Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Automating Separation Logic Ghost Operations for the Interactive Optimization of Array Programs: An LLM Case Study

Master Thesis

Morel Elian

December 16, 2025

Advisors: Dr. Arthur Charguéraud, Pr. Cédric Bastoul, Pr. Peter Müller

Department of Computer Science, ETH Zürich

Abstract

OptiTrust is a source-to-source transformation framework designed to bridge the gap between high-level algorithmic specifications and high-performance implementation. To guarantee that the optimization process introduces no bugs, Optitrust exploits semantic-preservation criteria that rely on a type system based on Separation Logic.

In practice, this type system requires pre- and post-conditions on functions, loop invariants and *ghost operations*. These ghost operations guide the entailment checking algorithm by explicitly manipulating logical views of memory resources without affecting the program execution. On complex array-intensive programs, manually managing these resources becomes a bottleneck due to high verbosity.

To resolve this issue, this thesis introduces Autofocus, an algorithmic elaboration pass designed to automatically infer the necessary ghost operations for array manipulation. By defining a canonical ordering for iterated resources and automating index instantiation, Autofocus allows the type system to deduce how to extract specific permissions (e.g., a cell or a slice) from a larger resource collection (e.g., a full matrix). We demonstrate that this mechanism drastically reduces the annotation burden. On standard linear algebra kernels, manual ghost annotations account for over 70% of the code size. With Autofocus, 80% to 100% of these annotations are automatically inferred.

We leverage the Autofocus mechanism to achieve *shape-level* verification of a realistic workload: the inference pipeline of a Large Language Model (LLM). We successfully established memory safety guarantees for the forward pass of a `llama2.c` implementation. This program comprises approximately 300 lines of code, for which the tool automatically generated over 1,200 lines of specification. Finally, we demonstrate the optimization potential of OptiTrust on this LLM program. By implementing a script that applies transformations such as token batching, loop fission, and strip mining, we improved data locality and parallelism, achieving a 4.6× speedup in token throughput compared to the baseline.

Overall, this work a first but critical step towards producing a formally verified, highly optimized implementation of a LLM inference code.

Acknowledgments

I would like to begin these acknowledgments by thanking Arthur Charguéraud, first for proposing this topic and then for his your constant involvement throughout the project. Arthur, thank you for your intense support, for always taking the time to explain countless concepts across a wide range of subjects, and for your guidance regarding my future career.

I also want to thank Cédric Bastoul, who provided valuable insights into the world of LLMs. Cédric, you saved us precious time regarding common optimizations when I was just discovering the field, sparing us the difficult experience of going down the wrong path for weeks.

I thank the entire team working around OptiTrust—Thomas Khoeler, Yanni Lefki, and Julien De Castelnau. You contributed significantly to this work, not only through your knowledge of Optitrust also through the many informal discussions we had. More generally, I would like to thank the Camus team for making me feel so warmly welcomed at Inria Strasbourg.

Finally, I wish to thank my girlfriend, my family, and my friends, who patiently listened to me grumble, doubt, and complain (in that order of appearance) when things were not working as expected, and who, beyond that, remain an unwavering source of support in all my adventures.

Contents

Contents **iii**

1 Introduction **1**
1.1 Problem Statement . . . 1
1.2 Demonstrating the Interest of OptiTrust for LLM Inference . . 4
1.3 Related Work . . . 6
1.4 Contribution . . . 7
1.4.1 Autofocus Presentation . . . 7
1.4.2 Enhancement of the OptiTrust Framework with Minor Features . . . 8
1.4.3 LLM Optimization . . . 8

2 Background **10**
2.1 Separation Logic . . . 10
2.2 OptiTrust . . . 13
2.2.1 OptiTrust Overview . . . 13
2.2.2 Optitrust Typing System . . . 16

3 Autofocus **22**
3.1 Motivation . . . 22
3.1.1 Description of a Focus . . . 22
3.1.2 Focusing on Array Slices . . . 24
3.2 Handled Cases and Limitations . . . 25
3.2.1 Description of Covered Cases . . . 25
3.2.2 Restrictions . . . 25
3.3 Autofocus Algorithm . . . 26
3.3.1 Problem Formulation . . . 26
3.3.2 Phase 1: Reordering . . . 28
3.3.3 Phase 2: Index Instantiation . . . 29
3.4 Integration in OptiTrust . . . 31

3.4.1 Typing Pass 31
3.4.2 Substitution 32
3.4.3 Range Extraction 32
3.4.4 Term Unification 33
3.4.5 Conversion to an Internal Representation 33
3.4.6 Focus List Construction 34
3.4.7 Sequence Retyping 34
3.4.8 Elaboration 35
3.5 Results 36
3.5.1 Illustrative Example 36
3.5.2 General Results 38

4 OptiTrust Extensions for Improved Matrix Handling **40**
4.1 Normalization of Matrix Terms 40
4.2 Matching Extension for Uninlining 42
4.3 Tiling Access Transformation 43

5 Large Language Models (LLMs) **47**
5.1 Introduction 47
5.1.1 Purpose of the Case Study 47
5.1.2 Overview of LLMs 47
5.2 Fitting the Original Code into OptiTrust 49
5.2.1 Llama2.c 49
5.2.2 Code Adjustments for OptiTrust 49
5.3 Transformations 50
5.3.1 Token Batching 50
5.3.2 Recognizing MatMul 54
5.3.3 Strip Mining 55
5.4 Results 62

6 Conclusion **63**
6.1 Summary of Contributions 63
6.2 Perspectives and Future Work 64

Bibliography **66**

Appendix **69**

Chapter 1

Introduction

1.1 Problem Statement

Program Optimization: A Difficult Task

When a programmer wishes to implement an algorithm with high performance, one possibility is to start by writing a short, unoptimized code. The purpose of this code is to remain close to the mathematical flow of the algorithm, which allows the programmer to focus on the correctness of his implementation.

Then comes the optimization phase, where the programmer wishes to produce a faster implementation, however this phase can quickly become problematic. By applying common optimization techniques (such as caching, loop tiling, etc.), one quickly transitions from a simple code of a few dozen lines to an inextricable tangle of several hundred lines that becomes very difficult to get right and to maintain to maintain.

Certain tools have been created to facilitate this task for the programmer. Compilers like LLVM [17] or GCC [24] provide an increasing number of embedded optimizations: the programmer writes a C code close to the mathematical specification, and the compiler is in charge of recognizing common patterns, and generating efficient machine code. However, general-purpose compilers often fail to automatically perform aggressive structural transformations. Consequently, the programmer is forced to manually rewrite the code into the complex, optimized version mentioned earlier to help the compiler achieve peak performance.

The limitations of fully automated, generalist compilers have motivated research on semi-automated compilers. Specialized compilers targeting specific domains have existed for decades. Later, around 2008, the concept of utilizing transformation scripts became visible, notably within the polyhedral model community. These ideas were eventually combined—a specialized com-

piler guided by a user script—and popularized by tools such as Halide [23]. Originally developed to increase performance for image and video processing algorithms, Halide has inspired other systems like TVM [9], which is dedicated to AI.

Program Correctness: A Major Challenge

Another major challenge, which applies to the tools mentioned above, is the following: how can we guarantee that the output code retains the same semantics as the original code? Often, programmers rely extensively on testing their implementation. While they realize this is insufficient to strictly guarantee correctness, constraints on time or expertise lead them to satisfy themselves with correct outputs for these tests. The problem with this method is that it does not provide a strict guarantee on the behavior of the code: an input not foreseen by the programmer could yield an incorrect output.

An alternative approach is to use formal methods to theoretically guarantee the correctness of an algorithm; one of the most promising methods is called *Proof-Carrying Code*. The idea is to annotate the input code with a *specification* and then derive a proof showing that the output code satisfies this specification. With Proof-Carrying Code, the user obtains a strict guarantee that the program will always respect the intented specification. In practice, one verify that the program follows the mathematical specification attached to it. Specifications can be expressed by writing *contracts*, which contain:

- A precondition, which describes the conditions on the input data under which the program is executed.
- A postcondition, which describes the state of the program at the end of execution, provided the preconditions were respected.

To give an illustrative example, consider the Knapsack problem. The precondition would describe the weights and values of available items and the maximum capacity; the postcondition would state that the resulting allocation yields the maximal value within the weight limit. Unfortunately, function contracts alone are often insufficient for the tool to construct the proof, and additional annotations must be provided. These include loop invariants, which allow the tool to determine the correctness of iterative structures, and *ghost code*. Ghost code consists of auxiliary instructions introduced solely to facilitate the verification process. It allows the programmer to track information or express properties that are not explicitly present in the original algorithm.

Proof-Carrying Code has been evolving for two decades: introduced by Necula [21], the concept was extended by Bannwart and Müller [1] for Java compilation, refined by Müller and Nordio [20] to handle abrupt termination, and further developed by Barthe et al. [2].

Figure 1.1: Optitrust global flow

OptiTrust: Performance Meets Correctness

OptiTrust [8, 3] is a source-to-source transformation framework for refining proof-carrying code. OptiTrust is developed in the CAMUS team at Inria Strasbourg and aims to address both of the complex problems raised above.

It uses ideas derived from proof-carrying code to, on one hand validate previously annotated input code and, on the other hand, derive proof elements for the output code, thereby validating that the output conforms to the specification associated with the input code.

Concretely, the user provides an input code annotated in Separation Logic. The user gives the contracts for each function as well as other annotations (loop invariants and ghost operations), along with an optimization script in the form of a sequence of transformations. OptiTrust then takes charge of:

- Verifying that the input code conforms to the input specification.
- Applying the transformations one by one, while maintaining a proof of correctness at each step and propagating the annotations so that each subsequent transformation can use them for the optimizations it performs. They can then use information that an automatic compiler would not have access to.
- Producing the final optimized output code.

It is important to note that OptiTrust implements proof-carrying code at two distinct levels. It can target *full functional correctness*, where the typing of the output code is sufficient to guarantee its correctness with respect to the specification. Alternatively, it can target *shape-only* typechecking. This mode guarantees strong properties such as memory safety and race-freedom, but requires that each transformation explicitly checks for semantic preservation, using criteria that exploit the shape information.

OptiTrust creates a synergy between the annotations initially written for correctness and the optimization potential concretized by the transformations: transformations can take advantage of the annotations to allow specific optimizations that would not have been performed by a standard compiler. The feasibility of this methodology has been previously illustrated in the original OptiTrust paper, which applied the tool to two kernels: a matrix-matrix multiplication and an image-processing routine from OpenCV.

```
float matvec(float* x, float* y, float *W, int m, int n){
    __writes("x ~> UninitMatrix1(m)");
    __reads("y ~> Matrix1(n), W ~> Matrix2(m,n)");
    // computes x = W * y
}
```

Figure 1.2: Example of OptiTrust syntax for specifying a contract

1.2 Demonstrating the Interest of OptiTrust for LLM Inference

In this master thesis, I aimed to showcase the capabilities of OptiTrust, and the advantages outlined above through a substantial, realistic case study. As mentioned above, the previous case studies were conducted on small codebases (less than 10 lines of code), a central objective of this internship was therefore to evaluate how OptiTrust performs on code of significantly larger size and complexity.

To that end, we selected LLM inference as our target application. It provides a highly relevant and timely example of modern performance challenges. The code we worked on is a minimal, unoptimized C implementation of roughly 200 lines that captures the core mechanism underlying systems such as ChatGPT or Gemini: The user sends to a previously trained model an input prompt, then the model will perform *token inference*: it generates a response token by token, where a token can be seen as a part of a word. Despite its compactness, the program consists of nested loops calling several computation kernels, making it sizable for a 6-month Master thesis while still rich in optimization opportunities.

Targeted Guarantee Level

Regarding correctness, our objective was to establish semantic guarantees for both the baseline code and the transformations applied to it. The level of assurance targeted corresponds to the notion of *shapes* [12]. As discussed earlier, contracts in OptiTrust require specifying the structural shape of the data manipulated by the program. Figure 1.2 shows an example of such annotations on a small code snippet.

In this example, the contract describes only the structural properties of the data rather than its values: for instance, `x` is declared as a vector of length `m`, `y` as a vector of length `n`, and `W` as a matrix of dimensions $\texttt{m} \times \texttt{n}$. At this level, we can ensure, for example, that no uninitialized memory is ever read, that all accesses remain within array bounds, and that accesses are strictly restricted to the memory cells explicitly mentioned in the preconditions, respecting the permissions specified.

Function-by-Function Annotations

In practice, to verify a program of this scale, we rely on *modular verification*. This approach consists of associating a specification (contract) with each function. The verification process then proceeds in two complementary steps:

1. The body of each function is verified to ensure it satisfies its postcondition, assuming its precondition holds.
2. When a function call occurs, the verification tool relies on the callee's specification, ignoring its internal implementation.

When a function f calls a function g, the tool checks that the current state satisfies the precondition of g. If successful, it then updates the current state by consuming the precondition and producing the postcondition of g.

However, drafting contracts is not sufficient on its own. A major challenge arises when there is a mismatch between the structure of the *current state* (how resources are currently described in the proof logic) and the structure required by the precondition of the next operation. This issue is not limited to function calls but also occurs within loops. For example, consider a scenario where the current state describes ownership of a complete array, but the function to be called requires access to a single specific cell within that array. Although owning the full array implies owning its elements, the verification tool may not automatically decompose this resource to match the specific precondition. To resolve this, the programmer must explicitly insert *ghost operations*. These instructions do not alter the actual execution but guide the verification tool to rearrange the logical view of the resources: they isolate the specific cell from the rest of the array, thereby explicitly exhibiting that the required resource is available to satisfy the function's contract.

Ghost Operations Can Become Quite Cumbersome

Figure 1.3 illustrates a concrete instance of this issue within the LLM inference code. The snippet displays the main function making calls to various computation kernels (such as `rmsnorm`). Regardless of the specific implementation of these kernels, their contracts strictly define the required input state. A mismatch arises here: while the calling context holds permissions for the full arrays (`embedding` and `mha_norm_weight`), the kernel's precondition expects only a specific *slice*. Consequently, as shown in the figure, ghost operations must be inserted to explicitly *focus* the resource view onto the relevant portions, thereby satisfying the kernel's contract.

As this example illustrates, writing such ghost code quickly becomes verbose and error-prone. For one line of code (the call to rmsnorm), we added two complex ghost operations. In a program of approximately 200 lines, the number of required annotations grows dramatically, making the code

```
const __ghost_fn __ghost_pair_2 = __ghost_begin(
    ro_group_focus,
    "i := 1, items := fun (l: int) -> for i1 in 0..e_dim -> "
    "&mha_norm_weight[MINDEX2(l_count, e_dim, l, i1)] ~> Cell");

const __ghost_fn __ghost_pair_1 = __ghost_begin(
    ro_group_focus,
    "i := 1, items := fun (l: int) -> for q in 0..q_head_count -> "
    "for h in 0..head_dim -> for e in 0..e_dim -> "
    "&mha_q_weight[MINDEX4(l_count, q_head_count, head_dim, "
    "e_dim, l, q, h, e)] ~> Cell");

rmsnorm(
    e_dim,
    &mha_norm[MINDEX2(seq_len, e_dim, i, 0)],
    &embedding[MINDEX2(seq_len, e_dim, i, 0)],
    &mha_norm_weight[MINDEX2(l_count, e_dim, l, 0)],
    epsilon);
__ghost_end(__ghost_pair_1);
__ghost_end(__ghost_pair_2);
```

Figure 1.3: Ghost operations required to focus resources before calling `rmsnorm`.

difficult to write and maintain. This motivated us to design a mechanism capable of inferring some of these obvious annotations automatically, thereby reducing the annotation overhead for the programmer.

1.3 Related Work

Significant research has been conducted on array analysis within the framework of Separation Logic, primarily focusing on permission inference. Notably, Calcagno et al. [6] developed *bi-abduction*. This technique allows for the automatic inference of the "anti-frame", i.e., the missing portion of a precondition required to make a proof succeed. More recently, specific inference techniques for arrays were developed by Dohrau et al. [13], who proposed an algorithm to infer the minimal permissions required for array accesses within loop bodies.

Our proposition differs fundamentally from these works. We do not aim to infer missing permissions or loop invariants; rather, we aim to strengthen the entailment checking algorithm itself. Specifically, we target the resolution of entailments involving *iterated stars*.

In the broader context of Separation Logic, several methods have been developed to enhance entailment checking. *Rule-based methods* are particularly prominent:

- **SLEEK** [10] is an automated solver that supports user-defined rules. It relies on proof search and backtracking to explore possible derivations.

