Análise de desempenho e produtividade de APIs baseadas em diretivas com suporte à programação paralela usando GPUs

Silvio de Souza Neves Neto[1], **Álvaro Luiz Fazenda**[1]

[1]Instituto de Ciência e Tecnologia – Universidade Federal de São Paulo (UNIFESP)
São José dos Campos, SP – Brasil

{silvio.neves, alvaro.fazenda}@unifesp.br

Abstract. *OpenMP and OpenACC are programming models developed to abstract the complexity of parallel programming, providing portability between different architectures, including heterogeneous systems. This work seeks to make a comparative analysis between these two patterns, from the point of view of development productivity and obtained performance. The evaluations will be conducted by applying the aforementioned standards to three common algorithms, running on GPUs. The results show a significant advantage to OpenACC in both performance and productivity, considering the scope limitation for the case of study.*

Resumo. *OpenMP e OpenACC são modelos de programação desenvolvidos com o objetivo de abstrair a complexidade da programação paralela, proporcionando portabilidade entre diferentes arquiteturas, incluindo sistemas heterogêneos. Este trabalho busca fazer uma análise comparativa entre estes dois padrões, dos pontos de vista de produtividade de desenvolvimento e de desempenho computacional obtido. As avaliações foram conduzidas a partir da programação aderente aos padrões citados, aplicada a três algoritmos conhecidos, com execução em arquitetura que faz uso de GPUs. Os resultados, limitados à análise dos casos de estudo, mostram uma clara vantagem em desempenho e produtividade para o uso de OpenACC.*

1. Introdução

Nos últimos anos, dois tipos de arquiteturas computacionais têm sido majoritariamente empregadas no âmbito da computação de alto desempenho: uma baseada em CPUs de múltiplos núcleos e outra baseada em aceleradores do tipo GPU ligados ao sistema principal [Vergara Larrea et al. 2020]. A edição de junho de 2022 da lista Top500[1] mostra que os equipamentos com aceleradores do tipo GPU, tanto da NVIDIA quanto AMD, conquistaram uma parcela significativa do mercado. A maior parte entre os dez sistemas mais bem ranqueados na lista faz uso desse tipo de hardware acelerador.

A computação heterogênea apresenta um potencial enorme em aplicações científicas que, por sua vez, têm se tornado cada vez mais complexas [Xu et al. 2013]. Aceleradores como as GPUs são utilizados como um co-processador massivamente paralelo para cálculos intensivos, no entanto, programar esses aceleradores e obter um bom

[1]https://top500.org/lists/top500/2022/06/

desempenho pode ser uma tarefa complexa e tediosa, levando em conta que a heterogeneidade e aceleração especializada de hardware adicionam um nível de complexidade a mais à programação paralela [Wienke et al. 2014].

Visando portabilidade e produtividade, modelos de programação paralela baseados em diretivas têm surgido. Esses modelos fornecem recursos de linguagem de alto nível que abstraem os detalhes da arquitetura de baixo nível, facilitando imensamente o porte de aplicações para hardware paralelo e tirando vantagem da computação heterogênea sem a necessidade de desenvolver e manter várias versões do código para cada tipo de dispositivo.

OpenMP [OpenMP Architecture Review Board 2021] e OpenACC [Chandrasekaran and Juckeland 2017] são duas das APIs baseadas em diretivas mais comumente usadas em programação paralela e também o foco deste trabalho. OpenACC oferece suporte à *offloading* para aceleradores desde sua primeira versão. Já OpenMP foi inicialmente criado para a programação de CPUs com múltiplos núcleos e recebeu suporte à programação de aceleradores a partir da versão 4.0. A lista de compiladores que implementam algum nível de suporte à OpenACC e OpenMP para aceleradores é crescente e inclui muitos dos mais populares usados na área de computação de alto desempenho, como os compiladores da NVIDIA, GCC, AMD e Cray.

