % width == 0);
      outofbounds     |= (pos % width == width-1);

      if (outofbounds)
        {
        cached[1-(i%2)][pos] = 0;
        }
      else
        {
        neighbours  = cached[i%2][up-1]   +cached[i%2][up]   +cached[i%2][up+1];
        neighbours += cached[i%2][pos-1]                     +cached[i%2][pos+1];
        neighbours += cached[i%2][down-1] +cached[i%2][down] +cached[i%2][down+1];
        cached[1-(i%2)][pos] = (cached[i%2][pos] && neighbours==2) || (neighbours==3);
        }
      }
    barrier(CLK_LOCAL_MEM_FENCE);
    }
  for (pos = get_global_id(0);pos<width*height;pos += get_global_size(0))
    nextgen[pos] = cached[iterations%2][pos];
  }
```

Listing 8.1: Localization Optimization[1]

[1]The fully annotated version can be found in appendix A.5

8.1 Implementation

Our last optimization is to use the local memory. Using the local memory in OpenCL usually corresponds to using a physically different kind of memory (also see fig. 3.1). The local memory is usually significantly faster (and more expensive, thus smaller) [10]. Local memory cannot be accessed from the host, so using local memory requires copying at the beginning and the end of the kernel. One can imagine that those write- and read operations require additional time, and that using local memory only pays off when a kernel does sufficient operations on that memory. In our case we use two fields representing the current state of the board and the state of the board for the next generation. With millions of iterations to calculate the millionth generation of the Game of Life, the fields are approximately read $\frac{9*iterations}{2}$, and $\frac{iterations}{2}$ times written to. Therefore, localizing the fields to the work group-level would allow for a significant speed up. Note that we still use only one work group, therefore it is possible to use the work group memory. Using multiple work groups could be possible, but would require us to synchronize the work groups; an operation that is not implemented in OpenCL with a simple mechanic like barriers. Each work group would calculate a part of the field, and upon synchronization the work groups would share the cells on the border of these fields. Another change in relation to our previous implementation is that, since we are swapping local memory instead of global memory, we have to use a barrier with a local memory fence, instead of a global one.

8.2 Verification

We have added two operations to our code:

- Copy the input to the local memory
- Copy the output from local memory to the global memory.

We have replaced all the occurences of the input and output buffer with the equivalent in the local memory. The changes can be seen as follows:

$$\begin{aligned} \{T_{pre}\} \\ copy\_to\_localmem \\ \{A\} \\ previous\_kernel\_implementation \\ \{T_{post}[local \backslash global]\} \\ copy\_to\_globalmem \\ \{T_{post}\} \end{aligned}$$

In this {A} is the assertion that describes that all the data is copied correctly to the local memory. Now we only have to prove that $\{T_{pre}\}copy\_to\_localmem\{A\}$ holds - so that the data is copied correctly, which we can re-use to show that we have copied the data correctly back. All the other proofs can be re-used from the previous proof, with the exception that all the accesses to global memory are substituted by the local cache, and that the barrier uses a local memory fence.

8.3 Conclusion

A common approach when optimizing a GPGPU-program is using the fast local memory. When we look at this optimization, in relation to the previous optimizations, we can see the following: provided that the copying to the local memory happens at the beginning and the end of the kernel, the localization optimization possibly introduces the least amount of additional proving.

Chapter 9

Results and Conclusion

We have looked at several implementations and common optimizations of an OpenCL kernel depicting Conway's Game of Life. We ran each of the implementations on a Tesla M2050 graphical card and timed the execution time. The amount of threads in the initial implementation was equal to the size of the board (width*height=$2^7 * 2^4$) and we calculated the resulting board after 2^{14} iterations. For our optimizations we used one workgroup, containing 1024 threads. Using a less excessive amount of threads, speeded our application up. Additionally, we chose for only one work group because that made synchronization between threads more feasible. Synchronization between multiple work groups, would possible force us to synchronize on the host side, and thus force us to verify the host code (which is out of the scope of this project). Table 9.1 and figure 9.1 shows that we have made 5 implementations. Only four of them are mentioned in the report, this is because **Localization1** did not show the expected speed up, which we solved in the version described in this thesis. The list of implementations is as follows:

- Main: This implementation is a simple implementation, which is optimized in the next implementations. This implementation calculates one iteration of the Game of Life, and for each cell on the board we need to start a thread.

- Threadcount: Here we started a less excessive amount of threads, with each thread being responsible for the calculation of the next generation of multiple cells.

- Barrier: We have introduced a barrier, allowing us to iterate through multiple generations of the Game of Life on the kernel side, instead of invoking the kernel from the host for each iteration.

- Localization: We have copied our buffers to the shared work group memory. This allowed us to take advantage of the faster memory available for the work groups on the GPU.

9.1 Verification

We verified our implementation using permission-based separation logic, with additional rules exclusively designed for the use with the OpenCL. We annotated the code with JML in a similar style as used with the VerCors project. The actual verification was done manually, based on the style described by Gordon. After the verification of the initial implementation we have looked at the changes needed in the verification when one would introduce certain optimizations.

9.1.1 Loops

In our first optimization we saw that with the help of a loop we could let a single thread do the work that in our previous version was done by multiple threads. Therefore, we had to introduce a loop structure. In our verification we needed to introduce a loop invariant, which we could base on the thread post condition from the previous version, known as T_{post}. And we needed to adjust $(T_{res}, T_{pre}, T_{post})$ to account for the fact that a single thread would now be responsible for the work that was previously done by multiple threads. The problem with this optimization is that we can not guarantee correctness when the tasks done by the new thread are not originally disjoint.

9.1.2 Barriers

With our barrier implementation we introduced iterations of the Game of Life on the kernel side. For each iteration we need the result of the previous iteration to calculate the new field. Since multiple threads are responsible for the calculation of the complete field, it was necessary to ensure that all the threads are always working on the same iteration. Therefore we introduced a barrier, along with the rules needed to verify the barrier. To verify our code we also had to introduce the use of ghost-variables, since the values stored in the board are overwritten every other iteration. For the barrier specification $(B_{res}, B_{pre}, B_{post})$, we have adapted $(T_{res}, T_{pre}, T_{post})$ to use as $(B_{res}, B_{pre}, B_{post})$. Furthermore, we have used T_{post} and K_{post} from the previous version, as a base for respectively B_{pre} and B_{post}.

9.1.3 Localization

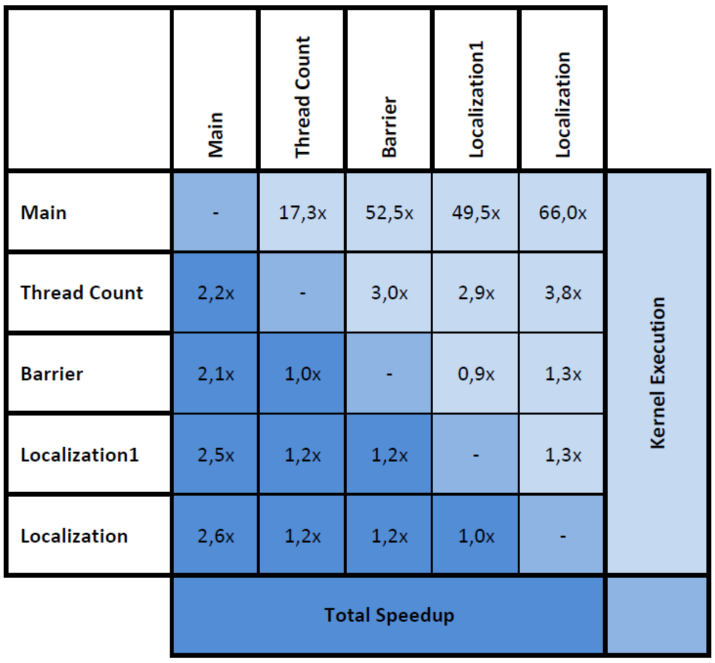
For localization we replace all the occurrences of a global buffer with its local equivalence. In the beginning and the end of the code we copy these values respectively from and to the global memory, protected by a barrier, to make sure that all the threads have their data copied before we start the "refurbished" middle of the code, being the old kernel. The verification of this optimization is rather easy, as long as we can assure that all the threads can access this local equivalence. When using multiple work groups, this can not always be guaranteed, and additional schemes would have to be devised to allow synchronization of the work groups and synchronization of the data used by the multiple work groups.

9.2 Future work

In this thesis we have seen a manual verification of a simple implementation of Conways Game of Life. We have seen both in this thesis as in the ongoing work of Huisman, Blom and Mihelcic that formal verification of OpenCL kernels, is both a possibility and a necessity. This work, even though it is limited to manual verification of OpenCL code and only using one work group shows that there might be valuable abstractions to automatically verify optimized OpenCL code, whilst still guaranteeing the correctness of that specific code. Therefore, future work should include the automatic verification of OpenCL-code on both the kernel and host side to provide verification for multiple, synchronizing, work groups. With a rigid theoretical platform and a tool set, perhaps similar to the VerCors tool set, the patterns discussed in this thesis could be implemented in a formally correct manner, and automatic optimization and verification suggestions could be presented to the user of the tool set.

Table 9.1: Table shows the execution time of the Kernels in ms

Kernel	Initializing Time	Compiling Time	Running Time	Completion Time	Cum.
Main	1	4029	4357	293	8680
ThreadCount	1	3628	252	151	4032
Barrier	0	3913	83	85	4081
Localization1	1	3228	88	121	3438
Localization	1	3248	66	75	3390



	Main	Thread Count	Barrier	Localization1	Localization
Main	-	17,3x	52,5x	49,5x	66,0x
Thread Count	2,2x	-	3,0x	2,9x	3,8x
Barrier	2,1x	1,0x	-	0,9x	1,3x
Localization1	2,5x	1,2x	1,2x	-	1,3x
Localization	2,6x	1,2x	1,2x	1,0x	-

Kernel Execution
Total Speedup

Figure 9.1: Speed ups for kernel execution time and total execution time

Chapter 10

Related work

10.1 GPUVerify

GPUVerify is a tool chain developed by Betts *et al.* under the CARP project. GPUVerify is specifically designed for verification of GPGPU programs and can handle input in the form of OpenCL or CUDA code [4]. For this purpose the authors devised Synchronous Delayed Visibility semantics, or SDV. Under the hood, the tool with the help of SDV, translates multiple threads to a sequential program. GPUVerify allows for thread interleaving, barriers and keeps track of shared variables. The tool then generates Boogie code that can be verified with conventional verification tools [9].

With this method GPUVerify is a tool that is fast - and produces a minimal amount of false negatives - that is still being improved and has been proven to work on real life examples [8][12].

10.2 Other Verification Work for GPGPU

Current verification methods are based on a range of techniques. Collingbourne requires an OpenCL kernel with its equivalent in normal C-code [7]. By means of symbolic execution of both versions it is able to detect errors when the symbolic value does not match after execution. Additionally, data races are detected by keeping track of memory access.

Another approach is used by Li and Gopalakrishnan [27]. Li suggested the tool PUG$_{\text{PARA}}$. The main motivation for creation of this tool is to see divergences between optimized and unoptimized versions of an CUDA kernel. The tool analyses only one parametrized thread of a kernel. By keeping track of the (parametrized) operations done by this thread it can infer, with help of a SMT-solver, what the effects of multiple threads executing identical (parametrized) operations are, as well as discover possible errors.

Bibliography

[1] P. Alvarez and S. Yamagiwa. Invitation to OpenCL. In *Networking and Computing (ICNC), 2011 Second International Conference on*, pages 8 –16, 30 2011-dec. 2 2011.

[2] AMD. Introduction to OpenCL Programming.

[3] A. Amighi, S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. The VerCors project: setting up basecamp. In *Proceedings of the sixth workshop on Programming languages meets program verification*, pages 71–82. ACM, 2012.

[4] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: a verifier for GPU kernels. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 113–132, New York, NY, USA, 2012. ACM.

[5] R. Bosch. Maximum density stable patterns in variants of Conway's game of Life. *Operations Research Letters*, 27(1):7–11, 2000.

[6] D. Chisnall. Introducing OpenCL, July 2010.

[7] P. Collingbourne, C. Cadar, and P. Kelly. Symbolic testing of OpenCL code. In *Haifa Verification Conference (HVC)*, 2011.

[8] P. Collingbourne, A. F. Donaldson, J. Ketema, and S. Qadeer. Interleaving and lock-step semantics for analysis and verification of GPU kernels. In *Programming Languages and Systems*, pages 270–289. Springer, 2013.

[9] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[10] J. Fang, A. Varbanescu, and H. Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216 –225, sept. 2011.

[11] C. Finney. On counting in the game of life. *Computers & Education*, 10(2):315 – 325, 1986.

[12] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity*, pages 500–517. Springer, 2001.

[13] B. Gaster, L. Howes, D. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous computing with OpenCL*. Morgan Kaufmann, 2011.

[14] M. Gordon. *Specification and Verification I*. Computer Science Tripos, Part II, 2009.

[15] R. W. Gosper. Exploiting regularities in large cellular spaces. *Physica D: Nonlinear Phenomena*, 10(1):75–80, 1984.

[16] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's reentrant locks. *Programming Languages and Systems*, 5356:171–187, 2008.

[17] C. Haack, M. Huisman, and C. Hurlin. Permission-based separation logic for multithreaded java programs. *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica*, 15:13–23, 2011.

[18] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[19] A. Hobor and C. Gherghina. Barriers in Concurrent Separation Logic: Now With Tool Support! *Logical Methods in Computer Science*, 8(2):1–36, 2012.

[20] M. Huisman and M. Mihelcic. Specification and Verification of GPGPU Programs using Permission-Based Separation Logic. 2013.

[21] C. Hurlin. Specification and Verification of Multithreaded Object-Oriented Programs with Seperation Logic. Master's thesis, University of Nice-Sophia Antipolis, 2009.

[22] P. Jaaskelainen, C. de La Lama, P. Huerta, and J. Takala. OpenCL-based design methodology for application-specific processors. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 223–230. IEEE, 2010.

[23] K. Karimi, N. G. Dickson, and F. Hamze. A Performance Comparison of CUDA and OpenCL. *CoRR*, abs/1005.2581:1–10, 2010.

[24] Khronos. *The OpenCL Specification*. Khronos OpenCL Working Group, v 1.2 edition, 11 2011.

[25] C. E. LaForest. ECE1724 Project Final Report: HashLife on GPU. 2010.

[26] G. Leavens and Y. Cheon. Design by Contract with JML. *Draft, available from jmlspecs. org*, 2006.

[27] G. Li and G. Gopalakrishnan. Parameterized verification of GPU kernel programs. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2450–2459. IEEE, 2012.

[28] K. S. Perumalla and B. G. Aaby. Data parallel execution challenges and runtime performance of agent simulations on gpus. In *Proceedings of the 2008 Spring simulation multiconference*, pages 116–123. Society for Computer Simulation International, 2008.

[29] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[30] T. Rumpf. Conway's Game of Life accelerated with OpenCL. In *Proceedings of the Eleventh International Conference on Membrane Computing*, 2010.

[31] R. Skudnov. Bitcoin clients. *Instructor*, 3(12):32, 2012.

[32] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 325–335. ACM, 2006.

[33] S. Wolfram. Theory and applications of cellular automata. 1986.

List of Figures

Appendices

Appendix A

Listing of Kernels

Listing A.1: Main Implementation Kernel

