Portabilidade e Eficiência do Método Fletcher de Aplicações Sísmicas em Arquiteturas Multicore e GPU*

Matheus S. Serpa[1], Pablo J. Pavan[1], Jairo Panetta[2], Antônio Azambuja[3], Alexandre Carissimi[1], Philippe O. A. Navaux[1]

[1] Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970, Porto Alegre – RS – Brasil

{msserpa, pjpavan, asc, navaux}@inf.ufrgs.br

[2]Divisão de Ciência da Computação – Instituto Tecnológico de Aeronáutica (ITA)
São José dos Campos – SP – Brasil

jairo.panetta@gmail.com

[3]Petróleo Brasileiro S.A
Rio de Janeiro – RJ – Brasil

antonio.azambuja@petrobras.com.br

Abstract. *The simulation of acoustic wave propagation is the backbone of seismic imaging tools used by the oil and gas industries. To perform such simulations, HPC architectures are employed, generating faster and more accurate results with each processor generation. However, to achieve high performance in these architectures, several challenges must be taken into account when developing the application. In this article, we optimized the Fletcher Method for multicore and GPU and evaluated the performance, energy consumption, and energy efficiency of eight versions of the code. The results show that the CUDA version has the best performance and energy efficiency; however, the OpenACC version that has the advantage of portability has a performance and energy efficiency degradation of only 10% and 8% compared with CUDA.*

Resumo. *A simulação da propagação de ondas acústicas é a base das ferramentas de imagem sísmica utilizadas pela indústria de petróleo e gás. Para realizar tais simulações, arquiteturas de CAD são empregadas, fornecendo resultados mais rápidos e com maior precisão a cada geração de processadores. Entretanto, para atingir alto desempenho nessas arquiteturas, vários desafios devem ser levados em consideração no momento do desenvolvimento da aplicação. Neste artigo, a Modelagem Fletcher foi otimizada para multicore e GPU e o desempenho, o consumo de energia e a eficiência energética de oito versões do código foram avaliados. Os resultados mostram que a versão CUDA tem o melhor desempenho e eficiência energética; no entanto, a versão OpenACC que tem a vantagem da portabilidade, tem um desempenho e degradação de eficiência energética de apenas 10 e 8% comparado com CUDA.*

*Este trabalho foi parcialmente financiado pelo projeto Petrobras 2016/00133-9 e pelo projeto "GREEN-CLOUD: Computação em Cloud com Computação Sustentável" (#16/2551-0000 488-9), da FAPERGS e do CNPq, programa PRONEX 12/2014.

1. Introdução

A geofísica de exploração vem sendo fundamental na procura de recursos energéticos, como Gás e Petróleo. Entretanto, altos custos de perfuração, com menos de 50% de precisão por broca, limitam seu uso [Qutob et al. 2004, Lukawski et al. 2014]. Assim, a indústria de Petróleo e Gás conta com *software* focados em Computação de Alto Desempenho (CAD) como uma forma economicamente viável de reduzir custos. A base de muitos mecanismos de *software* para geofísica de exploração é a simulação de propagação de ondas. Por exemplo, em ferramentas de imagem sísmica, modelagem, migração e inversão, essa simulação pode ser utilizada. Essas ferramentas são construídas tendo como base solucionadores de Equações Diferenciais Parciais (EDP), onde a EDP resolvida em cada caso define a precisão da aproximação à física real.

A aproximação do modelo de propagação de ondas acústicas é a base atual das ferramentas de imagem sísmica. Ela tem sido extensivamente aplicada nos últimos anos em reservatórios de óleo e gás com potencial de geração de imagens sob domos salinos. Essas aplicações devem ser continuamente portadas para o mais novo *hardware* de CAD disponível para manter a competitividade no mercado. Ao mesmo tempo, as arquiteturas de CAD evoluíram e a melhoria de desempenho apenas com o aumento da frequência de *clock* não existe mais. As soluções atualmente adotadas estão sendo substituídas por tecnologias *multicore* e *manycore* [Clapp et al. 2010, Clapp 2015].

A última década viu uma tendência de construção de sistemas de alto desempenho com dispositivos dedicados e aceleradores, que produzem um bom retorno em relação eficiência energética *(FLOPs/Watt)* [J. Dongarra and Strohmaier 2019]. Entre as alternativas disponíveis, os fabricantes de aceleradores têm dedicado esforços para fornecer de dezenas a milhares de unidades de processamento trabalhando em baixas frequências, tais como as *Graphics Processing Units (GPUs)*, e outros aceleradores [Witten et al. 2016]. Os processadores *multicore* ainda mais tradicionais, como a família *Intel Xeon*, incluem dezenas de núcleos nos processadores, trabalhando em altas frequências.

