DIRECTED ACYCLIC GRAPH OPTIMIZATION

Chi-Ning Chen [1] **Yi-Ching Chen** [1] **Yosen Lin** [1]

ABSTRACT

In modern machine learning systems, directed acyclic graph (DAG) optimization is critical for maximizing inference efficiency, especially for deep neural networks (DNNs) where transferring intermediate tensors between limited on-chip memory and slow DRAM dictates overall latency. However, finding the optimal DAG partition and tiling strategy to mitigate this "memory wall" is a complex combinatorial problem. Thus, we propose a hierarchical DAG scheduling framework that optimizes operator fusion and execution granularity. Our approach generates fusion candidates using a seed-growth strategy to capture non-linear dependencies, evaluating them against a roofline-based cost model. To ensure scalability across massive DAGs, we employ graph-cut decomposition combined with a frontier dynamic programming (DP) search. Evaluated against MLSys benchmark workloads, our scheduler achieves up to a 35.8% reduction in end-to-end latency for memory-bound networks compared to baseline interval-based methods, while reaching near-optimal roofline limits on compute-bound graphs.

1 INTRODUCTION

The computation of modern machine learning models, particularly DNNs, can be represented as a DAG of tensor operators, such as matrix multiplications and element-wise operations. While computational capabilities have scaled rapidly, optimizing these DAGs for execution encounters a critical hardware bottleneck: tight fast-memory budgets and exceptionally expensive data movement. Because on-chip fast memory is strictly limited, it is impossible to keep all intermediate tensors on-chip during execution. As a result, systems are forced to spill intermediate data to slow memory, incurring significant transfer overhead that often dominates the end-to-end inference latency.

To mitigate this bottleneck, operator fusion has been proposed to parallelize the execution of multiple tensor operators (e.g., convolutional layers). By fusing operators into localized groups, intermediate feature maps can be consumed immediately by subsequent operators and kept entirely within the on-chip buffer, eliminating redundant off-chip memory traffic. Thus, DAG optimization seeks to mitigate this memory wall by finding a scheduling strategy that partitions the computational graph into subgraphs and selects appropriate tiling granularities to minimize latency, subject to strict fast-memory capacity and memory-bandwidth constraints.

However, achieving an optimal schedule is extremely difficult. First, subgraph partitioning is an inherently combinatorial problem. Even a minor change in subgraph boundaries can drastically alter which tensors must be materialized as inputs or outputs and which can remain ephemeral. Second, hardware-sensitive tiling and retention choices present non-obvious trade-offs between compute and memory. For instance, naive choices that attempt to improve arithmetic intensity may inadvertently increase the memory footprint beyond the available fast-memory capacity.

While recent works like the Optimus (Cai et al., 2022) framework have successfully identified that intermediate feature maps form the primary memory bottleneck and applied dynamic programming for DAG partitioning, a comprehensive solution requires traversing multiple levels of the compilation stack. Our goal in this project is to solve the DAG optimization problem by constructing a comprehensive scheduler that maximizes computational efficiency while satisfying all system constraints, including fast-memory capacity, memory bandwidth, and execution granularity requirements. In pursuit of these objectives, this project is specifically developed within the context of the MLSys Competition (Track A)[1]. By targeting the performance metrics and hardware-specific constraints defined by this track, we aim to demonstrate the practical efficacy of our framework. Our implementation is designed to meet the competitive standards of the MLSys platform, ensuring that our strategies are not only theoretically robust but also fine-tuned for the benchmarks provided by the competition.

To address the combinatorial complexity of graph partitioning and the physical constraints of hardware, we propose a hierarchical execution framework to optimize massive computational graphs with limited on-chip memory. Specifically, our project makes the following contributions:

- **DAG-Aware Operator Fusion:** We designed and im-

[1] https://github.com/yarongmu-google/MLSys

plemented both interval-based and seed-growth candidate group generation algorithms, enabling automatic fusion of MatMul and Pointwise operators according to the DAG structure. This supports non-linear, complex dependency subgraph fusion and significantly reduces off-chip transfers of intermediate tensors.

- **Retain-Aware Partition Search:** Our project incorporates tensor retention decisions into the DP-based partition search, ensuring that each subgraph retains only the necessary tensors. We automatically infer cross-subgraph tensor residency, improving memory efficiency and avoiding unnecessary spills.

