A Multi-Level Superoptimizer for Tensor Programs

Mengdi Wu
Carnegie Mellon University
Pittsburgh, PA, USA
mengdiwu@andrew.cmu.edu

Xinhao Cheng
Carnegie Mellon University
Pittsburgh, PA, USA
xinhaoc@andrew.cmu.edu

Oded Padon
VMware Research
Palo Alto, CA, USA
oded.padon@gmail.com

Zhihao Jia
Carnegie Mellon University
Pittsburgh, PA, USA
zhihao@cmu.edu

Abstract

We introduce Mirage, the first multi-level superoptimizer for tensor programs. A key idea in Mirage is μGraphs, a uniform representation of tensor programs at the kernel, thread block, and thread levels of the GPU compute hierarchy. μGraphs enable Mirage to discover novel optimizations that combine algebraic transformations, schedule transformations, and generation of new custom kernels. To navigate the large search space, Mirage introduces a pruning technique based on abstraction that significantly reduces the search space and provides a certain optimality guarantee. To ensure that the optimized μGraph is equivalent to the input program, Mirage introduces a probabilistic equivalence verification procedure with strong theoretical guarantees. Our evaluation shows that Mirage outperforms existing approaches by up to 3.5× even for DNNs that are widely used and heavily optimized. Mirage is publicly available at https://github.com/mirage-project/mirage.

1 Introduction

Enabling high-performance execution of deep neural networks (DNNs) on GPUs is critical for modern ML applications. Today's DNN frameworks generally specify DNN computation using tensor programs, which are directed acyclic graphs whose nodes and edges represent tensor algebra operators (e.g., matrix multiplications) and tensors (i.e., n-dimensional array) shared between operators.

To optimize an input tensor program, existing frameworks (e.g., PyTorch [31] and TensorFlow [10]) use manually designed rules to map the tensor program to expert-written GPU kernels. These approaches generally require extensive engineering efforts to design and implement optimization rules, and may miss some optimization opportunities. To address these challenges, recent work introduced *automated* approaches to optimizing tensor programs by searching over a comprehensive space of program transformations and applying them based on their performance on target GPUs. These approaches generally fall into two categories.

The first category of work, including Halide [32], TVM [15], and Ansor [45], is motivated by the idea of algorithm and schedule separation[1] introduced in Halide and optimizes the *schedule* of a tensor program while fixing the algorithm. For a given algorithm, these optimizers automatically generate performant kernels by searching over possible strategies for executing the kernel on the target hardware. However, due to the linear algebra nature of DNNs, a tensor program can be represented by a wide spectrum of mathematically equivalent algorithms, and existing schedule-based optimizers only consider kernels whose algorithms are manually specified by users, resulting in missed optimization opportunities.

The second category of work, including TASO [26], Grappler [3], Tensat [43], and PET [41] consider *algebraic transformations*, which exploit mathematical equivalence between different algorithms for a tensor program. Example algebraic transformations include (1) converting a linear algebra operator into another such as a convolution to a matrix multiplication, (2) fusing multiple operators to reduce memory access and kernel overhead, and (3) reordering operators based on commutativity and associativity. These algorithm optimizers perform algebraic transformations at the algorithm level and require programmers to manually define the set of available kernels and their implementations. They are thus limited by the performance of the provided kernels.

All existing automated optimization approaches, from both categories, still require programmers to manually specify a set of kernels (each defined by a tensor function), and then explore the search space of algebraic *or* schedule transformations. However, some advanced performance optimizations require coordinated transformations at the kernel, thread block, and thread levels of the GPU compute hierarchy, and involve introducing completely new kernel computations (e.g., a custom kernel that decomposes standard kernels and fuses only some of their computations). Such optimizations are not part of the search space of existing automated methods and must still be implemented manually.

One such example is FlashAttention [19] (see §3 for details), which optimizes attention [42] on GPUs by reordering operators at the algorithm level (algebraic transformations), reorganizing the computation across GPU kernels (yielding

---

[1]In the schedule optimization literature, an algorithm describes what to compute in a kernel and a schedule specifies how to compute the kernel.
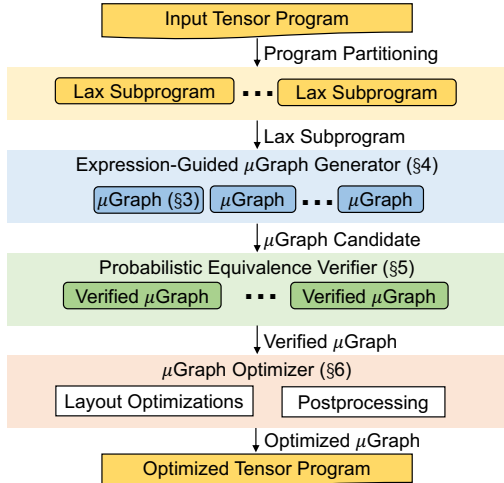
Figure 1. An overview of Mirage.

new custom kernels), and adapting the parallelization strategy of each kernel to the GPU architecture (schedule transformations). The transformations required for this example cannot be automatically discovered by existing frameworks and must therefore be implemented manually. An implementation of FlashAttention in Triton [38], a widely used tensor program optimizer, includes more than 600 lines of code [9].

We present Mirage, the first *multi-level superoptimizer* for tensor programs. Mirage is able to automatically discover and verify sophisticated tensor program optimizations that require joint optimization of algebraic transformations, schedule transformations, and discovery of new custom kernels.

A key idea in Mirage is μGraphs, a *hierarchical graph representation* that specifies a tensor program at multiple levels of the GPU compute hierarchy. By uniformly treating the kernel, thread block, and thread levels, μGraphs can capture both algebraic and schedule transformations. Moreover, optimizing a μGraph can introduce new custom kernels, which goes beyond both algebraic and schedule transformations. For example, Mirage automatically discovers the μGraphs that represent FlashAttention [19] and its inference variant FlashDecoding [5] as well as other μGraphs that outperform these manually designed kernels by up to 3.5× for certain use cases. Most of these Mirage-discovered optimizations are outside of the search space of existing approaches.

Figure 1 shows an overview of Mirage. Mirage first splits an input tensor program into subprograms which fall in the restricted Lax fragment. The Lax fragment, formally defined in § refsec:verify, includes multi-linear operators such as matrix multiplication and convolution, division (which is useful for normalizations), and limited use of exponentiation (which is useful for activations). The partitioning to Lax subprograms reduces the optimization search space while preserving most optimization opportunities; it also enables Mirage's probabilistic equivalence verifier.

Expression-guided μGraph generator. For each Lax subprogram, Mirage's *expression-guided generator* uses exhaustive search to find possible μGraphs that are equivalent to it. A key challenge Mirage has to address is its significantly larger search space compared to prior superoptimization techniques for ML. For example, TASO [26] and PET [41] only search for tensor programs at the kernel level by using a fixed set of pre-defined kernels, while Mirage simultaneously considers superoptimizations at the kernel, thread block, and thread levels. To efficiently navigate this significantly larger and more complex search space, Mirage introduces a novel pruning technique based on *abstract expressions*. This approach greatly reduces the number of μGraphs Mirage has to consider while providing a theoretical guarantee on the optimality of the discovered μGraphs (§ 4.3).

Probabilistic equivalence verifier. For a μGraph discovered by Mirage, verifying its functional equivalence with the input program introduces another challenge, since the input and output tensors of a program include up to many millions of elements. A key idea behind Mirage is *probabilistic equivalence verification*, which performs random tests over finite fields to check equivalence between μGraphs. While random tests can hardly provide any correctness guarantees for general programs, Mirage relies on a novel theoretical result to show that the restrictions of the Lax fragment ensure that for Lax programs, random tests over finite fields offer strong correctness guarantees. We essentially show that an algorithm for polynomial identity testing (PIT) [34, 48] can be generalized to Lax programs, yielding a randomized algorithm for Lax program equivalence that can be made arbitrarily precise. Mirage uses this randomized algorithm to (probabilistically) ensure that the output optimized program is equivalent to the input program.

μGraph optimizer. For each verified μGraph, Mirage's *μGraph optimizer* maximizes its runtime performance by considering potential data layouts for all intermediate tensors at all of the kernel, thread block, and thread levels. Finally, Mirage returns an optimized tensor program based on the best discovered μGraph for each individual Lax subprogram.

Evaluation results. We evaluate Mirage on 12 benchmarks commonly used in today's DNNs, including different variants of attention [11, 35, 42], low-rank adaptation [25], and multi-layer perceptron [24]. Even for DNN benchmarks that are widely used and heavily optimized by existing systems such as the group-query attention used in today's large language models [40], Mirage still outperforms existing systems by up to 3.5×, by exploiting subtle custom kernels and optimizations missing in existing systems.
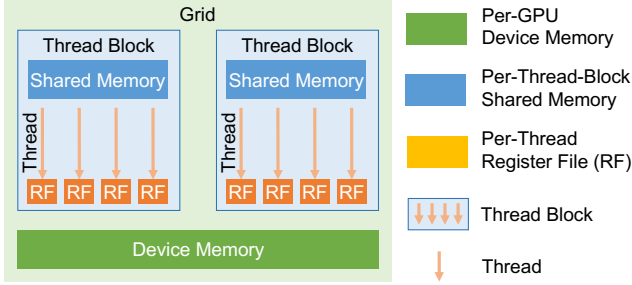
Figure 2. GPU compute and memory hierarchy.

2 Multi-Level Graph Representation

Mirage uses a μGraph to specify the execution of a tensor program on GPUs. A μGraph contains hierarchical graphs at multiple levels to represent computation at the kernel, block, and thread levels[2]. This section first describes the GPU hierarchy and uses Figure 4c as a running example to introduce the key components of a μGraph.

