RESEARCH-ARTICLE

Trinity: Three-Dimensional Tensor Program Optimization via Tile-level Equality Saturation

JAEHYEONG PARK, Korea Advanced Institute of Science and Technology, Daejeon, South Korea

YOUNGCHAN KIM, Korea Advanced Institute of Science and Technology, Daejeon, South Korea

HAECHAN AN, Korea Advanced Institute of Science and Technology, Daejeon, South Korea

GIEUN JEONG, Korea Advanced Institute of Science and Technology, Daejeon, South Korea

JEEHOON KANG

DONGSU HAN, Korea Advanced Institute of Science and Technology, Daejeon, South Korea

Published: 22 March 2026

Citation in BibTeX format

ASPLOS '26: 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems
March 22 - 26, 2026
PA, Pittsburgh, USA

Conference Sponsors:
SIGOPS
SIGPLAN
SIGARCH

Trinity: Three-Dimensional Tensor Program Optimization via Tile-level Equality Saturation

Jaehyeong Park
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
woguddlrj676@kaist.ac.kr

Youngchan Kim
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
chani227@kaist.ac.kr

Haechan An
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
haechan.an@kaist.ac.kr

Gieun Jeong
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
gieun.jeong@kaist.ac.kr

Jeehoon Kang*
FuriosaAI
Seoul, Republic of Korea
jeehoon.kang@furiosa.ai

Dongsu Han
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
dhan.ee@kaist.ac.kr

Abstract

Modern tensor program optimizers operate at two separate levels: graph-level optimizations (operator fusion, algebraic rewrites) and operator-level scheduling (tiling, parallelization). This separation prevents them from discovering cross-operator, tile-level optimizations that make hand-tuned kernels like FlashAttention effective.

We present Trinity, the first tensor program optimizer that achieves scalable joint optimization through tile-level equality saturation. Our key insight is that optimal performance requires simultaneously optimizing three interdependent dimensions — algebraic equivalence, memory I/O, and compute orchestration. To enable this, Trinity introduces a novel fine-grained IR that exposes all three axes as first-class, rewritable entities and applies equality saturation to perform scalable joint optimization. As a result, Trinity automatically discovers complex optimizations that require coordinated reasoning across all three dimensions. Across diverse Transformer variants, Trinity achieves up to 2.09× speedup over TensorRT and 2.35× over TorchInductor, both state-of-the-art production compilers.

CCS Concepts: • **Computing methodologies** → **Machine learning**.

Keywords: Tensor Program Optimization, Equality Saturation, Joint Optimization, Inference Latency

ACM Reference Format:
Jaehyeong Park, Youngchan Kim, Haechan An, Gieun Jeong, Jeehoon Kang, and Dongsu Han. 2026. Trinity: Three-Dimensional Tensor Program Optimization via Tile-level Equality Saturation. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 29 pages. https://doi.org/10.1145/3779212.3790240

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790240

1 Introduction

The rapid evolution of deep learning models and hardware accelerators has created a growing gap between what modern systems demand and what existing optimization approaches can deliver. Even within well-studied Transformer architectures, variants continue to proliferate—from different normalization schemes [21, 55] to architectural modifications [1, 37, 40] now standard in production LLMs [3, 7, 15, 44, 47, 58]. Simultaneously, GPU architectures are diversifying with different memory hierarchies, tensor core capabilities, and parallelism scales [31, 32].

Hand-optimizing kernels for *every* model-hardware combination is no longer feasible, and existing compilers produce suboptimal kernels while suffering from long compilation times. Therefore, efficient tensor program optimization remains critical: optimized kernels can reduce inference latency by orders of magnitude [17].

Current tensor program optimizers have evolved along two complementary but traditionally separate optimization paths: **Operator-level compilers** focus on the efficient implementation of individual tensor-level operators, including auto-scheduling systems like Ansor [62] and FlexTensor [65] that automatically search for optimal tensor programs. These compilers make decisions about how to tile tensors into chunks that fit in cache, how to order and fuse loops for better locality, how to parallelize dimensions across cores, and how to leverage specialized hardware units [5, 6, 16, 46]. **Graph-level optimizers** transform the high-level computational graph by fusing tensor-level operators to reduce memory traffic and kernel launches, reordering operations to

*Work done while at KAIST.

expose parallelism, and applying algebraic rewrites (e.g., distributivity, associativity) to minimize computation [5, 23, 38]. These systems reason about data flow between operations but generally do not modify their internal implementations.

While these two optimizations are complementary in principle, they have been applied mostly in isolation, relying on the interface of tensor-level operators. This separation prevents joint optimization: the optimal graph structure depends on how operators will be executed internally (tiling, parallelization, memory placement), while the optimal execution strategy depends on the graph structure (which operations are fused, how they are ordered). Current systems optimize each level independently, missing opportunities that require simultaneous reasoning about both graph-level transformations and tile-level execution.

Our key insight is that optimal performance requires holistic optimization across three interdependent dimensions: (1) **memory I/O**—managing data movement between memory hierarchies through tiling, caching, and reuse patterns, (2) **compute orchestration**—determining execution boundaries (which operations execute together) and parallelization strategies (how to distribute work across compute units), and (3) **algebraic equivalence**—cross-operator transformations that can restructure global computation patterns. These dimensions cannot be optimized in isolation—algebraic transformations change computation and memory access patterns, memory placement affects parallelization across allocated resources, and parallelization strategies influence which algebraic forms are profitable.

We present Trinity, the first tensor program optimizer that achieves scalable joint optimization through *tile-level equality saturation*. We introduce a novel IR that operates at the granularity where hardware actually executes—exposing memory operations (loads/stores), compute operations, and loop and sequential structures on tiles as first-class, rewritable entities. This fine-grained IR enables simultaneous reasoning about algebraic transformations, memory access patterns, and compute orchestration decisions within a unified framework. Moreover, we apply equality saturation [43] to this representation to address the search space explosion inherent in joint optimization, arising from the combinatorial interaction of the three dimensions. Trinity compactly represents the exponentially large space of equivalent programs and efficiently explores it without premature commitment.

This approach introduces two fundamental challenges:

Challenge 1: Designing a tile-level IR for equality saturation. Joint optimization requires an IR that exposes how tensors are tiled, when tiles are loaded and stored, and how loops iterate; however, these stateful operations conflict with the purely functional semantics traditionally assumed by equality saturation. A naïve application of equality saturation to the stateful IR leads to both exponential search space explosion and loss of correctness. We overcome this by designing the first tile-level IR with explicit memory operations and control flow (§ 3) and by developing a set of techniques that enable safe and efficient equality saturation, including expression propagation, sequence canonicalization, and semantic dependency checks (§ 4).

Challenge 2: Extracting optimal programs with context-dependent costs. In the joint optimization space, operation costs vary by execution context—loading a tile has minimal cost within the same kernel (on-chip) but significant cost across kernels (off-chip). Existing extraction algorithms assume fixed costs. We design a two-pass algorithm that first fixes loop structure to determine context, then extracts optimal kernels with accurate costs, efficiently handling e-graphs with more than 10^{17} equivalent programs (§ 5).

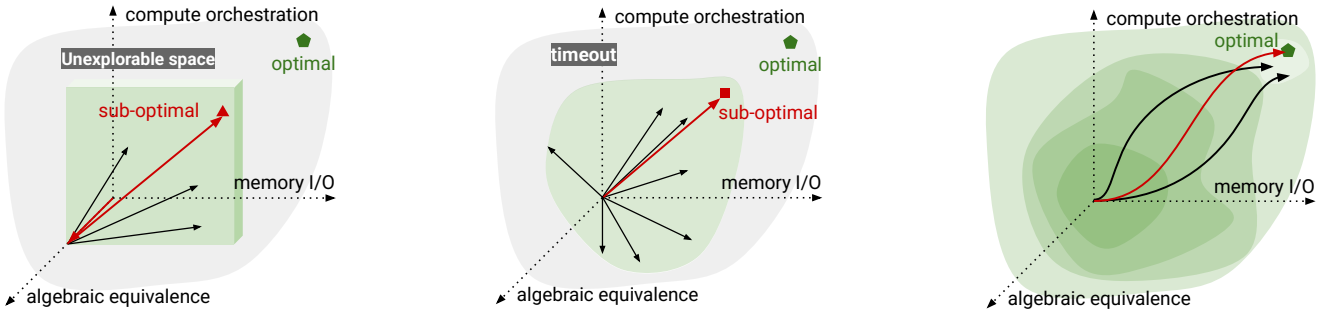
We evaluate Trinity on diverse dense Transformer variants [1, 21, 37, 40, 48, 55] across multiple hardware platforms. Trinity automatically generates kernels that achieve state-of-the-art performance: it achieves up to 2.09× speedup over TensorRT [33] (the state-of-the-art production compiler), and 3.07× speedup over prior unified approaches [53]. Notably, Trinity automatically finds optimization strategies that outperform manually engineered kernels by 1.35×. For example, in the vanilla transformer [48], Trinity not only rediscovers a FlashAttention [13, 14, 39]-style online softmax attention, but also goes further to find a fully fused attention that executes the QKV projection and the attention computation within a single kernel (§ 5.3).

2 Motivation & Approach

2.1 Limitation of Independent Optimization

Existing approaches optimize the three dimensions separately, as shown in Figure 1(a). Graph transformations based on algebraic equivalence [23, 49, 59] are performed on the computational graph of tensor-level operators. Then, operator-level compilers [10, 16, 46, 62] independently optimize each operator's implementation. Using tensor-level operators as the interface between these two stages leaves a large portion of the optimization space unexplored: at graph level, the implementation of each node is opaque, so optimizers can only move along the algebraic-equivalence dimension and must treat tile-level memory I/O and compute orchestration as fixed; at operator level, the tensor graph is already fixed, so compilers can only tune memory I/O and compute orchestration within a single operator and cannot apply algebraic rewrites that change the graph structure or move computation across operators.

However, discovering optimizations such as FlashAttention [13, 14, 39]—a highly efficient, manually engineered optimization—requires joint reasoning across algebraic transformations, memory access patterns, and parallelization strategies. These dimensions form a tightly coupled optimization



(a) Independent optimization. **(b)** Joint optimization. **(c)** Trinity's Joint optimization.

Figure 1. Comparison of three optimization approaches.

problem: algebraic transformations determine which memory access patterns are possible, memory placement constrains which parallelization strategies are viable, and parallelization choices in turn influence which algebraic forms are profitable.

Concretely, FlashAttention reduces $O(N^2)$ memory traffic by rewriting softmax into an online, tile-wise formulation. It organizes execution so that tiles of Q, K, and V, together with running statistics, remain in on-chip scratchpad memory, while partial results are accumulated within a fused kernel. This reduction from $O(N^2)$ to $O(N)$ memory traffic is possible only through the simultaneous selection of (1) a cross-operator algebraic reorganization (online softmax), (2) a tile-level memory schedule that preserves intermediates on-chip, and (3) an execution order and fusion strategy that respects loop-carried dependencies while maximizing parallelism. Further details are provided in § C.

2.2 The Right Abstraction Level

To enable joint optimization, we identify *tile granularity* as an ideal abstraction at which all three optimization axes become explicit and rewritable. A tile is a small block (e.g., 64×64 elements) of a tensor that fits in on-chip memory and is processed as a unit by modern accelerators such as GPUs and NPUs.

At this granularity, algebraic transformations appear as sequences of tile operations that can be reorganized. In addition, memory I/O becomes explicit load and store operations on tiles, enabling decisions about whether to keep tiles on-chip or spill them to off-chip memory. It is particularly crucial on modern accelerators, which must decompose tensors into tiles to exploit their high-bandwidth (tens of TB/s) but capacity-limited (only a few KB per compute unit) scratchpad. Compute orchestration similarly becomes a concrete set of decisions, including determining kernel boundaries (which tile operations fuse), selecting parallelization strategies (how loops map to thread blocks), and allocating hardware resources such as shared memory.

While several prior works [35, 41, 53] address parts of tile-level optimization, none exposes all three axes as first-class

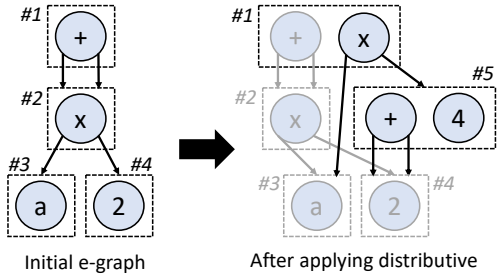


Figure 2. Example of e-graphs for expression $(a\times2)+(a\times2)$. Boxes represent e-classes and circles represent e-nodes.

rewritable constructs. Welder [41] and ASPEN [35] focus on tile-level scheduling and synchronization while assuming a fixed algebraic computation. Mirage [53] considers algebraic equivalence, but keeps memory I/O implicit in templates and, as discussed next, faces fundamental scalability limits.

Moreover, Trinity's tile granularity strikes a careful balance: it is fine enough to expose hardware-relevant execution details (unlike tensor-level abstractions), yet coarse enough to avoid combinatorial explosion. For example, while Triton [46] also exposes tile-level kernels and memory scopes, it includes many low-level syntactic knobs and tunable parameters (vector widths, pipeline stages). We explain our IR and rewriting rules in § 3.

2.3 Integration with Equality Saturation

We must systematically explore the vast space of equivalent tile-level programs because exhaustive search is intractable. Even for basic building blocks like vanilla Transformer, the search space contains over 10^{17} equivalent programs, making it impossible to exhaustively explore all possibilities. Prior work such as Mirage [53] attempts exhaustive exploration, but the search space is far too large to traverse within practical time limits, as shown in Figure 1(b). As a result, it fails to discover optimizations such as the fully fused attention that Trinity identifies (§ 5.3).

