A project report on

SAFECUDA: SOFTWARE-ONLY MEMORY-SAFETY FOR GPUs VIA PTX GUARDS AND MANAGED SHADOW METADATA CACHE

Submitted in partial fulfillment for the award of the degree of

Bachelor of Technology in Computer Science and Engineering

by

KIRAN P DAS (22BCE1216)

NAVIN KUMAR A O (22BCE1020)

ANIRUDH SRIDHAR (22BCE5252)

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

CHENNAI

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

November, 2025

SAFECUDA: SOFTWARE-ONLY MEMORY-SAFETY FOR GPUs VIA PTX GUARDS AND MANAGED SHADOW METADATA CACHE

Submitted in partial fulfillment for the award of the degree of

Bachelor of Technology in Computer Science and Engineering

by

KIRAN P DAS (22BCE1216)

NAVIN KUMAR A O (22BCE1020)

ANIRUDH SRIDHAR (22BCE5252)

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

CHENNAI

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

November, 2025

VIT®

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)
CHENNAI

DECLARATION

I hereby declare that the thesis entitled "SAFECUDA- SOFTWARE-ONLY MEMORY-SAFETY FOR GPUs VIA PTX GUARDS AND MANAGED SHADOW METADATA CACHE" submitted by A.O. NAVIN KUMAR (22BCE1020), for the award of the degree of Bachelor of Technology in Computer Science and Engineering, Vellore Institute of Technology, Chennai is a record of bonafide work carried out by me under the supervision of **Dr. MANJU G**.

I further declare that the work reported in this thesis has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Place: Chennai

Date: 29|10|25

Navin Kumar

Signature of the Candidate

VIT

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)
CHENNAI

School of Computer Science and Engineering

CERTIFICATE

This is to certify that the report entitled **"SafeCUDA: Software-only memory-safety for GPUs using PTX guards and Managed shadow metadata cache"** is prepared and submitted by **A.O. Navin Kumar (22BCE1020)** to Vellore Institute of Technology, Chennai, in partial fulfillment of the requirement for the award of the degree of **Bachelor of Technology in** Computer Science And Engineering is a bonafide record carried out under my guidance. The project fulfills the requirements as per the regulations of this University and in my opinion meets the necessary standards for submission. The contents of this report have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma and the same is certified.

Signature of the Guide:

Name: Dr. Manju G
Date: 04-11-2025

Signature of Examiner:
Name: Dr. P. M. Bharadhan
Date: 05-11-2025

Signature of Examiner:
Name: Dr. S. Aravind kumar.
Date: 5/11/25.

Approved by the Head of Department,
BTECH - Computer Science and Engineering

SCHOOL OF COMPUTER SCIENCE & ENGINEERING (SCOPE)
VELLORE INSTITUTE OF TECHNOLOGY
Chennai-600127

Name: Dr. J Prassana
Date

ABSTRACT

Memory safety remains as one of the largest and most difficult to address concerns in the world of programming. This is the case especially for CUDA-based applications that operate at large and distributed scale. Errors such as Buffer Overflow, Use-after-free, Double-free, and race conditions can lead to silent data corruption, unexpected behaviour or complete application failure, and most importantly, memory leaks can lead to many security issues with varying impact.

For CPUs there are many memory safety guaranteeing tools like AddressSanitizer or Valgrind, but that is not the case for GPUs due to the highly decoupled nature of GPU codes, which leads to memory safety tools requiring changes in source code, or sometimes even specialised hardware to function as intended. This project introduces a software-only memory-safety enforcement system for such CUDA applications, designed to detect and prevent memory-safety related errors without modifying GPU drivers or hardware, or the source code of the project itself. The proposed solution proposes a two-tier validation system, with both the compile-time and run-time processes that validate and check for memory safety issues.

Experimental evaluation shows that the proposed system can successfully handle and report any memory-safety violations that may occur while maintaining a relatively low runtime and memory overhead cost. This, combined with the fact that there is no source code modification or specialized hardware required for the proposed system to function, show that a software-only memory protection system is feasible and achievable for CUDA, thus providing a valuable debugging and verification tool for GPU developers. This work contributes toward safer and more reliable GPU software development by bridging the gap between performance-driven parallel programming and robust memory safety guarantees.

ACKNOWLEDGEMENT

It is my pleasure to express with deep sense of gratitude to **Dr. Manju G**, Professor, School of Computer Science and Engineering, Vellore Institute of Technology, Chennai, for her constant guidance, continual encouragement, understanding; more than all, she taught me patience in my endeavour. My association with her is not confined to academics only, but it is a great opportunity on my part to work with an intellectual expert in the field of Low level design and Compiler Design.

It is with gratitude that I would like to extend my thanks to the visionary leader Dr. G. Viswanathan our Honorable Chancellor, Mr. Sankar Viswanathan, Dr. Sekar Viswanathan, Dr. G V Selvam Vice Presidents, Dr. Sandhya Pentareddy, Executive Director, Ms. Kadhambari S. Viswanathan, Assistant Vice-Pesident, Dr. V. S. Kanchana Bhaaskaran Vice-Chancellor, Dr. T. Thyagarajan Pro-Vice Chancellor, VIT Chennai and Dr. P. K. Manoharan, Additional Registrar for providing an exceptional working environment and inspiring all of us during the tenure of the course.

Special mention to Dr. Viswanathan V, Dean, Dr. Nithyanandam P, Dr. Suganya G, Dr. Sweetlin Hemalatha C, Associate Deans, School of Computer Science and Engineering, Vellore Institute of Technology, Chennai for spending their valuable time and efforts in sharing their knowledge and for helping us in every aspect.

In jubilant state, I express ingeniously my whole-hearted thanks to Dr. Prasanna J, Head of the Department, B.Tech. Computer Science and Engineering and the Project Coordinators for their valuable support and encouragement to take up and complete the thesis.

My sincere thanks to all the faculties and staffs at Vellore Institute of Technology, Chennai who helped me acquire the requisite knowledge. I would like to thank my parents for their support. It is indeed a pleasure to thank my friends who encouraged me to take up and complete this task.

Place: Chennai

Date: 29|10|25

A.O. Navin Kumar

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ACRONYM	DEFINITION
CUDA	Complete Unified Device Architecture
GPU	Graphics Processing Unit
PTX	Parallel Thread Execution
CPU	Central Processing Unit
NVCC	NVIDIA CUDA Compiler
CVE	Common Vulnerabilities and Exposure
CICC	NVIDIA CUDA Compiler Driver
OOB	Out Of Bounds
CUBIN	CUDA Binary
FATBIN	Fat Binary
PTXAS	PTX Assembly
HPC	High Performance Computing
SM	Streaming Multiprocessor
IDE	Integrated Development Environment

CHAPTER 1
INTRODUCTION

1.1 INTRODUCTION

Graphics Processing Units (GPUs) have become a fundamental component in modern computing, enabling high performance execution of parallel workloads across fields such as Deep Learning, Data Analytics, Scientific Simulation, etc. The ability to be able to execute codes that use the GPU's memory from the user Central Processing Unit (CPU) enables us to parallelise and speed up hundreds of thousands of computational tasks that would otherwise take a very long time to compile and run. NVIDIA's Compute Unified Device Architecture (CUDA) provides developers with a platform and a flexible programming model to harness these capabilities through C/C++ extensions.

However, as applications and projects scale to higher levels of complexity, ensuring memory safety becomes a bigger and bigger challenge, due to the highly decoupled nature of CUDA. Memory Safety in this context refers to the correct and secure management of memory – initialisation and de-initialisation of all objects managed by the developer – during program execution. Common memory errors such as buffer overflows, invalid memory accesses and race conditions can lead to undefined behaviour, data corruption and security vulnerabilities. These issues are often difficult to detect because of CUDA's massive parallel execution model and limited debugging visibility inside device kernels.

To address memory vulnerabilities in CPU-based applications, mature tools such as Valgrind provide robust and reliable memory safety guarantees through dynamic analysis and instrumentation. However, equivalent solutions for GPU programs remain limited. Existing GPU debugging and safety tools often depend on specialized hardware support or introduce substantial performance and memory overheads. This limitation primarily arises from CUDA's dual-path compilation model, in which host and device code are compiled separately for the CPU and GPU architectures. Although these components follow distinct compilation paths, the NVIDIA CUDA compiler (NVCC) manages their integration during the linking phase, ensuring that kernel launches in host code correctly reference their corresponding device functions.

1.2 OVERVIEW OF THE PROJECT

The proposed system combines a runtime and a compile-time check system to ensure that all memory requests are properly validated and accounted for, and to minimise the occurrence of any such requests that aren't validated.

1.2.1 COMPILE-TIME PROCESS

During compilation, PTX instructions are generated by NVCC. The proposed system intercepts the generation of these instructions and injects specific instructions to make necessary modifications to ensure memory safety and validation while maintaining the internal compilation sequence.

1.2.2 RUNTIME PROCESS

After compilation and linking via NVCC, a global metadata cache is constructed alongside a CUDA managed dynamic array. These data structures will function as the main record keepers for all incoming memory requests, tracking and ensuring that no memory leaks happen.

1.2.3 SAFECUDA LINKING PROCESS

The proposed system provides a Shared Object file (.so) as the executable for developers to use. The developer has to link the shared object file by setting the LD_PRELOAD flag in the terminal, and then they can write CUDA code normally, and on compiling, the proposed system will flag all memory leaks and memory safety issues, if any.

1.3 CHALLENGES PRESENT IN THE DOMAIN

Ensuring memory safety in GPU-based applications presents a range of challenges that differ significantly from those encountered in traditional CPU programming. The CUDA programming model introduces additional layers of complexity due to its unique architecture, execution model, and compilation process. Some of the key challenges in this domain are outlined below:

1.3.1 PARALLELISM AND CONCURRENCY

CUDA programs execute thousands of lightweight threads concurrently across multiple cores. This massive parallelism makes it difficult to detect and reproduce memory errors such as race conditions, out-of-bounds accesses, or data corruption, which may only appear under specific thread scheduling or data patterns.

1.3.2 LIMITED DEBUGGING VISIBILITY

Unlike CPU programs, where tools such as Valgrind and AddressSanitizer provide detailed memory diagnostics, GPU kernels operate in a separate device memory space. This separation limits visibility into runtime memory operations and complicates error tracing, especially during device-only execution.

1.3.3 HARDWARE AND ARCHITECTURAL CONSTRAINTS

Many existing GPU memory safety tools depend on hardware-level features or custom instrumentation that are not available on all devices. Such reliance restricts portability and prevents developers from using these tools on general-purpose consumer GPUs.

1.3.4 HIGH PERFORMANCE SENSITIVITY

GPUs are highly optimized for parallel throughput. Any runtime instrumentation or additional safety checks can significantly impact performance. Designing memory safety systems that balance accuracy and overhead remains a critical challenge.

1.3.5 COMPLEX COMPILATION AND LINKING MODEL

CUDA’s dual-path compilation model—where host and device code are compiled separately—introduces potential inconsistencies in symbol resolution, data layout, and pointer translation between CPU and GPU memory spaces. Ensuring consistent tracking of memory objects across both environments adds substantial complexity to static and dynamic analysis tools.

1.3.6 LIMITED TOOLING AND ECOSYSTEM SUPPORT

While CPUs benefit from decades of mature debugging and analysis frameworks, GPU memory safety tooling remains in early stages. Many available tools

focus primarily on performance profiling or functional debugging rather than comprehensive memory safety enforcement.

1.4 PROJECT STATEMENT

Memory safety issues such as buffer overflows, invalid pointer dereferences, and race conditions pose a significant challenge in CUDA-based GPU applications. These errors can lead to data corruption, undefined behavior, or program crashes that are difficult to detect due to CUDA's massively parallel execution and limited debugging visibility. While CPUs have mature tools like Valgrind and AddressSanitizer for detecting such issues, equivalent solutions for GPUs are limited, often relying on specialized hardware or introducing substantial performance penalties.

This project aims to develop a software-only memory safety enforcement system for CUDA applications, capable of detecting and preventing memory violations without requiring hardware modifications or driver-level intervention. The proposed system leverages compile-time PTX instrumentation to insert guard instructions and runtime validation mechanisms to ensure safe and verified memory access throughout program execution.

1.5 OBJECTIVES

The primary objectives of this project is to make a bulletproof memory safety enforcement software that can detect and report any memory leaks and related CVEs so that developers can make memory-safe applications without incurring extra development time and effort. The system proposes a two-fold check system, wherein there is a check during compile-time and there is a check during runtime. The system instruments PTX during compile-time to inject specific instructions that adds calls to the related runtime bounds-checking functions while maintaining the overall execution of the compile-time the same. This facilitates the other half of the system, wherein runtime checks are performed on all memory requests and a global shadow metadata cache is maintained to track all requests and a fallback array is maintained for bounds checks for any un-accounted for memory requests. This solution also aims to be a drop-in addition to a CUDA project to help with memory safety without requiring the developers to make any modifications to the source code or add any specialised

hardware to facilitate the application. This solution also aims to minimise any memory and/or runtime overhead by being extremely lightweight and efficient. The tertiary objectives of this project include benchmarking on individual performance and against other existing solutions, to evaluate system's accuracy, speed and performance.

1.6 SCOPE OF THE PROJECT

The scope of this project is limited to a software-only solution for the GPU memory safety problem. The solution targets common memory safety CVEs like OOB accesses, use-after-free, double-free and invalid pointer-referencing within device code. It operates primarily at the PTX (Parallel Thread Execution) intermediate representation level, enabling compatibility across different NVIDIA GPU architectures and CUDA versions.

The system is intended primarily as a development and debugging tool, designed to assist researchers and developers in identifying and resolving memory-related vulnerabilities in GPU applications efficiently.

CHAPTER 2

LITERATURE SURVEY

2.1 INTRODUCTION

In recent years GPU Computing paradigm has become vital in optimizing performance in several different fields like Deep Learning, Physics Simulation, Data analytics, Cryptography etc. It serves as the backbone of all the LLM development and deployment in recent years owing to its ability to massively parallelize the workload several times faster and efficiently than CPUs. NVIDIA GPUs that are based on CUDA programming model [1] have been established as the dominant and cutting edge in the high-performance GPU market. This widespread adoption has given rise to several memory safety concerns that are well documented by several researchers.

Miele [2] presented a preliminary study that focuses on buffer overflow vulnerabilities in CUDA architecture. It demonstrates how a buffer overflow can be used to overwrite a function pointer. This essentially allows the user to completely change the flow of execution by replacing a given function with a malicious one. This makes the CUDA architecture very vulnerable to limited return-oriented programming exploits. The exploit can be used to corrupt data required for scientific research, theft or poisoning weights of a machine learning model and sabotaging critical systems like autonomous vehicles, medical imaging etc.

Sang-Ok Park et al. [3] introduced a novel GPU memory exploitation method that can be used to reliably decrease the accuracy of a deep learning model to random guessing. This exploit can be performed on three major deep learning frameworks like TensorFlow, CNTK, and Caffe running on GPUs using the CUDA architecture. This is achieved by hijacking the control flow of a GPU application and using that to gain read and write access to GPU memory. They also introduced a new GPU memory exploit technique called code warping exploit that uses both code reuse and code injection to modify the control flow of the application. The paper further also suggests certain defense mechanisms to protect DL models against these attacks. It involves revoking the write permission to memory pages that contain the GPU code to make sure the

attacker cannot hijack the control flow.

