GPU accelerated atmospheric chemical kinetics in the ECHAM/MESSy (EMAC) Earth system model (version 2.52)

Michail Alvanos[1] and Theodoros Christoudias[1]

[1]*The Cyprus Institute*, PO Box 27456, 1645 Nicosia, Cyprus

Correspondence to: Theodoros Christoudias (christoudias@cyi.ac.cy)

Abstract.

This paper presents an application of GPU accelerators in Earth system modelling. We focus on atmospheric chemical kinetics, one of the most computationally intensive tasks in climate-chemistry model simulations. We developed a software package that automatically generates CUDA kernels to numerically integrate atmospheric chemical kinetics in the global climate model ECHAM/MESSy Atmospheric Chemistry (EMAC), used to study climate change and air quality scenarios. A source-to-source compiler outputs a CUDA compatible kernel, by parsing the FORTRAN code generated by the Kinetic Pre-Processor (KPP) general analysis tool. All Rosenbrock methods that are available in the KPP numerical library are supported.

Performance evaluation, using Fermi and Pascal CUDA-enabled GPU accelerators shows achieved speedups of $4.5\times$ and $22.4\times$ respectively of the kernel execution time. A node-to-node real-world production performance comparison shows a $1.75\times$ speed-up over the non-accelerated application using the KPP 3-stage Rosenbrock solver. We provide a detailed description of the code optimizations used to improve the performance including memory optimizations, control code simplification, and reduction of idle time. The accuracy and correctness of the accelerated implementation are evaluated by comparing to the CPU-only version of the application. The relative difference is found to be less than 0.00005% when comparing the output of the accelerated kernel the CPU-only code, within the target level of relative accuracy (relative error tolerance) of 0.1%.

The approach followed, including the computational workload division and the developed GPU solver code can potentially be used as the basis for hardware acceleration of numerous geoscientific models that rely on KPP for atmospheric chemical kinetics applications.

1 Introduction

One of today's great scientific challenges is to predict how climate will change locally as gas concentrations change over time. The study of chemistry-climate interactions represents an important and, at the same time, difficult task of global Earth system research. The emerging issues of climate change, ozone depletion and air quality, which are challenging from both scientific and policy perspectives are represented in Chemistry-Climate Models (CCMs). Understanding how the chemistry and composition of the atmosphere may change over the 21st century is essential in preparing adaptive responses or establishing mitigation strategies.

The global Atmosphere-Chemistry model ECHAM/MESSy (EMAC) is a numerical chemistry and climate simulation system that includes sub-models describing tropospheric and middle atmosphere processes and their interaction with oceans, land and human influences (Jöckel et al., 2010). It uses the second version of the Modular Earth Submodel System (MESSy2) to link multi-institutional computer codes. The core atmospheric model is the 5th generation European Centre Hamburg general circulation model (Roeckner et al., 2006). The EMAC model runs on several platforms, but it is currently unsuitable for massively parallel computers, due to its scalability limitations. In climate simulation applications the numerical integration by chemical kinetics solvers can take up to 90% of execution time (Christou et al., 2016). To achieve realistic simulation times, researchers are forced to limit the resolution of model simulations.

This paper describes a method of accelerating the chemical kinetics calculations on modern high-performance supercomputers using GPU accelerators. We present a source-to-source parser, written in the Python programming language, that transforms the MESSy chemical kinetics FORTRAN source code to CUDA source code, suited for running on CUDA-enabled general purpose graphics processing unit (GPGPU) accelerators. The parser transforms the auto-generated FORTRAN code by the KPP preprocessor (Damian et al., 2002a) into CUDA-compatible accelerated code, allowing to offload all different numerical integration solvers to GPU accelerators. The parser also makes the appropriate changes in the MESSy software distribution for linking the accelerated code during the compilation phase.

The paper is organized as follows: Sec. 1.1 describes the Messy/MECCA frameworks, the parallelization approaches, and Sec. 1.2 the potential of GPU accelerators. Sec. 1.3 discusses previous work that is related to this research. In Sec. 2, we present our implementation of atmospheric chemical kinetics parallelisation, the source-to-source parser code, and GPU-specific optimizations, including memory optimizations, the control code restructuring, and the refactoring of the EMAC source code. An evaluation of the resulting GPU accelerated climate model appear in Sec. 3. We summarise the main outcomes, present our conclusions and planned future work in Sec. 4.

1.1 The EMAC framework

The numerical global atmosphere-chemistry model EMAC (ECHAM/MESSy Atmospheric Chemistry) is a modular global model that simulates the chemistry and dynamics of the stratosphere and troposphere. The model includes different sub-models for the calculation of concentrations in the atmosphere, their interaction with the ocean and land surfaces, and the anthropogenic influences. The EMAC model runs on several platforms, but it is currently unsuitable for massively parallel computers, due to its scalability limitations and large memory requirements per core.

The MESSy submodel MECCA executes independently the gas phase chemical kinetics because there are dependency free of its physical neighbours and is not limited by vertical adjacency relations. A typical configuration of MESSy with 155 species and 310 chemical reactions takes 70% of the simulation execution time (Christou et al., 2016). The percentage can go up to 90% of execution time in simulations with more complex chemistry.

