Code optimization by using GPU applied to a Dataflow numerical simulation model

Luiz E S Evangelista [1], Alvaro L Fazenda, Vincius V de Melo
Science and Technology Institute, Federal University of Sao Paulo (UNIFESP)
Avenida Cesare Mansueto Giulio Lattes, 1201 - 12247-014 - São José dos Campos - SP - Brazil
E-mail: `Luiz_evangelista@hotmail.com, alvaro.fazenda@unifesp.br and vinicius.melo@unifesp.br`

Abstract. This work aims to study techniques for parallel computing using GPU (Graphics Processing Unit) in order to optimize the performance of a fragment of computational code, implemented as a Dataflow system, which is part of a meteorological numerical model responsible for calculating the advection transportation phenomena. The possible algorithm limitations for GPU efficiency will also be addressed through an extensive code instrumentation. Considering the difficulties found on the original algorithm which implies a GPU accelerated code dealing with flow dependencies and coarse grain parallelism, the performance gain with GPU may be considered fair.

1. Introduction

The data processed in numerical simulation can reach big volumes nowadays. Thus parallel programming is widely used, due to the large amount of time to get the expected results. Like other areas in Computer Science, such programming paradigm evolved and currently is possible to use external devices like GPUs as arithmetic co-processors in a fine-grain parallelism model [11].

A simple way of programming a GPU consists in using a programming language that follows the OpenACC standard [16]. Such language provides a set of directives to be applied to the original source-code, allowing the programmer to define a specific part of the code to be accelerated by an external device. The major benefit of this programming model is the lower effort to port the original code, created to be used with traditional CPUs, when compared to the popular CUDA language [11].

This paper aims to optimize the performance of a source-code developed following a dataflow approach, used to numerically simulate the advection transportation phenomena through a code ported to GPU using the OpenAcc standard. This numerical model is used as part of a weather forecast and climatic model. There are several successful cases of numerical simulation accelerated by GPUs, but the existing data-flow approach in the code represents a significant difficulty to achieve satisfactory efficiency. The paper shows a detailed performance analysis for the developed code aiming to identify major performance bounds, differences in algorithm performance applied to data arranged in different coordinate axes and scalability related to domain size and meteorological fields increments. The code robustness are evaluated through powerful accelerators and higher domains. The performance behavior under different kind of synthetic data are also measured, trying to identify better and worse case performance.

[1] To whom any correspondence should be addressed.

Hartley et al [7] propose a new scheme to solve a particular problem in a hybrid system using CPUs and GPUs, through a task division method using a data-flow, in which it is possible to dynamically change the task loads in order to keep the system balanced. Boulos et. al [1] solve a signal processing problem by using some GPUs with a model based on a data-flow approach, launching concurrent jobs on GPU overlapping computation and communication between host and device. Wang et. al. [20] addresses the development of a data-flow system in GPU applied to a signal processing application. Meng et. al [13] shows a framework to generate GPU code based on a data-flow scheme, by analyzing loops in the source-code.

There are just a few numerical meteorological models fully ported to GPUs. Cumming et. al. [2] have accelerated a meteorological model called COSMO by using GPU through code refactoring. On that work, part of the model responsible for fluid dynamics uses an API created in C++ and CUDA language, while the other part of the model, responsible for physics simulation phenomena, was ported to GPU by using OpenAcc standard directives.

Michalakes and Vachharajani [14] show a part of the WRF (Weather Research and Forecasting) model used in physics simulation phenomena ported to GPUs in CUDA language. This portion of code reached a speedup of about 20, with 7.4 Gflops.

Wang et. al. [20] present an approach for porting three different subroutines of GRAPES model (Global/Regional Simulation Prediction System) to GPU by using CUDA Fortran language. They highlight the three following key factors to improve performance on GPUs: reducing data transfer time between host and device, reducing the amount of memory access, and a thread control flow avoidance mechanism.

Govett et. al. [6] describe the module responsible for computational fluid dynamics in the NIM model (Non-hydrostatic Icosahedral Model), adapted to be used in GPUs. The authors also wrote a compiler to convert FORTRAN into CUDA code called F2C-ACC (FORTRAN-to-CUDA ACCelerator), which works using directives. The ported module runs 34 times faster on a single GPU than on the CPU used in the comparison.

In Shi [17] Master's thesis the performance analysis and optimization to GPUs was applied to serial solvers used to evaluate several numerical procedures in the Met Office Unified Model. The speedup by using CUDA language reach 10 times.

In Haines [8] Master's thesis, several physics schemes from the WRF model have been ported to GPU by applying OpenAcc directives on the original code. According to the author, the preliminary results show a modest 1.2x speed-up by simply introducing approximately 20 lines of OpenAcc directives.

Other work focus just in optimization by GPUs applied to advection transportation phenomena. Some of them could be also applied to weather forecast models or just integrate a computational fluid dynamics software. Souza et al [18] describe a parallel finite element implementation for the 2D time-dependent advection-diffusion problem ported to GPU, and the relationship between performance and energy consumption when compared to a parallel version working on a CPU cluster. White and Dongarra [22] have written a code to solve the advection problem that overlaps processing and communication on GPUs and describe results for FORTRAN implementations using various combinations of MPI, OpenMP, and CUDA.

All related work depicted here differs from this current paper by the numerical method adopted. The Walcek numerical method [19] to solve the advection transportation phenomena imply in data-streams, which was mapped to a dataflow scheme by Fonseca et al [4] as a computational strategy to allow concurrency. In this work this problem was ported and optimized to work with GPUs.

In the following sections the readers will find this work organized as follows: Chapter 2 describes the monotonic advection algorithm, Chapter 3 shows the Data-Flow system applied to the original advection algorithm, Chapter 4 presents the results, and finally in Chapter 5 the discussion and conclusion are postulated.

