# Checking Memory Safety of CUDA Kernels

*Konstantinos Eleftheriou*

Thesis submitted in partial fulfillment of the requirements for the

*Masters' of Science degree in Computer Science and Engineering*

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Associate Prof. *Polyvios Pratikakis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

# Checking Memory Safety of CUDA Kernels

Thesis submitted by
**Konstantinos Eleftheriou**
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

|   |   |
|---|---|
| Author: | Konstantinos Eleftheriou |
| Committee approvals: | Polyvios Pratikakis<br>Associate Professor, Thesis Supervisor |
|  | Angelos Bilas<br>Professor, Committee Member |
|  | Giorgos Vasiliadis<br>Assistant Professor, Committee Member |
| Departmental approval: | Polyvios Pratikakis<br>Associate Professor, Director of Graduate Studies |

Heraklion, March 2024

# Checking Memory Safety of CUDA Kernels

## Abstract

Graphics processing units (GPUs) are heavily used nowadays for tasks like deep learning model construction and big data analysis, due to their high processing power. Nvidia's CUDA platform enables the use of GPUs for such tasks. CUDA extends the C++ programming language with special functions, called kernels, that run on multiple CUDA threads. Most of the time, kernels perform operations on data that reside in dynamically allocated memory, which can cause runtime errors during the kernel's execution, if a memory access occurs in unallocated memory.

To identify unsafe kernel executions, we perform static analysis on CUDA's intermediate assembly, called PTX. The reason for this is that the source code of Nvidia libraries, like cuBLAS, is not publicly available. At first, we convert the kernel's PTX code into LLVM's intermediate representation in order to detect the loops and the array accesses inside the kernel using LLVM passes. Then, we detect the loop bounds and the expressions used in array accesses and define them in terms of the kernel's paremeters. The analysis generates constraints that are solved using Microsoft's Z3 Theorem Prover. The satisfiability of the constraints determines if the kernel is safe to run.

# Έλεγχος ασφάλειας μνήμης σε CUDA kernels

## Περίληψη

Οι μονάδες επεξεργασίας γραφικών (GPUs) χρησιμοποιούνται ευρέως σήμερα για εργασίες όπως η κατασκευή deep learning μοντέλων και η ανάλυση μεγάλου όγκου δεδομένων, λόγω της υψηλής επεξεργαστικής τους ισχύος. Η πλατφόρμα CUDA της Nvidia επιτρέπει τη χρήση των GPUs για τέτοιου είδους εργασίες. Το CUDA επεκτείνει τη γλώσσα προγραμματισμού C++ με ειδικές συναρτήσεις, που ονομάζονται kernels και εκτελούνται σε πολλαπλά CUDA threads. Οι kernels συνήθως εκτελούν πράξεις σε δεδομένα που βρίσκονται σε δυναμικά δεσμευμένη μνήμη, κάτι που μπορεί να προκαλέσει σφάλματα κατά την εκτέλεση του kernel, αν υπάρξει πρόσβαση σε μη δεσμευμένη μνήμη.

Για να εντοπίσουμε μη ασφαλείς εκτελέσεις των kernels, πραγματοποιούμε στατική ανάλυση στην ενδιάμεση assembly του CUDA, η οποία ονομάζεται PTX. Η επιλογή αυτή οφείλεται στο γεγονός ότι ο πηγαίος κώδικας των βιβλιοθηκών της Nvidia, όπως η cuBLAS, δεν είναι δημόσια διαθέσιμος. Αρχικά, μετατρέπουμε τον PTX κώδικα του kernel σε ενδιάμεσο κώδικα του LLVM, προκειμένου να εντοπίσουμε βρόχους και προσβάσεις σε πίνακες μέσα στον kernel χρησιμοποιώντας passes του LLVM. Στη συνέχεια, εντοπίζουμε τα όρια των βρόχων και τις εκφράσεις που χρησιμοποιούνται για προσβάσεις σε στοιχεία πινάκων και τα ορίζουμε σε συνάρτηση των παραμέτρων του kernel. Η ανάλυση παράγει ανισώσεις που επιλύονται χρησιμοποιώντας τον Z3 Theorem Prover της Microsoft. Η ικανοποιησιμότητα των ανισώσεων καθορίζει εάν ο kernel είναι ασφαλής για εκτέλεση.

# Contents

# Chapter 1

# Introduction

Traditionally, graphics processing units (GPUs) have been used for graphics and video rendering. Lately, the need to accelerate the execution of applications that are compute-intensive has led to the use of GPUs in a wide range of fields, like machine learning and high performance computing. Due to their parallel structure, containing thousands of cores, GPUs can perform computations typically carried out by CPUs in much higher speeds. Several platforms have been developed to enable the execution of programs on GPUs. Compute Unified Device Architecture (CUDA) [2] is such a platform that allows the use of GPUs by applications for general-purpose computing. CUDA programs are written in an extension of C++, which exposes the features of CUDA's programming model. C++ is not a memory-safe language and memory accesses can cause program crashes or the mutation of the program's data. The code that the developers want to run in parallel is written in special C++ functions, called kernels. Our goal is to detect memory accesses in kernels and provide constraints on the memory that is used by the kernels.

## 1.1 CUDA Compilation Process

The source code of a CUDA program consists of the kernels that run on the the device, and the rest of the program that runs on the host. The term *host* refers to the CPU along with its dedicated part on the DRAM, while the term *device* refers to the GPU and the part of the memory that is assigned to it.

During the first step of the compilation process, the device functions and the host code are separated. The device code is then compiled by the Nvidia CUDA Compiler Driver (nvcc) to *cubin* or *PTX* code and the output is placed into a fatbinary. Parallel Thread Execution (PTX) is an intermediate assembly representation, containing instructions in a virtual ISA. PTX is target-independent and it is Just-in-time compiled to a device-specific assembly, before the device code execution. CUDA binaries (cubin) contain target-specific code. The generated fatbinary may contain binaries for multiple device architectures and, optionally, PTX code. The host code is compiled using a C++ host compiler and the fatbinary is embedded in the host object file.