Currently EMAC uses coarse-grained parallelism based on the Message Passing Interface (MPI). However, the current approach does not benefit from the accelerators that exist in modern hybrid HPC architectures. This puts severe limitations
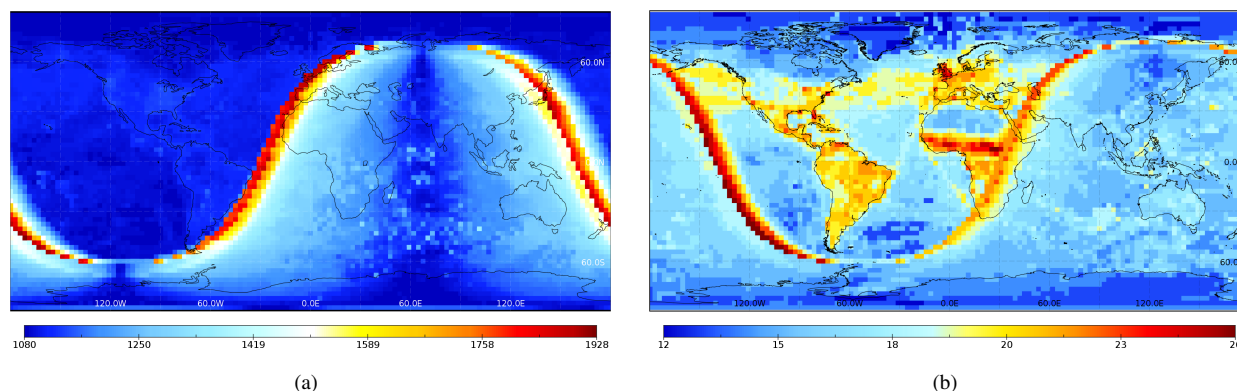
Figure 1. (a) Integration steps for column MECCA kernel execution, and (b) Intra-column maximum difference in execution steps (b). The adaptive time-step integrator shows a non-uniform run time caused by stratospheric photochemistry and natural and anthropogenic emissions.

on current global climate time-length atmospheric chemistry and pollution transport simulations in portability, complexity and resolution.

EMAC uses the Kinetic Pre-processor (KPP) (Damian et al., 2002a) open source general analysis tool to produce the chemical mechanism. KPP integrates very efficient numerical analysis routine and automatically generates FORTRAN and C code
5 that computes the time evolution of chemical species from a specification of the chemical mechanism in a domain-specific language. Taking a set of chemical reactions and their rate coefficients as input, KPP generates code of the resulting coupled ordinary differential equations (ODEs). Solving the ODE system allows the temporal integration of the kinetic system. Efficiency is obtained by exploiting the sparsity structures of the Jacobian matrix.

The biggest challenge to address in the application is the imbalance caused by the adaptive time-step integrator solving
10 the differential equations that describe the chemical equations computed. The varying light intensity at sunrise and sunset in combination with concentrations of precursors and gasses (such as NO_x, O_3) lead to photochemical reactions (mostly over mid-latitudes in the stratosphere) that heavily alter the stiffness of the ODEs. For example, Fig. 1(a) presents the cumulative number of execution steps required for the integration process for each model column and Fig. 1(b) presents the maximum difference in the number of steps between cells in each column. The difference in the number of steps inside and between
15 columns provides an indication of the strong imbalance created between execution time of different processes.

The ECHAM atmospheric dynamical circulation phase of EMAC only scales up to approximately a few hundred cores (Christou et al., 2016), due to the heavy all-to-all communication overhead of the spectral decomposition. At higher levels of parallelism, at or beyond approximately 1000 cores, the MECCA load imbalance due to the photochemistry also becomes a limiting factor.

1.2 GPU Accelerators

A common trend in computing today is the utilization of hardware accelerators that efficiently execute codes rich in data parallelism, to form high-performance heterogeneous systems. Graphical Processing Units (GPUs) are commonly used as accelerators due to high peak performance offered. GPU accelerators are module extensions connected using an interconnect to the CPU and memory sub-system. GPU accelerators feature on-board memory that provides high-bandwidth, medium-latency access to data. In contrast with the general purpose CPUs, the GPU processors are designed for massive parallelism by issuing thousands of simple parallel tasks. Programming a GPU accelerator can be a hard and error-prone process that requires specially designed programming models, such as the CUDA (Nvidia, 2015b) and OpenCL (Munshi, 2009). To unlock the potential performance of accelerators, programmers often apply incremental optimizations to their applications.

1.3 Related Developments

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

The adoption of intrinsic optimizations for sparse linear algebra (Zhang et al., 2011; Jacobson and Turco, 1994) is one of the first improvements implemented to speed up simulations of chemical kinetics. In a chemical kinetic solver, the majority of the time is spent solving the linear systems using implicit integration mechanisms. Only a fraction of the matrices have values different than zero, allowing for more efficient implementations using the sparsity of the structures. The sparse structure depends on the chemical reactions and thus the linear algebra can be solved off-line before the execution of the application.

Researchers have also improved the performance of the chemical kinetics portions of the code by using data and instruction level parallelism (DLP and ILP) when solving the chemical kinetics equations. Approaches include the introduction of SIMD instructions and annotation using the OpenMP programming model. However, these approaches often rely on the ability of the compiler to auto-vectorize the code that very often misses opportunities (Zhang et al., 2011), and the OpenMP implementation heavily burdens the performance due to high overhead of scheduling and managing threads.

A more coarse-grained approach is to use grid- or box- level parallelization (Linford et al., 2009; Linford, 2010; Christoudias and Alvanos, 2016). The application breaks the grid or box in cells allowing the calculation of concentrations independently between cells. Thus, the application assigns (equally) a number of cells to each thread to allow embarrassingly parallel execution of the chemical solvers. The biggest drawback of this approach is the limited parallelism and the imbalance that is created due to the photochemical processes in the lower stratosphere. A similar approach is used for the parallelization of chemical kinetics in the Cell processor and GPU accelerators by Damian et al. (2002b). However, these approaches exhibit the same limitations: limited parallelism and imbalance. Moreover, it is necessary for researchers to effectively rewrite their applications every time they run on the accelerators. Current commercial approaches of parallelizing the chemical kinetics use fine-grained parallelism, that is suitable only when the number of elements and chemical reactions are complex enough to justify the use of accelerators.