2. Algorithm to evaluate the Advection phenomena

The numerical advection model is a fundamental part for any fluid dynamics model. In the meteorology context the advection phenomena is usually responsible for the transport of scalar properties (e.g., mass density and mass mixing ratio) by the wind flow [5].

A new numerical method to evaluate the advection phenomena in a particular weather forecast model was proposed by Freitas et al. [5] and it will be referred in this work as monotonic advection. This new scheme is applied to the Coupled Chemistry-Aerosol-Tracer Transport model to the Brazilian developments on the Regional Atmospheric Modeling System (CCATT-BRAMS) [12], and contribute to more accurate results in numerical simulations over tropical and sub-tropical regions of South America. Those better results can improve the services available from CPTEC/INPE (Center for Weather Forecasting and Climate Research/National Institute for Space Research) in Brazil, where this model runs daily to simulate atmosphere pollution due gases emissions in urban centers and forest burns.

The first computational implementation of monotonic advection scheme applied in CCATT-BRAMS model has used a serial code without any parallelism [4], due to several dependencies in the original numerical algorithm. In Algorithm 1 it is possible to see part of this original code including two main loops, which iterate over an array in both positive and negative directions representing the discretized domain of the problem. The first loop (Algorithm 1) evaluates the advection transportation over adjacent cells with positive value for the wind velocity ($u_i > 0$) in a particular coordinate axis. The second loop does the same for the adjacent cells where the wind velocity is negative ($u_i < 0$). However, in the first loop there is a flow dependency related to instructions: $Qn_i = Fa(Flux_i, ...)$ and $Flux_i = Fb(Flux_{i-1}, Qn_i, ...)$. In the second loop, a similar dependency appears on instructions: $Qn_i = Fa(Flux_i, ...)$ and $Flux_{i-1} = Fb(Flux_i, Qn_i, ...)$, where Fa and Fb represent arithmetical procedures.

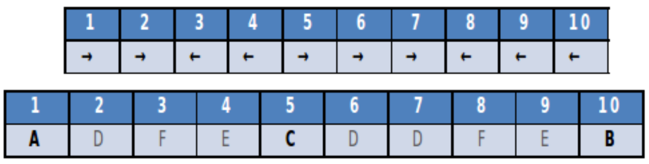
Algorithm 1 Pseudocode to evaluate the advection in an array [4, 19]

```
*Flux evaluation from right array boundary*
for i=2,n-1 do
    if $u_i < 0$ then
        Cycle
    end if
    if $vel_{i-1} < 0$ then
        $flux_i = F(u_i, ...)$
    else
        $Qn_i = Fa(Flux_i, ...)$
        $Flux_i = Fb(Flux_{i-1}, Qn_i, ...)$
    end if
end for
*Flux evaluation from left array boundary*
for i=n-1,2,-1 do
    if $u_{i-1} \geq 0$ then
        if $u_i < 0$ then
            $Qn_i = F(flux_i, Flux_{i-1}, ...)$
        end if
    else
        $Qn_i = Fa(Flux_i, ...)$
        $Flux_{i-1} = Fb(Flux_i, Qn_i, ...)$
    end if
end for
```

3. Data-flow scheme applied to advection

To deal with flow dependencies in the original form of monotonic advection (Figure **??**) implemented a Data-flow approach which detects and evaluates concurrent streams (sequence of adjacent cells with same wind direction) over the computational discrete domain. The first step to create the data-flow structure consists in identifying cells based on their wind direction neighborhood, and classifying each one like shown in Table 1.

The Figure 1 show an example for an array representing wind velocity direction, where each cell was classified according to the labels depicted in Table 1. The next step consists in joining cells with same wind direction as streams. The evaluation order of the labeled cells respecting the flow dependencies could be represented by a directed acyclic graph (DAG) [4], as shown in Figure 2.



1	2	3	4	5	6	7	8	9	10
→	→	←	←	→	→	→	←	←	←

1	2	3	4	5	6	7	8	9	10
A	D	F	E	**C**	D	D	F	E	**B**

Figure 1. Classified wind array [4].

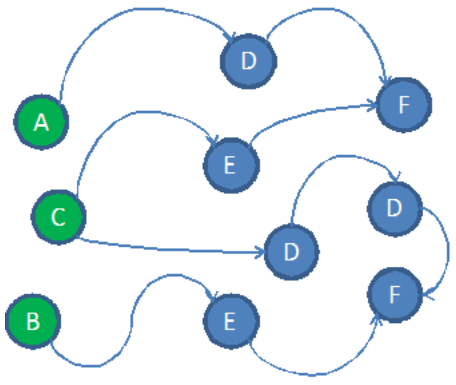


Figure 2. DAG representing flux dependencies among cells [4].

Table 1. Cell labels

$Vel(i-1)$	$Vel(i)$	Label	Description
X	$\rightarrow$	A	Inflow Left Border
$\leftarrow$	X	B	Inflow Right Border
$\leftarrow$	$\rightarrow$	C	Only Outflow
$\rightarrow$	$\rightarrow$	D	Positive stream
$\leftarrow$	$\leftarrow$	E	Negative stream
$\rightarrow$	$\leftarrow$	F	Only Inflow

The DAG in Figure 2 can be used in order to aggregate cells in streams, and can be represented by a Heap. In the Figures 1 and 2 it is possible to identify three concurrent independent streams and two dependent cells, as depicted in Table 2.