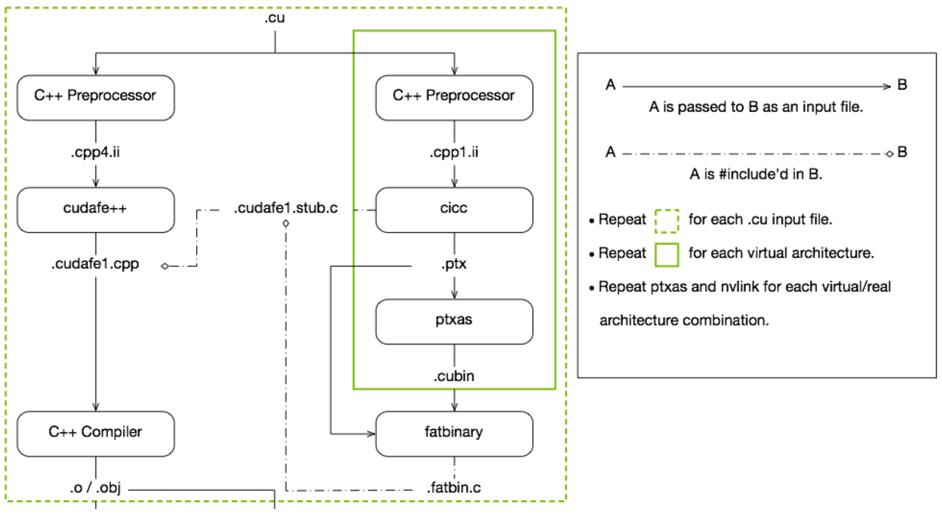
.cu
C++ Preprocessor
.cpp4.ii
cudafe++
.cudafe1.cpp
.cudafe1.stub.c
C++ Preprocessor
.cpp1.ii
cicc
.ptx
ptxas
.cubin
fatbinary
.fatbin.c
C++ Compiler
.o / .obj
A B
A is passed to B as an input file.
A B
A is #include'd in B.
• Repeat for each .cu input file.
• Repeat for each virtual architecture.
• Repeat ptxas and nvlink for each virtual/real architecture combination.

Figure 1.1: CUDA Compilation Process (Source: Nvidia Documentation)

## 1.2 Parallel Thread Execution (PTX)

PTX files contain the part of the code that is executed on the device, namely the kernel functions. This code is independent to the device architecture and it is JIT compiled to a binary in a specific GPU architecture. The syntax of the code in the PTX files is similar to that of a CPU's assembly.

PTX statements consist of *directive* and *instruction statements*. Directive statements include variable declarations in specific state spaces (global, shared, param, etc.), function declarations, kernel entry points and directives specifying information about the target architecture or performance tuning. The most important directive statements for our analysis are the kernel entry points and the kernel's parameters.

PTX instruction statements can be arithmetic or bit-wise operations, logic operations, control-flow or data movement instructions and so on. The instructions are composed of an opcode, zero to four operands and an optional guard predicate that indicates whether the instruction will be executed or not. The operands of an instruction can be virtual registers, constants, address expressions or label names.

## 1.3 LLVM Compiler Infrastructure

LLVM [3] is a framework for constructing compilers and optimizing code. It is composed of multiple language frontends, an intermediate representation and backends for different machine architectures. The frontend is the part of the infrastructure that parses the source code and generates the intermediate representation. The LLVM intermediate representation (IR) is a target-agnostic representation common for all frontends. LLVM provides a collection of transformations and analyses in order to optimize the IR. The IR is in Static Single Assignment (SSA) form, meaning that each virtual register is written once and it is defined before used. The backend converts the IR to the assembly of a specified platform.

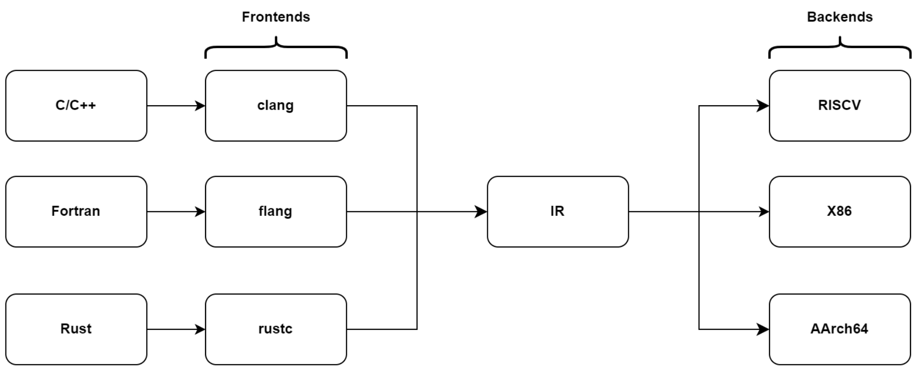

Figure 1.2: LLVM Structure

## 1.4 Kernel Static Analysis

We analyze the memory used by the kernels by performing static analysis on PTX code. We needed to take this path because the source code of Nvidia's libraries, like cuBLAS or cuDNN, is not publicly available. In order to perform the analysis, we create an LLVM frontend for PTX by parsing the PTX code and converting it into the LLVM IR. That way, we can use the passes that the LLVM provides to transform and analyze the code in the IR level, so that we can detect loops and array accesses inside the kernels. Then, we generate inequations based on the loop bounds and the array sizes, which we later solve using the Z3 Theorem Prover in order to determine their satisfiability.

# Chapter 2

# Related Work

While memory safety issues have been studied extensively, there is limited previous work about the memory safety of GPUs. In this section we mention some of those implementations. Honeycomb [5] is a software-based TEE for GPU applications. As part of its implementation, it validates the memory safety of kernels at load time by performing static analysis on the kernel source code. The difference of our implementation in the static analysis part, which is the only common part of the implementations, is that we analyze PTX code. G-Safe [6] is an implementation that divides the shared GPU address space into partitions for different applications to be executed concurrently on the GPU, each using its designated partition. It accomplishes that by instrumenting PTX code of kernels included in closed-source libraries. Our implementation could be used alongside G-Safe in order to avoid the overhead of instrumented code, when possible, by statically validating the safety of memory operations. GPUShield [4] implements region-based bounds checking to improve the memory safety of GPU applications. It performs static analysis using the LLVM IR to reduce the number of runtime checks. The difference with our approach is that it performs static analysis on source code, like Honeycomb. It also reports overflows caused by specific operations to the user and uses the analysis results to minimize runtime checks based on the findings of the analysis for each pointer. Our approach provides a result about the safety of the entire kernel and does not provide information about specific unsafe pointer operations. LocalityGuru [7] presents an approach for analyzing kernels at runtime before being launched. The difference with our implementation is that they perform the analysis on PTX code, instead of converting it to an IR. Also, the goal of LocalityGuru is not memory safety, but to improve the locality of data used by different thread blocks.

# Chapter 3

# Implementation

In this section we dwell on the analysis procedure. At first, we describe some key points about the conversion of PTX assembly into the LLVM IR. Next, we provide details about the analysis and the generation of the constraints. Finally, we outline the process of preparing and passing the constraints as input to the Z3 solver.

## 3.1 PTX to LLVM IR conversion

We convert the PTX statements to LLVM IR instructions by parsing the PTX file, storing the PTX statements in a vector and then converting the PTX statements to the corresponding LLVM IR instructions. Also, we store the mappings between PTX and LLVM instructions.

