UNIVERSITY OF CALIFORNIA,
IRVINE

Source to Source Optimizing Transformations
Fast-J as Case Study for Global Chemistry Codes

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Fausto Artico

Thesis Committee:
Professor Alex Nicolau, Chair
Professor Alex V. Veidenbaum
Professor Tony Givargis

2015

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ALGORITHMS

ABSTRACT OF THE THESIS

Source to Source Optimizing Transformations
Fast-J as Case Study for Global Chemistry Codes

By

Fausto Artico

Master of Science in Computer Science

University of California, Irvine, 2015

Professor Alex Nicolau, Chair

Chemistry Climate Model (CCM) codes are important [43] to understand how to mitigate global warming [18, 17]. However, to produce meaningful results, CCM codes require performance in the scale of PetaFlop (and soon ExaFlop) [39] calculations. These scales have to be achieved within a reasonable power budget. It is therefore important to speedup as much as possible the execution of the already highly optimized state-of-the-art CCM codes.

Fast-J [35] is a very important and widely used CCM code for simulations at different scales of magnitude, i.e., local, global and cosmic. Each scale of magnitude and its performance rely on a code, the Fast-J core code.

In this thesis, we speedup the Fast-J core, selecting and implementing some high-level compiler transformations, which are not efficiently performed by CPU compilers.

The optimizations consistently reach a performance speedup always greater than 4.5 for each scale of simulation. To quantify this performance improvement, we compare the execution times of the new optimizations with the execution times of the state-of-the-art, already highly optimized, CPU multi-threading code.

Chapter 1

Introduction

Climate change has been a hot area of research in the last decades, with few rivals in its impact on humanity's future. The key to progress in this critical area is the development and validation of ever more accurate chemistry climate models (CCMs) and the improvement in performance which makes a qualitative difference in the results of the complex, large scale simulations involved.

1.1 Chemistry Climate Models

The critical components of the CCMs are the numerical models that simulate the scattering and absorption of sunlight throughout the atmosphere, vegetation canopy, and upper ocean [40, 22], using 1D [20], 2D [24, 19] or 3D [30, 2] lattices. Such numerical models, when implemented, highly optimized, and accelerated, forecast the arrival of dangerous weather conditions [10, 21] - e.g. hurricanes [37] - and give insights on how to mitigate dangerous climate phenomenons citetsiactgfcacarg,tganftng - e.g. global warming [29, 15].

1.2 Simulation Characteristics

The CCM and the volumetric size of the simulation determine the density of the lattice (number of points), and the homogeneity of the lattice (which accounts for the elementary volumes of lattice which can contain a different number of points).

Multiple air columns compose each CCM lattice. Each processing node receives a subset of these air columns at pre-simulation time. During each simulation step, each processing node updates the values of some variables - e.g. wind and humidity - at each one of its lattice points, and if necessary, corrects them [14]. The processing nodes therefore propagate the results to its neighbor nodes at the end of each simulation step.

1.3 Computational Intensity

The number of simulation steps depends on the lattice size and the temporal horizon of the simulation [42]. The lattice of simulations, studying the causes of warming phenomenons, easily cover many countries [1] or even the whole earth [5]. The temporal horizon of the simulations usually is of decades [12, 9] or centuries [6, 25, 28].

The number of variables to update, the number of points composing the lattice and the temporal horizon translates in costly - very time consuming - computations. Such costs are big for any class of general purpose computers, including supercomputers. Hence, speeding up such simulations is paramount to advance state-of-the-art forecasting possibilities.

1.4 Approximations

Current radiative transfer (RT) models (e.g., the Rapid Radiative Transfer Model for Global Chemistry Models (RRTM-G) [16] as used by the National Center of Atmospheric Research (NCAR) [11] and in the Department of Energy Community Earth System Model (DOE CESM) [27]) make simplified assumptions and therefore only approximate the real values

of the physics variables. This is due to the very high computational cost of their executions compared to the computational cost of other elements of the climate systems (e.g., atmospheric dynamics, cloud physics, ocean circulation, sea ice, chemistry, biogeochemical cycles).

The faster the codes, the greater the achievable accuracies. The approximations create bias errors in the modeling of photochemistry, heating rates, and the distribution of photosynthetically active radiation (PAR). However, the greater their accuracy, the greater the understanding on how to rapidly mitigate short-lived climate forcing agents (SLCFs: tropospheric O3, CH4, some HFCs, black carbon and other aerosols), a potential near-term solution to simultaneously slow global warming and improve air quality [33, 32].

1.5 Fast-J To Simulate Photolysis

An important and widely used CCM using RT models is Fast-J [45, 4, 38, 44]. Fast-J is specifically used a) to study shorted-lived climate forcing agents [36, 8] - which, as said, are responsible for slow global warming [36] and to study air quality [8] - and b) CO_2 concentrations in the atmosphere [3] - CO_2 concentrations have to be kept within the bound of 2 °C. [IPCC, 2013]. Studies using Fast-J allow, therefore, not only to understand how to mitigate SLCFs in the atmosphere, devising some countermeasures to reduce the warming rate and at the same time improving air quality, but also to create insights on how to open up the global cap on CO_2 emissions.

1.6 Important Softwares Using Fast-J

The Community Atmosphere Model number 5 (CAM5) [26], running on the Department of Energy architectures at the National Energy Research Scientific Computing Center cluster (DOE NERSC) [31], has now absorbed Fast-J.

Fast-J is also used in several of the chemistry-climate models participating in the In-

tergovernmental Panel on Climate Change, 5th Assessment Report (IPCC AR5 [34]), Oslo-CTM2, GESOCCM and GISS-ER2 [23].

The Whole Atmosphere Community Climate Model (WACCM) [13] and the Community Atmosphere Model with Chemistry (CAM-chem) [7] use Fast-J for the execution of the Community Earth Science Model (CESM) [41] CAM5.

1.7 Accelerate The State-Of-The-Art Fast-J Code

Shortening the running time of the Fast-J code is fundamental to make it green, accelerate the simulations, and increase their accuracy. Fast-J runs a total of at least 10 million hours / year worldwide (likely an under-estimate). Even on super-computing, dedicated and specialized multi-core architectures, the execution of the Fast-J core code requires months per simulation, this for any meaningful forecasting, even do the code is already highly optimized.

In this thesis, we therefore accelerate the Fast-J core code to improve its energy efficiency, accelerate the simulations, and increase their accuracy. To accomplish this, we apply some source to source transformations at the existing state-of-the-art CPU multi-threading Fast-J core code.