Table 2. Independent Streams and dependent cell detected

Independent Streams	Dependent Cells
a) AD (1,2)	d) F (3)
b) ECDD (4,5,6,7)	e) F (8)
c) EB (9,10)	

The algorithm responsible to preprocess the data, identifying and classifying the streams/tasks, is $O(XYZ)$, where X, Y and Z represent the total points in each axis for three-dimensional discrete domain. The streams are inserted in a heap or array structure following the order: data in X axis in head, data in Y axis in the middle and data in Z axis in tail. For each axis the concurrent streams are in head and the dependent streams are in tail, in order to respect the dataflow scheme. The elapsed time to execute the preprocessing is minor than 0.5% of total time, for the biggest case evaluated.

It is important to notice that the tasks in Table 2 require different computational effort to be done due to stream length and possible different memory arrangement. These characteristics generally represent challenges to achieve high efficiency in GPUs.

The monotonic advection source-code can be applied to any array with a specific offset between adjacent cells for each coordinate axis. A three-dimensional array representing the discretized physical domain was used in all following tests, thus the application of the same algorithm to each axis requires just the correct specification for the first element and the offset to the next cells. The computational memory allocation order is organized as ZXY to follow the same scheme in the original meteorological model coded in FORTRAN. However the same numerical procedure must be applied to N scalar meteorological fields [5]. Therefore, the final array memory arrangement is $NZXY$, which is the same computational complexity $O(NZXY)$ for the monotonic advection algorithm applied to a three-dimensional domain.

The heap proposed in the data-flow approach for monotonic advection in order to address the tasks was substituted by an array of tasks in the GPU. The array follows the same heap order, which contains all concurrent tasks at the head and the dependent tasks at the tail. It is possible to check in Algorithm 2 and 3 the original serial code in CPU and the accelerated parallel code, respectively.

Algorithm 2 Original serial code - Tasks list stored in a heap

```
while $not(end(Heap))$ do
    $taskget\_job(Heap)$
    $advection(task)$
    $update\_heap(Heap)$
end while
```

4. Results

In order to improve the model performance, different versions were developed (all of them written in FORTRAN-90 language). Each new release implements improvements on data structure,

Algorithm 3 Accelerated parallel code - Tasks list in an array

```
Data transfer CPU $\rightarrow GPU(4Darrays, Taskslist)$
for (concurrent tasks) do                                  ▷ GPU Kernel
    advection(tasks)
end for
for (dependent tasks) do                                   ▷ GPU Kernel
    advection(tasks)
end for
Data Transfer GPU $\rightarrow CPU(4Darrays)$
```

numerical algorithm, data management on GPU, or in OpenACC directives. All versions developed are described in this work in order to clarify the optimization processes and to inspire methods to guide future efficient GPU code porting for other similar numerical algorithms.

All data advected by the numerical method was contained in a four-dimensional array arranged as $NXYZ$, where N represents the number of scalar meteorological fields and X, Y, and Z represent the total discrete points in each coordinate axis, respectively.

The developments were performed using two different machine configurations. Most tests used an CPU *Intel Core i7-975* with accelerator *NVidia Tesla C2050* , and a PGI Accelerator v.14.1-0 compiler which supports OpenACC directives. Another machine used only for more computationally expensive tests has a 12 cores *Intel Ivy Bridge* with an *NVidia Tesla K20X* accelerator, and PGI Accelerator v.14.7 compiler. Details about the different model versions/releases used along this work are described in the following items:

(1) **Original serial code in CPU [4]:** Original serial data-flow code to be executed in a CPU and used as a performance reference.

(2) **Parallel code with OpenMP:** Adaptation of the original serial code using OpenMP directives, which allows concurrent tasks advecting streams evaluation. The source-code requires a critical section responsible for heap management and synchronization points to comply with task dependency.

(3) **Naive code with OpenACC:** First version in GPU with basic (naive) OpenACC directives, such as data transfer sections and GPU kernels over the numerical method. Unnecessary data transfer was noticed in this source-code.

(4) **GPU logical data-structure tuned code:** Code tuned for GPU blocks and threads by setting OpenACC Gang and Vector data-structure clauses, where the outer loop (responsible for tasks list iteration) was fixed to use 512 blocks and the inner loop (responsible for traversing the meteorological fields) was fixed to use 32 threads. This configuration was determined after experimental tests. It is worth emphasizing the need to inform the compiler that these loops are totally independent (independent OpenACC clause), otherwise the compiler automatically defines dependency where they do not exist.

(5) **Data transfer optimization code:** Code tuned by intent optimal definition for all data used in the numerical simulation. The data intent defines different kinds of data transfers from host to device and vice-versa (copyin, copyout, copy and create OpenACC clauses). Data transfer reduction was also implemented in this version specially applied in the interval between each axis.

(6) **Better GPU occupancy code:** In this version a previous evaluation of a small part of code was performed out of main outer loop in order to allow both a better GPU occupancy [11] and better performance. Such modification requires more memory on the system. Regarding code portability, it does not cause significant performance changes when running in traditional CPUs.

(7) **Branches decrease code:** In this version the code gets a replacement of conditional structures (Ifs) by masked arithmetic expressions based on flags. This action reduced the average of divergent paths in 39.4% in a given problem tested, with improvement in performance.

(8) **Sorted stream code:** Version developed to sort the streams by a task rank according to their size and stream type, trying to reduce divergent branches in operations within a GPU Warp [11].

(9) **Hybrid code:** Code prepared to use simultaneously multicore CPU and GPU to perform tasks.

(10) **Previous calculation code:** Version joining two phases of the code: the first phase calculates everything that is possible with independent parallel floating point operations by each grid cell, and the second phase uses the data produced by the first phase as input to evaluate the advection following the data-flow approach over streams.

The first seven versions developed show a continuous performance increase, as may be observed in the following. Unfortunately, the last three versions did not increase performance when compared to version seven due to several reasons that will be detailed in Section 4.1. Figure 3 presents the speedup for the first seven successful versions applied to a problem with the following dimensions: $22 * 38 * 336 * 336 (NZXY)$. The input data was obtained from a real meteorological model simulation.