- **Conv-Style Accelerator Guidance:** The project integrates pseudo-conv mapping and a conv-style accelerator model to analyze working set, memory traffic, and tile propagation. These analyses are used as auxiliary signals for candidate selection and ranking, improving performance on convolution-like accelerators.

- **Scalable Hybrid Search:** For large DAGs (over 20 operators), we implemented graph-cut decomposition and frontier DP, splitting large graphs into segments and applying DP search for fusion strategies, balancing solution quality and runtime scalability.

- **Comprehensive Benchmarking and Workflow:** We established automated build, benchmarking, and result validation pipelines, and tested all public MLSys competition datasets to ensure that each optimization delivers real-world benefits.

2 PROBLEM

We model the DNN computation as a DAG $G = (V, E)$, where each node $v_i \in V$ represents a tensor operator (e.g., MATMUL or POINTWISE), and each directed edge $e_{ij} \in E$ encodes a data dependency between operators. The target hardware platform is characterized by three primary parameters: the fast-memory capacity C (on-chip SRAM), the slow-memory bandwidth B (DRAM bandwidth), and a native granularity G_{native} defining the smallest tile the hardware can execute.

The optimization objective is to partition V into a set of fused subgraphs $\{S_1, S_2, \ldots, S_k\}$ and assign each subgraph S_j a tile granularity $g_j = (g_w, g_h, k_{\text{step}})$ so as to minimize the total end-to-end execution latency:

$$\text{minimize} \quad \sum_{j=1}^{k} L(S_j,\, g_j) \tag{1}$$

The per-step latency of each subgraph is governed by the Roofline Model:

$$L_{\text{step}} = \max\left(\frac{\text{compute_cycles}}{1},\ \frac{\text{memory_bytes}}{B}\right) \tag{2}$$

This formulation captures the fundamental trade-off between arithmetic throughput and memory bandwidth. However, achieving an optimal schedule that satisfies these constraints in practice requires navigating three critical technical challenges.

First, systems are constrained by tight fast-memory budgets and expensive data movement. Because the fast-memory capacity C is strictly limited, not all intermediate tensors can be kept on-chip. Spilling these tensors to slow memory introduces significant transfer overhead that can easily dominate total execution latency. For example, if two consecutive operators produce and consume a massive activation tensor, executing them in separate subgraphs forces the activation to be written to and read from slow memory. While fusing them would ideally avoid this costly round-trip, it requires the fused subgraph's active memory footprint to stay strictly within C .

Second, identifying the optimal subgraph partitioning is inherently combinatorial and strictly bound by dependency and validity constraints. Deciding precisely where to cut the DAG fundamentally alters which tensors become materialized inputs or outputs and which intermediate results remain ephemeral. Furthermore, any partition must maintain topological validity. For instance, grouping three sequential operators $\{A \to B \to C\}$ into a single subgraph is perfectly valid, but grouping $\{A, C\}$ while isolating B breaks the data dependencies, as the selected operators no longer form a legally connected sub-DAG.

Third, hardware-sensitive tiling and tensor retention choices involve non-obvious trade-offs. The optimal tiling granularity g_j heavily depends on the complex interaction between compute throughput, memory bandwidth B, and padding overhead. Choices designed to maximize arithmetic intensity might simultaneously inflate the memory footprint beyond the hardware capacity C. As an illustration, utilizing a larger tile size can minimize kernel launch overhead and improve data reuse, but if the resulting footprint exceeds C, the ensuing register spills or cache thrashing will ultimately degrade the latency far more than using a conservatively smaller tile size.

To systematically address these challenges and bridge the gap between formal constraints and compiler optimizations, our project focuses on the following research questions (RQs):

- **RQ1 (DAG-Aware Fusion & Validity):** How can we efficiently generate and search the combinatorial space

of operator fusion candidates—using both interval-based and DAG-aware (seed-growth) strategies—to find memory-optimal subgraph partitions that strictly respect topological and dependency constraints?

- **RQ2 (Retain-Aware Scheduling):** What strategies enable optimal tensor retention and tiling decisions within each subgraph, balancing recomputation, spilling, and memory footprint to maximize throughput under fast-memory constraints?
- **RQ3 (Conv-Style Guidance & Scalability):** How can we leverage conv-style accelerator models and hybrid search techniques (e.g., graph-cut decomposition, DP, and conv-guided heuristics) to scale scheduling and fusion to large, real-world DAGs while maintaining solution quality and hardware alignment?

