Master's Thesis

in the Course of Study
Computer Science

in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Implementing a parallel Points-To Analysis

Submitted by

Lukas Böttcher
Matr. Nr.: 1125862
stu210239@mail.uni-kiel.de

at the Kiel University

University supervisors:	Prof. Dr. Dirk Nowotka
	Dipl.-Inf. Philipp Maximilian Sieweck
Version from:	July 9, 2024

Declaration

Abstract

Pointer analysis is a subgroup of static analysis techniques. In general, the goal is to detect a set of memory locations for each pointer element in a program. All major compiler systems make use of this information in order to optimize the given code during compilation and most importantly, to detect errors in the program. In this thesis an interprocedural inclusion-based pointer analysis, PTAGPU, is presented that utilizes the parallel execution of GPGPUs to speed up the pointer analysis as part of the SVF framework. As the SVF framework itself is built on top of the LLVM compiler infrastructure, it is possible to run the proposed analysis on any program that can be compiled inside the LLVM toolchain. A meaningful speedup was observed with the proposed parallel implementation compared to the single threaded highly optimized wave propagation inclusion-based pointer analysis. An evaluation of the proposed analysis is provided both through the SVF test suite and a collection of open source programs used as benchmarks.

Keywords: pointer analysis, static analysis, GPU acceleration

Acknowledgment

I would like to thank the dependable systems working group of the department of computer science at the Christian-Albrechts university for accommodating this master thesis and providing the hardware required to perform research for this thesis. Special thanks to my advisor Philipp Sieweck for providing guidance during the work on this thesis.

Contents

Chapter 1

Introduction

The goal of this thesis is the implementation of a parallel pointer analysis. As well as researching to what extent such an implementation presents advantages or disadvantages over other pointer analyses that are not strictly parallel in nature.

A digital version of this thesis together with the full source code of the developed software is available online at https://git.informatik.uni-kiel.de/stu210239/masterarbeit.

1.1 Structure of this Thesis

This thesis is divided into three chapters. The first chapter, chapter 1, lays the groundwork for the implementation and goes into detail what ideas were pursued in order to develop the implementation. All related current work and its influences on this thesis are discussed here, as well as the motivation for the implementation itself. Furthermore, the fundamentals of pointer analysis are explained here with code samples and an end to end analysis workflow that aims to illustrate the connection between actual code and its representation in a pointer analysis.

In the second chapter, chapter 2, the software, namely PTAGPU, that was developed as part of this thesis, is described in detail. Design decisions, integrations with other software libraries and correctness are elaborated here. The experimental benchmark results and how they were generated are also presented here.

The last chapter, chapter 3, covers possible future work that could further improve the implementation and explore more ideas concerning parallel pointer analyses. This chapter also discusses the experimental results from chapter 2.

1.2 Motivation

This thesis aims to explore the possibilities of parallelizing the Andersen style inclusion-based whole program pointer analysis. Specifically the goal is to improve static pointer analysis performance by using massively parallel GPGPUs. With the general trend of more complex software systems in software development, developers also require static analysis tools that are able to perform scalable analyses on entire codebases. Unfortunately general pointer analysis is an undecidable computational problem [Lan92], which prevents fully precise pointer analyses from being a possibility. For this reason all pointer analyses are approximate, and a balance must be found between performance and precision when analyzing code. Historically most pointer analysis solutions have been implemented as single threaded applications, because pointer analyses are challenging to parallelise [SYXL15]. By using GPGPUs the proposed library from this thesis aims to improve performance when analyzing entire programs by distributing the work across the many streaming processors modern GPGPUs possess. The performance improvement of the proposed implementation also should not decrease the analysis precision.

1.2.1 Static Analysis in Software Development

Currently, most compilers employ almost entirely intraprocedural pointer analyses to allow for a fast and scalable static analysis during compilation. Only recently limited support for interprocedural analyses was introduced in the GCC compiler[1]. With greater complexity in modern software projects, these ad hoc intraprocedural analyses during compilation are not sufficient for finding most bugs because they suffer from low accuracy and missing semantic information [GZJ+20]. Especially if a software project is composed of many components, interprocedural analysis is required for finding bugs that arise from interactions between individual components. One such project is the Linux kernel, which consists of a number of subcomponents that handle various parts of the kernel, such as drivers, cryptography and file systems. While utilizing static analysis tools on the Linux kernel is nothing new, precise interprocedural analysis still is a challenging problem for almost 30 million lines of code in the Linux kernel. For this reason this thesis explores options for improving scalability and performance of pointer analysis methods.

[1]`https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc/Static-Analyzer-Options.html`

1.3 Pointer Analysis

In general a pointer analysis tries to find the values of pointers in a program at runtime, without having to execute the program. So naturally this problem is undecidable [Lan92] following a reduction from the halting problem. As a result, performing a pointer analysis becomes a delicate balancing act between precision and performance. Commonly, analyses produce over-approximations of the targets each pointer can point towards at runtime while other parts of an analysis might omit or under-approximate certain parts for the sake of performance and scalability. As a result, these analyses are strictly speaking unsound or soundy as put forward by [LSS+15]:

> "We introduce the term soundy for such analyses. The concept of soundiness attempts to capture the balance, prevalent in practice, of over-approximated handling of most language features, yet deliberately under-approximated handling of a feature subset well recognized by experts. Soundiness is in fact what is meant in many papers that claim to describe a sound analysis. A soundy analysis aims to be as sound as possible without excessively compromising precision and/or scalability." [LSS+15]

Pointer analyses build the foundation for a variety of other static analyses such as value-flow analyses, since call-graph generation is directly dependent on a prior pointer analysis in order to resolve indirect or dynamic calls statically. Without pointer information most static analysis algorithms are simply unable to reason about the state of a given program.

One common pointer analysis is the inclusion-based Andersen analysis [And94]. The details of this type of analysis will be discussed later on, as it is the underlying basis for the proposed algorithm in chapter 2. The Andersen algorithm sacrifices precision in favor of performance and achieves an upper bound of $O(n^3)$ where n represents the number of pointer variables relevant to the analysis. This is known as the cubic bottleneck of general Andersen analysis [MP21]. This showcases the trade-off that all non-theoretical pointer analyses have to make in order to be applicable to real programs and avoid undecidability. Furthermore, Andersen's analysis is a P-complete problem and is therefore not trivially parallelizable [MP21].

As a general abstraction, pointer analyses can be seen as complex graph problems where programs are interpreted as graphs with nodes representing variables and edges representing relations between nodes, such as memory allocations and assignments between variables. This allows us to make use of a large body of previous research concerning graph problems and transform the general analysis into a better defined mathematical problem.

Another analysis closely related to pointer analysis is alias analysis, where two pointers are said to alias if their points-to sets have an intersection. An alias analysis produces a set of relations over all nodes in the analysis graph where nodes can either **NotAlias**, **MayAlias** or **MustAlias**. For two given nodes, a, b and their points-to sets $pts(a)$ and $pts(b)$, the following constraints describe the relations.

$$a\ \textbf{NotAlias}\ b \iff \forall ptd \in pts(a) \colon ptd \notin pts(b) \tag{1.1}$$

$$a\ \textbf{MayAlias}\ b \iff \exists ptd \in pts(a) \colon ptd \in pts(b) \tag{1.2}$$

$$a\ \textbf{MustAlias}\ b \iff \forall ptd \in pts(a) \colon ptd \in pts(b) \tag{1.3}$$

Both pointer analysis, alias analysis, and points-to analysis are all terms commonly used interchangeably in literature [Hin01]. From now on pointer analysis will be used in this thesis to refer to this type of static analysis from which an alias relation can be derived based on the pointer information.

A motivating example for pointer analyses is the detection of memory leaks in programs. This occurs when a memory location is allocated on the heap, for example with a call to `malloc` in glibc, and is not freed at a later stage in the program. It is in the interest of the developer to find such faults as to not exhaust the computer's memory during execution by repeatedly allocating memory in the heap without freeing previous allocations. Finding such logical errors can be accomplished via a related static analysis called data-flow analysis, that tries to determine where specific data might flow in a program. Here pointer information is vital, as pointers can represent lateral movement of data through the control flow of a program, independent of direct assignments and read operations. Ultimately almost all static analyses require some kind of information about pointers to fully determine the state of a program. Aside from error detection such as memory leaks, optimizations are another aspect of compiler systems, where pointer information is important to achieve better results, see Listing 1. More often than not the pointer information alone does not provide an immediate value to the compiler or analysis tool, instead other procedures build on top of this information to derive valuable information about a program.

1.3.1 Notions of Sensitivity in Pointer Analysis

As previously established, a complete general pointer analysis is undecidable. For this reason there are various notions of sensitivity when talking about pointer analysis. These notions represent a compromise between precision, scalability and complexity of the analysis. Following, some of the more common sensitivity notions will be illustrated to differentiate

```c
#include <stdlib.h>
void *iter;
iter = value;

/* depending on the data at value's memory location
the loop might not be necessary */

while(*iter)
{
    complex_computation(iter);
}
```

Listing 1: Optimizations in a c program

```c
int a;
char b;
struct Person {
    char *name;
    int *age;
} p1, p2;
p1.age = &a; // the analysis can differentiate between p1.age and p1.name
p1.name = &b;
```

Listing 2: Field-sensitivity by example

the more complex analyses from the less complex analyses and explain the impact of these sensitivities on actual performance when analyzing a program.

Field-sensitivity

Field-sensitivity describes how the pointer analysis algorithm handles structures in the program. Most programming languages that expose memory management to a developer, such as C, C++ or Rust, offer some form of structures to represent an object that internally holds multiple values where these values might be pointers, that reference memory locations. If an analysis is field-sensitive, each field of each struct is represented in the analysis as an independent node that can point to unique memory locations, as long as the field can be statically determined during the analysis, see Listing 2. If the field of a struct can not be statically determined, for example because of an arithmetic operation that produces multiple possible results for the offset during runtime, it is common for field-sensitive pointer analyses to fall back to a field-insensitive mode for the specific struct, wherein all fields of the struct are merged into a single abstract object. For the given example, `p1.age` and `p1.name` can point to different memory locations. Alternatively a field-insensitive analysis does not differentiate between any fields of a given struct at any point of the analysis. Therefore, only two nodes are created to represent the struct, $p1.*$ and $p2.*$. Another common alternative is field-base-sensitivity, where instead of omitting the individual fields of each struct, the fields of every struct are merged into a single instance of that struct. As a result the Person structs, p1 and p2, would be represented as a single object with fields name and age, such that `p1.age == p2.age` are represented by the same node in the analysis.

Array-sensitivity

Array-sensitivity is conceptually similar to field-sensitivity but often has different effects on the runtime of the analysis. For a given array $intarr[100]$ an array-sensitivie analysis would model each entry of the array, e.g. arr[0], arr[1], . . . , with a unique node, whereas an insensitive analysis would model the array as a single node. Generally speaking arrays are often homogeneous data structures that can hold a vast amount of data, compared to structs which are often more compact as they model attributes instead of raw data. Therefore, array-sensitivity if often omitted from whole program analyses, while field-sensitivity is common among pointer analyses.

Scope of the analysis

When designing a pointer analysis one has to make a decision about how to handle external code that the program depends on. Often times transitive dependencies of a program can dwarf the original code by several magnitudes in size [TG17]. Even a basic Hello World program in Java transitively depends on 3000 classes [KMZN16] from the Java standard library. For this reason most analyses either ignore external library code during analysis, or stub the most relevant library calls during analysis, such as `malloc` or `free`. This trade-off is well worth it, as most interesting properties in pointer analysis do not originate in external libraries, but the actual program code that is written by the developer. This does however not solve the problem of standard library code mutating the program state either via callbacks or mutation of values behind pointer arguments. Here, simply ignoring the external code during analysis would greatly decrease the accuracy of the analysis. For this reason, a lot of research is being done to develop methods that alleviate some of the problems that arise from analyzing external dependencies of a given program. Caching incremental results during analysis seems to be one of the most promising methods thus far [MGR13], where instead of solving the pointer analysis problem from the top-down, the analysis begins at the bottom and builds summaries for functions incrementally until a

```
int *manupulatePointer(int *ptr);
int main() {
    int *a, *b;
    b = manupulatePointer(a);
    /* intraprocedural analysis is unable
    to determine the state of a or b */
}
```

Listing 3: Limitations of intraprocedural analysis

result over the entire program is achieved. Hybrid approaches combining top-down and bottom-up analysis represent state-of-the-art analysis methods in use by production static analysis tools, such as Coverity [MGR13].

Interprocedural analysis

Another aspect that greatly influences the precision of analyses is whether they are interprocedural or intraprocedural. An intraprocedural analysis only analyzes each function in an isolated context and disregards any influences on other functions or global state. Interestingly most compilers rely mostly on intraprocedural analysis for bug detection as it can be performed in parallel for each function independently and is in general much faster than interprocedural analysis. The following example Listing 3 illustrates the shortcomings of only performing intraprocedural analysis. Essentially parameter passing, especially of pointers, is not taken into account properly for the calculation of points-to sets. An interprocedural analysis overcomes these limitations by connecting parameters of functions and the arguments at the respective call-sites as well as the resulting return values in the graph structure that is used to solve the pointer analysis. Fundamentally an interprocedural analysis is related to another notion of sensitivity, context sensitivity, since every context-sensitive analysis has to be interprocedural in order to capture the context of each function call [Lin15].

Flow-sensitivity

When one performs a flow-sensitive pointer analysis, this means that the analysis takes into account the control flow of the program when calculating points-to information. As can be seen in Listing 4, a flow-sensitive analysis is in general more precise than a flow-insensitive analysis. Meanwhile, running a flow-sensitive analysis can also be exponentially more expensive to compute as every step in a programs control flow carries its own state concerning points-to relations - especially when the control flow is complicated by complex

```
int *manupulatePointer(int *ptr);
int main() {
    int a, b, *x; // x -> {}
    if (something())
        x = &a; // x -> {a}
    else
        x = &b; // x -> {a,b} ?
    manupulatePointer(x);
    /* a flow insensitive analysis computes
    a points-to set {a,b} for x while in actuality
    x = &b and x = &a are mutually exclusive statements
    during execution */
}
```

Listing 4: Flow-sensitivity by example

conditional statements or recursive execution. Although this problem can be slightly alleviated, by only considering program statements that manipulate pointers. Empirical studies have shown that for context-insensitive analyses, adding flow-sensitivity to the points-to calculation does not offer a significant precision improvement over flow-insensitive analyses [Hin01]. This makes using a flow-sensitive analysis without context-sensitivity unattractive as an initial analysis run. One should rather combine context- and flow-sensitivity in subsequent analysis runs to refine the initial points-to results in certain regions of code that profit from a further refinement. Using a flow-sensitive pointer analysis also generates must-alias relations, compared to the comparatively imprecise may-alias relations from a flow-insensitive pointer analysis. Generating definitive information that two variables will unconditionally alias during runtime is very valuable when considering refactoring optimizations by a compiler. While a sound may-alias analysis requires that no possible alias relations are missed, a sound must-alias analysis requires analogously that no spurious alias relations are reported. Both are respectively over- and under-approximations of the true points-to results.

Context-sensitivity

As previously alluded to, context-sensitivity is directly related to interprocedural analyses, since it governs how call sites and called functions are interpreted during the analysis. More specifically a context-sensitive analysis tries to qualify variables both on the heap and stack with contextual information such that different contexts can be established, where for example points-to information for a variable differs, thus improving the precision of the analysis. To achieve this, context-sensitivity can be modeled by using call-sites and

```
int *manupulatePointer(int *ptr);
int main() {
    int a, b, *x;
    x = &a;
    manupulatePointer(x);
    x = &b;
    manupulatePointer(x);
    /* a context-sensitive analysis evaluates both
    calls to manupulatePointer as unique function calls
    since the context differs between both calls */
}
```

Listing 5: Context-sensitivity by example

objects to differentiate and qualify the context for variables [SB+15]. Depending on the programming language at hand, these methods can yield different precision. It has been established that object-oriented languages like Java greatly benefit from object-sensitivity over call-site-sensitivity, while more procedural languages like C benefit from call-site-sensitivity. While a context-sensitive analysis would in theory provide more precision and therefore decrease the average size of points-to sets, in practice most context-sensitive pointer analyses, when applied to sizeable codebases, quickly grow out of control as the analysis explodes in terms of running time and space requirements [SKB14]. In contrast, context-insensitive analyses anecdotally scale better than context-sensitive analyses.

1.3.2 Andersen's Analysis

Andersen's analysis is an inclusion-based interprocedural pointer analysis algorithm first proposed by [And94] in 1994. It is a field-sensitive, context-insensitive and flow-insensitive analysis. The algorithm was one of the first constraint based algorithms introduced for pointer analysis. Since it is lacking constext- and flow-sensitivity, it is often used as a base algorithm which produced broad over estimations of the points-to data and is later refined by more precise algorithms which improve the quality of the data and remove false positives from the points-to information generated by Andersen's algorithm. The underlying idea is that the Algorithm operates on a given program by converting statements from the program into mathematical constraints contained in a constraint graph. These constraints can be classified into a few types which can be seen in Table 1.1. It is worth noting that in literature the field-sensitivity aspect is often omitted from the definition of the Andersen analysis, although it was included in the original specification. As mentioned in chapter 1 the complexity of Andersen's analysis grows exponentially with regard to the

Statement	**Name**	**Description**	**Constraint**
$x = \&a$	alloca	The address of a is assigned to x.	$\{a\} \subseteq pts(x)$
$x = y$	copy	Variable y is assigned to x.	$pts(y) \subseteq pts(x)$
$x = *y$	load	Load value of y and assign to x.	$\forall p \in pts(y) \colon pts(p) \subseteq pts(x)$
$*x = y$	store	Store y into value of x.	$\forall p \in pts(x) \colon pts(y) \subseteq pts(p)$
$x = y.f$	field	Field f of variable y is assigned to x.	$pts(y.f) \subseteq pts(x)$

Table 1.1: Constraints of an inclusion-based pointer analysis.

number of pointer variables in a program. The reason for this exponential growth, among other aspects, is the field-sensitivity. Depending on the structure of the code under analysis, field-sensitivity might play the most influential part in the analysis' complexity. This will be further expanded upon in chapter 2. Ultimately this is the reason for specifically including field-sensitivity when discussing Andersen's analysis in this thesis. During execution an Andersen style pointer analysis repeatedly applies the constraints until a point is reached where no more changes are applied to the constraint graph at which point the execution concludes and the points-to sets for each variable are returned.

1.3.3 Steensgard's Analysis

Steensgard's analysis was introduced in 1996 by [Ste96]. It was inspired by Andersen's analysis and as such is also an interprocedural pointer analysis. The key proposition of Steensgard's work was to improve the runtime of Andersen's algorithm by using equalities instead of subsets for the constraints that are used as inputs for the algorithm, an overview for the constraints can be seen in Table 1.2 - the rules are nearly identical to the constraints for the Andersen algorithm. The change from subsets to equalities leads to an almost linear algorithm by utilizing union/find data structures for efficient computation of a fixpoint solution for a given set of pointers. The trade-off for this faster algorithm is precision, since Steensgard's algorithm quickly loses precision compared to Andersen's algorithm by losing the small differences between points-to sets of individual variables by equating them. An example for this precision loss can be seen in Listing 6.

```c
int main() {
    int a, b, *x, *y;
    x = &a; // pts(x) = {a}
    y = &b; // pts(y) = {b}
    y = x; // pts(y) = pts(x) = {a,b}
    /* by equating the points-to sets of x and y
    the fact that x never points to b is lost
    this leads to an obvious loss of precision */
}
```

Listing 6: Steensgard's analysis quickly looses precision during analysis.

Statement	**Name**	**Description**	**Constraint**
$x = \&a$	alloca	The address of a is assigned to x.	$pts(x) = \{a\} \cup pts(x)$
$x = y$	copy	Variable y is assigned to x.	$pts(y) = pts(x)$
$x = *y$	load	Load value of y and assign to x.	$\forall p \in pts(y) \colon pts(p) = pts(x)$
$*x = y$	store	Store y into value of x.	$\forall p \in pts(x) \colon pts(y) = pts(p)$
$x = y.f$	field	Field f of variable y is assigned to x.	$pts(y.f) = pts(x)$

Table 1.2: Constraints of an equality-based pointer analysis.

1.3.4 Wave Propagation

Since its first introduction in 1994, there have naturally been many incremental improvements to the Andersen style pointer analysis. Most current implementations are derived from [PB09], specifically the Wave Propagation Method, which is a highly optimized version of Andersen's algorithm. In Wave Propagation the procedure is separated into an insertion phase and a propagation phase. Furthermore, the constraint graph is topologically sorted and acyclic, which enables the algorithm to pass forward the computed points-to information in topological order, preventing redundant work since only set differences need to be propagated to the next nodes. The algorithm is intended to be more memory intensive in order to achieve better performance on large codebases [PB09]. The general algorithm for wave propagation is listed in Algorithm 1.

1.3.5 LLVM - Generating Data for the Analysis

By now the fundamentals of pointer analysis have been introduced, which unilaterally can be modeled as a graph problem. The missing part of the introduction is where the underlying data for such a pointer analysis algorithm comes from or how it is derived from a given program that has to be analyzed. Initially pointer analyses were solely implemented in compilers to detect errors and find possible optimizations during compilation. As compilers already employ an internal representation for the programs to be compiled, the data generation was not problematic. As the scope of pointer analyses expands from intraprocedural analysis part of a compiler towards standalone interprocedural whole program analyses, it is clear, that a new representation for programs is needed, on which analyses can run - independent of the compilation process.

