UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

PROGRAMAÇÃO PARALELA HÍBRIDA PARA CPU E GPU: UMA AVALIAÇÃO DO OPENACC FRENTE A OPENMP E CUDA

DISSERTAÇÃO DE MESTRADO

Maurício Sulzbach

Santa Maria, RS, Brasil

2014

PROGRAMAÇÃO PARALELA HÍBRIDA PARA CPU E GPU:

UMA AVALIAÇÃO DO OPENACC FRENTE A OPENMP E CUDA

Maurício Sulzbach

Dissertação apresentada ao Curso de Mestrado do Programa de Pós-Graduação em Informática, Área de Concentração em Sistemas Paralelos e Distribuídos, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação.

Orientador: Prof. Dr. Benhur de Oliveira Stein

Santa Maria, RS, Brasil

2014

Sulzbach, Maurício
PROGRAMAÇÃO PARALELA HÍBRIDA PARA CPU E GPU: UMA AVALIAÇÃO DO OPENACC FRENTE A OPENMP E CUDA / Maurício Sulzbach.- 2014.
97 p.; 30cm

Orientador: Benhur de Oliveira Stein
Dissertação (mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Informática, RS, 2014

1. Programação paralela híbrida 2. Desempenho 3. OpenMP 4. CUDA 5. OpenACC I. de Oliveira Stein, Benhur II. Título.

E-mail: mauriciosulzbach@msn.com

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós-Graduação em Informática

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

PROGRAMAÇÃO PARALELA HÍBRIDA PARA CPU E GPU: UMA AVALIAÇÃO DO OPENACC FRENTE A OPENMP E CUDA

elaborada por
Maurício Sulzbach

como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

COMISSÃO EXAMINADORA:

Benhur de Oliveira Stein, Dr.
(Presidente/Orientador)

Andrea Schwertner Charão, Drª. (UFSM)

Márcia Cristina Cera, Drª. (UNIPAMPA)

Santa Maria, 22 de agosto de 2014.

AGRADECIMENTOS

Em primeiro momento, gostaria de agradecer a Deus pela vida e por esse momento especial. Agradecer a Ele também por ter me guiado nesses mais de 30.000 km percorridos durante esse período de mestrado.

Aos meus pais José e Elvira pelos seus ensinamentos, incentivos e amor incondicional depositados em mim.

A minha esposa Patrícia pelo seu amor, dedicação e compreensão nesses quase 11 anos juntos. Por não me deixar desanimar nos momentos mais difíceis e por sempre ter um ombro amigo para confortar.

Ao meu orientador professor Benhur de Oliveira Stein pelos ensinamentos e conselhos. Um exemplo de mestre!

A professora Andrea Schwertner Charão pelos desafios propostos que trouxeram muito conhecimento, pela avaliação e pelas contribuições neste trabalho.

A professora Márcia Cristina Cera pela disponibilidade de avaliação e pelas contribuições realizadas neste trabalho.

Ao colega Douglas Pires Borges pela grande amizade e pelo compartilhamento de ideias e angústias.

Aos colegas de URI por suprir minhas ausências, que não foram poucas.

A todos meu sincero muito obrigado!

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

PROGRAMAÇÃO PARALELA HÍBRIDA PARA CPU E GPU: UMA AVALIAÇÃO DO OPENACC FRENTE A OPENMP E CUDA

AUTOR: MAURÍCIO SULZBACH
ORIENTADOR: BENHUR DE OLIVEIRA STEIN
Data e Local da Defesa: Santa Maria, 22 de agosto de 2014.

Como consequência do avanço das arquiteturas de CPU e GPU, nos últimos anos houve um aumento no número de APIs de programação paralela para os dois dispositivos. Enquanto que OpenMP é utilizada no processamento paralelo em CPU, CUDA e OpenACC são empregadas no processamento paralelo em GPU. Na programação para GPU, CUDA apresenta um modelo baseado em funções que deixam o código fonte extenso e propenso a erros, além de acarretar uma baixa produtividade no desenvolvimento. Objetivando solucionar esses problemas e sendo uma alternativa à utilização de CUDA surgiu o OpenACC. Semelhante ao OpenMP, essa API disponibiliza diretivas que facilitam o desenvolvimento de aplicações paralelas, porém para execução em GPU. Para aumentar ainda mais o desempenho e tirar proveito da capacidade de paralelismo de CPU e GPU, é possível desenvolver algoritmos híbridos que dividam o processamento nos dois dispositivos. Nesse sentido, este trabalho objetiva verificar se as facilidades que o OpenACC introduz também refletem positivamente na programação híbrida com OpenMP, se comparado ao modelo OpenMP + CUDA. Além disso, o trabalho visa relatar as limitações nos dois modelos de programação híbrida que possam influenciar no desempenho ou no desenvolvimento de aplicações. Como forma de cumprir essas metas, este trabalho apresenta o desenvolvimento de três algoritmos paralelos híbridos baseados nos algoritmos do *benchmark* Rodinia, a saber, RNG, Hotspot e SRAD, utilizando os modelos híbridos OpenMP + CUDA e OpenMP + OpenACC. Nesses algoritmos é atribuída ao OpenMP a execução paralela em CPU, enquanto que CUDA e OpenACC são responsáveis pelo processamento paralelo em GPU. Após as execuções dos algoritmos híbridos foram analisados o desempenho, a eficiência e a divisão da execução em cada um dos dispositivos. Verificou-se através das execuções dos algoritmos híbridos que nos dois modelos de programação propostos foi possível superar o desempenho de uma aplicação paralela em uma única API, com execução em apenas um dos dispositivos. Além disso, nos algoritmos híbridos RNG e Hotspot o desempenho de CUDA foi superior ao desempenho de OpenACC, enquanto que no algoritmo SRAD a API OpenACC apresentou uma execução mais rápida, se comparada à API CUDA.

Palavras-chave: CPU. GPU. OpenMP. CUDA. OpenACC. Programação paralela híbrida. Desempenho.

ABSTRACT

Master's Dissertation
Informatics Post-degree Program
Universidade Federal de Santa Maria

HYBRID PARALLEL PROGRAMMING FOR CPU AND GPU: AN EVALUATION OF OPENACC AS RELATED TO OPENMP AND CUDA

AUTHOR: MAURÍCIO SULZBACH
ADVISOR: BENHUR DE OLIVEIRA STEIN
Defense Place and Date: Santa Maria, August 22nd, 2014.

As a consequence of the CPU and GPU's architectures advance, in the last years there was a raise of the number of parallel programming APIs for both devices. While OpenMP is used to make parallel programs for the CPU, CUDA and OpenACC are employed in the parallel processing in the GPU. In the programming for the GPU, CUDA presents a model based on functions that make the source code extensive and prone to errors, in addition to leading to low development productivity. OpenACC emerged aiming to solve these problems and to be an alternative to the utilization of CUDA. Similar to OpenMP, this API has policies that ease the development of parallel applications that run on the GPU only. To further increase performance and take advantage of the parallel aspects of both CPU and GPU, it is possible to develop hybrid algorithms that split the processing on the two devices. In that sense, the main objective of this work is to verify if the advantages that OpenACC introduces are also positively reflected on the hybrid programming using OpenMP, if compared to the OpenMP + CUDA model. A second objective of this work is to identify aspects of the two programming models that could limit the performance or on the applications' development. As a way to accomplish these goals, this work presents the development of three hybrid parallel algorithms that are based on the Rodinia's benchmark algorithms, namely, RNG, Hotspot and SRAD, using the hybrid models OpenMP + CUDA and OpenMP + OpenACC. In these algorithms, the CPU part of the code is programmed using OpenMP, while it's assigned for the CUDA and OpenACC the parallel processing on the GPU. After the execution of the hybrid algorithms, the performance, efficiency and the processing's splitting in each one of the devices were analyzed. It was verified, through the hybrid algorithms' runs, that, in the two proposed programming models it was possible to outperform the performance of a parallel application that runs on a single API and in only one of the devices. In addition to that, in the hybrid algorithms RNG and Hotspot, CUDA's performance was superior to that of OpenACC, while in the SRAD algorithm OpenACC was faster than CUDA.

Keywords: CPU. GPU. OpenMP. CUDA. OpenACC. Hybrid parallel programming. Performance.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

ALU	*Arithmetic Logic Unit*
AMD	*Advanced Micro Devices*
API	*Application Programming Interface*
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
DDR3	*Double-Data-Rate 3*
DLP	*Data-Level Parallelism*
DNA	*Deoxyribonucleic Acid*
DRAM	*Dynamic Random Access Memory*
EPD	Equações Diferenciais Parciais
FPGA	*Field-Programmable Gate Array*
GB	*Gigabyte*
GCC	GNU *Compiler Collection*
GHz	*Gigahertz*
GPC	*Graphics Processing Cluster*
GPGPU	*General-Purpose Computation on* GPU
GPU	*Graphics Processing Unit*
ILP	*Instruction-Level parallelism*
LCG	*Linear Congruential Generator*
LU	*Lower Upper* (Decomposição de LU)
MB	*Megabyte*
MEG	Magnetoencefalografia
MHz	*Megahertz*
MIMD	*Multiple Instruction, Multiple Data*
MISD	*Multiple Instruction, Single Data*
MP	*Multi Processing*
MPI	*Message Passing Interface*
NUMA	*Non-Uniform Memory Access*
NVCC	NVIDIA CUDA *Compiler*
OpenACC	*Open Accelerators*
OpenCL	*Open Computing Language*
OpenMP	*Open Multi-Processing*
PGCC	Portland Group C *Compiler*
PGI	Portland Group Inc.
PLP	*Process-Level Parallelism*
RAM	*Random Access Memory*
RNG	*Random Number Generator*
SDRAM	*Synchronous Dynamic Random Access Memory*
SIMD	*Single Instruction, Multiple Data*
SISD	*Single Instruction, Single Data*
SMP	*Symmetric Multi-Processor*
SMX	*Stream Multiprocessors*
SRAD	*Speckle Reducing Anisotropic Diffusion*
TLP	*Thread-level parallelism*

SUMÁRIO

1 INTRODUÇÃO

1.1 Contexto e motivação

A programação de alto desempenho tem apresentado um crescimento intenso nos últimos tempos, visto que dispositivos como a *Central Processing Unit* (CPU) e a *Graphics Processing Unit* (GPU) evoluíram muito nas suas arquiteturas (TANENBAUM, 2007; STALLINGS, 2010) e com isso, possibilitaram um aumento na capacidade computacional (NVIDIA, 2014). A CPU, por exemplo, em contrapartida à estagnação da velocidade do *clock* teve a introdução de mais núcleos dentro de um único processador (KIRK and HUW, 2010). Isso possibilitou a divisão de tarefas e consequentemente o aumento da capacidade de paralelismo. Já a GPU passou de um simples processador gráfico para um coprocessador paralelo. Atualmente elas são capazes de executar milhares de operações simultaneamente, gerando uma grande capacidade computacional (IKEDA, 2013; NVIDIA, 2013), que em muitas vezes supera o poder de processamento das tradicionais CPUs (LINCK, 2010). Esse fato se dá principalmente pelo avanço da arquitetura das GPUs, que passaram a conter centenas ou até milhares de núcleos de execução e memória própria em um único dispositivo (KIRK and HUW, 2010; PACHECO, 2011).

Aliado a isso, o surgimento de algumas APIs (*Application Programming Interface*) com suporte ao paralelismo possibilitaram usufruir da real capacidade computacional desses dispositivos. *Open Multi-Processing* (OpenMP) se destaca na paralelização de algoritmos para CPU por apresentar um conjunto de bibliotecas que possibilita a divisão de tarefas de forma simples e produtiva, através da introdução de algumas diretivas. Essas diretivas possibilitam uma paralelização implícita com pequenas alterações no código fonte sequencial (OPENMP, 2013). Já para a execução paralela em GPU tem-se *Compute Unified Device Architecture* (CUDA) e *Open Computing Language* (OpenCL) como principais APIs. CUDA, desenvolvida pela NVIDIA em 2006 (NVDIA, 2013), foi uma das precursoras, sendo nos dias de hoje uma das APIs mais consolidadas para gerar instruções à GPU. Possibilita enviar código C, C++ e Fortran diretamente para a GPU sem a necessidade de uma nova linguagem de compilação (NVIDIA, 2013). Oferece um conjunto de bibliotecas e um compilador, onde se podem explicitar dentro do código fonte as instruções que devem ser executadas na CPU, GPU ou em ambas (LINCK, 2010). OpenCL destaca-se por ser um padrão aberto que possibilita a execução de algoritmos paralelos para diferentes plataformas (OPENCL, 2014),

diferente de CUDA que é voltada apenas para os dispositivos fabricados pela NVIDIA (MULLA, 2011).

Buscando aumentar a capacidade de processamento paralelo aliado a um custo acessível, tem-se desenvolvido nos últimos anos algoritmos paralelos de forma híbrida utilizando CPU e GPU. A programação paralela híbrida objetiva unir dispositivos de computação com diferentes arquiteturas trabalhando em conjunto, visando atingir um maior desempenho. Além disso, a programação híbrida busca fazer com que cada conjunto de instruções possa ser executado na arquitetura que melhor se adapte (RIBEIRO; FERNANDES, 2013).

Uma das alternativas para o desenvolvimento de algoritmos híbridos para CPU e GPU é a utilização das APIs OpenMP para CPU e CUDA ou OpenCL para GPU. A união de duas ou mais APIs para uma execução paralela, por muitas vezes gera um código fonte extenso, com muitas instruções para troca de informações entre CPU e GPU que podem se tornar burocráticas, pouco produtivas e propensas a erros. Porém, na maioria das vezes consegue-se alcançar bons índices de desempenho (MULLA, 2011; REYES, 2012).

Percebendo que era possível alcançar desempenho na execução de algoritmos paralelos híbridos e vendo que a programação através de múltiplas APIs apresentava problemas na codificação, surgiu em 2011, através de um grupo formado pelas empresas NVIDIA, Portland Group Inc, CAPS Enterprise e CRAY, o OpenACC (OPENACC, 2013). A API OpenACC é um padrão aberto que possibilita acelerar aplicações para CPU e GPU através de diretivas, semelhantes ao OpenMP. Da mesma forma que o OpenMP, o OpenACC apresenta uma série de facilidades para execução de código paralelo em CPU e GPU, tais como (PGI, 2014): produtividade, minimização de recodificação a cada mudança de *hardware*, portabilidade e facilidade de aprendizado. Também, possibilita realizar balanceamento de carga entre os dispositivos com boas possibilidades de aumento de desempenho (PGI, 2014). Entretanto, não é mensurado até o momento se as facilidades que o OpenACC introduz refletem positivamente também no desempenho de aplicações híbridas em CPU e GPU, se comparadas com a utilização de OpenMP e CUDA.

Conhecer o desempenho que o OpenACC oferece, bem como identificar as limitações nos dois modelos de programação híbrida (OpenMP + CUDA e OpenMP + OpenACC) que possam influenciar no desempenho ou no desenvolvimento de aplicações híbridas, são as principais motivações para o desenvolvimento deste trabalho.

1.2 Objetivos e contribuição

Analisar se as facilidades que o OpenACC introduz no desenvolvimento de aplicações também refletem positivamente no desempenho de aplicações híbridas, se comparadas à utilização de OpenMP + CUDA e identificar as principais limitações nos dois modelos de programação propostos (OpenMP + CUDA e OpenMP + OpenACC) que possam, de alguma forma, influenciar no desempenho ou no desenvolvimento de aplicações híbridas são os principais objetivos deste trabalho.

Para atingir esses objetivos, o presente trabalho testou três algoritmos paralelos do *benchmark* Rodinia[1], a saber, RNG (*Random Number Generator*), HotSpot e SRAD (*Speckle Reducing Anisotropic Diffusion*) e implementou-os de forma híbrida em CPU e GPU, através dos modelos OpenMP + CUDA e OpenMP + OpenACC. Em seguida foram realizados testes sobre os algoritmos e realizada uma comparação de desempenho entre as diferentes versões/configurações dos problemas propostos, dentre os quais: tempo de execução, *speedup*, eficiência, divisão da execução dos algoritmos em CPU e GPU e tempo de transferência de dados entre os dispositivos.

Como contribuições o presente trabalho:

a) Apresenta uma forma de divisão híbrida para execução em CPU e GPU;
b) Sugere os modelos OpenMP + CUDA e OpenMP + OpenACC para o desenvolvimento de aplicações híbridas como sendo alternativas para o ganho de desempenho;
c) Discute as limitações no desenvolvimento de aplicações híbridas utilizando OpenMP + CUDA e OpenMP + OpenACC;
d) Evidencia que aplicações que necessitem da resolução de problemas em menor tempo tem uma alternativa interessante ao dividir o processamento entre CPU e GPU através dos modelos propostos;
e) Mostra que a API OpenACC é uma boa alternativa para a programação em GPU e híbrida em CPU e GPU, através de uma maior produtividade no desenvolvimento.

A seguir, a seção 1.3 irá apresentar a organização deste trabalho.

[1] *Benchmark* para computação paralela em arquiteturas de CPU e GPU que está disponível em https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php

1.3 Organização do texto

A organização desta dissertação apresenta a seguinte estrutura de capítulos. No capítulo dois é apresentada a fundamentação teórica, detalhando a programação paralela, a programação paralela híbrida, as principais APIs de programação paralela e algumas métricas de desempenho. No capítulo três discorre-se sobre os algoritmos selecionados. O capítulo quatro descreve a metodologia utilizada no trabalho, a forma de paralelização híbrida de cada algoritmo e o ambiente de execução dos testes. A avaliação dos resultados dos testes realizados é discutida no capítulo cinco, onde é apresentado, para cada algoritmo, o desempenho, a eficiência, a divisão de processamento em cada dispositivo e o tempo consumido na transferência dos dados entre os dispositivos. No capítulo seis são apresentadas as limitações da programação híbrida utilizando OpenMP + CUDA e OpenMP + OpenACC que podem influenciar no desempenho ou no desenvolvimento de aplicações. As considerações finais e trabalhos futuros são o foco do capítulo sete.

2 FUNDAMENTAÇÃO TEÓRICA

A fundamentação teórica desta dissertação tem por objetivo apresentar os principais assuntos necessários para a compreensão deste trabalho. Ela aborda em primeiro momento o paralelismo e a programação híbrida. Na sequência são apresentadas as APIs de programação paralela OpenMP, CUDA e OpenACC. Para finalizar são apresentadas algumas métricas de desempenho utilizadas para medir o resultado da execução dos algoritmos.

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 (HIRATA, 2011; TSUCHIYAMA, 2012).

Esse paradigma é utilizado em diferentes áreas, tais como: aplicações de computação gráfica, simulações computacionais, pesquisa e classificação de dados (RAUBER; RÜNGER, 2010), aplicações científicas e de engenharia (GRAMA et al., 2003), onde o rendimento da aplicação é fortemente dependente da eficiência computacional do *hardware*. Também pode-se 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 na 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, 2012). 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. A seguir, na seção 2.2 serão apresentados alguns aspectos relevantes sobre o funcionamento do paralelismo.

2.2 Funcionamento do paralelismo

A programação paralela sempre foi uma área de grande importância na computação, principalmente na computação de alto desempenho. No entanto, mais recentemente, com o avanço do *hardware* para arquiteturas *multi-core* ou *many-core* ela tornou-se fundamental para quase todos os tipos de aplicações. O surgimento dos processadores com várias unidades de computação em um único *chip* fez com que simples computadores pessoais se tornassem potenciais sistemas em paralelo. Porém, como ocorre na maioria das vezes, o avanço tecnológico traz consigo um grande desafio: reestruturar os sistemas existentes de forma a aproveitar os recursos adicionais de computação. Para tirar proveito do poder computacional e buscar a aceleração de uma aplicação é necessário um esforço adicional a nível de *software* (RAUBER; RÜNGER, 2010).

Através de linguagens de programação ou APIs com recursos paralelos é possível dividir um problema em rotinas menores e executá-las de forma segura simultaneamente (MATTSON et al., 2005). A figura 1 ilustra um problema sendo dividido em instruções menores para ser executado por uma unidade processadora, como por exemplo CPU ou GPU.

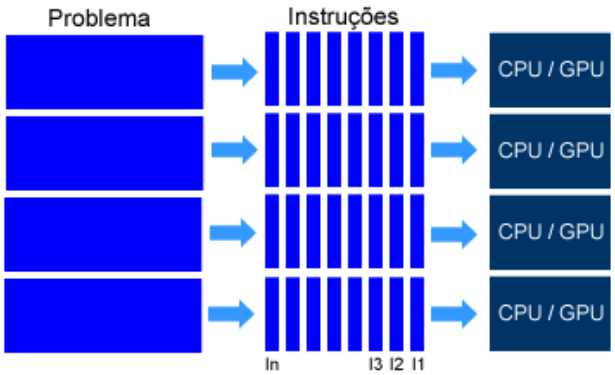

Figura 1 - Funcionamento do paralelismo
Fonte: Adaptação (BARNEY, 2013b).

Uma vez que o problema é dividido em rotinas menores, essas são atribuídas a processos ou *threads* que são enviados às unidades de processamento. *Scheduling* ou agendamento é o termo dado à atribuição das tarefas aos processos ou *threads*, onde também é fixada a ordem de execução das tarefas. O *scheduling* pode ser realizado de três maneiras (RAUBER; RÜNGER, 2010): (i) explícito no código fonte; (ii) através do ambiente de programação em tempo de compilação ou (iii) dinamicamente em tempo de execução. Depois disso, os processos ou *threads* são atribuídos às unidades de processamento, procedimento conhecido como mapeamento.

As tarefas de um algoritmo podem ser independentes uma das outras, ou seja, uma tarefa não depende do resultado de outra tarefa. Porém, há casos em que uma tarefa é dependente do resultado de outra tarefa. Quando isso ocorre é necessário que uma tarefa espere pela conclusão da outra tarefa. Como os programas paralelos geram tarefas que concorrem por recursos há a necessidade da coordenação e da sincronização entre os processos ou *threads*, a fim de executar corretamente (RAUBER; RÜNGER, 2010). Mesmo que um programa paralelo apresente o resultado de forma correta, ele pode deixar de realizar o melhor desempenho se não explorar a concorrência. Cuidados devem ser tomados para garantir que a sobrecarga gerada pela gestão da concorrência não interfira no tempo de execução do programa (EIJKHOUT et al., 2013; MATTSON et al., 2005). No processamento paralelo os métodos de sincronização e coordenação estão fortemente relacionados com a forma como a informação é trocada entre processos ou *threads* (RAUBER; RÜNGER, 2010).

Em computadores com memória distribuída os dados não podem ser acessados por todos os processadores. Para ocorrer a comunicação entre os processos é necessária a troca (envio e recebimento) de mensagens (GEBALI, 2011). Nos dispositivos com memória

compartilhada os dados podem ser acessados por todos os processadores ou núcleos. Nessa situação, todas as *threads* fazem acesso a mesma memória através de rotinas de leitura e escrita. A sincronização entre *threads* possibilita que o trabalho seja coordenado, de forma que uma *thread* não leia um dado antes que outra *thread* encerre a gravação na mesma área de memória. Especificar áreas de *barrier* (barreiras) também é uma forma de realizar o sincronismo entre processos ou *threads*. Somente depois que todos os processos ou *threads* tenham executado o código antes da barreira de sincronização, eles conseguem continuar a execução após a barreira (RAUBER; RÜNGER, 2010).

Para GEBALI (2011), o paralelismo por estar intimamente ligado a técnicas de *hardware* e *software* pode ser implementado em diferentes níveis:

a) Nível de dados: O *Data-level Parallelism* (DLP) opera simultaneamente em múltiplos dados, como por exemplo, adição e multiplicação de números binários e processamento vetorial;
b) Nível de instrução: Compreende a execução simultânea de mais de uma instrução por parte do processador. O *pipelining* é um exemplo do paralelismo em nível de instrução. Também chamado de *Instruction-level parallelism* (ILP);
c) Nível de *thread*: Mais conhecido por *Thread-level parallelism* (TLP). Nesse nível várias *threads* são executadas simultaneamente em um ou mais processadores compartilhando recursos computacionais;
d) Nível de processo: No *Process-Level Parallelism* (PLP) os processos são executados em um ou mais computadores, onde cada processo tem seus próprios recursos computacionais, como por exemplo, memória e registradores.

Encerrada a apresentação sobre o funcionamento do paralelismo, a seção 2.3 irá detalhar a programação paralela híbrida.

2.3 Programação paralela híbrida

O principal objetivo da programação paralela híbrida, também conhecida como programação heterogênea, é tirar proveito das melhores características dos diferentes modelos de programação, visando alcançar um maior desempenho. Ela consiste em mesclar a paralelização de tarefas complexas ou com intensas repetições, com atividades mais simples em mais de uma unidade processadora (RIBEIRO; FERNANDES, 2013) ou ainda, distribuir a

paralelização de acordo com as características computacionais dessas unidades. Ao trabalhar com paralelismo híbrido é importante considerar também como cada paradigma/API pode paralelizar o problema e como combiná-los de modo a alcançar o melhor resultado (SILVA, 2006).

A arquitetura do *hardware* que será utilizada deve ser sempre levada em consideração no momento que se projeta uma aplicação híbrida (RIBEIRO; FERNANDES, 2013). Esse fato atualmente ganha mais força pelo surgimento de uma gama de dispositivos com capacidades computacionais, porém de arquiteturas diferentes. Segundo SILVA (2006), o desenvolvimento de aplicações utilizando mais de um modelo de programação gera benefícios como a possibilidade de balanceamento de carga da aplicação e o poder computacional que as unidades processadoras em conjunto oferecem. Em contra partida, em algumas oportunidades há a necessidade de replicação de dados e sincronização de tarefas que acabam influenciando no tempo de execução. LAVALLEE e WAUTELET (2013) ainda apontam como desvantagens desse paradigma a complexidade no desenvolvimento e a exigência de conhecimento do *hardware*, além de não se ter uma garantia de ganho de desempenho, pois há custos extras adicionados.

A programação paralela híbrida pode ser utilizada em aplicações (LAVALLEE; WAUTELET, 2013):

- Que exigem balanceamento de carga dinâmico;
- Limitadas pelo tamanho de memória;
- Massivamente paralelas;
- Limitadas pela escalabilidade de seus algoritmos.

Como forma de classificar as unidades de processamento de acordo com a capacidade de paralelismo, a seção 2.4 irá abordar a taxonomia de Flynn.

2.4 Taxonomia de Flynn

Segundo STALLINGS (2010) e GEBALI (2011), a taxonomia introduzida por Flynn categoriza de forma simples a capacidade de processamento paralelo, levando em consideração os dados e as operações executadas sobre elas. A taxonomia de Flynn divide os computadores em:

a) *Single Instruction, Single Data* (SISD): Um único processador executa um único fluxo de instruções armazenado em uma única memória. Apenas uma

instrução é processada a cada ciclo. Os processadores de único núcleo são exemplos dessa categoria.

b) *Single Instruction, Multiple Data* (SIMD): Todos os processadores executam a mesma instrução em dados diferentes. Cada unidade de processamento tem seus dados em uma memória local, sendo que os dados são trocados entre os processadores através de rotinas de comunicação. Processadores vetoriais são exemplos enquadrados nessa categoria.

c) *Multiple Instruction, Single Data* (MISD): Conjunto de unidades de processamento executando operações distintas sobre o mesmo dado. Essa estrutura não foi implementada.

d) *Multiple Instruction, Multiple Data* (MIMD): Um conjunto de processadores executam simultaneamente instruções diferentes sobre diferentes conjuntos de dados. *Symmetric Multi-Processor* (SMP), *clusters* e sistemas *Non-Uniform Memory Access* (NUMA) se enquadram nessa categoria.

Para exemplificar a classificação criada por Flynn, a figura 2 ilustra as quatro subdivisões, bem como traz exemplos de processadores de cada categoria.

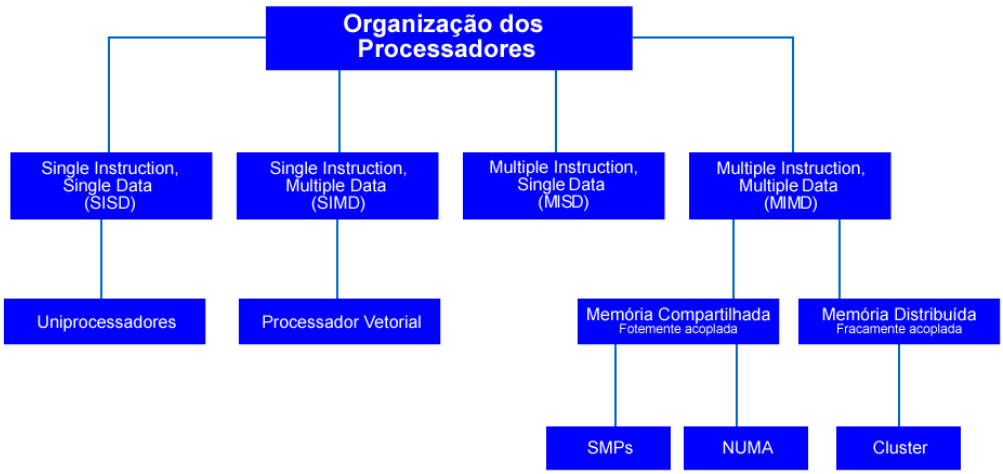

Figura 2 - Organização dos Computadores
Fonte: Adaptação STALLINGS (2010).