1.8 Source To Source Optimizing Transformations

The source to source transformation techniques we identify and propose are currently not implemented in CPU compilers or not efficiently executed by CPU compilers. One of the contribution of this paper is in fact the identification and the efficient implementation of such transformations to speedup the code execution without programming using intrinsics or assembly. Such transformations, which can be implemented as part of a source-to-source compiler are important, because they make the optimization techniques proposed in this paper deliver performing and portable code.

The source to source transformations not implemented or efficiently executed in / by

CPU compilers are 7: 1) layout modification, 2) reduction of the necessary number of data structures, 3) reduction of the number of instructions necessary for the lattice updates, 4) reduction of the necessary number of lattice updates, 5) further reduction of redundant calculations, 6) manual loop unrolling, and 7) transformations of some variables in constants.

1.9 Speedups and Robustness

The 7 source to source transformations make the new Fast-J core code at least 4.5 times faster than the state-of-the-art multi-threading, multi-wavelength, multi-layer, and multi-air-column Fast-J core code. Therefore, the newly optimized Fast-J core code opens research possibilities previously impossible.

The results are valid for every simulation, independently from the number of air columns, number of wavelengths per air column and number of layers per air column. This shows that the introduced optimizations are robust.

Chapter 2

The State-Of-The Art CPU Code

The state-of-the-art CPU Fast-J core code is already multi-threading, multi-air-column, multi-wavelength and multi-layer. Typical values for the number of wavelengths are 8, 16, 32, 64, 128, and 256. Typical values for the number of layers are 300, 325, 350, 375, 400, 425, 450, 475 and 500.

2.1 Explicit Index Expansion

The indexes for the access to the matrices are explicitly expanded. Fast-J has many multi dimensional matrices, e.g. $A[I][J][Z][W]$ - I, J, Z and W are dimensions. Index expansion takes care that each one of the occurrences of the string $A[i][j][z][w]$ is explicitly in the form $A[i \times J \times Z \times W + j \times Z \times W + z \times W]$.

Algorithm 1 State-Of-The-Art Fast-J - Explicit Index Expansion

```
1: procedure FAST-J(...)
2:     ...
3:     A[i × J × Z × W + j × Z × W + z × W] = ...
4:     r = ... + A[i × J × Z × W + j × Z × W + z × W]
5:     ...
6: end procedure
```

Explicit index expansion simplifies the job of the compiler about the analysis of data dependences. The easier the compiler job, the easier for the compiler to optimize the source code and so generate faster and more compact executable code.

2.2 Thread Synchronization

Synchronization problems among CPU threads have already been completely eliminated. During each execution, each thread has exclusive access to different parts of the data structures with the guarantee that no other thread will try to access the same parts during their reads and writes. This is one of the strengths of the CPU optimized code.

Algorithm 2 State-Of-The-Art Fast-J - Thread Synchronization

```
1:  procedure FAST-J(...)
2:      ...
3:      gti → global thread identifier
4:      iaC → index air column
5:      iWl → index wavelength
6:      Tt → total threads
7:      Wl → wavelength per air column
8:      Wpt → wavelengths per thread
9:      ...
10:     for w = 1 to Wpt do
11:         ...
12:         iaC ← (uint64_t) floor( (double) ( gti + w × Tt ) / Wl )
13:         if ( iaC == 0 ) iWl ← gti;  else  iWl ← ( gti + w * Tt ) − iaC × Wl
14:         ...
15:     end for
16:     ...
17: end procedure
```

2.3 The Two Main Functions

The core code has 2 main functions: the generator of the triangular systems and the solver, calling the generator. The generator and the solver contribute to 97% of the execution time

of each Fast-J simulation step. The remaining 3% is due to the other few small functions. The small functions only set the input parameters of the generator and the solver.

Algorithm 3 State-Of-The-Art Fast-J - The Two Main Functions

```
1: procedure FAST-J(...)
2:     ...
3:     aC → number of air columns
4:     Wl → wavelengths per air column
5:     ...
6:     procedure SOLVER(...)
7:         ...
8:         for c = 1 to aC do
9:             for w = 1 to Wl do
10:                ...
11:                procedure GENERATOR(...)
12:                    ...
13:                end procedure
14:                ...
15:            end for
16:        end for
17:        ...
18:        for c = 1 to aC do
19:            for w = 1 to Wl do
20:                ...
21:            end for
22:        end for
23:        ...
24:    end procedure
25:    ...
26: end procedure
```

The generator and the solver use 16 and 20 different multi-dimensional matrices. Many of these matrices have more than 2 dimensions: 13 have 4 dimensions, and 7 have 5 dimensions. The matrices are big. All the matrices with 4 or 5 dimensions have a dimension for the air-columns - range of thousands, a dimension for the wavelengths - range of hundreds, and a dimension for the layers - range of hundreds.

Chapter 3

Source To Source Transformations

We introduce and implement a total of 7 source to source optimizations. Each optimization is implemented at the top of the previous one. The source to source transformations are the following:

- Layout Modification. Layout modification is important to reduce the quantity of transferred data and so increase the probability of faster executions. L1 and L2 caches are very small and many threads execute the code. Making the data of the layers contiguous in global memory reduces the number of cache lines transferred from the global memory and among cache levels.

- Data Structure Eliminations. Eliminating data structures we reduce the total quantity of bytes necessary for the execution of the simulations and so the total quantity of bytes transferred per execution. This reduces the number and severity of potential bandwidth and latency memory bottlenecks.

- Pre-Calculation of Parts of the Indexes. Fast-J has many 2 or 3 level nested loops executing many vector to vector multiplications and additions. For each loop, we precalculate several parts of the indexes used in the next nested loop, before entering in the loop, this to further simplify the compilers analysis of the data dependencies and reduce the total number of instructions present in the final executable file.

- Data Locality Enhancement. Many matrix elements need to be updated many times. The elements to update can be moved in a variable before the start of the loop, kept local, updated many times, and only at the end, written back to the global memory. This saves many data transfers.

- Further Reductions. Some values are re-calcuated and used more times inside some nested loops. Such values can be calculated only one time outside the loop and the results write in some local variables, further reducing the number of necessary calculations and the number of cache line transfers.

- Manual Loop Unrolling. Fast-J has many small 2 level nested loops. The compiler does not know the dimension of the data structures at compiling time therefore cannot optimize in the best ways these nested loops. Therefore, we manually unroll the loops to simplify the optimization process of the compiler.

- Pre-Compiling Define Directives for the Dimension of the Data Structures. With this optimization the compiler knows at compiling time the exact dimensions of all the data structures and so can optimze the code in the best way. This optimization create a source code for each specific simulation - (number of air columns, number of wavelengths per air column , number of layers per air column).

