PROGRAMAÇÃO DATAFLOW DE APLICAÇÕES DE FLUXO DE DADOS CONTÍNUO PARA SISTEMAS HETEROGÊNEOS

Marcos Paulo Carneiro Rocha

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Alexandre Solon Nery

Rio de Janeiro
Outubro de 2017

PROGRAMAÇÃO DATAFLOW DE APLICAÇÕES DE FLUXO DE DADOS CONTÍNUO PARA SISTEMAS HETEROGÊNEOS

Marcos Paulo Carneiro Rocha

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Felipe Maia Galvão França, PhD.

Prof. Alexandre Solon Nery, D.Sc.

Prof. Claudio Luiz de Amorim, PhD.

Prof. Cristiana Barbosa Bentes, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
OUTUBRO DE 2017

Rocha, Marcos Paulo Carneiro

Programação Dataflow de Aplicações de Fluxo de Dados Contínuo para Sistemas Heterogêneos/Marcos Paulo Carneiro Rocha. – Rio de Janeiro: UFRJ/COPPE, 2017.

X, 56 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França
Alexandre Solon Nery

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2017.

Referências Bibliográficas: p. 52 – 56.

1. Dataflow. 2. GPU. 3. Sistemas Heterogêneos. I. França, Felipe Maia Galvão *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Agradecimentos

Agradeço a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior(CAPES) pelo suporte financeiro, Isabella Vieira, pelo apoio, companheirismo e compreensão durante esta jornada. Agradeço aos amigos e familiares pelo apoio durante todo este período. Agradeço aos professores Leandro Marzulo pela inspiração e conselhos que me motivaram a continuar os estudos ingressando no mestrado, Alexandre Nery, por todo o apoio, esforço e dedicação desde o momento que passou a me orientar e ao Felipe França pelos conhecimentos, experiências e orientações que moldaram o rumo deste trabalho. Agradeço também a fundação COPPETEC pela oportunidade de trabalhar em seus projetos durante esta jornada, permitindo que me desenvolvesse em áreas diferentes a minha linha de pesquisa do mestrado e pela oportunidade de conviver e trabalhar com pessoas incríveis e talentosas que na sua maioria eram companheiros de mestrado e doutorado também. Agradeço também ao Leandro Roubert, companheiro desde antes do inicio dessa jornada, compartilhando desde as dúvidas sobre a decisão de ingressar no mestrado, passando pela disciplinas juntos e seguindo até os momentos finais.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

DATAFLOW PROGRAMMING OF STREAM APLICATIONS IN HETEROGENOUS SYSTEMS

Marcos Paulo Carneiro Rocha

Outubro/2017

Orientadores: Felipe Maia Galvão França
Alexandre Solon Nery

Programa: Engenharia de Sistemas e Computação

Aplicações *stream* possuem demandam rigorosos requisitos de performance que são difíceis de serem atingidos utilizando modelos paralelos tradicionais em arquiteturas *many-cores* como *GPUs*. Por outro lado, os recentes modelos de computação *Dataflow* podem naturalmente explorar paralelismo em uma abrangente classe de aplicações. Este trabalho apresenta uma extensão para uma biblioteca *Dataflow* em Java. Esta extensão implementa construções em alto nível com múltiplas filas de comando que permitem a sobreposição de operações de memória e execução de *kernel* em *GPUs*. Os resultados deste trabalho mostraram que um significante *speedup* pode ser atingido para um conjunto de aplicações bem conhecidas de processamento *stream* como: *Ray-Casting*, *Path-Tracing* e filtro Sobel.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

PROGRAMAÇÃO DATAFLOW DE APLICAÇÕES DE FLUXO DE DADOS CONTÍNUO PARA SISTEMAS HETEROGÊNEOS

Marcos Paulo Carneiro Rocha

October/2017

Advisors: Felipe Maia Galvão França
Alexandre Solon Nery

Department: Systems Engineering and Computer Science

Stream processing applications have high-demanding performance requirements that are hard to tackle using traditional parallel models on modern many-core architectures, such as GPUs. On the other hand, recent dataflow computing models can naturally exploit parallelism for a wide class of applications. This work presents an extension to an existing dataflow library for Java. The library extension implements high-level constructs with multiple command queues to enable the superposition of memory operations and kernel executions on GPUs. Experimental results show that significant speedup can be achieved for a subset of well-known stream processing applications: Volume Ray-Casting, Path-Tracing and Sobel Filter.

Sumário

Lista de Figuras **ix**

1 Introdução **1**
1.1 Motivação . . . 2
1.2 Objetivos & Contribuições . . . 2
1.3 Benchmarks . . . 3
1.3.1 Volume Ray-Casting . . . 4
1.3.2 Path-Tracing . . . 4
1.3.3 Filtro Sobel . . . 7
1.4 Organização . . . 8

2 Fundamentação Teórica **9**
2.1 GPUs: Arquitetura e Programação . . . 9
2.1.1 Arquitetura . . . 10
2.1.2 Modelo de Programação de GPU . . . 11
2.2 OpenCL . . . 12
2.2.1 OpenCL em Java . . . 14
2.2.2 Comunicação com múltiplas filas . . . 15
2.3 Programação *Dataflow* usando Sucuri . . . 17
2.3.1 Modelo Dataflow . . . 17
2.3.2 Sucuri . . . 18
2.4 Trabalhos Relacionados . . . 21

3 Sucuri Java para Stream Processing **24**
3.1 Arquitetura . . . 24
3.2 Utilizando Múltiplas filas, execução concorrente de kernel para Stream Processing . . . 27

4 Resultados Experimentais **29**
4.1 Versionamento JVM . . . 34
4.2 Modelagem Dataflow JSucuri . . . 38
4.2.1 Modelagem Ray-Casting . . . 38

4.2.2 Modelagem Path-Tracing 41
4.2.3 Modelagem Sobel 44
4.3 Desempenho & Escalabilidade 47

Referências Bibliográficas **52**

Lista de Figuras

1.1 Exemplo da representação de um trecho de código descrito no modelo *Datalow*. 3
1.2 Exemplo de execução do algoritmo de Ray-Casting e da representação de um volume em camadas. 4
1.3 Exemplo dos vários raios difusos que podem ser usados para compor a cor de um pixel da imagem. 5
1.4 Exemplo de uma das primeiras versões da caixa de Cornell (esq.) e sua variação (dir.) usando apenas esferas. 6
1.5 Exemplo da aplicação do filtro Sobel. 8

2.1 *Pipeline* 3D simplificado. 10
2.2 Visão geral da arquitetura programável das GPUs NVidia. 11
2.3 Modelo de programação *GPU*. 12
2.4 Organização de tarefas e acesso à memória de um sistema OpenCL. . 14
2.5 Execução Concorrente de kernel. 16
2.6 Execução de kernel concorrente com cópia de dados. 16
2.7 Exemplo de aplicação descrita utilizando os nós *Source* e *Serializer*. . 19
2.8 Arquitetura da Sucuri. 20

3.1 Grafo Dataflow com os novos nós responsáveis pelas operações OpenCL 26
3.2 Ilustração da Dependência Adicional no Grafo para evitar Explosão de paralelismo. 28

4.1 Representação simplificada dos estágios de uma iteração de uma aplicação stream padrão 30
4.2 Representação da independência entre as diferentes iterações em um exemplo de aplicação stream 30
4.3 Representação da dependência entre as diferentes iterações executadas por um mesmo ramo do grafo 32
4.4 Representação simplificada da dependência entre as diferentes iterações executadas por um mesmo ramo do grafo 33

4.5 Comparativo do tempo de execução entre jvm 1.6 e 1.8 do algoritmo Ray-Casting com resolução de 320x240 35
4.6 Comparativo do tempo de execução entre jvm 1.6 e 1.8 do algoritmo Ray-Casting com resolução de 640x480 35
4.7 Comparativo do tempo de execução entre jvm 1.6 e 1.8 do algoritmo Ray-Casting com resolução de 800x600 36
4.8 Comparativo do tempo de execução entre jvm 1.6 e 1.8 do algoritmo Ray-Casting com resolução de 1280x800 36
4.9 Comparativo do tempo de execução entre jvm 1.6 e 1.8 do algoritmo Ray-Casting com resolução de 2000x1000 37
4.10 Grafo Dataflow modelado na JSucuri para a aplicação Ray-Casting . 40
4.11 Grafo Dataflow modelado na JSucuri para a aplicação Path-Tracing . 41
4.12 Grafo Dataflow modelado na JSucuri para a aplicação Sobel 44
4.13 O algoritmo Ray-Casting executado com apenas uma fila 47
4.14 O algoritmo Ray-Casting executado com 3 filas 47
4.15 Speedup do algoritmo Ray-Casting com relação ao número de filas para imagens de 640x320 49
4.16 Speedup do algoritmo Ray-Casting com relação ao número de filas para imagens de 1280x720 49
4.17 Speedup do algoritmo Ray-Casting com relação ao número de filas para imagens de 1920x1080 49
4.18 Speedup do algoritmo Sobel com relação ao número de filas para imagens de 640x320 50
4.19 Speedup do algoritmo Sobel com relação ao número de filas para imagens de 1280x720 50
4.20 Speedup do algoritmo Sobel com relação ao número de filas para imagens de 1920x1080 50
4.21 Speedup do algoritmo Path-Tracing com relação ao número de filas para imagens de 640x320 51
4.22 Speedup do algoritmo Path-Tracing com relação ao número de filas para imagens de 1280x720 51
4.23 Speedup do algoritmo Path-Tracing com relação ao número de filas para imagens de 1920x1080 51

Capítulo 1

Introdução

A Lei de Moore, uma observação feita pelo co-fundador da Intel Gordon Moore em 1965, dizia que o número de transistores em um chip integrado dobraria a cada 2 anos. Essa Lei se manteve por 50 anos e agora começa a perder força[1]. Até então, este maior número de transistores permitiu avanços relevantes na micro-arquitetura do processador, com a inclusão de caches maiores, hardware especializado para emissão múltipla de instruções, preditores de desvio, etc. Tais mudanças garantiam um ganho de desempenho em aplicações sem a necessidade de alteração do código fonte. Entretanto, essas novas e complexas funcionalidades que eram adicionadas aos processadores, em conjunto com o aumento da frequência do relógio a cada nova geração de processadores, também contribuíram para o aumento da dissipação de calor [2], até o ponto em que a dissipação atingiu patamares proibitivos, impossibilitando o contínuo aumento da frequência.

Diante das limitações de frequência do relógio, dissipação de calor e saturação de desempenho, as companhias desenvolvedoras de chips foram obrigadas a buscar novas arquiteturas que fossem capazes de sustentar seu lucrativo modelo de negócios, *i.e.*, que justificasse a compra de um novo processador a cada nova geração. A solução que se sobressaiu foi a replicação dos núcleos, dando início a era das arquiteturas *multi-core*. Posteriormente, devido ao aumento na importância de eficiência energética e por seu promissor potencial de paralelismo, as unidades de processamento gráfico (GPU – *Graphics Processing Unit*), antes utilizadas somente para processar/renderizar imagens, começaram a ser utilizadas para computação de propósito geral. Mais recentemente, arquiteturas re-configuráveis, como FPGAs (*Field-Programmable Gate Array)*, têm possibilitado o projeto de aceleradores em hardware FPGA customizados para processar uma aplicação específica ou um conjunto de aplicações [3], atingindo resultados mais eficientes de desempenho por potência dissipada (*performance-per-watt*) em comparação com as demais arquiteturas [4].

1.1 Motivação

Os sistemas computacionais modernos voltados para alto desempenho são cada vez mais compostos por diversos tipos de dispositivos, distribuídos em rede ou não, e equipados com diferentes tipos de micro-processadores [5, 6]. Estes distintos dispositivos apresentam diferentes capacidades de processamento e níveis de paralelismo. Para se atingir o máximo de desempenho, muitas vezes é necessário utilizá-los em conjunto numa aplicação. Aplicações de fluxo de dados contínuos constituem um nicho de aplicações que podem se beneficiar do potencial de processamento destes diversos dispositivos, em especial das características presentes nas *GPUs* modernas, como execução concorrente de *kernel* e cópia de dados concorrente. Estas características aumentam o potencial de paralelismo, permitindo operações simultâneas de diferentes iterações do fluxo de dados da aplicação.

As interações entre as diferentes iterações, dispositivos e tarefas de uma aplicação de fluxo de dados contínuos, podem ser enxergados como um grafo de dependência de dados, sendo o modelo *Dataflow* [7] um paradigma adequado para facilitar a escrita desse tipo de aplicação. No modelo *Dataflow* o programa é descrito como um grafo onde os nós representam as tarefas e as arestas as dependências de dados. Um nó pode executar quando todos os seus operandos de entrada estiverem disponíveis. A Figura 1.1 (Reproduzida de: [8]) ilustra um trecho de código descrito como um grafo *Dataflow*. No modelo *Dataflow* a execução de uma aplicação é guiada pelo fluxo de dados e não pelo contador de programa, como ocorre em programas Von Neuman. A execução guiada pelo fluxo de dados implica que todos os nós que tiverem suas dependências de dados satisfeitas podem ser executados de forma concorrente. Essa característica faz com o que o paralelismo real da aplicação seja exposto de forma natural. Em aplicações *stream* este potencial é multiplicado pois as diferentes iterações que são independentes entre si, aumentam o número de nós que podem ser executados concorrentemente. O uso de operações assíncronas dentro dos nós do grafo *Dataflow* pode aumentar o desempenho da aplicação pois permite que um nó tenha sua execução finalizada mais rapidamente e enquanto ocorre sua operação assíncrona, a aplicação está realizando a troca de operandos no grafo *Dataflow* sobrepondo computação e comunicação.

1.2 Objetivos & Contribuições

Este trabalho apresenta um sistema de execução *Dataflow* em Java para computação de alto desempenho em sistemas heterogêneos que utilizam *CPU* e *GPU*. Mais especificamente, este trabalho tem como objetivo permitir que aplicações descritas no modelo *Dataflow* possam também executar tarefas assíncronas, sobrepondo com-

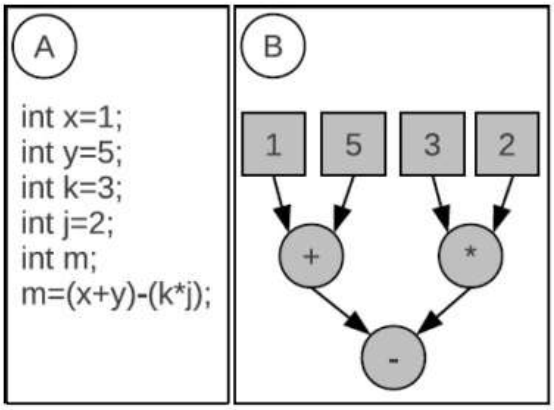


Figura 1.1: Exemplo da representação de um trecho de código descrito no modelo *Datalow*.

putação e cópia de dados. As contribuições deste trabalho são:

- Migração da biblioteca de execução *Dataflow* Sucuri[9] para Java, permitindo a construção de aplicações *Dataflow* com suporte a execução concorrente de *kernel* e cópia de dados concorrente, para sistemas de computação heterogêneos,
- Otimização da comunicação entre os nós *Dataflow* do modelo de execução, substituindo a serialização de objetos por suas referências. Desta forma, é possível que referências JavaCL sejam transmitidas de um nó para outro do grafo *Dataflow*, permitindo que nós distintos possam enfileirar na *GPU* comandos de execução de *kernel* e de cópia de dados, potencialmente sobrepondo-as.
- Facilitação do desenvolvimento de aplicações de fluxo de dados utilizando computação heterogênea com o modelo *Dataflow* em uma linguagem de alto nível e com menor esforço de programação, abstraindo detalhes de comunicação e concorrência.
- Avaliação de desempenho de aplicações de computação heterogênea com foco na execução *Dataflow* usando o modelo proposto. As aplicações avaliadas (*Ray Casting*, *Path Tracer* e Filtro *Sobel*) possuem grande potencial de processamento por fluxo de dados (*Stream Processing*).

1.3 Benchmarks

Esta Seção apresenta brevemente as aplicações (*Benchmarks*) que foram modeladas em grafos de fluxo de dados (*Dataflow*) para execução no sistema Sucuri-Java. Maiores detalhes sobre a modelagem dos grafos dataflow de cada aplicação serão apresentados no Capítulo 4.

1.3.1 Volume Ray-Casting

O algoritmo de visualização de volumes *Volume Ray-Casting* é um dos mais básicos e conhecidos algoritmos de renderização de volumes 3-D, capturados a partir de tomografia computadorizada e ressonância magnética [10]. Os volumes em três dimensões são descritos através de um conjuntos de dados de campos escalares, como mostra a Figura 1.2.

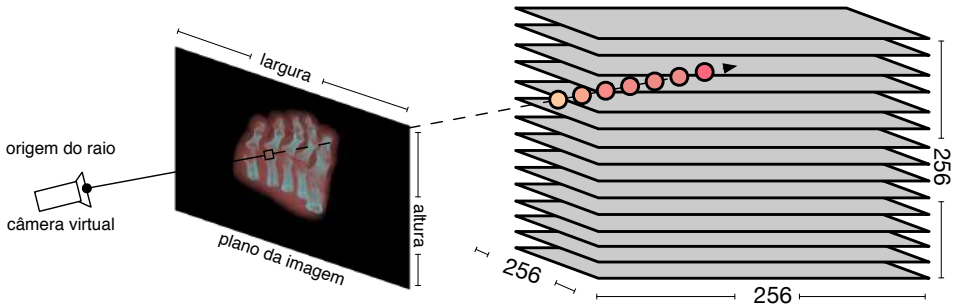


Figura 1.2: Exemplo de execução do algoritmo de Ray-Casting e da representação de um volume em camadas.

Conjunto de dados escalares representam cada ponto no espaço da cena 3-D que descrevem, como um valor. Por definição, são uma forma de representação independente de uso de coordenadas. O algoritmo calcula o valor de um *pixel*, como visto a partir de um ponto específico de observação, conhecido como olho da câmera ou olho do espectador. Através deste ponto é disparado um ou maios raios (parâmetro definido como amostra) para computar a quantidade de luz transposta por um raio até o plano da imagem [11]. O tamanho dos volumes utilizados nos experimentos é de $256 \times 256 \times 256$ *voxels*, nome que é atribuído aos dados que compõem o volume tridimensional. O volume também é conhecido por como *grid*. O pseudocódigo do Ray-Casting pode ser descrito como mostra o Algoritmo 1.

1.3.2 Path-Tracing

O algoritmo *Path-Tracing*, foi criado como uma solução para resolver a equação de renderização descrita por James Kajiya [12]. Este algoritmo é utilizado para renderização de imagens tridimensionais com efeitos ainda mais realísticos de luz, incluindo efeitos de suavização de sombras e iluminação indireta. A imagem é renderizada a partir de uma cena descrita por objetos tridimensionais, do ponto de vista de uma câmera. Ao contrário do algoritmo de *Ray-Tracing*, cujos raios secundários apenas são produzidos a partir de superfícies polidas, o algoritmo de *Path-Tracing* produz um ou mais raios secundários em diferentes direções a partir de superfícies

Algoritmo 1 Volume Ray-Casting

```
1:  função RAYCASTING(imagem, numAmostras, camera, grid)
2:      para i = 0 to imagem.largura faça
3:          para j = 0 to imagem.altura faça
4:              para s = 0 to numAmostras faça
5:                  raio = DISPARARRAIO(i, j)
6:                  se INTERCEPTAGRID(raio)) então
7:                      enquanto iter < MAX_ITER faça
8:                          calcula cor
9:                          se cor saturou então
10:                             devolve cor
11:                         fim se
12:                     fim enquanto
13:                 fim se
14:             fim para
15:             cor = media(cor, numAmostras)
16:         fim para
17:     fim para
18: fim função
```

difusas. A Figura 1.3 ilustra como um caminho de raios é formado após sucessivas rebatidas em objetos de uma cena.

