Accelerating Numerical Simulation of Multiple Scattering for Radiation Transport

Požgaj, Stjepan

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Stjepan Požgaj

ACCELERATING NUMERICAL SIMULATION OF MULTIPLE SCATTERING FOR RADIATION TRANSPORT

Diplomski rad

Voditelji rada:
prof. dr. sc. Zlatko Drmač
dr. sc. Sabine Grießbach

Zagreb, srpanj 2022.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

I sincerely thank Dr. Sabine Grießbach, Dr. Paul F. Baumeister and Dr. Lars Hoffman for giving me the opportunity to write this thesis in collaboration with them. Furthermore, I am grateful to prof. dr. sc. Zlatko Drmač for helping me get invited to the summer program where this journey began.

Zahvaljujem i svojoj obitelji, prijateljima i zaručnici Ani na podršci tijekom studiranja.

Contents

Introduction

Infrared measurements from polar orbiting satellite instruments are an important pillar of Earth observation systems. In order to derive the state of the atmosphere (temperature, pressure, trace gas concentrations, aerosol and cloud properties) from the measured spectra, radiative transfer through the atmosphere along a ray path given by the position and orientation of the satellite instrument has to be modeled. The atmosphere variables can be varied until the calculated spectra match with the measured spectra. This process is called retrieval and relies on a fast execution of the radiative transport computation.

The Juelich Rapid Spectral Simulation Code (JURASSIC) [18, 19, 20] is a fast radiative transfer model for the analysis of atmospheric remote sensing measurements in the mid-infrared spectral region. It performs the radiative transport forward calculation and provides the retrieval algorithm around it. It was developed by Hoffman [20]. JURASSIC was originally written in `C` and has been ported to GPUs using the CUDA programming language by Baumeister et al. [4, 5, 6]. An important research field is to incorporate particles (ice and water clouds, volcanic aerosol, dust particles, etc.) into the model. For this, scattering of the infrared radiation on the liquid or solid particles needs to be accounted for, so JURASSIC was cloned and extended to JURASSIC-scatter by Grießbach [14, 15]. JURASSIC without scattering is available as vectorized CPU and GPU version and as reference implementation, however, JURASSIC-scatter so far did not benefit from tuning and acceleration.

The main goal of this thesis was to accelerate JURASSIC-scatter by using JURASSIC-GPU in some of its parts to achieve better performance. Benchmarking was done on one of the world's top supercomputers. After that, it was decided to develop a separate project in which there would be no code duplicates and which could be used as a library for the JURASSIC reference implementation. This new unified code version of JURASSIC is called JURASSIC-unified.

The thesis is structured as follows: Chapter 1 introduces parallel programming models which were used in JURASSIC-unified implementation. In Chapter 2 atmospheric remote sensing and the JURASSIC models are presented. Chapter 3 shows how JURASSIC-scatter was accelerated and in Chapter 4 the JURASSIC-unified code is presented.

Chapter 1

Parallel computing

In this chapter a brief history of supercomputing as well as some modern supercomputers are presented. After that, the basics of the programming models used in the implementation of the JURASSIC forward model are covered.

1.1 Supercomputers

A supercomputer is a name for a particularly powerful computer. Supercomputers can be designed to solve a particular real-world computational problem or for general purpose computing. Supercomputers often have to deal with computation-heavy simulation problems that common computers cannot handle. These problems come from industry and various fields of science such as physics, astrophysics, astronomy, atmospheric science, materials science, genomics, bioinformatics and many others. Supercomputers also play an important role in artificial intelligence and machine learning, dominant and fast-growing areas that will require even better computing performance in the future.

The standard way to express the processing power of a supercomputer is by its FLOPS, which is an acronym for floating point operations per second. While clock speed [Hz] is a more popular speed measure for PCs, FLOPS is the metric for supercomputers because it covers multi-core architectures and vectorization. Of course, by increasing the clock speed of some processor, its FLOPS will be increased, but that does not mean that a computer with a higher clock speed also has to have more FLOPS. This is because the FLOPS performance is also highly dependent on the computer architecture.

The TOP500 list [3] shows the most powerful computer systems. This ranking started in 1993 and the list is updated twice a year. The LINPACK benchmark is used to compare supercomputers. In this benchmark, the time required to solve a dense system of linear equation using lower-upper (LU) factorization with partial pivoting is measured. Figure

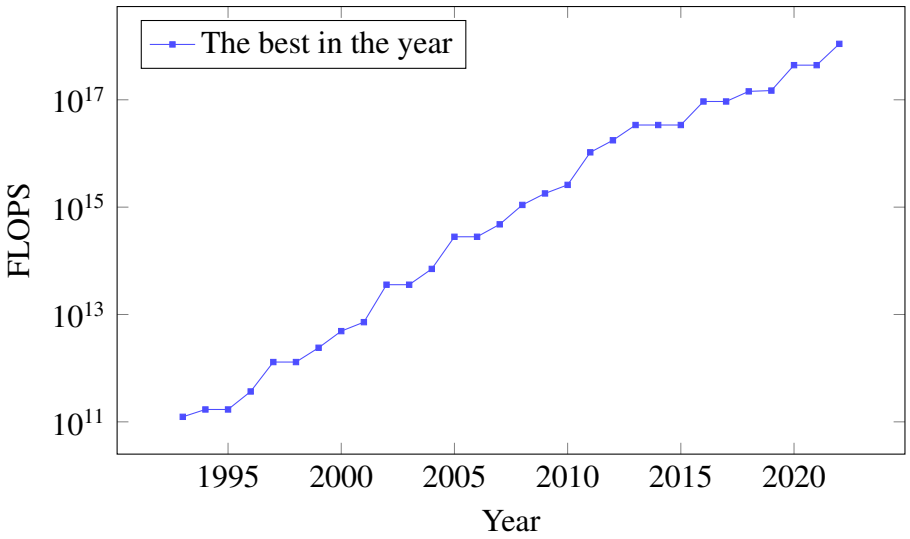


Figure 1.1: Performance of the year's top supercomputer on the LINPACK benchmark over the last three decades. Created based on the data from [3].

1.1 shows the rapid growth of the top supercomputer performance during the last three decades.

The CDC 6600 is considered the first supercomputer [8]. The CDC 6600 was designed in 1964 by Seymour Cray, who is called "the father of supercomputing", while he was working at a relatively small company named Control Data Corporation. It was ten times faster than the competing IBM computer and also a few times smaller, which makes it even more incredible. Its speed was 11 megaFLOPS, which is a few orders of magnitude less than today's mobile phones. In 1972, the ILLIAC IV was released. Only one model was built because the cost of production turned out to be 4 times higher than predicted. Production was delayed for a year and when it was completed some other supercomputers already had better performance. Nevertheless, ILLIAC IV is very important for the history of supercomputers because it was the first computer with a parallel architecture, which means that it had multiple processors working together. After developing the CDC 6600, Seymour Cray left CDC and founded his own company Cray where in 1976 the Cray-1 supercomputer was released. This single processor supercomputer was the first to successfully implement the vector processor design in which instruction operates on multiple data elements, rather than single data values. The last world-class supercomputer to use the classical vector processing design approach was the Earth Simulator developed in 2002. Its speed was 35.86 teraFLOPS and it was the fastest supercomputer in the world until 2004, when it was replaced by the IBM Blue Gene, which could perform 500 teraFLOPS. The Blue Gene/L had 131,000 processors, which were small enough to fit 32 dual processors in

a single microchip. Therefore, its size was significantly smaller than that of its competitors.

More than 80% of all supercomputers from the current TOP500 list are based on computer *clusters* [26]. A cluster is a set of independent computer systems integrated in a interconnection communication network. Developed in 2012, the Titan was the first supercomputer based on graphics processing units (GPUs) in addition to conventional central processing units (CPUs) to perform over 10 petaFLOPS. The combination of CPUs and GPUs for computing is called *heterogeneous computing*, and it has dominated the TOP500 list in recent years, because of the absolute performance and better performance per watt, which makes it more energy efficient. The fastest supercomputer at the moment is called Frontier [1]. It is the first supercomputer whose score on the LINPACK benchmark exceeded one exaFLOPS (10^{18} FLOPS).

All performance results reported in this thesis were obtained on the Jülich Wizzard for European Leadership Science (JUWELS) supercomputing system at the Jülich Supercomputing Centre, which consists of two partitions: Cluster and Booster [27]. The JUWELS Cluster nodes comprise a Dual Intel Xeon Platinum 8168 CPU with 2×24 cores. The JUWELS Booster nodes comprise a Dual AMD EPYC Rome 7402 CPU with 2×24 cores and four NVIDIA A100 GPUs. JUWELS Booster's score on the LINPACK benchmark is 44.12 petaFLOPS, which puts it in 11th place on the TOP500 list as of 1 June 2022, and makes it one of the three fastest European supercomputers [3].

1.2 Parallel programming models

The JURASSIC model is implemented as a hybrid of three parallel programming models: MPI, OpenMP and CUDA. The concept of these programming models needs to be understood first before speeding up and restructuring the code is explained. This is not meant to be a tutorial, but a brief overview of the models with simple examples.

MPI

The Message Passing Interface (MPI) [12] is a specification of message passing operations. This is the dominating programming interface for distributed memory programming. It is often used for computing clusters because it offers a set of application programming interface (API) functions for communication between processes. The communication is achieved by moving data from the address space of one process to that of another. MPI is implemented as a library with language bindings for Fortran and `C`.

Listing 1.1 shows how MPI is used to run JURASSIC forward simulations of multiple tests simultaneously and to determine the time required to perform the entire calculation. In the `C` programming language, to use MPI, one has to include the `mpi.h` header file and initialize the MPI library before MPI functions can be used. Also, after using MPI functions,

the MPI library has to be finalized. MPICH [2] is a high-performance implementation of MPI. The `mpicc` wrapper compiler is used to compile and link MPI programs written in `C`. To run an executable on multiple processes with a specified number of processes `mpiexec` has to be used. When an MPI program runs on multiple processes we are talking about the *single program, multiple data* (SPMD) programming technique, which is the most common style of parallel programming, but `mpiexec` also allows to run different executables on multiple processes which are able to communicate with each other (*multiple programs, multiple data*), which is for example exploited in modular supercomputing.

Listing 1.1: Simple MPI program. Shows how MPI is used in the implementation of the JURASSIC forward model.

