SECURING COMPUTATIONS WITH GPUS

by

Jun Duan

APPROVED BY SUPERVISORY COMMITTEE:

Kevin W. Hamlen, Chair

Gopal Gupta

Murat Kantarcioglu

Shuang Hao

The dissertation is dedicated to

everybody who has ever inspired and supported me through this journey.

SECURING COMPUTATIONS WITH GPUS

by

JUN DUAN, BS, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

May 2021

ACKNOWLEDGMENTS

First, the author feels forever grateful to his advisor, Dr. Kevin W. Hamlen, for all the guidance, inspiration and support through the whole program. The professor's wisdom enlightens the author in so many ways. He detangles situations and offers him perspicuous explanations no matter how baffling they are. His logical thinking inspires the author to reason on facts elegantly when he approaches problems. His persistence and passion for his work encourages the author to do the same in the future. His light sense of humour makes every meeting with him relaxing and pleasant. The author is really blessed to have been working with him. Without any of those, he would not achieve this academic accomplishment.

The author is very thankful to the other two co-authors of the past publications in the PhD degree program. His teammate Benjamin Ferrell is a sincere friend of the author's and always treats projects in earnest. The professor Dr. Kang Zhang is thoughtful and warm-hearted. He gives insightfull opinions and patiently discusses every detail with the author in the work.

The author also thanks the members of the supervising committee of his dissertation: Dr. Gopal Gupta, Dr. Murat Kantarcioglu, and Dr. Shuang Hao for their time to review his dissertation and attend his exams. He is also grateful to Dr. Bhavani Thuraisingham, Ms. Rhonda Walls, and Mr. Douglas Hyde for all the assistance on administrative affairs in the program.

He thanks all the great friends and colleagues whom he knows through the program: Wenhao Wang, Huseyin Ulusoy, Xiaoyang Xu, Erman Pattuk, Shamila Wickramasuriya, Masoud Ghaffarinia, Gilmore Lundquist, Erick Bauman, Imrul Anindya, Yasmeen Alufaisan, Frederico Araujo, Meera Sridhar, Harichandan Roy, and Dakota James Fisher. This journey would not be this wonderful without them.

The author, foremost, expresses his deep gratitude to his family. They share every predicament which they endure and every moment of joy which they celebrate. Without their support, the author could not reach the goals of his life.

The research work herein is supported in part by ONR award N00014-17-1-2995, NSF award #1513704, #1054629, AFOSR award FA9550-14-1-0173, an NSF I/UCRC award from Lockheed Martin, and an endowment from the Eugene McDermott Foundation. The author thanks all the above-mentioned funders for their support through the program. Any opinions or recommendations expressed are those of the author and not necessarily of the above supporters.

Last but not least, the author thanks The University of Texas at Dallas and Department of Computer Science very much for offering him this invaluable opportunity and perfect environment to study and live here and make him an amazing experience at the last stop of his student life.

April 2021

SECURING COMPUTATIONS WITH GPUS

Jun Duan, PhD
The University of Texas at Dallas, 2021

Supervising Professor: Kevin W. Hamlen, Chair

Many modern computers have two diverse computing models baked-in: CPUs and GPUs. The diversity of these computing models arises from their historically disparate purposes: Early CPU designs focused on rapidly executing sequences of scalar operations, whereas GPU designs have focused on computing matrices (e.g., of pixels) through highly parallel computation. This dissertation proposes that this natural computational diversity has a secondary benefit that has gone underutilized: the potential to harden computations on both CPUs and GPUs against cyberattacks, and to elevate assurance of computational correctness, safety, and security through simultaneous CPU-GPU computation.

To explore this thesis, the dissertation first addresses the challenge of bringing machine-checked formal methods assurances and tools to the domain of GPU architectures and computations. A prototype framework for formal, machine-checked validation of GPU pseudo-assembly code algorithms using the Coq proof assistant is therefore presented and discussed. The framework is the first to afford GPU programmers a reliable means of formally machine-validating high-assurance GPU computations without trusting any specific source-to-assembly compilation toolchain. A formal operational semantics for the PTX pseudo-assembly language is expressed as inductive, dependent Coq types, facilitating development of proofs and proof tactics that refer directly to compiled PTX object code.

Secondly, a method of detecting malicious intrusions and runtime faults in software is proposed, which replicates untrusted computations onto the diverse but often co-located instruction architectures implemented by CPUs and GPUs. Divergence between the replicated computations signals an intrusion or fault. A prototype implementation for Java demonstrates that the approach is realizable in practice, and can successfully detect exploitation of Java VM and runtime system vulnerabilities even when the vulnerabilities are not known in advance to defenders. It is shown that GPU parallelism can be leveraged to rapidly validate CPU computations that would otherwise exhibit unacceptable performance if executed on GPU alone.

Finally, GPU-based security is explored at the application level through the introduction of a computational methodology for reorienting, repositioning, and merging camera positions within a region under surveillance, so as to optimally cover all features of interest without overburdening human or machine analysts with an overabundance of video information. This streamlines many video monitoring applications, which are often hampered by the difficulty of manually identifying the few specific locations or frames relevant to a particular inquiry from a vast sea of scenes or recorded video clips. The approach ameliorates this problem by considering geographic information to select ideal locations and orientations of camera positions to fully cover the span without losing key visual details.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

This chapter introduces the prevalent viewpoints and current problems in the related research fields, and positions this dissertation within that landscape. It first considers the status of formal Verification & Validation (V&V) research with respect to GPUs, arguing that there is a paucity of such work for formal GPU computation verification. It then introduces the classic concept of n-variant systems for computer security, and posits that CPU-GPU computational diversity offer exceptional promise as a practical foundation for realizing n-variant assurances in modern computing systems. Finally, research problems around camera surveillance are shown as possible applications in CPU and GPU hybrid scenes. The chapter concludes with some insights of the author's work and an overview of the remainder of the dissertation.

1.1 Formal Validation & GPUs

Machine-checked formal verification of software has become the gold standard for developing high-assurance algorithms and implementations. In contrast to unit testing or fuzzing, which can only verify that software in a particular operating environment behaves correctly on a particular (usually finite) set of inputs, formal methods approaches build formal mathematical proofs that establish that the software obeys specified correctness, safety, and security properties for *every* possible environment and input within the (usually infinite) domain admitted by the specification. Unlike manual code reviews, these proofs are checked fully automatically within a mechanized proof environment whose soundness has been verified down to the foundations of mathematics. And unlike model-based verification, the proofs concern the actual software implementation rather than an idealized model of it.

These contrasts are particularly salient when verifying security properties, since cyber criminals have a well established record of success at identifying dangerous inputs that testing

missed, differences between the testing environment and the deployment environment that can be exploited to compromise software, obscure code vulnerabilities that human reviewers overlooked, and inconsistencies between actual software implementations and the models that attempt to approximate them. As a result, formal methods approaches offer significantly higher assurance than these alternatives.

The price of formal methods is typically the much greater effort and expertise required relative to less rigorous approaches. However, the rise of program-proof co-development environments, such as the Coq proof assistant (Bertot and Castéran, 2004), has made machine-validation of larger software systems much more feasible than in previous decades. For instance, in recent years Coq has been used to verify correctness of a full C compiler (Boldo et al., 2013), prove update consistency of software-defined networks (Reitblatt et al., 2012), model significant subsets of both the Intel x86 and the ARMv8 architectures (Flur et al., 2016; Kennedy et al., 2013), validate a machine language certifier for Google's Native Client architecture (Morrisett et al., 2012), and develop the F* language for secure distributed programming (Swamy et al., 2011), among many other successful applications.

The rise of GPU architectures as general-purpose computing platforms has made it increasingly desirable to make such validation approaches available for GPU algorithms and their implementations, toward developing high-assurance, GPU-based software. Many general-purpose GPU (GPGPU) programming tasks can benefit from such assurance; for example, GPUs are already being leveraged to more efficiently realize cryptography (Yamanouchi, 2007), scan for viruses (Seamans and Alexander, 2007), perform financial computations (Grauer-Gray et al., 2013), spot network intrusions (Alshawabkeh et al., 2010), and even detect underwater seismic faults (Deschizeaux and Blanc, 2007). Unforeseen flaws in these computations could have severe ramifications for users and the general public. Since highly parallelized architectures are notoriously difficult for humans to reason about, it is especially important to develop machine-validation approaches for analyzing the correctness of such software.

1.2 Pros & Cons of Machine-validating GPU Assemblies

prosandcons

Motivated by this need, we have developed the first encoding of a GPU pseudo-assembly language operational semantics in Coq, and used it to machine-validate some correctness properties of small GPU programs. Our prototype framework targets Nvidia's CUDA platform (though we anticipate that our general approach is extensible to other GPU architectures). In particular, our semantics model a Single Instruction, Multiple Threads (SIMT) architecture with warps, thread blocks, and grids.

In order to minimize the *trusted computing base* (TCB) of our validation framework, and to offer developers maximum flexibility, theorems and proofs in our framework operate at the level of Parallel Thread eXecution (PTX) pseudo-assembly language computations rather than source code programs. This frees software developers to implement their high-assurance algorithms in any available source language, and use any compilation tools, provided that the tools ultimately yield PTX programs as output.

The downside of this choice, of course, is that proofs must reason at a significantly lower level than source code, and can therefore require extra effort to formulate. We believe this is nevertheless a wise design decision in the long term, because foundational support for PTX can eventually be scaled up to higher levels of abstraction via future development of Coq theories and tactics that modularize and automate much of the proof work for common source idioms. Similar trade-offs have motivated prior work on modeling ISAs of other low-level architectures in Coq (e.g., (Atkey, 2007; Flur et al., 2016; Kennedy et al., 2013)).

As an example of this sort of scaling, a major success of our work is the formulation and validation of the first mechanized proof that CUDA's memory synchronization model ensures transparency of the thread scheduler. In particular, correctness of a computation under the assumption of a deterministic scheduler always implies correctness under a non-deterministic scheduler. This result greatly simplifies proofs of PTX code correctness by eliminating and

abstracting away the non-deterministic scheduler from the sources of parallelism that proofs must consider.

1.3 *N*-variant Systems

While formal methods offers the highest possible assurance that a given software implementation satisfies a given formal specification, many software systems lack any formal (i.e., machine-readable) specification. Their intended functionalities are expressed informally as natural language descriptions, whose imprecision is regularly exploited by adversaries to carry out cyberattacks. Securing this significant body of software therefore necessarily entails less rigorous protections. However, even without a formal specification of correctness, it is often possible to formally specify *inconsistency* of arbitrary software systems, which is a root of many vulnerabilities and critical failures.

N-variant systems (Cox et al., 2006) detect intrusions and other runtime anomalies in software by deploying diverse replicas of the software and monitoring their parallel execution for inconsistencies, which manifest as computational divergences. Divergence between the computations indicates that one or more replicas have exercised functionalities that were unintended by the program's developers, and that were therefore not replicated consistently across all the copies. The *n*-variant approach has been used for detecting memory corruption vulnerabilities in C/C++ programs (Volckaert et al., 2017), monitoring user-space processes (Salamat et al., 2009), defending data corruption attacks (Nguyen-Tuong et al., 2008), and securing embedded systems (Alkabani and Koushanfar, 2008).

Unfortunately, one major barrier to the realization of effective *n*-variant systems in practice has been the high difficulty and cost associated with creating and maintaining software copies that are appropriately diverse (not replicating bugs or vulnerabilities), yet consistent (preserving all desired program features). Achieving this can entail employing multiple in-

dependent software development teams, which can potentially multiply the cost and time associated with the project by a factor of n (Avižienis, 1985).

This high cost of independent, manual cultivation of diversity has led to a search for automated software diversity. For example, compilers have been proposed as natural diversity-introduction vehicles (Jackson et al., 2011), since they enjoy a range of options when translating source programs to distributable object code, including various possible object code and process memory layouts. However, many large classes of software attacks exploit low-level details that are fundamental to the target hardware architecture, and that are therefore difficult for compilers to meaningfully diversify. For example, Address Space Layout Randomization (ASLR) defenses, which randomize section base addresses in process memory at load-time, have proven vulnerable to derandomization attacks (Shacham et al., 2004) that exploit the prevalence of relative-address instruction operands in CISC instruction sets to learn the randomized addresses. Similarly, return-oriented programming (Shacham, 2007) and counterfeit object-oriented programming attacks (Schuster et al., 2015) abuse the semantics of return and call instructions, which is difficult to avoid when compiling to architectures with those instruction semantics.

1.4 Execution Divergence between CPUs and GPUs

The research in this dissertation is inspired by the observation that modern computing systems increasingly have two very different yet powerful instruction architectures available to them: CPU and GPU. This potential source of computational diversity has gone relatively unutilized as an opportunity to detect malicious software intrusions through n-variant computation. To explore this opportunity, we introduce Java Gpu-Assisted N-variant Guardian (J-Gang), a system that replicates Java computations onto CPU-GPU hybrid architectures and runs them concurrently in order to detect divergence-causing intrusions.

GPU computing models differ substantially from typical CPU computing models. This diversity offers many attractive opportunities for robust intrusion detection, but is also a source of significant technical challenges. For example, GPU architectures often suffer poor performance on computations with few threads; their advantages are only seen on computations with hundreds or thousands of simple but independent workloads. However, most CPU computations offer only limited parallelism on the order of a few threads. This makes running CPU computations in a brute-force fashion on GPUs impractical; it risks bottlenecking the system by lagging the GPU variant hopelessly behind the CPU variant, reducing both to a crawl.

To address these challenges, our research proposes a novel "trust but verify" n-variant architecture wherein GPU computations enjoy increasing parallelism opportunities the farther they lag behind CPU computations. This offers the former a means to keep pace with even the most serial workloads, allowing the framework to secure a broad domain of realistic computations without introducing unacceptable performance overhead.

1.5 Surveillance & Camera Positioning

Visual surveillance and camera positioning is a prime example of security-sensitive GPU computation in practice. This dissertation considers GPU computational security within the context of physical security by presenting a secure algorithm for positioning and orienting a limited number of video monitoring devices so as to maximize the information gain on the status of a pre-mapped route. The approach accommodates contexts in which both cameras and objects are mobile, and considers several applications, including VR scene generation, optimizing traffic monitoring to minimize coverage loss, audiovisual fingerprinting, and automatic scene detection.

1.6 Insight into our work

1.6.1 CUDA au Coq

Our work, CUDA au Coq (Ferrell et al., 2019), complements related works on GPU software debugging through unit testing (Holey et al., 2013; Zheng et al., 2011) or heuristic static analysis of source code (Coutinho et al., 2011) by offering a higher level of assurance than these traditional approaches can provide, but at the cost of (possibly significant) extra validation effort. That is, we anticipate that a typical development workflow for high-assurance GPU software should first employ these heuristic debugging techniques to identify and fix any demonstrable flaws, and then proceed to apply formal methods machine-validation to obtain complete proofs of correctness. Our work is an alternative to any runtime or hybrid fault detection approaches (Li et al., 2014; Zheng et al., 2011), since it imposes no runtime overhead (all validation is strictly static), and yields *a priori* guarantees that span the universe of all possible execution traces for verified code.

1.6.2 J-Gang

Our approach, J-Gang (Duan et al., 2019), therefore instead adopts an asynchronized model in which the CPU variant runs at full speed, logging its results at selected program *checkpoints* in the form of JVM state snapshots. A sequence $(\sigma_0, \ldots, \sigma_k)$ of such snapshots can be replicated and validated by a GPU using k concurrent workers, each of which validates the (σ_i, σ_{i+1}) portion of the computation by starting at state σ_i as a pre-condition and confirming that it reaches state σ_{i+1} as a post-condition $(\forall i \in 0..k-1)$. The computation is correct only if all these fragments pass validation. This allows the GPU to catch up to the CPU computation in spurts—the more it lags behind, the more opportunity for parallelism arises, since it can greedily consume more snapshots and validate them concurrently.

The high dissimilarity between GPU and CPU models of Java computation state allow J-Gang to detect many important vulnerability classes. For example, attacks that exploit

memory corruption vulnerabilities to hijack return addresses on the stack have a different effect upon GPU computations, since the GPU model has no explicit call stack with in-memory return addresses to corrupt. Moreover, our detection approach conservatively assumes that all exploited vulnerabilities are unknown to defenders (zero-days). No explicit knowledge of vulnerabilities is used to avoid preserving them in the GPU replica; divergences arise purely from the natural dissimilarity between the two instruction architectures.

1.6.3 VisualVital

Unlike prior work on related problems (see Chapter 5), our goal in the VisualVital (Duan et al., 2017) project is to retain and maximize overall information related to world status, not to specialize for a particular, known detail or event. For instance, prior work on traffic monitoring has demonstrated the effectiveness of probe vehicles and smart-phones for helping users avoid specific, known problem states (e.g., traffic congestion) (Astarita et al., 2014; Hoh et al., 2012); and Virtual trip lines (Hoh et al., 2008) can help travelers choose effective routes. But such approaches are not as applicable to identifying and analyzing arbitrary event details in which the user's interest cannot be predicted in advance (e.g., terrorist attack details whose relevance only become evident afterward). Our problem also generalizes beyond traffic scenarios.

Specifically, we consider the problem of observing a route (consisting of many segments, each with differing relevant details labeled) with a limited number of camera positions, so as to maximize the overall details observed by selecting optimal positions for the cameras. Each camera must fit in the model from the real world—the level of detail observed by each camera is inversely related to its geometric distance from the object. To maximize the overall information, we invent a method to first maximize local details with sufficient positions, and then gradually merge the selected positions to decease the camera count until it meets a desired target threshold.

1.6.4 Roadmap

The reminder of this dissertation is arranged as follows. Chapter 2 presents CUDA au Coq, the framework for machine-validating GPU assembly programs. Chapter 3 presents J-Gang, the n-variant framework of verification for Java source code on CPU and GPU hybrid platform. Chapter 4 presents VisualVital, the observation model for multiple sections of scenes. Chapter 5 lists all the related work. Chapter 6 concludes. At last, Appendix shows in details about the source code of Coq for the CUDA au Coq project, including the code and proofs of the PTX model, and explanation and proofs of the observation model for the VisualVital project.

CHAPTER 2

CUDA AU COQ[1]

2.1 Overview

Motivated by the importance of the correctness of software with parallel features and the extreme difficulties it poses for human reasoning without automated assistance, this chapter presents the GPU computation model encoded in Coq proof system and the procedures of validating parallel programs by machine. Nvidia's CUDA platform is selected as the target architecture, and we here limit our attention to the SIMT architecture, including SIMD. Architectures with mixed parallel features, such as Multiple Instruction, Multiple DATA (MIMD) or (Simultaneous Multithreading) SMT, are reserved for future work.

To build the model in Coq, the layer structures of CUDA's parallel computation, such as warps, thread blocks, and grids, are defined and rules of computation are dependently created and proved. The correctness of the model is listed in the contributions and shown in following sections. CUDA'S Intermediate Representation (IR), PTX, offers a unification of high-level languages running above it, so performing verification at the PTX level minimizes the Trusted Computing Base (TCB) of the system. The framework hence accommodates any high-level language that CUDA supports; code developers need only supply the low-level (compiled) code to check the correctness through this model. Our goal in this project is to offer a semi-automated framework whereby a human expert can write machine-checked proofs of PTX code properties. Automation of the proof search effort is the subject of related research.

Our prototype implementation in Coq includes 350 SLOC for the PTX model, 300 SLOC for theorems, and 140 SLOC of Ltacs. It is beneficial for our model to be as small as

[1]The material in this chapter was originally published as: Benjamin Ferrell, Jun Duan, and Kevin W. Hamlen, "CUDA au Coq: A Framework for Machine-validating GPU Assembly Programs," In *Proceedings of the 26th Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 474–479, March 2019.

possible in order to minimize the TCB. However, there is no limit necessitated on the number of theorems, as these only strengthen the implementation and shorten proofs without contributing to the TCB.

In summary, our contributions are as follows:

- We present the first machine-checkable formal validation framework for a GPU architecture, using the Coq proof assistant.
- Our framework formalizes Nvidia's CUDA architecture and the PTX pseudo-assembly instruction set as Coq inductive types and definitions, for formal deductive reasoning in Coq's interactive proof environment.
- The formalization supports all major forms of CUDA parallelism, including threads, warps, thread blocks, and grids. (Non-standardized, implementation-defined characteristics of the architecture are intentionally not modeled, so that proofs must establish correctness independently from architectural idiosyncrasies that may be unreliable across GPUs.)
- We proved and machine-validated a theorem establishing that CUDA's memory synchronization model successfully makes thread scheduling details transparent to PTX programs. Thus, correctness proofs can consider a simplified architectural model in which the scheduler is deterministic.
- Example theorems and proofs demonstrate how our prototype can be applied successfully to small but realistic GPU programs.

We proceed as follows: Section 2.2 begins with background material on the CUDA architecture that undergirds our approach to the formalization. Our formalized PTX semantics are presented in Section 2.3. Example theorems and proofs that demonstrate the utility of our formalization follow in Section 2.4. Section 2.5 summarize the chapter.

2.2 Background

Our semantics are based on PTX, which is an intermediate assembly language either produced from compiling CUDA C/C++ code or manual encoding. As a common instruction set with well-documented semantics, PTX insulates developers from the constantly evolving details of each new GPU hardware release. It is the lowest intermediate representation that abstracts away from GPU-specific details that cannot be relied upon for forward compatibility.

At run-time, the intermediate assembly language is further compiled by the device driver down to the appropriate machine code, which is then executed on the device. Since our focus is on PTX, we are not concerned with the CUDA to PTX compilation process, but we do trust the final PTX to machine code compilation process. PTX is well documented (nVIDIA, 2015) so our goal is to formalize its execution semantics in Coq in a clean, succinct fashion conducive to writing tractable proofs.

At a high level, CUDA GPUs systematically execute millions of threads to accomplish a task. The number of threads spawned is defined by two user-configurable parameters, `grid_size` and `block_size`, which are 3-dimensional vectors. When a job is dispatched to the GPU, a `grid_size` array of *thread blocks* are spawned with each thread block containing a `block_size` array of threads.

Thread blocks are scheduled to execute on various *streaming multiprocessors* (SMs). A GPU typically contains multiple SMs, which are each composed of one or more CUDA cores (ALUs), Load/Store units, and special function units. When a block is scheduled on an SM, threads are grouped into *warps* (sets of 32 threads) and scheduled for execution. Each time a warp executes an instruction, all threads do so in lock-step.

GPU memory is also hierarchically structured. At the top level are *global* and *constant memory*, to which all threads have access. SMs have a section of cache-like memory called

shared memory, to which all threads in the same thread block have quick access. At the bottom level are *local* and *register memory*, which each thread has in a private section.

2.3 Technical Approach

To describe our formal PTX model, we start from the bottom and work upward. It includes data types, registers, special registers, operands, memory, instructions, threads, warps, blocks, and grids. For this prototype, we currently support succinct PTX functions and try to add more to make the concept more complete. First we introduce abstract definitions with basic types of the Coq system. For a complete definition of our model in Coq, see the appendix.

2.3.1 Data Types

As the most basic units for this model, we model PTX types *dty* within Coq as a sum type consisting of unsigned integers (UI), signed integers (SI) and byte data (B), each parameterized by a bit width w.

$$w : \mathbb{N}$$

$$dty : \{\text{UI, SI, BD}\} \times \mathbb{N}$$

An ID *id* is a label to uniquely mark a storing unit or differentiate operational modules in the system.

$$id : \{\text{Id}\} \times \mathbb{N}$$

2.3.2 Memory

In order to properly capture how GPU memory semantically operates, there are a couple of elements to consider.

First, GPU memory is made up of different sections, or *state-spaces* (ss), each of which have a specific purpose. Due to its accessibility for threads and some restrictions, some memory can be manipulated by only one specific thread, some can be shared by the threads inside the same block, some can be only readable, and the others have well-defined features for more purposes. We will focus on three types state-spaces for memory: `global`, `const`, and `shared`.

`global` memory can be read or written by all threads, and there is no hardware synchronization mechanism restricting accesses.

`const` memory is read-only and shared by all threads.

`shared` memory can be read or written by all threads in the same block, and hardware synchronization is performed through memory barriers.

To match each type with an address, we model memory addresses *addr* as location-accessability pairs.

$$\begin{aligned} bid &: \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\ ss &: \{\text{Global, Const, Shared}\} \times bid \\ addr &: ss \times \mathbb{N} \end{aligned}$$

Second, many memory transactions are performed during execution without any explicit order, possibly introducing memory synchronization errors.

Taking into consideration these two items, we define memory μ to be a mapping from state-space and address to byte and boolean. The boolean value specifies whether a byte is valid or could possibly still be in flight—similar to a valid bit for cache memory.

$$\mu : (ss \times addr) \to (byte \times \mathbb{B})$$

At the launch of a program, only `global` and `constant` memory may have data, and their valid bits are set to `true`. During execution, global memory periodically updates as values are stored. Its valid bits are always `false`, since the hardware does not guarantee memory synchronization (excepting atomic instructions).

GPUs leverage data-independence to achieve high parallelism, so proper `global` memory synchronization is often a prerequisite for code correctness. This is a perennial source of GPU algorithm bugs, so our memory formalization is designed to support formal verification of such properties.

Thread intercommunication is possible only in thread blocks with the use of `shared` memory, which can be synchronized by barriers. A thread block performs some computation and threads begin to arrive at a barrier and wait. Once all threads have reached the barrier, the values stored in `shared` memory during this time are guaranteed to be valid when execution resumes. To model this, our semantics initially set a value's valid bit to `false` and switch to `true` when the entire block has reached the barrier.

2.3.3 Registers

PTX code computes with four different data types: unsigned integers, signed integers, floats, and untyped bytes. In this work we only consider unsigned (UI) and signed integers (SI), but our approach can be extended to include all data types.

Threads are allocated a private register file which contains a set PTX registers. A *register* (reg) holds temporary values during execution, and is uniquely identified by its data type, bit width, and index. We define the *register file* ρ to be a mapping from registers to integers.

$$reg : \{\text{UI}, \text{SI}\} \times \mathbb{N} \times \mathbb{N}$$

$$\rho : reg \to \mathbb{Z}$$

During program execution, not all SIMT threads follow the same path. Threads maintain a set of predicate registers, which optionally prefix any instruction and indicate whether it

should be executed or skipped. The case where the predicate is false is semantically equivalent to a Nop. In our implementation, we only consider branch instructions to optionally have prefixed predicates, so we introduce a pseudo-instruction to distinguish these from non-predicated branches. Coupled with the register file is a *predicate state* φ, which maps predicate indexes to boolean values.

$$\varphi : \mathbb{N} \to \mathbb{B}$$

2.3.4 Special Registers

Special registers contain static information about the grid configuration and a thread's location (i.e., index). They are unique to GPUs, and are useful when delegating work to each thread. There are four predominant special registers, each a 3-dimensional vector: the thread-index (T), block-index (B), block-size (NT), and grid-size (NB). Every thread has a unique combination of thread-index and block-index, but identical block-size and grid-size.

