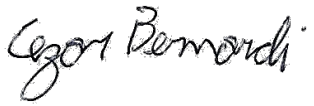
Relatório de Atividades Desenvolvidas

Uso de GPU para Aceleração de Simulações Atmosféricas com o Modelo CCATT-BRAMS.

Cezar Augusto Contini Bernardi[1]
Haroldo Fraga de Campos Velho[2]



_____	_____
Coordenador	Bolsista

[1] Bolsista INPE/CNPq, aluno de Ciência da Computação - UFSM
[2] Orientador, pesquisador do Laboratório Associado de Computação e Matemática Aplicada do INPE.

RESUMO DO PLANO DE TRABALHO

O inicio dos trabalhos foi dedicado aos estudos preparatórios, relativos a entendimento da arquitetura e funcionamento de GPGPUs (General Purpose Graphic Processing Units), além de analises das rotinas da parte de turbulência do CCATT-BRAMS, com intuito de identificar a melhor opção à ser explorada utilizando o *framework* OpenCL. Nesta fase também foram medidos os tempos gastos pelas rotinas, afim de ajudar na identificação da melhor opção.

Em sequência, passou-se a codificação de *kernels* em OpenCL para integração e testes de melhoria do desempenho do CCATT-BRAMS.

ATIVIDADES REALIZADAS

Inicialmente, foi realizado um estudo preparatório, com o intuíto de entender o funcionamento das GPGPUs, do *framework* utilizado e do CCATT-BRAMS. Nesse estágio foram feitos diversos exemplos afim de aprimorar o entendimento de arquiteturas paralelas visando a melhor implementação das rotinas de turbulência em OpenCL.

Após diversas análises e exercícios, passou-se a codificação da rotina *tkemy*, responsável pela parametrização de Mellor-Yamada, sendo publicado um artigo no ERAD 2013 com os resultados obtidos. Posteriormente, foi feita também a implementação da rotina *mxdefm*, responsável por cálculos de turbulência segundo Smagorinski, juntamente com uma implementação em CUDA feita por parceiros no projeto, rendendo também um artigo no WSCAD-WIC 2013 com a comparação entre os dois métodos e a aplicação original, em anexo.

PROGRESSOS E RESULTADOS

Com a implementação OpenCL concluída e testada em separado, foi feita a integração do código OpenCL, baseado em C, com o código em Fortran do CCATT-BRAMS. Com a integração finalizada, foram feitas medidas de tempo de execução para comparação com a versão original da rotina. Os resultados obtidos dessas primeiras tomadas de tempo não foram satisfatórias, pois o tempo que era gasto com transferência da grande quantidade de dados ultrapassava muito o tempo ganho na execução em si, anulando qualquer possibilidade de ganho geral de desempenho.

Então, analisando o *kernel*, viu-se uma possibilidade de melhoria deste, abrindo a possíbilidade de ganho de desempenho. Infelizmente, devido a algum possível erro de implementação, as saídas obtidas com o novo *kernel* continham diferenças muito grandes em relação a versão original, apesar do ganho de desempenho que foi alcançado no tempo de execução.

Visando encontrar novas oportunidades de melhoria, o *kernel* relativo a rotina *tkemy* foi deixado de lado e passou-se a implementação em OpenCL da rotina *mxdefm*, que havia sido codificada em CUDA por parceiros no projeto.

Sendo essa implementação mais fácil e baseada em código CUDA, que divide semelhanças com OpenCL, o caminho até os resultados foram mais ágeis.

Apesar das simplificações em estruturas de dados que foram aplicadas nessa tentativa, buscando melhores tempo de transferência e acesso, os resultados obtidos com ambas as implementações,

comparados a versão original, não foram satisfatórios. Também não apresentaram ganho de desempenho, devido ao mesmo problema encontrado anteriormente: a grande quantidade de dados sendo transferida em relação ao tempo de processamento. Após a redação e apresentação do artigo no WSCAD 2013, descobriu-se que o tempo extra tomado pela primeira iteração se deve ao carregamento do módulo do driver, pois o programa estava sendo testado em uma máquina que não fazia uso deste constantemente.

