High Performance Parallel Computing Assignment 5

António Maschio ; Dimitrios Anastasiou ; Georgios Sevastakis

March 2024

1 Task 1: Open Acc parallelise the program

In this assigment we had to implement OpenACC in order to parallelize a shallow waters simulation using GPU. The parts of the code that can be executed by a GPU are the exchange of the ghost lines and the calculations of the velocities and the elevation. We created two different versions of the parallelization. In the first one, after the data transfer that copies the water information into the GPU so it can be accessed by the threads, one big parallel region was created before calling the integrate function. Inside the integrate, each loop was parallelized and *#pragma acc wait* were added in order to synchronize the gangs. In the second version we moved the implementation of the boundary conditions inside the integrate function in order to avoid function calls as we got better results without them. We first created a data region inside the simulate function and then the integrate function is called. Inside the integrate function, three different parallel regions were initiated, one for the exchange of the ghost lines, one for the calculation of the velocities and the last one for the elevation. In table 1 we can see how the runtime is divided into different operations. In the first version, 99.6% of the time was spent on waiting for synchronization. In the second version, we avoided this waiting time with the cost of increasing the time of launching the kernels. Now, while still a big portion of the time is spent waiting, the second biggest bottleneck was memory allocation. For task 2, we used the second version of our code which seemed to perform better.

operation	*sw_parallel_oldv.cpp*	*sw_parallel.cpp*
cuStreamSynchronize	99.6%	45.8%
cuMemHostAlloc	0.4%	43.4%
cuLaunchKernel	-	7.3%

Table 1: Time spent on different operations.

