UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Modelo e Critérios de Teste Estrutural para Programas CUDA

Helder Jefferson Ferreira da Luz

Tese de Doutorado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP SÃO CARLOS

Helder Jefferson Ferreira da Luz

Modelo e Critérios de Teste Estrutural para Programas CUDA

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo - ICMC/USP, como parte dos requisitos para obtenção do título de Doutor em Ciências - Ciências de Computação e Matemática Computacional. VERSÃO REVISADA

Área de concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Paulo Sérgio Lopes de Souza

São Carlos

Março 2023

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

L979m

Luz, Helder Jefferson Ferreira da
Modelo e Critérios de Teste Estrutural para Programas CUDA / Helder Jefferson Ferreira da Luz; orientador Paulo Sérgio Lopes de Souza. -- São Carlos, 2023.
192 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2023.

1. Programação Paralela. 2. CUDA e GPU. 3. Teste de Software Estrutural. 4. Critérios de Teste Estrutural. 5. Ferramenta de Teste Estrutural. I. Lopes de Souza, Paulo Sérgio, orient. II. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Helder Jefferson Ferreira da Luz

Structural Testing Model and Criteria for CUDA Programs

Thesis submitted to the Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo - ICMC/USP, in partial fulfillment of the requirements for the degree of the Doctor in Science - Program in Computer Science and Computational Mathematics. FINAL VERSION

Concentration area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Paulo Sérgio Lopes de Souza

São Carlos

March 2023

Dedico esse trabalho aos meus pais,
que me apoiaram durante todas as dificuldades.

AGRADECIMENTOS

Gostaria de expressar meus sinceros agradecimentos a todas as pessoas que me ajudaram durante o meu período de pesquisa. Em especial, gostaria de agradecer:

Aos meus pais, Claúdia e Humberto, por todo o apoio e incentivo durante essa jornada acadêmica. Sem a ajuda de vocês, eu não teria chegado até aqui.

À minha namorada Andressa, por estar ao meu lado durante todo o processo, compreendendo as dificuldades e me apoiando em cada momento.

Ao meu orientador, prof. Dr. Paulo Sérgio, por ter me guiado e apoiado em todas as etapas da minha pesquisa. Sou grato pela sua paciência, conselhos e conversas presenciais e remotas.

À professora Simone, por toda a ajuda que me ofereceu em várias fases do meu doutorado.

Ao professor João Bispo, da Faculdade de Engenharia da Universidade do Porto, por ter me ajudado no desenvolvimento da ferramenta que utilizei na minha pesquisa.

Aos meus amigos do grupo SemDisquete, Davi, Danilo, Fernando, Richard, Fabio, Raphael e Guilherme, pelas conversas enriquecedoras e pelo apoio emocional durante todos esses anos.

Aos membros do LaSDPC, que conheci durante o período em que morei em São Carlos. Obrigado por compartilharem seus conhecimentos e me ajudarem a crescer profissionalmente.

Novamente, muito obrigado a todos que contribuíram para a realização desse trabalho.

"We are always seeking for those things which are in the clouds,
not for those that lie at our feet."
Henry Ford

RESUMO

LUZ, H. J. F. **Modelo e Critérios de Teste Estrutural para Programas CUDA**. 2023. 192p. Tese (Doutorado em Ciências) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2023.

As GPUs têm sido amplamente utilizadas na computação de propósito geral para resolver problemas em diversas áreas. Para utilizar esses processadores, é necessário adotar modelos de programação paralela, como o CUDA, que visam simplificar o desenvolvimento de aplicações de propósito geral para serem executadas em GPUs. A despeito das facilidades trazidas por esse modelo, desenvolver aplicações com CUDA não é trivial. Além disso, os desenvolvedores possuem pouca experiência na programação de aplicações paralelas, ocasionando diversos tipos de defeitos. Apesar da necessidade de mitigar defeitos em aplicações CUDA, há poucos critérios de teste estrutural para revelar defeitos desse modelo de programação. Buscando melhorar a qualidade de programas concorrentes CUDA, foi definido um modelo e critérios de teste estrutural para apoiar a atividade de teste, auxiliando na seleção de casos de teste e propiciando a revelação de defeitos. Para validar o modelo e critérios de teste propostos, foi desenvolvido a ferramenta ValiCUDA, que realiza a instrumentação e análise do programa, gerando os elementos requeridos para cada critério de teste. A ferramenta permite executar os programas e avaliar a cobertura alcançada para cada critério. Para avaliar a aplicação dos critérios de teste, foi realizado um experimento com validação estatística, buscando averiguar as métricas de custo, eficácia e *strength*. No experimento foram inseridos defeitos com base em uma taxonomia de defeitos para avaliar a eficácia dos critérios. Os resultados demonstraram que os critérios de teste propostos são capazes de revelar defeitos em aplicações CUDA e podem auxiliar o testador na atividade de teste deste tipo de aplicação.

Palavras-chave: Programação Paralela, CUDA e GPU, Teste de Software Estrutural, Critérios de Teste Estrutural, Ferramenta de Teste Estrutural.

ABSTRACT

LUZ, H. J. F. **Structural Testing Model and Criteria for CUDA Programs**. 2023. 192p. Thesis (Doctorate in Science) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2023.

GPUs have been widely used in general-purpose computing to solve problems in various fields. CUDA programs can be used on these processors to allow the execution of general-purpose applications on GPUs. Even with the facilities brought from this model, developing CUDA applications is not a trivial activity. Furthermore, developers have little experience with parallel programming, and this fact leads to several types of defects. Looking to improve CUDA applications quality, we defined a structural test model and criteria to support the test activity, helping in the test case selection and enabling the identification of defects. To validate the proposed test model and criteria, we developed the ValiCUDA tool, which instrument and analyze the program, generating required elements for each test criterion. The tool enables the program execution and assessment of the achieved coverage for each criterion. To evaluate the test criteria application, we performed a statistical validation experiment, looking for cost, effectiveness and strength metrics. In the experiment, we inserted defects based on a detect taxonomy to evaluate the criteria efficacy. The results showed that the proposed test criteria can identify defects in CUDA applications and can help the tester in the testing activity for this type of application.

Keywords: Parallel Programming, CUDA and GPU, Structural Software Testing, Structural Testing Criteria, Structural Testing Tool.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

ACM	Association for Computing Machinery
AMD	Advanced Micro Devices
API	Application Programming Interface
ARM	Advanced RISC Machine
CPR	Checkpoint / Restart
CPU	Central Processing Unit
CU	Control Unit
CUDA	Compute Unified Device Architecture
DP Unit	Double Precision Unit
DS	Data Stream
DSP	Digital Signal Processing
ECC	Error Correction Code
GFC	Grafo de Fluxo de Controle
GFCP	Grafo de Fluxo de Controle Paralelo
GFCPmc	Grafo de Fluxo de Controle Paralelo para Memória Compartilhada
GKLEE	GPU KLEE
GPGPU	General Purpose computation on Graphics Processing Unit
GPU	Graphics Processing Unit
HLSL	High Level Shading Language
HPC	High Performance Computing
ICMC/USP	Instituto de Ciências Matemáticas e de Computação / Universidade de São Paulo
ID	Identificador
IEEE	Institute of Electrical and Electronics Engineers

IPC	Interprocess Communication
IS	Instruction Stream
LaSDPC	Laboratório de Sistemas Distribuídos e Programação Concorrente
LD/ST	Load/Store
LM	Local Memory
MIMD	Multiple Instruction; Multiple Data
MISD	Multiple Instruction; Single Data
MPI	Message Passing Interface
MU	Memory Unit
NPU	Network Processing Unit
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
OpenMP	Open Multi-Processing
CFG	Control Flow Graph
PCFG	Parallel Control Flow Graph
PCFGsm	Parallel Control Flow Graph for Shared Memory
PThreads	Posix Threads
PU	Processing Unit
PUG	Prover of User GPU programs
RAM	Random Access Memory
SESA	Symbolic Execution and Static Analysis
SFU	Special Function Unit
SIMD	Single Instruction; Multiple Data
SIMT	Single Instruction; Multiple Threads
SISD	Single Instruction; Single Data
SM	Stream Multiprocessors

SMT	Satisfiability Modulo Theories
SP	Stream Processors
VGA	Video Graphic Array
VV&T	Validação, Verificação e Teste

SUMÁRIO

1 INTRODUÇÃO

A computação paralela vem sendo cada vez mais utilizada (GRAMA *et al.*, 2003; PATTERSON; HENNESSY, 2013). A disponibilidade da computação de alto desempenho (*High Performance Computing* - HPC) é estimulada por uma constante evolução e barateamento do hardware, onde verifica-se facilmente a existência de *clusters beowulf* (STERLING *et al.*, 1995), processadores com múltiplos núcleos (PATTERSON; HENNESSY, 2013) e processadores específicos para o processamento de vídeo (*Graphics Processing Unit* - GPU) (PATTERSON; HENNESSY, 2013), rede (*Network Processing Unit* - NPU) (GILADI, 2008), entre outros (TOP500.ORG, 2022). Uma característica marcante desse hardware paralelo é a presença de duas categorias de máquinas: as MIMD (*Multiple Instruction; Multiple Data*) e as SIMD (*Single Instruction; Multiple Data*). *Clusters* e processadores com múltiplos núcleos são bons exemplos de máquinas MIMD, respectivamente com memória distribuída e compartilhada. Processadores específicos, como as GPUs, são exemplos de máquinas SIMD[1]. Alguns exemplos onde a HPC é aplicada diretamente são: tratamento de *big data*, dinâmica molecular, processamento de imagens, previsão climática – incluindo terremotos e maremotos.

Do ponto de vista de *software*, os sistemas operacionais e modelos de programação estão evoluindo, na tentativa de oferecer suporte adequado para esses novos recursos de *hardware*. Nesse sentido, são comumente utilizados sistemas operacionais de rede, como muitas distribuições Linux (TANENBAUM; BOS, 2014); e modelos de programação concorrente com memória distribuída ou memória compartilhada (GRAMA *et al.*, 2003). O padrão MPI (*Message Passing Interface*) (MPI Forum, 2015) é usualmente encontrado para a HPC com memória distribuída. Já o OpenMP (*Open Multi-Processing*) (OpenMP ARB, 2015) e PThreads (*POSIX Threads*) (IEEE, 2008) são facilmente encontrados na tentativa de viabilizar a HPC com memória compartilhada. Outros modelos de programação com uma crescente utilização são CUDA (*Compute Unified Device Architecture*) (NVIDIA, 2022), OpenCL (*Open Computing Language*) (KHRONOS GROUP, 2021) e OpenACC (OPENACC ORGANIZATION, 2022), os quais viabilizam o uso de GPUs para o processamento de imagens e de outras computações genéricas, normalmente envoltas com problemas matriciais.

O OpenCL, inicialmente proposto pela Apple, é gerenciado pela *Khronos The Compute Working Group* (KHRONOS GROUP, 2021). *Khronos* é um consórcio de empresas como AMD (*Advanced Micro Devices*), Nvidia e ARM (*Advanced RISC Machine*) para

[1] Embora as GPUs atualmente tenham extrapolado os conceitos iniciais de uma máquina SIMD, elas ainda apresentam muitas das características SIMD quando as *threads* são executadas em paralelo

a criação de padrões abertos para a programação de sistemas heterogêneos. OpenCL é um *framework* para o desenvolvimento de programas concorrentes portáteis, que possam ser executados em diversas plataformas, como diversas arquiteturas de CPUs (*Central Processing Unit*), GPUs, DSPs (*Digital Signal Processing*), dentre outras (KHRONOS GROUP, 2021).

OpenACC foi desenvolvido por um grupo de empresas formado pela Cray, CAPS, Nvidia e PGI, buscando simplificar o desenvolvimento de programas paralelos heterogêneos para diversos tipos de sistemas computacionais (OPENACC ORGANIZATION, 2022). Sua utilização é similar ao do modelo de programação OpenMP.

CUDA, desenvolvido pela Nvidia, é um modelo de programação e plataforma de computação paralela para propósitos gerais voltado para GPUs da Nvidia. Ele facilita a implementação e solução de problemas computacionais complexos em GPU (CHENG; GROSSMAN; MCKERCHER, 2014). Ele possui um *kit* de desenvolvimento de *sofware* que permite utilizar linguagens de programação de alto nível como C/C++, Fortran, Java e Python para implementar aplicações paralelas heterogêneas, utilizando as capacidades computacionais oferecidas pela GPU. Apesar do seu uso restrito a uma marca, de acordo com a Pala (2022), a Nvidia domina hoje o mercado de placas gráficas, possuindo atualmente 82% do mercado de placas gráficas dedicadas no segundo trimestre de 2022. CUDA também apresenta um desempenho substancialmente maior que o OpenCL (FANG; VARBANESCU; SIPS, 2011) na maioria das aplicações e *benchmarks* encontrados no mercado. Devido a isso, neste momento, CUDA se apresenta como o principal modelo de programação para GPU utilizado para HPC e é considerado um padrão de fato na área e, com isso, foi utilizado neste projeto.

Os dois principais apelos para o uso de CUDA são: (i) a possibilidade de se obter ganhos significativos de desempenho; (ii) a possibilidade de desenvolvimento de código concorrente muito próximo ao sequencial, abstraindo-se vários dos detalhes da geração de processos/*threads* e da IPC (*Interprocess Communication*). Os ganhos expressivos de desempenho são obtidos com o uso de GPUs, as quais são compostas de várias unidades de processamento replicadas. Estas são alimentadas com instruções e dados vindos de memórias existentes na GPU. A replicação das unidades de processamento permite a execução simultânea de várias *threads*; a organização de memória da GPU fornece uma vazão de instruções e dados adequada, pois é baseada fortemente nos princípios de localidade espacial e temporal de caches (GRAMA *et al.*, 2003); evitando assim concorrer com o barramento e bancos de memória no computador hospedeiro da GPU.

Entretanto, sendo um modelo de programação heterogêneo, o desenvolvimento de programas concorrentes CUDA envolve a implementação de código fonte para ser executado na CPU e outra parte na GPU. A GPU depende da CPU para fazer a execução de sua parcela de código. Além disso, a arquitetura SIMD e os diversos tipos de memória

implicam em diversos desafios na implementação para alcançar um bom desempenho em uma aplicação semanticamente correta (CHENG; GROSSMAN; MCKERCHER, 2014).

Dado esse cenário, desenvolver aplicações com CUDA não é, de fato, uma atividade trivial. Isso ocorre principalmente porque os desenvolvedores estão acostumados a implementar programas sequenciais, onde suas execuções podem ser entendidas por uma pessoa ao observar o fluxo sequencial do código fonte (KIRK; WEN-MEI, 2013). Códigos em CUDA necessitam que o desenvolvedor tenha uma abstração maior e consiga visualizar o que de fato está acontecendo sobre o hardware disponível da GPU.

Tanenbaum e Bos (2014) afirmam que os programadores possuem pouca experiência na implementação de aplicações paralelas. Frequentemente profissionais que utilizam esses modelos de programação concorrente não são da área da computação, tendo pouca experiência com o desenvolvimento correto de soluções computacionais. Isso dificulta o desenvolvimento de aplicações de qualidade que apresentem ganhos de desempenho expressivos e, ao mesmo tempo, tenham poucos defeitos. Ainda segundo os mesmos autores, as aplicações desenvolvidas (concorrentes ou não) apresentam entre dois e dez defeitos não revelados para cada mil linhas de código (TANENBAUM; BOS, 2014). No caso do modelo de programação CUDA, isso pode acarretar em defeitos de natureza variada. Segundo Cook (2013), um exemplo de defeito é o acesso inválido de um vetor/matriz, que acontece quando não se limita o acesso das *threads* aos limites das variáveis. Em uma aplicação CUDA, frequentemente o número de *threads* será diferente do tamanho dos dados, dessa forma não ocorrendo um mapeamento *um-para-um* entre os mesmos. Esse defeito não é detectado facilmente de maneira estática (antes da execução) e pode ou não acontecer em função do fluxo de dados e da plataforma.

Para melhorar a qualidade do software, há um conjunto de três atividades bem conhecidas que podem ser aplicadas no processo de desenvolvimento do software: Validação, Verificação e Teste (VV&T) (DELAMARO; MALDONADO; JINO, 2007). A Validação busca garantir que o produto que está sendo construído está de acordo com os requisitos funcionais e não funcionais especificados (BALCI, 1998). A Verificação questiona se o produto está sendo construído corretamente, se o comportamento do mesmo é o esperado, buscando assegurar sua consistência, completitude e corretude. A atividade de Teste tem por objetivo revelar defeitos no programa ou modelo por meio da sua execução; se a aplicação não funcionar corretamente, diz-se que foi revelado um defeito na mesma (MYERS; SANDLER; BADGETT, 2011).

Neste cenário, o teste de aplicações paralelas de alto desempenho, incluindo aquelas desenvolvidas sob o modelo de programação CUDA, é um desafio (LEI; CARVER, 2006; YANG; POLLOCK, 1997). O teste de tais aplicações paralelas ainda é pouco desenvolvido e utilizado. Há a falta de modelos de teste estruturais específicos para extrair informações relevantes para testar tais aplicações, principalmente aquelas relacionadas à comunicação,

sincronização e ao não determinismo existentes usualmente em função do comportamento dinâmico das mesmas. Atrelados aos modelos de teste, são necessários novos critérios e ferramentas de teste que deem o suporte adequado à atividade de teste, de maneira a viabilizá-la na prática.

Considerando uma perspectiva histórica, na década de 90 os principais desafios para testar processos concorrentes eram (YANG; POLLOCK, 1997): (i) desenvolver técnicas para a análise estática; (ii) detectar situações indesejadas como: problemas de comunicação, sincronização, de fluxo de dados entre processos e *threads* e de *deadlock*; (iii) forçar uma execução determinística (*replay* de uma execução concorrente) com a mesma entrada de dados; (iv) gerar uma representação do programa concorrente que capture informações requeridas para o teste; (v) investigar se critérios de teste sequenciais são úteis para programas concorrentes; (vi) projetar critérios com foco no fluxo de dados, considerando passagem de mensagens e o uso de variáveis compartilhadas.

O teste de programas concorrentes evoluiu nas últimas duas décadas, porém, diferentes questões ainda estão sem resposta. Uma relação atual (não exaustiva) de possíveis desafios pode ser: (i) desenvolver técnicas de análise estática do código fonte combinadas a análise dinâmica, esta oriunda da execução dos processos; (ii) gerar um modelo de teste e critérios de teste que capturem informações de programas concorrentes que interagem concomitantemente por passagem de mensagens e por memória compartilhada; (iii) gerar modelos e critérios de teste ortogonais às linguagens, e modelos de programação em máquinas MIMD/SIMD; (iv) analisar a eficácia de novos critérios em revelar defeitos e estabelecer uma relação de inclusão dos mesmos, com vistas a otimização do teste e redução dos seus custos; (v) desenvolver ferramentas de teste viáveis e ortogonais às linguagens de programação concorrente; (vi) reduzir o custo da atividade de teste em processos concorrentes permitindo a aplicação de modelos e critérios a programas reais; (vii) investigar a aplicação das técnicas funcionais, baseadas em defeitos e estruturais no contexto de programas concorrentes, considerando aspectos complementares, de eficiência e de custo.

Estas questões são constantemente alvo de pesquisas e vêm sendo investigadas no ICMC/USP (Instituto de Ciências Matemáticas e de Computação / Universidade de São Paulo) pelo projeto TestPar - Mecanismos de Apoio ao Teste de Programas Paralelos (HAUSEN *et al.*, 2007; SARMANHO *et al.*, 2008; SOUZA *et al.*, 2005; SOUZA *et al.*, 2008; VERGILIO; SOUZA; SOUZA, 2005; SOUZA; SOUZA; ZALUSKA, 2014). De ordem prática, conforme as pesquisas desenvolvidas no projeto TestPar avançam, novas perguntas surgem e aumentam a lista de desafios acima. O projeto TestPar investiga o teste de programas concorrentes de maneira abrangente, considerando modelos, critérios e ferramentas de teste estrutural no contexto de aplicações baseadas nos paradigmas de memória compartilhada e passagem de mensagem para máquinas MIMD.

Os modelos representam informações sobre fluxos de controle e de dados de processos concorrentes. Os critérios fornecem uma medida de cobertura para avaliar o progresso do teste e assim guiar a geração de novos casos de teste. A ferramenta ValiPar implementa os conceitos investigados e instanciados nos modelos e critérios de teste (SOUZA *et al.*, 2008). Há versões desta ferramenta instanciadas para programas concorrentes com passagem de mensagem e memória compartilhada. São representantes do primeiro grupo as ferramentas ValiPVM, ValiMPI e ValiBPEL (SOUZA *et al.*, 2008; HAUSEN *et al.*, 2007; ENDO *et al.*, 2008). No segundo grupo está a ValiPThread (SARMANHO *et al.*, 2008) que testa programas implementados em C/Pthread. A ValiPar para Java unifica os modelos de passagem de mensagem e memória compartilhada (SOUZA *et al.*, 2013; PRADO *et al.*, 2015).

Apesar da evolução que o projeto TestPar já trouxe para a área de teste de programas concorrentes, um dos desafios ainda a serem descobertos é como aplicar o teste estrutural de programas concorrentes no contexto da programação CUDA, de modo a revelar defeitos difíceis de serem revelados. Este desafio é o foco principal deste projeto de pesquisa.

Dentro do projeto TestPar, nossa proposta é o desenvolvimento de um modelo de teste e critérios de teste estruturais para aplicações CUDA. O modelo captura estaticamente as informações relacionadas ao fluxo de controle, fluxo de dados, comunicação e sincronização do programa. Os critérios de teste propostos auxiliam na atividade de teste para revelar defeitos nesse tipo de aplicação.

O modelo e critérios de teste foram implementados na ferramenta ValiCUDA baseada na ferramenta ValiMPI. Uma das principais modificações realizadas foi o uso do *framework* Clava, que substitui o antigo módulo ValiInst, para o suporte à análise estática e instrumentação do código a ser testado. O *framework* Clava (BISPO; CARDOSO, 2020) é um compilador de código fonte baseado no projeto Clang (LATTNER, 2008). Ele possibilita a pesquisa, análise e transformação de código fonte usando a linguagem de programação LARA, que é baseada em *JavaScript* e permite a definição de estratégias para extrair informações e modificar o código fonte. A ValiCUDA ainda faz uso dos módulos ValiElem, ValiExec e ValiEval da ferramenta ValiMPI. Eles foram modificados para suportar a execução de programas CUDA, geração dos elementos requeridos e ser capaz de avaliar a cobertura dos elementos requeridos para todas as *threads* do programa.

Para validação do modelo e critérios, realizamos um experimento utilizando a ferramenta desenvolvida. Buscamos avaliar os critérios com base em três métricas: custo, eficácia e *strength*. Para a avaliação utilizamos um conjunto de *benchmarks* representativo das características encontradas em programas CUDA. Utilizando uma classificação de tipos de defeitos, inserimos defeitos nos programas com base em operadores de mutação. Executamos os programas utilizando um conjunto de casos de teste adequados a cada

programa e critério de teste. Com base nos resultados obtidos, realizamos uma análise estatística para avaliar o custo, eficácia e *strength* dos critérios. Os resultados demonstraram que os critérios de teste propostos são eficazes para revelar diversos tipos de defeitos encontrados em aplicações CUDA.

1.1 Justificativa

A principal justificativa para o desenvolvimento deste projeto é que os programas concorrentes em CUDA desenvolvidos até o momento não têm o devido suporte de técnicas de teste estruturais de uma atividade de teste, capaz de determinar se tais programas CUDA foram testados adequadamente. Essa situação pode ser verificada pela natureza e pouca quantidade de trabalhos relacionados encontrados na literatura sobre o teste estrutural de programas concorrentes com CUDA. Tal situação implica em códigos falhos, com defeitos não revelados, fato que compromete significativamente a qualidade das soluções paralelas propostas com CUDA.

De fato, não se conhece, até o momento, como devem ser construídos modelos, critérios e ferramentas de teste para dar suporte apropriado para tal atividade. Além disso, não se tem conhecimento como tais recursos de teste devem ser aplicados para revelar defeitos não conhecidos e, assim, minimizar o custo do desenvolvimento e da manutenção de programas concorrentes em CUDA.

Dessa forma, o desenvolvimento deste projeto de pesquisa justifica-se por buscar importantes respostas sobre o teste de HPC com CUDA. A principal delas é entender como quantificar e qualificar se um programa CUDA foi bem testado. Os aspectos quantitativos vêm da cobertura de novos critérios de teste e os aspectos qualitativos vêm da capacidade desses critérios em revelar defeitos. Como heurística, assume-se que se os critérios foram comprovadamente capazes de revelar defeitos e foram satisfeitos, então o programa em teste foi bem testado e, potencialmente, tem uma melhor qualidade[2].

1.2 Objetivo

De uma forma geral, o principal objetivo deste trabalho é investigar como melhorar a qualidade de programas concorrentes CUDA, por meio da definição de um modelo e critérios de teste estruturais que auxiliem na revelação de defeitos em tais programas, difíceis de serem revelados. Dessa forma, permitindo quantificar e qualificar se um programa CUDA foi bem testado e auxiliando o testador na escolha dos critérios de teste e na realização da atividade de teste.

[2] Ressalta-se que, por definição, a atividade de teste não garante que um programa está livre de defeitos; apenas que critérios de teste foram satisfeitos (DELAMARO; MALDONADO; JINO, 2007). Espera-se que os critérios sejam rígidos o suficiente para revelar defeitos, se estes existirem.

Os objetivos específicos são: **(i)** definir classes de defeitos para programas concorrentes CUDA, apresentando os tipos de erros cometidos pelos programadores nesse modelo de programação; **(ii)** desenvolver um modelo de teste estrutural que considere elementos que necessitam ser testados em programas concorrentes que utilizam o modelo de programação CUDA; **(iii)** desenvolver critérios de teste estrutural que contribuam para a melhoria do conjunto de casos de teste e assim auxiliem na busca por defeitos ainda não revelados em programas concorrentes CUDA; **(iv)** desenvolver um protótipo de ferramenta de teste que implemente o modelo e os critérios estruturais de teste propostos.

1.3 Organização

O Capítulo 2 apresenta a arquitetura e o modelo de programação CUDA.

O Capítulo 3 apresenta os conceitos fundamentais relacionados ao teste de software de programas sequenciais e programas concorrentes.

O Capítulo 4 mostra os tipos de defeitos encontrados em aplicações que utilizam o modelo de programação CUDA.

O Capítulo 5 apresenta a proposta de modelo de teste e critérios de teste voltadas às aplicações CUDA.

O Capítulo 6 demonstra o desenvolvimento da ferramenta de teste que implementa o modelo e critérios propostos.

O Capítulo 7 mostra o planejamento dos experimentos para validação do modelo e critérios de teste, e apresenta os resultados obtidos.

No Capítulo 8 são discutidos os trabalhos relacionados.

Finalmente, no Capítulo 9 apresentamos as nossas conclusões, as contribuições realizadas, limitações e os trabalhos futuros.

2 A PROGRAMAÇÃO CUDA

2.1 Considerações Iniciais

Neste capítulo apresentamos alguns dos principais conceitos relacionados ao modelo de programação CUDA, necessários para a definição dos critérios de teste neste contexto de programação. Inicialmente mostramos na Seção 2.2 o modelo arquitetural que dá suporte à programação CUDA e empregado nas GPUs da Nvidia, demonstrando detalhes de implementação e relacionando com a taxonomia de Flynn (FLYNN, 1972). Na seção 2.3 mostramos o modelo de programação CUDA em si, relacionando o código executado na CPU com o código executado na GPU. Na seção 2.4 apresentamos a hierarquia de memória da GPU oferecida às *threads* CUDA, funcionando em conjunto com a hierarquia de organização das *threads*. Por fim, a Seção 2.5 mostra os tipos de sincronizações de *threads* oferecidas pelo modelo de programação.

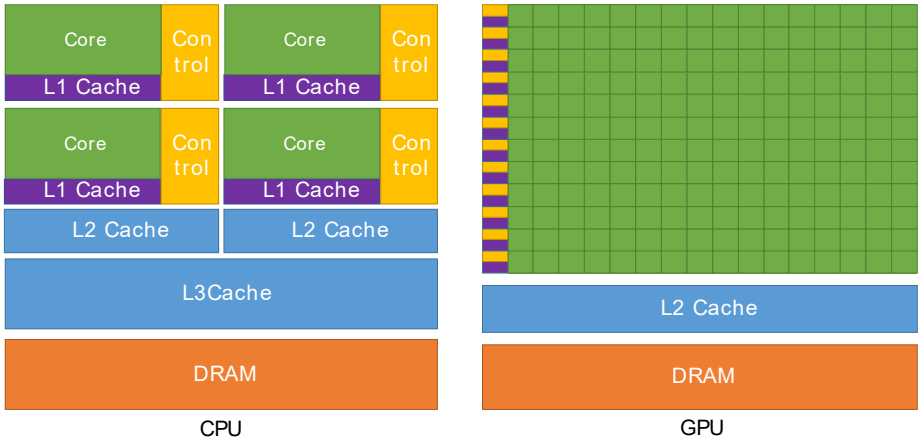
2.2 Modelo Arquitetural

CUDA (*Compute Unified Device Architecture*) é uma plataforma de *hardware* e *software* para computação paralela, voltada à implementação de aplicações de propósito geral para execução em GPU (*Graphics Processing Unit*) (STORTI; YURTOGLU, 2015). Desenvolvida pela Nvidia, ela foi introduzida ao mercado em 2006, buscando possibilitar o uso de suas GPUs em problemas computacionais complexos de forma mais eficiente que usando a CPU (NVIDIA, 2022).

GPUs são co-processadores que buscam auxiliar a CPU. Dessa forma, a CPU precisa informar as computações que ela irá fazer. Juntamente, ela não precisa conseguir executar todas as tarefas da CPU, pois em um computador que possua uma GPU ele também terá uma CPU, que poderá realizar essas tarefas. Isso permitiu dedicar o desenvolvimento da arquitetura da GPU exclusivamente para gráficos (PATTERSON; HENNESSY, 2020). Sendo o processamento gráfico uma atividade que costuma executar uma mesma instrução para vários elementos do gráfico, o processamento paralelo de dados foi o principal foco no desenvolvimento da GPU. A Figura 1 ilustra as diferenças arquiteturais entre a CPU e a GPU. Na GPU, grande parte do espaço físico é dedicado às unidades de processamento de dados e pouco para unidades de controle e *cache*, devido à finalidade de processar imagens e vídeos. Por sua vez, a CPU possui para cada unidade de processamento sua própria unidade de controle e cache, com unidades mais complexas e, com isso, dedicando menos espaço físico para o processamento dos dados em si.

Esse modelo arquitetural da GPU pode ser entendido por meio da Taxonomia de Flynn (1972), que classifica as arquiteturas de processamento paralelo da seguinte forma:

Figura 1 – Diferença arquitetural entre a CPU e a GPU.



Fonte: (NVIDIA, 2022)

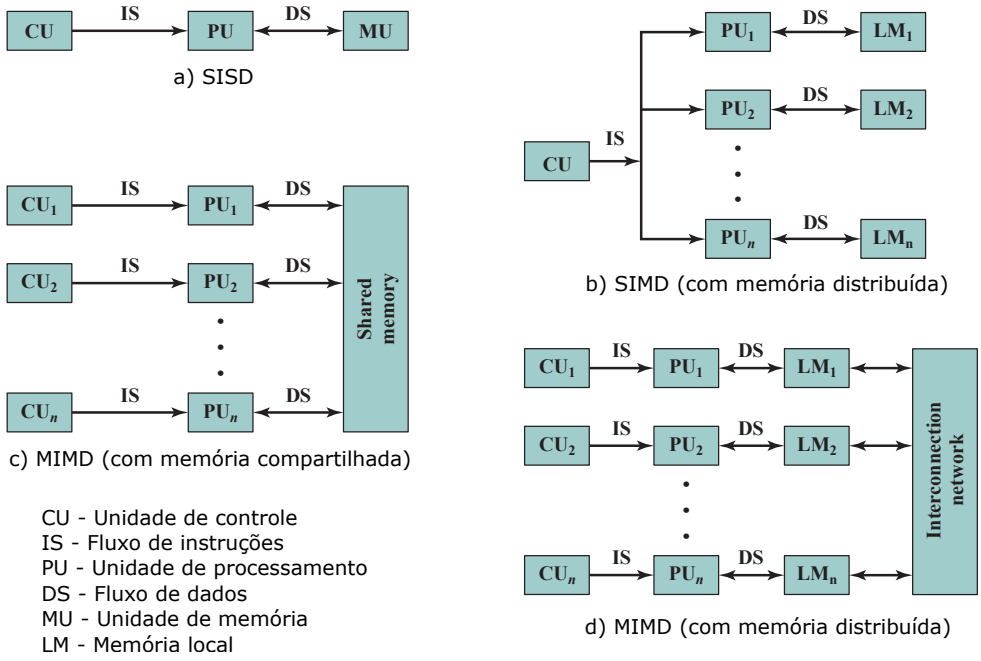
- SISD (*Single Instruction; Single Data*) - Um único processador executa um único fluxo de instruções que opera em dados armazenados em uma única memória (STALLINGS, 2012). A Figura 2 (a) apresenta um exemplo deste tipo de sistema. Neste caso um processador seria equivalente a, por exemplo, um núcleo das CPUs atuais. As arquiteturas da categoria SISD são vistas como máquinas sequenciais e não consideram todo o paralelismo existente abaixo do nível de Arquitetura do Conjunto de Instruções (STALLINGS, 2012).

- SIMD (*Single Instruction; Multiple Data*) - Uma única unidade de controle define uma única instrução para ser executada por diversas unidades de processamento. Cada unidade de processamento está associada a um dado de memória diferente, dessa forma uma mesma instrução é executada em dados diferentes no mesmo instante de tempo (STALLINGS, 2012). Na Figura 2 (b) é demonstrada a organização deste sistema.

- MISD (*Multiple Instruction; Single Data*) - Várias unidades de processamento com unidades de controle dedicadas operam em um mesmo fluxo de dados executando instruções diferentes (PATTERSON; HENNESSY, 2013).

- MIMD (*Multiple Instruction; Multiple Data*) - Várias unidades de processamento executam instruções distintas em dados diferentes (STALLINGS, 2012). Os atuais processadores x86_64 *multicore* se encontram nesta categoria. A Figura 2 mostra duas organizações desse sistema, uma com memória compartilhada (c) e outra com memória distribuída (d). O primeiro, como já citado, pode ser observado nos amplamente utilizados processadores *multicore* e o segundo em *clusters de computadores*.

Figura 2 – Organização de sistemas paralelos.



Fonte: (STALLINGS, 2012)

Nessa taxonomia, uma CPU *multicore* se baseia no modelo MIMD enquanto a GPU apresenta fortes características do modelo SIMD, com alguns aspectos do modelo MIMD , como a presença de múltiplas *threads* (vide explicações sobre SIMT e outros aspectos da GPU, ainda nesta seção). Esses modelos possuem funcionamento e finalidade diferentes. O núcleo da GPU tem foco em tarefas com alta paralelização de dados, possuindo lógica de controle simples e focando na vazão de dados. A CPU possui uma lógica de controle mais complexa, sendo dessa forma mais focada em programas sequenciais ou com poucos processos e *threads*. A CPU oferece bom desempenho em programas que apresentam muitas mudanças no fluxo de controle, causadas por estruturas de repetição e decisão, enquanto a GPU sofre um forte impacto negativo no desempenho nesses casos (CHENG; GROSSMAN; MCKERCHER, 2014), por ter que serializar a execução.

Em conjunto com as características do modelo SIMD e MIMD, o modelo CUDA traz também o conceito de *threads* à arquitetura. Para esse modelo, a Nvidia chamou de o modelo arquitetural de SIMT, de *Single Instruction; Multiple Threads* (CHENG; GROSSMAN; MCKERCHER, 2014). Combinando paralelismo em nível de instrução, SIMD, MIMD e *multithreading* (PATTERSON; HENNESSY, 2020), ele permite gerenciar, escalonar e executar milhares de *threads* CUDA na GPU. Essas *threads* são agrupadas em *warps* de até 32 *threads*. As *threads* de mesmo *warp* executam sincronamente a mesma instrução, conforme o modelo SIMD.

Nessa arquitetura, as unidades de processamento (SP - *Stream Processors*) são

chamadas de núcleos CUDA (CUDA *Cores*). Cada núcleo CUDA possui uma unidade de processamento de inteiros e uma unidade de processamento de ponto flutuante simples. Os SPs são agrupados em unidades chamadas de SM, sigla para *Stream Multiprocessors*. Cada SM possui dezenas ou centenas de núcleos CUDA, variando conforme a arquitetura. Especificamente na arquitetura Pascal um SM possui 64 núcleos CUDA. Dentro do SM, esses núcleos são novamente agrupados, como por exemplo em conjuntos de 16 ou 32 núcleos que podem executar uma única instrução concorrentemente, conforme o modelo SIMD. Conjuntos distintos podem executar instruções distintas, trazendo, dessa forma, características da arquitetura MIMD.

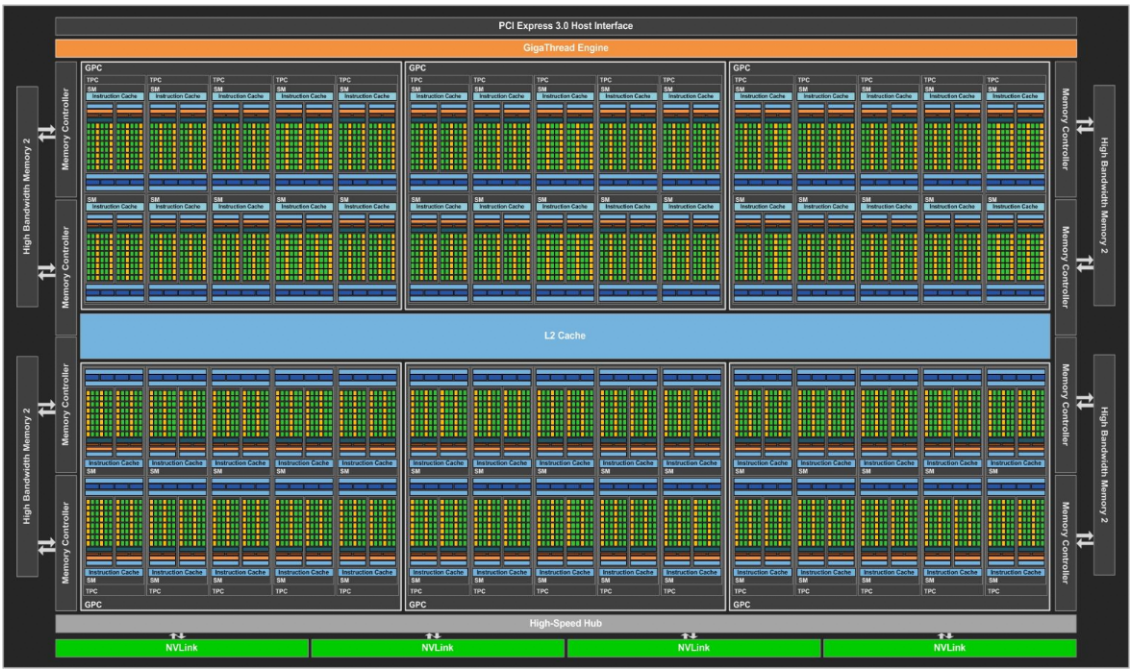
A Figura 3 ilustra uma unidade SM da arquitetura Pascal. Além dos núcleos CUDA, há unidades de precisão dupla chamada DP Unit (*Double Precision Unit*), unidades para leitura e escrita na memória (LD/ST - *Load/Store*) e unidades para cálculos de funções especiais SFU (*Special Function Unit*) (CHENG; GROSSMAN; MCKERCHER, 2014). Estas quatro unidades definem o conjunto de instruções suportadas pela GPU para processamento geral. O SM ainda possui cache dedicado aos núcleos CUDA e registradores exclusivos para cada núcleo. O *Warp Scheduler* faz o escalonamento dos *warps* aos núcleos para serem executados. Possuindo dois escalonadores, ele consegue escalonar dois *warps* por vez. A *dispatch unit* realiza o envio das instruções que serão executadas nas unidades de processamento. Nesta arquitetura, em específico, há dois *dispatch* por *warp scheduler*, permitindo que até duas instruções diferentes sejam processadas por *warp*, mas em outras arquiteturas pode estar limitado a apenas uma instrução por *warp*.

Uma GPU pode apresentar vários SMs. Na arquitetura Pascal completa há 60 SM, com 64 núcleos CUDA cada, totalizando 3840 núcleos de processamento em uma única placa gráfica. Versões reduzidas podem apresentar um número menor de SM. A Figura 4 ilustra a GPU Tesla P100 com arquitetura Pascal contendo 60 SM. Nela há ainda uma memória cache L2 compartilhado entre todos os SMs, um escalonador de trabalho chamado *GigaThread Engine* que envia os grupos de *threads* para cada unidade SM, e um controlador de memória que dá acesso à memória principal da GPU, compartilhada entre todos os núcleos CUDA.

Figura 3 – Organização do *Stream Multiprocessor* da arquitetura Pascal da Nvidia.

Fonte: NVIDIA Corporation (2016)

Figura 4 – GPU Tesla P100 completa com 60 SM.



Fonte: NVIDIA Corporation (2016)

2.3 Modelo de Programação

Para os desenvolvedores utilizarem a GPU para computação de propósito geral, a Nvidia fornece o modelo de programação CUDA por meio do *software* CUDA *Toolkit* (STORTI; YURTOGLU, 2015). Este modelo oferece um ambiente de *software* que permite ao desenvolvedor implementar programas para GPU utilizando linguagens de programação de alto nível, como C, C++, Fortran e Java. O modelo oferece primitivas que estendem essas linguagens, permitindo o desenvolvimento de aplicações de forma similar a modelos de programação para CPU, sem a necessidade de utilizar APIs gráficas.

O modelo de programação CUDA é voltada à computação heterogênea. O desenvolvimento de qualquer aplicação CUDA envolve a implementação de duas partes: o código que será executado na CPU e o código que será executado na GPU. Isso é devido à dependência que a GPU possui, sendo um co-processador, ela depende da CPU para enviar os comandos sobre o que deve ser executado, assim como transferir dados da memória principal para a memória da GPU, e da memória da GPU para a memória principal da CPU.

O desenvolvimento do código fonte CUDA é dividido em duas partes: a parcela executada na CPU e que utiliza a memória principal, também chamada de *host*; e a parcela executada na GPU, chamada de *device*, que utiliza sua memória dedicada (FARBER, 2012). As funções implementadas para execução no *device* são denominadas *kernel*. O *kernel* pode ser escrito como um código sequencial, no entanto cada *thread* irá executar uma cópia desse *kernel*, com o conjunto de *threads* sendo executados concorrentemente na GPU.

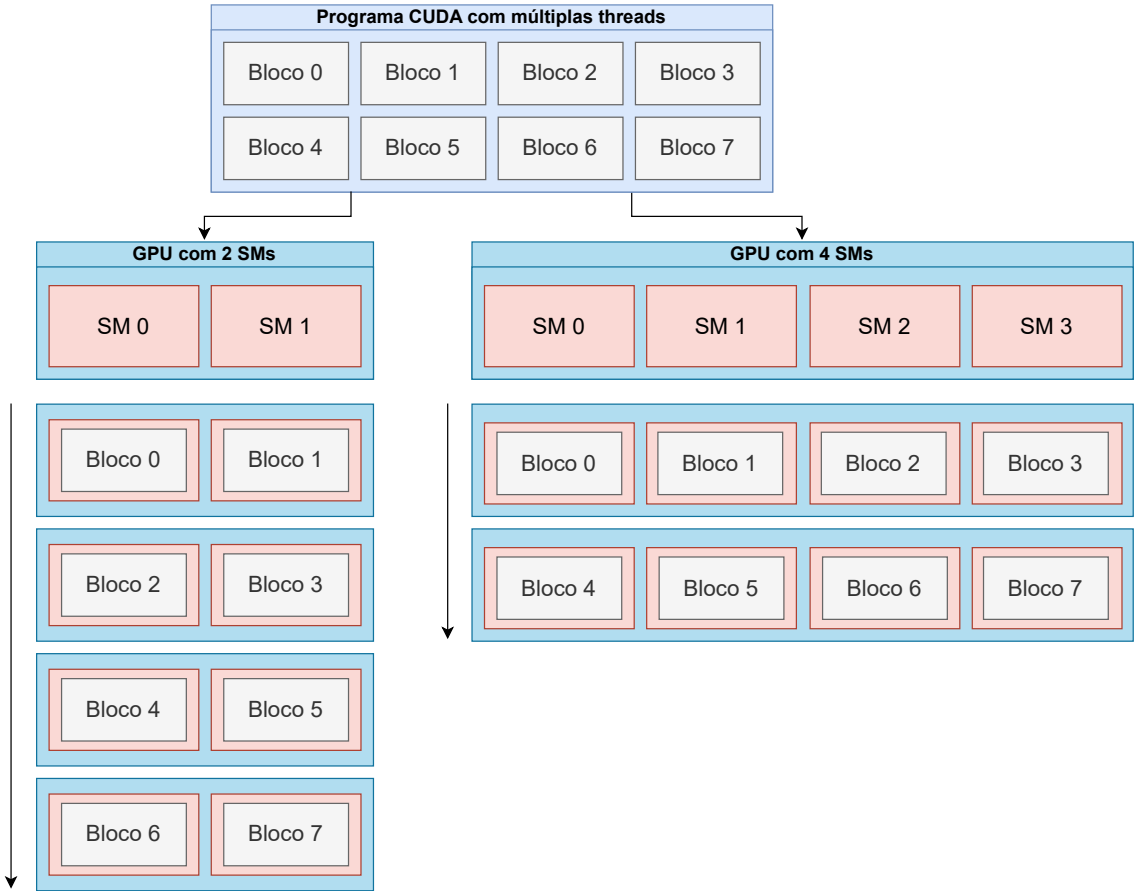
Para definir que uma função será executada no *host* ou no *device* é utilizado um qualificador de tipo de função. Uma função que não possui um qualificador indica que ele será executado no *host* mas, opcionalmente, neste caso pode ser utilizado o qualificador "___host___". Para funções que serão executadas no *device* é utilizado o qualificador "___global___" ou "___device___". Somente os *kernels* com qualificador "___global___" podem ser chamados pelo *host*.

O código do *host* pode operar de forma independente do código do *device* (CHENG; GROSSMAN; MCKERCHER, 2014). Quando um *kernel* é executado, o controle retorna imediatamente ao *host*, liberando a CPU para executar atividades complementares durante a execução do *kernel* no *device*. Em geral, as chamadas de *kernel* possuem comportamento assíncrono em relação ao *host*, assim como as chamadas a outras primitivas do modelo de programação CUDA. Isso viabiliza a sobreposição de computação entre a CPU e GPU. Entretanto, eles possuem por padrão um comportamento síncrono entre primitivas CUDA, desse modo, chamadas consecutivas de *kernel* possuem comportamento síncrono, assim como primitivas de transferência de memória, a menos que sejam utilizadas primitivas e

lógica que especifiquem um comportamento assíncrono entre eles.

Na chamada do *kernel* é especificada pelo *host* a quantidade de *threads* CUDA que executarão tal *kernel* (CHENG; GROSSMAN; MCKERCHER, 2014). A chamada é feita por meio da sintaxe $kernel <<< gridSize, blockSize >>>$, que cria um *grid* relacionado ao *kernel*. *Grid* para CUDA é uma unidade que agrupa todas as *threads* do *kernel*. Dentro dele as *threads* são agrupadas em blocos de até 1024 *threads* que, por sua vez, são divididos em *warps* de até 32 *threads*. Na chamada do *kernel*, o "gridSize" especifica a quantidade de blocos que o *grid* terá, e "blockSize" especifica quantas *threads* cada bloco terá. A relação entre as duas medidas dará a quantidade total de *threads* do *grid* executando o *kernel*. Essa organização por blocos permite ao modelo oferecer escalabilidade entre diferentes configurações de GPUs, que possuem quantidades diferentes de SM, como ilustrado na Figura 5.

Figura 5 – Escalabilidade do modelo de execução CUDA



Fonte: Adaptado de NVIDIA (2022)

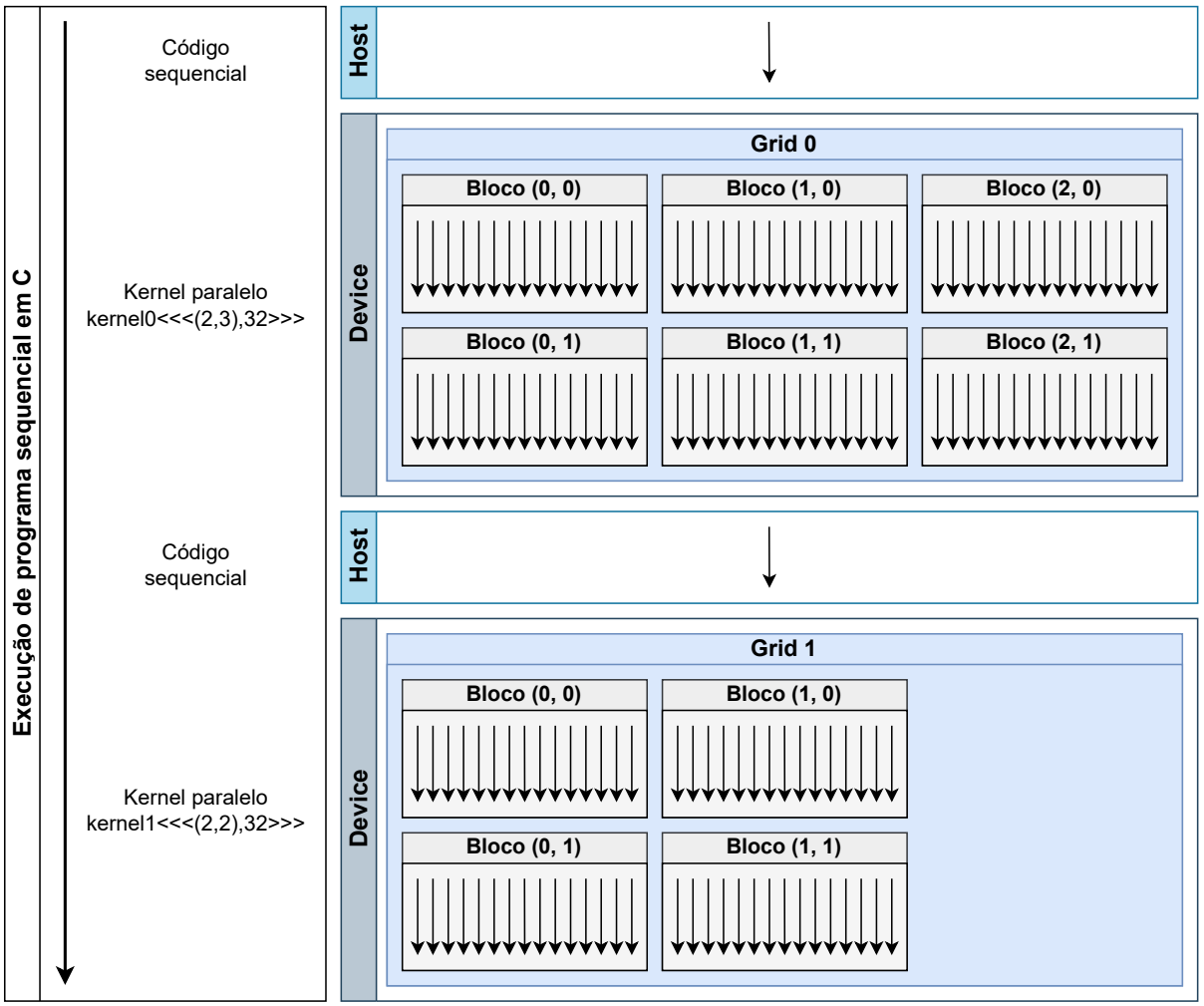
A execução das *threads* CUDA por meio da chamada de *kernel* no *host* segue o modelo *fork-join*. Nele, o *host* inicializa N quantidade de *threads* do *grid*, causando um *fork*.

Ao finalizar ocorre o *join* das *threads* do *grid* no *host* em um dos pontos de sincronização encontrados no código do *host*. Além disso, na chamada de *kernel* ainda há a comunicação entre o *host* e as *threads* por meio da passagem de variáveis para a função do *kernel*.

A quantidade de *threads* que executarão o *kernel* pode ser especificada em uma, duas ou três dimensões para facilitar o mapeamento das *threads* em relação à estrutura de dados. Os valores de "gridSize" e "blockSize" podem definir os tamanhos em mais de uma dimensão e, com base neles, podem ser calculados os identificadores (ID) das *threads* que são usados para definir em qual parte do dado cada *thread* irá trabalhar. O modelo fornece variáveis que permitem calcular o ID com base na sua posição em relação ao *grid* e bloco de *threads*. Cada *thread* possui acesso ao parâmetro local "threadIdx", que é uma estrutura com os valores do eixo X, Y e Z da *thread* dentro do bloco. Há ainda o "blockDim" que permite acessar o comprimento do bloco em relação ao eixo X, Y e Z; o "blockIdx" que informa a posição do bloco dentro do *grid*, e o "gridDim" que informa o comprimento do *grid* em relação aos três eixos. A Figura 6 apresenta um *grid* com uma configuração bidimensional de blocos, informando a identificação de cada bloco com base no eixo X e Y.

Logo, um programa que utiliza o modelo de programação CUDA possui uma parte serial ou paralela executada no *host*, e um ou mais *kernels* paralelos executados no *device* (KIRK; WEN-MEI, 2013). A Figura 6 ilustra o processo de execução típico de uma aplicação CUDA. O programa inicia com a execução do código do *host*, que faz a alocação de memória da GPU e copia os dados da memória da CPU para a memória da GPU para que o *grid* tenha acesso. Após, é chamada a execução do *kernel* na GPU, criando um *grid* para realizar a execução de instruções nos dados transferidos. Ao finalizar a execução do *grid*, o *host* retorna a executar, podendo realizar a cópia dos dados da memória da GPU de volta para a memória da CPU, para que ele tenha acesso ao resultado da computação. No caso da ilustração, um novo *grid* é executado em seguida.

Figura 6 – Fluxo de execução típico de uma aplicação CUDA



Fonte: Adaptado de NVIDIA (2022)

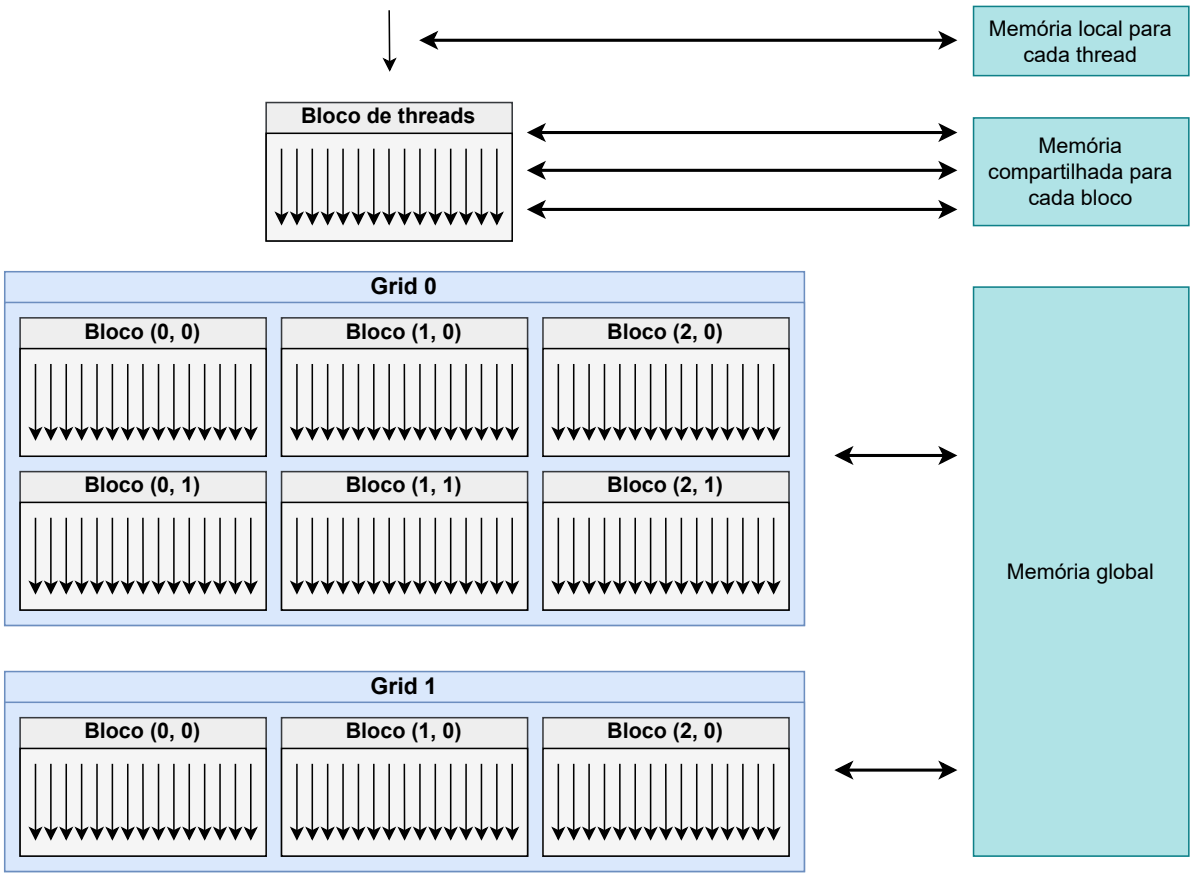
2.4 Modelo de Memória

Em conjunto com a hierarquia e organização de *threads* por meio do uso de *grid*, bloco e *warp*, o modelo de programação oferece uma hierarquia de memória que pode ser acessada pelas *threads*. A hierarquia de memória está intimamente ligada com a hierarquia de *threads* apresentada. Nela, cada *thread* possui uma memória local, visível apenas por ela. Cada bloco de *threads* CUDA possui uma memória compartilhada por bloco, onde todas as *threads* de um mesmo bloco possuem acesso, podendo se comunicar por meio dela. O último nível na hierarquia é a memória global, compartilhada entre todas as *threads* do *grid*. Ela permite que *threads* de blocos diferentes se comuniquem, entretanto ela é a memória mais lenta da hierarquia, possuindo alto custo de latência em relação à memória compartilhada, que é armazenada na memória cache do SM (CHENG; GROSSMAN;

MCKERCHER, 2014).

Essa hierarquia de memória é ilustrada na Figura 7. Ela relaciona a hierarquia das *threads* com a memória, onde há a memória local para cada uma das *threads*. Há a memória compartilhada por bloco para cada bloco de *threads*, e a memória global para os *grids*. Além dessas memórias, ainda há uma memória de constante, que é armazenado na memória principal do *device*, mas possui cache dedicado em cada SM, e uma memória de textura, também armazenada na memória principal do *device* e com cache dedicado em cada SM (CHENG; GROSSMAN; MCKERCHER, 2014).

Figura 7 – Hierarquia de *thread* e memória do modelo de programação CUDA.



Fonte: Adaptado de NVIDIA (2022)

2.5 Sincronização

Como CUDA é um modelo de programação paralela heterogênea, ele possui primitivas para sincronização da execução das *threads* do tipo barreira. Há sincronização em dois níveis: no nível de bloco da *thread*, onde todas as *threads* de um bloco precisam chegar no mesmo ponto de execução do código; e no nível de sistema, onde ambos o *host* e as *threads*

do *grid* precisam esperar para poder continuar (CHENG; GROSSMAN; MCKERCHER, 2014).

Em nível de sistema, as sincronizações são feitas no código do *host*. Elas podem ser sincronizações implícitas, ocorrendo pelo uso de primitivas que não possuem a finalidade específica de sincronização, como a execução de outro *kernel* ou a transferência de memória entre *host* e *device*. Ela também pode ocorrer de forma explícita, pelo uso de barreira de sincronização, como a primitiva "cudaDeviceSynchronize()". Sincronização no nível de sistema é a única forma de sincronizar todas as *threads* do *grid* (CHENG; GROSSMAN; MCKERCHER, 2014). Dessa forma, caso seja necessário esse nível de sincronização para resolver a computação, será necessário decompor o problema para ser executado em mais de uma chamada de *kernel*.

No nível de bloco, as sincronizações são definidas no código do *kernel*. Essas sincronizações ocorrem de forma explícita pelo uso da primitiva "___syncthreads()". Ela é uma barreira de sincronização que define que todas as *threads* de um mesmo bloco precisam esperar até que todas as *threads* tenham chegado ao ponto de sincronização (CHENG; GROSSMAN; MCKERCHER, 2014). Esta sincronização é utilizada para coordenar a comunicação entre as *threads* em relação ao uso da memória global e a memória compartilhada em nível de bloco, de forma a garantir que o conteúdo armazenado esteja visível para todas as *threads* do bloco. O lado negativo do uso das sincronizações é uma penalidade no desempenho.

A falta do uso da sincronização pode resultar em um erro na saída do programa. O Código-fonte Código 1 demonstra um possível defeito em aplicações que utilizam esse paradigma pela ausência da sincronização. Ele apresenta um *kernel* CUDA, que tem como objetivo deslocar os elementos de um vetor em uma posição para a esquerda. Cada *thread* irá copiar um elemento do vetor referente à posição indicada pelo identificador da *thread* e escrevê-lo na posição anterior. No entanto, essa atividade está sendo realizada no mesmo vetor. Desta forma, para que este programa apresente a semântica esperada é necessário que todas as *threads* leiam o valor do vetor, copiem para a variável local e então escrevam na posição anterior do vetor. Tendo um comportamento não-determinístico, não há garantias que a execução ocorrerá nessa ordem. Em diversas replicações de execução da aplicação ela pode apresentar o comportamento esperado, no entanto em certas replicações apresentar erro. Para garantir a semântica na execução, e dessa forma corrigir o erro, é necessário utilizar uma barreira de sincronização entre a atividade de leitura dos valores pelas *threads* e o armazenamento na posição anterior do vetor.