Vários desafios devem ser levados em consideração buscando atingir alto desempenho nessas arquiteturas. Um dos aspectos mais importantes é o comportamento do subsistema de memória, já que o acesso à memória possui um papel fundamental no desempenho [Serpa et al. 2019b]. O consumo de energia e a eficiência energética também são pontos a serem considerados [Subramaniam et al. 2013]. Ainda, a computação heterogênea tem ganho força, aumentando a complexidade do uso das arquiteturas. Esse aumento de complexidade ocorre devido a essas arquiteturas possuírem memórias e núcleos próprios que possuem latências diferentes. Além disso, dados devem ser copiados pelo programador de uma memória a outra, aumentando a dificuldade de programação. A partir disso, surgiram vários modelos de programação mais genéricos e de alto nível, os quais permitem a escrita de códigos portáteis, que podem ser executados em arquiteturas de *CPU* e *GPU* sem nenhuma alteração [Sabne et al. 2014].

Neste artigo, trabalhamos com a Modelagem *Fletcher*, uma aplicação de propagação de ondas utilizada por empresas de Petróleo. Nossos objetivos são:

- Explorar o paralelismo intrínseco da Modelagem *Fletcher* para utilizar eficientemente as arquiteturas *multicore* e *GPU*;
- Avaliar o custo das versões portáveis sob o ponto de vista de desempenho, consumo de energia e eficiência energética.

O artigo está organizado da seguinte forma. A Seção 2 discute os trabalhos relacionados. A Seção 3 apresenta a Modelagem Fletcher e detalhes de sua implementação. A Seção 4 descreve as arquiteturas, as métricas, e informações detalhadas sobre os experimentos. A Seção 5 fornece uma avaliação de desempenho, consumo de energia e eficiência energética das diferentes versões da aplicação. Finalmente, a Seção 6 apresenta conclusões e trabalhos futuros.

2. Trabalhos Relacionados

Arquiteturas recentes, incluindo *GPUs* e aceleradores, provaram ser adequadas para geofísica, magneto-hidrodinâmica e simulações de fluxo, superando os processadores de uso geral em eficiência [Yuen et al. 2013]. Para obter o máximo desempenho nesses novos dispositivos, é necessária alguma reengenharia das regiões do código, se não toda a aplicação. Assim, [Kukreja et al. 2016] gera automaticamente um código para simulação geofísica altamente otimizado para múltiplas arquiteturas, enquanto [Niu et al. 2014] sugere o uso de reconfiguração de sistemas em tempo de execução e um modelo de desempenho, para reduzir o consumo de recursos computacionais.

[Caballero et al. 2015] estudou o efeito de diferentes otimizações em nível de memória e computação em equações de propagação de ondas elásticas. Os resultados mostraram que comparando o processador e o coprocessador *Xeon Phi*, o processador foi até quatro vezes maior do que o coprocessador.

[Rubio et al. 2013] reescreveu um propagador de ondas elásticas genérico em processadores de uso geral, *GPUs* e *Xeon Phi*, mostrando que o coprocessador fornece bom desempenho a um custo de desenvolvimento reduzido. Nossas otimizações visam apenas um tipo de domínio para reduzir a complexidade do problema e para restringir o número de variáveis que estão sendo reproduzidas na análise.

Em [Andreolli et al. 2015], os autores focaram em equações de propagação de ondas acústicas, escolhendo as técnicas de otimização a partir do ajuste sistemático do algoritmo. O uso de *thread blocking*, *cache blocking*, reutilização de registradores, vetorização e redistribuição de dados do laço resultou em melhorias significativas no desempenho. Nossa proposta, busca analisar e otimizar uma simulação de imagem sísmica amplamente utilizada por empresas de Petróleo como a Petrobras.

Esforços de pesquisa como o apresentado em [Castro et al. 2016] melhoraram e avaliaram o desempenho da equação de propagação de onda acústica no coprocessador Intel Xeon Phi e compararam com o processador *manycore MPPA-256*, com processadores de uso geral e com uma GPU. As otimizações incluem *cache blocking*, alinhamento de memória com deslocamento de ponteiro e afinidade de *thread*. Elas mostram que os melhores resultados são obtidos a partir de uma combinação dos dois primeiros e que o desempenho com o *Xeon Phi* está próximo da *GPU*. Nosso trabalho vai um passo adiante ao estudar o compromisso entre a portabilidade, o desempenho e a eficiência energética de uma aplicação de geofísica real.

[Pavan et al. 2018] analisa e propõe otimizações para as operações de entrada e saída (E/S) de uma aplicação de geofísica que utiliza o algoritmo *Reverse Time Migration (RTM)*. Os resultados mostram que a utilização de *checkpoints* e o aumento do tamanho da requisição reduz o tempo de execução da aplicação em até $17,33\%$. Uma vez que

identificamos que a E/S não é o principal gargalo da nossa aplicação, focamos na análise de outros fatores ligados com o desempenho, consumo e portabilidade.

[Zhebel et al. 2013] comparou a escalabilidade de códigos não otimizados para diferenças finitas e algoritmos de elementos finitos no *Intel Xeon* e *Xeon Phi*. No *Xeon*, a escalabilidade foi semelhante e não-linear para todos os métodos, enquanto no *Xeon Phi*, apenas a diferença finita apresentou menor escalabilidade, devido a ociosidade da *thread* de controle de E/S. Nossa proposta vai além de uma análise de escalabilidade e busca uma maior compreensão do comportamento de uma aplicação real em arquiteturas *multicore* e GPU.