During initial review of literature for pointer analysis multiple methods for data generation were surveyed. Notably, simply parsing the source code was among the most common data extraction methods for programs. Another method was to extract an intermediate representation, called LLVM-IR, of the code during initial compilation of a program by means of using the low level virtual machine, LLVM. Using LLVM has some distinct advantages compared to parsing the source files of a program directly. For one, LLVM provides multiple compiler front-ends for various compiled programming languages, including C/C++ and Objective-C through the Clang compiler or Rust through the rustc compiler, which allows one to compile multiple languages without having to adapt the parser, as can be seen in Figure 1.1. Especially when working with older non-strictly standardized versions of the C language, utilizing all available tricks of an established compiler proves to be more resourceful compared to reinventing the wheel with new parsing tools. Secondly

Algorithm 1 General Wave Propagation Algorithm
Input: Constraint Graph $G = (V, E)$
Output: Modified Constraint Graph $G = (V, E)$ and points-to information.

```
Detect strongly connected components and find topological sorting for $G$.
Build topological node stack $T$.
repeat
    changed = False
    worklist = $\emptyset$
    while $T \neq \emptyset$ do
        node ← pop from T
        for edge (node,target) ∈ $out_{copy/gep}(node)$ do
            pts(target) ← union pts(node) pts(target)
            if pts(target) changed then
                changed = True ∧ add target to worklist
            end if
        end for
    end while
    while worklist $\neq \emptyset$ do
        node ← pop from worklist
        for edge (node,target) ∈ $out_{load}(node)$ do
            for ptsDst ∈ $pts(node)$ do
                add copy edge (ptsDst, target) to $G$.
                if edge added to $G$ then
                    changed = True
                end if
            end for
        end for
        for edge (src,node) ∈ $in_{store}(node)$ do
            for ptsDst ∈ $pts(node)$ do
                add copy edge (src, ptsDst) to $G$.
                if edge added to $G$ then
                    changed = True
                end if
            end for
        end for
    end while
    if Edge added to $G$ then
        changed = True
    end if
until changed = False
```

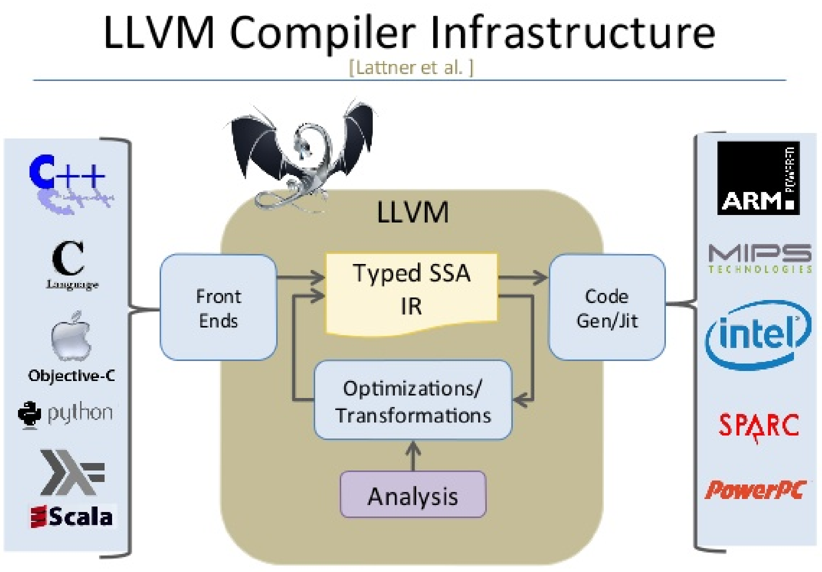


Figure 1.1: Illustration of the LLVM toolchain from Lattner et al.

one can easily verify the correctness of the extracted data for a program by simply executing the compiled intermediate representation since no matter the programming language, as part of the LLVM toolchain the program always gets compiled into the intermediate representation before being assembled and linked. The LLVM project provides specific tools for executing programs in LLVM-IR format using a just-in-time compiler [2]. Beyond the binary intermediate representation, also called bitcode, there exists a human-readable format. Both binary and text versions can be converted between with the LLVM tools `llvm-as` and `llvm-dis`, see Listing 7 for a basic hello world program in human-readable LLVM-IR.

LLVM Instructions

With the generated LLVM-IR we can now build a graph, by interpreting the individual LLVM instructions as constraints for a chosen pointer analysis. The LLVM-IR uses static single assignment (SSA) form for variables meaning that each variable can only be assigned a single time in a specific control flow in the intermediate representation. The use of SSA form is not directly relevant to Andersen style analyses but simplifies working with variables conceptually, since no variable can be reassigned at any point of the program. At this point it is also important to differentiate between address-taken variables and top-level variables. Top-level variables' values reside in registers and are conceptually ephemeral

[2]https://releases.llvm.org/9.0.1/docs/CommandGuide/lli.html

```
#include <stdio.h>
int main()
{
    int i;
    i = 10;
    printf("Hello World! N = %d\n", i);
    return 0;
}
// gets compiled into...

@.str = private unnamed_addr constant
    [21 x i8] c"Hello World! N = %d\0A\00", align 1
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 10, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
        ([21 x i8], [21 x i8]* @.str, i64 0, i64 0), i32 %3)
  ret i32 0
}
declare dso_local i32 @printf(i8*, ...) #1
```

Listing 7: A basic hello world program in human-readable LLVM-IR.

while address-taken variables are abstract memory objects which logically reside in memory. The specifics of memory locations and cpu registers are of course hardware specific and depend on the target architecture that the LLVM backend assembles the intermediate representation into. The connection between address-taken and top-level variables in the LLVM-IR is established by the `alloca` instruction which maps a top-level variable to a memory location represented by an address-taken variable. Following is an interpretation of the LLVM instructions for Andersen's pointer analysis.

Alloca Instruction The LLVM `alloca` instruction is used to allocate memory on the stack.

```
; allocate a pointer to a 32 bit integer on the stack
; and save a reference at ptr
%ptr = alloca i32*, align 8
```

The analog of the `alloca` instruction in a pointer analysis is the address-of operation $x = \&a$, since the top-level variable *ptr* points to the abstract memory location holding the actual value. This might come as a surprise, since the implementation of $x = \&a$ in C does not result in an `alloca` instruction in the LLVM-IR. This often leads to confusion when interpreting points-to results that are built using the LLVM-IR. Nonetheless the interpretation of the `alloca` instruction as an address-of pointer constraint is in line with the constraints of the Andersen algorithm.

Load Instruction The LLVM `load` instruction loads the value of a top-level variable into a new variable. Compared to the C programming language this is comparable to dereferencing a pointer variable.

```
%ptr = alloca i32*, align 8
; load the pointer to the 32 bit integer from ptr into %var
%var = load i32*, i32** %ptr, align 8
```

When interpreting LLVM load instructions for a pointer analysis, we can treat it as the complex load constraint that is already defined for Andersen's analysis in Table 1.1. Again, just like the `alloca` instruction, we can not draw a direct comparison between the definition of the constraint $x = *y$ and the equivalent statement in the C language, since we are working with the LLVM-IR in SSA form. The C equivalent of the `load` instruction would be the dereferencing of $*y$ alone. The assignment to x would require another `store` instruction.

Store Instruction The LLVM `store` instruction stores a value into a variable. The variable might be a top-level or an abstract memory object represented by an address-taken variable.

```
%ptr = alloca i32*, align 8
%var = load i32*, i32** %ptr, align 8
; store the literal value 10 into %var
store i32 10, i32* %var, align 4
```

The interpretation of the LLVM store instruction is similar to the load instruction. It can be interpreted as the complex store constraint defined as part of Andersen's analysis.

Getelementptr Instruction The LLVM `getelementptr` or `gep` instruction is used to get subelements from an aggregate data structure such as arrays or structs. When the gep instruction is invoked, it requires an index in order to perform the address calculation for the subelement. `Getelementptr` does not access memory, it only finds the correct address given a variable and an index.

```
int arr[10];
arr[5] = 10;

; create an integer array
%arr = alloca [10 x i32], align 16
; get the address for the 5th element
%idx = getelementptr inbounds [10 x i32], [10 x i32]* %arr,
    i64 0, i64 5
store i32 10, i32* %idx, align 4
```

Since we intend to run a static analysis, we need to consider the properties of the index value passed into the `getelementptr` instruction. Either the index is a constant value, and we can forward the `gep` instruction together with the constant index into the static analysis, or the index is a variable that is subject to change during execution, and we need to assume that every possible value can be realized during execution, i.e. every offset for an aggregate data structure might be referenced by such a `gep` instruction and this information must be passed to the static analysis. This behavior is handled by differentiating between two types of `gep` constraints in the pointer analysis, normal- and variant-gep constraints. The former specifying a singular offset and the latter every possible offset for a given aggregate data type, since the value can not be statically determined.

Copy Instruction and equivalent Instructions Other than the complex store, load and the address-of constraints, the Andersen inclusion-based pointer analysis operates also on simple constraints, see line 2 in Table 1.1. In terms of LLVM-IR instructions, we are only interested in those instructions that manipulate pointers. In the LLVM-IR specification lots of instructions can result in values being moved, including pointers. Therefore, we group those instructions that can move values between symbols under the simple inclusion-based copy constraint. The following instructions are interpreted as simple copy constraints:

- `Phi` instructions are part of the LLVM-IR to correctly resolve control flow. Depending on the conditional branch a new SSA variable is introduced that holds the resulting value from the branch. Since we are implementing a flow-insensitive pointer analysis we do not differentiate between control flows and simply interpret each phi instruction as a simple constraint connecting the conditional values to the resulting phi variable.

- `Select` instructions serve a similar purpose as `phi` instructions. Here a value is selected conditionally without creating branches. As such we interpret the `select` instruction as a simple constraint.

- `Call` instructions represent a function call. If the function call contains arguments the caller arguments and callee parameters need to be interpreted as a simple constraint. Beyond this, nothing is done to analyze the context of the function call, since we are performing a context-insensitive pointer analysis.

- Like the call instructions, `ret` instructions are simply interpreted as simple constraints so the returned function value is included in the pointer analysis.

- `ThreadFork` `ThreadJoin` are not LLVM-IR instructions. But just like the `call` and `ret` instructions, these must be interpreted as function calls with simple constraints in the context of pointer analysis.

- Lastly there are various instructions that are straightforward to interpret as simple constraints, including `bitcast`, `ptrtoint`, `constexpr`, `extractvalue` and `freeze` instructions. Furthermore, variable argument values and external library call parameters need to be handled like regular `call` instructions.

Now with `alloca`, `load` and `store` instructions introduced, we can analyze a basic example by applying Andersen constraints to perform a pointer analysis. We are going to observe a simple c program, which will compile into the following LLVM-IR.

```
int *p, x, *q;          %1 = alloca i32*, align 8
p = &x;                 %2 = alloca i32, align 4
q = p;                  %3 = alloca i32*, align 8
                        store i32* %2, i32** %1, align 8
                        %4 = load i32*, i32** %1, align 8
                        store i32* %4, i32** %3, align 8
```

An example for applying the constraint rules according to the Andersen algorithm in Table 1.1 can be observed in Table 1.3. The edge labels of the constraint graph are `p`, `s`, `l`, `c` corresponding to points-to (inverse alloca), store, load and copy constraints. Furthermore, the copy edges are immediately converted into points-to edges in order to simplify the constraint graph in this example and reduce the number of steps.

1.4 Context-free Languages

Looking at the problem definition for an Andersen style pointer analysis, most algorithms in the literature operate on graph data structures. In this section we will look at an alternative interpretation of the pointer analysis problem where the problem can be solved by transforming it into a graph-reachability problem which can be solved by using context-free languages, i.e. CFL-reachability. This approach was first introduced for static analysis purposes by [Rep98] and was found to be solvable in cubic time.

1.4.1 Definition of Context-free Languages and Grammars

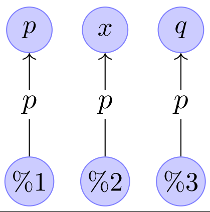
A context-free language is a language that can be generated by a context-free grammar. A context-free grammar is a 4-tuple $CFG = (V, \Sigma, R, S)$ that holds a finite set of nonterminal characters V, a finite set of terminal characters Σ, a finite relation (rewrite rules) over $V \times (V \cup \Sigma)^*$ and a start variable $S \in V$.

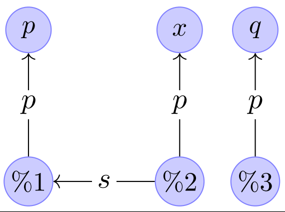
A prominent example for context-free languages is the balanced parenthesis language, which is defined by:

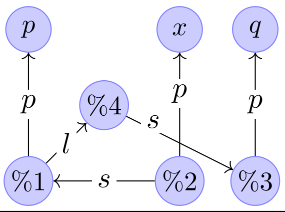
1. $CFG_{bpar} = (V, \Sigma, R, S)$
2. $V = \{S\}$
3. $S = S$
4. $\Sigma = \{(,)\}$

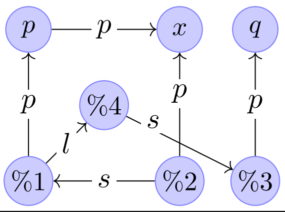
Table 1.3: Example for Applying Andersen Constraints

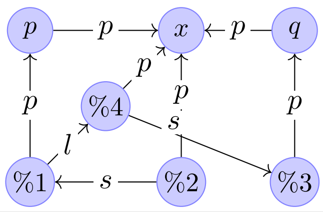
Code Snippet to analyze / comment	Constraint Graph

`%1 = alloca i32*, align 8`
`%2 = alloca i32, align 4`
`%3 = alloca i32*, align 8`
	

`store i32* %2, i32** %1, align 8`	

`%4 = load i32*, i32** %1, align 8`
`store i32* %4, i32** %3, align 8`
`; create constraints from every`
`; instruction`
	

`; apply first store`
`; constraint rule(s)`
	

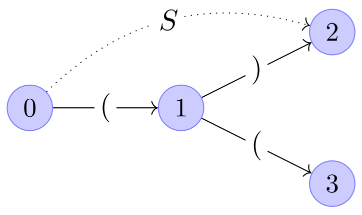
`; apply last load and store`
`; constraint rule(s)`
`;`
`; finally p pts-to x`
`; and q also pts-to x`
	

5. $R = \{S \rightarrow \epsilon, S \rightarrow SS, S \rightarrow (S)\}$

Given the grammar CFG_{bpar} the language $\mathcal{L}_{bpar}(CFG_{bpar})$ is defined as all words that can be generated by the grammar or formally:

$$\mathcal{L}_{bpar}(CFG_{bpar}) = \{w \in \Sigma^* | S \Rightarrow^*_{CFG_{bpar}} w\}$$

Words that are part of $\mathcal{L}_{bpar}$ are for example `(())` or `(()())()` - in general all parenthesis that are opened need to be closed at some point in a word. If we apply the concept of context-free languages to graphs, we can easily define a reachability relation based on context-free languages by interpreting edges as terminal characters and paths of edges, or concatenations of edge labels, as words - as long as the edge labels of a given graph are part of the alphabet for the context-free grammar. Given the following graph G:



The CFL-reachability defined by the grammar CFG_{bpar} would find node 2 to be reachable by node 0, since the word "()" is part of $\mathcal{L}_{bpar}$. This computed reachability can be saved in the graph, by inserting a new edge from node 0 to node 2 with the nonterminal label S, representing the balanced parenthesis property.

1.4.2 Andersen Analysis via CFL-Reachability

With the knowledge about the Andersen constraints, a set of logical base facts along with terminal characters can be defined for each constraint as can be seen in Table 1.4. Note that each terminal character x also has an inverse representation $\bar{x}$ which represents its inverse edge in the constraint graph. With these base facts one can define horn-clause rules for a points-to relation [Rep98]. These horn-clauses can then be reinterpreted as a context-free grammar via the respective production rules that represent an instance of the Andersen pointer analysis problem, see Table 1.5.

$$R = \{P \rightarrow \bar{a}, C \rightarrow c, P \rightarrow aC, C \rightarrow al, C \rightarrow sP\}$$

$$CFG_{ander} = (\{P, C\}, \{a, c, l, s, \bar{a}, \bar{c}, \bar{l}, \bar{s}\}, R, P)$$

Statement	Name	Base Fact	Terminal Character
$x = \&a$	alloca	alloca(a,x)	a
$x = y$	copy	simpleCopy(y,x)	c
$x = *y$	load	complexLoad(y,x)	l
$*x = y$	store	complexStore(y,x)	s
$x = y.f$	field	simpleField<N>(y,x)	f<N>

Table 1.4: Context-free Grammar Terminals and Base Facts for each Andersen Constraint. Field-sensitive base facts are implemented by a template e.g. one for each possible offset value N.

With the context-free grammar, the points-to information can then be computed by calculating the transitive closure of the points-to production rule P. In general a variable x points to a variable y iff a path $x \rightsquigarrow y$ exists such that the word created by the ordered edge labels of the path is in the language $\mathcal{L}(CFG_{ander})$ defined by the grammar. In fact, most interprocedural static analyses can be implemented this way by using a context-free grammar. One of the first times this approach was mentioned in the context of a pointer analysis was [ZR08] where context-free reachability was used to implement a demand-driven flow-insensitive alias analysis. This proved to be a faster and more resource efficient approach compared to other demand-driven alias analyses at the time.

1.4.3 Context-free Path Queries via Matrix Multiplications

While the transformation of the Andersen problem statement into a reachability relation on top of a context-free language is a helpful mathematical abstraction, it does nothing to improve performance, scalability or precision of the whole program Andersen algorithm. Subsequently, [AG18] proposed a matrix multiplication based approach that works with graph data and allows path queries by means of context-free grammars. The general answer to a path query is a set of triples of the form (P, a, b), such that there exists a path in the graph from node a to node b with a path labeling derived from the nonterminal P according to [AG18]. The central idea for the algorithm is to solve these context-free path queries by calculating the transitive closure of matrices. These matrices are constructed by capturing the adjacency matrices of each individual terminal edge label from the graph. Given two of these boolean adjacency matrices, we can compute the transitive closure by repeatedly multiplying them until no more changes are applied. To simplify these operations we can convert our context-free grammar that we derived from the Andersen constraints into Chomsky normal form. After the conversion of a grammar $G = (V, \Sigma, R, S)$ into Chomsky

Statement	Horn-clause rule	Production
$x = \&a$	pointsTo(x,a):- alloca(a,x)	$\{P \to \bar{a}\}$
$x = y$	pointsTo(x,a):- simpleCopy(y,x), pointsTo(y,a)	$\{P \to \bar{c}P\}$
$x = *y$	pointsTo(x,a):- complexLoad(y,x), pointsTo(y,z), pointsTo(z,a)	$\{P \to \bar{l}PP\}$
$*x = y$	pointsTo(a,b):- complexStore(y,x), pointsTo(x,a), pointsTo(y,b)	$\{P \to \bar{P}\bar{s}P\}$

Table 1.5: Context-free Grammar Productions and Horn-clauses for each Andersen Constraint.
Field-sensitive rules are omitted for simplicity.

normal form we are left with productions in the following form:

$$P \to AB, \qquad \text{for } P, A, B \in V \tag{1.4}$$

$$P \to v, \qquad \text{for } P \in V \wedge v \in \Sigma \tag{1.5}$$

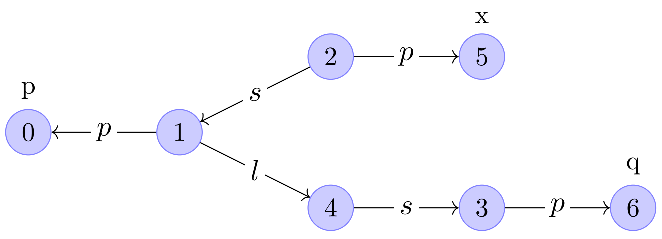
Following this, we iterate through all production rules in the form of Equation 1.5 and populate the adjacency matrices, then we iterate through all production rules in the form of Equation 1.4 and apply the matrix multiplications to find the transitive closures for each of these productions. An elaborate proof for the correctness of this approach can be found in [AG18]. Note that the cited paper uses a singular matrix containing sets for all edge relations between nodes instead of multiple boolean matrices for each edge type. This does not have any effect on the correctness of the algorithm and simply improves the performance, since boolean matrices can be represented more efficiently than matrices of sets. By rearranging the Andersen algorithm as repeated matrix multiplications, we indirectly profit from decades of research into efficient linear algebra algorithms. In this case we utilize a subset of linear algebra algorithms, specifically sparse boolean linear algebra. Whenever linear algebra is involved, it is often a good idea to use accelerators such as GPGPUs to speed up the calculations.

Next we will consider a basic pointer analysis example to illustrate the context-free path query approach that uses matrices for the calculation. The input constraint graph that we will use as an input for the Andersen style analysis is derived from the following c code:

```
int main()
{
    int *p, x, *q;
    p = &x;
    q = p;
}
```

```
define dso_local i32 @main() #0 {
    %1 = alloca i32*, align 8 ; p
    %2 = alloca i32, align 4 ; x
    %3 = alloca i32*, align 8 ; q
    store i32* %2, i32** %1, align 8
    %4 = load i32*, i32** %1, align 8
    store i32* %4, i32** %3, align 8
    ret i32 0
}
```

Which results in the following graph.



This is the same code snippet from Table 1.3, compiled into LLMV-IR and interpreted into an Andersen constraint graph.

If we take the context-free grammar we defined for an Andersen style analysis from subsection 1.4.2 and convert it into Chomsky normal form, we arrive at the following production rules and grammar.

$$R = \{P \to p, C \to c, S \to s, L \to l, P \to \bar{C}P, C \to SP, C \to \bar{P}L\}$$

$$CFG_{ander} = (\{P, C, L, S\}, \{a, c, l, s, \}, R, P)$$

Following this we iterate through all production rules of the form Equation 1.5 and create/fill boolean adjacency matrices for each nonterminal with the associated edges from the graph, where an edge (a, b, l) from node a to node b with label l is represented in the matrix element $L_{a,b}$.

$$P_{7\times7} = \begin{bmatrix} 0&0&0&0&0&0&0\\ 1&0&0&0&0&0&0\\ 0&0&0&0&0&1&0\\ 0&0&0&0&0&0&1\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0 \end{bmatrix} C_{7\times7} = \begin{bmatrix} 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0 \end{bmatrix}$$

$$L_{7\times7} = \begin{bmatrix} 0&0&0&0&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0 \end{bmatrix} S_{7\times7} = \begin{bmatrix} 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&1&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0 \end{bmatrix}$$

With these initial matrices in place, we now iterate over all production rules in the form of $P \to AB$, for $P, A, B \in V$, see Equation 1.4, which we apply by multiplying the corresponding nonterminal matrices. Note that we use the inverse matrix if we encounter a negated nonterminal. Looking at $C \to SP$, one of these production rules for Andersen's analysis, we update the $C_{7\times7}$ matrix as follows

$$C_{7\times7} = C_{7\times7} + S_{7\times7} \times P_{7\times7}$$

$$C_{7\times7} \begin{bmatrix} 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 1&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0 \end{bmatrix} = C_{7\times7} + S_{7\times7} \begin{matrix} 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&1&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0 \end{matrix} \times P_{7\times7} \begin{matrix} 0&0&0&0&0&0&0\\ 1&0&0&0&0&0&0\\ 0&0&0&0&0&1&0\\ 0&0&0&0&0&0&1\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0 \end{matrix}$$

Here a copy relation was added from node 2 to node 0. We repeat this operation for all these production rules until no more changes take place in the matrices. This can be done efficiently by storing the number of non-zero elements, nnz., for each matrix and only multiplying for a given production rule, if either of the RHS matrices have changed nnz values. Finally we arrive at the following matrix for the P nonterminal corresponding to the points-to relation:

$$P_{7\times 7} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Fortunately this is the same result we arrived at when manually applying the Andersen constraints in Table 1.3.

