UFAM - Engenharia da Computação

MODELO OPERACIONAL CUDA PARA VERIFICAÇÃO DE PROGRAMAS

Higo Ferreira Albuquerque

Monografia de Graduação apresentada à Coordenação de Engenharia da Computação, UFAM, da Universidade Federal do Amazonas, como parte dos requisitos necessários à obtenção do título de Engenheiro de Computação.

Orientador: Lucas Carvalho Cordeiro

Manaus

Janeiro de 2016

, Higo Ferreira Albuquerque

Modelo Operacional CUDA para Verificação de Programas/Higo Ferreira Albuquerque . – Manaus: UFAM, 2016.

XIII, 61 p.: il.; 29, 7cm.

Orientador: Lucas Carvalho Cordeiro

Monografia (graduação) – UFAM / Curso de Engenharia da Computação, 2016.

Referências Bibliográficas: p. 58 – 60.

1. Modelo Operacional. 2. CUDA. 3. Verificação de Software. 4. ESBMC. I. Cordeiro, Lucas Carvalho. II. Universidade Federal do Amazonas, UFAM, Curso de Engenharia da Computação. III. Título.

Modelo Operacional CUDA para Verificação de Programas

Higo Ferreira Albuquerque

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO CURSO DE ENGENHARIA DA COMPUTAÇÃO DA UNIVERSIDADE FEDERAL DO AMAZONAS COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE COMPUTAÇÃO.

Aprovada por:

_____
Prof. Lucas Carvalho Cordeiro, Ph.D.

_____
Prof. Celso Barbosa Carvalho, D.Sc.

_____
Prof. Carlos Augusto Moraes Cruz, D.Sc.

MANAUS, AM – BRASIL
JANEIRO DE 2016

"Quem acredita sempre alcança"

(Renato Russo)

Agradecimentos

- Agradeço a Deus pela jornada da vida e pelas conquistas alcançadas.
- Agradeço à minha mãe, Silvana Ferreira Albuquerque, e ao meu pai, Izannilson Geraldo dos Santos Albuquerque, que me educaram e me proporcionaram condições para seguir com esta graduação.
- Agradeço aos meus amigos do laboratório de verificação de *software* e sistemas, Phillipe Pereira, Isabela Silva, Hendrio Marques, Vanessa Santos, Hussama Ismail, Felipe Sousa, Mário Praia, Williame Rocha, Suzana Rita e João Campos.
- Agradeço aos meus amigos da graduação Felipe Brasil, Max Simões, Lais Negreiros, Flávio Tomé, Sávio Tomé, Jordan Rodrigues, Darlison Sousa, Walmir Aciole, Bruno Auzier, Ramon da Silva, Luciano Sobral, Tiago Melgueiros, Bárbara Lobato, que sempre me acompanharam durante essa jornada me dando alegrias para continuar.
- Agradeço em especial a minha amiga Bárbara Lobato que sempre foi paciente em me ajudar, deu-me força e confiança para vencer as etapas difíceis, também me ajudou a tomar decisões difíceis, o que permitiu minha chegada até aqui.
- Agradeço ao meu irmão Hugo Albuquerque e amigos de infância Rômulo Assunção e Eduardo Nascimento.
- Agradeço ao meu orientador Prof. Lucas Cordeiro pela paciência, pelos conselhos na graduação, pelas oportunidades e o suporte oferecidos para o desenvolvimento deste trabalho. Agradeço ao Prof. Waldir Sabino pelos conselhos e oportunidades durante a graduação. Agradeço ao Prof. Celso Barbosa que ajudou bastante nesta etapa final de TCC e estágio.

Resumo da Monografia apresentada à UFAM como parte dos requisitos necessários para a obtenção do grau de Engenheiro de Computação

MODELO OPERACIONAL CUDA PARA VERIFICAÇÃO DE PROGRAMAS

Higo Ferreira Albuquerque

Janeiro/2016

Orientador: Lucas Carvalho Cordeiro

Programa: Engenharia da Computação

Este trabalho propõe um modelo operacional baseado na plataforma Compute Unified Device Architecture (CUDA) para o verificador Efficient SMT-Based Context-Bounded Model Checker (ESBMC). Através desta abordagem, ESBMC é capaz de verificar propriedades (*eg*, condições de corrida de dados e indicadores de segurança) em programas escritos em CUDA. Inicialmente, um conjunto de *benchmarks* foram coletados a partir das ferramentas disponíveis que realizam a verificação dos programas para a unidade de processamento gráfico (GPU), a fim de criar um conjunto de testes robustos. Com base na análise desses benchmarks, foram identificadas todas as estruturas (it ie, métodos, classes, funções, etc.) específico de programas CUDA, a fim de classificar as estruturas a ser implementadas. O comportamento de cada estrutura identificada foi modelada no modelo operacional, com foco na verificação das propriedades relacionadas. Vale resaltar que o modelo operacional foi desenvolvido em ANSI-C/C++, a fim de garantir a compatibilidade com o ESBMC. Finalmente, o respectivo modelo operacional foi integrado e contribuiu para o desenvolvimento do ESBMC-GPU.

Abstract of Monograph presented to UFAM as a partial fulfillment of the requirements for the degree of Engineer

Higo Ferreira Albuquerque

January/2016

Advisor: Lucas Carvalho Cordeiro

Department: Computer Engineering

This work proposes an operational model based on the Compute Unified Device Architecture (CUDA) platform for the Efficient SMT-Based Context-Bounded Model Checker (ESBMC). Through this approach, ESBMC is able to verify properties (*e.g.*, data race conditions and pointers safety) in programs written in CUDA. Initially, a set of benchmarks were collected from the available tools that perform verification of programs for graphics processing unit (GPU), in order to create a robust test suite. Based on the analysis of such benchmarks, all structures (*i.e.*, methods, classes, functions, etc.) specific of CUDA programs were identified, in order to classify the structures that must be implemented. The behaviour of each identified structure has been modelled into the operational model, focusing on the verification of the properties related to them. It is worth noticing that the operational model was interelly developed in ASNI-C/C++, in order to ensure the compatibility with ESBMC. Finally, the respective operational model has been integrated into the model checker.

Sumário

Lista de Figuras

Lista de Tabelas

Abreviações

CUDA - *Compute Unified Device Architecture*

GPU - *Gaphics Processing Unit*

CPU - *Central Processing Unit*

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

ESBMC-GPU - *Efficient SMT Based Context-Bounded Model Checker -Gaphics Processing Unit*

MOC - *Modelo Operacional CUDA*

COM - *C++ Operation Model*

QtOM - *Qt Operation Model*

FMS - *Flexible Manufacturing System*

PLC - *Programmable Logic Controller*

CETELI - *CEntro de P&D em Tecnologia ELetrônica e da Informação*

SMT - *Satisfiability Modulo Theories*

SSA - *Single Static Assignment*

IREP - *Intermediate REPresentation*

STL - *Standard Template Library*

VC - *Verification Condition*

ALU - *Arithmetic Logic Unit*

DRAM - *Dynamic Random Access Memory*

SM - *Stream Multiprocessor*

Capítulo 1

Introdução

A técnica *model checking*, no português checagem de modelos é uma área em ascensão no cenário da pesquisa. *Model checking* nada mais é do que uma forma de verificar estados finitos de forma automática em sistemas concorrentes [4]. Surgiu no início dos anos 80 principalmente com as pesquisas dos professores Clarke e Emerson com a utilização da lógica temporal. Hoje possui aplicações em sistemas de *hardware*, *software* e protocolos de comunicação [4]. Garante ausência de *deadlocks* e comportamentos similares que possam levar ao travamento do sistema, por exemplo nos sistemas flexíveis de manufatura (FMS) que são um conglomerado de PLCs, em geral apresenta um elevado grau de compartilhamento de recursos [5][6][7], ou em algoritmos concorrentes como programas em C, C++ e CUDA [8][9][10].

A utilização da verificação, em especial a verificação formal, à códigos tem sido a alternativa para encontrar anomalias que causariam o travamento de sistemas. Normalmente programas são escritos e alguns erros como *overflow* e *array bounds* são imperceptíveis ao programador e a ferramenta de programação; esses erros podem ser encontrados usando ferramentas de verificação de programas.

A alta taxa de produção de programas eleva a possibilidade de erros como os citados anteriormente. Além do que, os programas são construídos por vários desenvolvedores e muitos são extensos, a ponto de possuírem milhares de linhas de código. Esses fatores influenciam na necessidade de ferramentas automáticas para detecção de falhas.

A necessidade de verificação de programas se dá em sistemas que em caso de falha, irão causar enormes impactos a sociedade ou financeiro, como exemplo

o foguete Ariana 5 que explodiu em 4 junho de 1996, quando um seus programas tentou realizar uma conversão de um número ponto flutuante de 64-bit para um valor inteiro de 16-bit, caracterizando em um *overflow* [4].

Com a evolução nos últimos 10 anos das unidades de processamento gráficos (do inglês, graphic process unit - GPUs), em especial as placas de vídeo, hoje é possível aplicá-las não somente para processamento gráfico, mais em qualquer processamento para cálculos de uso geral [11]. Um rápido desenvolvimento de *hardware* da GPU levou a um uso extensivo em aplicações científicas e comerciais, inúmeros trabalhos relatam o aumento de velocidade de projetos antes executados em CPUs [2].

A utilização da GPU depende de um elevado grau de paralelismo entre a carga de trabalho, também precisa de uma carga de trabalho substancial para esconder a latência da transição de dados entre memórias [2]. O número de cálculos numéricos realizados por operação de transição de dados de memória, deve ser elevada a fim de preservar a intensidade de computação [2].

Hoje existem aplicações que usam GPUs em controle de trafego aéreo, simulações, análise de riscos no mercado financeiro, climatologia, ciências médicas, manufatura industrial, onde é essencial haver uma verificação de programas [12].

A partir da necessidade foi iniciado um projeto no CETELI/UFAM para desenvolver uma ferramenta de verificação (ESBMC-GPU), voltada para códigos do *Compute Unified Device Architecture* (CUDA), extensão da linguagem ANSI-C e C++ usada para programação em GPUs CUDA. Cabe neste trabalho as partes que tive participação, em especial no modelo operacional que será descrito durante todo o trabalho.

1.1 Motivação

Busca-se desenvolver uma ferramenta de verificação de programas CUDA a partir da ferramenta ESBMC. O ESBMC atualmente suporta códigos escritos em ANSI-C e C++, e existe uma extensão do ANSI-C e C++ para CUDA. A motivação para desenvolver o modelo operacional CUDA (MOC) neste trabalho é auxiliar a extensão do ESBMC para suportar a verificação de códigos CUDA.

1.2 Descrição do Problema

O ESBMC atualmente não seria capaz de interpretar as funções e características do CUDA durante uma verificação, teria problemas com o *parser*, e não saberia como proceder com o código. A criação do modelo operacional CUDA (MOC) permitirá ao ESBMC ser capaz de interpretar e verificar códigos CUDA.

1.3 Objetivos

Objetivo geral:

- Desenvolver um modelo operacional que permitirá o ESBMC interpretar códigos CUDA.

Os objetivos específicos são:

- Identificar características de código específicas do CUDA, como funções e estruturas de dados.
- Investigar o comportamento de cada característica identificada, compreender os tipos de parâmetros, o processamento e retorno das funções.
- Implementar o modelo das características encontradas para criação do MOC.
- Compor o conjunto de *benchmarks* com códigos CUDA.
- Testar e avaliar a eficiência do MOC com os *benchmarks*.

1.4 Organização da Monografia

A monografia está organizada da seguinte maneira:

- No Capítulo 2 são apresentados os conceitos de arquitetura e programação da plataforma CUDA, é também apresentada a área da verificação e a ferramenta ESBMC, por fim o modelo operacional é descrito.
- O Capítulo 3 apresenta um resumo dos trabalhos relacionados à modelo operacional.

- O Capítulo 4 descreve os procedimentos adotados para desenvolvimento do MOC.
- No Capítulo 5 são apresentados os resultados de desempenho do MOC, e uma discussão dos resultados obtidos dentro do projeto com o MOC.
- O Capítulo 6, apresenta as conclusões do trabalho, além de apresentar sugestões para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Para o desenvolvimento do trabalho foi necessário o aprofundamento na plataforma CUDA, tanto na arquitetura quanto na linguagem usada que é a extensão do C e C++. Foi também explorado a teoria de verificação, em especial a verificação de modelos, houve a compreensão teórica da arquitetura e dos códigos do projeto ESBMC do, e a fixação dos conceitos em modelo operacional.

2.1 CUDA

O surgimento das GPUs acontece quando as CPUs não conseguiam atender as necessidades de processamento requeridas em novas aplicações, no caso tratava-se do processamento gráfico, por isso alguns fabricantes começaram a desenvolver as placas de vídeo que liberava a CPU de processamento gráfico [2]. As GPUs são microprocessadores especializadas que a partir do processador central aliviam e aceleram a renderização dos gráficos[11].

O CUDA é uma plataforma de computação paralela de propósito geral criada pela NVIDIA, que tira proveito das unidades de processamento gráfico (GPUs) e resolve muitos problemas computacionais complexos em uma fração do tempo menor que quando executada na CPU [11].

O poder de aplicação das GPUs não é mensurável, no caso da GPU CUDA há milhões de placas em uso e a cada dia desenvolvedores de programas e pesquisadores inovam descobrindo aplicações. Com o CUDA é possivel explorar o potencial computacional das GPUs até então restrito à computação gráfica. Atualmente é encontrado

GPUs CUDA em projetos de simulação, astrofísica, meteorologia, processamento de sinais, análise de vídeos, na área da saúde como em bio-química, bio-informática e visualização de moléculas, em projetos estruturais, mecânicos, eletromagnéticos, análise de riscos financeiros, análise do fluxo de tráfego aéreo e outros [11] [12].

2.1.1 CPU x GPU

A Figura 2.1 apresenta um gráfico que compara a evolução das GPUs e CPUs ao longo dos últimos 15 anos. Considerando de setembro de 2002 até agosto de 2013, a GPU elevou sua capacidade de processamento em mais de 20 vezes em precisão simples, enquanto a CPU elevou em até 6 vezes somente. O Gráfio também mostra a vantagem da GPU sobre à CPU quanto a processamento em precisão simples e dupla.

