Fast Fast-J GPU Codes

Fausto Artico, Michael J. Prather, Alexander V. Veidenbaum, Alexandru Nicolau

Center for Embedded and Cyber-Physical Systems

University of California, Irvine

Irvine, CA 92697-2620, USA

{fartico, mprather, aveidenb, anicolau}@uci.edu

CECS Technical Report 15-03

November 13, 2015

Fast Fast-J GPU Codes

Fausto Artico
Dep. of Computer Science
6210 Donald Bren Hall
Irvine, 92697, CA, USA
fartico@ics.uci.edu

Alex V. Veidenbaum
Dep. of Computer Science
6210 Donald Bren Hall
Irvine, 92697, CA, USA
alexv@uci.edu

Michael J. Prather
Dep. of Earth System Science
3200 Croul Hall St
Irvine, 92697, CA, USA
mprather@uci.edu

Alex Nicolau
Dep. of Computer Science
6210 Donald Bren Hall
Irvine, 92697, CA, USA
alex.nicolau@uci.edu

ABSTRACT

Chemistry Climate Model (CCM) numerical codes are important [68] to understand how to mitigate global warming [36, 35]. In order to produce meaningful results, CCM numerical codes require PetaFlop-scale performance and soon will require ExaFlop-scale performance [62]. Furthermore, such high-performance must be reached within a reasonable power budget. It is therefore important to speedup the executions of the state-of-the-art CCM codes and to make them more energy efficient. These codes are already optimized for CPUs but not for Graphics Processing Units (GPUs) [47, 74], which are the best candidates to achieve the above mentioned performance [39] and power [33] budget goals.

Among CCM codes, Fast-J [58] is very important and widely used in simulations at different scales of magnitude, i.e., local, global and cosmic. At any scale of simulation there is a core code called Fast-J core, which determines the performance of the simulation. In this paper we speedup the Fast-J core. To accomplish such goal, first the Fast-J core is ported from its highly-optimized CPU version to a GPU version. Second, a few high-level optimizations are identified and deployed to the GPU version of Fast-J. Some of the high-level optimizations are not currently part of the CUDA compiler and others are not efficiently implemented.

The newly ported and optimized GPU Fast-J core kernels are 50.0 and 25.8 times faster than the already highly optimized CPU multi-threaded code. Furthermore, speedups of at least 15.5 and 13.6 are consistently reached for each scale of simulation.

Categories and Subject Descriptors

G.4 [**Mathematics of Computing**]: Mathematical Software—*Parallel Implementations - Graphics Processing Units*

General Terms

Performance, Algorithms, Measurement, Experimentation

Keywords

HPC, Performance, Speedup, GPU, Optimization, Compiler, CCM, GCM, Fast-J, Green, CUDA, Nvcc, PTX, Chemistry Climate Model, Global Chemistry Model, G++, Icc

1. INTRODUCTION

Climate change has been a very active area of research in the last few decades. In order to address the problems presented by climate changes, the fields of Computer Science and Earth System Science need to develop and validate more accurate chemistry climate models (CCMs) and to improve their performance, in order to produce a qualitative difference in the results of the complex, large scale simulations involved.

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 [63] [63, 42], using 1D [40], 2D [44, 38] or 3D [52, 3] lattices. Such numerical models, when implemented, highly optimized, and accelerated, allow scientists to forecast the arrival of dangerous weather conditions [22, 41] - i.e. hurricanes [60] - and give insights on how to mitigate dangerous climate phenomena such as global warming [72, 32].

The CCM and the volumetric size of the simulation determine the density (number of points) and the homogeneity of the lattice (which accounts for that elementary volumes of lattice 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 update the values of some variables - i.e wind and humidity - at each one of its lattice points, and if necessary, correct them [31]. The processing nodes therefore propagates the results to their neighbor nodes at the end of each simulation step. The number of simulation steps depends from the lattice size and the temporal horizon of the simulation [66]. The lattice of simulations, studying the causes of warming phenomenons, easily cover many countries [2] or even the whole earth [9]. The temporal horizon of the simulations is usually in decades [29, 21] or centuries [17, 46, 51].

The numbers of variables to update, the numbers of points composing the lattices used by the simulations and the temporal horizons of the simulations are translated into costly and very time consuming computations. Such costs are large for any class of general purpose computers, including supercomputers. Hence, speeding up such simulations is paramount to advancing their state-of-the-art forecasting possibilities.

While the simplification of the CCM is possible, it is not a solution to the problem. In fact current radiative transfer (RT) models (e.g., the Rapid Radiative Transfer Model for Global Chemistry Models (RRTM-G) [34] as used by the National Center of Atmospheric Research (NCAR) [26] and in the Department of Energy Community Earth System Model (DOE CESM) [50]) make simplifying 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 approximations create bias errors in the modeling of photochemistry, heating rates, and the distribution of photo synthetically active radiation (PAR). However, the greater their accuracy, the greater the understanding on how 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 [55, 54]. One way to achieve greater accuracy is increasing the speeds of the codes. The faster the CCM codes, the greater the achievable accuracies.

An important CCM is Fast-J [73, 6, 71] [73, 6, 61, 71]. Fast-J is used to study a) the short-lived climate forcing agents that [59, 20] are responsible for slow global warming [59] and decreased air quality [20], and b) CO_2 concentrations in the atmosphere [4] that have to be kept within the bound of $2\,^\circ$C [IPCC, 2013].

Fast-J is widely used by several CCMs such as the Community Atmosphere Model number 5 (CAM5) [49] running at the National Energy Research Scientific Computing Center (NERSC) [53]. Fast-J is also integrated in several of the chemistry-climate models discussed in the Intergovernmental Panel on Climate Change, the 5th Assessment Report (IPCC AR5 [57]), the Oslo-CTM2, the GESOCCM and the GISS-ER2 [23]. The Whole Atmosphere Community Climate Model (WACCM) [30] and the Community Atmosphere Model with Chemistry (CAMWC) [19] use Fast-J too.

Shortening the running time of the Fast-J code is paramount. Fast-J currently runs a total of at least 10 million hours/year worldwide (likely an under-estimate). Even in super-computers using dedicated and specialized multi-core architectures, each execution of the Fast-J core code per simulation requires months for any meaningful forecasting, in spite of the fact that the code is already highly optimized for such architectures. In this article we accelerate the Fast-J core to improve its energy efficiency, accelerate the simulations, and increase their accuracy. First, we port the state-of-the-art, multi-threaded, multi-air column, multi-wavelength and multi-layer CPU source code to NVIDIA GPU architectures, and second, we customize the ported code using high-level optimizations.