Observa-se também que no fluxo MIMD os processadores são divididos em memória compartilhada e memória distribuída. Nos processadores com memória compartilhada a comunicação ocorre através do acesso a uma única memória. O tempo de acesso a qualquer região da memória é aproximadamente o mesmo para cada processador, diferente da

arquitetura NUMA onde o tempo de acesso à memória é diferente de uma região para outra. O exemplo mais conhecido é o *Symmetric Multi-Processor* (SMP) (STALLINGS, 2010).

Em um sistema de memória distribuída, um conjunto de processadores ou SMPs independentes podem ser interligados para formar um *cluster* (STALLINGS, 2010). Nessa situação, um processador acessa informações da memória de outro processador através do envio e recebimento de mensagens.

2.5 Arquiteturas paralelas

Os principais dispositivos processadores utilizados na programação paralela híbrida na atualidade são a *Central Processing Unit* (CPU) e a *Graphics Processing Unit* (GPU). Cada unidade possui características e funcionalidades diferentes, que se bem exploradas podem trazer benefícios de desempenho. Sendo assim, esta seção irá detalhar a arquitetura desses dois dispositivos.

2.5.1 CPU

A *Central Processing Unit* tem a função de executar programas armazenados na memória principal, buscando suas instruções, examinando-as e executando-as uma após a outra. Seus componentes são conectados via barramento, onde podem ser transmitidos endereços, dados e sinais de controle. Esses barramentos podem ser internos à CPU ou externos como os que conectam à memória e aos dispositivos de entrada e saída (TANENBAUM, 2007). A organização de um simples computador com uma CPU, memória principal e dois dispositivos de entrada e saída é apresentada pela figura 3.

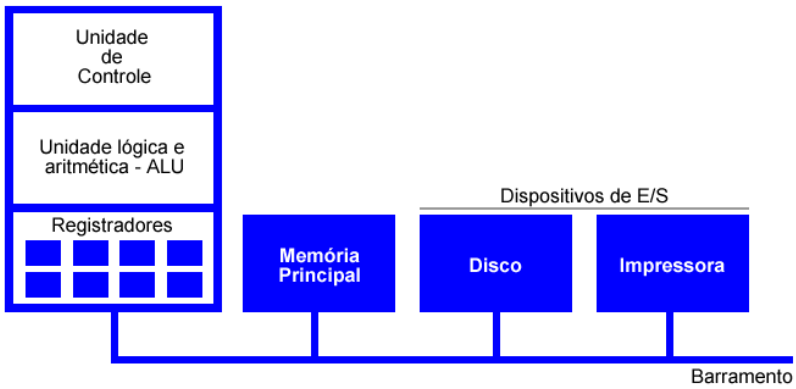

Figura 3 - Organização de um simples computador
Fonte: Adaptação TANENBAUM (2007).

Basicamente uma CPU é formada por algumas partes, dentre elas a unidade de controle, que tem a função de controlar a movimentação dos dados e das instruções que entram e saem do processador (STALLINGS, 2010). Já a unidade lógica e aritmética é a responsável por efetuar as operações, como por exemplo, adição e AND booleano, para executar as instruções. Também fazem parte da CPU os registradores que são utilizados para armazenar resultados temporários e para controles internos. Os registradores tem a vantagem de serem lidos e escritos em alta velocidade, pois estão localizados internamente à CPU (TANENBAUM, 2007).

Na sequência deste trabalho será apresentada a GPU, um coprocessador especializado que pode ser utilizado em conjunto com a CPU para auxiliar no processamento.

2.5.2 GPU

Inicialmente projetadas para o processamento gráfico, as GPUs (*Graphics Processing Unit*), ou unidades de processamento gráfico, passaram nos últimos anos por uma grande evolução, sendo atualmente consideradas como um coprocessador paralelo de propósito geral e muito utilizadas em aplicações de alto desempenho. Através desse crescimento e com o aumento da demanda por GPUs cada vez mais eficientes e flexíveis criou-se uma nova abordagem, a GPGPU (*General-Purpose Computation on* GPU). Esse novo enfoque objetiva explorar as vantagens das GPUs em aplicações de propósito geral altamente paralelizáveis, que exigem intenso fluxo de instruções (IKEDA, 2011).

Com o auxílio de linguagens de alto nível e APIs que dão o suporte ao desenvolvimento de aplicações para GPU, pode-se executar as partes sequenciais de um *software* em CPU ao mesmo tempo em se que acelera o processamento paralelo em GPU (KIRK and HUW, 2010; NVIDIA, 2013). A computação com GPU é possível porque esse dispositivo atualmente faz muito mais do que processar imagens. As modernas GPUs dispõem de *teraflops* de performance de ponto flutuante, processando tarefas de aplicativos projetados para diferentes segmentos, desde finanças, até medicina (NVIDIA, 2013).

A figura 4 ilustra a diferença da capacidade de processamento das GPUs se comparadas às CPUs. Também se observa que as CPUs apresentaram evolução na capacidade de processamento nos últimos anos, porém, o aumento da capacidade de processamento das GPUs foi maior. A diferença de desempenho entre uma CPU *multicore* e uma GPU *many-core* reside no *design* entre os dois tipos de processadores. O *design* da CPU é otimizado para desempenho de código sequencial. A CPU faz o uso de uma moderna lógica de controle,

permitindo que instruções de *threads* sejam executadas em paralelo, por vezes fora de sua ordem sequencial, porém mantendo a aparência de execução sequencial. Além disso, a CPU através de áreas maiores de memórias *cache* possibilita reduzir a latência no acesso aos dados de aplicações grandes e complexas (KIRK and HUW, 2010).

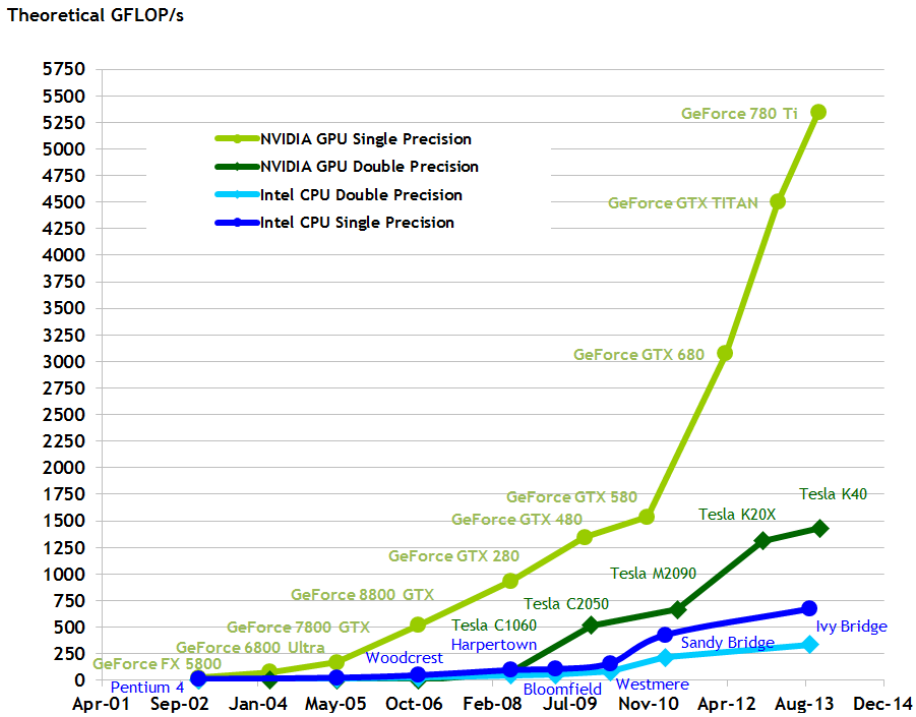

Figura 4 - Operações de ponto flutuante por segundo
Fonte: (NVIDIA, 2014b).

Moldada para jogos e aplicações gráficas, a arquitetura da GPU é baseada em um grande número de *threads* objetivando otimizar a execução. As GPUs apresentam pequenas memórias *cache* que auxiliam a controlar a largura de banda quando múltiplas *threads* acessam os mesmos dados, sem a necessidade de ir à memória DRAM (*Dynamic Random Access Memory*). Além disso, as GPUs são voltadas para computação intensa, altamente paralelizável. Dessa forma, nas GPUs existem muito mais áreas dedicadas ao processamento de ponto flutuante (KIRK and HUW, 2010) do que áreas de controle. Por serem processadores com características SIMD (*Single Instruction, Multiple Data*) (OWENS et al, 2007) – embora não puros, pois as GPUs atuais apresentam diversos núcleos que são capazes de executar fluxos de instruções independentes (PACHECO, 2011) – as GPUs também não necessitam de unidades de controle complexas e isso justifica as pequenas áreas de controle ilustradas pela figura 5 (OWENS et al, 2007).

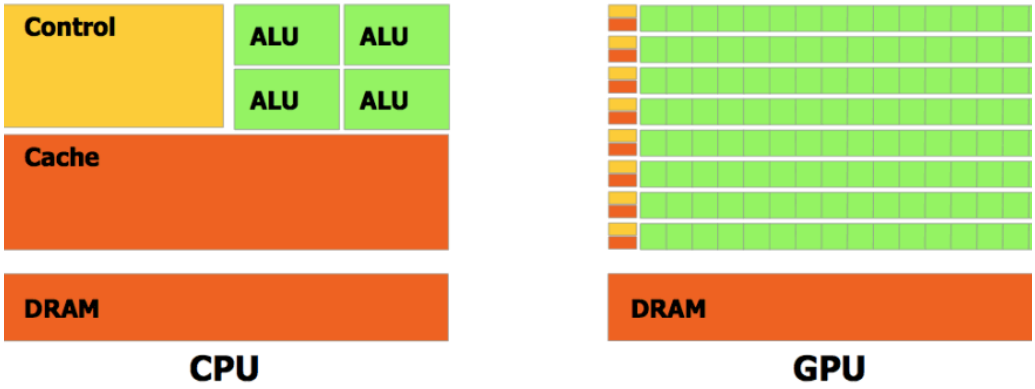

Figura 5 - Diferenças na arquitetura de CPUs e GPUs
Fonte: (KIRK and HUW, 2010).

As principais fabricantes de GPUs na atualidade, NVIDIA e AMD apresentam arquiteturas e capacidade de processamento semelhante em seus modelos. Na figura 6 é ilustrada a arquitetura Kepler desenvolvida pela NVIDIA, baseada em tecnologia de processo de 28 nanômetros.

Figura 6 - Arquitetura de GPU fabricada pela NVIDIA – família Kepler
Fonte: (NVIDIA, 2013).

A arquitetura Kepler é composta por diferentes configurações de *clusters* de processamento gráfico (GPC – *Graphics Processing Clusters*), multiprocessadores de *streaming* (SMX – *Streaming Multiprocessors*) e controladores de memória. Os SMX são o centro da arquitetura Kepler. A maioria das unidades de *hardware* essencial para o

processamento gráfico reside nos SMX. Nos núcleos SMX são computados os cálculos e realizadas a geometria de vértice e sombreamento. Os SMX agem como processadores simples, cada um executando uma ação simultânea, realizando um paralelismo intenso. O que geralmente difere um modelo de GPU de outro são a quantidade desses itens, além da quantidade de memória e frequências de memória e processamento.

A seguir serão apresentadas pela seção 2.6 as principais APIs de programação paralela para CPU e GPU utilizadas nesta pesquisa.

2.6 APIs de programação paralela

Atualmente a computação de alto desempenho dispõe de uma série de APIs para o desenvolvimento de aplicações paralelas. Destacam-se OpenMP para o processamento em CPU e CUDA e OpenACC para processamento em GPU. Essas três APIs serão mais bem apresentadas na sequência.

2.6.1 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, 2013) utilizando o conceito de *threads*, porém sem que o programador tenha que trabalhar diretamente com elas (MATLOFF, 2013). 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. A palavra *Open* presente no nome da API significa que é padrão aberto e está definido por uma especificação de domínio público e MP são as siglas de *Multi Processing* (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, 2013). 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). Esse modelo é ilustrado pela figura 7, onde cada uma das três regiões paralelas tem 3, 4 e 2 *threads*, respectivamente.

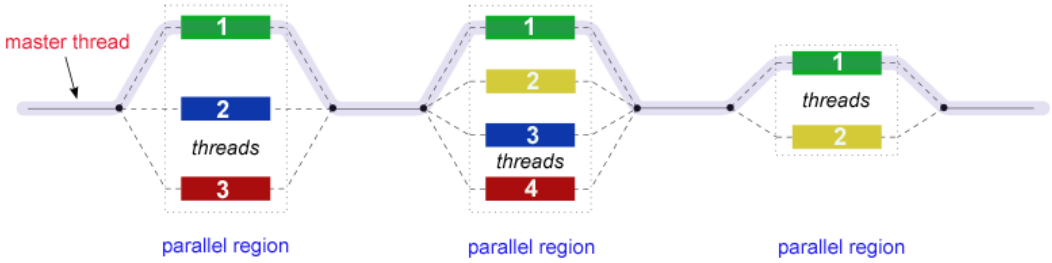

Figura 7 - Modelo *fork-join*
Fonte: (BARNEY, 2013a).

A API OpenMP prove uma série de vantagens na sua utilização. Através da inclusão de diretivas, são mínimas as alterações no código fonte sequencial. Além do mais, possibilita o ajuste dinâmico do número de *threads* com suporte a paralelismo aninhado, apresentando facilidade na compreensão e utilização. As funcionalidades do OpenMP são constituídas através das variáveis de ambiente (*OMP_NOME*), diretivas de compilação *(#pragma omp diretiva [cláusula]*) e bibliotecas de serviço (*omp_serviço*) (SENA; COSTA, 2008).

A figura 8 ilustra a utilização de algumas diretivas de compilação e variáveis de ambiente do OpenMP, além de demonstrar a ocorrência do modelo *fork-join*. Nesse exemplo, na linha 4 tem início o *fork*, onde ocorre a divisão e a execução das *threads*, sendo que cada *thread* possui um identificador privativo (*tid*). Na linha 6 é atribuído o valor do identificador privativo da *thread* e na linha seguinte o resultado é exibido. Entre as linhas 8 a 11 é mostrado o total de *threads* existentes. Essa etapa somente é executada pela *thread master* (quando *tid*

for igual a 0). Na linha 12, o *fork* é encerrado e iniciado o *join* onde as *threads* são sincronizadas.

```
1  #include <omp.h>
2  int main () {
3      int nthreads, tid;
4      #pragma omp parallel private(tid)
5      {
6          tid = omp_get_thread_num();
7          printf("Thread ID = %d\n", tid);
8          if (tid == 0) {
9              nthreads = omp_get_num_threads();
10             printf("Número de Threads = %d\n", nthreads);
11         }
12     }
13     return (0);
14 }
```

Figura 8 - Exemplo OpenMP
Fonte: Adaptação (BARNEY, 2013a).

Já para o processamento em GPU destacam-se as APIs CUDA, OpenCL e OpenACC. CUDA e OpenACC utilizadas neste trabalho serão apresentadas a seguir.

2.6.2 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, 2013). 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. Um aspecto importante da programação CUDA é a gestão de acesso à memória, conforme demonstra a figura 9.

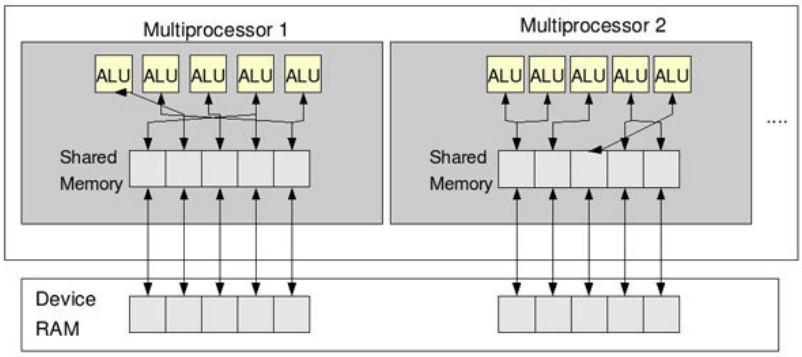

Figura 9 - Acesso à memória
Fonte: (MANAVSKI, 2012).

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. As ALUs podem acessar através de leitura e escrita a memória compartilhada do multiprocessador e a memória RAM (*Random Access Memory*) do dispositivo (MANAVSKI, 2012).

As GPUs atuais suportam até 1024 *threads* em cada bloco. Cada bloco é escalonado em um dos multiprocessadores das GPUs. Um multiprocessador cria, gerencia e executa de modo concorrente todas as *threads* de um bloco, com custo (*overhead*) zero de escalonamento. Quando um multiprocessador encerra o bloco de processamento atual, novos blocos são escalonados para eles (NVIDIA, 2014b). A figura 10 demonstra a estrutura de um *kernel*.

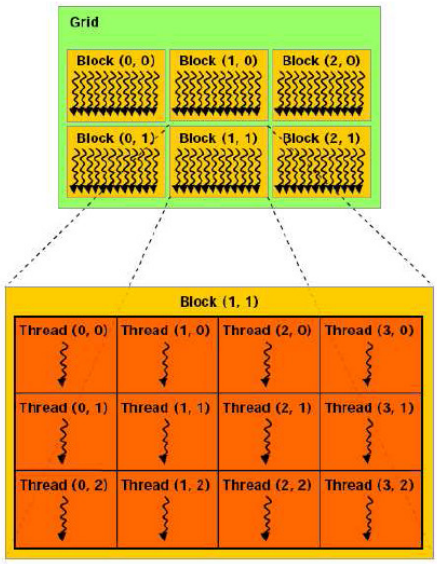

Figura 10 - Estrutura interna do *kernel*
Fonte: (NVIDIA, 2014b).

Um *kernel* nada mais é do que um núcleo composto por blocos de *threads*, onde o processamento no lado da GPU é executado. Os blocos podem ser compostos por uma ou por várias *threads*. A execução de instruções na GPU, na maioria dos casos, tende a apresentar um desempenho maior se comparada a execuções em CPU. Porém, instruções contendo muitos desvios condicionais podem apresentar um baixo desempenho quando executados em GPU. No lado do usuário, não há a necessidade de saber quantos multiprocessadores uma GPU possui, de forma a definir o número de blocos e *threads* por bloco. Essa tarefa é de responsabilidade do sistema que efetua o escalonamento dos blocos nos multiprocessadores da GPU. Para o usuário essa tarefa é completamente transparente (LINCK, 2010).

Um exemplo que demonstra o funcionamento da API CUDA é apresentado pela figura 11. Nela ocorre a soma das matrizes *A* e *B*, sendo o resultado armazenado na matriz *C*.

```
1  __global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
2      int i = threadIdx.x;
3      int j = threadIdx.y;
4      C[i][j] = A[i][j] + B[i][j];
5  }
6  int main() {
7      ...
8      int numBlocks = 1;
9      dim3 threadsPerBlock(N, N);
10     MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
11     ...
12     return(0);
13 }
```

Figura 11 - Exemplo CUDA
Fonte: Adaptação (NVIDIA, 2014b).

Em CUDA, cada execução em GPU deve ser através de um *kernel*. No exemplo anterior, isso ocorre entre as linhas 1 e 5. Nas linhas 8 e 9 são definidas a quantidade de blocos e o número de *threads* por bloco. Por fim, na linha 10 ocorre a invocação do *kernel* *MatAdd* para execução em GPU.

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. Diferente disso, o OpenACC vem para ser um novo padrão para programação paralela para diferentes dispositivos, através da introdução de diretivas muito semelhantes ao OpenMP e com pequenas alterações no código fonte sequencial (OPENACC, 2013). Essa API será mais bem abordada na próxima subseção.

2.6.3 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, 2013).

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, 2013a). 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, 2013).

O modelo de execução alvo do OpenACC são *hosts* (CPUs) 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, 2013).

As diretivas do OpenACC são muito semelhantes as diretivas do OpenMP. O exemplo da figura 12 ilustra sua utilização, onde é realizada a soma dos vetores *a* e *b*, e o resultado é armazenado no vetor *r*.

```
1  #include <stdio.h>
2  void vecaddgpu(float *restrict r, float *a, float *b, int n) {
3       #pragma acc kernels loop copyin(a[0:n], b[0:n]) copyout(r[0:n])
4       for(int i = 0; i < n; ++i)
5            r[i] = a[i] + b[i];
6  }
7  int main() {
8      ...
9      vecaddgpu(r, a, b, n);
10     ...
11     return(0);
12 }
13
```

Figura 12 - Exemplo OpenACC
Fonte: Adaptação (PGI, 2013b).

No exemplo anterior, a diretiva *#pragma acc* da linha 3 informa ao compilador para gerar um *kernel*, alocando e copiando a partir da memória do *host* para a memória da GPU *n* elementos dos vetores *a* e *b* (*copyin(a[0:n], b[0:n])*), antes da execução na GPU. Também aloca *n* elementos no vetor *r* antes da execução na GPU (*copyout(r[0:n])*) e após a execução no dispositivo acelerador, os *n* elementos do vetor *r* são copiados da memória da GPU para a memória do *host*. Esse paralelismo ocorre a partir da chamada da função *vecaddgpu* na linha 9. Observa-se que em relação a um algoritmo sequencial ou paralelo em OpenMP são pequenas as mudanças no código fonte.

O resultado da execução de um algoritmo paralelo pode trazer benefícios no desempenho. Para medir o desempenho de um algoritmo paralelo são utilizadas algumas métricas, que serão melhor exploradas na próxima seção.

2.7 Métricas de desempenho

Os grandes benefícios que a computação paralela traz são apresentar um maior desempenho se comparado a uma execução sequencial e também poder resolver problemas mais complexos, de difícil solução. Mensurar o quanto pode ser mais rápido a execução de um determinado problema com a introdução de *n* processadores ou medir o quanto realmente o algoritmo foi eficiente são tarefas muito importantes quando se trabalha com paralelismo. Para isso, existem algumas métricas de desempenho que medem uma série de fatores gerados pelo paralelismo.

2.7.1 Speedup

Segundo GEBALI (2011), *speedup* é uma métrica que mede o tempo que se leva para concluir uma tarefa em um único processador em relação ao tempo que se leva para completar a mesma tarefa com *p* processadores paralelos. A aceleração de um programa paralelo pode ser obtida por:

$$S(p) = \frac{Tp(1)}{Tp(p)}$$

onde *S(p)* é o *speedup* de *p* processadores paralelos, *Tp(1)* é o tempo de processamento em um único processador e *Tp(p)* é o tempo de execução em *p* processadores.

Alguns exemplos do cálculo do *speedup* podem ser ilustrados pela tabela 1, a seguir.

Tabela 1 - Exemplos de *speedup*

	1 Proc.	2 Proc.	4 Proc.	8 Proc.	16 Proc.
Tempo (ms)	10.000	5.200	2.800	1.600	950
S(p)	1	1,92	3,57	6,25	10,52

Na tabela 1 observa-se que através da introdução de novas unidades de processamento há uma diminuição do tempo de execução do algoritmo. Porém, como ocorre na grande maioria dos casos, o incremento de processadores faz com que a eficiência do *speedup* diminua. Esse fato ocorre principalmente devido a fatores como (RAUBER; RÜNGER, 2010): sincronização, *deadlock*, balanceamento de carga e troca de informações.

Na prática é difícil obter um *speedup* linear uma vez que múltiplos processos ou *threads*, quase que invariavelmente apresentam alguma sobrecarga. Um exemplo disso são os programas que fazem uso de memória compartilhada. Esses programas tendem a ter seções críticas, o que exigirá a utilização de algum mecanismo de exclusão mútua, como por exemplo, o *mutex*. O uso de *mutex* faz com que uma região crítica seja executada de forma sequencial. Nesse sentido, o aumento do número de *threads* em programas com regiões críticas fará com que se aumente o processamento sequencial (PACHECO, 2011).

Outro ponto que pode ser avaliado são os programas que utilizam memória distribuída. Nessa situação ocorre a troca de informações entre os processadores através de rede, o que normalmente é muito mais lento do que o acesso a memória local. O *speedup* tende a diminuir na medida em que se aumentam o número de processos. Com a adição de mais processos, provavelmente será necessário transmitir mais dados através da rede e o desempenho consequentemente diminuirá (PACHECO, 2011).

Não é comum, mas há casos onde a aceleração é superlinear, ou seja, $S(p) > p$. Nesses casos, o tempo de execução em paralelo através de p processadores é maior que p processadores. A razão para esse comportamento muitas vezes está no custo praticamente inexistente de comunicação/sincronização e na possibilidade de todos os dados do programa estarem na memória *cache* (RAUBER; RÜNGER, 2010).

2.7.2 Eficiência

Eficiência é a medida do grau de aproveitamento dos recursos computacionais. Pode ser expressa através de (RAUBER; RÜNGER, 2010):

$$E(p) = \frac{S(p)}{P}$$

onde *E(p)* é a eficiência de *p* processadores, *S(p)* é o *speedup* de *p* processadores e *P* é o número de unidades de processamento. Não havendo um *speedup* superlinear, a eficiência será E(p) <= 1. Para um *speedup* ideal, S(p) = p, a eficiência é E(p) = 1. A tabela 2, a seguir, ilustra alguns exemplos do cálculo da eficiência.

Tabela 2 - Exemplos de eficiência

	1 Proc.	2 Proc.	4 Proc.	8 Proc.	16 Proc.
S(p)	1	1,92	3,57	6,25	10,52
E(p)	1	0,96	0,89	0,78	0,65

Segundo PACHECO (2011) é notório que *speedup* e eficiência são intimamente dependentes do número de processadores. Ocorrendo aumento de *speedup* irá aumentar a eficiência da ocupação do recurso computacional. É possível verificar nos exemplos da tabela 2 que a execução paralela em dois processadores teve 96% da ocupação da capacidade computacional. Em contrapartida, com dezesseis processadores a ocupação foi de apenas 65%.

2.8 Granularidade

A decomposição de um algoritmo em tarefas é geralmente uma atividade complexa e trabalhosa, uma vez que podem existir diferentes formas de decomposição para o mesmo algoritmo. Definir a decomposição das tarefas de forma adequada é uma das principais atividades no desenvolvimento de um algoritmo paralelo (RAUBER; RÜNGER, 2010).

Segundo BARNEY (2013b), a granularidade pode ser definida como a medida entre a quantidade de computação realizada em uma tarefa paralela e a quantidade de comunicação necessária. Ainda para BARNEY (2013b), a granularidade pode ser classificada como fina e grossa. A granularidade fina apresenta muito pouco processamento por *byte* de comunicação.

Caracteriza-se por facilitar o balanceamento de carga, porém implica em uma sobrecarga na comunicação gerando menos oportunidades para um maior desempenho. Para LASTOVETSKY e DONGARRA (2009), quanto mais fina a granularidade de uma aplicação, mais comunicações são envolvidas na execução.

Já na granularidade grossa tem-se muito processamento por *byte* de comunicação. Oferece mais oportunidades para aumentar o desempenho, porém há mais dificuldade para realizar o balanceamento de carga de forma eficiente. A figura 13 exemplifica a classificação apresentada anteriormente. Enquanto que na granularidade fina há muita comunicação por computação, na granularidade grossa há pouca comunicação durante a computação.

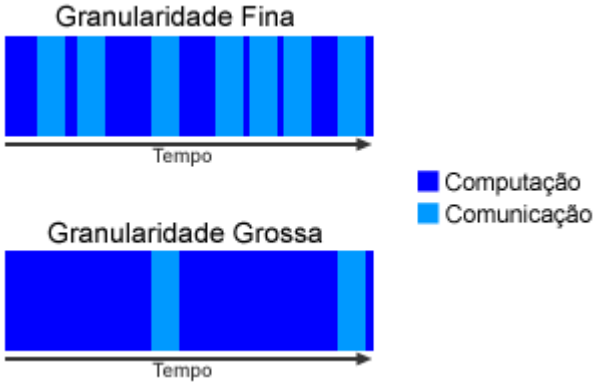

Figura 13 - Classificação da granularidade
Fonte: Adaptação (BARNEY, 2013b).

Sendo assim, quanto mais grossa a granularidade tende a ser melhor o desempenho. Paralelismo de grão fino pode auxiliar na redução de custos devido à facilidade de equilibrar a divisão das tarefas.

A seção 2.9, a seguir, apresenta os trabalhos correlatos como forma de situar a presente dissertação em relação às produções existentes na área.

2.9 Trabalhos correlatos

A busca por maior desempenho computacional fez com que ao longo dos últimos anos surgissem diversos dispositivos com novas características e variadas capacidades de processamento. Como forma de auxiliar no desenvolvimento e na execução de instruções nesses dispositivos, também foram criadas algumas APIs como OpenMP para execução em CPU, CUDA, OpenCL e mais recentemente OpenACC para execução em GPU. Ainda assim, aproveitando a capacidade computacional de CPU e GPU podem-se desenvolver aplicações

híbridas que explorem as capacidades e características de cada modelo. Alguns trabalhos envolvendo paralelismo híbrido para CPU e GPU, bem como que avaliem as APIs CUDA e OpenACC foram examinados durante a elaboração desta dissertação. Destacam-se os estudos de REYES et. al. (2012), MULLA (2011), CHATAIN (2012), LU et al. (2012) e WIENKE et al. (2012). A área hachurada no quadro 1 corresponde à presença do item neste e nos trabalhos pesquisados.