Mesmo assim, os ganhos no tempo de execução não foram suficientes para superar os tempos gastos em transferência de dados. Assim, concluiu-se que o CCATT-BRAMS não é um bom alvo de melhorias através desses frameworks, mesmo as rotinas tendo a estrutura desejável para esse tipo de aplicação.

Comparação entre CUDA e OpenCL na Parametrização de Turbulência em GPU para o Modelo Ambiental CCATT-BRAMS

Cezar Bernardi*, Otavio Madalosso*, Andrea Charão*,
Leandro Lessa†, Renata Ruiz† e Haroldo de Campos Velho†
*Laboratório de Sistemas de Computação (LSC)
Universidade Federal de Santa Maria (UFSM),
Av. Roraima, 1000 – 97105-900 – Santa Maria – RS – Brasil
Email: {cbernardi, omadalosso, andrea}@inf.ufsm.br
†Laboratório Associado de Computação e Matemática Aplicada (LAC)
Instituto Nacional de Pesquisas Espaciais (INPE)
Av dos Astronautas, 1758 – 12227-010 – São José dos Campos – SP – Brasil
Email: {renata, haroldo}@lac.inpe.br

***Resumo*—O modelo CCATT-BRAMS é um sistema de grande porte utilizado operacionalmente na previsão ambiental pelo CPTEC-INPE. O modelo permite simulações atmosféricas com emissão, transporte e interação química de poluentes. Por exigir computação intensiva, este modelo tem sido alvo de pesquisas sobre estratégias de aceleração. O presente estudo mostra resultados da combinação de CPU e GPU. Apresenta-se a exploração dos frameworks CUDA e OpenCL em uma sub-rotina do módulo de parametrização física de turbulência no modelo CCATT-BRAMS, visando sua paralelização em GPU. Os resultados colocam em evidência desafios na paralelização da sub-rotina escolhida e permitem comparar algumas características de CUDA e OpenCL, com impacto no desempenho do código.**

I. Introdução

O sistema de modelagem ambiental CCATT-BRAMS [1], [2] é um código de grande porte empregado operacionalmente na previsão ambiental pelo Centro de Previsão de Tempo e Estudos Climáticos (CPTEC) do Instituto Nacional de Pesquisas Espaciais (INPE). Este modelo permite simulações atmosféricas com emissão, transporte e interação química de poluentes, contribuindo para estudos e previsões sobre a qualidade do ar no Brasil.

Por exigir computação intensiva, o modelo tem sido alvo de pesquisas sobre estratégias de aceleração. No contexto do projeto Atmosfera Massiva II [1], investiga-se o comportamento do modelo CCATT-BRAMS e de outros modelos atmosféricos em arquiteturas heterogêneas, combinando CPUs com dispositivos aceleradores, tais como *Graphics Processing Units* (GPUs).

O uso de GPUs para aceleração do processamento já mostrou ser uma solução eficiente para uma variedade de aplicações em computação científica. No entanto, sabe-se que há desafios a enfrentar a cada nova aplicação investigada. Além disso, atualmente existem várias ferramentas de programação para GPU, aumentando assim as vias de investigação a serem consideradas.

Neste trabalho, compara-se os *frameworks* CUDA e OpenCL de programação para GPU, aplicados à paralelização de uma sub-rotina do modelo CCATT-BRAMS. A abordagem comparativa utilizada permite ressaltar particularidades de cada *framework* e, ao mesmo tempo, colocar em evidência desafios de paralelização do modelo em GPU.