GPU hierarchy. Figure 2 shows the hierarchy of today's GPUs. Computation on GPUs is organized as *kernels*, each of which is a function executed simultaneously on multiple GPU cores in a single-program-multiple-data (SPMD) fashion. A kernel includes a grid of *thread blocks*, each of which is executed on one GPU streaming multiprocessor and includes multiple *threads* to perform computation on individual data elements. Each thread is associated with a per-thread *register file*, and all threads within a thread block can access *shared memory* to enable collective operations. Finally, all inputs and outputs of a kernel are stored in GPU *device memory*.

Kernel graph. Each tensor program corresponds to one *kernel graph*, where each node represents a kernel running on an entire GPU and each edge is a tensor shared between kernels. All tensors in a kernel graph are stored in GPU device memory since different kernels cannot share data in register file or shared memory. Each node in a kernel graph can be a *pre-defined* kernel operator supported by existing kernel libraries such as convolution by cuDNN [17] and matrix multiplication by cuBLAS [18]. In addition, to enable fine-grained inter-kernel optimizations such as kernel fusion, a node in a kernel graph can also be a *graph-defined* kernel operator, whose semantic and behavior are defined by a lower-level (i.e., block) graph. As an example, both kernel operators in Figure 4c are graph-defined operators, each of which is specified by a block graph.

Block graph. A *block* graph specifies computation associated with a thread block[3], where each node denotes a *block operator* which specifies computation within a block

---

[2]For simplicity, we use the term *block* to refer to a thread block of a CUDA kernel and *thread* to refer to a single CUDA thread.

[3]In the CUDA programming model, a kernel's computation is defined as computations for independent thread blocks.

(a) $imap = \{x \leftrightarrow column\}$, $fmap = \{i \leftrightarrow row\}$

(b) $imap = \{x \leftrightarrow row, y \leftrightarrow \phi\}$, $fmap = \{i \leftrightarrow column\}$
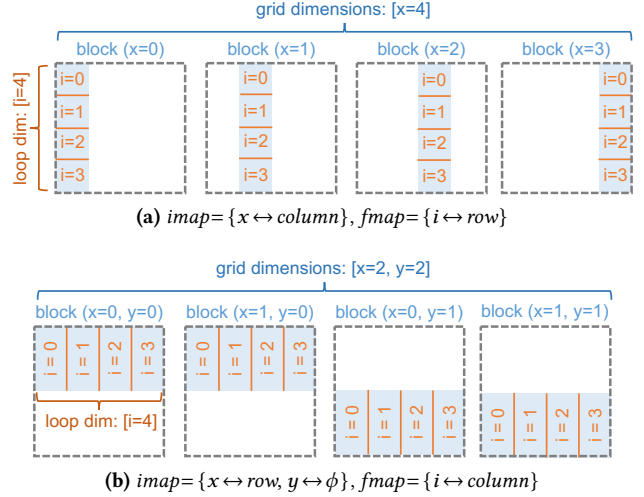
Figure 3. Demonstrating how an input tensor is partitioned across blocks and for-loop iterations with *imap* and *fmap*.

and each edge (blue arrows in Figure 4c) is a tensor shared between block operators. Mirage requires that all intermediate tensors within a block graph are stored in GPU *shared memory* for two considerations. First, GPU shared memory offers much higher bandwidth than device memory, and this design allows Mirage to reduce device memory access by maximally saving intermediate results in shared memory. Second, for tensors whose sizes exceed the shared memory capacity and must be stored in the device memory, Mirage uses these tensors to split computation into multiple block graphs, each of which only contains tensors in shared memory. This separation does not introduce additional access to device memory.

Each block graph is also associated with a few properties to specify its execution, which we introduce as follows.

Grid dimensions. All blocks within a kernel are organized by a mesh with up to 3 dimensions, identified as x, y, and z. Correspondingly, a block graph is associated with up to three *grid dimensions* that specify the number of blocks along the x, y, and z dimensions. The two block graphs in Figure 4c launch 80 (i.e., 8×10) and 64 (i.e., 8×8) blocks.

First, for each input tensor to a graph-defined kernel operator (e.g., Q, K, and V in the kernel graph in Figure 4c), the associated block graph contains an *imap*, which specifies how the input tensor is partitioned into sub-tensors for individual blocks. For each grid dimension (i.e., x, y, or z), the *imap* maps it to (1) a data dimension of the input tensor or (2) a special *replica* dimension ϕ. For (1), the mapped data dimension is *equally partitioned* across blocks along the grid dimension. For (2), the input tensor is *replicated* across these blocks. As an example, block graph 1 in Figure 4c takes three inputs — $\overline{Q}$, $\overline{K}$, and $\overline{V}$, which represent the input tensors to

each block. For $\overline{Q}$, its $imap = \{x \leftrightarrow h, y \leftrightarrow \phi\}$ indicates that the h dimension of tensor Q is partitioned into 8 equally sized chunks, and each of these chucks is replicated 10 times along the y dimension. As a result, each block takes an input tensor $\overline{Q}$ of shape $[h=8, s=1, d=64]$.

Second, for each output tensor of a block graph (e.g., $\overline{A}$ and $\overline{B}$ in Figure 4c), the block graph includes an *omap*, which specifies how the outputs of all blocks are concatenated to construct the final output of the kernel operator (e.g., A and B in Figure 4c). In an *omap*, each grid dimension must map to a data dimension of the output tensor, since different blocks must save to disjoint tensors in device memory. For $\overline{B}$ of shape $[h=1, s=8, d=64]$ in Figure 4c, its $omap = \{x \leftrightarrow h, y \leftrightarrow d\}$ indicates that blocks with the same x index are concatenated along the h dimension and that blocks with the same y index are concatenated along the d dimension, resulting in a tensor B of shape $[h=8, s=8, d=640]$.

For-loop dimensions. To fit large input tensors in shared memory and allow cache reuse, a second property associated with each block graph is *for-loop dimensions*, which collectively specify how many times the block graph is executed to complete a kernel. For example, block graph 1 in Figure 4c has a for-loop dimension $i=20$, indicating it is executed 20 times to finish the associated graph-defined kernel operator. Correspondingly, each input tensor to a block graph is first sent to an *input iterator* that loads a part of the tensor (e.g., $\overline{Q}$, $\overline{K}$, and $\overline{V}$) from device memory to shared memory. Each input iterator is associated with an *fmap* to specify which part of the input tensor to load in each iteration. Formally, the *fmap* maps each for-loop dimension to (1) a data dimension of the input tensor or (2) the replica dimension ϕ. Similar to the semantic of *imap*, the input tensor is equally partitioned along that dimension for (1) and replicated for (2). Figure 3 shows how an input matrix is partitioned across blocks and for-loop iterations with different *imap* and *fmap*.

In addition, a block graph contains *output accumulators* to accumulate its output across iterations in shared memory and save the final results back to device memory. Similar to an input iterator, an output accumulator is also associated with an *fmap* to specify how the output tensors of different iterations are combined to produce the final results. Specifically, the *fmap* maps each for-loop dimension to either a data dimension, which results in concatenation of the output along that dimension, or the replica dimension ϕ, which results in the output being accumulated in shared memory.

Thread graph. A *thread* graph further reduces the scope of computation from a block to a single thread. Similar to a block graph, each thread graph is also associated with *block dimensions*, which specify the organization of threads within the block, and *for-loop dimensions*, which define the total number of iterations to finish the defined computation. Each thread graph includes *input iterators*, each of which loads an

input tensor (e.g., $\overline{\overline{C}}$ in Figure 4c) from GPU shared memory to register file, and *output accumulators*, each of which saves an output tensor from register file back to shared memory (e.g., $\overline{\overline{D}}$ and $\overline{\overline{E}}$). A thread graph is the lowest level graph in a μGraph and only contains pre-defined thread operators.

Tensor layout. Each tensor in the kernel, block, or thread graph is associated with a *tensor layout* (omitted in Figure 4 for simplicity), which specifies how the tensor is linearized in memory. Note that tensor layout only affects the execution performance of a μGraph and has no impact on its output.

Comparison with prior work. Prior work separately considers algebraic [26, 41] or schedule transformations [15, 30, 32], while μGraphs can represent both in a uniform way. Specifically, the grid and for-loop dimensions and their corresponding mappings (i.e., *imap*, *omap*, and *fmap*) to tensor dimensions constitute a comprehensive search space of possible schedules for graph-defined operators. The hierarchical graphs at the kernel, block, and thread levels allow Mirage to explore algebraic transformations at these levels.
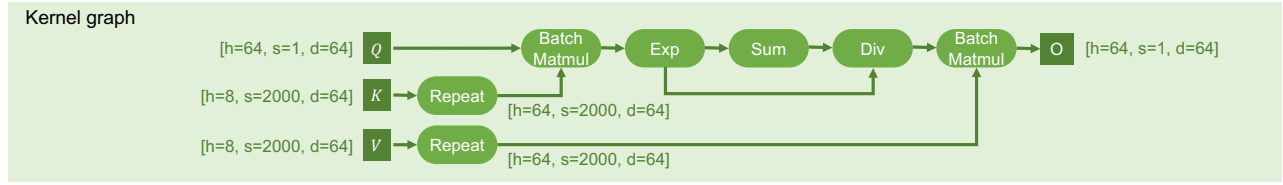
3 Case Study: Group-Query Attention

In this section, we use group-query attention [11] as a case study to demonstrate the advantages of the μGraph representation and Mirage's superoptimization approach. Group-query attention (GQA) reduces the memory requirement of storing keys and values compared to conventional multi-head attention [42], and has been widely used in recent large language models such as LLAMA-2-70B [40]. GQA allows a subset of query heads to share the same key and value head when computing attention. Formally, GQA takes a query tensor Q, a key tensor K, and a value tensor V as inputs and computes an output tensor O via scaled multiplicative formulations:

$$A_i = \frac{1}{\sqrt{d}}(Q_i \times K_{\lfloor i/g \rfloor}), H_i = \text{softmax}(A_i), O_i = H_i \times V_{\lfloor i/g \rfloor} \quad (1)$$

where d is the hidden dimension size (a constant), Q_i is the tensor for the i-th query head ($0 \leq i < h$), K_j and V_j denote the key and value tensors of the j-th key-value head ($0 \leq j < h/g$), and g is the number of subgroups in GQA.