To model this, our state model includes the following *auxiliary function* for special registers:

$$dim : \{\text{Dx, Dy, Dz}\}$$

$$sreg : \{\text{T, B, NT, NB}\} \times dim$$

$$sreg\_aux : tid \to sreg \to \mathbb{N} := get\_sreg(tid)$$

2.3.5 Operands

The types of instruction operands are are inferred during PTX parsing. We define 4 different types based on the origins of different state spaces to support the compatibility for source and destination operands in our model. Specifically, *Reg*, *SReg*, *Imm* and *RegImm* denote register, special register, immediate, and register-immediate operand types, respectively.

Also, there are specific rules to restrict the conversion between different types and source and destination operands of different length.

$$op : reg \uplus sreg \uplus \mathbb{Z} \uplus reg \times \mathbb{Z}$$

2.3.6 Instructions

We next model PTX instructions *instr* within Coq. Instruction parameter types must comply with the syntactic grammar and typing constraints of PTX. Programs running in CUDA can be decompiled into PTX code and translated to fit into our definition.

A program *prg* is modeled as a list of PTX instructions. Operand types within each instruction are checked according to the PTX typing rules translated to Coq. Typing rules for a core subset of instructions are listed in the following definition:

```
Inductive instr :=
| Bop (i:bop) (ty:dty) (d:reg) (a b:op)
| Top (i:top) (ty:dty) (d:reg) (a b c:op)
| Setp (c:cmp) (ty:dty) (p:id) (a b:op)
| Mov (ty:dty) (d:reg) (a:op)
| Ld (stsp:iss) (ty:dty) (d:reg) (a:op)
| St (stsp:iss) (ty:dty) (a:op) (d:reg)
| Bra (tgt:nat)
| PBra (p:id) (tgt:nat)
| Bar | Sync | Nop | Exit.
Definition prg := list instr.
```

2.3.7 Threads

Potentially millions of threads execute on a GPU, so we assign each an enumerated value for unique identification. Proofs do not typically exhaustively enumerate this potentially large

identifier space, of course; the identifiers typically take the form of universally quantified variables in proofs. The identifiers are passed to the auxiliary function in §2.3.4 to obtain the appropriate special registers.

As mentioned in §2.3.3, threads maintain a set of private registers ρ and predicates φ. Therefore, we define a *thread* θ to be a tuple of these three components.

$$\theta : \mathbb{N} \times \rho \times \varphi$$

In the following semantics, vector $\vec{t}$ denotes a thread state, including its ID *tid*, memory state ρ, and predicate state φ.

2.3.8 Warps

A warp is defined as a set of 32 threads, and is the smallest level of granularity to execute on an SM. All threads in a common warp execute the same instruction at the same time (i.e., in lock-step), which is efficient for most instructions except for a predicated branch. With a predicated branch, there is the potential for part of the warp to take the branch and the rest stay behind to execute the next immediate instruction. A warp in this state is called *divergent*, and must now execute the two (or more) paths serially, thereby increasing run-time. At the formal level, we define a warp to be in one of two states: uniform execution of a set of threads or divergent execution of two warps.

```
Inductive warp : Set :=
| Uni (pc:nat) (ts:list thread)
| Div (w1 w2:warp)
```

Hence, warps may form a tree of divergences.

Figure 2.1 lists the small-step semantic rules for warps. The semantics are distinguished by instruction input, which transforms the given warp according to the operational semantics of the instruction. The first 9 rules are fairly straightforward and only apply to uniform

warps. Instructions `Bop` and `Top` are arithmetic operations on two and three inputs, respectively. If a warp is divergent and the instruction is not `Sync`, then the *left*-most warp is executed.

The final case is the most complex due to the synchronization operation. When a warp diverges, it should at some point converge back to a uniform warp through a sync operation (see Figure 2.2). Compilers enforce this because memory barriers can cause undefined execution. In some executions, a warp could diverge with half of it, halting at a barrier while the other half continues to execute and eventually exit. Since all threads must be at the memory barrier in order for it to lift, this situation creates a deadlock and the program hangs or (more likely) crashes. Careful analysis is required to establish that correct code always avoids this situation. Our operational semantic encodings facilitate such reasoning in Coq proofs.

2.3.9 Blocks

Thread blocks are typically defined as sets of threads, but because they are grouped into warps, we formalize them as sets β of warps.

$$\beta : \vec{\omega}$$

Warps are selected by the scheduler to execute an instruction, but the details of the scheduling can vary between GPUs and other contextual factors. Proofs in our framework must therefore establish correctness independently of the scheduling algorithm. Our semantics hence formalize the scheduler non-deterministically.

Looking at Figure 2.3, we have two possible scenarios. In the first scenario, there exists a warp that is not at a memory barrier or has not finished, so it executes its next instruction. Notation $\beta[\omega'/\omega]$ denotes the capture-avoiding substitution of ω with ω', in order to update the block state. The second scenario applies when all warps have reached a memory barrier.

$$\frac{}{\texttt{Nop} \vdash \langle (pc, \vec{t}), \mu\rangle \rightarrow_1 \langle (pc+1, \vec{t}), \mu\rangle}(\text{nop})$$

$$\frac{\vec{t'} = \{(tid, \rho[r \mapsto op(a,b)], \varphi) \mid (tid, \rho, \varphi) \in \vec{t}\}}{\texttt{Bop}\ op\ r\ a\ b \vdash \langle (pc, \vec{t}), \mu\rangle \rightarrow_1 \langle (pc+1, \vec{t'}), \mu\rangle}(\text{bop})$$

$$\frac{\vec{t'} = \{(tid, \rho[r \mapsto op(a,b,c)], \varphi) \mid (tid, \rho, \varphi) \in \vec{t}\}}{\texttt{Top}\ op\ r\ a\ b\ c \vdash \langle (pc, \vec{t}), \mu\rangle \rightarrow_1 \langle (pc+1, \vec{t'}), \mu\rangle}(\text{top})$$

$$\frac{\vec{t'} = \{(tid, \rho[r \mapsto a], \varphi) \mid (tid, \rho, \varphi) \in \vec{t}\}}{\texttt{Mov}\ r\ a \vdash \langle (pc, \vec{t}), \mu\rangle \rightarrow_1 \langle (pc+1, \vec{t'}), \mu\rangle}(\text{mov})$$

$$\frac{\vec{t'} = \{(tid, \rho[\text{r} \mapsto \mu(ss, a)], \varphi) \mid (tid, \rho, \varphi) \in \vec{t}\}}{\texttt{Ld}\ ss\ r\ a \vdash \langle (pc, \vec{t}), \mu\rangle \rightarrow_1 \langle (pc+1, \vec{t'}), \mu\rangle}(\text{ld})$$

$$\frac{\vec{v} = \{(ss, a, \rho(r)) \mid (tid, \rho, \varphi) \in \vec{t}\} \quad \mu' = \text{update}(\mu, \vec{v})}{\texttt{St}\ ss\ a\ r \vdash \langle (pc, \vec{t}), \mu\rangle \rightarrow_1 \langle (pc+1, \vec{t}), \mu'\rangle}(\text{st})$$

$$\frac{}{\texttt{Bra}\ tgt \vdash \langle (pc, \vec{t}), \mu\rangle \rightarrow_1 \langle (tgt, \vec{t}), \mu\rangle}(\text{bra})$$

$$\frac{\vec{t'} = \{(tid, \rho, \varphi[p \mapsto cmp(a,b)]) \mid (tid, \rho, \varphi) \in \vec{t}\}}{\texttt{Setp}\ cmp\ p\ a\ b \vdash \langle (pc, \vec{t}), \mu\rangle \rightarrow_1 \langle (pc+1, \vec{t'}), \mu\rangle}(\text{setp})$$

$$\frac{\begin{array}{c}\vec{t_1} = \{(tid, \rho, \varphi) \mid (tid, \rho, \varphi) \in \vec{t} \wedge \varphi_i(p)\} \\ \vec{t_2} = \{(tid, \rho, \varphi) \mid (tid, \rho, \varphi) \in \vec{t} \wedge \neg\varphi_i(p)\} \\ \omega' = \text{sync}((pc+1, \vec{t_2}), (tgt, \vec{t_1}))\end{array}}{\texttt{PBra}\ p\ tgt \vdash \langle (pc, \vec{t}), \mu\rangle \rightarrow_1 \langle \omega', \mu\rangle}(\text{pbra})$$

$$\frac{i \neq \texttt{Sync} \qquad i \vdash \langle \omega_1, \mu\rangle \rightarrow_1 \langle \omega_1', \mu'\rangle}{i \vdash \langle (\omega_1, \omega_2), \mu\rangle \rightarrow_1 \langle (\omega_1', \omega_2), \mu'\rangle}(\text{div})$$

$$\frac{}{\texttt{Sync} \vdash \langle \omega, \mu\rangle \rightarrow_1 \langle \text{sync}(\omega), \mu\rangle}(\text{sync})$$

Figure 2.1: Warp Small-Step Semantics

At this time all `shared` memory is committed (i.e., all valid bits are set to `true`), and warp program counters are incremented to continue execution.

$$\text{sync}(\omega) = \begin{cases} (pc+1, \vec{t}), & \text{if } \omega = (pc, \vec{t}) \\ \text{sync}(\omega_2), & \text{if } \omega = ((pc_1, \{\}), \omega_2) \\ \text{sync}(\omega_1), & \text{if } \omega = (\omega_1, (pc_2, \{\})) \\ (pc_1 + 1, \vec{t_1} \cup \vec{t_2}), & \text{if } \omega = ((pc_1, \vec{t_1}), (pc_2, \vec{t_2})) \\ & \wedge\ pc_1 = pc_2 \\ (\omega_2, (pc_1, \vec{t_1})), & \text{if } \omega = ((pc_1, \vec{t_1}), \omega_2) \\ (\text{sync}(\omega_1), \omega_2), & \text{otherwise } \omega = (\omega_1, \omega_2) \end{cases}$$

Figure 2.2: Warp Sync Function

$$\frac{\exists \omega \in \beta \,.\, \pi(\omega_{pc}) \notin \{\text{Bar}, \text{Exit}\} \quad \pi(\omega_{pc}) \vdash \langle \omega, \mu \rangle \rightarrow_1 \langle \omega', \mu' \rangle}{\pi \vdash \langle \beta, \mu \rangle \rightarrow_1 \langle \beta[\omega'/\omega], \mu' \rangle}(\text{exec}_b)$$

$$\frac{\forall \omega \in \beta \,.\, \pi(\omega_{pc}) = \text{Bar}}{\pi \vdash \langle \beta, \mu \rangle \rightarrow_1 \langle \text{incr\_pc}(\beta), \text{commit}(\mu) \rangle}(\text{lift-bar})$$

Figure 2.3: Thread Block Small-Step Semantics

2.3.10 Grids

Grid execution is similar to thread block execution in the sense that thread blocks are non-deterministically chosen for execution. There is no hardware based memory synchronization at this level, so Figure 2.4 includes only one derivation rule. The rule chooses an unfinished thread block to execute and updates the grid with the new thread block state.

We define a thread block to be complete when all warps are at an `Exit` instruction. When a grid finishes executing, all thread blocks should be complete.

2.4 Example Validation Results

To demonstrate how our framework facilitates machine-checked validation of GPU software, we walk through the validation of a toy PTX program that sums two vectors. Listing 2.1 is

$$\text{complete}(\pi, \beta) \equiv (\forall \omega \in \beta \;.\; \pi(\omega_{pc}) = \text{ Exit})$$

$$\frac{\begin{array}{c}\exists \beta \in \gamma \;.\; \neg\text{complete}(\pi, \beta) \\ \pi \vdash \langle \beta, \mu \rangle \rightarrow_1 \langle \beta', \mu' \rangle\end{array}}{\pi \vdash \langle \gamma, \mu \rangle \rightarrow_1 \langle \gamma[\beta'/\beta], \mu' \rangle}(\text{exec}_g)$$

Figure 2.4: Grid Small-Step Semantics

the (almost) verbatim PTX code compiled from sources; our only modification is to rename the parameters in lines 10–13 to something more human-readable, for explanatory purposes.

2.4.1 Context Lifting

Listing 2.2 shows our translation of the PTX code to corresponding Coq definitions. At the beginning of all PTX functions is a declaration of the types and quantities of needed registers. We do not necessarily need to translate this to Coq, but for the sake of readability we do so in lines 1–4. Since PTX instructions take operands (not just registers) as input, we use a wrapper to turn all registers into operands (line 6), and prefix the variable with an underscore (e.g. _r1) to make the distinction. Lines 9–12 load the function arguments into registers. Loads have semantics equivalent to Moves in our framework, so the Coq translation uses `Mov` instructions.

All threads execute the same code concurrently, but each receives a distinct thread index, which is computed in lines 14–17. Since each thread is tasked with adding elements from a specific index in the vector, at least `size` threads are spawned, where `size` is the length of the vector. It could be the case that more than `size` threads are spawned, so a bounds check is implemented in lines 19–20. As mentioned in §2.3.8, warps can diverge when multiple execution paths exist. In this example there are two paths with a divergence point at line 20. The matching `Sync` instruction is inserted at line 35 (index 18 in the Coq instruction list).

Listing 2.1: PTX Assembly for Vector Sum

```
1  .reg .pred %p<2>;
2  .reg .u32 %r<9>;
3  .reg .u64 %rd<11>;
4
5
6
7
8
9  ld.param.u64 %rd1, [arr_A];
10 ld.param.u64 %rd2, [arr_B];
11 ld.param.u64 %rd3, [arr_C];
12 ld.param.u32 %r2, [size];
13
14 mov.u32 %r3, %ntid.x;
15 mov.u32 %r4, %ctaid.x;
16 mov.u32 %r5, %tid.x;
17 mad.lo.s32 %r1, %r4, %r3, %r5;
18
19 setp.ge.s32 %p1, %r1, %r2;
20 @%p1 bra BB0_2;
21
22 cvta.to.global.u64 %rd4, %rd1;
23 mul.wide.s32 %rd5, %r1, 4;
24 add.s64 %rd6, %rd4, %rd5;
25 cvta.to.global.u64 %rd7, %rd2;
26 add.s64 %rd8, %rd7, %rd5;
27 ld.global.u32 %r6, [%rd8];
28 ld.global.u32 %r7, [%rd6];
29
30 add.s32 %r8, %r6, %r7;
31 cvta.to.global.u64 %rd9, %rd3;
32 add.s64 %rd10, %rd9, %rd5;
33 st.global.u32 [%rd10], %r8;
34
35
36 BB0_2: ret;
```

Listing 2.2: Coq PTX Assembly

```
Definition r1 : reg := (Ul 32, 1).
Definition r2 : reg := (Ul 32, 2).
...
Definition rd1 : reg := (Ul 64, 1).
...
Definition _r1 : op := Reg r1.
...
Definition add_vector : prg := [
Mov rd1 arr_A;
Mov rd2 arr_B;
Mov rd3 arr_C;
Mov r2 size;

Mov r3 ntid_x;
Mov r4 ctaid_x;
Mov r5 tid_x;
Top MADLO r1 _r4 _r3 _r5;

Setp GE p1 _r1 _r2;
PBra p1 18;


Bop MULWD rd5 _r1 (Imm 4);
Bop ADD rd6 _rd1 _rd5;

Bop ADD rd8 _rd2 _rd5;
Ld Global r6 _rd8;
Ld Global r7 _rd6;

Bop ADD r8 _r6 _r7;

Bop ADD rd10 _rd3 _rd5;
St Global _rd10 r8;

Sync;
Exit ].
```

The translation of lines 22–33 omits the `cvta.to` instructions because they are implicit in our PTX formalization. Specifically, instruction `cvta.to` converts a generic address to a specified state-space, but our framework implicitly does this with the `ld` and `st` instructions,

which both require a state-space parameter. The final PTX instruction is `ret`, which we translate to `Exit` for this example to end the validation at function completion.

2.4.2 Proof Procedure

In Coq we can formally prove and machine-check theorems about the program's behavior on arbitrary inputs, such as termination and correctness of output. Even though this is a simple example, Coq has a rich collection of supporting libraries and theory modules that can be applied to reason about much more complex mathematical properties of larger programs. For example, there is extensive prior work on leveraging Coq to prove properties of cryptographic algorithms (Barthe et al., 2011, 2013; Petcher and Morrisett, 2015). For expository simplicity, we here limit our presentation to proving total correctness of this very simple algorithm.

Listing 2.3: Proving Termination of Vector Sum

```
1  Definition warp_complete (pi : prg) (w : warp) : bool :=
2    match pi (get_pc w) with
3    | Some Exit => true
4    | _ => false
5  end.
6
7  Definition block_complete (pi : prg) (b : block) : bool :=
8    forallb (warp_complete pi) b.
9
10 Definition terminated (pi : prg) (g : grid) : Prop :=
11   forallb (block_complete pi) g = true.
12
13 (* sample parameter configs *)
```

```
14 Definition kc : kconf := ((1,1,1),(32,1,1)).
15 Definition g : grid := generate_grid kc.
16 Definition mu : mem_f := ... (* initial memory state *)
17
18 Inductive grid_t
19 : prg → kconf → grid×mem_f → grid×mem_f → Prop :=
20 ...
21
22 Theorem add_vector_terminates :
23     ∀ (g' : grid) (mu' : mem_f),
24     n_apply 19 (grid_t add_vector kc) (g,mu) (g',mu') →
25     terminated add_vector g'.
26 Proof.
27   intros g' mu' Happ.
28   repeat (unroll_apply Happ).
29   compute. reflexivity.
30 Qed.
```

Listing 2.3 shows how termination is defined and proved for our toy program. A program's execution is considered terminated (or completed) when all threads have reached the `Exit` instruction. Definition `terminated` in the listing checks whether all blocks are complete, which entails confirming whether all warps are complete. We then give a few, but necessary, parameter configurations (`kc`, `g`, and `mu`) to initialize program execution. Finally we define our theorem `add_vector_terminates` by hypothesizing that after 19 small-steps of execution, the program terminates.

Proposition n_apply is inductively defined in Listing 2.5, which relates the number n of applications of a proposition f:A→A→Prop to an input a:A with an output a′:A. At its base case, the number of steps to take is zero, so this is essentially an identity proposition. The inductive case performs one application of f and recursively applies it n − 1 more times.

Proving a theorem in Coq typically begins with introducing universally quantified variables (g′ and mu′) and hypotheses (Happ) into the proof context, as shown. Next, the repeat tactic repeatedly applies a given proof tactic until it fails. The given tactic in this case is unroll_apply, which is defined in Listing 2.5.

Our unroll_apply tactic can be thought of as a primitive symbolic execution engine for PTX. It directly applies the operational semantics of PTX to a proof hypothesis through inversion reasoning, which infers a derivation rule's prerequisites from its consequent. In the case of derivation rules that encode operational semantics, this infers the set of new program states that may result from each instruction's execution, thereby symbolically interpreting the instruction within the proof environment.

In detail, unroll_apply considers the two possible cases of n_apply: either $n = 0$ (no more steps) or $n > 0$. In both cases we start off with the same three tactics: (1) inversion, (2) subst, and (3) clear. The inversion tactic reasons by the distinctness of constructors and only considers cases that could have been used to form the hypothesis. When $n = 0$, the only case that can be used is AppZero and proposes that the input a and output a′ are equivalent. Otherwise, n is positive so AppNext is the only case that applies. Inversion can produce many variable aliases, so we use subst to automatically substitute away any redundant, fresh variable names from the proof context. Finally, we clear the inverted hypothesis from the context once all its effects have been computed and it is no longer needed. Tactic step_grid applies a similar strategy for symbolic execution of grids; we here omit its definition for brevity.

Moving back to the theorem, our symbolic interpreter ultimately reveals the final states of g′ and mu′ and encodes them as hypotheses in the proof environment. The compute

tactic reduces `terminated add_vector g'` to `true = true` and we finally apply `reflexivity` to complete the proof.

Listing 2.4: Proving Correctness of Vector Sum

```
1  Fixpoint get_array (mu : mem_f) (a : addr) (len width : nat)
2   : list Z :=
3     match len with
4     | O => []
5     | S len' => let v := read_mem mu a width in
6                 v :: get_array mu (incr_addr a) len' width
7  end.
8
9  Definition add_array (a b : list Z) : list Z :=
10    let add_pair := fun p => match p with (x,y) => x+y end in
11    map add_pair (combine a b).
12
13 Definition arrA : addr := ...
14 Definition arrB : addr := ...
15 Definition arrC : addr := ...
16 Definition N : nat := numElements.
17
18 Theorem add_vector_correct :
19     ∀ (g' : grid) (mu' : mem_f) (A B C : list Z),
20     n_apply 19 (grid_t add_vector kc) (g,mu) (g',mu') →
21     A = get_array mu arrA N 32 →
22     B = get_array mu arrB N 32 →
```

```
23      C = get_array mu' arrC N 32 →
24      add_array A B = C.
25  Proof.
26    intros g' mu' A B C Happ HA HB HC.
27    repeat unroll_apply Happ.
28    subst. compute. reflexivity.
29  Qed.
```

Listing 2.4 proves partial correctness—i.e., that the result of the computation is the sum of the two input vectors, if it terminates. Coupled with Listing 2.3, this establishes total correctness of the computation. Vectors `A` and `B` come from the initial memory state `mu` and vector `C` is from the final memory state `mu'`. The theorem therefore posits that `A + B = C`.

The proof begins similarly to Listing 2.3, introducing the variables and hypotheses. Then we can apply our `unroll_apply` tactic to obtain the final memory state `mu'`. After some computation, the symbolic interpreter arrives at the desired symbolic expression for `C`, proving the correctness property.

Listing 2.5: Proof Automation Tactics for Symbolic Execution of PTX Code

```
1  Inductive n_apply {A:Type}
2  : nat → (A→A→Prop) → A → A → Prop :=
3  | AppZero (f : A→A→Prop) (a : A) :
4    n_apply 0 f a a
5  | AppNext (n : nat) (a a1 a' : A) (f : A→A→Prop)
6    (Hf : f a a1)
7    (Happ : n_apply n f a1 a') :
8    n_apply (S n) f a a'.
```

```
9
10 Ltac step_grid H := ...
11
12 Ltac unroll_apply H :=
13   match type of H with
14   | n_apply ?n _ _ _ =>
15     match n with
16     | O => inversion H; subst; clear H
17     | _ => let Hgrid := fresh "Hgrid" in
18            let Happ := fresh "Happ" in
19            inversion H as [ | ? ? ? ? ? Hgrid Happ]; subst; clear H;
20            step_grid Hgrid
21     end
22   | _ => fail
23 end.
```

Both of these proofs demonstrate the power of our automation tactics to reliably facilitate formal reasoning about PTX operation. By expressing the symbolic interpreter as proof tactics, we afford users the ability to quickly and easily reduce computations to symbolic expressions within a Coq proof, and subsequently apply the full power of Coq's mathematical theories to reason about the resulting symbolic expressions. This power comes without any additions to the TCB of the framework, since the tactics merely automate the application of the operational semantics rules; they do not introduce new rules that must be checked for accuracy.

2.4.3 Non-deterministic Execution

One of the most difficult aspects of parallel and concurrent programs to reason about is the inter-thread order in which instructions are executed. Explicitly considering all possible executions within proofs is infeasible and unmanageable even for small programs. To ease this burden, we have successfully proved a general theorem showing that the result of a PTX computation is always independent of the order in which the threads of a warp execute. It therefore suffices to only consider a sequential thread execution order within proofs. We here outline this theorem and its proof.

Listing 2.6: Non-deterministic Map

```
1  Inductive nth_ri {A:Type}
2  : nat → list A → A → list A → Prop :=
3  | RI_O a t :
4      nth_ri 0 (a::t) a t
5  | RI_S n t t' x a
6      (Hn: nth_ri n t a t') :
7      nth_ri (S n) (x::t) a (x::t').
8
9  Inductive nd_map {A B : Type}
10 : (A→B) → list A → list B → Prop :=
11 | NDNil (f : A → B) :
12   nd_map f [] []
13 | NDCons (f : A → B) (l l1 : list A) (l2 l' : list B) (a : A) (n : nat)
14   (Hl: nth_ri n l a l1)
15   (Hmap: nd_map f l1 l2)
16   (Hl': nth_ri n l' (f a) l2) :
```

```
17    nd_map f l l'.
```

Listing 2.6 first defines proposition `nth_ri`, which removes an element `a` at position `n` from a given given list `l`, and returns a new list `l'`. We use this definition to create a non-deterministic map function `nd_map`. It is non-deterministic in the sense that the elements are processed in an arbitrary order and not from front to back. This captures all possible thread schedules for warps, which execute threads in lock-step but in an unspecified order. Some components on the SM, such as SFUs, are physically limited in quantity, making it impossible for all threads in a warp to execute on the same clock cycle. Therefore, we seek to prove that a non-deterministic execution is equivalent to a deterministic one.

Listing 2.7: Non-deterministic/Deterministic Equiv

```
1  Theorem nd_map_eq :
2    ∀ (A B : Type) (f : A→B) (l : list A) (l' : list B),
3    nd_map f l l' ⟷ l' = map f l.
4  Proof.
5    intros A B f l l'.
6    split; intros H.
7
8    (* nd_map → map *)
9    induction H.
10     (* Case: NDNil *)
11     reflexivity.
12     (* Case: NDCons *)
13     inversion Hl'; subst; inversion Hl; subst; clear Hl Hl'.
14       (* SCase: n = 0 *)
15       reflexivity.
```

```
16        (* SCase: n is in the middle somewhere *)
17        inversion H3; subst; clear H3 H.
18        simpl. apply list_hd_eq.
19        generalize dependent t.
20        induction Hn0; intros.
21        inversion Hn; subst; clear Hn.
22        reflexivity.
23        inversion Hn; subst; clear Hn.
24        simpl. apply list_hd_eq. eapply IHHn0. exact Hn1.
25
26    (* map → nd_map *)
27    generalize dependent l'.
28    induction l; intros; subst; simpl.
29      (* Case l = [] *)
30      apply NDNil.
31      (* Case l = hd :: tl *)
32      eapply NDCons; try apply RI_O.
33      apply IHl. reflexivity.
34  Qed.
```

Listing 2.7 summarizes our equivalence proof. We start by inducting on the definition of `nd_map`, which generates two cases: the base case `NDNil` and inductive case `NDCons`. The base case is straightforward, so we here focus on the inductive case. It divides into two sub-cases: an element is chosen from the head, or from within the tail. The bulk of the proof focuses on the second sub-case. It leverages dependent inductive reasoning (Cornes and Terrasse, 1995; McBride, 2002) to universally consider all possible thread orderings.

In future work, we plan to extend this result to the warp scheduling algorithm as well, to also eliminate that source of non-determinism from proofs. This result is more challenging because it requires reasoning about the order of memory operations. One reason `nd_map` is equivalent to `map` is because computation does not affect an output at another index. Moving up in the thread hierarchy to thread blocks, one must additionally consider the order of memory operations, which could potentially affect the output of another thread.