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[]) {
  int myrank, numprocs;
  double start_time, end_time, duration;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  start_time = MPI_Wtime();
  /*************************************************
     JURASSIC forward model calculations for test
     myrank, myrank+numprocs, myrank+2*numprocs,...
  *************************************************/
  end_time = MPI_Wtime();
  duration = end - start;
  MPI_Reduce(&duration, &global, 1, MPI_DOUBLE,
             MPI_MAX, 0, MPI_COMM_WORLD);
    printf("Global runtime is %lf seconds\n", global);
  MPI_Finalize();
}
```

`MPI_COMM_WORLD` is an example of the simplest communicator that includes all available processes in communication. It is practical to use it when one process wants to send a message to all the others or all processes want to send a message to one of them. It is sufficient for the JURASSIC implementation, but when the number of MPI processes becomes quite large, more advanced communicators that group processes in a more efficient way should be used. As mentioned above, the same MPI program is run on different processes, and to prevent all processes to perform the same instruction flow, they must be

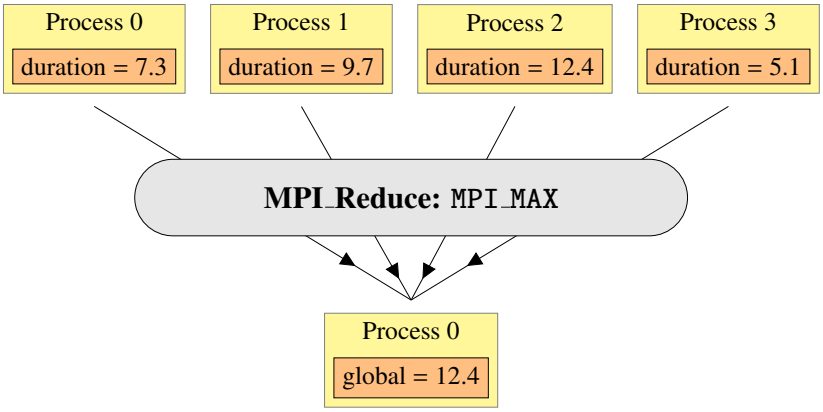


Figure 1.2: The `MPI_Reduce` procedure is used in JURASSIC for calculating the maximum execution time among the processes.

distinguished. Processes are identified using MPI ranks. Each process finds out its rank after calling the `MPI_Comm_rank` function. Also, to find out the total number of processes in a communicator, it is necessary to call the `MPI_Comm_size` function.

In Listing 1.1, the `MPI_Reduce` procedure is used to determine the time required to perform the entire calculation. Figure 1.2 demonstrates this idea. This is an example of an *all-to-one* collective procedure, because each process sends information about the time spent on the calculation to the process with rank zero, and the process with rank zero performs a MAX operation on the values, including its own. A procedure is *collective* when all processes need to invoke it. In addition to *all-to-one* (`MPI_Reduce`, `MPI_Gather`), there are also *one-to-all* (`MPI_Bcast`, `MPI_Scatter`) and *all-to-all* (`MPI_Allgather`, `MPI_Alltotall`, `MPI_Allreduce`) collective communication procedures.

In addition to collective communication in which all the processes participate, which are in the communicator, communication in MPI can also be direct. This type of communication is called *point-to-point* communication. In *point-to-point* communication, a sender must specify the rank of the process to which it wants to send the message and send it using the `MPI_Send` procedure. Also, in order to receive a message, the recipient must invoke the `MPI_Recv` procedure in which it must specify the rank of the process from which it expects the message.

`MPI_Reduce`, `MPI_Send` and `MPI_Recv` are examples of *blocking* procedures. In the `MPI_Reduce` case a process can continue with the execution of the program only when the procedure is executed, that is when all messages processes have sent their messages and the process to which it was sent has received the result. Unlike such procedures, there are also *non-blocking* procedures in which the execution of the program continues, but the user is not allowed to reuse resources specified by the call to the procedure before the communication has been completed using an appropriate completion procedure. This al-

lows communication patterns that would lead to a deadlock if programmed using *blocking variants* of the same operations. *Non-blocking* communication operations are split into *start* and *completion*. For example, the `MPI_Isend` and `MPI_Irecv` ("I" is for immediate) *point-to-point* routines produce a *request handle* that represents the in-flight operation and is consumed in the *completion* routine. In the *completion* part, the user has to use the `MPI_Test` procedure to find out whether the operation associated to some *request handle* is completed. *Blocking* send can be paired with a *non-blocking* receive procedure and vice versa. Also, a *non-blocking* operation immediately followed by the matching wait is equivalent to the *blocking* operation.

OpenMP

OpenMP [10] is an application programming interface and a programming language extension via directives for parallel computing on shared-memory platforms. Unlike MPI, which takes care of communication between different processes, in the case of OpenMP, parallelization is done using threads within the same process. The main difference between the two approaches is that threads do not have to communicate by sending messages because they have access to a shared memory space. With the shared memory available to all threads, each thread has its own stack and associated static memory. Threads are executed on different hardware threads. Therefore, the expected speedup when using OpenMP is limited by the number of cores, but a programmer must be aware that this is not the only limiting factor. For example, if the main memory bandwidth is a shared resource, the speedup is also limited by the number of memory channels in a server-class chip.

An OpenMP program is a program written in a base language (`C`, `C++` or Fortran) annotated with OpenMP directives or that calls OpenMP API runtime routines. In the `C` programming language, the `#pragma omp directive-name` is the way to write an OpenMP directive. It has to be placed before a structured block to which it is applied. An OpenMP program starts as one single-threaded process. `#pragma omp parallel` creates a team of threads to execute the code contained in the following structured block. Similar to the number of processes in a communicator and the ranks in the MPI case, here the functions `omp_get_num_threads` and `omp_get_thread_num` are used to determine the number of threads executing code and to identify threads to which consecutive numbers starting from zero are assigned, respectively. The number of threads to fork is not fixed, but can be controlled by the programmer. It is also worth mentioning that it is possible to allow OpenMP to fork a thread that is already part of a group of threads. If OpenMP nested parallelism is active, the number of threads at each *nested level* can be set by calling the `omp_set_num_threads` function. [25] Figure 1.3 illustrates an example of nested parallelism.

Listing 1.2 shows three ways to multiply two arrays. In the `mul_array_serial` function arrays a and b of length n are multiplied in serial using a simple for loop. In the

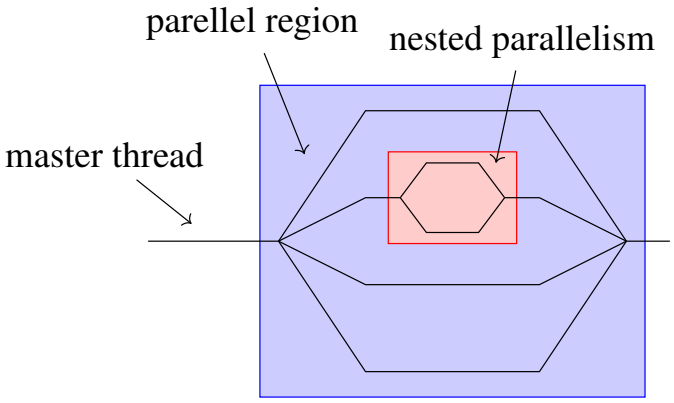


Figure 1.3: Nested parallelism with four threads at the first and two threads at the second level.

second function in Listing 1.2 the master thread is forked into a team of threads, which separately multiply arrays for a different sets of indices. If the OpenMP option is not activated when compiling a program, an OpenMP directive will be ignored so the following block will be executed by just one thread. Missing OpenMP functions cannot be ignored, so the case when the OpenMP library is not included is treated separately using an `#ifdef C` preprocessor macro. The `mul_array_parallel_for` function shows the simplest way to multiply two arrays. The idea is similar to the one in the previous function because the `#pragma omp for` directive divides the loop iterations between the spawned threads.

Listing 1.2: Array multiplication in serial and parallel with OpenMP.

```
void mul_array_serial(double a[], double const b[], int n) {
  for (int i = 0; i < n; i++) {
    a[i] *= b[i];
  }
}

void mul_array_parallel(double a[], double const b[], int n) {
  #pragma omp parallel default(none) shared(a, b, n)
  {
    #ifdef _OPENMP
    int thread_id = omp_get_thread_num();
    int num_of_threads = omp_get_num_threads();
    #else
    int thread_id = 0;
    int num_of_threads = 1;
    #endif
```

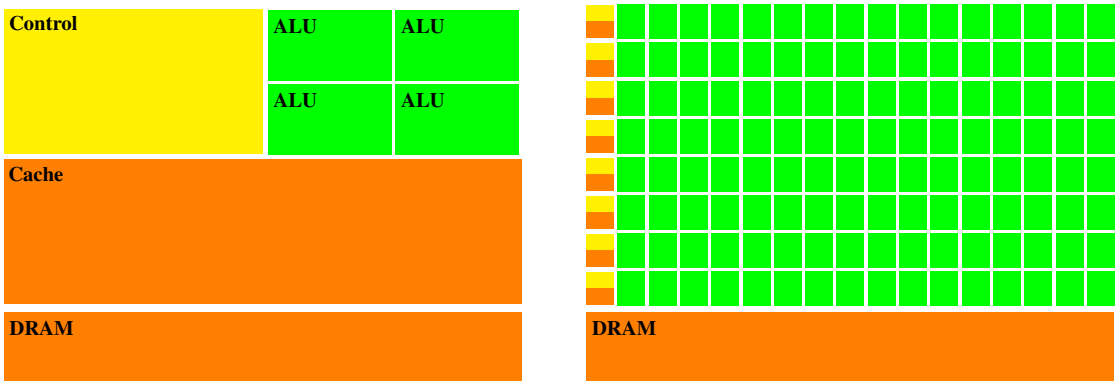
```
    for(int i = thread_id; i < n; i += num_of_threads)
      a[i] *= b[i];
  }
}

void mul_array_parallel_for(double a[], double const b[], int n) {
  #pragma parallel for
  for (int i = 0; i < n; i++) {
    a[i] *= b[i];
  }
}
```

Since communication between threads takes place by writing and reading to shared memory, problems can arise if multiple threads write to the same memory unit at the same time or if at least one thread tries to write and at least one tries to read from the same memory unit. In that case, we say that the *data race* happened. To avoid it, explicit thread synchronization has to be done. One way to do it is to use *barriers*. Threads are only allowed to continue the execution of code after the `#pragma omp barrier` when all threads in the current team have reached it. A programmer can also use the `#pragma omp critical` directive, which allows only one thread to execute the code in a *critical section* that follows the directive.

CUDA

Compute Unified Device Architecture (CUDA) [23] is a parallel computing platform that allows developers to use the computing power of graphics processing units (GPUs) for general-purpose processing. It is developed by NVIDIA and specially designed for their GPUs. In Figure 1.4 the CPU and GPU processor architectures are compared. A central processing unit has fewer cores with a lot of cache, while a graphics processing unit has more cores but each of them has less cache memory capacity. Because of this, the main characteristic of a CPU is low latency which makes it great for serial programming. While on the other hand, high throughput is the reason why GPUs are so powerful in parallel processing. The difference between these two performance measures is that latency i.e. the absolute time required to complete a task, while throughput is the number of tasks that can be executed per unit time. GPU processors are designed so that a larger proportion of transistors are intended for processing the data than for data caching and flow control. Because of that the maximum efficiency on a GPU is achieved if all instructions are of the same type. Fortunately, this condition is often possible when doing computer graphic computations where a single instruction has to be applied to a large amount of data. Similar patterns are found in applications for scientific simulation.



(a) CPU (b) GPU

Figure 1.4: Simplified comparison of multicore CPU and manycore GPU architectures based on Figure 1 from [9]. In GPUs, registers take the role of the L1 cache, so the green arithmetic logic unit (ALU) area contains some of what is labelled ”Cache” on the CPU side. DRAM is an abbreviation of ”dynamic random access memory”.

A CUDA processor contains a set of *Streaming Multiprocessors* (SM). Depending on the model, the number of SMs can be up to 130, but is usually around 30. Each streaming multiprocessor has up to 32 *scalar processors* (SP) and shared memory. All SMs have access to the GPUs device memory. There are four types of multiprocessor memory:

- local 32-bit registers
- shared memory, which is shared among the SPs of the same SM - very important for a program to perform efficiently
- constant cache – again shared by SPs, but read-only, only CPU can write
- texture cache – also read-only

All four types of multiprocessor memory are equally fast and recommended to be used over the slower device memory.

Flynn’s classification [11] is the basic division of computer architectures. According to it, computer architectures are divided into four types, depending on the number of instruction streams and memory streams:

- SISD – Single instruction stream, single data stream
- SIMD – Single instruction stream, multiple data streams

- MISD – Multiple instruction streams, single data stream
- MIMD – Multiple instruction streams, multiple data streams

In addition to this simple classification, there are other types of architectures such as:

- SIMT - Single instruction, multiple threads
- SPMD - Single program, multiple data streams

The style of execution in CUDA corresponds to a SIMT architecture. One instruction controls the behaviour of multiple processing elements. Programmers can divide the GPU threads among a multiprocessor using *blocks* that are executed independently of each other. The dispatcher inside a multiprocessor has to distribute threads to scalar processors. Each thread is given to one scalar processor on which it independently executes with its own set of registers and instruction addresses. When an SIMT unit gets one or more blocks that a multiprocessor has to process, it first divides the blocks into groups of 32 threads, which are called *warps*. Inside a warp identical instructions should ideally be performed. If threads inside a warp branch to different execution paths, for example, because of different evaluations of an *if statement*, the time required for the execution is significantly increased. This is called *warp divergence*. To avoid it, the programmer has to distribute threads to blocks in a way that all threads in a block perform identical instructions.

The basic task of the CUDA programming model is to enable efficient and transparent communication with parallel hardware. This is achieved by a minimal upgrade of the `C` programming language – CUDA `C`. Similarly, CUDA `C++` is an extension of the `C++` programming language. It is done by allowing the programmer to define functions, called *kernels*, that, when called, are executed in parallel by different CUDA threads. CUDA distinguishes three types of kernels according to where they are performed and who has the right to call them:

- `__host__` – standard CPU function which can be called only from the CPU
- `__global__` – executes on the GPU, but can be called only from the CPU
- `__device__` – executes on and can be called only from the GPU

CUDA kernel functions, which execute on a GPU (`__global__` and `__device__`) differ from standard CPU functions because:

- they must not be recursive
- static variables cannot be declared inside the function body

- they cannot call upper-level functions which are not adapted to it, such as *qsort*
- `__global__` kernels must be of void type

One CPU can control more than one GPU, but one CPU thread cannot control multiple GPUs simultaneously. For that multiple CPU threads are needed. Before executing a CUDA kernel, GPU and CPU memory must be declared and allocated. After that, initialized host data should be transferred from host to device. Similarly, after kernel execution, the result should be transferred from the device to the host. Modern NVIDIA GPUs also support the *unified* memory model in which memory transfers are hidden from the user so `managed` memory can be used by both host and device code without explicitly coded memory transfers.

It was already mentioned that CUDA threads are divided into blocks to distribute them among streaming multiprocessors. Threads are organized into one-dimensional, two-dimensional, or three-dimensional blocks. Just like threads, blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks. In Figure 1.5 you can see one-dimensional blocks of threads inside a one-dimensional grid of blocks. It corresponds to the kernel `mul_array_kernel`, which is launched in the `mul_array` function presented in Listing 1.3. Blocks and threads inside a kernel can be identified using inbuilt variables `blockIdx.x` and `threadIdx.x`, respectively, which is analogous to the result of `omp_get_thread_num()` in OpenMP. Since the number of elements in the array can be greater than the total number of threads, the number of blocks `gridDim.x`, the number of threads per block `blockDim.x` and a *grid stride loop* [16] are used. The arguments *grid_size* and *block_size* of the function `mul_array` from Listing 1.3 are the number of blocks and the number of CUDA threads in each block. A kernel launch is valid if $grid\_size \geq 1$ and $1 \leq block\_size \leq 1024$. Tuning of these parameters can be done to achieve the best possible performance.

Listing 1.3: Array multiplication with CUDA.