Since the transformation of the context-free grammar into a normal form can lead to a substantial increase of required matrices, [OEAG20] introduced a context-free path querying algorithm that utilizes the Kronecker product to realize recursive finite state machines which in turn do not require normalized grammars and can operate on the original context-free grammar for Andersen style pointer analysis. Unfortunately the Kronecker calculation is much more complex than conventional matrix multiplications and thus this convention did not yield performance improvements.

Further research by a research team collaborating with developers from JetBrains, a developer of integrated development environments, found that for synthetic graph data, GPU accelerated sparse boolean linear algebra were a promising method for context-free path queries [MSS+19]. Furthermore, this research lead to a software library designed specifically for sparse boolean linear algebra, `spbla` [OKKG21]. This library utilizes GPGPUs either through CUDA or OpenCL to accelerate boolean linear algebra operations.

Unfortunately the current state of the spbla project's[3] codebase contains errors for which reason the idea of using context-free path queries via matrix multiplications for solving Andersen's analysis was abandoned in favor of a more direct CUDA based implementation, see chapter 2. During the writing of this thesis another library with a focus on static analysis, SVF[4], introduced support for context-free path queries [LSDZ22]. Although the software developed as part of this thesis makes heavy use of the SVF library in chapter 2, standalone context-free path querying approaches for solving pointer analyses were not used during development as the support was introduced in SVF during the late development of PTAGPU.

[3]`https://github.com/JetBrains-Research/spbla`
[4]`https://github.com/SVF-tools/SVF`

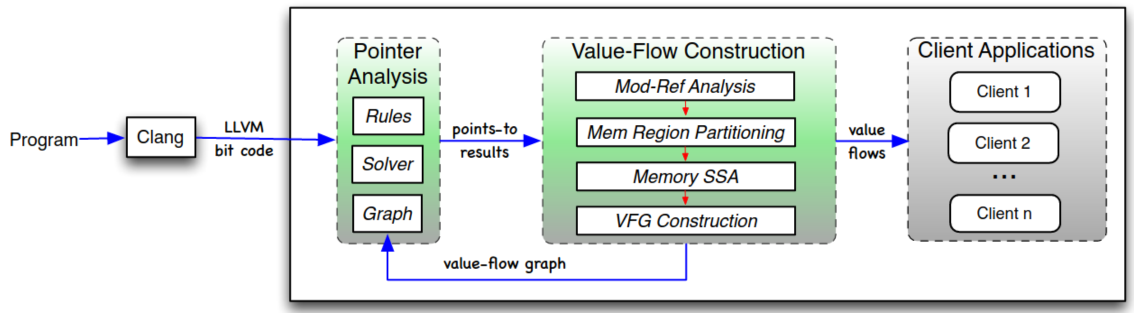


Figure 1.2: Overview of the SVF library from [SX16]

1.5 Related Work

1.5.1 SVF

As mentioned in section 1.3, a pointer analysis is the basis for many types of static analyses. One such analysis framework is SVF[5], a project that aims to enable scalable and precise interprocedural static value-flow analysis [SX16]. The SVF tool consists of a number of subcomponents that represent individual analysis use cases, such as use-after-free and source-sink error detection [SYX14b], whole program pointer analysis and on-demand value-flow analysis [SX18] to name a few. SVF as a framework also allows users to extend and implement custom analysis solutions that build on top of the subcomponents that make up SVF and LLVM, which serves as a base and data source for SVF, see Figure 1.2. This gives SVF a degree of modularity that other current static analysis frameworks, such as [SXW+18], often lack.

Currently, SVF requires version 14 of the LLVM project and makes extensive use of internal data structures. At the core of the SVF analysis tools is the LLVM-IR which is used to derive information from the source code of a program. Internally this intermediate representation is augmented into the SVF-IR, a graph data structure that models the program assignment graph, short PAG. The PAG is a mostly immutable graph that contains all program instructions and incorporates a model for the memory SSA. All further analyses use the PAG as a root of information from which the analysis results are derived. See Figure 1.3 for the resulting PAG of the example program in Table 1.3.

The typical analysis run with SVF starts off with a pointer analysis that first derives a constraint graph from the PAG, similar to the procedure described in subsection 1.3.2. The constraint graph is then fed into a pointer analysis algorithm such as the state-of-the-art interprocedural Andersen style Wave Propagation algorithm, see Algorithm 1. The result

[5]`http://svf-tools.github.io/SVF`

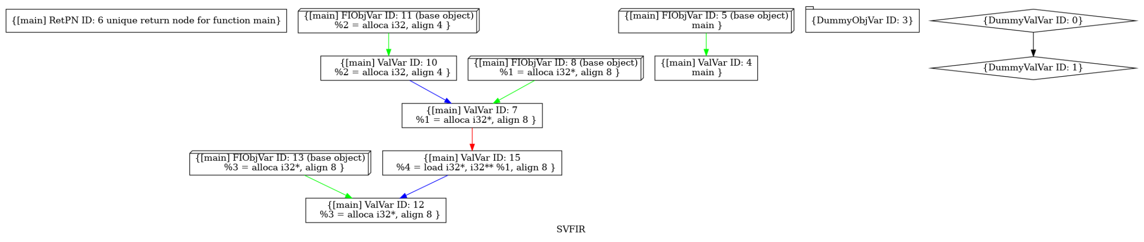

{[main] RetPN ID: 6 unique return node for function main}
{[main] FIObjVar ID: 11 (base object) %2 = alloca i32, align 4 }
{[main] FIObjVar ID: 5 (base object) main }
{DummyObjVar ID: 3}
{DummyValVar ID: 0}
{[main] ValVar ID: 10 %2 = alloca i32, align 4 }
{[main] FIObjVar ID: 8 (base object) %1 = alloca i32*, align 8 }
{[main] ValVar ID: 4 main }
{DummyValVar ID: 1}
{[main] ValVar ID: 7 %1 = alloca i32*, align 8 }
{[main] FIObjVar ID: 13 (base object) %3 = alloca i32*, align 8 }
{[main] ValVar ID: 15 %4 = load i32*, i32** %1, align 8 }
{[main] ValVar ID: 12 %3 = alloca i32*, align 8 }
SVFIR

Figure 1.3: The PAG for the Example from Table 1.3 computed by SVF. Green edges representing an alloca instruction, blue edges stores and red edges loads.

is an over-approximated points-to set for each top-level variable and address-taken variable, as well as a call-graph, that represents an over-approximation for each direct and indirect function call. Given the points-to information, SVF performs an initial mod-ref analysis run to find interprocedural side effects of variables. Given mod-ref and pointer information, defs and uses are then annotated in each procedure with alias sets of abstract memory objects that might be accessed indirectly by loads and stores. SVF also allows users to specify a memory partitioning strategy whereby the heap can be partitioned with varying granularity to allow for precision and scalability trade-offs. The properties that are of interest here are specifically the def-use chains of address-taken variables of pointers, which are difficult to compute compared to the def-use chains of top-level variables which are already available if the program is in SSA form [SX16]. The difficulty arises from the seeming ambiguity of indirect accesses to address-taken variables through loads and stores in the program. After connecting defs and uses of variables, the result is a value-flow graph, or sparse value-flow graph depending on the specified analysis details, that can then be used for more concrete analyses, e.g. use-after-free memory leak detection [SYX14b] or fed back into a more precise pointer analysis algorithm to increase precision with the gained value-flow information.

This thesis focuses on pointer analysis, thus, the details of pointer analyses inside SVF are especially relevant. SVF provides a wide set of pointer analyses to choose from, see Figure 1.4 taken from SVF's technical documentation[6]. The components of each implementation can be split into three groups, the Graph, which is a data structure derived from the PAG, that describes where the pointer analysis should be performed. A set of Rules, which dictate how points-to information should be derived from each statement in the Graph. And a Solver which dictates the order in which the Rules are to be applied on the Graph, according to [SX16]. A user can then choose which components to reuse and which to replace or augment with custom algorithms or data structures. This modular approach makes it convenient to experiment with different in-memory representations of points-to data, as well as testing different methods for solving the constraint graph. The software developed in chapter 2

[6]`https://github.com/svf-tools/SVF/wiki/Technical-documentation`

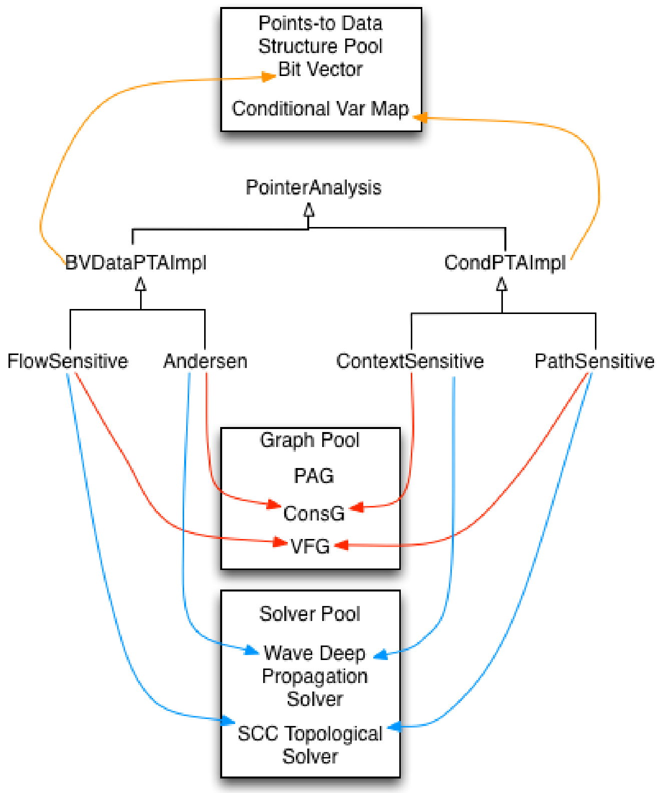


Figure 1.4: The class hierarchy of pointer analysis implementations in SVF.

will use SVF as a basis for the pointer analysis and reuse parts of SVF during the analysis.

1.5.2 Graspan

Graspan is a disk-based parallel graph system designed for computing transitive closures on very large graphs defined by context-free grammars. It was first introduced by [WHZ+17] with a CPU backend and later in [ZWH+21] with a GPU-based backend. Disk-based means, that a given constraint-graph is divided into smaller subgraphs, called partitions, that are stored on non-volatile memory and loaded into memory in pairs to calculate the transitive closures in steps that can lead to the desired pointer information - or any other static analysis solution that can be defined by a context-free grammar, see subsection 1.4.2. This approach is similar to many of the design decisions taken by common "BigData" solutions, such as Apache Kafka and Spark, where disk-based solutions are common.

Graspan is meant to be run on a single machine, hence the emphasis on saving memory by offloading to the disk. Another implementation that improves upon the ideas from the Graspan paper is [GZJ+20] where the computation is distributed across multiple nodes, making use of other "BigData" concepts.

Overall Graspan was able to perform a pointer analysis for the Linux kernel in 10.9 minutes with the GPU version [ZWH+21], while other tools fail to perform on graphs as large as the Linux kernel's constraint graph, which contained 250 million edges after preprocessing and inlining in Graspan. It should be noted that this result was achieved with a flow- and field-insensitive context-free grammar.

In terms of data structures, Graspan has specific requirements. The graph data is large, sparse and dynamic, consequently Graspan uses different in-memory representations for the graph in the CPU and GPU version. The CPU version uses two arrays of $(dst, label)$ pairs that represent the old and new outgoing edges for each node while the GPU version uses a sparse bit vector representation that, while containing the same information as the CPU arrays, is optimized for GPU SIMT parallelism. See Figure 1.5 for an illustration of the GPU optimized sparse bit vector data structure from [ZWH+21]. Internally this GPU optimized data structure is inspired by prior work from [MBP12] which will also be used in chapter 2 to develop PTAGPU.

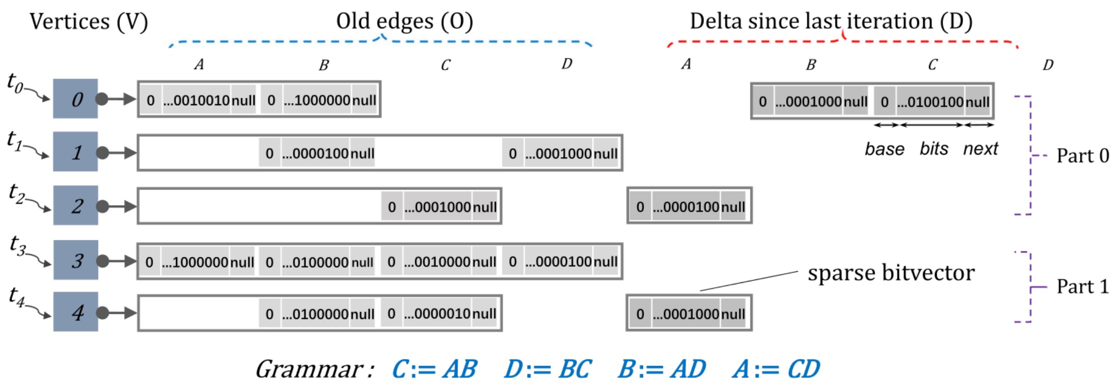


Figure 1.5: Data structure for the constraint graph in the Graspan GPU version, from [ZWH+21]

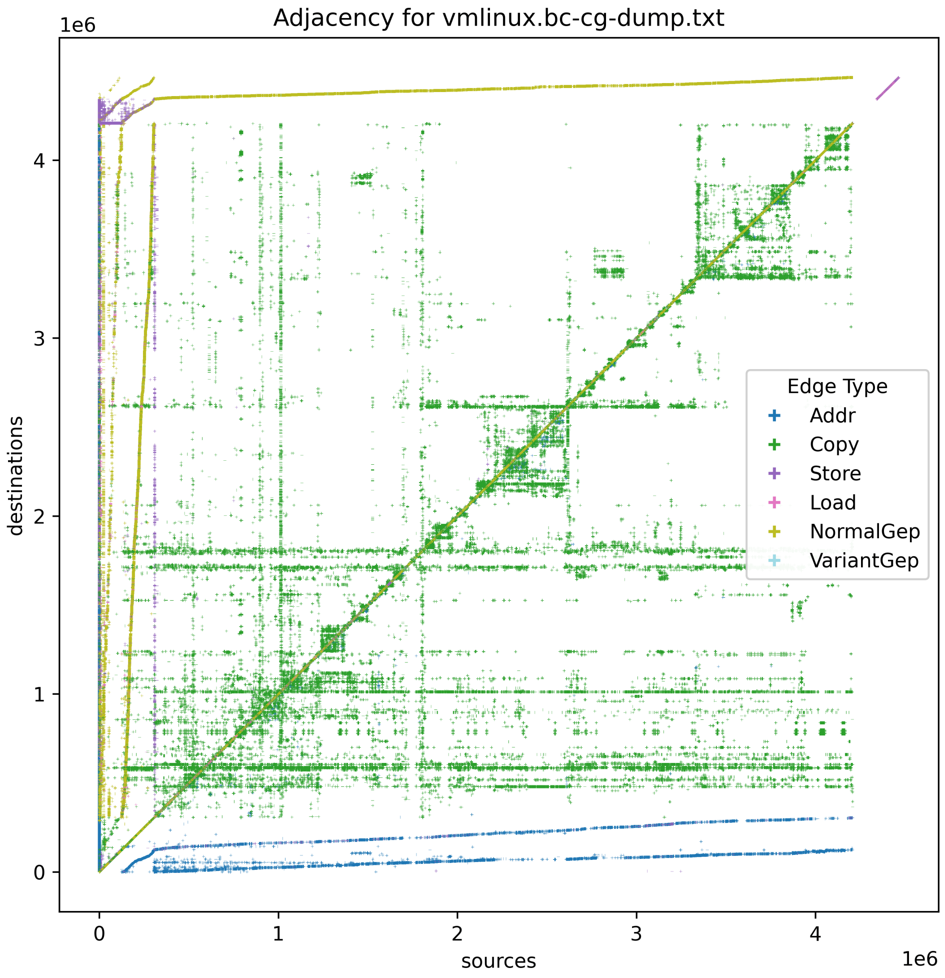


Figure 1.6: Adjacency Plot for the Constraint Graph of the Linux Kernel

Chapter 2

PTAGPU

This thesis presents a software library named PTAGPU, the name is derived from pointer analysis (PTA) and graphics processing unit (GPU). As the name suggests the core idea is to use GPUs for the purpose of performing a pointer analysis. The library PTAGPU was developed as a whole program analysis module inside the SVF framework which is in turn built on top of the LLVM compiler system.

2.1 Integrating PTAGPU into SVF

As described in subsection 1.5.1 the SVF framework is capable of processing the LLVM-IR of a compiled program and capture the individual LLVM-IR instructions in a program assignment graph, which can then be used to perform a pointer analysis on the program. When SVF is launched for the purpose of a whole program analysis, the program assignment graph is further processed into a constraint graph that holds all relevant constraints for an initial pointer analysis, see Table 1.1. At this point the constraint graph is passed into a class that inherits from the `PointerAnalysis` class, see Figure 1.4 for an overview of the pointer analysis class hierarchy in SVF. Since our goal is to implement a custom pointer analysis, we can inject out own implementation at this stage as a `PointerAnalysis` subclass. Specifically we inherit from the `Andersen` class which is itself a subclass of the `PointerAnalysis` class that implements the Andersen inclusion-based pointer analysis algorithm in SVF. By extensive use of runtime polymorphism, most of SVF is implemented via virtual member functions - a construct specific to C++ - allowing for function overriding in subclasses. This makes implementing a custom pointer analysis easy as we can reuse most of the initialization steps and program assignment graph processing from the superclasses. Part of the processing is an initial topological sorting and the previously mentioned interpretation of the LLVM-IR instructions into a constraint graph. As a result we end up with a constraint graph and all strongly connected components of said graph when we initialize our PTAGPU class that inherits from the `Andersen` class. The in-memory representation of constraint graphs or generic graphs in SVF is more akin to a linked list data structure where each node carries references to all outgoing and incoming edges and those edges carry references to source and destination nodes as well as auxiliary information, which is an ideal memory model for iterative algorithms such as the default Andersen algorithm where the algorithm works from one node to the next. Unfortunately this memory model is not ideal for parallel processing. For this reason we initially reinterpret the constraint graph into a more fitting data structure. While iterating through the entire constraint graph, we differentiate by edge types and collect all (src, dst) edge pairs in standard library vectors. If we refer back to the design principles of SVF in subsection 1.5.1, where pointer analyses were conceptually split into three components, the Rules, the Graph and the Solver, we effectively implement our own Graph component by using a different linear memory representation for the constraint graph. The underlying goal of putting the constraint graph into a linear in-memory data structure is to allow us to more easily copy the memory region containing the relevant information into GPU device memory, which is where our pointer analysis will operate on the data. Similar to the Graph component, we also modify the Solver and Rule components in the custom analysis implementation. The details of the implementation will be described in detail in section 2.3.

2.2 Goal of the Algorithm

Our goal is to use the provided program assignment graph from SVF and the derived constraint graph to compute a points-to set for each pointer variable in the program. Overall our algorithm is supposed to serve as an initial pointer analysis pass in the SVF framework. Together with the pointer information we can then proceed to build an over-approximated call-graph. One might assume that no pointer information is needed for SVF to build a call-graph for a given program. Unfortunately indirect function invocations, where functions are indirectly accessed via pointers, require us to build pointer information in order to create an over-approximated and soundy call-graph. The over-approximate nature stems from the imprecision of Andersen's analysis. This limitation always remains, no matter the algorithm used for the pointer analysis, since pointer analyses are fundamentally undecidable as was mentioned in section 1.3. It is important to consider that the initial pointer analysis does not directly produce much valuable information for static analysis purposes. Instead, we use the pointer information produced by the initial analysis for the call-graph which is then used to refine the analysis result by applying a more precise flow- and/or context-sensitive analysis that produces the relevant results, which can then be used to derive value-flow information that can directly be used for actual analysis purposes. Since the SVF analysis framework aims to apply a more precise pointer analysis at a later stage, one might argue that it would be wise to start off with such an analysis. For small programs this is a viable strategy, unfortunately these more precise analyses are currently not scalable for a whole program analysis of larger software and are only applied on-demand [SX16]. This is also one of the motivations for trying to accelerate the initial Andersen analysis specifically since it is performed on the entire program and thus can in theory profit from the parallelism of GPUs. Concluding, PTAGPU is intended to serve as an initial whole program pointer analysis in SVF. The results of PTAGPU can then be used by subsequent analyses to improve the precision of the points-to data.

2.3 Design of the Algorithm

The PTAGPU library uses CUDA[1], an application programming interface language from NVIDIA, to program GPUs in C++. CUDA accommodates developers with a collection of abstractions that simplify operations with GPU compute and memory. NVIDIA also provides a standalone library for common parallel operations such as sorting and transformations named Thrust[2] which is also employed by PTAGPU for common sorting and deduplication operations. In principle all calculations that are executed with a GPU are denoted as kernels in CUDA. While GPU kernels and CPU code can be shared, CUDA provides some intrinsic operations that can only function when executed on a GPU. Likewise, all CPU operations that rely on external libraries or non standard-library code are not supported in GPU kernels.

