This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2918798, IEEE Transactions on Parallel and Distributed Systems

Accelerating Atmospheric Chemical Kinetics for Climate Simulations

Michail Alvanos and Theodoros Christoudias

Abstract—The study of atmospheric chemistry-climate interactions is one of today's great computational challenges. Advances in the architecture of Graphics Processing Units (GPUs) in both raw computational power and memory bandwidth sparked the interest for General-Purpose computing on graphics accelerators in scientific applications. However, the introduction of GPUs in the High Performance Computing (HPC) landscape increased the complexity of software development, due to the inherent heterogeneity requirements of programming models and design approaches, creating a gap in uptake and attainable performance in the presently available scientific community codes. This paper provides an overview of the challenges encountered when using GPU accelerators to achieve optimal performance to calculate the kinetics of chemical tracers in climate models, the techniques used to address them and the insights gained from the process. The paper presents the development of a chemical kinetics code-to-code parser to automatically generate chemical kinetics calculations on three different generations of GPU accelerators (M2070, K80, and P100). The accelerated portion of the application achieves a speedup of up to 22×, equivalent to performance gains of +19% up to +90% compared with the processor-only version, when using a cluster of 8 Nodes with dual Intel E5-2680 v3 processor and a Kepler architecture (K80), allowing faster completion of the simulations. The paper also provides practical insights and relevant considerations for the development and acceleration of complex applications.

Index Terms—High Performance Computing, Heterogeneous systems, GPU acceleration, Climate Simulation

1 Introduction

The study of atmospheric chemistry-climate interactions represents an important and, at the same time, a demanding task of global Earth system modelling. Advancements in hardware architectures can greatly improve not only the spatial and temporal resolution of models but also the representation of key processes. With the expected advent of Exascale supercomputers, Earth System Models can achieve even higher resolutions and provide more accurate climate projections. Preparing the models and associated tools for the current and future High Performance Computing (HPC) architectures is a great challenge for the scientific community.

Researchers use Chemistry-Climate Models (CCMs) to understand the scientific and policy perspectives emerging from climate change and ambient air quality. Chemical kinetics is the study of chemical reactions, including their rates, the effect of relevant variables and formation of intermediates. In CCM simulations the numerical integration by chemical kinetics solvers can take up to 90% of execution time [1], prohibiting the use of complex chemical mechanisms in high-resolution simulations. Tackling this computational challenge can have far-reaching applications as the underlying chemical kinetics find use in different scientific fields, including biological systems chemistry, stratospheric ozone destruction, food decomposition, and combustion.

Current and future HPC technology development imposes new challenges and constraints. New programming models, such as the accelerator extensions of OpenMP [2] promise better portability and better performance compared with previous ad hoc approaches. The advances in the architecture of Graphics Processing Units (GPUs) in both raw computational power and memory bandwidth sparked the interest in General-purpose computing on GPUs for scientific applications. However, the introduction of GPUs in the HPC landscape increased the complexity of software development, due to the inherent heterogeneity in programming models and design approaches, creating a gap in uptake and limiting attainable performance in presently available community codes [3]. Programming a GPU accelerator can be a demanding and error-prone process that requires specially designed programming models, such as CUDA [4] and OpenCL [5].

This paper describes a method for accelerating, using GPUs, the Ordinary Differential Equations (ODE) solvers that represent the chemical processes in the climate models (chemical kinetics calculations). The paper also provides insights into the required effort and common pitfalls encountered in the process. The method is applied to the global Atmosphere-Chemistry model ECHAM/MESSy (EMAC) that includes sub-models describing the tropospheric and middle atmosphere processes and their interaction with oceans, land, and human influences [6]. Component contributions of this work include:

- A source-to-source parser that processes the chemical kinetics solvers and produces CUDA accelerated code to utilize the available GPUs in modern HPC facilities.
- A thorough quantitative performance study with a comparison of the accelerated and non-accelerated application. The experimental evaluation with the EMAC model indicates that the use of GPU accel-

• *Michail Alvanos and Theodoros Christoudias are with The Cyprus Institute, 20 Konstantinou Kavafi Street, 2121, Aglantzia, Cyprus*
E-mail: malvanos@gmail.com, christoudias@cyi.ac.cy

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2918798, IEEE Transactions on Parallel and Distributed Systems

erators can reduce the overall application runtime between 17% up to 50% under realistic scenarios.
- A list of our experiences, the common pitfalls, and our suggestions for accelerating climate models or developing future accelerated climate models.

The paper is organized as follows: Sec. 2 describes the EMAC/MECCA frameworks, and the performance of the application. Sec. 3 discusses previous work that is related to this research. Sec. 4 presents the design considerations for implementing the parser, the implementation of the parallelization, the GPU-specific optimizations, including memory optimizations, control code restructuring, and refactoring of the EMAC source code. An evaluation of the resulting GPU accelerated climate model appear in Sec. 5. Sec. 6 presents our experiences from porting the application. We summarize the main outcomes, present our conclusions, and future plans in Sec. 7.

2 Background

2.1 Climate models

Climate models use mathematical equations to describe and simulate the interactions of the important 'drivers' of the climate system: the atmosphere, oceans, land surface, ice, and solar radiation. Developing climate applications with high resolution that efficiently utilize the newest high-performance architectures will allow for skilled global and regional climate prediction. Accurate knowledge of the future climate will, in turn, provide valuable tools to enable informed policy planning for climate change adaptation and impacts mitigation. Although current HPC architectures can offer a tremendous amount of raw computational power, climate models are struggling to achieve the required scalability and resolution to accurately predict the future climate and thus render important benefits to the society.

2.2 The EMAC framework

The numerical global atmosphere-chemistry model EMAC (ECHAM/MESSy Atmospheric Chemistry) is a modular global model system that simulates the chemistry and dynamics of the troposphere and stratosphere. The model system includes modules that describe atmospheric processes and their interaction with oceans, land and human influences. The chemical kinetics module simulates the calculation of atmospheric chemical species concentrations and their interaction with radiation, the land and ocean.

For parallelization, EMAC uses the Message Passing Interface (MPI) standard to split the workload horizontally between processes in a rectangular decomposition along the latitudinal and longitudinal dimensions. The EMAC atmospheric dynamical circulation component only scales up to approximately a few hundred cores [1], due to i) the poor strong scaling efficiency of the ECHAM5 dynamical core, and ii) the workload imbalance created by the different modules.

2.3 Chemical Kinetics

Atmospheric chemistry investigates the chemical interaction between elements in the Earth's atmosphere. The underlying chemical kinetics find application in different scientific fields including climate prediction, meteorology, physics, oceanography, and geology. The EMAC chemical kinetics module employs the Kinetic Pre-processor (KPP) [7] open source general analysis tool to produce the chemical mechanism. The input to KPP comprises a set of chemical reactions and their rate coefficients, and the produced output is either FORTRAN or C code that integrates the time evolution of chemical species. EMAC uses a modified version of the KPP with additional pre-processing components to optimise performance specifically for climate simulations (known as KP4).

