Verificação de *Kernels* em Programas CUDA usando *Bounded Model Checking*

Phillipe A. Pereira[1], Higo F. Albuquerque[1], Hendrio M. Marques[1], Isabela S. Silva[1], Vanessa de S. Santos[1], Ricardo dos S. Ferreira[2] Celso B. Carvalho[1], Lucas C. Cordeiro[1]

[1]Universidade Federal do Amazonas (UFAM) – Manaus – AM – Brasil

[2]Universidade Federal de Viçosa (UFV) – Viçosa – MG – Brasil

{apphillipe,albuquerque.higo,hendriomm,isabelassilva}@gmail.com

{svs.vanessasantos,cacauvicosa,celsocarvalho75,lucasccordeiro}@gmail.com

Abstract. *This paper presents an extension to the Efficient SMT-Based Context-Bounded Model Checker (ESBMC) for verifying Graphics Processing Unit (GPU) programs, called ESBMC-GPU. In particular, ESBMC-GPU is a Context-Bounded Model Checker based on the Satisfiability Modulo Theories for programs written in Compute Unified Device Architecture (CUDA). It is based on an operational model, an abstract representation of the standard CUDA libraries that conservatively approximates their semantics. With ESBMC-GPU, it is possible to verify more realistic CUDA programs than other existing approaches.*

Resumo. *Este artigo apresenta uma extensão da ferramenta Efficient SMT-Based Context-Bounded Model Checker (ESBMC) para verificar programas que executam em unidades de processamento gráfico (GPU), chamado de ESBMC-GPU. Em especial, ESBMC-GPU é um verificador de modelos limitado baseado nas teorias do módulo da satisfatibilidade para programas desenvolvidos na arquitetura de dispositivo unificado de computação (CUDA). O ESBMC-GPU é baseado em um modelo operacional, uma representação abstrata das bibliotecas padrões do CUDA que conservadoramente aproxima suas semânticas. Com ESBMC-GPU, é possível verificar mais programas CUDA reais do que outras abordagens existentes.*

1. Introdução

A arquitetura de dispositivo unificado de computação (*Compute Unified Device Architecture*, CUDA) é uma plataforma de computação desenvolvida pela empresa NVIDIA [Cheng et al. 2014] que estende as linguagens C/C++/Fortran, criando um modelo computacional que objetiva explorar a capacidade computacional de unidades de processamento gráfico (*Graphical Processing Units*, GPUs) [Kirk and Hwu 2010].

Em linguagens de programação, erros ocorrem com frequência, tais como bloqueio fatal, estouro aritmético e divisão por zero. No caso de CUDA, por ser uma plataforma que lida com programação paralela, também apresenta frequentemente erros de condições de corrida, barreira de divergência e compartilhamento de memória [Betts et al. 2012].

A fim de detectar estes e outros possíveis erros, uma técnica em especial é utilizada para explorar sistematicamente o espaço de estados de um programa, a verificação por métodos formais [Baier and Katoen 2008]. Os métodos formais utilizam modelos matemáticos para realizar a análise de sistemas complexos, proporcionando uma verificação mais eficiente e consequentemente reduzindo o tempo de validação de programas. Apresentam como desvantagem explosão do espaço de estados que ocorre quando

o verificador tem que considerar múltiplas intercalações entre as *threads* e um conjunto muito grande de variáveis de dados.

Estes modelos baseiam-se em teorias matemáticas, por exemplo, a Satisfatibilidade Booleana (*Boolean Satisfiability*, SAT) ou Teorias do Módulo de Satisfatibilidade (*Satisfiability Modulo Theories*, SMT). Para contornar o problema da explosão de estados, utiliza-se a técnica de verificação de modelos limitada (*Bounded Model Checking*, BMC), que restringe o número de estados verificando a negação de propriedades até uma determinada profundidade [Biere et al. 1999].

O *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC) é uma das principais ferramentas utilizadas para verificação formal de programas escritos em C/C++ [Cordeiro et al. 2012, Ramalho et al. 2013]. Sua arquitetura é baseada nas técnicas BMC e SMT, sendo uma das ferramentas BMC mais eficientes até o momento, com destaques nas últimas competições de verificação de software [Morse et al. 2013, Morse et al. 2014]. Estender os benefícios da verificação de modelos baseado em SMT para programas escritos em CUDA e introduzir este método de verificação em programação paralela para processadores gráficos, é a principal contribuição deste artigo. Desta forma, torna-se possível detectar as falhas antes da execução do programa.

Utilizando modelos operacionais que simulam as bibliotecas do CUDA, em conjunto com otimizações implementadas no ESBMC e a chamada de *kernel*, desenvolvida especialmente para adaptar a sintaxe dos programas CUDA ao ESBMC, foi possível atingir resultados significativos na verificação de *kernels*, principalmente quando comparado a outros verificadores, sobre programas próximos dos casos reais (programa principal e *kernel*). Foi levado em conta aspectos como alocação dinâmica de memória, transferência de dados, descarte e estouro de memória (realizados necessariamente no programa principal), validade dos dados passados pelo programa principal ao *kernel*, além de proporcionar uma verificação mais precisa, com a desvantagem da explosão do espaço de estados gerada por verificar-se um número elevado de intercalações.