```
__global__ mul_array_kernel(double a[], double const b[], int n) {
    for(int i = blockIdx.x*blockDim.x + threadIdx.x; i < n;
           i += blockDim.x*gridDim.x) // grid stride loop
        a[i] *= b[i];
}

__host__ mul_array(double a[], double const b[], int n,
                   int grid_size, int block_size) {
    mul_array_kernel<<<grid_size, block_size>>> (a, b, n);
}
```

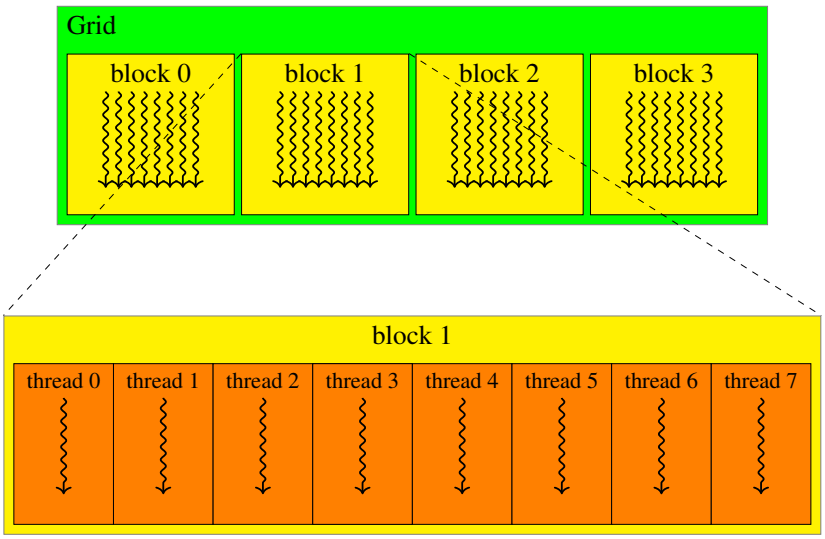
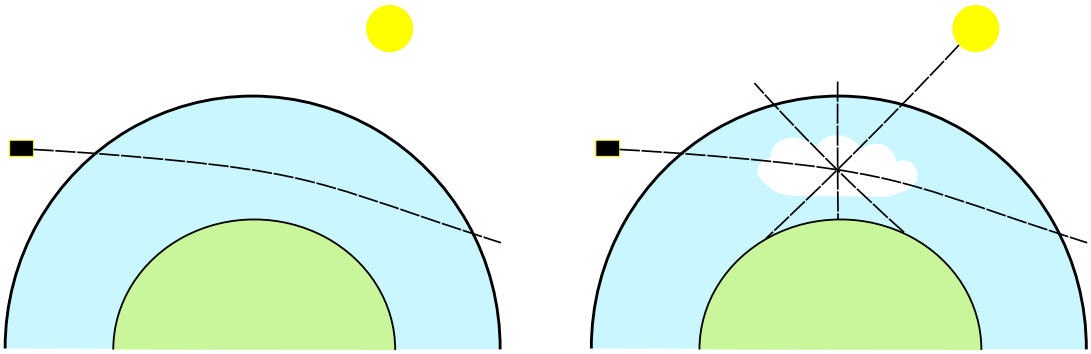


Figure 1.5: Schematic representation of CUDA's hierarchical threading model with a one-dimensional grid of one-dimensional blocks. Adapted from Figure 4 from [9].

Chapter 2

Simulation of radiation transport

JURASSIC is a coupled forward and retrieval fast radiative transfer model, which allows analyses of different remote sensing measurements. In this chapter the idea of atmospheric remote sensing is presented. After that, the JURASSIC forward models and their implementations as well as inverse modeling in JURASSIC are explained.



(a) Limb path through a clear atmosphere. (b) Limb path through a cloudy atmosphere.

Figure 2.1: The clear air case can be treated using JURASSIC or JURASSIC-GPU. In the cloudy case incoming radiance from all directions is scattered towards the detector by cloud particles and radiation along the line of sight is scattered out of the line of sight. The scattering process is treated by JURASSIC-scatter. The figures are based on Figure 1 from [15].

2.1 Atmospheric remote sensing

By measuring we want to determine the value of a quantity. An example of this quantity is height or width, as well as temperature and air pressure. A key feature of remote sensing theory is not that measurements are made from a distance, but that they are indirect. Such indirect measurements are used to determine the state of the atmosphere, but also in other areas because they are often simpler and cheaper than direct measurements. They are often used in medicine because such measurements are usually not physically invasive.

There are only a few instruments that directly measure the quantity of interest. Instead, indirect or remote measurement of the effect caused by the measurand is used. Retrieval methods are then used to determine the value of the desired quantity from the measurements. The retrieval problem consists of two separable key parts: the forward model and the inverse problem. The task of the forward model is to simulate the measurement of the effect measured by a measuring instrument for some hypothetical target quantity. In the case of determining the state of the atmosphere, the forward model must, for given target quantities such as temperature, pressure and gas concentrations, calculate the radiation that the measuring instrument on the satellite would measure. Figure 2.1a illustrates a satellite measuring the radiance in limb geometry[1].

To be more precise, terminology from [24] will be introduced. The quantities that are actually measured can be represented by the m-dimensional measurement vector $\mathbf{y}$, where m is the number of measurements. In the same way, n target quantities can be represented with the n-dimensional state vector $\mathbf{x}$. For each state vector there is a corresponding ideal measurement vector $\mathbf{y}_I$, for which states

$$\mathbf{y}_I = \mathbf{f}(\mathbf{x}), \tag{2.1}$$

where $\mathbf{f}$ formally describes the physics of the measurement. However, in addition to the fact that it is impossible to avoid experimental errors, in practice it is often necessary to approximate detailed physics with some forward model $\mathbf{F}(\mathbf{x})$ in order to make the model simpler and faster to perform. For the forward model $\mathbf{F}(\mathbf{x})$, the connection between the state vector and the measurement can be expressed with the following formula:

$$\mathbf{y} = \mathbf{F}(\mathbf{x}) + \epsilon, \tag{2.2}$$

where ϵ is the measurement error vector. The JURASSIC forward models for both clear air and cloudy scenarios is presented in Section 2.2. In inverse modeling the goal is to find the best state vector $\mathbf{x}'$ for a given forward model $\mathbf{F}(\mathbf{x})$ and measurement vector $\mathbf{y}$. In Section 2.3 the inverse modeling in JURASSIC is explained.

[1] In limb viewing geometry, the remote sensing instrument looks towards the limb (horizon) of the atmosphere, so the line of sight is pointed nearly tangentially to the Earth's surface.

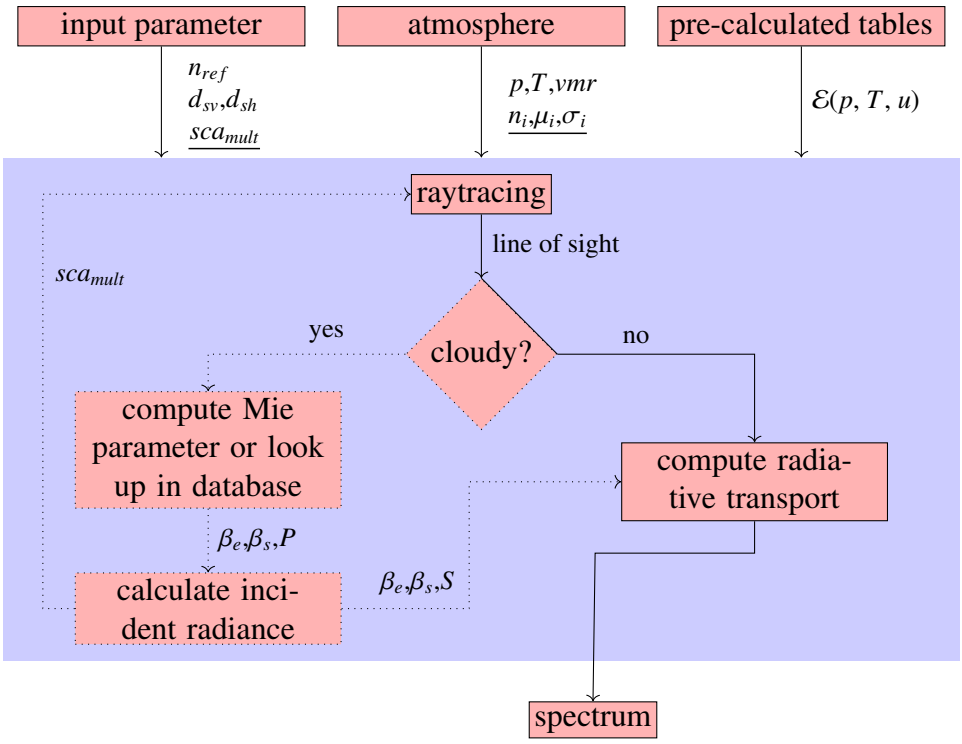


Figure 2.2: Forward models. Dotted arrows and underlined variables indicate the difference between the JURASSIC and the JURASSIC-scatter model. The figure is based on Figure 2 from [15].

2.2 JURASSIC forward models

JURASSIC forward model

The JURASSIC forward model is a radiative transfer model for the mid-infrared spectral region. It is shown in solid rectangles of Figure 2.2. In the input block, the pre-calculated emissivity tables, the atmospheric data containing pressure, temperature, volume mixing ratios of atmospheric gases, and the control file are given. Pre-calculated emissivity look-up tables of spectrally averaged emissivities have been prepared for JURASSIC by means of line-by-line calculations [19]. For generating the emissivity look-up tables, any radiative transfer model which allows the calculation of the transmission of a homogeneous gas cell depending on pressure, temperature and emitter column density can be used. The control file contains a list of emitters and radiation channels to be considered, as well as parameters such as, for example, the step length in the raytracing, which is one of the most important parameter for the accuracy and performance of JURASSIC.

At the beginning of a radiative transfer calculation the atmospheric state has to be

defined. The field quantities such as pressure, temperature and volume mixing ratios are given for some vertical profiles, and the values for the heights between them are determined using linear interpolation.

After that, the path of a single ray through the atmosphere, which is referred to as pencil beam, has to be calculated. In the case of a limb geometry the pencil beam is not just a straight line tangentially through the atmosphere but a curve refracted towards the Earth's surface. The positions along a single ray path are calculated using the Eikonal equation [7],

$$\frac{d}{ds}\left(n\frac{d\mathbf{r}}{ds}\right) = \nabla n, \tag{2.3}$$

where s is the spatial coordinate along the ray $\mathbf{r}(s)$ with the origin $s = 0$ being located at the position of the observing instrument. The degree of curvature depends on the change of the refractive index n of the atmosphere with altitude and the refractive index, in return, depends on atmospheric conditions. Equation 2.3 is solved numerically using an iterative scheme described by Hase and Hopfner [17]. The step length chosen for determining segments in raytracing has to fulfil two constraints: on the one hand fewer steps are advantageous when considering the computation time and on the other hand the step length must be short enough so that the atmosphere properties along one step can be assumed as constant. An example of a limb path through an atmosphere is given in Figure 2.1a.

After raytracing, the evaluation of the radiative transfer is done by calculating spectrally averaged radiances, emissivities and Planck's functions applying pre-calculated emissivity tables according to the instrument's characteristics. Segment emissivities are computed according to the emissivity growth approximation (EGA) method. In Figure 2.2 we refer to those three calculations as "computing radiative transport". In the case of clear air conditions in the infrared and in case of local thermodynamical equilibrium, scattering can be neglected and the radiative transfer along an arbitrary ray path can be described by the Schwarzschild equation,

$$dI(v, s) = \beta_a(v, s)[B(v, s) - I(v, s)]ds, \tag{2.4}$$

where I is the radiance, v the wavenumber, s is again the spatial coordinate along the ray path, β_a the absorption coefficient of the trace gases and particles, and $B(v, s)$ is Planck's function.

This original version of JURASSIC [18] was written in `C` by Lars Hoffman [20] and features an MPI/OpenMP hybrid parallelization for efficient use on supercomputers.

JURASSIC-GPU forward model

JURASSIC has been ported to GPUs using CUDA by Baumeister et al [4, 5, 6]. In addition to adding some GPU parameters that were not required in the reference version, the internal

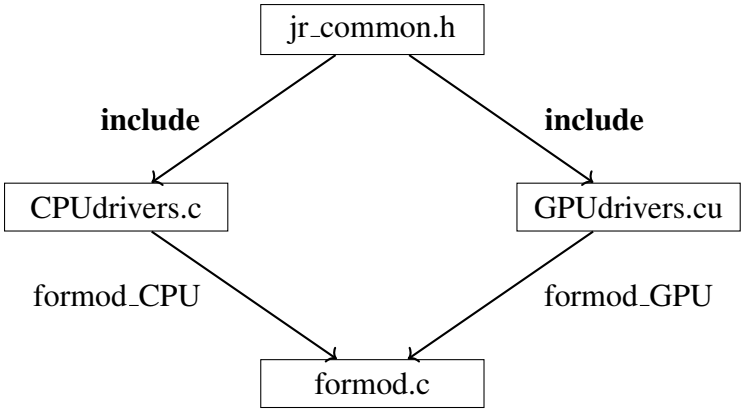


Figure 2.3: Common source code. The figure is based on Figure 4 from [4].

shape of some data structures was changed to achieve better performance, when porting that version to the GPU. For example, the structure in which the line of sight obtained by raytracing is stored and the struct in charge of emissivity look-up tables were changed. These differences are the main obstacle for connecting and merging JURASSIC-GPU and JURASSIC-scatter in a straightforward fashion.

The single source code policy of JURASSIC-GPU facilitates connecting the projects, because it makes the code easier to understand and maintain. The idea of separating functions in the files, shown in Figure 2.3, is that all functions that will run on both the CPU and the GPU are kept in a separate *jr_common.h* file and are included in architecture-specific implementations *CPUdrivers.c* and *GPUdrivers.cu*. At Listing 2.1 you can see how calculating Planck's function was done using this policy.

Listing 2.1: Shared code base for GPU and CPU implementation.