In the next sections, we analyze in more detail the features and the advantages given by each one of the 7 source to source transformations just briefly described.

3.1 Layout Modification

It is important to reduce the total quantity of data to transfer for the execution of a simulation. The bigger the total quantity of data to transfer, the greater the probability of slowdowns due to bandwidth and latency memory problems.

Loops run on the air columns, the wavelengths, the layers and the data per layer in such order. In any group of nested loops, the loops running on the data of the layers are therefore always the more internal loops. However, for all the matrices, the index positions to access the data of the layers always precede the index position to access the layer number - e.g. A[aC][Wl][Ix][Iy][L], where aC is the index for the air-column, Wl is the index for the wavelength, Ix is the x index for the data of the layer, Iy is the y index for the data of the layer and L is the layer number.

Algorithm 4 Layout Modification - Before

```
1: procedure FAST-J(...)
2:     ...
3:     iaC → index air columL
4:     iWl → index wavelength
5:     ...
6:     for l = 1 to L do
7:         for i = 1 to M do
8:             for j = 1 to M do
9:                 ...
10:                B[iaC][iWl][i][j][l] = B[iaC][iWl][i][j][l] + ...
11:                ...
12:            end for
13:        end for
14:    end for
15:    ...
16: end procedure
```

This order implies many cache line transfers. Data per layer are from a minimum of 8 to a maximum of 16 per matrix. Each single data has to be read several times during the executions and the number of layers is always greater of 300. Therefore, each time the code reads the data of a layer from a matrix, from 8 to 16 cache lines are transfered for that matrix.

This kills any speedup. The data of a layer always has to be read and updated more times and from several different matrices during each single simulation step. The bandwidths and the latencies of the memories are unable to sustain this overhead. Slowdown happens, executions become slower.

The solution to avoid slowdowns and accelerate the code is to modify the data layout. We move the dimension for the layers between the dimension for the wavelengths and the first dimension for the index of the data of the layers. There are at maximum 16 elements per air-column, per wavelength, per layer, per matrix. Preserving the actual loop order, but modifying the order of the matrix dimensions in this way and changing the data layout, now, when the code reads or updates the data of a layer, only a maximum of 2 cache L1 lines will be transferred - L1 cache lines are 64 bytes and the elements of all the matrices are doubles.

Algorithm 5 Layout Modification - After

```
1: procedure FAST-J(...)
2:     ...
3:     iaC → index air columL
4:     iWl → index wavelength
5:     ...
6:     for l = 1 to L do
7:         for i = 1 to M do
8:             for j = 1 to M do
9:                 ...
10:                B[iaC][iWl][l][i][j] = B[iaC][iWl][l][i][j] + ...
11:                ...
12:            end for
13:        end for
14:    end for
15:    ...
16: end procedure
```

3.2 Data Structure Eliminations

Smaller the data structures, better. Smaller the data structures, smaller the probability of bottlenecks generated by the bandwidths and latencies of the different off-chip and on-chip memories.

Data structure elimination is determined by the analysis of the data dependencies and the design of the reuse of the same parts of some data structures at different moments in time during the executions.

After this optimization, the quantity of bytes per node now necessary for the simulations is 80 times smaller for the generator and 7 times smaller for the solver. This happens because the number of air columns aC per simulation is always in the order of thousands and the number of wavelengths Wl per air column is always smaller than 257 while the number of threads used to execute the simulations is always smaller than 129.

Table 3.1: Generator: Matrices and their sizes before and after the data structure eliminations. aC is the number of air columns, Wl is the number of wavelengths per air column, L is the number of layers per air column, M and M2 are the number of data per air column, per wavelength, per layer and Tt is the total number of threads used to execute a simulation.

Size Before	Matrices	Size After
$aC \times M$	WT , EMU	$aC \times M$
$aC \times Wl$	RFL , ZTAU	$aC \times Wl$
$aC \times M2$	PM0	$aC \times M2$
$aC \times Wl \times L$	FZ , ZTAU	$aC \times Wl \times L$
$aC \times Wl \times M \times M$	S, T , U , V , Z	$Tt \times M \times M$
$aC \times Wl \times M \times M2$	PM	$aC \times Wl \times M \times M2$
$aC \times Wl \times M \times L$	A , C , H	$Tt \times M \times L$
$aC \times Wl \times M2 \times L$	POMEGA2	$aC \times Wl \times M2 \times L$
$aC \times Wl \times M \times M \times L$	B , AA , CC	$Tt \times M \times M \times L$

Table 3.2: Solver - Part 1: Matrices and their sizes before and after the data structure eliminations. aC is the number of air columns, Wl is the number of wavelengths per air column, L is the number of layers per air column, M and M2 are the number of data per air column, per wavelength, per layer and Tt is the total number of threads used to execute a simulation.

Size Before	Matrices	Size After
$aC \times M$	WT , EMU	$aC \times M$
$aC \times Wl$	RFL , ZFLUX , FJTOP , FJBOT	$aC \times Wl$

Table 3.3: Solver - Part 2: Matrices and their sizes before and after the data structure eliminations. aC is the number of air columns, Wl is the number of wavelengths per air column, L is the number of layers per air column, M and M2 are the number of data per air column, per wavelength, per layer and Tt is the total number of threads used to execute a simulation.

Size Before	Matrices	Size After
$aC \times Wl \times M \times M$	E	$Tt \times M \times M$
$aC \times Wl \times L$	FJ	$aC \times Wl \times L$
$aC \times Wl \times M \times L$	A , C , H , RR	$Tt \times M \times L$
$aC \times Wl \times M \times M \times L$	B , AA , CC , DD	$Tt \times M \times M \times L$

3.3 Pre-Calculation of Parts of the Indexes

Redundant calculations can be eliminated. After the elimination the number of data dependences is smaller so as is the number of instructions to analyze. The elimination simplifies the code. The compiler optimizing job becomes easier.

Fast-J has many 2 and 3 nested loops. All these loops run on the number of air-columns, the number of wavelengths per air-column, and the number of layers per air-column. Any of the 4 or 5 dimensional matrices are updated and read more times inside several nested

loops. Pre-calculating part of the access indexes to the matrices, before entering in the next nested loop, explicitly eliminates a great number of otherwise redundant calculations.