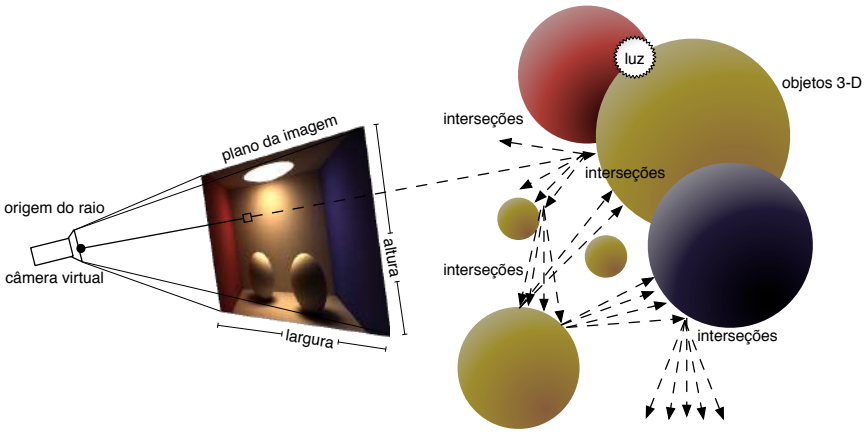


Figura 1.3: Exemplo dos vários raios difusos que podem ser usados para compor a cor de um pixel da imagem.

Neste algoritmo, para calcular o valor de cada *pixel* são emitidos raios que possuem sua origem no ponto que representa a câmera. Esses raios colidem com a superfície dos objetos presentes na cena e vão sendo rebatidos em direções aleatórias, formando uma cadeia de raios conectados por um caminho. A cor do *pixel* a ser retornada pelo algoritmo é calculada levando em consideração a cor da superfície de todos os objetos encontrados pelo caminho, a cor da fonte de luz, o ângulo cujo caminho atingiu a superfície e o ângulo cujo caminho foi rebatido para fora da su-

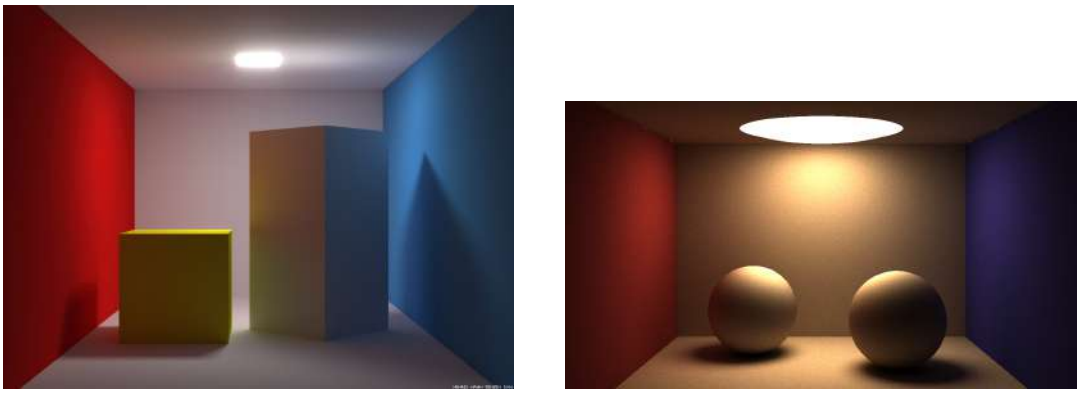
perfície. O pseudocódigo simplificado da computação realizada para o cálculo de um *pixel* pode ser descrito como mostra o Algoritmo 2.

Algoritmo 2 Path-Tracing

```
1: função PathTracing(objetosCena, raioCamera, objetosCena)
2:     para i = 0 to numeroAmostras faça
3:         para j = 0 to numRebatidas faça
4:             se não interceptouCena(raioCamera) então devolve corFundo
5:             senão
6:                 buscar objeto interceptado mais próximo
7:                 computar ponto interceptado
8:                 calcular direção prõximo raio
9:                 adicionar contribuição cor
10:            fim se
11:        fim para
12:    fim para
13: fim função
```

Como a direção dos raios rebatidos é gerada de forma aleatória, alguns caminhos gerados não encontraram interseções, o que pode gerar ruído na imagem, devido a variação introduzida pela aleatoriedade. Este ruído pode ser minimizado com o disparo de mais raios por *pixel* e utilizando-se a média destas amostras.

Na implementação do algoritmo *Path-Tracing* foi utilizada como cena de entrada uma variação da clássica cena conhecida como Caixa de *Cornell* [13], como mostra a Figura 1.4. Uma das primeiras versões da Caixa de *Cornell* é mostrada na Figura 1.4(a). Para este trabalho a cena foi modificada para ser descrita através de nove esferas com sobreposições entre elas, como mostra a Figura 1.4(b). A câmera é posicionada bem próxima da cena principal para passar a impressão de que se está de frente para uma sala quadrada. As paredes, chão e teto são partes das esferas e a luz também é uma esfera com a propriedade de emissão de luz.



(a) Caixa de Cornell [14].

(b) Variação da caixa de Cornell.

Figura 1.4: Exemplo de uma das primeiras versões da caixa de Cornell (esq.) e sua variação (dir.) usando apenas esferas.

Nesta implementação, a cena foi descrita como uma simples animação 3-D, para simular o comportamento de uma aplicação *stream*. Na animação, a esfera da direita se desloca para cima e para baixo com velocidade constante.

1.3.3 Filtro Sobel

O filtro de Sobel [15] é um algoritmo de processamento de imagens utilizado para enfatizar as bordas dos elementos (objetos, pessoas, etc.) presentes em uma imagem. Por isso é um algoritmo muito usado em aplicações de visão computacional. O filtro determina o valor de intensidade de cada *pixel* calculando o vetor gradiente ao longo dos eixos x e y. Para isso, são usadas duas matrizes de convolução, G_x e G_y, uma para cada um dos eixos, descritas na Equação 1.1.

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \qquad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{1.1}$$

O valor do vetor gradiente é dado pela Equação 1.2.

$$G = \sqrt{G_x^2 + G_y^2} \tag{1.2}$$

Uma outra forma de cálculo de gradiente, aplicada neste trabalho, também é dada pela Equação 1.3.

$$G = abs(G_x) + abs(G_y) \tag{1.3}$$

O pseudocódigo do Filtro de Sobel, realizado em cada *pixel*, está descrito no Algoritmo 3 e a Figura 1.5 ilustra o efeito do filtro aplicado a uma imagem de entrada.

Algoritmo 3 Filtro Sobel

```
1:  função FILTROSOBEL(imagem)
2:      G_x[3][3] = {{−1, 0, 1}, {−2, 0, 2}, {−1, 0, 1}}
3:      G_y[3][3] = {{1, 2, 1}, {0, 0, 0}, {−1, −2, −1}}
4:      para i = 1 to imagem.largura faça
5:          para j = 1 to imagem.altura faça
6:              cálculo de convolução no eixo x
7:              cálculo de convolução no eixo y
8:              cálculo do gradiente G
9:              imagem[i][j] = nova cor
10:         fim para
11:     fim para
12: fim função
```

(a) Entrada.

(b) Saída.

Figura 1.5: Exemplo da aplicação do filtro Sobel.

1.4 Organização

Este trabalho está organizado da seguinte forma: o Capítulo 2 discorre sobre tópicos referentes a programação paralela, *GPGPU*, computação heterogênea e sobre a linguagem Java, utilizada neste trabalho. O capítulo 3 explica os conceitos de aplicações *stream*, comunicação assíncrona e sobre a migração da biblioteca Sucuri[9] para Java, a fim de possibilitar a emissão de comandos do OpenCL juntamente com o modelo *Dataflow*, explorando, assim, características de placas de vídeo modernas como comunicação assíncrona em conjunto com execução concorrente de *kernel* e cópia de dados concorrente com execução de *kernel*. O capítulo 4 inclui a descrição dos modelos dataflow implementados nesse trabalho, bem como os resultados experimentais. Finalmente, o Capítulo **??** apresenta as conclusões deste trabalho.

Capítulo 2

Fundamentação Teórica

Este capítulo descreve todos os aspectos relevantes à compreensão do trabalho, com ênfase em Programação *GPU*, Programação *Dataflow* e OpenCL em Java. Além disso, ao término do capítulo, é apresentada uma revisão bibliográfica acerca dos trabalhos relacionados ao tema desta dissertação.

2.1 GPUs: Arquitetura e Programação

Os microprocessadores são uma classe de circuitos integrados cuja organização é tradicionalmente baseada na arquitetura de Von-Neumann e que, ao longo das últimas décadas, passou por diversas transformações com o propósito de aumentar a eficiência da execução de código sequencial, incluindo lógica em hardware que permite predição de desvios [16], despacho múltiplo de instruções, execução fora de ordem [16], etc.

Os avanços das tecnologias de produção em larga escala de transistores cada vez menores foi um dos grandes responsáveis por tornar possível a implementação de circuitos mais complexos em hardware e que, mais recentemente, permitiu o desenvolvimento de microprocessadores de 64-bits e múltiplos núcleos (*multicore*) [17], contendo dezenas de milhões de transistores na mesma área que, na década de 70, era ocupada por duas centenas de transistores que implementavam um único microprocessador de 4-bits.

A Lei de Moore [18], datada de meados da década de 60, já previa tal acontecimento com base na observação do avanço da indústria de circuitos integrados, profetizando que o número de transistores em um microchip dobraria aproximadamente a cada 18 meses. Obviamente, os microprocessadores gráficos (*GPUs*) também seguiram esta tendência, saltando de 3 milhões de transistores para 21 bilhões de transistores [19], entre os anos de 1997 e 2017.

Nas próximas seções serão apresentados os conceitos sobre a arquitetura, o modelo de programação e as linguagens para programação de *GPUs* modernas.

2.1.1 Arquitetura

Os microprocessadores gráficos, popularmente conhecidos como *GPUs*, são circuitos integrados dedicados à renderização de imagens a partir de cenas tridimensionais. Uma das primeiras *GPUs* foi desenvolvida pela IBM em meados da década de 80 [20], permitindo a realização de todo o processamento gráfico em hardware. Antes do desenvolvimento das *GPUs*, o processamento gráfico era executado via software pela própria Unidade Central de Processamento (UCP). Com a popularização dos PCs domésticos, ocorreu uma crescente demanda por jogos eletrônicos de maior qualidade, o que impulsionou a venda e a evolução das *GPUs*.

Nesta época, a arquitetura das *GPUs* seguia o simples modelo de *pipeline* fixo apresentado na Figura 2.1, por onde os dados gráficos passavam no processo de conversão de objetos 3D para uma representação dos objetos em 2D para que, finalmente, pudessem ser exibidos em monitores. Este processo também é conhecido por Rasterização.

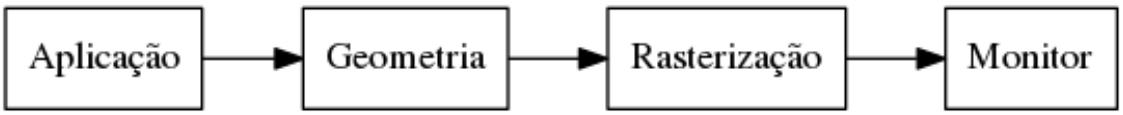


Figura 2.1: *Pipeline* 3D simplificado.

Este modelo de *pipeline* não permitia nenhum controle de como a imagem final era gerada, o que limitava os desenvolvedores de jogos e artistas profissionais. Para solucionar esta limitação, foi introduzida a programabilidade de alguns estágios do *pipeline* gráfico. Esta programação era realizada através de programas descritos em linguagens de baixo nível, ex., *OpenGL Shading Language - GLSL*[21], e substituíam suas respectivas seções no *pipeline* 3D.

Com o passar do tempo, foram criadas *APIs* (Interface de Programação de Aplicação) e linguagens de mais alto nível para facilitar a programação dos estágios do *pipeline*. A possibilidade de programar os estágios do *pipeline* acabou por introduzir também a capacidade de realizar outros tipos de processamento além do processamento gráfico, mesmo que através de linguagens e *APIs* gráficas. Tais *APIs* abstraem boa parte da complexidade de programação das *GPUs*, apresentando aos programadores-usuários um co-processador composto de múltiplos processadores que podem ser programados. Sendo assim, a arquitetura das *GPUs* modernas é capaz de produzir uma alta vazão de processamento.

As *GPUs* NVidia seguem o modelo de processamento por fluxo, i.e., *Stream Processing*, que organiza vários processadores SIMD (*Single-Instruction Multiple-Data*) em um arranjo de *Stream Multiprocessors*, ou simplesmente SM, como mostra a Fi-

gura 2.2. Dessa forma, a aplicação a ser executada precisa ser distribuída em blocos de *threads* que serão executados pelos núcleos SIMD de cada SM. Os processadores SIMD são conceitualmente mais simples do que uma *CPU* e são otimizados para processamento de fluxo de dados. Portanto, as aplicações podem executar de forma eficiente e com alto grau de paralelismo.

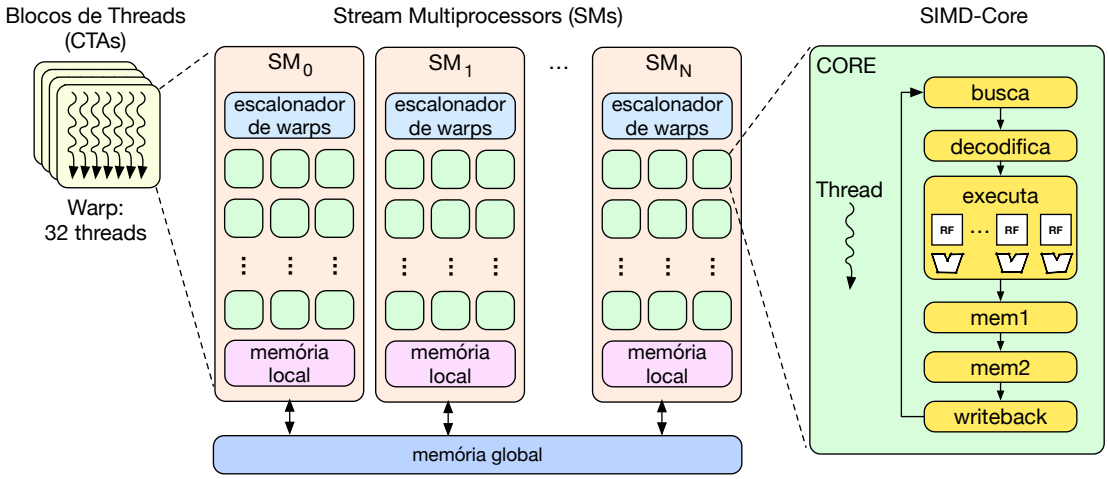


Figura 2.2: Visão geral da arquitetura programável das GPUs NVidia.

Ainda que esta seção apresente detalhes da arquitetura *GPU* Nvidia, este trabalho não é limitado pela marca, tipo ou modelo de arquitetura da *GPU*, podendo operar em qualquer tipo de *GPU* compatível com OpenCL, como será apresentado na Seção 2.1.2.

2.1.2 Modelo de Programação de GPU

Com a evolução das *GPUs*, os estágios do *pipeline* gráfico foram evoluindo de estágios fixos para programáveis a ponto de se tornarem genéricos o suficiente para considerar a *GPU* como um co-processador de propósito geral.

As *GPUs* são, em geral, programadas através de uma *API* proprietária de cada fabricante ou através da *API* OpenCL (*Open Computing Language*), voltada para programação de sistemas paralelos heterogêneos, ou seja, que são compostos de diferentes dispositivos. Do ponto de vista do programador, a *GPU* opera como um co-processador composto de vários núcleos computacionais, cada qual também composto por várias unidades de ponto flutuante. O co-processador é acessado, i.e., controlado, a partir de um computador hospedeiro, comumente chamado de *host*, via barramento de alta velocidade (ex: PCIe), como mostra a Figura 2.3.

O hospedeiro, por sua vez, emite comandos à *GPU* para alocação de memória, cópia de dados e execução de tarefas. Tais tarefas são chamadas de *kernels*, pois

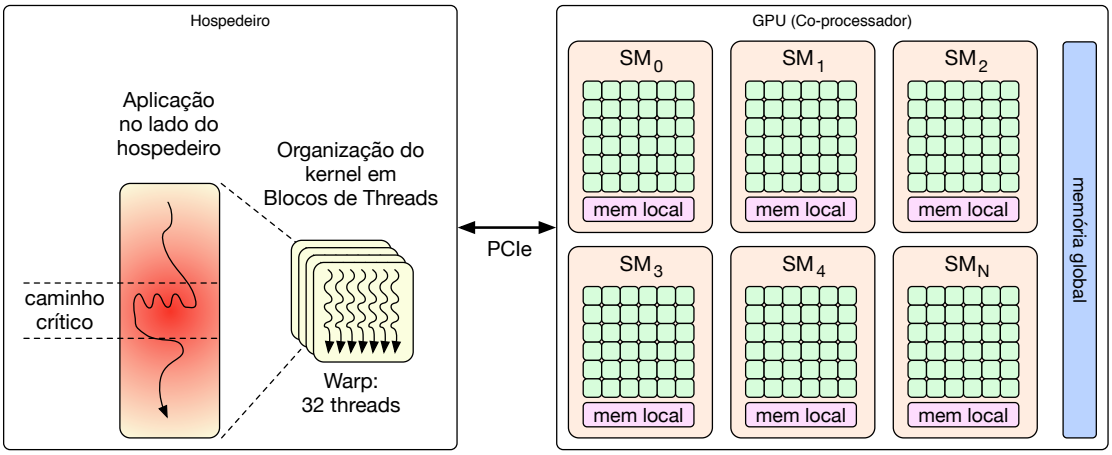


Figura 2.3: Modelo de programação *GPU*.

geralmente são aquelas que estão no caminho crítico da aplicação que se deseja executar (acelerar) em *GPU*, ou seja, que consomem mais tempo de processamento e que demandam mais recursos. Além disso, como uma *GPU* é utilizada como um co-processador, sua memória está separada da memória principal acessada pela *CPU*, sendo necessário então que ocorram cópias de dados da memória acessada pela *CPU* para a memória da *GPU*, e vice-versa.

A *GPU*, por sua natureza é voltada para a execução de código paralelo, o que permite um aproveitamento eficiente dos transistores proporcionados pela evolução da tecnologia dos semi-condutores. Embora hoje considere-se uma *GPU* um processador de propósito geral, seus núcleos são menos complexos e potentes do que um núcleo de uma *CPU*, que é voltada para execução eficiente de código sequencial. No entanto, uma *GPU* pode conter centenas ou milhares de núcleos ao contrário de uma *CPU*. O modelo NVidia GeForce 1080 Ti, com arquitetura Pascal, contém em torno de 3500 núcleos de processamento gráfico, popularmente conhecidos por CUDA-Cores.

