Paradigma dataflow para aplicação científica em GPGPU

Luiz E. S. Evangelista[1], Álvaro L. Fazenda[1], Vinícius V. de Melo[1]

[1]Instituto de Ciência e Tecnologia – Universidade Federal de São Paulo (UNIFESP)
Rua Talim, 330 – 12231-280 – São José dos Campos – SP – Brasil

{levangelista, alvaro.fazenda, 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 be also be addressed.*

Resumo. *Este trabalho visa estudar técnicas de computação paralela com uso de GPU (“Graphics Processing Unit”) com o objetivo de otimizar o desempenho de um trecho de um código computacional implementado como um sistema DataFlow. Tal código constitui parte de um modelo numérico para meteorologia, responsável por calcular o fenômeno de transporte por advecção. As possíveis limitações no algoritmo que impedem melhor eficiência em GPU também deverão ser tratadas.*

1. Introdução

A quantidade de dados processados em simulações numéricas atinge, atualmente, valores antes inimagináveis. Por esse motivo, programação paralela tem sido largamente empregada, pois permite a divisão de uma tarefa complexa em várias menores e independentes. Como em qualquer ramo da computação, essa técnica evoluiu ao longo do tempo e atualmente permite a aplicação de dispositivos externos à CPU (*Central Processing Unit*) como co-processadores, tais como as atuais GPUs (*Graphics Processing Units*) [Kirk e Hwu, 2010].

Uma das maneiras de se programar GPUs é por meio do padrão OpenACC [OpenACC, 2014], com o qual utilizando-se um conjunto de diretivas de compilação pode-se especificar um trecho de código a ser acelerado por meio de execução nos citados dispositivos. A principal vantagem desse modelo é o fato de exigir pouco esforço na reestruturação de um código já existente, quando comparado à tradicional linguagem CUDA [Kirk e Hwu, 2010], por exemplo. Além disso, toda a transferência dos dados entre a CPU e a GPU pode ocorrer de maneira transparente para o desenvolvedor.

Neste trabalho, procurou-se otimizar o desempenho de um código computacional desenvolvido com abordagem *dataflow* para a simulação numérica de transporte por advecção, de forma a obter eficiência no desempenho em GPUs por meio da introdução de diretivas do padrão OpenACC.

Alguns trabalhos foram desenvolvidos para se procurar soluções para problemas relacionados ao assunto estudado: Hartley *et. al.* (2010) propõe um esquema para divisão de tarefas através de um sistema *Dataflow* que permite modificar

dinamicamente o tamanho das tarefas a serem disparadas e balancear o sistema; Boulos *et. al.* (2012) resolvem um problema relacionado a processamento de sinais em GPU através de um modelo baseado em um sistema Dataflow, Wang *et. al.* (2012) aborda o desenvolvimento de um sistema Dataflow que otimiza o desempenho do processamento em GPU para uma aplicação de análise de sinais.

Esse documento está organizado da seguinte forma: O Capítulo 2 descreve o algoritmo para cálculo da advecção, o capítulo 3 trata da aplicação do sistema *Dataflow* ao problema, o capítulo 4 demonstra os resultados obtidos e finalmente o capítulo 5 disserta sobre as conclusões e desenvolvimentos futuros.

2. Cálculo de Advecção

O conceito de advecção pode ser entendido como um processo de transferência de energia por movimento horizontal. No contexto deste trabalho, o processo aplica-se à transmissão do calor por meio do movimento horizontal do ar atmosférico [Freitas *et al.,* 2011], agindo, por exemplo, na movimentação horizontal de uma massa de ar ou na transferência de calor das latitudes baixas para as altas.

No artigo de Freitas *et al.* (2011) é avaliado um novo esquema numérico para advecção, chamado de advecção monotônica, aplicado a um modelo numérico para previsão da qualidade do ar chamado de CCATT-BRAMS (*Coupled Chemistry Aerosol -Tracer Transport model to the Brazilian developments on the Regional Atmospheric Modelling System*) [Longo *et. al.,* 2013], apresentando melhorias nas simulações sobre a região tropical e subtropical da América do Sul. Este software é usado operacionalmente pelo CPTEC/INPE (Centro de Previsão de Tempo e Estudos Climáticos / Instituto Nacional de Pesquisas Espaciais) como modelo ambiental na previsão da qualidade do ar (Meio Ambiente ,2013), permitindo a simulação de fontes emissoras de gases poluidores em centros urbanos, de fontes oriundas de focos de queimadas em matas, incorporando o cálculo de interações químicas entre os gases atmosféricos além de um modelo numérico completo de previsão de tempo/clima.