Código 1 – Aplicação CUDA com defeito de sincronização.

```
1  __global__ void shift_vector(int *a, int n){
2      int tid = blockDim.x * blockIdx.x + threadIdx.x;
3      int copy;
4
5      if (tid > 0){
6          copy = a[tid-1];
7          // __syncthreads();
8          a[tid] = copy;
9      }
10 }
```

2.6 Considerações Finais

Neste capítulo nós mostramos a plataforma o modelo de programação CUDA nos níveis de *hardware* e *software*. No nível de *hardware* foi discutido o funcionamento do modelo arquitetural , destacando o conceito SIMT, o qual combina características dos modelos SIMD e MIMD. Para a utilização da arquitetura foi apresentado o modelo de programação CUDA, que oferece o CUDA *Toolkit* para programação paralela heterogênea, onde é necessário implementar o código do *host* e do *device*. Este modelo permitiu desenvolver aplicações voltadas à execução em GPU utilizando linguagens de alto nível como C/C++, sem a necessidade de se utilizar APIs gráficas, facilitando o desenvolvimento.

Apesar das facilidades trazidas com o modelo de programação CUDA para o desenvolvimento de programas concorrentes, o mapeamento dos dados e tarefas pelas *threads* não é um processo simples, o que pode acarretar em falhas difíceis de serem reveladas e/ou depuradas. Durante esse mapeamento, o desenvolvedor ainda precisa verificar outras questões de implementação, que podem resultar em deterioração de desempenho ou erros durante a execução, como realizar acesso coalescente[1] à memória principal, diminuir conflitos às memórias compartilhadas, mitigar divergências de *threads* que podem ocorrer devido à arquitetura SIMT da GPU. Por fim, é necessário que o desenvolvedor considere a utilização de barreiras de sincronização para manter a semântica da aplicação, assim como tornar certas operações atômicas.

[1] A leitura a memória global é feita em conjuntos de 128 bytes. Quando as *threads* fazem requisições de dado à memória, se estiverem em posições consecutivas, a GPU combina as requisições, diminuindo o número de transações com a memória.

3 TESTE DE SOFTWARE

3.1 Considerações Iniciais

Apresentamos neste capítulo os conceitos e recursos fundamentais relacionados ao teste de *software* sequencial e paralelo que serão necessários ao desenvolvimento da pesquisa descrita nesta monografia.

Na seção 3.2 são apresentados os conceitos básicos relacionados ao teste, como os termos comumente utilizados, as fases do teste e as técnicas de teste. Dentre as técnicas, estão o teste funcional, teste baseado em defeitos e teste estrutural. Utilizamos o teste estrutural no desenvolvimento deste projeto, e por isso nos aprofundamos nas explicações do mesmo. A seção 3.4 apresenta modelos e critérios de teste para programas concorrentes para paradigma de memória compartilhada e passagem de mensagem. Por fim, a seção 3.5 apresenta a ferramenta ValiPar relacionado ao projeto TestPar. O foco é na versão ValiMPI, demonstrando sua arquitetura e funcionamento, sendo essa ferramenta um dos pilares do desenvolvimento da ferramenta para teste de programas CUDA.

3.2 Conceitos Fundamentais de Teste

O teste de *software* faz parte de uma série de atividades da engenharia de *software* conhecida como Validação, Verificação e Teste (VV&T) (DELAMARO; MALDONADO; JINO, 2007). A validação busca assegurar que o programa criado está em conformidade com os requisitos do cliente. A verificação são atividades que buscam garantir que foi implementado corretamente uma função específica no programa (PRESSMAN; MAXIM, 2016). Boehm (1984) define a verificação com a pergunta “Estamos criando o produto corretamente?” e a validação com “Estamos criando o produto certo?”.

Segundo Myers, Sandler e Badgett (2011), o objetivo da atividade de teste de *software* é melhorar a qualidade e aumentar a confiabilidade do programa. Aumentar a confiabilidade significa encontrar e remover defeitos que levem a erros ou falhas do programa. As informações adquiridas durante a execução do teste que revelou a presença do defeito podem, posteriormente, ajudar na atividade de depuração para localizar e remover este defeito.

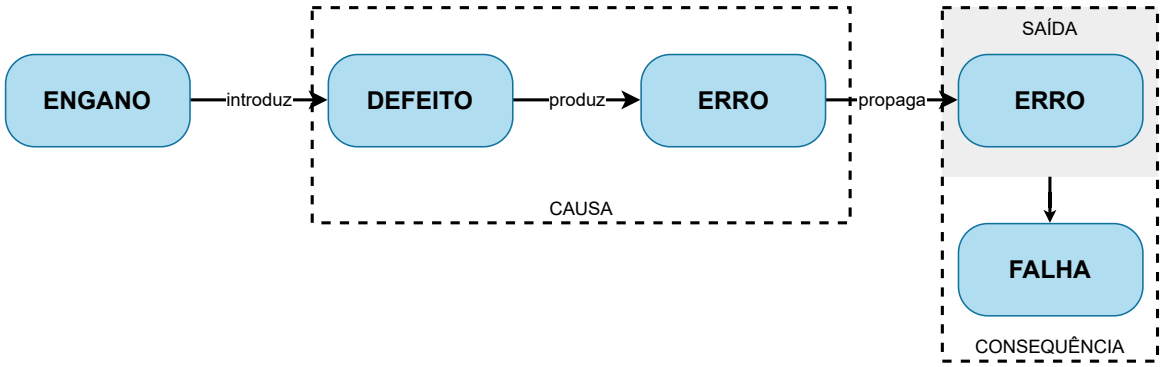
No contexto de teste de *software*, os vocábulos “engano”, “defeito”, “erro” e “falha” são comumente usados com significados semelhantes, entretanto eles possuem sentidos distintos. A norma IEEE (1990) define os termos utilizados neste trabalho como:

- Engano (*mistake*): Uma ação humana que produziu um resultado incorreto.

- Defeito (*fault*): Um passo, processo ou definição de dados incorretos, por exemplo, uma instrução incorreta em um programa.
- Erro (*error*): A diferença entre um valor ou condição computada, observada ou medida e o valor ou condição teoricamente correta ou especificada.
- Falha (*failure*): O resultado produzido na execução do programa não está de acordo com o esperado. Por exemplo, o resultado produzido pelo programa foi 12, entretanto o resultado esperado era 10.

A relação desses quatro termos é demonstrada na Figura 8. Nessa relação, um engano é cometido pelo desenvolvedor. Por sua vez, esse engano introduz um defeito no programa desenvolvido. O defeito pode produzir um erro em uma determinada execução do programa, que se propaga na saída do sistema apresentando uma falha no resultado. Apesar da padronização, é comum utilizar "erro" se referenciando a "defeito", "erro" ou "falha" (DELAMARO; MALDONADO; JINO, 2007).

Figura 8 – Relação de engano, defeito, erro e falha.



Fonte: Elaborada pelo autor.

Assim como o processo de desenvolvimento de *software*, o teste é dividido em algumas fases. Myers, Sandler e Badgett (2011) destacam diversas fases no processo de teste de *software*, sendo quatro delas para a revelar defeitos. Três fases são executadas durante o processo de desenvolvimento do *software* e uma após a entrega, durante o período de manutenção, sendo elas:

- Teste de unidade - também chamado de teste de métodos (MYERS; SANDLER; BADGETT, 2011). São testadas isoladamente as menores unidades do projeto do *software*, como funções, procedimentos, métodos, componentes ou módulos. Essa etapa busca identificar erros dentro dos limites da unidade relacionados aos seus requisitos, utilizando dados de teste para averiguar se o comportamento observado é igual ao especificado (IEEE, 1986);

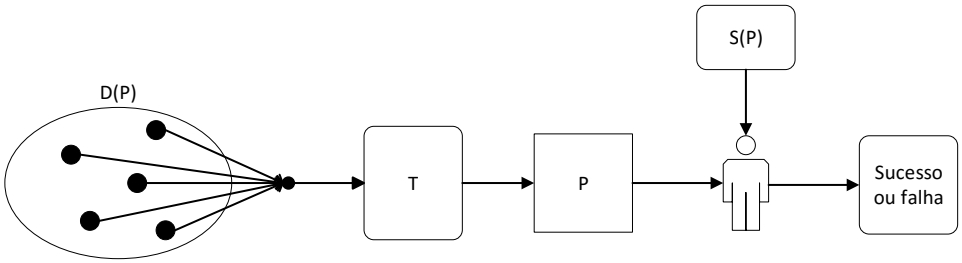
- Teste de integração - Busca revelar defeitos na interação das unidades do sistema, quando duas ou mais unidades são integradas;

- Teste de sistema - Aplicada após a integralização de todas as unidades do sistema, procura assegurar que as unidades foram integradas corretamente, funcionando conforme os requisitos especificados, e a revelar defeitos provenientes da integração. Nesta fase ainda são explorados aspectos de segurança, desempenho e robustez (DELAMARO; MALDONADO; JINO, 2007);

- Teste de regressão - Esta fase não está inclusa no processo de desenvolvimento do sistema. Ela ocorre após a entrega do sistema, durante o período de manutenção do *software*. Ao ocorrer modificações no sistema ou inclusão de novas funcionalidades, o teste de regressão busca garantir que foram corretamente implementadas e que as funcionalidades anteriormente testadas continuam funcionando corretamente (DELAMARO; MALDONADO; JINO, 2007).

Em cada fase realizam-se quatro etapas da atividade de teste: planejamento, projetos de casos de teste, execução e análise. A Figura 9 ilustra um cenário típico da atividade de teste (DELAMARO; MALDONADO; JINO, 2007). Sendo **P** o programa em teste, **D(P)** é o domínio de entrada do programa, ou seja, o conjunto com todos os valores de entrada possíveis, válidos e não válidos. Um "dado de teste" é um elemento do domínio de entrada de **P**, em conjunto com o resultado esperado temos um "caso de teste". Um conjunto de casos de teste **T** é executado no programa **P** e o resultado obtido é verificado. Em geral, cabe ao testador, com base na especificação **S(P)** do programa, analisar sobre a correção da execução dos casos de teste. Um caso de teste obtém sucesso quando consegue revelar um defeito no programa (MYERS; SANDLER; BADGETT, 2011).

Figura 9 – Cenário típico da atividade de teste.



Fonte: Adaptado de Delamaro, Maldonado e Jino (2007)

Garantir que um programa está livre de defeitos exige a execução de um teste exaustivo com todos os elementos do domínio de entrada do programa. Entretanto, isso pode ser impraticável em sistemas pequenos e inviável em sistemas maiores (MYERS;

SANDLER; BADGETT, 2011) devido à cardinalidade do domínio de entrada. Em um programa com uma variável de entrada numérica de 4 bytes exige 2^{32} casos de teste. O teste exaustivo se torna inviável em uma aplicação com diversas variáveis, além de implicar em um alto custo na sua aplicação.

Para viabilizar a atividade de teste, é necessário empregar alguma estratégia para o teste. Uma possível estratégia é selecionar apenas um subconjunto reduzido do domínio de entrada D(P) que consiga exprimir as características do domínio completo. A seleção dos dados de teste para o subconjunto pode ser feita de forma aleatória, conhecido como "teste aleatório", selecionando um grande número de casos de teste para, probabilisticamente, obter uma boa representação do domínio do programa. Uma segunda forma é estabelecer subdomínios de D(P) com base em alguma regra, e criar casos de teste para eles. Um método para a definição desses subdomínios são os critérios de teste.

Os critérios de teste buscam guiar o processo de definição e seleção dos casos de teste, sendo definidos com base na técnica de teste. A técnica de teste define a origem da informação utilizada para definir os subdomínios. Algumas das técnicas mais comuns são teste funcional, teste baseado em defeitos e teste estrutural.

3.2.1 Teste funcional

Considera o sistema como uma caixa preta, dessa forma não se tem acesso ao código fonte do programa. Para se definir os casos de teste é utilizada a especificação do sistema (MYERS; SANDLER; BADGETT, 2011). Isso exige que a especificação esteja atualizada e exprima corretamente o comportamento do sistema. Nesta técnica, as entradas são fornecidas baseadas em um critério e as saídas do sistema são aferidas para verificar se o comportamento é igual ao esperado. Alguns dos critérios mais conhecidos da técnica de teste funcional são: Particionamento em classes de equivalência, Análise do valor limite, Grafo causa-efeito e Error-guessing (MYERS; SANDLER; BADGETT, 2011).

3.2.2 Teste baseado em defeitos

Esta técnica utiliza a implementação do sistema e se baseia nos tipos mais comuns de defeitos encontrados durante o processo de desenvolvimento de *software*. Um critério conhecido desta técnica é o Teste de Mutação, também chamado de Análise de Mutantes. Como um bom caso de teste tem uma alta probabilidade de revelar defeitos (DELAMARO *et al.*, 2007), um objetivo dessa técnica é avaliar a qualidade do conjunto de casos de teste. Neste critério, modificações sintáticas são inseridas no código original, gerando versões chamadas mutantes. Os casos de teste são usados na execução do código original e dos mutantes. Caso as saídas do código original e do mutante sejam diferentes, é dito que o mutante está morto. Na circunstância da saída do código original e dos mutantes serem iguais, o mutante fica vivo. Nesse caso, os mutantes são equivalentes (DEMILLO; LIPTON;

SAYWARD, 1978), ou os dados de teste não foram capazes de demonstrar a diferença de comportamento entre ambos, sendo necessário melhorar a sua qualidade para demonstrar a diferença semântica entre eles. No caso de serem equivalentes, as modificações sintáticas não produziram semânticas diferentes em relação ao programa original.

3.3 Teste Estrutural

O teste estrutural, também conhecido como teste de caixa-branca, utiliza o conhecimento do código fonte do programa . Os critérios de teste utilizam informações relacionadas ao fluxo de controle e fluxo de dados do programa para derivar os requisitos de teste. O fluxo de controle refere-se as instruções sequências do programa, estruturas de decisão e estruturas de repetição utilizadas no código que influenciam no fluxo de execução do programa. O fluxo dados refere-se as ocorrências de variáveis no programa, sendo classificadas como definição e uso.

No teste estrutural, um programa P é representado por um Grafo de Fluxo de Controle (GFC). Ele é representado por G = (N, E, s), onde N é o conjunto de nós existentes no programa, sendo um nó um bloco sequencial de linhas de código sem desvio de execução. E é o conjunto de arestas, sendo uma aresta um possível desvio de execução do código. O s representa o nó inicial do programa (DELAMARO; MALDONADO; JINO, 2007). Nessa representação, a execução do primeiro comando de um bloco representado por um nó acarreta na execução de todos os outros comandos do bloco ¨

A partir do GFC, são definidos caminhos de execução compostos pelos nós e arestas. Um caminho é uma sequência finita de nós (n_1, n_2..., n_k), sendo $k >= 2$ e $n \in N$, e haja arestas de n_i para n_{i+1}, onde i=(1,2,...,k-1). Um caminho é uma sequência finita de nós interligadas pelas arestas, podendo ser definido como:

- simples: todos os nós que compõe o caminho são distintos, com exceção do primeiro e último;
- livre de laço: todos os nós que compõe o caminho são distintos;
- completo: o primeiro nó do caminho é o nó de entrada, e o último nó e o nó de saída do GFC.

Nesse contexto, Rapps e Weyuker (1982) estendem o GFC, definindo o grafo *Def-Uso* e adicionando informações referentes à definição e uso de variáveis. A definição de uma variável ocorre quando um valor é armazenado nela, isso pode ocorrer em uma atribuição ou passagem de valor em funções, por exemplo. O uso ocorre quando a variável é referenciada, sem modificar o seu valor. O uso da variável pode ser classificada em dois tipos: uso computacional (c-uso) ou uso predicativo (p-uso). O c-uso ocorre quando a variável é utilizada para realizar alguma computação. O p-uso ocorre quando a variável é

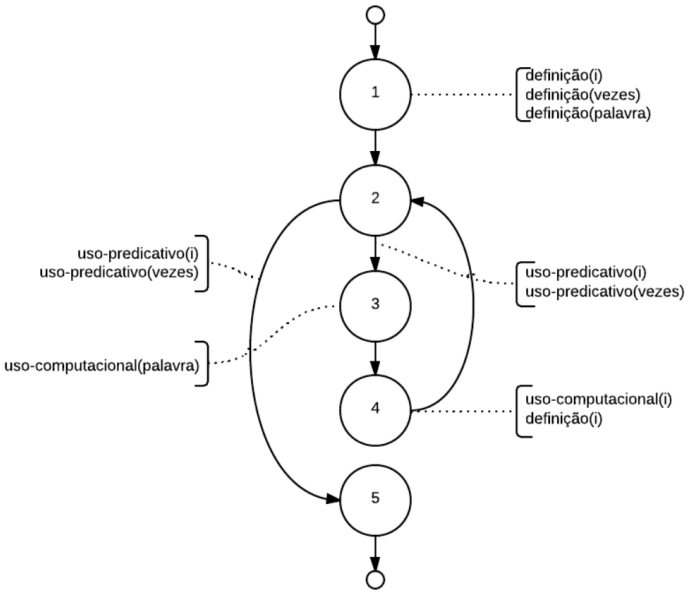
usada para determinar o fluxo de execução do programa (BARBOSA *et al.*, 2016). Com base no fluxo de dados, foi definido o caminho livre de definição, sendo um caminho entre uma definição de uma variável e seu posterior uso, sem a ocorrência de uma redefinição da variável.

A Figura 10 ilustra um exemplo de GFC com informações relacionadas ao fluxo de controle e fluxo de dados. Ele apresenta as informações extraídas do Código 2. O nó inicial 1 representa o início do bloco de código da função, contendo a definição das variáveis *palavra*, *vezes* e *i*. Os nós 2, 3, 4 e 5 representam a estrutura de repetição *while*, com a condicional representada no nó 2, e as suas derivações representada pelo uso predicativo das variáveis *i* e *vezes* nas arestas (2, 3) e (2, 5). O nó 3 possui o uso computacional da variável *palavra*, e o nó 4 a definição e uso computacional da variável *i*.

Código 2 – Exemplo de programa (PRADO, 2016).

```
1 #include <stdio.h>
2 /* 1 */ void imprime(char* palavra, int vezes) {
3 /* 1 */     int i = 0;
4 /* 2 */     while (i < vezes) {
5 /* 3 */         printf ("%s\n", palavra);
6 /* 4 */         i++;
7 /* 2 */     }
8 /* 5 */ }
```

Figura 10 – Grafo de fluxo de controle.



Fonte: Prado (2016)

Com base nas informações de fluxo de dados, as seguintes associações foram

definidas (BARBOSA *et al.*, 2016):

- **Associação c-uso**: É uma associação entre a definição de uma variável no nó n_i e seu subsequente c-uso no nó n_j, sendo definido pela tripla (n_i, n_j, x).
  - n_i: nó inicial onde foi definido a variável
  - n_j: nó onde ocorreu o uso computacional da variável
  - x: variável que foi definida e usada
- **Associação p-uso**: é uma associação entre a definição de uma variável no nó n_i e tem um p-uso na aresta (n_j, n_k), sendo definido pela tripla (n_i, (n_j, n_k), x).
  - n_i: nó inicial onde foi definido a variável
  - n_j: nó onde ocorreu o uso predicativo da variável
  - n_k: nó resultante do uso predicado da variável
  - x: variável que foi definida e usada

Fundamentado nas informações do GFC, os critérios de teste estruturais relacionados ao fluxo de controle e fluxo de dados foram definidos. Os critérios baseados em fluxo de controle são derivados dos controles de execução do programa, como o conjunto de comandos definidos pelos nós e os desvios de execução definidos pelas arestas. Os principais critérios baseados em fluxo de controle são (BARBOSA *et al.*, 2016):

- **Todos-nós**: requer que cada nó presente no GFC seja executado ao menos uma vez;
- **Todas-arestas**: requer que cada aresta presente no GFC seja executada ao menos uma vez;
- **Todos-caminhos**: requer que todos os caminhos possíveis do GFC sejam executados ao menos uma vez.

Os critérios baseados em fluxo de dados utilizam as informações relacionadas a definição de variável, ao uso computacional e predicativo e as associações entre definição e seu posterior uso. Neste, os principais critérios são (BARBOSA *et al.*, 2016):

- **Todos-p-usos**: Requer que todas as associações entre uma definição de variável e seus respectivos p-usos sejam exercitados, por ao menos um caminho livre de definição.
- **Todos-c-usos**: Requer que todas as associações entre uma definição de variável e seus respectivos c-usos sejam exercitados, por ao menos um caminho livre de definição.

- **Todas-definições**: Requer que toda definição de variável seja exercitada ao menos uma vez, seja por um c-uso ou p-uso.
- **Todos-usos**: Requer que toda associação entre definição e uso (c-uso / p-uso) de variável sejam exercitadas ao menos uma vez, por um caminho livre de definição.
- **Todos-Du-Caminhos**: Requer que toda associação entre definição e uso (c-uso / p-uso) de variável seja exercitada por todos os caminhos livres de definição e livres de laço.

Com base nos critérios de teste são definidos os elementos requeridos. Os elementos requeridos podem ser nós, arestas, caminhos, definições, usos e associações encontradas no programa, de acordo com o tipo de critério escolhido. Na atividade de teste, eles devem ser cobertos pelos casos de teste utilizados. Entretanto, nem todos os elementos requeridos são possíveis de serem cobertos, devido a alguns elementos serem não executáveis. Neste cenário, não há dados de teste que possibilitem a sua execução, devido as características do programa, como o uso de condicionais ou a redefinição de variável.

3.4 Teste Estrutural de Programas Concorrentes

Em geral, programas concorrentes possuem um maior grau de complexidade em relação aos programas sequenciais. Isso torna a atividade de teste também mais complexa, devido a características como o não-determinismo, concorrência entre processos e *threads*, sincronização e comunicação (SOUZA *et al.*, 2008). Essas características introduzem novos tipos de erros que não são cobertos pelas técnicas e critérios de teste voltadas à programas sequenciais.

Yang (1999) descreve alguns desafios para o teste de programas concorrentes como: (i) desenvolvimento de análise estática; (ii) detecção de condições de disputa e *deadlock* em programas não-determinísticos; (iii) forçar a execução de um caminho na presença do não-determinismo; (iv) reproduzir uma execução de teste usando os mesmos dados de entrada; (v) gerar o GFC de programas não-determinísticos; (vi) prover um ambiente de teste para aplicação de critérios de teste sequenciais para o teste programas paralelos; (vii) investigar a aplicabilidade de critérios de teste sequenciais em programas paralelos; e (viii) definir critérios de teste baseados em fluxo de controle e fluxo de dados.

Nesse sentido, Yang, Souter e Pollock (1998) apresentou o Grafo de Fluxo de Controle Paralelo (GFCP). Esse modelo foi estendido pelo projeto TestPar, permitindo a aplicação de critérios de teste voltados a programas concorrentes que utilizam o paradigma de passagem de mensagem (SOUZA *et al.*, 2008).

No modelo de teste proposto por Souza *et al.* (2008), um programa paralelo é representado por n processos paralelos, sendo $Prog = p^0, p^1, ..., p^{n-1}$. Cada processo p

possui o seu próprio GFC^p, que seguem os mesmos conceitos apresentados para o GFC de programas sequenciais. Todo GFC^p é composto por nós do conjunto N^p e por arestas do conjunto E^p. A ligação entre os nós de um mesmo processo são chamados de intra-processo. Um nó n^p de um processo pode estar associado a uma primitiva de comunicação *send* ou *receive*, sendo representadas pelas notações *send(p, k, t)* e *receive(p, k, t)*, indicando que um processo p envia ou recebe uma mensagem t de ou para um processo k.

Dois novos tipos de nós são definidos para o GFCP, o N_s e N_r. Eles representam os nós com primitivas *send* e *receive*, respectivamente. O conjunto de arestas E_s é definido, representando a comunicação entre dois processos diferentes. Para cada $n_i^p \in N_s$, um conjunto R_i^p é associado, contendo todos os nós que podem receber uma mensagem do nó n_i^p.

Nesse modelo, π^p é um caminho intra-processo, dado pela sequência de nós $\pi^p = (n_1^{p1}, n_2^{p1}, ..., n_{m-1}^{p1})$. $\Pi = (\pi^0, \pi^1, ..., \pi^k, S)$ é um caminho inter-processo, onde S se refere ao par de sincronização que foi executado, sendo $S \subseteq E_s$.

Nesse contexto, foram definidos os seguintes usos de variáveis:

- **uso computacional (c-uso)**: ocorre em uma computação relacionado a um nó N^p do GFCP.
- **uso predicativo (p-uso)**: ocorre em um uso predicativo, relacionado a uma aresta intra-processo (n^p, m^p) do GFCP.
- **uso comunicacional (s-uso)**: ocorre no uso da variável para comunicação, relacionado a arestas inter-processos $(n^{p1}, m^{p2}) \in E_s$.

Relacionando a definição e uso de variável, as seguintes associações foram definidas:

- **associação c-uso**: É uma associação entre a definição de uma variável no nó n^p e seu subsequente c-uso no nó m^p, sendo definido pela tripla (n^p, m^p, x).
- **associação p-uso**: é uma associação entre a definição de uma variável no nó n^p e um p-uso na aresta (m^p, k^p), sendo definido pela tripla $(n^p, (m^p, k^p), x)$.
- **associação s-uso**: é uma associação entre a definição de uma variável no nó n^{p1} e um s-uso na aresta (m^{p1}, k^{p2}), sendo definido pela tripla $(n^{p1}, (m^{p1}, k^{p2}), x)$.

Sendo a associação s-uso uma definição de variável e posterior comunicação entre processos, ele pressupõem a participam de um segundo processo, dessa forma as seguintes associações foram propostas:

- **associação s-c-uso**: é a associação entre a definição de uma variável no nó n^{p1}, seu posterior s-uso na aresta (m^{p1}, k^{p2}) e seu c-uso no nó l^{p2}, sendo definido pela quintupla (n^{p1}, (m^{p1}, k^{p2}), l^{p2}, x^{p1}, x^{p2}).

- **associação s-p-uso**: é a associação entre a definição de uma variável no nó n^{p1}, seu posterior s-uso na aresta (m^{p1}, k^{p2}) e seu p-uso na aresta (n^{p2}, m^{p2}), sendo definido pela quintupla (n^{p1}, (m^{p1}, k^{p2}), (n^{p2}, m^{p2}), x^{p1}, x^{p2})

Com base nas associações, foram derivados os seguintes critérios de fluxo de controle e fluxo de dados:

- **todos-nós-s**: requer a execução de todos os nós *send* do conjunto N_s.

- **todos-nós-r**: requer a execução de todos os nós *receive* do conjunto N_r.

- **todas-arestas-s**: requer a execução de todas as arestas inter-processos do conjunto E_s.

- **todos-s-usos**: requer a execução de todas as associações s-uso.

- **todos-s-c-usos**: requer a execução de todas as associações s-c-uso.

- **todos-s-p-usos**: requer a execução de todas as associações s-p-uso.

- **todas-defs**: deve executar ao menos uma associação c-uso, p-use ou s-uso para cada definição de variável.

- **todas-defs-s**: deve executar ao menos uma associação inter-processo (s-c-use ou s-p-use) para cada definição de variável.

3.4.1 Teste de programas concorrentes com memória compartilhada

Para o teste de programas concorrentes que utilizam o paradigma de memória compartilhada, Sarmanho *et al.* (2008) propôs o modelo de teste GFCPsm (Grafo de Fluxo de Controle Paralelo com Memória Compartilhada) para programas que utilizam PThreads. Neste modelo, MT = (t^0, t^1, ..., t^{n-1}) representa um programa *multithread* composto por *n threads t*. Cada *thread t* possui o seu próprio GFC^t, seguindo os mesmos conceitos aplicados nos modelos de teste apresentados anteriormente.

As sincronizações entre *threads* ocorrem pelas primitivas *post* e *wait*. Dessa forma, foram definidos os conjuntos N_p para nós que contenham a primitiva *post*, e N_w para os nós que contenham a primitiva *wait*. Para cada $n_i^t \in N_p$, um conjunto $\mathbf{M_w(n_i^t)}$ é associado, contendo os possíveis nós com *wait* que podem ser afetados pelo *post*. Da mesma forma, para cada $n_i^t \in N_w$, um conjunto $\mathbf{M_p(n_i^t}$ é associado, contendo todos os possíveis nós com *post* que podem ser afetados pelo *wait*. Ainda foi definido E_s, que representa todas as

combinações que formam pares de sincronização *post* e *wait* sobre uma mesma variável semáforo.

Considerando o fluxo de dados e a presença de múltiplas *threads* no programa, foram definidos conjuntos que representam os possíveis tipos de variáveis. Sendo V o conjunto de variáveis do programa, V_l é o conjunto de variáveis locais da *thread*. V_c são as variáveis compartilhadas, que possuem escopo global entre as *threads* e não são utilizadas para sincronização. V_s são as variáveis utilizadas na sincronização.

Considerando o fluxo de dados e a comunicação entre as *threads*, foram definidos os seguintes caminhos:

- **intra-thread**: quando não há aresta de sincronização, dessa forma, o caminho ocorre em uma única thread. Detonado por $\pi^t = \{n_1^t, n_2^t, ..., n_j^t\}$.
- **inter-thread**: possui ao menos uma aresta de sincronização. Denotado por $\Pi = (PATHS, SYNCS)$, onde $PATHS = \{\pi^1, \pi^2, ..., \pi^n\}$ e $SYNCS = \{(p_i^t, w_j^q) \mid (p_i^t, w_j^q) \in E_s\}$, sendo p um nó *post* na *thread* t e w um nó *wait* na *thread* q.

Foram definidos cinco tipos de uso de variáveis no contexto de memória compartilhada com PThreads, sendo:

- **uso computacional (c-uso)**: variável local utilizada para realizar alguma computação, onde x $\in V_l^t$.
- **uso predicativo (p-uso)**: variável local utilizada para determinar o fluxo de controle do programa, onde x $\in V_l^t$.
- **uso de sincronização (sync-use)**: uso de variável em instruções de sincronização, onde $x \in V_s^t$.
- **c-uso comunicacional (comm-c-use)**: uso computacional de variável compartilhada, onde $x \in V_c^t$.
- **p-uso comunicacional (comm-p-use)**: uso predicativo de variável compartilhada, onde $x \in V_c^t$.

Por meio das definições, foram derivadas as seguintes associações entre definição e uso de variável:

- **associação c-uso**: É uma associação entre a definição de uma variável no nó n_i^t e seu subsequente c-uso no nó n_j^t, sendo definido pela tripla (n_i^t, n_j^t, x).
- **associação p-uso**: é uma associação entre a definição de uma variável no nó n_i^t e um p-uso na aresta (n_j^t, n_k^t), sendo definido pela tripla (n_i^t, (n_j^t, n_k^t), x).

- **associação sync-use**: é uma associação entre a definição de uma variável no nó n_i^t e um sync-use na aresta (n_j^t, n_k^t), sendo definido pela tripla $(n_i^t, (n_j^t, n_k^q), \text{s})$, onde $s \in V_s$.

- **associação comm-c-use**: é a associação entre a definição de uma variável compartilhada no nó n_i^t e seu posterior c-uso no nó n_j^q, sendo definido pela tripla (n_i^t, n_j^q, x), onde $x \in V_c$.

- **associação comm-p-use**: é a associação entre a definição de uma variável compartilhada no nó n_i^t e seu posterior p-uso na aresta (n_j^q, n_k^q), sendo definido pela tripla $(n_i^t, (n_j^q, n_k^q), \text{x})$, onde $x \in V_c$.

Por meio das definições e associações, foram definidos os seguintes critérios de teste baseados em fluxo de controle e fluxo de controle, sincronização, fluxo de dados e comunicação:

- **Todos-p-nós**: requer que todos os nós *post* sejam executados, sendo $n_i^t \in N_p$.

- **Todos-w-nós**: requer que todos os nós *wait* sejam executados, sendo $n_i^t \in N_w$.

- **Todos-nós**: requer que todos os nós sejam executados, sendo $n_i^t \in N$.

- **Todas-s-arestas**: requer que todas as arestas de sincronização sejam executadas, sendo $(n_i^t, n_j^q) \in E_s$.

- **Todas-arestas**: requer que todas as arestas sejam executadas, sendo $(n_i^t, n_j^q) \in E$.

- **Todas-defs-com**: deve executar ao menos uma associação comm-c-use ou comm-p-use para cada definição de variável $x \in V_c^t$.

- **Todas-defs**: deve executar ao menos uma associação c-uso, p-use, comm-c-use ou comm-p-use para cada definição de variável.

- **Todas-com-c-usos**: deve executar todas as associações comm-c-use.

- **Todas-com-p-usos**: deve executar todas as associações comm-p-use.

- **Todos-c-usos**: deve executar todas as associações c-uso.

- **Todos-p-usos**: deve executar todas as associações p-uso.

- **Todos-sinc-usos**: deve executar todas as associações sync-use.

3.5 ValiPar

A ferramenta ValiPar vem sendo desenvolvida pelo projeto TestPar e tem por objetivo a implementação, aplicação e validação de modelo e critérios de teste estrutural para programas concorrentes (PRADO *et al.*, 2015). Ela possui versões com suporte ao teste de programas com paradigma de passagem de mensagem como BPEL, PVM, C/MPI e Erlang com as ferramentas ValiBPel-WEB (ENDO *et al.*, 2008), ValiPVM (SOUZA *et al.*, 2008), ValiMPI (HAUSEN, 2005; SOUZA *et al.*, 2005; SOUZA; SOUZA; ZALUSKA, 2014; DIAZ; SOUZA; SOUZA, 2019) e ValiErlang (OLIVEIRA; SOUZA; SOUZA, 2016), respectivamente. Suporta ainda o paradigma de memória compartilhada com C/PThreads na ValiPThread (SARMANHO *et al.*, 2008). Possuindo suporte aos diversos critérios de teste estruturais apresentados.

Para a implementação do modelo e critérios de teste propostos por nós foi selecionado a versão ValiMPI. No momento ele apresentava os seguintes critérios implementados: todos-nos, todas-arestas, todos-nos-r, todos-nos-s, todas-arestas-s, todos-c-usos, todos-p-usos, todos-eventos-s-loop, todos-eventos-r-loop, todos-sinc-eventos-loop, todos-s-usos-loop, todos-s-c-usos-loop, todos-s-p-usos-loop, e todas-def-recv-loop (HAUSEN, 2005; SOUZA *et al.*, 2005; SOUZA; SOUZA; ZALUSKA, 2014; DIAZ; SOUZA; SOUZA, 2019).

A arquitetura da ferramenta ValiMPI é apresentada na figura 11. Com exceção da ValiErlang, todas as versões possuem a mesma arquitetura. A ferramenta possui quatro módulos principais, sendo eles a ValiInst, ValiExec, ValiElem e ValiEval.

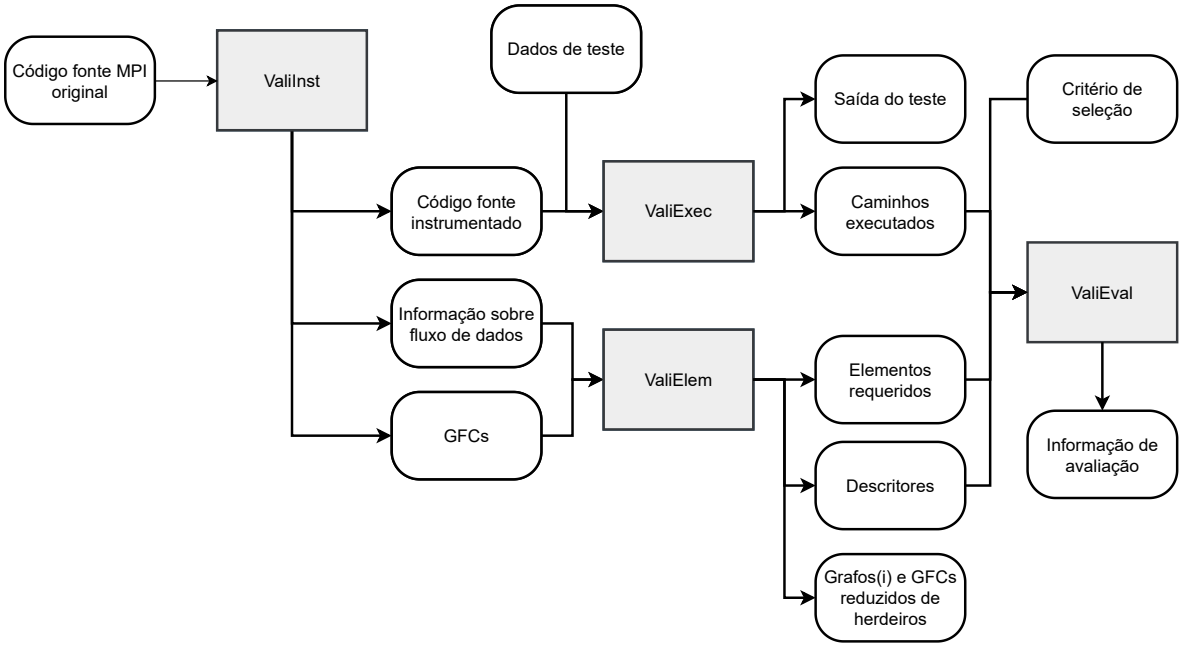
3.5.1 ValiInst

O módulo ValiInst possui a finalidade de instrumentar e extrair as informações de fluxo de controle e fluxo de dados. Como ilustrado na Figura 12, o módulo recebe como entrada o código fonte MPI original do programa livre de erros sintáticos, e retorna como saída três artefatos: o código fonte MPI instrumentado, informações sobre o fluxo de dados, e os Grafos de Fluxo de Controle (GFCs) referente aos processos do programa.

O código fonte instrumentado possui comandos do tipo *check-point* para permitir a identificação do rastro de execução. Inicialmente são removidos os comentários do código, e após são inseridos os comandos para a criação do rastro de execução baseados nos nós do GFC, permitindo identificar quais parcelas do código foram executadas. Também substitui algumas chamadas de funções para identificar as sincronizações e permitir a execução controlada dos processos.

Um GFC é criado para cada processo do programa. Cada grafo é criado em arquivo no formato DOT, que permite a representação visual do grafo por meio da ferramenta Graphviz. Neste arquivo são inseridos em formato de texto os nós e arestas referentes ao fluxo de controle do processo. São inseridos como comentários as definições e usos

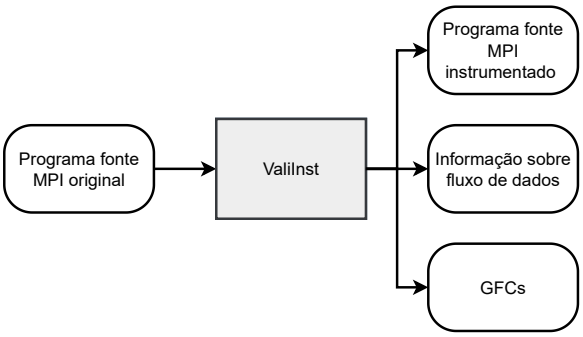
Figura 11 – Arquitetura de módulos da ferramenta ValiMPI.



Fonte: Adaptado de Hausen (2005)

computacionais, predicativos e comunicacionais das variáveis. O número de GFC gerado depende da quantidade de processos que o programa possui. Não é construído um PCFG com os grafos de todos os processos, sendo necessário analisar as sincronizações informadas nos grafos de cada processo para relacioná-los.

Figura 12 – Módulo ValiInst.



Fonte: Hausen (2005)

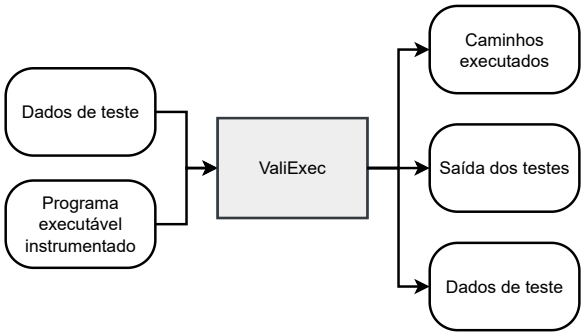
3.5.2 ValiExec

A ValiExec, apresentada na Figura 13, é composta por um conjunto de três *scripts*, Vali-cc, Vali-clean e Vali-exec. A Vali-cc recebe o código fonte instrumentado, faz a ligação

com as bibliotecas de perfilamento do MPI, necessárias para a geração do rastro e execução controlada, e compila o código, resultando no programa executável instrumentado. A Vali-clean tem a finalidade de apagar os arquivos de execuções anteriores dos casos de teste.

A Vali-exec recebe como entrada os dados de teste e o programa executável instrumentado. Ela inicia o ambiente paralelo e executa o código com base nas entradas fornecidas. Ao final da execução, a Vali-Exec gera como saída os próprios dados de teste utilizados, os rastros dos caminhos executados para cada caso de teste, e a saída dos testes. Com os valores da saída, o usuário pode averiguar se a saída obtida é igual à esperada. Caso seja diferente, um erro foi revelado e o usuário pode utilizar os artefados gerados pela execução para auxiliar no processo de depuração para encontrar o defeito.

Figura 13 – Módulo ValiExec.



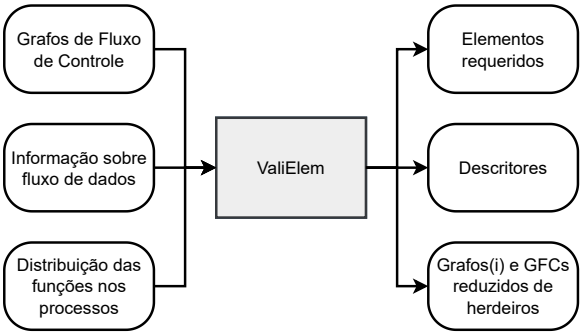
Fonte: Hausen (2005)

3.5.3 ValiElem

A ValiElem tem o papel de gerar os elementos requeridos (Figura 14). Para isso, ela recebe os GFCs gerados pela ValiInst contendo as informações sobre o fluxo de controle por meio dos nós e arestas, o fluxo de dados por meio da definição e uso das variáveis, e a distribuição das funções em teste entre os processos. A partir disso são gerados os elementos requeridos para os critérios implementados, assim como os descritores dos elementos e dois outros grafos, o grafo reduzido de herdeiros e o grafo(i).

O grafo reduzido de herdeiros implementa o conceito de arestas primitivas (CHUSHO, 1987). O conceito tem por finalidade diminuir o número de elementos requeridos para o teste e diminuir o número de arestas para representar o GFC (DIAZ; SOUZA; SOUZA, 2019), dessa forma diminuindo o custo para a atividade de teste. Esse conceito se baseia no fato que algumas arestas do GFC sempre serão executadas quando determinados arcos forem executados. As arestas que sempre são executadas quando outra aresta é executada são chamadas de arestas herdeiras ou não-essenciais, representadas por h na Figura 15, e

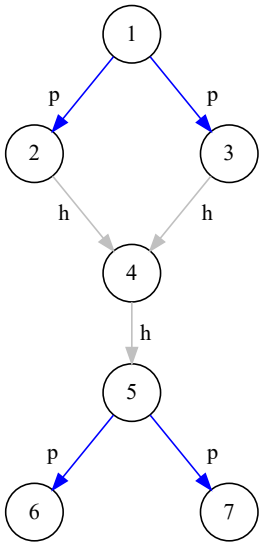
Figura 14 – Módulo ValiElem.



Fonte: Hausen (2005)

as outras são chamadas de arestas primitivas ou essenciais, representadas por p.

Figura 15 – GFC com arestas primitivas p e herdeiras h.



Fonte: Adaptado de Chaim, Maldonado e Jino (1991)

O grafo(i) é utilizado pelos critérios de fluxo de dados. Ele estabelece as associações entre as definições e uso de variáveis por um caminho livre de definição. Um grafo(i) é construído para cada nó n_i^p que possua uma definição de variável e, um nó n_j^p pertencerá ao grafo se existir um caminho entre eles livre de definição com relação a uma variável definida em n_i^p. Sendo representado em uma estrutura em árvore, o grafo(i) não possui laços de nós. Para evitar a ocorrência de caminhos infinitos, ao encontrar um nó já existente no ramo, o caminho é interrompido.

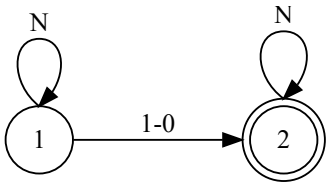
A ValiElem gera os elementos requeridos e para cada elemento é criado um descritor. O descritor é uma expressão regular utilizada para descrever o caminho necessário para

cobrir o elemento requerido. Cada critério de teste possui uma expressão regular definida, por exemplo, a expressão regular para o critério todos-nós é dada por *N* ni-p N**, que indica que ele será coberto se no processo p o caminho executado incluir o nó n_i. Para o critério todas-arestas a expressão regular é *N* ni-p nj-p N**, sendo coberto em caminhos em que os nós n_i e n_j do processo p forem executados consecutivamente. Os descritores são representados por um autômato finito determinístico (AFD) que contem em formato texto os seus estados, número de estados, transições, estados finais e número de autômatos. O Código 3 e a Figura 16 apresentam o descritor e o autômato utilizado para reconhecer o nó 1 do processo 0 do critério todos-nós. A Figura 17 ilustra como é realizada a leitura e interpretação do descritor do autômato presente no Código 3. O descritor inicia com valor 1, pois possui apenas um autômato. O próximo número indica a quantidade de estados presentes no autômato, que, conforme mostrado na Figura 16, é composto por 2 estados, sendo o segundo deles o estado final. Em seguida, é indicado o número do estado e suas possíveis transições. No primeiro estado, o autômato permanece no mesmo estado enquanto executa qualquer nó diferente do nó 1. Ao executar o nó 1, ocorre a transição para o segundo estado. No segundo estado, há apenas uma única transição, que mantém o autômato no estado final para qualquer nó executado.

Código 3 – Descritor do autômato para o critério todos-nós.

```
1 1 ) 1 2 2 1 1:N/1-0 2:1-0 0 2 2:N 0
```

Figura 16 – Autômato do critério todos-nós para reconhecer o elemento requerido 1-0.



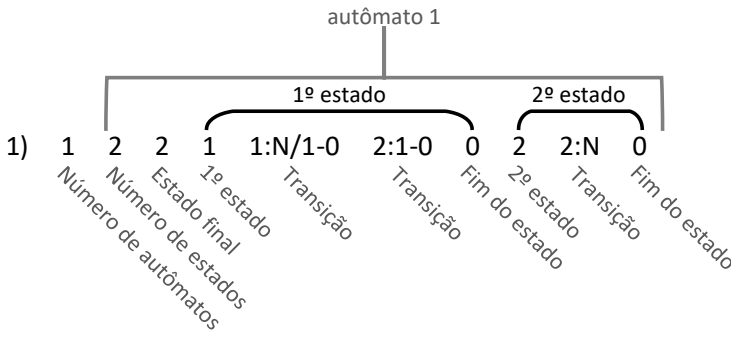
Fonte: Elaborado pelo autor.

3.5.4 ValiEval

O módulo ValiEval da Figura 18 avalia a cobertura alcançada pelos casos de teste. Ele recebe como entrada o critério de teste selecionado para avaliar a cobertura. Com base no critério são carregados os elementos requeridos, descritores gerados pela ValiElem para o programa em teste e os caminhos executados. Ao final, ele apresenta os resultados com a cobertura alcançada pelos casos de teste para o critério selecionado.

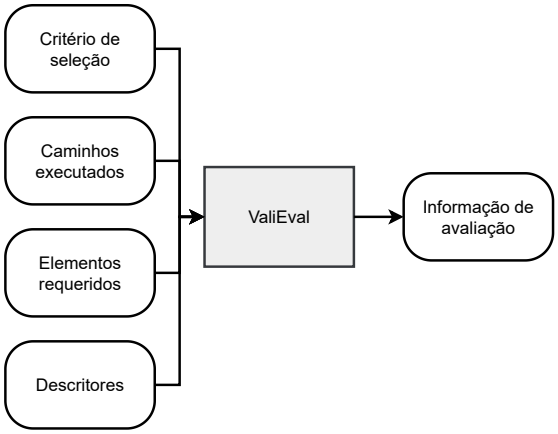
Com base nos descritores recebidos, a ValiEval constrói o autômato utilizado para averiguar se o elemento foi coberto. Para cada autômato, ela carrega os rastros dos caminhos executados para cada processo e verifica se o autômato é coberto por um deles.

Figura 17 – Legenda dos elementos que compõem o descritor do autômato do critério todos-nós.



Fonte: Elaborado pelo autor.

Figura 18 – Módulo ValiEval.



Fonte: Hausen (2005)

Se ao final do processo, o autômato estiver em um estado final, significa que o elemento requerido representado por esse autômato foi coberto pelo caso de teste.

Ao fim da execução de todos os casos de teste, é apresentado o relatório de cobertura alcançado. No relatório são apresentados todos os elementos requeridos cobertos, especificando qual foi o primeiro caso de teste a atingir a cobertura, e todos os elementos requeridos não cobertos pelo teste. Um percentual de cobertura para o critério é apresentado, sendo calculado pela relação do número de elementos requeridos cobertos, dividido pelo total de elementos requeridos do critério. Com base no relatório de elementos não cobertos, é possível ao usuário construir novos casos de teste que busquem alcançar a cobertura para esses elementos, melhorando dessa forma a qualidade do teste.

3.6 Considerações finais

Descrevemos os principais conceitos relacionados ao teste de *software*, com foco na técnica de teste estrutural. Apresentamos os critérios de teste básicos para programas sequências, que são utilizados como fundação para este trabalho. Também mostramos modelos e critérios de teste para programas concorrentes com paradigma de memória compartilhada e passagem de mensagem, conceitos análogos ao modelo de programação CUDA. Ao final, discorremos sobre a ferramenta ValiPar, com suporte a diferentes paradigmas e linguagens de programação. Destacamos a versão ValiMPI, evidenciando a sua arquitetura e o funcionamento de cada módulo que a compõe. Parte desses módulos foram modificados e utilizados para o desenvolvimento da ferramenta ValiCUDA.

4 PRINCIPAIS TIPOS DE DEFEITOS EM PROGRAMAS CUDA

4.1 Considerações iniciais

Neste capítulo nós apresentamos alguns defeitos típicos relacionados a programas concorrentes que utilizam o modelo de programação CUDA. Na Seção 4.2 são listados os tipos de defeitos considerando trabalhos relacionados e apresentamos a classificação de defeitos utilizada. Na Seção 4.3 há nossas considerações finais sobre a relação de defeitos.

4.2 Categorias de defeitos em CUDA

Para identificar os tipos de defeitos comuns em aplicações CUDA nós consultamos diferentes tipos de materiais. Verificamos em trabalhos relacionados, por meio de uma revisão da área, buscando por trabalhos relacionados a teste de programas voltadas ao uso de GPU. Consultamos livros sobre programação específica de programas CUDA e, por fim, verificamos o guia de referência de programação da Nvidia.

O foco da busca foram por defeitos que pudessem causar um erro no programa, entretanto, nos trabalhos relacionados foram identificados dois grupos principais de defeitos: defeitos relacionados ao código fonte e defeitos relacionados ao desempenho. Defeitos de desempenho afetam o tempo de execução e/ou resposta da aplicação, enquanto o defeito relacionado ao código fonte afeta a saída apresentada pela aplicação, conforme apresentado no capítulo 3. Os principais defeitos de desempenho identificados foram:

- **conflito de banco** (*bank conflict*): pode ocorrer em programas que utilizam memória compartilhada em nível de bloco. A memória cache utilizada pelas variáveis compartilhadas em nível de bloco é organizada em bancos. Quando múltiplos acessos ocorrem no mesmo banco, um conflito de banco pode ocorrer. Neste caso, a requisição é dividida em um número de transações livre de conflitos de banco necessárias para atender à solicitação, resultando em uma redução na largura de banda efetiva por um fator igual ao número de transações de memória separadas que foram necessárias. (CHENG; GROSSMAN; MCKERCHER, 2014);

- **acesso de memória não coalescente** (*non-coalesced memory access*): um acesso de memória não coalescente ocorre quando várias *threads* em um *warp* acessam locais de memória não contíguos, resultando em múltiplas transações de memória, causando uma redução significativa no desempenho do programa. O acesso coalescente de memória ocorre quando várias *threads* no mesmo *warp* acessam locais de memória contíguos. Essa técnica pode melhorar significativamente o desempenho do programa, pois as solicitações de várias *threads* podem ser satisfeitas em uma única transação

de memória, reduzindo assim o número total de transações necessárias (CHENG; GROSSMAN; MCKERCHER, 2014);

- **divergência de ramo** (*branch divergence*): ocorre quando duas ou mais *threads* de um mesmo *warp* precisam seguir diferentes caminhos de execução, causando a serialização das *threads*. Isso ocorre devido à característica de arquitetura SIMD que a GPU possui, que exige que as *threads* de um *warp* em um SM executem a mesma instrução. Com o conceito de arquitetura SIMT, essa exigência é flexibilizada, mas serializa as execuções das *threads*, degradando o desempenho final (COUTINHO *et al.*, 2012).

Os defeitos relacionados ao código fonte foram:

- **violação de atomicidade**: uma região crítica que deveria ser atômica, mas não usa nenhuma primitiva de sincronização que garanta o acesso de memória atômica nessa região (LU *et al.*, 2008).

- ***deadlock***: pode acontecer quando em um grupo de *threads* há uma dependência cíclica, onde cada *thread* está esperando por outra, que está bloqueada, para poder liberar o *lock*. Um exemplo disso é na divergência de barreira, onde *threads* do mesmo grupo podem divergir e chegar em barreiras de sincronização diferentes. Neste caso, o comportamento é indefinido pela Nvidia (BETTS *et al.*, 2012a), podendo ocorrer o *deadlock*.

- **erros de memória**: ocorre quando não se restringe o acesso das *threads* aos limites da estrutura de dados da variável, como um vetor ou matriz. Em um programa para GPU, é comum que o número de *threads* seja diferente do tamanho dos dados, dessa forma, não ocorrendo um mapeamento um por um entre os dados e as *threads*. Outro caso é quando a *thread* tenta acessar uma variável que não foi inicializada (COLLINGBOURNE; CADAR; KELLY, 2012).

- **condição de disputa**: (NETZER; MILLER, 1992) e Regehr (2011) definem dois tipos de condições de disputa: disputa geral e disputa de dados. A primeira ocorre quando o resultado depende da ordem dos eventos. A segunda ocorre quando o não-determinismo em regiões críticas afetam o resultado. Se não há o uso de primitivas de sincronização, o controle de acesso à Região Crítica por essas instruções não são garantidas, o que pode resultar em um comportamento incorreto.

- **falta de qualificador *volatile***: o compilador realiza otimizações nas operações de leitura e escrita na memória global e compartilhada em nível de bloco, por exemplo, armazenando os dados nos registradores ou em caches de diferentes níveis, enquanto mantém a ordem semântica e as sincronizações. O qualificador *volatile* desativa essas

otimizações, assumindo que o valor pode mudar ou ser usado a qualquer momento por outras *threads*, fazendo com que cada acesso de memória relacionado à variável resulte em uma leitura ou escrita direta à memória (NVIDIA, 2022).

Concomitantemente, Zhu e Zaidman (2020) propuseram alguns operadores de mutação para inserção de defeitos em programas que utilizam o modelo de programação concorrente CUDA. classificaram os operadores nas seguintes categorias:

- **gerenciamento de memória** (*memory management*):
  - **configuração da execução** (*execution configuration*): relacionada à configuração de execução do *kernel*, definindo as dimensões do *grid* e dos blocos de *thread* e, por consequência, a quantidade total de *threads*. Três operadores de mutação foram propostos: *alloc_swap*, *alloc_decrement* e *alloc_increment*.
  - **memória compartilhada** (*shared memory*): está relacionada com o uso da memória compartilhada em nível de bloco da GPU, onde se define o uso desse espaço pela variável por meio da primitiva "___shared___". Foi proposto o operador de mutação *share_removal*.
- **gerenciamento de *thread*** (*thread management*):
  - **indexação de GPU** (*GPU indexing*): relacionada com o método de definição do identificador (ID) da *thread*, onde são utilizadas as variáveis "threadIdx", "blockIdx" e "gridIdx" para definir o identificador. Ela implica no mapeamento da *thread* em relação a estrutura de dados. Três operadores de mutação foram propostos para essa categoria, sendo: *gpu_index_replacement*, *gpu_index_increment* e *gpu_index_decrement*.
  - **funções de sincronização** (*synchronisation functions*): relacionada com o uso das barreiras de sincronização que são utilizadas para coordenar o acesso às variáveis pelas *threads* de um mesmo bloco. Para essa categoria foi proposto o operador de mutação "sync_removal".
- **operações atômicas** (*atomic operations*): relativa à atomicidade de uma operação em uma região crítica, onde somente uma *thread* por vez pode acessar o recurso, garantindo a semântica da operação e sincronização das *threads*. O operador de mutação "atom_removal" foi proposto para essa categoria.

Com base nos tipos de defeitos abordados nos trabalhos e na classificação dos operadores de mutação feitos por, Zhu e Zaidman (2020), nós utilizamos a seguinte classificação para os defeitos:

- **erros de memória**: relacionado aos acessos fora dos limites de uma variável, como um vetor ou matriz, ou quando a variável não foi inicializada antes da tentativa de leitura do seu valor.
- **falta de sincronização**: engloba os defeitos relacionados à sincronização, como *deadlock* e condições de disputa listadas anteriormente.
- **violação de atomicidade**: ocorre quando o código possui uma região crítica que deveria ser implementada como atômica, mas não são utilizadas primitivas atômicas em sua implementação.
- **ID incorreto da *thread***: quando o identificador da *thread* é calculado incorretamente, causando um mapeamento incorreto entre as *threads* e as estruturas de dados.
- **erros de variável compartilhada**: quando uma variável que deveria ser compartilhada em nível de bloco não possui a primitiva "___shared___", tornando ela uma variável local.
- **configuração incorreta do *kernel***: ocorre quando o número de blocos por *grid* e/ou o número *threads* por bloco é configurado incorretamente, ou são invertidos.
- **falta de qualificador *volatile***: quando não é usada a primitiva *volatile* em uma variável que é utilizada em um contexto onde as otimizações de cache precisam ser desativadas.
- **uso incorreto de variável global**: este erro está relacionado à manipulação incorreta das variáveis globais do *device* no *host*, como a cópia incorreta de valores, o uso da variável errada, a definição errada do tamanho da variável e a inversão de variáveis em um "cudaMemcpy" são alguns exemplos desse tipo de erro.

Como os defeitos relacionados ao desempenho não são o foco deste trabalho, eles não foram adicionados nessa relação de defeitos. Da mesma forma, essa relação de defeitos de desempenho pode estar incompleta, por não termos utilizado termos relacionados ao desempenho na pesquisa, podendo ser usado apenas como uma referência para esse tipo de defeito.

4.3 Considerações finais

Nós apresentamos os tipos de defeitos comumente encontrados em programas concorrentes CUDA. Mostramos que há duas linhas principais de defeitos que afetam programas CUDA, sendo uma relacionada ao desempenho, e a outra que pode causar uma falha (uma saída contendo resultado incorreto do programa). Como o foco deste

trabalho são defeitos relacionados à segunda linha, nós apresentamos uma classificação de operadores de mutação para CUDA proposta por Zhu e Zaidman (2020) e, com base nos tipos de defeitos encontrados nos trabalhos relacionados, apresentamos uma classificação de defeitos englobando os defeitos encontrados, e acrescentando sobre o uso incorreto de variável global.

5 TESTE ESTRUTURAL PARA PROGRAMAS CUDA

5.1 Considerações Iniciais

Neste capítulo nós apresentamos a proposta do modelo e critérios de teste desenvolvidos para o teste de cobertura de programas CUDA. O modelo de teste captura informações relacionadas ao fluxo de controle, fluxo de dados e comunicação, considerando as características sequenciais e paralelas do modelo de programação. Ele define diretrizes para a recuperação de informações do programa, compreendendo a análise estática para a captura das informações que viabilizarão a definição dos critérios de teste e dos elementos requeridos com base em cada critério de teste.

Na seção 5.2 é apresentado o modelo de teste para programas CUDA. A seção 5.3 propõe novos critérios de teste para CUDA A seção 5.4 demonstra um exemplo de caso de teste utilizando o modelo de teste proposto, e a seção 5.5 apresenta as considerações finais deste capítulo.

5.2 Modelo de Teste PCFG para CUDA

O modelo de teste proposto para CUDA foi baseado em dois modelos pré-existentes, o PCFG (*Parallel Control Flow Graph*) e o PCFGsm (*Parallel Control Flow Graph for Shared Memory*). O PCFG foi proposto em (SOUZA *et al.*, 2005) e utilizado no teste de programas concorrentes com passagem de mensagem, apresentando um modelo que suporta as sincronizações, inicialização e finalização de *threads*, um processo que ocorre de forma similar entre o *host* e *device* no modelo CUDA. Um outro trabalho buscou estender o PCFG para oferecer suporte a outras primitivas de passagem de mensagens (SOUZA; SOUZA; ZALUSKA, 2014), dentre elas a barreira de sincronização, que é análogo às sincronizações ocorridas dentro do *kernel* no CUDA. O PCFGsm proposto em (SARMANHO *et al.*, 2008) é voltado para programas concorrentes com memória compartilhada no contexto de PThreads. O conceito é similar à memória compartilhada por bloco e global apresentado em CUDA.

O PCFG para CUDA busca viabilizar a atividade de teste em programas CUDA considerando as características específicas deste modelo de programação que prevê, por exemplo, o uso de um número muito maior de *threads*, em comparação com os modelos de teste prévios até então desenvolvidos para passagem de mensagens e memória compartilhada. Este novo modelo de teste apresenta uma representação em grafo do programa, incluindo o *host* e *device* com suas interações. O grafo apresenta o fluxo de controle do programa, informações relacionadas ao fluxo de dados, comunicação entre o código do *host* e o código do *device*, e as sincronizações realizadas dentro do *kernel*. A Figura 19 apresenta um

exemplo do grafo de um programa CUDA de exemplo dividido entre *host* e *grid*.