2.5 Summary

Our work shows promise in providing a feasible means to machine-verify GPU programs using a well-established, mature proof assistant—the Coq system. Coq's strong, dependent typing system facilitates a natural encoding of GPU assembly code operational semantics as inductive axioms, including modeling its traditionally challenging parallelism and thread scheduling properties. To streamline proofs, Coq's tactic language was leveraged to build a symbolic interpreter that automatically derives provably correct symbolic expressions for assembly code fragments within the proof environment.

Using the resulting framework, we constructed the first machine-checked proof of CUDA thread scheduling transparency, as well as validating correctness properties of some simple programs.

As with any formal validation, our approach does not replace code testing, but can provide stronger guarantees and assurances that do not rely upon comprehensiveness of test sets. By targeting assembly code programs, our framework is applicable to GPU code produced from arbitrary source languages and compilation toolchains. In future work we plan to further ease the task of proof-writing by formulating more extensive theorem libraries and tactic libraries that modularize and automate GPU code correctness proofs.

CHAPTER 3

J-GANG[1]

3.1 Overview

This chapter presents the design, implementation, and evaluation of a framework for translating Java source code to a CPU-GPU hybrid architecture suitable for secure, n-variant computation. Translation entails semantic preservation of the source Java code to each target architecture (CPU and GPU), and verification entails detection of semantic divergence between these two target computations.

The open library for analysis of the Java language, Java Spoon by Inria (Pawlak et al., 2015) and Javassist, is adopted to exact all the data and logic information from original code (or bytecode) both statically and dynamically. Additionally, it is also applied to record local variables for state logging operations. These are re-composed into new code satisfying the grammar rules of the GPU kernel language. After the translation, our system executes the code and the replica in parallel by using Java Aaprapi and OpenCL. Throughout this parallel execution, two series of memory states are logged and compared to detect divergences. To evaluate the prototype implementation, experiments on third-party Java programs demonstrate performance and security effectiveness of the approach.

The contributions of J-GANG can be summarized as follows:

- We introduce the first n-variant system for Java computation verification based on architectural differences between GPU and CPU.
- To harmonize the dissimilar performance advantages of the two architectures, we introduce an asynchronous *trust-but-verify* n-variant model, in which single- or few-

[1]The material in this chapter was originally published as: Jun Duan, Kevin W. Hamlen, and Benjamin Ferrell, "Better Late Than Never: An N-variant Framework of Verification for Java Source Code on CPU×GPU Hybrid Platform," In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pp. 207–208, June 2019.

threaded CPU computations are validated by many-threaded GPU computations on a short delay. This allows the GPU computation to quickly verify many iterations of CPU-executed loops concurrently.

- A prototype implementation establishes rules of translation from Java source code on the host side into GPU-executable kernel code, which offers a possible solution to facilitate GPU execution of general Java source code in future work.

- Evaluation of J-GANG on exploits of eight real-world JVM vulnerabilities exhibits reliable detection at reasonable overheads, even when the vulnerabilities are treated as zero-days (no vulnerability-specific defenses introduced).

- Methods of tracing variables and local data analysis are extensively tested, and we study the connection between overhead and state snapshot logs. Based on the relation, we control overhead for suitable variable-tracking jobs.

The remainder of this chapter is arranged as follows. Section 3.2 describes the system design and defines correctness. Section 3.3 details our prototype implementation. Experimental methodology and evaluation are discussed in Section 3.4. Finally, Section 3.5 summarizes the chapter.

3.2 System Design

3.2.1 Divergence Between Executions

Listing 3.1 exhibits a JVM vulnerability related to around 30 bugs and numerous DoS attacks against Java SE 1.6, and that was later identified as a root cause of array overflows, server VM crashes, and a variety of other potential software compromises before it was patched.[2]

[2] `https://bugs.java.com/view_bug.do?bug_id=5091921`

Listing 3.1: Exploit of JDK-5091921 (JavaSE 1.6, x86/x64 Win7)

```
1 int i = 0;
2 int j = Integer.MAX_VALUE;
3 boolean test = false;
4 while (i >= 0) {
5   i++;
6   if(i > j) {
7     test = true;
8      break;
9   }
10 }
11 System.out.println("Value of i: " + i);
12 if(test) i = 1;
13 System.out.println("Value of i: " + i);
```

Listing 3.2: Exploit of JDK-8189172 (JavaSE 1.8, x86/x64 Win7)

```
1 double b = 1.0 / 3.0;
2 double e = 2.0;
3 double r = Math.pow(b, e);
4 double n = b * b;
5 while (r == n) {
6      b += 1.0 / 3.0;
7      n = Math.pow(b, e);
8      r = b * b; }
9 println("b=" + b + " n=" + n + " r=" + r);
```

It returns different unstable values of i on each execution, and also prints the false result, *"Value of i: 1"* in line 13 when it should report overflowed value -2147483648. The flaw is an incorrect optimization in the (CPU-based) HotSpot compiler, which breaks integer overflow detection in certain loops. However, running this code in our J-GANG system as a GPU computation yields correct results, because GPUs apply a very different procedure for optimizing the loop. This natural difference in behavior offers a potential opportunity to detect the error without advance knowledge of the bug.

Listing 3.2 likewise demonstrates an exploit of JVM bug JDK-8189172,[3] which embodies an imprecision of floating point computations that in this case causes expressions $n \times n$ and

[3] https://bugs.java.com/view_bug.do?bug_id=8189172

n^2 to return unequal results. A correct JVM should loop infinitely, but unpatched JVMs halt with output $b = 4.\overline{9}$, $n = 24.\overline{9}$, $r = 24.999999999999993$. However, compiling the same code to a GPU architecture results in correct behavior—self-product and square yield equal results, and the program loops infinitely. Detecting this divergence of behavior has the potential to detect the exploit without the need to craft and deploy vulnerability-specific mitigations whose formulation require advance knowledge of the bug.

J-GANG detects both exploits by compiling the Java source code to two binary executables: (1) logging-enhanced Java bytecode, and (2) verification-enhanced OpenCL GPU code. The Java bytecode variant logs local state (e.g., variables b, e, r, and n in Listing 3.2) at the start of each loop iteration (line 5) and at loop exit (line 9). The GPU variant consumes this log stream in a verification loop. When consuming k available checkpoints, it spawns $k - 1$ workers that each initialize their local variable states in accordance with different checkpoints σ_i ($i < k - 1$). They then all execute one iteration of the loop in parallel, and confirm that the resulting states matches the succeeding checkpoints σ_{i+1}. The divergence is detected when the final worker obtains a different state than the CPU (e.g., equal values for r and n in Listing 3.2).

While both divergences could theoretically be detected by replicating programs to multiple, dissimilar CPU-based JVMs wherein at least one JVM emulates a GPGPU-style computational model, in practice there are at least two significant problems with this CPU-only approach. First, CPU-based emulation of GPU-style parallelism is highly inefficient. A CPU-based JVM that emulates the computational diversity of a GPGPU computation therefore cannot keep pace with the CPU computation it is seeking to verify, resulting in unacceptable performance bottlenecks.

Second, building and maintaining a new, dissimilar, production-level JVM is difficult and expensive, as witnessed by the fairly small and homogeneous set of production JVMs currently available despite over 25 years of Java infrastructure development. These JVMs intentionally offer little diversity, since diversity introduces maintainability and cross-compatibility

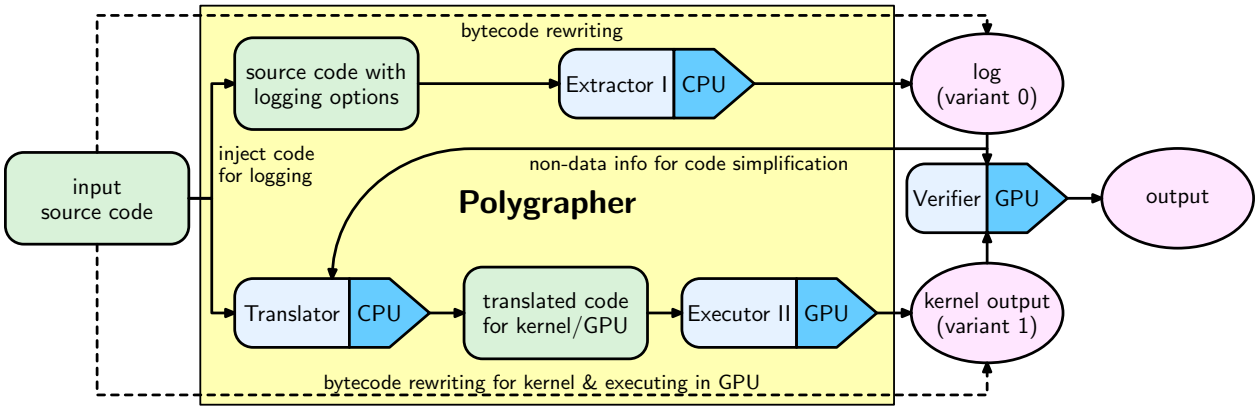


Figure 3.1: J-GANG Architecture

issues. For example, the flaw demonstrated by Listing 3.2 has been reported across several JVMs by many users, probably because it is rooted in runtime library code shared by many CPU-based JVM implementations. Leveraging a CPU×CPU hybrid model potentially offers greater diversity by extending dissimilarities down to the hardware level, yet avoiding overheads suffered by network communications between machines.

3.2.2 Model & TCB

Figure 3.1 shows the system architecture of J-GANG. The hardware differences between the two execution paths forms an ideal polygrapher, which is defined as a distributer to feed the executors with input. To generate the acceptable parallel equivalent states for CPU and GPU respectively, there are two working paths in the polygrapher. Since the input is Java source code, one path processes the original CPU execution. The other consists of an translation action and several processing behaviors of corresponding states expressed in the GPU. The two state streams are compared for semantic equality in an on-demand fashion.

The correctness of a compiler that transforms a source program (e.g., Java) into an object code program (e.g., Java bytecode or GPU bytecode) is defined in the literature in terms of *semantic transparency* (cf., (Leroy, 2009)), which asserts that the source code semantics and the compiled object code semantics yield equivalent program states. In the

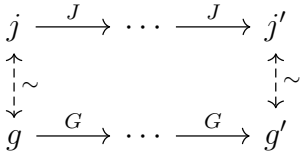


Figure 3.2: Semantic transparency

case of two compilers (source-to-JVM and source-to-GPU), we therefore transitively define relation $\sim \in J \times G$ to be the equivalence relation between the two object languages—JVM states $j \in J$ and GPU states $g \in G$—that is preserved by the two compilers' semantic transparencies. Specifically, we define $JVM \in (J, \rightarrow_J)$ to be a transition system that encodes the operational semantics of the Java bytecode virtual machine, such as ClassicJava (Flatt et al., 2002) or Featherweight Java (Igarashi et al., 2001). Similarly, define $GPU \in (G, \rightarrow_G)$ to be a transition system that encodes the operational semantics of GPU bytecode programs, such as PTX (Habermaier, 2011).

Figure 3.2 shows a commutative diagram illustrating how non-malicious executions that stay within the intended semantics of the two transition systems preserve relation $\sim$. As indicated by the diagram, this semantic equivalence is not necessarily step-wise; state equivalence is only checked periodically at checkpoints. This is important not only for performance, but also for reflecting differences in granularity between the two architectures. For example, certain computational steps by the CPU execution engine might correspond to a series of multiple computational steps on a GPU.

All non-determinism sources (e.g., random number generation, scheduling, user input, clock checks) are treated as inputs by J-Gang and logged by the CPU variant as local state. In general, this leaves three scenarios that can potentially falsify transparency:

1. one or both transition systems reach stuck states,

2. one system reaches a final state before the other, or

3. relation $\sim$ is falsified.

Condition 1 corresponds to a failure of J-GANG's implementation (the compilers, runtime systems, or validator). For example, Java language features unsupported by the prototype implementation (see §3.3) yield stuck states. Condition 2 corresponds to premature termination, as exhibited by the example in Listing 3.2. The most significant form of falsification arises from condition 3, which corresponds to developer-unintended behaviors that differ between the two transition systems. These include memory corruption, arithmetic errors, and type confusions indicative of many Java exploits.

3.2.3 GPU Feature Limitations

Current GPU instruction architectures support only a small subset of operations available on CPUs. For example, reference types (objects) and methods are not directly expressible in the kernel part of programs parsed in either the CUDA or OpenCL platforms. Likewise, GPU kernel code cannot directly access main memory during computations, since to access the main memory by shared virtual memory (SVM) is an optional feature of OpenCL and still not perfectly supported by AMD in Windows. J-GANG therefore does not rely upon it.

While these limitations may initially seem prohibitive to our goal of replicating general JVM computations to GPUs, they actually serve to enhance J-GANG's ability to detect attacks within our asynchronized validation model. Any CPU operation that cannot be supported on GPU is idealized during source-to-GPU compilation as an opaque input-output relation defined by the CPU variant's computation. For example, objects are reduced to their integer hash codes on the GPU side, and calls to their methods become checkpoints whose local states include numeric indexes of the called method and the return site. This allows the GPU variant to verify that the same object and method is called. A separate worker then validates the callee's computation and its return, avoiding an explicit method call or call stack on the GPU side. Usually these caller and callee computations are validated concurrently by the GPU.

The idealization and opacity of these operations on the GPU side is a source of many opportunities for detection of malicious computations. For example, exploits that corrupt the JVM's call stack or method tables to hijack code control-flows almost never have the same effect on J-GANG's GPU computations, which have no explicit call stack or method tables, and that exercise independent, parallel workers instead of performing serial method calls.

3.2.4 Validation Modes

J-GANG can be configured to execute in two possible modes:

Static Mode. The CPU variant can be configured to execute to completion before delivering its checkpoint log, whereupon the GPU variant validates the entire computation. This mode can be useful for terminating computations that demand high realtime efficiency, and that do not require immediate validation.

Dynamic Mode. In this mode, the CPU variant streams its checkpoint log to the GPU variant as the computation progresses. The GPU variant consumes the stream opportunistically, discarding the consumed checkpoints. This is the preferred mode, since it reduces space overheads for checkpointing, accommodates non-terminating computations, and detects intrusions live.

3.2.5 Checkpointing

Local State

Checkpoints produced by the CPU variant consist of local variable values, heap values (e.g., object hash codes and fields), and a numeric token that uniquely identifies the current code point. To control overhead, only the subset of local variable and heap values that are accessed by the CPU variant between this checkpoint and the next are included in each checkpoint. While purely static liveness analysis of Java code can be challenging (Nilsson-Nyman et al.,

2008), we avoid many of these complexities by simply logging the variable values that are actually read and written by the CPU variant as it runs, and by placing checkpoints at significant meets and joins in the control-flow graph (e.g., function and loop entry and exit points). In this way we avoid the need to accurately compute heap liveness or reachability, and all static analyses are intraprocedural. Liveness and reachability approximations are only used as optimizations to avoid unnecessary checkpoints.

If the GPU variant attempts to access a state element that was not included in its source checkpoint, or modifies a state element not included in its destination checkpoint, it signals a divergence. Thus, checkpoints and any analyses used to generate them remain untrusted by the verifier.

Frame State

The GPU variant also maintains a *frame state* comprising portions of the heap that were introduced by previous (now discarded) checkpoints, and that remain reachable, but that do not appear in the current checkpoints undergoing validation. This reduces checkpoint sizes by providing a means to validate the values of variables that are not read or modified for large portions of the program, but that remain live. It is maintained outside the GPU kernel code, and consists of an idealized JVM state representation in which objects are expressed as hash codes and their fields are expressed as hash tables.

For example, variable e in Listing 3.2 remains live throughout the loop, but is only accessed in line 7. By including e in the frame state, we can omit e from checkpoints for computation fragments that do not concern e. Checkpoints that assume $e = 2.0$ as a precondition can nevertheless be validated by consulting the frame state. Like other variables, modifications of frame elements are reported in checkpoints, and are therefore validated by the GPU, resulting in changes to its frame state.

$$
\begin{array}{rl}
s ::= v{\leftarrow}e \mid v{\leftarrow}o.m(\vec{e}) \mid s_1;\ s_2 \mid \mathsf{loop}(e)\ s & \text{(statements)} \\
\quad \mid \mathsf{branch}(e)\ s_1 \cdots s_n \mid \mathsf{try}\ s_1\ \mathsf{with}\ e \Rightarrow s_2 & \\
e & \text{(expressions)} \\
v & \text{(variables)} \\
\boxtimes & \text{(checkpoints)}
\end{array}
$$

$$
\begin{aligned}
T(v{\leftarrow}o.m(\vec{e})) &= (\vec{v}_{tmp}{\leftarrow}\vec{e};\ \boxtimes;\ v{\leftarrow}\mathsf{call}(o.m, \vec{v}_{tmp});\ \boxtimes) \\
T(s_1;\ s_2) &= T(s_1);\ T(s_2) \\
T(\mathsf{loop}(e)\ s) &= v_{tmp}{\leftarrow}e;\ \boxtimes;\ \mathsf{loop}(v_{tmp})\ (T(s);\ v_{tmp}{\leftarrow}e;\ \boxtimes) \\
T(\mathsf{branch}(e)\ s_1 \cdots s_n) &= v_{tmp}{\leftarrow}e;\ \boxtimes;\ \mathsf{branch}(v_{tmp})\ T(s_1) \cdots T(s_n) \\
T(\mathsf{try}\ s_1\ \mathsf{with}\ e \Rightarrow s_2) &= \mathsf{try}\ T(s_1)\ \mathsf{with}\ e \Rightarrow (\boxtimes;\ T(s_2))
\end{aligned}
$$

Figure 3.3: Translation of Java source to CPU×GPU code

3.2.6 Translation

Translation of Java source code to J-GANG's hybrid architecture is summarized in Figure 3.3. For simplicity of presentation, we here represent Java source code as a core language consisting of variable assignments $v{\leftarrow}e$, method calls $v{\leftarrow}o.m(\vec{e})$ (which have been factored out of expressions into separate statements), sequences, loops, n-way branches, and exception-handlers. Translation function T maps these programs to instrumented source code programs that can be compiled to native CPU/GPU architectures.

The translation process adds checkpoint operations $\boxtimes$, which have a different operational semantics depending on the target architecture. In the CPU variant, checkpoints log the local state to the verification log. In the GPU variant, checkpoints read the log to initialize the local state at the start of a worker computation, and to validate the local state at the end of each worker computation. Translation of loops, branches, and exception handlers entails adding checkpoints to meets and joins in the program's control-flow graph. To check loop and branch conditions, they are assigned to translator-introduced temporary variables v_{tmp}, which contribute to the local state and hence undergo checkpointing.

Translation of method calls invokes a call verification handler $\mathsf{call}(o.m, \vec{v})$ whose semantics likewise differ between the two architectures. On CPUs, object o's hashcode is logged to the checkpoint, method m of object o is called with arguments $\vec{v}$, and its return value is logged on return. On GPUs, where explicit calls do not exist, the logged hashcode is verified to equal the GPU state's object argument, and the return value is simply retrieved from the log file and used as the result. This works because the checkpointing placement ensures that a separate GPU worker always verifies the correctness of this return value when validating the callee's computation. (If the callee is not Java code, as in the case of JVM runtime system calls, this treatment simulates the GPU calling the external library with the same arguments and receiving the same result value.)

Each checkpoint also logs a program label that uniquely identifies the location of the checkpoint in the code. Thus, the GPU code consists entirely of a single function beginning with a branch that consults this label to conditionally jump to the code fragment being checked. Each GPU worker thereby executes a code fragment that begins at one checkpoint and ends at the next, and that consists entirely of side effect-free computational expressions suitable for GPU kernel code.

Figure 3.4 depicts the resulting execution streams for a simple loop. The CPU variant (left) executes the loop body iteratively in a serial stream, outputting one checkpoint for each iteration (and one additional one at start). A GPU variant (right) with n workers consumes all available checkpoint-pairs simultaneously, simulatig all iterations of the loop in parallel to validate the computation.

3.2.7 Verification Time Complexity

Modern GPGPU architectures are most efficient when threads execute *homogeneously*—i.e., each group of k threads executes the *same code* in lock-step (on possibly different data), and there is no significant communication between threads in the group. For example, Nvidia's

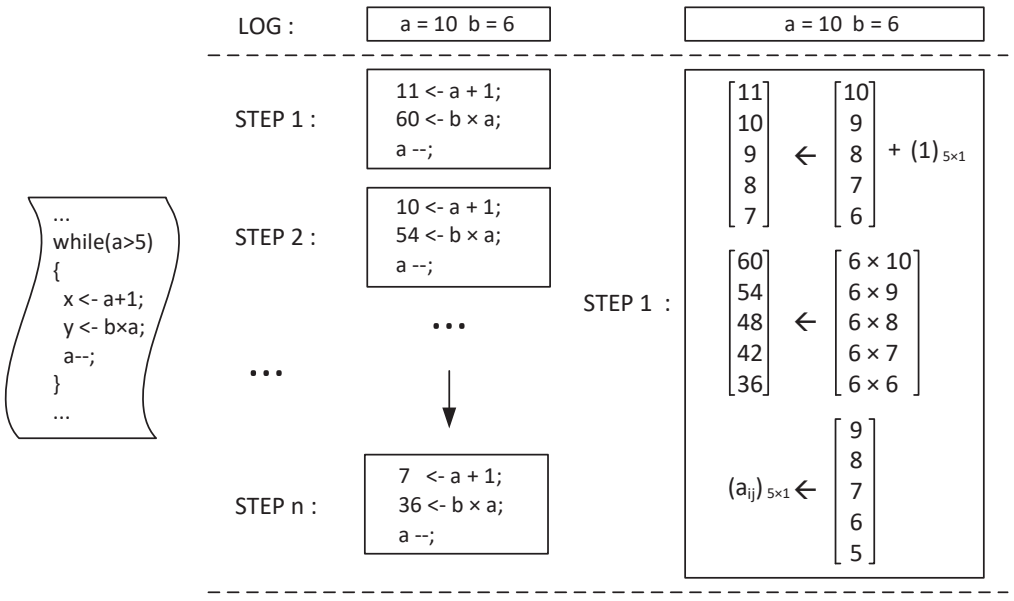


Figure 3.4: J-GANG computations for CPU (left) vs. GPU (right)

CUDA GPGPU architecture supports a Same Instruction Multiple Data (SIMD) model (as well as less efficient but more flexible MIMD models) (Maitre, 2013). On a GPGPU architecture with a single thread group of size k, J-GANG's GPU variant can obey this homogeneity constraint to achieve high efficiency, and thus keep pace with the CPU variant even after lagging behind the CPU by a factor of k, as shown by the following theorem.

Theorem. *If the time complexity of the CPU code is $O(f(n))$, then the time complexity of the GPU-translated code on an architecture with k homogenous threads is $O(f(n)/k)$.*

Proof Sketch. Code size c is constant relative to the input size n. By pigeon-hole principle, a checkpoint sequence of length $O(f(n))$ must therefore contain $\Omega(f(n)/c) = \Omega(f(n))$ checkpoint pairs that span *identical* code fragments. Translation function T (see §3.2.6) executes these homogeneous fragments in blocks of k for a total runtime of $O(f(n)/k)$. □

In practice this means that even though each GPU thread's serial computing speed is less than that of a typical CPU, with reasonably large k the GPU variant nevertheless keeps pace with the CPU. This allows J-GANG to scale to long computations.

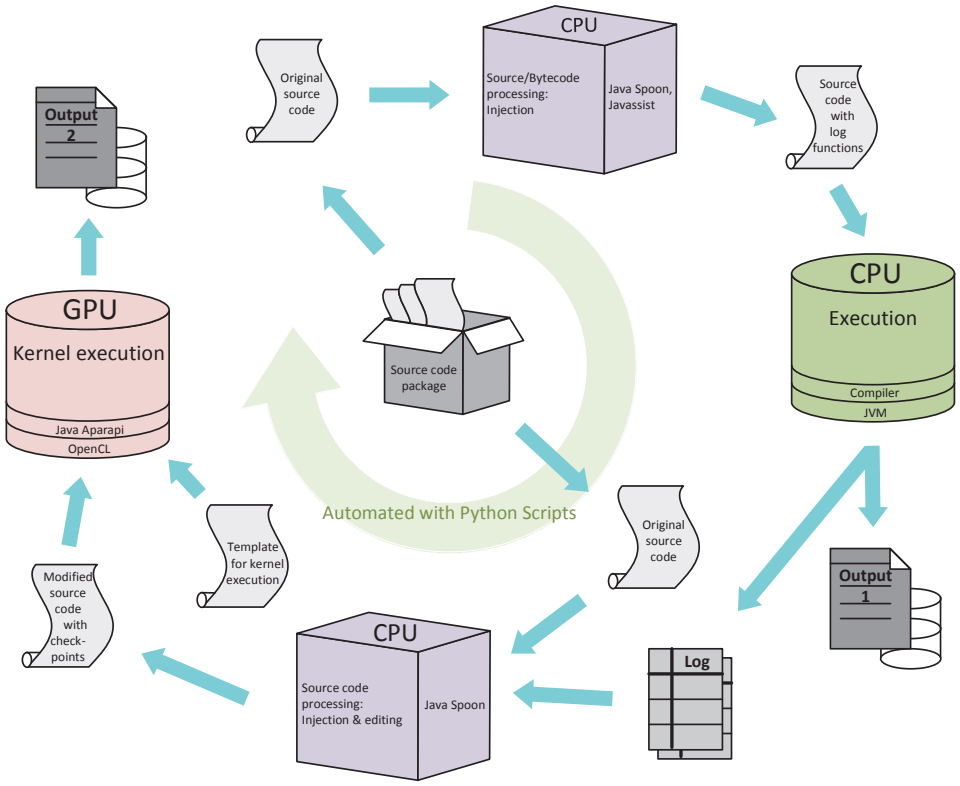


Figure 3.5: Procedure of variants generation and execution

3.3 Implementation

To test and evaluate J-GANG, we implemented an extensive translation infrastructure from Java source to GPU kernel code. This includes a new Java package handler implementation, a CPU-GPU communication library for live data logging and retrieving, translation from host code to kernel code, and procedural automation.

Figure 3.5 depicts the procedure and interaction between source code and processing units in static mode. (Dynamic mode omits python scripts and reloads modified classes with class-loaders.) It shows how the GPU Kernel code of verification for Java Aparapi is generated from source code and how we record the status of variables and create checkpoints in the kernel in basic mode.