- **Diaframe** [19] targets the verification of concurrent programs. It introduces a "hint" mechanism, to guide the verification process.

While these tools provide powerful general-purpose mechanisms, they often rely on search heuristics or backtracking, which can become costly or unpredictable. In contrast, Autofocus is designed specifically for the domain of arrays and iterated stars. Instead of a general proof search, we employ a deterministic, algorithm-based approach to resolve the combinatorial complexity of reshaping array permissions. This allows Autofocus to remain non-backtracking and efficient.

1.4 Contribution

Starting from a case study to demonstrate the interest of OptiTrust, we were able to realize one of its limitations: the heaviness of annotations on programs of a larger size than those on which the tool had previously been applied. I have proposed a systematic approach, called autofocus, for reducing the amount of ghost operations thereby resolving this problem and demonstrated its utility on a realistic case study while utilizing the power of the OptiTrust tool.

1.4.1 Autofocus Presentation

Autofocus attempts to provide an answer to the following problem: suppose a function demands a specific resource A, but the current context does not explicitly contain it. However, the context *does* contain a resource B that covers a superset of the cells required by A. The challenge is: can we automatically synthesize the formal steps, specifically, a sequence of "focus" ghost operations, that justify extracting A from B?

We have defined an algorithm allowing us to exhibit a list of ghost operations that enable the transition from the current state A to state B. This algorithm has been implemented and fully integrated into the OptiTrust framework. It has allowed for the automation of program specification that was previously out of reach or demanded an extremely substantial effort from the user. Regarding the Trusted Code Base (TCB), Autofocus was designed to rely primarily on the original OptiTrust typechecker. However, a small part of the logic remains unverified: we currently use unproven ghost operations to assert that one nest of iterated stars is a valid permutation of another. We therefore trust the implementation responsible for generating these index permutations. All others elaborated material present in B is processed by the original typechecker. Furthermore, on a broader note regarding the TCB, OptiTrust currently leaves certain arithmetic trivialities as *proof* obligations, as the connection to an SMT solver has not yet been implemented.

1.4.2 Enhancement of the OptiTrust Framework with Minor Features

Still within the context of LLM optimization, our case study also revealed a few additional limitations in the current framework.

- **Expression Normalization.** When multi-dimensional arrays are flattened, the index of a specific cell can be computed via several distinct yet equivalent formulas. In OptiTrust, these index calculations rely on macros, which may be nested. A challenge arises when the current state and the callee's precondition use syntactically different formulas that point to the same location. For instance, the formula:

  $$\text{for } i \text{ in } [0 \ldots n] \to (\&x[\text{MINDEX2}(m, n, 0, 0)])[\text{MINDEX1}(n, i)]$$

  and the formula:

  $$\text{for } i \text{ in } [0 \ldots n] \to x[\text{MINDEX2}(m, n, 0, i)]$$

  are equivalent, where the for i in $[0 \ldots n]$ represents *iterated stars*. However, the second form facilitates simpler syntactic matching and is essential for the Autofocus algorithm to function correctly. To address this, we introduced a normalization mechanism. This step canonicalizes these expressions, allowing the tool to systematically recognize their mathematical equivalence despite their syntactic differences.

- **Transformations.** On the transformation side, we identified the need for code transformations that had not been needed in previously handled case studies. In particular, I implemented data tiling, which proved essential in the context of LLM inference and has now been integrated into the tool.

1.4.3 LLM Optimization

The value of these new features, and, more broadly, of OptiTrust as a tool capable of optimizing non-trivial programs while preserving semantic guarantees, was demonstrated on an LLM inference workload.

- First, we established shape-level semantic guarantees on the baseline implementation, made possible through the automatic generation of ghost operations by Autofocus.

- Second, we showed that the main optimizations relevant to this code could be expressed naturally using OptiTrust, confirming that the tool can separate the unoptimized code and its specification from the description of its optimizations.

We emphasize that this work remains ongoing. To fully validate the interest of Autofocus, it is necessary to extend it so that it can operate not only in the shape-only mode but also in the fully verified mode of OptiTrust, which supports complete functional specifications. The thesis of Guillaume Berthelon demonstrated that both the transformation system and the underlying framework can be lifted from shape-level guarantees to full functional correctness.

Chapter 2

Background

2.1 Separation Logic

Separation logic [22] [7] is an extension of Hoare logic [15]; it allows one to reason about programs that manipulate shared mutable data structures. It is based on the notion of *partial heaps*, *fractional permissions*, and enables local reasoning thanks to the *frame rule*.

Program State

In reasoning about imperative programs, the program state typically consists of two components: the heap and the store. The *heap* models dynamically allocated memory, where each address corresponds to a memory cell containing a value, while the *store* maps program variables to values. From now on, and as is customary in separation logic, we will use the term *heap* to refer to a fragment of the program's global heap.

Separation Logic as an Extension of Hoare Logic

In Hoare Logic [15], the behavior of a computational term t is formally defined by a triple $\{P\}t\{Q\}$, where P represents the precondition describing the input state, and Q represents the postcondition describing the output state.

Separation Logic adopts this triple notation but introduces a fundamental semantic distinction. While assertions in classical Hoare Logic describe the *entire* global memory state, Separation Logic assertions describe only the specific portion of the heap that the term t accesses or modifies during its evaluation. The semantics are defined as follows: if the precondition P holds prior to the execution of t, and if t terminates, then the postcondition Q is guaranteed to hold. Notably, the postcondition Q specifies not only the final

state of the memory but also the value returned by t. This isolation of the memory footprint is what enables *local reasoning*.

Frame Rule

The frame rule expresses that one only needs to consider the part of memory that a term modifies. Formally, from $\{P\}C\{Q\}$, we can infer $\{P * R\}C\{Q * R\}$, where the resources in R are disjoint from those in P and Q. This principle ensures scalability and reusability in proofs.

Assertions in Separation Logic

Assertions describe properties of both let-bound variables and memory cells. They specify which parts of memory are allocated, how they are structured, and what values they contain.

Points-to Assertion ($x \mapsto v$) Indicates that variable x is a memory location of a heap cell containing value v, capturing ownership of a specific memory location. The value v may either denote a concrete value stored at that location or simply describe the shape or structure of the data, for instance, $x \rightsquigarrow \mathsf{Cell}$.

Separating Conjunction ($P * Q$) Asserts that the heap can be divided into two disjoint sub-heaps, one satisfying P and the other Q. This supports independent reasoning about separate memory segments.

Iterated Separating Conjunction ($\bigstar_{i=1}^{n} P_i$) Asserts that the heap can be partitioned into n independent sub-heaps, each satisfying the corresponding assertion P_i. This operator provides a compact way to describe distinct elements within aggregate structures, such as arrays. For example, the resource on an array a of size n can be described as:

$$\bigstar_{i=1}^{n} \&a[i] \rightsquigarrow \mathsf{Cell}$$

For the sake of conciseness, we omit the address-of operator & in these formulas over iterated variables, writing simply:

$$\bigstar_{i=1}^{n} a[i] \rightsquigarrow \mathsf{Cell}$$

Separating Implication or Magic Wand ($P \mathrel{-\!\!*} Q$) States that given a disjoint heap satisfying P, then the combined heap satisfies Q. Useful for reasoning about resources that may later be extended or combined.

Fractional Permissions

Fractional permissions [5] extend the notion of ownership in separation logic by allowing a heap location to be accessed in parts. Instead of requiring full ownership to read a value, a fraction of the permission suffices.

While this concept is widely used in concurrent separation logic, in the context of this work, we do not target general thread concurrency. Instead, we focus on *structured parallelism* expressed through parallel loops. Within this framework, fractional permissions serve two primary purposes:

- **Parallelism:** They allow multiple iterations of a parallel loop to simultaneously read from the same memory location.
- **Aliasing:** They enable passing the same resource multiple times as a read-only argument to a function call. For instance, a function `compare(A, A)` that reads from the same array `A` via two distinct arguments.

Concretely, the standard points-to assertion

$$x.f \mapsto v$$

is annotated with a fractional permission $q \in (0, 1]$:

$$x.f \overset{q}{\mapsto} v$$

A full permission ($q = 1$) grants both read and write access. This full permission can be split into smaller fractions ($0 < q < 1$) which grant read-only access. This splitting mechanism can be generalized to an N-way split: if a permission q (whether full or partial) is held, it can be divided among N threads—typically for a parallel loop—where conceptually each thread receives a fraction q/N.

Conversely, two fractional permissions can be combined, provided they refer to the same value and their sum does not exceed 1:

$$x.f \overset{q_1}{\mapsto} v * x.f \overset{q_2}{\mapsto} v \iff x.f \overset{q_1+q_2}{\mapsto} v$$

Uninitialized Cells

In separation logic, a distinction is made between the state of a memory cell being initialized or uninitialized. This distinction is crucial as it dictates the operations allowed on the resource.

A resource describing an uninitialized cell, denoted as $\mathsf{UninitCell}$, grants the permission to write to that memory location, as writing overwrites the current content regardless of its state. However, it prohibits reading, as

accessing uninitialized memory yields an undefined value. Writing to an uninitialized cell typically transitions its state to initialized (Cell), after which read operations become permissible.

Entailment

With the resources and principles defined above, the objective is now to verify whether a given set of available resources, Γ, is sufficient to satisfy the requirements specified by a target resource set, Γ'. When this relationship holds, we state that Γ *entails* Γ', denoted as $\Gamma \Rrightarrow \Gamma'$. If the entailment holds in both directions, the resource sets are considered equivalent, denoted as $\Gamma = \Gamma'$.

The following entailments illustrate permission splitting and state weakening:

$$x \mapsto \text{Cell} \;\Rrightarrow\; \alpha \cdot (x \mapsto \text{Cell}) \;\star\; (1-\alpha) \cdot (x \mapsto \text{Cell})$$

$$y \mapsto \text{Cell} \;\Rrightarrow\; y \mapsto \text{UninitCell}$$

2.2 OptiTrust

2.2.1 OptiTrust Overview

Purpose

Charguéraud et al. [8] introduce a framework for code transformations that bridges the gap between high-level specifications and optimized low-level implementations. OptiTrust integrates a type system based on Separation Logic, which serves to both validate the initial program and verify the correctness of the transformations applied to it. This section primarily summarizes the contributions of [3]; for further details, we refer the reader to the original publication.

High-Level Workflow

At a high level, the user provides two inputs: a source program written in *OptiC*, a C style language with stricter semantics and reduced undefined behavior; a transformation script describing the desired optimization steps. The tool produces a transformed *OptiC* program reflecting the requested optimizations and an interactive trace that visually explains how each transformation was applied.

This execution trace enables users to comprehend the precise effect of each step involved in the transformation pipeline. An illustrative trace is presented

in Figure 2.1. In this specific example, two transformations (loop fissions) have been applied. While the figure displays the differential between the original source code and the final output, the framework also allows for the inspection of incremental differences corresponding to each individual transformation step.

Trace for loop_fission_doc
- Loop.fission ~nest_of:2 [nbExact 1; tAfter; cFor "i"; cFor "j"; cWriteVar "y"];
- Loop.fission ~nest_of:2 [cForBody "k"; tBetweenAll];)

```
@@ -1,16 +1,24 @@
```

```
1  int main() {
2   int x, y, z;
3   for (int i = 0; i < 10; i++) {
4    for (int j = 0; j < 10; j++) {
5     x = i;
6     y = i;
7     z = i;
8    }
9   }
10  for (int i = 0; i < 10; i++) {
11   for (int k = 0; k < 10; k++) {
12    x = k;
13    y = k;
14   }
15  }
16 }
```

```
1  int main() {
2   int x, y, z;
3   for (int i = 0; i < 10; i++) {
4    for (int j = 0; j < 10; j++) {
5     x = i;
6     y = i;
7    }
8   }
9   for (int i = 0; i < 10; i++) {
10   for (int j = 0; j < 10; j++) {
11    z = i;
12   }
13  }
14  for (int i = 0; i < 10; i++) {
15   for (int k = 0; k < 10; k++) {
16    x = k;
17   }
18  }
19  for (int i = 0; i < 10; i++) {
20   for (int k = 0; k < 10; k++) {
21    y = k;
22   }
23  }
24 }
```

Figure 2.1: Left: list of applied transformations. Right: a diff showing code code before and after

OptiC

OptiC is a restricted subset of C designed to reduce sources of undefined behavior while preserving the expressiveness needed for optimization tasks. Within the scope of this work, the features offered by *OptiC* were sufficient to easily adapt the LLM code we based our transformations on.
However, some limitations remain. For instance:

- structures containing arrays are not yet supported;
- recursive functions are excluded from the current framework.

Despite these restrictions, *OptiC* captures a large and expressive subset of practical C code suitable for transformation and verification.

Internal Representation

Internally, transformations operate on an intermediate representation called *Opti*λ, a core imperative λ-calculus. The full formal definition of the language

is provided in [4].

Regarding verification—referred to as *typing* in the context of separation logic—a key feature of *Opti*λ is that every term of the Abstract Syntax Tree (AST) can carry *annotations*. These provide auxiliary information without affecting the program's operational semantics.

The internal AST currently distinguishes between two categories of annotations. First, there are annotations essential for the typing process:

- Separation logic contracts for functions and loops;
- Markers identifying *ghost instructions*, which have no effect on the program execution but are necessary to adapt the set of resources.

Second, there are annotations designed for tooling and user interaction:

- Style annotations that guide the pretty-printing (reverse translation) from *Opti*λ back to *OptiC*;
- User-defined *marks*, which allow transformation scripts to reference specific subterms by name;
- Explicit typing information for bindings, operators, and subterms.

Interactive Transformations

```
Loop.fission ~nest_of:2 [cForBody "k"; tBetweenAll];
```

Figure 2.2: Transformation syntax illustrated by the Loop fission transformation

Once the program has been translated to *Opti*λ, and successfully typechecked, the user can start applying transformations on that AST by writing transformations in an OCaml script, as illustrated by the Fig 2.2. Each script specifies *what* transformations to apply and *where* they should be applied in the program.

The nature of the operation (the *what*) is determined by the specific transformation function (here, `Loop.fission`). The targeted location (the *where*) is defined by a constraint-based targeting system that precisely identifies the application sites. For instance, the constraint `cForBody("k")` targets the body of any loop whose iterator is a variable named *k*, while the selector `tBetweenAll` specifies that the operation targets the interstices between every instruction. Extra arguments can also be required, here we specified `nest_of:2` to specify the number of loops concerned by the fission, therefore, the fission will separate the loop iterating over k and the one above, as it can be seen in Fig 2.1 When executed, the transformation engine applies the rules

sequentially, updating the annotated AST and generating a transformation trace.

Levels of Trust

Currently, OptiTrust distinguishes three levels of trust in its verification process:

Level 1 — Unverified transformations. Transformations are applied under full user responsibility, without any formal semantic guarantees.

Level 2 — Shape-checked transformations. The programmer annotates the code with *shapes* which are lightweight structural and permission contracts. These annotations ensure essential semantic properties, such as:

- correctness of read/write permissions on arrays,
- proper allocation and deallocation of memory,
- initialization before read.

Level 3 — Functional specification. At this level, transformations are validated against user-provided functional specifications. On can check that the final transformed code satisfies the same specification as the input code.

2.2.2 Optitrust Typing System

Resource Grammar

Traditional type checkers use a typing judgment of the form:

$$\Gamma \vdash t : \tau$$

In contrast, OptiTrust extends this model to account for *linear resources*, following the principles of Separation Logic. Therefore the typing judgement can be seen as: $\{\Gamma\}\ t\ \{\Gamma'\}$ where the input context Γ decomposes into two components:

- E: *pure resources* — duplicable, persistent information e.g., type declarations, logical propositions, or ghost functions
- F: *linear resources* — describing owned memory fragments. A resource $y : H$ indicates that the heap predicate H describes ownership of part of memory. Linear resources cannot be duplicated or discarded; they may be split into fractional permissions or rejoined.

Pure resources generally represent *ghost entities*: auxiliary information required for type checking but erased in the final executable. Similarly, the

```
void f (float *x, float *y){
  __reads("x ~> Cell");
  __writes("y ~> UninitCell");
  *y = (*x);
    }
```

Figure 2.3: Illustration of function contracts in Optitrust

output context is partitioned into E' and F', representing the pure and linear resources in the postcondition. Resources defined in E are accessible in F, E', and F', whereas those in E' are restricted to usage within F'.