The source-to-source optimizations that we identify and propose are currently not implemented in or efficiently executed by the NVIDIA compiler. One of the contributions of this paper is in fact the identification and efficient implementation of such optimizations to speedup GPU code executions without using intrinsics or assembly, which are tedious and error-prone. Such optimizations, which can be implemented as part of a source-to-source compiler, are important because they allow the delivery of performing and portable codes.

There are 7 source-to-source optimizations: 1) porting and linearization; 2) changing the data layout; 3) reducing the necessary number of data structures; 4) loop invariant removal; 5) scalar replacement of aggregates; 6) declaring the number of GPU threads and the dimensions of the data structures using pre-compiling directives; and 7) defining the previous variables that are now transformable into constants, after loop unrolling, at a pre-compilation time.

The 7 source-to-source optimizations make the 2 Fast-J core kernels, when running on a NVIDIA Tesla 2070 GPU, 50.0 and 25.8 times faster than the state-of-the-art, multi-threaded, multi-air column, multi-wavelength and multi-layer CPU Fast-J core kernels running on the Intel Core i7. The newly optimized GPU Fast-J core code therefore opens research opportunities that were previously impossible. Furthermore, for any scale of simulation, independent from the number of air columns, the number of wavelengths per air column, and the number of layers per air column, the new Fast-J core kernels are always at least 15.5 and 13.6 times faster than the state-of-the-art, multi-threaded, CPU code.

2. GRAPHICS PROCESSING UNITS

Graphics Processing Units (GPUs) are specialized hardware originally created to accelerate computer graphics and image processing [47]. However, their highly parallel structure [27] and their low cost per GFlop per Watt [64] make them attractive as energy efficient, performing architectures, which can be used to accelerate more general purpose, computationally-intensive scientific tasks [56]. In June 2014, the first 15 of the 500 most energy efficient computers on earth all used NVIDIA state of the art Kepler GPUs [1].

Accelerating codes on GPUs is difficult [69]. This is due to the fact that GPUs, when compared to CPUs, have many more functional units (hundreds versus less than 10 [14, 12]), less hierarchy memory levels (two instead of three [15, 11]) and smaller cache memories (an L2 cache of 256 KB instead of an L2 cache of 2 MB [13, 16]). Furthermore GPUs are optimized for high parallel arithmetic intensity instead of for branching sequential code like CPUs [28].

Sophisticated GPU compilers have yet to be developed. Such a compiler would have a more difficult time in optimizing and transforming code than a corresponding CPU compiler [10]. This is due to the difficulties in solving synchronization problems, data movements and data dependencies for thousands of threads.

Currently the only way to achieve peak performance in GPUs is to hand-optimize the code, which is only possible to a limited extent because the NVIDIA Instruction Set Architectures for all the NVIDIA GPUs post-2010 are undisclosed. The availability of a source-to-source compiler therefore assumes particular relevance in the context of optimizing GPU code. While not all of the possible source-to-source optimizations can be relevant, the few accurately selected source-to-source optimizations give significant and consistent speedups.

3. THE CPU STATE-OF-THE ART CODE

The CPU state-of-the art C++ Fast-J core code is already multi-threading, multi-air columns, multi-wavelengths, multi-layers, and multi-air columns. Typical values for the number of wavelengths are 8, 16, 32, 64, 128, and 256. Typical values for the number of layers are 300, 400, and 500.

The core code has 2 main functions: the generator of the triangular systems and the solver. The generator and the solver contribute to 17% and 80% 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.

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. These matrices are large. The matrices with 4 or 5 dimensions have: one dimension for the air columns - in range of thousand elements; one a dimension for the wavelengths - in the range of hundred elements; and one dimension for the layers - in the range of hundred elements.

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

4. SOURCE-TO-SOURCE OPTIMIZATIONS

We introduce and implement a total of 7 source-to-source optimizations. Each optimization is implemented on top of the previous one. The optimizations are as follow:

1) Porting from CPU to GPU and Linearization: We ported the code in such a way that it now works using any number of GPU threads. Usually, thousands of threads are needed to execute a generic GPU code. For this reason, it was necessary to introduce into the original state-of-the-art CPU code the necessary controls for the distribution of the data structures among GPU threads.

Synchronization among GPU threads is eliminated because it is very time consuming. We take care to port the code so that each GPU thread reads and updates only specific parts of the data structures, with the guarantee that no other thread will try to read or update such parts simultaneously. We implemented the necessary communication infrastructure between the CPU and the GPU. GPU code always has to be called inside CPU code. Using this infrastructure, the CPU passes the control to the GPU when necessary and waits each time to regain control before continuing with the execution of the CPU code.

CPU data structures were duplicated in the GPU global memory. We declared the necessary GPU pointers and allocated the necessary quantity of GPU global memory, copied the CPU data structures to the GPU data structures residing in the GPU global memory, prepared the stack for the execution of the GPU code, executed the GPU code, and transfered the results back to the CPU.

Anyway linearization is used. Using $A[i \times J \times Z \times W + j \times Z \times W + z \times W]$ instead of $A[i][j][z]$ 1) simplifies the implementation of the other optimizations, i.e. data layout modification, 2) allows us to implement some optimizations in particular ways, i.e. loop invariant removal, and 3) allows some optimizations to gain greater speedups, i.e. data locality enhancement.

2) GPU Layout Modification: Data layout modification reduces the probability of hitting bandwidth bottlenecks. If a code hits a bottleneck then its execution will be slowed down. To reduce the probability that this happens, the data layout therefore has to be changed in a way that reduces the average quantity of bytes transfered per clock cycle.

Transfers can easily kill any performance improvement. The GPU schedulers schedule 28 groups of 32 threads per clock cycle. If the data required by a group of threads are not consecutive in the global memory, then the architecture has to transfer many cache lines to satisfy the request of a single group (in the worst case scenario: 64 cache lines, each of which has 128 bytes per group, producing $28 \times 64 \times 128 = 229376$ bytes for the groups). In this case, even a single group request can easily amount to more than 128 bytes per clock cycle, therefore killing any possible improvement (128 bytes per clock cycle is the bandwidth between off-chip and on-chip memory for a NVIDIA Tesla C2070).