2. Preliminares

Esta seção descreve conceitos básicos sobre o funcionamento do ESBMC, que aplica a técnica de BMC baseado em SMT para verificar programas C/C++, e da plataforma CUDA, usada para programação de GPU.

2.1. Verificação de Modelos Limitada Baseada em SMT

A técnica BMC possui a ideia básica de checar a negação de uma propriedade até uma determinada profundidade. Dado um sistema de transição de estados M, uma propriedade ϕ, e um limite k, BMC desdobra o sistema k vezes e o traduz para uma condição de verificação (*verification condition*, VC) ψ, tal que ψ é satisfatível, se e somente se, ϕ possuir um contraexemplo de profundidade k ou menor [Cordeiro et al. 2012].

SMT resolve as fórmulas de satisfatibilidade de primeira ordem usando a combinação de diferentes teorias e assim generaliza a satisfatibilidade proposicional por suportar funções não interpretadas, aritmética linear e não linear, vetores de *bit*, tuplas, *arrays*, e outras teorias de primeira ordem.

Dada uma teoria τ e uma fórmula livre de quantificadores ψ, é dito que ψ é τ-satisfatível se e somente se existe uma estrutura que satisfaz a fórmula e a sentença de τ, ou equivalentemente, se $\tau \bigcup \{\psi\}$ é satisfatível [Bradley and Manna 2007]. Dado um conjunto $\Gamma \bigcup \{\psi\}$ de fórmula sobre τ, é dito que ψ é uma τ-consequência de Γ, e escrevemos $\Gamma \models_\tau \psi$, se e somente se, todo modelo de $\tau \bigcup \Gamma$ é também um modelo de ψ. Verificando $\Gamma \models_\tau \psi$ pode ser reduzido de maneira usual para checar a τ-satisfatibilidade de $\Gamma \bigcup \{\neg\psi\}$ [Barrett et al. 2009].

2.2. *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC)

A ferramenta ESBMC verifica modelos de contexto limitado usando solucionadores baseados em SMT. O ESBMC trabalha sobre uma arquitetura, onde o programa a ser verificado é estruturado a partir de um *parser* desenvolvido em flex/bison e convertido para o formato de uma árvore de sintaxe abstrata (*Abstract Syntax Tree*, AST). Neste ponto é feita uma checagem de tipos e logo em seguida, o programa é convertido para uma linguagem intermediária chamada de representação intermediária (*Intermediate Representation*, IRep), onde é adicionada a checagem sobre atribuições, *typecast*, inicialização de ponteiros, chamadas de funções, criação de *templates* e instanciações.

No próximo passo, a árvore IRep é convertida para expressões do tipo *goto*, que simplifica a representação de um programa convertendo instruções *switch* e *while* por *if* e *goto*, além de auxiliar no desdobramento dos *loops* e funções recursivas. Na execução simbólica do programa *goto*, o programa é convertido para expressões de atribuição estática única (*Single Static Assignment*, SSA) e assertivas são inseridas para verificação de propriedades de segurança. Após estes passos, dois conjuntos de fórmulas livres de quantificadores são criados baseados na SSA: $\mathcal{C}$ para representar o conjunto de restrições e $\mathcal{P}$ para representar o conjunto de propriedades [Cordeiro et al. 2012].

As duas funções recursivas $\mathcal{C}$ e $\mathcal{P}$ computam as restrições (*i.e.*, condicionais e atribuições de variáveis) e propriedades (*i.e.*, condições de segurança e assertivas definidas pelo usuário), respectivamente. Assim, é gerado automaticamente as VCs que checam, por exemplo, estouro aritmético, violação do limite de vetores e vazamento de memória. Ambas as funções acumulam o predicado do fluxo de controle de cada ponto do programa e usa estes predicados para guardar as restrições e propriedades, tal que elas propriamente reflitam a semântica do programa [Cordeiro et al. 2012].

Após este passo, as VCs são fornecidas para um solucionador SMT para serem analisadas, e se alguma violação da propriedade é detectada, o ESBMC fornece um contraexemplo indicando o caminho do erro ocorrido. Caso contrário, o ESBMC informa que o erro não foi encontrado para o determinado limite k, e a verificação retorna sucesso.

2.3. Plataforma CUDA

CUDA é uma plataforma de computação paralela de propósito geral que representa um modelo de programação desenvolvido pela empresa NVIDIA para executar em GPUs fabricadas pela mesma. É uma linguagem desenvolvida para ter uma rápida curva de aprendizado, cujo o ambiente pode ser facilmente utilizado por programadores das linguagens C/C++ e Fortran [NVIDIA 2015].

No modelo de programação de CUDA, o conceito de *kernel* é usado para uma função que executa n cópias paralelamente na GPU, onde n é o produto do número de blocos e *threads*. Um *kernel* é definido por um especificador `__global__` e sua chamada no programa é feita pela notação `kernel<<< B,T >>>` onde B é o número de blocos e T é o número de *threads*. Cada *kernel* é referenciado na GPU como *thread* e cada *thread* recebe um identificador único (ID) formado pelo número da *thread* e o número do bloco. O ID da *thread* é usado para indexar as suas tarefas (*i.e.*, posições de memória e cooperação). As *threads* são organizadas em blocos.

