CODIFING TURBULENCE ON GP-GPU FOR THE CCATT-BRAMS CODE: THE BRAZILIAN ENVIRONMENTAL PREDICTION SYSTEM

Leandro dos Santos Lessa

Renata Sampaio da Rocha Ruiz

Haroldo Fraga de Campos Velho

leandro_santos_lessa@yahoo.com.br

[renata, haroldo]@lac.inpe.br

Instituto Nacional de Pesquisas Espaciais (INPE)

Av. dos Astronautas 1758, 12227-010, São José dos Campos, SP, Brazil

Otavio Migliavacca Nadalosso

Cezar Augusto Contini Bernardi

Andrea Schwertener Charão

otaviomadalosso@gmail.com

[cbernardi, andrea]@inf.ufsm.br

Av. Roraima 1000, 97105-900, Camobi – Santa Maria, RS, Brazil

Abstract. *Atmospheric, ocean, and environmental prediction models are codes with intensive computation. One strategy to increase the performance is the use of hybrid architecture, combining CPU and accelerators devices, as GP-GPU (or simply GPU) and FPGA. The geophysical prediction codes can be characterized as having three modules: dynamical engine (used to integrate the Navier-Stokes equations), components for physics (representing turbulence, radiation, cloud dynamics, and precipitation), and geo-physical data (maps for topography, surface ocean temperature, soil moisture). This paper reports the turbulence module codification on GPU for the environmental code CCATT-BRAMS (Chemical Coupled Aerosols Tracers-Transport + Brazilian Regional Atmospheric System), developed and supported by the CPTEC-INPE (Centro de Previsão de Tempo e Estudos Climáticos). The tests were carried out using CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language). Numerical experiments were performed on a South American region, with 40 km for horizontal resolution. The results have shown a speed-up from 2 up to 14 times faster for GPU compared to serial CPU.*

Keywords: *CCATT-BRAMS, turbulence routine, GP-GPU, CUDA, OpenCL.*

1 INTRODUCTION

The environmental prediction model CCATT-BRAMS (Freitas et al., 2007; CPTEC: CATT-BRAMS) is a large and complex computational code, and the system is operationally employed by the Center for Weather Prediction and Climate Studies (CPTEC) from the National Institute for Space Research (INPE). This model allows atmospheric simulation with emission, transport, pollutant dispersion, and chemical reactions, contributing for the studies and forecasting on air quality to Brazil and South America.

The model requires intense computation, and it has been a target for research on strategies to speed up its performance. One issue under investigation in the project "*Massive atmosphere II: scaling atmospheric models to heterogeneous architectures with 10 K cores*" (CNPq support: Proc. 560178/2010-7, RFP: 09/2010) is the adaptation of the CCATT-BRAMS model on the heterogeneous computer systems, combining CPU with some accelarator (co-processor), such as Graphics Processing Units (GPUs).

The use of GPUs to speed up processing is already noticed as an eficient solution for several applications in scientific computing. However, there are challenges to be overcome for each new application. In addition, today has several programming tools dedicated to GPU, enhancing the options for investigation under consideration.

Geophysical prediction models (dynamics of atmosphere, ocean, air quality, ionosphere, and so on) apply numerical methods for space and time integration. The model resolution dictates the computational effort involved. Such models embrace several modules: geophysical data (topography, surface covering (water, ice, grass, forest, etc), dynamics (representing the convection/advection), physics (radiation, precipitation, turbulence). Here, two frameworks for GPU programming are evaluated for codifying the Smagorinsky turbulence parameterization.

2 CCATT-BRAMS MODEL

The system for environmental modeling CCATT-BRAMS is a software package developed by the CPTEC-INPE, requiring a lot of computation. It is already run on a parallel version, but depending on the model resolution (topography, and/or mesh points), even for high performance machines, this is a heavy computer model. Therefore, the research to improve the performance is always pursued.

For the numerical models applied to atmospheric dynamics, a good representation for the associated physical phenomena is an essential issue to obtain good forecasting. The first and the closest atmospheric layer to the surface should contain a representation for turbulence. Several parameterization were/are developed by the scientific community. One of them is due to Smagorinsky (1963), where the Reynolds fluxes are described using space discretization parameters. The latter turbulence model is one option for the CCATT-BRAMS.