The use of equality saturation [43] enables scalable optimization without premature commitment (Figure 1(c)) by

Tensor Declaration `tensor`	
(**input**|**output** X)	Declare (input, weight, or output) tensor X placed in global memory.
(**variable** X)	Declare (intermediate) tensor X that can be placed both in global memory and scratchpad.
Indexing Expression `idx`	
(**index** idx1 idx2 idx3 ...)	Multi-dimensional index, equivalent to [idx1][idx2][idx3]...
(**tile** n)	Tile slice [n:n+tile_n], where tile_n is the stride of the loop variable n.
(**full_tile**)	Full slice [:].
(**const_tile** base interval)	Tile slice [base:base+interval] that is independent of any loop variable.
(**elem** n)	Element index [⌊n/tile_n⌋], where tile_n is the stride of the loop variable n
Memory Access Operation `op`	
(**load** tensor idx)	Load a tile value from tensor with the index idx.
(**store** tensor tval idx)	Store the tile value tval to the tile of the tensor with the index idx.
Primitive Compute Operation `op`	
(**+**|**-**|*****|**/** tval1 tval2)	Element-wise operations between two tile values tval1 and tval2.
(**exp**|**sqrt**|**sigmoid** tval)	Element-wise operations on a tile value tval.
(**rsum** tval k)	Reduction operations on a tile value tval, along the axis k.
(**concat**|**permute**|**(un)squeeze**|**bcast** ...)	Reshape operations on tile values. Operands are omitted for simplicity.
(**matmul** tval1 tval2)	Matrix multiplication operation between two tile values tval1 and tval2.
Control Flow Operation `op`	
(**seq** op1 op2)	Execute op1 and op2 sequentially.
(**loop** start end tile_n n op)	Iterate the body op with n ranging [start,end), incremented by tile_n at each iteration.

Table 1. Constructs of Trinity IR.

compactly representing many equivalent programs in a single *e-graph*. An e-graph groups semantically equivalent expressions into *e-classes*, each containing multiple *e-nodes*. Figure 2 illustrates an example of e-graphs and how rewrite rules are applied. The initial e-graph (left) represents a single expression, $(a \times 2) + (a \times 2)$. When a rewrite rule matches, the e-graph does not discard the original expression; instead, it adds new e-nodes and merges e-classes, monotonically accumulating equivalences. The e-graph on the right shows the result of applying the rewrite rules, such as distributive transformation and constant folding. By adding a $\times$ e-node to e-class #1 and introducing e-class #5 containing the e-node $(2+2)$, the e-graph simultaneously represents both the original expression and $a \times (2+2)$. Furthermore, by adding the e-node "4" to e-class #5, the e-graph also captures the expression $a \times 4$.

Optimization proceeds in two phases. In the *saturation* phase, rewrite rules are repeatedly applied to saturate the e-graph, accumulating equivalent expressions until no new equivalences appear. The *extraction* phase then selects the best program from this saturated e-graph using a cost model, by systematically choosing the optimal e-node from each e-class. Because all variants are maintained and rewritten in parallel, equality saturation can explore a vast search space without enumerating each candidate program.

However, applying equality saturation to tile-level IR is challenging because its sequential, stateful nature violates the pure functional assumptions of traditional equality saturation, and because the cost of each operation depends heavily on its execution context (e.g., whether a load stays on-chip or crosses a kernel boundary). We address these challenges in § 4 and § 5.

3 Trinity IR

Trinity IR makes all three optimization dimensions—memory I/O, compute orchestration, and algebraic transformation—explicitly rewritable at tile granularity. This design enables two critical capabilities: representing highly- optimized kernels like FlashAttention directly in the IR (§ 3.1), and systematically discovering such optimizations from naïve implementations through rewrite rules (§ 3.2).

3.1 IR Definition

Trinity IR treats tiles as first-class citizens. The rest of this section describes each construct of the IR, as summarized in Table 1.

1) To expose memory I/O, the IR provides the following operators for data movement between memory hierarchies. *Explicit memory access:* The `load` and `store` operators make memory access patterns visible and optimizable as first-class IR citizens. This enables discovering optimizations like reordering independent memory operations for locality or eliminating redundant loads.

```
// C = A @ B; E = C + D
(seq
  // loop for matrix multiplication. C = A @ B
  (loop 0 M tile_m m (loop 0 N tile_n n (loop 0 K tile_k k
    (store (variable C)
      (+
        (load (variable C) (index (tile m) (tile n)))
        (matmul
          (load (input A) (index (tile m) (tile k)))
          (load (input B) (index (tile k) (tile n)))))
      (index (tile m) (tile n)))
  )))
  // loop for element-wise addition. E = C + D
  (loop 0 M tile_m m (loop 0 N tile_n n
    (store (output E)
    (+
      (load (variable C) (index (tile m) (tile n)))
      (load (input D) (index (tile m) (tile n))))
    (index (tile m) (tile n)))
  ))
)
```

Figure 3. Trinity IR example of consecutive matrix multiplication and element-wise addition.

Tile placement declaration: Trinity explicitly tracks memory placement through three operators: `input`, `output` (tiles from/to off-chip global memory), and `variable` (intermediate tiles that may reside either on-chip or off-chip memory). This distinction is crucial for discovering complex memory reuse patterns, such as keeping frequently accessed tiles in on-chip memory across loop iterations.

2) To manipulate compute orchestration, the IR provides rewritable control flow operators, namely *sequences* and *loops*. The `seq` operator explicitly represents execution order, while the `loop` represents iterations over tiles. These operators cover regular tiling patterns while keeping the search space tractable. Representing control flows as rewritable terms enables discovery of novel execution strategies like interleaving operations from multiple tensor-level operators into single kernels.

3) To enable algebraic transformation, Trinity IR treats tiles as small tensors and reuses standard compute operations as its compute primitives. This allows algebraic optimizations like distributivity and associativity to compose naturally with memory and loop transformations within the unified e-graph framework.

Example. Figure 3 shows an example of Trinity IR that represents a tile-level computational flow of consecutive matrix multiplication and element-wise addition. In the case of matrix multiplication between two 2D tensors, the computation is tiled along the output axes (`M, N`) and the reduction axis (`K`). For element-wise addition, the tiling simply partitions the output tensor along spatial dimensions, allowing parallel computation of each tile.

3.2 Rewriting Rules

Trinity defines rewrite rules that preserve program semantics while exposing optimization opportunities. These rules fall into two main categories: loop and algebraic transformations. All rewriting rules are detailed in § A.

Loop transformation rules. Trinity employs transformation rules to reorganize loop structures for tiling (§ A.1). It ensures that rules are applied only when they preserve semantics by carefully checking memory access.

Loop fusion, *fission*, and *loop-invariant code motion* are classical loop transformations [29]. Loop fusion allows two loop bodies to be executed within a single kernel, reducing kernel launch overhead. It also improves memory locality by reusing intermediate tiles. In contrast, loop fission enables different hardware execution strategies to be applied to the split loop bodies. Loop-invariant code motion moves the computation that produces identical results across iterations out of the loop, eliminating redundant computation.

Loop insertion is a rewrite rule that inserts a loop-invariant code into the loop body, serving as the opposite of loop-invariant code motion. Although this causes redundant computation, it enables the inserted operation and the original loop body to be executed within a single kernel. Traditional loop optimization avoids loop insertion as it increases computation, but it proves to be highly effective in inference workloads where memory I/O and kernel launch overhead often dominate computation costs.

Algebraic factoring in loop bodies extracts common operands from within loops. This transformation reduces operations inside the loop and, crucially, can remove loop-carried dependencies that previously blocked fusion opportunities.

Algebraic transformation rules. Trinity uses 31 rewrite rules for performing algebraic transformations of tile operations (§ A.2). We adopt rules from prior works [23, 59] on tensor-level graph rewriting, which have been either manually defined or automatically generated based on algebraic properties. Since tiles are small tensors, these rules remain equally effective at tile granularity.

4 Saturation Phase

Building on this IR formulation, Trinity applies equality saturation for joint optimization. In the saturation phase, Trinity initializes an e-graph and repeatedly applies rewrite rules until no new equivalences are discovered. To handle the challenges of applying equality saturation to a stateful IR, Trinity introduces three key techniques: expression propagation to enable algebraic reasoning with explicit memory I/O (§ 4.1); canonicalization of right-associative sequencing to control e-graph growth (§ 4.2); and semantic dependence checks to ensure correctness (§ 4.3).

4.1 Algebraic Reasoning with Explicit Memory I/O

In our stateful IR, memory accesses are expressed explicitly through `load` and `store` operations inside `seq`. This makes it difficult to apply algebraic transformation rules directly, because the algebraic structure of a value is fragmented

across multiple statements. For example, in the expression `(seq (store A (* 7 3) idx) (store B (/ (load A idx) 7) idx))`, the division by 7 is syntactically separated from the multiplication by 7 by an intermediate store to `A` and a later load from `A`. From the e-graph's perspective, the pattern `(/ (* 7 3) 7)` never appears contiguously in the AST, so algebraic rules such as cancellation or constant folding cannot match. As long as algebraic rules operate only on purely functional subtrees, explicit load/store boundaries act as barriers that block pattern matching.

Expression Propagation. To overcome this, Trinity performs *expression propagation* across sequential memory operations. Whenever a tile is stored to memory, Trinity records the symbolic expression being written, and subsequent loads from the same tile are rewritten to refer directly to that expression. In the running example, propagation rewrites the expression `(/ (load A idx) 7)` to `(/ (* 7 3) 7)`, enabling algebraic simplification. Trinity applies this propagation at the end of every rewrite iteration, updating the e-graph so that algebraic transformation rules always see the most precise expression available, despite the presence of explicit loads and stores. This allows us to keep memory operations explicit for correctness and cost modeling, while still enabling aggressive algebraic rewriting on top of a stateful IR.

4.2 Sequence Canonicalization

Explosion due to the sequence operator. Another critical performance challenge in saturation stems from the inherently sequential nature of tile-level IRs, which must encode ordered memory accesses and loop iterations. Applying equality saturation to such sequential constructs leads to exponential inflation of the e-graph. For example, a rule such as `(seq a b)` → `(seq b a)` is only applicable when its two operands `a` and `b` appear as direct children of a `seq` node. Given an expression like `(seq (seq op1 op2) (seq op3 op4))`, commuting `op2` and `op3` first requires rewriting the expression via associativity into a form such as `(seq (seq op1 (seq op2 op3)) op4)`. Further commutations, such as between `op2` and `op4`, require additional parenthesizations. Although these variants are semantically equivalent, all must be explicitly represented in the e-graph for pattern matching, leading to redundant exponential growth. In principle, variadic sequence patterns could alleviate this, but integrating general variadic matching into equality saturation frameworks is non-trivial and is not directly supported in common e-graph toolchains.

Leveraging canonical sequences. We prevent this explosion by maintaining all sequences in a canonical right-associative form, e.g., `(seq op (seq op (seq op ...)))`. Whenever a rewrite produces a non-canonical sequence, Trinity immediately normalizes it into this canonical form. Under this invariant, sequence transformations require only two rewrite patterns—the basic `(seq a b)` → `(seq b a)` and its nested variant `(seq a (seq b tail))` → `(seq b (seq a tail))`—without any explicit sequence associativity rules. By eliminating the redundant associative variants, Trinity avoids exponential e-graph growth while preserving full expressiveness for sequence transformations.

4.3 Ensuring Correctness on Stateful IR

Naively applying rewrite rules to stateful IRs can easily break correctness. For example, loop fusion between `(loop n ... body`$_1$`)` and `(loop n ... body`$_2$`)` is valid only when no write in `body`$_1$ is read or overwritten by `body`$_2$ in a later loop iteration. Traditional equality saturation systems rely primarily on syntactic pattern matching with simple type checks, which is insufficient for reasoning about such dependencies.

To guarantee correctness, Trinity augments syntax-directed rewriting with *semantic* checks implemented using egg [51]'s e-class analysis framework. For each e-class, Trinity maintains summary information about its memory behavior—such as sets of read and write regions, the tensors they may alias, and the loop variables on which the accesses depend—along with shape metadata. These summaries are updated and propagated throughout the e-graph as new equivalences are added. Each rewrite rule is then guarded by a semantic predicate that inspects these analyses before the rule fires. For instance, loop fusion and loop insertion perform dependency analysis to ensure that reordering does not introduce read–after–write or write–after–write hazards across iterations; if a potential conflict is detected, the rule is not applied.

Pure algebraic rewrites over tile values (e.g., distributivity) are treated as semantics-preserving up to standard floating-point rounding differences, consistent with prior systems [14, 53, 67]. In practice, we find that this semantics-aware use of e-class analysis enables loop and memory transformations while preserving correctness for all optimized kernels.

5 Extraction Phase

After applying a set of rewrite rules to saturate the e-graph, we should select the best kernel from all possible programs. Trinity extracts top candidates via a two-pass extraction algorithm (§ 5.1), then generates and profiles executable kernels to select the best performer (§ 5.2).

5.1 Extracting High-Performance Candidates

Extracting a single expression from the saturated e-graph amounts to choosing exactly one e-node from every e-class: starting at the root e-class, we pick one e-node, then recursively pick one e-node from each of its child e-classes, and so on. Prior equality saturation systems [18, 20, 51, 59] formulate extraction as a global optimization problem under a *fixed local cost* assumption: they assign each e-node a fixed cost, then search for a selection whose sum of chosen e-node

costs is minimized. This objective is typically solved either by greedy selection or by ILP solving.

However, this strategy fails in tile-level representation, where the cost of e-node depends on the hardware execution context and, therefore, cannot be fixed. For example, even the same arithmetic e-node `(+ a b)` can have very different cost depending on whether the surrounding loop must execute sequentially or can be parallelized. If the loop must execute sequentially, `(+ a b)` is executed once per iteration, so its cost under a *FLOPs per computational unit* model should scale with the iteration count. In contrast, if the loop can be mapped to a parallel loop and distributed across computational units, the per-unit cost of the same `(+ a b)` e-node collapses to roughly "one add."

Two-pass extraction algorithm. To overcome this challenge, Trinity introduces a two-pass extraction algorithm that separately extracts the loop structure and loop body. Our key observations are threefold. First, once the *loop structure* is fixed, the hardware execution context (e.g., kernel boundaries and parallelization) also gets fixed, which enables us to leverage existing extraction algorithms [18, 20, 51, 59] with fixed cost models. Second, the space of loop structures in tile-level e-graphs is much smaller, making it feasible to explore loop structure candidates with lightweight pruning. Third, the loop structure determines the number of kernel launches, a dominant performance factor due to kernel launch overhead and inter-kernel memory traffic, making kernel count a natural objective for the first pass.