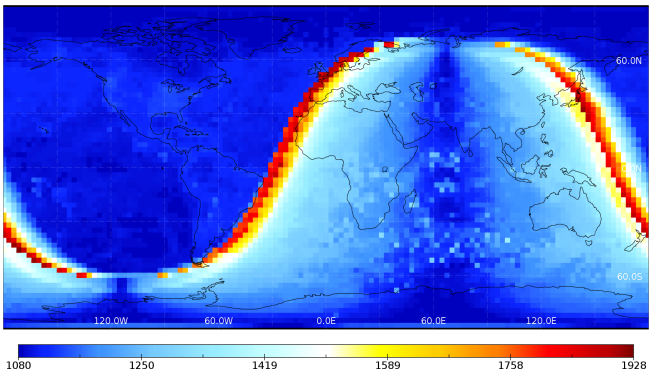


Fig. 1. Number of integration steps for each column during chemical kinetics calculation.

2.4 Performance

The chemical kinetics calculations constitute one of the most severe sources of imbalance between parallel processing elements. Fig. 1 presents the cumulative number of execution steps required for the integration process in each model vertical column in a typical configuration. The adaptive time-step integrator exhibits a non-uniform runtime caused by photochemistry during varying light intensity at sunrise and sunset. In addition, natural and anthropogenic emissions also cause imbalance between different cells inside each column and between columns and MPI processes [1]. The application uses barriers to synchronize all the MPI processes at each model time step. This introduces idle time in the fastest processes while waiting for the slowest process to finish, wasting resources and elongating time to completion.

For a typical climate model chemical mechanism with 155 species and 310 chemical reactions, the chemical kinetics calculations take ∼70% of the execution time [1]. Fig. 2 shows the time breakdown of the EMAC application for up to 192 processes using a dual socket system equipped with two 12-core Intel Xeon E5-2680 v3 CPUs. The convection, advection, and scavenging components are responsible for the atmospheric circulation and dynamics of the climate model. Chemical kinetics account for 73% of the total time in serial configuration and 30% when running with 192 processes. Communication burdens the performance over a large number of processors. Most of the time spent in the kernel module is due to the processing by network driver of the small messages responsible for the EMAC communication.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2918798, IEEE Transactions on Parallel and Distributed Systems

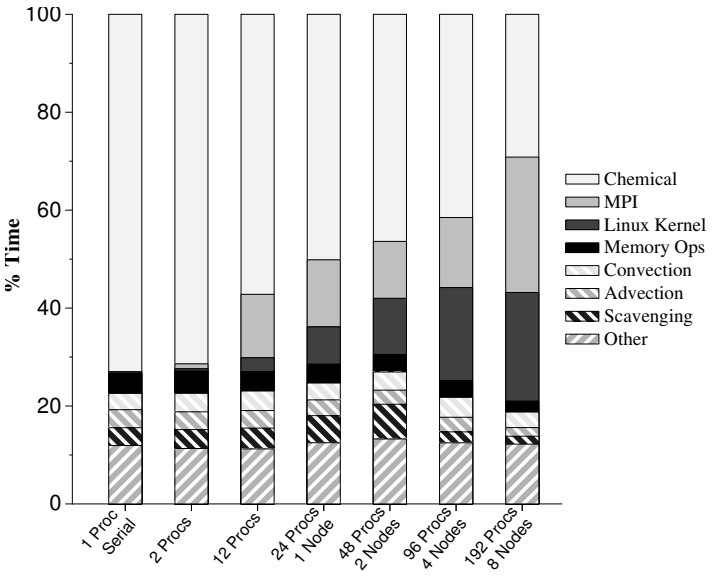


Fig. 2. Strong scaling execution time breakdown of the CPU-only EMAC application using chemical kinetics. The Convection, Advection, and Scavenging modules are responsible for the physics and dynamics of the climate model.

3 Related Work

Numerous efforts are documented in the literature on how to improve the performance of the climate-chemistry model simulations, specifically targeting chemical kinetics.

Initial efforts to improve the performance involved the introduction of sparse linear algebra optimisations [8], [9]. Only a fraction of the kinetics matrices is non-zero during the integration process of chemical kinetics, minimizing the number of necessary operations. The sparsity structure depends only on the chemical mechanism selected at compilation and remains unchanged during runtime. This allows optimization of the code by unrolling loops and removing indirect memory access.

A common approach to improve the performance is to parallelize the computation of chemical kinetics at the grid-point level [10], [11]. Source code modifications with the addition of OpenMP [2] directives exploit hybrid parallelism. The application assigns cells to individual threads to allow embarrassingly parallel execution of the chemical solvers, as there are no inter-dependencies in chemical kinetic calculations during a simulation time-step. The biggest drawback of this approach is the workload imbalance created by photochemical processes in the lower stratosphere [12].

Commercial approaches using accelerated fine-grained parallelism are also available [13]. However, these approaches exhibit limited parallelism and strong imbalance. Moreover, they are suitable only when the number of elements and chemical reactions are sufficiently large, beyond the chemical mechanism complexity of climate models.

Researchers use GPU accelerators to improve the performance of Earth system models. The ultra-high resolution global atmospheric circulation model NICAM [14] uses GPU accelerators for the shallow water computations (in the Dynamics module) [15], [16]. Components of the community Weather Research and Forecasting Model (WRF) were accelerated with GPUs. These include parts of the dynamics [17] and the radiation modules [18]. The ACME-Atmosphere model [19], [20] offloads the dynamics of the weather prediction to GPUs by refactoring the code using the CUDA programming model and OpenACC directives. Both of these approaches require extensive data layout and loop restructuring in order to obtain reasonable performance. Researchers have also accelerated the Non-Hydrostatic Weather Model ASUCA [15], GEOS-5 [21], and GRAPES [22] by using the CUDA programming model.

The Non-Hydrostatic Icosahedral Model (NIM) [23], [24] is designed and developed for massively parallel processors, using OpenACC directives to accelerate the code. The COSMO limited-area numerical weather prediction and the climate model were ported on GPU accelerators [25], [26]. The researchers refactored the code to use OpenACC compiler directives for most of the modules. These approaches require OpenACC support from the compiler that is not available in all compiler tool-chains used in High Performance Computing facilities.

Researchers rewrote the dynamics module of the COSMO limited-area numerical weather in CUDA, using the STELLA [27] domain-specific language for stencil codes on structured grids. Search and optimization techniques can also auto-tune 3D stencil (nearest-neighbor) computations on GPUs [28]. However, none of these implementations include consideration of atmospheric chemistry, or the acceleration of chemical kinetics components. Our study concentrates on solving the stiff ordinary differential equation (ODE) system describing atmospheric chemistry, with no nearest-neighbour computation that is suited to stencil optimization techniques.

4 Accelerating Chemical Kinetics

This section presents the design considerations, the implementation of the source-to-source parser, including an overview of performance optimizations, and the challenges addressed during the development.

4.1 Design Considerations

The implementation targets HPC machines equipped with Nvidia CUDA compatible accelerators. We decided to use the CUDA programming model [4] because the vendor provides strong debugging support and improved performance when using the CUDA compiler framework compared to other programming models, such as OpenCL [29].