3 Related Work

Prior work on DNN execution optimization spans two complementary levels of abstraction:(1) kernel-level techniques that restructure the computation within a single GPU kernel to reduce slow-memory traffic and (2) graph-level techniques that partition a DNN computation graph into fused operator groups to eliminate intermediate tensor round-trips to off-chip memory. Our system operates at the graph level and is most directly descended from the second line of work, but its design decisions are informed by insights from both.

3.1 Kernel-Level IO-Aware Optimization

On modern hardware accelerators, arithmetic throughput has grown faster than memory bandwidth, making data movement a dominant bottleneck rather than computation in DNN inference. Dao et al. (Dao et al., 2022) addresses this bottleneck specifically for the self-attention mechanism in Transformers. Standard attention implementations materialize the full N×N attention score matrix in high-bandwidth memory (HBM). FlashAttention reformulates the computation using tiling and on-chip recomputation to avoid this materialization entirely. Sun et al. (Sun et al., 2024) complement this work by providing a systematic empirical analysis of tensor kernel fusion on NVIDIA Tensor Core GPUs. They study the conditions under which fusing two consecutive GEMM kernels into a single GPU kernel is beneficial, finding that fusion is the most effective for tall-and-narrow matrices (N_0, N_1" $\in \{64, 128, 256\}$") typical of Transformer hidden dimensions, but that benefits diminish as matrix width grows and shared-memory pressure triggers register spilling. Neither FlashAttention nor Sun et al. address the problem of deciding which operators in a DNN computation graph should be fused together. They assume that the fusion boundary is fixed and optimize the implementation within that boundary. Our system takes the complementary view: given a hardware model characterized by a fast-memory capacity, a slow-memory bandwidth, and native tile granularity, we search over all possible fusion partitions of the operator graph to find the one that minimizes total slow-memory access.

3.2 Graph-Level Operator Fusion

Operator fusion at the graph level reduces intermediate feature-map traffic by scheduling the consumers of a tensor immediately after its producer, keeping intermediate results in fast on-chip memory rather than writing them to slow DRAM. Early work such as Fused-layer (Sun et al., 2024) demonstrated extensive fusion search but was limited to linear (non-branching) DNN topologies. DNNVM(Xing et al., 2019) and Efficient-S (Zheng et al., 2020) extended fusion to branching architectures by treating multi-input operators as barriers or by transforming the DAG into a linearized proxy, but these restrictions cause both methods to miss profitable fusion opportunities in networks with complex branch structures. Cai et al. propose Optimus [6], which addresses these shortcomings by formulating operator fusion as a minimum-memory-access optimization over all valid partitions of the DNN DAG, solved via a bottom-up dynamic programming algorithm. A key contribution is the relaxation of the constraint that all parameters of a fused group must reside on-chip simultaneously; by permitting parameter-induced memory access and accounting for it through a scheduler-driven cost model, Optimus achieves 98% estimation accuracy and obtains up to 3.66× energy efficiency over prior methods on standard CNN benchmarks.

3.3 Multi-Level Superoptimization

Mirage(Wu et al., 2025) introduces μGraphs, a hierarchical graph representation that captures computation at the GPU kernel, thread block, and thread levels within a unified formalism. Given an input tensor program, Mirage exhaustively searches the space of equivalent μGraphs, combining algebraic transformations such as reordering matrix multiplication and normalization operators, schedule transformations such as selection of parallelization strategies, and the synthesis of entirely new custom kernels. To make the exponentially large search space tractable, Mirage employs an expression-guided pruning technique based on abstract expressions and SMT-solver-backed subexpression checks and verifies the correctness of each candidate μGraph via probabilistic equivalence testing over finite fields. In representative LLM building blocks including group-query attention (GQA), RMSNorm, LoRA, and GatedMLP, Mirage outperforms heavily optimized baselines such as FlashAttention, TensorRT-LLM, and Triton by up to 3.3×.

4 OVERVIEW

The system takes as input a DAG of matrix multiplication and pointwise operations over 2D tensors with hardware specs (fast memory size, bandwidth, execution granularity), converted into an operator graph (OpGraph) for optimization.