Most PTX instruction conversions to LLVM IR are straightforward, one PTX instruction corresponds to one LLVM instruction. But, there are cases where one PTX instruction maps to two or more LLVM IR instructions or vice versa. In case that an LLVM instruction maps to many PTX instructions, we have to find patterns in the PTX code, in order to generate the right LLVM instructions. Such cases are described in the below paragraphs.

Figure *3.1* shows an example of the source code of a CUDA kernel, declared using the *__global__* specifier. This kernel iterates through the elements of the array *g_in*, increasing each element by 1 and storing the result in the array *g_out*. The compiler often performs loop unrolling, meaning that it rewrites the loops to contain repeated instructions, reducing the number of iterations. Reducing the number of iterations means reducing the executions of instructions that alter the control flow, ultimately reducing the execution time of the generated assembly. The PTX code that results from this example, breaks the kernel's *for* loop into two loops. The first loop is unrolled four times, so it will be executed *N/4* times, with *N* being the number of iterations. The second loop will execute the remaining iterations, if any. The first loop can be seen in the example of figure *3.5*, as the block labeled *$L__BB0_3* and the second loop in figure *3.2*, as the block with the label *$L__BB0_6*.

```
__global__ void incKernel(int *g_out, int *g_in, int N) {
  for (int i = 0; i < N; ++i) {
    g_out[i] = g_in[i] + 1;
  }
}
```

Figure 3.1: CUDA kernel example

### 3.1.1 Phi node generation

LLVM IR is in SSA form. That is not the case with PTX code, where a register can be defined or updated multiple times. So, when a operand used in an LLVM IR instruction corresponds to a register of this kind, a phi node must be created to specify the operand's value based on the control flow. So, the phi node contains one incoming value for each predecessor of this instruction's block.

```
33:          ; preds = %"$L__BB0_4"
    %34 = sext i32 %31 to i64
    %35 = getelementptr i32, ptr %param_0, i64 %34
    %36 = getelementptr i32, ptr %param_1, i64 %34
    br label %"$L__BB0_6"

"$L__BB0_6":         ; preds = %"$L__BB0_6", %33
    %37 = phi ptr [ %35, %33 ], [ %41, %"$L__BB0_6" ]
    %38 = phi ptr [ %36, %33 ], [ %42, %"$L__BB0_6" ]
    %loopVar1 = phi i32 [ %4, %33 ], [ %43, %"$L__BB0_6" ]
    %39 = load i32, ptr %38, align 4
    %40 = add i32 %39, 1
    store i32 %40, ptr %37, align 4
    %41 = getelementptr i32, ptr %37, i32 1
    %42 = getelementptr i32, ptr %38, i32 1
    %43 = add i32 %loopVar1, -1
    %44 = icmp ne i32 %43, 0
    br i1 %44, label %"$L__BB0_6", label %45
```

Figure 3.2: Loop example in LLVM IR

Figure *3.2* presents an example of a loop created by the compilation of the code in Figure *3.1*. Block *L_BB0_6* is the loop block and it contains three phi instructions, with two incoming values each. The first incoming value specifies the value that will be assigned to the result if the phi node is reached from the previous block (the block outside of the loop) and the second one, the value that will be assigned if the phi instruction is reached from the loop block itself, that is in the next iteration. So, the initial value of register *%37* is the value of *%35* coming from block *33*. After the first iteration and for the rest loop iterations its value will be the value of *%41*, which is updated in each iteration. The same for value *%38*, which is updated with the value of register *%42* after each iteration. The loop variable *%loopVar1* is initialized with the value of register *%4* and updated with the value of *%43*, which is the result of the addition of the step to the loop variable, after each loop.

The creation of the phi instructions goes as follows: We keep an IN and an OUT state for each basic block. Each state is a map of PTX registers to their LLVM IR values. The IN state is created when we enter a new basic block by merging the OUT states of all the predecessor blocks. If there is a conflict on a register it means that this register has been written on multiple predecessors, so we create a phi node containing the values of the OUT states of all predecessors as incoming values. This does not necessarily mean that the incoming values will be different, but in case that they are the same, LLVM drops the phi node during optimizations. The OUT state of each block is initialized with the calculated IN state when we enter a new basic block and updated when a register is written.

### 3.1.2 Getelementptr instruction

The *getelementptr* instruction is used to perform address computations. The first argument is the type of the result. The second argument is a pointer, which is the base address of the structure being indexed. The rest arguments are indices, indicating which elements of the structure are indexed. For example, the *getelementptr* instruction in line 3 of figure *3.2* calculates the address of the fifth element of the *%param_0* array, which contains elements of type i32. In PTX, addresses are calculated using *multiplication* or *shift* and *addition* instructions.

A PTX example containing address computations can be seen in figure *3.3*. This example is part of the result of the compilation of the CUDA source code shown in figure *3.1*. The first three lines of the example load the kernel parameters in virtual registers, the names of which begin with "%". *param_0* refers to the *g_out* array, *param_1* to the *g_in* array and *param_2* to *N*. The *cvta* instructions in the following lines convert the addresses of the arrays from addresses of the generic address space to variables of the global address space. The next *mul* and *add* instructions calculate the offset for the array accesses. The *mul* instruction, multiplies the contents of register *%26* by the size of each array element, which is 4 bytes. The *add* instructions add the result of the multiplication to the base addresses of the arrays to get the offset. The *wide* modifier in the *mul* command causes the result to be twice the size of the operand types, in that case *s64*.

Initially, the parser generates the equivalent instructions in LLVM. But, the addition instruction with pointers as operands is invalid in LLVM. So, after the conversion is completed, these *add* instructions are converted to *getelementptr* and the *mul* or *shl* instruction that calculates the offset is removed. Then, in case that the offset value is constant, it is divided by the size of the array's elements in order to get the index in the array. The *mul* and *add* instructions in lines 7-9 of figure *3.3* are converted to the instructions in lines 2-4 of figure *3.2*. The *wide* modifier causes the generation of the *sext* command, which performs sign extension on the operand in order to double its size. The combinations of the mul and add instructions correspond to the getelementptr instructions. The value returned by the sext instruction is the index of the element to be accessed in the array.

