PROGRAMAÇÃO PARALELA EM CPU E GPU: UMA AVALIAÇÃO DO DESEMPENHO DAS APIS OPENMP, CUDA, OPENCL E OPENACC

Parallel Programming in CPU and GPU: A Performance Evaluation of OpenMP, CUDA, OpenCL and OpenACC APIs

GUSTAVO ROSA CORRÊA[1]*, MAURÍCIO SULZBACH[1].

[1]Departamento de Engenharias e Ciência da Computação, URI – Câmpus de Frederico Westphalen – RS.
*mateuszagonel@hotmail.com.

Resumo: Este artigo teve como objetivo avaliar o desempenho das principais APIs de programação paralela, através da execução de algoritmos em CPU e GPU. Para que a avaliação de desempenho fosse possível, primeiramente foram instaladas em um computador as APIs CUDA (GPU), OpenCL (GPU), OpenACC (GPU) e OpenMP (CPU). Em seguida, foram selecionados para execução em cada API os algoritmos SRAD V1, HotSpot e PathFinder, que fazem parte do *benchmark* Rodinia. Para avaliar o desempenho de cada API foram realizadas trinta execuções de cada algoritmo. O tempo dessas trinta execuções foi computado, sendo a média o valor utilizado, e desse tempo de execução obteve-se o intervalo de confiança das amostras, e também o *speedup* que cada algoritmo obteve em relação à execução sequencial de cada API. Através desta pesquisa, foi possível constatar que a GPU obteve um desempenho superior em relação a CPU na maioria dos algoritmos. Acredita-se que esse resultado ocorreu pelo fato da GPU possuir mais núcleos se comparada a CPU, apesar da frequência dos núcleos ser inferior.

Palavras-chave: Desempenho, Algoritmos Paralelos, CPU, GPU.

Abstract: This article aimed to evaluate the performance of the main parallel programming APIs, through the implementation of algorithms in CPU and GPU. For the performance evaluation were possible, were first installed on a computer the CUDA APIs (GPU), OpenCL (GPU), OpenACC (GPU) and OpenMP (CPU). They were then selected to run in each API algorithms SRAD V1, HotSpot and PathFinder, that are part of Rodinia benchmark. To evaluate the performance of each API were performed thirty executions of each algorithm. The time these thirty plays been computed, the average value being used, and this runtime gave the confidence interval of the samples and also the *speedup* that each algorithm obtained regarding the sequential execution of each API. Through this research, it was found that the GPU achieved a higher performance in the CPU Most algorithms. It is believed that this result occurred by the GPU actually have more cores compared to CPU, despite the frequency of the cores to be lower.

Keywords: Performance, Parallel Algorithm, CPU, GPU.

1 INTRODUÇÃO

Como consequência do avanço das arquiteturas de CPU e GPU, nos últimos anos houve um aumento no número de APIs (*Application Programming Interface*) de programação paralela para os dois dispositivos. Atualmente tem-se P*thread* e OpenMP para programação paralela em CPU e CUDA, OpenCL e mais recentemente OpenACC para GPU. A partir do surgimento dessas APIs, muitas aplicações foram reescritas para serem executadas de forma paralela em CPU ou GPU, trazendo em muitos casos, uma redução do tempo de execução. Porém, não há uma avaliação que confirme qual dessas APIs apresenta maior desempenho e em que situações isso acontece.

Além disso, muitos problemas exigem um hardware mais robusto para sua execução e isso acaba envolvendo custos mais elevados para a sua aquisição. Atualmente com o surgimento e a popularização das GPUs, pode-se ter em um único computador uma capacidade grande de paralelismo, que unida a CPU possibilite a execução de tarefas em menor tempo, a um custo acessível.

Sendo assim, o objetivo dessa pesquisa foi avaliar o desempenho das principais APIs de programação paralela para CPU e GPU e verificar os principais motivos que ocasionaram ou não a obtenção do desempenho. Diante disso, na primeira etapa buscou-se avaliar o desempenho das principais APIs de programação paralela para CPU e GPU (OpenMP, CUDA, OpenCL e OpenACC). Para isso, essas APIs foram estudadas e instaladas, sendo inicialmente realizados testes para validar os estudos e analisar a metodologia dos testes. Na sequência, foram selecionados para execução em cada API os algoritmos SRAD V1, HotSpot e PathFinder, que fazem parte do benchmark Rodinia.