Figure 3.6 lists four sample code fragments representing the elements of Figure 3.5. All code shown in the figure is generated automatically by J-GANG, including source code with

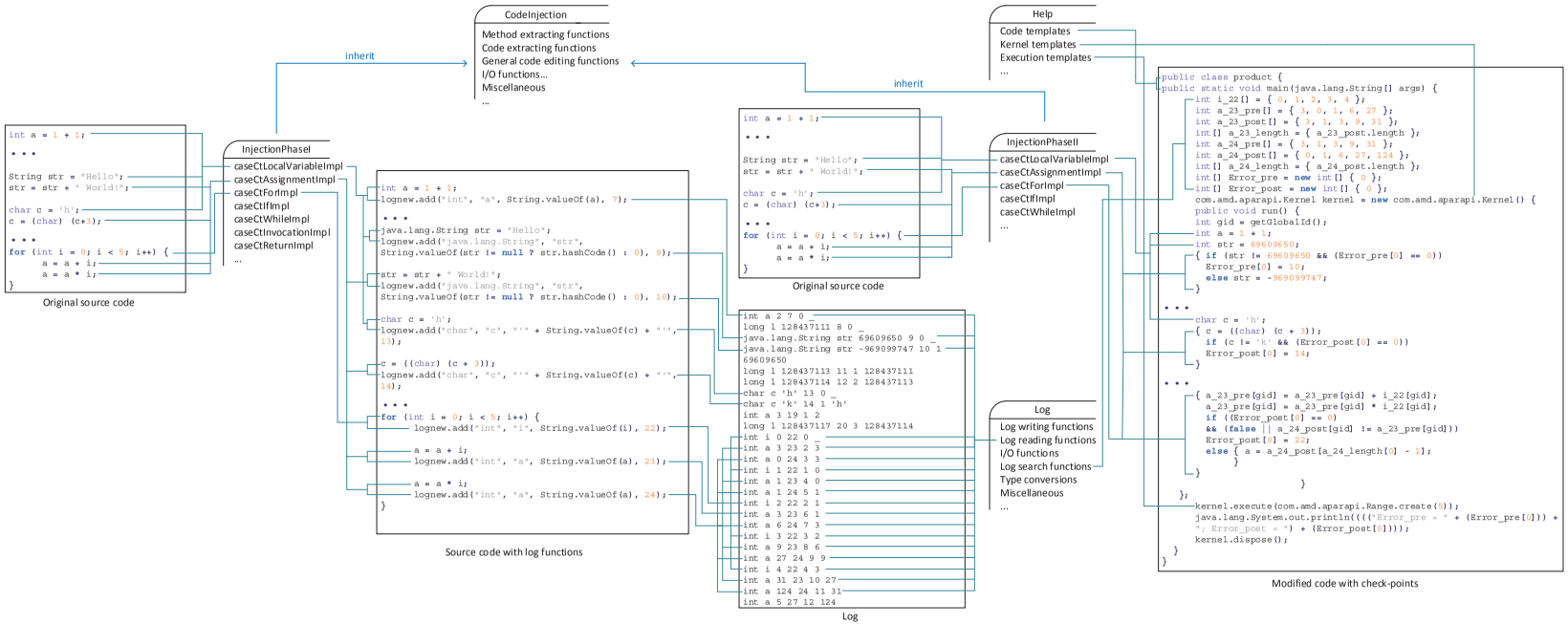


Figure 3.6: Code and checkpoints generation procedure

log functions, and code instrumented with checkpoints. Reading and writing of log tables to align the data of loops for parallel verification (see Fig. 3.4) is also automated. This is shown in Log and Modified code elements of the figure. In addition, the code with checkpoints in the rightmost side represents the initial GPU kernel code from Java Spoon (before optimizations based on dynamic analysis are applied, such as granularity adjustment).

The project is implemented in Java and Python on a machine with Intel Xeon CPU @3.4GHz, 64GB memory, and an AMD FirePro W5100 Graphic Card. Supporting infrastructure includes Java Aparapi, Java Spoon (Pawlak et al., 2015), and Javassist (Chiba, 2000). Aparapi by AMD provides Java bindings to enable the host to call APIs from OpenCL; it is the portal for Java code execution on GPU, and masks OpenCL's more detailed operations and memory management in the devices. For example, it automatically switches from Java's multi-threading and GPU task scheduling, making its style of kernel code more accordant with Java code. Java Spoon by Inria is used as a language-parsing and code-injecting tool to log variable information and transform the input source. Javassist offers the ability to rewrite code from input at the bytecode level.

3.3.1 Source Language Limitations

Since our prototype is implemented atop Java Spoon and Java Aparapi, it is presently limited to Java code that can be parsed by those tools. Aparapi offers a Java-style grammar wrapper on the kernel code of OpenCL, which is based on the C99 standard and does not support Java-level multi-threading or certain higher-order OOP constructs (e.g., first-class lambdas). For exact limitations, please see the documentation of the aforementioned tools. Our prototype follows the basic Java SE standard, and therefore does not yet support language features new to subsequent Java versions. In addition, some language optimization will also be limited due to the current GPU and CPU's architecture of communication. For example, the optimization mentioned in Figure 3.4 for nested loops or recursive function will be impossible.

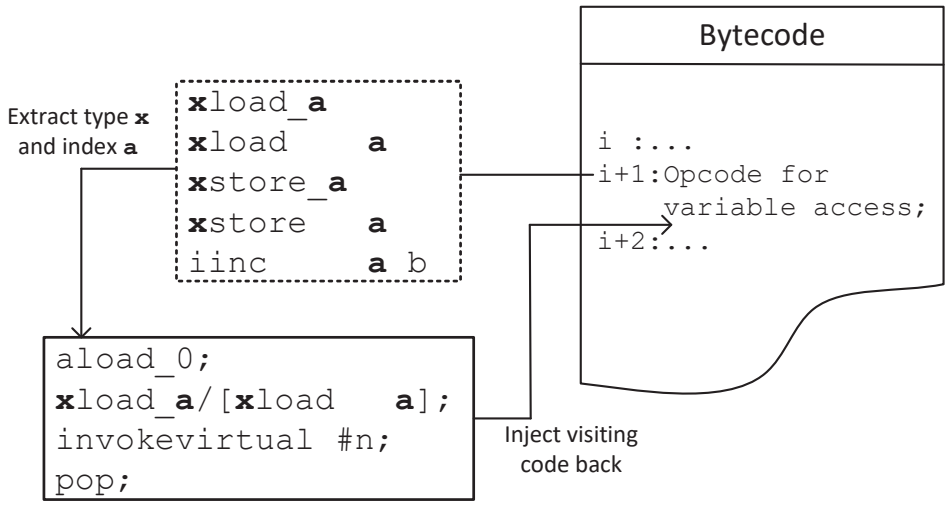


Figure 3.7: Bytecode injection to dynamically visit a variable

3.3.2 Bytecode Analysis

J-GANG uses bytecode analysis to log executions and dynamically rewrite GPU kernels (the dashed lines in Figure 3.1). In dynamic mode, the variables in the system are visited while executing the original source code. We wrote a toolkit package on Javassist for this task, since no convenient tool in the market currently provides functions for the Java language to visit local variables of methods in JVM at runtime. Java language extentions, such as AspectJ, offer indirect ways to achieve this, such as refactoring source code to expose local variables at compile time. Javassist and ASM offer manipulation in bytecode.

Figure 3.7 illustrates our procedure for logging local variable state. To locate the local variables in a Java bytecode method, the indices of local variables are first tracked by inspecting the opcode `iinc` and opcode family of `xload(_a)` and `xstore(_a)` of the method. From this we create the bytecode of the log statement with acquired line numbers and variables' indices of these opcodes, and in-line it into the method. Executing the instrumented method streams the log of variables to the verifier.

To dynamically rewrite the GPU kernel code, the source code is first translated to executable statements for the kernel and converted to its bytecode. This creates the code

block that will be executed on the GPU. To make it executable, we compile an empty kernel template to bytecode and inject the bytecode of the code block. The newly generated kernel must be compiled and dynamically loaded before the compiling procedure for the whole source code starts. This is because the template kernel has already registered in the JVM before the generation of the new kernel. This one-time reloading initializes a nonstop procedure from input source code directly to execution in the GPU, achieving live, streaming computation validation.

3.3.3 Primitives & References

In the static mode, Java Spoon is used to parse the input source code. All the statements about initialization and assignment of variables are first located with their line numbers. In this stage, the update sites of variables are analyzed for liveness, and duplicately-named variables are assigned unique indexes in the log.

Java's primitive types are all recorded directly into logs, since the OpenCL kernel uses the same data types during verification. For example, values of type `char` are logged as `unsigned short`. To log reference types, a method of lightweight recording is chosen: The system tracks references' hashcodes to monitor their changes, since the GPU kernel lacks first-class references. All non-primitive objects or attributes can be disassembled or converted into primitives (Goetz, 2014), affording verification of all references by the GPU.

3.3.4 State Consistency

Before verification starts, an initial memory state must be prepared so that both variants can begin computation in equivalent states. This *pre-state* corresponds to the precondition of a Hoare Triple. To keep the state size tractable, it is desirable to restrict each pre-state to only those variables that are referenced by the computation fragment being verified. To compose the pre-state, the code block to be verified and its line numbers are analyzed with

Java Spoon so that relevant variables can be selected out. Each selected variable's pre-state value is determined by identifying its last update in the checkpoint log, or its value in the frame state if the checkpoint log contains no updates.

To take advantage of parallel computing, J-GANG represents pre-state variables in different ways depending on the control-flow structures that contextualize each computational fragment being verified. Sequential and conditional control-flows offer only small opportunities for parallelism, so their variable values are stored separately in local memory. However, variables in (non-nested) loops are arranged into arrays and loaded into global memory for parallel verification. Our prototype does not yet perform this optimization for inner loops of nested loops, since doing so introduces complexities related to dynamically generating kernel code that anticipates how the various nesting levels interleave at runtime. This is an optimization we intend to pursue in future work.

3.3.5 GPU-based Verification

Executor II in Figure 3.1 is implemented as GPU kernel code analogous to the code that executes on the CPU. It first loads the initial computation state (pre-state) reported by the CPU version, and then compares the generated result state with the logged post-state. To leverage the performance strengths of GPGPU computing, J-GANG implements GPU code that enjoys two forms of parallelism: (1) Loops are parallelized into concurrent verification of their iterations, as shown in Figure 3.4. (2) Comparison of the post-state derived by the GPU to the one reported by the CPU is parallelized into concurrent comparisons of state partitions.

In the first part, when the system verifies the iterations of a loop in parallel, the values in each iteration for a variable in both the pre-state and post-state are aligned into an array, which affords efficient concurrent access by GPU kernel code. This asymptotically decreases the verification time by one layer of the loop, as proved in Section 3.2.7. With k parallel

processing units in the GPU, the number of iterations is reduced from n to n/k, where n is the number of CPU loop iterations.

In the second part, we efficiently compare the two post-states from the executors by converting them into two byte arrays. Bitwise XOR computation can be performed on the arrays to efficiently compare them for equality. On the GPU, this XOR computation is parallelized among all available workers. This method significantly accelerates state comparisons, which are otherwise slow on serial architectures since the states can be large.

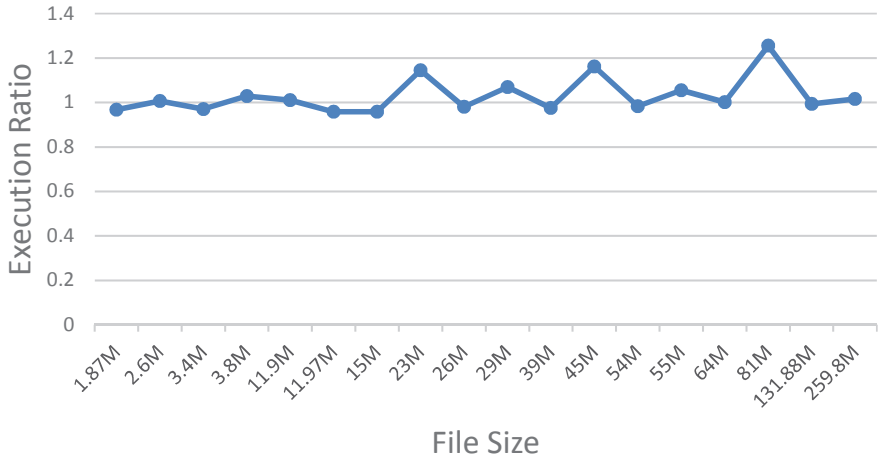
3.3.6 Code Pruning

When some part of the source code is never executed by Executor I (CPU) or has no effect upon the computation state (e.g., non-executed branches or effect-free code), this part can be trimmed from original code before the translation for Executor II (GPU). For example, it is not necessary to keep the discordant branches in the second execution since these are unreachable. This optimization is safe because divergent computations that include such code blocks are guaranteed to still exhibit divergence when omitting the effect-free blocks.

To implement this optimization, line numbers of variables are inspected in the log to determine the direction of flow before the translation. We record line numbers of variables with their updated values together during the checkpointing step. All statements, including those in branches, are tagged by the line numbers. During the first execution, the values of variables in statements visited by program flow are logged. These recorded lines indicate which branch updated the variables. By checking the number, we deduce which branches can be omitted from verification. If there is no variable recorded in the branch, the segment of code can be trimmed since it is effect-free.

Table 3.1: Performance evaluation of Java algorithms using J-GANG. Partial granularity omits verifying trusted API methods.

Program	Original (ms)	Granularity	Static (ms)	Dynamic (ms)	Delay (ns/B)	Log (KB)	Time
binary & sequential search ($n = 100000$)	200.101	partial	211.172	247.248	289.245	163	$O(n \log n)$
matrix multiplication ($n = 100$)	16.741	all	603.304	4098.165	2293.258	17799	$O(n^3)$
2-color algorithm for bipartite ($V = 500, E = random(\frac{V^2}{4})$)	2.561	all	68.917	2835.420	1098.829	25709	$O(V + E)$
mode of a set ($n = 1000$)	2.136	all	91.002	1318.665	251.448	52246	$O(n^2)$
all subsets in lexicographic order ($n = 15$)	4.155	all	249.721	4284.198	859.592	49656	$O(2^n \log n)$
nearest neighbor by linear search ($n = 1000, D = 2$)	0.078	all	0.463	29.476	1223.429	240	$O(nD)$

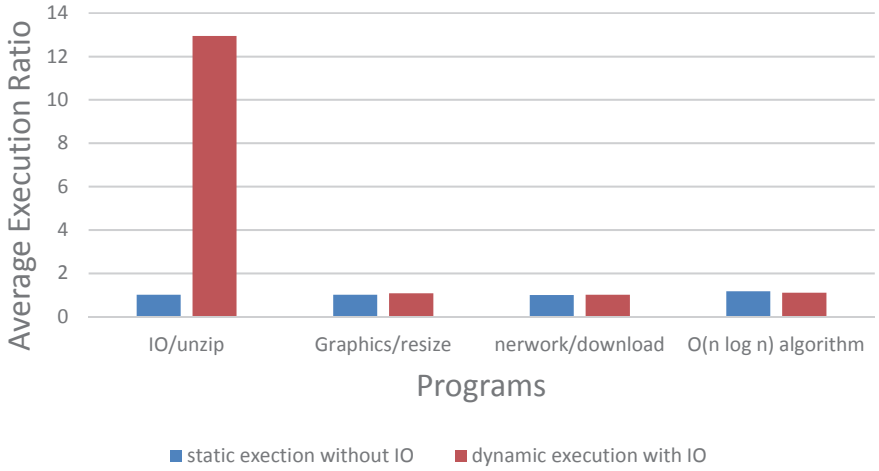


(a) Java IO/unzip

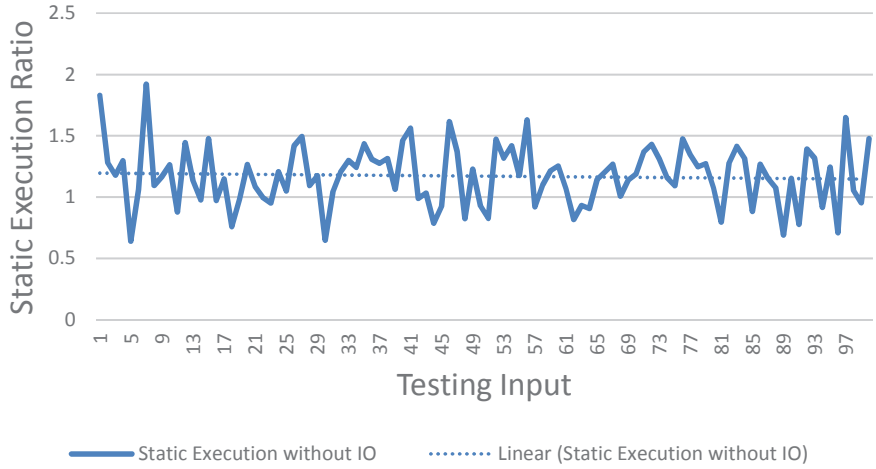
(b) Java networking/download

Figure 3.8: Experimental results with utility applications as input

(c) Java graphics/resize JPGs



(d) Execution Comparison



(e) Control overhead for $O(n \log n)$ binary search

Figure 3.8: Experimental results with utility applications as input (cont.)

3.4 Evaluation

Our experimental evaluation of J-GANG is grouped into vulnerability detection accuracy and runtime performance. The evaluation architecture is the same as the development framework reported in Section 3.3, and publicly available, independently authored Java input programs are selected from diverse sources for correctness and performance tests. Programs with different time complexity and utility are chosen to test running performance. Also, a group of relatively new Java bugs are selected from Oracle Java Bug Database to test the accuracy and utility of the framework for real-world scenarios. All bugs are treated as zero-days—no vulnerability-specific mitigations or controls are deployed for any of the experiments.

In the experiments, checkpointing is closely related to overhead. To control this trade-off, the granularity of checking can be flexibly tuned from the finest-grained level (checking every variable update immediately) to coarser-grained levels (checking variable updates after code blocks or function returns). For example, checkpoints can be inserted after each line of code within a loop, or only before and after the loop for greater efficiency. Coarser checking requires fewer checkpoints but potentially larger states to check at each checkpoint.

3.4.1 Running Efficiency

Performance evaluation of J-GANG can be characterized in terms of two metrics: (1) overall runtime overhead of the instrumented CPU computation, and (2) the delay between time-of-exploit and exploit-detection by the GPU verifier. Overheads measured by the first metric are primarily due to the extra time needed to log checkpoints for verification. Checkpointing is partly asynchronous, but there is still overhead incurred by initializing and spawning the asynchronous I/O. Overheads measured by the second metric are primarily driven by the size of the checkpoint stream, and are therefore measured in reciprocal-bandwidth (ns/B).

Our evaluations consider two categories of test application: classic algorithms (which afford investigation of time/space complexity effects, memory update frequency, and highly

optimized code loops), and practical utilities (which examine applicability of J-GANG to real-world software products). The latter include website-downloaders, compression tools, and image editors. They are randomly chosen for the testing, and demonstrate our approach's generality and versatility. Selected programs are all from independent authors and were tested for correctness before evaluation. Each test data point reported is an average over hundreds of trials.

Performance is reported for both the static and the dynamic verification mode. In static mode, the CPU computation runs at full speed and produces a complete log of checkpoints, which is verified by the GPU after the computation completes. In dynamic mode, the checkpoint log is consumed opportunistically by the GPU verifier as the CPU computation progresses, affording live, parallel validation of the computation. The static mode therefore incurs lower I/O overheads, but has the disadvantage of building a larger checkpoint log and offering only retroactive detection of exploits. The dynamic mode incurs higher I/O costs but does not need to retain the full checkpoint log in memory or on disk, and detects exploits on a short delay.

Table 3.1 and Figure 3.8e report runtime overheads for the first category of tests (classic algorithms). Figures 3.8(a–d) report overheads for the second category (practical applcations). The results indicate that tight, numerically intensive computations incur high overheads (due to the high cost of frequent checkpointing relative to streamlined mathematical computations), but most of the practical applications perform well under J-GANG. For example, utility programs running in static mode show overhead ratios of less than $\leq 1.008\%$, and average overheads of less than 7% in dynamic mode. All overheads are under 10% except for the outlier in Figure 3.8d for unzip files, which is investigated in more detail below.

The experiments reported here do not include any manual granularity tuning; we allowed J-GANG to select checkpoint locations, frame state update frequency, and loop verification parallelizations purely automatically. To better support the short, computationally intensive

algorithms in Table 3.1, we conjecture that a less frequent, time interval-based checkpointing regimen would perform better for such algorithms. Figure 3.8e investigates this conjecture by adjusting the checkpointing granularity for the binary search experiment, resulting in a more acceptable overhead of about 20%. Tuning the granularity in this way does not sacrifice assurance, since it preserves computational divergences somewhere within the checkpoint stream. It merely offers less parallelism opportunities to the verifier by clustering more verification data into fewer checkpoints. A more detailed investigation of the performance trade-offs of this tuning approach is reserved for future work.

Our experimental data also indicates that input file sizes affect the stability of performance measurements. When file sizes are too big or too small, the performance timers yield unstable outputs. This effect can be seen in Figures 3.8a and 3.8b, where the values on the right and left, respectively, fluctuate more widely than in the center. Values at the centers of the plots should therefore be considered more reliable and indicative of real-world observations.

Figure 3.8d contains an outlier for unzip files, which we investigated in detail to ascertain the cause. It occurs because the test program allocates a large buffer as one of its local variables and access it in an innermost loop, causing J-Gang to include it in many of its checkpoints. The performance could be greatly improved by introducing heuristics whereby J-Gang removes unmodified portions of arrays to its frame state (see §3.2.5) rather than including them in every checkpoint just becomes some elements were modified. This is another optimization that should be considered by future work.

There is a significant time difference between the dynamic and static modes, especially on experiments with mathematical algorithms. After investigating this, we determined that the higher runtimes reported for dynamic mode are almost entirely due to log access I/O costs that could be significantly improved in a production version of the system. In particular, our prototype stores logs by piping the output of I/O into *System.out* when in-lining the

methods in bytecode. All the dynamic test results are therefore bounded with the delay caused by the console output. In the related testing in the industry, this delay has been shown to be much more expensive than other forms of I/O (e.g., 100× higher than directing I/O to files).[45] Although we cannot isolate this portion of the overhead precisely, we can estimate it by artificially inflating the sizes of the log files and observing the relation between runtime overhead and log file size. The results of this analysis indicate that more than 90% of the dynamic mode overhead is due to console I/O. In the independent unit testing on I/O, the average result of delay on I/O for stream to console is around 454ns, which is within the range of the results for the Delay column of Table 3.1. An obvious next step to improving our prototype implementation is therefore to replace console logging with a high performance filesystem or other storage medium.

In addition, I/O overheads can be further minimized by performing I/O more asynchronously. Doing so avoids delaying the main computation at the cost of slightly increasing the delay between the full-speed CPU computation and the GPU verifier's detection of faults and intrusions. The delay due to I/O latency is only around 0.001 second/KB on average for our prototype.

The influence of memory overhead is minor relative to the I/O overhead, and its scale is determined by the size of input of a program. To avoid a predictable overhead to an uncertain input, it is best to adjust the logging granularity or optimize the tracking by in-lining some trusted methods.

3.4.2 Verification and Correctness

To verify the correctness, we tested whether our system can detect vulnerabilities of the JVM exploited by flawed or malicious input programs. For accuracy, we only chose the bugs

[4] https://stackoverflow.com/questions/4437715

[5] https://stackoverflow.com/questions/18584809

Table 3.2: Tested bugs

No.	Bug ID	Description
1	JDK-5091921	Sign flip issues in loop optimizer
2	JDK-8029302	Performance regression in Math.pow intrinsic
3	JDK-8063086	Math.pow yields different results upon repeated calls
4	JDK-8166742	SIGFPE in C2 Loop IV elimination
5	JDK-8184271	Time related C1 intrinsics produce inconsistent results when floating around
6	JDK-7063674	Wrong results from basic comparisons after calls to Long.bitCount(long)
7	JDK-8046516	Segmentation fault in JVM
8	JDK-8066103	Compiler C2's range check smearing allows out of bound array accesses

verified by Oracle Java Bug Database. Reproducing the vulnerabilities in Table 3.2 requires different versions of Jave SE. No simulated program is used in the testing for correctness. The 8 selected bugs are non-duplicated and 7 of them are not related except the second and third.

The vulnerabilities we tested span all officially released subversion of Java SE 6–8. Java SE 9 and 10 are not included because Java 9 non-critical bugs will not be fixed and added in the subversions, and Java 10 was released concurrently with our research. All JVM bugs and test code for them were drawn from Oracle's official bug database. There are usually several bugs (sometimes none) in each subversion that are related to Java official compiler Hotspot based on Windows x86/x64. Among them, we selected bugs that are testable and offer related source code. While our approach is applicable to vulnerabilities reported elsewhere, such as in malware threat reports, JVM bugs that have not yet been documented in Oracle's official database are extremely difficult to reproduce reliably, and are therefore not tested in this work.

Generally, the eight vulnerabilities listed in Table 3.2 arise from inaccurate calculations of CPUs in comparison with GPUs. Half of them cause CPUs to perform incorrect floating point

computations. Inaccuracies of this form undermine numerous secure computations, such as encryption, related to floating point. Other bugs in the list invite software compromises. For example, testers reported that the false access to arrays caused by JDK-8066103 can be abused to corrupt the heap in ways that victims are unlikely to notice for significant lengths of time. The sign flip problem JDK-5091921 is related to about 30 bug reports and is major facilitator of denial-of-service attacks against Java-based servers.

J-GANG detects all the exploits in Table 3.2 as a divergence of the CPU and GPU computations. Our testing methodology for confirming this is detailed below.

In each exploit of the 8 vulnerabilities, we first reproduce the exploit to confirm that we have vulnerable execution environment with a proper version of the Java SE and running flags. We then run the code on J-GANG and perform GPU-based validation of the computation. In some cases, we needed to make minor manual adjustments to the proof-of-exploit code to get it to execute, or to keep it compatible with our evaluation infrastructure. None of these manual adjustments affect the exploit itself, or introduce any vulnerability- or exploit-specific mitigations. Manual adjustments needed include the following:

- Some code with new or lesser used Java language features cannot be processed by some of the tool packages underlying our prototype implementation. Such code was adjusted to exclude the unsupported features when the features are not part of the exploit.

- Some exploits become inadvertently corrected merely by the introduction of J-GANG's logging code. For example, the logging code may deactivate a buggy JVM loop optimization. In a real deployment, this is an advantage to defenders since the instrumented code is no longer exploitable. However, to force the exploit to work and test its effect, we manually omitted or moved any checkpoint sites that had the side-effect of fixing the exploit being tested.

- Certain atypical forms of variable assignment, such as reflective updates, are not yet supported by our prototype. We converted such operations to supported equivalents when doing so did not affect the exploit being tested.

- Some proof-of-concept exploit code causes the JVM to freeze instead of hijacking or crashing the victim application. This is typically an artifact of the proof-of-exploit implementation (since real attacks tend to abuse the vulnerability to greater effect). Freezes yield no more checkpoints, so are detectable by timeout rather than by computation divergence. To change freezes into divergences, we artificially force a final checkpoint for such computations.