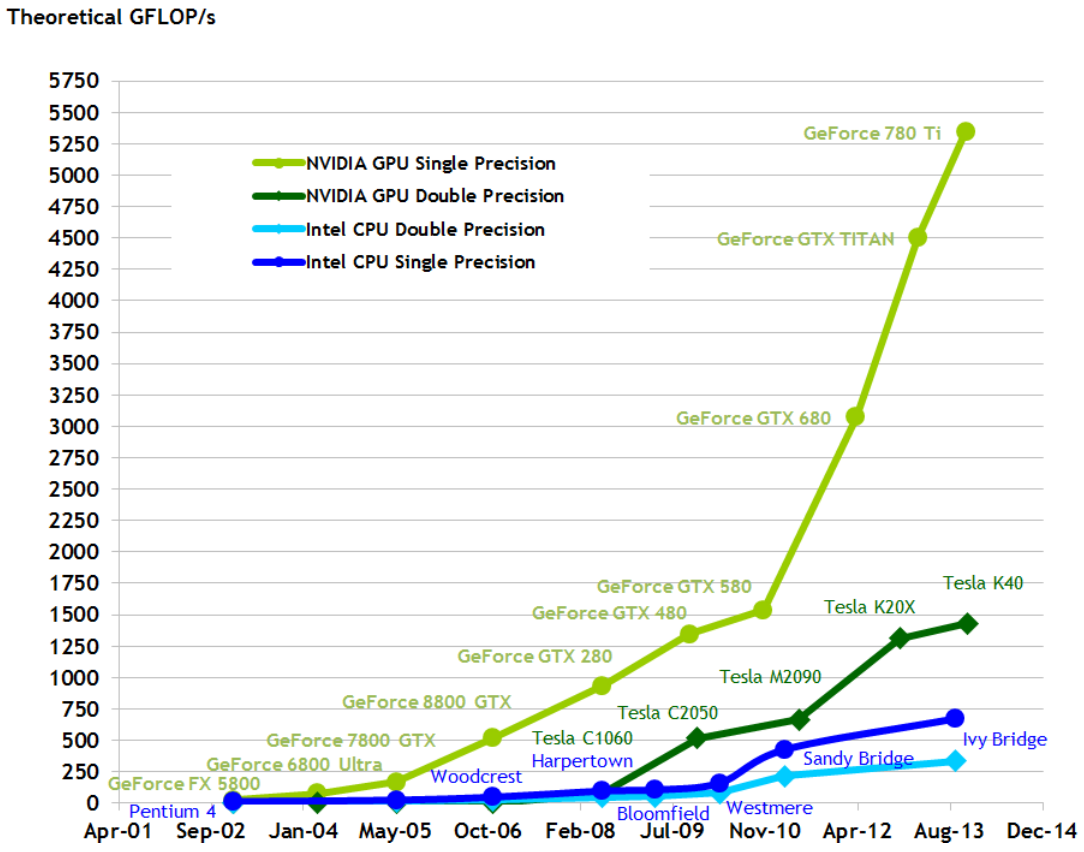

Figura 2.1: Operações em ponto flutuante por segundo para CPU e GPU [1].

A superioridade da GPU sobre a CPU se dá devido ao uso da computação paralela. As CPUs possuem transistores dedicados à *cache* e controle de fluxo sofisticado. A GPU possui transistores dedicados ao processamento de dados em vez da

cache de dados e controle de fluxo [1] [2]. A Figura 2.2 mostra que a CPU possui poucas unidades lógicas e aritméticas (ULA), mas possui uma *cache* e controle robusto e uma DRAM, enquanto a GPU possui muitas ULAs, uma pequena unidade de controle e *cache* para cada grupo de ULAs, e uma DRAM para transferências entre *host* e o *device*, e para as entradas e saídas da execução dos *kernels*.

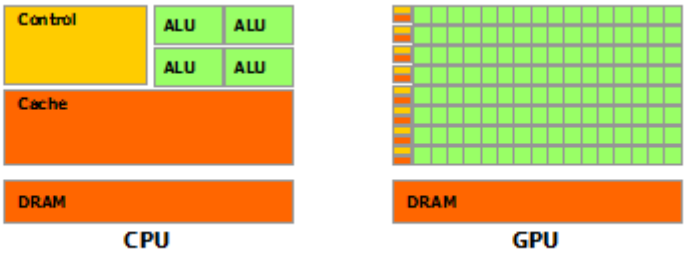

Figura 2.2: Arquiteturas CPU e GPU [1].

As CPUs podem ser consideradas como processadores multicores, uma vez que eles precisam de apenas algumas *threads* para usar sua capacidade plena, enquanto GPUs são processadores que precisam de milhares de *threads* para a sua utilização plena [2]. As GPUs poderiam ser vistas como unidades de coprocessamento para as CPUs, os quais são adequados para problemas que exibem elevada regularidade e intensidade aritmética [2]. Tipicamente, as partes sequenciais do programa são executadas na CPU, enquanto as partes de computação intensiva são transferidas para acelerar todo o processo [2].

A GPU é especializada em resolver problemas expressos com cálculos de dados paralelos, fazer uma grande quantidade de aritmética de ponto flutuante em dados completamente independentes, tais como transformar as posições dos vértices do triângulo ou geradora de cores de *pixels* [2]. Esta independência de dados, é uma das diferenças chave entre o *design* para projeto GPUs e CPUs. O mesmo *kernel* é executado por diferentes *threads* em paralelo com dados de entrada diferentes, essa abordagem na GPU permite um controle de fluxo menos sofisticado que na CPU [1][13], permite uma utilização de um paralelismo substancial do hardware e o processar grandes quantidades de dados [2].

A computação paralela tem processamento rápido, porém sua curva de aprendizado é elevada, existe muita dificuldade para gerar programas, e com a evolução da plataforma CUDA, o usuário passou a dispor de opções com extensões do C,

C++ e Fortran para escrever códigos que executem direto na GPU, não precisando mais da linguagem de compilação.

Por meio de uma linguagem de alto nível como C e C++, aplicações são escritas onde executam as partes sequenciais na CPU e possuem partes aceleradas por GPU, o que elimina a carga de trabalho da CPU.

2.1.2 Modelo de Programação Escalável

O núcleo do CUDA é segmentado em três abstrações: a hierarquia do grupo de *thread*, memórias compartilhadas e a sincronização de barreiras, que para o programador é um conjunto mínimo de extensões de linguagem de alto nível [1]. Estas abstrações fornecem paralelismo de dados refinados, guiam o programador para particionar o problema em sub-problemas que podem ser resolvidos de forma independente em paralelo por blocos de *threads*. Cada sub-problema é quebrado em partes menores que podem ser resolvidas de forma cooperativa em paralelo por todas as *threads* dentro do bloco [1], [13]. Essa decomposição preserva a linguagem e permite que *threads* cooperem na resolução de cada sub-problema e permite automatizar a escalabilidade. Cada bloco pode ser programado em qualquer multiprocessador na GPU e em qualquer ordem simultaneamente ou sequencialmente. A Figura 2.3 mostra uma ideia da estrutura onde a GPU é segmentada em blocos, e os blocos em *threads*, mas o usuário tem o poder de habilitar somente os blocos que acha necessário para sua aplicação, e expandir o uso conforme a carência [13].

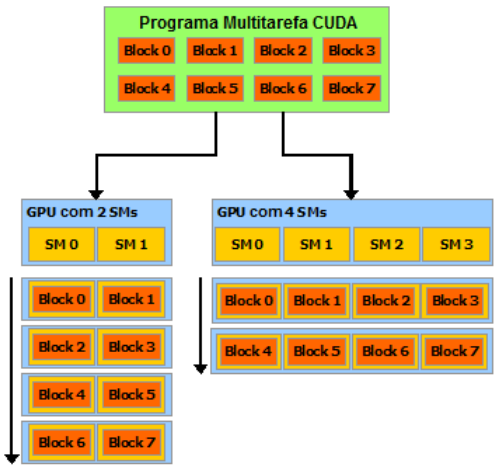

Figura 2.3: Escalabilidade conforme a estrutura da GPU [1].

2.1.3 Modelo de Programação

Os programas CUDA são executadas em coprocessamento. Tipicamente a aplicação consiste em duas partes: a parte sequencial é executada na CPU (*host*) que é responsável pela gestão de dados, transferências de memória, e configuração da execução da GPU; a parte paralela do código que é executado na GPU (*device*), como funções *kernels* que são chamadas pela CPU [2].

Um *kernel* é executado rapidamente por um grande número de *threads* em unidades de processamento chamadas por *streaming multiprocessors* (SMs). Cada *streaming multiprocessors* é um processador SIMD, constituído por até 32 processadores escalares [2]. As *threads* são organizadas em blocos e executadas em um único multiprocessador, e a execução do *kernel* é organizado como uma grade de blocos de *threads*, como mostrado na Figura 2.4. Blocos de *threads* são executados no mesmo SM e compartilham os recursos disponíveis, como os registos e memória compartilhada. O número de *streaming multiprocessors* variam de 2 até 32 unidades conforme o modeloo da GPU [2].

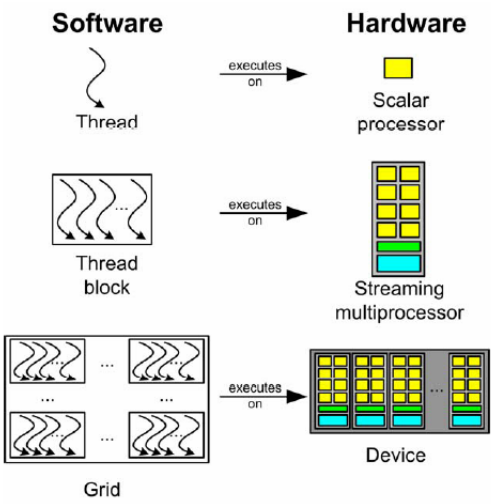

Figura 2.4: Modelo de execução CUDA [2].

O número de *threads* por bloco, e o número de blocos em uma grade são especificados através da configuração do *kernel* [2]. Bloco e grade podem ser multidimensional (1D, 2D e 3D) para suportar diferentes padrões de acesso à memória. Cada *thread* possui um único índice dentro de um bloco, e cada bloco possui um único identificador dentro da grade. As *threads* podem acessar esses identificadores e dimensões, através de variáveis especiais. Em geral, cada *thread* utiliza os seus

próprios índices para ramificação, e para simplificar o endereçamento de memória no tratamento de dados multidimensionais [2].

A plataforma CUDA fornece uma maneira de sincronizar *threads* no mesmo bloco usando a sincronização de barreira [2]. No entanto, *threads* em diferentes blocos não podem cooperar, uma vez que poderiam ser executado em diferentes *streaming multiprocessors*.

O CUDA C é uma extensão da linguagem C, permite o programador usar funções do C e as funções específicas do CUDA [1], permite definir funções normais em C, definir funções *kernel* que são executadas no *device* e permite realizar a chamada de *kernel* como mostra a linha 10 do código da Figura 2.5. O programa básico consiste de uma função *main()* e de no mínimo uma função *kernel* como mostrado na Figura 2.5.

Um *kernel* é definido utilizando a declaração do qualificador $\_\_global\_\_$ - em uma função fora da *main()*, como mostrado na Figura 2.5, entre as linhas 2 e 5. A configuração da chamada de *kernel* é especificada usando o *parser* $<<<$ *thread*, blocos $>>>$, onde indica que será usado N blocos no *device*, e o *kernel* será executado por N *threads* em cada bloco, onde cada *thread* iniciada pelo *kernel* possui um ID único que é acessível internamente ao *kernel*, no exemplo da Figura 2.5 o ID é acessado através da variável *threadIdx* [1].

A Figura 2.5 apresenta um exemplo de código de adição de vetores de tamanho N que armazena o resultado no vetor C [1].

```
1  // Kernel definition
2  __global__ void VecAdd(float* A, float* B, float* C){
3      int i = threadIdx.x;
4      C[i] = A[i] + B[i];
5  }
6
7  int main(){
8      ...
9      // Kernel invocation with N threads
10     VecAdd<<<1, N>>>(A, B, C);
11     ...
12 }
```

Figura 2.5: Soma de dois vetores de tamanho N [1].

2.1.4 Hierarquia de *threads*

O *threadIdx* é um vector de 3 componentes, este identificada o índice da *thread* de forma unidimensional, bidimensional ou tridimensional, formando um bloco unidimensional, bidimensional ou tridimensional de *threads* [1]. Isso faz uma referência involuntária, mas correta com um vector, matriz e volume, assim os elementos de computação são chamados.

O índice da *thread* e o ID da *thread* estão relacionados da seguinte forma: caso o bloco seja unidimensional, eles são os mesmos; caso o bloco seja bidimensional de tamanho (Dx, Dy), o ID da *thread* de índice (x, y) é (x + y Dx); caso o bloco seja tridimensional de tamanho (Dx, Dy, DZ), o ID da *thread* de índice (x, y, z) é (x + y + z Dx Dx Dy) [1].

```
1  // Kernel definition
2  __global__ void MatAdd(float A[N][N], float B[N][N], ...
       float C[N][N]){
3      int i = threadIdx.x;
4      int j = threadIdx.y;
5      C[i][j] = A[i][j] + B[i][j];
6  }
7
8  int main(){
9      ...
10     // Kernel invocation with one block of N * N * 1 threads
11     int numBlocks = 1;
12     dim3 threadsPerBlock(N, N);
13     MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
14     ...
15 }
```

Figura 2.6: Soma de duas matrizes de tamanho NxN [1].

A Figura 2.6 apresenta um exemplo de código de adição de matrizes de tamanho NxN que armazena o resultado na matriz C [1]. Neste exemplo é usado os índices das *threads* durante a execução do *kernel* para determinar o elemento da matriz, na prática cada elemento ixj da matriz é somado de forma paralela.

Por fim os blocos são organizados em uma grade unidimensional, bidimensional ou tridimensional de blocos de *threads*, como ilustrado na Figura 2.8. O número de blocos de *threads* numa grade é determinado pelo número de processadores da GPU, a quantidade de *threads* por blocos varia conforme a GPU.