2 Task 2: Strong and weak scaling

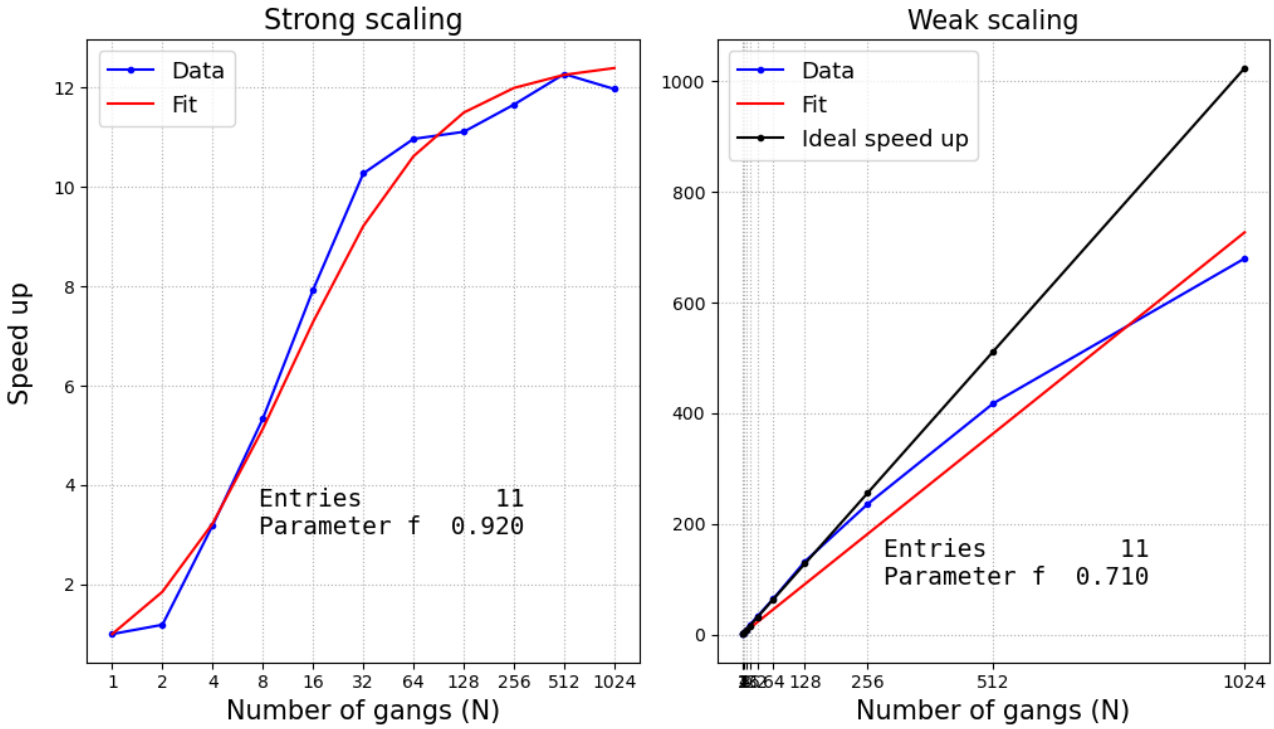

Figure 1: Speed up as a function of the number of gangs. For the strong scaling (left), the workload $N_x \cdot N_y$ was fixed and set to $(N_x, N_y) = (512, 512)$, while for the weak scaling (right), the workload $N_x \cdot N_y$ (left) was proportional to the number of gangs (N) with $N_x \cdot N_y = 16 * 16 * N$ (linear scaling).

Having calculated the Speed up S by dividing the elapsed time of the sequential version (1 gang) with that of the parallelized version (N gangs), we plotted the graphs above and fitted Ahmdahl's (left) and Gustafson's (right) law in the strong and weak scaling context respectively.

Ahmdahl's and Gustafson's laws are given by the following formulas

$$S_{Ahm.}(N) = \frac{1}{1 - f + \frac{f}{N}} \tag{1}$$

$$S_{Gus.}(N) = 1 + (N - 1)\, f \tag{2}$$

By fitting Equations 1 and 2, we were able to compute the parallel fractions f in each context, which were found to be $f = 0.902$ from Ahmdahl's law and $f = 0.710$ from Gustafson's law.

Since, physically, we have a total of 14 multiprocessors (SM's), we would expect the performance/speed up to peak when we choose 14 gangs. But from the left subplot of Figure 1. we notice that the speed up continues to increase after $N = 14$, having a linear trend until it reaches $N = 32$, after which the rate drops. The upward fashion, though, does not end until $N = 512$ is reached. To explain this phenomenon, we profiled our code using

$$nv - nsight - cu - cli - -clock - controlnone./sw\_parallel$$

By using this command we were able to monitor the occupancy percentage of our multiprocessors. We will restrict the discussion to the results for the last for–loop at line 135 of our code.

We found out that occupancy remains the same (6.25%) for $N \leq 14$, while it starts increasing after

$N = 15$. At $N = 512$ occupancy reached the outstanding number of 88.69%. This means that for $N = 512$ there are fewer idle cycles on the SM, as there are more active warps (and therefore, threads) ready to execute instructions. Consequently, the GPU's computational resources, such as execution units and registers, are better utilized, leading to increased throughput.

As far as weak scaling is concerned (right subplot of Figure 1.), we notice that our implementation performs well (good scalability), especially for $N \leq 128$, since our curve overlaps with the ideal one. For $N > 128$ the curve is not linear anymore but starts crumbling.

To have ideal speed up, we need to keep the time of the parallelizable versions the same when increasing the number of gangs and the workload. One of the main reasons the ideal speed up can't be reached is the data transfer from the CPU to the GPU and back (memcopy). This is due to the fact that memcopy depends only on the problem size and is therefore, independent of the number of gangs. Apart from that, the overheads of launching the kernels (significant in our case, since in one integration step we have 3 launches) and of the data synchronization are also independent of N. All of the aforementioned reasons lead to time penalties which are particularly visible when the trade off between them and parallelization tilts towards the former.

