Comparação e Análise de Desempenho de Aceleradores Gráficos no Processamento de Matrizes

Nielsen A. Gonçalves[1], Carlos A. N. Costa[2], Josivaldo de S. Araújo[1], Jessé C. Costa[2], Jairo Panetta[3]

[1]Instituto de Ciências Exatas e Naturais – Universidade Federal do Pará (UFPA)
Caixa Postal 479 – 66.075-110 – Pará – PA – Brazil

[2]Programa de Pós-Graduação em Geofísica – Universidade Federal do Pará (UFPA)
Caixa Postal 479 – 66.075-110 – Pará – PA – Brazil &
Instituto Nacional de Ciência e Tecnologia - Geofísica do Petróleo (INCT-GP)

[3]Centro de Previsão de Tempo e Estudos Climáticos- CPTEC
Instituto Nacional de Pesquisas Espaciais - INPE

Abstract. *In the last years the traditional solutions of High Performance Computing (HPC), such as the insertion or replacement of processors, have undergone major changes with the addition of new features. The use of graphics accelerators have been one of the features by which it has been possible to continue to expand the computational performance. However, as other techniques, this leads to the need for specific programming skills as well that facilitate the extraction of computational power offered by all CPU and GPU set. This paper makes a comparison between technologies like OpenACC, CUDA and OpenMP in performance evaluation for matrix processing.*

Resumo. *Nos últimos anos as tradicionais soluções da Computação de Alto Desempenho (HPC, do inglês* High Performance Computing*), como a inserção ou a substituição de processadores, vêm sofrendo grandes mudanças, com a inclusão de novos recursos. O uso de aceleradores gráficos têm sido um dos métodos pelos quais tem se tornado possível continuar a ampliar o desempenho computacional. Porém, assim como outras técnicas, esta também conduz à necessidade de habilidades específicas de programação que permitam a melhor extração do poder computacional oferecido pelo conjunto CPU e GPU. Este trabalho faz uma comparação entre tecnologias como OpenACC, CUDA e OpenMP na avaliação de desempenho no processamento de matrizes.*

1. Introdução

O uso de arquiteturas paralelas tem se tornado uma das abordagens mais utilizadas na computação científica. O avanço tecnológico resultou em equipamentos com custos mais baixos, facilitando o acesso de pesquisadores de diversas áreas e permitindo a obtenção dos resultados processados em intervalos de tempo menores. Isso pode ser evidenciado nos diversos trabalhos que demonstram melhorias significativas de desempenho obtidas em problemas de computação científica nas mais diversas áreas do conhecimento, como por exemplo, na biologia [Manavski and Valle 2008], na medicina [Pan et al. 2008], na engenharia [Souza et al. 2011], na sísmica [Abdelkhalek et al. 2009] e na economia [Bényász and Cser 2010].

Entre as arquiteturas paralelas utilizadas, uma ganhou destaque nos últimos anos, o uso dos processadores gráficos de propósito geral, que passaram a figurar, inclusive, na maioria dos equipamentos que compõem os primeiros lugares da lista Top500[1] de supercomputadores [Top500 2015].

Entretanto, a natureza heterogênea destes sistemas, apesar de viabilizar o aumento do desempenho, torna mais difícil a criação e adaptação de algoritmos, uma vez que há a necessidade de definir explicitamente os pormenores da comunicação entre os componentes do conjunto (processador, acelerador gráfico, memórias e disco) para que o nível de paralelismo desejado seja alcançado. Portanto, é cada vez mais necessário o desenvolvimento de modelos de computação paralela que implementem camadas de abstração de alto nível às arquiteturas heterogêneas, que irão "habilitar mais usuários a obter vantagens dessas arquiteturas sem a necessidade de conhecimento detalhado do hardware" [Reyes et al. 2012].

Dessa forma, tanto a indústria de GPUs (*Graphics Processing Unit*) quanto os desenvolvedores de software independentes, iniciaram o desenvolvimento de possíveis soluções. Este esforço resultou em novos modelos de programação, tanto proprietários, dentre os quais, destaca-se o CUDA (*Compute Unified Device Architecture*), no qual o programador não mais necessita utilizar diretamente a API gráfica para acessar as capacidades de computação paralela do dispositivo [Kirk and Hwu 2013]; quanto abertos, entre eles, o OpenACC (*Open Accelerators*), que é um modelo ainda mais simples e baseado em diretivas de compilação. [OpenACC 2014].

O presente trabalho tem por objetivo realizar um estudo comparativo entre modelos de computação paralela, tanto em termos de desempenho quanto de facilidade (esforço de adaptação do código-fonte) da programação. A comparação tem a finalidade de mostrar, através de um problema real da computação científica (solução de sistemas de equações lineares através da eliminação gaussiana) que é possível obter ganhos de desempenho, tanto por meio do uso de aceleradores gráficos, como o OpenACC e CUDA, quanto utilizando soluções, mais simples, como o OpenMP, utilizado em ambientes de memória compartilhada, com um esforço de programação cada vez menor.

