

RUSTILE: Safe GPU Programming with Tile-based Abstraction

Yue Wang Mingliang Liu Baojian Hua*

School of Software Engineering, University of Science and Technology of China, China
Suzhou Institute for Advanced Research, University of Science and Technology of China, China
wy_ustc123@mail.ustc.edu.cn liumingliang@mail.ustc.edu.cn bjhua@ustc.edu.cn

Abstract—Tile-based GPU programming abstractions have emerged as an effective approach for improving programmer productivity by raising the level of abstraction above low-level thread and memory management. However, existing tile-based frameworks provide limited static guarantees about program correctness and security. Prior approaches either prioritize productivity while overlooking static safety, or enforce strong correctness guarantees but sacrifice usability and abstraction, leaving a fundamental gap between developer productivity and security in GPU programming.

This paper presents RUSTILE, the first tile-based GPU programming language that bridges this gap by enforcing security inspired by ownership and borrowing disciplines at tile granularity. RUSTILE enables programmers to express parallel computation at the tile level while statically preventing unsafe overlapping accesses to global memory between concurrently executing tiles. All safety guarantees are enforced at compile time through static analysis, and programs are lowered to diverse tile-based GPU backends without incurring additional runtime overhead in our evaluation. We implemented a prototype for RUSTILE and leveraged Triton as a representative tile-based GPU backend for evaluation. Our results show that RUSTILE provides strong static safety guarantees while remaining expressive and achieving performance comparable to manually written tile-based GPU kernel programs.

Index Terms—GPU programming, Tile-based abstraction, static memory safety

I. INTRODUCTION

Graphics Processing Units (GPUs) have become indispensable for modern high-performance computing and artificial intelligence workloads [1], [2], providing massive parallelism and computational throughput. As GPU architectures and software stacks grow in complexity, high-level programming languages and abstractions are essential for improving developer productivity while fully exploiting hardware performance [3]–[6]. However, designing GPU programming abstractions that are both efficient and secure remains a fundamental challenge [7]–[9].

Recent research has increasingly shifted toward higher-level GPU programming abstractions without sacrificing execution efficiency. In particular, many systems adopt a *tile-based* programming model [10]–[13], which structures computation around regular, block-shaped regions of data mapped to GPU execution units. Tile-based abstractions improve programmability and performance portability by enabling programmers

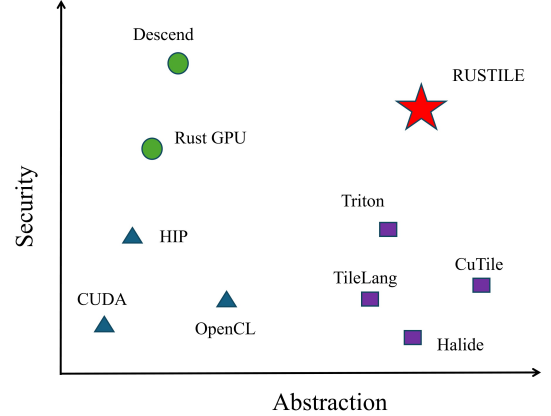


Fig. 1: Classification of GPU programming languages along abstraction and security dimensions.

to reason at the level of tiles rather than individual threads. Meanwhile, prior work has explored GPU safety mechanisms ranging from type-based approaches [14] to static verification techniques [9], [15], [16]. However, these approaches are largely orthogonal to tile-based programming abstractions. Existing tile-based frameworks provide limited static guarantees: overlapping accesses between concurrently executing tiles can silently introduce data races (see §II).

Specifically, a modern GPU programming abstraction should satisfy the following two requirements simultaneously:

- **(R1) Productivity**: enabling concise and expressive kernel implementations without managing low-level execution details;
- **(R2) Security**: statically detecting conflicting tile-level accesses to global memory across concurrently executing program instances.

Unfortunately, existing approaches typically satisfy at most one of these requirements. Low-level interfaces such as CUDA provide maximal control over execution and memory, but place the burden of correctness and safety on programmers [17], [18]. Recent tile-based frameworks, including Triton [10] and TileLang [12], raise the level of abstraction and improve programmability, but still allow unsafe overlapping tile-level accesses. Conversely, approaches that emphasize strong safety guarantees often sacrifice productivity. Languages inspired by Rust-style safety [19]–[21] offer powerful static reasoning

* The corresponding author.

through ownership and borrowing, but operate at relatively low levels of abstraction, requiring developers to reason explicitly about memory and execution details. Verification-oriented systems based on Hoare-style reasoning [9], [16], [22], [23] provide strong correctness guarantees, but typically require manual annotations and substantial proof effort. These limitations highlight a fundamental tension: higher-level abstractions obscure safety reasoning, while strong safety mechanisms require low-level reasoning.

In this paper, we present RUSTILE, a tile-based GPU programming language that combines tile-level programming abstractions with static checking of conflicting accesses to global memory across concurrently executing program instances. By integrating ownership-based analysis into the language, RUSTILE aims to detect such conflicts at compile time while preserving the productivity of tile-based programming. Our key observation is that this trade-off is not inherent, but arises from a mismatch between programming abstractions and correctness mechanisms. Bridging this gap requires rethinking how static safety reasoning applies to abstraction-oriented GPU programming models.

Figure 1 situates RUSTILE among existing GPU languages along the dimensions of productivity and security. While prior systems typically achieve at most one of these goals, RUSTILE achieves both.

To enable productivity, we adopt a tile-centric programming methodology that elevates tiles to the primary unit of parallel computation. This allows programmers to reason at the tile level while retaining explicit control over data layout and parallel structure. To ensure security, we employ an ownership-based static analysis inspired by Rust, lifting ownership and borrowing disciplines from individual memory locations to tile-granularity regions [19]–[21], [24]. This analysis checks for conflicting accesses to global memory across concurrently executing program instances at compile time, without adding runtime safety checks. We then lower accepted programs to existing tile-based GPU backends, reusing their code-generation and execution infrastructure.

We implement RUSTILE as a frontend language targeting existing tile-based GPU backends and evaluate it on representative kernels. We use Triton as a representative backend in our evaluation. Our results show that RUSTILE enforces strong static safety while incurring no observable runtime overhead and achieving performance comparable to existing systems.

