# Mirage: A Multi-Level Superoptimizer for Tensor Programs

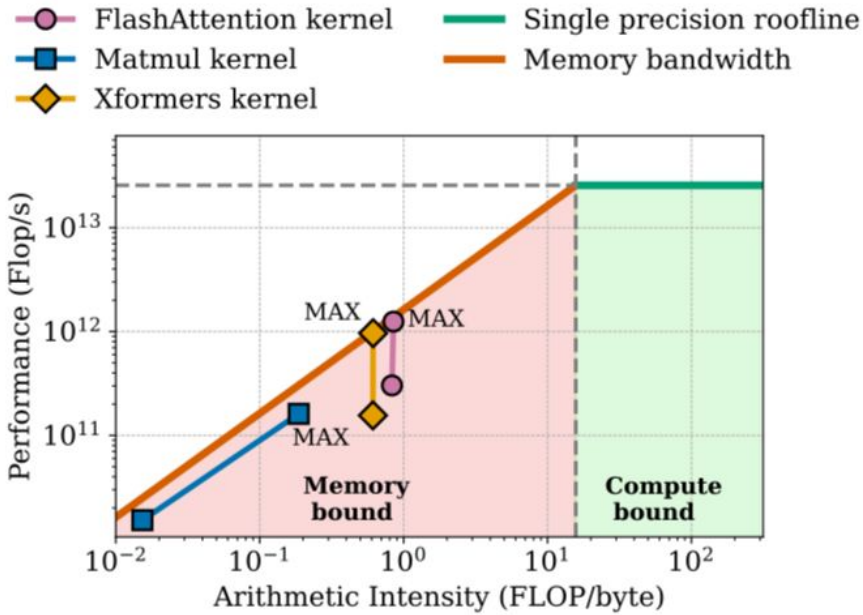
Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji,
Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, Zhihao Jia

Andy Cheng, Alexander Ingare, Egil Rolstad, Arya Tschand

# Background & Motivation

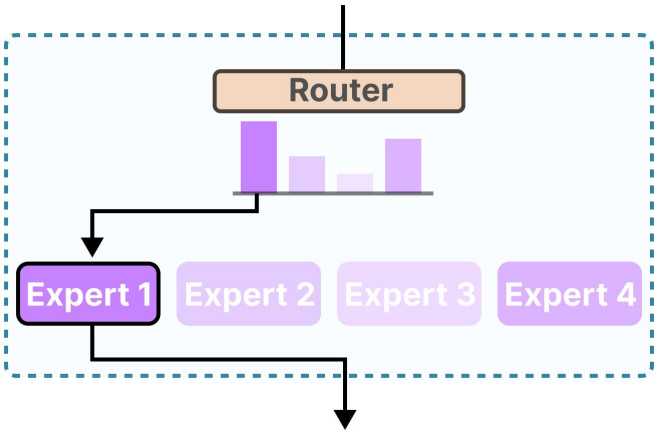
# Why Fast Kernels Matter

- LLM inference/training bottlenecked by GPU kernel performance
- A handful of ops dominate compute: attention, matmuls, normalization



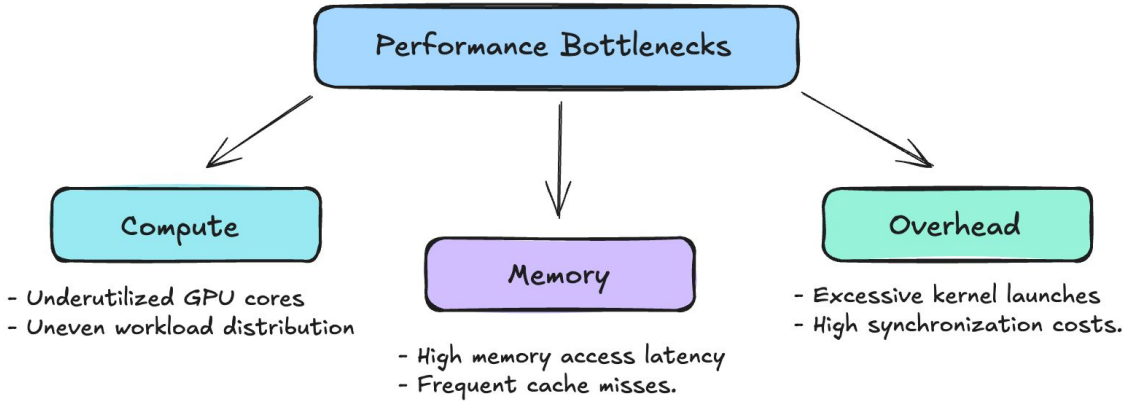
# Why Fast Kernels Matter

- LLM inference/training bottlenecked by GPU kernel performance
- A handful of ops dominate compute: attention, matmuls, normalization
- Big gap between peak GPU throughput and what frameworks actually achieve

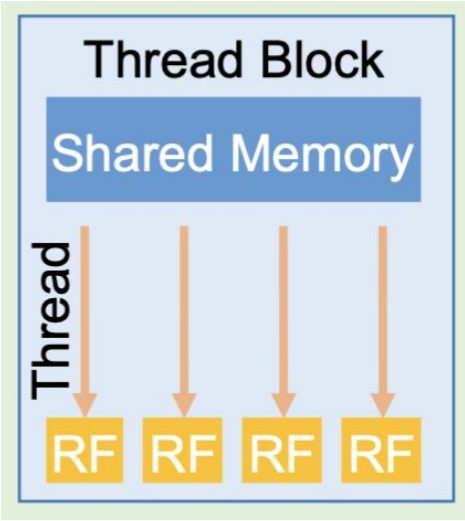


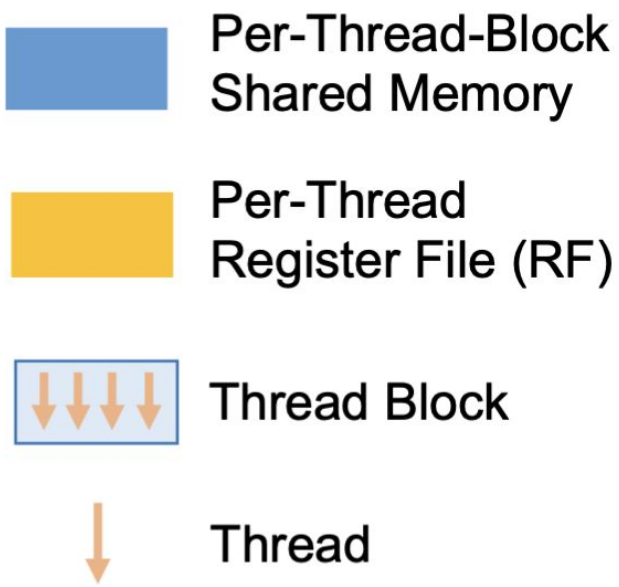
# Why Fast Kernels Matter

- LLM inference/training bottlenecked by GPU kernel performance
- A handful of ops dominate compute: attention, matmuls, normalization
- Big gap between peak GPU throughput and what frameworks actually achieve
- Causes: memory bandwidth limits, kernel launch overhead, suboptimal data movement across GPU memory levels



# GPU Compute + Memory Hierarchy

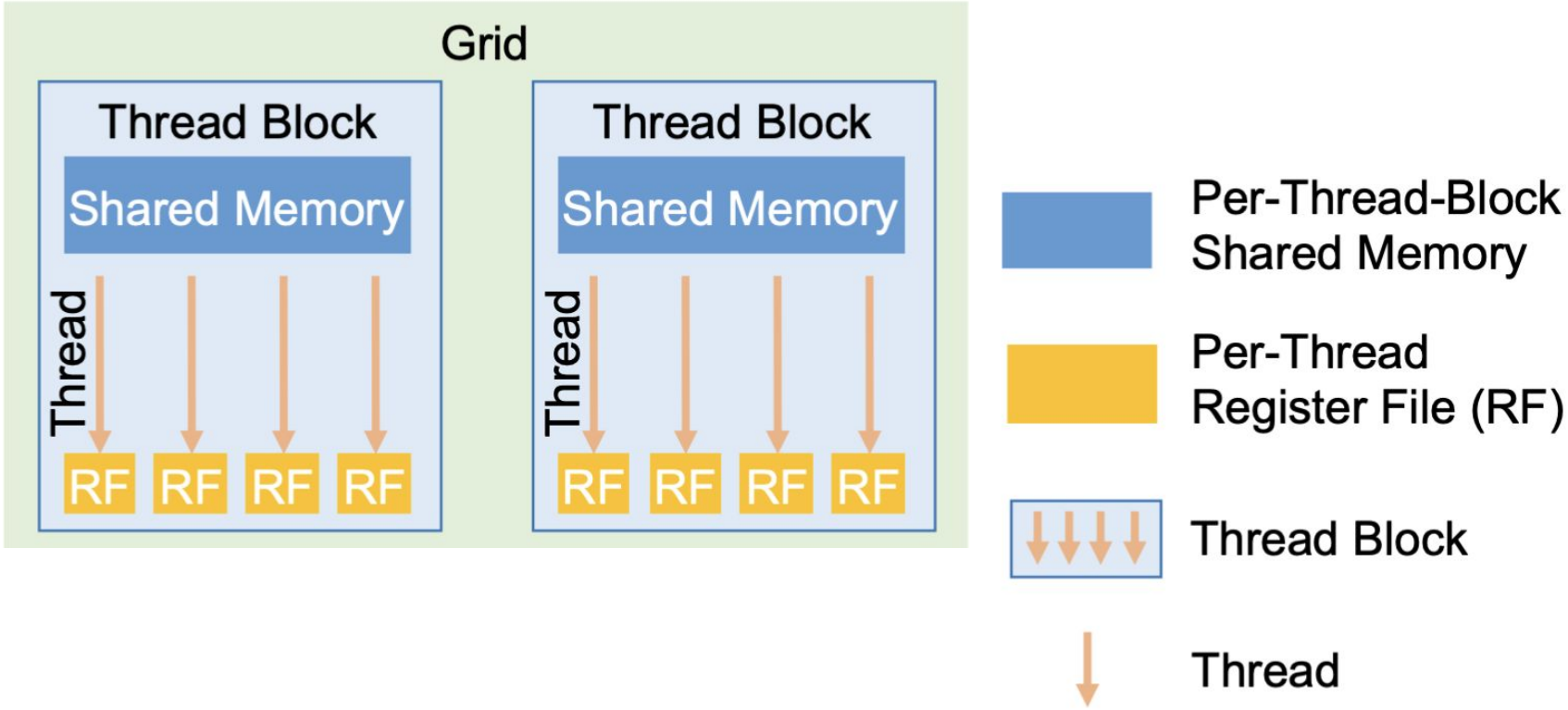


# GPU Compute + Memory Hierarchy

- **Thread blocks** → execute on streaming multiprocessors, use shared memory (fast)
- **Threads** → use per-thread register files (fastest)

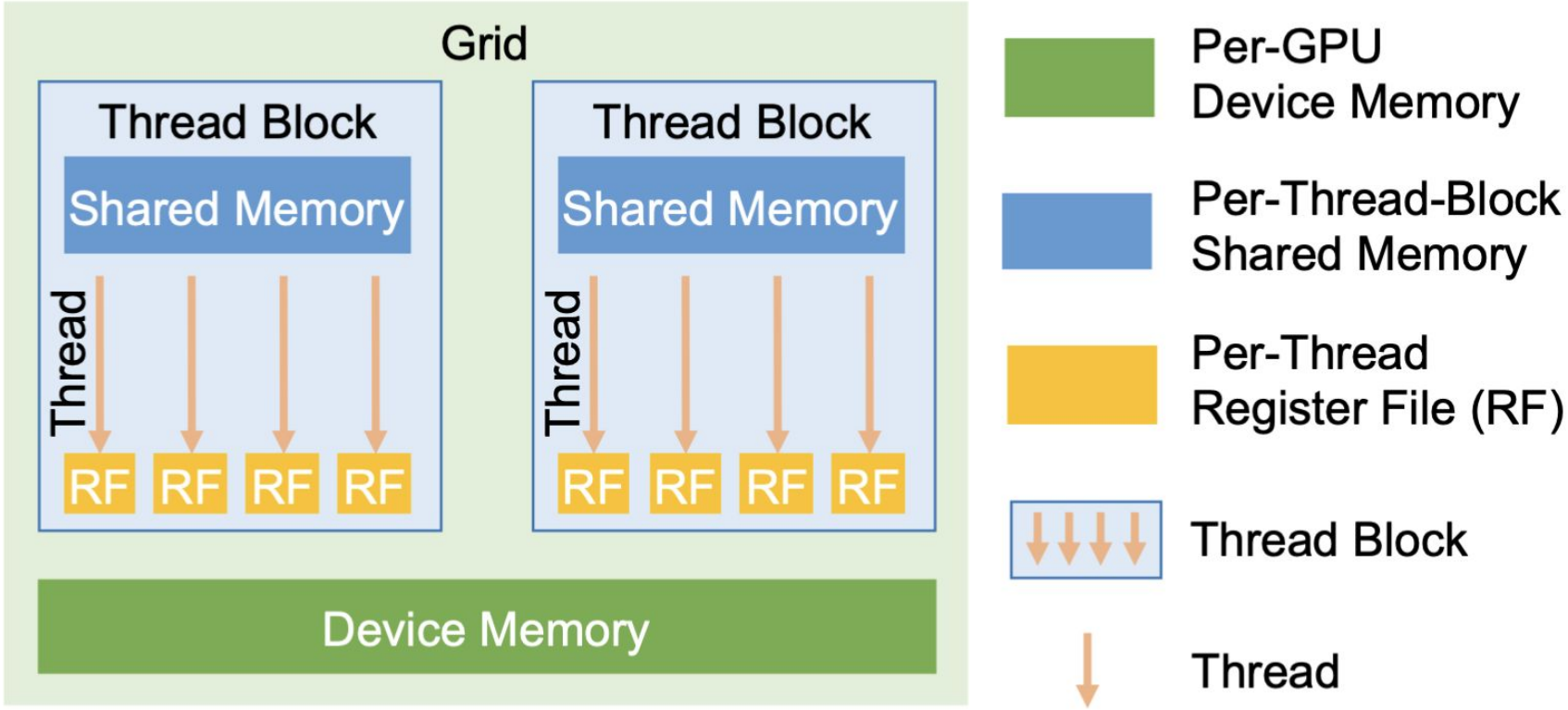
# GPU Compute + Memory Hierarchy



# GPU Compute + Memory Hierarchy

- **Thread blocks** → execute on streaming multiprocessors, use shared memory (fast)
- **Threads** → use per-thread register files (fastest)
- **Grid** → many thread blocks in parallel

# GPU Compute + Memory Hierarchy

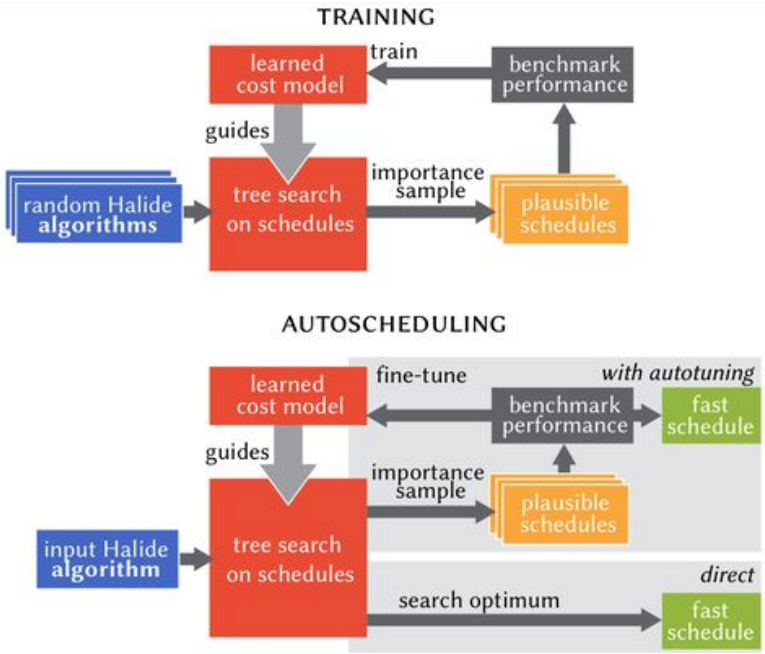


# GPU Compute + Memory Hierarchy

- **Thread blocks** → execute on streaming multiprocessors, use shared memory (fast)
- **Threads** → use per-thread register files (fastest)
- **Grid** → many thread blocks in parallel
- Device memory access is orders of magnitude more expensive than shared/register
- **Kernels** → execute on full GPU, read/write device memory (slowest)
- Good optimizations need to reason about all three levels together

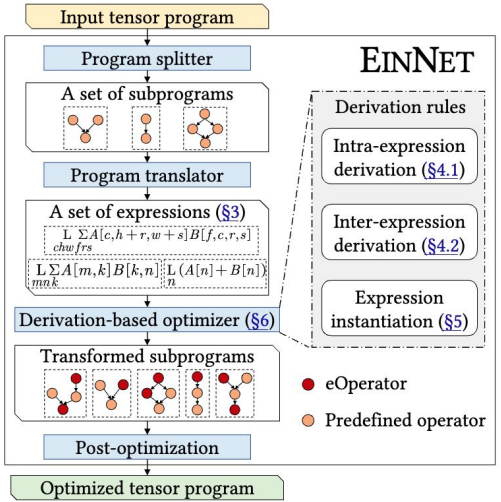
# What granularity does prior work optimize at

- **Schedule optimizers** (Halide, TVM, Ansor): fix the algorithm, search for *how* to execute it — can't change the math



# What granularity does prior work optimize at

- **Schedule optimizers** (Halide, TVM, Ansor): fix the algorithm, search for *how* to execute it — can't change the math
- **Algebraic optimizers** (TASO, PET, EINNET): transform the math, but treat kernels as black boxes — can't change internal kernel structure



# What granularity does prior work optimize at

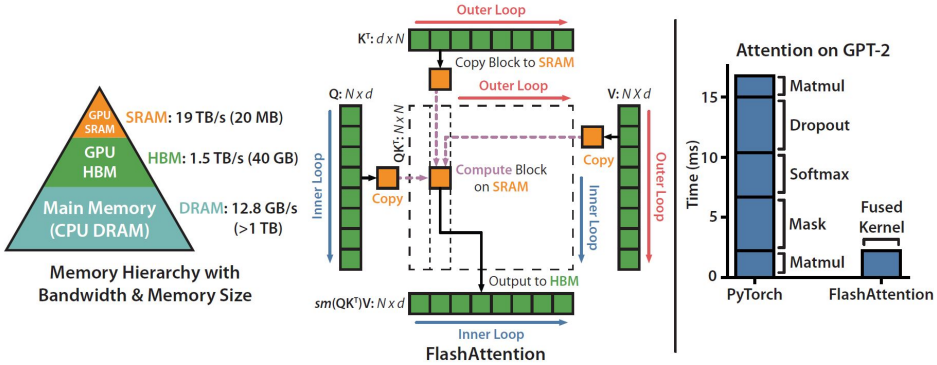
- **Schedule optimizers** (Halide, TVM, Ansor): fix the algorithm, search for *how* to execute it — can't change the math
- **Algebraic optimizers** (TASO, PET, EINNET): transform the math, but treat kernels as black boxes — can't change internal kernel structure
- Neither can introduce entirely new custom kernels
- Consequence: optimizations requiring coordinated changes across kernel/block/thread levels must be hand-written

# Motivating Example - Flash Attention

- Naive attention: separate kernels for Q×K, softmax, ×V → large intermediates written to device memory

# Motivating Example - Flash Attention

- Naive attention: separate kernels for Q×K, softmax, ×V → large intermediates written to device memory
- FlashAttention (hand-designed): reorders softmax decomposition, tiles across thread blocks, keeps intermediates in shared memory, fuses into one kernel
- Requires algebraic rewrites + schedule changes + new custom kernel design simultaneously
- **No existing automated framework can discover this**



# Mirage Baselines

- **PyTorch**: cuDNN/cuBLAS with manual dispatch rules — the standard baseline (KernelBench style)
- **TensorRT / TensorRT-LLM**: NVIDIA's inference optimizer, hand-tuned kernels for attention etc.
- **Triton**: schedule-based compiler, Python-like DSL → optimized GPU code (far from the hardware so performance is capped)
- **FlashAttention / FlashDecoding**: expert-designed custom kernels for attention (prefill vs. incremental decoding) — expert-optimized gold standard