Para avaliar o desempenho de cada API foram realizados trinta execuções de cada algoritmo. O tempo dessas trinta execuções foram computados, sendo a média o valor utilizado, e desse tempo de execução obteve-se o intervalo de confiança das amostras, e também o *speedup* que cada algoritmo obteve em relação a execução sequencial de cada API.

Os *speedups* dos testes mostraram que a GPU obteve um desempenho superior em relação a CPU na maioria dos algoritmos. Acredita-se que esse resultado ocorreu pelo fato da GPU possuir mais núcleos se comparada a CPU, apesar da frequência dos núcleos ser inferior. A única exceção onde o *speedup* das amostras foi melhor na CPU do que na GPU foi no algoritmo HotSpot. Isso ocorreu devido a três fatores principais: a limitada configuração da GPU utilizada para os testes, o tempo necessário para a troca de informações entre

CPU e GPU, que no caso dos algoritmos que utilizam apenas CPU não existe, e o pouco trecho de código paralelizável do algoritmo.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Programação Paralela

Programação paralela é uma forma de programação em que problemas sequenciais podem ser divididos em etapas para serem executadas simultaneamente aproveitando os recursos de hardware existentes. A ideia central da programação paralela consiste em dividir um grande problema em problemas menores para que possam ser resolvidos ao mesmo tempo por unidades de processamento diferentes, objetivando dessa forma ganhar desempenho. Essa técnica é conhecida como "divide and conquer" – dividir e conquistar (TSUCHIYAMA, 2015).

Esse paradigma é utilizado em diferentes áreas, tais como: aplicações de computação gráfica, simulações computacionais, pesquisa e classificação de dados, aplicações científicas e de engenharia, onde o rendimento da aplicação é fortemente dependente da eficiência computacional do hardware. Também se pode citar aplicações que envolvem pesquisa e análise de dados, medicina, geração energética e fatores climáticos (GRAMA et al., 2003) como exemplo de aplicações que utilizam o paralelismo.

Em tempos passados, o aumento de desempenho de uma aplicação (*speedup*) era obtido através de uma CPU com uma velocidade de *clock* maior, o que aumentava significativamente a cada ano que se passava. A partir de 2003, quando a velocidade do *clock* chegou a 4GHz, o aumento no consumo de energia e a dissipação do calor tornaram-se fatores limitantes, o que fez a velocidade do *clock* da CPU estagnar (KIRK and HUW, 2010). Dessa forma, os fabricantes de processadores foram forçados a desistir de seus esforços em aumentar a velocidade do *clock*, passando a adotar um novo método: aumentar o número de núcleos dentro do processador.

Uma vez que a velocidade do *clock* da CPU permaneceu a mesma ou até mais lenta, a fim de economizar energia, software sequencial desenvolvido para executar em um único processador não se torna mais rápido apenas substituindo a CPU por um modelo mais recente. Para tirar o máximo de proveito dos processadores atuais, o software deve ser projetado para executar processos em paralelo (KIRK and HUW, 2010).

Atualmente CPUs com mais de um *core* são comuns mesmo para os computadores portáteis de consumo básico. Isso mostra que o processamento paralelo não é apenas útil para a realização de cálculos avançados, mas que está se tornando comum em várias aplicações (TSUCHIYAMA, 2015). Os ambientes de computação também estão cada vez mais diversificados, explorando as capacidades de uma gama de processadores *multi-core*, de unidades de processamento central (CPUs), de processadores de sinais digitais e hardware reconfigurável (FPGAs) e de unidades gráficas de processamento (GPUs). Essa heterogeneidade faz com que o processo de desenvolvimento eficiente de software apresente muitas ferramentas e arquiteturas, resultando em uma série de desafios para a comunidade da programação (GASTER et al., 2012). Além disso, tem-se buscado cada vez mais aproveitar a capacidade computacional de dispositivos com arquiteturas diferentes.

2.2 OpenMP

OpenMP é uma especificação que fornece um modelo de programação paralela com compartilhamento de memória. Essa API é composta por um conjunto de diretivas que são adicionadas as linguagens C/C++ e Fortran (OPENMP, 2015) utilizando o conceito de *threads*, porém sem que o programador tenha que trabalhar diretamente com elas (MATLOFF, 2015). Esse conjunto de diretivas quando acionado e adequadamente configurado cria blocos de paralelização e distribui o processamento entre os núcleos disponíveis. O programador não necessita se preocupar em criar *threads* e dividir as tarefas manualmente no código fonte. O OpenMP se encarrega de fazer isso em alto nível.