The next block in the PTX code is a loop. The first three instructions load the value from the calculated offset of the *g_in* array, increase the value by 1 and then store the result in the same offset of *g_out*. The conversion of these instructions is trivial and can be seen in the lines 11-13 of the IR example. The next two *add* instructions compute the next offset for the array accesses of the next iteration. These are converted to *getelementptr* instructions and the offset is divided by the size of the array's elements in order for the index to be retrieved. The next *add* instruction is the step of the loop. The *setp* instruction sets the result of the loop's condition to register *%p5*, which is true if the loop variable stored in *%r27* has not reached 0. This instruction is converted to the *icmp* instruction in the IR, which results to true if the register *%43* is not 0 and stores the result in *%44*. The last instruction in the example is the branch instruction, which returns the control flow to the beginning of the loop. *@%p5* is called a guard predicate and it controls if the branch instruction will be executed. So, only if the value in *%p5* is true, the branch instruction is executed. The *bra* instruction is converted to a *br* instruction in IR. In the example, the *br* instructions checks the value of register *%44*, which contains the result of the previous comparison. If the value is true, the execution continues with the next iteration of the loop, otherwise it exits the loop.

### 3.1.3 Static arrays

PTX code may contain static array declarations. In PTX there is no information about the dimensions of the array, thus it is not visible from the type of the array. For example, a 2-dimensional 16x16 array would be shown as an 1-dimensional array containing 256 elements. In order to overcome this issue, we originally declare the LLVM IR array as an 1-dimensional array, as it is in the PTX, with elements of void type. Every time that a *mul* or *shl* operation happens before an array access to get the offset of an element in the array, we update the type of the array based on the instruction's operand. The first time that we encounter such an instruction before an array access, we assume that the variable is an 1-dimensional array. In that case, the value used for the multiplication or shift operation will be the size

```
    ld.param.u64  %rd13, [param_0];
    ld.param.u64  %rd14, [param_1];
    ld.param.u32  %r10, [param_2];
    cvta.to.global.u64  %rd1, %rd13;
    cvta.to.global.u64  %rd2, %rd14;
    ...
    mul.wide.s32  %rd15, %r26, 4;
    add.s64  %rd19, %rd1, %rd15;
    add.s64  %rd18, %rd2, %rd15;

$L__BB0_6:
    ld.global.u32  %r22, [%rd18];
    add.s32  %r23, %r22, 1;
    st.global.u32  [%rd19], %r23;
    add.s64  %rd19, %rd19, 4;
    add.s64  %rd18, %rd18, 4;
    add.s32  %r27, %r27, -1;
    setp.ne.s32  %p5, %r27, 0;
    @%p5 bra  $L__BB0_6;
```

Figure 3.3: Loop example in PTX

of each element in the array and the array type is updated accordingly. The second time, we assume that the variable is a 2-dimensional array. So, we divide the previously set element size with the updated one, as found in the current *mul* or *shl* instruction, to get the size of each row of the array. Then, we get the number of rows by dividing the number of the array's elements by the row size multiplied by the updated element size and we update the type of the array.

```
.shared .align 4 .b8 _ZZ13MatrixMulCUDAILi16EEvPfS0_S0_iiE2As[1024];
...
shl.b32  %r19, %r6, 6;
mov.u32  %r20, _ZZ13MatrixMulCUDAILi16EEvPfS0_S0_iiE2As;
add.s32  %r10, %r20, %r19;
shl.b32  %r21, %r1, 2;
add.s32  %r8, %r10, %r21;
```

(a) Static array declaration and indexing operations in PTX

```
@_ZZ13MatrixMulCUDAILi16EEvPfS0_S0_iiE2As =
    linkonce_odr addrspace(3) global [16 x [16 x i32]] zeroinitializer, align 4
```

(b) Static array declaration in LLVM IR

Figure 3.4: Static array conversion example

The example in figure *3.4* shows a static array declaration in PTX along with the corresponding declaration in LLVM. When we come across the *mov* instruction in line 4 of figure *3.4*, we create an 1-dimensional array in LLVM containing 1024 elements of void type. Supposing that the next instruction calculates the index of the element to be accessed in the 1D array, we update the type of the elements to be *i512*. The size of the elements is extracted from the value that the index is multiplied with, based on the immediate value of the shift instruction, which is 64. After we encounter the second *add* instruction, we eventually realize that the array has two dimensions. So, the previous operations calculate the row number and the current addition operation finds the index of the element in the calculated row number. The updated element size, as extracted by the second *shl* instruction, will be 4 bytes. By dividing the row size with the updated element size we notice that each row contains 16 elements. The number of rows will be 1024 / (16 * 4). Thus, the final type of the array will be [16 x [16 x i32]]. The 4-byte element type could be a float, instead of an integer, but we always generate an integer in this occasion.

### 3.1.4 Block terminators

Every basic block in LLVM IR must end with a *terminator* instruction, like *br*, *ret*, *switch* etc. These instructions indicate which basic block will be executed after the current block. If all instruction conversions in a basic block have finished and the PTX block does not contain instructions that correspond to a terminator instruction, we generate a *br* instruction that transfers the control flow to the next basic block, as it is in the PTX code. An example of such an instruction can be seen in line 5 of figure *3.2*. Here, a *br* instruction has been generated to transfer the control flow from block *33* to block *$L__BB0_6*.

## 3.2 Analysis and constraint generation

The next step is the analysis of the generated LLVM IR code. We detect and analyze the loop bounds and the array accesses inside the kernel. The aim of the analysis is to determine if the kernel code is safe to run before the kernel is executed. CUDA kernels, most of the time, perform operations on arrays which are dynamically allocated, so array accesses with out-of-bounds indices can cause runtime errors. By performing these checks statically, we can minimize the dynamic memory checks while the code is executed.

### 3.2.1 Loop information retrieval

In order to find the loops in the code and retrieve the loop bounds, we first have to run some LLVM passes. LLVM passes perform optimizations and transformations on the IR or compute information used by other passes. At first, we run the *mem2reg* pass, which promotes memory references to registers and it is required by many optimization passes. It basically converts the IR to SSA form by rewriting the *alloca*, *store* and *load* instructions. So, instead of allocating space to store values in memory and then retrieving those values using load instructions, the values are stored in registers. The next pass used is the *Loop Rotate* pass, which transforms the loops into do/while form, while adding a guard for the case that the loop is never executed. This pass is used by the *Scalar Evolution* pass. Next, we run the *Dominator Tree* pass, which is an analysis pass, therefore it does not mutate the code. It creates a tree that represents the dominance relationships in a control-flow graph, which are necessary in detecting the loops. A node i dominates another node j in the control-flow graph if every path from the entry node to j needs to go through i. Finally, we use the *Scalar Evolution* pass that analyzes and categorizes scalar expressions in loops. These expressions might include the loop condition or the induction variable. *Scalar Evolution* pass is used in order to find the loop bounds and recognize the induction variables.