Installing the CCATT-BRAMS is not a simple task, because the system has several files needing appropriated set up for compiling. Many dependencies (external libraries) must be carefully compiled. The computer code is written in Fortran 90, using about 530 files, summing more than 380 thousand lines of code.

For dealing with all complexitues, and taking into account preliminar studies, we focused on the GPU codification of the Smagorinsky parameterization: subroutine `MXDEFM`. The sub-routine uses over 190 programming lines, and it has some nested loops, acting over 3D arrays. Its execution time depends on the model resolution. For regular simulations, a call to the `MXDEFM` sub-routine requires dozens of miliseconds – in general, for the CCATT-BRAMS simulation, there is no dominance on a specific sub-routine.

As already known, for a relatively fast execution (even for low resolution), an effecient parallelization can be difficult. Therefore, it is relevant to evaluate alternative techniques to improve the performance, since the physics parameterization is called many times during a given simulation. High resolution and long period integration (climate studies, for example) increase the execution time.

2.1 Smagorinsky turbulence parameterization

In the Smagorinsky's formulation, the turbulent fluxes are parameterized using the theory of grandient-flux, the theory-K. In this parameterization, the tubulent fluxes (Reynolds tensors) are given as following:

$$\overline{u_i' u_j'} = -(K_m)_{ij}(D)_j \tag{1}$$

where $(K_m)_{ij}$ is the turbulent eddy diffusivity for the momentum-i on direction-j and

$$(D)_j \equiv \frac{\partial \overline{u}_i}{\partial x_j} + \frac{\partial \overline{u}_j}{\partial x_i} \ . \tag{2}$$

being $\overline{u}_i$ the mean wind component to direction-i, and x_i is the space direction (here: $i, j = 1, 2, 3$). The formulation was up-dated by Lilly (1962) and Hill (1974), the eddy diffusivity for the vertical direction can be parameterized as follows:

$$K_{mv} = \left(cs_z \Delta z\right)^2 \left[\left|D_v\right| + H(N)\right] f(R_i) \tag{3}$$

where cs_z is a fitting coefficient, Δz is the vertical grid discretization, and the term $|D_v|$, the magnitude to the deformation tensor for vertical direction, given by

$$D_v \equiv \left[\left(\frac{\partial \overline{u}}{\partial z}\right)^2 + \left(\frac{\partial \overline{v}}{\partial z}\right)^2\right]^{1/2} \ . \tag{4}$$

The contribution from the advection in the turbulence production $H(N)$ is expressed as

$$H(N) = \sqrt{\max\{0, -N^2\}} \ . \tag{5}$$

being $N = [(g/\theta)(d\theta/dz)]^{1/2}$ the Brunt-Väisälä frequency (g is gravity, and θ is the potential temperature), and the function $f(R_i)$ is written as

$$f(R_i) = \max\{0.1 - (K_{h,v}/K_{m,v})R_i\} \tag{6}$$

with K_{hv}/K_{mv} the ratio between the heat and momentum eddy diffusivities, and R_i is the gradiente Richardson number. The ratio K_{hv}/K_{mv} is especified by the user. The Smagorinsky parameterization is codified in the `MXDEFM` sub-routine.

3 PROGRAMMING ON GPU

3.1 CUDA system

CUDA (Compute Unified Device Architecture) (Nvidia: CUDA, 2012) is a framework developed by NVIDIA Corporation, looking at the parallel processing for the GPUs produced by this company. The CUDA programming could be considered an extension of languages like C, C++, and Fortran, adding qualifiers (for instance: *_global_*, *_device_*, or *shared*) to functions and data, producing kernels for execution on a GPU.

The submited job to a kernel can be divided among thousands of threads, organized into blocks, and grids, with dimensions *blockDim* and *gridDim*, respectively. The kernel uses indexes *blockIdx* and *threadIdx* for helping to define the job to be executed by a thread.

The CUDA system can only be used with Nvidia's GPU. Although the standard CUDA is widespread, and even with the opening CUDA source code made by NVIDIA, CUDA is a proprietary platform, implying restrictions on portability.