O trabalho está organizado da seguinte forma: Na Seção 2 serão apresentados os trabalhos relacionados. Na Seção 3 será apresentado o método de Eliminação Gaussiana, utilizado na solução de sistemas de equações lineares. A Seção 4 discorrerá sobre os modelos de computação paralela utilizados como referência, como a programação em OpenMP, CUDA e OpenACC. As Propostas de Metodologia de Avaliação aparecem na Seção 5. Já os Resultados Experimentais dos testes implementados serão apresentados na Seção 6 e na Seção 7 serão expostas as conclusões e os possíveis trabalhos futuros.

2. Trabalhos relacionados

A utilização de processadores gráficos de propósito geral em aplicações científicas tem sido o alvo de diversos trabalhos. Stringhini et al. [2012] apresenta o uso de GPUs dentre as principais ferramentas utilizadas em computação heterogênea aplicável à solução de problemas científicos. Embora não haja comparativos entre os modelos apresentados, os mesmos são descritos de maneira sucinta com exemplos simples.

[1] A lista Top500 é mantida pela universidade de Mannheim, Alemanha desde 1993 e é utilizada como referência de capacidade computacional pela indústria mundial de supercomputadores.

Pilla and Navaux [2010] apresentam uma comparação entre GPUs utilizando CUDA e sistemas multicore na execução de três aplicações da classificação *Dwarf Mine* (*MapReduce*, *Spectral Methods* e *Sparse Linear Algebra*), indicando a vantagem do uso de GPUs nestas classes de problemas em termos de desempenho, eficiência energética e redução de custos em relação aos sistemas baseados apenas em CPUs multicore.

Uma GPU, com a utilização da tecnologia CUDA, é empregada por Souza et al. [2011] na aceleração do método de Otimização por Enxame de Partículas (PSO - *Particle Swarm Optimization*), apresentando como resultado diminuições significativas nos tempos de execução, bem como a produção de um código mais legível e adaptável.

Fernandes et al. [2012] menciona o uso do CUDA e do OpenACC na paralelização de modelos de Cinética Química Atmosférica, sem contudo apresentar resultados mensuráveis. O trabalho aponta algumas dificuldades encontradas no uso de GPUs em modelo de computação heterogênea. Contudo, utilização de GPUs, inclusive por meio de outras interfaces de programação, é apontada no trabalho como uma tendência.

Santos et al. [2013] apresenta o uso de GPUs aplicadas ao processamento de tráfego em redes de alta velocidade, demonstrando o aumento na vazão de processamento para o sistema apresentado. O trabalho indica que o uso de GPUs, apesar de alguns problemas relativos ao *overhead* de entrada e saída, apresenta ao fim uma capacidade de gerar uma vazão de dados superior em relação às implementações de análise de tráfego baseadas em CPU.

Uma comparação entre o CUDA o OpenACC é feita por Hoshino et al. [2013], utilizando dois algoritmos simples (multiplicação de matrizes) e uma aplicação científica real (*stencil* de 5 pontos). Como resultado dos comparativos, classificado pelo autor como "*microbenchmarks*", as versões baseadas em OpenACC conseguiram alcançar até 65% do desempenho do seu equivalente em CUDA, dependendo, no entanto, de "diversos ajustes manuais".

Ledur et al. [2013] compara o desempenho de implementações em CUDA, OpenACC e OpenMP de aplicações computacionalmente intensivas (conjunto de *Mandelbrot*, N-Rainhas e multiplicação de matrizes). As implementações baseadas em OpenACC apresentaram melhor desempenho em duas das três aplicações. O autor menciona a complexidade na construção do código como um fator limitante na produção de um código CUDA mais eficiente.

Manfroi et al. [2014] apresenta uma avaliação de desempenho de aceleradores em ambiente virtualizado, aplicadas à Álgebra Linear Densa. As GPUs apresentaram tempos de execução muito próximos aos obtidos por meio do acesso direto pelo sistema hospedeiro, mostrando que as ferramentas de virtualização avaliadas permitem o uso satisfatório de dispositivos aceleradores para a classe de problemas avaliados.

3. Sistemas Lineares e Eliminação Gaussiana

Sistemas de Equações Lineares podem ser definidos como conjuntos de equações de primeiro grau que possuem as mesmas incógnitas [Barbosa 2011]. Com base nesta característica é possível organizar o sistema na forma de uma matriz de coeficientes que multiplicada por um vetor de incógnitas resulta no vetor de termos independentes. Resolver um sistema linear corresponde a encontrar os valores dos elementos do vetor de incógnitas

para o qual as equações sejam verdadeiras. A forma generalizada de um sistema linear é apresentada na Equação (1).

$$\begin{aligned} E_1 &: a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1, \\ E_2 &: a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2, \\ &\vdots \qquad\qquad\qquad \vdots \\ E_n &: a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n, \end{aligned} \tag{1}$$

Para resolver sistemas lineares, pode-se usar tanto métodos iterativos como métodos diretos [Gilat and Subramaniam 2008]. Métodos diretos são baseados na Eliminação Gaussiana [Erlangga 2005] e calculam a solução exata de um sistema linear, (exceto por erros de arredondamento), com um número finito e bem determinado de operações aritméticas. São adequados para resolver sistemas lineares densos.

