SAFETY VERIFICATION OF CUDA AND DEEP NEURAL NETWORKS

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2023

Student id: 10868835

Department of Computer Science

Contents

Word Count: 12342

List of Tables

List of Figures

Listings

Abstract

SAFETY VERIFICATION OF CUDA AND DEEP NEURAL NETWORKS
Xianzhiyu Li
A dissertation submitted to The University of Manchester
for the degree of Master of Science, 2023

CUDA [1], powered by NVIDIA's GPU, has gained traction in deep learning due to its computational efficiency. However, like other software, CUDA programs are prone to various errors, such as data race and array bounds violations, with their parallel nature complicating debugging. This complexity necessitates deeper knowledge of the CUDA architecture, highlighting the importance of safety verification tools. ESBMC-GPU [3], an initial solution using an abstract representation of CUDA APIs, But it is no longer available due to lost maintenance. ESBMC v7.3 is a powerful SMT-based bounded model checker to verify C and C++ programs. This project aims to port the ESBMC-GPU verification of CUDA programs to the latest ESBMC. We have improved ESBMC's Clang-based frontend, fixed outdated code, and optimised the verification time. We also make operational models for the convolution operations and activation functions in CuDNN. Finally, we use the benchmark to evaluate our work. The results show that this project successfully implements the CUDA program verification based on the latest ESBMC, and we get much better verification efficiency than ESBMC-GPU, and get a pass rate of 79.2%.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank my supervisor Dr. Lucas Cordeiro for his guidance during the project. This project cannot be left without his encouragement and professional advice. At the same time, I am also very thankful to the PhD student Kunjian Song in Dr. Lucas Cordeiro's laboratory for his code support and insights.

Chapter 1

Introduction

1.1 Motivation

Compute Unified Device Architecture (CUDA) [1] is a platform and programming model for parallel computing, based on the parallel computing power of NVIDIA Graphics Processing Unit (GPU), and is widely used in scientific computing such as deep learning because it enables faster and more efficient computing and data processing. Like other computer programs written by programmers, CUDA programs can encounter various types of errors, including data contention, array out-of-bounds, and division by zero violations, etc.

CUDA program executes part of the code in GPU to accelerate. Hence, it involves data transfer and synchronisation between GPU and CPU, which also makes it more complex to debug errors. For example, the problem of memory leaks may occur due to asynchronous execution between GPU and Central Processing Unit (CPU). In order to reduce the probability of program errors, developers need to have a deep understanding of the CUDA programming model and GPU architecture, which undoubtedly increases the learning cost. Due to this feature, a software capable of performing safety verification of CUDA programs becomes essential. In previous research, ESBMC-GPU [3] was proposed for verifying CUDA programs based on Efficient SMT-Based Context-Bounded Model Checker (ESBMC) v2.0 [4, 5, 6], an open-source model checker for C++/ANSI-C that relies on SMT solvers. It works by using an abstract representation of the native CUDA APIs that conservatively approximates their semantics, It's called CUDA operational model (COM). However, due to some system standard library and C++ syntax changes, the software has become inoperative.

CUDA deep neural network (CuDNN) [7] is an important extension of CUDA.

It provides various basic operations of deep neural network (DNN), such as convolution, pooling, activation function, etc. It also adjusts the algorithm to improve the computational speed of DNN and take full advantage of the performance of GPU. Nevertheless, DNN is high parallel and Its parameters are difficult to interpret, which makes troubleshooting difficult. In particular, some critical application areas can lead to serious consequences if errors occur, such as autonomous driving. With its increasing use in safety-critical applications, the security of its systems has become the focus of attention. An important aspect of securing DNN is to verify the correctness of the underlying software libraries - CuDNN.

1.2 Aims and Objectives

The primary aim of this project is to implement the safety verification of CUDA programs in the latest ESBMC v7.3[1], and on this basis, propose a method for verifying CuDNN, then successfully verify real applications that rely on CuDNN library. The objectives of the project are as follows:

- Fully understand the ESBMC workflow and source code to add CUDA features on top of the current front-end code.
- Investigate the principle of COM proposed in ESBMC-GPU, fix and improve the model to make it suitable for the latest ESBMC.
- Investigate the principle of CuDNN, try to model it, and ensure that the operational model returns the same results as the original APIs.
- Use ESBMC v7.3, which was improved in this project, to verify the existing benchmarks and compare it with the state-of-the-art CUDA validation framework.

1.3 Contribution

Over the past year, ESBMC team has integrated the latest Clang-based C++ frontend into ESBMC. ESBMC started with the C++ frontend of CPROVER [8]. However, C++ and its variant CUDA are constantly being updated [9], which brings a lot of

[1] `http://www.esbmc.org/`

maintenance costs and challenges. In this project, we ported the abstract representation of the CUDA library to our latest ESBMC, and fixed and improved the current Clang-based C++ frontend based on CUDA features. In view of the existing methods of CUDA verification, we propose a simplified process, which greatly reduces the time required to verify CUDA programs. We additionally implemented operational models from the CuDNN library on convolution operations and activation functions.

According to the evaluation results, ESBMC v7.3 has higher accuracy than other CUDA verification frameworks. Compared with ESBMC - GPU, increased about 4 times the verify efficiency at the same time, also keep up with the same level of performance.

1.4 Report Structure

The structure of this report has four layers. The first layer is the introduction section, which introduces the importance of verifying the correctness of CUDA and CuDNN and expounds the purpose of the current research, as well as our research objectives and expected results. The second level is a literature review, which introduces the current research, background and theory. The third layer is the methodology, which shows the improvement of the existing open source software ESBMC, and the porting of the CUDA operational model and the modelling of CuDNN. The fourth layer is for experiments and evaluation. We use benchmarks to compare with ESBMC-GPU and some other GPU verification software. The last layer is the conclusion and future work, which summarises the results achieved in this project and analyses the direction that can be improved in future research.

Chapter 2

Background

This chapter will lay the theoretical foundation for the following methodology and implementation. This chapter will start from the introduction of CUDA programs, to the verification theory based operational model proposed by ESBMC-GPU, and show some main function algorithms and verification strategies. Next, it introduces the implementation of parts in CuDNN using some working principles and formulas. Finally, we systematically introduces the architecture of latest ESBMC, as well as its internal verification framework and theoretical knowledge. Furthermore, we introduce the state-of-the-art CUDA verification frameworks.

2.1 CUDA Program

CUDA is a GPU programming model developed by NVIDIA Corporation [1]. Developers can use the simple interface of GPU programming to build applications based on GPU computing. Its underlying language provides support for other programming languages, such as C/C++, Python, Fortran and other languages. This project mainly focuses on CUDA C/C++ as the research direction.

In CUDA, the execution of programs is based on threads. It is designed so that thousands of threads can be executed concurrently, and it uses a hierarchical thread structure to manage a large number of threads efficiently [10]. Within the CUDA architecture, the thread stands as the foundational execution unit, bearing characteristics akin to those of CPU threads. The block serves as a conglomerate of multiple threads, collaboratively undertaking a unified task, while the grid functions as an assembly of such blocks. Intra-block threads benefit from shared memory, facilitating communication. However, inter-block thread interactions necessitate the use of global memory.

The thread hierarchy is shown in Figure 2.1.

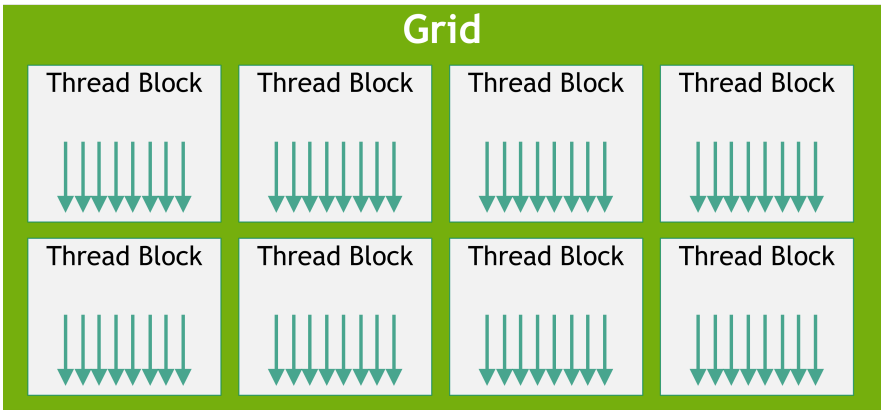

Figure 2.1: Thread Hierarchy [1]

CUDA programming model is heterogeneous, which means the cooperation of GPU and CPU. There are two important concepts in CUDA programming: **host** and **device**, the **host** represents the CPU and its memory, and the **device** represents the GPU and its memory. Their programs are both contained in the CUDA code and run in the CPU and GPU respectively. Therefore, the developer needs to distinguish the code on the host and device, In the CUDA program, the function type qualifier is added to distinguish the function on the host and device, such as the example CUDA code 2.1. The following are the three primary function type qualifiers:

(i) `__global__`: Executed on device, called from host, must return void, also known as the kernel function.

(ii) `__device__` : Only be called from the device and executed on the device.

(iii) `__host__` : Executed on the host and can only be called from the host. In general, it can be omitted.

Data copy can be carried out between the host and the device through communication. The execution process of a simple CUDA program can be summarised as follows:

1. Allocate GPU memory and copy data: Before executing the CUDA program, memory needs to be allocated on the GPU. This is done through the API functions provided by CUDA, such as `cudaMalloc`. Once the allocation is done, transfer the data from CPU memory to GPU memory using the `cudaMemcpy` API function.

2. Execute parallel computation on GPU: Use kernel functions to perform parallel computation on GPU. When a kernel is executed, specify the number of execution threads and the number of thread blocks, such as `kernel<<<grid, block>>>`. Thread blocks are collections of threads that can be assigned to different compute units on the GPU for execution. In summary, the computing unit can compute simultaneously while executing the same kernel function, thus achieving the acceleration effect of parallel computing.

3. Calculation done: Release the device memory after transferring the result of the calculation from the device memory back to the host memory.

Listing 2.1: An example CUDA code

```
1  #include <stdio.h>
2
3  __global__ void helloCUDA() {
4      printf("Hello␣CUDA!\n");
5  }
6
7  int main() {
8      helloCUDA<<<1,1>>>();
9      cudaDeviceSynchronize();
10     return 0;
11 }
```

2.2 CUDA Operational Model

This project will be developed based on the operational model proposed in ESBMC-GPU [3]. The operational model is designed to reliably simulate the behaviour of the CUDA library. A set of abstract representations of methods and data structures that roughly approximate the semantics of the CUDA library, each method simulates the library's actual behaviour. The correctness is ensured by setting assertions at the beginning and end of the method (*e.g.*, pre-conditions and post-conditions), and to simplify the operational model, remove methods that have no properties to check and keep the relevant methods [3]. As a result, model verification is made significantly simplified and takes less time. At the same time, the operational model also supports the functionality available in ESBMC, such as data race checking. The main principles and methods of this operational model are briefly reviewed in this chapter.

2.2.1 Principle

One of the models of the underlying functionality of CUDA is `cudaMalloc` [3], which aims to simulate the behaviour of CUDA programs to allocate memory in GPU device. To ensure the security of this method, assertions are added to the input and output parts of the model, and the function is realised by using the `Malloc` method of ANSI-C. Finally, the result of memory allocation is judged by checking whether the pointer is NULL, and the status of the method is recorded in real time. When the memory allocation error occurs, the assertion failure is triggered to find a counterexample.

Algorithm 1 cudaMalloc operational model

Function: $cudaMalloc(\textbf{\textit{void}} **devPtr, \textbf{\textit{size\_t}}\ size)$

1: Pre-condition Assertion: *size* must be greater than zero
2: Allocate block memory for pointer $devPtr$ by using $Malloc$ in ANSI-C, magnitude is *size*
3: If the pointer $devPtr$ in step 3 is not NULL, go to step 5; otherwise, go to step 6.
4: $CUDA\_SUCCESS$ is assigned to the global variable $LastError$
5: $CUDA\_ERROR\_OUT\_OF\_MEMORY$ is assigned to the global variable $LastError$
6: Post-condition Assertion: check the status is $CUDA\_SUCCESS$

Output: $LastError$

The second CUDA underlying functional model is `cudaFree` [3], algorithm 2 shows the operational model where a pointer to a variable is passed as an input parameter in order to free the allocated memory. As in the previous model, `Malloc` is referenced to simulate functionality, so the `cudaFree` function also references the ANSI-C `free` function, although the allocation and release of memory in CUDA programs are in the memory of the GPU. The reason why the simulation operation can be carried out in this way is that in the theory involved in ESBMC solving, tuples are used to represent the memory allocation and release model, and the memory hierarchy is not taken into account when performing the analysis. At the beginning of the function, the $devPtr$ pointer to the memory block allocated by `Malloc` is checked whether it is NULL to avoid freeing a nonexistent pointer or double freeing. If the operation is successful, then the method return $CUDA\_SUCCESS$ [3].

