Sociedade Educacional Três de Maio - SETREM
Bacharelado em Sistemas de Informação

CHARLES MICHAEL STEIN

PROGRAMAÇÃO PARALELA PARA GPU EM APLICAÇÕES DE PROCESSAMENTO DE *STREAM*

Três de Maio

2018

CHARLES MICHAEL STEIN

PROGRAMAÇÃO PARALELA PARA GPU EM APLICAÇÕES DE PROCESSAMENTO DE *STREAM*

Trabalho de Conclusão de Curso
Sociedade Educacional Três de Maio
Faculdade Três de Maio
Sistemas de Informação

Orientador:
Prof. Dr. Dalvan Griebler

Três de Maio
2018

TERMO DE APROVAÇÃO

CHARLES MICHAEL STEIN

PROGRAMAÇÃO PARALELA PARA GPU EM APLICAÇÕES DE PROCESSAMENTO DE *STREAM*

Relatório aprovado como requisito parcial para obtenção do título de **Bacharel em Sistemas de Informação** concedido pela Faculdade de Sistemas de Informação da Sociedade Educacional Três de Maio, pela seguinte Banca examinadora:

Orientador: Prof. Dalvan Jair Griebler, Dr.
Faculdade de Sistemas de Informação da SETREM

Prof. Samuel Camargo de Souza, M. Sc.
Faculdade de Sistemas de Informação da SETREM.

Prof. Claudio Schepke, Dr.
Faculdade de Sistemas de Informação da SETREM.

Profa. Vera Lúcia Lorenset Benedetti, M.Sc.
Coordenação do Curso Bacharelado em Sistemas de Informação
Faculdade de Sistemas de Informação da SETREM.

Três de Maio, 20 de Junho de 2018.

RESUMO

Aplicações de processamento de *stream* são utilizadas em diversas áreas. Elas geralmente demandam de processamento em tempo real e possuem uma alta carga computacional. A paralelização deste tipo de aplicação pode hipoteticamente aumentar o desempenho em aplicações de processamento de *stream*. Este trabalho apresenta o estudo e implementação de software paralelo de aplicações de processamento de *Stream* em GPU. Aplicações de diferentes áreas foram escolhidas e paralelizadas para CPU e GPU, realizando experimentos e análises sobre os resultados atingidos. Foram paralelizadas as aplicações Filtro Sobel, LZSS, *Dedup* e *Black-Scholes*. O filtro *Sobel* não obteve um aumento de desempenho, enquanto o *LZSS*, *dedup* e *black-scholes* obtiveram um *Speedup* de 36x, 13x e 6.9x respectivamente. Além do desempenho, foram medias as linhas de código das implementações nas bibliotecas CUDA e OpenCL a fim de medir a intrusão de código. Os testes realizados apresentaram que em algumas aplicações, a utilização de GPU é vantajosa, enquanto em outras não há ganhos significativos quando comparado a versões paralelas em CPU.

Palavras-chave: Sistemas de Informação, Programação Paralela, Processamento de *Stream*, GPU.

ABSTRACT

Stream processing applications are used in many areas. They usually require real-time processing and have a high computational load. The parallelization of this type of application is necessary. The use of GPUs can hypothetically increase the performance of this stream processing applications. This work presents the study and parallel software implementation for GPU on stream processing applications. Applications of different areas were chosen and parallelized for CPU and GPU. A set of experiments were conducted and the results achieved were analyzed. Therefore, the Sobel, LZSS, Dedup, and Black-Scholes applications were parallelized. The Sobel filter did not gain performance, while the LZSS, Dudup and Black-Scholes obtained a Speedup of 36x, 13x and 6.9x respectively. In addition to performance, the source lines of code from the implementations with CUDA and OpenCL libraries were measured in order to analyze the code intrusion. The tests performed showed that in some applications the use of GPU is advantageous, while in other applications there are no significant gains when compared to the parallel versions in CPU.

Keywords: Information Systems, Parallel Programming, Stream Processing, GPU.

LISTA DE FIGURAS

1 Visão geral de aplicações de processamento de *Stream* 18

2 Crescimento dos *gigaflops* de CPU e GPU 20

3 Processamento de imagem no YOLO 27

4 *Estágios do pipeline do dedup* 28

5 Hierarquia de memória da CPU 37

6 *Caches* dos núcleos da CPU 37

7 Arquitetura do *Stream Multiprocessor* de uma GPU Pascal . . . 39

8 Diagrama da arquitetura da GPU (G80/GT200) 40

9 Sistemas de memória heterogênea 42

10 Execução de *pipeline* com 5 elementos 46

11 Padrão paralelo *Map* 47

12 Padrão paralelo *Reduce* 48

13 Padrão paralelo *Stencil* 49

14 Arquitetura de camadas do *FastFlow* 52

15 Grafos de atividade *SPar* 56

16 *Fork* e *Join* em *pthreads* 58

17 Produtividade de bibliotecas paralelas 62

18 Desempenho de *Dedup* e *Ferret* com bibliotecas para CPU . . 63

19 Desempenho de Bzip2 com bibliotecas para CPU 63

20 Multiplicação de matrizes 10x10 78

21 Multiplicação de matrizes 100x100 79

22 Multiplicação de matrizes 1000x1000 80

23 Multiplicação de matrizes 2000x2000 81

24 *Speedup* das versões otimizadas 81

25 Aplicações utilizadas na análise de produtividade 82

26 Análise de produtividade das aplicações de *benchmark* 83

27 Exemplo de imagem do filtro sobel 91

28 CPU vs GPU 800x600 94

29 CPU vs GPU 3000x2250 95

30 CPU vs GPU 5000x5000 95

31 CPU vs GPU 10000x10000 96

32 Resultados com o uso combinado do paralelismo em CPU e GPU em arquivo. 96

33 Resultados com o uso combinado do paralelismo em CPU e GPU em memória. 97

34 Linhas de código por implementação do filtro Sobel. 98

35 *Pipeline* simples do LZSS 105

36 *Pipeline* de múltiplas GPUs do LZSS 106

37 Tempo total de compressão 108
38 Tempo de busca de maior ocorrência 108
39 *Speedup do LZSS em GPU* 109
40 *Speedup do LZSS por biblioteca* 110
41 Linhas de código por implementação 111
42 *Dedup* Sequencial 113
43 *Dedup* com SPar 113
44 *Dedup* Sequencial com GPU 115
45 Tempo total do *Dedup* sem compressão 117
46 Tempo total do *Dedup* com compressão 118
47 *Speedup* do *Dedup* sem compressão 119
48 *Speedup* do *Dedup* com compressão 119
49 Linhas de código por implementação do dedup 121
50 *Pipeline* do *Black-Scholes* 122
51 Tempo total por implementação do *Black-Scholes* 124
52 *Speedup* por implementação do *Black-Scholes* 124
53 Linhas de código por implementação do *Black-Scholes*. 125

LISTA DE QUADROS

1 Orçamento 24

2 Cronograma 24

3 Trabalhos relacionados 88

4 Resumo do *dataset* do *benchmark* 107

5 Comparação de tempo de busca de maior ocorrência em GPU . 109

LISTA DE CÓDIGOS

Código 1 Exemplo de *stream*. 27
Código 2 Exemplo de *pipeline* em TBB, extraído de McCool, Reinders e Robison (2012). 50
Código 3 *Pipeline* em *FastFlow*. 53
Código 4 *Stream* de dados seguindo o grafo A1, extraído de Griebler (2016). 56
Código 5 *Stream* de dados seguindo o grafo A2, extraído de Griebler (2016). 57
Código 6 *Pipeline* em *pthread* 59
Código 7 Kernel de soma em CUDA. 65
Código 8 Kernel de multiplicação em CUDA. 65
Código 9 Kernel de multiplicação atomica em CUDA. 66
Código 10 Separação de tarefa em blocos. 67
Código 11 Multiplicação de matrizes utilizando *Unified Memory*. 67
Código 12 Multiplicação de matrizes otimizada em CUDA. 68
Código 13 Soma de matrizes em OpenACC 70
Código 14 Multiplicação de matrizes em OpenACC 70
Código 15 Laço da multiplicação de matrizes em OpenACC com redução . . 71
Código 16 Inicialização do OpenCL 73
Código 17 Kernel de Produto OpenCL 74
Código 18 Kernel de Multiplicação de matrizes e OpenCL 74
Código 19 Kernel de Multiplicação de matrizes e OpenCL 75
Código 20 Aplicação com SPar. 93
Código 21 Sobel em CUDA 93

Código 22 Entrada de dados do LZSS. 99
Código 23 Resultado do LZSS 99
Código 24 Pseudocódigo de compressor LZSS 100
Código 25 Pseudocódigo de compressor LZSS em *batch* 102
Código 26 Pseudocódigo de compressor LZSS 103

LISTA DE SIGLAS

API	*Application Programming Interface*
BWT	*Burrows–Wheeler Transform*
GPGPU	*General Purpose Graphics Processing Unit*
DSL	*Domain Specific Language*
EMD	*Earth Mover's Distance*
FPS	*Frames Per Second*
ILP	*Instruction-level Paralelis*m
LZSS	*Lempel–Ziv–Storer–Szymanski*
MIMD	*Multiple Instruction, Multiple Data*
PDE	*Partial Differential Equation*
SIMD	*Single Instruction, Multiple Data*
SIMT	*Single Instruction, Multiple Threads*
SM	*Streaming Multiprocessor*
SP	*Streaming Processing*
SVD	*Singular Value Decomposition*

SUMÁRIO

INTRODUÇÃO 15
CAPíTULO 1: PLANO DE ESTUDO E PESQUISA 17
1.1 TEMA 17
1.1.1 Delimitação do tema 17
1.2 OBJETIVO GERAL 19
1.3 OBJETIVOS ESPECíFICOS 19
1.4 JUSTIFICATIVA 19
1.5 PROBLEMA 21
1.6 HIPÓTESES 21
1.7 METODOLOGIA 21
1.7.1 Abordagem 22
1.7.2 Procedimentos 22
1.7.3 Técnicas 23
1.8 ORÇAMENTO 23
1.9 CRONOGRAMA 24
CAPíTULO 2: BASE TEÓRICA 25
2.1 APLICAÇÕES DE PROCESSAMENTO DE *STREAM* 25
2.2 MEDIDAS DE AVALIAÇÃO 29
2.2.1 Desempenho 29
2.2.1.1 Latência 29
2.2.1.2 Throughput 30
2.2.1.3 Speedup 30
2.2.1.4 Eficiência 30
2.2.2 Produtividade 31
2.2.2.1 COCOMO 31
2.2.2.2 Experimento com pessoas 33
2.2.2.3 Code Churn 34
2.3 ARQUITETURAS PARALELAS 35
2.3.1 CPU 36
2.3.2 GPU 38
2.3.3 Sistemas de Memória Heterogênea 41
2.3.4 Cluster 42

2.4 PROGRAMAÇÃO PARALELA 43
2.4.1 Padrões paralelos 44
2.4.1.1 Pipeline 45
2.4.1.2 Farm 46
2.4.1.3 Map 47
2.4.1.4 Reduce 48
2.4.1.5 Stencil 49
2.4.2 Interfaces de programação paralela para CPUs 49
2.4.2.1 TBB 50
2.4.2.2 FastFlow 51
2.4.2.3 SPar 54
2.4.2.4 PThreads 57
2.4.2.5 Visão geral das principais interfaces para CPU 61
2.4.3 Interfaces de programação paralela para GPUs 64
2.4.3.1 CUDA 64
2.4.3.2 OpenACC 69
2.4.3.3 OpenCL 72
2.4.3.4 Visão geral das principais interfaces para GPU 77
2.5 TRABALHOS RELACIONADOS 83
CAPíTULO 3: EXPERIMENTOS E RESULTADOS 90
3.1 ESCOLHA DE BIBLITECAS E APLICAÇÕES 90
3.2 APLICAÇÕES 90
3.2.1 Filtro Sobel 91
3.2.1.1 Análise dos resultados 93
3.2.2 LZSS 98
3.2.2.1 Paralelização da aplicação 101
3.2.2.2 Paralelização em CPU 105
3.2.2.3 Análise dos resultados 106
3.2.3 Dedup 112
3.2.3.1 Análise de resultados 115
3.2.4 Ferret 120
3.2.5 Black-scholes 121
3.2.5.1 Análise dos resultados 123
3.3 VISÃO GERAL DOS RESULTADOS 125
CONCLUSÃO 128
REFERÊNCIAS 131

INTRODUÇÃO

Aplicações de processamento de *stream* são utilizadas em diversas áreas, as quais envolvem processamento em tempo real e alta carga computacional. Este tipo de aplicação é utilizado em várias áreas, como por exemplo análises de mercado para bolsa de valores, análises em tempo real para sistemas de prevenção de fraudes, análises sísmicas, inteligência de negócios, dentre outros.

Devido a grande quantidade de dados envolvidos, este tipo de aplicação muitas vezes necessita um grande poder computacional para possuir um desempenho aceitável. Para alcançar o desempenho necessário, muitas vezes a indústria recorre ao paralelismo de CPU, através do uso de *multicore* e *clusters*.

Por outro lado, o uso de co-processadores vem sendo muito utilizado em problemas de grande paralelismo no meio científico, enquanto que na indústria ainda são pouco utilizados.

Desta forma, as GPUs, com sua arquitetura e o modelo SIMT, apresentam-se como um co-processador eficiente em aplicações que podem fazer o uso de paralelismo. Muitas aplicações de processamento de *stream* já trabalham com o uso de múltiplos núcleos da CPU, porém a adição da GPU em processos que possam ser paralelizados massivamente pode representar em um grande ganho de desempenho para este tipo de aplicação.

Com isso, este trabalho realizou um estudo do uso de GPUs em aplicações de processamento de *stream*, implementando e analisando o comportamento das aplicações Filtro Sobel, LZSS, *Dedup* e *Black-scholes* em múltiplas interfaces de programação.

O Capítulo 1 apresenta o plano de pesquisa e estudo, o escopo do projeto e como será desenvolvido a pesquisa. O Capítulo 2 apresenta o embasamento teórico, as bibliotecas, aplicações e outros assuntos relacionados ao trabalho. No Capítulo 3 são apresentados os resultados obtidos das aplicações de processamento de *stream* implementadas.

CAPÍTULO 1: PLANO DE ESTUDO E PESQUISA

1.1 TEMA

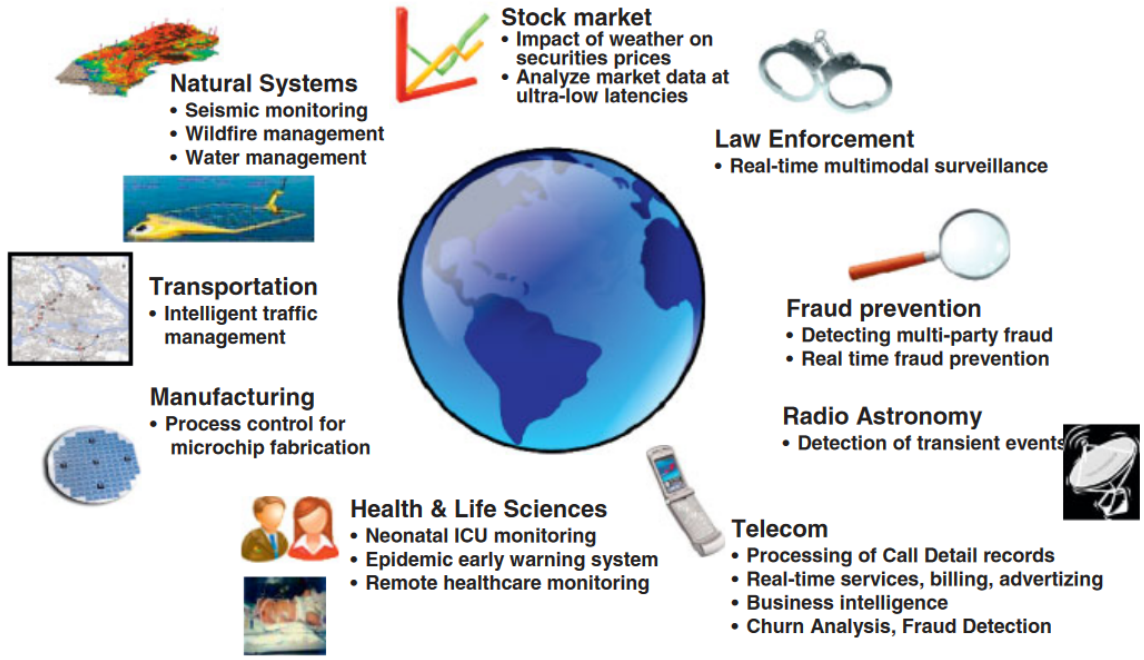
Explorando o Paralelismo das GPUs em Aplicações de Processamento de *Stream*.

1.1.1 Delimitação do tema

Aplicações de processamento de *stream* envolvem o processamento de uma grande quantidade de dados de forma contínua. Conforme ilustra a Figura 1, elas estão em diversas áreas, como análises de mercado na bolsa de valores, sistemas de detecção de eventos sísmicos, sistemas de segurança, dentre outros. Na computação, este tipo de aplicação vem sendo desenvolvida com a utilização exclusiva das CPUs. Em determinadas situações e cargas de trabalho, elas necessitam de computação paralela para responder em tempo real. Para atingir o desempenho necessário para o processamento em tempo real, estas aplicações precisam de muitos recursos computacionais, que podem tornar as aplicações inviáveis se não implementadas de forma eficiente.

Conforme Cook (2013), a GPU serve como uma alternativa para estas aplicações, oferecendo uma alta capacidade computacional para aplicações paralelas quando comparada com a CPU. Desta forma, aplicações paralelas precisam ser paralelizadas para esta arquitetura, para que possam atingir o desempenho necessário por um custo viável.

Figura 1: Visão geral de aplicações de processamento de *Stream*



Fonte: (Turaga et al., 2010, p. 1074)

O trabalho realizou um estudo e exploração de paralelismo de aplicações de processamento de *stream* em GPUs. Foi realizada uma revisão bibliográfica de interfaces de programação paralela (CUDA e OpenCL) usadas para exploração do paralelismo na GPU. Isso serviu de base para aprendizado e o desenvolvimento de aplicações de processamento de *stream*. Além disso, estas aplicações foram estudadas e escolhidas no decorrer do trabalho. Os estudos e análises foram feitos utilizando uma máquina com GPU NVIDIA. Junto ao desenvolvimento das aplicações, foi analisado o ganho de desempenho e programabilidade obtidos com a utilização das GPUs em aplicações de *stream*.

O trabalho foi desenvolvido pelo acadêmico Charles Michael Stein, para o Trabalho de Conclusão de Curso do Curso Bacharelado de Sistemas de Informação e orientado pelo Ph. D. Dalvan Jair Griebler. O período de estudo foi de agosto de 2017 á junho de 2018, sendo este realizado na Sociedade Educacional Três de Maio - SETREM na área específica de Programação Paralela no Laboratório de Pesquisa Avançada para Computação em Nuvem - LARCC.

1.2 OBJETIVO GERAL

O objetivo é explorar o paralelismo das GPUs usando as interfaces/bibliotecas de programação paralela, as quais também foram estudadas e avaliadas no desenvolvimento de aplicações paralelas de processamento de *stream*.

1.3 OBJETIVOS ESPECÍFICOS

- Aprender sobre as bibliotecas de programação paralela CUDA, OpenACC e OpenCL.
- Mapear trabalhos relacionados e elaborar uma síntese comparativa.
- Aprender sobre o domínio de aplicações de processamento de *stream*.
- Usar duas interfaces de programação paralela para paralelizar um conjunto aplicações de processamento de *stream*.
- Avaliação e Comparação do desempenho e produtividade de código na paralelização das aplicações de *stream*.
- Publicar artigo científico referente ao estudo realizado.

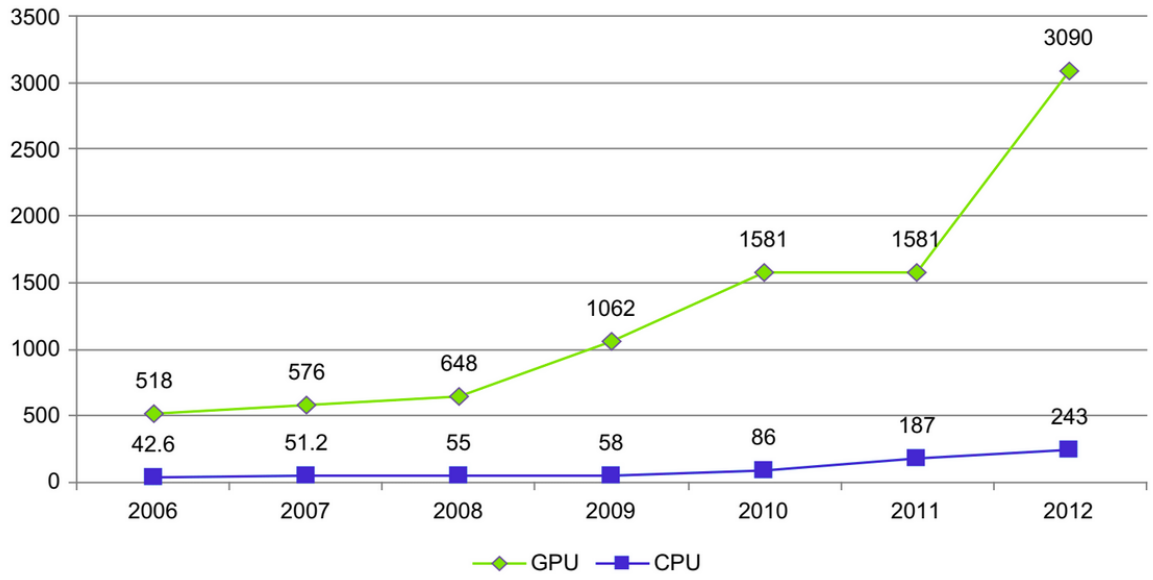
1.4 JUSTIFICATIVA

As aplicações de *streaming* consistem de aplicações que utilizam *pipeline* de dados. Este tipo de aplicação é utilizada em diferentes tipos de computação, como compactação de arquivos, *machine learning* e *streaming* de áudio e vídeo, cada vez mais presente na Web.

Devido à natureza dos dados neste tipo de aplicação, as GPUs podem auxiliar na melhora de desempenho. As GPUs são utilizadas por aplicações gráficas há muito tempo, porém conforme Cook (2013), apenas nos últimos anos é que se iniciou a utilização de placas gráficas para aplicações de propósito geral. Este

crescimento no uso da GPU se dá devido ao crescimento na capacidade de processamento das GPUs, apresentado na Figura 2. Como pode ser visto, o aumento no processamento em GPUs apresentou um crescimento muito maior do que as CPUs a partir de 2009.

Figura 2: Crescimento dos *gigaflops* de CPU e GPU



Fonte: (Cook, 2013, p. 14)

Conforme Cook (2013), a GPU possui um poder de paralelismo muito elevado quando considerado o seu preço. A GPU torna-se então uma boa alternativa para aplicações que exijam maior capacidade de processamento. Enquanto a CPU é utilizada para regiões sequenciais, a GPU executa regiões paralelas de forma massiva. Conforme Munshi et al. (2011), a este processo que faz a utilização da CPU e GPU, chama-se computação heterogênea.

O grande ganho de desempenho obtido pela computação heterogênea é devido a arquitetura paralela das GPUs. Desta forma, existem muitas aplicações de processamento de *stream* que podem ser paralelizadas para o uso de GPU. Com isso, o intuito deste trabalho é explorar o uso de paralelismo em GPU para aplicações de processamento de *stream* através do uso de computação heterogênea.

Segundo Cook (2013), o ganho de desempenho com a utilização de para-

lelismo em GPU é entre 5 a 10 vezes, quando comparado as mesmas aplicações em *multi-thread* na CPU. Desta forma, a lacuna encontrada é a compreensão de qual é o ganho de desempenho com a utilização de paralelismo em CPU e GPU em aplicações de processamento de *stream*.

1.5 PROBLEMA

Como foi apresentado Figura 2, a capacidade de processamento da GPU tem avançado muito rápido. Segundo Hwu (2011), elas podem obter um ganho de desempenho de aproximadamente 10x em comparação com a CPU, porém, em aplicações de processamento de *stream*, não se sabe exatamente qual o ganho obtido. Desta forma, questiona-se: qual é o ganho de desempenho com a utilização de GPUs, quando as mesmas são usadas em aplicações de processamento de *stream*?

Além disso, qual é a interface de programação para GPUs que atinge o melhor desempenho?

1.6 HIPÓTESES

- Assim como a literatura vem reportando para diversas aplicações de outros domínios (seção 3), as aplicações de processamento de *stream* paralelizadas neste trabalho também conseguem aumentar o desempenho em até 10 vezes.
- A biblioteca CUDA é mais eficiente que OpenCL para as aplicações de processamento de *stream* paralelizadas.

1.7 METODOLOGIA

Segundo Lovato (2013), a metodologia tem por finalidade a identificação dos caminhos que serão executados, para que sejam alcançados os objetivos do trabalho, onde a partir do problema, o pesquisador inicia o processo da prática de

pesquisa científica.

1.7.1 Abordagem

Segundo Lovato (2013), a abordagem pode ser vista de duas dimensões, sendo a primeira relacionada ao raciocínio que é utilizado para inferir uma conclusão e a segunda relaciona-se com a utilização de números e estatística. O método de abordagem dedutivo inicia em planos maiores, para planos menores, já a abordagem quantitativa, é utilizada quando os resultados da pesquisa podem ser traduzidos em números.

Com isso, o trabalho utilizou a abordagem dedutiva, onde o estudo e desenvolvimento serão obtidos com a realização de pesquisas e aplicações existentes sobre computação paralela e aplicações de processamento de *stream*,

A abordagem quantitativa foi utilizada na análise das aplicações, onde foi medido o desempenho das aplicações quando executadas de forma homogênea e heterogênea, comparando os resultados a partir do tempo de execução, latência e *throughput*.

1.7.2 Procedimentos

Segundo Lovato (2013), a pesquisa bibliográfica envolve a consulta de obras desenvolvidas por outros autores, sobre o assunto envolvido.

A pesquisa bibliográfica foi utilizada para a revisão da literatura, bem como as aplicações já desenvolvidas com as tecnologias envolvidas. Também foi utilizada para o estudo das bibliotecas envolvidas.

A análise de dados e estatística foi utilizada para avaliar os resultados de desempenho das aplicações de *stream*, bem como para analisar as diferenças entre as bibliotecas OpenCL e CUDA.

1.7.3 Técnicas

Segundo Lovato (2013), a técnica de testes é utilizada como um instrumento para obter dados que permitem medir rendimento, frequência e capacidade de forma quantitativa.

No trabalho foi utilizada a técnica de testes, para a realização dos testes das bibliotecas nos diferentes tipos de aplicações de *stream*.

Para a coleta de dados, foram programados *scripts* que realizam o teste e coleta de estatísticas, como tempo de execução e transferência de memória. As análises foram feitas através de gráficos programados com a biblioteca *matplotlib* do *Python*.

Para a hipótese de que "assim como os trabalhos relacionados vem reportando, as aplicações de processamento de *stream* paralelizadas neste trabalho também conseguem aumentar o desempenho em até 10 vezes", foram implementadas aplicações paralelas de processamento de *stream*, e analisado o *speedup* quando comparado às aplicações *multi-thread*.

Já para a hipótese de que "a biblioteca CUDA é mais eficiente que OpenCL para as aplicações de processamento de *stream*", foram implementadas as aplicações utilizando as tecnologias OpenCL e CUDA, onde foi analisado o desempenho obtido em cada uma delas. A hipótese é validada se a biblioteca CUDA obter melhor desempenho entre estas três bibliotecas.

1.8 ORÇAMENTO

O orçamento dos valores gastos para o desenvolvimento deste trabalho, é apresentado no Quadro 1, onde na primeira coluna é apresentada a atividade, e nas seguintes o valor unitário, quantidade e valor total de cada atividade.

Quadro 1: Orçamento

Atividade	**Quantidade**	**Valor Unitário**	**Valor Total**
Impressões	1000	R$ 0,30	R$ 300,00
Capa	2	R$ 25,00	R$ 50,00
Espirais	3	R$ 5,00	R$ 15,00
Publicação do artigo	3	R$ 100,00	R$ 300,00
Horas de trabalho	500	R$ 50,00	R$ 25.000,00
Total			R$ 25.665,00

1.9 CRONOGRAMA

No Quadro 2 é apresentado o cronograma do trabalho, onde as células com fundo cinza representam o período planejado, enquanto o X representa o realizado.

Quadro 2: Cronograma

Atividade	**2017**					**2018**				
	Ago	**Set**	**Out**	**Nov**	**Dez**	**Fev**	**Mar**	**Abr**	**Mai**	**Jun**
Desenvolver trabalho de pesquisa	X	X	X							
Entregar e apresentação do projeto de pesquisa				X						
Estudar as bibliotecas de programação paralela CUDA, OpenACC e OpenCL	X	X	X	X						
Desenvolver e documentar protótipos utilizando as bibliotecas citadas			X	X						
Mapear trabalhos relacionados			X	X		X				
Definir e desenvolver as aplicações de *streaming*					X	X	X		X	
Analisar o ganho de desempenho das aplicações e validação de hipóteses						X	X	X	X	X
Escrever trabalho final							X	X	X	X
Apresentar o trabalho final										X
Escrever artigo científico								X		

CAPÍTULO 2: BASE TEÓRICA

Este capítulo é dedicado à apresentação da base teórica utilizada para a realização deste estudo.

2.1 APLICAÇÕES DE PROCESSAMENTO DE *STREAM*

Com o avanço do poder computacional, cada vez mais existem aplicações presentes no dia a dia das pessoas. Com esse avanço, sempre mais informações são geradas, com vários formatos, várias fontes e de forma contínua. Neste contexto, as aplicações de processamento de *stream* apresentam sua importância, realizando o processamento de todos esses dados de forma contínua e ininterrupta.

Este tipo de aplicação é usado em várias áreas. Em sistemas sísmicos, os dados de sensores são lidos de forma contínua e em tempo real. O processamento destes dados deve ser feito a tempo de gerar alertas de eventos naturais. Desta forma, o uso do paralelismo faz-se necessário.

Na bolsa de valores, as aplicações de processamento de *stream* também estão presentes, comprando e vendendo ações de forma automatizada. Nestas aplicações, os dados de entrada são lidos a partir do fluxo de transações ocorrendo. Os robôs utilizam os dados e fazem análises preditivas para identificar uma compra ou venda vantajosa.

Na área da saúde, as aplicações de *stream* são utilizadas para monitora-

mento de pacientes de forma remota. Dados originados de sensores ligados ao paciente são analisados em tempo real para histórico e alertas relacionados a problemas de saúde.

Apesar destes sistemas estarem presentes em várias áreas, a implementação dos mesmos não é algo trivial. A partir destes exemplos, fica claro que aplicações de processamento de *stream* estão relacionadas com aplicações críticas e de tempo real. Turaga et al. (2010) detalha que devido ao fluxo de dados ser variável, este tipo de sistema deve ser desenvolvido de forma escalável, a fim de suportar altas cargas de dados. Griebler (2016) apresenta as aplicações de processamento de *stream* como sendo um fluxo contínuo de dados/tarefas/instruções, onde cada uma destas possui uma entrada, realizam um processamento e gera uma saída. De acordo com Turaga et al. (2010), existem quatro propriedades que caracterizam as aplicações de processamento de *stream*:

- Fontes de dados contínuas: a fonte de dados que alimenta a *stream* é contínua, independente e geralmente não possui um fim determinado. A *stream* é uma sequência de dados que possui uma noção de tempo ou ordem. Alguns exemplos são *timestamps* associado ao dado origem, *time-to-live*, sequência de números e ordem de chegada.
- Análise contínua: o processamento de *stream* está muitas vezes relacionados a processos de longa duração e em tempo real. Diferente de sistemas tradicionais que executam uma ação e terminam, a *stream* deve ser continuamente analisada, para garantir que o processo seja capaz de gerar os resultados em um tempo aceitável. Além disso, devido a natureza dinâmica da aplicação, os processos devem ser capazes de trabalhar com quantidade de recursos computacionais variáveis.
- Restrições de desempenho: processamento de *stream* muitas vezes está relacionado com aplicações críticas com restrições de latência e *throughput*, independente da quantidade de entrada de dados.
- Gerenciamento de estado e resiliência a falhas: por estar muitas vezes relaci-

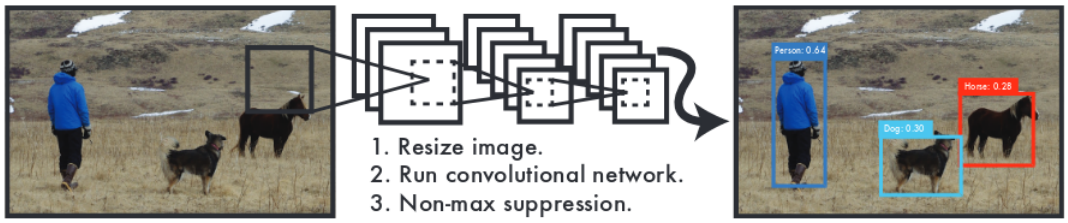
onado a sistemas distribuídos e heterogêneos, neste contexto diversos tipos de falhas podem ocorrer durante a execução da *stream*, como por exemplo: corrompimento de dados, falhas de rede e falhas da fonte de dados. As aplicações de processamento de *stream* devem estar preparadas para lidar com estes problemas.

Conforme Griebler (2016), de maneira geral as aplicações de processamento de *stream* trabalham na mesma forma que apresentado no Código 1. Neste exemplo, em cada interação do laço *while*, um elemento é lido da *stream* de entrada (linha 4), processados (linha 6), e o resultado é escrito em uma *stream* de saída (linha 7).

Código 1: Exemplo de *stream*.

```
1 void proc_seq(){
2   std::string stream_element;
3   while (1) {
4     read_in(stream_element); //reads each element of a given stream
5     if(stream_in.eof()) break; //test if stream is empty
6     compute(stream_element); //apply some computation to the stream
7     write_out(stream_element); //write the results into a given output stream
8   }
9 }
```

Figura 3: Processamento de imagem no YOLO



Fonte: (Redmon et al., 2015, p. 1)

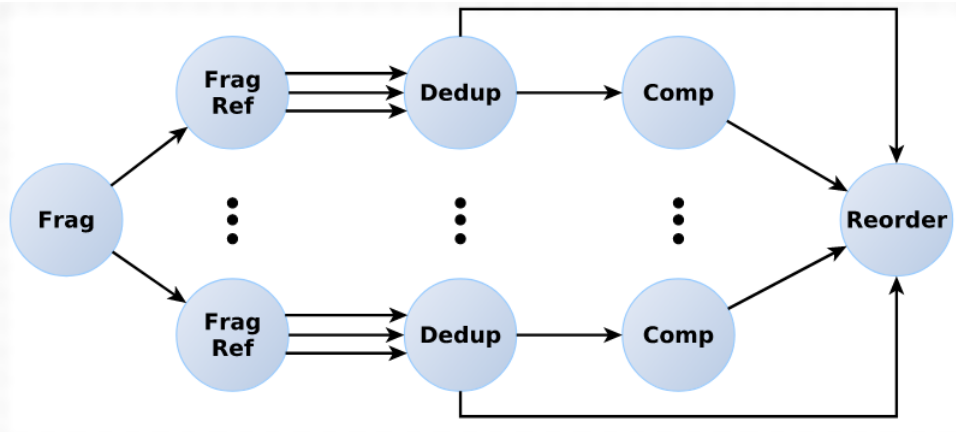
Existem diversos exemplos de aplicações de processamento de *Stream* que seguem as propriedades apresentadas. Na área de *deep learning*, um destes exemplos é o YOLO. De acordo com Redmon et al. (2015), o *You Only Look Once* - YOLO é uma aplicação que faz análise de imagens em tempo real para detecção

de objetos. Na Figura 3 são apresentadas as etapas de detecção de objetos em uma imagem.

