Universidade Federal do Amazonas

Faculdade de Tecnologia

Programa de Pós-Graduação em Engenharia Elétrica

Verificação Limitada de Modelos Baseada em SMT para Programas CUDA

Phillipe Arantes Pereira

Manaus – Amazonas

Fevereiro de 2019

Phillipe Arantes Pereira

Verificação Limitada de Modelos Baseada em SMT para Programas CUDA

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétrica. Área de concentração: Automação e Controle.

Orientador: Lucas Carvalho Cordeiro

Phillipe Arantes Pereira

Verificação Limitada de Modelos Baseada em SMT para Programas CUDA

Banca Examinadora

Prof. D.Sc. Lucas Carvalho Cordeiro – Presidente e Orientador
Universidade de Manchester – UK

Prof. D.Sc. Ricardo Santos Ferreira
Departamento de Informática – UFV

Prof. D.Sc. José Augusto Nacif
Instituto de Ciências Exatas e Tecnológicas – UFV

Manaus – Amazonas
Fevereiro de 2019

À minha família.

Agradecimentos

Agradeço primeiramente aos meus pais Wilton e Aline, que sempre apoiaram minhas decisões e acreditaram nas minhas escolhas. Agradeço também aos meus avós Sahara e (*in memoriam*, sempre) José e Maria de Lourdes, pelos princípios ensinados e por mostrarem a simplicidade pela qual podemos conduzir a vida. À minha irmã Raphaella, por me ensinar a ser paciente e a persistir nos objetivos. À minha namorada Ana Caroline por me incentivar e apoiar na conclusão deste trabalho. Aos familiares que diretamente ou indiretamente me influenciaram a alcançar meus objetivos.

Aos amigos e colegas de faculdade e de boa parte do mestrado que fizeram parte de momentos de estudos e diversão: João Paulo, Raimundo Williame, Mário e Higo; e do Laboratório de Verificação de Software: Isabela, Hendrio, Vanessa, Hussama e Felipe, pela colaboração e amizade.

Aos professores Celso Carvalho e Ricardo Ferreira por colaboração e apoio em alguns momentos deste trabalho. Por fim, ao Prof. Lucas Cordeiro por todo direcionamento desde a graduação, pela confiança no desenvolvimento de projetos, e principalmente pela paciência e determinação ímpar em motivar a conclusão desta dissertação.

A vitalidade é demonstrada não apenas pela persistência, mas pela capacidade de começar de novo.

F. Scott Fitgerald

Resumo

A ferramenta de verificação ESBMC-GPU é uma extensão do Verificador de Modelos de Contexto Limitado baseado em SMT (ESBMC), que tem como propósito verificar programas de Unidades de Processamento Gráfico (GPU) escritos na plataforma de Computação Unificada da Arquitetura de Dispositivos (CUDA). ESBMC-GPU usa um modelo operacional, *i.e.*, uma abstração das bibliotecas do CUDA, que conservativamente aproxima suas semânticas para verificar os programas. Assim, são exploradas as possíveis intercalações (sob um dado limite de contexto), enquanto trata cada uma simbolicamente. Além disso, a ferramenta implementa um algoritmo de redução monotônica de ordem parcial para realizar uma análise sobre duas *threads* a fim de reduzir a exploração do espaço de estados.

Resultados experimentais mostram que o ESBMC-GPU pode, com sucesso, verificar 82% dos *benchmarks* enquanto mantém baixas taxas de falsos resultados. A ferramenta também é capaz de detectar mais violações de propriedades quando comparada a outras ferramentas de verificação existentes. Isso é devido a sua capacidade de verificar erros no fluxo de execução do programa e detectar violações de condições de corrida e acessos fora dos limites de vetores.

Palavras-chave: GPU; programas CUDA; verificação formal; verificação de modelos.

Abstract

We present ESBMC-GPU tool, an extension to the Efficient SMT-Based Context-Bounded Model Checker (ESBMC), which is aimed at verifying Graphics Processing Unit (GPU) programs written for the Compute Unified Device Architecture (CUDA) platform. ESBMC-GPU uses an operational model, *i.e.*, an abstract representation of the standard CUDA libraries, which conservatively approximates their semantics, in order to verify CUDA-based programs. It then explicitly explores the possible interleavings (up to the given context bound), while treats each interleaving itself symbolically. Additionally, ESBMC-GPU employs the monotonic partial order reduction and the two-thread analysis to prune the state space exploration.

Experimental results show that ESBMC-GPU can successfully verify 82% of all benchmarks, while keeping lower rates of false results. Going further than previous attempts, ESBMC-GPU is able to detect more properties violations than other existing GPU verifiers due to its ability to verify errors of the program execution flow and to detect array out-of-bounds and data race violations.

Keywords: GPU; CUDA kernels; formal verification; model checking.

Índice

Índice de Figuras **xi**

Índice de Tabelas **xii**

Abreviações **xiii**

1 Introdução **1**

1.1 Descrição do Problema 1

1.2 Objetivos 2

1.3 Descrição da Solução 2

1.4 Contribuições 3

1.5 Organização da Dissertação 4

2 Fundamentação Teórica **5**

2.1 Linguagem de Programação CUDA 5

2.2 Teorias do Módulo de Satisfatibilidade (SMT) 6

2.3 ESBMC 7

2.3.1 Arquitetura do ESBMC 9

2.4 Resumo 10

3 Trabalhos Relacionados **11**

3.1 GPU Verify 11

3.2 SESA e GKLEE 11

3.3 PUG 12

3.4 CIVL 12

3.5 Resumo 12

4 Verificando Programas CUDA **14**

4.1 Modelos Operacionais para Bibliotecas CUDA 14

4.1.1 Corretude do Modelo Operacional CUDA (MOC) 18

4.2 Modelando o Kernel com Funções Pthread 19

4.2.1 Sincronização de Threads 21

4.2.2 Exemplo Ilustrativo 21

4.3 Redução Monotônica de Ordem Parcial 24

4.3.1 Diferenças entre MPOR e Outros Métodos de Redução de Intercalações 26

4.4 Análise por Duas *Threads* 27

4.5 Resumo 28

5 Avaliação Experimental **30**

5.1 Objetivos Experimentais 30

5.2 Configuração Experimental 30

5.3 Avaliação dos solucionadores SMT para Verificação de Programas CUDA . . . 33

5.4 Resultados Experimentais 33

5.5 Efetividade do ESBMC-GPU comparado a outros verificadores 36

6 Conclusões **38**

6.1 Trabalhos Futuros 38

Referências Bibliográficas **40**

A Publicações **45**

A.1 Referente à Pesquisa 45

Índice de Figuras

2.1 Arquitetura do ESBMC. Os retângulos branco representam entradas e saídas. O retângulo cinza representa passos de verificação. 9

4.1 Arquitetura entre o Modelo Operacional CUDA e o ESBMC. 15
4.2 Implementação da *dim3*. 15
4.3 Organização do Modelo Operacional CUDA(MOC). 16
4.4 Trecho de código para indexar o vetor. 23
4.5 Passos de conversão do MOC. 23
4.6 MPOR aplicado ao *kernel* com transições independentes (Fig. 4.6a) e dependentes (Fig. 4.6b). 26
4.7 Comparação entre a arquitetura da GPU NVIDIA Fermi e a abordagem de duas *threads* usada para manipular dados. 28

Índice de Tabelas

2.1 Exemplo de teorias suportadas. 7

3.1 Comparativo das ferramentas. 13

5.1 Propriedades e características exploradas pelos *kernels*. 31

5.2 Resultados do desempenho dos solucionadores SMT. **Verdadeiro Correto** representa o número de *benchmarks* sem falhas verificados corretamente, enquanto que **Verdadeiro Incorreto** representa o número dos verificados sem falhas incorretamente. Similarmente, **Falso Correto** representa o número de *benchmarks* verificados com falha corretamente, enquanto que **Falso Incorreto** o número dos verificados com falha incorretamente. Não suportados representa o número de *benchmarks* sem êxito na verificação, **Tempo** representa o tempo total de verificação de cada solucionador, e os números em negrito representam os melhores resultados de cada categoria. 34

5.3 Resultados experimentais. 35

Abreviações

AA - *Árvore de Alcançabilidade*

ASA - *Árvore de Sintaxe Abstrata*

BMC - *Bounded Model Checking*

CBMC - *C Bounded Model Checker*

CFG - *Control Flow Graph*

CPU - *Central Processing Unit*

CUDA - *Computação Unificada da Arquitetura de Dispositivos*

CV - *Condição de Verificação*

ESBMC - *Efficient SMT-Based Context-Bounded Model Checker*

GPU - *Graphics Processing Unit*

MOC - *Modelo Operacional CUDA*

MPOR - *Monotonic Partial Order Reduction*

PUG - *Prove of User GPU*

RI - *Representação Intermediária*

SDV - *Synchronous Delayed Visibility*

SESA - *Symbolic Executor with Static Analysis*

SMT - *Satisfiability Modulo Theories*

SSA - *Static Single Assignments*

Capítulo 1

Introdução

CUDA é uma plataforma de computação paralela e uma API criada pela NVIDIA [1], que estende C/C++ e Fortran, para criar um modelo computacional, que usa a capacidade de processamento das GPU's(*Graphics Processing Unit*) [2]. Como em outras linguagens de programação, erros de acesso fora dos limites de matrizes, estouro aritmético e violações de divisão por zero podem eventualmente ocorrer em programas CUDA. Além disso, uma vez que CUDA é uma plataforma que lida com programação paralela, erros específicos de concorrência relacionados a condições de corrida e divergência de barreira podem ser expostos devido ao comportamento não-determinístico das intercalações de *threads* [3] na execução dos programas.

1.1 Descrição do Problema

CUDA é uma plataforma nativamente utilizada para o desenvolvimento de aplicações gráficas e jogos eletrônicos. Atualmente a plataforma tem conquistado espaço na implementação de algoritmos, os quais realizam grandes operações de cálculos matemáticos como simulações meteorológicas, ciência dos dados, bioinformática e até no processamento de imagens médicas. Diante deste cenário, surgiu a necessidade de garantir que estes algoritmos sejam altamente precisos evitando que erros não perceptíveis alterem seus resultados.

Para detectar estes erros, é usada uma nova abordagem onde a problemática foca em identificar problemas conhecidos da literatura(*i.e.,* condições de corrida, divergência de barreira, dentre outros) e aplicar as técnicas de Verificação Formal de Software para analisar os algoritmos e exibir falhas que podem ocorrer durante a execução das aplicações. Estas apli-

cações processadas em GPU's executam sobre milhares de *threads*. Essa é uma característica de processamento devido a arquitetura da GPU, onde, ao aplicar as técnicas de verificação formal, pode resultar na explosão do espaço de estados. Esse é um desafio a mais neste tipo de análise, pois os possíveis estados de execução de um algoritmo são explorados. Isso pode inviabilizar a verificação de programas CUDA caso a quantidade de caminhos possíveis cresça exponencialmente, pois não seria possível verificar todos os caminhos e definir de maneira assertiva o resultado da análise do algoritmo.

1.2 Objetivos

O objetivo geral desta dissertação é aplicar a técnica de BMC, através do Verificador Eficiente de Modelos Limitados Baseado em SMT (ESBMC) [4, 5, 6], para auxiliar desenvolvedores de programas para GPU a detectar erros não perceptíveis na implementação de soluções com a plataforma CUDA.

Os objetivos específicos são:

- Propor uma ferramenta baseada no ESBMC que permita verificar programas CUDA aplicando a técnica de BMC;
- Implementar um modelo operacional (*e.g.*, uma representação abstrata de bibliotecas padrão que aproxima suas semânticas) para o ESBMC suportar CUDA;
- Propor uma metodologia para realizar análises sobre um número menor de *threads* CUDA;
- Desenvolver um algoritmo para redução de intercalações dos programas;
- Avaliar experimentalmente o desempenho da ferramenta com diferentes solucionadores SMT;
- Avaliar experimentalmente a eficiência da ferramenta quando comparada a outras ferramentas estado da arte na verificação de programas CUDA.

1.3 Descrição da Solução

Em contraste a esforços anteriores [3, 7, 8, 9], foi combinada a verificação simbólica de modelos baseada em Verificação de Modelos Limitados (BMC) com técnicas do Módulo

de Teorias da Satisfatibilidade (SMT) e exploração explícita do espaço de estados, similar a Codeiro *et al.* [4]. Em particular, foram exploradas as possíveis intercalações (sobre um contexto limitado), enquanto cada intercalação foi tratada simbolicamente com respeito a uma dada propriedade.