The final model of the underlying functions of CUDA is `cudaMemcpy`, When a user creates an array of data on the host and wants to compute it on the GPU, this method can transfer the array to GPU memory, and this method can also transfer the computed data back to the host memory. Algorithm 3 shows the model, a local variable is created,

Algorithm 2 cudaFree operational model

Function: $cudaFree(\textbf{\textit{void}} **devPtr)$

1: Pre-condition Assertion: *devPtr* must not be NULL
2: Deallocate memory from pointer *devPtr* using the free method in ANSI-C
3: *CUDA_SUCCESS* is assigned to the global variable *LastError*

Output: *LastError*

initialised, and copied. The Algorithm starts by checking that the initialised array is a safe size, and ends by checking that the copy operation was successful [3].

Algorithm 3 cudaMemcpy operational model

Function: $cudaMemcpy(\textbf{\textit{void}} *dst, \textbf{\textit{const void}} *src, \textbf{\textit{size\_t}}\ size, \textbf{\textit{cudaMemcpyKind}})$

1: Pre-condition Assertion: *size* must be greater than zero
2: Initialise local variable with *dst* and *src* content, named *cdst* and *csrc*
3: Initialise the number of bytes *numbytes* to copy based on the size
4: Copy *numbytes* positions from *csrc* to *cdst*
5: *CUDA_SUCCESS* is assigned to the global variable *LastError*
6: Post-condition Assertion *cdst* and *csrc* must be same data

Output: *LastError*

In summary, the functions of CUDA are represented by native functions in C/C++ programming language in the operational model for verification in ESBMC. These CUDA-native functions all behave in accordance with ANSI-C semantics. At the same time, the functions and memory hierarchy about hardware are omitted in COM, and only the critical operations are retained.

2.2.2 Call kernel

ESBMC can verify multi-threaded programs in C/C++ programming languages using POSIX library [4]. In COM, thread instructions of CPU will be used to simulate threads in GPU, in order to be able to use this method for CUDA kernel verification, intrinsic functions of ESBMC are used to transcode kernel calls [3].

Listing 2.2: Parameters struct

```
1 struct arg_struct
2 {
3   int *a;
4   int *b;
5   int *c;
6   void *(*func)(int *, int *, int *);
7 };
```

In CUDA programs, multiple arguments may be required to call a parallel function. In the C thread library it is usually allowed to pass a $void*$ argument to a thread function. To enable thread equations to use multiple parameters, a struct is created, and packing multiple arguments into it. finally pass the address of the struct to the thread function, as shown in Listing 2.2.

$$pthread\_create(\&tid, NULL, thread\_func, NULL); \tag{2.1}$$

Function 2.1 a method of the POSIX threads library, it is used to create new threads, where threads are the smallest independent unit of execution allowing concurrency. When new threads are created, each thread can run concurrently and perform a different task. Each thread operates with its own dedicated stack space, ensuring that threads function independently and without interference from one another. In COM, parallel operations on the GPU are emulated by this function to enable ESBMC to detect errors in multi-threaded processes.

$$pthread\_join(\&tid, NULL); \tag{2.2}$$

Function 2.2 is a method in the POSIX thread library for waiting for a specified thread to terminate and reclaiming the resources of the terminated thread. Within COM, its primary role is synchronisation. Specifically, when multiple threads compute and rely on global variables to store their outcomes, the main thread can pause, ensuring all threads finalise their computations before proceeding.

The ESBMC-GPU needs to modify the functions of the CUDA calling kernel for verification, such as replacing `kernel<<<grid, block>>>` with the inner function `ESBMC_verify_kernel(kernel, grid, block)`. The algorithm is tailored to verify CUDA programs with up to three arguments. In CUDA programming, *gridDim* and *blockDim* are two fundamental built-in variables. Specifically, *gridDim* denotes the number of blocks in a grid, while *blockDim* signifies the number of threads within a block. In COM, both are represented as three-dimensional struct types. This algorithm uses constructor functions to assign values to these struct. The essence of the model is encapsulated in the implementation of this function. It should be underscored that utilising ESBMC-GPU to verify CUDA code obviates the need for the CUDA toolkit installation.

Algorithm 4 kernel operational model

Function: $ESBMC\_verify\_kernel(\textbf{\textit{RET}} * kernel, \textbf{\textit{BLOCK}}\ blocks, \textbf{\textit{THREAD}}\ threads, \textbf{\textit{TI}}\ arg1, \textbf{\textit{T2}}\ arg2, \textbf{\textit{T3}}\ arg3)$

1: *gridDim* is assigned a value from the *blocks* parameter
2: *blockDim* is assigned a value from the *threads* parameter
3: Encapsulate the arguments within a struct and invoke *pthread_create* to spawn child threads, passing the struct's pointer as an argument.
4: Synchronise and terminate each child thread by calling *pthread_join*

2.2.3 Two-thread Analysis

To mitigate the state explosion issue and curtail the verification duration, ESBMC-GPU adopts a Two-thread analysis approach. Given that CUDA programs typically leverage array computations and deploy multiple threads to expedite the process, the Two-threaded analysis aptly emulates this behaviour. This approach retains potential errors, such as data races, thereby providing ESBMC with the necessary Features to identify errors.

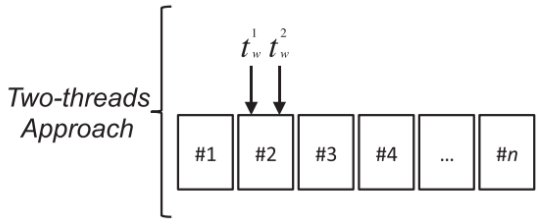

Figure 2.2: Two-thread verification method

2.3 CuDNN Library

CuDNN is a library developed by NVIDIA for deep neural network acceleration [7]. It is based on CUDA and can be used to speed up the training of neural networks. For example, Convolutional Neural Network (CNN) is one of the most widely used neural networks in deep learning. In the fields of speech recognition, natural language processing, computer vision, and others, it has produced excellent results. The basic structure of CNN includes convolutional layer, pooling layer, activation function, fully connected layer, etc.

2.3.1 Convolutional Method

CNN model operation, usually takes the longest time to calculate the convolutional layer, which is mainly used to extract the features of the image and requires a lot of computing resources [11]. Therefore, it needs to use high-performance computing devices such as GPU for calculation acceleration. Especially in deep CNN, the computation of convolutional layers can increase exponentially with the depth of the network, leading to a significant increase in training time [12]. The convolutional layer can be seen as sliding the filter over the image, extracting the local information in the image through the multiplication and addition operation, and generating a new feature map. An example of the calculation process is shown in Figure 2.3.

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t-\tau)\, d\tau \tag{2.3}$$

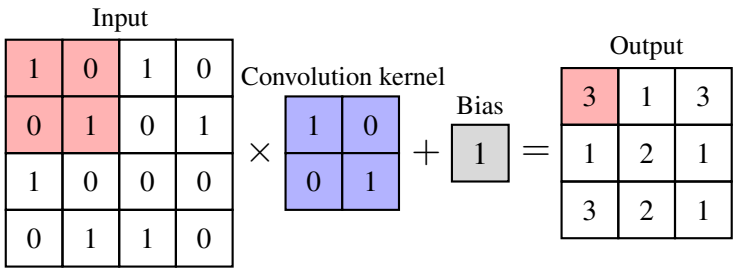

Figure 2.3: Convolution example

Specifically, when performing the convolution operation, the image and the convolution kernel are multiplied element-wise and added together to obtain the convolution result. This method will cause a large amount of computation. GPU are better at matrix operations than convolutions, so in CuDNN, converting convolutions into matrix operations makes the computation more efficient. For example, it expands the input image and convolution kernel into a matrix, and then performs matrix multiplication to calculate the convolution result. This approach take full advantage of the parallel computing of GPU [13].

2.3.2 Activation Function

One of the most important components of a neural network is the activation function, also referred to it as the hidden unit or nonlinear mapping function. Adding activation function to neural network introduces nonlinear property and enhances the learning

ability of neural network. So the main characteristic of activation function is non-linearity. There are various activation functions corresponding to different features [14]. In the actual modelling process, different activation functions need to be selected according to the requirements. CuDNN library provides a variety of activation function implementations.

The Rectified Linear Unit (ReLU) function 2.4 boasts rapid computational capabilities, delivering an output of 0 for inputs less than 0 and mirroring the input for values greater than 0. Its gradient is elegantly straightforward, yielding results of either 0 or 1, thereby mitigating the risk of gradient vanishing. Owing to its computational expediency and superior performance, ReLU is ubiquitously adopted in deep learning realms, notably within convolutional neural networks and deep residual networks.

$$ReLU(x) = \begin{cases} 0, & x \leq 0. \\ x, & x > 0. \end{cases} \tag{2.4}$$

$$ReLU'(x) = \begin{cases} 0, & x \leq 0. \\ 1, & x > 0. \end{cases} \tag{2.5}$$

The Sigmoid function 2.6 is smooth and continuous, with an output range confined between 0 and 1. This characteristic renders it particularly effective in scenarios where the output is to be interpreted as a probability. Furthermore, its curvaceous nature enables it to capture nonlinearities when employed as an activation function in neural networks. While the sigmoid function is notably straightforward and conducive to implementation, its training efficacy can diminish due to the vanishing gradient problem, especially when the function's input is excessively large or small. This phenomenon leads the gradient towards zero, impeding model optimisation. As a result, ReLU and other alternative activation functions have become more prevalent in contemporary deep learning applications.

$$Sigmoid(x) = \frac{1}{1+e^{-x}} \tag{2.6}$$

$$Sigmoid'(x) = \frac{e^{-x}}{(1+e^{-x})^2} \tag{2.7}$$

In deep learning, the Exponential Function often appears in the activation function, loss function, which usually has a large amount of calculation. CuDNN uses Fast Exponential Function Approximation method to approximate the value of the exponential

function. It works by approximating the curve of an exponential function using simple arithmetic operations, which greatly reduces the time taken to compute, but at the cost of some accuracy.

2.4 Bounded Model Checking (BMC)

The safety and reliability of software cannot be separated from safety verification. Model checking is an automated verification method in which systems are modelled as finite state Machine or finite transition system. It checks whether the system conforms to a given property by traversing all the states in the system [15]. In some cases, the system state space may be very large, so some restrictions can be added to improve the efficiency of model checking, such as bounded model checking. BMC limits the length of execution paths, allowing it to be used to verify systems with infinite state spaces [15].

State transition systems can be described by Kripke structures, it has set of states S. $s_0, s \subseteq S$ and s_0 represents the set of initial states, and s represents the current value of all key variables and the value of the program counter during the execution of the program [3]. Based on the Kripke structure, a verification condition formula in BMC is as follows [16]:

$$\psi_k^{\pi} = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} R(s_j, s_{j+1}) \wedge \neg\phi(s_i) \tag{2.8}$$

Where k represents the number of unwinding loops and i is the set of initial states, the range of $0 \leq i < k$. $R(s_j, s_{j+1})$ is the transition relation between state s_j and s_{j+1}, and $I(s_0) \wedge \bigwedge_{j=0}^{i-1} R(s_j, s_{j+1})$ represent executions of length i. In simple terms, if Eq (2.8) is not satisfied, then it indicates that the safety property is satisfied in all the state sequences, otherwise, it violates the safety property [16].