Como visto na Figura 3, cada imagem é processada em uma *pipeline* (Seção 2.4.1.1). As características nesta aplicação são vistas tanto no processamento de uma imagem, onde diferentes pedaços da imagem são processados, quanto no processamento continuo em tempo real. Na detecção em tempo real, cada frame de um vídeo deve passar pela *pipeline* detecção de imagem no YOLO de forma contínua.

Outra aplicação muito comum que faz o uso de *stream* é a compactação de arquivos. O *dedup* é uma aplicação de processamento de *stream* que realiza a compactação de dados. De acordo com Bienia et al. (2008), o *dedup* comprime dados com a combinação de compactação a nível global e local, executando o processo chamado de deduplicação. Conforme Griebler et al. (2018), o *dedup* é implementado através de uma *pipeline* de cinco estágios apresentados na Figura 4, que envolvem fragmentação, refinamento, deduplicação, compressão e reordenação.

Figura 4: ***Estágios do pipeline do dedup***



Fonte: (Griebler et al., 2018, p. 1)

No *dedup*, o fluxo de dados acontece através da leitura e fragmentação de um ou mais arquivos, executando a compressão em diversos pedaços do dado.

Além destas aplicações, ainda existem diversas áreas que fazem uso de aplicações de processamento de *stream*, conforme apresentado na Figura 1. Esta seção apenas descreveu alguns exemplos deste tipo de aplicação para destacar os desafios e a necessidade da implementação do paralelismo. As aplicações paralelizadas, serão melhor detalhadas e apresentadas no Capítulo 3.

2.2 MEDIDAS DE AVALIAÇÃO

2.2.1 Desempenho

Conforme McCool, Robison e Reinders (2012), o desempenho nas aplicações é obtido através de um dos seguintes itens:

- Redução do tempo total que uma computação leva para o processamento de um item (latência).
- Aumentar a quantidade de itens que podem ser computados por segundo (*throughput*).
- Reduzir o consumo de energia de uma computação

Desta forma, todo esse aumento de desempenho pode ser obtido através do paralelismo. Para a análise de paralelismo de aplicação, o processo ideal é partir de uma versão funcional de uma aplicação. Com as medidas apresentadas nas próximas seções, pode ser analisado o progresso obtido pelos diferentes formatos de paralelismo.

2.2.1.1 Latência

De acordo com McCool, Robison e Reinders (2012), a latência é o total de tempo que um item leva para ser processado. A medida é um tempo, que pode variar de nanosegundos até dias. Quanto menor a latência, melhor.

Outra medida semelhante a latência é o tempo de resposta. Este normalmente está relacionado a aplicações transacionais, como servidores *web* ou banco de dados.

2.2.1.2 Throughput

O *Throughput* é a medida que representa a quantidade de itens processados em um determinado tempo. De acordo com McCool, Robison e Reinders (2012), quanto maior o *throughput* é melhor. A medida é feita por uma contagem em uma unidade de tempo (ex: itens por segundo).

McCool, Robison e Reinders (2012) afirma que para se obter um melhor *throughput* as vezes a latência pode ser aumentada. Por exemplo em uma *pipeline*, devido ao *overhead* presente nela a latência aumenta, por outro lado o *throughput* aumenta.

2.2.1.3 Speedup

O *Speedup* compara a latência de resolver um processo de forma sequencial com o tempo paralelizado em P unidades paralelas. Conforme McCool, Robison e Reinders (2012), a fórmula do *speedup* pode ser dada da seguinte forma:

$$speedup = S_P = \frac{T_1}{T_P}$$

Nesta fórmula, o T_1 representa o tempo de forma sequencial, enquanto o T_P representa o tempo da tarefa paralelizada em *P* unidades.

2.2.1.4 Eficiência

Conforme McCool, Robison e Reinders (2012), a eficiência mede o investimento em *hardware* necessário para um certo processo baseado no *speedup*. A eficiência ideal é quando o *speedup* de um processamento consegue crescer de

forma linear conforme o número de trabalhadores, atingindo assim a eficiência de 1 (ou 100%).

A fórmula utilizada para o cálculo da eficiência leva em consideração o *speedup* (S_P) e a quantidade de trabalhadores (P), conforme apresentado a seguir.

$$eficiencia = \frac{S_P}{P} = \frac{T_1}{PT_p}$$

McCool, Robison e Reinders (2012) apresenta que o *speedup* com baixa eficiência não é viável, já que a quantidade de investimento em *hardware* não compensa o *speedup*. Por isso o aumento na eficiência também deve ser um foco das aplicações paralelas.

2.2.2 Produtividade

Existem diversas formas de se medir a produtividade em desenvolvimento de *software*. Dentre estas, estão o COCOMO e experimento com pessoas. Nas próximas seções serão detalhadas estas medidas.

2.2.2.1 COCOMO

O *Construtive Cost Model* - COCOMO é um procedimento utilizado para estimativa de esforço desenvolvido por Barry W. Boehm. Ele foi desenvolvido a partir da análise estatística de projetos de *software*.

Conforme Selby (2007), o COCOMO foi desenvolvido em 1981. Este estabeleceu uma nova perspectiva a estimativa de esforço, já que na época os métodos mais comuns eram a quantidade de linhas de código (SLOC). Visto que o COCOMO refletia as praticas de desenvolvimento de *software* de 1970, o COCOMO II foi lançado em 1995, refletindo as novas praticas adotadas pela indústria.

De acordo com Barry Boehm Apurva Jain (auth.), o COCOMO II é um

modelo determinístico, que determina o esforço de desenvolvimento a partir de fatores determinísticos.

Conforme Selby (2007), o COCOMO II pode ser aplicado em três diferentes modelos: Para aplicações com interface de usuário é oferecido o modelo de composição de aplicação. Em outros projetos, é oferecido um modelo para projetos com arquitetura ainda não desenvolvida, e outro para projetos com arquitetura já presente.

De acordo com Adam Trendowicz (2014), o COCOMO II leva em consideração vários fatores, e pode ser representado nas seguintes funções:

$$Esforco = A * Tamanho^{E} * \pi_{i=1}^{n} EM_i$$

Onde:

- $Esforco$ representa o esforço total do projeto
- A representa a produtividade do desenvolvimento, iniciada em $A = 2.94$
- EM representa os *drivers* de esforço
- E representa a escala que é calculada conforme a seguinte fórmula:

$$E = B + 0.01 * \sum_{j=1}^{5} SF_j$$

onde:

  - B representa uma constante representada inicialmente por $B = 0.91$
  - SF representa o fator de escala

O $Tamanho$ representa o volume do *software*, e pode ser representado pelas quantidade de linhas de código (SLOC). Os *drivers* de esforço (EM) são clas-

sificados conforme as características e produtivade da equipe do desenvolvimento do *software*.

As constantes A e B são os valores iniciais estipulados pelo COCOMO II, porém os mesmos devem ser ajustados conforme o contexto do projeto.

2.2.2.2 Experimento com pessoas

De acordo com Griebler (2012), o experimento com pessoas é utilizado na computação quando existe o fator humano envolvido ao *software*. É um método científico que leva em conta hipóteses estatísticas para analisar o benefício de certa ferramenta ou *software*.

O experimento com pessoas leva em consideração a distribuição probabilística, para que sejam aplicados testes de hipóteses. A hipótese pode ser validada ou rejeitada. De acordo com Griebler (2012), a experimentação deve ser caracterizada em quatro fazes: definição, planejamento, execução e avaliação.

Na fase de definição e planejamento deve ser formulada a precisão e as hipóteses. Através de precisão, é obtida a quantidade de pessoas que devem estar presente no experimento.

As hipóteses serão o que conduzirão o experimento e são representados pelo H. O estudo deve levar em consideração a hipótese nula (H_0) e as hipóteses alternativas (H_n). Esta pode representar a comparação de esforço necessário para paralelização em duas bibliotecas. Um exemplo pode ser dado através da comparação de esforço para *SPar* e *pthreads*. A hipótese nula de que a implementação em *SPar* (seção 2.4.2.3) possui a mesma produtividade que em *pthreads* (seçao 2.4.2.4) poderia ser representada no seguinte formato: $H_0 = \mu_{spar} == \mu_{pthread}$. A hipótese alternativa 1 seria que a *SPar* possui uma produtividade maior que *pthreads*, representada pelo H_1: $H_1 = \mu_{spar} > \mu_{pthread}$. Neste sentido, a hipótese alternativa 2 seria de que *pthreads* possui uma produtividade maior que *SPar*: $H_2 = \mu_{spar} < \mu_{pthread}$.

Com as hipóteses formuladas, deve ser analisada as métricas que serão utilizadas para as hipóteses, alguns exemplos seriam tempo de desenvolvimento ou SLOC. Na execução do experimento devem ser coletadas as métricas utilizadas para que as análises de probabilidade sejam realizadas. Em seguida, na etapa de avaliação, cada hipótese deve ser aceita ou rejeitada, para que no exemplo apresentado, possa afirmar que uma biblioteca exige menos esforço que a outra.

2.2.2.3 Code Churn

O *code churn* é utilizado para analisar as diferenças entre versões de um *software* ou um módulo. Ele pode ser utilizado para analisar a produtividade como um todo ou a nível de desenvolvedor.

Conforme Bird, Menzies e Zimmermann (2015), o *code churn* é definido pelas linhas adicionas, linhas alteradas e linhas excluídas em um único arquivo. A obtenção do *code churn* se torna simples quando se é utilizada uma ferramenta de controle de versão (VCS), através da análise da diferença entre duas versões.

Bird, Menzies e Zimmermann (2015) apresenta o *code churn* como sendo relacionado com o ciclo de desenvolvimento de um produto. No desenvolvimento de um *software*, o *code churn* é maior do que quando o sistema está em produção.

Além da relação com o ciclo do *software*, conforme Bird, Menzies e Zimmermann (2015) o *code churn* também é relacionado a ocorrência de problemas em uma aplicação. Quando maior o *code churn*, mais chances de *bugs* ocorrerem. Desta forma, pode ser utilizada esta métrica para a realização de testes em *software*, já que os arquivos com maior *code churn* possuem tendência a possuir mais problemas.

2.3 ARQUITETURAS PARALELAS

De acordo com Duncan (1990), as arquiteturas paralelas permitem aos programadores o desenvolvimento de aplicações que façam o uso de paralelismo através de linguagens de alto nível. Esta estrutura pode ser programada de forma explicita para fazer o uso de paralelismo, a fim de resolver um problema com execução concorrente. De acordo com McCool, Reinders e Robison (2012), os computadores paralelos podem ser classificados conforme a classificação de Flynn, onde o dispositivo é definido pelo controle de fluxo e dos dados que adotam, conforme descrito a seguir:

- *Single Instruction, Single Data* - SISD: São os computadores sequenciais, onde cada instrução opera sobre um único dado.
- *Single Instruction, Multiple Data* - SIMD: operam aplicando a mesma instrução em múltiplos *streams* de dados. SIMD é ideal para a paralelização de *loops* simples que operam em um grande vetor de dados, como por exemplo, a soma de dois vetores.
- *Multiple Instruction, Multiple Data* - MIMD: suportam múltiplas instruções em diferentes *streams* de dados. Sistemas MIMD geralmente consistem de diferentes coleções de unidades de processamento independentes, cada uma tendo sua própria unidade de controle. Existem dois principais tipos de MIMD: Sistemas de memória compartilhada e sistemas de memória distribuída. Sistemas de memória compartilhada normalmente são aplicados em um ou mais processadores *multicore*, enquanto os sistemas de memória distribuída são aplicados em *clusters* de computadores.
- *Multiple Instruction, Single Data* - MISD: suportam múltiplas instruções sobre um único dado.

Os computadores com arquitetura paralela também devem oferecer algum tipo de memória que todos os nodos tenham acesso. A memória pode ser classificada em memória compartilhada e memória distribuída. A memória compartilhada

é acessada através do barramento presente no computador, e é utilizada para sincronia de processos em um único computador. Conforme Duncan (1990), a memória distribuída é utilizada para a sincronia de tarefas entre nodos em *clusters* de computadores.

2.3.1 CPU

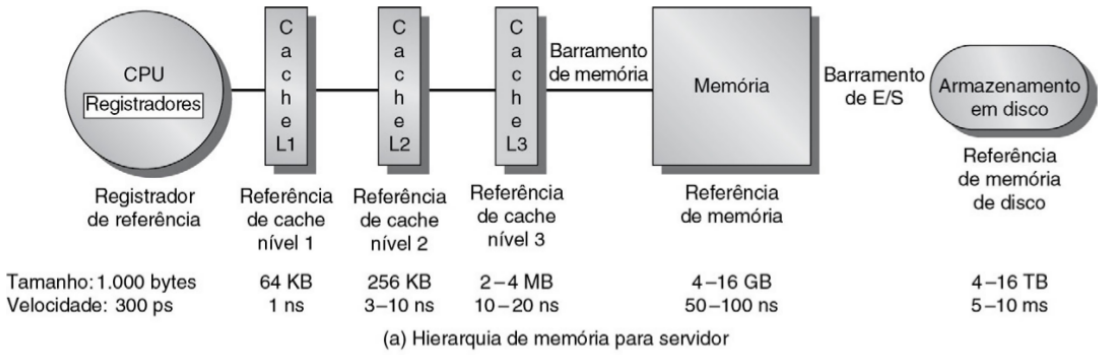
A *Central Processing Unit* - CPU é um dos principais componentes do computador. Ele é responsável pela execução das instruções dos programas.

Apesar da CPU ser responsável por executar as instruções, os dados processados precisam da memória para leituras, armazenamento e escritas. De acordo com Hennessy e Patterson (2014), um princípio presente na arquitetura dos computadores é a localidade temporal e localidade espacial. Nestas propriedades, os programas tendem a acessar os dados de forma contínua e de forma recorrente.

Visto que memórias de acesso mais rápido são mais caras, o processador faz o uso de *caches*. As *caches* são feitas utilizando multi níveis, com o princípio de que quanto mais perto do núcleo do processador, menor e mais rápida a memória. A Figura 5 representa esta hierarquia de memória, onde é apresentada a CPU de um servidor. Este computador possui o *cache* L1, L2 e L3, e em seguida a memória RAM e o disco.

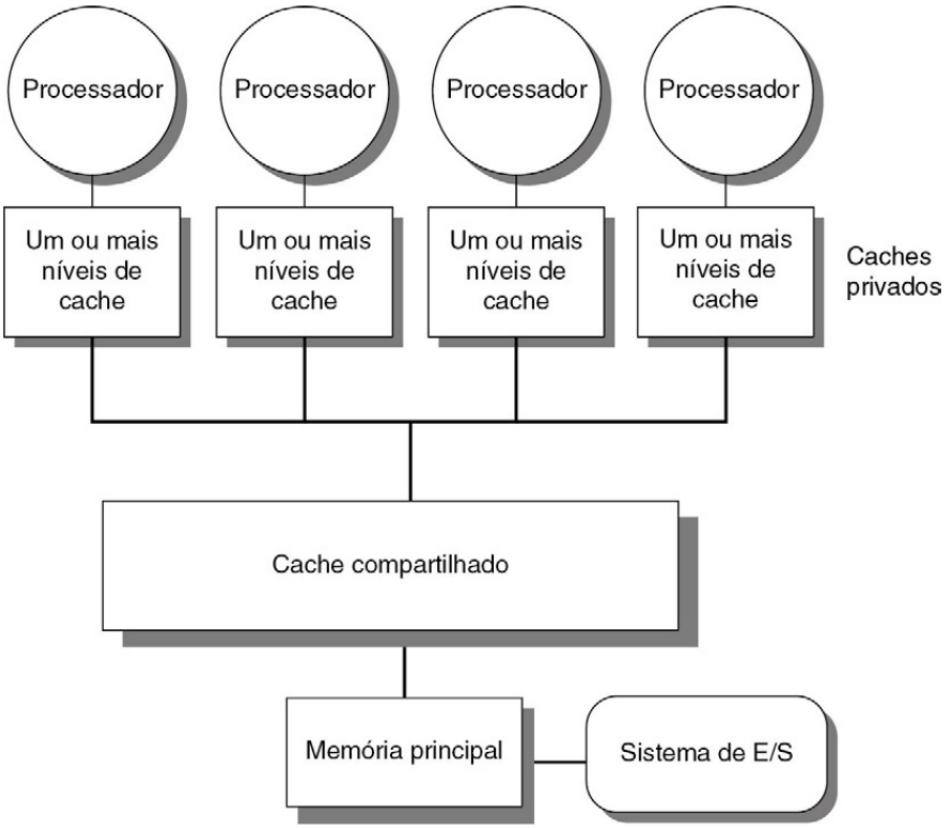
De acordo com Hennessy e Patterson (2014), processadores *multicore* possuem pelo menos um *cache* (L1) para cada núcleo, enquanto o *cache* L2 ou L3 estão disponíveis para todos os núcleos do processador. O uso dos *caches* do processador são controlados pela própria CPU, ficando evidente que os programas devem fazer o uso do princípio de localidade em sua implementação para uma execução otimizada. Na Figura 6 é apresentado a hierarquia dos caches de um CPU *multicore*. Como apresentado, os *caches* L1 são de uso exclusivo para cada núcleo, enquanto os *caches* de mais alto nível são compartilhados.

Figura 5: Hierarquia de memória da CPU



Fonte: (Hennessy e Patterson, 2014, p. 62)

Figura 6: *Caches* dos núcleos da CPU



Fonte: (Hennessy e Patterson, 2014, p. 305)

Além do formato *multicore* presente na Figura 6, a memória ainda pode ser distribuída entre os processadores, visando o aumento da largura de banda disponível. De acordo com Hennessy e Patterson (2014), esta arquitetura é chamada de NUMA (acesso não uniforme a memória), e ela apresenta a desvantagem de tornar a comunicação entre processadores mais complexa. A arquitetura NUMA é voltada especialmente para processadores com muitos núcleos. Ela faz com que o acesso a memória possua uma latência menor, impedindo que este seja um limitador de desempenho entre os núcleos.

2.3.2 GPU

As *Graphics Processing Units* - GPU apresentam uma arquitetura que se difere em muitas partes da CPU, oferecendo um maior número de núcleos com menor capacidade de processamento. De acordo com Cook (2013), a GPU apresenta três blocos chave: Memória (Global, constante e compartilhada), *Streaming Multiprocessors* (SMs) e *Streaming Processors* (SPs). A GPU é composta de um conjunto de SMs, onde cada SM contem N núcleos. Este é um dos principais conceitos que fazem a GPU ter eficiência. Quanto maior o número de SMs, mais tarefas ela pode processar simultaneamente. As SMs em GPU podem ser entendidas como um conjunto de núcleos, que compartilham o *cache* (*shared memory*) e outros componentes da SM, como o *warp scheduler*.

Graças a essa arquitetura que compartilha os componentes entre os núcleos, e possível que o crescimento de núcleos na GPU cresça. Enquanto que na CPU o incremento de um núcleo envolve a replicação de vários componentes como a *cache* e a unidade controladora, na GPU o aumento de núcleos é mais fácil de ser alcançado.

Os programas que são executados na GPU sao chamados de *kernels*. No modelo de execução dos *kernels*, as tarefas são enviadas para a GPU, onde são em executadas no conjunto de SMs.. Cada SM controla o fluxo de execução da tarefa em seus núcleos. De acordo com Cook (2013), o número de SMs em GPUs

vem sendo cada vez maior (Arquiteturas Kepler possuem 192 SMs).

Um SM possui diversos componentes, como é apresentado na Figura 7. Dentro das SMs, o *register file* (Azul escuro) é um componente que consiste de uma memória privada dos *threads* e que pode ser acessada sem tempo de espera, ou seja, no mesmo *clock* dos núcleos da GPU. Outro componente presente na SM é a memória compartilhada (*Shared Memory*) no inferior da Figura 7), podendo ser utilizada para sincronização e *cache* entre núcleos de um mesmo SM. Diferente da *cache* presente na CPU, a memória compartilhada é totalmente gerenciada pelo programador, cabendo a ele a implementação de otimizações de uso da mesma. O tamanho da memória compartilhada da GPU varia conforme sua arquitetura. Em GPUs NVIDIA, a arquitetura Fermi possui 16KB por *thread block*, enquanto que em arquiteturas mais novas como a Pascal, o tamanho é de 64KB.

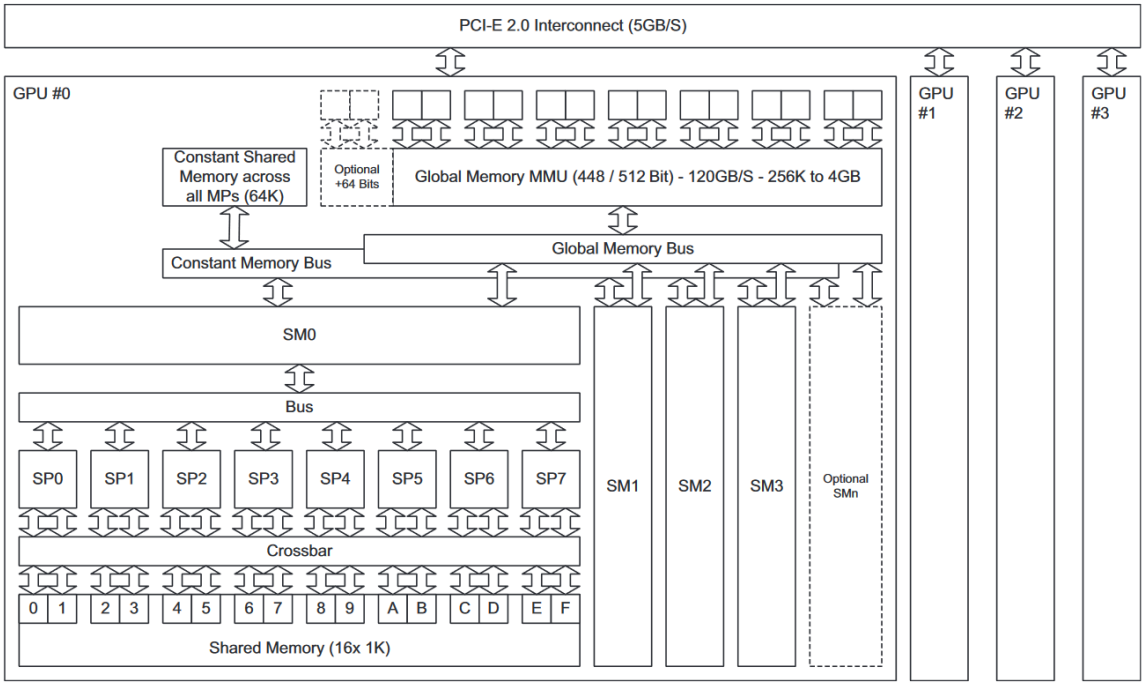
Figura 7: Arquitetura do *Stream Multiprocessor* de uma GPU Pascal

Fonte: Haris (2018)

Outro componente presente nas GPUs são os *warp schedulers*. Este componente é responsável por enviar as instruções para os inúmeros *Stream Processors*(núcleos) do SM. Em GPUs NVIDIA com arquitetura Pascal, cada SM possui dois *warp scheduler*, capazes de executar 32 instruções simultaneamente. É o *warp scheduler* que manda as instruções para os núcleos do SM (cor verde) e todos os núcleos de um *warp scheduler* irão estar executando a mesma tarefa. É devido a esta arquitetura de grupos de trabalho que a GPU consegue obter alto desempenho em programas paralelos.

Cook (2013) apresenta que cada SM possui barramentos para memória de textura, constante e global. A Figura 8 apresenta a arquitetura Kepler de uma GPU NVIDIA GT200, constando os barramentos, as memórias e o conjunto de SMs.

Figura 8: Diagrama da arquitetura da GPU (G80/GT200)



Fonte: (Cook, 2013, p. 43)

Com isso, Cook (2013) detalha que como em CPUs, um dos fatores que pode limita o desempenho destes processadores é a limitação que o programador impõe sobre a quantidade de *threads* que o programa suporta. Para que a GPU obtenha a maior eficiência possível de seu *hardware*, a tarefa sendo executada deve

conseguir fazer o uso de todas as *threads* disponíveis ou realizar o uso eficiente de banco de registradores e memória compartilhada.

2.3.3 Sistemas de Memória Heterogênea

Segundo Sanders e Kandrot (2010), os sistemas heterogêneos são utilizados em computadores de alto desempenho e fazem o uso conjunto de dois componentes: CPUs (*multicore*) e componentes massivamente paralelos de propósito específico (*manycore*). Alguns exemplos de componentes são as GPUs e FPGAs.

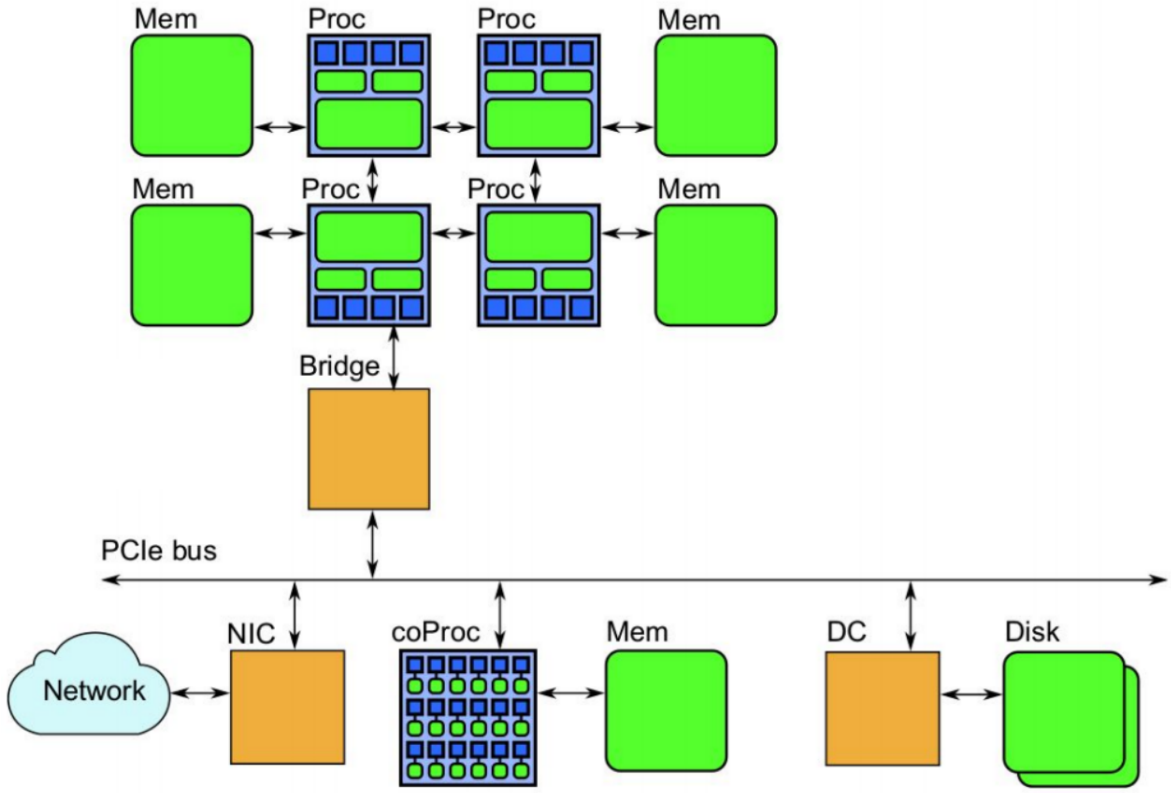
De acordo com Hwu (2015), sistemas de memória heterogênea permitem a integração entre CPUs e GPUs através do barramento PCI-Express presente no computador. Neste barramento é feita a comunicação entre processos da CPU e GPU.

De acordo com McCool, Reinders e Robison (2012) uma característica destes coprocessadores ligados ao barramento PCI-Express é que eles possuem *direct memory acces* (DMA). O DMA permite que o coprocessador leia e escreva dados em memória sem recorrer a CPU.

Como pode ser visto na Figura 9, os processos executados na CPU permanecem na memória RAM e fazem a comunicação com o dispositivo transferindo os dados para a memória da GPU. Quando a aplicação é organizada em *cluster* de computadores, o controlador de rede (NIC) fica responsável pela comunicação com os outros nodos da rede.

De acordo com Murarolli (2015), em alguns casos, a própria memória presente no computador não é suficiente para o processamento de um problema. Neste caso se faz o uso de memória distribuída. Um aglomerado de computadores realiza a computação em conjunto, e se comunica através de uma memória distribuída. Este conjunto é chamado de *cluster*, e também é um sistema de memória heterogênea

Figura 9: Sistemas de memória heterogênea



Fonte: (McCool, Reinders e Robison, 2012, p. 49)

Desta forma, a computação em sistemas de memória heterogênea permite a aceleração de aplicações e resolução de problemas complexos. Os computadores separam o processamento através da memória distribuída, realizando o processamento com CPU e co-processadores.

2.3.4 Cluster

Conforme McCool, Reinders e Robison (2012), *clusters* são sistemas de memória compartilhada interconectados com uma velocidade alta de comunitação via rede. Cada computador presente em um *cluster* é chamado de nodo. Os *clusters* são formados conectando sistemas que seriam independentes, sendo caracterizados como sistemas MIMD.

Conforme David A. Bader (2001), os *clusters* são muitas vezes utilizados

para aumentar o desempenho e a disponibilidade de um único computador. Sendo os nodos conectados via rede, as tarefas são processadas de forma paralela em diferentes máquinas através de memória compartilhada.

2.4 PROGRAMAÇÃO PARALELA

Conforme Pacheco (2011), muito do aumento da capacidade de processamento que existe atualmente, se dá devido ao aumento da densidade dos transistores em circuitos integrados, isto é, conforme o tamanho dos transistores diminui a velocidade aumenta. Por outro lado, circuítos muito pequenos diminuem a capacidade de dissipação de calor, chegando a um ponto que o circuíto pode se tornar não confiável. Por esse motivo, a indústria optou por ao invés de diminuir o tamanho dos circuitos integrados, aumentar o número de processadores em um único chip, criando assim os processadores *multicore*.

Desta forma, Pacheco (2011) explica que para que as aplicações consigam continuar a aumentar sua capacidade de processamento, elas devem ser explicitamente programadas para o uso do *multicore*, através da programação paralela.

A programação paralela pode ser implementada nos dois principais componentes de um algoritmo, as tarefas e os dados. Na paralelização de tarefas, múltiplas tarefas acontecem simultaneamente e de forma independente. Para McCool, Reinders e Robison (2012) o paralelismo de tarefas pode ser entendido como uma decomposição funcional e este não deve ser o principal foco na implementação de paralelismo, já que sua escalabilidade esta limitada ao número de tarefas que podem executar simultaneamente. Por exemplo, um programa que possui as tarefas A e B e C executadas em paralelo pode obter um *speedup* máximo de 3x.

Por outro lado, no paralelismo de dados o dado é distribuído entre diferentes núcleos que operam os dados em paralelo. McCool, Reinders e Robison (2012) define paralelismo de dados como qualquer paralelismo que cresce conforme os dados aumentam. Este tipo de paralelismo normalmente é implementado

separando matrizes de dados em pedaços que são processados separadamente.

Existem diversas maneiras de realizar a implementação de paralelismo em algoritmos, porém estes normalmente apresentam características em sua estrutura base, desta forma, na próxima seção serão apresentados alguns dos padrões paralelos mais utilizados e consagrados.

2.4.1 Padrões paralelos

Segundo McCool, Reinders e Robison (2012), atualmente todos computadores suportam paralelismo, desde pequenos *smartphones* até supercomputadores. Para a máxima eficiência deste *hardware* os programas devem explicitamente fazer o uso do paralelismo. Mesmo existindo abordagens automáticas de paralelização de programas sequenciais, estas falham pois muitas vezes a estrutura dos algoritmos deve ser alterada para a maior eficiência.

Para a simplificação da implementação deste tipo de aplicação os padrões paralelos servem como fundamentos organizacionais. Estes padrões são estruturas já utilizadas em aplicações e são vistos como melhores soluções para certos problemas.

Para McCool, Reinders e Robison (2012) as aplicações paralelas devem apresentar uma série de características, como descrito a seguir:

- Escalabilidade: capacidade de produzir programas que consigam fazer o uso eficiente de *hardwares* atuais e futuros. O uso do *hardware* não deve ser limitado pela implantação.
- Determinismo: Dadas as mesmas entradas, o resultado sempre deve ser o mesmo. O determinismo permite a criação de aplicações que sejam mais simples de depurar.
- Composição: é a capacidade de implementação de vários níveis de paralelismo. Para que seja possível a composição é necessária a capacidade de

aninhamento, ou seja, uma camada pode fazer o uso de paralelismo sem afetar a implementação de camadas acima ou abaixo.

- Portabilidade: Capacidade de ser transportado para outros sistemas operacionais e compiladores.

McCool, Reinders e Robison (2012) destaca que padrões de controle paralelo se relacionam a um ou mais padrões sequenciais. Nas próximas seções serão apresentados alguns padrões recorrentes.

2.4.1.1 Pipeline

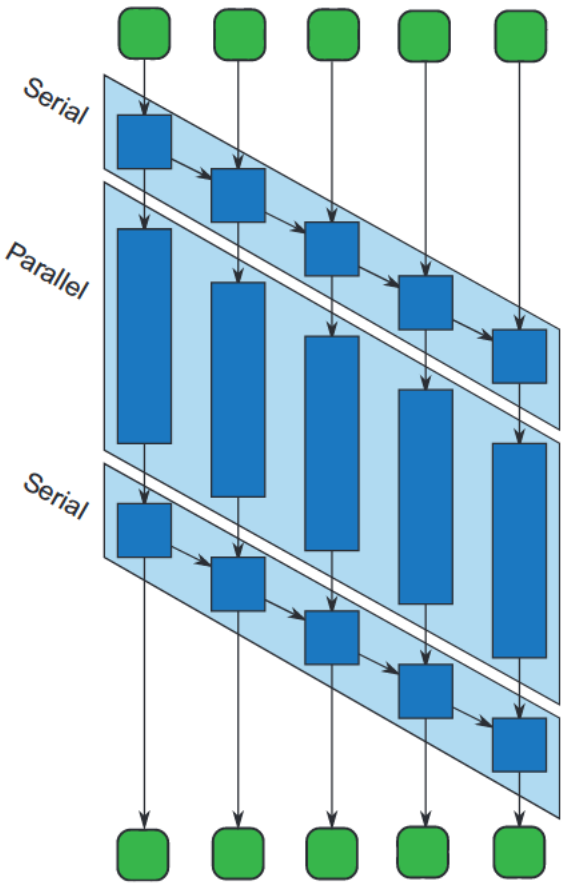
De acordo com McCool, Reinders e Robison (2012), algoritmos de *stream* (ou algoritmos online), precisam começar o processamento antes que todos os dados de entrada estejam disponíveis, e da mesma forma, a escrita do resultado é feita antes de todos os dados estarem disponíveis. Ou seja, computação e I/O são executados simultaneamente.

