Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXIX, págs. 7043-7051 (artículo completo)
Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (Eds.)
Buenos Aires, Argentina, 15-18 Noviembre 2010

GPU ACCELERATION OF CALPUFF

Pablo G. Cremades, Enrique S. Puliafito and Rafael P. Fernandez

Grupo de Estudios Atmosféricos y Ambientales, Universidad Tecnológica Nacional, Facultad Regional Mendoza, Rodríguez 273, 5500 Mendoza, Argentina.
http://www.frm.utn.edu.ar/geaa

Keywords: CUDA, Model Optimization, High performance computing.

Abstract. Weather and climate prediction, as well as air quality software models are very computing intensive, requiring high processing power in order to achieve precise results in a reasonable time. For many years, performance improvement has come from increasing processors speed. However, processor speed cannot be indefinitely increased. An alternative strategy for increasing performance is through the use of large-scale parallelism architectures. While recent models show the benefits of parallel computing, multicore systems or cluster may be cost ineffective in certain scenarios. CUDA (Compute Unified Device Architecture) is a general purpose parallel architecture introduced by NVIDIA®. CUDA-enabled graphics processing units have hundreds of cores that can collectively run thousands of threads at a fraction of a cost of other parallel computer classes.
This paper shows the performance improvement achievable in CALPUFF, an advanced non-steady-state meteorological and air quality modeling system, through parallelization and CUDA computing architecture. A runtime analysis of the model was conducted in order to find a candidate module for parallelization. Results from the optimized version are compared to those from original serial version of CALPUFF for error analysis.

1 INTRODUCTION

Since CUDA became available in 2007, many computing intensive applications, which can exploit the benefits of parallel computer architecture, have been ported to run on General Purpose Graphic Processing Units (GP-GPU) (Komatitsch et al., 2009), achieving speedups from x3 to x50. Recently, some research groups have shown x1.25 overall performance increased in the state of-the-art WRF model (Weather Research and Forecast) by porting only one physics module to run on the GPU (Michalakes and Vachharajani, 2009), leaving room for further optimization.

Simulation of the dispersion of chemically active or passive species on a gridded domain is a numerical problem that can be efficiently solve using parallel computing. Two different types of dynamic models are usually used to calculate the dispersion of chemical species from a single source: the so-called Eulerian and Lagrangian transport models. Eulerian models have the advantage of being computed on a three-dimensional spatial grid providing 3D descriptions of the meteorological fields rather than trajectories of single particles (Molnár et al., 2010).

CALPUFF is a multi-layer, multi-species Eulerian non-steady-state puff dispersion model which can simulate the effects of time and space-varying meteorological conditions on pollutant transport, transformation, and removal (Scire et al., 2000). GEAA (Grupo de Estudios Atmosféricos y Ambientales) has been working with CALPUFF model for air quality studies for many years (Puliafito et al., 2007a, 2007b; Allende et al., 2008). One of the objectives of the group is to have an on-line model for urban air quality prediction. However, the wall time clock required for the model to simulate a domain with several emission sources would not allow on-line implementation, even with last generation CPUs, because of the inherent serial implementation of CALPUFF. Nevertheless, proper parallelization of the code could lead to important speedup of the simulation.

This paper shows the procedure for porting a critical routine of CALPUFF to run on CUDA, and presents a general methodology to work with any other application. The performance improvement and inherent error due to hardware architecture differences are discussed in depth. Further possible optimizations are also proposed at the end.

2 METHODOLOGY

The approach suggested in this work consists on identifying the portion of code that takes most of the computational time, then analyze the source code to find any possible computation within that module which could be more efficiently solved by multiple threads, and finally rewrite that module to run on the GPU (Delgado et al., 2010).

The development platform is an Intel Pentium III Xeon @ 2.66 GHz with NVIDIA GeForce 9400GT graphic card.

2.1 Profiling

The first step is to identify the candidate module for optimization through parallelization. There are two possible approaches to solve the problem. Having a good understanding of how the model has been implemented would point out immediately what is the best candidate module for optimization. Although CALPUFF is open source, doing a deep analysis of the code would be very time consuming. The alternative, shown in this paper, is to do a performance analysis of the software on runtime with a profiling tool.

Profiling is a form of dynamic program analysis (as opposed to static code analysis); is the investigation of a program's behavior using information gathered as the program executes. The usual purpose of this analysis is to determine which sections of a program to optimize, to increase its overall speed, to decrease its memory requirement or both (Fenlason and Stallman, 1992).