The data per group request has to be consecutive in the global memory. If the data are consecutive, then the architecture will transfer the minimum number of cache lines. For example, supposing the data requested by a group of threads are 1) different, 2) 8 bytes each, 3) consecutive in the global memory, and are 4) not present in the small caches on-chip, then only 2 cache lines will be transfered instead of the previous 64.

The overhead time due to the layout modification is practically null. The layout has to be changed only one time before the execution of the simulation. A CCM simulation requires millions or billions of time steps and so weeks or months of execution time, but the layout modification requires only a few seconds in the worst case.

3) GPU Data Structure Eliminations: The smaller the data structures, the smaller the probability of bottlenecks being 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 reuse of the same parts of some data structures at different moments during the executions. After this optimization, it is now possible to execute simulations 7 times larger than before.

Size Before	Matrices
$aC \times M$	WT , EMU
$aC \times Wl$	RFL
$aC \times Wl$	ZTAU, ZFLUX
$aC \times Wl$	FJTOP , FJBOT
$aC \times M2$	PM0
$aC \times Wl \times M \times M$	E
$aC \times Wl \times L$	FJ
$aC \times Wl \times M \times M$	S, T , U , V , Z
$aC \times Wl \times M \times M2$	PM
$aC \times Wl \times M \times L$	A , C , H , RR
$aC \times Wl \times M2 \times L$	POMEGA2
$aC \times Wl \times M \times M \times L$	B , AA , CC , DD

Table 1: Matrices and their sizes before 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 (M and M2 are always equal to 4 and 8), while Tt is the total number of threads used to execute a simulation.

Size After	Matrices
$aC \times M$	WT , EMU
$aC \times Wl$	RFL
$aC \times Wl$	ZTAU, ZFLUX
$aC \times Wl$	FJTOP , FJBOT
$aC \times M2$	PM0
$Tt \times M \times M$	E
$aC \times Wl \times L$	FJ
$Tt \times M \times M$	S, T , U , V , Z
$aC \times Wl \times M \times M2$	PM
$Tt \times M \times L$	A , C , H , RR
$aC \times Wl \times M2 \times L$	POMEGA2
$Tt \times M \times M \times L$	B , AA , CC , DD

Table 2: Matrices and their sizes after the data structure eliminations. See Table 1 for an explanation of the symbols aC, Wl, L, M, M2, and Tt.

4) GPU Loop Invariant Removal: Redundant calculations can be eliminated. Fast-J has many 2 - and 3 - nested loops. All these loops run on a variable number of air columns, wavelengths, and layers per air column. Any of the 4 or 5 dimensional matrices are updated and read several times inside several nested loops. Pre-calculating parts of the access indexes to the matrices, before entering into the next nested loop, explicitly eliminates a great number of otherwise redundant calculations.

```
1: procedure FAST-J(...)
2:     for i = 1 to I do
3:         $i\_J\_Z = i \times J \times Z$
4:         for j = 1 to J do
5:             $j\_Z = j \times Z$
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:                 $C[i\_J\_Z\_j\_Z\_z] = \ldots$
10:            end for
11:        end for
12:    end for
13: end procedure
```

Algorithm 1: GPU Loop Invariant Removal

5) GPU Scalar Replacement of Aggregates: The code runs faster after eliminating useless data transfers. This is accomplished by moving data into local variables and updating them many times before updating the data structures in the global memory.

It is important to avoid accessing the global memory as much as possible. An instruction of the form $A[i][\ldots] = \ldots$ updates the data structure A in the global memory. The Fast-J code has many loops and often updates the same data many times in a loop. Many Fast-J instructions of the form $A[i][\ldots] = \ldots$ are therefore inefficient.

We avoid accessing the global memory modifying loop instructions. In each loop, for each instruction of the form $A[i][\ldots] = \ldots$ when it first appears in the loop, we move the data $A[i][\ldots]$ into a local variable and update the variable locally. This avoids frequent accessing of the global memory. When the last instruction $A[i][\ldots] = \ldots$ appears in the loop, then and only then do we update the data structure in the global memory.

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

Algorithm 2: Scalar Replacement of Aggregates

6) GPU Pre-Compiling Define Directives for GPU Threads and Data Structures: More information makes the compiler's job easier. We greatly simplify the compiler's loop transformation and optimization tasks using #define directives to provide to the compiler the number of GPU threads and the dimensions of all the data structures at a pre-compiling time.

Loop transformations and optimizations are important for Fast-J because Fast-J has a great number of nested loops. Once it knows the dimensions of all the data structures, the compiler acquires complete knowledge of the parameters necessary to decide if and how to split or combine loops, interchange or permute indexes, skew loops, or apply tiling and other optimizations.

7) GPU Constant Folding: The smaller the number of instructions to execute, the smaller a code's execution time. Manually unrolling the loops, with the previous optimizations, makes many previous calculations transformable into constants. The constants are therefore inserted into the code using some #define directives. This transformation contributes to further speedup the already highly optimized GPU code. This is due to the fact that Fast-J has many short 1 - and 2 - level nested loops.

5. HARDWARE AND SOFTWARE

Hardware configuration: the experiments 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 connected through PCI-Express v. 2 to a NVIDIA's Tesla C2070 GPU. The Intel Core i7 has 8 MB of L3 cache and 4 256 KB blocks of L2 cache. The NVIDIA's Tesla C2070 does not have an L3 cache and has only 676 KB of L2 cache.

Software environment: the operative system is Ubuntu 14.04. The g++ version is 4.8.2 . The icc version is 15. The CUDA driver version is 7.5 . The CUDA compiler version is 7.5.12. All these components are the current state-of-the-art - August 2015.

6. BINARY CODE GENERATIONS

We generate 16 different binaries for the CPU state-of-the-art Fast-J core code, 8 using g++ and 8 using icc - 4 for the generator and 4 for the solver per compiler. For each one of the 2 functions the 4 binaries generated by each compiler were generated using the -O0, -O1, -O2, and -O3 compiling options.

For the GPU we generate a total of X binary codes. This is due to the last 2 source to source transformations, the transformation using the pre-compiling define directives and constant folding. To understand why this happens let us briefly analyze the 7 archetype GPU codes that we use for the binary generations.