Building on these observations, Algorithm 1 yields k complete tile-level expressions. Pass 1 identifies the top-k *semi-expressions* with minimal kernel counts, prioritizing reduced memory I/O and kernel launch overhead. Pass 2 completes each semi-expression by selecting loop bodies with minimal FLOPs per computational unit, optimizing computational efficiency within the now-fixed execution context.

Pass 1: Extracting loop structures. The `ExtractLS` function (lines 7-20) recursively extracts loop structures using the number of kernels, or kernel count, as the coarse-grained, context-insensitive cost. We use kernel count as the proxy for the cost since fewer kernels typically lead to less kernel launch overhead and memory I/O. The number of kernels can be determined by counting the number of outermost loops.

Trinity chooses top-k semi-expressions with the lowest kernel counts, then proceeds to Pass 2. Here, a *semi-expression* is a partially extracted program obtained by selecting e-nodes for only a subset of e-classes—specifically, those containing loop-structure-defining operators (e.g., `seq` and `loop`)—while leaving the remaining e-classes unresolved.

Pass 2: Extracting loop bodies. `ExtractLoopBody` (lines 21-22) completes each of the k selected semi-expressions by extracting optimal loop bodies. With the loop structure fixed, the execution behavior is determined, allowing accurate and *fixed* cost assignment—here based on FLOPs per computational unit. `ExtractLoopBody` uses a greedy algorithm to select loop bodies with minimum cost. This yields an accurate estimate because the dominant context-dependent factor (kernel boundaries and parallel-vs-sequential structure) has already been resolved in Pass 1, while memory I/O considerations are largely captured by the kernel-count filtering.

The two-pass extraction algorithm is designed to select high-performance candidates within a tractable amount of time. Although it may miss some candidates, we empirically observed that it consistently selects performant candidates using dominant performance indicators as costs.

Algorithm 1 Two-Pass Extraction Algorithm

```
   Input: e-graph E, top_k
1: function TwoPassExtract(E, top_k)
2:     max_kernel ← 0, S ← [ ]
3:     while |S| < top_k do
4:         S ← S ∪ ExtractLS(E, max_kernel, E.root)
5:         max_kernel ← max_kernel + 1
6:     return [ ExtractLoopBody(E, s) | s ∈ S ]
   // Extracts loop structure candidates
7: function ExtractLS(E, max_kernel, id)
8:     if max_kernel < 0 then return [ ]
9:     res ← [ ]
10:    for all n ∈ E[id].nodes do
11:        if n is Seq() or Loop() then
12:            k ← 1 if n is level-0 loop, else 0
13:            k_r ← max_kernel − k
14:            C ← ⋃_{cid∈n.children()} ExtractLS(E, k_r, cid)
15:            for all c ∈ CartesianProduct(C) do
16:                if c.k ≤ max_kernel then
17:                    res.append(Format(n, c), c.k)
18:        else
19:            res.append(("body", 0))
20:    return res
21: function ExtractLoopBody(E, semi_expression)
22:    return Greedy(E, semi_expression, min_FLOPs)
```

5.2 Kernel Generation from IR

Trinity IR's explicit encoding of memory access, loops, and sequences enables direct and predictable generation of efficient hardware kernels. The translation from IR to kernel code follows systematic rules that preserve the optimizations discovered during equality saturation.

Kernel boundary decision. Trinity first analyzes each `loop` operator to determine parallel execution potential. Using our sequence-aware representation, it identifies loop-carried dependencies—loops with dependencies execute sequentially on single compute units, while independent iterations map to parallel units (e.g., GPU thread blocks).

Modern accelerators require fixed parallelism configurations per kernel launch. Executing an entire graph as one kernel would force a single configuration, causing resource under-utilization when different parts of the computation have different parallelism patterns. Trinity therefore partitions the graph at outermost parallel loop boundaries. Each parallel loop nest becomes a separate kernel with its own optimized configuration, while sequential operations between parallel loops are grouped together. This strategy ensures each kernel fully utilizes available parallelism without over-subscription or underutilization.

Memory placement optimization. With kernel boundaries established, Trinity determines optimal memory placement for each tile. The strategy prioritizes on-chip memory to minimize expensive off-chip accesses:

- *Load placement*: A tile is loaded from off-chip memory only (1) when the tile's store occurs in a different loop from the current load, or (2) when accessing an `input` tensor. Otherwise, the tile is reused from on-chip memory.
- *Store placement*: A tile is stored to off-chip memory only (1) when it will be loaded in a different loop, or (2) when it is an `output` tensor. All other intermediate tiles remain in on-chip memory.

This systematic approach ensures that tiles stay in the fastest available memory whenever possible, automatically achieving the kind of memory optimization that makes hand-tuned kernels like FlashAttention effective.

Trinity explicitly controls tile-level operations, memory accesses, and parallel execution through Triton [46] v3.4.0's interface, while benefiting from Triton's optimizations for tensor core utilization and register allocation. In particular, Trinity keeps tile sizes symbolic during equality saturation and defers concrete tile selection to a profiling stage that leverages Triton's block-size auto-tuning. This design avoids e-graph blowup while still capturing the dominant performance factors. The systematic translation ensures that the optimizations discovered during equality saturation are faithfully preserved in the generated hardware kernels.

Selecting the best through profiling. Finally, Trinity selects the optimal kernel by executing all k extracted candidates on actual hardware. Since the relative impact of kernel count versus computational efficiency varies with hardware characteristics (memory bandwidth, compute throughput, cache sizes), Trinity empirically determines the optimal kernel through hardware profiling.

```
(loop 0 4096 tile_n n (loop 0 4096 tile_k k        # QKV Projection
  (Q1,K1,V1[:,tile n] +=
    X[:,tile k] * WQ,WK,WV[tile k,n])))
(loop 0 4096 128 n                                 # Reshape, Permute
  (Q,K,V[elem n,:,:] = transpose(expand_dims(Q1,K1,V1[:,tile n],1),0,1))

(loop 0 32 1 h (loop 0 1024 tile_p p               # Attention
  (logit[elem h,:,tile p] = exp(Q[elem h,:,:] * K_cache[elem h,tile p,:]^T))))
(loop 0 32 1 h (loop 0 1024 tile_p p
  (accm[elem h,:] += reduce_sum(logit[elem h,:,tile p]))))
(loop 0 32 1 h (loop 0 1024 tile_p p
  (O[elem h,:,:] +=
    (logit[elem h,:,tile p] / accm[elem h,:]) * V_cache[elem h,tile p,:])))
```

(a) Initial program in Trinity IR.

```
# QKV Projection, reshape, permute ...

(loop 0 32 1 h (loop 0 1024 tile_p p               # Attention
  (logit[elem h,:,tile p] = exp(Q[elem h,:,:] * K_cache[elem h,tile p,:]^T))
  (accm[elem h,:] += reduce_sum(logit[elem h,:,tile p]))))
(loop 0 32 1 h (loop 0 1024 tile_p p
  (O[elem h,:,:] +=
    (logit[elem h,:,tile p] * V_cache[elem h,tile p,:]) / accm[elem h,:])))
```

(b) Apply loop fusion and distributivity.

```
# QKV Projection, reshape, permute ...

(loop 0 32 1 h (loop 0 1024 tile_p p               # Attention
  (logit[elem h,:,tile p] = exp(Q[elem h,:,:] * K_cache[elem h,tile p,:]^T))
  (accm[elem h,:] += reduce_sum(logit[elem h,:,tile p]))))
(loop 0 32 1 h
  (loop 0 1024 tile_p p
    (O[elem h,:,:] += logit[elem h,:,tile p] * V_cache[elem h,tile p,:]))
  (O[elem h,:,:] /= accm[elem h,:]))
```

(c) Apply algebraic factoring in loop body.

```
# QKV Projection, reshape, permute ...

(loop 0 32 1 h                                     # Attention
  (loop 0 1024 tile_p p
    (logit[elem h,:,tile p] = exp(Q[elem h,:,:] * K_cache[elem h,tile p,:]^T))
    (accm[elem h,:] += reduce_sum(logit[elem h,:,tile p]))
    (O[elem h,:,:] += logit[elem h,:,tile p] * V_cache[elem h,tile p,:]))
  (O[elem h,:,:] /= accm[elem h,:]))
```

(d) Apply further loop fusion.

```
(loop 0 4096 128 n
  (loop 0 4096 tile_k k      # QKV Projection, reshape, transpose
    (Q1,K1,V1[:,tile n] += X[:,tile k] * WQ,WK,WV[tile k,n])
  (Q,K,V[elem n,:,:] = transpose(expand_dims(Q1,K1,V1[:,tile n],1),0,1)))

# Attention ...
```

(e) Apply loop fusion with fixed tile size.

```
(loop 0 4096 128 n
  (loop 0 4096 tile_k k      # QKV Projection, reshape, transpose
    (Q1,K1,V1[:,tile n] += X[:,tile k] * WQ,WK,WV[tile k,n])
  (Q,K,V[elem n,:,:] = transpose(expand_dims(Q1,K1,V1[:,tile n],1),0,1))

  (loop 0 1024 tile_p p      # Attention
    (logit[elem n,:,tile p] = exp(Q[elem n,:,:] * K_cache[elem n,tile p,:]^T))
    (accm[elem n,:] += reduce_sum(logit[elem n,:,tile p]))
    (O[elem n,:,:] += logit[elem n,:,tile p] * V_cache[elem n,tile p,:]))
  (O[elem n,:,:] /= accm[elem n,:]))
```

(f) Apply loop fusion with iteration-space reindexing.

Figure 4. Optimization procedure of Transformer block discovered by Trinity. Loop bodies use syntactic sugar for readability.

5.3 Case Study: Fully Fused Attention

We demonstrate Trinity's capability through a case study on the Vanilla [48] transformer architecture. Figure 4 shows how it uncovers an optimization that existing systems fail to discover. The initial program (Figure 4(a))—QKV projection with reshaping into multiple heads (blue box) followed by the multi-head attention (green box)—corresponds to the tensor program for the decoding step of a Vanilla transformer

block. Through systematic application of rewrite rules, Trinity automatically discovers a significantly more efficient implementation:
(b) Loop fusion and distributivity: Within the attention operation (green box), Trinity first fuses the *logit* computation and the subsequent *reduce_sum* into a single loop (or equivalently, a single kernel). This reduces memory traffic and kernel launch overhead. However, the following computation of the output tensor O remains impossible to fuse due to a loop-carried dependency on the accumulator *accm*.

To break this dependency, Trinity applies the distributive law to move the division term involving *accm* outside the matrix multiplication. This transformation follows the same floating-point assumptions widely adopted in prior work [14, 53, 67]. Although the resulting computation is not bit-identical to the original formulation, it yields numerically stable outputs within reasonable error bounds.
(c) Algebraic factoring: Subsequently, Trinity applies the algebraic factoring in loop body rule, which hoists the division fully outside the inner p-loop. Once *accm* is no longer needed within the p-loop, the loop-carried dependency disappears, unlocking further fusion opportunities.
(d) Further loop fusion: With the dependency resolved, Trinity applies loop fusion to execute the entire attention computation within a single h-loop (i.e., a single kernel). This yields the same algorithm as FlashAttention [13, 14, 39], demonstrating that Trinity can automatically rediscover the FlashAttention optimization purely through equality saturation driven rewrites.
(e) Loop fusion: Going further, Trinity extends optimization beyond attention to include the preceding QKV projection and head-reshaping operations (blue box). It fuses the QKV projection and reshaping into a single loop. Although the tile size of QKV projection is fixed at 128—limiting tile size autotuning—this structured tiling enables subsequent fusion with attention, providing overall performance gains.
(f) Final fusion: Finally, Trinity applies the loop fusion with iteration-space reindexing rule. Although the loops `(loop 0 4096 128 n)` and `(loop 0 32 1 h)` differ syntactically, they share the same iteration count. Trinity detects this equivalence and rewrites all `(elem h)` expressions in h-loop as `(elem n)`, aligning their iteration spaces. Once their loop variables are unified, the loops become fusible, enabling a final fusion step.

Fully fused attention. To the best of our knowledge, we are the first to report this optimization. It executes all operations—from the initial processing of the input token to the full attention computation—within a single kernel. Whereas existing approaches first compute Q, K, V for all heads and then reshape the results for per-head processing, Trinity computes Q, K, V on a per-head basis and streams them directly into the attention computation within the same kernel. This design (1) eliminates the synchronization barrier required to wait for all heads' Q, K, V tensors, (2) reduces kernel-launch overheads, and (3) avoids writing intermediate tensors (such as Q) to off-chip memory. Consequently, Trinity achieves up to 1.35× speedup on H100 compared to the manually optimized implementation [60].

6 Evaluation

We evaluate Trinity to answer the following questions:

- Does Trinity discover faster hardware kernels than existing tensor program optimizers across various models and hardware? (§ 6.2)
- Does Trinity find the optimal kernel tailored to the characteristics of the given hardware? (§ 6.3)
- Does Trinity explore a large search space with reduced compile time by leveraging equality saturation? (§ 6.4)

6.1 Experimental Setup

Model architectures. We evaluate Trinity on transformer variants widely adopted in modern LLMs [3, 7, 15, 44, 47, 58]: Vanilla transformer [48], Pre-Norm [55], QK-Norm [21], RoCo [37], KeyFormer [1], and SwiGLU FFN [40]. We focus on transformers because they dominate current AI workloads and incorporate both dense layers and attention mechanisms. Nevertheless, Trinity is not limited to transformers: users can initialize tensor programs by converting tensor operators into Trinity IR through a deterministic procedure, enabling Trinity to support other model families, such as state-space models and diffusion models.

In a standard transformer block, input embeddings are multiplied by weight matrices W_{QKV} to produce query, key, and value representations, which are reshaped for multi-head attention computation. The evaluated architectures extend this base structure with additional operations; e.g., Pre-Norm adds RMS normalization before QKV projection, while RoCo includes auxiliary computations for KV cache management.

For attention variants [1, 21, 37, 48, 55], we optimize the entire attention block, from input embedding to attention output, to demonstrate Trinity's ability to discover joint optimizations in large input programs. For SwiGLU FFN [40], we optimize the entire feedforward network that succeeds the attention block. We evaluate two model configurations—LLaMA3 8B [15] (hidden dimension 4096, 32 heads) and Falcon 7B [3] (hidden dimension 4544, 71 heads)—in a speculative decoding [27] scenario where a 16-token block is verified on top of a 1008-token prefix, yielding a total KV-cache length of 1024 tokens.