Performing analysis of GPU memory vulnerability is considerably more challenging due to the lack of documentation for both hardware and software interfaces involved in a GPU. The ISA of a GPU is usually not made public by the vendor. Most of the critical parts of NVIDIA s' CUDA architecture like device firmware, drivers and runtime tools are closed source and proprietary. Thus most of the analysis that has been performed by researchers have either been by reverse engineering the architecture or a trial and error method. This is especially true for the NVIDIA s' Unified Memory feature which is not understood properly by the 3rd party researchers. Unified Memory feature allows both CPU and GPU to access the same pool of virtual memory while reducing data copying and improving the overall performance.

Xabier Arauzo et al.[4] exposed the internal behaviour of Unified Memory by using reverse engineering. Furthermore, this knowledge gained is also used to invent novel ways to avoid data migration and to reduce the runtime of the application. Apart from optimisation purposes this also helps researchers understand the possible memory vulnerabilities that could occur and different ways to overcome them.

Before delving into the realm of GPU memory safety and validation, it's prudent we understand more about how memory safety is implemented in CPU. Boivie et al.[5] introduced a hardware module that can perform spatial and temporal memory validation. D. Bruening et al.[6] implemented a memory checking tool that is compatible with both windows and linux using fast instrumentation, memory shadowing and transparent wrapping to improve performance. Stepanov et al. [7] proposed a tool called MemorySanitizer that uses compiler instrumentation instead of binary instrumentation and bit-precise shadow memory at run-time. Tony Chen et al. [8] proposed a change in ISA to make sure data can't be misused as code/data pointer by implementing a change in the pointer bit. Dudina et al.[9] implemented a static analysis to convert legacy C/C++ code into memory safe CHERI framework to provide memory safety at scale and with low overhead. V.E Moghadam [10] presented a survey that systematically classify, analyze and compare the advancements in memory safety strategies by providing taxonomy, evaluation criteria and critical discussion

The section focuses on two papers that tackle memory safety in CUDA using software-only and hardware assisted approaches respectively. Both of these papers originate from researchers working for NVIDIA and hence are very vital in analysing the possible solutions available.

2.2 cuCatch

Cucatch[11] is a runtime memory safety solution to ensure proper temporal and spatial memory safety in CUDA GPUs. It uses the principle of memory tagging and base & bounds checking to create a hybrid algorithm that provides memory safety without high runtime overhead. It uses a BST(Base Size Table) to store each allocation's base, size, and tag in metadata. Each time a memory allocation is made a metadata entry is created in the BST and if there are enough unused bits available in the pointer the BST index is directly stored in the pointer. Once the number of allocations exceeds the limit a shadow map is used to store the BST index.

Each time the pointer is used to make a load/store operation cuCatch first checks if the upper bits of the pointer contains the BST index. If it does it directly fetches the metadata or else it uses the shadow map to fetch the metadata. This metadata is stored in the registers and effectively propagated to other memory instructions using the same pointer. This effectively creates a fat pointer that contains all the information required for proper memory checking.

When the particular allocation is deleted a corresponding flag is set to prevent free after use memory error. Furthermore the shadow map entry associated with this is set to point to an illegal zeroth entry.

2.3 GPUArmour

The main idea behind GPUARMOUR[12] is using hardware assisted mechanisms to provide memory safety with negligible overhead. The software component first intercepts every memory allocation call and stores the metadata like

base, size, tag in a disjoint table called MPT (Memory Protection Table). Each time a kernel issues a load/store operation in the runtime on the allocated memory the metadata is first fetched and analysed to make sure this operation does not violate spatial or temporal memory safety.

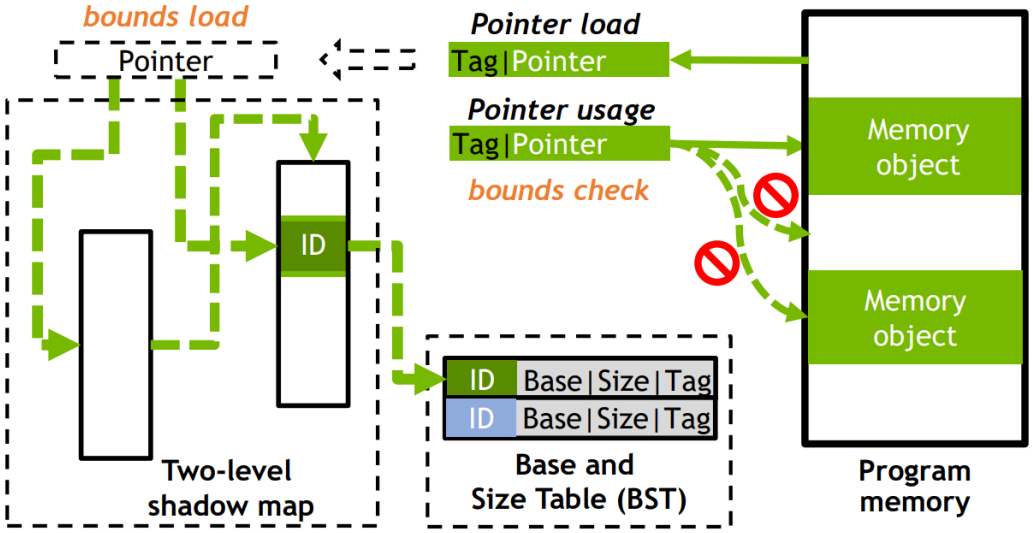

Figure 2.1: cuCatch execution flow

The main cause of runtime overhead in this scheme would be fetching the correct metadata associated with the reference. This is why other implementations that are software only like cuCatch[10] or GPUShield[13] incurs massive overheads. GPUARMOUR deals with this issue by using a dedicated hardware module to minimize the lookup latency by incorporating tag based indexing and caching. Each memory allocation is associated with a unique tag that is usually stored in the upper unused bit of a GPU pointer.

The hardware module called MLU(Metadata Loading Unit) intercepts the kernel memory calls like ld.global*/st.global*/ato.global*. It first searches for a unique tag in the MLB(Memory Lookaside Buffer) Cache to fetch the metadata entry. If the unique tag is not found in the cache then it searches the MPT and stores the result in the cache. This approach does not require any changes to the source code and can work seamlessly with the current CUDA architecture. The results have shown that this hardware accelerated approach only causes runtime overhead of 2.2% which is far better than the other software-only approach

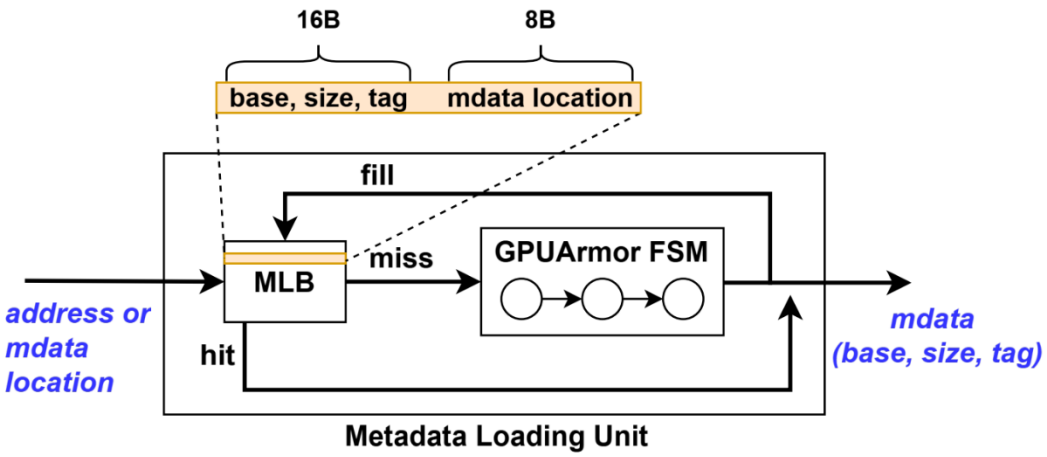

Figure 2.2: GPUArmour execution flow

2.4 OTHER RELEVANT RESEARCH

Valgrind[14] is a programming tool that is used to perform memory profiling, detect memory leak and debugging. Valgrind uses binary instrumentation and executes the program in virtual CPU using shadow memory to keep track of whether each byte is allocated, freed, lost etc. However this tool can not be used for GPU applications because the host and device maintain their own memory space. The CPU does not have full access to monitor the GPU memory space and Valgrind only operates on CPU memory and instruction. T.M. Baumann et al.[15] implemented a Valgrind wrapper that adds support for CUDA drivers to enable runtime checks on both host and device side memory. Although this solution is effective in covering the memory transaction, an additional wrapper is required to pass the arguments of the kernel that is being run. Furthermore the integration can be even more deepened by designing a virtual GPU that is capable of running the PTX code alongside the virtual CPU to fully emulate the GPU application.

B. Kopcke et al.[16] introduced Descend, a new memory safe programming language for GPU application. This new programming language is inspired by the Rust programming language and optimised for parallel programming. Rust programming language provides strict memory safety through its ownership system which enforces

several rules on borrowing and lifetime at compile time to prevent spatial and temporal memory bugs. It prevents data races by making sure if any one thread can mutate certain data then other threads can not have concurrent access to that data. Although efficient this might be in CPU programming, It is impractical in parallel programming. This pitfall is addressed by Descend by allowing different non overlapping execution resources (threads or groups) to mutate data at the same time. Descend matches the performance of CUDA while retaining low level control and providing memory safety. Although this seems like a very promising solution it is far away from adoption due to very immature tooling and a steep learning curve.

K. Huang et al.[17] presents a novel way to approach memory safety by isolating memory safe objects from memory unsafe objects to minimise the probability of a runtime error occurring and reducing the number of errors possible at compile time. They use static analysis and symbolic execution to make sure whether every alias (reference) to an object must only be used in ways that conform to spatial, temporal and type safety. If every alias to an object passes this condition then they are considered a safe object. Safe objects on a stack are stored in a completely safe stack and the ones on heap are stored within a safe zone. This type of isolation on both heap and stack memory depends on the ASLR (address space layout randomization). This majorly reduces the runtime checks as those only need to be performed on the objects that are considered unsafe. The results show that this approach can protect 85% of stack objects and 77% of heap objects with no runtime overhead. The paper does not explore ways to perform runtime checks on the objects deemed unsafe but suggests several previous research works like ASan[18] and FuZZan[19]. This is especially more effective because according to certain previous research approximately 90% of the pointers in c/c++ never use pointer arithmetic or type casting which are the major reasons why memory bugs occur (CCured[20]).

Y. Yang et al.[21] introduced a dynamic protection mechanism that dynamically detects and prevents GPU heap buffer overflows by using an efficient CPU-assisted mechanism. J. Lee et al. [22] proposed a novel way of achieving memory safety by embedding buffer size in the upper bits of a pointer and performing compiler analysis to mark pointer operations. A.J Calderón et al.[23] addressed the issue related to dynamic memory allocation by performing a static analysis of memory allocation

requests and placing them into centralized fixed-size pools to avoid dynamic failures. Henriksen et al.[24] introduced a compiler instrumentation in a high level language like Futhark to ensure memory safety. Z Chen[25] proposed a source code level instrumentation where a memory access is rewritten with metadata to perform smart memory to catch out-of-bounds and dangling-pointer errors. Bang Di et al.[26] implemented canary based runtime along with CPU assisted Unified memory check to ensure memory safety.

CHAPTER 3

METHODOLOGY

3.1 INTRODUCTION

The methodology for SafeCUDA is a carefully architected multi-layer solution designed to inject, propagate, and enforce memory safety both spatial and temporal across every phase of the GPU application lifecycle, while retaining compatibility with the familiar CUDA workflow. Building on the principle of minimal friction, this system is engineered to function seamlessly without requiring developers to rewrite source code, touch build scripts, or alter project structure. In this section, every stage from compilation, instrumentation, and runtime library interposition to safety enforcement and benchmarking is dissected, elaborated, and contextualized, capturing the engineering ingenuity and depth behind the entire approach.

3.2 GENERAL WORKFLOW

SafeCUDA's architecture revolves around the concept of a transparent wrapper layered above NVIDIA's canonical CUDA Compiler, `nvcc`. The wrapper, `sf-nvcc`, orchestrates an interception of PTX (Parallel Thread Execution) artifacts, injecting bespoke safety instrumentation without disturbing the internal sequencing of the standard compilation process. The essential core of this flow is preservation of build compatibility: applications constructed using make, CMake, or other toolchains remain unaware of the safety machinery embedded below. The process is drop-in, non-intrusive, and explicitly portable.

At the build stage, sf-nvcc leverages the keep-and-collect abilities of nvcc to capture PTX streams destined for the GPU. Devices are targeted through architectural flags, manifesting as SM (Streaming Multiprocessor) targets. sf-nvcc then analyzes PTX through multi-pass tokenization and pattern recognition, parsing individual directives and operands at the lowest level. When a memory operation be it a load, store, or atomic involving global memory is detected, the wrapper interjects a guarded

bounds-check, passing the effective pointer register forward for validation. Instrumentation is both fine-grained and location-aware: every access that could touch GPU heap is scrutinized at its execution site and proper checks are inserted.

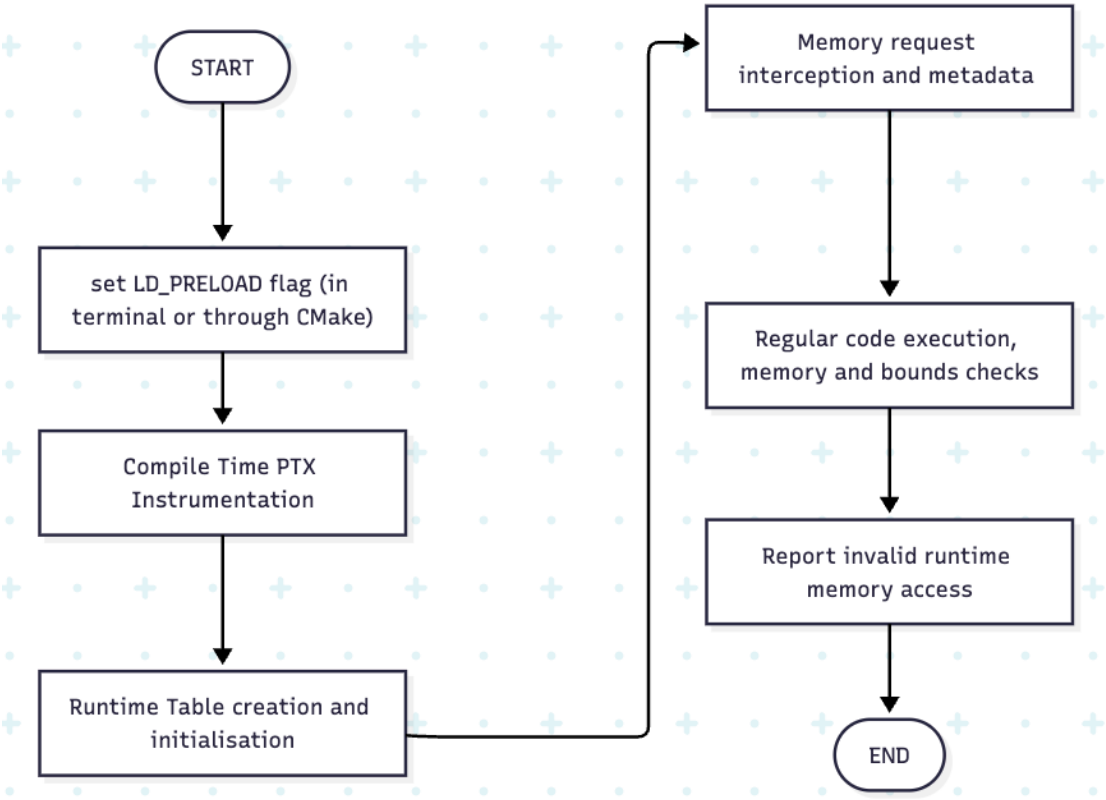

Figure 3.1: General Higher-level SafeCUDA Workflow

The host-side process then resumes as normal. CUDA's own build machinery packs cubins (the binary device code) into a fatbin (a fatbinary, which is a compiled file which has the data of multiple compiled cubins), which is embedded in the final host executable. None of this process disturbs host code, C++ extensions are automatically rewritten and compiled to match vanilla application structure.