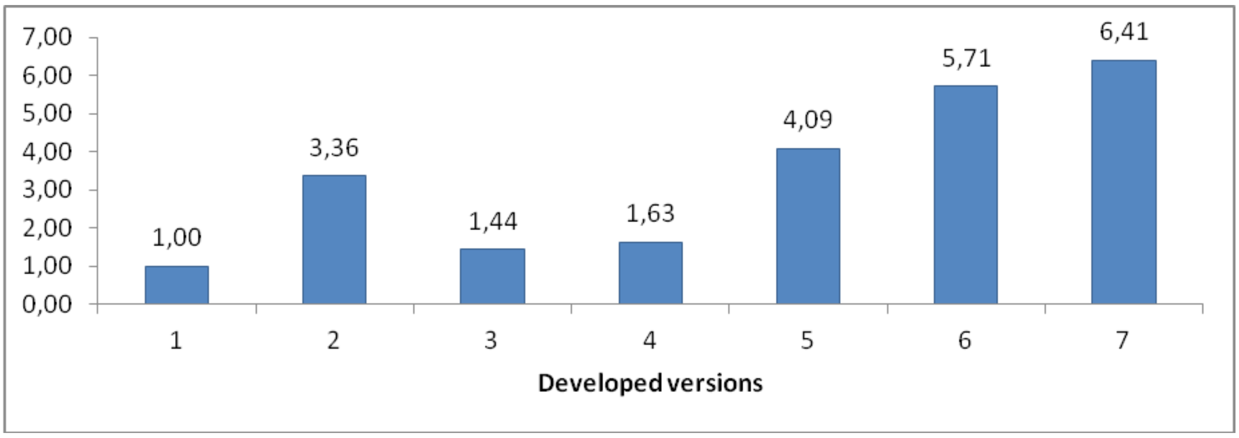


Figure 3. Speedup for the first seven stable versions

Besides the speedup, there are other evidences about the continuous performance evolution for the versions developed for GPU, like some metrics and events measured by a profiling tool called "nvprof" [3]:

- **ipc:** Instructions executed per cycle.
- **gld_efficiency:** Efficiency for load data requests in GPU global memory.
- **gst_efficiency:** Efficiency for store data requests in GPU global memory.
- **achieved_occupancy:** Real Occupancy achieved in GPU.
- **MIPS:** Millions of instructions per second.

Table 3 shows the above metrics for same seven developed GPU versions, considering a problem configured as described in the test for Figure 3. The IPC metric clearly shows successive performance increases along the versions which allow an increased number of operations executed per cycle. It is also possible to see the decrease in the use of branches (version seven) linked to an improvement in read access to global memory. The algorithmic modification (version six) which provides a better GPU occupancy also improves write access to the global memory. Furthermore, the Gigaflops rate shows continuous performance increase for all successive first seven versions.

Table 3. Metrics and events measured by nvprof for the first seven versions

Version	ipc	gld_efficiency	gst_efficiency	achieved_occupancy	MIPS	Gigaflops
3	0.421	44.3%	46.0%	7.1%	416896716.2	6.2
4	0.631	39.0%	23.5%	15.3%	478748992.8	7.1
5	0.641	42.7%	42.7%	15.3%	691176079.5	17.8
6	1.061	42.6%	42.6%	30.3%	835767651.2	24.8
7	1.189	51.9%	51.9%	30.3%	783768534.7	27.8

Another way to check the efficiency in memory usage consists in measurements in coalescing global memory access on GPU [11] with the following two formulas [15]:

$$\textit{Read coalescent memory ratio} = (l1\_global\_load\_hit + l1\_global\_load\_miss)/gld\_request \quad (1)$$

$$\textit{Write coalescent memory ratio} = global\_store\_transaction/gst\_request \quad (2)$$

Where the nvprof events depicted in Equations 1 and 2 mean:

- *l1_global_load_hit/l1_global_load_miss*: Total number of access with/without success in data on the GPU L1 cache memory.
- *gld_request/gst_request*: Total requests to read/write in GPU global memory.
- *global_store_transaction*: Total number of store transactions in GPU global memory.

The metric value measured for read/write coalescing memory access should be close to 1.0 for 32-bit data ("float" type) [15]. Values greater than 1.0 indicate non-coalescing access. For version seven, the most efficient one, the metric values measured for reading and writing were 1.33 and 1.63, respectively, showing a non perfect coalescing memory access, which needs further investigations to identify the causes and to minimize the effects.

It is also possible to estimate a major bound for the algorithm developed, determining whether it is memory or computationally bounded. To measure this characteristic two extra versions of code were developed [15]. The first extra version performs only memory access operations without computation, and the second one only performs related logic and numerical operations instead of the original code, using the data already loaded into GPU registers. The result is shown in Figure 4, indicating an equivalent time spent on memory and computation only operations, characterizing a balanced algorithm with performance not limited by just memory access or numerical and logical procedures. So, future optimizations should focus not only in one aspect of algorithm.

Furthermore, Figure 4 also shows that the total advection time (column "full") is only slightly higher than the two measurements focusing only memory or computation, indicating a possible and beneficial overlapping between memory and logical/arithmetic operations. It also indicates that memory latency is not a significant performance boundary in this case.