In summary, this paper makes the following contributions:

- **Problem analysis.** We identify a gap between productivity and static safety in tile-based GPU programming abstractions.
- **Language design.** We propose RUSTILE, a tile-based GPU programming language that integrates ownership-based static checking of conflicting tile-level accesses to global memory.
- **System implementation.** We implement a prototype compiler targeting tile-based GPU backends.
- **Experimental evaluation.** We evaluate the effectiveness of RUSTILE in detecting conflicting tile-level global-

memory accesses and measure the runtime performance of the generated kernels against existing implementations.

We publicly release the RUSTILE source code, benchmark suite, and evaluation artifacts to support open science and community reproducibility: <https://doi.org/10.5281/zenodo.19587814>.

The rest of this paper is organized as follows. Section II introduces the background and motivation of this work. Sections III and IV present the design and implementation of RUSTILE, respectively. Section V describes the experimental evaluation. Section VI discusses limitations and our future work. Section VII discusses related work, and Section VIII concludes the paper.

II. BACKGROUND AND MOTIVATION

A. Background

We summarize two strands of prior work that form the conceptual foundation of this paper: tile-based GPU programming abstractions and ownership-based static safety. We focus on core ideas necessary to understand the problem addressed.

Tile-Based GPU Programming Models. Tile-based programming abstracts computation around regular, block-shaped regions of data called tiles, rather than exposing threads and low-level synchronization. Each tile represents a unit of work that compilers map to parallel GPU resources, enabling concise kernels. This model underlies DSLs such as Triton and CuTile, frameworks like TVM, and MLIR-based GPU dialects [3]–[5], [10], [12], [13]. Programmers describe computation over multi-dimensional tiles, which compilers lower to efficient GPU code [3], [10], [12], [25].

In many tile-based systems, each program instance operates on a tile. Operations within a tile are executed by multiple threads, abstracting away explicit thread management while enabling optimizations such as memory coalescing and software pipelining [6], [25], [26]. Despite this abstraction, tile-level operations execute on SIMT hardware and follow GPU memory consistency models [17], [27], [28]. Tile-based systems therefore rely on programmers to reason about interactions between tiles and global memory.

Ownership-Based Static Safety. Ownership- and borrowing-based type systems provide strong compile-time guarantees for concurrency safety [19]–[21], [24]. By controlling how memory is accessed and shared, they provide static guarantees against data races within their supported abstraction level. Rust exemplifies this approach, with unique ownership of memory locations and borrowing rules controlling aliasing and mutability. Recent GPU efforts [24], [29] extend Rust-style safety, but remain thread-centric and low-level, misaligned with tile-based abstractions.

B. Motivation

Tile-based abstractions are widely adopted [3]–[5], [10], [12], simplifying kernel development. However, existing tile-based languages generally lack ownership-based static checking for conflicting accesses to global memory across parallel

```

@triton.jit
def kernel(out_ptr, BLOCK: tl.constexpr):
    pid = tl.program_id(0)
    offs = pid * BLOCK + tl.arange(0, BLOCK)

    # Incorrect index mapping introduces
    # overlap
    out_idx = offs % (2 * BLOCK)
    tl.store(out_ptr + out_idx, 1.0)

```

Fig. 2: A data race caused by overlapping tile-level writes

program instances. Overlapping tile-level accesses can cause data races [7], [8].

Figure 2 illustrates this problem with a simplified Triton kernel. Each program instance is intended to operate on a distinct tile of an output tensor. However, an incorrect index computation causes different instances to write to overlapping memory locations, potentially introducing a data race that is not detected during compilation.

Although each instance operates on a fixed-size tile (BLOCK), the modulo operation disrupts the intended mapping between program instances and output addresses. Consequently, different tiles may access the same global-memory locations. Such errors can silently corrupt results in performance-critical kernels [7], [8].

This limitation arises because existing tile-based programming models abstract away thread-level execution without explicitly associating tiles with ownership or access permissions. As a result, compilers generally lack the information needed to statically reason about interference between concurrent tile-level accesses [4], [10], [12]. Existing approaches address related problems through different trade-offs. Dynamic analysis can detect conflicts on executed paths but incurs runtime overhead, while verification-based approaches [8], [9], [15], [16], [22] may require additional annotations, specialized reasoning, or restrictions on the programming model.

This gap motivates a programming approach that combines the productivity of tile-based abstractions with ownership-based static analysis. Rather than reasoning about individual threads or memory locations directly, such an approach should reason about access permissions at the tile level and statically check conflicting accesses to global memory across concurrently executing program instances.

III. RUSTILE LANGUAGE DESIGN

In this section, we present the design of RUSTILE, a tile-based GPU language that combines high-level abstractions with static safety. We model GPU execution using tiles as first-class units, define explicit memory accesses through place expressions and tile views, and enforce Rust-style ownership and borrowing at tile granularity. This design enables compile-time detection of data races caused by overlapping tile-level accesses to global memory while generating efficient GPU code without runtime overhead [10], [12], [19]–[21], [24].

Execution Resources	$e ::= \text{gpu.grid}(s)$ $e.\text{forall}(d)$ $e.\text{tile}(s)$
Shape	$s ::= \eta_c \mid \eta_c, s$
Dimensions	$d ::= \text{dim}_0 \mid \text{dim}_1 \mid \text{dim}_2$
Compile Const	$\eta_c ::= n \mid \eta_c \oplus \eta_c$
Numeric Op	$\oplus ::= + \mid - \mid \times \mid \div$

Fig. 3: Grammar of execution resources in RUSTILE.

A. Execution Resources with Tile Abstractions

Modern GPUs expose hierarchical execution structures such as grids, blocks, and threads [2], [17]. These structures introduce complexity when reasoning about correctness and data races. RUSTILE instead adopts a tile-centric execution model. It groups threads into statically shaped, first-class units called *tiles* [10], [12]. Each tile represents a structured logical execution unit and serves as the basic abstraction for parallel computation.

Figure 3 defines the grammar of execution resources in RUSTILE. The construct $\text{gpu.grid}(s)$ defines a multidimensional execution space, where s specifies the compile-time shape of the execution space. The construct $e.\text{forall}(d)$ expresses parallel execution along dimension d , while $e.\text{tile}(s)$ associates an execution instance with a statically shaped tile.