3.2 OpenCL system

OpenCL (Open Computing Language) (Khronos Group: OpenCL, 2012) is a framework supported by the consortium Khronos Group, as an open library for heterogeneous computing, employing GPUs or others devices associated to the CPU. The OpenCL programming uses APIs for communication with the devices and a language based on C for kernel especifications, to be executed on supported accelerators.

Several actions must be explicitly performed in the CPU for using the OpenCL API. Among them: GPU identification¸ buffers allocation, number of the parameters for the kernel, number of threads, and the way adopted to do the execution. The kernel execution is divided among threads, or *work-items* (using the OpenCL terminology). The threads are organized on *work-groups*, equivalent to the *blocks* in CUDA, and the work-groups are indexed by local (inside the group) or global identifiers, respectively obtained by functions *get_local_id*() or *get_global_id*() (the last, without equivalent in CUDA). Each thread deals with differents data transfered to the GPU. The data return is done by buffers reading as defined as writing buffers, through other calls done by the CPU. As an open and independent standard, OpenCL appears as a convenient alternative for demanding performance and portability.

4 DEVELOPMENT

The parallelization procedures used for both GPU frameworks are described in this session. Secondly, some particular characteristics of each approach is mentioned. Only the codification of sub-routine MXDEFM is evaluated on the GPU. A profile for the sub-routine was carried out to identify parallelization opportunities. GPU parallelism was mainly focused on loops in the sub-routine. One important requirement is that the loops must not have dependencies with previous loops. For instance, for a loop with N iterations, iteration N-1 should not depend on results from iteration N-2.

The mentioned dependency was not verified, but some aspects are important to note: (a) nested loops are conditionally activated, depending on the set-up of the simulation,

(b) some loops perform few operations, implying on a limit to how much computation can be parallelized in the GPU version. Therefore, one nested loop was selected as the parallelization target, with dominance over the simulation for a given configuration of CCATT-BRAMS.

After this first evaluation, analysis on the definition of necessary parameters for the GPU kernel codification was performed. For this point, parts of the original Fortran code were re-codified in C, for becoming easier to work with the extensions employed in CUDA and OpenCL. There is a compiler with support to CUDA on Fortran (PGI CUDA Fortran), but there is no such compiler for OpenCL. Therefore, a mixing code using Fortran and C was developed. C and Fortran Intel compilers were applied.

Some parameters to be transferred to CUDA and OpenCL kernels are 3D arrays. One issue here, with hibrid programming (Fortran and C), is that the array records in memory are different for C and Fortran. For C, the vectors with more than one dimension (matrices) are allocated as row-major, while Fortran uses column-major. For dealing with these features, and looking at the work division to the parallel execution, the multi-dimension arrays were mapped to a one-dimension vector. This strategy allows the direct acess to the C language for these data, for transfering parameters with Fortran is always done by memory addresses, i.e., C pointers. Therefore, the parameters for the kernel, for CUDA or OpenCL, were transfered this way, from a Fortran call, where the multi-dimensional arrays were considered as one-dimensional ones.

Defined the strategy for data transfer, definitions for input and output variables, vectors, and matrices were established for CUDA and OpenCL, and then the allocation of buffers in the GPU memory. The latter procedure is important to allow reading and changing the data during kernel execution. The buffers are also necessary for CUDA and OpenCL callings, for input parameters definition from the kernel, only after this data is available to the GPU.

For the kernel generation, the job was divided in parts, each one executed in a GPU thread. After the kernel execution, a reading is done from the output buffers for collecting the processed data.

4.1 CUDA Implementation

CUDA implementation required few adaptation on the original MXDEFM sub-routine. Before calling to the kernel, the multi-dimensional arrays are mapped into a one-dimensional ones. The resulting arrays, coming from the GPU, were copied to the original arrays after the GPU execution.

The C code, an file with extension ".cu", employed CUDA calls to manager allocation, transfer, and clean up the memory (*cudaMalloc*, *cudaMemcpyAsync*, *cudaMemcpy*, and *cudaFree*). The kernel on CUDA needed to call some mathematical functions, all of them available in the CUDA API (see Figure 1).