Contracts in OptiTrust

Each function or loop is annotated with a contract specifying pre- and postconditions in terms of resources. For both function and loop contracts, we will first introduce the formalism for these contracts, and then provide examples of their concrete representation in OptiTrust

Function Contracts A function annotated as $\texttt{fun}(a_1, \ldots, a_n)\, \gamma \mapsto t$ carries a *contract*

$$\gamma = \{\, \text{pre} = \Gamma_{\text{pre}};\ \text{post} = \Gamma_{\text{post}} \,\}$$

The precondition lists the required resources and parameters, while the postcondition specifies the resources and values returned. Formally, a contract defines a Separation Logic triple:

$$\{\Gamma_{\text{pre}}\}\ f(a_1, \ldots, a_n)\ \{\Gamma_{\text{post}}\}$$

We use syntactic sugar in OptiTrust to express these contracts, as illustrated in Figure 2.3. The annotation `__reads` specifies that a read permission is required as a precondition and that this permission is returned as a postcondition. The annotation `__writes` signifies that a full permission is necessary in the precondition (which may be in the Uninit state), and that a permission will be returned in the postcondition (which must be in the Cell state).

Primitive heap operations are associated with built-in contracts, each describing the resources they require (preconditions) and produce (postconditions):

- `MALLOC(N)`: Heap allocation does not require any precondition. It yields `N` full permissions for the allocated array. Specifically, if a denotes the result of the allocation, the postcondition is defined as $\bigstar_{i=1}^{N} a[i] \rightsquigarrow$ UninitCell.

- `get(a)`: In order to retrieve the value pointed to by a, one must hold a fractional (or full) permission on the pointer: $a \overset{f}{\rightsquigarrow} \mathsf{Cell}$. This exact permission is restored in the postcondition.
- `set(a)`: To set the value pointed to by a, one must own a full permission on $a \rightsquigarrow \mathsf{UninitCell}$. The operation ensures the cell is initialized in the postcondition.
- `free(a)`: The free operation acts as the inverse of heap allocation. It requires the full permission associated with the allocated resource and consumes it entirely without restoring it.

Ghost Functions Ghost functions manipulate resources abstractly for verification purposes. They manipulate resources abstractly but are not executed in the compiled code. For example, `swap_groups` swaps iterated conjunctions in heap predicates :

$$\{\bigstar_{i \in R_i} \bigstar_{i \in R_j} H(i,j)\}\text{swap_groups}\{\bigstar_{i \in R_j} \bigstar_{i \in R_i} H(i,j)\}$$

Ghosts can also come in pairs: `ghost begin` allows performing a certain operation on resources, and `ghost end` allows reversing this operation; the operation must of course be reversible to enable the insertion of a ghost pair. Further examples are provided in Chapter 3. A call to a ghost function, as well as the insertion of a ghost pair, will be referred to as *ghost operations* in this manuscript.

For-Loop Contracts A for-loop annotated with contract χ has the form:

$$\text{for } i \in R \ \chi\{t\}$$

where χ has the following components:

- χ`.vars`: ghost variables scoped over the loop,
- χ`.excl.pre`, χ`.excl.post`: resources consumed and produced per iteration,
- χ`.shrd.reads`: read-only shared resources,
- χ`.shrd.inv`: sequential invariant (empty implies parallelizable loop).

To write loop contracts in Optitrust, one may use syntactic sugar, as presented in Fig 2.4

```
for(int i = 0; i < n; i++){
  __xwrites(x[i] ~> Cell);
  __sreads("y ~> Cell");
}
```

Figure 2.4: Loop contracts in Optitrust

Entailment by Subtraction Operation and Carving

Subtraction provides an algorithmic method to check entailment and infer variable instantiations. Given Γ and Γ', it computes the frame F such that $\Gamma \implies \Gamma' \star F$, where F represents the resources from Γ not consumed by Γ' that are preserved.

The subtraction operation also infers the instantiation map σ, which provides the witnesses for the instantiations of the variables that are bound (and therefore existentially quantified) in Γ'. This map acts as the substitution for the existential variables in Γ'.

Two variants of subtraction are used in OptiTrust:

- **Core subtraction** $\Gamma \boxminus \Gamma'$: may weaken initialized resources into uninitialized ones, but cannot split read-only permissions.
- **Carving subtraction** $\Gamma \boxdot \Gamma'$: extends core subtraction by allowing fractional carving, i.e., it can extract a portion of a read-only permission when needed.

If the subtraction succeeds with an empty frame,

$$\Gamma \boxminus \Gamma' = (\sigma, \emptyset), \quad \text{then} \quad \Gamma \Rightarrow \Gamma'.$$

Algorithmic Scheme The subtraction operation is implemented following a standard matching process:

1. Initialize the substitution map σ with bindings that associate each pure variable of Γ' to a fresh unification variable.
2. For each linear resource in Γ', syntactically match it against a corresponding resource from Γ. This process may trigger unifications, resulting in the partial or total resolution of certain unification variables.
3. If Γ' requests a resource of the form $\text{Uninit}(H)$ while Γ contains H, the algorithm performs an on-the-fly weakening:

$$H \Rightarrow \text{Uninit}(H).$$

4. The remaining items in Γ after matching are assigned to the frame F.

Carving Refinement When Γ' requests a fractional resource αH and Γ contains $\beta H'$, the algorithm performs a fractional split:

$$\beta H' \Rightarrow \alpha' H' * (\beta - \alpha')H', \quad \text{with a fresh fraction } \alpha'.$$

The binding $\alpha := \alpha'$ is added to σ, and the leftover fraction $(\beta - \alpha')H'$ remains in Γ for future matches.

To illustrate the concept of *carving*—and the subtraction algorithm more generally—we detail the following example:

```
void some_computation(float* r1, float* r2, float* w1, float* w2)
{
  __reads("r1 ~> Cell");
  __reads("r2 ~> Cell");
  __writes("w1 ~> UninitCell");
  __writes("w2 ~> UninitCell");
  *w1 = *r1 + *r2;
  *w2 = *r1 - *r2;
}

int main(float* p1, float* q1){
  // syntactic sugar for full permission on initialized cell
  __modifies("p1 ~> Cell");
  __writes("q1 ~> UninitCell");

  float* q2=(float*) malloc(sizeof(float));
  q2 = 2.f; // we own a full init permission on q1
  some_computation(p1, p1, q1, q2);
}
```

Figure 2.5: Example of resource subtraction with aliasing and weakening.

The `main` function possesses the following input permissions: a full initialized permission on `p1` and a full uninitialized permission on `q1`. The assignment `*q2 = 2.f;` establishes a full initialized permission on `q2`.

The call to `some_computation` requires read-only permissions for `r1` and `r2`, as well as full uninitialized permissions for `w1` and `w2`.

To type-check the call `some_computation(p1,p1,q1,q2)`, the system attempts to determine if it holds the necessary resources to satisfy the function's precondition. This is where the subtraction algorithm is invoked.

At the call site, the context Γ is as follows:

$$\Gamma = \{r1_main \rightsquigarrow \mathsf{Cell} \star w1_main \rightsquigarrow \mathsf{UninitCell} \star w2_main \rightsquigarrow \mathsf{Cell}\}$$

We seek to establish the entailment:

$$\Gamma \overset{?}{\Rightarrow} \{\alpha_1(r1_main \rightsquigarrow \mathsf{Cell}) \star \alpha_2(r1_main \rightsquigarrow \mathsf{Cell}) \\ \star w1_main \rightsquigarrow \mathsf{UninitCell} \star w2_main \rightsquigarrow \mathsf{UninitCell}\} \star F$$

where F represents the leftover frame.

The type checking proceeds resource by resource. First, we attempt to resolve the first read requirement:

$$\Gamma \overset{?}{\Rightarrow} \{\alpha_1(r1_main \rightsquigarrow \mathsf{Cell})\} \star F_1$$

Here, using *carving subtraction*, since the context holds the resource $r1_main \rightsquigarrow \mathsf{Cell}$, we can establish:

$$\Gamma \Rightarrow \{\alpha_1(r1_main \rightsquigarrow \mathsf{Cell})\} \star \Gamma_1$$

where $\Gamma_1 = \{(1-\alpha_1)(r1_main \rightsquigarrow \mathsf{Cell}) \star w1_main \rightsquigarrow \mathsf{UninitCell} \star w2_main \rightsquigarrow \mathsf{Cell}\}$

The subtraction process continues, using Γ_1 as the available resource set for the remaining requirements. Similarly, for the resource $\alpha_2(r1_main \rightsquigarrow \mathsf{Cell})$, the algorithm performs another carving subtraction step to derive:

$$\Gamma_1 \Rightarrow \{\alpha_2(r1_main \rightsquigarrow \mathsf{Cell})\} \star \Gamma_2$$

where

$$\Gamma_2 = \{(1-\alpha_1-\alpha_2)(r1_main \rightsquigarrow \mathsf{Cell}) \star w1_main \rightsquigarrow \mathsf{UninitCell} \star w2_main \rightsquigarrow \mathsf{Cell}\}$$

For `w1_main`, there is a direct syntactic match between the requested resource and the resource available in Γ_2. The entailment is therefore immediate:

$$\Gamma_2 \Rightarrow \{w1_main \rightsquigarrow \mathsf{UninitCell}\} \star \Gamma_3$$

$$\text{where } \Gamma_3 = \{(1-\alpha_1-\alpha_2)(r1_main \rightsquigarrow \mathsf{Cell}) \star w2_main \rightsquigarrow \mathsf{Cell}\}$$

Finally, for the resource $w2_main \rightsquigarrow \mathsf{UninitCell}$, the algorithm performs the on-the-fly weakening. This allows satisfying an uninitialized requirement using an initialized resource (effectively "forgetting" the initialization status):

$$\Gamma_3 \Rightarrow \{w2_main \rightsquigarrow \mathsf{UninitCell}\} \star \{(1-\alpha_1-\alpha_2)(r1_main \rightsquigarrow \mathsf{Cell})\}$$

Thus, the type checking succeeds, and we can establish the following final relation:

$$\begin{aligned}\Gamma \Rightarrow \{&\alpha_1(r1_main \rightsquigarrow \mathsf{Cell}) \star \alpha_2(r1_main \rightsquigarrow \mathsf{Cell}) \\ &\star w1_main \rightsquigarrow \mathsf{UninitCell} \star w2_main \rightsquigarrow \mathsf{UninitCell}\} \\ &\star \{(1-\alpha_1-\alpha_2)(r1_main \rightsquigarrow \mathsf{Cell})\}\end{aligned}$$

Chapter 3

Autofocus

3.1 Motivation

As discussed in the previous section, the validation of OptiTrust scripts requires explicit resource annotations. While such annotations ensure precise description of the permissions, their writing by a programmer can quickly become verbose and time-consuming.

Furthermore, many common programming patterns exhibit distinct and predictable *focus operations* regarding memory access. For instance, we observed that programs frequently require a temporary focus on a single element within a larger array, or on a sub-block of a matrix. Consequently, we investigate whether these *ghost operations* can be inferred automatically as part of an elaboration process, while maintaining the same level of formal safety as if all ghosts had been explicitly written by the programmer.

We call this elaboration feature *autofocus*. Concretely, from a resource represented by a group of iterated separating conjunction, it extracts the relevant subset of resources needed to type-check a fuction call.

3.1.1 Description of a Focus

As shown in Figure 3.1, the function `array_sum` computes and returns the sum of the floating point array a_{out}. In order to be able to read the array values, the function contract describes a permission over the entire array a_{out} written as : `for j in 0..n ->` a_{out}`[j]` $\rightsquigarrow$ `Cell`. However, the instruction inside the loop, a_{out}`[i]++`, requires write access to the specific cell a_{out}`[i]`.

To ensure that this sequence type-checks under the current permission system, the developer must add ghost annotations as shown in Figure 3.2. This addition consists of the insertion of a *ghost pair*, materialized in the code by a call to the `__ghost_begin` function, which invokes the desired *ghost function*, and by a corresponding `ghost_end` directive that reverts the resource changes

```
float array_sum(float* a_out, int n) {
  __reads("for j in 0..n -> &a_out[j] ~> Cell");
  float sum = 0.f;
  for (int i = 0; i < n; i++) {
     // Error: Resource a_out[i] ~> Cell not found
     // In Context: for j in 0..n -> a_out[j] ~> Cell
     sum +=a_out[i];
  }
  return sum;
}
```

Figure 3.1: Example of a loop where the permission held (Group) does not match the permission required (Cell) by the instruction.

```
float array_sum(float* a_out, int n) {
  __reads("for j in 0..n -> a_outx[j] ~> Cell");
  float sum = 0.f;
  for (int i = 0; i < n; i++) {
    const __ghost_fn focus_a_out_i =
          __ghost_begin(focus_matrix1, "matrix:=a_out, index:=i");
    sum +=a_out[i];
    __ghost_end(focus_a_out_i);
  }
  return sum
}
```

Figure 3.2: Example of the ghost instructions needed to focus on the specific cell `x[i]` inside the loop.

induced by the `ghost_begin`. As stated in 2.2, a *ghost function* is a function provided by the OptiTrust library that only impacts the description/view of resources over the scope of the ghost pair. The purpose of the ghost function is to rearrange the current resource state for an equivalent one, in order to match exactly the format of the precondition. In the example from 3.2 the ghost function, named `focus_matrix1`, can be formally described as the equivalence:

$$\bigstar_{i=1}^{n} a_{out}[i] \rightsquigarrow \mathsf{Cell} \iff \Big(x[i] \rightsquigarrow v\Big) \star \Big((x[i] \rightsquigarrow v) \mathrel{-\!\!\star} \Big(\bigstar_{i=1}^{n} a_{out}[i] \rightsquigarrow \mathsf{Cell}\Big)\Big).$$

The Magic Wand (recall 2.1), or Wand for short, captures the idea that the full permission over the entire array (described as the group on the left-hand side of the equivalence) is suspended until the specific element permission is returned.

In summary, as this example shows, even for a simple loop update, two additional ghost operations must be written: one to focus on the element $a_{out}[i]$, and another to restore the permission on the entire array.

3.1.2 Focusing on Array Slices

While the previous example may seem artificial, our motivation is grounded in realistic use cases. In particular, many deep-learning kernels, such as those used in LLM inference, operate on array *slices*: subsets of data defined along specific dimensions. For a 2D matrix, a slice corresponds to a row; for a 3D matrix, a slice can be either a 2D matrix or a row. Let us now imagine a context where we hold a permission on a 2D matrix, and we call a function requiring an array corresponding to a row of this matrix as an argument. There is a clear mismatch between the held permission and the one expected by the function. The situation becomes even more problematic in the LLM case study where a 4D array and the called function expects a 2D matrix; multiple focus operations are needed to exhibit the desired permission.

Figure 3.3 illustrates a typical scenario during the *embedding* computation. While the precise semantics of this computation are detailed in Chapter 5, they are not critical for the current discussion. The key observation is that even a simple matrix-vector multiplication necessitates several non-trivial focus operations. Here, we aim to multiply a vector (represented by the i-th row of the 2D matrix mha_q) by a matrix (represented by a slice of the 4D matrix mha_q_weight). Consequently, the programmer must explicitly write two verbose ghost pairs ($focus_q$ and $focus_norm$) to ensure the code type-checks, effectively tripling the size of this code.

```
for (int i = 0; i < seq_len; i++) {
  __sreads("mha_q_weight ~> Matrix4(l_count, qh_count, h_dim, e_dim)");
  __xmodifies("
  for i1 in 0..h_dim -> &mha_q[MINDEX3(qh_count, seq_len, h_dim, q, i, i1)]
  ~> UninitCell
  ");
  __xreads("&mha_norm[MINDEX2(seq_len, e_dim, i, 0)] ~> Matrix1(e_dim)");
  const __ghost_fn focus_q = __ghost_begin(ro_group_focus,
    "i := l, items := fun (l: int) -> for q in 0..qh_count ->
    for h in 0..h_dim -> for e in 0..e_dim ->
    &mha_q_weight[MINDEX4(l_count, qh_count, h_dim, e_dim, l, q, h, e)] ~> Cell");

  const __ghost_fn focus_norm = __ghost_begin(ro_group_focus,
    "i := q, items := fun (q: int) -> for h in 0..h_dim-> for e in 0..e_dim->
    &mha_q_weight[MINDEX4(l_count, qh_count, h_dim, e_dim, l, q, h, e)] ~> Cell");

  matvec(h_dim, e_dim,
         &mha_q[MINDEX3(qh_count, seq_len, h_dim, q, i, 0)],
         &mha_norm[MINDEX2(seq_len, e_dim, i, 0)],
         &mha_q_weight[MINDEX4(l_count, qh_count, h_dim,
                               e_dim, l, q, 0, 0)]);
  __ghost_end(focus_q);
  __ghost_end(focus_norm);
}
```

Figure 3.3: LLM code and annotations required to successfully type-check this sequence

The LLM code shown above would largely benefit from autofocus, as it would avoid the manual writing of the clumsy ghost operations. But, beyond LLM code, a wide range of High-Performance Computing code, block-based numerical algorithms or tiled linear algebra kernels, would also benefit from autofocus. In particular, any code exhibiting one or both of the following two properties would strongly benefit from the autofocus mechanism

1. **Consecutive storage, sequential computation:** Weight matrices are stored contiguously, but the computation proceeds sequentially through layers, meaning that submatrices are constantly being manipulated.
2. **Kernel-based computation:** Embedding and transformation operations are performed through repeated kernel calls, each operating on specific matrices slice.