Dentro de um bloco, a hierarquia de *threads* é definida pela variável chamada *threadIdx*. Esta variável é um vetor de três componentes permitindo usar índices uni-, bi- e tridimensionais. Por exemplo, para obter o *ID* de uma *thread* nestas configurações para um bloco bidimensional de tamanho (D_x, D_y), o *ID* da *thread* de índice (x, y) é obtido por $(x + yD_x)$, e para um bloco tridimensional de tamanho (D_x, D_y, D_z), a *ID* de uma *thread* de índice (x, y, z) é definido por $(x + yD_x + zD_xD_y)$ [NVIDIA 2015].

Os blocos também podem ser definidos em três dimensões, onde cada dimensão

pode ser acessada pela variável *blockIdx*. Esta variável também é formada por três componentes permitindo usar blocos uni-, bi- e tridimensionais. O número máximo de *threads* por bloco depende da geração da placa variando de 1024 a 2048 [NVIDIA 2015]. Os blocos possuem a característica de serem executados em qualquer ordem, podendo ser alocados a qualquer processador. Com isso, um *kernel* pode ser executado por múltiplos blocos de forma igual, e o número total de *threads* é o número de blocos vezes o número de *threads* por bloco.

Um conceito utilizado em CUDA, é fazer referência à GPU como *device* e a unidade de processamento central (CPU) como *host*. $\_\_device\_\_$ é um especificador para funções que executam e são chamadas somente pela GPU, e $\_\_host\_\_$ para funções que executam e são chamadas somente pela CPU. O fluxo comum da alocação de dados para o *device* é feito no *host* usando as funções *cudaMalloc*, *cudaFree* e *cudaMemcpy*. Estas são funções essenciais para um programa CUDA, pois os dados são transferidos do *host* para o *device* e vice-versa. A função *cudaMalloc* aloca uma quantidade de memória no *device*, a qual é liberada após sua utilização pela função *cudaFree*. A função *cudaMemcpy* é usada para copiar dados inicializados no *host* para o *device*. Suas interfaces são baseadas nas funções *malloc, free* e *memcpy* da linguagem C.

3. Verificação de *Kernels* em Programas CUDA usando BMC

Esta seção está dividida em subseções nas quais são apresentados: o algoritmo empregado para criar, executar e verificar *threads* em GPUs considerando a execução intercalada; os modelos operacionais desenvolvidos para simular o funcionamento das bibliotecas do CUDA; por fim, apresenta-se um exemplo de verificação de programa CUDA, a partir da utilização do algoritmo e ferramenta propostos.

3.1. Verificação Baseada em Exploração Preguiçosa

O algoritmo da nossa proposta de verificação de modelos para programas CUDA tem como base exploração preguiçosa, que no lugar de verificar de uma única vez todas as possíveis intercalações de um programa, verifica de forma incremental uma intercalação por vez [Cordeiro and Fischer 2011]. O algoritmo percorre, em profundidade, a árvore de alcançabilidade de estados (*Reachability Tree*, RT). Ao chegar em um nó folha, o algoritmo passa para o solucionador uma fórmula SMT que representa uma intercalação específica do programa CUDA. Caso esta fórmula seja satisfatível, uma violação da propriedade no programa CUDA foi encontrada e, assim, um contraexemplo é fornecido. Caso contrário, é realizado um *backtrack* na RT e uma nova intercalação é produzida e verificada pelo solucionador SMT. O algoritmo termina quando ou uma violação é encontrada pelo ESBMC ou todas as intercalações foram verificadas com sucesso.

No modelo de verificação, utiliza-se uma árvore RT dada por $\Upsilon = \{v_1, ..., v_n\}$, onde $\{v_1, ..., v_n\}$ representam nodos (ou instruções) do programa. Desta forma, Υ representa o desdobramento de um programa para um limite de contexto C, um limite de profundidade *k* dos *loops* e uma propriedade ϕ. Neste contexto, deriva-se uma VC, representada por ψ_π^k, para uma intercalação $\pi = \{v_1 ... v_k\}$, tal que ψ_π^k é satisfeita se, e somente se, a propriedade ϕ possuir um contraexemplo de profundidade *k* que pode ser gerado pela intercalação π. A VC ψ_π^k é uma fórmula livre de quantificadores em um subconjunto de lógica de primeira ordem, que é verificada quanto à satisfatibilidade por um solucionador SMT. O problema de verificação associado com a técnica BMC baseada em SMT para programas CUDA pode ser formulada a partir da Equação 1 [Cordeiro and Fischer 2011].

$$\psi_k^\pi = \overbrace{I(s_0) \wedge R(s_0, s_1) \wedge \ldots \wedge R(s_{k-1}, s_k)}^{Restrições} \wedge \overbrace{\neg\phi_k}^{Propriedade} \quad (1)$$

Na Equação 1, ϕ_k representa a propriedade de segurança (por exemplo, condição de corrida) no passo k, I se refere a escolha do estado inicial e $R(S_i, S_{i+1})$ é a relação de

transição entre os passos i e $i+1$, como descrito pelos estados nos nodos de π. A fim de verificar se a Equação 1 é ou não satisfatível, o solucionador SMT reduz os símbolos para uma dada teoria de fundamentação [De Moura and Bjørner 2008]. Caso a Equação 1 seja satisfatível, então a propriedade ϕ é violada e o solucionador SMT provê uma atribuição satisfatória, onde é possível extrair os valores de cada variável do programa, construir um contraexemplo e determinar a sequência de estados $s_0, s_1, ..., s_k$, sendo $s_0 \in S_0$ e $R(S_i, S_{i+1})$ para $0 \leq i < k$. Caso a Equação 1 seja insatisfatível, conclui-se que não há erro alcançável de profundidade k ao longo da intercalação π.