Segundo Fonseca (2012), a primeira codificação do modelo de advecção monotônica aplicado ao CATT-BRAMS usava programação serial, isto é, sem paralelismo, devido a restrições que impunham dependências de fluxo no algoritmo implementado para o mesmo, o qual pode ser visualizado na Figura 1, na qual é possível notar dois laços principais, responsáveis por percorrer um *array* nos sentidos positivo e negativo.

Observando o algoritmo da Figura 1 percebe-se que os laços principais percorrem toda a extensão de um *array*, o qual representa o domínio discretizado do problema. Sendo que o primeiro laço busca calcular o transporte advectivo para sequências de células onde o valor do vetor velocidade do vento em um determinado eixo coordenado do espaço tridimensional é positiva ($u_i>0$). O segundo laço faz um procedimento semelhente para as células onde a direção da velocidade do vento é negativa ($u_i<0$). Entretanto, para as sequências de células encontradas, ambos os procedimentos apresentam uma dependência de fluxo relacionada a $Qn_i = Fa(Flux_i)$ e $Flux_i = Fb(Flux_{i-1},Qn_i)$, no primeiro laço e $Qn_i = Fa(Flux_i)$ e $Flux_{i-1} = Fb(Flux_i,Qn_i)$ no segundo laço, sendo que *Fa* e *Fb* representam operações aritméticas com os dados envolvidos.

```
IF(u(1)>=zr0)  flux(1)= q0(1)*u(1)*dt*dd0(1)

DO i=2,idime-1
        IF(u(i)<zr0) CYCLE
        IF(u(i-1)<zr0) THEN
                flux(i) = q0(i)*u(i)*dt*dd0(i)      !  outflow←only cell
        ELSE                                      !  use upstream
                x1  = dt*u(i)/dxx(i)                !  Courant number
                x1n = (1.-x1)*(q0(i-1)←q0(i-1))/4.
                cf = q0(i) - x1n                                 !Eq←4a

                IF(imxmn(i-1)) cf = q0(i) -MAX(1.5,1.2  -.6 *x1)*x1n   !Eq←10b
                IF(imxmn(i-1)) cf = q0(i) -        (1.75 -.45*x1)*x1n   !Eq←10a

                cf1      = MIN( MAX( cf, MIN(q0(i),q0(i-1))  ), MAX(q0(i),q0(i-1))
                qn(i)    = MAX(  vcmin(i),   MIN(   vcmax(i),          &   !eq←3&8
                          (q0(i)*den0(i)-x1*cf1*dd0(i)-flux(i-1)/dxx(i))/den1(i) ))
                flux(i) = dxx(i)*(q0(i)*den0(i) - qn(i)*den1(i)) → flux(i-1)
        END IF                                                !Eq←9a
END DO

IF(u(idime-1)<zr0)  flux(idime-1)=q0(idime)*u(idime-1)*dt*dd0(idime-1)

DO i = idime-1,2,-1
        IF(u(i-1)>=zr0) THEN            ! Inflow←only cell
                IF(u(i)<zr0) qn(i) = MAX(  vcmin(i),   MIN(   vcmax(i),&
                                     (q0(i)*den0(i)-flux(i)/dxx(i) - flux(i-1)/dxx(i))/den1(i) ))
        ELSE
                x1  =  dt*ABS(u(i-1))/dxx(i)      ! Courant number
                x1n = (1.-x1)*(q0(i-1)-q0(i-1))/4.
                cf  = q0(i) - x1n                                 !Eq←4b
                IF(imxmn(i-1)) cf = q0(i) -MAX(1.5,1.2  -.6 *x1)*x1n   !Eq←10b
                IF(imxmn(i-1)) cf = q0(i) -   (1.75 -.45*x1)*x1n       !Eq←10a
                cf1 = MIN( MAX( cf, MIN(q0(i),q0(i-1)) ), MAX(q0(i),q0(i-1)) )

                IF(u(i)>=zr0) cf1= q0(i)      ! outflow←only cell upstream
                qn(i)= MAX(  vcmin(i),  MIN(   vcmax(i),           &   !Eq←3&8
                      (q0(i)*den0(i)-flux(i)/dxx(i)-x1*cf1*dd0(i-1))/den1(i) ))
                flux(i←1)=dxx(i)*(qn(i)*den1(i) - q0(i)*den0(i)) - flux(i)!Eq←9b
         END IF
END DO
```

Figura 1: Trecho de código serial que realiza o cálculo da advecção em uma *array*. [Fonseca, 2012]

3. Sistema *Dataflow* para cálculo de advecção

Para tratar da dependência dos dados que ocorre no algoritmo da advecção monotônica, Fonseca (2012) adotou uma abordagem por *Dataflow*, a qual possibilita identificar sequências independentes (*streams*) de dados contíguos que podem ser processados concorrentemente. Na busca por estas sequências foi desenvolvido um método para classificação de dados representando o componente do vetor velocidade que permite identificar o tipo de processamento a ser executado e suas possíveis dependências com os elementos adjacentes. Na Figura 2 é possível visualizar essa classificação.