3.3 DATA STRUCTURES AND METADATA ENGINEERING

The heart of SafeCUDA's memory safety system lies in the interplay of fast-access headers and a comprehensive metadata table. Every heap allocation, from the perspective of both kernel and host, is annotated via a header that sits precisely 16 bytes before the returned user pointer. Inside this header rests a magic tag and a pointer or index to a metadata row, which persists in a global table until program termination.

The metadata row includes a base address, size, and status flags, a bitmask flag used for reporting the different kinds of violations. It is engineered as a persistent, efficient lookup context: it supports fast insertions on allocation, explicit invalidation on free, all with zero impact on host code. Alignment padding, quarantining prior to reuse, and careful design of the cache coherence mechanisms ensure robust correctness under adversarial usage conditions.

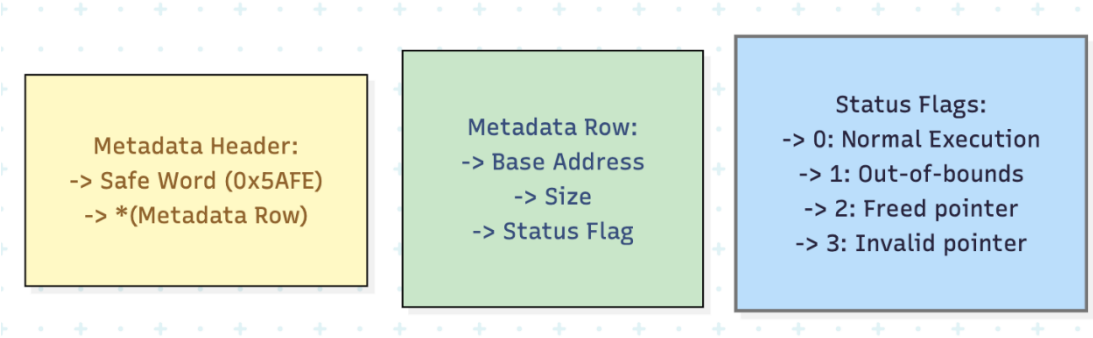

Figure 3.2: Data Structures used for Metadata

3.4 COMPILE-TIME INSTRUMENTATION

During compilation, sf-nvcc operates as an active observer and modifier of the PTX stream. Device-specific code undergoes preprocessing so each kernel's logic, parameters, and memory layout are spun off into intermediate representation. Through the NVIDIA CUDA Compiler's driver (cicc), high-level CUDA C++ is lowered into PTX a register-rich, assembly-like intermediate code that provides direct visibility into pointer manipulations, memory indexing, and operand lifecycles. Basically assembly but for Nvidia GPUs

The PTX parsing engine divides lines into tokens: directives, opcodes, operands, symbols, and literal constants. Sophisticated regular expression patterns hunt for global memory operations, with deep handling for pointer arithmetic (tracking arithmetic propagation through registers), type conversions (e.g., `cvt.s64.s32`), and moves. This systemt catches all direct accesses and also follows the trail of base kernel parameters through multiple instructions, so that every derived interior pointer is also properly guarded.

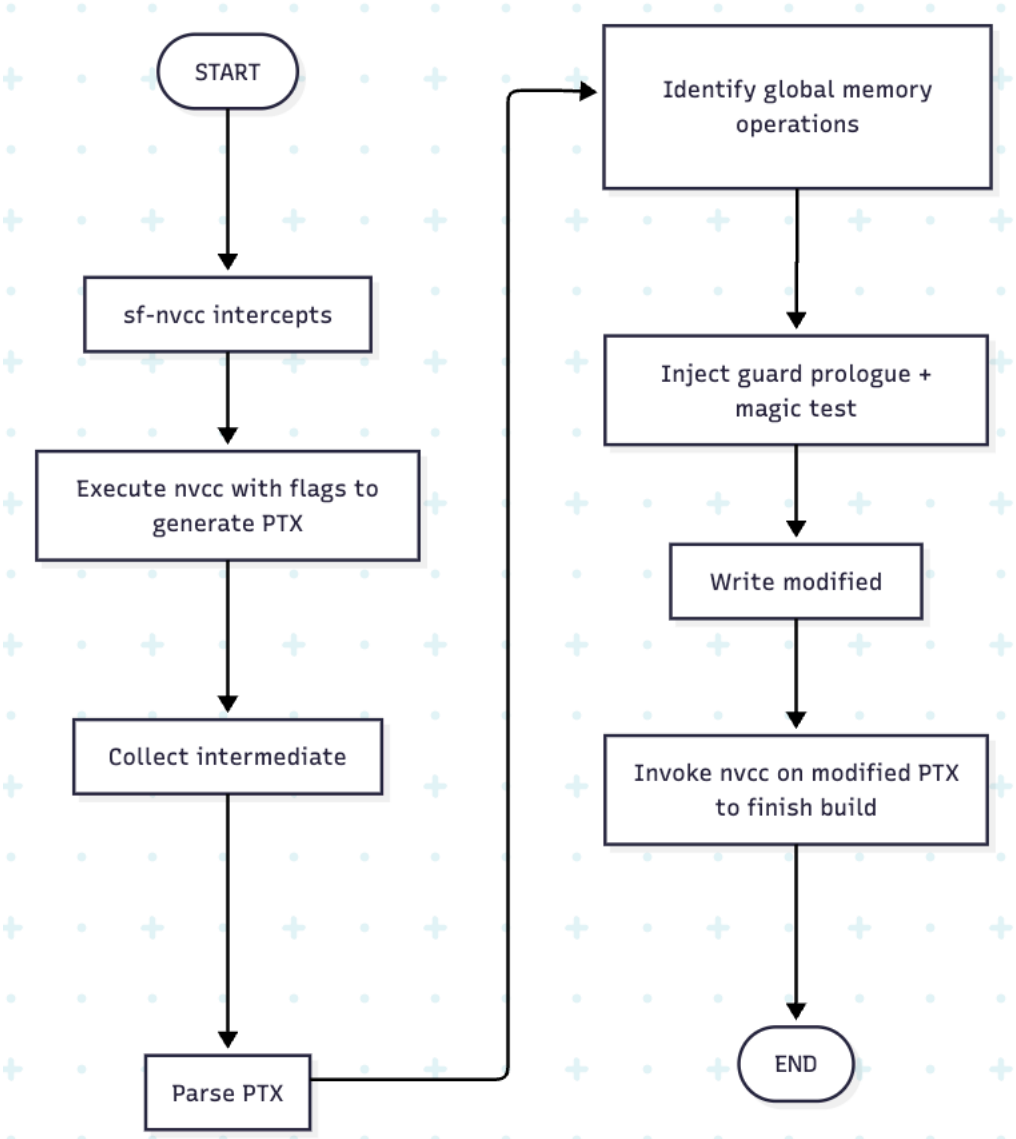

Figure 3.3: Compile-time sf-nvcc execution and PTX instrumentation

Instrumented calls to the bounds check routine are inserted before each identified global memory instruction. These calls pass the candidate address to a device function injected in every kernel, acting as a gatekeeper for legal versus illegal accesses. Critically, the logic ensures semantic preservation: original arithmetic, branching, and index calculation is left undisturbed. At the end of this stage, freshly modified PTX is fed back to the CUDA compilation process, converted to cubin, and included in the host binary.

3.5 RUNTIME OPERATION AND INTERPOSITION

Upon execution, the engineered safety mechanisms are quietly activated by setting the LD_PRELOAD environment variable to include libsafecuda.so. This shared library interposes on every relevant CUDA memory management call, most crucially cudaMalloc and cudaFree. When a new allocation is requested, the wrapper increases the size by 16 bytes, reserving a crisp metadata header at the very beginning of the allocation.

This 16-byte header is basically metadata: it includes a verifiable magic tag (`0x5AFE`), alignment padding (ensuring performance neutrality and no bugs due to misalignment), and a direct pointer to the entry in a persistent, process-lifetime metadata table. The metadata table itself is robust, storing each allocation's base, size, and flags, ready to answer queries from device or host at any moment.

When the running kernel attempts to access global device memory, every relevant instruction executes not just the operand, but also the previously-inserted bounds check. This bounds check performs a rapid scan of the metadata table, comparing the access against known base, size, and flags. Importantly, this lookup leverages the parallel nature of the GPU: threads co-operate to comb through the table without the drawbacks of sequential search or branching. If the access is valid, the routine transparently yields to the original memory operation. If not, a violation entry is flagged, the kernel is halted (according to runtime policy), and the error is surfaced at the next CUDA synchronization call, including information about the faulting address and nature of the violation.

When an allocation is freed, the metadata is promptly flagged as invalid introducing the crucial bit required to detect and block use-after-free scenarios. Rather than removing the entry, the freed state persists in the table until program exit, ensuring that any subsequent access to the same heap block (via dangling pointer or accidental memory overlap) is properly caught and reported.

CHAPTER 4
IMPLEMENTATION

4.1 INTRODUCTION

This chapter presents the implementation details for the SafeCUDA application. It focuses on the practical development of the runtime checks and data structures and compile-time instrumentations necessary to make this application a reality. The implementation transforms the conceptual system design into a functional solution capable of accurately detecting and reporting any present memory-safety vulnerabilities while maintaining the overall functionality and flow of execution the same as how it's generally done. This chapter includes complete code required to execute the final model, with minimal explanatory text, as the theoretical background has already been addressed in the earlier sections of the report.

Due to the extensive size of the complete source code, which includes the safecuda runtime, the safecache device logic, and the multi-file sf-nvcc compiler wrapper, the full source code is not included in this report's appendix. Instead, the full, verifiable source code for the SafeCUDA system is publicly hosted and available at:

GitHub Repository URL: https://www.github.com/GinOwO/SafeCUDA

This repository contains all files, including safecuda.cpp, safecache.cu, and the entire sf-nvcc toolchain, allowing for comprehensive review and validation of the implementation details. Only certain snippets of important functions are provided in the appendix

4.2 TOOLS AND LIBRARIES USED

The implementation of the proposed solution mainly requires libraries to achieve it's functionality optimally. Below is the list of all tools and libraries used:

I. CUDA & C++ - Main programming language used to build the application
II. <cuda_runtime.h> - CUDA runtime library used to access functions like

cudaMalloc and cudaFree.

III. <cuda.h> - the main CUDA library that allows us to build and execute kernels

IV. <cstddef> and <cstdint> - used for building and maintaining data types and data structures

4.3 DEVELOPMENT ENVIRONMENT SETUP

This application has been developed using C++ and the Nvidia CUDA Toolkit on Linux. Nvidia hardware used to test include a GTX 1650Ti and a RTX 3060Ti.

1. This project is currently designed to work only on Linux distributions.
2. This project requires the user to use CUDA 12.9.0/12.9.1 and gcc-13/g++-13.

4.4 FOLDER STRUCTURE

The project uses a traditional C++ project folder structure and is setup as follows:

Figure 4.1 : File Directory Structure

4.5 MAIN IMPLEMENTATION

4.5.1 SAFECUDA AND SAFECACHE HEADERS

The system's integrity relies on two main header files (code available in Appendix 1.1 and Appendix 1.2).

Code Explanation:

- “safecuda.h” is the header-file for the main driver program and is responsible for containing functions to initialise, de-initialise all memory structures and integrate runtime and compile-time functions etc.
- “safecache.cuh” is the header-file for the cuda-managed dynamic array which is used for bounds-checking memory pointer addresses not present in the main table. It is marked with the “cuh” file extension because it’s managed by cuda and can use <cuda_runtime.h> and other cuda libraries and their functions.
- The file “safecache.cuh” defines the core data structures used by the application, including the metadata table, table entries, and the enumeration of error codes that are both used internally and returned to the user. It also declares the extern allocation table, which, through the use of the “extern” keyword, ensures that the variable retains the same name across both the host and kernel sides. This approach eliminates name mangling and simplifies variable access and consistency between host and device code.

4.5.2 SAFECACHE IMPLEMENTATION

The device-side logic for memory safety validation is implemented in safecache.cu (code available in Appendix 1.3).

Code Explanation:

- This is the main driver code for the safecache, which is the global allocation and metadata table proposed to keep all the memory requests accounted for. Firstly, the global allocation table is initialised as a null pointer using the “extern” keyword to bypass name-mangling. Then, for the bounds check function, we have a void pointer as an input. The reason for the input being a void pointer

over a proper type like int or string is because it is extremely easy and efficient to cast and recast void pointers into a different type and vice versa, which means that we can input void pointers and return void pointers, which makes sure that there are no type mismatches, etc.

- The bounds check function checks for the input pointer address's existence inside the table's records. During checking, a linear search is performed instead of a binary search because of the fact that the table is managed by CUDA, which means that a linear search with an O(n) time complexity is effectively O(1) since the linear search is executed concurrently instead of sequentially.
- If the address falls into an entry and that entry's flags equals "safecuda::memory::NO_ERROR" (exact equality), the function immediately returns. This is the normal (non-error) fast-path: the pointer maps into a currently valid, non-errored allocation.
- If the pointer lies inside an entry whose flags indicate it was freed (checked with a bitwise mask in which the flag variable in the entry is compared to "safecuda::memory::ERROR_FREED_MEMORY" using a bitwise AND operation), the function records that a freed-memory access was observed by setting freed = true and saving that entry's index idx. If no matching entry is found at all through the scan, the pointer is considered out-of-bounds. The code uses a deferred / coalesced reporting mechanism: instead of immediately writing error state for every thread that discovers the problem, it uses a single reserved table entry (entries[0]) to atomically record the first observed fault and then calls __trap() to stop execution.
- To publish the error across concurrent threads safely, the code performs atomicOr(&d_table->entries[0].flags, <ERROR_BIT>). atomicOr returns the old flags value (old) so the code can detect whether this thread was the first to set that particular error bit. Only the first writer writes the additional diagnostic data:
    - For a freed-memory detection it sets entries[0].start_addr = addr and writes the freed-entry index into entries[0].block_size = idx (reusing the block_size field as an integer slot).
    - For an out-of-bounds detection it sets "entries[0].start_addr = reinterpret_cast<std::uintptr_t>(ptr)" (records offending address).

This atomic-first-writer pattern prevents races and avoids multiple threads clobbering diagnostic fields while still letting every offending thread immediately trap.