Figure 4a shows the computation graph of GQA, where Q, K, and V denote the three input tensors and numbers in bracket indicate their shapes. We use h, s, and d to refer to the head, sequence, and hidden dimensions of the tensors. The computation graph shows incremental decoding of GQA, where a single query token (i.e., $s=1$ for Q) attends to 2000 previous tokens to compute the attention output. To batch the first and last matrix multiplication in GQA, existing frameworks (logically) replicate the key and value tensors to match the head dimension of the query tensor. The Exp, Sum, and Div operators collectively compute softmax.

To optimize memory access, FlashAttention [19] and its inference variant FlashDecoding [5] apply a manually designed

(a) Computation graph for group-query attention.
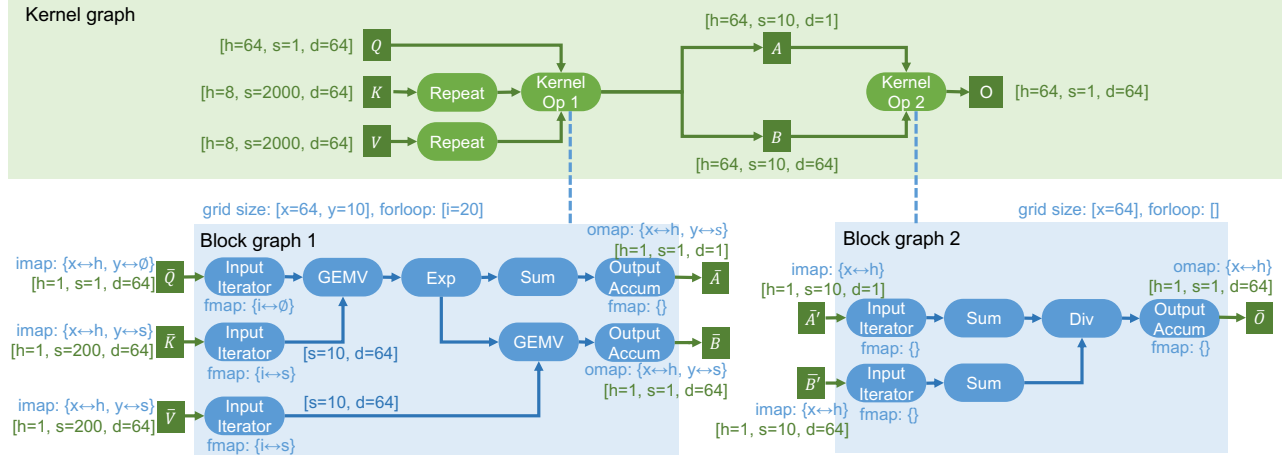
(b) Implementation of group-query attention in FlashDecoding [5] using two kernels. The first kernel computes two BatchMatmuls, Exp, and partial Sum, and accumulates intermediate results in A and B, while the second kernel concludes the Sum and computes Div.
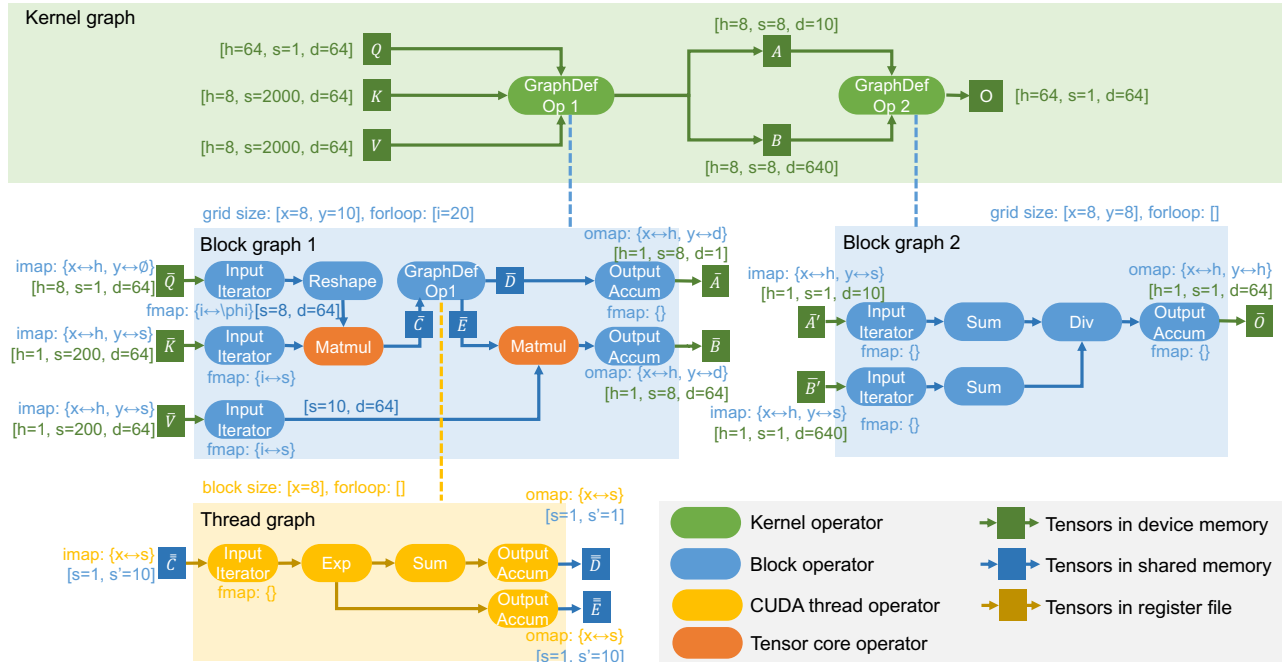
(c) An optimized μGraph discovered by Mirage for group-query attention.

Figure 4. μGraph examples: Figure 4a is the computation graph for group-query attention (GQA). Figure 4b is the μGraph for FlashDecoding, an expert-designed implementation of GQA [19]. Figure 4c demonstrates an optimized μGraph discovered by Mirage, which further improves over FlashDecoding by reducing device memory access and reorganizing computation to leverage tensor cores available on modern GPUs, which outperforms FlashDecoding by 2.2×. Numbers in brackets indicate tensor shapes, and numbers in braces show the *imap*, *omap*, or *fmap* for the corresponding operators.

5

strategy to maximally reuse shared memory, represented by the μGraph in Figure 4b. It reorders the final `BatchMatmul` with the `Div` and partitions the `Sum` across multiple blocks, which splits the computation associated with different query heads to individual blocks. An issue with FlashDecoding is that each block performs matrix-vector multiplication (GEMV) since the query tensor only includes a single token, so it does not benefit from tensor cores available on modern GPUs, which accelerates matrix-matrix multiplication.

Figure 4c shows the best μGraph automatically discovered by Mirage for computing GQA. The computation is split into two graph-defined kernel operators with different grid and loop dimensions to reduce device memory access. Next, we highlight the key differences between the μGraph discovered by Mirage and FlashDecoding. These differences involve discovering new custom kernels and combining algebraic and schedule transformations, making it infeasible to discover the final μGraph by separately considering algebraic and schedule transformations even when given the FlashDecoding μGraph as the starting point. First, Mirage does not use `Repeat` operators to replicate the key and value tensors (algebraic transformation) and instead processes a key-value head together with 8 query heads in a block (schedule transformation). Second, within a block, Mirage introduces a `Reshape` operator to reorganize the query tensor's shape from $[h=8, s=1, d=64]$ to $[h=1, s=8, d=64]$ (algebraic transformation), which can be considered as concatenating the token of each query heads into eight tokens of a single query head. This reorganization allows Mirage to perform matrix-matrix multiplications (i.e., GEMM) using tensor cores within each block. The `Reshape` also results in different tensor shapes for intermediate results A and B (schedule transformation). Finally, the Mirage-discovered μGraph uses different schedules and data layouts than FlashDecoding for processing the second block graph while resulting in the same final output. This μGraph outperforms FlashDecoding by 2.2× on NVIDIA A100 GPUs. In addition to the optimizations studied in this section, Mirage automatically discovers additional optimizations for attention and other DNN workloads, which we discuss in §7.

4 Expression-Guided μGraph Generator

This section introduces the Mirage μGraph generator, which automatically discovers potential μGraphs for an input tensor program. To generate μGraphs that capture optimizations at all of the kernel, block, and thread levels, Mirage must explore a significantly larger search space than existing superoptimizers that only consider optimizations at the kernel level. Mirage employs two key techniques to address this challenge. First, based on an important observation that optimizations at the kernel and block levels are substantially more critical to performance than optimizations at the thread level since accessing device and shared memory is orders
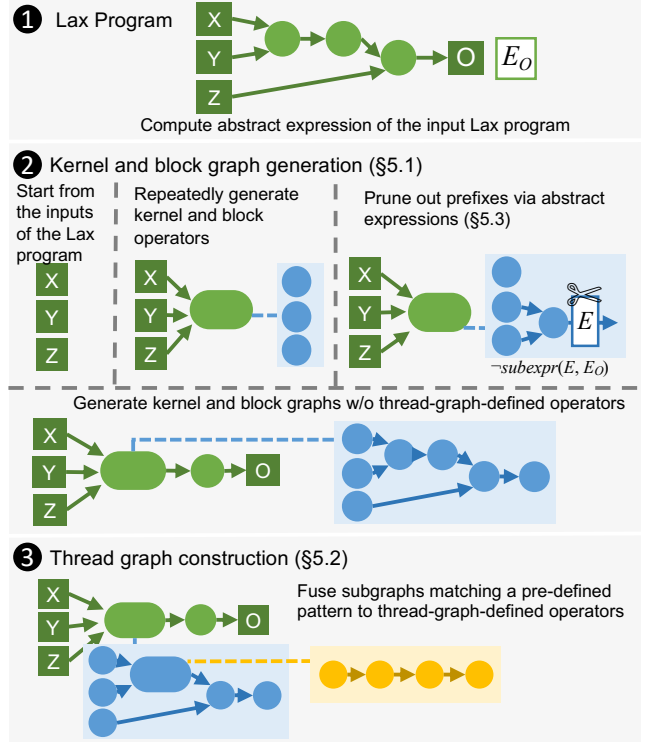

Figure 5. An overview of μGraph generator.

of magnitude more expensive than accessing register file, Mirage's μGraph generator employs a *hybrid approach*, considering all possible graphs up to a certain size at the kernel and block levels, and using a rule-based strategy to construct graphs at the thread level, which reduces the search space while retaining most performance optimizations. Second, to further prune the search space, Mirage introduces a pruning technique based on an abstraction of μGraphs called *abstract expression*, which reduces the number of μGraphs Mirage has to consider while providing a theoretical guarantee on the optimality of the discovered μGraphs.