```
// jr_common.h:
    inline double __host__ __device__
       Planck(double nu, double T) {
        return nu*nu*nu/(exp(nu/T) - 1);
       }

// CPU_drivers.c including jr_common.h:
#pragma omp for
    for(int i = 0; i < N; i++)
        s[i] = Planck(nu[i], temperature);

// GPU_drivers.cu including jr_common.h:
    for(int i = blockIdx.x*blockDim.x + threadId.x; i < N;
           i += blockDim.x*gridDim.x) // grid stride loop
```

```
s[i] = Planck(nu[i], temperature);
```

JURASSIC-scatter forward model

The implementation of scattering into JURASSIC is schematically shown in Figure 2.2, but in this case the dotted rectangles and the diamond also have to be included. Compared to the JURASSIC model without scattering, JURASSIC-scatter contains some new input parameters: scattering order, number of scattering modules, and log-normal parameters of the particle size distribution (particle number concentration, median radius and standard deviation). The scattering order sca_{mult} defines the number of scattering levels in the forward model (see Section 3.2). The other three new input parameters are not part of the JURASSIC forward model without scattering because those are aerosol and cloud properties.

In the JURASSIC-scatter forward model, the raytracing of the line of sight is again calculated first. But in this case, segments located in the cloud are determined and for each of these segments the extinction and scattering coefficients as well as the phase function are determined and new ray paths ending at that segment are set up to determine the incoming radiance from all directions. The incoming ray paths also pass through the cloud and undergo scattering processes. This process can be, from the computer science point of view, seen as a recursion with the two base cases. The first base case is when a segment is not part of the cloud, so no rays have to be generated from it. The second base case is when the recursion depth reaches the scattering order sca_{mult}. Based on this approach single or multiple scattering, depending on scattering order, can be computed. An example of a limb path through a cloudy atmosphere is shown in Figure 2.1b.

Similar to the equation 2.3 in the clear air case, the modeling radiative transfer with presence of clouds or aerosols corresponds to solving the following equation:

$$dI(\nu, s) = [-\beta_e(\nu, s)I(\nu, s) + \beta_a(\nu, s)B(\nu, s) + \beta_{sp}(\nu, s)S(\nu, s)]ds, \tag{2.5}$$

where β_e is the extinction coefficient, β_a is the absorption coefficient and β_{sp} is the particles scattering coefficient. The scattering source term

$$S(\nu, s) = \frac{1}{4\pi} \int_0^{4\pi} P(\nu, \omega', \omega, s)I(\nu, \omega', s)d\omega' \tag{2.6}$$

contains incident radiance from all directions ω' scattered into direction ω of the line of sight and weighted with the phase function $P(\nu, \omega', \omega, s)$.

JURASSIC-scatter [14], which has been developed by Griessbach at al. [15] is, as an upgrade of JURASSIC, also written in C and also features an MPI/OpenMP hybrid parallelization. We see that it is equivalent to JURASSIC without scattering in its base

cases. This is precisely the motivation for merging the JURASSIC and JURASSIC-GPU models to create an accelerated JURASSIC-scatter-GPU version, which was the main task in this project.

2.3 Inverse modeling in JURASSIC

As mentioned above, in inverse modeling the goal is to find the best state vector $\mathbf{x}'$ for a given forward model $\mathbf{F}(\mathbf{x})$ and measurement vector $\mathbf{y}$ [21]. For some measurements there are usually infinitely many possible state vectors, which, applied with the forward model, return a vector that is close to the measurement vector. These potential state vectors can differ considerably from each other. In order to decide on the best possible estimate of the quantity of interest, we are allowed to use knowledge about that quantity that is independent of the obtained measurements.

Optimal estimation

The idea is to use *a priori* knowledge of the quantity of interest. In the case of a retrieval of an atmospheric state from observed radiances, this knowledge may be previous measurements or the average value of the quantity in the past. Let $P(\mathbf{x})$ be the probability distribution function (pdf) of the state vector $\mathbf{x}$ before the measurements are made, that is, the *a priori* pdf. Also, let $P(\mathbf{x}|\mathbf{y})$ be the *a posteriori* pdf of state vector $\mathbf{x}$ for a given measurement vector $\mathbf{y}$. It is a common assumption that both distributions are Gaussian with some mean vectors and covariance matrices. In general, for mean vector μ and covariance matrix $\mathbf{S}$, the probability density function of a Gaussian random variable is

$$P(\mathbf{x}) = \frac{1}{(2\pi)^{\frac{n}{2}}|\mathbf{S}|^{\frac{1}{2}}} \exp(-\frac{1}{2}(\mathbf{x}-\mu)^T\mathbf{S}^{-1}(\mathbf{x}-\mu)). \tag{2.7}$$

The optimal or maximum *a posteriori* (MAP) solution for $\mathbf{x}$ is given by the maximum of $P(\mathbf{x}|\mathbf{y})$. Bayes' theorem tells us that:

$$P(\mathbf{x}|\mathbf{y}) = \frac{P(\mathbf{y}|\mathbf{x})P(\mathbf{x})}{P(\mathbf{y})}. \tag{2.8}$$

Using this, we get that finding the MAP solution is equivalent to finding the vector $\mathbf{x}$ for which the expression

$$-2\ln P(\mathbf{x}|\mathbf{y}) = (\mathbf{x}-\mathbf{x_a})^T{\mathbf{S_a}}^{-1}(\mathbf{x}-\mathbf{x_a}) + (\mathbf{y}-\mathbf{F}(\mathbf{x}))^T{\mathbf{S_y}}^{-1}(\mathbf{y}-\mathbf{F}(\mathbf{x})) \tag{2.9}$$

is minimal. In equation 2.9 $\mathbf{x_a}$ and $\mathbf{S_a}$ represent the mean vector and the covariance matrix of the *a priori* distribution, while $\mathbf{S_y}$ is an experimental error covariance.

Nonlinear optimization

The expression on the right side of equation 2.9 is moderately non-linear, so the standard approaches such as Newton's method or the inverse Hessian method can be used for minimizing such a function.

When minimizing any cost function $\mathbf{C}(\mathbf{x})$ using the inverse Hessian method, a minimum is obtained by the following iterative scheme:

$$\mathbf{x_{i+1}} = \mathbf{x_i} - [\nabla \mathbf{g}(\mathbf{x_i})]^{-1} \mathbf{g}(\mathbf{x_i}), \tag{2.10}$$

where $\mathbf{g}(\mathbf{x}) = \nabla C(\mathbf{x})$ is the first derivative of the cost function. This method looks for a place where the gradient of $\mathbf{C}$ is zero, so it may find a minimum, but it may also head off towards some other stationary point. Gradient descent is a simple method in which the idea is to move $\mathbf{x_i}$ in the direction which seems to make $\mathbf{C}$ become smaller faster than any other direction. Combining it with the inverse Hessian we get a method with more sophisticated iterations:

$$\mathbf{x_{i+1}} = \mathbf{x_i} - [\gamma + \nabla \mathbf{g}(\mathbf{x_i})]^{-1} \mathbf{g}(\mathbf{x_i}). \tag{2.11}$$

This is known as the Levenberg-Marquardt formula. The scalar parameter γ can be adjusted during iterations. For large γ the formula is similar to the gradient descent so it is more reliable, but converges slowly. On the other hand, when γ is small, the formula is similar to the inverse Hessian method which could move $\mathbf{x_i}$ towards the wrong stationary point of $\mathbf{g}$, but converges faster.

Chapter 3

Accelerating JURASSIC-scatter

The main task in this master's thesis project was to accelerate the JURASSIC-scatter forward model. As mentioned in Chapter 2, the JURASSIC forward model has been ported to GPUs using CUDA, while JURASSIC-scatter, in which scattering of infrared radiation on liquid or solid particles is accounted for, does not support GPUs. Figure 3.1 presents how the JURASSIC-scatter and JURASSIC-GPU implementations were connected to get the accelerated forward model, which takes scattering into account.

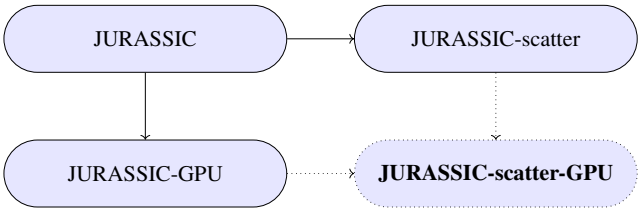


Figure 3.1: Projects based on JURASSIC. JURASSIC-scatter-GPU is created by combining JURASSIC-scatter and JURASSIC-GPU.

3.1 Connecting projects

This section shows how the JURASSIC-scatter code and the JURASSIC-GPU code are connected. Each of these codes is in a separate GitHub repository. The design question is whether a new repository should be created for JURASSIC-scatter-GPU or if it is better to combine these two projects to not further complicate code structures and maintainability.

Figure 3.2 shows the file organization of the JURASSIC-scatter-GPU project. There are actually more files than shown, but it is restricted to the most important ones. It was decided to use the `git clone --branch` feature to put JURASSIC-scatter and JURASSIC-GPU in the same place. In Chapter 4 is shown how the JURASSIC-scatter and JURASSIC-GPU

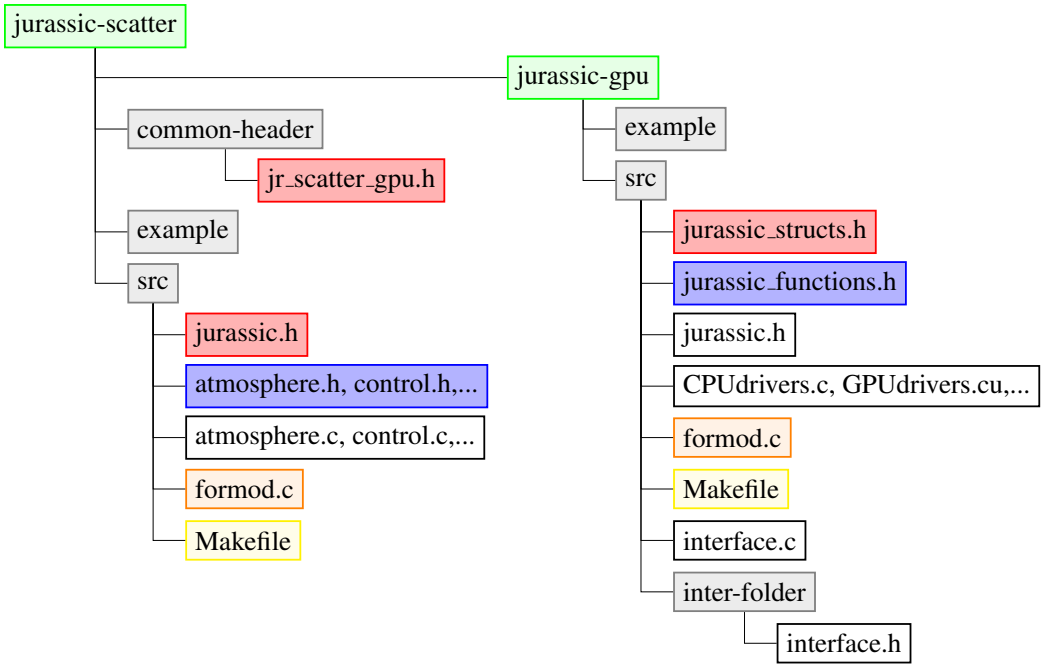


Figure 3.2: Connected repositories. In this phase of the master's thesis project it was decided to connect the projects by cloning the JURASSIC-GPU git repository in a subfolder inside JURASSIC-scatter. Similar folders and files from the two projects are colored the same.

project were merged into a new repository after it was verified that JURASSIC-scatter-GPU was working properly and was achieving better performance results than JURASSIC-scatter.

JURASSIC-scatter and JURASSIC-GPU both have their own header files that contain macros, constants and declarations of important structures. One way to connect them is to leave these header files separated and convert one type to another when needed, but since the ultimate goal of this project is to have a unified JURASSIC code, for the connecting it was decided to use a common header file that both projects must include. When connecting, declarations of all the functions from JURASSIC-GPU that JURASSIC-scatter directly uses were put in the header file `inter-folder/interface.h` inside the JURASSIC-GPU project and this file was included by the JURASSIC-scatter files from which the JURASSC-GPU functionalities are used. The projects have a number of functions with the same name, so the name collision problem was solved by adding the prefix "`jur_`" to the names of such functions of the JURASSIC-GPU project.

Since the JURASSIC-scatter and JURASSIC-GPU projects were not developed at the same time nor by the same person, an important part of connecting them was to determine

all differences between them. When connecting the projects to accelerate JURASSIC-scatter, it was not necessary to resolve all the differences in the best possible way, but rather make sure that the correctness of the program is not compromised. All the differences between the codes as well as the way in which they were resolved are given in Chapter 4.

3.2 JURASSIC-scatter-GPU realization

The number of scattering levels sca_{mult} is one of the input parameters of the JURASSIC-scatter forward model, which is not present in the JURASSIC forward model. This parameter can be considered as the depth of recursion. If $sca_{mult} = 0$, then JURASSIC-scatter calculates only primary rays, which are treated without scattering, so this case is equivalent to JURASSIC without scattering. If $sca_{mult} = 1$, then there are primary rays treated with scattering and secondary rays treated without scattering. This *single scattering* case is the most common in practice. In Figure 3.3 a scattering scenario for $sca_{mult} = 2$ is presented. Note that each ray at the lowest level of recursion will exactly perform the operations provided by the forward model in JURASSIC-GPU. This allows to limit the efforts of porting the scattering module to a restructuring of the high-level routines. This is done following the approach in JURASSIC-GPU. JURASSIC-scatter-GPU is divided into 3 phases:

1. CPU-Prepare
2. GPU-Execute
3. CPU-Collect

In the following subsections these three phases are explained in more detail.

CPU-Prepare

The main idea is that raytracing and recursively calculating the radiance of rays with scattering stays on the CPU and only in the case without scattering it is executed on the GPU, leveraging JURASSIC-GPU CUDA kernels. Hence, the CPU-Prepare phase, until coming to the lowest level of recursion, does the same as JURASSIC-scatter and at the lowest level the rays are saved in memory and sent to JURASSIC-GPU. There were multiple options of data structures to save the lowest level rays. One solution could be using an array, but for $sca_{mult} > 1$ it might be difficult to determine the size of the array and ray indices if allocating memory is done statically. To circumvent this, a queue structure, which is called *work-queue*, is used and in the CPU-Prepare phase the information about rays from the lowest level of recursion is pushed into it. Figure 3.3 shows a call-tree, which represents the recursive function calls. The highlighted leaf nodes of the graph represent the work-queue to which the information about rays from the leaf nodes is written into. First In, First

