SOLAR: AI-Powered Speed-of-Light Performance Analysis

Qijing Huang, Sana Damani, Zhifan Ye, Athinagoras Skiadopoulos
Siva Kumar Sastry Hari, Jason Clemons, Sahil Modi, Jingquan Wang
Aditya Kane, Edward C Lin, Humphrey Shi, Christos Kozyrakis
NVIDIA

Abstract

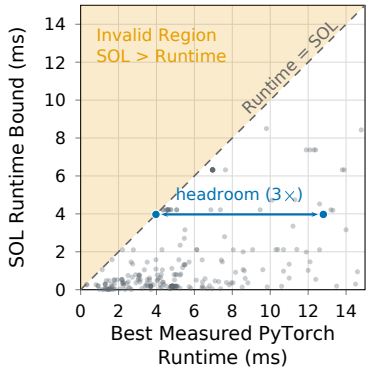
How fast *could* a deep-learning model run on target hardware, and how far is today's implementation from that limit? These questions are central to software, hardware, and algorithm optimizations. Speed-of-Light (SOL) analysis answers them by computing a workload's theoretical minimum execution time on a given architecture. Yet deriving SOL bounds remains manual, error-prone, and disconnected from rapid model development. To close this gap, we introduce SOLAR, a framework that automatically derives validated SOL bounds from PyTorch and JAX source code. SOLAR leverages both *generative* and *deterministic* components in its flow: an LLM frontend translates a source program into an executable Affine Loop IR, validated by output comparison; a deterministic flow lifts the IR into an einsum graph; and an analytical backend computes unfused, fused, and cache-aware SOL bounds. SOLAR provides comprehensive operator and language coverage, produces validated bounds with zero observed SOL violations, and offers multi-fidelity analysis that tightens bounds and surfaces optimization insights. We evaluate SOLAR across KernelBench, JAX/Flax models, and robotics workloads. These experiments demonstrate four use cases: headroom analysis at multiple fidelity levels, identifying optimization opportunities, cross-platform exploration, and inverse-roofline hardware provisioning. SOLAR is open source and available at `https://github.com/NVlabs/SOLAR`.

1 Introduction

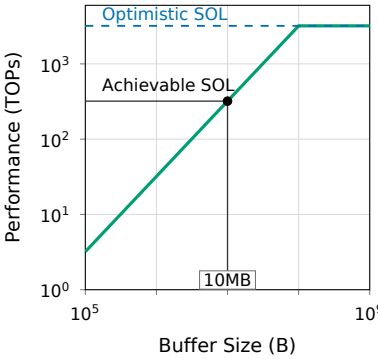
Modern deep-learning accelerators achieve peak throughputs measured in petaFLOPS, yet real workloads rarely approach these limits. How fast *could* a model or kernel run on target hardware, and how far is today's implementation from that limit? *Speed-of-Light* (SOL) analysis answers these questions by computing a workload's theoretical minimum execution time on a given architecture. These bounds are useful across the optimization stack: kernel and compiler engineers use them to identify bottlenecks and remaining headroom; algorithm designers use them to evaluate accuracy-performance-cost trade-offs; architects use them to provision compute and bandwidth for target workloads; and AI agents can use them to validate that generated optimizations remain physically attainable. Without analytical ceilings, optimization—and increasingly, agentic code generation—risks burning compute and tokens on work with little remaining headroom Hari et al. [2026].

Despite its importance, no existing tool derives SOL bounds automatically from source code. FLOP counters Meta AI [2023], Sovrasov [2023] suffer from narrow language and operator coverage (Figure 1c) and ignore I/O traffic; profilers NVIDIA [2023a] measure achieved rather than theoretical performance; and pure LLM estimation incurs errors. Meanwhile, the growing diversity of AI architectures, including Transformers Vaswani et al. [2017], MoE DeepSeek-AI [2024], SSMs Dao and Gu [2024], and hybrid models NVIDIA [2025], along with emerging domains such as robotics models, makes automated SOL analysis increasingly critical.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).



(a) SOL headroom on KernelBench.



(b) Naïve vs. cache-aware SOL.

Tool	Cov.	SOL
fvcore	75%	×
ptflops	84%	×
SOLAR	100%	✓

(c) Op coverage.

Figure 1: **What SOLAR provides.** SOLAR derives validated SOL bounds from source code, enabling three capabilities existing tools lack. (a) *SOL headroom analysis*: points below the diagonal reveal optimization opportunity; SOLAR exposes up to orders-of-magnitude headroom on KernelBench. (b) *Tighter bounds*: cache-aware analysis (OROJENESIS) accounts for on-chip buffer constraints, tightening SOL by up to $10\times$ over naïve roofline. (c) *Higher coverage*: FLOP counters Meta AI [2023], Sovrasov [2023] cover 75–84% of KernelBench and cannot derive SOL; SOLAR covers 100% with full SOL analysis.

To address this gap, we introduce SOLAR, **S**peed-of-Light **A**nalysis for **R**untime, a source-to-SOL framework with three goals: 1) broad operator and language coverage, 2) automated and validated SOL derivation, and 3) tighter bounds that surface optimization insights. SOLAR achieves these goals by separating *generative* translation with validation from *deterministic* analysis: an LLM translates source code into an executable Affine Loop IR, where correctness can be verified by numerical output comparison. A deterministic compiler then lifts the validated IR into an Einsum graph, and an analytical backend computes unfused, fused, and cache-aware SOL bounds.

This three-stage structure captures the best of both worlds: an agentic flow with output validation for easy translation from source languages to an Einsum representation, paired with deterministic lifting and performance analysis for reproducibility and auditable cost derivation. Figure 1a illustrates the result: each point plots a KernelBench workload's measured PyTorch runtime against SOLAR's fused SOL bound, with the gap to the diagonal representing optimization headroom—up to orders of magnitude on real workloads. Figure 1b further shows that cache-aware analysis Huang et al. [2024] in SOLAR tightens these bounds over naïve roofline by accounting for finite on-chip buffer capacity. This paper makes three contributions:

1. **A source-to-SOL framework.** SOLAR is the first tool to derive validated SOL bounds directly from PyTorch and JAX source code. It supports operator- and graph-level analysis with unfused, fused, and cache-aware bounds, achieving 100% analysis coverage on KernelBench.
2. **A three-stage pipeline separating generative and deterministic reasoning.** Rather than asking an LLM to estimate SOL directly, SOLAR confines the LLM to a verifiable task: translating source code into an executable Affine Loop IR (*Validated Agentic Frontend*). A deterministic compiler then lifts the IR into an Einsum graph (*Deterministic Einsum Conversion*), from which an analytical backend derives multi-fidelity SOL bounds.
3. **Evidence across kernel, algorithm, and architecture use cases.** Fusion analysis reveals up to $7.8\times$ additional headroom beyond per-operator analysis on KernelBench; cache-aware bounds tighten SOL by $2.06\times$ on L2. Einsum chain reordering identifies a $2.04\times$ FLOP reduction on DeepSeek MLA. Cross-platform projection spans four hardware targets without physical access, and inverse roofline reveals that 500 Hz robotics VLA targets require up to $19.7\times$ current bandwidth.

Table 1: **Performance analysis landscape.** SOLAR is the only approach that combines execution-validated translation, graph-level einsum extraction, multi-fidelity SOL bounds, cache-aware tightening, and language-agnostic input. "Graph" = analyzes operator graphs (not just individual kernels); "SOL" = derives theoretical bounds (not achieved performance); "Validated" = execution-validated translation; "Cache" = tighter SOL via buffer-size-aware tiling.