4.1 Kernel and Block Graph Generation

Mirage generates kernel and block graphs incrementally and leverages several pruning techniques to reduce the search space, as shown in the second part of Figure 5. Specifically, Mirage maintains a *prefix* of a valid μGraph, iteratively extending it with new operators. Here, prefix G' of G is defined as a subgraph of G such that $\forall u \in G', \forall (v, u) \in G, v \in G'$. Mirage generates the next operator in the kernel graph by enumerating the kernel operator type t and the input tensors I. If t stands for the graph-defined operator type, Mirage also incrementally generates the underlying block graph that defines its kernel computation. To generate a block graph, Mirage first enumerates the grid and for-loop dimensions

Algorithm 1 Mirage's hybrid μGraph generation algorithm.

Input: A Lax program with a computation graph G_{ref}
Output: A set of μGraphs $\mathcal{S}$
 1: $E_O \leftarrow E(G_{\text{ref}})$
 2: $\mathcal{S}_0, \mathcal{S} \leftarrow \varnothing, G_K \leftarrow \text{Inputs}(G_{\text{ref}})$
 3: GENERATENEXTKERNELOPERATOR(G_K)
 4: **for all** $G \in \mathcal{S}_0$ **do**
 5:   $\mathcal{S} \leftarrow \mathcal{S} \cup \{\text{THREADGRAPHCONSTRUCTION}(G)\}$
 6: **function** GENERATENEXTKERNELOPERATOR(G_K)
 7:   $\mathcal{S}_0 \leftarrow \mathcal{S}_0 \cup \{G_K\}$
 8:   **for all** kernel graph op type t; input set I **do**
 9:     **if** $(I, t) > (op.I, op.t)$ for each $op \in G_K$ **then**
10:       **if** t is a pre-defined operator **then**
11:         **if** $o := \text{CONSTRUCTOP}(G_K, I, t)$ is valid **then**
12:           GENERATENEXTKERNELOPERATOR($G_K \cup \{o\}$)
13:       **else**   ▷ t is a graph-defined operator
14:         **for all** *gridDims*; *forloopDims* **do**
15:           $G_B \leftarrow \text{TBGraph}(I, gridDimd, forloopDims)$
16:           GENERATENEXTBLOCKOPERATOR(G_K, G_B)
17: **function** GENERATENEXTBLOCKOPERATOR(G_K, G_B)
18:   **if** all shared tensors in G_B are consumed **then**
19:     **if** $o := \text{CONSTRUCTOP}(G_K, G_B.I, G_B)$ is valid **then**
20:       GENERATENEXTKERNELOPERATOR($G_K \cup \{o\}$)
21:   **for all** block graph op type t; input set I **do**
22:     **if** $(I, t) > (op.I, op.t)$ for each $op \in G_B$ **then**
23:       **if** $o := \text{CONSTRUCTOP}(G_B, I, t)$ is valid **then**
24:         GENERATENEXTBLOCKOPERATOR($G_K, G_B \cup \{o\}$)
25: **function** CONSTRUCTOP($G, I, attrs$)
26:   $E \leftarrow \text{EXPRINFR}(E(I), attrs)$ ▷ Table 1
27:   **if** SUBEXPR(E, E_O) **then**
28:     $S \leftarrow G.\text{outputTensorShapeInfr}(I, attrs)$ ▷ Tensor shape
29:     **if** S.valid, $G.\text{mAlloc} + S.\text{size} \leq G.\text{mLimit}$ **then** ▷ Memory
30:       **return** $G.\text{constructOp}(I, attrs)$
31:   **return** Invalid
32: **function** THREADGRAPHCONSTRUCTION(G)
33:   $\mathcal{P} \leftarrow$ pre-defined patterns
34:   $G_{\text{fused}} \leftarrow G$
35:   **for all** $(G_i, O_i) \in \mathcal{P}$ **do**
36:     **for all** subgraph G' of G matching G_i **do**
37:       Substitute G' with O_i in G_{fused}
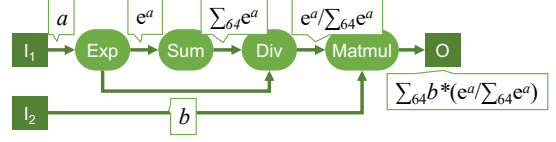38:   **return** G_{fused}

Figure 6. Illustration of abstract expressions. The abstract expressions of tensors are annotated on edges. A human-friendly notation is used here: e^a denotes $\exp(a)$, $\sum_k a$ denotes $\text{sum}(k, a)$, a/b denotes $\text{div}(a, b)$, and $a * b$ denotes $\text{mul}(a, b)$. The tensors I_1, I_2 and O are all 64×64 matrices.

Table 1. Operators supported by Mirage. The second column shows the levels of graphs supporting the operator (K, B and T stand for kernel, block and thread graphs, respectively). The last column defines the abstract expressions of the outputs of each operator. E is the mapping from tensors to the corresponding abstract expressions.

μGraph Operator	Graph Level	Abstract Expression of Output Tensor
InIter	B	$E(\text{InIter}(X)) = E(X)$
OutAccu	B	$E(\text{OutAccu}(k_f, m_f, X)) = \text{sum}(k_f, E(X))$ if $m_f = \phi$ else $E(X)$[1]
Matmul	K, B, T	$E(\text{Matmul}(X, Y)) = \text{sum}(k, \text{mul}(E(X), E(Y)))$[2]
Sum	K, B, T	$E(\text{Sum}(d_r, k_r, X)) = \text{sum}(k_r, E(X))$[3]
EwAdd	K, B, T	$E(\text{EwAdd}(X, Y)) = \text{add}(E(X), E(Y))$
EwMul	K, B, T	$E(\text{EwMul}(X, Y)) = \text{mul}(E(X), E(Y))$
EwDiv	K, B, T	$E(\text{EwDiv}(X, Y)) = \text{div}(E(X), E(Y))$
EwExp	K, B, T	$E(\text{EwExp}(X)) = \exp(E(X))$
Repeat	K, B	$E(\text{Repeat}(X)) = E(X)$
Reshape	K, B	$E(\text{Reshape}(X)) = E(X)$

[1] k_f is the for-loop dimension; m_f is *fmap*.
[2] k means the size of the last dimension of A, i.e., the reduction dimension. Matmul is performed on the inner most two dimensions and leading dimensions are batched.
[3] Sum along the dimension d_r for every k_r elements.

(introduced in §2), enabling Mirage to calculate the input tensor shapes of the block graph. Mirage then performs a nested generation similar to that at the kernel level but without considering graph-defined operators. Line 6-16 and line 17-24 in Algorithm 1 show how Mirage generates kernel and block operators, respectively.

Mirage checks tensor shape (line 28) and memory usage (line 29) before adding an operator, ensuring a valid prefix. A prefix passes the memory usage check if: (1) all tensors in the kernel graph can reside in device memory; and (2) all tensors in each block graph can fit in shared memory.

To ensure identical μGraphs are generated only once, Mirage defines the *canonical form* of μGraphs. Given a kernel or block graph G with its operators in topological order $o_1, \ldots, o_n$, the index of the j-th output of o_i is a tuple (i, j). Each operator o_i in G is assigned a rank (I_i, t_i), where I_i is

the list of input tensor indices of o_i and t_i is the operator type. A μGraph is in canonical form if, in all its kernel and block graphs, the operators are in the increasing order of ranks. Mirage generates only μGraphs in canonical form by requiring that operators are added in the increasing order of ranks (line 9 and line 22). This approach does not prune out any potential solutions, since each μGraph can be transformed to canonical form by reordering the operators.

In addition, Mirage utilizes the *abstract expression* technique to prune out prefixes that do not satisfy certain constraints, which will be introduced in §4.3.

4.2 Thread Graph Construction

Mirage constructs thread graphs in a way similar to operator fusion, as shown in the third part of Figure 5 and line 4-line 5 in Algorithm 1. Specifically, Mirage has a set of pre-defined patterns $\{(G_i, O_i)\}$ where each G_i is a graph consisting of block operators and O_i is a thread graph. Given a μGraph, Mirage traverses all its block graphs, replacing any subgraphs that match G_i with an operator defined by O_i.

Table 2. Axiomatization of abstract expressions used for pruning. Mirage checks whether an abstract expression E_1 is a subexpression of E_2 by querying an SMT solver to check if subexpr(E_1, E_2) is entailed by these axioms.