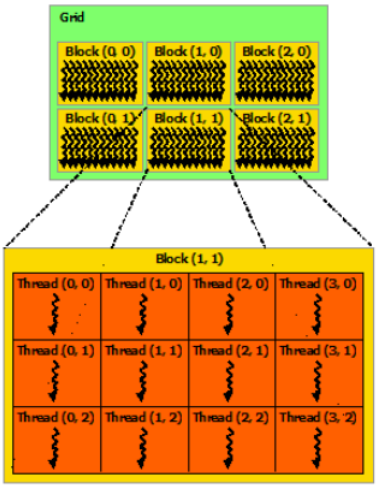

Figura 2.7: Grade de Blocos de *threads* [1].

2.1.5 Hierarquia de memória

No CUDA, as *threads* acessam múltiplas memórias durante uma execução como mostrado na Figura 2.8. Cada bloco de *threads* compartilha memória entre todas as *threads* do bloco, enquanto o tempo de vida entre as *threads* for o mesmo [1]. As *threads* tem acesso aos registros, memória local, memória compartilhada, a memória constante, memória de textura e memória global [2]. Outros dois espaços de memória estão acessíveis para as *threads*, mas somente como leitura, são as memórias constante e de textura [1].

Todas as *threads* tem acesso lento à memória global (DRAM) do dispositivo em comparação com outras memórias. Cada *thread* tem um acesso exclusivo aos dados alocados, registradores e memória local, cada *thread* de um bloco pode compartilhar dados com *thread* de outro bloco por meio da memoria compartilhada, assim o acesso à memórias locais é tão caro como o acesso à memória global [2].

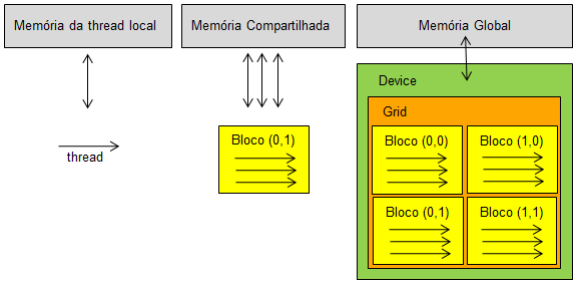

Figura 2.8: Hierarquia de memória [1].

2.1.6 Programação Heterogênea

O modelo de programação CUDA assume que as *threads* executam em um *device*, o qual está separado do *host* e funciona como um coprocessador. Este é o caso da programação heterogênea, quando as *threads* executam na GPU, a outra parte do programa C executa na CPU [1]. A Figura 2.9 mostra o *host* com um único segmento de código fazendo a chamada de *kernel*, o *device* executa várias *threads* com o *kernel* e retorna o resultado para o *host*, que faz mais uma requisição ao *device*.

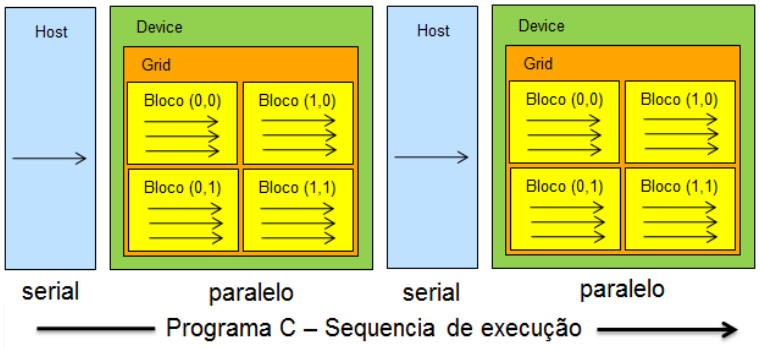

Figura 2.9: Fluxograma da computação heterogênea [1].

2.2 Métodos de Verificação

Sistemas computacionais estão cada vez mais complexos e, devido a competitividade do mercado, busca-se reduzir o seu tempo de desenvolvimento. Agilizar o desenvolvimento de um programa poderá criar problemas como inconsistências no código e, por displicência permitir a manufatura de um produto embarcando um programa com deficiências.

Os projetistas utilizam métodos de validação de sistemas no processo produtivo, tais como simulação, testes, verificação dedutiva e verificação de modelos [4].

Os experimentos com simulação e testes são realizados antes da implantação do sistema na prática, o primeiro abstrai um modelo do sistema para ser aplicado, enquanto que o segundo é empregado em um próprio sistema. Em ambos os casos, são inseridas entradas em partes do sistema e observada as saídas. Esses métodos tem boa relação custo-benefício para encontrar falhas, porém são consideradas todas as interações dentro do sistema, eles não garantem mapear todas as possibilidades [4].

A verificação dedutiva está relacionada ao uso de axiomas e prova de regras que validam se o sistema está correto. No início da pesquisa, o foco da verificação dedutiva foi garantir que o sistema não apresentasse erros. A importância de se ter um sistema correto era tanta, que o especialista poderia investir o tempo que fosse necessário para realização dessa tarefa [4]. Inicialmente as "provas" eram todas construídas manualmente, até que os pesquisadores perceberam que programas poderiam ser desenvolvidos para usar corretamente os axiomas e provas de regras. Tais ferramentas computacionais também podem aplicar uma busca sistemática para sugerir várias maneiras de progredir a partir da fase atual da prova [4]. A verificação dedutiva tem importância para ciência da computação pois influência no desenvolvimento de programas. Aplica-se em sistemas de estado infinito e, nenhum limite é imposto sobre a quantidade de tempo ou memória a fim de encontrar a prova [4].

A verificação de modelos é uma técnica que se aplica à sistemas de estados finitos concorrentes, e pode ser usada em conjunto com outras técnicas de verificação. Essa técnica é realizada de forma automática, onde é feita uma pesquisa exaustiva no espaço de estados do sistema para determinar se uma especificação é verdadeira ou falsa [4].

2.2.1 Métodos Formais

A verificação de sistemas complexos de *hardware* e *software* exige muito tempo e esforço. O uso de métodos formais prover técnicas de verificação mais eficiente, que reduz o tempo, miniminiza o esforço e aumenta a cobertura da verificação [14].

Os métodos formais tem o objetivo de estabelecer um rigor matemático na verificação de sistemas concorrentes [14], são técnicas de verificação "altamente recomendadas" para o desenvolvimento de sistemas críticos de segurança do software de acordo com a *International Electrotechnical Commission* (IEC) e a *European Space Agency* (ESA) [14]. As instituições *Federal Aviation Administration* (FAA) e *National Aeronautics and Space Administration* (NASA) reportaram resultados sobre os métodos formais concluindo que:

"*Os métodos formais deveriam ser parte da educação de cada engenheiro e cientista de software, assim como o ramo apropriado da matemática aplicada é uma*

parte necessária da educação de todos os outros engenheiros [14]"

Nas últimas duas décadas, as pesquisas com métodos formais permitiu o desenvolvimento de algumas técnicas de verificação que facilitam a detecção de falhas. Tais técnicas são acompanhadas de ferramentas utilizadas para automatizar vários passos da verificação [14]. A verificação formal mostrou que pode ser relevante ao expor defeitos como a missão Ariane 5 [4], missão Mars Pathfinder [14] e acidentes com a máquina de radiação Therac-25 [15].

2.2.2 Verificação de Modelos

Como já mencionado, a verificação de modelos atua em sistemas de estados finito concorrentes, garante a validação de sistemas programáveis. O objetivo da técnica é provar matematicamente por meio de métodos formais, que um algoritmo não viola uma propriedade considerando sua própria estrutura [4].

No início de 1980, a técnica de verificação de modelos passou por uma evolução, quando os pesquisadores Clarke e Emerson introduziram o uso da lógica temporal [4]. A verificação de um único modelo que satisfaz uma fórmula é mais fácil que provar a validade de uma fórmula para todos os modelos [4]. A lógica temporal se mostra útil para especificar sistemas concorrentes, pois ela descreve a ordem dos eventos [4].

A aplicação da verificação de modelos consiste de três partes: modelagem; especificação e verificação [4].

- **Modelagem**: Converte o que deseja-se verificar em um formalismo por uma ferramenta de verificação de modelos. Em muitos casos, isso é uma tarefa de compilação, em outros existem limitações de memória e tempo para a verificação, por isso a modelagem pode requerer o uso de abstrações e eliminar detalhes irrelevantes [4].

- **Especificação:** Antes de realizar a verificação é preciso saber quais propriedades serão verificadas. Essa especificação é geralmente realizada, através de algum formalismo lógico. Para sistemas de *hardware* e *software* é comum usar lógica temporal, podendo afirmar como se comporta a evolução do sistema ao longo do tempo. A verificação de modelos fornece meios para determinar se o

modelo do *hardware* ou *software* satisfaz uma determinada especificação, contudo é impossível determinar se a especificação abrange todas as propriedades que o sistema deve satisfazer [4].

- **Verificação:** A ideia é que a verificação seja completamente automática, mas existem casos que necessita do auxílio humano, como na análise dos resultados. Quando um resultado é negativo um contraexemplo é fornecido ao usuário, este contraexemplo pode ajudar a encontrar a origem do erro, indicar a propriedade que possui falha [4]. A análise do contraexemplo pode requerer a modificação do programa e reaplicar o algoritmo *model checking* [4]. O contraexemplo pode ser útil para detectar outros dois tipos de problemas, uma modelagem incorreta do sistema ou, uma especificação incorreta [4]. Uma última possibilidade é de que a tarefa de verificação falhe, devido ao tamanho do modelo que pode ser grande e ter elevado consumo de memória[4], neste caso, talvez seja preciso realizar ajustes no modelo.

O maior desafio dessa abordagem são as explosões do espaço de estados, isso ocorre quando durante uma verificação, muitos caminhos computacionais são traçados e verificados, consumindo todo potencial de memória da máquina que executa o teste. A medida que o número de variáveis de estado do sistema aumenta, o tamanho do espaço de estados cresce exponencialmente. Outros fatores que contribuem para esse crescimento são as interações das variáveis, a atuação em paralelo ou ainda o não determinismo presente em algumas estruturas do sistema (*e.g.*, variáveis sem valores definidos pelo usuário) [4].

A vantagem da verificação de modelos está em automatizar os testes, obter resultados em tempos menores, tornando-a preferível à verificação dedutiva. O processo realiza uma pesquisa exaustiva no espaço de estados do sistema, e ao terminar é gerada uma resposta que afirma se o modelo satisfaz a especificação, ou retorna um contraexemplo onde mostra o motivo pelo qual não é satisfatível [4].

2.3 ESBMC

As ferramentas de verificação de programas checam propriedades como *overflow* aritmético e violação dos limites de vetores em um código automaticamente,

encontram falhas em um programa não identificadas pelo desenvolvedor.

O ESBMC é uma ferramenta eficiente para a verificação de programas embarcados em ANSI-C e C++ [10], [16], [17], [18] que utiliza a técnica Verificação de Modelos Limitadas (do inglês, *Bounded Model Checking* - BMC) e solucionadores da Teorias do Módulo de Satisfatibilidade (do inglês, *Satisfiability Modulo Theories* - SMT), como o Z3 [19] Boolector [20]. A ferramenta permite realizar a validação de programas sequenciais ou multitarefas além de verificar bloqueio fatal, estouro aritmético, divisão por zero, limites de *array* e outros tipos de violações [16].

A ferramenta está no estado da arte e possui prêmios na competição internacional de verificação de *software* (SV-COMP) [21] [22] [23].

2.3.1 Arquitetura do ESBMC

O processo de verificação é automatizado e não exige que o usuário insira pré e pós-condições nos programas, e não é necessário alterar o programa original. A Figura 2.10 apresenta a arquitetura do ESBMC que verifica programas ANSI-C e C++ [10].

O primeiro passo durante a verificação como mostra a Figura 2.10 é a leitura (*scan*) do código à ser verificado, nesta etapa é realizada a análise sintática que determina se a estrutura gramatical do código segue a gramática formal. Esta etapa cria uma representação intermediária do programa, transforma o texto de entrada em uma estrutura de dados, uma árvore de sintaxe abstrata que na Figura 2.10 é representado por *Parse Tree*.

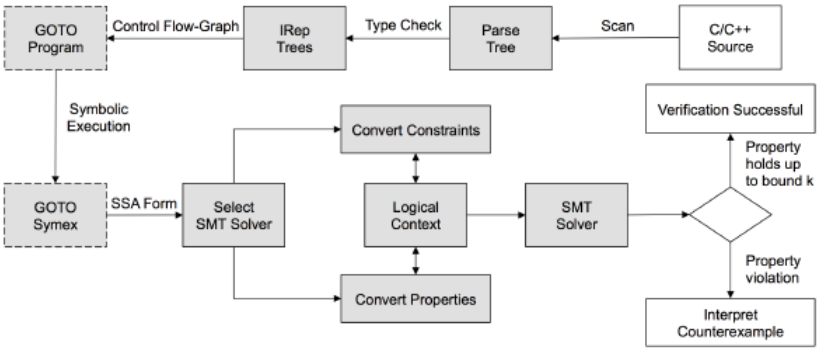

Figura 2.10: Arquitetura da ferramenta ESBMC [3].

No segundo passo no *type-checking*, são adicionadas verificações de atribuição, de *casting*, de inicialização de ponteiros e de chamada de funções, gerando uma árvore de representação intermediária (do inglês, *Intermediate Representation* - IRep).

O próximo passo é usar um grafo de controle de fluxo a partir da árvore IRep e gerar um programa GOTO equivalente. Isso permite simplificar as representações de instruções transformando expressões como *switch* e *while* em instruções *if* e *goto*.

O programa *goto* é executado simbolicamente pelo Goto-Symex e gera uma atribuição estática individual (do inglês *Single Static Assignment* - SSA). Depois é convertido em uma fórmula do tipo SMT e solucionado por um solucionador SMT. Se existir uma violação na propriedade, a geração do contraexemplo é realizada e o erro encontrado é informado [10] [18].

2.4 Modelo Operacional

O modelo operacional consiste basicamente de uma representação de um conjunto de métodos ou estruturas da linguagem de programação a ser verificada [10], [24], [25] . Por ser desenvolvido separadamente, ele é fornecido ao *model checking* no início da verificação como apresenta a Figura 2.11, funcionando como uma extensão da verificação de modelos.

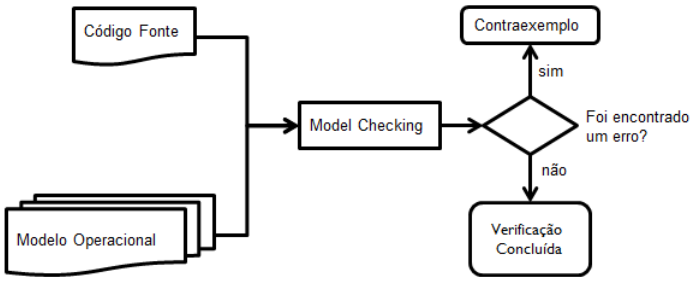

Figura 2.11: Associação do Modelo Operacional com o *Model Checking*.