Uma das formas desta concorrência ocorrer é através de *pipeline*. O *pipeline* pode ser utilizado tanto para execução de I/O e computação ao mesmo tempo, quanto para a execução de códigos onde uma pequena parte deste deve ser feita de forma sequencial.

Na *pipeline*, os dados fluem sobre os estágios de forma ordenada. O paralelismo neste modelo é atingido quando um dados está sendo processado em um estágio, enquanto outro é processado ao mesmo tempo no próximo estágio. A Figura 10 apresenta o processamento de 5 elementos em uma *pipeline* que possui três estágios.

Como pode ser visto, ao mesmo tempo em que o segundo elemento é processado no primeiro estágio, o primeiro elemento já está sendo processado no segundo estágio. Os estágios do *pipeline* também podem ser replicados para um paralelismo mais escalável. Os estágios replicados conseguem processar mais

Figura 10: Execução de *pipeline* com 5 elementos



Fonte: (McCool, Reinders e Robison, 2012, p. 256)

registros ao mesmo tempo. Um exemplo de replicação pode ser visto no estágio 2 da Figura 10, devido ao estágio ser independente de I/O ele pode ser replicado.

2.4.1.2 Farm

De acordo com Garcia et al. (2016), o *Farm*, também chamado de *Task Farm* ou *Stream Map*, pode ser utilizado quando um problema é melhor dividida em um conjunto de tarefas que podem ser executados simultaneamente de forma independente.

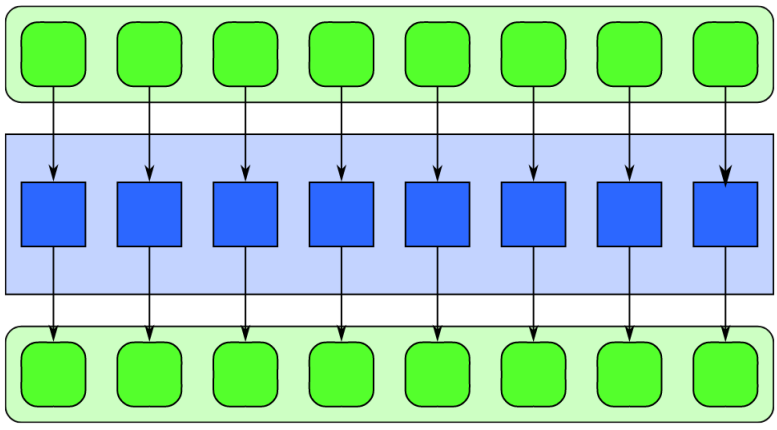
De forma geral, o padrão *Farm* aplica uma função de transformação em um dado no processamento de *Stream*. Um dado de entrada é dado para o *Farm*

que processa e gera uma saída. Ele pode ser visto como um *Map* (visto na seção 2.4.1.3) onde os dados de entrada não estão completamente presentes, podendo estes serem lidos de um fluxo contínuo de dados (*Stream*). O paralelismo neste padrão é atingido através da replica da função de transformação, para que mais elementos possam ser processados ao mesmo tempo.

2.4.1.3 Map

O padrão *Map* aplica uma função em todos elementos de um *array* paralelamente. Segundo McCool, Reinders e Robison (2012) a função que é executada é chamada de função elementar. A função elementar deve ser livre de efeitos colaterais. Desta forma, ela pode utilizar o máximo de concorrência e executar em ordem independente sem a necessidade de sincronizações. Na Figura 11 é apresentado o formato de execução paralelo do *map*.

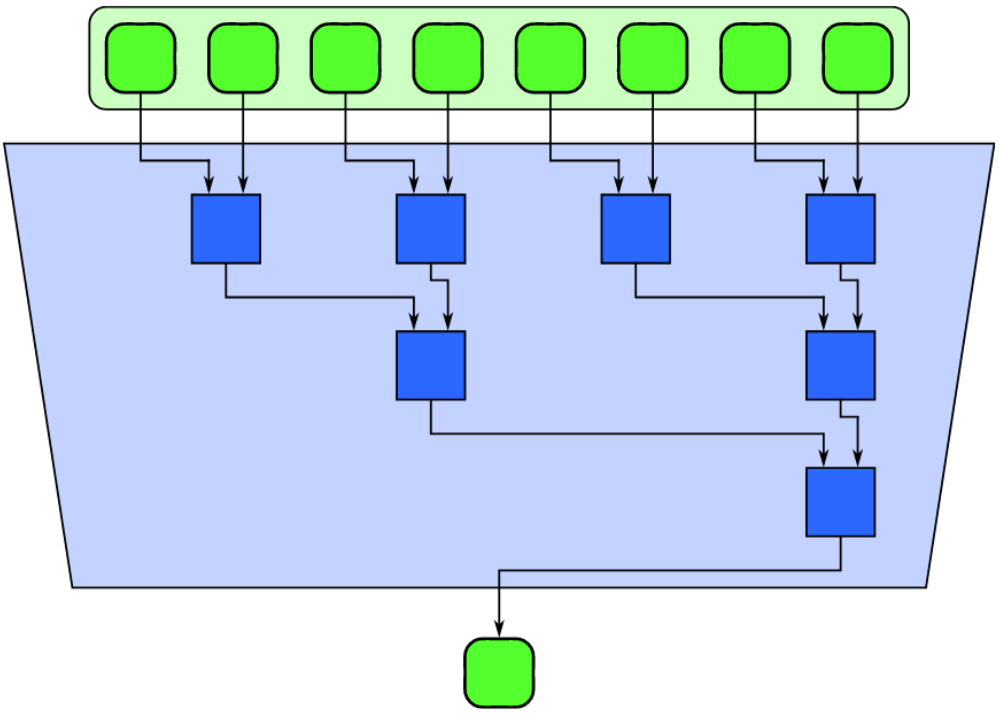
Figura 11: Padrão paralelo *Map*



Fonte: (McCool, Reinders e Robison, 2012, p. 89)

Como pode ser visto, para cada registro de um *array*, uma instância da função elementar é aplicada, gerando diversos resultados independentes. Este padrão pode ser implementado em fluxos de execução SIMD e SIMT, detalhados na seção 2.3.

Figura 12: Padrão paralelo *Reduce*



Fonte: (McCool, Reinders e Robison, 2012, p. 91)

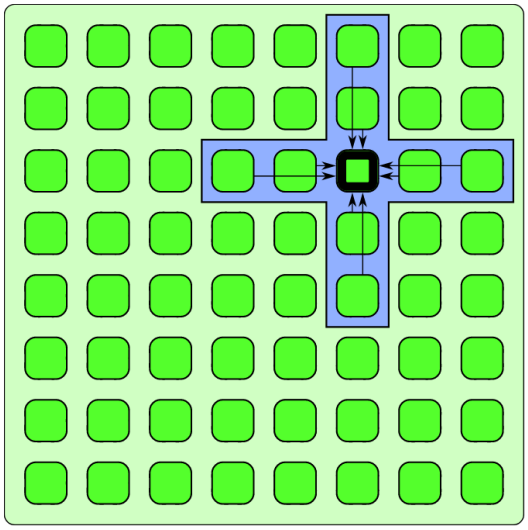
2.4.1.4 Reduce

O *reduce* é utilizado para estruturas mais complexas, onde o conjunto de trabalho não é totalmente independente. De acordo com McCool, Reinders e Robison (2012), o reduce permite que um *array* de dados seja sumarizado paralelamente.

Uma característica que o *reduce* deve possuir é a capacidade de os dados serem combinados em qualquer ordem. O *reduce* possui o *combiner*, que consiste de uma função que é executada em pares de dados, e deve retornar a combinação de dois valores.

A Figura 12 apresenta a caracterização do padrão. Como pode ser visto, a sumarização é alcançada através da execução paralela da combinação de pares de dados recursivamente.

Figura 13: Padrão paralelo *Stencil*



Fonte: (McCool, Reinders e Robison, 2012, p. 90)

2.4.1.5 Stencil

O *Stencil* é um tipo especial de *map*, onde o processamento de um elemento de uma matriz depende de dados vizinhos, seguindo um certo padrão de acesso a dados, conforme apresentado na Figura 13. De acordo com McCool, Reinders e Robison (2012), o *stencil* utiliza uma função elementar que faz o uso dos vizinhos de um elemento do *array*.

Neste padrão, otimizações podem ser feitas com o reuso de dados e localidade de processamentos. Os dados são acessados e mantidos em *cache* para que os próximos vizinhos o acessam de forma mais rápida. Por exemplo, em uma GPU os dados da memória global podem ser copiados para a memória compartilhada, de forma que outros *threads* o reutilizam.

2.4.2 Interfaces de programação paralela para CPUs

Nesta seção serão apresentadas algumas interfaces de programação que podem ser utilizada para o paralelismo de *stream*.

2.4.2.1 TBB

O *Thread Builing Blocks* - TBB é uma biblioteca em C++. De acordo com McCool, Reinders e Robison (2012) ele suporta paralelismo baseado em tarefas. O TBB oferece suporte a diversos padrões paralelos, como o *map*, *reduce* e *pipeline*, apresentados na seção 2.4.1.

O TBB é uma biblioteca que faz o uso dos *templates* do C++ para implementações genéricas. As implementações genéricas maximizam o reuso de código, já que elas podem ser utilizadas com diversos tipos de dados.

Conforme McCool, Reinders e Robison (2012), o paralelismo em TBB é baseado em tarefas, e não *threads*. A computação das tarefas é executada em um *thread pool* controlado pela biblioteca, que balanceia a execução entre diferentes tarefas. Esta característica permite o uso de composição de paralelismo, reduzindo o *overhead* e realizando o uso mais eficiente dos recursos.

O uso do TBB é possível em aplicações de *stream* através da função *parallel_pipeline* presente na biblioteca. O método *parallel_pipeline* recebe um conjunto de estágios, criados através do método *make_filter*. Cada estágio recebe como parâmetro o resultado do estágio anterior e retornado o dado que será repassado para o próximo estágio. No Código 2 é apresentado o uso do TBB para criar uma *pipeline* de compressão de arquivo com o BZIP2.

Código 2: Exemplo de *pipeline* em TBB, extraído de McCool, Reinders e Robison (2012).

```
1 int BZ2_compressFile(BZ2_compressFile)(FILE *stream, FILE *zStream
    , int blockSize100k, int verbosity, int workFactor) throw() {
2  ...
3  InputState in_state;
4  OutputState out_state(zStream);
5  tbb::parallel_pipeline(
6   ntoken,
7   tbb::make_filter<void,EState*>( tbb::filter::serial_in_order,
     [&]( tbb::flow_control& fc ) -> EState* {
8    if( feof(stream) || ferror(stream) ) {
9     fc.stop();
```

```
10      return NULL;
11     }
12     EState *s = BZ2_bzCompressInit(blockSize100k, verbosity,
       workFactor);
13     in_state.getOneBlock(stream,s);
14      return s;
15    } ) &
16    tbb::make_filter<EState*,EState*>( tbb::filter::parallel, [](
       EState*s ) -> EState* {
17     if( s->nblock )
18      CompressOneBlock(s);
19     return s;
20    } ) &
21    tbb::make_filter<EState*,void>( tbb::filter::serial_in_order,
       [&]( EState* s ) {
22     if( s->nblock )
23        out_state.putOneBlock(s);
24     FreeEState(s);
25    } )
26  );
27 }
```

No Código 2 existem três estágios, criados nas linhas 7, 16 e 21. Um ponto importante da implementação é que o primeiro e último estágio são especificados como sequenciais (através do parâmetro *tbb::filter::serial_in_order*), indicando a biblioteca que aquele estágio da *pipeline* só deve executar com uma instância.O segundo estágio é especificado como paralelo através do parâmetro *tbb::filter::parallel*, permitindo que a biblioteca gerencie quantas instancias daquele estágio são executadas paralelamente.

2.4.2.2 FastFlow

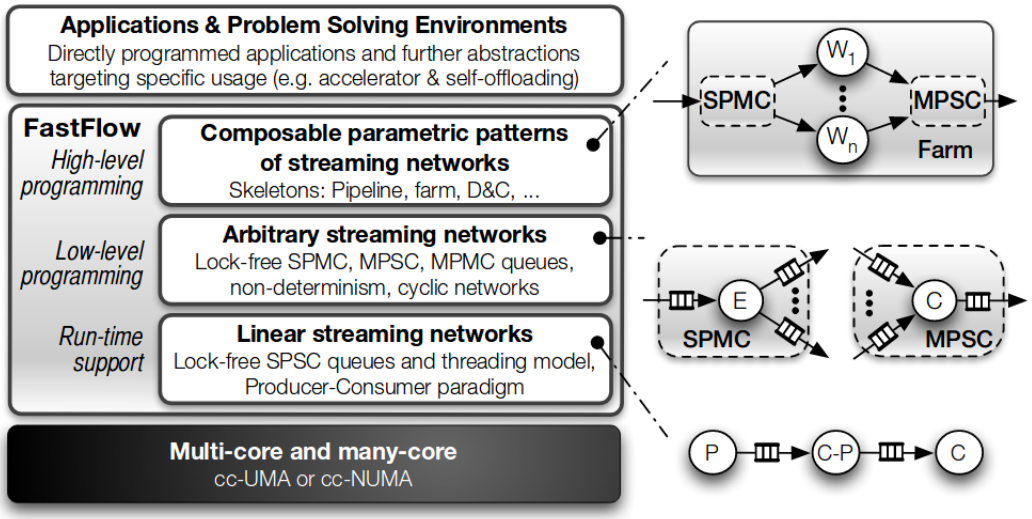
O *FastFlow* é um *framework* voltado para aplicações paralelas que fazem o processamento de *stream*. Ele busca oferecer eficiência em programas paralelos, enquanto oferece produtividade e abstração.

De acordo com Marco Aldinucci Marco Danelutto e Torquati (2014), o *FastFlow* foi construido baseado em quatro principios fundamentais: *design* em camadas, eficiência em mecanismos base, suporte a paralelismo de *stream* e progra-

mação baseada em padrões paralelos.

O *design* em camadas abstrai progressivamente conceitos como memória compartilhada e reuso de *cache*, porém ainda permite a customização destas camadas para criação de padrões mais complexos. Na Figura 14 é apresentada a arquitetura de camadas do *FastFlow*. A implementação base das filas do *FastFlow*, apresentado como a camada inferior, são livre de *lock* e esperas para filas um-produto-um-consumidor (SPSC). A camada acima estende a fila SPSC para o uso de filas um-para-muitos (SPMC), muitos-para-um (MPSC), e muitos-para-muitos (MPMC). Com as filas da segunda camada são realizadas implementações de padrões paralelos de alto nível como *pipeline* e *farm*. Estas implementações que estendem filas SPSC permitem comunicações em rede com pouco *overhead*, já que precisam de poucas barreiras de memória e oferecem um melhor reuso de *cache*. Um exemplo de implementação do padrão paralelo *farm* é mostrado no lado direito da Figura 14.

Figura 14: Arquitetura de camadas do *FastFlow*



Fonte: (Marco Aldinucci Marco Danelutto e Torquati, 2014, p. 4)

Assim como o TBB, o *FastFlow* também é uma biblioteca que faz o uso de *templates* do C++. Para se programar com esta biblioteca é necessária a criação de classes que representam cada estágio de um padrão paralelo. Um exemplo de implementação de *pipeline* é dado no Código 3. Neste exemplo, a *pipeline* cha-

mada de *pipe* cria os estágios conforme a variável *nStages*. Um estágio genérico é utilizado e implementado da linha 3 a 14. O estágio que estende a classe *ff_node* pode implementar os métodos *svc_init*, *svc_end*, que são chamados no início e fim do processamento da *pipeline* respectivamente. No método *svc* fica a implementação do estágio que será executada para cada tarefa, que neste caso, quando no primeiro estágio da *pipeline*, gera tarefas baseado em *nTasks*, enquanto que nos estágios seguintes é impresso o valor da tarefa.

Código 3: *Pipeline* em *FastFlow*.

```
1  #include <ff/pipeline.hpp>
2  using namespace ff;
3  class Stage:public ff_node{
4   int svc_init(){
5    printf("Stage %d\n",get_my_id());
6   }
7   void * svc(void * task){
8    if(ff_node::get_my_id() == 0)
9     for(long i=0; i < nTasks; ++i)
10     ff_send_out((void*)i);
11   else printf("Task=%d\n",(long)task);
12   return task;
13  }
14 };
15 int main(){
16  ff_pipeline pipe;
17  for(int i = 0; i < nStages; ++i)
18   pipe.add_stage(new Stage);
19  if(pipe.run_and_wait_end() < 0)
20   return -1;
21  return 0;
22 }
```

Apesar do Código 3 ser um exemplo simples, ele apresenta o funcionamento do *FastFlow*, desde como os padrões paralelos são instanciados no código até a criação da lógica a ser utilizada.

Além dos padrões paralelos, de acordo com Marco Aldinucci Marco Danelutto e Torquati (2014), o *FastFlow* oferece duas políticas de escalonamento das tarefas: *Dynamic Round-Robin* (DRR) e *on-demand* (OD). No DRR as tarefas são repassadas para os trabalhadores no formato *Round-Robin*, pulando trabalhadores com a fila de trabalho cheia. Na política OD os trabalhadores possuem uma fila de

trabalho pequena, que são alimentadas seguindo as regras do DRR. Além destas políticas, o *FastFlow* ainda permite a criação de políticas customizadas, através da implementação de uma classe que estenda a *ff_loadbalancer*.

2.4.2.3 SPar

O *SPar* é uma linguagem de domínio específico - DSL que foi projetada para simplificar o desenvolvimento de aplicações de *Stream*. De acordo com Griebler (2016), a linguagem consiste de um conjunto de atributos do C++ em código que representam o paralelismo de *stream* presente na estrutura.

De acordo com Griebler et al. (2017) o paralelismo de *stream* ainda é restrito a poucos *experts* da área, devido a este ainda ser baixo nível e muitas vezes complexo. A *SPar* busca simplificar o uso deste paralelismo, abstraindo conceitos de baixo nível como modelos de programação baixo nível, otimizações a nível de *hardware*, políticas de escalonamento de tarefas, *load balancing*, decomposição a nível de dados e tarefas e estratégia de paralelismos. As anotações em código são implementadas a partir do código sequencial, sem a necessidade de reescrita de código. Esta característica simplifica a manutenção e melhora a produtividade.

A portabilidade é outro ponto importante da *SPar*. Um programa desenvolvido com as anotações da *SPar* fica preparado para funcionar tanto em *multi-core* quanto em ambientes de *clusters*, sem a necessidade de alteração na implementação.

Os atributos (C++ atributes) estão presentes no C++ desde a versão C++11, e são tidos como padrões da linguagem. Devido as anotações da *SPar* serem feitas com este padrão, não há necessidade de aprender uma nova linguagem, mas sim apenas reutilizar os conhecimentos de C++.

De acordo com Griebler (2016), a *SPar* realiza transformações de código para que o código sequencial utilize bibliotecas já consagradas. O código é transformado para realizar a utilização de *FastFlow* e OpenMPI.

O *SPar* oferece flexibilidade na implementação de padrões paralelos utilizando as seguintes anotações: *ToStream*, *Stage*, *Input*, *Output* e *Replicate*. A anotação *ToStream* serve para indicar a DSL que aquela região anotada é uma região de *Stream*. O *ToStream* está normalmente anotado em laços como *for* e *while*.

Os estágios na *stream* representam operações que serão realizadas em cima dos dados, e são representados pelo atributo *Stage*. Os *Stages* devem ser utilizados dentro de um escopo do *ToStream*. De acordo com Griebler et al. (2017), o número de estágios que podem ser utilizados na *stream* é ilimitado.

Junto a anotação *Stage*, é possível adicionar a *Replicate*. Esta representa que o estágio pode ser paralelizado em múltiplas instâncias, e opcionalmente a quantidade de trabalhadores paralelizados. Griebler (2016) detalha que caso não seja informado o número de trabalhados, a *SPar* utiliza o valor da variável de ambiente SPAR_NUM_WORKERS.

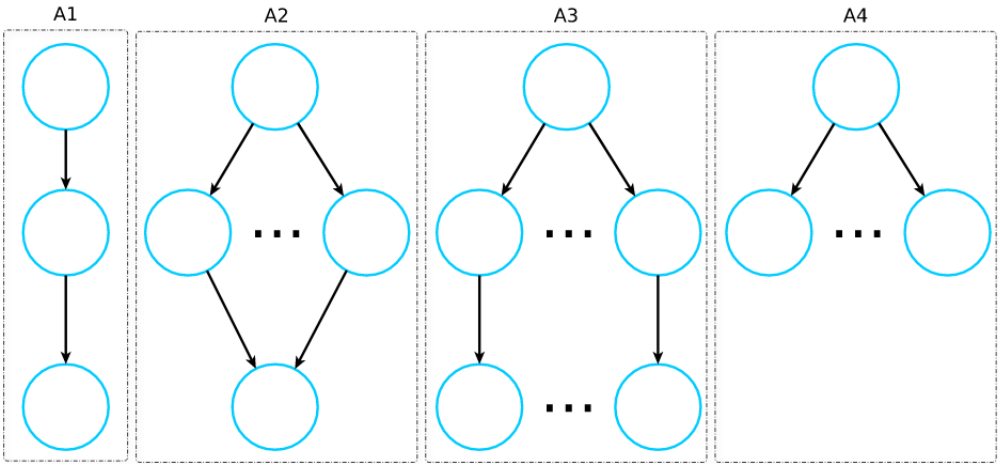
De acordo com Griebler (2016), a anotação *Input* deve ser utilizada para representar os dados que serão repassados para os estágios da *Stream*. A partir das variáveis passadas para o *Input*, serão criados os itens de dados que serão consumidos pelos estágios no *ToStream*. Por outro lado, o *Output* representa o que é produzido pelo estágio, e que será utilizado no próximo estágio da *stream*.

Além das anotações, o SPar oferece diferentes formatos de políticas de escalonamento através de opções do compilador. De acordo com Griebler et al. (2018), o SPar oferece três tipos de escalonamento:

- *spar_ondemand*: possuem a fila de processamento no tamanho de um, fazendo0 com que os estágios sejam processados de acordo com a demanda.
- *spar_ordered*: mantém a ordem de publicação das tarefas no escalonador.
- *spar_blocking*: o escalonador é bloqueado quando a fila de comunicação estiver cheia.

Através destas anotações, a *SPar* oferece uma grande flexibilidade para implementações de padrões paralelos como o *Pipeline* e *Farm*. Diferentes grafos de atividades podem ser implementados utilizando as anotações *Stage* e *Replacate*. Na Figura 15 são apresentados alguns grafos que podem ser implementados.

Figura 15: Grafos de atividade *SPar*



Fonte: (Griebler, 2016, p. 75)

Para a implementação do grafo de atividade A1, basta a anotação dos estágios da computação, já que nenhum estágio é paralelizado. Um exemplo de anotação no formato A1 é apresentado no Código 4. Neste exemplo, o laço *while* é denotado como *Stream*, e a leitura de dados é feita a partir de uma *string* em C++ na linha 3. Nota-se que que na linha 4 existe uma condição de finalização da *stream*, ela é utilizada *streams* onde a execução não é infinita, porém ainda é possível fazer uma leitura contínua, sem condições de pausa. as anotações *Input* e *Output* são utilizadas para que o *stream_element* seja repassado entre as *streams*.

Código 4: *Stream* de dados seguindo o grafo A1, extraído de Griebler (2016).

```
1 [[spar::ToStream]] while(1){
2   std::string stream_element;
3   read_in(stream_element);
4   if(stream_in.eof()) break;
5   [[spar::Stage, spar::Input(stream_element), spar::Output(
      stream_element)]]
6   { compute(stream_element);}
7   [[spar::stage,spar::Input(stream_element)]]
8   { write_out(stream_element); }
9 }
```

Por outro lado, para a implementação do grafo A2 da Figura 15, não são necessárias muitas alterações. Basta a adição do *Replicate* na anotação *Stage* do segundo estágio, presente na linha 5 do Código 4, resultando no código presente na linha 3 do Código 5.

Código 5: *Stream* de dados seguindo o grafo A2, extraído de Griebler (2016).

```
1 [[spar::ToStream]] while(1){
2   ...
3   [[spar::Stage, spar::Input(stream_element), spar::Output(
      stream_element), spar::Replicate(2)]]
4   {compute(stream_element);}
5   ...
6 }
```

Como pode ser visto, a flexibilidade oferecida no teste de diferentes formatos de *stream* é muito alta. Para a implementação dos exemplos A3 e A4 da Figura 15 podem ser criadas apenas alterando a anotação *Stage*, conforme a realizada para o A2.

2.4.2.4 PThreads

Além das bibliotecas já mencionadas nas seções anteriores, a programação de aplicações de processamento de *stream* também é possível através dos PThreads.

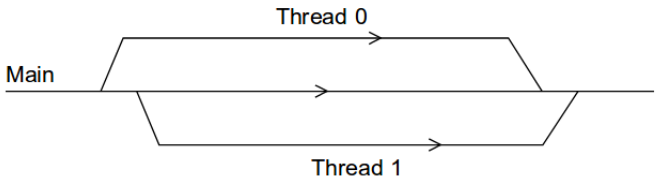
Segundo Pacheco (2011), o *thread* é mais leve que um processo, ao permitir o compartilhamento de memória dentro de um mesmo processo entre diversos *threads*. Dentro de um mesmo processo, o *thread* é um fluxo de controle que pode ser executado em paralelo.

O *pthread* é uma API para o uso de *thread* em sistemas POSIX. POSIX é uma padrão em sistemas *Unix-Like*, como Linux e Mac OS. De acordo com Pacheco (2011) o *pthread* é uma API para programação *multithread*.

Por padrão, todos processos iniciados pelo sistema operacional já pos-

suem um *thread*, chamado de *Main Thread*. Para se trabalhar com múltiplos *threads*, é utilizado o conceito de *Fork* e *Join*. o *Fork* inicia um novo *thread*, e indica a instrução inicial do mesmo. O *join* mantém o *thread* que o executou em espera até que o *thread* indicado termine sua execução. Na Figura 16 é apresentado um exemplo da execução do *fork* e *join*. O *Main Thread* cria dois *threads* (0 e 1) e espera a finalização através do *join*.

Figura 16: *Fork* e *Join* em *pthreads*



Fonte: (Pacheco, 2011, p. 158)

A criação de um *thread* em *pthread* é dada através da função *pthread_create*, e nela deve ser indicanda a instrução de início (função) e o local da memória que a referência do *thread* deve ser posta (ponteiro). De acordo com Pacheco (2011), a função *pthread_create* gera um *pthread_t*, que mantém os dados necessários para identificar o *thread* criado de forma única.

Para parar a execução do *thread*, a função *pthread_join* deve ser executada. Nela, deve ser indicando o respectivo *pthread_t* do *thread* criado pela aplicação.

Devido a memória que os *threads* utilizam em um mesmo processo ser compartilhada, o acesso a estas áreas deve ser controlado. De acordo com Pacheco (2011), as áreas que os *threads* podem alterar em paralelo são chamadas de seções críticas. Nesta áreas, deve ser utilizado um mecanismo que garanta que apenas um *thread* efetue a alteração por vez.

Uma opção de ser utilizada para regiões críticas são os *mutexes*. De acordo com Pacheco (2011), o *mutex* (ou exclusão mutua) pode ser utilizado para restringir o acesso a uma região crítica. O *mutex* oferece funções de bloqueio

e desbloqueio. Ao entrar na seção crítica, o *thread* deve executar a função *pthread_mutex_lock*, que irá aguardar até o *mutex* estar desbloqueado, para que em seguida o bloqueie. Ao final da seção crítica, a função *pthread_mutex_unlock* deve ser utilizada para desbloquear o *mutex*.

Para a sincronização entre *threads*, a API do *pthreads* oferece as condições. Condições permitem que os *threads* entrem em estado de espera até que um sinal seja dado para que a execução continue. Condições normalmente são utilizadas em conjunto com *mutexes*, para a criação de barreiras no código. Barreiras são regiões paralelas que só podem continuar a execução quando todos *threads* chegaram a um certo ponto do código. O *mutex* é utilizado em conjunto com a condição, já que a região que trabalha com a condição normalmente é uma seção crítica. Um exemplo é a implementação de barreiras. Barreiras são implementadas através de um contador, onde na região crítica o *thread* verifica se é o último *thread* a chegar naquele ponto do código. Caso não seja o último, o *thread* entra em espera pela condição. Caso seja o último, ao invés de esperar, o *thread* manda um sinal para a condição, e todos os *threads* aguardando a condição acontecer são liberados para continuar a execução.

A implementação de uma *pipeline* utilizando *pthreads* é possível através do uso de condições e de filas. Em um exemplo de pipeline de três estágios, apresentado no Código 6, é criada uma fila para cada estágio. Cada estágio faz a leitura de uma fila de dados e ao finalizar a tarefa enfileira a tarefa na fila do próximo estágio. As condições são utilizadas para que um *thread* com fila vazia entre em espera até que receba um sinal de que possui itens na fila. No Código apresentado esta lógica é abstraída pela classe *Channel* (linha 3), através das funções *send* e *receive*.

Código 6: *Pipeline* em *pthread*

```
1
2 template <class Task>
3 class Channel
4 {
5   public:
6     void send(Task t);
```

```
7       Task receive();
8   };
9
10  class Task
11  {
12    public:
13      int data;
14      bool end;
15  };
16
17
18  typedef struct __iochans
19  {
20      int stage_number;
21      Channel<Task> *in;
22      Channel<Task> *out;
23  } IO_Channels;
24
25
26  void *stage(IO_Channels *chans)
27  {
28      if(chans.stage_number == 0){
29          for (int i = 0; i < ntasks; i++){
30              Task t;
31              t.data = i;
32              (chans->out)->send(t); //publica tarefa
33          }
34          Task task_end;
35          task_end.end = true;
36          (chans->out)->send(task_end); //envia o fim da stream
37      }else{
38          Task t = (chans->in)->receive();
39          while(!t.end){
40              std::cout << "Task: "<< t.data << std::endl;
41              (chans->out)->send(t);
42              Task t = (chans->in)->receive();
43          }
44      }
45  }
46  int main(){
47      ... //Inicializa channels, threads e stages
48      stage1.in = NULL;
49      stage1.out = &ch12;
50      stage2.in = &ch12;
51      stage2.out = &ch23;
52      stage3.in = &ch23;
53      stage3.out = NULL;
54      pthread_create(&tid3, NULL, (void *(*)(void *))stage, (void *)
        &stage3)
55      pthread_create(&tid2, NULL, (void *(*)(void *))stage, (void *)
        &stage2)
```

```
56      pthread_create(&tid1, NULL, (void *(*)(void *))drain, (void *)
        &stage1)
57      pthread_join(tid3);
58      return 0;
59  }
```

A tarefa que irá fluir sobre a *pipeline* presente no Código 6, deve conter os dados a serem processados, e algo que defina que o *thread* deva terminar sua execução ao receber aquela tarefa. Neste exemplo, a variável *end* (linha 14) representa a condição de interrompimento da *stream*. O *IO_Channel* (linha 18) simplifica o estágio, ao passar para o estágio o *Channel* que ele deve usar para receber (*in*) e enviar (*out*).

Em relação ao estágios da *pipeline*, por questões de simplicidade, foi utilizado apenas um estágio repetido (linha 26), onde o primeiro estágio publica um conjunto de tarefas, e ao final envia uma tarefa de finalização de processamento. Os outros estágios usam o *Channel* para receber tarefas, realizar o processamento, e envia-lo para o próximo estágio.

Ao final, na função principal, os *threads* e *Channels* devem ser inicializados e invocados, para que ao final o *Main thread* espere a finalização do processamento da *stream*.

2.4.2.5 Visão geral das principais interfaces para CPU

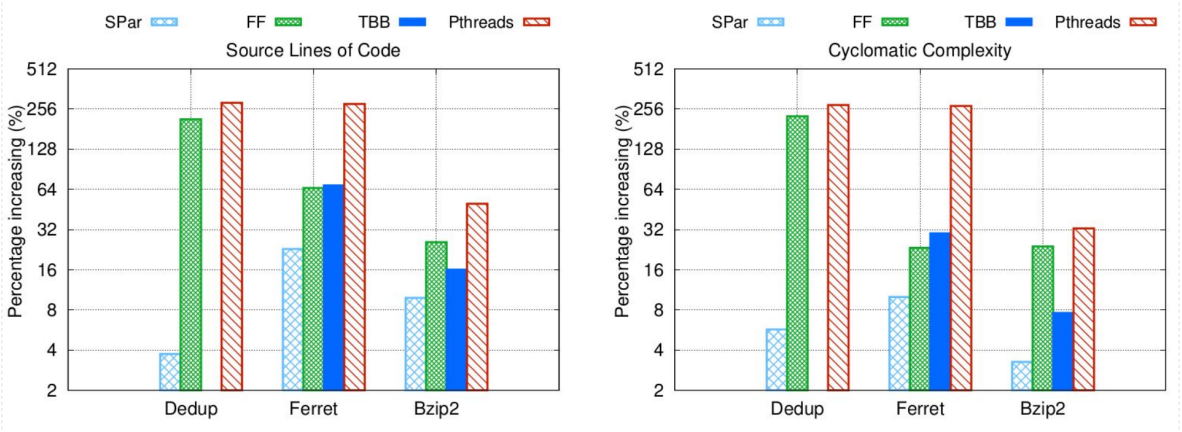
Existem diversas bibliotecas que permitem a programação paralela de processamento de *stream* em CPU, como o TBB, FastFlow, SPar e PThreads, apresentado nas seções anteriores.

O desempenho e produtividade são atributos muito importantes neste tipo de biblioteca. A capacidade de testar diferentes padrões paralelos de forma rápida permite a criação de soluções de alto desempenho. Ambos atributos já foram analisados entre as bibliotecas.

Griebler et al. (2018) realizou uma análise de produtividade e desempenho utilizando TBB, *FastFlow*, SPar e Pthreads nas aplicações Dedup, Ferret e Bzip2. Dedup e Bzip2 são aplicações de *stream* que realizam compactação da dados. Ferret é utilizado para busca de semelhanças entre imagens.

Na análise de produtividade, foi analisada a diferença entre o código sequencial com o código paralelizado com cada biblioteca. Foram medidas as linhas de código (SLOC) e a complexidade do código, através do *cyclomatic complexity*. O *cyclomatic complexity* mede a quantidade de caminhos que um programa pode conter. Na Figura 17 são apresentados a porcentagem de ambas-as medidas. Como pode ser visto, SPar precisou de menos de 10% de alterações no código em todas as aplicações. Esta produtividade é dada devido ao *SPar* necessitar apenas a anotação do código com os atributos do C++. Implementações com *pthreads* foram as que mais envolveram alterações de código. Isso se da devido ao controle de estágios das *pipelines* e paralelismo precisar ser todo controlado pela aplicação, sem auxílio de *bibliotecas*.