Researchers proposed different organization of the workload using Adaptive Mesh Refinement (AMR) as a way to increase locally the resolution for a area of interest (Keppens et al., 2003). The adaptive unstructured mesh representation in climate models can improve the overall performance of the application (Schepke and Maillard, 2012; Schepke, 2014). Compared to traditional horizontal partitioning these solutions offer greater scalability. However, the communication demands are dominant in a high number of processes and the potential imbalance created by the chemical reaction and the heterogeneity of the modern HPC machines is ignored by this approach.

An earlier prototype of the application in this paper is outlined in Christoudias and Alvanos (2016), focusing on the challenges of using GPU accelerators to exploit node-level heterogeneity. This paper significanlty expands on the previous work, both in detailed implementation and optimization. Moreover, this paper presents a performance evaluation of the first public release of the source code.

2 Implementation

The section presents the implementation of the source-to-source parser and the challenges addressed. The parser is written in the Python programming language and generates CUDA (Nvidia, 2015b) compatible solvers, by parsing the auto-generated FORTRAN code output by the KPP preprocessor. The process is automated and allows the parser to work with all possible user-defined chemical mechanisms, without requiring changes by the end-user on the accelerated kernel code. The FORTRAN compiler links the output CUDA object file with the rest of the application.

The user executes the parser from the `messy/util` directory to transform the code. The parser modifies the `messy/smcl/messy_mecca_kpp.f90` file and places a single call to the CUDA source file that contains the accelerated code (`messy/smcl/messy_mecca_kpp_acc.cu`) and a wrapper function for issuing the parallel kernels and copying the data to and from the GPU. The solver supports all five variations of Rosenbrock solvers available in KPP (Ros2,Ros3,Ros4,Rodas3, and Rodas4). The parser is also responsible for making the changes to the EMAC source Makefile for linking during compilation. The user must modify the configuration of the EMAC model to include the CUDA runtime link. Similar to the FORTRAN implementation, the computation is subdivided in runtime-specified arrays of columns. The memory of each array transferred to the GPU global memory and each grid box calculated on individual GPU cores.

The CUDA chemical kinetics solver comprises three steps, also presented diagrammatically in Figure 2. Each task is offloaded using three different calls:

1. The first step is the calculation of the reaction rate coefficients. The variable values are stored in a global array inside the GPU and used in the second step.

2. The second step is the ODE solver that includes all linear algebra functions. The computational kernel contains five different variations of the Rosenbrock family of numerical solvers, specified by the user at runtime – as supported by KPP/KP4.
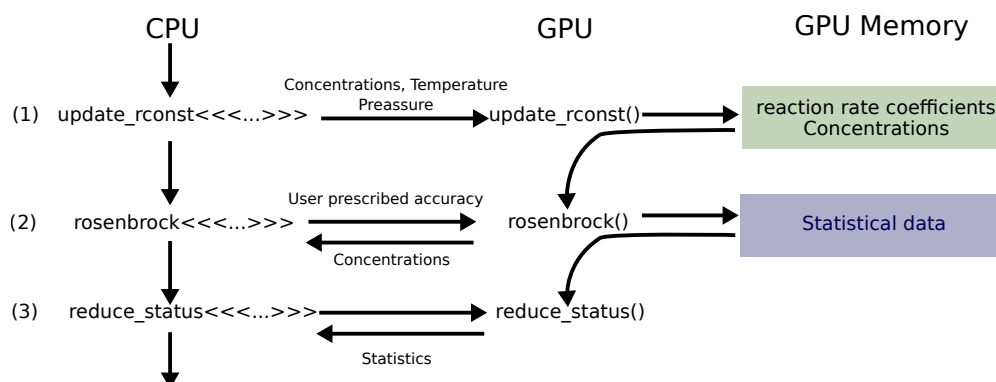
Figure 2. Flow-chart of the tasks offload execution on GPU accelerators.

3. The third and final step of the solver is the statistical reduction of the results, which demands very limited computational time compared with the other steps.

---

kpp_integrate($time\_step\_len, Conc, gstatus, VL\_GLO$)

1: // VL_GLO is NVL × levels_atmosphere

2: **for** DO $k = 1$ to VL_GLO **do**

3:     $status \leftarrow \emptyset$;

4:     $C(:) \leftarrow Conc(k,:)$

5:     $update\_rconst(C)$;

6:     $integrate(\ time\_step\_len,\ C,\ status)$ // Main kernel

7:     $Conc(k,:) \leftarrow C(:)$

Diagnostic Information

8:     $gstatus(k) \leftarrow status$

9: **end for**

Algorithm 1: CPU Chemical Kinetics Solver.

The parser replaces the code of integrator loops located in the `messy_mecca_kpp.f90` file with a single call to the CUDA accelerated code. The CUDA source file includes an entry function that is responsible for moving the data to and from the GPU accelerator and issuing the computation kernels. Algorithm 1 presents the outline of the integrator code. Algorithm 2 presents the outline of the GPU glue code. Each GPU thread calculates the chemical concentrations of an individual cell. The temporary arrays used between steps of the solver are allocated in the stack memory with the exception of the RCONST array. The RCONST array contains the reaction rate coefficients that are re-calculated in the first step of the integration process. Using the stack to allocate the memory revealed good results in the current and forthcoming generations of GPU architectures due to simpler indexing. However, the design architecture for memory allocation is subject to change when the impact of coalesced memory access is more prominent.