O OpenMP não é uma linguagem de programação. Ele representa um padrão que define como os compiladores devem gerar códigos paralelos através da incorporação nos programas sequenciais de diretivas que indicam como o trabalho será dividido entre os *cores*. Dessa forma, muitas aplicações podem tirar proveito desse padrão com pequenas modificações no código (SENA; COSTA, 2008). No OpenMP, a paralelização é realizada com múltiplas *threads* dentro de um mesmo processo. As *threads* são responsáveis por dividir o processo em duas ou mais tarefas que poderão ser executadas simultaneamente. Diferente dos processos em que cada um possui seu próprio espaço de memória, cada *thread* compartilha o mesmo endereço de memória com as outras *threads* do mesmo processo, porém cada *thread* tem a sua própria pilha de execução (SENA; COSTA, 2008).

O modelo de programação do OpenMP é conhecido por *fork-join*, onde um programa inicia com uma única *thread* que executa sozinha todas as instruções até encontrar uma região paralela, que é identificada por uma diretiva OpenMP (OPENMP, 2015). Ao chegar nessa região, um grupo de *threads* é alocado e juntas executam o código paralelizado. Ao finalizar a execução do paralelismo as *threads* são sincronizadas e a partir desse ponto somente uma *thread* (inicial) é que segue com a execução do código sequencial. O *fork-join* pode ocorrer diversas vezes e é dependente do número de regiões paralelas que o programa possui (SENA; COSTA, 2008).

2.3 CUDA

CUDA é uma plataforma de computação paralela e um modelo de programação criados pela NVIDIA em 2006. Seu objetivo é possibilitar ganhos significativos de desempenho computacional aproveitando os recursos das unidades de processamento gráfico (GPU). Através da API CUDA pode-se enviar código C, C++ e Fortran diretamente à GPU, sem necessitar de uma nova linguagem de compilação (NVIDIA, 2015b). A tecnologia CUDA é de abordagem proprietária, concebida para permitir acesso direto ao hardware gráfico específico da NVIDIA.

Ao utilizar CUDA também é possível gerar código tanto para a CPU como para a GPU. CUDA oferece um conjunto de bibliotecas e o compilador *NVCC*, onde é possível explicitar dentro do código fonte as instruções que devem ser

executadas na CPU, na GPU ou em ambas. Para isso tornou-se necessário adicionar algumas extensões à linguagem C padrão (LINCK, 2010). Em CUDA a GPU é vista como um dispositivo de computação adequado para aplicações paralelas. Tem seu próprio dispositivo de memória de acesso aleatório e pode ser executada através de um grande número de *threads* em paralelo.

Na arquitetura CUDA a GPU é implementada como um conjunto de multiprocessadores. Cada um dos multiprocessadores tem várias *Arithmetic Logic Unit* (ALU) que em qualquer ciclo de *clock* executam as mesmas instruções, mas em dados diferentes (MANAVSKI, 2015). CUDA por limitar a execução em dispositivos somente fabricados pela NVIDIA traz consigo um conjunto de instruções que não são compatíveis com as demais APIs. Ao se migrar uma aplicação para CUDA tem-se que codificar quase que na totalidade o algoritmo.

2.4 OpenCL

OpenCL é uma API independente de plataforma que permite aproveitar as arquiteturas de computação paralelas disponíveis, como CPUs *multi-core* e GPUs, tendo sido desenvolvida objetivando a portabilidade. Essa API, criada pelo Khronos Group em 2008, define uma especificação aberta em que os fornecedores de *hardware* podem implementar. Por ser independente de plataforma, a programação em OpenCL é mais complexa, comparando com uma API específica para uma plataforma, como é o caso da CUDA. Enquanto a arquitetura CUDA está restrita aos dispositivos fabricados pela NVIDIA, o OpenCL possui um amplo suporte de fabricantes, bastando apenas a adequação de seu SDK (*Software Development Kit*) ao *framework* (SANTOS, et al., 2015).