[Carrijo Nasciutti et al. 2018] fez uma análise de desempenho de uma aplicação sintética baseada em matrizes 3D em *GPUs* com foco no uso adequado da hierarquia de memória. Eles concluem que o código de maior desempenho é o que combina a reutilização de *cache* somente leitura, inserindo o laço da dimensão Z no *kernel* e reutilizando registradores do processador. Diferente desse trabalho, avaliamos o desempenho e a eficiência energética de uma aplicação real.

[Serpa et al. 2019a] propõe várias estratégias de otimização para um modelo de propagação de ondas para seis arquiteturas de *CPU* e *GPU*. O trabalho foca em melhorar o uso da memória *cache*, vetorização, balanceamento de carga e localidade na hierarquia de memória. Entretanto, o trabalho não leva em consideração o consumo de energia e a eficiência energética.

3. Modelagem Fletcher

A aplicação Modelagem *Fletcher* simula a propagação de ondas através do tempo. A propagação de ondas é definida pela equação acústica (Equação 1), sendo que, camadas geológicas distintas possuem velocidades distintas (Equação 2), onde $p(x, y, z, t)$ é a pressão em cada ponto do domínio ao longo do tempo, $V(x, y, z)$ é a velocidade de propagação e $\rho(x, y, z)$ é a densidade. A dedução das equações diferenciais parciais, tal como, condições de fronteira e estabilidade, pode ser encontrada em [Fletcher et al. 2009].

$$\frac{1}{V^2}.\frac{\partial^2 p}{\partial t^2} = \nabla^2 p \tag{1}$$

$$\frac{1}{V^2}.\frac{\partial^2 p}{\partial t^2} = \nabla^2 p - \frac{\nabla \rho}{\rho}.\nabla p \tag{2}$$

A modelagem simula a coleta de dados em um levantamento sísmico, como ilustrado na Figura 1. De tempos em tempos, equipamentos acoplados a um navio emitem ondas que refletem e refratam em mudanças de meio no subsolo. Eventualmente, essas ondas voltam à superfície do mar, sendo coletadas por microfones específicos acoplados a cabos rebocados pelo navio. O conjunto de sinais recebidos em cada fone ao longo do tempo constitui um traço sísmico. Para cada emissão de ondas, grava-se os traços sísmicos de todos os fones do cabo. O navio se desloca e emite sinais ao longo do tempo.

No decorrer do projeto Petrobras 2016/00133-9, o qual tem como objetivo analisar o emprego de arquiteturas atuais para executar aplicações sísmicas típicas da

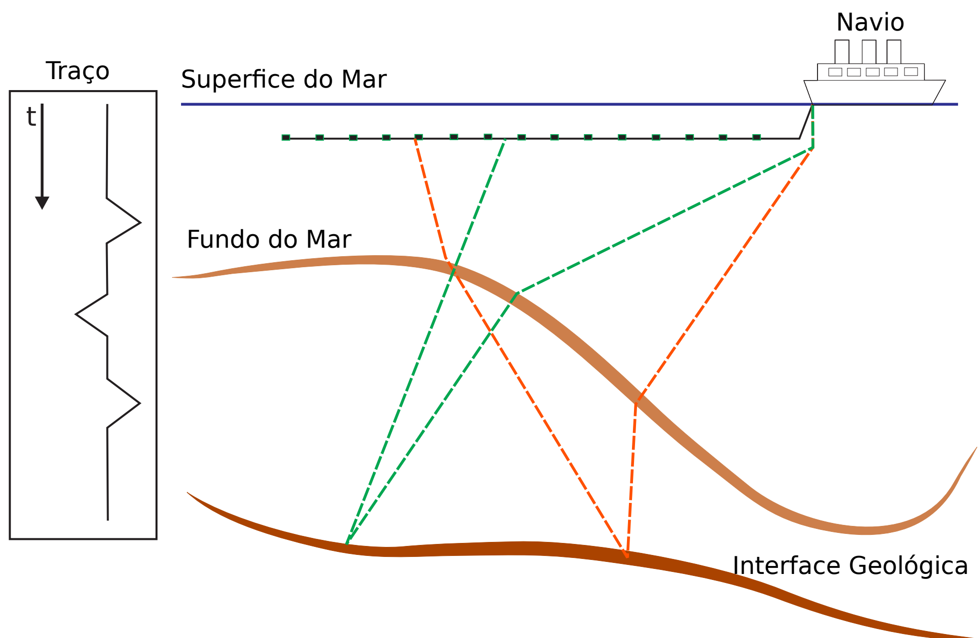

Figura 1. Coleta de dados em levantamento sísmico marítimo.

indústria de gás e petróleo, um programa de modelagem da propagação de ondas em meio isotrópico foi escrito. O código foi escrito em linguagem *C* e a discretização foi feita utilizando diferenças finitas. Foram feitas várias versões do código como uma versão *OpenMP* [Chandra et al. 2001] para explorar apenas processadores *multi-core*, uma versão *CUDA* [Sanders and Kandrot 2010] para *GPUs* e uma única versão *OpenACC* [Wienke et al. 2012] para ambas arquiteturas.