$kpp\_integrate\_acc(time\_step\_len, Conc, gstatus, VL\_GLO)$

1: $status \leftarrow \emptyset$;

Step 1: Initialize devices and allocate memory.

2: **if** $initialized$ is $FALSE$ **then**

3:     $dev\_conc\_ptr, dev\_rconst\_ptr, dev\_status\_ptr \leftarrow init\_first\_time(pe, VL\_GLO)$;

4: **end if**

5: $cudaMemcpy(Conc(:), device\_conc\_ptr, sizeof(double) \times VL\_GLO \times NSPEC)$

6: $Blocking \leftarrow 64$;   // Blocking can be changed to 128 by modifying the produced file

Step 2: Copy back the results.

7: $update\_rconst <<< VL\_GLO/Blocking, Blocking >>>(device\_conc\_ptr, device\_rconst\_ptr, VL\_GLO)$;

8: $Rosenbrock <<< VL\_GLO/Blocking, Blocking >>>(device\_conc\_ptr, device\_rconst\_ptr, dev\_status\_ptr, VL\_GLO)$;

9: $reduce\_status <<< VL\_GLO/Blocking, Blocking >>>(dev\_status\_ptr, VL\_GLO)$;

Step 3: Copy back the results.

10: $cudaMemcpy(device\_conc\_ptr, Conc(:), sizeof(double) \times VL\_GLO \times NSPEC)$

11: $cudaMemcpy(dev\_status\_ptr, gstatus, sizeof(double) \times VL\_GLO)$

Algorithm 2: GPU Accelerated Chemical Kinetics Solver.

The global GPU memory size suffices for offloading the full set of data, and it is not a limiting factor for the present and future projected chemistry mechanism complexity. The performance is also not limited by the overuse of any function unit. Global and local memory is used to store all required data, and local variables are used for each grid box for temporary variables during computation, with cache misses and register usage being the current performance limiting factor. The block size of the

5 GPU kernels is set to 64, and it can be changed to 128 for increased efficiency if future GPU architectures. The maximum number of cells that can be offloaded to GPU is 12288. Thus, this sets an upper limit to the value of the ECHAM5 `NVL` runtime parameter to 128, assuming 90 levels for the atmosphere.

The 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 mechanism. The required stack memory can be calculated: $stack\_allocated\_memory =$

10 $(number\_of\_elements \times 13 + lu\_nonzero \times 2 + spilledvariables) \times sizeof(double)$. For example, in workload of 155 species and 310 reactions, the required stacked memory including the temporary allocate arrays is $(155 \times 13 + 2 \times 1500 + 1200) \times 8 = 40120$ approximately. Note that the size of spilled variables depends on the complexity of the source code. The total memory required for a kernel is calculated for 12288 cells is: $12288 \times (stack\_allocated\_memory + global\_allocated\_mem) = 12288 \times 52000 \approx 638MB$. Moreover, during the GPU code loading, the GPU driver unrolls the loops and expands the of-

15 floaded code making it limiting reason of allocated memory. For example, the aforementioned chemistry of EMAC requires least 2GB of VRAM to be available on the Fermi architecture or more than 7GB in the newest Pascal architecture only for the code. Thus, the total memory required for each process depends not only on the size of chemistry but also in the complexity of the source code (number of reactions) and the architecture of the GPU accelerator.

2.1 Challenges and Optimizations

To achieve the best possible performance and efficiency three challenges had to be addressed: First, the biggest limitation in the performance of the chemical kinetics kernel is the memory required. To solve the ODEs and calculate the new concentrations, more memory is required in total than the on-chip memory available. For example, a set of 155 species and 310 reactions

5 requires the use of ~50KB of stack memory per cell. Thus, running multiple threads concurrently in GPU forces the use of global memory for storing the data. Second, the source code complexity of the solver kernel increases the usage of registers and limits the occupancy of the GPU Streaming Multiprocessor (SM). This results in increased execution time of the kernel. Third, the number of steps for solving the differential equations differs between different cells. This creates thread divergence that limits the performance.

10 To address these challenges we implemented incremental improvements in the source code running on the GPUs. Our aim is to improve the occupancy of GPU occupancy, reduce the memory impact, simplify the source code, and reduce the idle time.

2.1.1 Occupancy improvement

The high register pressure limits the occupancy, the number of concurrent threads per Streaming Multiprocessor (SM), and increases the overall execution time of the kernel. There are two ways of overcoming this challenge: either place a flag during

15 compilation that limits the number of registers, or use the Lunch Bound qualifier (`__launch_bounds__`) to a specific kernel. Our implementation uses the second option as a generic and future proof approach. The downside of limiting the register usage is the increase of register spilling, causing an increase of stack memory usage. The compiler allocates additional stack memory for spill loads and stores, creating additional local memory traffic, that does not fit on-chip. Thus, the application execution time is dominated by the global memory access latency.

20 ### 2.1.2 Memory optimizations

GPU accelerators employ a wide width of memory bus that allows for high-bandwidth transfers at the cost of high latency. To achieve the best execution efficiency on the GPU, we apply a number of memory optimizations: i) better utilization of the on-chip memory, ii) privatization of each thread data structure, and iii) prefetching.

Each SM contains a small amount of local memory that can be used as shared memory between threads or for Level-1 (L1)

25 cache. The size of temporary matrices is larger than the available on-chip memory. Thus, only a small portion of the memory can be used to store the data. In particular, the concentrations of chemical species that remain unchanged by chemical kinetics are stored in this memory. To increase the utilization of the local memory, we increased the portion of the L1 cache against the shared memory, through the `cudaFuncCachePreferL1` runtime call.