2.5 ESBMC

Satisfiability Modulo Theories (SMT)-Based Context-Bounded Model Checker (ESBMC) is an open-source and mature software verification tool. It is based on the principles of BMC and verifies the correctness of the program using an SMT solver. SMT is a research field that deals with the satisfiability of first-order formulas with respect to a background theory [17]. ESBMC supports not only C/C++, but also Kotlin, and Solidity programs. It can detect errors in programs, such as data races, deadlocks,

memory leaks, etc, and generate reports during the verification process to help developers find and fix existing errors when verifying programs, and improve the reliability and security of programs. This project is based on the verification extension of C++ program in ESBMC, the structure of the tool is shown in Figure 2.4.

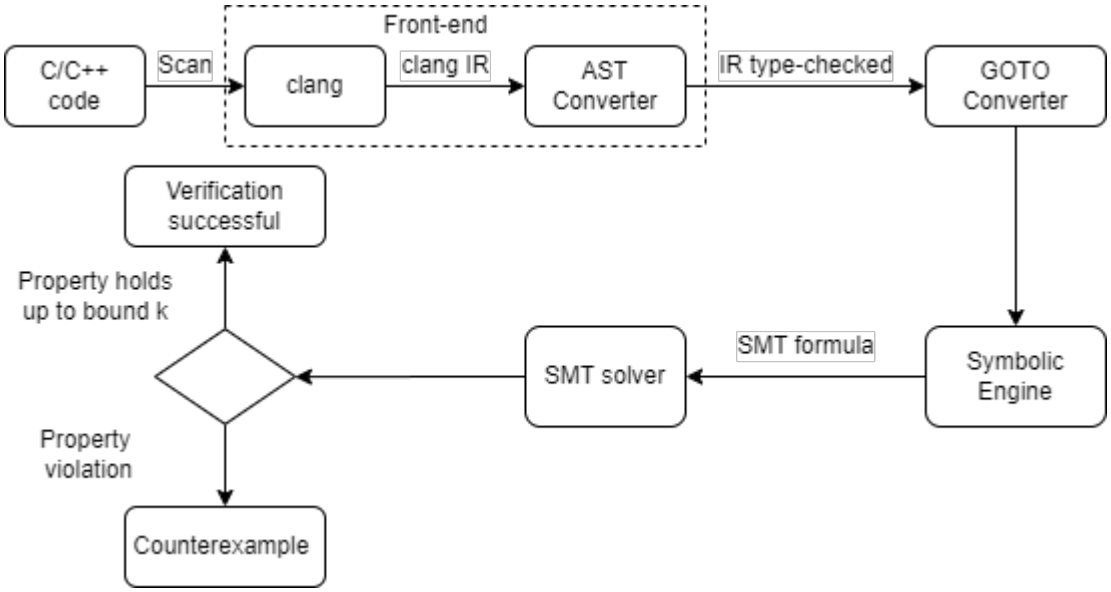

Figure 2.4: ESBMC architectural for C/C++ program [2]

The main difference from ESBMC-GPU [3] is that ESBMC is actively maintained and updated for several versions, ESBMC uses clang as its front-end, which has good compatibility, it can run on different platforms, and has excellent error analysis to provide more detailed error information [18]. The Clang-based front-end parses the program to produce a Abstract Syntax Tree (AST), which is converted to Intermediate Representation (IR), and then type-checked to avoid type errors, such as assignment checks, function call checks, etc.

To make the original programme simpler, the output IR generates the GOTO programme using the GOTO converter. The GOTO programme is run through symbolic execution, replaces all loops with GOTO language, and then converts it into Static Single Assignment (SSA) [19]. Finally, the logic formula $C \wedge \neg P$ (C represents constraints and P represents the properties) can be obtained based on SSA expressions, and the satisfiability of the formula can be verified by SMT solver. If the given property is violated, a counterexample will be generated, and if the property holds, the verification succeeds. ESBMC currently supports a variety of SMT solvers: Z3 [20], Bitwuzla [21], Boolector [22], MathSAT [23], CVC4 [24], Yices [25].

2.6 Clang-Based Frontend in ESBMC

The Low-Level Virtual Machine (LLVM) [26] is an open-source compiler project originating from Apple Inc., initially intended to supersede GCC for mac-OS with a focus on prioritising Objective-C. Utilising Static Single-Assignment for compilation, LLVM boasts a modular architecture, enabling proficient support for various programming languages. Presently, it serves as a foundational component for numerous commercial and research-based projects [27].

Clang serves as the front-end compiler for LLVM, natively supporting C/C++ languages and being implemented in C++. Notably, on mac-OS platforms, Clang's processing capability surpasses that of GCC by a factor of 1.6. When juxtaposed with the conventional GCC, Clang demonstrates superior attributes: it boasts a rapid compilation process, with its preprocessing being approximately 1.4 times more efficient, and exhibits optimised memory utilisation. Furthermore, Clang's static analysis, which includes building an abstract syntax tree, works about four times faster than GCC and uses less memory [26].

In essence, while traditional compilers focus on transforming high-level source code into executable binary or intermediate code for computational execution, the compiler within ESBMC primarily centres on source code analysis to detect potential inconsistencies or errors. Contrary to the output-driven nature of conventional compilers, ESBMC provides a correctness report on the source code, pinpointing vulnerabilities, violated assertions, or unsatisfied conditions.

2.6.1 Abstract Syntax Tree (AST)

The Abstract Syntax Tree (AST) illustrates the hierarchical structure of source code in a programming language. It captures both syntax and semantic information in a tree format, which means the tree's layout might not strictly follow the order of the original source code. In the AST, each node symbolises a fundamental semantic component from the code. Being "abstract", the AST omits some specific details from the actual code semantics.

Listing 2.3: Example addition code

```
1 void function(int a, int b, int &c) {
2     c = a + b;
3 }
```

For example, the function above calculates the sum of two integers and updates the result directly into the original variable. The AST generated by Clang is shown in Figure 2.5 using flag: *–parse-tree-only* from ESBMC.

```
|-FunctionDecl 0x8074000 <main3.cpp:1:1, line:3:1> line:1:6 function 'void (int, int, int &)'
| |-ParmVarDecl 0x8073de8 <col:17, col:21> col:21 used a 'int'
| |-ParmVarDecl 0x8073e68 <col:25, col:29> col:29 used b 'int'
| |-ParmVarDecl 0x8073f18 <col:33, col:38> col:38 used c 'int &'
| `-CompoundStmt 0x8074188 <col:41, line:3:1>
|   `-BinaryOperator 0x8074168 <line:2:3, col:11> 'int' lvalue '='
|     |-DeclRefExpr 0x80740b8 <col:3> 'int' lvalue ParmVar 0x8073f18 'c' 'int &'
|     `-BinaryOperator 0x8074148 <col:7, col:11> 'int' '+'
|       |-ImplicitCastExpr 0x8074118 <col:7> 'int' <LValueToRValue>
|       | `-DeclRefExpr 0x80740d8 <col:7> 'int' lvalue ParmVar 0x8073de8 'a' 'int'
|       `-ImplicitCastExpr 0x8074130 <col:11> 'int' <LValueToRValue>
|         `-DeclRefExpr 0x80740f8 <col:11> 'int' lvalue ParmVar 0x8073e68 'b' 'int'
```

Figure 2.5: AST for C/C++ function

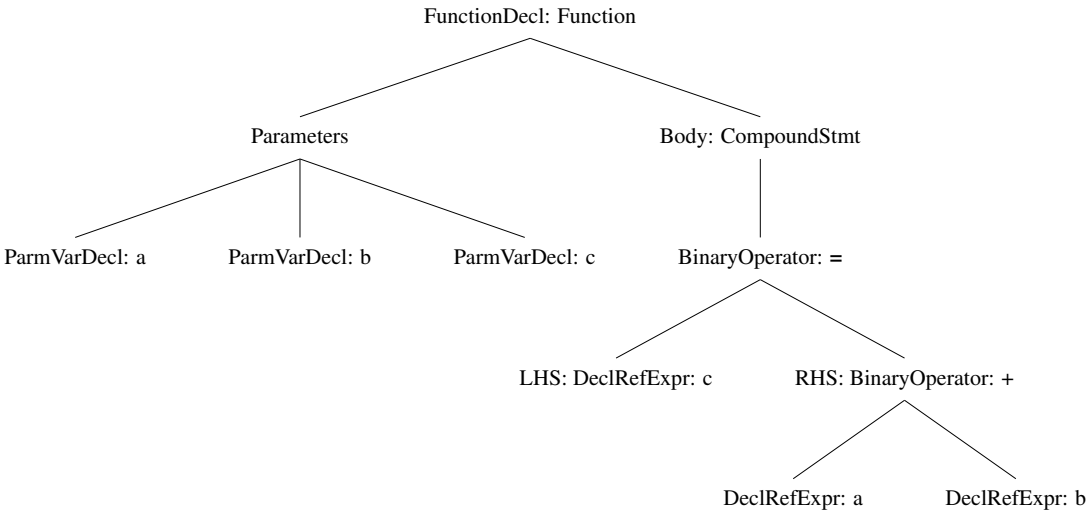

Figure 2.6: AST representation

In the AST, each node represents a specific construct or operation found in the source code. Termed as ”Node Types,” these nodes facilitate a refined and precise mechanism for the analysis, optimisation, and subsequent transformation of the underlying code. The following are some commonly used types in AST :

- *FunctionDecl*: The *FunctionDecl* node in Clang's Abstract Syntax Tree (AST) symbolises either a function declaration or a definition. Embedded within this node is crucial data such as the function's name, its return type, and an indicator of whether the function is defined. As soon as a function is defined, its *FunctionDecl* node will have child nodes that describe its parameters (through

ParmVarDecl nodes) and the body of the function, which is typically represented by a *CompoundStmt* node.

- *ParmVarDecl*: It is a dedicated node type in the AST signifying the declaration of parameter variables. Within a function or method declaration or definition, each parameter is represented using this node. Notably, *ParmVarDecl* encompasses detailed attributes of the parameter, including its type, name, and positional information within the function's signature.

- *CompoundStmt*: It pertains to a composite statement in the AST, embodying a block of code that envelops multiple other statements. In C and C++ languages, every code scope demarcated with curly braces ({}) gives rise to a *CompoundStmt* node in the AST. Its pivotal role lies in organising and containing other statement nodes.

- *VarDecl*: Representing a variable declaration, the *VarDecl* node can depict various forms of variables, be it global, local within a function, or even a class/structure member. This node encapsulates details such as the variable's name, its type, and potential initialises if they exist.

The Abstract Syntax Tree (AST), serving as an intermediate representation, intuitively captures the syntactic structure of a program. It not only offers efficient analysis capabilities but also encapsulates the entirety of the program's static structural information. Significantly, the AST is independent of the source language's syntax, making it an essential component in the latest versions of ESBMC.

2.6.2 Intermediate Representation (irept) in ESBMC

In model checking, AST and *irept* data structures are two distinct but related concepts, respectively. AST is a data structure widely used by compilers and static analysis tools to capture the core syntax of source code in a tree structure, eliminating unnecessary details such as parentheses. In contrast, *irept* was introduced in specific tools such as ESBMC and aims to provide a wider range of code description capabilities. They can not only describe traditional syntax elements, but also express complex code structures precisely because of their recursive and general nature. In these model checkers, *irept* is used as a highly flexible internal code representation that supports deep program analysis and verification. In short, while *irept* is similar to AST in some ways, it has features that make it more suitable for code analysis.

In the past, ESBMC used its own front-end to produce the *irept* structure [28]. This choice meant more work for the development team, as they had to handle its upkeep themselves. On the other hand, Clang provides a straightforward and effective way to create *irept* by using the AST it generates. This method is not only simpler but also more efficient, highlighting the benefits of using mature parsing tools in model checking. At present, ESBMC has gradually improved the C++ verification using Clang front-end [29].

2.7 Related Work

At present, many other tools have been proposed to verify CUDA programs. These tools use different methods, have their own characteristics, and aim at different property violations:

- *GPUVerify* [30]: It is a focused static analysis tool designed to check the correctness of GPU kernels, which are multi-threaded programs important for parallel computing. Using the ”dual scheduling” method, it looks into all thread interactions, searching for situations that might cause concurrency issues. Importantly, it supports both OpenCL and CUDA.

- *GKLEE* [31]: GKLEE builds upon KLEE, a tool designed for symbolic execution of LLVM intermediate code. It employs symbolic execution methods, enabling it to traverse every potential execution route in a program to uncover concealed mistakes. It has the ability to accurately simulate the concurrent execution of CUDA programs and thus find concurrency errors.