Correctness. For all architectures and configurations, we verified that Trinity's optimized kernels produce numerically equivalent outputs to the original programs, modulo floating-point rounding differences from operation reordering.

Baselines. We compare Trinity with widely used production-level tensor program optimizers: TorchInductor [5] v2.8.0

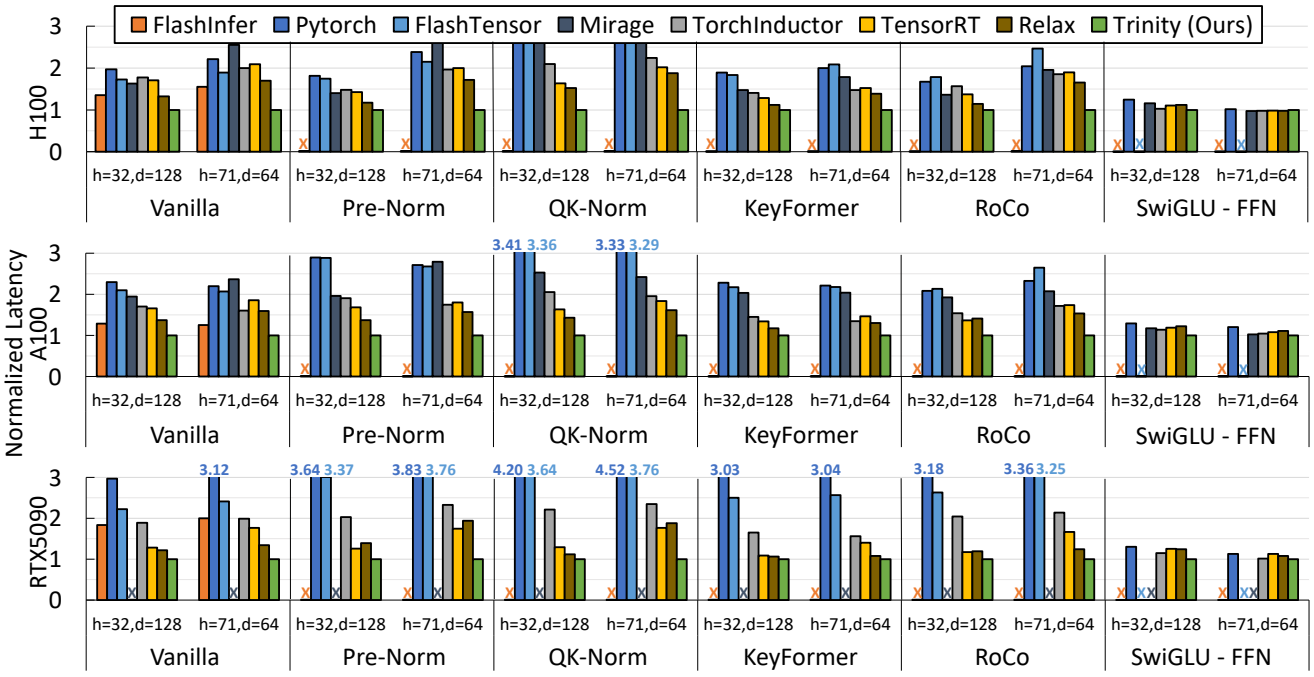


Figure 5. Normalized inference latency across various benchmarks and GPUs.

and TensorRT [33] v10.10.0.31. We also evaluate against FlashInfer [60] v0.5.3, a state-of-the-art hand-tuned attention kernel library for LLM inference. Since FlashInfer is dedicated solely to the core attention computation, QKV projection and reshaping are implemented using PyTorch. Additionally, we benchmark FlashTensor [67], the state-of-the-art in tensor-level graph rewriting, to emphasize the benefits of Trinity 's joint optimization. We further compare against Relax [26], a recent compiler supporting dynamic shapes through pre-defined fusion and layout heuristics. Finally, we evaluate Mirage [53], which performs exhaustive joint optimization, to highlight the advantage of large search space exploration via equality saturation. For Mirage and FlashTensor, which require extensive manual tuning, we followed their recommended configurations with careful hand-tuning (see § B).

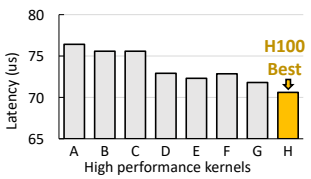
We conduct experiments on two server-grade and two consumer-grade GPUs. For server-grade GPUs, we use GCP cloud instances: an a3-highgpu-1g with NVIDIA H100 80GB (26 vCPUs, 234GB RAM) and an a2-highgpu-1g with NVIDIA A100 40GB (12 vCPUs, 85GB RAM). For consumer-grade GPUs, we use NVIDIA RTX 4090 and RTX 5090, both paired with Intel Xeon Silver 4210R CPUs and 208GB DRAM. We extract up to 512 candidates per benchmark and during the profiling phase, we use eight GPUs to parallelize the process. Similarly, we use eight threads to parallelize Mirage.

6.2 Inference Latency

Figure 5 presents the normalized latency of various model architectures. Compared to TensorRT, which is a state-of-the-art production compiler, Trinity achieves 1.71× speedup for Vanilla transformer, 1.43× for Pre-Norm, 1.63× for QK-Norm, 1.29× for KeyFormer, 1.37× for RoCo, and 1.10× for SwiGLU FFN on H100 (for LLaMA3 8B config). Furthermore, Trinity achieves up to 3.07× speedup on the H100 over Mirage, which performs joint optimization but is restricted in large search space handling because of its exhaustive search approach. The highest-performance kernel and its IR for each case are provided in § E.

FlashInfer, despite being a state-of-the-art hand-tuned attention kernel, cannot handle any architecture beyond Vanilla transformer. This limitation of manual approaches highlights the need for automatic optimization—hand-tuned kernels cannot cover the growing diversity of model architectures. Mirage's implementation does not yet support the Blackwell architecture (RTX 5090).

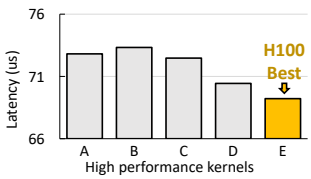
In the Vanilla transformer that FlashInfer does support, Trinity outperforms it by 1.35× on H100 by discovering the novel fully fused attention described in § 5.3. Trinity not only automatically discovers the same algorithmic structure as FlashAttention, but also holistically optimizes the preceding QKV projection and reshaping operations, enabling all components to be executed within a single kernel. Trinity even outperforms FlashInfer's FlashAttention3 implementation, despite relying on Triton and therefore not leveraging FA3's specialized hardware features. Such results highlight the strength of Trinity's equality saturation approach, which



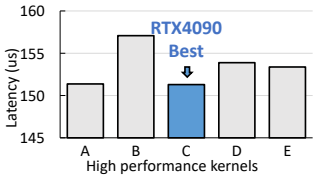
(a) H100

(b) RTX4090

Figure 6. Latency of high performance kernels for KeyFormer (LLaMA3 8B config) discovered by Trinity.



(a) H100



(b) RTX4090

Figure 7. Latency of high performance kernels for RoCo (LLaMA3 8B config) discovered by Trinity.

can uncover complex cross-operator transformations that existing systems—including Mirage, the state-of-the-art joint optimizer—fail to identify due to partitioning constraints or limited search scopes.

In the case of other attention variants, Trinity integrates extra operations to achieve single-kernel execution similar to FlashAttention. For the Pre-Norm architecture, Trinity achieves a dramatic 1.40× speedup over Mirage on H100 by further fusing RMS normalization into the fully fused attention. Conventional implementations require at least four kernels—one for computing the *RMS* statistic via a reduction, one for normalizing an input token embedding X via elementwise scaling, one for performing the QKV projection, and additional kernels for the remaining attention computation—each incurring separate kernel-launch and memory-transfer overheads. In contrast, Trinity uses equality saturation driven rewrites to restructure these operations so they can execute in a single pass: loop insertion brings *RMS* computation into the projection loop, distributivity moves scaling outside the matrix multiplication, and algebraic factoring eliminates loop-carried dependencies that prevent fusion. This sequence of transformations eliminates the need to materialize intermediate tensors (i.e., normalized X) in off-chip memory, thereby reducing memory traffic and removing synchronization barriers.

In the case of KeyFormer and RoCo, the attention computation is not limited to $\text{softmax}(QK) \times V$; they additionally compute values such as $\text{softmax}(QK + \text{noise})$ for subsequent KV cache eviction, or the reduction of $\text{softmax}(QK)$ along the query dimension. Through tile-level equality saturation, Trinity integrates these extra computations into one kernel for execution. TorchInductor and TensorRT use a manually written FlashAttention kernel, but they execute the extra computations in separate kernels. Mirage, on the other hand, suffers from an expanded search space due to the newly added operators, which increases compile time and prevents it from finding the optimal kernel.

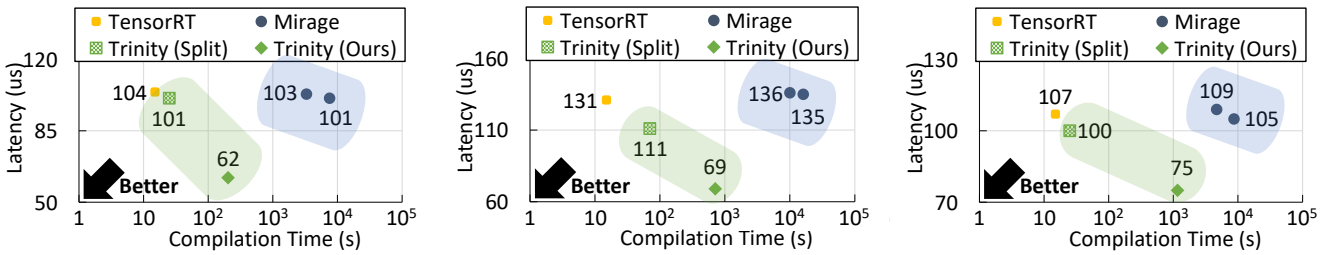
6.3 Automatic hardware adaptation

Unlike manual kernel optimization that requires separate implementations for each hardware platform, Trinity automatically discovers different optimal kernels tailored to each hardware platform. Figures 6 and 7 show the latency of high-performance kernels discovered by Trinity on RTX4090 and H100 GPUs. Each kernel represents the best-performing one under different GPUs (H100, A100, A40, RTX5090, RTX4090) and model configurations (LLaMA3 8B, Falcon 7B). For KeyFormer, eight unique optimal kernels were discovered across ten combinations, while for RoCo, five distinct kernels covered all ten.

For KeyFormer, contrasting hardware characteristics lead to different optimal strategies. On the H100, Trinity leverages the GPU's high memory bandwidth (3 TB/s) by adopting a *mid-kernel spill* strategy, where intermediate tiles (e.g., logits, accumulators) are temporarily written to off-chip memory instead of being held entirely in on-chip memory. This approach alleviates shared-memory pressure and enables the use of larger tile sizes (128), which in turn reduces the number of loop iterations and improves overall performance despite the additional memory traffic (kernel `H` in Figure 6). In contrast, on the bandwidth-constrained RTX 4090 (1 TB/s), the same spill-heavy schedule becomes inefficient, as the added off-chip traffic quickly saturates available memory bandwidth. The optimal kernel in this setting is kernel `A`, which keeps all intermediate values on-chip and minimizes global-memory traffic. Although this choice necessitates smaller tiles (64) and roughly twice as many loop iterations, it avoids bandwidth bottlenecks and ultimately delivers higher performance on the RTX 4090.

These results show how Trinity adapts to hardware: prioritizing computational efficiency on bandwidth-rich platforms (H100) while prioritizing memory locality on bandwidth-constrained platforms (RTX 4090). Such hardware-specific tuning is cumbersome to perform manually, highlighting the importance of automated optimization like Trinity.

Moreover, these findings explain why existing joint optimizers such as Mirage fall short. Although kernels A and H implement the same algebraic computation, they differ significantly in their data-movement behavior. Because Mirage's IR does not treat memory I/O and loop structure as first-class constructs, it collapses these hardware-specialized schedules into the same representation and therefore cannot truly perform joint optimization across the three optimization dimensions.



(a) Vanilla (LLaMA3 8B configuration) **(b)** RoCo (Falcon 7B configuration) **(c)** Pre-Norm (LLaMA3 8B configuration)

Figure 8. Inference latency over compile time (H100). Trinity (split) indicates results obtained by splitting the input into subprograms for optimization, as done in Mirage.

6.4 Compilation Time

Figure 8 shows the trade-off between compilation time and kernel performance across different approaches. We vary compilation time to observe how performance improves, revealing fundamental differences in optimization strategies.

TensorRT represents traditional separated optimization—it compiles quickly but results in suboptimal performance due to limited transformation scope. Mirage attempts joint optimization through exhaustive search, but compilation time explodes with program size. To remain tractable, Mirage must partition programs into small fragments and optimize them independently, sacrificing cross-partition optimizations. To demonstrate the trade-off, we intentionally constrained Trinity by applying the same program partitioning strategy as Mirage. Even with these artificial constraints, Trinity achieves comparable performance to Mirage in significantly less time, demonstrating the efficiency of equality saturation.

Trinity's true strength lies in optimizing large, unpartitioned search spaces. While Mirage cannot handle programs with more than 11 operators without partitioning, Trinity successfully optimizes entire architectural components: QK-Norm (21 operations), RoCo (22 operations), and SwiGLU-FFN (12 operations) in just 375, 710, and 266 seconds respectively. These holistic optimizations discover cross-operator patterns impossible to find with partitioned search, explaining Trinity's superior performance. For these same programs, Mirage's exhaustive approach would require prohibitive compilation times, making such optimization infeasible in practice.

Scalability to large search spaces. To evaluate Trinity's ability to handle the combinatorial explosion of tile-level optimization, we measure both the size of search spaces it explores and the compilation time required. Table 2 shows that Trinity successfully scales to astronomical search spaces, discovering up to 10^{21} equivalent programs for complex architectures like KeyFormer.