Vel [i←1]	Vel [i]	Regra	Descrição
X	→	A	Left Border Inflow
←	X	B	Right Border Inflow
→	→	D	Positive stream
←	←	E	Negative stream
→	←	F	Inflow←only
←	→	C	Outflow←only

Figura 2: Tipos pré-definidos para classificação. [Fonseca, 2012]

Para exemplificar, na Figura 3 tem-se um exemplo de um *array* representando os elementos do componente do vetor velocidade analisado. A classificação de cada

elemento de acordo com a tabela representada na Figura 2 é demonstrada na Figura 3.

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**

Figura 3: Exemplo de conjunto de dados e sua respectiva classificação. [Fonseca, 2012]

Segundo Fonseca (2012), as regras de prioridade de cálculo podem ser representadas em um grafo acíclico direcionado (*DAG - directed acyclic graph*), conforme pode ser visto na Figura 4.

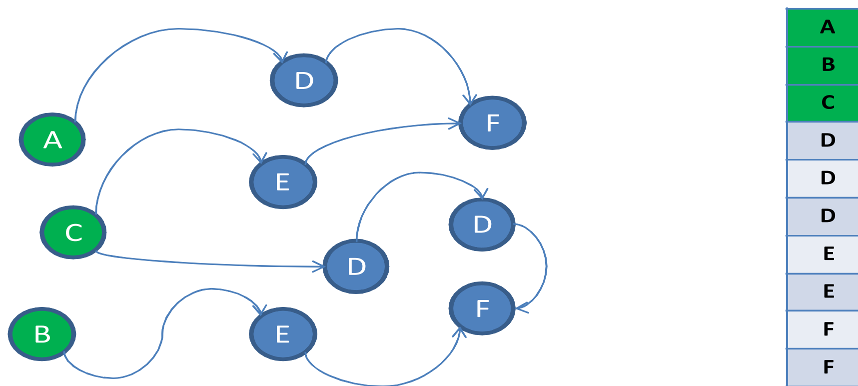

Figura 4: DAG representando a dependência entre as células e sua ordenação topológica na forma de *array,* a direita. [Fonseca, 2012]

Posteriormente, o DAG representado na Figura 4 passa por uma etapa de agregação de pontos em *streams*, sendo armazenado como uma estrutura de dados na forma de um *Heap*, o qual deverá reproduzir as regras de prioridade na solução dos *streams*. No exemplo representado pelas Figura 3 e Figura 4 foi possível identificar 3 conjuntos de dados (*streams*) independentes e dois conjuntos dependentes, os quais estão descritos na Tabela 1:

Tabela 1: Conjuntos detectados

Conjuntos independentes:	Conjuntos dependentes:
a) **A**D (1,2)	d) **F** (3)
b) E**C**DD (4,5,6,7)	e) **F** (8)
c) E**B** (9,10)	

Entretanto, o exemplo contendo um conjunto de tarefas independentes gerado (Tabela 1) é heterogêneo quanto ao esforço computacional e fluxo de execução, impondo também um padrão de acesso a memória irregular por tarefa, características estas que geralmente representam desafios na obtenção de eficiência computacional de um código executado em GPUs.

Cabe ainda citar que o algoritmo originalmente desenvolvido permite que o mesmo seja aplicado a qualquer *array* com espaçamento (*stride*) variável entre células adjacentes em um dado sentido do mesmo, o que possibilita aplicar o mesmo procedimento para matrizes multidimensionais, bastando percorrer *arrays* nos diversos eixos distintos. No testes desenvolvidos foi utilizada uma matriz tridimensional. A diferença na aplicação do algoritmo para tratar dados nos eixos *X*, *Y* e *Z* ocorre apenas na especificação do elemento inicial e do deslocamento (*stride*) entre os elementos adjacentes. No caso em questão, dada a característica da estrutura de dados do modelo meteorológico onde a advecção se aplica, a matriz 3D está organizada da forma: *ZXY*, com ordem de alocação de acordo com a regra adotada para programas em Fortran, que posiciona elementos em posições adjacentes na memória a partir do índice mais a esquerda. Além disto, o mesmo método deve ser aplicado a diversos campos meteorológicos, descritos em Freitas *et al.* (2011) como espécies químicas, com quantidades designados neste texto por *N*. Assim, a matriz que contém os dados a serem advectados tem a estrutura: *NZXY*.