- *Prover of User GPU Programs* [32]: PUG is a tool that checks GPU kernels using automatic tools called SMT solvers. It can find issues like data race, problems with barrier synchronisation, and conflicts when using shared memory.

- *CIVL* [33]: CIVL stands for ”The Concurrency Intermediate Verification Language”. It is a model checking and verification tool specially designed for concurrent software. The main purpose is to model check and verify concurrent software to ensure the correctness of software in multi-threaded or parallel environment. Its technical principle is relatively close to that of ESBMC-GPU.

Chapter 3

Methodology & Implementation

This chapter aims to detail the method introduced in the project, which allowed the latest ESBMC to recognise extension name of CUDA programs and adaptation COM. The chapter further delves into the limitations of ESBMC in verifying C++ programs, explaining how these limitations impact CUDA verification. The steps taken to address these issues are also discussed. Finally, we examined issues in COM, rectified any incorrect code, and employed a hash table for operational model simplification. In the remaining part, this project analysed the principle of the convolution and activation function in CuDNN, and established the corresponding operational model.

3.1 Filename Extension

ESBMC provides support for a variety of programming languages. A Clang-based Frontend[1] will read the suffix of the filename to identify the programming language and use a frontend specific to that language to operate on it [34]. For example, when a C++ file *main.cpp* is given as input, the frontend recognises the suffix *.cpp* and uses Clang C++ frontend to parse and convert it. Similarly, it will output an error message if it recognises a file type that is not supported.

As mentioned before in this paper, CUDA program is an extension of C/C++. In order to distinguish CUDA source code from ordinary C++ code, CUDA programs use *.cu* as the file extension name. To verify CUDA programs in ESBMC, the solution is to deal with it as an extension of C++ programs, *i.e.*, use C++ frontend to process CUDA programs. Compared to C++, CUDA code has some unique symbols for acceleration and memory management, hence these symbols cannot be used in C++. This project

[1] `https://github.com/esbmc/esbmc/wiki/Clang-Based-Frontend-for-Cpp`

intends to improve the Clang C++ frontend with the ability to handle these special symbols, figure 3.1 shows the improved process.

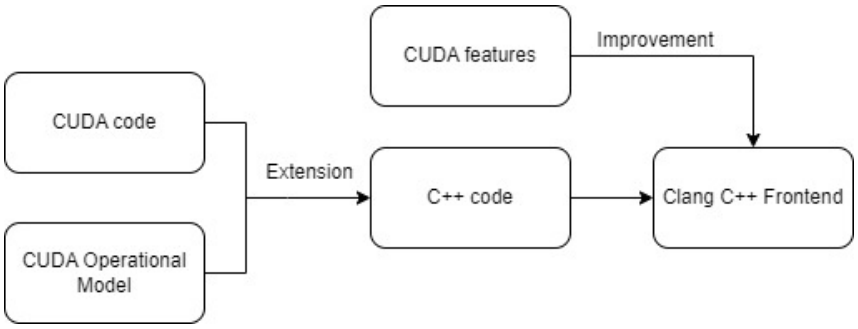

Figure 3.1: Clang frontend for CUDA program

C++ source file extensions exhibit diversity and are not universally standardised. Their names depending on factors like the compiler in use, the underlying operating system, and other environmental considerations. Common extensions, such as *.cc*, *.ipp*, and *.cxx*, manifest across different setups. Within the front-end of ESBMC, these extensions are meticulously defined, ensuring that files with matching extensions are put into to the C++ front-end for appropriate processing. As seen in Figure 3.1, we add the *.cu* extension to the list for C++ so that we can recognise CUDA programs.

Listing 3.1: C++ source file extensions

```
1 static const char *const extensions_cpp[] =
2   {"cpp", "cc", "cu", "ipp", "C", "cxx", nullptr};
3
4 static const language_desct language_desc_CPP = {"C++",
    extensions_cpp};
```

3.2 Clang-based Frontend Improvements

In this chapter we will show some improvements to the ESBMC C++ frontend. In order to port COM from older versions of ESBMC, we need to ensure that the existing frontend has the correct functionality and support for some of the operations involved in the model.

COM is developed based on the old ESBMC C++ compiler, which is called C++ parser. When input is C++ source code, the C++ parser first lexically analysis the

code, breaks it down and converts it into tokens, and finally generates an AST based on the tokens. However, the C++ language is constantly changing, and some syntax is updated and modified over time, so the C++ parser must be maintained frequently.

Ensuring correct parses is the basis of model checking. If the current frontend does not parse the code correctly, subsequent analysis and validation may parse based on incorrect premises, resulting in false positives or missing real errors. In our investigation, we found that the C++ front-end of ESBMC currently has some flaws in the handling of structs, causing some operations to fail, which is not acceptable, so we have made some fixes to these parsing and tuning of structs.

3.2.1 Struct Assignment

In CUDA, we used the struct data type to store 3D data, such as the index of the threads, the index of the blocks. Many of these require operations between structs. A struct is a compound data type that is a user-defined data type that contains multiple data members of different types. In contrast to C, structs in C++ can contain member functions, which can be user-defined. These member functions support basic struct operations such as initialisation, copy, move, and destruction. In general, we don't need to define all member functions explicitly; for simple structs, the compiler will often be able to generate them implicitly. Our C++ front-end lacks the ability to parse these implicit member functions. Here are some common features of member functions [35]:

- *Constructor*: A default constructor is a unique type of constructor which does not take any parameters. Typically, its role is to initialise member variables to default values. It is invoked when an object is instantiated without providing initial values, like when objects are created within arrays or through dynamic allocation. Notably, in scenarios where no constructors are explicitly defined for a class or structure, the compiler implicitly provides a default constructor, which generally performs no specific actions [36].

- *Copy Constructor*: The copy constructor receives a reference to an object of the same type and utilises it to initialise a new object. The utilisation of object passing as a function parameter, object return from a function, or object initialisation based on an existing object is particularly advantageous. The copy constructor, if not explicitly modified, executes a "shallow copy" operation.

- *Copy Assignment Operator*: The copy assignment operator is an overloaded version of the '=' operator, designed to assign the state of one object to another of the same type. Its primary utility surfaces when there's a need to assign values between pre-existing objects. Mirroring the behaviour of the default copy constructor, this operator performs a shallow copy by default. Such behaviour might not always be desirable, especially when handling dynamically allocated resources.

- *Destructor*: A destructor, distinguished by the tilde (˜) symbol preceding the class name, neither takes parameters nor returns a value. Its main purpose is to perform cleanup actions when an object goes out of scope or is explicitly deleted. In cases where the class manages dynamically allocated resources or other system resources like file handles, proper management in the destructor becomes crucial. The default destructor, if not defined explicitly, usually does nothing specific.

- *Move Constructor and Move Assignment Operator*: Introduced in C++ 11, both the move constructor and move assignment operator bolster the language's efficiency through "move semantics". Invoked primarily when initialising or assigning from a temporary object (often termed as an right value), they prevent unnecessary copying, leading to optimisations. If certain criteria are met, the compiler can implicitly provide default versions of these functions, which typically involve moving members individually.

Initially, the Clang frontend autonomously generates constructors, destructors, as well as copy/move assignment operators for the struct. This intrinsic generation substantially alleviates our workload, necessitating only an accurate extraction from the AST, as shown in Figure 3.2.

```
|-CXXRecordDecl 0x731ea90 <main2.cpp:3:1, line:6:1> line:3:8 referenced struct uint3 definition
| |-DefinitionData pass_in_registers aggregate standard_layout trivially_copyable pod trivial literal
| | |-DefaultConstructor exists trivial
| | |-CopyConstructor simple trivial has_const_param implicit_has_const_param
| | |-MoveConstructor exists simple trivial needs_implicit
| | |-CopyAssignment simple trivial has_const_param needs_implicit implicit_has_const_param
| | |-MoveAssignment exists simple trivial needs_implicit
| | `-Destructor simple irrelevant trivial needs_implicit
| |-CXXRecordDecl 0x731ebb0 <col:1, col:8> col:8 implicit struct uint3
| |-FieldDecl 0x7346300 <line:5:3, col:16> col:16 x 'unsigned int'
| |-FieldDecl 0x7346368 <col:3, col:19> col:19 y 'unsigned int'
| |-FieldDecl 0x73463d0 <col:3, col:22> col:22 z 'unsigned int'
```

Figure 3.2: AST for implicit member functions

By using the corresponding query function provided by `clang::CXXMethodDecl`, we can easily determine whether a method has a particular property or feature. We propose a new function `is_CopyOrMoveOperator`, it primary objective is to ascertain if a given C++ method, represented by the parameter *md* of type `clang::CXXMethodDecl`, is either a copy assignment operator or a move assignment operator. The method under scrutiny leverages two pivotal utility functions from `clang::CXXMethodDecl`:

1. *isCopyAssignmentOperator*(): Checks if the method *md* is a copy assignment operator. In C++, this operator is automatically created for classes if needed. Its main job is to let one object take the values from another object of the same type.

2. *isMoveAssignmentOperator*(): Confirms if the method *md* is a move assignment operator. Introduced in C++ 11, this operator is adept at efficiently reassigning resources from one object to another, resulting in the source object being left in a valid but unspecified state.

Listing 3.2: Determine copy/move assignment operator method

```
1 bool clang_cpp_convertert::is_CopyOrMoveOperator(const
      clang::CXXMethodDecl &md)
2 {
3   return md.isCopyAssignmentOperator() || md.
        isMoveAssignmentOperator();
4 }
```

The implicitly called member functions are then converted into symbols by improving the conversion functions in the C++ frontend. In the Clang C++ frontend, we usually inherit methods from the Clang C frontend. This is because C++ is almost fully compatible with C, which means most C programs can be compiled and run in the C++ compiler, so our frontend architecture enables code reuse. In this case, the C++ struct has a few more member functions than the C struct, so we just need to make some filtering and optimisations on the C++ front end, and then call the C conversion function on the front end to implement the desired functionality.

In the C++ frontend, the `get_method` function selectively omits certain implicit functions produced by Clang. This happens because Clang's AST creates numerous methods that aren't essential for model checking. Although this process streamlines the symbol table, it occasionally skips over certain implicit functions we require. To address this, we employ the function shown in Listing 3.2 to identify if an implicit

function is a copy or move assignment operator, This helps us convert AST Implicitly generated member functions into symbols.

It's important to mention that when Clang automatically creates member functions, it often leaves the parameters unnamed. This is done to prevent any name clashes with user-defined code and to make the internal processing simpler. While naming parameters can streamline operations, in ESBMC, we rely on these parameter names to build the symbol table. If the parameters are unnamed, we can't call the member function correctly. To address this, we introduced a function that assigns names to these parameters, algorithm 5 describes the behaviour of this function.

Algorithm 5 Parameter named function

Function: *name_param_and_continue*

1: Pre-condition Assertion: The name and id are empty
2: If the function is automatically created by the compiler and is a member function, we give its arguments a name by combining the base name with the string "ref"
3: Returns if it is not an implicit member function generated by the compiler

At this point, the ESBMC frontend is able to correctly parse the member functions generated by the Clang frontend and convert them to a symbol table. We can see the converted member functions under the flag *–symbol-table-only* of ESBMC. Figure a shows the copy assignment operator generated according to the above AST, as evident from the symbol table, the parameters have been appropriately assigned names.

```
Symbol......: c:@S@uint3@F@operator=#&1$@S@uint3#
Module......: main2
Base name...: operator=
Mode........: C++ (1)
Type........: struct uint3 & (struct uint3 *, const struct uint3 &)
Value.......:
{
this->x = operator=::ref->x;

this->y = operator=::ref->y;

this->z = operator=::ref->z;

return *this;

}
Flags.......: lvalue
Location....: file main2.cpp line 3 column 8
```

Figure 3.3: Symbol table of the Copy Assignment Operator

3.2.2 Temporary Variables

In our verification approach for a CUDA program, we first call the specific CUDA kernel function. This function then sets the *gridDim* and *blockDim* values based on the given arguments. Nevertheless, in the context of ESBMC's Clang C++ frontend, this specific situation presents challenges. The frontend doesn't effectively address this scenario, it is currently not equipped to effortlessly generate a temporary variable and subsequently assign it a value through a constructor call.

$$gridDim = dim3(blocks); \tag{3.1}$$

$$blockDim = dim3(threads); \tag{3.2}$$

In the above method calls, *dim*3 is a struct in CUDA that is mainly used to specify the size of three dimensions. While many CUDA applications only parallels in one or two dimensions, *dim*3 provides three dimensions: x, y, and z to support a wider range of use cases. Although *dim*3 has three dimensions, it is not necessary to specify the values of the three dimensions at definition time, this is achieved by adding default values in the constructor. listing a shows a simplified version of the dim3 definition.