Such recurring patterns strongly motivated us to introduce an automatic inference of focus permissions, reducing the cost of writing and maintaining OptiTrust input code.

3.2 Handled Cases and Limitations

3.2.1 Description of Covered Cases

The *autofocus* mechanism should apply to both reading and writing operations.

Formally, we want the typechecker to accept the following pattern: If a permission over a set of groups referring to disjoint cells, that we will refer to as *resource candidate*, is part of the context and if a function call requires a permission on a subset of these groups, referred to as *resource target*, possibly with some groups replaced, then the system should automatically insert the appropriate ghost operations and typecheck successfully.

Example

$$\textbf{Resource candidate:} \bigstar_{i=1}^{n} a[i] \rightsquigarrow \mathsf{Cell} \quad \textbf{Resource target:} \; a[0] \rightsquigarrow \mathsf{Cell}$$

Here, the typechecker deduces that the resource on the array a must be focus to the specific cell $a[0]$.

3.2.2 Restrictions

Solving autofocus in full generality can be arbitrarily difficult. Hence, some restrictions are necessary to leverage the existing OptiTrust infrastructure without having to fully adapt the tool. In particular:

- If a group appears both in the resource before focusing and in the focused resource, it must correspond to the same value range. If the range were to change, we would need to prove that the operation only affects a sub-range, which involves arithmetic comparisons between the values of arbitrary terms. This capability is not currently supported by OptiTrust, although it shouldn't be that hard to add the inclusion $m \leq n$ The following candidate and target will not generate autofocus ghosts:

  Resource candidate: $\bigstar_{i=0..n} a[i] \rightsquigarrow \mathsf{Cell}$ **Resource target:** $\bigstar_{i=0..m} a[i] \rightsquigarrow \mathsf{Cell}$

- The candidate resource must operate on initialized `Cells`. Else, it would mean that the part of the array that has been focused is initialized while the other part remains uninitialized. The current type system does not support heterogeneous initialization states within a single array. The following candidate and target will not generate autofocus ghosts:

  Resource candidate: $\bigstar_{i=0..n} a[i] \rightsquigarrow \mathsf{UninitCell}$ **Resource target:** $a[0] \rightsquigarrow \mathsf{UninitCell}$

3.3 Autofocus Algorithm

In this section, we formalize the Autofocus problem and define the underlying algorithm. The core objective is to determine if the possession the *candidate resource*, entails (or can satisfy) the *target resource*.

3.3.1 Problem Formulation

Let the *candidate resource* be defined as:

$$R_{\text{cand}} = \bigstar_{i_1=0}^{e_1} \dots \bigstar_{i_p=0}^{e_p} a[\text{MINDEX}_n(d_1, \dots, d_n, x_1, \dots, x_n)] \rightsquigarrow \mathsf{Cell}$$

and the *target resource* as:

$$R_{\text{target}} = \bigstar_{j_1=0}^{e'_1} \dots \bigstar_{j_k=0}^{e'_k} a[\text{MINDEX}_n(d'_1, \dots, d'_n, y_1, \dots, y_n)] \rightsquigarrow \mathsf{Cell}$$

where the x_i's and y_i's are arbitrary expressions. We introduce the following notation regarding candidate and target resources. Note that for target resources, the operators are applied to the y_i variables:

- $\text{Vars}(x_j)$ denotes the set of iterator variables present in x_j. For instance, given the resource $\bigstar_{i_1}^{n_1} \bigstar_{i_2}^{n_2} a[n_1, n_2, i_1, i_1 + i_2]$, we have $\text{Vars}(x_1) = \{i_1\}$ and $\text{Vars}(x_2) = \{i_1, i_2\}$.

- $\text{New}(x_j)$ denotes the set containing the iterator variables that appear in x_j but not in the the prior x_k's. Formally:

$$\text{New}(x_j) = \text{Vars}(x_j) \setminus \bigcup_{k<j} \text{Vars}(x_k)$$

We impose the following constraints and properties on these resources:

- Since our goal is to instantiate (and thus restrict) the candidate's iterators to match the target, we expect the candidate resource to have at least the same number of iterated stars as the target resource. Formally, we assume $k \leq p$.
- Both the candidate and the target resources should share the same dimensions, ensuring identical addressing computations:

$$\forall j \in \{1, \ldots, n\}, \quad d_j = d'_j$$

- Each iterative variable appears at least once in the indexing:

$$\forall j \in \{1, \ldots, p\}, \exists k \in \{1, \ldots, n\} \text{ such that } i_j \in \text{Vars}(x_k)$$

  For instance, the resource $\bigstar_{i_1}^{n_1} \bigstar_{i_2}^{n_2} a[\text{MINDEX}_2(n_1, n_2, i_1, i_1)] \rightsquigarrow \mathsf{Cell}$ does not fall within the scope of the autofocus feature. This is a necessary step to be able to find a canonical order for the iterated stars (more details on this are provided in 3.3.2)

- There is at most one *new* iterative variable per x_j. Formally:

$$\forall j \in \{1, \ldots, n\}, \quad |\text{New}(x_j)| \leq 1$$

  For example, the resource:

$$\bigstar_{i_1}^{n_1} \bigstar_{i_2}^{n_2} a[\text{MINDEX}_2(n_1, n_2, i_1, i_1 + i_2)]$$

  can be autofocused, whereas

$$\bigstar_{i_1}^{n_1} \bigstar_{i_2}^{n_2} a[\text{MINDEX}_2(n_1, n_2, i_1 + i_2, i_2)]$$

  is outside the scope of autofocus.

3.3.2 Phase 1: Reordering

The algorithm is able to reorder the iterated stars, meaning that the iterated stars may appear in a different order in the candidate resource than in the target resource. Consequently, we must first determine a method to align them. For instance, consider:

$$R_{\text{cand}} = \mathop{\bigstar}_{i_1}^{n_1} \mathop{\bigstar}_{i_2}^{n_2} a[\text{MINDEX}_2(n_1, n_2, i_1, i_2)]$$

and

$$R_{\text{target}} = \mathop{\bigstar}_{i_1}^{n_2} \mathop{\bigstar}_{i_2}^{n_1} a[\text{MINDEX}_2(n_1, n_2, i_2, i_1)].$$

We must establish that the variable i_2 corresponds to the second iterator in the candidate resource (acting on the second index), whereas it matches the first iterator in the target resource. To address this, we introduce a normalization operation called Reorder, which is applied independently to both the candidate and target resources. This operation uniquely modifies the order of the iterated stars without altering the predicate on the memory cell. Informally, the operation relies on a permutation π to reorder the iterated stars based on the order of appearance of the iterator variables within the index expressions.

Example For the resource:

$$R = \mathop{\bigstar}_{i_1}^{e_1} \mathop{\bigstar}_{i_2}^{e_2} \mathop{\bigstar}_{i_3}^{e_3} a[\text{MINDEX}_3(d_1, d_2, d_3, i_3, i_1, i_2)] \leadsto \mathsf{Cell}$$

we have $\pi(1) = 3$, $\pi(2) = 1$, $\pi(3) = 2$. Therefore:

$$\text{Reorder}(R) = \mathop{\bigstar}_{i_3}^{e_3} \mathop{\bigstar}_{i_1}^{e_1} \mathop{\bigstar}_{i_2}^{e_2} a[\text{MINDEX}_3(d_1, d_2, d_3, i_3, i_1, i_2)] \leadsto \mathsf{Cell}$$

Formalisation Retaining the previous notation, consider a resource R of the form:

$$R = \mathop{\bigstar}_{i_1}^{e_1} \dots \mathop{\bigstar}_{i_p}^{e_p} a[\text{MINDEX}_n(d_1, \dots, d_n, x_1, \dots, x_n)] \leadsto \mathsf{Cell}$$

and assume that it satisfies all the hypotheses defined in the previous subsection. We will define the permutation on an integer sequence $a_1, \dots, a_p$. This sequence is defined recursively by $a_0 = 0$, and

$$\forall j \text{ such that } a_{k-1} < j < a_k, \quad |\text{New}(x_j)| = 0 \quad \wedge \quad |\text{New}(x_{a_k})| = 1$$

The sequence $a_0, \dots, a_p$ is well-defined and strictly increasing because every iterative variable appears in the indexing (Hyp 3.1) and there is at most one

new iterative variable per x_i (Hyp 3.2). We can now define the permutation as follows:

$$\forall j \in \{1, \ldots, p\}, \quad \pi(j) = \alpha_j \quad \text{such that} \quad \mathrm{New}(x_{a_j}) = \{i_{\alpha_j}\}$$

By definition of a_j, which forms a strictly increasing sequence of p elements, π_{cand} properly defines a permutation on $\{1, \ldots, p\}$. We proceed similarly for π_{target}.

We can then reorder the stars by defining the operator Reorder

$$\mathrm{Reorder}(R) = \mathop{\bigstar}_{i_{\pi(1)}}^{e_{\pi(1)}} \ldots \mathop{\bigstar}_{i_{\pi(l)}}^{e_{\pi(l)}} a[\mathrm{MINDEX}_n(d_1, \ldots, d_n, x_1, \ldots, x_n)] \rightsquigarrow \mathsf{Cell}$$

This canonical form allows us to process dimensions in a deterministic order. From this point forward, we assume that both the candidate and the target resources are normalized:

$$R_{\text{cand}} \leftarrow \mathrm{Reorder}(R_{\text{cand}}) \quad \text{and} \quad R_{\text{target}} \leftarrow \mathrm{Reorder}(R_{\text{target}})$$

3.3.3 Phase 2: Index Instantiation

The second phase defines an instantiation mapping, denoted by σ, which determines how the index variables of the candidate resource must be instantiated to match the target. Formally, σ associates each iterator variable i_k with an expression t_k:

$$\sigma = \{i_1 \mapsto t_1, \ldots, i_l \mapsto t_l\}$$

Applying this instantiation to the candidate resource R_{cand} yields a new resource denoted as $\sigma(R_{\text{cand}})$. The behavior of the mapping for a given variable i_k is twofold:

- If $t_k = i_k$ (identity mapping), the index variable remains a bound loop variable, and the corresponding iterative star is preserved in the resource description.
- If t_k is an expression distinct from i_k (instantiation), the star is removed, and all occurrences of i_k in the resource are replaced by t_k.

The goal is to find a σ such that the instantiated candidate resource is equivalent to the target resource R_{target}, modulo α-renaming of the bound iteration variables (the ones preserving their stars). This condition is expressed as:

$$\sigma(R_{\text{cand}}) \equiv_\alpha R_{\text{target}}$$

In the following section, we define how to compute such a σ.

Algorithm for computing the mapping of iterator variables

Before formally defining the instantiation algorithm, we present two examples to illustrate its role and the complexity of the task.

Direct Mapping First, let us consider a simple yet general case where the iterator variables in the candidate resource correspond directly to the array access indices:

$$R_{\text{cand}} = \bigstar_{i_1=0}^{e_1} \bigstar_{i_2=0}^{e_2} \bigstar_{i_3=0}^{e_3} a[\texttt{MINDEX3}(n_1, n_2, n_3, i_1, i_2, i_3)]$$
$$R_{\text{target}} = \bigstar_{i_2=0}^{e_2} a[\texttt{MINDEX3}(n_1, n_2, n_3, 2, i_2, 5)]$$

Here, by comparing the arguments of the MINDEX3 access, we can immediately deduce the instantiation mapping $\sigma : \{i_1 \mapsto 2, i_2 \mapsto i_2, i_3 \mapsto 5\}$

Inter-variable Dependencies More complex scenarios arise where the instantiation of one variable depends on another. This highlights the subtleties that our algorithm must handle. Consider the following resources:

$$R_{\text{cand}} = \bigstar_{i_1=0}^{e_1} \bigstar_{i_2=0}^{e_2} \bigstar_{i_3=0}^{e_3} a[\texttt{MINDEX3}(n_1, n_2, n_3, i_1, i_2, i_3 - i_2)]$$
$$R_{\text{target}} = \bigstar_{i_1=0}^{e_1} a[\texttt{MINDEX3}(n_1, n_2, n_3, i_1, i_1 - a, b - (i_1 - a))]$$

In this scenario, the instantiation σ is non-trivial but we can establish $\sigma : \{i_1 \mapsto i_1,\ i_2 \mapsto i1 - a,\ i_3 \mapsto b\}$ Crucially, the algorithm must propagate instantiations. By substituting the known value of $\sigma(i_2)$, the matching equation for the third argument becomes: $\sigma(i_3) - (i_1 - a) = b - (i_1 - a)$, which simplifies to $\sigma(i_3) = b$.

Formalisation To compute the mapping, we maintain a map H initialized as the identity function over the iterator variables of the candidate resource. That is, initially:

$$\forall a \in \{0..l\}, \quad H[i_a] = i_a$$

We iterate through the spatial dimensions $a \in \{1..n\}$, analyzing the index expressions x_a (from the candidate) and y_a (from the target).

Our algorithm proceeds by analyzing the values of $New(x_a)$ and $New(y_a)$:

Case 1: Missing iteration resource ($|New(y_a)| = 1 \wedge |New(x_a)| = 0$). Here, the target resource requires iteration over the dimension (a star exists), by definition of $New(y_a)$ but the candidate resource provides a fixed index (a scalar). Since a scalar resource cannot satisfy a loop requirement, the entailment fails.

Case 2: Both resources iterate ($|New(y_a)| = 1 \wedge |New(x_a)| = 1$). Both resources iterate over this dimension. We check that the index expressions are equivalent modulo renaming ($x_a \equiv_\alpha y_a$).

Case 3: Both resources are fixed for this index ($|New(y_a)| = 0 \wedge |New(x_a)| = 0$). Both resources do not iterate over this dimension. We check that the index expressions are equivalent modulo renaming ($x_a \equiv_\alpha y_a$).

Case 4: Instantiation ($|V_{y_a}| = 0 \wedge |V_{x_a}| = 1$). The candidate has a single iterating variable i_α, but the target requires a specific instance (a fixed value). We need to "consume" the star by instantiating the variable.

We attempt to solve for a value β such that substituting i_α with β in x_a yields y_a. If a valid β exists, we proceed with the following updates:

1. **Record Mapping:** Update the map $H[i_\alpha] = \beta$.
2. **Propagate Substitution:** Since i_α is now fixed, we substitute i_α with β in all subsequent candidate index expressions ($x_k \leftarrow x_k[i_\alpha : \beta]$ for all $k > a$). Thanks to the Reordering phase, we can assume that i_α does not appear for any $k < a$.

If no such β can be determined, the algorithm fails.

Upon successful completion of all dimensions, the map H represents the necessary instantiation list.

3.4 Integration in OptiTrust

In this section, we describe the implementation of the previously defined algorithm within the OptiTrust framework. We delve into the concrete technical details and explain the specific implementation choices.

3.4.1 Typing Pass

During the typing process with *autofocus* elaboration, the handling of a function call follows a two-fold strategy. First, we attempt to apply the standard type-checking rules without autofocus; if this syntactic matching succeeds, the process concludes. Otherwise, if the standard check fails, we proceed to the second phase: for every resource required by the function call but missing from the current context, we attempt to automatically introduce a focus operation to provide the requested resource.

3.4.2 Substitution

The context may contain unification variables, known as evars in Rocq's terminology; some of these evars may be already resolved to concrete terms. We begin by substituting these resolved evars into the resources to simplify resource comparisons. In particular, the arguments in the function call are substituted.