Out (FIFO) is the most important property of the queue structure. It means that leaf nodes of the tree have to be visited in the same order in the CPU-Prepare and CPU-Collect phases, and therefore the CPU-prepare phase cannot be parallelized without further modification.

GPU-Execute

In the GPU-Execute phase the radiation for each ray in the work-queue is calculated using functions from the JURASSIC-GPU project. Rays in the work-queue are divided into smaller packages, which are passed to JURASSIC-GPU kernel functions in parallel using OpenMP parallelism. More precisely, the number of packages that can be processed at the same time on one GPU is calculated and the corresponding GPU memory for these packages is allocated at start. Then this number of OpenMP threads is used to calculate the radiances for the rays in the given packages in parallel. With this mechanism GPU memory balancing is ensured.

CPU-Collect

The CPU-Collect phase is very similar to the CPU-Prepare phase. The difference is that in this case instead of pushing information about leaf nodes of the call-tree to the work-queue, its radiances calculated in the GPU-execute phase have to be read and passed to the parent node. For levels that are not the lowest level, the calculation is again exactly the same as in the original JURASSIC-scatter.

Because of the FIFO property of the work-queue an additional adjustment was done to parallelize the CPU-Prepare and CPU-Collect phases. Therefore multiple work-queues are introduced. Each primary ray has its own work-queue, so inside a subtree of a primary ray node the calculation is performed in serial, but OpenMP can be used to parallelize over primary rays. Figure 3.4 visualizes this idea.

3.3 Performance results

All performance results reported here were obtained on the Jülich Wizzard for European Leadership Science (JUWELS) supercomputing system, which is introduced in Section 1.1. JURASSIC-scatter-GPU was benchmarked on the JUWELS Booster and JURASSIC-scatter on the JUWELS Cluster. Since a JUWELS Booster node contains four GPUs, four MPI ranks per node were used in both cases to exploit their full capacity. Consequently, for JURASSIC-scatter-GPU each MPI rank has its own GPU and for both JURASSIC-scatter-GPU and JURASSIC-scatter there are twelve CPU cores and OpenMP threads per MPI rank. The CUDA runtime and compiler version were 11.3, while GCC version 9.3.0 was

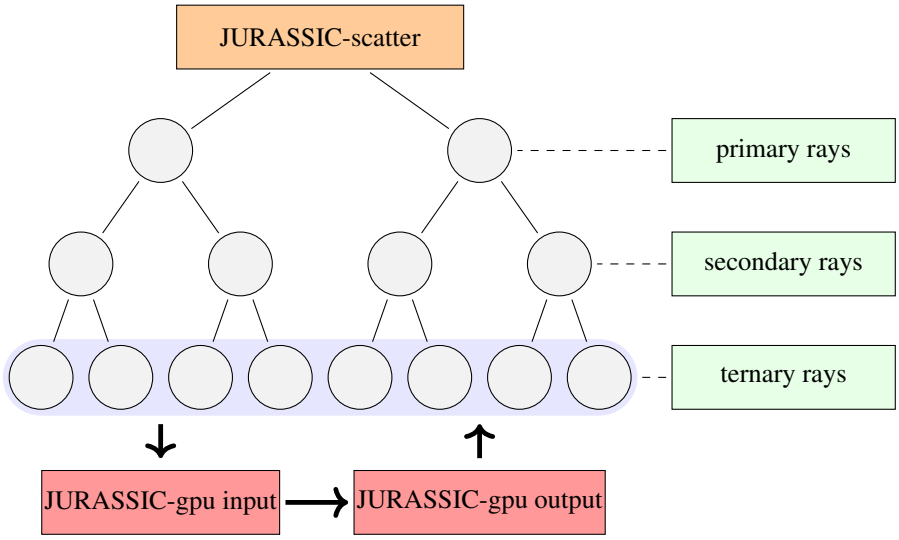


Figure 3.3: JURASSIC-scatter-GPU. Leaf nodes of the tree submit work packages to a work-queue, which is then processed on the GPU.

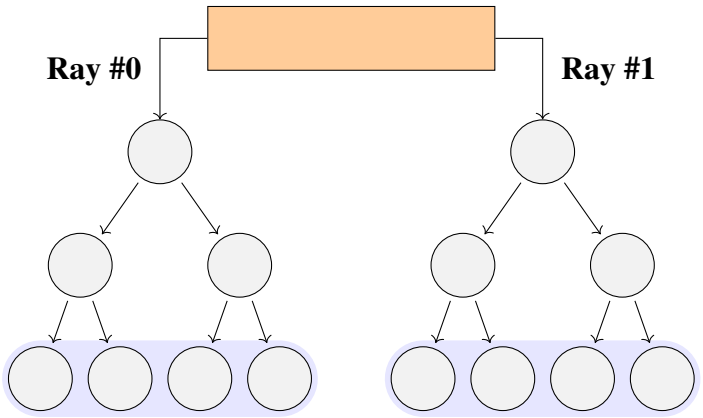


Figure 3.4: Multiple queues. Separate queues are necessary to exploit CPU thread parallelism during the prepare and collect phase.

employed to compile the CPU code. In addition, the GSL/2.6 module was loaded to run the programs.

Test cases:

- simulated data for the CRyogenic Infrared Spectrometers and Telescopes for the Atmosphere - New Frontiers (CRISTA-NF) instrument that it would measure in the presence of clouds [22]
- CRISTA-NF measures the thermal emissions of the atmosphere in the mid-infrared region from 4 to 15 μm in an altitude range from flight altitude (up to 20 km) down to approximately 5 km
- ~2000 test cases
- test cases depend on:
    1. profile of 13 trace gases, temperature and pressure
    2. cloud vertical thickness
    3. cloud layer bottom altitude
    4. particle size distribution (log-normal)
- 32 spectral radiance channels[1]
- 84 primary rays per test case
- $sca_{mult} = 1$: only primary and secondary rays – *single scattering*

Figure 3.5 shows a comparison of execution times of JURASSIC-scatter-GPU and the reference JURASSIC-scatter program for ten different cloud scenarios. In each scenario the number of primary rays equals 84, but the number of secondary rays depends on the thickness of the cloud, because in a thicker cloud more secondary rays are generated per primary ray so both programs need a longer execution time. In the first two scenarios the cloud vertical thickness is 0.5 km, in the second two 1 km, in the third two 2 km, in the fourth two 4 km, and in the last two 8 km. It can be seen that in all scenarios the achieved speedup is around 10×.

In Figure 3.6 the execution times of JURASSIC-scatter-GPU and JURASSIC-scatter are compared, for the same cloud scenario, but different numbers of primary rays. In each

[1] In the original CRISTA-NF data there are actually 33 spectral radiance channels, but it was decided to ignore one in this benchmark because the CUDA warp size is 32 and vectorization is done over radiance channels [4].

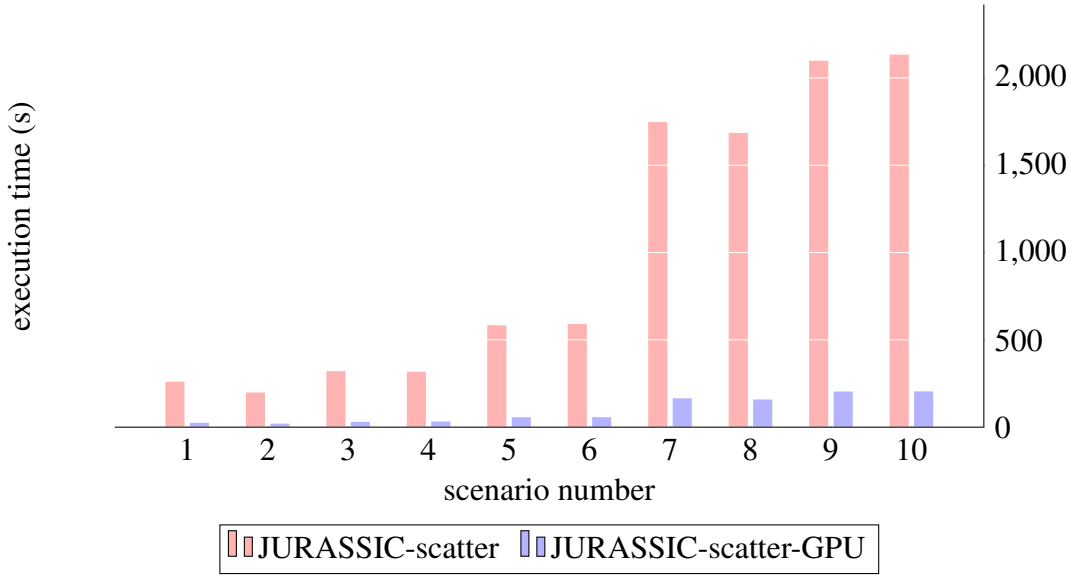


Figure 3.5: Execution times for different scenarios. JURASSIC-scatter-GPU was benchmarked on JUWELS Booster. JURASSIC-scatter was benchmarked on JUWELS Cluster because in practice is not efficient to occupy more expensive GPU nodes when not doing computations on GPUs. The execution times of JURASSIC-scatter were internally also measured on JUWELS booster, and in this case the speedup factor was even better because the EPYC CPU was on average 12% slower than the Intel CPU.

of these test cases there are around 2300 secondary rays per primary ray. The execution time of JURASSIC-scatter-GPU increases linearly with the number of primary rays. The execution time of the JURASSIC-scatter also increases linearly with the number of primary rays, although at first it may not seem so. The reason why JURASSIC-scatter in the scenario with 50 rays and the scenario with 60 rays has a similar execution time is that in both cases, since twelve CPU cores are used, at least one core must calculate the radiation for five rays. For the test case with 80 rays the achieved speedup is again around 10×, and for a smaller number of rays the speedup is even better. For example, the speedup for the test case with 10 rays is around 20×.

An additional reason for merging the JURASSIC-scatter and JURASSIC-GPU codes is that in JURASSIC-scatter-GPU, whose execution times are presented in Figures 3.5 and 3.6, raytracing is performed on CPUs. After removing duplicates from JURASSIC-scatter-GPU and porting its raytracer to GPUs, an even better speedup factors can be expected. Hence, in Chapter 4, merging the JURASSIC-scatter and JURASSIC-GPU projects into the unified code project as well as accelerating the raytracer that includes scattering is presented.

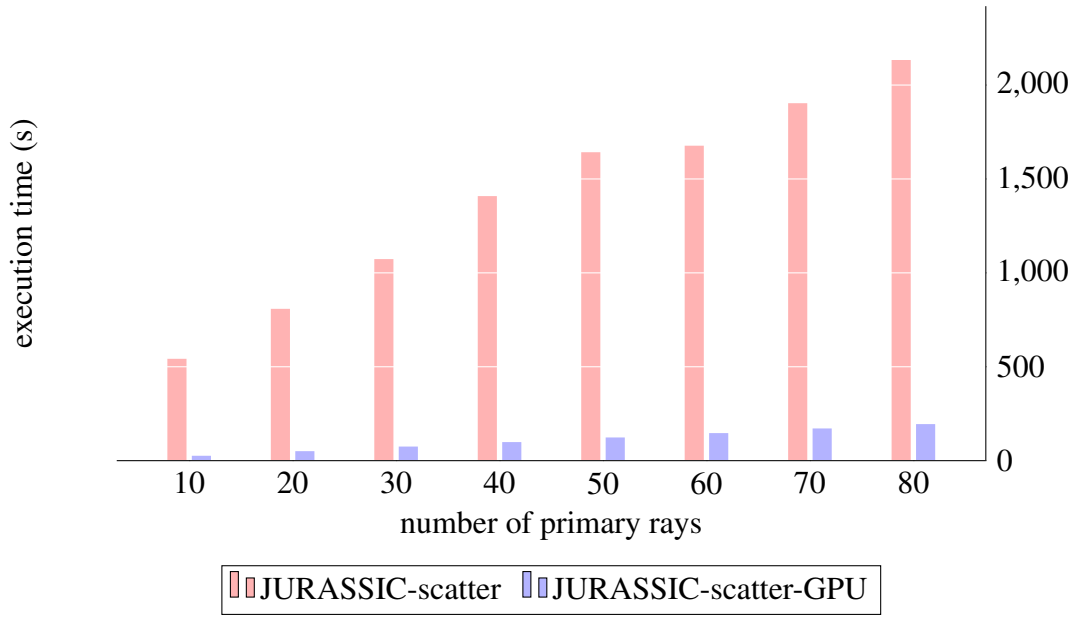


Figure 3.6: Execution times for different numbers of primary rays. The achieved speedup ranges from 10 (80 rays) to 20 (10 rays).

Chapter 4

JURASSIC-unified

After connecting the JURASSIC-GPU and JURASSIC-scatter projects to accelerate the numerical simulation of multiple scattering, the next step was to merge these two projects and JURASSIC-scatter-GPU – the accelerated version of JURASSIC-scatter – into the new separated project JURASSIC-unified. Figure 4.1 illustrates this idea. This was done because JURASSIC-scatter-GPU had a lot of code duplications, which made the code difficult to understand and to maintain.

In this chapter the main differences between the merged projects as well as the way in which they have been resolved are explained. One additional feature – simultaneous calculation of radiation transport for rays with different atmospheres, which is not part of any previous version, is presented. In the last part of this chapter it is explained how to link the JURASSIC-unified project as a library into the JURASSIC reference implementation to take advantage of the accelerated code and new features.

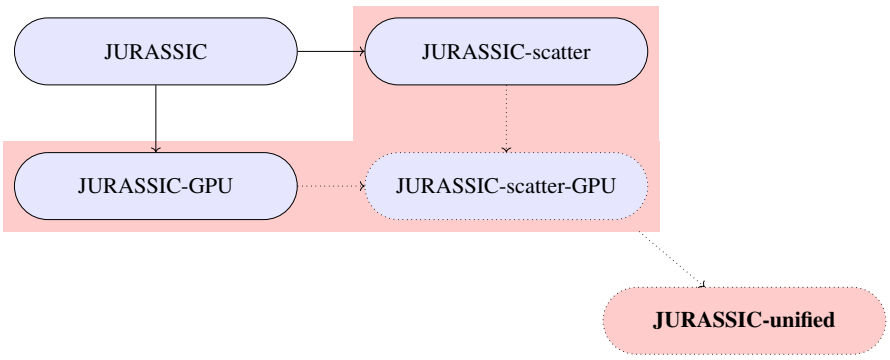


Figure 4.1: JURASSIC-scatter, JURASSIC-GPU and JURASSIC-scatter-GPU are merged into the new unified code project, which includes the GPU acceleration and (multiple) aerosol scattering. The JURASSIC-unified model is available at `https://github.com/slcs-jsc/jurassic-unified` (last access: 3 July 2022).

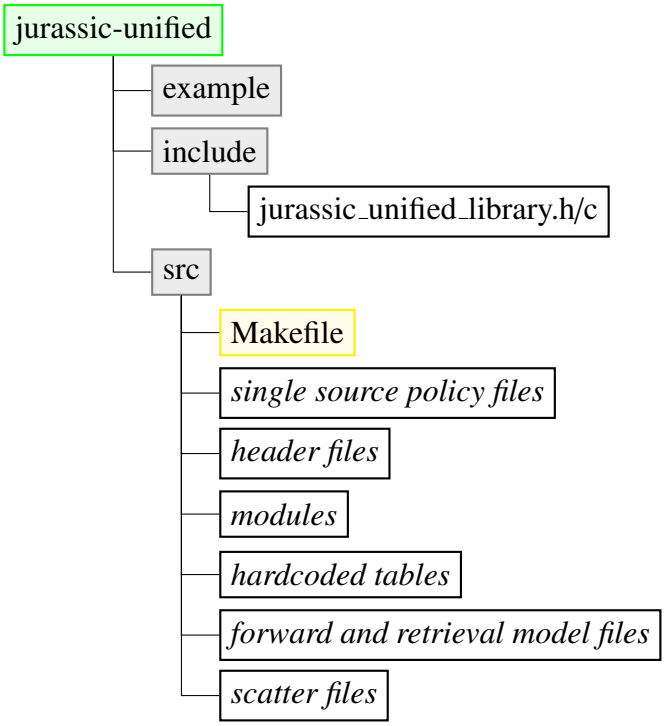


Figure 4.2: Files organization in the JURASSIC-unified project. Unlike Figure 3.2, there is a single source folder.

4.1 Merging projects

Files organization

When accelerating JURASSIC-scatter, the JURASSIC-GPU project was cloned into the JURASSIC-scatter repository, compiled and wrapped into the `C` static library, and then linked to JURASSIC-scatter. There were also a lot of name collisions of files, structs and functions from the projects. Figure 4.2 shows how files in JURASSIC-unified are organized. Again, not all files and folders are shown, but the most important ones are there. Also the source files are represented by their groups, which will be explained below in this section.

The main difference between the file organization in JURASSIC-scatter and JURASSIC-GPU is that in JURASSIC-scatter functions are divided into files depending on the part of the model they are performing. A single source code policy has been introduced in JURASSIC-GPU. JURASSIC-unified also has the single source code policy and all its functionalities related to raytracing and radiative transfer that should run both on CPU and GPU are defined in the `jr_common.h` header file, which is then included to the architecture-

specific implementations `CPUdrivers.c` and `GPUdrivers.cu`. Functionalities that are run only on the CPU, such as reading control parameters and reading observations data are defined in `jurassic.c`.

All radiative transfer functionalities in JURASSIC-scatter that are not part of JURASSIC-GPU, i.e. the functions in charge of scattering, run only on the CPU. When developing the JURASSIC-unified project, it was decided not to put these functions to the `jurassic.c` file, but instead to have them in files with the similar name as the files in which they were in JURASSIC-scatter project to distinguish functions in charge of scattering from the functions adopted from the JURASSIC-GPU project and to make it easier for JURASSIC-scatter project users to switch to JURASSIC-unified. Raytracing from JURASSIC-scatter in the JURASSIC-unified implementation should run on both CPUs and GPUs, so the `raytrace` function had to be adapted to make it possible to run it on the GPU. Since JURASSIC-unified is also adapted to be used by the reference project as a library, function name collisions are avoided by adding the "`jur_`" prefix to all function names. To make the functions introduced from the JURASSIC-scatter project even easier to distinguish, the prefix "`jur_sca_`" has been added to them.

In addition to the standard `.gitignore` and `Makefile` files, the source folder contains source files that are divided into several logical groups:

- *single source policy files*: `CPUdrivers.c`, `GPUdrivers.c`, `jr_common.h` – already described above.
- *header files*: `jurassic.h`, `jurassic_dimensions.h`, `jurassic_functions.h`, `jurassic_structs.h`, `sca_gpu_interface.h` – the header file from JURASSIC-GPU is here divided into multiple files for easier development and maintenance. This was necessary when preparing the JURASSIC-unified library because the JURASSIC reference project has to include its structs, but should not include all functions. Definitions of the functions declared in `jurassic_functions.h` are found in `jurassic.c`. The `sca_gpu_interface.h` file differs from the others because it contains declarations of functions in `CPUdrivers.c` and is included by the *scattering files*.
- *modules*: `module_brightness.c`, `module_climatology.c`, `module_interpolate.c`, `module_kernel.c`, `module_limb.c`, `module_nadir.c`, `module_planck.c`, `module_raytrace.c` – contain *main* functions, used to independently run parts of the model and to create climatologies.
- *hardcoded tables*: `climatology.tbl`, `ctmco2.tbl`, `ctmh2o.tbl`, `ctmn2.tbl`, `ctmo2.tbl` – used for creating climatologies and for evaluating the emission continua of the trace gases.

- *forward and retrieval model files*: `formod.c`, `sca_formod.c`, `multi_atm_formod.c`, `sca_retrieval.c` – contain *main* functions, modules for running the forward model without scattering, the forward model with scattering, the forward model without scattering for rays with possibly different atmospheres and the retrieval model with scattering, respectively.