O algoritmo desenvolvido por Fonseca (2012) para CPUs não é compatível com o uso de GPUs, assim, adaptações tiveram que ser implementadas para permitir a geração de paralelismo em dispositivos aceleradores. Dentre as adaptações, uma das mais importantes consistiu em armazenar as diversas tarefas mapeadas em uma lista ou *array,* em substituição à estrutura de *Heap* utilizada originalmente. Este *array* deve conter as tarefas independentes agrupadas em sequências e por último conter as tarefas dependentes. Desta forma o algoritmo em GPU desenvolvido resolve primeiramente todas as tarefas (lista de *streams*) independentes e posteriormente aquelas que apresentam dependência. A Tabela 2 mostra uma comparação entre a versão original e a versão adaptada à GPU.

Tabela 2: Trechos de código representando o percorrimento de tarefas. Algoritmo original a esquerda e versão acelerada com OpenAcc a direita (linhas em vermelho referem-se a ações executadas na GPU).

Versão original. Lista de tarefas armazenada em um *Heap*.	**Versão adaptada para GPU.** Lista de tarefas em um *array*.

while (not end(Heap))
    tarefa ← get_job(Heap)
    Advecção*(tarefa)*
    update_heap(Heap)
	Transfere dados CPU -> GPU (matrizes 4D, lista de tarefas)
	kernel em acelerador
	***for (tarefas* independentes)**
	    Advecção*(tarefas)*
	kernel em acelerador
	***for (tarefas* dependentes)**
	    Advecção(tarefas)
	Transfere dados GPU -> CPU (matriz 4D)

4. Resultados obtidos

Para se obter melhoria no desempenho computacional do modelo foram desenvolvidas diferentes versões do código (escrito em Fortran-90). A cada nova versão, modificações na estrutura de dados, no algoritmo, na configuração lógica da estrutura de dados na GPU ou modificações nas diretivas OpenACC foram implementadas. A descrição das diferentes versões usadas e/ou desenvolvidas e de seus respectivos

desempenhos estão descritas nos parágrafos seguintes.

Convém citar ainda a configuração da máquina utilizada durante o desenvolvimento: processador *Intel® Core™ i7-975* e acelerador *NVidia Tesla C2050*; bem como do compilador utilizado: *PGI Acelerator* versão 14.1-0. O conjunto de dados a ser manipulado pelo programa inclui dois *arrays* de 4 dimensões na forma $N*Z*X*Y$, sendo que N representa a quantidade de campos meteorológicos a serem advectados. Adiante, segue uma descrição das diferentes versões usadas e desenvolvidas:

1) **Código original serial em CPU (Fonseca, 2011)**: Código original, executado em CPU, para ser usado como referência.
2) **Código em CPU com OpenMP**: Adaptação do código original implementando diretivas OpenMP, permitndo o cálculo das tarefas mapeadas de forma concorrente. A implementação exigiu o uso de seção crítica para manipulação do *Heap* que representa a estrutura de dados. Alem disso, sincronizações foram acrescentados para respeitar a dependência de tarefas.
3) **Código em OpenACC não otimizado (*naive*)**: Nessa versão inicial, foram acrescentadas apenas as diretivas elementares do OpenACC dentro da rotina de cálculo da advecção, tais como seções especificando a transferência de dados entre CPU e GPU e regiões aceleradas de código (*kernels* em GPU). Para o processamento dos dados em cada eixo da matriz tridimensional, as transferências de dados eram repetidas, o que ocasionava desperdício de tempo.
4) **Ajustes de uso de blocos e vetores na GPU**: Ao se utilizar as diretivas do OpenACC, o próprio compilador define a quantidade de blocos *(blocks)* e *threads* dentro dos blocos, alocados na GPU, os quais em OpenACC podem ser mapeados para as cláusulas *Gang* e *Vector*. Mas a alocação automática dessa estrutura não se mostrou-se a mais adequada ao problema em questão. Assim, nesta versão foram explicitados novos valores para quantidade de blocos e *threads*. No laço mais externo (responsável por percorrer as tarefas a serem processadas), a quantidade ótima de blocos e *threads* foi definida como 512 e 32, respectivamente. Já no laço mais interno (responsável em percorrer os elementos químicos/campos), o valor de blocos e *threads* foi de 16 e 32. Cabe ressaltar a necessidade de se informar ao compilador, com o uso da cláusula "*independent*", que os laços deveriam ser paralelizados, pois do contrário o compilador automaticamente definia dependências onde as mesmas não existiam.
5) **Otimização de transferência de dados**: Conforme explicado anteriormente, um dos maiores problemas no uso da GPU ocorre em decorrência do tempo gasto na transferência dos dados. Foram aplicadas algumas otimizações para esse processo, definindo o escopo de abrangência de cada dado envolvido, ou seja, definindo dados de entrada, saída, entrada/saída e temporários (cláusulas "*copyin*", "*copyout*", "*copy*" e "*create*"). Essa diferenciação entre eles acabou diminuindo o tráfego dos mesmos, o que proporcionou uma redução no tempo de transferência. Outra mudança que também ocasionou um ganho de desempenho foi a modificação do algoritmo para que os dados fossem enviados uma única vez para a GPU e utilizados para processamento de todos os dados em qualquer eixo.