3.1. Versão Original

Existe um laço de repetição que representa o número de iterações da aplicação. Em cada iteração, primeiro as fontes das ondas são inseridas. Após, a propagação é feita através do cálculo de um *stencil*, que é comumente usado nesse tipo de aplicações científicas para resolver equações diferenciais parciais sobre grades multidimensionais. Esses cálculos são independentes porque cada ponto é uma combinação dos valores de um ponto e seus vizinhos na iteração anterior, como mostra na Figura 2. Para a computação de cada um desses pontos vários outros pontos precisam ser lidos das memórias *cache* e da memória principal. Isso gera mais acessos a memória do que computação por ponto. Nesse sentido, o comportamento de acesso à memória apresenta desafios para otimizar o desempenho.

O trecho de código avança a propagação da onda nos pontos internos de uma de grade (cubo). Os campos tridimensionais são mapeados em vetores unidimensionais na ordem (*x*, y, z), ou seja, *x* é uma direção que muda mais rapidamente no armazenamento dos vetores em memória. A direção da derivada é determinada pelo salto na memória (nas direções x, y e z). O paralelismo da versão *original* é imediato, pois, a propagação em um ponto da grade é independente da propagação em qualquer outro ponto da grade. Logo, os laços aninhados que percorrem a grade são totalmente paralelizáveis.

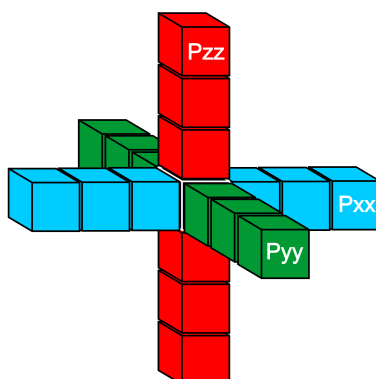

Figura 2. Estêncil 3D de 7 pontos.

3.2. Versão Otimizada

Foi identificado que o cálculo das derivadas cruzadas demandava 76% das operações de ponto flutuante na versão original. Essa constatação foi obtida usando a ferramenta de *profiler PAPI* [Terpstra et al. 2010] na execução sequencial e na execução com *OpenMP* da versão original de uma grade de 216 *x* 216 *x* 216 pontos. Constatação similar foi obtida com a ferramenta de *profiler NVProf* [Nvidia 2016] na *GPU* na mesma grade.

Nesse sentido, buscamos reduzir a quantidade de operações para calcular as derivadas cruzadas. A alteração realizada consiste em calcular a derivada cruzada em xy como primeira derivada em y da primeira derivada em x. Isso não reduz a quantidade de operações no cálculo da derivada cruzada em um ponto, mas reduz a quantidade de operações no cálculo da derivada cruzada em y consecutivos, por reutilizar sete derivadas previamente calculadas em x dentre as oito necessárias. Como a mesma redução ocorre nas demais derivadas cruzadas, isso foi aplicado para o cálculo de todas as derivadas cruzadas.

Inicialmente, foram calculadas e armazenadas as derivadas primeiras em x e em y de todos os pontos da grade. A primeira derivada em x foi usada para calcular as derivadas cruzadas em xy e em xz, e, a primeira derivada em y para calcular a derivada cruzada em yz. Como as derivadas primeiras são utilizadas apenas para calcular as derivadas cruzadas, os valores dessas derivadas em determinada profundidade z são necessários apenas em algumas profundidades acima e abaixo. Logo, o campo completo de cada derivada primeira que está armazenado em uma matriz pode ser substituído por um *buffer* circular, que armazena os valores dessa derivada em poucas profundidades, reduzindo o aumento do consumo de memória. Uma vez que o *buffer* circular é muito menor que a matriz que possui o campo completo, é possível melhorar o uso da hierarquia de memória, acelerando a computação. Nesse caso, existe uma região paralela contendo a inicialização, seguida pelo laço sequencial em profundidades, contendo duas regiões paralelas, a primeira para calcular a derivada restante e a segunda para aplicar a propagação.

4. Metodologia de Avaliação

As codificações *OpenMP*, *OpenACC* e *CUDA* foram executadas em 12 grades (cubos) de tamanhos crescentes, escolhidas de forma que o tamanho da grade em qualquer direção seja múltiplo de 32. O valor 32 foi escolhido pois é o tamanho dos *warps* de *CUDA*. Uma vez que os resultados foram semelhantes entre si, mostramos nesse trabalho os

resultados para um cubo de $504 \times 504 \times 504$, que é o maior tamanho que cabe na memória principal da GPU. Cada experimento foi executado 30 vezes com o número de *cores* virtuais de cada arquitetura. Os gráficos obtidos mostram os valores médios de tempo de execução e os intervalos de confiança de 95% segundo a distribuição *t de Student* [Ott and Longnecker 2015].

Os experimentos foram realizados nos ambientes *Broadwell* e *Pascal*. A *Broadwell* possui dois processadores *Intel Xeon E5-2699 v4* de 22 núcleos. Cada núcleo suporta um *2-way Simultaneous Multithreading (SMT)*, permitindo a execução de até 88 *threads*. Os núcleos possuem *caches* privadas L1 e L2, enquanto a *cache* L3 é compartilhada entre todos os núcleos de cada processador. A *Pascal* é uma *GPU NVIDIA P100* com 3584 *CUDA Cores*. A Tabela 1, possui informações detalhadas de cada ambiente.