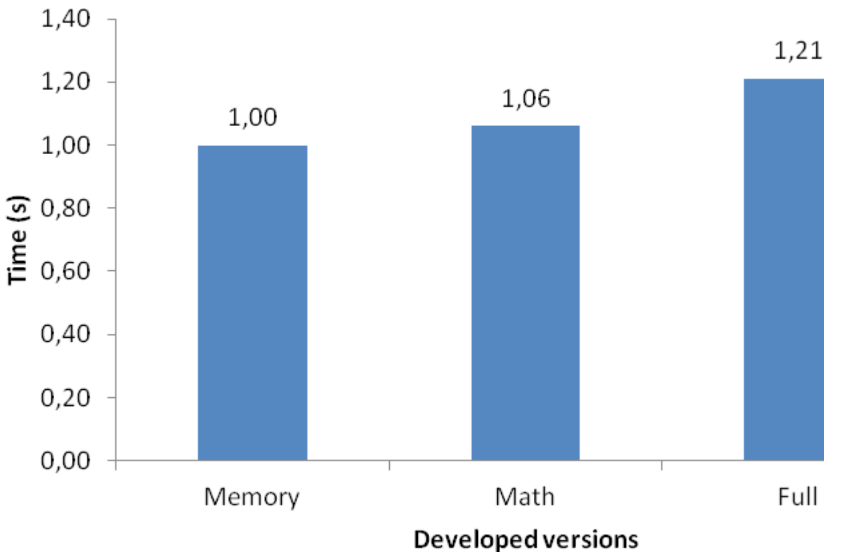


Figure 4. Memory / computation time spent

It was also possible to measure algorithm performance (for the best release: version seven) applied to data arranged in different coordinate axes of a three-dimensional computational grid, as shown in Table 4, considering the grid with the same dimensions mentioned configuration in Figure 3.

Table 4. Performance over coordinate axes

Coordinate Axis	Time (s)	Gigaflops
X	0.1026	66.6
Y	0.1037	65.4
Z	0.0998	70.0

According to Table 4 there was no significant performance difference between the data processed in different coordinate axes, despite the different memory arrangement of neighbor cells on each coordinate axis. The calculations over Z axis shows a slightly better performance comparing to others, which is expected, as the neighbor cells along Z axis are positioned closer to each other in the memory addresses when comparing to cell neighborhood in other axes, which suggests better memory access efficiency. However, it is important to highlight the differences in the number of tasks (and consequently in the number of streams), as well as their specific characteristics for each coordinate axis, which could interfere in performance.

The scalability of our method with respect to the problem domain size is shown in Figures 5 and 6. In Figure 5, the three-dimensional domain is configured to $168 * 168 * 38 (XYZ)$ elements and the number of meteorological fields (N) ranges from 1 to 60 fields. In Figure 6, the number of meteorological fields and the number of points in vertical direction (Z axis) was set to 10 and 38, respectively, and the number of horizontal elements over X and Y axis ranges from 48 to 336 elements.

In Figure 5 it is clear to notice that the runtime does not increase linearly with the proportional increase in meteorological fields, since the estimated number of floating point