Com duas APIs baseadas em diretivas bem estabelecidas, é natural o questionamento sobre qual usar. Diversos trabalhos foram feitos comparando essas tecnologias e diversas abordagens foram utilizadas: comparação baseada em padrões [Wienke et al. 2014], porte de *Mini-Apps* para sistemas heterogêneos [Vergara Larrea et al. 2020], tradução de programas em uma API para outra [Pino et al. 2017], aplicação em *benchmarks* que medem tempo de execução, consumo de energia e acréscimo no número de linhas de código [Memeti et al. 2017], entre outras.

Embora análises comparativas já tenham sido realizadas anteriormente com OpenMP e OpenACC, essa é uma tarefa que continuará sendo relevante enquanto esses modelos também forem, tendo em vista que novas versões são lançadas e os compiladores constantemente ampliam o suporte para novos recursos das versões mais recentes. Este trabalho irá analisar o desempenho e a produtividade no desenvolvimento de programas conhecidos, com programação aderente aos citados padrões, em execução paralela em GPUs.

2. Trabalhos Relacionados

Diversos trabalhos na literatura realizam análises comparativas entre APIs de programação paralela, pois, como já mencionado, com múltiplas ferramentas disponíveis para a execução de uma mesma tarefa, a avaliação de qual é a mais adequada para determinado contexto se faz necessária, e continua relevante enquanto novas ferramentas surgirem e as já existentes continuarem evoluindo.

O trabalho de [Memeti et al. 2017] realiza um estudo empírico das características das APIs OpenMP, OpenCL, OpenACC e CUDA em relação à produtividade de programação, desempenho computacional e consumo de energia. Para isso, é utilizada uma métrica de produtividade baseada na quantidade de linhas de código necessárias a paralelização. O estudo é realizado a partir do uso das APIs em aplicações de *benchmarks* pertencentes aos mais variados domínios, executadas em sistemas heterogêneos

contendo aceleradores, incluindo GPUs. No entanto, OpenACC e OpenMP não foram aplicados aos mesmos *benchmarks*. Desta forma, os resultados obtidos não apresentaram comparações entre estes dois padrões.

Um trabalho que busca avaliar especificamente OpenMP e OpenACC é o de [Wienke et al. 2014], onde uma comparação baseada em padrões entre as duas APIs para computação com aceleradores é realizada. Através da comparação dos construtores disponíveis, a programabilidade dos dois modelos são examinadas. A conclusão estabelecida é que OpenACC está um passo à frente de OpenMP no que diz respeito à programação com aceleradores, ao oferecer mais recursos para tal atividade. No entanto, este trabalho é de 2014 e, hoje, os padrões passaram por diversas mudanças e ganharam novos recursos em suas novas versões. Além disso, até a data de publicação do trabalho mencionado, implementações para OpenACC existiam apenas para GPUs, e em OpenMP estava-se limitado ao uso do acelerador Intel Xeon Phi, impossibilitando comparações de desempenho.

Adotando uma abordagem mais prática, o trabalho de [Pino et al. 2017] buscou traduzir códigos em OpenMP para OpenACC, explorando as similaridades e divergências de funcionalidade entre os dois modelos. Além disso, realizou um estudo empírico de desempenho e portabilidade entre plataformas *multicore* e GPUs para diferentes tamanhos de carga de trabalho. O estudo conclui que traduzir aplicações entre os dois modelos pode ser um processo complexo devido à ausência de funcionalidades e diferenças de padrão de paralelização. Como por exemplo o fato de OpenACC não apresentar construtores integralmente similares aos conceitos de *tasks* e *barriers* presentes em OpenMP.

O porte de programas para OpenACC e OpenMP em sistemas heterogêneos também foi realizado no trabalho de [Vergara Larrea et al. 2020], onde as APIs foram aplicadas em quatro *Mini-Apps*, representativos de aplicações reais de computação de alto desempenho, e as implementações compiladas com diferentes compiladores em cinco sistemas distintos. As maiores dificuldades relatadas encontram-se no esforço em adaptar os códigos para o uso em certos compiladores, pois, na prática, os compiladores possuem diferentes níveis de maturidade na implementação das diretivas. Quanto ao desempenho das diferentes versões, houveram poucas disparidades observadas em ambos os padrões. Suas possíveis causas, teorizadas pelos autores, seriam diferenças de maturidade do compilador ou falta de ajustes específicos para o modelo de programação alcançar um bom desempenho.