[1]Financiamento CNPq, Proc. 560178/2010-7, Edital N. 09/2010

II. Fundamentação

A. Programação em CUDA

CUDA (*Compute Unified Device Architecture*) [3] é um *framework* desenvolvido pela NVIDIA Corporation, visando o processamento paralelo em GPUs fabricados por esta empresa. A programação em CUDA se dá por meio de extensões a linguagens como C, C++ e Fortran, adicionando qualificadores (por exemplo, `__global__`, `__device__` ou `shared`) a funções e dados, compondo *kernels* para execução em GPU.

O trabalho em um *kernel* é potencialmente dividido entre milhares de *threads*, organizadas em *blocks* e *grids* com dimensões `blockDim` e `gridDim`, respectivamente. No interior de um *kernel*, usam-se os índices `blockIdx` e `threadIdx` para auxiliar a delimitação do trabalho a ser executado por uma *thread*.

A disponibilidade de CUDA restringe-se a GPUs NVIDIA. Além disso, embora a especificação CUDA seja amplamente disponível, e mesmo com a abertura de código-fonte CUDA feito pela NVIDIA, CUDA ainda é uma plataforma proprietária, tendendo a restringir a portabilidade de programas que a utilizam.

B. Programação em OpenCL

OpenCL (*Open Computing Language*) [4] é um *framework* mantido pelo consórcio Khronos Group, como um padrão aberto para computação heterogênea, utilizando GPUs ou outros dispositivos aceleradores junto à CPU. A programação em OpenCL utiliza APIs para comunicação com dispositivos e uma linguagem baseada em C para especificação de *kernels*, que executam nos aceleradores suportados.

Com OpenCL, há várias ações que devem ser realizadas explicitamente em CPU, usando a API OpenCL. Dentre elas, tem-se o reconhecimento da GPU, a alocação de

buffers, a definição dos argumentos do *kernel*, número de *threads* e forma com que essas farão a execução.

A execução do kernel é dividida entre as *threads* ou *work-items*, na terminologia OpenCL. Estas se organizam em *work-groups*, equivalentes aos *blocks* em CUDA, e se diferenciam por identificadores locais (dentro do grupo) ou globais, obtidos respectivamente através das funções `get_local_id()` ou `get_global_id()` (esta última, sem equivalente direto em CUDA). Cada *thread* manipula diferentes dados transferidos a GPU. O retorno dos dados é feito pela leitura dos *buffers* que foram definidos como *buffers* de escrita, através de outras chamadas feitas pela CPU.

Por ser um *framework* aberto e padronizado, independente de fabricante, OpenCL aparece como uma alternativa conveniente para sistemas computacionais que demandem desempenho e portabilidade.

C. Modelo CCATT-BRAMS

O sistema de modelagem ambiental CCATT-BRAMS é utilizado operacionalmente para previsão climática em todo o Brasil [1], [2]. Desenvolvido pelo INPE, este software requer alto poder de processamento. Mesmo já sendo paralelizado para execução em *cluster*, uma rodada do modelo pode consumir muito tempo, dependendo dos dados de entrada.

A instalação do CCATT-BRAMS é um tanto trabalhosa, por envolver diversos arquivos e ajustes nas configurações de compilação. Além disso, muitas das dependências (bibliotecas externas) precisam ser compiladas de forma adequada para funcionarem no sistema. Seu código, escrito em Fortran 90, é distribuído em cerca de 530 arquivos, totalizando mais de 380.000 linhas.

Em modelos numéricos da atmosfera, a adequada representação dos fenômenos físicos é imprescindível para se obter previsões confiáveis. Na primeira camada da atmosfera, que está em contato direto com o solo e com os oceanos, um dos efeitos que não podem ser negligenciados é o da turbulência. Para representação da turbulência atmosférica, diversas parametrizações foram produzidas pela comunidade científica. O modelo CCATT-BRAMS emprega diferentes parametrizações de turbulência, incluídas no código que trata da física do modelo.