As discussed in section 2, chemical mechanisms list the chemical reactions and their rate coefficients as input in domain-specific language. A modified KPP [7] utility (KP4) generates the FORTRAN code that computes the time evolution of chemical species from the mechanism. Subsequently, the code produced is compiled and linked with the EMAC model, with calls in the model time integration loop. To accelerate the produced FORTRAN code, a developer can use four different approaches, as presented in table 1. We took into consideration the advantages and disadvantages of each methodology regarding a cost-effective development with long-term maintenance sustainability, while at the same time being end-user friendly to ease on-boarding and promote adoption.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2918798, IEEE Transactions on Parallel and Distributed Systems

Methodology	Advantages	Disadvantages
Modify KP4 to directly produce CUDA code.	Covers all potential chemical mechanisms.	Requires long-term commitment from developers to keep KP4 up-to-date.
Transform FORTRAN to CUDA through compiler framework (e.g. ROSE).	Robust method, does not require modifying the EMAC source code [30].	Requires additional software packages. License issues.
Accelerate the chemical calculations with FORTRAN directives (e.g. OpenCL) or novel programming model (e.g. dCUDA).	Requires limited effort [31].	Performance can be limited [28], [32]. Not all HPC sites offer library support.
Use intermediate code to refactor the machine-produced FORTRAN code to CUDA.	Portable in many HPC environments. Requires minimal effort to add features.	Output code may require modifications for compilation.

TABLE 1
Advantages and Disadvantages of each method for accelerating the code.

The KP4, a specialized version of KPP for EMAC, requires extensive modifications for producing CUDA compatible code. In addition, source code modifications in a production-ready climate model require extensive testing and review by the maintainers, hindering rapid development cycles. Thus, we decided to create a parser that a user can run, before compiling the application, to minimize changes in the EMAC application source code repository.

Compiler frameworks and programming models, such as ROSE [30] and dCUDA [31], can improve the robustness of the output code and give the opportunity for further transformations to improve the performance. However, compiler frameworks require additional software package dependencies that are not always available in HPC environments. Thus, we decided not to use any specialized compiler frameworks to keep the implementation simple and portable. It is important for this research community application to provide a simple utility for acceleration without requiring the distribution of additional software packages with different and often proprietary licenses.

An alternative method, used for different accelerators [1], was the insertion of directives. This approach requires minimum effort and the produced code is relatively easy to maintain. However, this approach would still require the user to modify the source code in order to successfully run the application and achieve substantial performance benefits. Programming models using directives, such as the OpenACC [33], [34], [35], are helpful in simple configurations that do not require significant source code refactoring, but can be limiting regarding achievable performance in complex codes. Not all available compilers support the directives for acceleration and they are not mature enough, in terms of performance, to be considered in a production environment.

Our approach uses a source-to-source parser written in Python, to transform the FORTRAN-produced code to CUDA-accelerated. This approach keeps the implementation tailored but simple, and (mostly) independent from the changes in the KP4 or EMAC application. It does not introduce new library requirements, and allows to select an open software license that is still compatible with the EMAC application end-user license agreement. The main drawback is that the tool may sometimes, in the case of complex chemistry, produce non-compiling source code, which requires additional modifications by the user. Finally, even though the KP4 utility is mature and its source code stable, the parser is still susceptible to potential significant changes in the future.

4.2 Implementation

The code is distributed with the EMAC model as a utility to transform the code after generating the chemistry and before compiling. The parser modifies the KPP Fortran source file and places a single call to the CUDA object that contains a wrapper for issuing the accelerator kernels and the accelerator code. Alvanos et al. describe in detail the source-to-source parser software implementation [36].

The parser first strips the FORTRAN code of unnecessary elements such as comments and translates statements and variables into C equivalent code. It then constructs the required GPU data structures subdividing in runtime-specified arrays of columns in the atmosphere, with the memory of each array transferred to the GPU global memory. Temporary arrays are allocated in the GPU stack memory to store the intermediate results of the solvers.

Each grid box is calculated on a separate GPU core to achieve massive parallelization using three CUDA kernels constructed from predefined prototypes [37], [38]:

1) The first kernel calculates of the reaction rate coefficients using the variable values are stored in a global array inside the GPU .
2) The second, most computationally demanding kernel, includes all linear algebra operations of the ODE solvers. The parser injects the required variation of the Rosenbrock solver method from a set of integrator CUDA prototypes.
3) The third kernel performs statistical reduction, and demands limited computational time compared with other kernels.

The produced CUDA code supports all variations of Rosenbrock solvers included in the original KPP code, and transfers the array containing the cells and the input data required by the kernels.

Finally, the parser modifies the produced KPP file that contains the FORTRAN code responsible for the chemical kinetics calculations in EMAC, by adding a function for issuing the parallel GPU kernels and copying the data to and fro the GPU memory. The parser also makes the appropriate changes in the makefile to link the CUDA objects in the EMAC binary code.

The biggest challenge of accelerating the source code was the slowdown of the application due to large amounts of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2918798, IEEE Transactions on Parallel and Distributed Systems

memory bandwidth required. Each accelerated CPU process requires a chunk of the GPU VRAM memory, whose size is dependent on the number of species and reaction constants in the MECCA chemical mechanism. For instance, a set of 155 species and 310 reactions require the allocation of ∼50KB for intermediate results. On the GPU this is assigned to the global memory because the on-core memory is not enough. Overall, the effort associated with optimising the performance of the GPU kernel was an order of magnitude greater than porting the code to CUDA.

4.3 Optimizations

The complexity of the solver and the memory requirements per GPU thread are the main factors limiting performance. To improve the performance of the application, we gradually integrated GPU optimizations, while evaluating the results with performance counters. The initial code transformation to CUDA revealed significant shortcomings in the execution efficiency of the code. For example, the stall data requests were found to be responsible for 63% of total stall reasons.

To simplify the optimization improvement analysis, we organize the transformations into three groups: i) Occupancy improvement, ii) Memory improvements, and iii) Source code simplification. As elaborated in section 5, the impact of each optimization depends on the underlying architecture of the GPU accelerator.

Occupancy improvement: The first improvement is to ameliorate the occupancy of the GPU, the number of concurrent threads per Streaming Multiprocessor (SM). The programmer can either specify the number of registers at compile time, using the proper flag (`-maxrregcount=xxx`), or specify the launch bounds of the kernels inside the source code (`__launch_bounds__`). In any case, the performance gain from limiting the number of registers depends on the GPU architecture. Limiting the register usage has no or very limited impact in the Kepler architecture (K80). On the other hand, limiting of the registers in the Pascal architecture had a negative impact on the performance. Thus, the preprocessor asks the user to select the architecture during the compilation time, and based on the selection, it enables or disables the `-maxrregcount=128` compiler flag.

Memory optimizations: To solve the ODEs and calculate species concentrations, more memory is required in total than the on-chip memory available. To achieve the best execution efficiency on the GPU, we use a number of memory optimizations in the generated code: i) better utilization of the Level-1 (L1) on-chip memory; ii) privatization of data structures; and iii) prefetching. Due to high register usage, the kernel will use stack memory (local memory) for register spilling. In our tests, the NVCC compiler reports 37520 bytes spill loads for the 3-step Rosenbrock solver. In the Kepler architecture, each Streaming Multiprocessor (SMX) has 64 KB of on-chip memory that can be configured as 48 KB of Shared memory with 16 KB of L1 cache, or vice versa. The size of required temporary matrices is always larger than the available shared memory, necessitating the use of global and local memory.

To increase the utilization of local memory, we increased the portion of the L1 cache against the shared memory to 48KB per SMX, through the `cudaFuncCachePreferL1` runtime call. This resulted in a 5-10% decrease in the number of local memory transactions directed to the global memory, depending on the solver used. In the Pascal architecture, the local L1 memory is decoupled from the shared memory. Thus, the aforementioned runtime call does not impact the performance of the kernel execution. Additionally, we privatized the majority of the matrices by either replacing them with scalar variables or by using stack-allocated arrays, allowing simplified temporary array indexing code. Finally, the kernels employ prefetching of data to reduce the latency. Prefetching may have unpredictable behavior, especially in massively parallel architectures, such as GPUs. The preprocessor enables or disables prefetching based on the selected architecture.