There are only 2 specific GPU codes for each one the first 5 archetype GPU codes - 5 for the generator and 5 for the solver. The 2 specific GPU codes number 1 are the ports with linearization. The 2 specific GPU code number 2 are the 2 specific GPU codes number 1 with the addition of layout modification. The 2 specific GPU codes number 3 are the 2 specific GPU codes number 2 with the addition of the data structure eliminations. The 2 specific GPU codes number 4 are the 2 specific GPU codes number 3 with the addition of the pre-calculation of parts of the indexes. The 2 specific GPU codes number 5 are the 2 specific GPU codes number 4 with scalar replacement of aggregates. These 10 GPU codes - 5 per function - are unique for any possible experiment and any possible number of GPU threads.

There are instead many specific 6th and 7th GPU codes. The number of air columns, the number of wavelengths per air column, the number of layers per air column, the number of threads executing a simulation and the quantity of global memory to dedicate to the simulation uniquely determine the dimensions of the data structures. All these features has to be embedded in the codes 6th and 7th. Decided the quantity of global memory to dedicate to a simulation, the number of wavelength per air column, the number of layers per air column and the number of threads to use for the executions, we need to produce a specific 6th GPU code for all the possible combinations of the values of the previous parameters - we take the specific GPU code 5 of each function and specialize it for each one of the possible combinations. We also need to generate a specific 7th GPU code for each one of the 6th GPU codes. To accomplish this we take a specific 6th GPU code, we manually unroll short loops, transform some previous variables in constants, eliminate the redundant calculations corresponding to the previous variables now constants, and embed the constants in the new code using some pre-compiling directives.

7. COMPILER OPTIONS

For the generation of all the CPU binaries we set to true the -mtune=corei7-avx compiling option. The -mtune=corei7-avx option set the g++ and icc compilers for the production of binary code highly optimized specifically for the Intel Core i7 architecture. With this option we enable the 64-bit extensions, and the MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AES and PCLMUL instruction set supports.

For the generation of the GPU binaries we use the $-arch = compute\_20$ and $-code = sm\_20$ compiling options. The $-arch$ compiling option specifies to the compiler the abstract type of the target architecture. This is necessary to the compiler for the production of a more specific and optimized PTX code - PTX is a pseudo virtual assembly used by NVIDIA to make CUDA codes portable between different NVIDIA's GPU architectures. The $-code$ compiling option instead specifies to the compiler the real target architecture for the production of the optimized assembly code obtained taking in input the PTX code. We do not use the $-m = 64$ compiling option. This is because the GPU codes are automatically produced and optimized for the 64 bit architecture.

For GPUs no other GPU compiling optimizing options can be specified beyond the $-arch$, $-code$ and $-m$ options. We can use the $-Xptxas = -v$ option to get some insights on the number of hardware registers each thread will use during the executions or the number of bytes dedicated to the stacks but the option does not further optimize the code, simply return some binary statistics.

8. OTHER SYSTEM SETTINGS

We dedicated 4 GB of global memory on the CPU and on the GPU for the execution of the experiments. The number of wavelengths and the number of layers per air columns and the number of CPU or GPU threads used 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 wavelength per air columns 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 longer their vertical extension, the greater the number of layers per air column.

For the CPU experiments 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.

For the GPU experiments we run the GPU binary codes using a number of blocks of threads always equal to the number of stream multiprocessors - 14 for a NVIDIA's Tesla C2070 - and a number of warps per block going from 1 to 16 - with 16 we reach the maximum occupancy in number on hardware registers per stream multiprocessor.

The use of a number of blocks of threads equal to the number of stream multiprocessor is important. Doing this we minimize the overheads due to the block assignments and management during the executions. Blocks will be generated and assigned only one time at the beginning of the simulation and will remain resident in the same stream multiprocessor for the whole duration of the execution. Each stream multiprocessor will manage the minimum number of blocks - 1 - with consequent time saving but full occupancy of the stream multiprocessor resources.

9. EXPERIMENTAL PROCEDURES

We stop all the processes that could interfere with the executions. The lightdm process is terminated to avoid period checks and refreshes of the graphical environment by the GPU. We therefore connect from remote using ssh and open a screen session on the machine. We launch the scripts, detach the screen session and exit from the remote connection.

We run $6 \times 9 = 54$ experiments per function. 54 is the number of combinations (number of air columns , number of layers per air column). We dedicate 4 GB of global memory and to each experiment.

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 for the execution of the 2 functions.

The communication overhead between CPU and GPU and the GPU overhead for thread generation and assignment are included in each GPU execution time. The GPU timer is started just before the CPU passes the control to the GPU. The GPU, got the control, generates the GPU threads, assignes the GPU threads to the stream multiprocessors, and execute the code. After the last thread has completed its execution, the GPU returns the control to the CPU and the GPU timer stopped.

For each function, for each couple (number of wavelengths , number of layers), we select for the comparisons the best CPU and GPU average running times. Usually, the best CPU average running time, per (number of wavelengths , number of layers), per function, are runs using 8 CPU threads, while for the GPU are runs using launch configurations with 16 warps per thread block.

CPU and GPU timers have nanosecond and microsecond resolutions. For the CPU we use timespec and create a structure reading hardware counters. For the GPU we use the cudaEventRecord() and the cudaEventSynchronize() functions.

CPU and GPU timer resolutions are 8 and 5 order of magnitude smaller than any execution time. All the CPU simulation steps of each experiment require at least several hundreds of milliseconds - CPU timer resolution is of nanoseconds. All the GPU simulation steps of each experiment require at least several tents of milliseconds - GPU timer resolution is of microseconds.

The CPU and GPU average running times are meaningful. For each experiment - (number of wavelengths,number of layers) - we run several sub-experiments - (number of threads). Each sub-experiment is run hundred of times.

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

10. FINAL RESULTS

Tables 3 and 4 show the speedups of the different GPU codes against the basic *ported* code c_1^y (the code produced by the source-to-source transformation number 1). The first column of each table represents the number of layers per air column, L. The remaining columns c_x^y represent the GPU codes. For example, c_3^8 represents the GPU code for simulations that 1) use air columns with 8 wavelengths, and that 2) are accelerated using the first three optimizations: linearization, layout modification, and loop invariant removal. Note therefore that each column represents the results of applying a new optimization on top of the prior ones.