As we are currently running the model on a GNU/Linux system, we chose gprof as the profiling tool, which comes as part of the GNU development tools (Fenlason and Stallman, 1992). Gprof relies on a sampling process in order to measure functions execution time. It requires the application to be compiled in a special way, and it adds some overhead to the code, which could interfere with execution time measurements, but only in the case of routines or functions that run only a small amount of the overall time (Fenlason and Stallman, 1992). As we expect to find a very time consuming routine, this statistical inaccuracy is not to be considered.

Profiling requires the application under study to be run normally, taking care that execution flows reaches every routine in the code, or at least those that are of interest for the purpose of the analysis. In the specific case of CALPUFF, results will depend on how the model is setup. A real case of study of pollutant emission modeling over Buenos Aires city was used for the analysis (Allende D. et al., 2010).

2.2 Test Case

The domain consist of an area of 25Km x 25Km, centered at Buenos Aires city, with 500m spatial resolution, and 10 vertical levels, from ground up to 3000m above see level, logarithmically separated; see Figure 1.

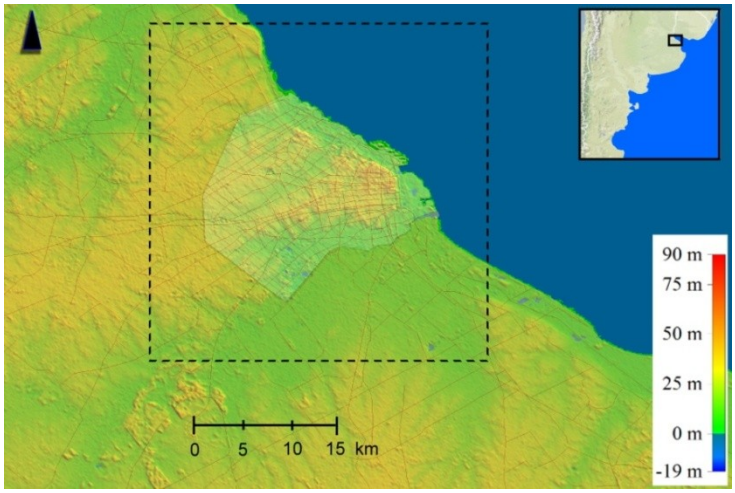

Figure 1: test case domain for profile analysis.

The emissions inventory used was developed by GEAA, and includes industrial, residential and mobile sources. Species modeled were CO, SO_2, NOx and PM_{10}. The simulation time was set to 408 hours. Wet removal and dry deposition options were turned off. This setup is particularly important because computation of wet and dry fluxes is carried out in the module we ported to CUDA, as we will explain later.

2.3 Profiling results

Because of the long time required by the model to complete the simulation, we decided to separate the sources by type, so we did four independent runs, and we got four execution profiles. Table 1 presents a summary of gprof results for one of the simulations; however, conclusions are valid for all of them.

% time	cumulative seconds	self seconds	self calls	name
16.71	108.45	108.45	1272192710	pufrecs_
13.08	193.34	84.89	2445448647	pfscrn_
12.8	276.39	83.05	18092280	calcpf_
11.75	352.67	76.28	2634726116	sigty_
9.49	414.27	61.6	1145982519	pfsamp_
7.01	459.75	45.48	797162413	vcoup_
6.58	502.42	42.67	1272192710	slgfrac_
5.1	535.5	33.08	922607992	sigtz_
4.67	565.82	30.32	1066744244	xvy_
2.92	584.79	18.97	1145982519	erfdif_
2.3	599.71	14.92	1272192710	ctadj_
1.97	612.51	12.8	1290232186	setcsig_
1.77	623.99	11.48	1	comp_
0.58	627.76	3.77	18092280	setpuf_

Table 1: gprof results, part 1. List of functions ordered by CPU time consumed.

Results show that almost 17% of the time is spent in a subroutine called *pufrecs*. Table 2 shows the chain of routine calls, the second part of gprof results. The line in yellow corresponds to the function of interest. All the lines above (in light blue) that are for calling functions and the lines below (in orange) are for functions called by *pufrecs*. In this case, the only caller of *pufrecs* is a function called *calcpf*, which, for a given puff, perform the loop over all receptors for each species. Examination of the

code reveals that the concentration for all species and for each grid receptor is computed in this module, see Figure 2.