3.5 Summary

This work proposed and implemented J-GANG, an n-variant system framework for verification of Java code by which vulnerabilities can be detected and exposed as the divergence of the execution between CPU and GPU computations. Our solution translates general source code and introduces it into kernels, which yields a solution for executing general Java code in GPUs. To overcome performance disadvantages related to executing mostly serial computations on GPUs, J-GANG leverages GPU parallelism to validate many CPU loop iterations concurrently, affording the GPU variant a means to keep pace with the CPU variant even on computations that are not automatically parallelizable outside of an n-variant setting.

We evaluate our system based on the source code of utility applications, known public vulnerabilities, and classic algorithms in Java. A clear security benefit of our work is to detect possible unknown vulnerabilities, including zero-day attacks, while vulnerable programs are executing. Intrusions are detected by the GPU variant on a small delay, whereupon the defense can potentially intervene by raising an alert, aborting the computation, and/or rolling the system back to a safe state.

Prototype implementation of the approach demonstrates significant promise, but exhibits some high overheads for certain operations, such as intensive mathematical computations and high-volume I/O. These observations motivate future work on optimizations that better parallelize nested loops and replace synchronous I/O with asynchronous I/O to improve runtimes.

CHAPTER 4

VISUALVITAL[1]

4.1 Overview

In this chapter, an observation model about computer vision is proposed and studied. This research is a sub-topic of joint work with Dr. Kang Zhang's team. The motivation is to efficiently observe with limited number of cameras (or positions) to a given area in map. First, the problem is formally defined and it is transformed into mathematical situations for optimally dividing weighted poly-lines into segments and selecting positions for cameras in order to maximize the overall weights of the observed objects. After the formulation of the model and problem, a series of algorithms with a total time complexity of $O(n \log n)$ are invented and proved to compute the optimal result as the solution, which is to first maximize the local weights with oversized number of positions and then shrink the number by merging the positions with receiving lower weights until the number descends to the limit. A implementation on Google Maps demonstrates this efficient method.

4.2 Introduction

> *"Dallas, Nov. 22—President John Fitzgerald Kennedy was shot and killed by an Assassin today. ... The killer fired the rifle from a building just off the motorcade route."* (Wicker, 1963)

The tragic events of November 22, 1963 are a grim reminder of the importance of concise yet comprehensive visual surveillance of security-critical events and venues. Despite the presence of many well-trained human eye-witnesses (e.g., police officers and Secret Service

[1]The material in this chapter was originally published as: Jun Duan, Kang Zhang, and Kevin W. Hamlen, "VisualVital: An Observation Model for Multiple Sections of Scenes," In *Proceedings of the 14th IEEE International Conference on Ubiquitous Intelligence and Computing (UIC)*, August 2017.

members), as well as television coverage, analysts continue to debate to this day exactly who fired the fatal shot and from what position. Due to humans' limited range of attention and visual scope, it is imperative to grasp parts and details that humans easily neglect.

This is especially important when graphic data processing is involved. Humans more efficiently process visual than textual information in many contexts (Cybulski et al., 2014; Heukelman and Obono, 2009; Pandey et al., 2014), motivating the use of visualization for data processing (Heer and Shneiderman, 2012). With advancement of sensory technology (e.g., consumer VR headsets), there is an elevated need to efficiently extract key points and timings for delivery to human users, since our brains have limited memory volume for long steaming visions or videos. By selecting these crucial scenes and positions (Lam et al., 2017; Nievas et al., 2011) and aggregating the total values of the importance of those pieces, people can control or understand the whole contents with less effort.

To this end, our research considers the specific problem of how to position and orient a limited number of video monitoring devices so as to maximize the information gain on the status of a route in a map. We anticipate that both cameras and objects may be dynamic (e.g., possibly moving observers of moving objects, such as traffic, pedestrians, or weather). Relevant applications include auto-generation of adaptable scenes for Oculus® VR with Google Maps®, which entails finding key positions and making a smooth line between them to shorten browsing time and maximize visual information along the path, or performing high-quality traffic monitoring with fewer cameras without coverage loss. In addition, this problem can be simplified into a 1-dimensional version, which is to search those key-frames inside a streaming file and find its related information instantly, such as for music and video fingerprinting (Lee and Yoo, 2008; Milano, 2012), copyright protecting (Hampapur et al., 2002; Yuan et al., 2004), and specific scene detection (Datta et al., 2002; Jansohn et al., 2009).

The contribution of the work can be summarized as follows:

- By applying human visual characteristics, we create an observing model to simulate the sight that a camera has.
- We implement a virtual camera's functionality in Google Maps®. By tagging each object with a weight (or detail), we can calculate the observed weight under different angles and distances.
- For a limited object in 2D, we find the maximal observed weight exists, and an observation position can be calculated from its minimal circumcircle.
- Our method finds optimal camera positions in $O(n \log n)$ time, and keeps the overall observed details above the baseline.

The remainder of this chapter is arranged as follows. Section 4.3 describes the three camera observing models and formulates the problem. Section 4.4 shows the details of our methods and mechanism of the two-phase algorithm. Experiments are implemented in Section 4.5 to show the correctness of our proposed solutions. Finally, Section 4.6 summarizes the chapter.

4.3 Notions & Models

This section presents three models that we combine to formulate the definition of the problem. The first model introduces the type of observed objects. The second one defines rules for observing and calculating *details* (or *weight*). The third one models how the position and angle of the camera affects the details, which extends the rules of the second model.

4.3.1 Observation Model

Figure 4.1(a) shows the basic notions used in this model, assuming a camera c is deployed at a position p, and has a fixed viewing scope ϕ, observing a segment ℓ. A sector is formed by

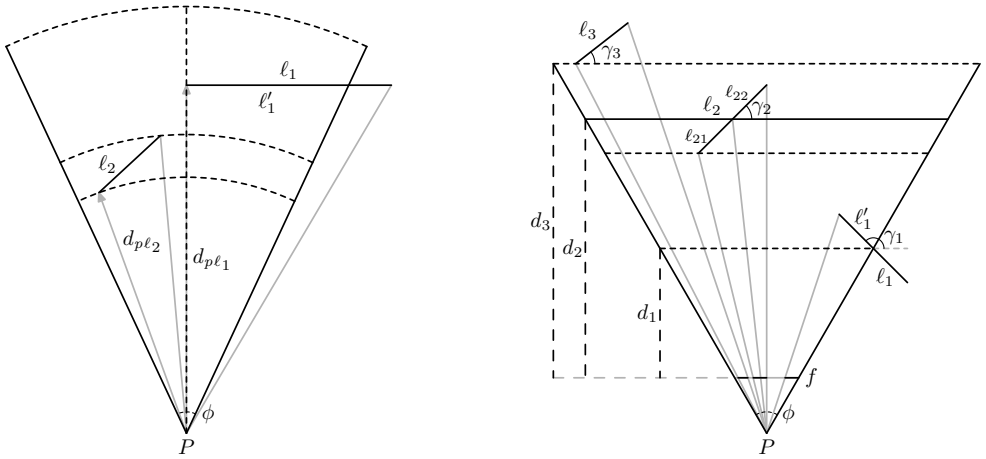


Figure 4.1: Complete presentation for an observation and related notions (a) Basic observation (b) Simplified observation with projections

the angle ϕ as c's vision range, whose depth extends from p to ∞. If this entire object ℓ falls in c's vision range, we say ℓ is *fully-observed*. Similarly, whenever part of ℓ stays in c's vision range, we say that ℓ is *partially-observed* by c. We denote the distance between position p and the nearest point of segment ℓ as the *viewing distance* $d_{p\ell}$ of ℓ, and the covered (or c-observed) part of ℓ as ℓ'.

In this model, the detail of the segment ℓ observed by camera c is defined as ℓ's weight within a ratio of $(0, 1]$, denoted as w_ℓ. The observed weight of ℓ through c is defined as w. For a single segment, w is a function of w_ℓ, $d_{p\ell}$, and ℓ'.

4.3.2 Weight Model

When the distance $d_{p\ell}$ between the camera c and the object grows, the amount of observed detail reduces; inversely, when the distance $d_{p\ell}$ is less than a specific value d_0, we cannot see more details than the original w_ℓ from a camera. So, d_0 is the *critical distance* for c's observation of objects. Thus, when ℓ is fully observed, we define the relation between the observed details w, d_0, and $d_{p\ell}$ as follows: (1) When $0 < d_{p\ell} \leq d_0$, we have $w = w_\ell$. (2) When $d_{p\ell} > d_0$, we have $w = \frac{d_0}{d_{p\ell}} w_\ell$.

The critical distance d_0 is defined before an arc $\bar{c}$ can be drawn along all the critical positions of the camera c. Then, a sector β consists of $\bar{c}$ and the open angle of camera c at p. Inside the area β, we assume no detail is lost in the observation. In the following, the fan-shaped lossless field is simplified into a triangle to be introduced in the next subsection.

Therefore, combining with the observation model, the observed weight (or details) w can be defined as follows: Since d_0 is a parameter of the camera, we can introduce a constant ratio α to formulate a relation $d_0 = \alpha w_\ell$ to simplify the formula

$$w = \begin{cases} \dfrac{\ell'}{\ell} w_\ell & \text{if } 0 < d_{p\ell} \leq d_0 \\ \dfrac{\ell'}{\ell} \cdot \dfrac{d_0}{d_{p\ell}} w_\ell = \dfrac{\ell' \alpha}{\ell d_{p\ell}} w_\ell^2 & \text{if } d_{p\ell} > d_0 \end{cases}$$

4.3.3 Camera Projection Model

We see different lengths of a line when viewing the line from its side and from one of its ends. Obviously, if a line object could project its entire side on the camera screen (without losing any details), this object should be completely inside the open fan-shaped vision sector β and perpendicular to the viewing direction. Here, the angle between the line object and the screen is denoted by γ. When γ grows to its maximum $\frac{\pi}{2}$, the line object is vertical to the camera screen and projects a dot on it, in which case we consider no detail can be observed. Based on the two critical conditions, our angle-continuous projection is modeled as follows.

Given a camera c with viewing scope ϕ at p, a line object ℓ with weight w_ℓ, and viewing distance $d_{p\ell}$ of ℓ, the weight w of ℓ's image can be approximated by $w = w_\ell |\cos\gamma|$. This is due to two considerations: (1) Computation can be drastically reduced from calculating values on an arc to calculating its corresponding line segment. (2) The difference between the length of an arc and its correspondent straight line in the observation becomes insignificant when the viewing distance is large. Also, $\Delta\ell$ is an increment when $d_{p\ell}$ increases from d_{min},

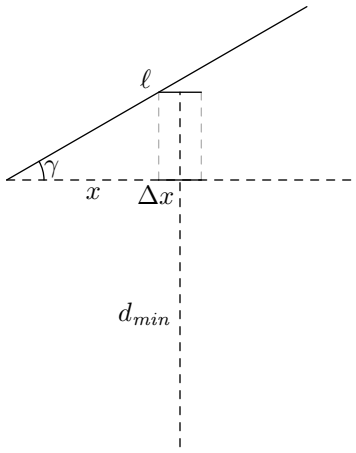


Figure 4.2: Accumulation for the calculus step

shown in Figure 4.2. So a piecewise function with integral is formulated. Taking the weight model into consideration, the length on the camera is

$$w = \begin{cases} 0 & \text{if } \gamma = \frac{1}{2}\pi \\ \dfrac{\ell'}{\ell} w_\ell \left|\cos\gamma\right| & \text{if } d_{p\ell} \in (0, d_0] \\ & \text{and } \gamma \in [0, \frac{1}{2}\pi) \cup (\frac{1}{2}\pi, \pi] \\ \displaystyle\int_0^{\ell'|\cos\gamma|} \frac{w_\ell d_0}{\ell(d_{p\ell} + x\left|\tan\gamma\right|)}\,\mathrm{d}x & \text{if } d_{p\ell} > d_0 \text{ and } \gamma \neq \frac{1}{2}\pi \end{cases}$$

Substituting $d_0 = \alpha w_\ell$ and evaluating the integral, we obtain

$$w = \begin{cases} 0 & \text{if } \gamma = \frac{1}{2}\pi \\ \dfrac{\ell'}{\ell} w_\ell \left|\cos\gamma\right| & \text{if } d_{p\ell} \in (0, d_0] \text{ and} \\ & \quad \gamma \in [0, \frac{1}{2}\pi) \cup (\frac{1}{2}\pi, \pi] \\ \dfrac{\alpha w_\ell^2}{\ell\left|\tan\gamma\right|} \ln\left(1 + \dfrac{\ell'\sin\gamma}{d_{p\ell}}\right) & \text{if } d_{p\ell} > d_0 \text{ and} \\ & \quad \gamma \in (0, \frac{1}{2}\pi) \cup (\frac{1}{2}\pi, \pi) \\ \dfrac{\ell'\alpha}{\ell d_{p\ell}} w_\ell^2 & \text{if } d_{p\ell} > d_0 \text{ and } \gamma \in \{0, \pi\} \end{cases}$$

In typical application scenarios, the camera usually observes segments from a long distance. Thus, the meaning of $d_{p\ell}$ is also changed, which indicates the distance between the

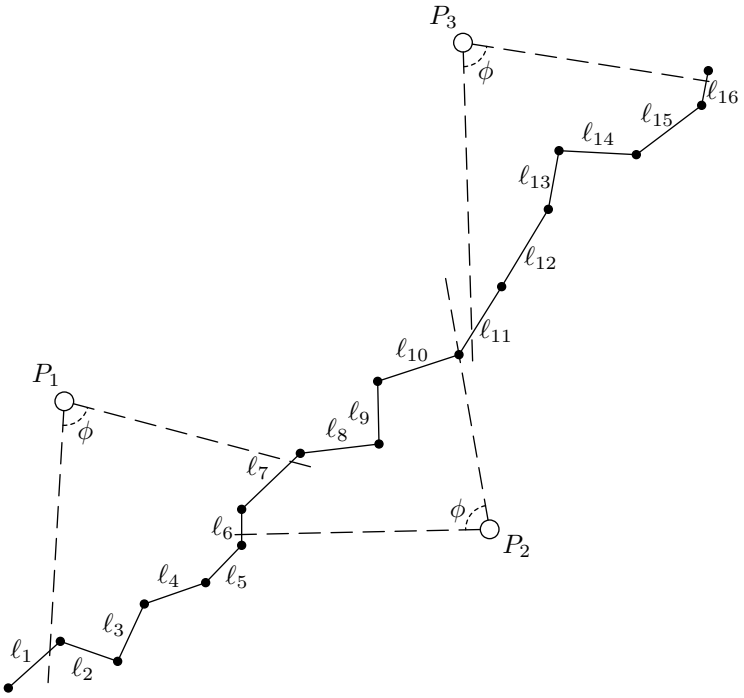


Figure 4.3: Camera c observes poly-line L at different positions P_1, P_2, P_3

nearest point in object ℓ' and position p. This significantly simplifies the computation for the weight-lossless area. The similar relation $d_0 = \alpha w_\ell$ can still be easily proved and holds after the transformation.

4.3.4 Problem Formulation

Given a series of continuous segments, a poly-line $L = \{\ell_0, \ell_1, \ell_2, \ldots, \ell_n\}$, with each segment ℓ_i assigned a weight w_{ℓ_i} (to denote the quantity of details of this segment), and a camera c whose viewing angle is ϕ, the total amount of observed detail is $w = \sum_{i=0}^{n} w_i$.

Having built the model, we formulate the problem as finding a set of positions that satisfies the following three requirements:

- fully covering each segment of a poly-line object,
- having a bounded number of spots (positions), and

- maximizing weight under the spot limits.

Given a set of positions, we will investigate whether there exists a shortest path for the camera to pass all the positions in the future.

4.4 Algorithm Design

For the two problems above, we need to build the position set first and then the orbits can be searched in the set. To find the set, we perform two major steps. First, we compute a *lossless weight set* (LWS) that deploys the fewest number of points to cover the target line without losing any details. That is, the weight to be observed at these positions should be equal to the original weight of the object. Second, we compute a *points constricted set* (PCS) by revising and deleting positions to retain the maximized total observed weight while minimizing the number of positions. The weight obtained from the positions in PCS is apparently smaller than or equal to that from those in LWS, and reaches its maximum under the number limit (threshold).

4.4.1 Lossless Weight Set

To find the LWS for a poly-line, we use an efficient method to sequentially record positions. The way of collecting the positions for LWS starts along a poly-line that is parallel to the object. Each time, we choose a position where the vision lossless range of the camera can cover the most weight on the line ℓ_i with the best viewing angle ($\gamma = 0$). This procedure of selecting positions continues until the length of ℓ_i is equal to 0. Algorithm 1 sketches the procedure for computing the LWS for a single side of the poly-line.

[1]Positions can also be chosen from both sides, but doing so requires the camera to cross the object under observation, which is undesirable and should be avoided. If all the selected positions are kept on the same side, the orbit never intersects the polyline.

Algorithm 1: Algorithm for building LWS on one side[1]

Input : $L = \{\ell_1, \ell_2, \ldots, \ell_n\}$; $\{w_{\ell_1}, w_{\ell_2}, \ldots, w_{\ell_n}\}$; d_0; ϕ
Output: *LWS*

```
LWS = ∅;
Choose an endpoint e (either from ℓ_1 or ℓ_n) along L;
for i = 1 to n do
    Construct a poly-line ℓ_ip perpendicular to ℓ_i at e;
    Place c at a position p on ℓ_ip at distance d_0 from e (where d_0 is c's critical
     distance). Position p could be at either side of ℓ_i. Without loss of generality, we
     assume it is on one specific side for all;
    Adjust the bisector b of c's viewing scope so b is perpendicular to poly-line ℓ_i and
     viewing scope open to ℓ_i;
    while len(ℓ_i) > 0 do
        Begin with one end of ℓ_ip and move c along the poly-line ℓ_1p. Record the
         position p, direction of camera's viewing scope b, and observed weight w
         when the total weight inside the viewing scope reaches its max;
        Delete the part of ℓ_1 whose weight has been recorded in the last step;
        LWS ← (p, w);
    end
end
```

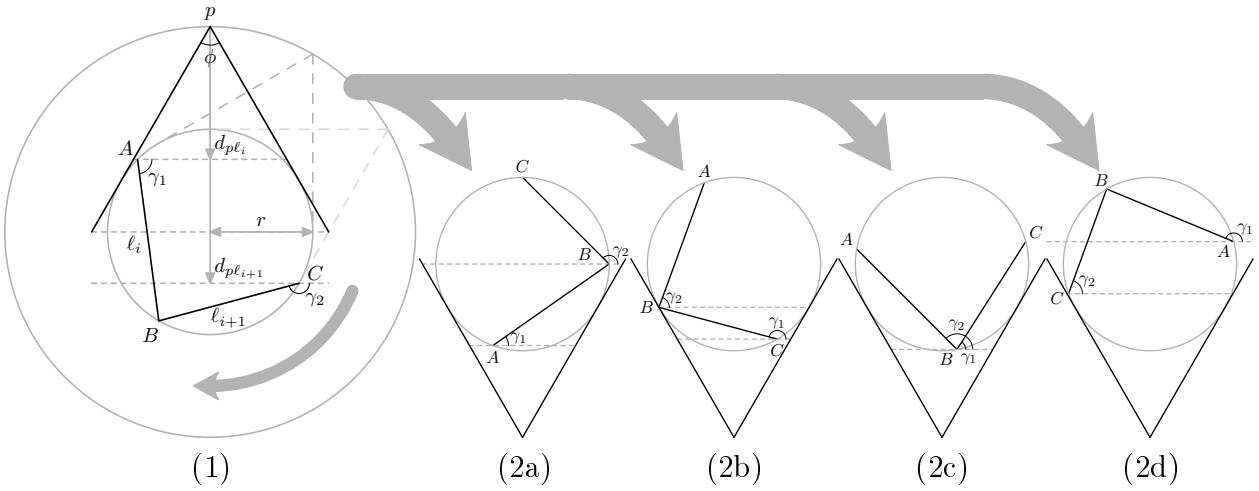


Figure 4.4: (1) All possible situations of camera view, and (2) four different angle relations between γ_1 and γ_2

Algorithm 1 states that the camera at each position p in LWS always observes the maximal weight when a line object is larger than the range c with viewing angle ϕ. Thus, all

the positions are consecutively selected into LWS along the moving trace of the camera. If a line or the remaining part of a line is smaller than the range c, its position is revised and combined to decrease the number of positions in the following algorithms.

If it still does not meet the limit, some positions are combined and adjusted to cover the part covered by the camera at more than one previous position. Since the LWS elements are chosen sequentially along poly-line $L_p = \ell_{1p}, \ell_{2p}, \ldots, \ell_{np}$, better options are always to replace the two nearby elements in the order of LWS by one best candidate to reduce the set size. In particular, two non-consecutive positions cannot be selected as a replaced pair because the new cover range overlaps the part between the two positions.

Consequently, this motivates a procedure to find which pair of consecutive positions should be replaced by a new position with the combined coverage of both former positions and the least loss of observed weight in the current LWS. Our approach attempts all pairs of neighboring positions in LWS and identifies candidate positions.

4.4.2 Points Constricted Set

The first step is to find positions and directions of the cameras to cover the range at 2 consecutive positions. Our model shows that the total observed weight is related to the angle formed by the two neighboring line segments.

We begin with the easiest case—the three endpoints are collinear. The camera simply moves away in the same direction until covering exactly the 2 segments. If the three endpoints are not collinear, however, a series of positions can be found based on the circumcircle of the three endpoints.

Theorem 1. *In 2D, given a single camera c with AOV (angle of view) $= \phi$ and limited objects (or poly-line) ℓ arranged not in a line segment, a circle $\bigcirc p$ with center p and radius r as the circumcircle of ℓ, the maximal observed weight of c exists and can be found when c is at one specific position on/in the circle with center p and radius $R = \frac{r}{\sin\frac{\phi}{2}}$.*

Proof. See Appendix. □

Assume that $\ell_i = AB$ and $\ell_{i+1} = BC$ are covered by a camera c at positions p' and p'' of LWS. If the area of the circumcircle of A, B, and C can be covered by c with AOV ϕ at a single position p_i, then ℓ_i and ℓ_{i+1} are included, too. From the formula of our camera projection model, it is known that w is a monotonic decreasing function of $d_{p\ell}$. So the nearest position to observe this circumcircle with any directions of c is the farthest vertex of the isosceles triangle formed by angle ϕ and the line through radius r, in which a half of the circumcircle is inscribed. The candidate positions also compose a circle $\bigcirc p$ and are shown in Figure 4.4. Observe that points inside $\bigcirc p$ may not fully cover ℓ_i and ℓ_{i+1}, and points outside $\bigcirc p$ must get smaller weight than points on $\bigcirc p$ due to the inverse relation between w and $d_{p\ell}$.

The second step is to select the best position on circle $\bigcirc p$ from candidate positions. This situation can occur when one object is blocked by another at some positions on the circle. Therefore, these positions are excluded from comparison. In Figure 4.4, conditions (2a) and (2b) are excluded. For the left two conditions, we choose the best position based on the formula from our model

$$w = \frac{\alpha w_\ell^2}{\ell \left|\tan\gamma\right|} \ln\left(1 + \frac{\ell' \sin\gamma}{d_{p\ell}}\right) \qquad \text{when } d_{p\ell} > d_0 \text{ and } \gamma \in (0, \tfrac{1}{2}\pi) \cup (\tfrac{1}{2}\pi, \pi)$$

In this function, w is related to $d_{p\ell}$ and γ. This is because the two left situations, (2c) and (2d), restrict $d_{p\ell}$ to a range, and can only be changed less than the radius of the circumcircle. Consequently, finding the right γ is the key in this search. Furthermore, we know the value of the angle formed by the two line objects, and can easily derive the relation between γ_1 and γ_2, which are the angles formed by the objects and the camera screen, respectively. In addition, the total weight is the sum of these two observed weights. As a result, the following steps of calculation find the critical point of γ_1 (or γ_2) which maximizes w within its interval.

Algorithm 2: Algorithm for Finding the Candidate Position

Input : $\ell_i = AB, \ell_{i+1} = BC$; $p_j, p_{j+1} \in LWS$; d_0; ϕ

Output: p_{max}, w_{max}

```
Calculate the angle α = ∠ABC ;
if α = 0 or α = π then
    Locate the position P_max to exactly cover ℓ_i and ℓ_{i+1} and use the same viewing
     direction as c did at p_j (or p_{j+1});
    Calculate w_max with c at p_max;
else
    Draw the circumcircle ○P of A, B, and C to obtain its center P and radius
     r_P = PA (Pedoe, 1957);
    Draw a circle ○P' with P as its center and r = r_P / sin(φ/2) as its radius;
    Get the equation of w_t = w_i + w_{i+1} for ℓ_i and ℓ_{i+1} based on the formula of w and
     γ, by using γ_{i+1} = f(γ_i, α) to replace γ_i;
    Insert the (2c) and (2d) conditions in Figure 4.4 to calculate maximal values w_{t1}
     and w_{t2};
    Compare the two values and choose the position p' of the larger w_t;
    Relocate towards P to exactly cover ℓ_i and ℓ_{i+1} and use the same view direction
     as c did at p';
    Calculate w_max and p_max with the current d_{pℓ};
end
Return w_max and p_max ;
```

After computing the best location on the circle, the final task is to adjust the distance between the camera and the two line objects. Usually the camera remains stationary or is relocated a bit towards the objects. When entering the circle, the camera at some spots can get more weight without losing any coverage for the objects; on other spots, it cannot keep the objects fully covered if it moves inside.

We design the following two algorithms based on the analysis above. Algorithm 2 calculates the best position between any two consecutive positions of LWS, and Algorithm 3 deletes and replaces the positions for the camera to get the total maximal weight subject to the number restriction. In the algorithms, p_{max} denotes the position where c can fully cover ℓ_i, ℓ_{i+1} and maximize the observed weight; w_{max} denotes the observed maximal weight.

Algorithm 3: Algorithm for Creating the PCS

Input : L; d_0; ϕ; $LWS = \{p_1, p_2, \ldots, p_{|LWS|}, \}$; *limit*
Output: *PCS*

```
1  PCS = LWS;
2  Initialize an ordered set TMP = {t_1, t_2, ..., t_{|LWS|-1}} and and t_i = {p, w};
3  if |LWS| ≤ limit then
4  |  Return PCS;
5  else
6  |  for i = 1 to |LWS| − 1 do
7  |  |  t_i = AFCP(p_i, p_{i+1});
8  |  end
9  |  while |LWS| > limit do
10 |  |  Find t_i satisfying p_i.w + next(p_i).w − t_i.w =
   |  |    min{p_1.w + next(p_1).w − t_1.w,  p_2.w + next(p_2).w − t_2.w,  ...};
11 |  |  PCS = PCS − next(p_i) ;
12 |  |  p_i = t_i;
13 |  |  if prev(t_i) exists then
14 |  |  |  prev(t_i) = AFCP(prev(p_i), p_i);
15 |  |  end
16 |  |  if next(t_i) exists then
17 |  |  |  next(t_i) = AFCP(p_i, next(p_i));
18 |  |  end
19 |  |  TMP = TMP − t_i;
20 |  end
21 |  Return PCS;
22 end
```

Assume that we select all the positions on one side of poly-line L; the side should be designated in advance. Also, the camera is always positioned over the critical distance from the poly-line, ensuring a near-maximal total weight observed without losing details in each calculation.