Category	Tool	Graph	SOL	Valid.	Cache	Input
FLOP counters	fvcore Meta AI [2023]	×	×	N/A	×	PyTorch
	ptflops Sovrasov [2023]	×	×	N/A	×	PyTorch
	torchinfo Yep [2020]	×	×	N/A	×	PyTorch
Profilers	NCU NVIDIA [2023a]	×	×	N/A	×	Any
	NSys NVIDIA [2023b]	✓	×	N/A	×	Any
Predictors	Paleo Qi et al. [2017]	✓	×	×	×	Caffe
	Habitat Yu et al. [2021]	✓	×	×	×	PyTorch
LLM-based	Pure LLM SOL	×	✓	×	×	Any
	ASAP Ding et al. [2025]	✓	×	×	×	JAX/XLA
	Opal Zaeed et al. [2025]	×	×	×	×	CUDA/HIP
Perf. modeling frameworks	Timeloop Parashar et al. [2019]	×	✓	✓	✓	Manual
	AccelForge Andrulis et al. [2026]	✓	✓	✓	✓	Manual
	SOLAR	✓	✓	✓	✓	**Any**

Table 2: **FLOP counter coverage on KernelBench.** fvcore and ptflops return 0 for element-wise ops and crash on L4. Neither reports I/O traffic or graph structure.

Level	N	fv OK	fv 0	fv fail	pt OK	pt 0	pt fail
L1	100	57	40	3	61	37	2
L2	100	100	0	0	100	0	0
L3	50	44	6	0	48	2	0
L4	20	1	0	19	19	0	1
All	**270**	**202**	**46**	**22**	**228**	**39**	**3**
		75% coverage			**84% coverage**		

Table 3: **Zero-shot LLM SOL accuracy on KernelBench.** Direct prompting achieves 83% overall but collapses on L3 (38%) due to spatial dimension tracking failures.

Level	N	Correct	Acc.	Top error mode
L1	100	96	96%	wrong formula
L2	100	100	100%	—
L3	50	19	38%	no spatial tracking
L4	20	9	45%	wrong formula
All	**270**	**224**	**83%**	

2 Related Work

No existing tool derives validated SOL bounds directly from source code. Table 1 positions SOLAR against the current landscape.

FLOP and parameter counters. Tools such as fvcore Meta AI [2023] and ptflops Sovrasov [2023] are limited to PyTorch inputs and cover only 75–84% of operators on KernelBench (Table 2). Because they count MACs and parameters but not total memory traffic, they cannot be used to derive precise and tight SOL bounds. Obtaining an operator graph through `torch.fx` or `torch.export` does not by itself fill this gap: SOL analysis still requires models of the operations' arithmetic costs and memory traffic. SOLAR derives these quantities from translated loop bodies.

Hardware profilers. Profilers such as NCU NVIDIA [2023a] and NSys NVIDIA [2023b] measure hardware utilization for a *specific implementation*, i.e., they report how close a kernel gets to the peak throughput of each hardware unit. This answers: *how well does this implementation use the hardware?* SOLAR answers a different question: *what is the minimum possible execution time for this algorithm on this hardware?* The distinction matters: not all algorithms can saturate all hardware units simultaneously. A memory-bound softmax will never reach 100% compute utilization; NCU would report headroom that is physically unachievable. SOLAR's bound is implementation-agnostic and tighter, considering both algorithmic properties and hardware constraints.

Direct LLM-based SOL estimation. Zero-shot LLM estimation (Claude Code with Opus 4.6) achieves 83% accuracy on KernelBench but collapses on composite workloads: accuracy falls to 38% on L3 subgraphs and 45% on L4 full models, with 20 of 46 errors exceeding $10\times$ (Table 3). Moreover, analyses that require systematic search, such as cache-aware tiling under buffer-capacity constraints Huang et al. [2024], are difficult to perform reliably through next-token prediction alone.

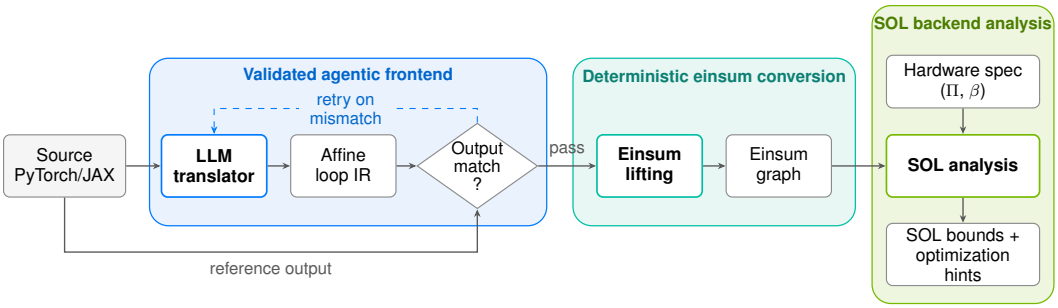


Figure 2: **SOLAR's three-stage pipeline.** (1) The validated agentic frontend translates source code into an executable Affine Loop IR, retrying on output mismatch against the source program. (2) Deterministic einsum conversion lifts the validated IR into an Einsum Graph representation. (3) SOL backend analysis combines this graph with target hardware specs to compute multi-fidelity roofline bounds and optimization hints.

Performance modeling frameworks. Timeloop Parashar et al. [2019] and AccelForge Andrulis et al. [2026] provide cache-aware performance modeling but require manual workload specification in their input formats; manual translation is error-prone and unvalidated. SOLAR bridges this gap by automatically translating source code into a validated einsum representation that these frameworks can consume directly.

LLM code translation. Frontier LLMs achieve 96% pass@1 on HumanEval Chen et al. [2021], 80% on SWE-bench Verified Jimenez et al. [2024], and 100% on KernelBench Ouyang et al. [2025] with agentic refinement, establishing strong source-to-source translation capabilities. SOLAR leverages this capability for verified code translation to enable automatic SOL derivation.

3 Methodology

SOLAR derives SOL bounds from source code through a three-stage pipeline (Figure 2). First, the **Validated Agentic Frontend** (Section 3.1) translates the source program into an executable *Affine Loop Intermediate Representation*. **Deterministic Einsum Conversion** (Section 3.2) then compiles the validated IR into an *Einsum Graph*, a DAG of extended einsums. Finally, the **SOL Analysis** (Section 3.3) stage computes multi-fidelity roofline bounds against target hardware specs.

3.1 Validated Agentic Frontend

This section describes the first stage of the SOLAR pipeline: a language-agnostic frontend that uses an LLM agent to translate tensor programs into an executable intermediate representation called the *Affine Loop IR*. The key idea is to treat the LLM as a programmable, retargetable parser whose output is validated by numerical comparison against the original program. LLM code generation cannot be trusted to produce correct output on the first attempt, so numerical validation closes this trust gap by providing a concrete pass/fail signal that gates downstream consumption.

3.1.1 Affine Loop IR

Using an LLM agent as a frontend for einsum generation places four requirements on the IR:

1. **Einsum extraction.** The representation must enable mechanical lifting to einsums by the backend.
2. **Expressiveness.** The IR must cover a wide range of DL workloads.
3. **LLM generation.** The IR must be easy for an LLM to produce correctly.
4. **Validation.** The LLM-generated IR must be executable and scalable for numerical validation.

To enable mechanical einsum extraction while retaining expressiveness, the Affine Loop IR (Figure 3b) splits a program into two parts: restricted affine loop kernels that represent individual einsum operators and an unrestricted composition layer that handles a wide range of DL workloads.

Each **Affine Loop Kernel** is restricted so that einsum equations can be derived mechanically from its structure. A `@kernel` function contains one or more perfectly nested loop nests over named-dimension tensors. Loop bounds are static tensor-shape constants, index expressions are affine in the loop variables, there is no data-dependent branching, and loop bodies contain only scalar arithmetic and a small set of fixed builtins.

Tensors carry the metadata the backend needs to derive byte counts. For example, `Tensor("B,M,K", dtype="fp16", role="weight", sparsity=0.5, fill=0)` declares an fp16 weight tensor with 50% sparsity and named dimensions `B`, `M`, and `K`. These named dimensions act as both loop dimensions and einsum subscripts, so the backend can derive an einsum equation from the loop nest.