4.2 OpenCL Implementation

The OpenCL framework is able to run for GPUs made from different companies. It is necessary to identify the device where the kernel will be executed. This process requires the application of calls to identify and/or allocate one or more devices (*clGetDeviceIDs*), contexts (*clCreateContext*), and command queues (*clCreateCommandQueue*).

```
__global__ void cuda_kernel_mxdefm(int akm,
                        int N,
                        float cc3,
                        float *dn1,
                        float *dn2,
                        float *dn3,
                        float *scrv ) {
  float tmp;
  int tid = threadIdx.x + blockDim.x * blockIdx.x;

  if (tid < N) {
    tmp= akm * 0.075 * powf(dn1[tid], 0.666667);
    scrv[tid] = dn2[tid] * fmax(tmp1, cc3 * dn1[tid] * sqrt(dn3[tid]));
  }
}
```

Figure 1: Turbulence parameterization: part of CUDA codification.

In our first experiments, the context set-up requires a long period of execution time in OpenCL. However, this call needs to be done only one time, using the same defined context for all simulation time-steps. This action needs to change the external code to the sub-routine MXDEFM, generating the context and making it available into a module to be accessed by the sub-routine.

The context is a data structure defined in C, which need to be handled in Fortran. OpenCL has no Fortran support, but there exists an independent module called FortranCL (http://code.google.com/p/fortrancl/), offering an OpenCL interface for this language. This allows the generation of an OpenCL context before simulation iterations call the sub-routine.

During code development, an advantage was noted using the flag *CL MEM ALLOC HOST PTR*. This flag does a new memory allocation, when building the buffers, and it was applied instead of the flag *CL MEM USE HOST PTR*. The latter flag just uses pointers for the memory already allocated.

```
__kernel void ocl_kernel_mxdefm(__global int *akm,
                                __global int *N,
                                __global float *cc3,
                                __global float *dn1,
                                __global float *dn2,
                                __global float *dn3,
                                __global float *scrv) {
  int tid, tmp1;
  tid = get_global_id(0);
  if (tid < *N) {
    tmp1 = *akm * 0.075 * pow(dn1[tid], 0.666667);
    scrv[tid] = dn2[tid] * maxmag(tmp1, *cc3 * dn1[tid] * sqrt(dn3[tid]));
  }
}
```

Figure 2: Turbulence parameterization: part of OpenCL codification.

5 RESULTS

In our experiments, the CCATT-BRAMS uses a computational grid with 70×70×30 grid points (two horizontal directions, and vertical direction). The number of timesteps used in the

simulation was 6480. Identical outputs were obtained for three platforms: CPU alone, CPU + GPU-CUDA, and CPU + GPU-OpenCL.

One machine employed to performe the experiments is a server with processor Intel Xeon E5620 2.4Ghz, 12 GB DDR3 (RAM), GPU Nvidia Tesla M2050 3 GB GDDR5 (hereafter GPU-1). The operational system is Debian 6 (Squeeze) 64 bits, kernel 3.9.2, with OpenCL 1.2, and compilers Intel icpc 12.1.4 and ifort 12.1.3. The code was prepared for recording execution time of different parts of the code.

Other hardware environment is a desktop with processor Intel Core-i7 3.06 GHz, 12 GB (RAM), GPU Nvidia GForce GTX 580 1.5 GB GDDR5 (hereafter GPU-2). The operational system is Linux Ubuntu 10.04 LTS 64 bits, with CUDA release 4.2, and compilers Intel icoc 12.0.4 and ifort 12.0.4. Only CUDA implementation was tested here.

Table 1 shows the time average obtained on different parts of the code. The average was computed from 23 samples (simulation time-steps), without consideration with the time spent with initialization for the OpenCL code (procedures before the simulation time-steps) – this period spends 1 second (1000 ms), approximately. Following Table 1, most of the execution time with OpenCL is taken with the kernel compilation (*clCreateProgramWithSource*), buffers reading (*clEnqueueReadBuffer*), and making memory available (*clReleaseMemObject*), the computation requires only 3.5% of the total time.