% time	self	children	called	name	index
	108.45	204.93	1272192710/1272192710	calcpf_	[4]
48.3	108.45	204.93	1272192710	pufrecs_	[5]
	12.62	51.47	1272124810/1290232186	setcsig_	[9]
	36.83	14.64	1272124810/2634726116	sigty_	[6]
	42.67	0	1272192710/1272192710	slgfrac_	[11]
	31.34	0.41	873941350/922607992	sigtz_	[12]
	14.92	0	1272192710/1272192710	ctadj_	[15]
	0.02	0.02	259853/1099295	grise_	[39]
	0	0	259853/274949	stktip_	[68]

Table 2: gprof results, part 2. Functions call chain.

Figure 2 shows a summarized version of the original code of subroutine *calcpf*, where a nested loop is implemented for computing concentration at each receptor influenced by current puff. *chisam(x,y,s)* contains the accumulated concentration at grid receptor of coordinates *(x,y)* and for each *s* specie.

```
      do 100 isamp=il,ir
         xr=float(isamp)
         do 100 jsamp=jb,jt
            yr=float(jsamp)
c
c ---       Compute nearest approach of puff to receptor and screen out
c ---       receptors that are too far away
            call pfscrn()

c ------------------------------------------------------------------------
c ---       Obtain receptor-specific sigmas and puff height (gradual
c ---       rise) including any terrain adjustment to height
c ------------------------------------------------------------------------
            call pufrecs()

c ------------------------------------------------------------------------
c ---       For each species, compute concentration at grid receptor.
c ------------------------------------------------------------------------
               do ipol=1,nspec
                  conc = vdpvd(ipol)*xtemp*(qold(ipol)*t1+qnew(ipol)*t2)
                  chisam(isamp,jsamp,ipol) = chisam(isamp,jsamp,ipol)+conc
               enddo
  100    continue
  101 continue
```

Figure 2: portion of the original CALPUFF code. Nested loop for concentration calculation.

We proposed a thread-per-receptor decomposition of this nested loop, solving contribution of current puff to each receptor in parallel.

2.4 Porting code to CUDA

NVIDIA has developed a set of C language extensions to ease CUDA programming (NVIDIA Corporation, 2010). Unfortunately, CALPUFF is written in Fortran 77. Consequently, we primarily rewrote all the necessary functions called in the loop in C, and checked that the results were correct. Finally, we wrote the CUDA kernel which launches one thread per grid receptor. Figure 3 shows a brief version of the kernel and how threads are distributed among the cores on the GPU. All the initialization

routines, called initialization phase in CALPUFF, were kept unchanged and run on the CPU as in the original version. These routines are represented by the blue box at the top of the flow chart in Figure 3. When the computational phase begins, it copies necessary data from CPU memory to GPU, sets the kernel parameters and calls the kernel. One thread is created for each grid receptor affected by the current puff, as the central box in the flow chart shows.

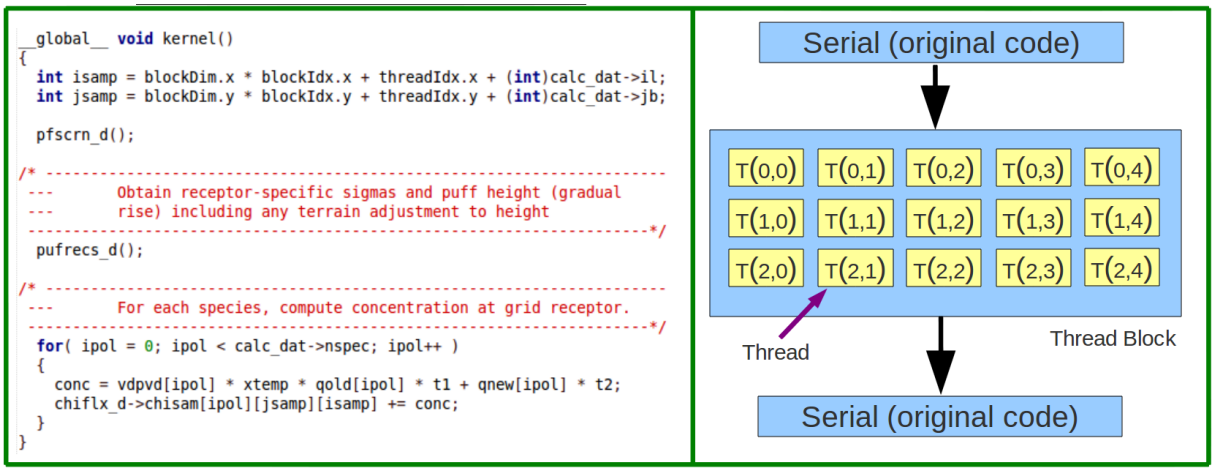

Figure 3: CUDA version of CALPUFF. Left: portion of kernel code. Right: Flow chart of CALPUFF.