2.2 OpenCL

O OpenCL (*Open Computing Language*) é um padrão proposto pelo grupo Khronos que visa unificar o desenvolvimento em plataformas heterogêneas constituídas de Unidades Centrais de Procesamento (*CPUs*), Unidades Gráficas de Processamento (*GPUs*), Processamento de Sinais Digitais (*DSPs*), Arranjo de Portas Lógicas Programáveis em Campo (‘FPGAs) e outros dispositivos [22]. O OpenCL especifica linguagens de programação (baseadas em C99 e C+11) para descrever o código do *kernel* que será executado pela *GPU*. O programa do *host* e o *kernel* são as duas

principais unidades de execução em OpenCL. As principais tarefas executadas pelo *host* são:

- Buscar pelos dispositivos OpenCL e seus atributos que estão disponíveis na máquina hospedeira;
- Definir um contexto de execução para os *kernels*;
- Compilar os programas dos *kernels*;
- Gerenciar a execução de um ou mais *kernels*.

Um *kernel* é geralmente instanciado para ser executado por diferentes *threads* na *GPU*. Logo, os *kernels* podem ser mapeados para diferentes unidades de execução, associando, por exemplo, um grupo de tarefas (*work-group*) a um *Stream Multiprocessor* e cada tarefa a um de seus núcleos de processamento. As tarefas são organizadas para execução na *GPU* pelo programa do hospedeiro, que cria um espaço N-dimensional de unidades de trabalho (*work-item*) onde cada tarefa será executada, como mostra a Figura 2.4. A memória privada é a que pode ser acessada por somente uma tarefa de trabalho e a que tem o menor tempo de acesso. O *host* e outras tarefas de trabalho não podem ler ou escrever na memória privada. A memória local é compartilhada entre as tarefas de trabalho dentro de um grupo de trabalho. Esta memória é útil se mais de uma tarefa de trabalho em um grupo precisa de acesso à um pedaço de memória global específico. O tempo de acesso à memória local é muito menor que o tempo de acesso à memória global. Além disso, o *host* não tem permissão para ler ou escrever na memória local. Por fim, a memória global, é onde todas as tarefas de trabalho podem acessar. Qualquer tarefa de trabalho pode ler ou escrever em um *buffer* declarado na memória global. Além disso, o *host* pode ler ou escrever na memória global.

As construções básicas de OpenCL são aprimoradas pelas *APIs* usadas para controlar o dispositivo e sua execução [22]. O OpenCL traz três importantes facilidades para os desenvolvimentos da computação de alto desempenho: (i) ele permite o desenvolvimento de aplicações para qualquer arquitetura suportada, (ii) ele fornece uma camada de abstração que permite que os desenvolvedores se concentrem na paralelização de seu algoritmo e (iii) ele deixa livre para os fabricantes implementarem as especificações do padrão de forma otimizada para seus dispositivos.[22]. Como a implementação do padrão OpenCL é de responsabilidade do fabricante e nem todas as funcionalidades descritas no padrão OpenCL são de implementação obrigatória, o desempenho e portabilidade das aplicações pode variar.

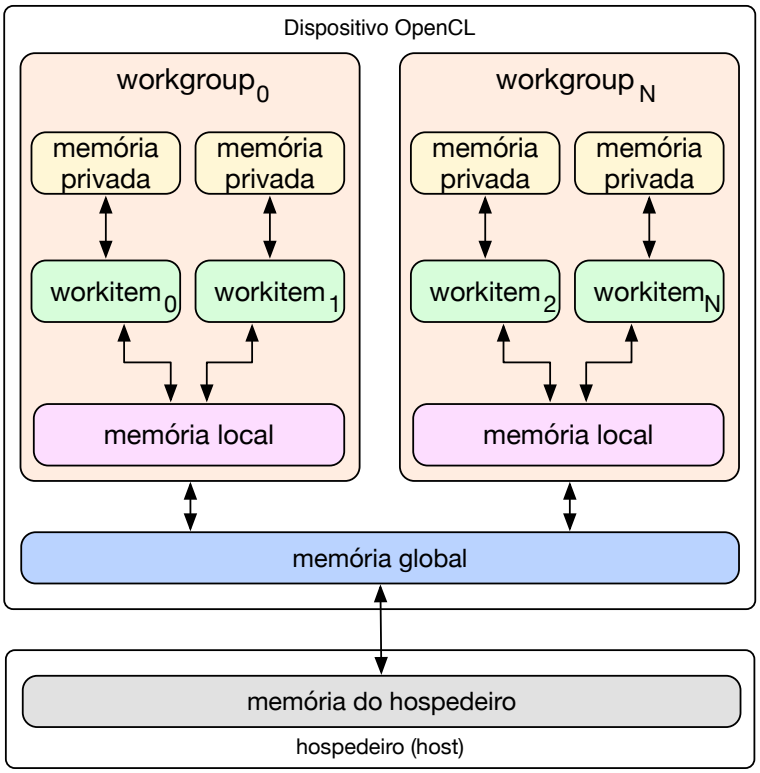


Figura 2.4: Organização de tarefas e acesso à memória de um sistema OpenCL.

2.2.1 OpenCL em Java

JavaCL[1] é uma biblioteca que encapsula as chamadas do padrão OpenCL através de uma *API* para a linguagem Java [23] e possui algumas implementações paralelas como *FFT*, BLAS. A biblioteca do JavaCL apresenta aos desenvolvedores algumas facilidades como o método *createBestContext* que cria um contexto com o dispositivo que contém a maior quantidade de unidades de trabalho. Além disso a *API* do JavaCL apresenta alguns algoritmos paralelizados, funções matemáticas e trigonométricas.

Dentre as vantagens de utilizar OpenCL com Java , podemos destacar a facilidade de programação em Java e o bom desempenho apresentado já que o *overhead* de chamadas da *API* é minimizado pelo tempo de transferência e de execução do *kernel*, que também ocorre nas demais linguagens que utilizam OpenCL.

As entidades especificadas no padrão OpenCL são mapeadas para objetos do JavaCL e dessa forma são gerenciados pelo coletor de lixo da linguagem Java. Quando os objetos do JavaCL são coletados pelo coletor de lixo os respectivos objetos do OpenCL são liberados. Entretanto esperar pelo coletor de lixo pode levar a exceções e problemas de falta de memória. Para evitar isso pode ser utilizado o método *release* presentes nos objetos da biblioteca do JavaCL. Os desenvolvedores da biblioteca Ja-

[1] `https://github.com/nativelibs4java/JavaCL/tree/master/JavaCL`

vaCL aconselham o uso da chamada *release* manual assim que os objetos do JavaCL não forem mais necessários [24]. A chamada ao método *release*, libera os objetos do OpenCL, associados ao objeto da biblioteca do JavaCL que o invocou.

Além da API do JavaCL existem outras *APIs* para uso do OpenCL em Java como a *Lightweight Java Game Library* (LWJGL)[25] que encapsula as chamadas dos padrões OpenCL e OpenGL, entretando não apresenta tantas abstrações como a *API* do JavaCL, sendo uma biblioteca com funções mais próximas da especificação original dos padrões que encapsula.

Um dos pontos negativos de utilizar a biblioteca JavaCL é com relação a sua compatibilidade com a serializaçao[26] da linguagem Java. O processo de serialização é responsável por transformar o objeto incluindo todas as informaç es de seu estado atual, em um *stream* de *bytes* de forma que esse *stream* possa ser revertido de volta em uma cópia do objeto. Para que um objeto utilize o mecanismo de serialização ele deve implementar a interface *Serializable* que fornece uma maneira básica de serialização. Os objetos encontrados na biblioteca do JavaCL não implementam a interface *Serializable*. Uma das implicações da falta de suporte a serialização é a impossibilidade de transmitir objetos pela rede. Para utilizar a JSucuri em ambientes distribuídos seria necessário utilizar um mecanismo de serialização, preferencialmente algum mais performático que o padrão da linguagem Java. A serialização oferecida pela linguagem conforme Lukasz Opyrchal e Atul Prakash em [**?**] apresenta problemas relacionados a performance e tamanho dos objetos gerados que serão enviados através da rede. Este mesmo artigo propõe soluções para estas limitações. Algumas bibliotecas também foram criadas com o propósito de melhorar o desempenho da serialização da linguagem Java como por exemplo a biblioteca Kyro[2].

Para utilizar GPU em ambientes distribuídos com a JSucuri, o escalonador deveria ser modificado para alocar os nós que fazem uso da GPU em uma mesma máquina, pois o contexto de utilização de uma GPU é válido apenas na mesma máquina não sendo possível por exemplo um nó alocado em uma outra máquina acessar diretamente os *buffers* alocados em uma GPU de outra máquina.

2.2.2 Comunicação com múltiplas filas

As *GPUs* modernas apresentam a possibilidade de execução concorrente de *kernels*. Essa funcionalidade permite que mais de um *kernel* seja executado ao mesmo tempo em uma única *GPU* caso haja recursos disponíveis na mesma. Em *GPUs* da NVidia, é possível utilizar este recursos invocando *kernels* através de distintas *threads* ou diferentes processos. A Figura 2.5 ilustra o funcionamento da execução concorrente

[2] `https://github.com/EsotericSoftware/kryo`

de *kernel* Além da execução concorrente de *kernel* outra funcionalidade presente nas placas de vídeos atuais é a execução de *kernel* concorrente com cópia de dados. Através desta funcionalidade é possível que enquanto um *kernel* está em execução dados possam ser transferidos entre a *GPU* e o *host*. A Figura 2.6 ilustra a execução de *kernel* concorrente com cópia de dados.

Figura 2.5: Execução Concorrente de kernel.

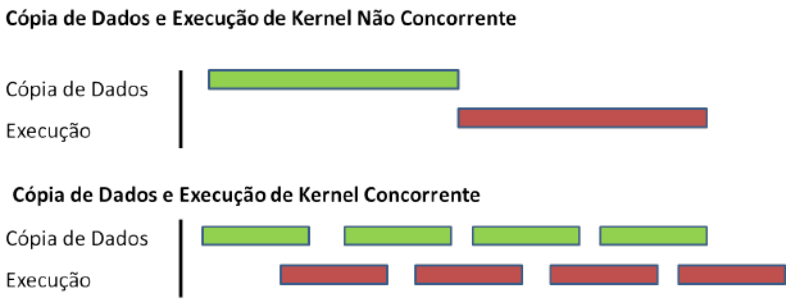


Figura 2.6: Execução de kernel concorrente com cópia de dados.

Em aplicações escritas utilizando OpenCL, todas as interações entre o *host* e os *devices* utilizados são realizadas através de uma fila de comandos. Os comandos suportados pelo OpenCL estão relacionados à execução de *kernel*, operações de transferências de dados, mapeamento de regiões de memória no *host* para execução no *device* e operações de sincronização utilizadas para garantir a ordenação de operações, quando necessário, em uma aplicação.

Uma fila de comandos possui os modos de execução em ordem e fora de ordem. No primeiro, os comandos são completados de acordo com a ordem que foram adicionados à fila, fazendo com que ela trabalhe de forma serial. Quando o modo de execução fora de ordem é executado, os comandos são despachados em ordem, porém de forma assíncrona. Ou seja, após executar uma operação, a próxima operação será acionada sem precisar esperar que a anterior esteja finalizada. Utilizando este modo, todas as operações que tiverem dependências de ordem de execução deverão utilizar os mecanismos de sincronização disponíveis para garantir esta ordenação. Em OpenCL podemos obter o mesmo comportamento de múltiplos processos e *threads*

utilizando mais de uma fila de comandos para uma mesma *GPU*. Filas de comando no OpenCL são responsáveis pela comunicação entre *host* e *devices*. Um fila de comando está relacionada a apenas um *device*, porém um *device* pode ter diversas filas associadas a ele. Com múltiplas filas de comando para um mesmo *device* em operações *stream* utilizamos também a funcionalidade cópia concorrente com execução de *kernel*. Dessa forma enquanto uma das iterações do *stream* está sendo executada os dados de outra iteração podem ser copiados da *GPU* de volta para o *host* e vice-versa. Quando utilizado mais de um fila com operações assíncronas, todas as dependências de ordenação também devem ser explicitadas através dos mecanismos de sincronização.

Entre os mecanismos de sincronização presentes no JavaCL estão os eventos, representados pela classe *CLEvent* [27]. Eventos são os objetos retornadas pelas chamadas assíncronas. Chamadas assíncronas retornam imediatamente após terem após empilhadar seu comando na fila de comandos do *device*, sem esperar o término da operação. Eventos são utilizados para aguardar o término de uma operação através da função *waitFor*. Dessa forma a computação da aplicação pode prosseguir realizando outros passos e quando realmente for necessário que uma determinada operação tenha terminado, pode utilizar o objeto evento da operação desejada para esperar o fim da operação.

2.3 Programação *Dataflow* usando Sucuri

2.3.1 Modelo Dataflow

A execução guiada por fluxo de dados(*Dataflow*), criada por Denis em 1979 [7] representa um modelo no qual programas são descritos como um grafo direcionado. Este grafo é composto por nós, que representam as tarefas e arestas que indicam as dependências de dados entre nós. No modelo *Dataflow* uma vez que os operandos de entrada de um nó estejam disponíveis, ele pode executar. Essa forma de execução aumenta de maneira natural o potencial de paralelismo de uma aplicação, pois a única limitação a execução concorrente são as dependências reais de dados. Para que os operandos sejam considerados disponíveis para um nó executar existem as abordagens: estática proposta por Dennis e Misunas[7] e a abordagem dinâmica ou baseada em tokens[28] [29] [30]. A abordagem estática simplifica a detecção de recebimento dos operandos, pois um nó pode receber apenas um único operando. Esta abordagem possui algumas limitações como a incapacidade de executar paralelamente mais de uma iteração de um *loop*. Na execução dinâmica esta limitação foi superada com a capacidade de recebimento de múltiplos operados por nó e cada um deles contendo um identificador para representar por exemplo a qual instância de

um *loop* o operador pertence[31]. Com a possibilidade de recebimento de múltiplos operandos com diferentes identificadores a regra de disparo foi modificada. Nessa abordagem para que um para que um nó seja considerado pronto para a execução, o mesmo deve ter recebido todos os operandos de uma mesma iteração.

A escrita de programas *Dataflow* segundo Johnston et al em,[32] deveria apresentar algumas características comuns a linguagens funcionais como ausência de efeitos colaterais e atribuição única de variáveis para garantir que as operações dos nós do grafo sejam atômicas e que os resultados obtidos por um mesmo conjunto de inputs seja sempre o mesmo. Além disso esses fatores podem ajudar no ganho de desempenho obtido com o uso do modelo *Dataflow*, evitando *locks* por acessos concorrentes a objetos compartilhados.

2.3.2 Sucuri

A Sucuri [9] é uma biblioteca para programação paralela *Dataflow* em alto nível. Ela permite a descrição de aplicações paralelas em *Pyhton*. Para tanto, um grafo *Dataflow* precisa ser modelado antes de sua execução.

A Sucuri não oferece suporte a execução de *loops* dentro do grafo *Dataflow*. O suporte a *loops* no grafo *Dataflow* envolveria a possibilidade de um nó possuir um seletor para enviar operandos para uma entrada enquanto estivesse dentro do *loop* e para outra ao fim do mesmo.

A Sucuri apresenta alguns nós com funções especiais como:

- Origem(*Source*): Recebe um objeto iterável, percorrendo todos os dados e enviando-os para os próximos nós a cada iteração. Os objetos enviados pelo nó Origem, são encapsulados em objetos do tipo ValorTaggeado. Estes objetos além do valor do operando, possui um campo extra contendo uma tag, que indica a ordenação original dos operandos em relação aos dados de entrada. Essa indicação permite que os operandos sejam processados paralelamente e fora de ordem, mas sendo escritos em ordem no final da execução da aplicação.

- Serializador(*Serializer*): Este nó recebe os operandos encapsulados como ValorTaggeado, fora de ordem e os armazena em um *buffer*, ordenados pelo atributo tag do objeto ValorTaggeado, para que possa ser processado em ordem. A figura 2.7(Reproduzida de: [9]) ilustra a utilização dos nós *Source* e *Serializer* na composição do grafo *Dataflow* de uma aplicação.

- Distribuidor(*Feeder*): Este nó não possui operandos de entrada, servindo como nó inicial do grafo. A função deste nó é repassar o operando recebido para os próximos nós no grafo.

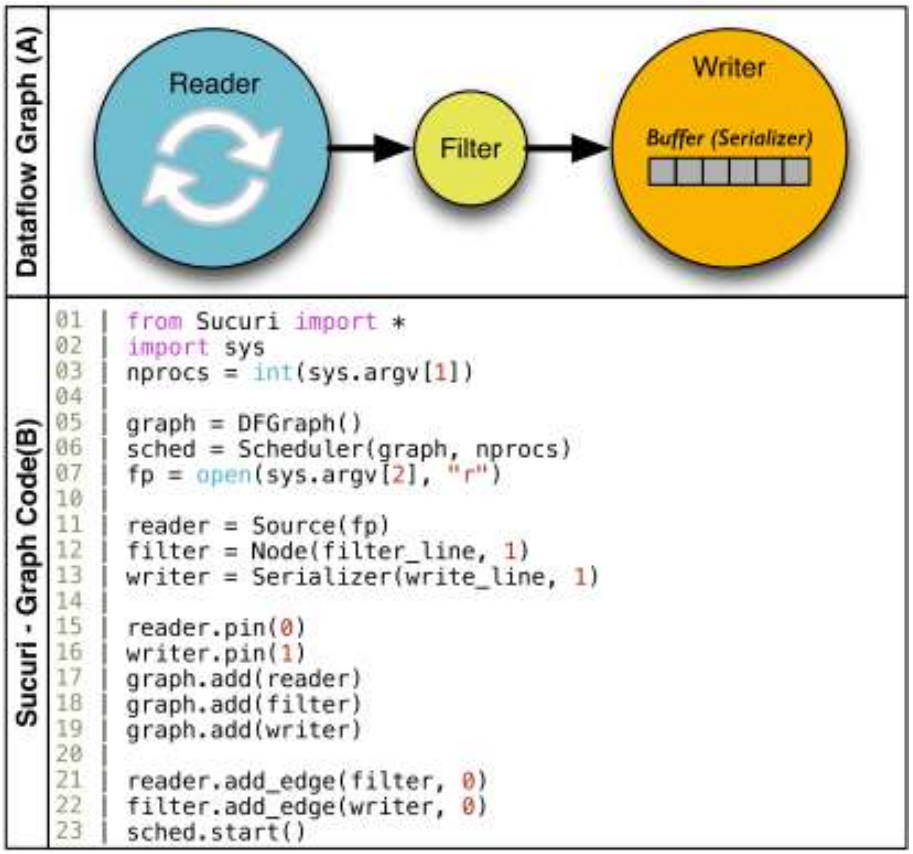


Figura 2.7: Exemplo de aplicação descrita utilizando os nós *Source* e *Serializer*.

Uma das vantagens da Sucuri é possibilidade de estender a execução de uma aplicação *multicore* para *clusters*, somente alterando uma *flag*. Esta *flag* faz com que a Sucuri passe a utilizar mpi para a comunicação entre os nós do grafo.

Como ilustra a Figura 2.8 (Reproduzida de: [9]), somente um escalonador dentre os possíveis *clusters* de nós espalhados por um rede na execução de um programa descrito com a Sucuri, conterá a unidade de *matching* e a fila de prontos. Este escalonador é responsável por receber os operandos de seus *workers* locais e dos escalonadores dos outros *clusters* de nós, gerar as tarefas e colocá-las na fila de prontos. Os escalonadores dos demais *clusters* de nós apenas encaminham as tarefas recebidas para seus *workers* locais

Apesar do escalonador centralizado ser um fator limitante com relação ao desempenho que poderia ser atingido, os experimentos realizados com a Sucuri, mostram uma performance próxima a implementação em *c* com *mpi*. Dessa forma Tiago et al em [9] mostraram que é possível utilizar programação *Dataflow* para paralelizar aplicações com linguagens de alto nível e ter um desempenho satisfatório.