We use a two-phase solver. First, candidate groups are generated via seed-growth BFS with beam pruning on a proxy latency score; large graphs are decomposed by graph-cut heuristics into independent segments. Second, a frontier-based DP selects a minimum-latency cover subject to memory constraints via memoization caps. Each candidate is evaluated with a roofline cost model (latency $=$ $\max(\text{compute},\ \text{memory bandwidth})$ per tile), with granularity search over tile parameters (width, height, depth) to find the optimal configuration. The output is a solution JSON specifying subgraph groupings, granularities, and retained decisions for deployment.

5 METHOD

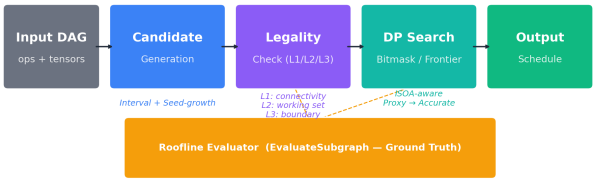


Figure 1. System overview of our operator fusion pipeline.

Figure 1 illustrates the overall pipeline of our system. The input DNN computation graph is first passed to the **Candidate Generation** module, which enumerates candidate fusion groups using both interval-based enumeration and our proposed seed-growth strategy. Each candidate is then checked for legality by the **Legality Checker**, which enforces fast-memory capacity constraints and acyclicity. Valid candidates are evaluated by a two-stage **Granularity Selector** that uses a lightweight proxy cost model to rank candidates before invoking the full Roofline evaluator. Finally, the **DP Search** module solves the optimal partition problem over the legal, evaluated candidates and emits the output schedule, consisting of colored subgraph assignments and their associated granularity configurations.

5.1 DAG Representation

We represent the computation graph as an `OpGraph` structure that explicitly maintains the following information for efficient fusion analysis.

Producer/consumer relationships. For each tensor t, we record its unique producer $\text{prod}(t)$ and its consumer set $\text{cons}(t)$.

Topological ordering. A deterministic topological ordering π is computed using Kahn's algorithm and used throughout the pipeline to ensure consistent candidate enumeration and DP traversal.

Group boundary analysis. For any operator subset $S \subseteq V$, we define:

- *Boundary inputs*: $\{t \mid t \text{ is consumed by } S,\ \text{prod}(t) \notin S\}$
- *Boundary outputs*: $\{t \mid t \text{ is produced by } S,\ \exists c \in \text{cons}(t),\ c \notin S\}$
- *Internal tensors*: $\{t \mid \text{prod}(t) \in S,\ \text{cons}(t) \subseteq S\}$

Internal tensors can be kept in fast memory throughout the execution of S, eliminating the slow-memory round-trip that would otherwise be incurred between separate operator launches.

5.2 Fusion Candidate Generation

5.2.1 *Interval-Based Candidates*

Following Optimus (Cai et al., 2022), we enumerate all contiguous topological ranges as a baseline set of candidates:

$$\mathcal{C}_{\text{interval}} = \big\{\{\pi(i), \pi(i{+}1), \ldots, \pi(j)\} \mid 1 \le i \le j \le N,\ j - i < M_{\max}\big\} \quad (3)$$

where $M_{\max}$ is a maximum group-size threshold. While straightforward to enumerate, interval-based candidates can only express contiguous sequences in topological order and therefore miss fusion opportunities that arise from operators on parallel branches of the DAG.

5.2.2 *Seed-Growth Candidates*

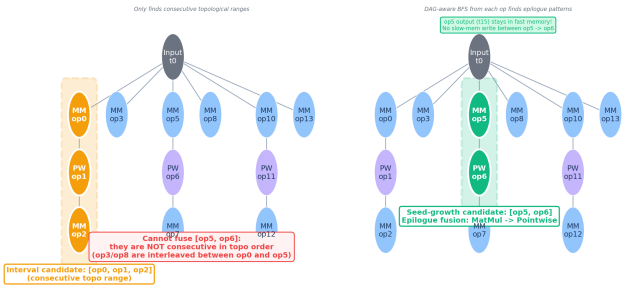


Figure 2. System overview of our operator fusion pipeline.

To capture cross-branch fusion opportunities, we propose a **seed-growth** strategy that treats each operator as a seed and expands outward along producer/consumer edges via breadth-first search. This allows the candidate generator to form groups that are non-contiguous in topological order but are nonetheless connected in the DAG and share internal tensors. As illustrated in Figure 2, seed-growth discovers groups such as $\{\text{op5}, \text{op6}\}$ on a parallel branch that interval enumeration cannot express, because `op5` and `op6` are not adjacent in topological order.