Para reduzir o espaço de estados explorado, foi aplicado o algoritmo de Redução Monotônica de Ordem Parcial (MPOR) [10] para programas CUDA, o qual elimina intercalações redundantes sem perder qualquer comportamento que possa ser exibido pelo programa. Desde que *kernels* (*i.e.*, programas que executam na GPU) normalmente produzem acessos regulares e independentes para explorar os benefícios do modelo de execução na GPU, a aplicação do MPOR rotineiramente leva a melhorias substanciais de desempenho em muitos *benchmarks*. Assim, ao usar modelos operacionais que simulam bibliotecas CUDA, junto com a implementação do MPOR no ESBMC-GPU, foram alcançados resultados significantes na verificação de *kernels* CUDA, principalmente quando comparado a outros verificadores (estado da arte) de programas para GPU [8, 3, 7, 9]. Adicionalmente, a abordagem proposta considera aspectos de baixo nível relacionado a alocação dinâmica de memória, transferência de dados, desalocação de memória e estouro aritmético, dos quais são tipicamente presentes em programas CUDA, porém são frequentemente ignorados pela maioria dos verificadores para GPU. Assim, é fornecida uma verificação mais precisa quando comparada às abordagens existentes, considerando dados transferidos pela função *main* do programa para o *kernel*, assumindo a consequência de aumentar o tempo de verificação.

1.4 Contribuições

Este trabalho faz quatro contribuições gerais:

- aplica a técnica de verificação de modelos de contexto limitado baseado em SMT para análise em programas CUDA;
- desenvolve uma aplicação do algoritmo MPOR para redução do espaço de estados eliminando intercalações redundantes;
- fornece uma ferramenta efetiva e eficiente (ESBMC-GPU) para suportar a verificação de diversos programas CUDA. ESBMC-GPU e todos os programas de teste usados durante o processo de avaliação estão disponíveis em `http://gpu.esbmc.org`;

- fornece uma avaliação experimental extensa do ESBMC-GPU contra GKLEE [8], GPU-Verify [3], PUG [7] e CIVL [9], usando programas padrão CUDA, que foram extraídos da literatura [1, 3, 11], contribuindo para a análise das diferentes técnicas aplicadas na verificação de programas CUDA.

Este trabalho estende trabalhos anteriores [12, 13]. A versão do ESBMC-GPU, descrita e avaliada aqui, tem sido otimizada e estendida. Ela agora expande o modelo operacional para suportar mais bibliotecas CUDA (*e.g.,* funções matemáticas) e inclui novos solucionadores SMT como *back-end* para o ESBMC-GPU. Também são fornecidos detalhes adicionais sobre o modelo operacional, a aplicação do MPOR para programas CUDA e a técnica de análise sobre duas *threads*. A base experimental foi significativamente estendida com a inclusão de um novo verificador (CIVL) e a avaliação experimental foi atualizada com as versões mais estáveis dos verificadores existentes.

1.5 Organização da Dissertação

O trabalho está organizado da seguinte maneira: o Capítulo 2 com uma breve introdução à CUDA e ao ESBMC, seguido pelo Capítulo 3 sobre verificadores existentes para programas de GPU. No Capítulo 4 é descrito o modelo operacional de bibliotecas CUDA, a aplicação do MPOR e a análise sobre duas *threads*. No Capítulo 5 são apresentados os resultados dos experimentos usando diversos programas disponíveis publicamente. No Capítulo 6 são apresentados a conclusão e trabalhos futuros.

Capítulo 2

Fundamentação Teórica

2.1 Linguagem de Programação CUDA

CUDA é uma plataforma de computação paralela(desenvolvida pela NVIDIA) para representar um modelo de programação para GPU's [1, 11].

No modelo de programação CUDA, o conceito de *kernel* é definido por um especificador *__global__* e usa a notação $kernel<<<B,T>>>$, onde B e T são o número de blocos e *threads* por bloco respectivamente. Cada *kernel* executa na GPU como *thread* e cada *thread* recebe um identificador único(ID), que é gerado a partir da linearização da posição da *thread* no bloco. O ID da *thread* é usado para indexar suas tarefas(*i.e.*, posições de memória e cooperação). *Threads* são normalmente organizadas por blocos na GPU onde, dentro de um bloco, a hierarquia de *threads* é definida por uma variável chamada *threadIdx*. Esta variável é representada por um vetor de três componentes que permite o uso de índices uni, bi e tridimensionais [11].

Blocos também podem ser definidos em três dimensões, onde cada dimensão pode ser acessada através da variável *blockIdx*. Essa variável é também composta por três componentes que permitem programas CUDA usar blocos uni, bi e tridimensionais. O número máximo de *threads* por bloco depende da plataforma e versão do *hardware*, mas normalmente varia de 1024 a 2048 [11]. Além disso, blocos tem uma propriedade que permite sua execução em qualquer ordem e não possuem um processador definido. Como consequência, um *kernel* deve também ser executado por múltiplos blocos, e o número total de *threads* representa o número de blocos multiplicado pelo número de *threads* por bloco.

No modelo de programação heterogênea de CUDA, a GPU é referenciada como *de-*

vice e a Unidade de Processamento Central(CPU) é referenciada como *host*. Isso admite que *threads* são executadas em dispositivos separados fisicamente, *i.e.*, *kernels* executam na GPU e o restante do programa executa na CPU [11]. Para isso foram definidos os especificadores de funções: *__device__*, que são para funções executadas e chamadas apenas pela GPU; *__host__*, que são para funções executadas e chamadas apenas pela CPU; e *__global__*, que é um especificador de função o qual opera como um ponto de entrada para executar *kernels*. CUDA também diferencia os espaços de memória. A alocação de dados no *device* é realizada pelo *host* usando as funções *cudaMalloc*, *cudaFree* e *cudaMemcpy*. Essas são funções essenciais para programas CUDA, que transferem dados do *host* para o *device* e vice-versa.

As *threads* devem acessar dados de diversos espaços de memória durante sua execução. Cada *thread* tem memória local privada e cada bloco tem memória compartilhada visível por todas as *threads* do bloco (assim como seu tempo de vida). Dentro do bloco, as *threads* podem cooperar por compartilhar dado através de memória compartilhada e por sincronizar sua execução para acessos coordenados. Pontos de sincronização podem ser especificados no *kernel* para chamar a função intrínseca *__syncthreads()*, que age como uma barreira. Todas as *threads* têm acesso à mesma memória global. Além disso, existe a memória unificada, que permite o gerenciamento de memória para fazer a ponte entre espaços do *host* e *device*, sendo acessível de todas as CPU's e GPU's do sistema como uma única e coerente imagem da memória. Esta capacidade elimina a necessidade de espelhar dados do *host* para o *device* por ser um espaço de endereçamento comum [11].

2.2 Teorias do Módulo de Satisfatibilidade (SMT)

SMT (*Satisfiability Module Theories*) verifica a satisfatibilidade de fórmulas de primeira ordem a partir de uma ou mais teorias de fundamentação, que são compostas por um conjunto de sentenças. De modo formal, $\sum$ - *theory* é uma coleção de sentenças sobre a assinatura $\sum$. Dada uma teoria T, nós dizemos que φ é um módulo satisfatível de T se $T \bigcup \{ \varphi \}$. Em outra definição, podemos dizer que uma teoria T é definida como uma classe de estruturas e φ é um módulo satisfatível se existe uma estrutura M em T que satisfaz φ (*i.e.*, $M \models \varphi$) [14].

Solucionadores SMT verificam a satisfatibilidade de fórmulas de primeira ordem escritas em uma linguagem contida por predicados e funções interpretadas. Esses símbolos interpretados são definidos por axiomas de primeira ordem (*e.g.*, axiomas de igualdade ou vetores

para operadores de leitura e escrita) ou por uma estrutura (*e.g.*, número inteiros equipados com constantes, soma, igualdade e desigualdades). Teorias frequentemente implementadas dentro de solucionadores SMT incluem a teoria vazia (a teoria de símbolos não interpretados com igualdade), a aritmética linear em inteiros e reais, vetores de bits e a teoria das matrizes [15].

Teoria	**Exemplo**
Igualdade	$x1 = x2 \wedge \neg(x2 = x3) \Rightarrow \neg(x1 = x3)$
Aritmética linear	$(4y_1 + 3y_2 \geq 4) \vee (y_2 - 3y_3 \leq 3)$
Vetores de bit	$(b \gg i)\&1 = 1$
Matrizes	$store(a, j, 2) \Rightarrow a[j] = 2$
Teorias combinadas	$(j \leq k \wedge a[j] = 2) \Rightarrow a[i] < 3$

Tabela 2.1: Exemplo de teorias suportadas.

A Tabela 2.1 mostra algumas das teorias suportados por solucionadores SMT usadas neste trabalho. A teoria da igualdade permite a verificação de igualdade e desigualdade entre predicados usando operadores $(=)$ $(\leq)$ $(<)$. A teoria de aritmética linear é apenas responsável por manipular funções aritméticas (adição, subtração, multiplicação e divisão) entre variáveis e constantes aritméticas. A teoria de vetores de bits permite operações bit a bit para diferentes arquiteturas (*e.g.*, 32 e 64 bits) e é descrita por alguns operadores como: e(&), ou($|$), ou-exclusivo($\oplus$), complemento($\sim$), deslocamento à direita($\gg$), deslocamento à esquerda($\ll$). Além disso, a teoria das matrizes permite a manipulação de operadores como *select* e *store*.

2.3 ESBMC

ESBMC é um premiado verificador de modelos de contexto limitado para programas C/C++ baseado em solucionadores SMT [16, 17]. A ferramenta é capaz de verificar programas sequencias e concorrentes com variáveis compartilhadas e sincronização usando Pthread/-POSIX [4, 5, 18, 19]. O ESBMC é uma das mais proeminentes ferramentas BMC para verificar programas C de acordo com as últimas edições da Competição Internacional de Verificação de Software (SV-COMP) [20, 21]. ESBMC pode verificar programas que contenham operações a nível de *bit*, matrizes, estruturas/*unions*, ponteiros, alocação dinâmica de memória, como também aritmética de ponto fixo. A ferramenta pode analisar *overflows*, segurança de ponteiro, vazamento de memória, acesso fora dos limites em matrizes, atomicidade e violações de ordem, *deadlocks*, condições de corrida e assertivas definidas pelos usuários.

No ESBMC os programas C/C++ são representados com um sistema de transição de estados $M = (S, R, s_0)$, que são tratados em um grafo de fluxo de controle(CFG). S representa o conjunto de estados, $R \subseteq S \times S$ representa o conjunto de transições e $s_0 \subseteq S$ representa o conjunto de estados iniciais. Um estado $s \in S$ consiste de um valor do contador de programa *pc* e os valores de todas as variáveis do programa. Um estado inicial s_0 atribui a posição inicial do programa no CFG ao *pc*. Cada transição $\gamma = (s_i, s_{i+1}) \in R$ entre dois estados s_i e s_{i+1}, é identificada como uma fórmula lógica $\gamma(s_i, s_{i+1})$, que capta as restrições sobre os valores correspondentes ao contador de programa e as variáveis do programa.

Considerando um sistema de transição M, uma propriedade de segurança ϕ, um limite de contexto C, e um limite k, ESBMC constrói uma Árvore de Alcançabilidade(AA), que representa o desenrolar do programa para C, k e ϕ. Uma condição de verificação(CV) ψ_k^π é derivada para cada intercalação dada $\pi = \{v_1, \ldots, v_k\}$, tal que ψ_k^π é satisfatível *sse* ϕ tem um contraexemplo de comprimento k, que é exibido por π. ψ_k^π é dado a seguir por

$$\psi_k^\pi = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \neg\phi(s_i) \tag{2.1}$$

I caracteriza o conjunto de estados iniciais de M e $\gamma(s_j, s_{j+1})$ é a relação de transição de M entre os passos j e $j+1$. Uma vez que $I(s_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ representa a execução de M de comprimento i e ψ_k^π pode ser satisfeito se, e somente se, para algum $i \leq k$ onde exista um estado alcançável ao longo de π no passo i onde ϕ é violado. ψ_k^π é uma fórmula livre de quantificadores em um subconjunto de decisões de lógica de primeira ordem, que é verificado para satisfatibilidade por um solucionador SMT. Se ψ_k^π é satisfatível, então ϕ é violado ao longo de π e o solucionador SMT fornece uma atribuição satisfatória, da qual os valores das variáveis do programa podem ser extraídas, para construir um contraexemplo. Um contraexemplo para uma determinada propriedade ϕ é uma sequência de estados $s_0, s_1, \ldots, s_k$ com $s_0 \in S_0$, $s_k \in S$, e $\gamma(s_i, s_{i+1})$ para $0 \leq i < k$. Se ψ_k^π não é satisfatível, pode ser concluído que não existe nenhum estado de erro em k passos ou menos ao longo de π. Finalmente pode ser definido $\psi_k = \bigwedge_\pi \psi_k^\pi$ e usar isto para verificar todos os caminhos.

ESBMC combina verificação simbólica de modelos com exploração explícita de estados. Essa explicitude explora as possíveis intercalações, sobre um dado limite de contexto, enquanto trata cada intercalação simbolicamente. ESMBC percorre a árvore de alcançabilidade primeiro em sua profundidade e chama os procedimentos de BMC para *threads* simples sobre a

intercalação sempre que alcançar um nó folha da árvore. O ESBMC para quando encontra um *bug* ou tiver sistematicamente explorado todas as possíveis intercalações da árvore.