3. Metodologia

Foram adotados 3 algoritmos como *benchmarks* para esta avaliação, sendo eles, a implementação do Jogo da Vida (em inglês, *Game of Life*) [Gardner 1970], um dos mais famosos automatos celulares, uma implementação de computação em *Stencil* [Sloot et al. 2003] com números de ponto-flutuante, operação muito comum em simulações computacionais de computação científica, tal como ocorre comumente na área de Dinâmica de Fluidos Computacional, e a operação de Multiplicação de Matrizes [Nykamp 2022], também considerando tipos de dados com ponto-flutuante, uma operação da Álgebra Linear comumente utilizada pelo método de Aprendizagem Profunda (do inglês, *Deep Learning*) [LeCun et al. 2015].

Todos os códigos-fonte foram programados em linguagem C. O compilador uti-

lizado para compilação dos programas testados foi o NVC V.21.5-0, disponível na ferramenta NVIDIA HPC *Software Development Kit* . Ele foi escolhido por possuir suporte para a maioria das funcionalidades do OpenACC versão 2.7 e para um grande subconjunto das funcionalidades referentes ao uso de GPUs do OpenMP 5.0 (*Offloading*), além de estar disponível gratuitamente. Os testes foram executados em um computador pessoal com uma placa gráfica *NVIDIA GeForce GTX 1650 Mobile* e um processador *Intel Core i5-9300H* (*Driver Version: 470.141.03, CUDA Version: 11.4*). Em todos os códigos com padrão OpenACC foi utilizada a diretiva *parallel* para indicar o trecho a ser portado para execução em GPU.

A métrica de desempenho adotada foi apenas o tempo de execução, em segundos. O tempo foi medido através de instrumentação do código-fonte e representa a execução de todo o processamento na GPU, incluindo as necessárias transferências de dados entre CPU e GPU, as quais são realizadas apenas uma única vez para todos os códigos criados. Assim, ocorre uma transferência de dados da memória principal da CPU para a GPU antes de qualquer processamento paralelo na GPU, e uma consequente transferência no sentido oposto ao final da execução de todos os cálculos na GPU. Dessa forma, os tempos dispendidos nas operações de transferências de dados entre memórias do sistema *host* e do dispositivo (*device*) foram minimizados. A métrica de produtividade utilizada foi baseada na diferença em relação da quantidade de caracteres e de palavras (diretivas e cláusulas) presentes nas linhas que representam as diretivas no código, tanto em OpenMP quanto em OpenACC.

4. Resultados

Para o *benchmark* Jogo da Vida, que trabalha apenas com números inteiros, e para o *benchmark* que simula a computação em *Stencil*, as avaliações de desempenho computacional podem ser conferidas nas Tabelas 1 e 2. Na primeira coluna para ambas as Tabelas, além da especificação da API utilizada (OpenACC ou OpenMP) para promover o paralelismo em GPU, encontram-se as legendas “nc”, “c1”, “c2” e “c1 & c2”, as quais significam, respectivamente:

1. “nc” - *no loop collapse*, onde o codígo-fonte não recebeu, nas diretivas em OpenACC ou OpenMP, nenhuma cláusula para colapsar laços aninhados existentes.
2. “c1” - *collapse loop 1*. Neste caso existe a cláusula para colapsar dois loops aninhados encontrados no código, os quais correspondem ao núcleo de cálculo da operação realizada (Computação para o autômato celular GOL ou Computação em *Stencil*). A cláusula para colapsar laços transforma dois ou mais laços aninhados em apenas um único e mais extenso laço de repetição, através da transformação automática do código e dos respectivos índices de elementos dos *arrays* envolvidos, permitindo aumentar a granularidade do paralelismo, aumentando a quantidade de operações concorrentes com menor carga computacional.
3. “c2” - *collapse loop 2*, referente ao uso de cláusula que permite colapsar dois laços aninhados responsáveis por copiar uma matriz bidimensional recém atualizada para um *array* de mesmo tamanho que representa o estado antigo (uma iteração anterior) da computação.
4. “c1 & c2” - *collapse loop 1 and 2*, onde se aplicam as opções “c1” e “c2” no código-fonte, ou seja, usa-se instruções para colapsar os laços 1 e 2 dos algoritmos que simulam GOL e *Stencil*.