O modelo permite analisar sintaticamente o código buscando extrair as informações relevantes do programa. As informações extraídas tendo como base os critérios de teste, permitem definir os elementos requeridos para a realização dos testes desejados. Com base nos elementos requeridos, guia-se a criação de novos casos de teste para aumentar a cobertura do programa CUDA sob teste.

O PCFG para CUDA estendeu os modelos utilizados como base. Em modelos de programação como PThreads, OpenMP e MPI, um programa normalmente possui algumas dezenas de *threads* e processos, com alguns casos possuindo centenas de *threads*. Em CUDA, o modelo permite que um programa possua milhares até milhões de *threads* no *device*.

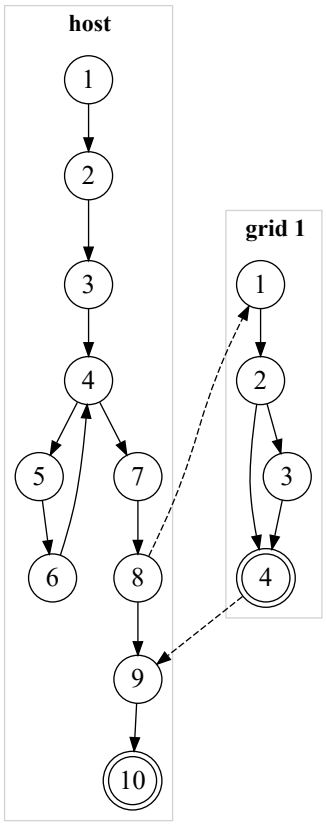
Nos outros modelos cada processo e *thread* possui seu próprio CFG, o que é inviável para o modelo de teste para CUDA com milhões de *threads*. No PCFG para CUDA foi considerada a hierarquia de *threads* oferecida, onde as *threads* são agrupadas em blocos, os blocos são agrupados em *grids*, que por sua vez representam a execução de um *kernel*. Dessa forma, o *grid* possui seu próprio CFG, representando todas as *threads* que o compõe. A Figura 19 possui um PCFG para CUDA de um programa para exemplificação, onde há um CFG para o *host* que possui um único processo e uma única *thread* e o *grid*, que possui um único CFG. Embora todas as *threads* do *device* sejam representadas por um único CFG, os rastros gerados pela execução do programa apresentam granularidade em nível de *thread*, permitindo que a cobertura seja avaliada neste mesmo nível. Esta organização foi planejada para dar escalabilidade ao modelo de teste em função da demanda apresentada por programas CUDA.

5.2.1 PCFG para CUDA

O modelo de teste busca recuperar informações relacionadas ao fluxo de controle, fluxo de dados e comunicação entre processos e *threads*. Ele é baseado em (RAPPS; WEYUKER, 1982), representando o programa por meio de nós e arestas. O fluxo de dados leva em consideração as variáveis, o seu escopo e a sua utilização. A comunicação implica em sincronização entre o *host* e o *grid*, com ou sem utilização de variáveis.

No modelo, o programa é representado por $Prog = \{h, g^1, g^2, ..., g^n\}$. O *host* possui o seu próprio CFG^h e cada *grid* possui o seu próprio CFG^{g^n}. O programa possui um único processo *host* com uma única *thread* e n é o número de *grids*, os quais podem representar milhões de *threads* sendo executadas no *device*. Dessa forma, no contexto do *host* não são abordadas variáveis compartilhadas. No caso do *grid* as variáveis podem ter escopo local, compartilhada no bloco ou compartilhada em nível global, entre todas as *threads* do *grid*.

Figura 19 – PCFG de um programa CUDA de exemplo.



Fonte: autor

No contexto de *Prog* temos N representando o conjunto de nós do PCFG. Cada nó corresponde a uma instrução ou conjunto de instruções do programa onde não há mudanças no fluxo de controle. N é composto pelo conjunto de nós N^h do CFG^h e N^{g^n} do CFG^{g^n}. Quando o nó possui uma instrução de sincronização ele é representado pelo conjunto N_{sync}, que é composto por N^{grid}_{sync} e N^{host}_{sync}, onde $N^{grid}_{sync} \in N^{g^n}$ e $N^{host}_{sync} \in N^h$. Nós com primitivas específicas do CUDA e executadas no *host* são representadas pelo conjunto N^{host}_{sync} devido a elas possuírem comportamento síncrono.

As arestas E interligam os nós, sendo divididas entre as arestas *intra-threads* e arestas *inter-threads*. As arestas *intra-threads* são divididas entre o conjunto de arestas do *host* E^h e o conjunto de arestas do *grid* E^{g^n}. As arestas *inter-threads* são representadas pelo E_{sync}, que compõe as arestas de sincronização que ligam o *host* com os *grids* quando ocorre a execução e finalização do *kernel*.

5.2.2 Sincronização

A sincronização em programas CUDA ocorre no contexto do *host*, do *grid* e na comunicação entre os dois. As sincronizações dentro do *host* são representadas pelos nós

N^{host}_{sync} e funcionam por meio da espera do *host* até a finalização do *grid*. As sincronizações do *grid* são representadas pelos nós N^{grid}_{sync}, enquanto as sincronizações da comunicação entre *host* e *grid* são representadas pelo E_{sync} e apresentada na Figura 19 pela seta de linha tracejada ligando os dois CFGs. O nó da chamada do *kernel* está incluso no conjunto N^{host}_{sync}.

No contexto do *host* há sincronização implícita e explícita. Quando há apenas uma única *thread* no *host*, apenas as primitivas CUDA podem ocasionar sincronização entre o *host* e o *grid*. As sincronizações explícitas funcionam como uma barreira, onde todas as *threads* do *host* e *grid* irão esperar até que todas executem a primitiva. A primitiva *cudaDeviceSynchronize* é um exemplo de sincronização explícita, ela bloqueia o *host* na expectativa da finalização de todos os *grids* em execução. A primitiva *cudaStreamSynchronize* é outro exemplo, ela bloqueia o *host* até que o *grid* em específico finalize a execução de suas tarefas.

No *host* o tipo de alocação de memória utilizada influencia na sincronização. Há duas formas de alocação de memória. No primeiro há dois espaços de endereçamento de memória, espaço do *host* e o espaço do *grid*. O *grid* possui acesso apenas à memória da GPU. Para que ele tenha acesso ao conteúdo alocado no *host* é necessário: (1) alocar memória do *host*, (2) atribuir conteúdo ao *host*, (3) alocar memória para as variáveis do *grid* e (4) fazer a cópia do conteúdo do *host* para o *grid*. O Código 4 apresenta a alocação do *host* na linha 5, a alocação do *grid* na linha 7, e as cópias de conteúdo nas linhas 12 e 14. Neste exemplo, as sincronizações são implícitas, ocorrendo nas cópias de conteúdo e execução do *kernel*. No segundo tipo de alocação há um único espaço de memória para o *host* e *grid* chamado Memória Unificada (HARRIS, 2017), que remove a necessidade de explicitamente copiar o conteúdo com o *cudaMemcpy*. Sem o *cudaMemcpy* após a execução do *kernel*, não há garantia de sincronização antes do *host* acessar os dados, sendo necessário usar uma sincronização explícita como o *cudaDeviceSynchonize*. Apesar das diferenças em termos de programação, internamente o funcionamento dessas memórias é o mesmo, o conteúdo continua sendo copiado de uma memória para outra (HARRIS, 2017). Dessa forma, o modelo PCFG para CUDA representa ambas as formas de alocação de memória como nós N^{host}_{sync}, onde $N^{host}_{sync} \in N$.

Código 4 – Código de exemplo do *host*.

```
1  int main (int argc, char* argv[]) {
2      int N = 4;                                                    // 1
3      size_t size = N*sizeof(int);                                  // 1
4      int *h_a;                                                     // 1
5      h_a = (int*)malloc(size);                                     // 1
6      int *d_a;                                                     // 1
7      cudaMalloc(&d_a, size);                                       // 2
8      int i = 0;                                                    // 2
9      for (i = 0; i < N; i++)                                       // 3, 4, 6
10         h_a[i] = i*atoi(argv[1]);                                 // 5
11
12     cudaMemcpy(d_a,h_a,size,cudaMemcpyHostToDevice);              // 7
13     exemplo<<<2,2>>>(d_a, N);                                     // 8
14     cudaMemcpy(h_a, d_a, size, cudaMemcpyDeviceToHost);           // 9
15     cudaFree(d_a);                                                // 10
16     free(h_a);                                                    // 10
17 }
```

No contexto do *grid* há apenas a sincronização explícita representada por N_{sync}^{grid}. A sincronização é feita pela primitiva *___syncThreads*, onde todas as *threads* de um bloco precisam esperar para que o bloco possa prosseguir com sua execução. Não há sincronização entre blocos dentro do *grid*. Para sincronizar os blocos é necessário fazer a sincronização no lado do *host*. O Código 5 apresenta um exemplo de *kernel* que será usado para exemplificação da aplicação do modelo de teste. Dessa forma, ele não possui sincronização explícita no kernel, possuindo sincronização apenas no *host*. Este é o *kernel* invocado pelo *host* no Código 4 para ser executado na GPU. Outros *kernels* testados possuem sincronização.

Código 5 – Código de exemplo do *Kernel*.

```
1  __global__ void exemplo (int *d_a, int N) {
2      int idx = threadIdx.x + blockDim.x * blockIdx.x;          // 1
3
4      if(d_a[idx] < 100)                                        // 2
5          d_a[idx] += 10;                                       // 3
6                                                                // 4
7  }
```

Em relação à comunicação entre *host* e *grid*, as sincronizações são causadas por primitivas no *host*. Essa comunicação ocorre em dois momentos. O primeiro é a chamada do *kernel*, onde o *host* requisita a execução e envia as informações para a GPU. Essa comunicação se assemelha ao modelo de passagem de mensagens, onde há um *send* e um *receive* que ocorre no processo de inicialização do *kernel* e em sua finalização. A execução do *kernel* é assíncrona em relação ao *host*, mas síncrona em relação à chamada de outros *kernels* e de primitivas CUDA, como as cópias de memória com *cudaMemcpy*. No modelo de teste estamos generalizando como *send* a chamada do *kernel* e o término de sua execução. Como *receive* são considerados o *cudaDeviceSynchronize*, *cudaMemcpy* com

cópia do *device* para o *host* e primitivas similares que causem sincronização no *host*. No Código 4 estão sendo considerados como pontos de sincronização as linhas 7, 12, 13, 14 e 15, sendo incluídas no conjunto de nós N^{host}_{sync}.

A Tabela 1 apresenta os conjuntos obtidos pela aplicação do modelo de teste no programa da Figura 19 com suas sincronizações. O PCFG para CUDA é representado pelo conjunto *Prog* composto de dois elementos, *h* referente ao *host*, o processo principal do programa, e o g^1 referente ao único *grid* do programa, composto por um conjunto de *n threads*. O *host h* possui os nós do conjunto N^h, enquanto o *grid* possui os nós do conjunto N^{g1}. Um subconjunto de N^h é composto por nós de sincronização N^{host}_{sync}, enquanto N^{grid}_{sync} é um subconjunto de N^{g^1}. As arestas do *host* e *grid* são apresentadas nos conjuntos E^h e E^{g^1}, respectivamente. A primeira aresta de sincronização E_{sync} ocorreu no momento da chamada do *kernel* no nó 11^h com a inicialização do *grid* no nó 1^{g^0}. A segunda aresta de sincronização ocorre na finalização do *grid* 21^{g^0} ligando a cópia do conteúdo para o *host* no nó 12^h. Se este código tivesse utilizado a memória unificada, a sincronização seria feita com o *cudaDeviceSynchronize*, tendo o mesmo efeito semântico.

Tabela 1 – Conjuntos

Conjuntos	**Elementos**
Prog	$\{h, g^1\}$
g^1	$\{gt^{1,0}, ..., gt^{1,n}\}$
N^h	$\{ 1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0, 8^0, 9^0, 10^0 \}$
N^g	$\{ N^{g1}, ..., N^{gn} \}$
N^{g^1}	$\{ 1^1, 2^1, 3^1, 4^1 \}$
N	$N^h \cup N^g$
N^{host}_{sync}	$\{ 2^0, 7^0, 8^0, 9^0, 10^0 \}$
N^{grid}_{sync}	$\{ \emptyset \}$
N_{sync}	$N^{host}_{sync} \cup N^{grid}_{sync}$
E^h	$\{ (1^0, 2^0), (2^0, 3^0), (3^0, 4^0), (4^0, 5^0), (4^0, 7^0), (5^0, 6^0), (6^0, 4^0), (7^0, 8^0), (8^0, 9^0), (9^0, 10^0) \}$
E^g	$\{ E^{g1}, ..., E^{gn} \}$
E^{g^1}	$\{ (1^1, 2^1), (2^1, 3^1), (2^1, 4^1), (3^1, 4^1) \}$
E_{sync}	$\{ (8^0, 1^1), (4^1, 9^0) \}$
E	$E^g \cup E^h \cup E_{sync}$

5.2.3 Fluxo de dados

O fluxo de dados considera a ocorrência das variáveis no programa. O modelo Graf-Def-Uso de Rapps e Weyuker (1982), adiciona informações ao Grafo de Fluxo de Controle relacionadas à definição e uso da variável. A definição acontece quando um valor é atribuído à variável x. As Tabelas 2 e 3 apresentam as definições de variáveis no *host* e *grid* em relação ao programa apresentado na Figura 19. No caso das definições de variáveis do *grid*, foram adicionadas as variáveis definidas implicitamente na criação do *grid* que são usadas para identificar as *threads*, sendo elas a *threadIdx*, *blockIdx*, *blockDim* e *gridDim*. O uso da variável ocorre quando o valor da mesma é utilizado, podendo ser de três tipos, c-uso, p-uso e s-uso.

- uso computacional (c-uso): se refere à utilização da variável x para um cálculo ou para apresentar seu valor;
- uso predicativo (p-uso): ocorre na utilização da variável x para decidir o fluxo de execução, como no uso em estruturas de repetição e decisão;
- uso de sincronização (s-uso): ocorre quando a variável x é utilizada em uma comunicação na chamada de *kernel* pelo *host* e na finalização do *kernel* e retorno ao *host*.

Tabela 2 – Definições de variáveis do Host

nó	**variável**
1^0	{ h_a, argc, argv, N, size }
2^0	{ i }
3^0	{ i }
5^0	{ h_a }
6^0	{ i }
7^0	{ d_a }
9^0	{ h_a }

Tabela 3 – Definições de variáveis do Grid 1

nó	**variável**
1^1	{ d_a, N, idx, threadIdx.x, threadIdx.y, threadIdx.z, blockIdx.x, blockIdx.y, blockIdx.z, blockDim.x, blockDim.y, blockDim.z, gridDim.x, gridDim.y, gridDim.z }
3^1	{ d_a }

Um programa CUDA possui diversos tipos de variáveis. Há variáveis definidas no *host*, no *grid* e variáveis sendo compartilhadas entre os dois. No contexto do *host*, o modelo considera as suas variáveis locais e no contexto do *grid* há variáveis locais, compartilhadas em nível de bloco e em nível global. As variáveis locais são visíveis apenas pela própria *thread*, não sendo utilizadas para comunicação. As variáveis compartilhadas em nível de bloco são visíveis por todas as *threads* do bloco, podendo ser utilizadas para comunicação e ter o seu acesso sincronizado durante a execução do *kernel*. As variáveis compartilhadas em nível global são criadas no *host* e são passadas como argumento durante a chamada do *kernel*. Elas são visíveis por todas as *threads* em execução no *grid* e podem ser utilizadas para comunicação, no entanto, a sincronização em relação ao acesso pode ser feita apenas no *host*, ou seja, é necessário terminar a execução do *kernel*. As variáveis globais perduram durante todo o tempo de vida do programa, podendo ser desalocadas pelo *host*.

As variáveis são representadas pelo conjunto V no modelo de teste para CUDA. O conjunto V é composto pelos subconjuntos V^{host} e V^{grid}. V^{host} inclui todas as variáveis usadas na *thread* do *host*. V^{grid} é composto pelas variáveis locais, compartilhadas em nível de bloco e compartilhadas em nível global. O conjunto das variáveis locais V^{grid}_{local} foi definido para capturar a definição e uso sequencial das variáveis. O conjunto das variáveis compartilhadas em nível de bloco V^{grid}_{block} e em nível global V^{grid}_{global} foram definidos para capturar as definições e usos nos dois níveis de compartilhamento de variável, pois há erros causados especificamente pela falta de sincronização em nível de bloco, e erros causados pela ausência de sincronização em nível de *grid*.

A Tabela 4 apresenta os conjuntos relacionados as informações de fluxo de dados do programa. O conjunto V^{host} apresenta as variáveis do *host*. $\mathrm{V}^{grid}_{local}$, $\mathrm{V}^{grid}_{blockshared}$ e $\mathrm{V}^{grid}_{globalshared}$ apresentam as variáveis locais, em nível de bloco e em nível global do *kernel*, respectivamente. Há ainda dois conjuntos adicionais, o V_{comm} e o $\mathrm{V}_{host-device}$. O conjunto V_{comm} apresenta especificamente as variáveis globais que foram passadas como argumento na chamada do *kernel*, enquanto $\mathrm{V}_{host-device}$ relaciona a variável criada no *host* e passada na chamada de *kernel*, com o nome da variável nos parâmetros do *kernel*. Dessa forma, mesmo havendo uma mudança de nome, é possível relacioná-los.

Tabela 4 – Conjuntos dos tipos de variáveis

Conjuntos	**Elementos**
V_{local}^{grid}	{ V_{local}^{grid1} }
V_{local}^{grid1}	{ idx, threadIdx.x, threadIdx.y, threadIdx.z, blockDim.x, blockDim.y, blockDim.z, blockIdx.x, blockIdx.y, blockIdx.z, gridDim.x, gridDim.y, gridDim.z, i }
$V_{globalshared}^{grid}$	{ $V_{globalshared}^{grid1}$ }
$V_{globalshared}^{grid1}$	{ d_a, N}
$V_{blockshared}^{grid}$	{ $V_{blockshared}^{grid1}$ }
$V_{blockshared}^{grid1}$	{ $\emptyset$ }
V_{comm}	{ d_a, N }
$V_{host-grid}$	{ ((d_a, d_a), g^1), ((N, N), g^1) }
V^{grid}	$V_{local}^{grid} \cup V_{globalshared}^{grid} \cup V_{blockshared}^{grid}$
V^{host}	{ argc, argv, N, size, h_a, d_a, i }
V	$V^{host} \cup V^{grid}$

Com base nos conjuntos de variáveis apresentados, foram definidos os seguintes tipos de uso de variáveis:

- **uso computacional (c-uso):** a variável x é utilizada para realizar alguma computação.
- **uso predicativo (p-uso):** a variável x é usada para determinar o fluxo de execução do programa.
- **uso computacional no *host* (c-uso^{host}):** a variável x é utilizada para realizar alguma computação, com a variável $x \in V^{host}$.
- **uso predicativo no *host* (p-uso^{host}):** a variável x é usada para determinar o fluxo de execução do programa, com a variável $x \in V^{host}$.
- **uso computacional no *grid* (c-uso^{grid}):** a variável x é utilizada para realizar alguma computação, com a variável $x \in V^{grid}$.
- **uso predicativo no *grid* (p-uso^{grid}):** a variável x é usada para determinar o fluxo de execução do programa, com a variável $x \in V^{grid}$.
- **uso computacional no bloco (bloco-c-uso^{grid}):** a variável compartilhada x em nível de bloco do *grid* é utilizada para realizar alguma computação, com a variável $x \in V_{blockshared}^{grid}$.

- **uso predicativo no bloco (bloco-p-uso^{grid}):** a variável compartilhada x em nível de bloco é usada para determinar o fluxo de execução do programa, com a variável $x \in V^{grid}_{blockshared}$.
- **uso computacional global (global-c-uso^{grid}):** a variável compartilhada x em nível global do *grid* é utilizada para realizar alguma computação, com a variável $x \in V^{grid}_{globalshared}$.
- **uso predicativo global (global-p-uso^{grid}:** a variável compartilhada x em nível global é usada para determinar o fluxo de execução do programa, com a variável $x \in V^{grid}_{globalshared}$.
- **uso de sincronização (s-uso):** a variável x é utilizada em situações de sincronização, onde ela é criada no *host* e enviada ao *device* por meio da execução do *kernel*, ou na finalização do *kernel* e sincronizando em um nó de sincronização do *host*, onde $x \in V_{comm}$.
- **uso computacional de variável de sincronização (s-c-uso):** a variável x é utilizada em situações de sincronização, e utilizada em uma computação no destino, onde $x \in V_{comm}$ e $x \in V$.
- **uso predicativo de variável de sincronização (s-p-uso):** a variável x é utilizada em situações de sincronização, e utilizada em uma mudança de fluxo de execução no destino, onde $x \in V_{comm}$ e $x \in V$.

As associações foram definidas com base na definição e uso das variáveis. Foram criadas associações relacionadas ao código sequencial do *host*, associações relacionadas ao *kernel* do *grid* e, por fim, associações relacionadas à comunicação entre *host* e *grid*, sendo elas:

- **Associação c-uso**: é uma associação entre a definição de uma variável no nó n_i e seu subsequente c-uso no nó n_j, sendo definida pela tripla (n_i, n_j, x).
  - n_i: nó inicial onde foi definida a variável
  - n_j: nó onde ocorreu o uso computacional da variável
  - x: variável que foi definida e usada
- **Associação p-uso**: é uma associação entre a definição de uma variável no nó n_i e tem um p-uso na aresta (n_j, n_k), sendo definida pela tripla (n_i, (n_j, n_k), x).
  - n_i: nó inicial onde foi definida a variável
  - n_j: nó onde ocorreu o uso predicativo da variável
  - n_k: nó resultante do uso predicativo da variável

  - x: variável que foi definida e usada

- **Associação c-uso**^{host}: É uma associação entre a definição de uma variável no nó n_i^h e seu subsequente c-uso no nó n_j^h, sendo definida pela tripla (n_i^h, n_j^h, x).
  - n_i^h: nó inicial onde foi definida a variável no *host*
  - n_j^h: nó onde ocorreu o uso computacional da variável *host*
  - x: variável que foi definida e usada
- **Associação p-uso**^{host}: é uma associação entre a definição de uma variável no nó n_i^h e tem um p-uso na aresta (n_j^h, n_k^h), sendo definida pela tripla (n_i^h, (n_j^h, n_k^h), x).
  - n_i^h: nó inicial onde foi definida a variável no *host*
  - n_j^h: nó onde ocorreu o uso predicativo da variável no *host*
  - n_k^h: nó resultante do uso predicativo da variável no *host*
  - x: variável que foi definida e usada
- **Associação c-uso**^{grid}: É uma associação entre a definição de uma variável no nó n_i^g e seu subsequente c-uso no nó n_j^g, sendo definida pela tripla (n_i^g, n_j^g, x).
  - n_i^g: nó inicial onde foi definida a variável no *grid*
  - n_j^g: nó onde ocorreu o uso computacional da variável *grid*
  - x: variável que foi definida e usada
- **Associação p-uso**^{grid}: é uma associação entre a definição de uma variável no nó n_i^g e tem um p-uso na aresta (n_j^g, n_k^g), sendo definida pela tripla (n_i^g, (n_j^g, n_k^g), x).
  - n_i^g: nó inicial onde foi definida a variável no *grid*
  - n_j^g: nó onde ocorreu o uso predicativo da variável no *grid*
  - n_k^g: nó resultante do uso predicativo da variável no *grid*
  - x: variável que foi definida e usada
- **Associação bloco-c-uso**^{grid}: É uma associação entre a definição de uma variável compartilhada em nível de bloco no nó n_i^g e seu subsequente c-uso no nó n_j^g, sendo definida pela tripla (n_i^g, n_j^g, x).
- **Associação bloco-p-uso**^{grid}: é uma associação entre a definição de uma variável compartilhada em nível de bloco no nó n_i^g e tem um p-uso na aresta (n_j^g, n_k^g), sendo definida pela tripla (n_i^g, (n_j^g, $^g n_k$), x).
- **Associação global-c-uso**^{grid}: É uma associação entre a definição de uma variável compartilhada em nível global no nó n_i^g e seu subsequente c-uso no nó n_j^g, sendo definida pela tripla (n_i^g, n_j^g, x).

- **Associação global-p-uso**^{grid}: é uma associação entre a definição de uma variável compartilhada em nível global no nó n_i^g e tem um p-uso na aresta (n_j^g, n_k^g), sendo definida pela tripla (n_i^g, (n_j^g, n_k^g), x).

- **Associação s-uso**: É uma associação entre a definição de uma variável x no nó n_i^{pa} e seu subsequente s-uso na aresta de sincronização (n_j^{pa}, n_k^{pb}), sendo definida pela tripla (n_i^{pa}, (n_j^{pa}, n_k^{pb}), x).

  - n_i^{pa}: nó inicial onde foi definida a variável
  - n_j^{pa}: nó que iniciou a comunicação da variável
  - n_k^{pb}: nó que recebeu a variável
  - x: variável que foi definida e enviada

- **Associação s-c-uso**: É uma associação entre a definição de uma variável x_a no nó n_i^{pa}, sua comunicação na aresta de sincronização (n_j^{pa}, n_k^{pb}), e seu subsequente c-uso no nó n_l^{pb}, sendo definida pela tupla (n_i^{pa}, (n_j^{pa}, n_k^{pb}), n_l^{pb}, x_a, x_b).

  - n_i^{pa}: nó inicial onde foi definida a variável
  - n_j^{pa}: nó que iniciou a comunicação da variável
  - n_k^{pb}: nó que recebeu a variável
  - n_l^{pb}: nó onde houve o uso computacional da variável
  - x_a: variável que foi definida
  - x_b: variável que foi usada

- **Associação s-p-uso**: é uma associação entre a definição de uma variável x_a no nó n_i^{pa}, sua comunicação na aresta de sincronização (n_j^{pa}, n_k^{pb}), e tem um p-uso na aresta (n_l^{pb}, n_m^{pb}), sendo definida pela tupla (n_i^{pa}, (n_j^{pa}, n_k^{pb}), (n_l^{pb}, n_m^{pb}), x_a, x_b).

  - n_i^{pa}: nó inicial onde foi definida a variável
  - n_j^{pa}: nó que iniciou a comunicação da variável
  - n_k^{pb}: nó que recebeu a variável
  - n_l^{pb}: nó onde ocorreu o uso predicativo da variável
  - n_m^{pb}: nó resultante do uso predicativo da variável
  - x_a: variável que foi definida
  - x_b: variável que foi usada

5.3 Critérios de Teste Estrutural para Programas CUDA

Com base no modelo de fluxo de controle e fluxo de dados, foram definidos os critérios para programas concorrentes CUDA. Foram levados em consideração os critérios existentes para programas sequenciais, programas concorrentes com memória compartilhada, e programas concorrentes com passagem de mensagem.

Critérios de fluxo de controle e sincronização:

- **todos-nós**: Requer que todos os nós N sejam exercitados pelos casos de teste. Critério já usado no teste de programas sequenciais, paralelo com passagem de mensagens e paralelo com memória compartilhada.
- **todos-nós-host**: Requer que todos os nós N^{host} sejam exercitados pelos casos de teste.
- **todos-nós-grid**: Requer que todos os nós N^{grid} sejam exercitados pelos casos de teste.
- **todos-nós-sync**: Requer que todos os nós N_{sync} sejam exercitados pelos casos de teste.
- **todos-nós-sync-host**: Requer que todos os nós N^{host}_{sync} sejam exercitados pelos casos de teste.
- **todos-nós-sync-grid**: Requer que todos os nós N^{grid}_{sync} sejam exercitados pelos casos de teste.
- **todas-arestas**: Requer que todas as arestas E sejam exercitadas pelos casos de teste.
- **todas-arestas-host**: Requer que todas as arestas E^{host} sejam exercitadas pelos casos de teste.
- **todas-arestas-grid**: Requer que todas as arestas E^{grid} sejam exercitadas pelos casos de teste.
- **todas-arestas-sync**: Requer que todas as arestas de sincronização E_{sync} sejam exercitadas pelos casos de teste.

Critérios de fluxo de dados e comunicação

- **todas-definições**: Requer que cada definição de variável $x \in V$ seja exercitada por ao menos um c-uso ou p-uso pelos casos de teste.

- **todas-definições-host**: Requer que cada definição de variável $x \in V^{host}$ seja exercitada por ao menos um c-uso^{host} ou p-uso^{host} pelos casos de teste.
- **todas-definições-grid**: Requer que cada definição de variável $x \in V^{grid}$ seja exercitada por ao menos um c-uso^{grid}, p-uso^{grid}, bloco-c-uso^{grid}, bloco-p-uso^{grid}, global-c-uso^{grid} ou global-p-uso^{grid} pelos casos de teste.
- **todos-c-uso**: Requer que todas as associações c-uso sejam exercitadas pelos casos de teste.
- **todos-c-uso-host**: Requer que todas as associações c-uso^{host} sejam exercitadas pelos casos de teste.
- **todos-c-uso-grid**: Requer que todas as associações c-uso^{grid} sejam exercitadas pelos casos de teste.
- **todos-p-uso**: Requer que todas as associações p-uso sejam exercitadas pelos casos de teste.
- **todos-p-uso-host**: Requer que todas as associações p-uso^{host} sejam exercitadas pelos casos de teste.
- **todos-p-uso-grid**: Requer que todas as associações p-uso^{grid} sejam exercitadas pelos casos de teste.
- **todos-bloco-c-uso-grid**: Requer que todas as associações bloco-c-uso^{grid} sejam exercitadas pelos casos de teste.
- **todos-bloco-p-uso-grid**: Requer que todas as associações bloco-p-uso^{grid} sejam exercitadas pelos casos de teste.
- **todos-global-c-uso-grid**: Requer que todas as associações global-c-uso^{grid} sejam exercitadas pelos casos de teste.
- **todos-global-p-uso-grid**: Requer que todas as associações global-p-uso^{grid} sejam exercitadas pelos casos de teste.
- **todos-usos**: Requer que todas as associações c-uso e p-uso sejam exercitadas pelos casos de teste.
- **todos-usos-host**: Requer que todas as associações c-uso^{host} e p-uso^{host} sejam exercitadas pelos casos de teste.
- **todos-usos-grid**: Requer que todas as associações c-uso^{grid}, p-uso^{grid} sejam exercitadas pelos casos de teste.

- **todos-s-uso**: Requer que todas as associações s-uso sejam exercitadas pelos casos de teste.
- **todos-s-c-uso**: Requer que todas as associações s-c-uso sejam exercitadas pelos casos de teste.
- **todos-s-p-uso**: Requer que todas as associações s-p-uso sejam exercitadas pelos casos de teste.

A Tabela 5 possui os elementos requeridos do programa de exemplo (Código Código 5) para os critérios propostos que foram implementados na ferramenta. Ela aplica algumas otimizações na geração de elementos requeridos, dessa forma, elementos que possuem definição e uso no mesmo nó não são gerados e adicionados à lista. Por consequência, a Tabela 5 não apresenta esses elementos.

Tabela 5 – Elementos requeridos

Critério	**Elementos requeridos**
todos-nós-host	{ 1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0, 8^0, 9^0, 10^0 }
todos-nós-grid	{ 1^1, 2^1, 3^1, 4^1 }
todos-nós	{ 1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0, 8^0, 9^0, 10^0, 1^1, 2^1, 3^1, 4^1 }
todos-nós-sync-host	{ 2^0, 7^0, 8^0, 9^0, 10^0 }
todos-nós-sync-grid	{ $\emptyset$ }
todos-nós-sync	{ 2^0, 7^0, 8^0, 9^0, 10^0 }
todas-arestas-host	{ (1^0, 2^0), (2^0, 3^0), (3^0, 4^0), (4^0, 5^0), (4^0, 7^0), (5^0, 6^0), (6^0, 4^0), (7^0, 8^0), (8^0, 9^0), (9^0, 10^0) }
todas-arestas-grid	{ (1^1, 2^1), (2^1, 3^1), (2^1, 4^1), (3^1, 4^1) }
todas-arestas-sync	{ (8^0, 1^1), (4^1, 9^0) }
todas-arestas	{ (1^0, 2^0), (2^0, 3^0), (3^0, 4^0), (4^0, 5^0), (4^0, 7^0), (5^0, 6^0), (6^0, 4^0), (7^0, 8^0), (8^0, 9^0), (9^0, 10^0), (1^1, 2^1), (2^1, 3^1), (2^1, 4^1), (3^1, 4^1), (8^0, 1^1), (4^1, 9^0) }
todos-c-usos-host	{ (1^0, 2^0, size), (3^0, 5^0, i), (3^0, 6^0, i), (7^0, 10^0, d_a), (9^0, 10^0, h_a) }
todos-c-usos-grid	{ (1^1, 2^1, idx), (1^1, 3^1, d_a), (1^1, 3^1, idx) }
todos-c-usos	{ (1^0, 2^0, size), (3^0, 5^0, i), (3^0, 6^0, i), (7^0, 10^0, d_a), (9^0, 10^0, h_a), (1^1, 2^1, idx), (1^1, 3^1, d_a), (1^1, 3^1, idx) }
todos-p-usos-host	{ (1^0, (4^0, 5^0), N), (1^0, (4^0, 7^0), N), (3^0, (4^0, 5^0), i), (3^0, (4^0, 7^0), i) }

todos-p-usos-grid	{ (1^1, (2^1, 3^1), d_a), (1^1, (2^1, 4^1), d_a) }
todos-p-usos	{ (1^0, (4^0, 5^0), N), (1^0, (4^0, 7^0), N), (3^0, (4^0, 5^0), i), (3^0, (4^0, 7^0), i), (1^1, (2^1, 3^1), d_a), (1^1, (2^1, 4^1), d_a) }
todos-usos-host	{ (1^0, 2^0, size), (3^0, 5^0, i), (3^0, 6^0, i), (7^0, 10^0, d_a), (9^0, 10^0, h_a), (1^0, (4^0, 5^0), N), (1^0, (4^0, 7^0), N), (3^0, (4^0, 5^0), i), (3^0, (4^0, 7^0), i) }
todos-usos-grid	{ (1^1, 2^1, idx), (1^1, 3^1, d_a), (1^1, 3^1, idx), (1^1, (2^1, 3^1), d_a), (1^1, (2^1, 4^1), d_a) }
todos-usos	{ (1^0, 2^0, size), (3^0, 5^0, i), (3^0, 6^0, i), (7^0, 10^0, d_a), (9^0, 10^0, h_a), (1^0, (4^0, 5^0), N), (1^0, (4^0, 7^0), N), (3^0, (4^0, 5^0), i), (3^0, (4^0, 7^0), i), (1^1, 2^1, idx), (1^1, 3^1, d_a), (1^1, 3^1, idx), (1^1, (2^1, 3^1), d_a), (1^1, (2^1, 4^1), d_a) }
todos-global-c-usos-grid	{ (1^1, 3^1, d_a) }
todos-bloco-c-usos-grid	∅
todos-global-p-usos-grid	{ (1^1, (2^1, 3^1), d_a), (1^1, (2^1, 4^1), d_a) }
todos-bloco-p-usos-grid	∅
todos-s-usos	{ (1^0, (8^0, 1^1), N), (7^0, (8^0, 1^1), d_a), (1^1, (4^1, 9^0), d_a), (3^1, (4^1, 9^0), d_a) }
todos-s-c-usos	{ (7^0, (8^0, 1^1), 3^1, d_a, d_a) }
todos-s-p-usos	{ (7^0, (8^0, 1^1), (2^1, 3^1), d_a, d_a), (7^0, (8^0, 1^1), (2^1, 4^1), d_a, d_a) }

5.4 Um exemplo de Aplicação do Teste Estrutural para Programação CUDA

Aplicamos o modelo de teste desenvolvido no programa de exemplo apresentado no Código 4 e no Código 5 para exemplificar o funcionamento do modelo, quais informações foram extraídas do código fonte e quais informações foram geradas pelo modelo de teste proposto. Dessa forma, foram identificadas as definições de variáveis no código fonte do *host* e do *grid*, apresentadas respectivamente nas Tabelas 2 e 3. O conjunto dos tipos de variáveis foi apresentado na Tabela 4. Os elementos requeridos pelos critérios de teste propostos e gerados pela ferramenta com base nas informações extraídas estão contidos na Tabela 5.

O código do programa exemplo foi instrumentado com base no PCFG apresentado na Figura 19, para exemplificar a aplicação dos critérios de teste. A instrumentação permite gerar o rastro da execução do *host* e do *grid* em relação aos nós do grafo. Esse rastro permite identificar o caminho executado pela única *thread* do *host*, assim como pelas *threads*

geradas no *grid*, permitindo assim aferir quais elementos requeridos foram executados em cada um. O número de elementos requeridos está intrinsecamente relacionado com a quantidade de *threads* geradas nas chamadas de *kernel* do programa.

Foram executados três casos de teste para avaliar a cobertura alcançada em cada critério. Para os três casos o número de *threads* executado pelo *grid* foi definido estaticamente para quatro, não variando em relação à entrada utilizada. Para o primeiro caso de teste foi utilizado o valor de entrada 0 atribuído à variável $argv[1]$. Para o segundo caso foi atribuído o valor 50 e para o terceiro caso foi inserido o valor 100. Os valores de cada caso de teste foram definidos para que as *threads* do *grid* executem diferentes caminhos em cada execução.

A Tabela 6 apresenta a cobertura alcançada para cada critério em relação à execução de cada caso de teste. Ainda inclui a cobertura total alcançada pela sobreposição de cobertura de todos os casos de teste. A equação 5.1 apresenta o cálculo utilizado para a taxa de cobertura. Ela considera o número de elementos requeridos, o número total de *threads* que deveriam executar esses elementos, e o número de *threads* que executaram os elementos. Na Tabela, a primeira coluna indica os critérios de teste utilizados. A segunda coluna apresenta o número de elementos requeridos, considerando a quantidade de elementos para o *host* e para o *grid*. A terceira coluna considera a quantidade de elementos requeridos totais, onde é contabilizado os elementos do *host* e para os elementos do *grid* são considerados a quantidade de *threads*. Nesse exemplo o *grid* possui 4 *threads*, com isso, cada elemento requerido do *grid* foi multiplicado por 4, pois na avaliação da ValiEval será verificado se cada uma das *threads* cobriram o elemento requerido. Nos casos onde o critério de teste não apresentou 100% na cobertura total, após a sobreposição dos três casos de teste, foi devido a esses elementos requeridos serem não executáveis, não havendo um dado de teste que possibilite a sua execução, devido as estruturas condicionais do código.

$$cobertura = \frac{thread^{host} * elementos^{host}_{executados} + threads^{grid} * elementos^{grid}_{executados}}{thread^{host} * elementos^{host}_{requeridos} + threads^{grid} * elementos^{grid}_{requeridos}} \quad (5.1)$$

Tabela 6 – Cobertura dos elementos requeridos alcançada pela execução dos casos de teste.

Critério	Elementos requeridos	Elementos totais	Cobertura caso 0	Cobertura caso 1	Cobertura caso 2	Elementos exec total	Elementos não executados	Cobertura total
todos-nos	14	26	100,00%	92,31%	88,46%	26	0	100,00%
todos-nos-grid	4	16	100,00%	87,50%	81,25%	16	0	100,00%
todos-nos-host	10	10	100,00%	100,00%	100,00%	10	0	100,00%
todos-nos-sinc	7	13	100,00%	100,00%	100,00%	13	0	100,00%
todos-nos-sinc-grid	2	8	100,00%	100,00%	100,00%	8	0	100,00%
todos-nos-sinc-host	5	5	100,00%	100,00%	100,00%	5	0	100,00%
todas-arestas	6	18	77,78%	77,78%	77,78%	17	1	94,44%
todas-arestas-grid	2	8	50,00%	50,00%	50,00%	7	1	87,50%
todas-arestas-host	2	2	100,00%	100,00%	100,00%	2	0	100,00%
todas-arestas-sinc	2	8	100,00%	100,00%	100,00%	8	0	100,00%
todos-c-usos	8	17	100,00%	76,47%	64,71%	17	0	100,00%
todos-c-usos-grid	3	12	100,00%	66,67%	50,00%	12	0	100,00%
todos-c-usos-host	5	5	100,00%	100,00%	100,00%	5	0	100,00%
todos-p-usos	6	12	66,67%	66,67%	66,67%	11	1	91,67%
todos-p-usos-grid	2	8	50,00%	50,00%	50,00%	7	1	87,50%
todos-p-usos-host	4	4	100,00%	100,00%	100,00%	4	0	100,00%
todos-usos	14	29	86,21%	72,41%	65,52%	28	1	96,55%
todos-usos-grid	5	20	80,00%	60,00%	50,00%	19	1	95,00%
todos-usos-host	9	9	100,00%	100,00%	100,00%	9	0	100,00%
todos-s-c-usos	1	4	100,00%	50,00%	25,00%	4	0	100,00%
todos-s-p-usos	2	8	50,00%	50,00%	50,00%	7	1	87,50%
todos-s-usos	4	16	75,00%	75,00%	75,00%	15	1	93,75%
todos-bloco-c-usos-grid	0							
todos-bloco-p-usos-grid	0							
todos-global-c-usos-grid	1	4	100,00%	50,00%	25,00%	4	0	100,00%
todos-global-p-usos-grid	2	8	50,00%	50,00%	50,00%	7	1	87,50%

5.5 Considerações Finais

Neste capítulo apresentamos o modelo de teste proposto para o teste de cobertura de programas CUDA. Demonstramos quais informações relacionadas ao fluxo de controle, fluxo de dados e comunicação são registrados e utilizados pelo modelo de teste com base em um exemplo de código CUDA. Apresentamos o modelo de grafo de fluxo de controle utilizado pelo modelo. Propusemos novos conjuntos elementos relacionados ao modelo de programação, incluindo as derivadas das sincronizações utilizadas na linguagem. Descrevemos os tipos de variáveis previstas no modelo e, com base nessas informações, foram propostos usos de variáveis e associações entre definição e uso das variáveis. Por fim, propusemos novos critérios de teste de fluxo de controle, fluxo de dados e comunicação específicos para o modelo de programação CUDA e demonstramos o seu uso no programa de exemplo. Utilizamos casos de teste buscando atingir todos os elementos requeridos executáveis e apresentamos a cobertura atingida para cada caso de teste, assim como a cobertura da sobreposição dos casos de teste, demonstrando a equação do cálculo de cobertura do modelo.

6 FERRAMENTA DE TESTE ESTRUTURAL PARA CUDA

6.1 Considerações iniciais

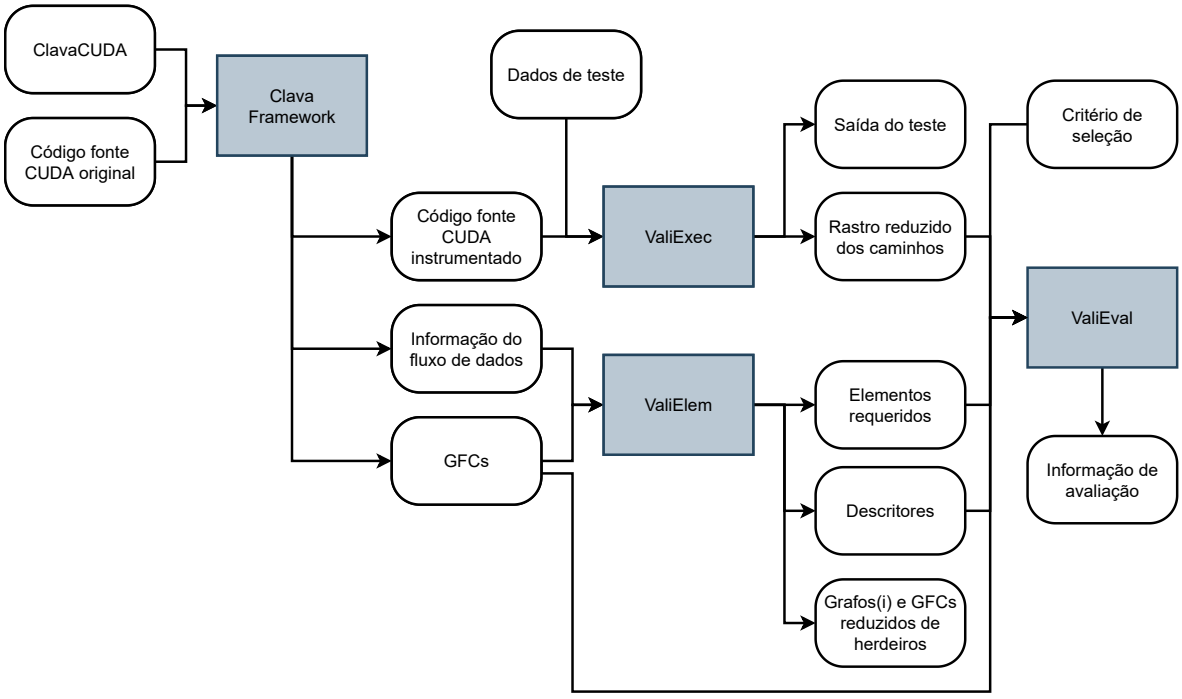
Apresentamos neste capítulo a implementação da ferramenta ValiCUDA, para teste de programas paralelos que utilizam o modelo de programação de CUDA. Esta ferramenta implementa o modelo e critérios de teste apresentados no Capítulo 5. Ele utiliza como base a ferramenta ValiMPI apresentada no Capítulo 3, na Seção 3.5, mas realizando as extensões necessárias para adicionar suporte aos programas que utilizam CUDA.

Na Seção 6.2 é apresentada a arquitetura geral da ferramenta ValiCUDA, apresentando cada um dos módulos que a compõe. Na Seção 6.3 é apresentado e contextualizado o *framework* Clava, uma ferramenta que nos permitiu instrumentar os códigos fontes e extrair as informações relacionadas ao fluxo de controle e fluxo de dados. Na Seção 6.4 é relatado sobre o *script* que possui as estratégias de instrumentação, demonstrando exemplos dos artefatos gerados. Na Seção 6.5 são relatadas as modificações feitas no módulo ValiElem, e demonstrados exemplos da saída que o módulo apresenta. A Seção 6.6 demonstra as modificações feitas no módulo ValiExec para permitir a execução dos programas. A Seção 6.7 demonstra como é feita a avaliação da cobertura dos casos de teste, exemplificando os relatórios gerados. Por fim, a Seção 6.8 possui as considerações finais.

6.2 ValiCUDA

Para o desenvolvimento da ValiCUDA foi utilizada a ferramenta ValiMPI como base, realizando modificações para atender o modelo de programação CUDA. O módulo de instrumentação ValiInst foi substituído pelo *framework* Clava (BISPO; CARDOSO, 2020) que permite realizar a análise, coleta de informações e instrumentação do programa utilizando uma linguagem de programação baseada na linguagem de programação *JavaScript*, facilitando o processo de implementação da instrumentação. Os módulos ValiElem, ValiExec e ValiEval foram modificados para suportarem o modelo de programação CUDA. Na ValiElem foi adicionado o suporte às informações adicionais geradas pelo *framework* Clava e adicionada a geração dos elementos requeridos para os novos critérios propostos. Foi adicionada à ValiExec a opção de executar programas desenvolvidos com CUDA. Na ValiEval foi adicionado suporte aos critérios de teste para CUDA, modificando a metodologia de teste e o relatório gerado pela ferramenta. A arquitetura da ValiCUDA já considerando estes módulos é apresentada na Figura 20.

Figura 20 – Arquitetura de módulos da ferramenta de teste ValiCUDA, utilizando Clava e ValiMPI.



Fonte: Elaborada pelo autor.

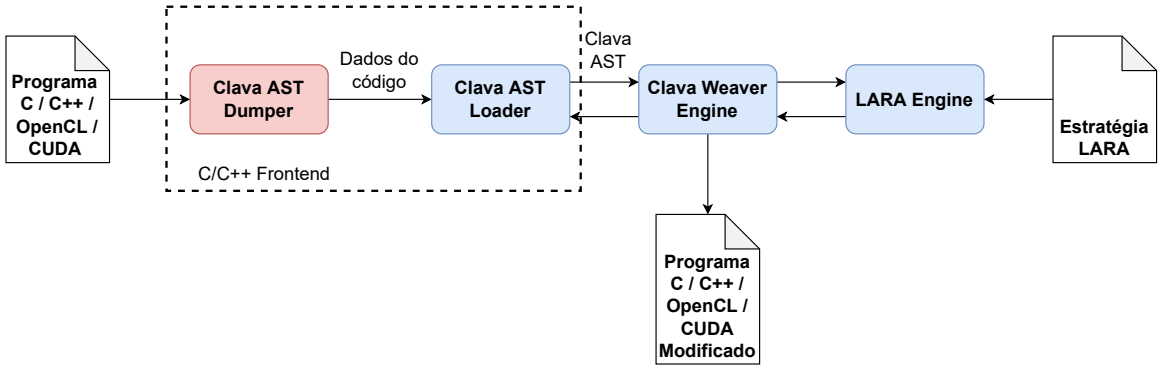
6.3 *Framework* Clava

Clava é um compilador de código fonte baseado no projeto Clang (LATTNER, 2008). Ele permite fazer pesquisa, análise e transformação de códigos escritos em C/C++ e OpenCL, permitindo paralelizar, parametrizar e instrumentar o código por meio da definição de estratégias. Essas estratégias são descritas utilizando a linguagem LARA, uma linguagem baseada no *JavaScript* que permite definir quais informações extrair do código, quais modificações serão feitas, e o que será inserido no código.

A Figura 21 apresenta a estrutura do *framework* Clava, possuindo quatro módulos principais. O "frontend C/C++" é composto pelos módulos "Clava AST Dumper" e "Clava AST Loader", que transformam o código fonte em uma Árvore Sintática Abstrata (*Abstract Syntax Tree* - AST). Ele é baseado no AST do Clang, mas com algumas mudanças para permitir a realização de transformações no AST e permitir a geração do código fonte a partir do AST. O "Clava Weaver Engine" provê informações sobre o código C/C++ baseado no AST recebido, e mantém internamente uma representação atualizada do código fonte, de acordo com as modificações instruídas pelo módulo LARA. A "Lara Engine" é composta por um analisador e um interpretador de código, que interpreta o *script* LARA e aplica as estratégias definidas pelo usuário. O *framework* recebe então dois arquivos de

entrada, o código fonte original do programa, e o código Lara com as estratégias definidas pelo usuário. Como saída ele apresenta o código fonte modificado do programa de acordo com as estratégias estabelecidas.

Figura 21 – Diagrama da estrutura do *framework* Clava e o fluxo de compilação



Fonte: adaptado de Bispo e Cardoso (2020)

O *framework* Clava foi utilizado para realizar a extração das informações e instrumentação do código CUDA. Para isso, foi necessário adicionar suporte ao CUDA. Dessa forma, os autores do framework (BISPO; CARDOSO, 2020) realizaram a atualização do Clang, para que o mesmo pudesse oferecer o suporte à geração de AST para códigos CUDA. Com isso foi adicionado ao Clava o suporte para realizar a extração de informações específicas do modelo de programação CUDA e a instrumentação do código com base nas diretivas do modelo. Isso viabilizou a criação do *script*, chamado ClavaCUDA, para definir as estratégias de extração e instrumentação do código.

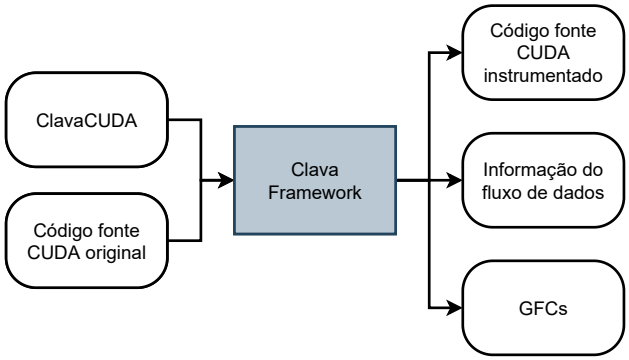
6.4 ClavaCUDA

ClavaCUDA é o *script* desenvolvido utilizando a linguagem LARA que possui as estratégias para a instrumentação e extração de informações de códigos fonte de programas CUDA. Conforme ilustrado na Figura 22, o *framework* Clava recebe como entrada o código fonte do programa CUDA e o *script ClavaCUDA*, que informa a análise e transformações que serão feitas no código. Ao final do processo, o *framework* entregará o código fonte instrumentado, as informações do fluxo de dados e os GFCs dos processos.

O código fonte é instrumentado de acordo com os nós do GFC do processo. Cada nó do grafo é referente a uma ou mais linhas do código fonte, sendo que para algumas primitivas do CUDA é criado um nó exclusivo.

Para cada processo do programa é criado um GFC diferente. No modelo de teste, consideramos como processo o *host*, possuindo uma única *thread*, e os *grids* criados por cada chamada de *kernel*. Para um programa CUDA com duas chamadas de *kernel*, serão

Figura 22 – *Script* ClavaCUDA em conjunto com o *framework* Clava.



Fonte: Elaborada pelo autor.

criados três GFCs, um para o processo do *host*, um para o primeiro *grid* e outro para o segundo *grid*. O número de *threads* criadas para cada *grid* não influenciam na criação dos GFCs. Como as *threads* de um mesmo *grid* possuem o mesmo código, eles possuem na prática o mesmo grafo de fluxo de controle.

A instrumentação realizada no código é distinta entre o código do *host* e do *kernel*. Para a instrumentação do *host* é inserido no início do programa o Código 6. Para cada ponto instrumentado, apresentado no Código 7, o procedimento é chamado, registrando no arquivo de rastro do *host* que o nó em específico foi executado. Ao final da execução, teremos o arquivo com o rastro do caminho executado, apresentado no Código 8. Independentemente do método de instrumentação, ele mantém a semântica e ordem original de execução do programa.

Código 6 – Função para registro de rastro do *host*.

```
1 __host__ void registerTraceHost(FILE *fp, unsigned char node)
2 {
3    fprintf(fp, "%d-0\t", node);
4 }
```

Na instrumentação do código do *kernel* não é possível para a *thread* do *grid* escrever diretamente no arquivo de rastro, como ocorre no *host* (esta é uma restrição do modelo de programação CUDA). Para a escrita é necessário passar a informação ao *host*, que faz a criação e escrita do arquivo. Para a instrumentação do *kernel* ocorrer é necessário criar uma estrutura de dado onde cada *thread* possa registrar a execução do nó. A estrutura pode ser visualizada como uma matriz, onde cada coluna é referente a uma *thread*, e cada linha é referente a um nó executado pela *thread*. Cada *thread* consegue registrar o caminho de execução específico. O número de linhas deve ter o tamanho máximo que um caminho executado possa ter. Isso pode ser feito instrumentando os nós para contar o número de vezes que são executados, executar o programa instrumentado com dados de entrada que

Código 7 – Código do *host* instrumentado.

```
int main(int argc, char *argv[])
{
   FILE *fpHost = fopen("trace.Host.p0", "w");

   registerTraceHost(fpHost, 1);
   int N = 4;
   size_t size = N * sizeof(int);
   int *h_a;
   h_a = (int *) malloc(size);
   int *d_a;
   /* node 2 */
   cudaMalloc(&d_a, size);
   int i = 0;
   for(/* node 3 */ i = 0; /* node 4 */ i < N; /* node 6 */ i++) {
      registerTraceHost(fpHost, 5);
      h_a[i] = i * atoi(argv[1]);
   }
   registerTraceHost(fpHost, 7);
   cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
   /* node 8 */

   unsigned char *instTrace;
   int numberOfThreads0 = 2 * 2;
   cudaMallocManaged(&instTrace, numberOfThreads0 * 3 * sizeof(unsigned char));
   exemplo<<<2, 2>>>(d_a, N, instTrace, numberOfThreads0);
   /* node 9 */
   cudaMemcpy(h_a, d_a, size, cudaMemcpyDeviceToHost);
   registerTraceHost(fpHost, 10);
   cudaFree(d_a);
   free(h_a);

   FILE *fp;
   fp = fopen("trace.Grid.p1", "w");
   for (int i = 0; i < numberOfThreads0; i++)
   {
      for (int j = 0; j < 3; j++)
      {
         if (instTrace[i + j * numberOfThreads0] != 0)
            fprintf(fp, "%d-1\t", instTrace[i + j * numberOfThreads0]);
      }
      fprintf(fp, "\n");
   }
   fclose(fp);

   fp = fopen("commsize", "w");
   fprintf(fp, "2");
   fclose(fp);
   fclose(fpHost);
}
```

Código 8 – Rastro de execução do *host*.

```
1-0     5-0     5-0     5-0     5-0     7-0     10-0
```

Código 9 – Função para registro de rastro das *threads* do *grid*.

```
1  __device__ void registerTrace(unsigned char *traceArray, int width, int id, int
       cont, unsigned char node)
2  {
3     traceArray[cont * width + id] = node;
4  }
```

Código 10 – Código do *kernel* instrumentado.

```
1  __global__ void exemplo(int *d_a, int N, unsigned char * traceArray, int
       numberOfThreads)
2  {
3     int map = (blockIdx.x  + gridDim.x  * blockIdx.y) * (blockDim.x * blockDim.y)
          + (threadIdx.x + blockDim.x * threadIdx.y);
4     int cont = 0;
5     registerTrace(traceArray, numberOfThreads, map, cont++, 1);
6     int idx = threadIdx.x + blockDim.x * blockIdx.x;
7     /* node 2 */
8     if(d_a[idx] < 100) { registerTrace(traceArray, numberOfThreads, map, cont++,
          3); d_a[idx] += 10; }
9     registerTrace(traceArray, numberOfThreads, map, cont++, 4);
10 }
```

Código 11 – Rastro de execução das *threads* do *grid*.

```
1 1-1     3-1     4-1
2 1-1     4-1
3 1-1     4-1
4 1-1     4-1
```

implique no maior número de execuções dos laços de repetição, verificar quais *threads* tiveram os maiores números de execução para cada nó e somá-los.

Para registrar o caminho executado pelas *threads* do *grid* é inserido no programa a função do Código 9. Esta função permite à *thread* inserir na estrutura de dado o nó do grafo que ela executou. Qual nó é executado é informado na chamada da função, apresentada no Código 10. Ao final da execução do programa, a estrutura de dados criada para o *grid* é acessada pelo *host* (Código 7) para a criação e escrita do arquivo de rastro, como mostrado no Código 11.

Devido à necessidade de ter a estrutura de dados completa na memória da GPU, duas estratégias foram empregadas buscando mitigar o uso de memória. O primeiro está relacionado ao tipo de dado da estrutura. Inicialmente foi utilizado o tipo *int* para registrar o número do nó, entretanto ele utiliza 4 *bytes*, dessa forma modificamos para *unsigned char*, utilizando somente 1 *byte* e permitindo utilizar até o nó 255. Como os *kernels* não costumam serem extensos, essa limitação não deve ser um problema.

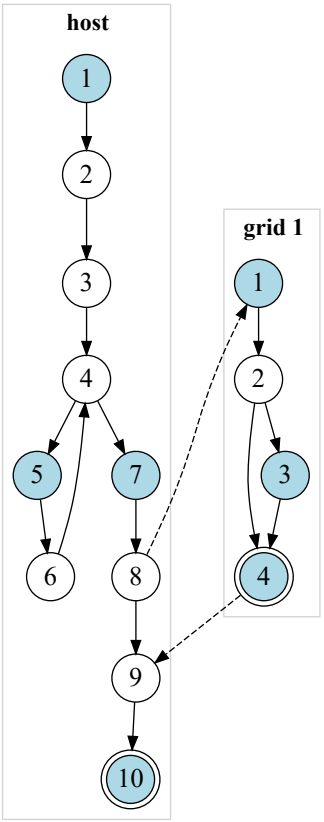
A segunda estratégia diz respeito a quais nós precisam ser instrumentados. Seguindo o conceito de arestas primitivas apresentada por Chusho (1987), o mesmo conceito pode ser aplicado na instrumentação do código. Como a instrumentação utiliza o conceito de nós, temos o que pode-se chamar de nós primitivos ou nós essenciais. Os nós primitivos

são os nós resultantes da bifurcação no fluxo de controle, ou seja, o nó derivado de um nó com cardinalidade maior que um. A Figura 23 apresenta os nós primitivos do exemplo no Código 4 e Código 5, e que foram instrumentados conforme apresentado no Código 7 e Código 10. Também foi considerado como nó primitivo para instrumentação o nó inicial e o nó final de cada GFC. Com isso, o rastro gerado é uma versão resumida do rastro completo, que pode ser reconstruído com base no GFC do processo.

Essas estratégias foram utilizadas buscando reduzir o uso de memória pela estrutura de dados do rastro de execução no *device*. Como um *kernel* pode ser executado por milhares de *threads*, reduzir o uso de memória permite realizar o teste com maiores quantidades de *threads*. Entretanto, a segunda estratégia implica no aumento do processamento na avaliação da cobertura alcançada. Como o rastro gerado não é completo, ele precisa ser reconstruído para ser utilizado pela ValiEval para testar a cobertura dos autômatos.

Ao final do processo, são gerados os arquivos DOT de cada processo. Cada arquivo contém os nós e arestas do GFC do processo, as definições e usos de variáveis, as variáveis compartilhadas em nível de bloco e global, e a comunicação feita entre *host* e *grid* na chamada do *kernel* e na finalização do *grid*. O Código 12 apresenta o arquivo DOT do *grid* do programa CUDA de exemplo.

Figura 23 – Nós instrumentados de acordo com o PCFG.