Something similar is also verified with CUDA codification. A "long period" of time is concentrated on data transfer, from the CPU to GPU, and vice-versa. Less time is spent on the computation. For the computational system used, the first data transfer took much more time. Neglecting this first data transfer, the average for total execution time is 1.11 ms (little lower than OpenCL), where the difference is mainly on allocation, decresing to 0.228 ms.

Table I: Execution time for OpenCL and CUDA sub-routine.

OpenCL parcial code	time (ms)	CUDA parcial code (GPU-1)	time (ms)	CUDA parcial code (GPU-2)	Time (ms)
clCreateCommandQueue	0.043	cudaMalloc + cudaMemcpyAsync (CPU to GPU)	52.397 + 0.353	cudaMalloc + cudaMemcpyAsync (CPU to GPU)	50.924+0.308
clCreateBuffer	0.012				
clCreateProgramWithoutSource	0.337	cuda_kernel_mxdefm_<<<...>>>(,,,)	0.019	cuda_kernel_mxdefm_<<<...>>>(,,,)	0.016
clSetKernelArg	0.008				
clEnqueueNDRangeKernel	0.045	cudaMemcpy (GPU to CPU)	0.319	cudaMemcpy (GPU to CPU)	0.571
clEnqueueReadBuffer	0.380	cudafree	0.174	cudafree	0.001
clReleaseMemObject	0.267				
Total	**1.263**	**Total**	**53.003**	**Total**	**51.820**

Results for *cudaMalloc* were obtained without activating the CUDA function *cudaThreadSyncronize*. Activating this CUDA function, the total execution time measured is

(GPU-2): 0.609 ms [*cudaMalloc* = 0.380 + *cudaMemcpyAsync* (CPU to GPU) = 0.017 + *cuda_kernel_mxdefm* = 0.20 + *cudaMemcpy* (GPU to CPU) = 0.010 + *cudafree* = 0.001).

6 CONCLUSION

A parallel implementation on GPU for the turbulence parameterization routine in the CCATT-BRAMS environmental prediction model was performed. Two frameworks were used in this implementation: CUDA and OpenCL. For both implementations, more execution time was employed to transfer data, between CPU and GPU, than the computation in the device. One important issue was to mapping the computational effort associated to each part of the CUDA and OpenCL codes. These results are going to help future implementations for other CCATT-BRAMS routines on GPU execution.

Our results have also shown different performances on different hardware devices (see execution time for GPU-1 and GPU-2 in Table 1). More investigations are necessary for better understanding such desagreement.

ACKNOWLEDGEMENTS

This project has been supported by FINEP under contract CT-INFO Edital Grade-01/2004. Authors thanks to the CNPq and FAPESP, Brazilians agencies for research support. The authors thank two anonymous referees for reviewing that helped make the text clearer.

REFERENCES

S. R. Freitas, S.R., Longo, K., Dias, M., Chatfield, R., Dias, P., Artaxo, P., Andreae, M., Grell, G., Rodrigues, L., Fazenda, A., & J. Panetta, 2007 "The coupled aerosol and tracer transport model to the brazilian developments on the regional atmospheric modeling system (CATT-BRAMS). part 1: Model description and evaluation". *Atmos. Chem. Phys. Discuss*., vol. 7, pp. 8525–8569.

CPTEC-INPE, "Modelo ccatt-brams / qualidade do ar," 2012, available at: http://meioambiente.cptec.inpe.br/modelo_cattbrams.php (accessed at: 29-Dec-2012).

Lilly, D. K., 1962, "On the numerical simulation of buoyant convection". *Tellus*, v. 14, pp. 168–172.

Hill, G. E., 1974 Factors controlling the size and spacing of cumulus clouds as revealed by numerical experiments". *J. Atmos. Sci.*, v. 31, pp. 646–673.

NVIDIA Corporation, "CUDA Toolkit Documentation," 2012, available at: http://docs.nvidia.com/cuda/ (accessed at: July-2013).

Hinton, Khronos Group, "The OpenCL specification – version 1.2," 2012, available at: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf (Accessed at: 30-Dec-2012).

Smagorinsky, J., 1963 "General circulation experiments with the primitive equations," *Mon. Weath. Rev.*, vol. 91, pp. 99–164.