To cover a wide range of DL workloads, an unrestricted top-level **Composition Layer** (`model()`) composes kernels, invokes library builtins, and expresses recurrence through control flow. Builtins handle patterns that do not fit the affine kernel model: data-movement operations (reshape, slice, stack, concat), data-dependent indexing (gather), and sparse operations (`apply_mask`, which propagates the tensor's `sparsity` field so that downstream byte and FLOP estimates reflect effective sparsity).

Finally, for ease of LLM generation and fast numerical validation, the IR is implemented as an **executable Python library** with Numba support. LLMs are already strong Python generators, so they can produce IR programs without fine-tuning, and every program can be run directly and compared against the original without any additional compilation. For scalable validation, a loop whose iterations carry no data dependence may be annotated `Dim("B", parallel=True)` by the LLM to enable parallel execution during validation.

3.1.2 Agentic IR Translation and Validation

In a traditional compiler, supporting a new source language means building a dedicated parser, type checker, and lowering pass. Instead, SOLAR uses an LLM agent that translates any source program into the shared Affine Loop IR through a generate-then-verify loop: the agent proposes a translation, the pipeline validates it against the original program's outputs, and on failure provides diagnostic feedback that guides the agent toward a corrected version (Li et al. [2022], Shinn et al. [2023]).

A two-part system prompt separates target-IR knowledge from source-language conventions for easy portability to new languages. The fixed *target-IR specification* defines the Affine Loop IR grammar and provides few-shot examples. The *source-language adapter* supplies language-specific conventions. The pipeline first captures a golden output by executing the original program, invokes the translation agent, executes the generated Numba-accelerated IR, compares the outputs, and provides feedback to the agent in a loop. Retries are bounded by a configurable limit; translations that never pass validation are reported as failures.

One limitation of our approach is that it does not provide formal guarantees; incorrect translations can pass if sampled inputs do not expose an output mismatch. Output-equivalent translations can also inflate compute and memory costs. We prompt the agent to minimize arithmetic operations and memory accesses, but this heuristic neither guarantees minimal cost nor replaces independent cost checks. Analysis is input-conditioned. For data-dependent control flow, SOL bounds cover only paths exercised by the supplied inputs; other paths require additional representative inputs.

Finally, support for a new source language requires only two additions: (1) a model-capture module and (2) a source-language adapter for the prompt. The LLM thus becomes a *verifiable language-agnostic parser*: it reads diverse source languages, but the downstream pipeline only ever receives translations that numerically match the original program's outputs.

3.2 Deterministic Einsum Conversion

The deterministic backend lifts the validated Affine Loop IR into an *Einsum Graph* representation from which all performance quantities are derived mechanically.

3.2.1 Einsum Graph

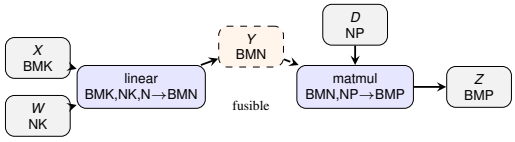
SOLAR lifts to einsums rather than analyzing the Affine Loop IR directly because the einsum subscript structure provides the minimal canonical form from which all roofline quantities can be derived.

```python
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Linear(256, 512, bias=True)
        self.D = nn.Parameter(torch.randn(512, 64))

    def forward(self, x):
        y = self.fc(x)
        return torch.matmul(y, self.D)

def get_inputs():
    """Return [batch=2, seq=128, hidden=256]."""
    torch.manual_seed(0)
    return [torch.randn(2, 128, 256)]
```

(a) Source Code.



Rank Sizes: B=2, M=128, K=256, N=512, P=64

Node	Einsum	MACs	Memory
`linear#0`	BMK,NK,N→BMN	BMKN	BMK+NK+N+BMN
`matmul#0`	BMN,NP→BMP	BMNP	BMN+NP+BMP

(c) Einsum Graph

```python
@kernel
def linear(X: Tensor("B,M,K"), W: Tensor("N,K"),
           Bias: Tensor("N")) -> Tensor("B,M,N"):
    Y = Tensor(B=X.shape[0], M=X.shape[1], N=W.shape[0])
    for b in Dim("B", parallel=True):
      for m in Dim("M"):
       for n in Dim("N"): Y[b, m, n] = Bias[n]
    for b in Dim("B", parallel=True):
      for m in Dim("M"):
       for n in Dim("N"):
        for k in Dim("K"): Y[b, m, n] += X[b, m, k] * W[n, k
            ]
    return Y

@kernel
def matmul(Y: Tensor("B,M,N"), D: Tensor("N,P")) -> Tensor(
      "B,M,P"):
    Z = Tensor(B=Y.shape[0], M=Y.shape[1], P=D.shape[1])
    for b in Dim("B", parallel=True):
      for m in Dim("M"):
       for p in Dim("P"):
        for n in Dim("N"): Z[b, m, p] += Y[b, m, n] * D[n, p
            ]
    return Z
```

(b) Affine Loop IR (LLM-translated).

Mode	Compute	Memory	SOL	Bottleneck
Unfused	83.9 MFLOP	2.03 MB	1.00 μs	Memory
Fused	83.9 MFLOP	0.99 MB	0.82 μs	Compute

(d) SOL Analysis (H100 PCIe)

Figure 3: **End-to-end SOLAR example** (LinearBiasMatmul). (a) PyTorch source. (b) Agent-translated Affine Loop IR with named-dimension tensors and affine loops. (c) Einsum Graph (fusible edge dashed) with extracted einsum equations. (d) SOL analysis: fusion eliminates the intermediate, shifting the bottleneck from memory to compute.

The Einsum Graph is a DAG whose nodes are *extended einsums* Odemuyiwa et al. [2024], Kjolstad et al. [2017] and whose edges are tensors that encode data dependencies between operations (Figure 3c). An einsum specifies a tensor contraction via subscript strings that name the dimensions of each operand and the output (e.g. `BMK,KN→BMN` for matrix multiplication, where `K` is summed over) Einstein [1922]. Extended einsums generalize this to support reduction operators beyond summation, e.g. max-reduction for softmax. The einsum representation makes compute cost and memory traffic derivable in closed form from the subscript structure alone. Given an einsum with output dimensions $\mathcal{D}_O$, reduction dimensions $\mathcal{D}_R$, and input tensors $\{T_i\}$:

- *MACs* $= \prod_{d \in \mathcal{D}_O \cup \mathcal{D}_R} |d|$: one multiply-accumulate per point in the full iteration space.
- *Memory traffic* $= \sum_i \texttt{sizeof}(T_i) \cdot \prod_{d \in \mathcal{D}_i} |d| \; + \; \texttt{sizeof}(T_{\text{out}}) \cdot \prod_{d \in \mathcal{D}_O} |d|$: each tensor is read (or written) once at its full size.

These formulas are exact for a single unfused einsum and require only the dimension sizes and dtypes. Fusion analysis extends this: when two adjacent einsums share an intermediate tensor, its traffic is eliminated from the fused cost, and the combined iteration space determines the new MAC count.

3.2.2 Affine Loop IR to Einsum Translation

The backend first extracts a dataflow graph from the verified IR by running the `model()` function under a lightweight tracer that intercepts every kernel and builtin call, records its inputs and outputs, and constructs the graph without executing any loop bodies. Nodes in the resulting graph represent kernel and builtin invocations, and edges are the tensors that flow between them, each carrying its associated metadata. This graph encodes the full dependency structure of the program and is the basis for all subsequent analysis. Next, from each kernel node, the backend reads named dimensions, identifies reduction dimensions (absent from the output), and emits einsum strings (Figure 3c).