Control code simplification: The GPU cores achieve the best performance when they execute simple instructions with the absence of control code. To increase the instruction-level parallelism, the source code uses three commonly used techniques: i) pre-calculated lookup tables; ii) loop unrolling; and iii) branch elimination. Lookup tables are used for selecting the solver and setting the proper values in specific variables for the available solvers. The benefits of loop unrolling are most profound in the preparation of the solvers due to the sparsity of the Jacobian matrices. Limited branch elimination by fusing loops or merging branches also improves execution time.

4.4 GPU Workload Management

The biggest challenge when using accelerators in hybrid HPC architectures is the imbalance created by the uneven workload of tasks. While the application uses GPUs as accelerators, some CPU cores are responsible only for communication and handling of GPUs, while the remaining cores are idling. Allocating more processes on the unused CPU cores, creates a greater imbalance between the accelerated processes and the CPU-only processes. The are two ways to do this: i) over-subscription by allowing more than two CPU cores (MPI processes) from a single node to offload to the accelerator, or ii) using the *Multi-Process Service (MPS)* [39]. Over-subscription can produce some performance gains, but it is not beneficial overall as processes that can under-utilize the available hardware. The number of GPUs per node and VRAM memory available in each GPU dictates the total number of CPU cores that can run simultaneously. The alternative approach of using the *MPS* allows concurrent execution of kernels, and memory transfers from different processes on each node. The latter requires the use of a GPU accelerator with computation capability 3.5 or higher. The *MPS* acts as a proxy for the different processes of the accelerated application to enable transparent sharing of the available accelerator(s) in order to minimize the idle time.

5 Experimental Evaluation

The following experimental evaluation assesses: (i) the effectiveness of the code optimizations; (ii) the impact of accelerator use on the performance of the application; and (iii) bottlenecks and limitations. Two different hardware and software environments are used to measure the impact of the code acceleration (Table 2):

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2918798, IEEE Transactions on Parallel and Distributed Systems

- An IBM® S822LC compute node equipped with two 10-core 2.92 GHz POWER8 processors [40], [41] with turbo up to 4 Ghz. Simultaneous multithreading is set to 4 for optimum performance in HPC applications. The computation node is equipped with a P100 GPU accelerator (Pascal Architecture). The application is compiled using the IBM compiler, `xlf` ver. 15.1.5.
- Compute nodes of the Jureca machine at the Jülich Supercomputing Centre. Each compute node is equipped with two 12-core Xeon®E5-2680 v3 CPUs with 128 GBytes of RAM. The GPU nodes contain two NVIDIA®K80 GPUs (Kepler Architecture). The application is compiled using the Intel compiler (`ifort` ver. 17.0.2) for improved native performance.

To evaluate the model, we use a representative simulation with a horizontal resolution of 128 grid points in the longitudinal direction and 64 grid points in the latitudinal direction, with 90 vertical levels totaling 737280 Grid points. Table 3 details the experimental setup for the results shown in this section. The default E5M2 KPP chemistry batch option is used, along with model namelist setup `NML_SETUP=E5M2/02b_qctm`, without dynamics nudging and with all diagnostic submodels switched off.

On the Intel platform, the additional flag `--ntasks-per-node=24` is used, and `--cpus-per-task=2` is used on the Jureca GPU node to ensure that a physical CPU core is assigned to each process. The execution of `mpirun` on the PowerPC machine specifies `--map-by L2cache --bind-to core:overload-allowed` to reduce the contention of the cache and function units.

5.1 Impact of Optimizations

The biggest programming effort was related to the development of the application and incremental performance improvements, achieved with the integration of various optimizations. The results for this section are based on the extracted chemical kinetics kernel, used as a microbenchmark to demonstrate the effectiveness of the code transformations. The microbenchmark uses as input the chemical species concentrations extracted from the execution of one time-step of the application. The microbenchmark is performed with five variations of the Rosenbrock numerical solver on the three different GPU accelerator types. Table 4 presents the mean speedup for each optimization set.

Occupancy optimization by either using the `-maxrregcount=xxx` flag, or specifying the launch bounds of the kernels inside the source code (`__launch_bounds__`) gives mixed results. In particular, the K80 accelerator performs worse, and the P100 shows no measurable difference. The number of registers or launch bounds depends on the source code characteristics and is selected empirically. In our implementation the flag is set to 128 registers for CUDA 2.0 architecture and the source-to-source parser inserts the flags in the appropriate generated files.

Changing the cache configuration to use more Level-1 (L1) cache instead of shared memory (`cudaFuncCachePreferL1`), shows little performance improvement on the K80. P100 contains separate L1 and shared memory, nullifying any effort to better allocate the on-chip memory. Privatization of the data significantly improves performance on all platforms. The impact is greater on older GPU architectures that have simpler on-chip memory organization. The most notable negative impact (-18%) in the performance occurs with the usage of the prefetch intrinsics inside the source code for the P100 accelerator. Newer accelerators contain advanced hardware prefetchers that decrease the impact of manual memory optimization. Thus, we are forced to disable the software prefetcher specifically for the P100 platform. On the other hand, the privatization of global arrays using shared memory, registers, and stack allocated arrays gives the biggest benefit by reducing off-chip memory traffic.

5.2 Speedup and Scalability Analysis

To gauge the acceleration performance, we compare the measured GPU speedup with the maximum theoretical speedup S, given by Amdahl's law [43]:

$$S = \frac{1}{1-f}$$

where f is the is the proportion of execution time that the part benefiting from GPU acceleration originally occupied, ranging from 73% for a single process to 29% for 192 CPU processes. Table 5 presents the execution time of the EMAC application and the maximum theoretical and GPU-measured performance speedup achieved for different configurations. The measured mean ratio of GPU-achieved to maximum theoretical S performance ranges between 79–84%, showing stable scaling performance of the accelerated component.

Increasing the number of cells per processes or the complexity of the solver increases the computation time and the potential speedup, as shown in Table 6. However, the application absolute speedup is still limited due to the fine-grained communication even if we increase the workload per GPU by a factor four. We emphasize that simulations with high resolution require not only greater computational power but also memory. Thus, the poor scalability of the application and the memory requirements both limit the resolution of the climate simulations.

Figure 3 presents the achieved speedup of the application compared with the serial implementation, without MPI support and without accelerators. Enabling the communication subroutines and using two processor cores gives only a 1.06× speedup, revealing the complexity of the communication mechanisms and the imbalance created between the two processes. Moreover, the accelerated version achieves better scalability, as it uses two accelerators in each compute node. Acceleration allows the application to reach the efficiency limit faster than the CPU version. For example, the performance with two CPU-only nodes is similar to the accelerated version running in one node. In addition, the computation partitioning of the application does not allow runs with larger than 192 processes. Thus, acceleration allows the application to reach the scalability limit faster than the CPU-only version.