Os resultados apresentam o desempenho em *samples* por segundo, o consumo de energia em *Joules* e a eficiência energética em *Joules* por *sample*. O desempenho é dado pela divisão do número de pontos da grade calculados pelo tempo de execução da aplicação. O consumo de energia é calculado pela integral da potência consumida pela placa mãe ao longo da execução. Essa potência é obtida via *Intelligent Platform Management Interface (IPMI)* [Slaight 2002] que mede a potência consumida pela placa mãe e a GPU. Por fim, a eficiência energética é obtida dividindo a energia pela quantidade de *samples*, ou seja, a quantidade de energia consumida para calcular um *sample* por segundo.

Tabela 1. Configuração das arquiteturas avaliados.

Nome	Parâmetro	Valor
Broadwell	Arquitetura	Broadwell-EP
	Processor	2 × Intel Xeon E5-2699 v4
		2 × 22 *cores*, 2-SMT
	Memória	22 × 32KB L1, 22 × 256KB L2
		55MB L3, 256GB DDR4-2400
Pascal	Arquitetura	Pascal GP100
	GPU	NVIDIA Tesla P100-SMX2
		3584 CUDA cores
	Registradores	56 × 256KB
	Memória	56 × 64KB *shared*, 4096KB L2
		56 × 24KB L1 / *texture (read-only)*
		16GB GDDR5

5. Resultados Experimentais

Inicialmente, avaliamos o desempenho das duas implementações da aplicação e, discutimos o compromisso entre o desempenho e portabilidade das diferentes versões da aplicação sobre diversas interfaces de programação paralela (IPPs). Após, avaliamos o consumo energético e a eficiência energética.

5.1. Avaliação de Desempenho e Portabilidade

Foi feita a avaliação de desempenho para determinar o custo da portabilidade entre *multicore* e *GPU*, contrastando o desempenho da codificação portátil (*OpenACC*) nas

duas arquiteturas com o das codificações não portáteis, específicas, para cada arquitetura (*OpenMP* no *multicore* e *CUDA* na *GPU*). A portabilidade de *OpenACC* é de tal ordem que uma única compilação gera código para tanto para a *multicore* como a *GPU*. Uma variável de ambiente define, durante a execução, a arquitetura a utilizar.

A Figura 3 apresenta o desempenho em milhões de *samples* por segundo, para as codificações *OpenMP* e *OpenACC* executando em *CPU*, *CUDA* e *OpenACC* executando em *GPU*. Na versão *original*, *OpenACC* é 12,9% mais rápido que *OpenMP* e 57.5% mais lento que *CUDA*. Na versão *otimizada*, *OpenACC* é 36% mais rápido que *OpenMP* e 10.5% mais lento que *CUDA*. Isso mostra que a nossa versão portátil (OpenACC *multicore/GPU*) é mais rápida do que *OpenMP* no *multicore*, mas na *GPU* perde no desempenho em relação ao *CUDA*.

Também analisamos o ganho da versão *otimizada* em relação a *original*. O desempenho da versão *otimizada* utilizando *OpenMP* foi 48,3% melhor que a versão *original OpenMP*. No caso da versão *CUDA*, o ganho foi de 73.6%. A versão *OpenACC* foi a que melhor se beneficiou dessa otimização. A melhora de desempenho foi de 78,7% na versão *OpenACC-CPU* e de 265,9% na versão *OpenACC-GPU*, se aproximando do desempenho da versão *CUDA*. Além disso, os resultados mostram que a substituição da derivada cruzada pela derivada primeira da derivada primeira é mais proveitosa para *GPU* do que para o *multicore*.