Abstract Expression Property	Comment
Equivalence Axioms A_{eq}	
$\forall x, y.\ \mathrm{add}(x, y) = \mathrm{add}(y, x)$	commutativity
$\forall x, y.\ \mathrm{mul}(x, y) = \mathrm{mul}(y, x)$	commutativity
$\forall x, y, z.\ \mathrm{add}(x, \mathrm{add}(y, z)) = \mathrm{add}(\mathrm{add}(x, y), z)$	associativity
$\forall x, y, z.\ \mathrm{mul}(x, \mathrm{mul}(y, z)) = \mathrm{mul}(\mathrm{mul}(x, y), z)$	associativity
$\forall x, y, z.\ \mathrm{add}(\mathrm{mul}(x, z), \mathrm{mul}(y, z)) = \mathrm{mul}(\mathrm{add}(x, y), z)$	distributivity
$\forall x, y, z.\ \mathrm{add}(\mathrm{div}(x, z), \mathrm{div}(y, z)) = \mathrm{div}(\mathrm{add}(x, y), z)$	associativity
$\forall x, y, z.\ \mathrm{mul}(x, \mathrm{div}(y, z)) = \mathrm{div}(\mathrm{mul}(x, y), z)$	associativity
$\forall x, y, z.\ \mathrm{div}(\mathrm{div}(x, y), z) = \mathrm{div}(x, \mathrm{mul}(y, z))$	associativity
$\forall x.\ x = \mathrm{sum}(1, x)$	identity reduction
$\forall x, i, j.\ \mathrm{sum}(i, \mathrm{sum}(j, x)) = \mathrm{sum}(i * j, x)$	associativity
$\forall x, y, i.\ \mathrm{sum}(i, \mathrm{add}(x, y)) = \mathrm{add}(\mathrm{sum}(i, x), \mathrm{sum}(i, y))$	associativity
$\forall x, y, i.\ \mathrm{sum}(i, \mathrm{mul}(x, y)) = \mathrm{mul}(\mathrm{sum}(i, x), y)$	distributivity
$\forall x, y, i.\ \mathrm{sum}(i, \mathrm{div}(x, y)) = \mathrm{div}(\mathrm{sum}(i, x), y)$	distributivity
Subexpression Axioms A_{sub}	
$\forall x, y.\ \mathrm{subexpr}(x, \mathrm{add}(x, y))$	
$\forall x, y.\ \mathrm{subexpr}(x, \mathrm{mul}(x, y))$	
$\forall x, y.\ \mathrm{subexpr}(x, \mathrm{div}(x, y))$	
$\forall x, y.\ \mathrm{subexpr}(y, \mathrm{div}(x, y))$	
$\forall x.\ \mathrm{subexpr}(x, \exp(x))$	
$\forall x, i.\ \mathrm{subexpr}(x, \mathrm{sum}(i, x))$	
$\forall x.\ \mathrm{subexpr}(x, x)$	reflexivity
$\forall x, y, z.\ \mathrm{subexpr}(x, y) \wedge \mathrm{subexpr}(y, z) \rightarrow \mathrm{subexpr}(x, z)$	transitivity

4.3 Pruning via Abstract Expressions

When searching the space of possible μGraphs, we aim to avoid μGraph prefixes whose intermediate results cannot contribute to the desired computation. For example, for the input program $X \cdot Z + Y \cdot Z$, we can prune a prefix that computes $X \cdot Y$, but we should not prune one that computes $X + Y$, as $(X + Y) \cdot Z$ is equivalent to the input program. However, how can we determine whether a prefix can contribute to a desired computation while searching for that computation? Below, we develop a pruning technique driven by this intuition that circumvents the "chicken and egg" problem via *abstraction*. First, we present the abstraction — *abstract expressions* — and then explain how it is used for pruning. Finally, we offer a theoretical guarantee that under certain conditions, this pruning does not exclude the optimal μGraph.

Abstract expressions. Recall that each edge in a μGraph corresponds to a tensor-valued function of the input tensors. Intuitively, abstract expressions abstract these functions by ignoring the differences between elements from the same input tensor. Formally, abstract expressions are first-order logic terms over the theory of integers and uninterpreted functions. In a μGraph, the abstract expression of each edge, denoted by $\mathrm{E}(\cdot)$, is defined as shown in Table 1. During the computation of a μGraph's abstract expression, all graph-defined operators are "inlined". Specifically, the expressions

computed for a graph-defined operator's inputs serve as inputs to the lower-level graph, and the expressions for the outputs of this lower-level graph become the output expressions of the graph-defined operator. Figure 6 shows the abstract expressions for a subgraph of attention.

While abstract expressions capture some information about the function computed at each edge, they also abstract away much of it. For example, if X is a $k \times k$ matrix, summing over the rows and summing over the columns lead to the same abstract expression—$\mathrm{sum}(k, \mathrm{E}(X))$. But keeping k as part of the abstract expression is crucial for effective pruning.

Abstract subexpression and pruning. We use abstract expressions to prune the search space of μGraphs by formalizing two relations over abstract expressions: equivalence and abstract subexpression. We then prune every μGraph prefix whose abstract expression is not a subexpression of some abstract expression that is equivalent to that of the input program. We formalize abstract expressions as uninterpreted functions in first-order logic over the theory of integer arithmetic and uninterpreted functions, and use an SMT solver to reason about them based on two sets of axioms defined in Table 2: A_{eq} and A_{sub}. A_{eq} axiomatizes equivalence between abstract expressions. As will become clear below, these axioms need not be sound—it is not necessary that μGraphs whose abstract expressions are equivalent are actually equivalent. (As mentioned earlier, non-equivalent μGraphs can have identical abstract expressions.) A_{sub} axiomatizes the abstract subexpression relation between abstract expressions. The key property of this relation is that whenever a μGraph G_1 is a prefix of G_2, meaning G_2 can be obtained from G_1 by adding more operators, then $\mathrm{E}(G_1)$ is an abstract subexpression of $\mathrm{E}(G_2)$, i.e., $A_{\mathrm{sub}} \models \mathrm{subexpr}(\mathrm{E}(G_1), \mathrm{E}(G_2))$, where $\models$ denotes entailment modulo the theory of integer arithmetic and uninterpreted functions.

During the search, Algorithm 1 first computes the abstract expression of the input LAX program, E_O, and prunes away any μGraph prefix G if $A_{\mathrm{eq}} \cup A_{\mathrm{sub}} \not\models \mathrm{subexpr}(\mathrm{E}(G), E_O)$. That is, we prune a graph if its abstract expression is not a subexpression of E_O. The check is performed by invoking an SMT solver (Z3 [20]). As an optimization, check results are cached and reused, since during the search Mirage may encounter multiple μGraphs with identical abstract expressions and SMT queries are relatively expensive.

Theoretical guarantee and the pruning-optimality tradeoff. Intuitively, our pruning would keep any prefix that can lead to a μGraph whose abstract expression is equivalent (according to A_{eq}) to that of the input LAX program. Formally:

Theorem 1 (Pruning via Abstract Expressions). *For an input μGraph G_0, and a μGraph G equivalent to G_0, if $A_{eq} \models E(G_0) = E(G)$ then G will be generated by Algorithm 1.*

Proof. By Tables 1 and 2, we show that for any operator op, if $Y = op(X_1, \ldots, X_n)$ then $A_{\mathrm{sub}} \models \mathrm{subexpr}(\mathrm{E}(X_i), \mathrm{E}(Y))$

(for $1 \le i \le n$). That is, an input to an operator is always an abstract subexpression of its output. By the reflexivity and transitivity axioms included in A_{sub}, it follows for any G' that is a prefix of G, $A_{\mathrm{sub}} \models \mathrm{subexpr}(\mathrm{E}(G'), \mathrm{E}(G))$. Together with the assumption that $A_{\mathrm{eq}} \models \mathrm{E}(G_0) = \mathrm{E}(G)$, it follows that $A_{\mathrm{eq}} \cup A_{\mathrm{sub}} \models \mathrm{subexpr}(\mathrm{E}(G'), \mathrm{E}(G_0))$. Thus, any prefix of G will not be pruned, and G will be generated by Mirage. □

The theorem highlights the role of abstract expressions in solving the "chicken and egg" problem outlined above. To decide if a prefix μGraph is useful, we reason about whether it is a prefix of a useful computation *in the abstract*. The choice of the abstraction and of the axioms A_{eq} represents a tradeoff between optimality and pruning. As Theorem 1 shows, we are only guaranteed to find the optimal μGraph such that A_{eq} imply equivalence of its abstract expression to that of the input program. The stronger the axioms, the theorem covers more μGraphs, but we also get less pruning because more prefixes would pass the subexpression test. In particular, note that A_{eq} does not consider cancellation (e.g., $\mathrm{div}(\mathrm{mul}(x, y), y) = y$). As a result, Mirage may miss some equivalent μGraphs. But including an axiom for cancellation of division and multiplication would make everything a subexpression of everything, therefore nulling the desired pruning. As our evaluation shows, our choice of A_{eq} yields a good balance between pruning and optimality.

5 Probabilistic Equivalence Verifier

Mirage's *probabilistic equivalence verifier* checks if a candidate μGraph is equivalent to the desired LAX subprogram. The key idea is to evaluate both on *random inputs* in two finite fields. Using finite fields instead of floating point numbers not only avoids floating point errors, but also leads to a strong theoretical guarantee: the probability of accepting a non-equivalent μGraph can be made arbitrarily low.

For general programs, random tests can hardly provide any correctness guarantees. However, we show that for LAX programs (formally defined below), random testing provides a probabilistic correctness guarantee, and repeated tests can reduce the error probability to an arbitrarily small threshold.

Prior work [41] has applied a similar technique to check equivalence between tensor programs that only contain linear operators (e.g., matrix multiplication, convolution). We develop a random testing technique that also supports division and exponentiation, which are needed for many DNN optimizations (e.g., the attention optimization in §3).

Mirage verifies equivalence between LAX μGraphs (<u>l</u>inear, <u>a</u>ivision, and <u>an</u> <u>ex</u>ponential) defined below. We introduce the main theoretical results in §5.1 and present Mirage's verification methodology in §5.2.

Definition 2 (LAX μGraph). A μGraph G is a LAX μGraph if (1) G contains only multi-linear operators[4], division, and exponentiation, and (2) every path from an input to an output in G includes at most one exponentiation.