2.3.1 CUDA Architecture

The CUDA programming model revolves around blocks of threads. Whenever a kernel is launched on the GPU, a set number of blocks is specified to work on the kernel. Each block of a kernel executes the same code with the same number of threads. Thread blocks are further divided into Warps, which are a collection of 32 threads each. This type of parallel processing is called SIMT, for single instruction, multiple threads. In reality a Warp is more analogous to a single vectorized operation that executes 32 units of work or lanes at once than a collection of individual threads, although each thread in a Warp has its

[1]`https://developer.nvidia.com/cuda-toolkit`
[2]`https://docs.nvidia.com/cuda/thrust/index.html`

own instruction address counter. Having instruction address counters per thread allows independent branching, resulting in thread divergence. The divergence is implemented by executing each conditional branch sequentially for all threads in a Warp and disabling the threads that do not execute the specific branch. This way each thread in a Warp always executes the same instruction if active. Since branches are not executed in parallel, thread divergence is discouraged if possible for performance reasons. These specifications of Warps are uniform in all CUDA hardware and resemble a single group of work that can be scheduled on the device specific number of multiprocessors. To more easily differentiate between different generations of hardware, the CUDA programming model is segmented into tiers of compute capabilities, where newer hardware with newer capabilities receives a higher compute capability. Each CUDA capable device has a number of streaming multiprocessors, in short SMs, which themselves have a certain amount of L1 shared memory and number of registers per core among other resources. While this is similar to how CPU cores operate, GPU programming uses a flat memory hierarchy with less reliance on caching and more on raw memory bandwidth. For this reason each core in an SM has a relatively large register file so that a single CUDA thread commonly uses hundreds of registers. The entire work that is to be performed by a single kernel is called a grid, which is divided evenly into blocks, which are divided into Warps. Both the grid and each block can be indexed in up to three dimensions, which is useful for working with shaders, global illumination rendering and less useful for static analyses. To start a computation, a collection of thread blocks are assigned to the available SMs of a GPU. Depending on the hardware and compute capability, a single SM can execute multiple Warps from the assigned thread blocks in parallel as long as enough resources are available in the SM. The execution order of individual Warps is handled by a Warp scheduler on each of the SMs, that decides what Warps get executed at what time. The purpose of over-provisioning SMs with more blocks/Warps than they can concurrently execute is that when a single Warp executes a memory read/write operation other Warps can execute while the device is busy fetching the data. Key specification for each SM of a certain compute capability are the following:

- Memory Bandwidth per SM
- Total shared memory per SM
- Max number of threads per SM
- Max number of blocks per SM
- Total number of registers of all cores in SM

[3]`https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/`

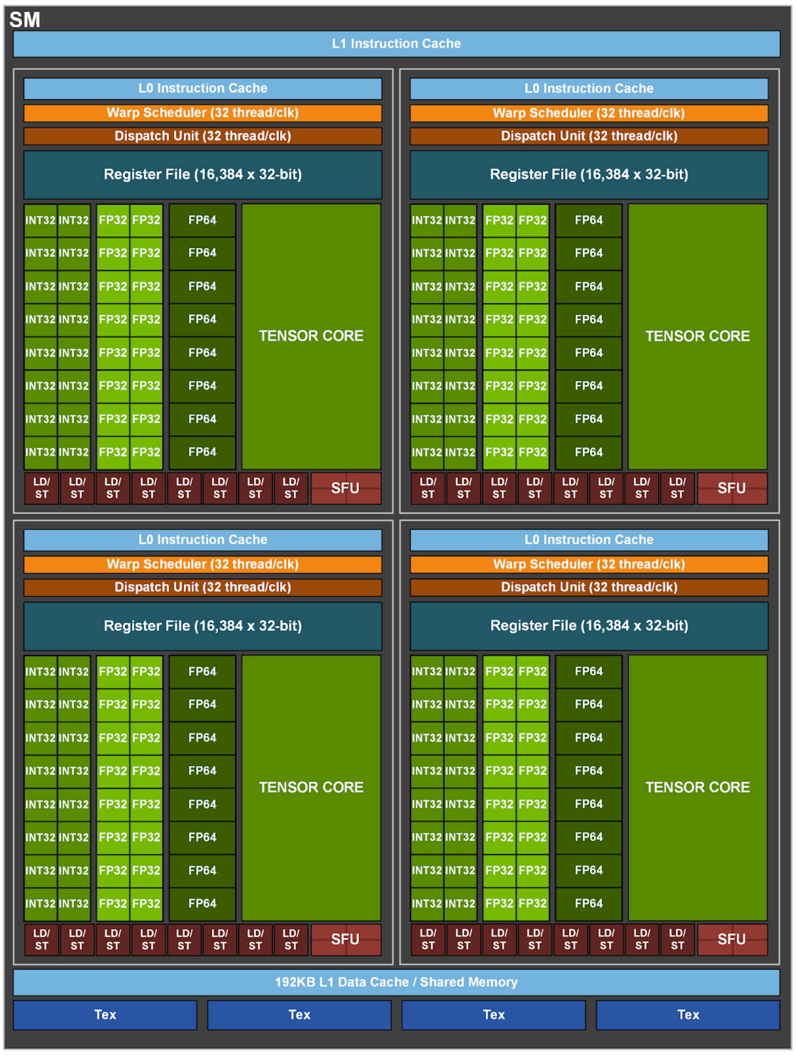

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

Figure 2.1: A single SM of an A100 GPU.
Taken from an NVIDIA Blog Post[3]

Occupancy

Since each SM has multiple limited resources that can be controlled by the developer, namely register count, shared memory and thread count, a kernel has to be designed with these limitations in mind such that a maximally concurrent execution is possible. Given a device with compute capability 8.0 and a kernel that requires 256 registers per thread while also running 256 threads per block, we would essentially limit our execution to a single block per SM, since a device with compute capability 8.0 has 64K available registers per SM. This might be disadvantageous, since the SM is in theory capable of working on up to 1024 threads. We might profit from reducing the amount of registers used per thread and thus increasing the occupancy on our GPU. Counterintuitively high occupancy does not always lead to better performance. Some programs disproportionately profit from the use of a single resource. For example compute intensive kernels require more registers per thread than kernels that are heavily reliant on memory operations. Increasing the use of shared memory does not always lead to a performance improvement and decreasing the number of required registers is not always possible. PTAGPU is very compute intensive, compared to typical GPU algorithms. As a result, a large number of registers are required during computation, which limits the maximal occupancy on the GPU.

Memory Accesses in CUDA

While modern GPUs can in theory perform multiple TFLOPS of calculations per second, effectively all calculations are limited by memory bandwidth. Consequently, how we access the GPU memory is very important for the overall performance of our analysis. Specifically coalesced memory accesses of individual Warps, where consecutive threads access consecutive memory addresses, are important, so the memory read operation can be performed within a single transaction and not multiple strided reads. The GPU memory controller on modern graphics cards can typically execute memory read or write operations in granularities of 32 bytes up to 128 bytes in total. As a result it is for example more efficient to load a single byte of memory per thread in a Warp, compared to loading 8 bytes per thread, since we would exceed the maximum of 128 bytes per memory transaction and require multiple memory accesses. In terms of time needed per instruction, global memory operations are typically two orders of magnitude slower compared to operations on SM registers or shared memory in L1 cache. For this reason CUDA programs can perform exponentially worse if memory accesses are random or strided inefficiently instead of coalesced. PTAGPU uses coalesced memory accesses per Warp to minimize the overhead associated with reading and writing GPU memory.

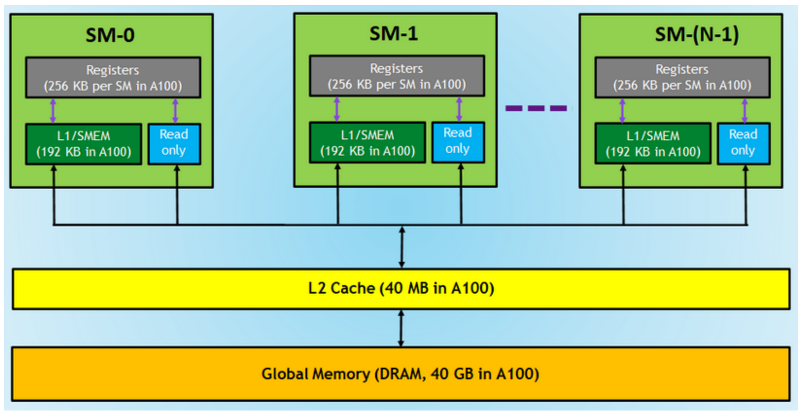


Figure 2.2: Diagram of the CUDA memory architecture for an A100 GPU. Taken from [Pra20].

Unified Memory

When CUDA code is executed on a 64-bit host system, the developer can use a single memory address space for host and device memory. Using this unified memory address space allows memory access from both CPU and GPU in the same address space without explicit memory copy operations. The memory in question is moved implicitly to the device that performs the read operation. The CUDA API also allows the developer to assign preferred residency for specific memory regions, as well as prefetching memory asynchronously for a device. When prefetching is properly employed, unified memory can achieve the same performance as dedicated device memory [Nik16]. The major advantage of unified memory is the fact that the memory allocation size is only limited by system memory, not device memory. This allows a CUDA program to potentially hold vastly more data in memory than would be possible within only GPU memory, while also moving the needed memory regions without much involvement of the developer. Crucially this keeps large data structures intact without splitting and partitioning them for incremental loading into GPU memory. This mirrors some core ideas of Graspan subsection 1.5.2 where parts of the graph are written to disk to conserve memory usage. Since many high performance computing environments have vastly more main memory available than individual GPUs have device memory, this theoretically enables us to expand the scope of our analysis without requiring specialized GPU hardware with more device memory.

CUDA Streams

GPUs are massively parallel. Oftentimes developers do want to perform multiple computations encapsulated in kernels in parallel. These computations require memory operations

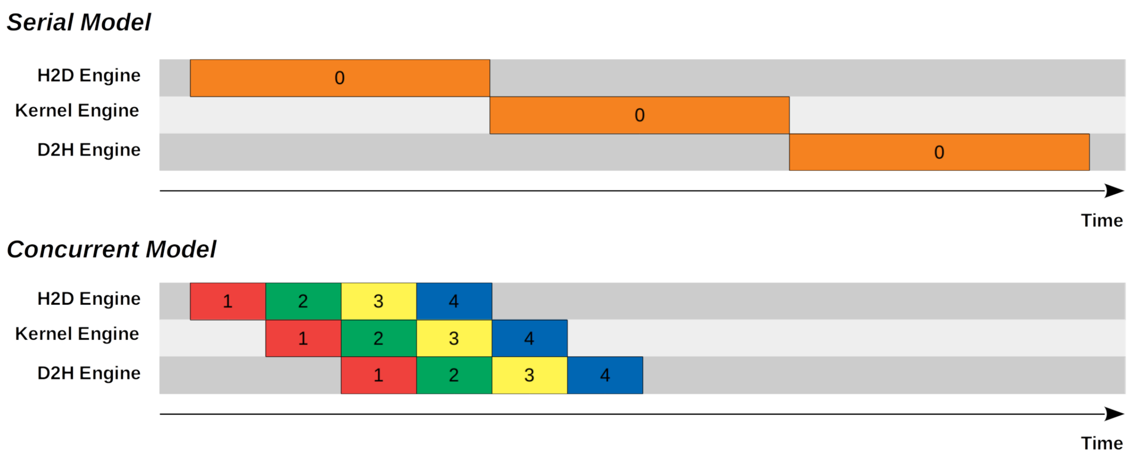
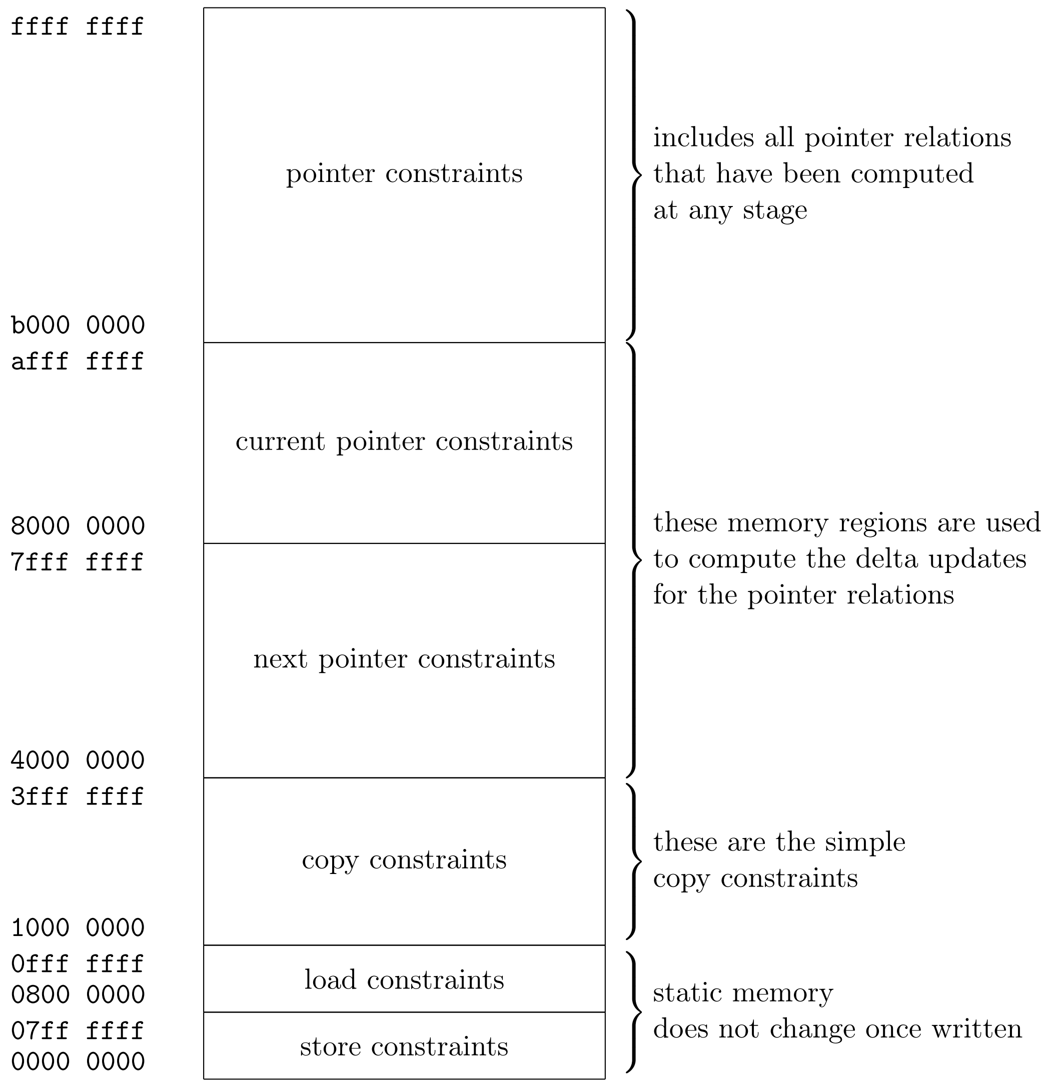


Figure 2.3: CUDA Stream: Serial Model vs Concurrent Model by Lei Mao licensed under CC BY 4.0; an illustration for the advantages of the CUDA stream API.

before and after to load and store the data they operate on. Since most GPUs allow for concurrent memory and compute operations it is desirable to be able to efficiently schedule concurrent operations such that while one kernel executes, another performs memory operations. Because kernels are launched asynchronously, we can utilize CUDA streams to string together multiple compute and memory operations, so that they are executed in order without interfering with other streams. The GPU is then able to efficiently schedule multiple streams in order to maximize GPU utilization, see Figure 2.3 for an illustration. By default, all CUDA operations are executed on the default stream and thus memory operations block each other. Streams have to be created by the developer and are represented by structs in the program. These elements are then passed as arguments to the CUDA API function calls to specify what operation use which stream.

2.3.2 Initialization of CUDA code

As soon as the constraint graph is handed over to the CUDA section of PTAGPU some global state is initialized with the information from the constraint graph. This includes various counter variables for the number of nodes in the graph, as well as the initial unified memory allocations. When the counter for the number of nodes is first initialized the node count is read from the constraint graph and increased by 20 percent to reserve headroom for new nodes that might be added to the constraint graph during the execution of the algorithm. This ensures that each node has a well-defined place in memory that can not be overwritten by subsequent nodes. In the current implementation, the algorithm allocates a fixed amount of unified memory accessible from the CPU and GPU. While this requires over-provisioning of memory for a given analysis without knowing the exact amount needed for the pointer analysis, dynamic allocation proves to be very challenging as it is difficult to deduce the exact required amount of GPU memory from a given constraint graph for a pointer analysis. Furthermore, the allocated unified memory is further split into partitions of fixed size for the individual relations of the Andersen analysis. This includes three memory regions for pointer relations, one for copy relations and two regions for load and store relations between nodes. See the following diagram for the memory layout of an example allocation of 32 GiB of unified memory and the resulting memory regions.


ffff ffff
b000 0000
afff ffff
8000 0000
7fff ffff
4000 0000
3fff ffff
1000 0000
0fff ffff
0800 0000
07ff ffff
0000 0000
pointer constraints
current pointer constraints
next pointer constraints
copy constraints
load constraints
store constraints
includes all pointer relations that have been computed at any stage
these memory regions are used to compute the delta updates for the pointer relations
these are the simple copy constraints
static memory does not change once written

Since a pointer analysis in general does not create new load or store relations in the constraint graph, these memory regions do not increase in size after initialization. Notably the pointer relations are split into three separate memory regions. This is needed for diffpoints calculations. This specific optimization was inspired by [SX16] and [MMP10] among other works and will be explained in detail in section 2.3.5. After memory allocation is completed, CUDA streams are initialized for concurrent write operations into each memory region, as well as one stream for each of the available devices to enable computation on multiple GPUs in parallel. Finally, every bit in the entire allocation of unified memory is set to one by `cudaMemset`. This ensures that the memory is not in an undefined state. Furthermore, we can utilize this when reading from memory, since a memory region with all bits set to one represents unused space.

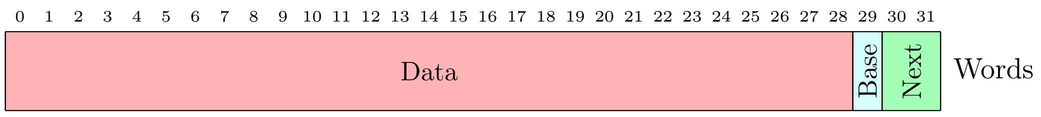
2.3.3 Sparse bit vectors

At the core of PTAGPU all edges of the constraint graph are stored in sparse bit vectors. Sparse bit vectors are a data structure that can be used for storing binary values, such as adjacency information of a graph. Sparse bit vectors are especially suited to represent edges of very sparse graphs and allow for dynamic addition of new edges and nodes, something other sparse graph representation such as the compressed sparse row format do not facilitate. Since the constraint graph of pointer analysis problems is typically very sparse [MBP12], sparse bit vectors are an ideal data structure for pointer analysis on the GPU. Find the adjacency plot for the constraint graph of the Linux kernel in Figure 1.6 as well as on overview for select open source programs and their constraint graph densities in Table 2.2. Individual bit vectors contain three separate components, a base value, a set of bits and a next pointer. For the bit vectors to efficiently work, the entire codomain of the directed edge relation of a graph is split into evenly sized partitions. The edges of each partition, if it contains edges, are then stored in the bits of a bit vector. If we omit empty partitions of the edge codomain, we create a set of sparse bit vectors. The base value of each sparse bit vector represent the current offset for all bits in the bit vector. Similar to linked lists, sparse bit vectors reference the next sparse bit vector in a field. This allows iteration through adjacent outgoing edges of a specific type for a given node by working through the next bit vectors until there is no next bit vector defined. The head bit vector for each node is stored in a pre-defined memory location, which is directly derivable from the node index. The exact position for each node in each memory region can be calculated by multiplying the node index with the width of a single element in a sparse bit vector. This offset is then added to the overall offset of the memory region to find the head bit vector for a specific edge type of a node, see Listing 8 for the implementation in PTAGPU. Following, each unit in the sparse bit vector linked list will be called an element, as is conventional for linked lists. When deciding on the memory characteristics of the individual elements, the traits of the CUDA API and hardware need to be taken into consideration. One possible optimization are coalesced memory accesses, where consecutive threads in a Warp access consecutive memory locations. Furthermore, we can optimize the memory operations of a

```
// src it the node for which we want to find the head bit vector
// rel is the andersen relation for which we want to find the outgoing edges
__host__ __device__ index_t getIndex(uint src, uint rel)
{
    switch (rel)
    {
    case PTS:
        return OFFSET_PTS + (ELEMENT_WIDTH * src);
    case PTS_CURR:
        return OFFSET_PTS_CURR + (ELEMENT_WIDTH * src);
    case PTS_NEXT:
        return OFFSET_PTS_NEXT + (ELEMENT_WIDTH * src);
    case COPY:
        return OFFSET_COPY + (ELEMENT_WIDTH * src);
    case LOAD:
        return OFFSET_LOAD + (ELEMENT_WIDTH * src);
    case STORE:
        return OFFSET_STORE + (ELEMENT_WIDTH * src);
    }
    return src * ELEMENT_WIDTH;
}
```

Listing 8: Calculating the correct index of a node's head bit vector in unified memory.

Warp such that we access 128 bytes in total per Warp, since this allows us to fully saturate the memory controller while only requiring a single coalesced memory transaction [MBP12]. If we take the optimal memory transaction of 128 bytes and split it evenly across all 32 threads of a Warp, we get 4 bytes of memory associated with each thread. As a result we set the size of each element to 128 bytes and each thread in a Warp manages a single word or 4 bytes of the element. Coincidentally an unsigned integer is 4 bytes wide on 64-bit systems, thus each thread accesses its memory in the form of an unsigned integer. Using unsigned integers also has the advantage of allowing for simple bitwise operations, which are required since we store individual bits in the 4 bytes of memory instead of numerical values for most of the words in each element. Numerical values are only stored in the base and next words of the element. Each sparse bit vector element has the following layout.



64-bit Addresses

This sparse bit vector layout is inspired by the layout presented in [MBP12]. The notable difference being a switch from 32-bit to 64-bit addresses to reference the next element in a sparse bitvector. This is a requirement for using an address space larger than 16 GiB, since we are addressing 4-byte wide words. While this reduces the data density of each bit vector by 4 bytes, the theoretical address space of 64 exbibytes, or 2^{66} bytes, is worth the trade-off as it eliminates any upper limits on memory allocation in the software.

Inserting new bit vectors

If our algorithm ever reaches a point where we need to add a new edge into an element with a mismatched base, we are required to create a new element with the correct base and append it to the linked list of bit vectors by referencing the correct memory address in the next field. Since we are operating on multiple nodes concurrently, we need to keep track of the already occupied memory for each of the memory regions representing the different edge types. This is achieved by utilizing some of the counters from the CUDA initialization. These are incremented via atomic CUDA instructions each time a new element is created. Using atomic operations allows for a synchronized state of used memory across the entire grid and prevents Warps from overwriting already used memory.