No momento da verificação, o código fonte a ser verificado e o modelo operacional são carregados pelo verificador. Sem uma especificação não seria possível o verificador interpretar um método da nova linguagem que deseja-se verificar, cabe ao modelo operacional conter essa representação. Algumas ferramentas que suportam o uso de modelo operacional são o ESBMC [10], o CBMC [26], o LLBMC [27] e o

DiVinE [28].

De acordo com a técnica de verificação, todo método usado no código a ser verificado precisa ter seu modelo operacional implementado, assim o verificador terá uma especificação do comportamento do método [24]. Porém, existem casos que a implementação é somente uma assinatura do método, como métodos de impressão na tela, uma vez que não há propriedade a ser verificada [10], [24]. Essa simplificação é uma vantagem da abordagens de modelos operacionais, pois são inseridos apenas códigos úteis para verificação de propriedades, e desconsidera chamadas irrelevantes [10], [24], o que implica em menos estruturas e consequentemente em menos tempo de verificação. Muitas vezes quando não existem propriedades no método para verificar, a assinatura do modelo operacional serve apenas para que o verificador não apresente problemas de análise sintática.

Outras características positivas da abordagem do modelo operacional, é a possibilidade de inserir assertivas para verificar pré e pós condições dos modelos. As pré-condições são as condições mínimas para que a funcionalidade do modelo seja utilizada corretamente [29].

A Figura 2.12 apresenta um modelo operacional para a função *cudaMalloc()*, que foi desenvolvida neste trabalho e é melhor descrita na Seção4.3.6.

```
1  cudaError_t cudaMalloc(void ** devPtr, size_t size) {
2      cudaError_t tmp;
3      //pre-conditions
4      __ESBMC_assert(size > 0, "Size to be allocated may not be ...
           less than zero");
5      *devPtr = malloc(size);
6
7      address[counter] = *devPtr; counter++;
8      if (*devPtr == NULL) {
9          tmp = CUDA_ERROR_OUT_OF_MEMORY;
10         exit(1);
11     } else {
12         tmp = CUDA_SUCCESS;
13     }
14     //post-conditions
15     __ESBMC_assert(tmp == CUDA_SUCCESS, "Memory was not allocated");
16     lastError = tmp;
17     return tmp;
18 }
```

Figura 2.12: MOC *cudaMalloc()*.

Para exemplificar o uso de assertivas na verificação de pré-condição, tem-se a Figura 2.12, linha 4, que mostra um *assert* padrão do verificador usado, o *__ESBMC_assert()*. Neste exemplo o *assert* é usado para determinar se o parâmetro *size* que indica o tamanho de memória a ser alocado, não é um valor menor ou igual a zero, o que caracterizaria uma falha.

O processo de criação do modelo operacional necessita compreender a linguagem a qual será criado o modelo operacional. A documentação da linguagem é fundamental para dominar as suas estruturas, métodos, variáveis, relacionamento entre métodos e todo o comportamento dos elementos da linguagem. Para manter um boa organização dos códigos dos modelos o ideal é manter a mesma estrutura organizacional das bibliotecas da linguagem. Isso facilita nas dependências entre os métodos e na manutenção posterior do modelo, principalmente para os desenvolvedores que não atuaram inicialmente no desenvolvimento.

2.4.1 Modelo operacional e o ESBMC

O uso dos modelos operacionais com o ESBMC funciona assim como é apresenta na Figura 2.11, onde o *model checking* (ESBMC) carrega o código fonte e o modelo. No caso do ESBMC, a *flag -I* é chamada durante a verificação para fazer a conexão com o modelo operacional, sendo necessário escrever o endereço para o local do modelo juntamente com a *flag*. Quanto ao código, este é lido normalmente como um código ANSI-C ou C++

Quando é feita a leitura do modelo operacional, no primeiro passo da verificação como mostra a Figura 2.10, o modelo é associado ao código fonte e forma a árvore de sintaxe abstrata, cria-se uma representação intermediária única para programa e o modelo operacional.

A partir da árvore de sintaxe abstrata, o fluxo da verificação segue conforme mostrado na Seção 2.3.1, passando pelo *type check*, gerando a árvore IRep, o programa GOTO equivalente, executando o *goto* simbolicamente até a criação das fórmulas SMT, e solucionado pelos solucionadores.

Capítulo 3

Trabalhos Relacionados

Atualmente existem algumas ferramentas de verificação para códigos da plataforma CUDA. A tendência é que a quantidade desses verificadores se eleve, devido à variedade de aplicações que vem surgindo, e a complexidade dos sistemas criados com o CUDA. Neste trabalho estão listados três verificadores que suportam a plataforma CUDA e que são considerados estado da arte, são o PUG [30], o GPUVerify [31] e o GKLEE [32].

Essas ferramentas citadas foram usadas para comparar os resultados da verificação com o ESBMC-GPU, ferramenta criada a partir do ESBMC e que faz uso do modelo operacional descrito neste trabalho.

A técnica do uso de modelos operacionais que auxiliou na criação do ESBMC-GPU, tem obtido bons resultados para geração de ferramentas de verificação, como exemplo o QtOM [29] [25] usado para verificar programas escritos com o *framework* Qt, e o próprio ESBMC V1.25 [10] para suportar a verificação de códigos em C++. Ambas ferramentas possuem a mesma metodologia de desenvolvimento seguidas neste trabalho e têm como uso as ferramentas ESBMC V1.22 [17] e ESBMC V1.25 [10].

3.1 Verificadores para CUDA

3.1.1 PUG

A ferramenta Prover of User GPU Programs (PUG) é um verificador simbólico, que usa solucionadores SMT para analisar *kernels* automaticamente e detectar erros de corrida de dados, sincronização de barreira e conflitos de banco em memória compartilhada [30]. O verificador tem implementado o algoritmo para redução de ordem parcial (MPOR), buscando mitigar o problema da explosão do espaço de estados [30]. Para realizar uma verificação com o PUG são necessárias várias modificações no código fonte, primeiro é preciso que a extensão do arquivo seja alterado de ".cu" para ".c", é necessário inserir no código fonte as bibliotecas do verificador *cutil.h* e *config.h*, o nome do *kernel* obrigatoriamente precisa ser "*kernel*", e é necessário remover a função *main()*, pois a ferramenta tem suporte somente ao *kernel* do programa.

3.1.2 GPUVerify

O GPUVerify é uma ferramenta de verificação de *kernels* GPUs, ela usa a análise semântica para a verificação de corrida de dados e divergência de barreira [31]. Sua entrada é somente o *kernel* sendo incapaz de verificar a função *main()*.

A ferramenta define divergência de barreira quando um grupo de *threads*, ao passa por uma barreira, não sejam habilitadas uniformemente. Para a verificação de corrida de dados, o cenário é quando as *threads* do mesmo grupo intercalam completamente de forma assíncrona, sem garantias quanto à ordem relativa de execução de instrução entre *threads* [31].

Para realizar a verificação é necessário algumas modificações, remover a função *main()*, inicializar variáveis usadas pelo *kernel*, substituir os *asserts* no kernel pela função *asserts* intrínseca do GPUVerify, remover funções exclusivas das bibliotecas ANSI-C e C++, pois não são suportadas pelo GPUVerify [24], [31].

3.1.3 GKLEE

O GKLEE é um verificador baseado em execução *concolic* (concreto mais simbólico) [24]. Verifica códigos CUDA C++ para detecção de corrida de dados e sincronização de barreira. Diferente do PUG e do GPUVerify que atua somente no *kernel*, este é capaz de verificar todo o código, evitando fazer alterações no código fonte [24].

Para realizar a verificação com o GKLEE é usado dois comandos de execução o *gklee-nvcc* e *gklee* O primeiro é usado com o código fonte, este gera um arquivo ".cpp" e um executável. O segundo comando é usado com o executável que foi gerado [24], [32].

3.2 Modelos Operacionais ESBMC

3.2.1 Modelo Operacional C++

O ESBMC era capaz de verificar somente códigos ANSI-C até a versão 1.22, a criação do Modelo Operacional C++ (do inglês, Operation Model C++ - COM) permitiu o ESBMC verificar códigos C++ e criou a versão 1.25 do ESBMC [10].

Durante o desenvolvimento do COM não foram usadas as fontes disponíveis das bibliotecas do C++, isso tornaria mais complicado as condições de verificação (do inglês, *Verification Condition - VC*) por ser estruturas complexas originais. Em vez disso foram desenvolvidas representações simplificadas para essas bibliotecas, sendo capaz de representar as classes, os métodos e outras características semelhantes as da estrutura real [10].

No processo de verificação, o modelo COM substitui as bibliotecas correspondentes do C++ [10]. A maior parte do COM referencia as bibliotecas padrão *Standard Template Library* (STL), onde estão grande parte das funcionalidades exigidas.

3.2.2 Modelo Operacional QT

O modelo operacional QT (do inglês, Operation Model Qt - QtOM) tem a mesma abordagem que o COM, mas este se aplica ao conjunto padrão de bibliotecas

Qt [25].

O Qt contém estruturas hierárquicas e complexas, e não existe um *model checker* para verificação do *framework* Qt, isso motivou a criação do modelo operacional QtOM e sua integração ao ESBMC [29].

A vantagem é que o QtOM é escrito em C++, sendo assim faz uso de modelos já implementados do COM que foram inseridos ao ESBMC. O objetivo do QtOM é conter as estruturas necessárias para verificar propriedades relacionadas com o *framework* Qt [29].

O ESBMC V 1.25 é uma associação do modelo operacional COM com o ESBMC V 1.22, enquanto que o QtMO é modelo operacional que, associado ao ESBMC V1.25, verifica códigos do *framework* Qt.

Capítulo 4

Modelo Operacional CUDA

Este capítulo apresenta a metodologia seguida para o desenvolvimento do modelo operacional CUDA (MOC). Inicialmente é descrito o processo para a constbrução de uma suíte de testes, em seguida é mostrado o levantamento de características do CUDA em quem o ESBMC C++ não suporta, e por fim é mostrada a implementação dessas características que constituirá o MOC.

4.1 Suíte de Teste

Essa etapa foi realizada em conjunto com integrantes do projeto desenvolvido no CETELI. Durante o levantamento bibliográfico foram encontradas a suíte de testes de duas ferramentas de verificação de *kernels* GPUs, o GPUVerify [31] e o PUG [30]. Os *benchmarks* encontrados continham apenas a função *kernel*, sem a função *main()*, pois essas ferramentas verificam somente os *kernels*.

Os *benchmarks* encontrados foram usados para compor a suíte de testes deste trabalho. Outros *benchmarks* foram criados especificamente para checar propriedades como *array bounds* e *data race*.

Ao todo são 160 *benchmarks* e estão divididos nos seguintes diretórios com as respequitivas quantidades: *array-bounds* (6), *assertion* (8), *data-racer* (14), *null-pointer* (5), *pass-fail* (90) e *not-supported* (37).

Os *benchmarks* escrevem em ponteiros, realizam chamadas de funções *device*, realizam operações matemáticas (soma, subtração, multiplicação), possuem funções específicas do C (*menset()*, *assert()*), funções específicas do CUDA (*atomicAdd()*,

cudaMemcpy(), *cudaMalloc()*, *cudaFree()*), de bibliotecas específicas do CUDA (*curand.h*, *cuda.h*, *vector_types.h*), trabalham com vários tipos e modificadores (*int*, *float*, *char*, *long*, *unsigned*), *typedefs* e tipos intrínsecos ao CUDA (*uint4*, *dim3*) [24]. Os *benchmarks* fazem uso de inúmeras estruturas de programação CUDA e C, possuem entre 60 e 90 linhas de códigos, todos com a função *main()* e chamadas de *kernel*.

4.1.1 Reconstrução dos casos de testes

As ferramentas encontradas só verificam os *kernels*, consequentemente suas suítes de testes só possuem os *kernels* de um programa CUDA. A proposta do ESBMC-GPU é verificar todos as partes de um programa escrito em CUDA. Além dos *kernels*, busca-se verificar a função *main()* do programa, por isso os casos de testes tiveram que receber linhas de código adicionais, para compor programas completos escritos em CUDA.

A Figura 4.1 apresenta um *benchmark* do GPUVerify, onde existe apenas o *kernel foo*. Neste exemplo tem-se o endereço de vetor C passado por parâmetro, o *threadIdx.x* indexa o vetor que recebe a soma de a e b. O *threadIdx.x* é o Id da *thread*, várias *threads* executam esse *kernel* ao mesmo tempo, porém acessam posições diferentes do vetor já que os *Ids* das *threads* são diferentes.

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

Figura 4.1: *Benchmark* obtido da base de testes do GPUVerify.

Os *benchmarks* obtidos no GPUVerify tiveram que passar por alterações, para possuir a função *main()*, isso é essencial pois é na *main()* que a chamada de *kernel* é realizada. A Figura 4.2 apresenta o código CUDA completo correspondente ao código da Figura 4.1.

No código da Figura 4.2 foi inserida a função *main()*, foram criadas e instanciadas as variáveis necessárias, foi feita a alocação de memória na GPU na linha 12,

```
1  #include <cuda.h>
2  #define N 2
3  __global__ void foo(int *c) {
4    int a = 2;
5    int b = 3;
6    c[threadIdx.x]= a+b;
7  }
8
9  int main(){
10     int *a; int *dev_a;
11     a = (int*)malloc(N*sizeof(int));
12     cudaMalloc((void**)&dev_a, N*sizeof(int));
13     cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
14
15     foo<<<1, N>>>(dev_a);
16
17     cudaMemcpy(a, dev_a, N*sizeof(int), cudaMemcpyDeviceToHost);
18     for (int t=0;t<N;t++){
19         printf ("%d ", a[t]);
20         assert(a[t]==5);
21     }
22     cudaFree(dev_a);
23     free(a);
24     return 0;
25 }
```

Figura 4.2: Reconstrução do código obtido da base de testes do GPUVerify.

cópia para a GPU na linha 13, chamada de *kernel* na linha 15 e a cópia de volta para a CPU na linha 17. Na linha 20 foi usando a função *assert()* para checar se as posições do vetor possuem o valor 5. Por fim, as linhas 22 e 23 liberam as memórias da GPU e da CPU.