### 3.2.2 Loop bounds constraints generation

After retrieving details about the loops from the results of the LLVM passes explained above, we need to generate constraints regarding the value range of the loop variable. We need to traverse the use-def chain of each value in the loop bounds expressions in order to express them in terms of the kernel parameters. The def-use chain of a value contains all the users of the value, meaning all the instructions that use it as an operand. Thus, for every value in the expression we find its definition, then for every operand in the definition instruction we find its own definition and so on. Another thing that we can retrieve from the previous analysis results is the direction of the loop. In case of forward direction the step is positive and in case of backward direction the step is negative. We need this information in order to know the direction of the generated inequalities. The constraints are of the form $initial\_value \leq loop\_variable \leq final\_value$ if the step is positive or $final\_value \leq loop\_variable \leq initial\_value$ if the step is negative. If a loop variable is unnamed, a new loop variable name will be generated.

Figure *3.5* presents another loop that resulted from the compilation of the source code in figure *3.1*. Here, the loop is the block *$L__BB0_3* with *%loopVar0* as the loop variable and *%28* as the step. The loop has a negative step and the loop variable's final value is 0, as seen from the condition in line 32. The initial value of the loop variable is the value of the register *%7*, as assigned from the *phi* instruction. We traverse the *Use-Def* tree of the initial value as follows. Initially, we look at the operands of *%7* and realize that the first operand can not be expanded further, so we expand the second operand which is an instruction. Then, we try to expand the operands of *%4*. Both of its operands are terminal values, that can not be expanded further, so we return the result. After traversing the *Use-Def* tree of the initial value, which is presented in figure *3.7*, the initial value takes the form shown in line 9 of figure *3.6* as the maximum value of the loop variable.

```
 1  2:          ; preds = %0
 2      %3 = add i32 %param_2, -1
 3      %4 = and i32 %param_2, 3
 4      %5 = icmp slt i32 %3, 3
 5      br i1 %5, label %"$L__BB0_4", label %6
 6
 7      6:          ; preds = %2
 8      %7 = sub i32 %param_2, %4
 9      br label %"$L__BB0_3"
10
11  "$L__BB0_3":            ; preds = %"$L__BB0_3", %6
12      %8 = phi i32 [ 0, %6 ], [ %25, %"$L__BB0_3" ]
13      %9 = phi ptr [ %param_1, %6 ], [ %26, %"$L__BB0_3" ]
14      %10 = phi ptr [ %param_0, %6 ], [ %27, %"$L__BB0_3" ]
15      %loopVar0 = phi i32 [ %7, %6 ], [ %28, %"$L__BB0_3" ]
16      ...
17      %13 = getelementptr i32, ptr %9, i32 1
18      ...
19      %16 = getelementptr i32, ptr %10, i32 1
20      ...
21      %17 = getelementptr i32, ptr %9, i32 2
22      ...
23      %20 = getelementptr i32, ptr %10, i32 2
24      ...
25      %21 = getelementptr i32, ptr %9, i32 3
26      ...
27      %24 = getelementptr i32, ptr %10, i32 3
28      ...
29      %26 = getelementptr i32, ptr %9, i32 4
30      %27 = getelementptr i32, ptr %10, i32 4
31      %28 = add i32 %loopVar0, -4
32      %29 = icmp ne i32 %28, 0
33      br i1 %29, label %"$L__BB0_3", label %30
```

Figure 3.5: Loop example in LLVM IR

The four *getelementptr* instructions for each pointer have occured from loop unrolling. In order to reduce the number of iterations, the compiler repeats the instructions in the loop four times. For the same reason the step is -4 instead of -1. But, the number of iterations will not always be a multiple of 4. The total number of iterations is $N$, as seen in the source code in figure *3.1*, which is value *%param_2* in the IR. The instruction in line 3 performs a bitwise *AND* operation in $N$ with the number 3, in order to get the last 2 bits of $N$. Then, the result is subtracted from $N$ in line 8 and the result is the number of iterations that will be executed in this loop, which is a multiple of 4. The remaining iterations, if any, will be executed in the loop of figure *3.2*.

```
0 <= ceiling((%param_2 - (%param_2 and 3)) / 4) * 1 <= sizeof(%param_0)
0 <= ceiling((%param_2 - (%param_2 and 3)) / 4) * 1 <= sizeof(%param_1)
0 <= ceiling((%param_2 - (%param_2 and 3)) / 4) * 2 <= sizeof(%param_0)
0 <= ceiling((%param_2 - (%param_2 and 3)) / 4) * 2 <= sizeof(%param_1)
0 <= ceiling((%param_2 - (%param_2 and 3)) / 4) * 3 <= sizeof(%param_0)
0 <= ceiling((%param_2 - (%param_2 and 3)) / 4) * 3 <= sizeof(%param_1)
0 <= ceiling((%param_2 - (%param_2 and 3)) / 4) * 4 <= sizeof(%param_0)
0 <= ceiling((%param_2 - (%param_2 and 3)) / 4) * 4 <= sizeof(%param_1)
0 <= loopVar0 <= (%param_2 - (%param_2 and 3))
```

Figure 3.6: Generated constraints

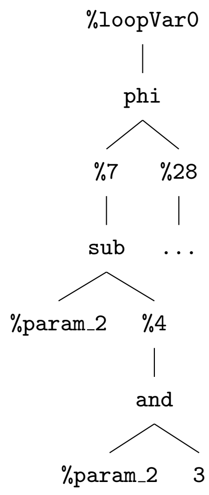

Figure 3.7: Use-Def tree of the loop variable

### 3.2.3 Array access constraints generation

Except for the loop bounds we need to examine the expressions used as indices in array accesses to determine if their values exceed the array sizes. We retrieve these expressions by extracting the indices of the *getelementptr* instructions inside the loop. Then, we perform a *Def-Use* tree traversal similar to what is described in the previous section, starting from the index expression, until we reach a result that depends on the kernel parameters. But, getting the index expression alone is not enough. The way that the loops are constructed, this expression is a constant and it is added to the array pointer on each iteration. So, the variable that changes throughout the loop's iterations is not the index expression but the array pointer. As a result, we have to find the number of iterations and multiply it with the index expression to find the maximum value used as the index. We get the number of iterations by dividing the final value of the loop variable, if the step is positive, or the initial value, if the step is negative, by the absolute value of the step. The number of iterations will be the ceiling of the previous operation's result, in case that the result is not an integer. Finally, the lower bound of the constraints will be 0 and the upper bound will be the size of the array that the pointer corresponds to.