The solver must keep intermediate results in temporary arrays during different steps of the solver. The temporary arrays are

30 larger than the available on-chip memory, forcing us to declare the arrays on global memory. Although accesses on these arrays can be coalesced, there is still overhead occurring due to cache misses. Moreover, different execution paths of the kernel can

complicate and limit the coalescing of the data. As a result, 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.

The last modification of the memory optimizations is to employ prefetching of data. Prefetching may have unpredictable behavior, especially in massively parallel architectures, such as GPUs. There are four possible prefetch instructions that can

5 be applied to the chemical kinetics. Micro-benchmarks showed better results by using the `prefetch.global.L1` and `prefetch.local.L1` inline PTX assembly for fetching the data to the L1 caches.

2.1.3 Control code simplification

The GPU cores are relatively simple units and their computation efficiency depends on the absence of control code. To increase the instruction level parallelism and avoid possible branch divergence, the implementation adopts three commonly used

10 techniques: i) use of lookup tables, ii) unrolling loops, and iii) branch elimination. The lookup tables are used for selecting the solver and setting the proper values in specific variables. The benefits of loop unrolling are most profound in the preparation of the Rosenbrock solver `ROS` when using the sparse matrix. Finally, limited branch elimination by fusing loops or merging branches also improved the execution time.

2.1.4 Decreasing the GPU idle time

15 The biggest challenge when using accelerators in hybrid HPC architectures is the imbalance created by the uneven workload of tasks. For example, while the application uses GPUs as accelerators, only the CPU cores responsible for communication and handling of GPUs are active, leaving the remaining cores idle. Allocating more processes on the unused CPU core creates more imbalance between the accelerated processes and the CPU-only processes. To address this, the goal is to increase the GPU utilization by assigning more than one GPU per process.

20 The are two ways to do this: i) over-subscription, and ii) using the *Multi-Process Service (MPS)* (Nvidia, 2015a). The first way is to allow more than two CPU cores (MPI processes) from one node to be offloaded to the accelerators. The downside of this approach is that only one process can access the GPU (exclusive access). Although there will be some benefit compared to the case of using a single process, it is possible that some of the tasks will underutilize 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.

25 An alternative approach is to use the *MPS* that allows concurrent execution of kernels and memory transfers from different processes on the same node. The latter requires a GPU accelerator with computation capability 3.5 or higher.

2.2 Future optimizations

The execution time between chemical kinetics tasks depends on the input data of each cell grid. Thus, different execution time of tasks creates imbalance between tasks running on the same GPU. A promising approach to addressing this challenge is

30 the usage of Dynamic Parallelism: the ability each GPU thread to call other kernels and spawn more GPU threads eventually

Node	CPU	Cores	RAM	Accelerators			Peak Performance
		#	GB	NVIDIA GPU	Cores	VRAM	GFlops (DP)
DELL® dx360	Xeon X5650	12	48	2× M2070	2×448	2×6 GB	1159 (128 CPU + 2×515 GPU)
IBM® S822LC	POWER8	20	256	4× P100	4×3584	4×16 GB	21667 (467 CPU + 4×5300 GPU)

Table 1. Hardware configurations used for performance evaluation.

Number of columns	8192 columns with 90 levels
Number of grid points	737280 grid points
Number of chemical species	155 species and 310 chemical reactions
Spectral resolution	T42L90MA
Ordinary Differential Equations Solver	Ros3: 3-stage, L-stable pair of order 3(2) (Sandu et al., 1997)

Table 2. Experimental configuration of the simulation.

increasing the available concurrency. Work is underway to further optimize the memory subsystem utilization and decrease the off-chip traffic.

3   Results and evaluation

This section presents: (i) the total impact on simulation time, and (ii) the accuracy and correctness of the accelerated model.
Two different hardware and software environments were used to measure the impact of the code acceleration (Table 1):

  – An iDataPlex dx360 M3 compute node that contains two Intel® Xeon®X5650 6-core processors running at 2.6GHz
    coupled with a Tesla M-series GPU M2070 accelerator (Fermi architecture). The application is compiled using the Intel
    compiler (`ifort` ver. 14.0.2) for improved native performance.

  – An IBM® S822LC compute node equipped with two 10-core 2.92 GHz POWER8 processors (Fluhr et al., 2014;
    Stuecheli, 2013) with turbo up to 4 Ghz. Simultaneous multithreading is set to 4, for optimum performance in HPC
    applications. The application is compiled using the IBM compiler (`xlf` ver. 15.1.5). The execution of the `mpirun`
    command includes `-map-by L2cache -bind-to core:overload-allowed` to reduce the contention of the
    cache and function units.

To test the model scalability inside for a computation node, the evaluation uses a representative benchmark 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. Table 2 details the experimental set-up for the results shown in this section. The default E5M2 KPP chemistry

Configuration	Median CPU exec time (s)	Median Accelerated exec time (s)	Performance over CPU
Intel Xeon X5650 + M2070	4.502	0.999	4.50×
IBM Power8 + P100	3.040	0.149	20.40×

Table 3. Median execution time and achieved speedup for the non-accelerated (CPU-only) code on a single core and the accelerated version of the kernel for the two platforms. The extracted execution timing results are for the timestep after the initialisation of the simulation.

batch option, along with model namelist set-up (NML_SETUP) `E5M2/02b_qctm`, without dynamics nudging and with the diagnostic submodels D14CO, DRADON, S4D, and VISO switched off.

3.1 Application performance

This section compares the performance of the CPU-only and the GPU-accelerated version of the application. The evaluation
5   uses only one node to avoid any MPI communication interference in the results and limits the period of simulated time to six hours. The execution time does not include the initialization of the application and the file I/O operations.