Epilogue fusion pattern. We additionally identify the pattern in which a MATMUL operator's sole output is consumed by exactly one downstream POINTWISE operator. In this case, the intermediate tensor can remain in fast memory across the two operators, saving two slow-memory transfers per step. The latency saving attributable to epilogue fusion is:

$$\Delta L_{\text{epilogue}} = \frac{|T_{\text{intermediate}}| \times 2}{B} \cdot \frac{N_{\text{steps}}}{k_{\text{step}}} \tag{4}$$

where $|T_{\text{intermediate}}|$ is the byte size of the fused tensor and N_{steps} is the total number of tile steps in the MATMUL.

5.3 Fusion Core Logic

The core of our scheduler is a fusion engine that partitions the operator DAG into a sequence of fused subgraphs (also called "groups" or "segments"). The goal is to minimize total execution latency by maximizing fusion opportunities, while strictly respecting hardware memory and bandwidth constraints. This section details the methodology, including candidate generation, legality analysis, and the dynamic programming (DP) search for optimal subgraph sequences.

5.3.1 Candidate Subgraph Generation

We generate candidate fused subgraphs using two complementary strategies:

Interval-based Candidates All possible contiguous ranges of operators in the DAG's topological order are enumerated. Each candidate is a set of consecutive operators, forming a potential fused group. This approach is efficient and ensures that all simple fusion opportunities are considered.

Seed-Growth Candidates To capture more flexible fusion opportunities, a seed-growth strategy is also employed. Here, the algorithm starts from a "seed" operator and iteratively expands the candidate group by adding adjacent operators (either predecessors or successors) in the DAG, provided that each expansion maintains legality. The expansion uses a queue and a visited set to ensure each candidate is a connected subgraph.

Algorithm 1 Seed-Growth Candidate Generation (Pseudocode)

```
Initialize queue with all possible seed ops
while queue not empty do
    Pop candidate group G from queue
    for each op o adjacent to G do
        if adding o to G is legal then
            Add G ∪ {o} to queue
        end if
    end for
end while
```

5.3.2 Fusion Boundary Analysis

For each candidate group, we compute its data boundaries to determine external dependencies:

- **Boundary Inputs:** Tensors required by the group that are produced outside the group.
- **Boundary Outputs:** Tensors produced by the group that are consumed outside the group.

This is implemented by iterating over all operators in the candidate group, collecting input tensors whose producers are outside the group (boundary inputs), and output tensors with consumers outside the group (boundary outputs). These sets are used to enforce correct data dependencies between subgraphs.

5.3.3 Legality and Feasibility Checks

Each candidate group is validated for the following properties:

1. **DAG Connectivity:** The group must form a connected subgraph in the operator DAG. This is checked by performing a BFS or DFS from any node in the group and ensuring all group nodes are reachable.
2. **Resource Constraints:** The working set (all live tensors during execution) must fit within the available fast memory. The total memory footprint is computed as the sum of all tensors live during the group's execution, including boundary tensors.
3. **No Partition Cycles:** Fusion must not introduce cycles in the partitioned DAG. This is checked by ensuring that, for any op in the group, no external path exists from its successors back into the group.

4. **Output Shape and Policy Constraints:** Additional rules, such as requiring all ops in a group to have compatible output shapes, are enforced as needed.

Candidates failing any check are filtered out. The legality checks are modular, allowing for easy extension or modification of fusion policies.

5.3.4 Subgraph Sequence Selection

To select the optimal sequence of fused subgraphs covering all operators, a dynamic programming (DP) search is employed:

DP State and Recursion The DP state is defined by the current position in the topological order and the set of retained tensors. The recursive search explores all legal candidate groups starting at the current position, recursively solving for the remainder of the DAG.

Algorithm 2 DP Search for Optimal Subgraph Sequence (Pseudocode)

```
function DP(pos, retained)
    if all ops are covered then
        return 0
    end if
    for each legal candidate group G starting at pos do
        cost ← latency of G + DP(next position, updated retained)
        if cost is better than best then
            Update best solution
        end if
    end for
    return best solution
end function
```

Memoization Partial solutions are memoized using a hash of the DP state to avoid redundant computation and ensure efficiency.