Métodos iterativos calculam a solução aproximada de um sistema linear e se beneficiam da esparsidade da matriz, exigindo pouca memória alocada, porém têm baixa velocidade de convergência para encontrar a solução do sistema linear, dependendo do condicionamento da matriz, e em alguns casos podem divergir. Desse modo, a solução por métodos baseados na Eliminação Gaussiana, apesar de demandar maior esforço computacional, é mais adequada nos casos em que as características do problema permitirem sua aplicação [Chapra and Canale 2011]. A paralelização do método de Eliminação Gaussiana permite a diminuição do seu tempo de execução, tornando viável o seu uso na obtenção da solução exata de sistemas de grande porte, como os utilizados no processamento sísmico e outras áreas da computação científica.

A Eliminação Gaussiana consiste na realização de transformações em um sistema linear, de modo a produzir um sistema equivalente, cuja solução possa ser obtida por substituição reversa. Seja um sistema linear da forma $Ax = b$, onde cada elemento $a_{i,j}^k$ pertença a matriz A de coeficientes, e cada elemento b_i^k pertença ao vetor de termos independentes b, as Equações (2) e (3), definem as k operações de eliminação necessárias à obtenção de cada um dos elementos $a_{i,j}^k$ e b_i^k, respectivamente:

$$a_{i,j}^k = a_{i,j}^{k-1} - \frac{a_{i,k-1}^{k-1}}{a_{k-1,k-1}^{k-1}} a_{k-1,j}^{k-1}, k = 2, ...n \tag{2}$$

$$b_i^k = b_i^{k-1} - \frac{a_{i,k-1}^{k-1}}{a_{k-1,k-1}^{k-1}} b_{k-1}^{k-1}, k = 2, ...n \tag{3}$$

3.1. Problema Proposto

O problema proposto para este trabalho tem por objetivo a solução de sistemas lineares densos, que são comuns em problemas na computação científica. Para tanto, foram utilizadas matrizes NxN, onde o valor de N variou nas seguintes ordens: 2000, 4000, 6000, 8000 e 10000. Os programas foram desenvolvidos, inicialmente, de forma sequencial e, em seguida, foram adaptados para as versões paralelas utilizando os modelos OpenACC, CUDA e OpenMP.

Para a execução dos experimentos foi utilizado um equipamento com as seguintes especificações:

- 4 Processadores Intel® Xeon E5-2665 (8 cores cada)
- 24 GB de memória RAM
- 4 Placas Gráficas NVIDIA® Tesla K10
- Sistema Operacional SUSE Linux Enterprise Server 11 SP2 (x86_64)

4. Programação Paralela

O desenvolvimento de modelos de Computação Paralela tem por objetivo oferecer APIs (*Application Programming Interface*) e demais ferramentas computacionais de modo a padronizar e simplificar a utilização dos recursos do hardware para processamento paralelo [Kirk and Hwu 2013].

4.1. OpenMP

Iniciado em 1997, o OpenMP consiste em uma API para processamento paralelo em ambiente de memória compartilhada, portável para diferentes plataformas e disponível para as linguagens C/C++ e *Fortran* [Chandra et al. 2001]. É possível distribuir o fluxo de trabalho em diversas *threads*, por meio da inserção de diretivas, delimitando o trecho de código no qual haja partes paralelizáveis. As diretivas consistem marcações especiais, processadas na etapa inicial da compilação, e que instruem o compilador a implementar um comportamento específico. Na linguagem C, as diretivas OpenMP são especificadas por meio dos mecanismos *pragma*, que fazem parte do padrão de implementação da linguagem. Caso o compilador não possua suporte ao OpenMP, as diretivas serão ignoradas e será produzido um programa com execução sequencial [OpenMP 2015]. Esta abordagem torna simples a adaptação de algoritmos sequenciais legados, sem que seja necessário alterar a estrutura dos mesmos.

4.2. CUDA

Criada pela empresa de placas gráficas NVIDIA, a *Compute Unified Device Architecture* (CUDA) fornece uma plataforma de computação paralela que permite uma programação, em processadores gráficos, menos complexa, por meio de um conjunto de instruções próprias para a arquitetura GPGPU (*General Purpose Graphics Processing Unit*). Dessa forma, os trechos de código que correspondem ao paradigma SPMD (*Single Program Multiple Data*) são implementados em uma função denominada *kernel*. Durante a execução, os dados a serem processados são copiados para a memória da GPU e a operação definida no *kernel* é aplicada em simultâneo a uma quantidade massiva de dados por meio de múltiplas *threads*, o que permite a redução do tempo de processamento. Ao final do processamento, os dados referentes aos resultados são transferidos para a memória principal do sistema *host* [Rauber and Rünger 2013].