Trabalhos	Execução paralela CPU		Execução paralela GPU					Execução paralela híbrida CPU + GPU			
	OpenMP	MPI	CUDA	hiCUDA	OpenACC	OpenCL	OpenMP Acelerator	OpenMP + CUDA	OpenMP + OpenACC	MPI + CUDA	MPI + OpenMP / CUDA
REYES			■	■	■						
MULLA			■				■				
CHATAIN			■		■						
LU	■	■	■							■	■
WIENKE					■	■					
Presente trabalho	■		■		■			■	■		

Quadro 1 - Trabalhos correlatos

O trabalho de REYES et. al. (2012) intitulado *A Comparative Study of OpenACC Implementations* apresenta uma comparação entre três modelos de programação para GPU baseados em diretivas: hiCUDA, OpenACC (compilado por PGI *Accelerator*) e OpenACC (compilada por *accULL*). Através de um exemplo de multiplicação de matrizes desenvolvido nos três modelos, os autores apresentam as diferenças de programação entre as plataformas e também realizam uma comparação de desempenho entre os três modelos e a versão nativa CUDA, através da utilização de algoritmos do *benchmark* Rodinia. Desse conjunto de problemas, os autores selecionaram os algoritmos HotSpot – simulação térmica utilizada para estimar a temperatura do processador – decomposição de LU e o método Needleman-Wunsch empregado no alinhamento de sequências de DNA.

No algoritmo Hotspot a implementação CUDA nativa foi a que apresentou maior desempenho para todos os tamanhos de problemas. O OpenACC, compilado por PGI *Accelerator,* não apresentou uma boa execução para tamanhos pequenos de problema, enquanto que hiCUDA e *accULL* alcançaram desempenho inferior a abordagem nativa CUDA (entre 40% e 75%). Para o algoritmo de decomposição de LU, em tamanhos pequenos de problemas, o maior desempenho foi conseguido através de OpenMP. Em tamanhos maiores, o maior desempenho foi alcançado pela aplicação nativa CUDA. O desempenho de *accULL* somente foi maior que o desempenho de OpenACC, compilado por PGI *Accelerator*,

em tamanhos pequenos. No algoritmo Needleman-Wunsch o maior desempenho foi através de OpenACC (compilado por PGI *Accelerator*). hiCUDA para tamanhos maiores de problemas apresentou desempenho semelhante ao OpenMP, enquanto que o *accULL* dentre os modelos utilizados teve o pior desempenho.

No trabalho *Directives Based Programming of GPU Accelerated Systems* de autoria de MULLA (2011) foram analisados os desempenhos das APIs CUDA e OpenMP *Accelerator* através de um algoritmo de dinâmica molecular, sendo utilizadas 128, 256 e 512 *threads* por bloco nos testes. Também foram calculados os tempos de transferência dos dados entre CPU e GPU, onde foi constatado que os tempos para cópia de informações em OpenMP *Accelerator* e CUDA é muito semelhante. Porém, esse tempo é um fator que deve ser levado em consideração, pois em aplicações paralelas com CPU ele não existe. Segundo os testes realizados, a aplicação desenvolvida com o OpenMP *Accelerator* resultou em um maior tempo de execução se comparada à implementação em CUDA. Para MULLA (2011), a questão do desempenho de CUDA está atrelada à homogeneidade com a arquitetura NVIDIA. O OpenMP *Accelerator*, por ser independente de arquitetura e ainda ser um modelo em desenvolvimento, apresentou-se mais lento.

Em CHATAIN (2012), no trabalho *Hybrid Parallel Programming - Evaluation of OpenACC* são identificadas as principais vantagens e limitações do OpenACC, além de analisar o seu desempenho. Para isso, o autor testou implementações dos algoritmos de matriz transposta, produto escalar e multiplicação de matrizes, com versões em CUDA e OpenACC. Os resultados das execuções dos três algoritmos mostraram que o desempenho do OpenACC frente a CUDA é inferior. Porém, se o desempenho do OpenACC for comparado a programas sequenciais executados em CPU há melhoras significativas. Em comparação a implementações CUDA menos otimizadas, OpenACC apresenta um desempenho semelhante, porém com um modelo de programação muito mais simples. Como limitações encontradas o autor aponta para o desempenho do OpenACC que foi abaixo do obtido por CUDA e a perda do controle sobre o código gerado, não possibilitando ao programador otimizar a programação para GPU. Além disso, CHATAIN (2012) relata que o OpenACC ainda representa um modelo mais limitado em comparação à CUDA, possibilitando paralelizar de forma eficiente um conjunto menor de problemas.

No trabalho de LU et al. (2012) intitulado *Performance evaluation of hybrid programming patterns for large CPU/GPU heterogeneous clusters* são realizados testes em ambientes híbridos utilizando MPI, OpenMP e CUDA, fazendo a combinação MPI + CUDA e MPI + OpenMP/CUDA. Para testar o desempenho do ambiente híbrido, os autores

utilizaram um código legado de dinâmica de fluídos computacional contendo três *kernels*, chamados EP, CG e MG. Após a análise das execuções dos modelos híbridos LU et al. (2012) obtiveram os seguintes resultados. Para aplicações massivamente paralelas o padrão MPI + OpenMP/CUDA é mais adequado em virtude da melhor possibilidade de distribuição da carga de trabalho entre CPU e GPU, de acordo com as capacidades computacionais. Em aplicativos onde a maior parte da carga de trabalho foi enviada para GPU, o padrão de programação que obteve melhor resultado é o MPI + CUDA.

No trabalho de WIENKE et al. (2012) realizou-se uma comparação de desempenho entre OpenACC e OpenCL através de duas aplicações do mundo real das áreas de engenharia e medicina. O trabalho intitulado *OpenACC - First Experiences with Real-World Applications* se propôs a analisar o desempenho do código produzido, bem como uma avaliação de produtividade. A primeira aplicação desenvolvida é um *software* de simulação 3D para a área da engenharia, utilizada no processo de corte de engrenagens para a indústria automotiva. A segunda aplicação vem do campo da medicina, mais precisamente da Magnetoencefalografia (MEG). O desempenho da aplicação OpenACC da área de engenharia foi de 80% da melhor execução OpenCL, entretanto a implementação em OpenACC ainda está incompleta. Em contrapartida, o desempenho da implementação OpenACC da área médica foi de apenas 40% da melhor execução da aplicação em OpenCL. Os autores acreditam que essa queda de desempenho seja devido às sincronizações para evitar as condições de corrida. De acordo com WIENKE et al. (2012), o OpenACC está em uma fase de crescimento, pois foi lançado recentemente, ainda necessitando de ajustes. Porém o OpenACC deve ter uma aceitação muito grande, pois vem para ser uma padronização de diretivas para dispositivos aceleradores, assim como o OpenMP é para CPU.

2.10 Considerações

Este capítulo apresentou a fundamentação teórica para melhor compreensão dos assuntos abordados neste trabalho. Foram abordados os conceitos de paralelismo e programação híbrida, apresentadas as APIs de programação paralela OpenMP, CUDA e OpenACC, além de algumas métricas de desempenho utilizadas para medir o resultado da execução dos algoritmos propostos neste. Também foram apresentados os trabalhos correlatos como forma de situar a presente dissertação em relação às produções existentes na área.

O terceiro capítulo irá abordar os algoritmos selecionados neste trabalho, o seu funcionamento e as suas principais características.

3 ALGORITMOS SELECIONADOS

Neste trabalho foram selecionados três algoritmos do *benchmark* Rodinia: RNG, Hotspot e SRAD. A escolha pelo *benchmark* Rodinia se deu por esse apresentar um bom conjunto de algoritmos, com resolução de problemas de diversas áreas, tais como, médica, matemática, física, processamento de imagens, pesquisa e ordenação e por já disponibilizar as versões em OpenMP e CUDA. As versões dos algoritmos nessas APIs apenas necessitaram ser compiladas e executadas, agilizando o desenvolvimento e garantindo a exatidão do código fonte. Os algoritmos selecionados serão mais bem detalhados nas próximas seções deste capítulo.

3.1 RNG

RNG ou *Random Number Generator* é um método utilizado para gerar uma sequência de números aleatórios, onde a produção de um resultado imprevisível é desejável. Pode ser utilizado em diversas aplicações, tais como, sorteios, jogos, amostragem, simulação e geração de chaves de criptografia (RANDOM, 2013), entre outras. A seleção desse algoritmo se deu por apresentar um número grande de iterações e pela necessidade de transferir uma grande quantidade de dados entre CPU e GPU. Através da transferência de um volume maior de dados, procurou-se com esse algoritmo verificar se o tempo gasto nesse processo acaba influenciando no desempenho da execução na GPU.

No *benchmark* Rodinia, versão 2.4, o algoritmo RNG é baseado em duas funções: *randu* e *randn*. A função *randu* utiliza uma função de *Linear Congruential Generator* (LCG) para geração dos números. São parâmetros dessa função um vetor e o índice selecionado desse vetor. Sua saída é um número uniformemente distribuído como ilustra a figura 14, onde as variáveis *A* e *C* são valores para LCG (RODINIA, 2013).

```
1 double randu(int *seed, int index){
2     int num = A * seed[index] + C;
3      seed[index] = num % INT_MAX;
4      return fabs(seed[index] / ((double) / INT_MAX));
5 }
```

Figura 14 - Função *randu* do algoritmo RNG
Fonte: Adaptação RODINIA (2013).

Já função *randn* utiliza a transformação de Box-Muller para a distribuição uniforme dos números gerados (RODINIA, 2013), conforme apresenta a figura 15.

```
1 double randn(int * seed, int index){
2       double u = randu(seed, index);
3       double v = randu(seed, index);
4       double cosine = cos(2 * PI * v);
5       double rt = - 2 * log(u);
6       return sqrt(rt) * cosine;
7 }
```

Figura 15 - Função *randn* do algoritmo RNG
Fonte: Adaptação RODINIA (2013).

Os parâmetros da função *randn* são um vetor e o índice do elemento que se irá trabalhar. Sua saída é um número aleatoriamente gerado. Os vetores utilizados nas funções do algoritmo RNG são inicializados com valores oriundos da função *time*.

```
1 for (i = 0; i < size; i++){
2       arrayX += 1  + 5 * randu(seed, i);
3       arrayY += -2 + 2 * randn(seed, i);
4 }
```

Figura 16 - Geração dos números no algoritmo RNG
Fonte: Adaptação RODINIA (2013).

Para gerar os números aleatórios, as funções *randu* e *randn* são executadas dentro de um laço de repetição, até que a quantidade de números desejada seja satisfeita. A cada repetição são gerados dois novos elementos, um através de cada função e os seus resultados são atribuídos aos vetores *arrayX* e *arrayY*, respectivamente, como ilustra a figura 16.

A próxima seção detalha o algoritmo Hotspot, também selecionado neste trabalho.

3.2 Hotspot

Hotspot é um algoritmo de simulação térmica utilizado para estimar a temperatura do processador, baseado em uma planta arquitetônica e em medições de potência. A simulação térmica de forma interativa resolve uma série de equações diferenciais por bloco, onde cada célula de saída representa o valor médio da temperatura da área correspondente do *chip* (RODINIA, 2013). O algoritmo Hotspot do *benchmark* Rodinia, versão 2.4, implementa um *kernel* transiente de simulação térmica. O algoritmo Hotspot consiste de duas rotinas, onde a primeira converte as equações diferenciais de transferência de calor para equações de diferenças, além de executar a segunda rotina, que tem por função a solução das equações de diferenças por um intervalo de tempo (RODINIA, 2013). A escolha pelo algoritmo Hotspot se deu por ele apresentar um grande número de repetições ao solucionar as equações de diferenças por um intervalo de tempo e por apresentar uma pequena quantidade de dados a ser transferida entre CPU e GPU. Além disso, através de testes preliminares na GPU percebeu-se

que o tempo de execução no dispositivo gráfico era muito pequeno em relação à mesma execução na CPU. Com isso, a escolha do algoritmo Hotspot também objetivou verificar se os algoritmos híbridos conseguem superar o desempenho de um algoritmo somente em GPU através de um pequeno tempo de execução.

Como entradas principais o algoritmo HotSpot recebe dois arquivos, onde o primeiro contém os valores da temperatura inicial e o segundo armazena os valores da potência dissipada, ambos para cada célula. O *benchmark* disponibiliza três conjuntos de arquivos de dados para entrada: 4.096, 262.144 e 1.048.576 registros cada, sendo esse último utilizado nos testes, em virtude do seu processamento durar mais tempo. A figura 17 ilustra parte do algoritmo Hotspot onde a função *compute_tran_temp* realiza a conversão das equações diferenciais de transferência de calor para equações de diferenças, enquanto que a função *single_iteration* soluciona as equações diferenças por um intervalo de tempo.

```
1  void compute_tran_temp(double  *result,
2                         int      num_iterations,
3                         double   *temperature,
4                         double   *power,
5                         int      row,
6                         int      col)  {
7      ...
8      /* Realiza a conversão das equações diferenciais
9      de transferência de calor para equações de diferenças*/
10     ...
11
12     for (int i = 0; i < num_iterations ; i++){
13         /* Solução das equações de diferença por intervalo de tempo */
14         single_iteration(result, temperature, power, row, col, ...);
15     }
16 }
```

Figura 17 - Parte do algoritmo Hotspot
Fonte: Adaptação RODINIA (2013).

Para finalizar a apresentação dos algoritmos selecionados, será detalhado pela seção 3.3 o algoritmo SRAD.

3.3 SRAD

SRAD ou *Speckle Reducing Anisotropic Diffusion* é um método de difusão para aplicação em imagens de ultrassom baseados em equações diferenciais parciais (EDP). Esse método é utilizado para remover ou suavizar manchas sem destruir funções importantes da imagem. A versão do *benchmark* Rodinia utilizada neste trabalho consiste de várias etapas, entre elas extração da imagem, iterações contínuas sobre a imagem para remoção ou

suavização da mancha e geração de uma nova imagem sem a possível mancha (RODINIA, 2013). A seleção desse algoritmo foi em virtude da importância que o seu resultado pode representar nos dias atuais para a área médica, além de apresentar uma solução rápida e precisa.

A estruturação do algoritmo SRAD é a seguinte: após a leitura dos parâmetros informados na execução, o algoritmo realiza o carregamento da imagem, onde os seus pontos são transferidos para uma estrutura do tipo vetor, com tamanho variando de acordo com as dimensões imagem. Na sequência é realizado o redimensionamento da imagem de acordo com o tamanho informado nos parâmetros do algoritmo. Alguns vetores são alocados dinamicamente, contendo informações das coordenadas da imagem e da mancha. O paralelismo ocorre no momento que é iniciado o processo de remoção da mancha. Nessa etapa ocorre uma repetição, de acordo com o parâmetro *niter* informado, conforme ilustra a figura 18, linha 4, onde para cada execução são realizados cálculos como derivadas direcionais (entre as linhas 11 e 20) e coeficiente de difusão (entre as linhas 23 e 28). Após encerrar a repetição uma nova imagem é salva.

```
1  /* Antes da repetição, realiza-se a alocação dinâmica de memória e a
2     população inicial dos dados */
3
4  for (iter = 0; iter < niter; iter++) {
5      for (int i = 0 ; i < rows ; i++) {
6          for (int j = 0; j < cols; j++) {
7             k = i * cols + j;
8             Jc = J[k];
9
10            /* Derivadas direcionais */
11            dN[k] = J[iN[i] * cols + j] - Jc;
12            dS[k] = J[iS[i] * cols + j] - Jc;
13            dW[k] = J[i * cols + jW[j]] - Jc;
14            dE[k] = J[i * cols + jE[j]] - Jc;
15            G2 = (dN[k]*dN[k] + dS[k]*dS[k]
16                  + dW[k]*dW[k] + dE[k]*dE[k]) / (Jc*Jc);
17            L = (dN[k] + dS[k] + dW[k] + dE[k]) / Jc;
18            num  = (0.5*G2) - ((1.0/16.0)*(L*L));
19            den  = 1 + (.25*L);
20            qsqr = num/(den*den);
21
22            /* Coeficiente de difusão */
23            den = (qsqr-q0sqr) / (q0sqr * (1+q0sqr));
24            c[k] = 1.0 / (1.0+den);
25
26            /* Saturação do coeficiente de difusão */
27            if (c[k] < 0) {c[k] = 0;}
28            else if (c[k] > 1) {c[k] = 1;}
29         }
30     }
31 }
```

Figura 18 - Parte do algoritmo SRAD
Fonte: Adaptação RODINIA (2013).

Como entradas o algoritmo SRAD requer: dimensão da imagem (número de linhas e colunas), posições da mancha (X1, X2, Y1 e Y2), número de *threads* e número de iterações realizadas.

A seguir, para finalizar o capítulo 3 são apresentadas as considerações finais deste.

3.4 Considerações

Este capítulo objetivou explicar as principais características, o funcionamento e os motivos que levaram a escolha dos algoritmos RNG, Hotstpot e SRAD para este trabalho. O quarto capítulo irá abordar a metodologia adotada nos testes, a forma de paralelização híbrida proposta para os algoritmos e o ambiente de execução.

4 METODOLOGIA

Por meio da intensa capacidade de processamento e paralelismo que CPU e GPU disponibilizam através de suas unidades de processamento, chamadas de *cores*, muitas aplicações podem ser executadas em menos tempo, ou até mesmo problemas muito complexos podem ser resolvidos. Com a utilização desses dois dispositivos em conjunto, aliados a APIs de programação paralela é possível dividir um problema e cada parte desse problema ser executado em CPU e GPU simultaneamente. A partir disso, ainda é possível realizar o balanceamento de carga de trabalho entre CPU e GPU, enviando uma determinada quantidade de processamento para cada dispositivo ou uma tarefa em que um ou outro execute mais rapidamente. Essa distribuição pode, em muitos casos, gerar boas possibilidades de aumento de desempenho (LINK, 2010).

Atualmente encontram-se disponíveis diversas APIs de programação paralela, tanto para CPU como para GPU. Para CPU destacam-se OpenMP e *Pthreads*. Já para GPU tem-se CUDA, OpenCL e mais recentemente OpenACC, essa com diretivas muito próximas ao OpenMP (OPENACC, 2013). Sabe-se que o OpenACC introduz uma série de facilidades e produtividade no desenvolvimento de aplicações para GPU, se comparadas a CUDA e OpenCL. Porém, não se conhece se esses fatores também refletem positivamente no desempenho de aplicações híbridas com OpenMP se comparadas à utilização de OpenMP + CUDA.

Dessa forma, este trabalho testou os algoritmos paralelos RNG, Hotspot e SRAD do *benchmark* Rodinia e programou-os de forma híbrida em CPU e GPU, através das APIs OpenMP para CPU e CUDA e OpenACC para GPU. Baseadas nessas versões OpenMP e CUDA, foram desenvolvidos os algoritmos na forma sequencial, paralela para GPU em OpenACC e híbridos utilizando OpenMP + CUDA e OpenMP + OpenACC, onde parte da execução é realizada em CPU e parte em GPU. A configuração do percentual de trabalho processado por CPU e GPU é realizada através de parâmetros informados na execução do algoritmo.

Foram analisados os tempos de execução de cada versão dos algoritmos, calculado o desvio padrão e o percentual de confiança das amostras, como uma forma de validar a quantidade de execuções necessárias sob cada versão e que apresente um índice de confiabilidade das informações que satisfaça esta pesquisa. Também foram calculados os *speedups* das versões OpenMP, CUDA, OpenACC e híbrida OpenMP + CUDA e OpenMP +

OpenACC, bem como as eficiências das versões híbridas. Nas versões híbridas foram atribuídas diferentes variações de cargas de trabalho para CPU e GPU, de forma a verificar qual foi a divisão de execução entre CPU e GPU que maior desempenho apresentou. Além disso, foram analisadas as divisões do tempo de execução em CPU e GPU dos algoritmos, com a finalidade de verificar se os algoritmos executam paralelamente e se o desempenho se manteve em relação à execução paralela em um único dispositivo. Foram também calculados os tempos de transferência de dados entre os dispositivos e verificadas as limitações que podem influenciar no desempenho ou no desenvolvimento nos dois modelos híbridos propostos.

A seguir serão detalhadas a metodologia adotada nos testes, o ambiente de execução, os algoritmos paralelos selecionados e implementados de forma híbrida em CPU e GPU, bem como a forma de paralelização híbrida adotada em cada algoritmo.

4.1 Metodologia adotada na execução dos algoritmos

Para a realização dos testes dos algoritmos híbridos, algumas medidas foram adotadas, com a finalidade de reduzir possíveis interferências. A primeira delas foi alterar o dispositivo padrão de processamento gráfico, passando a utilizar o *device onboard.* Isso foi adotado para que a GPU ficasse inteiramente disponível para o processamento dos algoritmos. O segundo procedimento foi desabilitar o controle gráfico do sistema operacional, parando o serviço *lightdm* do sistema operacional Ubuntu.

Cada versão dos algoritmos, sequencial, em uma única API ou híbrida foi testada 30 vezes e a média dessas execuções foi o parâmetro de tempo utilizado. Esse número foi escolhido por apresentar em todos os testes, intervalos de confiança superiores a 90%. Foram calculados ainda o desvio padrão, o *speedup* e a eficiência das versões utilizadas. Para a medição de tempo foram utilizadas as funções da biblioteca *time.h*, exceto para os algoritmos que utilizam CUDA, onde foram criados eventos para computar o tempo de inicialização, transferência de dados, processamento e finalização dos algoritmos. Ainda foram utilizados o *nvprof* (NVIDIA, 2014a) para os algoritmos que utilizavam CUDA e o parâmetro de compilação *–ta=nvidia,time* no compilador *pgcc* para os algoritmos que utilizavam OpenACC, como forma de validar os tempos de processamento calculados pelas funções da biblioteca *time.h* e pelos eventos CUDA, além de obter o *throughput*. O tempo de cada versão é referente à execução de todo o algoritmo. Para os algoritmos executados em GPU e para os

algoritmos híbridos, os tempos de execução incluíram a cópia dos dados da CPU para a GPU, o processamento dos dados e a cópia dos dados da GPU para a CPU.

De forma a manter o equilíbrio de processamento entre CPU e GPU, o tempo de execução nesses dispositivos deve ser praticamente o mesmo, impedindo que um ou outro fique ocioso. As tarefas executadas por CPU e GPU foram as mesmas, para uma iteração. Os percentuais de execução em CPU e GPU (número de iterações) foram de acordo com as características de cada algoritmo e de testes iniciais, onde foi possível verificar as melhores distribuições de trabalho para cada dispositivo.

Para o cálculo da eficiência dos algoritmos híbridos OpenMP + CUDA e OpenMP + OpenACC foi proposta uma eficiência global, calculada da seguinte forma. Em primeiro momento foram calculadas as eficiências das parcelas de processamento de CPU e GPU, sendo que a eficiência de cada dispositivo foi multiplicada pela quantidade de trabalho executada pelo dispositivo e o resultado foi dividido pela quantidade total de trabalho, conforme fórmula a seguir:

$$E = \frac{(Ec * Tc) + (Eg * Tg)}{Tc + Tg}$$

onde E é a eficiência global, Ec e Eg são as eficiências das parcelas de processamento da CPU e GPU e Tc e Tg representam o trabalho realizado pela CPU e GPU, respectivamente.

A seguir, a seção 4.2 detalha a forma de divisão das instruções entre CPU e GPU adotada nos algoritmos híbridos.

4.2 Forma de divisão híbrida adotada

A divisão da execução entre CPU e GPU nos algoritmos híbridos se deu de acordo com a figura 19. Em primeiro momento foi criada uma região paralela OpenMP (linha 1) e na sequência foi introduzida uma região *sections* (linha 3), que especifica um bloco de seções onde diferentes *threads* podem executar diferentes partes de um programa paralelo. Dentro dessa região foram criadas duas seções, sendo a primeira, entre as linhas 5 e 8 e a segunda entre as linhas 10 e 13 da figura 19.

A primeira seção dispõe de uma chamada a uma função que executa instruções paralelas em GPU e a segunda invoca uma função que executa instruções paralelas na CPU.

Dessa forma, o início da execução paralela em CPU e GPU sempre foi muito próximo, conforme é detalhado na divisão da execução entre os dispositivos no capítulo quatro.

```
1  #pragma omp parallel
2  {
3    #pragma omp sections
4    {
5      #pragma omp section
6      {
7       //Função que executa o percentual destinado a GPU de forma paralela
8      }
9
10     #pragma omp section
11     {
12       //Função que executa o percentual destinado a CPU de forma paralela
13     }
14   }
15 }
```

Figura 19 - Forma de divisão adotada para a execução híbrida utilizando as APIs OpenMP, CUDA e OpenACC

Na sequência serão apresentadas as estruturas híbridas dos algoritmos do *benchmark* Rodinia selecionados para este trabalho.

4.3 Estrutura do algoritmo híbrido RNG

A estrutura dos algoritmos híbridos OpenMP + CUDA e OpenMP + OpenACC para o cálculo do RNG segue a seguinte ordem. Em primeiro momento, de forma sequencial foram alocadas na memória da CPU as variáveis necessárias para o processamento do OpenMP e realizada a população dos dados iniciais ao processo. Foi criada uma região paralela (linha 1) e nela mais duas seções através da utilização de diretivas OpenMP (com início nas linhas 5 e 25), conforme ilustra a figura 20. Na função responsável pelo processamento na GPU são alocadas as variáveis na memória do dispositivo e os dados necessários são copiados da memória da CPU para a memória da GPU. O processamento em GPU é realizado (linha 15 – exposto de forma genérica para CUDA e OpenACC) e quando encerrado, o resultado é transferido da memória da GPU para a memória da CPU. Ao término do processamento em GPU as variáveis utilizadas são removidas da memória da GPU.

Na função responsável pela execução paralela em CPU (entre as linhas 25 a 40) são alocadas as *threads* para realizar a parte do cálculo que cabe a CPU (linha 30). É realizado o processamento paralelo na CPU (entre as linhas 32 e 37), sendo os dados atribuídos aos vetores *arrayX* e *arrayY*. Ao término do processamento paralelo na CPU, as variáveis alocadas na memória são removidas e o algoritmo RNG é encerrado. Na execução da função

RNG_GPU (linha 15) ocorre a mesma quantidade de cálculos apresentadas entre as linhas 34 e 37 da figura 20, porém com a repetição desses variando conforme a divisão do processamento entre CPU e GPU. A divisão híbrida entre CPU e GPU ocorre através da divisão da quantidade de números gerados nos dispositivos.

```
1  #pragma omp parallel
2  {
3    #pragma omp sections
4    {
5      #pragma omp section
6      {
7         /* Inicializa o temporizador do trabalho da GPU */
8
9         /* Inicializa o temporizador da transferência de dados CPU -> GPU */
10        /* Realiza a alocação dinâmica de memória na GPU */
11        /* Transfere os dados CPU -> GPU */
12        /* Finaliza o temporizador da transferência de dados CPU -> GPU */
13
14        /* Realiza o processamento na GPU
15        RNG_GPU(arrayX_GPU, array_GPU, seed_GPU, Size_GPU);
16
17        /* Inicializa o temporizador da transferência de dados GPU -> CPU */
18        /* Transfere os dados GPU -> CPU */
19        /* Remove as variáveis alocadas na memória da GPU */
20        /* Finaliza o temporizador da transferência de dados GPU -> CPU */
21
22        /* Finaliza o temporizador do trabalho da GPU */
23     }
24
25    #pragma omp section
26    {
27       /* Inicializa o temporizador do trabalho da CPU */
28
29       /* Define o número de threads */
30       omp_set_num_threads(NThreads);
31
32       #pragma omp parallel for shared(arrayX, arrayY, Size_CPU, seed)
33                                private(x)
34       for(x = 0; x < Size_CPU; x++) {
35              arrayX[x] += 1  + 5 * randu(seed, x);
36              arrayY[x] += -2 + 2 * randn(seed, x);
37       }
38
39       /* Finaliza o temporizador do trabalho da CPU */
40    }
41  }
42 }
```

Figura 20 - Estrutura do algoritmo híbrido RNG

A seguir, a seção 4.5 irá detalhar a estrutura do algoritmo híbrido Hotspot.