Algorithm 6 Pre-Calculation of Parts of the Indexes - Before

```
1: procedure FAST-J(...)
2:     for i = 1 to I do
3:         for j = 1 to J do
4:             for z = 1 to Z do
5:                 C[i × J × Z + j × Z + z] = ...
6:             end for
7:         end for
8:     end for
9: end procedure
```

Algorithm 7 Pre-Calculation of Parts of the Indexes - After

```
1: procedure FAST-J(...)
2:     for i = 1 to I do
3:         i_J_Z = i × J × Z
4:         for j = 1 to J do
5:             j_Z = j × Z
6:             i_J_Z_j_Z = i_J_Z + j_Z
7:             for z = 1 to Z do
8:                 i_J_Z_j_Z_z = i_J_Z_j_Z + z
9:                 C[i_J_Z_j_Z_z] = ...
10:            end for
11:        end for
12:    end for
13: end procedure
```

3.4 Data Locality Enhancement

The code runs faster eliminating useless data transfers. This is accomplished by moving data in local variables and updating the variables many times before updating the data structures in global memory.

It is important to avoid accesses to the global memory as much as possible. An instruction of the form $D[i][\ldots] = \ldots$ updates the data structure D in global memory. Fast-J has many

loops and often the Fast-J code updates the same data many times in a loop. Many Fast-J instructions of the form $D[i][\ldots] = \ldots$ are therefore bad.

Algorithm 8 Data Locality Enhacement - Before

```
1: procedure FAST-J(...)
2:     ...
3:     for i = ... do
4:         for j = ... do
5:             ...
6:             D[i_J_j] =
7:             D[i_J_j] = D[i_J_j] + ...
8:             D[i_J_j] = D[i_J_j] + ...
9:             ...
10:        end for
11:    end for
12:    ...
13: end procedure
```

We can avoid accesses to the global memory modifying many loop instructions. In each loop, for each instruction of the form $D[i][\ldots] = \ldots$, the first time the instruction appears in the loop, move the data $D[i][\ldots]$ to a local variable and update the variable locally, this to avoid the accesses to the global memory. When the last instruction $D[i][\ldots] = \ldots$ appears in the loop, then and only then update the data structure in global memory.

Algorithm 9 Data Localoty Enhacement - After

```
1: procedure FAST-J(...)
2:     ...
3:     for i = ... do
4:         for j = ... do
5:             ...
6:             d = D[i_J_j]
7:             d = d + ...
8:             d = d + ...
9:             D[i_J_j] = d
10:            ...
11:        end for
12:    end for
13:    ...
14: end procedure
```

3.5 Further Reductions

Inside a loop or a group of nested loops there are some redundant instructions, different from those necessary to calculate the indexes. Take for example a two nested loop on i and j where in the j loop the product $E[i] \times F[i]$ is used more times.

Algorithm 10 Further Reductions - Before

```
1: procedure FAST-J(...)
2:     for i = ... do
3:         for j = ... do
4:             ... = ... + E[i] × F[i]
5:         end for
6:     end for
7: end procedure
```

We can only calculate this product one time before entering the j loop and assigning it to a variable - this allows us to keep the result in a register, obtaining several different advantages.

Algorithm 11 Further Reductions - After

```
1: procedure FAST-J(...)
2:     for i = ... do
3:         ef = E[i] × F[i]
4:         for j = ... do
5:             ... = ... + ef
6:         end for
7:     end for
8: end procedure
```

With this solution the cache lines containing $E[i]$ and $F[i]$ will not be hit in the j loop - no further readings - and so a) the cache lines will be available for other data and b) there will be a reduction of the number of transfers necessary among different memory levels.

3.6 Manual Loop Unrolling

We manually unroll all the loop we can. This will save the execution of the instructions of increment, of comparison - they are expensive, and of jump for all the unrolled loops.

Algorithm 12 Manual Loop Unrolling - Before

```
 1: procedure FAST-J(...)
 2:     ...
 3:     for c = 1 to aC do
 4:         for w = 1 to Wl do
 5:             for l = 1 to L do
 6:                 for i = ... do
 7:                     for j = ... do
 8:                         ...
 9:                     end for
10:                 end for
11:             end for
12:         end for
13:     end for
14:     ...
15: end procedure
```

Fast-J has many short 2 level nested loops. These loops read and update the elements of each layer, of each wavelength, of each air column. The number of elements per matrix for each triplet (air column , wavelength , layer) is always in the range 8-16. All these 2 level nested loops can therefore be manually unrolled.

Without the manual unrolling, the compiler cannot efficiently optimize the code. The compiler, in fact, does not know the dimensions of the input data structures at compiling time. The compiler is harder, if not impossible, without this knowledge, so it cannot efficiently optimize the many 2, 3, 4 and 5 level nested loops of the Fast-J code.

Manually unrolling the loops, we greatly simplify the optimization process of the compiler. Loops on the air columns, the wavelengths and the layers have values of the order of magnitude of the thousands or hundreds. These loops cannot be thefore unrolled. However, there are many 2 level nested loops inside the nested loops scanning the air columns, the wavelengths and the layers. These are the loops manualy unrolled to speedup the code.

Algorithm 13 Manual Loop Unrolling - After

```
1: procedure FAST-J(...)
2:     ...
3:     for c = 1 to aC do
4:         for w = 1 to Wl do
5:             for l = 1 to L do
6:                 i = 0
7:                  ...
8:                 j = 0
9:                  ...
10:                j = 1
11:                 ...
12:                i = n
13:                 ...
14:                j = 0
15:                 ...
16:                j = 1
17:                 ...
18:            end for
19:        end for
20:    end for
21:    ...
22: end procedure
```

3.7 Pre-Compiling Define Directives

The greater the compiler knowledge of the code at compiling time, the greater the probability the compiler efficiently optimizes the whole code and the simpler its job of loop transformation and optimization.

For this optimization we built a source code for each simulation we ran. This is accomplished by building a code version for each possible combination of the values of the dimensions of the input data structures.

We declare the dimensions of the input data structures using the pre-compiling directive #define. All the declarations using the pre-compiling directive #define appear at the beginning of each specific version of the code - one version per simulation. At pre-compiling time all the dimensions of the data structures will therefore be substituted with their values. After this, the compiler will see all the dimensions as constants.

Embedding in each code version a possible combination of the values of the dimensions of the input data structures gives the compiler complete knowledge about the values of all the parameters appearing in the code. The compiler can now apply optimizations previously impossible.

Algorithm 14 Variables To Constants Transformation - After

```
1: ...
2: const aC...
3: const Wl...
4: const L...
5: const M...
6: const M2...
7: ...
8: procedure FAST-J(...)
9:     ...
10: end procedure
```

Chapter 4

Experiments and Results

In this chapter we briefly describe the machine we used for the experiments. Next, we explain the precautions used to generate the binary codes. Next we have a section on the particular settings used to run the experiments. We therefore explain why all the results we get are meaningful and accurate. Finally, we show that, counterintuitively, all the results are similar - we will explain why in the next chapter.