The accelerated version of the kernel achieves an order of magnitude performance improvement over with the CPU-only version, as shown in Table 3. We note that the theoretical performance difference between the CPU core and the accelerator is larger in the P100 platform. The performance improvement of the kernel can provide an indication of the expected application
10 production performance improvement.

Table 4 presents the execution time in seconds and the gain in performance between the GPU-accelerated and CPU-only versions for the two platforms. The node-to-node comparison allows us to compare the accelerated version with the best performance achieved using the CPU-only version, using all available cores. The application performance gain differs between the available platforms. The M2070 accelerator unit contains only 6GB of memory, limiting the number of processes that can be
15 offloaded in parallel to the accelerator. Moreover, the M2070 does not contain support for the *Multi-Process Service* (Nvidia, 2015a). Thus, in this experiment, we use the remaining processes to run on the CPU instead of accelerators. Despite the significant 2.27× performance gain over the two processes execution, the performance gain over when using the entire node is limited. The accelerated processes finish before the CPU-only processes, causing severe load imbalance between different MPI processes. Thus, the benefit of having additional accelerated cores is not reflected in the total attainable performance.
20 On the Power8 platform, the applications use the Multi-Process Service to interact with the accelerators using more than two MPI processes. The best CPU-only performance is achieved using over-subscription to the CPUs, by using two MPI processes assigned to physical cores. The accelerated version achieves 1.75× performance gain over the best performing CPU-only configuration, clearly showing the benefit of accelerating the chemical kinetics on GPUs. The accelerated version with 40 MPI processes achieves lower performance than the 20 MPI processes (not shown).

Configuration	MPI Processes	CPU exec time (s)	Accelerated exec time (s)	Performance over CPU
2× 6-core Intel Xeon X5650 + 2× NVIDIA M2070	2 MPI processes	5199	2358	2.27×
	12 MPI processes	1388	1368	1.01×
2× 10-core IBM POWER8 + 4× NVIDIA P100	4 MPI processes	2205	1652	2.50×
	20 MPI Processes	814	437	1.86×
	40 MPI Processes	766	-	1.75× over 20 MPI Procs

Table 4. Application execution time and achieved speedup for two node configurations.
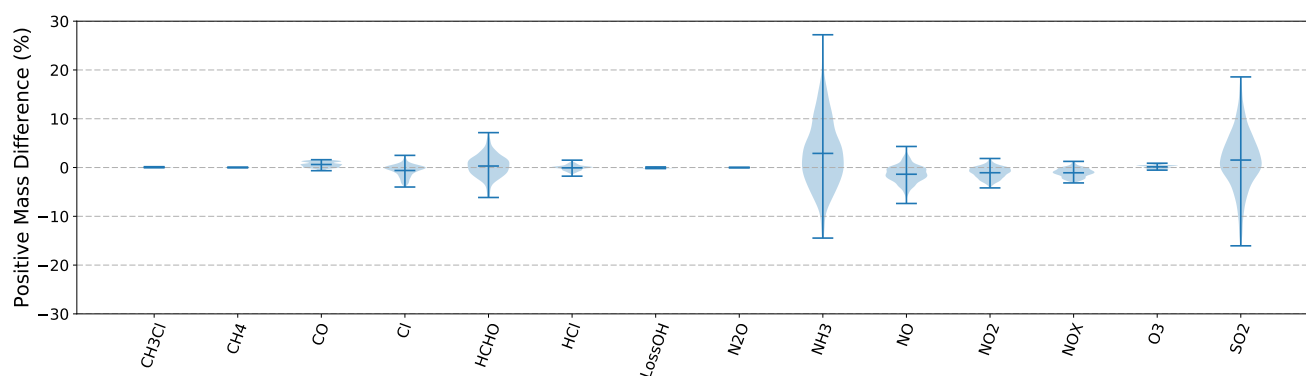
Figure 3. Aggregated mass difference between the accelerated and non-accelerated version of one year simulation for different time-steps with 15 chemical species.

3.2 Numerical and model accuracy

To validate the correctness, and evaluate the accuracy of the accelerated implementation we calculate the numerical and modeling errors over one year simulated time. The target level of relative accuracy (relative numerical error tolerance) for the chemical kinetics calculation is 0.1%, i.e. about 3 accurate digits (Zhang et al., 2011). To calculate the relative error, we com-
5   pare the output of chemical element concentrations between the CPU only and accelerated version after the first time-step. The results show a median difference of 0.00005% with the maximum difference value of 0.54%. using identical inputs, ensuring the high accuracy of the chemical solver. This is well within the accuracy criterion, assertaining the numerical correctness of the GPU kernel.

The variance in floating point results between different architectures is well known and observed in scientific applications
10   that require high-precision floating point operations (Corden and Kreitzer, 2012; Langlois et al., 2015). The `ift` compiler produces intermediate results stored with an extended precision that provide greater precision than the basic floating point formats of 64bit. On the other hand, the GPU accelerators don't support the extended precision values, reducing the floating point accuracy in the results. Furthermore, the `REAL(16)` declaration in Fortran programs is implemented in software for the

Power8 architecture. Despite the small difference in the results, the model remains stable when running over a full climatic year.

Furthermore, to examine the impact of data errors in the model accuracy, we compare the results of aggregated mass of the CPU-only and GPU-accelerated version, comparing the differences of the chemical differences over one year simulated time. Figure 3 presents the difference between the accelerated and non-accelerated version. For simplicity, we include 15 important chemical species. The results show that the median value of the difference in aggregated mass is less than 5%, well within the expected margin of differences stemming from architecture and compiler implementations (not specific to GPU).