All shape parameters are compile-time constants and are resolved during compilation. The dimension specified by $\text{forall}(d)$ must be valid for the enclosing execution space, and the tile shape must be compatible with its parent execution shape. These constraints make the execution structure of RUSTILE programs explicit to static analysis. In particular, the analysis can use this structure to identify potential parallel execution instances when checking tile-level memory accesses [4], [21]. However, the execution structure alone does not establish whether two memory accesses overlap. The analysis must also consider the memory regions accessed by those instances.

We lower tile-level execution resources to backend-specific configurations, such as Triton execution grids and other tile-based GPU backend representations [4], [10], [12]. This lowering introduces no additional runtime checks. The execution model defines how parallel computation is organized, but it does not specify the memory regions accessed by each tile. We next introduce place expressions and tile views to describe these accesses and provide the basis for overlap analysis.

B. Place Expressions and Tile Views

We extend the tile-level execution model with *place expressions* and *views*. Place expressions specify *where* memory is accessed, while views specify *how* the access is performed, including structured transformations and collective operations across multiple threads. This separation makes memory access patterns explicit and supports static reasoning about tile-level memory interactions [8], [21].

Place Expressions	p	$::=$	x
			$ $ $*p$
			$ $ $p.\text{fst}$
			$ $ $p.\text{snd}$
			$ $ $p[\eta_c]$
			$ $ $p.v$
Views	v	$::=$	$\text{group}\langle\eta_c\rangle$
			$ $ $\text{take_left}\langle\eta_c\rangle$
			$ $ $\text{take_right}\langle\eta_c\rangle$
			$ $ transpose
			$ $ reverse
			$ $ $\text{map}(v)$
			$ $ tile_load
			$ $ $\text{tile_store}\langle\eta_c\rangle$

Fig. 4: Grammar of place expressions and views in RUSTILE.

Every memory access in RUSTILE combines a place expression with a view. Ordinary views describe transformations of data layouts, while *tile-level views* represent collective memory operations performed by a tile. A tile view aggregates individual thread accesses into a tile-scoped operation. It is the only mechanism for expressing collective memory accesses within a tile [3], [10], [12], [30].

Figure 4 defines the syntax of place expressions and views. Place expressions identify memory locations through variables, dereferencing, projection, and statically known indexing. Views describe structured transformations and memory operations applied to these locations. For example, `tile_load` and `tile_store` represent collective memory operations performed by a tile, while `transpose` and `reverse` transform the organization of accessed data.

To make tile-level interactions explicit, RUSTILE forbids implicit collective accesses through unrestricted indexing or pointer arithmetic. Programmers must express collective memory operations using tile views. The shape associated with a tile-level view must also be consistent with the enclosing tile execution resource. These restrictions expose the structure of tile-level accesses to the type system.

The tile-aware type system checks the validity of view applications. For each memory access, the analysis must also determine the accessed memory region from the place expression and the applied view. It then uses this region to reason about potential overlap between accesses from concurrent execution instances. This analysis supports ownership and borrowing checks at tile granularity and follows established ownership-based reasoning principles [19]–[21]. We formalize these checks in the next section.

C. Tile-aware Type System

RUSTILE uses a tile-aware type system to track memory permissions and check potentially conflicting global-memory accesses across concurrent execution instances. The type system extends Rust-style ownership and borrowing to tile-level memory operations. It combines borrow permissions with static analysis of accessed memory regions to detect overlapping accesses that may cause data races.

Figure 5 presents the type grammar of RUSTILE. It defines the kinds, data types, function types, borrow permissions, execution contexts, memory spaces, and lifetimes used by the type system. A tile-aware reference has the form $\&_{\rho}^{\omega} * \mu * \delta @ \varepsilon$, where ω specifies the borrow permission, ρ denotes the lifetime, μ identifies the memory space, δ gives the referenced data type, and ε specifies the execution context. The `shrd` permission allows read access, while `uniq` allows exclusive mutation. The execution context distinguishes ordinary thread-level values from values associated with a tile-level execution instance.

Our current prototype supports `gpu.global` memory. The type system uses the judgment $\Gamma; \Sigma; \varepsilon \vdash e : \tau$ to state that expression e has type τ under typing environment Γ , tile-level borrowing state Σ , and execution context ε . The environment Γ records variable types and other typing information, while Σ tracks active borrow permissions for tile-level memory accesses. These permissions enforce local ownership constraints, but they do not alone establish that different execution instances access disjoint memory regions.

To check conflicts across concurrent execution instances, we consider a set A of statically identified global-memory accesses. Each access record has the form $a = (R, m, i)$, where R denotes the accessed memory region, m is the access mode (read or write), and i identifies the execution instance that performs the access. The analysis derives memory regions from place expressions, tile shapes, and supported index expressions. The set A represents other potentially concurrent accesses against which the current access is checked; it excludes the current access itself.

We define $\text{Overlap}(R_1, R_2)$ to hold when memory regions R_1 and R_2 intersect. Two accesses conflict when their execution instances may run concurrently, their regions overlap, and at least one access writes to memory:

$$\begin{aligned} \text{Conflict}(a_1, a_2) \iff & \text{Concurrent}(i_1, i_2) \\ & \wedge \text{Overlap}(R_1, R_2) \\ & \wedge (m_1 = \text{write} \vee m_2 = \text{write}), \end{aligned}$$

where $a_1 = (R_1, m_1, i_1)$ and $a_2 = (R_2, m_2, i_2)$. Thus, overlapping read-only accesses do not constitute a conflict, whereas overlapping read-write and write-write accesses may cause data races.

Using this definition, we define `NoOverlap` as the absence of conflicting overlapping accesses:

$$\text{NoOverlap}(a, A) \iff \forall a' \in A, \neg \text{Conflict}(a, a').$$

This predicate checks the current access against other potentially concurrent accesses, rather than requiring its memory region to be disjoint from every region in the program. In particular, the current access is not compared against itself. The predicate permits overlapping reads while rejecting conflicting accesses identified by the static analysis.