Nas mesmas citadas Tabelas 1 e 2, as colunas 2, 3 e 4, representam os tempos de computação em segundos medidos para computar sobre *arrays* 2D de tamanhos considerado pequeno (1250×1250), médio (2500×2500) e grande (5000×5000). Para ambos os casos os tempos medidos incluem 2000 iterações do laço externo principal, para a computação da solução, uma vez que ambos os experimentos constituem-se de problemas de solução iterativa. Deve-se citar ainda que os tempos apresentados em segundos representam também a média aritmética de 5 execuções. O desvio padrão medido para os testes são da ordem de 1% e 0,5% para os testes de tamanho 1250×1250 e 5000×5000, respectivamente.

Os resultados apresentados para os dois primeiros testes (Tabelas 1 e 2) mostram uma vantagem nos tempos de computação para o códigos aderentes ao padrão OpenACC, em comparação com o desempenho obtido usando-se OpenMP, em todas os casos simulados e medidos. A diferença se torna ainda mais evidente quando se verifica que para obtenção de maior eficiência computacional a versão com OpenMP exige que se usem cláusulas para colapsar todos os laços aninhados existentes, que operam sobre matrizes bidimensionais.

Na versão com OpenACC o uso de cláusulas para aninhamento de laços trouxe prejuízo ao desempenho para a simulação GOL (que utiliza-se de tipos de dados inteiros), mostrando que, neste código-fonte, o compilador conseguiu gerar código executável com paralelismo para a GPU de forma mais eficiente ao deixá-lo mais livre para realizar ações automáticas, o que favorece também a produtividade na codificação dos programas, visto que não exige cláusulas adicionais, o que não ocorreu no código com OpenMP. Considerando o caso de teste para a maior matriz no citado teste, o melhor desempenho obtido em OpenACC (sem colapsar laços) trouxe um *speedup* de 1,54 em relação à versão com OpenMP (colapsando laços), o que é bem significativo.

No teste com a computação em *Stencil*, que se utiliza de tipos de dados de ponto-flutuante, o uso de cláusulas para forçar o colapso de laços aninhados trouxe também benefícios no desempenho para a versão em OpenACC, mas com ganhos pouco significativos, atingindo um *speedup* de 1,03 (correspondente à aproximadamente 3% de ganho) para o caso com tamanho grande da matriz. Considerando o mesmo teste (*Stencil*) e caso (matriz grande), em OpenMP a inclusão da cláusula para colapsar laços aninhados permitiu um *speedup* de 12,5 (correspondente à um ganho de 92% no desempenho). Considerando ainda o mesmo caso de teste, para o mesmo tamanho de matriz, o melhor desempenho obtido em OpenACC trouxe um *speedup* de 1.008 em relação à versão com OpenMP, o que torna os dois códigos praticamente equivalentes em desempenho, com uma pequena vantagem para o caso com OpenACC, já que atinge um desempenho ligeiramente melhor sem necessidade de cláusulas adicionais para colapsar laços.