4.1 Enviroment

For the experiments we use a CPU Intel Core i7 950 3,06 Ghz LGA 1366 with Kingston DDR3 (3x2Gb) Triple Channel 2 GHz CL9 mounted on an Asus Motherboard P6T SE LGA 1366 X58. The Intel Core i7 has 8 MB of cache L3 and 4 256 KB blocks of cache L2.

4.2 Binary Code Generations

For the generation of the CPU binaries we set to true the -mtune=corei7-avx compiling option. Such a compiling option set the g++ and icc compilers for the production of binary code highly optimized specifically for the Intel Core i7 architecture. The -mtune=corei7-avx in fact enables the 64-bit extensions, and the MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1,

SSE4.2, AVX, AES and PCLMUL instruction set supports.

For each one of the 7 optimizations we generate 8 different binaries. Of these 8 binaries the first 4 are generated using g++ and the -O0, -O1, -O2, and -O3 compiling options. The remaining 4 binaries are generated using icc and the -O0, -O1, -O2, and -O3 compiling options.

4.3 Parametric Choices

We dedicate 4 GB of global memory for the execution of the experiments. The number of wavelengths, the number of layers per air-column, and the number of CPU threads determine the number of air columns fitting in 4 GB of global memory.

Wavelengths per air-column are 8, 16, 32, 64, 128 and 256. When Fast-J runs alone, 8, 16 and 32 wavelengths per air-column are selected. When Fast-J runs inside Cloud-J, 32, 64 or 128 wavelengths are selected. When Fast-J runs inside Cloud-J running inside Solar-J, 128 or 256 wavelengths per air-column are selected.

Layers per air columns go from 300 to 500 included, increasing at steps of 25. The number of layers per air column depends by the number of clouds per air-column and their vertical extension. The greater the number of clouds and shorter their vertical extension, the greater the number of layers per air-column.

We run the CPU binary codes using 4, 8, 16, 32, 64 and 128 CPU threads. The Intel Core-7 has 4 cores, therefore for each execution, 1, 2, 4, 8 , 16 or 32 CPU threads are resident per processor.

4.4 Execution Procedure

We stop all the processes that could interfere with the executions. The lightdm process is terminated to avoid periodic checks and refreshes of the graphic enviroment. We therefore remotly connect using ssh and open a screen session on the machine. We launch the scripts,

detach the screen session and exit from the remote connection.

CPU thread management overhead is minimized. The CPU threads are always created, initialized and set before the calls to the generator and the solver. The whole time necessary for the creation, the initialization and the set of the threads is therefore not counted in the executions.

We run $6 \times 9 = 54$ experiments. 54 is the number of combinations (number of air columns , number of layers per air column). After the runs, for the comparisons, we select the best CPU execution time of each couple (number of wavelengths , number of layers).

The CPU timer has a resolution of half a nanosecond. This happens because we use timespec and create a structure reading an hardware counter. Furthermore, the resolution of the CPU timer has an order of magnitude 8 times smaller than any execution time. All the simulation steps of each experiment in fact require at least several hundreds of milliseconds.

The average running times per experiment are meaningful. For each experiment - (number of wavelengths , number of layers) - we run several sub-experiments (number of treads). Each sub-experiment is run hundred of times to make sure the average running times are meaningful.

The average execution times are accurate. Given a binary code and a sub-experiment, the execution time variability of the binary code, for the sub-experiment, is always smaller than 1% of the average execution time. This proves that no other operative system processes are interfering with the executions.

4.5 Experimental Results

The 8 following tables show the results for the Fast-J core code. There is 1 table per number of wavelengths. The number of layers per air-column is reported on the left of each row. The 7 columns represent the 7 optimized codes. The number in a cell represents the speedup of the best launch configuration of the new CPU optimized code against the state-of-the-art.

Table 4.1: Speedups - 8 wavelengths per air column.

	v1	v2	v3	v4	v5	v6	v7
300	1.23	2.39	3.01	3.39	3.40	4.56	4.56
325	1.14	2.54	3.18	3.59	3.60	4.82	4.83
350	1.21	2.37	2.96	3.35	3.36	4.50	4.50
375	1.24	2.40	3.00	3.40	3.40	4.56	4.57
400	1.22	2.40	3.01	3.39	3.39	4.54	4.54
425	1.23	2.40	3.02	3.41	3.42	4.56	4.56
450	1.22	2.35	2.94	3.34	3.35	4.46	4.46
475	1.24	2.40	3.01	3.40	3.42	4.54	4.54
500	1.22	2.45	3.07	3.49	3.50	4.65	4.65

Table 4.2: Speedups - 16 wavelengths per air column.

	v1	v2	v3	v4	v5	v6	v7
300	1.24	2.40	3.01	3.40	3.40	4.56	4.55
325	1.15	2.53	3.17	3.59	3.61	4.82	4.83
350	1.20	2.38	2.97	3.36	3.37	4.51	4.51
375	1.25	2.41	3.00	3.40	3.41	4.56	4.56
400	1.22	2.42	3.01	3.39	3.39	4.54	4.54
425	1.24	2.39	3.00	3.40	3.42	4.56	4.54
450	1.23	2.36	2.95	3.33	3.35	4.47	4.47
475	1.23	2.41	3.00	3.40	3.42	4.56	4.56
500	1.24	2.46	3.07	3.49	3.49	4.64	4.65

Table 4.3: Speedups - 32 wavelengths per air column.

	v1	v2	v3	v4	v5	v6	v7
300	1.24	2.40	3.01	3.40	3.40	4.55	4.56
325	1.14	2.54	3.19	3.60	3.60	4.84	4.84
350	1.20	2.37	2.96	3.36	3.37	4.51	4.51
375	1.23	2.40	3.01	3.40	3.40	4.55	4.56
400	1.21	2.40	3.02	3.38	3.39	4.53	4.54
425	1.25	2.40	3.02	3.40	3.42	4.55	4.56
450	1.21	2.36	2.94	3.34	3.35	4.47	4.48
475	1.26	2.40	3.01	3.40	3.42	4.56	4.56
500	1.23	2.45	3.07	3.49	3.50	4.65	4.65

Table 4.4: Speedups - 64 wavelengths per air column.

	v1	v2	v3	v4	v5	v6	v7
300	1.25	2.39	3.01	3.39	3.40	4.55	4.56
325	1.12	2.53	3.19	3.59	3.61	4.84	4.84
350	1.22	2.37	2.97	3.36	3.36	4.51	4.51
375	1.25	2.40	3.01	3.40	3.40	4.56	4.56
400	1.23	2.40	3.01	3.39	3.39	4.53	4.54
425	1.21	2.40	3.01	3.39	3.41	4.56	4.56
450	1.24	2.36	2.95	3.34	3.36	4.48	4.47
475	1.26	2.41	3.02	3.41	3.42	4.56	4.55
500	1.25	2.45	3.08	3.48	3.50	4.66	4.64