Um dos possíveis pontos de extensão da Sucuri, que foi abordado neste trabalho, é o uso de *GPUs* integrado ao grafo *Dataflow*. Para esta adição escolhemos a biblioteca PyOpenCL[33]. Esta biblioteca encapsula as chamadas do OpenCL, permitindo seu uso em aplicações escritas na linguagem *Python*. Para maximizar o desempe-

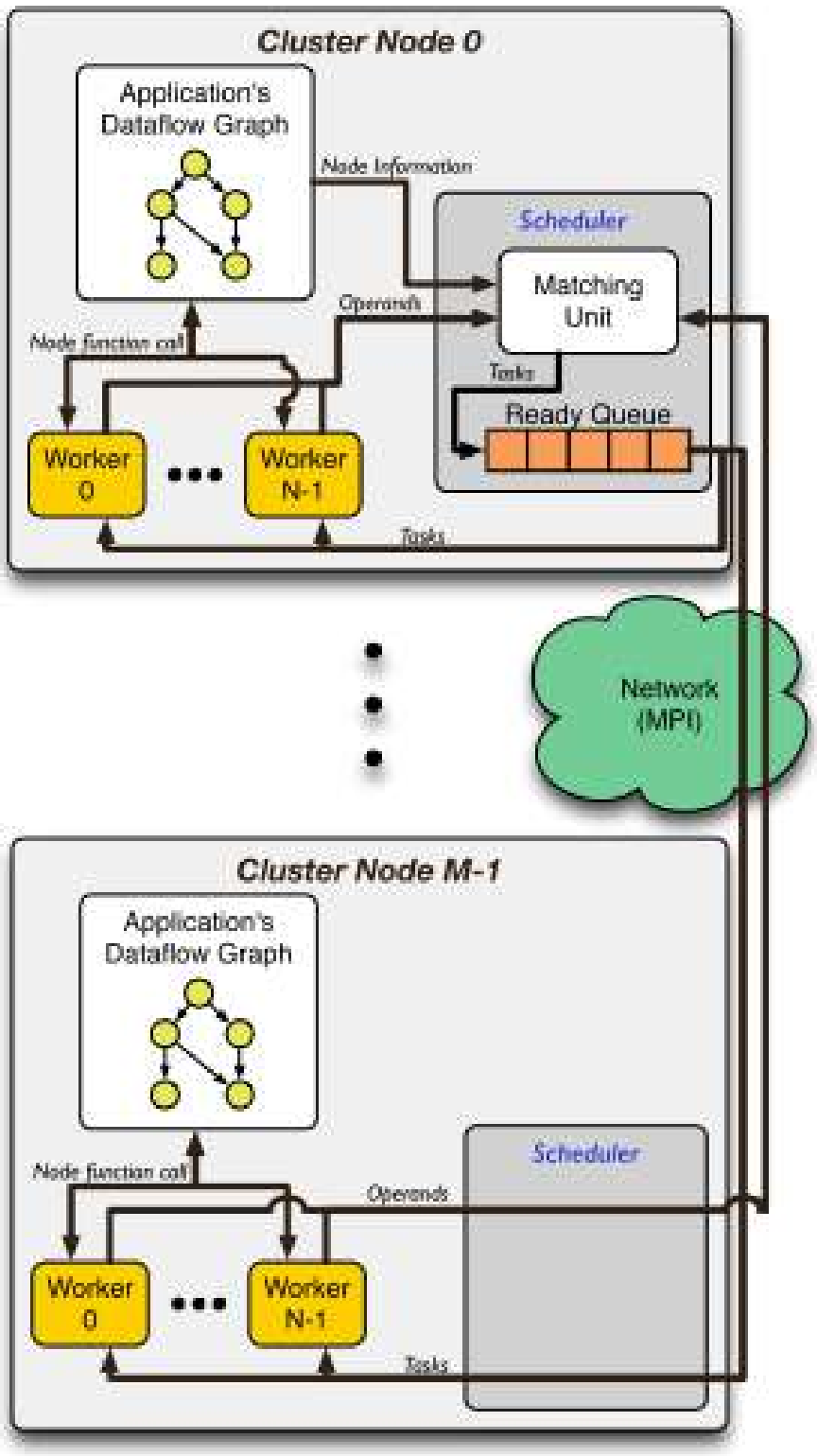


Figura 2.8: Arquitetura da Sucuri.

nho e explorar o uso da comunicação assíncrona tentamos criar novos nós da Sucuri com as funções de cópia dos operandos de entrada, execução do *kernel* e cópia dos operandos de saída da *GPU* para o *host*. Para aumentar o paralelismo disponível

para o escalonador, todos esses nós iriam realizar chamadas assíncronas e retornar o objeto evento correspondente para que o nó destino pudesse aguardar o fim da operação se necessário. Entretanto como os objetos da biblioteca de PyOpenCL, não são compatíveis com o *pickle*, modelo de serialização utilizado pela linguagem Python. A serialização era necessária pois é utilizada pela biblioteca de *multiprocessing*, responsável na implementação da Sucuri, pela troca de operando entre os nós. Para retirar a serialização, poderia ser utilizado memória compartilhada para a troca de operandos entre os nós. Essa alternativa foi descartada pois a linguagem Python utiliza um mecanismo de sincronização global, denominado *Global Interpreter Lock*(GIL)[34], que limita a execução de uma aplicação a apenas uma *thread* por vez reduzindo dessa forma o paralelismo da aplicação. Essa limitação foi a motivação para a migração deste trabalho para a plataforma Java, que utiliza *threads* de forma concorrente sem a limitação do GIL e apresenta facilidades para o uso de memória compartilhada. Para uso das funcionalidades do OpenCL em aplicações Java foi utilizada a biblioteca do JavaCL, descrita na seção 2.2.1. As modificações para utilização da linguagem Java e adição do suporte a *GPU* na Sucuri versão Java(JSucuri) serão descritas posteriormente neste trabalho na seção 3.1.

2.4 Trabalhos Relacionados

Além da Sucuri descrita na seção 2.3.2, outros trabalhos como Stream Jit [35] buscaram utilizar o modelo *Dataflow* em linguagens de alto nível. Stream Jit apresentou uma API para permitir otimizações na geração de código para máquina virtual Java. Entretanto estes trabalhos não suportavam o uso de *GPus* e portanto também não utilizavam comunicação assíncrona e outras características das *GPUs* modernas que foram utilizadas nesse trabalho visando um maior desempenho.

O uso de comunicação assíncrona foi utilizado nos trabalhos *rCUDA* [36] e Distributed CL[37] para redução do *overhead* de comunicação em ambientes distribuídos.

rCUDA é o mais moderno *framework* de virtualização de *GPUs* remoto. Foi criado pelo grupo *Parallel Architectures Group* associado a Universitat Politècnica de València, liderado por Federico Silla. O *framework* rCUDA permite utilizar *GPUs* remotamente realizando um serviço de virtualização de *GPU* em *clusters*. Dessa forma um ou apenas alguns nós possuem as *GPUs* e todos os demais nós do *cluster* podem acessá-las de forma transparente e sem necessidade de modificação no código já que ela possui uma biblioteca dinâmica que substitui as chamadas originais da linguagem CUDA[38]. Suporta a versão corrente de CUDA(8) sem as funções gráficas. Possui suporte a RDMA(*Remote Direct Memory Access*) através de redes infiniband e tcp/ip. O *overhead* de comunicação diminui com o aumento da computação. Conforme descrito em [39] ocorreu um aumento na vazão quando

utilizado operações assíncronas. Que também permitiram a sobreposição de cópia de dados para *GPU* com comunicação via rede. A comunicação assíncrona remota é realizada por *thread* dedicada.

O framework Distributed CL[37] utilizou a *API* existente do OpenCL para possibilitar seu uso em ambientes de processamento distribuído sem modificação de código. Este framework cria a abstração de uma única plataforma OpenCL e através de uma biblioteca dinâmica realiza a tradução e distribuição de objetos entre os diversos dispositivos. Este trabalho assume que as aplicações que o utilizam podem realizar comunicação assíncrona. Utilizando comunicação assíncrona os dados são enviados enquanto simultaneamente os comandos para a execução do *kernel* são sendo executados. O uso de comunicação assíncrona juntamente com o armazenamento de vários comandos antes do envio pela rede são apontados como responsáveis pelo redução no *overhead* de comunicação.

Tanto rCUDA como DistributedCL utilizaram comunicação assíncrona para obter redução do *overhead* de comunicação em ambientes distribuídos. Neste trabalho foi explorado o uso de comunicação assíncrona em conjunto com o modelo *Dataflow* em um ambiente *multicore* heterogêneo para aumentar o paralelismo do grafo *Dataflow* se aproveitando das funcionalidades de execução concorrente de *kernel* e execução de *kernel* concorrente com cópia de dados.

Como vimos na seção 2.2.2, as *GPus* modernas possuem funcionalidades como execução concorrente de *kernel* e execução de *kernel* com concorrência de dados. A execução concorrente de *kernel* pode gerar um aumento na vazão de programas que possam se beneficiar desta característica. Dentre os trabalhos que exploraram este recurso está o *Kernelmerge* [40] criado por Chris Gregg et al. *KernelMerge* é um escalonador de *kernels* concorrente. O escalonador do *KernelMerge* executa 2 *kernels* do OpenCL concorrentemente e mostra que para alguns conjuntos de *kernel* ocorre um aumento na vazão da aplicação enquanto em outros ocorrem uma perda de desempenho. No trabalho do *KernelMerge* foi explorado a execução de 2 *kernels* diferentes e como os *kernels* executados concorrentemente influenciam o uso de recursos da *GPU* e o desempenho, além da sugestão de mudanças na arquitetura das *GPUs* para aumentar o paralelismo em futuros escalonadores. No trabalho desta dissertação foi explorada a execução concorrente de um mesmo *kernel*, variando de 1 a 6 *kernels* para verificar se existia aumento na vazão da aplicação com a execução concorrente de *kernels*. No trabalho desta dissertação cada *kernel* representa uma iteração de uma aplicação *stream* sendo a aplicação descrita utilizando o modelo *Dataflow*. Dessa forma por se tratar de uma aplicação com várias iterações, além da execução concorrente de *kernel* , foi utilizado o recurso de execução de *kernel* concorrente com cópia de dados em operações assíncronas. O trabalho desta dissertação também mostrou como o aumento no número de iterações de uma aplicação *stream*

influencia no desempenho.

Capítulo 3

Sucuri Java para Stream Processing

Este capítulo se inicia com a descrição da modificação da Sucuri original para a linguagem Java. Em seguida serão detalhadas as adições que foram realizadas neste trabalho para criação de aplicações *stream processing* e como os grafos *Dataflow* destas aplicações foram implementados para utilizar *GPU* e explorar suas características de comunicação assíncrona, execução concorrente de *kernel* e execução de *kernel* concorrente com cópia de dados.

3.1 Arquitetura

Conforme descrito na seção 2.3.2, no decorrer deste trabalho se mostrou necessário realizar uma adaptação da Sucuri, originalmente escrita na linguagem Python, para linguagem Java. Essa mudança resultou na criação da JSucuri, mantendo todas as características da versão original em Python com exceção ao suporte para execução em *clusters*. Inicialmente na JSucuri foi feito uso de expressões *lambda* [41] para definir as funções a serem executadas pelos nós da mesma forma que era realizado na Sucuri original. Expressões *lambda* permitem a passagem de funções como argumentos de métodos e foi implementada apenas na versão 8 da linguagem Java. Entretanto após analisar os resultados observados na seção 4.1, verificamos que a versão atual da JVM não estava apresentando um desempenho escalável com o aumento do número de *threads* e por isso optamos então pela utilização da JVM 1.6. A decisão de utilizar a JVM 1.6 acarretou na impossibilidade do uso de versões posteriores a versão 6 da linguagem Java, o que impossibilitou a utilização de expressões *lambda*. No lugar das expressões *lambda*, utilizamos classes anônimas[42] para simular a passagem de funções como argumento para a classe *Node*.

A princípio na JSucuri utilizamos o mecanismo de serialização da linguagem Java

para troca de operandos entre os nós. Embora a serialização garantisse que cada nó recebesse uma nova cópia dos operandos de entrada, um dos princípios desejados por programas *Dataflow* descrito por Johnston et al em [32], ela também foi o grande gargalo encontrado nos testes iniciais. Além desse fato, também verificamos que os objetos da biblioteca JavaCL não possuem compatibilidade com a serialização da linguagem Java. Por esses motivos retirar a serialização da JSucuri e utilizar memória compartilhada se mostrou a melhor opção.

Para aumentar o potencial de paralelismo da aplicação, as operações de cópia de dados para a *GPU*, execução do *kernel* e cópia dos resultados para o *host*, foram separadas em diferentes nós, sendo cada um responsável por uma dessas funções. Conforme descrito na seção 2.4 as operações entre o *host* e a *GPU* no padrão OpenCL, são feitas através de filas. Cada nó que faz uso de operações envolvendo a *GPU* é responsável por enfileirar um comando na fila da *GPU*. Esses comandos são enfileirados de maneira assíncrona, para que um nó termine sua execução sem a necessidade de aguardar o término da operação do comando que adicionou a fila. Como as operações relacionadas a *GPU* são feita de forma assíncrona, os nós que estão envolvidos com essas operações devem utilizar os eventos da API JavaCL (descritos na Seção 2.4) para assegurar que os dados que necessitam já estão disponíveis.

A Figura 3.1 ilustra o comportamento padrão de uma aplicação que faz uso da *GPU*, descrita como um grafo *Dataflow*. Podemos observar que os nós que estão envolvidos diretamente com operações ligadas ao uso da *GPU* são: copiarImagemParaGpu, executarKernel, copiarImagemParaHost. Além disso o nó escreverImagem está relacionado indiretamente por precisar aguardar o fim da cópia de dados iniciada pelo nó copiarImagemParaHost para que possa escrever a imagem de saída quando os dados estiverem disponíveis. A relação entre nós é ilustrada na Figura 3.1.

O primeiro destes novos nós (nó denominado copiarImagemParaGPU na Figura 3.1) recebe os dados de entrada da aplicação lidos pelo nó lerImagem e o objeto de memória da biblioteca JavaCL, para onde os dados de entrada serão copiados. Os objetos de memória da biblioteca do JavaCL são recebidos como parâmetro de entrada pelos nós, para que seja feito o reuso de *buffers*. O nó copiarImagemParaGPU empilha o comando de cópia dos dados de entrada para o *buffer* recebido e envia para o nó executarKernel o *buffer* para onde os dados serão copiados e o respectivo evento que informam a conclusão da cópia. Para facilitar o envio dos objetos de memória e seus eventos foi criada uma nova classe na biblioteca JSucuri que encapsula estes dados.

O nó executarKernel, recebe através do nó copiarImagemParaGPU o *buffer* para onde os dados de entrada serão copiados, o *buffer* para onde os dados de saída serão copiados e o código do *kernel* a ser executado. O nó executar kernel é responsável por

(i) compilar o código do *kernel*. (ii) definir os tamanhos dos grupos de trabalho e o número de items de trabalho em cada grupo, (iii) setar os argumentos do *kernel*, (iv) empilhar a chamada de execução do *kernel* e (v) enviar os objetos de memória que conterão os resultados da computação, juntamente com o evento que será utilizado para determinar se a execução do *kernel* terminou. Antes de empilhar o comando de execução do *kernel* o nó executarKernel aguarda os eventos de cópia inicializados pelo nó anterior(copiarImagemParaGPU) para garantir que os dados necessários para computação do *kernel* já estarão disponíveis para que o *kernel* possa realizar sua computação. Em seguida, ele envia para o nó copiarImagemParaHost o evento retornado pela chamada a invocação do *kernel* e um objeto *buffer*. Este *buffer* conterá o resultado da computação realizado pelo *kernel*.

Por fim, o último dos novos nó (denominado copiarImagemParaHost na Figura 3.1) recebe os objetos de memória que devem ser copiados para o *host* com os resultados da computação. Antes de empilhar o comando de cópia dos resultados, o nó copiarImagemParaHost deve utilizar o evento enviado pelo nó executarKernel para se certificar que a execução do *kernel* terminou. Após empilhar o comando de cópia, este nó envia o objeto para onde os dados serão copiados juntamente com o evento que indica o fim da cópia para o nó escreverImagem

Figura 3.1: Grafo Dataflow com os novos nós responsáveis pelas operações OpenCL

Como descrito na seção 2.2.1, os objetos do JavaCL não são compatíveis com o mecanismo de serialização utilizado pela linguagem Java. Esta incompatibilidade foi um dos fatores que levaram a retirada da serialização e a utilização de memória compartilhada na implementação deste trabalho, para que fosse possível separar as operações envolvendo a *GPU* em diferentes nós e realizar o envio de seus objetos entre os nós do grafo da aplicação. A separação das operações em diferentes nós juntamente com a comunicação assíncrona aumenta o potencial de paralelismo da aplicação pois expõe mais nós do grafo *Dataflow*, que podem potencialmente serem executados em paralelo. Além deste aumento do potencial de paralelismo a retirada da serialização na cópia de dados também ocasionou um ganho de desempenho. Como veremos na próxima seção, algumas aplicações possuem um fluxo contínuo de entradas de dados, onde o grafo 3.1 é repetido para cada uma das iterações do fluxo de dados de entrada. Este tipo de processamento também é conhecido como *stream processing*.

3.2 Utilizando Múltiplas filas, execução concorrente de kernel para Stream Processing

Aplicações *stream* apresentam um fluxo regular de dados de entrada, onde cada entrada corresponde a uma iteração. Quando as iterações são independentes podemos executar mais de uma iteração ao mesmo. Cada iteração do fluxo de dados de entrada pode ser modelada como um grafo *Dataflow*. Neste trabalho utilizamos o modelo *Dataflow*, variando o número de iterações executadas simultaneamente, para explorar como a execução de iterações simultâneas influenciam o desempenho. Nas *GPUs* modernas, utilizando o padrão OpenCL com mais de uma fila para o mesmo *device*, podemos executar iterações em paralelo, fazendo uso dos recursos de execução concorrente de *kernel* e cópia de dados concorrente com execução de *kernel*.

O grafo *Dataflow* na JSucuri é criado por completo antes do início da execução. Dessa forma a aplicação *stream* que temos na verdade é um grande grafo *Dataflow* composto por subgrafos que representam uma iteração. No decorrer da execução da aplicação temos várias iterações simultâneas em execução, cada uma utilizando sua própria fila para se comunicar com a *GPU*.

Utilizando o exemplo da figura 3.1 podemos observar que o nó de leitura dos dados de entrada de uma iteração não possui nenhuma dependência de dados podendo então ser executado assim que a aplicação inicia. Dessa forma ao iniciar uma aplicação a leitura dos dados de entrada de todas as iterações poderiam potencialmente ocorrer em paralelo de acordo com a disponibilidade de recursos. Entretanto devido a limitações na quantidade de memória disponível e número de *devices* para execução dos comandos de *kernel* do OpenCL, esse comportamento pode gerar explosão de paralelismo, ocupando por exemplo toda a memória disponível com os dados de entrada. Para evitar que isso ocorresse foram adicionadas dependências aos nós de leitura para limitar o número de operações de leitura simultâneas. Essa dependência foi adicionada entre a escrita dos resultados do *kernel* de uma iteração com a leitura de dados da próxima iteração de um mesmo ramo. Essa aresta está destacada em laranja na Figura 3.2. As primeiras arestas de um ramo que não possui uma iteração anterior, recebem um *token* apenas, para que possam começar a computação.