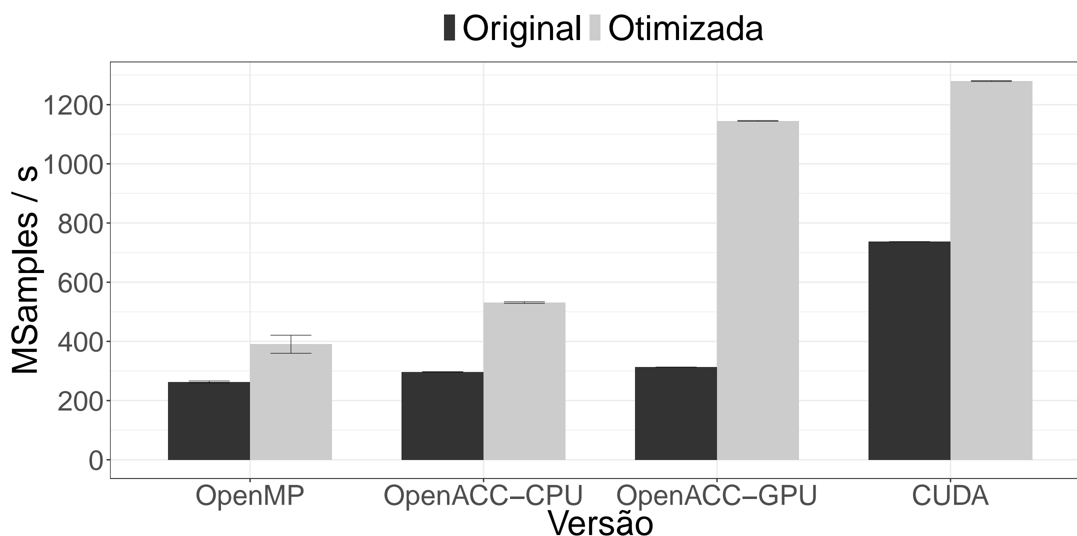

Figura 3. Desempenho das versões *original* e *otimizada* em diferentes IPPs.

5.2. Consumo de Energia e Eficiência Energética

A Figura 4 mostra o consumo médio energético de toda a máquina em *Kilo joules* para as codificações *OpenMP* e *OpenACC* executando em *CPU*, *CUDA* e *OpenACC* executando em *GPU*. Podemos notar que, em geral, a versão *otimizada* apresenta uma redução no consumo médio de energia em todas as implementações, em média a versão *original* consumiu 12,5kJ e a *otimizada* 10,8kJ, isso representa uma redução de 10,8% no consumo de energia.

Outro aspecto que podemos notar, é que as interfaces de programação paralela possuem comportamentos diferentes entre si. Analisando a implementação em *OpenMP*

para *multicore*, a versão *otimizada* obteve uma redução de 6,0% na energia utilizada. Enquanto a implementação *OpenACC* teve uma redução de 24,5% no *multicore* e 21,4% na *GPU*. A implementação *CUDA* foi a que teve a menor redução no consumo, cerca de 3,6%.

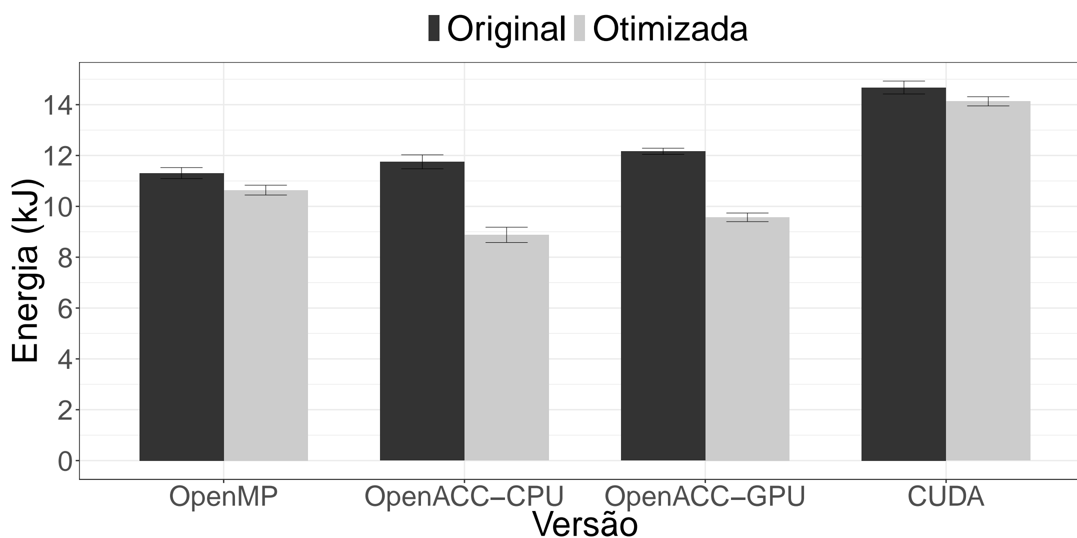

Figura 4. Consumo de Energia das versões *original* e *otimizada*.

Buscando analisar qual implementação possui a melhor eficiência energética, a Figura 5 apresenta a relação *nano joule* (nJ) por *sample* para cada implementação. Observando a relação geral entre a versão *original* e a *otimizada*, a *original* apresentou em média 1066 nJ/sample, já a *otimizada* apresentou em média 537 nJ/sample, isso representa um aumento na eficiência de 49,6%.

Analisando a eficiência energética entre as interfaces de programação paralela e as versões *original* e *otimizada*, é possível verificar que *OpenMP* tem uma eficiência 28,9% melhor na versão *otimizada* em relação a *original*. No caso da versão *OpenACC*, a eficiência energética foi melhorada em 52,8% no *multicore* e 74,6% na *GPU*. A implementação *CUDA otimizada* teve uma eficiência 42,6% melhor que a versão *original*.

Comparando o consumo de energia com a eficiência energética podemos notar que apesar das codificações *OpenMP* e *OpenACC* consumirem menos energia que a versão *CUDA*, elas apresentaram uma pior eficiência energética, em relação a ela, na versão *original*, onde a versão *CUDA* foi até 67,5% mais eficiente que as outras codificações.