A Tabela 3 mostra os tempos médios de execução obtidos com a multiplicação de matrizes, correspondentes a 5 rodadas, onde o desvio padrão foi de 0,2% e 0,06% para a menor e maior dimensão, respectivamente. Nesse caso os tamanhos das matrizes bidimensionais são: pequena com 4000×4000 células, média com 6000×6000, e grande com 8000×8000 células. O algoritmo para multiplicação de matrizes bidimensionais utilizado é a forma tradicional utilizada, que inclui 3 laços aninhados, onde os dois primeiros correspondem ao percorrimento das linhas e colunas das matrizes, e o laço mais interno corresponde ao cálculo do produto escalar entre a linha de uma matriz e a coluna de outra

Tabela 1. Tempos de execução (seg.) para as diferentes versões do Jogo da Vida

Versão		1250×1250	2500×2500	5000×5000
OpenACC	nc	0,381	1,147	4,150
	c1	0,430	1,374	4,406
	c2	0,455	1,433	4,791
	c1&c2	0,502	1,660	5,059
OpenMP	nc	5,031	16,361	89,864
	c1	1,361	4,631	45,377
	c2	4,259	13,753	50,233
	c1&c2	0,650	2,153	6,380

Tabela 2. Tempos de execução (seg.) para as versões da computação em *Stencil*.

Versão		1250×1250	2500×2500	5000×5000
OpenACC	nc	0,573	2,011	7,754
	c1	0,580	2,051	7,627
	c2	0,603	2,016	7,641
	c1&c2	0,609	2,067	7,512
OpenMP	nc	2,746	9,205	94,591
	c1	1,313	4,565	55,884
	c2	2,089	6,766	46,407
	c1&c2	0,686	2,291	7,577

correspondente em quantidade de elementos. A opção de colapsar laços neste *benchmark* corresponde aos dois primeiros laços mais externos apenas, sendo o laço responsável pelo produto escalar calculado de forma serial por *thread*. De forma surpreendente, a cláusula para colapsar os laços neste teste produziu grandes prejuízos no desempenho em ambas as versões. Na versão OpenACC o tempo decorrido quase dobrou para todos os tamanhos de matrizes. Na versão OpenMP o tempo foi tão superior que impediu que o cálculo se concluísse em tempo factível para a matriz de maior tamanho. Comparando o desempenho computacional das versões em OpenACC e OpenMP, nota-se que versão em OpenACC consegue um desempenho muito superior para este teste, conseguindo executar a mesma operação quase na metade do tempo que a versão OpenMP.

Tabela 3. Tempos de execução (seg.) para a multiplicação de matrizes.

Versão		4000×4000	6000×6000	8000×8000
OpenACC	nc	1,509	4,553	10,512
	c	2,474	8,274	19,145
OpenMP	nc	2,525	8,622	19,973
	c	37,681	570,843	>999,999

A Tabela 4 mostra a quantidade de caracteres e de cláusulas e diretivas em linhas que especificam instruções para as APIs analisadas. Em todos os casos, usou-se as versões identificadas por "nc", ou seja, sem as cláusulas necessárias para se colapsar os laços. As instruções em OpenACC ou OpenMP necessárias para essa ação apresenta os mesmos 11 caracteres, portanto, considerá-las na contagem irá causar adições fixas nos valores

apresentadas, na mesma proporção para todos os casos. Cabe também citar que não estão sendo contados as palavras reservadas $\#pragma$, bem como omp e acc. Fica claro que as versões em OpenACC exigem uma quantidade significativamente menor de caracteres nas linhas das diretivas, bem como uma quantidade menor de palavras, o que é uma clara vantagem de produtividade.

Tabela 4. Análise de produtividade para os *benchmarks* - Quantidade de caracteres e palavras em linhas de diretivas (versões "nc")

Versão		Caracteres	Diretivas
OpenACC	GoL	248	14
	Stencil	107	7
	MatMul	76	4
OpenMP	GoL	373	30
	Stencil	169	14
	MatMul	105	7

5. Conclusões e trabalhos futuros

Analisando os resultados obtidos na Seção 4, vemos que o padrão OpenACC sempre apresentou melhores resultados. Em um dos testes o ganho foi sutil (processamento em *Stencil*), porém, para a multiplicação de matrizes a versão OpenACC é muito mais eficiente, conseguindo *Speedup* de $1,54$ para as melhores versões de ambos.