Listing 3.3: Example struct

```
1  struct dim3
2  {
3    unsigned int x, y, z;
4
5    dim3(unsigned int vx = 1, unsigned int vy = 1, unsigned
         int vz = 1)
6    {
7      x = vx;
8      y = vy;
9      z = vz;
10   }
11 }
```

In C++, when an instance of the `dim3` type is instantiated via the call `dim3(blocks)`, the compiler undergoes a series of operations to facilitate this request [36]:

1. **Resolution and Matching of the Constructor**: The compiler initiates by examining available constructors and discerning the optimal match. In scenarios

with multiple constructors, the compiler undertakes an overloading resolution to ascertain the most fitting candidate.

2. **Storage Allocation**: Subsequent to constructor resolution, the compiler allocates memory for the new object. For local instances (e.g., those defined within a function), this typically corresponds to stack allocation. Conversely, for dynamically allocated objects, memory is sourced from the heap.

3. **Constructor Invocation**: The compiler subsequently emits instructions to invoke the matched constructor, which encompasses:

   - *Parameter Passing*: The compiler ensures the parameters are conveyed in the correct sequence, filling in absent parameters with default values. In the current context, a value of *blocks* is supplied for `vx`, while `vy` and `vz` inherit the default value of 1.
   - *Execution of the Constructor Body*: The code within the constructor executes, initialising the object's member variables.

4. **Handling of Temporary Objects**: In specific instances, temporary objects might be instantiated and subsequently destructed within this process.

In our analysis, we adapt in the goto program, which is an important intermediate level of the ESBMC source code. This design aims to streamline the subsequent phases of analysis and model checking. Firstly, the mentioned operations involve invoking the constructor within the structure. In ESBMC, we refer to such operations as side effects. These are statements or expressions that might introduce unintended changes, like function calls, assignments, or other actions that could modify the program's state. Recognising and addressing these side effects is pivotal for ensuring accurate checks. Typically, ESBMC would convert these expressions, which could have side effects, into a set of equivalent atomic operations that are side-effect free.

Initially, a temporary variable is instantiated to store the return value. This variable is assigned the return type of the struct dim3, and its location within the expression of the function call. A temporary variable named &*return_value*\$*_dim*3\$1 is created by concatenating the string *return_value*\$ and the struct name *dim*3.

Listing 3.4: goto program side effect handling

```
new_symbol.name = "return_value$";
new_symbol.type = expr.type();
new_symbol.location = expr.location();

new_symbol.id = tmp_symbol.prefix + id2string(new_symbol.
    name);
new_name(new_symbol);

goto_programt tmp_program;
const typet &ftype = call.function().type();
if(ftype.return_type().id() == "constructor")
{
  // for constructor, we need to add the implicit `this` as
      the first argument,
  // so convert to:
  // BLAH(&return_value$_BLAH$1, ...)
  side_effect_expr_function_callt ctor_call;
  ctor_call.function() = call.function();
  exprt::operandst &args = ctor_call.arguments();
  address_of_exprt tmp_result = address_of_exprt(call.lhs()
      );
  // first push the implicit `this` arg
  args.push_back(tmp_result);
  // then append the remaining operands
  args.insert(args.end(), call.arguments().begin(), call.
      arguments().end());
  ctor_call.location() = call.location();
  // now convert this expr to code
  codet ctor_call_code("expression");
  ctor_call_code.location() = call.location();
  ctor_call_code.move_to_operands(ctor_call);

  convert(ctor_call_code, tmp_program);
}
```

Finally, it gets the function type and checks if the function is a constructor. If it is a constructor, we need to take a special treatment. For constructor calls, we need to pass the implicit temporary variable pointer as the first argument and then add the previously initialised arguments to the argument list. At this point, the temporary variable *&return_value$_dim3$1* has been successfully assigned. Figure 3.4 is the final converted goto program that can be viewed using flag *–goto-functions-only* in

ESBMC.

```
main (c:@F@main#):
        // 37 file main3.cpp line 17 column 13 function main
        dim3 return_value$_dim3$1;
        // 38
        FUNCTION_CALL:  dim3(&return_value$_dim3$1, ( dim3 *)1, 1, 1)
        // 39 file main3.cpp line 17 column 11 function main
        FUNCTION_CALL:  operator=(&gridDIM, &return_value$_dim3$1)
        // 40 file main3.cpp line 18 column 3 function main
        RETURN: 0
        // 41 file main3.cpp line 19 column 1 function main
        END_FUNCTION // main
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Figure 3.4: Goto program for Temporary variables

3.3 Operational Model Fix

The Clang-based frontend improves efficiency for ESBMC and also provides support for newer C++ standards such as C++ 11 [37]. This brings new C++ features, but also errors in existing operational models, therefore, an important goal of the project was to fix and improve COM. In this process, we cannot simply repair OM randomly, and random repair is likely to lead to more difficult and confusing repair in the future. This repair plan follows the *Guidelines for Fixing OM* [2].

1. Start with a simple CUDA program and get its dependencies in COM.
2. Get the existing errors from all dependencies and fix them from the inside out according to the dependencies.
3. After the repair is completed, switch to a more complex CUDA program and repeat the operation of step 1.

Importantly, the repair of the model must preserve the original behaviour. For example, the C++ $< new >$ header files provide support for dynamic memory allocation by defining the operators new and delete and the associated exception handling mechanism. When a program needs to request or free memory at run time, this header ensures efficient resource management and proper exception handling, all of which are implemented by the $< new >$ operation model in ESBMC. In the detection through

[2] https://github.com/esbmc/esbmc/wiki/Guidelines-for-Fixing-Operational-Models-(OM)-in-ESBMC

the guidelines step, it is found that when we use the model, there will be a segmented fault. After investigation, it is found that the problem occurs in the exception handling function, as shown in Figure 3.5.

Listing 3.5: Outdated model

```
1  struct bad_alloc: public exception {
2  public:
3          bad_alloc() throw ();
4          bad_alloc(const bad_alloc&) throw ();
5          bad_alloc& operator=(const bad_alloc&) throw ();
6          const char* what() const throw () :
7                          message("std::bad_alloc") {
8                  return message;
9          }
10 };
```

Member initialises are frequently employed in constructors to initialise or invoke the base class constructor and to initialise the member variables of the class. In the preceding code snippet, the member initialisation list (*message*("*std* :: *bad_alloc*")) is used in the definition of a member function, which is not allowed in C++, For these outdated syntax, we follow the authoritative online C++ reference site[3], which provides a detailed, accurate list of components and functions in the C++ standard library. We have made some corrections to some erroneous models to ensure that they work properly, listing 3.6 shows the corrected code. We need to gradually update these outdated code usage to fit the Clang frontend.

Listing 3.6: The corrected model

```
1  struct bad_alloc: public exception {
2  public:
3    bad_alloc() throw() {}
4    bad_alloc(const bad_alloc& other) throw();
5    bad_alloc& operator=(const bad_alloc& other) throw();
6    virtual const char* what() const throw() {
7                  return "std::bad_alloc";
8          }
9  };
```

In COM, several methods employ *malloc* to allocate memory on the heap. This approach can be laborious, as memory assigned through *malloc* requires a subsequent

[3] https://en.cppreference.com/w/

Free function call to avoid potential memory leaks. Conversely, *alloca* is a more straightforward option for local variables, offering swift allocation and an automatic release after the function call concludes. In the latest version ESBMC provides an intrinsic function __ESBMC_alloca, it ensures consistent and valid memory allocation. Specifically, it was introduced to provide users with a mechanism to initialise byte arrays or simulate register addresses within the harness. In addition, it does not trigger any memory leaks or dangling pointer checks.

Listing 3.7: intrinsic function example