2.3.1 Arquitetura do ESBMC

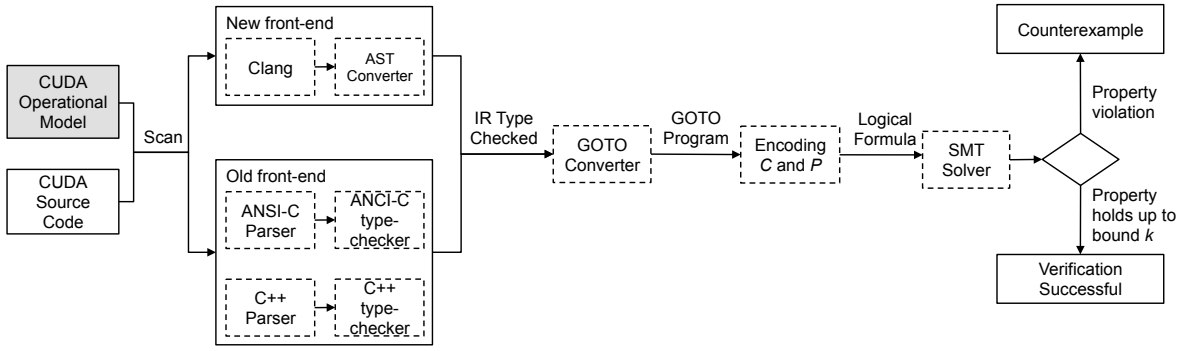

Figura 2.1: Arquitetura do ESBMC. Os retângulos branco representam entradas e saídas. O retângulo cinza representa passos de verificação.

A Fig. 2.1 exibe a arquitetura do ESBMC. O ESBMC propõe dois *front-ends* para converter e gerar a Representação Intermediária(RI) do programa. No *front-end* antigo baseado no CBMC [22], o ESBMC deveria primeiro converter o programa usando flex/bison [23] e depois gerar uma Árvore de Sintaxe Abstrata(ASA) para ser convertida em RI. A RI passa por uma verificação de tipos específicos da linguagem. Durante a verificação de tipos o código passa por uma análise estática incluindo verificação de atribuições, *casting*, inicialização de ponteiro, chamadas de funções e instanciação de *templates*.

O novo *front-end* propõe uma alternativa mais simples e robusta, mas ainda não suporta a linguagem C++. É usado Clang [24] para gerar uma ASA, que é convertida para uma RI. Clang é um compilador estado-da-arte para C/C++/ObjectiveC/ObjectiveC++, o qual já é fortemente usado na indústria [25]. O compilador fornece um analisador estático e uma API para acessar e percorrer a ASA interna, usada pelo ESBMC para gerar sua RI. Por usar clang evitamos a necessidade de manter nosso próprio *front-end*, um desafio real dado que os padrões ANSI-C e C++ atualmente possuem uma rápida evolução, e podemos focar no objetivo principal da ferramenta: a verificação formal do programa. O ESBMC atualmente pode apenas converter a ASA gerada para programas ANSI-C.

Os passos subsequentes são os mesmos já descritos em trabalhos anteriores [5]. Independente da escolha do *front-end* a saída é a RI, usada pelo conversor GOTO para gerar o programa GOTO, que é uma versão simplificada do programa original. A execução simbólica

executa o programa GOTO(desenrolando laços sobre um limite k) e converte as expressões para **S**tatic **S**ingle **A**ssignments (SSA) [26]. Finalmente, dois conjuntos de fórmulas livres de quantificadores são criados baseados nas expressões SSA(C para restrições e P para propriedades), e são usados como entrada para um solucionador SMT, que produzirá um contraexemplo se houver uma violação de uma dada propriedade ou uma resposta insatisfatível(*i.e.*, verificado com sucesso) se não tiver violação da propriedade.

2.4 Resumo

Este capítulo descreve alguns conhecimentos necessários para entender o desenvolvimento da ferramenta de verificação de programas CUDA.

Na primeira seção é descrita a plataforma CUDA com sua maneira de realizar as configurações de blocos e *threads* e a quantidade destas que a GPU suporta, explicitando o desafio em desenvolver um padrão de comportamento para executar a verificação sobre estas aplicações. CUDA define blocos de *threads*, que processam em contextos independentes e intercaladamente sem uma ordem pré-definida. As *threads* dentro dos blocos executam de maneira concorrente e podem originar erros devido estarem em um mesmo contexto. O paradigma de interpretação das funções, que podem ser executadas em GPU's ou CPU's, também aumentam a complexidade de verificação. As funções de alocação de memória na GPU se assemelham às de alocação na arquitetura de memória das CPU's, demonstrando como podemos simular o armazenamento desses dados na execução dos cálculos pela GPU.

Também foi apresentada a teoria envolvida na ferramenta ESBMC para verificação de modelos de contexto limitado. A ferramenta utiliza solucionadores SMT para resolver fórmulas lógicas e encontrar a violação de propriedades sob um dado limite. Os programas são convertidos para uma Representação Intermediária, onde ocorrem verificações de tipo, a partir de uma Árvore de Sintaxe Abstrata. O passo seguinte é a representação do programa em GOTO e sua execução simbólica para gerar expressões SSA e finalmente construir os conjuntos de fórmulas livres de quantificadores que produzirá o resultado da verificação.

A combinação da plataforma CUDA e o ESBMC resulta na ferramenta ESBMC-GPU.

Capítulo 3

Trabalhos Relacionados

3.1 GPU Verify

A ferramenta GPUVerify [3] propõe uma semântica operacional nomeada **S**ynchronous **D**elayed **V**isibility(SDV) para verificar *kernels* com o objetivo de detectar condições de corrida e divergência de barreira. É usado o sistema de verificação Boogie [27] para gerar condições de verificação, que são solucionadas pelos solucionadores SMT Z3 [28] ou CVC4 [29]. GPUVerify suporta apenas a função *kernel* como entrada e desconsidera a função *main* do programa. Essa condição tem como consequência a exposição de resultados incorretos na verificação de aspectos de baixo nível em programas CUDA.

3.2 SESA e GKLEE

***S**ymbolic **E**xecutor with **S**tatic **A**nalysis (SESA)* [30] e GPU + KLEE (GKLEE) [8] são verificadores baseados na execução *concolic*(concreto simbólica) de programas CUDA. Assim mesmo, elas usam abordagens diferentes para determinar variáveis simbólicas. Enquanto SESA executa uma avaliação automática, GKLEE necessita de entradas do usuário para definir as variáveis. Por um lado, SESA verifica aplicações reais usando a configuração original do número de *threads* e objetiva a detecção de condições de corrida, mas apresenta resultados inconclusivos com respeito a acessos à memória. Por outro lado, GKLEE suporta verificações relacionadas a sincronização de barreira, corretude funcional, desempenho e condição de corrida. SESA não verifica a função *main*, enquanto GKLEE considera ambos *kernel* e função *main*.

3.3 PUG

***P**rover of **U**ser **G**PU Programs* (PUG) [7] analisa *kernels* automaticamente usando solucionadores SMT e detecta condições de corrida, sincronização de barreira e conflitos em memória compartilhada. PUG enfrenta problemas de derivação de invariantes para laços, que podem levar a resultados incorretos e assim requer do usuário o fornecimento destas. Problemas também são encontrados em operações aritméticas de ponteiros e propriedades avançadas de C++.

3.4 CIVL

CIVL [9] é um *framework* para análise estática e verificação de programas concorrentes que suporta MPI, POSIX, OpenMP e CUDA. CIVL reconhece programas escritos em C++, CUDA–C e CIVL–C, que é uma linguagem intermediária para representar programas ANSI-C baseados nas bibliotecas citadas. CIVL é a ferramenta que mais se aproxima ao ESBMC-GPU em termos de técnicas de verificação, analisando programas concorrentes com algoritmo de redução de ordem parcial para eliminar intercalações desnecessárias. As propriedades suportadas incluem: assertivas especificadas pelo usuário, *deadlocks*, vazamento de memória, referência inválida de ponteiro, acesso fora dos limites de vetores e divisão por zero. O suporte do CIVL para bibliotecas CUDA está ainda em progresso sendo efetivo e eficiente em programas com *kernels* os quais não usam muitas funcionalidades CUDA.

3.5 Resumo

Neste capítulo foram apresentadas as ferramentas de maior destaque na verificação de programadas CUDA. CIVL [9] é a ferramenta mais recente e a que mais aproxima dos passos de verificação realizados pelo ESBMC-GPU. GPU Verify [3] propõe sua semântica e usa solucionadores SMT assim como o ESBMC-GPU. SESA [30] e GKLEE [8] são ferramentas baseadas na execução concreto simbólica e cada uma tem seus pontos de destaque. PUG [7] é uma ferramenta mais antiga e que apresenta o maior número de limitações quando comparado a outras.

A Tabela 3.1 exibe o comparativo entre as metodologias aplicadas pelas ferramentas. Apesar de PUG ser baseado em verificação formal e CIVL aplicar redução de ordem parcial,

Ferramenta	**Metodologia**	**Solucionador**	**Suporte à CUDA**
ESBMC-GPU	BMC e redução de ordem parcial	Z3, Boolector, CVC4, Yices e MathSAT	*kernel* e *main*
GPU Verify	Semântica operacional(SDV)	Z3, CVC4	*kernel*
SESA	*Concolic*(concreto + simbólica)	-	*kernel*
GKLEE	*Concolic*(concreto + simbólica) e *canonical scheduling*	-	*kernel* e *main*
PUG	Verificação formal(LLNL Rose)	Yices	*kernel*
CIVL	Análise estática e redução de ordem parcial	-	*kernel* e *main*

Tabela 3.1: Comparativo das ferramentas.

o ESBMC-GPU é a única ferramenta a aplicar verificação formal com BMC e redução de ordem parcial com o algoritmo MPOR. GPU Verify aplica sua própria metodologia de semântica operacional e SESA e GKLEE seguem a mesma linha de usar a metodologia *concolic*. ESBMC-GPU também oferece o maior suporte a solucionadores SMT dentre as ferramentas que aplicam esta abordagem. Além disso, ESBMC-GPU, GKLEE e CIVL suportam a verificação completa dos programas por analisarem tanto a função *kernel* quanto a função *main*, ressaltando que o ESBMC-GPU suporta o maior número de bibliotecas CUDA.

Capítulo 4

Verificando Programas CUDA

4.1 Modelos Operacionais para Bibliotecas CUDA

Os modelos operacionais foram desenvolvidos para suportar as funcionalidades e simular fielmente o comportamento das bibliotecas CUDA. Cada abordagem foi previamente analisada na verificação formal de programas C++ [6], aplicações Qt [31, 32] e aplicações Android para dispositivos móveis [33, 34]. O modelo operacional consiste na representação abstrata de um conjunto de métodos e estruturas de dados os quais mantêm aproximadamente a semântica das bibliotecas de CUDA. Todo método simula o comportamento real da biblioteca, onde são incluídas pré e pós-condições através de assertivas, para garantir a corretude da operação. Assim, o modelo operacional contém apenas métodos para verificação, ignorando chamadas irrelevantes (*e*.g., métodos de impressão na tela) onde não há propriedade a ser verificada em termos de software. Como resultado a verificação tem foco no modelo operacional das bibliotecas CUDA, e, como é usado para verificar programas CUDA do mundo real, isso simplifica significativamente o modelo e consequentemente reduz o tempo de verificação. O modelo operacional também inclui assertivas automaticamente, que verificam propriedades específicas (*e*.g., acesso fora dos limites de matrizes, estouro aritmético, segurança de ponteiros e condições de corrida). Como mostrado na Fig. 4.1, o modelo operacional é passado ao *front-end* do ESBMC junto ao código fonte a ser verificado para produzir a ASA com toda informação relevante para o processo de verificação.

No Modelo Operacional CUDA(MOC), métodos, tipos de dados, qualificadores e diretivas de CUDA foram implementados em C++. Como exemplo, a Figura 4.2 mostra a imple-

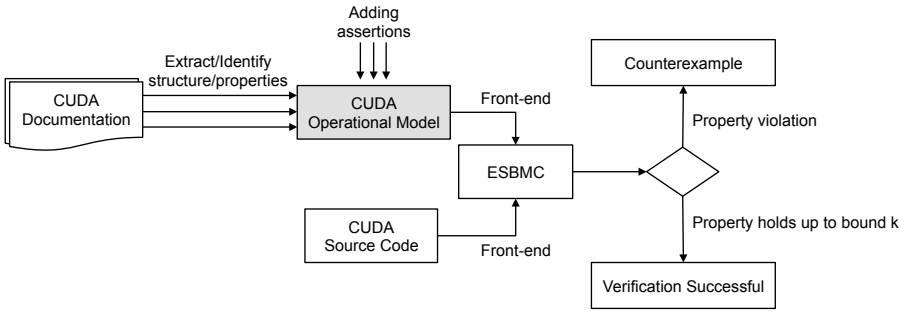

Figura 4.1: Arquitetura entre o Modelo Operacional CUDA e o ESBMC.

mentação do tipo de dado *dim3*, que é representado por uma *struct* com três variáveis: *x*, *y* e *z*. O modelo *dim3* tem um construtor padrão, um construtor o qual recebe uma variável *uint3* como parâmetro(esta variável é modelada por uma *struct* com três variáveis do tipo *uint*) e um construtor o qual é inicializado por uma variável *dim3*. Além dos tipos primitivos de dados de ANSI-C/C++, o MOC suporta tipos de dados específicos de CUDA (*e.g.*, *char1*, *short2* e *float2*).

```
1  struct __dim3
2  {
3    unsigned int x, y, z;
4    __dim3(unsigned int vx=1, unsigned int vy=1, unsigned int vz=1){
5      x=vx; y=vy; z=vz;
6    }
7    __dim3(uint3 v){
8      x=v.x; y=v.y; z=v.z;
9    }
10   __dim3(__dim3 d){
11     x=d.x; y=d.y; z=d.z;
12   }
13   operator uint3(void){
14     uint3 t; t.x=x; t.y=y; t.z=z;
15   return t;
16   }
17 };
18 typedef struct __dim3 dim3;
```

Figura 4.2: Implementação da *dim3*.