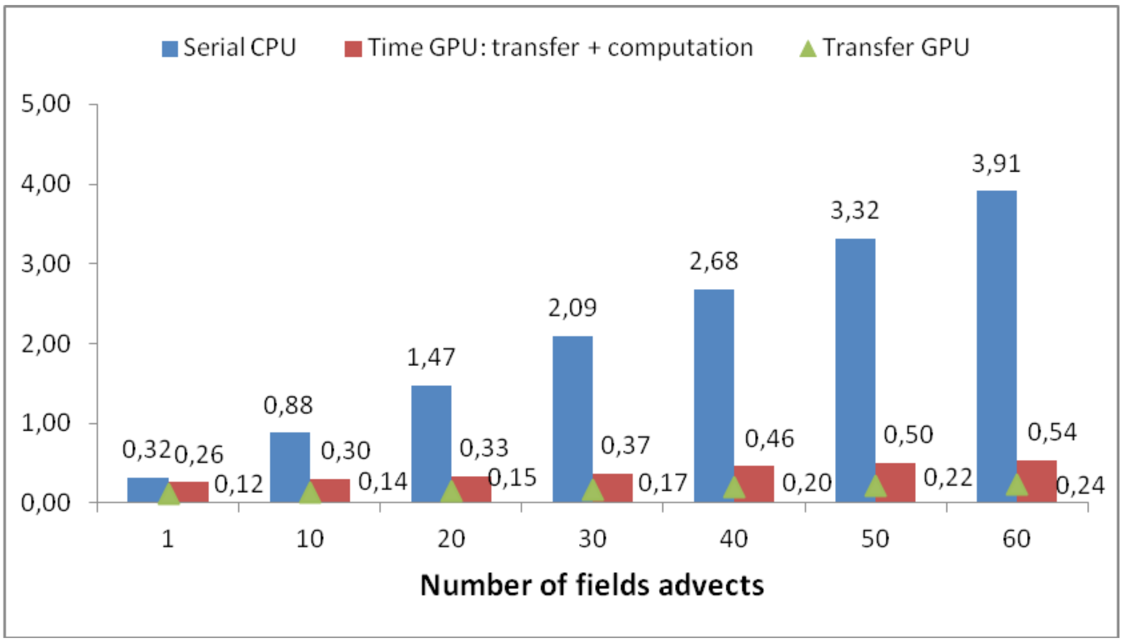


Figure 5. Scalability based on meteorological fields increasing

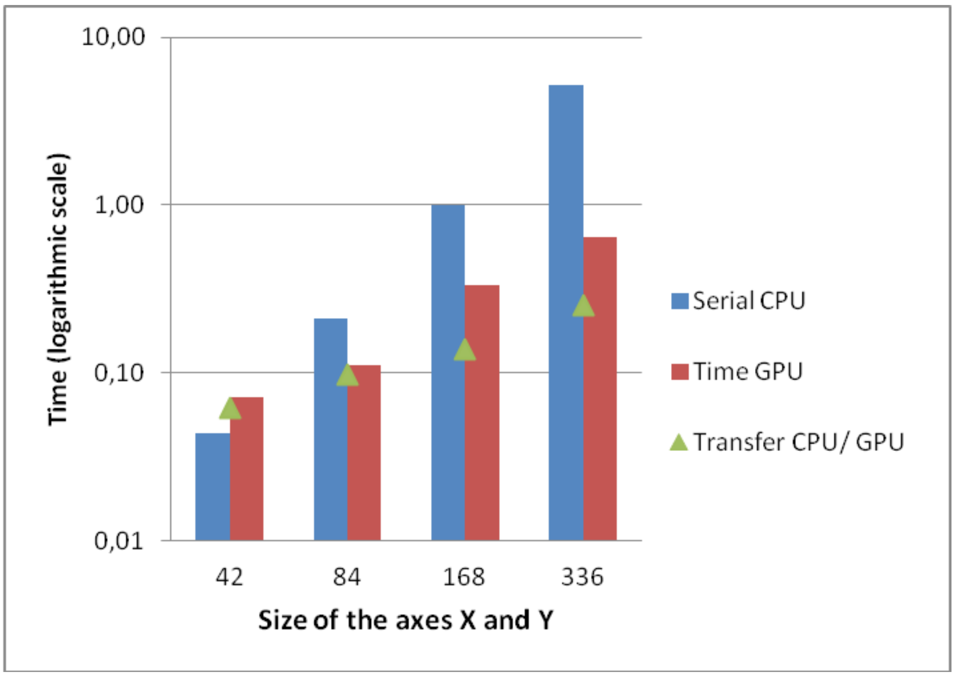


Figure 6. Scalability based on horizontal domain increasing

operations raises up to 60 times while runtime increases only 3.8 times. This could be confirmed observing the Gigaflops rate measured for the same case in Figure 7, which shows that the performance increases with the amount of data processed, reflecting a typical GPU behavior. It also reveals a non proportional transfer time increasing to a data volume growing.

Figure 6 shows a similar behavior as the execution time increase is not proportional to the continuous increase of 4 times for computational domain, implying in a computational effort

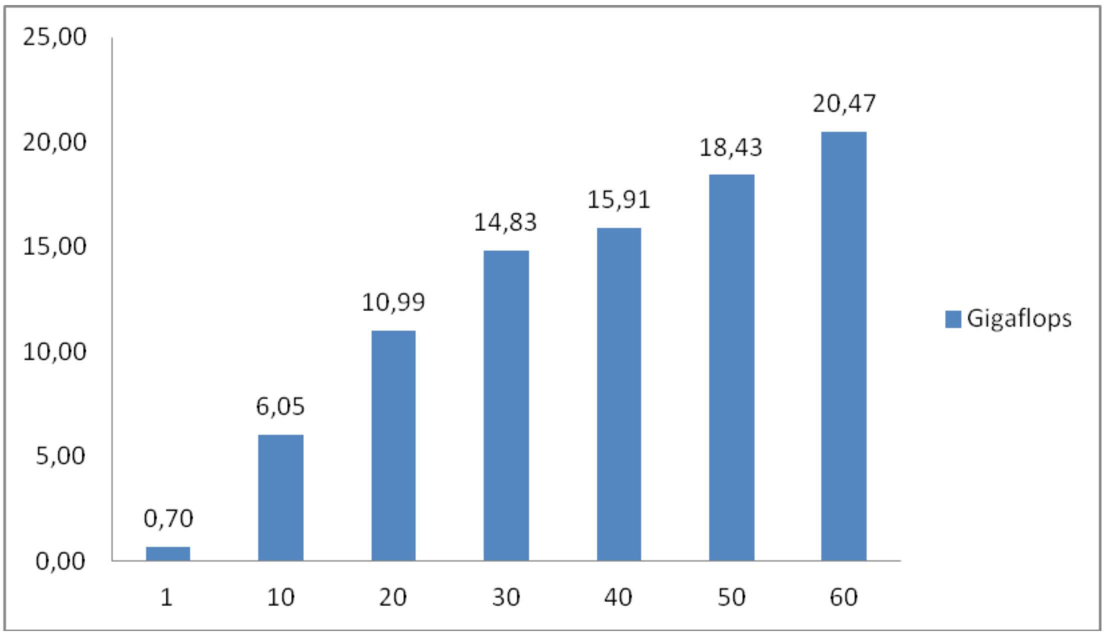


Figure 7. Scalability in Gigaflops based on meteorological fields increasing

increase of same order. Comparing the first and last domain size in same figure, it is possible to see a computational effort 64 times greater (42*24*38 and 336*336*38 respectively). However, the execution time grows to only 9.1 times the reference value. In the same figure one may observe the relative time spent on data transfers between host and device, which shows 85% of total time is due these transfers in lower domain but only 25% is spent for similar transfers for the greater domain. This shows an expected decreasing in data transfer importance related to increase of used data volume.

Comparing the scalability over a increasing number of meteorological fields (Figure 5) and over a increasing domain length (figure 6), the first case shows a better scalability, since the operations related to different meteorological fields are totally independent, generating more parallelism, while a domain variation could change the stream patterns, and the potential parallelism is dependent of the input data.

The robustness of the developed approach can be demonstrated using a more powerful accelerator and bigger problem domains. The results are shown in Table 5 for a computational domain size set as 60*38*336*336 ($NZXY$) in a machine configured with 12 cores *Intel Ivy Bridge* CPU, *NVidia Tesla K20X* accelerator and PGI Accelerator v.14.7 compiler, which shows GPU performance advantage comparing only up to 8 CPU cores. Using more CPU cores the OpenMP version reaches better results. It is worth to mention the significant data transfer time, which corresponds to 47% of total time. However, considering in the future a meteorological model fully ported to GPU acceleration, the cost of data transfer should be minimized since the input data could be already allocated on GPU memory originating from other part of code also running in same device. In this case the accelerated monotonic advection still has a performance 2 times better than a 12 cores CPU.