6) **Modificação algorítmica para adaptação à GPU**: Nessa versão, em um determinado trecho de código que possui duas estruturas de repetição, uma determinada expressão originalmente calculada apenas no laço mais externo passou a ser calculada previamente de forma separada. Tal modificação não causa nenhuma melhoria significativa quando em execução em CPU, porém em GPU permitiu melhorar a taxa de ocupação [Kirk e Hwu, 2010]. da mesma e consequentemente melhorar o desempenho.

7) **Diminuição do uso de *branches***: O uso de estruturas de decisão exige que a GPU execute, de forma preventiva, os caminhos prováveis que a aplicação irá percorrer [Kirk e Hwu, 2010]. Isso pode aumentar o tempo de execução, pois os dois caminhos possíveis devem ser executados. Dessa maneira, as estruturas condicionais (*IF's*) foram substituídas por expressões matemáticas que usam *flags* para selecionar a expressão correta. Essa ação diminuiu a média de caminhos divergentes em 39,41% em um determinado problema testado, com consequente melhoria no desempenho.

Vale registrar que foi desenvolvida uma versão extra na qual as tarefas eram ordenadas de acordo com o tamanho e tipo dos *streams* a serem processados, procurando evitar *branches* divergentes nas operações dentro de um *Warp* [Kirk e Hwu, 2010] em GPU. Todavia, essa ação não gerou melhorias significativas no desempenho.

A Figura 5 apresenta o *speedup* das respectivas versões citadas para um problema configurado da seguinte forma: 22*38*336*336 (*N*Z*X*Y*) com dados de entrada obtidos a partir de simulações reais do modelo meteorológico no qual o algoritmo se aplica:

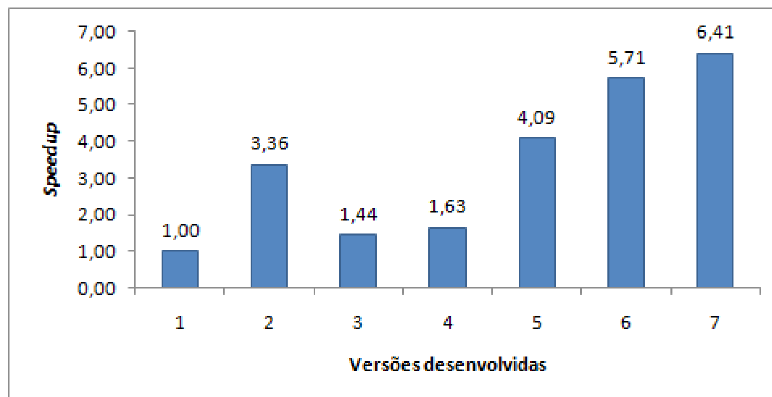

Figura 5: *Speedup* de cada versão desenvolvida

Além da informação de *speedup*, outras evidências embasam as evoluções das versões desenvolvidas para GPU. Para auxiliar nessa análise foi utilizada a ferramenta de *profiling* chamada "nvprof" [DocsNvidia, 2014], na qual pode-se extrair as seguintes métricas e eventos:

- **ipc**: Instruções executadas por ciclo.
- **gld_efficiency**: Taxa de eficiência no acesso a dados solicitados para leitura à

memória global.

- **gst_efficiency**: Taxa de eficiência no acesso a dados solicitados para escrita à memória global.
- **achieved_occupancy**: Taxa de ocupação atingida na GPU.
- **MIPS:** Razão referente à quantidade total de instruções executadas por segundo (em milhões de operações).

A Tabela 3 apresenta algumas das métricas recém descritas para as versões desenvolvidas e executadas em GPU, considerando um problema configurado da mesma forma como descrito na Figura 5. A partir da análise dessas informações é possível visualizar a evolução na eficiência do programa ao longo das versões desenvolvidas. A métrica "ipc" mostra claramente que houve aumentos sucessivos de desempenho permitindo um maior número de operações executadas por ciclo. É possível verificar também que a diminuição no uso de *branches* (versão 7) causou uma melhoria no acesso de leitura a memória global. A modificação algorítmica (versão 6) que proporcionou uma melhor taxa de ocupação na GPU, causou também uma melhoria na eficiência de acesso de escrita à memória global. Além disso, a quantidade de *Megaflops* demonstra o ganho de desempenho entre todas sucessivas versões.