A codificação *CUDA* apresentou a melhor eficiência na versão *otimizada*, seguido pela codificação *OpenACC-GPU* com uma diferença de 8,9%. Escrever um código em *OpenACC* é mais simples que em *CUDA* [Memeti et al. 2017] e a codificação é portátil, ou seja, a mesma codificação pode ser usada em *multicore* e em *GPU*. Entretanto, mostramos que existem perdas tanto em desempenho quanto em consumo de energia e eficiência energética.

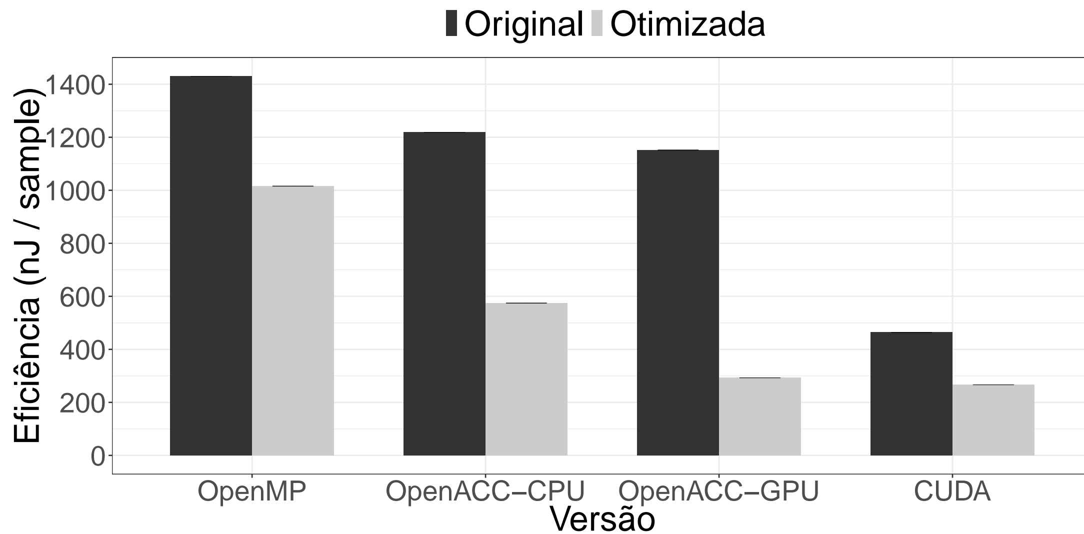

Figura 5. Eficiência Energética das versões *original* e *otimizada*.

6. Conclusão e Trabalhos Futuros

As arquiteturas atuais de *multicore* e *GPU* introduzem vários desafios, tais como a necessidade de codificar adequadamente as aplicações paralelas para tirar o melhor proveito delas. Neste artigo, otimizamos uma aplicação de propagação de ondas, a Modelagem *Fletcher*, para arquiteturas *multicore Intel Xeon* e *GPU NVIDIA Pascal*. Mostramos que o cálculo das derivadas cruzadas demandava 76% das operações de ponto flutuante e devido a isso propomos uma otimização para reduzir esse custo. Essa otimização também pode ser aplicada em outras aplicações similares e outras arquiteturas.

Em nossos experimentos, também analisamos o desempenho, o consumo de energia e a eficiência energética da aplicação. Além de fazer um estudo sobre a viabilidade de utilizar uma versão portátil do código. Os resultados mostraram que versão com maior desempenho foi a versão *CUDA*, seguida da versão *OpenACC-GPU*. A diferença de desempenho entre a versão *CUDA* e a versão *OpenACC-GPU*, que pode ser executada em diversas arquiteturas, foi de 10%. Isso mostra que existe um custo em utilizar uma versão portátil, mas que dependendo do caso de teste, pode ser interessante. Após, mostramos que o consumo de energia não é a única métrica determinante, pois, uma aplicação pode consumir mais energia, executando de forma rápida e assim, possuir a melhor eficiência energética. Nesse sentido, analisamos a eficiência energética, e mostramos que as versões *CUDA* e *OpenACC* tiverem a melhor eficiência, 266 e 292 nano *joules* por *sample*, respectivamente.

Como trabalho futuro, pretendemos avaliar arquiteturas com placas *FPGA* integradas programadas com *OpenCL*. Também estamos interessados em verificar a eficiência do uso de várias *GPUs* no mesmo nó e comparar nossos resultados com as soluções propostas pela literatura.

Referências

Andreolli, C., Thierry, P., Borges, L., Skinner, G., and Yount, C. (2015). Characterization and Optimization Methodology Applied to Stencil Computations. In Reinders, J.

and Jeffers, J., editors, *High Performance Parallelism Pearls*, pages 377–396. Morgan Kaufmann, Boston.

Caballero, D., Farrés, A., Duran, A., Hanzich, M., Fernández, S., and Martorell, X. (2015). Optimizing Fully Anisotropic Elastic Propagation on Intel Xeon Phi Coprocessors. In *2nd EAGE Workshop on HPC for Upstream*, pages 1–6.

Carrijo Nasciutti, T., Panetta, J., and Pais Lopes, P. (2018). Evaluating optimizations that reduce global memory accesses of stencil computations in gpgpus. *Concurrency and Computation: Practice and Experience*, page e4929.