We evaluate the effectiveness of our e-graph representation by measuring how compactly it represents the search space. For the Vanilla transformer, Trinity represents 2×10^{17} equivalent programs using just 434 e-classes and 2,058 e-nodes. This compact e-graph structure enables our two-pass extraction algorithm to efficiently identify 170 high-performance candidates in 54 seconds. With eight GPUs used for profiling, the entire process completes in 203 seconds.

Benchmark	# Possible	Trinity (ours)		Mirage
		Saturate/Extract/Profile	Total	
SwiGLU-FFN	2×10^{12}	10s / 30s / 226s	266s	348s (1.3x)
Vanilla	2×10^{17}	7s / 54s / 142s	203s	7741s (38.1x)
QK-Norm	4×10^{17}	10s / 167s / 198s	375s	4039s (10.7x)
Pre-Norm	10^{20}	14s / 922s / 226s	1162s	8678s (7.5x)
RoCo	2×10^{20}	60s / 592s / 58s	710s	16062s (22.6x)
KeyFormer	10^{21}	49s / 1305s / 105s	1459s	15963s (10.9x)

Table 2. Search space size and compilation time. # Possible indicates the number of equivalent programs discovered.

Compilation time scales reasonably with model complexity. Simple architectures like SwiGLU optimize in 266 seconds, while complex models with richer optimization opportunities like KeyFormer require up to 1,459 seconds. The breakdown shows that extraction time dominates for complex models (1,305s for KeyFormer), validating our two-pass algorithm design—the investment in careful extraction pays off by finding high-quality kernels from massive search spaces. These results confirm that Trinity achieves practical compilation times even when exploring search spaces that would require weeks to enumerate exhaustively. Note that Mirage's compilation time refers to the time taken to compile after splitting the input program into smaller subprograms.

7 Related Work & Discussion

Hand-optimized kernels like cuDNN [12] and FlashAttention [13, 14, 39] achieve performance through algorithmic innovations. **Tile-based programming interfaces**, including CUTLASS [45], Triton [46], Graphene [19], Cypress [57], and ThunderKittens [42] reduce manual effort by exposing tunable tile-level abstractions, including blocking, memory movement, and execution schedules. SpatialDSL [25] similarly provides a hardware-agnostic accelerator programming model, while NKI [4] exposes a vendor-specific tile programming interface for Amazon accelerators. Despite improved productivity, these interfaces still rely on human-designed

algorithms. Trinity automatically discovers these optimizations through tile-level exploration.

Tensor graph optimization. TASO [23], TENSAT [59], Hartmann et al. [20], PET [49], EINNET [63], and FlashTensor [67] are graph rewriting systems that apply algebraic transformations at the tensor level, missing tile-level optimizations like memory reuse patterns and kernel fusion that require reasoning about tensor partitioning.

Operator fusion [10, 28, 30, 33, 34, 66] groups adjacent operators into single kernels to reduce memory traffic and kernel launch overhead. These systems typically use pattern matching to identify fusible operator sequences—for example, fusing element-wise operations following matrix multiplication. However, fusion decisions must be made before optimization based on predetermined patterns, limiting their effectiveness. The systems cannot discover novel fusion opportunities that require algebraic transformation. For instance, they cannot automatically discover that QKV projection and attention can be fused after algebraic reordering (Figure 4), since this requires interleaving their computations rather than simple adjacency.

Operator-level optimization. Tensor compilers optimize individual operators through loop transformations. Halide [36], AutoTVM [11], and TorchInductor [5] apply loop transformations on templates. FlexTensor [65] and Ansor [62] automatically generate templates. TensorIR [16] focuses on tensorization, Tiramisu [6] and AKG [61] utilize polyhedral representation, while TACO [24] and UCF [52] target sparse operators. However, these approaches optimize operators independently, missing cross-operator opportunities that Trinity captures through tile-level optimization.

Joint optimization attempts. Recent work targets specific aspects of multi-level optimization: Welder [41] improves memory reuse through tile-level scheduling, ASPEN [35] removes synchronization barriers, Rammer [28] optimizes kernel scheduling, ALT [56] jointly optimizes layout and loops within operators. In the context of spatial accelerators, AMOS [64] and Marvel [9] automate tile-level mapping and dataflow decisions, jointly reasoning about computation partitioning, memory hierarchy usage, and on-chip communication. Korch [22] and Souffle [54] find optimal kernel boundaries top-down, and Relax [26] unifies abstractions for dynamic shapes. Each addresses specific optimization goals with specialized techniques. Trinity provides a general framework where these optimizations emerge naturally from equality saturation without hard-coding strategies.

Mirage [53], the most recent related work, explores joint transformations but exhaustively generates μGraphs by incrementally adding operators. This exponential approach forces partitioning into small fragments (5 kernels, 11 block operators), preventing cross-partition optimizations.

Equality saturation [43] is being adopted in various domains to replace heuristic optimizations. Cranelift [8] is a compiler backend for code generation, which embeds multi-pass optimizations into rewrite rules under the equality saturation framework. TENSAT [59] performs tensor-level graph optimization, while SPORES [50] performs sketch-based program optimization using equality saturation.

Limitations and future opportunities. Our work presents significant opportunities for future research. (1) By adding operators and rewrite rules to support backpropagation graphs and training-time optimization, Trinity could be extended to optimize training workloads. (2) We are confident that Trinity can be extended to support the advanced GPU components and optimizations of NVIDIA's Hopper architectures, such as warp specialization and TMA units, which are necessary for FlashAttention3 [39], for further performance improvement. Currently, Trinity relies on Triton and does not yet support these features. (3) Trinity can be extended to support novel mathematical operators introduced by new models. For instance, we can support grouped query attention [2] by introducing rewrite rules for hierarchical tiling on the same axis to enable parallel reduction. (4) Trinity struggles with very large programs (e.g., combined attention and FFN) despite equality saturation's improved scalability. This stems from applying all rules uniformly for fixed iterations; larger programs require more selective rule application.

8 Conclusion

Trinity fundamentally rethinks tensor program optimization by introducing tile-level equality saturation, automatically discovering optimizations that previously required manual expertise. The key insight is that optimal performance requires jointly optimizing three interdependent dimensions: algebraic equivalence, memory I/O, and compute orchestration. By applying equality saturation at tile granularity, Trinity discovers complex optimizations that existing systems miss. Our IR and two-pass extraction algorithm make this joint optimization tractable, achieving up to 2.09× speedup over existing state-of-the-art systems. Trinity demonstrates that sophisticated kernel optimizations emerge naturally from systematic exploration rather than manual engineering.

Acknowledgments

We thank Seunggeun Cho and Jihyuk Lee for their assistance with Mirage and Relax reproduction. We also thank the anonymous reviewers and our shepherd, Olivia Hsu, for providing helpful feedback and suggestions to improve our work. This work was supported by National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2024-00340099), and Institute of Information & Communications Technology Planning & Evaluation (IITP) of the Korea government (MSIT) (No. RS-2024-00398157).

References

[1] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant J Nair, Ilya Soloveychik, and Purushotham Kamath. 2024. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. *Proceedings of Machine Learning and Systems* 6 (2024), 114–127.

[2] Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245* (2023).

[3] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, et al. 2023. The falcon series of open language models. *arXiv preprint arXiv:2311.16867* (2023).

[4] Amazon Web Services. 2024. Neuron Kernel Interface (NKI). https://awsdocs-neuron.readthedocs-hosted.com/en/latest/nki/index.html. Accessed: 2026-01-05.

[5] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 929–947.

[6] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.

[7] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[8] Bytecode Alliance. [n. d.]. Cranelift E-graph (RFC). https://github.com/bytecodealliance/rfcs/blob/main/accepted/cranelift-egraph.md. Accessed: 2026-01-05.

[9] Prasanth Chatarasi, Hyoukjun Kwon, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. 2021. Marvel: A data-centric approach for mapping deep learning operators on spatial accelerators. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 1 (2021), 1–26.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[11] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems* 31 (2018).

[12] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[13] Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691* (2023).

[14] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems* 35 (2022), 16344–16359.

[15] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv e-prints* (2024), arXiv–2407.

[16] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. 2023. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 804–817.

[17] Jared Fernandez, Clara Na, Vashisth Tiwari, Yonatan Bisk, Sasha Luccioni, and Emma Strubell. 2025. Energy considerations of large language model inference and efficiency optimizations. *arXiv preprint arXiv:2504.17674* (2025).

[18] Amir Kafshdar Goharshady, Chun Kit Lam, and Lionel Parreaux. 2024. Fast and optimal extraction for sparse equality graphs. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 2551–2577.

[19] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. 2023. Graphene: An ir for optimized tensor computations on gpus. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 302–313.

[20] Jakob Hartmann, Guoliang He, and Eiko Yoneki. 2024. Optimizing Tensor Computation Graphs with Equality Saturation and Monte Carlo Tree Search. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*. 40–52.

[21] Alex Henry, Prudhvi Raj Dachapally, Shubham Shantaram Pawar, and Yuxuan Chen. 2020. Query-Key Normalization for Transformers. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 4246–4253. doi:10.18653/v1/2020.findings-emnlp.379

[22] Muyan Hu, Ashwin Venkatram, Shreyashri Biswas, Balamurugan Marimuthu, Bohan Hou, Gabriele Oliaro, Haojie Wang, Liyan Zheng, Xupeng Miao, Jidong Zhai, et al. 2024. Optimal kernel orchestration for tensor programs with korch. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 755–769.

[23] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62.

[24] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[25] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. 2018. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 296–311.

[26] Ruihang Lai, Junru Shao, Siyuan Feng, Steven Lyubomirsky, Bohan Hou, Wuwei Lin, Zihao Ye, Hongyi Jin, Yuchen Jin, Jiawei Liu, et al. 2025. Relax: composable abstractions for end-to-end dynamic machine learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 998–1013.

[27] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*. PMLR, 19274–19286.

[28] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 881–897.

[29] Kathryn S McKinley, Steve Carr, and Chau-Wen Tseng. 1996. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18, 4 (1996), 424–453.

[30] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 883–898.

[31] NVIDIA Corporation. 2020. *NVIDIA A100 Tensor Core GPU Architecture*. White Paper. NVIDIA Corporation. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf Version 1.0.

[32] NVIDIA Corporation. 2023. *NVIDIA H100 Tensor Core GPU Architecture*. White Paper. NVIDIA Corporation. https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c Version 1.0.

[33] NVIDIA Corporation. 2025. NVIDIA/TensorRT – GitHub Repository. https://github.com/NVIDIA/TensorRT. Accessed: 2025-08-10.

[34] OpenXLA Project Developers. 2024. XLA - OpenXLA Project. OpenXLA Website. https://openxla.org/xla

[35] Jongseok Park, Kyungmin Bin, Gibum Park, Sangtae Ha, and Kyunghan Lee. 2023. Aspen: Breaking operator barriers for efficient parallelization of deep neural networks. *Advances in Neural Information Processing Systems* 36 (2023), 68625–68638.

[36] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[37] Siyu Ren and Kenny Q Zhu. 2024. On the efficacy of eviction policy for key-value constrained generative language model inference. *arXiv preprint arXiv:2402.06262* (2024).

[38] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages*. 58–68.

[39] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems* 37 (2024), 68658–68685.

[40] Noam Shazeer. 2020. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202* (2020).

[41] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. 2023. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 701–718.

[42] Benjamin F Spector, Simran Arora, Aaryan Singhal, Daniel Y Fu, and Christopher Ré. 2024. Thunderkittens: Simple, fast, and adorable ai kernels. *arXiv preprint arXiv:2410.20399* (2024).

[43] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 264–276.

[44] Chameleon Team. 2024. Chameleon: Mixed-modal early-fusion foundation models. *arXiv preprint arXiv:2405.09818* (2024).

[45] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. 2023. *CUTLASS*. https://github.com/NVIDIA/cutlass

[46] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.

[47] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[49] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. 2021. {PET}: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 37–54.

[50] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: sum-product optimization via relational equality saturation for large scale linear algebra. *Proc. VLDB Endow.* 13, 12 (July 2020), 1919–1932. doi:10.14778/3407790.3407799

[51] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29.

[52] Jaeyeon Won, Changwan Hong, Charith Mendis, Joel Emer, and Saman Amarasinghe. 2023. Unified Convolution Framework: A compiler-based approach to support sparse convolutions. *Proceedings of Machine Learning and Systems* 5 (2023), 666–679.

[53] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. 2025. Mirage: A {Multi-Level} Superoptimizer for Tensor Programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 21–38.

[54] Chunwei Xia, Jiacheng Zhao, Qianqi Sun, Zheng Wang, Yuan Wen, Teng Yu, Xiaobing Feng, and Huimin Cui. 2024. Optimizing deep learning inference via global analysis and tensor expressions. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 286–301.

[55] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. 2020. On Layer Normalization in the Transformer Architecture. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 10524–10533. https://proceedings.mlr.press/v119/xiong20b.html

[56] Zhiying Xu, Jiafan Xu, Hongding Peng, Wei Wang, Xiaoliang Wang, Haoran Wan, Haipeng Dai, Yixu Xu, Hao Cheng, Kun Wang, et al. 2023. ALT: Breaking the wall between data layout and loop optimizations for deep learning compilation. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 199–214.

[57] Rohan Yadav, Michael Garland, Alex Aiken, and Michael Bauer. 2025. Task-Based Tensor Computations on Modern GPUs. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 396–420.

[58] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025).

[59] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality saturation for tensor graph superoptimization. *Proceedings of Machine Learning and Systems* 3 (2021), 255–268.

[60] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. 2025. FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving. *arXiv preprint arXiv:2501.01005* (2025). https://arxiv.org/abs/2501.01005

[61] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, et al. 2021. AKG: automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1233–1248.

[62] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[63] Liyan Zheng, Haojie Wang, Jidong Zhai, Muyan Hu, Zixuan Ma, Tuowei Wang, Shuhong Huang, Xupeng Miao, Shizhi Tang, Kezhao Huang, et al. 2023. {EINNET}: Optimizing tensor programs with {Derivation-Based} transformations. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 739–755.

[64] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 874–887.

[65] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 859–873.