Executar várias iterações ao mesmo tempo expõe ao escalonador mais nós que podem potencialmente ser executados em paralelos. Como as operações que envolvem a *GPU* são realizadas de forma assíncrona, um nó termina sua execução sem precisar esperar o final do comando que empilhou na fila da *GPU*. Com isso um nó acaba sua execução enquanto a operação que empilhou está ocorrendo na *GPU*. Dessa forma temos uma sobreposição de comunicação entre nós do grafo e de com-

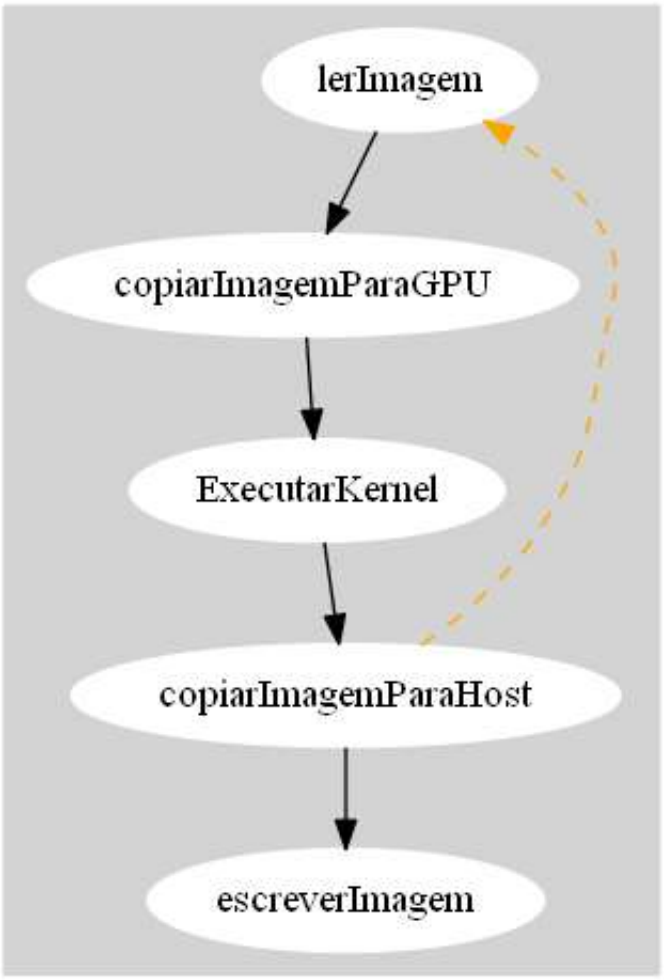


Figura 3.2: Ilustração da Dependência Adicional no Grafo para evitar Explosão de paralelismo.

putação na *GPU*. E se utilizando das funções de execução concorrente de *kernel* e execução de *kernel* concorrente com cópia de dados podemos executar o *kernel* de mais de uma iteração simultaneamente, ou enquanto uma iteração está realizando cópia de dados, outra pode estar executando seu *kernel*. Conforme veremos na seção 4.3, mesmo utilizando apenas um *device*, explorando o uso de comunicação assíncrona, execução concorrente de *kernel* e execução de *kernel* concorrente com cópia de dados é possível obter ganhos de desempenho.

Capítulo 4

Resultados Experimentais

Para avaliação deste trabalho foram executados experimentos em três relevantes aplicações de fluxo de dados (*stream*), visando avaliar como o número de iterações sendo executadas simultaneamente e em conjunto com a utilização de comunicação assíncrona influencia no desempenho das aplicações.

Aplicações *stream* são comuns no contexto de processamento sinais, áudio, vídeo e representam uma classe de significativa importância em computações de alto desempenho [43]. Aplicações *stream* possuem um fluxo constante de dados de entrada podendo ser ilimitado em algumas aplicações. Cada uma das iterações do fluxo de entrada pode ser dividida em diferentes estágios de processamento. Alguns destes estágios não possuem dependências podendo ser executados paralelamente e em muitos casos as diferentes iterações do fluxo de entrada são totalmente independentes entre si. Essas características tornam esse tipo de aplicação aderente à arquiteturas paralelas e heterogêneas. Além disso, aplicações *stream* se adéquam ao modelo *Dataflow*, pois sua computação é descrita através dos diferentes estágios de cada iteração que juntos formam um grafo que representa o fluxo de dados da aplicação. De maneira semelhante ao modo como são descritas e executadas as aplicações no modelo *Dataflow*, já que as aplicações descritas utilizando este modelo, são executadas com base no fluxo de dados e suas dependências. O modelo *Dataflow* por essência, permite que os nós do grafo que já receberam todas os seus dados de entrada, possam ser executados. Caso mais de um nó esteja apto a ser executado, estes nós podem potencialmente ser executados em paralelo, o que torna o modelo um uma escolha natural para explorar a independência entre os estágios e iterações de aplicações *stream* e extrair mais desempenho.

As aplicações utilizadas neste trabalho quando representadas como um grafo *Dataflow* modelado na JSucuri podem ser enxergadas como um conjunto de subgrafos contendo o fluxo padrão de uma programa OpenCL que utiliza *GPU* já que a JSucuri não permite a criação de loops no grafo. A Figura 4.1 ilustra subgrafo que é replicado para cada iteração do fluxo de entrada da aplicação. No caso das

aplicações utilizadas para avaliação cada uma dessas iterações é independente como ilustra a Figura 4.2. Como cada uma das iterações pode ser calculada de forma

imagens/1Iteracao.png

Figura 4.1: Representação simplificada dos estágios de uma iteração de uma aplicação stream padrão

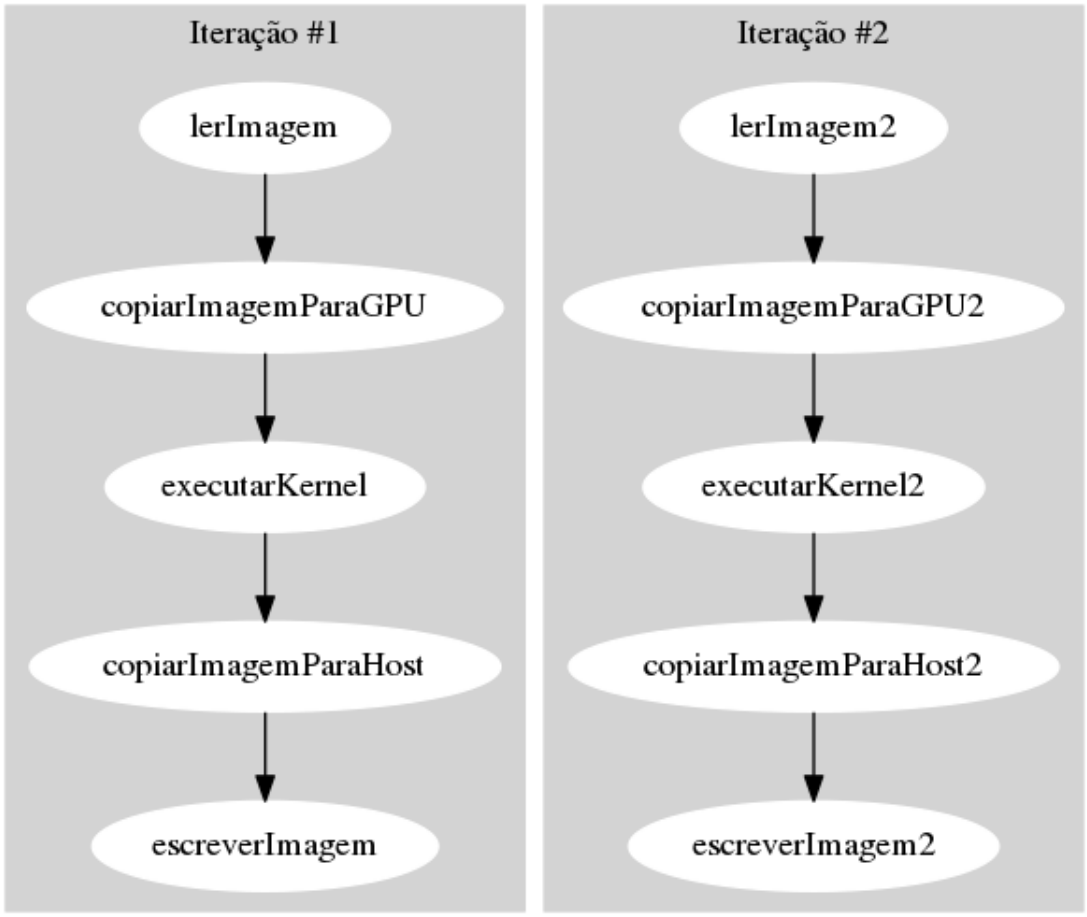


Figura 4.2: Representação da independência entre as diferentes iterações em um exemplo de aplicação stream

independente, neste caso poderíamos executá-las todas em paralelo, não fossem as outras limitações como número de *devices* disponíveis e memória do *host*. Devido as limitações de devices disponíveis, utilizamos para este trabalho um número de execuções simultâneas entre 1 e 6 onde cada uma utiliza sua própria fila de comandos para comunicação com a *GPU*. Com esta variação é possível verificar em cada aplicação, até que ponto o aumento no número de execuções simultâneas se con-

vertia em ganho de desempenho. Cada uma dessas iterações representa um ramo do grafo *Dataflow* da aplicação, onde cada ramo contém um número de subgrafos equivalente a quantidade de iterações que serão realizados naquele ramo. Podemos observar analisando a Figura 4.2 que entre os subgrafos de ramos distintos não existe nenhuma dependência. As dependências só ocorrem entre subgrafos de um mesmo ramo como mostra a Figura 4.3. Esta dependência ocorre pois para evitar a criação de objetos desnecessários, optamos por reutilizar os mesmos objetos de memória em cada uma das iterações de um mesmo ramo. Para utilizar os mesmos objetos nos subgrafos de um ramo, dependências adicionais entre subgrafos de iterações diferentes de um mesmo ramo, tiveram que ser adicionadas ao grafo da aplicação conforme ilustra a Figura 4.4.

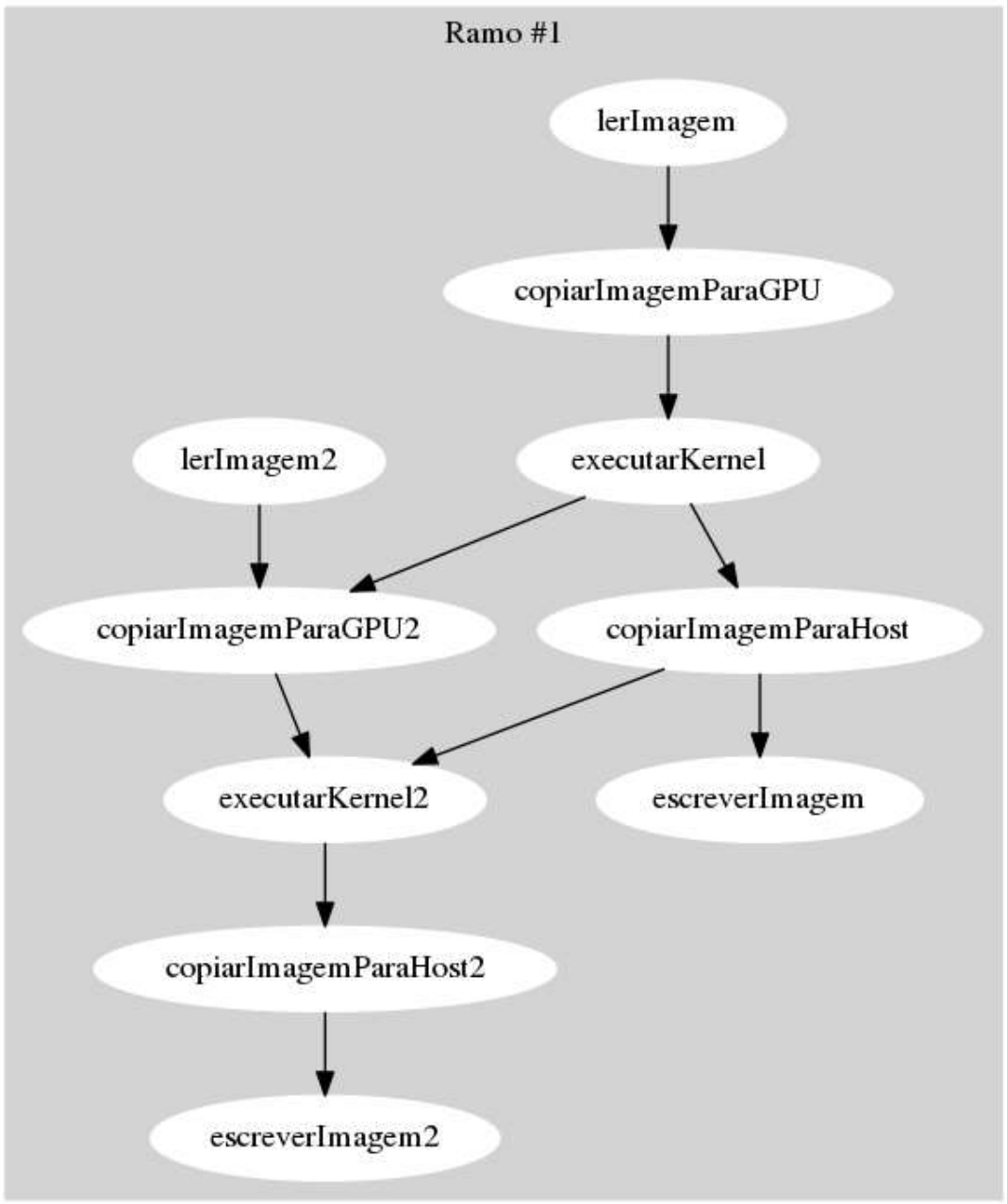


Figura 4.3: Representação da dependência entre as diferentes iterações executadas por um mesmo ramo do grafo

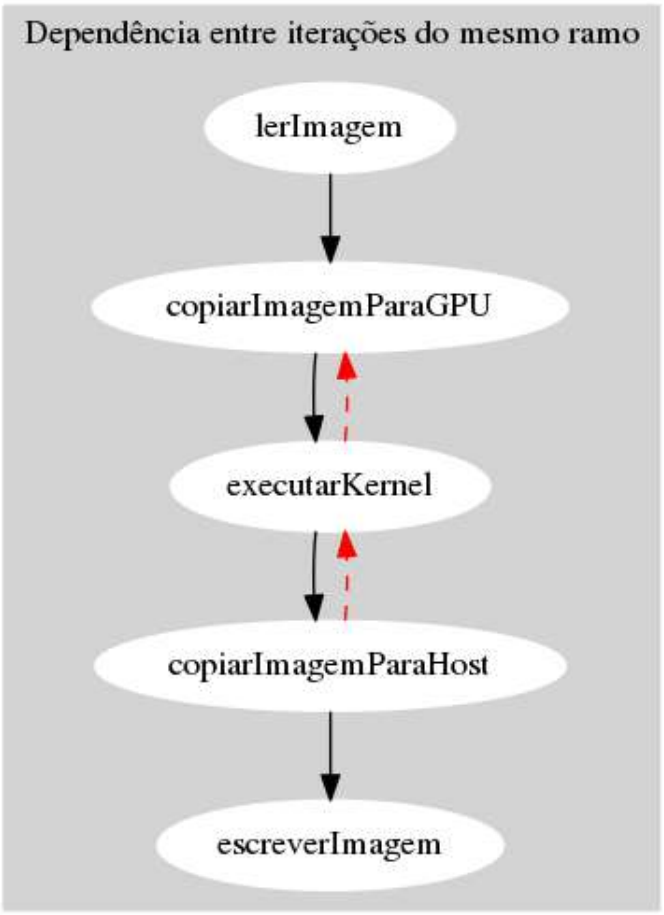


Figura 4.4: Representação simplificada da dependência entre as diferentes iterações executadas por um mesmo ramo do grafo

O subgrafo padrão para aplicações escolhidas para este trabalho é dividido em cinco estágios: (i) leitura/criação dos dados de entrada, (ii) cópia de dados para a *GPU*, (iii) execução do *kernel*, (iv) cópia dos resultados calculados pela *GPU* para o *host* e (v) escrita dos resultados calculados em disco. Cada uma dessas tarefas é realizada por um nó no grafo *Dataflow* da aplicação (ver Figura 3.1). Além das arestas adicionais incluídas para compartilhamento de objetos entre subgrafos de diferentes iterações do mesmo ramo outras arestas foram incluídas para conter a explosão de paralelismo que ocorreria caso as leituras dos dados de entrada de cada iteração continuasse sem dependência, o que tornaria todas as leituras de todas as iterações aptas a execução assim que a aplicação fosse inciada.

As próximas seções deste capítulo abordarão os resultados que motivaram a utilização da JVM 1.6 seguido de como os grafos *Dataflow* das aplicações utilizadas foram construídos e finalizando com os resultados alcançados.

4.1 Versionamento JVM

Após os os problemas encontrados com o mecanismo de serialização da linguagem python descritos na seção 2.3.2, para conseguirmos prosseguir com desenvolvimento deste trabalho optamos pela migração do desenvolvimento para a linguagem Java. Utilizamos então a versão 8 da linguagem Java, a mais atual no inicio do desenvolvimento da Jsucuri. Entretanto ao concluirmos a migração para a linguagem Java e iniciarmos os testes da Jsucuri, utilizando a aplicação *Ray Casting* nos deparamos com um comportamento inesperado. A aplicação *Ray Casting* que possui um alto potencial de paralelismo, pertencente a classe de aplicações embaraçosamente paralelas, não estava apresentando ganhos de desempenho com o aumento do número de *threads*. Diante deste acontecimento, optamos por testar a vesão 1.6 da JVM.

Em nossos testes compratívos utilizando o algoritmo *Ray Casting* nas versões 1.8 e 1.6 da JVM, observamos que a JVM 1.6 apresentou tempos menores e com maior escalabilidade com relação ao aumento no número de *threads*. O algoritmo *Ray Casting* foi executado com 1 amostra e um *grid* imaginário para envolver a cena, cuja as dimensões eram de 256x256x256. Podemos observar nas imagens 4.5, 4.6, 4.7 que o algoritmo *Ray Casting* executado na JVM 1.8 não apresentou ganho de desempenho com o aumento no número de *threads*. O mesmo algoritmo executado na JVM 1.6 por sua vez apresentou redução no tempo de execução com o acréscimo de *threads*. Já na imagem 4.8 com o *Ray Casting* sendo executado na JVM 1.8 observamos que utilizando entre 3 e 11 *threads* o tempo de execução permaneceu próximo variando um pouco para mais ou para menos. Sendo executado na JVM 1.6 por sua vez, a redução do tempo de execução foi constante com o aumento do número de *threads*. Na figura 4.9 observamos que tanto na JVM 1.8 o algoritmo

apresentou pouca variação no tempo de execução com um número de *threads* maior do que 6, enquanto que na JVM 1.6 a redução continuou até 12 *threads* além de apresentar um tempo de execução menor para a maioria do número de *threads*.