Examples of array access constraints are presented in figure *3.6*. The first four constraints correspond to the *getelementptr* instructions of figure *3.5* in the same order. The other four getelementptr instructions generate identical constraints and they have being removed as duplicates. The step of the loop is negative so we need to get the initial value of the loop variable. The initial value of the loop variable is %param_2 - (%param_2 and 3), so that the number of iterations will be a multiple of 4, and is divided by the step value, which is 4. The ceiling of the result is multiplied by the corresponding index. For example, for the instruction of line 30 of figure *3.5*, it will be multiplied by 4. The upper bound of the same instruction's constraint will be the size of the *%param_0* array, as can be seen from the *phi* instruction, of which the value is assigned to register *%9*. In this example, all of the indices are constant, but this is not always the case.

## 3.3 Constraint solving

The next step is to solve the generated constraints and determine if they are satisfiable. The constraints are satisfiable, if there are values for the variables in the expressions that evaluate the expressions to true. For this purpose, we use Microsoft's Z3 SMT solver [1], which extends the SMT-LIB2 standard. SMT or satisfiability modulo theories are the problems of detecting if a mathematical formula is satisfiable. SMT-LIB is a library used for expressing such problems.

Initially, we have to convert the constraints into the syntax supported by Z3. Every instruction needs to be converted to the corresponding Z3 symbol, e.g. *add* to "+". Also, in case of bitwise logical operations like *and* and *or*, we use the *bv2int* or *int2bv* functions in order to convert the integers into binary vectors, perform the logical operation and then convert them back. We declare each loop variable using the *declare-const* command. In addition, we declare a *ceiling* function used in the array accesses constraints using the *declare-fun* command. Each constraint is added into Z3's stack using the *assert* command. After creating a string containing all the declarations and the assertions, we use Z3's C++ API in order to parse it and solve the constraints. The constraints are solved using the *check-sat* command, which returns *sat* if the constraints are satisfiable, *unsat* if they are not satisfiable and *unknown* if the satisfiability can not be determined. The constraints are meant to be solved dynamically on the host, before the kernel launch.

We have also created the functions *min2* and *max2* that calculate the minimum and maximum value between two integers and correspond to PTX's *min* and *max* instructions. We define the recursive function *maxn* that calculates the maximum of $n$ numbers. This function is used when we come across phi nodes during the constraint generation process in order to get the maximum value of all the incoming values of the phi node. The *shl* function calculates the result of the bitwise left shift operation between two integers and is the equivalent to PTX's *shl* instruction. The declarations of these functions can be seen in figure *3.8*.

```
(define-fun min2((x Int) (y Int)) Int (ite (< x y) x y))
(define-fun max2((x Int) (y Int)) Int (ite (> x y) x y))
(define-fun shl ((x Int) (y Int)) Int
    (bv2int (bvshl ((_ int2bv 32) x) ((_ int2bv 32) y))))
(define-fun-rec maxn ((ls (List Real))) Real
    (if ((_ is nil) ls)
        0
        (let ((hd (head ls))
            (tl (tail ls)))
            (if ((_ is nil) tl)
                hd
                (let ((tlmax (maxn tl)))
                (if (> hd tlmax) hd tlmax))))))
```

Figure 3.8: Custom Z3 functions

Figure *3.9* shows the constraints of figure *3.6* after being converted into Z3's format. The first line declares the loop variable *loopVar0* using *(declare-const loopVar0 Int)*, which is syntactic sugar for *(declare-fun loopVar0 () Int)* and declares a variable of integer type. The second line defines the *ceiling* function, which receives a real number as a parameter and calculates the smallest integer greater than or equal to that number. The operations are using prefix notation. All arithmetic operators have been converted to their corresponding symbols, and the *and* operator has been converted into *bvand*. The *bvand* operation performs bitwise *AND* between two bit vectors. *(_ int2bv 32)* casts the integers into a 32-bit vector used in the *bvand* operation and *bv2int* converts the result back to an integer. The number 1000 is the value of *N*, which is not known statically, so the value has been hardcoded in this example.

```
(declare-const loopVar0 Int)
(define-fun ceiling ((x Real)) Int
(ite (>= (- x (to_real (to_int x))) 0.0) (to_int x) (+ (to_int x) 1)))
(assert
(<= 0
    (*
    (ceiling
    (/ (- 1000 (bv2int (bvand ((_ int2bv 32) 1000) ((_ int2bv 32) 3)))) 4))
    1)
    1000))
(assert
(<= 0
    (*
    (ceiling
    (/ (- 1000 (bv2int (bvand ((_ int2bv 32) 1000) ((_ int2bv 32) 3)))) 4))
    2)
    1000))
(assert
(<= 0
    (*
    (ceiling
    (/ (- 1000 (bv2int (bvand ((_ int2bv 32) 1000) ((_ int2bv 32) 3)))) 4))
    3)
    1000))
(assert
(<= 0
    (*
    (ceiling
    (/ (- 1000 (bv2int (bvand ((_ int2bv 32) 1000) ((_ int2bv 32) 3)))) 4))
    4)
    1000))
(assert
(<= 0 loopVar0 (- 1000 (bv2int (bvand ((_ int2bv 32) 1000) ((_ int2bv 32) 3))))))
```

Figure 3.9: Constraints in Z3 format

# Chapter 4

# Evaluation

In order to test the implementation we analyzed kernels that perform tensor computations from PyTorch repository. Specifically, the PyTorch kernels perform average pooling on 4D tensors. We also analyzed a kernel performing matrix multiplication included in the samples of CUDA 11.6. *AveragePooling0* contains 7 loops and 6 array accesses. The kernel analysis generates 13 compound inequalities, one for each loop and one for every array access. The same is the case with *AveragePooling1*. *AveragePooling2* contains 4 loops and the analysis generates 4 constraints based on the loop bounds, one for each loop. For this kernel, the number of constraints generated due to array accesses is not equal to the number of array accesses present inside the kernel. This happens because of duplicates. There are accesses in the same element of the same array inside the kernel or accesses to arrays of equal sizes. *MatrixMul* contains 1 loop and 10 array accesses and as in the previous example, this kernel generates duplicate constraints too.

| **Kernel** | **Loops** | **Array Accesses** | **Constraints** |
|---|---|---|---|
| AveragePooling0 | 7 | 6 | 13 |
| AveragePooling1 | 7 | 6 | 13 |
| AveragePooling2 | 4 | 10 | 11 |
| MatrixMul | 1 | 10 | 8 |

Table 4.1: Number of constraints generated per kernel

An example of a constraint generated from the analysis of the bounds of a loop in the *AveragePooling0* kernel is shown in figure 4.1. Figures *4.1c* and *4.1b* show the instructions that correspond to the constraint's expressions in LLVM IR and PTX, respectively. The constants shown in the constraint are values that are not known statically and have been hardcoded for illustration purposes and for the constraints to be solvable in this example. The numbers 10, 100, 1000, 1024 and 2048 correspond to the values of *%tid*, *%ntid*, *%ctaid*, *param_2* and *param_3*, as shown in the PTX. The registers *%tid*, *%ntid* and *%ctaid* are vectors referring to the ID of the thread that executes the kernel, the number of thread blocks that execute the kernel and the block ID in which the current thread belongs to. The blocks can have up to 3 dimensions, so the *x* element retrieves the information for the first dimension of the block. These values are retrieved in the LLVM IR using the *@llvm.nvvm.read.ptx.sreg.ntid.x()*, *@llvm.nvvm.read.ptx.sreg.tid.x()* and *@llvm.nvvm.read.ptx.sreg.ctaid.x()* intrinsics shown in figure *4.1c*. The constants 1024 and 2048 correspond to *param_2* and *param_3*, which are integer parameters being passed to the kernel. The constants will be replaced with their real values when the implementation is deployed, by replacing our placeholders with the corresponding values during the kernel launch.