[66] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang, and Wei Lin. 2020. Fusionstitching: boosting memory intensive computations for deep learning workloads. *arXiv preprint arXiv:2009.10924* (2020).

[67] Runxin Zhong, Yuyang Jin, Chen Zhang, Kinman Lei, Shuangyu Li, and Jidong Zhai. 2025. FlashTensor: Optimizing Tensor Programs by Leveraging Fine-grained Tensor Property. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 183–196.

A List of Transformation Rules

A.1 Loop Transformations

Rule Name	IR transformation	Safety Condition
seq-comm	(seq a b) ⟶ (seq b a)	∀var∈vars(a)∪vars(b). no_dep(a->b, var) && a != (seq ...) && b != (seq ...)
seq-comm-tail	(seq a (seq b tail)) ⟶ (seq b (seq a tail))	∀var∈vars(a)∪vars(b). no_dep(a->b, var) && a != (seq ...) && b != (seq ...)
loop-fusion	(seq (loop n tile_n var body1) (seq (loop n tile_n var body2) others)) ⟶ (seq (loop n tile_n var (seq body1 body2)) others)	no_raw_waw_dep(body1->body2, var)
loop-fusion-tail	(seq (loop n tile_n var body1) (loop n tile_n var body2)) ⟶ (loop n tile_n var (seq body1 body2))	no_raw_waw_dep(body1->body2, var)
loop-fission-tail	(seq (loop 0 n tile_n var (seq body1 body2)) others) ⟶ (seq (loop 0 n tile_n var body1) (seq (loop 0 n tile_n var body2) others))	no_raw_waw_dep(body1->body2, var)
loop-fission	(loop 0 n tile_n var (seq body1 body2)) ⟶ (seq (loop 0 n tile_n var body1) (loop 0 n tile_n var body2))	no_raw_waw_dep(body1->body2, var)
loop-insertion1-tail	(seq (loop 0 n tile_n var body1) (seq body2 others)) ⟶ (seq (loop 0 n tile_n var (seq body1 body2)) others)	no_raw_waw_dep(body1->body2, var)
loop-insertion1	(seq (loop 0 n tile_n var body1) body2) ⟶ (loop 0 n tile_n var (seq body1 body2))	body2 != (seq ...) && no_raw_waw_dep(body1->body2, var)
loop-insertion2-tail	(seq body1 (seq (loop 0 n tile_n var body2) others)) ⟶ (seq (loop 0 n tile_n var (seq body1 body2)) others)	no_raw_waw_dep(body1->body2, var)
loop-insertion2	(seq body1 (loop 0 n tile_n var body2)) ⟶ (loop 0 n tile_n var (seq body1 body2))	no_raw_waw_dep(body1->body2, var)
loop-invar-code-motion	(loop 0 n tile_n var body) ⟶ body	var ∉ vars(body)

alge-factor-loop-body-div	(loop 0 n tile_n var
	  (store b (+ (* (load b idx) accm) (/ val1 val2)) idx))
	→ (seq (loop 0 n tile_n var
	  (store b (+ (* (load b idx) accm) val1) idx))
	  (store b (/ (load b idx) val2) idx))
		var ∉ vars(val2)

alge-factor-loop-body-div-tail	(seq (loop 0 n tile_n var
	  (store b (+ (* (load b idx) accm) (/ val1 val2)) idx)) others)
	→ (seq (loop 0 n tile_n var
	  (store b (+ (* (load b idx) accm) val1) idx))
	  (seq (store b (/ (load b idx) val2) idx) others))
		var ∉ vars(val2)

alge-factor-loop-body-mul	(loop 0 n tile_n var
	  (store b (+ (* (load b idx) accm) (* val1 val2)) idx))
	→ (seq (loop 0 n tile_n var
	  (store b (+ (* (load b idx) accm) val1) idx))
	  (store b (* (load b idx) val2) idx))
		var ∉ vars(val2)

alge-factor-loop-body-mul-tail	(seq (loop 0 n tile_n var
	  (store b (+ (* (load b idx) accm) (* val1 val2)) idx)) others)
	→ (seq (loop 0 n tile_n var
	  (store b (+ (* (load b idx) accm) val1) jidx))
	  (seq (store b (* (load b idx) val2) idx) others)
		var ∉ vars(val2)

A.2 Algebraic Transformations

Rules	IR transformation	Safety Condition
comm-add	(+ a b) ⟶ (+ b a)	
assoc-add	(+ a (+ b c)) ⟶ (+ (+ a b) c)	
assoc-add2	(+ (+ a b) c) ⟶ (+ a (+ b c))	
comm-mul	(* a b) ⟶ (* b a)	
assoc-mul	(* a (* b c)) ⟶ (* (* a b) c)	
assoc-mul2	(* (* a b) c) ⟶ (* a (* b c))	
assoc-matmul	(matmul a (matmul b c)) ⟶ (matmul (matmul a b) c)	
assoc-matmul2	(matmul (matmul a b) c) ⟶ (matmul a (matmul b c))	
assoc-div-matmul	(matmul (/ a (bcast b axis)) c) ⟶ (/ (matmul a c) (bcast b axis))	
dist-mul-add	(* a (+ b c)) ⟶ (+ (* a b) (* a c))	
dist-mul-sub	(* a (- b c)) ⟶ (- (* a b) (* a c))	
dist-matmul-add	(matmul a (+ b c)) ⟶ (+ (matmul a b) (matmul a c))	
dist-matmul-sub	(matmul a (- b c)) ⟶ (- (matmul a b) (matmul a c))	
factor-mul-add	(+ (* a b) (* a c)) ⟶ (* a (+ b c))	
factor-mul-sub	(- (* a b) (* a c)) ⟶ (* a (- b c))	
factor-matmul-add	(+ (matmul a b) (matmul a c)) ⟶ (matmul a (+ b c))	
factor-matmul-sub	(- (matmul a b) (matmul a c)) ⟶ (matmul a (- b c))	
exp-mul	(* (exp a) (exp b)) ⟶ (exp (+ a b))	
exp-div	(/ (exp a) (exp b)) ⟶ (exp (- a b))	
exp0	(exp 0) ⟶ 1	
recip-mul-div	(* x (/ 1 x)) ⟶ 1	x != 0
geometry-of-concat	(concat (concat x z 0) (concat y w 0) 1) ⟶ (concat (concat x y 1) (concat z w 1) 0)	
geometry-of-concat-inv	(concat (concat x y 1) (concat z w 1) 0) ⟶ (concat (concat x z 0) (concat y w 0) 1)	
operator-comm6	(+ (concat x z a) (concat y w a)) ⟶ (concat (+ x y) (+ z w) a)	
operator-comm6-inv	(concat (+ x y) (+ z w) a) ⟶ (+ (concat x z a) (concat y w a))	
operator-comm7	(* (concat x z a) (concat y w a)) ⟶ (concat (* x y) (* z w) a)	
operator-comm7-inv	(concat (* x y) (* z w) a) ⟶ (* (concat x z a) (concat y w a))	
concat-and-matmul0	(matmul x (concat y z 1)) ⟶ (concat (matmul x y) (matmul x z) 1)	
concat-and-matmul0-inv	(concat (matmul x y) (matmul x z) 1) ⟶ (matmul x (concat y z 1))	
concat-and-matmul1	(+ (matmul a b) (matmul c d)) ⟶ (matmul (concat a c 1) (concat b d 0))	
concat-and-matmul1-inv	(matmul (concat a c 1) (concat b d 0)) ⟶ (+ (matmul a b) (matmul c d))	

B Implementation Details

B.1 Trinity implementation

We implement our end-to-end tensor program optimizer based on the Rust egg library [51], which is an e-graph framework optimized for equality saturation. It provides convenient APIs and utilities for defining custom IRs and rewrite rules. In particular, egg provides a feature called e-class analysis, which enables rewrites that account not only for syntax but also for semantics. Trinity leverages this to track memory dependencies by recording read/write accesses in each e-class and to check tile shapes. Additionally, to reduce overhead, algorithms for rewrite pattern matching like illegal sequence flattening and expression propagation are triggered only when changes occur in e-class analysis.

Trinity applies rewrite rules for up to 10 iterations. While ideally the process continues until the e-graph saturates, we observe that applying up to 10 iterations consistently yields meaningful performance improvements. Beyond that point, further rewrites mostly involve redundant commutativity among operations, resulting in negligible performance gains.

Using operations in Table 1, we can represent activation functions used in our benchmark such as SiLU, tanh, and softmax. Trinity is an extensible system allowing new operators and rewrite rules, such as max and erf, to represent ReLU and GELU. During profiling in a cloud environment, we observed no signs of throttling. Furthermore, multiple profiling runs produced consistent results.

B.2 Mirage Implementation

To measure the latency of various attention or MLP kernels, we modified several configurations in Mirage v0.2.4. Since valid μGraphs could not be found with the default settings, we increased the maximum number of kernel graph operations from 5 to 7, based on Mirage v0.2.2, and changed the maximum number of threadblock graph operations to 11, following the recent Mirage paper's configurations.

Additionally, when attempting graph transpiling with version 0.2.4 on H100 environments, we encountered compiler assertion failures. To resolve this issue, we modified the code to use the same graph transpiler for H100 environments as the one used for A100.

When implementing each attention or MLP kernel in Mirage, we included as many Mirage-supported operations as possible. However, in cases where graph exploration was not feasible or operations were unsupported, we replaced them with PyTorch operators.

B.3 FlashTensor Implementation

Our evaluation of FlashTensor as a baseline faced several implementation challenges. First, its current implementation focuses solely on attention operations, leaving essential operators unsupported—specifically, reduce sum, sigmoid, and square root. Second, FlashTensor relies on heuristic-based rules for kernel partitioning, which leads to "bad partitioning" when extending beyond predefined attention patterns. To ensure a fair comparison, we adopted the following approach. For operations not supported by FlashTensor, such as QKV projections and normalization layers, we implemented them using PyTorch and measured their performance independently.

C Why FlashAttention requires all three dimensions.

Standard attention implementations [48] compute the attention head ($\texttt{softmax}(Q \cdot K^T) \cdot V$) by first materializing the entire intermediate tensor $Q \cdot K^T$ into global memory. This is because softmax includes finding the global maximum value for each row. This requires $O(N^2)$ memory access for the sequence length of N, becoming prohibitive for long sequences.

FlashAttention's online softmax allows tiled computation throughout the entire attention head, allowing intermediate tensors to stay in scratchpad memory, reducing memory access from $O(N^2)$ to $O(N)$. This optimization requires simultaneous reasoning across all the following three dimensions:

1. Algebraic equivalence: The key insight is transforming softmax from a two-pass algorithm (compute denominator, then normalize) to a single-pass incremental algorithm. Online softmax maintains running statistics (maximum and sum) that can be updated as each tile arrives. When processing a new tile, it computes the local maximum and sum, updates the global maximum as $m_{\text{global}} = \max(m_{\text{global}}, m_{\text{local}})$, and rescales the previous sum using $s_{\text{global}} = s_{\text{global}} \cdot e^{m_{\text{old}} - m_{\text{new}}} + s_{\text{local}}$. This algebraic transformation from standard to online softmax isn't a simple local rewrite. It requires reorganizing the entire computation pattern to maintain statistics incrementally rather than computing them in separate passes.

2. Memory I/O: The transformation only becomes profitable when combined with careful memory management. Tiles of Q, K, and V must be loaded in a specific order, intermediate statistics (running max and sum) kept in on-chip memory across iterations, and partial results accumulated without writing back to off-chip memory until complete. This reduces memory traffic from $O(N^2)$ to $O(N)$.

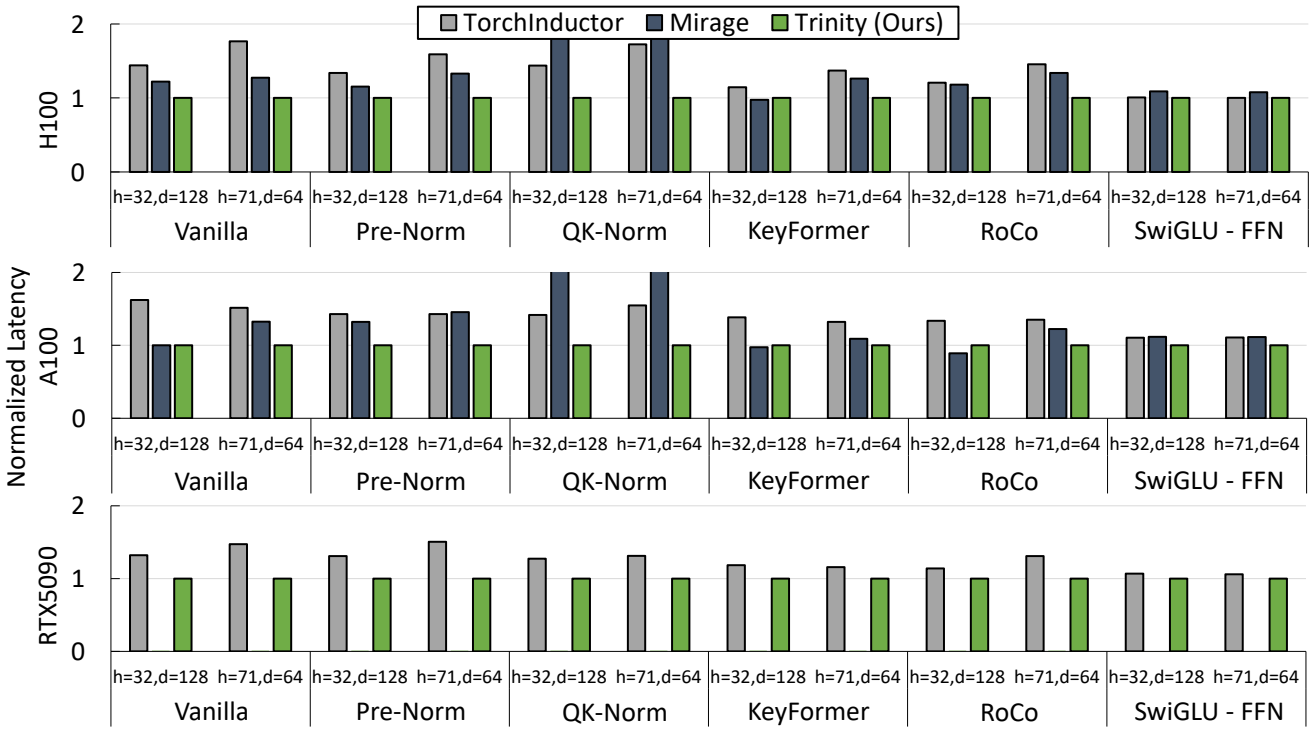


Figure 9. Normalized inference latency across various benchmarks and GPU without CUDA Graphs.