5.1 Theoretical Foundations

Without loss of generality, we assume a LAX μGraph G takes n input tensors and produces one output tensor. Our theoretical results can directly generalize to LAX μGraph with multiple outputs. Since each LAX μGraph includes linear operators, divisions, and at most one exponentiation along each path, the computation for each entry of the output tensor can be expressed in the following form (by using standard identities such as $\frac{a}{b} \cdot \frac{c}{d} = \frac{ad}{bc}$, $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$, $e^x e^y = e^{x+y}$):

$$\sum_{i=1}^{k} \frac{f_i}{g_i} e^{\left(\frac{h_i}{u_i}\right)}, \tag{2}$$

where f_i, g_i, h_i, and u_i ($1 \le i \le k$) are polynomials over the entries of the input tensors.

The main theoretical result that underpins our randomized equivalence verification is the following theorem, which extends polynomial identity testing (PIT) [34, 48] on finite fields to LAX μGraphs. Note that the difference of two LAX μGraphs is also of the form of Equation (2). Therefore, identity testing of two LAX μGraphs can be done by testing if an expression of that form is zero. Because of the use of exponentiation, we use two finite fields instead of one.[5]

Theorem 3. *Let* $\mathbb{Z}_p, \mathbb{Z}_q$ *be the finite field of integers modulo* p *and* q, *respectively, where* p, q *are primes such that* q *divides* $p-1$. *Let* $d, k \in \mathbb{N}$ *s.t.* $dk^4 < q$. *Let* $f_1, \ldots, f_k, g_1, \ldots, g_k : \mathbb{Z}_p^N \to \mathbb{Z}_p$ *be polynomials over* $\mathbb{Z}_p$ *and* $h_1, \ldots, h_k, u_1, \ldots, u_k : \mathbb{Z}_q^M \to \mathbb{Z}_q$ *be polynomials over* $\mathbb{Z}_q$, *where the degrees of all polynomials are at most* d. *Let* $\mathcal{G}$ *be the set of* q-*th roots of unity in* $\mathbb{Z}_p$. *If* $h_i/u_i \not\equiv h_j/u_j$ *for all* $i \ne j$, *and* $f_i, g_i, h_i \ne 0$ *for all* i, *then*

$$\Pr_{\mathbf{x} \leftarrow \mathbb{Z}_p^N, \mathbf{y} \leftarrow \mathbb{Z}_q^M, \omega \leftarrow \mathcal{G}} \left[\sum_{i=1}^{k} \frac{f_i(\mathbf{x})}{g_i(\mathbf{x})} \omega^{\frac{h_i(\mathbf{y})}{u_i(\mathbf{y})}} = 0 \wedge \mathcal{E} \right] \le 1 - \frac{1}{k} + o\left(\frac{1}{k}\right),$$

where $\mathcal{E}$ *is the event that* $g_i(\mathbf{x}), u_i(\mathbf{y}) \ne 0$ *for all* i.

5.2 Random Tests over Finite Fields

Mirage leverages Theorem 3 to probabilistically verify the equivalence of two μGraphs by performing random testing over the finite fields $\mathbb{Z}_p$ and $\mathbb{Z}_q$ as defined in Theorem 3. To check the equivalence of two μGraphs, Mirage first generates the input tensors, where each entry is uniformly sampled from $\mathbb{Z}_p \times \mathbb{Z}_q$. Mirage also samples ω, which is used for

---

[4]An operator op with n inputs is multi-linear if op is linear to all inputs I_k:
(1) $\forall X, Y . op(I_1, ..., I_{k-1}, X, I_{k+1}, ..., I_n) + op(I_1, ..., I_{k-1}, Y, I_{k+1}, ..., I_n) = op(I_1, ..., I_{k-1}, X + Y, I_{k+1}, ..., I_n)$, and
(2) $\alpha \cdot op(I_1, ..., I_{k-1}, X, I_{k+1}, ..., I_n) = op(I_1, ..., I_{k-1}, \alpha \cdot X, I_{k+1}, ..., I_n)$.
[5]We use two primes p and q for polynomial identity testing [34, 48] outside and inside the exponents, respectively. The condition q divides $p - 1$ is to ensure the existence of q-th roots of unity in $\mathbb{Z}_p$.

Table 3. Arithmetic operations for random testing. Mirage selects two prime numbers p and q such that q divides $p - 1$. x_p and x_q are values from $\mathbb{Z}_p$ and $\mathbb{Z}_q$, respectively.

Opt.	Opd. 1	Opd. 2	Output
Add.	(x_p, x_q)	(y_p, y_q)	$(x_p + y_p \,(\mathrm{mod}\,p), x_q + y_q \,(\mathrm{mod}\,q))$
Sub.	(x_p, x_q)	(y_p, y_q)	$(x_p - y_p \,(\mathrm{mod}\,p), x_q - y_q \,(\mathrm{mod}\,q))$
Mul.	(x_p, x_q)	(y_p, y_q)	$(x_p y_p \,(\mathrm{mod}\,p), x_q y_q \,(\mathrm{mod}\,q))$
Div.	(x_p, x_q)	(y_p, y_q)	$(x_p y_p^{-1} \,(\mathrm{mod}\,p), x_q y_q^{-1} \,(\mathrm{mod}\,q))$
Exp.	(x_p, x_q)	-	$(\omega^{x_q} \,(\mathrm{mod}\,p), -)$

Table 4. DNN benchmarks used in our evaluation.

Name	Description	Base Architecture
MHA	Multi-head attention (3 modes)	LLaMA-7B [39]
GQA	Group-query attention (3 modes)	LLaMA-2-70B [40]
MQA	Multi-query attention (3 modes)	Falcon-7B [12]
MLP	Multi-layer perceptron	Adapter Tuning [24]
MoE	Mixture-of-experts	Mixtral-7B [27]
LoRA	Low-rank adaptation	LLaMA-7B-LoRA [6]

exponentiation, uniformly from all the q-roots of unity in $\mathbb{Z}_p$. Mirage then evaluates the two μGraphs over the finite fields for the same inputs using the operations defined in Table 3. As explained in §5.1, $\mathbb{Z}_p$ and $\mathbb{Z}_q$ are used outside and inside the exponent, respectively. All operations except exponentiation are implemented via modular arithmetic on $\mathbb{Z}_p$ and $\mathbb{Z}_q$ separately. Exponentiation uses the value x_q from $\mathbb{Z}_q$ and transforms it to a value in $\mathbb{Z}_p$ by computing $\omega^{x_q} (\bmod p)$. Note that in a LAX μGraph, exponentiation will be performed at most once along each path. Finally, Mirage checks whether the two μGraphs produce identical outputs. This process is repeated multiple times, and the two μGraphs pass the equivalence test if they pass all random tests. The following theorem, which follows from Theorem 3, shows that this process can yield an arbitrarily low error rate.

Theorem 4. *Equivalent μGraphs always pass μGraph verification. For two non-equivalent μGraphs and a given probability threshold $0 < \delta \le 1$, the μGraphs pass all $\Omega(k \cdot \ln \frac{1}{\delta})$ random tests with probability at most δ.*

6 μGraph Optimizer

For each verified μGraph, Mirage's *μGraph optimizer* maximizes its runtime performance by considering possible data layouts for all intermediate tensors at the kernel, block, and thread levels. Mirage defers layout optimizations after verification to enable two benefits. First, data layouts do not affect the correctness of a generated μGraph, and omitting layouts when generating μGraphs reduces the search space Mirage must consider since μGraphs with the same graph topology and different data layouts are considered identical by the μGraph generator. Second, optimizing layout after verification minimizes the layout optimization workload and allows Mirage to explore all possible layout combinations.

For each μGraph, the optimizer enumerates all supported layouts for each intermediate tensor of the μGraph at the kernel, block, and thread levels, profiles the performance of the final kernels on target hardware, and selects the layouts that yield the best performance for the μGraph. Finally, Mirage selects the best discovered μGraph as the output program.

7 Evaluation

7.1 Implementation

Mirage is implemented in 13K lines of C++ and CUDA code. Kernel operators are implemented with the cuDNN and cuBLAS libraries [17, 18], and block and thread operators are implemented using cuTLASS [2] and CUDA functions. Mirage uses Z3 4.12.6 as the SMT solver [20].

Our implementation supports the operators listed in Table 1. Mirage can be extended to include new linear operators such as variants of convolution or matrix multiplication at the kernel, block, and/or thread levels. To support a new linear operator, Mirage requires (1) an efficient floating point implementation of the operator at the kernel, block, and/or thread levels; (2) an implementation of the operator over modular arithmetic (see §5); and (3) an extension to the abstract expressions axioms A_{eq} and A_{sub} for it (see §4.3).

To utilize Theorems 3 and 4, random tests should be performed with large enough prime numbers p and q and should be iterated multiple times. Our current implementation uses the largest values of p and q whose product fits in 16-bit integers (i.e., $p = 227$, $q = 113$) to perform the random testing on GPUs, leveraging Mirage's GPU optimizations such as maintaining intermediate results in GPU shared memory, which allows Mirage to accelerate its search procedure. We also use a single random test without iterating it and compare all elements of the output tensors. We note that this equivalence verification procedure still does not introduce any false negatives. While it may introduce false positives, we have not observed any in practice. For these reasons, we consider this procedure to be sufficient for the search process, and we plan to add an additional verification step that provides the theoretical guarantees only for the best μGraph at the end of the optimization process.