In the last step, p_{max} is inclined to be chosen in the later side of the record. That is because if the candidate positions are continuous, the poly-line is more likly to be divided into small segments at the beginning. Since the positions are chosen greedily, the poly-line can be broken into two discontinuous parts. Thus, we have another algorithm based on

Algorithm 2 to justify the best positions in those sub-polylines. Again, we always stay on one side.

4.5 Simulation

To verify the correctness and evaluate the performance of our algorithms, we conducted experiments on geographic data obtained from Google Maps®. Streets of 4 different cities around world are chosen for the simulation. Some cities and districts have neat planning such that all of their streets follow a specific pattern. For example, Ginza in Tokyo follows a rectangular pattern. In other cities, such as the districts in Cairo, Egypt, most of the streets evolved historically without any regular pattern. So, the reason we choose different cities across the world is that these specific city planings of density would affect how much weight to be arranged on the different scenes of these cities. Our simulation results show that all the calculated camera spots can cover all the streets when the number of spots is decreasing. Also, it retains over 90% of observed details when removing 20% camera spots.

4.5.1 Experiment Setup

The experiment settings receive two inputs: a camera with fixed parameters and selected routes (i.e. streets in the cities). It outputs the observed details with the boundaries of all camera spots. A real-world 24mm wide-angle camera is simulated, with viewing scope $\phi = 84°$, viewing distance $dis \in (0, \infty)$, and maximal lossless distance $d_0 = \sqrt{2}$. All the routes are listed in Table 4.1, which are approximated as poly-lines, and each line segment is set with weight (or detail) according to its traffic volume (or importance). For example, a path from the Great Pyramid at Giza to the Egyptian Museum is weighted and transformed to a poly-line based on its relative coordinates. The original path in Google Maps®, an open simple curve, is to be extracted and input into Microsoft Visio® so that a poly-line approximating the curve can be created and coordinates of the joints and endpoints will be

Table 4.1: Routes Information

No.	City	Segments	Baseline Spots	Total Weight
1	Dallas	19	155	148
2	Paris	18	216	146
3	Tokyo	25	153	362
4	Cairo	42	226	426

collected. Adding a weight to each segment of this poly-line, a weighted poly-line is ready as input.

For each poly-line, a set of baseline spots are first computed by finding the minimum sum of camera spots to fully cover the line without losing any detail. We then collect the total weights with different boundaries of different spots. The performance of the algorithm is evaluated by examining how slowly (or quickly) the observed details are lost when reducing the number of spots.

4.5.2 Performance

The performance of our experiment can be evaluated by comparing the baseline and results from our algorithm, as shown in Figure 4.5. The first step is to draw a line with the maximal weights without covering the entire poly-line within the limits of camera spots. In the second step, we compute the sum of the weights when decreasing the number of spots. Normally, we can implement a random algorithm to get the data as the result of the control(unoptimized) group. For this experiment, the results from random-deployed cameras along the polyline will be much worse than the results from the baseline. Because the observed weight from each camera is not only affected by the position but also the direction of its viewing scope. Either the observing distance may be too great or the camera may not face towards the object. Obviously, the results from an unthoughtful situation will not exceed the outcome from a simply optimized one—the baseline above, which is a situation that camera are ideally

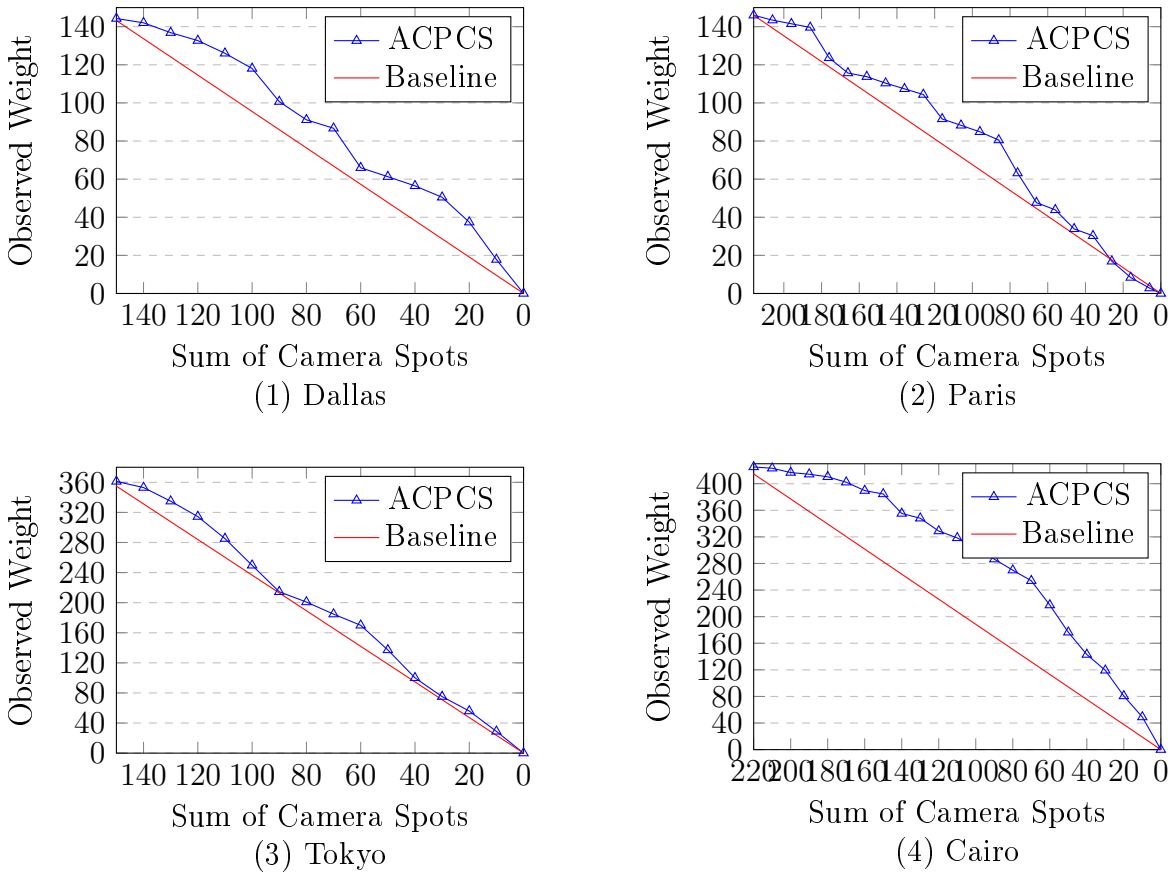


Figure 4.5: Detail-preservation when decreasing the number of spots on sampled path data from four different cities (Dallas, Paris, Tokyo, Cairo)

deployed at the critical distance away from the segments, towards the segment, without considering whether the object is fully covered or not.

Figure 4.5 shows that at the beginning of decreasing spots (among the first 20% of span), our method can keep more than 90% of the original weight. For all the four cities, the results show that our algorithms maintain more weights than the baseline set 98% of the time in the collected samples. The only scenario in which the total weight is lower than the baseline weight happens when the limit of spots is under 8%. When the limit is below 8%, one must set the camera spots far away from the observed lines in order to cover the full range. Moreover, our experiment shows that with an increasing number of segments, our method generates better results, as compared for (2) and (4) in Figure 4.5.

Our experiment reveals that with a series of weighted objects, the segments of the highest weights are selected first. The system always attempts to merge the objects of lower weights, and is inclined to cover those of higher weights with better angles when weights must be reduced. These features are exactly our objectives when designing the algorithms.

4.5.3 Analysis

Since the geographic information and details are crucial in the project, we now explain how our method can compute the best current positions to keep the parts with crucial details (higher weight) being preferentially observed. There are two ways in the algorithms to downgrade those insignificant parts when a decision on decreasing the count of camera positions must be made.

By applying Algorithm 3, all the segments are sorted by weights; then the two adjacent ones with the least total weight are picked. For the pair, a new best camera position is recalculated by Algorithm 2 to substitute the former two positions and still fully cover the two segments with minimal sacrifice to observable details. The procedure decreases only the relatively less important segments—the pair with the least total weight. So, the parts with key visual details tend to remain untouched when the threshold is small. Our experiment shows that the processing of the shadow list entails a sorting procedure. We choose merge sort in the simulation because it has the best average time efficiency, which also crucially determines the performance of our method. The related explanation can be found in Section 4.5.4.

When Algorithm 2 recalculates a new position for a pair of adjacent segments, our method attempts to preserve the part with more details. The preference of position selection in the algorithm can be illustrated with its reactions under two typical conditions. If the two segments have different lengths and weights, our camera leans toward the segment with higher weight to reduce the weight loss due to the angle. In other words, the segment with

higher weight gets better angles than the one with lower weight. The other condition is that the two chosen segments have the same length and weight. Our experimental results show that a camera position having the same angle with the two segments is generated. In other words, both segments lose the same amount of weight to keep the total weight maximal. From these facts, we conclude that the search for local weight maximum is not based on any sorting algorithm, but rather based on the Extreme Value Theorem. This also means that the time efficiency of Algorithm 2 is $O(1)$, because the extremum can be found directly via the first derivative of our piecewise function by the Extreme Value Theorem.

4.5.4 Time Complexity

By analyzing the pseudo-code of our algorithm, we derive the average run time to be $O(n \log n)$. The first phase takes $O(n)$ time to segment a poly-line. In the second phase, a shadow list recording the difference of elements is generated and merge-sorted. Then, a while-loop inserts elements into the list and decreases the count toward the limit, obtained from input as parameter a. So, Phase 2 is $O(n) + O(n \log n) + O(an + c) = O(n \log n)$. Since a is an input constant less than n, the overall time complexity of our algorithm is $O(n \log n)$. The efficiency has also been demonstrated experimentally.

There are many choices for the sorting procedure. Merge sort is selected because it has a stable performance since its worst time complexity equals its average time complexity. Furthermore, the performance of our method relates critically to the choice of the sorting algorithm, since other parts of our method have a time complexity of $O(n)$. The best or worst run time efficiency obviously also depends on which sorting algorithm is chosen.

4.6 Summary

This work has explored the problem of how to determine a minimal number of spots for a camera to cover an entire range of scenes in a geographical region with minimal loss of

details. We have proposed a visual model, which consists of projection, measurement of weights, and observation range, given a set of geometrical and geographical constraints.

Having defined the problem, we propose a 2-phase algorithm to gradually remove and/or merge camera positions and eventually meet the requirements. Most importantly, the algorithm finds the maximal observed weight for a single spot. For situations of multiple spots, the solution is at least NP-hard; thus, we propose a heuristic yet efficient approach and an algorithm with the time complexity of $O(n \log n)$.

We have also experimentally verified our method with several realistic geographical scenarios. Our results greatly outperform the baseline where the weights are evenly distributed.

As the future work, more experiments will be conducted on the scenes with single dimensions or objects with multiples features. We will apply our approach to geographical visualization systems and evaluate its effectiveness in such applications.

CHAPTER 5

RELATED WORK

5.1 Execution Variance

When n-version programming was first introduced in 1978, it opened the field to further improvements in the reliability of software execution, including advantages for fault-avoidance and fault-tolerance (Avižienis, 1985; Chen and Avižienis, 1978). Subsequent experiments identified independence and diversity of software variants as a critical challenge for the approach (Knight and Leveson, 1986). In particular, software ecosystems created by independent humans from a common specification exhibit surprisingly low diversity, since humans are prone to making similar mistakes.

In addition, progress in n-version programming was severely hindered by the cost of its implementation. Multiple teams of developers were required to build their own version of each piece of software, which was then collected into a single system moderated via a voting strategy to produce results and maintain consistency. This highly manual approach potentially multiplied software development and maintenance costs by a factor of n, deterring many practical deployments.

These obstacles motivated automated diversity as a potential amelioration of these dilemmas (Cohen, 1993). Instead of requiring multiple teams, execution diversity can be created by automatically generating variants. Proposed sources of diversity include transformation of nonfunctional code, changing memory layout, and code reordering (Forrest et al., 1997). For example, prior efforts have maintained and monitored software properties throughout its maintenance lifecycle to help detect when core behaviors could potentially change (Yang and Evans, 2004), or have leveraged address space randomization (Shacham et al., 2004) to probabilistically defend against memory errors (Bhatkar et al., 2005).

The introduction of automation also raised the opportunity to apply n-variant programming to address another major rising software problem: cybersecurity. For example, diverse

replication was applied to frustrate attempts to hijack operating systems (Cox et al., 2006). Within the past decade, this strategy has seen significant progress as software-producing tools, such as compilers, have reached a level of maturity suitable for large-scale, automated n-variant deployment (cf., (Larsen et al., 2014)). Recent works have inferred semantics from source code to locate semantic bugs based on multiple different implementations (Min et al., 2015), and to build multi-variant execution environments with multi-threading to detect memory corruption vulnerabilities in C/C++ programs (Volckaert et al., 2017).

5.2 Heterogeneous Computing

Modern computer programs take advantage of both CPU and GPU components when needed. A survey (Mittal and Vetter, 2015) published in 2015 gives a thorough introduction on this topic. It mentions that one of the motivations for heterogeneous computing is leveraging the unique architectural strength of each processing unit, which corresponds to our idea of utilizing the ability of a GPU to process loops in the program flow. It also introduces hybrid applications and programming languages that span CPUs and GPUs, such as Map-Reduce framework (Chen et al., 2012; Dean and Ghemawat, 2008; Shirahata et al., 2010; Tsoi and Luk, 2010) and programming frameworks that eliminate the boundary between CPU and GPU (Hong et al., 2010; Jiang and Agrawal, 2012; Pai et al., 2010; Veldema et al., 2011). Map-reduce-like frameworks (Hong et al., 2010; Jiang and Agrawal, 2012) describe methods for executing source code on both CPUs and GPUs without any modification.

Another way to bridge the differences among hardware is to adopt an intermediate representation. Through this, a program can be automatically dispatched into suitable processing units (Pai et al., 2010). For-loop optimizations partition loop iterations across multiple concurrent workers to form a *parallel-for loop* in Java (Veldema et al., 2011). In our work, the purpose of optimization on the loop is only for verification; so we can evaluate iterations of for-loops in parallel regardless of whether they are computationally parallelizable. This is

due to the fact that the CPU replica reveals the (untrusted) input and output states of each iteration in advance.

There are ways to seamlessly develop on GPUs with Java (Pratt-Szeliga et al., 2012). For example, prior work has applied this technique to translate Java bytecode to OpenCL and implement efficient sample pixel rendering (Aciu and Ciocarlie, 2016). There are also ways to compile languages into a hybrid environment (Garg and Amaral, 2010). Our work does not utilize these approaches since many modifications, including simplification for GPU and optimization, would be required to realize them for the general-purpose computations that we envision as potential subjects of validation.

5.3 Verification

In our work, we consider shrinking the possible state space in the redundant execution since the processing ability of a single processing unit in GPUs is a subset of the CPU computation. Some states must be simplified or canceled, and the verification in our work is used to describe the assurance of execution results. Through the implementation, we still found some methods to guarantee the quality and scalability for formal verification (D'Silva et al., 2008).

To avoid the problem of state space explosion in the procedure of precise verification, multiple strategies can be adopted. One approach is to compress the information of states and still offer explicit checking (Holzmann, 1997). Partial order reduction can be used to prune the possible increased space of states (Godefroid, 1996).

Verification of Java computations is a subject of many prior works (cf., (Stärk et al., 2012)). Java Pathfinder (Visser et al., 2004) implements model-checking based on an intermediate language (Havelund and Pressburger, 2000) to analyze Java bytecode. Primitive types and references are bound to the JVM instructions and incorporated into searches. Type-based abstract interpretation can validate JVM executions (Leroy, 2003). Horn solvers

have also been developed for Java verification based on logic programming (Kahsai et al., 2016). Ahead-of-time compilation is another proposed approach (Baxter, 2017). Machine-checked proofs have been constructed to obtain highest possible assurance for Java computations (Hubert and Marché, 2005), although these approaches currently require significant manual effort.

5.4 Dataflow Analysis

To track inner local variables of methods, our work leverages static dataflow analysis. Such analysis is a staple of program analysis surveyed by numerous prior studies (e.g., (Su et al., 2017)). Related works have studied the collection of profiling information in statements via dataflow tracking (Agrawal, 1999; Ball and Larus, 1996; Rapps and Weyuker, 1985), detection of confidentiality leaks in Java (Mongiovì et al., 2015), and troubleshooting software errors caused by misconfiguration by tracking dataflows embodying interprocess communications (Attariyan and Flinn, 2010).

5.5 Correctness in GPGPU

Since the advent of general purpose GPU programming, an increasing number of programs are taking advantage of hardware acceleration to improve run-time efficiency. Examples of GPU computing that provide substantial commercial value and cost savings relative to CPU-only computing include integer programming (Soner and Özturan, 2012), application acceleration (Messmer et al., 2008), motion tracking (Huang et al., 2008), and data processing on large data sets (Bakkum and Skadron, 2010).

Reasoning about concurrency is well recognized as being especially challenging for human programmers, making GPGPU programs exceptionally difficult to get right. Since the GPUs in the modern computers are not merely focusing on the graph-related accelerating and manufacturers opened these resources of parallel computing to all the programmers,

more and more codes based on this high-performance structure have been keeping created. When these programs are used to generate formal business projects like one related to integer programing(Soner and Özturan, 2012), application accelerating(Messmer et al., 2008), motion tracking(Huang et al., 2008), GPU computing is showing its huge commercial value and saving a mount of time comparing with the old-school way(only CPU-computing related) to deal with the drastically increased data(Bakkum and Skadron, 2010). But every coin has two sides. By introducing general-purpose computing on GPU(GPGPU) into those commercial projects, It does not only bring economic benefits, but also injects platform-embedded code bugs and security risks into this new territory, like data race, state discordance. Obviously, these problems rarely even unlikely happen in the code processing of single-core or multi-core CPUs and methods of discovering problems in CPU-related applications will hardly work in this field. Thus, ways of verifying and rules of rewriting GPU-related programs were invented to eliminate these possible problems.

Common GPGPU programming errors include inadvertent reliance upon platform-dependent hardware features or idiosyncrasies, as well as data races and barrier divergence (Kirner et al., 2010; Li et al., 2014).

Initially, detecting multi-threaded errors in GPU programs was challenging as limited crossover from multi-threaded CPU verification techniques existed due to differing hardware architectures. Significant characteristics that distinguish GPUs from CPUs include massive parallelism, limited sequential consistency enforcement, and potentially higher competition for shared resources.

Since then, the field has expanded to include methods of verifying and implementing rules for rewriting GPU programs to eliminate many bugs (Collingbourne et al., 2013; Diamos et al., 2010). Ocelot (Diamos et al., 2010), a well-known emulator for GPU code, translates PTX kernels to equivalent CPU code (x86 multi-core, IBM CELL Processor), which can then be used for verification purposes. Panoptes (Kennelly, 2012) is an on-the-fly binary

translation framework for CUDA, which avoids emulation performance overhead while still performing code analysis and validity checks.

5.6 Data Race & Divergence

Our static, formal validation approach complements these related works by drawing upon well-established methodologies for formalizing parallel computing architectures as derivation rules of an operational semantics.

These include the seminal work of Kahn (Kahn, 1974), who defined semantics for parallel programming and surveyed the principles of parallel computing. More recently, the advent of *skeleton-based parallel programming* (Aldinucci and Danelutto, 2007) has simplified many of the traditional idioms by incorporating unique behavior to decompose cumbersome operations into primitive operations. This is useful when analyzing complex parallel code in order to reduce the occurrence of logic mistakes.

One prominent logic error is the barrier divergence problem, which occurs when barrier code is in a loop or block and is not executed by all threads in the system. This causes threads to hang while others (possibly erroneously) continue. Prior work has developed semantic encodings that formally define this class of bugs, and that are therefore potentially useful in formal analyses (Bardsley et al., 2014; Betts et al., 2012). Some methods attempt to reduce the barrier divergence problem through optimization techniques that reduce the number of branches in a program. With less branching, it is more likely for threads to remain synchronized and executing the same code. Other useful optimization techniques include iteration delaying (Han and Abdelrahman, 2011) and branch fusion (Coutinho et al., 2011).

Data races are another traditionally hard-to-detect flaw in GPU code. These typically occur when multiple threads have write capability to the same memory location. Reliably detecting such bugs requires testing sets expansive enough to cover the universe of all value

sets that could possibly be written to the shared memory location under any run of the program. This can quickly exceed the search space that unit testers can feasibly explore. Recent work has therefore combined static analysis of CUDA code with dynamic checking to use a two-step method to detect and tag possible data races (Zheng et al., 2014, 2011). While this reduces the number of false positives, the dynamic checking still adds substantial overhead (roughly 20%). Subsequent work (Holey et al., 2013) implements a new race condition detection mechanism via a new hardware component, the Race Detection Unit (RDU). The RDU lowers run-time overhead to a negligible 1%, but increases in memory overhead by about 27%. By developing strictly static machine-validation tools for GPU code, our work seeks to offer an alternative solution to such problems that trades greater software development effort for complete assurance with no runtime overhead.

5.7 Visual Models

Many research works focusing on target tracking and observation have emerged in the past few years. These applications and the popularization of computer vision technologies, like Oculus®, motivate the invention of approaches to observe paths and transportation based on the web mapping service.

Visual models based on human eyesight criteria have been proposed and applied in many relevant researches on computer vision and image processing. In the early stage of this topic, quantitative work are done by the researchers to build models for human visions. The book (Cornsweet, 1970) establishes the basis of this genre on the physical data. Based on the connection between images and human vision, the author of the work (Stockham, 1972) firstly proposes a visual model to predict some visual processing features. Later, a human visual model in paper (Nill, 1985) is incorporated into the procedure of image compression to improve the performance. In paper (Karunasekera and Kingsbury, 1995), the authors propose a more refined visual model to give distortion measure for the blocking

artifacts in images and extra parameters for visual sensitivity are taken into consideration. A recent study (Frome et al., 2013) about machine learning leverages annotations of images to train visual models and improves the performance of visual recognition. And the most popular application about visual models is the virtual reality (VR). The paper (Alvarez-Morales et al., 2017) investigates the similarity and correlation between the visual models and acoustic models of objects, which can serve a better purpose of the immersion in VR.

5.8 Virtual Reality

The increasingly rapid evolution and improving affordability of VR equipment over the past decade has made consumer VR systems extremely popular in the current market. Early work on this subject innovated Head Mounted Displays to bring people into virtual worlds (Chung et al., 1989). Other work uses surround-screens to present an immersive virtual world by projection (Cruz-Neira et al., 1993). Over time, these implementations have been gradually miniaturized. The VR technology of current generation is not limited to merely project the screens into those immersive displays. The researches expand to generate the focuses based on the eyeball movement and pictures and it brings eyes the impression of near-true reality. Many studies support focus cues and several works are listed as follows (Favalora, 2005; Hua and Javidi, 2014; Huang et al., 2015; Lanman and Luebke, 2013). In the frontier research, this article (Bastug et al., 2017) elaborates the opportunities and challenges of Virtual Reality hinged on wireless connection, such as 5G, fog/edge computing, and examines several case studies focusing on AR/VR. At the end, The authors propose questions about the envision that AR/VR will blur the boundary between computer simulation and reality.

Wearable VR goggles with projection are now being used in a variety of contexts. For instance, they are being used in job training related to operating precision instruments (Grantcharov et al., 2004). In the medical field, they are employed to create imaginal exposure or surgical proxies to cure patients psychologically and physically (Rothbaum et al., 2001; Satava, 1995).

Also, simulations related to VR have been extensively explored. In (Menzies et al., 2016) and (Chiarovano et al., 2015), the stability and balance of human in the virtual immersive world are systematically studied.

5.9 Algorithms on Tracking & Coverage

Inside our work, we need study the positioning and coverage of the cameras. So, works related to tracking and coverage are investigated.

The category of monitored targets is an important aspect of this problem space that must be taken into consideration. Targets are classified into different types, such as multi-positions, regions and barriers (e.g., lines and bands). For example, the Pan and Scan Problem entails covering as many targets as possible by deploying specific numbers of cameras in a plane with vast observed targets (Johnson and Bar-Noy, 2011). Prior work on this problem has developed a 2-approximation algorithm that applies Voronoi tessellation to regions, with the goal of lessening the overlap of the cameras' covering fields (Kulkarni et al., 2007). Extensive research has examined the problem of camera coverage of region barriers. Selecting the least number of cameras to cover specific barriers is one focus of this research (Ma et al., 2012).

In most cases, the monitored objects are movable and not fixed at positions. Prior work has innovated a system of localization and tracking built atop mobile phone networks (Kansal and Zhao, 2007). By implementing tracking mechanisms based on the virtual data derived from content inside a camera network, this work improves localization beyond what can be easily achieved with a cellular network alone. Comparing and matching objects inside the visual field facilitates the calculation of information related to targets and their locations. In another work, a system for tracking persons in a 3-dimensional area is proposed (Heath and Guibas, 2008). The cameras in this project detect specific human targets by marking multiple points on objects and tracing them in different cameras. Recently, a method of tracking motion objects more effectively has been devised (Cehovin et al., 2013). It creatively

interlaces objects' global and local features to observe the object more accurately than former approaches.

5.10 Mapping Services

Digital mapping and Global Positioning Systems (GPSes) have opened computer vision research to applications related to transportation and localization. This is now a prolific area of study, so only some contributions related to our work are presented here.

Methods of localizing images with architectural features have been invented and implemented for Google Maps® by leveraging the GPS and Google Maps Street View® information (Zamir and Shah, 2010). The approach accumulates vote counts based on feature matching to find the GPS position of a query image with the highest votes. Input images containing adequate details (interest points) are crucial for successful queries. Most recent research is already investigating the possible information of localization inside cameras—images, and analyzing the location, authenticity and so on. In (Bunk et al., 2017), the authors show two effective methods to detect the genuineness of the location information for image forgeries with the aid of neutral networks. A solid method of estimating the camera position is presented based on the visual information of 2D images in (Sattler et al., 2017) and the authors propose a searching mechanism with low computational complexity to help to locate the position of camera from query images. In contrast, our work focuses on navigation-related scenarios.

Decision support systems have also been leveraged to calculate vehicle routing in Google Maps® (Santos et al., 2011). The major contribution of this work, which is unique to the research on navigation systems, is that many criteria other than distance or time are considered, and these principles influence the standards of evaluation for routing. This affects the role of weight-tagging for each path segment in our problem domain.