3.1.1. Redução de Ordem Parcial Monotônica

Para reduzir o número de intercalações em programas CUDA, o ESBMC-GPU implementa o algoritmo de redução de ordem parcial monotônica (*Monotonic Partial Order Reduction*, MPOR) proposto por [Kahlon et al. 2009]. O algoritmo classifica as transições dentro de um programa multitarefa que são independentes de ou dependentes de transições em outras *threads*, objetivando determinar se pares de intercalações sempre computam o mesmo estado, descartando os estados da RT que são duplicados [Morse 2015].

Para programas CUDA, o algoritmo MPOR foi aplicado para identificar os acessos à posições diferentes em vetores compartilhados, o que caracteriza acesso local, e permite desconsiderar os estados redundantes que seriam gerados referentes as trocas de contextos entre as *threads*. O acesso à posições diferentes ocorre comumente devido ao paradigma de programação concorrente do modelo CUDA, onde cada *thread* acessa uma posição do vetor baseado na linearização da configuração de *threads* e blocos. Isto garante o desempenho dos programas, pois não há necessidade de inserir barreiras de sincronização, não há condições de corrida e as operações de leitura/escrita podem ser feitas no mesmo ciclo.

3.2. Modelos operacionais

Em nossa proposta foram desenvolvidos modelos operacionais para simular o funcionamento da plataforma CUDA. O modelo operacional consiste basicamente de uma representação de um conjunto de métodos ou estruturas da linguagem de programação a ser verificada. Do ponto de vista de verificação, todo método usado em um código precisa ter seu modelo operacional implementado para simular o comportamento real do método, além de ser necessário incluir neste modelo as pré-condições, que devem ser verdadeiras antes da execução de um determinado trecho de código, e pós-condições, que devem ser verdadeiras após a execução deste trecho de código. Estes modelos são representados por um grafo de fluxo de controle que são posteriormente executados simbólicamente a fim de se criar os dois conjuntos de fórmulas SMT (*i.e.*, $\mathcal{C}$ e $\mathcal{P}$).

O modelo operacional também fornece meios para representar as chamadas de funções realizadas dentro de cada método. Apesar disso, a criação do modelo operacional insere apenas códigos úteis para verificação, desconsiderando, assim, chamadas irrelevantes, tais como métodos de impressão na tela, uma vez que não há propriedade a ser verificada, pois não focamos na verificação do *hardware*, ou métodos para criação de números aleatórios, onde não é necessário verificar o processo de geração dos números, pois tais funções são substituídas por funções não determinísticas implementadas no próprio verificador. Nestes exemplos, é necessário somente uma descrição da estrutura, a fim de que o verificador possa reconhecer a chamada da função como parte da linguagem. Com esta abordagem, também é possível simplificar o modelo a ser verificado, além de reduzir o tempo de verificação. Além disso, o modelo operacional inclui assertivas que objetivam verificar propriedades, tais como divisão por zero e limites de acesso aos vetores, o que garante a corretude de execução do programa.

Em função da plataforma CUDA ser fechada e não oferecer acesso aos seus códigos fonte, foi utilizado o guia de programação do CUDA como base para implementar os modelos operacionais [NVIDIA 2015]. Dentre as características das bibliotecas CUDA (*e.g.*, as funções da interface de programação de aplicativos), já reconhecidas pela ferramenta estão funções atômicas, randômicas, funções da API *Math*, funções de gerenciamento de memória, qualificadores e tipos de dados.

A Figura 1 apresenta um exemplo do modelo operacional desenvolvido para o método *cudaMalloc* que possui como argumentos o ponteiro para alocar memória no *device* e o tamanho em *bytes* necessário para a alocação. Comenta-se que o modelo operacional da função *malloc* do C/C++ foi utilizado para representar a alocação de memória no *device*, que verifica se a alocação foi realizada com sucesso (linhas 7 a 12). Em caso positivo, a função retorna `CUDA_SUCCESS`; caso contrário, é retornado o erro `CUDA_ERROR_OUT_OF_MEMORY`. A variável *lastError* é global e armazena o último `cudaError_t` para ser usado no método *cudaLastError()*. O método *cudaMalloc()* tem como pré-condição a alocação de memória de tamanho positivo, neste sentido as linhas 4 e 5 apresentam uma assertiva, na qual o tamanho a ser alocado deve ser maior do que zero. Caso haja uma violação nessa pré-condição, a ferramenta informa uma mensagem descrevendo o erro (*i.e.*, "Size must be greater than zero").

```
1  cudaError_t cudaMalloc(void ** devPtr, size_t size) {
2    cudaError_t tmp;
3    //pre-conditions
4    __ESBMC_assert(size > 0, ''Size must be greater
5    than zero'');
6    *devPtr = malloc(size);
7    if (*devPtr == NULL) {
8      tmp = CUDA_ERROR_OUT_OF_MEMORY;
9      exit(1);
10   } else {
11     tmp = CUDA_SUCCESS;
12   }
13   lastError = tmp;
14   return tmp;
15 }
```

Figura 1. Modelo operacional do método *cudaMalloc*.