Dada as dimensões do modelo CCATT-BRAMS, e considerando o caráter exploratório do presente trabalho, escolheu-se apenas uma sub-rotina para investigação da paralelização em GPU. A sub-rotina escolhida, chamada MXDEFM, é responsável pela parametrização de turbulência de Smagorinsky [5]. Essa sub-rotina distribui-se por cerca de 190 linhas e conta com laços de repetição aninhados, que iteram sobre *arrays* tridimensionais. Seu tempo de execução depende principalmente do tamanho da grade de simulação utilizada, sendo que, em simulações comuns em computadores atuais, uma chamada a MXDEFM dura na ordem de unidades de milissegundos [2]. Tratando-se de uma execução relativamente rápida, sabe-se que sua paralelização eficiente pode ser difícil. Mesmo assim, há interesse em melhorar seu desempenho, uma vez que é uma sub-rotina chamada muitas vezes durante uma simulação. Além disso, simulações maiores podem exigir mais tempo.

[2]Em uma simulação com CCATT-BRAMS, geralmente não há uma sub-rotina que domine o tempo de execução.

III. Desenvolvimento

A abordagem utilizada neste trabalho restringiu-se à sub-rotina MXDEFM, buscando acelerar sua execução em GPU. O trabalho inicial de paralelização, independente do *framework*, consistiu em analisar o código para identificar oportunidades de paralelização. Tendo a GPU como arquitetura paralela alvo, a análise concentrou-se nos laços presentes na sub-rotina. Um importante requisito é que os laços não tenham dependências com os laços anteriores. Por exemplo, em um laço de i a n, a rodada n-1 não pode ter um acesso à posição n-2, o que poderia causar acesso a dados indefinidos.

Não encontrou-se nenhuma dependência que impossibilitasse a paralelização, mas notou-se que: (i) os laços aninhados eram acionados condicionalmente, dependendo de uma opção de configuração da simulação e (ii) alguns laços realizavam poucas operações, limitando a quantidade de cálculo a ser executado em paralelo na GPU. Assim, escolheu-se apenas um dos laços aninhados como alvo da paralelização, cujo processamento era dominante para uma das opções de configuração da simulação.

Passou-se, então, à definição de parâmetros necessários à codificação do *kernel* em GPU e analisou-se como seriam os recebimentos e acessos a esses parâmetros. Neste ponto, foi necessário traduzir parte do código original em Fortran para C, devido às extensões de linguagem utilizadas nos *frameworks* CUDA e OpenCL. Sabe-se que existe um compilador (PGI CUDA Fortran) com suporte direto a CUDA em Fortran, mas não há algo equivalente para OpenCL, por isso neste trabalho optou-se pela utilização mista de C e Fortran, com compiladores Intel.

Analisando-se os parâmetros que deveriam ser passados ao *kernel* CUDA ou OpenCL, notou-se que alguns desses representavam *arrays* tridimensionais. Para utilizar estes parâmetros, foi necessário atentar para o fato de que a alocação de memória em Fortran difere da forma feita em C. A passagem de parâmetros em Fortran sempre é feita através de referências a endereços de memória, ou seja, ponteiros em C. Sendo assim, os argumentos do *kernel*, tanto em CUDA como em OpenCL, foram todos recebidos dessa forma, a partir de uma chamada em Fortran, fazendo com que os *arrays* n-dimensionais fossem interpretadas como unidimensionais.

No passo seguinte foram feitas as definições de variáveis, vetores e matrizes de entrada e saída, preparação dos dispositivos, contextos, filas de comandos, alocação dos *buffers* na memória da GPU, compilação e execução do *kernel* de forma paralela, ou seja, dividindo em pequenas partes que são executadas cada uma em uma *thread* da GPU, e leitura dos *buffers* de retorno.