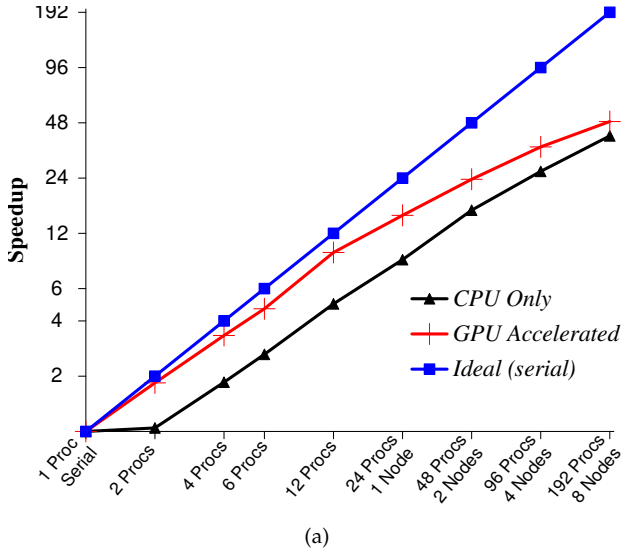
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2918798, IEEE Transactions on Parallel and Distributed Systems

Node	CPU	Cores #	RAM GB	Accelerators NVIDIA GPU	Cores	VRAM	Bandwidth	Peak Performance GFlops (DP)
Jureca GPU Node	E5-2680 v3	24	128	2× K80	2×4992	4×12 GB	2 × 240 GB/s	4208: 480 CPU + 2×1864 GPU
IBM® S822LC	POWER8	20	256	4× P100	4×3584	4×16 GB	720 GB/s	21667: 467 CPU + 4×5300 GPU

TABLE 2
Hardware configurations used for performance evaluation.

Number of columns	lat x lon = 64 x 128 = 8192
Vertical atm. levels	90 levels (L90MA)
Total grid points	737280 grid points
Chemical mechanism	155 species, 310 chemical reactions
Spectral truncation	T42
ODE Solver	3-stage, L-stable pair of order 3(2) [42]

TABLE 3
Experimental configuration for model evaluation simulations.



(a)

Fig. 3. Scalability test of the EMAC application.

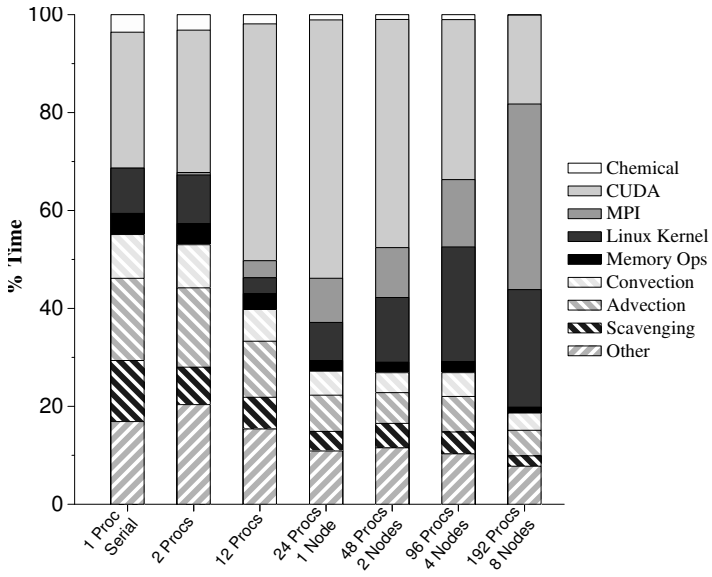


Fig. 4. Strong scaling execution time breakdown of the accelerated EMAC application using chemical kinetics.

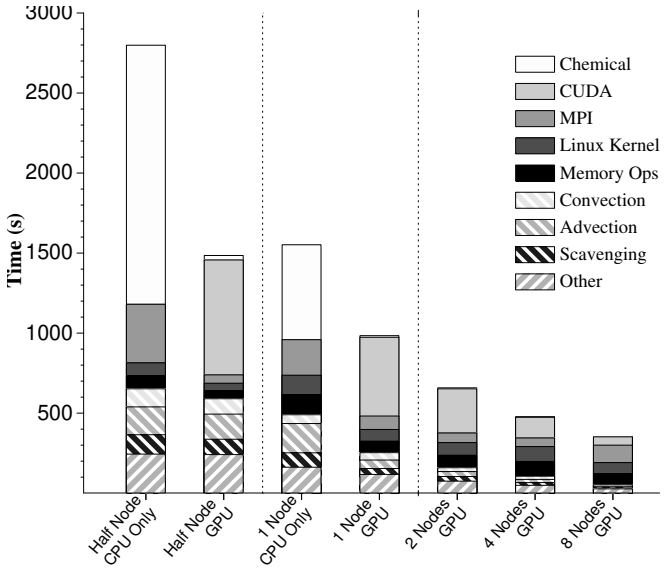


Fig. 5. Execution time of the accelerated EMAC application using chemical kinetics compared with the single-node CPU-only version.

5.3 Time allocation

Figure 4 presents the breakdown of the relative execution time for the accelerated version of the application. The modules of Convection, Advection, and Scavenging are responsible for different phases of the application and have not been accelerated.

Communication burdens the performance on a large number of MPI processes. This happens due to the exchange communication pattern of the application from the ECHAM model framework. The MPI portion of the execution time increases from 54 seconds in eight nodes to 108 seconds when running with 16 nodes as Figure 5 shows. The experiment uses a fixed total problem size (strong scaling) that does not allow to scale to more than 192 MPI processes.

Moreover, the kernel overhead increases from 48 seconds in one node to 92 seconds in eight nodes. We tracked down the Linux kernel overhead to the network driver and its locking mechanisms, which is an indication of inefficient fine-grained communication due to the large number of small messages. The portion of wall-time dedicated to chemical kinetics is decreasing faster than in the CPU only version. For example, the application assigns 3840 number of cells per process when running with 192 processes, increasing the overhead of processing the model.

5.4 Summary

The evaluation indicates that platform agnostic GPU kernel optimization is very challenging. Our implementation uses compile-time flags to select the best options and code paths

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2918798, IEEE Transactions on Parallel and Distributed Systems

Accelerator	Occupancy	Prefer L1	Privatization	SW Prefetch	Simplification	Total Speedup
Mean Speedup K80	-7.9% ± 0.39	+1.25% ± 0.21	+15.12% ± 2.77	+3.15% ± 1.81	+0.21% ± 0.19	+17.12% ± 3.78
Mean Speedup P100	+0.06% ± 1.09	-0.04% ± 2.81	+22.48% ± 3.87	-18.14% ± 2.03	+9.62% ± 3.30	+40.57% ± 3.04

TABLE 4
Mean speedup of the kernel solvers using K80, and P100 CUDA enabled accelerators using synthetic input. Software Prefetch optimization is disabled when calculating the total speedup on the P100 platform.

Processes	CPU Execution Total Time (s)	Accelerated Total Time (s)	Accelerated Proportion f	Measured Speedup	Max. Theoretical Speedup S	Achieved Performance
12	2829	1485	57.2%	1.90×	2.33×	81.5%
24	1625	932	50.1%	1.74×	2.00×	87.0%
48	874	593	46.4%	1.47×	1.86×	79.0%
96	537	395	41.5%	1.35×	1.71×	78.9%
192	344	287	29.2%	1.19×	1.41×	84.4%

TABLE 5
Execution time, excluding initialization, of the EMAC application for 24 hours simulation time, measured and maximum theoretical speed-up, and achieved (measured over theoretical speedup) performance fraction.