Cameras have been used for decades for security monitoring and other situational awareness applications. By combining the concurrent observations of many cameras, visual networks can be built to act for surveillance. Inside this network, every camera is treated as a sensor node and collaborates with the other nodes to collect instant data (in form of video and photos) synchronously (Soro and Heinzelman, 2009). In research involving camera sensor networks, most of the work entails maximizing the covered visual fields and minimizing the number or energy of camera sensors for optimization. Advances include optimized schedules and algorithms to apply camera networks to diverse scenes.

CHAPTER 6

CONCLUSION

The series of original research works presented in this dissertation explore security challenges and opportunities at the boundary between CPU and GPU computations. Verification & Validation plays an irreplaceable role in the current software industry, so a feasible strategy is proposed to bring the heterogeneous computations into the software development life cycle.

One of the most prominent mainstream parallel computation architectures—Nvidia CUDA platform—is selected as our research target, and the computational logic of its intermediate representation is formally described with operational semantics in Coq. Challenges modeling PTX's complex and highly parallelized computation model in Coq, with sufficient clarity and generality to tractably prove useful properties of realistic GPU programs, are discussed. Coq's strict mathematical model with dependent types offers a trustworthy foundation for analysts to inspect parallel programs running on the platform. With evaluation, the correctness, safety, and security of the programs can be machine-validated with mathematical proofs.

To address the associated problem of trustworthy computation of programs that lack formal specifications, an n-variant approach to dynamically detecting faults and intrusions in CPU/GPU software was next presented. Specifically, untrusted Java source code is first translated into GPU compatible code with static analysis techniques. The translated code is packed into GPU kernels and executed by the GPU while the original code runs in the CPU. With this double-lane execution, the divergence of program states at two sides is monitored by the GPU, which signals the CPU to halt the execution with positive detection. Significant differences between the CPU and GPU computational models lead to high natural diversity between the replicas, affording detection of large exploit classes without laborious manual diversification of the code. The implementation shows that the prototype is able to detect publicly recorded exploits without any notification in advance.

Finally, a computational methodology for optimizing physical camera security using CPU and GPU platforms is introduced. The range of camera vision is mathematically modelized and united, so positions of cameras can be calculated and located in a map to offer maximal field of surveillance with least number of cameras. Given a target quantity of cameras, it merges relatively unimportant camera positions to reduce the quantity of video information that must be collected, maintained, and presented. Experiments apply the technique to paths chosen from maps of different cities around the world with various target camera quantities. The approach finds detail-optimizing positions with a time complexity of $O(n \log n)$.

APPENDIX

MODELS AND PROOFS

A.1 PTX Model in Chapter 2

Listing A.1: PTX Model in Coq

```
(* Data Types *)
Inductive dty := UI (w:N) | SI (w:N) | BD (w:N) | Pred.

(* Registers *)
Definition reg : Type := (dty*N*N)%type.
Definition reg_f : Type := reg → Z.

(* Special Registers *)
Inductive dim := Dx | Dy | Dz.

Inductive sreg :=
| T (d : dim)
| NT (d : dim)
| B (d : dim)
| NB (d : dim).
Definition sreg_f : Type := sreg → N.

(* Kernel/Grid Configuration *)
Definition kconf : Type := ((N*N*N)*(N*N*N))%type.
```

```
(* Operands *)
Inductive op :=
| Reg (r:reg)
| SReg (s:sreg)
| Imm (i:Z)
| RegImm (r:reg) (i:Z).

Definition op_f : Type := op → Z.

(* Memory *)
Inductive stsp :=
| Global
| Const
| Shared (bid : N*N*N).

Definition mem_f : Type := (stsp*N) → (Z*bool).

Inductive hl := HI | LO | WIDE.
Inductive bop := ADD | SUB | MUL (v:hl).
Inductive top := MAD (v:hl).
Inductive cmp := EQ | NEQ | LT | GT.
Inductive mss := GSS | CSS | SSS.

(* list of PTX instructions *)
Inductive instr :=
```

```
| Bop (i:bop) (w:N) (d:reg) (a b:op)
| Top (i:top) (w:N) (d:reg) (a b c:op)
| Setp (c:cmp) (w:N) (p:reg) (a b:op)
| Mov (w:N) (d:reg) (a:op)
| Ld (ss:mss) (w:N) (d:reg) (a:op)
| St (ss:mss) (w:N) (a:op) (d:reg)
| Bra (tgt:nat)
| PBra (p:reg) (tgt:nat)
| Bar | Sync | Nop | Exit.

Definition prg : Type := list instr.
Definition prg_f : Type := nat → option instr.

Inductive thread := (N*reg_f)%type.

(* Thread Small Step Transition *)
Inductive thread_step (kc:kconf) (mu:mem_f) : instr → thread → thread → Prop
    :=
| TBop : forall  o w d a b tid  rho f  n
            (Hf: f  = op_exec kc (tid,rho))
            (Hn: bop_eval f w o a b n),
  thread_step kc mu (Bop o w d a b) (tid,rho) (tid,rho r[d ↦ n]])
| TTop : forall  o w d a b c tid  rho f  n
            (Hf: f  = op_exec kc (tid,rho))
            (Hn: top_eval f w o a b c n),
```

```
  thread_step kc mu (Top o w d a b c) (tid,rho) (tid, rho r[ d ↦ n]])
| TSetp : forall  c  w p a b tid  rho f  n
            (Hf: f  = op_exec kc (tid,rho))
            (Hn: setp_eval f w c a  b n),
  thread_step kc mu (Setp c  w p a  b) (tid,rho) (tid , rho r[ p ↦ n]])
| TMov : forall  w d a tid  rho n f
            (Hf: f  = op_exec kc (tid,rho))
            (Hn: n = f a),
  thread_step kc mu (Mov w d a) (tid,rho) (tid,rho r[d ↦ n]])
| TLd  : forall  ss  w d a tid  rho n f  s
            (Hf: f  = op_exec kc (tid,rho))
            (Hs: s  = get_stsp kc tid ss)
            (Hn: n = Z.of_N (read_mem mu (s, Z.to_N (f a)) (N.to_nat (w/8)))),
  thread_step kc mu (Ld ss w d a) (tid,rho) (tid , rho r[ d ↦ n]]).

Inductive warp :=
| Uni (pc :  nat) (ts  :  list  thread)
| Div (w1 w2 : warp).

(* Warp Small Step Transition *)
Inductive warp_t (kc:kconf) (mu:mem_f) : instr → warp → warp * mem_f →
    Prop :=
(* Nonconflict  instructions  *)
| WNop : forall  pc ts,
  warp_t kc mu Nop (Uni pc ts) (Uni (pc+1) ts, mu)
```

```
| WBop : forall  o w d a b ts  ts'  pc
          (Hts':  ts'  = nd_map (thread_exec kc mu (Bop o w d a b)) ts),
  warp_t kc mu (Bop o w d a b) (Uni pc ts) (Uni (pc+1) ts', mu)
| WTop : forall  o w d a b c ts  ts'  pc
          (Hts':  ts'  = nd_map (thread_exec kc mu (Top o w d a b c)) ts),
  warp_t kc mu (Top o w d a b c) (Uni pc ts) (Uni (pc+1) ts', mu)
| WSetp : forall  c  w p a b ts  ts'  pc
          (Hts':  ts'  = nd_map (thread_exec kc mu (Setp c w p a b)) ts),
  warp_t kc mu (Setp c w p a b) (Uni pc ts) (Uni (pc+1) ts', mu)
| WMov : forall  w d a ts  ts'  pc
          (Hts':  ts'  = nd_map (thread_exec kc mu (Mov w d a)) ts),
  warp_t kc mu (Mov w d a) (Uni pc ts) (Uni (pc+1) ts', mu)
| WLd  :  forall  ss  w d a ts  ts'  pc
          (Hts':  ts'  = nd_map (thread_exec kc mu (Ld ss w d a)) ts),
  warp_t kc mu (Ld ss w d a) (Uni pc ts) (Uni (pc+1) ts', mu)
| WBra : forall  pc tgt  ts,
  warp_t kc mu (Bra tgt) (Uni pc ts) (Uni tgt ts,  mu)
| WPBra : forall p tgt  pc ts  ts1  ts2  w'
          (Hsplit:  (ts1,ts2)  = partition  (prd_bool p) ts)
          (Hw': w' = match is_empty ts1, is_empty ts2 with
                     | true,  _ ⇒ Uni (pc+1) ts2
                     | false ,  true ⇒ Uni tgt ts1
                     | false ,  false  ⇒ Div (Uni tgt ts1) (Uni (pc+1) ts2) end),
  warp_t kc mu (PBra p tgt) (Uni pc ts) (w', mu)
(* Potential  conflict  instructions  *)
```

```
| WStG : forall  w a d pc ts  mu' f l
          (Hl: l  = map (st_eval kc Global w a d) ts)
          (Hf: f  = fun m p ⇒ match p with (a,v) ⇒ m m[a ↦ (v,false)]] end)
          (Hmu': mu' = fold_left f l  mu),
  warp_t kc mu (St GSS w a d) (Uni pc ts) (Uni (pc+1) ts, mu')
| WStS : forall  w a d pc ts  mu' f l
          (Hl: l  = map (st_eval kc (Shared (get_cta kc ts)) w a d) ts)
          (Hf: f  = fun m p ⇒ match p with (a,v) ⇒ m m[a ↦ (v,false)]] end)
          (Hmu': mu' = fold_left f l  mu),
  warp_t kc mu (St SSS w a d) (Uni pc ts) (Uni (pc+1) ts, mu')
| WSync : forall w w'
          (Hsync: sync w w'),
  warp_t kc mu Sync w (w',mu)
(* Recursive call  to  get  appropriate (leftmost)  warp *)
| WDiv : forall  i  w1 w1' w2 mu'
          (Hi: i  <> Sync)
          (Hw': warp_t kc mu i w1 (w1',mu')),
  warp_t kc mu i (Div w1 w2) (Div w1' w2, mu').

(* Block Small Step Transition *)
Definition block : Type := list warp.

Inductive block_t (kc:kconf) (pi:prg_f) (mu:mem_f) : block → block * mem_f →
    Prop :=
| BStep  :  forall  i  b b1 b'  w w' mu' n
```

```
                (Hb: nth_ri n b w b1)
                (Hi: pi (warp_pc w) = Some i)
                (Hi1: i <> Bar /\ i <> Exit)
                (Hstep: warp_t kc mu i w (w',mu'))
                (Hb': nth_ri n b' w' b1),
    block_t kc pi mu b (b',mu')
| BEndBar : forall b b'
                (Hbar: block_barred pi b = true)
                (Hlb: b' = lift_bar b),
    block_t kc pi mu b (b',mu).

(* Grid Small Step Transition *)
Definition grid : Type := list block.
Inductive grid_t (kc:kconf) (pi:prg_f) (mu:mem_f) : grid → grid * mem_f →
    Prop :=
| GStep : forall n g g1 g' b b' mu'
                (Hg: nth_ri n g b g1)
                (Hbc: block_complete pi b = false)
                (Hstep: block_t kc pi mu b (b',mu'))
                (Hg': nth_ri n g' b' g1),
    grid_t kc pi mu g (g',mu').

(* Kernel Configuration *)
Definition empty_reg : reg_f := fun _ ⇒ 0%Z.
```

```
Fixpoint enumerate_threads (n:nat) (tid:N) : list thread :=
  match n with
  | O ⇒ []
  | S n' ⇒ (tid, empty_reg) :: enumerate_threads n' (tid+1)
end.

Fixpoint generate_grid (n bsize tid:nat) : grid :=
  match n with
  | O ⇒ []
  | S n' ⇒ let bts := enumerate_threads bsize (N.of_nat tid) in
            let ws := group bts 32 in
            let b := map (fun ts ⇒ Uni 0 ts) ws in
            b :: generate_grid n' bsize (tid+bsize)
end.

Definition generate_kernel (kc:kconf) : grid :=
  match kc with ((xg,yg,zg),(xb,yb,zb)) =>
  let nb := (xg*yg*zg)%N in
  let nt := (xb*yb*zb)%N in
  generate_grid (N.to_nat nb) (N.to_nat nt) 0
end.
```

A.2 SDV Rules for lock steps in Chapter 2

Listing A.2: SDV Rules for lock steps in COQ

```
Inductive ruleSDV :
threadStates_n × list (stmt×pred_f) →
threadStates_n × list (stmt×pred_f) → Prop :=
| gBasic n ss  bs  p Sigma Sigma' def:
  (forall  (i :  nat),
   (i <n) → rulePRE ((nth i Sigma def), bs ,p) (nth i Sigma' def)
  ) →
  ruleSDV (Sigma, (bcstt bs, p):: ss )  (Sigma', ss )

| gDivergence Sigma p ss n def :
  (exists  (i  j :  nat),  (i <>j /\ i<n /\ j<n) →
   (  ((th1_prd (nth i Sigma def)) (th1_id (nth i Sigma def)) = true) /\
      ((th1_prd (nth j Sigma def)) (th1_id (nth j Sigma def)) = false)
   )
  ) →
  ruleSDV (Sigma, (barrier,  p):: ss )  errorSDV

| gNoOp Sigma p ss n def :
  (forall  (i : nat),
   (i <n) → ((th1_prd (nth i Sigma def)) (th1_id (nth i Sigma def)) = false)
  ) →
 ruleSDV (Sigma, (barrier,  p):: ss )  (Sigma, ss)

| gRace Sigma p ss n def :
  (forall  (i : nat),
```

```
    (i <n) → ((th1_prd (nth i Sigma def)) (th1_id (nth i Sigma def)) = true)
     /\  (races  Sigma)
   ) →
  ruleSDV (Sigma, (barrier, p):: ss )  errorSDV

| gSync Sigma p ss n def Sigma' :
   (forall   (i : nat),
    (i <n) → ((th1_prd (nth i Sigma def)) (th1_id (nth i Sigma def)) = true)
     /\  ~(races  Sigma)
     /\ (forall   (i : nat),  (i <n) → (nth i Sigma' def) =
   (th1_thrd (nth i Sigma def), merge Sigma, empty_set word, empty_set word) )
   ) →
  ruleSDV (Sigma, (barrier, p):: ss )  (Sigma', ss )

| gSeq Sigma S1 S2 p ss:
  ruleSDV (Sigma, (squ S1 S2, p)::ss)  (Sigma, (S1, p)::( S2, p):: ss )

| gVar Sigma x S p ss v Vloc:
   ( set_In v Vloc
   ) →
  ruleSDV (Sigma, (local x S, p):: ss )  (Sigma,(sub_var S x v, p)::ss )

| gIf  Sigma p ss e S1 S2 v Vloc    :
   ( set_In v Vloc
   ) →
```

```
 ruleSDV (Sigma, (sif e S1 S2,p):: ss )
(Sigma,
(bcstt (bsc v e), opra_and p v)::(S1, p)::( S2, opra_and p (opra_not v))::ss)

| gOpen Sigma p ss e S v Vloc:
  ( set_In v Vloc
  ) →
 ruleSDV (Sigma, (swhile e S, p):: ss )
          (Sigma, (swhile_ e (belim S v), opra_and p (opra_not v))::ss)

| gIter  Sigma n p e v Vloc u q S ss def:
  ( exists  (i : nat), (i <n) → evalue (opra_and p e) (nth i Sigma def)
    /\ set_In v Vloc
    /\ set_In u Vloc
    /\ q =  opra_and (opra_and p v) (opra_not v)
  ) →
 ruleSDV (Sigma, (swhile_ e S, p)::ss )
          (Sigma, (bcstt (bsc u e), p)::( celim S v, q)::( swhile_ e S, p):: ss )

| gDone n p e Sigma def S ss:
  ( exists  (i : nat), (i <n) → ~ (evalue (opra_and p e) (nth i Sigma def))
  ) →
 ruleSDV (Sigma, (swhile e S, p):: ss ) (Sigma, ss)

| gCall v u Vloc S f Sigma p ss e:
```

```
  ( set_In v Vloc
    /\ set_In u Vloc
    /\ S = sub_var (Body f) (Param f) u
  ) →
  ruleSDV (Sigma, (namefor (Param f) e, p)::ss)
(Sigma, (squ (bcstt (bsc u e)) (relim S v), opra_and p (opra_not v))::ss).
```

A.3 Proof of Theorem 1 in Chapter 4

Proof. Given the condition that all the objects are not in a line, ℓ is a polyline. For the following calculations, the circumcircle $\bigcirc p$ must be found first. Without loss of generality, assume that circle $\bigcirc p$ has p as its center and r as its radius, as is shown in Figure A.1. This circumcircle is the minimal area that the view range of camera c must cover in order to fully observe object ℓ with any angle toward it. Place camera c with AOV ϕ at a position where its two boundary lines are tangent to $\bigcirc p$, and let d be the distance between c and p. All the candidate positions for c comprise an orbit circle $\bigcirc p'$ with center p and radius d. This is because the view scope is able to precisely cover the circumcircle irrespective of the position on the orbit where the camera is placed. Also, circumcircle $\bigcirc p$ is a demarcation:

- When camera c observes the objects ℓ at a position outside $\bigcirc p$, it gains no more observed weight than it does on the boundary of the circle, since w is inverse to $d_{p\ell}$ in the formula of the camera model. The further the camera moves away from the object beyond critical distance d_0, the less observed weight it collects.

- When camera x observes the objects ℓ at a position inside $\bigcirc p$, it may gain more observed weight than it does on the boundary of the circle, but it may not cover the objects fully. This is because $\bigcirc p$ is the minimal area that the view range of camera c must cover in order to fully observe object ℓ.

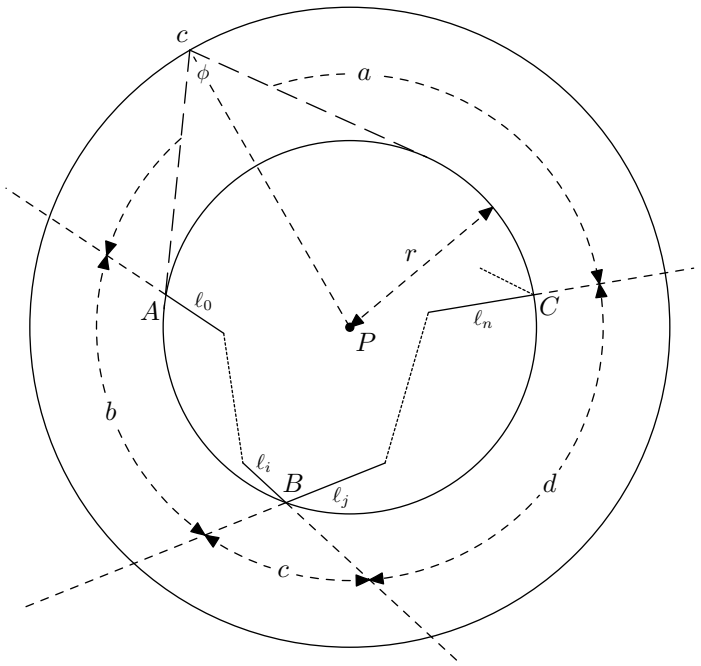


Figure A.1: The orbit boundaries of camera c when c moving along the circle $\bigcirc p\prime$. It is for calculating the local extramum of the observed weight.

It is easy to calculate that $d = \frac{r}{\sin(\phi/2)}$. With those two facts about the circumcircle above, the theorem is proved. □

A.4 The optimal position for a single camera in Chapter 4

This section introduces the method of finding the specific position for one single camera c to collect the maximal observed weight. For convenience, we here refer to this as its *optimal position*.

As shown in Figure A.1, given a polyline $\ell = \{\ell_0, \ldots, \ell_i, \ell_j, \ldots, \ell_n, \ldots\}$ and $\bigcirc p$ the circumcircle of ℓ, $\bigcirc p\prime$ is the orbit circle of camera c. Since only one specific circle can pass three non-colinear points, without loss of generality assume A, B, and C are three[1] chosen

[1]Although a circumcircle may intersect more than 3 points of ℓ, in this procedure it suffices to pick exactly 3 points; the existence of any extra points on the circumcircle will not affect the result.

points on the circumcircle of ℓ, and they are also the endpoints of line segments a, b, c, and d, respectively.

Because of Theorem 1, we know the optimal position exists on or in the circle. It helps to eliminate all the area outside the circle $\bigcirc p\prime$, so the first step is to search for the optimal position on the orbit circle for camera c. From the given condition, each segment of ℓ forms a different angle γ with the straight line $\overline{cp}$ between camera c and p. When c starts moving, record every angle γ_{0i} formed by each segment with the line $\overline{cp}$ at the starting position. All angles γ_i must change by the same $\Delta\gamma$ while c is moving, given by $\gamma_i = \gamma_{0i} + \Delta\gamma$. Then, a function can be created between observed weight w and $\Delta\gamma$ based on the formula of the obeserving model:

$$w = \sum_{i=0}^{n} w_i = \sum_{i=0}^{n} f(\gamma_{0i} + \Delta\gamma) \text{ where } \Delta\gamma \in [0, 2\pi)$$

Here, f is the formula in Section 4.3.3. By calculating the first derivative of this function of summation with respect to $\Delta\gamma$, the maximum can be found in the extrema. This yields the optimal position on the orbit.

To check whether the optimal position on orbit is the final optimal one, we must check whether camera c is able to move toward ℓ under the condition that the visual scope of c can fully cover ℓ. If the boundary lines of the visual scope of c already touch ℓ at some point on ℓ, then the optimal position on the orbit is the final optimal position. Otherwise, the camera can be moved from the optimal position on the orbit circle toward center p, stopping at the position where the boundary lines of the visual scope of c exactly touch ℓ at some point on ℓ. This position inside circle $\bigcirc p\prime$ is the final optimal position.

The procedure above infers the optimal position for a single camera relative to a continuous polyline of any number of segments. This generalizes the problem considered in the project of VisualVital. To specialize it to our problem, we divide the candidate area into four parts to accommodate situations where segments could block each other. The division

procedure can be implemented as follows: As shown in Figure A.1, segments ℓ_0, ℓ_i, ℓ_j and ℓ_n will be extended outwards to divide the candidate area into four parts: a, b, c, and d. Parts b and d are eliminated due to the blocking situation. Therefore, the search for the optimal position in this instance limits the candidate area to parts a and c. The optimal position can be computed by comparing the four boundary values on a and c With the extrema by calculating the first derivative of the function above along the two intervals a and c.

BIBLIOGRAPHY

Aciu, R.-M. and H. Ciocarlie (2016). Runtime translation of the Java bytecode to OpenCL and GPU execution of the resulted code. *Acta Polytechnica Hungarica 13*(3), 25–44.

Agrawal, H. (1999). Efficient coverage testing using global dominator graphs. In *Proc. ACM SIGPLAN-SIGSOFT Work. Program Analysis for Software Tools and Engineering (PASTE)*, pp. 11–20.

Aldinucci, M. and M. Danelutto (2007, Oct.-Dec.). Skeleton-Based Parallel Programming: Functional and Parallel Semantics in a Single Shot. *Computer Languages, Systems & Structures 33*(3-4), 179–192. Elsevier.

Alkabani, Y. and F. Koushanfar (2008). N-variant IC design: Methodology and applications. In *Proc. 45th ACM/IEEE Design Automation Conf. (DAC)*, pp. 546–551.

Alshawabkeh, M., B. Jang, and D. R. Kaeli (2010). Accelerating the local outlier factor algorithm on a GPU for intrusion detection systems. In *Proc. 3rd Work. General Purpose Processing Using Graphics Processing Units (GPGPU)*, pp. 104–110.

Alvarez-Morales, L., J. F. Molina-Rozalem, S. GirÃşn, A. Alonso, P. Bustamante, and A. Alvarez-Corbacho (2017, July). Virtual reality in church acoustics: Visual and acoustic experience in the cathedral of seville, spain. In *Proc. 2017 International Congress on Sound and Vibration (ICSV)*.

Astarita, V., G. Guido, and V. P. Giofré (2014). Co-operative ITS: Smartphone based measurement systems for road safety assessment. *Procedia Computer Science 37*, 404–409.

Atkey, R. (2007). CoqJVM: An executable specification of the Java virtual machine using dependent types. In *Proc. Int. Conf. Types for Proofs and Programs (TYPES)*, pp. 18–32.

Attariyan, M. and J. Flinn (2010). Automating configuration troubleshooting with dynamic information flow analysis. In *Proc. USENIX Sym. Operating Systems Design and Implementation (OSDI)*, pp. 237–250.

Avižienis, A. (1985). The n-version approach to fault-tolerant software. *IEEE Trans. Software Engineering (TSE) 11*(12), 1491–1501.

Bakkum, P. and K. Skadron (2010). Accelerating SQL database operations on a GPU with CUDA. In *Proc. 3rd Work. General-purpose Processing Graphics Processing Units (GPGPU)*, pp. 94–103.

Ball, T. and J. R. Larus (1996). Efficient path profiling. In *Proc. 29th Annual ACM/IEEE Int. Sym. Microarchitecture (MICRO)*, pp. 46–57.

Bardsley, E., A. Betts, N. Chong, P. Collingbourne, P. Deligiannis, A. F. Donaldson, J. Ketema, D. Liew, and S. Qadeer (2014). Engineering a static verification tool for GPU kernels. In *Proc. 26th Int. Conf. Computer Aided Verification (CAV)*, pp. 226–242.

Barthe, G., B. Grégoire, S. Heraud, and S. Zanella-Béguelin (2011). Computer-aided security proofs for the working cryptographer. In *Proc. 31st Annual Conf. Advances in Cryptology (CRYPTO)*, pp. 71–90.

Barthe, G., B. Köpf, F. Olmedo, and S. Zanella-Béguelin (2013). Probabilistic relational reasoning for differential privacy. *ACM Trans. Programming Languages and Systems (TOPLAS) 35*(3).

Bastug, E., M. Bennis, M. Medard, and M. Debbah (2017). Toward interconnected virtual reality: Opportunities, challenges, and enablers. *IEEE Communications Magazine 55*(6), 110–117.

Baxter, J. (2017). An approach to verification of safety-critical Java virtual machines with ahead-of-time compilation. Technical report, University of York.

Bertot, Y. and P. Castéran (2004). *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer-Verlag.

Betts, A., N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson (2012). GPUVerify: A verifier for GPU kernels. In *Proc. ACM Int. Conf. Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pp. 113–132.

Bhatkar, S., R. Sekar, and D. C. DuVarney (2005). Efficient techniques for comprehensive protection from memory error exploits. In *Proc. 14th USENIX Security Sym.*

Boldo, S., J.-H. Jourdan, X. Leroy, and G. Melquiond (2013). A formally-verified C compiler supporting floating-point arithmetic. In *Proc. 21st IEEE Int. Sym. Computer Arithmetic (ARITH)*, pp. 107–115.

Bunk, J., J. H. Bappy, T. M. Mohammed, L. Nataraj, A. Flenner, B. Manjunath, S. Chandrasekaran, A. K. Roy-Chowdhury, and L. Peterson (2017). Detection and localization of image forgeries using resampling features and deep learning. In *Proc. 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 1881–1889.