# Looking forward

Can LLMs write superhuman kernels? We’re close

# Looking forward

Can LLMs write superhuman kernels? We’re close

Can LLMs discover mathematical improvements for kernels? Further away

# Looking forward

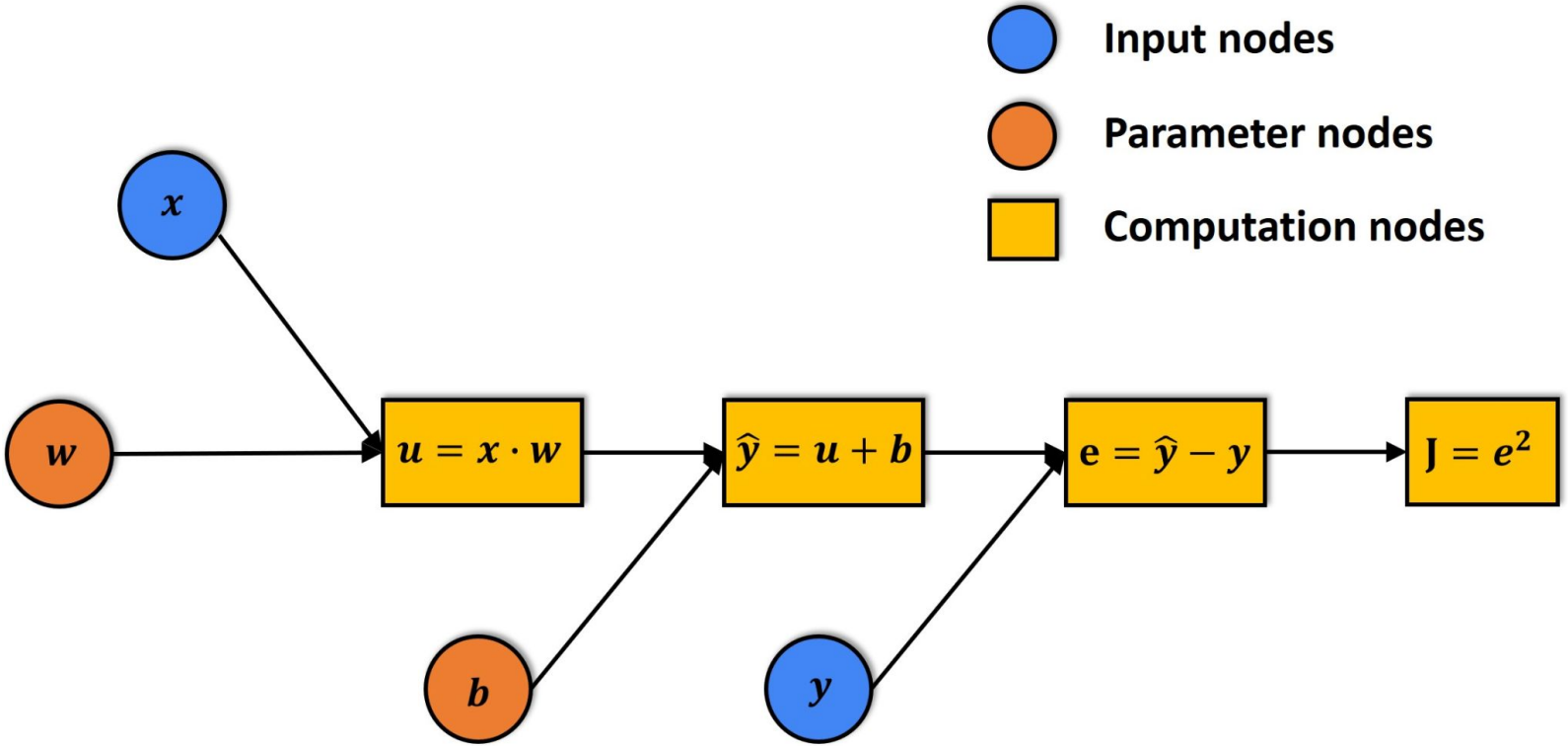
Can LLMs write superhuman kernels? We're close

Can LLMs discover mathematical improvements for kernels? Further away

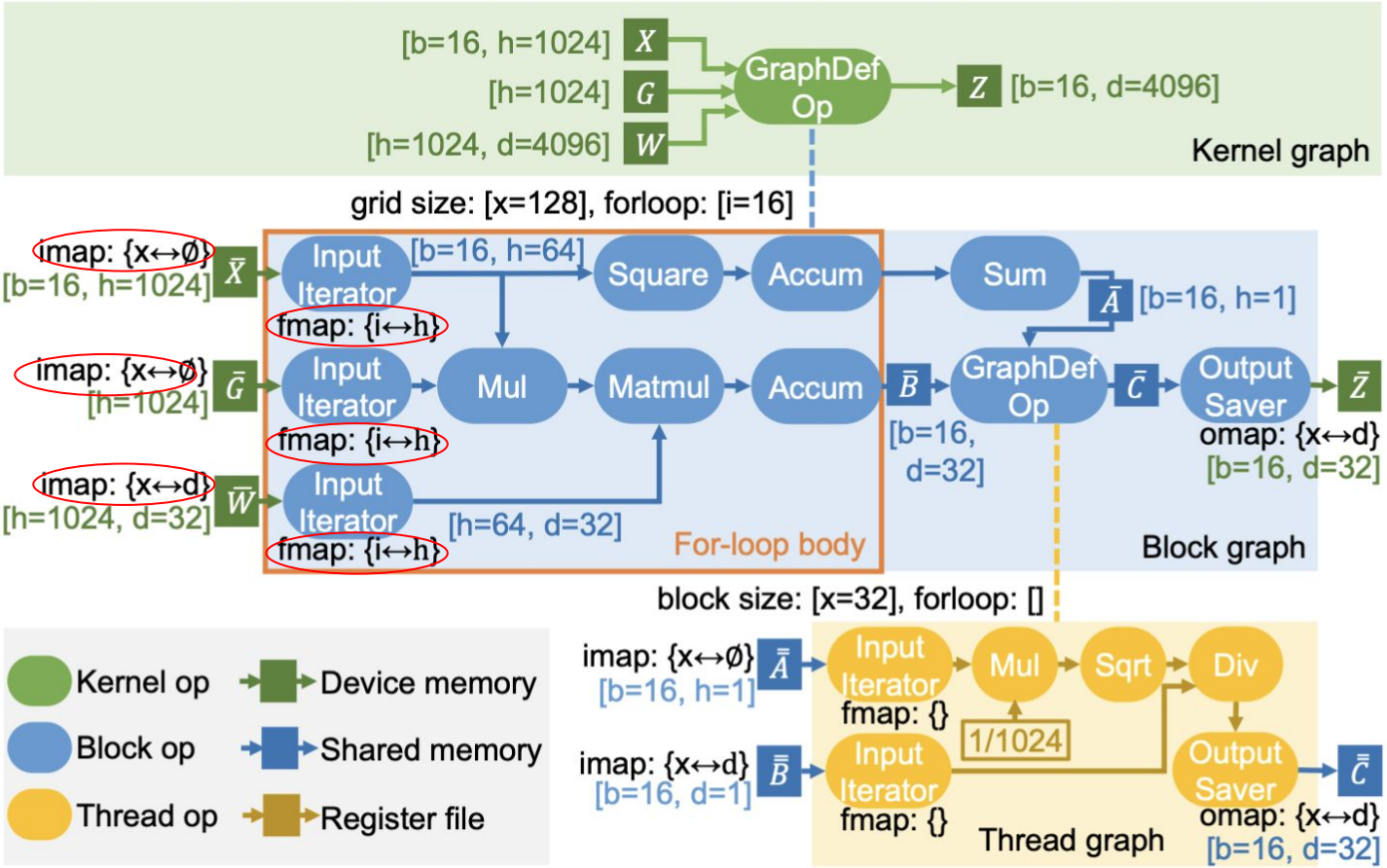
How can we put these together? This will make a lot of impact

# How to optimize over both algorithmic and scheduling transformations?

# Computational graphs

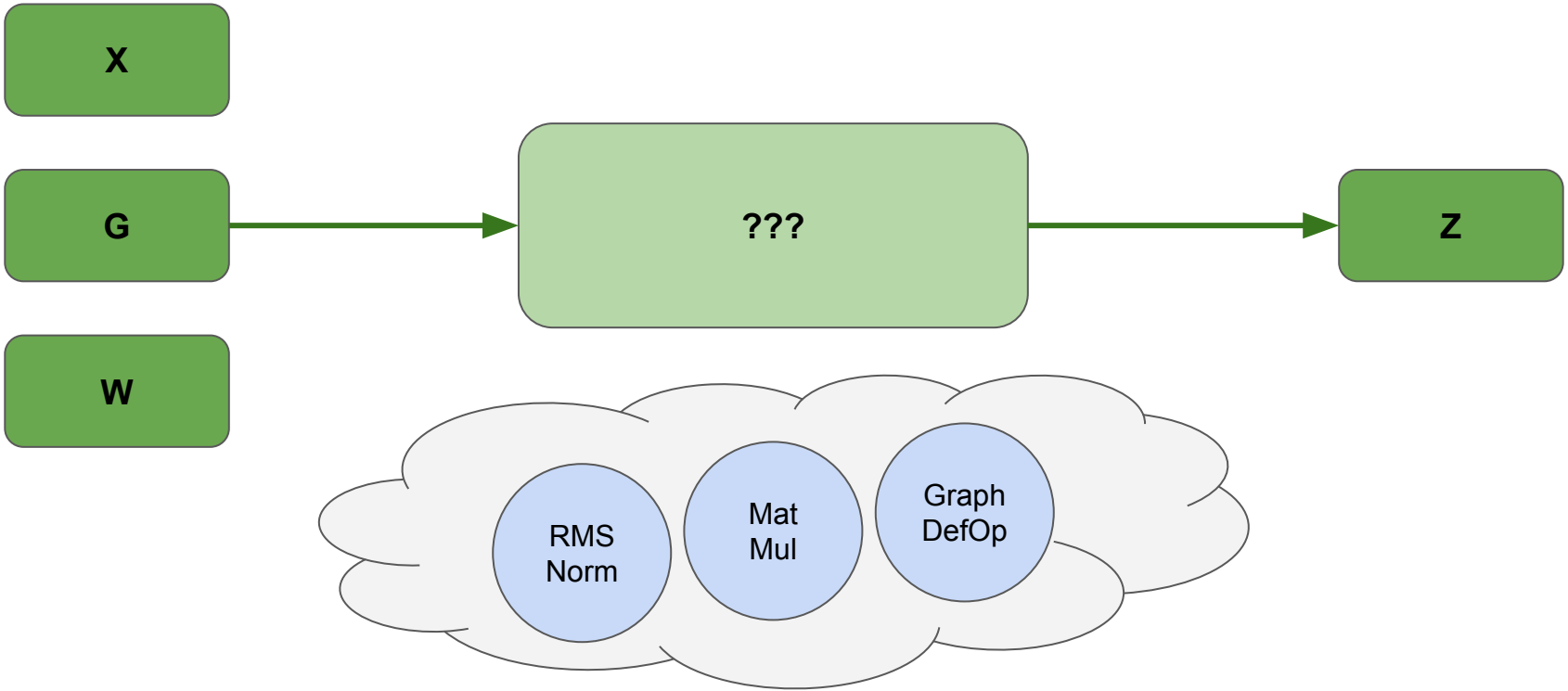


# $\mu$-graph

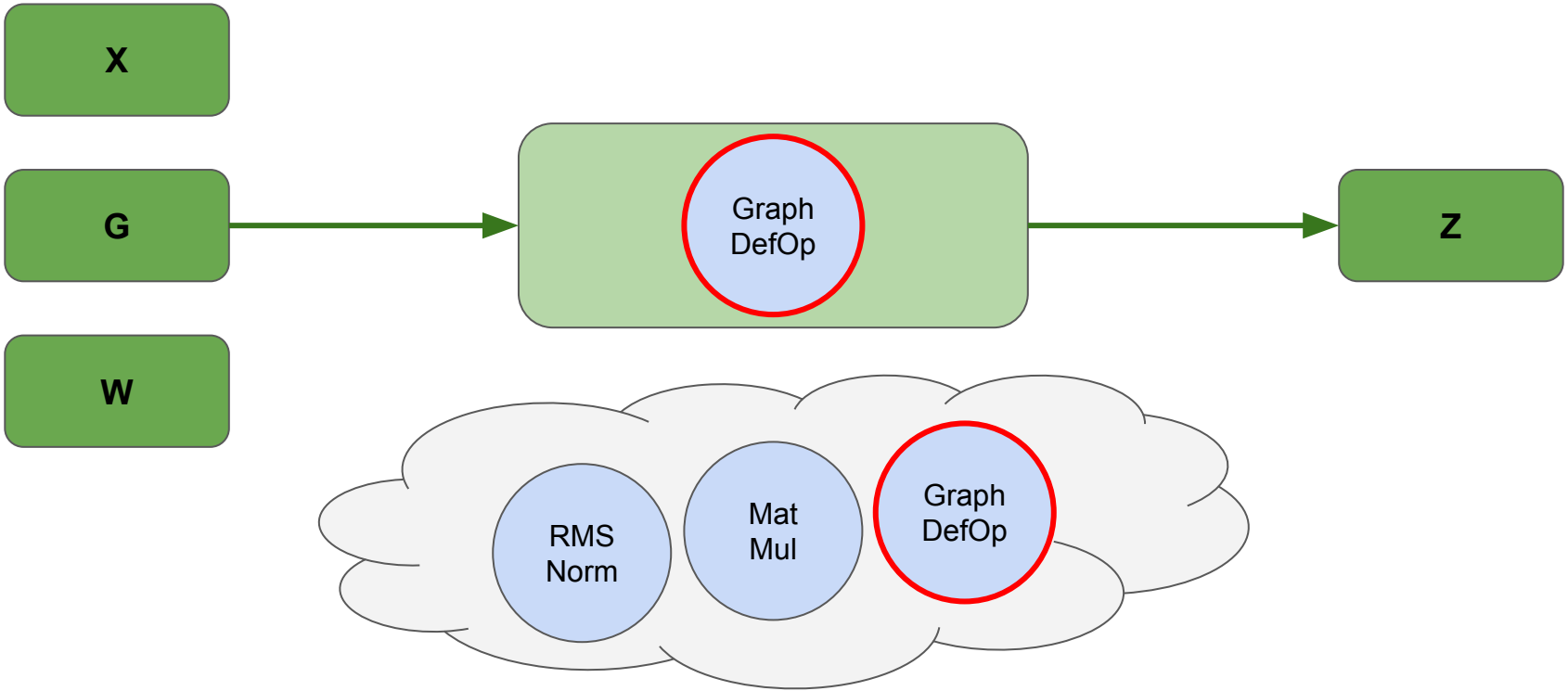

[b=16, h=1024]
X
[h=1024]
G
[h=1024, d=4096]
W
GraphDef Op
Z
[b=16, d=4096]
Kernel graph
grid size: [x=128], forloop: [i=16]
imap: {x↔∅}
[b=16, h=1024]
Input Iterator
[b=16, h=64]
Square
Accum
Sum
[b=16, h=1]
fmap: {i↔h}
imap: {x↔∅}
[h=1024]
Input Iterator
Mul
Matmul
Accum
GraphDef Op
Output Saver
omap: {x↔d}
[b=16, d=32]
[b=16, d=32]
fmap: {i↔h}
imap: {x↔d}
[h=1024, d=32]
Input Iterator
[h=64, d=32]
fmap: {i↔h}
For-loop body
Block graph
block size: [x=32], forloop: []
Kernel op
Block op
Thread op
Device memory
Shared memory
Register file
imap: {x↔∅}
[b=16, h=1]
Input Iterator
Mul
Sqrt
Div
fmap: {}
1/1024
imap: {x↔d}
[b=16, d=1]
Input Iterator
Output Saver
fmap: {}
omap: {x↔d}
[b=16, d=32]
Thread graph