The typing rules for tile-level memory operations are de-

Kinds	$\kappa ::= \text{dty} \mid \text{lft} \mid \text{mem} \mid \text{nat}$	Borrow	$\omega ::= \text{shrd} \mid \text{uniq}$
Data Types	$\delta ::= \text{bool} \mid \text{int} \mid \text{float}$ $\mid (\delta_1, \dots, \delta_n)$ $\mid [\delta; \eta]$ $\mid [[\delta; \eta]]$ $\mid \text{TileView} \langle \delta, s \rangle$ $\mid \&\mathcal{Z}_\rho^\omega * \mu * \delta @ \varepsilon$	Exec Context	$\varepsilon ::= \text{Any} \mid \text{gpu.Thread}$ $\mid \text{gpu.BlockAbstract } s \ \eta$
Function Type	$\tau ::= \langle \bar{k} : \bar{\kappa} \rangle$ $\mid \langle y : \varepsilon \rangle (\bar{e} : \bar{\delta}) \rightarrow \delta$	Memory Spaces	$\mu ::= \text{gpu.global}$
		Lifetimes	$\rho ::= \ell$

Fig. 5: Grammar of kinds and types in RUSTILE.

defined as follows:

$$\frac{\Gamma \vdash p : \text{TileView} \langle \delta, s \rangle \quad \varepsilon = \text{gpu.BlockAbstract } s \quad \text{NoOverlap}(a_{\text{read}}, A)}{\Gamma; \Sigma; \varepsilon \vdash p.\text{tile_load}() : \delta} \quad (\text{T-READ-TILE})$$

$$\frac{\Gamma \vdash p : \text{TileView} \langle \delta, s \rangle \quad \varepsilon = \text{gpu.BlockAbstract } s \quad \Gamma; \Sigma; \varepsilon \vdash v : \delta \quad \Sigma(p) = \text{uniq} \quad \text{NoOverlap}(a_{\text{write}}, A)}{\Gamma; \Sigma; \varepsilon \vdash p.\text{tile_store}(v) : ()} \quad (\text{T-WRITE-TILE})$$

In these rules, a_{read} and a_{write} denote the access records associated with the current tile load and tile store, respectively. The access records and the set A are inputs to the static overlap analysis; they do not require the compiler to perform additional checks at runtime.

The T-READ-TILE rule requires the operand to have a compatible tile-view type and the current execution context to match the tile shape. It also requires NoOverlap to hold for the current read. Consequently, the analysis rejects a read when it identifies a potentially concurrent write to an overlapping global-memory region.

The T-WRITE-TILE rule imposes the same tile-shape and execution context requirements. It additionally requires an exclusive borrow, expressed by $\Sigma(p) = \text{uniq}$, and checks the current write against other potentially concurrent accesses. This check rejects an overlapping read or write when the two accesses may execute concurrently. The exclusive-borrow requirement and the overlap check serve different purposes: the former enforces the borrowing discipline, while the latter checks conflicts between memory regions accessed by concurrent execution instances.

Safety Guarantees and Scope. These rules describe how RUSTILE checks tile-level global-memory accesses within the access patterns supported by its static analysis. The analysis relies on its ability to identify accessed memory regions and determine whether execution instances may run concurrently. It does not claim to resolve arbitrary dynamic indices, non-affine expressions, or indirect addressing when these prevent the analysis from determining the accessed regions. The current design also does not address shared-memory safety, synchronization correctness, atomic-operation semantics, or algorithmic errors. Instead, it focuses on detecting potentially

conflicting overlapping global-memory accesses across concurrent tile-level execution instances. The prototype performs these checks at compile time and does not insert additional runtime safety checks into the generated GPU code.

IV. PROTOTYPE IMPLEMENTATION

We implement a prototype compiler for RUSTILE that translates tile-based programs into GPU kernels. The implementation validates that Rust-inspired ownership and borrowing mechanisms can be integrated with tile-level GPU programming abstractions while leveraging existing high-performance GPU backends [31]–[33].

Figure 6 presents the overall compilation workflow. The compiler follows a three-stage architecture consisting of a frontend parser, a tile-aware static checker, and a backend code generation module. The frontend parses RUSTILE programs and constructs an abstract syntax tree (AST), which captures execution resources, tile views, and memory access operations. The AST serves as the primary intermediate representation throughout the compiler pipeline.

The static checker analyzes the AST and performs ownership- and borrowing-based verification at tile granularity [19], [20], [29]. It tracks tile-level borrow states, execution contexts, and memory regions to identify potentially conflicting accesses [8], [9]. Programs with unsafe overlapping accesses to global memory are rejected before code generation, ensuring that only statically verified programs are lowered to GPU backends.

The code generation module lowers verified ASTs into backend-specific GPU program representations [34]. It adopts a visitor-based translation mechanism to traverse the RUSTILE AST and generate corresponding backend constructs. A backend selector determines the target code generation strategy, allowing the same frontend language and safety analysis framework to support different tile-based GPU runtimes.

During lowering, tile-level abstractions such as tile loads, tile stores, and execution resources are translated into corresponding backend operations while preserving the semantics established by the static checker. In our prototype, the code generation infrastructure provides lowering paths for Triton and TileLang backends, demonstrating the extensibility of RUSTILE toward different tile-based GPU programming systems [10], [12].