Código 12 – Arquivo C.Grid0.dot com informações do CFG e fluxo de dados.

```
digraph cfg{
    node [shape = doublecircle] 4;
    node [shape = circle];
    /* definition of threadIdx.x at 1 */
    /* definition of threadIdx.y at 1 */
    /* definition of threadIdx.z at 1 */
    /* definition of blockIdx.x at 1 */
    /* definition of blockIdx.y at 1 */
    /* definition of blockIdx.z at 1 */
    /* definition of blockDim.x at 1 */
    /* definition of blockDim.y at 1 */
    /* definition of blockDim.z at 1 */
    /* definition of gridDim.x at 1 */
    /* definition of gridDim.y at 1 */
    /* definition of gridDim.z at 1 */
    /* definition of d_a at 3 */
    /* definition of d_a at 1 */
    /* definition of N at 1 */
    /* definition of idx at 1 */
    /* cusage of threadIdx.x at 1 */
    /* cusage of blockDim.x at 1 */
    /* cusage of blockIdx.x at 1 */
    /* cusage of idx at 2 */
    /* cusage of d_a at 3 */
    /* cusage of idx at 3 */
    /* pusage of d_a at 2 */
    /* pusage of d_a at 2 */
    /* susage of d_a at 4 */
    /* m_recv.0 of d_a at 1 */
    /* m_recv.0 of N at 1 */
    /* m_send.0 of d_a at 4 */
    /* global of d_a at 0 */
    /* global of N at 0 */
    1 -> 2;
    2 -> 3;
    3 -> 4;
    2 -> 4;
}
```

6.5 ValiElem

No módulo ValiElem foram adicionadas a geração dos elementos requeridos para os novos critérios do modelo de teste para CUDA. O funcionamento original do módulo foi mantido, sendo necessário na execução passar o argumento `-cuda` para gerar os elementos requeridos com base no arquivo DOT criado pelo *framework* Clava. Para a geração dos elementos relacionados ao fluxo de controle foram adicionados suporte aos nós de sincronização do programa, que são informados no arquivo DOT. Para o fluxo de dados foi necessário adicionar suporte às variáveis compartilhadas em nível de bloco e em nível global do *grid*. Para as sincronizações relacionadas entre o *host* e os *grids* criados, foi reutilizada a implementação original de primitivas *send* e *receive*, diferenciando o método de sincronização nas descrições contidas no arquivo DOT para relacionar os envios de dados.

A arquitetura apresentada na Figura 24 se manteve igual ao original. O módulo recebe os GFCs e informações de fluxo de dados relacionados com cada processo *host*

Código 13 – Arquivo todos-s-usos.req contendo os elementos requeridos do critério s-usos.

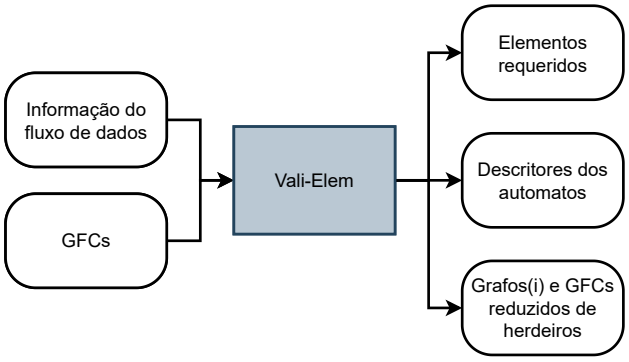
```
ELEMENTOS REQUERIDOS PELO CRITERIO TODOS-S-USOS

1) <1-0, (8-0, 1-1), N>
2) <7-0, (8-0, 1-1), d_a>
3) <1-1, (4-1, 9-0), d_a>
4) <3-1, (4-1, 9-0), d_a>
```

Código 14 – Arquivo todos-s-usos.aut contendo os autômatos do critério s-usos.

```
1) 2 3 3 1 1:Np0/1-0 2:1-0 0 2 2:Np0n5/8-0 3:8-0 0 3 3:N 0 2 2 1 1:N/1-1 2:1-1 0 2 2:N 0
2) 2 3 3 1 1:Np0/7-0 2:7-0 0 2 2:Np0n2/8-0 3:8-0 0 3 3:N 0 2 2 1 1:N/1-1 2:1-1 0 2 2:N 0
3) 2 3 3 1 1:Np1/1-1 2:1-1 0 2 2:Np1n12/4-1 3:4-1 0 3 3:N 0 2 2 1 1:N/9-0 2:9-0 0 2 2:N 0
4) 2 3 3 1 1:Np1/3-1 2:3-1 0 2 2:Np1n12/4-1 3:4-1 0 3 3:N 0 2 2 1 1:N/9-0 2:9-0 0 2 2:N 0
```

e *grid* por meio dos arquivos DOT. Com base neles são criados os grafos(i) e GFCs reduzidos de herdeiros para a criação dos elementos requeridos de fluxo de controle e fluxo de dados. Em conjunto com a geração dos elementos requeridos, são criados os descritores dos autômatos para cada elemento. O padrão utilizado na criação dos autômatos e na criação dos descritores se manteve o mesmo para manter a compatibilidade com o módulo ValiEval.

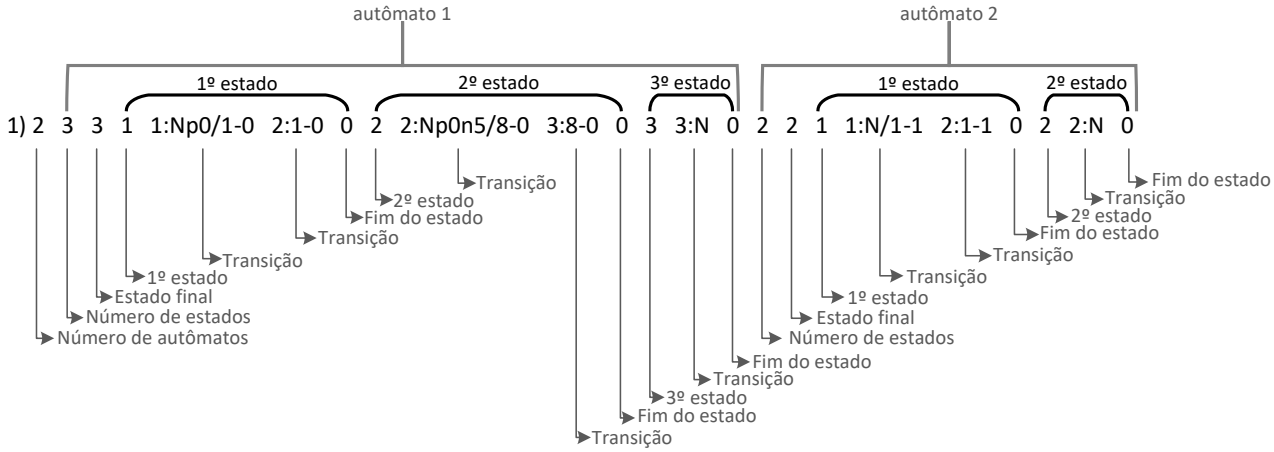
Figura 24 – Módulo ValiElem.



Fonte: Elaborada pelo autor.

Um exemplo dos elementos requeridos e o descritor gerado para o critério todos-s-usos podem ser verificados nos Código 13 e Código 14, respectivamente. A Figura 25 demonstra como é feita a leitura e a interpretação do autômato do exemplo. Para este critério, como há uma comunicação entre o *host* e um *grid*, o descritor possui dois autômatos, denotado pelo primeiro número. O segundo e terceiro número representam a quantidade de estados do autômato e qual é o estado final, respectivamente. Após são descritos cada estado do autômato, começando pelo 1º estado, posteriormente suas transições, até chegar ao fim desse estado, para iniciar o próximo.

Figura 25 – Legenda dos elementos que compõem o descritor de um autômato.



Fonte: Elaborado pelo autor com base em Sarmanho *et al.* (2008).

A criação dos descritores dos elementos requeridos para os critérios de teste de programas CUDA seguiu dois padrões. Para os critérios específicos apenas para *host* ou *grid*, o descritor possui somente um único autômato para verificar a execução dos nós, arestas, definições e uso de variáveis, conforme a especificação do critério. Para os critérios de teste onde há a comunicação entre o *host* e o *grid*, o descritor possui dois autômatos, um para verificar a execução dos estados do processo que fez o envio, e o segundo autômato para verificar a execução dos estados do processo que recebeu a comunicação. O descritor apresentado na Figura 25 é um exemplo do segundo caso. Ele busca verificar se houve a sincronização entre o processo que enviou e o que recebeu a informação.

Os autômatos implementados para os novos critérios de teste para programas CUDA podem ser averiguados nas Figuras 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39 e 40. Na representação, N é o conjunto de nós do GFCP, sendo h referente ao *host* e g referente ao *grid* e suas *threads*. Os nós x são os que possuem definição de variável, c possuem uso computacional, i e j possuem uso predicativo, s e r possuem uso comunicacional de variável, e n e m são nós do processo. Ngn, Nhn e Np_an são nós sem redefinição da variável.

Figura 26 – Automato do critério todos-nos-host.

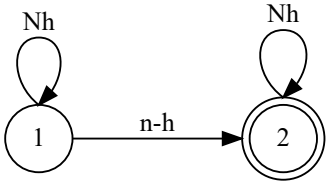


Figura 27 – Automato do critério todos-nos-grid.

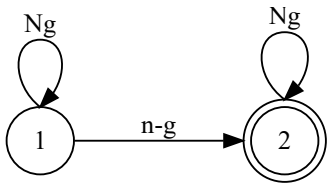


Figura 28 – Automato do critério todos-nos-sinc.

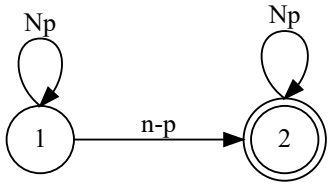


Figura 29 – Automato do critério todos-nos-host-sinc.

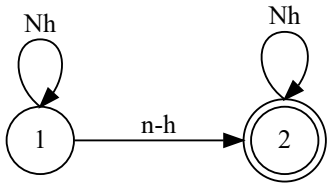


Figura 30 – Automato do critério todos-nos-grid-sinc.

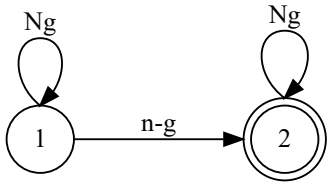


Figura 31 – Automato do critério todas-arestas-host.

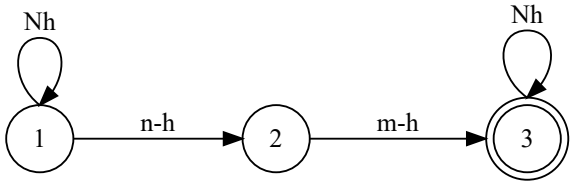


Figura 32 – Automato do critério todas-arestas-grid.

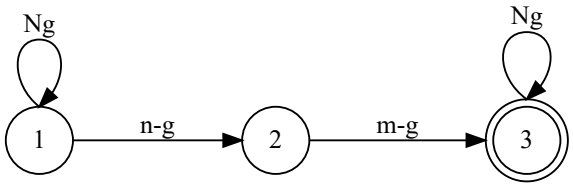


Figura 33 – Automato do critério todas-arestas-sinc.

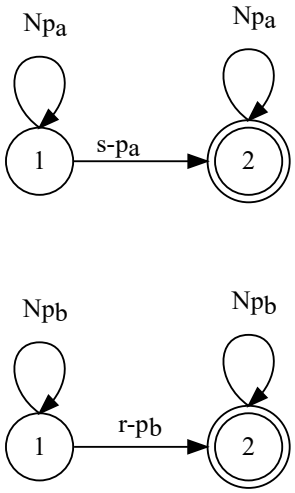


Figura 34 – Automato do critério todos-c-usos-host.

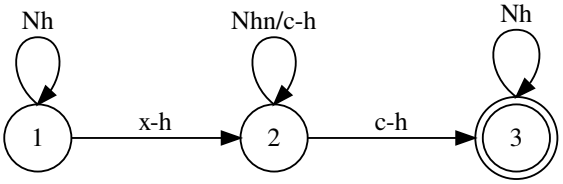


Figura 35 – Automato do critério todos-c-usos-grid.

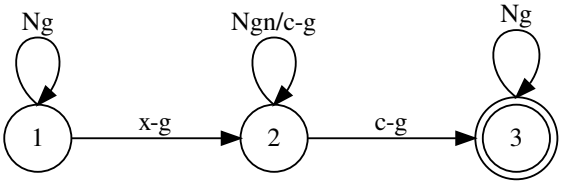


Figura 36 – Automato do critério todos-p-usos-host.

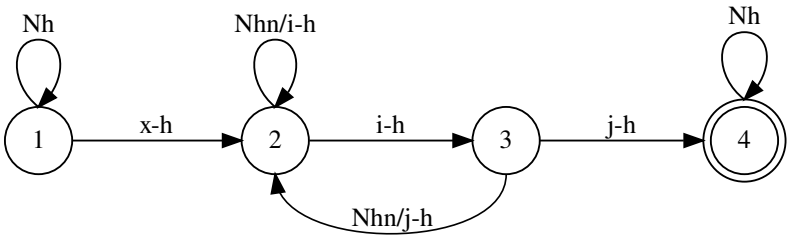


Figura 37 – Automato do critério todos-p-usos-grid.

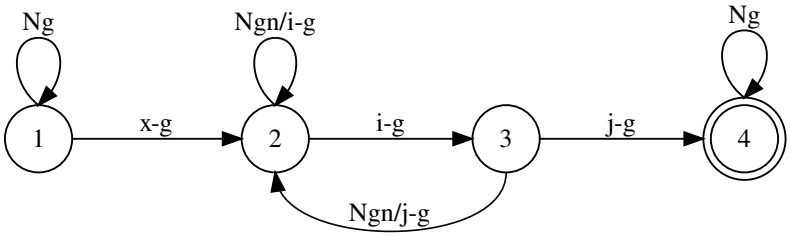


Figura 38 – Automato do critério todos-s-usos.

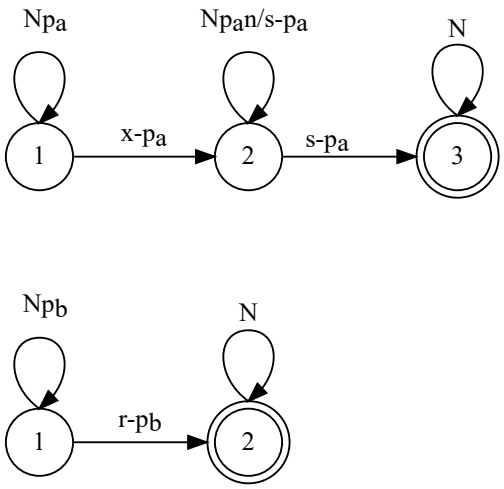


Figura 39 – Automato do critério todos-s-c-usos.

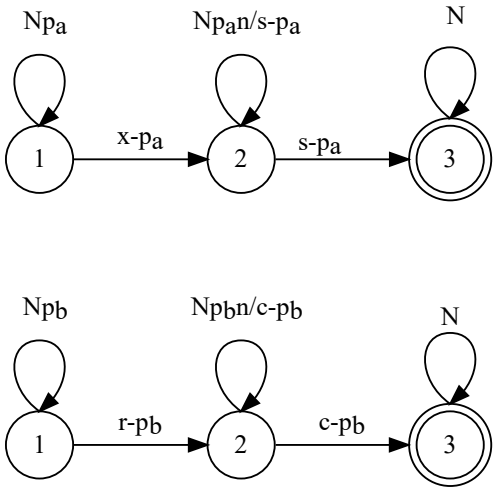
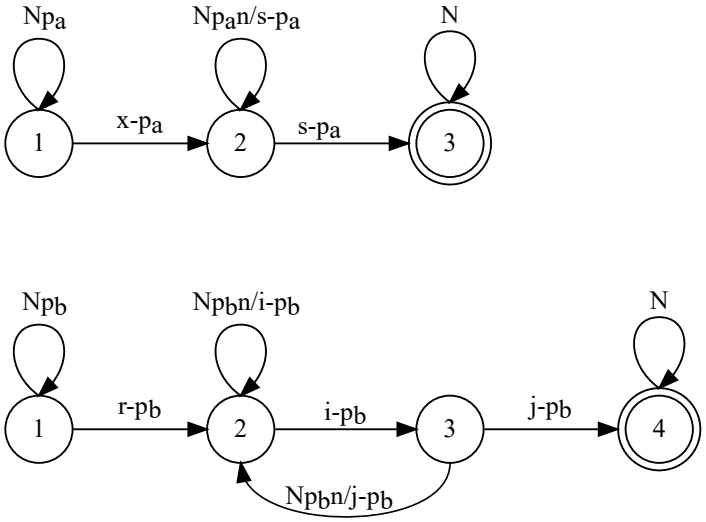


Figura 40 – Automato do critério todos-s-p-usos.

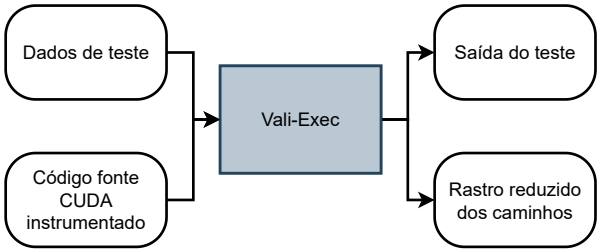


6.6 ValiExec

A ValiExec tem a finalidade de executar programas compilados escritos em C que utilizam o padrão MPI. Logo, ao chamá-lo ele irá executar o programa utilizando o comando `mpirun` ou `mpiexec`, dependendo da versão MPI informada. Portanto, foi necessário modificá-lo para executar programas em CUDA, mas mantendo as funcionalidades originais. Com esse objetivo foram adicionados dois modos de execução da ValiExec, o primeiro

passando o argumento `-mpi`, que mantém o processo de execução original. O segundo modo é usado ao passar o argumento `-cuda`, executando o programa CUDA informado. Na Figura 41 temos as informações de entrada e saída da ValiExec. Nos dados de teste são informados o modo de execução, a numeração que será dada ao caso de teste, o número de processos do programa e os argumentos que serão passados para o programa em execução. Ao fim da execução são gerados os arquivos pelo programa instrumentado, como a saída do teste e o rastro reduzido dos caminhos executados por cada *thread* e processo.

Figura 41 – Módulo ValiExec.



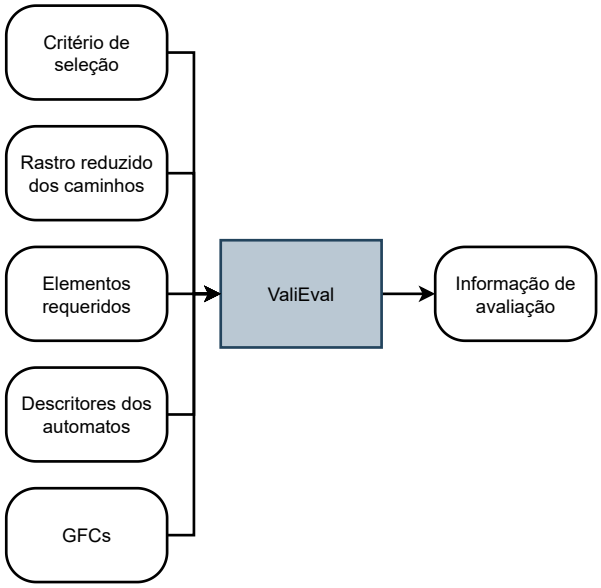
Fonte: Elaborada pelo autor.

6.7 ValiEval

Foram necessárias diversas adaptações ao módulo ValiEval para se adequar aos artefatos de teste gerados pelo *framework* Clava, ValiElem e pela execução do programa instrumentado. Primeiramente, foi adicionado o argumento `-cuda` para executar o módulo para a avaliação dos casos de teste para CUDA. Caso não seja passado este argumento na chamada do módulo, ele executará as funções implementadas para a avaliação de cobertura de programas em C/MPI, como implementado originalmente. No modo de execução para CUDA, é necessário informar o número de processos e a quantidade de *threads* que cada *grid* possui, permitindo o teste de programas com mais de um *grid* com número de *threads* diferentes. Ele ainda recebe como entrada o critério de teste que será avaliado, os rastros reduzidos de caminhos gerados na execução do programa instrumentado, os elementos requeridos e descritores gerados pelo ValiElem e, por fim, os GFCs criados pelo *framework* Clava, conforme demonstrado na Figura 42. Ao final da execução é apresentado o relatório de avaliação da cobertura.

Originalmente, ele recebe e executa um rastro completo do caminho executado pelos processos. Devido ao rastro do programa CUDA ser reduzido, com base nos nós primitivos, foi necessário implementar a reconstrução do caminho completo antes de testá-lo com base no autômato. Para a reconstrução, ele recebe os GFCs gerados pelo *framework* Clava e, por meio dos grafos, são construídos máquinas de estado finito para cada grafo. Por meio da execução são reconstruídos os caminhos completos de cada *thread* antes de testar a

Figura 42 – Módulo ValiEval.



Fonte: Elaborada pelo autor.

cobertura do autômato do elemento requerido.

Não foram necessárias modificações para os descritores dos autômatos. Os descritores possuem o mesmo padrão utilizado pelos critérios anteriores, por isso, não foi necessário realizar adaptações na leitura e construção dos autômatos no módulo ValiEval, assim como não foi necessário modificar o teste do caminho completo em relação ao autômato.

Entretanto, foi modificada a forma como são marcadas as coberturas dos autômatos. Originalmente, a ValiEval marca se o caminho executado pelo processo cobriu o elemento requerido. Contudo, para a ValiCUDA o programa possui poucos processos, mas uma grande quantidade de *threads*, sendo necessário avaliar a cobertura por *thread*, e não apenas por processo. Com isso, a ferramenta continua marcando a cobertura do elemento requerido pelo processo quando o caminho de uma *thread* o cobre, mas é registrada também a cobertura atingida isoladamente para cada *thread* do processo, para permitir a avaliação de cobertura nesse nível de granularidade. Ao final do teste, é calculada a cobertura alcançada com base na equação 5.1 já apresentada anteriormente para cada caso de teste e, ainda, para a sobreposição da execução de todos os casos de teste.

Devido ao alto número de *threads* de programas CUDA em relação à programas sequenciais e paralelos com MPI, foi necessário modificar o relatório de avaliação de cobertura. Foram implementados dois tipos de relatórios diferentes, um resumido e outro estendido. A execução padrão apresenta o relatório resumido, onde são listados todos os elementos requeridos cobertos e não cobertos. No caso de elementos requeridos relacionados ao *grid*, são apresentadas as taxas de cobertura alcançadas para cada elemento requerido

com base no número de *threads* que o executaram em relação ao número de *threads* totais do *grid* em específico. No final do relatório é apresentado o numerador e denominador da equação 5.1 e a cobertura total alcançada pelo critério, conforme mostrado no Código 15 para o critério todas-arestas na avaliação de cobertura do programa de exemplo para um único caso de teste.

O segundo tipo de relatório implementado foi o estendido. Ele é executado ao passar o argumento `-verbose` na execução do ValiEval. Neste caso, são listados todos os elementos requeridos para cada *thread* em específico. Com isso, é possível observar quais *threads* não conseguiram executar um determinado elemento requerido. O Código 16 apresenta a saída da cobertura alcançada no critério todas-arestas com a execução de três casos de teste diferentes.

Código 15 – Exemplo de saída resumida da ValiEval para o critério todas-arestas de um único caso de teste.

```
1  -- RESULTADO RESUMIDO --
2
3  -- ELEMENTOS REQUERIDOS COBERTOS --
4  1) 4-0 5-0, coberto
5  2) 4-0 7-0, coberto
6  3) 2-1 3-1, cobertura 25.00%
7  4) 2-1 4-1, cobertura 75.00%
8  5) 8-0 1-1, cobertura 100.00%
9  6) 4-1 9-0, cobertura 100.00%
10
11 -- ELEMENTOS REQUERIDOS NAO COBERTOS --
12
13 Threads: 14 / 18
14 Coverage for the todas-arestas criterion: 77.78%
```

Código 16 – Exemplo de saída estendida da ValiEval para o critério todas-arestas com sobreposição de todos os casos de teste.

```
1  -- RESULTADO ESTENDIDO --
2
3  -- ELEMENTOS REQUERIDOS COBERTOS --
4  Elemento                    Thread
5  1) 4-0 5-0                       -
6  2) 4-0 7-0                       -
7  3) 2-1 3-1                       0
8  3) 2-1 3-1                       1
9  3) 2-1 3-1                       2
10 3) 2-1 3-1                       3
11 4) 2-1 4-1                       1
12 4) 2-1 4-1                       2
13 4) 2-1 4-1                       3
14 5) 8-0 1-1                       0
15 5) 8-0 1-1                       1
16 5) 8-0 1-1                       2
17 5) 8-0 1-1                       3
18 6) 4-1 9-0                       0
19 6) 4-1 9-0                       1
20 6) 4-1 9-0                       2
21 6) 4-1 9-0                       3
22
23 -- ELEMENTOS REQUERIDOS NAO COBERTOS --
24 4) 2-1 4-1                       0
25
26 Threads: 17 / 18
27 Coverage for the todas-arestas criterion: 94.44%
```

6.8 Considerações finais

Apresentamos a implementação do modelo e critérios de teste propostos na ferramenta ValiMPI. Demonstramos as modificações que foram necessárias em cada módulo da ferramenta para se adaptar ao modelo e critérios de teste propostos. Para a instrumentação e extração das informações do fluxo de dados o módulo original da ValiMPI foi substituído por um novo *framework*, em conjunto com a nossa implementação do *script* que define as estratégias de análise do código.

Demonstramos as modificações nos três módulos utilizados da ValiMPI. Modificamos a ValiExec para permitir a execução de programas CUDA. Adicionamos na ValiElem a capacidade de lidar com as novas informações de fluxo de controle e fluxo de dados provenientes do modelo de programação, e a implementação dos novos critérios de teste para a geração dos elementos requeridos e dos descritores. Por fim relatamos as mudanças na ValiEval, adicionando os novos critérios, dando suporte ao teste de programas com processos que possuem múltiplas *threads* utilizando apenas um GFC por processo, e a geração de relatório com modalidade resumida e estendida. Na versão resumida é apresentada a cobertura alcançada em percentual por elemento requerido, enquanto a estendida apresentada a cobertura detalhada dos elementos requeridos por *thread*.

Essa implementação permitiu realizar as atividades de teste em programas CUDA que possuam as características e recursos previstos no modelo de teste.

7 VALIDAÇÃO DO TESTE ESTRUTURAL PARA PROGRAMAS CUDA

7.1 Considerações iniciais

Neste capítulo nós mostramos o processo de validação do teste estrutural para programas concorrentes que utilizam o modelo de programação CUDA. A validação do teste estrutural para programas CUDA proposto neste trabalho foi realizado por meio de estudos experimentais com benchmarks seguindo a metodologia de condução de experimentos apresentada por Wohlin et al. (2012). Na Seção 7.2 mostramos o conjunto de *benchmarks* utilizados para a validação e as suas características. O planejamento do experimento, com suas métricas, hipóteses, variáveis e passo a passo para a realização é apresentado na Seção 7.3. Os resultados obtidos nos experimentos são mostrados na Seção 7.4. O teste de hipóteses em relação às métricas selecionadas é apresentado na Seção 7.5, e as conclusões acerca dos resultados são discutidas na Seção 7.6.

7.2 Benchmarks utilizados

Os *benchmarks* foram selecionados com o principal propósito de avaliar o modelo de teste, os critérios de teste propostos e a ferramenta de teste ValiCUDA. Foram selecionados ao todo 20 *benchmarks*. Do conjunto, 9 *benchmarks* são provenientes de terceiros e 11 desenvolvemos neste projeto para avaliar o modelo e critérios teste, testar a ferramenta, ou validar o suporte a algum recurso específico do modelo CUDA, como o suporte à execução de múltiplos *grids*. Os *benchmarks* apresentam diferentes características a utilização de diversas primitivas do modelo de programação CUDA, como: a alocação de memória para as variáveis do *host* e do *device*; as transferências de conteúdo da memória entre *host* e *device*; a chamada de *kernel* para a realização da computação na GPU; a definição do número de *threads* do *grid* com base no número de blocos do *grid* e no número de *threads* do bloco; a utilização de memória local no *kernel*; a utilização de memória compartilhada por bloco por meio da primitiva *shared*; a utilização de memória compartilhada em nível global ao *grid* por meio das variáveis definidas no *host*; e as sincronizações entre *threads* em nível de bloco e *grid*.

Um *benchmark* da seleção foi criado com a finalidade de demonstrar o uso do modelo e critérios de teste propostos nos Capítulos 5 e 6, e alguns *benchmarks* foram desenvolvidos para testar o funcionamento e suporte das ferramentas. Eles foram desenvolvidos para testar a instrumentação, a geração de elementos requeridos e a avaliação de cobertura de alguns recursos pontuais, como operações atômicas, uso de múltiplos *grids* e *grids* com diferentes configurações de *threads*. Esses códigos foram incluídos na avaliação de custo dos critérios.

Dessa forma, para validar o modelo de teste e critérios de teste propostos, a seguinte relação de *benchmarks* foi selecionada:

- **2_grids**: código para exemplificar e testar o modelo de teste e ferramenta em programas com duas chamadas de *kernel* (desenvolvido pelo autor).
- **3_grids**: código para exemplificar e testar o modelo de teste e ferramenta em programas com três chamadas de *kernel* que possuam números diferentes de *threads* (desenvolvido pelo autor).
- **chamada**: código de exemplo de chamada de *kernel* com possibilidade do *kernel* não ser chamado devido ao dado de entrada (desenvolvido pelo autor).
- **conta_ocorrencia**: contabiliza quantas vezes um determinado valor se repete em uma sequência de números (desenvolvido pelo autor).
- **exemplo**: código implementado para exemplificação da aplicação do modelo e critérios de teste propostos, apresentados nos Capítulos 5 e 6 (desenvolvido pelo autor).
- **hello**: implementação do *Hello World* para CUDA (CHENG; GROSSMAN; MCKERCHER, 2014).
- **linha_coluna**: executa a soma das linhas e das colunas de uma matriz quadrada de tamanho $N \times N$ (desenvolvido pelo autor).
- **maior_menor**: verifica qual o maior e o menor valor encontrado em um vetor de tamanho N. Ele possui dois *kernels*, um para cada verificação, sendo realizadas duas chamadas para cada um, dessa forma criando ao todo quatro *grids* no processo. É uma adaptação da implementação apresentada por Harris (2007).
- **matriz_diag**: monta a matriz diagonal a partir de uma matriz quadrada de tamanho $N \times N$ (desenvolvido pelo autor).
- **matriz_escalar**: efetua a multiplicação de uma matriz quadrada de tamanho $N \times N$ por um escalar (desenvolvido pelo autor).
- **matriz_vetor**: calcula a multiplicação de uma matriz quadrada de tamanho $N \times N$ por um vetor de tamanho N. É uma adaptação da implementação apresentada por Papatheodore (2017).
- **matriz_trans**: apresenta a transposta de uma matriz quadrada de tamanho $N \times N$. É uma adaptação da implementação apresentada por Harris (2013).
- **mult_diag**: implementa a multiplicação da diagonal de uma matriz quadrada de tamanho $N \times N$ por um escalar (desenvolvido pelo autor).

- **mult_matrizes**: faz a multiplicação de duas matrizes quadradas de tamanho $N \times N$. É uma adaptação da implementação apresentada por Kirk e Wen-Mei (2016).
- **soma_atomica**: código para exemplificar e testar o modelo de teste e ferramenta em um programa que utilize operação atômica (desenvolvido pelo autor).
- **soma_elem**: apresenta a soma de todos os elementos por meio da aplicação do padrão de redução. Possui dois *kernels* com duas formas diferentes de redução. É uma adaptação da implementação apresentada por Cheng, Grossman e McKercher (2014).
- **soma_matriz**: realiza a soma de duas matrizes quadradas de tamanho $N \times N$ (CHENG; GROSSMAN; MCKERCHER, 2014).
- **soma_sinc**: código de exemplo utilizado em aula de programação concorrente, implementa o incremento sincronizado de valores em um vetor de tamanho N (desenvolvido pelo autor).
- **soma_vetores**: realiza a soma de dois vetores de tamanho N (CHENG; GROSSMAN; MCKERCHER, 2014).
- **vetores_iguais**: verifica se dois vetores de tamanho N são iguais (desenvolvido pelo autor).

Os números de blocos por *grid* e de *threads* por bloco nos *benchmarks* foram definidos em um valor fixo. Isso foi feito para permitir avaliar a cobertura alcançada pela execução de vários casos de teste, pois é necessário verificar se uma mesma *thread* cobriu os elementos requeridos em cada caso de teste. Contudo, para cada *benchmark* foi definido um número diferente de *threads*.

Cada um dos *benchmarks* possuem características diferentes, sintetizadas na Tabela 7. O número de *threads* apresentado é a *thread* do processo do *host* somadas as *threads* de cada *grid*. Devido a isso, "soma_elem" que possui 2 *grids* com 1024 *threads* cada, totaliza 2049 *threads*. O mesmo ocorre no número de nós e arestas. O valor expressado é a soma do apresentado no *host* e a quantidade apresentada no *kernel* multiplicado pelo número de *grids* do programa. As colunas "total nós" e "total arestas" apresentam a quantidade de elementos multiplicado pelo número de *threads*. A Complexidade Ciclomática foi calculada com base no número de nós e arestas apresentados. Também apresentamos na tabela algumas características relacionadas ao código do *kernel*, como a presença de laços de repetição com "for", estruturas de decisão com "if" e "else", a presença de variáveis compartilhadas em nível de bloco com o uso da primitiva "shared" e a utilização de operação atômica, especificamente "atomicAdd". Durante todo o processo de avaliação dos critérios, foram utilizados os onze primeiros *benchmarks*. Já os últimos dez *benchmarks*

foram empregados para o desenvolvimento e validação de diversas funcionalidades na ferramenta de teste. Por isso, eles são apresentados apenas nas tabelas e figuras que tratam do custo dos critérios.

Tabela 7 – Lista de programas CUDA e suas características usados como benchmarks para a validação do modelo de teste e dos critérios de teste.

	Código fonte					Código em execução				O código fonte do kernel possui					
Benchmark	**Nº de linhas**	**Nº de nós**	**Nº de arestas**	**Nº de kernels**	**Complexidade ciclomática**	**Nº de grids**	**Nº de threads**	**Total nós**	**Total arestas**	**If**	**Else**	**For**	**Syncthread**	**Variável shared**	**Operação atômica**
matriz_diag	88	54	69	1	17	1	4097	143379	188437	sim	sim	não	não	não	não
soma_elem	118	64	78	2	16	2	2049	37915	46109	sim	não	sim	sim	não	não
maior_menor	106	87	112	2	15	4	4609	78355	96788	sim	não	sim	sim	sim	não
mult_diag	89	43	53	1	12	1	4097	90133	114711	sim	sim	não	não	não	não
mult_matrizes	98	49	58	1	11	1	1025	26647	31769	sim	sim	sim	sim	sim	não
matriz_vetor	80	40	48	1	10	1	257	3610	4381	sim	não	sim	não	não	não
soma_matriz	77	39	47	1	10	1	16385	196635	245790	sim	não	não	não	não	não
conta_ocorrencia	72	33	41	1	10	1	8193	139280	172050	sim	não	sim	sim	sim	não
matriz_escalar	58	23	27	1	6	1	4097	16403	16405	não	não	sim	não	não	não
vetores_iguais	55	24	27	1	5	1	8193	32788	32789	sim	não	não	não	não	não
soma_vetores	47	21	23	1	4	1	1025	2067	1044	não	não	não	não	não	não
3_grids	39	28	35	1	9	3	89	368	369	sim	não	não	não	não	não
soma_sinc	90	40	47	2	9	2	2049	16408	16411	não	não	sim	sim	sim	não
linha_coluna	78	33	39	1	8	1	257	2838	3352	sim	não	sim	não	não	não
2_grids	37	23	28	1	7	2	513	2063	2064	sim	não	não	não	não	não
chamada	37	21	25	1	6	1	257	1041	1043	sim	não	não	não	não	não
matriz_trans	56	17	19	1	4	1	1025	4109	4109	sim	não	não	não	não	não
exemplo	29	14	16	1	4	1	5	26	26	sim	não	não	não	não	não
soma_atomica	24	10	11	1	3	1	4097	8200	4104	não	não	não	não	não	sim
hello	16	5	5	1	2	1	11	23	12	não	não	não	não	não	não

7.3 Planejamento

Nesta seção definimos o processo aplicado na condução dos experimentos. Apresentamos os objetivos do experimento, quais métricas foram usadas para a avaliação, qual o ambiente utilizado para a realização dos experimentos, ameaças à validade dos resultados e como foram gerados os casos de teste. Utilizamos como base nos nossos experimentos, a metodologia de condução de experimentos apresentada por Wohlin *et al.* (2012).

7.3.1 Objetivo dos experimentos

O objetivo dos experimentos é demonstrar que o modelo de teste e critérios de teste propostos são capazes de guiar a escolha de casos de teste que revelam defeitos em programas CUDA. Os critérios avaliados são: todos-nos (TN), todos-nos-grid (TNG), todos-nos-host (TNH), todos-nos-sinc (TNS), todos-nos-sinc-grid (TNSG), todos-nos-sinc-host (TNSH), todas-arestas (TA), todas-arestas-grid (TAG), todas-arestas-host (TAH), todas-arestas-sinc (TAS), todos-c-usos (TCU), todos-c-usos-grid (TCUG), todos-c-usos-host (TCUH), todos-p-usos (TPU), todos-p-usos-grid (TPUG), todos-p-usos-host (TPUH), todos-usos (TU), todos-usos-grid (TUG), todos-usos-host (TUH), todos-s-c-usos (TSCU), todos-s-p-usos (TSPU), todos-s-usos (TSU), todos-bloco-c-usos-grid (TBCUG), todos-bloco-p-usos-grid (TBPUG), todos-global-c-usos-grid (TGCUG) e todos-global-p-usos-grid (TGPUG).

7.3.2 Métricas

Três métricas foram utilizadas para a avaliação dos critérios: custo, eficácia e *strength* (dificuldade de satisfação). Estas métricas são usadas para responder três perguntas: "Qual o custo de aplicação dos critérios de teste?", "Qual a eficácia dos critérios de teste em revelar de programas CUDA?" e "Qual a relação de inclusão dos critérios de teste?". As respostas ajudarão o testador a escolher melhor o critério de teste a ser implementado, conhecendo a sua relação de custo para utilização e a sua eficácia para revelar defeitos.

A avaliação de custo é determinada pelo tempo necessário para se utilizar o critério de teste. O custo pode ser medido pelo tempo necessário para se construir manualmente os casos de teste, o tempo para aprender a utilizar uma ferramenta que permita a aplicação dos critérios, no sentido de analisar associações, caminhos ou mutantes do programa, dentre outras formas (FELIZARDO *et al.*, 2016). Neste experimento o custo de aplicação dos critérios de teste estrutural para programas CUDA foi avaliado pela quantidade de casos de teste adequados para a satisfação de cada critério de teste, número de elementos requeridos e número de elementos requeridos não executáveis para cada critério de teste.

A avaliação de eficácia é a capacidade do critério de teste em revelar defeitos. Ele é calculado com base no número de defeitos inseridos no programa e o número de defeitos revelados pelos casos de teste definidos pelo critério de teste, como apresentado na seguinte equação:

$$Eficacia \ = \ \frac{Numero\ de\ defeitos\ revelados}{Numero\ de\ defeitos\ inseridos} \ * \ 100 \tag{7.1}$$

O *strength* se refere à probabilidade de um determinado critério ser satisfeito com base na satisfação de outro critério, causando uma relação de inclusão entre os critérios. Ele é estimado aplicando o conjunto de casos de teste adequado de um determinado critério$_1$ em um outro critério$_2$, comparando a cobertura alcançada do critério$_2$ com o critério$_1$. Dessa forma, se $cobertura(criterio_2) > cobertura(criterio_1)$, então o critério$_2$ possui um *strength* maior.

7.3.3 Hipóteses

Com base nas métricas de avaliação escolhidas, uma pergunta foi definida para cada métrica, buscando respostas por meio de uma hipótese nula e uma hipótese alternativa.

"Qual o custo de aplicação dos critérios de teste para programas concorrentes que utilizam o modelo de programação CUDA?"

Hipótese Nula($\mathbf{HN_1}$): Não há diferença de custo entre os critérios de teste estrutural definidos para programas concorrentes que utilizam o modelo de programação CUDA.

$$HN_1 : Custo(criterio_i) = Custo(criterio_{i+1})$$

Hipótese Alternativa($\mathbf{HA_1}$): O custo é diferente para ao menos um dos critérios de teste estrutural definidos para programas concorrentes que utilizam o modelo de programação CUDA.

$$HA_1 : Custo(criterio_i) \neq Custo(criterio_{i+1})$$

"Qual a eficácia dos critérios de teste em revelar defeitos em programas concorrentes que utilizam o modelo de programação CUDA?"

Hipótese Nula($\mathbf{HN_2}$): Não há diferença de eficácia entre os critérios de teste estrutural definidos para programas concorrentes que utilizam o modelo de programação CUDA.

$$HN_2 : Eficacia(criterio_i) = Eficacia(criterio_{i+1})$$

Hipótese Alternativa($\mathbf{HA_2}$): A eficácia é diferente para ao menos um dos critérios de teste estrutural definidos para programas concorrentes que utilizam o modelo de programação CUDA.

$$HA_2 : Eficacia(criterio_i) \neq Eficacia(criterio_{i+1})$$

"Qual a relação de inclusão dos critérios de teste para programas concorrentes que utilizam o modelo de programação CUDA?"

Hipótese Nula($\mathbf{HN_3}$): Não há relação de inclusão entre os critérios de teste estrutural definidos para programas concorrentes que utilizam o modelo de programação CUDA.

$$HN_3 : ConjuntoCasosTeste(criterio_i) \not\subset ConjuntoCasosTeste(criterio_{i+1})$$

Hipótese Alternativa($\mathbf{HA_3}$): Ao menos um dos critérios de teste estrutural definidos para programas concorrentes CUDA inclui outro critério.

$$HA_3 : ConjuntoCasosTeste(criterio_i) \subset ConjuntoCasosTeste(criterio_{i+1})$$

7.3.4 Variáveis

- **Independentes**:
  - Critérios de teste
  - *Benchmarks* selecionados
  - Conjunto de defeitos inseridos
  - Metodologia para inserção de defeitos
  - Metodologia para geração dos casos de teste

- **Dependentes**:
  - Número de elementos requeridos
  - Número de elementos requeridos não executáveis
  - Número de casos de teste adequados
  - Cobertura alcançada
  - Custo dos critérios
  - Eficácia dos critérios
  - *Strength* dos critérios

7.3.5 Planejamento dos experimentos

Para a realização dos experimentos são necessários alguns passos com base nos objetivos, métricas e hipóteses definidos. Primeiramente é necessário preparar o ambiente de execução para a realização dos experimentos. Após, é necessário fazer a inserção de defeitos nos *benchmarks* escolhidos para permitir realizar a análise da eficácia dos critérios. Por fim, é necessário fazer a geração dos casos de teste.

7.3.5.1 Ambiente de realização dos experimentos

Para a realização dos experimentos foi configurado e utilizado um ambiente com a seguinte configuração:

- Intel Core i7-7700HQ, 16GB RAM
- Sistema Operacional Ubuntu 20.04 por meio do WSL2 (Subsistema Windows para Linux)
- Clava *framework*
- ClavaCUDA
- ValiMPI
- GCC (GNU Compiler Collection) 9.3.0
- NVCC (Nvidia CUDA Compiler) 11.4.100
- CUDA 11.4
- Nvidia Driver 471.68

7.3.5.2 Semeadura de defeitos

Para o desenvolvimento e teste da ferramenta foram utilizados 20 *benchmarks* apresentados na Tabela 7, entretanto para a avaliação de custo e eficácia foi utilizado um subconjunto com 11 *benchmarks* desta seleção. Isso foi feito devido ao fato dos outros 09 *benchmarks* terem sido desenvolvidos para testar recursos mais específicos durante o desenvolvimento. Os seguintes programas foram selecionados para a inserção de defeitos: vetores_iguais, soma_elem, maior_menor, mult_matrizes, matriz_vetor, matriz_escalar, soma_matriz, mult_diag, matriz_diag e soma_vetores, conta_ocorrencia.

Para a avaliação de eficácia dos critérios foram inseridos defeitos nos *benchmarks* com base na relação de defeitos apresentada no Capítulo 4. Os defeitos foram inseridos manualmente em cada programa utilizando o conceito dos operadores de mutação definidos por Zhu e Zaidman (2020). Para cada defeito inserido foi gerado um arquivo distinto, de forma que nenhum código teve dois ou mais defeitos inseridos simultaneamente. A quantidade e tipos de defeitos inseridos em cada programa é apresentado na Tabela 8. O tipo de defeito inserido levou em consideração as características do programa.

Tabela 8 – Número de defeitos inseridos em cada benchmark.

	Benchmark											
Defeito	**maior menor**	**soma elem**	**mult matrizes**	**matriz escalar**	**soma vetores**	**vetores iguais**	**matriz diag**	**soma matriz**	**mult diag**	**matriz vetor**	**conta ocorrencia**	**Total**
Erro de memória	1	0	2	2	0	0	3	0	0	0	0	**8**
Falta de sincronização	4	4	2	0	0	0	0	0	0	1	2	**13**
Erro de variável compartilhada	2	0	4	0	0	0	0	1	1	0	1	**9**
ID incorreto da thread	2	3	3	0	1	0	2	0	0	1	0	**12**
Uso incorreto de variável global	1	0	0	1	1	1	0	1	1	1	0	**7**
Configuração incorreta do kernel	2	1	0	0	1	1	0	0	0	0	1	**6**
Total	**12**	**8**	**11**	**3**	**3**	**2**	**5**	**2**	**2**	**3**	**4**	**55**

O Código 17 apresenta um exemplo de defeito inserido. No fragmento de código, um defeito foi inserido na linha 2 onde, na variável compartilhada *s_b* os índices de acesso à variável foram invertidos. Sendo um defeito relacionado ao uso de variável, em uma situação condicional restrita, a revelação desse defeito pode ser difícil de ser revelada dependendo do critério de teste utilizado. Em contrapartida, o Código 18 apresenta o mesmo código, sem a presença do defeito.

Código 17 – Código exemplo com um defeito inserido.

```
1 for (i = 0; i < TILE_DIM; i++) {
2     if(s_a[threadIdx.y][i] != 0 && s_b[threadIdx.x][i] != 0){ // defeito
3         temp_result += s_a[threadIdx.y][i] * s_b[i][threadIdx.x];
4     }
5 }
```

Código 18 – Código exemplo sem a inserção de defeito.

```
for (i = 0; i < TILE_DIM; i++) {
    if(s_a[threadIdx.y][i] != 0 && s_b[i][threadIdx.x] != 0){
        temp_result += s_a[threadIdx.y][i] * s_b[i][threadIdx.x];
    }
}
```

7.3.5.3 Geração dos casos de teste

Para realizar a avaliação dos critérios de teste listados na Seção 7.3.1 com base nas métricas de custo, eficácia e *strength*, é necessário gerar um conjunto de casos de teste adequado, que busque atingir a maior taxa de cobertura possível de cada critério. A execução do programa com todos os valores de entrada possíveis, embora desejado, é inviável. Dessa forma, é necessário adotar estratégias para a geração dos casos de teste, de forma que atinja a maior cobertura, mas sem a presença de casos de teste redundantes no conjunto. Para a geração dos casos de teste foram executados os seguintes passos no experimento:

- **Instrumentação do código**: o código de cada programa foi instrumentado utilizando a ferramenta ClavaCUDA descrita na Seção 6. Junto com a instrumentação são geradas as informações relacionadas aos nós, arestas, comunicação, definição e uso de variáveis. O código instrumentado é então compilado utilizando o NVCC;

- **Geração dos elementos requeridos**: Com base nas informações geradas pela ClavaCUDA, usamos o módulo ValiElem da ValiMPI para gerar os elementos requeridos de cada critério de teste para todos os *benchmarks* selecionados. O número dos elementos requeridos nesta etapa são utilizados para a análise de custo dos critérios, entretanto é necessário considerar o número de *threads*, pois a ValiElem gera um elemento requerido para cada *grid* e não na granularidade de *thread*, dessa forma diminuindo o tamanho dos artefatos gerados;

- **Seleção dos casos de teste**: os casos de teste foram selecionados para cada critério de todos os *benchmarks* selecionados buscando atingir a maior cobertura. Para fazer a seleção foi utilizado o particionamento em classes de equivalência e a análise do valor limite. Devido ao número fixo de *threads* não foram utilizados casos fora do limite. Com isso, foi selecionado o dado de entrada e aplicado ao programa, analisando a taxa de cobertura alcançada. Um novo caso de teste foi adicionado buscando alcançar uma cobertura maior. Ao alcançar a máxima cobertura, foi verificado se algum dos casos de teste se tornou redundante em relação aos outros, neste caso, sendo removido do conjunto. Ao final, temos um conjunto de casos de teste adequado para cada programa. Esse conjunto é utilizado posteriormente para avaliar a capacidade dos critérios de revelar defeitos nos programas com defeitos inseridos;

- **Execução dos casos de teste**: o módulo ValiExec é utilizado para executar cada caso de teste em cada programa. Ao final das execuções, temos a saída do programa para cada caso de teste, permitindo analisar se o resultado está correto. Também são gerados os rastros de execução do *host* e de todos os *grids* que permitem avaliar a cobertura alcançada;

- **Avaliação da taxa de cobertura alcançada pelos casos de teste**: utilizando o módulo ValiEval, a taxa de cobertura alcançada é avaliada para cada caso de teste de cada programa. Isto avaliar a cobertura parcial alcançada para cada caso de teste. A ValiEval ainda apresenta a taxa de cobertura total alcançada pela sobreposição da execução de todos os casos de teste;

- **Identificação dos elementos requeridos não executáveis**: após a avaliação da cobertura, os elementos requeridos que não foram cobertos são analisados manualmente pelo testador. É preciso verificar se é necessário adicionar um novo caso de teste que permita cobrir tal execução, dessa forma aumentando a cobertura total do conjunto de casos de teste. Se isso não é possível, o elemento requerido é marcado como não executável;

- **avaliação do custo**: de posse do número de elementos requeridos, número de elementos requeridos não executáveis e número de casos de teste adequados para cada critério de cada programa, obtidos nos passos anteriores, é feita a avaliação de custo de cada critério;

- **avaliação de eficácia**: para a avaliação da eficácia, executamos todas as suas versões com defeitos inseridos utilizando o conjunto de casos de teste adequados. Devido à característica não determinística, os programas foram executados até dez vezes para verificar se a saída é igual à apresentada pelo código original. Com base nisso, verificamos se cada critério conseguiu revelar o defeito inserido no programa. Averiguando todos os defeitos revelados para cada critério e com base no número de defeitos inseridos, é calculada a eficácia alcançada para cada critério e para cada tipo de defeito; e

- **avaliação do *strength***: para a avaliação do *strength* foi realizada a execução dos *benchmarks* utilizando o conjunto de casos de teste de cada critério para obter a cobertura atingida para todos os outros critérios. Assim, comparando a cobertura atingida entre os critérios, em uma relação todos com todos, obtêm-se o *strength* de cada critério.

7.3.6 Ameaças à validade

As seguintes ameaças à validade foram consideradas neste trabalho.

- **Validade de conclusão**: As conclusões apresentadas foram feitas com base na metodologia de experimentos apresentada por Wohlin *et al.* (2012). Utilizamos métodos estatísticos para avaliar as hipóteses em relação aos dados coletados durante o processo do experimento. Utilizamos o programa Minitab [1] para realizar o teste de normalidade Ryan-Joiner, semelhante ao teste de Shapiro-Wilk, para verificar se os dados seguem uma distribuição normal ou não. Em caso afirmativo, utilizamos o teste ANOVA (*ANalysis Of VAriance*) para verificar se há provas suficientes para rejeitar a hipótese nula. No caso da distribuição não ser normal, utilizamos o teste não paramétrico Kruskal-Wallis. Todos os testes com nível de significância de 0,05.

- **Validade interna**: a validade interna pode ser comprometida por diversos fatores, como a instrumentação, geração dos casos de teste, inserção de defeitos e seleção dos *benchmarks*. Nós desenvolvemos o módulo de instrumentação do código ClavaCUDA, buscando mitigar possíveis problemas e usamos o *framework* Clava que auxilia no processo de instrumentação e extração de informações do código. Para mitigar possíveis problemas, também realizamos o processo manualmente e comparamos com o apresentado pela ferramenta, buscando identificar e corrigir possíveis erros. Para minimizar problemas na geração dos casos de teste buscamos utilizar critérios de seleção de dados com base nas classes de equivalência e análise do valor limite para construir um conjunto de casos de teste adequado aos critérios. Buscando minimizar a ameaça de validação interna na seleção e inserção de defeitos nos programas, nós nos baseamos nas classificações de defeitos de outros trabalhos relacionados, e utilizamos operadores de mutação para realizar a inserção de defeitos nos programas selecionados. Os programas selecionados também podem ser uma ameaça à validade. Nós buscamos selecionar programas desenvolvidos por terceiros, que possuam características diferentes, entretanto não foram abordadas todas as características presentes no modelo de programação CUDA (vide Seções 9.4 e 9.5).

- **Validade externa**: uma ameaça à validade externa são os programas selecionados não representarem amplamente os tipos de programas desenvolvidos e utilizados em CUDA. Nós buscamos selecionar programas que realizem computações diferentes, façam o uso de primitivas e características diferentes do modelo de programação, buscando mitigar essa ameaça, entretanto há recursos que não foram cobertos, o que impede que as conclusões desse estudo possam ser generalizadas para outros programas com características distintas das apresentadas no conjunto de *benchmarks* selecionado.

- **Validade de construção**: o favorecimento a uma das hipóteses ou critérios pode ser uma ameaça, assim como o conhecimento do código fonte do programa. Para mitigar o problema foi utilizada a ferramenta ValiMPI que gera automaticamente

[1] https://www.minitab.com/

os elementos requeridos e avalia a cobertura alcançada para cada critério. Para minimizar problemas relacionados ao conhecimento do código fonte foram utilizadas metodologias para a inserção dos defeitos nos programas e para a geração dos casos de teste utilizados.

7.4 Resultados Obtidos

Nesta seção apresentamos os resultados obtidos oriundos da execução dos passos descritos na Seção 7.3 para a realização dos experimentos. Instrumentamos todos os programas originais e com defeitos inseridos utilizando o Clava *framework* e o *script* ClavaCUDA. Geramos os elementos requeridos dos critérios de teste utilizando o módulo ValiElem da ferramenta ValiMPI. Executamos todos os 20 programas com os casos de teste gerados, utilizando o módulo ValiExec para gerar os rastros de execução e a saída dos programas. Avaliamos a cobertura alcançada para cada critério de teste pelo conjunto dos casos de teste utilizando a ValiEval.

A Tabela 9 apresenta a taxa de cobertura alcançada para cada critério proposto em relação a todos os *benchmarks* listados na Seção 7.2. A taxa de cobertura considera todos os elementos requeridos executáveis e não executáveis gerados pela ValiElem. Nos casos em que alcançaram 100% de cobertura, não há elementos não executáveis para o critério. Nos casos onde é apresentado o caractere "-", nenhum elemento requerido foi gerado para o critério.

Na Seção 7.4.1 mostramos o custo dos critérios em relação ao número de elementos requeridos, elementos requeridos não executáveis e número de casos de teste adequados a cada critério. Na Seção 7.4.2 apresentamos a eficácia alcançada pelos critérios de teste em revelar os defeitos inseridos nos programas demonstrados na Tabela 8. Na Seção 7.4.3 mostramos a relação de *strength* de cada critério para cada *benchmark*. Nas avaliações de custo, eficácia e *strength* foram utilizados o subconjunto dos *benchmarks*, conforme listado na semeadura de defeitos.

7.4.1 Resultado do custo dos critérios

Para a métrica de desempenho custo foi utilizado o número de elementos requeridos para cada critério de teste, o número de elementos requeridos não executáveis e o número de casos de teste adequados para os critérios. Os elementos requeridos foram gerados pelo módulo ValiElem e o total de elementos requeridos foi contabilizada pela ValiEval, considerando a quantidade de *threads* de cada *grid*. O número de casos de teste foram definidos com base na cobertura dos critérios de teste e a quantidade de elementos requeridos não executáveis foi contabilizada conforme não foram executados e após verificação manual.

A Tabela 10 apresenta a quantidade de elementos requeridos e elementos requeridos não executáveis de cada *benchmark* selecionado. Os onze primeiros *benchmarks* são os

Tabela 9 – Cobertura atingida para cada critério.

Benchmark	Critério (%)												
	TN	TNG	TNH	TNS	TNSG	TNSH	TA	TAG	TAH	TAS	TCU	TCUG	TCUH
vetores_iguais	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
soma_elem	91,90	91,90	100,00	100,00	100,00	100,00	85,00	81,20	100,00	100,00	65,50	65,40	100,00
maior_menor	84,30	84,30	100,00	100,00	100,00	100,00	75,90	69,10	100,00	100,00	51,30	51,30	100,00
mult_matrizes	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	96,00	96,00	100,00
matriz_vetor	100,00	100,00	100,00	100,00	100,00	100,00	99,90	99,90	100,00	100,00	92,40	92,30	100,00
matriz_escalar	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	94,40
soma_matriz	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
mult_diag	95,60	95,60	100,00	100,00	100,00	100,00	93,50	92,50	100,00	100,00	87,60	87,60	100,00
matriz_diag	88,60	88,60	100,00	100,00	100,00	100,00	77,80	75,00	100,00	100,00	70,00	70,00	100,00
soma_vetores	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	-	100,00
conta_ocorr	91,20	91,20	100,00	100,00	100,00	100,00	83,30	78,50	100,00	100,00	48,10	48,10	100,00
matriz_trans	100,00	100,00	100,00	100,00	100,00	100,00	99,98	99,95	100,00	100,00	99,98	100,00	92,86
linha_coluna	100,00	100,00	100,00	100,00	100,00	100,00	99,92	99,87	100,00	100,00	88,260	88,240	92,310
soma_atomica	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
exemplo	100,00	100,00	100,00	100,00	100,00	100,00	94,44	87,50	100,00	100,00	100,00	100,00	100,00
2_grids	87,59	87,50	100,00	100,00	100,00	100,00	87,52	75,00	100,00	100,00	83,44	83,33	100,00
3_grids	79,89	78,98	100,00	100,00	100,00	100,00	75,21	50,00	100,00	100,00	60,22	57,96	100,00
soma_sinc	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	90,01	90,00	100,00
hello	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	-	100,00
chamada	87,70	87,50	100,00	100,00	100,00	100,00	75,02	50,00	80,00	100,00	83,55	83,33	100,00
Benchmark	TPU	TPUG	TPUH	TU	TUG	TUH	TSCU	TSPU	TSU	TBCUG	TBPUG	TGCUG	TGPUG
vetores_iguais	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	-	-	100,00	100,00
soma_elem	97,20	97,20	100,00	75,50	75,50	100,00	100,00	100,00	83,30	-	-	100,00	100,00
maior_menor	64,10	64,10	100,00	57,50	57,50	100,00	89,10	87,50	82,70	11,20	23,20	89,10	87,10
mult_matrizes	89,60	89,60	100,00	93,30	93,30	100,00	100,00	100,00	100,00	61,70	63,80	100,00	100,00
matriz_vetor	99,80	99,80	100,00	96,70	96,70	100,00	100,00	99,80	99,90	100,00	-	100,00	99,80
matriz_escalar	100,00	100,00	100,00	100,00	100,00	96,40	100,00	100,00	100,00	-	-	100,00	100,00
soma_matriz	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
mult_diag	100,00	100,00	100,00	90,50	90,50	100,00	89,20	100,00	87,90	3,10	-	89,20	100,00
matriz_diag	100,00	100,00	100,00	82,60	82,60	100,00	67,20	100,00	69,20	-	-	67,20	100,00
soma_vetores	100,00	-	100,00	100,00	-	100,00	-	-	100,00	-	-	-	-
conta_ocorr	87,50	87,50	100,00	69,80	69,80	100,00	-	100,00	83,30	20,80	-	-	100,00
matriz_trans	99,95	99,95	100,00	99,96	99,97	93,75	100,00	99,95	99,98	-	-	100,00	99,95
linha_coluna	99,87	99,87	100,00	93,06	93,05	94,44	100,00	99,87	99,89	-	-	100,00	99,87
soma_atomica	100,00	-	100,00	100,00	100,00	100,00	-	-	100,00	-	-	100,00	-
exemplo	91,67	87,50	100,00	96,55	95,00	100,00	100,00	87,50	93,75	-	-	100,00	87,50
2_grids	100,00	-	100,00	83,48	83,33	100,00	100,00	-	87,50	-	-	100,00	-
3_grids	100,00	-	100,00	61,05	57,96	100,00	0,00	-	75,00	-	-	-	-
soma_sinc	100,00	100,00	100,00	91,67	91,67	100,00	-	-	100,00	75,00	-	-	-
hello	-	-	-	100,00	-	100,00	-	-	-	-	-	-	-
chamada	50,39	50,00	83,33	70,29	70,00	93,75	100,00	-	75,00	-	-	100,00	-

utilizados nas análises de custo e eficácia. Podemos observar que para os critérios todos-bloco-c-uso-grid (*TBCUG*) e todos-bloco-p-uso-grid (*TBPUG*) há poucos programas com elementos requeridos. Isso é devido à necessidade de haver variáveis compartilhadas em nível de bloco no *kernel* e com uso computacional ou predicativo. Concomitantemente há um alto número de elementos não executáveis nesses critérios devido ao seu uso depender de condições bastante restritivas de estruturas de decisão, dessa forma não sendo possível executá-los. No caso específico do programa "hello", não houve a passagem de variáveis para o *kernel*, dessa forma não foram gerados elementos requeridos relacionados à comunicação entre o *host* e o *grid* para os critérios todos-s-usos, todos-s-c-usos e todos-s-p-usos.

A Figura 43 apresenta graficamente o número de elementos requeridos dos critérios

Tabela 10 – Elementos requeridos / elementos requeridos não executáveis.