The number of threads that the kernel actually launches simultaneously depends on the number of CUDA cores available on the device. So, the code can automatically take advantages of new devices without the need to be modified or recompiled.

After the kernel finishes computation, it returns control to the CPU, which continues running the serial code, and copies back the results from the GPU memory to the main memory.

Two problems arose when porting the module to CUDA. First, as a consequence of the intrinsically serial implementation of CALPUFF, the code makes uses of lots of COMMON BLOCKS in order to share variables among subroutines. When running more than one thread concurrently, it is necessary to have local copies of these structures in order to avoid race conditions.

The second problem is the amount of data that needs to be copied to GPU before kernel launch. As CALPUFF is written in Fortran 77, it has a lot of huge, fixed size arrays, which dimension is set at compile time. GPU memory should be kept synchronized with main memory to avoid errors in calculation. Frequently copying from main memory to GPU global memory turns into a bottleneck, decreasing performance even below that of the original application. We used a debugging tool (NVIDIA Corporation, 2010) to find all the routines in the original code that modify data structures needed on the GPU and managed to update them only when changed by the CPU instead of copying all data before every call to the kernel.

3 RESULTS

For evaluation of the ported version of CALPUFF, we setup and run a 24 hours simulation for the test case described in section 2.2. Benchmark reveals a 30% simulation time reduction, compared with the original version of the code. This speedup could be greater for GPU devices with more CUDA cores and higher clock rates. Table 3 shows the output of *time* GNU utility for the original and the CUDA versions of CALPUFF.

Original	CUDA
real 1m2.573 s user 1m1.868 s sys 0m0.056 s	real 0m40.613 s user 0m39.286 s sys 0m1.012s

Table 2: benchmark results. Output of GNU *time* utility for 24 hours simulation.

Figure shows 4 hour-averaged concentration of SO_2 (in ug/m3) for the original and the CUDA version of CALPUFF. For other species, results are similar.

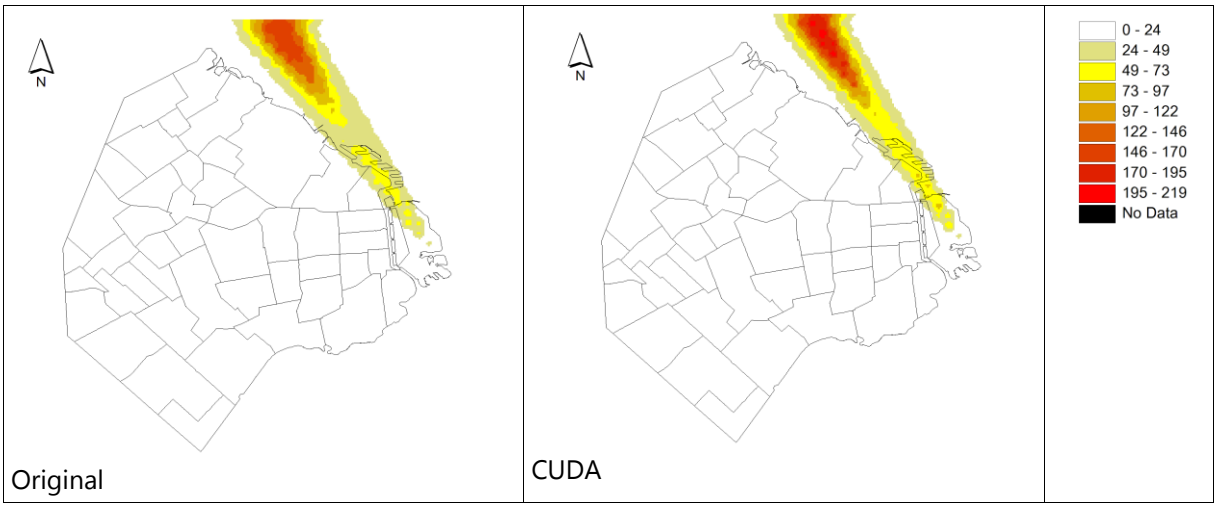

Figure 4: 4-hours average concentration map (in ug/m^3)output of original CALPUFF (left) and CUDA version (right).

The output of CUDA version shows a narrower plume, with higher concentration values along the plume axis. Moreover, pick values are higher for the optimized version, than for the original version. A later analysis with a debugging tool and a review of original code revealed that the algorithm relies on comparisons between

integer and float variables, without proper cast, for determining if current puff affects a given grid receptor or not. Result of such a comparison depends on the hardware architecture on which the application runs. This could explain differences between results for the two versions.