# Demo: Tensor Partitioning

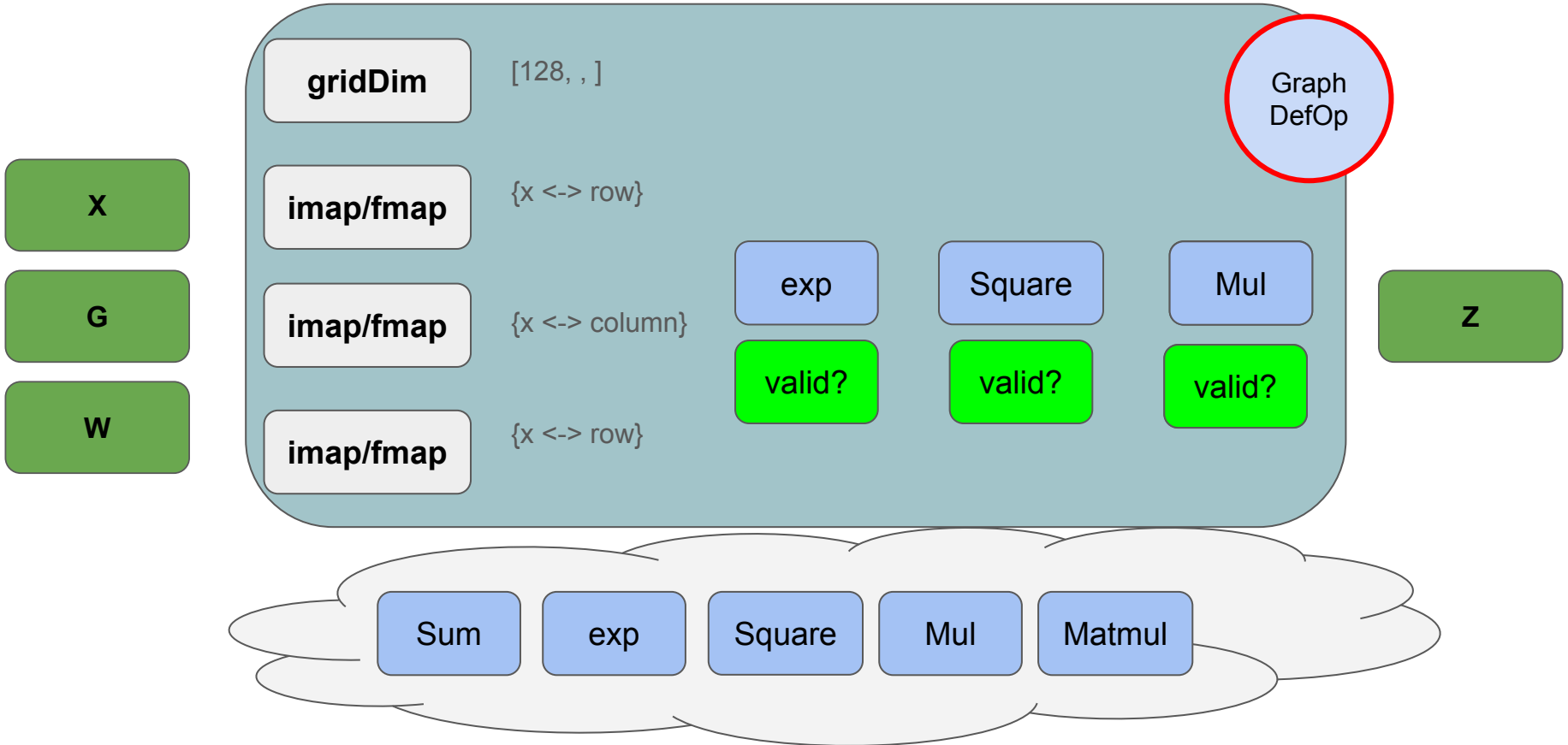
# How to build a $\mu$-graph



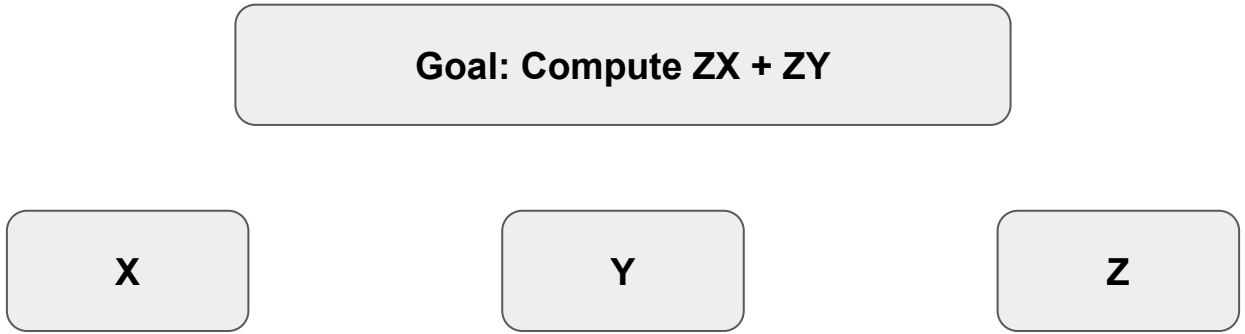
# How to build a $\mu$-graph



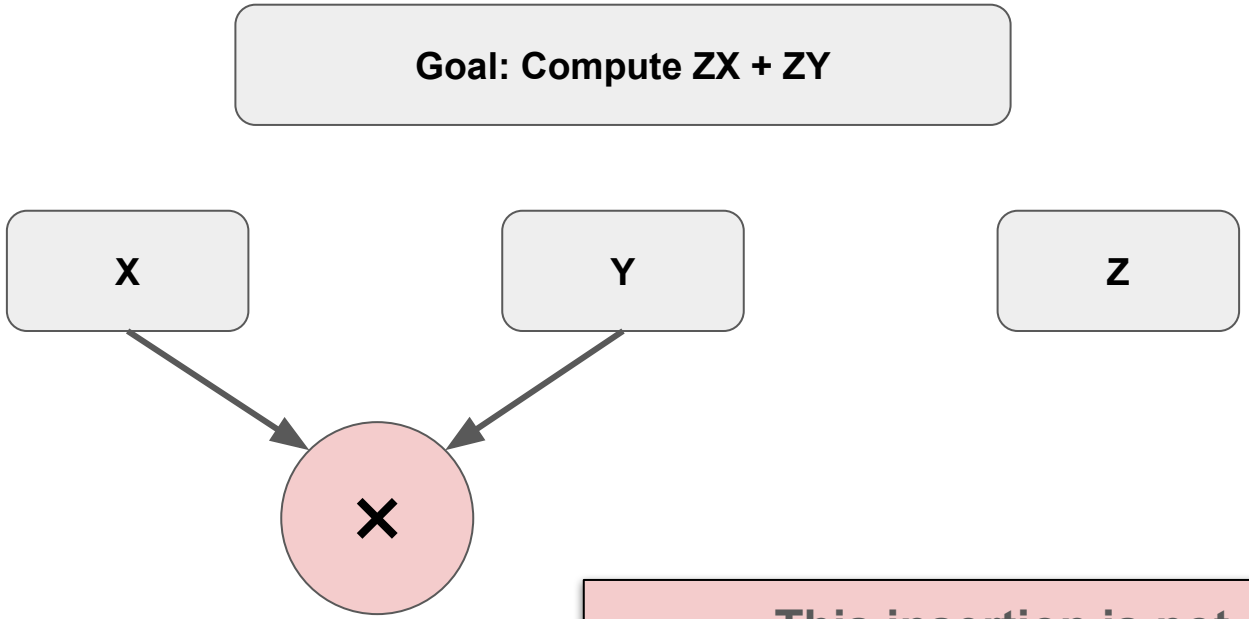
# How to build a $\mu$-graph



# How to prune a $\mu$-graph?



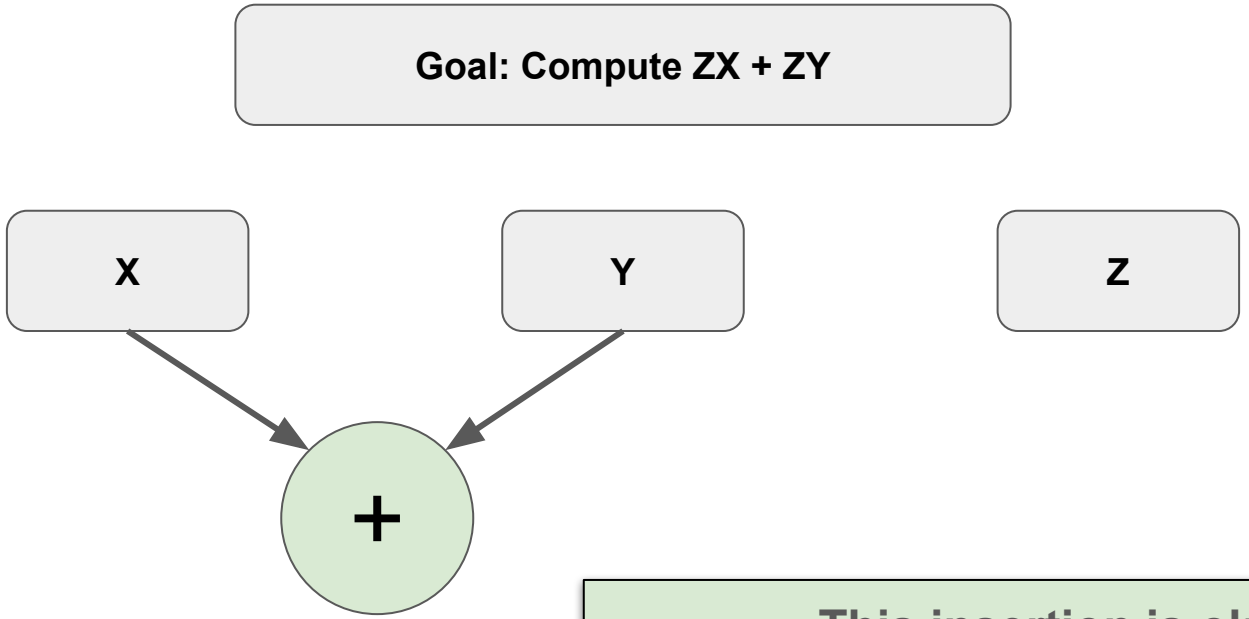
# How to prune a $\mu$-graph?




**This insertion is not ok:**

(X*Y) is **not** a sub-expression of ZX + ZY

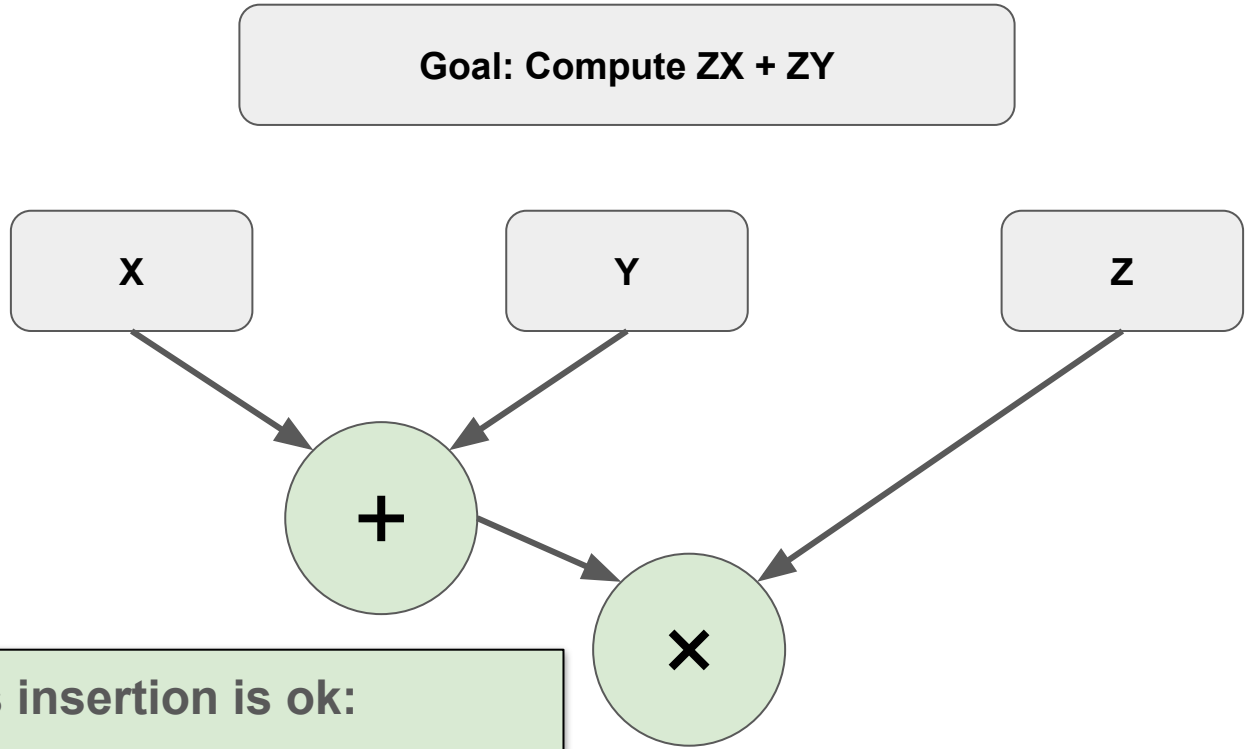
# How to prune a $\mu$-graph?




**This insertion is ok:**

(X + Y) is a sub-expression of ZX + ZY

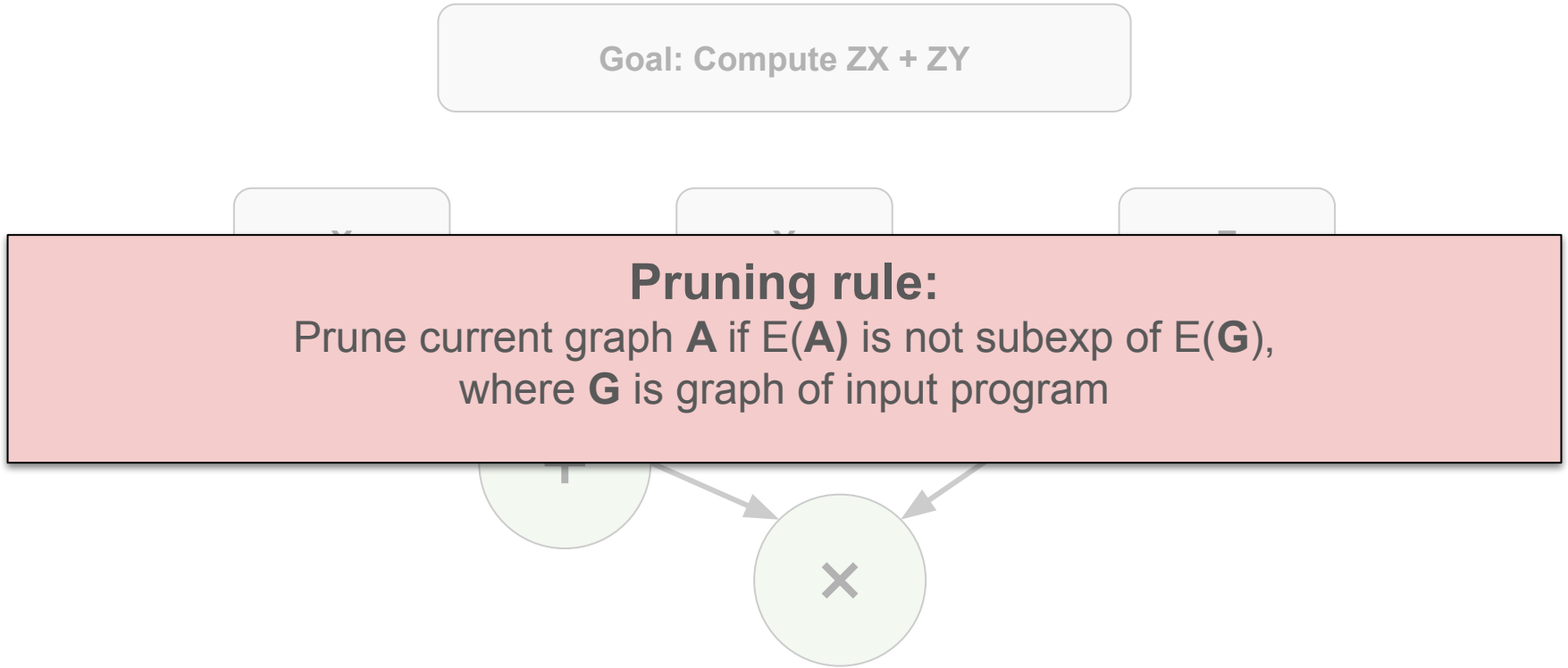
# How to prune a $\mu$-graph?




**This insertion is ok:**

Z(X + Y) is equal to ZX + ZY

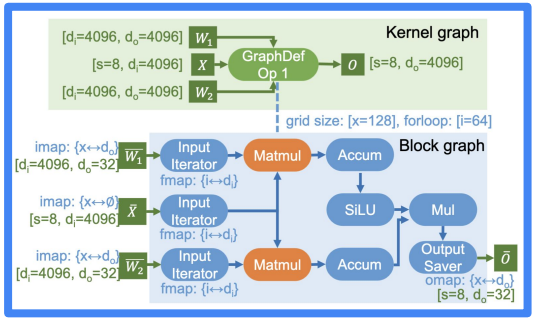
# How to prune a $\mu$-graph?

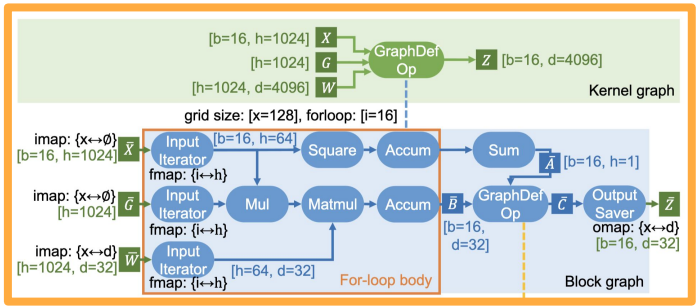



**Pruning rule:**
Prune current graph **A** if E(**A**) is not subexp of E(**G**),
where **G** is graph of input program

# *μ*Graph Verification

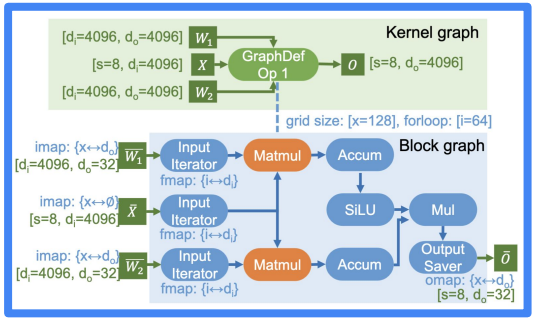
# μGraph Verification: Probabilistic Equivalence

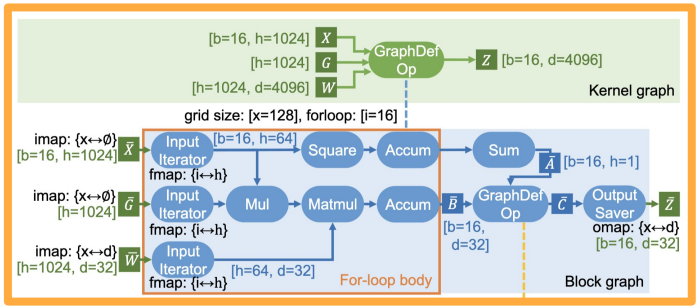


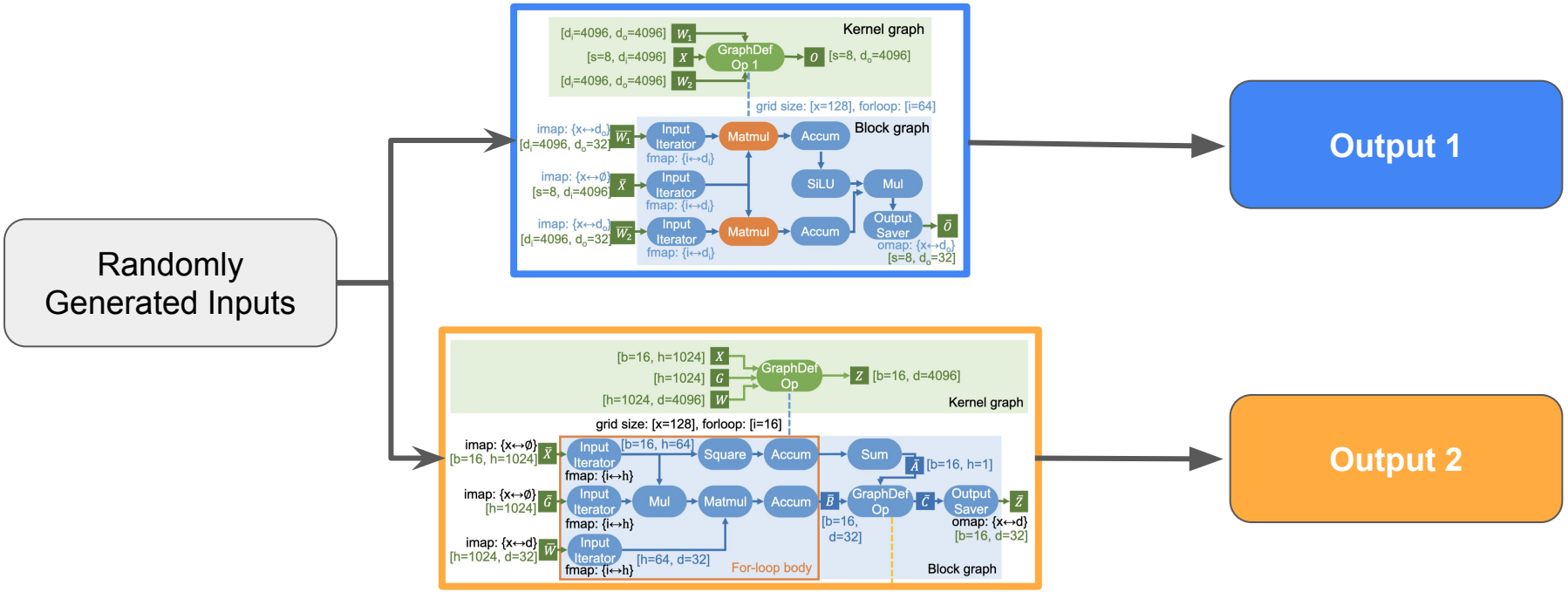
# μGraph Verification: Probabilistic Equivalence