```
1  /*ESBMC_verify_kernel()*/
2  void ESBMC_verify_kernel_no_params(void *(*kernel)(), int
       blocks, int threads)
3  {
4    __ESBMC_atomic_begin();
5    threads_id = (pthread_t *)__ESBMC_alloca(GPU_threads *
         sizeof(pthread_t));
6
7    dev_no_params.func = kernel;
8
9    int i = 0, tmp;
10   assignIndexes();
11   while(i < GPU_threads)
12   {
13     pthread_create(&threads_id[i], NULL,
           ESBMC_execute_kernel_no_params, NULL);
14     i++;
15   }
16   __ESBMC_atomic_end();
17 }
```

In Figure 3.7, the code aims to dynamically allocate space on the stack for a array to store thread IDs. The array's size is determined by the value of *GPU_threads* . The function __ESBMC_alloca is utilised for this allocation. It streamlines the process and, by ensuring that memory leaks or dangling pointers won't occur, allows us to skip those specific checks, thereby enhancing verification efficiency.

3.4 Optimisation of COM

When ESBMC checks CUDA programs, it starts by preprocessing the code before compiling it. During this preprocessing, it recognises the "#include" directive and

inserts the content of the specified file into the main code. This action links them together. For instance, if a function from COM is used in a CUDA program, the preprocessing ensures that its definition is available from COM. To sum up, when we validate a program with ESBMC, we are not only validating the behaviour of the code, but also the used functions in the header file.

In CUDA's parallel framework, determining the index of blocks and threads is essential [38]. Each thread needs to handle a unique segment of data, and its index determines its position within the dataset, by using the index, we can specify the work part of each thread. During the calculation, the linear value is an intermediate variable used to convert the multidimensional index to a single dimension, First, we calculate the index of the block, which is calculated by the following formula:

$$\text{linear\_value} = \frac{id}{\text{blockDim.x} \times \text{blockDim.y} \times \text{blockDim.z}} \tag{3.3}$$

By converting the linear value to a multidimensional index, the x, y and z of the thread or block index can be calculated:

$$\text{block\_index}.z = \left\lfloor \frac{\text{linear\_value}}{\text{gridDim.x} \times \text{gridDim.y}} \right\rfloor \tag{3.4}$$

$$\text{block\_index}.y = \left\lfloor \frac{\text{linear\_value mod } (\text{gridDim.x} \times \text{gridDim.y})}{\text{gridDim.x}} \right\rfloor \tag{3.5}$$

$$\begin{aligned} \text{block\_index}.x = \text{linear\_value mod } &(\text{gridDim.x} \times \text{gridDim.y}) \\ &\text{mod gridDim.x} \end{aligned} \tag{3.6}$$

After the block index is obtained, the thread index is calculated. Given a linear thread ID, the goal is to derive its corresponding 3D thread index within the CUDA computational model considering both the block and grid structures. It can be defined as follows:

$$\text{block\_size} = \text{blockDim.x} \times \text{blockDim.y} \times \text{blockDim.z} \tag{3.7}$$

$$\begin{aligned} \text{grid\_position} = &\text{ indexOfBlock}[id].x \\ &+ \text{indexOfBlock}[id].y \times \text{gridDim.x} \\ &+ \text{indexOfBlock}[id].z \times \text{gridDim.x} \times \text{gridDim.y} \end{aligned} \tag{3.8}$$

From these, the linear position of the thread within its block is:

$$\text{linear\_value} = id - \text{grid\_position} \times \text{block\_size} \tag{3.9}$$

With this linear value, the 3D index of threads is derived as:

$$\text{thread\_index}.z = \left\lfloor \frac{\text{linear\_value}}{\text{blockDim.x} \times \text{blockDim.y}} \right\rfloor \tag{3.10}$$

$$\text{thread\_index}.y = \left\lfloor \frac{\text{linear\_value mod } (\text{blockDim.x} \times \text{blockDim.y})}{\text{blockDim.x}} \right\rfloor \tag{3.11}$$

$$\begin{aligned} \text{thread\_index}.x = \text{linear\_value mod } &(\text{blockDim.x} \times \text{blockDim.y}) \\ &\text{mod blockDim.x} \end{aligned} \tag{3.12}$$

In typical programs, index calculation can be a tedious process. However, for most programs, this calculation might not significantly affect performance, as the value becomes clear during execution. But for ESBMC, it's not about a specific run. Instead, ESBMC generates a constraint to express the semantics of the calculation, considering all possible runs. This is because ESBMC aims to identify inputs and paths that could lead to errors in the program. For our purposes, we focus on verifying the correctness of the CUDA program itself, and for the computation of COM, we need to make the maximum simplification to improve the efficiency of software verification.

In the verification process of most CUDA programs, two-thread analysis can find errors. For CUDA architecture, two-thread can be divided into two cases, namely, a thread block containing two threads is enabled, and the corresponding kernel function is $kernel <<< 1,2 >>> ()$, in the other case, two thread blocks are enabled, each with a single thread, and the corresponding kernel function is $kernel <<< 2,1 >>> ()$. This gives us a finite range of block and thread indices, which we can plug into the above formula to obtain the mathematical model. In order to simplify the complex model, we create the following lookup table 3.1. The purpose is that when we output a value, the result will be immediately obtained through the lookup table, optimising the time complexity of this calculation to $O(1)$.

id	gridDim	blockDim	getBlockIdx	getThreadIdx
0	{1, 1, 1}	{2, 1, 1}	{0, 0, 0}	{0, 0, 0}
1	{1, 1, 1}	{2, 1, 1}	{0, 0, 0}	{1, 0, 0}
0	{2, 1, 1}	{1, 1, 1}	{0, 0, 0}	{0, 0, 0}
1	{2, 1, 1}	{1, 1, 1}	{1, 0, 0}	{0, 0, 0}

Table 3.1: The lookup table for the index

3.5 Modelling CuDNN

This subsection provides an in-depth examination of the techniques within CuDNN. Initially, we study the acceleration of convolution, employing images and matrices for clarity. We then replicate the convolution operation in CuDNN through simulation. Following this, we delve into its activation function, deriving an approximate version using a hash table. Finally, we compared the precision selection of the sigmoid approximation function and selected the appropriate precision .

3.5.1 Convolution in CuDNN

Convolution is widely used not only in neural networks, but also in operations such as polynomial multiplication, which works by convolving two discrete series. While in neural networks, it is used as edge detection of images, image blurring, sharpening, etc. In the field of image processing, this convolution is also called two-dimensional discrete convolution, and its naive method explains the principle and process of convolution in neural networks well [39]:

1. **Iterating Through the Input**: The outer loops traverse the width W and height H of the input data, allowing the kernel or filter to cover each part of the input.

2. **Filter Application**: The next two loops with variables x and y iterate through the filter's width and height K, respectively. This process helps align the filter with the input at different positions.

3. **Multiple Filters Application**: The subsequent loop with variable m iterates through M different filters, allowing the algorithm to learn multiple features from the input.

4. **Depth Iteration**: The innermost loop with variable d iterates through the depth or channels D of the input, applying the filter across all channels of the input.

5. **Convolution Computation**: Inside all these loops, the algorithm multiplies the corresponding elements of the input and filter and adds them to the output at the corresponding position w, h, m. This forms the convolution between the input and filter at that position.

In summary, the algorithm systematically applies a kernel across the input data's width, height, and depth, taking the element-wise product and summing the results to form the convolution. The output represents a transformed version of the input, with features detected by the filters. This naive convolution algorithm, designed in the CPU, is based on sequential execution when it is serial. However, GPU is dominated by parallel computing. In GPU-accelerated libraries such as CuDNN, traditional convolution methods cannot be paralleled, resulting in low execution efficiency on GPU. Therefore, it is necessary to design a new convolution algorithm to make GPU take full advantage of its parallelism to improve the efficiency of convolution operation and reduce the training time of neural network [40].

In CuDNN, normalised matrix multiplication is employed for convolution. This process breaks down the convolution operation into two steps. First, the *im2col* method is used to reorganise the image blocks, aligning each row sequentially, this computation additionally preserves the structure of its matrix:

$$M = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \tag{3.13}$$

$$\text{im2col}(M) = [1, 2, 3, 4] \tag{3.14}$$

Specifically, for kernels and images, *im2col* also performs additional operations based on the number of steps, padding, and so on of the convolution kernel. Given an image of dimensions $H \times W$, it can be transformed into a matrix of size $K \times N$. The value of N denotes the number of features extracted by sliding the convolution kernel across the input image. It is mathematically expressed as:

$$N = \left(\frac{H + 2 \times \text{Pad} - k}{\text{stride}} + 1 \right) \times \left(\frac{W + 2 \times \text{Pad} - k}{\text{stride}} + 1 \right) \tag{3.15}$$

Where:

- Pad is the padding value for both height and width of the image.
- k denotes the size of the square convolution kernel.

- stride is the stride value in both height and width directions.

The K rows in the resultant matrix symbolize the data associated with each $k \times k$ patch in the image, as delineated by the convolution kernel. The kernel is much easier to manipulate. After the *im2col* operation, the kernel is flattened, i.e., from a 2-D k x k shape to a 1-D vector of length k times k. To make it easier to understand, we'll use a matrix to represent this process:

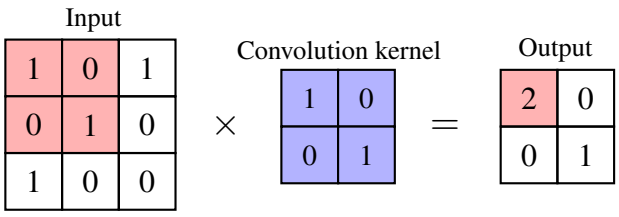

Figure 3.5: Convolution example

As demonstrated in the example provided (see 3.5), there exists a 3x3 input matrix and a 2x2 kernel. To perform the convolution operation, we slide the kernel over the input starting from the top left corner of the input matrix. At each discrete position of the sliding window, the convolution kernel is element-wise multiplied with the corresponding elements of the input matrix. The resulting products are then summed together to obtain a single value. The sum value is incorporated as an element within the resulting matrix. Initially, the kernel encompasses the 2x2 region situated in the upper-left corner of the input matrix. The element-wise multiplication and summation operations as previously described are executed, resulting in the extraction of the initial element from the output matrix. Subsequently, the kernel is shifted one position to the right, followed by the execution of element-wise multiplication and summation, resulting in the derivation of the subsequent element of the output matrix. As the process of shifting the kernel persists, it gradually encompasses all conceivable 2x2 regions within the input matrix. Ultimately, the outcome is a 2x2 output matrix that accurately represents the comprehensive convolution result.

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 0 \\ 1 \end{bmatrix} \rightarrow \text{resize to } 2 \times 2 \Rightarrow \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \tag{3.16}$$

The matrices in equation 3.16 originate from above foundational convolution operation, When compared to the convolution operation above, they give the same result. Specifically:

- The 4×4 matrix represents the input image transformed by the *im2col* process. This technique reshapes overlapping patches of the input image into columns, producing the said matrix.

- The 3×1 vector mirrors the flattened convolution kernel, designed to align with the columns of the transformed input.

Multiplying these matrices effectively replicates the convolutional process. After obtaining two large matrices, all convolution calculations can be done at once by just multiplying these two matrices. In this way, the calculation speed of convolution operation is greatly improved in CuDNN. It should be noted that CuDNN is not open source and its internal implementation is not publicly available, therefore, we have modeled its internal convolutional behaviour through existing literature and the formula already method mentioned above, simulating the operation of $im2col$.

3.5.2 Activation Function

In neural networks and other machine learning models, the Sigmoid function is often employed as a non-linear activation function, which transforms real numbers into values between 0 and 1. However, calculating this function involves exponential operations which can be time-consuming, especially in scenarios like hardware implementations or real-time computations.

To address this challenge, the lookup table (LUT) method has been introduced for discretizing the Sigmoid function [41]. A lookup table is a pre-computed data structure that stores the function's output values within a specific range. This allows for quick retrieval of results without the need for intricate real-time calculations. In this paper, the sigmoid function is modeled and the lookup table is implemented:

1. *Range*: The characteristic of the Sigmoid function is that the output of the function tends to 1 or 0 when the input value is large or small, so it does not make sense to add additional calculation points in these regions. Typically, when x is in the range [-5, 5], the output of the Sigmoid function will cover most of the range from 0 to 1, making [-5, 5] a commonly used range. This range can be adjusted according to the actual application requirements. If we know that most input values will be concentrated in a particular region, we can narrow or expand that range. Therefore, in order to improve the robustness, a large range is safe.

2. *Precision*: The quantity of data points in the lookup table is determined by this factor. A greater number of data points in the lookup table leads to a higher level of precision, resulting in a more precise approximation. Using 0.01 as an example, this means that we precompute the *Sigmoid* at every 0.01 interval in the range [-5, 5]. So there will be 1,000 data points (from -5.00, -4.99,... , 4.98, 4.99, 5.00). Note that while increasing the precision of the lookup table increases the accuracy of the approximation, it also increases the space required to store the lookup table and the time required to initialise the lookup table.

3. *Computation*: Commences the process of traversing the complete range of definitions with the specified level of precision. The *Sigmoid* function is employed to compute the corresponding output value for each value of x within the given range. The resulting value is subsequently stored within a lookup table. For example, when x = 0, calculate $Sigmoid(0)$ and store the result.

4. *Reference*: During the run of the model, when the Sigmoid value of some input x needs to be calculated, instead of computing the whole function, we directly find the output corresponding to the closest value of x in the lookup table.

Algorithm 6 Sigmoid Look Up Table

Function: $sigmoidLUT(float\ u)$

1: Converts u to index using the specified precision
2: If the index is less than the table range, *Result* is 0
3: If the index is greater than the table range, *Result* is 1
4: If the index is within the range, the *Result* value is lookup[index], where $lookup$ is the pre-generated lookup table

Output: Result

For each input value in the range, we have the output of the original Sigmoid and our discrete approximation. The error is the difference between these two:

$$\text{Error} = \text{Original Sigmoid Output} - \text{Discrete Approximation} \tag{3.17}$$

We can observe the error brought by different accuracies in a limited range. As can be seen in Figure 3.6, when the accuracy = 0.01, the error with the original sigmoid function is almost negligible. Therefore, we finally chose a precision of 0.01 to build the lookup table.

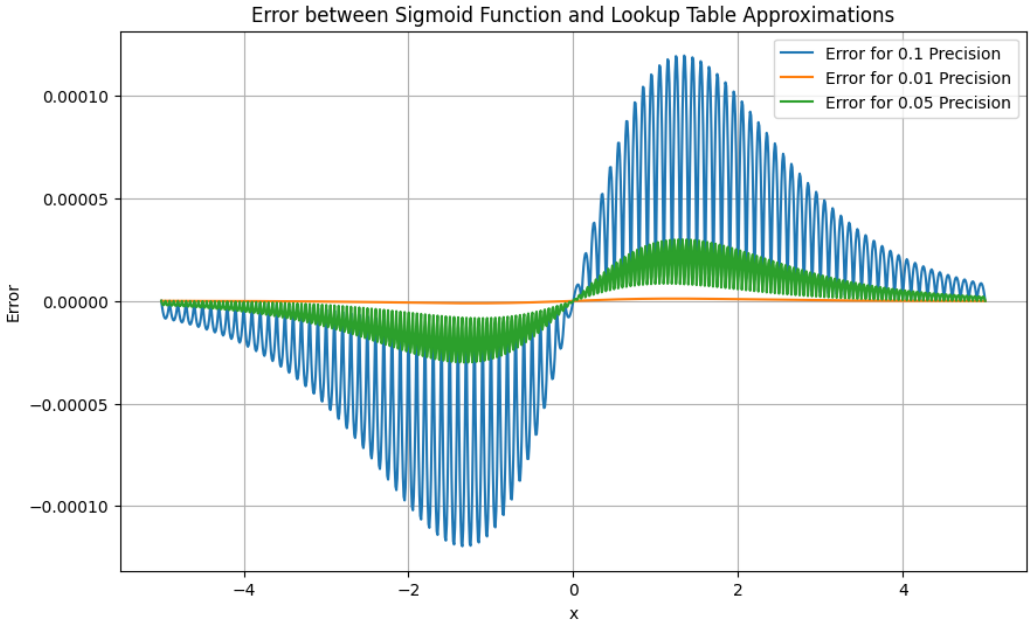

Figure 3.6: Precision comparison

Chapter 4

Evaluation

In this chapter, we first introduce the benchmarks used, focusing on different types of test cases, and then we compare them with other state-of-the-art CUDA verification frameworks. For the COM simplification method we discussed earlier, we evaluate all compatible test cases to contrast the verification duration before and after the simplification. Details of the test environment are provided follow:

- Environment: Windows Subsystem for Linux (WSL2)
- CPU: Intel i7-9750h, 6 core
- RAM: 16 GB
- Operating System: Ubuntu 22.04.2 LTS
- ESBMC Verision: v7.3

4.1 Benchmarks

In this project, we've incorporated a benchmark suite to evaluate fundamental functions typically utilised in genuine CUDA applications from the research on ESBMC-GPU [3]. Our chosen benchmarks include the NVIDIA SDK v2.0, which contains a collection of 20 CUDA kernels, another 20 from Microsoft C++ Projects, and an additional 104 CUDA programs reflecting a broad spectrum of CUDA capabilities. Notably, these benchmarks have been employed in prior studies to gauge the efficiency and accuracy of various GPU verifiers [3]. We will cover these different types of Test Cases (TC) in the benchmark.

4.1.1 TC type: Data Race

The occurrence of data races is a prevalent issue in parallel programmes. Concurrency occurs when multiple threads concurrently access a shared memory location, resulting in potential conflicts when one or more threads attempt to modify the memory location. If the programmer does not properly synchronise the operations of the threads, unpredictable read and write results can occur. In CUDA, threads run in parallel in the core, so a CUDA application may have thousands of threads running simultaneously. In general, CUDA brings powerful parallel capabilities to developers, but it also brings concurrency problem - data races.

Listing 4.1: Data Race Example