Table 4.5: Speedups - 128 wavelengths per air column.

	v1	v2	v3	v4	v5	v6	v7
300	1.23	2.40	3.01	3.40	3.40	4.56	4.56
325	1.14	2.54	3.19	3.60	3.60	4.84	4.83
350	1.21	2.37	2.97	3.36	3.37	4.52	4.51
375	1.24	2.40	3.01	3.39	3.41	4.56	4.57
400	1.22	2.41	3.01	3.39	3.40	4.55	4.55
425	1.23	2.40	3.01	3.41	3.42	4.56	4.57
450	1.22	2.35	2.94	3.33	3.35	4.48	4.48
475	1.24	2.40	3.01	3.40	3.41	4.56	4.56
500	1.22	2.45	3.08	3.49	3.50	4.66	4.66

Table 4.6: Speedups - 256 wavelengths per air column.

	v1	v2	v3	v4	v5	v6	v7
300	1.22	2.39	3.01	3.39	3.43	4.56	4.59
325	1.15	2.54	3.18	3.60	3.64	4.84	4.84
350	1.23	2.37	2.97	3.35	3.38	4.52	4.54
375	1.24	2.41	3.01	3.40	3.43	4.56	4.58
400	1.22	2.41	3.01	3.39	3.42	4.54	4.55
425	1.21	2.40	3.02	3.41	3.45	4.56	4.57
450	1.22	2.35	2.95	3.34	3.36	4.46	4.48
475	1.26	2.40	3.00	3.40	3.42	4.56	4.59
500	1.23	2.44	3.08	3.49	3.51	4.66	4.68

Chapter 5

Discussion

The speedups are similar for every simulation, with any number of air columns, number of wavelengths, and number of layers. In this chapter, we explain why this happens and underline the other optimizations contributions.

5.1 Optimization Contributions

Given a cell in a table, we can take a second cell on the same row but to the left of the first cell and divide the speedups of the two cells quantifying the contributions of the optimizations implemented in the code version of the first cell, but not in the code version of the second cell.

Repeating this procedure more times for different cells, the reader can see how 5 of the 7 optimizations are effective because they significantly contribute to the total final speedup for each one of the couples (number of wavelengths per air column , number of layers per air column).

The only 2 optimizations not giving significant speedups are (3.5) and (3.7):

- The optimization (3.5) does not contribute significantly to speedup the code because in the Fast-J core there are not many redundant calculations different from those used

to calculate the indexes - these calculations are eliminated by the optimization (3.3);

- The optimization (3.7) that transforms variables into constants is not useful to the compiler because the total quantity of data required for the data structures necessary to execute the simulations is of the order of the GB and so the compiler is not able to exploit this additional information.

  The optimization (3.7) however remains important for problems using smaller data structures where the compiler can more easily improve and facilitate the reuse of local data.

However, the contribute of each one of the 7 optimizations is nearly stationary for all the different couples (wavelengths per air column , number of layers per air column) proving that each one of the 7 optimizations are robust.

5.2 Optimization Robustness

The speedup of each single optimization for a couple (wavelengths per air column , number of layers per air column) is nearly equal:

- P_1) to the speedups of a couple with a different number of wavelengths per air column but the same number of layers per air column - different table but same cell;

- P_2) to the speedups of a couple with the same number of wavelengths per air column but a different number of layers per air column - same table, different row, but same column.

For P_1 and P_2 then the speedup of each single optimization is nearly stationary for all the different couples (wavelengths per air column , number of layers per air column) - different tables but same column - and so, as consequence, the differences among the speedups of the different single optimizations are nearly stationary in all of the cases.

This is due at I_1) the procedures used to allocate the memory necessary for the data structures, I_2) the layout modification (3.1) and I_3) the Fast-J code:

- I_1) Programmers write multi dimensional matrices in the form $M[D1][D2][...][Dn]$. If they do not allocate dynamically the matrices at runtime then each matrix will become a single mono dimensional array contigous in global memory. This is what we do for the Fast-J code. During the executions data are near and contiguous so it is necessary a smaller number of transfers.

- I_2) Thanks to I_1 and the layout modification optimization (3.1) we a) reduce the number of transfers and b) reduce the quantity of data per transfer.

- I_3) The Fast-J code executes a constant number of instructions per layer per wavelength per air column. These instructions consider different parts of diffedent matrices but are always vector to vector multiplications, additions, and divisions instructions. Increasing the size of the input problems therefore linearly increases the size of all the data structures necessary for the executions and so linearly increases the quantity of necessary calculations.

For I_1, I_2 and I_3, the bigger the size of the input problems, the bigger the quantity of time required for the executions but the rapports among the execution times of the different optimized code versions remain nearly constants - the optimizations are therefore robust.

For codes similar to the Fast-J code, the optimizations will be robust too. The execution times will be different and maybe the contribution of each single optimization will be different too, but the speedups obtained for the codes will be nearly constant too, because the same previous explanation used for Fast-J is true and valid also for the similar codes.

5.3 Effective, Efficient, and Portable Optimizations

The optimizations a) do not consider low level architectural features, b) are always implementable using any high or low level language and c) are applicable to every possible code structure so they will speedup any type of code on any type of architecture.

A programmer can implement any number of optimizations and do it in any order. Each optimization is independent from the others and independently increases the speedup without the risk that its runtime processes interfer with the runtime processes of the others.

Chapter 6

Conclusions And Future Research

In this thesis, we showed how some simple and overlooked source to source optimizations considerably speed up the already state-of-the-art CPU multi-threading important Fast-J code.

The optimization are a) effective because they give substantial speedups also for codes already parallel, b) efficient because they are easy to implement, c) robust because they give nearly constant speedup increases for different numbers of air columns and/or wavelengths and/or layers per air column for different dimensions of the input data structures and d) portable because 1) they are implementable using any high level language without the necessity of knowing undisclosed or hard to quantify architectural features and 2) they are usable for any code on any architecture.

Now, we plan a) to extend the optimization process to the other modules connected to Fast-J, b) port the state-of-the-art CPU multi-threading Fast-J code on GPUs and run it on complex CPU-GPU multiprocessor architectures, and c) implement a GPU version of Solar-J. Let us therefore briefly analyze in more detail each one of these research directions.