Assim como no exemplo mostrado nas Figuras 4.1 e 4.2, todos os *benchmarks* provenientes do GPUVerify tiveram que passar por mudanças. Para validar se os *benchmarks* estavam sintaticamente corretos, os mesmos foram executados na IDE *nsight* [33]. Pequenos ajustes em alguns *benchmarks* foram necessários até obter sua execução correta nesta IDE.

4.1.2 Adaptação dos *benchmarks*

Todos os programas da base de dados criados tiveram modificações na chamada de *kernel*, o parser original conforme mostra a linha 2 da Figura 4.3. Foi alterado para realizar o *parser* convencional de uma função *C*, conforme mostra a linha 3 da mesma figura. Essa mudança foi realizada devido ao ESBMC ainda não

suporta o *parser* do CUDA.

```
1       ...
2       //foo<<<1, N>>>(dev_a);
3       ESBMC_verify_kernel(foo,1,N,dev_A);
4       ...
```

Figura 4.3: Mudança na chamada *kernel* dos *benchmarks*.

4.2 Levantamento de características

Todos os *benchmarks* foram analisados após constituída a suíte de teste, assim as funções, tipos de dados, qualificadores, estruturas e características intrínsecas ao CUDA puderam ser identificadas. Identificar esses elementos é tedioso mas um importante passo, pois são trechos de códigos que o ESBMC não é capaz de interpretar. O ESBMC não possui nenhuma definição para esses elementos internamente, logo não saberia proceder durante a criação da árvore de *parser* como mostra a Figura 2.10 na Seção 2.3.1. Como visto na Seção 2.4, o modelo operacional permite a ferramenta abstrair a representação dessas características, dando continuidade no fluxo de código da ferramenta.

A Tabela 4.1 apresenta os elementos pertencentes ao CUDA que foram encontrados nos *benchmarks*.

Tabela 4.1: Características do CUDA para implementar

Características	**Exemplo**
Qualificador de Tipo de Função	__device__, __global__, __host__
Qualificador de tipo de variáveis	__device__, __constant__, __shared__, __managed__
Tipos de Vetor	int1, int2, int3, int4, float1, float2, float3, float4
Variáveis Internas	gridDim, blockDim, threadIdx, blockIdx
Funções Gerenciamento de Memória	cudaMalloc(), cudaMemcpy(), cudaFree()
Funções Atômicas	atomicAdd(), atomicAdd(), atomicSub(), atomicInc(), atomicDec(), atomicAnd()
Funções Matemáticas	powf(), sqrtf(), cosf(), logf()
Outros	cudaError_t, chamada de *kernel*

4.3 Modelagem e implementação

A plataforma CUDA é fechada e não disponibiliza o código fonte das suas funções e estruturas de programação, isto dificulta a implementação do modelo operacional. Todos os itens identificados na Seção 4.2 precisam ter seu comportamento modelado antes da implementação. O guia de programação CUDA [1] foi usado para entender o que cada item faz, quais parâmetros de entrada, qual tipo de retorno. Foram verificados itens que dependem de outros, por isso foram implementados antes os itens primários como *enuns*, qualificadores, *structs*, tipos de dados, e por último as funções.

O modelo operacional será composto de *8* arquivos que fazem referência as bibliotecas nativas do CUDA. Estes arquivos possuirão as implementação dos modelos referente aos elementos identificados que pertencem a biblioteca de mesmo nome. Dessa forma, tenta-se preservar a estrutura organizacional do CUDA no MOC.

4.3.1 *cudaError_t*

A maioria das funções em CUDA retornam um tipo *cudaErro_t* e os resultados são passados por parâmetros usando ponteiros. O *cudaError_t* é um tipo de dado que armazena tipos de erros. Quando existe alguma falha na execução de uma função, é possível identificar o que houve pelo *cudaError_t* retornado.

Sua implementação é apresentada na Figura 4.4, um *enum* é descrito com os tipos de erros, são 78 tipos de erros, conforme o Guia de Programação CUDA [1]. Sua implementação está definida no arquivo *cuda_runtime_api*.

```
typedef enum cudaError {

    CUDA_SUCCESS = 0, cudaSuccess = 0,
    CUDA_ERROR_INVALID_VALUE = 1,
    CUDA_ERROR_OUT_OF_MEMORY = 2,
    CUDA_ERROR_NOT_INITIALIZED = 3,
    CUDA_ERROR_DEINITIALIZED = 4,
    CUDA_ERROR_PROFILER_DISABLED = 5,

    . . .

    CUDA_ERROR_UNKNOWN = 999

} CUresult;

typedef enum cudaError cudaError_t;
```

Figura 4.4: Modelo para o *cudaError_t*.

4.3.2 Qualificador de Tipo de Função

Esses qualificadores identificam se a função será executada no *host* ou no *device*, e define se a função foi chamada pelo *host* ou pelo *device* [1].

Qualificadores de tipo de função são *__device__*, *__global__*, *__host__*. O qualificador *__device__* declara que uma função executa no *device* e somente pode ser chamada por funções descritas no *device*. O qualificador *__host__* declara que a função é executada no *host* e pode ser chamada somente no *host*. O qualificador *__global__* declara a função como sendo um *kernel*, logo é executada no *device*, mas sua chamada é realizada no *host*.

A abordagem neste trabalho quanto aos qualificadores visa que a ferramenta não tenha problemas com o *parser* durante a verificação, ainda não há uma representação interna que faça distinção entre *host* e *device*. Portanto sua representação consiste de *defines* conforme mostra a Figura 4.5, sua representação está no arquivo *host_defines.h*.

```
#define __device__ void
#define __host__ void
#define __global__ void
```

Figura 4.5: Funções tipo qualificadores.

4.3.3 Qualificador de Tipo de Variáveis

Os qualificadores de variáveis especificam a localização da variável dentre os tipos de memória da GPU. Assim como na Seção 4.3.2, esses qualificadores são *defines*, tendo objetivo de evitar erros com o *parser*. A Figura 4.6 apresenta os qualificadores de tipo de variáveis.

```
1 #define __device__ void
2 #define __constant__ void
3 #define __shared__ void
4 #define __managed__ void
```

Figura 4.6: Variáveis tipo qualificadores.

4.3.4 Tipos de Vetor

Dentre a plataforma CUDA são definidos tipos de vetores que derivam de tipos básicos como *int*, são estruturas podendo conter de 1 a 4 componentes, com campos x, y, z e w. A Figura 4.8 mostra os tipos de vetores *int1*, *int2*, *int3* e *int4*.

```
1  ...
2  struct __device_builtin__ __int1{
3      int x;
4  };
5  struct __device_builtin__ __int2{
6      int x, y;
7  };
8  struct __device_builtin__ __int3{
9      int x, y, z;
10 };
11 struct __device_builtin__ __int4{
12     int x, y, z, w;
13 };
14 typedef __device_builtin__ struct __int1 int1;
15 typedef __device_builtin__ struct __int2 int2;
16 typedef __device_builtin__ struct __int3 int3;
17 typedef __device_builtin__ struct __int4 int4;
18 ...
```

Figura 4.7: Tipos de vetor *int*.

Todos os outros tipos de dados seguem a mesma abordagem do *int* mostrado na Figura 4.8. A Tabela 4.2 mostra os outros tipos conforme o tipo básico e a quantidade de campos. Essas estruturas são descritas no arquivo *vector_types.h*.

Tabela 4.2: Tipos de vetores.

Tipo Básico	Tipo Derivado
char	char1, char2, char3, char4
uchar	uchar1, uchar2, uchar3, uchar4
short	short1, short2, short3, short4
ushort	ushort1, ushort2, ushort3, ushort4
int	int1, int2, int3, int4
uint	uint1, uint2, uint3, uint4
long	long1, long2, long3, long4
ulong	ulong1, ulong2, ulong3, ulong4
longlong	longlong1, longlong2
ulonglong	ulonglong1, ulonglong2
float	float1, float2, float3, float4
double	double1, double2

Outro tipo importante é o *dim3*, é baseado no *uint3* visto na Tabela 4.2. Esse tipo possui 3 componentes e é usado para especificar as dimensões de *threads* e blocos.

```
1 struct __device_builtin__ __dim3{
2     unsigned int x, y, z;
3 };
4
5 typedef __device_builtin__ struct __dim3 dim3;
```

Figura 4.8: Tipos dim3.

4.3.5 Variáveis Internas

Na plataforma CUDA existem variáveis que determinam as dimensões da grade, do bloco e os índices dos blocos e *threads*. Elas são usadas internamente pelas funções *kernel* e foram descritas na Seção 2.1.4.

A variável *gridDim* determina o tamanho da grade nas três dimensões, determinando o número de blocos que serão executados na chamada de *kernel*. A variável *blockDim* determina o tamanho do bloco nas três dimensões, ou seja, o número de *threads* de cada bloco. Essas duas variáveis são do tipo *dim3* mostrado na Seção4.3.4.

A variável *threadIdx* determina o índice das três componentes da *thread* dentro do bloco. A variável *blockIdx* determina o índice das três componentes do bloco

dentro da grade. Essas variáveis são vetores do tipo *unit3* como mostra a Figura 4.9.

```
1 uint3 indexOfThread[1024];
2 uint3 indexOfBlock[1024];
3 #define threadIdx indexOfThread[__ESBMC_get_thread_id()-1]
4 #define blockIdx  indexOfBlock[__ESBMC_get_thread_id()-1]
```

Figura 4.9: Representação para *blockIdx* e *blockIdx*.

No CUDA uma *thread* pode ser acessada por seu *ID* ou pelo *threadIdx*. Como está sendo usado o modelo operacional da biblioteca *pthread* existente no ESBMC para representar as *threads* CUDA. As *threads* possuem um *ID* onde é preciso determinar os valores para o *threadIdx*.

Na Figura 4.10 é apresentada a função *getThreadIdx()* que gera os índices das *threads* a partir do seu *ID*.

```
1  uint3 getThreadIdx(unsigned int id){
2      __ESBMC_atomic_begin();
3
4      unsigned int linear_value_total = id;
5      unsigned int block_size = blockDim.x *
6                                blockDim.y *
7                                blockDim.z;
8      unsigned int grid_position = (unsigned int)
9                          indexOfBlock[id].x +
10                         indexOfBlock[id].y*gridDim.x +
11                         indexOfBlock[id].z*gridDim.x*gridDim.y;
12     unsigned int linear_value = linear_value_total -
13                                  grid_position*block_size;
14
15     uint3 thread_index;
16     thread_index.z = (unsigned int) (linear_value /
17                         (blockDim.x * blockDim.y));
18     thread_index.y = (unsigned int)
19                         ((linear_value %
20                         (blockDim.x*blockDim.y))/
21                          blockDim.x );
22     thread_index.x = (unsigned int)
23                          ( (linear_value %
24                          (blockDim.x*blockDim.y))%
25                         blockDim.x );
26     return thread_index;
27
28     __ESBMC_atomic_end();
29 }
```

Figura 4.10: Função *getThreadIdx()*.

Na Figura 4.11 é apresentado a função *getBlockIdx()*, esta gera os índices do *bloco*.

```
uint3 getBlockIdx(unsigned int id){
    __ESBMC_atomic_begin();

    unsigned int linear_value = (unsigned int) id
                    / (blockDim.x*blockDim.y*blockDim.z);

    uint3 block_index;
    block_index.z = (unsigned int)
                    (linear_value / (gridDim.x * gridDim.y));
    block_index.y = (unsigned int)
                    ( (linear_value %
                    (gridDim.x*gridDim.y)) / gridDim.x );
    block_index.x = (unsigned int)
                    ( (linear_value %
                    (gridDim.x*gridDim.y))%
                    gridDim.x );

    return block_index;

    __ESBMC_atomic_end();
}
```

Figura 4.11: Função *getBlockIdx()*

A função *assignIndexes()* usa a *getThreadIdx()* e *getBlockIdx()* para gerar o *threadIdx* e *blockIdx* conforme todas as *threads*.

```
void assignIndexes(){
    __ESBMC_atomic_begin();
    int i;
    for(i = 0; i < 2; i++)
        indexOfBlock[i] = getBlockIdx(i);
    for(i = 0; i < 2; i++)
        indexOfThread[i] = getThreadIdx(i);
    __ESBMC_atomic_end();
}
```

Figura 4.12: Variáveis Internas.

O código descrito nesta seção está situado no arquivo *device_launch_parameters.h* do modelo operacional.

4.3.6 Funções de Gerenciamento de Memória

Esta seção descreve funções que estão relacionadas ao gerenciamento de memória entre o *host* e o *device*. Como visto na Seção 2.1.5, qualquer programa CUDA necessita alocar e liberar memória do *device* e transferir dados entre o *host* e *device*. As principais funções usadas neste processo são *cudaMalloc() cudaMemcpy()* e *cudaFree()*.

Dentre a proposta do modelo operacional não há uma distinção entre o *host* e *device*, mas é necessário simular suas operações de alocação de memória e transferência de dados. O ESBMC já possui implementação de funções referentes a memória para C e C++, o modelo operacional das funções em CUDA se apropria dessas funções já existentes do ESBMC.

A função *cudaMalloc()* é responsável pela alocação de memória no *device*, sua implementação é apresentada na Figura 4.13. O parâmetro *devPtr* é o ponteiro para alocar memória no *device*, e *size* indica o tamanho em bytes para alocação. A função tem como retorno o *cudaError_t cudaSuccess* quando a alocação é realizada corretamente e *cudaErrorMemoryAllocation* quando ocorre uma falha.

```
1  cudaError_t cudaMalloc(void ** devPtr, size_t size) {
2
3      cudaError_t tmp;
4      //pre-conditions
5      __ESBMC_assert(size > 0, "Size to be allocated may not be ...
           less than zero");
6      *devPtr = malloc(size);
7
8      address[counter] = *devPtr; counter++;
9
10     if (*devPtr == NULL) {
11         tmp = CUDA_ERROR_OUT_OF_MEMORY;
12         exit(1);
13     } else {
14         tmp = CUDA_SUCCESS;
15     }
16
17     //post-conditions
18     __ESBMC_assert(tmp == CUDA_SUCCESS, "Memory was not allocated");
19
20     lastError = tmp;
21     return tmp;
22 }
```

Figura 4.13: Modelo operacional da função *cudaMalloc()*.