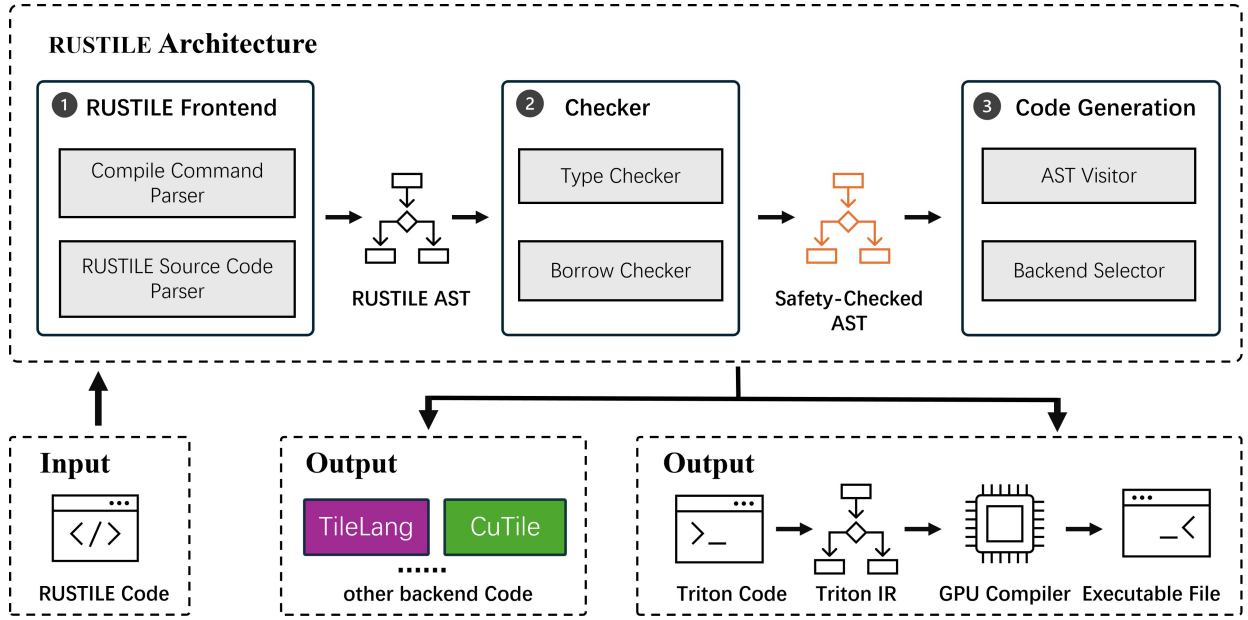
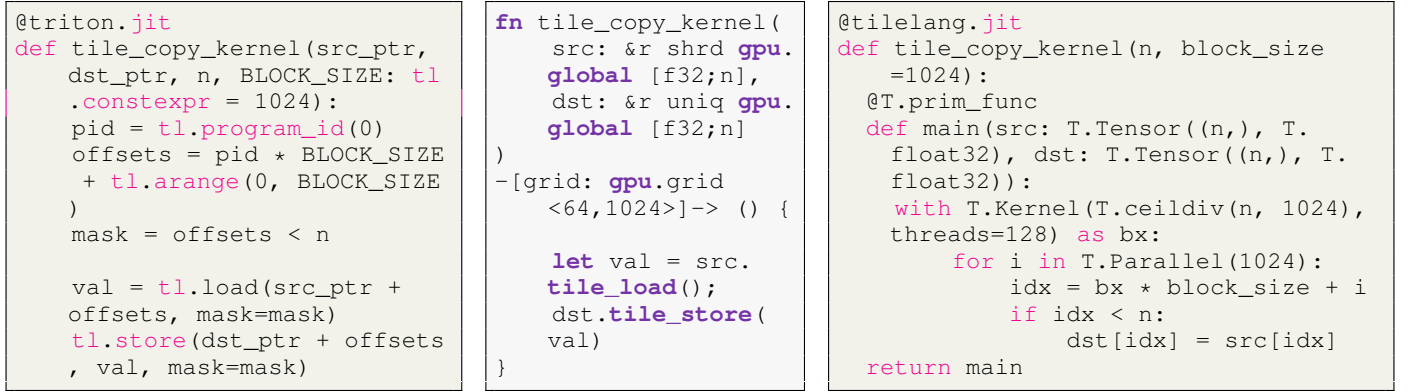


Fig. 6: The compilation workflow of RUSTILE.



(a) Triton backend

(b) RUSTILE program

(c) TileLang backend

Fig. 7: Backend extensibility of RUSTILE. The same RUSTILE program can be lowered into different tile-based GPU backend representations.

Figure 7 shows an example of multi-backend lowering in RUSTILE. The same tile-copy program is first expressed using RUSTILE’s tile-level abstractions and then translated into backend-specific GPU kernels.

V. EVALUATION

We evaluate RUSTILE along three dimensions: detection of conflicting tile-level global-memory accesses, runtime performance, and handling of representative overlapping-access patterns.

We ask three research questions:

RQ1: Can RUSTILE statically detect conflicting global-memory accesses to overlapping regions across concurrent tiles?

RQ2: Does RUSTILE preserve runtime performance compared with hand-written GPU kernels?

RQ3: How does RUSTILE detect tile-level global-memory access conflicts in representative GPU kernels?

We conduct all experiments on a server equipped with an NVIDIA GeForce RTX 3060 GPU (12 GB VRAM), running Ubuntu 24.04. The software environment uses Triton 3.5.0 and PyTorch 2.9.0. We focus on Triton-style tile-based kernels and conflicting global-memory accesses [3], [8], [10], [12], [17], [27].

A. Effectiveness (RQ1)

We evaluate whether RUSTILE can statically detect conflicting global-memory accesses to overlapping regions across concurrent tiles.

We use a curated benchmark of representative tile-based GPU kernels. The benchmark covers common computation patterns, including vector addition, matrix multiplication, softmax, and tiled transpose [3], [5], [10], [12]. These kernels expose explicit global-memory accesses, allowing RUSTILE to analyze potential conflicts at the same abstraction level as the programmer.

TABLE I: Categories of Injected Tile-Level Global-Memory Conflicts

Race Type	Count
Overlapping tile stores	3
Read-write tile conflicts	2
Partial tile overlaps	2
Output buffer reuse	1
Non-contiguous tile stores	1

For each correct kernel, we construct an unsafe variant by injecting controlled tile-level access conflicts while preserving the original computation structure. The injected defects cover overlapping tile stores, read-write tile conflicts, partial tile overlaps, output-buffer reuse, and non-contiguous tile stores [8], [9], [15], [23]. Table I summarizes the five categories and their counts. We focus on global memory because the current static analysis targets global-memory accesses.

Our benchmark contains nine correct kernels and nine corresponding variants with injected access conflicts. We apply the RUSTILE static checker to all 18 kernels and record whether it accepts or rejects each kernel. As shown in Table II, RUSTILE accepts all nine correct kernels and rejects all nine variants with injected conflicts. Thus, it detects all nine injected conflict cases in this benchmark, with no false positives among the nine correct kernels. These results demonstrate that RUSTILE detects the selected overlapping access patterns. However, this benchmark does not establish complete detection for arbitrary GPU programs.