The chemistry-climate simulations needed to address SLCFs, long-term caps on CO_2, and future air quality all require a fully functional Cloud-J module (or equivalent) linked to Fast-J and the CAM chemistry codes (e.g., LLNLs fast/super-fast chemistry, CAM-Chem,

WACCM). The results we got can easily be extended to these modules.

To execute very big simulations we will need to use very complex CPU-GPU multi-processor architectures - such as the DOE Titan. The tuning phase on these architectures will be a great challenge. We will need to carefully partition the computations between multiple CPU hosts and their GPUs and we wiil need to efficiently port to GPUs at least some parts of the Fast-J code. To accomplish this, we will consider the sensitivity of GPUs to load balancing issues, we will analyze the CPU-GPU architectural differences, and we will overlap CPU-GPU computations and communications.

Solar-J uses a very high number of wavelengths per air column. This is necessary to execute more accurate simulations. We will implement a GPU Solar-J code at the top of the new Fast-J code to preserve its improved efficiency.

Bibliography

[1] M. B. Arajo, D. Alagador, M. Cabezal, D. N. Bravo1, and W. Thuiller. Climate Change Threatens European Conservation Areas. *Ecology Letters*, 14(5):484–492, 2011.

[2] H. W. Barker, J. J. Morcrette, and G. D. Alexander. Broadband Solar Fluxes and Heating Rates for Atmospheres with 3D Broken Clouds. *Quarterly Journal of the Royal Meteorological Societ*, 124(548):1245–1271, 1998.

[3] J. C. Barnard, E. G. Chapman, J. D. Fast, J. R. Schmelzer, J. R. Slusser, and R. E. Shetter. An Evaluation of the Fast-J Photolysis Algorithm for Predicting Nitrogen Dioxide Photolysis Rates under Clear and Cloudy Sky Conditions. *Atmospheric Environment*, 38(21):3393–3403, 2004.

[4] H. Bian and M. J. Prather. Fast-J2: Accurate Simulation of Stratospheric Photolysis in Global Chemical Models. *J. of Atmospheric Chemistry*, 41:281–296, 2002.

[5] S. N. Collins, R. S. James, P. Ray, K. Chen, A. Lassman, and J. Brownlee. Grids in Numerical Weather and Climate Models. `http://cdn.intechopen.com/pdfs-wm/43438.pdf`, 2013. [Online; accessed 26-Decemberr-2014].

[6] E. V. der Werf and S. Peterson. Modeling Linkages Between Climate Policy and Land Use: An Overview. *CCMP Climate Change Modelling and Policy*, pages 1–34, 2007. [Online; accessed 24-Decemberr-2014].

[7] N. E. S. L. A. C. Division. Community Atmosphere Model with Chemistry. `https://www2.acd.ucar.edu/gcm/cam-chem`. [Online; accessed 26-Decemberr-2014].

[8] U. C. C. Division. International Efforts Focusing on Short-Lived Climate Forcers. `https://www.globalmethane.org/documents/events_steer_101411_openplenary_gunning.pdf`, 2011. [Online; accessed 20-Decemberr-2014].

[9] M. Donatelli, A. Srivastava, G. Duveiller, and S. Niemeyer. Estimating Impact Assessment and Adaptation Strategies under Climate Change Scenarios for Crops at EU27 Scale. *International Congress on Environmental Modelling and Software Managing Resources of a Limited Planet*, pages 1–8, 2012.

[10] A. B. et Al. Integrated Meteorology Chemistry Models: Challenges, Gaps, Needs and Future Directions. *Atmospheric Chemistry and Physics*, pages 317–398, 1014. [Online; accessed 27-Decemberr-2014].

[11] N. C. for atmospheric Research. Main Website. http://ncar.ucar.edu/. [Online; accessed 28-Decemberr-2014].

[12] A. M. Greene, M. Hellmuth, and T. Lumsden. Stochastic Decadal Climate Simulations for the Berg and Breede Water Management Areas, Western Cape province, South Africa. *Water Resources Research*, 48:1–13, 2012.

[13] W. A. W. Group. Whole Atmosphere Community Climate Model. https://www2.cesm.ucar.edu/working-groups/wawg. [Online; accessed 24-Decemberr-2014].

[14] I. Haddeland1, J. Heinke, F. Vob, S. Eisner, C. Chen, S. Hagemann, and F. Ludwig. Effects of climate model radiation, humidity and wind estimates on hydrological simulations. *Hydrology and Earth System Sciences*, 16:305–318, 2012.

[15] J. Hansen, G. Russell, D. Rind, P. Stone, A. Lacis, S. Lebedeff, R. Ruedy, and L. Travis. Efficient Three-Dimensional Global Models for Climate Studies: Models I and II. *American Meteorological Society*, 111(4):609–662, 1983.

[16] M. J. Iacono. Application of Improved Radiation Modeling to General Circulation Models. *Atmospheric and Environmental Research*, pages 1–39, 2011.

[17] A. D. M. C. A. in the Higher Colleges of Technology (HCT). Problem and Solution: Global Warming. http://www.admc.hct.ac.ae/hd1/english/probsoln/prob_solv_gw2.htm. [Online; accessed 21-Decemberr-2014].

[18] R. Ireland. Implications for Customs of Climate Change Mitigation and Adaptation Policy Options: a Preliminary Examination. *World Customs Journal*, 4(2):21–36, 2010.

[19] F. Jiang and C. Hu. Application of Lattice Boltzmann Method for Simulation of Turbulent Diffusion from a CO_2 Lake in Deep Ocean. *J. of Novel Carbon Resource Sciences*, pages 10–18, 2012.

[20] M. A. Katsoulakis, A. J. Majda, and D. G. Vlachos. Coarse-Grained Stochastic Processes for Microscopic Lattice Systems. *Proceedings of the National Academy of Sciences of the United States of America*, 100(3):782–787, 2003.

[21] J. Kukkonen, T. Balk, D. M. Schultz, A. Baklanov, T. Klein, A. I. Miranda, A. Monteiro, M. Hirtl, V. Tarvainen, M. Boy, V. H. Peuch, A. Poupkou, I. Kioutsioukis, S. Finardi, M. Sofiev, R. Sokhi, K. Lehtinen, K. Karatzas, R. S. Jos, M. Astitha, G. Kallos, M. Schaap, E. Reimer, H. Jakobs, and K. Eben. Operational Chemical Weather Forecasting Models on a Regional Scale in Europe. *Atmospheric Chemistry and Physics*, pages 5985–6162, 2011.