Nota-se que para o padrão OpenACC inexiste a necessidade de se adicionar cláusulas adicionais em códigos com laços aninhados, indicando para o compilador colapsa-los em um único laço. Pelo melhor entendimento dos autores, a forma automática de tratamento de laços aninhados do compilador utilizado em códigos com instruções em OpenACC produz um programa executável de forma eficiente. O mesmo não ocorre com a versão em OpenMP, que exige tais instruções para execução com tempos mais próximos da versão comparada.

A utilização de programação aderente ao padrão OpenACC permite ainda maior portabilidade do código, pois o mesmo código-fonte, sem nenhuma alteração pode ser compilado para utilizar processamento paralelo em CPUs. Para a versão OpenMP é necessário que se tenham instruções específicas para execução em CPUs e GPUs, gerando muito mais linhas com diretivas ou diferentes versões.

O OpenACC traz vantagens na produtividade por permitir códigos onde as linhas de diretivas tem menor quantidade de caracteres e cláusulas específicas. Entretanto, devido ao fato do padrão OpenMP ser mais antigo e, consequentemente, mais difundido entre seus potenciais usuários, a produtividade pode também se beneficiar de um melhor conhecimento prévio sobre o padrão, ainda que as diretivas e cláusulas para promover paralelismo em GPUs sejam significativamente diferentes das instruções para processamento *multithread* em CPUs. Tal fato poderá ser fruto de análise em trabalhos futuros.

O presente trabalho apresenta a limitação de usar apenas um único compilador, produzido e mantido pela NVIDIA, que pode apresentar características de desempenho particulares que podem ter influenciado nos resultados de desempenho. Além disso a quantidade de programas testados também pode favorecer características específicas para

uma determinada API. O mesmo se pode dizer em relação ao *hardware* utilizado. Para superar essas limitações, prevê-se uma extensão desse trabalho incluindo mais testes com diferentes programas, padrões de programação paralela, diferentes compiladores e diferentes arquiteturas, além de investigar a aplicação de novas métricas de produtividade.

6. Agradecimentos

Este trabalho foi parcialmente financiado pelo Projeto #2019/26702-8, Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP).

Referências

Chandrasekaran, S. and Juckeland, G. (2017). *OpenACC for Programmers*. Addison-Wesley Educational, Boston, MA.

Gardner, M. (1970). Mathematical games. *Scientific American*, 223(4):120–123.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

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*, ARMS-CC '17, page 1–6, New York, NY, USA. Association for Computing Machinery.

Nykamp, D. Q. (2022). Math insight. `https://mathinsight.org/`.

OpenMP Architecture Review Board (2021). Openmp application programming interface - version 5.2 november 2021. `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf`.

Pino, S., Pollock, L., and Chandrasekaran, S. (2017). Exploring translation of openmp to openacc 2.5: lessons learned. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 673–682.

Sloot, P. M. A., Tan, C. J. K., Dongarra, J. J., and Hoekstra, A. G., editors (2003). *Computational science - ICCS 2002*. Lecture Notes in Computer Science. Springer Berlin, Berlin, Germany, 2002 edition.

Vergara Larrea, V. G., Budiardja, R. D., Gayatri, R., Daley, C., Hernandez, O., and Joubert, W. (2020). Experiences in porting mini-applications to openacc and openmp on heterogeneous systems. *Concurrency and Computation: Practice and Experience*, 32(20):e5780.

Wienke, S., Terboven, C., Beyer, J. C., and Müller, M. S. (2014). A pattern-based comparison of openacc and openmp for accelerator computing. In Silva, F., Dutra, I., and Santos Costa, V., editors, *Euro-Par 2014 Parallel Processing*, pages 812–823, Cham. Springer International Publishing.

Xu, R., Chandrasekaran, S., and Chapman, B. (2013). Exploring programming multi-gpus using openmp and openacc-based hybrimodel. In *2013 IEEE International Symposium on Parallel DistributeProcessing, Workshops and Phd Forum*, pages 1169–1176.