Appendices

A C++ code

```cpp
#include <vector>
#include <iostream>
#include <fstream>
#include <chrono>
#include <cmath>
#include <numeric>
#include <cassert>
#include <array>
#include <algorithm>

using real_t = float;
constexpr size_t NX = 512, NY = 512; //World Size
using grid_t = std::array<std::array<real_t, NX>, NY>;

class Sim_Configuration {
public:
    int iter = 10000;  // Number of iterations
    double dt = 0.05;       // Size of the integration time step
    real_t g = 9.80665;     // Gravitational acceleration
    real_t dx = 1;          // Integration step size in the horizontal direction
    real_t dy = 1;          // Integration step size in the vertical direction
    int data_period = 10000;// how often to save coordinate to file
    int numgangs = 1;
    std::string filename = "sw_output.data";   // name of the output file with history

    Sim_Configuration(std::vector <std::string> argument){
        for (long unsigned int i = 1; i<argument.size() ; i += 2){
            std::string arg = argument[i];
            if(arg=="-h"){ // Write help
                std::cout << "./par --iter <number of iterations> --dt <time step>"
                          << " --g <gravitational const> --dx <x grid size> --dy <y grid size>"
                          << "--fperiod <iterations between each save> --out <name of output file> --numgangs <Number of gangs>\n";
                exit(0);
            } else if (i == argument.size() - 1)
                throw std::invalid_argument("The last argument (" + arg +") must have a value");
            else if(arg=="--iter"){
                if ((iter = std::stoi(argument[i+1])) < 0)
                    throw std::invalid_argument("iter most be a positive integer (e.g. -iter 1000)");
            } else if(arg=="--dt"){
                if ((dt = std::stod(argument[i+1])) < 0)
                    throw std::invalid_argument("dt most be a positive real number (e.g. -dt 0.05)");
            } else if(arg=="--g"){
                g = std::stod(argument[i+1]);
            } else if(arg=="--dx"){
                if ((dx = std::stod(argument[i+1])) < 0)
                    throw std::invalid_argument("dx most be a positive real number (e.g. -dx 1)");
            } else if(arg=="--dy"){
                if ((dy = std::stod(argument[i+1])) < 0)
                    throw std::invalid_argument("dy most be a positive real number (e.g. -dy 1)");
```

```
            } else if(arg=="--fperiod"){
                if ((data_period = std::stoi(argument[i+1])) < 0)
                    throw std::invalid_argument("dy most be a positive integer (e.g. -fperiod 100)");
            } else if(arg=="--out"){
                filename = argument[i+1];
            } else if (arg=="--numgangs"){
                if ((numgangs = std::stoi(argument[i+1])) < 0)
                    throw std::invalid_argument("numgangs most be a positive integer (e.g. -numgangs 1000)");
            } else{
                std::cout << "---> error: the argument type is not recognized \n";
            }
        }
    }
};

/** Representation of a water world including ghost lines, which is a "1-cell padding" of rows and columns
 *  around the world. These ghost lines is a technique to implement periodic boundary conditions. */
class Water {
public:
    grid_t u{}; // The speed in the horizontal direction.
    grid_t v{}; // The speed in the vertical direction.
    grid_t e{}; // The water elevation.
    Water() {
        for (size_t i = 1; i < NY - 1; ++i)
        for (size_t j = 1; j < NX - 1; ++j) {
            real_t ii = 100.0 * (i - (NY - 2.0) / 2.0) / NY;
            real_t jj = 100.0 * (j - (NX - 2.0) / 2.0) / NX;
            e[i][j] = std::exp(-0.02 * (ii * ii + jj * jj));
        }
    }
};

void writeVectorVectorDoubleToBinary(const std::vector<grid_t>& data, const std::string& filename);


/* Write a history of the water heights to an ASCII file
 *
 * @param water_history  Vector of the all water worlds to write
 * @param filename       The output filename of the ASCII file
*/
void to_file(const std::vector<grid_t> &water_history, const std::string &filename){
    std::ofstream file(filename);