Another test was developed considering synthetic data instead of real input data like in previous tests, in order to verify the performance in a set of different input data configurations, which modify the number of concurrent stream tasks. The computational domain was set to 22*38*336*336 ($NXYZ$), and a synthetic data generation was applied and analyzed just for the data along X axis where the quantity of cells in left and right side of task streams (with different wind directions) was progressively increased, so, the stream size grows with a proportional decreasing in the total number of tasks to be processed. The results in Table 6 confirm an expected better performance with the maximum number of tasks, and consequent small size of streams, because it allows more parallelism. In real simulation, such situation occurs in cases where the wind direction changes frequently. Input cases where the wind has just a few variations, as it happens in high atmosphere, imply in long length stream and lower number of tasks, reducing the GPU performance.

Table 5. Performance with synthetic input data

Threads OpenMP or GPU	Time (s)	Speedup
1	12.03	1.0
2	6.33	1.9
4	3.36	3.6
6	2.44	4.9
8	1.84	6.5
10	1.44	8.4
12	1.30	9.3
GPU	1.49	8.1

Table 6. Results obtained with the same synthetic data size

Stream left	Stream right	Size of stream	Number of tasks	Time (s)
1	1	2	4004964	0.09
3	3	6	1347660	0.15
6	6	12	674316	0.15
18	18	36	241452	0.15
54	54	108	108782	0.15
162	162	324	60686	0.17

4.1. Inefficient versions
The last three versions presented in the beginning of this Section show no performance gains when compared to the final efficient version (release seven). The first one (release eight) consists of a new code where the task list is sorted according to both the stream type and size, trying to avoid divergent branches within a GPU Warp. However, this action did not generate significant improvements in performance. For a problem configured in the same way as described in Figure 4 (22*38*336*336 - $NZYX$) with real input data, the runtime was similar to that obtained without tasks ordering, considering the advection time plus the overhead necessary to sort the task list.

The second inefficient version works with hybrid processing, where both CPU and GPU work together and simultaneously to obtain the final result. The algorithm is designed allowing asynchronous GPU kernel launches overlapped with CPU processing. The GPU tasks are taken from the beginning of tasks list and the CPU tasks are taken from tail, in opposite direction. The designed algorithm was inspired in the OpenMP version (release two) and allows dynamic load balancing. However, the processing time shows just slightly improvements in performance (2% gain). The main reason to justify why the hybrid version does not show a significantly gain over the best computational time obtained rely on data transfers elapsed time. Since we are dividing the discrete domain between streams (or tasks) to be evaluated by CPUs and GPU, according the algorithm depicted in Listing 1, it is not possible to transfer to/from GPU just the portion related to some streams, because it represent a irregular domain over the 4D array. So, all domain data are transferred taking about 70% of OpenMP total time. The remaining time is insufficient to receive significant contributions by CPUs, since any stream processed simultaneously by 12 CPUs spend about 2.6 times than GPU (considering just Kernel execution time). Besides, it is necessary to post-process the data merging the results coming from different devices, which takes more time. The Listing 1 shows the current developed hybrid version. A new version where the domain is divided by atmospheric fields to be processed by CPUs and GPU are in progress. With this new version will be possible to transfer to/from GPU only the portion used on it.

Listing 1. Hybrid version

```
!$OMP PARALLEL &
!$OMP shared(ini, end, task_remain, chunk, i) &
!$OMP private(j, kernel_running)
do while (task_remain>0)

    ! Check if it is possible to launch a new GPU kernel
    kernel_running = .false.
    !$OMP CRITICAL
    if (acc_async_test_all()) then !kernel in execution?
        ! GPU is free
        call kernel_gpu_openacc(...)
        ini = min(ini+chunk, (TotalTasks - 1))
    else
        ! update total tasks remaining
        end = end - 1
        task_remain = end - ini + 1
        kernel_running = .true.
    endif
    !$OMP END CRITICAL

    ! Check if it is possible to launch a CPU thread
    if (task_remain>0 .and. kernel_running) then
        ! new CPU thread
        call process_advec_cpu(...)
    endif

enddo
!$OMP END PARALLEL
```

The motivation to create the last version (release ten) comes from an analysis of the source-code where it is possible to identify independent arithmetic operations by each grid cell. Therefore, a new version was developed joining two independent phases of the code: in the first phase a new numerical algorithm calculates everything that is possible to perform with independent parallel floating point operations by each grid cell, and the second phase uses the data produced by the first one as input to evaluate the advection following the depicted data-flow approach. The results show a second phase solving the advection phenomena in 77% of total time, but the algorithm for the first phase presents a performance higher than 23% of total time, so there is no performance gain. This code also requires significantly more GPU memory to store temporary data for the first part of code.

5. Discussions and Conclusions

The naive introduction of OpenAcc directives was not sufficient to achieve performance efficiency in GPU for the monotonic advection. It required several modifications such as algorithmic modifications and complementary OpenAcc directives applied to the original source-code, which improved the performance in 14.6 times since the first version in GPU. It was possible to drop the wall clock from about 3.33s (1.9 Gflops) to the final efficient version running the same problem with wall clock of about 0.74s (27.8 GFlops). Auxiliary tests also show balancing between the time spent on memory access and floating point operations in the final version, and a beneficial overlapping of them.

However, the amount of code changes and the programmer's effort were quite significant, which raises doubts about the expected high productivity of applying OpenACC directives [9] over original code versus the traditional CUDA language development; furthermore, there are some evidence related on this kind of problem in literature [10].