- After publishing the diagnostic to the reserved slot (or if another thread already published it), the function calls "__trap()" to cause an immediate device-side abort (useful during debugging/safety checks). This ensures that illegal memory uses do not continue executing and that a deterministic first-fault record is available in entries[0] for host-side inspection after the kernel fails.
- Using this global array, safecuda is implemented, as we will see in the next subchapter.

4.5.3 SAFECUDA IMPLEMENTATION

The host-side logic for the safecuda application is implemented in safecuda.cpp (code available in Appendix 1.4).

Code Explanation:

- The file implements SafeCUDA's host-side interception layer, which wraps key CUDA runtime APIs to add runtime memory safety without altering user code. It intercepts functions such as cudaMalloc, cudaFree, cudaLaunchKernel, and cudaDeviceSynchronize to insert additional checks and metadata tracking.
- SafeCUDA uses dynamic linking interposition via dlsym(RTLD_NEXT) to retrieve the original CUDA function addresses. This allows it to redefine these functions under the same names while still calling the real CUDA implementations internally, enabling transparent integration through LD_PRELOAD.
- During initialization, it allocates both a host-side allocation table ("h_table") and a device-side counterpart **("**d_table_ptr"**)**. These tables track every allocated GPU memory block's base address, size, and state flags. The host table is pinned (cudaHostAllocMapped) for efficient synchronization with the device, and the device table allows kernel-level bounds checking.
- For every allocation (cudaMalloc or cudaMallocManaged), SafeCUDA adds 16 bytes of metadata before the actual memory region. This metadata includes a

magic identifier and a pointer to the corresponding entry in the allocation table, creating a link between user pointers and their tracking records.

- On deallocation (cudaFree), SafeCUDA reads the metadata, validates the allocation, and updates the entry's flag to ERROR_FREED_MEMORY, marking the block as invalid. This state is propagated to the device so that subsequent GPU accesses to freed memory trigger traps and errors.
- The error-checking mechanism relies on a reserved table slot (entries[0]) where the GPU reports the first detected fault (out-of-bounds, freed memory, or invalid pointer). After each kernel execution or synchronization call, SafeCUDA checks this slot, prints an appropriate diagnostic message, and raises an exception to halt execution deterministically.
- When launching kernels, SafeCUDA modifies the argument list to inject a pointer to the device allocation table (d_table_ptr) as the first parameter. This enables GPU-side code to perform in-kernel validation of pointers through the same global table, bridging host and device memory tracking seamlessly.
- Overall, this interposition layer forms the core of SafeCUDA's memory safety system, combining lightweight metadata tracking with transparent host–device synchronization. It offers a Valgrind-like mechanism for GPUs, detecting memory violations such as out-of-bounds access and use-after-free with minimal changes to user workflows.

4.6 SF-NVCC IMPLEMENTATION

The implementation details for the sf-nvcc compiler wrapper are available in Appendix 1.5, Appendix 1.6 and Appendix 1.7.

4.6.1 INTRODUCTION AND FLAGS

sf-nvcc is a wrapper of nvcc, the CUDA compiler made by NVIDIA, used for enabling safecuda functionality but also keeping the core functionality and usage methods of nvcc the same. To achieve this, developers only need to make one change to the general build script, and that is to use sf-nvcc instead of nvcc. sf-nvcc also adds some extra functionality to help developers debug outputs better etc.

Flags that can be added to the sf-nvcc build command include:

1. -sf-help : shows the help message and other details about sf-nvcc
2. -sf-version: displays the sf-nvcc version and applicable SafeCUDA build info.
3. -sf-debug: can assume a value of either true or false (default: false). This flag enables the debug mode which provides detailed PTX modification logs, which instructions are being instrumented and metadata operations, etc. If this flag is enabled "-sf-verbose" is also automatically enabled.
4. -sf-verbose: can assume a value of either true or false (default: false). This flag enables verbose compilation output.
5. -sf-fail-fast: can assume a value of either true or false (default: true). This flag makes the kernel abort execution as soon as the first bounds violation is encountered. If this flag has a value false, then kernel execution continues even though bounds violations are encountered, and they are logged.
6. -sf-keep-dir: This flag requires a path value as a parameter. It stores the intermediate files in a path specified by the user instead of the default directory (system temporary /tmp directory).

CHAPTER 5

RESULTS AND ANALYSIS

5.1 INTRODUCTION

This chapter provides a comprehensive analysis of the expected and observed results of the performance of the solution. The purpose of this chapter is to assess the performance, reliability, and robustness of the solution with respect to tests that are both realistic and synthetic to properly gauge it's capabilities and limitations. The performance of safecuda is evaluated according to mean and median runtimes and is compared with runtimes of "nvcc" without safecuda for an accurate comparison benchmark and to obtain proper estimates.

A sample PTX file from before and after modification by sf-nvcc are included in the Appendix 2.

5.2 PERFORMANCE TESTING AND BENCHMARKING

To gauge the real-world impact and resilience of SafeCUDA, a rigorous performance testing regime was instituted. The benchmarking suite was implemented in Python for modularity and reproducibility, orchestrates dual builds: one using vanilla nvcc, the other using sf-nvcc instrumentation. Each build path targets a spectrum of test kernels, meticulously chosen to represent both familiar and edge-case usage scenarios.

Benchmarks encompass:

1. Large vector operations, emulating scientific workloads with contiguous memory strides
2. Parallel sum reduction, capturing kernel patterns ubiquitous in HPC
3. Memory copy and scaling, reflecting real-world ML preprocessing pipelines
4. Realistic high-compute kernels, typically found in deep learning training and scientific simulation
5. Synthetic memory stress test (maximizing global memory traffic)
6. Synthetic multi-allocation stress test (fragmented, sparse allocation patterns)

Each test is subject to a series of warm-up runs, discarded to mitigate initialization bias, followed by 500 precise, measured iterations to build robust averages and percentile statistics. The suite is designed to probe aggregate overhead, distribution and latency outliers, and ensure that the safety machinery performs under both normal and artificial conditions. Through these tests, SafeCUDA's efficacy its ability to add minimal runtime cost in authentic workloads while flagging all categories of violations aids in statistically validating it's efficiency and lower overhead.

#	Tool	Mean (ms)	Median (ms)	p_low (ms)	p_high (ms)	Tmin (ms)	Tmax (ms)	Overhead %
1	nvcc	300.749	300.037	282.261	322.302	265.386	358.500	1.613
	sfnvcc	305.601	304.967	286.635	326.219	274.762	353.142	1.613
2	nvcc	303.903	302.927	285.911	326.853	274.559	381.088	0.765
	sfnvcc	306.229	305.003	288.196	327.579	279.251	352.858	0.765
3	nvcc	303.066	302.582	281.770	326.216	269.097	340.626	4.568
	sfnvcc	316.910	316.042	296.699	337.887	285.498	367.778	4.568
4	nvcc	327.359	327.188	306.121	351.574	290.211	364.239	0.737
	sfnvcc	329.771	328.946	307.766	354.655	299.902	373.471	0.737
5	nvcc	312.259	310.644	292.855	336.859	281.209	379.072	11.068
	sfnvcc	346.821	345.683	321.741	373.270	309.254	456.770	11.068
6	nvcc	355.931	351.681	330.349	401.468	311.496	427.474	17.419
	sfnvcc	417.930	418.180	393.012	444.448	382.542	467.195	17.419

Table 5.1: Performance summary statistics

Figures 5.1 and 5.2 compares the average execution times (in milliseconds) of six benchmark kernels (perf1–perf6) compiled using the standard nvcc compiler and the instrumented sf-nvcc compiler. The blue bars represent nvcc results, while the orange bars show sf-nvcc, with error bars denoting run-to-run variability. The red line illustrates the runtime overhead (%) introduced by sf-nvcc relative to nvcc, plotted on the secondary y-axis. Across the benchmarks, sf-nvcc shows slightly higher execution times due to the insertion of safety instrumentation and runtime checks. The overhead remains minimal (below ~5%) for real world adjacent workloads but increases gradually for synthetic artificial workloads, reaching around 15–17% in the final tests. This indicates that while sf-nvcc introduces a measurable performance cost, it achieves significantly improved memory safety diagnostics with only moderate

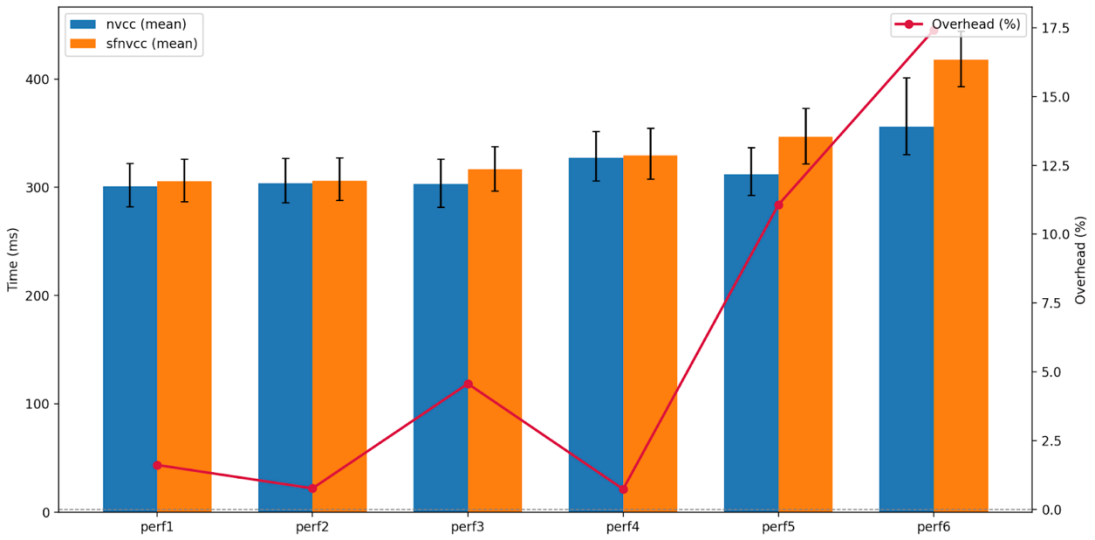

Figure 5.1: Bar plot depicting the performance test results

impact on overall execution time and that performance is hampered more by number of allocations than reads/writes which is less of an issue in the real world as the number of cuda allocations in real world paradigms are usually significantly lower than what we have tested with a measurable performance cost, it achieves significantly improved memory safety diagnostics with only moderate impact on overall execution time.

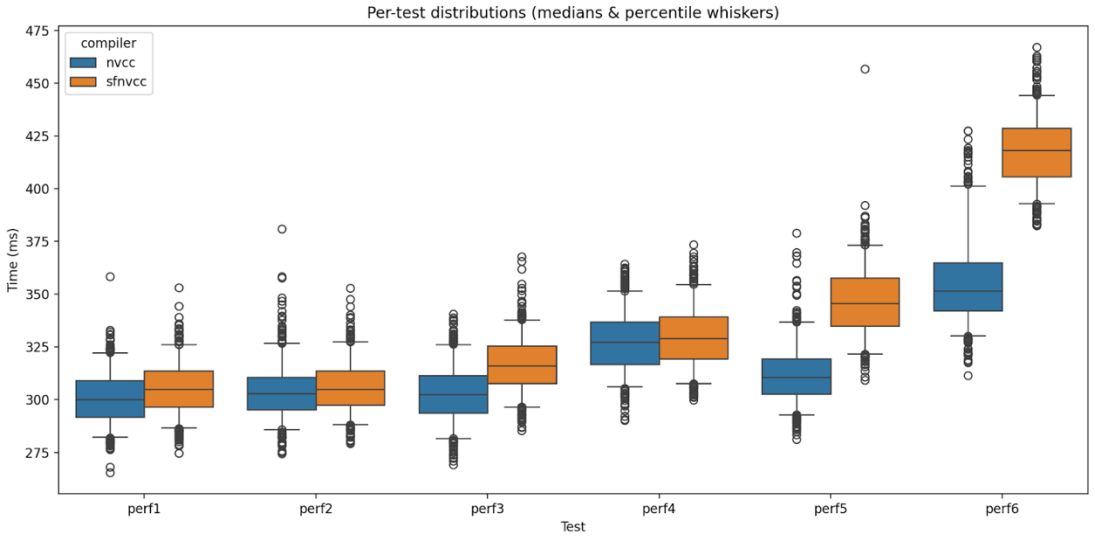

Figure 5.2: Box plot depicting the performance test results

5.3 COMPLEXITY ANALYSIS

Both time and space complexity are theoretically O(N), however practically they are negligible.

On the time side, this is because the search algorithm is implemented as a linear search running on the GPU on multiple threads in parallel meaning that this would be much faster than on a traditional CPU. The rationale behind using a simple linear search over a binary search is that it has less conditional branching statements which can severely impact performance on GPU applications.

On the memory side, this is because the maximum possible entries in the table is a generous fixed number of 1023. This means that the maximum possible memory used from all entries, metadata and the table itself is a mere 48kB.

CHAPTER-6
CONCLUSION AND FUTURE WORK

6.1 CONCLUSION

The challenge of ensuring robust memory safety in the massively parallel NVIDIA CUDA ecosystem, remains a critical and complex issue. Errors like buffer overflows, use-after-free, and invalid pointer accesses can lead to silent data corruption or severe security vulnerabilities, yet existing solutions often mandate hardware modifications or incur prohibitive runtime costs.

This project successfully addressed this gap by developing SafeCUDA, a novel, purely software-only memory safety enforcement system for CUDA applications. By engineering a novel software-only, two-tier validation system (compile-time instrumentation and runtime interposition), we demonstrated that robust memory safety guarantees for GPU applications are feasible without requiring changes to source code, specialized hardware, or driver modifications.

It is capable of detecting critical memory safety CVEs such as OOB accesses, use-after-free, and double-free within GPU device code. The success of the adopted approach using PTX intermediate representation instrumentation via the sf-nvcc wrapper was validated by its ability to transparently inject bounds-checking calls while maintaining the original compilation sequence. The complementary runtime interposition layer effectively bridges the decoupled host and device memory spaces by using a managed shadow metadata cache, enabling synchronized pointer tracking and deterministic error reporting at synchronization points.

6.2 SUMMARY OF RESULTS

The experimental validation in Chapter 5 demonstrated that SafeCUDA offers effective memory safety with a justifiable performance overhead.

SafeCUDA successfully identified and reported all tested memory-safety violations, confirming its role as a robust debugging tool. The atomic-first-writer pattern implemented in the safecache logic ensures deterministic error reporting from the device to the host upon synchronization.

For general and high-compute workloads (Perf1 through Perf4), the runtime overhead introduced by SafeCUDA's instrumentation remained low, under 5%. This confirms that the parallel nature of the device-side linear search for metadata is highly efficient. For synthetic stress tests involving extensive global memory traffic and high allocation fragmentation (Perf5 and Perf6), the overhead increased but remained within an acceptable range for a memory safety diagnostic tool, peaking at approximately 17%.