Tabela 3: Métricas e eventos medidos nas versões desenvolvidas para GPU

Versão	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%	60,3%	15,3%	691176079,5	17,8
6	1,061	42,6%	78,6%	30,3%	835767651,2	24,8
7	1,189	51,9%	78,6%	30,3%	783768534,7	27.8

Outra métrica utilizada para se comprovar a eficiência do uso de memória consiste na verificação de acesso coalescente em memória global na GPU [Kirk & Hwu, 2010]. Micikevicius (2012) indica que é possível verificar se as ações leitura e escrita em memória global estão sendo executadas de forma coalescente através de métricas que se utilizam de eventos medidos pelo *profiling* "nvprof":

- ***Leitura:*** *(l1_global_load_hit + l1_global_load_miss)/gld_request* (2)
- ***Escrita:*** *global_store_transaction/gst_request* (3)

Onde os eventos citados nas fórmulas 2 e 3 referem-se à:

- ***l1_global_load_hit/l1_global_load_miss*:** Total de acessos com/sem sucesso em dados na memória *cache* L1 da GPU.
- ***gld_request/gst_request***: Total de requisições para leitura/escrita à memória global.
- ***global_store_transaction***: Número total de transações para escrita em memória global.

Micikevicius (2012) indica que as taxas indicadas deverão ter um valor aproximado de 1.0 (um) quando se manipulam dados de 32 *bits* (*float*). Valores acima de 1.0 para as métrica das fórmulas 2 e 3 indicam acessos não coalescentes. Na última versão desenvolvida, as métricas obtidas para leitura e escrita foram 1,33 e 1,63, respectivamente. Assim, é possível notar que o acesso não é perfeitamente coalescente, o qual deverá implicar em futuras investigações para que se identifique suas causas e planeje-se alterações no código visando minimizar seus efeitos.

Foram também realizados testes para se medir o desempenho do algoritmo aplicado a dados dispostos nos diferentes eixos coordenados da grade computacional tridimensional, conforme mostrado na Tabela 4, considerando dimensões da grade com a mesma configuração citada para a Figura 5.

Tabela 4: Desempenho do algoritmo avaliado por eixo da matriz tridimensional

Eixo	**Tempo (segundos)**	**Gigaflops**
X	0,1026	66,6
Y	0,1037	65,4
Z	0,0998	70,0

De acordo com a Tabela 4 não foi observada diferença significativa de desempenho entre os dados distribuídos em diferentes eixos, apesar da diferente localidade em memória dos dados vizinhos em cada eixo da matriz tridimensional. Os cálculos efetuados no eixo *Z* apresentam discreto menor desempenho em relação aos demais, o que causa surpresa, visto que os dados vizinhos no eixo *Z* são alocados fisicamente de forma adjacente na memória global, o que não ocorre com os demais eixos. Tal característica sugeria que a melhor eficiência no acesso aos dados na memória fosse causar melhoria de desempenho, o que não se efetivou. Entretanto, deve-se lembrar que a quantidade de tarefas, e consequentemente de *streams*, bem como sua natureza, são completamente distintas para cada eixo da matriz.

Foi verificada também a escalabilidade do programa desenvolvido em relação ao tamanho do domínio do problema. Para tanto foram efetuados diversos testes variando o tamanho da matriz tridimensional representando o domínio de cálculo, bem como a quantidade de campos meteorológicos advectados. Ao alterar a quantidade de campos meteorológicos (variando-o de 1 a 60) e fixando o tamanho da grade computacional em 168*168*38 (*X*Y*Z*) elementos, é possível constatar que o tempo de execução não aumenta linearmente com o incremento na quantidade de campos meteorológicos, conforme se vê na Figura 6, onde percebe-se que o aumento no tempo de execução usando de 1 a 60 campos foi de apenas 3,8 vezes, aproximadamente, enquanto que o aumento na quantidade de operações de ponto flutuante variou 60 vezes. Isso pode ser confirmado observando-se a taxa de Gigaflops medidos para o mesmo caso estudado, conforme a Figura 7, onde pode-se notar que o desempenho aumenta quando se utilizam uma maior quantidade de dados a serem processados, o que reflete um comportamento típico de GPUs. Deve-se relevar, também, que o tempo de transferência apresenta um aumento de tempo não proporcional ao aumento de tempo total de processamento. Isso demonstra que quanto maior o problema, menos custo será acrescido ao tempo total com transferências de dados.