```
// MODEL VARIABLES
// gtid is used as shorthand for the unique (global) thread identifier.
//@ private model int gtid;
//@ private represents gtid <- get_global_id(0);
// gtid_size returns the total amount of threads.
//@ private model int gtid_size;
//@ private represents gtid_size == get_global_size(0);
// oob stands for Out Of Bounds, and tells us whether cell i, is on the
    border of the field.
//@ private model function oob(i);
//@ private represents oob(i) <- (i < width)||(i > (width * (height-1)))
    ||(i % width == 0)||(i % width == width-1);
// nb, gives the amount of direct, live neighbours of cell i.
//@ private model function nb(i);
//@ private represents nb(i) <- board[i-width-1]+board[i-width]+board[i-
    width+1]+board[i-1]+board[i+1]+board[i+width-1]+board[i+width]+
    board[i+width+1];
// gol returns the expected state of cell i, given the current board,
    respecting the rules of the game of life
//@ private model function gol(i);
//@ private represents gol(i) <- (board[i] && nb(i) == 2) || (nb(i) ==
    3)

// THREAD SPECIFICATION
// Tres - resources needed for the thread
  //@ requires perm(width,p) ** perm(height,p);
  //@ requires !oob(gtid) ==> perm(board[gtid-width-1],p) ** perm(board[
      gtid-width],p) **
  //@   perm(board[gtid-width+1],p) ** perm(board[gtid-1],p) ** perm(
      board[gtid+1],p) **
  //@   perm(board[gtid+width-1],p) ** perm(board[gtid+width],p) ** perm
      (board[gtid+width+1],p)
  //@ requires perm(nextgen[gtid],1)