4.4 Estrutura do algoritmo híbrido Hotspot

No algoritmo Hostpot, nas versões OpenMP + CUDA e OpenMP + OpenACC antes de iniciar o processamento paralelo em CPU e GPU, de forma sequencial ocorre a alocação

dinâmica de memória e a população inicial dos dados das variáveis utilizadas no processamento deste algoritmo. A estrutura híbrida entre CPU e GPU do algoritmo híbrido Hotspot obedece a estrutura apresentada na seção 4.3. Na seção que compete a execução da GPU são alocadas as variáveis necessárias para o cálculo no dispositivo, como apresenta a figura 21. Os dados da memória da CPU são transferidos para a memória da GPU, nas variáveis alocadas anteriormente. A função *compute_tran_temp_GPU* (linha 14) é executada, dando início ao processamento na GPU. Ao término do processamento no dispositivo gráfico, os dados na memória da GPU são transferidos para a memória da CPU e as variáveis utilizadas na GPU são removidas da memória do dispositivo. Na seção que compete ao processamento paralelo na CPU é definido o número de *threads* (linha 29) que irá dividir a parcela que cabe a CPU no cálculo. Após isso, é iniciado o processamento na CPU através da execução da função *compute_tran_temp_CPU* (linha 31). Ao encerrar o processamento na CPU, de forma sequencial as variáveis são removidas da memória e o algoritmo é encerrado.

```
1  #pragma omp parallel
2  {
3    #pragma omp sections
4    {
5      #pragma omp section
6      {
7         /* Inicializa o temporizador do trabalho da GPU */
8
9         /* Inicializa o temporizador da transferência de dados CPU -> GPU */
10        /* Transfere os dados CPU -> GPU */
11        /* Finaliza o temporizador da transferência de dados CPU -> GPU */
12
13        /* Realiza o processamento na GPU
14        compute_tran_temp_GPU(result_GPU, sim_time_GPU, temp_GPU, power_GPU,
15                              grid_rows, grid_cols);
16
17        /* Inicializa o temporizador da transferência de dados GPU -> CPU */
18        /* Transfere os dados GPU -> CPU */
19        /* Finaliza o temporizador da transferência de dados GPU -> CPU */
20
21        /* Finaliza o temporizador do trabalho da GPU */
22     }
23
24     #pragma omp section
25     {
26        /* Inicializa o temporizador do trabalho da CPU */
27
28        /* Define o número de threads */
29        omp_set_num_threads(NThreads);
30
31        compute_tran_temp_CPU(result_CPU,sim_time_CPU, temp_CPU, power_CPU,
32                              grid_rows, grid_cols);
33
34        /* Finaliza o temporizador do trabalho da CPU */
35     }
36   }
37 }
```

Figura 21 - Estrutura do algoritmo híbrido Hotspot

No algoritmo Hotspot a hibridez se deu através da divisão do número de iterações que o algoritmo é submetido. Isso foi adotado, uma vez que o algoritmo Hotspot, em sua essência faz o uso de uma grande série de repetições. Dessa forma, o número de iterações em CPU e GPU refere-se ao percentual de carga de trabalho estabelecido para cada dispositivo. As instruções em CPU e GPU são as mesmas, diferenciando apenas na quantidade de vezes que essas instruções são executadas em cada dispositivo. As funções *compute_tran_temp_GPU* (linha 14) e *compute_tran_temp_CPU* (linha 31) tem as instruções como apresentadas pela figura 17 na seção 3.2.

Para finalizar as apresentações das estruturas dos algoritmos híbridos, a seção 4.6 expõe a organização do algoritmo híbrido SRAD.

4.5 Estrutura do algoritmo híbrido SRAD

De forma semelhante ao algoritmo Hostpot, no algoritmo RNG, de forma sequencial são alocadas na memória da CPU as variáveis utilizadas no processamento do algoritmo e realizada a atribuição inicial dos dados. A divisão híbrida entre CPU e GPU do algoritmo híbrido SRAD compreende na primeira seção da execução na GPU, onde são alocadas as variáveis necessárias para o cálculo no dispositivo gráfico e realizada a transferência dos dados entre CPU e GPU, como ilustra a figura 22. Após a transferência dos dados é iniciado o processamento na GPU, através da chamada da função *srad_gpu* (linha 14). Quando essa função é encerrada, os dados na memória da GPU são transferidos para a memória da CPU e as variáveis utilizadas na GPU são removidas da memória do dispositivo.

No processamento paralelo na CPU é definido o número de *threads* (linha 28) que irá dividir a parcela que cabe a CPU no cálculo. Após isso, é iniciado o processamento na CPU através da execução da função *srad_cpu* (linha 30). Ao encerrar o processamento na CPU, de forma sequencial as variáveis são removidas da memória e o algoritmo é encerrado. A divisão híbrida ocorreu através da partilha do número de iterações sobre a imagem. Sendo assim, o número de iterações em CPU e GPU refere-se ao percentual de carga de trabalho estabelecido para cada dispositivo. As instruções em CPU e GPU são as mesmas, diferenciando apenas a quantidade de vezes que essas instruções são executadas em cada dispositivo. As instruções das funções *srad_gpu* (linha 14) e *srad_gpu* (linha 30) foram apresentadas pela figura 18 na seção 3.3.

```
1  #pragma omp parallel
2  {
3    #pragma omp sections
4    {
5      #pragma omp section
6      {
7         /* Inicializa o temporizador do trabalho da GPU */
8
9         /* Inicializa o temporizador da transferência de dados CPU -> GPU */
10        /* Transfere os dados CPU -> GPU */
11        /* Finaliza o temporizador da transferência de dados CPU -> GPU */
12
13        /* Realiza o processamento na GPU
14        srad_gpu(rows_gpu, cols_gpu, ..., NumIteracoes_gpu);
15
16        /* Inicializa o temporizador da transferência de dados GPU -> CPU */
17        /* Transfere os dados GPU -> CPU */
18        /* Finaliza o temporizador da transferência de dados GPU -> CPU */
19
20        /* Finaliza o temporizador do trabalho da GPU */
21     }
22
23    #pragma omp section
24    {
25       /* Inicializa o temporizador do trabalho da CPU */
26
27       /* Define o número de threads */
28       omp_set_num_threads(NThreads);
29
30       srad_cpu(rows_cpu, cols_cpu, ..., NumIteracoes_cpu);
36
37       /* Finaliza o temporizador do trabalho da CPU */
38    }
39  }
40 }
```

Figura 22 - Estrutura do algoritmo híbrido SRAD

A seção 4.6, a seguir, detalhará o ambiente de execução utilizado neste trabalho.

4.6 Ambiente de execução utilizado

O ambiente de execução utilizado neste trabalho foi o seguinte:

a) Hardware

- CPU: Intel Core i7-3632QM – 2.20 GHz – 8 *threads* – 6 MB *cache*
- Memória: 6 GB SDRAM DDR3 – 1600 MHz
- GPU: NVIDIA GeForce GT630M – 96 núcleos – *clock* 950 MHz – memória: 1 GB DDR3

b) Software

- Sistema Operacional Ubuntu 12.04 – 64 bits
- OpenMP versão 3.0
- CUDA versão 5.5

- OpenACC versão 2.0
- Compilador GCC versão 4.6.3
- Compilador NVCC versão 5.5
- Compilador PGCC versão 14.1

c) Parâmetros de compilação utilizados

- Algoritmo sequencial: gcc –o saída entrada.c
- Algoritmo OpenMP: gcc –o saída entrada.c –fopenmp
- Algoritmo CUDA: nvcc –o saída entrada.c
- Algoritmo OpenACC: pgcc –o saída entrada.c –Minfo –fast –acc –ta=nvidia,time
- Algoritmo híbrido OpenMP + CUDA: nvcc –Xcompiler –fopenmp –lgomp –o saída entrada.cu
- Algoritmo híbrido Híbrido OpenMP + OpenACC: pgcc –o saída entrada.c –Minfo –fast –acc –ta=nvidia,time –mp

Para finalizar este capítulo, serão apresentadas as considerações finais.

4.7 Considerações

Este capítulo objetivou explicar o experimento realizado e a metodologia adotada nos testes. Além disso, foram apresentadas para os três algoritmos selecionados a forma de divisão híbrida nos modelos OpenMP + CUDA e OpenMP + OpenACC, propostos neste. Para finalizar foi detalhado o ambiente de execução dos algoritmos propostos no trabalho.

No capítulo cinco serão apresentados os resultados das análises dos algoritmos RNG, Hotspot e SRAD, detalhando principalmente o desempenho, a eficiência, a divisão da execução entre CPU e GPU e o tempo de transferência de dados entre os dispositivos.

5 ANÁLISE DOS RESULTADOS

Neste capítulo são detalhadas as análises realizadas através das execuções dos algoritmos RNG, Hotspot e SRAD.

5.1 RNG

Os resultados das diferentes versões/configurações do algoritmo RNG, seja ele executado em CPU, GPU ou híbrido serão detalhados nesta seção. Mais precisamente, são apresentados o cenário proposto para o algoritmo, os tempos de execução das diferentes versões/configurações, os *speedups* alcançados, a divisão da execução em CPU e GPU e a eficiência obtida. Para encerrar esta seção são apresentadas as considerações sobre a utilização de OpenMP, CUDA e OpenACC nesse algoritmo.

5.1.1 Cenário proposto

O algoritmo RNG foi testado em diferentes versões: sequencial, OpenMP, CUDA, OpenACC e em dez variações híbridas de OpenMP + CUDA e OpenMP + OpenACC. O cenário proposto para a execução foi a geração de 80.000.000 números. Foram utilizadas 8 *threads* na CPU e um bloco com 512 *threads* na GPU. Os percentuais de execução em CPU e GPU do algoritmo RNG são apresentados pela tabela 3. Esses foram adotados com maior processamento em GPU, pois nos testes realizados a unidade de processamento gráfico apresentou maior desempenho se comparado a CPU.

Tabela 3 - Percentuais de execução em CPU e GPU no algoritmo RNG

Teste	% de execução em CPU	% de execução em GPU
1	50	50
2	40	60
3	30	70
4	20	80
5	10	90

5.1.2 Tempo de processamento

Os tempos de execução de cada configuração do algoritmo RNG são apresentados na figura 23. Através da versão sequencial obteve-se o tempo de 13,440s. Nas versões paralelas e híbridas o tempo de processamento variou entre 1,998s no algoritmo CUDA e 4,361s no algoritmo híbrido OpenMP + OpenACC (50% CPU + 50% GPU). No algoritmo RNG as versões híbridas OpenMP + CUDA superaram o tempo de resposta da aplicação que utiliza somente CUDA (1,998s) a partir da divisão de 30% CPU + 70% GPU, sendo o melhor cenário desse algoritmo a divisão de 20% CPU + 80% GPU com uma execução de 1,869s.

Já as versões híbridas do algoritmo OpenMP + OpenACC não conseguiram atingir o desempenho obtido pela versão OpenACC (2,312s), sendo o seu melhor desempenho a divisão de 20% CPU + 80% GPU com um tempo de 2,368s. O que impossibilitou esse algoritmo híbrido de apresentar maior desempenho foi a execução do OpenMP, que apresentou um tempo de execução acima do esperado. Esse ponto será melhor detalhado na subseção 5.1.6.

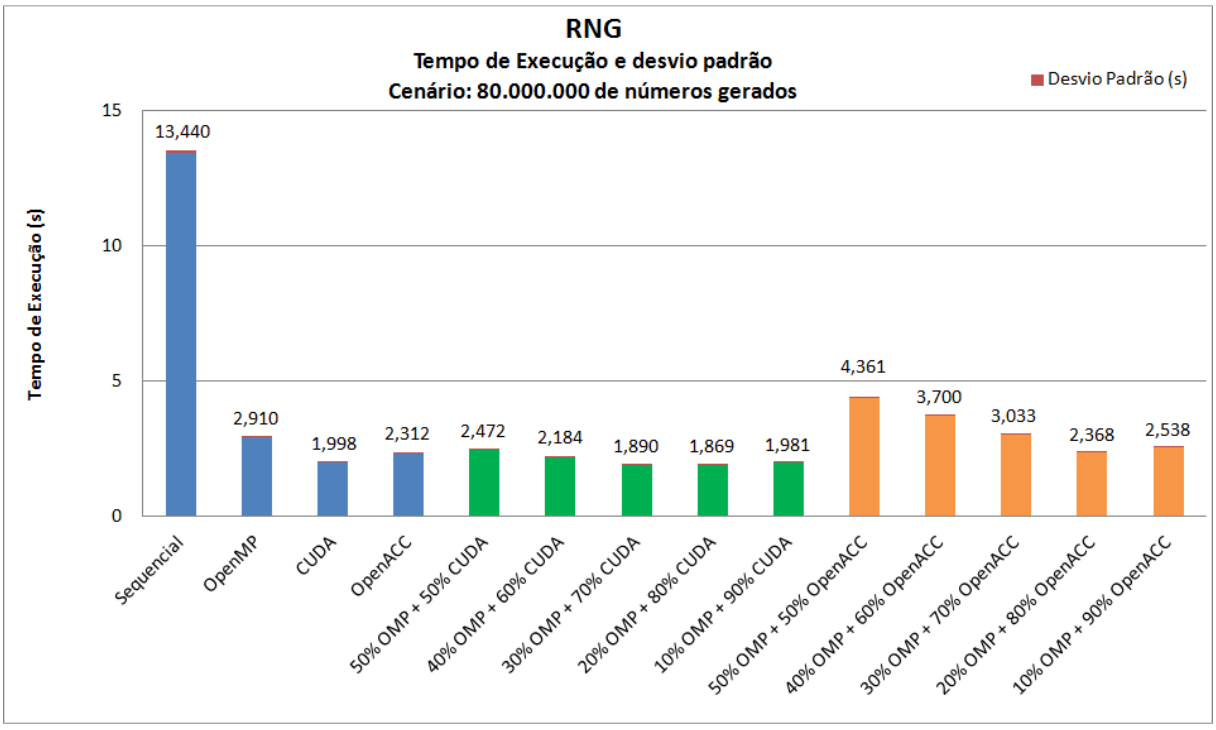

Figura 23 - Tempos de execução do algoritmo RNG

Ainda de acordo com a figura 23, verificou-se uma pequena variação no desvio padrão, de 0,007s nas versões CUDA e híbrida OpenMP + OpenACC (20% CPU + 80% GPU) até 0,085s na versão sequencial. Em geral, as versões híbridas que utilizaram

OpenACC tiveram menor desvio padrão se comparadas com a utilização de OpenMP + CUDA. A média do desvio padrão das configurações do algoritmo híbrido OpenMP + CUDA foi de 0,032s, enquanto que para o algoritmo híbrido OpenMP + OpenACC foi de 0,013s.

Os percentuais de confiança das amostras são apresentados pela tabela 4, onde esse percentual representa a chance de um novo valor estar dentro da média obtida, tendo uma probabilidade de erro de 1%. Os percentuais de confiança das amostras dos algoritmos das versões OpenMP, CUDA, OpenACC e híbridos OpenMP + CUDA e OpenMP + OpenACC tiveram índices superiores a 98% e alguns deles superando a casa dos 99%, baseado nos resultados das 30 amostras executadas para cada versão/configuração. O algoritmo sequencial por apresentar um desvio padrão maior que os demais obteve um percentual de confiança menor, 96,02%.

Tabela 4 - Percentuais de confiança das amostras do algoritmo RNG

Algoritmo	% confiança	Algoritmo	% confiança	Algoritmo	% confiança
Sequencial	96,02	50% OMP + 50% CUDA	99,09	50% OMP + 50% ACC	99,47
OpenMP	98,45	40% OMP + 60% CUDA	98,58	40% OMP + 60% ACC	99,48
CUDA	99,69	30% OMP + 70% CUDA	98,25	30% OMP + 70% ACC	99,51
OpenACC	99,58	20% OMP + 80% CUDA	98,89	20% OMP + 80% ACC	99,65
		10% OMP + 90% CUDA	98,62	10% OMP + 90% ACC	98,77

A subseção 5.1.3, a seguir, detalhará os *speedups* obtidos ao utilizar OpenMP e CUDA no algoritmo RNG.

5.1.3 *Speedups* obtidos utilizando OpenMP e CUDA

Os *speedups* obtidos pelas variações do algoritmo RNG utilizando OpenMP e CUDA, seja ele executado somente em CPU ou GPU, ou através de execuções híbridas são apresentados pela figura 24. Através desse gráfico é possível observar que os melhores desempenhos ficam por conta do algoritmo híbrido OpenMP + CUDA, 20% do processamento na CPU e 80% na GPU com *speedup* de 7,189, seguido da divisão de 30% CPU + 70% GPU através de um *speedup* de 7,109. A diferença em relação ao algoritmo CUDA executado somente na GPU foi pequena, pois o seu *speedup* foi de 6,728. Sendo assim, o algoritmo híbrido OpenMP + CUDA (20% CPU + 80% GPU) teve sua execução aproximadamente 6,9% mais rápida que o algoritmo CUDA. Todas as versões híbridas

utilizando OpenMP e CUDA apresentaram *speedup* superior a versão do algoritmo OpenMP executado somente na CPU.

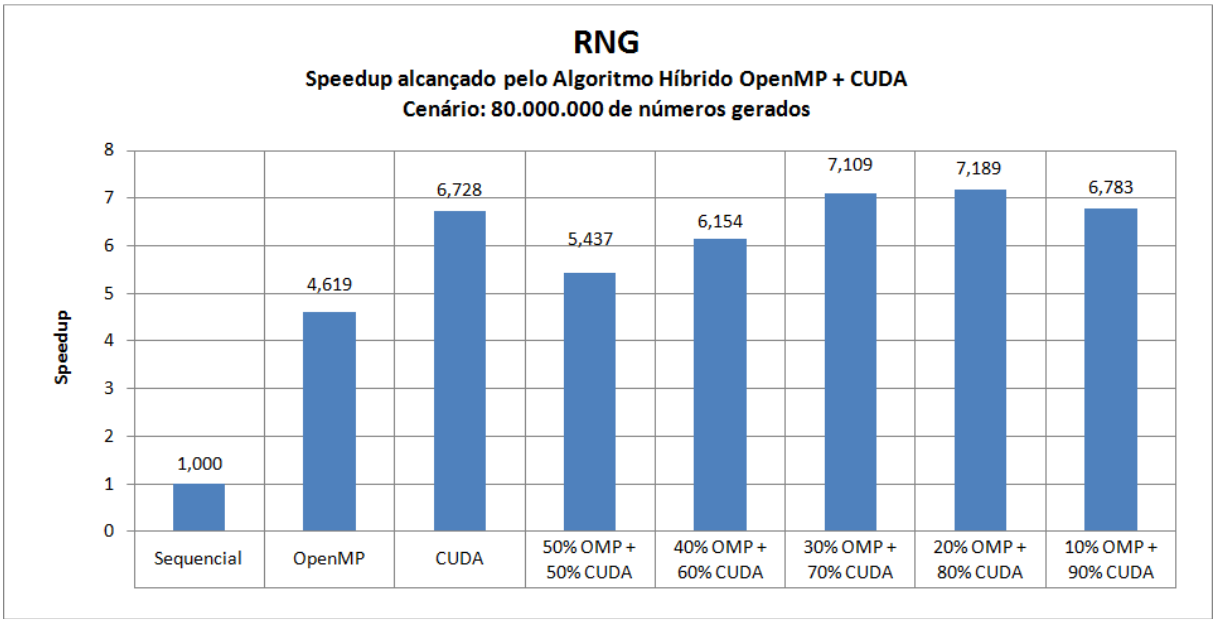

Figura 24 - *Speedups* obtidos pelas versões do algoritmo RNG utilizando OpenMP e CUDA

A seguir, a subseção 5.1.4 irá apresentar os *speedups* obtidos pelas configurações do algoritmo híbrido OpenMP + OpenACC.

5.1.4 *Speedups* obtidos utilizando OpenMP e OpenACC

Com a utilização do OpenACC também ocorreu aceleração no algoritmo RNG, porém em menor intensidade, se comparada com a utilização de CUDA, como ilustra o gráfico da figura 25. As acelerações obtidas pelos algoritmos híbridos que utilizam OpenACC em suas melhores configurações (20% CPU + 80% GPU e 10% CPU + 90% GPU) ficaram muito próximas do *speedup* obtido pelo algoritmo OpenACC (5,812), porém não o superaram. No algoritmo híbrido OpenMP + OpenACC a medida em que mais iterações foram processadas pela GPU, maior foi o desempenho, até a configuração 20% CPU + 80% GPU (*speedup* de 5,676). Observa-se através do gráfico da figura 25 que ao serem processadas 10% das instruções em CPU e 90% em GPU o desempenho diminuiu, pois o tempo de processamento na GPU superou o tempo de processamento na CPU. No melhor cenário do algoritmo híbrido OpenMP + OpenACC (20% CPU + 80% GPU) seu desempenho foi de aproximadamente 97,6% do desempenho do algoritmo OpenACC.

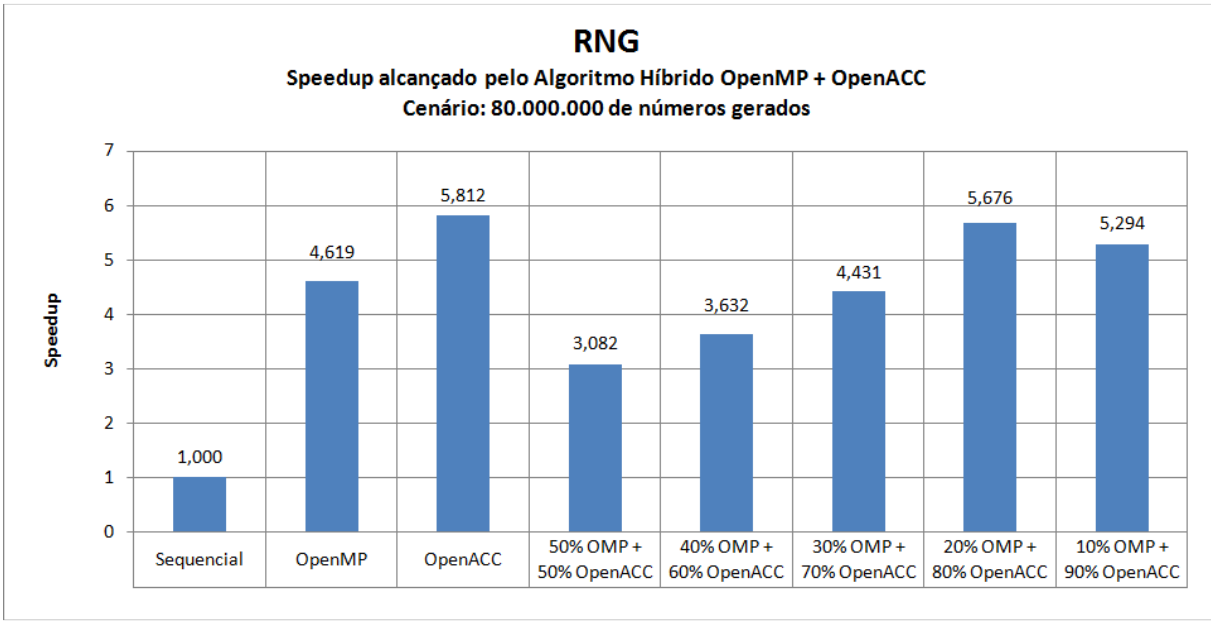

Figura 25 - *Speedups* obtidos pelas versões do algoritmo RNG utilizando OpenMP e OpenACC

Na sequência, na subseção 5.1.5, serão apresentadas as divisões (em tempo) da execução das versões em CPU e GPU do algoritmo RNG.

5.1.5 Análise da execução dos algoritmos em CPU ou GPU

A distribuição da execução dos algoritmos sequencial, OpenMP, CUDA e OpenACC é apresentada pelo gráfico da figura 26, onde a "Inicialização" é a etapa em que ocorre a alocação dinâmica de memória dos vetores e a atribuição inicial dos dados. Após o processamento que leva à geração dos 80.000.000 números ocorre o processo de liberação da memória das variáveis utilizadas, sendo essa etapa chamada no gráfico de "Término". Nos algoritmos CUDA e OpenACC no início da etapa "Processamento Paralelo GPU" ocorre o envio dos dados da CPU para a GPU. Encerrada a transferência dos dados é iniciado o processamento para geração dos números na GPU e ao seu término ocorre a transferência do resultado da GPU para a CPU.

No algoritmo RNG, a versão sequencial apresentou tempo médio de execução de 13,440s, sendo que o processamento para geração dos 80.000.000 números durou 13.02s. Em contrapartida, o algoritmo OpenMP processou por 2,910s, sendo que a geração dos números foi em 2,477s. O processamento paralelo em GPU tanto em CUDA como em OpenACC apresentou tempo de resposta menor se comparado aos algoritmos sequencial e OpenMP. Em CUDA, o tempo de processamento paralelo em GPU para gerar os números foi de apenas

1,008s, enquanto que em OpenACC o tempo foi de 1,257s, sendo que o tempo total de execução foi de 1,998s e 2,312s para os algoritmos CUDA e OpenACC, respectivamente.

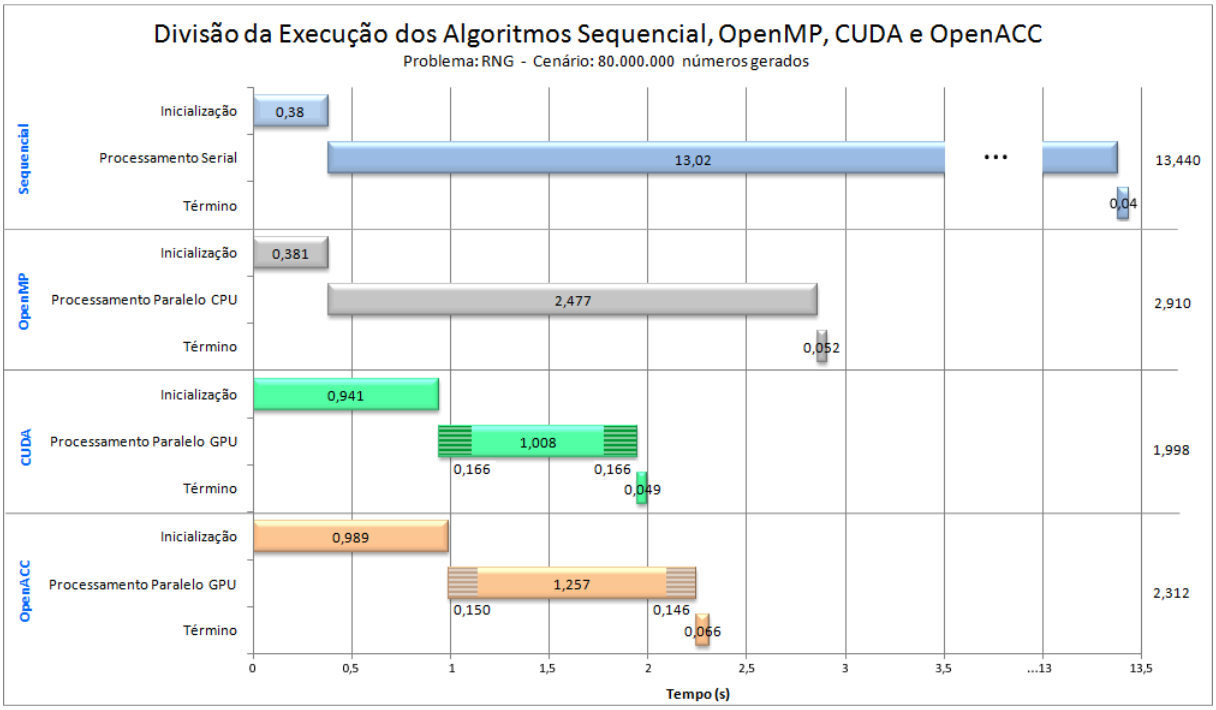

Figura 26 - Divisão da execução das versões sequencial, OpenMP, CUDA e OpenACC do algoritmo RNG

Nos algoritmos paralelos em GPU, a área do gráfico hachurada (início e término do "Processamento Paralelo GPU") corresponde ao envio dos dados da CPU para a GPU e da GPU para a CPU. No algoritmo CUDA o tempo de envio das informações em cada sentido levou em média 0,166s. O tempo médio de transferência de dados da CPU para a GPU no algoritmo OpenACC durou 0,150s, enquanto que no sentido contrário levou 0,146s. O tempo de execução em GPU computa a transferência de dados nos dois sentidos e o processamento no dispositivo gráfico.

Apesar dos algoritmos processados em GPU terem maior desempenho, o tempo para alocação dos vetores e atribuição inicial dos dados foi maior em relação aos algoritmos sequencial e OpenMP, pois são alocados e inicializados o dobro de vetores. Nesses algoritmos são alocados e inicializados 6 vetores, sendo 3 para a CPU e 3 para a GPU. Nos algoritmos sequencial e OpenMP são alocados e inicializados apenas 3 vetores na CPU.

5.1.6 Análise da execução dos algoritmos híbridos

Na divisão da execução dos algoritmos híbridos OpenMP + CUDA e OpenMP + OpenACC são comparadas as execuções das versões híbridas 50% CPU + 50% GPU e a versão com maior desempenho de cada algoritmo, 20% CPU + 80% GPU nos dois algoritmos híbridos, conforme apresenta a figura 27. Verificou-se que o tempo denominado "Inicialização" responsável por alocar os vetores e inicializar os dados apresentou pequena variação, entre 0,949s e 0,961s.