Esses resultados foram a motivação para a utilização da JVM 1.6 nos experimentos deste trabalho. Esta mudança acarretou na necessidade de adequação do código da JSucuri para a versão 6 da linguagem Java, conforme descrito na seção 3.1

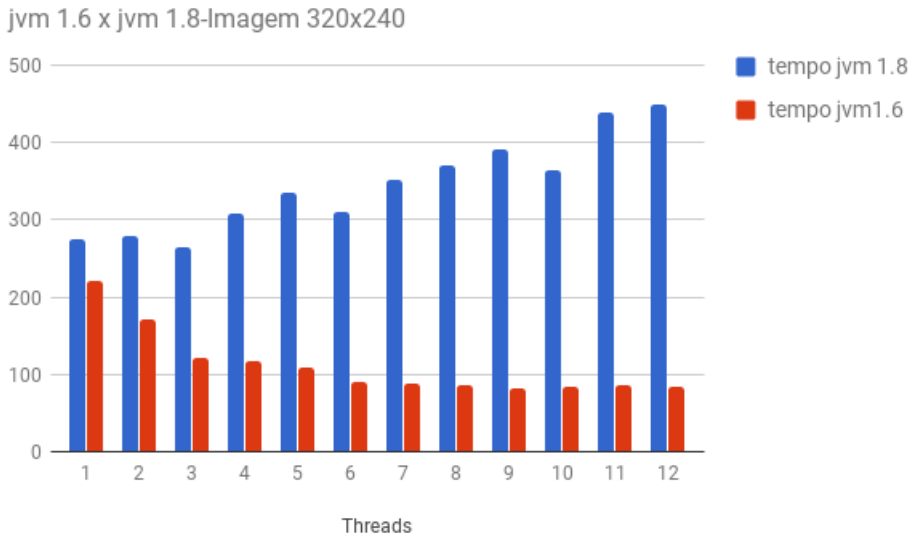


Figura 4.5: Comparativo do tempo de execução entre jvm 1.6 e 1.8 do algoritmo Ray-Casting com resolução de 320x240

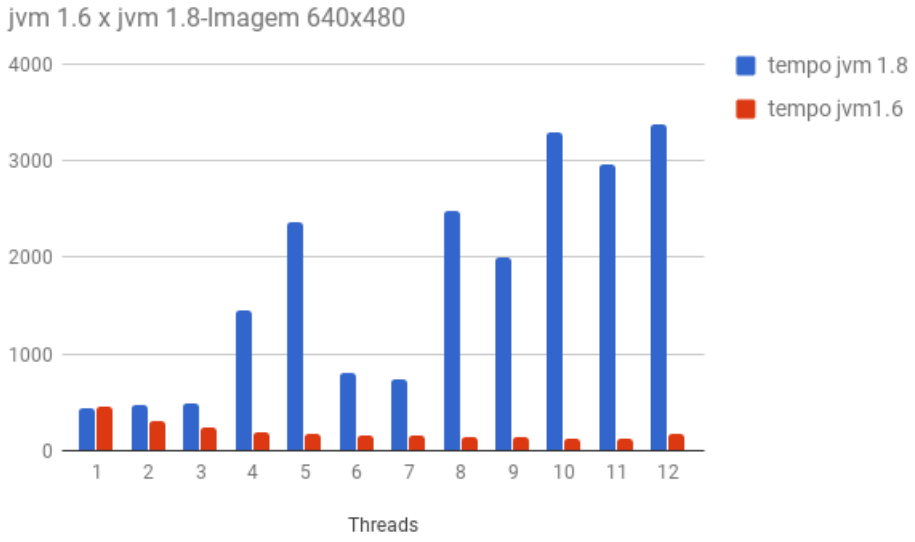


Figura 4.6: Comparativo do tempo de execução entre jvm 1.6 e 1.8 do algoritmo Ray-Casting com resolução de 640x480

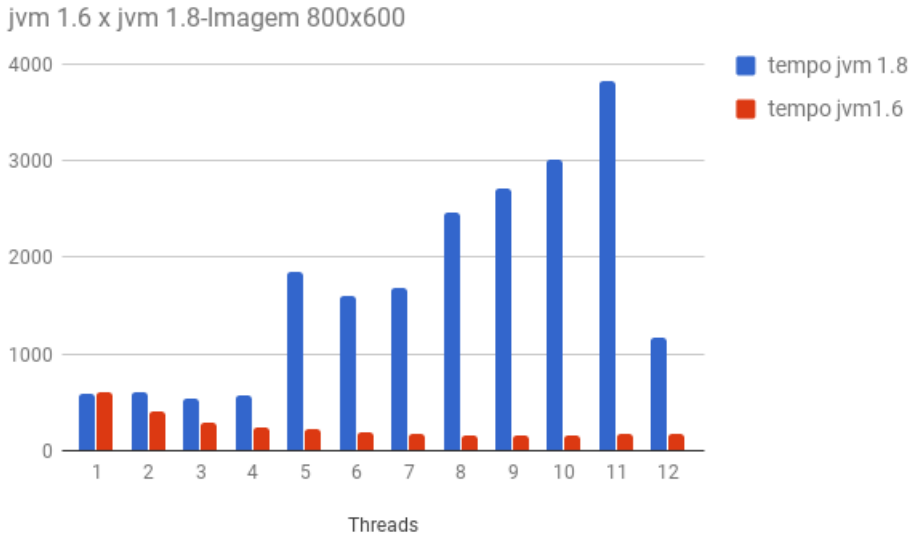


Figura 4.7: Comparativo do tempo de execução entre jvm 1.6 e 1.8 do algoritmo Ray-Casting com resolução de 800x600

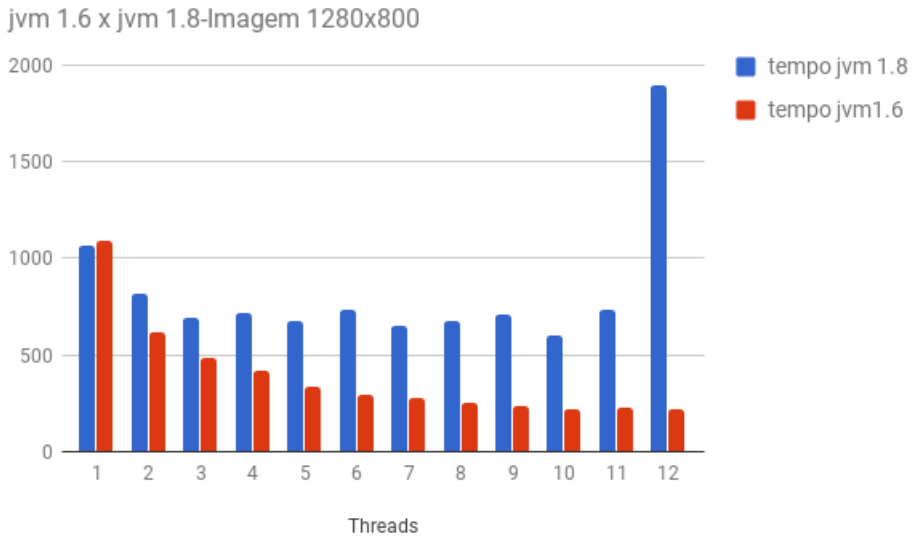


Figura 4.8: Comparativo do tempo de execução entre jvm 1.6 e 1.8 do algoritmo Ray-Casting com resolução de 1280x800

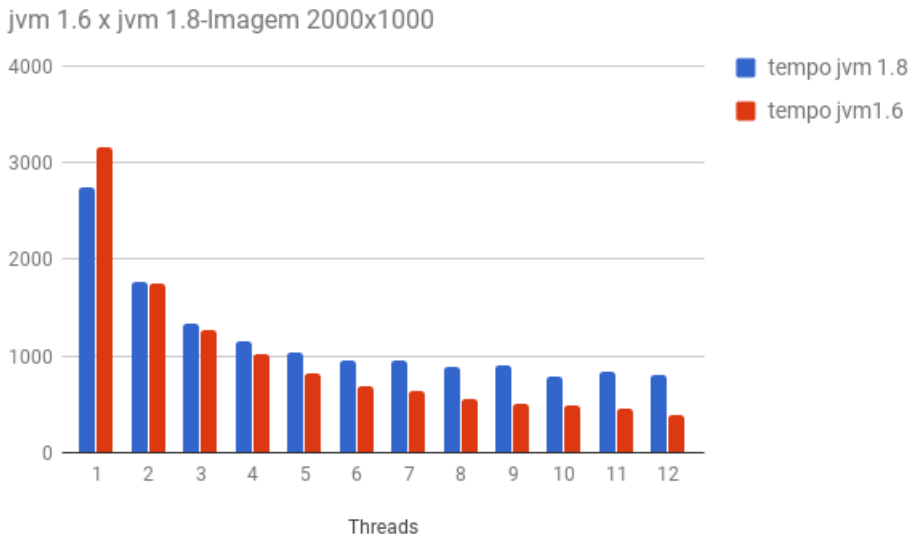


Figura 4.9: Comparativo do tempo de execução entre jvm 1.6 e 1.8 do algoritmo Ray-Casting com resolução de 2000x1000

4.2 Modelagem Dataflow JSucuri

Nesta seção serão apresentados os grafos *Dataflow* modelados na JSucuri para as aplicações utilizadas nos experimentos deste trabalho. Serão detalhados todos os nós, suas arestas incluindo aquelas que existem entre os nós de diferentes iterações, as utilizadas para impedir explosão de paralelismo e as que são utilizadas para permitir o reuso de *buffers*.

4.2.1 Modelagem Ray-Casting

O Grafo *Dataflow* modelado na JSucuri para a aplicação *Ray-Casting* está representado na Figura 4.10 e cada iteração é composta pelos seguintes nós:

- distribuidorArquivoEntrada: Este nó, do tipo Distribuidor, envia o nome da imagem de entrada da iteração atual.
- distribuidorDimensoes: Este nó, do tipo Distribuidor, é responsável pelo envio das dimensões do *grid* imaginário que envolve a cena.
- leitorImagem: Recebe as dimensões da cena através do nó distribuidorDimensoes e a preenche com os valores contidos no arquivo de entrada enviado pelo nó distribuidorArquivoEntrada.
- copiadorImagemParaGPU: Este nó utilizando o objeto fila recebido pelo grafo empilha o comando de cópia dos dados de entrada enviado pelo nó leitorImagem para o *buffer* enviado pelo nó distribuidorBufferEntrada. Este nó deve aguardar a execução do nó executorKernel da iteração anterior deste ramo do grafo. Caso seja a primeira iteração deste ramo, este nó recebe um *token* no lugar do evento da execução do *kernel*. Este nó envia o objeto *buffer* recebido e o evento de cópia a ser aguardado pelo nó executorKernel.
- distribuidorCamera: Este nó do tipo distribuidor é responsável pelo envio do objeto câmera, que representa o ponto de vista através do qual a imagem será renderizada pelo algoritmo *Ray-Casting*.
- distribuidorGrid: Este nó do tipo distribuidor é responsável pelo envio do objeto *grid* imaginário que envolve a cena no algoritmo *Ray-Casting*.
- distribuidorFila: Este nó, do tipo Distribuidor, realiza o envio do objeto fila da biblioteca JavaCL. Cada ramo do grafo utiliza uma instância diferente deste objeto.

- distribuidorContexto: Este nó, do tipo Distribuidor, realiza o envio do objeto contexto da biblioteca JavaCL, só existe uma instância desse objeto na aplicação.

- executorKernel: Este nó cria um objeto do tipo *Kernel*, para o contexto recebido através do nó distribuidorContexto. Esse nó também define como será feita a divisão do trabalho que será realizado pelas *threads* que executam o *kernel*. Nesta implementação do *Ray-Casting* definimos o tamanho do grupo de trabalho global como o resultado da multiplicação da largura da Imagem pela altura da imagem enquanto o tamanho do grupo local foi de 64 *threads*. Os dados de altura e largura são atributos do objeto câmera enviado pelo nó distribuidorCamera. Este nó também recebe pelo grafo os *buffers* de entrada e de saída através dos nós distribuidorBufferEntrada e distribuidorBufferSaida respectivamente. Antes de empilhar o comando de *kernel* na fila recebida através do nó distribuidorFila, este nó aguarda a cópia de dados no *buffer* de entrada utilizando o evento enviado pelo nó copiadorImagemParaGPU. Outro evento que deve ter terminado antes da execução do *kernel* é a cópia dos resultados da execução do *kernel* da iteração anterior deste ramo. Este evento a ser aguardado é enviado pelo nó copiadorRetornoGPU da iteração anterior deste ramo e caso esta seja a primeira execução do ramo, este nó recebe um *token* no lugar deste evento.

- distribuidorNumeroIteracao: Este nó, do tipo Distribuidor, envia o número da iteração ao qual o processamento deste ramo do grafo *Dataflow* pertence.

- distribuidorBufferEntrada: Este nó, do tipo Distribuidor, realiza o envio do objeto *buffer* da biblioteca JavaCL para onde os dados de entrada da aplicação serão copiados. Cada ramo do grafo utiliza uma instância diferente deste objeto para que seja possível explorar as funcionalidades de execução concorrente de *kernel* e execução de *kernel* concorrente com cópia de dados

- copiadorRetornoGPU: Este nó cria um objeto que receberá os dados de saída da computação realizada pelo *kernel*. Este objeto tem um número de elementos correspondente ao resultado da multiplicação da largura da imagem a ser pela altura multiplicado por 4. As dimensões da imagem são uns dos parâmetros de entrada da aplicação. A multiplicação das dimensões da imagem pelo número 4 é devido a representação dos valores RGB da imagem serem definidos separadamente em objetos do tipo ponto flutuante. As dimensões da imagem são atributos contidos no objeto câmera enviado pelo nó distribuidorCamera.

  Utilizando a fila enviada pelo nó distribuidorFila, o nó copiadorRetornoGPU empilha o comando de cópia dos dados para o *buffer* recebido pelo nó distri-

buidorBufferSaida. Este comando é executado após aguardar o fim do evento enviado pelo nó executorKernel, para garantir que os dados de saída só serão copiados após o fim da computação realizada pelo *kernel*.

- escritorImagemSaida: Este nó escreve a imagem de saída correspondente ao número da iteração recebida através do nó distribuidorNumeroIteracao. Esta imagem tem o tamanho definido pelo resultado multiplicação da largura da Imagem pela altura da Imagem. Esses dados são atributos do objeto câmera enviado pelos nó distribuidorCamera. O nó escritorImagemSaida recebe do nó copiadorRetornoGPU os dados a serem escritos e o evento que informa que estes dados já estão disponíveis. Este nó aguarda o fim do evento que informa a completude da cópia dos resultados calculados na *GPU*, recebido pelo nó copiadorRetornoGPU, para escrever o arquivo que contém a imagem de saída.

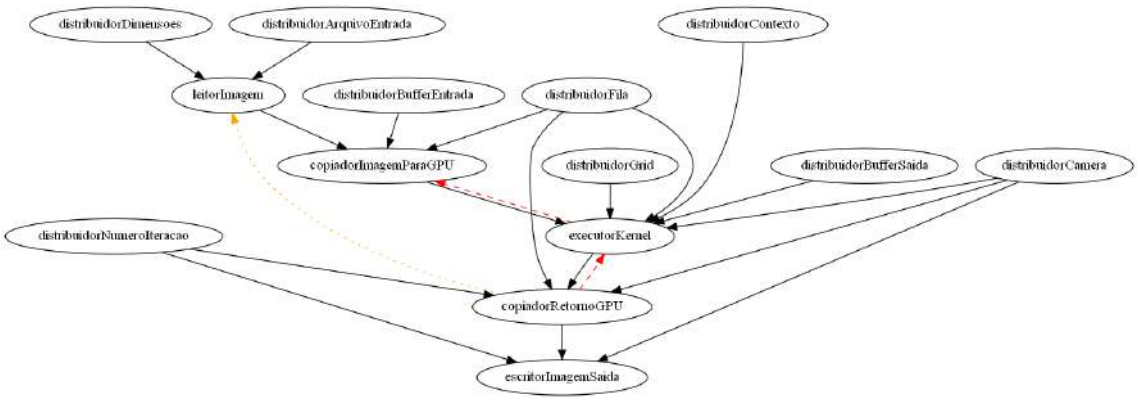


Figura 4.10: Grafo Dataflow modelado na JSucuri para a aplicação Ray-Casting

As arestas descritas em vermelho ilustram as dependências entre diferentes iterações do mesmo ramo do grafo. Estas dependências são necessárias pois as iterações do mesmo ramo do grafo, compartilham os mesmos objetos de memória. A aresta em vermelho entre o nó executorKernel e copiadorImagemParaGPU, garante que uma nova imagem não seja copiada para dentro do *buffer* que está sendo utilizado pelo *kernel* da iteração anterior do mesmo ramo. Já a dependência entre o nó copiadorRetornoGPU e executorKernel, impede que um *kernel* da iteração corrente do ramo comece a ser computado sem que o resultado da iteração anterior deste ramo tenha sido copiado para o *host*. A dependência ilustrada em laranja entre o nó copiadorImagemParaGPU e o nó leitorImagem foi inserida para impedir que muitas leituras ocorrem ao mesmo tempo causando explosão de paralelismo. Sem essa dependência todas as leituras da imagem de entrada de todas as iterações poderiam ocorrer em paralelo, aumentando a criação de objetos, podendo exceder a memória disponível.

Nos experimentos realizados utilizando este algoritmo utilizamos imagens com as resoluções de: (i)640x7320 (ii) 1280x720 e (iii)19820x1080 e 1 amostra por *pixel* . Para cada uma dessas resoluções e números de amostras foram utilizadas de 1 a 6 iterações simultâneas, cada uma com sua própria fila.

4.2.2 Modelagem Path-Tracing

O Grafo *Dataflow* modelado na JSucuri para a aplicação *Path-Tracing* está representado na Figura 4.11 e cada iteração é composta pelos seguintes nós:

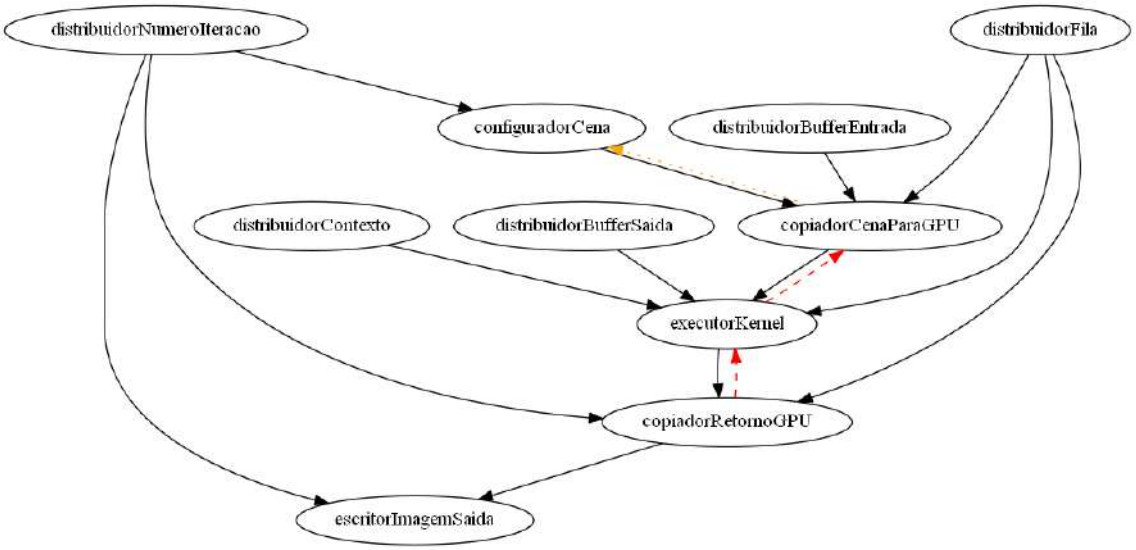


Figura 4.11: Grafo Dataflow modelado na JSucuri para a aplicação Path-Tracing

A função de cada um dos nós desta aplicação esta descrita abaixo:

- configuradorCena: Nó responsável por criar as esferas que compõem a cena. A cena é composta por 9 esferas, representadas no total por 143 pontos flutuantes. A dimensão da imagem que é um dos parâmetros de entrada desse algoritmo, não influencia no número de objetos utilizados para representar a cena.