The greatest speedup contributions, and thus the greatest energy efficiencies are given by the data layout modification (roughly 9), the scalar replacement of aggregates (roughly 4), and by the use of pre-compiling #define directives (roughly 9). While linearization, data structure elimination and loop invariant removal give smaller contributions, they are nevertheless fundamental to the effectiveness of the other performing optimizations. For example, if linearization is absent, the data layout modification and the loop invariant removal cannot be implemented as described.

The 9 times speedup due to the data layout modification shows that a data per warp request must be contiguous in the global memory. This is because of the architectural behaviors explained in section 4.

L/C	c_2^8	c_3^8	c_4^8	c_5^8	c_6^8	c_7^8
300	10.34	13.01	13.03	14.03	23.50	**23.53**
325	10.59	14.07	14.11	14.12	23.63	**23.69**
350	10.20	13.53	13.63	13.99	23.21	**23.25**
375	10.48	13.69	13.81	14.52	24.30	**24.36**
400	10.55	13.10	13.21	14.01	23.74	**23.77**
425	11.86	12.69	12.89	13.84	23.65	**23.69**
450	11.43	12.18	12.37	13.33	22.85	**22.89**
475	10.22	13.87	13.93	13.96	22.83	**22.87**
500	11.94	13.63	13.78	13.89	22.69	**22.75**

L/C	c_2^{16}	c_3^{16}	c_4^{16}	c_5^{16}	c_6^{16}	c_7^{16}
300	11.66	13.00	13.33	14.00	23.50	**23.53**
325	11.82	14.07	14.12	14.16	23.62	**23.64**
350	11.88	13.52	13.76	13.96	23.20	**23.24**
375	12.14	13.78	13.82	14.52	24.34	**24.37**
400	12.11	13.06	13.32	14.10	23.75	**23.79**
425	11.87	12.68	12.79	13.84	23.65	**23.68**
450	11.43	12.19	12.34	13.34	22.82	**22.85**
475	11.67	13.89	13.93	13.98	22.85	**22.89**
500	11.95	13.63	13.75	13.89	22.70	**22.73**

L/C	c_2^{32}	c_3^{32}	c_4^{32}	c_5^{32}	c_6^{32}	c_7^{32}
300	12.02	13.01	13.12	14.00	23.50	**23.52**
325	12.18	14.04	14.06	14.08	23.55	**23.57**
350	11.90	13.54	13.72	13.96	23.21	**23.23**
375	12.44	13.82	13.89	14.55	24.41	**24.44**
400	12.13	13.05	13.12	14.09	23.75	**23.77**
425	11.86	12.69	12.78	13.84	23.64	**23.69**
450	11.44	12.18	12.34	13.33	22.83	**22.85**
475	12.20	13.90	13.94	14.01	22.89	**22.92**
500	11.95	13.62	13.73	13.88	22.70	**22.73**

L/C	c_2^{64}	c_3^{64}	c_4^{64}	c_5^{64}	c_6^{64}	c_7^{64}
300	12.04	13.01	13.14	14.02	23.53	**23.56**
325	12.20	14.04	14.09	14.11	23.53	**23.57**
350	12.07	13.57	13.63	13.97	23.28	**23.32**
375	12.50	13.79	13.87	14.54	24.40	**24.44**
400	11.90	13.06	12.15	14.08	23.73	**23.74**
425	11.85	12.68	12.73	13.82	23.62	**23.65**
450	11.43	12.18	12.23	13.31	22.80	**22.82**
475	12.12	13.93	14.00	14.02	22.94	**22.97**
500	11.95	13.64	13.74	13.90	22.72	**22.73**

L/C	c_2^{128}	c_3^{128}	c_4^{128}	c_5^{128}	c_6^{128}	c_7^{128}
300	12.05	13.01	13.12	14.03	23.55	**23.59**
325	12.19	14.05	14.07	14.08	23.54	**23.58**
350	12.30	13.71	13.78	14.34	23.56	**23.61**
375	12.54	13.79	13.81	14.55	24.40	**24.43**
400	11.90	13.06	13.12	14.08	23.73	**23.76**
425	11.81	12.62	12.73	13.75	23.55	**23.57**
450	11.44	12.18	12.21	13.33	22.82	**22.85**
475	12.23	13.94	13.97	14.02	22.99	**23.03**
500	11.86	13.74	13.83	14.00	22.86	**22.89**

L/C	c_2^{256}	c_3^{256}	c_4^{256}	c_5^{256}	c_6^{256}	c_7^{256}
300	12.04	13.01	13.07	14.02	23.54	**23.59**
325	11.98	13.71	13.75	13.78	22.99	**23.05**
350	12.33	13.68	13.77	14.35	23.58	**23.61**
375	12.53	13.54	13.60	14.62	24.14	**24.17**
400	12.06	13.02	13.14	14.01	23.66	**23.69**
425	11.75	12.54	12.63	13.64	23.39	**23.42**
450	11.35	12.08	12.11	13.20	22.60	**22.64**
475	12.24	13.92	13.97	14.02	22.98	**23.02**
500	12.07	13.71	13.83	14.03	22.96	**22.99**

Table 3: Speedups - Generator - GPU vs GPU - 8, 16, 32, 64, 128 and 256 wavelengths per air column.

L/C	c_2^8	c_3^8	c_4^8	c_5^8	c_6^8	c_7^8
300	9.28	9.43	9.50	13.69	21.75	**22.69**
325	9.83	9.94	10.12	14.40	23.82	**24.51**
350	8.59	8.94	9.05	13.34	21.81	**22.37**
375	10.11	10.58	10.70	15.10	24.18	**25.25**
400	8.94	9.29	9.36	13.88	21.90	**22.78**
425	9.42	9.86	9.93	14.08	23.23	**23.93**
450	9.68	9.83	10.00	14.11	23.03	**23.58**
475	9.55	9.63	9.88	13.77	22.93	**23.40**
500	9.33	9.47	9.66	13.56	22.54	**23.06**

L/C	c_2^{16}	c_3^{16}	c_4^{16}	c_5^{16}	c_6^{16}	c_7^{16}
300	9.48	9.59	9.67	13.92	21.95	**22.86**
325	9.95	10.09	10.29	14.49	23.92	**24.60**
350	9.47	9.67	9.85	13.62	22.39	**23.00**
375	11.02	11.16	11.22	15.88	25.34	**26.45**
400	10.03	10.17	10.23	14.53	23.02	**23.98**
425	10.04	10.22	10.36	14.48	23.57	**24.15**
450	9.81	9.96	10.11	13.97	22.97	**23.55**
475	9.52	9.66	9.83	13.57	22.12	**22.61**
500	9.51	9.67	9.82	13.52	22.33	**22.87**