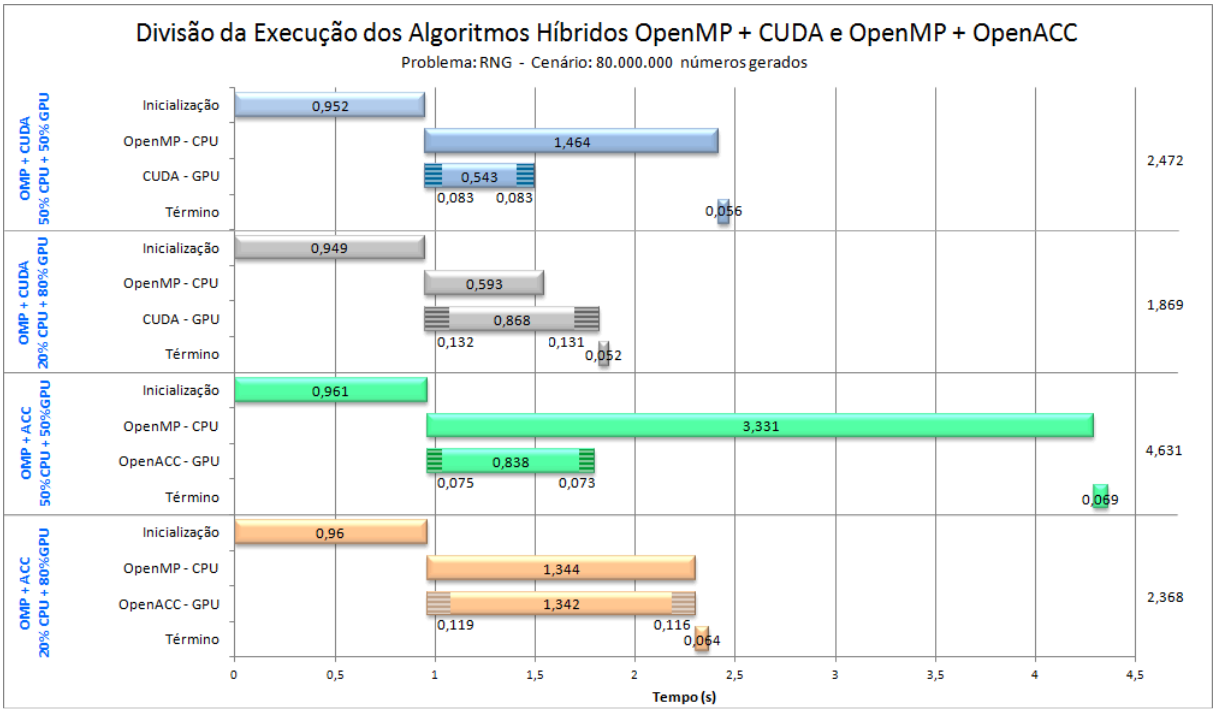

Figura 27 - Divisão da execução das versões híbridas OpenMP + CUDA e OpenMP + OpenACC do algoritmo RNG

O maior desempenho do algoritmo híbrido OpenMP + CUDA foi ao dividir o processamento em 20% na CPU e 80% em GPU. Nessa configuração o processamento paralelo inicia no segundo 0,949, processando por 0,593s na CPU e por 0,868s na GPU. As áreas hachuradas no início e no término do processamento na GPU correspondem ao tempo de transferência dos dados entre os dispositivos. No algoritmo OpenMP + CUDA com 50% do processamento em cada unidade a transferência entre os dispositivos leva em média 0,083s, nos dois sentidos. Na divisão 20% CPU + 80% GPU o envio dos dados para a GPU leva 0,132s e o retorno das informações para a CPU dura em média 0,131s.

O algoritmo híbrido OpenMP + OpenACC, apesar de não conseguir superar o desempenho do algoritmo OpenACC executado apenas na GPU, também apresentou o seu

maior *speedup* ao dividir a execução em 20% na CPU e 80% na GPU, onde o processamento paralelo inicia no segundo 0,960, executando por 1,344s na CPU e por 1,342s na GPU. Também no algoritmo híbrido OpenMP + OpenACC verificou-se que o desempenho do OpenMP foi de apenas 37,18% do desempenho dessa mesma API no algoritmo OpenMP. Na seção 6.1 será apresentado o motivo que levou a esse baixo desempenho. Na divisão de 50% do processamento em cada dispositivo o tempo gasto com a transferência entre os dispositivos foi de 0,148s, sendo 0,075s para o envio e 0,073s para o recebimento. Com 20% de processamento na CPU e 80% na GPU a transferência de dados durou 0,235s, desses 0,119s gastos para enviar os dados da CPU para a GPU e 0,116s para retornar as informações para a CPU após o processamento. O processo de finalização dos algoritmos híbridos, responsável por liberar da memória as variáveis utilizadas teve tempo de execução entre 0,052s e 0,069s, não impactando no tempo total dos algoritmos.

5.1.7 Eficiência das versões do algoritmo RNG

O cálculo da eficiência do algoritmo RNG é baseado na apresentação realizada anteriormente na seção 4.1. O algoritmo híbrido OpenMP + CUDA apresentou eficiências entre 68,62% e 77,23%, enquanto que no algoritmo híbrido OpenMP + OpenACC a eficiência ficou entre 44,94% e 60,62%, conforme ilustra o gráfico da figura 28.

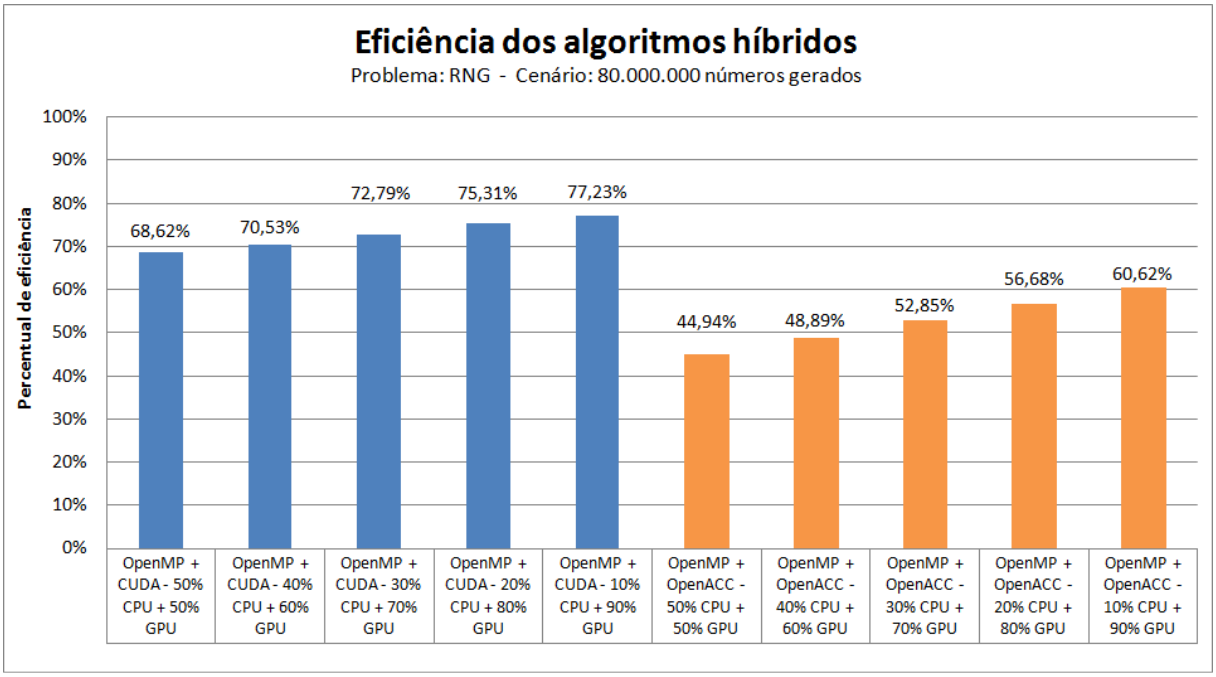

Figura 28 - Eficiência das versões híbridas OpenMP + CUDA e OpenMP + OpenACC do algoritmo RNG

A diferença entre as eficiências dos dois algoritmos híbridos reside nos fatos de que o desempenho do OpenMP no algoritmo híbrido OpenMP + CUDA foi aproximadamente 2,2 vezes superior ao desempenho do OpenMP no algoritmo híbrido OpenMP + OpenACC e o desempenho de CUDA foi aproximadamente 1,5 vezes superior ao desempenho do OpenACC nos algoritmos híbridos.

Mesmo que as distribuições de 10% CPU + 90% GPU dos dois algoritmos híbridos não apresentaram o melhor desempenho, tiveram as melhores eficiências. Isso aconteceu porque o *speedup* do processamento na GPU foi calculado com base no tempo de execução sequencial no dispositivo gráfico. Dessa forma, os *speedups* entre uma divisão e outra de processamento ficam muito próximos e o que acabou definindo uma eficiência maior é a quantidade de trabalho em um ou outro dispositivo.

Para finalizar a abordagem dos resultados obtidos pelos algoritmos híbridos serão apresentadas na subseção 5.1.8, a seguir, as considerações finais sobre a utilização de OpenMP, CUDA e OpenACC no algoritmo RNG.

5.1.8 Considerações

Verificou-se que nas configurações de divisão de trabalho dos algoritmos híbridos, quando bem balanceado o processamento entre CPU e GPU, apresentaram maior desempenho se comparadas as execuções de OpenMP, CUDA e OpenACC paralelizadas para um único dispositivo. Nos algoritmos híbridos o desempenho utilizando CUDA foi aproximadamente 1,5 vezes maior do que se utilizando OpenACC, apesar da programação ter apresentado mais instruções para a troca de informações entre os dispositivos e na criação e execução do *kernel*. OpenACC nas versões híbridas possibilitou um rápido desenvolvimento, pois suas instruções assemelham-se as instruções do OpenMP. Verificou-se que a execução do OpenMP no algoritmo híbrido OpenMP + OpenACC teve desempenho inferior (aproximadamente 2,2 vezes) ao desempenho do OpenMP do algoritmo híbrido OpenMP + CUDA, mesmo que a execução tenha ocorrido de forma paralela. Esse fato contribuiu para que o algoritmo híbrido OpenMP + OpenACC não apresentasse um desempenho maior.

Observou-se também que no algoritmo RNG as execuções paralelas em GPU utilizando CUDA e OpenACC resultaram em bons índices de desempenho, maiores que o desempenho paralelo em CPU utilizando OpenMP, porém inferior ao desempenho híbrido. Esse fato evidencia que a capacidade de processamento das GPUs tem aumentado e esse dispositivo tem sido uma alternativa interessante para auxiliar no processamento. Nesse

algoritmo, o desempenho de CUDA foi superior ao desempenho de OpenACC quando executado juntamente com o OpenMP de forma híbrida. O desempenho de OpenACC híbrido foi de 64,8% do desempenho de CUDA híbrido no algoritmo RNG.

Encerrada a apresentação da análise dos resultados do algoritmo RNG, a seguir, na seção 5.2 serão apresentados os resultados dos testes obtidos pelas versões/configurações do algoritmo Hotspot.

5.2 Hotspot

Os resultados das diferentes versões/configurações do algoritmo Hotspot são apresentados nesta seção. Em primeiro momento é apresentado o cenário proposto para o algoritmo e na sequência são detalhados os tempos de execução das diferentes versões/configurações, os *speedups* alcançados, a divisão da execução em CPU e GPU em forma de *timeline* e a eficiência obtida. Para finalizar são apresentadas as considerações sobre a utilização de OpenMP, CUDA e OpenACC nesse algoritmo.

5.2.1 Cenário proposto

O algoritmo Hotspot foi testado obedecendo à mesma sistemática do algoritmo RNG, ou seja, versão sequencial, OpenMP, CUDA e OpenACC, além das variações híbridas de OpenMP + CUDA e OpenMP + OpenACC. Entretanto foi proposta mais uma configuração de distribuição de trabalho, 5% CPU + 95% GPU, conforme apresenta a tabela 5, pois essa variação foi a de maior desempenho no algoritmo híbrido OpenMP + CUDA. Foram utilizadas 8 *threads* na CPU e um bloco com 256 *threads* na GPU.

Tabela 5 - Percentuais de execução em CPU e GPU no algoritmo Hotspot

Teste	% de execução em CPU	% de execução em GPU
1	50	50
2	40	60
3	30	70
4	20	80
5	10	90
6	5	95

O cenário de execução do algoritmo Hotspot foi o seguinte: 5.000 iterações utilizando arquivos de temperatura inicial e de potência dissipada de 1.048.576 registros cada.

5.2.2 Tempo de processamento

Os tempos de execução e desvio padrão de cada versão/configuração do algoritmo Hotspot são apresentados pela figura 29. A versão sequencial teve tempo de execução de 126,082s e foi dentre os testes desse cenário a que apresentou maior desvio padrão (0,122s). A versão OpenMP apresentou um *speedup* de 4,594 com o tempo de 27,448s. Os tempos de execução de 3,136s e 4,795s foram obtidos pelos algoritmos CUDA e OpenACC, respectivamente. Nos algoritmos híbridos OpenMP + CUDA e OpenMP + OpenACC apenas na configuração 5% CPU + 95% GPU do algoritmo híbrido OpenMP + CUDA obteve-se tempo de processamento menor que uma versão paralela para GPU. Enquanto que o tempo de processamento em CUDA foi de 3,136s no algoritmo híbrido OpenMP + CUDA (5% CPU + 95% GPU) o tempo de execução foi de 3,088s.

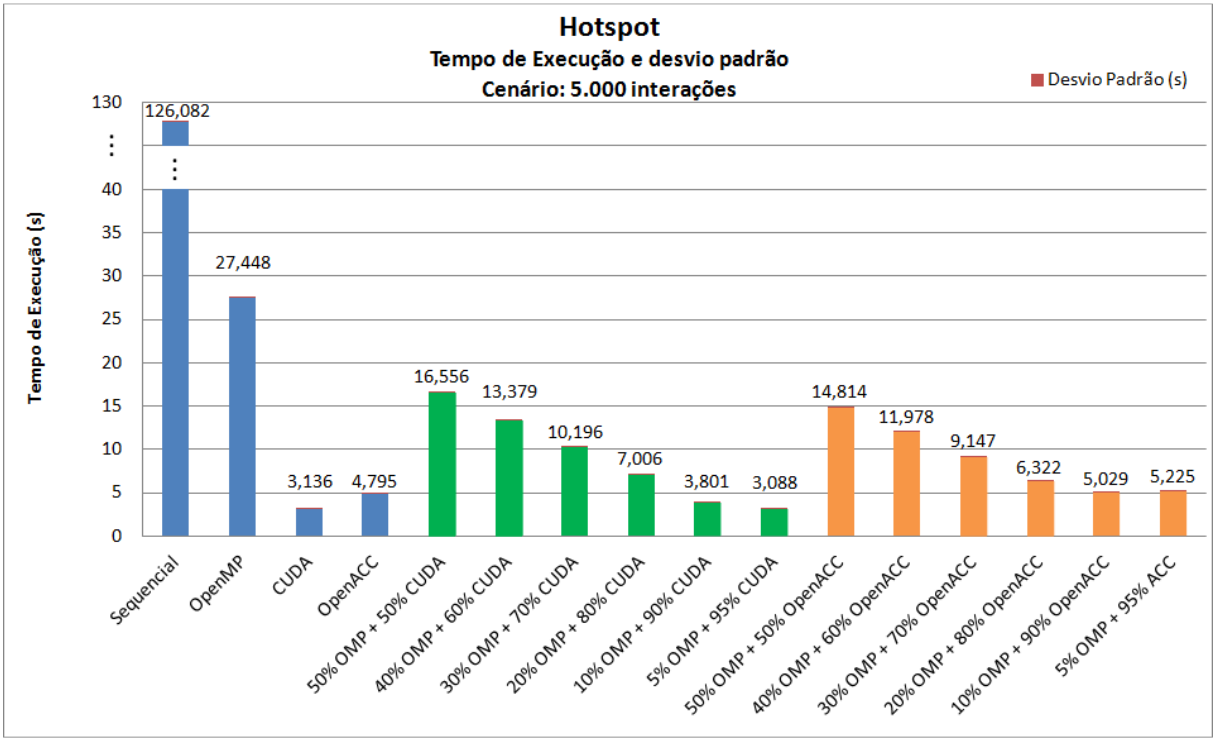

Figura 29 - Tempos de execução do algoritmo Hotspot

A melhor distribuição de trabalho entre CPU e GPU do algoritmo híbrido OpenMP + OpenACC ocorreu ao executar 10% das iterações em CPU e 90% em GPU, com um tempo de 5,029s, menor que o tempo do algoritmo OpenMP (27,448s), porém maior que o tempo de execução do algoritmo OpenACC (4,795s).

Os percentuais de confiança das amostras são apresentados pela tabela 6, onde esse percentual representa a chance de um novo valor estar dentro da média obtida, tendo uma probabilidade de erro de 1%.

Tabela 6 - Percentuais de confiança das amostras do algoritmo Hotspot

Algoritmo	% confiança	Algoritmo	% confiança	Algoritmo	% confiança
Sequencial	94,28	50% OMP + 50% CUDA	98,42	50% OMP + 50% ACC	97,07
OpenMP	97,31	40% OMP + 60% CUDA	98,47	40% OMP + 60% ACC	98,51
CUDA	99,70	30% OMP + 70% CUDA	98,36	30% OMP + 70% ACC	99,17
OpenACC	99,22	20% OMP + 80% CUDA	99,32	20% OMP + 80% ACC	99,16
		10% OMP + 90% CUDA	98,13	10% OMP + 90% ACC	98,70
		05% OMP + 95% CUDA	99,10	05% OMP + 95% ACC	99,65

O percentual de confiança para os algoritmos paralelos ficou acima de 97%, sendo que algumas versões como OpenMP + CUDA (5% CPU + 95% GPU) e OpenMP + OpenACC (30% CPU + 70% GPU) alcançaram percentual de confiança acima de 99%. A versão sequencial entre as dezesseis versões/configurações testadas foi a que obteve menor percentual de confiança (94,28%), uma vez que apresentou o maior desvio padrão (0,122s).

Os *speedups* alcançados pelos algoritmos Hotspot que utilizaram OpenMP e CUDA são detalhados na subseção 5.2.3, a seguir.

5.2.3 *Speedups* obtidos utilizando OpenMP e CUDA

Os desempenhos alcançados pelas variações do algoritmo Hotspot utilizando OpenMP e CUDA, com execução somente em CPU ou GPU, ou através de processamento híbrido são ilustrados pelo gráfico da figura 30. Observa-se através desse gráfico que à medida que mais instruções são processadas pela GPU no algoritmo híbrido OpenMP + CUDA maior foi o desempenho alcançado. O maior *speedup* obtido pelo algoritmo Hotspot, dentre as diferentes versões/configurações foi ao dividir 5% das iterações em CPU e 95% em GPU, 40,830, sendo 8,88 vezes mais rápida sua execução em relação ao OpenMP (4,594s) e apenas 1,015 vezes mais veloz que o algoritmo CUDA (3,136s). Além disso, todas as versões híbridas utilizando OpenMP e CUDA apresentaram *speedup* superior a versão do algoritmo OpenMP.

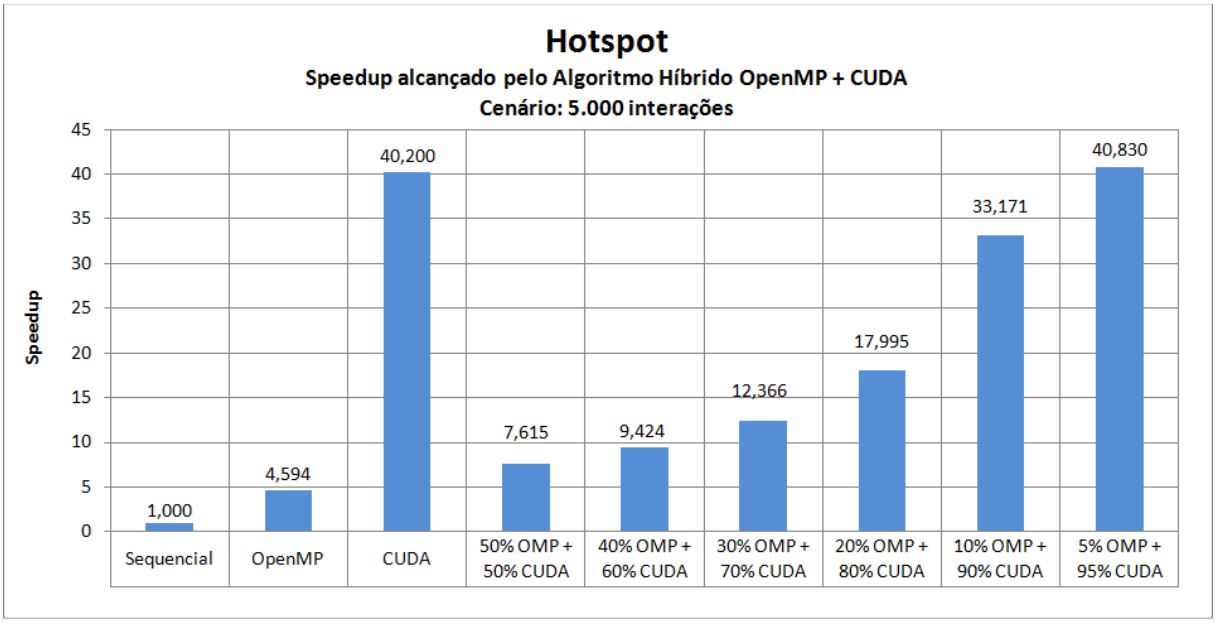

Figura 30 - *Speedups* obtidos pelas versões do algoritmo Hotspot utilizando OpenMP e CUDA

A seguir, na subseção 5.2.4 serão apresentados os *speedups* obtidos através da utilização de OpenMP + OpenACC.

5.2.4 *Speedups* obtidos utilizando OpenMP e OpenACC

Os desempenhos obtidos pelas diferentes versões/configurações do algoritmo Hotspot utilizando OpenMP + OpenACC são apresentados pela figura 31. Verifica-se que os desempenhos apresentados por CUDA na subseção anterior (5.2.3) foram maiores se comparados aos *speedups* obtidos pelo OpenACC. Enquanto que o desempenho de CUDA foi de 40,200, OpenACC apresentou *speedup* de 26,293. O maior *speedup* obtido no algoritmo híbrido OpenMP + OpenACC foi 25,071 ao dividir 10% do processamento em CPU e 90% em GPU.

Também de acordo com a figura 31, à medida que mais iterações foram processadas pela GPU no algoritmo híbrido OpenMP + OpenACC, até a distribuição 10% CPU + 90% GPU, maior foi o desempenho. Na distribuição 5% CPU e 95% GPU o desempenho foi inferior à configuração 10% CPU e 90% GPU, pois através da configuração 5% CPU e 95% GPU aumenta-se a diferença entre o tempo de processamento nos dois dispositivos (aumenta o tempo de processamento na GPU enquanto diminui o tempo de execução na CPU), impossibilitando um maior desempenho.

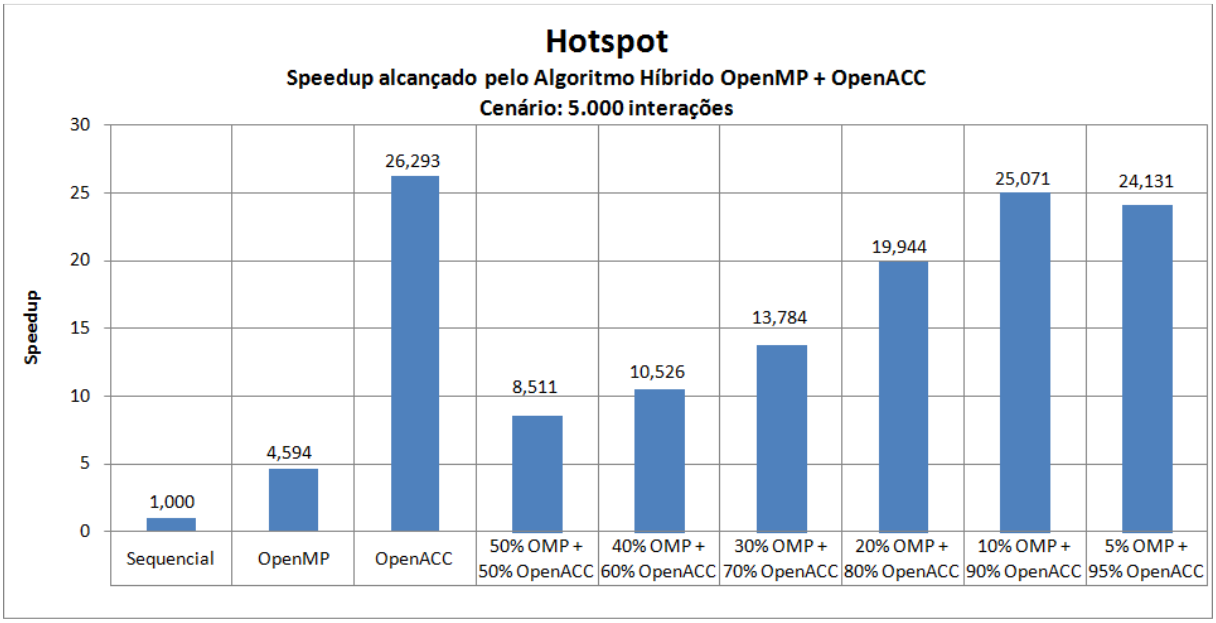

Figura 31 - *Speedups* obtidos pelas versões do algoritmo Hotspot utilizando OpenMP e OpenACC

O desempenho da melhor divisão híbrida do algoritmo OpenMP + OpenACC (10% CPU + 90% GPU) foi 95,3% do desempenho do algoritmo OpenACC executado apenas na GPU. Finalizada a apresentação dos *speedups* das versões/configurações do algoritmo Hotspot, serão detalhadas nas duas próximas subseções a divisão da execução em CPU e GPU.

5.2.5 Análise da execução dos algoritmos em CPU ou GPU

A distribuição das etapas da execução dos algoritmos sequencial, OpenMP, CUDA e OpenACC é ilustrada pela figura 32, onde "Inicialização" é o processo em que ocorre a alocação dinâmica de memória das variáveis e a leitura dos arquivos de temperatura e potência. Após o processamento das 5.000 iterações ocorre a etapa de liberação da memória das variáveis utilizadas, denominado no gráfico de "Término".

No algoritmo Hotspot a versão sequencial apresentou tempo médio de processamento de 126,082s, sendo que a etapa de inicialização das variáveis e leitura dos arquivos teve tempo médio de 0,62s. As 5.000 iterações foram processadas sequencialmente em 125,460s e a liberação das variáveis utilizadas da memória levou 0,002s. Em OpenMP o processo de inicialização do algoritmo levou 0,614s e a geração das 5.000 iterações durou 26,832s, sendo que o algoritmo processou por 27,448s. Nos algoritmos CUDA e OpenACC antes do início da etapa de processamento ocorreu o envio dos dados da CPU para a GPU e após o término do

processamento na GPU aconteceu a transferência das informações do dispositivo gráfico para a CPU.

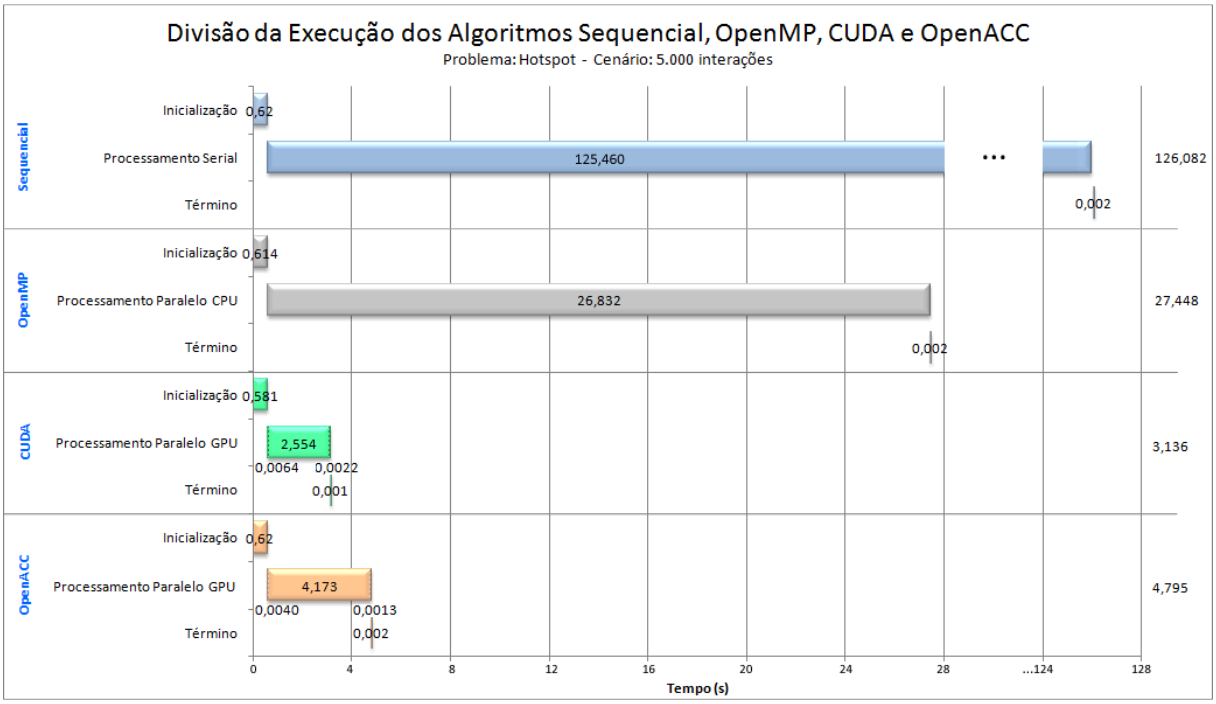

Figura 32 - Divisão da execução das versões sequencial, OpenMP, CUDA e OpenACC do algoritmo Hotspot

Nos algoritmos CUDA e OpenACC o tempo de inicialização e finalização foi semelhante aos obtidos pelos algoritmos sequencial e OpenMP. Porém a execução das iterações na GPU utilizando CUDA durou 2,554s enquanto que no algoritmo OpenACC o processamento apresentou tempo de 4,173s. As versões CUDA e OpenACC obtiveram tempo total de processamento de 3,316s e 4,795s, respectivamente. Os tempos de transferência entre os dispositivos no algoritmo CUDA, da CPU para a GPU durou 0,0064s, enquanto que no sentido contrário levou 0,0022s, conforme mostram as pequenas áreas hachuradas da figura 32. Na versão OpenACC o tempo gasto no envio dos dados para a CPU levou 0,0040s e, após o processamento, o recebimento das informações da GPU durou 0,0013s. O tempo de execução em GPU computou a transferência de dados nos dois sentidos e o processamento no dispositivo gráfico. No algoritmo Hotspot foram enviados para a GPU três vetores do tipo *double* contendo cada um 1.048.576 posições e retornado um vetor de igual tamanho. Mais informações a respeito da transferência de dados entre CPU e GPU são apresentadas pela subseção 5.4.2.