O MOC também modela métodos usados para desenvolver aplicações reais de CUDA, onde são suportados mais *drivers* CUDA e API's de tempo real: ESBMC-GPU suporta a chamada de *kernel* CUDA (*device_launch_parameters* e *vector_types*), driver_types (*math_functions*, *cuda_runtime_api* e *host_definitions*), sm_atomic_functions (*vector_types*) e curand_kernel (*curand*). A Figura 4.3 mostra a organização do MOC(internamente) implementado no ESBMC-GPU.

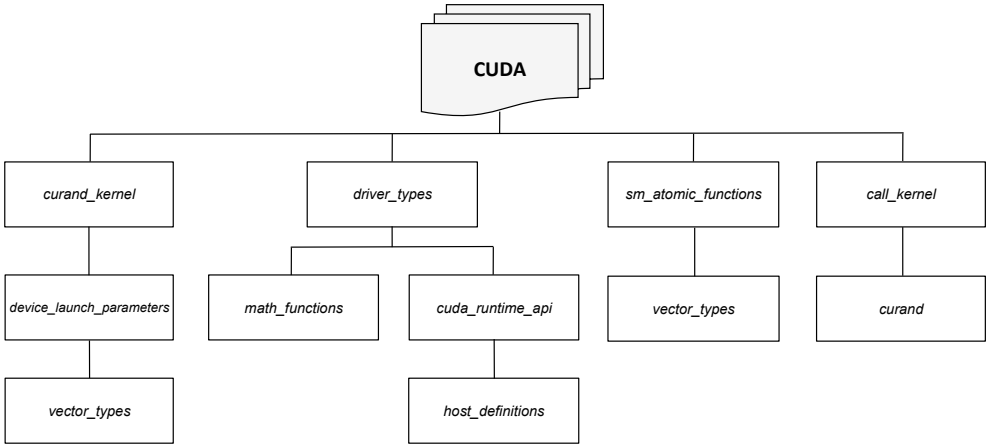

Figura 4.3: Organização do Modelo Operacional CUDA(MOC).

Existem três funções essenciais na implementação de um programa CUDA: *cudaMalloc*, *cudaMemcpy* e *cudaFree*. Essas funções são usadas para alocação, desalocação e cópia de dados para GPU. O Algoritmo 1 mostra um exemplo de um modelo operacional desenvolvido para a função *cudaMalloc*, que abstrai a hierarquia de memória da GPU e aceita como argumento de entrada um ponteiro para alocar memória no dispositivo(*i*.e., *devPtr*) e o tamanho em *bytes* necessários para alocação (*i.e.*, *size*). Para alocar memória no ponteiro *devPtr*(no passo 3) este algoritmo usa a função *malloc*, que representa a alocação de memória no *device*, e verifica se houve sucesso(passo 4) retornando *CUDA_SUCCESS*. Caso contrário, a função retorna um erro especificado por *CUDA_ERROR_OUT_OF_MEMORY*. A variável *lastError* é de escopo global e armazena o último valor de *cudaError_t* para ser usado pela função *cudaLastError()*. Além disso, a função *cudaMalloc()* tem como pré-condição o tamanho da alocação de memória positivo. No passo 2 do Algoritmo 1, uma assertiva é incluída para que o tamanho alocado seja maior do que zero e, caso exista uma violação nesta pré-condição, o ESBMC-GPU retorna uma mensagem especificando o erro e fornecendo um contra-exemplo. Como pós-condição, é verificado se a memória foi alocada corretamente (passo 7).

O Algoritmo 2 apresenta o modelo operacional da função $cudaMemcpy()$. Como pré-condição é verificado o tamanho de memória a ser copiada (passo 2). Em seguida duas variáveis locais, *cdst* e *csrc*, recebem argumentos que representam o destino e a origem do dado a ser copiado (passos 3 e 4). Este modelo define o número de *bytes* a serem copiados (passo 5). A cópia do dado é realizada (passo 6) entre *device* e *host*. Como pós-condição é verificado se *cdst* e *csrc* contém o mesmo dado (passo 7). Finalmente, a função *cudaMemcpy* retorna *CUDA_SUCCESS*.

Algoritmo 1 Algoritmo da função *cudaMalloc*.

1: **Início** CUDAMALLOC(***void*** ** *devPtr,* ***size_t*** *size*)
2: Verifica pré-condição: *size* deve ser maior que zero
3: Aloca um bloco de memória de magnitude igual a *size*, para o ponteiro $devPtr$
4: Se $*devPtr$ é igual a `NULL`, vai para o passo 5, caso contrário passo 6
5: Atribui `CUDA_ERROR_OUT_OF_MEMORY` para a variável global $lastError$
6: Atribui `CUDA_SUCCESS` para a variável global $lastError$
7: Verifica pós-condição: $devPtr$ deve ser diferente de NULL
8: Retorna $lastError$ e finaliza
9: **Fim**

Algoritmo 2 Função *cudaMemcpy*.

1: **Início** CUDAMEMCPY(***void*** **dst,* ***const void*** **src,* ***size_t*** *size,* ***cudaMemcpyKind*** *kind*)
2: Verifica pré-condição: *size* deve ser maior que zero
3: Inicializa a variável *cdst* com o conteúdo de *dst*
4: Inicializa a variável *csrc* com o conteúdo de *src*
5: Inicializa a variável *numbytes* com o número de bytes a serem copiados (*i.e.*, *size*)
6: Copia *numbytes* posições de *csrc* para *cdst*
7: Verifica pós-condição: *cdst* e *csrc* deve conter o mesmo dado
8: Atribui `CUDA_SUCCESS` para a variável global $lastError$
9: Retorna $lastError$ e finaliza
10: **Fim**

Similar a *cudaMalloc* e *cudaMemcpy*, foi desenvolvido o modelo operacional da função *cudaFree*. O Algoritmo 3 exibe o modelo onde um ponteiro para uma variável é passado como argumento de entrada tal que a memória alocada seja liberada. Como pré-condição é verificado se o ponteiro $devPtr$, que aponta para o bloco de memória previamente alocado por *malloc*, é não nulo para evitar a dupla desalocação de memória. Além disso, pode ser observado(no passo 3) que a função *cudaFree* é referenciada à função *free* da linguagem de programação ANSI-C. Neste caso, a memória é liberada pela GPU. Esta situação não afeta os resultados da verificação, devido o modelo de alocação e desalocação de memória ser representado por uma tupla nas teorias (suportadas) dos solucionadores SMT, e as propriedades analisadas não consideram a hierarquia de memória [35]. Finalmente, é atribuído *CUDA_SUCCESS* para *lastError*, que tem a mesma função como em *cudaMalloc*.

Note que os modelos operacionais são implementados de acordo com o Guia de Programação NVIDIA [11]. O comportamento das funções pode ser representado por funções nativas das linguagens de programação C/C++ que são suportadas pelo ESBMC (*e*.g., *malloc*, *free*, *assert*). A prova da corretude destas funções nativas suportadas pelo ESBMC, pode ser encontrada em Cordeiro *e*t al. [5]. Como exemplo, a função *cudaMalloc* opera similarmente à função

Algoritmo 3 Função *cudaFree*.

1: **Início** CUDAFREE(***void** *devPtr*)
2: Verifica pré-condição: $devPtr$ deve ser diferente de NULL
3: Executa a função ANSI-C $free$ para desalocar a memória do ponteiro $devPtr$
4: Atribui $CUDA\_SUCCESS$ a variável global $lastError$
5: Retorna $lastError$ e finaliza
6: **Fim**

malloc, que recebe como argumento de entrada o tamanho da variável a ser alocada e tem o seu comportamento de acordo com a semântica de ANSI-C. A diferença conceitual para programas CUDA está na alocação de memória realizada na GPU, onde são abstraídas tanto as funções de hardware quanto a hierarquia de memória, que não são consideradas no modelo operacional assim como em [3, 7, 8]. A função *cudaMemcpy* é implementada similarmente à função *Memcpy* onde a única diferença está em um parâmetro adicional o qual determina se a operação é de *device* para *host* ou vice-versa.

4.1.1 Corretude do Modelo Operacional CUDA (MOC)

A ideia de criar um ambiente com modelo operacional para executar a verificação de aplicações reais, tem sido aplicada em outros trabalhos [31, 32, 33, 34]. A confiança na corretude dos modelos é atualmente a principal questão e, consequentemente, a utilidade desta abordagem depende do fato de que o MOC representa corretamente as bibliotecas CUDA. Neste sentido, tudo desenvolvido no MOC foi manualmente verificado e exaustivamente comparado com os originais para garantir o mesmo comportamento. Todos contêm pré-condições e pós-condições para garantir que um predicado esteja correto antes e depois da execução de uma função, respectivamente.

Além disso, embora o MOC seja uma nova implementação, ele consiste em construir fielmente um modelo simplificado de bibliotecas CUDA, usando a mesma linguagem através do código e documentação originais (pré e pós-condições são testadas usando assertivas dentro do próprio código), o que desta maneira tende a reduzir o número resultante de erros. Note que o comportamento das funções das bibliotecas CUDA são representadas nas linguagens C/C++ usando funções nativas(*e*.g., *malloc*, *free*, *assert*). A prova de corretude para estas funções nativas, que são suportadas pelo ESBMC, podem ser encontradas em Cordeiro *e*t al. [4, 5]. Embora as provas com respeito a corretude de todo o MOC possa ser demonstrada, isto

representa uma difícil tarefa devido ao modelo de memória [36] adotado. Além de melhorar a corretude do MOC, testes de conformidade podem ser empregados na prática [37, 38]. Contudo, esta opção não está disponível no caso atual, embora seja um interessante possibilidade de trabalho futuro.

4.2 Modelando o Kernel com Funções Pthread

A arquitetura do ESBMC foi desenvolvida para manipular programas concorrentes nas linguagens C/C++ usando a biblioteca Pthread/POSIX [39]. A verificação de modelo do ESBMC é direcionada por métodos de processamento usados pela CPU através desta biblioteca, onde as instruções de uma *thread* podem intercalar para formar (diferentes) caminhos de execução. Para aplicar esta metodologia à verificação de *kernels* CUDA, é necessário realizar transformações de código para executar o *kernel* usando funções intrínsecas do ESBMC. As execuções do *kernel* fornecem a configuração de *thread*/bloco em um programa CUDA e os parâmetros usados para a função intrínseca são obtidos da mesma *struct* de chamada do *kernel*. Como resultado, o primeiro passo para verificar um *kernel* CUDA é criar a nova função *ESBMC_verify_kernel* usando *templates* para suportar diferentes tipos de dados e parâmetros.

Algoritmo 4 Função *ESBMC_verify_kernel*.

1: **Início** ESBMC_VERIFY_KERNEL(***RET*** **kernel,* ***BLOCK*** *blocks,* ***THREAD*** *threads,* ***T1*** *arg1,* ***T2*** *arg2,* ***T3*** *arg3*)
2: Verifica pré-condição: *kernel* deve ser diferente de NULL, *blocks* e *threads* devem ser maior do que zero
3: Inicializa a variável *gridDim* com a dimensão das *threads* usando a função *dim*3
4: Inicializa a variável *blockDIM* com a dimensão dos blocos(*blocks*) usando a função *dim*3
5: Executa a função $ESBMC\_verify\_kernel\_wta$ para determinar o tipo de $*kernel$
6: Executa a função ANSI-C *pthread_join* para cada *thread* GPU e finaliza a execução
7: **Fim**

O algoritmo 4 mostra a implementação da função *ESBMC_verify_kernel*, que suporta seis parâmetros de entrada. O parâmero *kernel* é um ponteiro para a função *kernel* original do programa. Os parâmetros *blocks* e *threads* recebem a configuração de blocos e *threads*, que podem ser do tipo *int* ou *dim3*. Os parâmetros *arg1*, *arg2* e *arg3* correspondem aos valores enviados por suas respectivas funções. São verificadas pré-condições como se *kernel* é um ponteiro válido e se *blocks* e *threads* são maiores do que zero. Internamente existem duas

variáveis, *gridDim*(passo 3) e *blockDim*(passo 4), que recebem a saída do construtor *dim3* usado para configurar a dimensão dos blocos e *threads*. A função *ESBMC_verify_kernel_wta*(passo 5) particulariza o tipo de argumento para *int* e determina o tipo de ponteiro da função *kernel*. No fim do processamento do *kernel*, há o laço(passo 6) que sincroniza, por meio da função *pthread_join*, as *threads* que foram criadas. A função *ESBMC_verify_kernel* é implementada para operar com o número real de *threads* no programa, contudo, com a redução para a análise de duas *threads*(veja Seção 4.4), o respectivo laço no passo 6 é limitado a duas iterações.