4 Conclusions

The global climate model ECHAM/MESSy Atmospheric Chemistry (EMAC) is used to study climate change and air quality scenarios. The EMAC model is constituted by a nonlocal dynamical part with low scalability and local physical/chemical processes with high scalability. Advancements in hardware architectures over the last three decades have greatly improved not only the spatial and temporal resolution of climate models but also the representation of key processes. The slow adoption of accelerators in the climate and weather models has positioned the climate simulation community behind other scientific communities (Nvidia, 2017).

In this paper, we presented the acceleration of the KPP chemical kinetics solver in the EMAC chemistry-climate model using a source-to-source parsers to transform the solver to CUDA accelerated code. The parser supports all Rosenbrock family numerical solvers (Ros2, Ros3, Ros4, Rodas3, Rodas4) that are available in the KPP numerical library. A performance evaluation, using two CUDA enabled accelerators shows an achieved speedup of up to $22.4\times$ of the kernel execution time and up to $1.75\times$ node-to-node application performance gain. A comparison of the aggregated global mass of 15 chemical elements between the CPU-only and accelerated version of the application to verify the correctness of the transformation. The numerical accuracy was also assertained, with the relative error less than 0.01% when using the same data as input to the accelerated kernel and the CPU-only chemical kinetics segment of the code.

The approach followed, including the computational workload division and the developed GPU solver code can potentially be used as the basis for hardware acceleration of numerous geoscientific models that rely on KPP for atmospheric chemical kinetics applications.

This work is a marked step forward to increase the resolution of climate simulations using chemical kinetics and expand the usage of GPU accelerators in Earth system modeling.

5 Code availability

The Modular Earth Submodel System (MESSy) is continuously further developed and applied by a consortium of institutions. The usage of MESSy and access to the source code is licensed to all affiliates of institutions which are members of the

MESSy Consortium. Institutions can become a member of the MESSy Consortium by signing the MESSy Memorandum of Understanding. More information can be found on the MESSy Consortium Website (http://www.messy-interface.org).

The FORTRAN to CUDA code-to-code compiler is developed in C and Python and it is included in the EMAC model development release. In addition, the source code is included in public repository under open license, to allow the developer

5   community to contribute (Alvanos, 2017; The Cyprus Intitute, 2016). It parses the auto-generated MECCA KPP solver code and produces a CUDA library that can be linked to and called directly from within the MESSy FORTRAN code at the time of compilation. In addition, the source code a test script that transforms a simplified chemistry included in the release using the five available Rosenbrock solvers. The "driver.sh" test script inside the test directory makes the following tests: (i) Checks if the parser executes without errors, (ii) checks if the produced file compiles without errors, (iii) runs the application and check

10   for possible errors and memory violations, and (iv) compares the output of the accelerated version with the output of the initial Fortran code.

Appendix A: Optimization flags

For completeness we include the optimization flags for the compilers used in the two systems:

- For Intel platform, we used the ifort version 14.0.2 with the following optimization flags: `-O3 -g -debug full`

15   `-traceback -fp-model precise -fp-model source -fp-speculation=safe -fpp -align all`

- For IBM PowerPC platform, we used the xlf compiler version 15.1.5 with the following optimization flags: `-O3 -g`
`-qnohot -qarch=auto -qcache=auto -qsource -qtune=auto -qsimd=auto`
`-qinline=auto:level=10 -qprefetch=aggressive:dscr=7`

20   ### Appendix B: Impact of optimizations

The evaluation uses a microbenchmark based on the accelerated kernel to demonstrate the effectiveness of the code transformations. The microbenchmark uses the input concentrations extracted from the execution of one time-step of the application. We use five different variations of the Rosenbrock solver and two GPU accelerators, the M2070 and the P100, and we record only the GPU execution time. Tables 5 and 6 of the evaluation present the results of the kernels for each group of optimizations

25   applied.

The *occupancy* optimization and the change of the cache organization (*Prefer L1*) have minor performance gains. Furthermore, the performance gain of these optimizations in the P100 accelerators is limited within the margin error. The most notable negative impact (-18%) in the performance is the usage of the *Prefetch* intrinsics inside the source code in the P100 accelerator. The newest accelerators contain advanced hardware prefetchers that decrease the impact of the memory optimization. Thus,

30   this forced us to modify the source code for the P100 platform by disabling the software prefetcher.

Solver	Base (s)	Occupancy (s)	Prefer L1 (s)	Privatization (s)	Prefetch (s)	Simplification (s)	Total Speedup
Ros2	9.60	9.348	9.41	6.77	6.42	6.05	+58.48%
Ros3	10.71	10.43	10.43	7.79	7.39	7.16	+49.58%
Ros4	12.54	12.28	12.24	9.52	9.18	9.05	+38.65%
Rodas3	12.58	12.31	12.28	9.51	9.17	9.01	+39.59%
Rodas4	17.67	17.31	17.27	14.16	14.06	14.10	+25.33%
Mean Speedup	-	+2.21% ± 0.3	+0.22% ± 0.43	+29.18% ± 6.35	+3.69% ± 1.93	+2.58% ± 2.59	+19.14% ± 5.12

Table 5. Execution time in seconds and speedup running the kernel on a M2070 CUDA enabled accelerator placed in a node equipped with two Intel Xeon X5650 processors at 2.67 GHz with synthetic input.