Atualmente na sua versão 2.1 (KHRONOS GROUP, 2015), a especificação OpenCL é realizada em três partes: linguagem, camada de plataforma e *runtime*. A especificação da linguagem descreve a sintaxe e a API para escrita de código em OpenCL, que executa nos aceleradores suportados: CPUs *multi-core*, GPUs *many-core* e processadores OpenCL dedicados. A camada de plataforma fornece ao desenvolvedor acesso às rotinas que buscam o número e os tipos de dispositivos no sistema. Assim, o desenvolvedor pode escolher e inicializar os dispositivos adequados para o processamento. Já o *runtime* possibilita ao desenvolvedor enfileirar comandos para execução nos dispositivos, sendo também o responsável por gerenciar os recursos de memória e computação disponíveis.

2.5 OpenACC

Desenvolvida por um grupo de empresas incluindo principalmente NVIDIA, Portland Group Inc, CAPS Enterprise e CRAY, o OpenACC define uma especificação para execução de programas desenvolvidos em C, C++ e Fortran a partir de uma CPU para um dispositivo acelerador. Seus métodos provêm um modelo de programação para realizar a aceleração de instruções para diferentes tipos de dispositivos *multi-core* e *many-core* (OPENACC, 2015).

Através de um conjunto de diretivas, o OpenACC analisa a estrutura e os dados do programa e quais partes foram divididas entre o *host* (CPU) e o dispositivo acelerador. A partir dessa etapa, um mapeamento otimizado é gerado para ser executado em *cores* paralelos (PGI, 2015). Além da aceleração, que é o principal objetivo dessa API, o OpenACC fornece uma forma de migrar aplicativos através de pequenas mudanças na forma sequencial do algoritmo (OPENACC, 2015).

O modelo de execução alvo do OpenACC são *hosts* em conjunto com dispositivos aceleradores, como é o caso da GPU. A responsabilidade do *host* é receber a carga do aplicativo e direcionar as ações para o dispositivo acelerador. Essas ações são compostas geralmente por regiões que contém instruções de repetição ou *kernels*. O dispositivo acelerador apenas executa as instruções que lhe foram repassadas pelo *host*. O *host* deve ainda gerenciar a alocação de memória no dispositivo acelerador, transferir os dados do e para o *host*, enviar as instruções de execução e aguardar o término da execução (OPENACC, 2015).

3 RESULTADOS E DISCUSSÕES

3.1 Metodologia dos Testes

A primeira tarefa proposta baseou-se em estudar as APIs de programação paralela: OpenMP (CPU) e CUDA (GPU), OpenCL (GPU) e OpenACC (GPU). Procedeu-se então o estudo de alguns algoritmos de cada API para entender o método de programação, como ocorre o paralelismo, inicialmente com algoritmos simples, até a execução de algoritmos mais complexos. Através dessa atividade foram realizados alguns testes de desempenho, mas sem computar nenhum dado.

Já na segunda etapa, realizaram-se testes em três algoritmos do *bechmark* Rodinia: SRAD V1, HotSpot e PathFinder, com a finalidade de avaliar o desempenho de cada API. Foram executados os três algoritmos do *benchmark* nas APIs OpenMP, CUDA, OpenCL e OpenACC. A metodologia dos testes foi assim definida. Os algoritmos foram compilados através dos compiladores *gcc* (OpenMP), *nvcc* (CUDA) *pgcc* (OpenACC), e *gcc* (OpenCL). Os três algoritmos foram executados sequencialmente (1 *thread* em CPU) e paralelamente em CPU (4 *threads*) e em GPU com a quantidade máxima de *threads* suportada pela placa gráfica. Trinta execuções para cada configuração foi a quantidade utilizada que atendeu de forma satisfatória ao intervalo de confiança das amostras.

As execuções foram via terminal Linux. O ambiente gráfico foi desabilitado para que a placa gráfica ficasse totalmente destina aos cálculos. O comando *time* foi utilizado no terminal para capturar o tempo de execuções dos algoritmos.

O hardware utilizado nos testes foi:

- CPU Intel Core i5-330 CPU @ 3.00GHz x 4
  - Memória Física: 6 GB
  - Frequência: 3.00GHz
  - Largura de Dados: 64 bit
  - Núcleos: 4
  - *Threads*: 4
  - Cache: 6 MB
- GPU GeForce 210
  - Memória: 512MB
  - Clock de Mémoria: 500MHz
  - Núcleos: 16

- Clock Gráfico (MHz): 586MHz
- Clock do Processador (MHz): 1402MHz
- Desempenho Gráfico: Low-580

3.2 Algoritmo SRAD V1

SRAD (*Speckle Reducing Anisotropic Diffusion*) traduzindo para o português: Reduzir Pequenas Manchas com Difusão Anisotrópica é um método de difusão para aplicações de imagem de ultra-som e radar, baseados em equações diferenciais parciais (PDE). Ele é usado para remover o ruído correlacionado localmente, conhecido como manchas, sem destruir características importantes da imagem. SRAD consiste de várias etapas de trabalho: extração de imagem, iterações contínuas sobre a imagem (preparação, redução, estatística, computação 1 e computação 2) e compressão de imagem. A dependência sequencial entre todos esses estágios requer sincronização após cada etapa (porque cada estágio opera em toda a imagem) (RODINIA, 2016).