2.3.4 Edge Insertion

With the concept of sparse bit vectors established and required variables initialized, the PTAGPU algorithm can begin inserting the linear in-memory representation of the constraint graph derived from SVF, see section 2.1, into sparse bit vectors residing in unified memory. The core idea is to handle all edge labels concurrently in individual streams by invoking an edge insertion kernel for each type of edge. Inside each kernel every source node is processed by a single Warp which inserts the outgoing edges into the correct memory location of the correct element inside the sparse bit vector for the given source node. In order to efficiently distribute the work across all available SMs, some preprocessing is needed before each insertion kernel is launched. Given a set of edges of the same type to be preprocessed for insertion, for example all store edges in the constraint graph, the preprocessing steps are illustrated in Figure 2.4. The preprocessing steps are made up of four steps, 1) the linear in-memory representation of the constraint graph is copied into device memory via `cuda::memcpy` 2) all edge tuples (src, dst) are sorted with src as the key, 3) the algorithm iterates through all (src, dst) pairs and eliminates all duplicates,

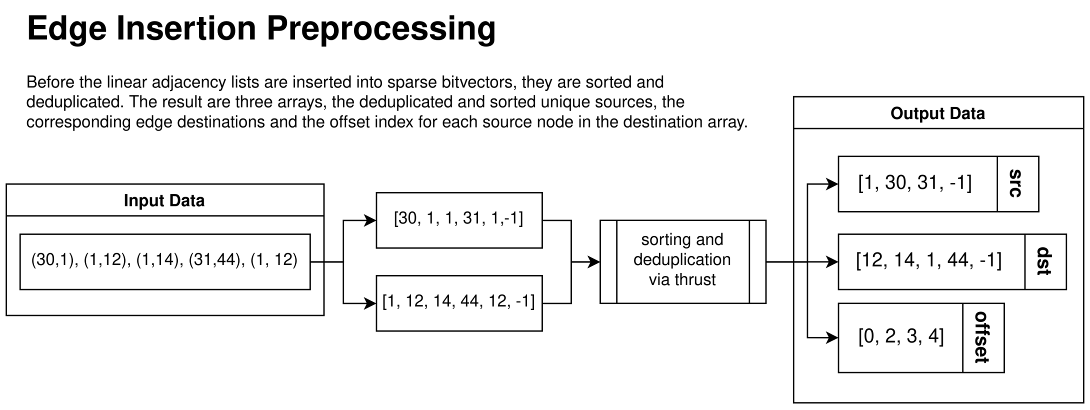
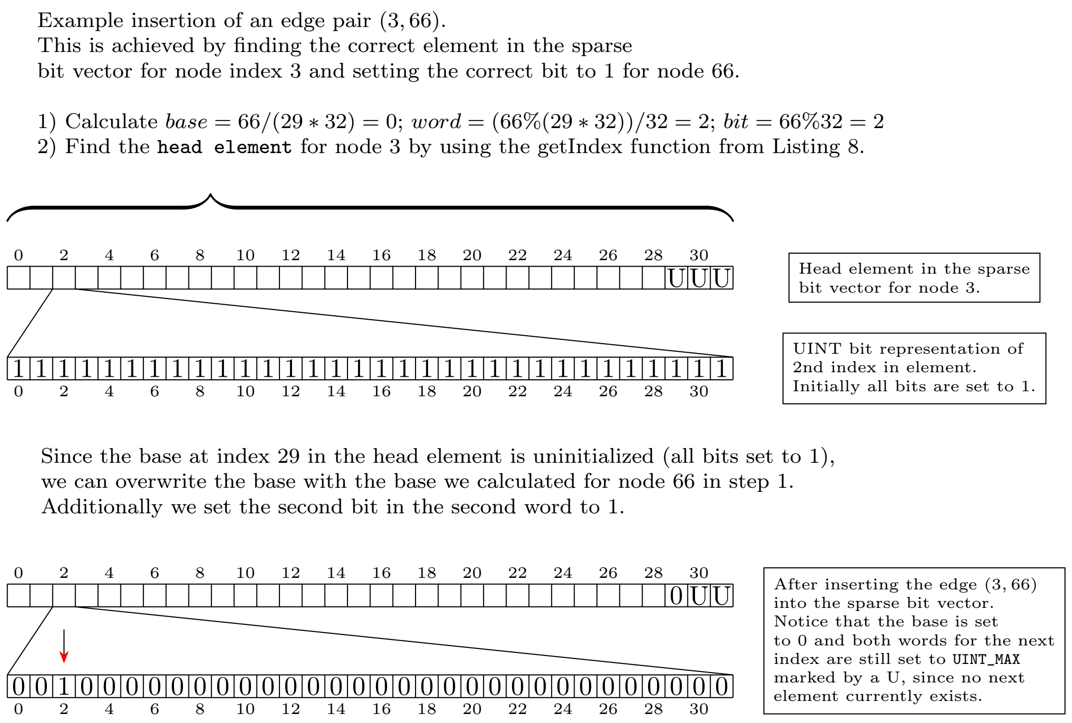

Edge Insertion Preprocessing
Before the linear adjacency lists are inserted into sparse bitvectors, they are sorted and deduplicated. The result are three arrays, the deduplicated and sorted unique sources, the corresponding edge destinations and the offset index for each source node in the destination array.
Input Data
(30,1), (1,12), (1,14), (31,44), (1, 12)
[30, 1, 1, 31, 1,-1]
[1, 12, 14, 44, 12, -1]
sorting and deduplication via thrust
Output Data
[1, 30, 31, -1]
src
[12, 14, 1, 44, -1]
dst
[0, 2, 3, 4]
offset

Figure 2.4: Preprocessing of edges during edge insertion.

4) the algorithm iterates through all (src, dst) pairs and saves the index each time src changes in a third offset array. Since we are operating on linear memory, we can utilize the CUDA thrust library to accelerate these operations on the GPU. Step 2 can be performed by the `thrust::sort` algorithm and step 3 and 4 can be performed at the same time by `thrust::unique_by_key_copy`. One small but important detail is that before we start the preprocessing we insert a pair of `UINT_MAX` values into the array for the source indices and into the array for the destination indices. Having one additional pair assures that step 3 writes the end index for the last actual (src, dst) pair into the offset array.

After the preprocessing is done, all three arrays are fed into an insertion kernel that distributes all source nodes among the available Warps, which then insert the destinations indices into the elements of the sparse bit vector. Insertion into the element first calculates the correct base and then sets the correct bit to one if an element already exists, else a new element is first created.

The example in Figure 2.5 showcases the process for inserting a single edge into an empty sparse bit vector in the insertion kernel. Later insertions always check whether a base is already defined in each element of the sparse bit vector and insert the edges accordingly. The base of subsequent elements in the sparse bit vectors is always sorted in ascending order for efficient random access. To maintain the order, whenever new elements are created, for example if a given base is not yet present in a sparse bit vector, elements might have to be shifted around to maintain the order. In practice these insertions are executed for each source node in parallel by multiple Warps on the GPU.



Example insertion of an edge pair $(3, 66)$.
This is achieved by finding the correct element in the sparse
bit vector for node index 3 and setting the correct bit to 1 for node 66.

1) Calculate $base = 66/(29 * 32) = 0$; $word = (66\%(29 * 32))/32 = 2$; $bit = 66\%32 = 2$
2) Find the `head element` for node 3 by using the getIndex function from Listing 8.

Head element in the sparse bit vector for node 3.

UINT bit representation of 2nd index in element. Initially all bits are set to 1.

Since the base at index 29 in the head element is uninitialized (all bits set to 1),
we can overwrite the base with the base we calculated for node 66 in step 1.
Additionally we set the second bit in the second word to 1.

After inserting the edge $(3, 66)$ into the sparse bit vector. Notice that the base is set to 0 and both words for the next index are still set to `UINT_MAX` marked by a U, since no next element currently exists.

Figure 2.5: An example procedure for inserting a single edge into a sparse bit vector.

2.3.5 Concurrent Graph Rewriting

As soon as all required edges from the constraint graph have been inserted into the correct sparse bit vectors, the Solver part of PTAGPU needs to apply the Andersen constraints from Table 1.5 on the graph in order to calculate the correct points-to sets. As with all calculations on the GPU we need to explicitly take care of concurrency. Similar to the edge insertion kernel, the ideal method of distributing work on the GPU is to assign units of work to individual Warps with minimal thread divergence. When applying the production rules introduced in Table 1.5 we can achieve this by simply using a worklist to spread out all node indices from the constraint graph across all available Warps. This way each Warp handles the outgoing rewrite rules for a single node. Furthermore, this removes any need for specialized scheduling, because in the case that one Warp takes comparatively more time than others, the other Warps simply request new node indices to work on. The inherent synchronization of the worklist already takes care of optimally distributing the work.

Unfortunately we cannot simply use the production rules from the context free grammar approach in Table 1.5. The reason being that these rules can not be mapped onto strictly unidirectional rewrite paths in the graph. As an example, consider the copy production rule $P \to \bar{C}P$. If we interpret this production rule in the context of graph rewriting, we get

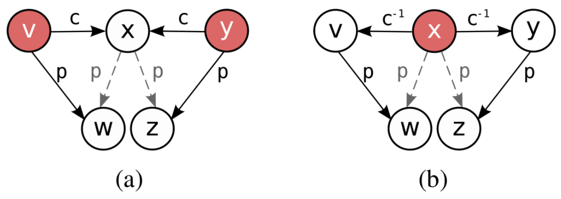


Figure 2.6: Taken from [MBP12], illustrates the problem of cross node mutation when applying unmodified Andersen constraints.

Statement	**Name**	**Rewrite Rule**
$x = y$	copy	$x \xrightarrow{c^{-1}} y \xrightarrow{p} z \rightsquigarrow x \xrightarrow{p} z$
$x = *y$	load	$x \xrightarrow{l^{-1}} y \xrightarrow{p} z \rightsquigarrow x \xrightarrow{c^{-1}} z$
$*x = y$	store	$x \xrightarrow{p^{-1}} y \xrightarrow{s^{-1}} z \rightsquigarrow x \xrightarrow{c^{-1}} z$

Table 2.1: Final unidirectional, partly inverted Graph Rewriting Rules proposed by [MBP12]. Field-sensitive rules are omitted, since they are exclusively handled on the CPU in PTAGPU, see subsection 2.3.6.

that for every node v with an outgoing copy edge to node x and an outgoing points-to edge to w, we need to add a new points-to edge from x to w. The problem arises when multiple Warps operating on outgoing edges of different nodes, simultaneously try to mutate another node's outgoing edges. See Figure 2.6 a) for an illustration of this problem from [MBP12]. A key insight from [MBP12] was that, "Parallel execution of the rewrite rules . . . requires synchronization in the graph data structure.". The intuitive solution to this problem is shown in Figure 2.6 b). For most constraints, simply inverting the edges is enough to allow for concurrent addition of new edges into the sparse bit vectors without mutatung another node's outgoing edges. The resulting graph rewriting rules from [MBP12] are listed in Table 2.1. These new rewrite rules can be interpreted as strictly unidirectional paths in the constraint graph, meaning that when a node x is processed by a Warp, new edges are only added to node x and not to any of the adjacent nodes. One remaining problem is, that the store rewrite rule currently requires inverse points-to edge information which, if additionally stored, would greatly increase the required memory by keeping track of outgoing and incoming points-to edges at the same time for each node to allow efficient access. The proposed solution to this problem from [MBP12] is to apply the store rewrite rules in two steps. First all node pairs (x, y) are collected such that y has outgoing store edges and points to x. Second all pairs with matching x are assigned to the same Warp and the resulting inverse copy edges are added to the sparse bit vectors of x, denoting the destinations of the outgoing inverse store edges from node y. Using this method requires one additional step compared to the copy and load rules, but this step can be processed very efficiently with thrust library calls, similar to the edge insertion routine. With this two-step procedure in place, we can preserve the property that each Warp only appends outgoing edges to the sparse bit vector of it's currently assigned node and refrain from explicit synchronization.

Rule Application Algorithm

With both data structures and graph rewrite rules in place and optimized for operation on the GPU, the main algorithm of PTAGPU can be launched. The pseudocode for the most important procedures of PTAGPU is available, the main loop can be found in Algorithm 2. The first step is to execute the initialization steps from subsection 2.3.2. Since we want to

Algorithm 2 Main Algorithm of PTAGPU

```
initialization steps
while new points-to edges written into memory $\land$ not done do
    run updatepts kernel                                  ▷ see Algorithm 3
    launch CPU code asynchronously                        ▷ see Algorithm 9
    run copy/load/storecollect kernel                     ▷ see Algorithm 4
    sort $(x, y)$ pairs for the store constraints
    split pairs by unique x component
    run store kernel                                      ▷ see Algorithm 8
    await asynchronous CPU execution
    insert new edges found by CPU code
    report memory statistics and new edge count                ▷ optional
end while
free memory and pass results back to SVF
```

apply all rewrite rules of the Andersen style analysis until no further changes are applied, we perform the application of the rules in a while loop that breaks if no changes are detected after one iteration finishes. We can observe the changes by counting the number of recently added points-to edges present in the `newPts` sparse bit vectors. While it might seem inefficient to count through all elements in each iteration of the loop, this can be done very efficiently, since we employ massively parallel GPU kernels. If changes to the points-to memory region have been detected, the algorithm continues with updating the `newPts`, `currentPts` and `oldPts` memory. This serves the purpose of preventing redundant work, by only applying rewrite rules on the points-to edges that have been added since the last iteration. This process is called a delta-update or diffpoints-update, the specifics of this optimization are presented in section 2.3.5 and the pseudocode is available in Algorithm 3. After delta-updates are done, all points-to edges that are new since the last iteration are

Algorithm 3 Update Points Kernel

```
for each node i in constraint graph do
    traverse nextPoints sparse bit vector for node i
    collect all set bits
    compute $\Delta\mathcal{P} = nextPoints \setminus oldPoints$    ▷ find all new bits by excluding the old
    update $currentPoints = \Delta\mathcal{P};\ oldPoints = \Delta\mathcal{P} \cup oldPoints;\ newPoints = \emptyset$
    if $\Delta\mathcal{P} \neq \emptyset$ then
        set done to false
    end if
end for
```

present in the `currentPts` memory region. Before launching the first kernel for the rewrite rules, we can launch the CPU portion of our analysis asynchronously. In practice this is realized by using C++ asynchronous futures. See subsection 2.3.6 for the details of this forked CPU portion of the main loop.

While the asynchronous CPU code executes on a different set of CPU threads, the first GPU kernel can be launched. The parameters for CUDA kernel launches include the block- and gridsize, as well as amount of shared memory and stream assignments. Since GPUs have a set amount of SMs that can process thread blocks in parallel, it is a good idea to split the entire body of work into at least as many thread blocks as SMs are available. During development both an RTX 2080 and a RTX 3080 Ti with 48 and 82 SMs respectively were used. The PTAGPU algorithm queries the available GPUs' SM count and uses it as the gridsize. Because the first GPU kernel processes the bulk of all rewrite rules, it heavily uses registers in each thread during execution. An individual thread requires over 128 active registers to apply the rewrite rules. As such we are limited to 64K registers per thread block per the specifications of the CUDA compute capabilities 7.0 and up. Following this hardware limitation is the fact that we can at most create 256 threads per thread block in order to not overflow the available registers per SM. The result is a blocksize of 256, which is split into eight addressable Warps of size 32 each.

PTAGPU also includes experimental support for multi-GPU CUDA execution. This requires a unique stream per device which are initialized in the initialization steps, see subsection 2.3.2. The stream, together with 256 bytes of shared memory and the grid-/ blocksize specifications, make up the arguments for the first kernel launch. Later kernel launches, including the edge insertion kernel, follow the same process for finding optimal launch parameters. Furthermore, CUDA provides helper functions such as `cudaOccupancyMaxPotentialBlockSize` for finding the optimal grid- and blocksizes of a specific kernel.

The pseudocode for the first GPU kernel is available in Algorithm 4 as well as subsequent device function invocations. As a brief summary, the `Copy Load Storecollect` kernel

Algorithm 4 Copy Load Storecollect Kernel

```
smem ← allocate 256 bytes of shared memory per Warp
upperlimit ← totalNodeCounter
src ← incrementWorklist()
while src < upperlimit do
    applyRewriteRule<Copy>(src,smem)                    ▷ see Algorithm 5
    applyRewriteRule<Load>(src,smem)
    src ← incrementWorklist()
end while
upperlimit ← totalStoreCounter
src ← incrementStoreWorklist()
while src < upperlimit do
    applyRewriteRule<CollectStore>(src,smem)            ▷ collect store/pts pairs
    src ← incrementStoreWorklist()
end while
reset worklists
```

loops through all nodes in the constraint graph and assigns them to available Warps by using a worklist approach. The node that is assigned to a Warp at any time is called the origin node. As soon as a Warp receives an origin node from the constraint graph, the copy, load and the first step of the store rewrite rule are applied in this order. See Table 2.1 for an overview of the rewrite rules that were established by [MBP12]. Each rewrite rule is applied by a template function, see Algorithm 5. Template functions are a C++ construct

Algorithm 5 `template<Type> applyRewriteRule` procedure
Input: src: currently processing node
smem: shared memory

```
index ← getIndex(src)
repeat
    bits ← memory[index + threadIdx.x]
    collectBitvectorTargets<Type>(src,bits,smem)        ▷ see Algorithm 6
    index ← nextBits
until index $\neq$ ULLONG_MAX
if Type = StoreCollect then
    insert all pairs $(src, smem[:])$ into store map    ▷ later used in the store kernel
else
    mergeBitvectors(src,smem)                           ▷ see Algorithm 7
end if
```

that allows using placeholder parameters. Each function invocation with different template parameters results in a different symbol during compilation. While using templates in C++ is good practice for writing reusable, efficient, and error-free code, here we use the fact that CUDA supports templates in device function to prevent the PTX compiler from assuming recursion and falsely limiting the available number of threads to prevent register exhaustion.

Inside the `applyRewriteRule` function, depending on the current rewrite rule, the destinations of all outgoing edges of a certain type are collected by the `collectBitvectorTargets` function, see Algorithm 6. The destinations of the outgoing edges are collected by traversing

Algorithm 6 `template<Type> collectBitvectorTargets` procedure
Input: to: currently processing node
bits: of the head element in to's sparse bit vector
smem: shared memory

```
if bits $\neq$ 0 then
    for i = 0...31 do
        if (1«i) $\wedge$ bits then                                    ▷ check if bit is set
            target $\leftarrow$ base * (29*32) + threadIdx.x * 32 + i
            append target to smem        ▷ target represents the destination of the edge
        end if
    end for
end if
```

the correct sparse bit vector of the origin node, element by element. Following this, all outgoing edges of the collected destination nodes are merged with the outgoing edges of the current origin node of the Warp. For example, applying the copy rewrite rule $origin \xrightarrow{c^{-1}} y \xrightarrow{p} z \rightsquigarrow origin \xrightarrow{p} z$ PTAGPU first collects all targets of the outgoing inverse copy edges, then merges all the outgoing points-to edges from the collected nodes with the outgoing points-to edges of the origin node. This is realized by the `mergeBitvectors` function, see Algorithm 7, at the end of the `applyRewriteRule` function. In principle the bitwise merging of two sparse bit vectors works exactly the same as during edge insertion, shown in Figure 2.5, during initialization. Notably, applying the `storeCollect` rewrite rule, does not merge any sparse bit vectors, but instead collects store and points-to pairs which are later processed in another kernel for the reasons outlined in the beginning of subsection 2.3.5. After all rewrite rules are applied and worklists are reset, the first kernel execution concludes.

Following the first kernel execution in Algorithm 4, the store and points-to pairs previously collected are preprocessed by the thrust library, see Figure 2.4 for an overview of the preprocessing steps, and then fed into the second GPU kernel Algorithm 8 where the last rewrite rule is finally applied.

After the second and final kernel execution, we await the conclusion of the asynchronous CPU procedure. Since the CPU procedure stores all new edges it finds in a pair of vectors, representing the adjacency information of all new edges, we can insert all new edges into the sparse bit vectors by reusing the edge insertion code from the initialization phase.

Algorithm 7 `mergeBitvectors` procedure
Input: to: node that outgoing edges are merged into
smem: array of nodes whose outgoing edges are to be merged with to

```
for from ← smem do
    toIndex ← getIndex(to)                          ▷ find the index of the head element
    fromIndex ← getIndex(from)
    while next element ≠ ULLONG_MAX do
        if fromBase = toBase then
            merge element at fromIndex into toIndex
        else if fromBase < toBase then
            insert new element before toIndex           ▷ elements are sorted by base
        else
            if toIndex has next element then
                toIndex ← toNext
            else
                create new element after toIndex and insert fromIndex
            end if
        end if
        toIndex ← toNext                            ▷ read next elements in sparse bit vectors
        fromIndex ← fromNext
    end while
end for
```

Algorithm 8 Store Kernel

```
smem ← allocate 256 bytes of shared memory per Warp
for some x of collected $(x, y)$ pts/store$^{-1}$ pairs do
    for each unique y of collected $(x, y)$ pts/store$^{-1}$ pairs do
        append y to smem
    end for
    mergeBitvectors(x,smem)                                     ▷ see Algorithm 7
end for
```

At the end of each iteration of the main loop PTAGPU optionally presents the current memory usage and distribution among the memory regions for debugging end evaluation purposes. This includes statistics for the elapsed time of each component in PTAGPU - as measured by the C++ chrono[4] library.

Diffpoints Updates

If the main loop of PTAGPU were to repeatedly iterate and apply the same rewrite rules on every node, every iteration would repeat the work of the previous iteration, making the algorithm very inefficient. This problem does not only apply to PTAGPU, all pointer analyses need to prevent excessive repeating of already finished work. As such many implementations use a concept called diffpoints or delta-updates that is meant to prevent redundant work. Also, pointer analyses inside SVF make use of diffpoints in order to speed up the calculations.

Diffpoints work by dividing the points-to relations calculated during application of the constraints into separate sets. One set describing all new edges that are calculated in the current iteration, one set for all points-to edges calculated in the previous iteration and one set for the bulk of all prior points-to edges. As can be seen in Algorithm 3, the update process between iterations can be realized by simple set operations or more optimized bitwise operations.

$$\Delta\mathcal{P} = nextPoints \setminus oldPoints$$

$$currentPoints = \Delta\mathcal{P} \wedge oldPoints = \Delta\mathcal{P} \cup oldPoints \wedge newPoints = \emptyset$$

After the update at the beginning of each iteration, all procedures apply the rewrite rules on the `currentPoints` set and write new points-to edges into `newPoints`. With diffpoints in place we can ensure that each rewrite rule is only processed once, since any points-to edges are only in the set `currentPoints` exactly once.

One problem that arises from using diffpoints, is the fact that whenever a new copy edge is added into the constraint graph - which can happen dynamically by applying rewrite rules - the next iteration does not process all points-to edges that need to be considered because of the addition of the copy edge. Instead, only those points-to edges are processed, which are in `currentPoints`. The straightforward solution to this problem is to always explicitly work through all relevant points-to edges in `oldPoints` whenever a new copy edge is added. This ensures that no points-to edges are skipped during processing.