A Figura 2 apresenta o modelo operacional do método $cudaMemcpy()$. No modelo é verificado a pré-condição para o tamanho de memória que será copiada (linha 3). Duas variáveis locais $cdst$ e $csrc$ (linhas 5 e 6) são usadas para receber os argumentos que representam o destino e origem da cópia de dados. Além disso, define-se a quantidade de *bytes* a ser copiada (linha 7) e realiza-se a cópia (linhas 8 e 9) dos dados entre o *device* e o *host*. Por fim, o método $cudaMemcpy()$ retorna para a estrutura tmp do tipo $cudaError\_t$, o valor $CUDA\_SUCCESS$.

3.3. Exemplo de Verificação de Programas CUDA

O código apresentado na Figura 3 é composto por 1 bloco e 3 *threads*, ao todo 3 *threads* são executadas (*i.e.*, $N = 3$). Este programa CUDA possui um *kernel* (linhas 3 a 5) que atribui valores dos índices das *threads* para o *array* que é passado por parâmetro. O objetivo do programador é instanciar as posições do *array*, conforme o índice da *thread*, mas há um equívoco no índice do *array*, pois o valor 1 é acidentalmente adicionado ao índice da *thread* (na linha 4). Como pode ser observado, na função *main()*, as posições do *array* são instanciadas com valor 0 (linha 11), e após a chamada de *kernel* (linha 13) é

```
1  cudaError_t cudaMemcpy(void *dst, const void *src,
2          size_t count, enum cudaMemcpyKind kind) {
3    __ESBMC_assert(size > 0, ''Size must be greater
4    than zero'');
5    char *cdst = (char *) dst;
6    const char *csrc = (const char *)src;
7    int numbytes = count/(sizeof(char));
8    for (int i = 0; i < numbytes; i++)
9      cdst[i] = csrc[i];
10   lastError = CUDA_SUCCESS;
11   return CUDA_SUCCESS;
12 }
```

Figura 2. Modelo operacional do método *cudaMemcpy*.

esperado pelo programador que $a[0] == 0, a[1] == 1, a[2] == 2$. Note que as variáveis M e N correspondem ao número de blocos e *threads*, respectivamente.

Neste exemplo, o algoritmo de verificação detecta a violação da propriedade de limites de *array*. De fato, o programa CUDA tenta acessar uma região de memória que não foi alocada, pois quando $threadIdx.x = 2$, o programa tenta acessar a posição $a[3]$. Analisando o modelo operacional do método *cudaMalloc()*, existe uma pré-condição que verifica se o valor do tamanho da memória a ser alocado é maior do que zero. Assertivas que verificam se o resultado corresponde ao esperado (linha 14) representam as pós-condições. Para este caso, em especial, foram produzidas 219 intercalações corretas e 18 falhas. Uma intercalação que falha, seria a execução em sequência das threads $t0 : a[1] = 0; t1 : a[2] = 1; t2 : a[3] = 2$, onde $a[3] = 2$ representa um acesso incorreto no índice do *array* a.

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

Figura 3. Fragmento de um programa CUDA para indexar *array*.

4. Resultados Experimentais

Esta seção descreve o planejamento, desenvolvimento, execução e análise de testes executados na ferramenta ESBMC-GPU, a fim de averiguar a corretude da verificação feita por esta, em códigos escritos usando a plataforma CUDA. Além disso, uma comparação entre as ferramentas ESBMC-GPU, *GPUVerify* [Kirk and Hwu 2010] e

PUG [Li and Gopalakrishnan 2010] é realizada. Os experimentos foram conduzidos em um computador equipado com Intel Core i7-4790 CPU 3.60 GHz, com 16 GB de RAM e SO Linux.

4.1. Planejamento e Desenvolvimento

Esta avaliação experimental analisa a habilidade da ferramenta ESBMC-GPU para verificar programas CUDA que possuam as seguintes características: realizem operações matemáticas (*e.g.*, soma, subtração, multiplicação), escritas em ponteiros, chamadas de funções `__device__`, uso de funções específicas de C (*e.g.*, *memset*, *assert*), de funções específicas de CUDA (*e.g.*, *atomicAdd*, *cudaMemcpy*, *cudaMalloc*, *cudaFree*, *__syncthreads*), de bibliotecas específicas de CUDA (*e.g.*, *curand.h*, *curand_kernel.h*, *curand_mtgp32_host.h*) e trabalhem com as variáveis *int*, *float*, *char*, bem como os modificadores de tipo *long*, *unsigned*, ponteiros para essas variáveis, ponteiros para funções, *typedefs* e variáveis intrínsecas de CUDA (*e.g.*, *uint4*).

Os experimentos realizados objetivam responder as seguintes perguntas: (1) Quais resultados o ESBMC-GPU obteve ao verificar os casos que compõem a suíte de testes especificada? (2) Qual é o desempenho do ESBMC-GPU quando comparado com o *GPUVerify* e PUG que são atualmente os verificadores de *kernels* para GPU estado da arte?

Para responder a estas perguntas, foram considerados 160 *benchmarks*, a maioria proveniente da suíte de testes da ferramenta *GPUVerify*[1]. É importante ressaltar que este verificador reconhece apenas funções *kernel*. Por isso, os casos que compõem a suíte não possuem suas respectivas funções *main*. Desta forma, foi necessário realizar a escrita de funções *main*, possibilitando a verificação completa por meio do ESBMC-GPU. Somente a chamada de *kernel* na função *main* deve ser substituída pelo *verify_kernel()* equivalente.