Definition 1 (Einsum extraction)**.** *Given a `@kernel` with output dimensions $\mathcal{D}_O$ and loop indices $\mathcal{D}$, the reduction dimensions are $\mathcal{D}_R = \mathcal{D} \setminus \mathcal{D}_O$. Each input tensor T_i accessing dimensions $\mathcal{D}_i$ yields an einsum operand $\mathcal{D}_i \to \mathcal{D}_O$ with implicit summation over $\mathcal{D}_R$.*

The Einsum Graph annotates each node with derived quantities (MACs, memory traffic, arithmetic intensity) and fusibility. Edges encode data dependencies via intermediate tensors, and the graph structure determines which operators can share on-chip buffers and eliminate DRAM round-trips when fused (Figure 3c). Each translated program additionally emits a YAML sidecar that records the graph in language-independent form, enabling downstream tools such as Timeloop Parashar et al. [2019] and AccelForge Andrulis et al. [2026] to consume the workload. The backend detects non-einsum nodes (builtins such as `gather` and `select`, and `reshape` kernels) and classifies them as data-movement-only. The SOL analysis assigns these nodes zero MACs while still deriving their memory traffic from the tensor dimensions.

3.3 SOL Analysis

Given an Einsum Graph and architecture specs (peak dense throughput Π in FLOP/s and DRAM bandwidth β in B/s), SOLAR computes the roofline lower bound Williams et al. [2009] (Figure 3d):

$$T_{\text{SOL}} = \max\left(\frac{\text{FLOPs}}{\Pi}, \frac{\text{Traffic}}{\beta}\right) \tag{1}$$

A kernel must both execute all FLOPs and transfer all data; assuming perfect overlap, the slower term dominates. The *arithmetic intensity* $\mathcal{A} = \text{FLOPs}/\text{Traffic}$ determines the bottleneck regime: compute-bound when $\mathcal{A} > \Pi/\beta$, memory-bound otherwise. The boundary Π/β is the *ridge point*.

Graph-level analysis. Unlike per-kernel roofline, SOLAR operates on the full Einsum Graph. Fusing adjacent operators eliminates intermediate DRAM round-trips, reducing Traffic and tightening T_{SOL}. SOLAR infers fusibility from data dependencies in the graph. On KernelBench L3, fused-graph SOL reveals $7.8\times$ additional headroom beyond per-operator analysis.

Multi-fidelity modes. SOLAR provides three SOL fidelity levels, each refining the traffic estimate in Equation (1). All three share the same FLOPs; they differ only in how DRAM traffic is counted.

- *Unfused*: each operator is analyzed independently; all intermediate tensors are read from and written to DRAM between operators. Within each operator, **unlimited on-chip cache** is assumed—every element is transferred exactly once. $T_{\text{SOL}}^{\text{unfused}} = \sum_v T_{\text{SOL}}(v)$.
- *Fused*: adjacent fusible operators share on-chip buffers, eliminating intermediate DRAM round-trips. The same **unlimited cache** assumption applies: each remaining input/output element is transferred exactly once.
- *Cache-aware fused (*OROJENESIS*)*: models **finite on-chip buffer capacity** with fusion. When the working set of a (fused) operator exceeds the hardware cache, tiling forces elements to be re-read from DRAM. The OROJENESIS/AccelForge backend searches fusion-chain lengths, tilings, and chaining factors, selecting minimum-traffic configurations within its search space subject to the modeled L2 capacity Huang et al. [2024], Andrulis et al. [2026].

Inverse roofline. Given a target latency T_{target}, SOLAR derives minimum hardware specs: $\Pi_{\text{min}} = \text{FLOPs}/T_{\text{target}}$ and $\beta_{\text{min}} = \text{Traffic}/T_{\text{target}}$. This enables hardware architects to evaluate whether a proposed platform meets deployment constraints before silicon exists.

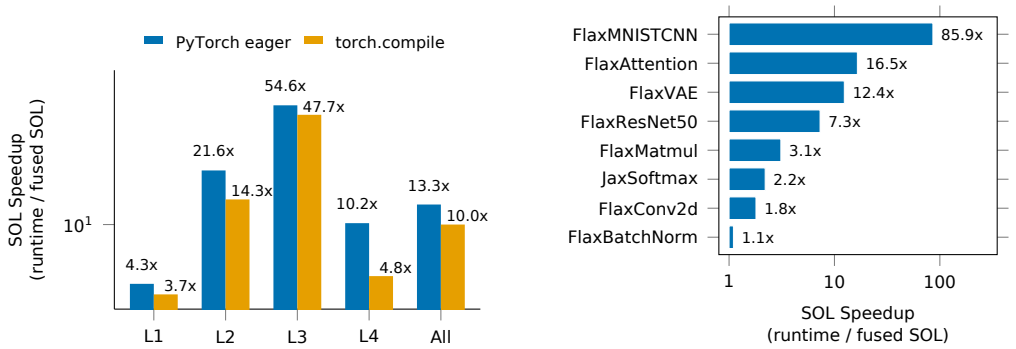
Limitation. A current limitation of SOLAR is that its analysis is based solely on tensor shapes rather than values. Consequently, it cannot capture value-dependent optimizations such as compression or constant propagation, and may overlook performance gains from structured or repeated data that enable more efficient memory access or algebraic simplifications. Additionally, the SOL bound may not be tight in practice due to hardware variability, such as power capping or thermal throttling.

4 Evaluation

We evaluate SOLAR on KernelBench Ouyang et al. [2025] (270 problems, L1–L4; PyTorch, H100 PCIe), JAX/Flax workloads (8 programs), and robotics models (3 configurations on Jetson Thor), organized around three practitioner questions (Sections 4.1 to 4.3).

4.1 How Much Headroom Exists and How Do I Close It?

Quantifying the gap. SOLAR achieves 100% validated analysis coverage on KernelBench. This is eventual coverage of all 270 programs: 72% validate on the first attempt and 96% within three; the



(a) KernelBench headroom (L1–L4).

(b) JAX/Flax headroom.

Figure 4: **SOL headroom analysis.** (a) KernelBench: measured runtime over fused SOL across L1–L4 levels. (b) JAX/Flax: headroom (baseline / fused SOL) spans from $1.1\times$ (BatchNorm) to $85.9\times$ (MNISTCNN), demonstrating language-agnostic frontend coverage.

remainder require additional automatic retries with capped validation workloads and search-assisted source lookup. Figure 4a plots the geomean SOL speedup (runtime / fused SOL) for PyTorch eager and `torch.compile` across all four levels. SOL headroom grows with graph complexity: L1 shows $4.3\times$ (eager) and $3.7\times$ (`compile`), indicating that the compiler captures most of the single-operator gap. L2 widens to $21.6\times$ vs. $14.3\times$—intermediate tensor traffic creates substantial headroom that `compile` only partially closes. L3 shows the largest gap ($54.6\times$ eager, $47.7\times$ `compile`), confirming that graph-level fusion is essential for composite workloads. On L4, `compile` closes most of the gap ($4.8\times$ vs. $10.2\times$ eager) because it already fuses aggressively on full models. The gap between eager and `compile` bars quantifies how much headroom the compiler has already captured; the remaining `compile` bar height is the headroom that requires kernel-level or algorithmic optimization. Fused vs. unfused SOL analysis (Figure 6a) further decomposes this headroom into fusion and tiling components. Per-problem breakdowns are in Appendix A; SOL-ExecBench Lin et al. [2026] (3,957 workloads) results are in Appendix B.

Does it work beyond PyTorch? SOLAR also handles JAX/Flax (Figure 4b). We evaluated 8 programs spanning single operators to full ResNet-50: headroom ranges from $1.1\times$ (BatchNorm, correctly identifying no fusion benefit) to $85.9\times$ (FlaxMNISTCNN), with FlaxAttention at $16.5\times$. Additional validation covers JAXBench Tschand et al. [2026] programs, CuTe/CUTLASS programs, and Triton programs, including attention, normalization, and MLP blocks. Some Triton implementations are from Liger Kernel LinkedIn [2024].