Greedy Fallback For very large graphs where DP is infeasible, a greedy heuristic is used: at each step, select the best available candidate group according to a cost model.

5.3.5 Solution Construction and Retain Assignment

The final solution is a sequence of fused subgraphs, each with:

- A set of assigned operators.
- Chosen tile granularity (spatial/reduction dimensions).
- A set of tensors to retain, ensuring all downstream dependencies are satisfied.

Retain Logic Implementation A key technical challenge is ensuring that every tensor needed by a later subgraph is retained by the last subgraph that produces it. This is implemented as follows:

Algorithm 3 Retain Assignment for Cross-Subgraph Tensors (Pseudocode)

```
for each tensor t needed by a later subgraph do
    Find the last subgraph S that produces t
    Add t to S's retain set
end for
```

The retain set for each subgraph is the union of the dynamic programming memo retain set and the last-producer retain set, guaranteeing correctness even if the DP memo is incomplete.

Validation The constructed solution is validated by simulating execution, checking that all subgraphs have the tensors they need, and that memory constraints are never violated.

5.3.6 Summary

This fusion core logic enables the scheduler to exploit both local and global fusion opportunities, balancing aggressive fusion with strict correctness and resource guarantees. The modular implementation, with clear algorithmic structure and extensive debug instrumentation, facilitates both correctness and extensibility.

5.4 Legality Checking

Every candidate group S is subjected to three legality checks applied in order of increasing computational cost, so that cheap checks prune the space before expensive ones are invoked.

ISOA-aware working set estimation. For linear MATMUL chains, we apply an In-Situ Output Activation (ISOA) policy in which an output tile is immediately consumed by the next operator as soon as it is produced, so only one tile's worth of buffer is required rather than the full tensor. The corrected working set estimate is:

$$WS_{\text{ISOA}}(S) = \sum_{t \in \text{boundary}} \text{tile_bytes}(t) + \sum_{t \in \text{internal}} \text{tile_bytes}(t) \tag{5}$$

The naive estimate WS_{naive} may overcount internal tensor capacity by a factor proportional to the number of tiles, causing otherwise legal fusion groups to be incorrectly rejected. Adopting WS_{ISOA} substantially increases the number of groups that pass L2 without compromising correctness.

Table 1. Legality check levels applied to each fusion candidate.

Level	Condition	Rationale
L1	Connected sub-DAG	The subgraph induced by S must be connected; isolated nodes indicate a malformed candidate.
L2	Working set $\leq C$	The total byte footprint of all boundary and internal tiles must fit in fast memory.
L3	Boundary tensor validity	Every boundary output tensor must be fully produced; partial computation of an output is not permitted.

5.5 Granularity Selection

5.5.1 *Two-Stage Search*

For each legal candidate group S, we select the optimal tile granularity using a two-stage procedure that balances search quality against evaluation cost.

Stage 1 — Proxy ranking. A lightweight proxy cost model estimates the approximate latency of all candidate granularities and ranks them in ascending order, retaining the top-K for accurate evaluation:

$$L_{\text{proxy}}(g) = \max\left(\text{compute_cost}(g),\ \frac{\text{active_bytes}(g)}{B}\right) \tag{6}$$

Stage 2 — Accurate evaluation. The top-K candidates are passed to a tile-level Roofline simulator (`EvaluateSubgraph`) that models the full memory access pattern including LRU cache behavior across k_{step} iterations. The granularity with the lowest simulated latency is selected.

5.5.2 *Forced Max-Height Evaluation*

We observe a systematic bias in the proxy model: it tends to underestimate the benefit of large-height, small-k_{step} tile configurations, because it does not track per-step LRU cache hit rates. As shown in Figure 3, the proxy model predicts monotonically increasing latency with tile height, whereas the accurate evaluator reveals a minimum in the large-tile regime.