// Tpre - preconditions for the thread
  // The amount of threads must equal the size of the boards
  //@ requires (width*height) == gtid_size;
  //@ requires sizeof(board)==sizeof(nextgen) && sizeof(board)/sizeof(
      int)==(width*height);

// KERNEL SPECIFICATION
// Kres - resources needed for the kernel
  // require write access to each location on the output-board, and read
       access to each location of the input-board
  //@ requires \forall int i; i>=0 && i<(width*height); perm(board[i],p)
  //@ requires \forall int i; i>=0 && i<(width*height); perm(nextgen[i
      ],1)

  //@ requires perm(width,p) ** perm(height,p);
```

```
// Kpre - precondtions for the kernel
  // The amount of threads must equal the size of the boards
  //@ requires (width*height) == gtid_size;
  //@ requires sizeof(board)==sizeof(nextgen) && sizeof(board)/sizeof(
      int)==(width*height);


__kernel void Kernel( __global  unsigned int * nextgen,
                      __global  unsigned int * board,
                      const     unsigned int height,
                      const     unsigned int width)
{

  int pos, up, down, outofbounds, neighbours;

  pos    = get_global_id(0);

  // pos -/+ width gives same position but one row higher/lower
  up     = pos - width;
  down   = pos + width;

  //out of bounds is when pos is on an edge
  outofbounds     = (pos < width);          //upper edge
  outofbounds    |= (pos > (width * (height-1)));//lower edge
  outofbounds    |= (pos % width == 0);        //left edge
  outofbounds    |= (pos % width == width-1);  //right edge

// Outofbounds should be correct here
//@ assert outofbounds == oob(gtid)
//@ assert up = pos - width;
//@ assert down = pos + width;
//@ assert pos = gtid;

  //all live cells on the edges die. Period. Therefore, a thread does
      not need read permissions for a cell that is out of bounds
  if (outofbounds)
  {
    nextgen[pos] = 0;
  }
  else
  {
    //sum up all the neighbours
    neighbours      = board[up-1]     +board[up]     +board[up+1];
    neighbours     += board[pos-1]             +board[pos+1];
    neighbours     += board[down-1]   +board[down]   +board[down+1];

    /** \brief We play "B3/S23 Life"
    So a cell comes alive when it has exactly 2 neighbours.
    Stays in the same state when it has 2 or 3 neighbours.
    And dies in all other cases
    **/
    nextgen[pos] = (board[pos] && neighbours == 2) || (neighbours == 3);
  }

}

//THREAD SPECIFICATION
// Tpost
  // if we are out of bounds, the cell is always dead. Otherwise the
      result should be the result of the rules of the Game of Life
  //@ ensures oob(gtid) ==> nextgen[gtid]==0;
  //@ ensures !oob(gtid) ==> nextgen[gtid]==gol(gtid)

//KERNEL SPECIFICATION
// Kpost
  // Make sure that the correct value is calculated for all the cells.
  //@ ensures \forall int i; i>=0 && i<width*height; oob(gtid)?nextgen[i
      ]==0:nextgen[i]==gol(i)
```

Listing A.2: Thread Count Optimzation Kernel

```
// MODEL VARIABLES
// gtid is used as shorthand for the unique (global) thread identifier.
//@ private represents gtid <- get_global_id(0);
// gtid_size returns the total amount of threads.
//@ private represents gtid_size == get_global_size(0);
// oob stands for Out Of Bounds, and tells us whether cell i, is on the
     border of the field.
//@ private represents oob(i) <- (i < width)||(i > (width * (height-1)))
     ||(i % width == 0)||(i % width == width-1);
// nb, gives the amount of direct, live neighbours of cell i.
//@ private represents nb(i) <- board[i-width-1]+board[i-width]+board[i-
     width+1]+board[i-1]+board[i+1]+board[i+width-1]+board[i+width]+
     board[i+width+1];
// gol returns the expected state of cell i, given the current board,
     respecting the rules of the game of life
//@ private represents gol(i) <- (board[i] && nb(i) == 2) || (nb(i) ==
     3)

// THREAD SPECIFICATION
// Tres - resources needed for the thread
  //@ requires perm(width,p) && perm(height,p);
  //@ requires \forall int i; i>=gtid && i<width*height && (i-gtid)%
       gtid_size==0;!oob(i)
  //@   ==> perm(board[i-width-1],p) ** perm(board[i-width],p) ** perm(
       board[i-width+1],p) **
  //@    perm(board[i-1],p) ** perm(board[i+1],p) ** perm(board[i+width
       -1],p) **
  //@    perm(board[i+width],p) ** perm(board[i+width+1],p) ** perm(
       nextgen[i],1)

// Tpre - preconditions for the thread
  // The given size must equal the size of the boards
  //@ requires sizeof(board)==sizeof(nextgen) && sizeof(board)/sizeof(
       int)==(width*height);

// KERNEL SPECIFICATION
// Kres - resources needed for the kernel
  // require write access to each location on the output-board, and read
        access to each location of the input-board
  //@ requires \forall int i; i>=0 && i<width*height; perm(board[i],p)
  //@ requires \forall int i; i>=0 && i<width*height; perm(nextgen[i],1)

  //@ requires perm(width,1) ** perm(height,1);

// Kpre - precondtions for the kernel
  // The given size must equal the size of the boards
  //@ requires sizeof(board)==sizeof(nextgen) && sizeof(board)/sizeof(
       int)==(width*height);


__kernel void Kernel( ___global  unsigned int * nextgen,
                       ___global  unsigned int * board,
                      const      unsigned int height,
                      const      unsigned int width)
{

int pos, up, down, outofbounds, neighbours;

// LOOP - calculates all the cells strided (amount: gtid_size) and with
     offset gtid

// the loop invariant consists of two parts:
// 1) Keeping the loop within bounds
//   pos>=gtid && pos<width*height+gtid_size && (pos-gtid)%gtid_size==0
// 2) Asserting that all the calculated results are correct
//   \forall int i;i>=gtid && i<pos && (i-gtid)%gtid_size==0;oob(i)?
     nextgen[i]==0:nextgen[i]==gol(i)

//@ loop_invariant pos>=gtid && pos<width*height+gtid_size && (pos-gtid)
     %gtid_size==0 && \forall int i;i>=gtid && i<pos && (i-gtid)%
     gtid_size==0;oob(i)?nextgen[i]==0:nextgen[i]==gol(i)
for (pos = get_global_id(0);pos<width*height;pos += get_global_size(0))
  {
    // pos -/+ width gives same position but one row higher/lower
    up     = pos - width;
    down   = pos + width;

    //out of bounds is when pos is on an edge
    outofbounds   = (pos < width);          //upper edge
    outofbounds   |= (pos > (width * (height-1)));//lower edge
    outofbounds   |= (pos % width == 0);       //left edge
    outofbounds   |= (pos % width == width-1);  //right edge

  // Outofbounds should be correct here
  //@ assert outofbounds == oob(gtid)
  //@ assert up = pos - width;
  //@ assert down = pos + width;

    //all live cells on the edges die. Period. Therefore, a thread does
        not need read permissions for a cell that is out of bounds
    if (outofbounds)
    {
      nextgen[pos] = 0;
    }
    else
    {
      //sum up all the neighbours
      neighbours   = board[up-1]    +board[up]    +board[up+1];
      neighbours   += board[pos-1]             +board[pos+1];
      neighbours   += board[down-1]  +board[down]  +board[down+1];

      /** \brief We play "B3/S23 Life"
      So a cell comes alive when it has exactly 2 neighbours.
      Stays in the same state when it has 2 or 3 neighbours.
      And dies in all other cases
      **/
      nextgen[pos] = (board[pos] && neighbours == 2) || (neighbours ==
          3);
    }
  // make sure that the calculated result for this cell is correct
  //@ assert oob(pos)?nextgen[pos]==0:nextgen[pos]==gol(pos)
  }
}


//THREAD SPECIFICATION
// Tpost
  // We check all the cells that where reached (strided) by this thread.
  // If we are out of bounds, the cell is always dead. Otherwise the
      result should be the result of the rules of the Game of Life
  //@ ensures \forall int i; i>=gtid && i<width*height && (i-gtid)%
      gtid_size==0;oob(i) ==> nextgen[i]==0;
  //@ ensures \forall int i; i>=gtid && i<width*height && (i-gtid)%
      gtid_size==0;!oob(i) ==> nextgen[i]==gol(i)