Our overlap analysis also has limitations. Out-of-bounds checking and overlap detection address different properties. An access can remain within buffer bounds and still conflict with an access from another program instance. Conversely, an out-of-bounds access does not by itself establish that two accesses overlap. Our current heuristic may also miss repeated-index writes in scatter-style operations, especially when the buffer is large or the index expressions are complex. Therefore, our evaluation does not establish that the checker can detect all conflicts involving indirect or non-affine memory accesses.

Finally, we construct our benchmark by injecting controlled defects into representative kernels. We do not use a corpus of defects collected from real-world GPU applications. Our results therefore demonstrate detection of selected, known overlapping-access patterns rather than general effectiveness on real-world GPU defects.

B. Performance (RQ2)

We evaluate whether tile-aware ownership and borrowing enforcement affects the runtime performance of generated GPU kernels. We compare RUSTILE-generated kernels with handwritten Triton kernels to assess the runtime cost of our approach.

We conduct all experiments using the Triton backend. We choose Triton because it is a representative tile-based GPU programming system and provides a mature environment for performance evaluation [3], [4], [10], [12].

TABLE II: Static Detection of Overlapping Global-Memory Access Conflicts

Category	# Kernels	Outcome
Correct kernels	9	Accepted
Data-race variants	9	Rejected
Total	18	Complete separation

We compare RUSTILE-generated kernels against manually optimized Triton kernels under the same input sizes, tiling configurations, and execution settings [5], [10], [26]. Our benchmark covers representative workloads, including vector operations, matrix multiplication, softmax, transpose, attention, convolution, and stencil computation [3], [5], [10], [12].

For each workload, we evaluate multiple input sizes. We perform 10 warmup iterations before measurement. We then collect 30 execution-time measurements for each configuration. Table III reports the mean execution time, the 95% confidence interval, the normalized performance ratio, and the statistical test results.

We compute the performance ratio as follows:

$$\text{Ratio} = \frac{T_{\text{RUSTILE}}}{T_{\text{Handwritten}}}$$

A ratio below 1 indicates that the RUSTILE-generated kernel runs faster than the handwritten kernel. A ratio above 1 indicates that it runs slower. A ratio close to 1 indicates similar execution times.

We use the Wilcoxon signed-rank test to compare paired execution-time measurements. We use $p < 0.05$ as the threshold for statistical significance. However, statistical significance does not necessarily imply a substantial performance difference. For example, the large-input vector-addition configuration has a ratio of 1.001 and a p -value of 0.0183. The result is statistically significant, but the mean execution-time difference is only about 0.1%. Similarly, the matrix-multiplication configuration with input size 512 has a ratio of 0.988. This corresponds to a runtime reduction of about 1.2%, although the test also identifies a statistically significant difference. These examples show why we consider both the p -values and the performance ratios when interpreting the results.

Overall, the performance ratios range from 0.955 to 1.031 across the evaluated configurations. Most ratios are close to 1, and the observed differences are small in absolute terms. Some configurations run slightly faster with RUSTILE, while others run slightly slower. We therefore find no consistent evidence of a substantial runtime performance penalty in the evaluated workloads. Our design performs ownership and borrowing checks at compile time and does not insert additional runtime safety checks into the generated kernels [19]–[21].

C. Case Study: Detecting Overlapping Writes (RQ3)

We use a simplified case study to illustrate how RUSTILE detects overlapping tile-level global-memory accesses. We

TABLE III: Performance comparison between handwritten Triton kernels and automatically generated RUSTILE kernels

Kernel Name	Input Size	Handwritten (ms)	RUSTILE (ms)	Ratio	<i>p</i> -value
vecadd	16384	0.0151 ± 0.0026	0.0145 ± 0.0023	0.960	0.0778
	262144	0.0184 ± 0.0010	0.0182 ± 0.0008	0.989	0.128
	16777216	0.6089 ± 0.0013	0.6096 ± 0.0014	1.001	0.0183
matmul	128	0.0260 ± 0.0011	0.0260 ± 0.0009	1.000	0.307
	512	0.0572 ± 0.0011	0.0565 ± 0.0007	0.988	0.00111
	1024	0.2646 ± 0.0014	0.2637 ± 0.0013	0.997	0.00265
softmax	256	0.0359 ± 0.0014	0.0367 ± 0.0044	1.022	0.945
	1024	0.1113 ± 0.0023	0.1109 ± 0.0039	0.996	0.00361
	4096	0.4134 ± 0.0024	0.4129 ± 0.0047	0.999	0.00414
transpose	256	0.0140 ± 0.0018	0.0135 ± 0.0007	0.964	0.00914
	1024	0.0355 ± 0.0008	0.0352 ± 0.0013	0.992	0.0126
	4096	0.4277 ± 0.0013	0.4275 ± 0.0011	0.999	0.554
tile_copy	16384	0.0140 ± 0.0031	0.0137 ± 0.0044	0.979	0.103
	262144	0.0134 ± 0.0011	0.0131 ± 0.0007	0.978	0.0599
	16777216	0.4133 ± 0.0038	0.4129 ± 0.0016	0.999	0.411
row_reduction	256	0.0227 ± 0.0011	0.0234 ± 0.0012	1.031	0.000116
	1024	0.0598 ± 0.0012	0.0600 ± 0.0011	1.003	0.237
	4096	0.2066 ± 0.0013	0.2074 ± 0.0045	1.004	0.476
flash_attention	256 x 256 x 64	0.0245 ± 0.0020	0.0234 ± 0.0009	0.955	0.00144
	384 x 384 x 64	0.0278 ± 0.0019	0.0272 ± 0.0006	0.978	0.0935
	512 x 512 x 64	0.0309 ± 0.0009	0.0303 ± 0.0006	0.981	0.000309
fused_depthwise_conv2d	4 x 32 x 64 x 64	0.0506 ± 0.0021	0.0493 ± 0.0007	0.974	1.76×10^{-5}
	8 x 64 x 64 x 64	0.1192 ± 0.0012	0.1190 ± 0.0009	0.998	0.502
	8 x 64 x 128 x 128	0.4224 ± 0.0056	0.4222 ± 0.0021	1.000	0.0313
stencil_3d_multistep	32 x 64 x 64	0.1209 ± 0.0034	0.1195 ± 0.0021	0.988	0.0184
	64 x 64 x 64	0.1217 ± 0.0025	0.1206 ± 0.0017	0.991	0.0292
	64 x 128 x 128	0.5426 ± 0.0011	0.5442 ± 0.0064	1.003	0.266

Execution time is reported as mean $\pm$ 95% confidence interval (CI) over 30 measurement runs after 10 warmup iterations.