Disponibilizada para as linguagens C/C++ e *Fortran*, permite um alto nível de paralelismo, já que foi projetada para obter ganhos de desempenho por meio do aumento da vazão (*throughput*) de processamento [Kirk and Hwu 2013]. No entanto, apresenta, talvez, como única restrição, o fato do conjunto necessário para o desenvolvimento utilizando a plataforma, consistindo em uma placa gráfica NVIDIA compatível, o driver de dispositivo e o conjunto de ferramentas CUDA *Toolkit*, que provê, entre outros itens,

os compiladores e bibliotecas específicos [Sanders and Kandrot 2010], serem fornecidos apenas pela própria NVIDIA.

Em termos lógicos, CUDA apresenta as *threads* agrupadas em *Blocks*. Os *Blocks*, por sua vez, são organizados em *Grids*, desse modo, é necessário, ao executar um *kernel* CUDA, estabelecer quantos *Blocks* de *threads* irão executar a mesma operação, e quantas *threads* comporão cada *Block*. Apenas *threads* de um mesmo *Block* são capazes de trocar informações em memória compartilhada entre si. Os limites de tamanho dos *Blocks* e *Grids* são estabelecidos de acordo com as especificações de hardware de cada placa. conforme ilustrado na Figura 1.

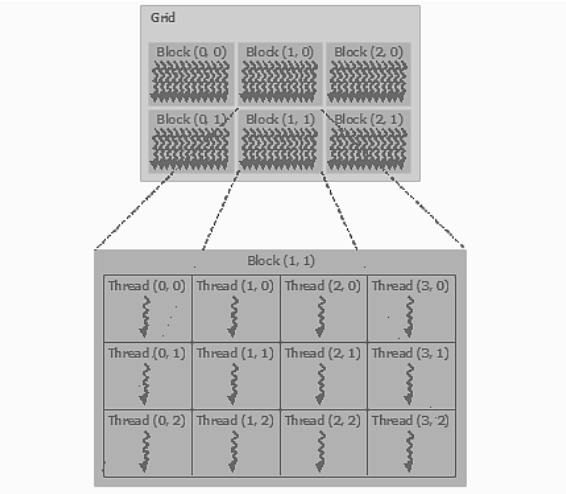


Figura 1. Arquitetura CUDA (Fonte: [NVIDIA 2013])

Fisicamente, cada GPU é composta por um conjunto de *Streaming Multiprocessors* (SM ou SMX). Cada SM dispõe de um espaço de memória de alta velocidade de acesso, que pode ser compartilhada dentro dos *Blocks* da *Grid* entre as *threads* que compõem o respectivo *Block*. Uma representação esquemática da arquitetura de um sistema GPU/CPU é apresentado na Figura 2, na qual são apresentados os SMs, as estruturas de cache, e a possível conexão de múltiplas GPUs em um mesmo *host*, por meio de barramento.

4.3. OpenACC

Fundado por um consórcio entre as empresas NVIDIA, PGI, CRAY e CAPS Enterprise, o OpenACC implementa um conjunto de diretivas de compilação para as linguagens C/C++ e *Fortran* que permite a criação de códigos capazes de tirar proveito de aceleradores (como os Processadores Gráficos de Propósito Geral) por meio da identificação dos laços com diretivas de compilação, semelhantes às utilizadas pelo modelo OpenMP [OpenACC 2014]. Futuramente, as diretivas OpenACC deverão passar a integrar as especificações do modelo OpenMP [OpenACC 2014], permitindo que programadores possam fazer uso dos aceleradores disponíveis com pouco esforço, abstraindo os pormenores da comunicação do processador principal com os mesmos. O OpenACC torna-se adequado a situações nas quais os programadores não estão familiarizados com o modelo de programação CUDA ou estão satisfeitos em abstrair os detalhes da arquitetura-alvo [Cook 2013].

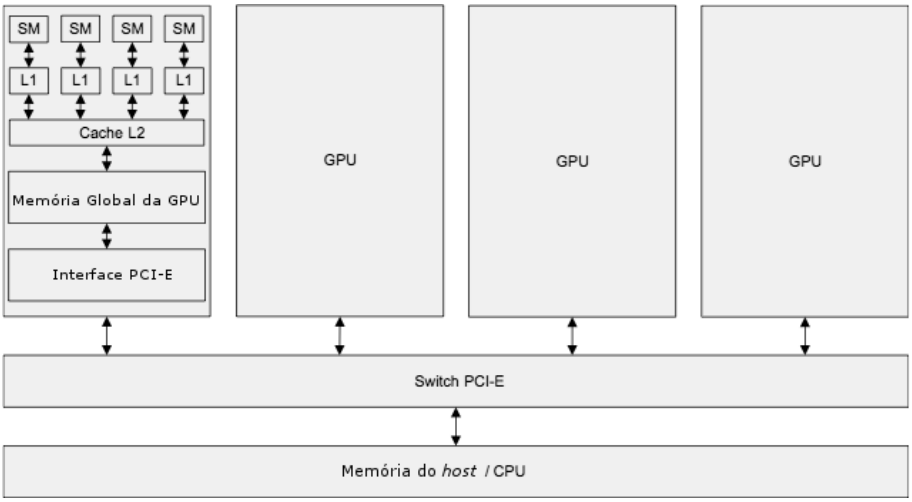


Figura 2. Representação de uma GPGPU (adaptado de [Cook 2013])

5. Proposta de Metodologia de Avaliação

Os testes realizados consistiram na adaptação do trecho de maior intensidade computacional, correspondendo à implementação do método de eliminação gaussiana. Os códigos foram escritos na linguagem C e posteriormente incluídas as diretivas de cada ferramenta, para a execução dos comparativos.