Para responder à pergunta (1), deve-se ressaltar que o ESBMC-GPU foi executado no sistema operacional *Linux* via terminal, utilizando o comando *esbmc file.cu*. Além disso, foram usadas algumas opções que configuram parâmetros para facilitar ou acelerar a verificação do ESBMC-GPU. Especificamente na verificação para programas CUDA, são necessárias as opções `--force-malloc-success`, que considera sempre haver memória suficiente no dispositivo para a verificação; `-DGPU_threads=NThreads` que informa o número total de *threads* que serão executadas, isto é, a soma de todas as *threads* de todos os blocos; `--context-switch C` para considerar um limite de troca de contexto entre *threads*; e `-I libraries` para especificar o diretório com as bibliotecas usadas pelo ESBMC-GPU. Como exemplo, tem-se: `esbmc arquivo.cu --force-malloc-success -DGPU_threads=3 --context-switch 2 -I libraries`.

Para responder à pergunta (2), aplica-se o *GPUVerify* à suíte de testes do ESBMC-GPU, sendo, para isso, necessário realizar algumas modificações: (a) retirar a função *main*; (b) verificar se a inicialização das variáveis realizada na função *main* é responsável pelo controle de alguma declaração condicional dentro do *kernel* e, se este é o caso, tal variável deve ser inicializada através da função `__requires()`; (c) verificar se há algum *assert* no *kernel*, se sim, este deve ser substituído pela função `__assert()`; d) verificar se existem outras funções exclusivas de bibliotecas C/C++ e retirá-las, pois não são suportadas pelo *GPUVerify*. No comando de execução, duas opções devem ser usadas `--gridDim=M` e `--blockDim=N`, onde são passados a quantidade de blocos e a quantidade de *threads* por blocos, respectivamente. Como exemplo da linha de comando, tem-se `gpuverify arquivo.cu --gridDim=8 --blockDim=2`. Para verificar a suíte de testes com a ferramenta PUG, foram necessárias algumas alterações no arquivo a ser verificado: (a) A extensão do arquivo é alterada de ".cu" para ".c"; (b) Devido o

[1]Os *benchmarks* estão disponíveis em http://esbmc-gpu.org

PUG não verificar a *main()*, esta é retirada ficando somente o *kernel*; (c) Bibliotecas proprietárias do PUG *my_cutil.h* e *config.h* são chamadas no arquivo. A primeira consta as definições para *structs*, qualificadores e tipos de dados. A segunda define o número de blocos e de *threads* por bloco; (d) O nome da função *kernel* deve obrigatoriamente ser chamada de "*kernel*". A linha de comando `pug kernel.c` executa a verificação.

4.2. Resultados Experimentais

Os resultados mostrados na Tabela 1 foram obtidos após executar as ferramentas ESBMC-GPU, *GPUVerify* e *PUG* usando os *benchmarks* descritos na seção anterior, onde cada linha desta 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 programas da suíte de testes.

Tabela 1. Resultados das ferramentas ESBMC-GPU ,*GPUVerify* e *PUG*

Ferramenta	**ESBMC-GPU**	**GPUVerify**	**PUG**
Resultados Corretos	**107**	94	53
Falsos Negativos	3	**2**	14
Falsos Positivos	**4**	5	5
Não Suportado	**37**	59	88
Timeout	9	**0**	**0**
Tempo(s)	15 912	160	**10**

A Tabela 1 mostra que o ESBMC-GPU encontrou 66.8% dos resultados corretos, o *GPUVerify* 58.7% e o PUG 33.1%. Tal porcentagem é extremamente notável, tendo em vista o curto tempo de desenvolvimento da ferramenta ESBMC-GPU comparado ao tempo de desenvolvimento da ferramentas *GPUVerify* e PUG.

O ESBMC-GPU gerou 3 resultados falsos negativos, devido à assertivas incluídas no *kernel* que não deveriam retornar falha (2) e à cobertura parcial da função *cudaMalloc()* para cópia de variáveis do tipo *float* (1). O *GPUVerify* gerou 2 resultados falsos negativos, que se devem a assertivas incluídas no *kernel* que não deveriam retornar falha (1) e à detecção incorreta de condições de corrida (1). Quanto ao PUG, foram gerados 14 resultados falsos negativos, todos ocasionados por condições de corrida detectadas incorretamente.

Quanto aos resultados falsos positivos, o ESBMC-GPU gerou apenas 4 , enquanto que o *GPUVerify* e PUG geraram 5 cada um. No ESBMC-GPU, este número se deve à inclusão de uma assertiva que deveria ter ocasionado a falha do *benchmark* (1), condições de corrida em memória compartilhada (2) e acesso à ponteiro nulo (1). No PUG, devem-se a não detecção de acesso à ponteiro nulo (1), a não detecção do erro de violação de limites de *array* (1), à inclusão de uma assertiva que deveria ter ocasionado a falha do *benchmark* (1) e a não detecção de condições de corrida (2). No *GPUVerify*, deve-se à condições de corrida (2) e à incapacidade de retornar os valores das funções *__device__* para o *kernel* de onde elas foram chamadas (3).