Algoritmo 5 Função *ESBMC_verify_kernel_wta*.

1: **Início** ESBMC_VERIFY_KERNEL_WTA(***void*** **(*kernel)(int*,int*,int*),* ***int*** *blocks,* ***int*** *threads,* ***void*** **arg1,* ***void*** **arg2,* ***void*** **arg3*)
2: Aloca um bloco de memória para o ponteiro *threads_id*, que armazena os ID's das *threads*
3: Aloca um bloco de memória para cada componente de *dev_three*
4: Verifica pré-condição: *threads_id* e *dev_three* deve ser diferente de NULL
5: Atribui *arg*1, *arg*2 e *arg*3 para cada componente de *dev_three*, respectivamente
6: Atribui $*kernel$ à componente do ponteiro de função *dev_three*
7: Executa a função *assignIndexes*
8: Executa a função ANSI-C *pthread_create* para criar cada GPU *thread* e finaliza
9: **Fim**

O Algoritmo 5 mostra o raciocínio por trás da função *ESBMC_verify_kernel_wta*, que é responsável por traduzir *threads* GPU em *threads* POSIX. O ponteiro *threads_id*(passo 2) armazena os ID's das *threads* do tipo *pthread_t*. A variável *dev_three*(passos 3 ao 6) é uma instância de *struct* e armazena um ponteiro para função *kernel* e a função *assignIndexes*(passo 7) calcula a posição da *thread* no *grid* usando o ID da *thread*(*pthread*) e a configuração de blocos e *threads* no programa CUDA. Os valores são armazenados em um vetor *dim3* e são estaticamente calculados para reduzir os caminhos gerados durante a verificação. Por último, há um laço(passo 8) onde *threads* são criadas pela chamada da função *pthread_create* e, a função que corresponderá a cada *thread*, é chamada via *ESBMC_execute_kernel_t*.

Algoritmo 6 Função *ESBMC_execute_kernel_t*.

1: **Início** ESBMC_EXECUTE_KERNEL_T(***void*** **args*)
2: Verifica pré-condição: *args* deve ser diferente de NULL
3: Executa a componente do ponteiro de função *dev_three* (*i.e.*, $dev\_three.func$) e finaliza
4: **Fim**

O Algoritmo 6 mostra a implementação da função *ESBMC_execute_kernel_t*. A função original do *kernel* é verificada com seus parâmetros que são enviados como argumento pela

struct dev_three. Assim, o ESBMC-GPU verifica a função e intercala seus possíveis caminhos de execução usando suas funções nativas.

4.2.1 Sincronização de Threads

Na abordagem proposta neste trabalho, cada *thread* CUDA é mapeada a uma representação de *thread* interna do ESBMC(via biblioteca *Pthread*) [4]. No ESBMC, a *thread t* é uma sublista de comandos entre *begin_thread* e *end_thread*, que representa apenas a construção do escopo, e não contribui para a expansão da árvore de alcançabilidade. *Threads* são criadas via chamadas de procedimento assíncronas(*start_thread*), que retornam um inteiro que pode ser usado com identificador da *thread* na sincronização(*join_thread*), uma vez que a criação dinâmica da *thread* é permitida. Assim, a representação da *thread* no ESBMC segue uma abordagem similar ao escalonador oficial de CUDA [1, 2, 3].

Uma característica de CUDA relacionada a sincronização das *threads* suportada pelo ESBMC-GPU é a função intrínseca *__syncthreads()*, que especifica pontos de sincronização em um *kernel* GPU. Tal alternativa garante que *threads* em um determinado bloco esperem todas as outras antes de procederem [1]. Como descrito no Algoritmo 7, esta característica foi implementada no modelo operacional usando funções *PThread*. Na implementação uma variável global *count* é inicializada para controlar quantas vezes a função *syncthreads* é executada. Na primeira execução, um mutex é inicializado e uma variável *pthread* condicional é definida(passos 2 ao 5). Logo, para cada execução subsequente, a variável *count* é incrementada(passo 6) e a *thread* executora é colocada em espera até receber o sinal(passo 8). Quando todas as *threads* GPU estão em espera, a função *pthread_cond_signal* envia um sinal para todas as *threads* esperando uma condição específica(passo 9) e o mutex é destravado(passo 10).

4.2.2 Exemplo Ilustrativo

O fragmento de código mostrado na Fig. 4.4 tem 1 bloco e 2 *threads*, *i*.e., $M = 1$ e $N = 2$, respectivamente. Este programa CUDA tem um *kernel*(linhas 4 a 6), o qual atribui os valores dos índices das *threads* para um vetor passado como argumento de entrada. A meta é instanciar as posições do vetor de acordo com o índice da *thread*. Nesta operação há um erro no índice do vetor devido ao valor 1 ser acidentalmente adicionado ao índice da *thread*(linha 5). Como mostrado na função principal, as posições dos vetores são atribuídos valores 0(linha

Algoritmo 7 Função *__syncthreads*.

1: **Início** __SYNCTHREADS( )
2: Se *count* é igual a 0, executa os passos 3, 4 e 5; caso contrário vá para o passo 6
3: Executa a função *pthread_mutex_init* para inicializar o mutex
4: Executa a função *pthread_cond_init* para inicializar a variável de condição
5: Executa a função *pthread_mutex_lock* para travar o mutex indicado
6: Incrementa a variável *count*
7: Se *count* é diferente do número de *threads* GPU, vá para o passo 8, caso contrário vá para o passo 9
8: Executa a função *pthread_cond_wait* para bloquear a *thread* indicada até que ela receba o sinal
9: Executa a função *pthread_cond_signal* para todas *threads* GPU
10: Executa a função *pthread_mutex_unlock* e finaliza
11: **Fim**

14) e depois da chamada do *kernel*(linha 16) é esperado pelo programador que $a[0] == 0$ e $a[1] == 1$.

Neste exemplo, o ESBMC-GPU detecta uma violação dos limites do vetor. De fato, este programa CUDA requisita uma região de memória a qual não foi previamente alocada tal que, quando $threadIdx.x = 1$, o programa tenta acessar a posição $a[2]$. Analisando o modelo operacional da função *cudaMalloc()* existe uma pré-condição que verifica se o tamanho de memória a ser alocada é maior que zero. Assertivas verificam se o resultado do modelo atinge as pós-condições esperadas(linha 19). A verificação deste programa específico produz 34 intercalações de sucesso e 6 de falha no ESBMC-GPU. Uma possível intercalação de falha é representada pela execução das *threads* $t0 : a[1] = 0; t1 : a[2] = 1$, onde $a[2] = 1$ representa um acesso incorreto ao índice do vetor a.

ESBMC-GPU e GKLEE são capazes de detectar esta violação do limite de vetor. GPUVerify e PUG falham na detecção desta violação apresentando um 'verdadeiro incorreto'(não detecção de erro) e CIVL não suporta o programa por não identificar *threadIdx*.

A função *cudaMalloc*(linha 12) verifica a pré-condição para identificar se *size* é maior que zero. Internamente, a alocação da área de memória é convertida para uma função *malloc*, que é interpretada pelo ESBMC-GPU como uma função ANSI-C. A função *cudaMemcpy*(linhas 15 e 17) também verifica se *size* é maior do que zero e, caso verdadeiro, executa uma cópia usando a função *memcpy*.

A função *ESBMC_verify_kernel* recebe como parâmetro o nome da função kernel(*kernel*), o número de blocos(M) e *threads*(N) e o parâmetro *int dev_a*. Esta função executa a configu-

```
1  #define M 1
2  #define N 2
3
4  __global__ void kernel(int *A) {
5    A[threadIdx.x + 1] = threadIdx.x;
6  }
7
8  int main(){
9    int *a; int *dev_a;
10   int size = N * sizeof(int);
11   a = (int*) malloc(size);
12   cudaMalloc((void**)&dev_a, size);
13   for (int i = 0; i < N; i++)
14       a[i] = 0;
15   cudaMemcpy(dev_a,a,size, cudaMemcpyHostToDevice);
16   ESBMC_verify_kernel(kernel, M, N, dev_a);
17   cudaMemcpy(a,dev_a,size, cudaMemcpyDeviceToHost);
18   for (int i = 0; i < N; i++)
19       assert(a[i]==i);
20   cudaFree(dev_a);
21   free(a);
22 }
```

Figura 4.4: Trecho de código para indexar o vetor.

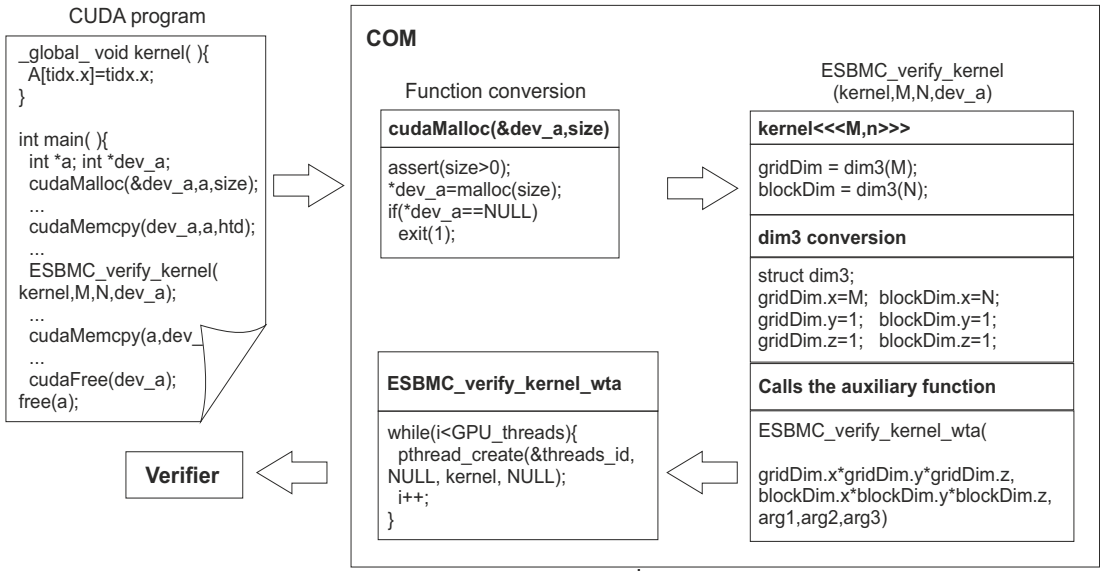

Figura 4.5: Passos de conversão do MOC.

ração do *grid*, criando uma representação *dim3* e salvando nas variáveis *gridDim* e *blockDim*. Neste ponto, a função *assignIndexes* é executada para criar um vetor com os ID's das *threads* pré-processados evitando uma explosão de estados nos caminhos de verificação. Um laço dispara as *threads* com *pthread_create* e depois conclui sua execução. A *thread pthread_join* sincroniza e finaliza a execução da *ESBMC_verify_kernel*. A função *assert* verifica se existe

qualquer valor inesperado no *kernel*. Finalmente, a função *cudaFree* libera a memória alocada pelas variáveis e verifica a ocorrência de erros(*e*.g., liberação de memória não alocada).

A Figura 4.5 mostra os passos executados pela ferramenta ESBMC-GPU para verificar o programa mostrado na Figura 4.4. As instruções da função principal são modeladas e convertidas para funções nativas de C/C++ juntamente com o MOC. As variáveis e funções C/C++ são também reconhecidas pelo ESBMC-GPU não sendo necessário convertê-las(*e*.g., linhas 10 e 11). As funções CUDA *cudaMalloc*, *cudaMemcpy* e *cudaFree* e a *ESBMC_verify_kernel* são parte do MOC e assim são convertidas para C/C++ e *threads* POSIX.

4.3 Redução Monotônica de Ordem Parcial

Para reduzir o número de intercalações das *threads* nos programas CUDA, o ESBMC-GPU implementa a Redução Monotônica de Ordem Parcial (MPOR). Essa técnica foi inicialmente proposta por Kahlon *e*t al. [10] e sua implementação no ESBMC-GPU é herdada do seu predecessor ESBMC [40]. Este algoritmo classifica as transições dentro de um programa *multi-thread* como dependente ou independente, que determina se os pares de intercalações sempre computam o mesmo estado, resultando na remoção de estados duplicados na árvore de alcançabilidade. Para transições dependentes, o MPOR considera as possíveis ordens da execução de uma *thread* para garantir que todos os estados do programa são alcançados. Se uma transição é independente, o algoritmo MPOR considera apenas uma ordem pois o estado do programa é o mesmo para outras ordens de execução. Como o MPOR depende do próximo estado em relação ao estado atual, o ESBMC-GPU verifica o estado anterior a partir do estado atual para identificar se existe qualquer dependência entre as *threads*.