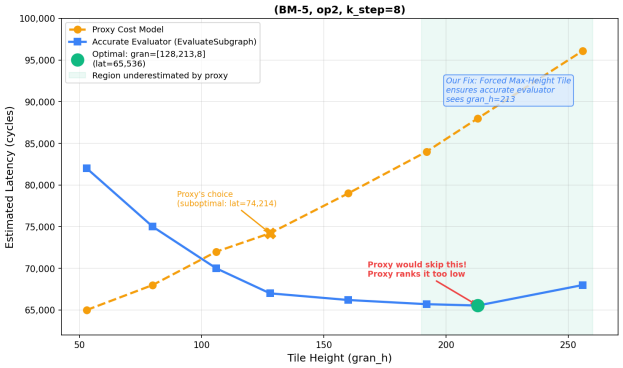


Figure 3. System overview of our operator fusion pipeline.

In a representative case (BM-5, `op2`), the proxy selects granularity $(128, 128, 32)$ with estimated latency 74,214 cycles, while the accurate evaluator identifies $(128, 213, 8)$ with latency 65,536 cycles—a gap of approximately 12%.

To correct for this bias, `BuildBestCandidate` forces evaluation of the $(w_{\max}, h_{\max})$ tile for every feasible k_{step} value, ensuring that large-tile configurations always enter Stage 2 regardless of their proxy ranking.

5.6 Optimal Partition Search

5.6.1 *Bitmask DP for Small DAGs ($N \leq 20$)*

For computation graphs with at most 20 operators, we solve the optimal partition problem exactly using bitmask dynamic programming. Let each subset of operators be represented as a bitmask; the DP recurrence is:

$$dp[\text{mask}] = \min_{\substack{S \subseteq \text{mask} \\ S \text{ legal}}} \left(dp[\text{mask} \setminus S] + L^*(S)\right) \tag{7}$$

where $L^*(S)$ denotes the latency of group S under its optimal granularity, and the base case is $dp[0] = 0$. The total search space is $O(3^N)$, corresponding to the number of ways to partition a set into two subsets summed over all masks. For $N \leq 20$, this is tractable within a few seconds on a single CPU core.

5.6.2 *Frontier DP for Large DAGs ($N > 20$)*

For larger graphs, exhaustive bitmask DP becomes intractable. We propose **Frontier DP**, which combines arbitrary-width bitset representations of operator subsets with a graph-cut decomposition strategy.

Graph-cut decomposition. We identify cut points in the topological ordering at which no tensor crosses the boundary. Formally, position c is a valid cut point if and only if:

$$\forall\, t \in \text{tensors}: \quad \text{prod}(t) \leq c \implies \text{cons}(t) \leq c \quad (8)$$

The DAG is split at these cut points into independent segments, each solved by bitmask DP separately. The optimal schedules of the segments are then concatenated to form the global solution. As illustrated in Figure 4, this decomposition reduces wall-clock search time from approximately 60 seconds to 3 seconds on a 40-operator benchmark, while retaining optimality within each segment. Since valid cut points by definition have no cross-boundary tensors, no internal-tensor savings are lost at segment boundaries.

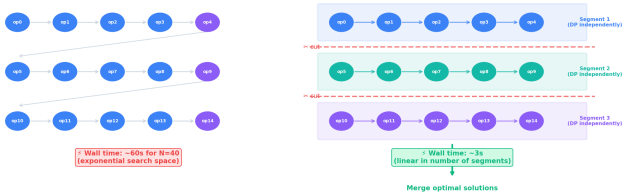


Figure 4. System overview of our operator fusion pipeline.

6 Evaluation

Datasets

The following table lists the 5 benchmark problem instances from the MLSys 2026 competition, ranging from 5 to 103 operations:

Benchmark	Ops	Tensors	Mem (B)	BW (B/cycle)
mlsys-2026-1	5	9	60,000	20
mlsys-2026-5	19	29	30,000	15
mlsys-2026-9	32	49	250,000	25
mlsys-2026-13	63	100	600,000	50
mlsys-2026-17	103	160	500,000	100

Table 2. Benchmark configurations

As shown in Table 2, all share native granularity [128, 128]. Op types are MatMul and Pointwise only.

Evaluation Metric

Total latency is calculated as:

$$\sum_{\text{subgraphs}} \sum_{\text{tile steps}} \max(\text{compute_time}, \text{memory_time})$$

as computed by the provided reference evaluator. Lower is better.

Experimental Settings

We ran our scheduler on a simulated hardware, so there were no real GPU involved. Solver runs on CPU, and wall time was measured locally. The evaluator is deterministic given a fixed solution.