A função *cudaMemcpy()* é responsável pela transferência de dados entre o *host* e *device*, sua implementação é apresentada na Figura 4.15. O parâmetro *dst* é o endereço de memória do destino, *src* é o endereço de memória da origem, *count* o tamanho em bytes e *kind* é o tipo de transferência.

O *cudaMemcpyKind* é a estrutura que possui as transferências realizadas pelo *cudaMemcpy()*, está descrito no arquivo *driver_types.h*. A Tabela 4.3 apresenta os tipos de transferências, e a Figura 4.14 sua implementação.

Tabela 4.3: Tipos de transferências entre *host* e *device*.

Tipo Transferência	**Especificação**
Host -> Host	cudaMemcpyHostToHost
Host -> Device	cudaMemcpyHostToDevice
Device -> Host	cudaMemcpyDeviceToHost
Device -> Device	cudaMemcpyDeviceToDevice

```
1 enum __device_builtin__ cudaMemcpyKind{
2     cudaMemcpyHostToHost          =   0,
3     cudaMemcpyHostToDevice        =   1,
4     cudaMemcpyDeviceToHost        =   2,
5     cudaMemcpyDeviceToDevice      =   3
6 };
```

Figura 4.14: Modelo operacional do *cudaMemcpyKind*.

```
1  cudaError_t __cudaMemcpy(void *dst, const void *src, size_t ...
       count, enum cudaMemcpyKind kind)
2  {
3      __ESBMC_assert(count > 0, "Size to be allocated may not be ...
           less than zero");
4
5      char *cdst = (char *) dst;
6      const char *csrc = (const char *)src;
7      int numbytes = count/(sizeof(char));
8
9      for (int i = 0; i < numbytes; i++)
10         cdst[i] = csrc[i];
11
12     lastError = CUDA_SUCCESS;
13     return CUDA_SUCCESS;
14 }
```

Figura 4.15: Modelo operacional da função *cudaMemcpy()*.

A função *cudaFree()* é responsável por liberar a memória no *device*, a Figura 4.16 apresenta sua implementação. O parâmetro *devPtr* aponta para o endereço de memória no *device* a ser liberar.

```
1 cudaError_t cudaFree(void *devPtr) {
2     free(devPtr);
3     lastError = CUDA_SUCCESS;
4     return CUDA_SUCCESS;
5 }
```

Figura 4.16: Modelo operacional da função *cudaFree()*.

4.3.7 Funções Atômicas

As funções atômicas realizam uma leitura, processa o dado e escreve o resultado em memória como uma única operação, garante que essas etapas citadas sejam executadas em uma *thread* sem interferência de outras *threads*. Nenhuma outra *thread* pode acessar o endereço de memória que está em uso por uma função atômica. Como o modelo operacional será usado em conjunto com a ferramenta ESBMC, pode-se usar os elementos do ESBMC dentro do modelo. O ESBMC possui duas funções, o *__ESBMC_atomic_begin()* e *__ESBMC_atomic_end()*, garantem que o código escrito entre elas sejam executadas sem interferências de outras *threads*.

O modelo da função *atomicAdd()* é apresentada na Figura 4.17. O valor do endereço de memória *address* é lido e somado com o valor *val*, o modelo retorna o valor antigo de *address*, e o resultado da soma é armazenado em *address*.

```
1  extern __device__ int __iAtomicAdd(int *address, int val){
2      __ESBMC_atomic_begin();
3        int old_value, new_value;
4        old_value = *address;
5        new_value = old_value + val;
6        *address = new_value;
7        return old_value;
8      __ESBMC_atomic_end();
9  }
10 static __device__ int atomicAdd(int *address, int val){
11         return __iAtomicAdd(address, val);
12 }
```

Figura 4.17: Modelo operacional da função *atomicAdd()*.

O código apresentado na Figura 4.17 refere-se a soma para o tipo *int*, mas foram implementados a soma atômica para outros tipos conforme a Tabela 4.4.

Tabela 4.4: Assinaturas para tipos de *atomicAdd()*

Tipo Básico	**Assinatura**
int	atomicAdd(int* address, int val);
unsigned int	atomicAdd(unsigned int* address, unsigned int val);
unsigned long long	atomicAdd(unsigned long long* address, unsigned long long val);
float	atomicAdd(float* address, float val);

O modelo da função *atomicSub()* realiza uma subtração seguindo o mesmo código da *atomicAdd()*, apenas o sinal positivo que muda para negativo na linha 7 da Figura 4.17. Os mesmos tipos encontrados na Tabela 4.4 se aplica para *atomicSub()*.

Todas as funções atómicas tem como base a implementação da Figura 4.17, mudando apenas a operação interna.

A Figura 4.18 apresenta o modelo operacional da função *atomicMin()*. A função elege o menor valor entre o parâmetro *val* e o valor contido no endereço *address*, armazena este valor em *address* e retorna o valor antigo de *address*.

```
1  extern __device__ int           __iAtomicMin(int *address, int val){
2
3      __ESBMC_atomic_begin();
4      int old_value;
5
6      old_value = *address;
7      if(val < old_value)
8          *address = val;
9
10     return old_value;
11     __ESBMC_atomic_end();
12 }
13
14 static __inline__ __device__ int atomicMin(int *address, int val){
15   return __iAtomicMin(address, val);
16 }
```

Figura 4.18: Modelo operacional da função *atomicMin()*.

A função *atomicMax()* é semelhante ao *atomicMin()*, mas elege o maior valor entre *val* e *address*. Quando os valores de *val* e *address* são iguais no *atomicMin()* e *atomicMax()*, não há alteração do endereço *address*.

Foram implementados os tipos *int*, *unsigned int*, *long long* e *unsigned long long* para o *atomicMin()* e *atomicMax()*.

A Figura 4.19 apresenta o modelo operacional do *atomicInc()*. A função *atomicInc()* incrementa o valor passado no endreço *address* em uma unidade caso não seja maior ou igual ao valor *val*, caso contrário é armazenado valor zero no endereço para iniciar a contagem.

```
1  extern __device__ unsigned int __uAtomicInc(unsigned int ...
       *address, unsigned int val){
2       __ESBMC_atomic_begin();
3       unsigned int old_value;
4       old_value = *address;
5       if(old_value >= val)
6           *address = 0;
7       else
8           *address = (old_value+1);
9       return old_value;
10      __ESBMC_atomic_end();
11 }
12 static __inline__ __device__ unsigned int atomicInc(unsigned int ...
       *address, unsigned int val){
13    return __uAtomicInc(address, val);
14 }
```

Figura 4.19: Modelo operacional da função *atomicInc()*.

O modelo da função *atomicDec()* é apresentado na Figura 4.20. Essa função decrementa o valor do endereço *address*, mas caso seja igual a zero ou maior que o valor *val*, *address* recebe o valor *val*.

```
1  extern __device__ unsigned int __uAtomicDec(unsigned int ...
       *address, unsigned int val){
2       __ESBMC_atomic_begin();
3       unsigned int old_value;
4       old_value = *address;
5       if((old_value == val) | (old_value > val))
6           *address = val;
7       else
8           *address = (old_value-1);
9
10      return old_value;
11      __ESBMC_atomic_end();
12 }
13
14 __device__ unsigned int atomicDec(unsigned int *address, ...
       unsigned int val){
15    return __uAtomicDec(address, val);
16 }
```

Figura 4.20: Modelo operacional da função *atomicDec()*.

A função *atomicExch()* tem seu modelo apresentado na Figura 4.21. A função substitui o valor contino no endereço *address* pelo valor *val*, e retorna o valor antigo. Foram implementados o *atomicExch()* para os tipos *int*, *unsigned int*, *unsigned long long int* e *float*.

```
1  extern __device__ int           __iAtomicExch(int *address, int val){
2      __ESBMC_atomic_begin();
3      int old_value;
4      old_value = *address;
5      *address = val;
6
7      return old_value;
8      __ESBMC_atomic_end();
9  }
10
11 static __inline__ __device__ int atomicExch(int *address, int val){
12   return __iAtomicExch(address, val);
13 }
```

Figura 4.21: Modelo operacional da função *atomicExch()*.

A função *atomicCas()* tem seu modelo apresentado na Figura 4.22. O *atomicCas()* compara o valor do endereço *address* com o parâmetro *compare*, se forem iguais *address* assume o valor de *val*, caso contrário *address* permanece inalterado. A função retorna o valor antigo de *address*.

```
1  extern __device__ int           __iAtomicCAS(int *address, int ...
       compare, int val){
2      __ESBMC_atomic_begin();
3
4      int old_value;
5      old_value = *address;
6      if(old_value == compare)
7          *address = val;
8      else
9          *address = old_value;
10
11     return old_value;
12     __ESBMC_atomic_end();
13 }
14
15 static __inline__ __device__ int atomicCAS(int *address, int ...
       compare, int val){
16   return __iAtomicCAS(address, compare, val);
17 }
```

Figura 4.22: Modelo operacional da função *atomicCas()*.

A função *atomicAnd()* é uma função bit a bit, realiza a operação *and* entre o valor de *address* e *valor*. A Figura 4.23 apresenta o modelo operacional de *atomicAnd()*.

```
1  extern __device__ int          __iAtomicAnd(int *address, int val){
2
3      __ESBMC_atomic_begin();
4      int old_value;
5      old_value = *address;
6      *address = (old_value & val);
7      return old_value;
8      __ESBMC_atomic_end();
9
10 }
11
12 __device__ int atomicAnd(int *address, int val){
13     return __iAtomicAnd(address, val);
14 }
```

Figura 4.23: Modelo operacional da função *atomicAnd()*.

A função *atomicOr()* é uma função bit a bit, realiza a operação *or* entre o valor de *address* e *valor*. A Figura 4.24 apresenta o modelo operacional de *atomicOr()*. Foram implementados os tipos *int* e *unsigned int*.

```
1  extern __device__ int          __iAtomicOr(int *address, int val){
2
3      __ESBMC_atomic_begin();
4      int old_value;
5      old_value = *address;
6      *address = (old_value | val);
7      return old_value;
8      __ESBMC_atomic_end();
9
10 }
11
12 __device__ int atomicOr(int *address, int val){
13     return __iAtomicOr(address, val);
14 }
```

Figura 4.24: Modelo operacional da função *atomicOr()*.

A função *atomicXor()* é uma função bit a bit, realiza a operação *Xor* entre o valor de *address* e *valor*. A Figura 4.25 apresenta o modelo operacional de *atomicXor()*. Foram implementados os tipos *int* e *unsigned int*.

```
1  extern __device__ int          __iAtomicXor(int *address, int val){
2      __ESBMC_atomic_begin();
3      int old_value;
4      old_value = *address;
5      *address = (old_value ^ val);
6      return old_value;
7      __ESBMC_atomic_end();
8  }
9  static __inline__ __device__ int atomicXor(int *address, int val){
10   return __iAtomicXor(address, val);
11 }
```

Figura 4.25: Modelo operacional da função *atomicXor()*.

4.3.8 Funções Matemáticas

O CUDA possuem bibliotecas especificas para funções matemáticas que são executadas no *device*, como exemplo a *Single Precision Intrinsics* que agrupam funções intrínsecas de precisão simples e, a *Single Precision Mathematical Functions* que agrupam as funções usadas diretamente pelos usuários. As funções da *Single Precision Mathematical Functions* em muitos casos fazem uso das funções intrínsecas. Durante o desenvolvimento do modelo operacional foram mantidas as funções intrínsecas, pois é possível que futuramente existam *benchmarks* que faça uso dessas funções. O modelo operacional de todas funções matemáticas estão estão concentradas em um único arquivo, o *math_functions.h*.

Inicialmente foram feitas algumas definições para valores fixos que são usados nos modelos como apresentado a Figura 4.26.

```
1  #define M_PI     3.14159265358979323846
2  #define M_PI_2   1.57079632679489661923132169164
3  #define NEPERIANO 2.718281828
4  #define PREC 1e-16
```

Figura 4.26: *Defines* no math_functions.h.

A seguir são apresentados modelos operacionais de funções logarítmicas.

A função *logf()* calcula o logaritmo natural de uma entrada, seu modelo é apresento na Figura 4.27. Foi criado o modelo da função intrínseca __*logf()* por meio da série de *Taylor* como mostrado na Equação 4.1, em seguida é mostrado o modelo para *logf()*.

```
1  __DEVICE_FUNCTIONS_DECL__ float __logf ( float  x ){
2      __ESBMC_assert(n < 0, "The number must be greater or equal ...
           than 0");
3      int i;
4      float r=0;
5
6      for(i=0;i<15;i++){
7          float pA = powf((x-1)/(x+1),((2*i)+1));
8          float pB = (1/((2*i) + 1));
9          r = r + pA*pB;
10     }
11     return r*2;
12 }
13 __device__ float logf (float x){
14     return __logf(x);
15 }
```

Figura 4.27: Modelo operacional da função *logf()*.

$$ln(z) = 2\sum_{n=0}^{\infty} \frac{1}{2n+1}\left(\frac{z-1}{z+1}\right)^{2n+1} \tag{4.1}$$

A função *log2f()* calcula o logaritmo na base 2 do valor de entrada, seu modelo é apresentado na Figura 4.28, faz uso da propriedade da potência do logaritmo, que afirma que quando um logaritmo estiver elevado a um expoente, esse expoente multiplica o resultado desse logaritmo, como mostra o elemento m na Equação 4.2.

$$log_a x^m = m * log_a x \tag{4.2}$$

```
1 __DEVICE_FUNCTIONS_DECL__ float __log2f ( float  x ){
2     __ESBMC_assert(n < 0, "The number must be greater or equal ...
          than 0");
3     return  x*__logf(2);
4 }
5 float log2f(float n){
6     return __log2f(x);
7 }
```

Figura 4.28: Modelo operacional da função *log2f()*.

A propriedade da potência do logaritmo mostrado na Equação 4.2 é usada também no modelo da função *log10f()*, esta calcula o logaritmo na base 10 do valor de entrada e seu modelo é apresentado na Figura 4.29.

```
1 __DEVICE_FUNCTIONS_DECL__ float __log10f ( float  x ){
2     __ESBMC_assert(n < 0, "The number must be greater or equal ...
          than 0");
3     return  x*logf(10);
4 }
5 __device__ float log10f (float x){
6     return __log10f(x);
7 }
```

Figura 4.29: Modelo operacional da função *log10f()*.