3.4.3 Range Extraction

To manage resources over a set of disjoint cells such as:

$$\bigstar_{i_1=0}^{d_1} \dots \bigstar_{i_n=0}^{d_n} a[\text{MINDEX}_n(d_1, \dots, d_n, x_1, \dots, x_n)] \rightsquigarrow \mathsf{Cell}$$

The OptiTrust syntax defines "groups". A group consists of an iterator, a range, and a formula over which it iterates. Consequently, the formula above would be represented as:

```
group i1 (range 0 d1) (group i2 (range 0 d2) (... cell (array_access a [d_1;...;d_n] [x1;...;xn])))
```

This recursive structure is ill-suited for the autofocus mechanism because it obscures the relationship between the accessed indices for each dimension $x_1, \dots, x_n$ and the iteration variables $i_1, \dots, i_n$.

Therefore, in this preliminary step, we extract the following:

- **The list of groups:** specifically, the iterator i_j and the range $0..d_j$.
- **The term referring to the array element:** the component $a[\dots]$ for which we hold permission. This is required to reconstruct the syntax expected by OptiTrust.

Example.

– **Resource:**

$$\bigstar_{i_1=0}^{d_1} \dots \bigstar_{i_2=0}^{d_2} a[\text{MINDEX}_2(d_1, d_2, i_1/2, i_2/2)] \rightsquigarrow \mathsf{Cell}$$

– **Representation in OptiTrust:**

```
let res = group i1 (range 0 d1) (group i2 (range 0 d2) (... cell (array_access a [d_1;d_2] [i_1/2; i_2/2])))
```

– **Intermediate representation used in the Autofocus implementation:**

```
let ranges = ([(i1, range 0 d1); (i2, range 0 d2)], cell (array_access a [d_1;d_2] [i_1/2; i_2 /2]))
```

3.4.4 Term Unification

The unification process is a standard mechanism in typecheckers. It is used to determine if two terms can be unified: that is, if there exists a valid assignment of their evars that renders them identical. For instance, unifying the terms `foo(?x, 3, ?z, ?u)` and `foo(4, ?y, ?z, ?v)` results in the instantiation of the evars $?x := 4$, $?y := 3$, and $?v :=?u$. This process yields the resulting term `foo(4, 3, ?z, ?u)`, while the variable $?z$ remains uninstantiated.

In the autofocus context, before proceeding, we must ensure compatibility between base terms. We cannot rely on complete syntactic equality, as the purpose of *autofocus* is to allow non-syntactic matches. Therefore we must perform the following check

- The base terms (arrays) must unify in the current context.
- The relevant dimensions must unify.

3.4.5 Conversion to an Internal Representation

This step constitutes the concrete implementation of the `Reorder` function defined in the formalization. Its purpose is to transform the resource into a reordered form.

To achieve this, we reorder the groups based on the sequence defined by the arguments of the `MINDEX` call. We first prepare a list of `var_iterators`, which contains all variables acting as iterators over the groups. For each index term in the access function, we identify the variables present and compute their intersection with `var_iterators`. Three cases arise:

- The intersection is empty: the index term does not correspond to a group (it is a fixed index).
- The intersection contains exactly one variable: the index term corresponds exactly to a specific group iteration.
- The intersection contains multiple variables: multiple iteration variables act on the same index. In this case, the autofocus algorithm aborts, as no total ordering of groups can be safely determined.

Additionally, we verify the consistency between group ranges and dimensions. As stated in the formal requirements, each group must iterate over the entire allocated dimension. If this condition is not met, the system would require complex arithmetic proofs to ensure that the target resource's ranges are strictly included in the candidate's, which is outside the scope of this implementation.

Example:

– Resource:

$$\mathop{\bigstar}_{i_2=0}^{d_2} \ldots \mathop{\bigstar}_{i_1=0}^{d_1} a[\text{MINDEX}_2(d_1, d_2, i_1/2, i_2/2)] \rightsquigarrow \mathsf{Cell}$$

– Representation for Autofocus (range extraction):

```
let ranges = ([(i2, range 0 d2); (i1, range 0 d1)], cell (array_access a [d1;d2] [i1 /2; i2 /2]))
```

– Representation for Autofocus(group repr):

```
let group_repr = ([Star(i1, range 0 d1, i1 /2),  Star(i2, range 0 d2, i2/2)])
```

3.4.6 Focus List Construction

Having finalized the preparatory steps, we now execute the algorithm's core logic. The purpose here is to generate a focus list containing elements that define the iterator instantiation, along with the corresponding pre-resource and post-resource for this step. Groups are compared one by one according to their aligned order.

As presented in the algorithm section, we compare groups one by one and distinguish several scenarios:

- Both the target and candidate resources contain a star at the same index: the terms must unify.
- Two indices are present: similarly, the groups must unify.
- A star appears in the candidate resource and an index in the target resource: this is where *autofocus* applies. We must verify the previously defined entailment rule: there must exist an instantiation of the candidate's group variable that allows recovering the index in the target resource. We add to the focus list the instantiation and the resource before instantiation and after instantiation.
- A star appears in the target resource and an index in the candidate resource: autofocus is not possible.

3.4.7 Sequence Retyping

If the typing of the function call succeeds, we create a sequence of terms surrounding the call along with the ghosts described above.

Two cases arise: if the sequence corresponds to an instruction returning a value, we produce the sequence shown on the right of Figure 3.4; otherwise, if the function returns nothing, we produce the sequence on the left.

```
{
const __ghost_fn __ghost_pair =
    ghost_begin(...)
instr;
ghost_end(ghost_pair)
}
```

```
{
const __ghost_fn __ghost_pair =
    ghost_begin(...)
fun_type a = instr;
ghost_end()
a;
}
```

Figure 3.4: Instrumentation of function calls with ghosts. The left side shows the sequence for a void instruction, while the right side handles an instruction returning a value.

We then retype the entire sequence without autofocus to verify that the focus has generated a valid sequence. This ensures that correctness and soundness are guaranteed based on the entailment and subtract rules as they were defined without autofocus.

Additionally, we store inside the AST, for every node where an autofocus was involved, the list of ghost operations that were applied. This information is stored in a field named elaborate, and will be exploited at the next phase for materializing the focus operation as proper (ghost) operations to appear in the form statements in sequence nodes in the AST.

3.4.8 Elaboration

Finally, the autofocus process concludes with another traversal of the AST. Whenever a node is marked for elaboration, we insert the corresponding sequence of instructions.

This elaboration phase is accompanied by a *sequence flattening* phase. Indeed, the typed instruction might not be a statement (e.g., it is part of an expression), which can lead to an ill-positioned sequence and syntactically invalid C code, as shown in Figure 3.5.

```
// Invalid C: Block within expression
float x = f{
  const __ghost_fn __ghost_pair =
      ghost_begin(...)
  float a = compute_value(...);
  ghost_end(ghost_pair)
  a
};
```

```
// Valid C: Flattened sequence
{
  const __ghost_fn __ghost_pair =
      ghost_begin(...)
  float a = compute_value(...);
  ghost_end()
  float x = f(a);
}
```

Figure 3.5: Handling sequence elaboration

3.5 Results

3.5.1 Illustrative Example

Prior to presenting our results, we illustrate the capabilities of the autofocus mechanism through a concrete and representative example: computing the trace of a matrix. The base implementation has two functions: `trace`, which actually computes the matrix trace; and `main`, which calls `trace` and stores the result.

```
#include "optitrust.h"

float trace(float *x, int n) {
  float sum = 0.f;
  for (int i = 0; i < n; i++) {
    sum += x[MINDEX2(n, n, i, i)];
  }
  return sum;
}

int main(float *x, int n) {
  float sum = trace(x, n);
  return 0;
}
```

Figure 3.6: Base implementation of matrix trace computation.

As presented in Figure 3.7, in order to ensure the code is successfully typechecked by the OptiTrust typechecker, we must add function contracts, on both `main` and `trace`.

```
#include "optitrust.h"

float trace(float *x, int n) {
  __reads("for i in 0..n -> &x[MINDEX2(n,n,i,i)] ~> Cell");
  float sum = 0.f;
  for (int i = 0; i < n; i++) {
    sum += x[MINDEX2(n, n, i, i)];
  }
  return sum;
}

int main(float *x, int n) {
  __reads("x ~> Matrix2(n,n)");
  float sum = trace(x, n);
  return 0;
}
```

Figure 3.7: Trace computation with necessary function contracts.

From this state, the autofocus mechanism automatically generates the missing annotations required to verify the code:

- The resource requested by the `trace` function concerns only the diagonal of the matrix. Consequently, the system must focus the entire matrix resource down to the diagonal elements only. This step leverages the propagated instantiation mechanism discussed previously.
- Within the `trace` function loop, the system must further focus on each individual element of the diagonal to allow access.
- We also demonstrate the extraction of ghost operations from expressions. Since the call to `trace` appears on the right-hand side of a let-binding, the system lifts this call, storing the result in a temporary variable `autofocus_tmp`, wrapping the call with the necessary ghost instructions.

Figure 3.8 displays the complete result generated by the autofocus.

```
#include "optitrust.h"

float trace(float* x, int n) {
  __reads("for i in 0..n -> &x[MINDEX2(n, n, i, i)] ~> Cell");
  float sum = 0.f;
  for (int i = 0; i < n; i++) {
    const __ghost_fn __ghost_pair_1 = __ghost_begin(
      [&]() {
        __consumes(
        "_RO(#_1 / 2, for i in 0..n -> &x[MINDEX2(n, n, i, i)] ~> Cell)");
        __produces("_RO(#_1 / 2, &x[MINDEX2(n, n, i, i)] ~> Cell)");
        __admitted();
      });
    const float autofocus_tmp = x[MINDEX2(n, n, i, i)];
    __ghost_end(__ghost_pair_1);
    sum += autofocus_tmp;
  }
  return sum;
}

int main(float* x, int n) {
  __reads("x ~> Matrix2(n, n)");
  const __ghost_fn __ghost_pair_2 = __ghost_begin(
    [&]() {
      __consumes("_RO(#_1 / 2, x ~> Matrix2(n, n))");
      __produces(
        "_RO(#_1 / 2, for i1 in 0..n -> &x[MINDEX2(n, n, i1, i1)] ~> "
        "Cell)");
      __admitted();
    });
  const float autofocus_tmp = trace(x, n);
  __ghost_end(__ghost_pair_2);
  float sum = autofocus_tmp;
  return 1;
}
```

Figure 3.8: Final code generated by autofocus, including ghost extraction and resource focusing.

3.5.2 General Results

We applied the autofocus to real-world scenarios. We utilized the computation kernels upon which LLM inference code depends. These are typically short functions (fewer than 15 lines), their precise utility is not relevant here, but more details can be found in chapter 5.

The table below presents the results obtained. The reported line counts for all code variants were measured after formatting with standard `clang-format` settings, excluding empty lines within functions.

Kernel	Pure	Without Autofocus		With Autofocus	
	Code	Spec	Manual Ghosts	Manual Ghosts	Generated
RoPE	12	1	22	**0**	65
RMSNorm	13	3	8	**0**	68
Softmax	16	1	9	**4**	86
MatVec	9	3	5	**0**	87
MatMul	11	3	5	**0**	155
Case study					
Forward Pass	144	22	Not done	5	1450

Table 3.1: Annotation effort comparison

Several key observations can be drawn from these results:

- In the context of array management code, autofocus significantly reduces the volume of required annotations without removing them entirely. Experience shows that typically only function contracts and certain loop contracts remain, while the majority of ghost operations are automated. For kernels without autofocus, annotations previously added over 70% to the code size; with autofocus, there is almost no need for annotations for array programs.
- The Rotary Positional Embedding (RoPE) kernel is particularly illustrative. This code manipulates array elements in pairs, involving both read and modify operations. Manually annotating this pattern requires a significant number of ghost instructions, which autofocus handles automatically.
- It is important to note that the code generated by autofocus is often more verbose than manually written specifications. A programmer annotating by hand can utilize high-level ghost functions (e.g., `matrix2_focus` to focus a cell in a 2D array in a single line). In contrast, to maintain generality and algorithmic correctness as detailed previously, our system generates atomic ghost operations one step at a time.

While this ensures correctness, it increases the length of the generated code.

Finally, the `forward` function represents a distinct category. As the main entry point for the LLM inference, enabling its verification was one of the primary motivations for creating this feature. We do not provide a "Manual Spec LOC" figure for this function because manually annotating it represents a disproportionate and error-prone effort. The code continuously operates on sub-matrices within loop nests 3 to 4 levels deep, which induces a massive amount of ghost logic that is difficult for programmers to maintain. The autofocus successfully handles this complexity, generating over 1400 lines of specification, demonstrating the scalability of our approach.

Chapter 4

OptiTrust Extensions for Improved Matrix Handling

Beyond the Autofocus mechanism, our case study revealed the need for several important features in OptiTrust. These extensions are all closely related to the manipulation of submatrices, which tends to generate cascading indexing issues in many different forms. Although these features are conceptually simpler to design and integrate than autofocus, they also play an essential role for handling the LLM case study. We describe here the problems we encountered and the solutions we implemented.

4.1 Normalization of Matrix Terms

During the typing process, it is common for array indexing to become nested, which prevents simple syntactic matching between required resources and candidate resources. For instance, consider the code in Figure 4.1:

```
void some_vector_computation(float* x, int n1){
  __requires("for i1 in 0..n1 -> x[MINDEX1(n1,i1)] ~> Cell");
  __admitted();
}

void caller(float* x, int n1, int n2){
  __writes("for i1 in 0..n1 -> for i2 in 0..n2 -> x[MINDEX2(n1,n2,i1,i2)] ~> Cell");
  // We pass a pointer to the start of a row
  some_vector_computation(&x[MINDEX2(n1,n2,0,0)], n2);
}
```

Figure 4.1: Example of needed matrix normalization

We want to call the function on one row of the matrix. Therefore, after

substitution, the *required* resource we are looking for is:

$$\bigstar_{i_2=0}^{n_2} ((\&x[\mathsf{MINDEX2}(n_1, n_2, 0, 0)])[\mathsf{MINDEX1}(n_2, i_2)] \rightsquigarrow \mathsf{Cell})$$

which semantically, corresponds exactly to the *candidate* resource:

$$\bigstar_{i_2=0}^{n_2} (x[\mathsf{MINDEX2}(n_1, n_2, 0, i_2)] \rightsquigarrow \mathsf{Cell})$$

Although these expressions denote the same underlying resource, they do not match syntactically. Consequently, in the original version of the first type-checker pass, the two resources would fail to unify. Furthermore, Autofocus cannot be applied in this context because the array accesses do not share the same base: one term is a resource on the matrix x, while the other is a resource on the pointer $\&x[\mathsf{MINDEX2}(n_1, n_2, 0, 0)]$.

Normalization algorithm. Our proposal is to normalize nested matrix accesses of the form:

$$\&(\&x[\text{MINDEX}_{\text{N}_{\text{in}}}(\text{dims}_{\text{in}}, \text{inds}_{\text{in}}, 0, \dots, 0)])[\text{MINDEX}_{\text{N}_{\text{out}}}(\text{dims}_{\text{out}}, \text{inds}_{\text{out}})]$$

into the simpler form:

$$\&x[\text{MINDEX}_{\text{N}_{\text{in}}}(\text{dims}_{\text{in}}, \text{inds}_{\text{in}}, \text{inds}_{\text{out}})].$$

In the two formulas above $dims_{in}$ and $inds_{in}$ represent lists of dimensions and indices respectively.

To ensure correctness, several conditions must be checked:

- **Addressing base agreement:** both terms must refer to the same base variable x.
- **Dimension compatibility:** Let d^{in} and d^{out} be the dimension arrays. For all $k \in \{0, \dots, N_{\text{out}} - 1\}$, the inner dimensions must unify with the outer dimensions:
  $$d^{\text{in}}_{N_{\text{in}}-k} \text{ unifies with } d^{\text{out}}_k$$
  This ensures that the inner and outer dimensions align correctly.
- **Zero-prefix constraint:** in this first version of the algorithm, normalization is allowed only when the trailing indices of the inner access are zero:
  $$\forall k \in \{0, \dots, N_{\text{out}} - 1\}, \quad \text{inds}_{\text{in}}[\text{k}] = 0,$$
  enabling a direct replacement of trailing zeros by inds_{out}.

Integration in OptiTrust. We insert the normalization phase before the syntactic matching phase. Concretely, whenever an algorithm attempt to determine whether two terms may syntactically match, both terms are first passed through our normalization procedure.