Foi utilizada uma rotina para gerar matrizes de acordo com a ordem fornecida como argumento e os sistemas lineares correspondentes foram solucionados. O programa foi executado de maneira sequencial por dez vezes. O tempo médio de execução foi obtido, servindo como referência para o cálculo de desempenho (*speedup*) das versões paralelas. O trecho principal da versão serial do código é apresentado no Código 1:

```
1 for(i=0;i<(n-1);i++)
2         for(j=(i+1);j<n;j++)
3             ratio = A[j][i] / A[i][i];
4                 for(k=i;k<n;k++)
5                     A[j][k] -= (ratio * A[i][k]);
6                 b[j] -= (ratio * b[i]);
```

Código 1. Trecho do Código da Versão Serial

Na primeira etapa de paralelização do código sequencial foram adicionadas as diretivas OpenMP ao código, sendo este executado utilizando todos os núcleos das CPUs disponíveis, perfazendo um total de trinta e dois núcleos. Este passo foi repetido por dez vezes para a obtenção do tempo médio de execução. No Código 2 é apresentado o trecho principal do código, acrescido das informações pragmáticas do OpenMP.

```
1 #pragma omp parallel shared(A,b,n,i) private(j,k)
2
3 for(i=0;i<(n-1);i++)
4 #pragma omp parallel for
5         for(j=(i+1);j<n;j++)
6         ratio = A[j][i] / A[i][i];
7                 for(k=i;k<n;k++)
8                       A[j][k] -= (ratio * A[i][k]);
```

```
9                   b[j] -= (ratio * b[i]);
```

Código 2. Trecho do Código da Versão em OpenMP

Na segunda etapa de paralelização, foram adicionadas as diretivas OpenACC, compatíveis com o código anteriormente modificado para execução em OpenMP, por meio de variáveis específicas definidas no ambiente de execução, pode-se comprovar a utilização dos processadores gráficos na execução paralela dos laços. O algoritmo foi novamente executado por dez vezes, para obtenção do tempo médio de execução. O trecho principal acrescido das informações pragmáticas do OpenACC é mostrado no Código 3.

```
1  #pragma acc data copy(A[:n][:n],b[0:n])
2
3  for(i=0;i<(n-1);i++)
4  #pragma acc region
5  #pragma acc loop independent
6          for(j=(i+1);j<n;j++)
7                  ratio = A[j][i] / A[i][i];
8  #pragma acc loop independent
9                  for(k=i;k<n;k++)
10                        A[j][k] -= (ratio * A[i][k]);
11                 b[j] -= (ratio * b[i]);
```

Código 3. Trecho do Código na Versão em OpenACC

Na terceira etapa, o código foi parcialmente reescrito, para tornar-se compatível com o CUDA C, com a inserção de um *kernel* e de funções responsáveis pela troca de dados com os processadores gráficos. O código em CUDA C foi otimizado, por meio do uso de estruturas de memória compartilhada de acesso rápido (*Shared Memory*). A utilização da *Shared Memory* é possível entre *threads* de um mesmo bloco. Os valores com maior quantidade de acessos, correspondentes à linha de referência para cada iteração, foram carregados para a *Shared Memory*, aumentando a velocidade de acesso. O código reescrito foi então compilado com o compilador C da NVIDIA, disponível no CUDA *Toolkit*, e executado por dez vezes para a obtenção do tempo médio de execução. O trecho principal do código, referente ao *kernel* CUDA produzido, é apresentado no Código 4.

```
1  __global__ void factor(double *a_d, int size, long i)
2  {
3   long j = blockIdx.y*blockDim.y+threadIdx.y;
4   long k = blockIdx.x*blockDim.x+threadIdx.x;
5
6   __shared__ double lpivo[BLOCKSIZE+1];
7
8   if(threadIdx.y==0)
9    lpivo[threadIdx.x] = a_d[i*(size+1)+k];
10  __syncthreads();
11
12  double pivo = a_d[i*(size+1)+i];
13  if(j<(size) && k<(size+1)){
14   const double ratio = a_d[j*(size+1)+i]/pivo;
15   if(j>=(i+1) && k>=(i)){
16    a_d[j*(size+1)+k] = a_d[j*(size+1)+k] - (ratio*lpivo[threadIdx.x]);
17   }
18  }
```

```
19  }
```

Código 4. Trecho do *Kernel* em CUDA

Outro fator analisado, neste trabalho, foi a quantidade de linhas necessárias para a adaptação do código-fonte original. Tais quantidades de linhas inseridas são apresentadas na Tabela 1. A implementação em CUDA demandou um acréscimo maior na quantidade de linhas pois requer que as funções para alocação de dados na memória da GPU, bem como para transferências de dados entre as memórias sejam explicitadas no código. Além disso, é necessária a definição de uma função separada (kernel) a ser invocada para execução na GPU. Tais estruturas não se fizeram necessárias nas versões baseadas em diretivas, ou puderam ser inseridas de forma mais concisa.