7.2 Experimental Setup

We evaluated Mirage on 12 benchmarks commonly used by existing DNNs, summarized in Table 4. MHA and MLP are the two main building blocks of today's large language models (LLMs). GQA and MQA reduce memory requirements of MHA and have been widely deployed in recent LLMs. MoE uses multiple experts, each of which is an MLP architecture, to improve the predictive performance of a DNN without increasing its latency. LoRA enables low-rank adaptation
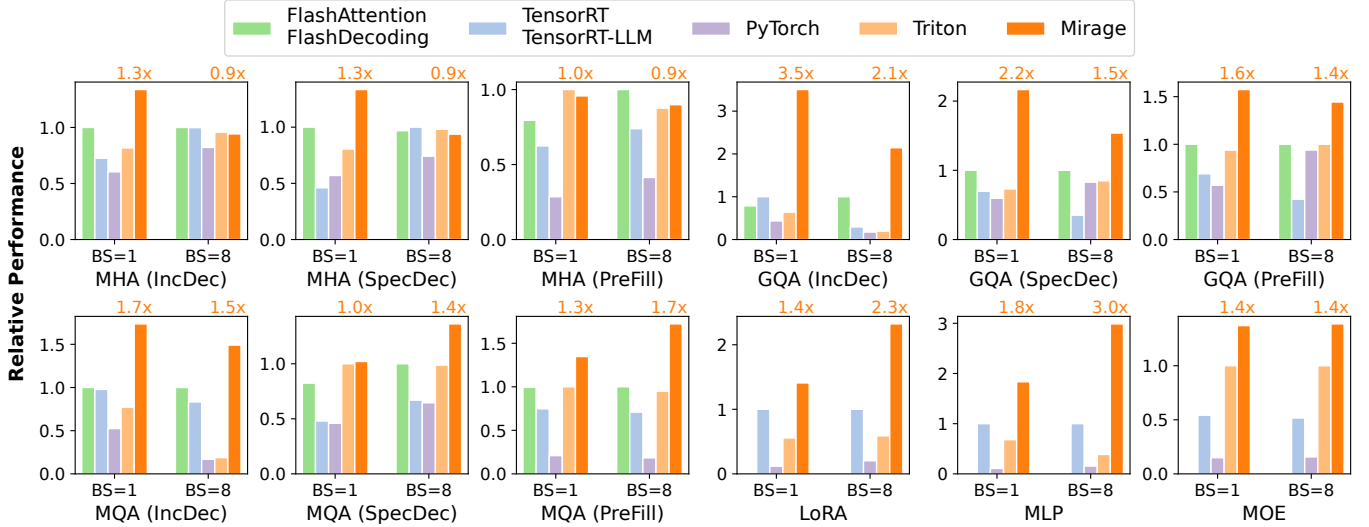
Figure 7. Comparing Mirage with existing systems for 12 benchmarks on an A100 GPU. IncDec, SpecDec, and PreFill indicate the incremental decoding, speculative decoding, and prefilling phases of attention. The performance of all systems are normalized by the best prior result (higher is better). Numbers above the Mirage bars show the speedup over the best existing systems.

for finetuning a DNN on different tasks. For all attention mechanisms, we assume 4096 key-value tokens and evaluate Mirage in three different scenarios: incremental decoding (1 query token), speculative decoding (32 query tokens), and chunked pre-filling (512 query tokens).

We use a Perlmutter compute node for all experiments [8], which is equipped with four NVIDIA 40GB A100 GPUs and 256 GB DRAM. All our benchmarks fit on a single A100 GPU except GQA (used for LLaMA-2-70B), which is generally parallelized across four A100 GPUs using tensor model parallelism [36]. Therefore, we evaluate GQA under the tensor model parallelism strategy (i.e., the 8 key-value heads are equally partitioned across four GPUs). Since the performance of Mirage and all baselines only depends on the shapes of input tensors, we repeat all performance experiments 1,000 times using random inputs and report the average run time. We observed negligible variance across different runs.

One of our benchmarks, MLP, uses the ReLU non-linear operator, which is not natively supported by Mirage. To apply Mirage to MLP, we replace ReLU by exponentiation, which is also a non-linear function and is not used by MLP. We then change exponentiation in the resulting optimized μGraph back to ReLU and verify equivalence by manual examination.

Another one of our benchmarks, LoRA, requires concatenation to express a common tensor optimization: fusing two matrix multiplications via concatenation. To support this optimization in Mirage, we add a new linear operator that takes four inputs and computes $f(W, X, Y, Z) = (W\|X) \times (Y\|Z)$, where $\|$ is tensor concatenation. This operator is equivalent

to computing $W \times Y + X \times Z$. We define the abstract expression associated with the new operator as: $E(f(W, X, Y, Z)) = \text{add}(\text{sum}(k_1, \text{mul}(E(W), E(Y))), \text{sum}(k_2, \text{mul}(E(X), E(Z))))$, where k_1 and k_2 are the last dimensions of W and X.

Unless otherwise stated, Mirage considers up to 5 operators in the kernel graph and up to 7 operators in each block graph.

7.3 Performance Results

Figure 7 compares the performance of Mirage and existing tensor program optimizers on 12 DNN benchmarks using two different batch sizes. All systems use half-precision floating points to serve all DNN benchmarks. PyTorch [31] uses the highly-engineered cuDNN and cuBLAS libraries [17, 18] to perform DNN operators on GPUs. TensorRT and its LLM variant TensorRT-LLM include a set of manually-designed and highly-optimized kernels for common tensor operators such as attention [37]. Triton [38] is a schedule-based optimizer for generating high-performance tensor programs and has been deployed in existing DNN systems, achieving superior performance than other schedule-based optimizers. We do not compare Mirage with existing superoptimizers (e.g., TASO [26] or PET [41]) since the DNNs we use as input do not contain purely algebraic optimization opportunities at the kernel level.

MHA, GQA, and MQA. Multi-head, multi-query, and group-query attention are the backbone of today's large language models and have been heavily optimized by existing

frameworks. For example, FlashAttention and FlashDecoding are expert designed kernels for attention and have been adopted in most of today's LLM inference systems [19]. For MHA, Mirage discovers both the FlashAttention and FlashDecoding kernels and selects a μGraph similar to FlashDecoding, achieving on-par performance as existing systems.

For GQA and MQA, Mirage discovers kernels that outperform best existing systems by up to 3.5× and 1.7× respectively. In addition to the optimization demonstrated in Figure 4, the speedup is also achieved by Mirage's ability to automatically parallelize attention computation efficiently across blocks within a kernel. In particular, the blocks of a kernel are organized by a mesh with up to three dimensions, but attention computation can be parallelized in more dimensions. As a result, existing manually-designed kernels use different heuristics to select up to 3 dimensions to parallelize attention. For example, FlashAttention [19] parallelizes attention across blocks using the *sample*, *head*, and *query sequence* dimensions, while FlashDecoding leverages the *sample*, *head*, and *key-value sequence* dimensions, both of which are efficient for MHA with many attention heads but suboptimal for MQA and GQA with limited attention heads. Specifically, the GQA kernel from LLaMA-2-70B only contains two key-value heads. For single-batch speculative decoding, which involves computing attention across 256 queries and 1024 key-value pairs, utilizing all 108 streaming multiprocessors on an A100 GPU requires splitting the query sequence into 64 chunks for FlashAttention (or splitting the key-value sequence into 64 chunks for FlashDecoding), as illustrated in Figure 8(a). Each block loads 256/64=4 query vectors, all 1024 key vectors, and all 1024 value vectors, resulting in 2052 vector loads.

In contrast, Mirage automatically selects the most efficient dimensions among sample, head, query sequence, and key-value sequence, and uses different μGraphs for different attention scenarios, reducing memory access and improving performance. Figure 8 compares the FlashAttention kernel and Mirage's generated kernel for GQA in the single-batch speculative decoding phase. Instead of parallelizing across the sample, head, and query sequence dimensions, Mirage finds a μGraph that parallelizes across the head, query sequence, and key-value sequence dimensions. As a result, each block only needs to load 256/8=32 query vectors, 1024/8=128 key vectors, and 1024/8=128 value vectors, reducing memory access by 7× and kernel execution time by 2.2×.

Implementing Mirage's μGraphs in existing systems is possible but requires extensive engineering effort to support different kernels for different scenarios, while Mirage automatically generates them and verify their correctness.

LoRA. Low-rank adaptation (LoRA) introduces a pair of low-rank adapters to the linear operators of a pre-trained DNN to improve its predictive performance on downstream tasks. Existing tensor program optimizers launch separate
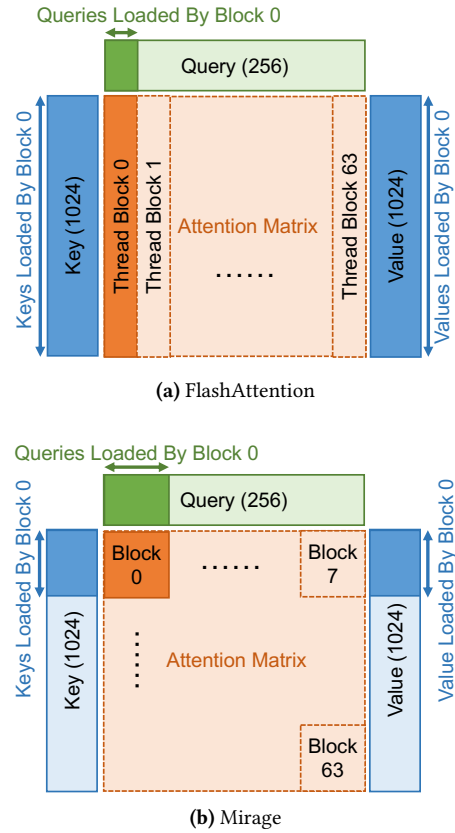

(a) FlashAttention

(b) Mirage

Figure 8. Comparing thread block assignments between FlashAttention and the μGraph discovered by Mirage. FlashAttention and FlashDecoding only parallelize Attention computation across the query *or* the key-value dimension; Mirage opportunistically leverages both dimensions to reduce the queries/keys/values loaded by each thread block.

kernels for the original linear operator and the two new linear operators in LoRA (Figure 9a), which introduces high kernel launch overheads since the LoRA operators involve very low computational costs. Figure 9b shows the best μGraph discovered by Mirage for LoRA, which fuses the three Matmuls and the subsequent Add into a single kernel. Mirage reorganizes the computation into two thread-block level Matmuls by leveraging the following algebraic transformation: $W \times X + B \times A \times X = (W \| B) \times (X \| (A \times X))$. The Concats in Figure 9b do not involve any computation and are performed by updating tensor offsets in GPU shared memory. This μGraph reduces the execution cost of LoRA by 2.3×.