O ESBMC-GPU possui 37 *benchmarks* não suportados. Estes relacionam-se ao acesso à memória constante (4), ao uso de funções específicas de CUDA (*__mul24()*, *__threadfence()*) (3), ao uso de bibliotecas específicas de CUDA (*curand.h*

e *math_functions.h*) (9), e ao uso de ponteiros para funções, estruturas e variáveis do tipo *char* como argumentos em chamadas de *kernels* (21). Quanto ao *GPUVerify*, foram obtidos 59 *benchmarks* não suportados (22 a mais do que o ESBMC-GPU). O não suporte à verificação da função *main* explica a maioria destes casos (45), e os demais são explicados pela falta de suporte de ponteiros para funções (14), seja como argumento em funções *kernel*, seja em qualquer outra parte do programa CUDA. Quanto ao PUG, foram obtidos 88 casos não suportados (51 a mais do que o ESBMC-GPU). Assim como no *GPUVerify*, a não verificação da função *main* explica a maioria dos casos (40), enquanto os demais são explicados pela falta de suporte ao uso da função *__syncthreads* (12), de ponteiros para função (9) e da biblioteca *curand.h* (7); por não admitir o uso do modificador de tipo *unsigned* como argumento para a função *atomicAdd* (6), pela não detecção de modificações sobre variáveis guardadas em memória constante (4), pela falta de suporte a estruturas (2), a variáveis com o qualificador *__device__* (2) e do tipo *size_t* (1), além de outros casos que tiveram sua verificação abortada por motivos desconhecidos (5).

No ESBMC-GPU, a quantidade elevada de *threads* necessárias para a execução é o principal motivo de *timeouts* (9); para quantidade superiores a 4 *threads*, as intercalações geradas pela execução do *kernel* aumentam substancialmente o tempo de execução; apesar deste aumento considerável no tempo, estas intercalações são importantes para detectar erros de execução em funções *__device__*, que geraram falsos positivos no *GPUVerify*.

Analisando o tempo para verificação entre as três ferramentas, tem-se um tempo elevado de verificação referente ao ESBMC-GPU. Como mencionado, isso ocorre devido à execução concreta das intercalações do programa que leva em consideração todas as possíveis trocas de contexto entre as *threads*, percorrendo os possíveis caminhos de execução, podendo levar ao problema de explosão de estados e *timeouts*. No *GPUVerify* a análise é estática, realizada somente a nível de *kernel* sem considerar a intercalação deste com a *thread* principal. O tempo inferior do PUG é explicado pela análise sobre 2 *threads* somente.

Apesar de mais demorada, a análise das intercalações tem como vantagem o retorno correto para as funções *kernel* dos valores das funções *__device__*. Melhorias no tempo de execução têm sido implementadas durante o desenvolvimento, como o MPOR (Seção 3.1.1), que gerou uma melhoria de 80% no tempo de execução da suíte de testes, reduzindo consideravelmente a quantidade de intercalações na verificação dos programas quando, comparado a versões anteriores da ferramenta.

5. Trabalhos Relacionados

Existem poucas ferramentas que verificam programas para GPU, e as existentes estão restritas a identificar erros específicos, além de serem limitadas por suas técnicas de verificação.

A ferramenta *GPUVerify* [Betts et al. 2012] usa uma semântica para verificação de *kernels* (*synchronous, delayed visibility*, SDV) para detecção de condições de corrida e divergência de barreira. Além disso, executa sobre o sistema de verificação Boogie [Le Goues et al. 2011] e o solucionador SMT Z3 [De Moura and Bjørner 2008]. *GPUVerify* tem como entrada apenas o *kernel*, não levando em consideração as funções da *main*, sendo incapaz de verificar a execução do programa e podendo assim gerar resultados incorretos em programas CUDA (conforme mostrado nos resultados experimentais).

SESA (*Symbolic Executor with Static Analysis*) [Li et al. 2014] e GKLEE [Li et al. 2012] são baseados na execução *concolic* (concreto mais simbólico) sobre programas C++/CUDA e são diferentes na maneira de determinar as variáveis simbólicas. Enquanto SESA faz uma avaliação automática, GKLEE necessita que o usuário defina estas variáveis. Além disso, SESA verifica aplicações reais usando a configuração original do número de *threads* e seu foco é a verificação de condições

de corrida. Apresenta também resultados não estáveis para erros de acesso fora dos limites em *arrays*. Embora tenha como entrada um arquivo com o *kernel* e a *main*, não apresenta a verificação para erros de execução, onde os resultados obtidos são diferentes do esperado. GKLEE apresenta resultados para verificação de sincronização de barreira, corretude funcional através de assertivas, defeitos de desempenho e condições de corrida.

PUG (*Prover of User GPU Programs*) [Li and Gopalakrishnan 2010] analisa *kernels* automaticamente usando solucionadores SMT e detecta erros de condições de corrida, sincronização de barreira e conflitos de banco em memória compartilhada. PUG enfrenta problemas com derivação de invariantes para *loops* que podem levar a resultados incorretos na verificação, sendo necessário que o usuário forneça as invariantes. Problemas também são encontrados em operações de aritmética de ponteiros e no suporte à funcionalidades avançadas de C++.