- *scatter files*: `sca_forwardmodel.h/c`, `sca_scatter.h/c`, `sca_retrievalmodel.h/c`, `sca_workqueue.h/c` – files that contain functionality that allows the simulation including aerosol scattering.

After each code change, the program was tested on a set of test cases to make sure it was still working properly. To run the program on a supercomputer, `submit.sh`, which may take the type of the forward/retrieval model as an argument, can be ran. The python script `diff.py` is used to compare the calculated radiations with the reference data, calculated by the reference projects. In the `testing` folder, which is not contained in JURASSIC-scatter and JURASSIC-GPU repositories, tests used during the development of the project are contained. Five types of tests were used:

- `clear_air_test` – to test the forward model without scattering.
- `large_testset` – to test and benchmark the forward model with scattering.
- `multiple_atmospheres` – to test the multiple atmospheres feature.
- `small_testset` – similar to `large_testest`, but smaller workload.
- `retrieval` – to test a retrieval run with scattering.

Source code differences

The first observed difference between JURASSIC-scatter and JURASSIC-GPU were the different names of the constants for dimensions, e.g. in JURASSIC-scatter, the constant which describes the number of rays is called `NRMAX`, while the same constant in JURASSIC-GPU and in the JURASSIC reference project is called `NR`. It was decided to use the option from JURASSIC-scatter in the JURASSIC-unified project. In Section 4.3 will be explained why having "`MAX`" in dimension variables names is necessary when preparing JURASSIC-unified as a library.

The main difference between the JURASSIC-scatter and JURASSIC-GPU projects were the different data structures they use, so here the purpose of all structs, their differences and the way how they were resolved are presented:

- `ctl_t` – forward model control parameters: minor differences, control parameters from JURASSIC-scatter which were not contained in JURASSIC-GPU were added to get the `ctl_t` struct in JURASSIC-unified. Also, the function that reads the control parameters had to be changed so that the control parameters from both JURASSIC-scatter and JURASSIC-GPU are included.

- `los_t` – line-of-sight data: in JURASSIC-scatter a structure of arrays (SoA) is used to represent the segments of the line of sight, while in JURASSIC-GPU that is done by array of structures (AoS). During accelerating JURASSIC-scatter, both were used and the `C` structs were converted one to another when necessary. For JURASSIC-unified it was decided to use only an array of structures to represent the segments of the line of sight.

- `obs_t` – observation geometry and radiance data: structs from the two projects are almost equal, but the difference is that the indices of the 2d-array `rad` are not in the same order. In JURASSIC-scatter it is declared as `rad[ND][NR]` and in JURASSIC-GPU as `rad[NR][ND]`, where `NR` is number of rays and `ND` number of detectors or radiance channels. It was decided to use `rad[NR][ND]` in JURASSIC-unified, because of more efficient memory access.

- `tbl_t` – emissivity look-up tables: here was the similar problem as with `obs_t`. JURASSIC-scatter uses the same array-ordering to store emissivity look-up tables as the JURASSIC reference implementation, and in JURASSIC-GPU arrays were restructured to achieve vectorization over radiance channels to get the best possible exploitation of the GPU memory bandwidth [4]. During accelerating JURASSIC-scatter both JURASSIC-scatter and JURASSIC-GPU tables were used, so the reading of the tables was performed twice. One more difference between them is that JURASSIC-GPU tables are optimized to be read in ASCII format and JURASSIC-scatter can also be read in binary format. For JURASSIC-unified, it was decided to use only the tables from the JURASSIC-GPU project. In the JURASSIC-scatter project, `tbl_t` is used when computing the segment emissivities according to the EGA method so the `intpol_tbl` function from JURASSIC-scatter was replaced by the `apply_ega_core` from the JURASSIC-GPU project. Similarly, in order to avoid the duplication of code in charge of evaluating the emission continua of trace gasses and evaluating the Planck source function, the `formod_continua` was replaced by `continua_core_CPU` and `srcfunc_planck` was replaced by the `src_planck_core` function from the JURASSIC-GPU project.

- `atm_t` – atmospheric data: This is the only structure which was completely the same in both projects.

- `aero_t` – aerosol and cloud properties: this is not part of JURASSIC-GPU and therefore it was simple to introduce it to JURASSIC-scatter-GPU.
- `ret_t` – retrieval control parameters: This is only a part of JURASSIC-scatter.

Accelerating the raytracer

The raytrace algorithm should run on both, CPUs and GPUs, so the JURASSIC-scatter implementation has to be adapted to make it possible to run it on GPUs. The main difference between the JURASSIC-scatter raytracer and the JURASSIC-GPU raytracer is that in the version with scattering the function `add_aerosol_layers` is called after raytracing. This function adds additional points to the computed line of sight to make sure that points with altitudes similar to the altitudes of the borders of the aerosol layers are added. In the original JURASSIC-scatter implementation the line of sight points were copied to a dynamically allocated array and after that points around cloud edges were added. To avoid this dynamic allocations, the procedure was adapted in a way that new points are added to an existing array. One more problem with porting the function `add_aerosol_layers` to the GPU was the use of upper-level functions such as `gsl_stats_min`, `gsl_stats_min_index` and `gsl_sort`. The first two were easily replaced, but the sorting function had to be written from scratch. Two different sorting algorithms were tried: iterative merge sort and bubble sort, and one of them had to be chosen. Merge sort has to be iterative because recursions are not allowed inside CUDA kernels. Bubble sort has an inferior time complexity then merge sort, but does not use additional memory, and since the array which has to be sorted usually has around a hundred elements, bubble sort is chosen for the current version, but it can be easily replaced with iterative merge sort in the future.

GPU register tuning similar to the one in [4] was performed to decrease the number of used registers and the cumulative stack size. The NVIDIA `C` compiler can report on register usage, spill load/stores and cumulative stack size for each CUDA kernel. The raytracer kernel in the JURASSIC-GPU implementation used 130 registers and had 1272 bytes cumulative stack size. After the `add_aerosol_layers` function, which is called in the simulation of radiance with scattering, was added to the end of the raytracer, the kernel used 136 registers and had 2014 bytes cumulative stack size. The number of registers used by the kernel as well as its cumulative stack size is determined during compilation. Since in the case without scattering the `add_aerosol_layers` function is not called, registers and stack memory allocated for this functions are superfluous. Because of that, multiversioning has been applied to generate two versions of the raytracer with a single code. After that, the raytracer kernel for the version without scattering used 130 registers and had 1288 bytes cumulative stack size, which is very similar to the JURASSIC-GPU raytracer kernel. The JURASSIC-unified raytracer kernel with scattering now uses 136 registers and has 2096

bytes cumulative stack size, which is again very similar to the JURASSIC-unified raytracer kernel before multiversioning.

After determining the lines of sight with the raytracer, radiative transfer among those lines has to be calculated. These two parts are in the CUDA implementation written in two CUDA kernels which are launched one after another. CUDA kernel launches are asynchronous, which means that the CPU thread continues processing host code before the launched GPU kernel has actually begun executing. On the GPU side, if CUDA kernels are launched in the same stream, as in the JURASSIC-unified, they will be executed in the order of calling. Adding a `cudaDeviceSynchronize()` barrier after the first CUDA call, to force the CPU thread to wait for the compute device to finish, further improved the overall execution time. The most probable reason for this is that, after separating the two kernel with the synchronization, the GPU cache memory usage became more efficient. An in-depth analysis would be required to proof this hypothesis.

In addition to the execution times of the JURASSIC-scatter and JURASSIC-scatter-GPU implementations presented in Figure 3.5, Figure 4.3 contains the execution times of the JURASSIC-unified implementation.

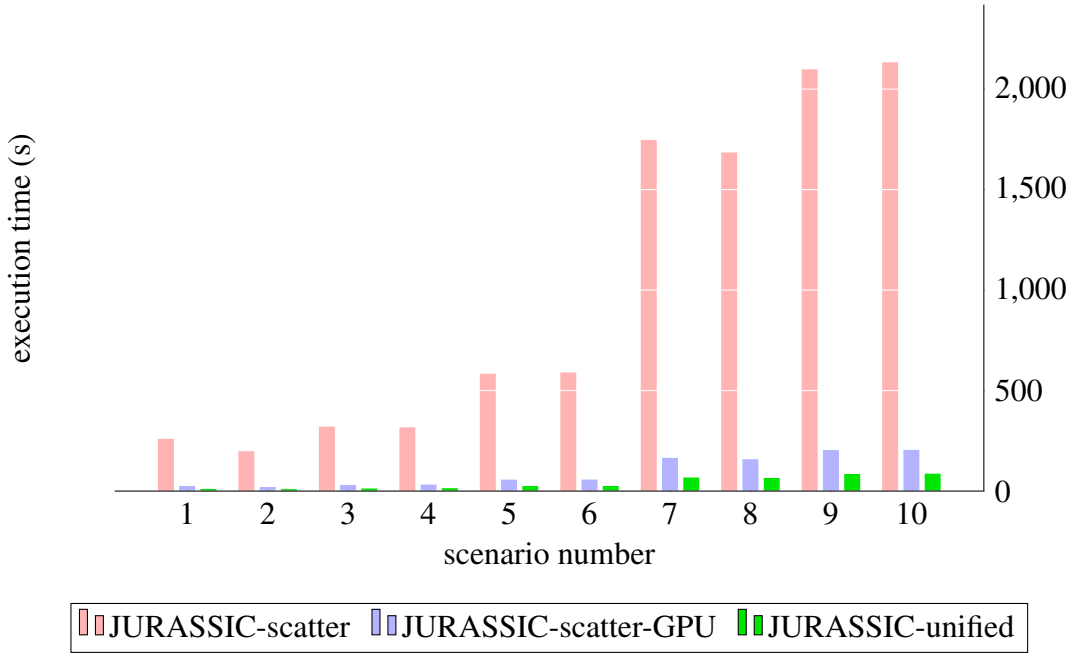


Figure 4.3: Execution times for different scenarios. Along with results shown in Figure 3.5, the results for a radiative transport calculation with scattering, after accelerating the raytracer, are shown in green. Both GPU implementations are executed on JUWELS Booster nodes whereas the reference code has been timed on JUWELS Cluster nodes comprising CPUs only.

Figure 4.4 shows the speed factors achieved by the JURASSIC-scatter-GPU and JURASSIC-unified implementation comparing the JURASSIC-scatter model, which executes only on CPUs. Unifying the code and accelerating the raytracer significantly increased the speedup factor.

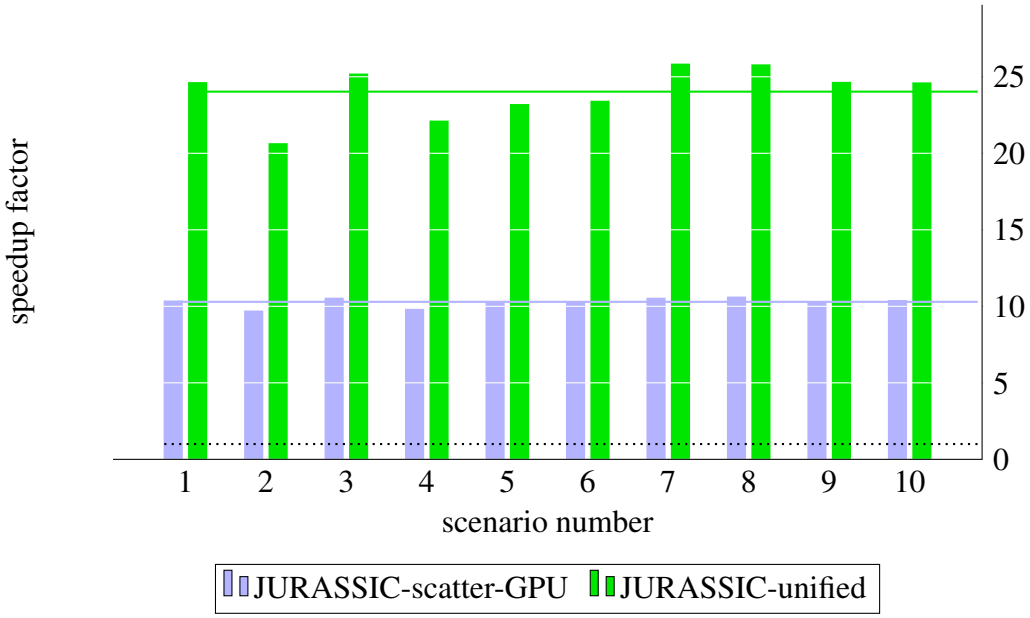


Figure 4.4: Comparison of the speedup factors. By unifying the code and accelerating the raytracer, the average speedup factor was increased from 10 to 24. The JURASSIC-scatter reference version v1.3 has been taken as baseline (dotted line).

New control parameters

Compared to the JURASSIC-scatter v1.3 documentation [13], JURASSIC-unified has two new control parameters. You can see more information about the new parameters in Table 4.1. Depending on the values of these two new parameters different modules are activated:

- if `MAX_QUEUE`=0: the work-queue is not used, so this module is very similar to the original JURASSIC-scatter v1.3
- if `MAX_QUEUE`<0: the memory for |`MAX_QUEUE`| rays is statically allocated for the work-queue, but in this scenario the Execute phase is performed on CPUs, as part of the JURASSIC-scatter implementation
- if `MAX_QUEUE`>0 and `USEGPU`=0: the Execute phase is again performed on CPUs, but as part of the JURASSIC-GPU implementation

- if `MAX_QUEUE`>0 and `USEGPU`=1: the Execute phase is performed on GPUs, if the code is not compiled with CUDA, the program will abort with an error
- if `MAX_QUEUE`>0 and `USEGPU`=−1: the Execute phase is tried to be performed on GPUs, if the code is not compiled with CUDA, the CPU-Execute phase will be performed

These different modules are also shown in Figure 4.5.

Table 4.1: New control flags for JURASSIC-unified.

flag name	purpose	default	options
Accelerating parameters			
MAX_QUEUE	upper bound of number of rays in the work-queue	10^6	0: do not use work-queue >0: call JURASSIC-GPU functions <0: do not call JURASSIC-GPU functions, in that case size of the work-queue is |MAX_QUEUE|
USEGPU	Use GPU-accelerated formod implementation	−1	0: never 1: always −1: if compiled with CUDA

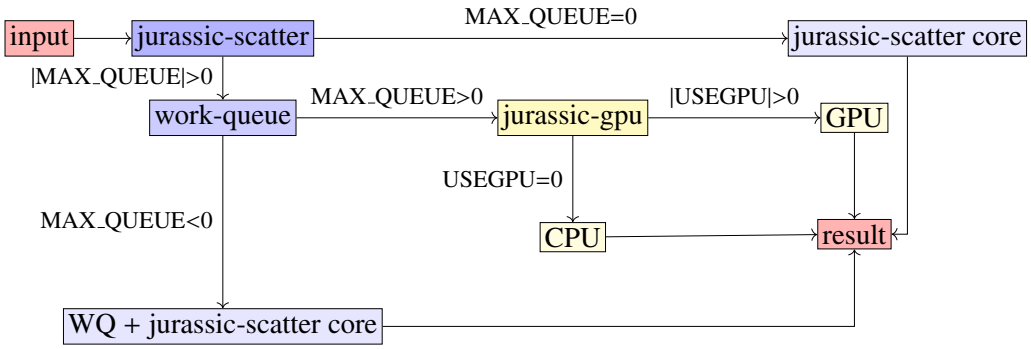


Figure 4.5: Control and data flow of JURASSIC-scatter-GPU. Using a work-queue structure, JURASSIC-scatter can call JURASSIC-GPU as solver.

4.2 Multiple atmospheres feature

One of the most important features that is supported in JURASSIC-unified and was not part of the previous JURASSIC versions is simultaneous simulation of radiation transport for observations with different atmospheric data. This feature is supported only when simulating radiance without scattering, because it was developed for the JURASSIC-unified

library, which will be used by the JURASSIC reference model. Listing 4.1 shows the declaration of the function that enables the use of the multiple atmospheres feature as well as the definition of the function which is similar to the standard `formod` function in the JURASSIC models.

Listing 4.1: Declaration of the `jur_formod_multiple_packages` function and definition of the `jur_formod` function in the JURASSIC-unified implementation.