Figura 17: Produtividade de bibliotecas paralelas



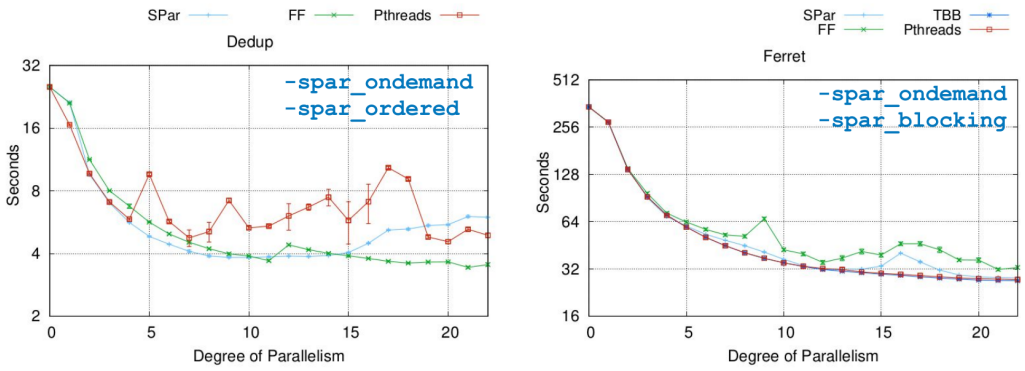
Fonte: Griebler et al. (2018)

Em relação ao desempenho das aplicações, Griebler et al. (2018) utilizou uma máquina com Intel (R) Xeon (R) CPU E5-2620 v3 2.40GHz, 24GB de memória RAM no sistema operacional Ubuntu Server. Foram realizados testes com as aplicações implementadas por especialistas em cada uma das bibliotecas. O teste

foi executado 10 vezes, com diferentes graus de paralelismo.

De acordo com Griebler et al. (2018) na execução do *Dedup*, a diferença entre as bibliotecas foi negligenciável, exceto pela implementação com *pthreads*, que obteve menor desempenho. A Figura 18 apresenta o desempenho do *Dedup* e *Ferret*, com um os diferentes graus paralelismo.

Figura 18: Desempenho de *Dedup* e *Ferret* com bibliotecas para CPU



Fonte: Griebler et al. (2018)

Na execução do *ferret*, conforme Griebler et al. (2018) o *SPar* obteve um desempenho inesperado, chegando a ser 10% mais lento no pior caso. Na Figura 19, é mostrado o desempenho das bibliotecas na compressão e descompressão do Bzip2. O desempenho entre as bibliotecas foi semelhante em ambos os casos.

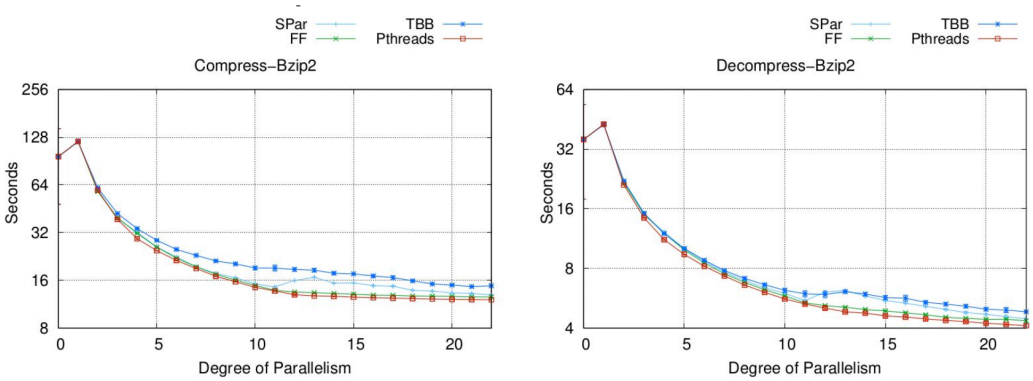


Figura 19: Desempenho de Bzip2 com bibliotecas para CPU

Fonte: Griebler et al. (2018)

Desta forma, dentre as bibliotecas, a SPar apresentou a melhor produtivi-

dade, incrementando o SLOC em 23% e o CNN em até 10% nos piores casos. Em relação ao desempenho, o SPar apresentou pouca degradação em relação as outras bibliotecas, mostrando que não existe um *overhead* ao utilizar uma biblioteca mais produtiva.

2.4.3 Interfaces de programação paralela para GPUs

Nesta seção serão apresentadas algumas interfaces de programação para GPU, exemplificando com o uso em multiplicação de matrizes e analisando os resultados.

2.4.3.1 CUDA

CUDA (*Compute Unified Device Architecture*) é uma biblioteca desenvolvida pela NVIDIA, que busca oferecer uma interface de programação para o uso de suas GPUs em aplicações de propósito geral, estando disponível nas linguagens C, C++ e Fortran através do compilador presente no CUDA *Toolkit*.

A forma de abstração dos *Streaming Multiprocessors* (apresentados na seção 2.3.2) da GPU adotada pelo CUDA, é a declaração de Blocos e *Threads*. Cada bloco e *thread* pode ter até 3 dimensões, sendo assim, as *grids* de *threads* de execução podem ser de tamanho X, XY, e XYZ. O máximo de *threads* suportados pelo CUDA em cada bloco é 1024, com isso, a multiplicação de XYZ deve ser menor ou igual a 1024. Um exemplo de escalonamento da *grid* pode ser representado através do paralelismo de uma tarefa que possua 2048 *threads*, em um vetor unidimensional, a tarefa deve ser no mínimo dividida em dois blocos de 1024 *threads* cada.

O código que é executado na GPU, é chamado de *kernel*, e deve ser declarado com a anotação __global__. Ao iniciar a execução do código, o CUDA se encarrega de invocar este método e realizar o escalonamento de acordo com os blocos e *threads* da invocação. Ficam disponíveis para o programador as variá-

veis blockIdx, blockDim e threadIdx, que representam respectivamente, o índice do bloco sendo executado, o tamanho dos blocos, e o índice do *thread* sendo executado. Com isso, a soma de dois vetores é apresentada no Código 7. Neste *kernel*, é invocado o kernel passando três vetores, o *a* e *b* representando os dados de entrada, enquanto o *c* armazena o resultado da operação.

Código 7: Kernel de soma em CUDA.

```
1 __global__
2 void sum(int *a, int *b, int *c){
3         int i = blockIdx.x*blockDim.x+threadIdx.x;
4         c[i] = a[i] + b[i];
5 }
```

Como apresentado na seção 2.3.2, as GPUs possuem uma memória que é separada da memória do CPU. Com isso, é necessário fazer o controle da transmissão entre os dados do Dispositivo e do Hospedeiro. O CUDA oferece suporte a esta transmissão através das funções *cudaMalloc*, *cudaMemcpy* e *cudaMemset*, sendo estas funções são semelhantes ao *malloc*, *memcpy* e *memset* da linguagem C.

Outro opção de gerenciamento de memória, oferecida pelo CUDA desde a versão 6, é o *Unified Memory*. Segundo CUDA (2018), o *Unified Memory* é uma memória gerenciada, que faz com que o GPU e CPU trabalhem como sendo um ponteiro unificado, e ao ser acessada, é automaticamente transmitida entre as GPUs e CPUs. Este formato simplifica a programação ao eliminar a necessidade da invocação do *cudaMemcpy* ou *cudaMemset* explicitamente e permite o desenvolvimento de um código mais simples de ser mantido.

A partir dos estudos realizados na biblioteca CUDA, foi desenvolvido um protótipo de multiplicação de matrizes. O escalonamento do problema foi realizado a nível de itens da matriz. O *kernel* se encarregou de realizar o cálculo de uma posição da matriz, conforme é apresentado no Código 8, nas linhas 3 e 4 são obtidos os índices da matrizes, para que nas linhas 6 a 10 seja computado o valor do item da matriz.

Código 8: Kernel de multiplicação em CUDA.

```
1  __global__
2  void multiply_item(int *a, int *b, int *c, int size) {
3      int i = blockIdx.y*blockDim.y+threadIdx.y;
4      int j = blockIdx.x*blockDim.x+threadIdx.x;
5      if (i < size && j < size) {
6          int temp = 0;
7          for (int k = 0; k < size; k++) {
8              temp+= (a[i*size+k] * b[k * size + j]);
9          }
10         c[i*size + j] = temp;
11     }
12 }
```

Outro recurso presente no CUDA são as operações atômicas, que segundo CUDA (2018), são funções que executam a leitura, alteração e escrita de dados presentes na memória global ou compartilhada da GPU.

Com isso, o mesmo problema de multiplicação de matrizes, ainda pode ser implementado utilizando três dimensões, fazendo o uso da operação *atomicAdd*, conforme é apresentado no Código do kernel 9. O método *atomicAdd* executa a operação de soma de forma atômica na linha 8, garantindo que o acesso concorrente dos dados em memória não resulte em nenhum problema.

Código 9: Kernel de multiplicação atomica em CUDA.

```
1  __global__
2  void multiply_item(int *a, int *b, int *c, int size){
3      int i = blockIdx.y*blockDim.y+threadIdx.y;
4      int j = blockIdx.x*blockDim.x+threadIdx.x;
5      int k = blockIdx.z*blockDim.z+threadIdx.z;
6
7      if(i < size && j < size && k < size){
8          atomicAdd(&c[i*size + j], a[i*size+k] * b[k * size + j]);
9      }
10 }
```

Um ponto importante que deve ser notado, é que como cada bloco pode conter no máximo 1024 *threads*, a execução deve ser executada em múltiplos blocos. O cálculo da de quantos blocos e *threads* devem ser executados, é apresentado no Código 10. Neste exemplo, possuindo os *threads* por bloco nos tamanhos

de 8x8x8, resultam em 512 *threads* por bloco. O número de blocos é obtido através da divisão do tamanho da matriz, com o número de *threads* que a dimensão suporta.

Código 10: Separação de tarefa em blocos.

```
1 #define BLOCK_SIZE 8
2 dim3 blocksPerGrid(1,1,1);
3 dim3 threadsPerBlock(BLOCK_SIZE,BLOCK_SIZE,BLOCK_SIZE);
4 blocksPerGrid.x = ceil_int(size, threadsPerBlock.x);
5 blocksPerGrid.y = ceil_int(size, threadsPerBlock.y);
6 blocksPerGrid.z = ceil_int(size, threadsPerBlock.z);
```

Para complementar o estudo, o código que executa o Kernel do Código 8 foi modificado para fazer a utilização do recurso *Unified Memory* do CUDA, e com isso foi simplificado o processo de transferência de dados entre CPU e GPU, onde o mesmo é apresentado no Código 11. A alocação de memória presente nas linhas 3 a 5 permitem que os dados sejam manipulados no *Main Thread*, bem como no *kernel*.

Código 11: Multiplicação de matrizes utilizando *Unified Memory*.

```
1 ...
2 int *matrix1, *matrix2, *matrix;
3 cudaMallocManaged((void**)&matrix1,sizeof(int) * size * size);
4 cudaMallocManaged((void**)&matrix2,sizeof(int) * size * size);
5 cudaMallocManaged((void**)&matrix, sizeof(int) * size * size);
6 ...
7 multiply_item<<<blocksPerGrid, threadsPerBlock>>>(matrix1,matrix2,
    matrix, size);
8 matrix = multiply(matrix1, matrix2, size);
9 cudaFree(matrix1);
10 cudaFree(matrix2);
11 cudaFree(matrix);
```

Além do uso do *Unified Memory*, outra forma de implementação da multiplicação de matrizes foi a utilização de uma matriz de duas dimensões, através do uso de *structs* do C++ e o recurso de *Unified Memory* do CUDA. O mesmo apresentou uma grande degradação de desempenho, devido a memória alocada não ser contínua, como nas matrizes tratadas com apenas uma dimensão.

Devido a localização da memória compartilhada do SM (Seção 2.3.2), ela

apresenta um acesso mais rápido que o acesso a memória global, desta forma, foi desenvolvida uma versão do algoritmo que faz o uso de *cache* de memória na memória compartilhada. O *cache*, apresentado no Código 12, é feito com o uso de matrizes 8x8 na memória do grupo de trabalho (bloco), para que desta forma seja otimizado o acesso à memória que são feitos de forma repetida. Cada *thread* do grupo de trabalho copia um elemento da matriz para a matriz em *cache* (Linhas 24 e 25), para que então seja feita a multiplicação de matrizes daquela submatriz (Linhas 27, 28 e 29). O resultado da operação é acumulado até que o *cache* percorra toda a matriz. Ao final de cada *thread*, o resultado é transferido para a matriz *c* (resultado).

Código 12: Multiplicação de matrizes otimizada em CUDA.

```
1
2  __global__
3  void multiply_item(int *a, int *b, int *c, int n) {
4      int i = blockIdx.x;
5      int j = blockIdx.y;
6
7      int global_column = blockIdx.y*blockDim.y+threadIdx.y;
8      int global_line = blockIdx.x*blockDim.x+threadIdx.x;
9
10     int idX = threadIdx.x;
11     int idY = threadIdx.y;
12
13     int qq = gridDim.x * blockDim.x;
14
15     int numSubMat = qq/BLOCK_SIZE;
16
17     int resp = 0;
18
19     __shared__ int A[BLOCK_SIZE][BLOCK_SIZE];
20     __shared__ int B[BLOCK_SIZE][BLOCK_SIZE];
21
22     for (int k=0; k<numSubMat; k++)
23     {
24         A[idX][idY] = BLOCK_SIZE * i + idX < n && BLOCK_SIZE * k
       + idY < n ? a[BLOCK_SIZE * i + idX + BLOCK_SIZE * k + idY
       ]: 0;
25         B[idX][idY] = BLOCK_SIZE*k + idX< n && (BLOCK_SIZE*j+idY)
       < n ? b[BLOCK_SIZE*k + idX + (BLOCK_SIZE*j+idY)] : 0;
26         __syncthreads();
27         for (int k2 = 0; k2 < BLOCK_SIZE; k2++)
28         {
29             resp += A[idX][k2] * B[k2][idY];
30         }
```

```
31            __syncthreads();
32
33        }
34        if(global_line < n && global_column < n){
35            c[global_line * n + global_column] = resp;
36        }
37    }
```

2.4.3.2 OpenACC

Outra biblioteca que permite a execução de código em GPU é o OpenACC. Ele oferecendo uma interface mais alto nível quando comparado com o CUDA ou OpenCL.

Segundo Farber (2016), OpenACC é um conjunto de diretivas que expressam o paralelismo e o movimento de dados no código sequencial. Com essas diretivas o compilador é capaz de criar e otimizar aplicações paralelas, que podem ser portadas para arquiteturas paralelas como GPU e CPU.

O OpenACC possui uma especificação padronizada, que pode ser utilizada nas linguagens de programação C, C++ e FORTRAN. Segundo The OpenACC™ Application Programming Interface (2013), o modelo de execução do OpenACC consiste da utilização de paralelismo tanto no hospedeiro quando no dispositivo, onde grande parte dos processos são dirigidos pelo hospedeiro (CPU), e partes que exigem uso intensivo de computação, são executados em *offloading* no dispositivo (GPU). Estas partes são conhecidas como *kernels*.

Para expressar o paralelismo, o OpenACC apresenta as diretivas *parallel* e *kernel*. Segundo Farber (2016), a diretiva *parallel* indica ao compilador que todo o escopo do laço executara apenas uma operação, enquanto o *kernel* oferece mais flexibilidade, permitindo a execução de múltiplas operações e a escolha de qual dispositivo deve executar as operações (ex: CPU, GPU).

A forma em que o paralelismo de ambas as diretivas será executada, pode ser parametrizadas em três níveis distintos, sendo estas especificadas pelas direti-

vas *gang*, *worker* e *vector*. De acordo com The OpenACC™ Application Programming Interface (2013), um *gang* é composto de um ou mais *workers*, e o *vector* é utilizado dentro dos *workers* para execução de operações *Single Instruction, Multiple Data* (SIMD). Na execução do código compilado com o OpenACC em um dispositivo, um ou mais *gangs* são executados, cada um composto de um ou mais *workers*.

OpenACC apresenta uma API mais alto nível quando comparado a CUDA e OpenCL. Desta forma, uma soma de matrizes se torna trivial com a utilização da anotação *parallel*, como apresentado no Código 13, onde o paralelismo é expressado com a anotação na linha 1, sem a necessidade de alteração do código sequencial.

Código 13: Soma de matrizes em OpenACC

```
1 #pragma acc parallel loop
2 for (int i = 0; i < size; i++) {
3     matrix3[i] = matrix1[i] + matrix2[i];
4 }
```

Quando o código em OpenACC poder ser utilizado em ambientes heterogêneos, deve se preferencialmente especificar explicitamente a transição de memória entre dispositivo e hospedeiro através da diretiva *data*. A diretiva *data* permite especificar a copia ou criação de uma variável entre dispositivo e hospedeiro. Com esta diretiva, o compilador se encarrega de realizar as operações de movimentação de memória entre hospedeiro e dispositivo de uma forma compatível em múltiplas plataformas.

Para a multiplicação de matrizes, não é necessária a alteração do código de multiplicação sequencial, apenas é necessário adicionar as anotações de *loop* e *data*, como é apresentado no Código 14.

Código 14: Multiplicação de matrizes em OpenACC

```
1 #pragma acc data copyin(matrix1[0:size * size]) copyin(matrix2[0:
    size* size]) create(matrix[0:size*size]) copyout(matrix[0:size*
    size])
2 #pragma acc kernels
```

```
3  #pragma acc loop independent vector(16)
4  for (int j = 0; j < size; j++) {
5      #pragma acc loop independent vector(16)
6      for (int i = 0; i < size; i++) {
7          matrix[i*size+j] = 0;
8          for (int k = 0; k < size; k++) {
9              matrix[i*size+j] += matrix1[i*size+k] *matrix2[k*size+
       j];
10         }
11     }
12 }
```

Na linha 1 do Código 14, é explicitamente especificado que devem ser copiadas as matrizes 1 e 2 da CPU para GPU, enquanto que a matriz resultado deve ser criada na GPU, e copiada para o hospedeiro no final da operação. As linhas 3 e 5 expressam o paralelismo dos laços, e fazem com que as instruções sejam executadas no dispositivo em vetores de tamanho 16.

O OpenACC também permite a especificação de reduções, que são utilizadas quando se busca transformar um vetor de dados em um único valor. Soma, subtração e multiplicação em listas podem ser considerados reduções. Desta forma, ainda nesta mesma aplicação, é possível fazer com que o compilador otimize ainda mais este código, transformando o laço da variável K (linha 8) em uma redução de soma, conforme é apresentado no Código 15, com a adição da diretiva *reduction* na linha 8, indicando a soma da variável *sum*.

Código 15: Laço da multiplicação de matrizes em OpenACC com redução

```
6  ...
7  int sum = 0;
8  #pragma acc loop reduction(+:sum)
9  for (int k = 0; k < size; k++) {
10     sum += matrix1[i*size+k] *matrix2[k*size+j];
11 }
12 matrix[i*size+j] = sum;
```

Com os exemplos apresentados, nota-se que apesar do OpenACC apresentar uma biblioteca de alto nível, ainda é necessário o conhecimento da arquitetura e funcionamento das GPUs para se obter um bom desempenho.

2.4.3.3 OpenCL

Além do OpenACC e CUDA, o OpenCL também é uma interface de programação, que se destaca por ser multiplataforma e abranger todo o contexto de computação heterogênea.

Open Computing Language (OpenCL) é uma biblioteca voltada para a computação heterogênea. Segundo Scarpino (2012). O OpenCL é baseado em um padrão aberto e multi-plataforma, que pode ser usado na execução de rotinas em Co-processadores, CPUs e GPUs de grandes fabricantes, como NVIDIA, AMD e Intel.

De acordo com Scarpino (2012), uma das grandes vantagens do OpenCL quando comparado à outras bibliotecas é sua portabilidade, que permite executar a mesma rotina em múltiplos dispositivos, desde que os dispositivos acoplados ao computador tenham suporte a OpenCL.

Apesar do OpenCL possuir um padrão, existem várias extensões, que servem tanto para capacidades específicas de algumas plataformas, quanto para funcionalidades presentes em dispositivos como *GPU*.

O OpenCL é uma linguagem baseada em C99, e para se programar em OpenCL devem ser criados *kernels*, que consistem de funções que podem ser executadas nos dispositivos. A compilação do código é feita em tempo de execução e a mesma é voltada a um dispositivo em específico, como GPU, CPU ou FPGA.

Para controlar a execução em um destes dispositivos, é necessária a criação de um contexto. Scarpino (2012) afirma que as aplicações hospedeira gerenciam os dispositivos que irão executar as funções através do *context*, onde a partir destes são criados *programs*, que são o resultado da compilação em tempo de execução do OpenCL.

A partir do momento em que o *context* e o *program* existem no OpenCL, é

possível fazer a execução do código através de uma ou mais *command queues*. As *command queues* são filas que são controladas no hospedeiro, e através dela que a CPU pode coordenar a execução do código, sincronização e as transferências de memória entre dispositivos. A *Command Queue* é semelhante à uma *stream* do CUDA, onde podem ser criadas inúmeras *Command Queues* para execução de *kernels* independentes.

Desta forma, o Código 16 apresenta o código em C++ necessário para rodar um programa com OpenCL que efetua o cálculo do produto das matrizes, extraido do livro Scarpino (2012).

Código 16: Inicialização do OpenCL

```
int main() {
    ...
    //Criação do contexto
    context = clCreateContext(NULL, 1, &device, NULL,
    NULL, &err);
    //Leitura e criação do programa
    ...
    //Compilação
    clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    //Criação do kernel
    kernel = clCreateKernel(program, KERNEL_FUNC, &err);
    //Command queue
    queue = clCreateCommandQueue(context, device, 0, &err);
    //Transferencia de dados
    mat_buff = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(float)*16, mat, &err);
    vec_buff = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(float)*4, vec, &err);
    res_buff = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
    sizeof(float)*4, NULL, &err);
    //Argumentos do kernel
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &mat_buff);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &vec_buff);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &res_buff);
    work_units_per_kernel = 4;
    //Invocação do kernel
    clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
    &work_units_per_kernel, NULL, 0, NULL, NULL);
    //Leitura do resultado
    clEnqueueReadBuffer(queue, res_buff, CL_TRUE, 0,
    sizeof(float)*4, result, 0, NULL, NULL);

    //Clean up
```

```
34      ...
35  }
```

No Código 16, a escolha do dispositivo GPU é abstraida.. O código OpenCL pode ser lido de um arquivo ou de um texto em memória. O texto lido é utilizado para realizar a compilação do OpenCL, presente na linha 9. O *command queue* é criado na linha 13, onde em seguida os dados são transferidos para a GPU, para que nas linhas 22 a 27 seja enfileirada a execução do *kernel*, e finalmente na linha 30 os dados resultantes são transferidos para o hospedeiro.

Dentro do código OpenCL, o processamento deve ser feito a partir do índice atribuído ao *thread* atual, semelhante ao formato do CUDA. O código que é executado no *thread* se assemelha ao código que estaria presente no escopo de um laço *for* sequencial, porém ao invés da utilização do laço para obtenção das variáveis *i*,*j* ou *k*, se faz o uso da função *get_global_id*.

No Código 17 é apresentado o código do *kernel*, que efetua o cálculo do produto de cada elemento da matriz. Pode ser visto que o índice do elemento sendo processado é obtido na linha 4, para que seja feito o cálculo em seguida.

Código 17: Kernel de Produto OpenCL

```
1 __kernel void matvec_mult(__global float4* matrix,
2  __global float4* vector,
3  __global float* result) {
4       int i = get_global_id(0);
5       result[i] = dot(matrix[i], vector[0]);
6 }
```

Desta forma, para o cálculo de multiplicação de matrizes, o código do *kernel* acaba não diferindo muito das outras bibliotecas, onde apenas é alterado o formato que obtenção das variáveis que representam a linha e coluna sendo calculada, conforme apresentado no Código 18.

Código 18: Kernel de Multiplicação de matrizes e OpenCL

```
1 kernel void mtrix_mult(int n, global int *a, global int *b, global
      int *c){
2     int j = get_global_id(0);
```

```
3       int i = get_global_id(1);
4       if (i < n && j < n) {
5           c[i * n + j] = 0;
6           for(int k = 0; k < n; k++){
7              c[i * n + j] += a[i * n + k] * b[k * n + j];
8           }
9       }
10  }
```

Existem ainda outras formas de se trabalhar com o índice do *thread* sendo executado. A carga de tarefas do OpenCL pode ser dividida em grupos de trabalho de até 1024 *threads*. Nestes casos, para a identificação do grupo de trabalho sendo executado, o método *get_local_id* retorna o índice do *thread* dentro do grupo de trabalho, enquanto o *get_group_id* traz o índice do atual grupo de trabalho no contexto global.

No OpenCL existem três níveis de memória, a global, compartilhada e privada. A memória global é acessível de todos os grupos de trabalho, a compartilhada a nível de grupos de trabalho, e a privada ao *thread* atual. A memória compartilhada pode ser utilizada em otimizações de algoritmos e sincronização dentro do grupo de trabalho, já que seu acesso é 10 vezes mais rápido quando comparado ao global.

Com isso, para a otimização do uso de memória, foi desenvolvido um *kernel* para o OpenCL que realiza a multiplicação de matrizes, copiando trechos de matrizes 8x8 para a memória compartilhada de forma cooperativa entre o grupo de trabalho. Todas os *threads* que fazem o uso daquele trecho da matriz, realizam suas operações na submatriz de forma acumulativa. O código otimizado do OpenCL é apresentado no Código 19. Nas linhas 28 e 29 cada *thread* copia um elemento da matriz global para a compartilhada, para que na linha 31 seja feita uma sincronização entre o grupo de trabalho. Em seguida nas linhas 32 a 36 são realizadas as multiplicações de forma acumulativa, para que ao final (linha 39) o resultado seja atribuído a memória global.

Código 19: Kernel de Multiplicação de matrizes e OpenCL

```
1
```

```
#define BLOCK_SIZE 8

__kernel __attribute__((reqd_work_group_size(BLOCK_SIZE,
    BLOCK_SIZE, 1)))
void matrix_mult(int n, __global int * M1, __global int * M2,
    __global int * MResp)
{

    int i = get_group_id(0);
    int j = get_group_id(1);

    //Identificação do work-item
    int idX = get_local_id(0);
    int idY = get_local_id(1);

    int global_column = get_global_id(1);
    int global_line = get_global_id(0);
    int qq = get_global_size(0);

    //Número de submatrizes a ser processada por cada worker
    int numSubMat = qq/BLOCK_SIZE;

    int resp = 0;
    __local int A[BLOCK_SIZE][BLOCK_SIZE];
    __local int B[BLOCK_SIZE][BLOCK_SIZE];

    for (int k=0; k<numSubMat; k++)
    {
        A[idX][idY] = BLOCK_SIZE * i + idX < n && BLOCK_SIZE * k
    + idY < n ? M1[BLOCK_SIZE * i + idX + BLOCK_SIZE * k + idY
    ]: 0;
        B[idX][idY] = BLOCK_SIZE*k + idX< n && (BLOCK_SIZE*j+idY)
    < n ? M2[BLOCK_SIZE*k + idX + (BLOCK_SIZE*j+idY)] : 0;

        barrier(CLK_LOCAL_MEM_FENCE);
        for (int k2 = 0; k2 < BLOCK_SIZE; k2++)
        {
            resp += A[idX][k2] * B[k2][idY];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    if(global_line < n && global_column < n){
        MResp[global_line * n + global_column] = resp;
    }
}
```

2.4.3.4 Visão geral das principais interfaces para GPU

Buscando o estudo e melhor entendimento das bibliotecas referenciadas nas seções 2.4.3.1, 2.4.3.2 e 2.4.3.3, foi escolhida a aplicação de multiplicação de matrizes como aplicação de teste das bibliotecas, devido a este ser um algoritmo de fácil paralelização, e utilizado em muitos livros como apresentação das bibliotecas.

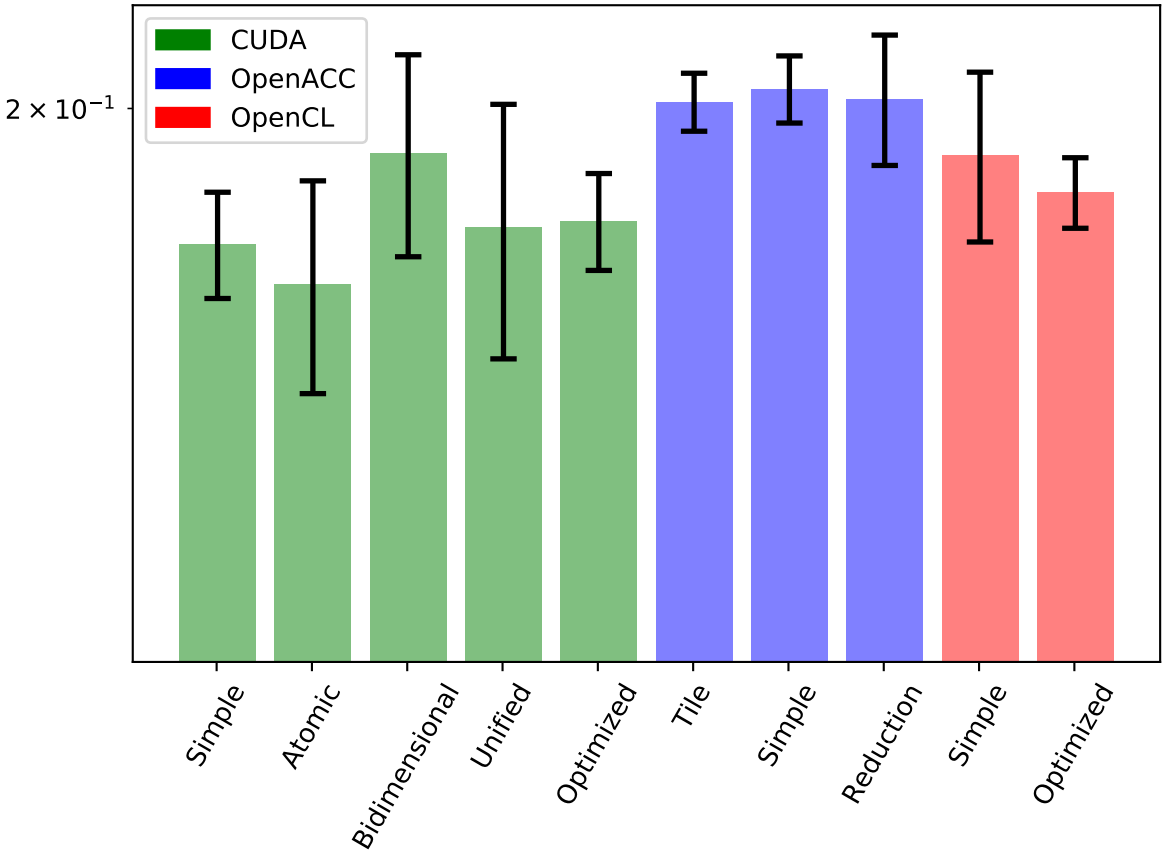
Com isso, todas as aplicações de multiplicação de matrizes desenvolvidas com as bibliotecas presentes neste artigo, foram submetidas a um *benchmark*, onde foi avaliado o tempo total de execução em matrizes de dimensões 10x10, 100x100, 1000x1000 e 2000x2000.

Os testes foram realizados em um *hardware* com CPU Intel Xeon E5-2620 v3 @2.40GHz de 12 núcleos, possuindo uma GPU Titan X (pascal) de 12GB e 3584 CUDA cores. Cada teste foi replicado 10 vezes e a opção de otimização -03 foi utilizada em todos copiladores.

A Figura 20 apresenta um gráfico com a média de tempo de execução e o desvio padrão entre os testes realizados com matrizes 10x10. Como pode ser visto, as versões otimizadas e simplificadas não apresentam muita diferença em cada biblioteca, algumas vezes sendo até melhores as mais simples. Como pode ser visto, as versões otimizadas de OpenCL e CUDA apresentam um desempenho semelhante as versões simples. Devido a computação realizar interação sobre poucos elementos, não se nota um ganho de desempenho quando comparado as versões simples, porém mesmo nesse caso o desempenho é satisfatório quando comparado as versões simples.

Nos testes de matrizes 100x100, apresentados na Figura 21, as versões otimizadas e simples apresentam desempenho semelhante, onde a versão bidimensional de CUDA e Tile do OpenACC começam a apresentar a degradação no desempenho. A degradação apresentada por estas versões acontece devido ao formato de processamento não utilizar bem o *cache* da GPU, já que os acessos não são sequenciais.

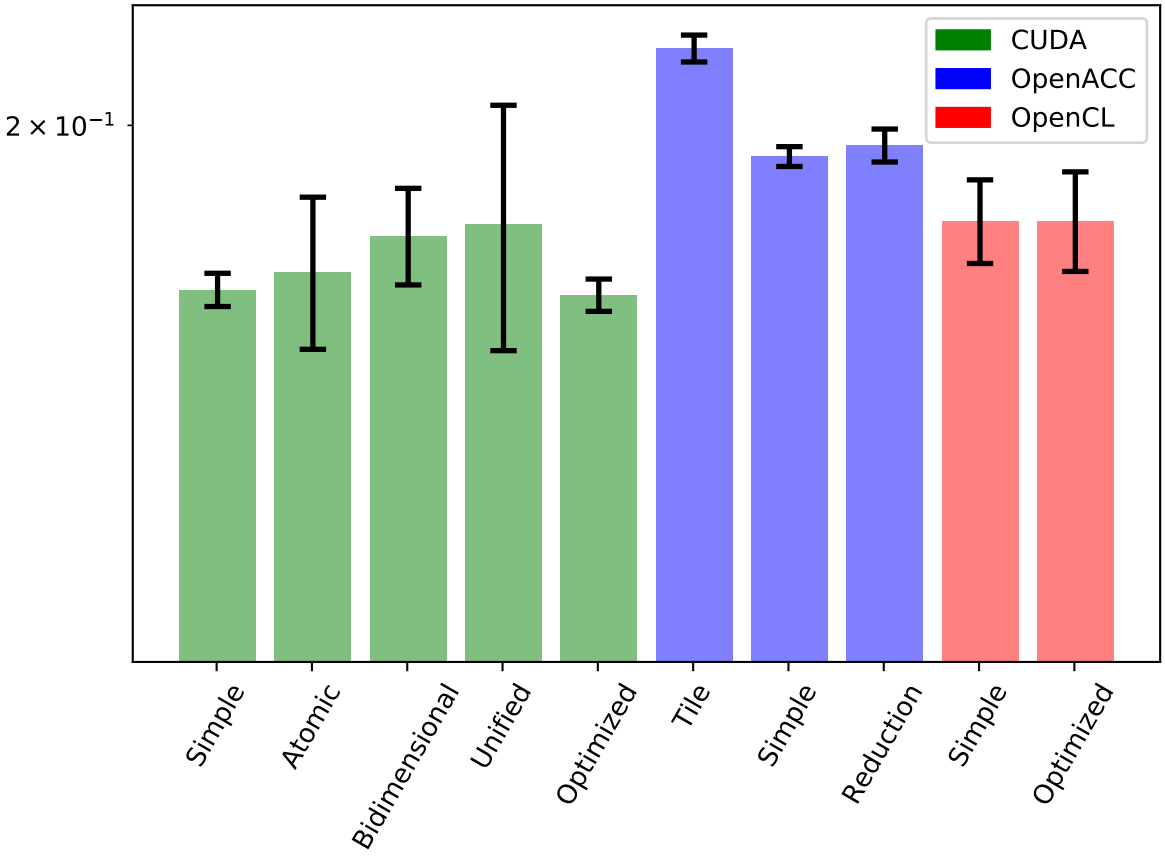
Figura 20: Multiplicação de matrizes 10x10



Na execução de matrizes 1000x1000, presente na Figura 22, a versão otimizada de CUDA e OpenCL começam a apresentar melhor desempenho que suas versões simples. A versão *Atomic* do CUDA apresenta um pior desempenho devido ao *overhead* de sincronização que é necessário ser feito na GPU. No OpenCL a versão otimizada apresenta um desempenho 4% maior que a versão simples enquanto que a versão do CUDA é 2% mais rápida que a simples. O ganho de desempenho das versões otimizadas é maior nesses casos devido ao uso da memória compartilhada da GPU, evitando acesso a memória global para dados que são reutilizados.