L/C	c_2^{32}	c_3^{32}	c_4^{32}	c_5^{32}	c_6^{32}	c_7^{32}
300	9.86	10.00	10.12	14.28	22.43	**23.39**
325	10.51	10.70	10.83	15.03	24.43	**25.16**
350	9.78	9.95	10.07	13.86	22.98	**23.73**
375	11.37	11.50	11.55	16.22	25.66	**26.97**
400	10.36	10.48	10.55	14.86	23.25	**24.30**
425	10.38	10.56	10.68	14.82	24.05	**24.79**
450	10.04	10.17	10.29	14.14	23.42	**24.14**
475	9.74	9.83	9.98	13.69	21.74	**22.30**
500	9.68	9.92	10.04	13.55	22.52	**23.06**

L/C	c_2^{64}	c_3^{64}	c_4^{64}	c_5^{64}	c_6^{64}	c_7^{64}
300	10.13	10.28	10.39	14.65	23.29	**24.45**
325	10.67	10.83	10.96	15.23	24.83	**26.00**
350	10.04	10.14	10.33	14.10	23.52	**24.44**
375	11.84	12.00	12.11	16.69	26.33	**27.50**
400	10.76	10.88	10.98	15.29	24.52	**25.82**
425	10.63	10.69	10.93	15.13	24.59	**25.58**
450	10.43	10.61	10.73	14.60	24.84	**25.83**
475	10.06	10.17	10.37	13.91	22.71	**23.11**
500	10.11	10.23	10.43	13.90	23.51	**24.08**

L/C	c_2^{128}	c_3^{128}	c_4^{128}	c_5^{128}	c_6^{128}	c_7^{128}
300	10.27	10.38	10.54	14.64	22.93	**24.10**
325	10.90	11.11	11.24	15.57	25.69	**26.99**
350	10.12	10.30	10.47	14.25	24.10	**25.34**
375	12.07	12.23	12.38	17.09	27.00	**28.47**
400	10.92	11.09	11.20	15.51	24.83	**26.21**
425	10.82	11.10	11.19	15.35	25.15	**26.20**
450	10.54	10.74	10.83	14.73	25.25	**26.48**
475	10.11	10.29	10.44	14.10	24.89	**25.40**
500	10.22	10.34	10.49	14.09	24.55	**25.17**

L/C	c_2^{256}	c_3^{256}	c_4^{256}	c_5^{256}	c_6^{256}	c_7^{256}
300	10.32	10.41	10.54	14.72	23.92	**24.78**
325	11.20	11.40	11.47	15.80	26.27	**27.78**
350	10.00	10.24	10.36	14.18	24.20	**25.31**
375	12.16	12.32	12.38	17.07	27.39	**28.95**
400	10.95	11.13	11.23	15.56	24.75	**26.15**
425	10.82	11.07	11.13	15.38	25.72	**26.86**
450	10.57	10.78	10.86	14.77	25.64	**26.74**
475	10.16	10.41	10.53	14.11	24.65	**25.28**
500	10.25	10.48	10.62	14.09	25.79	**26.47**

Table 4: Speedups - Solver - GPU vs GPU - 8, 16, 32, 64, 128 and 256 wavelengths per air column.

The speedups due to loop invariant removal and scalar replacement of aggregates show that the compiler is either not applying or not efficiently applying these optimizations. These two optimizations give small speedups because the introduction of the set of new variables that are necessary for their implementation increases register pressure and so increases the number of back and forth data movements among the GPU memories.

The optimization number 6 that uses the #define directives is important because it shows that, when complete knowledge of the data structure dimensions and the number of threads used to execute the codes is given to the compiler, the compiler's optimization job is greatly simplified.

Finally, the improvements due to constant folding show that, even with complete knowledge, the compiler does not unroll the short loops and does not transform variables into constants. However, both are important for eliminating instructions and reducing register pressure to get back some of the gains due to loop invariant removal and scalar replacement of aggregates.

Tables 5 and 6 compare the best runs of the GPU c_7^y codes, which use all the optimizations, against the best runs of the *CPU* codes.

L/C	S^8	S^{16}	S^{32}	S^{64}	S^{128}	S^{256}
300	40.0	29.9	38.3	41.8	41.6	39.4
325	43.9	41.7	28.5	38.4	41.8	47.0
350	36.7	33.4	37.5	45.9	**50.0**	41.0
375	**15.5**	17.1	18.5	18.4	17.5	17.4
400	43.2	31.6	44.5	44.8	49.0	32.3
425	16.9	18.0	19.3	19.9	19.5	19.0
450	17.3	17.4	19.9	18.4	17.7	18.3
475	17.4	17.4	20.0	18.4	17.8	18.3
500	49.4	47.2	15.9	38.4	44.9	49.7

Table 5: Speedups - Generator - GPU vs CPU.

L/C	S^8	S^{16}	S^{32}	S^{64}	S^{128}	S^{256}
300	20.6	18.3	16.0	21.4	17.6	22.1
325	16.2	19.4	14.9	22.1	19.8	22.7
350	**13.6**	17.5	18.7	18.3	19.3	20.5
375	14.6	15.5	15.2	16.8	18.4	15.5
400	14.0	13.8	15.1	14.6	15.8	15.0
425	15.0	15.4	14.8	15.5	16.1	20.6
450	16.4	16.3	16.5	15.2	18.6	15.5
475	15.8	17.3	16.7	17.7	15.7	17.2
500	21.7	23.5	23.4	24.8	**25.8**	25.2

Table 6: Speedups - Solver - GPU vs CPU.

The overall speedups obtained by applying all the optimizations varies somewhat with the problem size, but they are roughly of the order of a factor of 32 and 19 on average, respectively. The relatively straight-froward optimizations proposed, when applied in the order proposed, are therefore able to dramatically speedup the execution of the Fast-J core kernels, which can now run on GPUs 50.0 and 25.8 times faster than the state-of-the-art CPU codes.

