Solutions to Exercises in Chapter 3

Exercise 1 – Examine the sequential code

Prefix the *time* command before the executable to measure the runtime of the sequential code. It should be not more than 15 seconds for the default grid size (1024) and $2^1$1 game steps. A simple profile can be acquired with the following three steps for PGI and GNU compilers:

1. Compile the program with the *-pg* option.
   PGI: *pgcc -pg gol_ex1_seq.c -o gol_ex1_seq*
   GNU: *gcc -pg gol_ex1_seq.c -o gol_ex1_seq*
2. Execute the program ./gol_ex1_seq. The file gmon.out is generated.
3. Analyze the profile by executing *gprof ./gol_ex1_seq*.

The output of *gprof* should look similar to the following:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 99.87     15.46    15.46     2048     7.55     7.55  gol
  0.13     15.48     0.02                             main
```

Obviously, the the gol-routine, which is called in a for-loop over the game steps, is dominating the program runtime. To further investigate the runtime within the gol-routine, manual instrumentation can be applied. In case of Score-P this requires to add the following lines:

```
#include <scorep/SCOREP_User.h>

SCOREP_USER_REGION_DEFINE( my_region )
// more declarations

SCOREP_USER_REGION_BEGIN( my_region, "foo", \\
                          SCOREP_USER_REGION_TYPE_COMMON )
// do something
SCOREP_USER_REGION_END( my_region )
```

For further details read the Score-P manual (www.score-p.org).

Exercise 2 – Identify independent loops

The game of life is trivial to parallelize. Loops in the gol-routine do not carry data dependencies. Annotate the loops with `#pragma acc loop independent`.

Use the kernels construct around the loops and loop nests of the gol-routine. Use the compiler option *-acc* to evaluate OpenACC directives in the code. Use *-Minfo=accel* to get compiler feedback on the OpenACC directives in the code. The PGI compiler provides some output like “Loop is parallelizable”, if the loops have been annotated.

1. Compiler the program:
   pgcc -acc -Minfo=accel gol_ex2_loops_error.c -o gol_ex2_loops_error
2. Execute the program: *./gol_ex2_loops_error*
3. Output: “Illegal address during kernel execution”

The error output tells us that a data access was invalid. The size of `grid` and `newGrid` cannot be determined and respective copy operations not performed correctly. Hence, we need to add the following data clauses to the kernels construct: `copy(grid[0:arraySize]) create(newGrid[0:arraySize])`.

The runtime of the correct OpenACC code (*gol_ex2_loops.c*) can be determined as in exercise 1 with the *time* command. It should be less than the runtime of the sequential code. More detailed runtime information on the OpenACC program can be determined with *pgprof*. The command *pgprof* can be simply prefixed before the executable. The output should contain profiling results, similar to the following:

```
pgprof ./gol_ex2_loops

==17727== Profiling result:
Time(%)      Time   Calls       Avg       Min       Max  Name
 39.23%   1.1324s    2048  552.9us  547.0us  561.1us  [Memcpy HtoD]
 39.14%   1.1297s    2048  551.6us  546.9us  557.8us  [Memcpy DtoH]
 14.89%   429.8ms    2048  209.9us  209.1us  217.1us  gol_38_gpu
  6.14%   177.3ms    2048  86.58us  86.27us  87.45us  gol_65_gpu
  0.38%   11.11ms    2048  5.420us  5.280us  6.495us  gol_26_gpu
  0.21%   5.942ms    2048  2.901us  2.816us  3.936us  gol_16_gpu
...
```

The output provides also information on individual CUDA API calls and OpenACC activities. However, the most obvious issue are the data transfers between host and device, where most of the runtime is spent.

Exercise 3 – Data Region

Add a data region around the loop that triggers a game step. Copy the `grid` to and from the accelerator and create the `newGrid` field on the accelerator. The kernels construct in the gol-routine needs to know that `grid` and `newGrid` are already present. The file *gol_ex3_data.c* contains the respective changes.

The performance improvement should gain another factor compared to the code from exercise 2. Use *pgprof* to get details on the runtime improvements.

```
pgprof ./gol_ex3_data

==18369== Profiling result:
Time(%)      Time   Calls       Avg       Min       Max  Name
 68.69%  428.7ms    2048  209.3us  208.60us  216.44us  gol_38_gpu
 28.39%  177.2ms    2048  86.50us  86.206us  87.134us  gol_65_gpu
  1.80%  11.21ms    2048  5.472us  5.3110us  6.9120us  gol_26_gpu
  0.95%  5.935ms    2048  2.898us  2.8160us  4.0640us  gol_16_gpu
  0.09%  553.6us       1  553.6us  553.56us  553.56us  [Memcpy HtoD]
  0.09%  549.0us       1  549.0us  548.98us  548.98us  [Memcpy DtoH]
...
```

The CUDA kernels that have been generated from the OpenACC code determine the program runtime. Data transfers are reduce to one to the device and one from the device.

Exercise 4 – Alive

It is also possible to count the living cells on the GPU using parallel construct with the reduction clause. This might reduce the runtime for large grid sizes, as the grid does not need to be copied back to the host. The profile shows that there are two more kernels generated for the loop nest in the *main*-routine and one additional host-to-device data transfer. The OpenACC profile shows that most of the runtime is spent in *acc_wait@gol_ex4_alive.c:65*