A seguir são apresentados os modelos para as funções exponencias.

A função *expf()* calcula o exponencial na base *neperiana* do valor de entrada. Seu modelo é apresentado na Figura 4.30 e faz uso de relações matemáticas para realizar o cálculo.

```
1 __DEVICE_FUNCTIONS_DECL__ float __expf ( float  x ){
2     return __powf(NEPERIANO,x);
3 }
4 __device__ float    expf (float x){
5     return __expf(x);
6 }
```

Figura 4.30: Modelo operacional da função *expf()*.

A função *exp10f()* calcula o exponencial na base *10* do valor de entrada. Seu modelo segue a mesma linha que o modelo da função *expf()* e é apresentado na Figura 4.31.

```
__DEVICE_FUNCTIONS_DECL__ float __exp10f ( float  x ){
    return __powf(10,x);
}
__device__ float exp10f (float x){
    return __exp10f(x);
}
```

Figura 4.31: Modelo operacional da função *exp10f()*.

A função *exp2ef()* calcula o exponencial na base *2* do valor de entrada. Seu modelo segue a mesma linha que o modelo da função *expf()* e é apresentado na Figura 4.32.

```
__device__ float exp2f (float x){
    return __powf(2,x);
}
```

Figura 4.32: Modelo operacional da função *exp2f()*.

A seguir são apresentados modelos para funções trigonométricas.

A função *cosf()* calcula o cosseno do valor de um entrada em radianos, seu modelo é apresentado na Figura 4.33.

```
__DEVICE_FUNCTIONS_DECL__ float __cosf ( float  x ){
    float t, s;
    int p = 0;
    s = 1.0; t = 1.0;
    x = fmodf(x + M_PI, M_PI * 2) - M_PI;
    double xsqr = x*x;
    double ab = 1;
    while((ab > PREC) && (p < 15)){
        p++;
        t = (-t * xsqr) / (((p<<1) - 1) * (p<<1));
        s += t;
        ab = (s==0) ? 1 : fabsf(t/s);
    }
    __ESBMC_assert((s < -1) || (s > 1), "The number must belong ...
       to the interval [-1, +1]");
    return s;
}
__device__ float    cosf (float x){
    return __cosf ( x );
}
```

Figura 4.33: Modelo operacional da função *cosf()*.

A função *sinf()* calcula o seno de uma entrada em radianos, seu modelo é apresentado na Figura 4.34.

```
__DEVICE_FUNCTIONS_DECL__ float     __sinf ( float  x ){
    return __cosf(x-M_PI_2);
}
__device__ float    sinf (float x){
    return __sinf( x );
}
```

Figura 4.34: Modelo operacional da função *sinf()*.

O modelo da função *tanf()* calcula a tangente do valor de uma entrada em radianos, seu modelo é apresentado na Figura 4.35

```
__device__ float    tanf (float x){
    return __tanf(x);
}
__DEVICE_FUNCTIONS_DECL__ float     __tanf ( float  x ){
    if(x==0)
        return 0;
    return __sinf(x)/__cosf(x);
}
```

Figura 4.35: Modelo operacional da função *tanf()*.

A função *sqrtf()* calcula a raiz quadrada do valor de entrada, seu modelo é apresentado na Figura 4.36.

```
__device__ float    sqrtf (float n){
      /*We are using n itself as initial approximation This can ...
         definitely be improved */
      float x = n;
      float y = 1;
      float e = 1;
      int i = 0;
      while(i++ < 15){ //Change this line to increase precision
        x = (x + y)/2.0;
        y = n/x;
      }
      return x;
}
```

Figura 4.36: Modelo operacional da função *sqrtf()*.

A função *powf()* calcula a potência y de um valor x conforme o modelo apresentado na Figura 4.37.

```
1  float   __powf ( float  x, float  y ){
2          int result = 1;
3
4          if (y == 0)
5              return result;
6
7          if (y < 0)
8              return 1 / pow(x, -y);
9
10         float temp = pow(x, y / 2);
11         if ((int)y % 2 == 0)
12            return temp * temp;
13         else
14            return (x * temp * temp);
15 }
16 __device__ float    powf (float a, float b){
17         return __powf ( a , b );
18 }
```

Figura 4.37: Modelo operacional da função *powf*.

4.3.9 Chamada de Kernel

A abordagem seguida no projeto do ESBMC-GPU converte as *threads* de programas CUDA para programas ANSI-C, devido o ESBMC já possuir suporte à biblioteca *pthread POSIX*. Portanto quando o programa CUDA faz a chamada de *kernel*, o verificador inicia *threads* da biblioteca *pthreads* para simulá-las no CUDA.

No código CUDA como visto na Seção 2.1.3 existe a chamada de *kernel*. Quando a execução do *kernel* é iniciada são disparadas as *threads*. No CUDA é usado um identificador $<<<,>>>$ para especificar a chamada de *threads*. No modelo operacional foi implementada a função *ESBMC_ verify_ kernel()* que faz a chamada de *kernel*.

A função *ESBMC_ verify_ kernel()* é um *template* com os mesmos parâmetros da chamada de *kernel* padrão como apresenta a Figura 4.38. O parâmetro *kernel* é um ponteiro para função, *blocks* indica o número de blocos, *threads* indica o número de *threads* e *arg* contém o argumento passado na chamada de *kernel*.

O *template* apresentado na Figura 4.38, mostra nas linhas 8 e 9 as variáveis *gridDim* e *blockDim* sendo instanciadas pelo valor do bloco e *threads* por bloco que foram passados por parâmetro. As variáveis *gridDim* e *blockDim* foram mostradas na Seção 4.3.5. A linha 11 chama a função *ESBMC_ verify_ kernel_ one_ arg()* onde serão criadas *threads*, o *gridDim* e *blockDim* tem suas dimensões decompostas e multiplicadas para saber a quantidade total de blocos e *threads* por blocos que irão ser iniciadas.

```
1  template<class RET, class BLOCK, class THREAD, class T1>
2  void ESBMC_verify_kernel(RET *kernel,
3                           BLOCK blocks,
4                           THREAD threads,
5                           T1 arg)
6  {
7
8     gridDim = dim3(blocks);
9     blockDim = dim3(threads);
10
11    ESBMC_verify_kernel_one_arg(
12                          (voidFunction_one) kernel,
13                          gridDim.x * gridDim.y * gridDim.z,
14                          blockDim.x * blockDim.y * blockDim.z,
15                          (void*) arg);
16
17    int i = 0;
18    for (i = 0; i < GPU_threads; i++)
19       pthread_join(threads_id[i], NULL);
20 }
```

Figura 4.38: Template *ESBMC_ verify_ kernel()*.

A Figura 4.39 apresenta a função *ESBMC_ verify_ kernel_ one_ arg()*, onde são iniciadas as *threads*. A variável *threads_ id* é declarada na linha 13, e tem memória alocada para todas as *threads* na linha 21.

A variável *dev_ one* é global do tipo *Targ1*, uma *struct arg_ struct1* como mostra entre as linhas 1 e 6 da Figura 4.39. A *dev_ one* é instanciada com o argumento *arg1* e o ponteiro para função *kerner* nas linhas 24,25 e 26.

A função *assignIndexes()* determina os índices para as *threads*, sua implementação é apresentada na Seção4.3.5.

O *loop while* entre as linhas 31 e 40 inicia as *threads*, é passado o endereço de cada elemento do vetor de *threads* e a função *ESBMC_ execute_ kernel_ one()* que é descrita entre as linhas 8 e 11, e possui o *dev_ one*.

Após iniciar as *threads* com *ESBMC_ execute_ kernel_ one()* e o término da *ESBMC_ verify_ kernel_ one_ arg()*, na Figura 4.38 nas linhas 18 e 19, um *loop for* é usado para executar a *pthread_ join()* sobre todas as *threads* iniciadas, fazendo assim esperá-las executar por completo.

```
1  struct arg_struct1 {
2      int *a;
3      void* (*func)(int*);
4  };
5  typedef struct arg_struct1 Targ1;
6  Targ1 dev_one;
7
8  void *ESBMC_execute_kernel_one(void *args){
9      dev_one.func(dev_one.a);
10     return NULL;
11 }
12
13 pthread_t *threads_id;
14
15 void ESBMC_verify_kernel_one_arg(void *(*kernel)(int*),
16                                  int blocks,
17                                  int threads,
18                                  void* arg1)
19 {
20     __ESBMC_atomic_begin();
21     threads_id = (pthread_t *) malloc(GPU_threads *
22                                                 sizeof(pthread_t));
23
24     dev_one.a = (int*) malloc(GPU_threads * sizeof(int));
25     dev_one.a = (int*) arg1;
26     dev_one.func = kernel;
27
28     int i = 0, tmp;
29     assignIndexes();
30
31     while (i < GPU_threads)
32     {
33         pthread_create(&threads_id[i],
34                        NULL,
35                        ESBMC_execute_kernel_one,
36                        NULL);
37         i++;
38     }
39     __ESBMC_atomic_end();
40 }
```

Figura 4.39: Função *ESBMC_ verify_ kernel_ one_ arg()* e iniciação das *threads*.

As figuras 4.38 e 4.39 apresenta como é feito em uma chamada de *kernel* que possui um parâmetro, o código acaba sendo replicado para ter suporte para *kernels* que possua mais parâmetros.

Essa abordagem foi seguida pois existem *kernels* com diferentes tipos de dados e quantidades de parâmetros. A utilização de *templates* permite a utilização de múltiplos tipos de parâmetros. O uso de ponteiro para função permite iniciar uma *thread* usando o *pthread_ create* sem precisar definir *structs* com os campos específicos.

As funções desta seção estão descritas no arquivo *call_ kernel.h*.

Capítulo 5

Resultados e Discussão

Este capítulo apresenta os resultados de desempenho do modelo operacional proposto e análises dos resultados. Na Seção 5.1 são apresentados os experimentos realizados com MOC aplicado à ferramenta ESBMC-GPU. A Seção 5.2 apresenta uma comparação entre a ferramenta gerada com o MOC criado e outros verificadores. A Seção 5.3 apresenta os resultados gerais, análise do MOC e a avaliação dos resultados.

5.1 Objetivo dos Experimentos

Os experimentos realizados visam estimar a habilidade da ferramenta que faz uso do MOC, o ESBMC-GPU, e além disso, identificar os *benchmarks* e respectivas limitações da ferramenta unido ao uso do MOC.

5.1.1 Configuração dos Experimentos e Benchmarks

Os *benchmarks* foram criados para avaliar a ferramenta ESBMC-GPU, e consequentemente o modelo operacional. Dentre as dificuldades em criar os *benchmarks* foi garantir que os códigos correspondam seu propósito, se o código com erro contém o erro para qual foi idealizado, e se código destinado para ser correto não possue falhas.

Todos os casos de teste foram analisados para identificar seu objetivo, e confrontar com o resultado de sua execução. Cada caso de teste foi executado pela IDE *nsight* [33]. Alguns casos apresentavam erros de programação como instanciação incorreta de variáveis. Este processo de ajustes no *benchmark* e teste na IDE foram realizados até que a IDE pudesse compilar e executar o programa com o objetivo correto do código.

Importante ressaltar aqui, que mesmo a IDE não detectando nenhum erro ou *warnings* em um código CUDA, que aparentemente está livre de qualquer problema, na verdade o código pode apresentar falhas como corrida de dados, *overflow* e outras propriedades, que só serão detectados pelo verificador.

Foram usados neste processo de validação dos *benchmark* laptops DELL Latitude com Intel® Core i7, 8GB de RAM, Placa de vídeo GeForce GT 740M e SO Ubuntu 14.04.

5.1.2 Modelo Operacional

Com os *benchmarks* constituídos foi possível testar o ESBMC-GPU e assim avaliar o MOC. Um *makefile* foi usado para verificar com o ESBMC-GPU o conjunto de *benchmarks* de forma automática. A ferramenta criada herdou as características do ESBMC, assim, ao verificar um programa, o mesmo manteve o comportamento de sua origem, isso inclui os contraexemplos das propriedades verificadas e indicadores dos locais das funções quando ocorre algum erro. Vale ressaltar que nenhum processo a mais é necessário ser executado, somenteo ESBMC-GPU.

Para fazer os testes dos modelos operacionais (ESBMC-GPU) foi usado um computador com 16GB de RAM, processador Intel® Core i7 e SO Fedora 20.

5.2 Comparativo com outras ferramentas

Os *benchmarks* foram aplicados em outras três ferramentas, GPUVerify, PUG e GKLEE. No caso do GPUVerify e PUG, como ambas só suportam verificar o *kernel*, foi removida a função *main()* dos *benchmarks* e usado somente o *kernel* na verificação, com o GKLEE foi usado todo o código para a verificação.

A Tabela 5.1 extraída de [24], apresenta um comparativo entre a ferramenta

criada que usa o modelo operacional e outros três verificadores, cada linha da tabela significa: (1) nome da ferramenta (Ferramenta); (2) Número total de *benchmarks* que foram verificados corretamente (Resultados Corretos); (3) Número total de *benchmarks* cuja verificação retornou um erro falso (Falsos Negativos); (4) Número total de *benchmarks* cujo erro não foi detectado pela ferramenta (Falsos Positivos); (5) Número total de *benchmarks* não suportados pela ferramenta (Não Suportado); (6) Número total de *benchmarks* cuja verificação não foi bem sucedida por exceder o tempo máximo de verificação estabelecido (4800 segundos) (Timeout); (7) Tempo de execução, em segundos, da verificação para todos os *benchmarks* [24].

Tabela 5.1: Comparativo de ferramentas.

Ferramenta	**ESBMC-GPU**	**GPUVerify**	**PUG**	**GKLEE**
Resultados Corretos	113	94	53	114
Falsos Negativos	3	2	14	11
Falsos Positivos	4	5	5	11
Não Suportado	37	59	88	23
Timeout	3	0	0	1
Tempo	15 912s	160s	10s	760s

O ESBMC-GPU obteve 70,6% dos resultados corretos enquanto que o GPUVerify teve 58,7%, PUG 33,1% e GKLEE 71,2%. Os resultados do ESBMC-GPU se aproximam aos do GKLEE, mas como o ESBMC-GPU teve menos resultados falsos negativos e falsos positivos que o GKLEE, a ferramenta criada que faz uso do modelo operacional apresentado neste trabalho é mais confiável que as demais. O ESBMC-GPU obteve menor quantidades de *benchmarks* não suportados em relação ao GPUVerify e PUG. Outro fator interessante é o *Timeout*, como apenas 3 casos não foram verificados no tempo limite pelo ESBMC-GPU, implica dizer que a representação do MOC está simples o bastante a ponto de não estourar o tempo de verificação, ao mesmo tempo que obteve bons resultados, quando comparado com ouras ferramentas.