//KERNEL SPECIFICATION
// Kpost
  // Make sure that the correct value is calculated for all the cells.
  //@ ensures \forall int i; i>=0 && i<width*height; oob(gtid)?nextgen[i
      ]==0:nextgen[i]==gol(i)
```

Listing A.3: Barrier Optimization Kernel

```
// MODEL VARIABLES
// gtid is used as shorthand for the unique (global) thread identifier.
//@ private represents gtid <- get_global_id(0);
// gtid_size returns the total amount of threads.
//@ private represents gtid_size == get_global_size(0);

// oob stands for Out Of Bounds, and tells us whether cell n, is on the
     border of the field.
//@ private represents oob(n) <- (n < width)||(n > (width * (height-1)))
     ||(n % width == 0)||(n % width == width-1);

// nb, gives the amount of direct, live neighbours of cell n at
     iteration i.
//@ private represents nb(i,n) <- ghostboard(i)[n-width-1]+ghostboard(i)
     [n-width]+ghostboard(i)[n-width+1]+ghostboard(i)[n-1]+ghostboard(i)
     [n+1]+ghostboard(i)[n+width-1]+ghostboard(i)[n+width]+ghostboard(i)
     [n+width+1];

// gol returns the expected state of cell n, given the board A and
     iteration i, respecting the rules of the game of life
//@ private represents gol(i,n) <- (ghostboard(i)[n] && nb(i) == 2) || (
     nb(i) == 3) :

// ghostboard gives the array representing the board at a given
     iteration
//@ private represents ghostboard(i) <- (i%2==0)?ghost(A,i):ghost(B,i);
// ghostnextgen gives  array representing nextgen at a given iteration
//@ private represents ghostnextgen(i) <- (i%2==1)?ghost(A,i):ghost(B,i)
     ;


// GHOST VARIABLES
// ghost(A,i), ghost(B,i) respectivly give the state of A and B at
     iteration i. With ghost(A,0)==A && ghost(B,0)==B


// THREAD SPECIFICATION
// Tres - resources needed for the thread
  //@ requires perm(width,p) && perm(height,p);
  //@ requires \forall int i; i>=gtid && i<width*height && (i-gtid)%
       gtid_size==0;!oob(i)
  //@   ==> perm(board[i-width-1],p) ** perm(board[i-width],p) ** perm(
       board[i-width+1],p) **
  //@    perm(board[i-1],p) ** perm(board[i+1],p) ** perm(board[i+width
       -1],p) **
  //@    perm(board[i+width],p) ** perm(board[i+width+1],p) ** perm(
       nextgen[i],1)

// Tpre - preconditions for the thread
  // The given size must equal the size of the boards
  //@ requires sizeof(board)==sizeof(nextgen) && sizeof(board)/sizeof(
       int)==(width*height);

// KERNEL SPECIFICATION
// Kres - resources needed for the kernel
  // require write access to each location on the output-board, and the
       input-board
  //@ requires \forall int i; i>=0 && i<width*height; perm(board[i],1)
       ** perm(nextgen[i],1)

  //@ requires perm(width,1) ** perm(height,1) ** perm(iterations,1);

// Kpre - precondtions for the kernel
  // The given size must equal the size of the boards
  //@ requires sizeof(board)==sizeof(nextgen) && sizeof(board)/sizeof(
       int)==(width*height);


__kernel void Kernel(          ___global  unsigned int * B, //nextgen
                   ___global  unsigned int * A, //board
                   const      unsigned int height,
                   const      unsigned int width,
                   const      unsigned int iterations)
```

```
{

int pos, up, down, outofbounds, neighbours;

__global unsigned int * board;
__global unsigned int * nextgen;

//@ loop_invariant i>=0 && i<=iterations &&
//@   \forall int j;j>=0 && j<i;
//@     \forall k>=gtid && k<=width*height+group_size && (k-gtid)%
    gtid_size==0 &&
//@       ghostboard(j)[k] == gol(j,k);
for (int i=0;i<iterations;i++)
  {
  board = (i%2==0)?A:B;
  nextgen = (i%2==0)?B:A;
  //@ assert  board = (i%2==0)?A:B;
  //@ assert nextgen  = (i%2==0)?B:A;


  //@ loop_invariant pos>=gtid && pos<width*height+gtid_size && (pos-
      gtid)%gtid_size==0 &&
  //@   \forall int j;j>=gtid && j<pos && (j-gtid)%gtid_size==0;oob(j)?
      nextgen[j]==0:nextgen[j]==gol(i,j)
  for (pos = get_global_id(0);pos<width*height;pos += get_global_size(0)
      )
    {

    // pos -/+ width gives same position but one row higher/lower
    up    = pos - width;
    down  = pos + width;

    //out of bounds is when pos is on an edge
    outofbounds = (pos < width);          //upper edge
    outofbounds   |= (pos > (width * (height-1)));//lower edge
    outofbounds   |= (pos % width == 0);        //left edge
    outofbounds   |= (pos % width == width-1);  //right edge

    // Outofbounds should be correct here
    //@ assert outofbounds == oob(gtid)
    //@ assert up = pos - width;
    //@ assert down = pos + width;



    //all live cells on the edges die. Period.
    if (outofbounds)
      {
      nextgen[pos] = 0;
      }
    else
      {
      //sum up all the neighbours
      neighbours     = board[up-1]     +board[up]     +board[up+1];
      neighbours    += board[pos-1]            +board[pos+1];
      neighbours    += board[down-1]  +board[down]   +board[down+1];

      nextgen[pos] = (board[pos] && neighbours == 2) || (neighbours ==
          3);
      }
    // make sure that the calculated result for this cell is correct
    //@ assert oob(pos)?nextgen[pos]==0:nextgen[pos]==gol(i,pos)
    }

  //Bpre
  //@ assert \forall int i; i>=gtid && i<width*height && (i-gtid)%
      gtid_size==0;oob(i) ==> nextgen[i]==0;
  //@ assert \forall int i; i>=gtid && i<width*height && (i-gtid)%
      gtid_size==0;!oob(i) ==> nextgen[i]==gol(i)

  barrier(CLK_GLOBAL_MEM_FENCE);

  //Bres
  //flip rights
```

```
    //@ ensures \forall int i; i>=gtid && i<width*height && (i-gtid)%
        gtid_size==0;!oob(i) ==>
    //@   perm(((i%2==0)?B:A)[i-width-1],p) ** perm(((i%2==0)?B:A)[i-width
        ],p) ** perm(((i%2==0)?B:A)[i-width+1],p) **
    //@   perm(((i%2==0)?B:A)[i-1],p) ** perm(((i%2==0)?B:A)[i+1],p) **
        perm(((i%2==0)?B:A)[i+width-1],p) **
    //@   perm(((i%2==0)?B:A)[i+width],p) ** perm(((i%2==0)?B:A)[i+width
        +1],p) ** perm(((i%2==0)?A:B)[i],1)

    //Bpost
    // make sure we calculated the right result
    //@ ensures \forall int n; n>=0 && i<width*height;oob(n) ==> nextgen[n
        ]==0;
    //@ ensures \forall int n; n>=0 && i<width*height;!oob(n) ==> nextgen[
        n]==gol(n,i)


    }

  if (iterations%2==0)
    for (pos = get_global_id(0);pos<width*height;pos += get_global_size(0)
        )
      B[pos] = A[pos];

  }

  //THREAD
  //@ ensures true

  //KERNEL
  //@ ensures
  //@   \forall int j;j>=0 && j<iterations;
  //@     \forall k>=gtid && k<=width*height+group_size && (k-gtid)%
      group_size==0 &&
  //@       ghostboard(j)[k] == gol(j,k);
```

Listing A.4: Localization Optimization Kernel - first version

```
__kernel void Kernel(          __global  unsigned int * B, //nextgen
                    __global  unsigned int * A, //board
                    const     unsigned int height,
                    const     unsigned int width,
                    const     unsigned int iterations)
{

int pos, up, down, outofbounds, neighbours;

//needs the id to span from 0 to height*width AND all to be in the same
    memory (workgroup)
//__local int cached[width*height];
__local int cached[16*128];


for (int i=0;i<iterations;i++)
  {
  for (pos = get_global_id(0);pos<width*height;pos += get_global_size(0)
      )
    cached[pos] = (i%2==0?A:B)[pos];

  barrier(CLK_LOCAL_MEM_FENCE);


  for (pos = get_global_id(0);pos<width*height;pos += get_global_size(0)
      )
    {