5.2.6 Análise da execução dos algoritmos híbridos

As execuções híbridas utilizando OpenMP + CUDA e OpenMP + OpenACC apresentaram tempo aproximado de 0,6s para a alocação das variáveis e leitura dos arquivos de temperatura e potência, semelhante aos tempos dos algoritmos apresentados na subseção 5.2.5, repetindo também a proximidade no tempo para finalização dos algoritmos (etapa "Término"). Nesta subseção para os algoritmos híbridos OpenMP + CUDA e OpenMP + OpenACC são apresentadas duas divisões de trabalho em CPU e GPU, a saber: 50% em CPU mais 50% em GPU para ambos os algoritmos e a distribuição que maior desempenho resultou em cada algoritmo. Utilizando OpenMP + CUDA a configuração que maior *speedup* resultou foi ao dividir 5% do processamento em CPU e 95% em GPU, enquanto que no algoritmo híbrido OpenMP + OpenACC a melhor divisão do número de iterações foi ao processar 10% das iterações em CPU e 90% em GPU. O gráfico da figura 33 ilustra a divisão de trabalho entre CPU e GPU dos algoritmos híbridos OpenMP + CUDA e OpenMP + OpenACC.

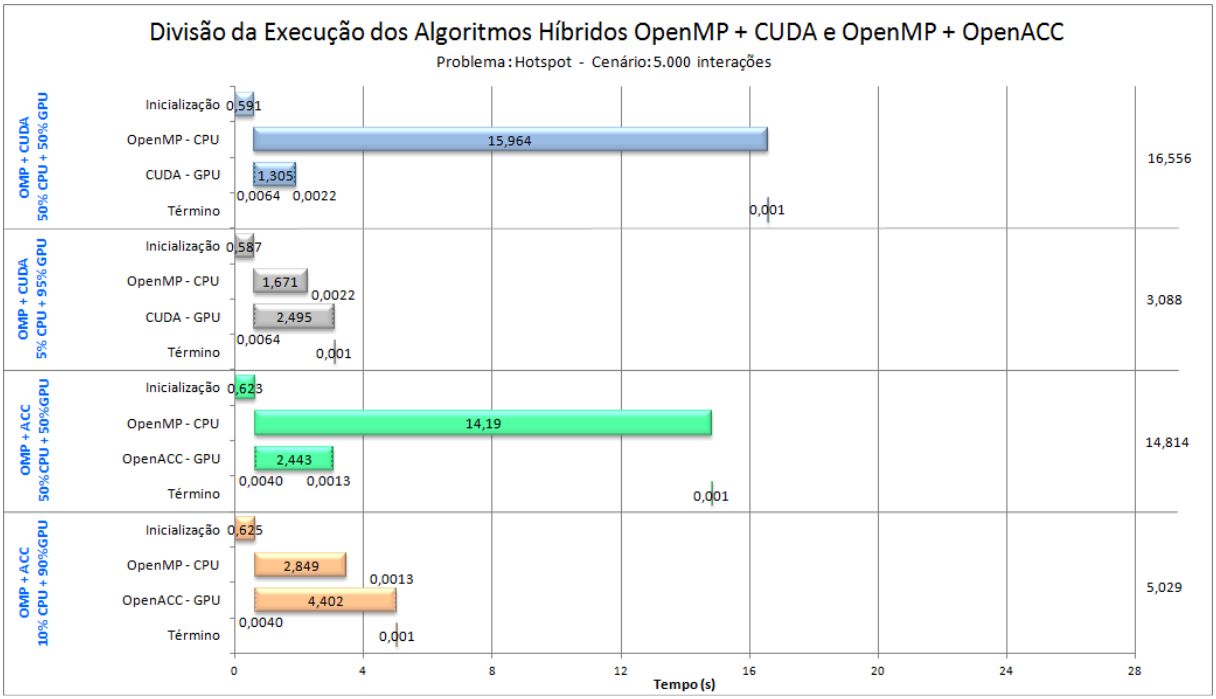

Figura 33 - Divisão da execução das versões híbridas OpenMP + CUDA e OpenMP + OpenACC do algoritmo Hotspot

No algoritmo híbrido OpenMP + CUDA (50% CPU + 50% GPU) percebe-se que a GPU está ociosa por um longo período em que a CPU está processando. Enquanto que o processamento do OpenMP leva 15,964s, o tempo de execução em GPU é de apenas 1,305s. Nesta divisão de trabalho o tempo médio de execução do algoritmo foi de 16,556s. Quando

ocorre a divisão de 5% das iterações em CPU e 95% em GPU há uma diminuição do tempo de execução em CPU para 1.671s, enquanto que no processamento em CUDA ocorre um aumento no tempo para 2,495s, fazendo com que esta configuração seja a que execute as 5.000 iterações em menor tempo, 3,088s.

Já no algoritmo híbrido OpenMP + OpenACC o processamento em CPU para uma divisão igualitária de iterações nos dois dispositivos foi de 14,190s, menor que o tempo de processamento da mesma configuração do algoritmo hibrido OpenMP + CUDA (15,964s). Essa diferença ocorreu porque a otimização para o código OpenMP criada pelo compilador PGCC foi melhor que a criada pelo compilador NVCC. Porém o tempo de processamento na GPU no algoritmo híbrido OpenMP + OpenACC foi maior que do algoritmo híbrido OpenMP + CUDA, 2,443s contra 1,305s, respectivamente, para uma divisão de 50% em cada dispositivo. Nesta divisão, o algoritmo OpenMP + ACC processou por 14,814s.

O maior desempenho do algoritmo híbrido OpenMP + OpenACC ocorreu ao dividir 10% das iterações em CPU e 90% em GPU, sendo que o processamento por parte do OpenMP durou 2,849s e o processamento em GPU através do OpenACC levou 4,402s (iniciando no segundo 0,626), perfazendo um tempo total de 5,029s e assim, impossibilitando que uma divisão maior em favor da GPU pudesse alcançar maior desempenho, pois o tempo de processamento em GPU superou o tempo de execução em CPU. Nos algoritmos híbridos o tempo para processamento em CPU através do OpenMP foi menor nas configurações do algoritmo híbrido OpenMP + OpenACC, aproximadamente 12%. Entretanto, o tempo de execução em GPU foi menor ao se utilizar CUDA ao invés de OpenACC, em torno de 54%. O tempo gasto com a transferência das informações entre os dispositivos foi de 0,0064s e 0,0022s no algoritmo OpenMP + CUDA, da CPU para a GPU e da GPU para a CPU, respectivamente. No algoritmo OpenMP + OpenACC o tempo consumido no envio dos dados para a GPU custou 0,0040s e o retorno para a GPU após o processamento durou 0,0013s.

5.2.7 Eficiência das versões do algoritmo Hotspot

As eficiências alcançadas pelas configurações de distribuição de trabalho do algoritmo híbrido OpenMP + CUDA, foram em todas as situações, superiores as eficiências obtidas pelas versões do algoritmo OpenMP + OpenACC, conforme apresenta o gráfico da figura 34. Nas configurações do algoritmo híbrido OpenMP + CUDA há uma variação no percentual de eficiência entre 65,13% e 80,25%, sempre aumentando conforme mais iterações foram executadas pela GPU. Já no algoritmo híbrido OpenMP + OpenACC a eficiência obtida foi

menor se comparada a utilização de OpenMP + CUDA, ficando entre 63,68% e 79,33%, também aumentando a medida em que mais processamento foi realizado pelo dispositivo gráfico.

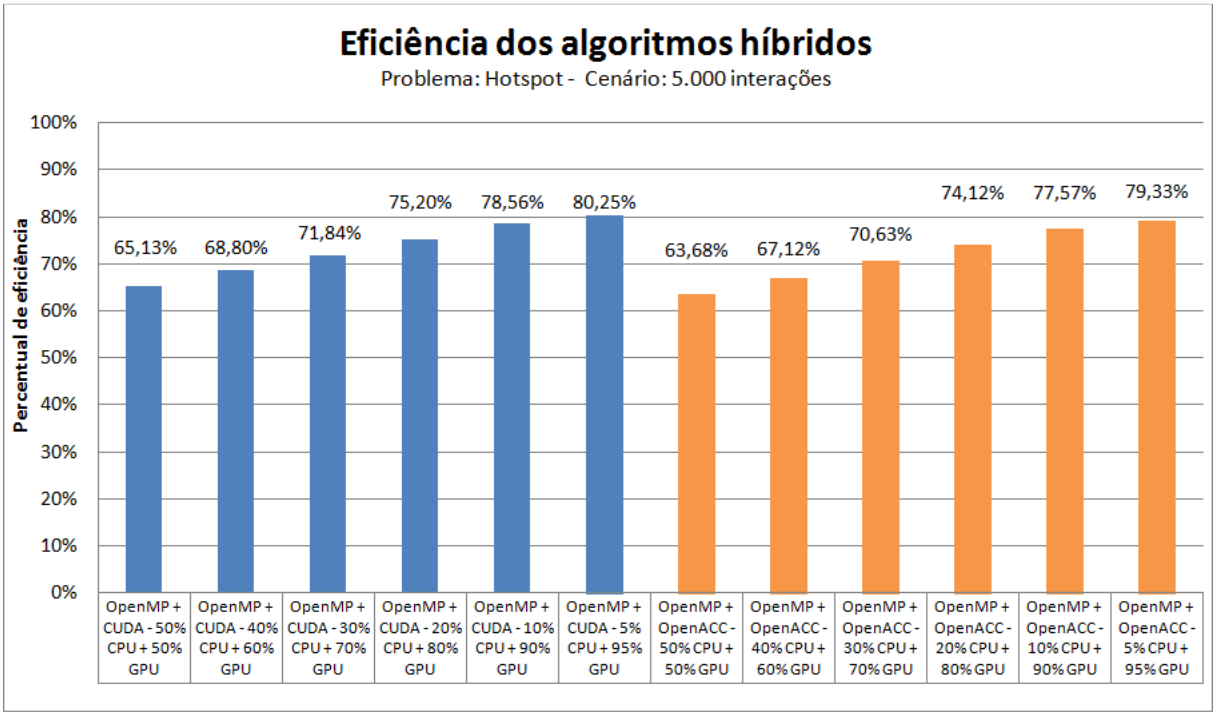

Figura 34 - Eficiência das versões híbridas OpenMP + CUDA e OpenMP + OpenACC do algoritmo Hotspot

A principal diferença entre os percentuais de eficiência obtidos pelas versões dos algoritmos híbridos OpenMP + CUDA e OpenMP + OpenACC está no fato de o tempo de execução em GPU no algoritmo híbrido OpenMP + OpenACC ser maior que o tempo de processamento em GPU do algoritmo híbrido OpenMP + CUDA. Encerrada a apresentação dos resultados dos testes efetuados sobre as diferentes versões/configurações do algoritmo Hotspot, na subseção seguinte (5.2.8) serão apresentadas as considerações finais sobre a utilização das APIs OpenMP, CUDA e OpenACC nesse algoritmo.

5.2.8 Considerações

O processamento paralelo em GPU por si só apresentou *speedups* de 40,200 quando executado por CUDA e 26,293 quando processado por OpenACC, enquanto que o desempenho de OpenMP foi de 4,594. Esse fato mostra que a GPU além de auxiliar no processamento apresentou bons índices de desempenho em ambas APIs. Da mesma forma que aconteceu no algoritmo RNG, o desempenho de CUDA frente à OpenACC executando

somente em GPU foi maior. Em REYES et. al. (2012) nos testes realizados no algoritmo Hotspot o desempenho de CUDA também foi superior ao desempenho de OpenACC. Nas variações do algoritmo híbrido OpenMP + CUDA obteve-se o maior desempenho para o algoritmo Hotspot ao dividir 5% das iterações em CPU e 95% em GPU através de um de *speedup* de 40,830, sendo a única das configurações de processamento entre CPU e GPU que superou o desempenho obtido pelo algoritmo CUDA, aproximadamente 1,6% mais rápido. Não se conseguiu em nenhuma das variações da divisão de trabalho do algoritmo híbrido OpenMP + OpenACC superar o desempenho da versão OpenACC, sendo que a configuração 10% CPU + 90% GPU foi a que mais se aproximou (*speedup* de 25,071). Seu desempenho foi de 95,3% do desempenho do algoritmo OpenACC executado apenas na GPU

O desempenho mais favorável atingido por CUDA nos algoritmos híbridos se dá principalmente pela maior eficiência que o algoritmo conseguiu empregar e por manter os tempos de processamento entre CPU e GPU iniciando e terminando quase que juntos. Houve dificuldade em deixar os tempos de processamento em CPU e GPU próximos ao utilizar OpenACC. Nos algoritmos híbridos o tempo de processamento em CPU através do OpenMP foi menor nas configurações do algoritmo OpenMP + OpenACC. Entretanto o tempo de execução em GPU foi maior ao se utilizar OpenACC ao invés de CUDA. Nas configurações propostas para o algoritmo Hotspot híbrido o desempenho de CUDA foi superior ao desempenho do OpenACC, aproximadamente 54%.

A seção 5.3, a seguir, irá apresentar o resultado dos testes obtidos pelas versões do algoritmo SRAD (*Speckle Reducing Anisotropic Diffusion*).

5.3 SRAD (*Speckle Reducing Anisotropic Diffusion*)

A análise realizada através dos resultados alcançados pelo algoritmo SRAD são detalhados nesta seção. A seguir, serão apresentados o cenário proposto para o algoritmo, os tempos de execução das diferentes versões/configurações, os *speedups* alcançados, a divisão da execução em CPU e GPU e a eficiência obtida. Ainda, para finalizar são apresentadas as considerações sobre a utilização de OpenMP, CUDA e OpenACC nesse algoritmo.

5.3.1 Cenário proposto

O algoritmo SRAD foi testado seguindo à mesma metodologia adotada nos algoritmos RNG e Hotspot através das versões sequencial, OpenMP, CUDA e OpenACC, além dez

variações híbridas de OpenMP + CUDA e OpenMP + OpenACC. As configurações propostas para a divisão de trabalho entre CPU e GPU são apresentadas pela tabela 7, a seguir.

Tabela 7 - Percentuais de execução em CPU e GPU no algoritmo SRAD

Teste	% de execução em CPU	% de execução em GPU
1	50	50
2	40	60
3	30	70
4	20	80
5	10	90

Como cenário de execução do algoritmo SRAD foram utilizados os seguintes parâmetros:

- Nº de linhas da imagem: 2048
- Nº de colunas da imagem: 2048
- Posição da mancha Y1: 0
- Posição da mancha Y2: 127
- Posição da mancha X1: 0
- Posição da mancha X2: 127
- Nº de *threads* na CPU: 8
- Nº de blocos/*threads* na GPU: 1/256
- Nº de iterações: 400

5.3.2 Tempo de processamento

A partir das 30 execuções de cada versão/configuração do algoritmo SRAD foram calculadas as médias do tempo de processamento em CPU, GPU ou em ambas, sendo apresentadas a seguir pelo gráfico da figura 35. Na versão sequencial as 400 iterações executadas pelo algoritmo SRAD resultaram em um processamento de 67,113s. As demais versões/configurações apresentam tempo de execução variando entre 8,022s para o algoritmo híbrido OpenMP + OpenACC (20% CPU + 80% GPU) e 32,112s para o algoritmo OpenMP.

A execução paralela em GPU com CUDA foi de 12,218s e com OpenACC foi de 9,543s. Os tempos de execução dos algoritmos híbridos OpenMP + CUDA e OpenMP + OpenACC são próximos para as mesmas divisões de processamento entre CPU e GPU, porém o tempo de processamento do algoritmo OpenMP + OpenACC foi menor em todas as configurações. Esse fato ocorreu porque o tempo de resposta do OpenACC foi menor que o tempo de execução de CUDA e o tempo de processamento do OpenMP compilado por PGCC foi menor que o tempo de execução do OpenMP compilado por NVCC.

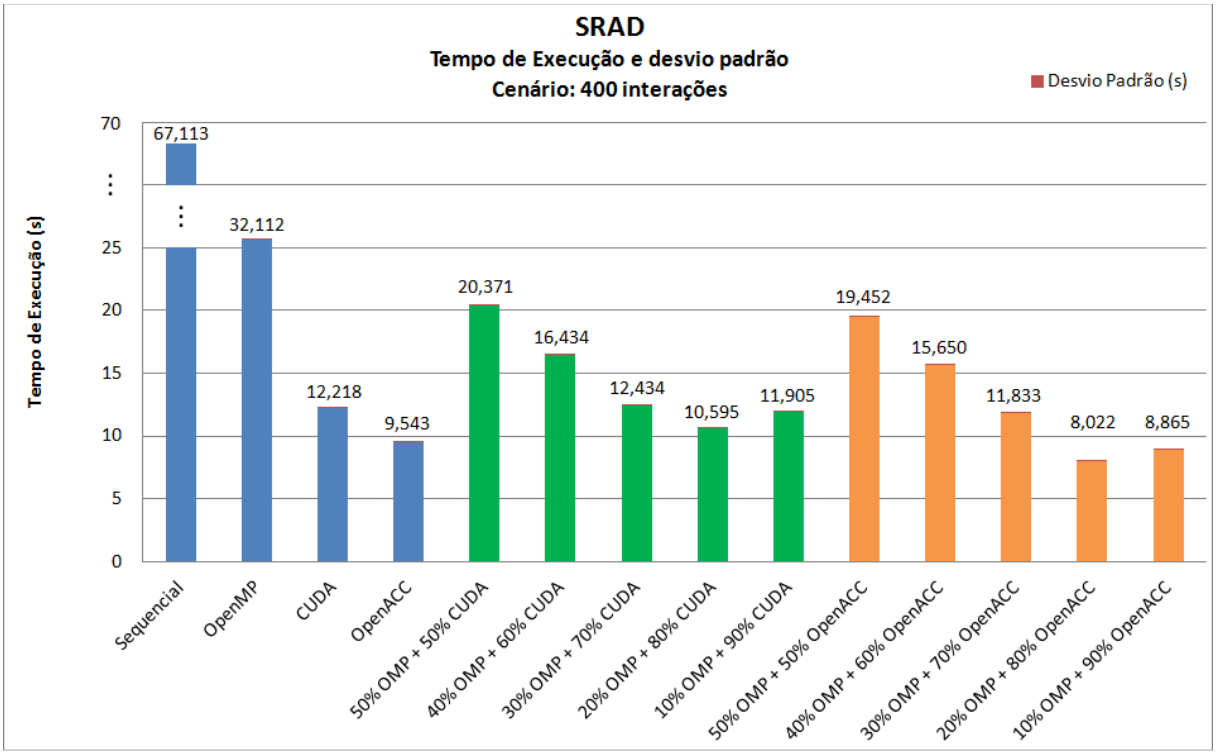

Figura 35 - Tempos de execução do algoritmo SRAD

No melhor cenário obtido pelo algoritmo híbrido OpenMP + OpenACC (20% CPU + 80% GPU) a execução levou 8,022s, inferior ao tempo de processamento das versões CUDA (12,218s) e OpenACC (9,543s). Essa mesma distribuição de trabalho entre CPU e GPU no algoritmo híbrido OpenMP + CUDA resultou em um tempo de 10,595s. A distribuição de 10% CPU + 90% GPU do algoritmo híbrido OpenMP + OpenACC apresentou tempo superior a divisão 20% CPU + 80% GPU, pois o tempo de processamento em GPU superou o tempo de processamento em CPU e em consequência disso o tempo de execução foi maior. Os maiores desvios padrões ficaram por conta dos algoritmos sequencial (0,191s) e híbrido OpenMP + CUDA (50% CPU + 50% GPU) com 0,125s. As demais versões/configurações apresentaram média de desvio padrão de 0,036s.

Os percentuais de confiança das amostras do algoritmo SRAD são apresentados pela tabela 8. Verifica-se que os algoritmos Sequencial e híbrido OpenMP + CUDA (50% CPU + 50% GPU) apresentaram os menores percentuais de confiança, 91,04% e 94,12%, respectivamente, em virtude dessas versões terem apresentado os maiores desvios padrões. As demais variações do algoritmo SRAD obtiveram percentual de confiança das amostras superior a 96%.

Tabela 8 - Percentuais de confiança das amostras do algoritmo SRAD

Algoritmo	% confiança	Algoritmo	% confiança	Algoritmo	% confiança
Sequencial	91,04	50% OMP + 50% CUDA	94,12	50% OMP + 50% ACC	96,61
OpenMP	99,40	40% OMP + 60% CUDA	97,81	40% OMP + 60% ACC	96,86
CUDA	98,67	30% OMP + 70% CUDA	98,64	30% OMP + 70% ACC	98,43
OpenACC	99,49	20% OMP + 80% CUDA	98,22	20% OMP + 80% ACC	98,88
		10% OMP + 90% CUDA	97,83	10% OMP + 90% ACC	98,60

Baseado na amostragem do tempo de execução das versões/configurações do algoritmo SRAD foram calculados os *speedups*, que serão mais bem detalhados a seguir.

5.3.3 *Speedups* obtidos utilizando OpenMP e CUDA

Os *speedups* obtidos pelas configurações do algoritmo SRAD utilizando OpenMP e CUDA, com execução somente em CPU ou GPU, ou através de processamento híbrido são apresentados pela figura 36, a seguir. Ao utilizar OpenMP + CUDA os desempenhos das divisões 20% CPU + 80% GPU e 10% CPU + 90% GPU superaram o maior desempenho obtido ao se executar paralelamente em um único dispositivo. Através do algoritmo em CUDA o *speedup* foi de 5,493 e quando executado em OpenMP o desempenho foi de 2,090. Verificou-se que o desempenho do algoritmo OpenMP foi pequeno, pois no algoritmo SRAD há um grande processamento sequencial na seção de processamento do OpenMP, antes de inicializar a paralelização. Já no algoritmo híbrido OpenMP + CUDA com execução de 20% CPU + 80% GPU e 10% CPU + 90% GPU os *speedups* foram de 6,334 e 5,638, respectivamente.

Verifica-se também que há um aumento do desempenho à medida que mais iterações são executadas na GPU, exceto quando são processadas 10% das iterações em CPU e 90% em GPU, onde o desempenho diminui. Nessa configuração o desempenho é inferior, pois com 80% do processamento em GPU o tempo gasto no dispositivo gráfico é maior do que o tempo de execução em CPU. Através da execução de 90% das iterações em GPU esse tempo aumenta mais, fazendo com que o desempenho fique abaixo da divisão anterior.

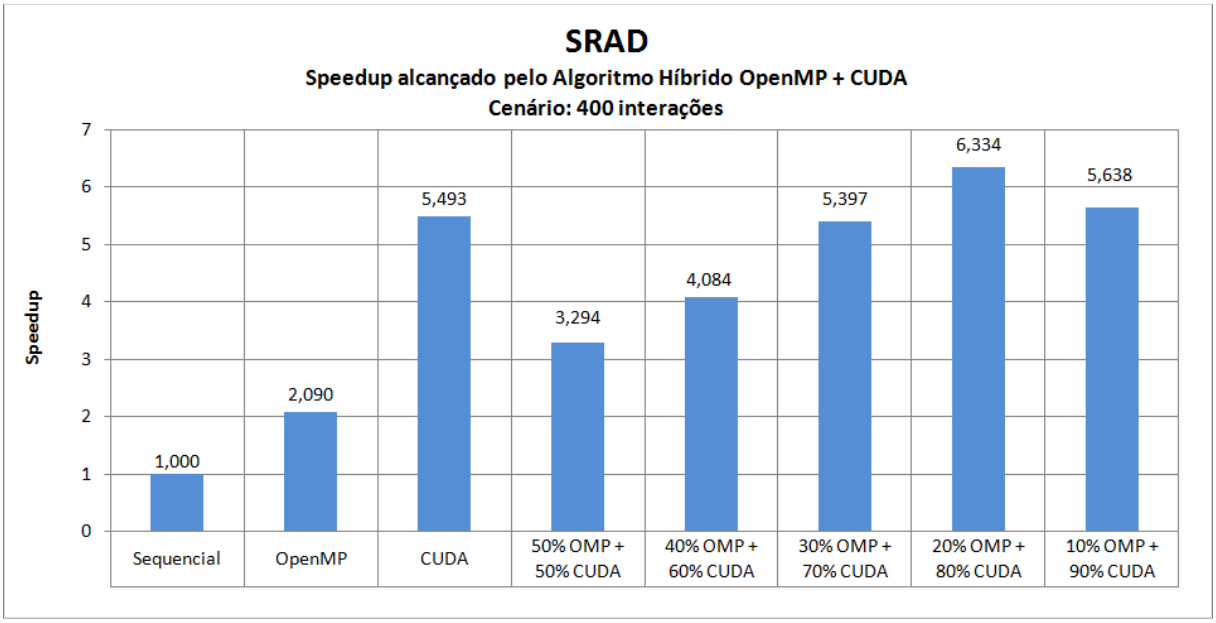

Figura 36 - *Speedups* obtidos pelas versões do algoritmo SRAD utilizando OpenMP e CUDA

Da mesma forma que foram apresentados os *speedups* dos algoritmos que utilizaram OpenMP e CUDA, são apresentados pela subseção 5.3.4 os desempenhos dos algoritmos que utilizaram OpenMP e OpenACC.

5.3.4 *Speedups* obtidos utilizando OpenMP e OpenACC

Os desempenhos obtidos pelo algoritmo híbrido OpenMP + OpenACC nas configurações de 20% CPU + 80% GPU e 10% CPU + 90% GPU além de superarem os *speedups* das execuções paralelas em OpenMP e OpenACC (em um único dispositivo) foram dentre as quatorze versões/configurações do algoritmo SRAD que apresentaram maior *speedup*, 8,366 e 7,570, respectivamente. Analisando o gráfico da figura 37 observa-se que o crescimento do desempenho das configurações do algoritmo híbrido OpenMP + OpenACC é semelhante as configurações do algoritmo híbrido OpenMP + CUDA, apresentado na subseção 5.3.3 anteriormente, porém com maiores *speedups* em todas as configurações de divisão de trabalho entre CPU e GPU.

O principal motivo das configurações do algoritmo híbrido OpenMP + OpenACC apresentarem maior desempenho reside no tempo de execução em CPU e GPU. Tanto em OpenMP quanto em OpenACC o tempo de execução é menor se comparado aos tempos de processamento de OpenMP e CUDA no algoritmo híbrido OpenMP + CUDA. Verificou-se também que o desempenho do OpenMP difere ao ser compilado por PGCC e NVCC (utilizando os mesmos parâmetros de compilação). No algoritmo SRAD o desempenho do

OpenMP em uma compilação através do PGCC foi superior a uma execução do OpenMP gerado pelo NVCC.