Headroom against expert kernels. Table 4 compares SOL with FlashInfer Ye et al. [2025] library CUDA and CuTe/CUTLASS implementations evaluated on 356 FlashInferBench workloads across 17 kernels. Figure 5 shows FlashAttention-4 (FA4) headroom for two BF16 configurations at batch size one over sequence lengths 1K–128K: MHA with $(d_{qk}, d_v, h_q, h_{kv}) = (192, 128, 128, 128)$ and GQA with $(128, 128, 64, 8)$. At 128K, headroom is $1.25\times$ for MHA and $1.31\times$ for GQA. Detailed CuTe/CUTLASS and Triton headroom results on H100 PCIe are in Appendix D.

Where should I focus optimization effort? Different SOL fidelities surface different bottlenecks (Figure 6a). On L1, cache-aware SOL ($1.9\times$) is tighter than unfused/fused ($3.7\times$/$4.3\times$) because finite cache forces intra-operator tiling re-reads. On L2, unfused ($5.1\times$) is tighter than cache-aware ($10.5\times$) because intermediate tensor traffic between isolated kernels dominates. OROJENESIS cache-aware bounds tighten SOL by up to $2.25\times$ overall, identifying cases where tiling strategy—not just fusion—is the bottleneck. The Einsum graph also enables contraction order enumeration: Figure 6b shows that optimal ordering of DeepSeek MLA's 4-tensor chain achieves a $2.04\times$ FLOP reduction over naïve left-to-right ordering (a $22\times$ range across all 5 orders).

4.2 How Do I Design Efficient Algorithms for Target Hardware?

SOLAR enables design-space exploration *without hardware access* by sweeping architectural parameters through the analytical pipeline.

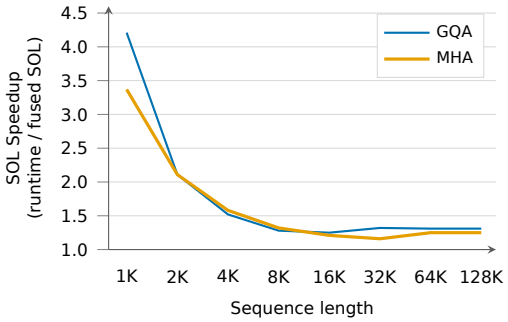
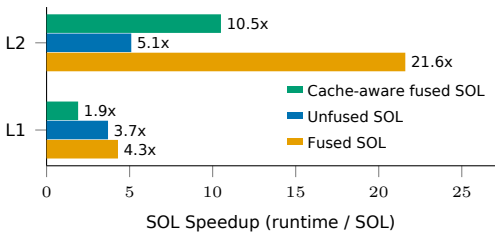


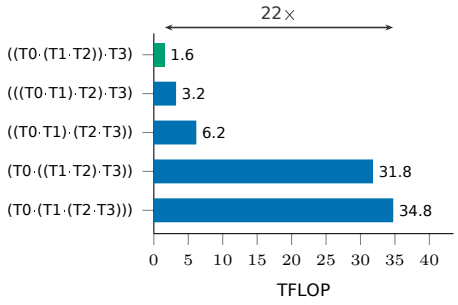
Figure 5: **FA4 headroom on B200.** Runtime / fused SOL for MHA and GQA (batch size one, BF16).

Table 4: **FlashInfer runtime / SOL on B200.** CUDA and CuTe/CUTLASS implementations evaluated on FlashInferBench workloads; $1\times$ is an analytical floor, not a guaranteed attainable runtime.

Kernel	GPU	Shapes	Ratio range
Norm	B200	88	1.17–3.15×
GEMM	B200	268	1.10–7.02×



(a) Multi-fidelity SOL speedup for L1/L2. Cache-aware is tightest on L1; unfused is tightest on L2.



(b) Einsum chain reordering for DeepSeek MLA.

Figure 6: **Optimization hints.** (a) Multi-fidelity SOL on L1/L2: on L1, cache-aware is tightest (intra-operator tiling costs dominate); on L2, unfused is tightest (intermediate tensor traffic dominates). (b) All 5 association orders of a 4-tensor MLA chain; a $22\times$ TFLOP range, with the optimal order achieving a $2.04\times$ reduction over naïve left-to-right ordering.

Parameter sensitivity. We sweep four axes of a Qwen3-4B block on Jetson Thor (Figure 7): batch size (1–64), sequence length (512–8192), hidden dimension, and intermediate dimension. Batch size drives linear scaling (2.1 ms to 136.4 ms, $64\times$), with the block remaining compute-bound throughout. Sequence length triggers a memory-to-compute regime shift: at 512 tokens, the block is memory-bound, but at 4K+ tokens quadratic attention traffic makes it compute-bound. Hidden and intermediate dimensions show sublinear scaling ($\sim 2\times$) because the MLP dominates and attention traffic is largely dimension-invariant. Unfused SOL is up to $20\times$ higher than fused (10–$15\times$ typical), confirming fusion is critical across operating points.

Cross-platform projection. Figure 8 projects a Qwen3-4B block onto four platforms: fused SOL spans $5.8\times$ from B200 (0.61 ms) to A6000 (3.56 ms). Fusion benefit ranges from $1.8\times$ on B200 (8 TB/s) to $14.5\times$ on Jetson Thor (273 GB/s), quantifying the bandwidth-fusion interaction. As a separate measured check, GR00T N1.6 takes 36 ms against 13.6 ms SOL on H100, and 92 ms against 20 ms SOL on Jetson Thor. Both remain above SOL; these observations do not establish tightness for every projected platform.

4.3 What Hardware Do I Need for Real-Time Deployment?

SOLAR's inverse roofline derives minimum Π and β from latency constraints, answering: *what hardware must I provision to meet a deployment target?* We analyze three robotics models (pi0 (3B), GR00T N1.6 System 1, DreamZero WAM (14B)) on Jetson Thor (1,035 TFLOP/s FP4, 273 GB/s) targeting 500 Hz servo control loops (Figure 9).

All three are memory-bound: pi0 requires $\beta_{\text{min}} = 1{,}750$ GB/s ($6.4\times$ current) for 500 Hz; GR00T N1.6 needs only $0.3\times$ current compute, confirming it is entirely bandwidth-limited. DreamZero WAM requires $19.7\times$ bandwidth *and* $8.3\times$ compute—no single hardware upgrade suffices, pointing

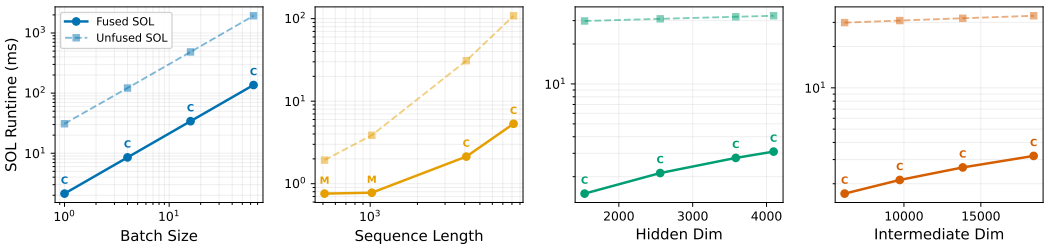


Figure 7: **Qwen3-4B block parameter sensitivity on Jetson Thor.** Fused and unfused SOL across four architectural axes. C/M = compute/memory-bound. Batch size drives linear scaling; sequence length triggers a memory→compute regime shift at ∼4K tokens.