Em matrizes 2000x2000 ambas as versões otimizadas de CUDA e OpenCL apresentam desempenho semelhante, onde o CUDA apresenta-se 5,2% mais rápido que a versão otimizada do OpenCL. Os resultados das matrizes 2000x2000 são apresentado na Figura 23. Pode ser visto que com as matrizes grandes, as

Figura 21: Multiplicação de matrizes 100x100

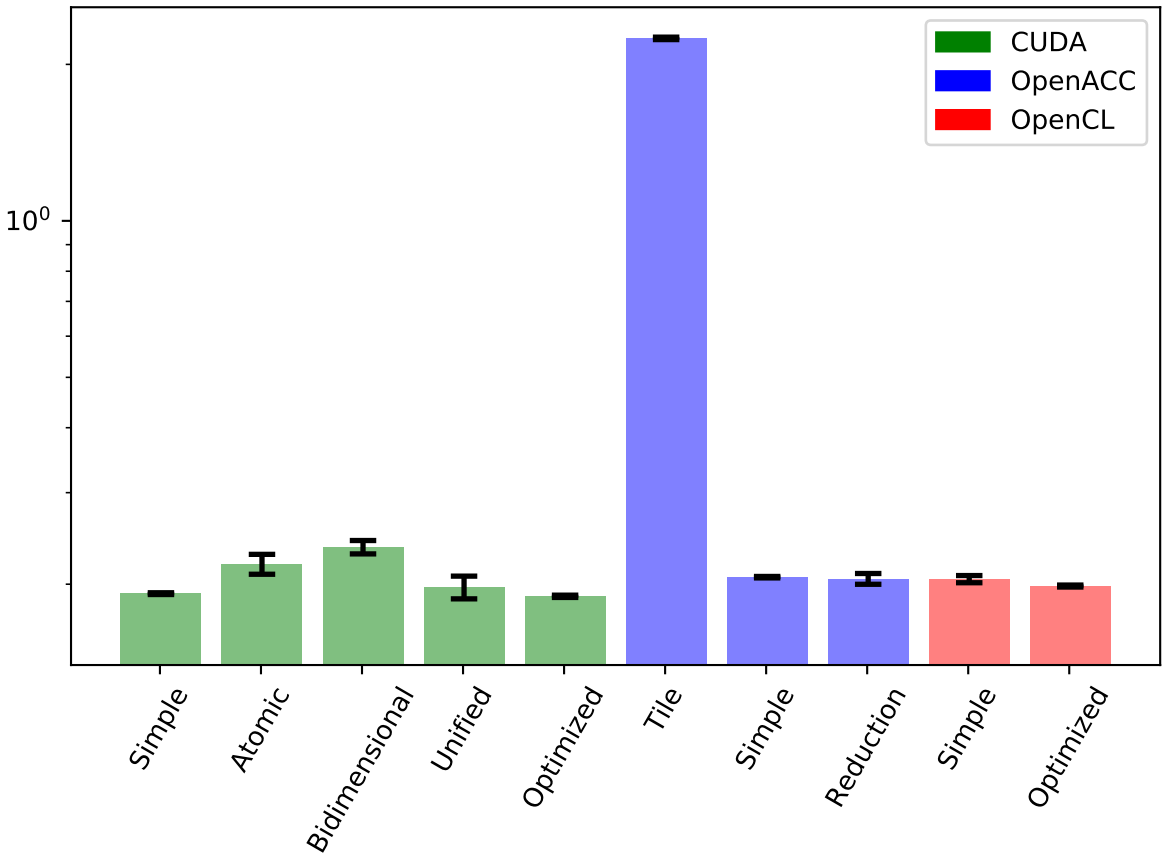


operações atômicas não são vantajosas devido a concorrência que acontece entre o mesmo recurso.

Utilizando as versões que apresentaram melhor desempenho em cada interface de programação, foi analisado o ganho de desempenho obtido em relação ao código sequencial. A Figura 24 apresenta o *speedup* das versões otimizadas de cada biblioteca. Devido à perda de desempenho nas versões 10x10 e 100x100, foram apresentadas apenas as versões que obtiveram desempenho melhor que a versão sequencial. Em matrizes 2000x2000 o ganho de desempenho teve uma variação entre 40 e 49 vezes, onde a interface que obteve maior desempenho foi a CUDA.

A partir dos testes realizados, percebe-se que o OpenACC, apesar a ser uma biblioteca de mais alto nível, consegue apresentar um desempenho semelhante as outras bibliotecas, desde que utilizado com os parâmetros de otimiza-

Figura 22: Multiplicação de matrizes 1000x1000



ções apropriados. O CUDA apresenta um bom desempenho mesmo sem grandes otimizações, como pode ser notado que a versão simples é apenas 17% mais lenta que a versão otimizada para matrizes 2000x2000. O OpenCL consegue atingir desempenho semelhante ao CUDA, porém são necessárias otimizações de acesso à memória.

A produtividade obtida no desenvolvimento da multiplicação de matrizes não foi analisada, porém Memeti et al. (2017) já realizou uma análise comparando as três bibliotecas apresentadas, juntamente com o OpenMP. Em sua análise, foram utilizadas diversas aplicações presentes nas ferramentas de *benchmark* SPEC e Rodinea. Foram analisadas 19 aplicações, presentes na Figura 25. Também é apresentada a área de domínio da aplicação, bem como as bibliotecas que possuíam implementações da aplicação.

A análise de produtividade foi realizada utilizando a ferramenta *CodeStat*.

Figura 23: Multiplicação de matrizes 2000x2000

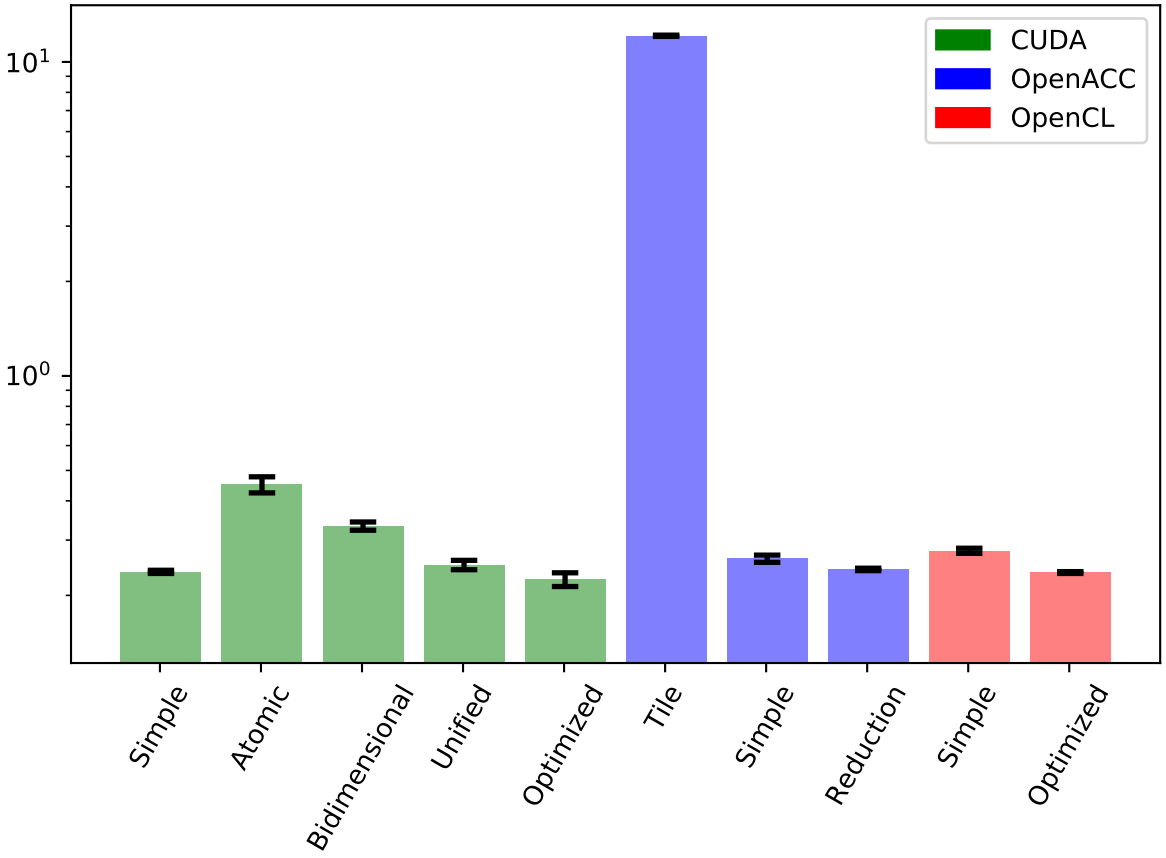
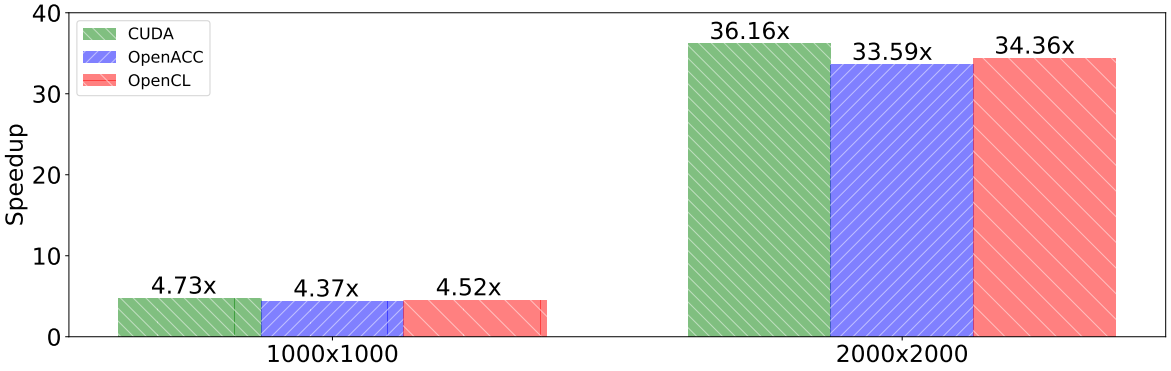


Figura 24: *Speedup* das versões otimizadas



De acordo com Memeti et al. (2017), o *CodeStat* analisa a quantidade de produtividade de uma biblioteca a partir da contagem do uso de funções específicas de cada biblioteca, como por exemplo o *pragma* no OpenACC ou o *cudaMemcpy* no CUDA.

Desta forma, o esforço foi medido através da quantidade de linhas do có-

Figura 25: Aplicações utilizadas na análise de produtividade

		SPEC Accel		Rodinia		
Application	Domain	OpenCL	OpenACC	OpenMP	OpenCL	CUDA
LBM	Fluid Dynamics	x	x			
MRI-Q	Medicine	x	x			
Stencil	Thermodynamics	x	x			
BFS	Graph Algorithms	x		x	x	x
CFD	Fluid Dynamics	x		x	x	x
HotSpot	Physics Simulation	x		x	x	x
LUD	Linear Algebra	x		x	x	x
NW	Bioinformatics	x			x	x
B+Tree	Search	x			x	x
GE	Linear Algebra	x			x	x
Heartwall	Medical Imaging	x			x	x
Kmeans	Data Mining	x			x	x
LavaMD	Molecular Dynamics	x			x	x
SRAD	Image Processing	x			x	x
BP	Pattern Recognition				x	x
k-NN	Data Mining				x	x
Myocyte	Biological Simulation				x	x
PF	Medical Imaging				x	x
SC	Data Mining				x	x

Fonte: (Memeti et al., 2017, p. 6)

digo que fizeram o uso das funções de bibliotecas. Na fórmula a seguir, o LOC_{par} representa a quantidade de linhas de uma biblioteca (medida pelo *CodeStat*), enquanto que o LOC_{total} representa a quantidade total de linhas da aplicação.

$$Esforco = 100 * LOC_{par}/LOC_{total}$$

Através da fórmula apresentada, é calculado o *delta* de esforço. Este método foi aplicado em todas as aplicações da Figura 25, e o resultado é apresentado na Figura 26.

Conforme Memeti et al. (2017), nas aplicações do *benchmark* SPEC, a programação com OpenCL requer um esforço bem maior que o OpenACC, é possível observar que em média a programação com OpenCL requer 6.7x mais esforço que o OpenACC. No Rodinia, a programação com CUDA requer em média a metade do esforço do OpenCL.

Figura 26: Análise de produtividade das aplicações de *benchmark*

	SPEC Accel		Rodinia		
	OpenCL[%]	OpenACC[%]	OpenMP[%]	OpenCL[%]	CUDA[%]
LBM	3.21	0.87			
MRI-Q	5.70	0.64			
Stencil	4.70	0.61			
BFS	6.95		4.86	9.07	12.50
CFD	5.83		2.53	9.00	8.08
HotSpot	4.75		2.67	13.18	8.20
LUD	5.78		2.30	9.72	7.82
NW	6.56			18.34	8.85
B+Tree	4.89			6.79	4.51
GE	9.63			14.21	9.76
Heartwall	5.34			6.74	3.97
Kmeans	2.80			2.67	2.17
LavaMD	4.61			9.24	7.74
SRAD	7.81			13.00	10.28
BP				12.21	5.95
k-NN				15.83	5.07
Myocyte				8.25	1.21
PF				17.83	9.47
SC				5.81	2.66

Fonte: (Memeti et al., 2017, p. 7)

Outro resultado encontrado é que o fator humano impacta na produtividade. Pode se observar que a aplicação BFS em OpenCL obteve um índice de 6.95 no SPEC, enquanto que no Rodinia o índice foi 9.07. O mesmo pode ser visto nas aplicações CFD, HotSpot, LUD, NW, B+Tree, GE, Heartwall, Kmeans, LavaMD, e SRAD.

2.5 TRABALHOS RELACIONADOS

Aplicações de *streaming* estão presentes em muitas áreas da computação. Para aplicações que precisam de alto desemepnho o paralelismo geralmente é implementado para a CPU. Por outro lado, o uso da GPU em aplicações de processamento de *stream* ainda é pouco explorado. Em seguida serão apresentados alguns dos trabalhos relacionados.

Ueno e Suzumura (2012) estudaram a utilização de GPUs para aplicações de *streaming*. Este estudo foi focado no desenvolvimento do *workload* de SVD (*Singular Value Decomposition*), que é utilizado em *data mining*, em aplicações como detecção de anomalias e processamento de imagens. A implementação utilizou

paralelismo de tarefas para a execução na GPU. A aplicação SVD foi testada em um conjunto de nodos de GPUs NVIDIA, conseguindo lidar com o processamento de 1525 sensores simultaneamente, obtendo um *speedup* de 7.6 vezes.

O trabalho teve como contribuição um algoritmo genérico para SVD e apresentou a caracterização de um *workload* de detecção de anomalias implementado em CUDA. O trabalho se relaciona com este trabalho ao realizar uma caracterização de aplicação de processamento de *stream* em CUDA. O presente trabalho buscou estudar aplicações de processamento de *stream*, porém não foi possível implementar uma aplicação na área de *data mining* devido a restrições de tempo. Por outro lado, o trabalho de Ueno e Suzumura (2012) implementou paralelismo de tarefas na GPU, enquanto este trabalho implementou apenas paralelismo de dados na GPU, deixando o paralelismo de tarefas para a CPU.

Liang (2011) realizou a implementação de uma *Hopfield Neural Network* utilizando a biblioteca CUDA. Em seus testes o desempenho da versão em GPU variava conforme o número de neurônios da rede neural e consegue alcançar um *speedup* de até 6x. Para a paralelização foi realizada uma redução no processo de propagação da rede neural, realizando assim o paralelismo de dados. O trabalho contribuiu com uma versão genérica de *Hopfield Neural Network*, que pode ser utilizado em algoritmos genéricos e aplicações de agendamento.

O estudo de Liang (2011) se assemelha ao presente trabalho por realizar a implementação de uma aplicação de processamento *stream* utilizando GPU. Este estudo não realizou nenhuma implementação de redes neurais em GPU, pois este tipo de aplicação já é comum entre os *Frameworks* relacionados a *machine learning*.

O H.264 em CUDA foi implementado por Wu et al. (2012). Em seu estudo primeiramente explorado o paralelismo do *encoder*, para que em seguida fosse realizada a implementação em CUDA. O trabalho apresentou uma implementação eficiente do H.264, onde foi otimizando o uso de memória através do eficiência em transferências entre CPU e GPU. Para o mapeamento do paralelismo do H.264 foi

necessário a alteração da estratégia de paralelização, onde foi aumentada a quantidade de dados processados em paralelo para utilizar a GPU de forma eficiente. Wu et al. (2012) apresenta otimizações de memória que poderão ser utilizadas em múltiplas aplicações e bibliotecas, que serão implementadas neste trabalho.

Ozsoy e Swany (2011) realizaram uma implementação do LZSS (*Lempel–Ziv– Storer–Szymanski*) em CUDA. LZSS é um algoritmo de compressão de dados sem perdas. O *speedup* obtido foi de 18 vezes comparado a versão serial, e três vezes mais rápido que a versão paralelizada para CPU. O estudo implementou o LZSS em GPU não apresentou nenhum aumento na quantidade *storage* necessário, quando comparado a versão em CPU.

O trabalho foi o primeiro a apresentar compressão de dados sem perdas em GPU que superasse versões em CPU. Em sua implementação, todo o processo de compressão foi movido para a GPU, enquanto que na CPU foi utilizada uma *pipeline* para realizar a transferência de dados. No presente trabalho, o LZSS também foi implementado, porém apenas a etapa que possui paralelismo de dados foi transferida para a GPU (busca de maior ocorrência), mantendo o restante do processo em CPU. Este trabalho também realizou a implementação em GPU para CUDA e OpenCL, enquanto que o paralelismo de tarefas na CPU foi realizado utilizando a DSL SPar.

Outra implementação realizada por Suttisirikul e Uthayopas (2012), foi a deduplicação em GPU utilizando CUDA. Deduplicação é uma técnica de compressão que faz com que dados repetidos sejam salvos apenas uma vez. O *streaming* implementado processa os *fingerprints* (hashes de blocos de dados), e foi paralelizado em nível de blocos, onde cada *thread* gera o SHA256 de um bloco. Mesmo o SHA256 apresentando um desempenho mais lento executado em *thread* na GPU, a quantidade de *threads* executando fez com que o *throughtput* fosse maior, aumentando assim a quantidade de *fingerprints* geradas por segundo. Com a implementação é possível fazer utilização de deduplicação para *backups* em nuvem, tanto em nível de cliente como servidor. O trabalho mostra que o *overhead* de

transferência de memória entre CPU e GPU deixa de ser um problema, conforme a quantidade de dados aumenta.

O trabalho de Suttisirikul e Uthayopas (2012) realizou uma implementação própria do *dedup*, porém não realizou a compressão dos blocos deduplicados como no presente trabalho. Além disso, este trabalho realizou a implementação do *dedup* com paralelismo em CPU (utilizando SPar) e GPU (CUDA e OpenCL).

Uma implementação do PARSEC Benchmark Suite foi desenvolvida por Ray (2010). Dentre as *workloads* implementadas estão a ferret e X.264, que são aplicações de processamento de *stream*. O ferret é utilizado para analisar similaridades entre imagens, onde é utilizada *pipeline* com os seguintes estágios: segmentação, extração de *features*, indexação e conjunto de candidatos. Devido ao ferret ser uma aplicação *memory bound*, as otimizações realizadas envolveram o uso de memória compartilhada na GPU.

Para o X.264, foi implementado em GPU o processo de *Motion Estimation*, onde cada *thread* busca o melhor macro bloco para representar aquele conjunto de *frames*. Os *workloads* foram implementados e otimizados, obtendo um *speedup* entre 4 e 6 vezes mais rápidos que as versões em CPU. Este trabalho utilizou o PARSEC Benchmark como base de implementação da aplicação *dedup*, porém não fez o uso dos outros *workloads* como Ray (2010).

Su et al. (2012) realizou uma comparação de desempenho entre as interfaces de programação OpenCL e CUDA. Os testes utilizaram três *benchmarks* com as aplicações *Sobel Filter* e *Gaussian Filter*, e foram realizados em diferentes sequências de filmes. O CUDA apresentou um desempenho entre 3,8% e 5,4% melhor que o OpenCL devido ao tempo de compilação do OpenCL.

O trabalho de Su et al. (2012) se assemelha ao presente estudo em fazer comparações entre mais de uma biblioteca. Por outro lado apesar de o *benchmark* ser realizado com uma aplicação de processamento de *stream*, o trabalho não realizou nenhum mapeamento e implementação em GPU.

De e Gupta (2015) paralelizou a aplicação de dedispersão coerente, voltada ao *Giant Metrewave Radio Telescope*. A aplicação processa o *Fast Fourier Transform* para conseguir processar os dados em tempo real, desta forma a contribuição do trabalho foi a *pipeline* em tempo real, permitindo um *throughput* que não necessitava salvar dados em disco. O *Speedup* obtido foi de 2.5x. A aplicação se relaciona ao utilizar uma aplicação de processamento de *stream* em GPU em tempo real e ao utilizar *pipeline* para paralelização em CPU.

Uma implementação de *bzip2*, utilizada para comprimir dados, foi realizada por Deshpande (2014). Em seu trabalho, a etapa de *Burrows-Wheeler Transform* - BWT foi paralelizada de forma hibrida entre a CPU e GPU. A GPU realizou a ordenação dos dados de forma parcial, enquanto a CPU realizou um *Merge sort* com a ordenação inicial da GPU. Na versão hibrida do BWT o *speedup* foi de 2.9x, sendo melhor que versões que faziam uso exclusivo de CPU ou GPU. O formato hibrido de paralelismo faz com que o uso da GPU e CPU seja aproveitado ao máximo, dividindo as tarefas entre os dois dispositivos.

A implementação de Deshpande (2014) se assemelha a este trabalho por ser uma aplicação de compressão de dados em GPU, como o LZSS e o Dedup realizados neste trabalho. O uso híbrido dos dispositivos não foi realizado neste trabalho, porém a implementação de diferentes etapas em formato de *pipeline* foi implementada, onde etapas sequenciais executam em CPU e etapas com paralelismo de dados executam em GPU.

Desta forma, o Quadro 3 apresenta uma visão geral do trabalhos relacionados, onde foram categorizados alguns aspectos, como aplicação e biblioteca utilizadas.

Quadro 3: Trabalhos relacionados

Trabalho	Objetivo	Aplicação	Biblioteca	Metodologia	Estratégia	Hardware	Desempenho	Contribuição	Ano
Ueno e Suzumura (2012)	Detecção de anomalias em sensores com o uso de GPU	Singular Spectrum Transformation	CUDA	Análise de desempenho e testes em GPU.	Map Reduce	NVIDIA Geforce 8800 GTS 512 e CPU Quadcore.	7.6x	Otimização da aplicação SVD paralelizada para GPU em processamento de Stream	2012
Liang (2011)	Implementação de Hopfield Neural Network em GPU	Hopfield Neural Network	CUDA	Exploração de paralelismo e análise de desempenho em GPU.	Reduce	GeForce GTX 480 e Intel Xeon W2540 2.93GHz.	6x	Um algoritmo genérico para Hopfield Neural Network que pode ser utilizado para aplicações como algoritmos genéricos e problemas de agendamento	2011
Wu et al. (2012)	Implementação de um Encoder H.264 em GPU utilizando CUDA	H.264 Encoder	CUDA	Exploração de paralelismo, testes e análise de desempenho em CPU e GPU.	Stencil, Reduce	NVIDIA GTX260+ e Intel E8200 Dual-core	15.77x a 17x	H264 mais rápido que o estado da arte através da eficiência no uso de memória e paralelismo multinível.	2012
Ozsoy e Swany (2011)	Implementar o algoritmo LZSS utilizando a GPU com melhor desempenho que a versão em CPU	LZSS	CUDA	Exploração de paralelismo em GPU, testes, análise de resultados e comparação com outras aplicações.	Map Reduce	GeForce GTX 480 e Intel(R) Core(TM) i7 CPU 920 2.67GHz.	18x	Primeira implementação de Loseless data compression em GPU com desempenho melhor que CPU.	2011
Suttisirikul e Uthayopas (2012)	Implementar o processo de geração de fingerprint para deduplicação utilizando GPU	Deduplication	CUDA	Exploração de paralelismo em GPU e análise de desempenho.	Stencil	Intel(R) Xeon(R) E5620 e Nvidia Tesla M2050.	53x	Geração de fingerprint para deduplicação de dados em GPU	2012
Ray (2010)	Implementar o PARSEC Benchmark Suite em CUDA.	Blackscholes, Ferret, Fluidanimate, Raytrace, Streamcluster, Swaptions, X264.	CUDA	Exploração de paralelismo em GPU e otimização.	Map, Reduce, Stencil	NVIDIA GTX 280	4x a 6x	Implementação de sete workloads do PARSE Benchmark Suite em GPU com CUDA.	2010
Su et al. (2012)	Comparação de aplicações em GPGPU através de uso de benchmark.	Sobel Filter e Gaussian Filter.	CUDA e OpenCL	Benchmarks e análise de desempenho em GPU	-	Nvidia Quadro 4000 e CPU L2200 1.2GHz.	1.9x.	Pouca diferença entre OpenCL e CUDA em implementações para GPU.	2012
De e Gupta (2015)	Implementar dedisperção de coerência em tempo real em GPU	Fast Fourier Transforms	CUDA	Exploração de paralelismo em CPU e GPU	Pipeline, Map	Tesla C2075 GPU, Intel Xeon X5550	1.6x a 2.5x throughput	Processamento em tempo real de FFT sem necessidade de salvar dados em disco para o processamento no giant metrewave radio telescope	2015
Deshpande (2014)	Combinar paralelismo de tarefas com paralelismo de dados através de CPU e GPU	Bzip2(BWT)	CUDA	Exploração de paralelismo em CPU e GPU	Pipeline, Work Sharing e Map	CPUs Intel Core 2 Duo P8600, i7 920, i7 980x. GPUs Nvidia 8600M GT, GTX 480, T10	2.9x	O uso de pipeline de forma efetiva faz com que o uso de GPU em conjunto com o CPU obtenho melhor desempenho	2014
Este trabalho	**Implementar e analisar aplicações de processamento de *stream* em GPU.**	**Filtro Sobel, LZSS e *Dedup*.**	**CUDA e OpenCL**	**Exploração de paralelismo em CPU e GPU e análise de desempenho.**	**-**	**Titan Xp e E5-2620 v3 @2.40GHz de 12 núcleos**	**-**	**-**	**2018**

No presente trabalho, serão utilizados CUDA e OpenCL como interfaces de programação. Através das aplicações que serão implementadas, serão estudado as melhores formas de implementação de aplicações de *streaming* no ambiente heterogêneo.

CAPÍTULO 3: EXPERIMENTOS E RESULTADOS

3.1 ESCOLHA DE BIBLITECAS E APLICAÇÕES

A escolha das bibliotecas de programação paralela para a GPU, levou em consideração os resultados dos testes de multiplicação de matrizes. A partir dos testes de multiplicação de matrizes, realizados e apresentados na seção 2.4.3.4, pode-se perceber que as bibliotecas CUDA e OpenCL foram as mais eficientes em relação ao desempenho.

Além de serem mais eficientes, pode-se perceber que o paradigma de programação é semelhante, tornando mais simples a implementação de aplicações semelhantes. Por estes motivos, para as aplicações implementadas neste trabalho foram escolhidas as bibliotecas CUDA e OpenCL.

Por outro lado, para a escolha das aplicações de processamento de *stream* foram considerados dois aspectos: as aplicações já estudadas pelos trabalhos relacionados e a disponibilidade de código fonte sequencial.

3.2 APLICAÇÕES

Nesta seção, serão apresentadas as aplicações de processamento de *stream* escolhidas para o trabalho.

3.2.1 Filtro Sobel

O operador Sobel é uma transformação aplicada sobre uma imagem a fim de destacar suas bordas. Esta aplicação foi adicionada no trabalho por representar um uso de aplicaçoes de processamento de *stream* em transformações aplicadas a imagens e vídeos. Esta transformação é utilizada em processamento de imagem e visão computacional para detecção de bordas. Um exemplo desta transformação é apresentada na Figura 27, a esquerda é apresentada a imagem original, enquanto na direita é apresentado o resultado.

Figura 27: Exemplo de imagem do filtro sobel

De acordo com Kaehler e Bradski (2016), o filtro *sobel* aplica duas operações de matrizes 3x3 para cada pixel da imagem. Esta operação é apresentada na fórmula a seguir. Primeiramente é cálculado o valor do operador sobel para a horizontal e vertical, apresentado nas fórmulas 3.1 e 3.2.

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A \tag{3.1}$$

$$G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A \tag{3.2}$$

Em seguida é calculado o valor do pixel, através da equação 3.3.

$$G = \sqrt{G_x * G_y} \tag{3.3}$$

A matriz A presente no cálculo do filtro *Sobel* é uma matriz 3x1 que representa o pixel anterior, atual e próximo sendo processado.

Para esta simulação, a aplicação de filtro sobel lê todas as imagens de um diretório e destaca as bordas das imagens. O operador sobel é aplicado em cada pixel da imagem, que leva em consideração os pixeis vizinhos para obter um valor que representa a diferença de corres. Foram realizadas implementações em GPU utilizando o CUDA e o OpenCL. Além do paralelismo na GPU, também foi implementado o paralelismo em CPU utilizando o SPar.

O Código 21 apresenta o operador/função *Sobel* em formato de um *kernel* CUDA. Como pode ser observado, a execução ocorre em blocos que são calculados baseado no tamanho da imagem. O *kernel* aplica o filtro no índice atual da imagem e escreve o resultado em uma variável global, que ao final é transferida para a CPU.

O paralelismo em CPU implementou um padrão *Farm*, e é apresentado no Código 20. Para cada arquivo, um trabalhador lê, aplica o filtro e salva o resultado. Note que ambos os códigos são versões totalmente voltados para a sua arquitetura alvo. Para realizar a combinação, foram mantida as anotações do Código 20, porém, ao invés de chamar o operador sobel sequencial na linha 10, inseriu-se todo o código necessário para inicializar e chamar o *kernel* do Código 21, usando a biblioteca CUDA.

Código 20: Aplicação com SPar.

```
DIR *dptr = opendir(...);
struct dirent *dfptr;
[[spar::ToStream, spar::Input(
    dptr, dfptr, tot_img, tot_not)
    , spar::Output(tot_img,
    tot_not)]]
while ((dfptr = readdir(dptr)) !=
     NULL){
  // preprocessing
  if (file_extension == "bmp"){
   tot_img++;
   im = read(name, h, w);
   [[spar::Stage, spar::Input(h, w
     , im, newname), spar::Output(
     new_im), spar::Replicate(2)]]{
    new_im = sobel(im, h, w);
    write(newname, new_im, h, w);
   } //end stage
  }else{ tot_not++; }
}
```

Código 21: Sobel em CUDA

```
__global__ void sobel_k(unsigned
    char *fi, unsigned char *im,
    int w, int h){
     int y = blockIdx.y * blockDim
    .y + threadIdx.y + 1;
     int x = blockIdx.x * blockDim
    .x + threadIdx.x + 1;
     unsigned char b[3][3];
     if (x < (w-1) && y < (h-1)){
          for (int v = 0; v < 3; v
    ++){
               for (int u = 0; u <
    3; u++){
                    b[v][u] = im[(((y
     + v - 1) * w) + (x + u - 1))
    ];
                    fi[((y*w)+x)]=
    Sobel(b);
               }
          }
     }
}
```

A transferência de dados entre CPU e GPU, para fins de teste, foi implementada de três formas diferentes. A primeira faz a gestão do dados de forma explicita com a API do CUDA. A outra utilizou o recurso *stream*, permitindo que a comunicação CPU e GPU e o processamento dos *kernels* ocorram de forma simultânea. Por último, foi utilizado um recurso *Unified Memory* que faz as transferências automaticamente.

Nas implementações em OpenCL o código foi lido de um arquivo e compilado de forma *just in time*. A execução foi realizada usando uma *Command queue* para todas as imagens.

3.2.1.1 Análise dos resultados

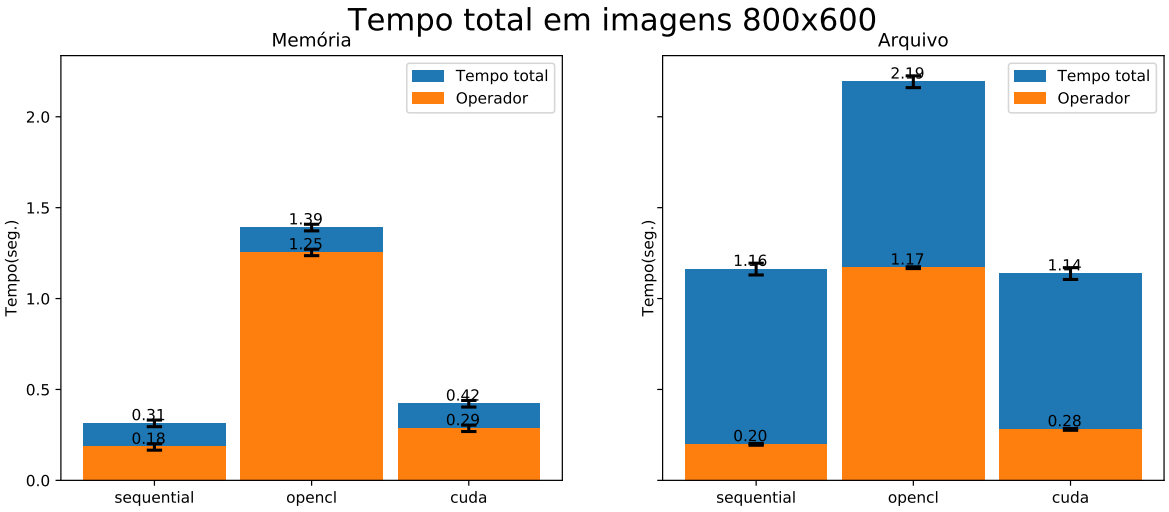
Para analisar o desempenho das versões implementadas, foram usadas 100 imagens com dimensões de 800x600, 3000x2250, 5000x5000 e 10000-x10000. O *hardware* utilizado foi um computador com CPU Intel Xeon E5-2620 v3 @2.40GHz

de 12 núcleos, possuindo uma GPU Titan X (pascal) de 12GB e 3584 CUDA cores.

Os testes foram realizados tanto em memória quanto em arquivo, ou seja, no primeiro os arquivos utilizados são pré-carregados para a memória RAM, para que em seguida seja medido o tempo total da aplicação, e a leitura realizada seja apenas a cópia da memória. Por outro lado, nos testes com arquivo, o tempo de leitura em disco também é considerado.