ESBMC-GPU usa uma abordagem próxima ao PUG e não apresenta problemas com invariantes [Rocha et al. 2015], geradas automaticamente, por possuir resultados estáveis na verificação de programas C/C++, incluindo operações com aritmética de ponteiros e suporte à funcionalidades avançadas da linguagem (como tratamento de exceção, polimorfismo e herança) [Ramalho et al. 2013]. Como todas as ferramentas supracitadas, ESBMC-GPU é capaz de detectar condições de corrida em *kernels* e, além disso, identificar acessos fora dos limites em *arrays* e apresentar resultados concretos para verificação da execução dos programas CUDA.

6. Conclusões e Trabalhos Futuros

O trabalho alcançou o objetivo esperado, que consistia no desenvolvimento de um método capaz de reconhecer programas produzidos sob a plataforma CUDA em C/C++, além de validar o programa CUDA usando o verificador de modelos ESBMC, ferramenta baseada nas técnicas BMC e SMT para verificação de sistemas, utilizando modelos operacionais para reconhecer as diretivas do CUDA. Esta é a primeira aplicação do algoritmo de verificação preguiçosa usando MPOR para verificar programas que executam em GPU.

Os resultados dos experimentos comprovaram a eficácia do verificador, apesar da redução de desempenho causada pelo aumento do número de *threads*. De todos os *benchmarks* verificados, $66,8\%$ apresentaram verificação bem-sucedida, comparados com os $58,7\%$ de cobertura da ferramenta *GPUVerify* e $33,1\%$ da ferramenta PUG.

Os próximos passos desta pesquisa incluem a detecção de divergência de barreira, aumento de suporte a diferentes tipos de argumentos na função de verificação de *kernels*, suporte a novos tipos de memória (*e.g.*, memória pinada, unificada) e aplicação de técnicas para redução da quantidade de intercalações, preservando a semântica do programa. Além disso, técnicas para redução do tempo de verificação (*e.g.*, verificação sobre duas *threads*) estão sendo analisadas e novas comparações com ferramentas de verificação de *kernels* (*e.g.*, GKLEE e SESA) estão em desenvolvimento.

Agradecimentos

Este projeto de pesquisa foi apoiado pelo Instituto de Desenvolvimento Tecnológico (INDT).

References

Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. The MIT Press.

Barrett, C. W., Sebastiani, R., Seshia, S. A., and Tinelli, C. (2009). Satisfiability Modulo Theories. In *Handbook of Satisfiability*, pages 825–885.

Betts, A., Chong, N., Donaldson, A. F., Qadeer, S., and Thomson, P. (2012). GPUVerify: a verifier for GPU kernels. In *OOPSLA*, pages 113–132.

Biere, A., Cimatti, A., Clarke, E. M., Fujita, M., and Zhu, Y. (1999). Symbolic Model Checking Using SAT Procedures instead of BDDs. volume 1579 of *LNCS*, pages 317–320.

Bradley, A. R. and Manna, Z. (2007). *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer.

Cheng, J., Grossman, M., and McKercher, T. (2014). *Professional CUDA C Programming*. John Wiley and Sons, Inc.

Cordeiro, L. C. and Fischer, B. (2011). Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking. In *ICSE*, pages 331–340.

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

De Moura, L. and Bjørner, N. (2008). Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340.

Kahlon, V., Wang, C., and Gupta, A. (2009). Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In *CAV*, volume 5643 of *LNCS*, pages 398–413.

Kirk, D. B. and Hwu, W.-m. W. (2010). *Programming Massively Parallel Processors*. Elsevier Inc., 1st edition.

Le Goues, C., Leino, K. R. M., and Moskal, M. (2011). The Boogie Verification Debugger (Tool Paper). In *SEFM*, volume 7041 of *LNCS*, pages 407–414.

Li, G. and Gopalakrishnan, G. (2010). Scalable SMT-based Verification of GPU Kernel Functions. In *FSE*, pages 187–196.

Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., and Rajan, S. P. (2012). Gklee: Concolic verification and test generation for gpus. In *PPoPP*, pages 215–224. ACM.

Li, P., Li, G., and Gopalakrishnan, G. (2014). Practical Symbolic Race Checking of GPU Programs. In *SC*, pages 179–190.

Morse, J. (2015). *Expressive and Efficient Bounded Model Checking of Concurrent Software*. University of Southampton, PhD Thesis.

Morse, J., Cordeiro, L. C., Nicole, D., and Fischer, B. (2013). Handling Unbounded Loops with ESBMC 1.20 - (Competition Contribution). In *TACAS*, volume 7795 of *LNCS*, pages 619–622.

Morse, J., Ramalho, M., Cordeiro, L. C., Nicole, D., and Fischer, B. (2014). ESBMC 1.22 (Competition Contribution). In *TACAS*, volume 8413 of *LNCS*, pages 405–407.

NVIDIA (2015). *CUDA C Programming Guide*. NVIDIA Corporation, v7.0 edition.

Ramalho, M., Freitas, M., Sousa, F., Marques, H., Cordeiro, L., and Fischer, B. (2013). SMT-Based Bounded Model Checking of C++ Programs. In *ECBS*, pages 147–156.

Rocha, H., Ismail, H., Cordeiro, L. C., and Barreto, R. S. (2015). Model Checking C Programs with Loops via k-Induction and Invariants. *CoRR*, abs/1502.02327.