Randomly Generated Inputs

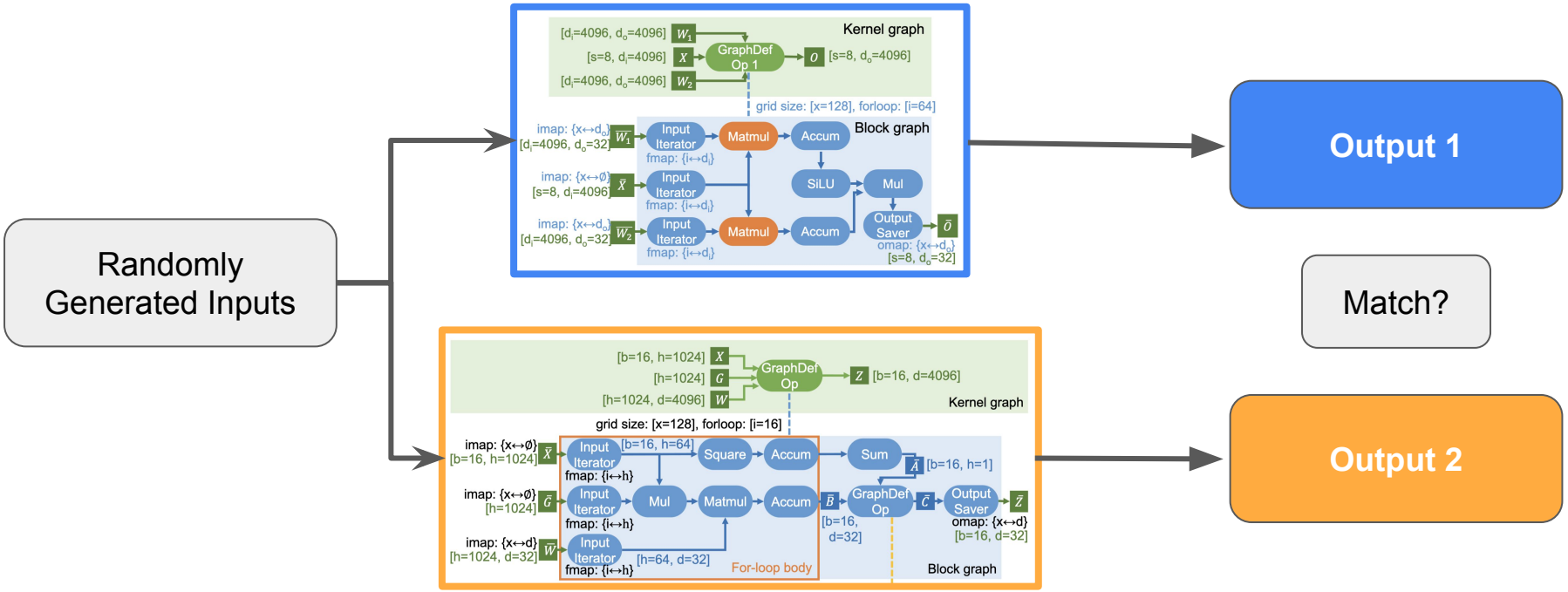


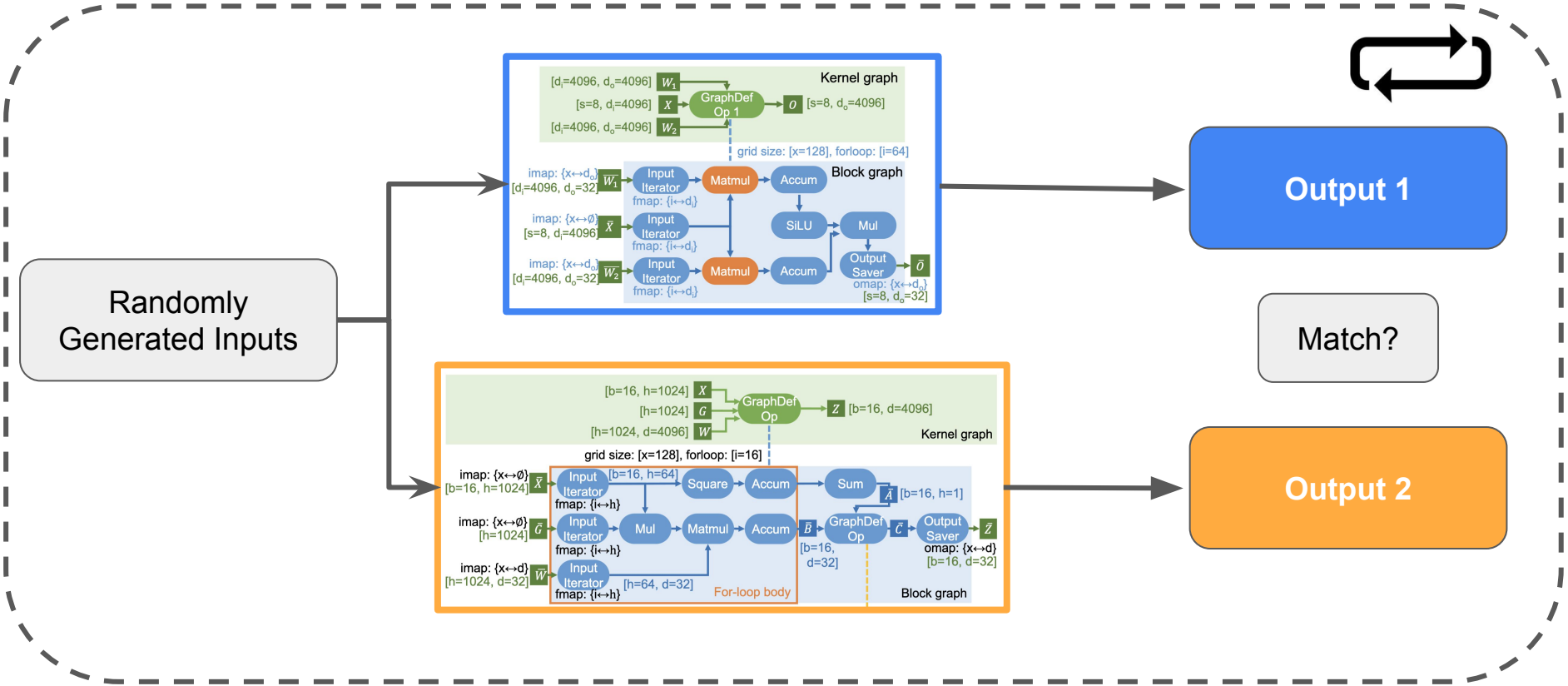
# μGraph Verification: Probabilistic Equivalence



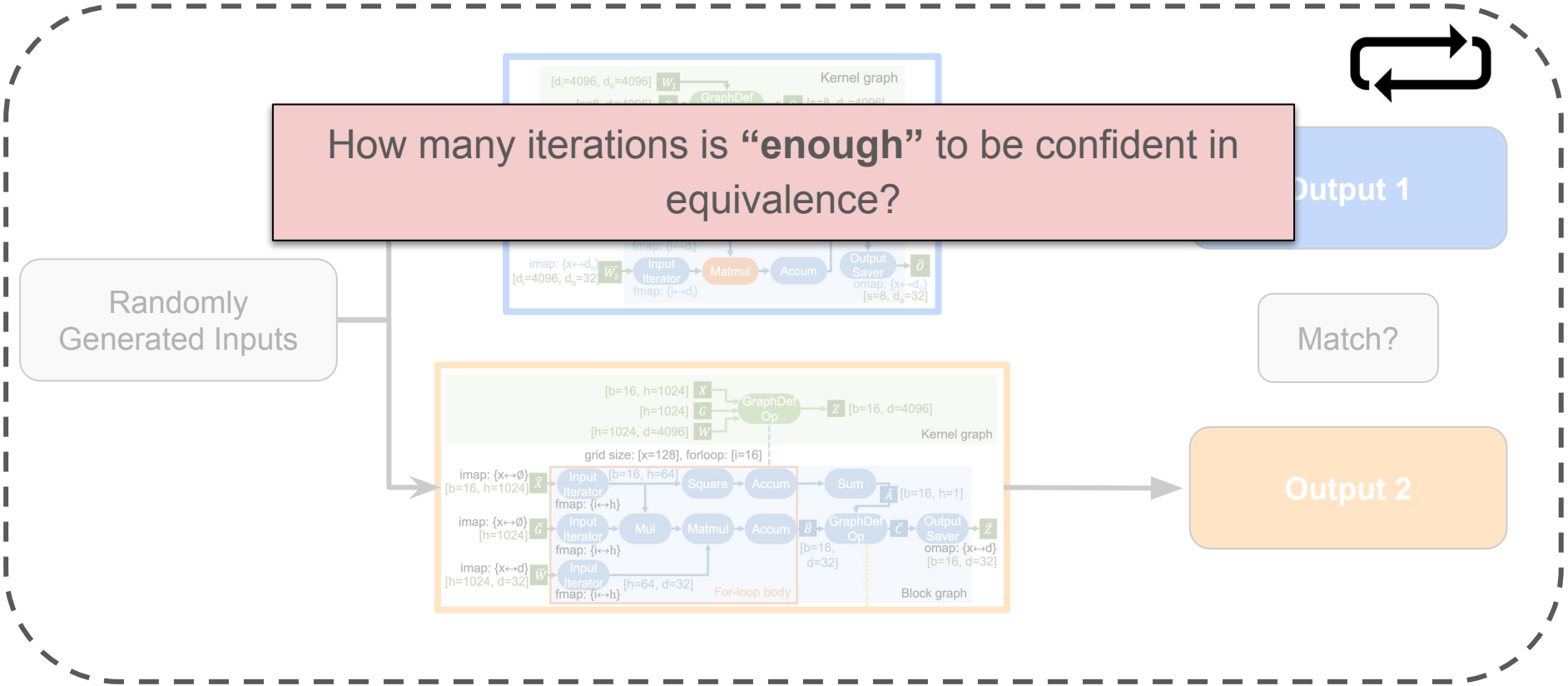
# μGraph Verification: Probabilistic Equivalence



# μGraph Verification: Probabilistic Equivalence



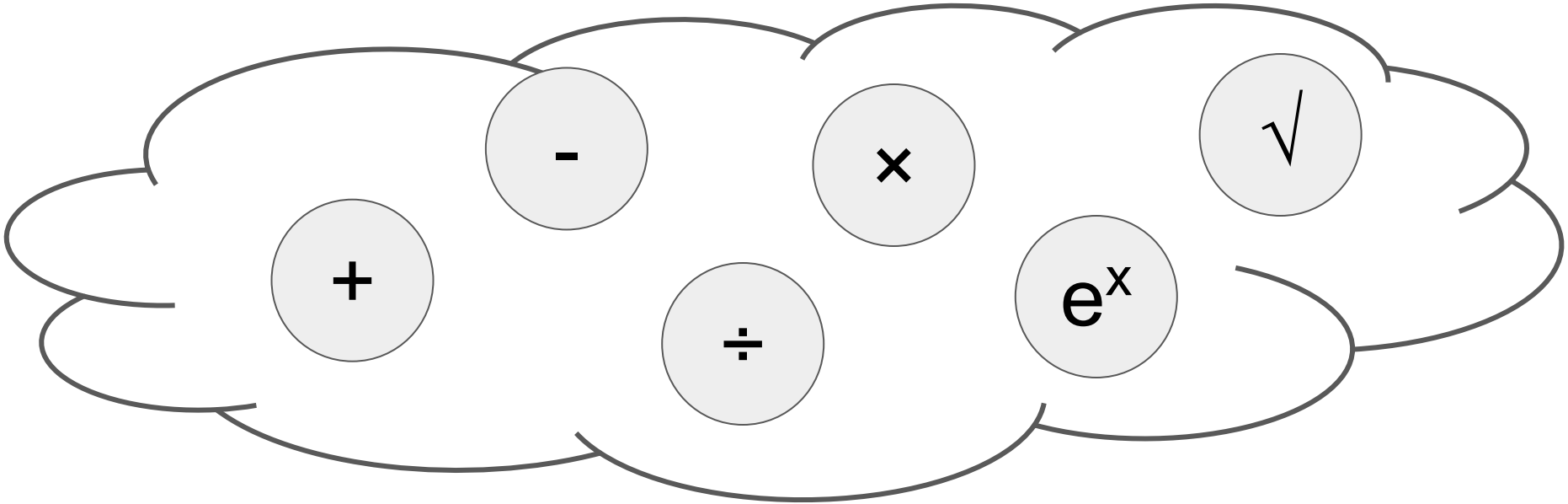
# μGraph Verification: Probabilistic Equivalence



# *μ*Graph Verification: Probabilistic Equivalence

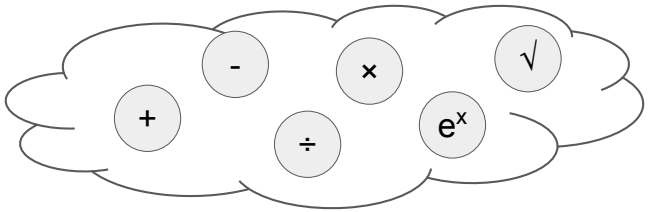
**Restricting to LAX *μ*Graphs**

Multi-linear ops, division, and (at most one per execution path) exponentiation



# *μ*Graph Verification: Probabilistic Equivalence

**LAX *μ*Graphs can be universally formalized**

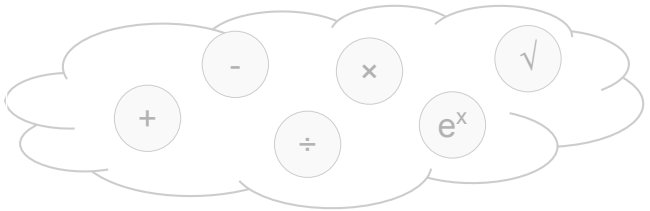



$$\frac{\sum_{i=1}^{k} f_i \exp(g_i/h_i)}{\sum_{i=1}^{k'} f'_i \exp(g'_i/h'_i)}$$

# μGraph Verification: Probabilistic Equivalence

**LAX μGraphs can be universally formalized**

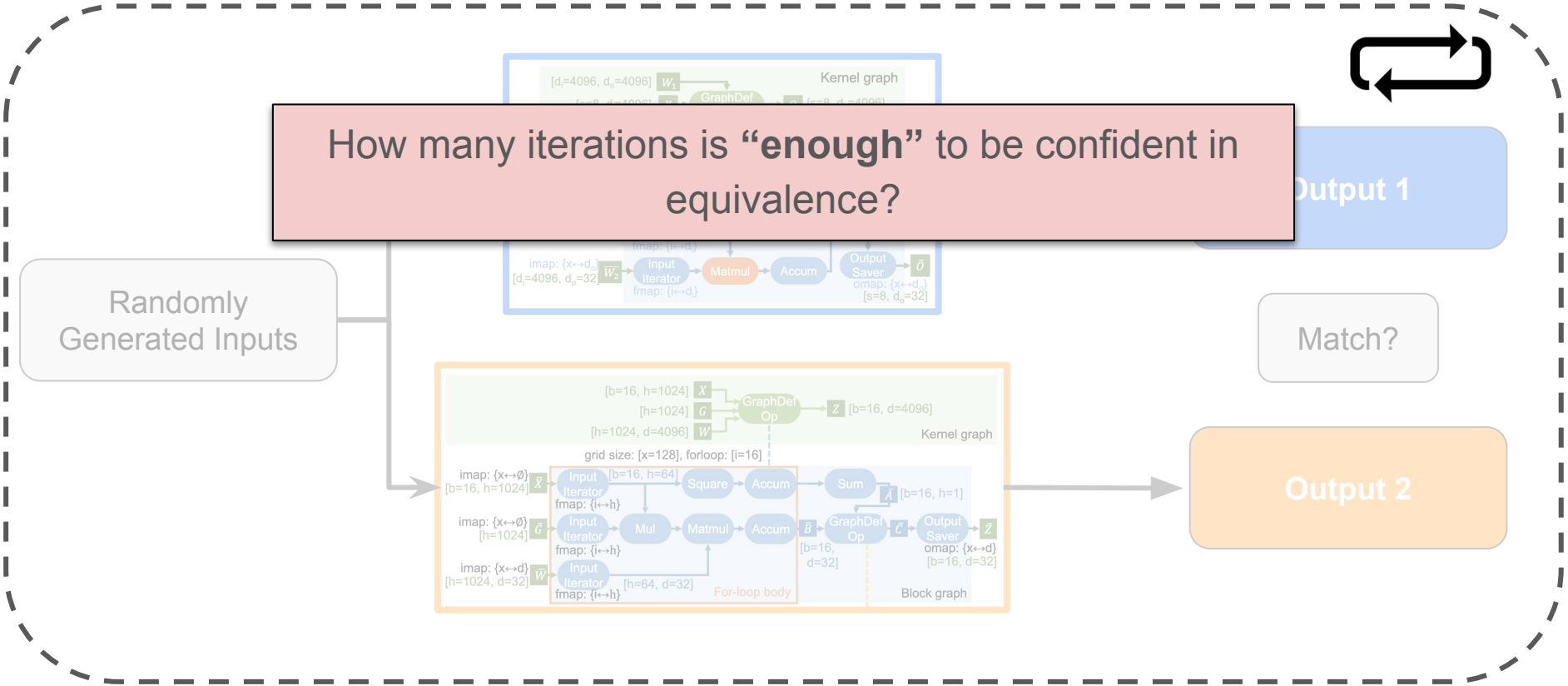
**Can calculate probability of random inputs resulting in the same output**



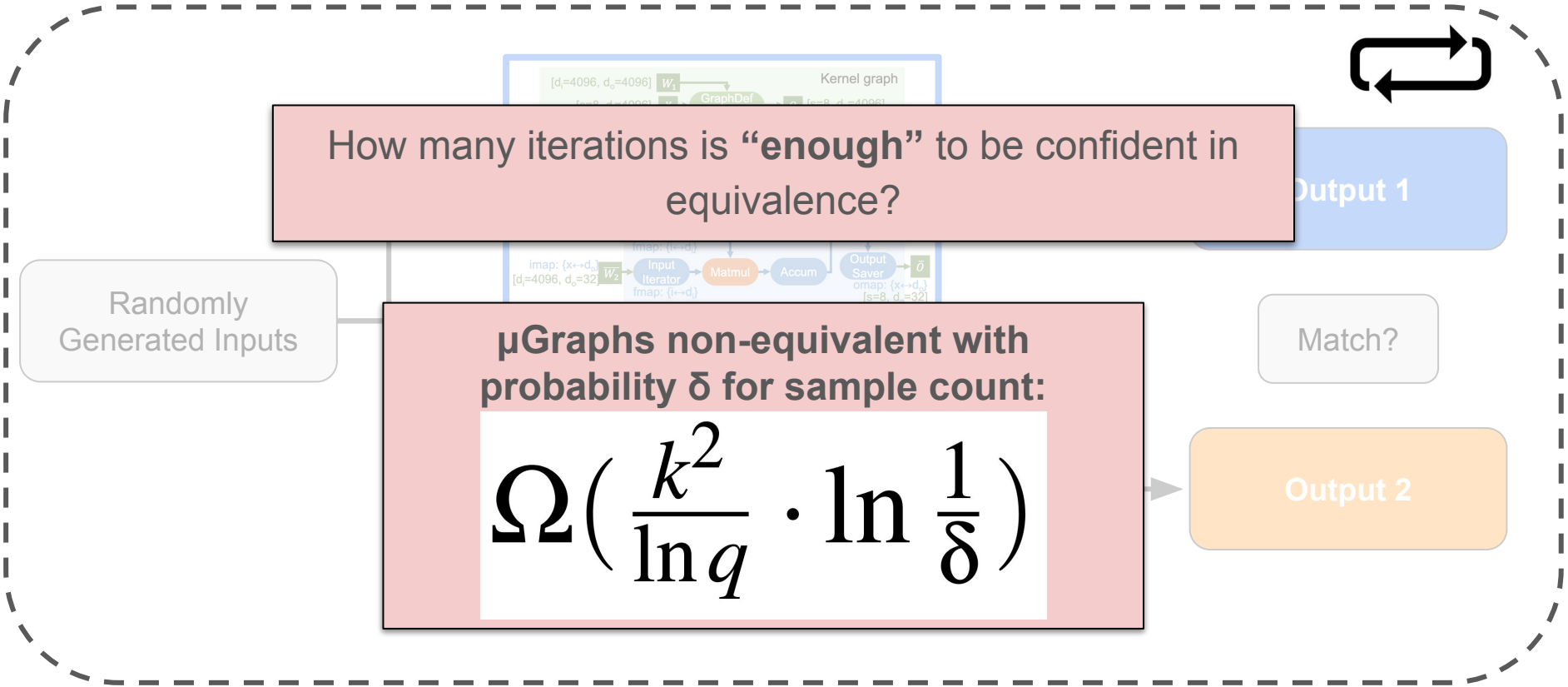

$$\frac{\sum_{i=1}^{k} f_i \exp(g_i/h_i)}{\sum_{i=1}^{k'} f'_i \exp(g'_i/h'_i)}$$

$$\Pr_{(\vec{u},\vec{v},\omega)\leftarrow\mathbb{Z}_p^N\times\mathbb{Z}_q^N\times\mathcal{G}}\left[\frac{\sum_{i=1}^{k} f_i(\vec{u})\omega^{g_i(\vec{v})/h_i(\vec{v})}}{\sum_{i=1}^{k'} f'_i(\vec{u})\omega^{g'_i(\vec{v})/h'_i(\vec{v})}}\right] \leq 8dk^4/q + q^{-1/k^2}$$