No ESBMC-GPU, a implementação do MPOR é aplicada para identificar acessos a diferentes posições em vetores compartilhados. Tipicamente as *threads* acessam posições únicas nos vetores globais(*global*), que não têm dependência entre si permitindo remover estados redundantes gerados pelas possíveis ordens de execução. Baseado nesta observação, o algoritmo MPOR foi estendido para identificar transições em que *threads* acessam diferentes posições de um vetor. Este tipo de acesso resulta no mesmo estado independentemente da ordem de execução da *thread*. Múltiplos acessos a posições específicas de memória em programas CUDA acontecem devido a sua natureza concorrente baseada na configuração de linearização das *threads* e blocos[1].

A seguir é descrito o algoritmo MPOR para identificar transições nas quais *threads* acessam diferentes posições de um vetor e também a execução usando um exemplo. Seja Π a árvore de alcançabilidade, cada nó v em Π seja representado como uma tupla $v = (A_i, C_i, s_i)$ para um dado passo i, onde A_i representa a *thread* ativa. C_i representa o número da troca de contexto e s_i representa o estado atual. O Algoritmo 8 mostra os passos principais do MPOR no ESBMC-GPU.

Algoritmo 8 MPOR para identificar múltiplos acessos em vetores compartilhados.

1: **Início** MPOR(v, Π)
2: Verifica se s_i existe em Π; caso contrário, segue para o passo 4.
3: Verifica se A_i tem acesso de leitura/escrita para s_i; caso contrário, segue para o passo 5
4: Analisa se $\gamma(s_{i-1}, s_i)$ é independente em Π; caso contrário, segue para o passo 6
5: Retorna "independente" em Π e finaliza.
6: Retorna "dependente" em Π e finaliza.
7: **Fim**

Como exemplo, a Fig. 4.6(a) mostra a aplicação do MPOR em um *kernel* CUDA onde a variável global a é escrita em uma posição relativa ao ID da *thread*. Esta variável determina o estado de execução do programa e, na primeira intercalação, a *thread* t_1 alcança v_1 resultando em $a = [0,0]$. A *thread* t_2 executa e alcança v_2 resultando em $a = [0,1]$. O algoritmo MPOR verifica se v_2 não existe(passo 2) e se a transição $t_1 \rightarrow t_2$ é definida como independente(passo 4). Ambas condições não ocorrem e a transição $t_1 \rightarrow t_2$ é definida como dependente(passo 6). Assim o ESBMC-GPU executa todas as intercalações possíveis e a próxima execução começa com a *thread* t_2 a partir de v_0 que altera o conteúdo do vetor para $a = [0,1]$. A *thread* t_1 executa e alcança v_4 que é similar a v_2. O Algoritmo 8 verifica se a *thread* t_1 alcançou um estado redundante(passo 2). Como a *thread* t_1 alcança somente v_1, o algoritmo conclui que a transição $t_2 \rightarrow t_1$ é independente e esta é desconsiderada(representado na Fig. 4.6(a) por linhas pontilhadas).

A Fig. 4.6(b) mostra outro exemplo onde a execução das *threads* resulta em estados diferentes. Na primeira intercalação, a execução da *thread* t_1 alcança v_1 e a execução da *thread* t_2 alcança v_2. Desde que o conteúdo de $a[1]$ é diferente em ambos os nós da árvore de alcançabilidade, a condição(passo 2) não ocorre. Na segunda intercalação, a *thread* t_2 modifica o vetor a para $a = [0,1]$ e a *thread* t_1 acidentalmente escreve em $a[2]$. O Algoritmo 8 verifica se v_4 existe em Π(passo 2) e, desde que a condição(passo 4) não ocorre, a transição $t_2 \rightarrow t_1$ é definida como dependente. Neste caso, ambas intercalações são consideradas e estas duas ordens de execução

resultam em uma violação de acesso fora do limite do vetor devido ao vetor *a* ter comprimento 2 (*e.g.*, v_4 em linhas pontilhadas).

Na avaliação experimental, é observado que a aplicação do MPOR para programas CUDA leva a uma melhoria substancial no desempenho do ESBMC-GPU. O principal motivo para esta melhoria é que a implementação do MPOR simbolicamente codifica as intercalações apenas se as transições são definidas como dependente. Quando as fórmulas SMT são construídas o ESBMC-GPU descarta as intercalações que resultam em estados redundantes e apenas envia ao solucionador SMT as intercalações com transições dependentes, resultando em novos estados do programa. Com a análise MPOR, a primeira intercalação é definida e as intercalações seguintes serão consideradas apenas se não forem redundantes(a prova está descrita em Kahlon *e*t al. [10]).

```
1 __global__ void kernel1(int *a) {
2   a[threadIdx.x] = threadIdx.x;
3 }
```

```
1 __global__ void kernel2(int *a) {
2   if(a[1]==1){
3     a[threadIdx.x+2] = threadIdx.x;
4   }
5   else { a[threadIdx.x] = threadIdx.x; }
6 }
```

Figura 4.6: MPOR aplicado ao *kernel* com transições independentes (Fig. 4.6a) e dependentes (Fig. 4.6b).

4.3.1 Diferenças entre MPOR e Outros Métodos de Redução de Intercalações

Existem softwares de verificação, que aplicam métodos de redução de intercalações para verificação de programas CUDA. Como exemplo, GKLEE implementa uma técnica chamada *canonical scheduling*, que registra todas as leituras e escritas realizadas por cada *thread* dentro de um intervalo de barreira antes de continuar para a próxima *thread*. Os registros são relacionados em pares que podem ter conflitos e, por meios de um solucionador SMT, são verificados

para detectar se condições de corrida existem. Se não for detectado conflitos, o *canonical schedule* é definido para representar o intervalo de barreira para a verificação de outras propriedades. O algoritmo MPOR executa sua análise e redução durante a verificação e analisa a violação de propriedades por codificar e analisar pequenas fórmulas SMT. GKLEE primeiro verifica condição de corrida para reduzir escalonamentos e então verifica outras propriedades. É observado que ambas abordagens executam diversas vezes seu motor de verificação em *back-end*, sendo o GKLEE com *canonical scheduling* e o ESBMC-GPU com BMC e *Pthread* usando MPOR.

4.4 Análise por Duas *Threads*

A arquitetura da GPU é composta por multiprocessadores construídos sobre conjuntos de elementos de processamento(PE) [1, 11]. Estes PE's são arranjados em subgrupos(chamados *warps*), os quais executam em um mesmo *lockstep*, garantindo que *threads* dentro dos PE's possam sincronizar usando barreiras, enquanto *threads* de um subgrupo executam independentemente [3] de *threads* de outro subgrupo.

Similar ao GPUVerify [3](para verificar a ausência de condições de corrida e divergência de barreira) e PUG [7], ESBMC-GPU também reduz o programa CUDA à verificação de duas *threads* para reduzir o tempo de verificação e evitar o problema da explosão do espaço de estados. Desde que *kernels* CUDA normalmente manipulam um elemento de um vetor, e para cada elemento uma *thread* é usada, a análise de duas *threads* garante que erros(*e*.g., condições de corrida) detectados entre duas *threads* de um determinado subgrupo, devido aos acessos não sincronizados a variáveis compartilhadas, são suficientes para justificar a violação de propriedades no programa.

Para demonstrar a ideia sobre a análise de duas *threads*, uma arquitetura NVIDIA Fermi GPU [11] composta por 32 núcleos de processamento é mostrada na Figura 4.7. Na Figura 4.7a é mostrado o modelo de arquitetura desta GPU com 16 núcleos de processamento(destacados nas caixas cinzas), porque o processamento das *threads* é executado em *half-warps* [41], *i*.e., 16 *threads* do mesmo *warp* são escalonadas, garantindo a sincronização quando uma barreira for encontrada. A Figura 4.7b representa as *threads* que acessam uma memória compartilhada. Primeiramente, as caixas brancas numeradas representam as posições de memória compartilhada a serem acessadas, ${t_r}^n$ são as *threads* de leitura, e ${t_w}^n$ são as *threads* de escrita e n é

(a) (b)

Figura 4.7: Comparação entre a arquitetura da GPU NVIDIA Fermi e a abordagem de duas *threads* usada para manipular dados.

o índice de cada *thread*. É notório que as *threads* são executadas em um multiprocessador de *streaming*. No primeiro caso da Figura 4.7b (*i.e.*, S_1), as *threads*, que possuem o *kernel* implementado sem erros, não apresentam condição de corrida uma vez que acessam posições diferentes de memória. No segundo caso da Figura 4.7b (*i.e.*, S_2), a execução das *threads* do *kernel* resulta em uma condição de corrida, onde diversas *threads* leem e escrevem na mesma posição de memória no mesmo *lock-step*. Finalmente, o terceiro caso na Figura 4.7b é bem semelhante a situação ocorrida em S_2, porém, ao invés de analisar múltiplas *threads* para detectar condições de corrida, o ESBMC-GPU garante que tais erros podem ser detectados através da análise do comportamento de duas *threads* as quais operam sobre a mesma posição de memória.

Note que a análise de duas *threads* afeta propriamente a verificação de condição de corrida. O programa deve ser analisado sobre suas possíveis intercalações, que direcionem para uma ordem de execução das instruções e resulte em erros. Nos *benchmarks* foi observado uma melhoria de desempenho considerando apenas duas *threads*, enquanto que a taxa de resultados incorretos na detecção de erros apresentou uma baixa taxa.

4.5 Resumo

Neste capítulo foram apresentadas as implementações que foram realizadas para construção da ferramenta ESBMC-GPU bem como os métodos aplicados para melhoria do seu desempenho e a viabilização do seu uso.

O método de modelo operacional permitiu criar representações dos métodos e tipos de

dados da plataforma CUDA para viabilizar a verificação de violação de propriedades usando o ESBMC. Além disso, foram inseridas verificações de propriedades e pré e pós-condições para aumentar a detecção de falhas nos *benchmarks* testados. Com o exemplo do tipo de dado *dim3* e de métodos essenciais como *cudaMalloc*, *cudaMemcpy* e *cudaFree*, foi exibido o modo de identificação das violações de propriedades(*i.e.*, alocação incorreta de memória).

As *threads* CUDA são modeladas como funções *Pthread* e para serem verificadas pelo ESBMC-GPU são utilizadas chamadas de funções específicas para cada tipo de parâmetro passado na função do *kernel* original. Estas funções são responsáveis por verificar pré e pós-condições, além de executar um *loop* para criar e inicializar as *threads pthread*.

O algoritmo MPOR [10] é usado para reduzir o número de intercalações possíveis em programas concorrentes para classificar suas transições como dependentes ou independentes. As transições dependentes são mantidas e sua fórmula SMT é usada na verificação da violação de propriedades. As transições independentes são descartadas reduzindo o espaço de exploração de estados diminuindo a complexidade da análise de fórmulas SMT. A aplicação principal nos programas CUDA explora sua característica de linearização das *threads*, que em alguns casos, resultam em acessos em posições diferentes de memória. Neste caso, as transições que seriam geradas são determinadas como independentes e resultam em uma melhoria considerável no desempenho da verificação.

Por fim, a análise por duas *threads* é uma estratégia usada nas ferramentas GPU Verify [3] e PUG [7] para redução da complexidade da verificação garantindo resultados satisfatórios na detecção de violação de propriedades. Como as *threads* na GPU são processadas em *half-warps*, ou seja, para cada subgrupo de *threads* em um *warp* metade destas são processadas em um *lock-step*, a condição de corrida pode ocorrer apenas nestas *threads* que executam ao mesmo tempo.

A aplicação destes métodos influenciaram na viabilidade do uso da ferramenta reduzindo bastante o tempo de execução e mantendo altos índices de resultados corretos. Esse fato é devido principalmente à redução do espaço de estados simplificando as fórmulas submetidas aos solucionadores SMT.

Capítulo 5

Avaliação Experimental

5.1 Objetivos Experimentais

O objetivo dos experimentos é de investigar o desempenho do ESBMC-GPU na verificação de programas CUDA e avaliar se os métodos de redução de *threads* e intercalações resultam no efeito e redução de tempo esperado mantendo a assertividade da ferramenta. Além disso, realizar uma análise dos solucionadores SMT suportados para verificar qual deles apresenta a melhor relação de assertividade *versus* tempo de verificação e definir o solucionador padrão. Por fim, comparar o desempenho às ferramentas GKLEE [8], GPUVerify [3], PUG [7] e CIVL [9] para certificar a aplicação da técnica de BMC em conjunto com MPOR na verificação de *kernels* CUDA.

Para especificar os objetivos, os experimentos responderam à duas questões de pesquisa:

- RQ1 (*sanity check*) que resultados o ESBMC-GPU obtém sobre a verificação dos *benchmarks* que compõem a *suite* especificada?
- RQ2 (comparação com outras ferramentas) qual o desempenho do ESBMC-GPU quando comparado ao GKLEE, GPUVerify, PUG e CIVL?

5.2 Configuração Experimental

Para responder às questões de pesquisa, foram verificados 154 *benchmarks*[1], que foram extraídos de [1, 3, 11]. Os *benchmarks* que não possuíam a função *main*, tiveram esta imple-

[1]ESBMC-GPU e os *benchmarks* estão disponíveis em http://gpu.esbmc.org

mentada. A Tabela 5.1 exibe as principais características que os *kernels* exploram.