This preprocessing step rewrites the original expressions into their normalized counterparts, ensuring that the matching algorithm operates on canonical representations rather than on raw, potentially non-uniform terms. Normalization is applied recursively and in a bottom-up fashion: each subterm is visited, and all occurrences of nested matrix accesses are normalized whenever the conditions from the previous section are satisfied. Let us illustrate how normalization applies in cascade. A term such as:

$$(\&(\&x[\text{MINDEX3}(n_1, n_2, n_3, i_1, 0, 0)])[\text{MINDEX2}(n_2, n_3, i_2, 0)])[\text{MINDEX1}(n_3, i_3)]$$

would be transformed into the single-level canonical form:

$$x[\text{MINDEX3}(n_1, n_2, n_3, i_1, i_2, i_3)].$$

4.2 Matching Extension for Uninlining

A similar difficulty related to syntactic matching arises in the context of *uninlining*. The uninlining transformation attempts to replace a sequence of instructions t_ℓ with a function call whose body t_f performs the same computation. Crucially, this transformation must determine the specific arguments for the call such that the operation acts as the exact inverse of inlining. This identification process relies on *pattern matching*, which proceeds structurally: each subterm of t_f must syntactically match the corresponding subterm of t_ℓ, possibly instantiating arguments of f along the way.

Array-access problem. As shown in Figure 4.2, a unification failure occurs when the function parameter acts as a slice of a larger array. In the function `sum_row`, the computation is defined over the array y using the indexing function MINDEX_1. However, the target sequence in `sum_matrix` operates on x using the indexing function MINDEX_2. Using syntactic pattern matching, it is impossible to find an instantiation of y that succeeds, as the access terms do not match structurally. Therefore, we must find a solution to expose the indexing function expected by t_f.

Note that this problem is the inverse of the normalization described in the previous section. Here, instead of flattening nested accesses, we must decompose a higher-order access found in the target code into a nested form to match the function body.

```
float sum_row(float *y, int n1){
  float sum = 0;
  // The function body includes a MINDEX1 on parameter y
  for(int i1 = 0; i1 < n1; i1++){
    sum += y[MINDEX1(n1,i1)];
  }
  return sum;
}

float* sum_matrix(float *x, int n1, int n2){
  float* sums = MALLOC1(float, SIZE1(n1));
  for(int i1 = 0; i1 < n1; i1++){
    float sum_row = 0;
    // The target code contains a MINDEX2 on variable x
    for(int i2 = 0; i2 < n2; i2++){
      sum_row += x[MINDEX2(n1,n2,i1,i2)];
    }
    sums[i1] = sum_row;
  }
  return sums;
}
```

Figure 4.2: Example of unapplicable uninlining: there is no syntactic match between MINDEX2 and MINDEX1

Implementation insight. To resolve this, we rely on the pointer arithmetic identity mentioned in the previous section. While normalization flattens accesses, we use the identity here to decompose a higher-order access into an access on a subarray. Specifically, we identify that the access $x[\mathrm{MINDEX}_2(n_1, n_2, i_1, i_2)]$ is semantically equivalent to accessing the index i_2 of a subarray starting at $x[\mathrm{MINDEX}_2(n_1, n_2, i_1, 0)]$. By rewriting the target term into this nested form:

$$(\&x[\mathrm{MINDEX}_2(n_1, n_2, i_1, 0)])[\mathrm{MINDEX}_1(n_2, i_2)]$$

the syntactic matcher can successfully unify the function parameter y with the address of the subarray, i.e., $y \mapsto \&x[\mathrm{MINDEX}_2(n_1, n_2, i_1, 0)]$.

Integration in Optitrust First, to avoid cascading mismatches due to previously introduced non-normalized forms, we always begin by normalizing both terms as described in the previous section. During matching, if t_1 is an access in the function body, and t_2 is an access in the unfolded instruction sequence, then we extract the indexing structure of both terms and attempt to rewrite t_2 into the form expected by t_1. This extension enables the uninlining transformation as shown in Figure 4.3.

4.3 Tiling Access Transformation

We now introduce a modest but practically useful transformation for matrix tiling: if we have at hand an array viewed as a 1-dimensional array accessed

Transformation Script

```
Function.uninline [cFunDef "sum_matrix"; cForBody "i1" ]
```

Before

```
float sum_row(float *y, int n1){
  float sum = 0;
  for(int i1 = 0; i1 < n1; i1++){
    sum += y[MINDEX1(n1,i1)];
  }
  return sum;
}
float* sum_matrix(float *x, int n1, int n2){
  float* sums = MALLOC1(float, SIZE1(n1));
  for(int i1 = 0; i1 < n1; i1++){
    float sum_row = 0;
    for(int i2 = 0; i2 < n2; i2++){
      sum_row += x[MINDEX2(n1,n2,i1,i2)];
    }
    sums[i1] = sum_row;
  }
}
```

After

```
float sum_row(float *y, int n1){
  float sum = 0;
  for(int i1 = 0; i1 < n1; i1++){
    sum += y[MINDEX1(n1,i1)];
  }
  return sum;
}
float* sum_matrix(float *x, int n1, int n2){
  float* sums = MALLOC1(float, SIZE1(n1));
  for(int i1 = 0; i1 < n1; i1++){
    // x[MINDEX2] acts as arg for sum_row
    float sum_row_res =
      sum_row(&x[MINDEX2(n1,n2,i1,0)], n2);
    sums[i1] = sum_row_res;
  }
}
```

Figure 4.3: Uninlining transformation example

using MINDEX1, we may wish to now view the same array as a 2-dimensional array accessed using MINDEX2, for example to align with the way the code iterates over the cells of this array.

Motivating example. The example in Figure 4.4 shows that an implicit tiling occurs in the data accesses, but this structure is invisible in the data layout and only appears through the indexing arithmetic inside MINDEX_1. The value of explicit tiling arises mainly in combination with other transformations. For instance, once the tiled layout is explicit, OptiTrust can isolate computations that operate on an entire block and use subsequent transformations (e.g., to

pre-load tile bases).

Transformation Syntax

```
Matrix.tile(
      ~block_size:(trm_find_var "bs" [])
      ~nb_block:(trm_find_var "nb" [])
      ~index_dim:0  (* 0 for the first dimension *)
      [ cVarDef "x" ])
```

Before

```
void tiling_example(int n){
  if (n % 4 != 0) return;
  float* x = MALLOC1(float, SIZE1(n));
  int nb = n/4;
  int bs = 4;
  float sum = 0;
  for(int i1 = 0; i1 < nb; i1++){
    for(int i2 = 0; i2 < bs; i2++){
      // Implicit tiling via arithmetic
      sum += x[MINDEX1(n, i1*bs + i2)];
    }
  }
}
```

After

```
void tiling_example(int n){
  if (n % 4 != 0) return;
  // x is now a 2D array in logical view
  float* x = MALLOC2(float, SIZE2(nb,bs));
  int nb = n/4;
  int bs = 4;
  float sum = 0;
  for(int i1 = 0; i1 < nb; i1++){
    for(int i2 = 0; i2 < bs; i2++){
      // Access rewritten with / and %
      sum += x[MINDEX2(nb, bs,
                (i1*bs + i2) / bs,
                (i1*bs + i2) % bs)];
    }
  }
}
```

Figure 4.4: Tiling transformation: Syntax and Effect. The 1D array is logically viewed as 2D blocks.

Implementation. Throughout this section, we write *nb* for the number of blocks and *bs* for the block size. The user first specifies the block size *bs* and optionally the number of blocks *nb*. If *nb* is not provided, we insert in the code a statement to compute its value using the formula:

$$nb = \left\lfloor \frac{n + bs - 1}{bs} \right\rfloor$$

This formula ensures coverage of all the cells even if n is not divisible by bs. Currently, this produces a padded solution. In the future, OptiTrust will generate a proof obligation $n \bmod bs = 0$ to enforce strict divisibility, but this verification mechanism is not yet implemented. The user must also specify the `index_dim` on which he wants to apply tiling. Finally, The *target* for this transformation must be the variable declaration on which tiling applies.

Once triggered, the variable declaration is rewritten with updated dimensions. Furthermore, every access to this array is rewritten. If we target the k-th dimension of an array of rank L, the access is transformed from a MINDEX_L to a MINDEX_{L+1}. The dimension n_k is split into nb and bs, and the index i_k is split using Euclidean division. Formally, the access pattern:

$$\text{MINDEX}_L(n_1, \ldots, n_k, \ldots, n_L, \quad i_1, \ldots, i_k, \ldots, i_L)$$

is rewritten into:

$$\text{MINDEX}_{L+1}(n_1, \ldots, nb, bs, \ldots, n_L, \quad i_1, \ldots, i_k/bs, i_k \bmod bs, \ldots, i_L)$$

Correctness Proof. Let us prove that the rewriting is correct with respect to the definition of the `MINDEX` macros. Recall that:

$$\text{MINDEX}_L(n_1, \ldots, n_L; q_1, \ldots, q_L) = \sum_{i=1}^{L} \left(q_i \cdot \prod_{j=i+1}^{L} n_j \right).$$

Suppose that the k-th dimension is being transformed. After tiling, the new array has dimensions:

$$(n_1, \ldots, n_k/bs, \ bs, \ldots, n_L),$$

and the access index q_k is replaced by the pair $(\lfloor q_k/bs \rfloor, \ q_k \bmod bs)$. The corresponding linearized index becomes:

$$\begin{aligned}
&\text{MINDEX}_{L+1}(\ldots) \\
&\quad = \sum_{i=1}^{k-1} q_i \cdot \Big(\prod_{j=i+1}^{L} n_j \Big) \quad \text{(dims product is unchanged)} \\
&\qquad + \left\lfloor \frac{q_k}{bs} \right\rfloor \cdot \left(bs \cdot \prod_{j=k+1}^{L} n_j \right) \\
&\qquad + (q_k \bmod bs) \cdot \left(\prod_{j=k+1}^{L} n_j \right) \\
&\qquad + \sum_{i=k+1}^{L} \left(q_i \cdot \prod_{j=i+1}^{L} n_j \right).
\end{aligned}$$

By factoring out the term $\prod_{j=k+1}^{L} n_j$ in the middle parts, and using the identity:

$$bs \cdot \left\lfloor \frac{q_k}{bs} \right\rfloor + (q_k \bmod bs) = q_k,$$

we recover exactly:

$$\sum_{i=1}^{L} \left(q_i \cdot \prod_{j=i+1}^{L} n_j \right),$$

which is precisely the original MINDEX_L expression. Thus, the tiling transformation strictly preserves the linearized memory layout.

Chapter 5

Large Language Models (LLMs)

5.1 Introduction

5.1.1 Purpose of the Case Study

The purpose of the case study considered in my Master thesis is to demonstrate the practical relevance of OptiTrust through the optimization of a real-world codebase, specifically LLM inference.

Large Language Model codes represent a relevant target for OptiTrust. Such codes possess enough complexity that fully automatic tools fail to produce optimal performance, yet the source code remains small enough to handle within a research prototype. Our approach began by retrieving a Transformer code performing realistic calculations without prior optimization. We then attempted to apply transformations at "Level 1" (without formal validity guarantees) to the source. A major objective involved typing the code effectively. Proper typing paves the way for "Level 2" or "Level 3" verifications, which would eventually provide strong guarantees regarding the correctness of the generated code.

The subsequent section will describe the structure of Large Language Models, as understanding the architecture is necessary to explain the applied transformations.

5.1.2 Overview of LLMs

General Presentation

Large Language Models (LLMs) are artificial intelligence systems designed to process, understand, and generate human language. In practice, most modern LLMs rely on the *Transformer* architecture, which forms the backbone of models such as GPT, LLaMA, or Mistral. The LLM is structured around two main phases: training and inference. The *training phase* is a data-intensive

process where the model performs self-supervised learning to predict the next token in a sequence, ultimately producing a set of weight matrices and normalization parameters. However, this case study does not cover the training codebase, which is computationally expensive and typically restricted to large GPU clusters. Instead, we focus exclusively on the *inference phase*, which is extensively described in the following section.

LLM Inference

The goal of the inference phase is to predict the next token given a prompt and the model parameters obtained during training. Since this part of the process corresponds to the code we analyze in this case study, we will detail its main computational stages.

Inference is typically divided into two main stages:

1. **Prompt Processing**, which encodes the user input into internal representations;
2. **Token Generation**, which iteratively produces new tokens based on previously generated ones.

Prompt Processing During prompt processing, the textual input is first tokenized, i.e., converted into a sequence of integer identifiers corresponding to entries in the model vocabulary. For each token, the model retrieves its associated *embedding vector*, which is then processed through a series of Transformer layers.

Each layer performs several key operations designed to enrich the representation of each token with contextual information:

- **Normalization**, such as layer normalization, which stabilizes the distribution of activations and improves convergence;
- **Matrix-vector multiplications** with learned weight matrices, used to compute the query (Q), key (K), and value (V) representations;
- **Attention computation**, which combines the Q, K, and V matrices to produce context-aware embeddings.
- **Feed-forward network**, which applies matrix-vector multiplications with learned weight matrices to each position independently, enhancing the representational capacity of the model.

The result of these transformations is a set of enriched embeddings that encode both the content and context of the input prompt.

Token Generation The token generation phase operates iteratively. The enriched embeddings obtained from the prompt is projected onto a *logit*

matrix, yielding a probability distribution over all tokens in the vocabulary. A sampling procedure (e.g., greedy decoding, top-k sampling, or nucleus sampling) then selects the next token according to this distribution. This new token is appended to the input sequence, and the process repeats.

The generation stops either when a special end-of-sequence token is produced, or when a predefined token limit is reached. This iterative procedure shares the same computational structure as prompt processing but includes the additional sampling step.

5.2 Fitting the Original Code into OptiTrust

5.2.1 Llama2.c

For our case study, we based our analysis on the implementation `llama2.c` [16], a concise and efficient standalone version of the LLaMA 2 inference code, widely recognized as a minimalist reference implementation. This program, composed of roughly one thousand lines of C, encapsulates the full inference pipeline of a transformer-based language model. The core of the computation, the forward pass described in the previous section, occupies approximately 300 lines. This code provides an excellent foundation for our study because it is fully self-contained, requiring no external dependencies or frameworks, making it easy to compile, modify, and reason about. This makes it particularly well-suited for integration into OptiTrust, where transformations and benchmarking can be performed with minimal adaptation.

5.2.2 Code Adjustments for OptiTrust

Since OptiTrust operates on a subset of the C language, several preliminary adaptations were necessary to make the code compatible with the framework. OptiTrust is particularly well-suited for programs that rely on array-based computations and already provides a wide range of verified transformations, but its current feature set is not exhaustive. The goal of this case study is to leverage OptiTrust rather than extend its entry language. Therefore, we chose to slightly modify the input code instead of adapting the framework itself to support the original version.

Structure Flattening. The original `llama2.c` code uses hierarchical data structures to store model parameters and intermediate buffers. In order to make these structures analyzable within OptiTrust, we flattened them into explicit arrays. This transformation simplifies memory access patterns and allows us to reason more easily about aliasing and ownership. Moreover, in the original version, intermediate vectors were pre-allocated within these structures. In our adaptation, a new vector is allocated for each token before entering the forward pass, in order to facilitate subsequent transformations.

Array Indexing with `mindex`. OptiTrust requires explicit indexing through a built-in function called `mindex`, which computes memory offsets based on the dimensions of multi-dimensional arrays, instead of performing manual pointer arithmetic. This change enhances both readability and analyzability, enabling the framework to maintain strict control over bounds and dimensions. Therefore, with the two changes that have been described, for a piece of code such as:

```
int loff = l * p->seq_len * kv_dim; //layer offset
        s->k = s->key_cache + loff + pos * kv_dim;
```

I manually rewrote it into:

```
float* k = k_cache[MINDEX4(l_count, kv_hc,
                    context_len,head_dim, l, h, i, 0)];
```

Kernels instead of Inline Computations In the original implementation, several normalization and transformation operations such as RMS normalization and RoPE were implemented inline. To make these operations more modular and easier to type-check, we extracted them into separate kernel functions. This uninlining process leverages the type system and contract-based verification of OptiTrust: annotating and verifying independent kernels is significantly simpler than reasoning about inlined computations spread throughout the code. In later sections, we will return to these design choices when discussing the annotation and validation of the transformation scripts.

5.3 Transformations

In the following, we describe a series of transformations designed to improve both parallelism and data locality. Each optimization follows a consistent structure: we first outline the underlying intuition, then detail the transformation sequence. For every low-level step, we display the corresponding OptiTrust script line alongside a before/after code comparison.