6.3 LIMITATIONS

While successful, the project encountered specific limitations that bound the current scope:

1. CUDA Driver Interactions: The use of certain standard CUDA functions, specifically device functions within the curand library, can trigger a SIGSEGV (Segmentation Fault) on the device. This occurs due to how nullptrs are utilized internally within curand to set random seeds. A different approach or wrapper for seed initialization is necessary to circumvent this specific driver-level issue.
2. Architectural Compatibility: The current build was developed and thoroughly tested only on the NVIDIA GeForce GTX 1650 Ti (SM 7.5) . Proper operation on other GPU architectures may require additional testing and configuration.
3. Performance Ceiling: Although performance is favorable in realistic scenarios, very large, non-realistic synthetic tests currently cause the execution time to double (100% overhead). This pathological case indicates that optimization is needed for extreme memory-intensive edge cases, though it is unlikely to be encountered in standard high-performance computing (HPC) applications.

6.4 COMPARISON TO EXISTING WORK

Currently existing solutions include cuCatch [11] and GPUArmour [12]. cuCatch [11], while not requiring any source code modifications or specialised hardware, presents an overhead of ~17%. On the other hand, GPUArmour [12] presents an overhead of ~2.5% but with the added condition that it requires specialised hardware to operate. Our proposed solution, after rigorous testing presents an overhead of ~11%.

SafeCUDA can be most closely compared with cuCatch, because like cuCatch, it operates as a drop-in replacement, and requires no additional specialised hardware.

6.5 FUTURE WORK

1. Performance and Optimization:
    a. Micro-Optimized Bounds Checking: Develop a better micro-optimized bounds checking algorithm. While the concurrent linear search is effective, future research could explore more advanced index structures or managed metadata caching strategies to reduce the reliance on repeated global memory lookups, potentially lowering overhead across all tests.
    b. Advanced PTX Optimization Passes: Further research into advanced PTX optimization passes could minimize the inherent runtime cost. Specifically, passes could be designed to aggressively reduce the register pressure and branch divergence caused by the injected guard instructions, aiming to push the overhead for memory-intensive workloads below the currently observed 15–17% level.
    c. Architecture Specific Optimization: Future research could develop better low level architecture specific optimizations to further improve performance.
2. Expanded Coverage and Functionality:
    a. Support for Other Memory Spaces: While the current system focuses on global heap memory, future work must expand coverage to automatically instrument and track other CUDA memory spaces, including private, shared memory, and host-managed memory types such as CUDA array, host/pinned memory, and texture memory.
    b. Multi-GPU Support: Integrate support for Multi-GPU applications, requiring a coordinated mechanism to manage and synchronize the shadow metadata cache across different devices and potentially different hosts.

In conclusion, SafeCUDA provides a valuable contribution to the GPU software ecosystem. It bridges the historical gap between high-performance parallel

programming and rigorous memory safety guarantees by delivering a comprehensive, non-intrusive safety solution. This enables researchers and developers to build more reliable and secure CUDA applications without specialized hardware investment.

REFERENCES

[1] NVIDIA Corporation, CUDA Toolkit Documentation, 2024. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[2] Andrea Miele. 2016. Buffer overflow vulnerabilities in CUDA: a preliminary analysis. Journal of Computer Virology and Hacking Techniques 12, 2 (2016), 113–120.

[3] Park, Sang-Ok & Kwon, Ohmin & Kim, Yonggon & Cha, Sang & Yoon, Hyunsoo. (2020). Mind Control Attack: Undermining Deep Learning with GPU Memory Exploitation. Computers & Security. 102. 102115. 10.1016/j.cose.2020.102115.

[4] Arauzo, Xabier & Yarza, Irune & Kosmidis, Leonidas & Calderón, Jhosue & Rodriguez, Marcos. (2023). Unraveling the Mystery of NVIDIA's Unified Memory for Safety-Critical GPU Systems. 366-372. 10.1109/DSD60849.2023.00058.

[5] Gururaj Saileshwar, Rick Boivie, Tong Chen, Benjamin Segal, and Alper Buyuktosunoglu. 2022. HeapCheck: Low-cost Hardware Support for Memory Safety. ACM Trans. Archit. Code Optim. 19, 1, Article 10 (March 2022), 24 pages. https://doi.org/10.1145/3495152

[6] [24] Bruening, Derek & Zhao, Qin. (2011). Practical memory checking with Dr. Memory. 213 - 223. 10.1109/CGO.2011.5764689.

[7] Stepanov, Evgeniy & Serebryany, Konstantin. (2015). MemorySanitizer: Fast detector of uninitialized memory use in C++. 46-55. 10.1109/CGO.2015.7054186.

[8] Chen, T., & Chisnall, D. (2019, July). Pointer tagging for memory safety (Technical Report No. MSR-TR-2019-17). Microsoft Research. https://www.microsoft.com/en-us/research/publication/pointer-tagging-for-memory-safety/

[9] Dudina, I., Stark, I. Static analysis to make the most of CHERI C/C++ for existing code: improving memory safety at scale. Int J Softw Tools Technol Transfer 27, 225–237 (2025). https://doi.org/10.1007/s10009-025-00781-6

[10] Moghadam, Vahid & Serra, Gabriele & Aromolo, Federico & Buttazzo, Giorgio & Prinetto, Paolo. (2024). Memory Integrity Techniques for Memory-Unsafe Languages: A Survey. IEEE Access. PP. 1-1. 10.1109/ACCESS.2024.3380478.

[11] Ziad, Mohamed & Damani, Sana & Jaleel, Aamer & Keckler, Stephen & Stephenson, Mark. (2023). cuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications. Proceedings of the ACM on Programming Languages. 7. 124-147. 10.1145/3591225.

[12] Tarek Ibn Ziad, M., Damani, S., Stephenson, M., Keckler, S. W., and Jaleel, A., “GPUArmor: A Hardware-Software Co-design for Efficient and Scalable Memory Safety on GPUs”, arXiv, Art. no. arXiv:2502.17780, 2025. doi:10.48550/arXiv.2502.17780.

[13] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. 2022. Securing GPU via region-based bounds checking. In ISCA ’22: Proceedings of the 49th Annual International Symposium on Computer Architecture. New York, NY, USA, 27–41. https://doi.org/10.1145/3470496.3527420

[14] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07). Association for Computing Machinery, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[15] Baumann, Thomas & Gracia, Jose. (2013). Cudagrind: A valgrind extension for CUDA. Advances in Parallel Computing. 25. 10.3233/978-1-61499-381-0-763.

[16] Köpcke, Bastian & Gorlatch, Sergei & Steuwer, Michel. (2024). Descend: A Safe GPU Systems Programming Language. Proceedings of the ACM on Programming Languages. 8. 841-864. 10.1145/3656411.

[17] K. Huang, M. Payer, Z. Qian, J. Sampson, G. Tan and T. Jaeger, "Comprehensive Memory Safety Validation: An Alternative Approach to Memory Safety," in IEEE

Security & Privacy, vol. 22, no. 4, pp. 40-49, July-Aug. 2024, doi: 10.1109/MSEC.2024.3379947

[18] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC'12). USENIX Association, USA, 28

[19] Yuseok Jeon, Wookhyun Han, Nathan Burow, and Mathias Payer. 2020. FuZZan: efficient sanitizer metadata design for fuzzing. In Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20). USENIX Association, USA, Article 17, 249–263.

[20] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. ACM Trans. Program. Lang. Syst. 27, 3 (May 2005), 477–526. https://doi.org/10.1145/1065887.1065892

[21] Yaning Yang, Xiaoqi Wang, Shaoliang Peng. A Dynamic Protection Mechanism for GPU Memory Overflow. 17th IFIP International Conference on Network and Parallel Computing (NPC), Sep 2020, Zhengzhou, China. pp.30-40, ff10.1007/978-3-030-79478-1_3ff. ffhal-03768732f

[22] J. Lee et al., "Let-Me-In: (Still) Employing In-pointer Bounds Metadata for Fine-grained GPU Memory Safety," 2025 IEEE International Symposium on High Performance Computer Architecture (HPCA), Las Vegas, NV, USA, 2025, pp. 1648-1661, doi: 10.1109/HPCA61900.2025.00122.

[23] Calderón, Jhosue & Kosmidis, Leonidas & Nicolas, Carlos & Cazorla, Francisco. (2024). XeroZerox: Analysis and Optimization of GPU Memory Management for High-Integrity Autonomous Systems. IEEE Access. PP. 1-1. 10.1109/ACCESS.2024.3406893.

[24] Henriksen, Troels. (2021). Bounds Checking on GPU. International Journal of Parallel Programming. 49. 10.1007/s10766-021-00703-4.

[25] Z. Chen, Q. Zhang, J. Wu, J. Yan and J. Xue, "A Source-Level Instrumentation Framework for the Dynamic Analysis of Memory Safety," in IEEE Transactions on Software Engineering, vol. 49, no. 4, pp. 2107-2127, 1 April 2023, doi: 10.1109/TSE.2022.3210580.

[26] Bang Di, Jianhua Sun, Hao Chen, and Dong Li. 2021. Efficient buffer overflow detection on GPU. IEEE Transactions on Parallel Distributed Systems 32, 5 (2021), 1161–1177.

APPENDIX 1

SOURCE CODE

1. SafeCUDA/include/safecuda.h:

```
#ifndef SAFECUDA_H
#define SAFECUDA_H
#include <cuda_runtime.h>
#include <cuda.h>
#include <cstddef>
namespace safecuda
{
using cudaMalloc_t = cudaError_t (*)(void **, std::size_t);
using cudaMallocManaged_t = cudaError_t (*)(void **, std::size_t, unsigned int);
using cudaFree_t = cudaError_t (*)(void *);
using cudaDeviceSynchronize_t = cudaError_t (*)();
using cudaGetLastError_t = cudaError_t (*)();
using cudaConfigureCall_t = cudaError_t (*)(dim3, dim3, size_t, cudaStream_t);
using cudaSetupArgument_t = cudaError_t (*)(const void *, size_t, size_t);
using cudaLaunch_t = cudaError_t (*)(const void *);
using cuLaunchKernel_t = CUresult (*)(CUfunction, unsigned int, unsigned int,
                                      unsigned int, unsigned int, unsigned int,
                                      unsigned int, unsigned int, CUstream,
                                      void **, void **);
using cudaLaunchKernel_t = cudaError_t (*)(const void *func, dim3 gridDim,
                                           dim3 blockDim, void **args,
                                           size_t sharedMem,
                                           cudaStream_t stream);
extern cudaMalloc_t real_cudaMalloc;
extern cudaMallocManaged_t real_cudaMallocManaged;
extern cudaFree_t real_cudaFree;
extern cudaDeviceSynchronize_t real_cudaDeviceSynchronize;
extern cudaGetLastError_t real_cudaGetLastError;
extern cudaLaunchKernel_t real_cudaLaunchKernel;
```

```
void init_safecuda();
void shutdown_safecuda();
}
#endif // SAFECUDA_H
```

2. SafeCUDA/include/safecache.cuh

```
#ifndef SAFECACHE_H
#define SAFECACHE_H
#include <cuda_runtime.h>
#include <cstdint>
namespace safecuda::memory
{
struct Entry {
        std::uintptr_t start_addr;
        std::uint32_t block_size;
        std::uint32_t flags;
        std::uint32_t __reserved;
};
struct Metadata {
        std::uint16_t magic;
        std::uint8_t padding[6];
        Entry *entry;
};
struct AllocationTable {
        Entry *entries;
        std::uint32_t count;
        std::uint32_t capacity;
        std::uint32_t __reserved;
};
enum ErrorCode {
        NO_ERROR = 0,
        ERROR_OUT_OF_BOUNDS = 1 << 0,
        ERROR_FREED_MEMORY = 1 << 1,
        ERROR_INVALID_POINTER = 1 << 2
};
}
extern "C" __device__ safecuda::memory::AllocationTable *d_table;
extern "C" __device__ void __bounds_check_safecuda(void *ptr);
```

```
#endif
```

3. SafeCUDA/src/safecache.cu:

```
#include "safecache.cuh"
extern "C" __device__ safecuda::memory::AllocationTable *d_table = nullptr;
__device__ void __bounds_check_safecuda(void *ptr)
{
        const auto addr = reinterpret_cast<std::uintptr_t>(ptr);
        bool freed = false;
        std::int32_t idx = -1;
        for (std::uint32_t i = 1; i < d_table->count; ++i) {
                safecuda::memory::Entry *entry = &d_table->entries[i];
                if (entry->start_addr <= addr &&
                   addr < entry->start_addr + entry->block_size) {
                        // if valid just return
                        if (entry->flags == safecuda::memory::NO_ERROR)
                                return;
                        // we need a deferred mechanism for freed mem here
                        if (entry->flags &
                           safecuda::memory::ERROR_FREED_MEMORY) {
                                freed = true;
                                idx = i;
                        }
                }
        }
        if (freed) {
                const auto old = atomicOr(&d_table->entries[0].flags,

safecuda::memory::ERROR_FREED_MEMORY);
                if ((old & safecuda::memory::ERROR_FREED_MEMORY) == 0)
{
                        d_table->entries[0].start_addr = addr;
                        d_table->entries[0].block_size = idx;
                }
                __trap();
```

```
                return;
        }
        const auto old = atomicOr(&d_table->entries[0].flags,

safecuda::memory::ERROR_OUT_OF_BOUNDS);
        if ((old & safecuda::memory::ERROR_OUT_OF_BOUNDS) == 0) {
                d_table->entries[0].start_addr =
                        reinterpret_cast<std::uintptr_t>(ptr);
        }
        __trap();
}

__device__ void __bounds_check_safecuda_no_trap(void *ptr)
{
        const auto addr = reinterpret_cast<std::uintptr_t>(ptr);
        bool freed = false;
        std::int32_t idx = -1;
        for (std::uint32_t i = 1; i < d_table->count; ++i) {
                safecuda::memory::Entry *entry = &d_table->entries[i];
                if (entry->start_addr <= addr &&
                   addr < entry->start_addr + entry->block_size) {
                        // if valid just return
                        if (entry->flags == safecuda::memory::NO_ERROR)
                                return;
                        // we need a deferred mechanism for freed mem here
                        if (entry->flags &
                           safecuda::memory::ERROR_FREED_MEMORY) {
                                freed = true;
                                idx = i;
                        }
                }
        }
        if (freed) {
                const auto old = atomicOr(&d_table->entries[0].flags,
```

```
safecuda::memory::ERROR_FREED_MEMORY);
                    if ((old & safecuda::memory::ERROR_FREED_MEMORY) == 0)
{
                            d_table->entries[0].start_addr = addr;
                            d_table->entries[0].block_size = idx;
                    }
                    return;
            }
            const auto old = atomicOr(&d_table->entries[0].flags,

safecuda::memory::ERROR_OUT_OF_BOUNDS);
            if ((old & safecuda::memory::ERROR_OUT_OF_BOUNDS) == 0) {
                    d_table->entries[0].start_addr =
                            reinterpret_cast<std::uintptr_t>(ptr);
            }
}
```