3. Compute orchestration: The online softmax algorithm fundamentally changes both execution boundaries and parallelization opportunities. It enforces sequential key tile processing within each query to maintain running statistics, creating loop-carried dependencies through rescaling operations. To maximize efficiency despite this constraint, implementations fuse QKV projection through attention into a single kernel, avoiding intermediate materialization.

D Inference latency with CUDA Graphs

CUDA Graphs are a CUDA runtime feature that records a fixed sequence of GPU operations and replays them as a single graph, reducing launch overhead and improving performance when the execution pattern is highly regular. Figure 9 shows normalized inference latency with CUDA Graphs enabled. As shown, Trinity still outperforms baselines when CUDA Graphs are applied.

However, CUDA Graphs are applicable only under restricted conditions-requiring static shapes and stable execution paths-and most baselines do not enable them in their default evaluation modes. To maintain fairness and reflect commonly deployable settings, our main evaluation does not include CUDA Graph results.

E Trinity IR Programs for Benchmarks

We present several examples of Trinity IR along with the corresponding hardware kernel code generated from it. For brevity, only the function body of each hardware kernel is shown. Note that boilerplate code such as offset, mask, and block size is also automatically generated by Trinity.

Figure 10 shows our IR for the entire Vanilla architecture described in § 5.3, and Figure 11 presents the hardware kernel automatically generated from it. Figure 12 shows our best IR for the entire KeyFormer architecture described in § 6.3 on H100, and Figure 13 presents the hardware kernel automatically generated from it. Figure 14 shows our best IR for the entire KeyFormer architecture described in § 6.3 on RTX 4090, and Figure 15 presents the hardware kernel automatically generated from it.

```
(loop 0 4096 128 n (seq
 (loop 0 4096 tile_k k k
   (store (tensor Q1,K1,V1)
     (+(* (load (tensor Q1,K1,V1) (index fulltile (tile n))) 1)
       (matmul (load (input X) (index fulltile (tile k)))
         (load (input WQ,WK,WV) (index (tile k) (tile n))))
     ) (index fulltile (tile n))))
 (seq
   (loop 0 1024 tile_p p p
     (seq dummy (seq
       (store (tensor Q,K,V)
        (permute3 (unsqueeze (load (tensor Q1,K1,V1) (index fulltile (tile n))) 1) 1 0 2)
         (index (elem n) fulltile fulltile))
       (seq
         (store (input K_cache,V_cache)
           (load (tensor K,V) (index (elem n) fulltile fulltile))
           (index (elem n) (const_tile 1008 16) fulltile))
         (seq dummy (seq
           (store (tensor C_exp)
             (exp
              (matmul (load (tensor Q) (index (elem n) fulltile fulltile))
              (permute3 (load (input K_cache) (index (elem n) (tile p) fulltile)) 0 2 1))
             ) (index (elem n) fulltile (tile p)))
           (seq
             (store (tensor C_sum)
               (+(rsum (load (tensor C_exp) (index (elem n) fulltile (tile p))) 2)
                 (* 1 (load (tensor C_sum) (index (elem n) fulltile)))
               ) (index (elem n) fulltile))
             (store (tensor O)
               (+(matmul (load (tensor C_exp) (index (elem n) fulltile (tile p)))
                   (load (input V_cache) (index (elem n) (tile p) fulltile)))
                 (* 1 (load (tensor O) (index (elem n) fulltile fulltile)))
               ) (index (elem n) fulltile fulltile)))))))))
   (seq (loop 0 1024 tile_p p dummy )) (seq
     (store (tensor O)
       (/(load (tensor O) (index (elem n) fulltile fulltile))
         (bcast (load (tensor C_sum) (index (elem n) fulltile)) 2)
       ) (index (elem n) fulltile fulltile))
     (seq dummy (store (output O2)
       (squeeze (permute3 (load (tensor O) (index (elem n) fulltile fulltile)) 1 0 2) 1)
       (index fulltile (tile n))))))))
```

Figure 10. The highest performance IR of Vanilla architecture on NVIDIA H100 GPU.

```
# Allocate intermediate tensors
C_exp = tl.zeros((1, M, BLOCK_P), dtype=tl.float32)
C_sum = tl.zeros((1, M), dtype=tl.float32)
K = tl.zeros((1, M, D), dtype=tl.float16)
K1 = tl.zeros((M, BLOCK_N), dtype=tl.float16)
O = tl.zeros((1, M, D), dtype=tl.float32)
Q = tl.zeros((1, M, D), dtype=tl.float16)
Q1 = tl.zeros((M, BLOCK_N), dtype=tl.float16)
V = tl.zeros((1, M, D), dtype=tl.float16)
V1 = tl.zeros((M, BLOCK_N), dtype=tl.float16)

# Parallel loop n from 0 to Q1_dim1 with tile size BLOCK_N
n = 0 + tl.program_id(0) * BLOCK_N

# Sequential loop k from 0 to 4096 with tile size BLOCK_K
for k in range(0, 4096, BLOCK_K):
   temp_0 = tl.load(X_ptr + offset_0, mask=mask_0, other=0.0)
   temp_1 = tl.load(WQ_ptr + offset_1, mask=mask_1, other=0.0)
   Q1 = (tl.dot(temp_0, temp_1).to(tl.float16) + (Q1 * 1).to(tl.float16)).to(tl.float16)
   temp_2 = tl.load(WK_ptr + offset_2, mask=mask_2, other=0.0)
   K1 = (tl.dot(temp_0, temp_2).to(tl.float16) + (K1 * 1).to(tl.float16)).to(tl.float16)
   temp_3 = tl.load(WV_ptr + offset_3, mask=mask_3, other=0.0)
   V1 = (tl.dot(temp_0, temp_3).to(tl.float16) + (V1 * 1).to(tl.float16)).to(tl.float16)
# Sequential loop p from 0 to 1024 with tile size BLOCK_P
for p in range(0, 1024, BLOCK_P):
   temp_4 = tl.expand_dims(Q1, 1)
   Q = tl.permute(temp_4, (1, 0, 2))
   temp_5 = tl.expand_dims(K1, 1)
   K = tl.permute(temp_5, (1, 0, 2))
   temp_6 = tl.expand_dims(V1, 1)
   V = tl.permute(temp_6, (1, 0, 2))
   tl.store(K_cache_ptr + offset_4, K)
   tl.store(V_cache_ptr + offset_5, V)
   temp_7 = tl.load(K_cache_ptr + offset_6, mask=mask_4, other=0.0)
   temp_8 = tl.permute(temp_7, (0, 2, 1))
   C_exp = tl.exp(tl.dot(Q, temp_8).to(tl.float32))
   C_sum = ((C_sum * 1) + tl.sum(C_exp, axis=2))
   temp_9 = tl.load(V_cache_ptr + offset_7, mask=mask_5, other=0.0)
   O = ((O * 1) + tl.dot(C_exp, temp_9.to(tl.float32)))
# Skipped empty sloop with dummy body
O = (O / C_sum[:, :, None])
temp_10 = tl.permute(O, (1, 0, 2))
tl.store(O2_ptr + offset_8, tl.reshape(temp_10, (M, D)).to(tl.float16), mask=mask_6)
```

Figure 11. The highest performance kernel of Vanilla architecture on NVIDIA H100 GPU generated from Figure 10. Boilerplate code such as offset, mask, and block size are omitted.

```
(loop 0 4096 128 n (seq
 (loop 0 4096 tile_k k (store (tensor Q1,K1,V1)
   (+(matmul (load (input X) (index fulltile (tile k)))
       (load (input WQ,WK,WV) (index (tile k) (tile n))))
     (* 1 (load (tensor Q1,K1,V1) (index fulltile (tile n))))
   ) (index fulltile (tile n))))
 (seq dummy (seq (store (tensor Q,K,V)
    (permute3 (unsqueeze (load (tensor Q1,K1,V1) (index fulltile (tile n))) 1) 1 0 2)
    (index (elem n) fulltile fulltile))
   (seq (store (input K_cache,V_cache)
       (load (tensor K,V) (index (elem n) fulltile fulltile))
       (index (elem n) (const_tile 1008 16) fulltile))
     (seq (loop 0 1024 tile_p p (store (tensor C)
         (matmul (load (tensor Q) (index (elem n) fulltile fulltile))
           (permute3 (load (input K_cache) (index (elem n) (tile p) fulltile)) 0 2 1)
         ) (index (elem n) fulltile (tile p))))
       (seq (loop 0 1024 tile_p p (seq
           (store (tensor C_perturb)
             (/(+(load (tensor C) (index (elem n) fulltile (tile p)))
               (load (input noise) (index (elem n) fulltile (tile p)))
             ) 1.5) (index (elem n) fulltile (tile p)))
           (store (tensor C_exp,C_exp_perturb)
             (exp (load (tensor C,C_perturb) (index (elem n) fulltile (tile p))))
             (index (elem n) fulltile (tile p)))))
        (seq (loop 0 1024 tile_p p (store (tensor C_sum,C_sum_perturb)
        (+(rsum (load (tensor C_exp,C_exp_perturb) (index (elem n) fulltile (tile p))) 2)
               (* 1 (load (tensor C_sum,C_sum_perturb) (index (elem n) fulltile))))
             (index (elem n) fulltile)))
           (seq (loop 0 1024 tile_p p (seq dummy
               (seq dummy (store (output C_out)
                 (rsum (/(load (tensor C_exp_perturb) (index (elem n) fulltile (tile p)))
                   (bcast (load (tensor C_sum_perturb) (index (elem n) fulltile)) 2)) 1)
                 (index (elem n) (tile p))))))
             (seq (loop 0 1024 tile_p p (store (tensor O)
                 (+(matmul (load (tensor C_exp) (index (elem n) fulltile (tile p)))
                     (load (input V_cache) (index (elem n) (tile p) fulltile)))
                   (* 1 (load (tensor O) (index (elem n) fulltile fulltile))))
                 (index (elem n) fulltile fulltile)))
               (seq (store (tensor O)
                   (/(load (tensor O) (index (elem n) fulltile fulltile))
                     (bcast (load (tensor C_sum) (index (elem n) fulltile)) 2))
                   (index (elem n) fulltile fulltile))
                 (seq dummy (store (output O2)
                   (squeeze (permute3
                     (load (tensor O) (index (elem n) fulltile fulltile))1 0 2) 1)
                   (index fulltile(tile n))
))))))))))))
```

Figure 12. The highest performance IR of KeyFormer architecture on NVIDIA H100 GPU.

```
# Allocate intermediate tensors
C_perturb = tl.zeros((1, M, BLOCK_P), dtype=tl.float16)
C_sum = tl.zeros((1, M), dtype=tl.float32)
C_sum_perturb = tl.zeros((1, M), dtype=tl.float32)
K1 = tl.zeros((M, BLOCK_N), dtype=tl.float16)
O = tl.zeros((1, M, D), dtype=tl.float32)
Q1 = tl.zeros((1, M, D), dtype=tl.float16)
V1 = tl.zeros((M, BLOCK_N), dtype=tl.float16)

n = 0 + tl.program_id(0) * BLOCK_N
for k in range(0, 4096, BLOCK_K):
   temp_0 = tl.load(X_ptr + offset_0, mask=mask_0, other=0.0)
   temp_1 = tl.load(WQ_ptr + offset_1, mask=mask_1, other=0.0)
   Q1 = (tl.dot(temp_0, temp_1).to(tl.float16) + (1 * Q1).to(tl.float16)).to(tl.float16)
   temp_2 = tl.load(WK_ptr + offset_2, mask=mask_2, other=0.0)
   K1 = (tl.dot(temp_0, temp_2).to(tl.float16) + (1 * K1).to(tl.float16)).to(tl.float16)
   temp_3 = tl.load(WV_ptr + offset_3, mask=mask_3, other=0.0)
   V1 = (tl.dot(temp_0, temp_3).to(tl.float16) + (1 * V1).to(tl.float16)).to(tl.float16)
temp_4 = tl.expand_dims(Q1, 1)
Q = tl.permute(temp_4, (1, 0, 2))
temp_5 = tl.expand_dims(K1, 1)
K = tl.permute(temp_5, (1, 0, 2))
temp_6 = tl.expand_dims(V1, 1)
V = tl.permute(temp_6, (1, 0, 2))
tl.store(K_cache_ptr + offset_4, K)
tl.store(V_cache_ptr + offset_5, V)
for p in range(0, 1024, BLOCK_P):
   temp_7 = tl.load(K_cache_ptr + offset_6, mask=mask_4, other=0.0)
   temp_8 = tl.permute(temp_7, (0, 2, 1))
   tl.store(C_ptr + offset_7, tl.dot(Q, temp_8).to(tl.float16), mask=mask_5)
for p in range(0, 1024, BLOCK_P):
   temp_9 = tl.load(C_ptr + offset_8, mask=mask_6, other=0.0)
   temp_10 = tl.load(noise_ptr + offset_9, mask=mask_7, other=0.0)
   C_perturb = ((temp_9 + temp_10).to(tl.float16) / 1.5).to(tl.float16)
   tl.store(C_exp_ptr + offset_10, tl.exp(temp_9.to(tl.float32)), mask=mask_8)
   tl.store(C_exp_perturb_ptr + offset_11, tl.exp(C_perturb.to(tl.float32)), mask=mask_9)
for p in range(0, 1024, BLOCK_P):
   temp_11 = tl.load(C_exp_ptr + offset_12, mask=mask_10, other=0.0)
   C_sum = (tl.sum(temp_11, axis=2) + (1 * C_sum)))
   temp_12 = tl.load(C_exp_perturb_ptr + offset_13, mask=mask_11, other=0.0)
   C_sum_perturb = (tl.sum(temp_12, axis=2) + (1 * C_sum_perturb))
for p in range(0, 1024, BLOCK_P):
   temp_13 = tl.load(C_exp_perturb_ptr + offset_14, mask=mask_12, other=0.0)
   tl.store(C_out_ptr + offset_15, tl.sum((temp_13 / C_sum_perturb[:, :,
None]).to(tl.float16), axis=1, dtype=tl.float16), mask=mask_13)
for p in range(0, 1024, BLOCK_P):
   temp_14 = tl.load(C_exp_ptr + offset_16, mask=mask_14, other=0.0)
   temp_15 = tl.load(V_cache_ptr + offset_17, mask=mask_15, other=0.0)
   O = (tl.dot(temp_14, temp_15.to(tl.float32)) + (1 * O))
O = (O / C_sum[:, :, None])
temp_16 = tl.permute(O, (1, 0, 2))
tl.store(O2_ptr + offset_18, tl.reshape(temp_16, (M, D)).to(tl.float16), mask=mask_16)
```