Ratio = $T_{\text{RUSTILE}}/T_{\text{Handwritten}}$. A ratio close to 1 indicates comparable performance.

Statistical significance is evaluated using the Wilcoxon signed-rank test, where $p < 0.05$ indicates statistically significant difference.

focus on a representative GPU kernel in which concurrent program instances write to the same output region.

Figure 8 shows a Triton kernel in which each program instance loads a tile and writes it to the same output region. The kernel uses the same offsets in every program instance. When multiple instances execute concurrently, their writes target overlapping global-memory locations, creating a write–write data race. Triton does not statically reject these overlapping writes [9].

Figure 8 also shows the same computation in RUSTILE. We annotate the output with a unique (`uniq`) tile-level reference, which requires exclusive write access to the referenced memory region. This annotation extends Rust-style ownership to tile-level memory accesses [29].

During compilation, RUSTILE detects that parallel blocks write to the same global-memory region through `tile_store`. These accesses violate the uniqueness constraint on the output reference. The checker therefore rejects the program before execution and reports the overlapping writes, as shown in Figure 8.

This example illustrates how RUSTILE detects a specific overlapping-write pattern at compile time. It shows how tile-aware ownership checking can identify a potential data race before the kernel runs. However, this simplified case study does not establish that RUSTILE detects all possible overlapping-access conflicts in GPU programs.

```
@triton.jit
def data_race_write(input_ptr,
                    output_ptr,
                    BLOCK_SIZE:
tl.constexpr = 256):
    pid = tl.program_id(0)
    offsets = tl.arange(0, BLOCK_SIZE)
    data = tl.load(input_ptr + offsets)
    tl.store(output_ptr + offsets, data)
```

```
#[block_size(256)]
fn data_race_write<r: prv>(
    input: &r shrd gpu.global [f32; 256],
    output: &r uniq gpu.global [f32; 256]
) -[grid: gpu.grid<2, 256>]-> () {
    let data = input.tile_load();
    let out = data;
    output.tile_store(out)
}
```

```
error: Data race detected
tile_store to 'output' causes overlapping writes across parallel blocks.
```

Fig. 8: Comparison between a buggy Triton kernel and its RUSTILE counterpart with compile-time overlapping-write detection.

VI. DISCUSSION

A. Scope and Design Rationale

RUSTILE focuses on preventing data races caused by conflicting tile-level accesses to global memory. This scope is motivated by the observation that global-memory interactions represent the primary source of cross-program-instance interference in tile-based GPU programming models. Existing tile-based languages abstract away thread-level execution details and allow programmers to reason directly about tiles as the basic units of computation. Therefore, enforcing safety at the tile level naturally requires a memory model that matches this abstraction boundary.

Other GPU memory hierarchies, such as shared memory and synchronization primitives, involve lower-level execution mechanisms that are typically hidden by tile-based abstractions. Supporting these mechanisms would require exposing additional hardware-specific details, including intra-tile synchronization behavior, thread cooperation patterns, and atomic operation semantics. Integrating such features into the current ownership model would significantly increase the complexity of the language design and type system.

Therefore, RUSTILE deliberately targets global-memory safety as the first step toward safe tile-based GPU programming. By aligning ownership reasoning with the abstraction level exposed to programmers, RUSTILE demonstrates that Rust-style safety guarantees can be effectively lifted from individual memory locations to tile-level regions without sacrificing programmability or performance.

B. Future Directions

Several directions remain for extending RUSTILE beyond its current design. First, we plan to investigate richer GPU memory models, including shared-memory accesses, synchronization operations, and atomic primitives. These extensions require developing more fine-grained ownership and borrowing rules that can reason about interactions within and across tiles while preserving the simplicity of the tile-level abstraction.

Second, we aim to extend RUSTILE with broader backend support and optimize the lowering infrastructure for additional tile-based GPU frameworks. Although the current prototype demonstrates backend extensibility through multiple lowering

paths, supporting more execution models would further improve the portability of the language.

Finally, we plan to evaluate RUSTILE on larger-scale GPU applications and more complex workloads. Such evaluations can provide further evidence of the scalability of tile-aware ownership checking and identify additional programming patterns that require advanced safety reasoning.

VII. RELATED WORK

GPU Data-Race Analysis and Static Safety. GPU data-race analysis has been extensively studied in prior work. Dynamic tools, such as NVIDIA Compute Sanitizer (formerly CUDA-MEMCHECK), detect conflicting memory accesses at runtime [17], while static verification frameworks, such as GPUVerify, prove race-freedom properties for CUDA and OpenCL kernels before execution [9]. These approaches provide useful guarantees, but they either incur runtime overhead or rely on restrictive assumptions about kernel structure and synchronization. Inspired by these principles, ownership-based type systems provide another direction for preventing data races through static reasoning. Rust demonstrates that ownership and borrowing can enforce strong compile-time safety in practice [21], [29]. However, these approaches typically assume thread-level execution and do not directly model GPU-specific abstractions such as tiles.

Tile-Based GPU Programming and Our Position. Tile-based abstractions have become an increasingly important approach for high-level GPU programming. Systems such as Triton and related frameworks allow programmers to express computation over tiles, improving both productivity and performance [3], [5], [10]–[12]. However, existing tile-based programming systems generally lack explicit language-level mechanisms to express ownership or non-aliasing constraints at tile granularity. We position RUSTILE at the intersection of these directions. We bring ownership-based reasoning to tile-level abstractions and enforce safety through a tile-aware type system. This design enables compile-time detection and prevention of overlapping accesses to global memory while preserving compatibility with existing GPU backends.