Castro, M., Francesquini, E., Dupros, F., Aochi, H., Navaux, P. O. A., and Méhaut, J.-F. (2016). Seismic wave propagation simulations on low-power and performance-centric manycores. *Parallel Computing*, 54.

Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D., and McDonald, J. (2001). *Parallel programming in OpenMP*. Morgan kaufmann.

Clapp, R. G. (2015). Seismic Processing and the Computer Revolution(s). In *Society of Exploration Geophysicists (SEG) Technical Program Expanded Abstracts 2015*, pages 4832–4837.

Clapp, R. G., Fu, H., and Lindtjorn, O. (2010). Selecting the right hardware for reverse time migration. *The Leading Edge*, 29(1).

Fletcher, R. P., Du, X., and Fowler, P. J. (2009). Reverse time migration in tilted transversely isotropic (tti) media. *Geophysics*, 74(6):WCA179–WCA187.

J. Dongarra, H. M. and Strohmaier, E. (2019). Top500 supercomputer: June 2019. `https://www.top500.org/lists/2019/06/`. [Acesso em: 10 Jul. 2019].

Kukreja, N., Louboutin, M., Vieira, F., Luporini, F., Lange, M., and Gorman, G. (2016). Devito: Automated fast finite difference computation. In *Procs. of the 6th Intl. Workshop on Domain-Spec. Lang. and High-Level Frameworks for HPC*, WOLFHPC '16, pages 11–19. IEEE Press.

Lukawski, M. Z., Anderson, B. J., Augustine, C., Capuano Jr, L. E., Beckers, K. F., Livesay, B., and Tester, J. W. (2014). Cost analysis of oil, gas, and geothermal well drilling. *Journal of Petroleum Science and Engineering*, 118:1–14.

Memeti, S., Li, L., Pllana, S., Kołodziej, J., and Kessler, C. (2017). Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, pages 1–6. ACM.

Niu, X., Jin, Q., Luk, W., and Weston, S. (2014). A Self-Aware Tuning and Self-Aware Evaluation Method for Finite-Difference Applications in Reconfigurable Systems. *ACM Trans. on Reconf. Technology and Systems*, 7(2).

Nvidia (2016). Developer Zone - CUDA Toolkit Documentation.

Ott, R. L. and Longnecker, M. T. (2015). *An introduction to statistical methods and data analysis*. Nelson Education.

Pavan, P. J., Serpa, M. S., Padoin, E. L., Schnorr, L. M., Navaux, P. O. A., and Panetta, J. (2018). Improving i/o performance of rtm algorithm for oil and gas simulation. In

2018 Symposium on High Performance Computing Systems (WSCAD), pages 270–270. IEEE.

Qutob, H. et al. (2004). Underbalanced drilling; remedy for formation damage, lost circulation, & other related conventional drilling problems. In *Abu Dhabi International Conference and Exhibition*. Society of Petroleum Engineers.

Rubio, F., Farrés, A., Hanzich, M., de la Puente, J., and Ferrer, M. (2013). Optimizing Isotropic and Fully-anisotropic Elastic Modelling on Multi-GPU Platforms. In *75th EAGE Conference & Exhibition*, pages 10–13. EAGE.

Sabne, A., Sakdhnagool, P., Lee, S., and Vetter, J. S. (2014). Evaluating performance portability of openacc. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 51–66. Springer.

Sanders, J. and Kandrot, E. (2010). *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

Serpa, M. S., Cruz, E. H., Diener, M., Krause, A. M., Navaux, P. O. A., Panetta, J., Farrés, A., Rosas, C., and Hanzich, M. (2019a). Optimization strategies for geophysics models on manycore systems. *The International Journal of High Performance Computing Applications*, 33(3):473–486.

Serpa, M. S., Moreira, F. B., Navaux, P. O., Cruz, E. H., Diener, M., Griebler, D., and Fernandes, L. G. (2019b). Memory performance and bottlenecks in multicore and gpu architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 233–236. IEEE.

Slaight, T. (2002). Platform management ipmi controllers, sensors, and tools. In *Intel Developer Forum*.

Subramaniam, B., Saunders, W., Scogland, T., and Feng, W.-c. (2013). Trends in energy-efficient computing: A perspective from the green500. In *2013 International Green Computing Conference Proceedings*, pages 1–8. IEEE.

Terpstra, D., Jagode, H., You, H., and Dongarra, J. (2010). Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer.

Wienke, S., Springer, P., Terboven, C., and an Mey, D. (2012). Openacc—first experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer.

Witten, I. H., Frank, E., Hall, M. A., and Pal, C. J. (2016). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

Yuen, D. A., Wang, L., Chi, X., Johnsson, L., Ge, W., and Shi, Y. (2013). *GPU solutions to multi-scale problems in science and engineering*. Springer.

Zhebel, E., Minisini, S., Kononov, A., and Mulder, W. (2013). Performance and scalability of finite-difference and finite-element wave-propagation modeling on Intel's Xeon Phi. In *Society of Exploration Geophysicists (SEG) Technical Program Expanded Abstracts 2013*, pages 3386–3390.