11. RELATED WORK

For NVIDIA architectures post 2010, the compiler code is closed and the assembly is not disclosed. After 2010, people started to optimize code for the complex NVIDIA GPU architectures by only working at source level. However, even before 2010, when people could modify the compiler and work at assembly level, with very very rare exceptions, people always preferred to work at source level using CUDA or OpenCL. This was due to the difficulty of working at assembly level and to the many undisclosed and unquantified NVIDIA GPU architectural features and behaviors.

CUDA is the parallel computing platform and programming model invented by NVIDIA specifically for its GPUs. CUDA works as an extension of the C language, hides from programmers many low level GPU architectural details and increases code portability.

The means for optimizing CUDA codes can be divided into three categories: auto-tuning tools, frameworks for code analysis, and hybrids of the previous two. Many techniques do not easily fit into only one of the previous categories - i.e. a proposed technique might be 80% framework for analysis and 20% auto-tuning tool. For this reason, paper classification of these techniques is difficult. Therefore we classify them using the most emphasized technique.

In the auto-tuning category much work has been done to transform C codes to CUDA codes - C codes do not run on NVIDIA GPUs. One example is [5], where sequential C codes are automatically transformed into parallel codes for NVIDIA GPUs. The tools implemented in these works are important because they relieve users of the burden of managing the GPU memory hierarchy and the parallel interactions among GPU threads, both of which are important to reasonably speedup code development.

Some auto-tuning tools are in reality new programming notations. Layout modification usually gives good speedups and so it is one of the possible targets of the new programming notations. Instrumenting the codes using the programming notations, the compiler can better optimize the codes to produce speedups of up to two orders of magnitude, but this happens only for very specific codes [7].

Other auto-tuning tools include new programming languages. Some of these languages, like [8], relieve users of the burden of explicitly managing numerous low-level architectural details about the communications, the synchronizations among GPU threads, and the different GPU memories.

Communications between CPUs and GPUs are one of the culprits behind low performance executions. Some tools, like [37], do not depend on the strength of static compile-time analyses or user-supplied annotations, but are rather a set of compiler transformations and run-time libraries that take care to efficiently manage and optimize all CPU-GPU communications.

To alleviate the productivity bottlenecks in GPU programming, [43] studied the ability of GPU programs to adapt to different data inputs. A framework was implemented for the study. Given an input, the framework reduces the large optimization space and explores each one of the single optimizations.

Before 2010, NVIDIA programmers could use the NVIDIA assembly and so a greater number of compiler optimizations were possible at that time. Yang et. al [75] implemented some modules at the top of the contemporaneous compiler. The modules checked both coalesced and not-coalesced memory accesses to then modify, if possible, the data layout to make all the accesses coalesced.

Control flow in GPU applications is one of the most important optimization techniques. If more GPU threads in a warp follow one or more distinct branches then the whole GPU application will slow down considerably. Ocelot [18] characterized and transformed unstructured control flows in GPU applications. More recently, Ocelot has also become a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems.

Some frameworks, like [48], translate code. These frameworks usually receive in input CPU code skeletons. Beyond translating, these frameworks try to predict GPU performance. While the effort is noteworthy, different GPUs have different architectures, sometimes completely different architectures, so the frameworks are not able to produce good speedups and good performance predictions for codes running on the new GPU architectures.

Frameworks can also be used for code analysis, [24]. Such frameworks instrument PTX code. PTX (Parallel Thread Execution Programming Model) is a virtual pseudo-assembly introduced by NVIDIA to make CUDA codes portable on different NVIDIA GPU architectures. Workload characterization and load unbalancing information can be executed and extracted only by instrumenting PTX code. These frameworks instrument PTX codes, simplifying this cumbersome and often error-prone user job.

Hybrids like [70] are closer to being general tuning tools. These hybrids decide the correct memories to use, deal with the memory hierarchy, and code the data transfers among the different memories.

Relations between the size and shape of thread blocks used in the launch configurations, stream multiprocessor occupancy, and global memory access pattern are an important dimensional combination for code optimization. Their influence and relationships are studied for specific architectures in [67].

The efficiency of GPU applications is also influenced by dynamic irregular memory references. Pure software solutions like [76] have been developed to eliminate dynamic irregularities on the fly. The advantages of these solutions are that they do not require hardware extensions or off-line profiling. The optimization overhead is minimal and does not jeopardize the efficiencies of the base codes.

The importance of layout modification to produce coalesced accesses, and the importance of workload balancing for code executions, has also been demonstrated for more recent architectures, [65]. This also remains true if the new architectures are intrinsically different in their number, type and size of different GPU memories.

Writing efficient GPU code is often difficult and requires the exercise of specialized architectural features. Good performance can usually be achieved only after an intensive manual tuning phase. A programmer usually needs to test combinations of multi code versions, architectural parameters, and data inputs. Some hybrids, like [25], automate much of the effort needed to instrument codes with abstraction commands. The system, using the abstractions, explores the optimization space.

Producing high performance code on GPUs is often very time consuming. Furthermore, the whole tuning process has to be repeated for each target code and platform. Papers like [45] point out that saturating the GPU hardware resources can be used to reduce the tuning overhead of one order of magnitude.

12. CONCLUSION

It is of paramount importance to accelerate the Fast-J code, not only because it is integrated in many important climate and global chemistry models, but also because current Fast-J executions easily require months of simulation time, even when using high performance, multi-processor, and multi-threaded computing architectures. Worldwide, Fast-J requires at least 10 million hours per year of simulation time (likely an under-estimate).

GPUs are the best candidates for speeding up the Fast-J code and making it more energy efficient. This is in spite of GPU's complex architectures and very time consuming software optimization processes. Furthermore, effectively porting state-of-the-art CPU codes onto GPUs is challenging because of the significant architectural differences between CPUs and GPUs, while optimization processes are difficult because of the many undisclosed and unquantified low level architectural GPU features and behaviors, and because of the closed, undocumented compiler code.

In this paper we first efficiently ported the state-of-the-art, multi-threaded, CPU Fast-J code onto GPUs, and next selected and implemented some effective source-to-source high level optimizations. These do not require knowledge of low level GPU architectural details or the use of GPU assembly intrinsics - this makes the codes portable among different GPU families.

The newly ported and optimized GPU Fast-J kernel codes are 50.0 and 25.8 times faster than the already highly optimized CPU multi-threaded codes. Furthermore, the newly ported and optimized GPU Fast-J kernel codes consistently reach speedups of at least 15.5 and 13.6 for each scale of simulation.