- distribuidorNumeroIteracao: Este nó, do tipo Distribuidor, envia o número da iteração ao qual o processamento deste ramo do grafo *Dataflow* pertence.

- distribuidorAlturaImagem: Nó do tipo de distribuidor, responsável pelo envio do tamanho da altura da imagem de saída a ser renderizada pelo algoritmo do *Path-Tracing*.

- distribuidorLarguraImagem: Nó do tipo de distribuidor, responsável pelo envio do tamanho da largura da imagem de saída a ser renderizada pelo algoritmo do *Path-Tracing*.

- copiadorCenaParaGPU: Este nó utilizando os objetos contexto e fila recebidos pelo grafo empilha o comando de cópia dos dados da cena. Os dados de entrada da cena são enviados pelo nó configuradorCena para o *buffer* que representa os dados de entrada. Este *buffer* é enviado pelo nó distribuidorBufferEntrada. Este nó deve aguardar a execução do nó executorKernel da iteração anterior deste ramo do grafo. Caso seja a primeira iteração deste ramo, este nó recebe um *token* no lugar do evento da execução do *kernel*. O nó copiadorCenaParaGPU envia o objeto *buffer* recebido e o evento de cópia a ser aguardado pelo nó executorKernel.

- distribuirArquivoKernel: Este nó, do tipo Distribuidor, envia o nome do arquivo que contém o *kernel* que contém o código do algoritmo *Path-Tracing* a ser executado pela *GPU*.

- distribuidorContexto: Este nó, do tipo Distribuidor, realiza o envio do objeto contexto da biblioteca JavaCL, só existe uma instância desse objeto na aplicação.

- distribuidorFila: Este nó, do tipo Distribuidor, realiza o envio do objeto fila da biblioteca JavaCL. Cada ramo do grafo utiliza uma instância diferente deste objeto.

- executorKernel: Este nó cria um objeto do tipo *Kernel* para o contexto, recebido através do nó distribuidorContexto, que irá a executar a função enviada pelo nó distribidorFuncaoKernel. O nó executorKernel compila o código fonte do *kernel* contido no arquivo cujo nome é enviado pelo nó distribuirArquivoKernel. O nó executorKernel também define a divisão do trabalho das *threads* que executarão o *kernel* definindo o tamanho do grupo de trabalho global como o resultado da multiplicação da largura da Imagem de saída pela altura. Esses dados são parâmetros de entrada da aplicação e no grafo são enviados respectivamente pelos nós distribuidorLarguraImagem e distribuidorAlturaImagem. Os dados de altura e largura da imagem também são configurados como parâmetros do *kernel*, juntamente com os *buffers* de entrada e de saída recebidos através dos nós distribuidorBufferEntrada e distribuidorBufferSaida respectivamente. Antes de empilhar o comando de *kernel* na fila recebida através do nó distribuidorFila, este nó aguarda a cópia de dados no *buffer* de entrada utilizando o evento enviado pelo nó copiadorImagemParaGPU. Outro evento que deve ter terminado antes da execução do *kernel* é a cópia dos resultados da execução do *kernel* da iteração anterior deste ramo. Este evento a ser aguardado é enviado pelo nó copiadorRetornoGPU da iteração anterior deste ramo. Caso esta seja a primeira iteração deste ramo ao invés de aguardar o fim

da cópia do resultado da computação do *kernel* de volta para o *host* através do evento enviado pelo nó copiadorRetornoGPU este nó recebe um *token* para que consiga ter todos os parâmetros de entrada necessários para iniciar sua computação, como define o modelo de execução de um nó no modelo *Dataflow*.

- distribuidorBufferEntrada: Este nó, do tipo Distribuidor, realiza o envio do objeto *buffer* da biblioteca JavaCL que receberá os dados de entrada da aplicação. Cada ramo do grafo utiliza uma instância diferente deste objeto.

- distribuidorBufferSaida: Este nó, do tipo Distribuidor, realiza o envio do objeto *buffer* da biblioteca JavaCL para onde os dados de saída do *kernel* serão copiados. Cada ramo do grafo utiliza uma instância diferente deste objeto.

- copiadorRetornoGPU: Este nó cria um objeto que receberá os dados de saída da computação realizada pelo *kernel*. Este objeto tem um número de elementos correspondente ao resultado da multiplicação da largura da imagem pela altura da imagem. Esses dados são enviados respectivamente pelos nós distribuidorLarguraImagem e distribuidorAlturaImagem. Utilizando a fila enviada pelo nó distribuidorFila, o nó copiadorRetornoGPU empilha o comando de cópia dos dados para o *buffer* recebido pelo nó distribuidorBufferSaida. Este comando é executado após aguardar o fim da computação do *kernel* através do evento enviado pelo nó executorKernel, para garantir que os dados de saída só serão copiados após o fim da computação realizada pelo *kernel*.

- escritorImagemSaida: Este nó escreve a imagem de saída correspondente da iteração ao qual este ramo pertence. O número da iteração é recebido através do nó distribuidorNumeroIteracao. Esta imagem tem o tamanho definido pelo resultado multiplicação da largura da imagem pela altura da imagem. Esses dados são enviados respectivamente pelos nós distribuidorLarguraImagem e distribuidorAlturaImagem. O nó escritorImagemSaida recebe do nó copiadorRetornoGPU os dados a serem escritos e o evento que informa que estes dados já estão disponíveis. Utilizando o evento recebido o nó escritorImagemSaida aguarda o fim do evento para escrever a imagem de saída.

Para as avaliações dos resultados deste experimento foram variados os seguintes parâmetros: (i)resolução da imagem, (ii)número de amostras e (iii) números de quadros da animação. Os testes foram realizados variando o número de quadros de animação para cada tamanho de imagem. O número de amostras por *pixel* foi fixado em 2048 para garantir uma qualidade de imagem aceitável. O tamanho da imagem representa a resolução das imagens de saída, alterando consequentemente o número de *pixels* a ser computado pelo algoritmo. Foram utilizadas as seguintes

resoluções: (i) 640 x 320 (ii) 1280 x 720 (iii) 1920 x 1080. O número de quadros da animação utilizado na animação foi de 10 e 20 e o número de filas e iterações computadas simultaneamente variou de 1 a 6. Como este algoritmo não apresentou variação significativa nem com o aumento do número de filas, nem com o aumento do tamanho da imagem de saída como será explicado adiante na seção 4.3, executamos este experimento para menos cenários que os demais.

4.2.3 Modelagem Sobel

O Grafo *Dataflow* modelado na JSucuri para a aplicação *Sobel* está representado na Figura 4.12 e cada iteração é composta pelos seguintes nós:

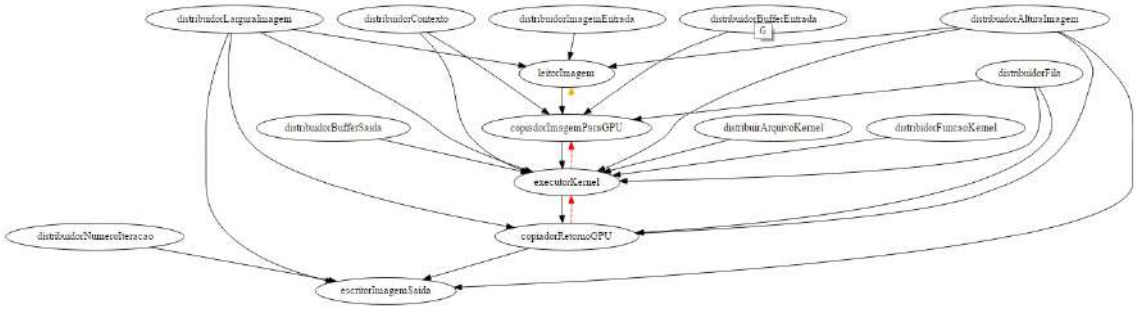


Figura 4.12: Grafo Dataflow modelado na JSucuri para a aplicação Sobel

O Grafo *Dataflow* modelado na JSucuri para esta aplicação, representado na Figura 4.12 contém os seguintes nós:

- distribuidorImagemEntrada: Este nó, do tipo Distribuidor, envia o nome da imagem de entrada da iteração atual.
- distribuidorAlturaImagem: Nó do tipo de distribuidor, responsável pelo envio do tamanho da altura da imagem de saída a ser renderizada pelo algoritmo do *Sobel*.
- distribuidorLarguraImagem: Nó do tipo de distribuidor, responsável pelo envio do tamanho da largura da imagem de saída a ser renderizada pelo algoritmo do *Sobel*.
- distribuidorContexto: Este nó, do tipo Distribuidor, realiza o envio do objeto contexto da biblioteca JavaCL, só existe uma instância desse objeto na aplicação.
- distribuidorFila: Este, do tipo Distribuidor, nó realiza o envio do objeto Fila da biblioteca JavaCL. Cada ramo do grafo utiliza uma instância diferente deste objeto.

- distribuidorBufferEntrada: Este, do tipo Distribuidor, nó realiza o envio do objeto *buffer* da biblioteca JavaCL que receberá os dados de entrada da aplicação. Cada ramo do grafo utiliza uma instância diferente deste objeto.

- leitorImagem: Este nó executa a leitura e configuração dos dados contidos na imagem, enviada pelo nó distribuidorImagemEntrada, para que estejam prontos para serem copiados para *GPU*.

- copiadorImagemParaGPU: Este nó utilizando o objeto contexto e fila recebidos pelo grafo empilha o comando de cópia do dados de entrada enviado pelo nó leitorImagem para o *buffer* que receberá os dados de entrada do kernel. Este *buffer* é enviado pelo nó distribuidorBufferEntrada e seu tamanho varia de acordo com a resolução da imagem de saída a ser computada. O seu tamanho é o resultado da multiplicação da largura da imagem pela resolução multiplicado por 3. Antes que o nó copiadorImagemParaGPU possa empilhar o comando de cópia ele deve aguardar a execução do nó executorKernel da iteração anterior deste ramo do grafo. Essa espera é necessária para que novos dados não sejam copiados enquanto o *kernel* anterior está realizando processamento nos dados de entrada que compartilham o mesmo *buffer* em todas as iterações de um mesmo ramo do grafo. Caso seja a primeira iteração deste ramo, este nó recebe um *token* no lugar do evento da execução do *kernel*. Este nó envia o objeto *buffer* recebido e o evento de cópia a ser aguardado pelo nó executorKernel.

- distribuirArquivoKernel: Este nó, do tipo Distribuidor, envia o nome do arquivo que contém o *kernel* a ser executado pela *GPU*.

- distribidorFuncaoKernel: Este nó, do tipo Distribuidor, envia o nome da função principal do *kernel* a ser invocada pela aplicação.

- distribuidorBufferSaida: Este nó, do tipo Distribuidor, realiza o envio do objeto *buffer* da biblioteca JavaCL para onde os dados de saída do *kernel* serão copiados. Cada ramo do grafo utiliza uma instância diferente deste objeto.

- executorKernel: Este nó cria um objeto *kernel* para o contexto recebido por distribuidorContexto que irá a executar a função enviada pelo nó distribidorFuncaoKernel compilando o código fonte do *kernel* enviado pelo nó distribuirArquivoKernel. Esse nó também define a divisão do trabalho das *threads* que executaram o *kernel* definindo o tamanho do grupo de trabalho global como o resultado da multiplicação da largura da imagem pela altura da imagem. Esses dados são enviados respectivamente pelos nós distribuidorLarguraImagem e distribuidorAlturaImagem. Estes dados de altura e largura da imagem

também são configurados como parâmetros do *kernel*, juntamente com os *buffers* de entrada e de saída recebidos através dos nós distribuidorBufferEntrada e distribuidorBufferSaida respectivamente. Antes de empilhar o comando do *kernel* na fila recebida através do nó distribuidorFila, este nó aguarda a cópia de dados no *buffer* de entrada utilizando o evento enviado pelo nó copiadorImagemParaGPU. Outro evento que deve ter terminado antes da execução do *kernel* é a cópia dos resultados da execução do *kernel* da iteração anterior deste ramo. Este evento a ser aguardado é enviado pelo nó copiadorRetornoGPU da iteração anterior deste ramo.

- copiadorRetornoGPU: Este nó cria um objeto que receberá os dados de saída da computação realizada pelo *kernel*. Este objeto tem um número de elementos correspondente ao resultado da multiplicação da largura da imagem pela altura da imagem multiplicado por 3, pois os valores RGB da cor são copiados separadamente um em cada elemento. Os dados da dimensão da imagem são enviados respectivamente pelos nós distribuidorLarguraImagem e distribuidorAlturaImagem. Utilizando a fila enviada pelo nó distribuidorFila, o nó copiadorRetornoGPU empilha o comando de cópia dos dados para o *buffer* recebido pelo nó distribuidorBufferSaida. Este comando é executado após aguardar o fim da computação do *kernel* da iteração atual aguardando o evento enviado pelo nó executorKernel, para garantir que os dados de saída só serão copiados após o fim da computação realizada pelo *kernel*.

- distribuidorNumeroIteracao: Este nó, do tipo Distribuidor, envia o número da iteração atual.

- escritorImagemSaida: Este nó escreve a imagem de saída correspondente ao número da iteração recebida através do nó distribuidorNumeroIteracao. Esta imagem tem o tamanho definido pelo resultado multiplicação da largura da Imagem pela altura da Imagem. Esses dados são enviados respectivamente pelos nós distribuidorLarguraImagem e distribuidorAlturaImagem. O nó escritorImagemSaida recebe do nó copiadorRetornoGPU os dados a serem escritos e o evento que informa que estes dados já estão disponiveis. Utilizando o evento recebido o nó escritorImagemSaida aguarda o fim do evento para escrever a imagem de saída.

Para a execução dos testes deste algoritmo foram usados como entrada imagens extraídas de vídeos utilizando o *framework* FFmpeg[44]. Foram utilizadas as seguintes resoluções de imagem: (i)640 x 320 (ii)1280 x 720 (iii) 1920 x 1080. A quantidades de quadros de entrada utilizados variaram de 10 a 100, sendo incrementados de 10 em 10. Cada combinação de tamanho de imagem e número de quadros

de entrada foram executados com os números de filas variando de 1 a 6. Nesse algoritmo o a quantidade de dados copiada para a *GPU* em cada iteração é igual ao tamanho da imagem de entrada multiplicado por 4.

4.3 Desempenho & Escalabilidade

Nesta seção iremos apresentar como o desempenho das aplicações executadas nesse trabalho, variaram de acordo com o número de filas simultâneas, número de quadros de entrada e resolução da imagem de saída. Apresentaremos também as causas para tais resultados

As imagens 4.13 e 4.14 criadas a partir dos experimentos deste trabalho executados com auxílio de um *profiling*, ilustram como o número de filas simultâneas influenciam no número de operações que envolvem a *GPU* sendo executadas simultaneamente e consequentemente no tempo de execução com veremos a seguir. Nas imagens abaixo os retângulos verdes representam os *kernels* e as barras amarelas e vermelhas, representam respectivamente as cópias do *host* para e *GPU* e da *GPU* de volta para o *host*.

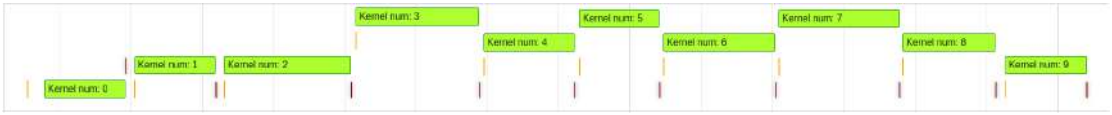


Figura 4.13: O algoritmo Ray-Casting executado com apenas uma fila

Figura 4.14: O algoritmo Ray-Casting executado com 3 filas

Com relação ao *speedup* em comparação com a implementação sequencial, a aplicação *Ray-Casting* foi a que apresentou o melhor desempenho das aplicações utilizadas nesse trabalho, aproximadamente 2745 vezes. Nas imagens 4.15, 4.16 e 4.17 podemos observar que o *speedup* aumentou significativamente com o aumento da resolução da imagem de imagem processada. O *speedup* máximo para a resolução de 640x320 foi de 535 aumentando para 1400 para imagens com resolução de 1280x720 e para imagens de 1920x1280 o maior *speedup* alcançado foi 2700. Para imagens com resolução de 640x320 *pixels* como ilustra a figura 4.15, o número de filas simultâneas que apresentou o melhor *speedup* variou entre 3 e 6 conforme o número de quadros processados, sendo que em alguns casos o desempenho foi bem próximo. Já para

imagens com resolução 1280x720 e 1920x1280 conforme ilustram as figuras 4.16 e 4.17 utilizar 3 filas apresentou o maior *speedup* para a maioria dos casos.

A aplicação do *Path-Tracing* apresentou um *speedup* máximo de aproximadamente 600 vezes. Esse *speedup* não apresentou mudanças significativas com o aumento do número de quadros e da resolução da imagem de saída como ilustram as imagens 4.21, 4.22 e 4.23

Já o algoritmo *Sobel* para imagens com a resolução de 640x320, para a maioria do número de quadros, utilizar 3 filas foi a opção que apresentou o maior ganho de desempenho como podemos observar na Figura 4.18. Entretanto observando o gráfico para imagens com a resolução de 1280x720 4.19 podemos observar que para os cenários de 40 quadros em diante o *speedup* de 4 filas é bem próximo ao atingido com 3 filas, chegando a ser superior em alguns casos como 90 e 100 quadros. Já para imagens de 1920x1080 o uso de 4 filas foi o que apresentou o maior desempenho na maioria dos cenários como ilustra a Figura 4.20.

Com relação a escalabilidade com o aumento no número de filas simultâneas, a aplicação mais se beneficiou do aumento de filas, foi a aplicação do filtro *Sobel* como ilustram as imagens 4.18, 4.19 e 4.20. Esta aplicação como ilustra a imagem 4.20, em alguns casos teve um melhor desempenho utilizando 4 filas simultâneas, sendo que para 100 quadros, o aumento no *speedup* utilizando 4 filas foi 74% maior em comparação a utilizar 1 fila. Uma das características que contribuíram para este resulto foi um maior equilíbrio entre computação e comunicação, pois seu *kernel* não é tão intenso a ponto de superar o tempo de cópia de dados e nem tão leve a ponto da cópia de dados superar o tempo de computação do *kernel*. A aplicação *Ray-Casting* mostrou se beneficiar um pouco menos com relação ao numero de filas simultâneas que apresentou um melhor resultado em comparação ao filtro *Sobel*. A aplicação *Ray-Casting* como ilustram as imagens 4.16 e 4.17 na maioria dos casos apresentou um maior ganho de desempenho utilizando 3 filas simultâneas. Já o algoritmo *Path-Tracing* não apresentou ganhos significativos com o uso de múltiplas filas como podemos observar na Figuras 4.21, 4.22 e 4.23. Seu *kernel* é mais intenso computacionalmente, então os recursos da placa de vídeo permanecem ocupados por mais tempo, além disso sua entrada é bem pequena(um vetor de apenas 143 valores do tipo ponto flutuante), sendo o tempo de computação maior que o da cópia de dados o que diminui o ganho de cópia concorrente com execução do *kernel*.