VIII. CONCLUSION

We presented RUSTILE, a tile-based GPU programming language that combines high-level tile abstractions with strong

static memory safety guarantees. By lifting Rust-style ownership and borrowing disciplines to tile granularity, RUSTILE enables compile-time detection of conflicting global-memory accesses between concurrently executing tiles while preserving the productivity benefits of tile-based programming. We implement a prototype compiler that translates RUSTILE programs into existing tile-based GPU backends and evaluate its effectiveness and performance on representative GPU kernels. The results show that RUSTILE successfully prevents tile-level data races through static analysis and achieves performance comparable to handwritten GPU implementations without introducing runtime overhead. Overall, RUSTILE demonstrates that productivity, static safety, and high performance can coexist in a unified tile-based GPU programming framework, providing a foundation for future safe and extensible GPU programming abstractions.

REFERENCES

- [1] S. Heldens, P. Hijma, B. Van Werkhoven, J. Maassen, and R. V. Van Nieuwpoort, “Lightning: Scaling the gpu programming model beyond a single gpu,” in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 492–503.
- [2] P. Hijma, S. Heldens, A. Sclocco, B. Van Werkhoven, and H. E. Bal, “Optimization techniques for gpu programming,” *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–81, 2023.
- [3] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, “{TVM}: An automated {End-to-End} optimizing compiler for deep learning,” in *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, 2018, pp. 578–594.
- [4] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, “Mlir: Scaling compiler infrastructure for domain specific computation,” in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.
- [5] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, “Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions,” *arXiv preprint arXiv:1802.04730*, 2018.
- [6] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.
- [7] C. Peng, “On the correctness of gpu programs,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 443–447.
- [8] D. Liew, T. Cogumbreiro, and J. Lange, “Sound and partially-complete static analysis of data-races in gpu programs,” *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 2434–2461, 2024.
- [9] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, “Gpuverify: a verifier for gpu kernels,” in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2012, pp. 113–132.
- [10] P. Tillet, H.-T. Kung, and D. Cox, “Triton: an intermediate language and compiler for tiled neural network computations,” in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019, pp. 10–19.
- [11] A. Shanbhag, B. W. Yogatama, X. Yu, and S. Madden, “Tile-based lightweight integer compression in gpu,” in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1390–1403.
- [12] L. Wang, Y. Cheng, Y. Shi, Z. Tang, Z. Mo, W. Xie, L. Ma, Y. Xia, J. Xue, F. Yang *et al.*, “Tilelang: A composable tiled programming model for ai systems,” *arXiv preprint arXiv:2504.17577*, 2025.
- [13] Y. Shi, Z. Yang, J. Xue, L. Ma, Y. Xia, Z. Miao, Y. Guo, F. Yang, and L. Zhou, “Welder: Scheduling deep learning memory access via tile-graph,” in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 701–718.
- [14] C. Kim, J. Kim, S. Kim, D. Kim, N. Kim, G. Na, Y. H. Oh, H. G. Cho, and J. W. Lee, “Typed architectures: Architectural support for lightweight scripting,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 77–90.
- [15] A. Betts, N. Chong, A. F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, and J. Wickerson, “The design and implementation of a verification technique for gpu kernels,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 37, no. 3, pp. 1–49, 2015.
- [16] K. Kojima and A. Igarashi, “A hoare logic for gpu kernels,” *ACM Transactions on Computational Logic (TOCL)*, vol. 18, no. 1, pp. 1–43, 2017.
- [17] D. Guide, “Cuda c++ programming guide,” *NVIDIA*, July, 2020.
- [18] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL programming guide*. Pearson Education, 2011.
- [19] S. Klabnik and C. Nichols, *The Rust programming language*. No Starch Press, 2023.
- [20] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, “Rustbelt: Securing the foundations of the rust programming language,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–34, 2017.
- [21] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, “Stacked borrows: an aliasing model for rust,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, 2019.
- [22] K. Kojima, A. Imanishi, and A. Igarashi, “Automated verification of functional correctness of race-free gpu programs,” in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2016, pp. 90–106.
- [23] M. Zheng, G. Agrawal, F. Qin, and V. T. Ravi, “Grace: a low-overhead mechanism for detecting data races in gpu programs,” *Sigplan Notices*, 2011.
- [24] B. Köpcke, S. Gorlatch, and M. Steuwer, “Descend: a safe gpu systems programming language,” *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 841–864, 2024.
- [25] M. Steuwer, T. Koehler, B. Köpcke, and F. Pizzuti, “Rise & shine: Language-oriented compiler design,” *arXiv preprint arXiv:2201.03611*, 2022.
- [26] B. Hagedorn, J. Lenfers, T. Koehler, X. Qin, S. Gorlatch, and M. Steuwer, “Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, pp. 1–29, 2020.
- [27] D. Lustig, S. Sahasrabudhe, and O. Giroux, “A formal analysis of the nvidia ptx memory consistency model,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 257–270.
- [28] P. E. de Vilhena, O. Lahav, V. Vafeiadis, and A. Raad, “Extending the c/c++ memory model with inline assembly,” *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 1081–1107, 2024.
- [29] A. Bychkov and V. Nikolskiy, “Rust language for gpu programming,” in *Russian Supercomputing Days*. Springer, 2022, pp. 522–532.
- [30] S. Li, J. Zhao, Y. Liu, S. Zhang, L. Chen, Y. Li, Y. Zhang, Z. Li, R. Zhou, X. Shi *et al.*, “From threads to tiles: T2t, a compiler for cuda-to-npu translation via 2d vectorization,” in *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2026, pp. 362–374.
- [31] M. Levental, A. Kamatar, R. Chard, K. Chard, and I. Foster, “nelli: a lightweight frontend for mlir,” *arXiv preprint arXiv:2307.16080*, 2023.
- [32] A. Kothari, A. R. Noor, M. Xu, H. Uddin, D. Baronia, S. Baziotis, V. Adve, C. Mendis, and S. Sengupta, “Hydride: A retargetable and extensible synthesis-based compiler for modern hardware architectures,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 514–529.
- [33] A. Wang, X. Yi, and Y. Yan, “Upir: toward the design of unified parallel intermediate representation for parallel programming models,” in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2022, pp. 530–531.
- [34] F. Pizzuti, M. Steuwer, and C. Dubach, “Generating work efficient scan implementations for gpus the functional way,” in *European Conference on Parallel Processing*. Springer, 2022, pp. 335–349.