    file.write((const char*)(water_history.data()), sizeof(grid_t)*water_history.size());
}

/** One integration step
 *
 * @param w The water world to update.
 */
void integrate(Water &w, const real_t dt, const real_t dx, const real_t dy, const real_t g,const int numgangs) {

    #pragma acc parallel num_gangs(numgangs) present(w)
    {
        #pragma acc loop gang
        for (uint64_t j = 0; j < NX; ++j) {
```

```
105             w.e[0][j]       = w.e[NY-2][j];
106             w.e[NY-1][j]    = w.e[1][j];
107 
108             w.v[0][j]       = w.v[NY-2][j];
109             w.v[NY-1][j]    = w.v[1][j];
110 
111         }
112         #pragma acc loop gang
113         for (uint64_t j = 0; j < NY; ++j) {
114 
115             w.e[j][0] = w.e[j][NX-2];
116             w.e[j][NX-1] = w.e[j][1];
117 
118             w.u[j][0] = w.u[j][NX-2];
119             w.u[j][NX-1] = w.u[j][1];
120         }
121         // #pragma acc wait
122     }
123     #pragma acc parallel num_gangs(numgangs) present(w)
124     {
125         #pragma acc loop gang collapse(2)
126         for (uint64_t i = 0; i < NY - 1; ++i)
127         for (uint64_t j = 0; j < NX - 1; ++j) {
128             w.u[i][j] -= dt / dx * g * (w.e[i][j+1] - w.e[i][j]);
129             w.v[i][j] -= dt / dy * g * (w.e[i + 1][j] - w.e[i][j]);
130         }
131     }
132 
133     #pragma acc parallel num_gangs(numgangs) present(w)
134     {
135         #pragma acc loop gang collapse(2) // gives hints about the grid // using tile
136         for (uint64_t i = 1; i < NY - 1; ++i)
137         for (uint64_t j = 1; j < NX - 1; ++j) {
138             w.e[i][j] -= dt / dx * (w.u[i][j] - w.u[i][j-1])
139                     + dt / dy * (w.v[i][j] - w.v[i-1][j]);
140         }
141     }
142 }
143 
144 /** Simulation of shallow water
145  *
146  * @param num_of_iterations  The number of time steps to simulate
147  * @param size               The size of the water world excluding ghost lines
148  * @param output_filename    The filename of the written water world history (HDF5
    file)
149  */
150 void simulate(const Sim_Configuration config) {
151     Water water_world = Water();
152 
153     std::vector <grid_t> water_history;
154     auto begin = std::chrono::steady_clock::now();
155     #pragma acc data copy(water_world)
156     {
157         for (uint64_t t = 0; t < config.iter; ++t) {
158 
159             integrate(water_world, config.dt,  config.dx, config.dy, config.g, config
    .numgangs);
160 
161             if (t % config.data_period == 0) {
162                 #pragma acc update host(water_world.e)
163                 water_history.push_back(water_world.e);
164             }
```

```
        }
    }
    auto end = std::chrono::steady_clock::now();

    to_file(water_history, config.filename);
    // writeVectorVectorDoubleToBinary(water_history,config.filename);


    std::cout << "checksum: " << std::accumulate(water_world.e.front().begin(),
    water_world.e.back().end(), 0.0) << std::endl;
    std::cout << "elapsed time: " << (end - begin).count() / 1000000000.0 << " sec"
    << std::endl;
}

/** Main function that parses the command line and start the simulation */
int main(int argc, char **argv) {
    auto config = Sim_Configuration({argv, argv+argc});
    simulate(config);
    return 0;
}


void writeVectorVectorDoubleToBinary(const std::vector<grid_t>& data, const std::
    string& filename) {
    std::ofstream outFile(filename, std::ios::binary);
    if (!outFile.is_open()) {
        std::cerr << "Failed to open file for writing: " << filename << std::endl;
        return;
    }

    // Write the number of rows
    size_t numRows = data.size();
    outFile.write(reinterpret_cast<const char*>(&numRows), sizeof(numRows));

    for (const auto& row : data) {
        // Write the number of elements in each row
        size_t numElements = row.size();
        outFile.write(reinterpret_cast<const char*>(&numElements), sizeof(numElements
    ));

        // Write the elements of the row
        outFile.write(reinterpret_cast<const char*>(row.data()), numElements * sizeof
    (double));
    }

    outFile.close();
}
```

Complete code.