MLP. Multi-layer perceptron (MLP) is commonly used in DNNs to capture non-linear representations. We use the MLP configuration introduced in adapter tuning [24]. Existing tensor program optimizers generally fuse the first Matmul with
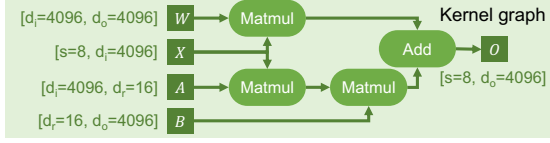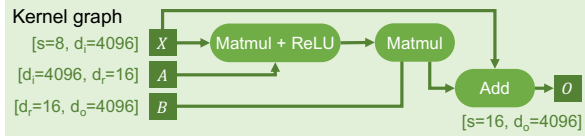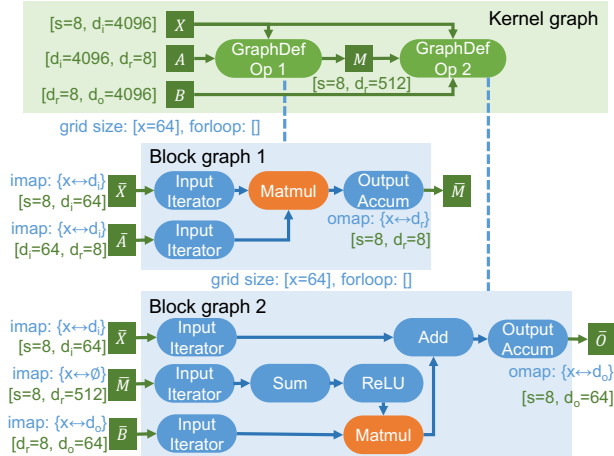
(a) The kernel graph for LoRA in existing systems.
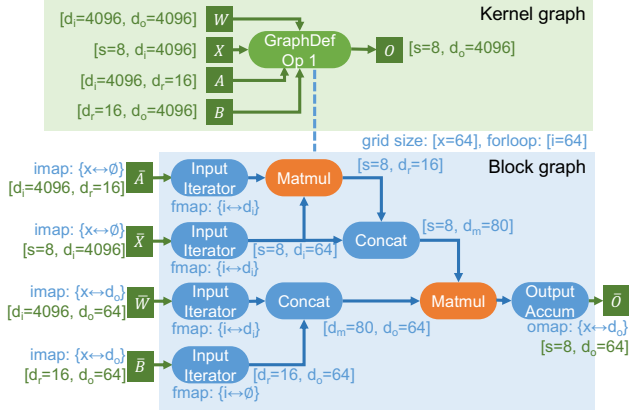
(b) The best μGraph discovered by Mirage for LoRA.

Figure 9. Comparing the tensor programs used by existing optimizers and by Mirage for LoRA: $O = W \times X + B \times A \times X$. Note that both matrices A and B are low-rank.

(a) The kernel graph for MLP in existing systems.

(b) The best μGraph discovered by Mirage for MLP.

Figure 10. Comparing the μGraphs used by existing optimizers and Mirage for MLP.

Table 5. Ablation study on Mirage's techniques to accelerate μGraph generation. We incrementally disable abstract expression (§4.3) and canonical form (§4.1), then evaluate the search times for GQA as we adjust the maximum number of operators within a block graph.

Max # Ops in a block graph	Mirage	w/o abstract expression	w/o canonical form of μGraphs
4	9sec	$<$1sec	$<$1sec
5	2.3 min	249 min	$>$12h
6	20 min	$>$12h	$>$12h
7	76 min	$>$12h	$>$12h

the subsequent ReLU to perform ReLU activation immediately after matrix multiplication to reduce kernel launch overhead and access to GPU device memory, as shown in Figure 10a. However, the first Matmul has a small output tensor shape (e.g., $B \times 16$ in adapter tuning where B is the batch size), which cannot fully utilize an A100 GPU with 108 stream multi-processors. In contrast, the best μGraph discovered by Mirage (Figure 10b) performs *partial summation* of the first Matmul in the first kernel, and fuses the remaining summation, ReLU, second Matmul, and Add in the second kernel. This design enables more parallelism for the first kernel and better utilizes an A100 GPU, yielding a 3× speedup.

7.4 Optimization Time and Ablation Study

Generally, Mirage takes up to six hours to optimize each of our benchmarks. We present more detailed results and an ablation study focusing on GQA. We evaluate how our techniques allow Mirage to explore large μGraphs while maintaining low search time. We focus on two techniques: pruning via abstract expressions (§ 4.3) and the canonical form of μGraphs (§ 4.1). Table 5 reports the search times for GQA as we adjust the maximum number of operators in a block graph. For searches that consider 4 operators in a block graph, pruning via abstract expressions slightly increases the search time due to the cost of SMT queries. For larger-scale searches, the pruning allows Mirage to explore μGraphs whose block graphs can each have at most 7 operators, while disabling abstract expression (and canonical form of μGraphs) restricts Mirage to consider up to 5 (and 4) operators in a block graph in order to finish the search in 12 hours. Note that Mirage needs to consider 7 operators in a block graph in order to discover the optimizations in Figure 4 for GQA.

8 Related Work

Manually-designed kernels have been widely used in existing frameworks such as TensorFlow XLA [1, 10], PyTorch [31], and TensorRT [37]. Recently, significant engineering effort has been dedicated to manually designing, implementing, and optimizing GPU kernels for commonly used DNNs (known

as foundation models [14]). For example, to optimize the attention mechanism of Transformer [42], recent work has introduced various kernels based on FlashAttention [4, 5, 19, 23]. Due to the increasing complexity of modern GPU architectures (e.g., tensor cores in A100s [28] and thread block clusters in H100s [7]), manually designed kernels may miss subtle optimizations that are hard to discover manually.

Superoptimization-based approaches. Superoptimization was originally introduced to find optimal code for an instruction sequences [13, 29, 33]. Recent work has applied superoptimization techniques to tensor programs [26, 41, 44, 46]. All these attempts only consider algebraic transformations at the kernel level and cannot discover sophisticated optimizations that require jointly considering algebraic and schedule transformations at all of the kernel, block, and thread levels.

Schedule-based approaches, including Halide [32], TVM [15, 16], and Ansor [45] among others [21, 22, 47], are based on the idea of algorithm-schedule separation introduced in Halide and search for an optimized schedule to execute a given algorithm on GPUs. Schedule-based approaches rely on users to explicitly specify the algorithm for each kernel and their performance is limited to the provided algorithms.

9 Conclusion

This paper proposes Mirage, the first multi-level superoptimizer for tensor programs. Mirage introduces a hierarchy graph representation to specify a tensor program at the kernel, thread block, and thread levels of the GPU compute hierarchy, and uses a novel pruning technique based on abstraction to significantly reduce the search space Mirage needs to consider while providing a certain optimality guarantee. Mirage outperforms existing tensor program optimizers by up to 3.5× even for DNNs that are widely used and heavily optimized.

References

[1] Xla: Optimizing compiler for tensorflow. https://www.tensorflow.org/xla, 2017.

[2] Nvidia/cutlass: Cuda templates for linear algebra subroutines. https://github.com/NVIDIA/cutlass, 2019.

[3] Tensorflow graph optimization with grappler. https://www.tensorflow.org/guide/graph_optimization, 2019.

[4] Transformer related optimizations. https://github.com/NVIDIA/FasterTransformer, 2020.

[5] Flash-decoding for long-context inference. https://crfm.stanford.edu/2023/10/12/flashdecoding.html, 2023.

[6] Llama-7b-lora. https://huggingface.co/Laurie/llama7b-lora-merged/tree/main, 2023.

[7] Nvidia h100 tensor core gpu. https://www.nvidia.com/en-us/data-center/h100/, 2023.

[8] Perlmutter supercomputer. https://docs.nersc.gov/systems/perlmutter/architecture/, 2023.

[9] A Triton implementation of the FlashAttention2 algorithm. https://triton-lang.org/main/getting-started/tutorials/06-fused-attention.html, 2023.

[10] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016.

[11] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[12] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Merouane Debbah, Etienne Goffinet, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. Falcon-40B: an open large language model with state-of-the-art performance. 2023.

[13] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, 2006.

[14] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the opportunities and risks of foundation models, 2022.

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018.

[16] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems 31*, NeurIPS'18. 2018.

[17] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[18] Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas, 2016.

[19] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-decoding for long-context inference, 2023.

[20] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 2008.

[21] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. Tensorir: An abstraction for automatic tensorized program optimization, 2022.

[22] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. 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*, ASPLOS 2023, page 302–313, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. Flashdecoding++: Faster large language model inference on gpus, 2024.

[24] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.

[25] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[26] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023.

[28] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2018.

[29] Henry Massalin. Superoptimizer: a look at the smallest program. In *ACM SIGARCH Computer Architecture News*, volume 15, 1987.

[30] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4), 2016.

[31] Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://pytorch.org, 2017.

[32] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.

[33] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM SIGPLAN Notices*, volume 48, 2013.

[34] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, oct 1980.

[35] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

[36] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[37] NVIDIA TensorRT: Programmable inference accelerator. https://developer.nvidia.com/tensorrt, 2017.

[38] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery.

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

[40] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models, 2023.

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

[42] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art machine learning for pytorch, tensorflow, and jax. https://github.com/huggingface/transformers, 2022.

[43] Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization, 2021.

[44] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality Saturation for Tensor Graph Superoptimization. *Proceedings of Machine Learning and Systems*, 3:255–268, March 2021.

[45] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor : Generating high-performance tensor programs for deep learning. *CoRR*, abs/2006.06762, 2020.

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

[47] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 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*, ASPLOS '20, page 859–873, New York, NY, USA, 2020. Association for Computing Machinery.

[48] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *International symposium on symbolic and algebraic manipulation*, pages 216–226. Springer, 1979.