# μGraph Verification: Probabilistic Equivalence

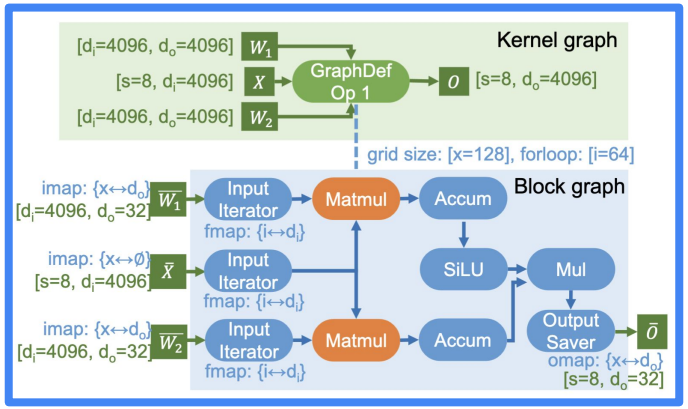


# μGraph Verification: Probabilistic Equivalence



# *μ*Graph Optimizations

# μGraph Optimization: Overall Flow



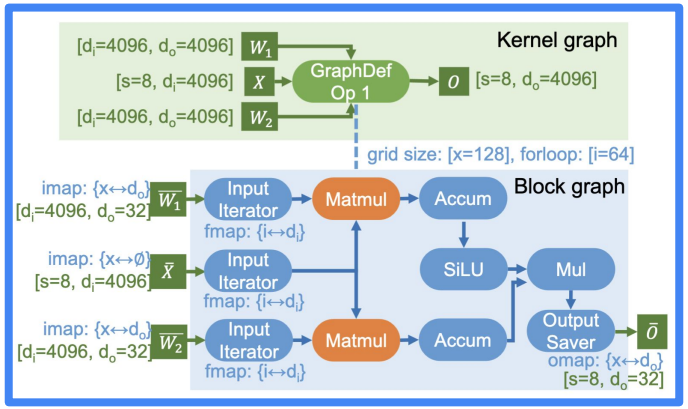

1. Block-Level Fusion

3. Kernel Memory Planning

2. Tensor Layout Planning

4. Thread Operation Scheduling

# μGraph Optimization: Overall Flow



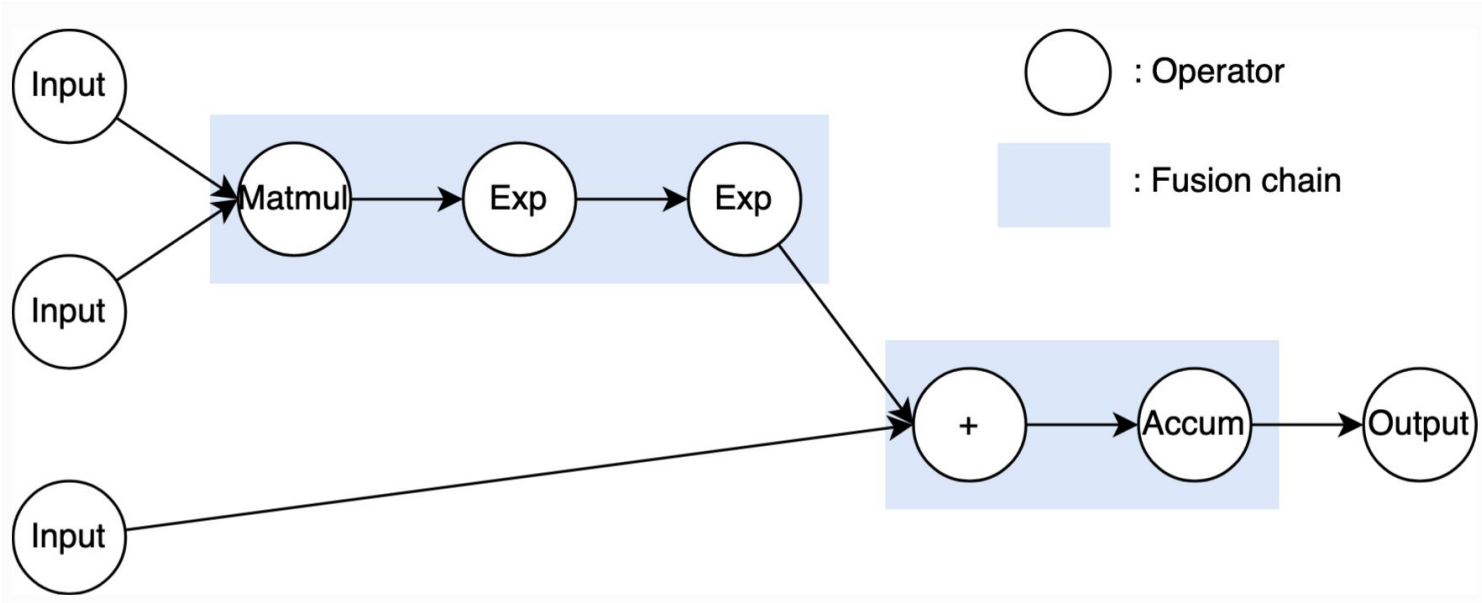

1. Block-Level Fusion

3. Kernel Memory Planning

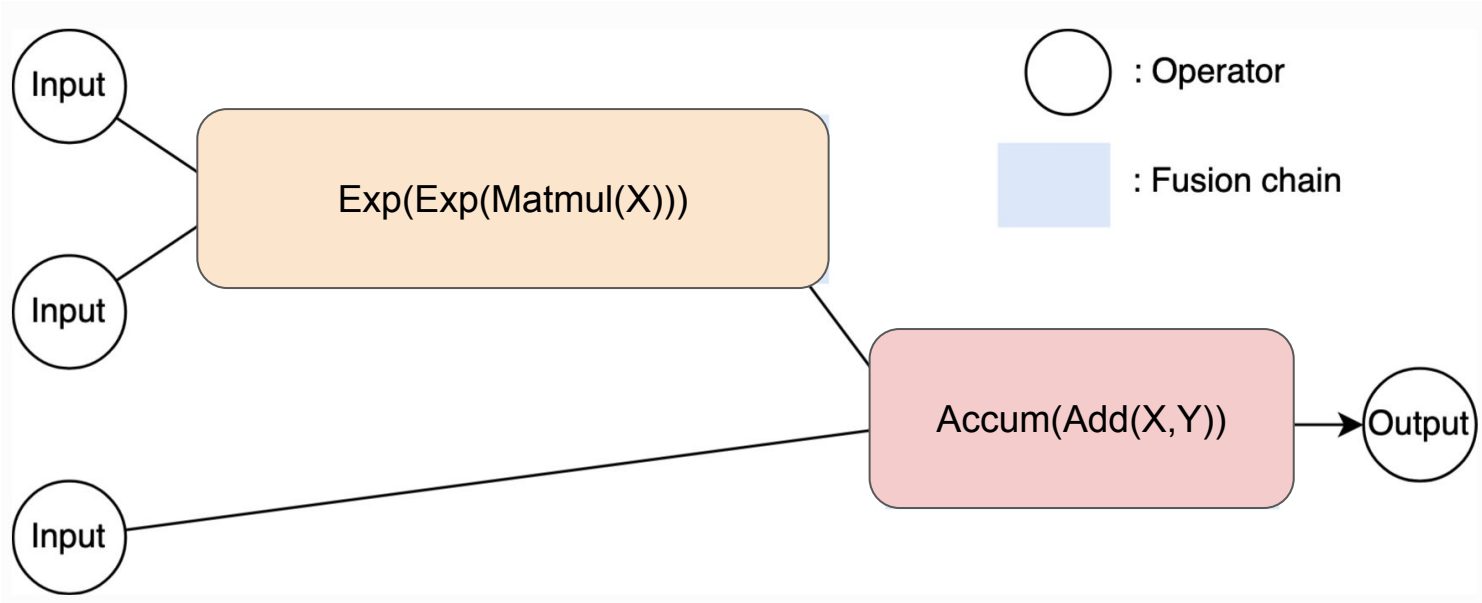
2. Tensor Layout Planning

4. Thread Operation Scheduling

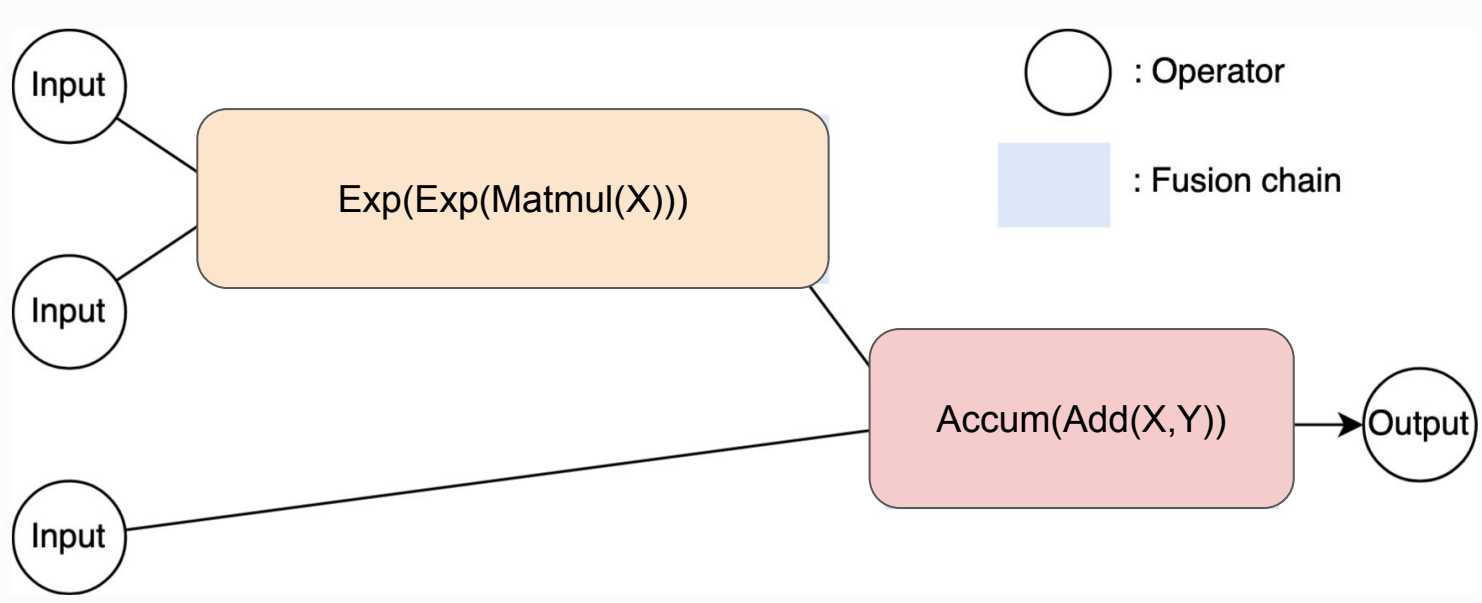
# $\mu$Graph Optimization: Block-Level Fusion



# $\mu$Graph Optimization: Block-Level Fusion

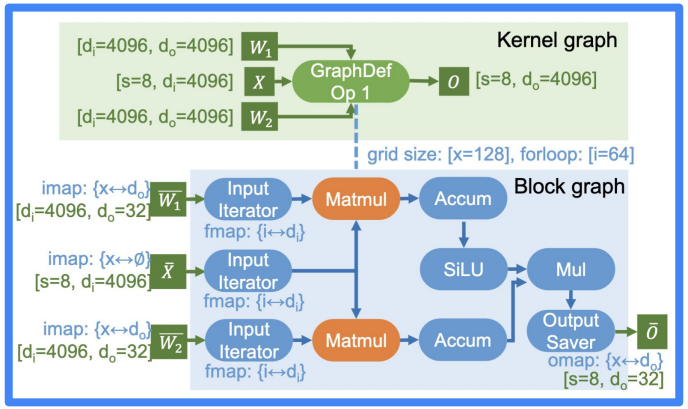


# μGraph Optimization: Block-Level Fusion




**Maximize data reuse by fusing trailing (unary) operations**

# μGraph Optimization: Overall Flow



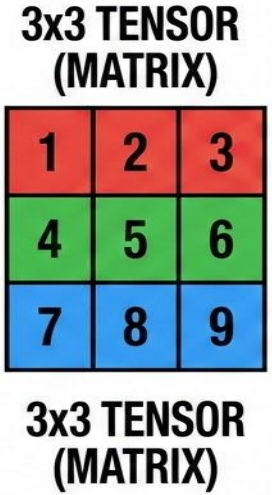

1. Block-Level Fusion

3. Kernel Memory Planning

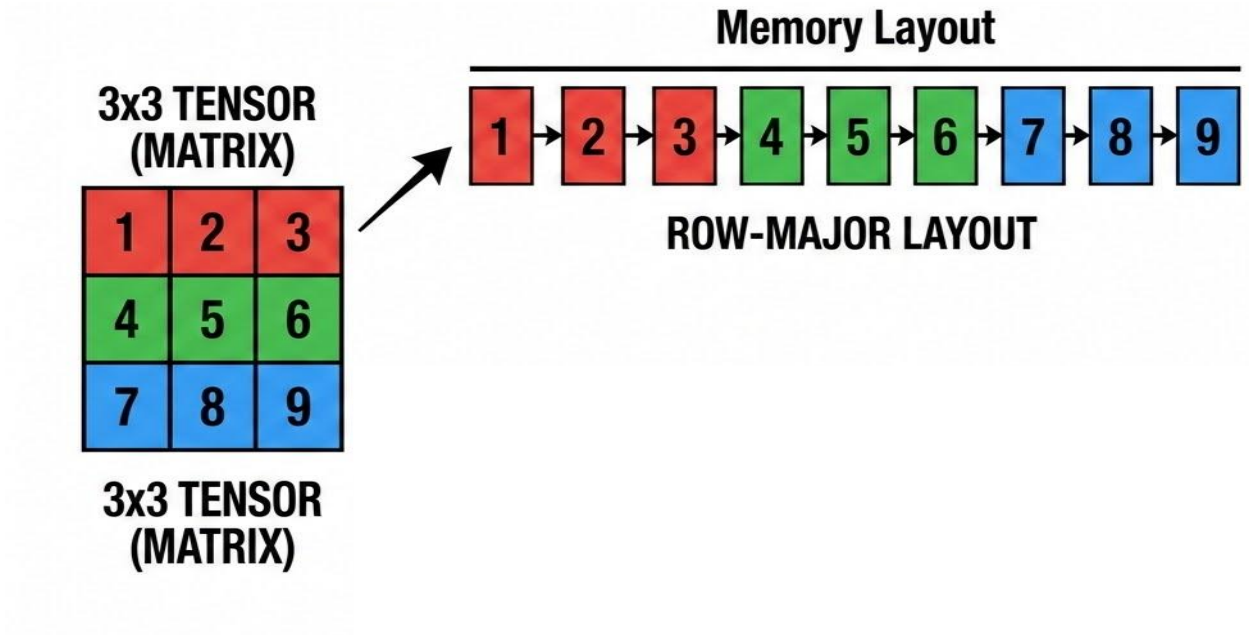
2. Tensor Layout Planning

4. Thread Operation Scheduling

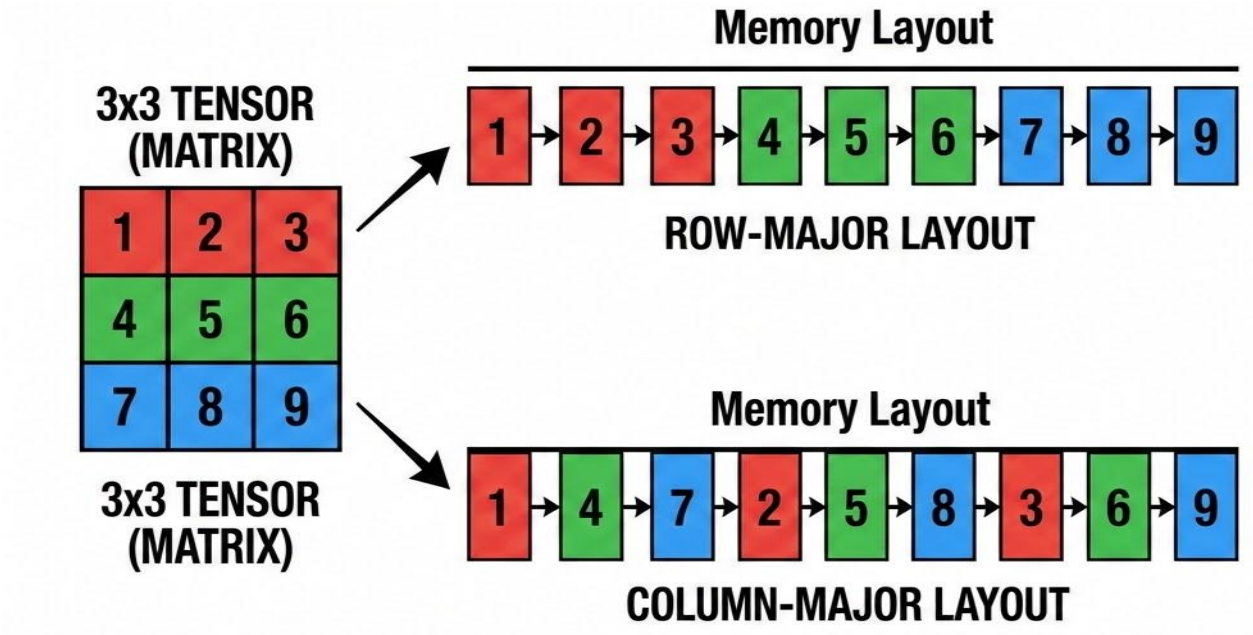
# $\mu$Graph Optimization: Tensor Layouts



# $\mu$Graph Optimization: Tensor Layouts



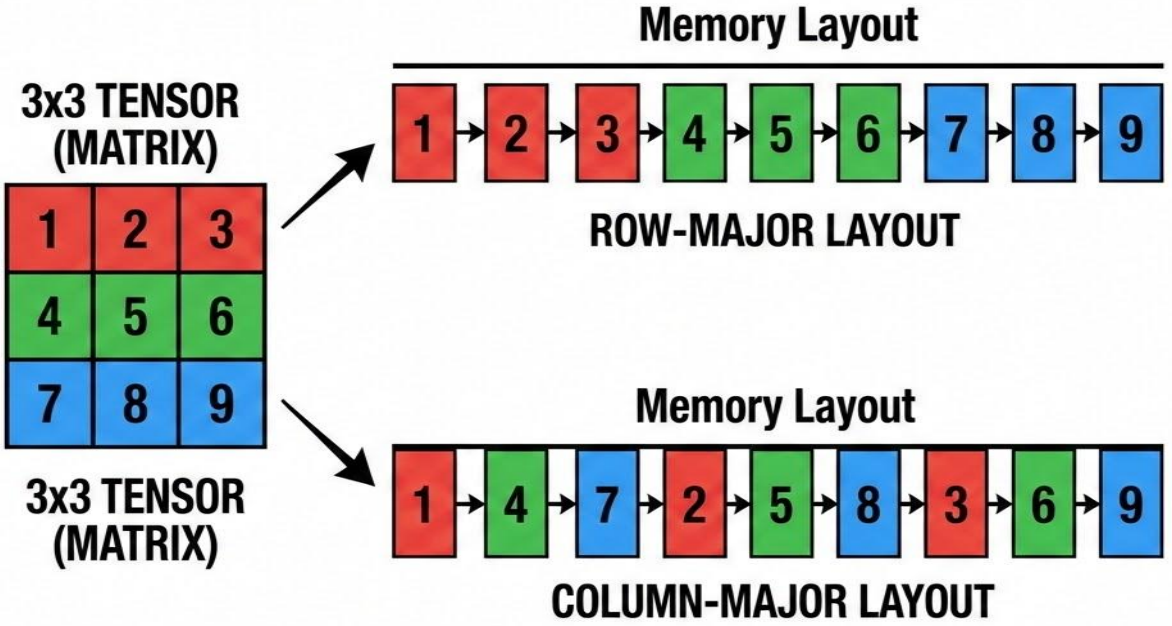
# μGraph Optimization: Tensor Layouts



# $\mu$Graph Optimization: Tensor Layouts

Key Factors:

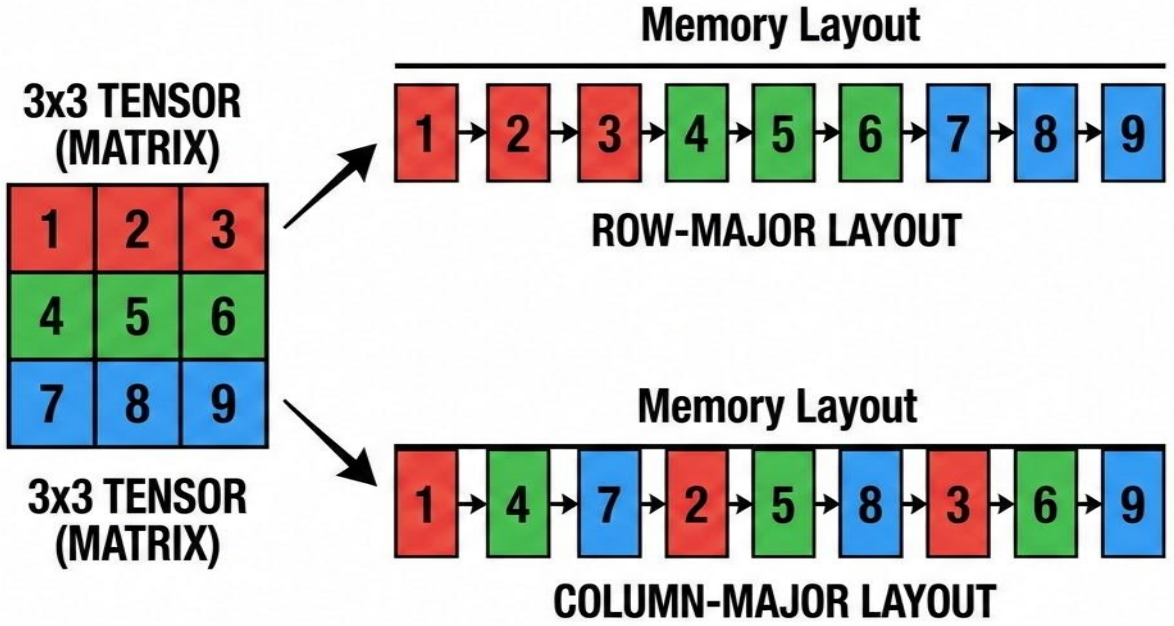
Data Transfers
Concurrent Accesses
Arithmetic Operations
…



# $\mu$Graph Optimization: Tensor Layouts

Key Factors:

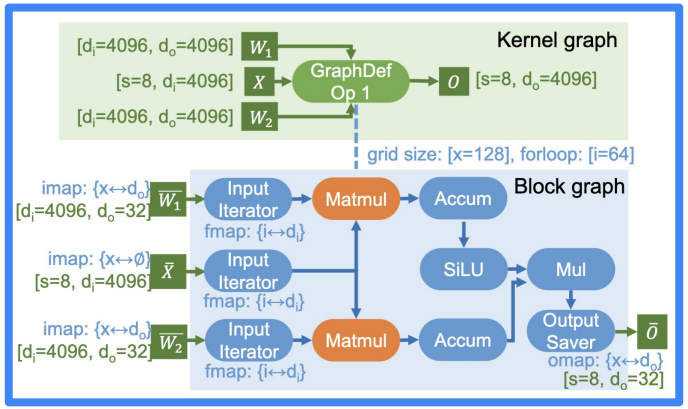
Data Transfers
Concurrent Accesses
Arithmetic Operations
…




**Increase performance via optimal leading dimension choices**

(Use an ILP solver to calculate optimal choices)

# μGraph Optimization: Overall Flow



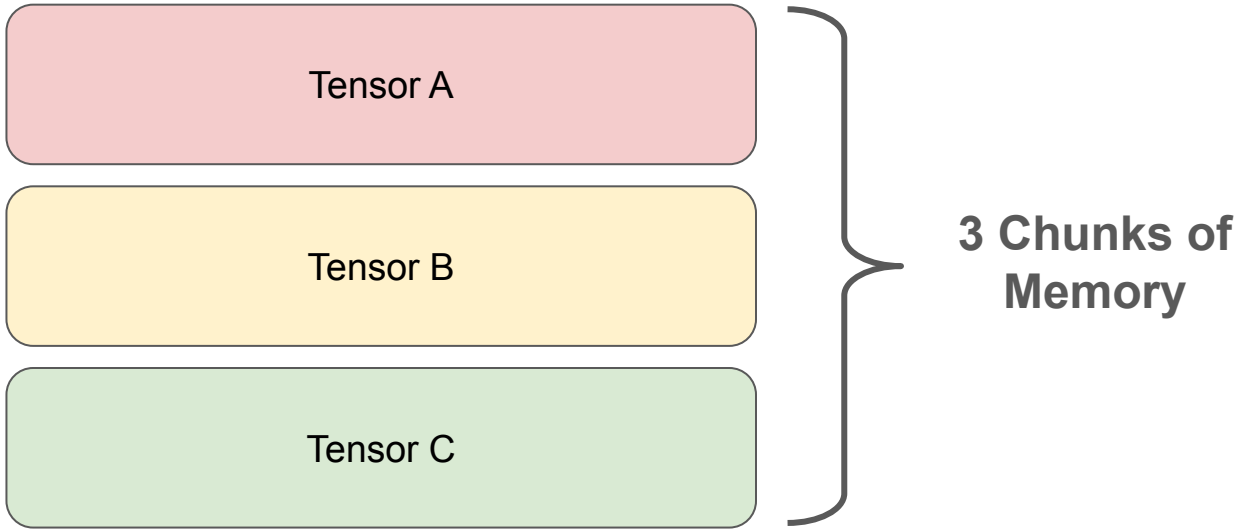

1. Block-Level Fusion

3. Kernel Memory Planning

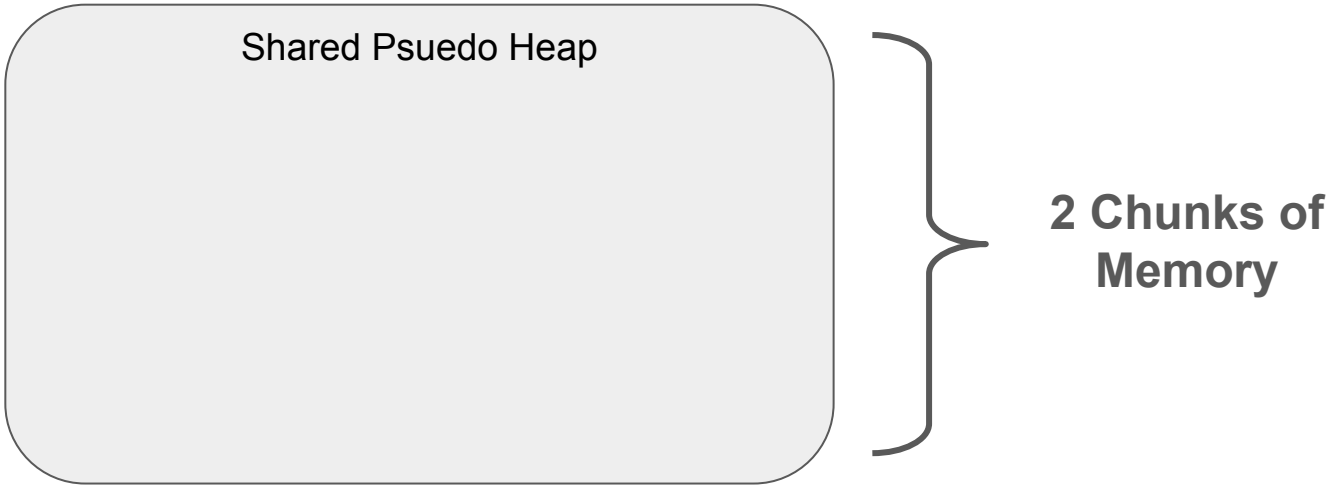
2. Tensor Layout Planning

4. Thread Operation Scheduling

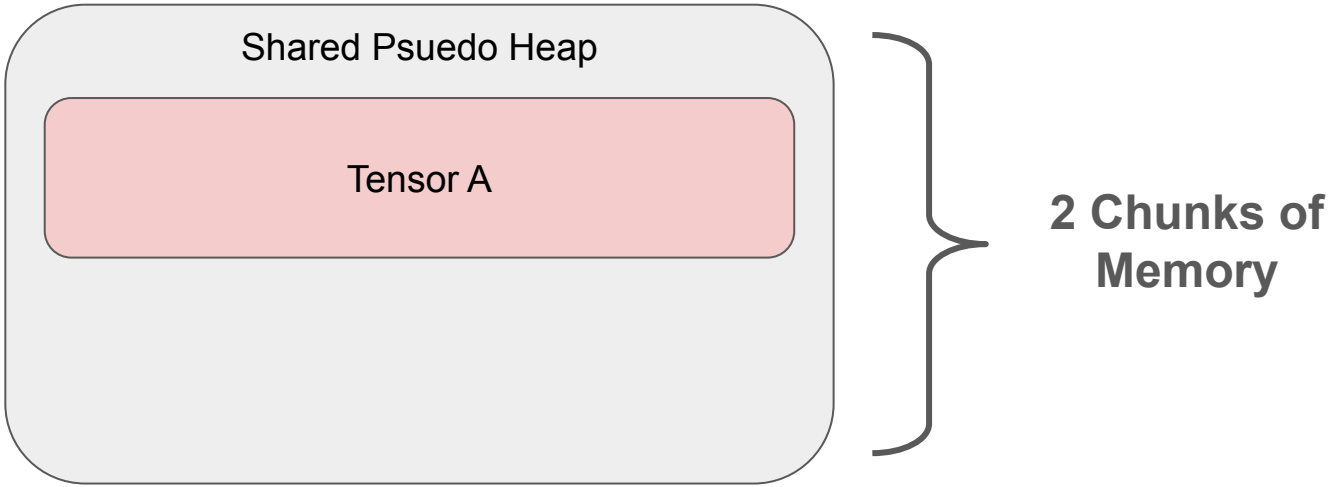
# $\mu$Graph Optimization: Memory Planning



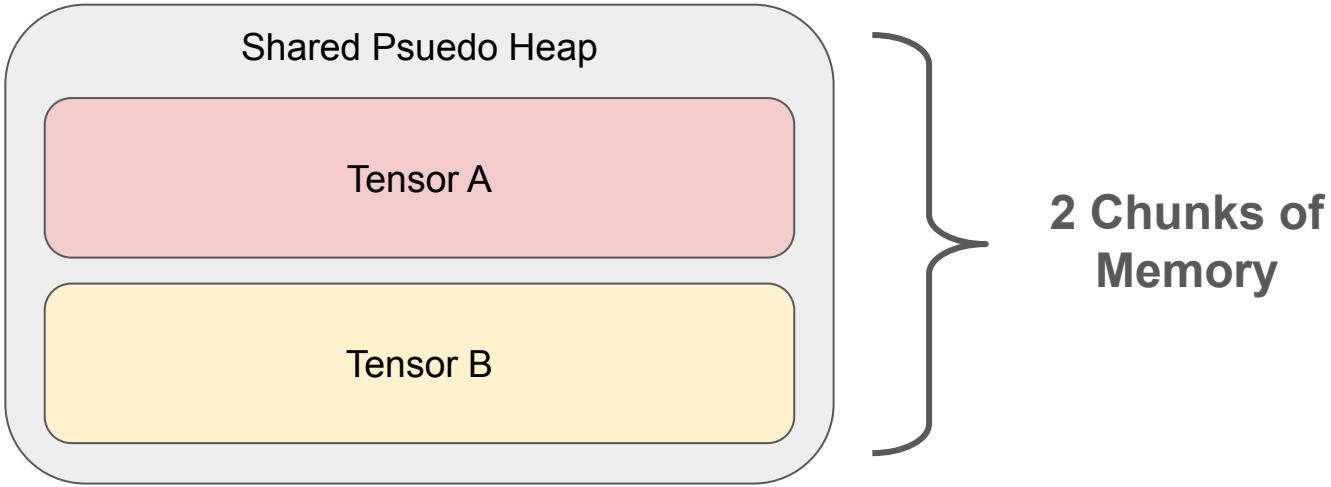
# *μ*Graph Optimization: Memory Planning



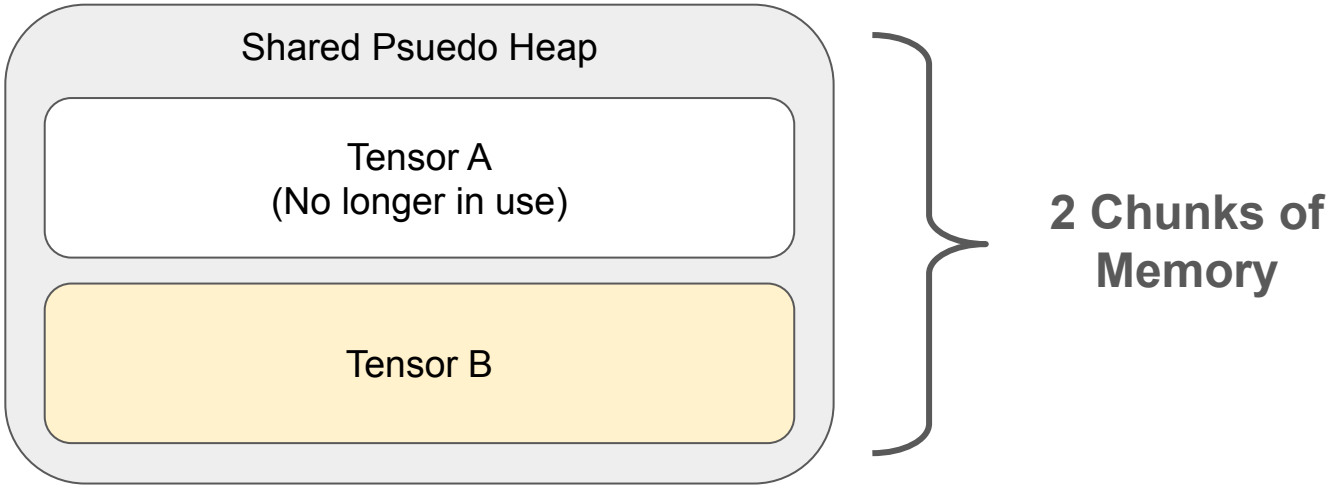
# $\mu$Graph Optimization: Memory Planning



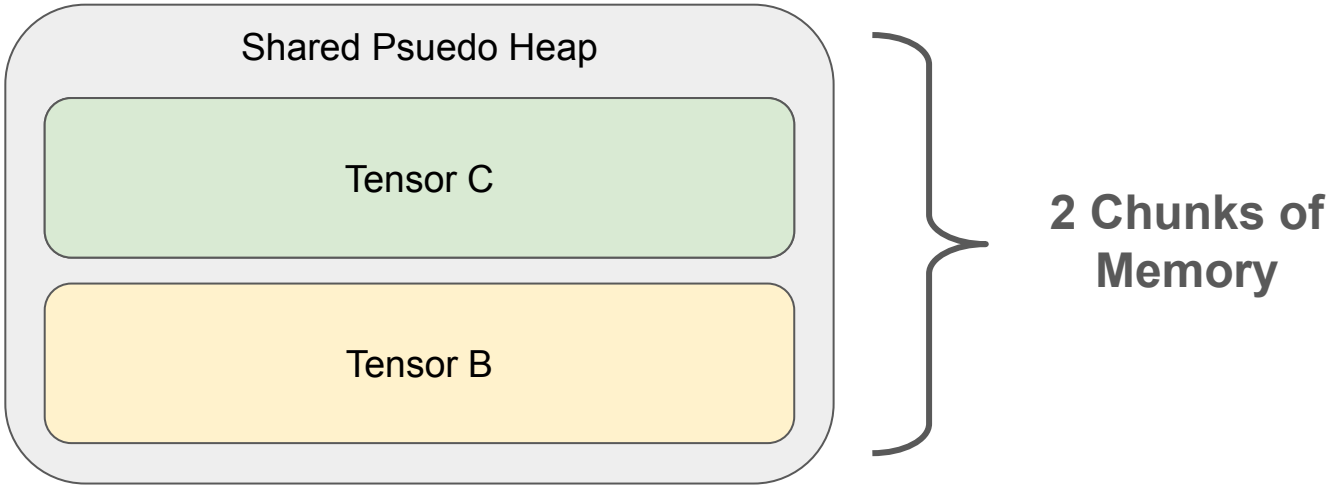
# $\mu$Graph Optimization: Memory Planning



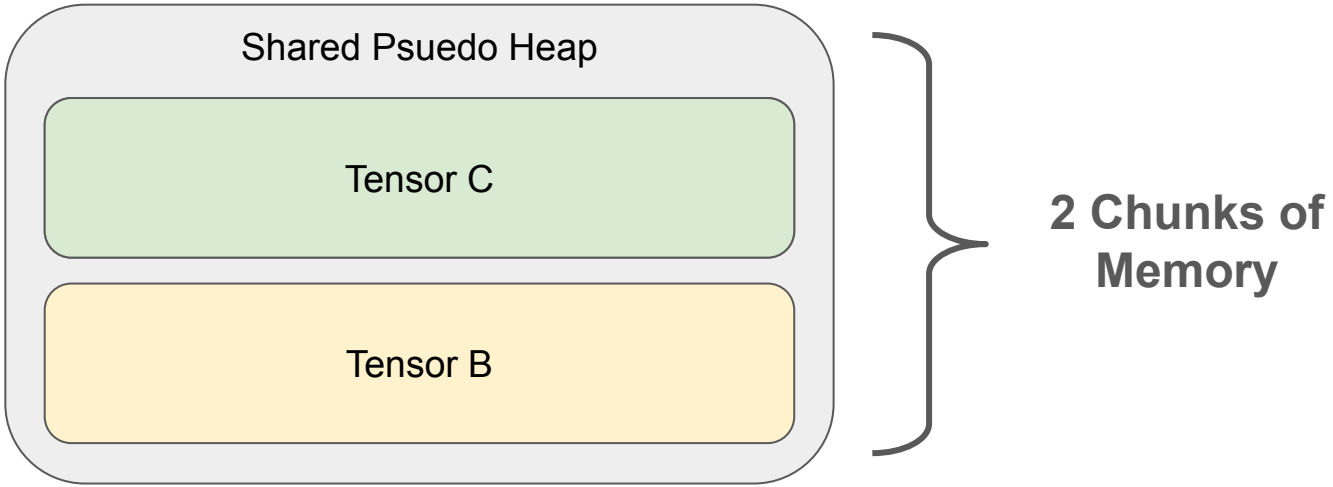
# $\mu$Graph Optimization: Memory Planning