5.3.1 Token Batching

Description

The first and most impactful optimization is token batching, a transformation systematically used in modern LLM inference codebases. During prompt processing, the model fills the key-value (KV) cache for each layer, but no new token needs to be produced before completing this step. Therefore, apart from the attention mechanism, which remains inherently sequential because computing attention for a token requires the KV cache of all previous tokens, most operations across tokens are independent. This includes normalization,

Before

```
// Loop over tokens (outermost)
for (int i = 0; i < seq_len; i++) {
  // Allocation per token
  float* t = malloc(..);

  // Embedding representation
  for (int l = 0; l < layer_count; l++) {
    // Weight matrix loaded seq_len times
    for(int q = 0; q < q_hc; q++){
      matvec(t, ...);
    }
    for(int q = 0; h < q_hc; q++){
      matvec(t, ...);
    }
    // other layer processing (7 loop nests)
  }
  // final normalization for the current token
}
```

After

```
// Hoisted allocation for all tokens
float* t = malloc(.. * seq_len);

// Embedding representation for every token
for (int l = 0; l < layer_count; l++) {

  // Loop over tokens pushed innermost
  for(int q = 0; q < q_hc; q++){
    for (int i = 0; i < seq_len; i++) {
      // Weight matrix loaded once, reused for all i
      matvec(t[i], ...);
    }
  }

  for(int q = 0; h < q_hc; q++){
    for (int i = 0; i < seq_len; i++) {
      matvec(t[i], ...);
    }
  }
  // other layer processing (7 loop nests)
}
// final normalization for every token
```

Figure 5.1: Token Batching: Inverting loop order to maximize weight reuse.

KV cache computation, and the feed-forward network. Batching tokens provides a significant benefit in terms of memory locality and cache efficiency. In the original implementation, weight matrices (for an embedding dimension of 4096 it leads to matrices of size 32 MB) are loaded sequentially for each kernel computation. As shown in Figure **??**, the size of the matrices loaded during each subphase exceeds the capacity of typical L2 ($\approx 1\,MB$) and even L3 caches ($\approx$ 30 to 60 MB). This results in frequent cache evictions. Processing multiple tokens simultaneously allows each matrix to be loaded only once per layer and reused across all tokens, significantly improving

cache locality and arithmetic intensity.

As shown in Figure 5.1, the token batching strategy consists of pushing the token loop (`for i...`) as deep as possible, placing it immediately around the computation kernels. To achieve this result, we must devise a sequence of low-level transformations using OptiTrust primitives, this optimization is realized through a combination of loop reordering, loop fission, and variable *hoisting*, as detailed in the following paragraphs.

Transformation Strategy

Transformation Script

```
Function.inline [ f; cCall "forward" ]
```

Before

```
void generate_prompt_proc(...) {
 for (int i = 0; i < seq_len; i++){
  forward(...)
 }
}
```

After

```
void generate_prompt_proc(...) {
for (int i = 0; i < seq_len; i++) {
    // embedding representation
    for (int l = 0; l < l_count; l++) {
      // layer processing
    }
    //final normalization and freeing
  }
}
```

Figure 5.2: Inlining forward transformation

Inlining. We start by inlining the `forward` function. As shown in Figure 5.2, the effect is to unfold the computation, exposing the nested loops directly at the level of the main routine. This inlining transformation allows us to manipulate the control flow around the loops that we aim to transform.

Hoisting. As stated before, originally, each intermediate vector is allocated independently for every token. Since our goal is to compute all tokens in parallel, we *hoist* these definitions outside the per-token loops. Instead of redefining a vector for each token, we allocate a two-dimensional array of size `n_token` × `vec_size`. Each access is then indexed by the token index i, ensuring distinct storage for every token while preserving data independence. The result of this transformation can be seen in Figure 5.3.

Transformation Script

```
Loop.hoist [nbMulti; cVarDefs ["embedding"; "mha_norm"; ..]]
```

Before

```
for (int i = 0; i < seq_len; i++) {
    float* const embedding =
        (float*)malloc(MSIZE1(embedding_dim)
        * sizeof(float));
    float* const mha_norm =
        (float*)malloc(MSIZE1(embedding_dim)
        * sizeof(float));

    // Accesses are local to the iteration
    ... embedding[MINDEX1(embedding_dim, k)] ...
}
```

After

```
float* const embedding =
    (float*)malloc(
        MSIZE2(seq_len, embedding_dim)
        * sizeof(float));
float* const mha_norm =
    (float*)malloc(
        MSIZE2(seq_len, embedding_dim)
        * sizeof(float));

for (int i = 0; i < seq_len; i++) {
    // Accesses now include the loop index i
    ... embedding[MINDEX2(seq_len, embedding_dim, i, k)] ...
}
```

Figure 5.3: Hoisting definitions of temporary vectors

Loop Fission (1). As shown in Fig 5.4, the next step is to separate three main computational phases: the transformation from tokens to their initial embeddings, the iteration over the Transformer layers, and the final normalization step.

Loop Reordering (1). We then focus on the loop over layers and swap the order of the loops over tokens (i) and layers (l). This reordering allows each layer to process all tokens in batch before moving to the next one, as illustrated in Figure 5.5.

Loop Fission (2). Once the per-layer processing loop encompasses all tokens, we perform an additional loop fission to isolate independent operations within each layer iteration.This step allows us to compute the KV cache, the attention mechanism, and the feed-forward network sequentially across all tokens, processing each operation for the entire batch before moving on to the next phase.

Transformation Script

```
Loop.fission [f; cForBody "i"; tBetweenAll];
```

Before

```
for (int i = 0; i < seq_len; i++) {
    // embedding encoding
    ...
    for (int l = 0; l < l_count; l++) {
        // layer processing
        ...
    }
    // final normalization
}
```

After

```
for (int i = 0; i < seq_len; i++) {
    // embedding encoding
    ...
}
for (int i = 0; i < seq_len; i++) {
    for (int l = 0; l < l_count; l++) {
        // layer processing
        ...
    }
}
for (int i = 0; i < seq_len; i++) {
    // final normalization
    ...

}
```

Figure 5.4: Loop fission transformation

Loop Reordering (2). Inside each layer, we iterate over the attention heads. Given the layout of the data, where weight matrices are stored per head (one matrix per head and per layer), it is more efficient to invert the order of the loops over tokens (i) and heads (h).

The final loop fission and reordering steps are presented in Figure 5.6. As the effects of these individual transformations have already been detailed, we display them combined.

5.3.2 Recognizing MatMul

Description

With the current code structure, the pattern shown in Figure 5.7 becomes recognizable. We can exploit this structure to transform the sequence of matrix-vector operations into a single matrix-matrix multiplication.

As shown in Figure 5.8, matrix-matrix multiplication represents the vast majority of the computation time; therefore, it is crucial to use a highly

Transformation Script

```
Loop.reorder_at ~order:["l"; "i"] [f;cForBody "l";dSeqNth 0];
```

Before

```
for (int i = 0; i < seq_len; i++) {
    for (int l = 0; l < l_count; l++) {
        // layer processing
        ...
    }
}
```

After

```
for (int l = 0; l < l_count; l++) {
    for (int i = 0; i < seq_len; i++) {
        // layer processing
        ...
    }
}
```

Figure 5.5: Reordering loops on tokens and layers

optimized version of this kernel. While implementing such an optimized kernel was outside the scope of this master thesis, OptiTrust is designed to support this workflow. A user can implement an optimized version of `matmul` that leverages dedicated hardware support or low-level primitives, formally prove its semantic equivalence to the naive version, and then simply swap one for the other in the case study without re-verifying the entire pipeline.

Transformation Strategy

Uninlining. As shown in Figure 5.9, the first step is to *inline* each `matvec` operation, unfolding it to expose the underlying computation. The granularity of OptiTrust allows us to unfold only the specific `matvec` operations of interest. For example, the KV-cache spans the entire context and does not correspond to a strict submatrix multiplication; for simplicity, we do not treat it here.

MatMul Recognition. Thanks to the pattern-matching extension described in Section 4.2, OptiTrust can now recognize matrix multiplications on subarrays. This enables the *uninlining* of the Q-value computation and the feed-forward network operations, allowing them to be treated as standard matrix multiplications, which can then be optimized more effectively.

5.3.3 Strip Mining

The number of attention heads used for the Q-values typically differs from the number of heads used for the KV cache. Typical configurations are

Transformation Script

```
Loop.fission [cForBody "l"; cForBody "i"; tBetweenAll];
Loop.reorder_at ~order:["q"; "i"]
                [nbMulti; cForBody "q"; dSeqNth 0];
Loop.reorder_at ~order:["h"; "i"]
                [nbMulti; cForBody "h"; dSeqNth 0];
```

Before

```
for (int l = 0; l < l_count; l++) {
  // Single loop over tokens
  for (int i = 0; i < seq_len; i++) {
    for(int q = 0; q < q_hc; q++){ ... }
    for(int h = 0; h < kv_hc; h++){ ... }
  }
}
```

After

```
for (int l = 0; l < layer_count; l++) {
  for(int q = 0; q < q_hc; q++){
    for (int i = 0; i < seq_len; i++) {
      ...
    }
  }
  for(int h = 0; h < kv_hc; h++){
    for (int i = 0; i < seq_len; i++) {
      ...
    }
  }
}
```

Figure 5.6: Applying loop fission and reordering to separate Q and KV processing phases.

```
for (int i = 0; i < seq_len; i++){
  matvec(seq[i], ...);
}
```

Figure 5.7: Code pattern identifying a sequence of matrix-vector multiplications, amenable to transformation into a matrix-matrix multiplication.

$q_hc = 32$ and $kv_headcount = 8$.

Despite this difference in dimensions, the computed vectors participate in the same computational process. Consider the following example: for each embedding vector, we compute an `mha_score` matrix of dimensions `q_head_count` × `head_dim`. We can visualize this as computing an "attention score" for each head dimension. For every vector, the score is calculated using the standard attention formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

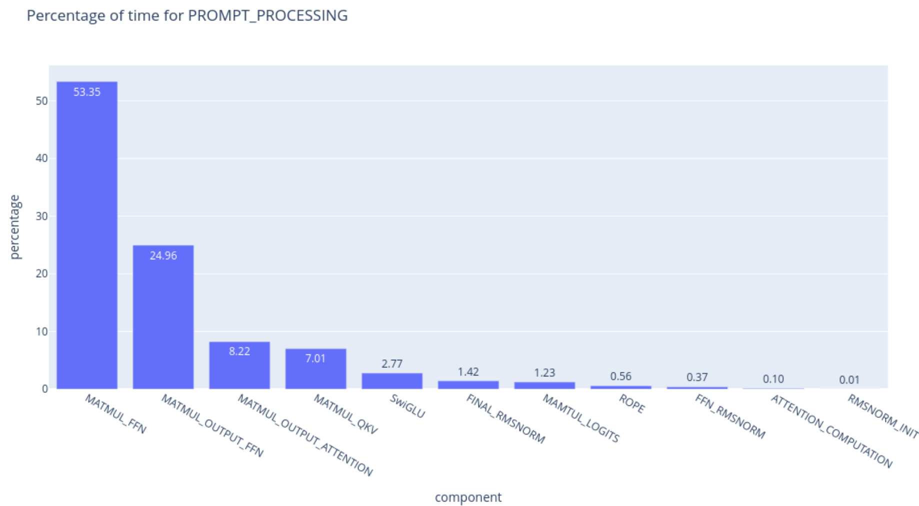

Figure 5.8: Distribution of execution time across the different computation kernels

Transformation Script

```
Function.inline [nbMulti; cCall "matvec"; occIndices [ 0; 3; 4; 5 ]]
```

Before

```
for (int i = 0 ; i < seq_len;i++){
  matvec(...);
}
```

After

```
for(int i = 0; i < seq_len; i++){
    for(int j = 0; j < dim; j++){
        for(int k = 0; k < dim; k++){ ... }
    }
 }
```

Figure 5.9: Inlining matvec

where:

- Q is the query matrix (derived from `mha_q`),
- K is the key matrix (derived from `k_cache`),
- d_k is the dimension of the keys (used for scaling).

In our specific context, we focus on the score computation (QK^T). A naive implementation of this calculation is shown in Figure 5.11.

However, during this attention computation, the same `k_cache` matrix is reused across multiple heads of the `mha_score`. Specifically, groups of query

Transformation Script

```
Function.uninline ~f:[cFunDef "matmul"] [nbMulti; cFor "i"]
```

Before

```
for(int i = 0; i < seq_len; i++){
    for(int j = 0; j < dim; j++){
        for(int k = 0; k < dim; k++){ ... }
    }
 }
```

After

```
matmul(...);
```

Figure 5.10: Uninlining matmul

```
for (int q = 0; q < q_hc; q++) {
    // Implicit mapping from query head to KV head
    int h = q / q_per_kv;

    for (int p = 0; p <= pos; p++) {
        mha_score[MINDEX2(q_hc, con_len, q, p)] = 0.f;
        for (int e = 0; e < head_dim; e++) {
            mha_score[MINDEX2(q_hc, con_len, q, p)] +=
                mha_q[MINDEX2(q_hc, h_dim, q, e)] *
                k_cache[MINDEX4(l_count, kv_hc, con_len, h_dim,
                                l, h, p, e)];
        }
        mha_score[MINDEX2(q_hc, con_len, q, p)] /= sqrtf(h_dim);
    }
}
```

Figure 5.11: Naive implementation for attention computation.

heads in the range $[k \times q_per_kv, (k+1) \times q_per_kv]$ share the same key-value head k.

In the naive implementation (Figure 5.11), the code fails to explicitly reflect this dependency. The relationship is obscured by the use of integer arithmetic to derive the KV index h from the query index q:

```
int h = q / q_per_kv;
```

This arithmetic dependency hides the structure of the data reuse from the compiler. Consequently, we aim to perform a *strip mining* transformation on the loops. The objective is to improve data locality and enable the compiler to perform lower-level optimizations, such as vectorization and more efficient thread dispatching.

Transformation Strategy

Our strategy follows a three-step process: explicitly tiling the data structures, tiling the loops to match the data layout, and simplifying the resulting arithmetic expressions. The ultimate goal is to obtain direct accesses of the form `mha_q[h2][q2][b][c]` without any complex arithmetic overhead.

Matrix Tiling. We begin with a preliminary data tiling step, illustrated in Fig 5.12. As previously noted, any matrix containing the dimension `q_head_count` uses a data layout that does not naturally reflect the computation order or its relationship with matrices dimensioned by `kv_head_count`.

To handle this discrepancy, we apply a transformation to make this layout explicit. Since, by construction, $q_hc = kv_headcount \times q_per_kv$, we can reshape the arrays. In the full script, we apply this transformation to three matrices: `mha_score`, `mha_blend`, and `mha_q`. For the sake of brevity in this illustration, we demonstrate the transformation on `mha_q` only. We specify both the block size and the number of blocks to reshape the flat array into a nested structure.

Transformation Script

```
Matrix.tile ~block_size:q_per_kv ~nb_blocks:kv_hc
      ~index_dim:0
      [cVarDef "mha_q"];
```

Before

```
float* const mha_q = (float*)MALLOC3(
    (q_hc, token_count, head_dim) *
    sizeof(float));
..
for (int q = 0; q < q_hc; q ++){
    // accesses are mha_q[q][b][c]
}
```

After

```
float* const mha_q = (float*)MALLOC4(
    (kv_hc,q_per_kv,
    token_count, head_dim) *
    sizeof(float));
..
for (int q = 0; q < q_hc; q ++){
    // accesses are mha_q[q/q_per_kv][q\%q_per_kv][b][c]
}
```

Figure 5.12: Matrix tiling transformation applied to `mha_q`.

Loop Tiling. To align the control flow with this new data representation, we use the `Loop.grid_enumerate` transformation as shown in Fig 5.13. This

Transformation Script

```
Loop.grid_enumerate [("h2", kv_hc);("q2", q_per_kv)]
        [nbMulti; cFor "q"];
```

Before

```
float* const mha_q = (float*)MALLOC4(
    (kv_hc,q_per_kv,
    token_count, head_dim) *
    sizeof(float));
..
for (int q = 0; q < q_hc; q ++){
// accesses are mha_q[q/q_per_kv][q\%q_per_kv][b][c]
}
```

After

```
float* const mha_q = (float*)MALLOC4(
    (kv_hc,q_per_kv,
    token_count, head_dim) *
    sizeof(float));
..
for(int h2 = 0; h2 < kv_hc; h2++) {
    for(int q2 = 0; q2 < q_per_kv; q2++) {
        const int q = h2 * q_per_kv + q2;
        ..
        // accesses are mha_q[q/q_per_kv][q\%q_per_kv][b][c]
    }
}
```

Figure 5.13: Loop tiling via grid enumeration.

allows us to decompose a single iteration space into nested loops.