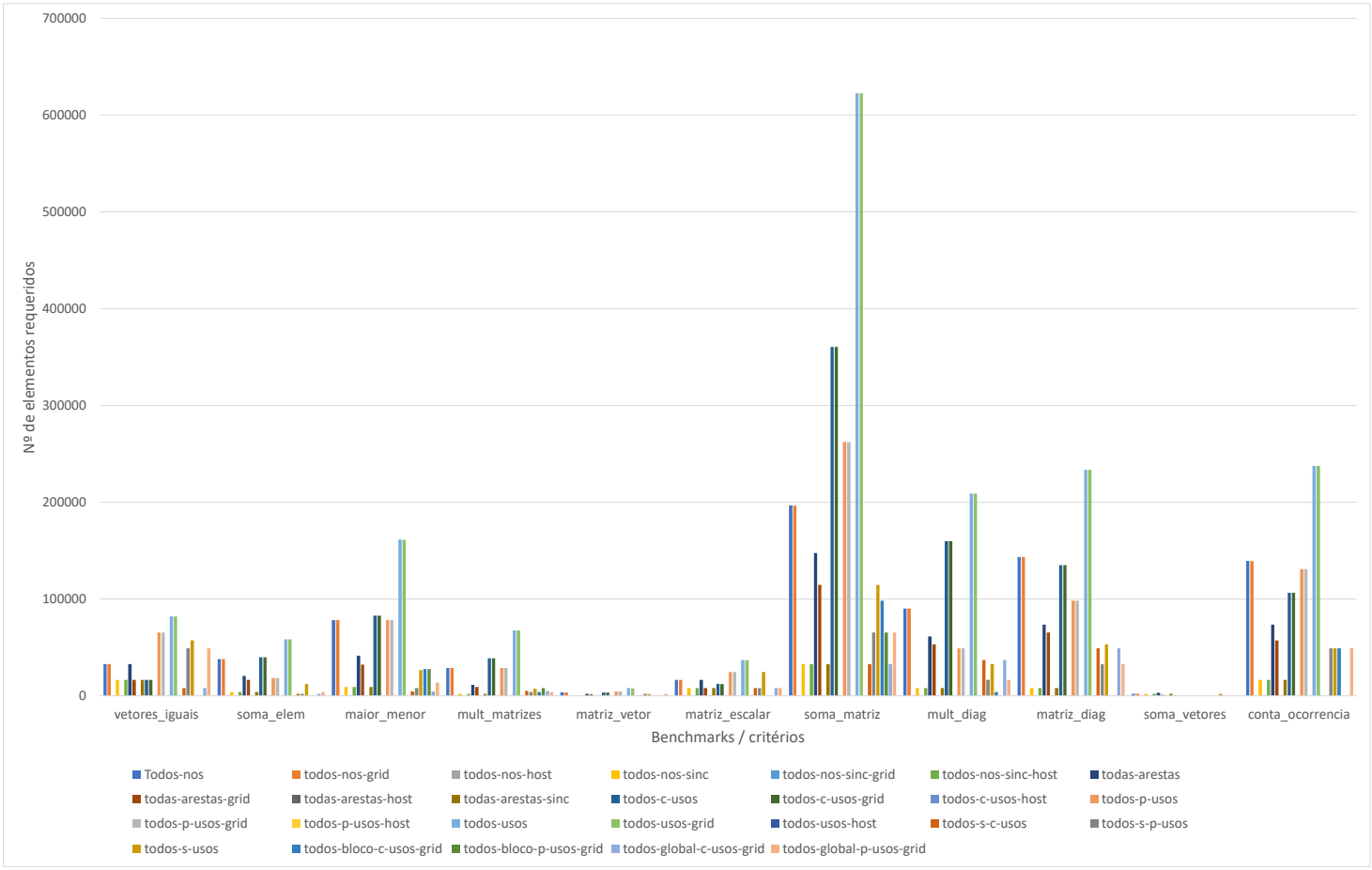
Benchmark	Critério												
	TN	TNG	TNH	TNS	TNSG	TNSH	TA	TAG	TAH	TAS	TCU	TCUG	TCUH
vetores_iguais	32788/0	32768/0	20/0	16395/0	16384/0	11/0	32771/0	16384/0	3/0	16384/0	16403/0	16384/0	19/0
soma_elem	37915/3066	37888/3066	27/0	4110/0	4096/0	14/0	20484/3082	16384/3082	4/0	4096/0	39970/13808	39936/13808	34/0
maior_menor	78355/12262	78336/12262	19/0	9226/0	9216/0	10/0	41475/9979	32256/9979	3/0	9216/0	82963/40370	82944/40370	19/0
mult_matrizes	28699/0	28672/0	27/0	2058/0	2048/0	10/0	11269/1	9216/1	5/0	2048/0	38937/1568	38912/1568	25/0
matriz_vetor	3610/0	3584/0	26/0	522/0	512/0	10/0	2053/2	1536/2	5/0	512/0	3353/256	3328/256	25/0
matriz_escalar	16404/0	16384/0	20/0	8199/0	8192/0	7/0	16388/1	8192/1	4/0	8192/0	12306/1	12288/0	18/1
soma_matriz	196635/0	196608/0	27/0	32778/0	32768/0	10/0	147461/2	114688/2	5/0	32768/0	360473/0	360448/0	25/0
mult_diag	90133/3968	90112/3968	21/0	8200/0	8192/0	8/0	61444/3970	53248/3970	4/0	8192/0	159762/19840	159744/19840	18/0
matriz_diag	143379/16382	143360/16382	19/0	8199/0	8192/0	7/0	73732/16386	65536/16386	4/0	8192/0	135186/40571	135168/40571	18/0
soma_vetores	2067/0	2048/0	19/0	2058/0	2048/0	10/0	3075/0	1024/0	3/0	2048/0	10/0	-/-	10/0
conta_ocorrencia	139280/12280	139264/12280	16/0	16389/0	16384/0	5/0	73732/12297	57344/12297	4/0	16384/0	106510/55280	106496/55280	14/0
matriz_trans	4109/0	4096/0	13/0	2056/0	2048/0	8/0	4098/1	2048/1	2/0	2048/0	5134/1	5120/0	14/1
linha_coluna	11291/0	11264/0	27/0	2058/0	2048/0	10/0	5125/1	3072/1	5/0	2048/0	17437/2050	17408/2048	29/2
soma_atomica	8200/0	8192/0	8/0	8194/0	8192/0	2/0	12290/0	4096/0	2/0	8192/0	8194/0	8192/0	2/0
exemplo	26/0	16/0	10/0	13/0	8/0	5/0	18/1	8/1	2/0	8/0	17/0	12/0	5/0
2_grids	2063/256	2048/256	15/0	1030/0	1024/0	6/0	2051/256	1024/256	3/0	1024/0	1546/256	1536/256	10/0
3_grids	368/74	352/74	16/0	183/0	176/0	7/0	355/88	176/88	3/0	176/0	186/74	176/74	10/0
soma_sinc	16408/0	16384/0	24/0	4104/0	4096/0	8/0	8197/0	4096/0	5/0	4096/0	20490/2048	20480/2048	10/0
hello	23/0	20/0	3/0	2/0	20/0	2/0	11/0	10/0	1/0	20/0	1/0	-/-	1/0
chamada	1041/128	1024/128	17/0	518/0	512/0	6/0	1029/257	512/256	5/1	512/0	778/128	768/128	10/0
Benchmark	TPU	TPUG	TPUH	TU	TUG	TUH	TSCU	TSPU	TSU	TBCUG	TBPUG	TGCUG	TGPUG
vetores_iguais	65544/0	65536/0	8/0	81947/0	81920/0	27/0	8192/0	49152/0	57344/0	-/-	-/-	8192/0	49152/0
soma_elem	18444/520	18432/520	12/0	58414/14328	58368/14328	46/0	2048/0	2048/1	12288/2048	-/-	-/-	2048/0	4096/2
maior_menor	78344/28145	78336/28145	8/0	161307/68515	161280/68515	27/0	4608/504	8192/1025	26624/4608	27648/24544	27648/21237	4608/504	13824/1786
mult_matrizes	28684/2969	28672/2969	12/0	67621/4537	67584/4537	37/0	5120/0	4096/2	7168/1	4096/1568	8192/2963	5120/0	4096/2
matriz_vetor	4624/7	4608/7	16/0	7977/263	7936/263	41/0	768/0	2048/3	1792/1	256/0	-/-	768/0	2048/3
matriz_escalar	24586/3	24576/3	10/0	36892/4	36864/3	28/1	8192/0	8192/1	24576/1	-/-	-/-	8192/0	8192/1
soma_matriz	262156/6	262144/6	12/0	622629/6	622592/6	37/0	32768/0	65536/2	114688/1	98304/0	65536/0	32768/0	65536/2
mult_diag	49162/6	49152/6	10/0	208924/19846	208896/19846	28/0	36864/3968	16384/2	32768/3968	4096/3968	-/-	36864/3968	16384/2
matriz_diag	98314/12	98304/12	10/0	233500/40583	233472/40583	28/0	49152/16126	32768/4	53248/16383	-/-	-/-	49152/16126	32768/4
soma_vetores	8/0	-/-	8/0	18/0	-/-	18/0	-/-	-/-	2048/0	-/-	-/-	-/-	-/-
conta_ocorrencia	131082/16402	131072/16402	10/0	237592/71682	237568/71682	24/0	-/-	49152/1	49152/8192	49152/38904	-/-	-/-	49152/1
matriz_trans	8194/4	8192/4	2/0	13328/5	13312/4	16/1	3072/0	4096/2	6144/1	-/-	-/-	3072/0	4096/2
linha_coluna	12302/4	12288/4	14/0	29739/2054	29696/2052	43/2	4096/0	6144/2	7168/2	-/-	-/-	4096/0	6144/2
soma_atomica	4/0	-/-	4/0	8198/0	8192/0	6/0	-/-	-/-	8192/0	-/-	-/-	4096/0	-/-
exemplo	12/1	8/1	4/0	29/1	20/1	9/0	4/0	8/1	16/1	-/-	-/-	4/0	8/1
2_grids	4/0	-/-	4/0	1550/256	1536/256	14/0	256/0	-/-	2048/256	-/-	-/-	512/0	-/-
3_grids	4/0	-/-	4/0	190/74	176/74	14/0	64/64	-/-	352/88	-/-	-/-	-/-	-/-
soma_sinc	4102/0	4096/0	6/0	24592/2048	24576/2048	16/0	-/-	-/-	4096/0	8192/2048	-/-	-/-	-/-
hello	-/-	-/-	-/-	1/0	-/-	1/0	-/-	-/-	-/-	-/-	-/-	-/-	-/-
chamada	518/257	512/256	6/1	1296/385	1280/384	16/1	256/0	-/-	1024/256	-/-	-/-	256/0	-/-

para cada um dos programas usados. Podemos observar que os programas “soma_matriz”, "mult_diag", “matriz_diag” e "conta_ocorrencia"possuem os maiores números de elementos. Isso é devido primeiramente a quantidade de *threads* que ambos possuem. Sendo um fator multiplicador de elementos requeridos para os critérios relacionados ao *grid*, ele afeta significativamente o custo. A complexidade do *benchmark* é outro fator, sendo expresso na quantidade de nós, arestas e de definições e uso de variáveis. O programa “mult_matrizes” possui o *kernel* com a maior quantidade de nós e arestas, mas devido à quantidade menor de *threads* em relação aos *benchmarks* citados, apresentou um custo de elementos requeridos menor. Na Figura 44, apresentamos o custo em escala logarítmica, onde podemos observar que o programa soma_vetores possui o menor custo. No entanto, é possível notar uma diferença significativa de custo entre os critérios derivados do fluxo de controle em comparação com os critérios de fluxo de dados. Essa diferença ocorre devido ao fato do *kernel* do programa apresentar definição e uso de variáveis no mesmo nó do CFG. Como resultado, a ferramenta de teste não cria os elementos requeridos para otimizar a atividade de teste nesse caso. Nas Figuras apresentadas, alguns critérios estão agrupados em pares devido à relação de inclusão entre eles. Por exemplo, o critério todos-nos-grid e todos-nos-host estão incluídos no critério todos-nos. Como o número de elementos requeridos pelos

critérios relacionados ao *host* é consideravelmente menor do que os do *grid*, no gráfico os critérios do *grid* acabam se equiparando aos critérios gerais que estão incluídos.

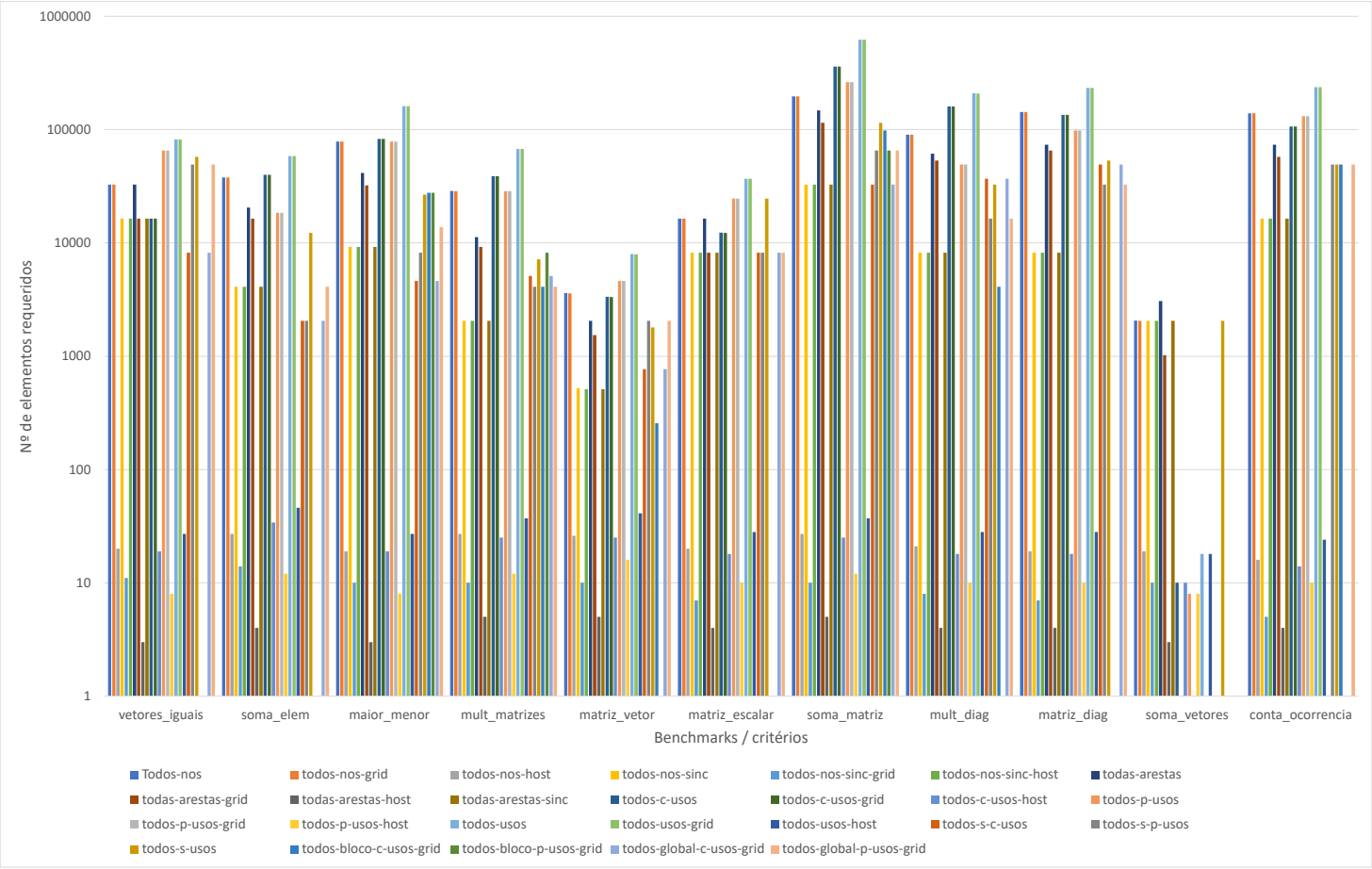
Na Figura 45 apresentamos o custo considerando os critérios de teste. Podemos observar que os critérios relacionados ao *host* possuem o menor custo, devido ao *host* possuir uma única *thread* de execução. Os critérios todos-usos e todos-usos-grid apresentaram o maior custo de elementos requeridos, seguido por todos-c-usos, todos-c-usos-grid, todos-p-usos, e todos-p-usos-grid. A Figura 46 mostra os mesmos dados, mas em escala logarítmica. É interessante observar que entre os critérios relacionados ao *host*, o critério todas-arestas-host apresentou o menor custo. Essa vantagem pode ser explicada pela otimização que a ferramenta de teste realiza em relação aos elementos requeridos, utilizando o conceito das arestas primitivas (CHUSHO, 1987). Esse conceito ajuda a reduzir o custo do critério, o que pode explicar o resultado obtido.

Figura 43 – Custo dos elementos requeridos dos critérios de teste para cada um dos programas selecionados.



O custo dos elementos requeridos não executáveis é apresentado na Figura 47. Os programas com um *kernel* mais simples, com pouca ou nenhuma estrutura de repetição e

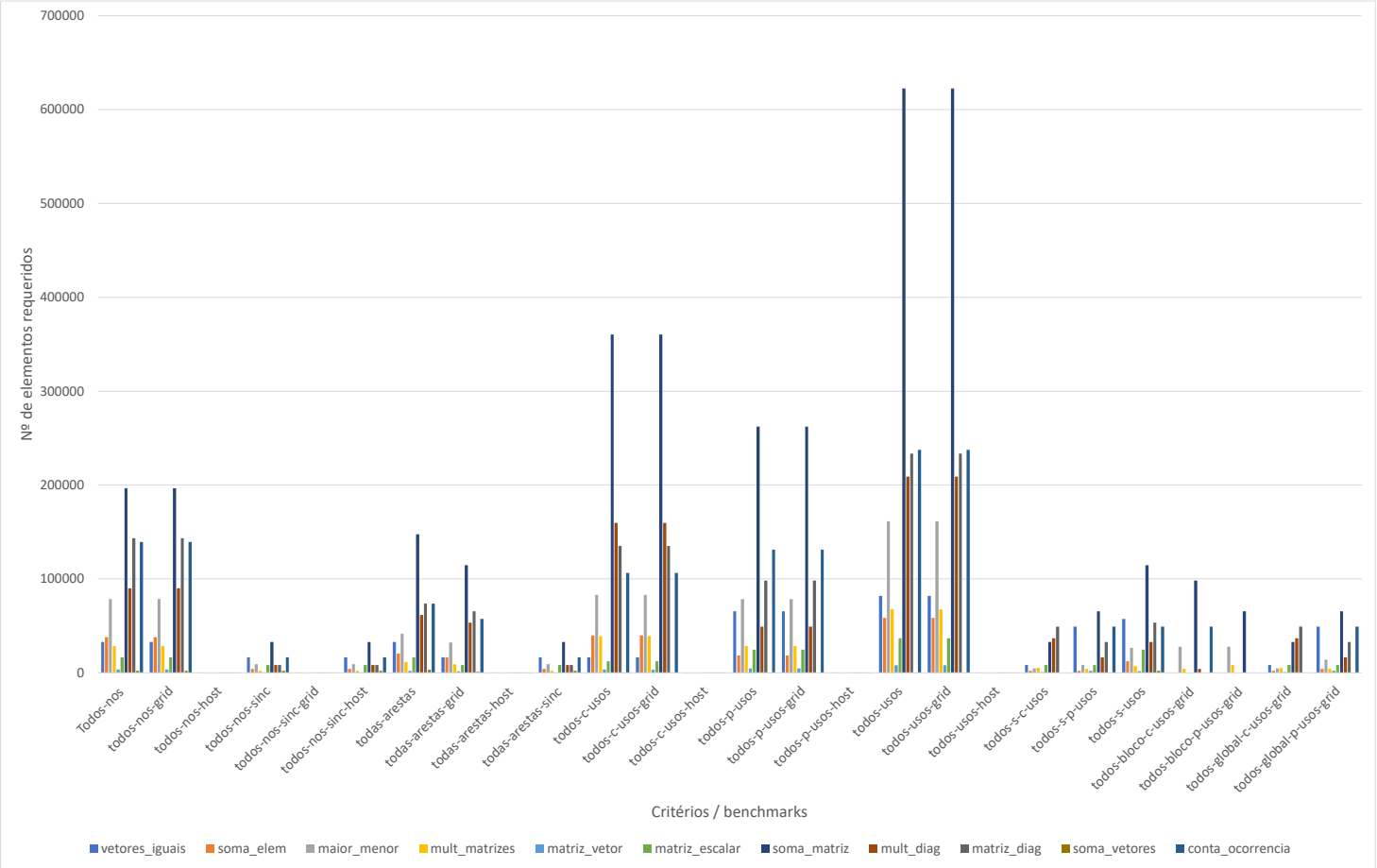
Figura 44 – Custo dos elementos requeridos em escala logarítmica dos critérios de teste para cada um dos programas selecionados.



decisão apresentaram uma quantidade reduzida de elementos não executáveis. Os programas “maior_menor” e "conta_ocorrencia"apresentaram o maior número de elementos não executáveis devido ao uso de estruturas de decisão que restringem o número de *threads* que podem executar esses elementos. Isto pode ser observado também na Tabela 9 que apresenta a cobertura atingida pelos critérios relacionados à definição e ao uso de variáveis. A Figura 48 apresenta o custo dos elementos não executáveis em escala logarítmica. É possível notar que os programas vetores_iguais e soma_vetores não apresentaram nenhum elemento não executável devido ao baixo nível de complexidade de seus *kernels*. Já os programas matriz_escalar e soma_matriz apresentaram um baixo número de elementos não executáveis devido a condições específicas onde alguma *thread* não consegue executar esses elementos.

Na Figura 49 apresenta a relação dos elementos requeridos não executáveis considerando os critérios de teste. Ele corrobora com a cobertura apresentada, onde os critérios relacionados à definição e ao uso de variável apresentaram maior número de elementos

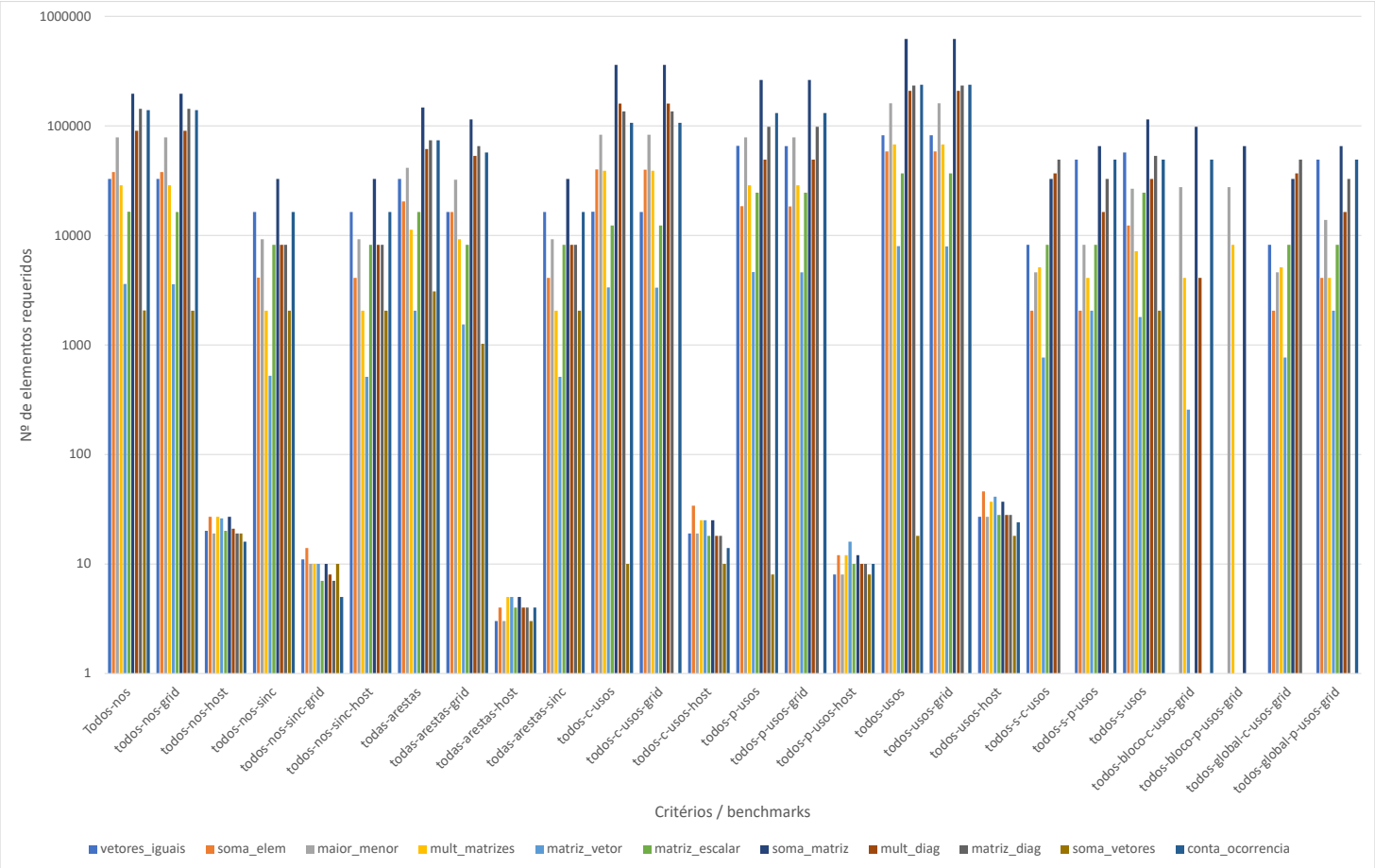
Figura 45 – Custo dos elementos requeridos para cada critério de teste.



não executáveis, com destaque para os critérios todos-usos e todos-usos-grid. A Figura 50 mostra os mesmos dados em escala logarítmica.

A quantidade de casos de teste adequada aos programas foram geradas de acordo com os critérios de teste. A Tabela 11 apresenta a quantidade necessária para cada programa. O programa “mult_matrizes” apresentou uma alta quantidade de casos de teste necessário para os critérios relacionados à definição e uso computacional de variável, dessa forma implicando nos critérios todos-c-usos, todos-c-usos-grid, todos-usos e todos-usos-grid. Nenhum dos casos de teste apresentaram redundância, implicando no aumento da cobertura atingida para esses critérios. A Figura 51 mostra a relação da quantidade de casos de teste necessários para cada programa. O Programa “soma_vetores” foi o único em que apenas um caso de teste atendeu à adequação exigida. Isto ocorreu porque o *kernel do programa* não possui estruturas condicionais ou de repetição, fazendo com que todas as *threads* sempre executem todos os elementos requeridos.

Figura 46 – Custo dos elementos requeridos em escala logarítmica para cada critério de teste.

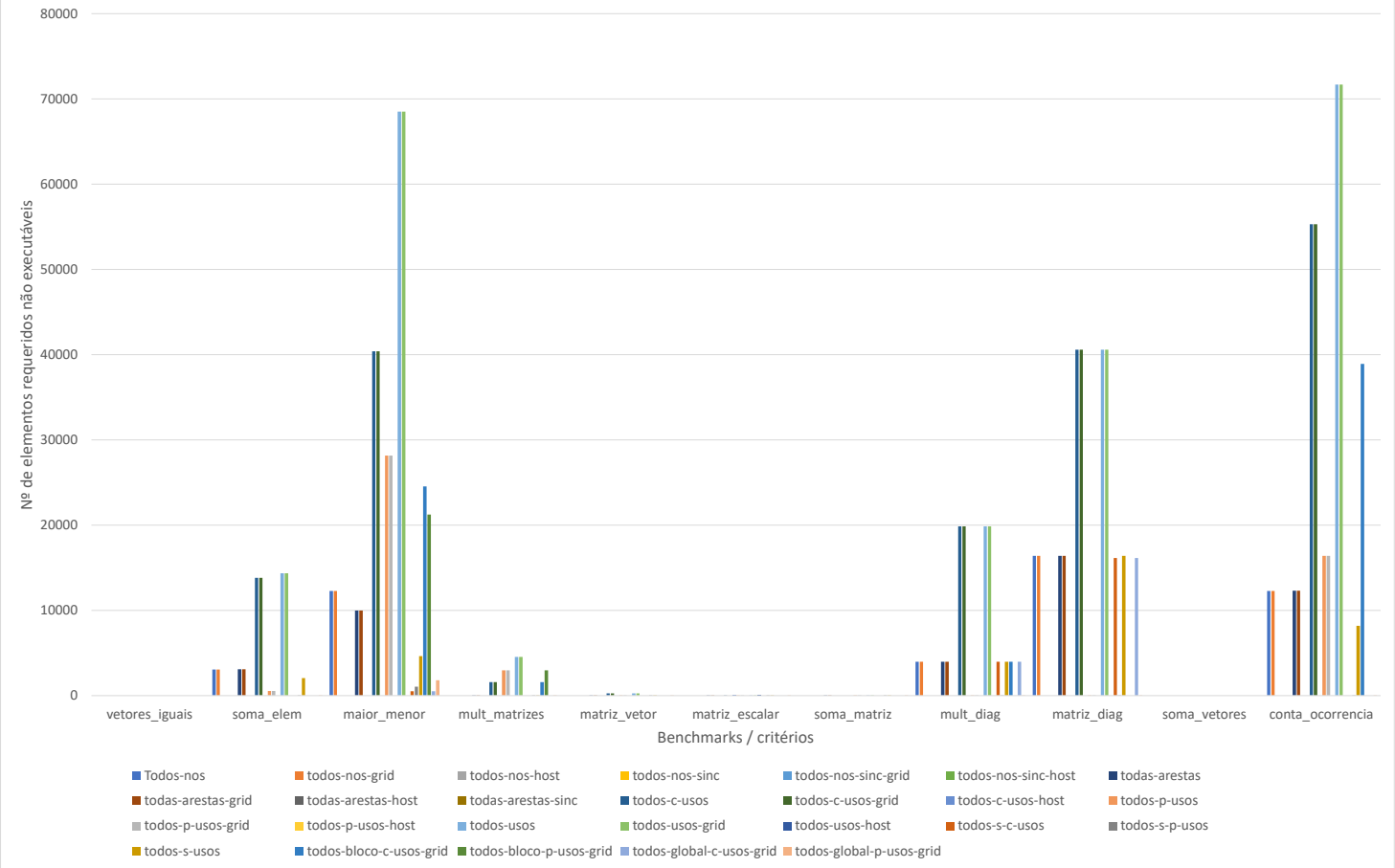


7.4.2 Resultado da eficácia dos critérios

Nós inserimos defeitos nos *benchmarks* selecionados para avaliar a métrica de eficácia, conforme sintetizado na Tabela 8. Executamos cada versão dos programas usando a ValiExec e o conjunto de casos de teste adequados para gerar os rastros de execução e a saída de cada programa. Analisamos a cobertura alcançada para cada critério e comparamos a saída resultante do programa modificado com a saída do programa original para verificar se houve erro. Baseado nisso, avaliamos para cada critério de teste se ele auxiliou na revelação do defeito. O cálculo da eficácia é feito usando o número de defeitos revelados pelo critério e a quantidade de defeitos inseridos, conforme apresentado na equação 5.1.

A Tabela 12 apresenta a eficácia alcançada para cada critério em relação aos tipos de defeitos inseridos nos *benchmarks* selecionados. Para os defeitos de erro de memória, erro de variável compartilhada, ID incorreto da *thread* e uso incorreto de variável, ao menos um critério de teste conseguiu revelar todos os defeitos inseridos nos programas. Apesar

Figura 47 – Custo dos elementos requeridos não executáveis dos critérios de teste para cada um dos programas selecionados.

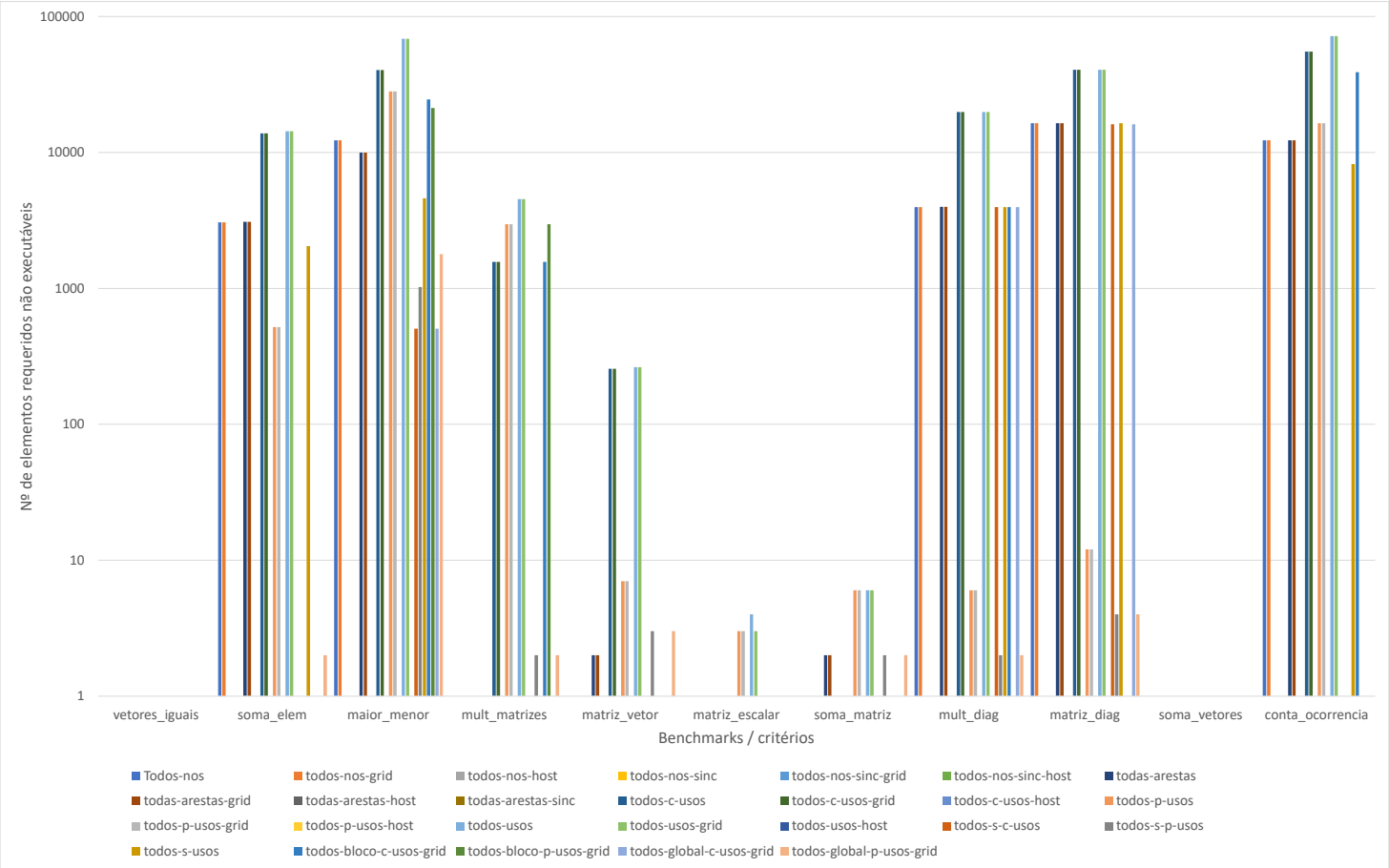


da ferramenta não permitir a execução determinística das *threads* do programa, alguns critérios conseguiram revelar parte dos defeitos de falta de sincronização, atingindo até 61,5% de eficácia. Para esse defeito, foram realizadas ao menos 10 execuções dos programas para averiguar se o defeito foi revelado.

A Figura 52 apresenta o gráfico relacionado à Tabela 12 de eficácia dos critérios. Na Figura 53 podemos observar que os critérios todos-usos, todos-usos-grid, todos-c-usos e todos-c-usos-grid atingiram a maior taxa de eficácia entre os critérios testados. A Figura 54 apresenta o gráfico *boxplot* da eficácia dos critérios de teste, onde é possível observar a média e mediana dos critérios de teste. O critério todo-usos, todos-usos-grid, todos-c-usos e todos-c-usos-grid obtiveram a maior mediana, na faixa dos 90%, indicando que foram os critérios mais eficazes entre eles.

Critérios relacionados ao *host* demonstraram baixa eficácia. Como o código do *host* possui baixa complexidade em relação ao código do *kernel*, com exceção do programa conta_ocorrencia, somente um caso de teste foi necessário para os critérios relacionados

Figura 48 – Custo dos elementos requeridos não executáveis em escala logarítmica dos critérios de teste para cada um dos programas selecionados.

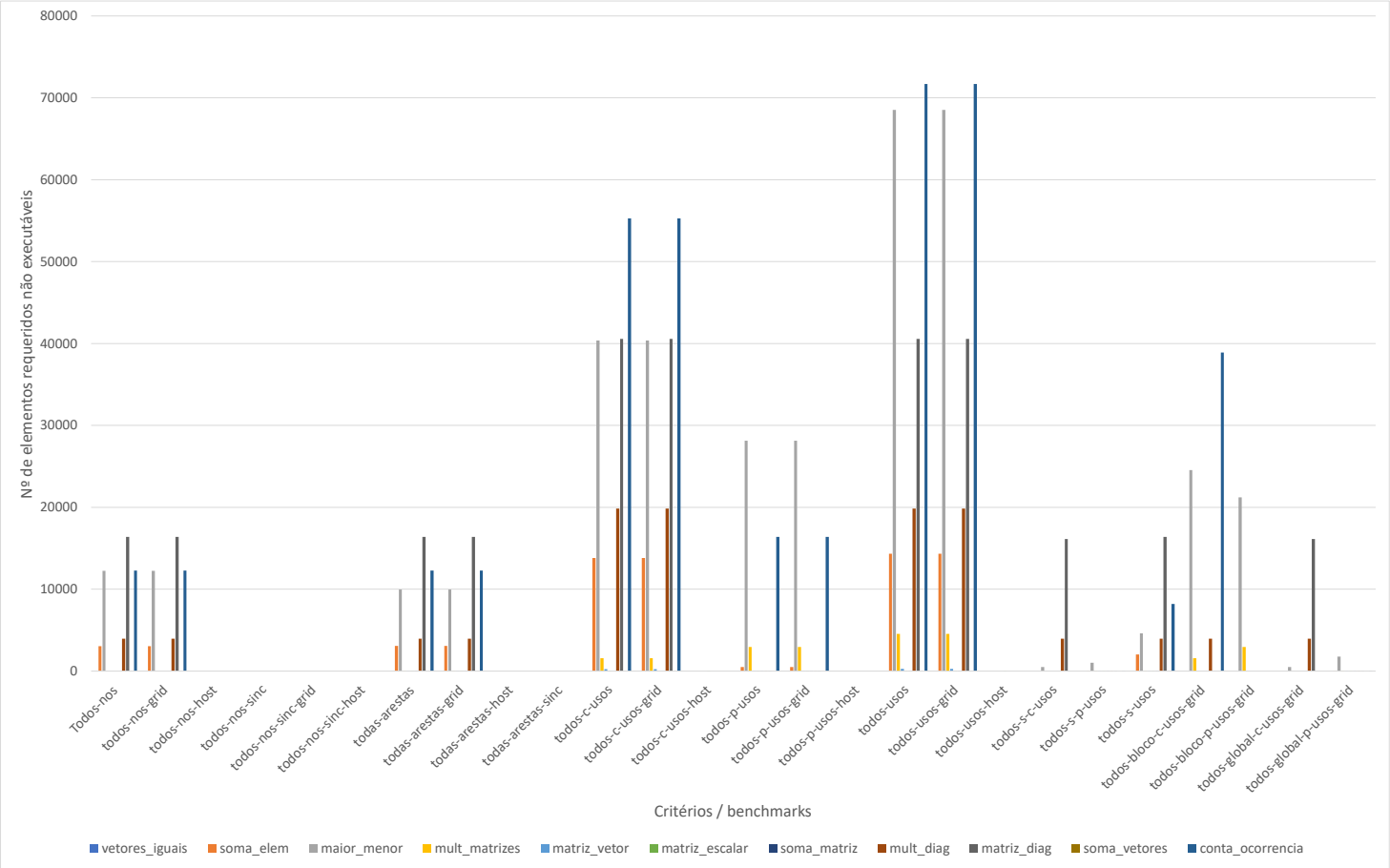


ao *host*, afetando a eficácia desses critérios.

Os critérios relacionados ao uso de variáveis em nível de bloco e em nível global também apresentaram uma eficiência inferior. No caso destes critérios, tal eficiência inferior ocorreu devido a diversos *benchmarks* não possuírem elementos requeridos relacionados a eles. Para o critério todos-bloco-p-usos-grid, apenas três *benchmarks* possuem elementos requeridos. Entretanto, ao considerar apenas os *benchmarks* que possuem os elementos requeridos para estes critérios, e comparando com o critério todos-usos, eles atingem uma eficácia mais próxima do critério todos-bloco-p-usos-grid. As Tabelas 13, 14, 15 e 16 apresentam a comparação entre os quatro critérios com o critério todos-usos. O critério todos-bloco-c-usos-grid atingiu a mesma eficácia do critério todos-usos. Todos-blocos-p-usos-grid e todos-global-p-usos-grid atingiram uma eficácia próxima.

Ao considerar a revelação de defeitos combinada da utilização dos quatro critérios relacionados ao uso de variáveis em nível de bloco e em nível global, a eficácia atingida foi aumentada. A Tabela 17 demonstra a eficácia dessa combinação em relação ao critério

Figura 49 – Custo dos elementos requeridos não executáveis para cada critério de teste.



todos-usos. Neste contexto, apesar da utilização de mais de um critério, o custo para a sua utilização é menor. Por exemplo, no programa mult_matrizes, o critério todos-usos possui 67621 elementos requeridos, enquanto a somatória dos elementos requeridos dos quatro critérios resulta em 21504 elementos, equivalente a 31,8% do custo do critério todos-usos.

7.4.3 Resultado do *strength* dos critérios

Na avaliação do *strength*, foi utilizado o conjunto de casos de teste adequado de cada critério para avaliar a cobertura alcançada nos outros critérios, para averiguar a relação de inclusão. Foi considerado os programas conta_ocorrencia, maior_menor, matriz_diagonal, matriz_escalar, matriz_vetor, mult_diag, mult_matrizes, soma_elem, soma_matriz, soma_vetores e vetores_iguais respectivamente nas Tabelas 18 a 28. Em cada tabela, a primeira coluna indica de qual critério é o conjunto de teste utilizado. Nas colunas seguintes apresenta o percentual de cobertura alcançada para cada critério. Quando não há elementos requeridos de determinado critério de teste, é inserido o caractere "-".

Tabela 11 – Número de casos de teste necessários para cada critério e programa.

Critério	vetores iguais	soma elem	maior menor	mult matrizes	matriz vetor	matriz escalar	soma matriz	mult diag	matriz diag	soma vetores	conta ocorrencia
todos-nos	1	2	2	2	1	1	2	5	4	1	2
todos-nos-grid	1	2	2	2	1	1	2	5	4	1	2
todos-nos-host	1	1	1	1	1	1	1	1	1	1	1
todos-nos-sinc	1	1	1	1	1	1	1	1	1	1	1
todos-nos-sinc-grid	1	1	1	1	1	1	1	1	1	1	1
todos-nos-sinc-host	1	1	1	1	1	1	1	1	1	1	1
todas-arestas	2	3	3	3	2	2	3	5	8	1	3
todas-arestas-grid	2	3	3	3	2	2	3	5	8	1	3
todas-arestas-host	1	1	1	1	1	1	1	1	1	1	2
todas-arestas-sinc	1	1	1	1	1	1	1	1	1	1	1
todos-c-usos	1	3	3	64	1	1	2	5	4	1	2
todos-c-usos-grid	1	3	3	64	1	1	2	5	4	-	2
todos-c-usos-host	1	1	1	1	1	1	1	1	1	1	2
todos-p-usos	2	3	4	33	2	2	3	2	8	1	3
todos-p-usos-grid	2	3	4	33	2	2	3	2	8	-	3
todos-p-usos-host	1	1	1	1	1	1	1	1	1	1	2
todos-usos	2	3	4	64	2	2	3	5	8	1	3
todos-usos-grid	2	3	4	64	2	2	3	5	8	-	3
todos-usos-host	1	1	1	1	1	1	1	1	1	1	2
todos-s-c-usos	1	1	1	2	1	1	1	5	4	-	-
todos-s-p-usos	2	3	3	2	2	2	2	2	8	-	3
todos-s-usos	1	1	1	2	2	2	2	3	8	1	1
todos-bloco-c-usos-grid	-	-	3	32	1	-	2	3	-	-	2
todos-bloco-p-usos-grid	-	-	4	33	-	-	2	-	-	-	-
todos-global-c-usos-grid	1	1	1	2	1	1	1	5	4	-	-
todos-global-p-usos-grid	2	3	2	2	2	2	2	2	8	-	3

Tabela 12 – Eficácia dos critérios.

	Defeitos revelados (%)						
Critério	**Erro de memória**	**Falta de sincronização**	**Erro de variável compartilhada**	**ID incorreto da thread**	**Uso incorreto de variável global**	**Configuração incorreta do kernel**	**Média**
Todos-nos	87,50	30,77	55,56	100,00	100,00	50,00	70,64
todos-nos-grid	87,50	30,77	55,56	100,00	100,00	50,00	70,64
todos-nos-host	25,00	7,69	11,11	66,67	57,14	33,33	33,49
todos-nos-sinc	25,00	15,38	0,00	66,67	57,14	33,33	32,92
todos-nos-sinc-grid	25,00	15,38	0,00	66,67	57,14	33,33	32,92
todos-nos-sinc-host	25,00	0,00	0,00	66,67	57,14	16,67	27,58
todas-arestas	87,50	38,46	88,89	100,00	100,00	83,33	83,03
todas-arestas-grid	87,50	38,46	88,89	100,00	100,00	83,33	83,03
todas-arestas-host	25,00	7,69	11,11	66,67	57,14	33,33	33,49
todas-arestas-sinc	25,00	23,08	11,11	66,67	57,14	50,00	38,83
todos-c-usos	100,00	61,54	100,00	100,00	100,00	83,33	90,81
todos-c-usos-grid	100,00	61,54	100,00	100,00	100,00	83,33	90,81
todos-c-usos-host	25,00	7,69	11,11	58,33	42,86	16,67	26,94
todos-p-usos	87,50	61,54	88,89	100,00	100,00	83,33	86,88
todos-p-usos-grid	87,50	61,54	88,89	100,00	100,00	83,33	86,88
todos-p-usos-host	25,00	7,69	11,11	58,33	42,86	16,67	26,94
todos-usos	100,00	61,54	100,00	100,00	100,00	83,33	90,81
todos-usos-grid	100,00	61,54	100,00	100,00	100,00	83,33	90,81
todos-usos-host	25,00	7,69	11,11	58,33	42,86	16,67	26,94
todos-s-c-usos	87,50	30,77	44,44	100,00	85,71	50,00	66,40
todos-s-p-usos	87,50	53,85	66,67	91,67	85,71	66,67	75,34
todos-s-usos	75,00	30,77	55,56	83,33	85,71	50,00	69,94
todos-bloco-c-usos-grid	37,50	46,15	100,00	58,33	71,43	66,67	56,80
todos-bloco-p-usos-grid	25,00	23,08	66,67	41,67	28,57	33,33	36,39
todos-global-c-usos-grid	87,50	30,77	44,44	91,67	71,43	33,33	59,86
todos-global-p-usos-grid	87,50	53,85	66,67	91,67	85,71	66,67	75,34

Figura 50 – Custo dos elementos requeridos não executáveis em escala logarítmica para cada critério de teste.

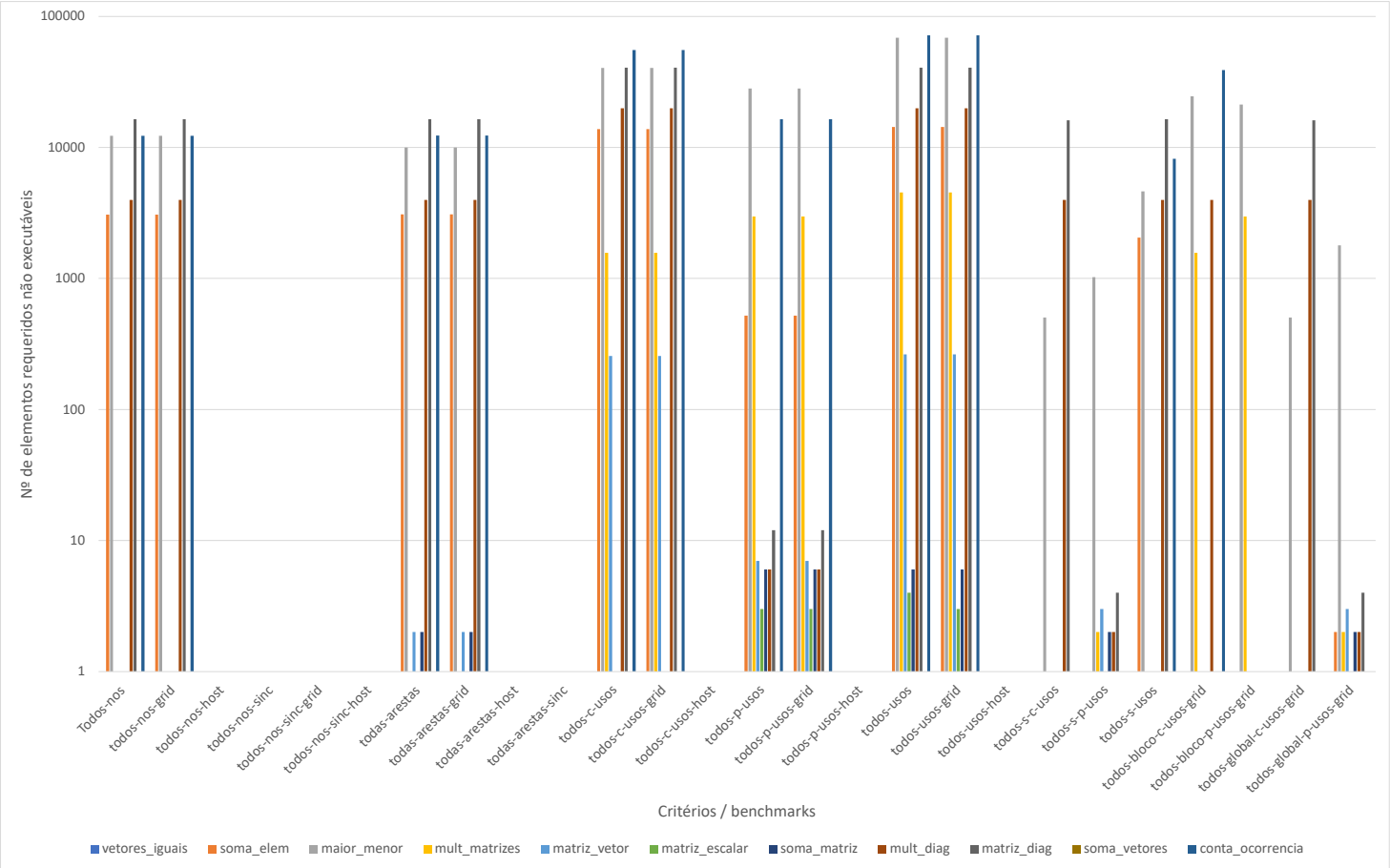


Tabela 13 – Eficácia do critério todos-blocos-c-usos-grid.

Critério	Defeitos revelados (%)						
	Erro de memória	Falta de sincronização	Erro de variável compartilhada	ID incorreto da thread	Uso incorreto de variável global	Configuração incorreta do kernel	Média
todos-usos	100,00	66,67	100,00	100,00	100,00	100,00	94,44
todos-bloco-c-usos-grid	100,00	66,67	100,00	100,00	100,00	100,00	94,44

Tabela 14 – Eficácia do critério todos-blocos-p-usos-grid.

Critério	Defeitos revelados (%)						
	Erro de memória	Falta de sincronização	Erro de variável compartilhada	ID incorreto da thread	Uso incorreto de variável global	Configuração incorreta do kernel	Média
todos-usos	100,00	66,67	100,00	100,00	100,00	100,00	94,44
todos-bloco-p-usos-grid	66,67	50,00	85,71	100,00	100,00	100,00	83,73

Tabela 15 – Eficácia do critério todos-global-c-usos-grid.

Critério	Defeitos revelados (%)						
	Erro de memória	Falta de sincronização	Erro de variável compartilhada	ID incorreto da thread	Uso incorreto de variável global	Configuração incorreta do kernel	Média
todos-usos	100,00	54,55	100,00	100,00	100,00	75,00	88,26
todos-global-c-usos-grid	87,50	36,36	50,00	100,00	83,33	50,00	67,87

Figura 51 – Número de casos de teste adequados para cada critério.

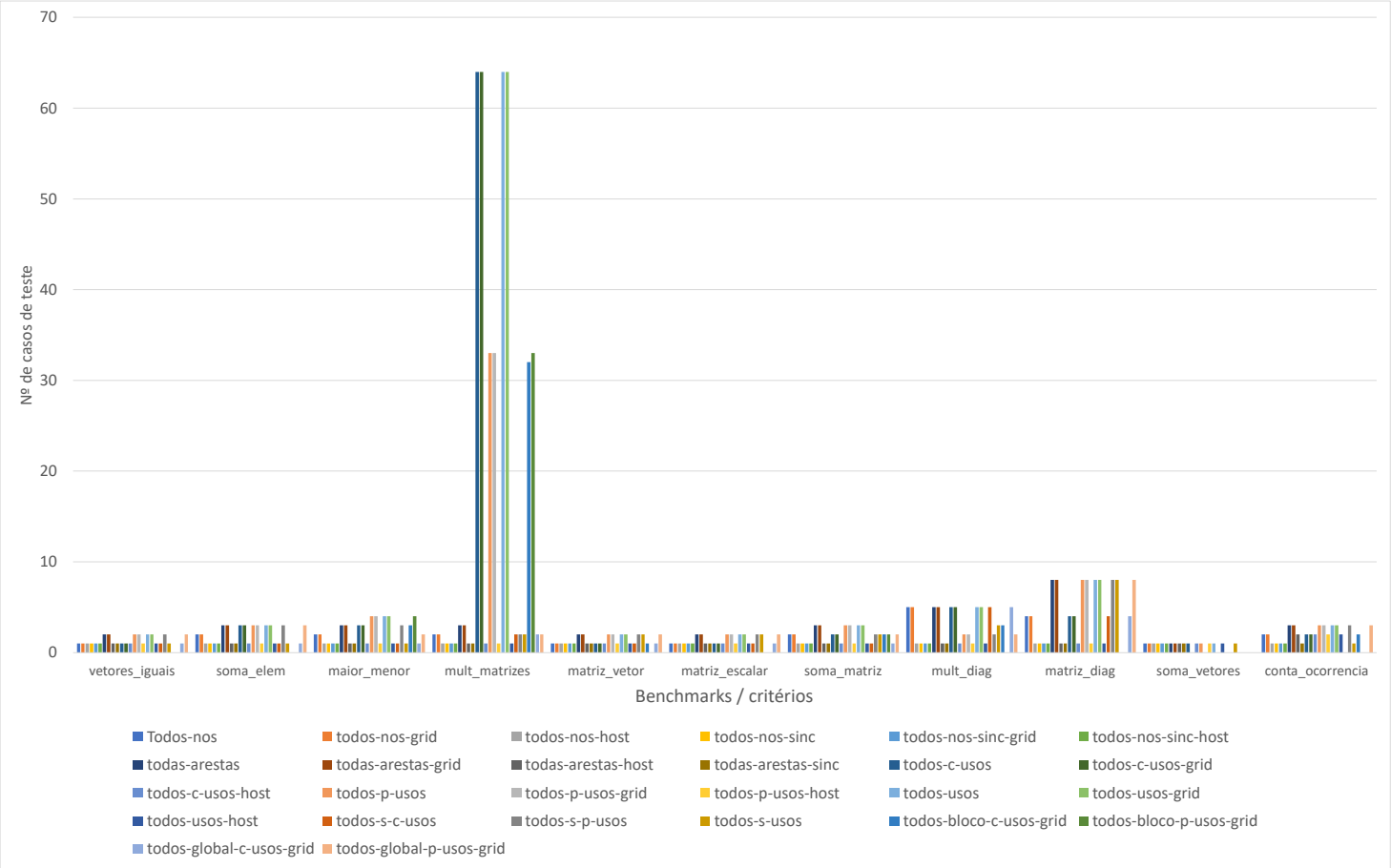


Tabela 16 – Eficácia do critério todos-global-p-usos-grid.

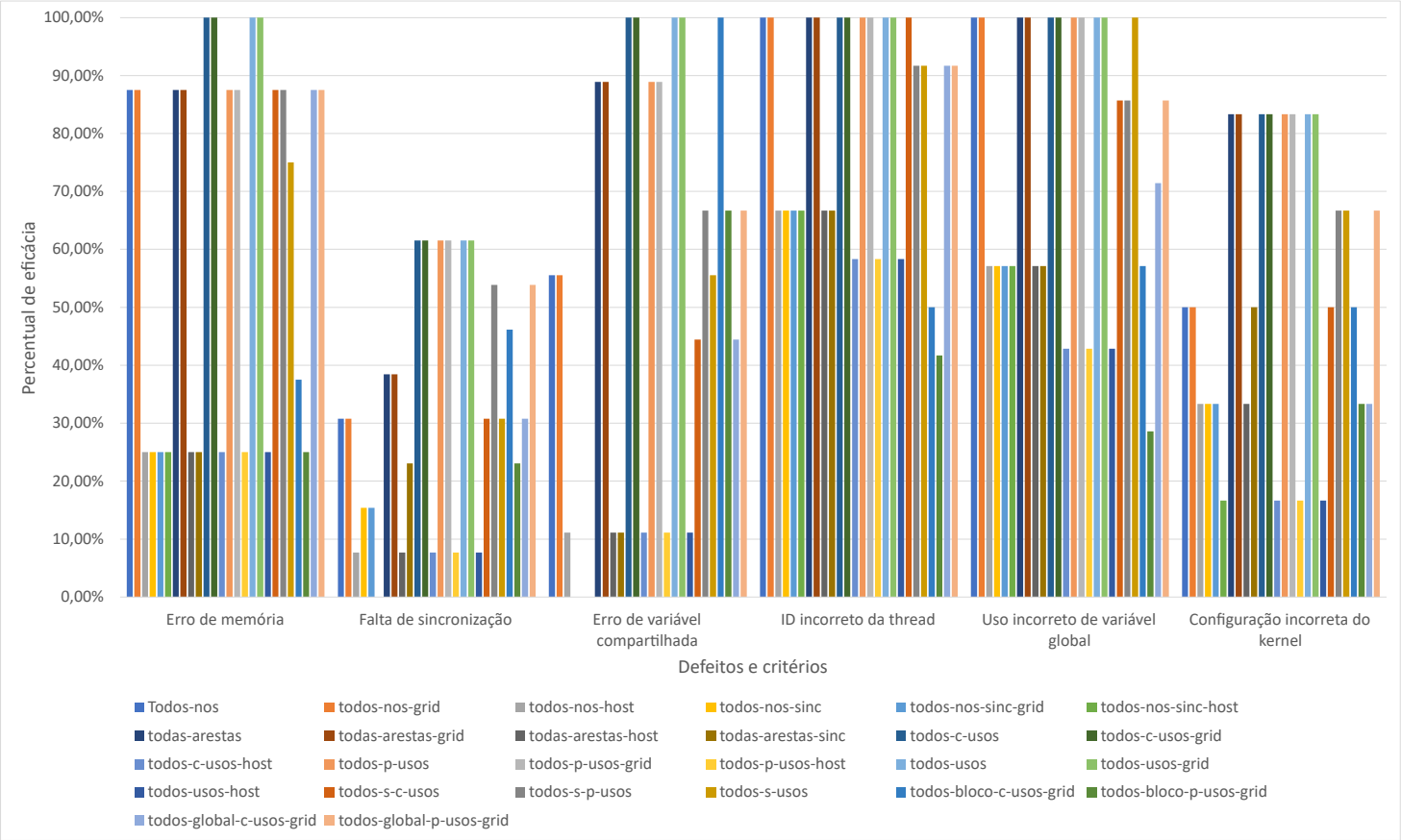
	Defeitos revelados (%)						
Critério	**Erro de memória**	**Falta de sincronização**	**Erro de variável compartilhada**	**ID incorreto da thread**	**Uso incorreto de variável global**	**Configuração incorreta do kernel**	**Média**
todos-usos	100,00	61,54	100,00	100,00	100,00	80,00	90,26
todos-global-p-usos-grid	87,50	53,85	66,67	100,00	100,00	80,00	81,34

Tabela 17 – Eficácia da combinação dos critérios todos-bloco-c-usos-grid, todos-bloco-p-usos-grid, todos-global-c-usos-grid e todos-global-p-usos-grid.

	Defeitos revelados (%)						
Critério	**Erro de memória**	**Falta de sincronização**	**Erro de variável compartilhada**	**ID incorreto da thread**	**Uso incorreto de variável global**	**Configuração incorreta do kernel**	**Média**
todos-usos	100,00	61,54	100,00	100,00	100,00	80,00	90,26
bloco-global	100,00	61,54	100,00	100,00	100,00	80,00	90,26

Os critérios que são satisfeitos com apenas um caso de teste acabam atingindo 100% de cobertura para o conjunto de casos de teste de qualquer critério. Isso pode ser observado para os critérios todos-nos-host e todos-nos-sinc. Para os critérios todos-nos-grid-sinc e todos-nos-sinc o mesmo ocorre, com exceção do programa soma_elem, devido a um comando *return* no código do *kernel* que implica na finalização antecipada do *kernel* para

Figura 52 – Eficácia dos critérios de teste para cada um dos defeitos.



determinadas *threads*. Essa característica também afeta o critério todas-arestas-sinc.

Critérios que necessitam de um conjunto com mais casos de teste alcançam uma maior taxa de cobertura quando aplicado aos outros critérios. Assim, os critérios todos-usos e todos-usos-grid obtiveram a maior taxa de cobertura nos outros critérios, quando o seu conjunto de casos de teste adequado foi aplicado em cada um dos programas.