# $\mu$Graph Optimization: Memory Planning

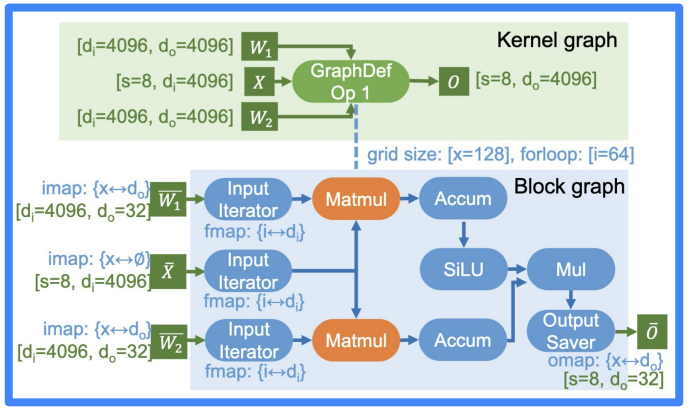


# $\mu$Graph Optimization: Memory Planning




**Save shared memory via pre-compiled “dynamic” allocation**

# μGraph Optimization: Overall Flow



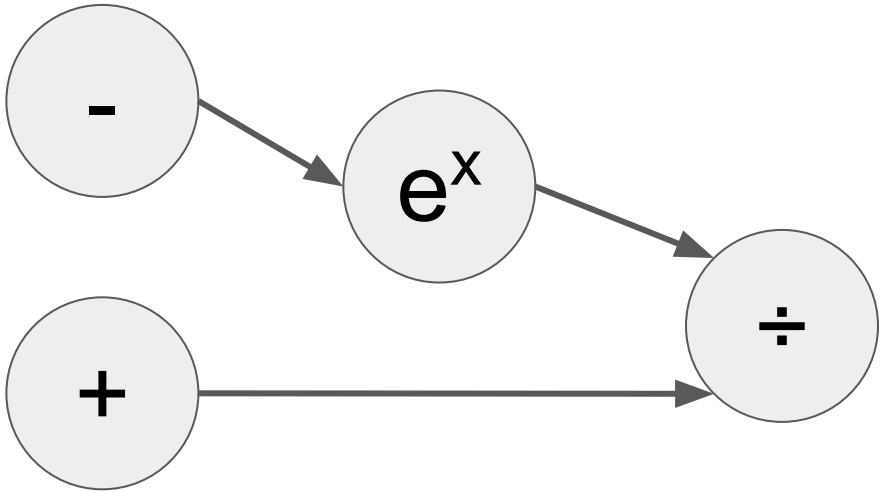

1. Block-Level Fusion

3. Kernel Memory Planning

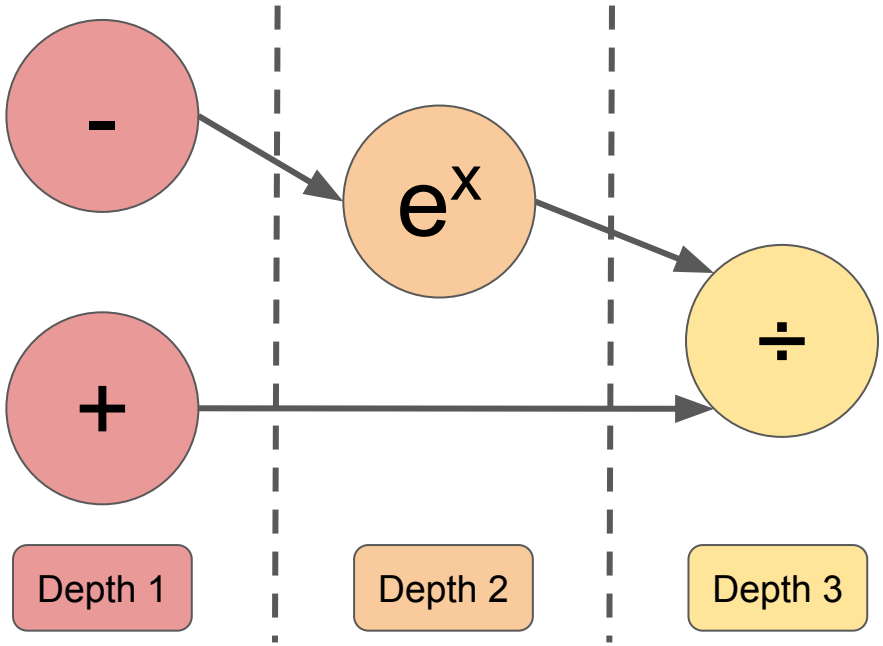
2. Tensor Layout Planning

4. Thread Operation Scheduling

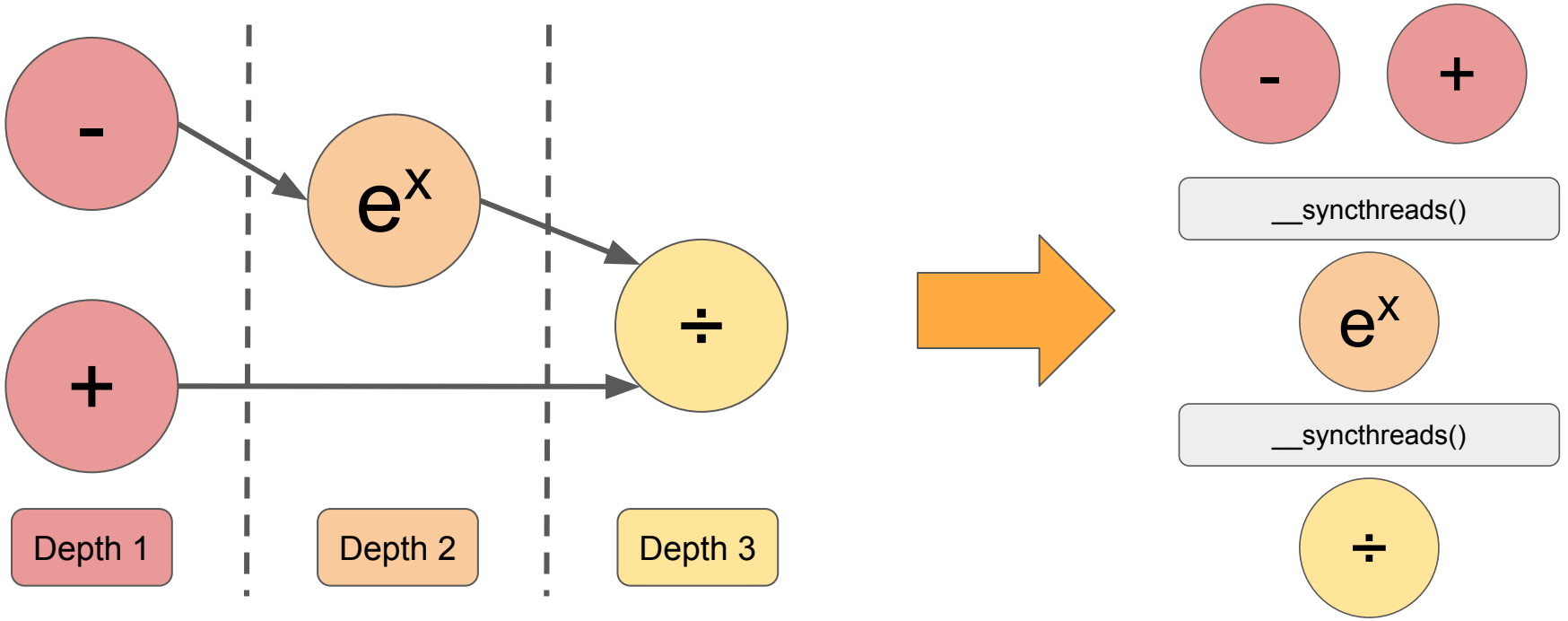
# $\mu$Graph Optimization: Thread Operator Scheduling



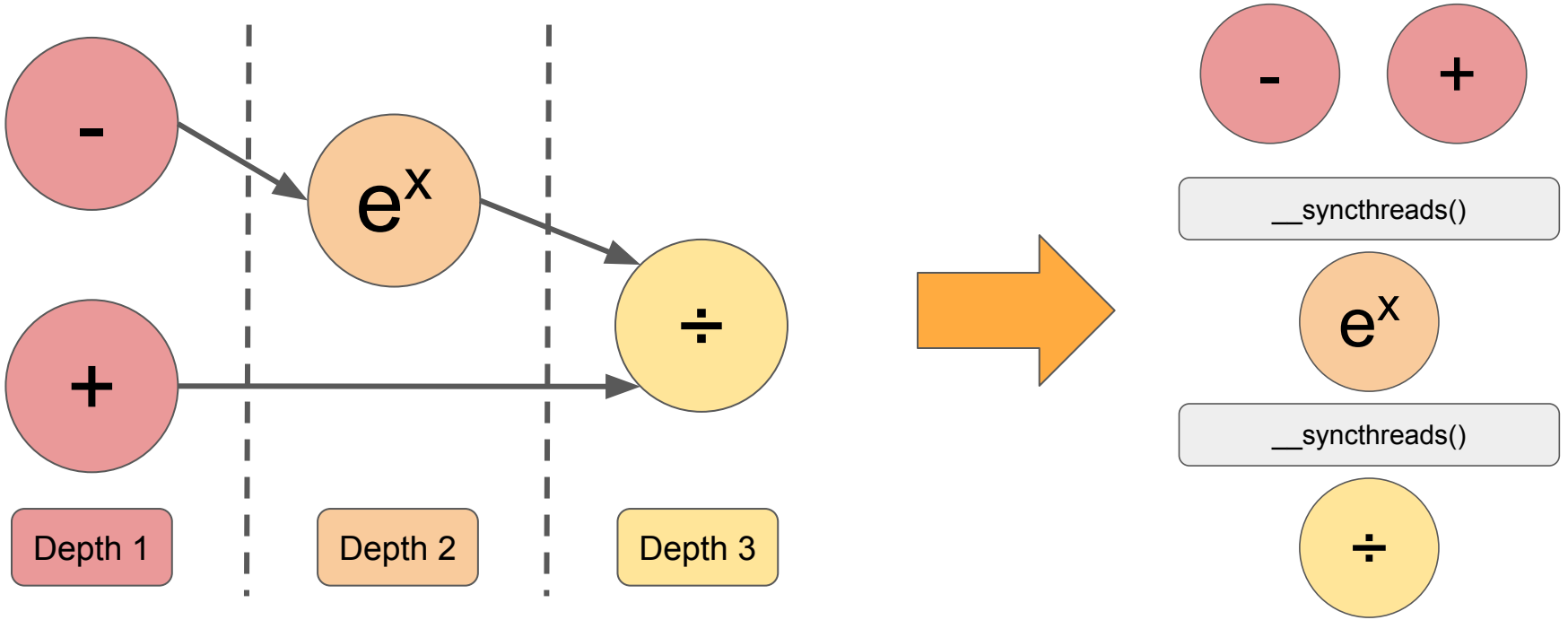
# $\mu$Graph Optimization: Thread Operator Scheduling



# μGraph Optimization: Thread Operator Scheduling



# μGraph Optimization: Thread Operator Scheduling




**Synchronize threads only when absolutely necessary**

# Results

# Does Mirage actually beat hand-tuned systems?

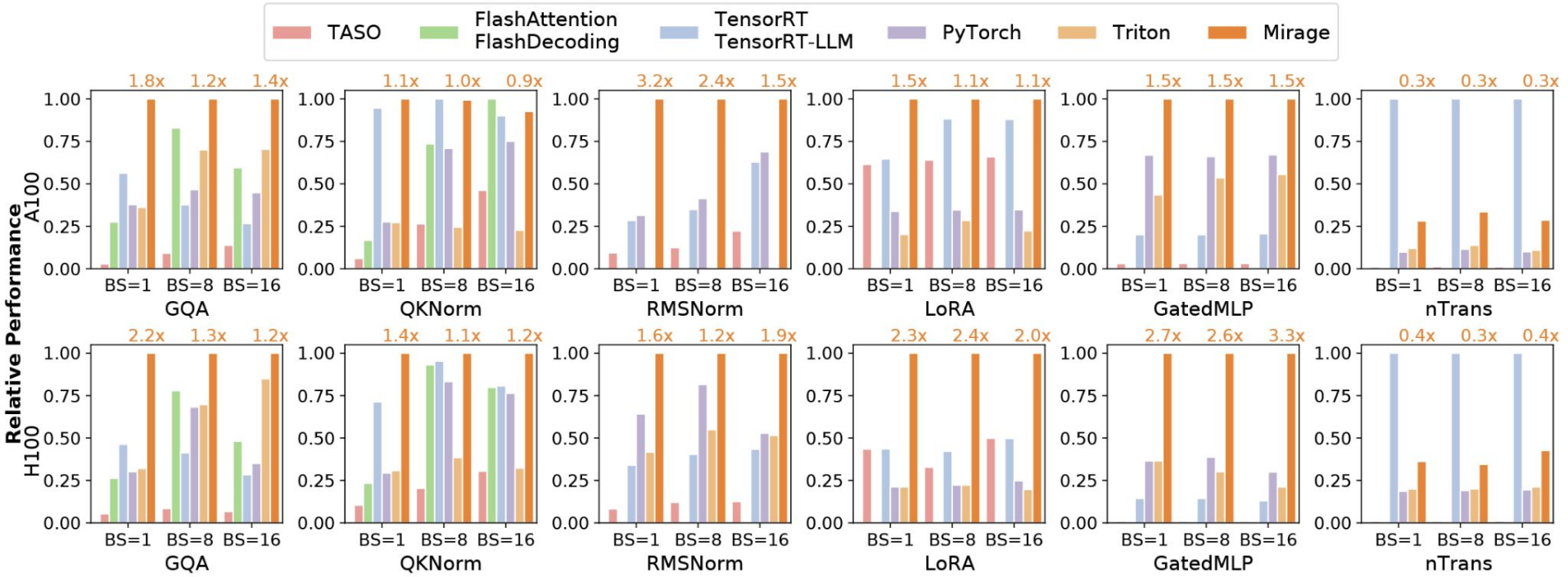
- **Baselines**
  - TASO/PET
  - PyTorch
  - TensorRT
  - TensorRT-LLM
  - Triton
  - FlashAttention / FlashDecoding
- **Metrics**
  - Runtime (avg/1000 runs)
- **Experimental Setup:**
  - NVIDIA A100/H100
  - Batch sizes = {1, 8, 16}

Table 4: DNN benchmarks used in our evaluation.

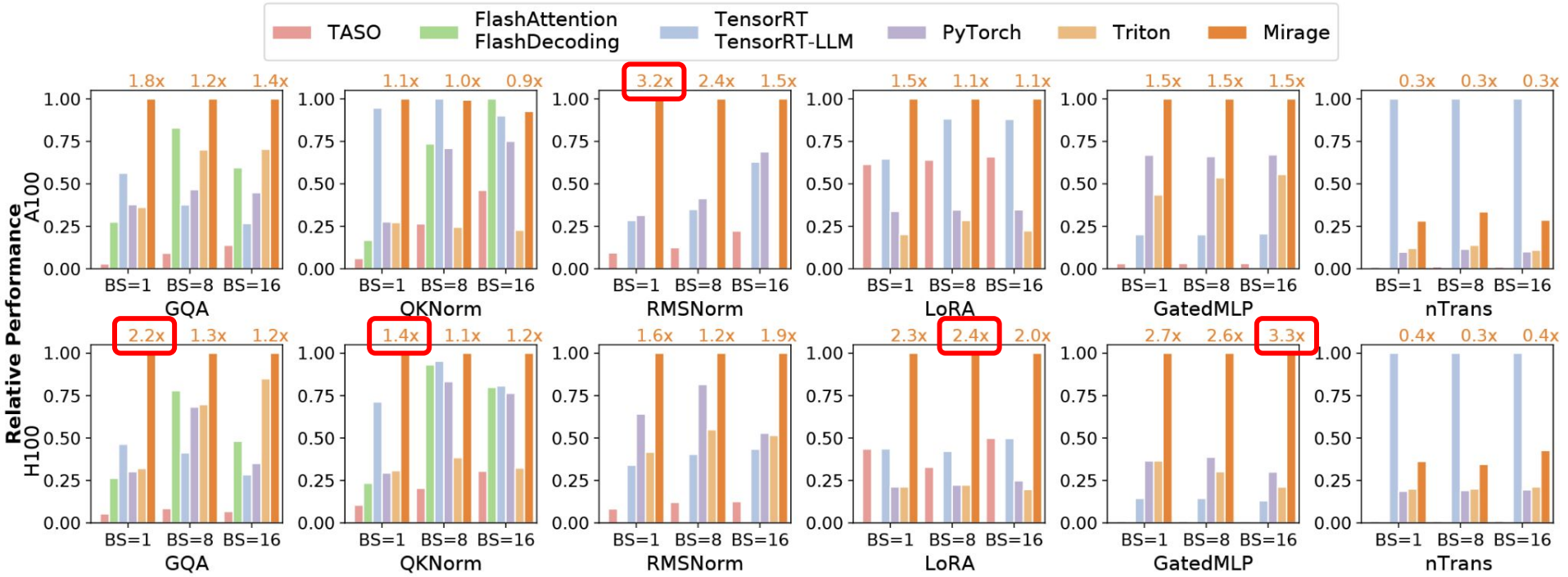
| Name | Description | Base Architecture |
|---|---|---|
| GQA | Group-query attention | LLaMA-3-70B [41] |
| QKNorm | QK normalization with attention | Chameleon-7B [40] |
| RMSNorm | RMS normalization with linear | LLaMA-2-7B [44] |
| LoRA | Low-rank adaptation | GPT-3-7B-LoRA [6] |
| GatedMLP | Gated multi-layer perceptron | Falcon-7B [10] |
| nTrans | Normalized Tarnsformer | nGPT-1B [28] |

***Mirage improves over the best prior system by up to 3.3×***

# Mirage usually wins, sometimes by a lot

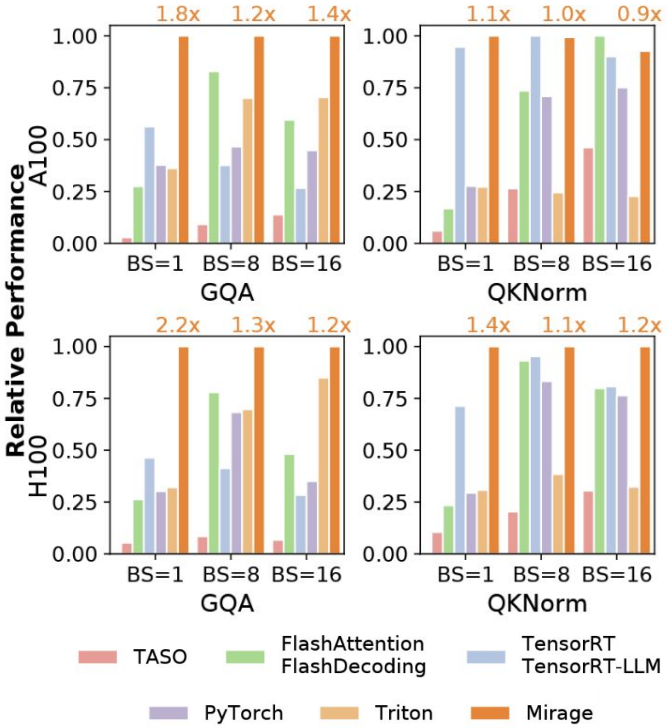


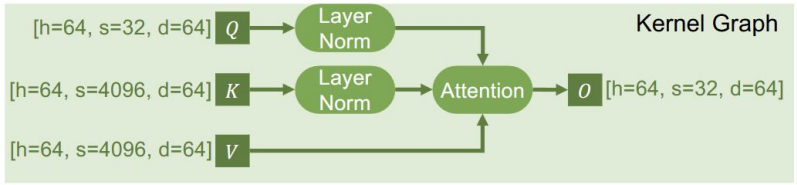
# Mirage usually wins, sometimes by a lot



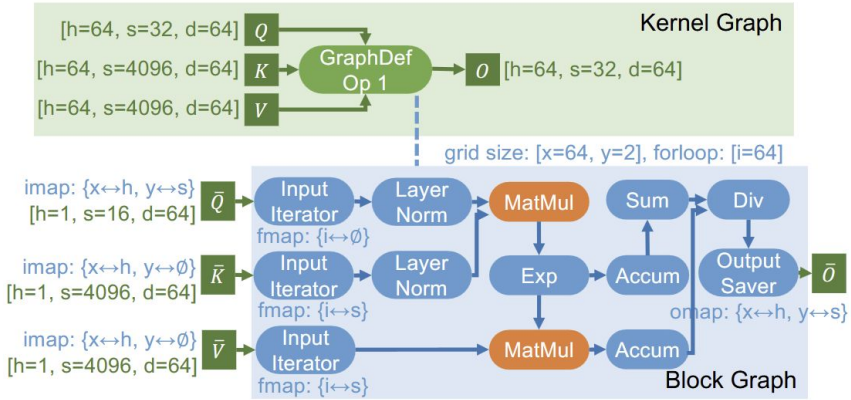

**Joint algebraic + scheduling + custom-kernel generation beats systems that optimize only on dimension**

# Case study 1: attention-family kernels



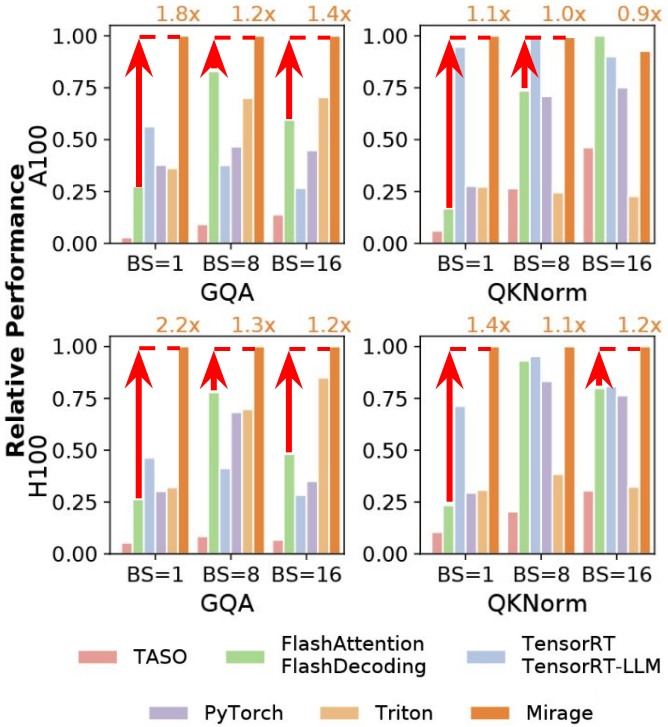



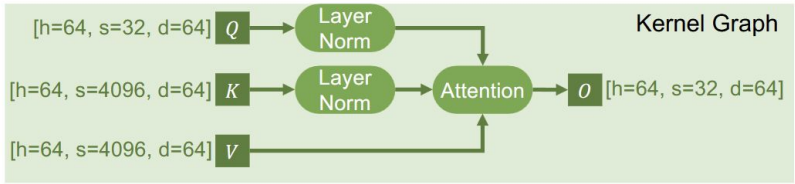

(a) The kernel graph for QKNorm and attention in existing systems.