Tabela 1. Quantidades de linhas de código acrescentadas ao algoritmo de Eliminação Gaussiana

Implementação	Número de linhas adicionadas
OpenMP	4
OpenACC	6
CUDA	28

6. Resultados Experimentais

Os tempos de processamento coletados nos testes, utilizando o método de Eliminação Gaussiana, nas versões OpenMP, OpenACC e CUDA, para as matrizes de cada uma das ordens fornecidas, são expressos na Tabela 2 e no gráfico da Figura 3.

Tabela 2. Tempo médio em segundos (M) e desvio padrão (DP) dos experimentos

	2000		4000		6000		8000		10000	
	M	**DP**	**M**	**DP**	**M**	**DP**	**M**	**DP**	**M**	**DP**
Serial	13,65	0,02	110,34	0,28	376,28	5,24	889,69	11,84	1737,37	22,70
OpenMP	2,00	0,01	13,71	0,03	44,06	0,07	101,66	0,11	208,22	0,74
OpenACC	1,60	0,01	6,12	0,03	17,90	0,11	41,21	0,08	81,28	0,09
CUDA	1,12	0,03	3,65	0,03	9,79	0,04	20,93	0,04	39,04	0,08

Foram obtidos valores de *speedup* superiores a 44 em relação à versão sequencial para as matrizes de maior ordem. As médias dos valores de *speedup* obtidos para diferentes ordens de matrizes são apresentadas pelo gráfico da Figura 4.

Finalmente, foram observadas as execuções dos programas baseados em OpenACC e CUDA por meio da ferramenta NVIDIA Visual *Profiler*. A ferramenta, que integra o CUDA *Toolkit*, permite analisar o desempenho de aplicações CUDA por meio do fornecimento de informações sobre a execução das mesmas, tais como alocação de memória e chamadas a *kernels*, apresentando-as graficamente. Os gráficos de nível de utilização das versões OpenACC e CUDA, para uma matriz de ordem 10000, são apresentados no NVIDIA Visual *Profiler* pela Figura 5.

A análise das informações de execução fornecidas pelo *Profiler* indica que a versão OpenACC, apesar de obter um melhor aproveitamento da transferência de memória,

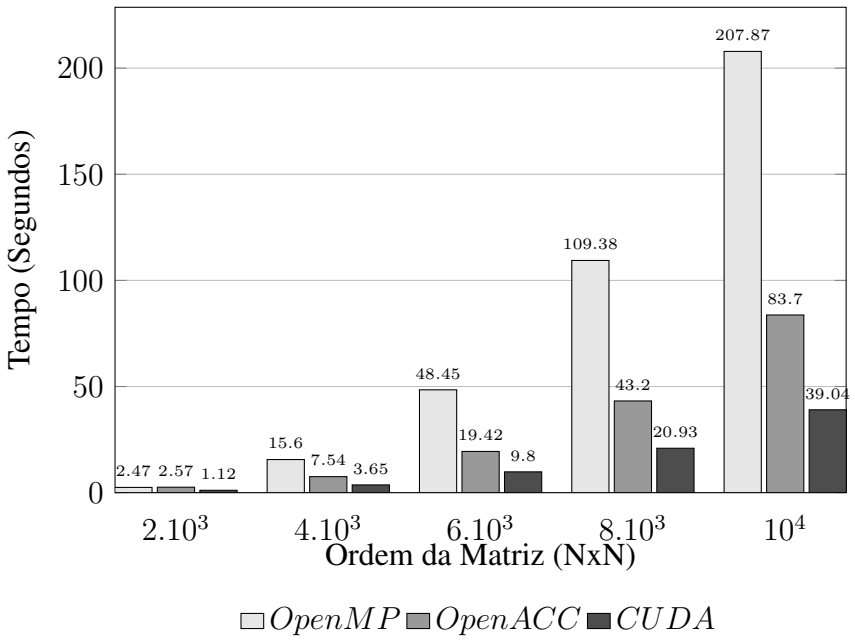


Figura 3. Tempos de Processamento para o Método de Eliminação Gaussiana

Tabela 3. Informações de desempenho para as versões CUDA e OpenACC

	OpenACC	**CUDA**
Tamanho do Bloco (x,y,z)	128,1,1	16,16,1
Tempo total de execução do kernel	80s	46s
Ocupância	74%	90%
Quantidade de Shared Memory (por bloco)	8B	4KB
Vazão de armazenamento em memória global	38GB/s	7,6GB/s
Vazão de carregamento em memória global	84,6GB/s	22,8GB/s
Quantidade de acessos à memória compartilhada	36980	421675250
Vazão de armazenamento em memória compartilhada	117GB/s	60,8GB/s
Vazão de carregamento em memória compartilhada	468GB/s	548GB/s

produziu um executável com menor nível de aproveitamento do paralelismo. A menor quantidade de memória compartilhada utilizada revela que a versão OpenACC teve uma perda de desempenho relacionada ao tempo de acesso à memória, uma vez que o tempo de acesso à memória global é mais elevado.