Características	**Funções**
ANSI-C	memeset, assert
CUDA	chamadas da função *__device__*, cudaMalloc, cudaMemcpy, cudaFree, _syncthreads
Biblioteca atomic	atomicAdd
Biblioteca cuRAND	curand, curand_kernel, curand_mtgp32_host
Tipos de dados	int, float, char e os modificadores de tipo long e unsigned, tipos intrínsecos de CUDA(*e*.g., uint4), ponteiros(variáveis e funções)

Tabela 5.1: Propriedades e características exploradas pelos *kernels*.

É possível observar que foram usados *benchmarks* de várias fontes para avaliar a precisão e o desempenho do ESBMC-GPU, que incluíram a *suite* de *benchmarks* CUDA para verificar funções básicas que normalmente são usadas por aplicações CUDA reais. A *suite* de *benchmarks* compreende 20 *kernels* CUDA de *NVIDIA GPU Computing SDK* v2.0 [42], 20 *kernels* CUDA de *Microsoft C++ AMP Sample Projects* [43] e 104 programas CUDA que exploram uma grande quantidade de funcionalidades CUDA [1, 11]. Embora existam alguns *benchmarks* CUDA que não implementam aplicações reais, estes ainda são valiosos para analisar a capacidade do ESBMC-GPU em manipular e detectar (conhecidos) erros. Os *benchmarks* foram previamente usados para comparar o desempenho e a precisão de diferentes verificadores de programas para GPU [3]. Os *benchmarks* relacionados com OpenCL [44] não foram aplicados em nossa avaliação experimental visto que o ESBMC-GPU ainda não possui suporte.

Para responder a RQ1, ESBMC-GPU é executado com: *--force-malloc-success*, que considera sempre uma quantidade de memória suficiente no dispositivo; *--context-switch C*[2], que considera o limite de trocas de contexto entre todas as *threads*; e *--I libraries*, que especifica o diretório das bibliotecas. A chamada de *kernel* foi substituída pela respectiva função *ESBMC_verify_kernel* usando um bloco e duas *threads* por bloco. Como exemplo, uma linha de comando do ESBMC-GPU: *esbmc arquivo.cu --force-malloc-success –context-switch 2 –I bibliotecas*.

Para responder a RQ2, GKLEE, GPUVerify, PUG e CIVL foram aplicados aos *benchmarks*. Com GKLEE, foram usados dois comandos: *gklee-nvcc* e *gklee*. O primeiro analisa o arquivo a ser verificado, com a extensão ".cu". Quando este comando é executado, dois novos

[2]O valor de C varia de 2 a 4 trocas de contexto.

arquivos são gerados: um ".cpp" e um executável(sem extensão). Logo, o segundo comando é usado com o arquivo executável gerado. A linha de comando do GKLEE é: *gklee-nvcc arquivo.cu -libc=uclibc* ou *gklee-nvcc arquivo.cu -libc=klee; gklee arquivo*.

Para verificar um programa CUDA com o GPUVerify, são necessárias as seguintes alterações: (a) remover a função *main*; (b) verificar se a variável de inicialização executada pela função *main* é responsável por controlar alguma declaração condicional dentro do *kernel*, e, caso positivo, esta variável deve ser inicializada pela função *__requires()*; (c) verificar se o *kernel* possui alguma assertiva, e, caso positivo, esta assertiva é substituída por *__assert()*; (d) verificar se existem funções específicas de bibliotecas C/C++, e, caso positivo, estas são removidas pois não são suportadas pelo GPUVerify. Para executar o GPUVerify, duas opções devem ser usadas: *--gridDim=M* e *--blockDim=N* para atribuir o número de blocos e *threads*(por bloco), respectivamente. O GPUVerify é executado pelo comando: *gpuverify file.cu --blockDim=2 –gridDim=1*.

Opções adicionais são necessárias para que o ESBMC-GPU e GPUVerify verifiquem condições de corrida e acessos fora dos limites de vetores, respectivamente.

Algumas alterações são necessárias nos *benchmarks* para usar o PUG: (a) a extensão do arquivo deve ser alterada de ".cu" para ".c"; (b) dado que o PUG não verifica funções *main*, esta deve ser removida e mantida apenas a função *kernel*; (c) as bibliotecas proprietárias do PUG *my_cutil.h* e *config.h* devem ser chamadas dentro do arquivo ".c". A primeira biblioteca tem definições de *structs*, qualificadores e tipos de dados. A segunda define o número de blocos e *threads*(por bloco); (d) O nome da função *kernel* deve ser "kernel". Como exemplo, o PUG é executado pelo comando: *pug kernel.c*.

Para verificar programa CUDA com o CIVL é necessário usar o comando *civl verify file.cu*. Se o *benchmark* contém funções *malloc* e *free*, deve ser adicionado manualmente a biblioteca padrão ANSI-C *stdlib.h* ao programa CUDA, embora isso não seja mandatório de acordo com a documentação CUDA [1].

Todos os experimentos foram conduzidos na plataforma Intel Core i7-4790 CPU 3.60 GHz, 16 GB of RAM e Linux OS. Todos os tempos dados são em segundos e medidos pelo comando *time* do UNIX.

5.3 Avaliação dos solucionadores SMT para Verificação de Programas CUDA

O ESBMC-GPU suporta cinco solucionadores SMT diferentes: Z3 v4.0 [28], Boolector v2.0.1 [45], Yices v2.3.1 [46], MathSAT v5 [47] e CVC4 [48]. Os solucionadores executam em *back-end* e são acoplados ao ESBMC-GPU através de interfaces SMT-LIB ou API nativa[5]. A *flag* $--$solver(*e.g.*,$--$boolector) é usada para definir um solucionador SMT, que é diferente da configuração padrão do ESBMC-GPU.

O desempenho dos solucionadores é mostrado na Tabela 5.2, onde a coluna "ferramenta" descreve o nome da ferramenta; "verdadeiro correto" mostra os resultados onde nenhum erro foi encontrado em *benchmarks* livres de *bug*; "falso correto" mostra os resultados onde a ferramenta detectou um *bug* corretamente; "verdadeiro incorreto" mostra os resultados onde a ferramenta não detectou um *bug*; "falso incorreto" mostra os resultados onde a ferramenta incorretamente detectou um *bug*; "não suportado" mostra os resultados que não são suportados pela respectiva ferramenta; e "tempo" mostra o tempo total de verificação.

A Tabela 5.2 mostra que Z3, Boolector, Yices e MathSAT produzem exatamente o mesmo número de resultados "verdadeiro correto"(56). Z3, Boolector e Yices apresentam 63 resultados "falso correto", enquanto que MathSAT produz apenas 61. Estes resultados garantem que Z3, Boolector e Yices são solucionadores SMT mais efetivos para verificar programas CUDA com o ESBMC-GPU, onde a corretude é um parâmetro importante para avaliar o desempenho dos solucionadores.

Considerando o tempo de verificação, Z3 tem um desempenho melhor em $3,6\%$ sobre Yices. Assim, Z3 foi definido como solucionador SMT(padrão) para verificar programas CUDA no ESBMC-GPU.

5.4 Resultados Experimentais

A Tabela 5.3 mostra os resultados do ESBMC-GPU, GKLEE, GPUVerify, PUG e CIVL. Cada linha significa: nome da ferramenta (Ferramenta); número total de *benchmarks* onde o programa analisou estar livre de erros(verdadeiro correto); número total de *benchmarks* onde o erro no programa foi encontrado e o caminho de erro foi reportado(falso correto); número total de *benchmarks* onde o programa tem um erro, mas o verificador não detectou(verdadeiro in-

correto); número total de *benchmarks* onde um erro é reportado para um programa que cumpre sua especificação(falso incorreto); número total de *benchmarks* que não são suportados(não suportado); tempo de verificação, em segundos, para todos os *benchmarks*(tempo); metodologias usadas pelas ferramentas (metodologias); solucionador usado pelas ferramentas que aplicam esta metodologia (solucionador); funções suportadas pelas ferramentas (suporte à).

ESBMC-GPU é capaz de verificar corretamente 82.5% dos *benchmarks*, GKLEE 71.4%, GPUVerify 57.1%, PUG 35.1% e CIVL 30.5%. O ESBMC-GPU produz 1 resultado "verdadeiro incorreto" enquanto que o GKLEE produz 14, GPUVerify produz 9, PUG produz 7 e CIVL não produz "verdadeiro incorreto". Com o ESBMC-GPU este resultado é devido a um acesso a ponteiro nulo. Com GKLEE, os erros são devido a falha em detectar condição de corrida(10), não detecção de tentativas em modificar memória constante(2), assertivas detectadas incorretamente(1) e acesso a ponteiro nulo(2). GPUVerify não detecta condição de corrida(7), não detecta violação de limites de vetores(1) e não detecta violação de assertiva(1). PUG não detecta acesso a ponteiro nulo(1), condição de corrida(4), violação de limites de vetores(1) e assertiva detectada incorretamente(1).

ESBMC-GPU gera 3 resultados "falso incorreto" devido a assertivas incluídas no *kernel*, que retornam verdadeiro, mas são falhas(2), e a cobertura parcial da função *cudaMalloc* para cópias de variáveis do tipo *float*(1). GKLEE gera 7 resultados "falso incorreto" que são causados por detectar assertivas incorretamente(4), condições de corrida(1), acesso fora dos limites de vetores(1) e falha na chamada do solucionador(1). GPUVerify gera 8 resultados "falso in-

Tabela 5.2: Resultados do desempenho dos solucionadores SMT. **Verdadeiro Correto** representa o número de *benchmarks* sem falhas verificados corretamente, enquanto que **Verdadeiro Incorreto** representa o número dos verificados sem falhas incorretamente. Similarmente, **Falso Correto** representa o número de *benchmarks* verificados com falha corretamente, enquanto que **Falso Incorreto** o número dos verificados com falha incorretamente. Não suportados representa o número de *benchmarks* sem êxito na verificação, **Tempo** representa o tempo total de verificação de cada solucionador, e os números em negrito representam os melhores resultados de cada categoria.

Resultado/Ferramenta	**Z3 4.0**	**Boolector 2.0.1**	**Yices 2.3.1**	**CVC 4**	**MathSAT 5**
Verdadeiro Correto	**56**	**56**	**56**	35	**56**
Falso Correto	**63**	**63**	**63**	60	61
Verdadeiro Incorreto	**2**	**2**	**2**	4	**2**
Falso Incorreto	5	5	4	28	**2**
Não suportado	28	28	29	**27**	33
Tempo (s)	**1828**	4614	1894	3326	8398

correto" que são causados por detectar incorretamente assertivas(2) e condições de corrida(6). PUG produz 11 resultados "falso incorreto" devido a condições de corrida detectadas incorretamente. CIVL produz 3 resultados "falso incorreto" devido a detecção de erro em alocação de memória(2) e assertivas(1).

ESBMC-GPU tem 23 *benchmarks* que não foram suportados. Estes são relacionados à acessos a memória constante(3), ao uso de bibliotecas específicas de CUDA(*e.g., curand.h*, 7) e ao uso de ponteiros para função, estruturas e variáveis do tipo *char* usadas em argumentos de *kernels*(13). GKLEE tem 22 *benchmarks* que não foram suportados devido a assertivas incorretamente detectadas(2), ponteiros de função(11) seja usadas como argumentos do *kernel* ou em qualquer outra parte do programa CUDA, bibliotecas específicas de CUDA(*curand.h*, 7) e funções *switch-case*(2).

GPUVerify não suportou 49 *benchmarks*. O não suporte a funções *main* explica a maioria dos casos(39). Também não suporta o uso da função *memset*(2), acesso a limites fora de vetor(2), o uso de *math_functions*(1) e os outros casos são explicados pela ausência de suporte a ponteiros de função, seja como argumento da função *kernel* ou em qualquer outra parte do programa CUDA(5).

Tabela 5.3: Resultados experimentais.

Ferramenta/ Resultado	**ESBMC-GPU**	**GKLEE**	**GPUVerify**	**PUG**	**CIVL**
Verdadeiro correto	**60**	53	58	39	23
Falso correto	**67**	57	30	15	24
Verdadeiro incorreto	1	14	9	7	**0**
Falso incorreto	**3**	8	8	11	**3**
Não suportado	23	**22**	49	82	104
Tempo (s)	811	128	147	**12**	158
Metodologias	BMC e redução de ordem parcial	*Concolic* e *canonical scheduling*	Semântica operacional (SDV)	Verificação formal	Análise estática e redução de ordem parcial
Solucionador	Z3	-	Z3	Yices	-
Suporte à	*kernel* e *main*	*kernel* e *main*	*kernel*	*kernel*	*kernel* e *main*

PUG tem 82 *benchmarks* não suportados. Como GPUVerify, PUG não verifica funções *main* e isto explica a maioria dos casos(31), enquanto outros são explicados pela falta de suporte a função *__syncthreads*(12), ponteiros de função(9) e a biblioteca *curand.h*(7). Além disso, PUG não suporta o uso do modificador *unsigned* como argumento par a função *atomicAdd*(6), alterações em variáveis armazenadas em memória constante(3) e falta de suporte a manipulação de *structs*, variáveis com o qualificador *__device__*(2) e tipo *size_t*(1). Em outros casos o PUG retornou um falso acesso a ponteio nulo(7) ou por não reconhecer o identificador NULL(2).