4. SafeCUDA/src/safecuda.cpp:

```
#include "safecuda.h"
#include "safecache.cuh"
#include <cuda_runtime.h>
#include <cstdint>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <dlfcn.h>
#include <stdexcept>
static safecuda::memory::AllocationTable *h_table = nullptr;
static safecuda::memory::AllocationTable *d_table_ptr = nullptr;
namespace safecuda
{
constexpr uint32_t TABLE_ENTRIES = 1024;
cudaMalloc_t real_cudaMalloc = nullptr;
cudaMallocManaged_t real_cudaMallocManaged = nullptr;
cudaFree_t real_cudaFree = nullptr;
cudaDeviceSynchronize_t real_cudaDeviceSynchronize = nullptr;
cudaGetLastError_t real_cudaGetLastError = nullptr;
cudaLaunchKernel_t real_cudaLaunchKernel = nullptr;
static void sync_table_to_device()
{
        constexpr size_t entry_bytes = TABLE_ENTRIES * sizeof(memory::Entry);
        constexpr size_t table_bytes =
                sizeof(memory::AllocationTable) + entry_bytes;
        cudaMemcpy(d_table_ptr, h_table, table_bytes,
cudaMemcpyHostToDevice);
}
void init_safecuda()
{
        real_cudaMalloc =
```

```
                reinterpret_cast<cudaMalloc_t>(dlsym(RTLD_NEXT,
"cudaMalloc"));
        real_cudaMallocManaged = reinterpret_cast<cudaMallocManaged_t>(
                dlsym(RTLD_NEXT, "cudaMallocManaged"));
        real_cudaFree =
                reinterpret_cast<cudaFree_t>(dlsym(RTLD_NEXT, "cudaFree"));
        real_cudaDeviceSynchronize =
reinterpret_cast<cudaDeviceSynchronize_t>(
                dlsym(RTLD_NEXT, "cudaDeviceSynchronize"));
        real_cudaGetLastError = reinterpret_cast<cudaGetLastError_t>(
                dlsym(RTLD_NEXT, "cudaGetLastError"));
        real_cudaLaunchKernel = reinterpret_cast<cudaLaunchKernel_t>(
                dlsym(RTLD_NEXT, "cudaLaunchKernel"));
        constexpr size_t entry_bytes = TABLE_ENTRIES * sizeof(memory::Entry);
        constexpr size_t table_bytes =
                sizeof(memory::AllocationTable) + entry_bytes;
        cudaHostAlloc(&h_table, table_bytes, cudaHostAllocMapped);
        if (!h_table) {
                printf("Failed to allocate to h_table\n");
                exit(1);
        }
        h_table->entries = reinterpret_cast<memory::Entry *>(h_table + 1);
        h_table->count = 1;
        h_table->capacity = TABLE_ENTRIES;
        std::memset(h_table->entries, 0, entry_bytes);
        void *ptr = nullptr;
        real_cudaMalloc(&ptr, table_bytes);
        d_table_ptr = static_cast<memory::AllocationTable *>(ptr);
        cudaMemcpy(d_table_ptr, h_table, table_bytes,
cudaMemcpyHostToDevice);
        real_cudaDeviceSynchronize();
}
void shutdown_safecuda()
{
```

```
        if (d_table_ptr)
                cudaFree(d_table_ptr);
        if (h_table)
                cudaFreeHost(h_table);
        d_table_ptr = nullptr;
        h_table = nullptr;
}
void check_and_report_errors()
{
        if (h_table->entries[0].flags == memory::NO_ERROR)
                return;
        const std::uintptr_t addr = h_table->entries[0].start_addr;
        std::uint32_t code = h_table->entries[0].flags;
        if (code & memory::ERROR_OUT_OF_BOUNDS) {
                char addr_buf[32];
                std::snprintf(addr_buf, sizeof(addr_buf), "0x%lx", addr);
                const std::string error_msg =
                        std::string("[SafeCUDA] Out-of-bounds access at ") +
                        addr_buf + " (code=0x" + std::to_string(code) + ")";

                std::fprintf(stderr, "%s\n", error_msg.c_str());
                throw std::runtime_error(error_msg);
        }
        if (code & memory::ERROR_FREED_MEMORY) {
                char addr_buf[32];
                std::snprintf(addr_buf, sizeof(addr_buf), "0x%lx", addr);
                const std::string error_msg =
                        std::string("[SafeCUDA] Use-after-free at ") +
                        addr_buf + " (code=0x" + std::to_string(code) + ")";
                std::fprintf(stderr, "%s\n", error_msg.c_str());
                throw std::runtime_error(error_msg);
        }
        if (code & memory::ERROR_INVALID_POINTER) {
                char addr_buf[32];
```

```
                std::snprintf(addr_buf, sizeof(addr_buf), "0x%lx", addr);
                const std::string error_msg =
                        std::string("[SafeCUDA] Invalid pointer access at ") +
                        addr_buf + " (code=0x" + std::to_string(code) + ")";
                std::fprintf(stderr, "%s\n", error_msg.c_str());
                throw std::runtime_error(error_msg);
        }
        h_table->entries[0].flags = 0;
        h_table->entries[0].start_addr = 0;
}
} // end of namespace safecuda
extern "C" cudaError_t cudaMalloc(void **devPtr, const std::size_t size)
{
        if (!safecuda::real_cudaMalloc)
                safecuda::init_safecuda();
        void *base = nullptr;
        cudaError_t err = safecuda::real_cudaMalloc(&base, size + 16);
        if (err != cudaSuccess) {
                std::fprintf(stderr, "[SafeCUDA] cudaMalloc failed: %s\n",
                                cudaGetErrorString(err));
                return err;
        }
        if (h_table->count >= safecuda::TABLE_ENTRIES) {
                safecuda::real_cudaFree(base);
                std::fprintf(stderr, "[SafeCUDA] Allocation table full\n");
                return cudaErrorMemoryAllocation;
        }
        void *user_ptr = static_cast<char *>(base) + 16;
        safecuda::memory::Entry *entry = &h_table->entries[h_table->count++];
        entry->start_addr = reinterpret_cast<std::uintptr_t>(user_ptr);
        entry->block_size = size;
        entry->flags = safecuda::memory::NO_ERROR;
        const safecuda::memory::Metadata meta = {0x5AFE, {0}, entry};
        cudaMemcpy(base, &meta, 16, cudaMemcpyHostToDevice);
```

```
        safecuda::sync_table_to_device();
        *devPtr = user_ptr;
        return cudaSuccess;
}
extern "C" cudaError_t cudaMallocManaged(void **devPtr, const std::size_t size,
                                         const unsigned int flags)
{
        if (!safecuda::real_cudaMallocManaged)
                safecuda::init_safecuda();
        void *base = nullptr;
        const cudaError_t err =
                safecuda::real_cudaMallocManaged(&base, size + 16, flags);
        if (err != cudaSuccess) {
                std::fprintf(stderr,
                        "[SafeCUDA] cudaMallocManaged failed: %s\n",
                        cudaGetErrorString(err));
                return err;
        }
        if (h_table->count >= 1024) {
                safecuda::real_cudaFree(base);
                std::fprintf(stderr, "[SafeCUDA] Allocation table full\n");
                return cudaErrorMemoryAllocation;
        }
        void *user_ptr = static_cast<std::int8_t *>(base) + 16;
        safecuda::memory::Entry *entry = &h_table->entries[h_table->count++];
        entry->start_addr = reinterpret_cast<std::uintptr_t>(user_ptr);
        entry->block_size = size;
        entry->flags = safecuda::memory::NO_ERROR;
        const safecuda::memory::Metadata meta = {0x5AFE, {0}, entry};
        cudaMemcpy(base, &meta, 16, cudaMemcpyHostToDevice);
        safecuda::sync_table_to_device();
        *devPtr = user_ptr;
        return cudaSuccess;
}
```

```
extern "C" cudaError_t cudaFree(void *devPtr)
{
	if (!devPtr)
		return cudaSuccess;

	void *base = static_cast<std::int8_t *>(devPtr) - 16;
	safecuda::memory::Metadata meta{};
	cudaMemcpy(&meta, base, 16, cudaMemcpyDeviceToHost);
	if (meta.magic == 0x5AFE && meta.entry)
		meta.entry->flags |=
safecuda::memory::ERROR_FREED_MEMORY;
	safecuda::sync_table_to_device();
	return safecuda::real_cudaFree(base);
}
extern "C" cudaError_t cudaDeviceSynchronize()
{
	if (!safecuda::real_cudaDeviceSynchronize)
		safecuda::init_safecuda();
	const cudaError_t err = safecuda::real_cudaDeviceSynchronize();
	safecuda::check_and_report_errors();
	return err;
}
extern "C" cudaError_t cudaGetLastError()
{
	if (!safecuda::real_cudaGetLastError)
		safecuda::init_safecuda();
	safecuda::check_and_report_errors();
	return safecuda::real_cudaGetLastError();
}
extern "C" cudaError_t cudaLaunchKernel(const void *func, dim3 gridDim,
                                        dim3 blockDim, void **args,
                                        size_t sharedMem, cudaStream_t
stream)
{
```

```
        if (!safecuda::real_cudaLaunchKernel)
                safecuda::init_safecuda();

        constexpr int numParams = 6;
        constexpr size_t size = (numParams + 2) * sizeof(void *);
        void **newParams = static_cast<void **>(alloca(size));
        newParams[0] = &d_table_ptr;
        newParams[1] = args[0];
        newParams[2] = args[1];
        newParams[3] = args[2];
        newParams[4] = args[3];
        newParams[5] = args[4];
        newParams[6] = args[5];
        newParams[numParams] = nullptr;
        return safecuda::real_cudaLaunchKernel(func, gridDim, blockDim,
                                                newParams, sharedMem, stream);
}
```

5. SafeCUDA/tools/sf-nvcc/src/main.cpp:

```
#include "nvcc_wrapper.h"
#include "sf_options.h"
#include <iostream>
#include <stdexcept>
#include <unordered_map>
using namespace safecuda;
int main(const int argc, char *argv[])
{
        try {
                tools::sf_nvcc::SfNvccOptions sf_nvcc_options =
                        tools::sf_nvcc::parse_command_line(argc, argv);
                auto [safecuda_opts, nvcc_args] = sf_nvcc_options;
                tools::sf_nvcc::TemporaryFileManager temp_mgr(safecuda_opts);
                if (safecuda_opts.enable_verbose)
                        tools::sf_nvcc::print_args(safecuda_opts, nvcc_args);
                // Execute nvcc -dryrun to get compilation pipeline
                tools::sf_nvcc::DryRunParser parser =
                        tools::sf_nvcc::execute_dryrun(nvcc_args,
                                                        safecuda_opts);
                // Execute commands up to and including PTX generation
                std::vector<std::filesystem::path> ptx_paths =
                        tools::sf_nvcc::execute_pre_ptx_stage(
                                parser, safecuda_opts, temp_mgr);
                // Modify all generated PTX files
                std::unordered_map<std::string, std::string> modified_ptx_map;
                for (const auto &ptx_path : ptx_paths) {
                        auto result = tools::sf_nvcc::modify_ptx(ptx_path,
                                                                safecuda_opts);
                        modified_ptx_map[ptx_path.string()] =
                                result.modified_ptx_path;
                }
                std::fflush(stdout);
```

```
                // Execute remaining compilation commands with modified PTX
                tools::sf_nvcc::execute_post_ptx_stage(
                        modified_ptx_map, parser, safecuda_opts, nvcc_args);
        } catch (std::invalid_argument &e) {
                std::cerr << e.what() << '\n';
                return EXIT_FAILURE;
        } catch (std::runtime_error &e) {
                std::cerr << e.what() << '\n';
                return EXIT_FAILURE;
        }
        return EXIT_SUCCESS;
}
```

6. SafeCUDA/tools/sf-nvcc/include/sf_options.h:

```
#ifndef SAFECUDA_SF_OPTIONS_H
#define SAFECUDA_SF_OPTIONS_H
#include <string>
#include <vector>
// ANSI COLOR BUILDER
#define ACOL(C, TB) "\033[" TB C "m"
#define ACOL_RESET() "\033[0m"
#define ACOL_DF "3"
#define ACOL_BF "9"
#define ACOL_DB "4"
#define ACOL_BB "10"
#define ACOL_K "0"
#define ACOL_R "1"
#define ACOL_G "2"
#define ACOL_Y "3"
#define ACOL_BL "4"
#define ACOL_M "5"
#define ACOL_C "6"
#define ACOL_W "7"
namespace safecuda::tools::sf_nvcc
{
/**
 * @brief Configuration options for SafeCUDA PTX modification
 *
 * Holds all SafeCUDA-specific options parsed from -sf-* command line arguments.
 */
struct SafeCudaOptions {
        bool enable_bounds_check = true; ///< Enable runtime bounds checking.
        bool enable_debug = false; ///< Enable debug instrumentation.
        bool enable_verbose = false; ///< Enable verbose logging.
        bool fail_fast = true; ///< Abort on first violation.
        std::string keep_dir; ///< Directory to store intermediate files.
```

```
};

/**
 * @brief Configuration options for NVCC
 *
 * Holds all NVCC-specific options parsed from command line arguments.
 */
struct NvccOptions {
        std::vector<std::string> input_files; ///< NVCC input files
        std::vector<std::string> nvcc_args; ///< Remaining NVCC arguments.
        std::string output_path; ///< NVCC output file path
};

/**
 * @brief Complete option set for sf-nvcc compilation
 *
 * Contains both SafeCUDA-specific options and standard NVCC arguments.
 */
struct SfNvccOptions {
        SafeCudaOptions safecuda_opts; ///< SafeCUDA-specific options.
        NvccOptions nvcc_opts; ///< NVCC arguments
};

/**
 * @brief Parse command line arguments into structured options
 *
 * Separates SafeCUDA-specific options (starting with -sf-) from standard
 * NVCC arguments. Validates option syntax and provides error reporting.
 *
 * @param argc Number of command line arguments
 * @param argv Array of command line argument strings
 * @return SfNvccOptions Parsed options structure
 * @throws std::invalid_argument for malformed SafeCUDA options
 *
```

```
 * @note All NVCC arguments are preserved and passed through except -dryrun
 */
SfNvccOptions parse_command_line(int argc, char *argv[]);

/**
 * @brief Print sf-nvcc help text.
 */
void print_help();

/**
 * @brief Print sf-nvcc version info.
 */
void print_version();

/**
 * @brief Print verbose arguments
 */
void print_args(const SafeCudaOptions &safecuda_opts,
                const NvccOptions &nvcc_opts);

} // namespace safecuda::tools::sf_nvcc
#endif // SAFECUDA_SF_OPTIONS_H
```