The performance gain with GPUs may be considered only modest compared to traditional optimizations advection methods [18, 21], where performance reaches hundreds of Gflops in GPUs. However, in this work the solution adopted to deal with flow dependence generates a coarse grain parallelism for GPU, due to grouping sets of data into streams, which needs to be evaluated serially. To exemplify the coarse grain parallelism, may be considered one case where the code developed uses approximately only 230,000 independent tasks and a few dozen dependent tasks to evaluate monotonic advection over a three-dimensional domain set to 38*336*336 cells (ZXY), which has more than 4 million cells. Thus, on average, each task is responsible for calculating the advection for 56 adjacent cells, while traditional numerical advection methods usually addresses concurrent tasks for each computational cell, generating much more parallelism.

The data access in GPU global memory for the developed code shows imperfect coalescing. Such effect may become the subject for future investigations trying to identify its causes and propose optimizations in order to minimize the effects.

It is also important to note that the gain related to CPU has the same order in different architectures used. Hybrid versions, which involve simultaneous CPU and GPU computation, did not improve performance significantly for the characteristics of the investigated problems.

Future developments to expand this work include, in addition to the aforementioned improvements in coalescing data access, code extensions allowing multiple GPUs in conjunction with distributed processing via MPI programming. Benchmarks considering productivity and performance using OpenACC and CUDA could also be addressed, as well as using other known accelerator as the *Intel Xeon Phi.*

References

[1] Boulos V, Huet V, Fristot V, Salvo L and Houzet D 2012 *Efficient implementation of data flow graphs on multi-gpu clusters.* Journal of Real-Time Image Processing 9 217-232
[2] Cumming C, Osuna T, Gysi T, Bianco M, Lapillonne X, Fuhrer O and Schulthess T C 2013 *A review of the challenges and results of refactoring the community climate code cosmo for hybrid cray hpc systems.* Proceedings of Cray User Group
[3] DocsNvidia 2014 *Profiler User's Guide CUDA Toolkit documentation*

[4] Fonseca R M, et al. 2012 *Dataflow paradigm approach applied to improve computational efficiency of a monotonic advection scheme.* Proceedings of Workshop PADTempo/XVII Congresso Brasileiro de Meteorologia. Gramado

[5] Freitas S R, Rodrigues L F, Longo K M and Panetta J 2011 *Impact of a monotonic advection scheme with low numerical diffusion on transport modeling of emissions from biomass burning. Journal of Advances in Modeling Earth Systems* 3

[6] Govett M, Middlecoff J and Henderson T 2010 *Running the NIM Next-Generation Weather Model on GPUs.* 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid) 792-796

[7] Hartley T D R, Saule E and Catalyurek U V 2010 *Automatic dataflow application tuning for heterogeneous systems.* International Conference on High Performance Computing (HiPC)

[8] Haines W A 2013 *Acceleration of the Weather Research Forecasting (WRF) Model using OpenAcc and Case Study of the August 2012 Great Arctic Cyclone.* Master's thesis The Ohio State University

[9] Herdman J A, Gaudin W P, McIntosh-Smith S, Boulton M, Beckingsale D A, Mallinson A C and Jarvis S A 2012 *Accelerating Hydrocodes with OpenACC, OpeCL and CUDA.* SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC) 465-471

[10] Hoshino T, Maruyama N, Matsuoka S and Takaki R 2013 *CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application.* 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid) 136-143

[11] Kirk D and Hwu W W 2010 *Programming Massively Parallel Processors: a hands-on approach.* Morgan Kaufmann 288

[12] Longo K M, Freitas S R, Pirre M, Marcal V, Rodrigues L F, Alonso M F and Mello R 2013 *The Chemistry CATT-BRAMS model (CCATT-BRAMS 4.5): a regional atmospheric model system for integrated air quality and weather forecasting and research.* Geoscientific Model Development 6 1389-1405

[13] Meng J, Morozov V A, Vishwanath V and Kumaran K 2012 *Dataflow-driven gpu performance projection for multi-kernel transformations.* SC '12 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis

[14] Michalakes J and Vachharajani M 2008 *GPU acceleration of numerical weather prediction.* IEEE International Symposium on Parallel and Distributed Processing, IPDPS 1-7

[15] Micikevicius P 2012 *GPU Performance Analysis and Optimization.* GTC2012

[16] OpenACC-Standard.org 2013 *The OpenACC Application Programming Interface.* OpenACC.org

[17] Shi S 2012 *GPU Implementation of Iterative Solvers in Numerical Weather Predicting Models.* Master's thesis The University of Edinburgh

[18] Souza A F, Veronese L, Lima L M, Badue C and Catabriga L 2012 *Evaluation of two parallel finite element implementations of the time-dependent advection diffusion problem: Gpu versus cluster considering time and energy consumption.* High Performance Computing for Computational Science - VECPAR 149-162

[19] Walcek C J *Minor flux adjustment near mixing ratio extremes for simplified yet highly accurate monotonic calculation of tracer advection.* Journal of Geophysical Research: Atmospheres, 105:93359348, 2000.

[20] Wang L, Shen C, Seetharaman G, Palaniappa K and Bhattacharyy S S 2012 *Multidimensional dataflow graph modeling and mapping for efficient gpu implementation.* IEEE Workshop on Signal Proc. Systems (SiPS) 300-305

[21] Wang Z, Xu X, Xiong N, Yang L T and Zhao W 2011 *GPU Acceleration for GRAPES Meteorological Model.* 13th IEEE International Conference on High Performance Computing and Communications (HPC) 365-372

[22] White J B and Dongarra J J 2011 *Overlapping computation and communication for advection on hybrid parallel computers.* IEEE International Parallel Distributed Processing Symposium (IPDPS) 59-67