[22] J. Kukkonen, T. Olsson, D. M. Schultz, A. Baklanov, T. Klein, A. I. Miranda, A. Monteiro, M. Hirtl, V. Tarvainen, M. Boy, V.-H. Peuch, A. Poupkou, I. Kioutsioukis, S. Finardi, M. Sofiev, R. Sokhi, K. E. J. Lehtinen, K. Karatzas, R. S. Jose, M. Astitha, G. Kallos, M. Schaap, E. Reimer, H. Jakobs, and K. Eben. A Review of Operational, Regional-Scale, Chemical Weather Forecasting Models in Europe. *Atmospheric Chemistry and Physics*, pages 1–87, 2012.

[23] J.-F. Lamarque, D. T. Shindell, B. Josse, P. J. Young, I. Cionni, V. Eyring, D. Bergmann, P. C. Smith, W. J. Collins, R. Doherty, S. Dalsoren, G. Faluvegi2, G. Folberth9, S. J. Ghan12, L. W. Horowitz13, Y. H. Lee2, I. A. MacKenzie10, T. Nagashima14, V. Naik15, D. Plummer16, M. Righi7, S. T. Rumbold9, M. Schulz17, R. B. Skeie11, D. S. Stevenson, S. Strode, K. Sudo, S. Szopa, A. Voulgarakis, and G. Zeng. The Atmospheric Chemistry and Climate Model Intercomparison Project (ACCMIP): Overview and Description of Models, Simulations and Climate Diagnostics. *Geoscientific Model Development*, 6:179–206, 2013.

[24] G. Lu, D. J. DePaolo, Q. Kang, and D. Zhang. Lattice Boltzmann Simulation of Snow Crystal Growth in Clouds. *J. of Geophysical Research: Atmospheres*, 114:1–14, 2009.

[25] M. D. Mastrandrea. Calculating the Benefits of Climate Policy: Examining the Assumptions of Integrated Assessment Models. *Pew Center on Global Climate Change*, pages 1–60, 2009. [Online; accessed 23-Decemberr-2014].

[26] C. E. S. Model. Community Atmosphere Model Num. 5. `http://www.cesm.ucar.edu/working_groups/Atmosphere/development/`. [Online; accessed 27-Decemberr-2014].

[27] C. E. S. Model. Community Climate Model. `https://www2.cesm.ucar.edu/about`. [Online; accessed 30-Decemberr-2014].

[28] W. D. Nordhaus. Managing the Global Commons: The Economics of Climate Change . *MIT Press*, 1994.

[29] M. J. P. O. Wild1 and H. Akimoto1. Indirect Long-Term Global Radiative Cooling from NO_x Emissions. *Geophysical Research Letters*, 28(9):1719–1722, 2001.

[30] C. Obrecht, F. Kuznik, L. Merlier, J.-J. Roux, and B. Tourancheau. Towards Aeraulic Simulations at Urban Scale Using the Lattice Boltzmann Method: Environmental Fluid Mechanics. *Springer Verlag*, pages 1–20, 2014.

[31] D. of Energy and L. B. N. Laboratory. National Energy Research Scientific Computiong Center. `https://www.nersc.gov/`. [Online; accessed 30-Decemberr-2014].

[32] W. on Short Lived Climate Forcers. Addressing Black Carbon and Ozone as Short-Lived Climate Forcers. `http://www.cleanairinfo.com/slcf/documents/Workshop%20Summary%20Web.pdf`, 2010. [Online; accessed 26-Decemberr-2014].

[33] T. F. on ShortLived Climate Forcers. Recommendations to Reduce Black Carbon and Methane Emissions to Slow Arctic Climate Change. *Arctic Council*, page 20, 2011.

[34] I. C. C. Panel. Fifth Assessment Report (AR5). `http://www.ipcc.ch/index.htm`, 2009. [Online; accessed 17-Decemberr-2014].

[35] M. J. Prather. The Fast-J Software. `http://www.ess.uci.edu/group/prather/scholar_software`. [Online; accessed 29-Decemberr-2014].

[36] U. N. E. Programme. Short-lived Climate Forcers and their Impacts on Air Quality and Climate. http://www.unep.org/dewa/Portals/67/pdf/SL_climateforcers_02.pdf, 2012. [Online; accessed 22-Decemberr-2014].

[37] D. Randall, R. Wood, S. Bony, R. Colman, T. Fichefet, J. Fyfe, V. Kattsov, A. Pitman, J. Shukla, J. Srinivasan, R. Stouffer, A. Sumiand, and K. Taylor. Cilmate Models and Their Evaluation. *Climate Change 2007: The Physical Science Basis. Contribution of Working Group I to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change*, pages 1–74, 2007.

[38] E. Real and K. Sartelet. Modeling of Photolysis Rates over Europe: Impact on Chemical Gaseous Species and Aerosols. *Atmospheric Chemistry and Physics*, 11:1711–1727, 2011.

[39] P. Ricoux, J. Y. Berthou, and T. Bidot. European Exascale Software Initiative (EESI2): Towards Exascale Roadmap Implementations. http://www.eesi-project.eu/pages/menu/homepage.php, 2014. [Online; accessed 19-Decemberr-2014].

[40] J. L. Schnoor. Environmental Modeling: Fate and Transport of Pollutants in Water, Air, and Soil. *John Wiley and Sons*, pages 1–682, 1996.

[41] N. U. C. E. System. Community Earth System Model. https://www2.cesm.ucar.edu/models. [Online; accessed 28-Decemberr-2014].

[42] M. Tobis, C. Schafer, I. Foster, R. Jacob, and J. Anderson. FOAM: Expanding the Horizons of Climate Modeling. *ACM/IEEE Conference on Supercomputing*, pages 1–27, 1997.

[43] K. Tourpali, A. F. Bais, A. Kazantzidis, C. S. Zerefos, H. Akiyoshi, J. Austin, C. Bruhl, N. Butchart, M. P. Chipperfield, M. Dameris, M. Deushi, V. Eyring, M. A. Giorgetta, D. E. Kinnison, E. Mancini, D. R. Marsh, T. Nagashima, G. Pitari, D. A. Plummer, E. Rozanov, K. Shibata, and W. Tian. Clear Sky UV Simulations for the 21st Century based on Ozone and Temperature Projections from Chemistry-Climate Models. *Atmospheric Chemistry and Physics*, pages 1165–1172, 2009.

[44] G. Wiki. Fast-J Photolysis Mechanism. http://wiki.seas.harvard.edu/geos-chem/index.php/FAST-J_photolysis_mechanism, 2014. [Online; accessed 29-Decemberr-2014].

[45] O. Wild, X. Zhu, and M. J. Prather. Fast-J: Accurate Simulation of in- and below- Cloud Photolysis in Tropospheric Chemical Models. *J. of Atmospheric Chemistry*, 37:245–282, 2000.