```
1 __global__ void dataRace(int *sum) {
2     // Every thread tries to increment the same memory
          location
3     sum[0] += 1;
4 }
```

In this program fragment, we employ a global variable named *sum*. As the code runs, each thread executes this function. Assuming an initial value of zero for *sum*, the ideal outcome should equate *sum* to the total number of threads. However, due to a lack of synchronisation in the code, each thread might perceive a different value for *sum* as they all attempt to increment it simultaneously. This discrepancy arises from a phenomenon known as data race, leading to the possibility that the final *sum* might not equal to the thread count.

Flag *–data-races-check* in ESBMC is used to check for data races, which will add assertions when verifying the program. When multiple threads reads and writes to the same memory, the assertion will be violated to indicate that a data race has occurred. Upon identifying a data race, ESBMC promptly provides a counterexample pinpointing the location of the race, as illustrated in Figure 4.1. Here, ”w/w” denotes simultaneous writes to an identical memory spot across multiple threads. At the end, accompanied by the variable’s name that was written.

In the benchmarks of this project, we added the inclusion of various types of data races, such as computation, accumulation, etc. By using indexes in COM, we can have different threads access the same memory address to reproduce the data race problem.

```
[Counterexample]


State 3 file datar.cu line 9 column 5 function dataRace thread 1
-------------------------------------------------
Violated property:
  file datar.cu line 9 column 5 function dataRace
  W/W data race on c:@sum
  !tmp_c:@sum


VERIFICATION FAILED
```

Figure 4.1: Data race check results

4.1.2 TC type: Constant Memory

In CUDA programming, constant memory is frequently utilised. This memory type is defined using the *__constant__* modifier, signifying its read-only nature—writing to it is prohibited. Designed for speed, the constant cache ensures quicker and more streamlined data reads when many threads access the same data simultaneously. However, constant memory is only available for data that does not change during the entire execution of the kernel. As a result, problems can arise when trying to write to variables that are in constant memory.

Listing 4.2: Constant Memory Example

```
1 __constant__ int A[N] = {0, 1};
2
3 __global__ void foo(int *B) {
4         A[threadIdx.x] = 1;
5 }
```

The provided code 4.2 defines an array A in constant memory with the *__constant__* modifier and initialises it with the values 0, 1. The function foo is a *__global__* kernel set to run on the GPU. Within this function, there's an attempt to modify an element of the array *A*. In essence, this code tries to alter a read-only constant memory space, which might result in unpredictable outcomes or even a program failure.

The benchmarks include some of scenarios showcasing the use of Constant Memory in CUDA. For instance, there are cases where numerous threads simultaneously access the same data — a scenario where Constant Memory, with its optimised broadcasting capabilities. Another common use demonstrated is static lookup tables; these

are predefined tables where the values serve as references or conversions during computations. It's crucial to note that within these benchmarks, the values stored in Constant Memory remain consistent and unaltered throughout the duration of the kernel's execution.

4.1.3 TC type: Null Pointer

The null pointer issue is a frequent challenge in CUDA programming. This problem typically arises when a program attempts to access a memory address that hasn't been initialised or has already been released. For example, if a developer forgets to allocate memory for a variable with *cudaMalloc* but tries to use that pointer, it will become a null pointer, or if a developer has freed memory with *cudaFree* but tries to use that memory afterwards, this is also a problem called a dangling pointer.

As shown in code 4.3, inside the kernel function, an integer pointer is initialised to *nullptr*, which means that it does not point to any valid memory address. At the end of the code, trying to use the *ptr* pointer to access and modify memory, this line of code will result in an illegal memory access.

Listing 4.3: Null Pointer Example

```
1 __global__ void nPointer() {
2     int *ptr = nullptr;
3     int idx = threadIdx.x;
4     ptr[idx] = idx;
5         // Null Pointer
6 }
```

In a CUDA program, when a null pointer issue arises, it typically doesn't trigger an immediate error notification. This can lead the program to keep running until it encounters another error or completes its execution. As a result, developers frequently rely on additional memory detection tools to identify runtime problems such as null pointer references. The benchmark includes various instances of null pointer usage in its test cases, with the goal of assessing ESBMC's capability to detect null pointers in CUDA programs.

4.1.4 TC type: Unit Testing

Unit testing is the method used in this project to verify the correctness of a particular function or method in COM. For CUDA programs, unit testing is an important part, because the characteristics of GPU parallel computing makes it difficult to detect errors. On the other hand, it is necessary to ensure that the behaviour of parallel functions simulated in COM is accurate.

Listing 4.4: Parallel computing example

```
1 __global__ void Sums(int *a, int *b, int *c){
2     c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
3 }
```

In the code provided, our aim is to test the Sums function. This function adds elements from arrays a and b and saves the results in array *C*. Given the challenges in directly observing parallel processes, we choose to use a serial version of the function to generate a reference output for the CUDA parallel version. By applying identical input data to both versions, we can generate output for cross-validation. By comparing the CUDA version's results to the serial version's, if they align, it confirms the accuracy of our parallel implementation in the CUDA environment.

Listing 4.5: Verification method