[4]`https://en.cppreference.com/w/cpp/chrono`

```
// Launch an asynchronous task
std::future<int> result = std::async([]() {
    // Perform some long-running computation
    int sum = 0;
    for (int i = 0; i < 1000000000; ++i)
    {
        sum += i;
    }
    return sum;
});

// Do something else in the meantime
std::cout << "Hello, world!" << std::endl;

// Wait for the asynchronous task to complete
int final_result = result.get();
std::cout << "The final result is: " << final_result << std::endl;
```

Listing 9: C++ Sample illustrating Asynchronous Futures

2.3.6 Combining CPU and GPU execution

PTAGPU strives to achieve optimal concurrent execution. This includes not only all GPU cores, but also all CPU cores. To achieve this, parts of the main loop, see Algorithm 2, are split into an asynchronous CPU execution. This asynchronous CPU part is implemented with asynchronous futures. Asynchronous futures are a feature of the C++ programming language that allows for the implementation of asynchronous programming. Asynchronous programming allows for the concurrent execution of multiple tasks within a single program, enabling improved efficiency and scalability. By using asynchronous futures, developers can write code that can take advantage of multiple cores or perform time-consuming tasks without blocking the main thread of the program. This can greatly improve the performance of C++ programs, especially on systems with multiple CPU cores or heterogeneous compute capabilities. See the following basic example in Listing 9.

To further improve the scalability of the CPU portion of PTAGPU, the asynchronously executed code is parallelized using OpenMP pragmas. The pseudocode for the CPU code is available in Algorithm 9. As we do not make use of multiple CPU cores in any other part of the program, we can utilize a majority of the available CPU cores for the asynchronous execution. The calculation that is performed on the CPU reads the available points-to information to resolve all `getelementpointer` edges in the constraint graph. While other GPU implementations of Andersen's analysis, such as [MBP12], also perform the `gep` calculations on the GPU, this leaves the CPU mostly idle during the execution.

Algorithm 9 `CPU async` procedure

```
typedef pair<vector<uint>,vector<uint>> edgeSet   ▷ type for adjacency information
procedure ASYNCCPU(edgeSet *ptsSet, edgeSet *copySet)
    #pragma omp parallel for num_threads(16)
    for each GEP edge in constraint graph do
        dstPTS ← $\emptyset$
        srcPTS ← collectBitvectorTargets<PTS>(edge.src)
        for id in srcPTS do
            fieldOffsetNode ← consCG->getGepObjVar(id, edge.offset)
            #pragma omp critical            ▷ only executed by one thread at a time
            append fieldOffsetNode to dstPTS
        end for
        #pragma omp critical
        for id in dstPTS do
            addPts(edge.dst,id)
        end for
    end for
    updateCallGraph(getIndirectCallsites())  ▷ these are SVF superclass methods
    // some of getIndirectCallsites's internal function calls are
    // overwritten to use unified memory instead of SVF data structures
end procedure
```

Furthermore, parallelizing the application of `gep` constraints is not as easily done as for the copy, load and store constraints, since each `gep` instruction carries an offset value. Given the offset value and the points-to target of a node we need to calculate the correct abstract memory object that the offset references. This can be done on the GPU as demonstrated by [MBP12], but is more efficient to implement on a CPU, since `gep` instructions can reference random regions in memory, which would require explicit synchronization on the GPU and slow down the application of the other rewrite rules. Beyond difficult synchronization on a GPU, it is also ideal to use the CPU for `gep` constraint resolution, since PTAGPU is directly integrated into SVF, which already offers a variety of optimized methods for `gep` constraint resolution. Lastly the asynchronous CPU code also resolves all indirect call-sites and updates the call-graph. Arguably this is one of the most important aspects of PTAGPU that differentiates it from other GPU implementations of Andersen's analysis. Indirect call-site resolution is often omitted from academic implementations like [MBP12], [ZWH+21] or [SYX14a]. Because an over-approximated call-graph is one of the fundamental results of a pointer analysis next to points-to information as described in subsection 1.5.1, resolving indirect call-sites is an important step for the analysis to be usable by further analyses inside SVF. Furthermore, connecting the arguments and parameters at indirect call-sites is required to produce complete pointer information.

2.3.7 Feeding the Results back into SVF

At some point PTAGPU concludes its calculation and has produced a constraint graph inside unified memory, which includes an over-approximation of the points-to set for each pointer variable. On its own the constraint graph in the form of sparse bit vectors is not digestible by SVF. For this reason an interface is created to enable SVF to use the points-to information produced by SVF. Internally the `PointerAnalysis` superclass inside SVF provides an `alias` method that takes two node indices and outputs the alias relation of both nodes. To provide pointer information for SVF, PTAGPU overwrites the alias method with its own implementation that reads points-to edges from the sparse bit vectors inside unified memory and outputs one of the possible alias relations from Equation 1.1 and Equation 1.2. One of the consumers of this information is the SVF test suite[5] which will be used in subsection 2.4.1 for verifying the results PTAGPU produces against a know corpus of test programs that include various pointer relations.

Beyond points-to information, the `updateCallGraph` method in Algorithm 9, which resolves indirect call-sites, also requires access to the points-to information of PTAGPU. To facilitate this, some SVF implementations for call-site resolution are overwritten in PTAGPU to redirect the reading of points-to information into the unified memory. The benefit of using C++ polymorphism to overwrite these internal methods from SVF is that the points-to information can remain in unified memory and does not have to be explicitly written back into main memory to enable SVF to access it. This improves performance without impeding the functions of SVF.

2.4 Experimental Results

To validate PTAGPU in terms of correctness and performance, various test cases and benchmark programs were used. Following, the experimental results are split into correctness testing and performance testing. While for correctness testing a pre-defined test suite was used, a collection of current open source programs of varying complexity was compiled for real world performance testing.

[5] `https://github.com/SVF-tools/Test-Suite`

```
int main()
{
    int a,b,*c,*d;
    c = &a;
    d = &a;
    MUSTALIAS(c,d); // c and d may alias each other
    NOALIAS(&b,d); // the address of b and d must not alias
}
```

Listing 10: A unit test in the PTABENCH test suite.

2.4.1 Test Suite

For correctness testing the PTABENCH[6], a subproject of the SVF framework, was used. The PTABENCH repository includes a set of small test programs that are compiled into LLVM bitcode. These test programs contain a set of constraints that represent a state that should be reached by the respective analyses during testing. The test suite implements unit tests for all variants of pointer analysis included in SVF as well as more complex analyses, such as memory leak detection and multithreaded flow analyses. Part of the test corpus are specific edge cases that test against quirks of real programs, such as SPEC CPU2000, in order to ensure all analyses produce sound results given complex input programs. As part of PTAGPU, testing with PTABENCH was enabled, to apply the unit tests and ensure that the results produced are sound and complete. This was achieved by overwriting parts of the test invocation implementation inside SVF as alluded to in subsection 2.3.7.

The constraints in the test programs are implemented as function calls that relay the meaning of each property that is to be checked by the test case. The functions do not have to be defined for the constraint to be checked. When one of these special function calls is encountered at the end of the analysis, an alias check is performed for both function parameters. This check directly reads the points-to relations from unified memory and reports whether both variables may alias or are in fact not aliased. Note that the lack of precision of our whole program context- and flow-insensitive analysis prevents us from making any definitive statements about whether or not two variables have to alias, only that they may alias. For this reason any must alias constraints are interpreted as may alias constraints in our pointer analysis, similar to other pointer analyses in SVF. As an example, see the following test program in Listing 10 that checks whether a pointer analysis in SVF correctly resolves load and store constraints.

When executing the PTABENCH test suite while using PTAGPU as the pointer analysis

[6]https://github.com/SVF-tools/Test-Suite

```
for i in Test-Suite/test_cases_bc/basic_c**/*;
    do build/ptagpu/runptagpu -stat=0 $i ;
    if [ $? -ne 0 ]; then
        # break
    fi
done
```

Listing 11: Sample bash script for applying all unit tests on PTAGPU.

backend, all 107 unit tests applicable to pointer analyses successfully produce the expected results regarding pointer and indirect call-site information. The results can be replicated within the source code either by applying the test suite manually with a simple bash script like the one in Listing 11 or by using the CTest library of CMake and building the `test` target. From this it follows that PTAGPU produces sound, field-sensitive pointer analysis results and does not reduce the accuracy of other pointer analyses in SVF in any way. Interestingly, the current state-of-the-art pointer analysis implementation of SVF, the Andersen based wave propagation algorithm with diffpoints, `wavediff` in short, successfully analyzes only 106 of the 107 unit tests. One test case concerning nested structs failed during testing of the wavediff algorithm. This discrepancy was not further investigated.

2.4.2 Benchmark Suite

To assess the performance of PTAGPU and establish a set of metrics that can be compared to other pointer analysis implementations, a suite of open source projects was compiled and used as input for pointer analyses. The collection of open source projects was selected to present a diverse set of code bases in terms of code style and size of the programs. The collection consists of 16 programs, each varying in complexity. See Table 2.2 for an overview of all the selected benchmark programs. As PTAGPU and SVF rely on the LLVM project for data generation and input in the form of compiled bitcode, the selection of benchmark programs is somewhat limited to those written in languages for which LLVM compiler frontends exist. This includes programming languages such as C, C++, Rust, FORTRAN, and many other systems programming languages.

The evaluation was performed on PTAGPU, the proposed algorithm of this thesis, and both the Andersen based wave propagation algorithm, wavediff, and the naive implementation of the Andersen analysis in SVF. The Andersen wave propagation algorithm represents a highly optimized current state-of-the-art inclusion-based, field-sensitive, context- and flow-insensitive whole program pointer analysis. Furthermore, the results will showcase the differences between sequential CPU implementations and parallel GPU implementations.

program	#V	#E	bc size	density	avg. degree	version
bash	238	77	5.4M	7.514×10^{-5}	0.327	5.1.16
bison	146	59	3.4M	1.417×10^{-5}	0.407	3.8
diff	54	17	1.3M	5.794×10^{-6}	0.317	3.8
git	869	379	25M	3.399×10^{-3}	0.437	2.37.4
htop	48	20	1.6M	8.447×10^{-6}	0.413	3.2.1
httpd	169	95	1.4M	3.180×10^{-5}	0.561	2.4.54
nano	6	2	298K	5.887×10^{-5}	0.399	6.4
perl	445	206	4.9M	1.617×10^{-4}	0.464	5.37.3
php	1582	611	52M	1.689×10^{-4}	0.386	7.4.31
postgres	1432	721	18M	2.642×10^{-4}	0.504	14.4
python	742	313	21M	3.453×10^{-4}	0.422	3.10.6
redis	207	67	4.8M	6.507×10^{-4}	0.327	7.0.5
vim	696	280	7.7M	7.676×10^{-5}	0.403	9.0
linux	4464	2206	72M	5.629×10^{-4}	0.494	5.14
linux-minimal	393	157	5.4M	7.872×10^{-3}	0.401	5.14
zstd	280	101	2.3M	6.260×10^{-5}	0.362	1.5.2

Table 2.2: List of benchmark programs used to evaluate PTAGPU.
Number of nodes and edges in thousands, bitcode file size in kilobytes, density and average out degree for each constraint graph.

Methodology

To generate the bitcode for the pointer analyses, each of the 16 benchmark projects was compiled with the clang[7] compiler. During compilation of each object file that was compiled and assembled, equivalent bitcode was generated which was then linked along with the object code to produce both a binary - or a set of binaries, depending on the project - and a bitcode file for the entire program. Instead of manually intercepting each compile operation for each source file, the wllvm[8] utility was used which is specifically designed to extract whole program bitcode from source packages by injecting bitcode generating compiler flags during the build process. Since most of the selected benchmark programs use common open source toolchains, such as Makefiles and bootstrapping scripts, the custom wllvm compiler interface was specified through either standard environment variables `CC` and `CXX` or arguments to the configure-scripts. Each of the resulting bitcode files are provided together with the source code of PTAGPU. Furthermore, if a given project provided a test suite, that test suite was used to verify the functions of the compiled program to ensure that the compilation successfully produced a working binary-bitcode-pair.

After the bitcode was generated, it was used as input for all three pointer analysis algorithms via a driver program that loads the bitcode into memory, initializes SVF and executes the

[7] `https://clang.llvm.org`
[8] `https://github.com/travitch/whole-program-llvm`

correct pointer analysis. All analyses that were evaluated were compiled with GCC and the optimization flag `-O2`. Each analysis was performed 5 times to ensure the results are consistent and reliable and not falsified by external factors. This was necessary, since two of the hardware setups were part of a shared server. The individual time measurements were performed by using the C++ chrono[9] library.

Hardware Setup

Three separate machines were used to evaluate the pointer analyses. Machine A was equipped with a 12 core AMD Ryzen 5900x CPU, 32 GiB of memory and an NVIDIA RTX 3080Ti, machine B was equipped with two sockets of 16 core Intel Xeon Gold 6242 processors capable of simultaneous multithreading, 1.5 TiB of memory and 8 NVIDIA RTX 2080 GPUs and machine C was a cloud instance equipped with 256 vCPUs and an A100 GPU with 80 GiB of HBM2 memory. Because some of the larger programs in the benchmark suite exceeded the 32 GiB of system memory on machine A during analysis, the reported results were mostly gathered on machine B and C, except for Figure 2.12 where the effects of the different hardware specifications were evaluated. To test how much the larger benchmark programs benefit from more graphics memory, the set of benchmarks were executed on the A100 GPU of machine C as well, which is equipped with more graphics memory than available on the RTX 3080Ti and RTX 2080.

Results

Initially the PTAGPU analysis and both Andersen based pointer analyses, wavediff and naive-ander, were executed on machine B. The results of which can be found in Table 2.3 where the runtimes of all three analyses are reported in elapsed seconds from beginning to end of each analysis. The results from machine B showed that the PTAGPU analysis had a high speedup (greater than 2) for several programs, including "perl", "python", "postgres", and "vim", indicating that it was significantly faster than the wavediff analysis for these programs. However, the PTAGPU analysis had a low speedup (less than 0.5) for several programs, including "diff", "git", "nano" and "php", indicating that it was noticeably slower than the wavediff analysis for these programs. Most of the programs with sub-one speedup factors, are smaller in terms of node and edge count of the constraint graph, such as "diff", "htop", "nano" and "zstd". These slow results for the smaller programs can be explained by the overhead associated with CUDA initialization. For example the PTAGPU analysis for the "nano" program took 1687 ms, where 1533 ms were spent on

[9] `https://en.cppreference.com/w/cpp/chrono`

program	t-wavediff	t-ptagpu	speedup	t-naive-ander	GPU memory
bash	16.222	15.489	1.05	102 195.752	1024 MiB
bison	18.977	9.837	1.93	119 999.054	260 MiB
diff	1.469	4.231	0.35	1561.577	71 MiB
git	557.953	4690.010	0.12	33 404 492.853	21 467 MiB
htop	2.912	5.275	0.55	5696.107	93 MiB
httpd	5.321	6.414	0.83	2889.208	160 MiB
nano	0.087	1.751	0.05	98.5	7 MiB
perl	103.338	45.093	2.29	2 688 610.846	3999 MiB
php	645.697	64 965.400	0.01	6 530 636.248	27 561 MiB
postgres	997.355	465.527	2.14	6 481 597.401	16 430 MiB
python	536.515	203.649	2.63	1 731 373.999	9016 MiB
redis-server	8.679	11.592	0.75	4834.759	207 MiB
vim	1052.995	268.628	3.92	-^{b}	11 966 MiB
vmlinux	32 100.566	-^{a}	-^{a}	-^{a}	-^{a}
vmlinux-tiny	91.479	188.315	0.49	1 410 004.628	2175 MiB
zstd	11.063	13.172	0.84	7958.999	454 MiB

Table 2.3: Benchmark results comparing PTAGPU, Andersen wavediff and naive Andersen analysis measured in seconds. Executed on machine B using an RTX 2080 GPU.
The ^{a} denotes that the analysis did not finish in under 24 hours.
The ^{b} denotes that the analysis failed to compute a solution.

CUDA memory allocations and deallocations and 145 ms on the actual compute kernels. Comparing this to the 87 ms required for the wavediff analysis, the PTAGPU analysis runtime is dominated by overhead from the CUDA memory management. This explains the relatively slow performance of PTAGPU on the smaller benchmarks.

Notably the results for the benchmarks "git" and "php" achieved an especially low speedup of 0.12 and 0.01 respectively on machine B, although both programs are relatively large and the analysis was not dominated by CUDA memory management, but the primary kernel instead, as can be seen in Figure 2.7, indicating a large workload on the GPU. Looking at the third graph in Figure 2.8, it is clear that exceeding the 12 GiB of graphics memory available per GPU on machine B had a disproportionately negative effect on performance. This can be attributed to the fact that allocating more unified memory than available graphics memory leads to an excess of page faults, as soon as the graphics memory is exhausted during migration from the CPU to the GPU. While this could be mitigated by prefetching only the required memory regions into the GPU memory on-demand during analysis, this would require partitioning the sparse bit vectors into chunks that entirely fit into graphics memory, which is not trivially done for larger analyses, especially not dynamically. To verify that the performance impact was caused by page faults in the CUDA driver, the `strace` tool was used to inspect the system calls during the PTAGPU analysis of

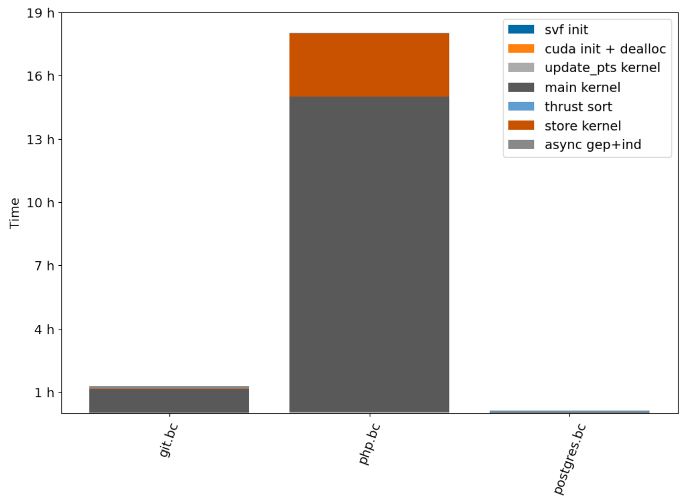


Figure 2.7: Stacked bar graph for PTAGPU runtime on git, postgres and php benchmarks in a memory constrained environment on machine B using an RTX 2080 GPU. Runtime is dominated by the main and the store kernel.

the php program, which showed `ioctl` system calls to and from the CUDA unified memory address space, verifying the suspected problem.

To further evaluate the performance of the PTAGPU algorithm without being memory constrained, machine C was used to analyze the benchmark programs with PTAGPU again. Using the NVIDIA A100 GPU on machine C, all analyses except "vmlinux" fit into the 80 GiB of graphics memory. The results of the second benchmarks can be found in Table 2.4.

The benchmarks on the A100 GPU overall improved the performance of PTAGPU compared to machine B, increasing the speedup factor for all programs. See Figure 2.9 for an overview of the speedup factors PTAGPU achieved over the wavediff algorithm. Specifically the "git", "php", "postgres" and "python" benchmarks improved the most, since the analysis was no longer limited by GPU memory. Depending on the benchmark program, PTAGPU was up to four times faster than the wavediff algorithm. Compared to the naive implementation of Andersen's algorithm in SVF, PTAGPU was several orders of magnitude faster on every one of the benchmarks. At its slowest PTAGPU was about 50% as fast as the wavediff algorithm, ignoring minimal analyses, such as "nano".

Looking at the distribution of time spent on the individual components of PTAGPU in Figure 2.10, it is evident that the bulk of the time is spent in the main kernel and the asynchronous resolution of `gep` constraints and indirect call-sites. With an increase in size of the constraint graph, the relative time spent in the asynchronous CPU portion increases, since the number of `gep` constraints tends to increase with the number of pointer variables in larger programs.

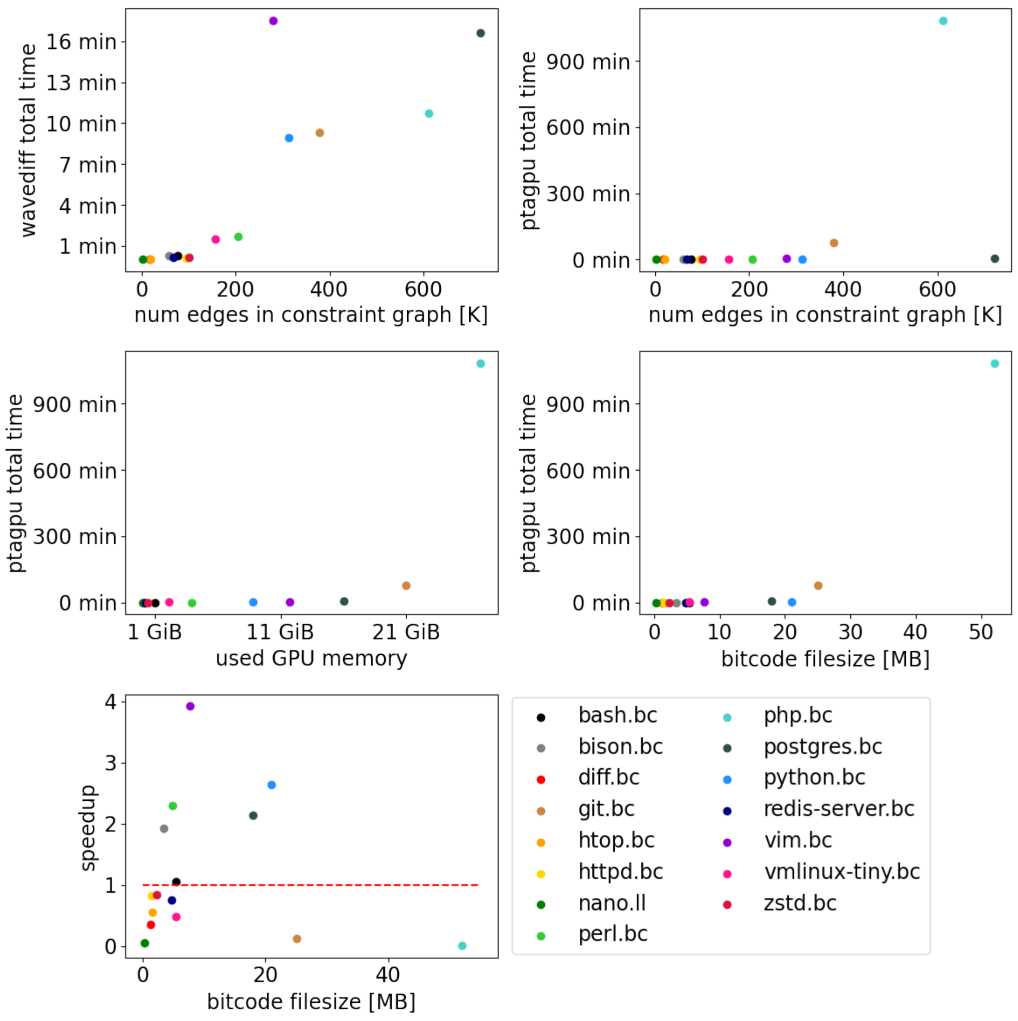


Figure 2.8: Detailed statistics from pointer analyses on machine B using an RTX 2080 GPU.

program	t-wavediff	t-ptagpu-a100	speedup	GPU memory
bash	16.222	13.126	1.23	1024.0 MiB
bison	18.977	7.599	2.49	260 MiB
diff	1.469	3.305	0.44	71 MiB
git	557.953	869.302	0.64	21 467 MiB
htop	2.912	4.039	0.72	93 MiB
httpd	5.321	4.720	1.12	160 MiB
nano	0.087	1.626	0.05	7 MiB
perl	103.338	41.854	2.46	3999 MiB
php	645.697	500.086	1.29	27 561 MiB
postgres	997.355	290.219	3.43	16 430 MiB
python	536.515	172.701	3.10	9016 MiB
redis-server	8.679	8.830	0.98	207 MiB
vim	1052.995	247.859	4.24	11 966 MiB
vmlinux	32 100.566	-[a]	-[a]	-[a]
vmlinux-tiny	91.479	136.164	0.67	2175 MiB
zstd	11.063	9.795	1.12	454 MiB

Table 2.4: Benchmark results comparing PTAGPU and Andersen wavediff measured in seconds. Executed on machine C using an A100 GPU.
The [a] denotes that the analysis ran out of memory.