O algoritmo SRAD V1 foi configurado para ser executado em CPU e GPU com os seguintes parâmetros:

- Linhas: 502;
- Colunas: 458;
- Coeficiente de Saturação: 0.5;
- Número de Iterações: 10.000;

A figura 1 apresenta o *speedup* da configuração do algoritmo SRAD V1 utilizando OpenMP com 4 *threads*, CUDA, OpenCL e OpenACC, em relação a execução sequencial do algoritmo.

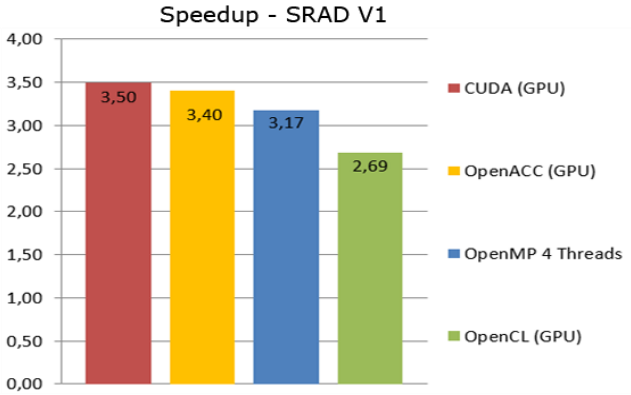

Fig. 1. *Speedup* do algoritmo SRAD V1.

Os melhores desempenhos ocorreram em dois algoritmos que paralelizam na GPU. O melhor desempenho foi do algoritmo em CUDA, com um *speedup* de 3,50. Em seguida vem o SRAD V1 em OpenACC, com um *speedup* de 3,40. Depois tem-se o OpenMP (CPU) com 4 *threads* obtendo um *speedup* de 3,17. E por fim a API OpenCL (GPU) com um *speedup* de 2,69.

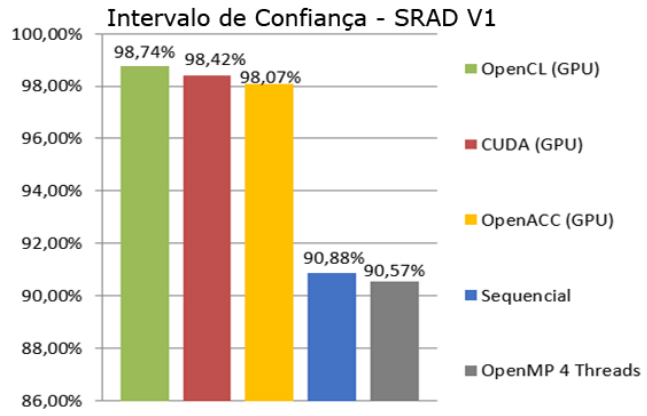

Fig. 2. Intervalo de Confiança do algoritmo SRAD V1.

O gráfico da figura 2 mostra o intervalo de confiança de todas as API's. Os resultados dos testes demostraram que os algoritmos executados na GPU obtiveram os melhores intervalos de confiança (valor estimado onde a média dos trinta testes realizados tem uma dada probabilidade de ocorrer).

O melhor intervalo de confiança foi da API OpenCL (GPU) com 98,74%, com uma média de tempo de execução de 22,21s e representando um *speedup* de 2,69. O segundo melhor intervalo de confiança foi do algoritmo em CUDA com 98,42%, com uma média de tempo de execução de 17,08s e *speedup* de 3,50. Logo em seguida vem o algoritmo SRAD V1 em OpenACC com um intervalo de Confiança de 98,07%, com uma média de tempo de execução de 17,55s e *speedup* de 3,40. Depois aparecem os algoritmos executados na CPU, com um intervalo de confiança consideravelmente mais baixo que os algoritmos executados na GPU. Sequencialmente o SRAD V1 obteve um intervalo de confiança de 90,88% com uma média de tempo de execução de 59,75s. Utilizando a API do OpenMP com 4 *thread* o intervalo de confiança foi de 90,57% e média de tempo de execução de 18,83s, representando um *speedup* de 3,17.