Figure 13. The highest performance kernel of KeyFormer architecture on NVIDIA H100 GPU generated from Figure 12. Boilerplate code such as offset, mask, and block size are omitted.

```
(loop 0 4096 64 n (seq
 (loop 0 4096 tile_k k (store (tensor Q1,K1,V1)
   (+(* (load (tensor Q1,K1,V1) (index fulltile (tile n))) 1)
     (matmul (load (input X) (index fulltile (tile k)))
       (load (input WQ,WK,WV) (index (tile k) (tile n)))))
   (index fulltile (tile n))))
 (seq (loop 0 1024 tile_p p (seq dummy
   (seq (store (tensor Q,K,V)
       (permute3 (unsqueeze (load (tensor Q1,K1,V1) (index fulltile (tile n))) 1) 1 0 2)
       (index (elem n) fulltile fulltile))
     (seq (store (input K_cache,V_cache)
         (load (tensor K,V) (index (elem n) fulltile fulltile))
         (index (elem n) (const_tile 1008 16) fulltile))
       (seq (store (tensor C)
           (matmul (load (tensor Q) (index (elem n) fulltile fulltile))
             (permute3 (load (input K_cache) (index (elem n) (tile p) fulltile)) 0 2 1))
           (index (elem n) fulltile (tile p)))
         (seq (store (tensor C_perturb)
             (/(+(matmul (load (tensor Q) (index (elem n) fulltile fulltile))
              (permute3 (load (input K_cache) (index (elem n) (tile p) fulltile)) 0 2 1))
               (load (input noise) (index (elem n) fulltile (tile p)))) 1.5)
             (index (elem n) fulltile (tile p)))
           (seq (store (tensor C_exp,C_exp_perturb)
               (exp (load (tensor C,C_perturb) (index (elem n) fulltile (tile p))))
               (index (elem n) fulltile (tile p)))
             (seq (store (tensor C_sum,C_sum_perturb)
                 (+(* 1 (load (tensor C_sum,C_sum_perturb) (index (elem n) fulltile)))
                   (rsum (exp
                     (load (tensor C,C_perturb) (index (elem n) fulltile (tile p)))) 2))
                 (index (elem n) fulltile))
               (store (tensor O)
                 (+(* (load (tensor O) (index (elem n) fulltile fulltile)) 1)
                   (matmul (load (tensor C_exp) (index (elem n) fulltile (tile p)))
                     (load (input V_cache) (index (elem n) (tile p) fulltile))))
                 (index (elem n) fulltile fulltile)))))))))))
 (seq (loop 0 1024 tile_p p dummy)
   (seq (loop 0 1024 tile_p p (seq dummy (store (output C_out)
       (rsum
         (/(load (tensor C_exp_perturb) (index (elem n) fulltile (tile p)))
           (bcast (load (tensor C_sum_perturb) (index (elem n) fulltile)) 2)) 1)
       (index (elem n) (tile p)))))
     (seq (store (tensor O)
         (/(load (tensor O) (index (elem n) fulltile fulltile))
           (bcast (load (tensor C_sum) (index (elem n) fulltile)) 2))
         (index (elem n) fulltile fulltile))
       (seq dummy (store (output O2)
         (squeeze (permute3
           (load (tensor O) (index (elem n) fulltile fulltile)) 1 0 2) 1)
         (index fulltile (tile n))
))))))))
```

Figure 14. The highest performance IR of KeyFormer architecture on NVIDIA RTX4090 GPU.

```
# Allocate intermediate tensors
C = tl.zeros((1, M, BLOCK_P), dtype=tl.float16)
C_exp = tl.zeros((1, M, BLOCK_P), dtype=tl.float32)
C_perturb = tl.zeros((1, M, BLOCK_P), dtype=tl.float16)
C_sum = tl.zeros((1, M), dtype=tl.float32)
C_sum_perturb = tl.zeros((1, M), dtype=tl.float32)
K = tl.zeros((1, M, D), dtype=tl.float16)
K1 = tl.zeros((M, BLOCK_N), dtype=tl.float16)
O = tl.zeros((1, M, D), dtype=tl.float32)
Q = tl.zeros((1, M, D), dtype=tl.float16)
Q1 = tl.zeros((M, BLOCK_N), dtype=tl.float16)
V = tl.zeros((1, M, D), dtype=tl.float16)
V1 = tl.zeros((M, BLOCK_N), dtype=tl.float16)

n = 0 + tl.program_id(0) * BLOCK_N
for k in range(0, 4096, BLOCK_K):
   temp_0 = tl.load(X_ptr + offset_0, mask=mask_0, other=0.0)
   temp_1 = tl.load(WQ_ptr + offset_1, mask=mask_1, other=0.0)
   Q1 = ((Q1 * 1).to(tl.float16) + tl.dot(temp_0, temp_1).to(tl.float16)).to(tl.float16)
   temp_2 = tl.load(WK_ptr + offset_2, mask=mask_2, other=0.0)
   K1 = ((K1 * 1).to(tl.float16) + tl.dot(temp_0, temp_2).to(tl.float16)).to(tl.float16)
   temp_3 = tl.load(WV_ptr + offset_3, mask=mask_3, other=0.0)
   V1 = ((V1 * 1).to(tl.float16) + tl.dot(temp_0, temp_3).to(tl.float16)).to(tl.float16)
for p in range(0, 1024, BLOCK_P):
   temp_4 = tl.expand_dims(Q1, 1)
   Q = tl.permute(temp_4, (1, 0, 2))
   temp_5 = tl.expand_dims(K1, 1)
   K = tl.permute(temp_5, (1, 0, 2))
   temp_6 = tl.expand_dims(V1, 1)
   V = tl.permute(temp_6, (1, 0, 2))
   tl.store(K_cache_ptr + offset_4, K)
   tl.store(V_cache_ptr + offset_5, V)
   temp_7 = tl.load(K_cache_ptr + offset_6, mask=mask_4, other=0.0)
   temp_8 = tl.permute(temp_7, (0, 2, 1))
   C = tl.dot(Q, temp_8).to(tl.float16)
   temp_9 = tl.permute(temp_7, (0, 2, 1))
   temp_10 = tl.load(noise_ptr + offset_7, mask=mask_5, other=0.0)
   C_perturb = ((tl.dot(Q, temp_9).to(tl.float16) + temp_10).to(tl.float16) / 1.5)
   C_exp = tl.exp(C.to(tl.float32))
   tl.store(C_exp_perturb_ptr + offset_8, tl.exp(C_perturb.to(tl.float32)).to(tl.float16),
mask=mask_6)
   C_sum = ((1 * C_sum) + tl.sum(tl.exp(C.to(tl.float32)), axis=2,
dtype=tl.float16)).to(tl.float16)
   C_sum_perturb = ((1 * C_sum_perturb) + tl.sum(tl.exp(C_perturb.to(tl.float32)), axis=2,
dtype=tl.float16)).to(tl.float16)
   temp_11 = tl.load(V_cache_ptr + offset_9, mask=mask_7, other=0.0)
   O = ((O * 1) + tl.dot(C_exp, temp_11.to(tl.float32))).to(tl.float16)
for p in range(0, 1024, BLOCK_P):
   temp_12 = tl.load(C_exp_perturb_ptr + offset_10, mask=mask_8, other=0.0)
   tl.store(C_out_ptr + offset_11, tl.sum((temp_12 / C_sum_perturb[:, :,
None]).to(tl.float16),
axis=1, dtype=tl.float16), mask=mask_9)
O = (O / C_sum[:, :, None])
temp_13 = tl.permute(O, (1, 0, 2))
tl.store(O2_ptr + offset_12, tl.reshape(temp_13, (M, D)).to(tl.float16), mask=mask_10)
```

Figure 15. The highest performance kernel of KeyFormer architecture on NVIDIA RTX4090 GPU generated from Figure 14. Boilerplate code such as offset, mask, and block size are omitted.

A Artifact Appendix

A.1 Abstract

This artifact contains Trinity, a tensor program optimizer that achieves joint optimization through tile-level equality saturation. The artifact includes: (1) the equality saturation engine implemented in Rust using the egg library and (2) a backend that lowers optimized IR to Triton kernels. The artifact enables reproduction of all performance results in the paper.

A.2 Artifact check-list (meta-information)

- **Compilation:** Rust(cargo), Triton JIT compilation
- **Metrics:** Kernel latency (μs), compile time (s), speedup ratio
- **Output:** An optimized GPU kernel file along with numerical latency measurement for that kernel.
- **How much disk space required (approximately)?:** 5GB
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes
- **How much time is needed to complete experiments (approximately)?:** 4~8 hours (depending GPU and number of benchmarks)

A.3 Description

How to access: Clone repository from Github (https://github.com/kaist-ina/Trinity-AE).=

Hardware dependencies: We evaluate both cloud and on-premise GPU platforms, including GCP a3-highgpu-1g (H100 80 GB) and a2-highgpu-1g (A100 40 GB), as well as local RTX 5090 and RTX 4090 servers equipped with Intel Xeon Silver 4210R CPUs and 208 GB system memory.

Software dependencies: Rust 1.75+, Python 3.11+, PyTorch 2.8.0, Triton 3.4.0, TensorRT 10.10.0.31, FlashInfer 0.5.3

A.4 Installation & Basic test

```
sudo apt-get install libz3-dev
cd backend
pip install -r requirements.txt
```

To verify that the artifact is set up correctly, we provide a minimal test command that checks the basic functionality.

```
cd backend
python run_eval.py --o 2 --m llama --t vanilla --n 946
```

If the test is successful, the script will generate a Triton kernel file for the specified IR case without runtime errors.

A.5 Experiment workflow

The workflow of Trinity consists of three main steps: *IR optimization, profiling-based kernel selection, and performance evaluation.* First, Trinity starts from an initial IR. This IR is passed to the optimizer, where equality saturation is applied to explore a large space of semantically equivalent IRs. As a result, the optimizer produces a final list of candidate IR expressions with high performance potential. Second, the generated IR list is passed to the profiling stage. For each IR candidate, the backend generates corresponding Triton GPU code and measures its execution time. Through this profiling process, Trinity identifies the best-performing kernel among all IR candidates. Finally, the selected best kernel is compared against baseline implementations to evaluate its performance. This comparison quantifies the performance benefits achieved by Trinity through IR-level optimization and profiling-based kernel selection.

A.6 Evaluation and expected results

1. **IR Optimization:** In the first stage, equality saturation and cost model are applied to a base IR which is defined in the optimizer test file. You can follow this command to generate the IR candidates:

```
cd optimizer
# compile for all test case
./run_all_optimizer_test.sh
# compile for single test
cargo test --test {test_file} {test_function} -- --nocapture
```

The resulting IR candidate lists are pre-generated and stored in `backend/evaluation/`, organized by architecture. These results are used directly in the kernel generation strategy to ensure reproducibility

2. **Profiling-based Kernel Selection:** In the second stage, `backend` evaluates the IR candidates produced by `optimizer`. Using the IR lists obtained from the first stage, profiling benchmarks are executed to identify the fastest-performing kernel. When the profiling finishes, the top-5 fast IR candidates and their execution times are saved in the `top5.json` file under each architecture directory in `backend/evaluation/`.

```
cd backend
# Profiling for specific architecture
python profile/{method}_{model}_benchmark.py --all
```

3. **Performance Evaluation:** Finally, we measure the latency of the best kernel obtained through profiling and compare it against other baselines. We have pre-stored the best kernels identified through profiling in our environment in the `results` and `figure67` directory. Using these kernels, our experiments for Figures 5 to 7 and Table 2 can be reproduced as follows. We measured latency three times and report the minimum value.

```
cd backend
# Run all benchmarks (figure 5), GPU: 5090, A100, H100
./scripts/evaluate_all.sh {GPU}
# Run for figure 6 & 7
./scripts/evaluate67.sh
```

Depending on your profiling environment, the best IR identified may differ from the one we provided. The procedure for measuring the latency of custom IR is described below. The available command-line options denoted by `[options]` are summarized in Table 3

```
cd backend
# Run for single test case
python run_eval.py [options]
```

Option	Description	Default
`--m`	Model type (`llama`, `falcon`)	`llama`
`--t`	Architecture (`vanilla`, `prenorm`, `qknorm`, `keyformer`, `roco`, `ffn`)	`vanilla`
`--n`	IR case number	N/A
`--o`	0: convert IR to Triton only/ 1: run benchmark only/ 2: convert and run benchmark	2
`--d`	CUDA device number	0
`--baseline`	trinity, tensorrt, pytorch, inductor, flashinfer, flashtensor	All(if not specified)
`--print_output`	Print kernel output values	off

Table 3. Command-line options for single evaluation

By following the provided instructions, the experimental results are expected to reproduce the main findings reported in the paper. In particular, the measured inference latency should be consistent with the trends shown in Figures 5 to 7 and Table 2, demonstrating that Trinity achieves lower latency compared to existing baselines across various Transformer architectures and GPU platforms. Minor numerical differences may occur depending on the hardware environment and system configuration, but the overall relative performance trends should remain consistent.

A.7 Mirage Experiments

Among our baselines, Mirage can be executed as follows.

```
cd backend/mirage_eval
# Install dependency
uv sync && . .venv/bin/activate
# Run for benchmark and remove cache for the next benchmark
python benchmark/mirage/vanilla.py && rm *.json
```