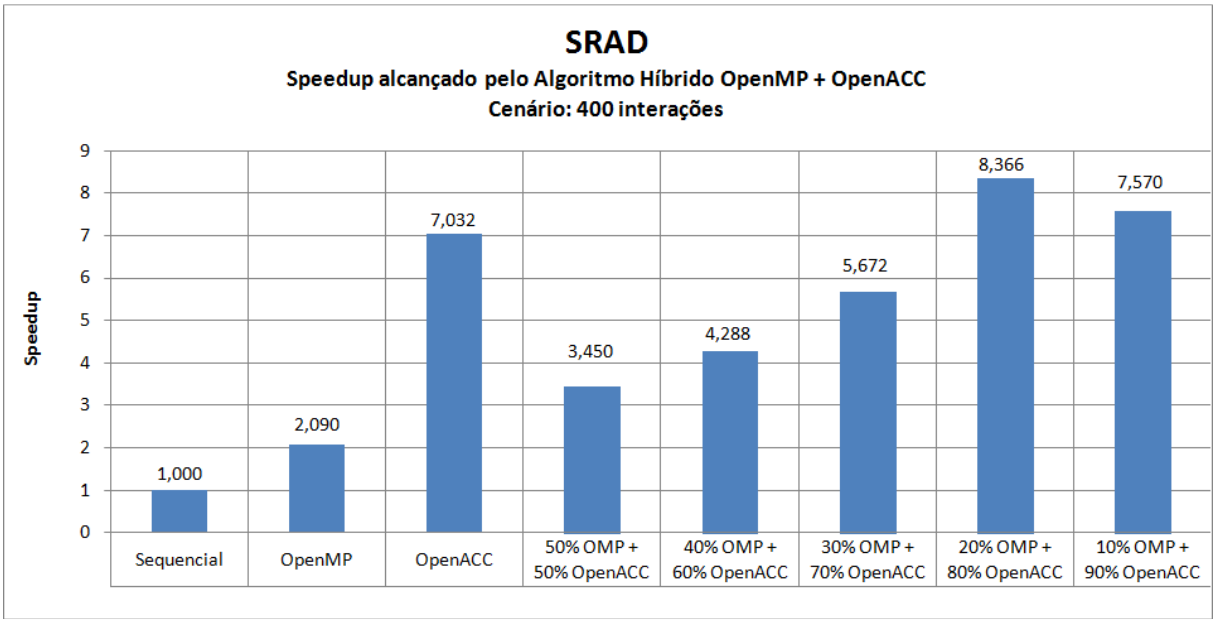

Figura 37 - *Speedups* obtidos pelas versões do algoritmo SRAD utilizando OpenMP e OpenACC

A seguir, na subseção 5.3.5 são apresentadas as divisões das execuções do algoritmo SRAD em um único dispositivo.

5.3.5 Análise da execução dos algoritmos executados em CPU ou GPU

A distribuição do tempo de processamento das etapas de execução das versões sequencial, OpenMP, CUDA e OpenACC do algoritmo SRAD é apresentada pela figura 38. As etapas “Inicialização” e “Término” correspondem respectivamente a alocação dinâmica de memória, inicialização das variáveis, incluindo o carregamento da imagem e a liberação da memória das variáveis utilizadas no algoritmo após o processamento. Para os algoritmos executados em GPU (CUDA e OpenACC) ocorre o envio dos dados da CPU para a GPU antes de iniciar o processamento no dispositivo gráfico. No momento que o processamento na GPU é encerrado ocorre a transferência dos dados da GPU para a CPU.

Observou-se que o tempo da etapa "Inicialização" dos algoritmos compilados por GCC (Sequencial e OpenMP) apresentou menor execução se comparados aos algoritmos compilados por NVCC (CUDA) e PGCC (OpenACC), pois são alocadas e inicializadas também as variáveis utilizadas na GPU. Enquanto que os algoritmos Sequencial e OpenMP

executaram a etapa de "Inicialização" em 0,066s e 0,055s, respectivamente, nos algoritmos CUDA e OpenACC esse mesmo passo superou os 0,300s. O algoritmo sequencial executou por 67,113s, sendo que a parte do cálculo durou 67,04s.

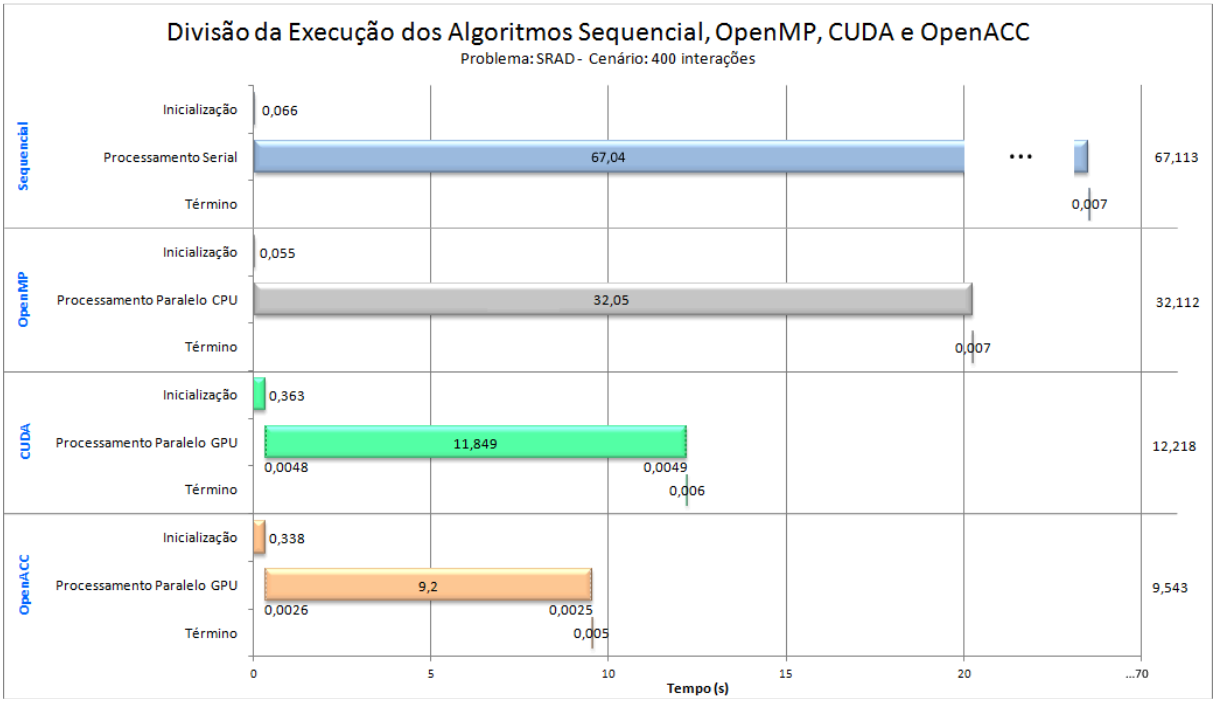

Figura 38 - Divisão da execução das versões sequencial, OpenMP, CUDA e OpenACC do algoritmo SRAD

No algoritmo OpenMP o tempo processamento foi de 32,112s, sendo que a execução paralela custou 32,05s. As execuções dos algoritmos CUDA e OpenACC processaram por 12,218s e 9,543s, respectivamente e o tempo de processamento paralelo foi de 11,849s para o algoritmo CUDA e 9,2s para algoritmo OpenACC. O tempo de execução em GPU computa a transferência de dados nos dois sentidos e o processamento no dispositivo gráfico. A etapa "Término" apresentou tempos entre 0,005s e 0,007s na remoção das variáveis utilizadas. A transferência de informações no algoritmo CUDA, da CPU para GPU durou 0,0048s e da GPU para CPU levou 0,0049s. No algoritmo OpenACC esse mesmo processo custou 0,0026s para o envio dos dados para a GPU e 0,0025s para o retorno das informações para a CPU. O conteúdo da transferência foi um vetor de 4.194.304 posições do tipo de dados *float*.

A seguir, a subseção 5.3.6 irá apresentar a distribuição da execução dos algoritmos híbridos OpenMP + CUDA e OpenMP + OpenACC. Para cada algoritmo serão detalhados os processos de execução em CPU e GPU através da divisão de 50% para cada dispositivo e também da divisão de trabalho que maior desempenho apresentou em cada algoritmo híbrido.

5.3.6 Análise da execução dos algoritmos híbridos

A divisão da execução dos algoritmos híbridos OpenMP + CUDA e OpenMP + OpenACC para 50% das iterações em cada dispositivo e para a melhor divisão de trabalho nos dois algoritmos híbridos (20% CPU + 80% GPU) é apresentada pela figura 39. O processo de inicialização das variáveis nos algoritmos híbridos variou entre 0,352s e 0,362s.

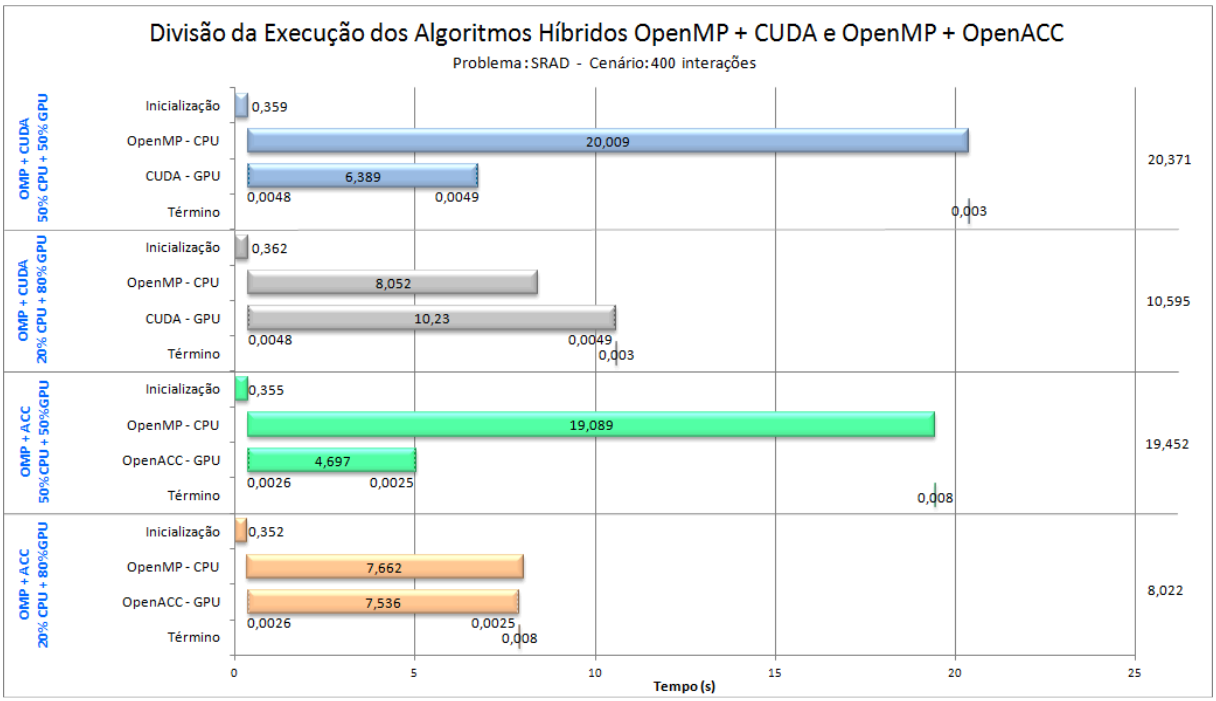

Figura 39 - Divisão da execução das versões híbridas OpenMP + CUDA e OpenMP + OpenACC do algoritmo SRAD

A execução do algoritmo híbrido OpenMP + CUDA (50% CPU + 50% GPU) apresentou tempo 20,009s para o processamento em OpenMP e 6,389s para a execução em CUDA, sendo que o tempo total de processamento foi de 20,371s. Para uma divisão de 20% CPU + 80% GPU o tempo total de execução foi de 10,595s, onde os primeiros 0,362s foram gastos com a alocação das variáveis e atribuição inicial dos dados. Após essa etapa é iniciado o processamento em CPU e GPU. Na CPU através do OpenMP ocorre o processamento de 20% das iterações em 8,052s. Em CUDA o processamento de 80% das iterações leva 10,230s. O tempo gasto na transferência foi de 0,0048s da CPU para GPU e 0,0049s da GPU para a CPU, totalizando 0,0097s.

Já no algoritmo híbrido OpenMP + OpenACC o desempenho do OpenMP foi maior do que no algoritmo híbrido OpenMP + CUDA (19,089s contra 20,009s) para uma divisão de 50% entre os dispositivos, sendo 4,8% mais rápido. O tempo total de processamento foi de

19,452s, sendo que na GPU através do OpenACC a execução levou 4,697s. O melhor cenário do algoritmo híbrido OpenMP + OpenACC também foi ao dividir 20% das iterações em CPU e 80% em GPU. Nesta divisão o tempo de processamento do OpenMP foi de 7,662s, enquanto que a execução do OpenACC durou 7,536s, sendo que o algoritmo apresentou tempo total de 8,022s. O tempo gasto com a transferência de dados entre os dispositivos no algoritmo OpenMP + OpenACC foi de 0,0026s e 0,0025s, da CPU para GPU e da GPU para CPU, conforme já apresentado anteriormente pela subseção 5.3.5, anteriormente. Confrontando os melhores cenários de cada algoritmo híbrido, o desempenho do algoritmo OpenMP + OpenACC foi 24,2% mais rápido do que o algoritmo OpenMP + CUDA.

5.3.7 Eficiência das versões do algoritmo SRAD

Diferente do que aconteceu nos algoritmos RNG e Hotspot, no algoritmo SRAD as eficiências alcançadas sempre foram maiores no algoritmo híbrido OpenMP + OpenACC. Enquanto que no algoritmo híbrido OpenMP + CUDA os percentuais de eficiência variaram entre 55,52% e 83,31%, no algoritmo híbrido OpenMP + OpenACC a eficiência esteve entre 56,94% e 89,27%, conforme ilustra a figura 40.

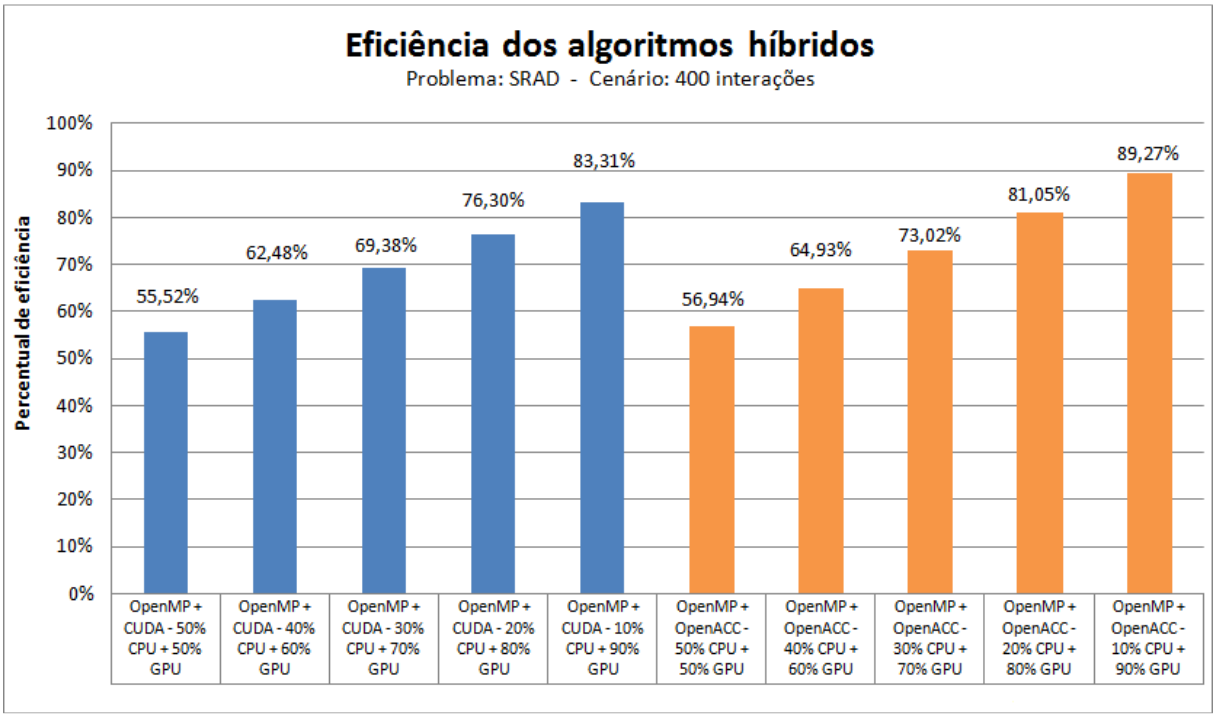

Figura 40 - Eficiência das versões híbridas OpenMP + CUDA e OpenMP + OpenACC do algoritmo SRAD

O percentual de eficiência sempre aumenta na medida em que mais iterações são processadas pela GPU nos dois algoritmos híbridos. A eficiência do algoritmo híbrido OpenMP + OpenACC foi maior pois os desempenhos de OpenMP e de OpenACC, quando compilados por PGCC, foram maiores que os desempenhos de OpenMP e de CUDA ao serem compilados por NVCC, como já relatados na subseção 5.3.6.

Após a apresentação dos resultados dos testes efetuados no algoritmo SRAD, a subseção 5.3.8 irá apresentar as considerações finais sobre a utilização das APIs OpenMP, CUDA e OpenACC nesse algoritmo.

5.3.8 Considerações

A utilização de OpenMP, CUDA e OpenACC no algoritmo SRAD resultou nos seguintes *speedups*: 2,090, 5,493 e 7,032, respectivamente. O algoritmo SRAD quando executado em GPU, da mesma forma que ocorreu nos algoritmos RNG e HotSpot, apresentou maior desempenho se comparado a mesma execução em CPU. Ao utilizar OpenMP + CUDA e OpenMP + OpenACC em duas variações de carga entre CPU e GPU de cada algoritmo obteve-se desempenhos maiores daqueles executados apenas em CPU ou GPU paralelamente. No algoritmo híbrido OpenMP + CUDA o desempenho alcançado nas divisões 20% CPU + 80% GPU e 10% CPU + 90% GPU superou o *speedup* obtido pelo algoritmo OpenMP e pelo algoritmo CUDA com 6,334 e 5,638, respectivamente.

Porém, os maiores desempenhos obtidos no algoritmo SRAD foram alcançados pelas configurações 20% CPU + 80% GPU e 10% CPU + 90% GPU do algoritmo híbrido OpenMP + OpenACC, onde os *speedups* foram 8,366 e 7,570, respectivamente. O motivo que levou o algoritmo híbrido OpenMP + OpenACC a apresentar maior desempenho reside nos tempos de resposta de OpenMP e OpenACC. No algoritmo híbrido OpenMP + OpenACC o desempenho do OpenMP foi maior em relação a mesma execução do OpenMP no algoritmo híbrido OpenMP + CUDA, aproximadamente 4,8%. Utilizando os mesmos parâmetros de compilação verificou-se que a otimização criada pelo compilador PGCC foi mais eficiente da criada pelo compilador NVCC. Além disso, o *speedup* do OpenACC no algoritmo híbrido OpenMP + OpenACC foi em todas as configurações testadas superior ao desempenho de CUDA no algoritmo híbrido OpenMP + CUDA, aproximadamente 36%. A utilização de OpenMP + OpenACC em todas as configurações obteve uma maior eficiência na utilização dos recursos, chegando a um percentual de 89,27% quando executado 10% das iterações em CPU e 90% em GPU. Além disso, no algoritmo SRAD observou-se que a programação híbrida entre CPU

e GPU, tanto na utilização de OpenMP + CUDA quanto de OpenMP + OpenACC superou os desempenhos obtidos ao executar paralelamente em CPU ou GPU, sendo assim, uma alternativa para aumento de desempenho ou para auxílio na resolução de problemas mais complexos.

Após serem apresentados os resultados das execuções dos algoritmos RNG, Hotspot e SRAD serão mais bem detalhados na seção 5.4 os tempos de transferência das informações entre CPU e GPU nos dois sentidos. Esse ponto passa a ser importante à medida que ele pode ser um forte fator de influência no desempenho dos algoritmos que necessitem enviar dados para um coprocessador.

5.4 Transferência de dados entre CPU e GPU

Os tempos de duração da transferência de dados entre CPU e GPU e entre GPU e CPU dos algoritmos híbridos executados neste trabalho são analisados nesta seção. Para cada algoritmo testado são apresentados os tempos gastos na transferência de informações entre os dispositivos e principalmente quanto isso influenciou no processamento na GPU.

5.4.1 RNG

A transferência de informações entre CPU e GPU nos dois sentidos pode ser um forte fator de influência no desempenho de algoritmos. No algoritmo RNG ao utilizar OpenMP + CUDA e OpenMP + OpenACC esse ponto fica mais evidente, onde o percentual total de execução na GPU comprometido com a transferência de dados é superior a 30% no algoritmo híbrido OpenMP + CUDA e superior a 17% no algoritmo híbrido OpenMP + OpenACC, conforme ilustra a tabela 9. O percentual do tempo gasto com a transferência de dados no algoritmo híbrido OpenMP + CUDA foi maior por dois motivos. O primeiro deles se dá porque o tempo na transferência com CUDA nesse algoritmo é maior do que ao utilizar OpenACC. O segundo ponto é o tempo de processamento em CUDA, que é menor que o tempo de execução do OpenACC, fazendo com que o percentual do tempo comprometido com a transferência seja maior no algoritmo híbrido OpenMP + CUDA.

Tabela 9 - Tempos de transferência de dados entre CPU e GPU e entre GPU e CPU no algoritmo RNG

Configuração do algoritmo RNG	Tempo de transferência CPU para GPU (s)	*Throughput* CPU para GPU (GB/s)	Tempo de transferência GPU para CPU (s)	*Throughput* GPU para CPU (GB/s)	Tempo total de transferência (s)	Tamanho aproximado (MB)	Tempo total processamento na GPU (s))	Tempo de cálculo GPU (s)	% do tempo total de execução na GPU comprometido com a transferência de dados
50% OMP + 50% CUDA	0,083	5,36	0,083	5,36	0,166	457	0,543	0,377	30,57%
40% OMP + 60% CUDA	0,101	5,27	0,100	5,34	0,201	549	0,656	0,455	30,64%
30% OMP + 70% CUDA	0,117	5,33	0,116	5,36	0,233	640	0,763	0,530	30,54%
20% OMP + 80% CUDA	0,132	5,40	0,131	5,45	0,263	732	0,868	0,605	30,30%
10% OMP + 90% CUDA	0,149	5,38	0,151	5,30	0,300	823	0,981	0,681	30,58%
50% OMP + 50% OpenACC	0,075	5,94	0,073	6,10	0,148	457	0,838	0,690	17,66%
40% OMP + 60% OpenACC	0,089	5,98	0,087	6,14	0,176	549	1,006	0,830	17,49%
30% OMP + 70% OpenACC	0,104	6,00	0,101	6,16	0,205	640	1,174	0,969	17,46%
20% OMP + 80% OpenACC	0,119	6,00	0,116	6,16	0,235	732	1,342	1,107	17,51%
10% OMP + 90% OpenACC	0,134	5,99	0,131	6,11	0,265	823	1,509	1,244	17,56%

Os tempos de transferência dos dados no algoritmo RNG estão intimamente relacionados a quantidade de informações envolvida na operação. Para o algoritmo RNG são transferidos da CPU para GPU três vetores (*seed*, *arrayX* e *arrayY*) do tipo de dados *int*. A quantidade de transferência (em MB) do algoritmo RNG é proporcional ao percentual de processamento em GPU, sendo que 100% de cada vetor correspondem a 80.000.000 posições. Após o término do processamento em GPU esses três vetores são transferidos da GPU para CPU. O tempo de transferência entre os dispositivos nos algoritmos OpenMP + CUDA e OpenMP + OpenACC é semelhante nos dois sentidos: CPU para GPU e GPU para CPU.

Verificou-se também que o tempo total de transferência de dados entre os dispositivos foi menor em todas as variações de carga de trabalho no algoritmo híbrido OpenMP + OpenACC em relação as mesmas divisões de trabalho do algoritmo híbrido OpenMP + CUDA.

5.4.2 Hotspot

No algoritmo Hotspot são transferidos da CPU para a GPU três vetores do tipo de dados *double* (*temp*, *power* e *result*) com 1.048.576 posições cada, totalizando 24 MB. Após o processamento na GPU o vetor *result* é transferido para a CPU (8 MB). Indiferente do percentual da divisão do processamento entre os dispositivos, a transferência de dados da CPU para a GPU sempre foi de 24 MB e a transferência da GPU para a CPU sempre foi de 8 MB. Os tempos de transferência de dados e o percentual total de execução na GPU comprometido nesse processo são ilustrados pela tabela 10. São apresentadas as informações

do cenário de 50% de processamento para cada dispositivo e a divisão que maior desempenho apresentou em cada algoritmo híbrido.

Tabela 10 - Tempos de transferência de dados entre CPU e GPU e entre GPU e CPU no algoritmo Hotspot

Configuração do algoritmo Hotspot	Tempo de transferência CPU para GPU (s)	*Throughput* CPU para GPU (GB/s)	Tempo de transferência GPU para CPU (s)	*Throughput* GPU para CPU (GB/s)	Tempo total de transferência (s)	Tamanho aproximado (MB)[2]	Tempo total processamento na GPU (s))	Tempo de cálculo GPU (s)	% do tempo total de execução na GPU comprometido com a transferência de dados
50% OMP + 50% CUDA	0,0064	3,63	0,0022	3,46	0,0086	24 + 8	1,305	1,296	0,66%
5% OMP + 95% CUDA	0,0064	3,63	0,0022	3,46	0,0086	24 + 8	2,495	2,486	0,34%
50% OMP + 50% OpenACC	0,0040	5,73	0,0013	5,97	0,0053	24 + 8	2,443	2,437	0,21%
10% OMP + 90% OpenACC	0,0040	5,73	0,0013	5,97	0,0053	24 + 8	4,402	4,396	0,12%

Como a quantidade de informação transferida entre os dispositivos é relativamente pequena (24 MB CPU para GPU e 8 MB GPU para CPU), os tempos de transferência também foram menores se comparados ao algoritmo RNG apresentado anteriormente. No algoritmo híbrido OpenMP + CUDA o tempo médio de envio dos dados para a GPU foi de 0,0064s enquanto que o recebimento das informações após a execução da GPU levou 0,0022s, totalizando 0,0086s. Ao utilizar OpenMP + CUDA o percentual total de execução na GPU comprometido com a transferência de dados foi de 0,66% e 0,34% para as divisões de 50% CPU + 50% GPU e 5% CPU + 95% GPU, respectivamente.

No algoritmo híbrido OpenMP + OpenACC o tempo médio gasto na transferência dos dados para a GPU foi de 0,0040s. Após o processamento, o retorno dos dados para a CPU levou em média 0,0013s. Na configuração de 50% CPU + 50% GPU do algoritmo híbrido OpenMP + OpenACC o percentual total de execução na GPU comprometido com a transferência de dados foi de 0,21%, enquanto que para a divisão de 10% CPU + 90% GPU foi de 0,12%. A transferência de dados entre os dispositivos se mostrou mais eficaz ao utilizar OpenMP + OpenACC. Da mesma forma que ocorreu no algoritmo RNG, no algoritmo Hotspot a utilização de OpenMP + OpenACC apresentou maior *throughput* em relação a OpenMP + CUDA, em média 5,85 GB/s quando utilizado OpenACC e 3,54 GB/s quando empregado CUDA, incluindo o envio e o recebimento dos dados.

[2] A quantidade de dados transferida da CPU para a GPU foi de 24 MB. Após o processamento na GPU foram transferidos 8 MB da GPU para a CPU.

5.4.3 SRAD

Para as divisões de processamento do algoritmo SRAD foram transferidos um vetor do tipo de dados *float* com 4.194.304 posições, totalizando 16 MB. Ao final do processamento na GPU esse mesmo vetor tem seus dados retornados para a CPU. O tempo total de transferência entre os dispositivos no algoritmo híbrido OpenMP + CUDA foi de 0,097s, sendo 0,0048s para o envio dos dados para a GPU e 0,0049s para a retorno das informações para a CPU. O percentual total de execução na GPU comprometido com a transferência de dados foi de 0,15% na divisão 50% CPU + 50% GPU. Na configuração da divisão da execução de maior desempenho do algoritmo híbrido OpenMP + CUDA (20% CPU + 80% GPU) esse percentual foi de apenas 0,09%, conforme apresenta a tabela 11.

Tabela 11 - Tempos de transferência de dados entre CPU e GPU e entre GPU e CPU no algoritmo SRAD

Configuração do algoritmo SRAD	Tempo de transferência CPU para GPU (s)	*Throughput* CPU para GPU (GB/s)	Tempo de transferência GPU para CPU (s)	*Throughput* GPU para CPU (GB/s)	Tempo total de transferência (s)	Tamanho aproximado (MB)	Tempo total processamento na GPU (s))	Tempo de cálculo GPU (s)	% do tempo total de execução na GPU comprometido com a transferência de dados
50% OMP + 50% CUDA	0,0048	3,25	0,0049	3,18	0,0097	16	6,389	6,379	0,15%
20% OMP + 80% CUDA	0,0048	3,25	0,0049	3,18	0,0097	16	10,230	10,220	0,09%
50% OMP + 50% OpenACC	0,0026	5,98	0,0025	6,12	0,0051	16	4,697	4,691	0,10%
20% OMP + 80% OpenACC	0,0026	5,98	0,0025	6,12	0,0051	16	7,536	7,530	0,06%

No algoritmo híbrido OpenMP + OpenACC o tempo consumido na transferência dos 16 MB da CPU para a GPU e da GPU para a CPU foi de 0,051s, sendo pouco mais da metade do tempo gasto pelo algoritmo híbrido OpenMP + CUDA no mesmo processo. O envio dos dados para a GPU consumiu 0,0026s e o retorno das informações para a CPU após o processamento no dispositivo gráfico durou 0,0025s. Na divisão de 50% do processamento para cada dispositivo o percentual total da execução na GPU comprometido com a transferência de dados foi de 0,10%, enquanto que na distribuição de 20% CPU + 80% GPU foi de 0,06%. A média do *throughput* no algoritmo híbrido OpenMP + CUDA foi de 3,21 GB/s e no algoritmo híbrido OpenMP + OpenACC foi de 6,05 GB/s, computando o envio e o recebimento dos dados.