```
void jur_formod_multiple_packages(ctl_t const *ctl,
                                  atm_t *atm,
                                  obs_t *obs, int n,
                                  int32_t const *atm_id,
                                  aero_t const *aero);

void jur_formod(ctl_t const *ctl, atm_t *atm, obs_t *obs) {
    jur_formod_multiple_packages(ctl, atm, obs, 1, NULL, NULL);
}
```

In each `obs_t` instance, a package of up to `NR` (or `NRMAX`) observations is stored, where `NR` is the predetermined maximum number of rays in one package. To simulate the radiance for more then `NR` rays simultaneously, multiple packages have to be used. For example, if `NR` = 1000 and the radiance of 2500 rays has to be calculated, three packages of sizes 1000, 1000 and 500, respectively, will be used. The fourth parameter `n` in the `jur_formod_multiple_packages` function is the number of packages and the third argument, pointer `obs`, should point to the first element of the array of `n` observations packages. When simulating the radiance without scattering, the last argument, `aero`, must be set to `NULL`.

The `formod` function from the JURASSIC reference implementation takes the pointers to the `ctl_t`, `atm_t` and `obs_t` instances as arguments. The `jur_formod_multiple_packages` is similar, but to be able to use the multiple atmospheres feature, the pointer to the first element of the array of `atm_t` instances should be given as the second argument. The `atm_id` array is used to determine which atmosphere belongs to which observation. Because of that, the length of the `atm_id` array must be equal to the number of rays for which the radiance is simulated. The elements of this array should be non-negative integers, strictly less than the number of atmospheres in the array `atm`. Each number in the `atm_id` represents the index of the corresponding atmosphere from the `atm` array.

In the previous example scenario with 2500 rays, when calculating the radiance with multiple atmospheres, the `atm_id` should have 2500 elements. If each ray had its own atmosphere, the `atm` would also have 2500 elements, and the `atm_id` would contain 0, 1, 2,..., 2499, respectively. Of course, rays can also share an atmosphere, but in that case the length of the `atm` array would be less than 2500. The `jur_formod_multiple_packages` function can also be used to calculate the radiance for the rays with the same atmosphere.

For that the atm_id argument should be set to NULL or contain all zeros.

4.3 JURASSIC-unified as a library

JURASSIC-unified successfully replaces the JURASSIC-GPU and JURASSIC-scatter implementations, and even offers some new features. Although it would be best if JURASSIC-unified completely replaced the JURASSIC reference implementation, due to some new features of the JURASSIC reference model, it was decided to adapt JURASSIC-unified so that scientists who are using the JURASSIC reference model can use the benefits of GPU acceleration of the model, with minimal change of their code. Because of that, JURASSIC-unified was adapted to be used from the JURASSIC reference project as a library.

The main challenge in preparing the JURASSIC-unified project to be a library was the fact that many structs and functions share the name with those from the JURASSIC reference implementation. The function name collisions were resolved by adding the "jur_" prefix to the names of the functions from the JURASSIC-unified project. The similar thing could be done for the structs, but this would impair the readability of the code, so it was decided to leave the original struct names as ctl_t, obs_t and atm_t.

Figure 4.6 illustrates how the JURASSIC-unified library should be integrated into the JURASSIC reference project. At the beginning, one has to run the shell script generate_library.sh inside the JURASSIC-unified home directory. This shell script copies the JURASSIC-unified source files into the folder "unified_library" and adds the "jur_" prefix to the copied code struct names. At the end of the script, the code is compiled and wrapped into the C static library libjurassic_unified.a.

This C static library has to be linked when compiling the JURASSIC reference project, but to use it, the files from the include folder have to be included into the JURASSIC reference code. More precisely, the jurassic_unified_library.h file has to be included into the JURASSIC reference files from which the library functions will be used. The jurassic_unified_library.h file contains declarations of the functions offered by the JURASSIC-unified library.

In the JURASSIC projects dimensions are stored as C macro constants. The difference between the dimension names in JURASSIC reference and JURASSIC-unified is that those from the JURASSIC-unified project have the suffix "MAX". For example, the number of rays in the package is in the JURASSIC reference project called NR and in the JURASSIC-unified project NRMAX. The jurassic_unified_library.h, using #if, #error and #endif directives, takes care of the dimension macros. The compilation fails if the corresponding dimension from the JURASSIC-unified project is not greater than or equal to that from the JURASSIC reference project. When that happens, the JURASSIC-unified dimensions declared inside the src/jurassic_dimensions.h have

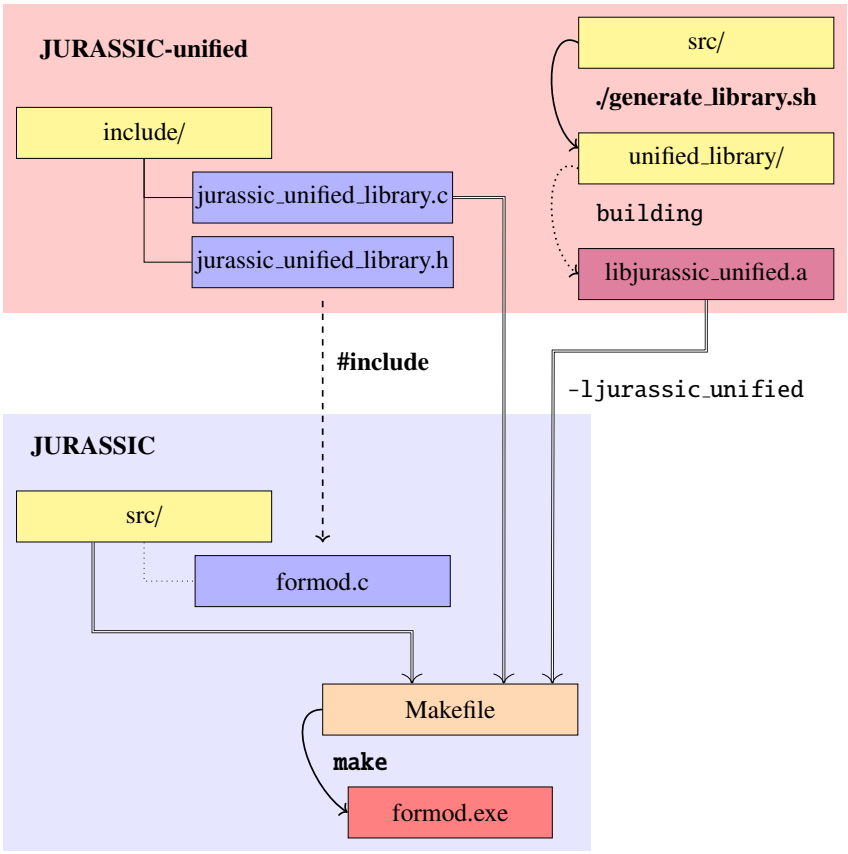


Figure 4.6: Integration of the JURASSIC-unified library into the reference project.

to be increased, and the `generate_library.sh` script **must** be run again to update the C static library file.

The `jurassic_unified_library.c` file with definitions of this functions has to be compiled and linked when compiling the JURASSIC reference project. In the home folder of the JURASSIC-unified repository you can find the "`reference_Makefile`" file which can be used as a *Makefile* for the JURASSIC reference implementation when connecting it with the JURASSIC-unified library.

To be simple for usage, the JURASSIC-unified library offers just two functions, whose declarations are shown in Listing 4.2.

Listing 4.2: Declaration of the functions offered by the JURASSIC-unified library.