Another source of error is the use of fast math functions from CUDA math libraries, which are faster than the standard C functions but less accurate.

4 CONCLUSIONS

We have successfully ported a module of CALPUFF to CUDA, taking advantages of fine grain parallelism, achieving an overall speedup of 30%. This paper shows how low-cost/high-gflops-per-second GP-GPUs can effectively reduce simulation time for numerical weather prediction and dispersion modeling.

We have shown a general procedure to port any application to CUDA, by doing a runtime analysis of the software to identify computational intensive modules.

Though 30% speedup is not enough for supporting strong development of a CUDA version of CALPUFF model, we have identified further possible optimizations and a theoretical speedup of x5 (see section 5). Moreover, the code could take advantages of future developments on CUDA architecture.

5 FUTURE PLANS

In order to keep the code as simple as possible, we decided not to include calculation of wet and dry deposition fluxes in the kernel, as CALPUFF has an option to turn these features off. We are planning to include these routines in the kernel soon.

After profiling the CUDA version of CALPUFF, we noted that even though we are keeping to the minimum the number of copies between main memory and GPU memory, there is an important performance penalization from this process. As an exercise for determining the maximum speedup that could be obtained, we removed all the synchronization routines between CPU and GPU memory. Although the results are not correct, the model runs 5 times faster than the original version, which is a lot faster compared with the 30% performance improvement obtained. Further analysis of the code reveals that most of the data structures that are frequently copied to GPU, are only necessary on routines running on the device. Consequently, it is unnecessary to have these structures on main memory. The next step will be to port to CUDA all the routines that set or modify those structures.

Moreover, the profile results show other opportunities for optimizations, as there are other routines that are now the most time consuming. So, with the same procedure, a new kernel could be developed to parallelize other routines in the code.

REFERENCES

Allende, D., Castro, F., and Puliafito, E., Air Pollution Characterization and Modeling of an Industrial Intermediate City. *International Journal of Applied Environmental*

Sciences, 5:275–296, 2010.

Allende, D., Cremades, P., Puliafito, E., Perez Gunella, F., Fernandez, R., Estimación de un factor de riesgo de exposición a la contaminación urbana para la población de la Ciudad de Buenos Aires. *Avances en Energías Renovables y Medio Ambiente*, 2010.

Allende, D., and Puliafito, E., Comparación de modelos de dispersión en el modelado de emisiones gaseosas industriales del Gran Mendoza. *Proyecto Integrador para la Mitigación de la Contaminación Atmosférica PROIMCA*, 2008.

Delgado, J., Gazolla, J., Clua1, E., and Sadjadi, S.M., An Incremental Approach to Porting Complex Scientific Applications to GPU/CUDA. *In Proceedings of the IV Brazilian E-Science Workshop, XXX Brazilian Computer Science Conference, Belo Horizonte, Brazil*, July 2010.

Fenlason, J. and Stallman, R., The GNU Profiler. *Free Software Foundation, Inc*., 1992.

Graham, S.L., Kessler, P.B., and McKusick, M.K., gprof: a Call Graph Execution Profiler. *Computer Science Division Electrical Engineering and Computer Science Department University of California, Berkeley Berkeley*, 1993.

Komatitsch, D., Michéa, D., and Erlebacher, G., Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *J. Parallel Distrib. Comput*., 69:451-460, 2009.

Michalakes, J., and Vachharajani, M., GPU Acceleration of Numerical Weather Prediction. *National Center for Atmospheric Research*, 2009.

Molnár, F.Jr., Szakály, T., Mészáros, R., and Lagzi, I., Air pollution modelling using a Graphics Processing Unit with CUDA. *Computer Physics Communications*, 181:105–112, 2010.

NVIDIA Corporation, CUDA-GDB (NVIDIA CUDA Debugger), Santa Clara, California, January 2010.

NVIDIA Corporation, NVIDIA CUDA C Programming Guide, Version 3.1.1, Santa Clara, California, July 2010.

Puliafito E., and Allende, D., Calidad de aire en ciudades intermedias. *Revista Proyecciones*, 5:33-52, 2007a.

Puliafito, E., and Allende, D., Patrones de emisión de la Contaminación Urbana. *Revista Facultad de Ingeniería Universidad de Antioquia*, 42:38-56, 2007b.

Scire, J.S., Strimaitis, D.G., and Yamartino, R.J., A User's Guide for the CALPUFF Dispersion Model (Version 5). *Earth Tech, Inc. 196 Baker Avenue Concord, MA 01742*, January 2000.