3.3 Algoritmo HotSpot

HotSpot é uma ferramenta amplamente utilizada para estimar a temperatura do processador baseado em uma planta em arquitetura e medições de potência simulados. A simulação térmica de forma iterativa resolve uma série de equações diferenciais para bloco. Cada célula de saída na grade computacional representa o valor médio de temperatura da área correspondente do chip (RODINIA, 2016).

O algoritmo HotSpot foi configurado para ser executado em CPU e GPU com os seguintes parâmetros:

- Linhas: 512;
- Colunas: 512;
- Número de Iterações: 10.000;

A figura 3 ilustra o *speedup* dos trinta testes executados no algoritmo HotSpot em cada API.

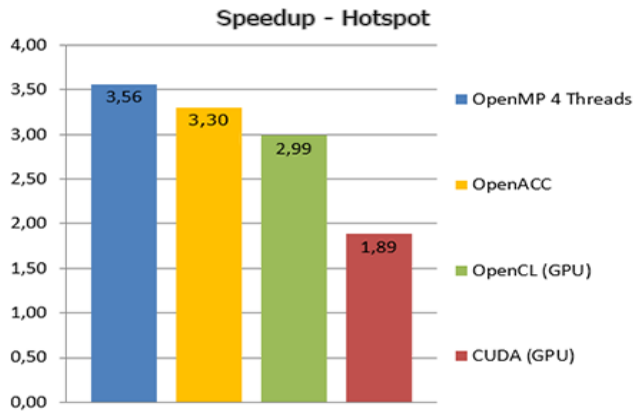

Fig. 3. *Speedup* do algoritmo HotSpot.

Nesse algoritmo, a CPU superou a GPU no *speedup*. Isso pode ter ocorrido em razão do algoritmo ter uma pequena parte de paralelização, ou seja, o tempo paralelizado na GPU não compensa, pois o tempo de transferir os dados para GPU, para aí sim paralelizar, acaba influenciando muito em um código que não tem um trecho significativo a ser paralelizado. O algoritmo OpenMP (CPU) com 4 *threads* obteve um *speedup* de 3,56, enquanto o algoritmo OpenACC (GPU) obteve um *speedup* de 3,30. A API OpenCL obteve um *speedup* de 2,99, e no algoritmo CUDA o *speedup* foi de 1,89.

A figura 4 demonstra novamente que os algoritmos executados na GPU obtiveram os melhores intervalos de confiança. O melhor foi da API OpenCL (GPU) com intervalo de confiança de 99,63%, com uma média de tempo de execução de 16,30s e representando um *speedup* de 2,99. O segundo melhor intervalo de confiança foi do algoritmo em OpenACC, com um intervalo de confiança de 98,30%, com uma média de tempo de execução de 14,73s e *speedup* de 3,30. Logo em seguida vem o algoritmo HotSpot em CUDA com um intervalo de confiança de 97,47%, com uma média de tempo de execução de 25,70s e *speedup* de 1,89. Depois aparecem os algoritmos executados na CPU, com intervalo de confiança consideravelmente mais baixo que os algoritmos executados na GPU. Utilizando a API do OpenMP com 4 *threads* adquiriu-se um intervalo de confiança de 90,60%, com uma média de tempo de execução de 13,68s e representando um *speedup* de 3,56. Sequencialmente o HotSpot obteve intervalo de confiança de 90,60% e média de tempo de execução de 48,67s.

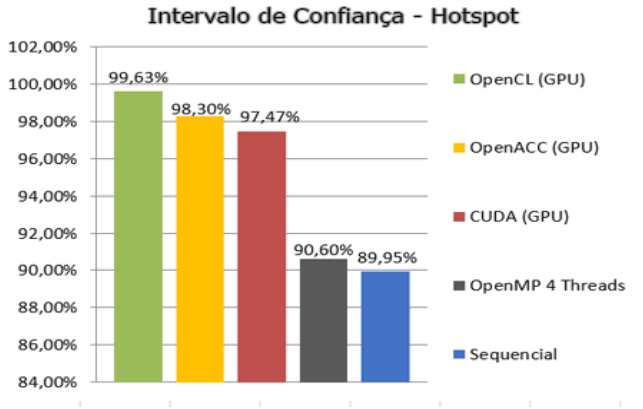

Fig. 4. Intervalo de Confiança do algoritmo HotSpot.