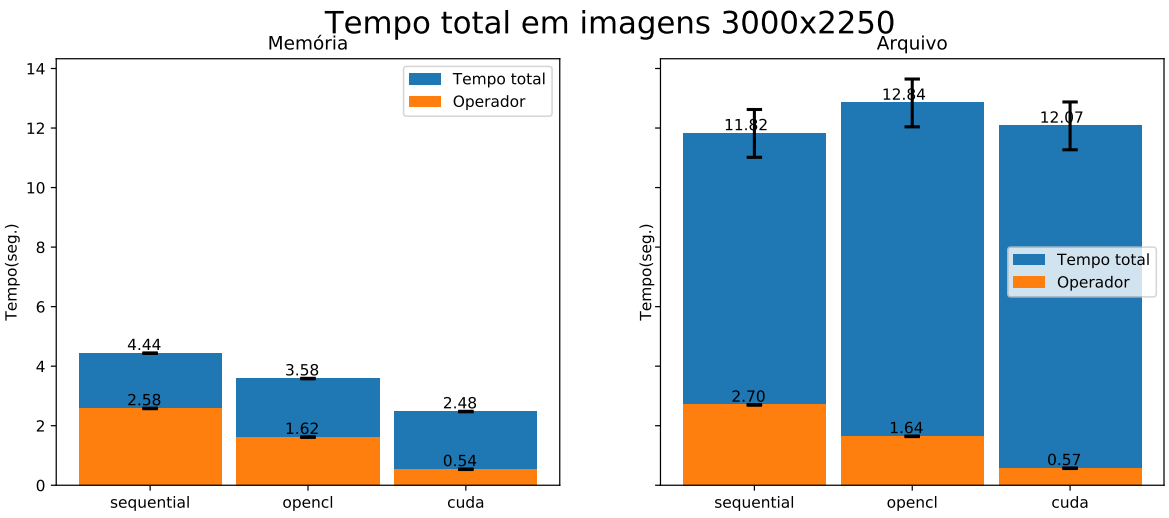
O primeiro teste foi a comparação do desempenho das implementações em GPU com a sequencial. A Figura 28 apresenta o desempenho nas diferentes com a carga de dados 800x600, destacando o tempo total da execução e o tempo da região paralela (operador sobel em CUDA e OpenCL). Nota-se no teste em memória que o OpenCL levou um tempo bem maior que o CUDA. Isso ocorre devido a compilação *Just In Time* realizada pelo OpenCL, que é medida junto a região paralela. Com a carga de dados de 800x600 o melhor desempenho foi da versão sequencial.

Figura 28: CPU vs GPU 800x600



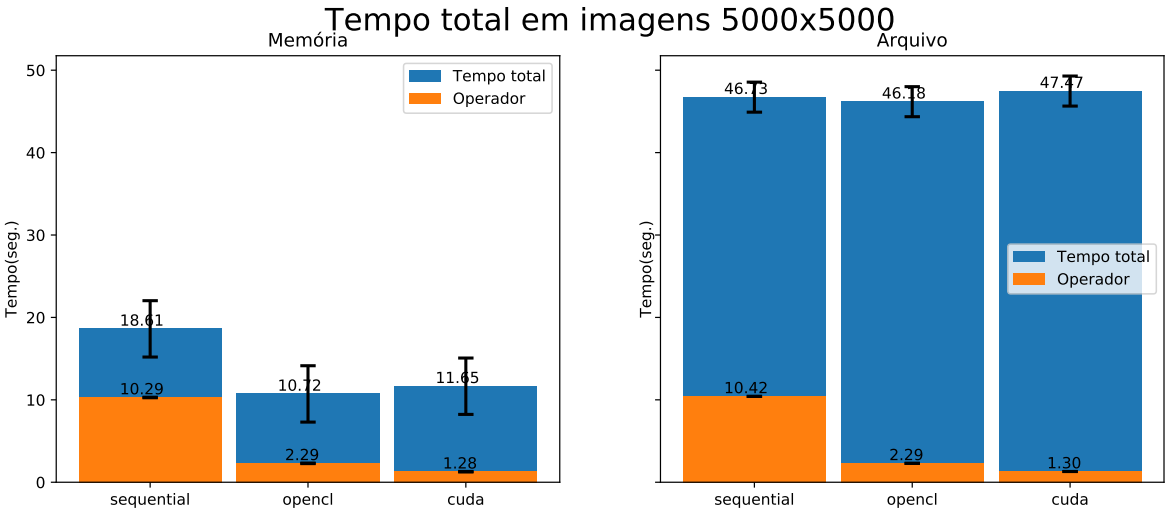
Na Figura 29 é apresentado o tempo nas imagens 3000x2250. Nota-se que o tempo de compilação do OpenCL ainda faz com que ele seja mais lento, porém ambas as implementações em GPU obtiveram um melhor desempenho em memória. Nos testes em arquivo nota-se que o tempo de I/O representa grande parte da execução, e as versões em GPU não apresentam *speedup*.

Figura 29: CPU vs GPU 3000x2250



Com a carga de dados 5000x5000, apresentado na Figura 30, nota-se que o OpenCL em memória apresenta um melhor desempenho que o CUDA. Nos testes em arquivo não há diferença no tempo total das execuções.

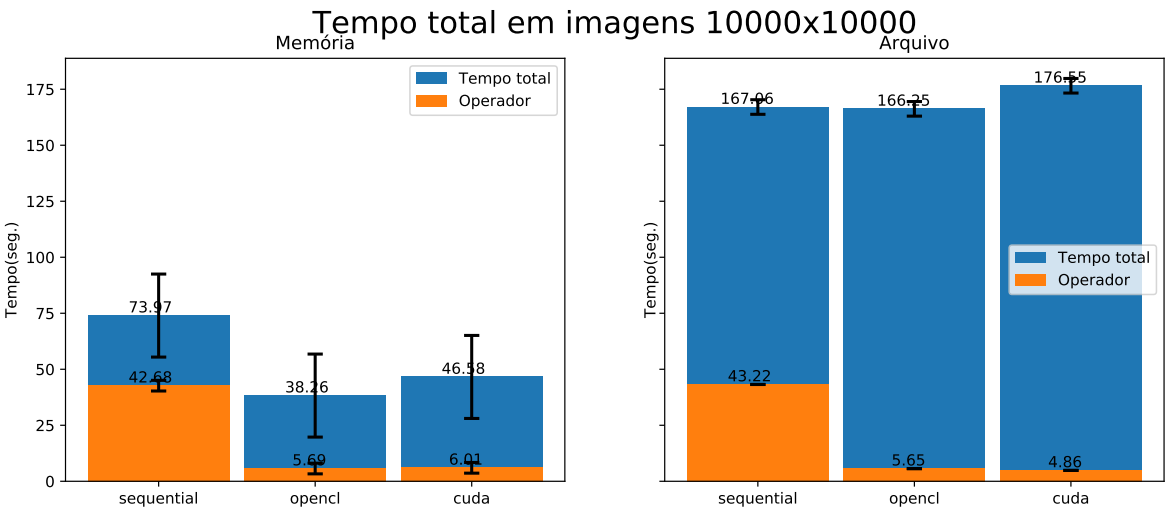
Figura 30: CPU vs GPU 5000x5000



Na maior carga de dados, composta de imagens 10000x10000 e apresentado na Figura 31, nota-se um desempenho quase duas vezes maior em GPU nos testes em memória, enquanto que nos testes de arquivo não há ganho de desempenho em GPU.

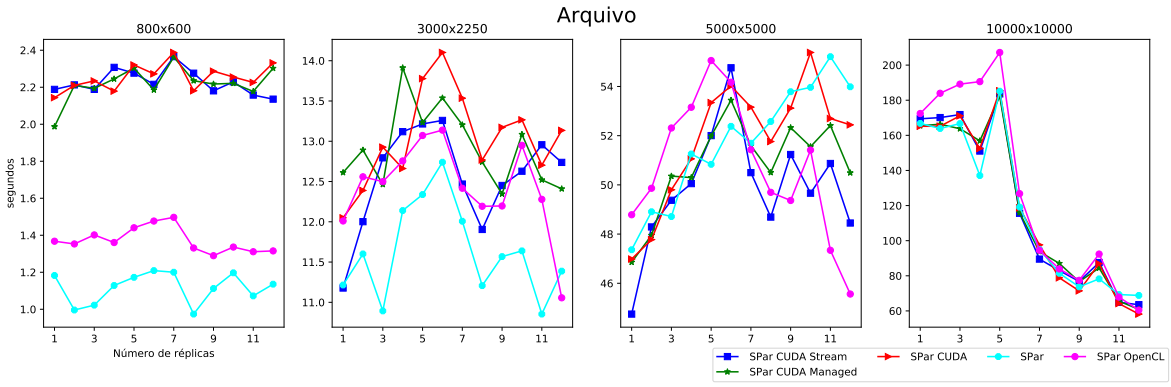
A SPar permitiu o paralelismo de *Stream* para CPU de forma simples e ao

Figura 31: CPU vs GPU 10000x10000



mesmo tempo eficiente conforme podemos ver na Figura 32, onde o tempo total de processamento das diferentes versões desenvolvidas foi plotado. Nota-se que a execução em imagens de baixa resolução na GPU apresentou menor desempenho que as versões paralelizadas apenas em CPU (versão SPar). Em imagens a partir de 5000x5000, o desempenho entre versões que utilizam GPU se assemelha ao uso exclusivo de CPU. Embora existe o suporte ao paralelismo nos dois níveis arquiteturais, a frequente cópia dos dados e as operações de I/O afetaram a escalabilidade nesta aplicação. Na carga de imagens 5000x5000, nota-se claramente um problema de balanceamento de carga com a SPar, ocorrendo também nas outras cargas quando testado com um número de réplicas em específico.

Figura 32: Resultados com o uso combinado do paralelismo em CPU e GPU em arquivo.

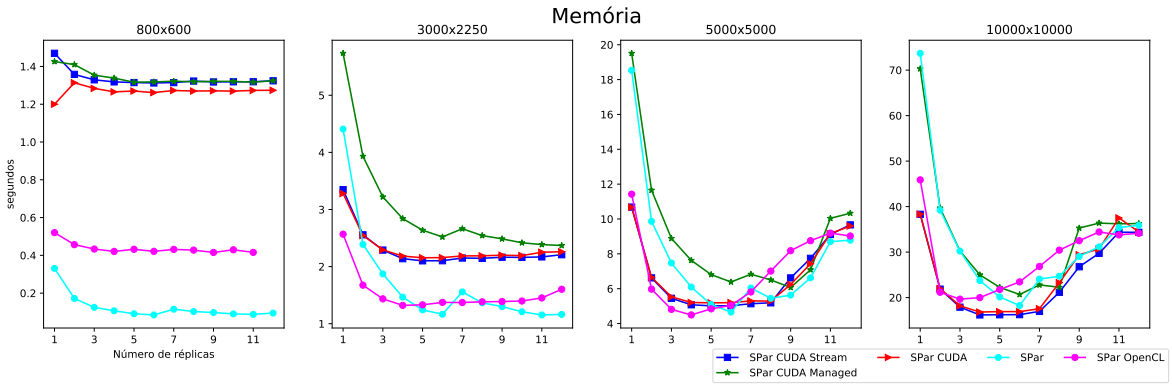


Devido ao I/O ter um grande impacto nos resultados, foram realizados tes-

tes em memória, apresentados na Figura 33. Em imagens 800x600, nota-se que não há grande distinção no tempo nos diferentes números de *threads*. Nota-se que o OpenCL teve um melhor desempenho que o CUDA, porém não foi possível executar o teste com 12 *threads*, devido a um problema na aplicação que ainda não foi identificado.

Nas demais execuções, nota-se um desbalanceamento na execução com 7 *threads*, tanto em CPU quanto em GPU. Nas imagens 5000x5000 e 10000x10000 nota-se que o desempenho piora a partir dos 7 *threads*. Nota-se que o resultado das versões em CPU, foi semelhante ao trabalho feito por Griebler et al. (2015), onde a mesma aplicação foi paralelizada para CPU utilizando as bibliotecas *SPar*, *FastFlow* e *OpenMP*, porém em uma outra configuração de *hardware*.

Figura 33: Resultados com o uso combinado do paralelismo em CPU e GPU em memória.



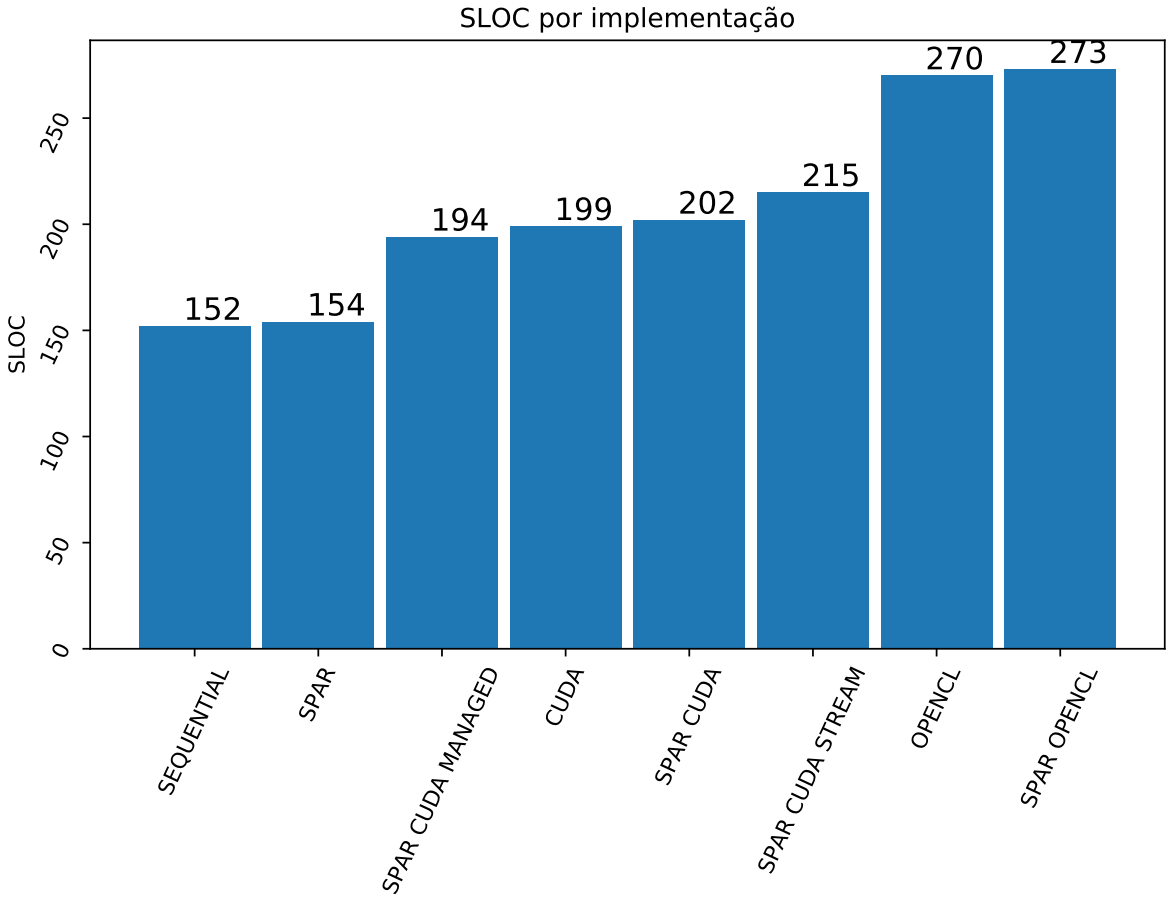
Em relação a intrusão de código obtida, foi medida a quantidade de linhas de código em cada implementação. Na Figura 34 é apresentado a quantidade de linhas de código de cada implementação. Nota-se que as versões que usa o *Unified memory* do CUDA reduzem as linhas de código necessárias, já que mesmo com SPar, a versão SPar CUDA Managed obteve menos linhas de código que a versão em CUDA.

As versões com OpenCL necessitaram de mais linhas de código, já que a compilação *Just in Time* precisa ser programada. A adição de paralelismo com SPar representou pouca diferença das versões sem SPar, já que só foi necessária

a anotação do código existente.

As versões que usam o recurso de *stream* do CUDA apresentam um incremento nas linhas de código devido as chamadas assíncronas realizadas. Não foram necessárias mais linhas de código para *Stream* devido a um recurso do CUDA que gerencia a *Stream* de acordo com o *thread* do CPU, tirando esta responsabilidade do programador.

Figura 34: Linhas de código por implementação do filtro Sobel.



3.2.2 LZSS

Conforme David Salomon G. Motta (2007), *Lempel–Ziv–Storer–Szymanski* - LZSS é um algoritmo de compressão sem perdas da família *Lampel-Ziv*. Este algoritmo é utilizado em várias aplicações de compressão de dados, dentre estes o PKZip e o RAR. Por sua característica de leitura contínua de arquivos (ou até mesmo dados em rede) ele é considerado uma aplicação de processamento de

stream. Dado um conjunto de dados lido de forma contínua, o algoritmo executa a compressão e gera um resultado.

Um exemplo desta compressão de texto pode ser visto no Código 22 e 23, extraidos de (Ozsoy e Swany, 2011, p.2). Neste exemplo o arquivo original com 102 caracteres é reduzido a 56 caracteres.

Código 22: Entrada de dados do LZSS.

```
 0:  I meant what I said
20:  and I said what I meant
44:
45:  From there to here
64:  from here to there
83:  I said what I meant
```

Código 23: Resultado do LZSS

```
 0:  I meant what I said
20:  and(12,7)(7,8)(2,5)
30:
31:  From there to (51,4)
47:  f(46,4)(51,8)(50,5)
55:  (24,19)
```

Os algoritmos de compressão da família LZ utilizam o conceito de *slidingWindow*, que servem como um dicionário que é alimentado pela sequência de bytes do próprio arquivo. A família LZ tenta reutilizar ao máximo os bytes presentes no *slidingWindow* para comprimir o resultado. A reutilização dos bytes é feita através de uma busca no *slidingWindow* pela maior ocorrência de um conjunto de *bytes*. Quando o algoritmo consegue encontrar uma ocorrência, ele salva o índice em que ele encontrou aquela ocorrência no *slidingWindow*, e o tamanho da ocorrência. Com esta transformação, é possível salvar o resultado omitindo os dados que já estão presentes no arquivo.

Como pode ser visto no Código 23, as referências são salvas com o índice do *slidingWindow* e o tamanho da ocorrência. Para a codificação do código 22, é realizada uma leitura sequencial do texto de entrada. Para cada caractere, é realizada uma busca no dicionário existente (*slidingWindow*) pela maior ocorrência da sequência atual (*uncodedLookahead*). Na primeira vez que um caractere é lido, ele é salvo no arquivo com um bit que representa que aquele caractere não está codificado (*uncoded*). Quando uma ocorrência de um conjunto de caracteres já está presente no *slidingWindow*, é salva apenas uma referência do *slidingWindow*, representado pelo índice e tamanho no *slidingWindow*.

O *slidingWindow* no LZSS possui um tamanho máximo (neste exemplo 4096), desta forma, quando o *slidingWindow* preenche 4096 caracteres, a leitura faz com que o *slidingWindow* substitua os caracteres mais antigos pelos novos. O *uncodedLookahead* representa os texto utilizado para a busca da maior ocorrência. Ele deve ser do tamanho da maior ocorrência que o codificador LZSS suporta. O tamanho máximo de uma ocorrência é dado pelo número de bits utilizado para escrever uma ocorrência. Na aplicação implementada o tamanho escolhido foi de 12 bits para representar o índice do *slidingWindow* e 6 bit para representar o tamanho.

Esta lógica pode ser vista no pseudo código sequencial de codificação do LZSS, presente no pseudocódigo 24. onde é dada a entrada da variável arquivo, para que seja escrito o resultado na saída do programa.

Código 24: Pseudocódigo de compressor LZSS

```
ALGORITMO "Codifica LZSS":
    VAR:
        uncodedLookahead := lista(18)
        Encoded := 1
        Uncoded := 0
        slidingWindow := lista(4096)
        restante := 0
        arquivo,
        i
    INICIO:
        PARA i DE 0 ATÉ 18 FAÇA
            AdicionaCaractere(uncodedLookahead, LerLetra(arquivo))
        FIM PARA
        ENQUANTO tamanho(uncodedLookahead) > 0 FAÇA
            //Busca maior ocorrencia do uncodedLookahead no slidingWindow
            ocorrencia := MaiorOcorrencia(slidingWindow,
    uncodedLookahead)
            SE ocorrencia.tamanho > 2 FAÇA
                EscreveBit(Encoded)
                Escreve(ocorrencia.indice)
                Escreve(ocorrencia.tamanho)
            SENAO FAÇA
                EscreveBit(Uncoded)
                Escreve(uncodedLookahead[0])
                ocorrencia.tamanho := 1
            FIM SE

            restante := ocorrencia.tamanho
```

```
28              ENQUANTO restante > 0 E NÃO Fim(arquivo) FAÇA
29                  // Move primeiro caractere de uncodedLookahead para
    slidingWindow
30                  RemovePrimeiro(slidingWindow)
31                  AdicionaCaractere(slidingWindow,uncodedLookahead
    [0])
32                  RemovePrimeiro(uncodedLookahead)
33                  AdicionaCaractere(uncodedLookahead, LerLetra(
    arquivo))
34                  restante := restante - 1
35              FIM ENQUANTO
36
37              SE Fim(arquivo) FAÇA
38                  ENQUANTO restante > 0:
39                      RemovePrimeiro(slidingWindow)
40                      RemovePrimeiro(uncodedLookahead)
41                      restante := restante - 1
42                  FIM ENQUANTO
43              FIM SE
44          FIM ENQUANTO
45      FIM ALGORITMO
```

A decodificação do LZSS é possível através da recriação do *slidingWindow* através da leitura sequencial do arquivo comprimido. O programa realiza um laço lendo o próximo bit do arquivo, quando este representa um *bit uncoded*, a letra é adicionada no *slidingWindow* e escrita na saida. Quando o bit que representa o *encoded*, é lido o índice e o tamanho da ocorrência nos próximos 18 bits. O real dado é lido do *slidingWindow* e escrito na saida.

3.2.2.1 Paralelização da aplicação

Para a paralelização da aplicação primeiramente foi medido o tempo de cada etapa do algoritmo. Notou-se que o *hotspot* do algoritmo está na busca da maior ocorrência no *slidingWindow*, ocupando aproximadamente 88% do tempo total da compressão.

Em seguida foi analisado como paralelizar esta etapa do algorítmo. O LZSS da forma apresentado no pseudocódigo 24 trabalha de forma totalmente sequencial. Cada elemento a ser processado, depende da *slidingWindow* atual, não

permitindo uma paralelização eficiente para a GPU. Desta forma, para a paralelização da aplicação foi necessária a alteração da estrutura algorítmica do aplicação a fim de oferecer a possíbilidade de paralelismo de dados. No formato original, o *slidingWindow* e o *uncodedLookahead* são afetados por cada caractere lido, fazendo com que a busca do próximo item no dicionário dependa do resultado da busca anterior.

Notou-se que apesar do *slidingWindow* ser alterado para cada caractere, existia um padrão no formato de busca de maior ocorrência. Cada elemento a realizar a busca, realizava uma busca sequencial nos últimos 4096 elementos por uma ocorrência do elemento atual. A paralelização da busca de ocorrências foi possível através da alteração do código de forma a processar o arquivo em *batches*, ou seja, realizar a leitura de um bloco de tamanho N do arquivo, realizar a busca em todo o bloco, e em seguida escrever o resultado no arquivo de saída.

Como a busca de maior ocorrência pode encontrar uma ocorrência de até 18 caracteres, também são necessários os próximos 18 elementos para realizar a busca. Desta forma, a busca pela maior ocorrência depende de um *batch* formado por 4096 + *batchSize* + 18, para que a busca de todos os itens do *batchSize* seja realizada. O pseudocódigo 25 representa a lógica utilizada para o processamento em *batch* do LZSS.

Código 25: Pseudocódigo de compressor LZSS em *batch*

```
1  ALGORITMO "Codifica LZSS em Batch":
2      VAR:
3          Encoded := 1
4          Uncoded := 0
5          batchSize := 8128
6          buffer := lista(4096 + batchSize + 18)
7          resultadoBuscaBatch := lista(batchSize)
8          atual := 0
9          arquivo,
10         i
11     INICIO:
12
13         ENQUANTO NÃO fim(arquivo) FAÇA
14             SE atual > 0 FAÇA
15                 //Copia o buffer anterior para o começo
```

```
16                  PARA i DE 0 ATÉ 4096 + 18 FAÇA
17                      buffer[i] := buffer[i + batchSize]
18                  FIM PARA
19                  LimpaListaApos(buffer,4096+18)
20              SENÃO FAÇA
21                  PARA i DE 0 ATÉ 4096 + 18 FAÇA
22                      AdicionaCaractere(buffer, ' ')
23                  FIM PARA
24              FIM SE
25
26              PARA i DE 0 ATÉ batchSize FAÇA
27                  AdicionaCaractere(buffer, LerLetra(arquivo))
28              FIM ENQUANTO
29
30              resultadoBuscaBatch := MaiorOcorrenciaBatch(buffer,
    4096, 18)
31
32              atual := atual + 1
33
34              PARA i DE 0 ATÉ batchSize FAÇA
35                  ocorrencia := resultadoBuscaBatch[i]
36                  SE ocorrencia.tamanho > 2 FAÇA
37                      EscreveBit(Encoded)
38                      Escreve(ocorrencia.indice)
39                      Escreve(ocorrencia.tamanho)
40                      //Pula caracteres que já foram escritos
41                      i := i + ocorrencia.tamanho
42                  SENAO FAÇA
43                      EscreveBit(Uncoded)
44                      Escreve(uncodedLookahead[0])
45                      ocorrencia.tamanho := 1
46                  FIM SE
47              FIM PARA
48
49          FIM ENQUANTO
50      FIM ALGORITMO
```

Com o processamento em *batch*, a etapa que leva mais tempo para ser executada pode ser realizada de forma paralela. O psudocódigo 26 representa a função *MaiorOcorrenciaBatch* de forma sequencial, presente na linha 22 do Código 25.

Código 26: Pseudocódigo de compressor LZSS

```
1  ALGORITMO "LZSS Maior ocorrência batch":
2      ENTRADA:
3          batchSize := 8128
4          buffer := lista(4096 + batchSize + 18)
```

```
5           resultadoBuscaBatch := lista(batchSize)
6       VAR:
7           i
8       INICIO:
9
10          PARA i DE 0 ATÉ batchSize FAÇA
11              //Faz a busca da maior ocorrência no buffer da posição atual - 4096 e os próximos 18 itens
12              resultadoBuscaBatch[i] = MaiorOcorrencia(buffer[i - 4096 : i - 1], buffer[i:i+18])
13          FIM PARA
14          RETORNA resultadoBuscaBatch
15      FIM ALGORITMO
```

Como pode ser visto na linha 12 do Código 26, a busca pela maior ocorrência utiliza o *slidingWindow* com um subvetor do *buffer*, iniciando na posição atual menos 4096 até o item atua. O *uncodedLookahead* é representado por um *subvetor* do buffer da posição atual mais 18. Devido a esta característica de acesso aos dados, o *batch* (variável *buffer*) deve ser uma lista com os *4096* bytes do *buffer* anterior, e os 18 do próximo *buffer*.

Devido ao novo formato de busca de maior ocorrência ser uma laço de processamento independente, a execução da função que executa a busca da maior ocorrência pode ser paralelizada. Na implementação GPU cada *thread* fica responsável pelo processamento de um dos itens do *batch*.

Nota-se que no formato sequencial o algoritmo é otimizado de forma a pular algumas buscas desnecessárias conforme o tamanho da ocorrência anterior. Para ser possível o paralelismo da busca, todas os itens do *batch* são utilizados na busca, enquanto a CPU se encarrega de filtrar as buscas úteis. Foram realizados testes procurando utilizar esta otimização em GPU, onde cada *thread* realiza a busca por um número maior de ocorrências, pulando buscas desnecessárias, porém a mesma apresentou uma degradação de desempenho devido a interdependência dos dados.

As implementações foram realizadas utilizando duas interfaces: CUDA e OpenCL. Ambas as bibliotecas foram implementadas seguindo o mesmo algorítmo.

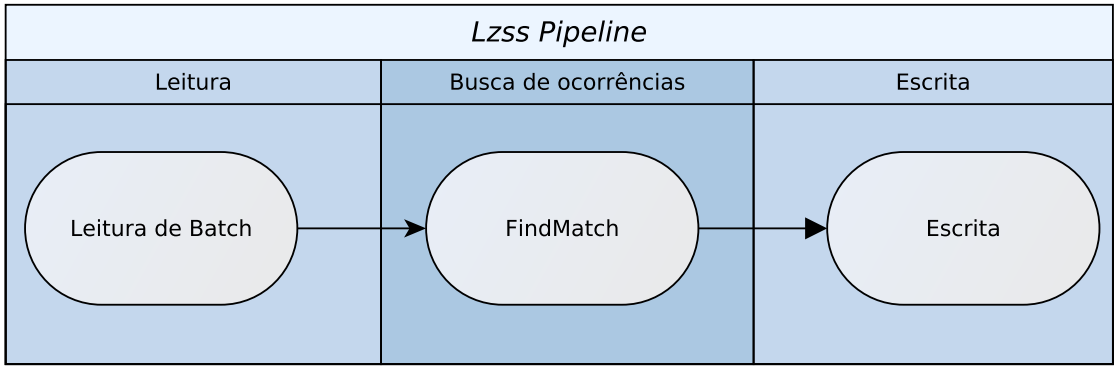
Foram encontradas implementações já existentes em CUDA no trabalho de Ozsoy e Swany (2011), porem a aplicação foi testada e o arquivo resultante da compressão não representava o arquivo original, havendo perdas de dados. Além disso, em sua implementação todo o processo de compressão de dados era executado inteiramente na GPU, enquanto que neste trabalho apenas o paralelismo de dados foi realizado na GPU.

3.2.2.2 Paralelização em CPU

Além do paralelismo de dados existente na busca de maior ocorrência, ainda é possível otimizar a execução do LZSS através do paralelismo em CPU.

O algoritmo paralelizado do LZSS pode ser representado através de uma *pipeline*, onde a execução da leitura, busca de ocorrência e escrita podem ser executadas de forma concorrente. A Figura 35 apresenta o grafo de execução da *pipeline*.

Figura 35: *Pipeline* simples do LZSS



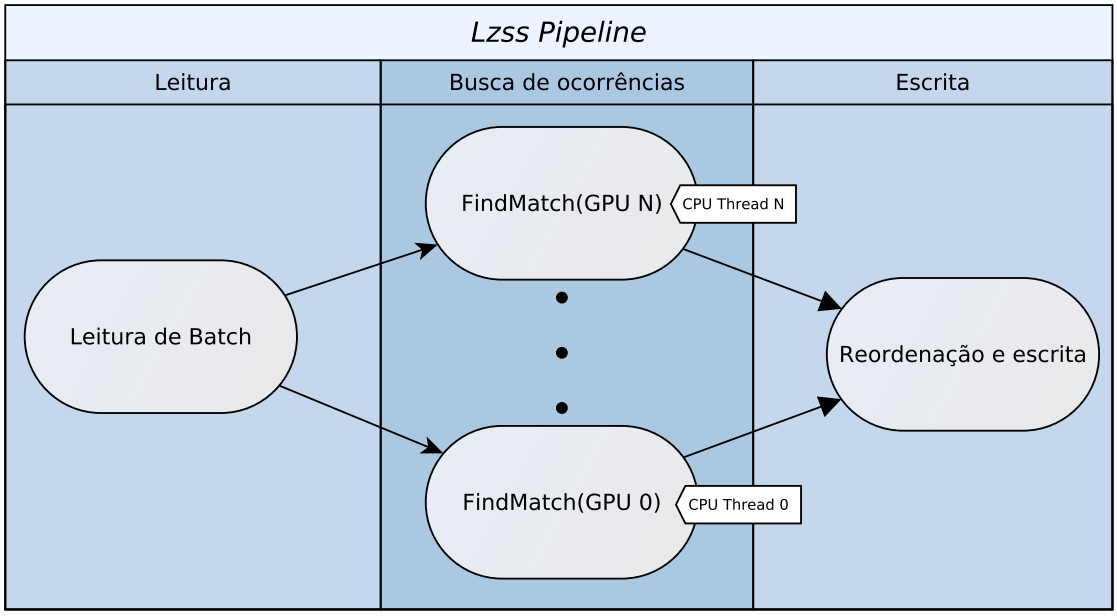
A implementação da *Pipeline* apresentada na Figura 35 foi realizada utilizando a DSL SPar. Através do uso da SPar, o código LZSS foi anotado com os estágios representados anteriormente.

Devido a GPU ser uma aceleradora de aplicações, é comum o uso de múltiplas GPUs a fim de aumentar o desempenho de uma aplicação. Para o uso

de múltiplas GPUs o estágio de busca de ocorrências foi paralelizado a fim de fazer o uso completo das GPUs.

Para cada GPU presente no computador, um *thread* da CPU fica responsável pelo controle da execução da GPU. Devido a execução concorrente de múltiplas instâncias do estágio *FindMatch* a escrita do arquivo precisa reordenar os dados conforme a leitura realizada. Com o uso da SPar isso pode ser obtido através do argumento *-spar_ordered* na compilação, sem a necessidade de alterações no código. A Figura 36 apresenta a *pipeline* utilizada par auso de múltiplas GPUs.

Figura 36: *Pipeline* de múltiplas GPUs do LZSS



3.2.2.3 Análise dos resultados

A partir das implementações realizadas do LZSS, foi realizado um *benchmark* onde se buscou analisar o ganho de desempenho obtido. Visto que esta aplicação é utilizada em compressão de arquivos, os testes foram realizados com arquivos reais que representam o real uso destas aplicações.

O primeiro *dataset* utilizado é o Silesia Corpus. Conforme Deorowicz (2018), este é composto por um conjunto de dados que representam dados de diversas

áreas, como bancos de dados, arquivos multimídia e códigos fonte. O Linux *tarbal* representa o uso de compressão em códigos fonte, sendo composto por todo o código do *kernel* do Linux. Por último, o arquivo *CreadtBackup80* representa um *backup* de banco de dados relacional. Detalhes destes *datasets* são apresentados no Quadro 4.

Quadro 4: Resumo do *dataset* do *benchmark*

Dataset	**Entrada (MB)**	**Saida (MB)**	**Compressão (%)**
silesia.tar	202.13	90.07	44.56
linux-4.16-rc4.tar	797.57	249.34	31.26
CreditBackup80.bak	155.08	75.48	48.67

O *benchmark* foi realizado com a máquina com configurações apresentada na seção 3.2.1. Visando uma melhor qualidade dos resultados, cada teste foi realizado tanto com o tempo de Leitura e escrita do arquivo, quanto em memória. Cada teste foi repetido 5 vezes. Os códigos foram compilados utilizando a opção -O3, que permite o compilador realizar todas otimizações disponíveis.

Na Figura 37 é apresentada a comparação do tempo total nas implementações usando GPU com a versão original em CPU. Nota-se que nesta aplicação o tempo de I/O apresenta pouca diferença no tempo total quando comparado ao filtro sobel, incrementando aproximadamente em 1% do tempo total em CPU e 13% em GPU. A leitura da versão em GPU teve menos impacto de I/O devido a leitura em *batch*, enquanto a versão em CPU efetua a leitura de um item e logo em seguida realiza o processamento, precisando assim fazer mais chamadas ao sistema para a leitura.

Através da Figura 38 é possível analisar o tempo da operação de buscar a maior ocorrência no texto. É possível notar que esta operação ocupa aproximadamente 90% do tempo total na versão em CPU, enquanto que em GPU a operação corresponde a aproximadamente 98% do tempo total. Isto acontece devido versão sequencial ter um impacto maior do I/O, como apresentado anteriormente.