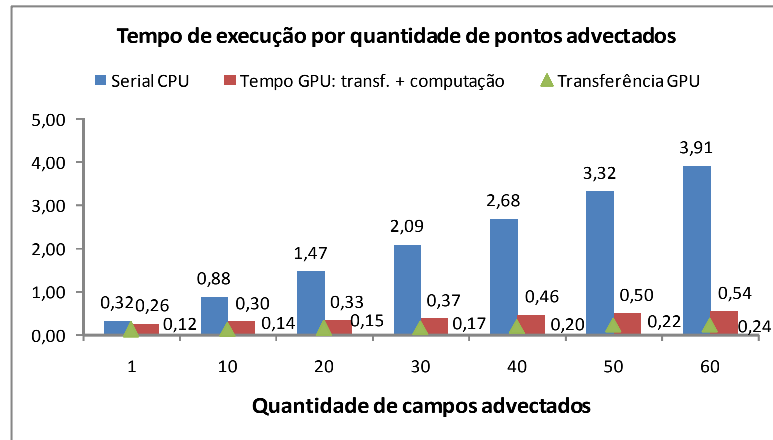

Figura 6: Variação do tempo de execução por quantidade de campos meteorológicos

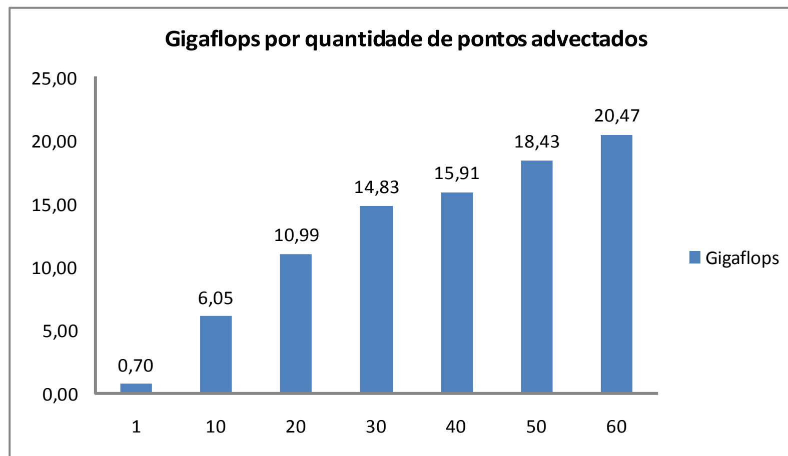

Figura 7: Variação em Gigaflops por quantidade de campos meteorológicos

Variando-se o tamanho da matriz tridimensional e fixando-se em 10 campos meteorológicos a serem advectados, a escalabilidade com o tamanho do problema tem um comportamento similar ao se variar apenas a quantidade de campos meteorológicos. A Figura 8 apresenta-se esse cenário, onde a abscissa do gráfico possui os valores para a quantidade de pontos nos eixos X e Y da matriz tridimensional (fixando-se a quantidade de pontos no eixo Z em 38). Assim, a cada medida apresentada no gráfico da Figura 8 a quantidade de elementos da matriz, e consequentemente de operações de ponto flutuante, aumenta em 4 vezes. Comparando-se o desempenho para advectar dados em uma matriz de tamanho 336*336*38 com uma matriz de 42*42*38 pontos, tem-se aproximadamente 64 vezes mais operações de ponto flutuante, entretanto, o tempo de execução é apenas 9,1 vezes maior que o valor de referência. No mesmo gráfico da Figura 8 é possível verificar também que para uma malha de tamanho 42*42*38 o tempo de transferência de dados entre CPU e GPU equivale a aproximadamente 85% do tempo total de processamento, indicando um baixo volume de dados a serem processados em relação aos dados transferidos. No entanto, para a malha de maior tamanho (336*336*48) o custo de transferência equivale a apenas 25% do tempo total. Dessa forma, o custo da transferência de dados varia com o tamanho do domínio de

cálculo, ficando progressivamente menos significativo a medida que o tamanho da malha aumenta. Importante observar que, ao se comparar a escalabilidade com o aumento na quantidade de campos (Figura 6) com a escalabilidade com o aumento no tamanho da malha (Figura 8), percebe-se que no primeiro caso o algoritmo apresenta melhor escalabilidade, uma vez que as operações envolvendo diferentes campos meteorológicos são totalmente independentes, o que permite que seja gerado mais tarefas paralelas, enquanto que o aumento no tamanho da malha pode implicar em um aumento de paralelismo de menor escala, o qual depende da análise de quantidade de *streams* independentes encontrados nos dados de entrada.

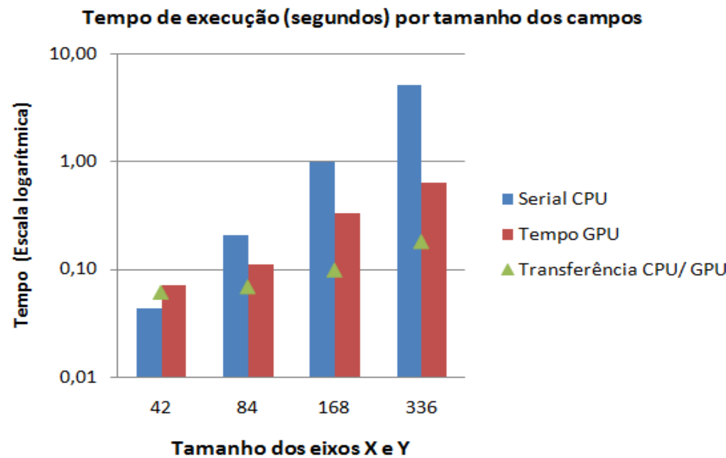

Figura 8: Variação de tempo de execução por tamanho do domínio tridimensional (10 campos meteorológicos)