3.4 Algoritmo PathFinder

PathFinder é um algoritmo utilizado para encontrar a rota mais curta entre dois pontos. Pathfinder é uma variante mais prática na resolução de labirintos. Esse algoritmo utiliza programação dinâmica para encontrar um caminho em uma grade 2-D, a partir da linha de fundo para a fila superior com os menores pesos acumulados, onde cada passo do caminho se move para frente ou diagonalmente em frente. Ele repete linha por linha. Cada nó escolhe um nó vizinho na linha anterior que tem o menor peso acumulado e adiciona seu próprio peso à soma (RODINIA, 2016).

O algoritmo HotSpot foi configurado para ser executado em CPU e GPU com os seguintes parâmetros:

- Largura da Pirâmide: 10.000.000;
- Número de Passos: 140;
- Altura da Pirâmide: 20;

A figura 5 apresenta o *speedup* das versões dos algoritmos CUDA, OpenCL, OpenACC e OpenMP com 4 *threads*, em relação a execução sequencial do algoritmo.

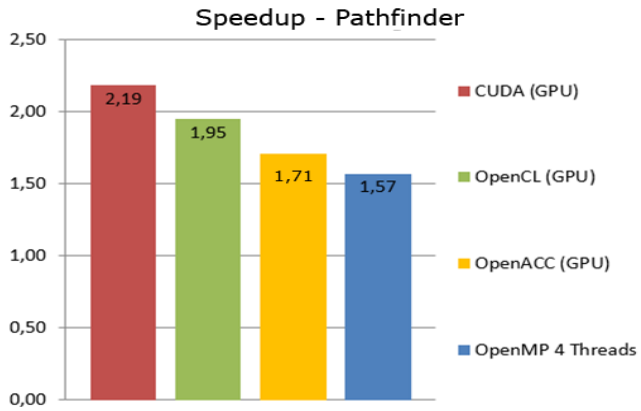

Fig. 5. *Speedup* do algoritmo PathFinder.

Pode-se perceber que a GPU se sobressaiu de forma significativa em relação a CPU. O algoritmo em CUDA obteve *speedup* de 2,19, o algoritmo OpenCL obteve um *speedup* de 1,95, o OpenACC obteve 1,71, e por fim vem a CPU com o OpenMP 4 *threads*, que obteve um *speedup* de 1,57 em relação a execução sequencial.

A figura 6 apresenta o intervalo de confiança de cada API. O melhor intervalo de confiança foi da API CUDA (GPU) com 98,89%, com uma média de tempo de execução de 14,94s e representando um *speedup* de 2,19. O segundo melhor intervalo de confiança foi do algoritmo OpenCL, com um intervalo de confiança de 98,23%, com uma média de tempo de execução de 16,76s e *speedup* de 1,95. Logo em seguida vem o algoritmo Sequencial com um intervalo de confiança de 98,17% e com uma média de tempo de execução de 32,66s. Utilizando a API do OpenMP com 4 *threads* adquiriu-se um intervalo de confiança de 97,94%, com uma média de tempo de execução de 20,85s e *speedup* de 1,57. Por fim o intervalo de confiança no algoritmo OpenACC foi de 97,91%, com uma média de tempo de execução de 19,19s e representando um *speedup* de 1,71. Percebe-se que no algoritmo PathFinder o intervalo de confiança foi praticamente perfeito em todas as APIs.

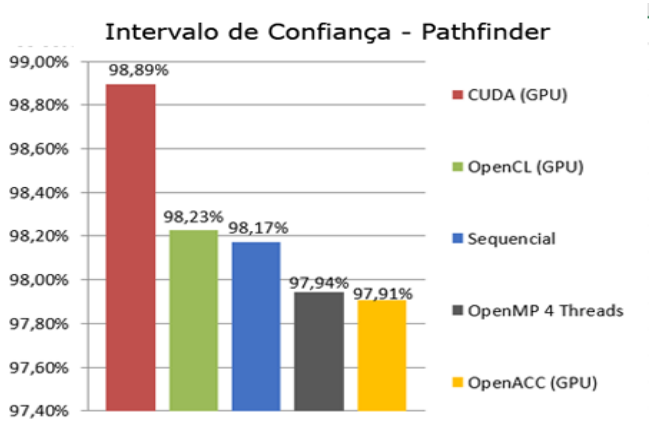

Fig. 6. Intervalo de Confiança do algoritmo PathFinder.