Alguns critérios obtiveram taxas de cobertura iguais. Alguns exemplos são os critérios todos-nos e todos-nos-grid, todas-arestas e todas-arestas-grid, todos-usos e todos-usos-grid, dentre outros. Isso é devido aos critérios relacionados a *grid* serem um subconjunto dos critérios mais gerais. Os critérios gerais incluem os elementos requeridos relacionados ao *host* e *grid* e, em alguns casos, incluem elementos relacionados a sincronização. Como os critérios relacionados ao *grid* possuem um número maior de elementos requeridos e requerem mais casos de teste, a diferença no número de elementos requeridos entre os critérios gerais e os critérios do *grid* acaba sendo mínima.

Figura 53 – Eficácia dos critérios de teste.

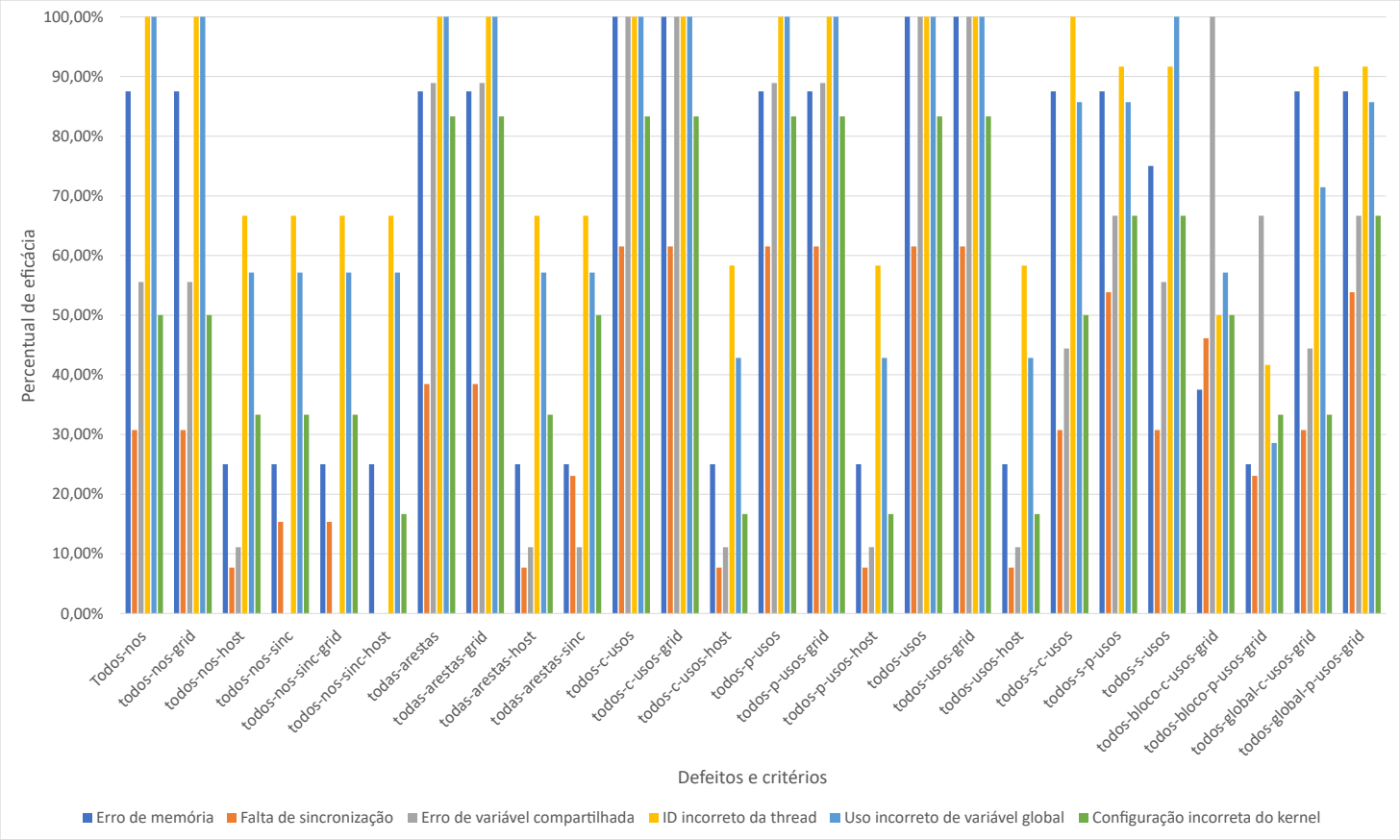


Figura 54 – *Boxplot* da eficácia dos critérios de teste.

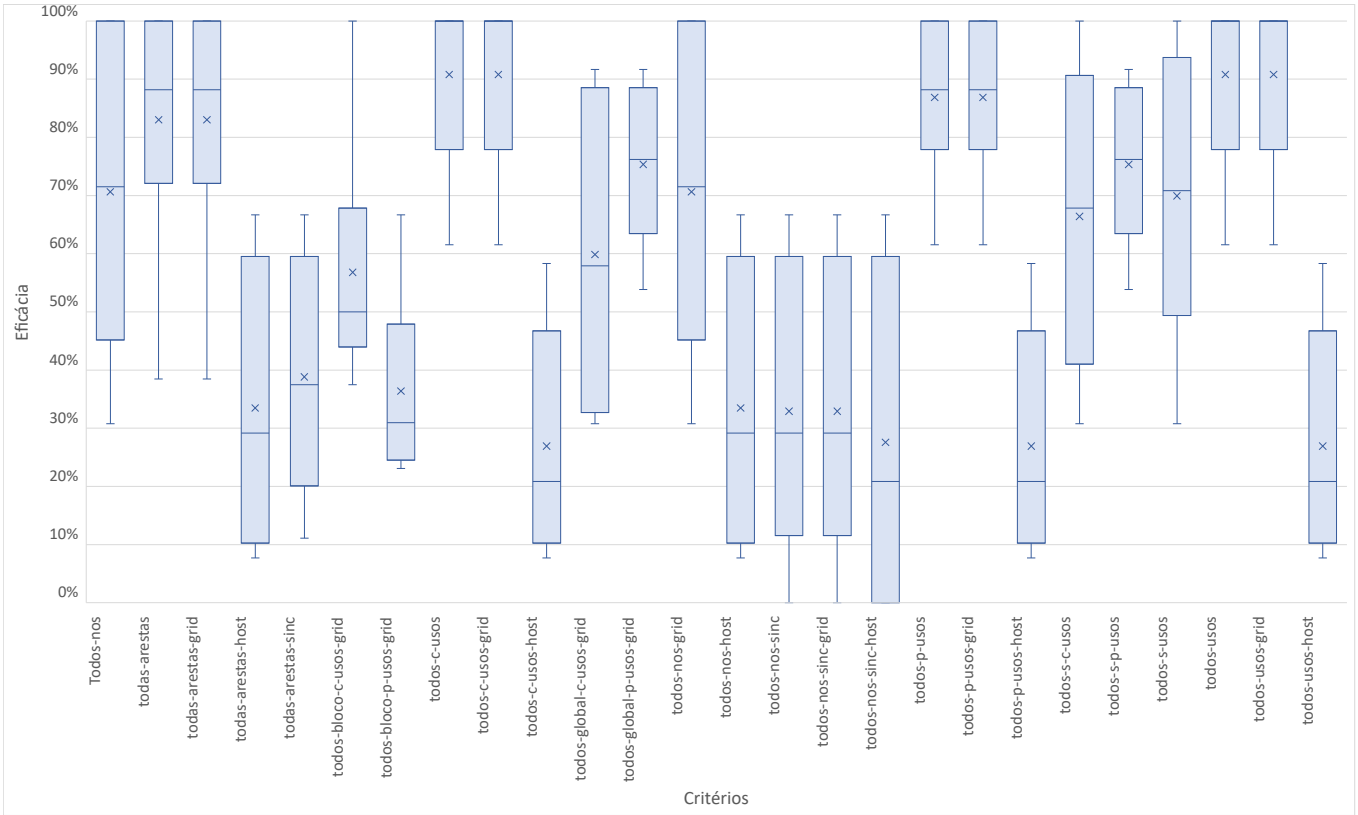


Tabela 18 – Strength dos critérios para o programa conta_ocorrencia.

Strength (%)	TN	TNG	TNH	TNS	TNSG	TNSH	TA	TAG	TAH	TAS	TCU	TCUG	TCUH
TN	91,2	91,2	100,0	100,0	100,0	100,0	61,1	50,0	75,0	100,0	44,2	44,2	92,8
TNG	91,2	91,2	100,0	100,0	100,0	100,0	61,1	50,0	75,0	100,0	44,2	44,2	92,8
TNH	77,9	77,9	100,0	100,0	100,0	100,0	61,1	50,0	75,0	100,0	32,7	32,7	92,8
TNS	73,5	73,5	75,0	100,0	100,0	100,0	61,1	50,0	50,0	100,0	28,9	28,9	64,3
TNSG	73,5	73,5	75,0	100,0	100,0	100,0	61,1	50,0	50,0	100,0	28,9	28,9	64,3
TNSH	73,5	73,5	75,0	100,0	100,0	100,0	61,1	50,0	50,0	100,0	28,9	28,9	64,3
TA	91,2	91,2	100,0	100,0	100,0	100,0	83,3	78,5	100,0	100,0	48,1	48,1	100,0
TAG	91,2	91,2	100,0	100,0	100,0	100,0	83,3	78,5	100,0	100,0	48,1	48,1	100,0
TAH	77,9	77,9	100,0	100,0	100,0	100,0	63,9	53,5	100,0	100,0	33,7	33,7	100,0
TAS	73,5	73,5	75,0	100,0	100,0	100,0	61,1	50,0	50,0	100,0	28,9	28,9	64,3
TCU	91,2	91,2	100,0	100,0	100,0	100,0	72,2	64,3	75,0	100,0	48,1	48,1	92,8
TCUG	91,2	91,2	100,0	100,0	100,0	100,0	72,2	64,3	75,0	100,0	48,1	48,1	92,8
TCUH	77,9	77,9	100,0	100,0	100,0	100,0	63,9	53,5	100,0	100,0	33,7	33,7	100,0
TPU	91,2	91,2	100,0	100,0	100,0	100,0	83,3	78,5	100,0	100,0	48,1	48,1	100,0
TPUG	91,2	91,2	100,0	100,0	100,0	100,0	83,3	78,5	100,0	100,0	48,1	48,1	100,0
TPUH	77,9	77,9	100,0	100,0	100,0	100,0	63,9	53,5	100,0	100,0	33,7	33,7	100,0
TU	91,2	91,2	100,0	100,0	100,0	100,0	83,3	78,5	100,0	100,0	48,1	48,1	100,0
TUG	91,2	91,2	100,0	100,0	100,0	100,0	83,3	78,5	100,0	100,0	48,1	48,1	100,0
TUH	77,9	77,9	100,0	100,0	100,0	100,0	63,9	53,5	100,0	100,0	33,7	33,7	100,0
TSCU	-	-	-	-	-	-	-	-	-	-	-	-	-
TSPU	91,2	91,2	100,0	100,0	100,0	100,0	83,3	78,5	100,0	100,0	48,1	48,1	100,0
TSU	77,9	77,9	100,0	100,0	100,0	100,0	61,1	50,0	75,0	100,0	32,7	32,7	92,8
TBCUG	91,2	91,2	100,0	100,0	100,0	100,0	72,2	64,3	75,0	100,0	48,1	48,1	92,8
TBPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGCUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGPUG	91,2	91,2	100,0	100,0	100,0	100,0	83,3	78,5	100,0	100,0	48,1	48,1	100,0

Strength	TPU	TPUG	TPUH	TU	TUG	TUH	TSCU	TSPU	TSU	TBCUG	TBPUG	TGCUG	TGPUG
TN	62,5	62,5	90,0	54,3	54,3	91,7	-	50,0	83,3	12,5	-	-	50,0
TNG	62,5	62,5	90,0	54,3	54,3	91,7	-	50,0	83,3	12,5	-	-	50,0
TNH	53,1	53,1	90,0	44,0	44,0	91,7	-	25,0	83,3	12,5	-	-	25,0
TNS	50,0	50,0	50,0	40,5	40,5	58,3	-	16,7	83,3	12,5	-	-	16,7
TNSG	50,0	50,0	50,0	40,5	40,5	58,3	-	16,7	83,3	12,5	-	-	16,7
TNSH	50,0	50,0	50,0	40,5	40,5	58,3	-	16,7	83,3	12,5	-	-	16,7
TA	87,5	87,5	100,0	69,8	69,8	100,0	-	100,0	83,3	20,8	-	-	100,0
TAG	87,5	87,5	100,0	69,8	69,8	100,0	-	100,0	83,3	20,8	-	-	100,0
TAH	56,2	56,2	100,0	46,1	46,1	100,0	-	29,2	83,3	14,6	-	-	29,2
TAS	50,0	50,0	50,0	40,5	40,5	58,3	-	16,7	83,3	12,5	-	-	16,7
TCU	75,0	75,0	90,0	62,9	62,9	91,7	-	83,3	83,3	20,8	-	-	83,3
TCUG	75,0	75,0	90,0	62,9	62,9	91,7	-	83,3	83,3	20,8	-	-	83,3
TCUH	56,2	56,2	100,0	46,1	46,1	100,0	-	29,2	83,3	14,6	-	-	29,2
TPU	87,5	87,5	100,0	69,8	69,8	100,0	-	100,0	83,3	20,8	-	-	100,0
TPUG	87,5	87,5	100,0	69,8	69,8	100,0	-	100,0	83,3	20,8	-	-	100,0
TPUH	56,2	56,2	100,0	46,1	46,1	100,0	-	29,2	83,3	14,6	-	-	29,2
TU	87,5	87,5	100,0	69,8	69,8	100,0	-	100,0	83,3	20,8	-	-	100,0
TUG	87,5	87,5	100,0	69,8	69,8	100,0	-	100,0	83,3	20,8	-	-	100,0
TUH	56,2	56,2	100,0	46,1	46,1	100,0	-	29,2	83,3	14,6	-	-	29,2
TSCU	-	-	-	-	-	-	-	-	-	-	-	-	-
TSPU	87,5	87,5	100,0	69,8	69,8	100,0	-	100,0	83,3	20,8	-	-	100,0
TSU	53,1	53,1	90,0	44,0	44,0	91,7	-	25,0	83,3	12,5	-	-	25,0
TBCUG	75,0	75,0	90,0	62,9	62,9	91,7	-	83,3	83,3	20,8	-	-	83,3
TBPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGCUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGPUG	87,5	87,5	100,0	69,8	69,8	100,0	-	100,0	83,3	20,8	-	-	100,0

Tabela 19 – Strength dos critérios para o programa maior_menor.

Strength (%)	TN	TNG	TNH	TNS	TNSG	TNSH	TA	TAG	TAH	TAS	TCU	TCUG	TCUH
TN	84,3	84,3	100,0	100,0	100,0	100,0	66,0	56,3	100,0	100,0	51,3	51,3	100,0
TNG	84,3	84,3	100,0	100,0	100,0	100,0	66,0	56,3	100,0	100,0	51,3	51,3	100,0
TNH	73,9	73,9	100,0	100,0	100,0	100,0	58,6	46,8	100,0	100,0	25,4	23,3	100,0
TNS	73,9	73,9	100,0	100,0	100,0	100,0	58,6	46,8	100,0	100,0	25,4	23,3	100,0
TNSG	73,9	73,9	100,0	100,0	100,0	100,0	58,6	46,8	100,0	100,0	25,4	23,3	100,0
TNSH	73,9	73,9	100,0	100,0	100,0	100,0	58,6	46,8	100,0	100,0	25,4	23,3	100,0
TA	84,3	84,3	100,0	100,0	100,0	100,0	75,9	69,1	100,0	100,0	51,3	51,3	100,0
TAG	84,3	84,3	100,0	100,0	100,0	100,0	75,9	69,1	100,0	100,0	51,3	51,3	100,0
TAH	73,9	73,9	100,0	100,0	100,0	100,0	58,6	46,8	100,0	100,0	25,4	23,3	100,0
TAS	73,9	73,9	100,0	100,0	100,0	100,0	58,6	46,8	100,0	100,0	25,4	23,3	100,0
TCU	84,3	84,3	100,0	100,0	100,0	100,0	75,9	69,1	100,0	100,0	51,3	51,3	100,0
TCUG	84,3	84,3	100,0	100,0	100,0	100,0	75,9	69,1	100,0	100,0	51,3	51,3	100,0
TCUH	73,9	73,9	100,0	100,0	100,0	100,0	58,6	46,8	100,0	100,0	25,4	23,3	100,0
TPU	84,3	84,3	100,0	100,0	100,0	100,0	75,9	69,1	100,0	100,0	51,3	51,3	100,0
TPUG	84,3	84,3	100,0	100,0	100,0	100,0	75,9	69,1	100,0	100,0	51,3	51,3	100,0
TPUH	73,9	73,9	100,0	100,0	100,0	100,0	58,6	46,8	100,0	100,0	25,4	23,3	100,0
TU	84,3	84,3	100,0	100,0	100,0	100,0	75,9	69,1	100,0	100,0	51,3	51,3	100,0
TUG	84,3	84,3	100,0	100,0	100,0	100,0	75,9	69,1	100,0	100,0	51,3	51,3	100,0
TUH	73,9	73,9	100,0	100,0	100,0	100,0	58,6	46,8	100,0	100,0	25,4	23,3	100,0
TSCU	83,0	83,0	100,0	100,0	100,0	100,0	61,1	50,0	100,0	100,0	47,0	47,0	100,0
TSPU	83,0	83,0	100,0	100,0	100,0	100,0	73,5	65,9	100,0	100,0	47,0	47,0	100,0
TSU	83,0	83,0	100,0	100,0	100,0	100,0	73,5	65,9	100,0	100,0	47,0	47,0	100,0
TBCUG	84,3	84,3	100,0	100,0	100,0	100,0	75,9	69,1	100,0	100,0	51,3	51,3	100,0
TBPUG	84,3	84,3	100,0	100,0	100,0	100,0	75,9	69,1	100,0	100,0	51,3	51,3	100,0
TGCUG	83,0	83,0	100,0	100,0	100,0	100,0	61,1	50,0	100,0	100,0	47,0	47,0	100,0
TGPUG	83,0	83,0	100,0	100,0	100,0	100,0	73,5	65,9	100,0	100,0	47,0	47,0	100,0

Strength	TPU	TPUG	TPUH	TU	TUG	TUH	TSCU	TSPU	TSU	TBCUG	TBPUG	TGCUG	TGPUG
TN	52,3	52,3	100,0	51,8	51,8	100,0	89,1	62,5	82,7	11,2	19,5	89,1	57,4
TNG	52,3	52,3	100,0	51,8	51,8	100,0	89,1	62,5	82,7	11,2	19,5	89,1	57,4
TNH	43,1	43,1	100,0	34,0	34,0	100,0	0,1	50,0	82,7	0,0	4,6	0,1	50,0
TNS	43,1	43,1	100,0	34,0	34,0	100,0	0,1	50,0	82,7	0,0	4,6	0,1	50,0
TNSG	43,1	43,1	100,0	34,0	34,0	100,0	0,1	50,0	82,7	0,0	4,6	0,1	50,0
TNSH	43,1	43,1	100,0	34,0	34,0	100,0	0,1	50,0	82,7	0,0	4,6	0,1	50,0
TA	64,1	64,1	100,0	57,5	57,5	100,0	89,1	87,5	82,7	11,2	23,2	89,1	87,1
TAG	64,1	64,1	100,0	57,5	57,5	100,0	89,1	87,5	82,7	11,2	23,2	89,1	87,1
TAH	43,1	43,1	100,0	34,0	34,0	100,0	0,1	50,0	82,7	0,0	4,6	0,1	50,0
TAS	43,1	43,1	100,0	34,0	34,0	100,0	0,1	50,0	82,7	0,0	4,6	0,1	50,0
TCU	64,1	64,1	100,0	57,5	57,5	100,0	89,1	87,5	82,7	11,2	23,2	89,1	87,1
TCUG	64,1	64,1	100,0	57,5	57,5	100,0	89,1	87,5	82,7	11,2	23,2	89,1	87,1
TCUH	43,1	43,1	100,0	34,0	34,0	100,0	0,1	50,0	82,7	0,0	4,6	0,1	50,0
TPU	64,1	64,1	100,0	57,5	57,5	100,0	89,1	87,5	82,7	11,2	23,2	89,1	87,1
TPUG	64,1	64,1	100,0	57,5	57,5	100,0	89,1	87,5	82,7	11,2	23,2	89,1	87,1
TPUH	43,1	43,1	100,0	34,0	34,0	100,0	0,1	50,0	82,7	0,0	4,6	0,1	50,0
TU	64,1	64,1	100,0	57,5	57,5	100,0	89,1	87,5	82,7	11,2	23,2	89,1	87,1
TUG	64,1	64,1	100,0	57,5	57,5	100,0	89,1	87,5	82,7	11,2	23,2	89,1	87,1
TUH	43,1	43,1	100,0	34,0	34,0	100,0	0,1	50,0	82,7	0,0	4,6	0,1	50,0
TSCU	49,0	49,0	100,0	48,0	48,0	100,0	89,1	62,5	82,7	5,6	10,2	89,1	57,4
TSPU	60,8	60,8	100,0	53,7	53,7	100,0	89,1	87,5	82,7	5,6	13,9	89,1	87,1
TSU	60,8	60,8	100,0	53,7	53,7	100,0	89,1	87,5	82,7	5,6	13,9	89,1	87,1
TBCUG	64,1	64,1	100,0	57,5	57,5	100,0	89,1	87,5	82,7	11,2	23,2	89,1	87,1
TBPUG	64,1	64,1	100,0	57,5	57,5	100,0	89,1	87,5	82,7	11,2	23,2	89,1	87,1
TGCUG	49,0	49,0	100,0	48,0	48,0	100,0	89,1	62,5	82,7	5,6	10,2	89,1	57,4
TGPUG	60,8	60,8	100,0	53,7	53,7	100,0	89,1	87,5	82,7	5,6	13,9	89,1	87,1

Tabela 20 – Strength dos critérios para o programa matriz_diag.

Strength (%)	TN	TNG	TNH	TNS	TNSG	TNSH	TA	TAG	TAH	TAS	TCU	TCUG	TCUH
TN	88,6	88,6	100,0	100,0	100,0	100,0	55,5	50,0	100,0	100,0	70,0	70,0	100,0
TNG	88,6	88,6	100,0	100,0	100,0	100,0	55,5	50,0	100,0	100,0	70,0	70,0	100,0
TNH	31,4	31,4	100,0	100,0	100,0	100,0	33,3	25,0	100,0	100,0	12,1	12,1	100,0
TNS	31,4	31,4	100,0	100,0	100,0	100,0	33,3	25,0	100,0	100,0	12,1	12,1	100,0
TNSG	31,4	31,4	100,0	100,0	100,0	100,0	33,3	25,0	100,0	100,0	12,1	12,1	100,0
TNSH	31,4	31,4	100,0	100,0	100,0	100,0	33,3	25,0	100,0	100,0	12,1	12,1	100,0
TA	88,6	88,6	100,0	100,0	100,0	100,0	77,8	75,0	100,0	100,0	70,0	70,0	100,0
TAG	88,6	88,6	100,0	100,0	100,0	100,0	77,8	75,0	100,0	100,0	70,0	70,0	100,0
TAH	31,4	31,4	100,0	100,0	100,0	100,0	33,3	25,0	100,0	100,0	12,1	12,1	100,0
TAS	31,4	31,4	100,0	100,0	100,0	100,0	33,3	25,0	100,0	100,0	12,1	12,1	100,0
TCU	88,6	88,6	100,0	100,0	100,0	100,0	55,5	50,0	100,0	100,0	70,0	70,0	100,0
TCUG	88,6	88,6	100,0	100,0	100,0	100,0	55,5	50,0	100,0	100,0	70,0	70,0	100,0
TCUH	31,4	31,4	100,0	100,0	100,0	100,0	33,3	25,0	100,0	100,0	12,1	12,1	100,0
TPU	88,6	88,6	100,0	100,0	100,0	100,0	77,8	75,0	100,0	100,0	70,0	70,0	100,0
TPUG	88,6	88,6	100,0	100,0	100,0	100,0	77,8	75,0	100,0	100,0	70,0	70,0	100,0
TPUH	31,4	31,4	100,0	100,0	100,0	100,0	33,3	25,0	100,0	100,0	12,1	12,1	100,0
TU	88,6	88,6	100,0	100,0	100,0	100,0	77,8	75,0	100,0	100,0	70,0	70,0	100,0
TUG	88,6	88,6	100,0	100,0	100,0	100,0	77,8	75,0	100,0	100,0	70,0	70,0	100,0
TUH	31,4	31,4	100,0	100,0	100,0	100,0	33,3	25,0	100,0	100,0	12,1	12,1	100,0
TSCU	88,6	88,6	100,0	100,0	100,0	100,0	55,5	50,0	100,0	100,0	70,0	70,0	100,0
TSPU	88,6	88,6	100,0	100,0	100,0	100,0	77,8	75,0	100,0	100,0	70,0	70,0	100,0
TSU	88,6	88,6	100,0	100,0	100,0	100,0	77,8	75,0	100,0	100,0	70,0	70,0	100,0
TBCUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TBPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGCUG	88,6	88,6	100,0	100,0	100,0	100,0	77,8	75,0	100,0	100,0	70,0	70,0	100,0
TGPUG	88,6	88,6	100,0	100,0	100,0	100,0	77,8	75,0	100,0	100,0	70,0	70,0	100,0

Strength	TPU	TPUG	TPUH	TU	TUG	TUH	TSCU	TSPU	TSU	TBCUG	TBPUG	TGCUG	TGPUG
TN	50,0	50,0	100,0	61,6	61,6	100,0	67,2	50,0	61,5	-	-	67,2	50,0
TNG	50,0	50,0	100,0	61,6	61,6	100,0	67,2	50,0	61,5	-	-	67,2	50,0
TNH	12,5	12,5	100,0	12,3	12,3	100,0	33,3	12,5	38,5	-	-	33,3	12,5
TNS	12,5	12,5	100,0	12,3	12,3	100,0	33,3	12,5	38,5	-	-	33,3	12,5
TNSG	12,5	12,5	100,0	12,3	12,3	100,0	33,3	12,5	38,5	-	-	33,3	12,5
TNSH	12,5	12,5	100,0	12,3	12,3	100,0	33,3	12,5	38,5	-	-	33,3	12,5
TA	100,0	100,0	100,0	82,6	82,6	100,0	67,2	100,0	69,2	-	-	67,2	100,0
TAG	100,0	100,0	100,0	82,6	82,6	100,0	67,2	100,0	69,2	-	-	67,2	100,0
TAH	12,5	12,5	100,0	12,3	12,3	100,0	33,3	12,5	38,5	-	-	33,3	12,5
TAS	12,5	12,5	100,0	12,3	12,3	100,0	33,3	12,5	38,5	-	-	33,3	12,5
TCU	50,0	50,0	100,0	61,6	61,6	100,0	67,2	50,0	61,5	-	-	67,2	50,0
TCUG	50,0	50,0	100,0	61,6	61,6	100,0	67,2	50,0	61,5	-	-	67,2	50,0
TCUH	12,5	12,5	100,0	12,3	12,3	100,0	33,3	12,5	38,5	-	-	33,3	12,5
TPU	100,0	100,0	100,0	82,6	82,6	100,0	67,2	100,0	69,2	-	-	67,2	100,0
TPUG	100,0	100,0	100,0	82,6	82,6	100,0	67,2	100,0	69,2	-	-	67,2	100,0
TPUH	12,5	12,5	100,0	12,3	12,3	100,0	33,3	12,5	38,5	-	-	33,3	12,5
TU	100,0	100,0	100,0	82,6	82,6	100,0	67,2	100,0	69,2	-	-	67,2	100,0
TUG	100,0	100,0	100,0	82,6	82,6	100,0	67,2	100,0	69,2	-	-	67,2	100,0
TUH	12,5	12,5	100,0	12,3	12,3	100,0	33,3	12,5	38,5	-	-	33,3	12,5
TSCU	50,0	50,0	100,0	61,6	61,6	100,0	67,2	50,0	61,5	-	-	67,2	50,0
TSPU	100,0	100,0	100,0	82,6	82,6	100,0	67,2	100,0	69,2	-	-	67,2	100,0
TSU	100,0	100,0	100,0	82,6	82,6	100,0	67,2	100,0	69,2	-	-	67,2	100,0
TBCUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TBPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGCUG	100,0	100,0	100,0	82,6	82,6	100,0	67,2	100,0	69,2	-	-	67,2	100,0
TGPUG	100,0	100,0	100,0	82,6	82,6	100,0	67,2	100,0	69,2	-	-	67,2	100,0

Tabela 21 – Strength dos critérios para o programa matriz_escalar.

Strength (%)	TN	TNG	TNH	TNS	TNSG	TNSH	TA	TAG	TAH	TAS	TCU	TCUG	TCUH
TN	100,0	100,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	100,0	100,0	94,4
TNG	100,0	100,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	100,0	100,0	94,4
TNH	75,0	75,0	100,0	100,0	100,0	100,0	75,0	40,0	100,0	100,0	0,2	0,0	94,4
TNS	75,0	75,0	100,0	100,0	100,0	100,0	75,0	40,0	100,0	100,0	0,2	0,0	94,4
TNSG	75,0	75,0	100,0	100,0	100,0	100,0	75,0	40,0	100,0	100,0	0,2	0,0	94,4
TNSH	75,0	75,0	100,0	100,0	100,0	100,0	75,0	40,0	100,0	100,0	0,2	0,0	94,4
TA	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	94,4
TAG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	94,4
TAH	75,0	75,0	100,0	100,0	100,0	100,0	75,0	40,0	100,0	100,0	0,2	0,0	94,4
TAS	75,0	75,0	100,0	100,0	100,0	100,0	75,0	40,0	100,0	100,0	0,2	0,0	94,4
TCU	100,0	100,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	100,0	100,0	94,4
TCUG	100,0	100,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	100,0	100,0	94,4
TCUH	75,0	75,0	100,0	100,0	100,0	100,0	75,0	40,0	100,0	100,0	0,2	0,0	94,4
TPU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	94,4
TPUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	94,4
TPUH	75,0	75,0	100,0	100,0	100,0	100,0	75,0	40,0	100,0	100,0	0,2	0,0	94,4
TU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	94,4
TUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	94,4
TUH	75,0	75,0	100,0	100,0	100,0	100,0	75,0	40,0	100,0	100,0	0,2	0,0	94,4
TSCU	100,0	100,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	100,0	100,0	94,4
TSPU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	94,4
TSU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	94,4
TBCUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TBPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGCUG	100,0	100,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	100,0	100,0	94,4
TGPUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	94,4

Strength	TPU	TPUG	TPUH	TU	TUG	TUH	TSCU	TSPU	TSU	TBCUG	TBPUG	TGCUG	TGPUG
TN	50,0	50,0	100,0	66,7	66,7	96,4	100,0	50,0	83,3	-	-	100,0	50,0
TNG	50,0	50,0	100,0	66,7	66,7	96,4	100,0	50,0	83,3	-	-	100,0	50,0
TNH	50,0	50,0	100,0	33,4	33,3	96,4	0,0	50,0	83,3	-	-	0,0	50,0
TNS	50,0	50,0	100,0	33,4	33,3	96,4	0,0	50,0	83,3	-	-	0,0	50,0
TNSG	50,0	50,0	100,0	33,4	33,3	96,4	0,0	50,0	83,3	-	-	0,0	50,0
TNSH	50,0	50,0	100,0	33,4	33,3	96,4	0,0	50,0	83,3	-	-	0,0	50,0
TA	100,0	100,0	100,0	100,0	100,0	96,4	100,0	100,0	100,0	-	-	100,0	100,0
TAG	100,0	100,0	100,0	100,0	100,0	96,4	100,0	100,0	100,0	-	-	100,0	100,0
TAH	50,0	50,0	100,0	33,4	33,3	96,4	0,0	50,0	83,3	-	-	0,0	50,0
TAS	50,0	50,0	100,0	33,4	33,3	96,4	0,0	50,0	83,3	-	-	0,0	50,0
TCU	50,0	50,0	100,0	66,7	66,7	96,4	100,0	50,0	83,3	-	-	100,0	50,0
TCUG	50,0	50,0	100,0	66,7	66,7	96,4	100,0	50,0	83,3	-	-	100,0	50,0
TCUH	50,0	50,0	100,0	33,4	33,3	96,4	0,0	50,0	83,3	-	-	0,0	50,0
TPU	100,0	100,0	100,0	100,0	100,0	96,4	100,0	100,0	100,0	-	-	100,0	100,0
TPUG	100,0	100,0	100,0	100,0	100,0	96,4	100,0	100,0	100,0	-	-	100,0	100,0
TPUH	50,0	50,0	100,0	33,4	33,3	96,4	0,0	50,0	83,3	-	-	0,0	50,0
TU	100,0	100,0	100,0	100,0	100,0	96,4	100,0	100,0	100,0	-	-	100,0	100,0
TUG	100,0	100,0	100,0	100,0	100,0	96,4	100,0	100,0	100,0	-	-	100,0	100,0
TUH	50,0	50,0	100,0	33,4	33,3	96,4	0,0	50,0	83,3	-	-	0,0	50,0
TSCU	50,0	50,0	100,0	66,7	66,7	96,4	100,0	50,0	83,3	-	-	100,0	50,0
TSPU	100,0	100,0	100,0	100,0	100,0	96,4	100,0	100,0	100,0	-	-	100,0	100,0
TSU	100,0	100,0	100,0	100,0	100,0	96,4	100,0	100,0	100,0	-	-	100,0	100,0
TBCUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TBPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGCUG	50,0	50,0	100,0	66,7	66,7	96,4	100,0	50,0	83,3	-	-	100,0	50,0
TGPUG	100,0	100,0	100,0	100,0	100,0	96,4	100,0	100,0	100,0	-	-	100,0	100,0

Tabela 22 – Strength dos critérios para o programa matriz_vetor.

Strength (%)	TN	TNG	TNH	TNS	TNSG	TNSH	TA	TAG	TAH	TAS	TCU	TCUG	TCUH
TN	100,0	100,0	100,0	100,0	100,0	100,0	75,1	66,7	100,0	100,0	92,4	92,3	100,0
TNG	100,0	100,0	100,0	100,0	100,0	100,0	75,1	66,7	100,0	100,0	92,4	92,3	100,0
TNH	57,6	57,3	100,0	100,0	100,0	100,0	50,2	33,5	100,0	100,0	1,1	0,3	100,0
TNS	57,6	57,3	100,0	100,0	100,0	100,0	50,2	33,5	100,0	100,0	1,1	0,3	100,0
TNSG	57,6	57,3	100,0	100,0	100,0	100,0	50,2	33,5	100,0	100,0	1,1	0,3	100,0
TNSH	57,6	57,3	100,0	100,0	100,0	100,0	50,2	33,5	100,0	100,0	1,1	0,3	100,0
TA	100,0	100,0	100,0	100,0	100,0	100,0	99,9	99,9	100,0	100,0	92,4	92,3	100,0
TAG	100,0	100,0	100,0	100,0	100,0	100,0	99,9	99,9	100,0	100,0	92,4	92,3	100,0
TAH	57,6	57,3	100,0	100,0	100,0	100,0	50,2	33,5	100,0	100,0	1,1	0,3	100,0
TAS	57,6	57,3	100,0	100,0	100,0	100,0	50,2	33,5	100,0	100,0	1,1	0,3	100,0
TCU	100,0	100,0	100,0	100,0	100,0	100,0	75,1	66,7	100,0	100,0	92,4	92,3	100,0
TCUG	100,0	100,0	100,0	100,0	100,0	100,0	75,1	66,7	100,0	100,0	92,4	92,3	100,0
TCUH	57,6	57,3	100,0	100,0	100,0	100,0	50,2	33,5	100,0	100,0	1,1	0,3	100,0
TPU	100,0	100,0	100,0	100,0	100,0	100,0	99,9	99,9	100,0	100,0	92,4	92,3	100,0
TPUG	100,0	100,0	100,0	100,0	100,0	100,0	99,9	99,9	100,0	100,0	92,4	92,3	100,0
TPUH	57,6	57,3	100,0	100,0	100,0	100,0	50,2	33,5	100,0	100,0	1,1	0,3	100,0
TU	100,0	100,0	100,0	100,0	100,0	100,0	99,9	99,9	100,0	100,0	92,4	92,3	100,0
TUG	100,0	100,0	100,0	100,0	100,0	100,0	99,9	99,9	100,0	100,0	92,4	92,3	100,0
TUH	57,6	57,3	100,0	100,0	100,0	100,0	50,2	33,5	100,0	100,0	1,1	0,3	100,0
TSCU	100,0	100,0	100,0	100,0	100,0	100,0	75,1	66,7	100,0	100,0	92,4	92,3	100,0
TSPU	100,0	100,0	100,0	100,0	100,0	100,0	99,9	99,9	100,0	100,0	92,4	92,3	100,0
TSU	100,0	100,0	100,0	100,0	100,0	100,0	99,9	99,9	100,0	100,0	92,4	92,3	100,0
TBCUG	100,0	100,0	100,0	100,0	100,0	100,0	75,1	66,7	100,0	100,0	92,4	92,3	100,0
TBPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGCUG	100,0	100,0	100,0	100,0	100,0	100,0	75,1	66,7	100,0	100,0	92,4	92,3	100,0
TGPUG	100,0	100,0	100,0	100,0	100,0	100,0	99,9	99,9	100,0	100,0	92,4	92,3	100,0

Strength	TPU	TPUG	TPUH	TU	TUG	TUH	TSCU	TSPU	TSU	TBCUG	TBPUG	TGCUG	TGPUG
TN	61,2	61,1	100,0	74,3	74,2	100,0	100,0	62,5	85,7	100,0	-	100,0	62,5
TNG	61,2	61,1	100,0	74,3	74,2	100,0	100,0	62,5	85,7	100,0	-	100,0	62,5
TNH	39,2	39,0	100,0	23,2	22,8	100,0	0,4	37,6	85,7	0,4	-	0,4	37,6
TNS	39,2	39,0	100,0	23,2	22,8	100,0	0,4	37,6	85,7	0,4	-	0,4	37,6
TNSG	39,2	39,0	100,0	23,2	22,8	100,0	0,4	37,6	85,7	0,4	-	0,4	37,6
TNSH	39,2	39,0	100,0	23,2	22,8	100,0	0,4	37,6	85,7	0,4	-	0,4	37,6
TA	99,8	99,8	100,0	96,7	96,7	100,0	100,0	99,8	99,9	100,0	-	100,0	99,8
TAG	99,8	99,8	100,0	96,7	96,7	100,0	100,0	99,8	99,9	100,0	-	100,0	99,8
TAH	39,2	39,0	100,0	23,2	22,8	100,0	0,4	37,6	85,7	0,4	-	0,4	37,6
TAS	39,2	39,0	100,0	23,2	22,8	100,0	0,4	37,6	85,7	0,4	-	0,4	37,6
TCU	61,2	61,1	100,0	74,3	74,2	100,0	100,0	62,5	85,7	100,0	-	100,0	62,5
TCUG	61,2	61,1	100,0	74,3	74,2	100,0	100,0	62,5	85,7	100,0	-	100,0	62,5
TCUH	39,2	39,0	100,0	23,2	22,8	100,0	0,4	37,6	85,7	0,4	-	0,4	37,6
TPU	99,8	99,8	100,0	96,7	96,7	100,0	100,0	99,8	99,9	100,0	-	100,0	99,8
TPUG	99,8	99,8	100,0	96,7	96,7	100,0	100,0	99,8	99,9	100,0	-	100,0	99,8
TPUH	39,2	39,0	100,0	23,2	22,8	100,0	0,4	37,6	85,7	0,4	-	0,4	37,6
TU	99,8	99,8	100,0	96,7	96,7	100,0	100,0	99,8	99,9	100,0	-	100,0	99,8
TUG	99,8	99,8	100,0	96,7	96,7	100,0	100,0	99,8	99,9	100,0	-	100,0	99,8
TUH	39,2	39,0	100,0	23,2	22,8	100,0	0,4	37,6	85,7	0,4	-	0,4	37,6
TSCU	61,2	61,1	100,0	74,3	74,2	100,0	100,0	62,5	85,7	100,0	-	100,0	62,5
TSPU	99,8	99,8	100,0	96,7	96,7	100,0	100,0	99,8	99,9	100,0	-	100,0	99,8
TSU	99,8	99,8	100,0	96,7	96,7	100,0	100,0	99,8	99,9	100,0	-	100,0	99,8
TBCUG	61,2	61,1	100,0	74,3	74,2	100,0	100,0	62,5	85,7	100,0	-	100,0	62,5
TBPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGCUG	61,2	61,1	100,0	74,3	74,2	100,0	100,0	62,5	85,7	100,0	-	100,0	62,5
TGPUG	99,8	99,8	100,0	96,7	96,7	100,0	100,0	99,8	99,9	100,0	-	100,0	99,8

Tabela 23 – Strength dos critérios para o programa mult_diag.

Strength (%)	TN	TNG	TNH	TNS	TNSG	TNSH	TA	TAG	TAH	TAS	TCU	TCUG	TCUH
TN	95,6	95,6	100,0	100,0	100,0	100,0	93,5	92,5	100,0	100,0	87,6	87,6	100,0
TNG	95,6	95,6	100,0	100,0	100,0	100,0	93,5	92,5	100,0	100,0	87,6	87,6	100,0
TNH	68,2	68,2	100,0	100,0	100,0	100,0	53,3	46,1	100,0	100,0	18,0	17,9	100,0
TNS	68,2	68,2	100,0	100,0	100,0	100,0	53,3	46,1	100,0	100,0	18,0	17,9	100,0
TNSG	68,2	68,2	100,0	100,0	100,0	100,0	53,3	46,1	100,0	100,0	18,0	17,9	100,0
TNSH	68,2	68,2	100,0	100,0	100,0	100,0	53,3	46,1	100,0	100,0	18,0	17,9	100,0
TA	95,6	95,6	100,0	100,0	100,0	100,0	93,5	92,5	100,0	100,0	87,6	87,6	100,0
TAG	95,6	95,6	100,0	100,0	100,0	100,0	93,5	92,5	100,0	100,0	87,6	87,6	100,0
TAH	68,2	68,2	100,0	100,0	100,0	100,0	53,3	46,1	100,0	100,0	18,0	17,9	100,0
TAS	68,2	68,2	100,0	100,0	100,0	100,0	53,3	46,1	100,0	100,0	18,0	17,9	100,0
TCU	95,6	95,6	100,0	100,0	100,0	100,0	93,5	92,5	100,0	100,0	87,6	87,6	100,0
TCUG	95,6	95,6	100,0	100,0	100,0	100,0	93,5	92,5	100,0	100,0	87,6	87,6	100,0
TCUH	68,2	68,2	100,0	100,0	100,0	100,0	53,3	46,1	100,0	100,0	18,0	17,9	100,0
TPU	81,9	81,9	100,0	100,0	100,0	100,0	66,7	61,5	100,0	100,0	33,5	33,5	100,0
TPUG	81,9	81,9	100,0	100,0	100,0	100,0	66,7	61,5	100,0	100,0	33,5	33,5	100,0
TPUH	68,2	68,2	100,0	100,0	100,0	100,0	53,3	46,1	100,0	100,0	18,0	17,9	100,0
TU	95,6	95,6	100,0	100,0	100,0	100,0	93,5	92,5	100,0	100,0	87,6	87,6	100,0
TUG	95,6	95,6	100,0	100,0	100,0	100,0	93,5	92,5	100,0	100,0	87,6	87,6	100,0
TUH	68,2	68,2	100,0	100,0	100,0	100,0	53,3	46,1	100,0	100,0	18,0	17,9	100,0
TSCU	95,6	95,6	100,0	100,0	100,0	100,0	93,5	92,5	100,0	100,0	87,6	87,6	100,0
TSPU	81,9	81,9	100,0	100,0	100,0	100,0	66,7	61,5	100,0	100,0	33,5	33,5	100,0
TSU	86,5	86,5	100,0	100,0	100,0	100,0	80,2	77,2	100,0	100,0	51,7	51,7	100,0
TBCUG	86,5	86,5	100,0	100,0	100,0	100,0	80,2	77,2	100,0	100,0	51,7	51,7	100,0
TBPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGCUG	95,6	95,6	100,0	100,0	100,0	100,0	93,5	92,5	100,0	100,0	87,6	87,6	100,0
TGPUG	81,9	81,9	100,0	100,0	100,0	100,0	66,7	61,5	100,0	100,0	33,5	33,5	100,0

Strength	TPU	TPUG	TPUH	TU	TUG	TUH	TSCU	TSPU	TSU	TBCUG	TBPUG	TGCUG	TGPUG
TN	100,0	100,0	100,0	90,5	90,5	100,0	89,2	100,0	87,9	3,1	-	89,2	100,0
TNG	100,0	100,0	100,0	90,5	90,5	100,0	89,2	100,0	87,9	3,1	-	89,2	100,0
TNH	50,0	50,0	100,0	25,5	25,5	100,0	44,4	50,0	87,5	0,0	-	44,4	50,0
TNS	50,0	50,0	100,0	25,5	25,5	100,0	44,4	50,0	87,5	0,0	-	44,4	50,0
TNSG	50,0	50,0	100,0	25,5	25,5	100,0	44,4	50,0	87,5	0,0	-	44,4	50,0
TNSH	50,0	50,0	100,0	25,5	25,5	100,0	44,4	50,0	87,5	0,0	-	44,4	50,0
TA	100,0	100,0	100,0	90,5	90,5	100,0	89,2	100,0	87,9	3,1	-	89,2	100,0
TAG	100,0	100,0	100,0	90,5	90,5	100,0	89,2	100,0	87,9	3,1	-	89,2	100,0
TAH	50,0	50,0	100,0	25,5	25,5	100,0	44,4	50,0	87,5	0,0	-	44,4	50,0
TAS	50,0	50,0	100,0	25,5	25,5	100,0	44,4	50,0	87,5	0,0	-	44,4	50,0
TCU	100,0	100,0	100,0	90,5	90,5	100,0	89,2	100,0	87,9	3,1	-	89,2	100,0
TCUG	100,0	100,0	100,0	90,5	90,5	100,0	89,2	100,0	87,9	3,1	-	89,2	100,0
TCUH	50,0	50,0	100,0	25,5	25,5	100,0	44,4	50,0	87,5	0,0	-	44,4	50,0
TPU	100,0	100,0	100,0	49,2	49,2	100,0	55,7	100,0	87,7	1,6	-	55,7	100,0
TPUG	100,0	100,0	100,0	49,2	49,2	100,0	55,7	100,0	87,7	1,6	-	55,7	100,0
TPUH	50,0	50,0	100,0	25,5	25,5	100,0	44,4	50,0	87,5	0,0	-	44,4	50,0
TU	100,0	100,0	100,0	90,5	90,5	100,0	89,2	100,0	87,9	3,1	-	89,2	100,0
TUG	100,0	100,0	100,0	90,5	90,5	100,0	89,2	100,0	87,9	3,1	-	89,2	100,0
TUH	50,0	50,0	100,0	25,5	25,5	100,0	44,4	50,0	87,5	0,0	-	44,4	50,0
TSCU	100,0	100,0	100,0	90,5	90,5	100,0	89,2	100,0	87,9	3,1	-	89,2	100,0
TSPU	100,0	100,0	100,0	49,2	49,2	100,0	55,7	100,0	87,7	1,6	-	55,7	100,0
TSU	100,0	100,0	100,0	63,0	63,0	100,0	67,0	100,0	87,9	3,1	-	67,0	100,0
TBCUG	100,0	100,0	100,0	63,0	63,0	100,0	67,0	100,0	87,9	3,1	-	67,0	100,0
TBPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGCUG	100,0	100,0	100,0	90,5	90,5	100,0	89,2	100,0	87,9	3,1	-	89,2	100,0
TGPUG	100,0	100,0	100,0	49,2	49,2	100,0	55,7	100,0	87,7	1,6	-	55,7	100,0

Tabela 24 – Strength dos critérios para o programa mult_matrizes.

Strength (%)	TN	TNG	TNH	TNS	TNSG	TNSH	TA	TAG	TAH	TAS	TCU	TCUG	TCUH
TN	100,0	100,0	100,0	100,0	100,0	100,0	90,9	88,9	100,0	100,0	92,1	92,1	100,0
TNG	100,0	100,0	100,0	100,0	100,0	100,0	90,9	88,9	100,0	100,0	92,1	92,1	100,0
TNH	64,3	64,3	100,0	100,0	100,0	100,0	45,5	33,4	100,0	100,0	36,9	36,9	100,0
TNS	64,3	64,3	100,0	100,0	100,0	100,0	45,5	33,4	100,0	100,0	36,9	36,9	100,0
TNSG	64,3	64,3	100,0	100,0	100,0	100,0	45,5	33,4	100,0	100,0	36,9	36,9	100,0
TNSH	64,3	64,3	100,0	100,0	100,0	100,0	45,5	33,4	100,0	100,0	36,9	36,9	100,0
TA	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	92,1	92,1	100,0
TAG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	92,1	92,1	100,0
TAH	64,3	64,3	100,0	100,0	100,0	100,0	45,5	33,4	100,0	100,0	36,9	36,9	100,0
TAS	64,3	64,3	100,0	100,0	100,0	100,0	45,5	33,4	100,0	100,0	36,9	36,9	100,0
TCU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	96,0	96,0	100,0
TCUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	96,0	96,0	100,0
TCUH	64,3	64,3	100,0	100,0	100,0	100,0	45,5	33,4	100,0	100,0	36,9	36,9	100,0
TPU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	95,9	95,9	100,0
TPUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	95,9	95,9	100,0
TPUH	64,3	64,3	100,0	100,0	100,0	100,0	45,5	33,4	100,0	100,0	36,9	36,9	100,0
TU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	96,0	96,0	100,0
TUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	96,0	96,0	100,0
TUH	64,3	64,3	100,0	100,0	100,0	100,0	45,5	33,4	100,0	100,0	36,9	36,9	100,0
TSCU	89,3	89,3	100,0	100,0	100,0	100,0	63,6	55,5	100,0	100,0	63,2	63,1	100,0
TSPU	100,0	100,0	100,0	100,0	100,0	100,0	90,9	88,9	100,0	100,0	92,1	92,1	100,0
TSU	100,0	100,0	100,0	100,0	100,0	100,0	90,9	88,9	100,0	100,0	92,1	92,1	100,0
TBCUG	100,0	100,0	100,0	100,0	100,0	100,0	99,4	99,3	100,0	100,0	93,3	93,3	100,0
TBPUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	95,9	95,9	100,0
TGCUG	89,3	89,3	100,0	100,0	100,0	100,0	63,6	55,5	100,0	100,0	63,2	63,1	100,0
TGPUG	100,0	100,0	100,0	100,0	100,0	100,0	90,9	88,9	100,0	100,0	92,1	92,1	100,0

Strength	TPU	TPUG	TPUH	TU	TUG	TUH	TSCU	TSPU	TSU	TBCUG	TBPUG	TGCUG	TGPUG
TN	78,6	78,6	100,0	86,4	86,4	100,0	100,0	100,0	100,0	50,0	25,0	100,0	100,0
TNG	78,6	78,6	100,0	86,4	86,4	100,0	100,0	100,0	100,0	50,0	25,0	100,0	100,0
TNH	35,8	35,7	100,0	36,4	36,4	100,0	60,1	50,0	85,7	0,0	0,0	60,1	50,0
TNS	35,8	35,7	100,0	36,4	36,4	100,0	60,1	50,0	85,7	0,0	0,0	60,1	50,0
TNSG	35,8	35,7	100,0	36,4	36,4	100,0	60,1	50,0	85,7	0,0	0,0	60,1	50,0
TNSH	35,8	35,7	100,0	36,4	36,4	100,0	60,1	50,0	85,7	0,0	0,0	60,1	50,0
TA	85,7	85,7	100,0	89,4	89,4	100,0	100,0	100,0	100,0	50,0	50,0	100,0	100,0
TAG	85,7	85,7	100,0	89,4	89,4	100,0	100,0	100,0	100,0	50,0	50,0	100,0	100,0
TAH	35,8	35,7	100,0	36,4	36,4	100,0	60,1	50,0	85,7	0,0	0,0	60,1	50,0
TAS	35,8	35,7	100,0	36,4	36,4	100,0	60,1	50,0	85,7	0,0	0,0	60,1	50,0
TCU	89,6	89,6	100,0	93,3	93,3	100,0	100,0	100,0	100,0	61,7	63,8	100,0	100,0
TCUG	89,6	89,6	100,0	93,3	93,3	100,0	100,0	100,0	100,0	61,7	63,8	100,0	100,0
TCUH	35,8	35,7	100,0	36,4	36,4	100,0	60,1	50,0	85,7	0,0	0,0	60,1	50,0
TPU	89,6	89,6	100,0	93,2	93,2	100,0	100,0	100,0	100,0	61,7	63,8	100,0	100,0
TPUG	89,6	89,6	100,0	93,2	93,2	100,0	100,0	100,0	100,0	61,7	63,8	100,0	100,0
TPUH	35,8	35,7	100,0	36,4	36,4	100,0	60,1	50,0	85,7	0,0	0,0	60,1	50,0
TU	89,6	89,6	100,0	93,3	93,3	100,0	100,0	100,0	100,0	61,7	63,8	100,0	100,0
TUG	89,6	89,6	100,0	93,3	93,3	100,0	100,0	100,0	100,0	61,7	63,8	100,0	100,0
TUH	35,8	35,7	100,0	36,4	36,4	100,0	60,1	50,0	85,7	0,0	0,0	60,1	50,0
TSCU	50,0	50,0	100,0	57,6	57,6	100,0	100,0	50,0	85,7	0,0	25,0	100,0	50,0
TSPU	78,6	78,6	100,0	86,4	86,4	100,0	100,0	100,0	100,0	50,0	25,0	100,0	100,0
TSU	78,6	78,6	100,0	86,4	86,4	100,0	100,0	100,0	100,0	50,0	25,0	100,0	100,0
TBCUG	89,1	89,1	100,0	91,5	91,5	100,0	100,0	100,0	100,0	61,7	61,9	100,0	100,0
TBPUG	89,6	89,6	100,0	93,2	93,2	100,0	100,0	100,0	100,0	61,7	63,8	100,0	100,0
TGCUG	50,0	50,0	100,0	57,6	57,6	100,0	100,0	50,0	85,7	0,0	25,0	100,0	50,0
TGPUG	78,6	78,6	100,0	86,4	86,4	100,0	100,0	100,0	100,0	50,0	25,0	100,0	100,0

Tabela 25 – Strength dos critérios para o programa soma_elem.

Strength (%)	TN	TNG	TNH	TNS	TNSG	TNSH	TA	TAG	TAH	TAS	TCU	TCUG	TCUH
TN	91,9	91,9	100,0	100,0	100,0	100,0	75,0	68,7	100,0	100,0	62,9	62,9	100,0
TNG	91,9	91,9	100,0	100,0	100,0	100,0	75,0	68,7	100,0	100,0	62,9	62,9	100,0
TNH	16,3	16,3	100,0	50,2	50,0	100,0	20,1	12,5	100,0	50,0	0,2	0,1	100,0
TNS	81,1	81,1	100,0	100,0	100,0	100,0	65,0	56,2	100,0	100,0	57,8	57,7	100,0
TNSG	81,1	81,1	100,0	100,0	100,0	100,0	65,0	56,2	100,0	100,0	57,8	57,7	100,0
TNSH	16,3	16,3	100,0	50,2	50,0	100,0	20,1	12,5	100,0	50,0	0,2	0,1	100,0
TA	91,9	91,9	100,0	100,0	100,0	100,0	85,0	81,2	100,0	100,0	65,5	65,4	100,0
TAG	91,9	91,9	100,0	100,0	100,0	100,0	85,0	81,2	100,0	100,0	65,5	65,4	100,0
TAH	16,3	16,3	100,0	50,2	50,0	100,0	20,1	12,5	100,0	50,0	0,2	0,1	100,0
TAS	81,1	81,1	100,0	100,0	100,0	100,0	65,0	56,2	100,0	100,0	57,8	57,7	100,0
TCU	86,5	86,5	100,0	100,0	100,0	100,0	75,0	68,7	100,0	100,0	65,5	65,4	100,0
TCUG	86,5	86,5	100,0	100,0	100,0	100,0	75,0	68,7	100,0	100,0	65,5	65,4	100,0
TCUH	16,3	16,3	100,0	50,2	50,0	100,0	20,1	12,5	100,0	50,0	0,2	0,1	100,0
TPU	91,9	91,9	100,0	100,0	100,0	100,0	85,0	81,2	100,0	100,0	65,5	65,4	100,0
TPUG	91,9	91,9	100,0	100,0	100,0	100,0	85,0	81,2	100,0	100,0	65,5	65,4	100,0
TPUH	16,3	16,3	100,0	50,2	50,0	100,0	20,1	12,5	100,0	50,0	0,2	0,1	100,0
TU	91,9	91,9	100,0	100,0	100,0	100,0	85,0	81,2	100,0	100,0	65,5	65,4	100,0
TUG	91,9	91,9	100,0	100,0	100,0	100,0	85,0	81,2	100,0	100,0	65,5	65,4	100,0
TUH	16,3	16,3	100,0	50,2	50,0	100,0	20,1	12,5	100,0	50,0	0,2	0,1	100,0
TSCU	81,1	81,1	100,0	100,0	100,0	100,0	65,0	56,2	100,0	100,0	57,8	57,7	100,0
TSPU	86,5	86,5	100,0	100,0	100,0	100,0	75,0	68,7	100,0	100,0	65,5	65,4	100,0
TSU	81,1	81,1	100,0	100,0	100,0	100,0	65,0	56,2	100,0	100,0	57,8	57,7	100,0
TBCUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TBPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGCUG	81,1	81,1	100,0	100,0	100,0	100,0	65,0	56,2	100,0	100,0	57,8	57,7	100,0
TGPUG	91,9	91,9	100,0	100,0	100,0	100,0	75,0	68,7	100,0	100,0	62,9	62,9	100,0

Strength	TPU	TPUG	TPUH	TU	TUG	TUH	TSCU	TSPU	TSU	TBCUG	TBPUG	TGCUG	TGPUG
TN	86,1	86,1	100,0	70,2	70,2	100,0	100,0	100,0	83,3	-	-	100,0	100,0
TNG	86,1	86,1	100,0	70,2	70,2	100,0	100,0	100,0	83,3	-	-	100,0	100,0
TNH	22,3	22,3	100,0	7,2	7,1	100,0	0,1	50,0	50,0	-	-	0,1	50,0
TNS	63,9	63,9	100,0	59,7	59,7	100,0	100,0	50,0	83,3	-	-	100,0	50,0
TNSG	63,9	63,9	100,0	59,7	59,7	100,0	100,0	50,0	83,3	-	-	100,0	50,0
TNSH	22,3	22,3	100,0	7,2	7,1	100,0	0,1	50,0	50,0	-	-	0,1	50,0
TA	97,2	97,2	100,0	75,5	75,5	100,0	100,0	100,0	83,3	-	-	100,0	100,0
TAG	97,2	97,2	100,0	75,5	75,5	100,0	100,0	100,0	83,3	-	-	100,0	100,0
TAH	22,3	22,3	100,0	7,2	7,1	100,0	0,1	50,0	50,0	-	-	0,1	50,0
TAS	63,9	63,9	100,0	59,7	59,7	100,0	100,0	50,0	83,3	-	-	100,0	50,0
TCU	75,0	75,0	100,0	68,5	68,4	100,0	100,0	50,0	83,3	-	-	100,0	50,0
TCUG	75,0	75,0	100,0	68,5	68,4	100,0	100,0	50,0	83,3	-	-	100,0	50,0
TCUH	22,3	22,3	100,0	7,2	7,1	100,0	0,1	50,0	50,0	-	-	0,1	50,0
TPU	97,2	97,2	100,0	75,5	75,5	100,0	100,0	100,0	83,3	-	-	100,0	100,0
TPUG	97,2	97,2	100,0	75,5	75,5	100,0	100,0	100,0	83,3	-	-	100,0	100,0
TPUH	22,3	22,3	100,0	7,2	7,1	100,0	0,1	50,0	50,0	-	-	0,1	50,0
TU	97,2	97,2	100,0	75,5	75,5	100,0	100,0	100,0	83,3	-	-	100,0	100,0
TUG	97,2	97,2	100,0	75,5	75,5	100,0	100,0	100,0	83,3	-	-	100,0	100,0
TUH	22,3	22,3	100,0	7,2	7,1	100,0	0,1	50,0	50,0	-	-	0,1	50,0
TSCU	63,9	63,9	100,0	59,7	59,7	100,0	100,0	50,0	83,3	-	-	100,0	50,0
TSPU	75,0	75,0	100,0	68,5	68,4	100,0	100,0	50,0	83,3	-	-	100,0	50,0
TSU	63,9	63,9	100,0	59,7	59,7	100,0	100,0	50,0	83,3	-	-	100,0	50,0
TBCUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TBPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGCUG	63,9	63,9	100,0	59,7	59,7	100,0	100,0	50,0	83,3	-	-	100,0	50,0
TGPUG	86,1	86,1	100,0	70,2	70,2	100,0	100,0	100,0	83,3	-	-	100,0	100,0

Tabela 26 – Strength dos critérios para o programa soma_matriz.