    // pos -/+ width gives same position but one row higher/lower
    up     = pos - width;
    down   = pos + width;

    //out of bounds is when pos is on an edge
    outofbounds = (pos < width);           //upper edge
    outofbounds   |= (pos > (width * (height-1)));//lower edge
    outofbounds   |= (pos % width == 0);        //left edge
    outofbounds   |= (pos % width == width-1);  //right edge

    //all life on the edges die. Period.
    if (outofbounds)
      {
      (i%2==0?B:A)[pos] = 0;
      }
    else
      {
      //sum up all the neighbours
      neighbours     = cached[up-1]    +cached[up]    +cached[up+1];
      neighbours    += cached[pos-1]              +cached[pos+1];
      neighbours    += cached[down-1] +cached[down] +cached[down+1];

      (i%2==0?B:A)[pos] = (cached[pos] && neighbours == 2) || (
          neighbours == 3);
      }
    }
  barrier(CLK_GLOBAL_MEM_FENCE);
  }

if (iterations%2==0)
  for (pos = get_global_id(0);pos<width*height;pos += get_global_size(0)
      )
    B[pos] = A[pos];

}
```

Listing A.5: Localization Optimization Kernel

```
// MODEL VARIABLES
// gtid is used as shorthand for the unique (global) thread identifier.
//@ private represents gtid <- get_global_id(0);
// gtid_size returns the total amount of threads.
//@ private represents gtid_size == get_global_size(0);

// oob stands for Out Of Bounds, and tells us whether cell n, is on the
     border of the field.
//@ private represents oob(n) <- (n < width)||(n > (width * (height-1)))
     ||(n % width == 0)||(n % width == width-1);

// nb, gives the amount of direct, live neighbours of cell n at
     iteration i.
//@ private represents nb(i,n) <- ghostboard(i)[n-width-1]+ghostboard(i)
     [n-width]+ghostboard(i)[n-width+1]+ghostboard(i)[n-1]+ghostboard(i)
     [n+1]+ghostboard(i)[n+width-1]+ghostboard(i)[n+width]+ghostboard(i)
     [n+width+1];

// gol returns the expected state of cell n, given the board A and
     iteration i, respecting the rules of the game of life
//@ private represents gol(i,n) <- (ghostboard(i)[n] && nb(i) == 2) || (
     nb(i) == 3) :

// ghostboard gives the array representing the board at a given
     iteration
//@ private represents ghostboard(i) <-  ghost(i)[i%2];
// ghostnextgen gives  array representing nextgen at a given iteration
//@ private represents ghostnextgen(i) <- ghost(i)[1-(i%2)];


// GHOST VARIABLES
// ghost(i), represents cached at generation i. With at i==0, ghost[i
     ][0]==board and ghost[i][1]==nextgen


// THREAD SPECIFICATION
// Tres - resources needed for the thread
  //@ requires perm(width,p) && perm(height,p);
  //@ requires \forall int i; i>=gtid && i<width*height && (i-gtid)%
       gtid_size==0;!oob(i)
  //@    ==> perm(board[i-width-1],p) ** perm(board[i-width],p) ** perm(
       board[i-width+1],p) **
  //@    perm(board[i-1],p) ** perm(board[i+1],p) ** perm(board[i+width
       -1],p) **
  //@    perm(board[i+width],p) ** perm(board[i+width+1],p) ** perm(
       nextgen[i],1)

// Tpre - preconditions for the thread
  // The given size must equal the size of the boards
  //@ requires sizeof(board)==sizeof(nextgen) && sizeof(board)/sizeof(
       int)==(width*height);

// KERNEL SPECIFICATION
// Kres - resources needed for the kernel
  // require write access to each location on the output-board, and the
       input-board
  //@ requires \forall int i; i>=0 && i<width*height; perm(board[i],1)
       ** perm(nextgen[i],1)

  //@ requires perm(width,1) ** perm(height,1) ** perm(iterations,1);

// Kpre - precondtions for the kernel
  // The given size must equal the size of the boards
  //@ requires sizeof(board)==sizeof(nextgen) && sizeof(board)/sizeof(
       int)==(width*height);


__kernel void Kernel(          __global  unsigned int * nextgen, //
     nextgen
                    __global  unsigned int * board, //board
                    const     unsigned int height,
                    const     unsigned int width,
                    const     unsigned int iterations)
```

```
{

int pos, up, down, outofbounds, neighbours;

//needs the id to span from 0 to height*width AND all to be in the same
    memory (workgroup)
__local int cached[2][width*height];

for (pos = get_global_id(0);pos<width*height;pos += get_global_size(0))
  {
  cached[0][pos] = board[pos];
  cached[1][pos] = nextgen[pos];
  }

//Bpre
//@ ensures \forall int i; i>=gtid && i<width*height; (i%gtid_size==0);
    cached[0][i] = board[i] && cached[1][i] = nextgen[i]
//Bres
// true
barrier(CLK_LOCAL_MEM_FENCE);
//Bpost
//@ ensures \forall int i; i>=0 && i<width*height;  cached[0][i] = board
    [i] && cached[1][i] = nextgen[i]


//@ loop_invariant i>=0 && i<=iterations &&
//@   \forall int j;j>=0 && j<i;
//@     \forall k>=gtid && k<=width*height+group_size && (k-gtid)%
    gtid_size==0 &&
//@       ghostboard(j)[k] == gol(j,k);
for (int i=0;i<iterations;i++)
  {

  //@ loop_invariant

  for (pos = get_global_id(0);pos<width*height;pos += get_global_size(0)
      )
    {

    // pos -/+ width gives same position but one row higher/lower
    up    = pos - width;
    down  = pos + width;

    //out of bounds is when pos is on an edge
    outofbounds = (pos < width);          //upper edge
    outofbounds   |= (pos > (width * (height-1)));//lower edge
    outofbounds   |= (pos % width == 0);        //left edge
    outofbounds   |= (pos % width == width-1);  //right edge

    // Outofbounds should be correct here
    //@ assert outofbounds == oob(gtid)
    //@ assert up = pos - width;
    //@ assert down = pos + width;



    //all live cells on the edges die. Period.
    if (outofbounds)
      {
      cached[1-(i%2)][pos] = 0;
      }
    else
      {
      //sum up all the neighbours
      neighbours     = cached[i%2][up-1]  +cached[i%2][up]     +cached[i
          %2][up+1];
      neighbours    += cached[i%2][pos-1]                +cached[i%2][pos
          +1];
      neighbours    += cached[i%2][down-1]  +cached[i%2][down]     +
          cached[i%2][down+1];

      cached[1-(i%2)][pos] = (cached[i%2][pos] && neighbours == 2) || (
          neighbours == 3);
      }
      // make sure that the calculated result for this cell is correct
      //@ assert oob(pos)?cached[1-(i%2)][pos]==0:cached[1-(i%2)][pos]==
          gol(i,pos)
      }

    //Bpre
    //@ assert \forall int i; i>=gtid && i<width*height && (i-gtid)%
        gtid_size==0;oob(i) ==> cached[1-(i%2)][i]==0;
    //@ assert \forall int i; i>=gtid && i<width*height && (i-gtid)%
        gtid_size==0;!oob(i) ==> cached[1-(i%2)][i]==gol(i)

    barrier(CLK_GLOBAL_MEM_FENCE);

    //Bres
    //flip rights
    //@ ensures \forall int i; i>=gtid && i<width*height && (i-gtid)%
        gtid_size==0;!oob(i) ==>
    //@   perm(cached[1-(i%2)][i-width-1],p) ** perm(cached[1-(i%2)][i-
        width],p) ** perm(cached[1-(i%2)][i-width+1],p) **
    //@   perm(cached[1-(i%2)][i-1],p) ** perm(cached[1-(i%2)][i+1],p) **
        perm(cached[1-(i%2)][i+width-1],p) **
    //@   perm(cached[1-(i%2)][i+width],p) ** perm(cached[1-(i%2)][i+width
        +1],p) ** cached[i%2][i],1) **
    //@   perm(nextgen[i],1)

    //Bpost
    // make sure we calculated the right result
    //@ ensures \forall int n; n>=0 && i<width*height;oob(n) ==> nextgen[n
        ]==0;
    //@ ensures \forall int n; n>=0 && i<width*height;!oob(n) ==> nextgen[
        n]==gol(n,i)


    }

// copy everything to the output, we do not need a barrier, since OpenCL
      waits for all the threads to finish, before the host can read the
    result
for (pos = get_global_id(0);pos<width*height;pos += get_global_size(0))
  nextgen[pos] = cached[iterations%2][pos];

}

//THREAD
//@ ensures true

//KERNEL
//@ ensures
//@   \forall int j;j>=0 && j<iterations;
//@     \forall k>=gtid && k<=width*height+group_size && (k-gtid)%
    group_size==0 &&
//@       ghostboard(j)[k] == gol(j,k);
```

Appendix B

Hostcode

Listing B.1: Host code

```
#include "Main.hpp"

/* ================================================================

General host code for "FORMAL SPECICATION AND VERIFICATION OF OPENCL
    KERNEL OPTIMIZATION"
by Jeroen Vonk (2013)

Based on a template by AMD
Copyright (c) 2009 Advanced Micro Devices, Inc.  All rights reserved.