Specifically, we replace the loop iterating over q with tiled loops using a tile width of `q_per_kv`. The transformation identifies the target loops, replaces the iteration variables, and reconstructs the original index q.

Arithmetic Simplification. To complete the strip mining process, we must perform several arithmetic simplifications. The array accesses currently rely on complex division and modulo operations. However, because the loop structure now enforces $q = h2 \times q_per_kv + q2$, we can simplify these expressions.

First, as shown in Fig 5.14, we inline the definition of q. This results in indices such as $(h2 \times q_per_kv + q2)/q_per_kv$. Next, as demonstrated on Fig 5.15 we apply specific arithmetic rewrite rules to eliminate the division and modulo operators. Crucially, this yields the exact structure we targeted: accesses are now of the clean form `mha_q[h2][q2][b][c]`, completely devoid of non-trivial index calculations. This explicitly correlates the KV cache usage, where one KV cache matrix is used for a block of `q_per_kv` query matrices.

Transformation Script

```
Variable.inline [nbMulti; f; cVarDef "q"];
```

Before

```
..
for(int h2 = 0; h2 < kv_hc; h2++) {
    for(int q2 = 0; q2 < q_per_kv; q2++) {
    const int q = h2 * q_per_kv + q2;
    ..
    // accesses are mha_q[q/q_per_kv][q\%q_per_kv][b][c]
    }
}
```

After

```
for(int h2 = 0; h2 < kv_hc; h2++) {
    for(int q2 = 0; q2 < q_per_kv; q2++) {
    ..
    // accesses are now :
    // mha_q[(h2 * q_per_kv + q2)/q_per_kv]
    //       [(h2 * q_per_kv + q2)\%q_per_kv][b][c]
    }
}
```

Figure 5.14: Inlining the variable q.

Transformation Script

```
Rewrite.equiv_at "int i; int j ; int k; ==> (i*j +k) /j == i"
~indepth:true [];
Rewrite.equiv_at "int i; int j ; int k; ==> (i*j +k) \%j == k"
~indepth:true [];
```

Before

```
for(int h2 = 0; h2 < kv_hc; h2++) {
    for(int q2 = 0; q2 < q_per_kv; q2++) {
    ..
    // accesses are now :
    // mha_q[(h2 * q_per_kv + q2)/q_per_kv]
    //       [(h2 * q_per_kv + q2)\%q_per_kv][b][c]
    }
}
```

After

```
for(int h2 = 0; h2 < kv_hc; h2++) {
    for(int q2 = 0; q2 < q_per_kv; q2++) {
    ..
    // accesses are now :
    // mha_q[h2][q2][b][c]
    }
}
```

Figure 5.15: Arithmetic simplification of array indices.

5.4 Results

The primary objective of applying these transformations was to demonstrate OptiTrust's capacity to automatically deploy standard optimization patterns on complex codebases. The goal was to validate the feasibility of the approach rather than to immediately outperform state-of-the-art implementations.

Metric and Setup. We evaluate performance using *throughput*, measured in tokens per second (tok/s). This metric represents the number of tokens processed by the model per second. The size of the input prompt is a critical parameter: a prompt that is too small fails to exhibit the memory-bound characteristics that token batching aims to solve. Consequently, we conducted our experiments using a prompt size of 800 tokens. Measurements were performed on non-parallelized implementations, repeating the execution multiple times to report the average throughput.

Performance Analysis. Table 5.1 summarizes our findings. The baseline `llama2.c` implementation achieves 1.2 tok/s. Our OptiTrust-optimized version reaches 5.6 tok/s, representing a speedup of 4.6×. Table 5.1 summarizes our findings. The baseline `llama2.c` implementation achieves 1.2 tok/s, while our OptiTrust-optimized version reaches 5.6 tok/s, representing a speedup of 4.6×. Crucially, this gain stems entirely from the token batching strategy, achieved without relying on external optimized matrix multiplication libraries.

We include `llama.cpp`[14] as a reference point. `llama.cpp` is a state-of-the-art, highly optimized C++ inference engine designed for efficient LLM execution on consumer hardware. While our current implementation significantly improves upon the baseline, it remains slower than `llama.cpp` (15.2 tok/s). This performance gap is expected, as `llama.cpp` leverages advanced algorithmic optimizations not yet covered in this case study, most notably Flash Attention [11] and Speculative Decoding [18]. It is worth noting that the matrix multiplication implementation in `llama.cpp` is also naive.

Implementation	Details	Throughput	Speedup
Llama2.c	Input of our work	1.2 tok/s	1.0×
OptiTrust	**Output of our work**	**5.6 tok/s**	**4.6×**
Llama.cpp	Reference	15.2 tok/s	12.7×

Experimental Setup: HP EliteBook 840 G11 (Ubuntu 24.04, Linux 6.14). Compiled with GCC 13 (Single Thread execution).

Table 5.1: Throughput comparison between the baseline, our OptiTrust implementation, and the reference library.

Chapter 6

Conclusion

6.1 Summary of Contributions

In this Master thesis, I addressed the dual challenge of writing high performance code while ensuring its correctness. This Master thesis contributed several improvements to OptiTrust, a framework that aims to tackle the dual challenge of producing high-performance code and providing formal correctness guarantees for this code. While modern compilers offer automated optimizations, they often fail to implement aggressive structural changes required for peak performance. Conversely, manual optimization leads to complex, unmaintainable, and error-prone code. The OptiTrust framework offers a solution by separating the functional unoptimized code from the optimization script, and by leveraging transformations able to preserve separation logic derivations.

The starting point of this Master thesis was to consider a realistic piece of software, namely a LLM inference code, whose scale was much larger than the previous case studies considered in OptiTrust. Working on this LLM code revealed that significant hurdles regarding the verbosity of annotations required by the framework.

My work made three primary contributions to address these challenges:

1. **The Autofocus Mechanism:** In manual verification of array-intensive code, annotations can represent up to 70% of the total file size. The explicit management of focus operations—temporarily restricting permissions to specific slices or cells constitutes a major bottleneck. To address the issue, we developed Autofocus, an algorithmic elaboration pass that automatically infers ghost operations. By defining a canonical ordering for iterated resources and automating index instantiation, the optitrust framework can now automatically find the right focus operations to syntactically match the resource required. On code struc-

tures similar to the LLM case study, Autofocus reduces the number of required annotations by 80% to 100%.

2. **Framework Extensions:** To support the complexity of the LLM inference code, we implemented several key extensions to OptiTrust. These include a normalization mechanism for nested matrix accesses, enhanced pattern matching for uninlining, and explicit support for matrix tiling. These features improved the robustness of the tool and its ability to handle real-world C code patterns.

3. **LLM Inference Case Study:** We demonstrated the maturity of the framework by applying several critical transformations on a complete execution pipeline of a Large Language Model (`llama2.c`). Moving beyond the small kernels used in previous studies, we tackled the full forward pass, which consists of approximately 300 lines of computational code. We successfully implemented a transformation script that applies token batching, loop tiling, and strip mining. This resulted in a 4.6× speedup over the baseline implementation, validating that OptiTrust can drive significant performance gains on complex, modern workloads.

6.2 Perspectives and Future Work

Towards Full Functional Correctness. In this case study, we focused on *shape-level* verification, which guarantees memory safety (no out-of-bounds accesses, no data races) but does not verify the full functional correctness of the output values and output states. A natural next step is to upgrade the specifications to target full functional correctness, as other researchers working on Optitrust has already done for two other case studies (matrix multiplication and image blur computation). The Autofocus mechanism was designed to be compatible with functional correctness assertions, but further work remains needed to generalize our implementation of autofocus to handle general assertions. In particular, the statement of the ghost focus and unfocus operations needs to be slightly refined to reflect a change of the memory contents upon write operations.

Advanced Optimizations. Although we achieved a significant speedup, our implementation still trails behind highly specialized libraries like `llama.cpp`. To bridge this gap, future work could focus on implementing algorithmic optimizations such as Flash Attention or integrating verified, hand-optimized assembly kernels (e.g., AVX2/AVX-512 BLAS routines) into the pipeline. The modular nature of Proof-Carrying Code allows these optimized kernels to be verified independently and swapped into the main pipeline.

Conclusion. This Master's thesis provided the opportunity to combine theoretical research with practical application. I actively contributed to the concrete development of OptiTrust (over 150 commits and 20k additions), implemented the *Continuous Integration* pipeline, and published the documentation, all while mastering diverse and novel topics such as Separation Logic and LLM inference.

In conclusion, the present work acts as a first step toward handling industrial code with OptiTrust. Time will tell whether the framework can derive LLM implementations matching the performance of industrial counterparts. By automating tedious verification steps with tools like Autofocus, the approach significantly decreases the number of required annotations. Such a reduction lowers the cost of employing the framework for high-performance, high-assurance software.

Bibliography

[1] Fabian Bannwart and Peter Müller. A program logic for bytecode. *Electronic Notes in Theoretical Computer Science*, 141(1):255–273, 2005. Proceedings of the First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2005).

[2] Gilles Barthe, Benjamin Grégoire, César Kunz, and Tamara Rezk. Certificate translation for optimizing compilers. *ACM Trans. Program. Lang. Syst.*, 31(5), jul 2009.

[3] Guillaume Bertholon. *Interactive compilation via trustworthy source-to-source transformations*. PhD thesis, 2025. Thèse de doctorat dirigée par Charguéraud, Arthur Informatique Strasbourg 2025.

[4] Guillaume Bertholon and Arthur Charguéraud. Bidirectional Translation between a C-like Language and an Imperative Lambda-calculus. In *36es Journées Francophones des Langages Applicatifs (JFLA 2025)*, Roiffé, France, January 2025.

[5] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 259–270, New York, NY, USA, 2005. Association for Computing Machinery.

[6] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6), December 2011.

[7] Arthur Charguéraud. *A Modern Eye on Separation Logic for Sequential Programs*. Habilitation à diriger des recherches, Université de Strasbourg, February 2023.

[8] Arthur Charguéraud, Begatim Bytyqi, Damien Rouhling, and Yann A Barsamian. OptiTrust: an Interactive Framework for Source-to-Source Transformations. working paper or preprint, September 2022.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*. USENIX Association, 2018.

[10] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1006–1036, 2012. The Programming Languages track at the 24th ACM Symposium on Applied Computing (SAC'09).

[11] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[12] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems - 12th International Conference, TACAS 2006. Held as Part of the Joint European Conf. on Theory and Practice of Software, ETAPS 2006*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 287–302. Springer Verlag, 2006. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2006. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006 ; Conference date: 25-03-2006 Through 02-04-2006.

[13] Jérôme Dohrau, Alexander J. Summers, Caterina Urban, Severin Münger, and Peter Müller. Permission inference for array programs. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 55–74. Springer, 2018.

[14] Georgi Gerganov. llama.cpp: Port of facebook's llama model in c/c++. https://github.com/ggml-org/llama.cpp, 2023.

[15] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[16] Andrej Karpathy. llama2.c: Inference llama 2 in one file of pure c. https://github.com/karpathy/llama2.c, 2023.

[17] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

[18] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding, unpublished, 2023.

[19] Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: automated verification of fine-grained concurrent programs in iris. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 809–824, New York, NY, USA, 2022. Association for Computing Machinery.

[20] Peter Müller and Martin Nordio. Proof-transforming compilation of programs with abrupt termination. In *Proceedings of the 2007 Conference on Specification and Verification of Component-Based Systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, SAVCBS '07, page 39–46, New York, NY, USA, 2007. Association for Computing Machinery.

[21] George Ciprian Necula. *Compiling with proofs*. PhD thesis, Carnegie Mellon University, 1998.

[22] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 1–19. Springer Verlag, 2001. Publisher Copyright: © Springer-Verlag Berlin Heidelberg 2001.; 15th International Workshop on Computer Science Logic, CSL 2001 and 10th Annual Conference of the European Association for Computer Science Logic, EACSL 2001 ; Conference date: 10-09-2001 Through 13-09-2001.

[23] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Conference on Programming Language Design and Implementation*, 2013.

[24] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Scotts Valley, CA, 2009.

Appendix

The following figure illustrates the data size metrics discussed in Chapter 5.

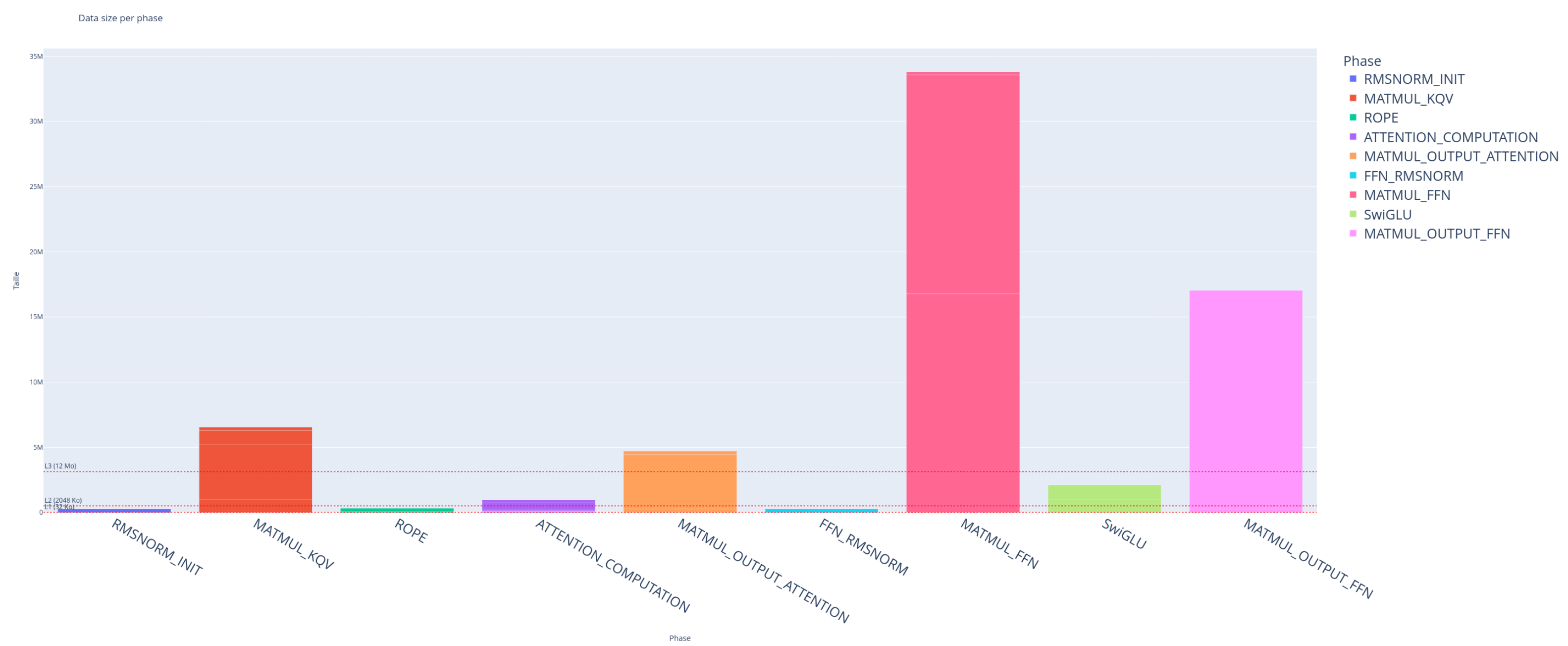

Figure 1: Data size for every weight matrix during the inference process.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. **In consultation with the supervisor**, one of the following two options must be selected:

☐ I hereby declare that I authored the work in question independently, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

☒ I hereby declare that I authored the work in question independently. In doing so I only used the authorised aids, which included suggestions from the supervisor regarding language and content and generative artificial intelligence technologies. The use of the latter and the respective source declarations proceeded in consultation with the supervisor.

Title of paper or thesis:

Automating Ghost Operations in Separation Logic for the Interactive Optimization of Array Programs: An LLM Case Study.

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
MOREL	ELIAN

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guidelines.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Strasbourg, 16/12/25	[signature]

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] For further information please consult the ETH Zurich websites, e.g. https://ethz.ch/en/the-eth-zurich/education/ai-in-education.html and https://library.ethz.ch/en/researching-and-publishing/scientific-writing-at-eth-zurich.html (subject to change).