Hyperparameters

Parameter	Value
Tile spatial candidates	{output dim, 128, 64, 32}
Max segment size	25
Candidate strategy	Seed-growth

Table 3. Configuration parameters for fusion and spatial tiling optimization

Table 3 details the specific configurations used for optimization. The *Tile spatial candidates* parameter defines the per-op tile sizes explored for each spatial dimension, where "output dim" indicates that no tiling is applied in that specific dimension. For grouping operations, the *Max segment size* is set to 25, establishing an upper bound on the number of operations per fused group. Finally, the *Candidate strategy* utilizes a "Seed-growth" approach, which relies on DAG-edge traversal to generate fusion groups.

Results

As shown in Table 4, scheduling improvements yield the most pronounced gains on smaller, memory-bound benchmarks, with benchmarks 1 and 5 seeing latency reductions of 15.2% and 35.8% respectively. This is primarily attributed to DAG-aware operator fusion and retention-aware partition search, which eliminate unnecessary slow-memory round-trips for intermediate tensors.

In contrast, benchmarks 9, 13, and 17 are dominated by compute-bound matrix multiplications (MatMuls), where measured latency sits within 2% of the theoretical hardware minimum:

$$T_{min} \approx \frac{\sum \text{base_cost} \times \text{total_tiles}}{\text{native_tile}} \quad (9)$$

This proximity to the roofline limit leaves essentially no headroom for further scheduling improvements.

7 Conclusion

We present a Directed Acyclic Graph (DAG) scheduler that partitions complex graphs of MatMul and Pointwise operations into fused subgraphs with optimized tile granularities to minimize total execution latency under a roofline cost model. To maintain search tractability for large DAGs, our solver integrates seed-growth Breadth-First Search (BFS) candidate generation with a frontier set-cover Dynamic Programming (DP) approach, further utilizing graph-cut decomposition. We enhance search quality through multithreaded candidate evaluation, beam pruning, and analytical tile sizing specifically targeting the memory-bandwidth roofline knee. Empirical results demonstrate a 17% latency reduction on the mlsys-2026-5 benchmark compared to the baseline interval-candidate solver, while effectively matching baseline performance on compute-bound kernels. Our analysis

Method	bench-1	bench-5	bench-9	bench-13	bench-17
Baseline	471,501	1,013,623	167,531,000	166,415,000	46,441,851
+ Retain-Aware Partition Search	—	759,125 (−25%)	—	—	—
+ DAG-Aware Operator Fusion	—	650,528 (−12%)	—	—	46,338,000 (−0.2%)
+ Scalable Hybrid Search	—	—	164,506,000 (−1.8%)	166,403,000 (∼0%)	*(wall time only)*
+ Small Optimizations	400,000 (−15.2%)	—	—	—	—
Final	**400,000**	**650,528**	**164,506,000**	**166,403,000**	**46,338,000**
vs. Baseline	−15.2%	−35.8%	−1.8%	∼0%	−0.2%

Table 4. Ablation study of each optimization component across benchmarks (latency in cycles). Percentages indicate reduction relative to the previous row. "—" indicates the component does not apply to that benchmark.

confirms that benchmarks 9, 13, and 17 are constrained by irreducible arithmetic work, establishing a performance floor where neither fusion nor tiling strategies can yield further optimization.

REFERENCES

Cai, X., Wang, Y., and Zhang, L. Optimus: An operator fusion framework for deep neural networks. *ACM Transactions on Embedded Computing Systems*, 22(1):1–26, 2022.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, volume 35, pp. 16344–16359, 2022.

Sun, W., Li, A., Stuijk, S., and Corporaal, H. How much can we gain from tensor kernel fusion on GPUs? *IEEE Access*, 12:126135–126144, 2024.

Wu, M., Cheng, X., Liu, S., Shi, C., Ji, J., Ao, M. K., Velliengiri, P., Miao, X., Padon, O., and Jia, Z. Mirage: A multi-level superoptimizer for tensor programs. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation*, 2025.

Xing, Y., Liang, S., Sui, L., Zhang, Z., Qiu, J., Jia, X., Liu, X., Wang, Y., Shan, Y., and Wang, Y. DNNVM: End-to-end compiler leveraging operation fusion on FPGA-based CNN accelerators. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019.

Zheng, S., Zhang, X., Ou, D., Tang, S., Liu, L., Wei, S., and Yin, S. Efficient scheduling of irregular network structures on CNN accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3408–3419, 2020.