  ================================================================ */


/*
 * \brief Host Initialization
 *        Allocate and initialize memory
 *        on the host. Print input array.
 */
int
initializeHost(void)
{
    width               = pow(2,7);
    input               = NULL;
    output              = NULL;
    height             = pow(2,4);
  size          = width*height;

  // NOTE: work_size equals size when using MAIN IMPLEMENTATION
  work_size       = 1024;
  iterations        = pow(2,20);

  printarray        = false;

    ////////////////////////////////////////////////////////////////
    // Allocate and initialize memory used by host
    ////////////////////////////////////////////////////////////////
    cl_uint sizeInBytes = size * sizeof(cl_uint);
    input = (cl_uint *) malloc(sizeInBytes);
    if(!input)
    {
        std::cout << "Error: Failed to allocate input memory on host\n";
        return SDK_FAILURE;
    }

    output = (cl_uint *) malloc(sizeInBytes);
    if(!output)
    {
        std::cout << "Error: Failed to allocate input memory on host\n";
        return SDK_FAILURE;
    }


  for (cl_uint i=0; i<size; i++)
    input[i] = false;

    return SDK_SUCCESS;
}


/**
 * Loads a saved game of life.
 * @return returns SDK_SUCCESS on success and the pattern fits in the
       field and SDK_FAILURE otherwise
 */
int loadField(void)
{

  std::string const filename  = "input.txt";

  std::ifstream file(filename.c_str());
  file.ignore( 256, '\n' );

    cl_uint x,y;

  std::cout << "Setting␣following␣cells␣alive:" << std::endl;

    while (file >> x >> y)
  {
  std::cout << "␣␣" << x << ":" << y << std::endl;
    if (x>0 && x<width-1 && y>0 && y<height-1) {
      input[x+(y*width)] = true;
    } else {
      std::cout << "Error:␣Cell␣out␣of␣bounds" << std::endl;
      return SDK_FAILURE;
    }
  }
  std::cout << std::endl;
    print2DArray(std::string("Input").c_str(), input, height, width);
  return SDK_SUCCESS;
}

/**
 * Saves the field given
 * @return returns SDK_SUCCESS on success and SDK_FAILURE otherwise
 */
int saveField(void)
{
  std::string const filename  = "output.txt";
  std::ofstream file(filename.c_str(),std::ios::trunc);
  file << "#Life␣1.06" << std::endl;

  for (cl_uint i=0; i<size; i++){
    if (input[i])
      file << i%width << "␣" << i/width << std::endl;
  }
  return SDK_SUCCESS;
}



/*
 * Converts the contents of a file into a string
 */
std::string
convertToString(const char *filename)
{
    size_t size;
    char*  str;
    std::string s;

    std::fstream f(filename, (std::fstream::in | std::fstream::binary));
    if(f.is_open())
    {
        size_t fileSize;
        f.seekg(0, std::fstream::end);
        size = fileSize = (size_t)f.tellg();
        f.seekg(0, std::fstream::beg);

        str = new char[size+1];
        if(!str)
        {
            f.close();
            std::cout << "Memory␣allocation␣failed";
            return NULL;
        }

        f.read(str, fileSize);
        f.close();
        str[size] = '\0';

        s = str;
        delete[] str;
        return s;
    }
    else
    {
        std::cout << "\nFile␣containg␣the␣kernel␣code(\".cl\")␣not␣found
            .␣Please␣copy␣the␣required␣file␣in␣the␣folder␣containg␣the␣
            executable.\n";
        exit(1);
    }
    return NULL;
}

/*
 * \brief OpenCL related initialization
 *        Create Context, Device list, Command Queue
 *        Create OpenCL memory buffer objects
 *        Load CL file, compile, link CL source
 *      Build program and kernel objects
 */
int
initializeCL(void)
{
    cl_int status = 0;
    size_t deviceListSize;

    ///////////////////////////////////////////////////////////////////
    // STEP 1 Getting Platform.
    ///////////////////////////////////////////////////////////////////

    /*
     * Have a look at the available platforms and pick either
     * the NVIDIA one if available or a reasonable default.
     */

    cl_uint numPlatforms;
    cl_platform_id platform = NULL;
    status = clGetPlatformIDs(0, NULL, &numPlatforms);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error:␣Getting␣Platforms.␣(clGetPlatformsIDs)\n";
        return SDK_FAILURE;
    }

    if(numPlatforms > 0)
    {
        cl_platform_id* platforms = new cl_platform_id[numPlatforms];
        status = clGetPlatformIDs(numPlatforms, platforms, NULL);
        if(status != CL_SUCCESS)
        {
            std::cout << "Error:␣Getting␣Platform␣Ids.␣(
                clGetPlatformsIDs)\n";
            return SDK_FAILURE;
        }
        for(unsigned int i=0; i < numPlatforms; ++i)
        {
            char pbuff[100];
            status = clGetPlatformInfo(
                        platforms[i],
                        CL_PLATFORM_VENDOR,
                        sizeof(pbuff),
                        pbuff,
                        NULL);
            if(status != CL_SUCCESS)
            {
                std::cout << "Error:␣Getting␣Platform␣Info.(
                    clGetPlatformInfo)\n";
                return SDK_FAILURE;
            }
            platform = platforms[i];

            if(!strcmp(pbuff, "NVIDIA␣Corporation"))
            {
                break;
            }
        }
        delete platforms;
    }

    if(NULL == platform)
    {
        std::cout << "NULL␣platform␣found␣so␣Exiting␣Application." <<
            std::endl;
        return SDK_FAILURE;
    }


    /////////////////////////////////////////////////////////////////////
    // STEP 2 Creating context using the platform selected
    //      Context created from type includes all available
    //      devices of the specified type from the selected platform
    /////////////////////////////////////////////////////////////////////


    /*
     * If we could find our platform, use it. Otherwise use just
         available platform.
     */
    cl_context_properties cps[3] = { CL_CONTEXT_PLATFORM, (
        cl_context_properties)platform, 0 };

    context = clCreateContextFromType(cps,
                                      CL_DEVICE_TYPE_GPU,
                                      NULL,
                                      NULL,
                                      &status);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error:␣Creating␣Context.␣(clCreateContextFromType)
            \n";
        return SDK_FAILURE;
    }


    /////////////////////////////////////////////////////////////////////
    // STEP 3
    //      3.1 Query context for the device list size,
    //      3.2 Allocate that much memory using malloc or new
    //      3.3 Again query context info to get the array of device
    //              available in the created context
    /////////////////////////////////////////////////////////////////////

    // First, get the size of device list data
    status = clGetContextInfo(context,
                              CL_CONTEXT_DEVICES,
                              0,
                              NULL,
                              &deviceListSize);
    if(status != CL_SUCCESS)
    {
        std::cout <<
            "Error: Getting Context Info \
            (device list size, clGetContextInfo)\n";
        return SDK_FAILURE;
    }

    devices = (cl_device_id *)malloc(deviceListSize);
    if(devices == 0)
    {
        std::cout << "Error: No devices found.\n";
        return SDK_FAILURE;
    }

    // Now, get the device list data
    status = clGetContextInfo(
                 context,
                 CL_CONTEXT_DEVICES,
                 deviceListSize,
                 devices,
                 NULL);
    if(status != CL_SUCCESS)
    {
        std::cout <<
            "Error: Getting Context Info \
            (device list, clGetContextInfo)\n";
        return SDK_FAILURE;
    }

    /////////////////////////////////////////////////////////////////
    // STEP 4 Creating command queue for a single device
    //        Each device in the context can have a
    //        dedicated commandqueue object for itself
    /////////////////////////////////////////////////////////////////

    commandQueue = clCreateCommandQueue(
                       context,
                       devices[0],
                       0,
                       &status);
    if(status != CL_SUCCESS)
    {
        std::cout << "Creating Command Queue. (clCreateCommandQueue)\n";
        return SDK_FAILURE;
    }