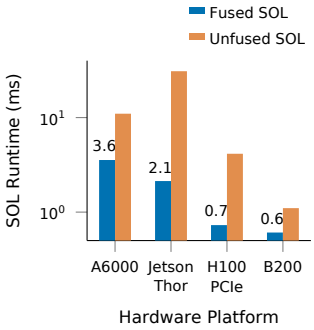
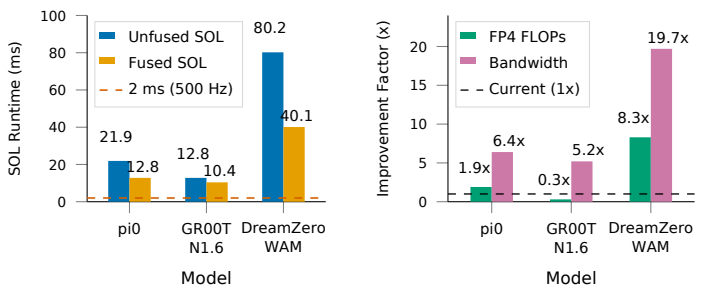


Figure 8: **Cross-hardware Qwen3-4B block SOL.** SOLAR enables designers to compare deployment targets and identify a suitable platform.



(a) SOL runtime (unfused vs. fused). (b) HW improvement required.

Figure 9: **Robotics model analysis on Jetson Thor.** (a) Fused SOL is significantly lower than unfused for all models. (b) Bandwidth improvement dominates compute improvement for all models, confirming memory-bound behavior. DreamZero WAM requires 19.7× current bandwidth for 500 Hz.

to model compression and algorithmic changes as necessary complements to silicon. Full analysis is in Appendix C.

5 Conclusion

We presented SOLAR, the first framework to derive validated Speed-of-Light bounds directly from PyTorch and JAX source code. By separating generative translation from deterministic analysis, SOLAR's source-to-SOL flow makes SOL derivation accessible without manual modeling or profiling hardware. Across KernelBench, JAX/Flax models, and robotics workloads, SOLAR quantifies optimization headroom, identifies fusion and chain-reordering opportunities, enables cross-platform exploration in the absence of hardware access, and derives hardware provisioning targets.

Acknowledgments and Disclosure of Funding

We thank Simon Guo, Vignesh Balaji, Angshuman Parashar, Po-An Tsai, and Joel Emer for helpful discussions and feedback on SOLAR. We also thank Nestor Qin, Fengzhe Zhou, Yuan Zhang, Dheeraj Peri, Ouye Xie, Michael Behar, Rishabh Mehta, Vartika Singh, Vikram Sharma Mailthody, Terry Chen, Zihao Ye, Hanfeng Chen, Tianqi Chen, Vinod Grover, Wei Chen, Wei Liu, Eric Chung, Luis Ceze, Roger Bringmann, Michael Lightstone, and members of the NVIDIA ATLAS team for their contributions to SOL-EXECBENCH. We thank Tanner Andrulis and Michael Gilbert for their help with AccelForge.

This work was supported by NVIDIA Corporation, where all authors are employed. The authors declare no additional competing interests.

References

Tanner Andrulis, Michael Gilbert, Vivienne Sze, and Joel S. Emer. AccelForge: Comprehensive modeling and co-design framework for AI accelerators, 2026. URL https://arxiv.org/abs/2609.11906.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Tri Dao and Albert Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.

DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Yuran Ding, Xinwei Chen, Xiaofan Zhang, and Zongwei Zhou. ASAP: An agentic solution to auto-optimize performance of large-scale LLM training. *arXiv preprint arXiv:2511.03844*, 2025.

Albert Einstein. The general theory of relativity. In *The Meaning of Relativity*, pages 54–75. Princeton University Press, 1922.

Siva Kumar Sastry Hari, Vignesh Balaji, Sana Damani, Qijing Huang, and Christos Kozyrakis. Improving efficiency of GPU kernel optimization agents using a domain-specific language and speed-of-light guidance. *arXiv preprint arXiv:2603.29010v1*, 2026.

Qijing Huang, Po-An Tsai, Joel S. Emer, and Angshuman Parashar. Mind the gap: Attainable data movement and operational intensity bounds for tensor algorithms. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024. doi: 10.1109/ISCA59077.2024.00021.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

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

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158.

Edward Lin, Sahil Modi, Siva Kumar Sastry Hari, Qijing Huang, Zhifan Ye, Nestor Qin, Fengzhe Zhou, Yuan Zhang, Jingquan Wang, Sana Damani, et al. SOL-ExecBench: Speed-of-light benchmarking for real-world GPU kernels against hardware limits. *arXiv preprint arXiv:2603.19173*, 2026. URL https://arxiv.org/abs/2603.19173.

LinkedIn. Liger Kernel: Efficient Triton kernels for LLM training, 2024. URL https://github.com/linkedin/Liger-Kernel.

Meta AI. fvcore: FLOP counter for PyTorch models. https://github.com/facebookresearch/fvcore, 2023.

NVIDIA. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute, 2023a.

NVIDIA. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems, 2023b.

NVIDIA. Nemotron-H: A family of hybrid SSM-transformer models. Technical report, NVIDIA, 2025.

Toluwanimi O. Odemuyiwa, Michael Pellauer, and Joel S. Emer. The EDGE language: Extended general einsums for graph algorithms. *arXiv preprint arXiv:2404.11591*, 2024.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. Timeloop: A systematic approach to DNN accelerator evaluation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019.

Hang Qi, Evan R. Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

Vladislav Sovrasov. ptflops: FLOPs counter for PyTorch models. `https://github.com/sovrasov/flops-counter.pytorch`, 2023.

Arya Tschand, Charles Hong, Julian Walker, Nina Cai, Shangkun Wang, Suvinay Subramanian, Sundar Dev, Vijay Janapa Reddi, Amir Yazdanbakhsh, and Sethu Sankaran. JAXBench: Benchmarking autonomous TPU kernel optimization. *arXiv preprint arXiv:2607.20466*, 2026. URL `https://arxiv.org/abs/2607.20466`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and customizable attention engine for LLM inference serving. In *Proceedings of Machine Learning and Systems*, volume 7, 2025. URL `https://proceedings.mlsys.org/paper_files/paper/2025/hash/dbf02b21d77409a2db30e56866a8ab3a-Abstract-Conference.html`.

Tyler Yep. torchinfo: View model summaries in PyTorch, 2020. URL `https://github.com/TylerYep/torchinfo`.

Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. Habitat: A runtime-based computational performance predictor for deep neural network training. In *USENIX ATC*, 2021.

Mohammad Zaeed, Tanzima Z. Islam, and Vladimir Inđić. Opal: A modular framework for optimizing performance using analytics and LLMs. *arXiv preprint arXiv:2510.00932*, 2025.

A KernelBench Details

KernelBench Ouyang et al. [2025] contains 300 problems organized into four levels of increasing complexity (L1: single operators, L2: operator sequences, L3: model subgraphs, L4: full models). Our evaluation uses a 270-problem subset (100 L1, 100 L2, 50 L3, 20 L4) for which both SOL and PyTorch baseline runtimes are available. Geomean SOL speedup (PyTorch eager / fused SOL) ranges from $4.3\times$ on L1 to $54.6\times$ on L3, with L4 at $10.2\times$.

Figures 10 to 13 show per-problem SOL speedup (PyTorch eager runtime / fused SOL) for each level, sorted in descending order.

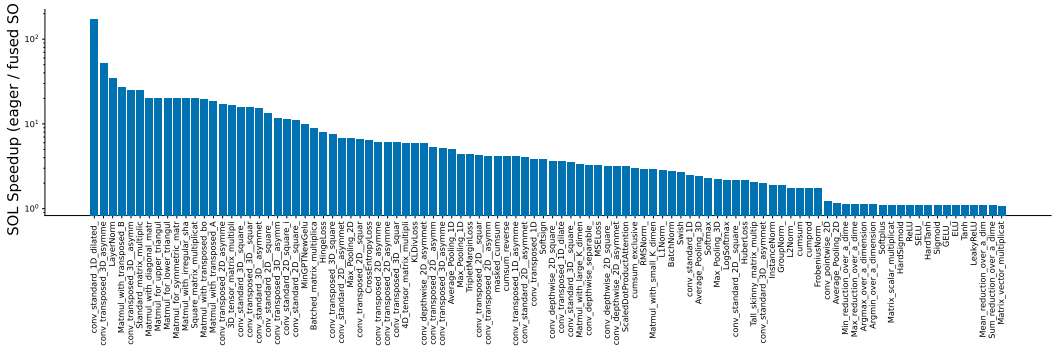


Figure 10: KernelBench L1: per-problem SOL speedup (geomean $4.3\times$).