Cehovin, L., M. Kristan, and A. Leonardis (2013). Robust visual tracking using an adaptive coupled-layer visual model. *IEEE Trans. Pattern Analysis and Machine Intelligence 35*(4), 941–953.

Chen, L. and A. Avižienis (1978). N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. IEEE Int. Conf. Fault-tolerant Computing (FTCS)*, pp. 3–9.

Chen, L., X. Huo, and G. Agrawal (2012). Accelerating MapReduce on a coupled CPU-GPU architecture. In *Proc. 24th Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*.

Chiarovano, E., C. de Waele, H. G. MacDougall, S. J. Rogers, A. M. Burgess, and I. S. Curthoys2 (2015, Jul). Maintaining balance when looking at a virtual reality three-dimensional display of a field of moving dots or at a virtual reality scene. *Frontiers in Neurology* (6).

Chiba, S. (2000). Load-time structural reflection in Java. In *Proc. 14th European Conf. Object-oriented Programming (ECOOP)*, pp. 313–336.

Chung, J. C., M. R. Harris, F. P. Brooks, H. Fuchs, M. T. Kelley, J. Hughes, M. Ouh-Young, C. Cheung, R. L. Holloway, and M. Pique (1989). Exploring virtual worlds with head-mounted displays. In *Three-Dimensional Visualization and Display Technologies, SPIE Proc. 1083*, pp. 42–52.

Cohen, F. (1993). Operating system protection through program evolution. *Computers and Security 12*(6), 565–584.

Collingbourne, P., A. F. Donaldson, J. Ketema, and S. Qadeer (2013). Interleaving and lock-step semantics for analysis and verification of GPU kernels. In *Proc. 22nd Int. Conf. European Symposium on Programming (ESOP)*, pp. 270–289.

Cornes, C. and D. Terrasse (1995). Automating inversion of inductive predicates in Coq. In *Proc. Int. Conf. Types for Proofs and Programs (TYPES)*, pp. 85–104.

Cornsweet, T. N. (1970). *Visual Perception*. Academic Press.

Coutinho, B., D. Sampaio, F. M. Q. Pereira, and W. Meira, Jr. (2011). Divergence analysis and optimizations. In *Proc. Int. Conf. Parallel Architectures and Compilation Techniques (PACT)*, pp. 320–329.

Cox, B., D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser (2006). N-variant systems: A secretless framework for security through diversity. In *Proc. 15th USENIX Security Sym.*

Cruz-Neira, C., D. J. Sandin, and T. A. DeFanti (1993). Surround-screen projection-based virtual reality: The design and implementation of the CAVE. In *Proc. 20th Annual Conf. Computer Graphics and Interactive Techniques (SIGGRAPH)*, pp. 135–142.

Cybulski, J. L., S. Keller, and D. Saundage (2014). Metaphors in interactive visual analytics. In *Proc. 7th Int. Sym. Visual Information Communication and Interaction (VINCI)*, pp. 212–215.

Datta, A., M. Shah, and N. D. V. Lobo (2002). Person-on-person violence detection in video data. In *Proc. 16th Int. Conf. Pattern Recognition (ICPR)*.

Dean, J. and S. Ghemawat (2008). MapReduce: Simplified data processing on large clusters. *Communications ACM (CACM) 51*(1), 107–113.

Deschizeaux, B. and J.-Y. Blanc (2007). Imaging earth's subsurface using CUDA. In H. Nguyen (Ed.), *GPU Gems 3*. NVidia Corporation.

Diamos, G., A. Kerr, and S. Yalamanchili (2010). Ocelot: An open source debugging and compilation framework for CUDA. Technical report, GPU Technology Conference (GTC).

D'Silva, V., D. Kroening, and G. Weissenbacher (2008). A survey of automated techniques for formal software verification. *IEEE Trans. Computer-aided Design Integrated Circuits and Systems (TCAD) 27*(7), 1165–1178.

Duan, J., K. W. Hamlen, and B. Ferrell (2019). Better Late Than Never: An n-variant framework of verification for java source code on CPU x GPU hybrid platform. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19, pp. 207–218.

Duan, J., K. Zhang, and K. W. Hamlen (2017, Aug). VisualVital: An observation model for multiple sections of scenes. In *2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pp. 1–8.

Favalora, G. E. (2005, Aug). Volumetric 3d displays and application infrastructure. *Compute 38*(8), 37–44.

Ferrell, B., J. Duan, and K. W. Hamlen (2019, March). CUDA au Coq: A framework for machine-validating GPU assembly programs. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 474–479.

Flatt, M., S. Krishnamurthi, and M. Felleisen (2002). A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pp. 241–269. Springer.

Flur, S., K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell (2016). Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *Proc. 43rd ACM SIGPLAN-SIGACT Sym. Principles of Programming Languages (POPL)*.

Forrest, S., A. Somayaji, and D. H. Ackley (1997). Building diverse computer systems. In *Proc. 6th Work. Hot Topics in Operating Systems (HotOS)*, pp. 67–72.

Frome, A., G. S. Corrado, J. Shlens, S. Bengio, J. Dean, M. Ranzato, and T. Mikolov (2013). Devise: A deep visual-semantic embedding model. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems 26*, pp. 2121–2129. Curran Associates, Inc.

Garg, R. and J. N. Amaral (2010). Compiling Python to a hybrid execution environment. In *Proc. 3rd Work. General-purpose Computation Graphics Processing Units (GPGPU)*, pp. 19–30.

Godefroid, P. (1996). *Partial-order Methods for the Verification of Concurrent Systems: An Approach to the State-explosion Problem*. Berlin, Heidelberg: Springer-Verlag.

Goetz, B. (2014, December). State of the specialization. `http://cr.openjdk.java.net/~briangoetz/valhalla/specialization.html`.

Grantcharov, T. P., V. B. Kristiansen, J. Bendix, L. Bardram, J. Rosenberg, and P. Funch-Jensen (2004). Randomized clinical trial of virtual reality simulation for laparoscopic skills training. *British J. Surgery 91*(2), 146–150.

Grauer-Gray, S., W. Killian, R. Searles, and J. Cavazos (2013). Accelerating financial applications on the GPU. In *Proc. 6th Work. General Purpose Processing Using Graphics Processing Units (GPGPU)*, pp. 127–136.

Habermaier, A. (2011). The model of computation of CUDA and its formal semantics. Technical Report 2011-14, Institut für Informatik, U. Augsburg.

Hampapur, A., K.-H. Hyun, and R. M. Bolle (2002). Comparison of sequence matching techniques for video copy detection. In *Storage and Retrieval for Media Databases, SPIE Proc. 4676*, pp. 194–201.

Han, T. D. and T. S. Abdelrahman (2011). Reducing branch divergence in GPU programs. In *Proc. 4th Work. General Purpose Processing Graphics Processing Units (GPGPU)*.

Havelund, K. and T. Pressburger (2000). Model checking JAVA programs using JAVA PathFinder. *Int. J. Software Tools for Technology Transfer (STTT) 2*(4), 366–381.

Heath, K. and L. Guibas (2008). Multi-person tracking from sparse 3D trajectories in a camera sensor network. In *Proc. 2nd ACM/IEEE Int. Conf. Distributed Smart Cameras (ICDSC)*, pp. 1–9.

Heer, J. and B. Shneiderman (2012). Interactive dynamics for visual analysis. *Communications ACM (CACM) 55*(4), 45–54.

Heukelman, D. and S. E. Obono (2009). Exploring the African village metaphor for computer user interface icons. In *Proc. Annual Research Conf. South African Institute of Computer Scientists and Information Technologists (SAICSIT)*, pp. 132–140.

Hoh, B., M. Gruteser, R. Herring, J. Ban, D. Work, J.-C. Herrera, A. M. Bayen, M. Annavaram, and Q. Jacobson (2008). Virtual trip lines for distributed privacy-preserving traffic monitoring. In *Proc. 6th Int. Conf. Mobile Systems, Applications, and Services (MobiSys)*, pp. 15–28.

Hoh, B., T. Iwuchukwu, Q. Jacobson, D. Work, A. M. Bayen, R. Herring, J.-C. Herrera, M. Gruteser, M. Annavaram, and J. Ban (2012). Enhancing privacy and accuracy in probe vehicle-based traffic monitoring via virtual trip lines. *IEEE Trans. Mobile Computing 11*(5), 849–864.

Holey, A., V. Mekkat, and A. Zhail (2013). HAccRG: Hardware-accelerated data race detection in GPUs. In *Proc. 42nd Int. Conf. Parallel Processing (ICPP)*, pp. 60–69.

Holzmann, G. J. (1997). State compression in SPIN: Recursive indexing and compression training runs. In *Proc. 3rd Int. SPIN Work.*

Hong, C., D. Chen, W. Chen, W. Zheng, and H. Lin (2010). MapCG: Writing parallel program portable between CPU and GPU. In *Proc. 19th Int. Conf. Parallel Architectures and Compilation Techniques (PACT)*, pp. 217–226.

Hua, H. and B. Javidi (2014, Jun). A 3d integral imaging optical see-through head-mounted display. *Opt. Express 22*(11), 13484–13491.

Huang, F.-C., K. Chen, and G. Wetzstein (2015, jul). The light field stereoscope: Immersive computer graphics via factored near-eye light field displays with focus cues. *ACM Trans. Graph. 34*(4), 60:1–60:12.

Huang, J., S. P. Ponce, S. I. Park, Y. Cao, and F. Quek (2008). GPU-accelerated computation for robust motion tracking using the CUDA framework. In *Proc. 5th Int. Conf. Visual Information Engineering (VIE)*, pp. 437–442.

Hubert, T. and C. Marché (2005). A case study of C source code verification: The Schorr-Waite algorithm. In *Proc. 3rd IEEE Int. Conf. Software Engineering and Formal Methods (SEFM)*, pp. 190–199.

Igarashi, A., B. C. Pierce, and P. Wadler (2001). Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Programming Languages And Systems (TOPLAS) 23*(3), 396–450.

Jackson, T., B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz (2011). Compiler-generated software diversity. In *Moving Target Defense*, pp. 77–98. Springer.

Jansohn, C., A. Ulges, and T. M. Breuel (2009). Detecting pornographic video content by combining image features with motion information. In *Proc. 17th ACM Int. Conf. Multimedia (MM)*, pp. 601–604.

Jiang, W. and G. Agrawal (2012). MATE-CG: A map reduce-like framework for accelerating data-intensive computations on heterogeneous clusters. In *Proc. IEEE 26th Int. Parallel and Distributed Processing Sym. (IPDPS)*, pp. 644–655.

Johnson, M. P. and A. Bar-Noy (2011). Pan and scan: Configuring cameras for coverage. In *Proc. 30th IEEE Int. Conf. Computer Communications (INFOCOM)*, pp. 1071–1079.

Kahn, G. (1974). The semantics of a simple language for parallel programming. In *Proc. IFIP Congress*, pp. 471–475.

Kahsai, T., P. Rümmer, H. Sanchez, and M. Schäf (2016). JayHorn: A framework for verifying Java programs. In *Proc. 28th Int. Conf. Computer Aided Verification (CAV)*, pp. 352–358.

Kansal, A. and F. Zhao (2007). Location and mobility in a sensor network of mobile phones. In *Proc. 17th ACM SIGMM Int. Work. Network and Operating Systems Support for Digital Audio & Video (NOSSDAV)*.

Karunasekera, S. A. and N. G. Kingsbury (1995, June). A distortion measure for blocking artifacts in images based on human visual sensitivity. *IEEE Transactions on Image Processing 4*(6), 713–724.

Kennedy, A., N. Benton, J. B. Jensen, and P.-E. Dagand (2013). Coq: The world's best macro assembler? In *Proc. 15th Sym. Principles and Practice of Declarative Programming (PPDP)*, pp. 13–24.

Kennelly, C. (2012). Panoptes: A binary translation framework for CUDA. Technical report, GPU Technology Conference (GTC).

Kirner, R., S. Herhut, and S.-B. Scholz (2010). Compiler-support for robust multi-core computing. In *Proc. 4th Int. Conf. International Symposium on Leveraging Applications (ISoLA)*, pp. 47–57.

Knight, J. C. and N. G. Leveson (1986). An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Software Engineering (TSE) 12*(1), 96–109.

Kulkarni, P., P. Shenoy, and D. Ganesan (2007). Approximate initialization of camera sensor networks. In *Proc. 4th European Conf. Wireless Sensor Networks (EWSN)*, pp. 67–82.

Lam, V., S. Phan, D.-D. Le, D. A. Duong, and S. Satoh (2017). Evaluation of multiple features for violent scenes detection. *Multimedia Tools and Applications 76*(5), 7041–7065.

Lanman, D. and D. Luebke (2013, nov). Near-eye light field displays. *ACM Trans. Graph. 32*(6), 220:1–220:10.

Larsen, P., A. Homescu, S. Brunthaler, and M. Franz (2014). SoK: Automated software diversity. In *Proc. 35th IEEE Sym. Security & Privacy (S&P)*, pp. 276–291.

Lee, S. and C. D. Yoo (2008). Robust video fingerprinting for content-based video identification. *IEEE Trans. Circuits and Systems for Video Technology 18*(7), 983–988.

Leroy, X. (2003). Java bytecode verification: Algorithms and formalizations. *J. Automated Reasoning 30*(3), 235–269.

Leroy, X. (2009). Formal verification of a realistic compiler. *Communications ACM (CACM) 52*(7), 107–115.

Li, P., C. Ding, X. Hu, and T. Soyata (2014). LDetector: A low overhead race detector for GPU programs. In *Proc. 5th Int. Conf. Workshop on Determinism and Correctness in Parallel Programming (WoDet)*.

Ma, H., M. Yang, D. Li, Y. Hong, and W. Chen (2012). Minimum camera barrier coverage in wireless camera sensor networks. In *Proc. 31st IEEE Int. Conf. Computer Communications (INFOCOM)*, pp. 217–225.

Maitre, O. (2013). Understanding NVIDIA GPGPU hardware. In S. Tsutsui and P. Collet (Eds.), *Massively Parallel Evolutionary Computation on GPGPUs*, pp. 15–34. Springer.

McBride, C. (2002). Elimination with a motive. In *Proc. Int. Conf. Types for Proofs and Programs (TYPES)*, pp. 197–216.

Menzies, R. J., S. J. Rogers, A. M. Phillips, E. Chiarovano, C. de Waele, F. A. J. Verstraten, and H. MacDougall (2016, Sep). An objective measure for the visual fidelity of virtual reality and the risks of falls in a virtual environment. *Virtual Reality 20*(3).

Messmer, P., P. J. Mullowney, and B. E. Granger (2008). GPULib: GPU computing in high-level languages. *Computing in Science & Engineering 10*(5), 70–73.

Milano, D. (2012). Content control: Digital watermarking and fingerprinting. White Paper, Rhozet, Harmonic Inc.

Min, C., S. Kashyap, B. Lee, C. Song, and T. Kim (2015). Cross-checking semantic correctness: The case of finding file system bugs. In *Proc. 25th Sym. Operating Systems Principles (SOSP)*, pp. 361–377.

Mittal, S. and J. S. Vetter (2015). A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys (CSUR) 47*(4).

Mongiovì, M., G. Giannone, A. Fornaia, G. Pappalardo, and E. Tramontana (2015). Combining static and dynamic data flow analysis: A hybrid approach for detecting data leaks in Java applications. In *Proc. 30th Annual ACM Sym. Applied Computing (SAC)*, pp. 1573–1579.

Morrisett, G., G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan (2012). RockSalt: Better, faster, stronger SFI for the x86. In *Proc. 33rd ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pp. 395–404.

Nguyen-Tuong, A., D. Evans, J. C. Knight, B. Cox, and J. W. Davidson (2008). Security through redundant data diversity. In *Proc. IEEE Int. Conf. Dependable Systems and Networks (DSN)*, pp. 187–196.

Nievas, E. B., O. D. Suarez, G. B. García, and R. Sukthankar (2011). Violence detection in video using computer vision techniques. In *Proc. 14th Int. Conf. Computer Analysis of Images and Patterns (CAIP)*, pp. 332–339.

Nill, N. B. (1985, June). A visual model weighted cosine transform for image compression and quality assessment. *IEEE Transactions on Communications 33*(6), 551–557.

Nilsson-Nyman, E., G. Hedin, E. Magnusson, and T. Ekman (2008). Declarative intraprocedural flow analysis of Java source code. In *Proc. 8th Work. Language Descriptions, Tools and Applications (LDTA)*, pp. 155–171.

nVIDIA (2015). Parallel thread execution ISA v4.3 application guide. `http://docs.nvidia.com/cuda/pdf/ptx_isa_4.3.pdf`.

Pai, S., R. Govindarajan, and M. J. Thazhuthaveetil (2010). PLASMA: Portable programming for SIMD heterogeneous accelerators. In *Proc. Work. Language, Compiler, and Architecture Support for GPGPU*.

Pandey, A. V., A. Manivannan, O. Nov, M. Satterthwaite, and E. Bertini (2014). The persuasive power of data visualization. *IEEE Trans. Visualization and Computer Graphics 20*(12), 2211–2220.

Pawlak, R., M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier (2015). SPOON: A library for implementing analyses and transformations of Java source code. *Software: Practice and Experience 46*(9), 1155–1179.

Pedoe, D. (1957). *Circles: A Mathematical View*. International Series of Monographs on Pure and Applied Mathematics. Dover Publications.

Petcher, A. and G. Morrisett (2015). The foundational cryptography framework. In *Proc. 4th Int. Conf. Principles of Security and Trust (POST)*, pp. 11–18.

Pratt-Szeliga, P. C., J. W. Fawcett, and R. D. Welch (2012). Rootbeer: Seamlessly using GPUs from Java. In *Proc. IEEE 14th Int. Conf. High Performance Computing and Communication (HPCC) & IEEE 9th Int. Conf. Embedded Software and Systems (ICESS)*, pp. 375–380.

Rapps, S. and E. J. Weyuker (1985). Selecting software test data using data flow information. *IEEE Trans. Software Engineering (TSE) 11*(4), 367–375.

Reitblatt, M., N. Foster, J. Rexford, C. Schlesinger, and D. Walker (2012). Abstractions for network update. In *Proc. ACM Conf. Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pp. 323–334.

Rothbaum, B. O., L. F. Hodges, D. Ready, K. Graap, and R. D. Alarcon (2001). Virtual reality exposure therapy for Vietnam veterans with posttraumatic stress disorder. *J. Clinical Psychiatry 62*(8), 617–622.

Salamat, B., T. Jackson, A. Gal, and M. Franz (2009). Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proc. 4th ACM European Conf. Computer Systems (EuroSys)*.

Santos, L., J. Coutinho-Rodrigues, and C. H. Antunes (2011). A web spatial decision support system for vehicle routing using Google Maps. *Decision Support Systems 51*(1), 1–9.

Satava, R. M. (1995). Virtual reality, telesurgery, and the new world order of medicine. *J. Image Guided Surgery 1*(1), 12–16.

Sattler, T., B. Leibe, and L. Kobbelt (2017, Sept). Efficient effective prioritized matching for large-scale image-based localization. *IEEE Transactions on Pattern Analysis and Machine Intelligence 39*(9), 1744–1756.

Schuster, F., T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz (2015). Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *Proc. 36th IEEE Sym. Security & Privacy (S&P)*, pp. 745–762.

Seamans, E. and T. Alexander (2007). Fast virus signature matching on the GPU. In H. Nguyen (Ed.), *GPU Gems 3*. NVidia Corporation.

Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. 14th ACM Conf. Computer and Communications Security (CCS)*, pp. 552–561.

Shacham, H., M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh (2004). On the effectiveness of address-space randomization. In *Proc. 11th ACM Conf. Computer and Communications Security (CCS)*, pp. 298–307.

Shirahata, K., H. Sato, and S. Matsuoka (2010). Hybrid map task scheduling for GPU-based heterogeneous clusters. In *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science (CloudCom)*, pp. 733–740.

Soner, S. and C. Özturan (2012). Integer programming based heterogeneous CPU-GPU cluster scheduler for SLURM resource manager. In *Proc. 14th High Performance Computing and Communication & 9th Int. Conf. Embedded Software and Systems (HPCC-ICESS)*, pp. 418–424.

Soro, S. and W. Heinzelman (2009). A survey of visual sensor networks. *Advances in Multimedia 2009*. doi:10.1155/2009/640386.

Stärk, R. F., J. Schmid, and E. Börger (2012). *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer Science & Business Media.

Stockham, T. G. (1972). Image processing in the context of a visual model. *Proceedings of the IEEE 60*(7), 828–842.

Su, T., K. Wu, W. Miao, G. Pu, J. He, Y. Chen, and Z. Su (2017). A survey on data-flow testing. *ACM Computing Surveys (CSUR) 50*(1).

Swamy, N., J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang (2011). Secure distributed programming with value-dependent types. In *Proc. 16th ACM SIGPLAN Int. Conf. Functional Programming (ICFP)*, pp. 266–278.

Tsoi, K. H. and W. Luk (2010). Axel: A heterogeneous cluster with FPGAs and GPUs. In *Proc. 18th Annual ACM/SIGDA Int. Sym. Field Programmable Gate Arrays (FPGA)*, pp. 115–124.

Veldema, R., T. Blass, and M. Philippsen (2011). Enabling multiple accelerator acceleration for Java/OpenMP. In *Proc. 3rd USENIX Conf. Hot Topic in Parallelism (HotPar)*.

Visser, W., C. S. Păsăreanu, and S. Khurshid (2004). Test input generation with java PathFinder. In *Proc. ACM SIGSOFT Int. Sym. Software Testing and Analysis (ISSTA)*, pp. 97–107.

Volckaert, S., B. Coppens, B. D. Sutter, K. D. Bosschere, P. Larsen, and M. Franz (2017). Taming parallelism in a multi-variant execution environment. In *Proc. 12th European Conf. Computer Systems (EuroSys)*, pp. 270–285.

Wicker, T. (1963, Nov. 22,). Kennedy is killed by sniper as he rides in car in Dallas; Johnson sworn in on plane. *The New York Times*, 1.

Yamanouchi, T. (2007). AES encryption and decryption on the GPU. In H. Nguyen (Ed.), *GPU Gems 3*. NVidia Corporation.

Yang, J. and D. Evans (2004). Automatically inferring temporal properties for program evolution. In *Proc. 15th Int. Sym. Software Reliability Engineering (ISSRE)*, pp. 340–351.

Yuan, J., L.-Y. Duan, Q. Tian, S. Ranganath, and C. Xu (2004). Fast and robust short video clip search for copy detection. In *Advances in Multimedia Information Processing, 5th Pacific Rim Conf. Multimedia (PCM)*, pp. 479–488.

Zamir, A. R. and M. Shah (2010). Accurate image localization based on Google Maps street view. In *Proc. 11th European Conf. Computer Vision (ECCV)*, pp. 255–268.

Zheng, M., V. T. Ravi, F. Qin, and G. Agrawa1 (2014). GMRace: Detecting data races in GPU programs via a low-overhead scheme. *IEEE Trans. Parallel and Distributed Systems 25*(1), 104–115.

Zheng, M., V. T. Ravi, F. Qin, and G. Agrawal (2011). GRace: A low-overhead mechanism for detecting data races in GPU programs. In *Proc. 16th ACM Sym. Principles and Practice Parallel Programming (PPoPP)*, pp. 135–146.

BIOGRAPHICAL SKETCH

Born in Shanxi, China, Jun Duan came to the United States to pursue his doctoral degree in computer science at The University of Texas at Dallas. At the end of his first year there, he joined the Software Language Security Laboratory led by Dr. Kevin W. Hamlen. The constantly-emerged research topics in the area of cybersecurity opened a brand-new world to him and he decided to concentrate in the domain of GPU-related security. Since then, he has been conducting a fair amount of research in the field and co-authored several papers which were published in renowned international venues. He has also been invited to present his work to prestigious conferences, such as ACM HPDC and DATE.

Before enrolling into the PhD program, he studied and worked in Beijing, China. He acquired his Master's Degree in computer science at Renmin (People's) University of China in the summer of 2011. There, he was advised by Dr. Deying Li and engaged in research on computer networks. Due to his persistent effort and publication of his work, he was honored with the Outstanding Graduate Student Award. After graduation, he joined the Department of Software Development of China Life Insurance Company, the largest insurer in China, where he worked as a software engineer for two years before moving to the United States.

CURRICULUM VITAE

Jun Duan

March 9th, 2021

Contact Information:

Department of Computer Science
The University of Texas at Dallas
800 W. Campbell Rd.
Richardson, TX 75080-3021, U.S.A.

Email: jun.duan@utdallas.edu

Educational History:

M.S., Computer Science, Renmin University of China, 2011
M.S., Computer Science, The University of Texas at Dallas, 2019
Ph.D., Computer Science, The University of Texas at Dallas, 2021

Securing Computations with GPUs
Ph.D. Dissertation
Erik Jonsson School of Engineering & Computer Science, The University of Texas at Dallas
Advisors: Dr. Kevin W. Hamlen

Geometric Routing Algorithms in 3-Dimensional Wireless Sensor Networks
M.S. Dissertation
School of Information, Renmin University of China
Advisor: Dr. Deying Li

Employment History:

Software Engineer, China Life Insurance Company, August 2011 – August 2013

Professional Recognitions and Honors:

Outstanding Graduate Student Award, Renmin University of China, 2011

Publications:

•*Better Late Than Never: An n-Variant Framework of Verification for Java Source Code on CPU × GPU Hybrid Platform.* Jun Duan, Kevin W. Hamlen, and Benjamin Ferrell (2019). In *Proc. 28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*

•*CUDA au Coq: A Framework for Machine-validating GPU Assembly Programs.* Benjamin Ferrell, Jun Duan, and Kevin W. Hamlen (2019). In *Proc. 26th Design, Automation & Test in Europe Conference & Exhibition (DATE)*

• *VisualVital: An Observation Model for Multiple Sections of Scenes.* Jun Duan, Kang Zhang, and Kevin W. Hamlen (2017). In *Proc. 14th IEEE International Conference on Ubiquitous Intelligence and Computing (UIC)*

• *3D geometric routing without loops and dead ends in wireless sensor networks.* Jun Duan, Deying Li, Wenping Chen, and Zewen Liu (2014). In *Journal of Ad Hoc Networks, Vol. 13*

• *A New Localized Geometric Routing with Guaranteed Delivery on 3-D Wireless Networks.* Jun Duan, Donghyun Kim, Wenping Chen, and Deying Li (2012). In *Proc. 21st International Conference on Computer Communications and Networks (ICCCN/IC3N)*

• *Geometric Routing Precluding Loops and Dead Ends in 3-D Wireless Sensor Networks.* Jun Duan, Deying Li, and Wenping Chen (2010). In *Proc. 2010 IEEE Global Telecommunications Conference (GLOBECOM)*