*$L__BB0_13* is the loop block. The variable *loopVar5* shown in the constraint and the LLVM IR is the loop's induction variable. This variable corresponds to the register *%r117* in the PTX. The lower bound of the loop is composed of the expressions in lines 4-17 of figure 4.1a, which have derived from the lines 1-14 of the PTX and the lines 1-10 of the LLVM IR. The lower bound is, basically, the initial value of register *%117* before entering the loop in the PTX and the second incoming value of the phi node in line 14 of the LLVM IR. The upper bound is the value 2048. It corresponds to *param_3* and we can see that it is the value that terminates the loop in the *setp* instruction of the PTX code and the *icmp* instruction of the LLVM IR.

Breaking down the lower bound expression in the constraint of figure 4.1a, we firstly observe a division between *ctaid* and *param_2* created from line 6 of the IR. The result is multiplied by the *ntid* and the *tid* is added to the calculated product. The result is converted into a 32-bit vector using the *int2bv* command, in order for the *bvand* operation to take place. The *bvand* operation performs a bitwise AND operation with -1 to keep the 32 least significant bits of the previously calculated number. At last, the result is converted back to an integer using the *bv2int* command, in order to be used in the comparison. These operations have been generated from the instructions in the lines 8-10 in the IR. The *and* operation has emerged from the *lo* modifier in line 12 of figure *4.1b*, which forces only the lower half of the result to be written in the destination register.

Figures *4.2*, *4.4* and *4.3* present another example of constraint generation with the instructions that correspond to it in PTX and LLVM IR. Here, the constraint has been created due to an array access in the *AveragePoolin0* kernel. Just like the previous example, this example contains some hardcoded values. The constants 512, 1024, 2048, 4096, 8192 correspond to registers *param_2*, *param_10*, *param_11*, *param_12*, *param_13*, respectively. These are parameters passed to the kernel. Constant 100 is the value of register *%ctaid.x*.

The minimum and maximum values of the inequality in this example depend on the size of the array. The size of the array is another value that is not available statically, so it has been hardcoded in this example as the constant 1000000. The minimum index that can be used for the array access is, obviously, zero. The array access occurs in line 39 in the IR using the index calculated by the *getelementptr* instruction in line 37. The equivalent instruction for the array access in the PTX is the *ld* instruction in line 32 and the index is calculated by the *shl* and *add* instructions in line 29-30. The index constraint is generated by the value used in the *getelementptr* instruction. The expression shown in the middle of the inequality in figure *4.2* is the index's expression.

Examining the constraint outwards, we observe a division between the *ctaid* and *param_2* with the result multiplied by *param_2* and an *and* operation with -1 applied to the calculated value. The result is converted to an integer using the *bv2int* instruction and then subtracted from the *ctaid.* These expressions have been created from the instructions in lines 2-6 in the IR and 2-5 in the PTX. Next, there is a multiplication with *param_10*, the result is converted to a 32-bit vector in order for an *and* operation with -1 to take place. The computed value is converted back to an integer by the *bv2int* command. The instructions that led to this expression can be seen in line 8-10 in the IR and lines 7-8 in the PTX. In the IR there is also a *sext* value that has been generated due to the *cvt* instruction in the PTX, which doubles the width of the destination register. This instruction has no effect in the constraint. The expression analyzed above is the second operand of an addition. The first operand is the result of an *and* operation performed in the product of *loopVar3* with *param_12.* The instructions corresponding to these expression can be seen in lines 18-21 in the IR and 14-16 in the PTX. The *loopVar3* variable is defined in line 13 of the IR and has been created because these instructions exist inside a loop block. Again, there is a *sext* instruction in the lines mentioned with no effect in the generated expressions of the constraint. Next, there is an addition of the previously computed value and the result of an *and* operation applied to the product of *loopVar4* with *param_13.* In the lines 25-28 of the IR and 19-21 of the PTX code we can see the relevant instructions. The next expression in the constraint, is a multiplication of *loopVar5* with *param_11.* A bitwise *and* operation with -1 is applied to the result. The calculated value is added to the result of the previous addition. The instructions that caused the generation of these expressions are those in lines 33-16 in the IR and 26-30 in the PTX. The variable *loopVar5* is defined in the line 32 of the IR and it is the induction variable of the loop block *$L__BB0_13.* The *and* operations in this constraint has been generated due to the *lo* modifiers of the *mul* instructions in the PTX.

```
(declare-const loopVar5 Int)
(assert
  (<=
    (bv2int
      (bvand
        ((_ int2bv 32)
          (+
            (*
              (/ 100 1024)
              1000
            )
            10
          )
        )
        ((_ int2bv 32) -1)
      )
    )
    loopVar5
    2048
  )
)
```

(a) Loop Constraint

```
    ld.param.u32 %r46, [param_2];
    ld.param.u32 %r47, [param_3];
    ...
    mov.u32 %r4, %ntid.x;
    ...
    mov.u32 %r6, %tid.x;
    ...
    mov.u32 %r64, %ctaid.x;
    div.u32 %r65, %r64, %r46;
    mul.lo.s32 %r66, %r65, %r46;
    sub.s32 %r12, %r64, %r66;
    mad.lo.s32 %r13, %r65, %r4, %
    r6;
    ...
    mov.u32 %r117, %r13;
    ...
$L__BB0_13:
    ...
    add.s32 %r117, %r117, %r18;
    setp.lt.s32 %p8, %r117, %r47;
    @%p8 bra $L__BB0_13;
    ...
```