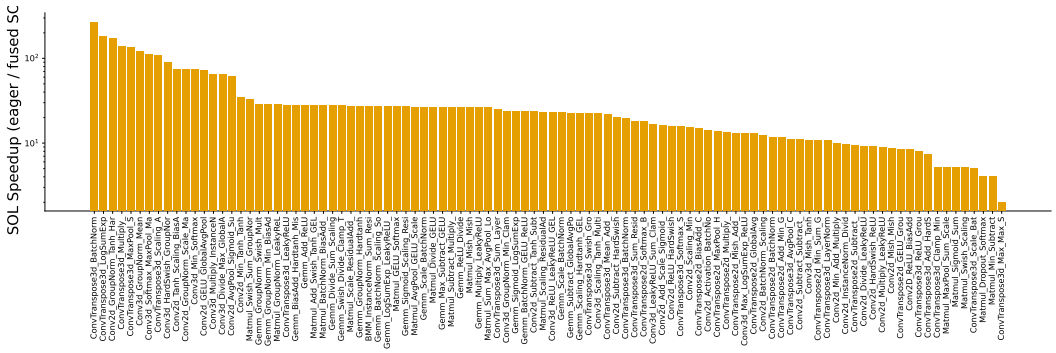


Figure 11: KernelBench L2: per-problem SOL speedup (geomean $21.6\times$).

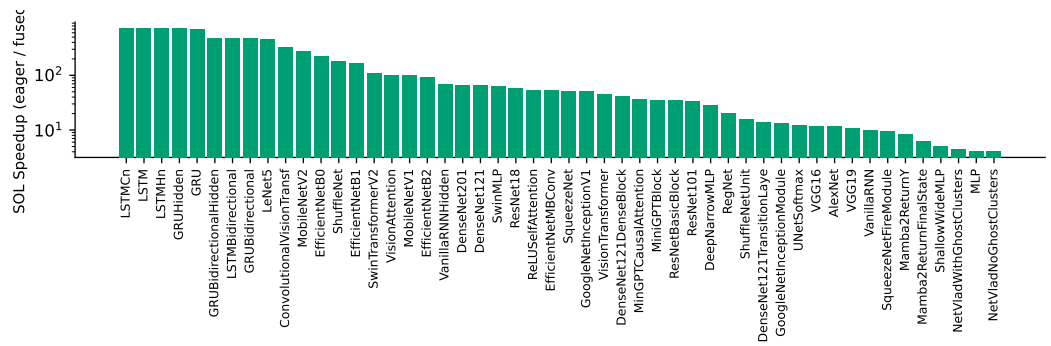


Figure 12: KernelBench L3: per-problem SOL speedup (geomean $54.6\times$).

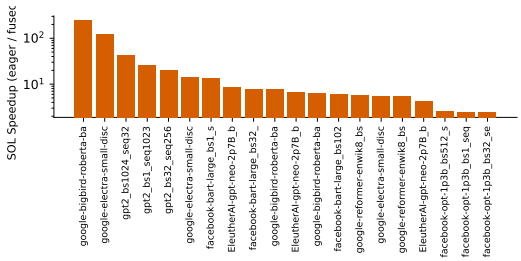
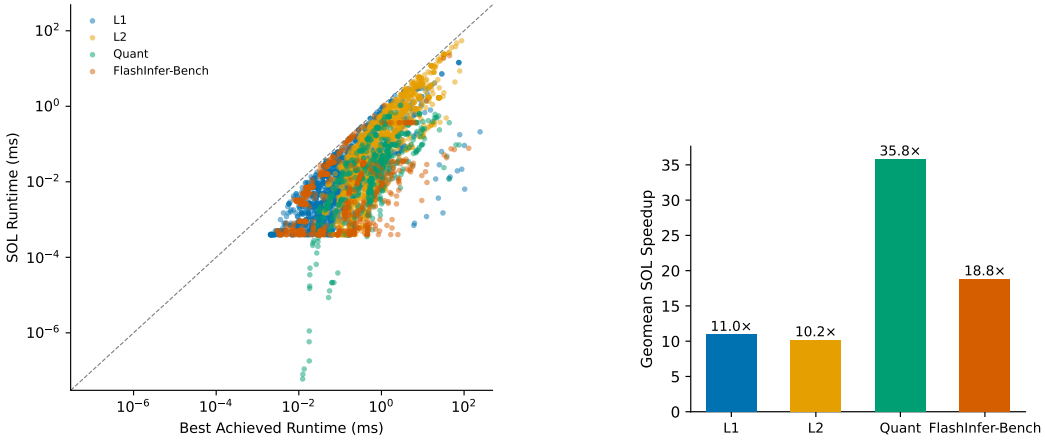


Figure 13: KernelBench L4: per-problem SOL speedup (geomean $10.2\times$).

B SOL-ExecBench SOL Analysis

SOL-ExecBench Lin et al. [2026] is a benchmark of 3,957 real-world GPU kernel workloads spanning four subsets: L1 (1,480 single operators), L2 (1,299 operator sequences), Quant (518 quantization kernels), and FlashInfer-Bench (660 attention kernels). Unlike KernelBench, which targets algorithmic diversity, SOL-ExecBench emphasizes production workload shapes drawn from deployed systems. We apply SOLAR to derive SOL scores for all 3,957 workloads. Figure 14a plots SOL runtime against best achieved runtime; points below the diagonal indicate optimization headroom. Figure 14b summarizes the geometric mean SOL speedup per subset.



(a) SOL vs. best achieved runtime.

(b) Geomean SOL speedup per subset.

Figure 14: **SOL-ExecBench SOL analysis.** (a) Each point is a workload; gap to the diagonal represents optimization headroom. (b) Quant kernels show the largest headroom ($35.8\times$), followed by FlashInfer-Bench ($18.8\times$), L1 ($11.0\times$), and L2 ($10.2\times$).

C Robotics Model SOL Analysis

Table 5 provides the full SOL analysis for seven robotics model configurations on Jetson Thor (1,035 TFLOP/s FP4, 273 GB/s). All models use W4A4 precision. The table covers compute and memory characteristics, unfused and fused SOL runtimes, achievable inference frequencies, and the hardware improvements required to meet target deployment frequencies.

Table 5: **Full robotics model SOL analysis on Jetson Thor.** All models are memory-bound. Target frequencies reflect deployment requirements (500 Hz for servo control, 30 Hz for vision). N/A indicates no target was specified for that configuration.

Metric	pi0	GR00T N1.6	GR00T N1.6 Sys 1	GR00T N1.6 Sys 2	DreamZero	DreamZero WAM	DreamZero VLB
Precision	W4A4	W4A4	W4A4	W4A4	W4A4	W4A4	W4A4
Denoising Steps	10	4	4	1	4	4	1
Compute (GFLOP)	4,007	1,725	665	1,059	20,899	17,179	3,720
Data Movement Unfused (GB)	5.95	5.38	3.43	1.95	27.0	22.0	5.0
Data Movement Fused (GB)	3.50	3.73	2.85	0.88	14.0	10.75	3.25
Arithmetic Intensity	1,145	463	233	1,210	1,493	1,598	1,145
Bottleneck	memory	memory	memory	memory	memory	memory	memory
Runtime Unfused (ms)	21.9	19.8	12.8	7.2	99.4	80.2	19.2
Runtime Fused (ms)	12.8	13.6	10.4	3.2	52.0	40.1	11.9
Frequency Unfused (Hz)	45.7	50.6	78.1	138.9	10.1	12.5	52.0
Frequency Fused (Hz)	78.0	73.5	96.2	312.5	19.2	25.0	84.0
Target Frequency (Hz)	500	N/A	500	30	N/A	500	30
Speedup Required	6.4×	N/A	5.2×	0.1×	N/A	20.0×	0.4×
Desired Π (TFLOP/s)	2,004	N/A	333	32	N/A	8,590	112
Π Improvement	1.9×	N/A	0.3×	0.03×	N/A	8.3×	0.1×
Desired β (GB/s)	1,750	N/A	1,425	26	N/A	5,375	98
β Improvement	6.4×	N/A	5.2×	0.1×	N/A	19.7×	0.4×

D Expert Kernel Headroom

Table 6 reports measured runtimes, unfused and fused SOL, and the corresponding headroom for five CuTe/CUTLASS and ten Triton workloads on H100 PCIe. The Triton workloads include implementations from Liger Kernel LinkedIn [2024], such as RMSNorm, RoPE, and SwiGLU. All 15 workloads pass output validation. Runtime / fused SOL ranges from $1.09\times$ to $4.68\times$ for CuTe/CUTLASS and from $1.08\times$ to $2.60\times$ for Triton.