7. SafeCUDA/tools/sf-nvcc/include/nvcc_wrapper.h:

```
#ifndef SAFECUDA_NVCC_WRAPPER_H
#define SAFECUDA_NVCC_WRAPPER_H
#include "sf_options.h"
#include "ptx_modifier.h"
#include <filesystem>
#include <string>
#include <unordered_map>
#include <vector>
namespace safecuda::tools::sf_nvcc
{
/**
 * @brief RAII wrapper for managing temporary compilation files
 *
 * Tracks intermediate files and ensures cleanup on failure. Provides
 * thread-safe access to temporary file paths and optional file preservation
 * for debugging.
 *
 * @note This class is move-only to ensure unique ownership of temp directories
 * @note Not thread-safe - each compilation thread should have its own instance
 */
class TemporaryFileManager {
        bool preserve_on_exit;
        std::filesystem::path dir_path;
        std::vector<std::filesystem::path> temp_files;

   public:
        explicit TemporaryFileManager(const SafeCudaOptions &sf_opts);
        ~TemporaryFileManager();
        TemporaryFileManager(const TemporaryFileManager &) = delete;
        TemporaryFileManager &operator=(const TemporaryFileManager &) =
delete;
        TemporaryFileManager(TemporaryFileManager &&) = default;
```

```
        TemporaryFileManager &operator=(TemporaryFileManager &&) = default;

        /**
    * @brief Returns the working directory being used, else use temp directory
    *
    * @return Working folder directory
    */
        [[nodiscard]] std::filesystem::path get_working_dir() const noexcept;

        /**
    * @brief Filter PTX files from intermediate files
    *
    * Filters PTX files (.ptx extension) from NVCC intermediate files.
    *
    * @return Vector of valid PTX file paths
    */
        [[nodiscard]] std::vector<std::filesystem::path>
        filter_ptx_paths() const noexcept;

        /**
    * @brief Returns list of all intermediate files generated by nvcc
    *
    * @return Vector of all intermediate files generated
    */
        [[nodiscard]] std::vector<std::filesystem::path>
        get_intermediate_files() const noexcept;

        /**
    * @brief Add a file path to store;
    */
        void add_file(const std::filesystem::path &path) noexcept;
};

/**
```

```
 * @brief Parser for nvcc -dryrun output
 *
 * Parses the command sequence produced by nvcc -dryrun and identifies
 * compilation stages, particularly PTX generation commands.
 */
struct DryRunParser {
        std::vector<std::string> commands; ///< All commands from dryrun
        std::unordered_map<std::string, std::string>
                ptx_files; ///< Maps PTX paths to source files
        size_t ptx_stage_end_index{0}; ///< Index after last PTX generation

        /**
    * @brief Parse nvcc -dryrun output into command stages
    *
    * @param dryrun_output The full output from nvcc -dryrun
    * @throws std::runtime_error if parsing fails
    */
        void parse(const std::string &dryrun_output);

        /**
    * @brief Get commands that execute up to and including PTX generation
    *
    * @return Vector of commands to execute before PTX modification
    */
        [[nodiscard]] std::vector<std::string> get_pre_ptx_commands() const;

        /**
    * @brief Get commands that execute after PTX generation
    *
    * @param modified_ptx_map Map from original PTX paths to modified PTX
paths
    * @return Vector of commands with modified PTX paths substituted
    */
        [[nodiscard]] std::vector<std::string>
```

```
        get_post_ptx_commands(const std::unordered_map<std::string, std::string>
                                        &modified_ptx_map) const;
};

/**
 * @brief Execute nvcc -dryrun and parse the compilation pipeline
 *
 * Runs nvcc with -dryrun flag using user's original arguments to extract
 * the exact sequence of compilation commands nvcc would execute.
 *
 * @param nvcc_opts NVCC arguments including source files and compilation flags
 * @param sf_opts SafeCUDA options for controlling compilation
 * @return DryRunParser containing parsed command sequence
 * @throws std::runtime_error if nvcc -dryrun execution fails
 */
DryRunParser execute_dryrun(const NvccOptions &nvcc_opts,
                            const SafeCudaOptions &sf_opts);

/**
 * @brief Execute pre-PTX compilation commands
 *
 * Executes all commands from preprocessing through PTX generation,
 * collecting intermediate PTX file paths for modification.
 *
 * @param parser DryRunParser with pre-PTX commands
 * @param sf_opts SafeCUDA options for controlling execution
 * @param temp_mgr TemporaryFileManager to track generated files
 * @return Vector of generated PTX file paths
 * @throws std::runtime_error if any command execution fails
 */
std::vector<std::filesystem::path>
execute_pre_ptx_stage(const DryRunParser &parser,
                      const SafeCudaOptions &sf_opts,
                      TemporaryFileManager &temp_mgr);
```

```
/**
 * @brief Execute post-PTX compilation commands with modified PTX
 *
 * Executes remaining commands (ptxas, fatbinary, linking) using
 * SafeCUDA-modified PTX files instead of original PTX.
 *
 * @param modified_ptx_map Map from original PTX paths to modified PTX paths
 * @param parser DryRunParser with post-PTX commands
 * @param sf_opts SafeCUDA options for controlling execution
 * @return true if compilation completes successfully
 * @throws std::runtime_error if any command execution fails
 */
bool execute_post_ptx_stage(
        const std::unordered_map<std::string, std::string> &modified_ptx_map,
        const DryRunParser &parser, const SafeCudaOptions &sf_opts,
        const NvccOptions &nvcc_opts);

} // namespace safecuda::tools::sf_nvcc
#endif // SAFECUDA_NVCC_WRAPPER_H
```

The above codes are the implementation (and headers, for functions and their descriptions) of the sf-nvcc executable. Running this makes and provides the "libsafecuda.so" file we need in the "cmake-build-<release type>" directory. It also implements the aforementioned flags, as well as some other functions like PTX instrumentation, etc.

Shown below is the implementation of the pre and post-PTX instrumentation operations:

```
std::vector<fs::path>
sf_nvcc::execute_pre_ptx_stage(const DryRunParser &parser,
                               const SafeCudaOptions &sf_opts,
                               TemporaryFileManager &temp_mgr)
```

```
{
        auto pre_ptx_cmds = parser.get_pre_ptx_commands();

        if (sf_opts.enable_verbose) {
                std::cout << ACOL(ACOL_Y, ACOL_BB) << ACOL(ACOL_K,
ACOL_DF)
                          << "Executing pre-PTX compilation stage ("
                          << pre_ptx_cmds.size() << " commands)" <<
ACOL_RESET()
                          << "\n";
        }
        for (const auto &cmd : pre_ptx_cmds) {
                if (sf_opts.enable_verbose) {
                        std::cout << ACOL(ACOL_C, ACOL_DF)
                                  << "Executing: " << ACOL_RESET() << cmd
                                  << "\n";
                }
                int ret = std::system(cmd.c_str());
                if (ret != 0) {
                        throw std::runtime_error(
                                "Pre-PTX command failed with exit code " +
                                std::to_string(ret) + ": " + cmd);
                }
        }

        std::vector<fs::path> ptx_paths;
        for (const auto &[ptx_file, _] : parser.ptx_files) {
                fs::path ptx_path(ptx_file);
                if (fs::exists(ptx_path)) {
                        ptx_paths.push_back(ptx_path);
                        temp_mgr.add_file(ptx_path);

                        if (sf_opts.enable_verbose) {
                                std::cout << ACOL(ACOL_G, ACOL_BF)
```

```
                                                << "Generated PTX: " <<
ACOL_RESET()
                                                << ptx_path << "\n";
                                }
                        } else {
                                throw std::runtime_error(
                                        "Expected PTX file not generated: " + ptx_file);
                        }
                }
                return ptx_paths;
}
bool sf_nvcc::execute_post_ptx_stage(
                const std::unordered_map<std::string, std::string> &modified_ptx_map,
                const DryRunParser &parser, const SafeCudaOptions &sf_opts,
                const NvccOptions &nvcc_opts)
{
                auto post_ptx_cmds = parser.get_post_ptx_commands(modified_ptx_map);
                if (sf_opts.enable_verbose) {
                        std::cout << ACOL(ACOL_Y, ACOL_BB) << ACOL(ACOL_K,
ACOL_DF)
                                        << "Executing post-PTX compilation stage ("
                                        << post_ptx_cmds.size() << " commands)"
                                        << ACOL_RESET() << "\n";
                }

                const std::string op_path{"-o \"" + nvcc_opts.output_path + "\""};
                for (auto &cmd : post_ptx_cmds) {
                        if (sf_opts.enable_verbose) {
                                std::cout << ACOL(ACOL_C, ACOL_DF)
                                                << "Executing: " << ACOL_RESET() << cmd
                                                << "\n";
                        }
                        int ret = std::system(cmd.c_str());
                        if (ret != 0) {
```

```
                throw std::runtime_error(
                    "Post-PTX command failed with exit code " +
                    std::to_string(ret) + ": " + cmd);
            }
        }
        return true;
}
```

APPENDIX 2

PTX FILES

A. Before Modification

```
//
// Generated by NVIDIA NVVM Compiler
//
// Compiler Build ID: CL-36037853
// Cuda compilation tools, release 12.9, V12.9.86
// Based on NVVM 7.0.1
//

.version 8.8
.target sm_75
.address_size 64

        // .globl   _Z10scaleArrayPfi

.visible .entry _Z10scaleArrayPfi(
        .param .u64 _Z10scaleArrayPfi_param_0,
        .param .u32 _Z10scaleArrayPfi_param_1
)
{
        .reg .pred      %p<2>;
        .reg .f32  %f<3>;
        .reg .b32 %r<6>;
        .reg .b64 %rd<5>;


        ld.param.u64    %rd1, [_Z10scaleArrayPfi_param_0];
        ld.param.u32    %r2, [_Z10scaleArrayPfi_param_1];
        mov.u32 %r3, %ctaid.x;
```

```
        mov.u32 %r4, %ntid.x;
        mov.u32 %r5, %tid.x;
        mad.lo.s32      %r1, %r3, %r4, %r5;
        setp.ge.s32     %p1, %r1, %r2;
        @%p1 bra        $L__BB0_2;

        cvta.to.global.u64  %rd2, %rd1;
        mul.wide.s32    %rd3, %r1, 4;
        add.s64   %rd4, %rd2, %rd3;
        ld.global.f32   %f1, [%rd4];
        add.f32   %f2, %f1, %f1;
        st.global.f32   [%rd4], %f2;

$L__BB0_2:
        ret;

}
        // .globl   _Z6addOnePfi
.visible .entry _Z6addOnePfi(
        .param .u64 _Z6addOnePfi_param_0,
        .param .u32 _Z6addOnePfi_param_1
)
{
        .reg .pred      %p<2>;
        .reg .f32  %f<3>;
        .reg .b32 %r<6>;
        .reg .b64 %rd<5>;


        ld.param.u64    %rd1, [_Z6addOnePfi_param_0];
        ld.param.u32    %r2, [_Z6addOnePfi_param_1];
        mov.u32 %r3, %ctaid.x;
        mov.u32 %r4, %ntid.x;
        mov.u32 %r5, %tid.x;
```

```
	mad.lo.s32 	%r1, %r3, %r4, %r5;
	setp.ge.s32 	%p1, %r1, %r2;
	@%p1 bra 	$L__BB1_2;

	cvta.to.global.u64 	%rd2, %rd1;
	mul.wide.s32 	%rd3, %r1, 4;
	add.s64 	%rd4, %rd2, %rd3;
	ld.global.f32 	%f1, [%rd4];
	add.f32 	%f2, %f1, 0f3F800000;
	st.global.f32 	[%rd4], %f2;

$L__BB1_2:
	ret;

}
	// .globl	_Z19computeSeriesKernelPyi
.visible .entry _Z19computeSeriesKernelPyi(
	.param .u64 _Z19computeSeriesKernelPyi_param_0,
	.param .u32 _Z19computeSeriesKernelPyi_param_1
)
{
	.reg .pred 	%p<6>;
	.reg .f32 	%f<32>;
	.reg .b32 	%r<53>;
	.reg .b64 	%rd<28>;


	ld.param.u64 	%rd8, [_Z19computeSeriesKernelPyi_param_0];
	ld.param.u32 	%r29, [_Z19computeSeriesKernelPyi_param_1];
	mov.u32 %r1, %ntid.x;
	mov.u32 %r2, %ctaid.x;
	mov.u32 %r3, %tid.x;
	mad.lo.s32 	%r50, %r2, %r1, %r3;
	mov.u32 %r5, %nctaid.x;
```

```
        mul.lo.s32      %r6, %r1, %r5;
        setp.ge.s32     %p1, %r50, %r29;
        mov.u64 %rd27, 0;
        @%p1 bra        $L__BB2_6;

        cvt.rn.f32.s32  %f1, %r29;
        add.s32 %r45, %r50, %r6;
        not.b32 %r30, %r45;
        add.s32 %r31, %r6, %r29;
        add.s32 %r32, %r31, %r30;
        div.u32 %r33, %r32, %r6;
        add.s32 %r8, %r33, 1;
        and.b32 %r52, %r8, 3;
        setp.lt.u32     %p2, %r33, 3;
        mov.u64 %rd27, 0;
        @%p2 bra        $L__BB2_4;

        sub.s32 %r49, %r8, %r52;
        mad.lo.s32      %r34, %r5, 3, %r2;
        mad.lo.s32      %r47, %r1, %r34, %r3;
        shl.b32 %r12, %r6, 2;
        shl.b32 %r35, %r5, 1;
        add.s32 %r36, %r2, %r35;
        mad.lo.s32      %r46, %r1, %r36, %r3;
        mov.u64 %rd27, 0;

$L__BB2_3:
        cvt.rn.f32.s32  %f2, %r50;
        div.rn.f32      %f3, %f2, %f1;
        add.f32 %f4, %f3, %f3;
        fma.rn.f32      %f5, %f3, %f3, %f4;
        add.f32 %f6, %f5, 0f3F800000;
        mul.f32 %f7, %f6, 0f447A0000;
        cvt.rzi.s32.f32 %r37, %f7;
```

```
cvt.s64.s32         %rd13, %r37;
add.s64   %rd14, %rd27, %rd13;
cvt.rn.f32.s32      %f8, %r45;
div.rn.f32          %f9, %f8, %f1;
add.f32   %f10, %f9, %f9;
fma.rn.f32          %f11, %f9, %f9, %f10;
add.f32   %f12, %f11, 0f3F800000;
mul.f32   %f13, %f12, 0f447A0000;
cvt.rzi.s32.f32     %r38, %f13;
cvt.s64.s32         %rd15, %r38;
add.s64   %rd16, %rd14, %rd15;
add.s32   %r39, %r50, %r6;
add.s32   %r40, %r39, %r6;
cvt.rn.f32.s32      %f14, %r46;
div.rn.f32          %f15, %f14, %f1;
add.f32   %f16, %f15, %f15;
fma.rn.f32          %f17, %f15, %f15, %f16;
add.f32   %f18, %f17, 0f3F800000;
mul.f32   %f19, %f18, 0f447A0000;
cvt.rzi.s32.f32     %r41, %f19;
cvt.s64.s32         %rd17, %r41;
add.s64   %rd18, %rd16, %rd17;
add.s32   %r42, %r40, %r6;
cvt.rn.f32.s32      %f20, %r47;
div.rn.f32          %f21, %f20, %f1;
add.f32   %f22, %f21, %f21;
fma.rn.f32          %f23, %f21, %f21, %f22;
add.f32   %f24, %f23, 0f3F800000;
mul.f32   %f25, %f24, 0f447A0000;
cvt.rzi.s32.f32     %r43, %f25;
cvt.s64.s32         %rd19, %r43;
add.s64   %rd27, %rd18, %rd19;
add.s32   %r50, %r42, %r6;
add.s32   %r47, %r47, %r12;
```

```
        add.s32   %r46, %r46, %r12;
        add.s32   %r45, %r45, %r12;
        add.s32   %r49, %r49, -4;
        setp.ne.s32       %p3, %r49, 0;
        @%p3 bra          $L__BB2_3;

$L__BB2_4:
        setp.eq.s32       %p4, %r52, 0;
        @%p4 bra          $L__BB2_6;

$L__BB2_5:
        .pragma "nounroll";
        cvt.rn.f32.s32    %f26, %r50;
        div.rn.f32        %f27, %f26, %f1;
        add.f32   %f28, %f27, %f27;
        fma.rn.f32        %f29, %f27, %f27, %f28;
        add.f32   %f30, %f29, 0f3F800000;
        mul.f32   %f31, %f30, 0f447A0000;
        cvt.rzi.s32.f32   %r44, %f31;
        cvt.s64.s32       %rd20, %r44;
        add.s64   %rd27, %rd27, %rd20;
        add.s32   %r50, %r50, %r6;
        add.s32   %r52, %r52, -1;
        setp.ne.s32       %p5, %r52, 0;
        @%p5 bra          $L__BB2_5;

$L__BB2_6:
        cvta.to.global.u64  %rd21, %rd8;
        atom.global.add.u64         %rd22, [%rd21], %rd27;
        ret;

}
        // .globl   _Z17outOfBoundsKernelPfi
.visible .entry _Z17outOfBoundsKernelPfi(
```

```
        .param .u64 _Z17outOfBoundsKernelPfi_param_0,
        .param .u32 _Z17outOfBoundsKernelPfi_param_1
)
{
        .reg .b32 %r<9>;
        .reg .b64 %rd<5>;


        ld.param.u64        %rd1, [_Z17outOfBoundsKernelPfi_param_0];
        ld.param.u32        %r1, [_Z17outOfBoundsKernelPfi_param_1];
        cvta.to.global.u64  %rd2, %rd1;
        mov.u32 %r2, %ctaid.x;
        mov.u32 %r3, %ntid.x;
        mov.u32 %r4, %tid.x;
        mad.lo.s32          %r5, %r2, %r3, %r4;
        add.s32   %r6, %r5, %r1;
        add.s32   %r7, %r6, 100;
        mul.wide.s32        %rd3, %r7, 4;
        add.s64   %rd4, %rd2, %rd3;
        mov.u32 %r8, 1109917696;
        st.global.u32       [%rd4], %r8;
        ret;

}
```