    /////////////////////////////////////////////////////////////////
    // STEP 5 Creating cl_buffer objects from host buffer
    //          These buffer objects can be passed to the kernel
    //          as kernel arguments
    /////////////////////////////////////////////////////////////////
    inputBuffer = clCreateBuffer(
                      context,
                      CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                      sizeof(cl_uint) * size,
                      input,
                      &status);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error: clCreateBuffer (inputBuffer)\n";
        return SDK_FAILURE;
    }

    outputBuffer = clCreateBuffer(
                       context,
                      CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                       sizeof(cl_uint) * size,
                       output,
                       &status);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error: clCreateBuffer (outputBuffer)\n";
        return SDK_FAILURE;
    }


    //////////////////////////////////////////////////////////////////
    // STEP 6. Building Kernel
    //       6.1 Load CL file , using basic file i/o
    //       6.2 Build CL program object
    //       6.3 Create CL kernel object
    //////////////////////////////////////////////////////////////////
    const char * filename  = "Kernel.cl";
    std::string  sourceStr = convertToString(filename);
    const char * source    = sourceStr.c_str();
    size_t sourceSize[]    = { strlen(source) };

    program = clCreateProgramWithSource(
                  context,
                  1,
                  &source,
                  sourceSize,
                  &status);
    if(status != CL_SUCCESS)
    {
      std::cout <<
                "Error: Loading Binary into cl_program \
                (clCreateProgramWithBinary)\n";
      return SDK_FAILURE;
    }

    // create a cl program executable for all the devices specified
    status = clBuildProgram(program, 1, devices, NULL, NULL, NULL);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error: Building Program (clBuildProgram)\n";

        // Determine the size of the log
        size_t log_size;
        clGetProgramBuildInfo(program, devices[0], CL_PROGRAM_BUILD_LOG,
             0, NULL, &log_size);

        // Allocate memory for the log
        char *log = (char *) malloc(log_size);

        // Get the log
        clGetProgramBuildInfo(program, devices[0], CL_PROGRAM_BUILD_LOG,
             log_size, log, NULL);

        // Print the log
        printf("%s\n", log);

        return SDK_FAILURE;
    }



    // get a kernel object handle for a kernel with the given name
    kernel = clCreateKernel(program, "Kernel", &status);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error: Creating Kernel from program. (
             clCreateKernel)\n";
        return SDK_FAILURE;
    }

    return SDK_SUCCESS;
}


/*
 * \brief Run OpenCL program
 *
 *        Bind host variables to kernel arguments
 *        Run the CL kernel
 */
int
```

```
runCLKernels(void)
{
    cl_int    status;
    cl_uint maxDims;
    cl_event events[2];
    size_t globalThreads[1];
    size_t localThreads[1];
    size_t maxWorkGroupSize;
    size_t maxWorkItemSizes[3];

    //////////////////////////////////////////////////////////////////////
    // STEP 7 Analyzing proper workgroup size for the kernel
    //           by querying device information
    //     7.1 Device Info CL_DEVICE_MAX_WORK_GROUP_SIZE
    //     7.2 Device Info CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS
    //     7.3 Device Info CL_DEVICE_MAX_WORK_ITEM_SIZES
    //////////////////////////////////////////////////////////////////////



    /**
    * Query device capabilities. Maximum
    * work item dimensions and the maximmum
    * work item sizes
    */
    status = clGetDeviceInfo(
        devices[0],
        CL_DEVICE_MAX_WORK_GROUP_SIZE,
        sizeof(size_t),
        (void*)&maxWorkGroupSize,
        NULL);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error: Getting Device Info. (clGetDeviceInfo)\n";
        return SDK_FAILURE;
    }

    status = clGetDeviceInfo(
        devices[0],
        CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS,
        sizeof(cl_uint),
        (void*)&maxDims,
        NULL);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error: Getting Device Info. (clGetDeviceInfo)\n";
        return SDK_FAILURE;
    }

    status = clGetDeviceInfo(
        devices[0],
        CL_DEVICE_MAX_WORK_ITEM_SIZES,
        sizeof(size_t)*maxDims,
        (void*)maxWorkItemSizes,
        NULL);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error: Getting Device Info. (clGetDeviceInfo)\n";
        return SDK_FAILURE;
    }

    cl_uint size_t;

    status = clGetDeviceInfo(
        devices[0],
        CL_DEVICE_ADDRESS_BITS,
        sizeof(cl_uint),
        (void*)&size_t,
        NULL);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error: Getting Device Info. (
            CL_DEVICE_ADDRESS_BITS)\n";
        return SDK_FAILURE;
        }

        if (work_size>pow(2,size_t))
        {
          std::cout << "Error:␣array␣to␣big␣for␣this␣implementation\n";
            return SDK_FAILURE;
        } else
          globalThreads[0] = work_size;


        if (size>maxWorkGroupSize) {
          globalThreads[0] = maxWorkGroupSize;
          std::cout << std::endl << "Implementation␣runs␣on␣maxWorkGroupSize
               ␣("<<maxWorkGroupSize<<")\n";
        }
        localThreads[0]  = globalThreads[0];

        if(localThreads[0] > maxWorkGroupSize ||
            localThreads[0] > maxWorkItemSizes[0])
        {
            std::cout << "Unsupported:␣Device␣does␣not␣support␣requested␣
                 number␣of␣work␣items.";
            return SDK_FAILURE;
        }

        //Print info
        printDevice();

        /////////////////////////////////////////////////////////////////////
        // STEP 8 Set appropriate arguments to the kernel
        //     8.1 Kernel Arg outputBuffer ( cl_mem object)
        //     8.2 Kernel Arg inputBuffer (cl_mem object)
        //     8.3 Kernel Arg height (cl_uint)
        //     8.4 Kernel Arg width (cl_uint)
        //     8.5 Kernel Arg iterations (cl_uint)
        /////////////////////////////////////////////////////////////////////

        // the output array to the kernel
        status = clSetKernelArg(
                        kernel,
                        0,
                        sizeof(cl_mem),
                        (void *)&outputBuffer);
        if(status != CL_SUCCESS)
        {
            std::cout << "Error:␣Setting␣kernel␣argument.␣(output)\n";
            return SDK_FAILURE;
        }

        // the input array to the kernel
        status = clSetKernelArg(
                        kernel,
                        1,
                        sizeof(cl_mem),
                        (void *)&inputBuffer);
        if(status != CL_SUCCESS)
        {
            std::cout << "Error:␣Setting␣kernel␣argument.␣(input)\n";
            return SDK_FAILURE;
        }

        // height
        status = clSetKernelArg(
                        kernel,
                        2,
                        sizeof(cl_uint),
                        (void *)&height);
        if(status != CL_SUCCESS)
        {
            std::cout << "Error:␣Setting␣kernel␣argument.␣(height)\n";
            return SDK_FAILURE;
        }
```

```
    // width
    status = clSetKernelArg(
                    kernel,
                    3,
                    sizeof(cl_uint),
                    (void *)&width);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error:␣Setting␣kernel␣argument.␣(width)\n";
        return SDK_FAILURE;
    }

  // NOTE: iterations is not passed to the MAIN IMPLEMENTATION
    // iterations
    status = clSetKernelArg(
                    kernel,
                    4,
                    sizeof(cl_uint),
                    (void *)&iterations);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error:␣Setting␣kernel␣argument.␣(iterations)\n";
        return SDK_FAILURE;
    }
  // NOTE: iterations is not passed to the MAIN IMPLEMENTATION

    ///////////////////////////////////////////////////////////////////
    // STEP 9 Enqueue a kernel run call.
    //          Wait till the event completes and release the event
    ///////////////////////////////////////////////////////////////////

    std::cout << std::endl << "Running␣for␣" << iterations << "␣
        iterations\n";

  // NOTE: a for-loop is used in the MAIN IMPLEMENTATION
  //
//////  for (int i=1; i <= iterations; ++i) {
  //


  status = clEnqueueNDRangeKernel(
          commandQueue,
          kernel,
          1,
          NULL,
          globalThreads,
          localThreads,
          0,
          NULL,
          &events[0]);
  if(status != CL_SUCCESS)
  {
    std::cout <<
      "Error:␣Enqueueing␣kernel␣onto␣command␣queue.␣\
␣␣␣␣␣␣(clEnqueueNDRangeKernel)\n";
    return SDK_FAILURE;
  }