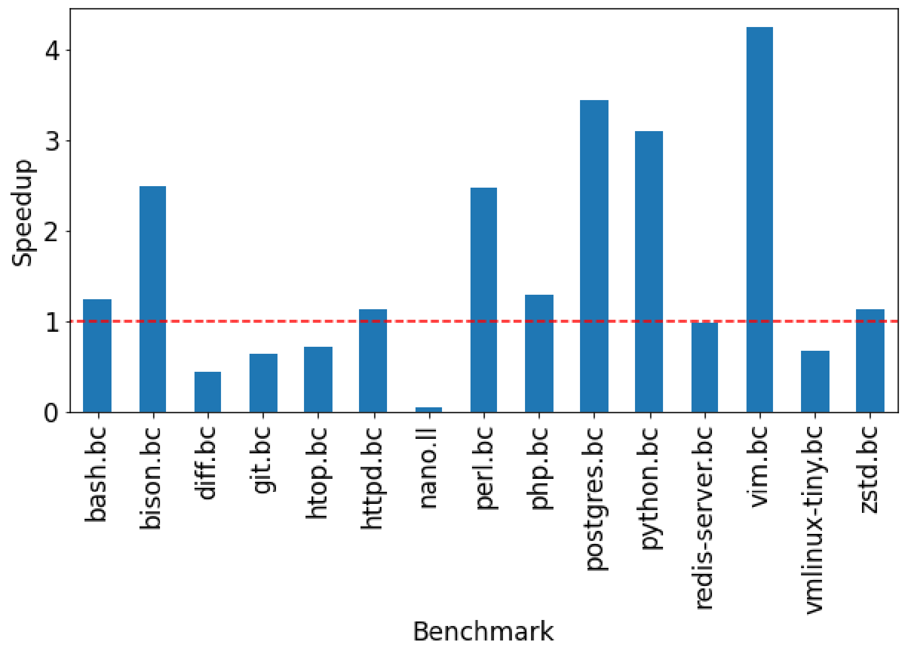


Figure 2.9: PTAGPU speedup for each benchmark program over wavediff algorithm on machine C with an A100 GPU.

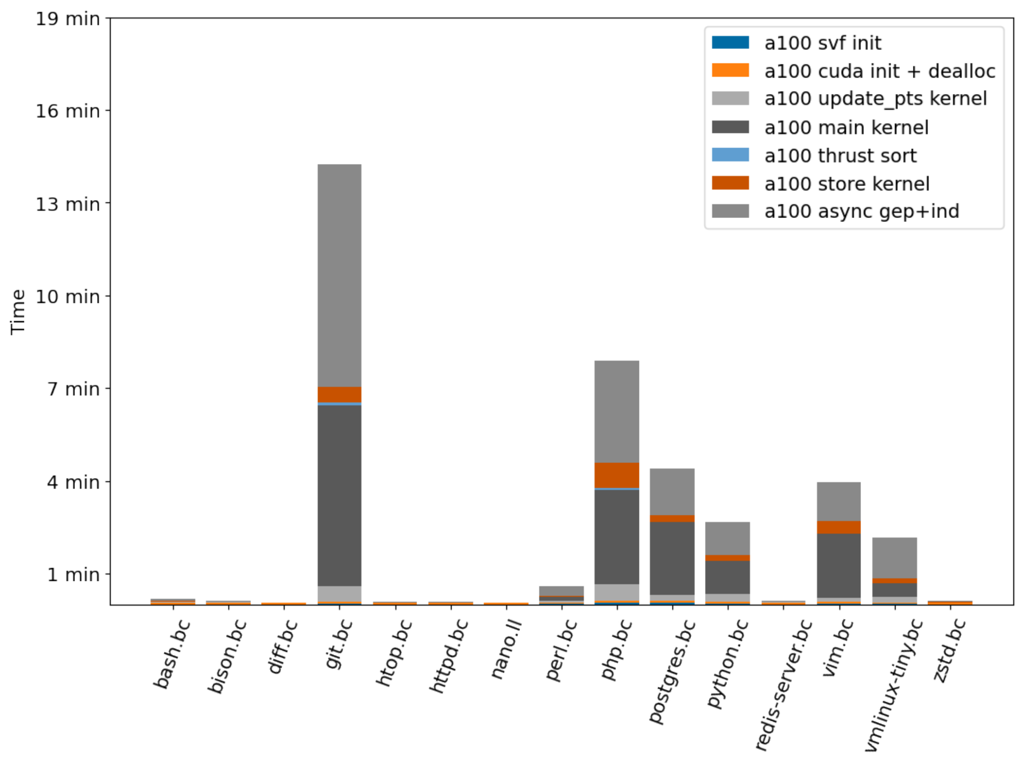


Figure 2.10: Time spent on individual PTAGPU components on machine C.

Looking at the statistics for the benchmarks on machine C in Figure 2.11, neither program size nor size of the constraint graph can be used to accurately estimate the performance of either pointer analysis. For example the "postgres" benchmark program consists of both more nodes and more edges in the constraint graph, but performs considerably better during analysis than benchmarks with smaller constraint graphs, such as "git". For PTAGPU, unified memory usage during analysis is a good indicator for runtime performance. Interestingly the performance on the wavediff algorithm was worse in "postgres" than in "git" indicating that the CPU algorithms are more influenced by the overall size of the constraint graph. One hypothesis for these results is that the internal code structure of the benchmark programs greatly influences the parallel performance on the GPU. While relatively flat programs can benefit from parallel application of the Andersen constraints, programs that contain long nested chains of instructions do not benefit from concurrent rule application as much, since each instruction depends on the previous application of the constraint rules. See Listing 12 for an intuitive example for flat and nested program structures. Note that this example is not realistic and would likely be optimized by a compiler. In real programs, nested structs or classes in C++ tend to result in a nested program structure. This theory can explain the discrepancy between the benchmark results of "postgres" and "git". The C-specific `struct` keyword was used 13124 times in the source

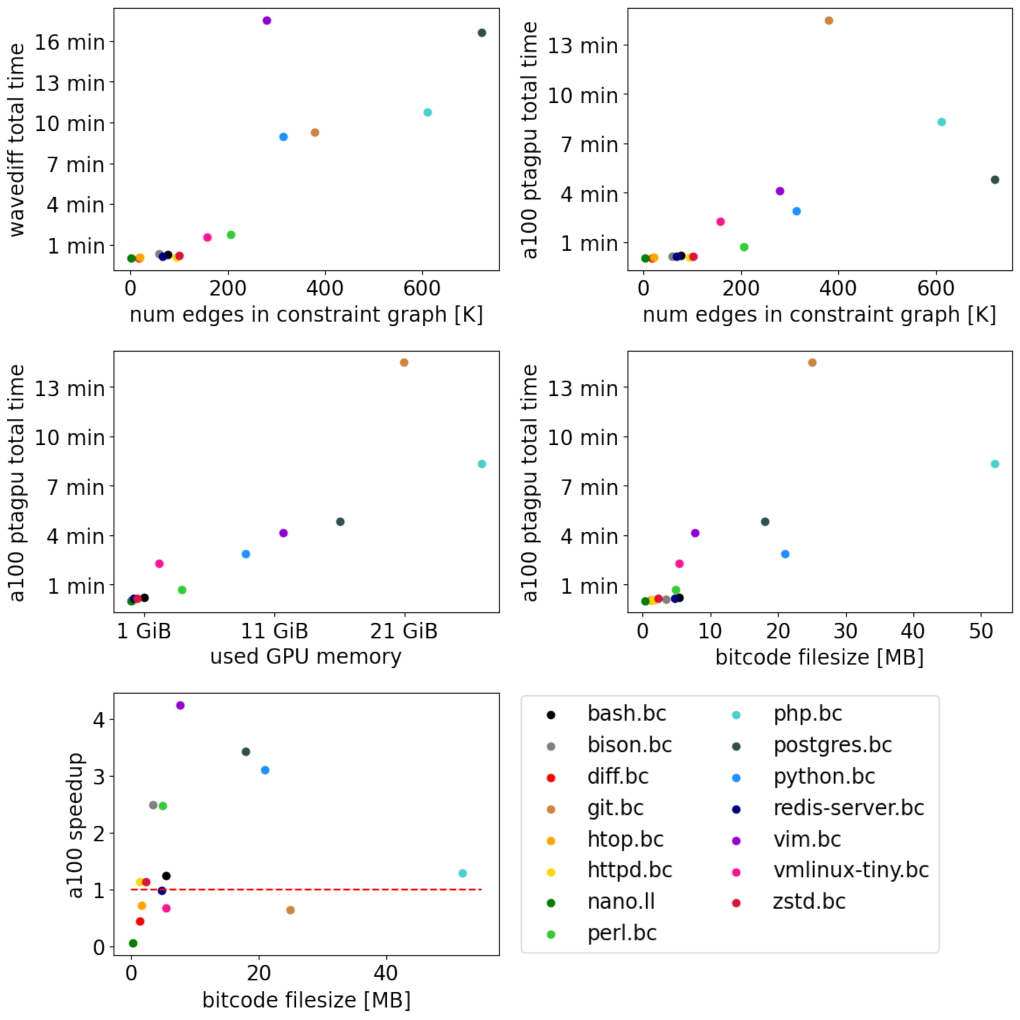


Figure 2.11: Detailed statistics from pointer analyses on machine C.
Memory constraints removed by using an A100 GPU.

```
// flat program structure
int value1 = 14;
int value2 = value1 * 2;
method(value1, value2);
// nested program structure
struct param { int f1; int f2;} p;
p.f1 = 14; p.f2 =  p.f1 * 2;
method(p);
```

Listing 12: Intuitive illustration of a "flat" and a "nested" program structure.

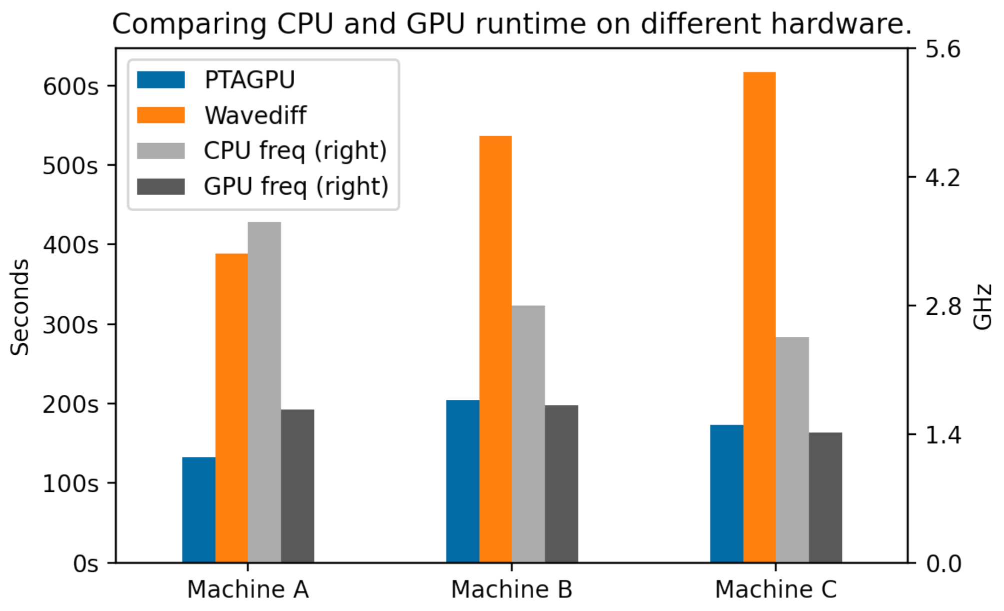


Figure 2.12: Comparison of the CPU and GPU runtime on the different hardware configurations.
Here the benchmark program `python` was used to compare the different machines.

code of the "postgres" program, while being used 27459 times in the git source code. Hence, the apparent nested structure in the git source code could lead to a decreased speedup compared to the much larger but flatter code structure in the "postgres" source code.

Comparing the three different hardware configurations in Figure 2.12, the influence of CPU frequency can be seen on the wavediff pointer analysis. Similarly, PTAGPU also seems to benefit from higher boost frequencies of the GPU, since Machine A with an RTX 3080 Ti clocked at 1665 MHz outperformed the A100 clocked at 1410 MHz. Total graphics memory does not affect the analysis performance of PTAGPU, as long as enough memory is available.

Lastly, an experimental multi-GPU implementation of PTAGPU was briefly tested against some of the benchmarks. Since PTAGPU synchronizes across each of the thread blocks via atomic incrementation of a worklist variable, the scope of the atomic incrementation can be extended from a single device to system-wide atomic access as long as the devices support the CUDA compute capability 6.0 or above. In theory this allows for simple multi-GPU execution of the PTAGPU analysis. In practice, when executing PTAGPU with multiple RTX 2080 GPUs on machine B, the performance decreased substantially. This can be explained by the fact that atomic access to a variable across multiple devices on a system requires more expensive synchronization than on a single device. Furthermore, cross-device memory access is relatively slow. Here using faster GPU interconnects, like NVLink bridges between the GPUs could improve the performance. Improving the performance of PTAGPU regarding the multi-GPU synchronization overhead would require partitioning the constraint graph into partitions that could individually be processed on each of the GPUs, which is beyond the scope of the current PTAGPU implementation.

Chapter 3

Discussion

3.1 Evaluation of Results

To summarize the research conducted up to this point, a GPU-supported parallel Andersen based pointer analysis, PTAGPU, was implemented on top of LLVM and the SVF framework using the NVIDIA CUDA SDK. PTAGPU was both tested for correctness and performance in comparison to established CPU-based pointer analyses in multiple benchmarks and hardware combinations. The primary research question was to what extent such an implementation presents advantages or disadvantages over other analyses that are not strictly parallel in nature.

The experimental results presented in section 2.4.2 indicate that PTAGPU is a viable implementation to improve the performance of a whole program field-sensitive pointer analysis without sacrificing precision of the analysis. PTAGPU achieved an average speedup of 1.60 across 15 benchmarks over the SVF wavediff algorithm. Since the individual benchmarks with lower speedups are almost entirely smaller programs, the absolute time saved by using PTAGPU was 28 minutes and 59 seconds over the total 1 hour 07 minutes and 30 seconds required by wavediff to analyze the 15 benchmark programs. In terms of absolute time, the speedup factor was 1.75. This represents a meaningful improvement over the CPU algorithm. Unfortunately this does not include the Linux benchmark, as the analysis did not fit into graphics memory and did not finish within a reasonable amount of time, which was decided to be 24 hours. Although enough total unified memory was available, the analysis of the Linux source code did become impeded by excessive page faults as illustrated in section 2.4.2. Judging from the wavediff analysis of the Linux kernel, roughly 300 GiB of memory are required for a successful whole program field-sensitive pointer analysis, which is currently impossible to compute entirely in the graphics memory of a single graphics card.

One key insight from the experimental results was, that there are no precise trivial prior indicators for analysis runtime, since the program structure is seemingly more influential on analysis runtime than lines of code, nodes in the constraint graph or size of the compiled program. This is supported by prior research from [MBP12] or [SYXL15], where the selected benchmark programs - some of which overlap with the selected benchmark programs of this thesis - also had vastly different speedups for the GPU-based analysis over the CPU-based analyses. It was found that the use of nested programming techniques, such as structures in a C program, can lead to a lower speedup on parallel hardware. As a result, programs like git and the Linux kernel, that make extensive use of C structs, are performing worse in PTAGPU compared to other benchmark programs. Because PTAGPU is field-sensitive, the effect of structures with fields in the source code disproportionately influences the runtime of the analysis compared to other instructions that can more easily be parallelized. Still, the majority of benchmark programs did profit from the parallel analysis in PTAGPU and performed better on the GPU than on the CPU.

Although the results indicate that PTAGPU generally outperforms the CPU counterparts in larger benchmarks, it should be acknowledged that it is difficult to establish a fair comparison between CPU and GPU pointer analysis implementations. Both in terms of upfront cost and power consumption both implementations are vastly different. While the SVF wavediff algorithm can be executed on almost any modern computer with sufficient system memory, PTAGPU requires specialized hardware. Especially for larger analyses, a GPU with enough graphics memory is required for an efficient analysis with PTAGPU.

Comparing PTAGPU Results to other GPU-based implementations.

Some other GPU-based Andersen style pointer analyses are [MBP12] and [SYXL15]. Unfortunately the results from both these GPU-based pointer analysis algorithms are not directly comparable with PTAGPU in terms of total runtime, since the reported constraint graph sizes are vastly smaller than those produced by SVF, indicating that they might be incomplete in some way. Intuitively the reported result for performing a whole program field-sensitive inclusion-based pointer analysis on the Linux kernel in under ten seconds [SYXL15] does seem too fast, given the results from SVF. Although the input data generation for these analyses was based on LLVM, according to a previous paper by Mendez et al. [MMP10], the source code for the data generation is not available. Furthermore, indirect call-sites were not resolved during either of these analyses making them fundamentally incompatible with the results of PTAGPU. Because the source code of neither of these implementations is available online and no other GPU-based implementations of Andersen style pointer analyses are known at this time, direct comparisons with other GPU implementations are omitted from the evaluation.

3.2 Future Work

During the development of PTAGPU and evaluation of the results, multiple novel approaches and ideas were discovered but not implemented, which will be summarized in the following section. Some of these approaches could be used to further improve the performance of PTAGPU and alleviate shortcomings and limitations discovered during development. Furthermore, some ideas could lead to improvements in the SVF project upstream, and inspire new research directions.

Improvements for PTAGPU

The hypothesized approach of using unified memory in a CUDA program to extend the scalability of PTAGPU and get around the limitation of only using the device's graphics memory did not work during experimental testing. It was still necessary that the GPU had enough graphics memory to hold the entire analysis, because the performance of the algorithm would drastically decrease if graphics memory were to be exhausted while the analysis was still running. The reason for this was that accessing sparse bit vector elements outside of graphics memory would result in page faults and require moving the requested memory region into graphics memory, replacing a previous section of memory and slowing down the analysis. The CUDA API does provide functions to prevent this behavior. By using `cudaMemAdvise` and `cudaMemPrefetchAsync` the developer can give hints to the memory controller as to where specific memory regions are preferred to be stored and loaded into before the data is actually required by the algorithm. Fundamentally the problem with prefetching memory is the irregularity of a pointer analysis, making it very difficult to predict which memory regions are required in each iteration of the algorithm.

One possible solution to this problem would be a partitioning strategy, where the entire constraint graph is divided into roughly equally sized partitions. The algorithm would then load each partition into device memory and resolve all constraints $x \rightarrow y \rightarrow z \rightsquigarrow x \rightarrow z$ where all nodes, x, y and z are included in the currently loaded partition. To solve constraints that require loading nodes outside the current partition, two partitions are always loaded into memory in pairs. Constraints that involve nodes of both partitions are then solved. This is repeated until all constraints are resolved. This exact approach was used by Graspan in [ZWH+21] using the file system instead of unified memory. The idea remains the same.

Developing a partitioning system instead of loading the entire constraint graph into graphics memory, would also enable the use of multiple GPUs without having to synchronize access to individual nodes across all devices. The downside with partitioning the constraint graph lies in the fact that with an increasing number of partitions, the overhead associated with loading and unloading partitions increases. Splitting the constraint graph into 16 partitions would require loading and unloading all 15 adjacent partitions in order to resolve all constraints of a single partition in the worst case. Part of this problem can be reduced by performing an offline analysis and associating nodes with partitions such that a minimal number of edges overlap between partitions, leading to an optimal distribution of nodes for parallel graph rewriting.

Other than partitioning the constraint graph, there are various optimizations proposed in [MBP12] that did not yield any immediate performance improvements during the development of PTAGPU but might bring improvements given more thorough testing. These optimizations include online cycle detection and pointer-equivalent variable detection on the GPU during analysis. Variables are pointer-equivalent if they possess the same outgoing points-to edges. These might be interesting to find, since applying a copy rewrite rule associated with any of the pointer-equivalent variables, results in the same operation on the GPU and would be redundant. Similarly, a cycle of copy edges in the constraint graph would always hold the same points-to edges for each node in the cycle and thus should be detected during analysis to prevent redundant work.

Another approach for improving the scalability of PTAGPU would be the use of peer-to-peer memory access. Peer-to-peer memory access is a capability of certain NVIDIA GPUs whereby multiple graphics cards can pool their memory though special NVLink interconnects on the GPUs, bypassing the bandwidth restrictions of the PCIe connections. This would allow one kernel on a single device to access the graphics memory of several graphics cards and increase the available amount of memory during analysis and improving the scalability.