(b) PTX

```
    %7 = call i32 @llvm.nvvm.read.ptx.sreg.ntid.x()
    ...
    %10 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
    ...
    %24 = call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
    %25 = udiv i32 %24, %param_2
    ...
    %29 = mul i32 %25, %7
    %30 = add i32 %29, %10
    %31 = and i32 %30, -1
    ...
"$L__BB0_13":       ; preds = %"$L__BB0_13", %94
    ...
    %loopVar5 = phi i32 [ %109, %"$L__BB0_13" ], [ %31, %94 ]
    ...
    %109 = add i32 %loopVar5, %55
    %110 = icmp slt i32 %109, %param_3
    br i1 %110, label %"$L__BB0_13", label %"$L__BB0_14.loopexit"
```

(c) LLVM IR

Figure 4.1: AveragePooling0 loop constraint generation example

```
(assert
(<=
  0
  (+
    (+
      (+
        (bv2int
          (bvand
            ((_ int2bv 32) (* loopVar3 4096))
            ((_ int2bv 32) -1)))
        (bv2int
          (bvand
            ((_ int2bv 32)
              (*
                (-
                  100
                  (bv2int
                    (bvand
                      ((_ int2bv 32)
                        (* (/  100 512) 512)) ((_ int2bv 32) -1))))
                1024))
            ((_ int2bv 32) -1))))
      (bv2int
        (bvand
          ((_ int2bv 32) (* loopVar4 8192))
          ((_ int2bv 32) -1))))
    (bv2int
      (bvand((_ int2bv 32) (* loopVar5 2048))((_ int2bv 32) -1))))
  1000000))

```

Figure 4.2: AveragePooling0 array access constraint generation example

```
    ...
    %24 = call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
    %25 = udiv i32 %24, %param_2
    %26 = mul i32 %25, %param_2
    %27 = and i32 %26, -1
    %28 = sub i32 %24, %27
    ...
    %67 = mul i32 %28, %param_10
    %68 = and i32 %67, -1
    %69 = sext i32 %68 to i64
    ...
"$L__BB0_9":        ; preds = %"$L__BB0_15", %87
    %loopVar3 = phi i32 [ %113, %"$L__BB0_15" ], [ %73, %87 ]
    %88 = icmp sle i32 %85, %81
    br i1 %88, label %"$L__BB0_15", label %89

89:     ; preds = %"$L__BB0_9"
    %90 = mul i32 %loopVar3, %param_12
    %91 = and i32 %90, -1
    %92 = sext i32 %91 to i64
    %93 = add i64 %92, %69
    br label %"$L__BB0_11"
    ...
94:     ; preds = %"$L__BB0_11"
    %95 = mul i32 %loopVar4, %param_13
    %96 = and i32 %95, -1
    %97 = sext i32 %96 to i64
    %98 = add i64 %93, %97
    br label %"$L__BB0_13"
"$L__BB0_13":       ; preds = %"$L__BB0_13", %94
    ...
    %loopVar5 = phi i32 [ %109, %"$L__BB0_13" ], [ %31, %94 ]
    %100 = mul i32 %loopVar5, %param_11
    %101 = and i32 %100, -1
    %102 = sext i32 %101 to i64
    %103 = add i64 %98, %102
    %104 = getelementptr i32, ptr %param_0, i64 %103
    ...
    %106 = load double, ptr %104, align 8
    ...
    %109 = add i32 %loopVar5, %55
    %110 = icmp slt i32 %109, %param_3
    br i1 %110, label %"$L__BB0_13", label %"$L__BB0_14.loopexit"
```

Figure 4.3: AveragePooling0 array access LLVM IR example

```
    ...
    mov.u32 %r64, %ctaid.x;
    div.u32 %r65, %r64, %r46;
    mul.lo.s32 %r66, %r65, %r46;
    sub.s32 %r12, %r64, %r66;
    ...
    mul.lo.s32 %r95, %r12, %r54;
    cvt.s64.s32 %rd10, %r95;
    ...
    $L__BB0_9:
    setp.le.s32 %p6, %r27, %r26;
    @%p6 bra $L__BB0_15;

    mul.lo.s32 %r94, %r114, %r56;
    cvt.s64.s32 %rd9, %r94;
    add.s64 %rd2, %rd9, %rd10;
    mov.u32 %r115, %r26;
    ...
    mul.lo.s32 %r96, %r115, %r57;
    cvt.s64.s32 %rd11, %r96;
    add.s64 %rd3, %rd2, %rd11;
    mov.u32 %r116, %r19;
    mov.u32 %r117, %r13;

$L__BB0_13:
    mul.lo.s32 %r97, %r117, %r55;
    cvt.s64.s32 %rd12, %r97;
    add.s64 %rd13, %rd3, %rd12;
    shl.b64 %rd14, %rd13, 3;
    add.s64 %rd15, %rd1, %rd14;
    ...
    ld.global.nc.f64 %fd3, [%rd15];
    ...
    add.s32 %r117, %r117, %r18;
    setp.lt.s32 %p8, %r117, %r47;
    @%p8 bra $L__BB0_13;

```

Figure 4.4: AveragePooling0 array access PTX example

Chapter 5

# Conclusion

In this thesis, we demonstrated an approach for identifying unsafe kernel executions, by performing static analysis on kernel code. The analysis process involved converting the kernel's PTX assembly into the LLVM IR and the use of LLVM's passes in order to detect loop bounds and array access inside the kernels. The analysis generates constraints, solved using the Z3 SMT solver. The satisfiability of the constraints determines the memory safety of the kernel. We evaluated the implementation by analyzing kernels from the PyTorch repository. We presented the relation between the number of loops and array accesses inside the kernels and the number of generated constraints. Finally, we presented examples of constraints that were generated from PyTorch kernels. Our results show that the static generation of constraints on the memory used by a kernel is possible, so that the dynamic memory checks for every memory access inside the kernels can be reduced.

# Bibliography

[1] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[2] David Kirk. NVIDIA CUDA software and GPU parallel computing architecture. volume 7, pages 103–104, 10 2007.

[3] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[4] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. Securing GPU via region-based bounds checking. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 27–41, New York, NY, USA, 2022.

[5] HaoHui Mai, Jiacheng Zhao, Hongren Zheng, Yiyang Zhao, Zibin Liu, Mingyu Gao, Cong Wang, Huimin Cui, Xiaobing Feng, and Christos Kozyrakis. Honeycomb: Secure and efficient GPU executions via static validation. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 155–172, Boston, MA, July 2023.

[6] Manos Pavlidakis, Stelios Mavridis, Antony Chazapis, Giorgos Vasiliadis, and Angelos Bilas. G-safe: Safe GPU Sharing in Multi-Tenant Environments. In *Under submission*, 2024.

[7] Devashree Tripathy, AmirAli Abdolrashidi, Quan Fan, Daniel Wong, and Manoranjan Satpathy. LocalityGuru: A PTX Analyzer for Extracting Thread Block-level Locality in GPGPUs. In *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2021.