  // wait for the kernel call to finish execution

  status = clReleaseEvent(events[0]);
  if(status != CL_SUCCESS)
  {
    std::cout <<
      "Error:␣Release␣event␣object.␣\
␣␣␣␣␣␣(clReleaseEvent)\n";
    return SDK_FAILURE;
  }

    ///////////////////////////////////////////////////////////////////
    // STEP 10  Enqueue readBuffer to read the output back
    //  Wait for the event and release the event
    ///////////////////////////////////////////////////////////////////
    status = clEnqueueReadBuffer(
                commandQueue,
                outputBuffer,
                CL_TRUE,
                0,
                size * sizeof(cl_uint),
                output,
                0,
                NULL,
                &events[1]);

    if(status != CL_SUCCESS)
    {
        std::cout <<
            "Error:␣clEnqueueReadBuffer␣failed.␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣(clEnqueueReadBuffer)\n";
        return SDK_FAILURE;
    }

    // Wait for the read buffer to finish execution
    status = clWaitForEvents(1, &events[1]);
    if(status != CL_SUCCESS)
    {
        std::cout <<
            "Error:␣Waiting␣for␣read␣buffer␣call␣to␣finish.␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣(clWaitForEvents)\n";
        return SDK_FAILURE;
    }

    status = clReleaseEvent(events[1]);
    if(status != CL_SUCCESS)
    {
        std::cout <<
            "Error:␣Release␣event␣object.␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣(clReleaseEvent)\n";
        return SDK_FAILURE;
    }

  // NOTE: a buffer swap is used in MAIN IMPLEMENTATION

//////int error;
//////error = clEnqueueCopyBuffer(commandQueue, outputBuffer,
    inputBuffer, 0, 0, size*sizeof(cl_uint), 0, NULL, NULL );
//////if (error != CL_SUCCESS)
//////{
//////  std::cout <<
//////     "Error: Waiting for buffers to copy. \
//////     (clEnqueueCopyBuffer)\n";
//////  return SDK_FAILURE;
//////}

  // NOTE: a for-loop is used in the MAIN IMPLEMENTATION

////// } \\ end of the for-loop

    return SDK_SUCCESS;
}


/*
 * \brief Release OpenCL resources (Context, Memory etc.)
 */
int
cleanupCL(void)
{
    cl_int status;

    ///////////////////////////////////////////////////////////////////
    // STEP 11  CLean up the opencl resources used
    ///////////////////////////////////////////////////////////////////

    status = clReleaseKernel(kernel);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error:␣In␣clReleaseKernel␣\n";
        return SDK_FAILURE;
    }
    status = clReleaseProgram(program);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error:␣In␣clReleaseProgram\n";
        return SDK_FAILURE;
    }
    status = clReleaseMemObject(inputBuffer);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error:␣In␣clReleaseMemObject␣(inputBuffer)\n";
        return SDK_FAILURE;
    }
    status = clReleaseMemObject(outputBuffer);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error:␣In␣clReleaseMemObject␣(outputBuffer)\n";
        return SDK_FAILURE;
    }
    status = clReleaseCommandQueue(commandQueue);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error:␣In␣clReleaseCommandQueue\n";
        return SDK_FAILURE;
    }
    status = clReleaseContext(context);
    if(status != CL_SUCCESS)
    {
        std::cout << "Error:␣In␣clReleaseContext\n";
        return SDK_FAILURE;
    }
    return SDK_SUCCESS;
}


/*
 * \brief Releases program's resources
 */
void
cleanupHost(void)
{
    if(input != NULL)
    {
        free(input);
        input = NULL;
    }
    if(output != NULL)
    {
        free(output);
        output = NULL;
    }
    if(devices != NULL)
    {
        free(devices);
        devices = NULL;
    }
}


/*
 * \brief Print no more than 256 elements of the given array.
 *
 *        Print Array name followed by elements.
 */
void print1DArray(
         const std::string arrayName,
         const unsigned int * arrayData,
         const unsigned int length)
{
    cl_uint i;
    cl_uint numElementsToPrint = (256 < length) ? 256 : length;
    std::cout << std::endl;
    std::cout << arrayName << ":" << std::endl;
    for(i = 0; i < numElementsToPrint; ++i)
    {
        std::cout << arrayData[i] << " ";
    }
    std::cout << std::endl;

}


/*
 * \brief Print no more than 256 elements of the given array.
 *
 *        Print Array name followed by elements.
 */
void print2DArray(
         const std::string arrayName,
         const cl_uint * arrayData,
         const unsigned int height,
     const unsigned int width)
{
  if (! printarray)
    return;

    cl_uint x,y;
    cl_uint numElementsToPrint = (256 < length) ? 256 : length;

    std::cout << std::endl;
    std::cout << arrayName << ":" << std::endl;
    for(y = 0; y < height; ++y)
    {
      for(x = 0; x < width; ++x)
    {
      std::cout << (arrayData[(y*width)+x]?"X":"-");// << " ";
    }
  std::cout << std::endl;
  }
    std::cout << std::endl;

}

void printTime(std::string text) {
  struct timeval cur;
  gettimeofday(&cur, NULL);

  long long t1 = ((cur.tv_sec*1000) + (cur.tv_usec/1000) + 0.5) - ((
      delta.tv_sec*1000) + (delta.tv_usec/1000) + 0.5);
  long long t2 = ((cur.tv_sec*1000) + (cur.tv_usec/1000) + 0.5) - ((
      start.tv_sec*1000) + (start.tv_usec/1000) + 0.5);;
  gettimeofday(&delta, NULL);

  std::cout << std::endl << text << ": " << std::endl;
  std::cout << "  delta: " << t1 << " ms" << std::endl;
    std::cout << "  total: " << t2 << " ms" << std::endl;
}


void printDevice() {
    char* value;
    size_t valueSize;
    cl_uint maxComputeUnits;

    std::cout << std::endl;

            // print device name
            clGetDeviceInfo(devices[0], CL_DEVICE_NAME, 0, NULL, &
                valueSize);
            value = (char*) malloc(valueSize);
            clGetDeviceInfo(devices[0], CL_DEVICE_NAME, valueSize, value
                , NULL);
            printf("Device: %s\n",value);
            free(value);
            // print hardware device version
            clGetDeviceInfo(devices[0], CL_DEVICE_VERSION, 0, NULL, &
                valueSize);
            value = (char*) malloc(valueSize);
            clGetDeviceInfo(devices[0], CL_DEVICE_VERSION, valueSize,
                value, NULL);
            printf("  Hardware version: %s\n", value);
            free(value);

            // print software driver version
            clGetDeviceInfo(devices[0], CL_DRIVER_VERSION, 0, NULL, &
                valueSize);
            value = (char*) malloc(valueSize);
            clGetDeviceInfo(devices[0], CL_DRIVER_VERSION, valueSize,
                value, NULL);
            printf("  Software version: %s\n", value);
            free(value);

            // print c version supported by compiler for device
            clGetDeviceInfo(devices[0], CL_DEVICE_OPENCL_C_VERSION, 0,
                NULL, &valueSize);
            value = (char*) malloc(valueSize);
            clGetDeviceInfo(devices[0], CL_DEVICE_OPENCL_C_VERSION,
                valueSize, value, NULL);
            printf("  OpenCL C version: %s\n", value);
            free(value);

            // print parallel compute units
            clGetDeviceInfo(devices[0], CL_DEVICE_MAX_COMPUTE_UNITS,
                    sizeof(maxComputeUnits), &maxComputeUnits, NULL);
            printf("  Parallel compute units: %d\n", maxComputeUnits);

}



int
main(int argc, char * argv[])
{
  //init clock
  gettimeofday(&delta, NULL);
  gettimeofday(&start, NULL);


  // Initialize Host application
    if(initializeHost() != SDK_SUCCESS)
        return SDK_FAILURE;

    // Load the field
    if(loadField() != SDK_SUCCESS)
        return SDK_FAILURE;

    printTime("Hostside");


    // Initialize OpenCL resources
    if(initializeCL() != SDK_SUCCESS)
        return SDK_FAILURE;

    printTime("Initialized CL");

    // Run the CL program
    if(runCLKernels() != SDK_SUCCESS)
        return SDK_FAILURE;

    printTime("Ran Kernels, loaded result");



    // Print output array
  print2DArray("Output: ", output, height, width);
    printTime("Printing");

    // Save the field
    if(saveField() != SDK_SUCCESS)
        return SDK_FAILURE;


    // Releases OpenCL resources
    if(cleanupCL()!= SDK_SUCCESS)
        return SDK_FAILURE;

    // Release host resources
    cleanupHost();

    printTime("Done");

    return SDK_SUCCESS;
}
```