Grid Resolution	CPU (s)	Accel (s)	Cells/Proc.	Speedup
8192 Columns	344	287	3840	1.19×
32768 Columns	1238	1007	46080	1.23×

TABLE 6
Execution time of the EMAC application for 24 hours simulation time using two different grid resolutions.

for optimal performance. The performance gain can reach up to +40% using the latest Pascal architecture. Platform-specific optimizations can ultimately determine whether the application will obtain any benefit from acceleration. Our scalability tests show that our implementation achieves performance ranges between 79–84% of the expected performance gain by accelerating the application, leaving some room for improvement. GPU acceleration allows to reach the scalability limits more efficiently than the CPU-only version, due to reduced communication.

6 Experiences

This section presents an overview of our collected experiences and gives suggestions to guide future efforts to port complex applications, such as climate models. The porting to GPUs and optimizing the chemical kinetics mechanisms for Atmosphere-Chemistry modelling was particularly challenging, requiring a generic solution and not a specific platform or accelerator.

6.1 Optimizations

Currently, there are many available GPU architectures, and code optimizations applicable on current architectures do not guarantee efficient execution in future architectures. Table 7 presents a list of the available CUDA-enabled architectures. The memory hierarchy and the cache size evolve with each architecture iteration. On the Kepler architecture a maximum of only 48 KB L1 cache can be used. The Pascal architecture in practice transforms the L1 cache to 24 KB of Read-Only memory and adds 64 KB of dedicated shared memory. The introduction of read-only cache in Kepler architecture can improve the performance in certain scenarios.

Fine-tuning for a specific GPU architecture does not guarantee future-proof performance. Prefetching has limited impact on the Kepler architecture. The performance on Pascal architectures is actually made worse, due to the interference of the software prefetching mechanism with the hardware prefetchers, as presented earlier in Table 4. Moreover, there are features that are not available on all architectures. The most notable example is the uncached load that can have significant impact on performance in the latest architectures. Thus, the developers have to be careful to create platform agnostic optimizations that run in all architectures or develop specialized code for each GPU architecture. Even if the use of future architectures will likely not require significant changes in the source code, the downside of the generic approach is that the application will not achieve the maximum possible performance.

Our experience shows that the memory access and simplification optimizations have the greatest impact, but the developer should also explore novel ideas for improving performance. For example, in contrast with the common approach that all accesses must be in parallel in global memory, the performance of the application improved by moving the temporary arrays locally, using registers and stack memory. In this case, the high usage of internal cache memories due to register spilling, in combination with sparse matrices significantly reduce the performance of the traditional approach of parallel accesses.

The alternative approach is to create a specialized version of the GPU kernel for each architecture. The CUDA programming model allows, with the help of the preprocessor, to selectively enable or disable parts of the code. Thus, multiple versions of the same accelerated code are necessary in order to achieve the best possible performance. In our implementation, the source-to-source parser asks for the targeted architecture and makes the appropriate changes in the output source code. An additional generic version of the produced code is also provided, including only the parts that are platform-agnostic and provide performance benefits on all GPU accelerators.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2918798, IEEE Transactions on Parallel and Distributed Systems

Name	M2070	K80	P100	V100
Architecture	Fermi	Kepler	Pascal	Volta
Chip	GF100	2× GK210	GP100	GV100
Cores	448	4992	3584	5120
Shared Mem / L1	64 KB	64 KB	-	128 KB
De/ted Shared Mem	-	-	64KB	-
L1 / Rd-only Cache	-	-	24 KB	-
Rd-only cache	-	48 KB	-	-
L2 Cache Size	768 KB	1536 KB	4096 KB	6144 KB
DP GFlops	515	1864	4036	7500
VRAM (GB)	6	2×12	16	16
Bandwidth (GB/s)	144	2×240	720	900

TABLE 7
Available GPU architectures.

6.2 Compilation

The compiler framework of each architecture can play a significant role on the required programming effort and in the implementation of an accelerated application. Additional information regarding the usage and relationship between variables and arrays, can significantly improve the performance of the kernel. For instance, the compiler can create instructions of uncached memory accesses loads when the `restrict` and `const` keywords are used. The compiler uses these keywords in two ways. Firstly, to understand that the pointers cannot point to memory that overlaps and apply optimizations. Secondly, with the introduction of the CUDA 3.5 architecture, the compiler can automatically generate the `LDX` instructions that allow using the texture cache for read-only data. However, the compiler can not guarantee the creation of these instructions if there is missing compile-time information. Thus, it is advisable that the developer examines the output of the compiler when planning to use architecture specific characteristics or extensions.

The second challenge we encountered during development was the compiler memory allocation. The CUDA compiler framework is very demanding in CPU and memory resources, for both static and dynamic compilations. Large applications can result in long compilation times and large memory consumption. For example, in our scenarios with 2200 chemical reactions, the portion to be offloaded is in the order of 80K lines of code. The big compilation file resulted in more than 40 GB of memory required during compilation, and a lengthy compilation time. Furthermore, we were unable to create a version with 'debug' symbols, because the memory requirements were greater than 220 GB during compilation. It is thus important to take into account the compilation requirements in complex scientific applications with a large amount of offloaded code, as they may be a limiting factor. Finally, when running multiple processes per node, the very large amount of memory allocated by the just-in-time compilation can cause excessive memory swapping or even application crashes. The solution is to include the production of architecture-targeted byte code during compilation at the expense of additional compilation time.

6.3 Multi-Process Service

The multiprocess service is necessary for running in HPC machines where multiple CPU cores are available per GPU accelerator. It is supported on accelerators with virtual architecture 3.0 or newer. The developer must take into account the additional latency created when the application uses the MPS in multiple processes attached to a single accelerator. For example, the execution latency of kernels using the P100 GPU configuration increased from the order of 130 ms to 150–300 ms. The transfers to and from the host are responsible for the greatest decrease in performance. The timing increase is usually one or two orders of magnitude for small transfer sizes. On the Power 8 platform, the data transfer time of the application was $4\times$, mostly due to higher communication latency when processes were trying to use an accelerator simultaneously.

6.4 Load Imbalance

The simulation of chemical reactions amplifies the computational imbalance created by natural phenomena, such as sunset and sunrise, in the climate simulation models. The introduction of heterogeneity in HPC machines complicates scheduling and introduces an additional source of imbalance. Dynamically sharing of GPU accelerators using the Multi-Process Process reduces the latter imbalance and increases the overall utilization between the accelerated processes of the application. The scheduling efficiency in climate models can be further improved. For example, the changing solar radiation during the diurnal cycle can double the execution time of the kinetics numerical solver due to the stiffness of the differential equations. By calculating the solar irradiance a priori based on the time and spatial coordinates, the developer can opt for higher execution priority and/or allocation of additional resources. However, the discretization of past and present production climate model codes and the CUDA programming model limit the potential for using more advanced scheduling techniques. Future investigation and the advent of a new generation of Earth system models is necessary to address the Earth system multi-process and multi-scale simulation imbalance and to fully exploit the heterogeneity of modern HPC architectures.

6.5 Verification of the Results

The verification of climate model output is a complex research topic. Unit testing with proper code coverage and simulation tests over long periods of the order of years is necessary to ensure stability and correctness.

To test the numerical accuracy, we compare the relative difference between the CPU-only and GPU-accelerated codes after single time steps with identical initial conditions. The results show a median difference less than 0.000000001%, well within the 0.1% accuracy criterion for the chemical kinetics calculation [8].