5.3 Resultados Gerais

Dentre os casos de testes que a ferramenta apresentou falso negativo (3), 1 caso está relacionado ao modelo operacional, devido o MOC da função *cudaMalloc()* apresentar problemas com o tipo *float*.

A ferramenta não conseguiu verificar 4 casos devido à representação da memória constante, que é feita de forma à validar apenas o *parser* do código, outros 3 casos não suportados, possuem funções a qual seus modelos operacionais não foram implementados, como *mul24()* e *threadfence()*. Funções das biblioteca *curand.h* e *mathfunctions.h* tiveram problemas em alguns modelos operacionais, somando mais 9 *benchmarks* não suportados pela ferramenta.

No total 17 *benchmarks* apresentaram problemas durante a verificação por falhas no modelo operacional criado.

Capítulo 6

Conclusões

Nesta monografia, foi apresentado o desenvolvimento da maior parte do modelo operacional do CUDA, que é usado para criar a ferramenta ESBMC-GPU, capaz de verificar códigos escritos em CUDA. Este trabalho constituiu: a formação de um conjunto de *benchmarks* para avaliar a ferramenta ESBMC-GPU e o modelo operacional CUDA; a implementação dos modelos das características CUDA não suportadas pelo ESBMC; além de testes e avaliações dos modelos aplicados à ferramenta criada.

A elaboração do conjunto de *benchmarks* teve grande importância no projeto, primeiro porque permitiu observar quais funções e características CUDA deveriam ter prioridade na implementação do modelo, segundo porque é essencial para avaliar a ferramenta e do MOC, saber se a ferramenta criada consegue encontrar os problemas para qual foi projetada.

Para constituir o conjunto de *benchmarks*, houve grande esforço pois não existem suítes de testes consolidadas voltadas para verificação em CUDA, como em ANSI-C por exemplo. Anualmente acontece uma competição de verificação de códigos escritos em C chamada SV-COMP [23], várias ferramentas são submetidas à testes em uma base de dados com *benchmarks* escrito em ANSI-C. Assim como no SV-COMP com o ANSI-C, aqui foi formado um conjunto de *benchmarks* para avaliar ferramentas de verificação CUDA.

O conjunto de *benchmarks* possui 85% dos casos absorvidos da base de testes do GPUVerify, outros 15% foram extraídos de livros [13], [34], do guia de programação CUDA ou criados [1]. O GPUVerify verifica apenas os *kernels* de um código CUDA, por isso todos os códigos obtidos do GPUVerify tiveram que sofrer um processo de reconstituição de código como visto na Seção 4.1.1. A maior preocupação foi em não alterar o objetivo do código original, teve-se que compreender cada código para que durante a criação da função *main()* e demais linhas de código, o objetivo do programa não fosse mudado.

A abordagem do modelo operacional com a ferramenta ESBMC obteve sucesso para criação de ferramentas de verificação em C++ e o *framework* Qt. O modelo operacional CUDA (MOC) teve grande importância para a criação do ESBMC-GPU, tendo resultados satisfatório também. Grande destaque da ferramenta está em verificar todo o código CUDA, e não somente os *kernels* como o GPUVerify e PUG. Isso foi facilitado devido ao ESBMC verificar códigos C e C++, sendo necessário apenas implementar as características específicas do CUDA.

Inicialmente, foram feitas as representações dos qualificadores, *structs*, *enums*, tipos de vetor e outras estruturas mais simples pertencentes ao CUDA e que não tinham no ESBMC; em seguida foram implementados os modelos das funções que fazem uso dessas estruturas citadas anteriormente.

Vale ressaltar que estruturas e funções CUDA foram encontradas na literatura, mas não existiam nos *benchmarks*, por isso não tiveram seus modelos operacionais implementados, uma vez que não seria possivel testá-los, e o objetivo era superar as ferramentas existentes, conforme o conjunto de *benchmarks* construído.

Os problemas encontrados com o modelo se deu pela falta de uma representação mais categórica para diferenciar *device* e *host*, algumas estruturas que estão apenas com *define* para *void*, apenas para que durante a verificação não haja problema com *parser*, algumas funções não foram implementadas como *mul24()*, outras presisam de melhorias como as funções randômicas.

A criação do MOC permitiu um aprofundamento na programação em CUDA, pois era preciso o entendimento de cada função para a sua modelagem. Como a arquitetura da GPU é diferente da CPU, também foi preciso assimilar a arquitetura GPU CUDA. A linguagem CUDA usada é uma extensão do C, portanto a pro-

gramação em C também precisou ser revisada. Alguns modelos como *cudaFree()*, *atomicAdd()* e funções matemáticas tiveram em sua implementação o uso de parte do código do ESBMC. Por isso foi necessário estudar a estrutura e organização do projeto ESBMC. Durante o trabalho foi aprendido conhecimentos sobre controle de versionamento com uso da ferramenta GIT e metodologias ágeis como SCRUM.

6.1 Propostas para Trabalhos Futuros

Nesta seção, são apresentadas algumas propostas para desenvolvimentos futuros relacionados ao modelo operacional descrito nesta monografia. Estas propostas estão divididas em duas categorias. Na primeira, é relacionado ao modelo operacional CUDA, na segunda está relacionado à extensão do modelo para outra plataforma GPU.

Melhorias do MOC e suíte de testes:

- Inserir novos *benchmarks* na suíte de testes para o CUDA, criar mais casos que tenha as falhas que o verificador suporta, códigos com corrida de dados, *array bouds*, *null-pointer*.
- Implementar mais modelos, como funções matemáticas, funções de sincronização e funções relacionados à gráficos e vídeos.
- Inserir ao modelo uma representação que diferencie regiões de *host* e *device*.

Implementação do modelo operacional OpenCL:

- OpenCL é a plataforma concorrente do CUDA em processamento de GPUs. Possui uma estrutura parecida com a do CUDA, pois foi idealizada a partir do CUDA. Implementar o modelo para o OpenCL dará mais abrangência para o ESBMC-GPU em verificação de códigos aplicados a GPUs.
- Criar uma suíte de testes para o OpenCL para testar o modelo operacional OpenCL.

Referências Bibliográficas

[1] NVIDIA. *CUDA C Programming Guide*. v7.0. NVIDIA Corporation, 2015. Disponível em: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>.

[2] MISIC, M.; DURDEVIC, D.; TOMASEVIC, M. Evolution and trends in gpu computing. In: IEEE. *MIPRO, 2012 Proceedings of the 35th International Convention*. [S.l.], 2012. p. 289 – 294.

[3] JANUÁRIO, F. et al. BMCLua: Verification of Lua programs in digital TV interactive applications. In: *Consumer Electronics (GCCE), 2014 IEEE 3rd Global Conference on*. [S.l.: s.n.], 2014. p. 707 – 708.

[4] CLARKE, E. M.; GRUMBERG, O.; PELED, D. A. *Model Checking*. [S.l.]: The MIT Press, 1999. ISBN 978-0-262-03270-4.

[5] PATIL, S.; VYATKIN, V.; PANG, C. Counterexample-guided simulation framework for formal verification of flexible automation systems. In: IEEE. *Industrial Informatics (INDIN), 2015 IEEE 13th International Conference on*. [S.l.], 2015. p. 1192 – 1197.

[6] HARTONAS-GARMHAUSEN, V.; CLARKE, E. M.; CAMPOS, S. Deadlock prevention in flexible manufacturing systems using symbolic model checking. In: IEEE. *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*. [S.l.], 1996. p. 527 – 532 vol.1.

[7] WANG, Y.; WU, Z. Deadlock avoidance control synthesis in manufacturing systems using model checking. In: IEEE. *American Control Conference, 2003. Proceedings of the 2003*. [S.l.], 2003. p. 1702 – 1704.

[8] FISCHER, B.; INVERSO, O.; PARLATO, G. CSeq: A Concurrency Preprocessor for Sequential C Verification Tools. In: *ASE Automated Software Engineering, IEEE/ACM 28th International Conference*. [S.l.: s.n.], 2013. p. 710 – 713.

[9] ZHENG, M. et al. CIVL: Formal Verification of Parallel Programs. In: *IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2015. p. 830 – 835.

[10] RAMALHO, M. et al. Smt-based bounded model checking of c++ programs. In: *Proceedings of the 20th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems*. Washington, DC, USA: IEEE Computer Society, 2013. (ECBS '13), p. 147–156. ISBN 978-0-7695-4991-0. Disponível em: <http://dx.doi.org/10.1109/ECBS.2013.15>.

[11] NVIDIA. *CUDA Home new*. v7.0. NVIDIA Corporation, 2015. Disponível em: <http://www.nvidia.com/object/cuda_home_new.html>.

[12] SCOTT, S. The evolution of gpu accelerated computing. In: IEEE. *High Performance Computing, Networking, Storage and Analysis (SCC)*. [S.l.], 2012. p. 1636 – 1672.

[13] CHENG, J.; GROSSMAN, M.; MCKERCHER, T. *Professional CUDA C Programming*. [S.l.]: WROX, 2012. ISBN 987-1-118-73932-7.

[14] BAIER, C.; KATOEN, J.-P. *Principles of Model Checking*. [S.l.]: The MIT Press, 2008. ISBN 978-0-262-02649-9.

[15] LEVESON, N.; TURNER, C. An investigation of the therac-25 accidents. In: IEEE. *IEEE Computer Society*. [S.l.], 2002. p. 18 – 41.

[16] CORDEIRO, L.; FISCHER, B. Verifying multi-threaded software using smt-based context-bounded model checking. In: . Washington, DC, USA: IEEE Computer Society, 2011. p. 137–148. ISBN 978-0-7695-3891-4.

[17] CORDEIRO, L.; FISCHER, B.; MARQUES-SILVA, J. Smt-based bounded model checking for embedded ansi-c software. In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009. (ASE '09), p. 137–148. ISBN 978-0-7695-3891-4.

[18] ROCHA, H. et al. Understanding programming bugs in ansi-c software using bounded model checking counter-examples. In: *iFM 9th International Conference on Integrated Formal Methods*. [S.l.: s.n.], 2012. p. 128 – 142.

[19] MOURA, L. D.; BJØRNER, N. Z3: An Efficient SMT Solver. In: *TACAS*. [S.l.: s.n.], 2008. (LNCS, v. 4963), p. 337–340.

[20] BRUMMAYER, R.; BIERE, A. Boolector: An efficient smt solver for bit-vectors and arrays. In: *TACAS*. [S.l.: s.n.], 2009. (LNCS, v. 5505), p. 174 – 177.

[21] MORSE, J. et al. Handling Unbounded Loops with ESBMC 1.20 - (Competition Contribution). In: *TACAS*. [S.l.: s.n.], 2013. (LNCS, v. 7795), p. 619–622.

[22] MORSE, J. et al. ESBMC 1.22 (Competition Contribution). In: *TACAS*. [S.l.: s.n.], 2014. (LNCS, v. 8413), p. 405–407.

[23] SV-COMP. (TACAS) International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2016. Disponível em: <http://sv-comp.sosy-lab.org/2016/>.

[24] PEREIRA, P. et al. Verificação de Kernels em Programas CUDA usando Bounded Model Checking. In: *WSCAD XV Simpósio de Sistemas Computacionais de Alto Desempenho*. [S.l.: s.n.], 2015. p. 24 – 25.

[25] SOUSA, F.; CORDEIRO, L.; LIMA, E. Bounded Model Checking of C++ Programs Based on the Qt Framework. In: *GCCE IEEE 4th Global Conference on Consumer Electronics*. [S.l.: s.n.], 2015. p. 446 – 447.

[26] CLARKE, E. M.; KROENING, D.; LERDA, F. A tool for checking ANSI-C programs. In: *TACAS*. [S.l.: s.n.], 2004. p. 168 – 176.

[27] BARNAT, J. et al. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In: *Computer Aided Verification (CAV 2013)*. [S.l.]: Springer, 2013. (LNCS, v. 8044), p. 863–868.

[28] FALKE, S.; MERZ, F.; SINZ, C. The bounded model checker LLBMC. In: *ASE*. [s.n.], 2013. p. 706–709. Disponível em: <http://dx.doi.org/10.1109/ASE.2013.6693138>.

[29] SOUSA, F.; CORDEIRO, L.; LIMA, E. Verificação de Programas C++ Baseados no Framework Multiplataforma Qt. In: *ENCOSIS IV Encontro Reginal de Computação e Sistemas de Informação*. [S.l.: s.n.], 2015. p. 181 – 190.

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

[31] BETTS, A. et al. Gpuverify: a verifier for GPU kernels. In: *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. [s.n.], 2012. p. 113–132. Disponível em: <http://doi.acm.org/10.1145/2384616.2384625>.

[32] LI, G. et al. Gklee: Concolic verification and test generation for gpus. In: *PPoPP*. [S.l.]: ACM, 2012. p. 215–224.

[33] NVIDIA. *Nsight Eclipse Edition Getting Started Guide*. 2015. Disponível em: <http://docs.nvidia.com/cuda/nsight-eclipse-edition-getting-started-guide/>.

[34] COOK, S. *CUDA Programming, A developer's guide to parallel computing with GPUs*. [S.l.]: Morgan Kaufmann, 2013. ISBN 987-0-12-415933-4.

Apêndice A

Publicações

Referente ao trabalho

- Pereira, P. A., **Albuquerque, H. F.**, Marques, H. M., Silva, I. S., Santos, V. S., Carvalho, C. B., Cordeiro, L. C. Verifying CUDA Programs using SMT-Based Context-Bounded Model Checking. ACM Symposium on Applied Computing (SAC), 2016.

- Pereira, P. A., **Albuquerque, H. F.**, Marques, H. M., Silva, I. S., Santos, V. S., Ferreira, R. S., Carvalho, C. B., Cordeiro, L. C. Verificação de Kernels em Programas CUDA usando Bounded Model Checking. XVI Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD), 2015.