Figura 37: Tempo total de compressão

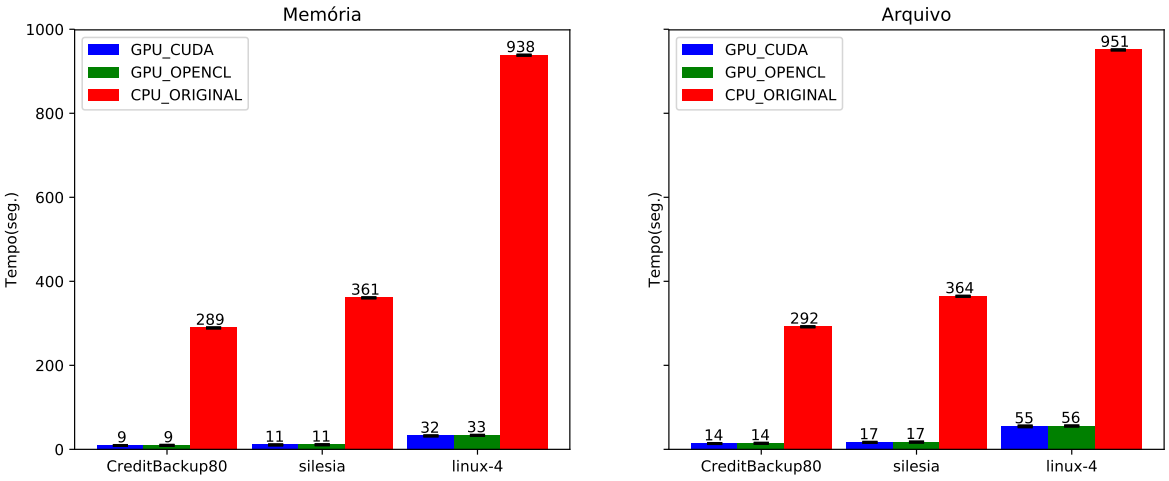
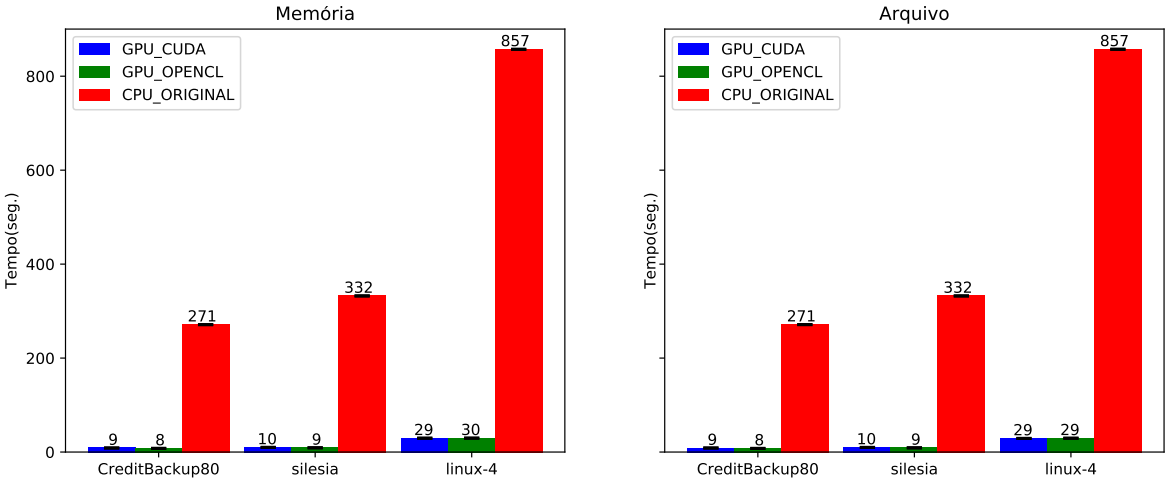


Figura 38: Tempo de busca de maior ocorrência

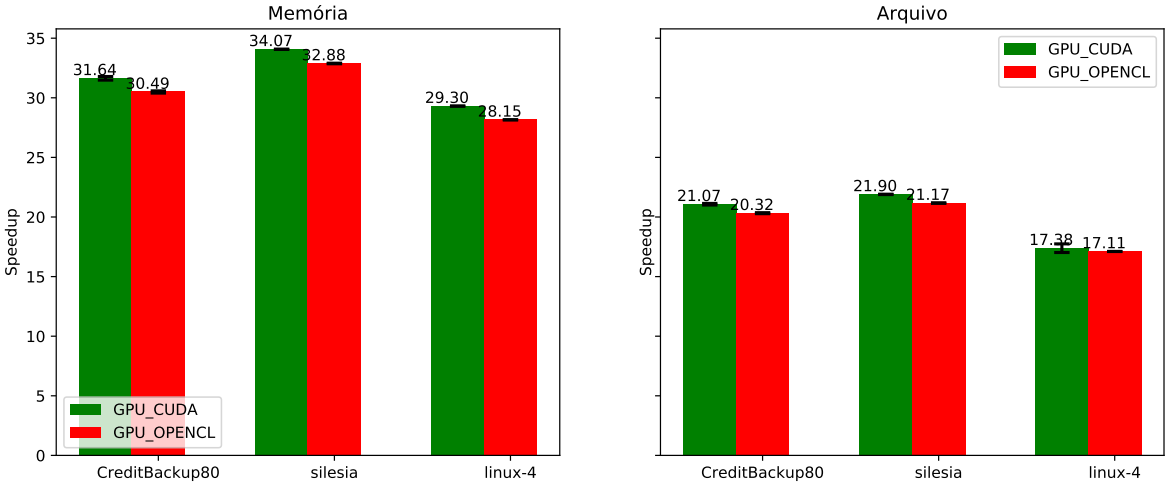


O ganho de desempenho obtido na paralelização da operação de busca de maior ocorrência, mesmo realizando buscas que em CPU não seriam necessárias, conseguiu atingir um *speedup* entre 4.5 e 6.5. A grande capacidade de execução paralela da GPU faz com que executar todas as buscas se torne mais eficiente do que sincronizar as buscas a fim de evitar buscas desnecessárias. Na Figura 39 é possível visualizar o *speedup* detalhado para cada *dataset*.

Nos testes em que a busca de maior ocorrência era realizada em blocos maiores, ignorando buscas desnecessárias o desempenho foi aproximadamente 50% pior que a versão em GPU atual. Esta perda de desempenho se da devido a arquitetura GPU favorecer a execução onde existe um balanceamento entre as

tarefa em cada *thread*, ao implementar possíveis atalhos nos *threads*, o algoritmo se torna menos determinístico, diminuindo o paralelismo dos *warps* na GPU.

Figura 39: *Speedup do LZSS em GPU*



Um grande limitador no desempenho do filtro Sobel foi o tempo de transferência CPU-GPU, desta forma no LZSS foi analisado quanto tempo a transferência dos dados representa. No Quadro 5 é apresentado o tempo de cópia do *Host* para o *Device*, execução do *kernel* e a cópia *Device* para *Host*.

Quadro 5: Comparação de tempo de busca de maior ocorrência em GPU

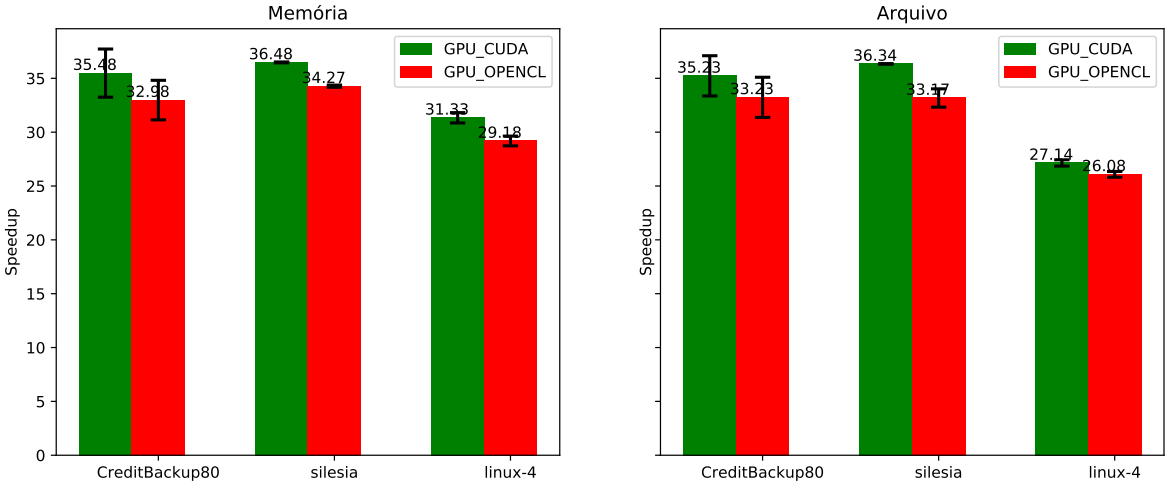
Dataset	Biblioteca	HostDevice	Kernel	DeviceHost	Total
CreditBackup80.BAK	CUDA	0.14	7.23	0.03	8.59
CreditBackup80.BAK	OPENCL	0.11	7.53	0.03	7.72
linux-4.16-rc4.tar	CUDA	0.66	27.17	0.17	29.20
linux-4.16-rc4.tar	OPENCL	0.51	28.55	0.14	29.46
silesia.tar	CUDA	0.18	8.44	0.04	9.84
silesia.tar	OPENCL	0.14	8.78	0.04	9.02

Pode se analisar que no LZSS é realizada a transferência do *batch* atual para a GPU, e este é reaproveitado em todos os *threads*, fazendo com que o *kernel* represente a maior parte do tempo da operação de busca de maior ocorrência. Devido a esta característica de reuso de dados, a transferência de dados para esta

aplicação não apresenta grande *overhead*. Além de possuir blocos reutilizáveis de dados, a busca realizada sobre o texto possui uma alta complexidade, fazendo com que a paralelização em GPU represente um alto ganho de desempenho.

Como foi visto, foi realizada uma implementação que faz o uso de múltiplas GPUs através de uma *Pipeline* implementada em SPar. Na Figura 40 é possível ver o *speedup* obtido nas versões com *SPar*, com o uso de apenas uma GPU.

Figura 40: *Speedup do LZSS por biblioteca*

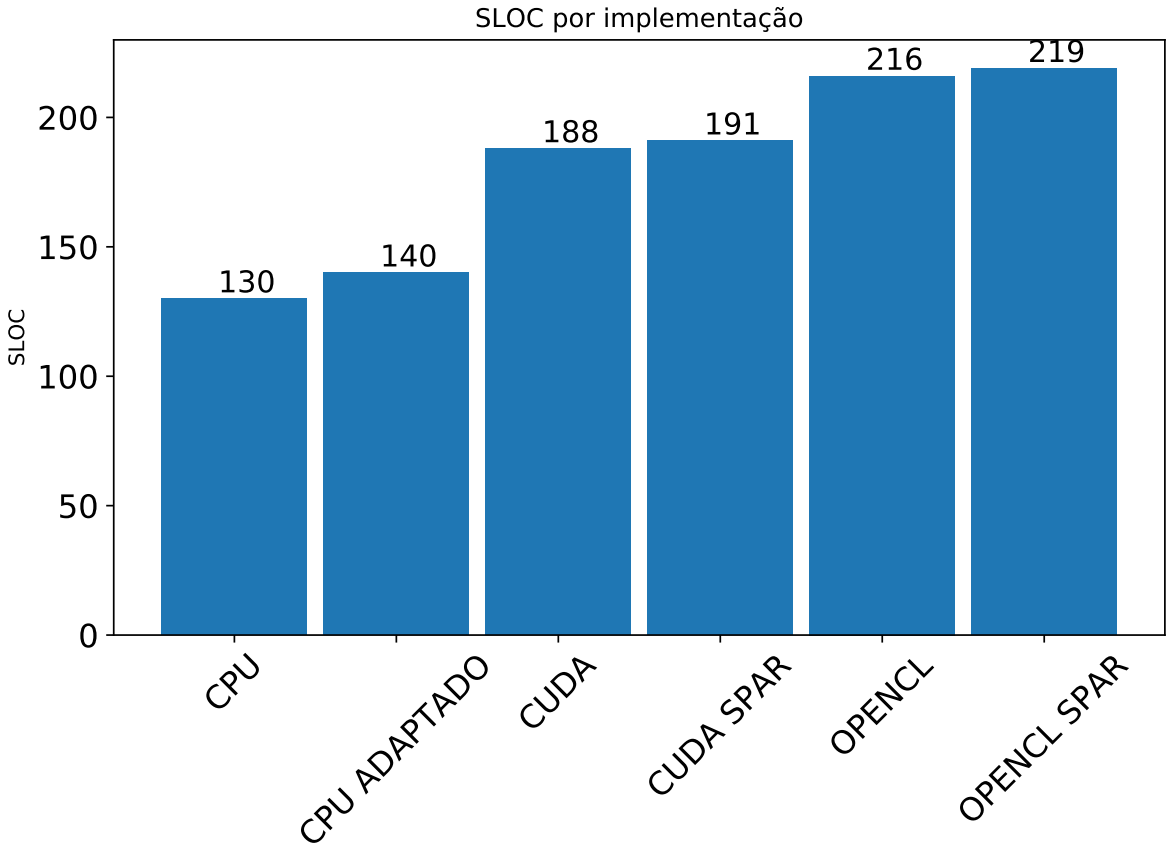


Nota-se que através do uso da *pipeline*, o uso da GPU foi feito mais eficiente, já que as versões com interferência do I/O do arquivo obtiveram quase o mesmo *speedup* das em memória. Com a *pipeline* em CPU a GPU não permanece inoperante quando a CPU está realizando leitura ou escrita. As diferenças entre CUDA e OpenCL permanecem com semelhante as versões sem paralelismo de CPU.

Para analisar a intrusão de código obtida com cada biblioteca, foram comparadas as linhas de código fonte em cada biblioteca. Já que foi necessária a alteração da estrutura algorítmica para a paralelização em GPU, foi adicionada a análise a versão sequencial da processamento em *batch*. Na Figura 41 é possível visualizar a quantidade de linhas de código de cada biblioteca.

Nota-se que a versão sequencial exigiu 10 linhas a mais que a versão

Figura 41: Linhas de código por implementação



sequencial original. Para a implementação em CUDA foi necessário um aumento de 34% no código fonte. Como o Cuda possui um compilador próprio, a intrusão de sua implementação é apenas definida pelas transferências de dados e a criação do *Kernel*, sendo que a transferência de dados ainda pode ser simplificada se utilizado o recurso de *Unified Memory*, apresentado na seção 2.4.3.1. O OpenCL exigiu 28 linhas a mais que a implementação em CUDA. Grande parte das linhas de código que foram incrementadas com a versão em OpenCL se da devido a etapa de compilação e criação de contexto que faz-se necessário no OpenCL, devido as suas características multi-plataforma.

Para as implementações com paralelismo de GPU e CPU, como foi utilizada a DSL SPar, foi necessário apenas adicionar as anotações necessárias para a criação da *pipeline*, gerando um incremento de 3 linhas de código a partir da versão de GPU.

3.2.3 Dedup

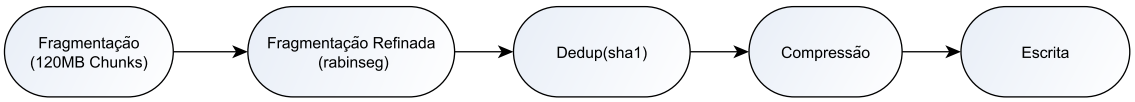
O *Dedup* é outra aplicação de processamento *stream* utilizado para compressão de dados. Está é uma aplicação utilizada em vários tipos de aplicações e está presente na ferramenta de *benchmark* PARSEC 3. Devido a sua capacidade de economizar espaço onde arquivos ou blocos repetidos, pode ser utilizado tanto para salvar espaço em disco quanto para evitar transferência de dados duplicados entre servidores. Conforme Griebler et al. (2018) o *Dedup* comprime a *stream* de dados a níveis globais e locais. A deduplicação em seu formato sequencial é separada em 4 estágios:

- Framentação: Neste estágio a *stream* de dados é framentada em blocos de tamanho fixo de 128MB, para que este bloco seja processado.
- Framentação Refinada: O bloco de dados são segregados em um nível mais granular, utilizando o algoritmo de *Rabin Fingerprint*. Este estágio gera os blocos de tamanhos variáveis para que sejam encontrados blocos repetidos.
- Deduplicação: São realizadas as buscas pelos blocos refinados baseado em um *hashtable* do histórico de blocos.
- Compressão: Os novos blocos (ainda não presentes no *hashtable*) são comprimidos.
- Escrita: Nesta etapa o bloco é escrito no arquivo de saida. Para blocos deduplicados, apenas o *sha1* é escrito, por outro lado nos blocos não deduplicados o dado comprimido é escrito.

Esta aplicação utilizou como base de código fonte a implementação do *dedup* presente no PARSEC 3. Através da Figura 42 é possível ver todas as etapas do algoritmo sequencial do *dedup* em CPU.

O algoritmo de *Rabin Fingerprint* é utilizado para a segmentação dos dados, cada bloco gerado pode ser do tamanho de 32 bytes até o tamanho total do

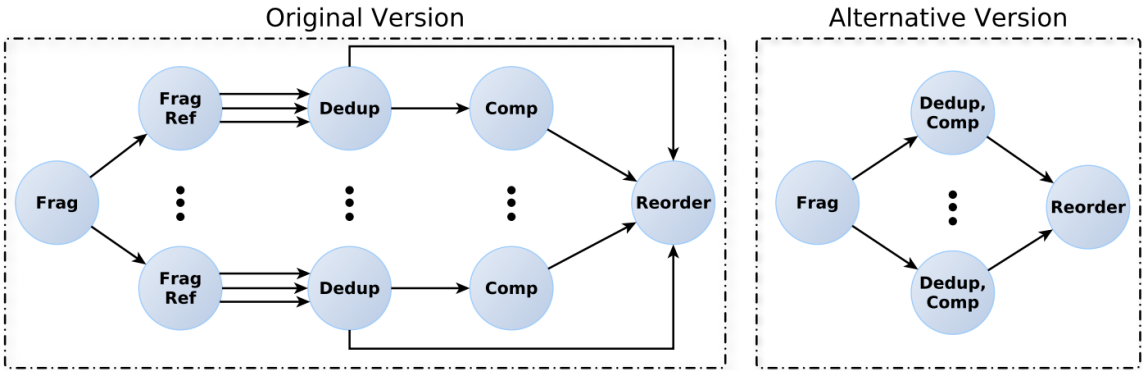
Figura 42: *Dedup* Sequencial



fragmento. Nota-se que para o estágio de deduplicação é utilizado um *hashtable* para a busca de blocos duplicados, sendo necessário o cálculo do *hash sha1* para cada bloco refinado, gerado no segundo estágio.

Uma implementação do *Dedup* com base no PARSEC foi realizada por Griebler et al. (2018) utilizando o SPar. Em sua versão, a *pipeline* do Dedup foi alterada, onde o paralelismo de tarefas era gerado a partir do *Rabin Fingerprint*. Cada bloco refinado era processado por um *thread*, realizando a deduplicação e compressão, conforme apresentado na Figura 43, onde é comparada a implementação em *pthreads* original do PARSEC com a versão alternativa em *SPar*.

Figura 43: *Dedup* com SPar



Neste estudo, para fins de comparação, a versão em SPar foi utilizada como base de implementação paralela em CPU, já que de acordo com Griebler et al. (2018) o desempenho foi semelhante a versão em *pthreads*, sendo até mais rápida em alguns casos.

O algoritmo *sha1* representa uma aplicação de difícil paralelização, desta forma para a execução em GPU a estratégia de paralelismo teve que ser adaptada, a fim de cada *thread* da GPU realizar o *hash* de um bloco. Conforme o estudo de

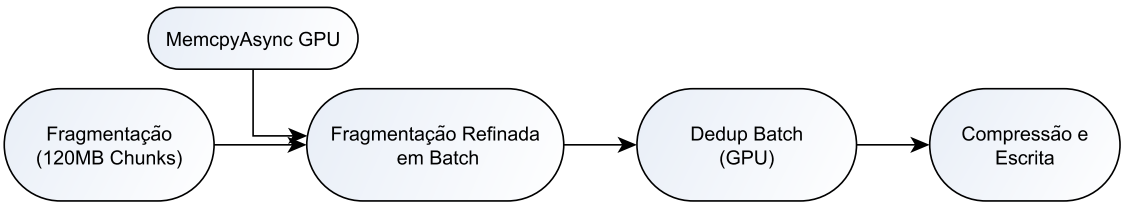
Deshpande (2014), o algoritmo de *hash* no *dedup* pode ser executado em blocos maiores na GPU, onde cada *thread* fica responsável por calcular o *hash* de uma parte daquele bloco. Nesta estratégia de paralelismo, o ganho de desempenho acontece aumentando o *throughput* (*hashes* por segundo), utilizar o algoritmo de *sha1*.

Para que esta implementação fosse possível, a estratégia de paralelismo precisou ser diferente da versão em CPU. Na versão em CPU o paralelismo de dados é dado através da distribuição de tarefas da fragmentação refinada. Na GPU este formato não permite o uso eficiente dos recursos da GPU. Busca-se maximizar a sua utilização através do processamento de blocos maiores, ou seja, para um bloco, realizar o maior número de operações para aquele bloco de forma paralela. Isso é possível através do paralelismo a nível de fragmentos maiores, desta forma o paralelismo é dado pelos blocos de 128MB gerados no primeiro estágio da *pipeline*.

Um ponto interessante do *sha1* que precisa ser gerado é que os blocos possuem um desbalanceamento, já que o tamanho dos blocos do estágio dois variam. Para a GPU este desbalançamento faz com que além do bloco inteiro ser transferido para a GPU, os tamanhos e *offsets* gerados pelo escalonador de *rabin fingerprints* também sejam transferidos. Como a carga de trabalho varia para cada *thread* na GPU, o desempenho pode ser reduzido devido a menor eficiência no uso do *warp* da GPU.

Como pode ser visto na *pipeline* da Figura 42, a segmentação dos blocos é feita em dois níveis, os blocos de 128MB e os blocos do *Rabin*. Como os blocos *rabin* representam apenas índices e *offsets* do bloco de 128MB, a partir do fragmentação inicial os dados, o bloco de 128MB já podem ser transferido de forma assíncrona para a GPU, maximizando assim o uso da GPU e CPU. Enquanto os índices do *rabin* são calculados para todo o bloco, os dados já estão sendo transferidos. Desta forma quando o bloco estiver pronto para o *sha1*, os dados já estão disponíveis em GPU. A Figura 44 apresenta todas as etapas da execução da GPU.

Figura 44: *Dedup* Sequencial com GPU



Como apresentado na Figura 44, diferente das operações em CPU que são executadas de forma sequencial, para a execução na GPU todas as operações até o estágio de escrita do arquivo são realizadas em *batch*, ou seja, ao invés da *pipeline* ser executada para cada bloco do segundo estágio, as operações são acumulados para que a GPU possa executar em paralelo para todos os itens do blocos.

Após a etapa de deduplicação, os blocos não deduplicados (ainda não presentes no *hashtable*) podem ser comprimidos. A versão original oferecia suporte a compressão com BZIP2 e GZIP, porém como o trabalho já realizou a implementação do *LZSS* em GPU, foi implementado o suporte do LZSS ao *dedup* tanto para a versão sequencial quanto na em GPU. A GPU se beneficia do LZSS devido aos dados necessários para a compressão já estarem presentes no dispositivo, evitando transferências de pequenos blocos que causariam um *overhead*.

Além do paralelismo da GPU, ainda existe a possibilidade de aumentar o desempenho da aplicação através do uso de paralelismo em CPU. Foi implementado o paralelismo de CPU utilizando a DSL SPar. Com ela foi construída uma *pipeline*, como a presente na Figura 44, para que o uso da GPU fosse o mais próximo de 100%.

3.2.3.1 Análise de resultados

Após a implementação do *dedup*, conforme descrita na seção anterior, foram realizados testes de desempenho onde buscou-se encontrar o ganho obtido

com o uso de paralelismo.

Neste *benchmark*, todas as versões de *dedup* descritas na seção 3.2.3 foram executadas 5 vezes. O *dataset* utilizado para este teste foi o *input_native*, presente nos arquivos de *benchmark* do PARSEC 3.0. Para fins de análise de desempenho para cada estágio realizado na GPU, os testes foram realizados sem a compressão de arquivos e com. O *hardware* utilizado foi o mesmo descrito na seção 3.2.1.1.

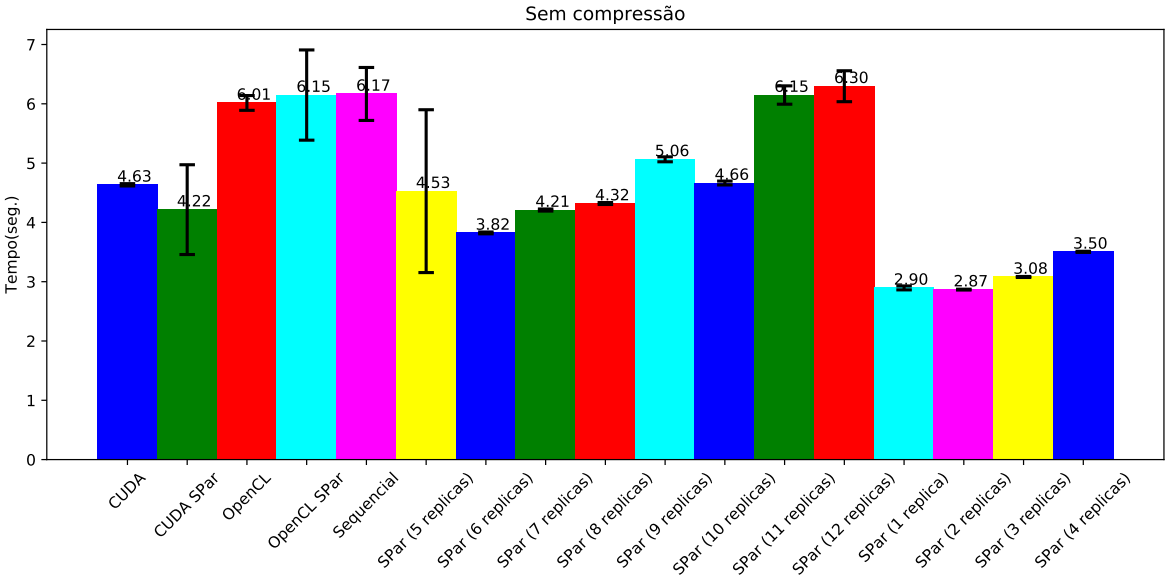
Primeiramente foi analisado o ganho de desempenho da execução do algoritmo de *hash* na GPU. Para isso, foi executado o teste sem a compressão de arquivos, fazendo o *hotspot* da aplicação ser o algoritmo de *hash*, e não a compressão. Nesta etapa, foram testadas a aplicação sequencial, as versões utilizando apenas GPU, versões com GPU e *pipeline*, bem como as versões que utilizam apenas o paralelismo de CPU. As versões que utilizam apenas *SPar* tem o seu paralelismo alterado conforme o número de *threads* executando, por isso estes foram executados com diferentes quantidades de *threads*.

Na Figura 45 é possível perceber que as versões em GPU possuem um desempenho superior a versão sequencial. O uso da GPU no estágio de *sha1* obtem melhora de desempenho por ser um estágio que possuem grande paralelismo de dados. O CUDA acaba tendo um desempenho maior que o OpenCL devido ao tempo de compilação *Just in time* presente na biblioteca. Nota-se que as versões de GPU que utilizaram o paralelismo de CPU não obtiveram ganhos quando comparado as versões utilizando apenas GPU, devido ao estágio mais lento da *pipeline* de execução ser o processo de geração de *hash*, não sendo assim muito afetado pelo I/O da aplicação.

Em relação as versões de CPU (*spar*), nota-se que o desempenho piora conforme aumentam-se o número de *threads* executando. Isso se dá devido a implementação realizar a compressão e deduplicação em paralelo, com isso o espaço de memória que mantém um *hashtable* dos blocos já deduplicados é compartilhado, e seu acesso deve ser controlado por um *mutex*. Devido a neste teste

não se estar utilizando compressão, cada *thread* acaba apenas realizando o *hash* para um bloco pequento. Este é um processo rápido para a CPU executar, fazendo assim com que todos os *threads* tentam acessar o *hashtable* ao mesmo tempo, sendo travados pelo *mutex* e aumentando o tempo de execução.

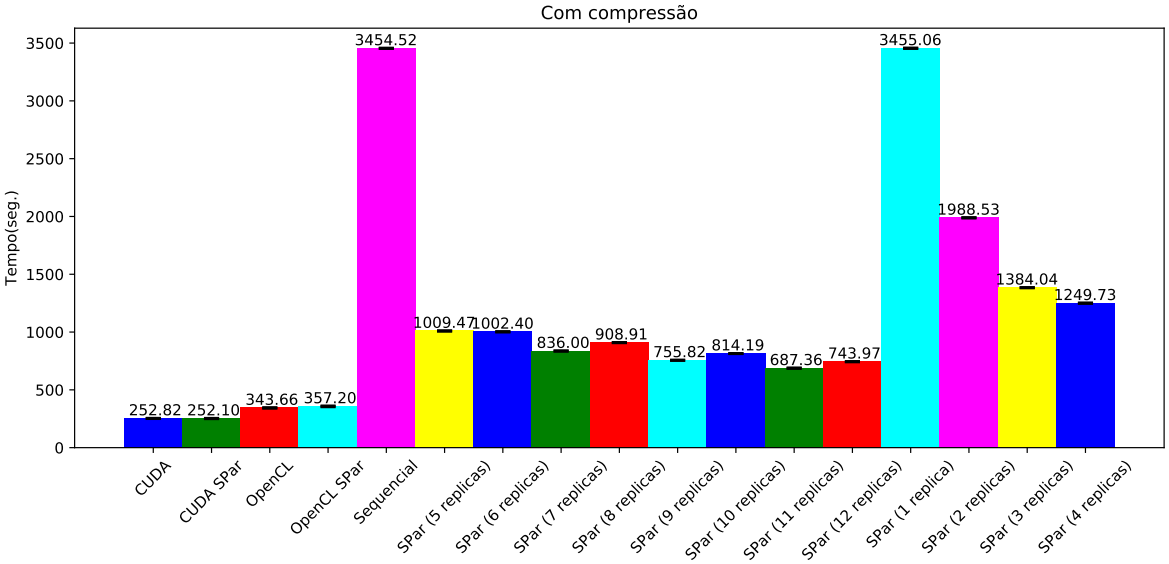
Figura 45: Tempo total do *Dedup* sem compressão



Quando o teste é executado com a compressão de dados, pode ser visto na Figura 46 que o comportamento muda. As versões em CUDA ainda são mais rápidas que OpenCL, porém a disparidade de tempo aumenta. Esta grande disparidade não era esperada, o grupo acredita que devido a grande quantidade de *kernels* sendo executada pela GPU se torna mais lento em OpenCL. Em relação as implementações CPU-GPU, não há ganhos significativos para esta aplicação. A implementação em *SPar* agora possui outro comportamento, devido a maior carga de trabalho o desempenho melhora conforme o número de *threads* aumenta.

Em relação ao *speedup* obtido, na Figura 47 é apresentado o *speedup* de cada versão sem compressão. Nota-se que as versões em CUDA foram 33% mais rápido que a versão sequencial, porém o OpenCL não obteve melhora de desempenho deviso ao tempo de compilação. Apesar das versões em GPU terem sido mais rápidas que as versões sequenciais, nota-se que as versões em *SPar* foram mais rápidas em todos casos.

Figura 46: Tempo total do *Dedup* com compressão



Nas execuções que utilizam compressão, apresentadas na Figura 48, pode ser visto que as versões em CUDA obtiveram o melhor desempenho, superando até mesmo as versões em SPar. Este ganho de desempenho se dá devido a compressão LZSS estar implementada em GPU, onde o *speedup* já foi apresentado na seção 3.2.2.3. Em relação a versão em *SPar* nota-se que devido a carga de trabalho ser maior, o desempenho melhora conforme o número de *threads* aumenta. Mesmo assim, com a compressão as versões em GPU foram mais rápidas.

As versões paralelizadas para CPU, obtiveram um comportamento semelhante ao apontado por Griebler et al. (2018), onde a mesma foi paralelizada utilizando a DSL *SPar*. Em sua implementação, não foi utilizado o LZSS, mas sim o BZIP2, fazendo com que o desempenho fosse diferente do deste presente trabalho. Além disso, devido a um *hardware* e compressões diferentes do utilizado nestes testes, o *speedup* obtido foi diferente. Enquanto que neste trabalho, através do uso do LZSS se obteve o melhor desempenho em CPU de 5.03x, no trabalho de Griebler et al. (2018) o *speedup* em SPar foi de 6.59x.

Em relação a intrusão de código de cada biblioteca, foram contabilizadas as linhas de código a fim de representar o quando cada biblioteca interfere no código final. Como o *dedup* apresenta etapas de deduplicação e compressão,

Figura 47: *Speedup* do *Dedup* sem compressão

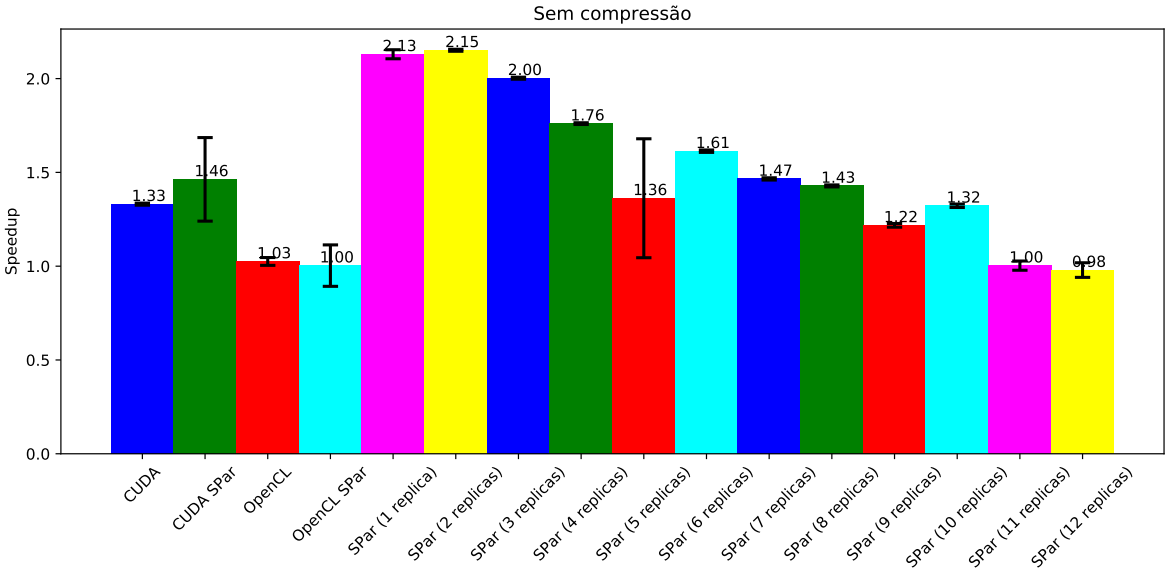
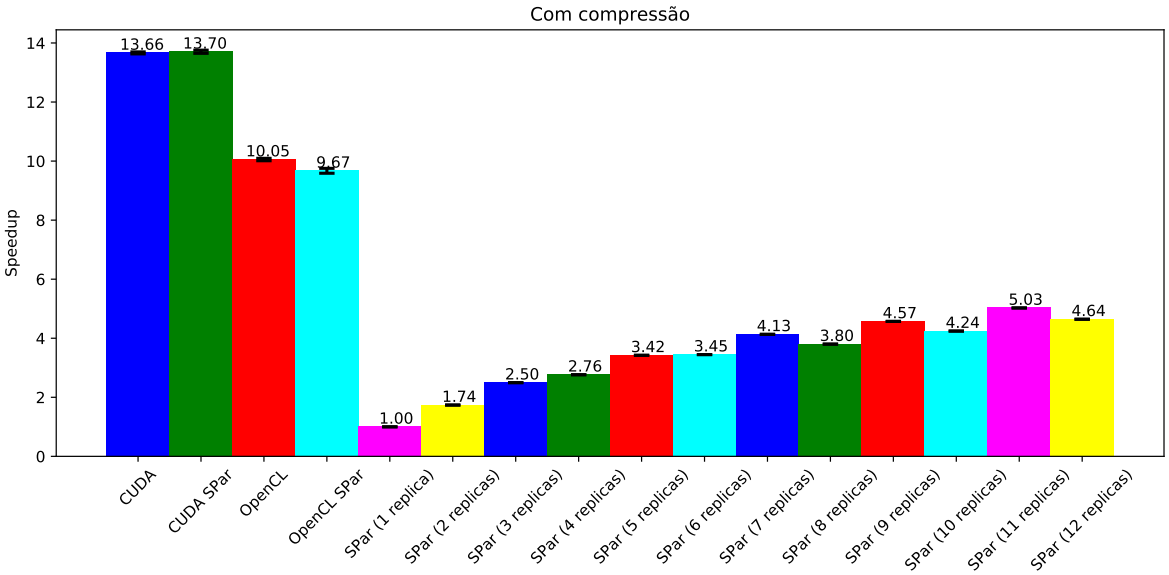


Figura 48: *Speedup* do *Dedup* com compressão



neste experimento foram medidas apenas as linhas de códigos necessárias para a deduplicação, já que as linhas de código do LZSS já foram apresentadas na seção 3.2.2.3.