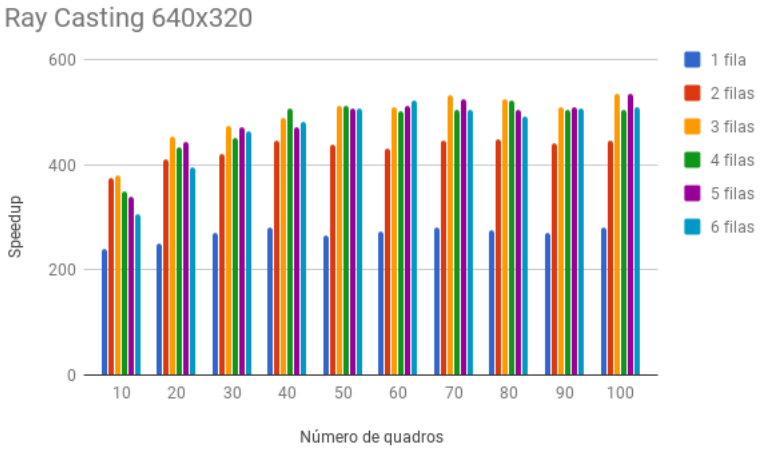


Figura 4.15: Speedup do algoritmo Ray-Casting com relação ao número de filas para imagens de 640x320

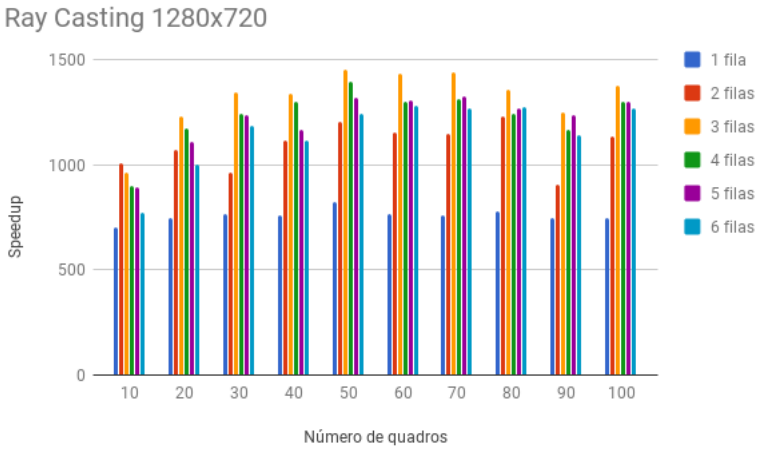


Figura 4.16: Speedup do algoritmo Ray-Casting com relação ao número de filas para imagens de 1280x720

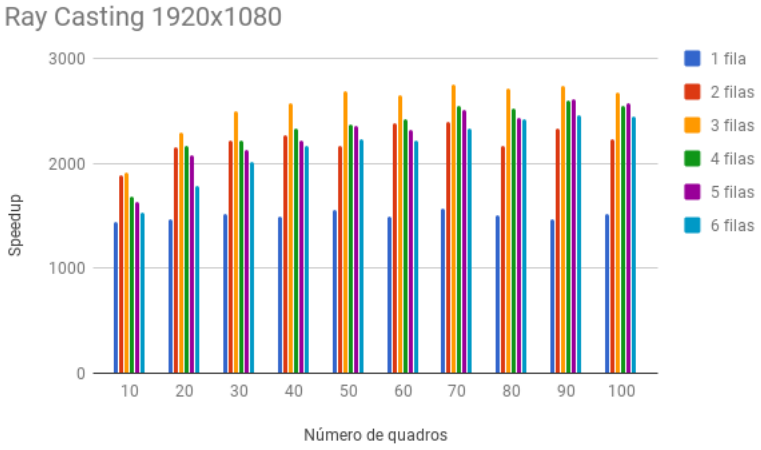


Figura 4.17: Speedup do algoritmo Ray-Casting com relação ao número de filas para imagens de 1920x1080

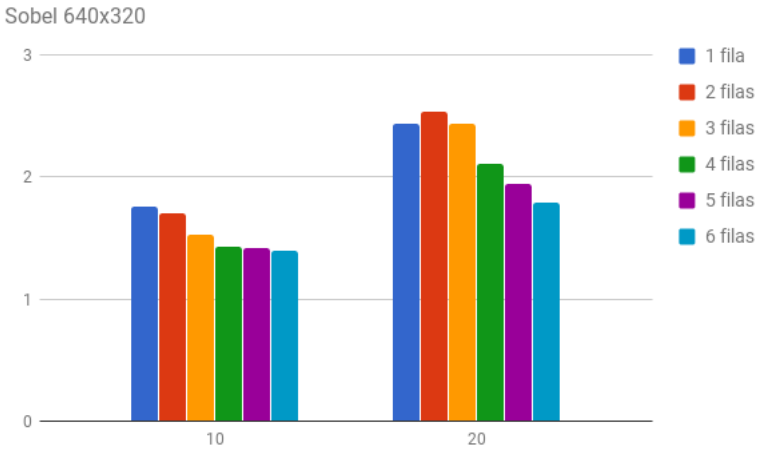


Figura 4.18: Speedup do algoritmo Sobel com relação ao número de filas para imagens de 640x320

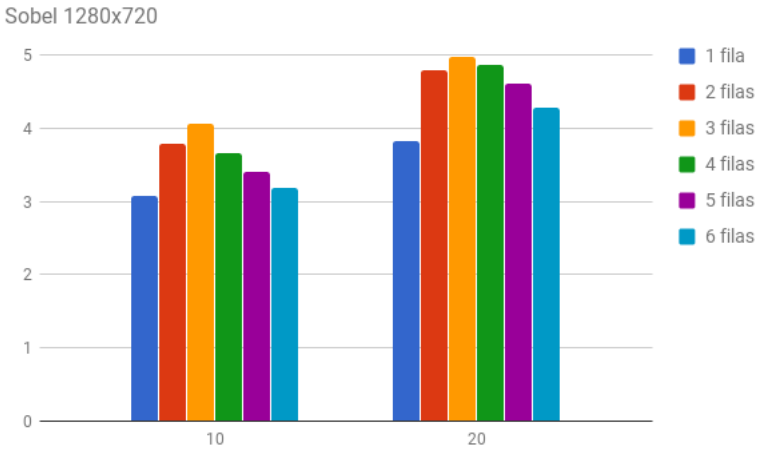


Figura 4.19: Speedup do algoritmo Sobel com relação ao número de filas para imagens de 1280x720

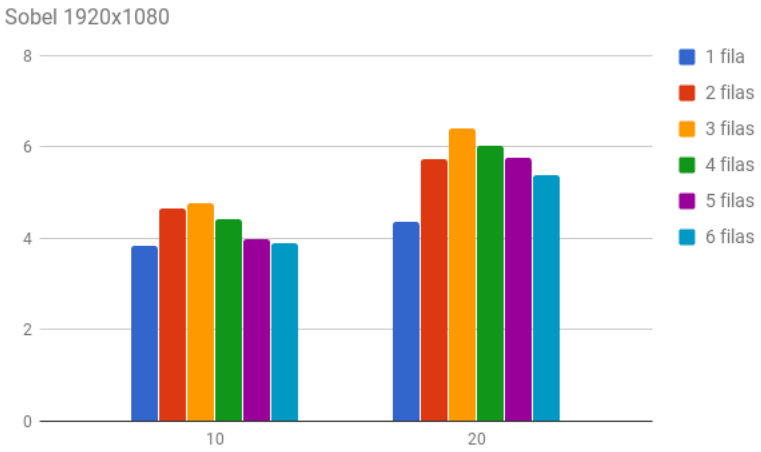


Figura 4.20: Speedup do algoritmo Sobel com relação ao número de filas para imagens de 1920x1080

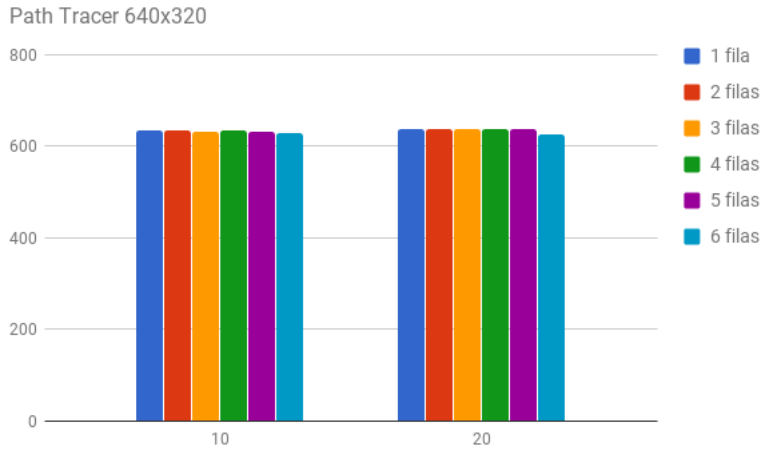


Figura 4.21: Speedup do algoritmo Path-Tracing com relação ao número de filas para imagens de 640x320

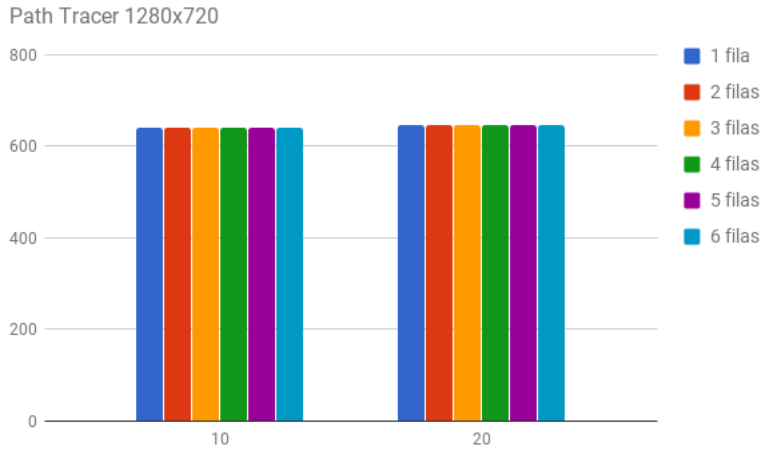


Figura 4.22: Speedup do algoritmo Path-Tracing com relação ao número de filas para imagens de 1280x720

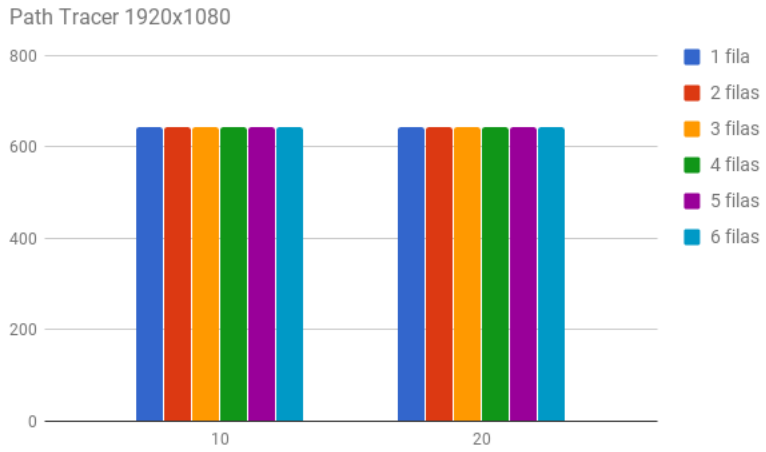


Figura 4.23: Speedup do algoritmo Path-Tracing com relação ao número de filas para imagens de 1920x1080

Referências Bibliográficas

[1] "Intel Puts the Brakes on Moore's Law Intel Puts the Brakes on Moore's Law". `https://www.technologyreview.com/s/601102/intel-puts-the-brakes-on-moores-law/`. Accessed: 2017-01-31.

[2] PARKHURST, J., DARRINGER, J., GRUNDMANN, B. "From single core to multi-core: preparing for a new exponential". In: *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pp. 67–72. ACM, 2006.

[3] GRAY, J. "GRVI Phalanx: A Massively Parallel RISC-V FPGA Accelerator Accelerator". In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 17–20, May 2016. doi: 10.1109/FCCM.2016.12.

[4] "Proven Power Reduction with Xilinx UltraScale FPGAs". `https://www.xilinx.com/support/documentation/white_papers/wp466-proven-ultrascale-power-leaders.pdf`. Accessed: 2017-08-08.

[5] WELOLI, J. W., BILAVARN, S., DERRADJI, S., et al. "Efficiency Modeling and Analysis of 64-bit ARM Clusters for HPC". In: *2016 Euromicro Conference on Digital System Design (DSD)*, pp. 342–347, Aug 2016. doi: 10.1109/DSD.2016.74.

[6] PRONGNUCH, S., WIANGTONG, T. "Heterogeneous Computing Platform for data processing". In: *2016 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, pp. 1–4, Oct 2016. doi: 10.1109/ISPACS.2016.7824762.

[7] DENNIS, J. B. *The varieties of data flow computers*. IEEE Computer Society Press, 1986.

[8] MARZULO, L. A. J. *Explorando Linhas de Execuçao Paralelas com Programaçao Orientada por Fluxo de Dados*. Tese de Doutorado, Ph. D. dissertation, Universidade Federal do Rio de Janeiro, 2011.

[9] ALVES, T. A., GOLDSTEIN, B. F., FRANÇA, F. M., et al. "A minimalistic dataflow programming library for python". In: *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on*, pp. 96–101, 2014.

[10] LEVOY, M. "Display of surfaces from volume data", *IEEE Computer Graphics and Applications*, v. 8, n. 3, pp. 29–37, May 1988. ISSN: 0272-1716. doi: 10.1109/38.511.

[11] HUGHES, J. F., FOLEY, J. D. *Computer graphics: principles and practice*. Pearson Education, 2014.

[12] KAJIYA, J. T., VON HERZEN, B. P. "Ray tracing volume densities". In: *ACM SIGGRAPH Computer Graphics*, v. 18, pp. 165–174. ACM, 1984.

[13] "Cornell Box Cornell Box". `http://www.graphics.cornell.edu/online/box/`. Accessed: 2017-06-07.

[14] "cbox cbox". `http://graphics.stanford.edu/~henrik/images/cbox.html)`. Accessed: 2017-08-16.

[15] SOBEL, I., FELDMAN, G. "A 3x3 isotropic gradient operator for image processing", *a talk at the Stanford Artificial Project in*, pp. 271–272, 1968.

[16] PATTERSON, D. *Computer Organization and Design, Third Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 2004.

[17] GEPNER, P., KOWALIK, M. F. "Multi-core processors: New way to achieve high system performance". In: *Parallel Computing in Electrical Engineering, 2006. PAR ELEC 2006. International Symposium on*, pp. 9–13. IEEE, 2006.

[18] MOORE, G. "Cramming more Components onto Integrated Circuits. Electronics April 19, 1965". 1965.

[19] "voltaArchitecture voltaArchitecture". `https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/#new-gpu`. Accessed: 2017-07-13.

[20] MCCLANAHAN, C. "History and evolution of gpu architecture", *A Survey Paper*, p. 9, 2010.

[21] "OpenGL OpenGL". `https://www.opengl.org/about/#1`, . Accessed: 2017-05-26.

[22] PASSERAT-PALMBACH, J. *Contributions to parallel stochastic simulation: Application of good software engineering practices to the distribution of pseudorandom streams in hybrid Monte-Carlo simulations.* Tese de Doutorado, Université Blaise Pascal-Clermont-Ferrand II, 2013.

[23] "Java The Java Language Environment". `http://www.oracle.com/technetwork/java/intro-141325.html`. Accessed: 2017-01-27.

[24] "GC_JavaCL JavaCL-FAQ.wiki". `https://code.google.com/archive/p/javacl/wikis/ReadFromAndWriteToBuffersAndImages.wiki`. Accessed: 2017-01-28.

[25] "lwjgl lwjgl". `https://www.lwjgl.org/#learn-more`. Accessed: 2017-07-13.

[26] "javaSerialization javaSerialization". `https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html`. Accessed: 2017-07-28.

[27] "CLEvent CLEvent". `http://nativelibs4java.sourceforge.net/javacl/api/1.0.0-RC4/com/nativelibs4java/opencl/CLEvent.html#waitFor-com.nativelibs4java.opencl.CLEvent...-`. Accessed: 2017-07-29.

[28] WATSON, I., GURD, J. "A prototype data flow computer with token labelling". In: *afips*, p. 623. IEEE, 1899.

[29] ARVIND, CULLER, D. *The tagged token dataflow architecture (preliminary version).* Relatório técnico, Tech. Rep. Laboratory for Computer Science, MIT, Cambridge, MA, 1983.

[30] NIKHIL, R., OTHERS. "Executing a program on the MIT tagged-token dataflow architecture", *IEEE Transactions on computers*, v. 39, n. 3, pp. 300–318, 1990.

[31] JURIJ'ILCY, B. R., UNGERER, T. "Asynchrony in parallel computing: From dataflow to multithreading", 1998.

[32] JOHNSTON, W. M., HANNA, J., MILLAR, R. J. "Advances in dataflow programming languages", *ACM Computing Surveys (CSUR)*, v. 36, n. 1, pp. 1–34, 2004.

[33] "PyOpenCL PyOpenCL". `https://mathema.tician.de/software/pyopencl/`. Accessed: 2017-01-29.

[34] "GIL GIL". `https://docs.python.org/2/glossary.html#term-global-interpreter-lock`. Accessed: 2017-06-27.

[35] BOSBOOM, J., RAJADURAI, S., WONG, W.-F., et al. *StreamJIT: A commensal compiler for high-performance stream programming*, v. 49. ACM, 2014.

[36] "rCuda rCuda". `http://www.rcuda.net/index.php/what-s-rcuda.html`. Accessed: 2017-07-12.

[37] TUPINAMBÁ, A., SZTAJNBERG, A. "DistributedCL: a framework for transparent distributed GPU processing using the OpenCL API". In: *Computer Systems (WSCAD-SSC), 2012 13th Symposium on*, pp. 187–193. IEEE, 2012.

[38] "Cuda C Programming Guide". `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4idxrP1Vq/`. Accessed: 2017-05-30.

[39] PEÑA, A. J., REAÑO, C., SILLA, F., et al. "A complete and efficient CUDA-sharing solution for HPC clusters", *Parallel Computing*, v. 40, n. 10, pp. 574–588, 2014.

[40] GREGG, C., DORN, J., HAZELWOOD, K. M., et al. "Fine-Grained Resource Sharing for Concurrent GPGPU Kernels." In: *HotPar*, 2012.

[41] "LambdaExpressionsJava LambdaExpressionsJava". `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`. Accessed: 2017-08-03.

[42] "anonymousclasses anonymousclasses". `https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html`. Accessed: 2017-08-03.

[43] THIES, W., AMARASINGHE, S. "An empirical characterization of stream programs and its implications for language and compiler design". In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pp. 365–376. ACM, 2010.

[44] "FFmpeg FFmpeg". `https://ffmpeg.org/`. Accessed: 2017-06-09.

[45] "OpenCL OpenCL". `https://www.khronos.org/opencl/`, . Accessed: 2017-05-27.

[46] "Khronos Khronos". `https://www.khronos.org/`. Accessed: 2017-05-29.

[47] ROTH, S. D. "Ray casting for modeling solids", *Computer Graphics and Image Processing*, v. 18, n. 2, pp. 109–144, 1982. doi: 10.1016/0146-664X(82)

90169-1. Disponível em: <https://doi.org/10.1016/0146-664X(82)90169-1>.

[48] MUNSHI, A., GASTER, B., MATTSON, T. G., et al. *OpenCL programming guide*. Pearson Education, 2011.

[49] "threadSafe threadSafe". https://docs.oracle.com/cd/E19455-01/806-5257/6je9h033e/index.html. Accessed: 2017-08-06.

[50] "rayTracingWiki rayTracingWiki". https://en.wikipedia.org/wiki/Ray_tracing_(graphics). Accessed: 2017-08-16.

[51] "pathTracerRay pathTracerRay". http://ilchoi.weebly.com/monte-carlo-path-tracing.html). Accessed: 2017-08-16.