```
1 for (int i = 0; i < N; i++) {
2     v[i] = a[i] + b[i];
3     assert(c[i]==v[i]);
4 }
```

In this verification method shown in Listing 4.5, the assert function is used to check whether the output array c of the parallel code matches the output array v of the serial code. If any of the elements don't match, the assertion fails with an error. In this way, we can verify the correctness of each function and function in the CUDA parallel program by checking whether it conforms to the expected result.

In addition to benchmarks, we also applied this validation method to the convolutions in CuDNN, where the result of the convolutions is precomputed and stored in an array. First we need to compute the convolutions using the convolution API in CuDNN operational model. Then we use the verification array to compare the results of the calculation. When the assertion is violated, it means that the convolution operation is correct.

4.2 Threats to the Validity

Internal Validity

In the benchmark, each faulty code is designed to have one and only one type of vulnerability. This is to ensure that the results are not derived from other factors. For example, when we verify code with data races, we not only expect a verification failure, but also need to confirm whether the information in the counterexample provided by ESBMC is accurate, and when both are satisfied, the experimental result is considered correct.

External Validity

Our benchmarks are derived from some basic applications of CUDA programs, and some of them are from the official CUDA documentation, which proves the generalisation of this project. At the same time, there are some test cases that are not meaningful, but they have test meaning and allow us to analyse specific errors. It's important to note that both CUDA and C++ are regularly updated. As a result, the outcomes of certain validations might vary over time. To address this, maintenance is essential.

4.3 Experimental Setup

Our experiments aim to investigate two questions:

1. What are the results of ESBMC v7.3 running the above benchmarks?
2. How does ESBMC v7.3 compare with ESBMC-GPU and other CUDA verification frameworks such as GKLEE and GPUVerify?

To validate a CUDA program using ESBMC v7.3, we first need to make some changes to the program so that it compiles in ESBMC, replacing the kernel call function in CUDA with $ESBMC\_verify\_kernel$, we call two threads and a block, or use one thread for each of the two blocks. After the preparation, we need some flags to support the verification in ESBMC:

- *–force-malloc-success*: When it comes to allocating memory, we need to consider whether the available memory on the device is sufficient. This flag indicates that there is always enough memory on the device.

- *–context-bound n*: When we verify concurrent programs, all possible thread schedules as well as interleaving are considered. However, the number of thread interleavings can grow quickly. Therefore, this flag is used to limit the maximum number of instructions a thread can execute before it is switched.

- *–data-races-check*: ESBMC does not actively perform data race checks due to resource overhead. We need to use this flag when we want to check for data races.

- *-I library*: ESBMC supports external libraries, and we need to specify the path to the library using this flag.

We expect two outcomes in our tests: VERIFICATION FAILED, which stands for ”ESBMC found an error in the program,” and then it prints out a counterexample and some tips. VERIFICATION SUCCESSFUL, which means ESBMC found no errors in the program, and no assertions in the program have been violated. All our experiments will be performed using the Python scripts provided by ESBMC, which work by judging the result of the validation by the matching of regular expressions.

$$python3\ testing\_tool.py \tag{4.1}$$

Regarding the second question, it's important to mention that ESBMC-GPU is not actively maintained anymore. As a result, we couldn't test it directly. Instead, we relied on data from relevant research paper [3] for our comparisons.

4.4 Experiments Results

- **True Positive (TP)**: A scenario where a defect is present, and the detection tool accurately identifies it.

- **False Positive (FP)**: A scenario where no defect is present, yet the detection tool mistakenly reports one.

- **True Negative (TN)**: A scenario where no defect is present, and the detection tool correctly refrains from reporting any.

- **False Negative (FN)**: A scenario where a defect is present, but the detection tool fails to recognise and report it.

- **Not Supported (NS)**: CUDA programs are not supported.

	ESBMC v7.3	ESBMC-GPU	GKLEE	GPUVerify	PUG	CIVL
TP	57	67	57	30	15	24
TN	65	60	53	58	39	23
FP	1	3	8	8	11	3
FN	6	1	14	9	7	0
NS	25	23	22	49	82	104
Time (s)	216	811	128	147	12	158

Table 4.1: Comparison of Different GPU Verification Tools

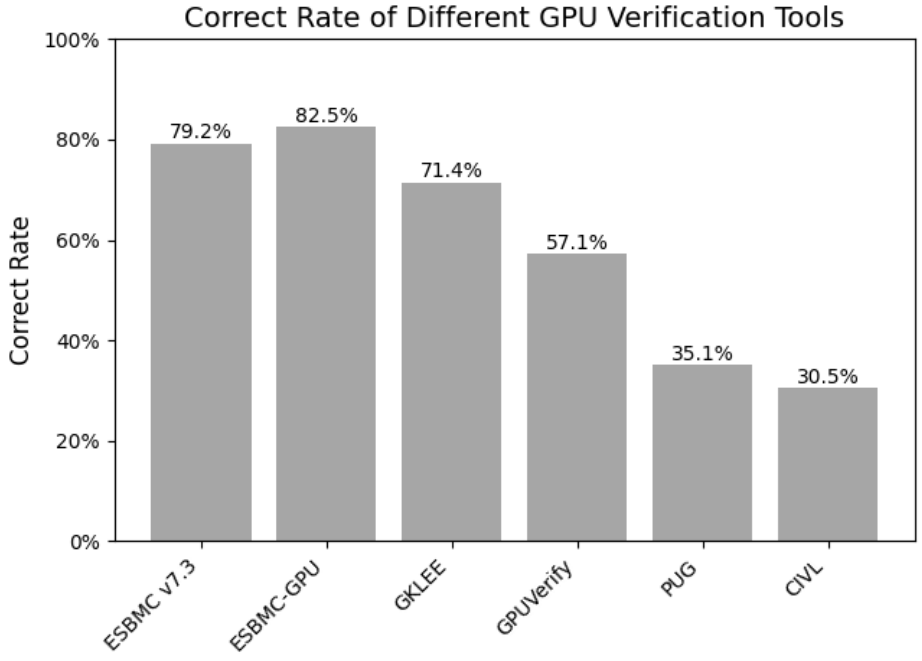

Figure 4.2: Correct Rate of Different GPU Verification Tools

In the benchmark tests for correctness, ESBMC v7.3 had an accuracy of 79.2%, a bit less than ESBMC-GPU at 82.5%. Other tools showed varied results: GKLEE had 71.4%, GPUVerify had 57.1%, PUG scored 35.1%, and CIVL had 30.5%. Looking closer at these numbers, there were some differences between ESBMC v7.3 and ESBMC-GPU. ESBMC v7.3 can not handle with some kind of data races, missing a few error programs that ESBMC-GPU caught. GKLEE had several issues: it didn't always detect data races, had problems with changing constant memory, got some assertions wrong, and had some null pointer access errors. GPUVerify missed some data

races, had problems with array limits, and also got some assertions wrong. PUG's main issues were with null pointer checks, data races, and not always catching when an array went out of its limits. ESBMC-GPU had some errors too, mostly with kernel assertions not returning right and with the *cudaMalloc* function when it worked with copies of floating-point variables. GKLEE also had problems with getting assertions wrong, missing some data races, getting array limits wrong, and had some issues with its solver. PUG mainly had problems with data race checks. CIVL's errors were mostly about getting memory allocation checks wrong and assertion violation.

ESBMC v7.3 has a few more benchmarks than ESBMC-GPU that are not supported. This is mainly because ESBMC-GPU has its own compiler that can make some tweaks and give special meanings to modifier. On the other hand, ESBMC v7.3 relies on the Clang compiler and might need extra changes to add certain features. However, both tools face common challenges, such as accessing constant memory, using certain CUDA libraries, and using pointers to functions, structures, and character variables when making kernel calls.

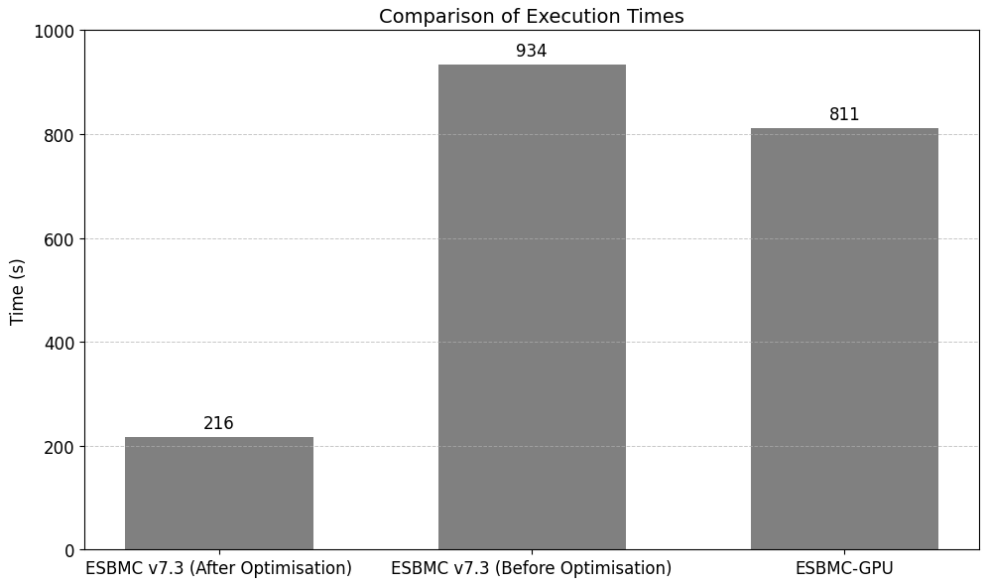

Figure 4.3: Execution Time Comparison

We implemented certain optimisations on COM, which greatly simplify the computation in the model and reduce its time complexity while retaining its original behaviour. As shown in Figure 4.3, compared with the pre-optimisation, our verification time is reduced to one-fifth of the original, which means the verification efficiency of

ESBMC v7.3 is improved. The optimisations we propose in this project also have a significant advantage over ESBMC-GPU. Compared to other CUDA verification frameworks mentioned in this paper, our verification time has become competitive.

Chapter 5

Conclusion and Further Work

This chapter offers an overview of the project. It is structured into three sections. The first section presents an evaluation of the project's outcomes, assessing its successes and areas for improvement. The subsequent section reflects on the project's progression and highlights key insights gained throughout its execution. The concluding section suggests potential avenues for future research, building upon areas identified within this project.

5.1 Achievements

This project delves into the potential risks that CUDA programs may encounter during execution. It underscores the importance of verifying CUDA programs and aims to bridge the gap in CUDA program verification within ESBMC. We adapted the operational model of ESBMC-GPU, which was previously unmaintained and unavailable, to the current ESBMC v7.3.

To enhance the current ESBMC, we expanded its filename detection and refined the Clang C++ front-end to better accommodate code behaviour in the operational model. Notably, we implemented the implicit generation and utilisation of member functions in C++ structs, catering to operations like assignment within structs. An improve was also made for temporary variables, ensuring their correct generation when constructors are directly applied. Subsequent steps included rectifying errors in the operational model, revising certain non-standard code, and streamlining sections of the model to expedite the validation process. Finally, we extend the CuDNN operation model. We analyse and exemplify how the convolution operation and activation function operation work in CuDNN, and build a model based on this theory.

After evaluation and analysis using the benchmark, we obtain a pass rate of 79.2%, which is close to ESBMC-GPU. Compared with other CUDA verification frameworks, ESBMC version 7.3 exhibits not only a higher correct rate, but also an enhanced capability to identify instances of array out-of-bounds and data race violations. By comparing the verification time, we conclude that ESBMC v7.3 improves the verification efficiency by nearly 4 times, which greatly reduces the time spent in our verification.

5.2 Reflection

It can be challenging to migrate application components that are out of maintenance. This project involves a lot of knowledge, including model checking, compilers, ESBMC internal concepts, deep neural networks, etc. Finally, the porting work is completed and the validation is optimised, and the results of the evaluation show that our work is meaningful.

In the development process, we met a lot of problems, the main challenge is to learn the unknown field, ESBMC as a large C++ open source software, the source code involved is complex and difficult for beginners to get started. Second, we had timing issues because the Clang-based C++ frontend had just been released and the team was working on improving it. Therefore, we should use parallelism in our projects to increase efficiency and avoid taking too long to run into each problem.

5.3 Further Work

Despite obtaining satisfactory outcomes in our benchmark, the tabulated data indicates that there remains considerable scope for enhancing our work. For example, ESBMC v7.3 has some issues with data race detection. Specifically, it is better at detecting global variables and weaker at detecting pointer races. In addition, the implementation of the random function model in CUDA is also an important part, in many applications, especially statistical applications, efficient random number generation is very important, we can simulate this process in the future to achieve this feature.

In addition, our modeling of CuDNN is still in the initial stage, so far it can only support convolutions and activation functions. For future work, we can focus on building an entire neural network, such as pooling layer, fully connected layer, etc. Once all the neural network models are built, we can start to validate the CuDNN-based neural networks.

Bibliography

[1] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[2] Felipe R. Monteiro, Mikhail R. Gadelha, and Lucas C. Cordeiro. Model checking C++ programs. *CoRR*, abs/2107.01093, 2021.

[3] Phillipe Pereira, Higo Albuquerque, Isabela da Silva, Hendrio Marques, Felipe Monteiro, Ricardo Ferreira, and Lucas Cordeiro. Smt-based context-bounded model checking for cuda programs. *Concurrency and Computation: Practice and Experience*, 29(22):e3934, 2017.

[4] Lucas Cordeiro. Smt-based bounded model checking for multi-threaded software in embedded systems. In *Association for Computing Machinery*, ICSE '10, page 373–376, 2010.

[5] Mikhail Ramalho, Mauro Freitas, Felipe Sousa, Hendrio Marques, Lucas Cordeiro, and Bernd Fischer. Smt-based bounded model checking of c++ programs. In *2013 20th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*, pages 147–156, 2013.

[6] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, 38(4):957–974, 2012.

[7] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[8] Felipe R. Monteiro, Mário A. P. Garcia, Lucas C. Cordeiro, and Eddie B.

de Lima Filho. Bounded model checking of c++ programs based on the qt cross-platform framework (journal-first abstract). In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 954–954, 2018.

[9] P. Deitel, P.J. Deitel, H.M. Deitel, and an O'Reilly Media Company Safari. *C++ how to Program: Introducing the New C++ 14 Standard*. Deitel Series Page. Pearson, 2016.

[10] Rob Farber. *CUDA application design and development*. Elsevier, 2011.

[11] Marc Jorda, Pedro Valero-Lara, and Antonio J Pena. Performance evaluation of cudnn convolution algorithms on nvidia volta gpus. *IEEE Access*, 7:70461–70473, 2019.

[12] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, 2014.

[13] Marc Jordà, Pedro Valero-Lara, and Antonio J. Peña. Performance evaluation of cudnn convolution algorithms on nvidia volta gpus. *IEEE Access*, 7:70461–70473, 2019.

[14] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[15] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Handbook of satisfiability*, 185(99):457–481, 2009.

[16] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, 38(4):957–974, 2012.

[17] Clark Barrett and Cesare Tinelli. *Satisfiability modulo theories*. Springer, 2018.

[18] Bruno Cardoso Lopes and Rafael Auler. *Getting started with LLVM core libraries*. Packt Publishing Ltd, 2014.

[19] Ronald Gary Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *ACM-SIGACT Symposium on Principles of Programming Languages*, 1989.

[20] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[21] Aina Niemetz and Mathias Preiner. Bitwuzla. In *International Conference on Computer Aided Verification*, pages 3–17. Springer, 2023.

[22] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1):53–58, 2014.

[23] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The mathsat 4 smt solver: Tool paper. In *Computer Aided Verification: 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7-14, 2008 Proceedings 20*, pages 299–303. Springer, 2008.

[24] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 171–177. Springer, 2011.

[25] Bruno Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014.

[26] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, volume 5, pages 1–20, 2008.

[27] Bruno Cardoso Lopes and Rafael Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014.

[28] Felipe R Monteiro, Mikhail R Gadelha, and Lucas C Cordeiro. Summary of model checking c++ programs. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 461–461. IEEE, 2022.

[29] Kunjian Song, Mikhail R Gadelha, Franz Brauße, Rafael S Menezes, and Lucas C Cordeiro. Esbmc v7. 3: Model checking c++ programs using clang ast. *arXiv preprint arXiv:2308.05649*, 2023.

[30] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. Gpuverify: a verifier for gpu kernels. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 113–132, 2012.

[31] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P Rajan. Gklee: concolic verification and test generation for gpus. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 215–224, 2012.

[32] Guodong Li and Ganesh Gopalakrishnan. Scalable smt-based verification of gpu kernel functions. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, 2010.

[33] Manchun Zheng, Michael S Rogers, Ziqing Luo, Matthew B Dwyer, and Stephen F Siegel. Civl: formal verification of parallel programs. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 830–835. IEEE, 2015.

[34] Felipe R. Monteiro, Mikhail R. Gadelha, and Lucas C. Cordeiro. Summary of model checking c++ programs. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 461–461, 2022.

[35] Bjarne Stroustrup. The c++ programming language fourth edition, 2013.

[36] Nicolai M Josuttis. The c++ standard library: a tutorial and reference. 2012.

[37] Scott Meyers. *Effective modern C++: 42 specific ways to improve your use of C++ 11 and C++ 14.* " O'Reilly Media, Inc.", 2014.

[38] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA c programming*. John Wiley & Sons, 2014.

[39] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[40] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. Comparative study of caffe, neon, theano, and torch for deep learning. 2016.

[41] Diego Manzanas Lopez, Taylor Johnson, Hoang-Dung Tran, Stanley Bak, Xin Chen, and Kerianne L Hobbs. Verification of neural network compression of acas xu lookup tables with star set reachability. In *AIAA Scitech 2021 Forum*, page 0995, 2021.