Solver	Base (s)	Occupancy (s)	Prefer L1 (s)	Privatization (s)	Prefetch* (s)	Simplification (s)	Total Speedup
Ros2	1.23	1.26	1.18	0.98	1.20	0.90	37.24%
Ros3	1.39	1.39	1.39	1.14	1.36	1.04	34.17%
Ros4	1.75	1.74	1.74	1.44	1.70	1.29	40.79%
Rodas3	1.74	1.75	1.76	1.39	1.70	1.23	41.23%
Rodas4	2.75	2.74	2.75	2.12	2.66	1.95	40.57%
Mean Speedup	-	+0.06% ± 1.09	-0.14% ± 2.81	+22.48% ± 3.87	-18.14% ± 2.03	+9.62% ± 3.30	+40.57% ± 3.04

Table 6. Results running the kernel on a P100 CUDA enabled accelerator hosted in a node equipped with two Power8 processors. In the P100 platform, we removed the prefetch optimization, as it decreases the performance in all cases.

On the other hand, the *Privatization* of global arrays using shared memory, registers, and stack allocated arrays, gave the biggest benefit by reducing the off-chip memory traffic. The performance gain for the memory optimizations is less pronounced in the newest CUDA enabled architectures, as they provide better memory subsystem compared with the M2070 accelerator. The simplification of the source code has the greatest impact in the newest architectures due to reduced control dependencies inside the accelerated code.

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

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.

References

Alvanos, M. (2017). MECCA - KPP Fortran to CUDA source-to-source pre- processor - Alpha Version. doi:10.5281/zenodo.439896.

Christou, M., Christoudias, T., Morillo, J., Alvarez, D., and Merx, H. (2016). Earth system modelling on system-level heterogeneous architectures: Emac (version 2.42) on the dynamical exascale entry platform (deep). *Geoscientific Model Development*, 9(9):3483.

5 Christoudias, T. and Alvanos, M. (2016). Accelerated chemical kinetics in the EMAC chemistry-climate model. In *High Performance Computing & Simulation (HPCS), 2016 International Conference on*, pages 886–889. IEEE.

Corden, M. J. and Kreitzer, D. (2012). Consistency of Floating-Point Results using the Intel® Compiler. *Intel Corporation*.

Damian, V., Sandu, A., Damian, M., Potra, F., and Carmichael, G. R. (2002a). The kinetic preprocessor KPP-a software environment for solving chemical kinetics. *Computers & Chemical Engineering*, 26(11):1567–1579.

10 Damian, V., Sandu, A., Damian, M., Potra, F., and Carmichael, G. R. (2002b). The kinetic preprocessor KPP-a software environment for solving chemical kinetics. *Computers & Chemical Engineering*, 26(11):1567–1579.

Fluhr, E. J., Friedrich, J., Dreps, D., Zyuban, V., Still, G., Gonzalez, C., Hall, A., Hogenmiller, D., Malgioglio, F., Nett, R., et al. (2014). 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*, pages 96–97. IEEE.

15 Jacobson, M. Z. and Turco, R. P. (1994). Smvgear: A sparse-matrix, vectorized gear code for atmospheric models. *Atmospheric Environment*, 28(2):273–284.

Jöckel, P., Kerkweg, A., Pozzer, A., Sander, R., Tost, H., Riede, H., Baumgaertner, A., Gromov, S., and Kern, B. (2010). Development cycle 2 of the modular earth submodel system (messy2). *Geoscientific Model Development*, 3(2):717–752.

Keppens, R., Nool, M., Tóth, G., and Goedbloed, J. (2003). Adaptive mesh refinement for conservative systems: multi-dimensional efficiency
20 evaluation. *Computer Physics Communications*, 153(3):317–339.

Langlois, P., Nheili, R., and Denis, C. (2015). Numerical reproducibility: Feasibility issues. In *New Technologies, Mobility and Security (NTMS), 2015 7th International Conference on*, pages 1–5. IEEE.

Linford, J. C. (2010). Accelerating atmospheric modeling through emerging multi-core technologies.

Linford, J. C., Michalakes, J., Vachharajani, M., and Sandu, A. (2009). Multi-core acceleration of chemical kinetics for simulation and
25 prediction. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 7. ACM.

Munshi, A. (2009). The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE.

Nvidia (2015a). Multi-process service.

Nvidia (2017). Gpu accelerated applications,http://www.nvidia.com/object/gpu-applications.htm.

Nvidia, C. (2015b). Programming guide.

30 Roeckner, E., Brokopf, R., Esch, M., Giorgetta, M., Hagemann, S., Kornblueh, L., Manzini, E., Schlese, U., and Schulzweida, U. (2006). Sensitivity of simulated climate to horizontal and vertical resolution in the echam5 atmosphere model. *Journal of Climate*, 19(16):3771–3791.

Sandu, A., Verwer, J., Blom, J., Spee, E., Carmichael, G., and Potra, F. (1997). Benchmarking stiff ode solvers for atmospheric chemistry problems ii: Rosenbrock solvers. *Atmospheric environment*, 31(20):3459–3472.

35 Schepke, C. (2014). *Improving Multi-level Parallelism and Online Mesh Refinement for Atmospheric Models*.

Schepke, C. and Maillard, N. (2012). Exploring multi-level parallelism in atmospheric applications. In *Computer Systems (WSCAD-SSC), 2012 13th Symposium on*, pages 41–48. IEEE.

Stuecheli, J. (2013). Power8. In *Hot Chips*, volume 25, page 2013.

The Cyprus Intitute (2016). Mecca - kpp fortran to cuda source-to-source pre-processor, https://github.com/CyIClimate/medina.

Zhang, H., Linford, J. C., Sandu, A., and Sander, R. (2011). Chemical Mechanism Solvers in Air Quality Models. *Atmosphere*, 2(3):510–532.