A ferramenta de *profile* da NVIDIA fornece ainda outros dados relevantes à analise do desempenho, tais como tamanhos de *Grid* e de *Blocks* utilizados, ocupância da GPU (nível de aproveitamento do paralelismo), quantidade de memória compartilhada utilizada, e vazão média de dados. Tais valores são apresentados na Tabela 3, relativa a execução das versões CUDA e OpenACC.

7. Conclusões e Trabalhos Futuros

A partir dos resultados obtidos percebe-se que a implementação em OpenACC obteve entre 46% e 50% do desempenho da versão CUDA otimizada, sendo entre 2 e 2,5 mais

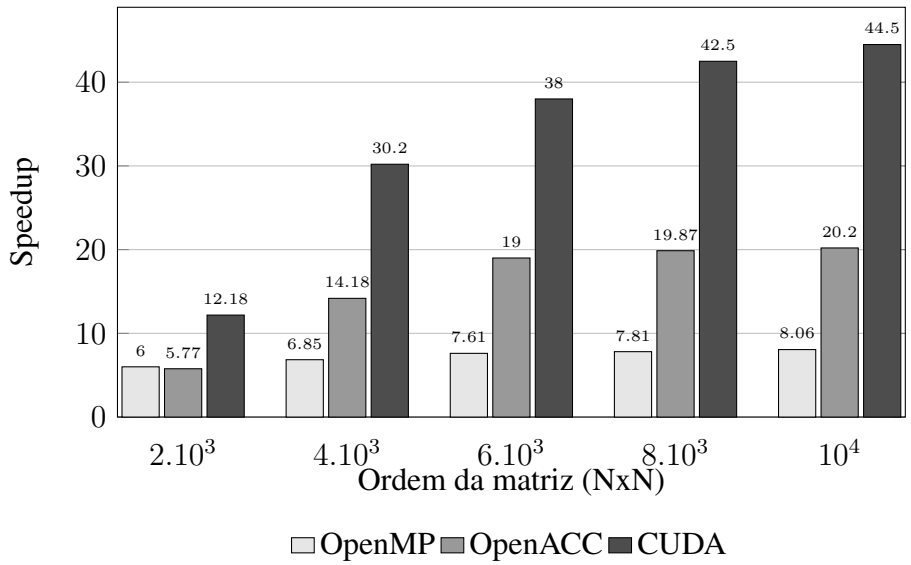
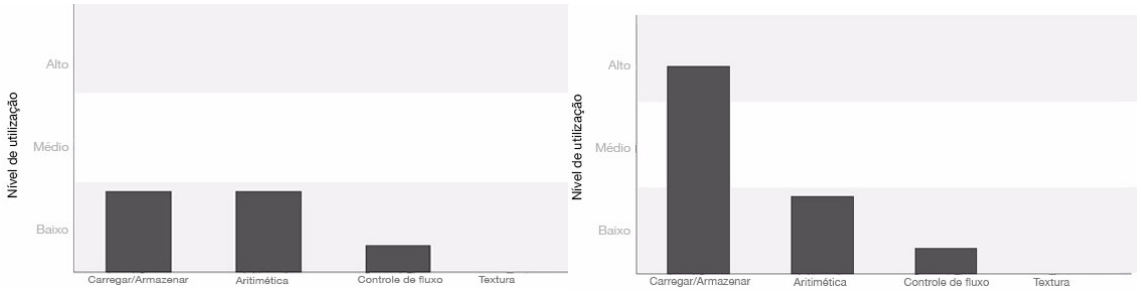


Figura 4. Valores de *speedup* do algoritmo de Eliminação Gaussiana



(a) OpenACC (b) CUDA

Figura 5. Gráficos de nível de utilização produzidos pelo NVIDIA Visual Profiler (Adaptados)

rápida que a implementação em OpenMP, sem a necessidade de grandes modificações estruturais, mostrando-se, portanto, uma alternativa adequada à paralelização de aplicações legadas baseadas em OpenMP, oferecendo ganhos expressivos de desempenho e permitindo que o desenvolvimento de melhorias se concentre no aprimoramento da estrutura básica do algoritmo e não em estruturas de controle do paralelismo, que é abstraído por meio das diretivas.

Entretanto, por ainda se tratar de um modelo em desenvolvimento, o OpenACC apresenta algumas limitações. Durante os testes executados para este trabalho houveram problemas com a utilização de valores complexos e com o uso de bibliotecas para manipulação de matrizes esparsas.