5. Considerações finais

Ao longo do desenvolvimento foi observado que a simples introdução de diretivas elementares do OpenAcc não gerou códigos eficientes a serem executados em GPU. Para se otimizar o desempenho neste dispositivo diversas alterações tiveram que ser aplicadas ao código-fonte tanto na especificação de diretivas auxiliares em OpenAcc quanto modificações algorítmicas. Assim, o tempo inicial de 3,33s em GPU da primeira versão desenvolvida (1,9 GFlops) foi reduzido para 0,74s na última versão (27,8 GFlops). Testes adicionais permitiram verificar que o custo relativo com operações relacionadas exclusivamente a manipulação de memória e com operações relacionadas exclusivamente a computação são equivalentes para o algoritmo desenvolvido, mostrando um problema balanceado neste aspecto. Detectou-se ainda que existe o benéfico efeito da sobreposição (*overlaping*) de operações envolvendo memória e computação para o mesmo algoritmo desenvolvido.

O ganho de desempenho obtido com o uso de GPUs através do algoritmo desenvolvido pode ser considerado modesto, em comparação com otimizações em outros métodos de advecção ou métodos tradicionais para dinâmica de fluídos computacional (DFC), porém, para o algoritmo otimizado, devido a sua forma de solução por *dataflow*, a granularidade do paralelismo atingiu um valor que pode ser considerado alto para o uso em GPU. O problema testado em sua maior configuração apresenta aproximadamente 230 mil tarefas independentes, as quais deverão calcular dados que cobrem uma matriz tridimensional representando o domínio do problema com aproximadamente 13 milhões de células. Assim, em média, cada tarefa é

responsável pelo cálculo da advecção em 56 células. Métodos numéricos para DFC apresentam quantidades de tarefas de magnitude similar a quantidade de células, ou seja, gerando muito mais paralelismo. Assim, a granularidade alta torna-se o maior impedimento para atingir um melhor desempenho. Desenvolvimentos futuros incluem a investigação de melhorias no acesso coalescente dos dados em memória e modificações algorítmicas que permitam calcular trechos do procedimento original de forma totalmente independente para cada célula do domínio de cálculo. Comparações de desempenho entre o código OpenAcc e código em CUDA também podem ser desenvolvidas, procurando ponderar facilidade de programação com desempenho.

6. Referências Bibliográficas

Boulos, V.; Huet, V.; Fristot, V; Salvo, L; and Houzet, D. "Efficient implementation of data flow graphs on multi-gpu clusters". Journal of Real-Time Image Processing, 2012.

DocsNvidia, "*Profiler User's Guide*", http://docs.nvidia.com/cuda/profiler-users-guide, acessado em Junho/2014.

Fonseca, R. M. "Dataflow paradigm approach applied to improve computational efficiency of a monotonic advection scheme.". Workshop PADTempo-XVII Congresso Brasileiro de Meteorologia, 2012, Gramado, RS, Brasil.

Freitas, S. R.; Rodrigues, L. F.; Longo, K. M.; and Panetta, J. "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, v. 3, 2011.

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

Kirk, D., B. e Hwu, W. W. "Programming Massively Parallel Processors: a hands-on approach", Elsevier, 2010

Longo, K. M.; Freitas S. R.; Pirre, M.; Marécal, V.; Rodrigues, L. F.; Alonso, M. F.; and Mello, R. ."The chemistry-catt brams model: a new efficient tool for atmospheric". A regional atmospheric model system for integrated air quality and weather forecasting and research. Geoscientific Model Development, v. 6, p. 1389-1405, 2013.

Micikevicius, P. "GPU Performance Analysis and Optimization", http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf, 2012. Acessado em Junho/2014.

Meio Ambiente, http://meioambiente.cptec.inpe.br/modelo_cattbrams.php?lang=pt, acessado em Abril/2013.

OpenACC, "OpenACC Directives for Accelerators", http://www.openacc.org/, acessado em Maio/2014.

Rood R. B. "Numerical advection algorithms and their role in atmospheric transport and chemistry models". Rev. Geophys, v. 25, p. 71-100, 1987.

Wang L.; Shen C.; Seetharaman, G.; Palaniappa K. and Bhattacharyy S. S. "Multidimensional dataflow graph modeling and mapping for efficient gpu implementation". 2012 IEEE Workshop on Signal Proc. Systems, p.301-305, 2012.