Strength (%)	TN	TNG	TNH	TNS	TNSG	TNSH	TA	TAG	TAH	TAS	TCU	TCUG	TCUH
TN	100,0	100,0	100,0	100,0	100,0	100,0	77,8	71,4	100,0	100,0	100,0	100,0	100,0
TNG	100,0	100,0	100,0	100,0	100,0	100,0	77,8	71,4	100,0	100,0	100,0	100,0	100,0
TNH	41,7	41,7	100,0	100,0	100,0	100,0	44,4	28,6	100,0	100,0	0,0	0,0	100,0
TNS	41,7	41,7	100,0	100,0	100,0	100,0	44,4	28,6	100,0	100,0	0,0	0,0	100,0
TNSG	41,7	41,7	100,0	100,0	100,0	100,0	44,4	28,6	100,0	100,0	0,0	0,0	100,0
TNSH	41,7	41,7	100,0	100,0	100,0	100,0	44,4	28,6	100,0	100,0	0,0	0,0	100,0
TA	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TAG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TAH	41,7	41,7	100,0	100,0	100,0	100,0	44,4	28,6	100,0	100,0	0,0	0,0	100,0
TAS	41,7	41,7	100,0	100,0	100,0	100,0	44,4	28,6	100,0	100,0	0,0	0,0	100,0
TCU	100,0	100,0	100,0	100,0	100,0	100,0	77,8	71,4	100,0	100,0	100,0	100,0	100,0
TCUG	100,0	100,0	100,0	100,0	100,0	100,0	77,8	71,4	100,0	100,0	100,0	100,0	100,0
TCUH	41,7	41,7	100,0	100,0	100,0	100,0	44,4	28,6	100,0	100,0	0,0	0,0	100,0
TPU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TPUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TPUH	41,7	41,7	100,0	100,0	100,0	100,0	44,4	28,6	100,0	100,0	0,0	0,0	100,0
TU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TUH	41,7	41,7	100,0	100,0	100,0	100,0	44,4	28,6	100,0	100,0	0,0	0,0	100,0
TSCU	91,7	91,7	100,0	100,0	100,0	100,0	55,5	42,9	100,0	100,0	77,3	77,3	100,0
TSPU	91,7	91,7	100,0	100,0	100,0	100,0	77,8	71,4	100,0	100,0	77,3	77,3	100,0
TSU	91,7	91,7	100,0	100,0	100,0	100,0	77,8	71,4	100,0	100,0	77,3	77,3	100,0
TBCUG	100,0	100,0	100,0	100,0	100,0	100,0	77,8	71,4	100,0	100,0	100,0	100,0	100,0
TBPUG	100,0	100,0	100,0	100,0	100,0	100,0	77,8	71,4	100,0	100,0	100,0	100,0	100,0
TGCUG	91,7	91,7	100,0	100,0	100,0	100,0	55,5	42,9	100,0	100,0	77,3	77,3	100,0
TGPUG	91,7	91,7	100,0	100,0	100,0	100,0	77,8	71,4	100,0	100,0	77,3	77,3	100,0
Strength	**TPU**	**TPUG**	**TPUH**	**TU**	**TUG**	**TUH**	**TSCU**	**TSPU**	**TSU**	**TBCUG**	**TBPUG**	**TGCUG**	**TGPUG**
TN	62,5	62,5	100,0	84,2	84,2	100,0	100,0	50,0	85,7	100,0	100,0	100,0	50,0
TNG	62,5	62,5	100,0	84,2	84,2	100,0	100,0	50,0	85,7	100,0	100,0	100,0	50,0
TNH	37,5	37,5	100,0	15,8	15,8	100,0	0,0	50,0	85,7	0,0	0,0	0,0	50,0
TNS	37,5	37,5	100,0	15,8	15,8	100,0	0,0	50,0	85,7	0,0	0,0	0,0	50,0
TNSG	37,5	37,5	100,0	15,8	15,8	100,0	0,0	50,0	85,7	0,0	0,0	0,0	50,0
TNSH	37,5	37,5	100,0	15,8	15,8	100,0	0,0	50,0	85,7	0,0	0,0	0,0	50,0
TA	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TAG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TAH	37,5	37,5	100,0	15,8	15,8	100,0	0,0	50,0	85,7	0,0	0,0	0,0	50,0
TAS	37,5	37,5	100,0	15,8	15,8	100,0	0,0	50,0	85,7	0,0	0,0	0,0	50,0
TCU	62,5	62,5	100,0	84,2	84,2	100,0	100,0	50,0	85,7	100,0	100,0	100,0	50,0
TCUG	62,5	62,5	100,0	84,2	84,2	100,0	100,0	50,0	85,7	100,0	100,0	100,0	50,0
TCUH	37,5	37,5	100,0	15,8	15,8	100,0	0,0	50,0	85,7	0,0	0,0	0,0	50,0
TPU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TPUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TPUH	37,5	37,5	100,0	15,8	15,8	100,0	0,0	50,0	85,7	0,0	0,0	0,0	50,0
TU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TUH	37,5	37,5	100,0	15,8	15,8	100,0	0,0	50,0	85,7	0,0	0,0	0,0	50,0
TSCU	50,0	50,0	100,0	65,8	65,8	100,0	100,0	50,0	85,7	50,0	50,0	100,0	50,0
TSPU	87,5	87,5	100,0	81,6	81,6	100,0	100,0	100,0	100,0	50,0	50,0	100,0	100,0
TSU	87,5	87,5	100,0	81,6	81,6	100,0	100,0	100,0	100,0	50,0	50,0	100,0	100,0
TBCUG	62,5	62,5	100,0	84,2	84,2	100,0	100,0	50,0	85,7	100,0	100,0	100,0	50,0
TBPUG	62,5	62,5	100,0	84,2	84,2	100,0	100,0	50,0	85,7	100,0	100,0	100,0	50,0
TGCUG	50,0	50,0	100,0	65,8	65,8	100,0	100,0	50,0	85,7	50,0	50,0	100,0	50,0
TGPUG	87,5	87,5	100,0	81,6	81,6	100,0	100,0	100,0	100,0	50,0	50,0	100,0	100,0

Tabela 27 – Strength dos critérios para o programa soma_vetores.

Strength (%)	TN	TNG	TNH	TNS	TNSG	TNSH	TA	TAG	TAH	TAS	TCU	TCUG	TCUH
TN	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TNG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TNH	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TNS	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TNSG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TNSH	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TA	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TAG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TAH	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TAS	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TCU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TCUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TCUH	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TPU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TPUH	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TUH	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TSCU	-	-	-	-	-	-	-	-	-	-	-	-	-
TSPU	-	-	-	-	-	-	-	-	-	-	-	-	-
TSU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	100,0
TBCUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TBPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGCUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGPUG	-	-	-	-	-	-	-	-	-	-	-	-	-

Strength	TPU	TPUG	TPUH	TU	TUG	TUH	TSCU	TSPU	TSU	TBCUG	TBPUG	TGCUG	TGPUG
TN	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TNG	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TNH	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TNS	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TNSG	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TNSH	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TA	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TAG	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TAH	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TAS	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TCU	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TCUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TCUH	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TPU	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TPUH	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TU	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TUH	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TSCU	-	-	-	-	-	-	-	-	-	-	-	-	-
TSPU	-	-	-	-	-	-	-	-	-	-	-	-	-
TSU	100,0	-	100,0	100,0	-	100,0	-	-	100,0	-	-	-	-
TBCUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TBPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGCUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGPUG	-	-	-	-	-	-	-	-	-	-	-	-	-

Tabela 28 – Strength dos critérios para o programa vetores_iguais.

Strength (%)	TN	TNG	TNH	TNS	TNSG	TNSH	TA	TAG	TAH	TAS	TCU	TCUG	TCUH
TN	100,0	100,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	100,0	100,0	100,0
TNG	100,0	100,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	100,0	100,0	100,0
TNH	75,0	75,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	50,0	50,0	100,0
TNS	75,0	75,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	50,0	50,0	100,0
TNSG	75,0	75,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	50,0	50,0	100,0
TNSH	75,0	75,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	50,0	50,0	100,0
TA	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TAG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TAH	75,0	75,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	50,0	50,0	100,0
TAS	75,0	75,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	50,0	50,0	100,0
TCU	100,0	100,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	100,0	100,0	100,0
TCUG	100,0	100,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	100,0	100,0	100,0
TCUH	75,0	75,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	50,0	50,0	100,0
TPU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TPUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TPUH	75,0	75,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	50,0	50,0	100,0
TU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TUH	75,0	75,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	50,0	50,0	100,0
TSCU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TSPU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0
TSU	75,0	75,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	50,0	50,0	100,0
TBCUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TBPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGCUG	100,0	100,0	100,0	100,0	100,0	100,0	75,0	50,0	100,0	100,0	100,0	100,0	100,0
TGPUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0

Strength	TPU	TPUG	TPUH	TU	TUG	TUH	TSCU	TSPU	TSU	TBCUG	TBPUG	TGCUG	TGPUG
TN	50,0	50,0	100,0	60,0	60,0	100,0	100,0	50,0	100,0	-	-	100,0	50,0
TNG	50,0	50,0	100,0	60,0	60,0	100,0	100,0	50,0	100,0	-	-	100,0	50,0
TNH	50,0	50,0	100,0	50,0	50,0	100,0	0,0	50,0	100,0	-	-	0,0	50,0
TNS	50,0	50,0	100,0	50,0	50,0	100,0	0,0	50,0	100,0	-	-	0,0	50,0
TNSG	50,0	50,0	100,0	50,0	50,0	100,0	0,0	50,0	100,0	-	-	0,0	50,0
TNSH	50,0	50,0	100,0	50,0	50,0	100,0	0,0	50,0	100,0	-	-	0,0	50,0
TA	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	-	100,0	100,0
TAG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	-	100,0	100,0
TAH	50,0	50,0	100,0	50,0	50,0	100,0	0,0	50,0	100,0	-	-	0,0	50,0
TAS	50,0	50,0	100,0	50,0	50,0	100,0	0,0	50,0	100,0	-	-	0,0	50,0
TCU	50,0	50,0	100,0	60,0	60,0	100,0	100,0	50,0	100,0	-	-	100,0	50,0
TCUG	50,0	50,0	100,0	60,0	60,0	100,0	100,0	50,0	100,0	-	-	100,0	50,0
TCUH	50,0	50,0	100,0	50,0	50,0	100,0	0,0	50,0	100,0	-	-	0,0	50,0
TPU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	-	100,0	100,0
TPUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	-	100,0	100,0
TPUH	50,0	50,0	100,0	50,0	50,0	100,0	0,0	50,0	100,0	-	-	0,0	50,0
TU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	-	100,0	100,0
TUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	-	100,0	100,0
TUH	50,0	50,0	100,0	50,0	50,0	100,0	0,0	50,0	100,0	-	-	0,0	50,0
TSCU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	-	100,0	100,0
TSPU	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	-	100,0	100,0
TSU	50,0	50,0	100,0	50,0	50,0	100,0	0,0	50,0	100,0	-	-	0,0	50,0
TBCUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TBPUG	-	-	-	-	-	-	-	-	-	-	-	-	-
TGCUG	50,0	50,0	100,0	60,0	60,0	100,0	100,0	50,0	100,0	-	-	100,0	50,0
TGPUG	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	100,0	-	-	100,0	100,0

7.5 Teste de hipóteses

Para testar as hipóteses apresentadas na Seção 7.3 foi utilizado o programa Minitab para aplicar métodos estatísticos para análise. Os resultados dos métodos selecionados resultam na aceitação ou rejeição das hipóteses nulas ou alternativas e tirar conclusões do experimento realizado. Com isso, mostramos a análise estatística dos resultados da seção anterior para o do custo dos critérios de teste e da eficácia alcançada.

7.5.1 Análise do custo dos critérios

Na análise do teste de custo, consideramos o número de elementos requeridos, número de elementos requeridos não executáveis e quantidade de casos de teste adequado para cada critério.

Para o número de elementos requeridos, primeiramente verificamos se os dados obtidos possuem uma distribuição normal utilizando o teste de Ryan-Joiner, similar ao do Shapiro-Wilk (YAP; SIM, 2011). O teste apresentou valor RJ 0,741, e Valor-P <0,010 sendo inferior a 0,05, indicando que a distribuição não é normal. Dessa forma, foi utilizado o teste não paramétrico Kruskal-Wallis para analisar os dados. Esse método realiza uma comparação par a par entre os critérios buscando verificar se há diferenças significativas entre eles. O resultado do teste é apresentado na Tabela 29. A primeira coluna indica o método utilizado, como há na relação de dados critérios que possuem o mesmo valor de elementos requeridos, ele faz a análise considerando esses empates e sem considerá-los. A segunda coluna indica os graus de liberdade da amostra. A terceira coluna é o teste estatístico e a última coluna é uma probabilidade que mede a evidência contra a hipótese nula. Para ambos os métodos o Valor-p ficou em 0, sendo menor que 0.05 e rejeitando a hipótese nula (HN_1) por consequência. A Tabela 30 apresenta a estatística descritiva do teste para os elementos requeridos. A primeira coluna indica o critério de teste. A segunda é o número de amostras de cada critério. A terceira é o ponto médio do conjunto de dados. Posto médio é uma classificação dos postos para todas as observações dentro de cada amostra. Valor-Z indica como a classificação média de cada grupo se compara à classificação média de todas as observações.

Tabela 29 – Resultado do teste não paramétrico Kruskal-Wallis para os elementos requeridos.

Método	**GL**	**Valor H**	**Valor-p**
Não ajustado para empates	25	177,91	0,000
Ajustado para empates	25	178,01	0,000

Para a análise estatística do número de elementos requeridos não executáveis foi novamente utilizado o teste de Ryan-Joiner. Ele apresentou valor RJ 0,761 e Valor-P <0,010, o que indica que a distribuição também não é normal. Portanto, foi utilizado o

Tabela 30 – Estatística descritiva do teste não paramétrico Kruskal-Wallis para os elementos requeridos.

Critério	**N**	**Mediana**	**Posto médio**	**Valor-Z**
todas-arestas	11	32771	173	1,8
todas-arestas-grid	11	16384	158	1,13
todas-arestas-host	11	4	6,1	-5,61
todas-arestas-sinc	11	8192	119,6	-0,57
todos-bloco-c-usos-grid	6	15872	144,8	0,4
todos-bloco-p-usos-grid	3	27648	167,2	0,79
todos-c-usos	11	39970	183,1	2,24
todos-c-usos-grid	10	61440	197,1	2,73
todos-c-usos-host	11	19	45,7	-3,85
todos-global-c-usos-grid	9	8192	131,3	-0,05
todos-global-p-usos-grid	10	15104	148,5	0,68
Todos-nos	11	37915	189,3	2,52
todos-nos-grid	11	37888	186	2,37
todos-nos-host	11	20	50,8	-3,63
todos-nos-sinc	11	8199	125,9	-0,29
todos-nos-sinc-grid	11	10	22,4	-4,89
todos-nos-sinc-host	11	8192	119,6	-0,57
todos-p-usos	11	49162	181,1	2,16
todos-p-usos-grid	10	57344	195,6	2,66
todos-p-usos-host	11	10	26,1	-4,72
todos-s-c-usos	9	8192	131,3	-0,05
todos-s-p-usos	10	12288	144,8	0,52
todos-s-usos	11	26624	160,1	1,22
todos-usos	11	81947	207	3,31
todos-usos-grid	10	121600	222,5	3,8
todos-usos-host	11	28	60,9	-3,18
Global	264		132,5	

teste não paramétrico Kruskal-Wallis. A Tabela 31 apresenta os resultados obtidos. O Valor-p em ambos os métodos ficou em 0,00, sendo menor que 0,05 e assim rejeitando a hipótese nula (HN_1). A estatística descritiva do teste é mostrada na Tabela 32.

Tabela 31 – Resultado do teste não paramétrico Kruskal-Wallis para os elementos requeridos não executáveis.

Método	**GL**	**Valor H**	**Valor-p**
Não ajustado para empates	25	109,82	0,000
Ajustado para empates	25	131,14	0,000

O teste de Ryan-Joiner foi aplicado nos dados relacionados aos casos de teste e apresentou valor RJ 0,572 e Valor-P <0,010, apontando que a distribuição dos dados não

Tabela 32 – Estatística descritiva do teste não paramétrico Kruskal-Wallis para os elementos requeridos não executáveis.

Critérios	**N**	**Mediana**	**Posto médio**	**Valor-Z**
todas-arestas	11	2	173	1,8
todas-arestas-grid	11	2	173	1,8
todas-arestas-host	11	0	72,5	-2,66
todas-arestas-sinc	11	0	72,5	-2,66
todos-bloco-c-usos-grid	6	2768	176,6	1,43
todos-bloco-p-usos-grid	3	2963	174,8	0,97
todos-c-usos	11	1568	182,3	2,21
todos-c-usos-grid	10	7688	185,3	2,23
todos-c-usos-host	11	0	79,8	-2,34
todos-global-c-usos-grid	9	0	119,8	-0,51
todos-global-p-usos-grid	10	2	159,6	1,14
Todos-nos	11	0	140,6	0,36
todos-nos-grid	11	0	140,6	0,36
todos-nos-host	11	0	72,5	-2,66
todos-nos-sinc	11	0	72,5	-2,66
todos-nos-sinc-grid	11	0	72,5	-2,66
todos-nos-sinc-host	11	0	72,5	-2,66
todos-p-usos	11	7	177,6	2
todos-p-usos-grid	10	9,5	188,2	2,35
todos-p-usos-host	11	0	72,5	-2,66
todos-s-c-usos	9	0	119,8	-0,51
todos-s-p-usos	10	2	157,8	1,07
todos-s-usos	11	1	168,5	1,6
todos-usos	11	4537	198,2	2,91
todos-usos-grid	10	9432,5	210,3	3,29
todos-usos-host	11	0	79,8	-2,34
Global	264		132,5	

é normal. Logo, foi utilizado o teste Kruskal-Wallis para análise dos dados. Na Tabela 33 mostramos o resultado obtido no teste, sendo o Valor-p em 0,00 para ambos os métodos, consequentemente rejeitando a hipótese nula (HN_1). A estatística descritiva do teste é mostrada na Tabela 34.

Tabela 33 – Resultado do teste não paramétrico Kruskal-Wallis para o número de casos de teste.

Método	**GL**	**Valor H**	**Valor-p**
Não ajustado para empates	25	130,84	0,000
Ajustado para empates	25	151,97	0,000

Tabela 34 – Estatística descritiva do teste não paramétrico Kruskal-Wallis para o número de casos de teste.

Critérios	**N**	**Mediana**	**Posto médio**	**Valor-Z**
todas-arestas	11	3	190,5	2,58
todas-arestas-grid	11	3	190,5	2,58
todas-arestas-host	11	1	75,2	-2,54
todas-arestas-sinc	11	1	66,5	-2,93
todos-bloco-c-usos-grid	6	2,5	178,8	1,5
todos-bloco-p-usos-grid	3	4	217,5	1,94
todos-c-usos	11	2	158,8	1,17
todos-c-usos-grid	10	2,5	168	1,5
todos-c-usos-host	11	1	75,2	-2,54
todos-global-c-usos-grid	9	1	115,1	-0,7
todos-global-p-usos-grid	10	2	180,8	2,04
Todos-nos	11	2	141,2	0,39
todos-nos-grid	11	2	141,2	0,39
todos-nos-host	11	1	66,5	-2,93
todos-nos-sinc	11	1	66,5	-2,93
todos-nos-sinc-grid	11	1	66,5	-2,93
todos-nos-sinc-host	11	1	66,5	-2,93
todos-p-usos	11	3	189,7	2,54
todos-p-usos-grid	10	3	202,1	2,94
todos-p-usos-host	11	1	75,2	-2,54
todos-s-c-usos	9	1	115,1	-0,7
todos-s-p-usos	10	2	185,6	2,24
todos-s-usos	11	2	131,3	-0,05
todos-usos	11	3	197,4	2,88
todos-usos-grid	10	3	210,4	3,29
todos-usos-host	11	1	75,2	-2,54
Global	264		132,5	

7.5.2 Análise da eficácia dos critérios

Na análise da eficácia dos critérios de teste, realizamos o teste de normalidade Ryan-Joiner, que apresentou valor RJ 0,986, e Valor-P <0,010, demonstrando que a distribuição dos dados não é normal. Dessa forma, o teste Kruskal-Wallis foi utilizado para a análise dos dados. No resultado apresentado Tabela 35 o Valor-p está em 0,00 para ambos os métodos, portanto, rejeitando a hipótese nula (HN_2). Isso indica que há diferenças entre as eficácias de pelo menos um dos critérios de teste propostos. Na Tabela 36 é mostrada a estatística descritiva do teste.

Tabela 35 – Resultado do teste não paramétrico Kruskal-Wallis para a eficácia.

Método	**GL**	**Valor H**	**Valor-p**
Não ajustado para empates	25	112,60	0,000
Ajustado para empates	25	113,22	0,000

Tabela 36 – Estatística descritiva do teste não paramétrico Kruskal-Wallis para a eficácia.

Critérios	**N**	**Mediana**	**Posto médio**	**Valor-Z**
todas-arestas	11	3	190,5	2,58
todas-arestas-grid	11	3	190,5	2,58
todas-arestas-host	11	1	75,2	-2,54
todas-arestas-sinc	11	1	66,5	-2,93
todos-bloco-c-usos-grid	6	2,5	178,8	1,5
todos-bloco-p-usos-grid	3	4	217,5	1,94
todos-c-usos	11	2	158,8	1,17
todos-c-usos-grid	10	2,5	168	1,5
todos-c-usos-host	11	1	75,2	-2,54
todos-global-c-usos-grid	9	1	115,1	-0,7
todos-global-p-usos-grid	10	2	180,8	2,04
Todos-nos	11	2	141,2	0,39
todos-nos-grid	11	2	141,2	0,39
todos-nos-host	11	1	66,5	-2,93
todos-nos-sinc	11	1	66,5	-2,93
todos-nos-sinc-grid	11	1	66,5	-2,93
todos-nos-sinc-host	11	1	66,5	-2,93
todos-p-usos	11	3	189,7	2,54
todos-p-usos-grid	10	3	202,1	2,94
todos-p-usos-host	11	1	75,2	-2,54
todos-s-c-usos	9	1	115,1	-0,7
todos-s-p-usos	10	2	185,6	2,24
todos-s-usos	11	2	131,3	-0,05
todos-usos	11	3	197,4	2,88
todos-usos-grid	10	3	210,4	3,29
todos-usos-host	11	1	75,2	-2,54
Global	264		132,5	

7.5.3 Análise do strength dos critérios

Na análise do *strength* dos critérios utilizamos o método de análise multivariada chamado análise de agrupamento. Ele tem por objetivo agrupar os elementos em grupos com base em suas similaridades. Desse modo, é possível analisar as relações entre os critérios definidos e examinar as relações de inclusão dentre os critérios, permitindo responder a terceira hipótese definida. Nesse método é usado o dendrograma para visualizar as relações de inclusão e as diferenças entre os critérios. Para a construção dos dendrogramas foi utilizado como base o valor médio das coberturas de cada critério apresentadas nas Tabelas de 18 a 28. No dendrograma, o eixo vertical indica a diferença entre os critérios, dessa forma, para que haja uma relação de inclusão, os critérios precisam estar no mesmo grupo e no nível zero do eixo vertical.

Alguns critérios possuem uma clara relação de inclusão entre eles. Com isso, o dendrograma gerado é idêntico e por isso não foram incluídos aqui. Essa relação de inclusão aparece em todos os dendrogramas listados, logo, eles não serão comentados em cada figura. Essas relações de inclusão são: todos-nos e todos-nos-grid; todos-nos-sinc e todos-nos-sinc-grid; todas-arestas e todas-arestas-grid; todos-c-usos e todos-c-usos-grid; todos-p-usos e todos-p-usos-grid; e todos-usos e todos-usos-grid. Os dendrogramas dos critérios todos-c-usos-host, todos-p-usos-host são idênticos ao do critério todos-usos-host e por isso também foram omitidos.

Excetuando as inclusões citadas, foram identificadas onze relações diferentes de inclusão de critérios nos dendrogramas nas Figuras 55 e 72. A primeira relação de inclusão possui os critérios todas-arestas-sinc, todos-nos-sinc-host, todos-nos-sinc-grid, todos-nos-grid e todos-nos-sinc, identificado nas Figuras 55, 62, 68 e 69.

A segunda relação de inclusão são dos critérios todos-s-c-usos e todos-global-c-usos-grid. Ele pode ser observado em todos os dendrogramas, com exceção do relacionado ao *strength* do conjunto de casos de teste do critério todos-blocos-p-usos-grid. Esses critérios apresentaram uma forte relação devido ao todos-s-c-usos considerar a comunicação e o uso computacional das variáveis compartilhadas em nível global.

A terceira relação de inclusão são dos critérios todos-c-usos-host e todos-usos-host. Ela ocorreu em quase todos os dendrogramas, podendo ser observado nas Figuras 55, 56, 59, 60, 62, 63, 64, 65, 66, 67, 68, 69, 71 e 72.

Nos dendrogramas das Figuras 56, 58, 60 e 65 foi identificada a relação de inclusão dos critérios todos-nos-sinc, todos-nos-sinc-grid e todas-arestas-sinc.

Os critérios todos-nos-host e todos-nos-sinc-host possuem indícios de inclusão no dendrograma da Figura 56 relacionado ao conjunto de casos de teste do critério todos-nos-host.

Os critérios todos-s-p-usos e todos-global-p-usos-grid possuem relação de inclusão

apresentada quase todos os dendrogramas. Essa inclusão ocorre devido ao todos-s-p-usos incluir a comunicação e o uso predicativo de variável compartilhada em nível global. Essa relação pode ser observada nas Figuras 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70 e 72.

Outra inclusão identificada nos dendrogramas das Figuras 57 e 61 possui os critérios todas-arestas-sinc, todos-nos-sinc-host, todos-nos-sinc e todos-nos-sinc-grid.

A relação de inclusão dos critérios todas-arestas-host e todos-p-usos-host pode ser identificada nas Figuras 57, 58 e 61, relacionados aos conjuntos de casos de teste dos critérios todos-nos-sinc-grid, todos-nos-sinc-host e todas-arestas-sinc, respectivamente.

Foi identificada a relação de inclusão dos critérios todos-p-usos-host, todas-arestas-sinc, todas-arestas-host, todos-nos-sinc-host, todos-nos-sinc-grid, todos-nos-host e todos-nos-sinc. Essa relação pode ser observada nas Figuras 59, 63, 64, 66, 67, 71 e 72.

Os dendrogramas do conjunto de casos de teste dos critérios todos-nos-sinc-grid (Figura 57) e todas-arestas-sinc (Figura 59) apresentam a relação de inclusão dos critérios todos-p-usos-host, todas-arestas-host, todos-nos-host e todos-nos-sinc-host.

Por último, no dendrograma do critério todos-bloco-p-usos-grid na Figura 70 foi identificada a relação de inclusão dos critérios todos-usos-host, todos-p-usos-host, todos-c-usos-host, todas-arestas-sinc, todas-arestas-host, todos-nos-sinc-host, todos-nos-sinc-grid e todos-nos-sinc.

Diversos critérios relacionados ao *host* apresentaram relação de inclusão nos dendrogramas. Isso pode ser justificado pelo nível de complexidade do código do *host* apresentado nos *benchmarks*, que focam na resolução do problema dentro do *kernel* do programa.

Com base nos resultados das relações de inclusão apresentadas pela análise de agrupamento utilizando os dendrogramas, a hipótese alternativa (HA_3) é aceita, pois ao menos um dos critérios de teste estrutural para programas concorrentes CUDA apresentou uma relação de inclusão com outro critério.

Figura 55 – Dendrograma para o *strength* do critério todos-nos-grid.

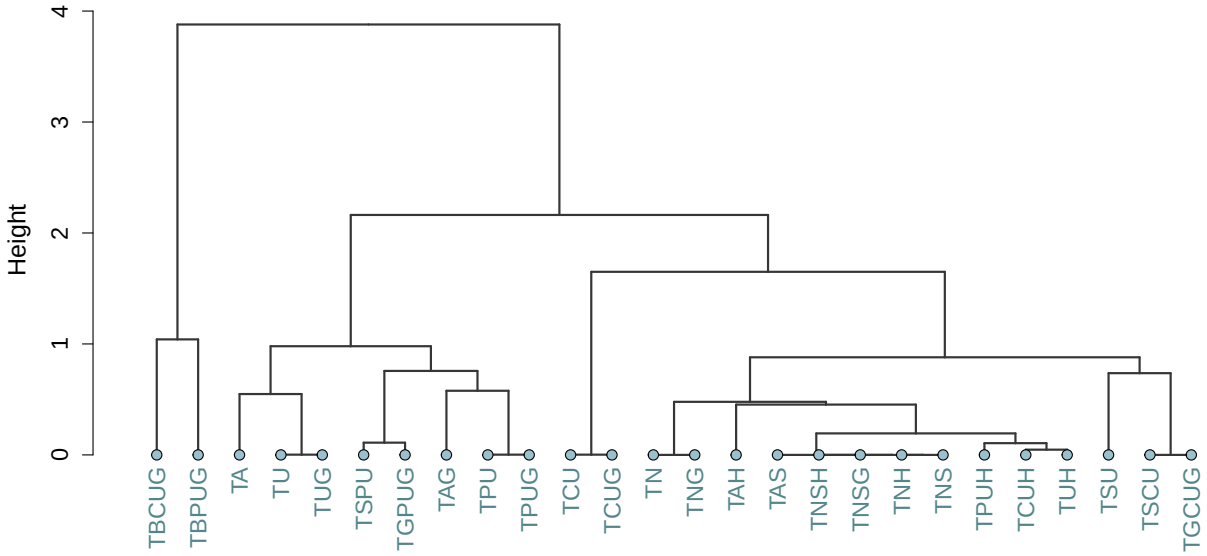


Figura 56 – Dendrograma para o *strength* do critério todos-nos-host.

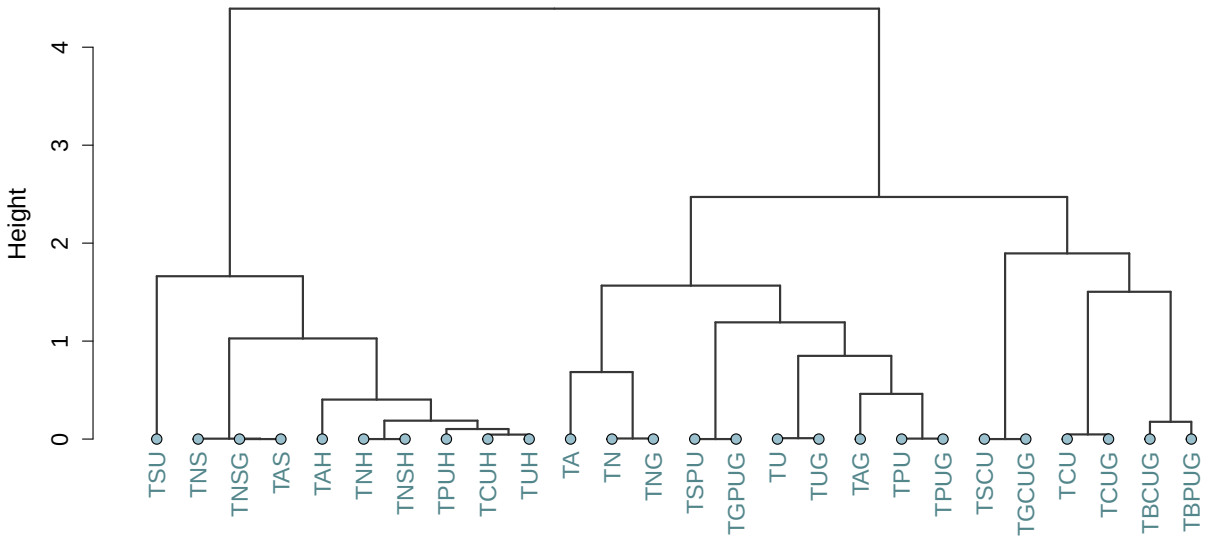


Figura 57 – Dendrograma para o *strength* do critério todos-nos-sinc-grid.

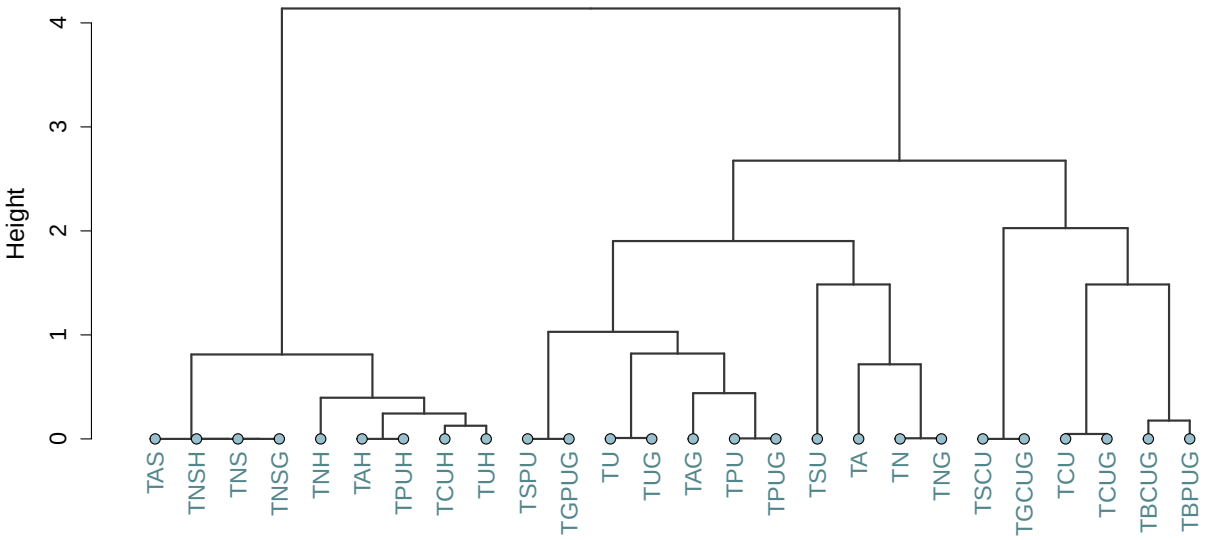


Figura 58 – Dendrograma para o *strength* do critério todos-nos-sinc-host.

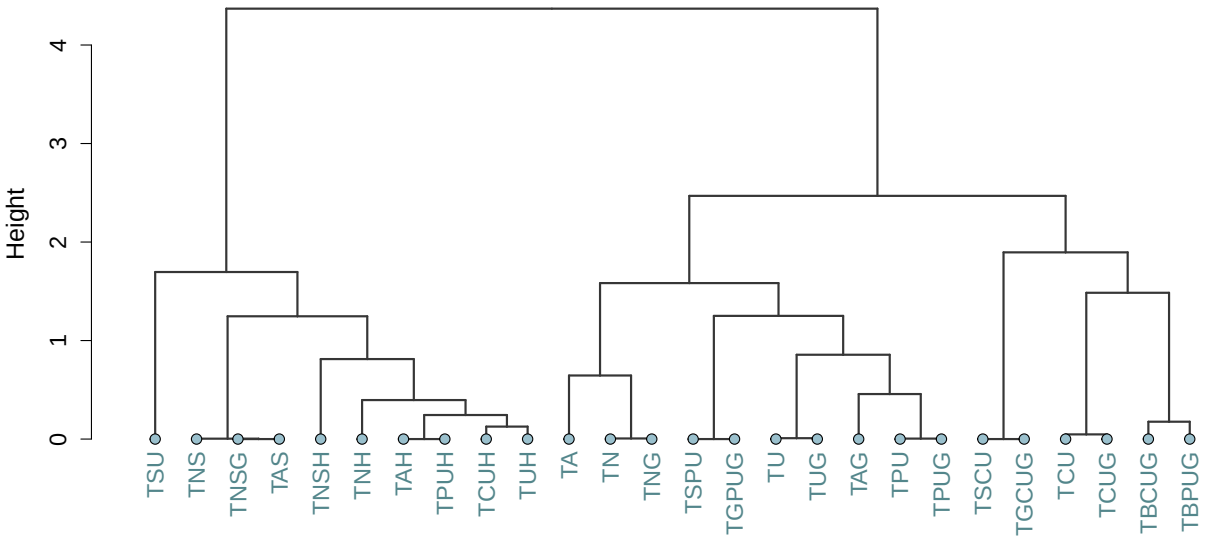


Figura 59 – Dendrograma para o *strength* do critério todas-arestas-grid.

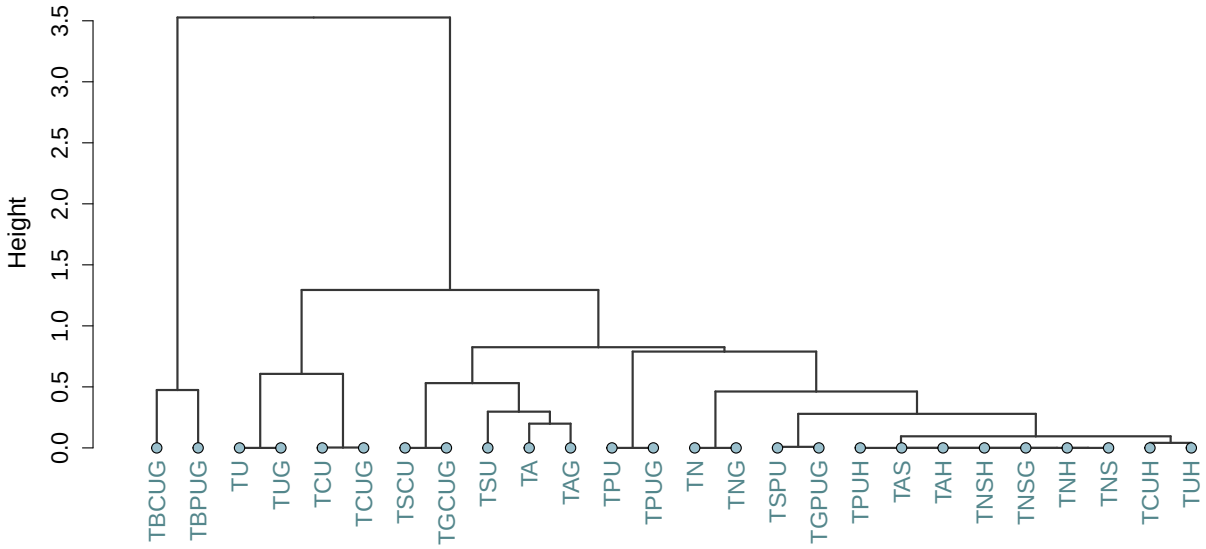


Figura 60 – Dendrograma para o *strength* do critério todas-arestas-host.

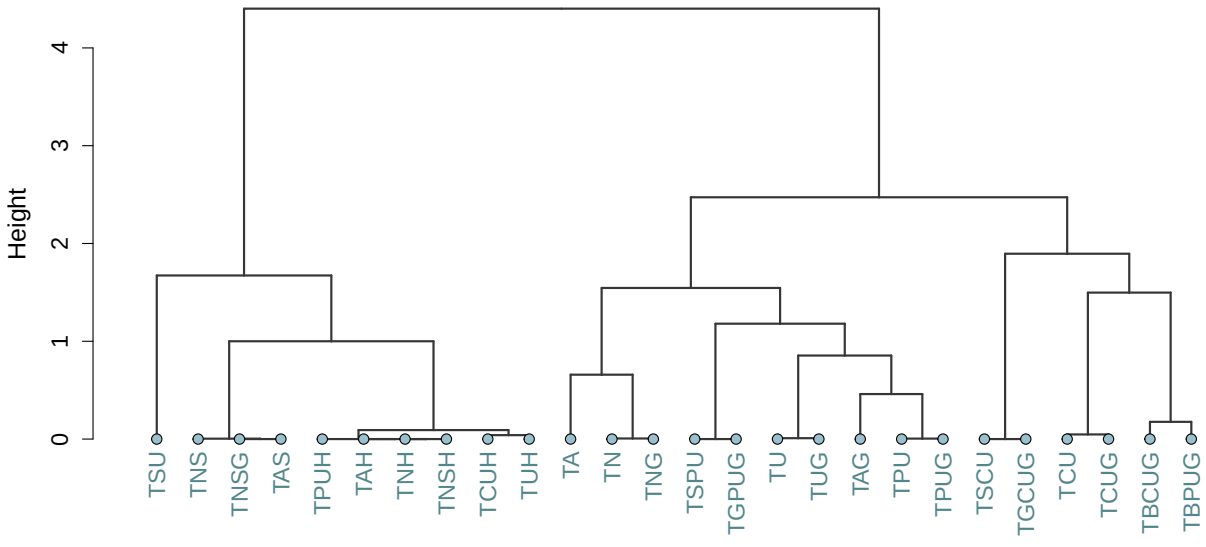


Figura 61 – Dendrograma para o *strength* do critério todas-arestas-sinc.

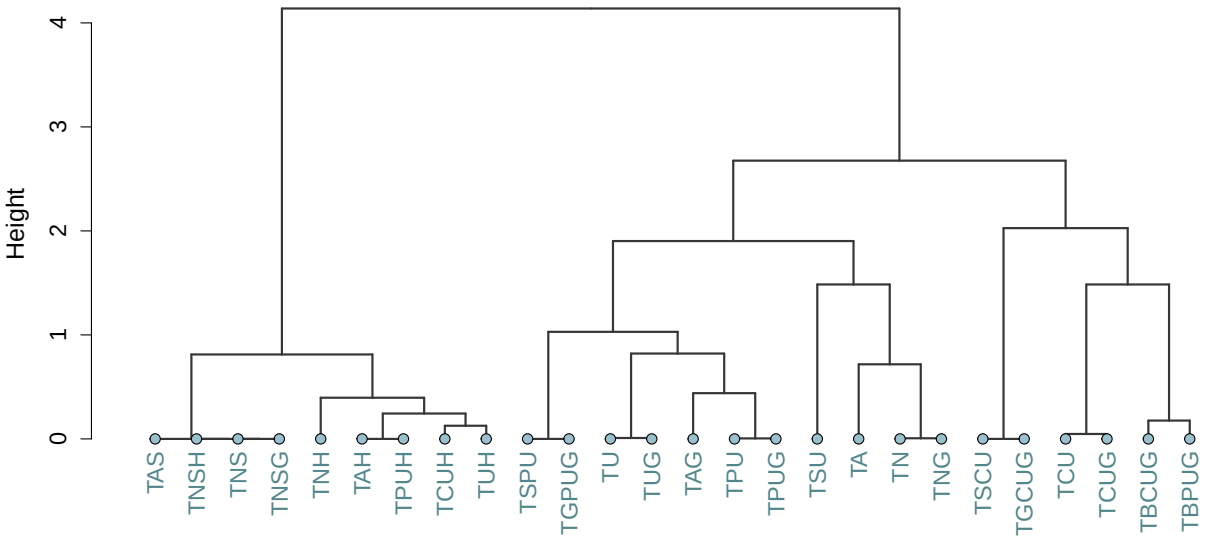


Figura 62 – Dendrograma para o *strength* do critério todos-c-usos-grid.

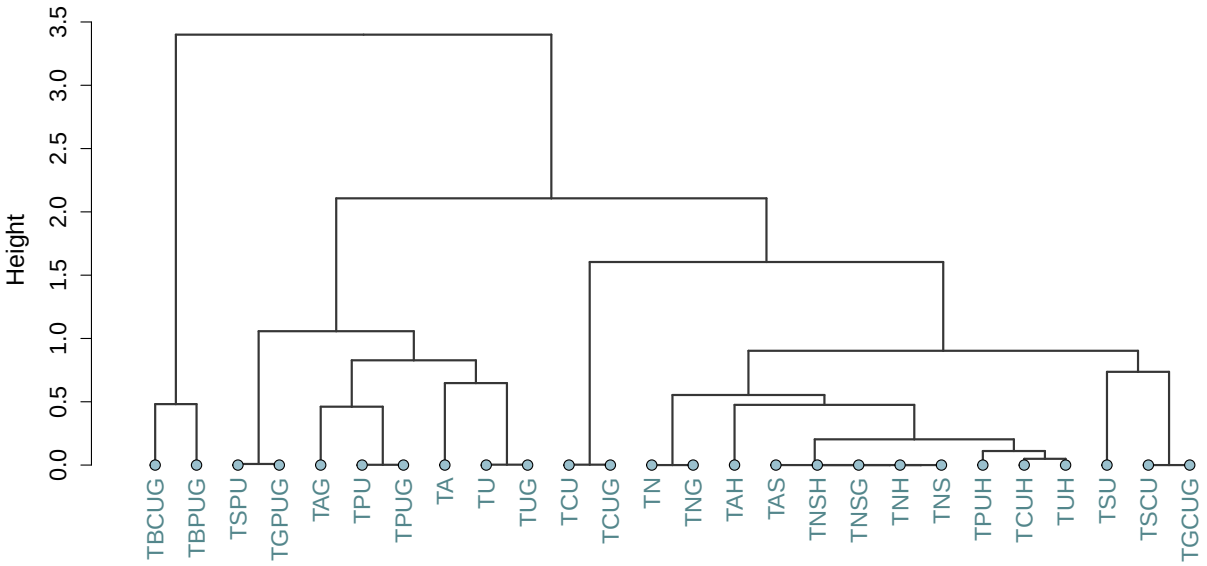


Figura 63 – Dendrograma para o *strength* do critério todos-p-usos-grid.

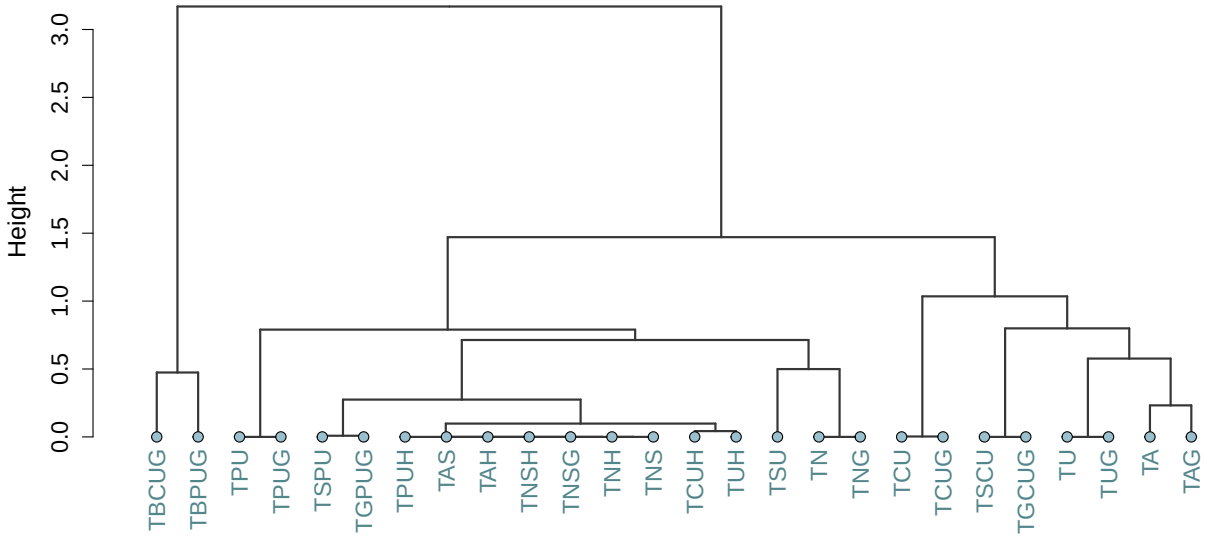


Figura 64 – Dendrograma para o *strength* do critério todos-usos-grid.

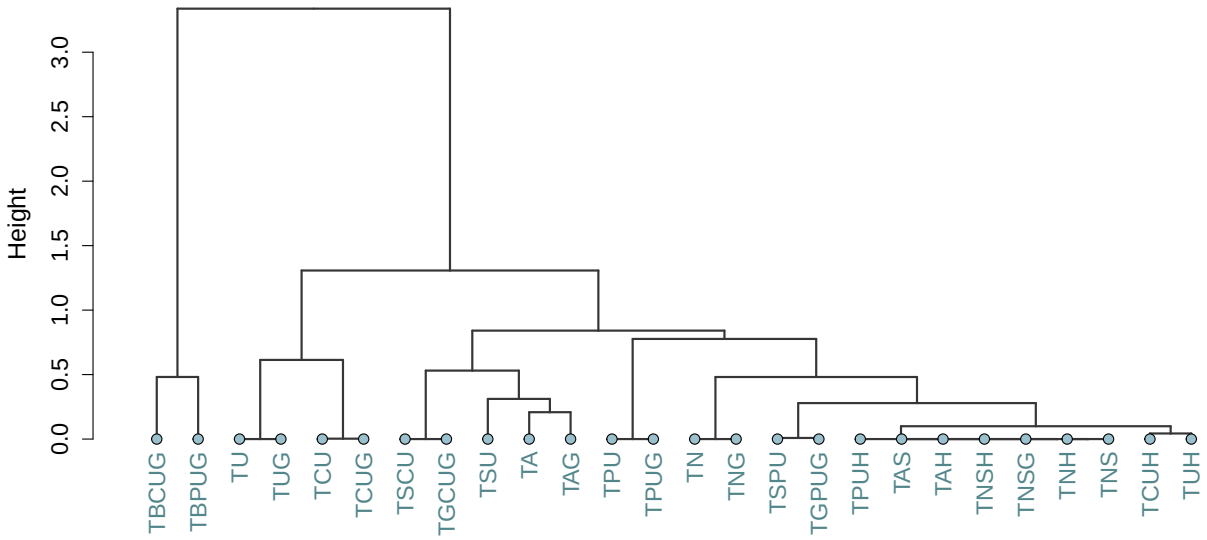


Figura 65 – Dendrograma para o *strength* do critério todos-usos-host.

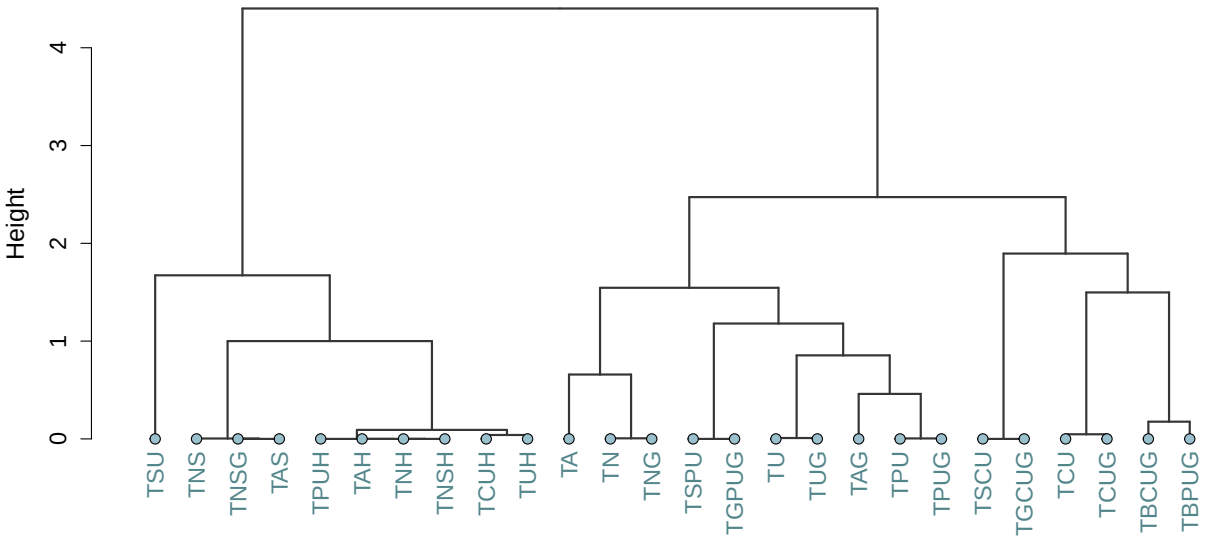


Figura 66 – Dendrograma para o *strength* do critério todos-s-c-usos.

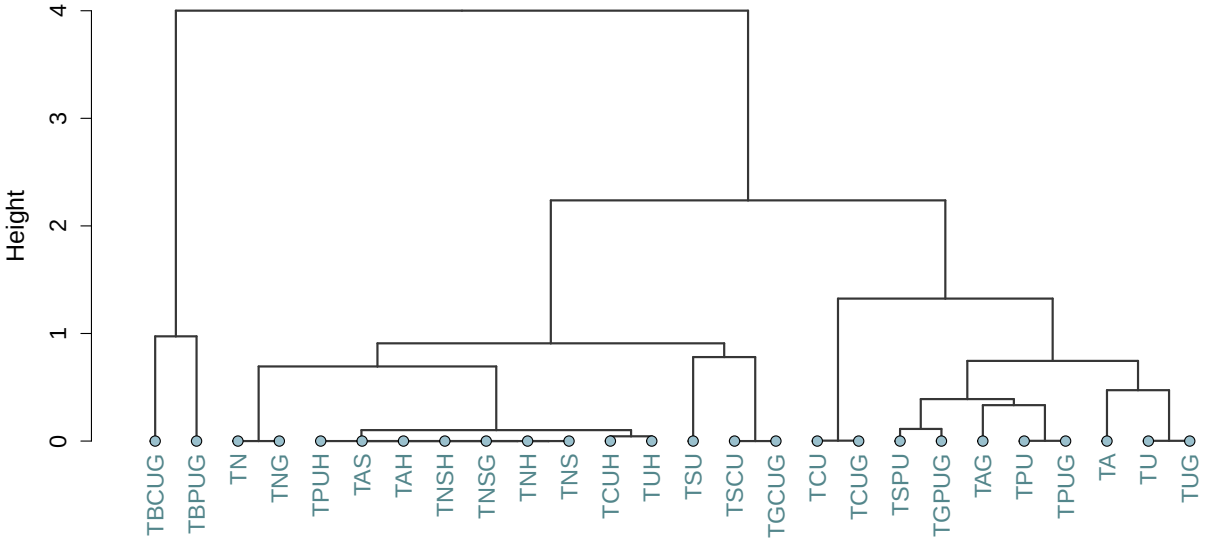


Figura 67 – Dendrograma para o *strength* do critério todos-s-p-usos.

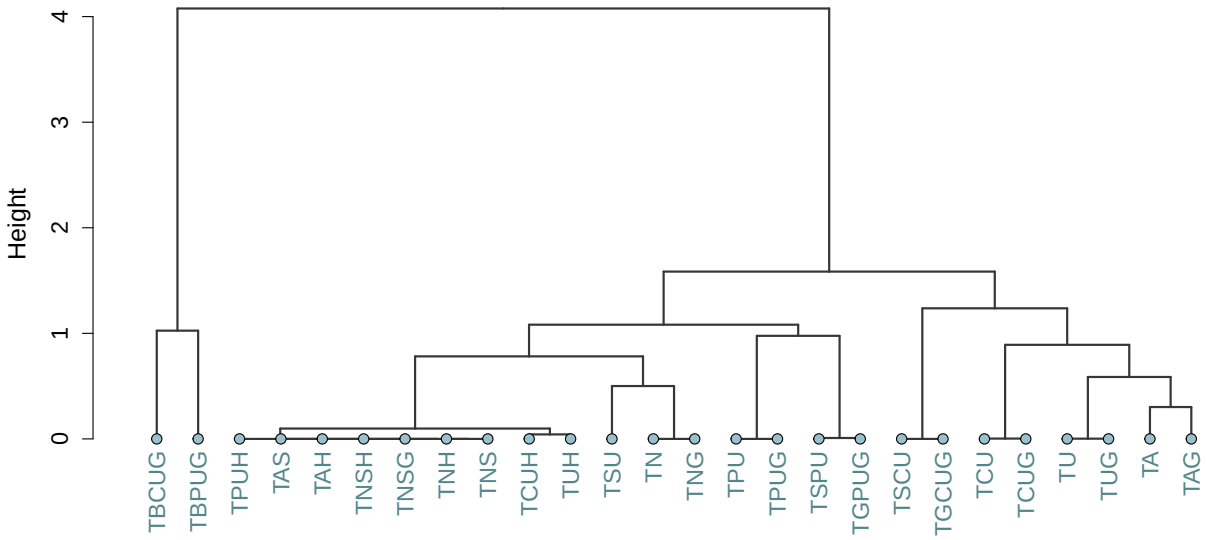


Figura 68 – Dendrograma para o *strength* do critério todos-s-usos.

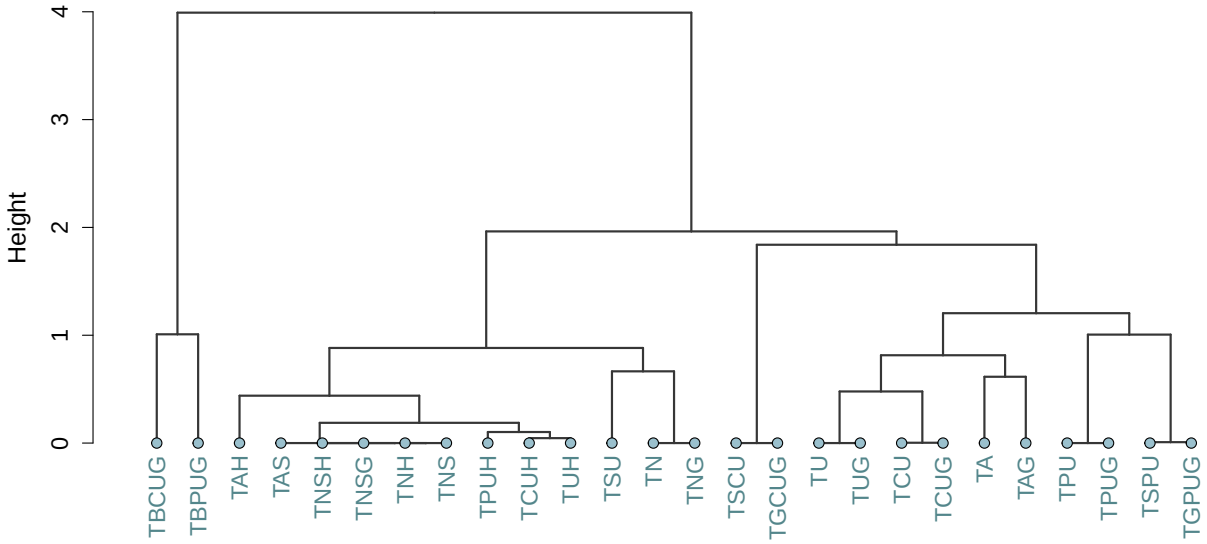


Figura 69 – Dendrograma para o *strength* do critério todos-bloco-c-usos-grid.

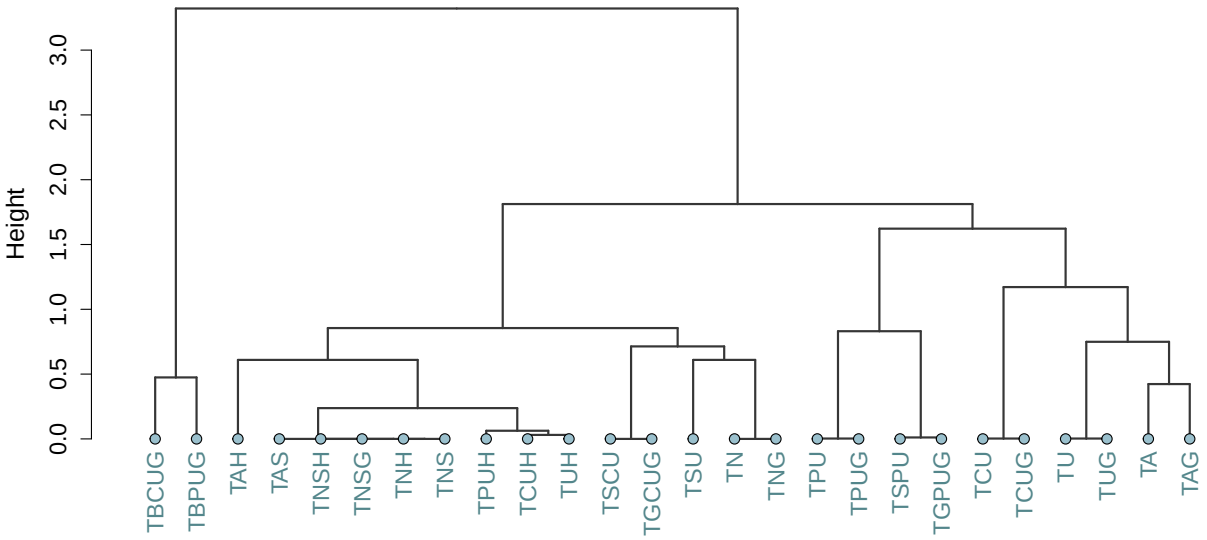


Figura 70 – Dendrograma para o *strength* do critério todos-bloco-p-usos-grid.

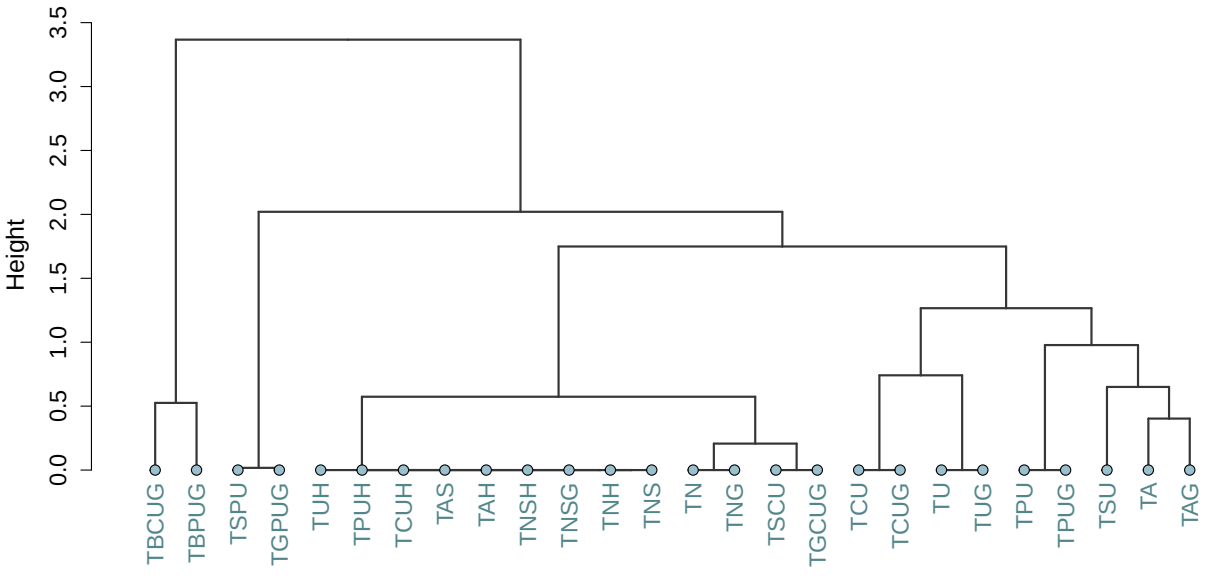


Figura 71 – Dendrograma para o *strength* do critério todos-global-c-usos-grid.