Como trabalho futuro, pretende-se aprimorar os métodos implementados, proporcionando a redução do tempo de execução dos mesmos. Como alternativa para a melhoria dos resultados com o CUDA, pretende-se aplicar otimizações, por meio de outras estratégias de paralelização ou através do uso de bibliotecas específicas para álgebra linear, como por exemplo a CuBLAS [NVIDIA 2013]. Além disso, serão testadas técnicas que possibilitem o processamento de problemas maiores que a capacidade de memória de uma

única placa gráfica, utilizando diferentes modelos de computação distribuída.

Agradecimentos

Os autores agradecem ao Instituto Nacional de Ciência e Tecnologia - Geofísica do Petróleo (INCT-GP) e a Agência de pesquisa brasileira FINEP pelo apoio ao desenvolvimento deste trabalho.

Referências

Abdelkhalek, R., Calandra, H., Coulaud, O., Roman, J., and Latu, G. (2009). Fast seismic modeling and reverse time migration on a gpu cluster. In Smari, W. W. and McIntire, J. P., editors, *HPCS*, pages 36–43. IEEE.

Barbosa, J. (2011). *Noções sobre Matrizes e Sistemas de Equações Lineares*. FEUP Edições, 2a edição edition.

Bényász, G. and Cser, L. (2010). Clustering financial time series on cuda. In *Conference of PHD Students in Computer Science Institute of Informatics of the University of Szeged*, Hungary. University of Szeged.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R. (2001). *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Chapra, S. and Canale, R. (2011). *Métodos Numéricos para Engenharia*. McGraw Hill Brasil.

Cook, S. (2013). *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Applications of GPU Computing Series. Morgan Kaufmann.

Erlangga, Y. A. (2005). *A robust and efficient iterative method for the numerical solution of the Helmholtz equation*. fl.126, Delft University. Tese (Doutorado).

Fernandes, A. d. A., Stephany, S., and Panetta, J. (2012). Paralelização do modelo de cinética química atmosférica do cptec/inpe para utilizar placas gráficas. In *XII Workshop de Computação Aplicada*.

Gilat, A. and Subramaniam, V. (2008). *Metodos Numéricos para Engenheiros e Cientistas: Uma Introdução com Aplicações Usando o MATLAB*. Bookman.

Hoshino, T., Maruyama, N., Matsuoka, S., and Takaki, R. (2013). Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 136–143.

Kirk, D. B. and Hwu, W.-m. W. (2013). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2st edition.

Ledur, C. L., Zeve, C. M., and dos Anjos, J. C. (2013). Comparative analysis of openacc, openmp and cuda using sequential and parallel algorithms.

Manavski, S. and Valle, G. (2008). Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinformatics*, 9:S10.

Manfroi, L. L. F., Schulze, B., Pinto, R. C. G., Mury, A. R., and Ferro, M. (2014). Avaliação de arquiteturas manycore e do uso da virtualização de gpus. In de Computação (SBC), S. B., editor, *XXXIV Congresso da Sociedade Brasileira de Computação*, pages 1837–1850.

NVIDIA (2013). Cuda toolkit documentation. http://docs.nvidia.com/cuda/.

OpenACC (2014). Openacc. http://www.openacc-standard.org/.

OpenMP (2015). Openmp. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

Pan, L., Gu, L., and Xu, J. (2008). Implementation of medical image segmentation in cuda. In *Information Technology and Applications in Biomedicine, 2008. ITAB 2008. International Conference on*, pages 82–85.

Pilla, L. L. and Navaux, P. O. A. (2010). Uso da classificação dwarf mine para a avaliação comparativa entre a arquitetura cuda e multicores. In de Computação (SBC), S. B., editor, *XXX Congresso da Sociedade Brasileira de Computação*.

Rauber, T. and Rünger, G. (2013). *Parallel Programming: For Multicore and Cluster Systems*. Springer Berlin Heidelberg.

Reyes, R., López-Rodríguez, I., Fumero, J. J., and de Sande, F. (2012). accull: An openacc implementation with cuda and opencl support. In Kaklamanis, C., Papatheodorou, T. S., and Spirakis, P. G., editors, *Euro-Par*, volume 7484 of *Lecture Notes in Computer Science*, pages 871–882. Springer.

Sanders, J. and Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition.

Santos, A. F., Júnior, P. G. L., Fernandes, S. F. L., and Sadok, D. F. H. (2013). Explorando o processamento paralelo na classificação de tráfego em redes de alta velocidade. In de Computação (SBC), S. B., editor, *XXXI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 47–60.

Souza, D. L., Monteiro, G. D., Martins, T. C., Dmitriev, V. A., and Teixeira, O. N. (2011). Pso-gpu: accelerating particle swarm optimization in cuda-based graphics processing units. In Krasnogor, N. and Lanzi, P. L., editors, *GECCO (Companion)*, pages 837–838. ACM.

Stringhini, D., Gonçalves, R. A., and Goldman, A. (2012). Introdução à computação heterogênea. In *XXXI Jornadas de Atualização em Informática*, pages 262–309. Sociedade Brasileira de Computação (SBC).

Top500 (2015). Top 500 supercomputer sites. http://www.top500.org/.