Finally, PTAGPU would greatly benefit from dynamic memory allocation. Currently, the total amount of unified memory has to be set during compilation, which is then allocated during runtime. Since pointer analyses are very irregular calculations, it is very difficult to set the correct amount of memory before the analysis is run. For this reason a dynamic memory allocation algorithm could improve the usability of PTAGPU by predicting and reallocating the amount of memory needed for the analysis.

Improvements for SVF

As PTAGPU and prior work has shown, there are lots of possibilities for parallelizing a pointer analysis. Since GPUs are not the only devices capable of parallel execution, it would be another interesting avenue of research to apply some of the techniques discussed in this thesis on the CPU-based pointer analyses of SVF. Seeing how other papers, like [MBP12], compared GPU and CPU parallel pointer analysis implementations, it would be interesting to compare a parallelized SVF-based pointer analysis on the CPU to PTAGPU. Especially for smaller programs, this might bring performance improvements. Part of a parallel CPU implementation in SVF would be to create thread safe versions of the internal SVF data structures. This would also indirectly improve the performance of PTAGPU, since the asynchronous CPU section of PTAGPU also utilizes parts of SVF that currently can not be parallelized. Especially improving the parallel resolution of `getelementpointer` constraints could yield great improvements for PTAGPU, since dealing with nested code structures is currently one of its weaknesses. It would also be interesting to further analyze the effects of using specific language constructs on the runtime performance on parallel pointer analyses.

Ultimately it would be a good idea to implement a hybrid pointer analysis in SVF that would be capable of utilizing both CPUs and GPUs for the analysis depending on the program to be analyzed. Using the CPU implementation for smaller programs and the GPU implementation for larger programs until graphics memory is exhausted, could maximize the performance for different kinds of input programs. Since pointer analyses do not remove any already computed points-to data, a given analysis could switch between CPU and GPU implementations relatively uninterruptedly, without having to repeat a large amount of work. Using diffpoints further improves this process.

Finally, another avenue for future research is the reevaluation of context-free-grammars, possibly in a parallel execution environment, for the purpose of pointer analysis. Recent additions to the SVF framework include a solver component that explicitly utilizes context-free-grammars [LSDZ22] and is allegedly more performant than Graspan at querying CFL-reachability in a graph, according to the cited paper.

3.3 Conclusion

Concluding, PTAGPU was capable of meaningfully improving the performance of a whole program field-sensitive pointer analysis on top of LLVM and the SVF framework. The main advantages lie with relatively large programs that are built using a relatively flat code structure, where PTAGPU achieves a respectable speedup over CPU-based pointer analyses. Given how general Andersen style pointer analyses are a P-complete computational problem [MP21], this result is not achieved by trivial methods, but requires a wide set of complex analysis techniques and optimizations. All while being able to use a reproducible and well-defined input format in the form of LLVM bitcode.

The main disadvantages are that using GPUs for a pointer analysis always introduces an overhead for graphics memory management and conversion of input data into a format suitable for the GPU. This makes using GPUs for analyzing small programs (less than 1 MB in size) inefficient. For optimal performance PTAGPU also requires that the entire analysis fits into graphics memory, which currently prevents some of the largest software projects, such as the Linux kernel, from being analyzed by PTAGPU efficiently.

Overall the parallel GPU-based pointer analysis PTAGPU has shown to be a promising approach for improving the efficiency of pointer analysis. By leveraging the parallel processing capabilities of GPUs, PTAGPU was able to achieve an average speedup of 1.60 over the state-of-the-art wavediff pointer analysis in the SVF framework. This demonstrates the potential of PTAGPU to significantly reduce the time required for pointer analysis, making it a valuable tool for developers and researchers working in this field. This implementation should provide a good reference for future research into parallel pointer analyses and can easily be expanded upon based on the modular design of the SVF framework.

Appendix A

Raw Data

id	file	GPU memory MiB	filesize MB	wavediff-t	naiveander-t	# nodes	# edges	version
1	bash	1024	5.400	16222.113	102195.752	238	77	5.1.16
2	bison	260	3.400	18977.376	119999.054	146	59	3.8
3	diff	71	1.300	1469.056	1561.577	54	17	3.8
4	git	21467	25.000	557953.924	33404492.853	869	379	2.37.4
5	htop	93	1.600	2912.693	5696.107	48	20	3.2.1
6	httpd	160	1.400	5321.631	2889.208	169	95	2.4.54
7	nano	7	0.298	87.438	98.500	6	2	6.4
8	perl	3999	4.900	103338.824	2688610.846	445	206	5.37.3
9	php	27561	52.000	645697.175	6530636.248	1582	611	7.4.31
10	postgres	16430	18.000	997355.718	6481597.401	1432	721	14.4
11	python	9016	21.000	536515.479	1731373.999	742	313	3.10.6
12	redis-server	207	4.800	8679.144	4834.759	207	67	7.0.5
13	vim	11966	7.700	1052995.525	NaN	696	280	9.0
14	vmlinux	NaN	72.000	32100566.652	NaN	4464	2206	5.14
15	vmlinux-tiny	2175	5.400	91479.697	1410004.628	393	157	5.14
16	zstd	454	2.300	11063.214	7958.999	280	101	1.5.2

Table A.1: Raw Data of Baseline results for wavediff and naive-ander Pointer Analyses Node and Edge count in thousands, times in milliseconds.

id	ptagpu-t	svf init	cuda init	update-k	main-k	thrust sort	store-k	async CPU	S
1	15489.00	366.515	2489.389	1129.063	1324.698	743.670	858.932	5190.693	1.05
2	9837.06	214.357	1650.840	830.177	234.046	675.748	54.633	4317.114	1.93
3	4231.58	66.945	2065.479	103.057	33.120	379.843	23.591	953.283	0.35
4	4690010.00	2190.250	17095.315	70148.000	3968158.215	11743.372	68373.295	538454.710	0.12
5	5275.51	68.425	2053.573	229.714	202.955	433.321	41.092	1587.968	0.55
6	6414.69	309.354	1481.509	303.996	42.605	511.057	8.219	1637.408	0.83
7	1751.01	7.732	1533.113	1.829	3.040	45.140	1.057	94.275	0.05
8	45093.40	775.915	2939.635	3457.350	6125.819	767.334	2751.206	21457.059	2.29
9	64965400.00	2809.670	20928.407	193679.543	53783100.092	12298.489	10732741.786	187398.213	0.01
10	465527.00	2911.590	8824.470	65379.251	192989.575	1374.555	32019.477	135372.093	2.14
11	203649.00	1344.910	4088.639	14380.164	63762.440	16295.704	12199.745	79598.063	2.63
12	11592.50	357.262	2785.032	688.684	95.318	1158.755	42.613	3688.319	0.75
13	268628.00	1166.830	9292.169	10709.775	113872.762	947.744	20876.445	100689.414	3.92
14	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
15	188315.00	654.966	2405.784	10600.173	25890.805	12319.814	7893.207	121492.773	0.49
16	13172.60	388.764	2791.627	680.571	90.320	687.799	80.554	4526.521	0.84

Table A.2: Raw Data of PTAGPU on Machine B, times in milliseconds.

id	ptagpu-t	svf init	cuda init	update-k	main-k	thrust sort	store-k	async CPU	S
1	13126.316	316.589	3339.273	955.132	1258.228	103.923	824.671	3404.405	1.23
2	7599.453	181.425	3397.507	657.559	203.553	92.256	41.323	2941.927	2.49
3	3304.592	53.196	3119.867	100.198	29.448	73.122	13.813	379.511	0.44
4	869302.000	1489.250	2852.620	31480.079	350891.929	4856.937	30923.576	431554.218	0.64
5	4038.684	58.770	3238.629	212.349	189.568	74.528	20.536	852.136	0.72
6	4720.701	233.173	3125.778	258.309	39.757	57.139	5.049	848.704	1.12
7	1626.456	5.655	3163.307	2.024	4.073	8.778	0.786	18.179	0.05
8	41853.936	644.417	3277.899	3452.222	6873.348	164.523	2906.182	18027.103	2.46
9	500086.000	2561.200	3815.678	32016.810	184075.139	2797.468	49062.434	198397.500	1.29
10	290219.000	2465.070	4439.149	12040.787	140270.103	211.504	14064.883	89735.395	3.43
11	172701.000	1088.500	3468.036	14790.821	65522.991	203.848	11336.972	63986.167	3.10
12	8830.066	247.339	3339.556	615.407	103.871	109.152	24.580	2143.869	0.98
13	247859.000	963.959	3336.741	7599.231	125708.265	123.820	23102.001	75995.761	4.24
14	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
15	136164.602	554.447	3152.182	11286.157	26563.670	236.409	8721.936	79417.327	0.67
16	9795.036	312.077	3104.891	546.614	70.509	75.957	61.621	2587.828	1.12

Table A.3: Raw Data of PTAGPU on Machine C, times in milliseconds.

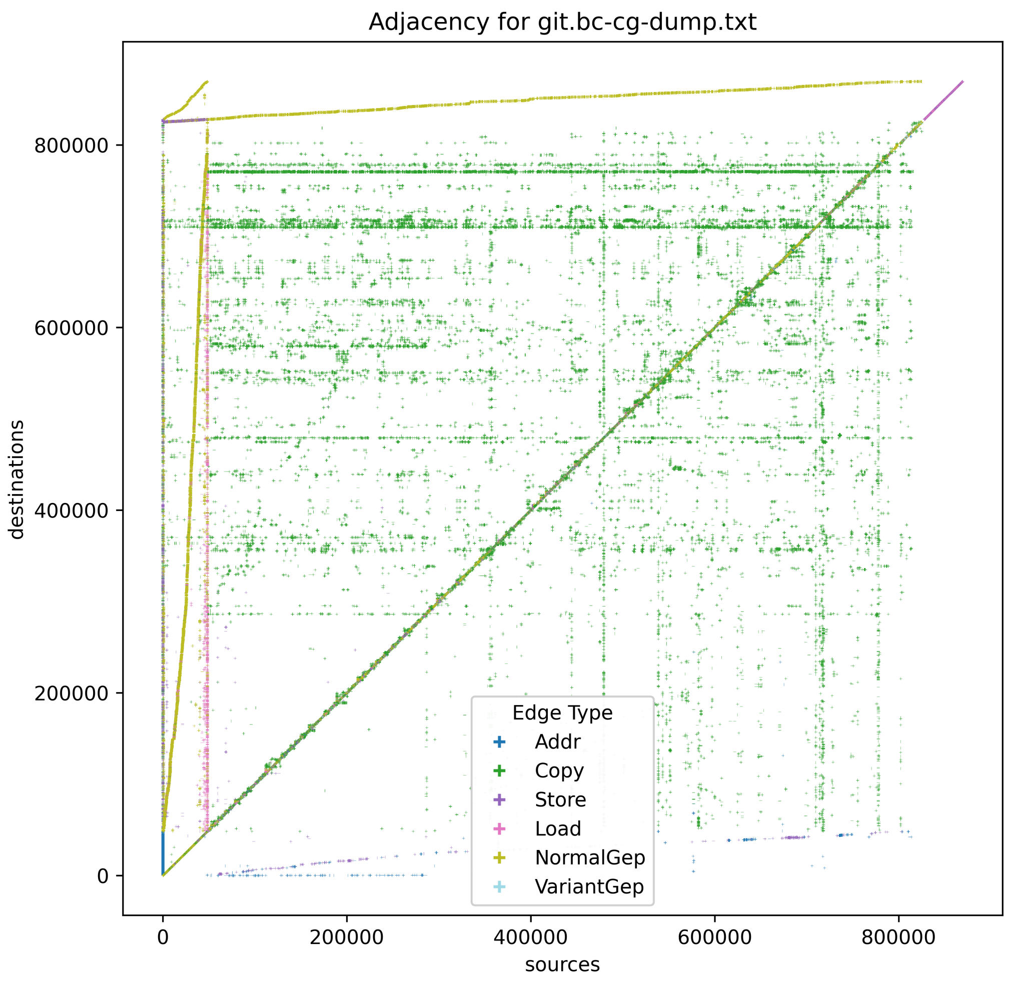


Figure A.1: Adjacency Plot for the Constraint Graph of the Git Client

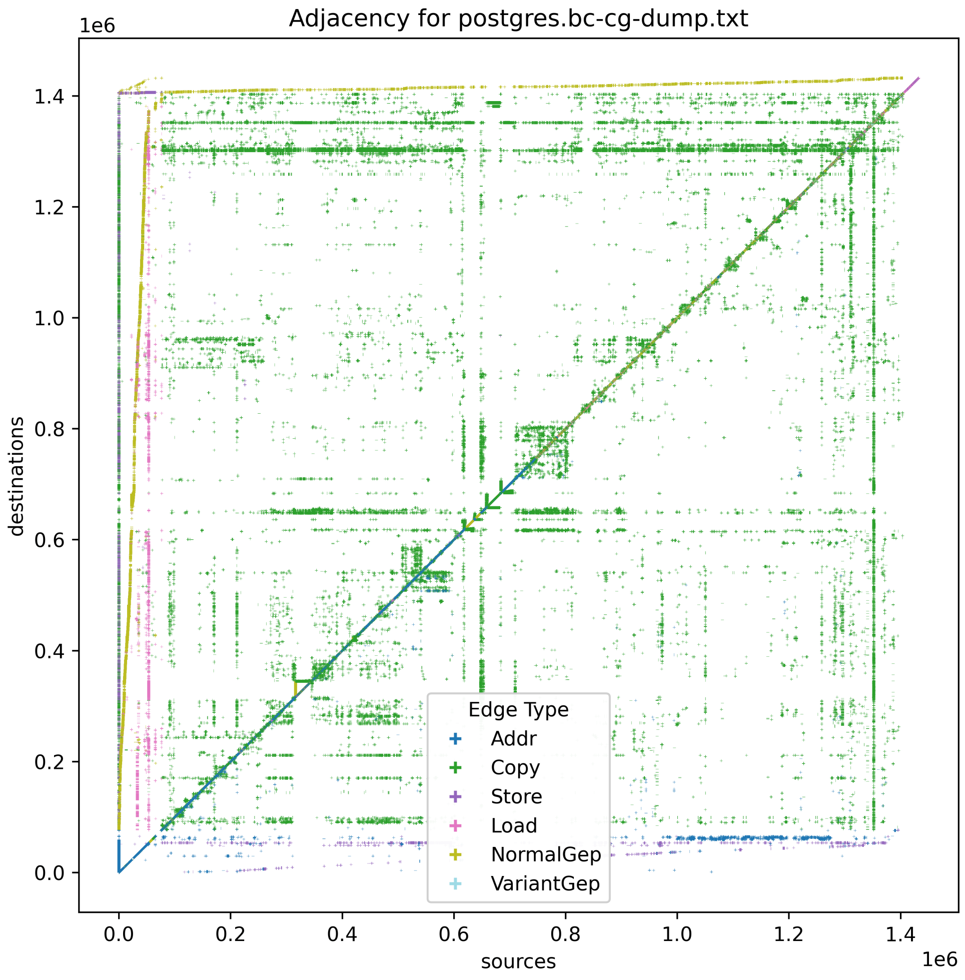


Figure A.2: Adjacency Plot for the Constraint Graph of Postgres

List of Figures

List of Tables

List of Algorithms

Bibliography

[AG18] R. Azimov and S. Grigorev. “Context-free path querying by matrix multiplication”. In: *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 2018, pp. 1–10 (cit. on pp. 22, 23).

[And94] L. O. Andersen. “Program analysis and specialization for the C programming language”. PhD thesis. Citeseer, 1994 (cit. on pp. 3, 9).

[GZJ+20] R. Gu, Z. Zuo, X. Jiang, H. Yin, Z. Wang, L. Wang, X. Li, and Y. Huang. “Towards efficient large-scale interprocedural program static analysis on distributed data-parallel computation”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.4 (2020), pp. 867–883 (cit. on pp. 2, 30).

[Hin01] M. Hind. “Pointer analysis: Haven’t we solved this problem yet?” In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 2001, pp. 54–61 (cit. on pp. 4, 8).

[KMZN16] S. Kulkarni, R. Mangal, X. Zhang, and M. Naik. “Accelerating program analyses by cross-program training”. In: *ACM SIGPLAN Notices* 51.10 (2016), pp. 359–377 (cit. on p. 6).

[Lan92] W. Landi. “Undecidability of static analysis”. In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.4 (1992), pp. 323–337 (cit. on pp. 2, 3).

[Lin15] S.-H. Lin. *Alias analysis in LLVM*. Lehigh University, 2015 (cit. on p. 7).

[LSDZ22] Y. Lei, Y. Sui, S. Ding, and Q. Zhang. “Taming transitive redundancy for context-free language reachability”. In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2 (2022), pp. 1556–1582 (cit. on pp. 26, 73).

[LSS+15] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. “In defense of soundiness: A manifesto”. In: *Communications of the ACM* 58.2 (2015), pp. 44–46 (cit. on p. 3).

[MBP12] M. Mendez-Lojo, M. Burtscher, and K. Pingali. “A GPU implementation of inclusion-based points-to analysis”. In: *ACM SIGPLAN Notices* 47.8 (2012), pp. 107–116 (cit. on pp. 30, 42–44, 47, 50, 54, 55, 70, 72, 73).

[MGR13] S. McPeak, C.-H. Gros, and M. K. Ramanathan. “Scalable and incremental software bug detection”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* 2013, pp. 554–564 (cit. on pp. 6, 7).

[MMP10] M. Méndez-Lojo, A. Mathew, and K. Pingali. “Parallel inclusion-based points-to analysis”. In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications.* 2010, pp. 428–443 (cit. on pp. 41, 70).

[MP21] A. A. Mathiasen and A. Pavlogiannis. “The fine-grained and parallel complexity of Andersen’s pointer analysis”. In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021), pp. 1–29 (cit. on pp. 3, 74).

[MSS+19] N. Mishin, I. Sokolov, E. Spirin, V. Kutuev, E. Nemchinov, S. Gorbatyuk, and S. Grigorev. “Evaluation of the context-free path querying algorithm based on matrix multiplication”. In: *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA).* 2019, pp. 1–5 (cit. on p. 26).

[Nik16] S. Nikolay. *Beyond GPU memory limits with unified memory on pascal.* NVIDIA Developer Blog, Dec. 2016. URL: `https://developer.nvidia.com/blog/beyond-gpu-memory-limits-unified-memory-pascal/` (visited on 11/16/2022) (cit. on p. 39).

[OEAG20] E. Orachev, I. Epelbaum, R. Azimov, and S. Grigorev. “Context-free path querying by kronecker product”. In: *European Conference on Advances in Databases and Information Systems.* Springer. 2020, pp. 49–59 (cit. on p. 26).

[OKKG21] E. Orachev, M. Karpenko, A. Khoroshev, and S. Grigorev. “SPbLA: The Library of GPGPU-Powered Sparse Boolean Linear Algebra Operations”. In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* IEEE. 2021, pp. 272–275 (cit. on p. 26).

[PB09] F. M. Q. Pereira and D. Berlin. “Wave Propagation and Deep Propagation for Pointer Analysis”. In: *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization.* CGO ’09. USA: IEEE Computer Society, 2009, pp. 126–135. ISBN: 9780769535760. DOI: `10.1109/CGO.2009.9`. URL: `https://doi.org/10.1109/CGO.2009.9` (cit. on p. 12).

[Pra20] G. Pradeep. *CUDA Refresher: The CUDA Programming Model.* NVIDIA Developer Blog, June 2020. URL: `https://developer.nvidia.com/blog/beyond-gpu-memory-limits-unified-memory-pascal/` (visited on 08/12/2022) (cit. on p. 39).

[Rep98] T. Reps. “Program analysis via graph reachability”. In: *Information and software technology* 40.11-12 (1998), pp. 701–726 (cit. on pp. 19, 21).

[SB+15] Y. Smaragdakis, G. Balatsouras, et al. “Pointer analysis”. In: *Foundations and Trends® in Programming Languages* 2.1 (2015), pp. 1–69 (cit. on p. 9).

[SKB14] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras. “Introspective analysis: context-sensitivity, across the board”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2014, pp. 485–495 (cit. on p. 9).

[Ste96] B. Steensgaard. “Points-to analysis in almost linear time”. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 1996, pp. 32–41 (cit. on p. 10).

[SX16] Y. Sui and J. Xue. “SVF: interprocedural static value-flow analysis in LLVM”. In: *Proceedings of the 25th international conference on compiler construction.* ACM. 2016, pp. 265–266 (cit. on pp. 27, 28, 35, 41).

[SX18] Y. Sui and J. Xue. “Value-flow-based demand-driven pointer analysis for C and C++”. In: *IEEE Transactions on Software Engineering* 46.8 (2018), pp. 812–835 (cit. on p. 27).

[SXW+18] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang. “Pinpoint: Fast and precise sparse value flow analysis for million lines of code”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2018, pp. 693–706 (cit. on p. 27).

[SYX14a] Y. Su, D. Ye, and J. Xue. “Parallel pointer analysis with CFL-reachability”. In: *2014 43rd International Conference on Parallel Processing.* IEEE. 2014, pp. 451–460 (cit. on p. 55).

[SYX14b] Y. Sui, D. Ye, and J. Xue. “Detecting memory leaks statically with full-sparse value-flow analysis”. In: *IEEE Transactions on Software Engineering* 40.2 (2014), pp. 107–122 (cit. on pp. 27, 28).

[SYXL15] Y. Su, D. Ye, J. Xue, and X.-K. Liao. “An efficient GPU implementation of inclusion-based pointer analysis”. In: *IEEE Transactions on Parallel and Distributed Systems* 27.2 (2015), pp. 353–366 (cit. on pp. 2, 70).

[TG17] J. Toman and D. Grossman. “Taming the static analysis beast”. In: *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017 (cit. on p. 6).

[WHZ+17] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani. “Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code”. In: *ACM SIGARCH Computer Architecture News* 45.1 (2017), pp. 389–404 (cit. on p. 30).

[ZR08] X. Zheng and R. Rugina. “Demand-driven alias analysis for C”. In: *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2008, pp. 197–208 (cit. on p. 22).

[ZWH+21] Z. Zuo, K. Wang, A. Hussain, A. A. Sani, Y. Zhang, S. Lu, W. Dou, L. Wang, X. Li, C. Wang, et al. “Systemizing Interprocedural Static Analysis of Large-scale Systems Code with Graspan”. In: *ACM Transactions on Computer Systems (TOCS)* 38.1-2 (2021), pp. 1–39 (cit. on pp. 30, 31, 55, 72).