Table 6: **Detailed CuTe/CUTLASS and Triton headroom on H100 PCIe.** All runtimes are in μs. Ratios are reproduced as reported and may differ from ratios of rounded runtimes. CutlassHopperFmha is a FlashAttention-3-like implementation.

Workload	Measured runtime	Unfused SOL	Fused SOL	Runtime / unfused SOL	Runtime / fused SOL
CuTe/CUTLASS					
VectorAdd	108.0	99	99	1.09×	1.09×
CutlassHopperGemm	861.4	454	454	1.90×	1.90×
CutlassPtrArrayBatchedGemm	729.7	364	364	2.01×	2.01×
CutlassHopperGett	851.5	182	182	4.68×	4.68×
CutlassHopperFmha	681.3	3,720	182	0.18×	3.75×
Triton					
LigerRMSNorm	288.9	395	263	0.73×	1.10×
LigerRoPE	1,284.0	2,110	528	0.61×	2.43×
LigerSwiGLU	428.1	395	395	1.08×	1.08×
TritonSoftmax	297.7	263	263	1.13×	1.13×
TritonMatmul	297.9	182	182	1.64×	1.64×
TritonBatchNorm	451.0	403	403	1.12×	1.12×
TritonAttention	1,506.9	10,798	727	0.14×	2.07×
TritonVAE	162.6	62	62	2.60×	2.60×
LlamaMLP	10,312.9	5,863	5,863	1.76×	1.76×
LigerLlamaBlock	17,985.3	10,226	10,226	1.76×	1.76×

TritonAttention takes $1{,}506.9\,\mu$s versus $727\,\mu$s fused SOL and $10{,}798\,\mu$s unfused SOL. Beating the unfused estimate reflects eliminated materialization, not a violation of the fused bound. The same distinction applies to CutlassHopperFmha, LigerRMSNorm, and LigerRoPE, whose measured runtimes are below unfused SOL but above fused SOL. These comparisons distinguish headroom due to eliminated intermediate traffic from the remaining gap to fused SOL; the latter is an analytical floor, not a guaranteed attainable runtime.

NeurIPS Paper Checklist

1. **Claims.** Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? Answer: [Yes] The abstract and introduction describe the source-to-SOL framework, its separation of generative translation from deterministic analysis, and its software, algorithm, and hardware use cases, supported by Sections 3 and 4. Translation validation is empirical rather than a formal correctness guarantee, and SOL estimates are conditional on workload and hardware-model assumptions (Sections 3.1.2 and 3.3).
2. **Limitations.** Does the paper discuss the limitations of the work performed by the authors? Answer: [Yes] Section 3.1.2 discusses missed translation errors, output-equivalent translations with inflated costs, and input-conditioned control-flow coverage. Section 3.3 states the peak-throughput, overlap, and storage assumptions and discusses shape-based modeling limits and hardware variability.
3. **Theory assumptions and proofs.** For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof? Answer: [N/A] The paper presents a systems framework rather than new formal theorems. Section 3.3 defines the roofline estimate (Equation (1)) and its throughput, bandwidth, overlap, and storage assumptions; empirical translation validation is not a proof of equivalence.
4. **Experimental result reproducibility.** Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)? Answer: [No] Sections 3 and 4 describe the analytical pipeline, workloads, and hardware settings, and the public code provides a path to reproducing the analysis. A complete experiment-specific specification of LLM generation settings and timing procedures, together with a corresponding experimental artifact, is not yet provided for all reported results.
5. **Open access to data and code.** Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material? Answer: [No] SOLAR code is publicly available at `https://github.com/NVlabs/SOLAR` under Apache-2.0, with installation instructions, examples, and analysis commands. Experiment scripts are to be provided through GitHub.
6. **Experimental setting/details.** Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results? Answer: [Yes] Section 4 specifies the evaluated workload groups, hardware targets, FA4 shapes and precision, Qwen3-4B sweep axes, and robotics deployment targets; Appendices C and D provide additional configurations and measured comparisons. No model training, data splitting, or optimizer selection is performed; full experimental reproducibility remains subject to the gaps stated above.
7. **Experiment statistical significance.** Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments? Answer: [No] We report per-workload results, geometric means, headroom ranges, and aggregate translation success rates, but not repeated-run confidence intervals or significance tests. Analytical bounds are deterministic for fixed translated workloads and hardware assumptions, whereas LLM translation and measured hardware runtimes can vary; this variability is not quantified with error bars.
8. **Experiments compute resources.** For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments? Answer: [No] Section 4 and appendix D identify hardware platforms for runtime measurements and distinguish them from analytical projection targets. We do not provide a complete accounting of CPU/GPU hours, host-memory requirements, LLM token usage, or total compute including preliminary and failed runs.
9. **Code of ethics.** Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`? Answer: [Yes] This study develops a performance-analysis framework without human-subject experiments or collection of personal data. We credit external work, disclose the use

of LLMs and the limitations of numerical validation, and include funding and competing-interest disclosures.

10. **Broader impacts.** Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed? Answer: [No] The paper discusses benefits for optimization and hardware planning but does not include a systematic assessment of negative societal impacts. Incorrect or overly optimistic estimates could misguide resource allocation or real-time deployment decisions; SOL estimates should not replace measured performance or safety validation.
11. **Safeguards.** Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)? Answer: [N/A] The released asset is performance-analysis software, not a new pretrained generative model or a scraped dataset. The work therefore does not involve a high-risk model or dataset release of the kind addressed by this question.
12. **Licenses for existing assets.** Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected? Answer: [No] External assets are cited, including KernelBench, JAXBench, SOL-EXECBENCH, FlashInfer, fvcore, ptflops, Timeloop, and AccelForge, and the public repository includes third-party notices. However, the paper does not enumerate the exact version, license, and terms for every evaluated codebase, model, and external service, so we do not claim complete documentation of all parts of this question.
13. **New assets.** Are new assets introduced in the paper well documented and is the documentation provided alongside the assets? Answer: [Yes] The new asset is the public SOLAR software, whose GitHub repository documents installation, command-line and Python interfaces, examples, architecture configurations, validation limitations, and its Apache-2.0 license. This software documentation is distinct from the complete experimental artifact that is not yet available; no new datasets or pretrained models are released.
14. **Crowdsourcing and research with human subjects.** For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)? Answer: [N/A] This work does not involve crowdsourcing or human subjects.
15. **Institutional review board (IRB) approvals or equivalent for research with human subjects.** Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained? Answer: [N/A] This work does not involve human subjects research.
16. **Declaration of LLM usage.** Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required. Answer: [Yes] Sections 3.1 and 3.1.2 describe LLM-based source-to-IR translation, numerical validation, and feedback-driven retries. The translation frontend uses Opus 4.6; the separate direct-SOL-estimation baseline uses Claude Code with Opus 4.6 (Table 3).