B. After Modification

```
//
// Generated by NVIDIA NVVM Compiler
//
// Compiler Build ID: CL-36037853
// Cuda compilation tools, release 12.9, V12.9.86
// Based on NVVM 7.0.1
//

.version 8.8
.target sm_75
.address_size 64



.visible .global .align 8 .u64 d_table;

.visible .func __bounds_check_safecuda(
        .param .b64 __bounds_check_safecuda_param_0
)
{
        .reg .pred      %p<12>;
        .reg .b16 %rs<11>;
        .reg .b32 %r<24>;
        .reg .b64 %rd<17>;


        ld.param.u64    %rd6, [__bounds_check_safecuda_param_0];
        ld.global.u64   %rd7, [d_table];
        add.s64 %rd1, %rd7, 8;
        ld.u32  %r1, [%rd7+8];
        setp.lt.u32     %p1, %r1, 2;
        ld.u64  %rd2, [%rd7];
        mov.u32 %r22, -1;
```

```
        mov.u16 %rs9, 0;
        @%p1 bra        $L__BB0_7;

        mov.u16 %rs9, 0;
        mov.u32 %r22, -1;
        mov.u32 %r20, 1;

$L__BB0_2:
        cvt.u64.u32     %rd3, %r20;
        mul.wide.u32    %rd8, %r20, 24;
        add.s64 %rd4, %rd2, %rd8;
        ld.u64  %rd5, [%rd4];
        setp.gt.u64     %p2, %rd5, %rd6;
        @%p2 bra        $L__BB0_6;

        ld.u32  %rd9, [%rd4+8];
        add.s64 %rd10, %rd5, %rd9;
        setp.le.u64     %p3, %rd10, %rd6;
        @%p3 bra        $L__BB0_6;

        ld.u32  %r4, [%rd4+12];
        setp.eq.s32     %p4, %r4, 0;
        @%p4 bra        $L__BB0_14;

        cvt.u32.u64     %r12, %rd3;
        shr.u32 %r13, %r4, 1;
        and.b32 %r14, %r13, 1;
        setp.eq.b32     %p5, %r14, 1;
        selp.b16 %rs9, 1, %rs9, %p5;
        selp.b32 %r22, %r12, %r22, %p5;

$L__BB0_6:
        cvt.u32.u64     %r15, %rd3;
        add.s32 %r20, %r15, 1;
```

```
        setp.lt.u32     %p6, %r20, %r1;
        @%p6 bra        $L__BB0_2;

$L__BB0_7:
        and.b16  %rs7, %rs9, 255;
        setp.eq.s16     %p7, %rs7, 0;
        @%p7 bra        $L__BB0_11;

        add.s64  %rd11, %rd2, 12;
        atom.or.b32     %r16, [%rd11], 2;
        and.b32  %r17, %r16, 2;
        setp.ne.s32     %p8, %r17, 0;
        @%p8 bra        $L__BB0_10;

        ld.u64   %rd12, [%rd1+-8];
        st.u64   [%rd12], %rd6;
        ld.global.u64    %rd13, [d_table];
        ld.u64   %rd14, [%rd13];
        st.u32   [%rd14+8], %r22;

$L__BB0_10:
        // begin inline asm
        trap;
        // end inline asm
        bra.uni  $L__BB0_14;

$L__BB0_11:
        add.s64  %rd15, %rd2, 12;
        atom.or.b32     %r18, [%rd15], 1;
        and.b32  %r19, %r18, 1;
        setp.eq.b32     %p9, %r19, 1;
        mov.pred        %p10, 0;
        xor.pred        %p11, %p9, %p10;
        @%p11 bra       $L__BB0_13;
```

```
        ld.u64      %rd16, [%rd1+-8];
        st.u64      [%rd16], %rd6;

$L__BB0_13:
        // begin inline asm
        trap;
        // end inline asm

$L__BB0_14:
        ret;

}




        // .globl   _Z10scaleArrayPfi

.visible .entry _Z10scaleArrayPfi(.param .u64 d_table_param,
        .param .u64 _Z10scaleArrayPfi_param_0,
        .param .u32 _Z10scaleArrayPfi_param_1
)
{
.reg .b64 %rd_table;
ld.param.u64 %rd_table, [d_table_param];
st.global.u64 [d_table], %rd_table;
        .reg .pred          %p<2>;
        .reg .f32  %f<3>;
        .reg .b32 %r<6>;
        .reg .b64 %rd<5>;


        ld.param.u64        %rd1, [_Z10scaleArrayPfi_param_0];
```

```
        ld.param.u32    %r2, [_Z10scaleArrayPfi_param_1];
        mov.u32 %r3, %ctaid.x;
        mov.u32 %r4, %ntid.x;
        mov.u32 %r5, %tid.x;
        mad.lo.s32      %r1, %r3, %r4, %r5;
        setp.ge.s32     %p1, %r1, %r2;
        @%p1 bra        $L__BB0_2;

        cvta.to.global.u64  %rd2, %rd1;
        mul.wide.s32    %rd3, %r1, 4;
        add.s64   %rd4, %rd2, %rd3;
        call __bounds_check_safecuda, (%rd4);
        ld.global.f32   %f1, [%rd4];
        add.f32   %f2, %f1, %f1;
        call __bounds_check_safecuda, (%rd4);
        st.global.f32   [%rd4], %f2;

$L__BB0_2:
        ret;

}
        // .globl   _Z6addOnePfi
.visible .entry _Z6addOnePfi(.param .u64 d_table_param,
        .param .u64 _Z6addOnePfi_param_0,
        .param .u32 _Z6addOnePfi_param_1
)
{
.reg .b64 %rd_table;
ld.param.u64 %rd_table, [d_table_param];
st.global.u64 [d_table], %rd_table;
        .reg .pred      %p<2>;
        .reg .f32  %f<3>;
        .reg .b32 %r<6>;
        .reg .b64 %rd<5>;
```

```
        ld.param.u64    %rd1, [_Z6addOnePfi_param_0];
        ld.param.u32    %r2, [_Z6addOnePfi_param_1];
        mov.u32 %r3, %ctaid.x;
        mov.u32 %r4, %ntid.x;
        mov.u32 %r5, %tid.x;
        mad.lo.s32      %r1, %r3, %r4, %r5;
        setp.ge.s32     %p1, %r1, %r2;
        @%p1 bra        $L__BB1_2;

        cvta.to.global.u64  %rd2, %rd1;
        mul.wide.s32    %rd3, %r1, 4;
        add.s64   %rd4, %rd2, %rd3;
        call __bounds_check_safecuda, (%rd4);
        ld.global.f32   %f1, [%rd4];
        add.f32   %f2, %f1, 0f3F800000;
        call __bounds_check_safecuda, (%rd4);
        st.global.f32   [%rd4], %f2;

$L__BB1_2:
        ret;

}
        // .globl   _Z19computeSeriesKernelPyi
.visible .entry _Z19computeSeriesKernelPyi(.param .u64 d_table_param,
        .param .u64 _Z19computeSeriesKernelPyi_param_0,
        .param .u32 _Z19computeSeriesKernelPyi_param_1
)
{
.reg .b64 %rd_table;
ld.param.u64 %rd_table, [d_table_param];
st.global.u64 [d_table], %rd_table;
        .reg .pred      %p<6>;
```

```
.reg .f32  %f<32>;
.reg .b32 %r<53>;
.reg .b64 %rd<28>;


ld.param.u64        %rd8, [_Z19computeSeriesKernelPyi_param_0];
ld.param.u32        %r29, [_Z19computeSeriesKernelPyi_param_1];
mov.u32 %r1, %ntid.x;
mov.u32 %r2, %ctaid.x;
mov.u32 %r3, %tid.x;
mad.lo.s32          %r50, %r2, %r1, %r3;
mov.u32 %r5, %nctaid.x;
mul.lo.s32          %r6, %r1, %r5;
setp.ge.s32         %p1, %r50, %r29;
mov.u64 %rd27, 0;
@%p1 bra            $L__BB2_6;

cvt.rn.f32.s32      %f1, %r29;
add.s32   %r45, %r50, %r6;
not.b32   %r30, %r45;
add.s32   %r31, %r6, %r29;
add.s32   %r32, %r31, %r30;
div.u32   %r33, %r32, %r6;
add.s32   %r8, %r33, 1;
and.b32   %r52, %r8, 3;
setp.lt.u32         %p2, %r33, 3;
mov.u64 %rd27, 0;
@%p2 bra            $L__BB2_4;

sub.s32   %r49, %r8, %r52;
mad.lo.s32          %r34, %r5, 3, %r2;
mad.lo.s32          %r47, %r1, %r34, %r3;
shl.b32   %r12, %r6, 2;
shl.b32   %r35, %r5, 1;
```

```
        add.s32   %r36, %r2, %r35;
        mad.lo.s32        %r46, %r1, %r36, %r3;
        mov.u64 %rd27, 0;

$L__BB2_3:
        cvt.rn.f32.s32        %f2, %r50;
        div.rn.f32        %f3, %f2, %f1;
        add.f32   %f4, %f3, %f3;
        fma.rn.f32        %f5, %f3, %f3, %f4;
        add.f32   %f6, %f5, 0f3F800000;
        mul.f32   %f7, %f6, 0f447A0000;
        cvt.rzi.s32.f32       %r37, %f7;
        cvt.s64.s32       %rd13, %r37;
        add.s64   %rd14, %rd27, %rd13;
        cvt.rn.f32.s32        %f8, %r45;
        div.rn.f32        %f9, %f8, %f1;
        add.f32   %f10, %f9, %f9;
        fma.rn.f32        %f11, %f9, %f9, %f10;
        add.f32   %f12, %f11, 0f3F800000;
        mul.f32   %f13, %f12, 0f447A0000;
        cvt.rzi.s32.f32       %r38, %f13;
        cvt.s64.s32       %rd15, %r38;
        add.s64   %rd16, %rd14, %rd15;
        add.s32   %r39, %r50, %r6;
        add.s32   %r40, %r39, %r6;
        cvt.rn.f32.s32        %f14, %r46;
        div.rn.f32        %f15, %f14, %f1;
        add.f32   %f16, %f15, %f15;
        fma.rn.f32        %f17, %f15, %f15, %f16;
        add.f32   %f18, %f17, 0f3F800000;
        mul.f32   %f19, %f18, 0f447A0000;
        cvt.rzi.s32.f32       %r41, %f19;
        cvt.s64.s32       %rd17, %r41;
        add.s64   %rd18, %rd16, %rd17;
```

```
        add.s32   %r42, %r40, %r6;
        cvt.rn.f32.s32        %f20, %r47;
        div.rn.f32            %f21, %f20, %f1;
        add.f32   %f22, %f21, %f21;
        fma.rn.f32            %f23, %f21, %f21, %f22;
        add.f32   %f24, %f23, 0f3F800000;
        mul.f32   %f25, %f24, 0f447A0000;
        cvt.rzi.s32.f32       %r43, %f25;
        cvt.s64.s32           %rd19, %r43;
        add.s64   %rd27, %rd18, %rd19;
        add.s32   %r50, %r42, %r6;
        add.s32   %r47, %r47, %r12;
        add.s32   %r46, %r46, %r12;
        add.s32   %r45, %r45, %r12;
        add.s32   %r49, %r49, -4;
        setp.ne.s32           %p3, %r49, 0;
        @%p3 bra              $L__BB2_3;

$L__BB2_4:
        setp.eq.s32           %p4, %r52, 0;
        @%p4 bra              $L__BB2_6;

$L__BB2_5:
        .pragma "nounroll";
        cvt.rn.f32.s32        %f26, %r50;
        div.rn.f32            %f27, %f26, %f1;
        add.f32   %f28, %f27, %f27;
        fma.rn.f32            %f29, %f27, %f27, %f28;
        add.f32   %f30, %f29, 0f3F800000;
        mul.f32   %f31, %f30, 0f447A0000;
        cvt.rzi.s32.f32       %r44, %f31;
        cvt.s64.s32           %rd20, %r44;
        add.s64   %rd27, %rd27, %rd20;
        add.s32   %r50, %r50, %r6;
```

```
        add.s32   %r52, %r52, -1;
        setp.ne.s32         %p5, %r52, 0;
        @%p5 bra            $L__BB2_5;

$L__BB2_6:
        cvta.to.global.u64  %rd21, %rd8;
        call __bounds_check_safecuda, (%rd21);
        atom.global.add.u64         %rd22, [%rd21], %rd27;
        ret;

}
        // .globl   _Z17outOfBoundsKernelPfi
.visible .entry _Z17outOfBoundsKernelPfi(.param .u64 d_table_param,
        .param .u64 _Z17outOfBoundsKernelPfi_param_0,
        .param .u32 _Z17outOfBoundsKernelPfi_param_1
)
{
.reg .b64 %rd_table;
ld.param.u64 %rd_table, [d_table_param];
st.global.u64 [d_table], %rd_table;
        .reg .b32 %r<9>;
        .reg .b64 %rd<5>;


        ld.param.u64        %rd1, [_Z17outOfBoundsKernelPfi_param_0];
        ld.param.u32        %r1, [_Z17outOfBoundsKernelPfi_param_1];
        cvta.to.global.u64  %rd2, %rd1;
        mov.u32 %r2, %ctaid.x;
        mov.u32 %r3, %ntid.x;
        mov.u32 %r4, %tid.x;
        mad.lo.s32          %r5, %r2, %r3, %r4;
        add.s32   %r6, %r5, %r1;
        add.s32   %r7, %r6, 100;
        mul.wide.s32        %rd3, %r7, 4;
```

```
        add.s64   %rd4, %rd2, %rd3;
        mov.u32 %r8, 1109917696;
        call __bounds_check_safecuda, (%rd4);
        st.global.u32         [%rd4], %r8;
        ret;

}
```

APPENDIX 3

ADDITIONAL DIAGRAMS