The GPU-accelerated model remains stable when running over a 2-year simulation period and accurately reproduces the expected zonal and surface distributions of key species (Ozone, Hydroxyl radical, Ammonia, and Sulphur Dioxide) [36]. The median value of the CPU-GPU difference in aggregated mass of chemical species is less

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2918798, IEEE Transactions on Parallel and Distributed Systems

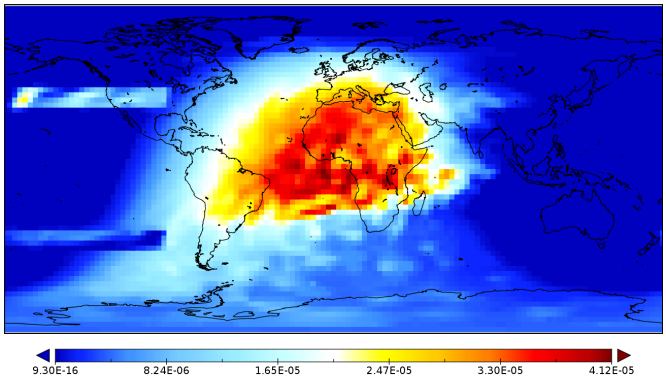


Fig. 6. Visualization of atomic oxygen concentration (mol/mol) during silent data corruption. Notice unphysical boxes on the left side of the domain.

than 5%. Inclusion of wet scavenging, to test the effect of boundary conditions, was shown to amplify the range of extreme values, with median values still falling within the 5% limit. This is within the margin observed by the climate modelling community stemming from differences between architectures and compilers (not specific to GPU).

Developers should be aware of possible silent data corruption. The memory allocation for chemical kinetics on the GPU varies depending on the stiffness of the solver. In the case of very stiff systems GPU contexts can crash due to excessive memory demands, returning from the execution of the solver without updating the chemical concentrations. Figure 6 shows an example of silent data corruption in a simulation. It is highly recommended that developers make diligent use of error-checking APIs, such as the one provided by the CUDA programming model, to ensure proper execution of GPU-offloaded code.

6.6 Acceptance from Users and Developers

The integration of GPU accelerators in modern HPC architectures drives a transformation of the software used by computational scientists. Our design and development choices aim towards minimal changes in legacy coding habits, and strive to not expose the model developers to the complexity and heterogeneity of new systems. However, we foresee that to effectively adopt novel technologies, either advanced external user support or embedding of specialised computational scientists in traditional scientific code development communities will be a necessity.

In the case of GPU chemical kinetics, during the development phase the response of the EMAC model community developers and users was mostly positive. The key requirements set before acceptance were twofold: to not introduce modifications in the development process and build procedure of the EMAC application, and to avoid additional external software dependencies. The parser is released and included in the EMAC distribution, thus no external dependencies are necessary. The decision to create an additional parser tool and decouple the GPU code from the main source code allowed the developers to independently proceed with development and licensing, and did not require additional external library requirements, other than the CUDA environment, which is in any case readily available in HPC centers with NVidia accelerators. It also does not in any way hinder the portability of the code in the HPC centres where the application is currently deployed. The only present obstacle for users to finally adopt and use the accelerated version is the additional step required before compiling the code during configuration.

The parser supports the batch chemical mechanisms that are bundled in the EMAC distribution out-of-the-box, covering the majority of use cases. Corner cases that fail are custom user-defined mechanisms of large size and complexity that include additional operations in the derivation of reaction rates. These cases are rare and involve users with advanced specilisation on atmopsheric chemistry. There is great interest from these users to benefit from this work as they have great computational demands for chemical kinetics and there is ongoing effort to jointly develop support.

7 Conclusions

This paper presents a method for accelerating atmospheric chemical kinetics calculations in climate models using GPU accelerators, and shares the experiences of the developers as well as insights from the process. The implementation can be used on three generations of accelerators (M2070, K80, and P100), by applying uniform code optimizations independently of the underlying architecture. The accelerated kernel achieves up to a factor of 22 speed-up over the same portion of the code running on CPU in real-world tests on HPC clusters. Accelerator optimizations achieve between +17% up to +40% performance gain over the basic porting of the code to GPU and the accelerated application achieves between 1.19× up to a 1.90× speed-up, allowing faster completion of the simulations. Finally, the paper presents an overview of our experiences regarding code optimization, compilation, use of the multi-process service, task imbalance, and verification of the results.

This paper presents a significant contribution to improving the performance of climate applications that rely on chemical kinetics simulations through GPU acceleration on heterogeneous architectures. To the best of our knowledge, it is the first time that such a significant performance improvement has been achieved using GPU accelerators in a real-world global climate modelling application. Beyond the GPU acceleration of chemical kinetics calculations, there is still room for further optimizations in Earth system modelling. It is expected that advanced dynamic scheduling of the tasks can reduce the load imbalance and increase the potential performance benefit.

8 Code Availability

The FORTRAN to CUDA code-to-code parser is developed using the C and Python programming languages and it is released and distributed with the EMAC model code. In addition, the source code is available in a public repository under an open license to allow contributions from the developer community [37], [38]. The code parses the auto-generated MECCA KPP solver code to produce a CUDA file that is compiled and linked with the EMAC/MECCA FORTRAN code.

9 Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under Grant Agreement No 287530 and from the European Union's Horizon 2020 research and innovation programme under grant agreements No 675121 and No 676629. This work was supported by the Cy-Tera Project, which is co-funded by the European Regional Development Fund and the Republic of Cyprus through the Research Promotion Foundation. Additional computational time was granted on the JURECA supercomputer at the Jülich Supercomputing Centre.

Any opinions, findings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] M. Christou, T. Christoudias, J. Morillo, D. Alvarez, and H. Merx, "Earth system modelling on system-level heterogeneous architectures: EMAC (version 2.42) on the Dynamical Exascale Entry Platform (DEEP)," *Geoscientific Model Development*, vol. 9, no. 9, p. 3483, 2016.

[2] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," in *IEEE International Conference on Computational Science and Engineering*, 1998.

[3] Nvidia, "Gpu accelerated applications, http://www.nvidia.com/object/gpu-applications.htm," 2017.

[4] ——, "Programming guide," 2015.

[5] A. Munshi, "The OpenCL specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009, pp. 1–314.

[6] P. Jöckel, A. Kerkweg, A. Pozzer, R. Sander, H. Tost, H. Riede, A. Baumgaertner, S. Gromov, and B. Kern, "Development cycle 2 of the modular earth submodel system (messy2)," *Geoscientific Model Development*, vol. 3, no. 2, pp. 717–752, 2010.

[7] V. Damian, A. Sandu, M. Damian, F. Potra, and G. R. Carmichael, "The kinetic preprocessor KPP-a software environment for solving chemical kinetics," *Computers & Chemical Engineering*, vol. 26, no. 11, pp. 1567–1579, 2002.

[8] H. Zhang, J. C. Linford, A. Sandu, and R. Sander, "Chemical Mechanism Solvers in Air Quality Models," *Atmosphere*, vol. 2, no. 3, pp. 510–532, 2011.

[9] M. Z. Jacobson and R. P. Turco, "SMVGEAR: A sparse-matrix, vectorized Gear code for atmospheric models," *Atmospheric Environment*, vol. 28, no. 2, pp. 273–284, 1994.