13. REFERENCES

[1] T. G. 500. Ranking the World's Most Energy-Efficient Supercomputers. `http://www.green500.org/`. [Online; accessed 16-August-2014].

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

[3] 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.

[4] 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.

[5] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, pages 244–263, 2010.

[6] 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.

[7] G. V. D. Braak, B. Mesman, and H. Corporaal. Compile-Time GPU Memory Access Optimizations. *International Conference on Embedded Computer Systems*, pages 200–207, 2010.

[8] P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a High-Level Language for GPUs: (Via Language Support for Architectures and Compilers). *Conference on Programming Language Design and Implementation*, pages 1–12, 2012.

[9] 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-August-2014].

[10] K. D. Cooper. Compiler Support for GPUs: Challenges, Obstacles, and Opportunities. `http://www.cs.unc.edu/Events/Conferences/GP2/slides/Cooper.pdf`. [Online; accessed 23-August-2014].

[11] I. Corporation. 2nd Generation Intel Core vPro Processor Family. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/core-vpro-2nd-generation-core-vpro-processor-family-paper.pdf`. [Online; accessed 30-August-2014].

[12] I. Corporation. Intel Core i7-3960X Processor Extreme Ed. `http://ark.intel.com/products/63696`. [Online; accessed 27-August-2014].

[13] N. Corporation. NVIDIA GeForce GTX 750 Ti. `http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf`. [Online; accessed 22-August-2014].

[14] N. Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. `http://www.nvidia.com/content/pdf/fermi\_white\_papers/nvidiafermicomputear-chitecturewhitepaper.pdf`. [Online; accessed 24-August-2014].

[15] N. Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf`. [Online; accessed 29-August-2014].

[16] I. Corporbation. A Quantum Leap in Enterprise Computing. `http://www.intel.com/Assets/en\_US/PDF/prodbrief/323499.pdf`. [Online; accessed 26-August-2014].

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

[18] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: a Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems. *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 353–364, 2010.

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

[20] 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-August-2014].

[21] 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.

[22] 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-August-2014].

[23] J. L. et al. 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] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan. A Framework for Dynamically Instrumenting GPU Compute Applications within GPU Ocelot. *4th Workshop on General-Purpose Computation on Graphics Procesing Units*, 2011.

[25] N. Farooqui, C. Rossbach, Y. Yu, and K. Schwan. Leo: A Profile-Driven Dynamic Optimization Framework for GPU Applications. *Conference on Timely Results in Operative Systems*, pages 1–14, 2014.

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

[27] J. Ghorpade, J. Parande, M. Kulkarni, and A. Bawaska. GPGPU Processing in CUDA Architecture. *Advanced Computing: An International Journ.*, 3(1):1–16, 2012.

[28] P. N. Glaskowsky. NVIDIA's Fermi: The First

Complete GPU Computing Architecture . `http://sbel.wisc.edu/Courses/ME964/Literature/whitePaperFermiGlaskowsky.pdf`. [Online; accessed 19-August-2014].

[29] 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.

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

[31] 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.

[32] 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.

[33] S. Huang, S. Xiao, and W. Feng. On the Energy Efficiency of Graphics Processing Units for Scientific Computing. *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2009.

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

[35] 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-August-2014].

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

[37] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU Communication Management and Optimization. *Conference on Programming Language Design and Implementation*, pages 142–151, 2011.

[38] 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.

[39] Y. Jiao, H. Lin, P. Balaji, and W. Feng. Power and Performance Characterization of Computational Kernels on the GPU. *2010 IEEE/ACM International Conference on Physical and Social Computing (CPSCom)*, pages 221–228, 2010.

[40] 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.

[41] 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. JosÃľ, 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.

[42] 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.

[43] Y. Liu, E. Z. Zhang, and X. Shen. A Cross-Input Adaptive Framework for GPU Program Optimizations. *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10, 2009.

[44] 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.

[45] A. Magni, C. Dubach, and M. F. P. O. Boyle. Exploiting GPU Hardware Saturation for Fast Compiler Optimization. *Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–8, 2014.

[46] 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-August-2014].

[47] C. McClanahan. History and Evolution of GPU Architecture. `http://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf`, 2011. [Online; accessed 20-August-2014].

[48] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram. GROPHECY: GPU Performance Projection from CPU Code Skeletons. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[49] C. E. S. Model. Community Atmosphere Model Num. 5. `http://www.cesm.ucar.edu/working\_groups/Atmosphere/development/`. [Online; accessed 27-August-2014].

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

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

[52] 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.

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

[54] 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-August-2014].

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

[56] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Graphics Processing Units - Powerful, Programmable, and Highly Parallel - are Increasingly Targeting General-Purpose Computing Applications. *Proceedings of the IEEE*, 96(5):879–889, 2008. [Online; accessed 22-August-2014].

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

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

[59] 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-August-2014].

[60] 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.

[61] 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.

[62] 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-August-2014].

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

[64] G. Sissons and B. McMillan. Improving the Efficiency of GPU Clusters. [Online; accessed 28-August-2014].

[65] J. A. Stratton, N. Anssari, C. Rodrigues, S. I., N. Obeid, C. Liwen, G. Liu, and W. Hwu. Optimization and Architecture Effects on GPU Computing Workload Performance. *Innovative Parallel Computing*, pages 1–10, 2012.

[66] 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.

[67] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos. Understanding the Impact of CUDA Tuning Techniques for Fermi. *High Performance Computing Symposium*, pages 631–639, 2011.

[68] 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.

[69] D. Tristram and K. Bradshaw. Determining the Difficulty of Accelerating Problems on a GPUU. *South African Computer Journal*, 53:1–15, 2014.

[70] S. Ueng, M. Lathara, S. S. Baghsorkhi, and W. W. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. *Languages and Compilers for Parallel Computing*, pages 1–15, 2008.

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

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

[73] 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.

[74] B. Wilkinson. Emergence of GPU systems and clusters for general purpose High Performance Computing, 2011. [Online; accessed 21-August-2014].

[75] Y. Yang, P. Xiang, J. Kong, and H. Zhou. An Optimizing Compiler for GPGPU Programs with Input-Data Sharing. *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 343–344, 2010.

[76] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-Fly Elimination of Dynamic Irregularities for GPU Computing. *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 369–380, 2011.