Um empecilho na comunicação entre C e Fortran, é a forma de alocação de vetores n-dimensionais. Em C, os vetores com mais de uma dimensão (matrizes) são alocados como *row-major*, enquanto em Fortran, essa alocação é *column-major* [6]. Por esse motivo, foi feito um achatamento das estruturas, copiando todos os dados de vetores n-dimensionais para vetores unidimensionais, permitindo com que o acesso a essas estruturas em C fosse feita de forma direta.

Além disso, foi necessária a alocação de *buffers* na memória da GPU. Esta alocação é necessária para que os dados possam ser lidos e alterados durante a execução do kernel. Esses *buffers* também são necessários para as chamadas OpenCL responsáveis pela definição dos parâmetros de entrada do *kernel*, para só então estarem realmente disponíveis para a GPU. O código fonte do *kernel* é compilado através de uma chamada OpenCL, em tempo de execução e alocado em um programa. Este programa é utilizado para a criação da fila de comando a ser executada em GPU. Após a execução, é feita a leitura dos *buffers* de saída para coletar os dados processados.

A. Implementação usando CUDA

A implementação em CUDA exigiu poucas alterações na sub-rotina MXDEFM original. Antes da chamada ao *kernel*, foi adicionado um código para criar *arrays* unidimensionais a partir dos *arrays* n-dimensionais originais, conforme descrito nos parágrafos anteriores. Os *arrays* resultantes, provenientes da GPU, foram copiados para os *arrays* originais após a execução em GPU.

O código em C, criado em um arquivo .cu, utilizou chamadas CUDA para gerenciar alocações, transferências e liberações de memória (`cudaMalloc`, `cudaMemcpyAsync`, `cudaMemcpy` e `cudaFree`). O *kernel* em CUDA precisou invocar algumas funções matemáticas, todas elas disponíveis na API CUDA (ver figura 1).

B. Implementação com OpenCL

Sendo o OpenCL um *framework* compatível com várias fabricantes de GPUs presentes no mercado, e dada a grande diversidade de diferentes modelos desses fabricantes, é necessário fazer a identificação do dispositivo no qual será executado o *kernel*. Esse processo envolve a utilização das chamadas que identificam e alocam um ou mais dispositivos (`clGetDeviceIDs`), contextos (`clCreateContext` e filas de comando (`clCreateCommandQueue`).

Em experimentos iniciais, notou-se que a criação de contexto exigida pelo OpenCL tomava muito tempo de execução, e que esta chamada poderia ser feita apenas uma vez, utilizando o mesmo contexto para os diversos passos de simulação. Para isso, no entanto, foi necessário alterar código externo à sub-rotina MXDEFM, criando o contexto e disponibilizando-o em um módulo acessível à sub-rotina.

O contexto é uma estrutura de dados definida em C, que precisaria ser manipulada em Fortran. O OpenCL não tem suporte para Fortran, mas existe um módulo independente chamado FortranCL [3], que oferece uma interface OpenCL para esta linguagem, possibilitando a criação de um contexto OpenCL antes da execução dos passos de simulação que invocam a sub-rotina.

Ainda, durante o desenvolvimento do código, notou-se a vantagem da utilização do *flag* CL_MEM_ALLOC_HOST_PTR, que faz uma nova alocação de memória, durante a criação dos *buffers*, no lugar do *flag* CL_MEM_USE_HOST_PTR, que apenas utiliza ponteiros para a memória já alocada.

IV. Resultados

O código foi instrumentado a fim de medir tempos envolvidos nas diferentes partes do código, utilizando-se o relógio do sistema e clGetEventProfilingInfo para a execução em GPU OpenCL. O computador utilizado para os testes possui um processador Intel Xeon E5620 2,4Ghz, 12 GB DDR3 de RAM e uma GPU Nvidia Tesla M2050 3 GB GDDR5. O sistema operacional instalado é o Debian 6 (Squeeze) 64 bits, kernel 3.9.2, com OpenCL 1.2 e compiladores Intel icpc 12.1.4 e ifort 12.1.3.