Na Figura 49 são apresentadas as linhas de código por implementação. Nota-se que o código em SPar apresentou um aumento nas linhas de código, já que o código sequencial teve que ser alterado para ter suporte a paralelismo. As versões CUDA representaram um aumento nas linhas de código devido a necessidade de criação do método *sha1* para a GPU, já que não é possível utilizar a implementação do C++.

Com o OpenCL, alguns recursos do C não são suportados pelo *kernel* do OpenCL. Desta forma, além das linhas de código que representam a compilação do *kernel*, fez-se necessária uma implementação diferente do *sha1*.

Em relação a adição da SPar nas implementações para a GPU foram necessárias apenas as linhas de código que anotam cada estágio.

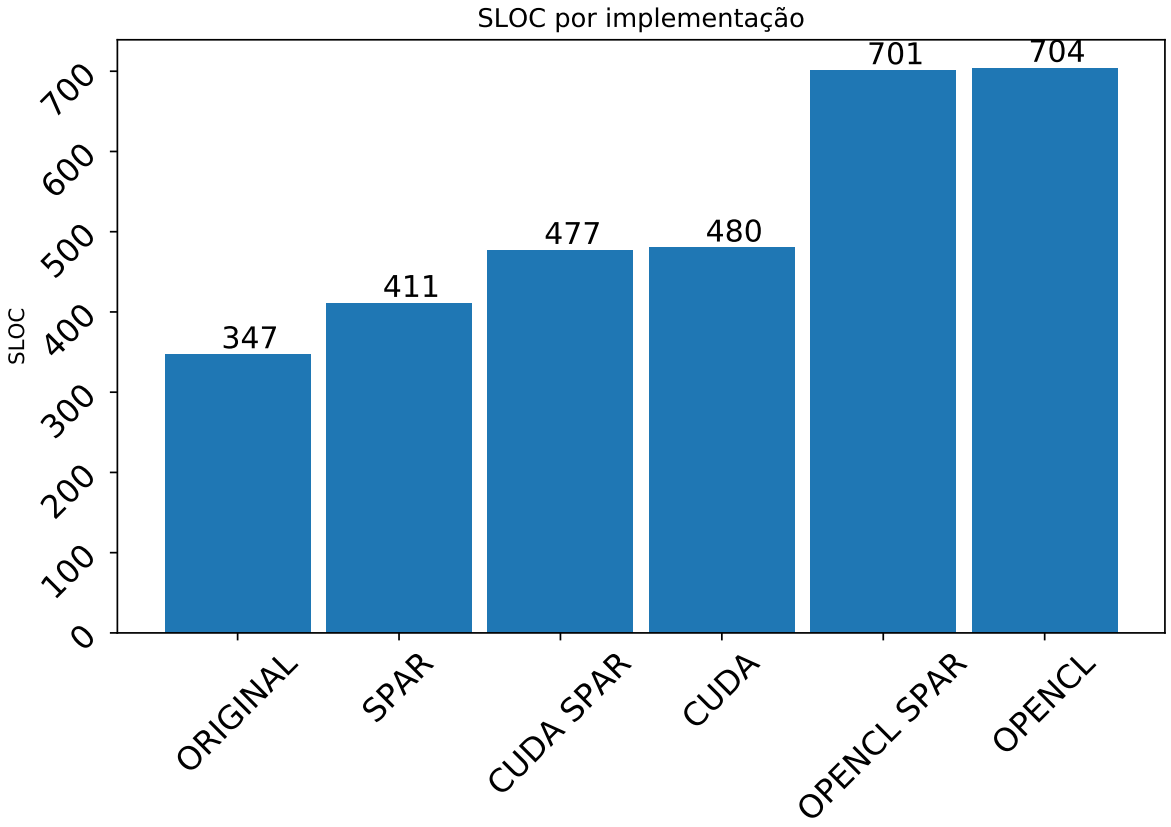
3.2.4 Ferret

Conforme Bienia et al. (2008), o Ferret é uma aplicação de busca de similaridades em imagens. A aplicação foi desenvolvida pela Universidade de Princeton.

Esta aplicação representa a última geração de busca em conteúdos, e por isso está presente no PARSEC *Benchmark*. Como base de código fonte, foi utilizado o código disponibilizado pelo PARSEC.

A aplicação *Ferret* utiliza um conjunto de imagens como banco de busca de dados, e realiza uma operação comparando cada uma destas imagens. A operação realizada para buscar similaridades no *Ferret* é conhecida como *Earth mover's distance* - EMD. Conforme Rubner, Tomasi e Guibas (1998), o *earth mover's distance* representa o menor trabalho possível para mover duas distribuições. Rubner, Tomasi e Guibas (1998) apresenta que o EMD pode ser utilizado como uma métrica

Figura 49: Linhas de código por implementação do dedup



para diferenciar imagens.

Conforme testes executados na aplicação, a etapa de EMD representa 84% do tempo total do *Ferret*. Desta forma, foi realizada uma tentativa de paralelizar esta etapa. Ao analisar o código, percebeu-se que o EMD consiste de laços não determinísticos. O EMD consiste de um laço principal, onde dentro dele dois laços executam, cada um alterando os dados do próximo laço.

Isso impediu a paralelização desta aplicação, já que nas outras partes do código não seria possível obter um *speedup* significativo para a aplicação.

3.2.5 Black-scholes

Aplicações de bolsa de valores são outro exemplo de aplicação de processamento de *stream*. Além de muitas vezes trabalhar com uma grande quan-

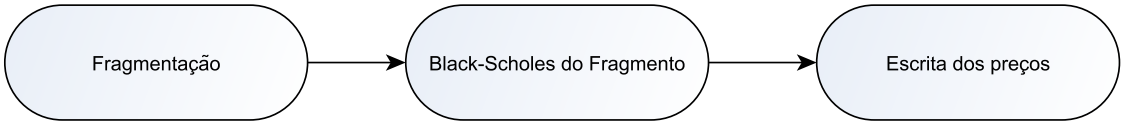
tidade de dados, o desempenho neste tipo de aplicação é crítico. O mínimo de aumento de desempenho de uma aplicação deste tipo pode representar milhões em dinheiro. Nesta área, algoritmos mais rápidos podem gerar mais lucro.

Desta forma, neste estudo foi utilizada uma aplicação para o mercado de opções, chamada *Black-Scholes*. Segundo Christian Bienia Sanjeev Kumar e Li† (2008), o *Black-scholes* calcula o preço de um portfólio de opções Europeias, através da operação de *Black-scholes partial differential equation* (PDE). O código fonte de base do *Black-scholes* foi retirado da aplicação PARSEC, porém naquela implementação o comportamento da aplicação não se caracterizava como aplicação de processamento de *stream*. Desta forma, a aplicação foi alterada para fazer uma leitura continua de dados a partir de um arquivo de entrada. Uma certa quantidade de elementos é lida, processada e salva em um arquivo de saída.

O paralelismo nesta aplicação é feito através de um *map* (seção 2.4.1.3). Cada item resulta em apenas uma saída, podendo ser paralelizado para a GPU de forma simples, onde cada *thread* processa um elemento. Para este paralelismo ser possível, foi criada uma *pipeline* que processa *batches* de dados (aproximadamente cem mil por vez), pois dessa forma o *overhead* de mover os dados para a memória da GPU não impacta tanto no desempenho.

Além do paralelismo de GPU, o paralelismo na CPU foi implementado utilizando a DSL SPar. Com a utilização desta DSL, foi criada uma *pipeline* com o objetivo de maximizar o uso da GPU. A *pipeline* implementada contém três estágio, sendo estes a leitura do arquivo, o cálculo do *Black-scholes* para o conjunto, e a escrita do resultado. Esta *pipeline* pode ser visualizada na Figura 50.

Figura 50: *Pipeline* do *Black-Scholes*



Além da *pipeline* em GPU, a segunda etapa também foi implementada uti-

lizando apenas a CPU para fins de comparação.

3.2.5.1 Análise dos resultados

A partir das implementações do *Black-scholes* apresentadas, foi realizado um *benchmark* para analisar o ganho de desempenho obtido. O teste foi realizado com o mesmo *hardware* apresentado na seção 3.2.1.1. Foi utilizado o *dataset input_native* do *Black-scholes*, presente no PARSEC *benchmark*.

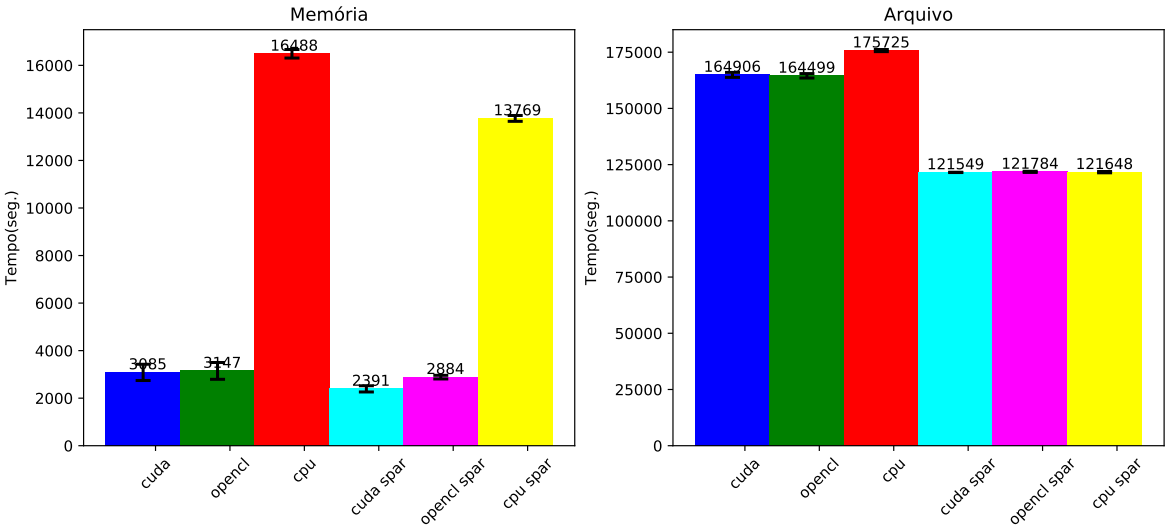
Como a implementação do modo de execução da aplicação foi alterada para se comportar como uma aplicação de processamento de *stream*, o *dataset* original foi aumentado em 10 vezes, já que a versão original não representava uma carga que leva-se tempo suficiente para comparações. Os testes com o *dataset* foram repetidos 5 vezes para cada implementação, tanto trabalhando com a leitura de arquivo, quanto trabalhando com o arquivo em memória para comparar o tempo sem I/O.

Na Figura 51, é apresentado o tempo total das implementações. Nota-se que em todas versões em que a leitura dependeu do tempo de IO, o tempo total foi degradado devido ao estágio mais lento da *pipeline* acabar sendo a leitura e escrita de arquivos. Este fenômeno é notado também nas versões onde a *pipeline* foi implementada, o tempo de leitura e escrita acaba sendo o estágio mais lento da *pipeline*, prejudicando o tempo total.

Nas versões em memória fica mais claro o desempenho obtido em cada versão. Sem a interferência de I/O nota-se que as versões SPar (com *pipeline*) não apresentam ganhos comparados as versões sem. Além disso, é possível notar uma pequena diferença entre o OpenCL e o CUDA devido ao tempo de compilação presente no OpenCL.

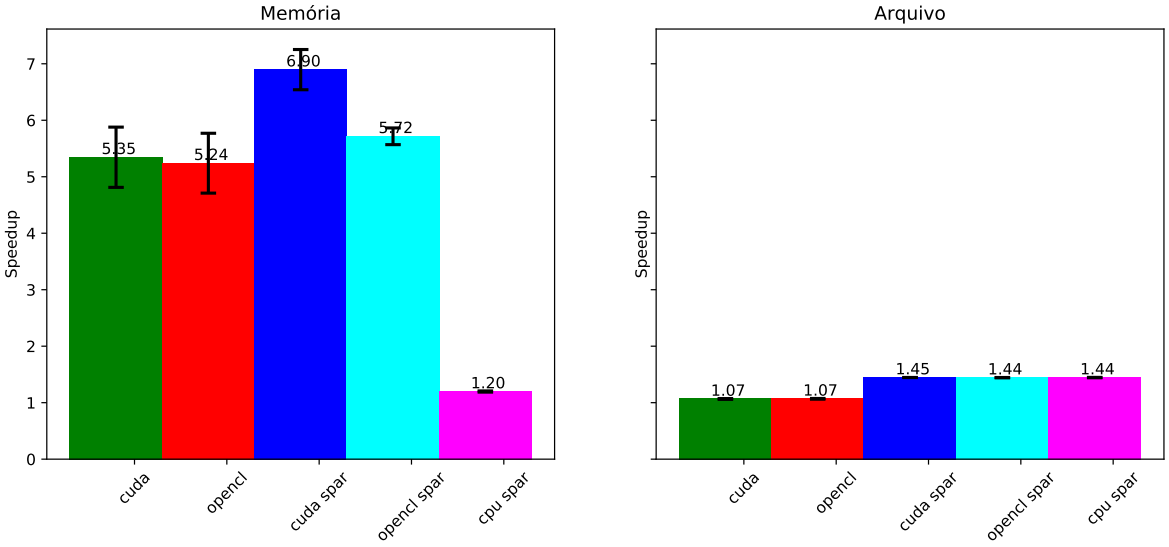
A Figura 52 mostra o *speedup* obtido com a paralelização da aplicação. Como já comentado, pode ser visto que nesta aplicação o I/O representa uma grande quantidade do tempo, limitando o *speedup* das versões em arquivo, che-

Figura 51: Tempo total por implementação do *Black-Scholes*



gando a um máximo de 1.44x de *speedup*. Por outro lado, nas versões executadas em memória nota-se que o maior *speedup* foi de 6.9x através do paralelismo de GPU e CPU.

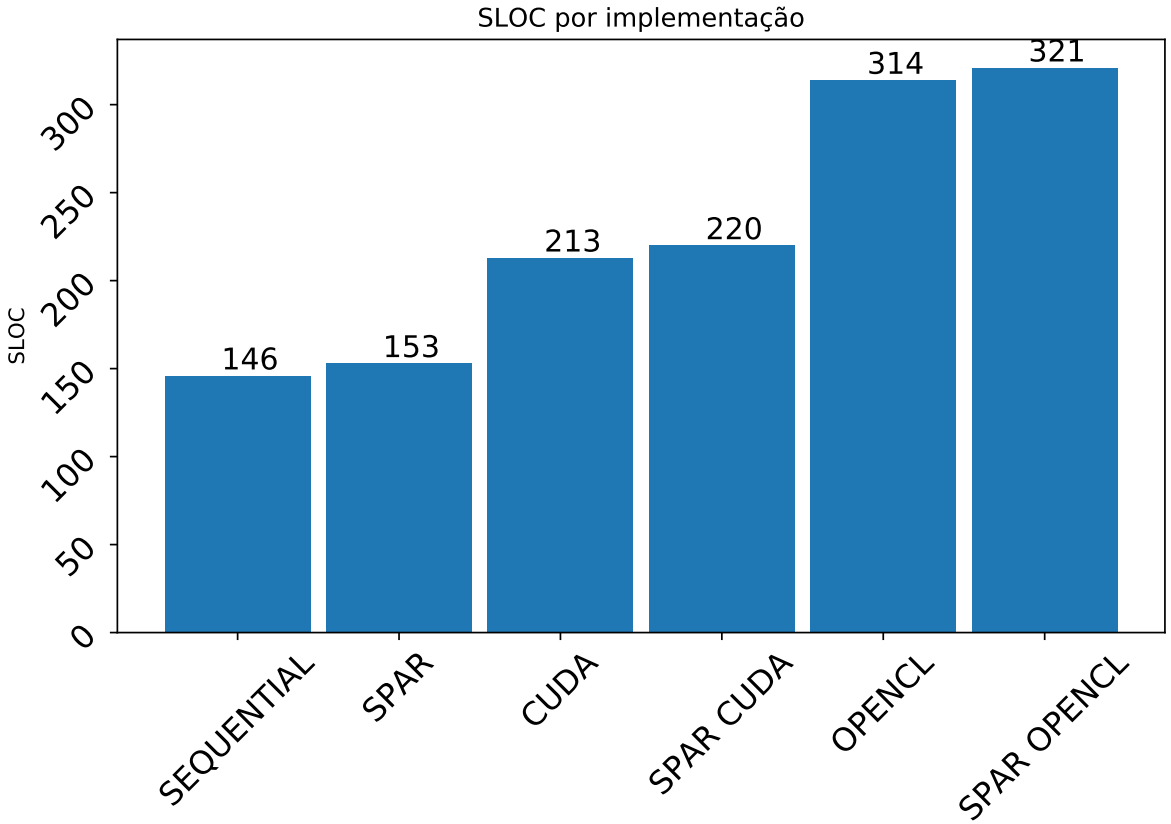
Figura 52: *Speedup* por implementação do *Black-Scholes*



Com isso, nota-se que o uso de GPU nesta aplicação se faz vantajoso quando o tempo de IO da aplicação não for um limitador para o desempenho. Ao realizar a leitura de arquivos para o cálculo as versões em GPU não foram melhores que utilizando apenas a *pipeline* em CPU. Por outro lado para grandes cargas de processamento, a GPU pode alcançar até 6.9x de *speedup*.

Em relação a intrusão de código para o *Black-scholes*, foram medidas as linhas de código necessárias para cada uma das implementações. Na Figura 53 é apresentado a quantidade de linhas para cada implementação. É possível notar que novamente as implementações com SPar apresentam pouco incremento no código sequencial. Por outro lado, as implementações com CUDA representam um aumento significativo nas linhas de código. Em relação ao OpenCL nota-se que novamente as linhas necessárias para a compilação JIT incrementam consideravelmente a intrusão de código.

Figura 53: Linhas de código por implementação do *Black-Scholes*.



3.3 VISÃO GERAL DOS RESULTADOS

A partir das aplicações paralelizadas, é possível ter uma visão de como aplicações de processamento de *stream* podem se beneficiar da GPU. Além disso, foi feito o uso do paralelismo combinado de CPU e GPU, para que assim fosse possível extrair o máximo do *hardware* disponível no computador.

Como foi visto, nem todas aplicações de processamento de *stream* conseguem obter um desempenho mais alto na GPU do que versões paralelizadas na CPU.

Além do desempenho variar, é possível notar que o paralelismo para a GPU requer um conhecimento maior da arquitetura da placa de vídeo e um alto conhecimento do problema a ser resolvido, visto que muitas vezes o paralelismo de GPU envolve alterações até mesmo na forma que o algoritmo trabalha.

Isso pode ser visto em aplicações como *Dedup*, *Black-scholes* e LZSS, onde foi necessária a alteração em como o algoritmo trabalha. No *dedup* e *Black-scholes*, o processamento da *stream* precisou ser modificado para que o paralelismo de dados fosse aplicado em uma quantidade maior de dados. No LZSS, foi necessário um estudo mais profundo do algoritmo para que a parte mais custosa do algoritmo fosse transferida para a GPU e os resultados fossem corrigidos na CPU.

O uso de GPU nas aplicações de processamento de *stream* resultou em um *speedup* que variou conforme a aplicação. No Filtro *Sobel* notou-se que não se obteve desempenho devido ao tempo de I/O tanto da leitura de arquivos quanto a transferencia CPU-GPU. O trabalho realizado pela GPU mesmo sendo paralelizado ao máximo, não foi vantajoso devido ao programa perder mais tempo realizando transferências de memória do que processando.

O LZSS obteve um grande ganho de desempenho, chegando a atingir até 36x de *speedup*. Isto se justifica devido a pouca transferência CPU-GPU e o processamento na GPU envolve uma alta demanda computacional. A alteração no algoritmo de LZSS permitiu um melhor uso tanto da GPU quanto da CPU, obtendo um desempenho melhor que o estado da arte. Enquanto Ozsoy e Swany (2011) conseguiram um *Speedup* de 18x utilizando CPU e GPU no *dataset* linux *tarball*, este trabalho alcançou o *speedup* de 36x.

No *Dedup* foi possível notar que o ganho de desempenho foi obtido tanto

sem compressão quanto com compressão. No entanto, sem a compressão, o paralelismo de CPU foi maior do que com uso de GPU. Semelhante ao filtro *Sobel*, a demanda computacional sem a compressão era muito baixa, limitando o ganho de desempenho neste formato. Suttisirikul e Uthayopas (2012) conseguiu um *speedup* de 53x, porém em sua implementação a operação de *hash* utilizada foi o *sha2*. Além disso, não foi paralelizada a operação de compressão como a presente neste trabalho.

Por último, no *Black-scholes* pode ser considerada uma aplicação que envolve computações intensivas de I/O, já que o *speedup* de 6x só foi obtido em memória. Utilizando a leitura de arquivos nos testes o *speedup* foi em grande parte obtido pela *pipeline* na CPU. Ray (2010) obteve 41% de *speedup* em seus testes, porém em sua implementação não foi paralelizada a aplicação no formato de processamento de *Stream*.

Na aplicação *ferret* não foi possível a paralelização. Isso porque o algoritmo possui interdependência de dados, tornando o algoritmo não determinístico.

Em relação a intrusão de código, foi possível notar que o *OpenCL* sempre envolve mais linhas de código para a mesma funcionalidade do CUDA. Isso ocorre devido as chamadas do CUDA serem mais simples e o compilador do CUDA realiza mais otimizações. No OpenCL, por ser multiplataforma e suportar paralelismo em outros dispositivos de *hardware*, o código é mais verboso e envolve a compilação *Just in Time*. O paralelismo de CPU foi simplificado através do uso da DSL SPar, onde em todos os casos precisou menos de 10 linhas de código para atingir o paralelismo em CPU.

CONCLUSÃO

O desenvolvimento do presente estudo possibilitou avaliar o comportamento de aplicações de processamento de *stream* (Filtro Sobel, LZSS, Dedup e *Black-Scholes*) quando paralelizadas para a GPU. Além disso, os testes realizados permitiram uma comparação de quando é vantajoso o uso de GPU neste tipo de aplicação.

Os testes mostraram que aplicações como LZSS, Dedup e *Black-Scholes* obtiveram um *speedup* de 36x, 13x e 6.9x quando executados em memória e com o uso de paralelismo de CPU e GPU. Por outro lado, as aplicações de *Filtro sobel* e *Black-scholes* executadas com o tempo de I/O não apresentaram um ganho de desempenho significativo, sendo mais vantajoso o uso de paralelismo apenas em CPU.

A partir da paralelização e execução de *benchmarks*, nota-se que o uso da GPU em aplicações de processamento de *stream* aumenta o desempenho em algumas dessas aplicações, porém, em outras aplicações o uso da GPU não é mais rápido que o uso do paralelismo em CPU. O *overhead* na transferência de memória para a GPU faz com que aplicações de processamento de *stream* precisem processar *batches* de dados para fazer o uso eficiente da GPU.

Em relação a intrusão de código, é possível notar que o CUDA exige menos linhas de código que o OpenCL em todos os casos devido a compilação JIT.

No entanto, ambas as bibliotecas incrementam de forma significativa o código sequencial. Diante das aplicações paralelizadas em CUDA e OpenCL, foi possível avaliar como as aplicações de processamento de *stream* se comportam na GPU.

A publicação de um artigo científico sobre o trabalho também foi realizada em Stein e Griebler (2018). A publicação do artigo foi através da exploração do paralelismo na aplicação de Filtro Sobel usando CUDA e SPar. Isso incluiu também uma análise comparativa do desempenho e programabilidade.

Em relação as hipóteses, os testes mostraram que a hipótese de que o uso da GPU nas aplicações de processamento de *stream* conseguem aumentar o desempenho em até 10 vezes foi corroborado para as aplicações *Dedup* e *Black-scholes*. Na aplicação LZSS a hipótese não foi corroborada já que o aumento foi maior do que o esperado, atingindo 36x. No filtro *Sobel* a hipótese não foi corroborada devido a aplicação não ter obtido *speedup*.

A segunda hipótese, de que a biblioteca CUDA é mais eficiente que o OpenCL para as aplicações de processamento de *stream* paralelizadas foi corroborada, já que em todos testes o CUDA apresentou um desempenho melhor que OpenCL. Além disso, o número de linhas de código exigido para a paralelização em CUDA foi menor em todos os casos.

Nota-se que a implementação de paralelismo de GPU envolve bastante conhecimento e alteração do código sequencial, tendo vezes em que nem é possível ou vantajoso o paralelismo em GPU, como pode ser visto ao se tentar paralelizar a aplicação *ferret*, que devido a equação não ser deterministica não permitiu o paralelismo para GPU.

Desta forma, este trabalho contribuiu com o estudo de aplicações de processamento de *stream*, dando suporte ao paralelismo em GPU para as aplicações LZSS, *dedup*, *filtro sobel* e *Black-Scholes*.

Apesar deste trabalho ter abordado 4 aplicações de processamento de

stream, estudos futuros ainda podem ser realizados, como o estudo de outras aplicações de processamento de *stream*, como o YOLO na área *Machine Learning* e outros algoritmos de compressão como o BZIP2.

Além disso, este trabalho tratou a produtividade apenas analisando a intrusão de código, seria possível aplicar ainda outras metodologias para medir a produtividade das bibliotecas apresentadas.

REFERÊNCIAS

ADAM TRENDOWICZ, R. J. a. **Software Project Effort Estimation**: foundations and best practice guidelines for success. [S.l.: s.n.], 2014.

BARRY BOEHM APURVA JAIN (AUTH.), Q. W. D. P. D. M. R. P. W. e. **Software Process Change**: international software process workshop and international workshop on software process simulation and modeling, spw/prosim 2006, shanghai, china, may 20-21, 2006. proceedings. [S.l.: s.n.].

BIENIA, C. et al. The PARSEC Benchmark Suite: characterization and architectural implications. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 17., New York, NY, USA. **Proceedings...** ACM, 2008. p.72–81. (PACT '08).

BIRD, C.; MENZIES, T.; ZIMMERMANN, T. **The Art and Science of Analyzing Software Data**. [S.l.]: Elsevier Science, 2015.

CHRISTIAN BIENIA SANJEEV KUMAR, J. P. S.; LI†, K. The PARSEC Benchmark Suite: characterization and architectural implications. , [S.l.], p.72–81, 10 2008.

COOK, S. **CUDA Programming: a developer's guide to parallel computing with gpus**. [S.l.]: Elsevier, 2013.

CUDA. **CUDA Toolkit Documentation** <**http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd**>. Último acesso em março, 2018.

DAVID A. BADER, R. P. Cluster Computing: applications. **The International Journal of High Performance Computing**, https://www.cc.gatech.edu/ bader/papers/ijhpca.html, v.2, n.15, p.181–185, May 2001.

DAVID SALOMON G. MOTTA, D. B. **Data compression**: the complete reference. 3.ed. [S.l.]: Springer, 2007.

DE, K.; GUPTA, Y. A Real-time Coherent Dedispersion Pipeline for the Giant Metrewave Radio Telescope. **Experimental Astronomy**, [S.l.], v.41, n.1-2, p.67–93, sep 2015.

DEOROWICZ, S. [S.l.]: Silesia Corpus, 2018.

DESHPANDE, A. **Combining Data Parallelism and Task Parallelism for Efficient Performance on Hybrid CPU and GPU Systems**. 2014. Tese (Doutorado em Ciência da Computação) — International Institute of Information Technology Hyderabad.

DUNCAN, R. A survey of parallel computer architectures. **Computer**, [S.l.], v.23, n.2, p.5–16, Feb 1990.

FARBER, R. **Parallel Programming with OpenACC**. [S.l.]: Elsevier/Morgan Kaufmann, 2016. (Morgan Kaufmann).

GARCIA, J. D. et al. **Stream parallelism patterns**<**http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0374r0.pdf**>. Acessado em março de 2017.

GRIEBLER, D. et al. An Embedded C++ Domain-Specific Language for Stream Parallelism. In: PARALLEL COMPUTING: ON THE ROAD TO EXASCALE, PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON PARALLEL COMPUTING, Edinburgh, Scotland, UK. **Anais...** IOS Press, 2015. p.317–326. (ParCo'15).

GRIEBLER, D. et al. SPar: A DSL for High-Level and Productive Stream Parallelism. **Parallel Processing Letters**, [S.l.], v.27, n.01, p.1740005, 2017.

GRIEBLER, D. et al. High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. **International Journal of Parallel Programming**, [S.l.], p.1–19, 2018.

GRIEBLER, D. J. **Proposta de uma Linguagem Específica de Domínio de Programação Paralela Orientada a Padrões Paralelos: um estudo de caso baseado no padrão mestre/escravo para arquiteturas multi-core**. 2012. Dissertação (Mestrado em Ciência da Computação) — Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil.

GRIEBLER, D. J. **Domain-Specific Language Support Tools for High-Level Stream Parallelism**. 2016. Tese (Doutorado em Ciência da Computação) — Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil.

HARIS, M. **Inside Pascal: nvidia's newest computing platform** <**https://devblogs.nvidia.com/parallelforall/inside-pascal/**>. Acessado em março de 2018.

HENNESSY, J.; PATTERSON, D. **Arquitetura de Computadores**: uma abordagem quantitativa. [S.l.]: Elsevier Brasil, 2014.

HWU, W. **Heterogeneous System Architecture**: a new compute platform infrastructure. [S.l.]: Elsevier Science, 2015.

HWU, W.-m. W. **GPU Computing Gems Emerald Edition**. 1st.ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

KAEHLER, A.; BRADSKI, G. **Learning OpenCV 3**: computer vision in c++ with the opencv library. [S.l.]: O'Reilly Media, 2016.

LIANG, L. **Parallel Implementations of Hopfield Neural Networks On GPU**. 2011. 37p. Dissertação (Mestrado em Ciência da Computação) — INRIA-IRISA Rennes Bretagne Atlantique, équipe CAIRN.

LOVATO, A. **Metodologia da Pesquisa**. [S.l.]: SETREM, 2013.

MARCO ALDINUCCI MARCO DANELUTTO, P. K.; TORQUATI, M. FastFlow: high-level and efficient streaming on multi-core. , [S.l.], v.1, May 2014.

MCCOOL, M.; REINDERS, J.; ROBISON, A. **Structured Parallel Programming**: patterns for efficient computation. [S.l.]: Elsevier Science, 2012.

MCCOOL, M.; ROBISON, A.; REINDERS, J. **Structured Parallel Programming: patterns for efficient computation**. [S.l.]: Elsevier/Morgan Kaufmann, 2012. (Morgan Kaufmann).

MEMETI, S. et al. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. **CoRR**, [S.l.], v.abs/1704.05316, 2017.

MUNSHI, A. et al. **OpenCL Programming Guide**. 1st.ed. [S.l.]: Addison-Wesley Professional, 2011.

MURAROLLI, P. **Inovações tecnológicas nas perspectivas computacionais**. [S.l.]: BIBLIOTECA 24 HORAS, 2015.

OZSOY, A.; SWANY, M. CULZSS: LZSS lossless data compression on CUDA. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING, 2011. **Anais...** IEEE, 2011.

PACHECO, P. **An Introduction to Parallel Programming**. [S.l.]: Morgan Kaufmann, 2011.

RAY, A. PARSEC Benchmark Suite: a parallel implementation on gpu using cuda. In: IEEE. **Anais...** [S.l.: s.n.], 2010.

REDMON, J. et al. You Only Look Once: unified, real-time object detection. **CoRR**, [S.l.], v.abs/1506.02640, 2015.

RUBNER, Y.; TOMASI, C.; GUIBAS, L. A metric for distributions with applications to image databases. In: SIXTH INTERNATIONAL CONFERENCE ON COMPUTER VISION (IEEE CAT. NO.98CH36271). **Anais...** Narosa Publishing House, 1998.

SANDERS, J.; KANDROT, E. **CUDA by Example**: an introduction to general-purpose gpu programming, portable documents. [S.l.]: Pearson Education, 2010.

SCARPINO, M. How to accelerate graphics and computation, OpenCL in Action. In: OPENCL IN ACTION. **Anais...** Manning, 2012.

SELBY, R. **Software Engineering**: barry w. boehm's lifetime contributions to software development, management, and research. [S.l.]: Wiley, 2007. (Practitioners Series).

STEIN, C. M.; GRIEBLER, D. Explorando o Paralelismo de Stream em CPU e de Dados em GPU na Aplicação de Filtro Sobel. In: ESCOLA REGIONAL DE ALTO DESEMPENHO DO ESTADO DO RIO GRANDE DO SUL (ERAD/RS), 18., Porto Alegre, RS, Brazil. **Anais...** Sociedade Brasileira de Computação, 2018. p.137–140.

SU, C.-L. et al. Overview and comparison of OpenCL and CUDA technology for GPGPU. In: APCCAS. **Anais...** [S.l.: s.n.], 2012.

SUTTISIRIKUL, K.; UTHAYOPAS, P. Accelerating the Cloud Backup Using GPU Based Data Deduplication. In: IEEE 18TH INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED SYSTEMS, 2012. **Anais...** IEEE, 2012.

The OpenACC™ Application Programming Interface. [S.l.: s.n.], 2013.

TURAGA, D. et al. Design Principles for Developing Stream Processing Applications. **Softw. Pract. Exper.**, New York, NY, USA, v.40, n.12, p.1073–1104, Nov. 2010.

UENO, K.; SUZUMURA, T. GPU task parallelism for scalable anomaly detection. In: IEEE. **Anais...** [S.l.: s.n.], 2012.

WU, N. et al. A Parallel H.264 Encoder with CUDA: mapping and evaluation. In: IEEE 18TH INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED SYSTEMS, 2012. **Anais...** IEEE, 2012.