```
jur_ctl_t *jur_unified_init(int argc, char *argv[]);
void jur_unified_formod_multiple_packages(atm_t const *atm,
                                          obs_t *obs,
                                          int num_of_obs_packages,
```

```
                                                int32_t const *atm_id);
```

At the beginning of the program, `jur_unified_init(argc, argv)` must be called. This function reads the control parameters and stores them into a static variable. Similarly, the emissivity look-up table is initialized in this function.

Since the JURASSIC reference model ignores scattering, the `jur_unified_formod_multiple_packages` function does not have `aero` among its parameters. It also does not take `ctl` and `tbl` as arguments, because they are already stored in the static variables initialized in the `jur_unified_init` function. The `jur_unified_formod_multiple_packages` function offers similar functionality as `jur_formod_multiple_packages` presented in Section 4.2, it supports simultaneous calculation of the radiances for possibly multiple observation packages, for rays with possibly different atmospheres.

Bibliography

[1] *Frontier First to Break the Exaflop Ceiling*, https://www.top500.org/news/ornls-frontier-first-to-break-the-exaflop-ceiling/, visited on 2022-06-20.

[2] *MPICH*, https://www.mpich.org/about/overview/, visited on 2022-06-20.

[3] *TOP500 list*, https://www.top500.org/project/, visited on 2022-06-20.

[4] P. F. Baumeister and L. Hoffmann, *Fast infrared radiative transfer calculations using graphics processing units: Jurassic-gpu v2.0*, Geoscientific Model Development **15** (2022), 1855–1874.

[5] P. F. Baumeister and L. Hoffmann, *GitHub Source Repository of JURASSIC-GPU*, https://github.com/slcs-jsc/jurassic-gpu, 2022.

[6] P. F. Baumeister, B. Rombach, T. Hater, S. Griessbach, L. Hoffmann, M. Bühler, and D. Pleiter, *Strategies for forward modelling of infrared radiative transfer on GPUs*, Parallel Computing is Everywhere **32** (2017), 369–380.

[7] M. Born, E. Wolf, A. B. Bhatia, P. C. Clemmow, D. Gabor, A. R. Stokes, A. M. Taylor, P. A. Wayman, and W. L. Wilcock, *Principles of Optics: Electromagnetic Theory of Propagation, Interference and Diffraction of Light*, 7th ed., Cambridge University Press, 1999.

[8] Pawsey Supercomputing Centre, *A brief history of supercomputing*, 2019, https://pawsey.org.au/wp-content/uploads/2019/01/PawseyHistorySupercomputing2018v3.pdf, visited on 2022-06-20.

[9] NVIDIA Corporation, *Cuda c++ programming guide*, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, visited on 2022-06-20.

[10] L. Dagum and R. Menon, *OpenMP: an industry standard API for shared-memory programming*, Computational Science & Engineering, IEEE **5** (1998), no. 1, 46–55.

[11] M. J. Flynn, *Very high speed computers*, Proceedings of the IEEE **54** (1966), no. 12, 1901–1909.

[12] MPI Forum, *MPI: A Message-Passing Interface Standard*, Techn. rep., USA, 1994.

[13] S. Griessbach and L. Hoffmann, *JURASSIC-scatter v1.3 documentation*, https://github.com/slcs-jsc/jurassic-scatter/blob/v1.3/docu/jurassic.pdf, 2019.

[14] S. Griessbach and L. Hoffmann, *GitHub Source Repository of JURASSIC-scatter*, https://github.com/slcs-jsc/jurassic-scatter, 2022.

[15] S. Griessbach, L. Hoffmann, M. Höpfner, M. Riese, and R. Spang, *Scattering in infrared radiative transfer: A comparison between the spectrally averaging model JURASSIC and the line-by-line model KOPRA*, Journal of Quantitative Spectroscopy and Radiative Transfer **127** (2013), 102–118, ISSN 0022-4073, https://www.sciencedirect.com/science/article/pii/S0022407313001969.

[16] M. Harris, *CUDA Pro Tip: Write Flexible Kernels with Gride-Stride Loops*, https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/, 2013.

[17] F. Hase and M. Höpfner, *Atmospheric ray path modeling for radiative transfer algorithms*, Appl. Opt. **38** (1999), no. 15, 3129–3133, http://opg.optica.org/ao/abstract.cfm?URI=ao-38-15-3129.

[18] L. Hoffmann, *GitHub Source Repository of JURASSIC*, https://github.com/slcs-jsc/jurassic, 2022.

[19] L. Hoffmann and M. J. Alexander, *Retrieval of stratospheric temperatures from Atmospheric Infrared Sounder radiance measurements for gravity wave studies*, J. Geophys. **114** (2009).

[20] L. Hoffmann, M. Kaufmann, R. Spang, R. Müller, J. J. Remedios, D. P. Moore, C. M. Volk, T. von Clarmann, and M. Riese, *Envisat MIPAS measurements of CFC-11: retrieval, validation, and climatology*, Atmos. Chem. Phys. **8** (2008), 3671–3688.

[21] D. J. Jacob, *Inverse modeling techniques. in observing systems for atmospheric composition*, (2007), 230–237.

[22] C. Kalicinsky, S. Griessbach, and R. Spang, *Radiative transfer simulations and observations of infrared spectra in the presence of polar stratospheric clouds: Detection and discrimination of cloud types*, (2020).

[23] NVIDIA, P. Vingelmann, and F.H.P. Fitzek, *Cuda, release: 10.2.89*, 2020, https://developer.nvidia.com/cuda-toolkit, visited on 2022-06-20.

[24] C.D. Rodgers, *Inverse Methods for Atmospheric Sounding: Theory and Practice*, Series on atmospheric, oceanic and planetary physics, World Scientific, 2000, ISBN 9789810227401, https://books.google.hr/books?id=p3b3ngEACAAJ.

[25] B. Slivnik, R. Trobec, B. Robic, and P. Bulić, *Introduction to Parallel Computing*, October 2018, ISBN 978-3-319-98832-0.

[26] T. Sterling, M. Anderson, and M. Brodowicz, *Chapter 3 - Commodity Clusters*, High Performance Computing (T. Sterling, M. Anderson, and M. Brodowicz, eds.), Morgan Kaufmann, Boston, 2018, pp. 83–114, ISBN 978-0-12-420158-3, https://www.sciencedirect.com/science/article/pii/B9780124201583000034.

[27] Supercomputing Support, *JUWELS: Modular Tier-0/1 Supercomputer at Jülich Supercomputing Centre*, Journal of large-scale research facilities JLSRF **5** (2019).

Sažetak

Juelich Rapid Spectral Simulation Code (JURASSIC) je model radijacijskog transporta koji se koristi u analizi atmosferskih daljinskih mjerenja zračenja iz srednje infracrvenog područja. Služi za otkrivanje stanja atmosfere na temelju zadanih mjerenja. Napisan je u programskom jeziku `C` i pomoću MPI/OpenMP hibridne paralelizacije prilagođen za izvođenje na superračunalima. JURASSIC-GPU je verzija JURASSIC-a kojoj je uz pomoć programskog jezika CUDA omogućeno efikasnije izvršavanje na grafičkim procesorima. Kako bi se u model uključile čestice, raspršenje infracrvenog zračenja je dodano u JURASSIC-scatter modelu, no taj model još nije imao koristi od ubrzavanja JURASSIC modela koji zanemaruje čestice.

Glavni cilj ovog diplomskog rada bio je ubrzati JURASSIC-scatter model. To je napravljeno njegovim kombiniranjem s JURASSIC-GPU modelom. Performanse implementacije koja koristi oba modela izmjerene su na JUWELS-u – jednom on najboljih svjetskih superračunala, i postignuto je ubrzanje od oko 10 puta. Nakon toga, kako bi se maknuli duplikati iz ta dva projekta i napravio razumljiviji kod koji će biti jednostavniji za održavanje, odlučeno je da se JURASSIC-scatter i JURASSIC-GPU spoje u novi projekt – JURASSIC-unified. Takva jedinstvena implementacija ima još bolje performanse – ubrzanje u odnosu na JURASSIC-scatter je oko 24 puta. Uz dodavanje jedne nove funkcionalnosti, JURASSIC-unified projekt je prilagođen tako da ga i model koji zanemaruje čestice može koristiti kao biblioteku.

Summary

The Juelich Rapid Spectral Simulation Code (JURASSIC) is a fast radiative transfer model for the analysis of atmospheric remote sensing measurements in the mid-infrared spectral region. It is used to derive the state of the atmosphere from the measurements. It was written in C and features an MPI/OpenMP hybrid parallelization for use on supercomputers. JURASSIC-GPU was developed by porting JURASSIC to GPUs using the CUDA programming language. To incorporate particles into the model, scattering of the infrared radiation on the particles was accounted for in the JURASSIC-scatter model, but this model so far did not benefit from tuning and acceleration of the reference model.

The goal of the work in this thesis was to accelerate the JURASSIC-scatter forward model. It was done by combining it with JURASSIC-GPU. The implementation that uses both models was benchmarked on JUWELS – one of the world's top supercomputers, and the achieved speedup was around 10×. After that, to remove code duplicates and to make the code easier to understand and maintain, JURASSIC-scatter and JURASSIC-GPU were merged into the new unified code project named JURASSIC-unified. This unified implementation has even better performance – it is around 24× faster then the initial JURASSIC-scatter implementation. In addition to a new feature, the JURASSIC-unified project was adapted so that even the JURASSIC reference project can use it as a library.

Životopis

Rođen sam u Čakovcu, 3. prosinca 1997. godine. Završio sam Osnovnu školu Belica, a 2012. upisao Gimnaziju Josipa Slavenskog Čakovec. Preddiplomski sveučilišni studij matematike na Prirodoslovno-matematičkom fakultetu Sveučilišta u Zagrebu upisao sam 2016., a završio 2019. godine. Iste godine upisao sam diplomski studij *Računarstvo i matematika*, također na Prirodoslovno-matematičkom fakultetu. Tijekom studiranja sam tri puta sudjelovao na Srednjoeuropskom studentskom ICPC natjecanju (CERC) i pet godina bio jedan od sastavljača zadataka na Državnom natjecanju iz informatike. Za izuzetan uspjeh na diplomskom studiju dodijeljene su mi Nagrada Matematičkog odsjeka i Dekanova nagrada. Od kolovoza do listopada 2021. sudjelovao sam na ljetnoj školi koju je organizirao institut Forschungszentrum Jülich – Jülich Supercomputing Centre, jedan od najvećih HPC centara u Europi, u suradnji s kojim je napravljen i ovaj diplomski rad.