4 Conclusão

Os resultados demostram, com exceção do algoritmo HotSpot, que algoritmos executados na GPU através de (CUDA, OpenCL, OpenACC) comprovam a teoria de que quanto mais núcleos se tem a disposição para paralelizar o programa, melhor será o desempenho.

É claro que nem sempre isso se concretiza. Um exemplo claro disso foi o algoritmo HotSpot, que executado na CPU obteve um *speedup* melhor que todos os testes do mesmo algoritmo executados nas APIs para GPUs. Como comentado anteriormente, isso ocorreu devido a três principais fatores, são eles: a limitada configuração da GPU utilizada para os testes, o tempo necessário para a troca de informações entre CPU e GPU, que no caso dos algoritmos que utilizam apenas CPU não existe, e o pouco trecho de código paralelizável do algoritmo.

Após todos esses testes serem executados, computados e comparados, pode-se concluir que a programação paralela é necessária para implementar diversos tipos de sistemas complexos e pode trazer diversos benefícios aos sistemas tradicionais, mas infelizmente ela não é viável para todo tipo de cenário. Desta forma, uma das maneiras de avaliar se é uma boa ideia utilizar programação paralela é identificando se um determinado tipo de problema precisa ou pode ser paralelizado. Para isso, deve se analisar vários fatores, como a quantidade de trabalho do algoritmo; número de núcleos disponível para que o algoritmo possa conseguir um *speedup* no mínimo significativo em relação à execução sequencial; buscar identificar se um programa é *Embarrassingly Parallel*, isto é, um problema que sua solução pode ser dividida em várias partes independentes, sem que seja necessária uma sincronização entre elas para chegar à solução final. Problemas, ou sistemas, *Embarrassingly Parallel* permitem que tarefas sejam executadas simultaneamente, com pouca ou nenhuma necessidade de inter-coordenação. Se estes principais fatores forem levados em consideração antes se utilizar da programação paralela, será possível alcançar um desempenho melhor.

Referências

RODINIA. Accelerating Compute-Intensive Applications with Accelerators. Disponível em: <http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerators>. Acesso em: 03 jul. 2016.

GRAMA, A.; KARYPIS, G.; KUMAR, V.; GUPTA, A. Introduction to Parallel Computing. Secound Edition. Addison Wesley, 2003.

KHRONOS GROUP. OpenCL - The open standard for parallel programming of heterogeneous systems. Disponível em: <http://www.khronos.org/opencl>. Acesso em: 10 mar. 2015.

KIRK, D. B.; HUW, W. W. Programming Massively Parallel Processors. Morgan Kaufmann - Elsevier, 2010.

LINCK, G. Um componente para a exploração da capacidade de processamento de GPUs em Grades Computacionais. Santa Maria: UFSM, 2010.

MANAVSKI, S. A.; VALLE, G. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. Disponível em: <http://www.biomedcentral.com/1471-2105/9/S2/S10>. Acesso em: 09 fev. 2015.

OPENACC. Directives for Accelerators. Disponível em: <http://www.openacc.org/>. Acesso em: 11 fev. 2015.

OPENMP. Application Program Interface. Disponível em: <http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>. Acesso em: 13 fev. 2015.

SANTOS, T. S. e LIMA. K. A. B. e COSTA, D. A. da. e AIRES, K. R. T. Algoritmos de Visão Computacional em Tempo Real Utilizando CUDA e OpenCL. Disponível em: <http://www.die.ufpi.br/ercemapi2011/artigos/ST4_20.pdf>. Acesso em: 11 fev. 2015.

SENA, M. C. R. e COSTA, J. A. C. Tutorial OpenMP C/C++. Programa Campus Ambassador HPC, SUN Microsystems – Maceió, 2008.

MATLOFF, N. Programming on Parallel Machines. Disponível em: <http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf>. Acesso em: 12 jun 2016.

NVIDIA. Plataforma de Computação Paralela. Disponível em: <http://www.nvidia.com.br/object/cuda_home_new_br.html>. Acesso em: 15 mar. 2015.

TSUCHIYAMA, R. e NAKAMURA, T. e IIZUKA, T. e ASAHARA, A. e MIKI, S. The OpenCL Programming Book. Group, 2009.