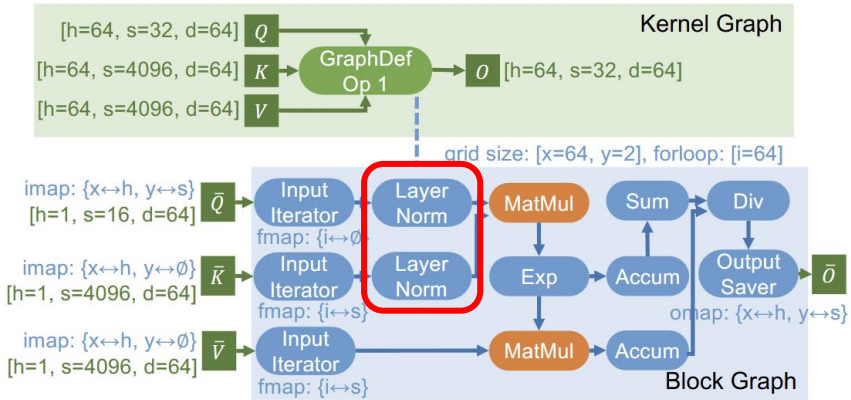

(b) The best $\mu$Graph discovered by Mirage for QKNorm and attention.

# Case study 1: attention-family kernels




(a) The kernel graph for QKNorm and attention in existing systems.



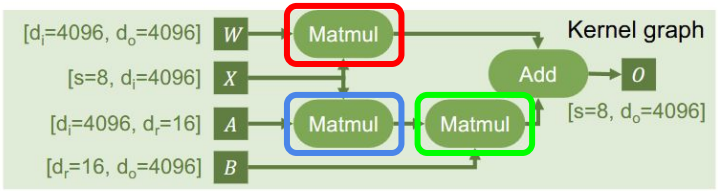

(b) The best $\mu$Graph discovered by Mirage for QKNorm and attention.

***Attention is the strongest proof point because Mirage is beating kernels people already spent years hand-optimizing.***

# Case study 2: Where LoRA gains come from

$$W \times X + B \times A \times X = (W \parallel B) \times (X \parallel (A \times X))$$

- Existing systems launch separate kernels for the linear op and the two low-rank LoRA ops

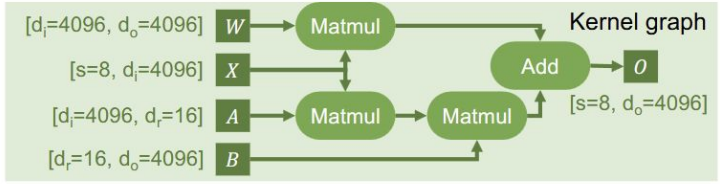



(a) The kernel graph for LoRA in existing systems.

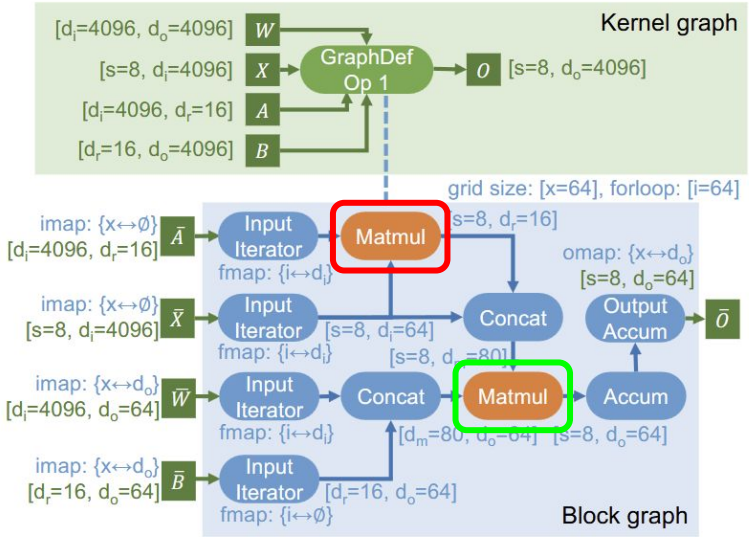
# Case study 2: Where LoRA gains come from

$$W \times X + B \times A \times X = (W \parallel B) \times (X \parallel (A \times X))$$

- Existing systems launch separate kernels for the linear op and the two low-rank LoRA ops
- Mirage fuses the three MatMuls and the Add
  - Algebraic rewrite first
  - Fused block-level implementation second




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

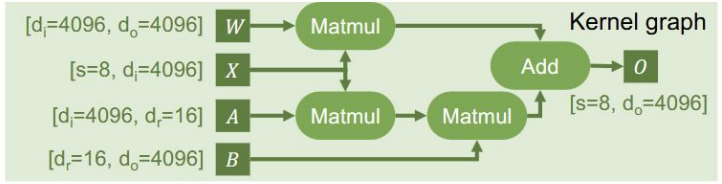



(b) The best $\mu$Graph discovered by Mirage for LoRA.

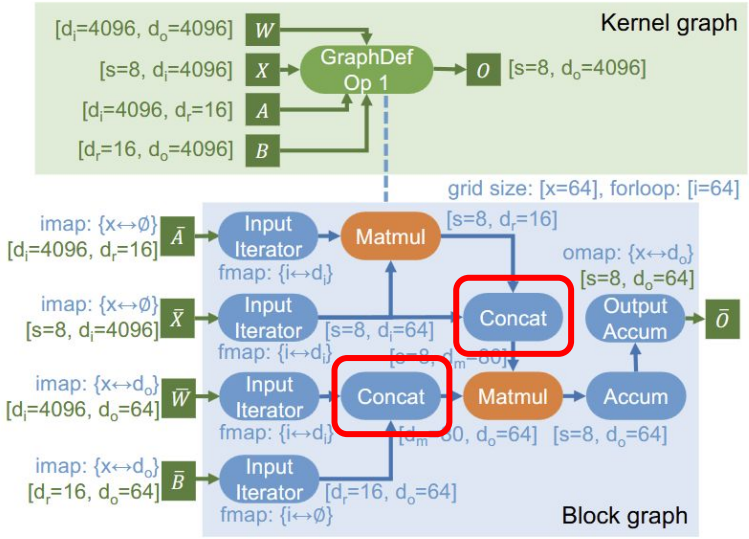
# Case study 2: Where LoRA gains come from

$$W \times X + B \times A \times X = (W \parallel B) \times (X \parallel (A \times X))$$

- Existing systems launch separate kernels for the linear op and the two low-rank LoRA ops
- Mirage fuses the three MatMuls and the Add
  - Algebraic rewrite first
  - Fused block-level implementation second
- **Concat Nodes**:
  - Do not do real computation
  - Tensor-offset changes in shared memory
  - Makes the rewrite especially attractive
- **Result**: 1.1× to 2.4× speedup



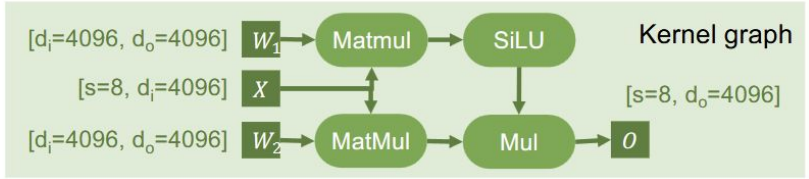

(a) The kernel graph for LoRA in existing systems.



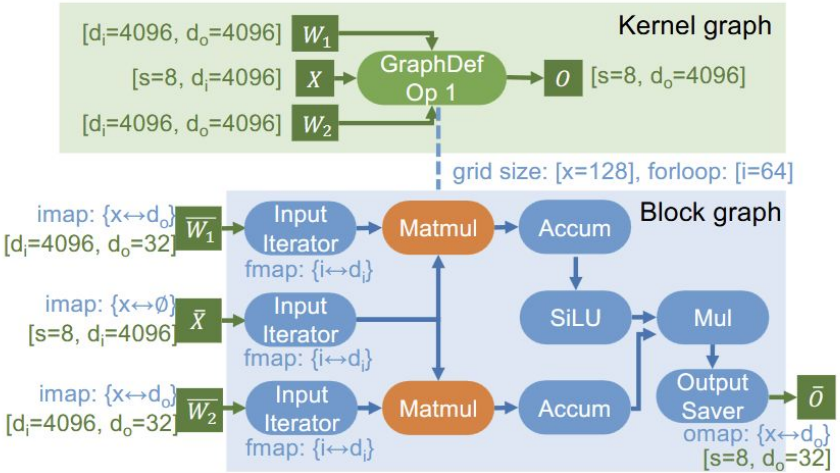

(b) The best $\mu$Graph discovered by Mirage for LoRA.

# Case study 3: GatedMLP success & nTrans failure

- **GatedMLP**: 3.3×



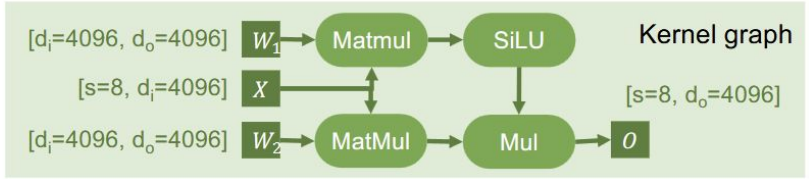

(a) The kernel graph for GatedMLP.



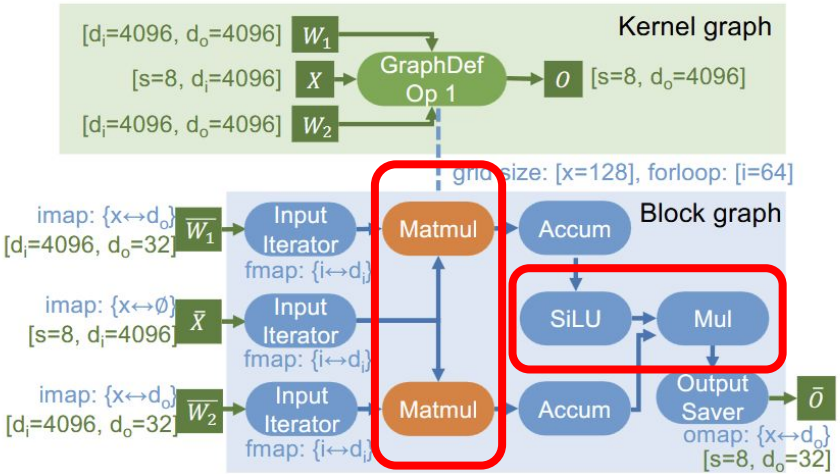

(b) The best $\mu$Graph discovered by Mirage for GatedMLP.

# Case study 3: GatedMLP success & nTrans failure

- **GatedMLP**: 3.3×
- 2 parallel Matmuls
- Fused SiLU + Mul



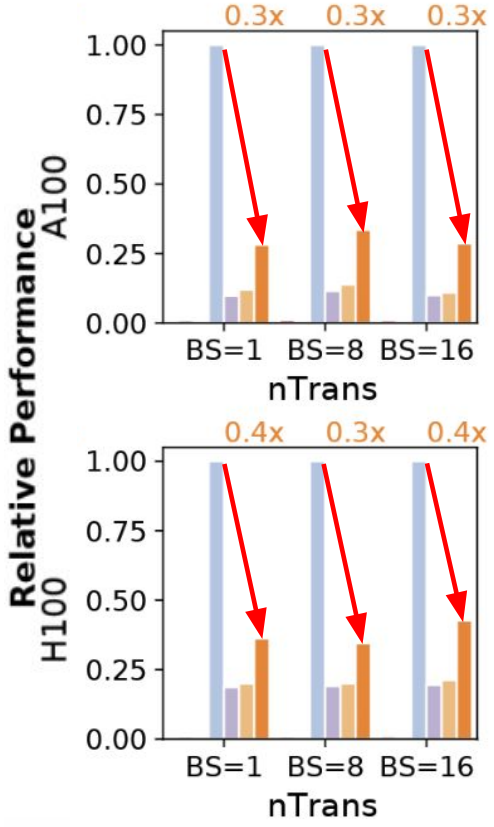

(a) The kernel graph for GatedMLP.

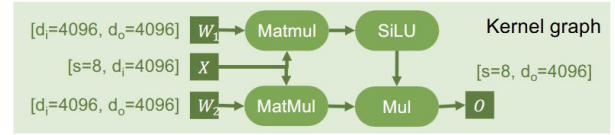



(b) The best $\mu$Graph discovered by Mirage for GatedMLP.

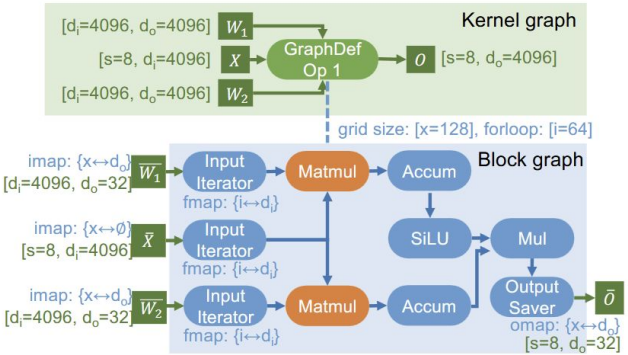
# Case study 3: GatedMLP success & nTrans failure

- **GatedMLP**: 3.3×
- 2 parallel Matmuls
- Fused SiLU + Mul
- **nTrans**: light-compute
- slower than TensorRT
  - SMEM overhead!



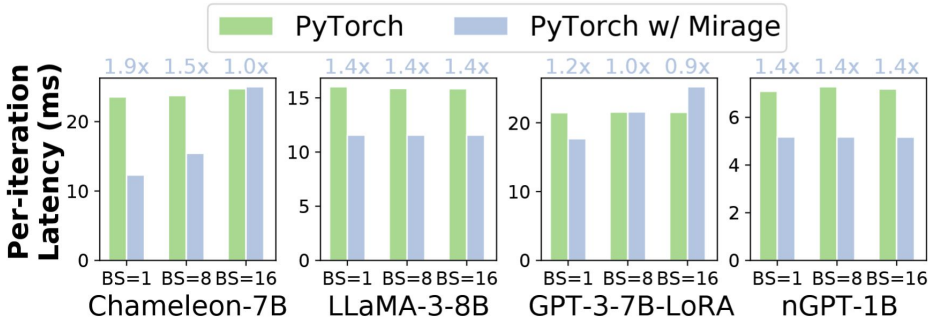

(a) The kernel graph for GatedMLP.




(b) The best $\mu$Graph discovered by Mirage for GatedMLP.

# End-to-end impact and practicality

- End-to-end latency
  - Chameleon-7B: **1.9×**
  - LLaMA-3-8B: **1.4×**
  - GPT-3-7B-LoRA: **1.2×**
  - nGPT-1B: **1.4×**



# End-to-end impact and practicality

- End-to-end latency
  - Chameleon-7B: **1.9×**
  - LLaMA-3-8B: **1.4×**
  - GPT-3-7B-LoRA: **1.2×**
  - nGPT-1B: **1.4×**
- Search is **slow**
  - 4 hours per LAX program
  - Some searches exceed 10 hours

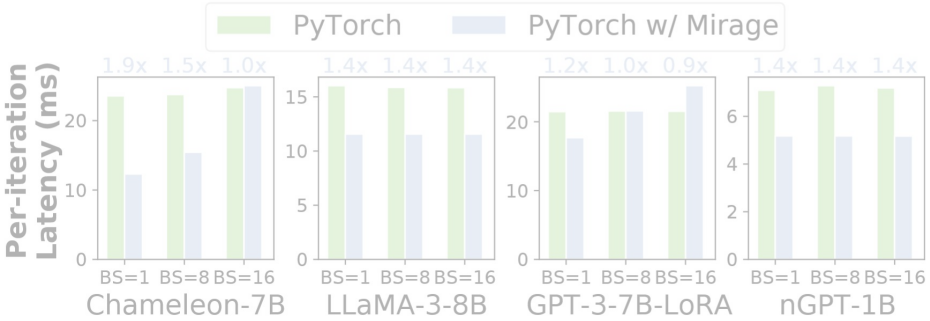



Table 5: Ablation study on Mirage's techniques to accelerate $\mu$Graph generation. We evaluate the impact of multi-threading and abstract expressions on search time for RMSNorm.

| Max # Ops in a Block Graph | Mirage | Mirage w/o Multithreading | Mirage w/o Abstract Expression |
|---|---|---|---|
| 5 | 11 sec | 58 sec | 768 sec |
| 6 | 16 sec | 93 sec | 19934 sec |
| 7 | 22 sec | 150 sec | $>$10 h |
| 8 | 24 sec | 152 sec | $>$10 h |
| 9 | 26 sec | 166 sec | $>$10 h |
| 10 | 26 sec | 166 sec | $>$10 h |
| 11 | 28 sec | 183 sec | $>$10 h |

# Mirage ablation and takeaway

- **What matters?**
  - Thread graph construction
  - Layout optimization
  - Operator scheduling
  - Memory planning

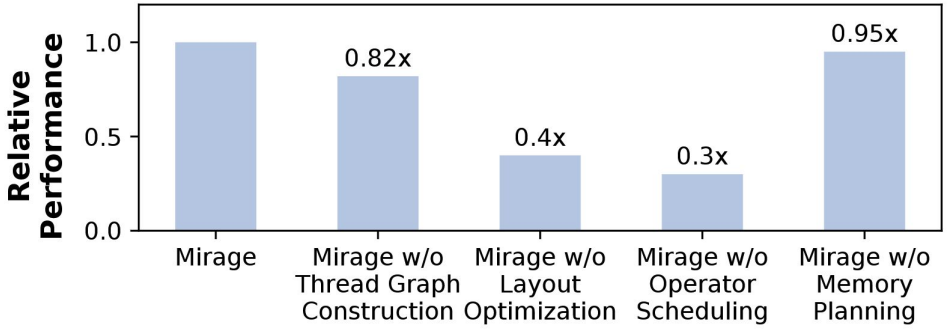



Figure 12: Ablation study on optimizations used in Mirage. We evaluate the performance degradation when disabling each optimization independently. The evaluation is performed on A100 for GQA with batch size 1.

# Mirage ablation and takeaway

- **What matters?**
  - Thread graph construction
  - Layout optimization
  - Operator scheduling
  - Memory planning
- Removing one hurts performance

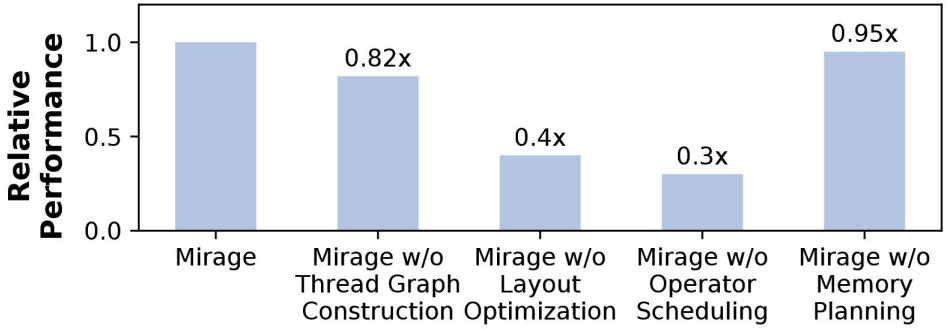



Figure 12: Ablation study on optimizations used in Mirage. We evaluate the performance degradation when disabling each optimization independently. The evaluation is performed on A100 for GQA with batch size 1.

***Multi-level joint optimization is why Mirage beats both graph-level and schedule-level prior work***

# Questions?