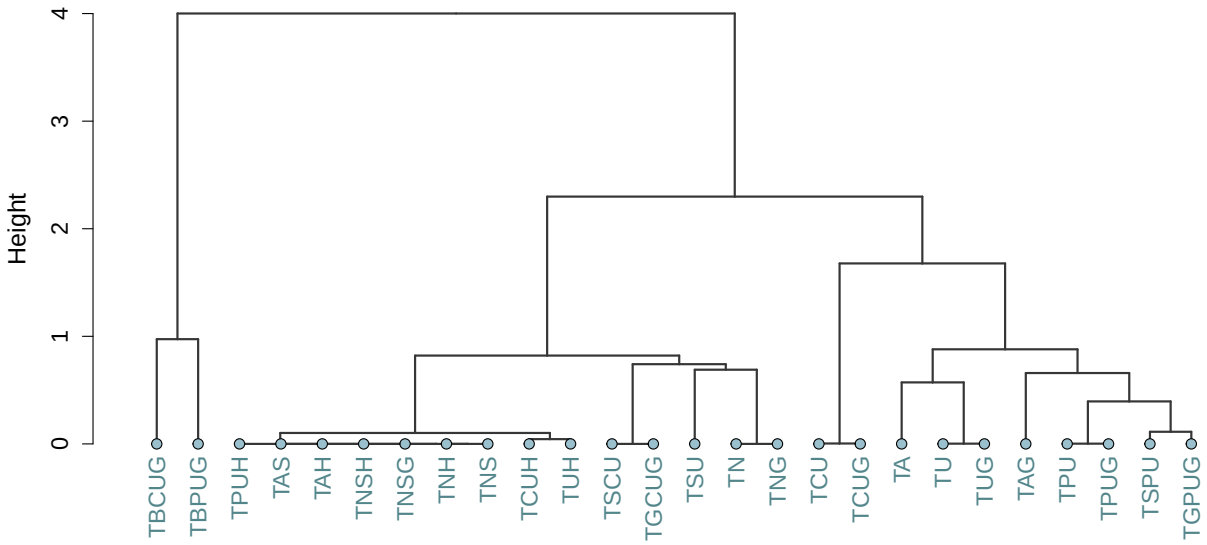
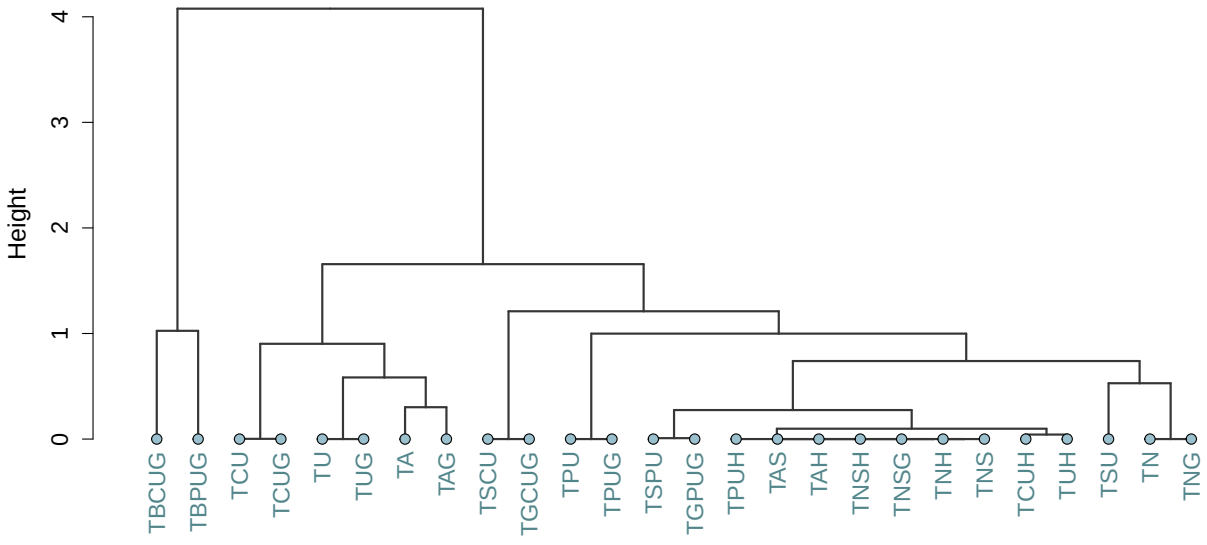


Figura 72 – Dendrograma para o *strength* do critério todos-global-p-usos-grid.



7.6 Conclusões Obtidas com Base nos Resultados

O resultado de custo dos critérios de teste para os elementos requeridos demonstrou que o custo para a atividade de teste está intrinsecamente ligada ao número de *threads* que o programa possui, sendo um fator multiplicador. O *benchmark* "soma_matriz" apresentou um custo de elementos requeridos superior aos outros *benchmarks* devido à quantidade de *threads*. O número de elementos requeridos apresentados no código fonte também influenciam no custo, como número de nós e arestas. Laços de repetição e estruturas de decisão que restringem a entrada de *threads* implicam na quantidade de elementos requeridos não executáveis. O resultado ainda mostrou que os critérios de teste relacionados ao *host* possuem um menor custo, devido a não considerar as características do *grid* e não ter o número de *threads* como um multiplicador de custo. Os critérios de maior custo são os relacionados aos elementos requeridos do *grid*, dentre eles todos-usos e todos-usos-grid, estes apresentando o maior custo entre os critérios de teste, seguidos por todos-c-usos e todos-c-usos-grid. O critério todas-arestas apresentou um menor número de elementos requeridos em relação ao critério todos-nos, devido ao programa ValiElem considerar apenas as arestas essenciais na geração dos elementos requeridos. Para os elementos requeridos não executáveis, os critérios relacionados à definição e uso de variáveis apresentou maior custo, sendo novamente os critérios todos-usos e todos-usos-grid com maior custo, seguido do critério todos-c-usos e todos-c-usos-grid.

A eficácia dos critérios demonstrou ter uma relação com o custo apresentado por eles. O critério todos-usos apresentou a maior mediana de eficácia, 91%, como mostrado na Figura 54. Entretanto, critérios com um custo menor apresentaram eficácia próxima, como os critérios todos-c-usos, todos-c-usos-grid, todos-p-usos e todos-p-usos-grid. Os critérios todos-blocos-c-usos-grid, todos-blocos-p-usos-grid, todos-global-c-usos-grid e todos-global-p-usos-grid apresentaram um custo muito menor que o critério todos-usos, mas com uma eficácia menor. Isso foi devido a alguns *benchmarks* não possuírem elementos requeridos para esses critérios. Entretanto, considerando apenas os programas que possuem esses critérios, a eficácia se igualou ou ficou próxima ao todos-usos. Ainda, utilizando os quatro critérios, eles apresentaram a mesma eficácia do todos-usos, mas com um terço do custo de elementos requeridos. Esses critérios conseguiram revelar todos os defeitos relacionados ao erro de memória, erro de variável compartilhada, ID incorreto da *thread* e uso incorreto de variável global inseridos nos *benchmarks*. Para erros do tipo falta de sincronização eles foram capazes de revelar 61,5% dos defeitos, apesar da ferramenta não permitir a execução determinística das *threads* do *grid*.

A análise do *strength* demonstrou a relação de inclusão de alguns critérios. O critério todos-s-c-usos demonstrou inclusão do critério todos-global-c-usos-grid, devido a ele considerar a comunicação e o uso computacional de variável global. O mesmo ocorreu com os critérios todos-s-p-usos e todos-global-p-usos-grid. Critérios relacionados ao *host*

demonstraram inclusão entre si devido aos *benchmarks* focarem a resolução dos problemas no código do *kernel* do programa. Os critérios todos-blocos-c-usos-grid, todos-blocos-p-usos-grid, todos-global-c-usos-grid e todos-global-p-usos-grid não demonstraram relação de inclusão, corroborando com a utilização combinada desses critérios, conforme a eficácia apresentada na Tabela 17.

Os experimentos conseguiram atender aos objetivos do trabalho. Os critérios de teste propostos apresentaram a capacidade de auxiliar o testador a revelar os defeitos inseridos nos programas CUDA considerados. Demonstramos as diferenças entre os critérios em relação aos seus custos, eficácia e *strength* para revelar diferentes tipos de defeitos, podendo auxiliar no processo de escolha dos critérios para o teste de programas CUDA. Neste processo, o testador pode iniciar o teste com critérios de menor custo, como o critério todos-global-p-usos-grid e, se necessário, aplicar critérios mais eficazes, como o todos-usos.

7.7 Considerações finais

Apresentamos neste capítulo a metodologia aplicada aos experimentos realizados. Mostramos o conjunto de *benchmarks* utilizados na avaliação, relacionando as características de cada programa. Utilizando três métricas de avaliação, demonstramos o custo, a eficácia e o *strength* relacionados a cada critério para revelar determinados tipos de defeitos para programas com diferentes características, que pode auxiliar o testador a escolher o melhor critério de teste para programas CUDA que se aplique ao seu cenário. Por fim, analisamos três hipóteses relacionadas às métricas de avaliação, confirmando as diferenças entre os critérios de teste propostos.

8 TRABALHOS RELACIONADOS

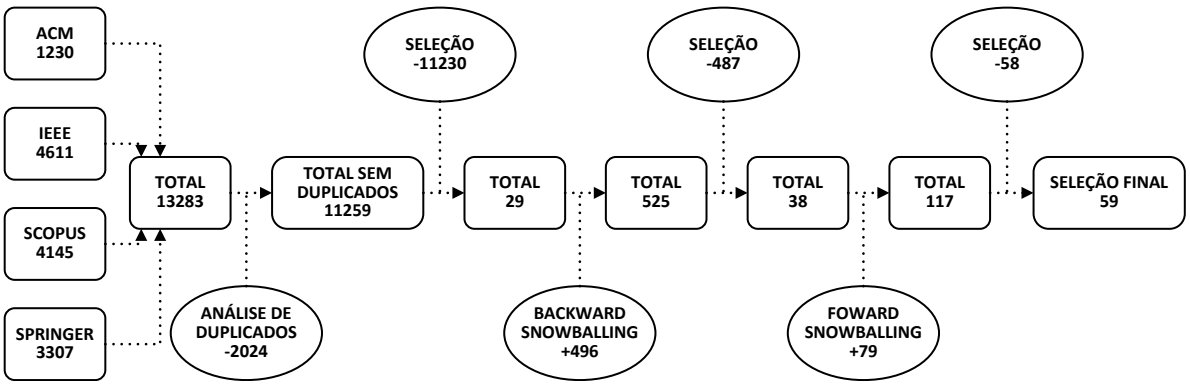
Foi realizada uma revisão sistemática para identificar os trabalhos relacionados ao teste de programas para GPU. A revisão seguiu o processo definido por Kitchenham e Charters (2007), utilizando o termo de busca apresentado na Tabela 37. Ele foi aplicado em quatro banco de dados de pesquisa: ACM, IEEE Xplore, Scopus e Springer.

Tabela 37 – Termos de busca.

(cuda OR opencl OR gpu) AND (test OR testing OR analysis OR validation OR debug OR verification OR replay OR coverage OR checking)

A pesquisa retornou ao todo 13283 trabalhos. No processo de seleção foram removidos os artigos duplicados e, posteriormente, foi realizada a seleção com base na análise do título, resumo, palavras chave, introdução, conclusão e leitura completa do artigo, se necessário. Ao final do processo foram selecionados 59 artigos, conforme ilustrado na Figura 73. Desse total, diversos artigos abordam aspectos das mesmas ferramentas, totalizando 30 ferramentas diferentes.

Figura 73 – Processo de seleção da revisão sistemática.



A Tabela 38 apresenta as ferramentas de teste encontradas e os modelos de programação suportados em cada uma. As ferramentas GPUVerify, GPURepair e Sorcar suportam ambos os modelos CUDA e OpenCL. ACC_TEST é a única ferramenta que suporta programação híbrida com OpenACC e MPI. A Maior parte das ferramentas suportam apenas CUDA, e sete ferramentas suportam o OpenCL, e apenas dois suportam o OpenACC (ACC_TEST e PCAST). Alguns artigos citam que a ferramenta pode ser facilmente estendida para outros modelos de programação, mas nenhum trabalho foi feito para atingir isso até o momento da pesquisa.

Das ferramentas encontradas, somente dois apresentaram a aplicação de teste funcional. Considerando a relevância do teste funcional e suas características para testar

Tabela 38 – Modelo de programação coberto para cada ferramenta.

Ferramenta	**Modelo prog.**	**Ferramenta**	**Modelo prog.**
ACC_TEST	OpenACC MPI	**Grace/GMRace**	CUDA
Alur	CUDA	**KLEE-CL**	OpenCL
auCS	CUDA	**LD**	CUDA
AutoSync	CUDA	**mutGPU**	CUDA
BARRACUDA	CUDA	**PCAST**	OpenACC
Boyer	CUDA	**PUG**	CUDA
CLFuzz	OpenCL	**Sarkar**	CUDA
CLTestCheck	OpenCL	**SESA**	CUDA
CUDA au Coq	CUDA	**Simulee**	CUDA
Cudagrind	CUDA	**Sorcar**	CUDA OpenCL
CURD	CUDA	**Test Amplification**	CUDA
ESBMC-GPU	CUDA	**Vercors**	OpenCL
GKLEE	CUDA	**Vericuda**	CUDA
GPURepair	CUDA OpenCL	**WEFT**	CUDA
GPUVerify	CUDA OpenCL	**Xing**	CUDA

programas como uma caixa preta, era esperado encontrar mais iniciativas usando essa técnica. Entre as ferramentas, o KLEE-CL (COLLINGBOURNE; CADAR; KELLY, 2012; COLLINGBOURNE; CADAR; KELLY, 2014) usou o teste funcional para verificar a corretude de programas OpenCL e comparando com a saída de uma versão C/C++ sequencial supostamente bem testada. PCAST (AHMAD; WOLFE, 2018) aplicou um conceito semelhante, mas comparando os resultados intermediários das execuções dos *kernels* de programas OpenACC com partes iguais de uma versão para CPU para revelar defeitos.

Quinze ferramentas utilizaram o código fonte do programa para detectar defeitos. Boyer, Skadron e Weimer (2008) apresentou a primeira ferramenta para teste de aplicações CUDA para revelar condições de disputa em memória compartilhada, ao executar o programa em modo de emulação e analisar os dados gerados buscando por erros. Grace/GMRace busca detectar condições de disputa inter-warp e intra-warp em memória compartilhada usando análise estática e análise dinâmica (ZHENG *et al.*, 2011; ZHENG *et al.*, 2014). Test Amplification busca detectar condições de disputa executando o programa, e usando análise estática para ampliar o alcance do teste (LEUNG *et al.*, 2012). GPUVerify também verifica condições de disputa utilizando análise estática (BETTS *et al.*, 2012b; CHONG *et al.*, 2013; COLLINGBOURNE *et al.*, 2013; BARDSLEY; DONALDSON;

WICKERSON, 2014; CHONG; DONALDSON; KETEMA, 2014; BARDSLEY *et al.*, 2014; BARDSLEY; DONALDSON, 2014; BETTS *et al.*, 2015; DONALDSON *et al.*, 2017; BETTS *et al.*, 2018). LD verifica condições de disputa em memória compartilhada sem realizar a instrumentação do código, usando verificação baseada em valor (*value-based checking*) (LI; DING, 2014) (LI *et al.*, 2017) . BARRACUDA busca por condições de disputa inter-bloco e intra-bloco na memória compartilhada e memória global (EIZENBERG *et al.*, 2017). Ele modela e a execução do programa como um rastro para identificar as corridas. Diferente de outras soluções, ele suporta operações atômicas e cercas de memória (*memory fence*). CURD utiliza a ferramenta BARRACUDA como base, mas oferece um menor *overhead* na execução (PENG; GROVER; DEVIETTI, 2018). Cudagrind utiliza teste estrutural para encontrar erros de memoria (BAUMANN; GRACIA, 2014). ACC_TEST é a única ferramenta que busca testar aplicações híbridas, usando MPI e OpenACC em busca de condições de disputa, *deadlock* e erros de memória (ALGHAMDI; EASSA, 2019; ALGHAMDI *et al.*, 2020; EASSA *et al.*, 2020). Sarkar utiliza análise estática para detectar condições de disputa em CUDA (SARKAR *et al.*, 2018). Alur, Devietti e Singhania (2018) buscaram verificar por erros relacionados a configuração do tamanho dos blocos de *threads*. GPURepair (JOSHI; MUDUGANTI, 2021), AutoSync (ANAND; POLIKARPOVA, 2018) e auCS (WU *et al.*, 2019) utilizaram a ferramenta GPUVerify como um oráculo de teste para identificar possíveis locais de condição de disputa para automaticamente implementar barreiras de sincronização, enquanto evita causar divergência de barreira. CLFuzz (PENG; RAJAN, 2020) usa teste baseado em mutação para gerar dados de entrada para *kernels* OpenCL, enquanto usa a ferramenta CLTestCheck para medir a cobertura de arestas atingidas pelos casos de teste e buscando revelar condições de disputa (PENG; RAJAN, 2019).

Algumas das ferramentas utilizaram verificação formal para revelar defeitos. PUG foi a primeira ferramenta que busca relevar condições de disputa por meio de verificação formal. Ele utiliza Teorias do Módulo da Satisfação (*Satisfiability Modulo Theories* - SMT), que utiliza fórmulas lógicas para representar o programa (LI; GOPALAKRISHNAN, 2010). O programa é analisado usando valores simbólicos com um solucionador SMT, como o Microsoft Z3. No SMT, ele usa análise simbólica em *kernels* individuais buscando por condições de disputa em memória compartilhada. GKLEE analisa o programa completo por condições de disputa na memória global e compartilhada, por *deadlocks*, e a omissão de qualificador *volatile* usando execução simbólica e execução concreta (execução concólica) (LI; LI; GOPALAKRISHNAN, 2012; LI; GOPALAKRISHNAN, 2012; LI *et al.*, 2012; CHIANG *et al.*, 2013). Eles ainda propõem um teste de cobertura considerando granularidade de *thread*, buscando garantir que eles executaram cada linha de código. Um critério similar ao todos-nós. A ferramenta SESA também utiliza execução simbólica (LI; LI; GOPALAKRISHNAN, 2014). Ele busca melhorar o suporte a *kernel CUDA* e aumentar o desempenho em *kernels* com milhares de *threads*. SESA busca identificar automaticamente

as entradas que devem ter valores concretos, para melhorar o desempenho sem diminuir a cobertura em comparação com o GKLEE. Vercors também utiliza o SMT para verificar programas em OpenCL por condições de disputa, e tendo suporte a operações atômicas (HUISMAN; MIHELCIC, 2013; BLOM; HUISMAN, 2014; BLOM *et al.*, 2017; BLOM; HUISMAN; MIHELCIC, 2014; AMIGHI *et al.*, 2015; HUISMAN *et al.*, 2018; SAFARI *et al.*, 2020). Entretanto ele verifica apenas o *kernel*, não verificando o código do *host*. ESBMC-GPU detecta condições de disputa e erros de memória usando SMT (PEREIRA *et al.*, 2016) . Sorcar utiliza SMT para detectar condições de disputa (NEIDER *et al.*, 2019). Xing *et al.* (2018) também utiliza SMT para detectar condições de disputa, mas modela o programa com base no código em nível PTX (*assembly*) ao invés do código CUDA original. Vericuda utiliza SMT para verificar a correção funcional do programa (KOJIMA; IMANISHI; IGARASHI, 2018). CUDA au Coq (FERRELL; DUAN; HAMLEN, 2019) realiza a validação de programas CUDA em nível de *assembly* utilizando verificação formal por meio do provador de teorema Coq (BERTOT; CASTÉRAN, 2013). Como uma característica das ferramentas de verificação que utilizam SMT, esses trabalhos sofrem de diferentes graus de falsos-positivos e normalmente suportam apenas *kernels* pequenos, devido ao problema de explosão do espaço de estado.

WEFT utiliza teste baseado em modelo para detectar condição de disputa e *deadlock* em programas CUDA que possuam *kernels* com *warps* especializados, onde diferentes computações são designadas para cada *warp* de um mesmo bloco (SHARMA; BAUER; AIKEN, 2015). Simulee utiliza teste baseado em modelo em conjunto com programação evolutiva. Ele gera os dados de teste para um modelo de acesso de memória, simulando o ambiente de execução para detectar condições de disputa por meio das informações relacionadas ao acesso de memória (WU *et al.*, 2020).

Em relação ao teste baseado em defeitos, apenas duas ferramentas foram encontradas. CLTestCheck busca medir a qualidade do teste de programas OpenCL por meio da cobertura de código (PENG; RAJAN, 2019). Ele insere defeitos nos *kernels* e busca verificar se os casos de teste foram capazes de revelar os defeitos inseridos. A segunda ferramenta foi desenvolvida para avaliar se a programação para GPU pode se beneficiar do teste de mutação. MutGPU aplica o teste de mutação em programas CUDA, propondo nove operações de mutação para CUDA que podem ajudar a melhorar a qualidade dos casos de teste Zhu e Zaidman (2020). Esses operadores foram apresentados no Capítulo 4.

A maior parte dessas ferramentas focam em condições de disputa, com algumas cobrindo *deadlock*, erros de memória e outros tipos de defeitos. Para isso eles utilizaram teste funcional, teste estrutural, verificação formal, teste baseado em modelo e teste baseado em defeitos para revelar defeitos em programas que utilizam o modelo de programação CUDA, OpenCL e/ou OpenACC. As ferramentas encontradas cobrem parte das funcionalidades dos modelos de programação, limitando o escopo do teste e os defeitos revelados. Somente

dois estudos abordaram teste baseado em defeito para OpenCL e CUDA. Apenas duas utilizaram teste funcional. Especificamente para OpenACC, nenhum estudo abordou teste baseado em modelo ou baseado em defeitos. Para CUDA, OpenCL e OpenACC, a maior parte das ferramentas não propôs e cobriu nenhum critério de teste estrutural para programas para GPU.

9 CONCLUSÕES

9.1 Caracterização da Pesquisa

Os modelos de programação paralela para GPU carecem de modelo e critérios de teste estruturais para auxiliar na atividade de teste. Isso dificulta a realização da atividade de teste para revelar defeitos e, por consequência, determinar se um programa foi bem testado. Nos trabalhos relacionados houve poucos critérios de teste propostos.

Este trabalho teve como principal objetivo auxiliar na atividade de teste de aplicações CUDA por meio da proposta de critérios de teste estruturais. Nós propusemos um modelo de teste que caracteriza o fluxo de controle e fluxo de dados de aplicações CUDA e representa o programa por meio de um GFCP. Com base no modelo de teste, propusemos novos critérios de teste que busquem revelar defeitos encontrados em aplicações CUDA e que consideram os aspectos de sincronização, comunicação e uso de memória compartilhada por milhares de *threads* CUDA.

Para realizar a validação, o modelo e critérios de teste foram implementados na ferramenta ValiCUDA. Foi implementado um instrumentador para programas CUDA utilizando o *framework* Clava, por meio do desenvolvimento do *script* ClavaCUDA, que instrumenta o código para a geração dos rastros de execução feitas pelo *host* e pelas *threads* de cada *grid* do programa. Ele gera os GFCs com as informações do fluxo de controle, fluxo de dados e comunicação do *host* e *grid*. Com base nas informações extraídas foram feitas modificações na ferramenta ValiMPI. Foi modificado o módulo ValiElem para gerar os elementos requeridos dos critérios de teste propostos e na ValiEval para avaliar a cobertura dos critérios de teste considerando a execução de todas as *threads* do programa.

Utilizando a ferramenta desenvolvida, foi feita a validação do modelo e critérios por meio da execução de experimentos. Utilizamos um conjunto de 20 *benchmarks* com diversas características diferentes encontradas em programas CUDA, como estruturas de decisão e repetição, primitivas de sincronização, chamadas a múltiplos *kernels* e uso de memória compartilhada em nível global e em nível de bloco. Inserimos defeitos em cada *benchmark* com base na taxonomia apresentada no Capítulo 4 e utilizando operadores de mutação. Avaliando as métricas de custo, eficácia e *strength*, demonstramos que o modelo e critérios propostos conseguem revelar defeitos e podem auxiliar na realização da atividade de teste.

9.2 Principais Resultados e Contribuições

A principal contribuição está na proposta do modelo de teste e critérios de teste estrutural para aplicações CUDA. O modelo de teste captura as informações relacionadas

ao fluxo de controle, fluxo de dados e comunicação, com base nas características sequenciais e paralelas do modelo de programação CUDA. Os critérios de teste propostos buscam gerar elementos requeridos que revelem defeitos de sincronização e uso de dados entre *threads*, auxiliando na atividade de teste e ajudando a melhorar a qualidade de programas CUDA.

A ferramenta é outra contribuição deste trabalho. Implementamos o modelo e critérios de teste na ValiCUDA. Desenvolvendo um instrumentador utilizando o *framework* Clava, que realiza a instrumentação do código CUDA para gerar os rastros de execução das *threads* e extrai informações do fluxo de controle e fluxo de dados para a geração dos elementos requeridos. Nós modificamos a ValiMPI para adicionar suporte à geração de elementos requeridos de aplicações CUDA e para a avaliação de cobertura dos critérios de teste com base no rastro de execução de milhares de *threads*.

A realização do experimento, demonstrando a capacidade do modelo e critérios de teste é outra contribuição deste trabalho. O experimento demonstrou a capacidade do modelo e critérios de teste de revelar defeitos como erros de memória e falta de sincronização em programas com milhares de *threads*. A análise estatística realizada para validar os critérios demonstrou a efetividade destes para revelar defeitos. As avaliações utilizaram as métricas de custo, eficácia e *strength*, auxiliando o testador no planejamento da atividade de teste, provendo fundamentos para a seleção dos critérios de teste que melhor se adéquem ao seu cenário de teste.

Por fim, a disponibilização dos artefatos gerados. Disponibilizamos em um repositório no GitHub[1] os artefatos gerados durante a execução dos experimentos realizados. Futuramente a ferramenta de teste também será disponibilizada, permitindo que outros pesquisadores utilizem a ferramenta, repliquem os experimentos, implementem novos critérios e adicionem novos recursos à ferramenta.

Dessa forma, o trabalho atingiu os objetivos propostos. Foi definida uma relação de tipos de defeitos específicos para programas CUDA. Foi desenvolvido um modelo de teste estrutural que considera as características e elementos comuns que necessitam ser testados, como suporte a estruturas de decisão e repetição, o uso de barreiras de sincronização, a comunicação entre *host* e *grid*, o uso de variáveis compartilhadas em nível de bloco e global, e o suporte a milhares de *threads*. Foram desenvolvidos critérios de teste estruturais que auxiliam na melhoria dos casos de teste e contribuem na revelação de diversos tipos de defeitos, e diminuindo o custo da atividade de teste com base na escolha dos critérios de teste com relação ao menor custo e maior eficácia. Por fim, foi desenvolvido um protótipo de ferramenta de teste que implementa o modelo e critérios de teste propostos neste trabalho.

[1] https://github.com/Helderjfl/artefatos-ValiCUDA

9.3 Dificuldades Encontradas

Uma das dificuldades encontradas foi em relação à identificação dos trabalhos relacionados. Não encontramos nenhum mapeamento sistemático relacionado ao teste de programas concorrentes para GPU. Na realização da revisão, os termos de busca testados não restringiram a busca de forma a diminuir consideravelmente a quantidade de artigos retornados, sem deixar um dos artigos de controle de fora da seleção. Dessa forma, o processo de seleção foi oneroso. Além disso, a falta na literatura de uma taxonomia uniforme dos tipos de técnicas de teste dificultou o processo de definição das técnicas de teste utilizadas nos trabalhos relacionados.

Outra dificuldade foi sobre a identificação dos tipos de defeitos. Não há uma taxonomia única dos tipos de defeitos aplicáveis no contexto de modelos de programação concorrente para CUDA. Dessa forma, realizamos um agrupamento de defeitos com base nos tipos encontrados em diversos artigos e livros do modelo de programação CUDA.

A instrumentação do código CUDA apresenta algumas limitações que dificultaram o processo de geração do rastro de execução. Não é viável escrever o rastro de execução em um arquivo para cada *thread* diretamente do código do *kernel*. Ao mesmo tempo, é inviável criar estruturas de memória dinâmica para armazenar o rastro de cada *thread*. Dessa forma, foi necessário alocar toda a estrutura de memória para o registro dos rastros antes da execução do *kernel*, o que implica em restrições no uso de memória. Algumas técnicas foram empregadas para diminuir esse custo, como a instrumentação apenas dos nós essenciais.

O desenvolvimento da ferramenta também se apresentou com um desafio no processo. A utilização do *framework* Clava ajudou nesse processo. Ao mesmo tempo, a necessidade de realizar adaptações e implementar novas funções nos módulos da ValiMPI também foi um desafio. Foi necessário analisar e entender o funcionamento de todos os módulos para poder implementar os recursos relacionados aos critérios de teste para CUDA.

A utilização de métodos estatísticos para a avaliação dos critérios foi um outro desafio. Não sendo um componente da nossa área, foi necessário realizar a pesquisa em livros relacionados a experimentos, consultar livros de estatística que abordem os principais métodos. Buscar outros trabalhos que utilizaram métodos estatísticos similares para avaliação dos critérios de teste, e lidar com o uso de programas estatísticos para realizar a avaliação das hipóteses.

9.4 Limitações

A ferramenta não suporta alguns recursos relacionados ao modelo de programação CUDA, que são abordados nos trabalhos futuros. Dos recursos previstos no modelo de teste, a ferramenta não suporta o argumento na chamada do *kernel* ter nome diferente

do parâmetro apresentado no *kernel*, sendo necessário que ambos possuam nomes iguais para que a ferramenta faça a ligação entre eles e crie corretamente os elementos requeridos relacionados a essa comunicação. Caso possuam nomes diferentes, é necessário criar manualmente os elementos requeridos e autômatos específicos. A ValiEval possui a capacidade de avaliar a cobertura desses elementos requeridos.

O modelo de teste e a ferramenta não suportam a chamada de funções dentro do *kernel* e a passagem de argumentos para ele. Por consequência, não há suporte para paralelismo dinâmico previsto no modelo CUDA.

Há suporte apenas a uma estrutura de repetição e uma estrutura de decisão. A ferramenta suporta apenas a estrutura *for*, não suportando outros tipos de estruturas de repetição, como o *while*. Em termos de estrutura de decisão, não há suporte para o uso de *switch*.

O modelo de teste e a ferramenta consideram a existência de apenas uma *thread* no *host*. Dessa forma, programas CUDA que possuam mais de uma *thread* no *host* não são previstos pelos critérios de teste apresentados.

No experimento realizado, não abordamos dois tipos de defeitos descritos no Capítulo 4. A ferramenta utilizada apresenta somente uma implementação inicial para operações atômicas, portanto, não consideramos o defeito de violação de atomicidade nos experimentos. Além disso, não abordamos o defeito de falta de qualificador *volatile*, pois essa funcionalidade ainda não foi implementada na ferramenta.

9.5 Trabalhos Futuros

Para trabalhos futuros, alguns pontos que podem ser explorados em relação ao modelo, critérios de teste e ferramenta são:

- avaliação dos critérios de teste propostos utilizando outros *benchmarks*, que possuam características diferentes das apresentadas pelo conjunto utilizado. Dentre essas características, inclui o teste de aplicações com mais configurações diferentes de blocos e *threads*.
- adicionar suporte a mais operações atômicas à ferramenta. Isto possibilitará testar a capacidade dos critérios atuais em revelar defeitos pela falta de atomicidade em regiões críticas.
- adicionar suporte ao CUDA *Streams*. Isso permite que *kernels* de *streams* diferentes sejam executados concorrentemente, dessa forma, afetando o modo de sincronização no *host*. É necessário avaliar como o modelo de teste e os critérios atuais podem se adequar ao uso de *streams*, e avaliar se novos critérios podem ser mais adequados a esse cenário.

- adicionar suporte a *template*. Diversos códigos do CUDA *Toolkit* de exemplo utilizam o recurso de *template*, o que impede a ValiCUDA instrumentar e avaliar a cobertura nesses códigos sem realizar uma adaptação neles. Adicionar suporte na ferramenta a esse recurso permitirá a execução de mais códigos desenvolvidos por terceiros.

- adicionar suporte a outros recursos do modelo CUDA, como *cooperative groups*, *memory fence*, *cluster* de blocos de *threads* e primitivas assíncronas, como a cópia de memória assíncrona.

- avaliar novas técnicas para diminuir o uso de recursos por parte da instrumentação. Aplicamos a ideia baseada nas arestas primitivas para a instrumentação do código do *kernel*, entretanto outras possibilidades podem ser exploradas para diminuir o uso de memória pela instrumentação.

- otimizar o rastro do caminho gerado pelas *threads*. Os rastros gerados podem ser otimizados de forma a diminuir o custo da avaliação dos autômatos na ValiEval. Algumas possíveis melhorias em relação ao rastro está em mitigar repetições de caminhos, de forma a diminuir o comprimento total do caminho. Outra melhoria nesse sentido seria verificar quais rastros são iguais, agrupar esses rastros, identificando quais *threads* executaram, permitindo que na ValiEval seja necessário avaliar a execução de apenas uma repetição do rastro, podendo aplicar o seu resultado para todas as outras *threads* que geraram o mesmo rastro.

- Estender o modelo e critérios de teste para programas CUDA que possuam mais de uma *thread* ou processo no *host*, por meio do uso de PThreads, OpenMP ou MPI.

- Aprimorar a qualidade da avaliação dos critérios de teste propostos utilizando outras técnicas de geração de dados de teste, como o teste de matriz ortogonal, que é uma técnica utilizada na área de para reduzir o tamanho do conjunto de testes. Além disso, pode-se utilizar a geração de mutantes através da ferramenta MutGPU (ZHU; ZAIDMAN, 2020). Dessa forma, poderíamos obter uma avaliação mais completa e abrangente dos critérios propostos.

REFERÊNCIAS

AHMAD, K.; WOLFE, M. Automatic testing of OpenACC applications. *In*: CHANDRASEKARAN, S.; JUCKELAND, G. (ed.). **Accelerator Programming Using Directives**. Cham: Springer International Publishing, 2018. p. 145–159. ISBN 978-3-319-74896-2.

ALGHAMDI, A. M.; EASSA, F. E. Openacc errors classification and static detection techniques. **IEEE Access**, v. 7, p. 113235–113253, 2019.

ALGHAMDI, A. M. *et al.* Parallel hybrid testing techniques for the dual-programming models-based programs. **Symmetry**, v. 12, n. 9, 2020. ISSN 2073-8994. Disponível em: https://www.mdpi.com/2073-8994/12/9/1555.

ALUR, R.; DEVIETTI, J.; SINGHANIA, N. Block-size independence for gpu programs. *In*: PODELSKI, A. (ed.). **Static Analysis**. Cham: Springer International Publishing, 2018. p. 107–126. ISBN 978-3-319-99725-4.

AMIGHI, A. *et al.* Specification and Verification of Atomic Operations in GPGPU Programs. *In*: COUNSELL, S.; NÚÑEZ, M. (ed.). **Software Engineering and Formal Methods**. Cham: Springer International Publishing, 2015. v. 8368, p. 69–83. ISBN 978-3-319-22969-0. Disponível em: https://doi.org/10.1007/978-3-319-22969-0_5.

ANAND, S.; POLIKARPOVA, N. Automatic Synchronization for GPU Kernels. *In*: **2018 Formal Methods in Computer Aided Design (FMCAD)**. [*S.l.: s.n.*]: FMCAD Inc, 2018. p. 1–9.

BALCI, O. Verification, validation, and testing. **Handbook of simulation**, John Wiley and Sons, v. 10, p. 335–393, 1998.

BARBOSA, E. F. *et al.* Teste estrutural. *In*: DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. (ed.). **Introdução ao Teste de Software**. 2st. ed. [*S.l.: s.n.*]: Elsevier, 2016. cap. 4. ISBN 9788535283532.

BARDSLEY, E. *et al.* Engineering a static verification tool for gpu kernels. *In*: **Computer Aided Verification**. Cham: Springer International Publishing, 2014. p. 226–242. ISBN 978-3-319-08867-9. Disponível em: https://doi.org/10.1007/978-3-319-08867-9_15.

BARDSLEY, E.; DONALDSON, A. F. Warps and atomics: Beyond barrier synchronization in the verification of gpu kernels. *In*: **NASA Formal Methods**. Cham: Springer International Publishing, 2014. p. 230–245. ISBN 978-3-319-06200-6. Disponível em: http://doi.org/10.1007/978-3-319-06200-6_18.

BARDSLEY, E.; DONALDSON, A. F.; WICKERSON, J. Kernelinterceptor: Automating gpu kernel verification by intercepting kernels and their parameters. *In*: **Proceedings of the International Workshop on OpenCL 2013 & 2014**. New York, NY, USA: ACM, 2014. (IWOCL '14). ISBN 978-1-4503-3007-7. Disponível em: https://doi.org/10.1145/2664666.2664673.

BAUMANN, T. M.; GRACIA, J. Cudagrind: Memory-Usage Checking for CUDA. *In*: **Tools for High Performance Computing 2013**. Cham: Springer International Publishing, 2014. p. 67–78. ISBN 978-3-642-31475-9.

BERTOT, Y.; CASTÉRAN, P. **Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions**. [*S.l.: s.n.*]: Springer Science & Business Media, 2013.

BETTS, A. *et al.* Implementing and evaluating candidate-based invariant generation. **IEEE Transactions on Software Engineering**, v. 44, n. 7, p. 631–650, 2018.

BETTS, A. *et al.* Gpuverify: A verifier for gpu kernels. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 47, n. 10, p. 113–132, out. 2012. ISSN 0362-1340. Disponível em: https://doi.org/10.1145/2398857.2384625.

BETTS, A. *et al.* Gpuverify: A verifier for gpu kernels. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 47, n. 10, p. 113–132, out. 2012. ISSN 0362-1340. Disponível em: http://doi.org/10.1145/2398857.2384625.

BETTS, A. *et al.* The Design and Implementation of a Verification Technique for GPU Kernels. **ACM Transactions on Programming Languages and Systems**, v. 37, n. 3, p. 1–49, 2015. ISSN 0164-0925. Disponível em: https://doi.org/10.1145/2743017.

BISPO, J.; CARDOSO, J. M. Clava: C/c++ source-to-source compilation using lara. **SoftwareX**, v. 12, p. 100565, 2020. ISSN 2352-7110. Disponível em: https://doi.org/10.1016/j.softx.2020.100565.

BLOM, S. *et al.* The vercors tool set: Verification of parallel and concurrent software. *In*: POLIKARPOVA, N.; SCHNEIDER, S. (ed.). **Integrated Formal Methods**. Cham: Springer International Publishing, 2017. p. 102–110. ISBN 978-3-319-66845-1.

BLOM, S.; HUISMAN, M. The vercors tool for verification of concurrent programs. *In*: **FM 2014: Formal Methods**. Cham: Springer International Publishing, 2014. p. 127–131. ISBN 978-3-319-06409-3. Disponível em: https://doi.org/10.1007/978-3-319-06410-9_9.

BLOM, S.; HUISMAN, M.; MIHELCIC, M. Specification and verification of GPGPU programs. **Science of Computer Programming**, v. 95, n. P3, p. 376–388, 2014. ISSN 01676423.

BOEHM, B. W. Software engineering economics. **IEEE Transactions on Software Engineering**, SE-10, n. 1, p. 4–21, 1984.

BOYER, M.; SKADRON, K.; WEIMER, W. Automated dynamic analysis of cuda programs. *In*: **Third Workshop on Software Tools for MultiCore Systems**. [*S.l.: s.n.*], 2008. p. 33.

CHAIM, M. L.; MALDONADO, J. C.; JINO, M. Poke-tool - uma ferramenta para suporte ao teste estrutural de programas baseado em analise de fluxo de dados. *In*: **Simposio Brasileiro de Engenharia de Software**. [*S.l.: s.n.*]: Sbc, 1991.

CHENG, J.; GROSSMAN, M.; MCKERCHER, T. **Professional CUDA C Programming**. [*S.l.: s.n.*]: Wiley, 2014. (Wrox : Programmer to Programmer). ISBN 9781118739310.

CHIANG, W.-F. *et al.* Formal analysis of gpu programs with atomics via conflict-directed delay-bounding. *In*: **NASA Formal Methods**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 213–228. ISBN 978-3-642-38088-4. Disponível em: https://doi.org/10.1007/978-3-642-38088-4_15.

CHONG, N. *et al.* Barrier invariants: A shared state abstraction for the analysis of data-dependent gpu kernels. *In*: **Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications**. New York, NY, USA: ACM, 2013. (OOPSLA '13), p. 605–622. ISBN 978-1-4503-2374-1. Disponível em: http://doi.org/10.1145/2509136.2509517.

CHONG, N.; DONALDSON, A. F.; KETEMA, J. A sound and complete abstraction for reasoning about parallel prefix sums. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 49, n. 1, p. 397–409, 2014. ISSN 0362-1340. Disponível em: https://doi.org/10.1145/2578855.2535882.

CHUSHO, T. Test data selection and quality estimation based on the concept of essential branches for path testing. **IEEE Transactions on Software Engineering**, SE-13, n. 5, p. 509–517, 1987. Disponível em: https://doi.org/10.1109/TSE.1987.233196.

COLLINGBOURNE, P.; CADAR, C.; KELLY, P. Symbolic testing of opencl code. **Hardware and Software: Verification and Testing**, Springer, p. 203–218, 2012.

COLLINGBOURNE, P.; CADAR, C.; KELLY, P. H. J. Symbolic crosschecking of data-parallel floating-point code. **IEEE Transactions on Software Engineering**, v. 40, n. 7, p. 710–737, July 2014. ISSN 0098-5589.

COLLINGBOURNE, P. *et al.* Interleaving and lock-step semantics for analysis and verification of gpu kernels. *In*: **Programming Languages and Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 270–289. ISBN 978-3-642-37036-6. Disponível em: https://doi.org/10.1007/978-3-642-37036-6_16.

COOK, S. **CUDA Programming: A Developer's Guide to Parallel Computing with GPUs**. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN 9780124159334.

COUTINHO, B. *et al.* Profiling divergences in GPU applications. **Concurrency and Computation: Practice and Experience**, v. 25, n. 6, p. 775–789, apr 2012. ISSN 15320626.

DELAMARO, M. E. *et al.* Teste de mutação. *In*: DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. (ed.). **Introdução ao Teste de Software**. 1st. ed. [*S.l.: s.n.*]: Elsevier, 2007. cap. 2. ISBN 8535226346.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao Teste de Software**. 1th. ed. [*S.l.: s.n.*]: Elsevier, 2007. ISBN 8535226346.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 11, n. 4, p. 34–41, abr. 1978. ISSN 0018-9162. Disponível em: http://dx.doi.org/10.1109/C-M.1978.218136.

DIAZ, S.; SOUZA, P.; SOUZA, S. A structural testing tool for mpi programs with loops. *In*: **Anais do XX Simpósio em Sistemas Computacionais de Alto Desempenho**. Porto Alegre, RS, Brasil: SBC, 2019. p. 406–417.

DONALDSON, A. F. *et al.* Forward Progress on GPU Concurrency (Invited Talk). *In*: MEYER, R.; NESTMANN, U. (ed.). **28th International Conference on Concurrency Theory (CONCUR 2017)**. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. (Leibniz International Proceedings in Informatics (LIPIcs), v. 85). ISBN 978-3-95977-048-4. Disponível em: http://doi.org/10.4230/LIPIcs.CONCUR.2017.1.

EASSA, F. E. *et al.* Acc_test: Hybrid testing approach for openacc-based programs. **IEEE Access**, v. 8, p. 80358–80368, 2020.

EIZENBERG, A. *et al.* BARRACUDA: binary-level analysis of runtime RAces in CUDA programs. **ACM SIGPLAN Notices**, v. 52, n. 6, p. 126–140, 2017. ISSN 03621340. Disponível em: https://doi.org/10.1145/3140587.3062342.

ENDO, A. T. *et al.* Web services composition testing: A strategy based on structural testing of parallel programs. *In*: **Testing: Academic & Industrial Conference - Practice and Research Techniques (taic part 2008)**. [*S.l.: s.n.*], 2008. p. 3–12.

FANG, J.; VARBANESCU, A.; SIPS, H. A comprehensive performance comparison of cuda and opencl. *In*: **International Conference on Parallel Processing**. [*S.l.: s.n.*], 2011. p. 216–225. ISSN 0190-3918.

FARBER, R. **CUDA Application Design and Development**. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN 9780123884268, 9780123884329.

FELIZARDO, K. R. *et al.* Estudos teóricos e experimentais. *In*: DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. (ed.). **Introdução ao Teste de Software**. 2st. ed. [*S.l.: s.n.*]: Elsevier, 2016. cap. 11. ISBN 9788535283532.

FERRELL, B.; DUAN, J.; HAMLEN, K. W. Cuda au coq: A framework for machine-validating gpu assembly programs. *In*: **2019 Design, Automation Test in Europe Conference Exhibition (DATE)**. [*S.l.: s.n.*], 2019. p. 474–479.

FLYNN, M. Some computer organizations and their effectiveness. **IEEE Transactions on Computers**, C-21, n. 9, p. 948–960, Sept 1972. ISSN 0018-9340.

GILADI, R. **Network Processors: Architecture, Programming, and Implementation**. [*S.l.: s.n.*]: Elsevier Science, 2008. (Systems on Silicon). ISBN 9780080919591.

GRAMA, A. *et al.* Introduction to parallel computing. **Introduction to Parallel Computing, 2nd ed, Pearson Education Limited**, v. 1, 2003.

HARRIS, M. Optimizing parallel reduction in cuda. **Nvidia developer technology**, Nvidia Corporation Santa Clara, CA, USA, 2007.

HARRIS, M. **An Efficient Matrix Transpose in CUDA C/C++**. 2013. Technical Blog. Disponível em: https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/. Acesso em: 23/11/2019.

HARRIS, M. **Unified Memory for CUDA Beginners**. 2017. Disponível em: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/. Acesso em: 12/10/2020.

HAUSEN, A. C. **ValiMPI : uma ferramenta de teste estrutural para programas paralelos em ambiente de passagem de mensagem**. 2005. Dissertação (Mestrado) — Universidade Federal do Paraná, 2005.

HAUSEN, A. C. *et al.* A tool for structural testing of mpi programs. *In*: **8th IEEE Latin-American Test Workshop**. [*S.l.: s.n.*], 2007.

HUISMAN, M. *et al.* Program correctness by transformation. *In*: **Leveraging Applications of Formal Methods, Verification and Validation. Modeling**. Cham: Springer International Publishing, 2018. p. 365–380. ISBN 978-3-030-03418-4. Disponível em: https://doi.org/10.1007/978-3-030-03418-4_22.

HUISMAN, M.; MIHELCIC, M. Specification and Verification of GPGPU Programs using Permission-Based Separation Logic. 2013.

IEEE. IEEE standard for software unit testing. **ANSI/IEEE Std 1008-1987**, 1986.

IEEE. IEEE standard glossary of software engineering terminology. **IEEE Std 610.12-1990**, 1990.

IEEE. **1003.1-2008 - Standard for Information Technology - Portable Operating System Interface (POSIX(R))**. 2008.

JOSHI, S.; MUDUGANTI, G. Gpurepair: Automated repair of gpu kernels. *In*: HENGLEIN, F.; SHOHAM, S.; VIZEL, Y. (ed.). **Verification, Model Checking, and Abstract Interpretation**. Cham: Springer International Publishing, 2021. p. 401–414. ISBN 978-3-030-67067-2.

KHRONOS GROUP. **The open standard for parallel programming of heterogeneous systems**. 2021. Disponível em: https://www.khronos.org/opencl/. Acesso em: 24/09/2021.

KIRK, D. B.; WEN-MEI, W. H. **Programming massively parallel processors: a hands-on approach**. [*S.l.: s.n.*]: Morgan Kaufmann, 2013. ISBN 978-0-12-415992-1.

KIRK, D. B.; WEN-MEI, W. H. **Programming massively parallel processors: a hands-on approach**. [*S.l.: s.n.*]: Morgan kaufmann, 2016.

KITCHENHAM, B.; CHARTERS, S. **Guidelines for performing Systematic Literature Reviews in Software Engineering**. [*S.l.*], 2007.

KOJIMA, K.; IMANISHI, A.; IGARASHI, A. Automated Verification of Functional Correctness of Race-Free GPU Programs. **Journal of Automated Reasoning**, Springer Netherlands, v. 60, n. 3, p. 279–298, 2018. ISSN 15730670.

LATTNER, C. Llvm and clang: Next generation compiler technology. *In*: **The BSD conference**. [*S.l.: s.n.*], 2008. v. 5, p. 1–20.

LEI, Y.; CARVER, R. H. Reachability testing of concurrent programs. **IEEE Trans. Softw. Eng.**, v. 32, n. 6, p. 382–403, 2006.

LEUNG, A. *et al.* Verifying gpu kernels by test amplification. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 47, n. 6, p. 383–394, jun 2012. ISSN 0362-1340. Disponível em: https://doi.org/10.1145/2345156.2254110.

LI, G.; GOPALAKRISHNAN, G. Scalable smt-based verification of gpu kernel functions. *In*: **Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering**. New York, NY, USA: ACM, 2010. (FSE '10), p. 187–196. ISBN 978-1-60558-791-2. Disponível em: http://doi.org/10.1145/1882291.1882320.

LI, G.; GOPALAKRISHNAN, G. Parameterized verification of gpu kernel programs. *In*: **2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum**. [*S.l.: s.n.*], 2012. p. 2450–2459.

LI, G. *et al.* Gklee: Concolic verification and test generation for gpus. *In*: **Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. New York, NY, USA: ACM, 2012. (PPoPP '12), p. 215–224. ISBN 978-1-4503-1160-1. Disponível em: http://doi.org/10.1145/2145816.2145844.

LI, P.; DING, C. LDetector : A Low Overhead Race Detector For GPU Programs. p. 1–6, 2014.

LI, P. *et al.* LD: Low-Overhead GPU Race Detection Without Access Monitoring. **ACM Trans. Archit. Code Optim.**, v. 14, n. 1, 2017. ISSN 1544-3566. Disponível em: http://doi.org/10.1145/3046678.

LI, P.; LI, G.; GOPALAKRISHNAN, G. Parametric flows: Automated behavior equivalencing for symbolic analysis of races in cuda programs. *In*: **Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis**. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012. (SC '12). ISBN 978-1-4673-0804-5. Disponível em: http://dl.acm.org/citation.cfm?id=2388996.2389036.

LI, P.; LI, G.; GOPALAKRISHNAN, G. Practical symbolic race checking of gpu programs. *In*: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. Piscataway, NJ, USA: IEEE Press, 2014. (SC '14), p. 179–190. ISBN 978-1-4799-5500-8. Disponível em: https://doi.org/10.1109/SC.2014.20.

LU, S. *et al.* Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. **SIGARCH Comput. Archit. News**, Association for Computing Machinery, New York, NY, USA, v. 36, n. 1, p. 329–339, 2008. ISSN 0163-5964. Disponível em: https://doi.org/10.1145/1353534.1346323.

MPI Forum. **Message Passing Interface Forum**. 2015. Disponível em: http://www.mpi-forum.org/. Acesso em: 31/10/2015.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. 3rd. ed. [*S.l.: s.n.*]: Wiley Publishing, 2011. ISBN 1118031962.

NEIDER, D. *et al.* Sorcar: Property-driven algorithms for learning conjunctive invariants. *In*: CHANG, B.-Y. E. (ed.). **Static Analysis**. Cham: Springer International Publishing, 2019. p. 323–346. ISBN 978-3-030-32304-2.

NETZER, R. H.; MILLER, B. P. What are race conditions?: Some issues and formalizations. **ACM Letters on Programming Languages and Systems (LOPLAS)**, ACM, v. 1, n. 1, p. 74–88, 1992.

NVIDIA. **NVIDIA CUDA C++ Programming Guide**. 2022. Version 11.7.

NVIDIA Corporation. **Whitepaper NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU**. 2016. Version 1.1.

OLIVEIRA, A. P.; SOUZA, P. S. L.; SOUZA, S. R. S. Valierlang: A structural testing tool for erlang programs. *In*: **Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing**. New York, NY, USA: Association for Computing Machinery, 2016. (SAST). ISBN 9781450347662. Disponível em: https://doi.org/10.1145/2993288.2993300.

OPENACC ORGANIZATION. **OpenACC**. 2022. Disponível em: https://www.openacc.org. Acesso em: 27/09/2022.

OpenMP ARB. **The OpenMP® API specification for parallel programming**. 2015. Disponível em: http://openmp.org. Acesso em: 31/10/2015.

PALA, P. **Taking a Look at Nvidia's Stock Amidst The Semiconductor Chip Gut**. 2022. Disponível em: https://www.entrepreneur.com/finance/taking-a-look-at-nvidias-stock-amidst-the-semiconductor/432115. Acesso em: 01/10/2022.

PAPATHEODORE, T. **Introduction to CUDA C/C++**. 2017. Oak Ridge National Laboratory. Disponível em: https://www.olcf.ornl.gov/wp-content/uploads/2018/03/Intro_to_CUDA.pdf. Acesso em: 17/04/2019.

PATTERSON, D.; HENNESSY, J. **Computer Organization and Design: The Hardware/Software Interface**. [*S.l.: s.n.*]: Elsevier Science, 2013. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 9780124078864.

PATTERSON, D.; HENNESSY, J. **Computer Organization and Design RISC-V Edition: The Hardware Software Interface**. [*S.l.: s.n.*]: Elsevier Science, 2020. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 9780128203316.

PENG, C.; RAJAN, A. Cltestcheck: Measuring test effectiveness for gpu kernels. *In*: HÄHNLE, R.; AALST, W. van der (ed.). **Fundamental Approaches to Software Engineering**. Cham: Springer International Publishing, 2019. p. 315–331. ISBN 978-3-030-16722-6.

PENG, C.; RAJAN, A. Automated test generation for opencl kernels using fuzzing and constraint solving. *In*: **Proceedings of the 13th Annual Workshop on General Purpose Processing Using Graphics Processing Unit**. New York, NY, USA: Association for Computing Machinery, 2020. (GPGPU '20), p. 61–70. ISBN 9781450370257. Disponível em: https://doi.org/10.1145/3366428.3380768.

PENG, Y.; GROVER, V.; DEVIETTI, J. Curd: A dynamic cuda race detector. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 53, n. 4, p. 390–403, 2018. ISSN 0362-1340. Disponível em: https://doi.org/10.1145/3296979.3192368.

PEREIRA, P. *et al.* Verifying cuda programs using smt-based context-bounded model checking. *In*: **Proceedings of the 31st Annual ACM Symposium on Applied Computing**. New York, NY, USA: ACM, 2016. (SAC '16), p. 1648–1653. ISBN 978-1-4503-3739-7. Disponível em: http://doi.org/10.1145/2851613.2851830.

PRADO, R. R. *et al.* Extracting static and dynamic structural information from java concurrent programs for coverage testing. *In*: **2015 Latin American Computing Conference (CLEI)**. [*S.l.: s.n.*], 2015. p. 1–8.

PRADO, R. R. do. **Teste estrutural de programas concorrentes como composição de serviços na WEB**. 2016. Dissertação (Mestrado) — Universidade de São Paulo, 2016.

PRESSMAN, R.; MAXIM, B. **Engenharia de Software - 8ª Edição**. [*S.l.: s.n.*], 2016. ISBN 9788580555349.

RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for test data selection. *In*: **Proceedings of the 6th International Conference on Software Engineering**. Los Alamitos, CA, USA: IEEE Computer Society Press, 1982. (ICSE '82), p. 272–278. Disponível em: http://dl.acm.org/citation.cfm?id=800254.807769.

REGEHR, J. **Race Condition vs. Data Race**. 2011. Disponível em: https://blog.regehr.org/archives/490. Acesso em: 12/08/2019.

SAFARI, M. *et al.* Formal verification of parallel prefix sum. *In*: LEE, R. *et al.* (ed.). **NASA Formal Methods**. Cham: Springer International Publishing, 2020. p. 170–186. ISBN 978-3-030-55754-6.

SARKAR, S. *et al.* Analysis of gpgpu programs for data-race and barrier divergence. *In*: INSTICC. **Proceedings of the 13th International Conference on Software Technologies - ICSOFT,**. [*S.l.: s.n.*]: SciTePress, 2018. p. 460–471. ISBN 978-989-758-320-9. ISSN 2184-2833.

SARMANHO, F. S. *et al.* Structural testing for semaphore-based multithread programs. *In*: BUBAK, M. *et al.* (ed.). **Computational Science – ICCS 2008**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 337–346. ISBN 978-3-540-69384-0.

SHARMA, R.; BAUER, M.; AIKEN, A. Verification of producer-consumer synchronization in gpu programs. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 50, n. 6, p. 88–98, 2015. ISSN 0362-1340. Disponível em: https://doi.org/10.1145/2813885.2737962.

SOUZA, P. S.; SOUZA, S. R.; ZALUSKA, E. Structural testing for message-passing concurrent programs: an extended test model. **Concurrency and Computation: Practice and Experience**, v. 26, n. 1, p. 21–50, 2014.

SOUZA, P. S. L. *et al.* Valipvm - a graphical tool for structural testing of pvm programs. *In*: **15th European PVM/MPI - LNCS - Recent Advances in PVM and MPI**. Dublin: [*S.l.: s.n.*], 2008. p. 257–264. ISBN 978-3-540-87474-4.

SOUZA, P. S. L. *et al.* Data flow testing in concurrent programs with message passing and shared memory paradigms. **Procedia Computer Science**, v. 18, n. 0, p. 149 – 158, 2013. ISSN 1877-0509. Int. Conf. on Computational Science. Disponível em: http://www.sciencedirect.com/science/article/pii/S1877050913003219.

SOUZA, S. d. R. S. d. *et al.* Valipar: a testing tool for message-passing parallel programs. *In*: **International Conference on Software Engineering & Knowledge Engineering**. [*S.l.: s.n.*]: UND/Tung Hai University, 2005.

SOUZA, S. R. S. *et al.* Structural testing criteria for message-passing parallel programs. **Concurrency and Computation: Practice and Experience**, v. 20, n. 16, p. 1893–1916, 2008.

STALLINGS, W. **Computer Organization and Architecture**. 9. ed. [*S.l.: s.n.*]: Prentice Hall, 2012. ISBN 013293633X,9780132936330.

STERLING, T. *et al.* Beowulf: A parallel workstation for scientific computation. *In*: **In Proceedings of the 24th Int. Conf. on Parallel Processing**. [*S.l.: s.n.*]: CRC Press, 1995. p. 11–14.

STORTI, D.; YURTOGLU, M. **CUDA for Engineers: An Introduction to High-performance Parallel Computing**. [*S.l.: s.n.*]: Addison-Wesley, 2015. (Always learning). ISBN 9780134177410.

TANENBAUM, A. S.; BOS, H. **Modern Operating Systems**. 4th. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. ISBN 013359162X, 9780133591620.

TOP500.ORG. **Top500 Supercomputer**. 2022. Disponível em: https://top500.org/lists/top500/2022/06/. Acesso em: 28/07/2022.

VERGILIO, S. R.; SOUZA, S. R. S.; SOUZA, P. S. L. Coverage testing criteria for message-passing parallel programs. *In*: **6th IEEE Latin American Test Workshop**. [*S.l.: s.n.*], 2005. p. 161–166.

WOHLIN, C. *et al.* **Experimentation in software engineering**. [*S.l.: s.n.*]: Springer Science & Business Media, 2012.

WU, M. *et al.* Simulee: Detecting cuda synchronization bugs via memory-access modeling. *In*: **Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2020. (ICSE '20), p. 937–948. ISBN 9781450371216. Disponível em: https://doi.org/10.1145/3377811.3380358.

WU, M. *et al.* Automating cuda synchronization via program transformation. *In*: **2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [*S.l.: s.n.*], 2019. p. 748–759.

XING, Y. *et al.* A formal instruction-level GPU model for scalable verification. **Proceedings of the International Conference on Computer-Aided Design - ICCAD '18**, p. 1–8, 2018. Disponível em: https://doi.org/10.1145/3240765.3240771.

YANG, C.-S.; POLLOCK, L. L. The challenges in automated testing of multithreaded programs. *In*: **In Proceedings of the 14th Int. Conf. on Testing Computer Software**. [*S.l.: s.n.*], 1997. p. 157–166.

YANG, C.-S. D. **Program-based, structural testing of shared memory parallel programs**. 1999. Dissertação (Mestrado) — University of Delaware, 1999.

YANG, C.-S. D.; SOUTER, A. L.; POLLOCK, L. L. All-du-path coverage for parallel programs. **SIGSOFT Softw. Eng. Notes**, Association for Computing Machinery, New York, NY, USA, v. 23, n. 2, p. 153–162, mar 1998. ISSN 0163-5948.

YAP, B. W.; SIM, C. H. Comparisons of various types of normality tests. **Journal of Statistical Computation and Simulation**, Taylor & Francis, v. 81, n. 12, p. 2141–2155, 2011.

ZHENG, M. *et al.* Grace: A low-overhead mechanism for detecting data races in gpu programs. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 46, n. 8, p. 135–146, fev. 2011. ISSN 0362-1340. Disponível em: https://doi.org/10.1145/1941553.1941574.

ZHENG, M. *et al.* Gmrace: Detecting data races in gpu programs via a low-overhead scheme. **IEEE Transactions on Parallel and Distributed Systems**, v. 25, n. 1, p. 104–115, Jan 2014. ISSN 1045-9219.

ZHU, Q.; ZAIDMAN, A. Massively parallel, highly efficient, but what about the test suite quality? applying mutation testing to gpu programs. *In*: **2020 IEEE 13th International Conference on Software Testing, Verification and Validation (ICST)**. [*S.l.: s.n.*], 2020.