Finalizada a apresentação dos tempos consumidos na transferência de dados entre os dispositivos e o quanto isso influenciou no processamento na GPU, são apresentadas pela subseção 5.4.4 as considerações finais acerca desse tema.

5.4.4 Considerações

Através dos resultados das execuções do algoritmo RNG foi possível verificar que a transferência de dados entre os dispositivos pode ser um forte fator de influência no tempo de execução na GPU. O percentual total de execução na GPU comprometido com a transferência de dados desse algoritmo superou os 30% no algoritmo híbrido OpenMP + CUDA e os 17% no algoritmo híbrido OpenMP + OpenACC. Nos algoritmos Hotspot e SRAD o tempo gasto na transferência de dados pelas versões que utilizavam OpenACC também foi menor daqueles que utilizavam CUDA. Em linhas gerais, o desempenho na transferência de dados entre os dispositivos nos três algoritmos testados sempre foi maior com a utilização da API OpenACC.

Finalizada a apresentação do tempo gasto com a transferência de dados entre os dispositivos e o quanto isso comprometeu no tempo total de execução na GPU, a seguir, são apresentadas pelo capítulo seis as limitações encontradas nos dois modelos de programação propostos (OpenMP + CUDA e OpenMP + OpenACC) que podem influenciar no desempenho ou no desenvolvimento de aplicações híbridas.

6 LIMITAÇÕES NA PROGRAMAÇÃO HÍBRIDA

Durante o desenvolvimento e execução dos algoritmos híbridos foram encontradas algumas limitações envolvendo a utilização de OpenMP + CUDA e OpenMP + OpenACC, que serão detalhadas a seguir.

6.1 Desempenho

Não são todos os problemas que paralelizados de forma híbrida apresentam maior desempenho se comparados a algoritmos paralelos em CPU ou GPU. Muitas vezes as rotinas de divisão, sincronização e envio de dados da CPU para GPU e da GPU para CPU acabam comprometendo o desempenho. Além disso, em alguns problemas há necessidade de alterar parte da lógica para obter-se maior desempenho.

Nos três algoritmos híbridos executados neste trabalho o desempenho de OpenMP + CUDA foi maior em dois deles, nos algoritmos RNG e Hotspot. A programação híbrida utilizando OpenMP + OpenACC teve desempenho superior à OpenMP + CUDA no algoritmo SRAD. O desempenho dos algoritmos híbridos superou o desempenho obtido pelos algoritmos desenvolvidos nas mesmas APIs envolvidas, porém executadas em um único dispositivo, exceto nos algoritmos RNG e Hotspot, onde o desempenho do algoritmo OpenACC foi superior ao obtido pelo modelo híbrido OpenMP + OpenACC. Entretanto, se for analisar o desempenho individual de OpenMP, CUDA e OpenACC nos modelos híbridos propostos, esse foi inferior ao desempenho dos algoritmos paralelos executados nessas mesmas APIs.

A figura 41 ilustra o percentual do desempenho individual das APIs nos algoritmos híbridos RNG, Hotspot e SRAD em comparação ao desempenho dessa mesma API executada em um único dispositivo. Verificou-se que o desempenho do OpenMP no algoritmo RNG através da utilização de OpenMP + OpenACC foi de apenas 37,18% do desempenho do algoritmo paralelo OpenMP. Isso aconteceu porque a execução na CPU da função *randn*, ao encontrar o comando *return*, demorou mais que nas outras versões do algoritmo RNG. Os desempenhos do OpenMP nos algoritmos híbridos foram superiores a 80%. Esse percentual foi atingido porque das oito *threads* destinadas a execução no algoritmo OpenMP, sete foram utilizadas nos algoritmos híbridos para execução em CPU e uma *thread* ficou responsável por alocar, transferir e processar as informações na GPU.

Os desempenhos das APIs CUDA e OpenACC nos algoritmos híbridos foram mais próximos dos desempenhos obtidos por essas mesmas APIs executadas em GPU. O desempenho de CUDA híbrido no algoritmo RNG foi de 92,82%, enquanto que nos algoritmos Hotspot e SRAD o desempenho foi de 97,85% e 92,73%, respectivamente, do desempenho do algoritmo CUDA paralelo. Já o OpenACC híbrido no algoritmo RNG apresentou desempenho de 75% do desempenho do algoritmo OpenACC paralelo em GPU. Nos algoritmos Hotspot e SRAD os desempenhos foram de 85,41% e 97,93%, respectivamente, do desempenho do algoritmo OpenACC paralelo em GPU.

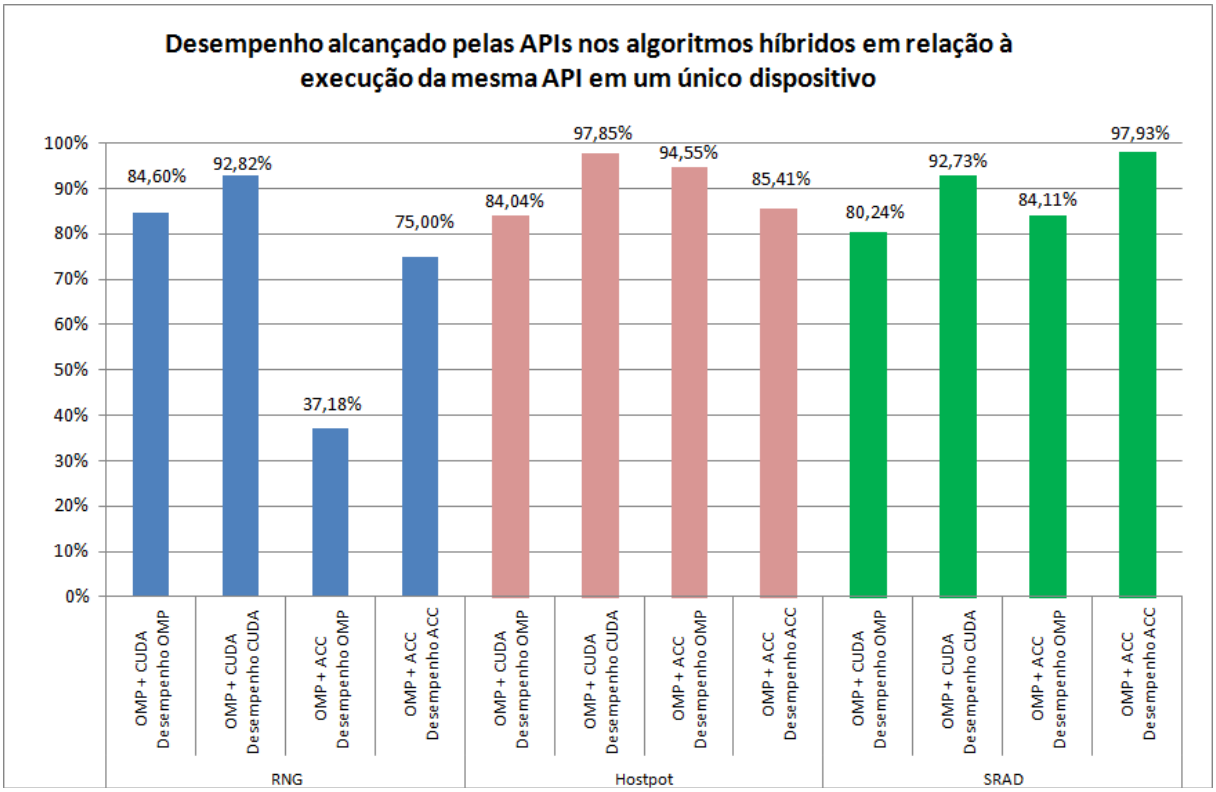

Figura 41 - Desempenho alcançado pelas APIs nos algoritmos híbridos em relação à execução da mesma API em um único dispositivo

Percebeu-se ainda que o desempenho do OpenMP híbrido ao ser compilado por PGCC, utilizando o parâmetro de compilação *–mp* apresentou desempenho maior se comparado aos compiladores GCC e NVCC, ambos utilizando o parâmetro *–fopenmp* nos algoritmos Hotspot e SRAD.

6.2 Tempo de transferência dos dados

Nos algoritmos testados neste trabalho verificou-se que para pequenas quantidades de dados transferidos da CPU para a GPU e da GPU para a CPU o tempo gasto nesse processo foi pequeno e não comprometeu no tempo total de execução na GPU. Esse fato ocorreu nos algoritmos Hotspot e SRAD. Porém, ao transferir um volume maior de dados, como ocorreu no algoritmo RNG, o percentual do tempo total de execução na GPU comprometido com a transferência de dados foi superior a 30% no algoritmo híbrido OpenMP + CUDA e superior a 17% no algoritmo híbrido OpenMP + OpenACC. Nesse algoritmo, para uma divisão de 10% CPU + 90% GPU foram transferidos aproximadamente 823 MB em cada sentido. Esse ponto evidencia que a transferência de dados entre CPU e GPU pode ser um forte fator de influencia no desempenho de algoritmos que utilizam a GPU como dispositivo de processamento.

Indiferente da quantidade de dados e do sentido (CPU para GPU ou GPU para CPU), o processo de transferência de informações entre os dispositivos sempre foi menor utilizando OpenACC nos algoritmos testados neste trabalho.

6.3 Decomposição do problema

A forma como um problema é decomposto para ser processado em um ou mais dispositivos é um ponto que pode se tornar trabalhoso, visto que alguns fatores devem ser observados. Para que um maior desempenho possa ser obtido, aspectos como tamanho de tarefas, sincronização, *deadlock* e transferência de dados devem ser levados em consideração no desenvolvimento de um algoritmo paralelo. Em algumas oportunidades uma má gerência dessas questões pode acarretar em um desempenho abaixo do esperado. Além disso, evitar que em pequenas quantidades de processamento haja muita comunicação (granularidade fina) tende a apresentar melhores oportunidades de aumento de desempenho.

Neste trabalho, o balanceamento de carga entre os dispositivos nos algoritmos híbridos mostrou que um maior trabalho em favor da GPU apresentou maior desempenho. Já a transferência de dados em pequenas quantidades pouco comprometeu na execução dos algoritmos, porém, em quantidades maiores auxiliou para que o desempenho diminuísse.

6.4 Código fonte

Para a divisão do processamento entre CPU e GPU nos algoritmos híbridos OpenMP + CUDA e OpenMP + OpenACC foi criada uma região paralela OpenMP. Nessa região, foi criada uma diretiva *sections* (*#pragma omp sections*) e dentro dela foram criadas duas novas seções através da diretiva *section* (*#pragma omp section*). A primeira *section* foi responsável por executar o código paralelo na GPU, enquanto que a segunda *section* dividiu o processamento na CPU. Na *section* que corresponde a execução paralela em GPU não foi possível mesclar instruções OpenMP e OpenACC, pois o compilador PGCC não reconheceu as diretivas do OpenACC dentro de uma região OpenMP.

A solução adotada foi criar uma função que continha as instruções OpenACC para execução paralela em GPU. Na região OpenMP que ocorre a divisão do processamento entre CPU e GPU essa função foi chamada. A partir disso, o compilador PGCC reconheceu as diretivas OpenMP e OpenACC, reproduzindo as funções adequadamente. Nos códigos fonte dos algoritmos híbridos OpenMP + CUDA esse problema não ocorreu.

6.5 Produtividade

Por ser um modelo baseado em diretivas e também por assemelhar-se ao OpenMP, a API OpenACC apresentou-se mais produtiva do que a API CUDA, tanto nos algoritmos paralelos para GPU quanto nos algoritmos híbridos testados neste trabalho. Esse ponto está intimamente relacionado com a possibilidade do OpenACC incluir várias funcionalidades dentro de uma mesma diretiva, como por exemplo, a paralelização de um comando de repetição unida às diretivas de envio e recebimento de dados, o que diminuiu o número de instruções e agilizou o desenvolvimento. Apesar de CUDA ser um modelo mais maduro, pois está a mais tempo disponível, suas instruções foram em maior número, mais burocráticas e consequentemente mais propensa a erros.

6.6 Controle

O fato da API OpenACC ser mais produtiva que a API CUDA traz consigo um preço. Há uma perda do controle sobre o código gerado em aplicações que fazem uso de diretivas, como por exemplo, o OpenACC. As diretivas do OpenACC apresentam funcionalidades que podem ser adicionadas ou configuradas no código fonte, porém elas não permitem um ajuste

que possibilite uma maior otimização das instruções. Além do mais, na API OpenACC o paralelismo ocorre de forma implícita, não cabendo ao programador a divisão e o escalonamento das tarefas. Isso impossibilita em algumas oportunidades a geração de código mais otimizado por parte do programador e que possa, de alguma forma, resultar em um maior desempenho.

6.7 Diversidade de plataformas

OpenACC tem a seu favor a independência de plataforma, o que possibilita a sua execução em uma diversidade de dispositivos. Esse fato passa a ser importante, pois se tem mais alternativas à utilização de diferentes dispositivos, com diferentes características e capacidades de processamento, permitindo uma maior avaliação de custo-benefício. Além disso, possibilita portar instruções para diversas arquiteturas de GPUs, sem alterações no código fonte. CUDA, em contrapartida restringe a execução de código paralelo apenas para as GPUs fabricadas pela NVIDIA. Desenvolver algoritmos híbridos com OpenACC passa a ser um ponto positivo pela diversidade de plataformas e pela portabilidade do código.

6.8 Considerações

Neste capítulo foram apresentadas algumas limitações que influenciaram no desempenho e no desenvolvimento dos dois modelos de programação híbrida propostos neste trabalho, OpenMP + CUDA e OpenMP + OpenACC. Como limitações de desempenho, citou-se que os *speedups* das APIs utilizadas na programação híbrida sempre foram inferiores aos *speedups* alcançados pela mesma API, porém executada em um único dispositivo. Além disso, o tempo de transferência dos dados entre os dispositivos pode ser um forte fator de influência no desempenho de aplicações para GPU. A forma como um problema deve ser decomposto também foi apontada como um fator importante na busca por um maior desempenho. Como limitações de desenvolvimento foram apresentadas a organização necessária no código fonte para que houvesse o reconhecimento das diretivas OpenMP e OpenACC unidas, bem como questões de produtividade, controle do código fonte e de diversidade de plataformas.

Encerrada a apresentação dos resultados deste trabalho, a seguir, o capítulo sete irá apresentar a conclusão, bem como as sugestões de trabalhos futuros.

7 CONCLUSÃO

Como encerramento deste trabalho, o presente capítulo apresenta as conclusões e as sugestões de trabalhos futuros.

7.1 Conclusão

Este trabalho inicialmente almejou verificar se as facilidades e a produtividade que a API OpenACC introduz também refletiam positivamente no desempenho de algoritmos híbridos em CPU e GPU, através do modelo OpenMP + OpenACC se comparado ao modelo OpenMP + CUDA. Além disso, também objetivava apresentar as limitações nos dois modelos de programação híbrida que pudessem influenciar no desempenho ou no desenvolvimento de aplicações. Para isso, foram desenvolvidos para três dos algoritmos do *benchmark* Rodina, RNG, Hotspot e SRAD versões dos algoritmos híbridos OpenMP + CUDA e OpenMP + OpenACC, sendo os três algoritmos testados em diferentes configurações de divisão de trabalho entre CPU e GPU. Nesses algoritmos a API OpenMP foi a responsável pelo paralelismo em CPU, enquanto que CUDA e OpenACC foram os executores do processamento paralelo em GPU.

No algoritmo RNG, a programação híbrida através de OpenMP + CUDA teve desempenho de aproximadamente 1,5 vezes maior do que se utilizando OpenMP + OpenACC. Ao se comparar o desempenho na GPU, OpenACC híbrido foi 64,8% do desempenho de CUDA híbrido. Verificou-se que a execução do OpenMP no algoritmo híbrido OpenMP + OpenACC teve *speedup* inferior (aproximadamente 2,2 vezes), contribuindo para que esse algoritmo não apresentasse um desempenho maior.

No problema Hotspot, apenas no algoritmo híbrido OpenMP + CUDA obteve-se um desempenho que superasse o desempenho de um algoritmo com execução em um único dispositivo. Ao dividir 5% das iterações em CPU e 95% em GPU (*speedup* de 40,830) o algoritmo híbrido OpenMP + CUDA superou o desempenho obtido pelo algoritmo CUDA (maior *speedup* executando em uma única API) em aproximadamente 1,6%. Em nenhuma das variações da divisão de trabalho do algoritmo híbrido OpenMP + OpenACC se conseguiu superar o desempenho da versão OpenACC, sendo que o seu desempenho foi de 95,3% do desempenho do algoritmo OpenACC executado apenas na GPU.

Dos três algoritmos testados neste trabalho, apenas no SRAD o desempenho utilizando OpenMP + OpenACC foi superior ao desempenho do algoritmo híbrido OpenMP + CUDA. O desempenho do OpenACC no algoritmo híbrido OpenMP + OpenACC foi aproximadamente 36% superior ao desempenho de CUDA no algoritmo híbrido OpenMP + CUDA.

Algumas limitações nos modelos de programação híbrida OpenMP + CUDA e OpenMP + OpenACC propostos neste foram encontradas, a citar: desempenho, tempo de transferência de dados entre os dispositivos, decomposição do problema, divisão do código fonte, produtividade, perda de controle do código fonte e disponibilidade de plataformas. No primeiro ponto, nos três algoritmos híbridos executados neste trabalho o desempenho de OpenMP + CUDA foi maior em dois deles, no algoritmo RNG e no algoritmo Hotspot. A programação híbrida utilizando OpenMP + OpenACC teve desempenho superior à OpenMP + CUDA apenas no algoritmo SRAD. Em todos os algoritmos híbridos ao analisar separadamente o desempenho de cada API envolvida, esse sempre foi inferior ao desempenho dessa mesma API paralelizada em um único dispositivo (CPU ou GPU).

Através das análises realizadas foi possível ver que a transferência de dados entre os dispositivos pode ser um fator de grande influência no desempenho de algoritmos que executam instruções na GPU, tornando-se um ponto de limitação. No algoritmo RNG, por exemplo, mais de 30% do tempo total de processamento na GPU ficou por conta da transferência de dados ao se utilizar OpenMP + CUDA. No algoritmo híbrido OpenMP + OpenACC esse percentual foi superior a 17%. O desempenho na transferência de dados foi maior ao utilizar o OpenACC em todos os algoritmos testados.

A decomposição do problema foi outra limitação abordada. Nela é importante avaliar alguns aspectos como: tamanho das tarefas, sincronização, *deadlock* e transferência de dados para que ocorram mais possibilidades de aumento de desempenho. Ainda nas limitações encontradas, na questão do código fonte não foi possível mesclar instruções OpenACC dentro de regiões OpenMP, pois o compilador PGCC não reconheceu as diretivas do OpenACC. Para solucionar isso foi criada uma função externa à região do OpenMP contendo as instruções OpenACC, sendo essa função chamada na região OpenMP. Nos códigos fonte dos algoritmos híbridos OpenMP + CUDA esse problema não ocorreu.

Para finalizar as limitações, a API CUDA neste trabalho apresentou-se menos produtiva que a API OpenACC, tanto nos algoritmos paralelos para GPU quanto nos algoritmos híbridos. Através da possibilidade de configuração de diferentes funcionalidades em uma mesma diretiva, nos algoritmos desenvolvidos com OpenACC houve uma redução do número de instruções, agilizando o desenvolvimento se comparado a utilização de CUDA.

Também, através do OpenACC são menores as alterações para transformar um código fonte sequencial para paralelo. A API CUDA apesar ser um modelo mais maduro, pois se encontra a mais tempo disponível, apresentou na maioria das vezes maior número de instruções, tornando o código fonte mais burocrático e propenso a erros. Em contrapartida, as facilidades no desenvolvimento através de diretivas influenciam na perda do controle do código gerado, diminuindo as possibilidades do programador otimizar a aplicação, podendo limitar em alguns casos um maior desempenho. Como benefício, API OpenACC tem a seu favor a independência de plataforma, o que possibilita a sua execução em uma diversidade de dispositivos, diferente de CUDA que permite apenas a execução de código paralelo para as GPUs desenvolvidas pela NVIDIA.

Após os testes e análises realizadas foi possível verificar que a programação paralela híbrida pode ser uma boa alternativa para o ganho de desempenho. Quando há um adequado equilíbrio de processamento entre CPU e GPU aumenta-se a possibilidade de se alcançar um maior *speedup*. Os modelos híbridos propostos neste trabalho apresentaram bons resultados, conseguindo superar o desempenho dos algoritmos executados paralelamente em um único dispositivo, exceto nos algoritmos RNG e Hotspot ao utilizar OpenMP + OpenACC, porém através de um esforço maior na programação. Os desempenhos de CUDA e OpenACC, além de superarem o *speedup* do OpenMP evidenciaram que a capacidade de processamento das GPUs tem aumentado e esse dispositivo tem sido uma alternativa interessante para auxiliar na execução de tarefas.

Ao finalizar as conclusões deste trabalho são apresentadas as sugestões de trabalhos futuros.

7.2 Sugestões de trabalhos futuros

Ao finalizar este trabalho são sugeridas as seguintes propostas de trabalhos futuros:

- Programação híbrida em CPU e GPU com os modelos OpenMP + CUDA, OpenMP + OpenACC e OpenMP + OpenCL, como forma de verificar os desempenhos das três principais APIs de programação para GPU;
- Programação híbrida em CPU e GPU utilizando OpenMP e OpenCL para execução em CPU e CUDA, OpenACC e OpenCL (quando não utilizado em CPU) para execução em GPU, com a intenção de analisar o desempenho das APIs envolvidas;

- Programação híbrida em CPU e GPU utilizando múltiplas GPUs com o objetivo de avaliar o desempenho de OpenMP, CUDA, OpenACC e OpenCL;
- Programação híbrida em CPU ou GPU e dispositivo acelerador, como por exemplo, o Intel Xeon Phi, utilizando OpenMP, OpenCL e OpenACC com a finalidade de avaliar o desempenho das APIs.

REFERÊNCIAS

BARNEY, B. **Introduction to Parallel Computing.** Disponível em: < https://computing.llnl.gov/tutorials/parallel_comp/>. Acesso em: 19 mai. 2013b.

BARNEY, B. **OpenMP.** Disponível em: <https://computing.llnl.gov/tutorials/openMP/>. Acesso em: 27 abr. 2013a.

CHATAIN, L. P. **Hybrid Parallel Programming - Evaluation of OpenACC.** Porto Alegre: UFRGS, 2012.

EIJKHOUT, V,; CHOW, E.; GEIJN, R. V. de. **Introduction to High Performance Scientific Computing.** Disponível em: <http://tacc-web.austin.utexas.edu/veijkhout/public_html/Articles/EijkhoutIntroToHPC.pdf>. Acesso em: 19 mai. 2013.

GASTER, B.; HOWES, L.; Kaeli, D. R.; MISTRY, P.; SCHAA, D. **Heterogeneous Computing with OpenCL.** Waltham, USA: Morgan Kaufmann, 2012.

GEBALI, F. **Algorithms and Parallel Computing.** New Jersey: Wiley, 2011.

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

HIRATA, H. V. **Investigando o uso de GPUs para aplicações de Bioinformática.** São José do Rio Preto: UNESP, 2011.

IKEDA, P. A. **Um estudo do uso eficiente de programas em placas gráficas.** São Paulo: USP, 2011.

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

LASTOVETSKY, A. L.; DONGARRA, J. J. **High Peformance Heterogeneous Computing.** Wiley: New Jersey, 2009.

LAVALLEE, P. F.; WAUTELET, P. **Hybrid MPI-OpenMP Programming.** Version 1.10. Institute for Development and Resources in Intensive Scientific Computing, 2013.

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

LU, F.; SONG, J.; YIN, F.; ZHU, X. **Performance evaluation of hybrid programming patterns for large CPU/GPU heterogeneous clusters.** Computer Physics Communications 183, 2012.

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 dez. 2012.

MATLOFF, N. **Programming on Parallel Machines.** Disponível em: <http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf>. Acesso em: 27 abr. 2013.

MATTSON, T. G.; SANDERS, B. A.; MASSINGILL, B. L. **Patterns for Parallel Programming.** Pearson, 2005

MULLA, S.; **Directives Based Programming of GPU Accelerated Systems.** University of Edinburgh. MSc in High Performance Computing, 2011.

NVIDIA. **CUDA Toolkit Documentation.** Disponível em: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>. Acesso em 19 jun. 2014b.

NVIDIA. **NVIDIA Visual Profiler.** Disponível em: <https://developer.nvidia.com/nvidia-visual-profiler/>. Acesso em 14 abr. 2014a.

NVIDIA. **Plataforma de Computação Paralela.** Disponível em: <http://www.nvidia.com.br/object/cuda_home_new_br.html>. Acesso em: 21 abr. 2013.

OPENACC. **The OpenACC™ Application Programming Interface.** Version 2.0. Disponível em: <http://openacc.org/sites/default/files/OpenACC-2.0-draft.pdf>. Acesso em 28 abr. 2013.

OPENCL. **The OpenCL Specification**. Version 2.0. Disponível em: <https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf>. Acesso em: 25 mai. 2014.

OPENMP. **OpenMP Application Program Interface** – Version 4.0 – July 2013. Disponível em: < http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>. Acesso em 12 ago. 2013.

OWENS, J. D.; LUEBKE, D.; GOVINDARAJU, N.; HARRIS, M.; KRÜGER, J.; LEFOHN, A. E.; PURCELL, T. J.; **A Survey of General-Purpose Computation on Graphics Hardware.** Computer Graphics Forum, 2007.

PACHECO, P. **An Introduction to Parallel Programming.** Burlington: Morgan Kaufmann, 2011.

PGI. **Compilers & Tools for HPC.** Disponível em: <http://www.pgroup.com/lit/presentations/pgi_2014.pptx>. Acesso em: 01 mai. 2014.

PGI. **PGI Accelerator Compilers With OpenACC Directives.** Disponível em: <http://www.pgroup.com/resources/accel.htm>. Acesso em 28 abr. 2013a.

PGI. **PGI AcceleratorTM Compilers OpenACCGetting Started Guide.** Version 13.2. Disponível em: <http://www.pgroup.com/doc/openACC_gs.pdf>. Acesso em: 29 abr. 2013b.

RANDOM. **Introduction to Randomness and Random Numbers**. Disponível em: <http://www.random.org/randomness/>. Acesso em: 18 dez. 2013.

RAUBER, T.; RÜNGER, G. **Parallel Programming for Multicore and Cluster Systems.** Springer, 2010.

REYES, R.; LOPEZ, I.; FUMERO, J. J.; DE SANDE, F. **A Comparative Study of OpenACC Implementations.** XXIII Jornadas de Paralelismo, 2012.

RIBEIRO, N. S.; FERNANDES, L. G. L. **Programação Híbrida: MPI e OpenMP.** Disponível em: <http://www.inf.pucrs.br/gmap/pdfs/Neumar/Artigo-IP2%20-%20Neumar%20Ribeiro.pdf>. Acesso em: 27 abr. 2013.

RODINIA. **Rodinia:Accelerating Compute-Intensive Applications with Accelerators**. Disponível em: <https://www.cs.virginia.edu/~skadron/wiki/rodinia>. Acesso em: 18 dez. 2013.

SENA, M. C. R.; COSTA, J. A. C. **Tutorial OpenMP C/C++.** Programa Campus Ambassador HPC - SUN Microsystems, Maceio, 2008.

SILVA, L. N. **Modelo Híbrido de Programação Paralela para uma Aplicação de Elasticidade Linear Baseada no Método dos Elementos Finitos.** Brasília: UnB, 2006.

STALLINGS, W. **Computer Organization and Architecture**: Designing for Performance. Eighth Edition. New Jersey: Prentice Hall, 2010.

TANENBAUM, A. S. **Organização Estruturada de Computadores.** 5ª Edição. São Paulo: Pearson Prentice Hall, 2007.

TSUCHIYAMA, R.; NAKAMURA, T.; IIZUKA, T.; ASAHARA, A.; MIKI, S. **The OpenCL Programming Book - Parallel Programming for MultiCore CPU and GPU.** Disponível em: <http://www.fixstars.com/en/opencl/book/OpenCL_Book_sample.pdf>. Acesso em: 09 dez. 2012.

WIENKE, S.; SPRINGER, P.; TERBOVEN, C.; MEY, D. **OpenACC - First Experiences with Real-World Applications**. Euro-Par 2012, LNCS, Springer Berlin/Heidelberg(2012), 2012.