A tabela 1 apresenta os tempos médios obtidos em cada parte do código, sendo a média de 23 *timesteps*. Porém, a tabela OpenCL desconsidera o tempo gasto com a criação do contexto antes do início dos *timesteps*, passo que leva aproximadamente de 1 segundo (1000 ms). Como pode ser visto na tabela 1, a maior parte do tempo da execução em OpenCL se concentra na compilação do kernel(clCreateProgramWithSource), leitura dos buffers (clEnqueueReadBuffer) e liberações de memória (clReleaseMemObject), tendo apenas 3,5% do tempo destinado aos cálculos.

Algo semelhante também é visto em CUDA, muito do tempo total se concentra em transferência de dados, de CPU para GPU e vice versa, e pouco na parte do código onde se faz os cálculos. Ainda, nos tempos em CUDA, se for desconsiderada a primeira rodada, a média total de tempo fica em 1.11 ms, sendo que a diferença se concentra na alocação de memória, que cai para 0.228 ms.

V. Conclusão

Neste trabalho, apresentou-se a paralelização em GPU de uma sub-rotina do modelo ambiental CCATT-BRAMS, utilizando os *frameworks* CUDA e OpenCL comparativamente. Ambas as implementações evidenciaram sobrecargas em algumas operações, principalmente no que diz respeito ao manuseio da memória. Isso, aliado ao fato de que a sub-rotina executa relativamente rápido nos experimentos efetuados, tornou a paralelização ineficiente, sendo a versão original, executada totalmente em CPU, mais rápida. Mesmo assim, foi possível comparar detalhadamente os custos envolvidos em cada parte dos programas implementados em CUDA e OpenCL, o que constitui um subsídio importante para desenvolvimentos futuros, visando adaptar outras partes do modelo CCATT-BRAMS para execução em GPU.

[3] http://code.google.com/p/fortrancl/

```
__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]));
  }
}
```

Figura 1. Extrato de código da implementação em CUDA

```
__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]));
  }
}
```

Figura 2. Extrato de código da implementação em OpenCL

Tabela I
TEMPOS DE EXECUÇÃO DA SUB-ROTINA EM OPENCL E CUDA

Parte do programa (OpenCL)	Tempo de execução (ms)*	Parte do programa (CUDA)	Tempo de execução (ms)
clCreateCommandQueue	0,043		
clCreateBuffer	0,012	cudaMalloc (alocação de memória na GPU) + cudaMemcpyAsync (cópia dos dados da CPU para GPU)	52.397 + 0.353
clCreateProgramWithSource	0,337		
clSetKernelArg	0,0008		
clEnqueueNDRangeKernel	0,045	cuda_kernel_mxdefm<<< ... >>>(...); (execução do kernel)	0.019
clEnqueueReadBuffer	0,380	cudaMemcpy (cópia dos dados da GPU para CPU)	0.319
clReleaseMemObject	0,267	cudaFree	0.174
Total	1,263	Total	53,003
Total Serial	2,549		

REFERÊNCIAS

[1] S. R. Freitas, K. Longo, M. Dias, R. Chatfield, P. Dias, P. Artaxo, M. Andreae, G. Grell, L. Rodrigues, A. Fazenda, and J. Panetta, "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, 2007.

[2] CPTEC-INPE, "Modelo ccatt-brams / qualidade do ar," 2012, disponível em: http://meioambiente.cptec.inpe.br/modelo_cattbrams.php Acesso em: 29 dez. 2012.

[3] NVIDIA Corporation, "CUDA Toolkit Documentation," 2012, disponível em: http://docs.nvidia.com/cuda/ Acesso em: jul. 2013.

[4] Khronos Group, "The OpenCL specification – version 1.2," 2012, disponível em: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf Acesso em: 30 dez. 2012.

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

[6] A. Agay and A. Vajhoejo, Eds., *User Notes on Fortran Programming*. Ibiblio, 1998, ch. Array Storage Order.