```
pgprof ./gol_ex4_alive

==18760== Profiling result:
Time(%)       Time  Calls        Avg        Min        Max  Name
 68.75%  428.83ms   2048  209.39us  208.60us  216.76us  gol_38_gpu
 28.41%  177.22ms   2048  86.532us  86.270us  87.519us  gol_65_gpu
  1.80%  11.200ms   2048  5.4680us  5.2800us  6.4320us  gol_26_gpu
  0.95%  5.9453ms   2048  2.9030us  2.8160us  4.0320us  gol_16_gpu
  0.09%  552.09us      2  276.04us  2.2080us  549.88us
[Memcpy HtoD]
  0.01%  41.215us      1  41.215us  41.215us  41.215us
main_114_gpu
  0.00%  7.1680us      1  7.1680us  7.1680us  7.1680us
main_114_gpu_red
  0.00%  3.8710us      1  3.8710us  3.8710us  3.8710us
[Memcpy DtoH]

==18760== OpenACC (excl):
Time(%)       Time     Calls        Avg        Min        Max  Name
  87.79%  623.84ms      2048  304.61us  115.50us  459.38us
acc_wait@gol_ex4_alive.c:65
...
```

Exercise 5 – In-depth Analysis

Use a system where Score-P, Cube, and Vampir are installed. Otherwise, you may install the tools yourself or use the NVIDIA profiling tools in case your OpenACC target device is a CUDA-capable GPU.

a) Instrument your optimized OpenACC GoL code with Score-P using the following command:

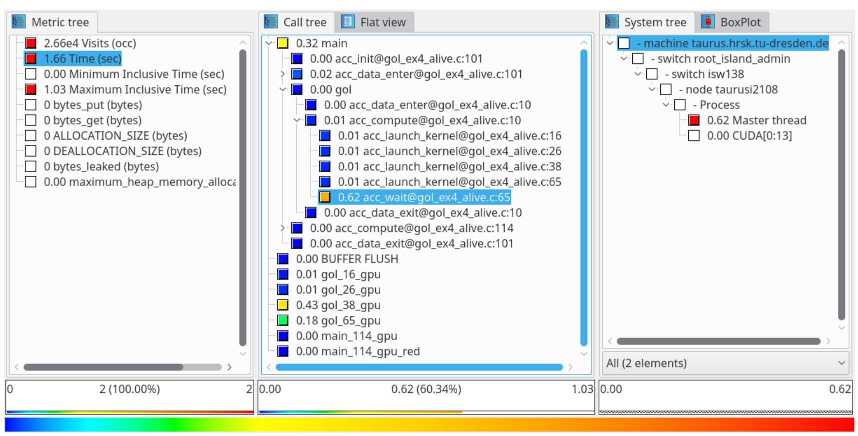
   scorep –openacc –cuda pgcc -acc -Minfo=accel gol_ex4_alive.c -o gol_scorep

   This assumes that we want to use a CUDA target. Otherwise *–cuda* is not needed.

b) Set the following environment variables to record OpenACC and CUDA events:

```
export ACC_PROFLIB=/path/to/libscorep_adapter_openacc_event.so
export SCOREP_OPENACC_ENABLE=1,kernel_properties
export SCOREP_CUDA_ENABLE=kernel,memcpy,flushatexit
export SCOREP_EXPERIMENT_DIRECTORY=gol_profile
```

c) Generate a profile by simply executing the generated executable *gol_scorep*. The directory *gol_profile* is generated. It contains the profiling results in file *profile.cubex*. Visualize the profile with the CUBE GUI by calling *cube gol_profile/profile.cubex*.

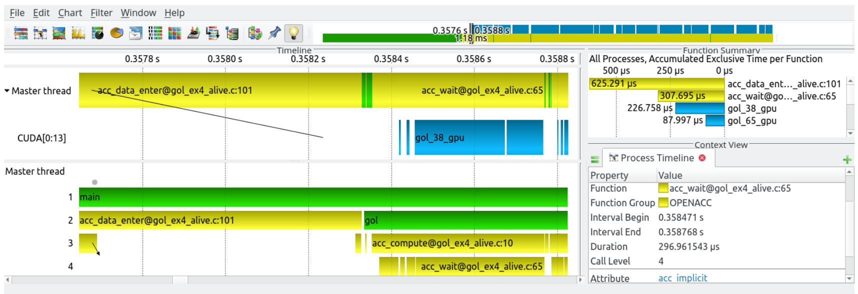


Cube highlights the severity of a metric with a color coding from blue to red. The most runtime consuming program region is the *acc_wait* operation.

d) Enable tracing mode and change the experiment directory:

```
export SCOREP_ENABLE_TRACING=true
export SCOREP_ENABLE_PROFILING=false
export SCOREP_EXPERIMENT_DIRECTORY=gol_trace
```

Re-run the executable. The directory *gol_trace* is generated. It contains the tracing results and provides the anchor file *traces.otf2*. Open the file *gol_trace/traces.otf2* with Vampir.



The screenshot shows the data transfer of the initial grid (black line between *Masterthread* and CUDA stream 13 on device 0) and the first game iteration, which is dominated by the CUDA kernels *gol_38_gpu* and *gol_65_gpu*. The host is waiting for each game iteration while the GPU is computing. The *acc_wait* is implicitly generated by the OpenACC runtime (see *Context View* on the bottom right).

To squeeze out even more performance, it is possible to asynchronously launch the kernels and wait for them before counting the final living cells (see *gol_async.c*).