CIVL não suporta 104 *benchmarks*. Isto tem como causa o uso de funções atômicas(18), *cudaThreadSynchronize*(10), *threadIdx*(10), função *typedef*(8), funções *curand*(7), *dim3*(6), *math_functions*(5), *uint4*(5), variáveis do tipo *__constant__*(5), *__attribute__*(5), *__restrict*(2), *structs*(2), *scanf*(2), funções booleanas(e.g *AND* e *OR*) (2), *uint*(2), *extern* C(1), *__threadfence*(1), *typecast*(1). Outros *benchmarks* não foram verificados devido a condições de corrida(11) e sem definição(1).

5.5 Efetividade do ESBMC-GPU comparado a outros verificadores

O MPOR produziu uma melhoria de desempenho em cerca de 80% sobre os *benchmarks*. O tempo de verificação foi reduzido de 16 para 3 horas. Com a análise de duas *threads*, o tempo de verificação reduziu de 3 horas para 811 segundos. Embora a aplicação destas técnicas tenha melhorado consideravelmente o desempenho do ESBMC-GPU, o tempo de verificação ainda é superior ao de outras ferramentas. Isto é devido ao modelo atual de execução das intercalações das *threads*(que combina verificação simbólica de modelos com exploração explícita de espaço de estados), enquanto que o GPUVerify faz sua análise completamente simbólica, executada a nível de *kernel*, sem considerar intercalação de *threads* com a *thread main*. PUG tem baixo tempo de verificação devido a análise de duas *threads* e por não verificar a função *main*. GKLEE apresenta um tempo reduzido devido a seu método de redução de estado / caminho.

Para melhorar o desempenho da verificação, o ESBMC-GPU usa uma representação abstrata de bibliotecas CUDA, a qual devidamente reflete sua semântica. Esta representação abstrata inclui pré e pós-condições e simulações de *features* que considera apenas comporta-

mento relevante para ser explorado na perspectiva da verificação, *i*.e., fragmentos de código irrelevante que complicam a geração da Condição de Verificação são abstraídos. Como resultado, durante o processo de verificação do ESBMC-GPU, o MOC substitui as bibliotecas CUDA melhorando a efetividade na detecção de mais violações de propriedades quando comparado a outros verificadores de programas para GPU, como mostrado na avaliação experimental.

Capítulo 6

Conclusões

O ESBMC-GPU é capaz de verificar programas CUDA usando verificação de modelos de contexto limitado baseado em SMT e modelos operacionais, que reconhece diretivas CUDA e simplifica o modelo de verificação. Este trabalho define a primeira aplicação de verificação simbólica de modelos com exploração explícita do espaço de estados usando MPOR para verificar programas CUDA. Em particular, o aplicação do MPOR resulta em 80% de melhoria de desempenho com relação aos *benchmarks* testados.

ESBMC-GPU também apresenta uma melhoria na habilidade de detectar acessos fora dos limites de vetores e condições de corrida se comparado ao GKLEE, GPUVerify, PUG e CIVL. Além disso, ESBMC-GPU fornece a menor taxa de resultados incorretos do que os outros verificadores.

Os resultados experimentais mostram que o ESBMC-GPU apresenta uma taxa de verificação de sucesso de 82.5% comparado ao GKLEE com 71.4%, 57.1% do GPUVerify, 35.1% do PUG e 30.5% do CIVL.

6.1 Trabalhos Futuros

Para trabalhos futuros, o suporte a novos tipos de argumentos em funções *kernel* será melhorado, implementar o suporte a intercalação de *stream* e aplicar técnicas para reduzir o número de intercalação de *threads*. Além disso, testes de conformidade serão desenvolvidos para melhorar a validação do MOC, que também será aplicado a módulos CUDA. Por fim, verificação por indução matemática será investigada com o propósito de provar a corretude

programas CUDA [49, 50].

Referências Bibliográficas

[1] Cheng J, Grossman M, McKercher T. *Professional CUDA C Programming*. [S.l.]: John Wiley and Sons, Inc., 2014.

[2] Kirk D, Hwu W. *Programming Massively Parallel Processors*. [S.l.]: Eslsevier Inc., 2010.

[3] Betts A, Chong N, Donaldson AF, Qadeer S, Thomson P. *GPUVerify: A Verifier for GPU Kernels*. [S.l.]: OOPSLA, 2012; 113–132.

[4] Cordeiro L, Fischer B. *Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking*. [S.l.]: ICSE, 2011; 331–340.

[5] Cordeiro L, Fischer B, Marques-Silva J. *SMT-Based Bounded Model Checking for Embedded ANSI-C Software*. [S.l.]: IEEE Trans. Software Eng., 2012; 38(4):957–974.

[6] Ramalho M, Freitas M, Sousa F, Marques H, Cordeiro L, Fischer B. *SMT-Based Bounded Model Checking of C++ Programs*. [S.l.]: ECBS, 2013; 147–156.

[7] Li G, Gopalakrishnan G. *Scalable SMT-based Verification of GPU Kernel Functions*. [S.l.]: FSE, 2010; 187–196.

[8] Li G, Li P, Sawaya G, Gopalakrishnan G, Ghosh I, Rajan SP. *GKLEE: Concolic Verification and Test Generation for GPUs.* [S.l.]: PPoPP, 2012; 215–224.

[9] Zheng M, Rogers M, Luo Z, Dwyer M, Siegel S. *CIVL: Formal Verification of Parallel Programs*. [S.l.]: ASE, 2015.

[10] Kahlon V, Wang C, Gupta A. *Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique*. [S.l.]: CAV, 2009; 398–413.

[11] NVIDIA. *CUDA C Programming Guide*. [S.l.]: NVIDIA Corporation, 2015.

[12] Pereira P, Albuquerque H, Marques H, Silva I, Carvalho C, Santos V, Ferreira R, Cordeiro L. *Verificação de Kernels em Programas CUDA usando Bounded Model Checking.* [S.l.]: WSCAD-SSC, 2015; 24–35.

[13] Pereira P, Albuquerque H, Marques H, Silva I, Carvalho C, Santos V, Ferreira R, Cordeiro L. *Verifying CUDA Programs using SMT-Based Context-Bounded Model Checking.* [S.l.]: SAC SVT track, 2016; 1648–1653.

[14] de Moura L, Bjorner N. *Satisfiability modulo theories: An appetizer*. [S.l.]: Springer, 2009.

[15] Barrett C, de Moura L, Fontaine P. *Proofs in Satisfiability Modulo Theories*. [S.l.]: College Publications, 2015.

[16] CORDEIRO, L. C. et al. Context-bounded model checking with ESBMC 1.17 - (competition contribution). In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. [S.l.]: Springer, 2012. (Lecture Notes in Computer Science, v. 7214), p. 534–537.

[17] GADELHA, M. Y. R. et al. ESBMC 5.0: an industrial-strength C model checker. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. [S.l.]: ACM, 2018. p. 888–891.

[18] Morse J, Cordeiro L, Nicole D, Fischer B. *Context-Bounded Model Checking of LTL Properties for ANSI-C Software.* [S.l.]: SEFM, 2011; 302–317.

[19] Morse J, Cordeiro L, Nicole D, Fischer B. *Model checking LTL properties over ANSI-C programs with bounded traces.* [S.l.]: Software and System Modeling, 2015; 14(1): 65–81.

[20] Beyer, D. *Status report on software verification.* [S.l.]: TACAS, 2014; 373–388.

[21] Beyer, D. *Software verification and verifiable witnesses.* [S.l.]: TACAS, 2015; 401–416.

[22] Clarke E, Kroening D, Lerda F. *A Tool for Checking ANSI-C Programs*. [S.l.]: TACAS, 2004; 168–176.

[23] Levine J, John L. *Flex & Bison*. [S.l.]: 1st Edition, O'Reilly Media, Inc., 2009.

[24] Lopes B, Auler R. *Getting Started with LLVM Core Libraries*. [S.l.]: Packt Publishing, 2014.

[25] M. Cade. *Why Apple swift language will instantly remake computer programming*. [S.l.]: http://www.wired.com/2014/07/apple-swift/, 2016.

[26] Cytron R, Ferrante J, Rosen B, Wegman M, Zadeck F. *An Efficient Method of Computing Static Single Assignment Form*. [S.l.]: POPL, 1989; 25–35.

[27] Le Goues C, Leino KRM, Moskal M. *The Boogie Verification Debugger*. [S.l.]: SEFM, 2011; 407–414.

[28] Moura L, Bjorner N. *Z3: An Efficient SMT Solver*. [S.l.]: TACAS, 2008; 337–340.

[29] Barrett C, Conway C, Deters M, Hadarean L, Jovanović D, King T, Reynolds A, Tinelli C. *CVC4*. [S.l.]: CAV, 2011; 171–177.

[30] Li P, Li G, Gopalakrishnan G. *Practical Symbolic Race Checking of GPU Programs*. [S.l.]: SC, 2014; 179–190.

[31] Monteiro F, Cordeiro L, de Lima Filho E. *Bounded Model Checking of C++ Programs Based on the Qt Framework*. [S.l.]: GCCE, 2015; 179–447.

[32] Garcia M, Monteiro F, Cordeiro L, de Lima Filho E. *ESBMC^{QtOM}: A Bounded Model Checking Tool to Verify Qt Applications*. [S.l.]: SPIN, 2016; 97–103.

[33] Heila van der Merwe, Brink van der Merwe, and Willem Visser. *Verifying android applications using Java PathFinder*. [S.l.]: SIGSOFT Softw. Eng. Notes 37, 6, 2012; 1–5.

[34] Heila van der Merwe, Oksana Tkachuk, Brink van der Merwe, and Willem Visser. *Generation of Library Models for Verification of Android Applications*. [S.l.]: SIGSOFT Softw. Eng. Notes 40, 1, 2015; 1–5.

[35] Morse J, Ramalho M, Cordeiro L, Nicole D, Fischer B. *ESBMC 1.22 - (Competition Contribution)*. [S.l.]: TACAS, 2014; 405–407.

[36] Mehtaa F, Nipkowb T. *Proving pointer programs in higher-order logic*. [S.l.]: Information and Computation, 2005; 199(1), 200–227.

[37] Cámara P, Castro J, Gallardo M, Merino P. *Verification support for ARINC-653-based avionics software*. [S.l.]: JSTVR, 2011; 21(4): 267–298.

[38] Cámara P, Gallardo M, Merino P, Sanán D. *Checking the reliability of socket based communication software*. [S.l.]: STTT, 2009; 11(5): 359–374.

[39] Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) Base Specifications, IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*. [S.l.]: IEEE, 2008.

[40] Morse J. *Expressive and Efficient Bounded Model Checking of Concurrent Software*. [S.l.]: University of Southampton, PhD Thesis, 2015.

[41] NVIDIA. *CUDA Compute Architecture: Fermi*. [S.l.]: NVIDIA Corporation, 2009.

[42] NVIDIA. *CUDA Toolkit Release Archive*. [S.l.]: https://developer.nvidia.com/cuda-toolkit-archive, 2015.

[43] Microsoft Corporation. *C++ AMP sample projects for download (MSDN blog)*. [S.l.]: blogs.msdn.com/b/nativeconcurrency/archive/2012/01/30/c-amp-sample-projects-for-download.aspx, 2012.

[44] Khronos OpenCL Working Group. *The OpenCL Specification - version 1.1*. [S.l.]: Document Revision: 44.

[45] Brummayer R, Biere A. *Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays*. [S.l.]: TACAS, 2009; 174–177.

[46] B. Dutertre. *Yices 2.2*. [S.l.]: CAV, 2014; 737–744.

[47] Cimatti A, Griggio A, Schaafsma B, Sebastiani R. *The MathSAT5 SMT Solver*. [S.l.]: TACAS, 2013; 93–107.

[48] Deters M, Reynolds A, King T, Barrett C, Tinelli C. *A tour of CVC4: How it works, and how to use it*. [S.l.]: FMCAD, 2014.

[49] GADELHA, M. Y. R.; ISMAIL, H. I.; CORDEIRO, L. C. Handling loops in bounded model checking of C programs via k-induction. *STTT*, v. 19, n. 1, p. 97–114, 2017.

[50] GADELHA, M. Y. R. et al. Towards counterexample-guided k-induction for fast bug detection. In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. [S.l.]: ACM, 2018. p. 765–769.

Apêndice A

Publicações

A.1 Referente à Pesquisa

- **Pereira, P.**, Albuquerque H., Marques H., Silva I., Carvalho C., Santos V., Ferreira R., Cordeiro L. **Verificação de Kernels em Programas CUDA usando Bounded Model Checking** In WSCAD-SSC, pp. 24-35, 2015

- **Pereira, P.**, Albuquerque H., Marques H., Silva I., Carvalho C., Santos V., Ferreira R., Cordeiro L. **Verifying CUDA Programs using SMT-Based Context-Bounded Model Checking** In SAC SVT Track, 1648-1653, 2016

- **Pereira, P.**, Albuquerque H., Silva I., Marques H., Monteiro F., Ferreira R., Cordeiro L. **SMT-based context-bounded model checking for CUDA programs** In Concurrency and Computation-Practice & Experience, v. 29, p. 1-20, 2016