[10] J. C. Linford, J. Michalakes, M. Vachharajani, and A. Sandu, "Multi-core acceleration of chemical kinetics for simulation and prediction," in *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*. IEEE, 2009, pp. 1–11.

[11] J. C. Linford, "Accelerating atmospheric modeling through emerging multi-core technologies," Ph.D. dissertation, Virginia Tech, 2010.

[12] T. Christoudias and M. Alvanos, "Accelerated chemical kinetics in the EMAC chemistry-climate model," in *High Performance Computing & Simulation (HPCS), 2016 International Conference on*. IEEE, July 2016, pp. 886–889.

[13] V. Damian, A. Sandu, M. Damian, F. Potra, and G. R. Carmichael, "The kinetic preprocessor KPP-a software environment for solving chemical kinetics," *Computers & Chemical Engineering*, vol. 26, no. 11, pp. 1567–1579, 2002.

[14] M. Satoh, T. Matsuno, H. Tomita, H. Miura, T. Nasuno, and S.-i. Iga, "Nonhydrostatic icosahedral atmospheric model (nicam) for global cloud resolving simulations," *Journal of Computational Physics*, vol. 227, no. 7, pp. 3486–3514, 2008.

[15] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka, "An 80-fold speedup, 15.0 tflops full gpu acceleration of non-hydrostatic weather model asuca production code," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[16] I. Demeshko, N. Maruyama, H. Tomita, and S. Matsuoka, "Multi-gpu implementation of the nicam atmospheric model," in *European Conference on Parallel Processing*. Springer, 2012, pp. 175–184.

[17] J. Michalakes and M. Vachharajani, "Gpu acceleration of numerical weather prediction," *Parallel Processing Letters*, vol. 18, no. 04, pp. 531–548, 2008.

[18] J. Mielikainen, B. Huang, H.-L. A. Huang, and M. D. Goldberg, "Gpu acceleration of the updated goddard shortwave radiation scheme in the weather research and forecasting (wrf) model," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 5, no. 2, pp. 555–562, 2012.

[19] M. A. Taylor, "The ACME project's dycore performance strategy for next generation architectures." Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2015.

[20] M. R. Norman, A. Mametjanov, and M. Taylor, "Exascale programming approaches for the accelerated model for climate and energy," 2017.

[21] W. Putnam, "Graphics processing unit (GPU) acceleration of the Goddard Earth observing system atmospheric model," 2011.

[22] Z. Wang, X. Xu, N. Xiong, L. T. Yang, and W. Zhao, "Gpu acceleration for grapes meteorological model," in *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*. IEEE, 2011, pp. 365–372.

[23] M. W. Govett, J. Middlecoff, and T. Henderson, "Running the NIM next-generation weather model on GPUs," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2010, pp. 792–796.

[24] M. Govett, J. Middlecoff, and T. Henderson, "Directive-based parallelization of the NIM weather model for GPUs," in *Proceedings of the First Workshop on Accelerator Programming using Directives*. IEEE Press, 2014, pp. 55–61.

[25] O. Fuhrer, C. Osuna, X. Lapillonne, T. Gysi, M. Bianco, and T. Schulthess, "Towards GPU-accelerated Operational Weather Forecasting," in *The GPU Technology Conference*, 2013.

[26] O. Fuhrer, C. Osuna, X. Lapillonne, T. Gysi, B. Cumming, M. Bianco, A. Arteaga, and T. C. Schulthess, "Towards a performance portable, architecture agnostic implementation strategy for weather and climate models," *Supercomputing frontiers and innovations*, vol. 1, no. 1, pp. 45–62, 2014.

[27] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess, "Stella: A domain-specific tool for structured grid methods in weather and climate models," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 41:1–41:12.

[28] Y. Zhang and F. Mueller, "Autogeneration and autotuning of 3d stencil codes on homogeneous and heterogeneous gpu clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 3, pp. 417–427, 2013.

[29] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[30] D. Quinlan, C. Liao, J. Too, R. P. Matzke, and M. Schordan, "ROSE compiler infrastructure," 2012.

[31] T. Gysi, J. Baer, and T. Hoefler, "dCUDA: Hardware Supported Overlap of Computation and Communication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC16)*. IEEE Press, Nov. 2016, pp. 52:1–52:12.

[32] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 676–687.

[33] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openaccfirst experiences with real-world applications," *Euro-Par 2012 Parallel Processing*, pp. 859–870, 2012.

[34] M. Norman, J. Larkin, A. Vose, and K. Evans, "A case study of cuda fortran and openacc for an atmospheric climate kernel," *Journal of computational science*, vol. 9, pp. 1–6, 2015.

[35] O. W. Group *et al.*, "The OpenACC Application Programming Interface," 2015.

[36] M. Alvanos and T. Christoudias, "GPU-accelerated atmospheric chemical kinetics in the ECHAM/MESSy (EMAC) Earth system model (version 2.52)," *Geoscientific Model Development*, vol. 10, no. 10, pp. 3679–3693, 2017. [Online]. Available: https://www.geosci-model-dev.net/10/3679/2017/

[37] ——, "MEDINA: MECCA Development in Accelerators–KPP Fortran to CUDA source-to-source Pre-processor," *Journal of Open Research Software*, vol. 5, no. 1, 2017.

[38] The Cyprus Intitute, "MECCA - KPP Fortran to CUDA source-to-source pre-processor, https://github.com/CyIClimate/medina," 2016.
[39] Nvidia, "Multi-process service," 2015.
[40] E. J. Fluhr, J. Friedrich, D. Dreps, V. Zyuban, G. Still, C. Gonzalez, A. Hall, D. Hogenmiller, F. Malgioglio, R. Nett *et al.*, "5.1 POWER8 TM: A 12-core server-class processor in 22nm SOI with 7.6 Tb/s off-chip bandwidth," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*. IEEE, Feb 2014, pp. 96–97.
[41] J. Stuecheli, "Power8," in *Hot Chips*, vol. 25, 2013, p. 2013.
[42] A. Sandu, J. Verwer, J. Blom, E. Spee, G. Carmichael, and F. Potra, "Benchmarking stiff ode solvers for atmospheric chemistry problems ii: Rosenbrock solvers," *Atmospheric environment*, vol. 31, no. 20, pp. 3459–3472, 1997.
[43] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.

Michail Alvanos Dr Michail Alvanos is Computational Scientist at the Cyprus Institute. He is working on accelerating climate modelling applications at the Institute's Computation-based Science and Technology Research Center (CaSToRC). He received his Ph.D. from the Department of Computer Architecture (DAC) of the Technical University of Catalonia (UPC) in Barcelona, Spain. He has a Master of Science degree in the field of Parallel and Distributed Systems in the Computer Science Department at the University of Crete. His research interests include High Performance Computing Systems, GPU acceleration, Computer Architecture, Compilation Techniques, Compiler Design, and Static Analysis.

Theodoros Christoudias Dr Theodoros Christoudias is an Assistant Professor at the Cyprus Institute. His research interests include global climate modelling and tracer transport, regional air quality modelling, computational model development and optimization, and scientific data visualization. He holds a PhD in Physics (2009) and a BSc in Physics (2005), both from Imperial College London, UK. He was previously a Computational Scientist at the Cyprus Institute and an International Fellow at the Fermi National Accelerator Laboratory (Fermilab), USA.