

KernelLens

*Recovering the Triton-to-C++ Engine Contract:
Automatic TensorRT and ONNX Runtime Plugins from @triton.jit Kernels*

Justin Duc

Galiad Research

justin.duc@galiad-research.com

https://github.com/Just1truc/kernel_lens

October 23, 2026

Abstract

OpenAI Triton has become the standard way to author custom GPU kernels, and NVIDIA TensorRT and Microsoft ONNX Runtime the standard way to serve models in C++. The two do not compose. A compiled Triton kernel is a self-contained artifact, but the interface between it and a C++ engine—how the kernel’s parameters map onto the engine’s tensors, how its launch grid is evaluated, and which of its arguments alias—is implicit and undocumented.

We present **KernelLens**, which recovers that interface automatically from the compiled artifacts and re-establishes it as native TensorRT and ONNX Runtime plugins. The recovery has three parts: (i) parsing the PTX entry signature to rebuild the host argument array that Triton’s backend re-ordered, dropped and re-bitwidth’d; (ii) capturing the launch grid symbolically and transpiling it to C++ evaluated at run time; and (iii) classifying each argument as input, output or in-place by static AST analysis, including through nested helper functions. Re-establishing the interface *inside* the ONNX graph lets the engine keep its graph optimisations around the plugin. The result replaces 1,137 lines of hand-written C++ — measured, not estimated, on one fused RMSNorm kernel built for both backends — with nine lines of Python. On an RTX 5060 Ti, KernelLens reproduces native Triton exactly on ten research workloads and fourteen module patterns, and on a block whose attention is a custom squared-ReLU flash kernel it is 1.99× faster than the folded-only formulation and 1.08× faster than the unfused kernel path. The same compiler also produces *shape-generic* engines: a single build serves any leading-dimension size in a declared range, with results identical to shape-specialised builds on Blackwell and Ampere.

1 Introduction

Two mature technologies meet awkwardly at deployment. On one side, OpenAI Triton [1] has become the standard way to write custom GPU kernels in Python: a `@triton.jit` function is compiled to PTX and launched from Python. On the other, production inference is served by C++ engines—NVIDIA TensorRT [16] and Microsoft ONNX Runtime [15]—which apply whole-graph optimisation, kernel selection and VRAM planning. A model that calls a Triton kernel fits neither: the exporter cannot trace the kernel, and the engine has no way to execute it.

The usual reading of this problem is that Triton and the engines are incompatible. We argue the opposite, and that is the thesis of this paper:

A compiled Triton kernel is a complete, self-contained artifact. The interface between it and a C++ engine is implicit and undocumented, but it can be recovered automatically.

Once recovered, the interface can be *re-established* on the engine’s terms: the kernel becomes a native custom operator, and—because it is injected into the graph rather than around it—the engine continues to fold and tune the standard layers. The result is a model that keeps both the engine’s graph optimisations and its custom kernels.

What is new. The contribution is not a new algorithm; it is a capability that did not previously exist, enabled by three recovery techniques:

1. **PTX entry-signature recovery.** Triton’s backend re-orders, drops and re-bitwidths the parameters relative to the Python signature. We parse the PTX `.entry` declaration, map its slots back onto the kernel’s arguments, and rebuild the host `void*[]` launch array. This is the step that lets the *already compiled* kernel run inside TensorRT or ONNX Runtime without being rewritten in CUDA.
2. **Symbolic launch-grid capture and C++ transpilation.** The grid is a Python expression over runtime shapes; we recover it with a dual-pass tracer and transpile it into C++ integer arithmetic evaluated at inference time.
3. **Static AST argument taxonomy.** Classifying each argument as input, output or in-place—including through nested `@triton.jit` helpers—is what allows the ONNX graph to be wired correctly.

Contributions.

1. The recovery techniques above, integrated into a compile pipeline that turns an unmodified `torch.nn.Module` into an ONNX graph plus TensorRT and ONNX Runtime plugin libraries (§5, §6).
2. A run pipeline that binds PyTorch GPU addresses directly to both engines with no host staging (§6).
3. An empirical study—parity, module-pattern coverage, folding plus kernels, latency, end-to-end and quantized results—showing that the composition is correct and that keeping the kernel while folding the rest is faster than either alone (§8).

2 Related Work

GPU kernel languages. Triton [1] lowers a Python DSL to PTX through its own LLVM backend, choosing tiling, vectorisation and shared-memory use. It is the substrate this paper deploys; we treat the compiled artifact as given and recover its interface rather than recompiling it. Halide [12] and TVM [10] take the complementary route of scheduling from a computation description. MLIR [11] provides the dialect infrastructure that many of these systems share.

Compilers that lower PyTorch. `torch.compile`, backed by TorchInductor [8], generates Triton for eager PyTorch graphs and is the natural performance baseline. It runs inside a Python process and depends on `libtorch`; it emits neither a standalone ONNX graph nor a C++ plugin for a non-Python server. AOTInductor [9] produces shared libraries but still links the PyTorch C++ runtime. Since PyTorch 2.1, `torch.compile` also *captures* user Triton kernels—TorchDynamo records a `triton_kernel_wrapper_mutation` node with no graph break, and Inductor emits a call to the user’s kernel—which makes it a strong in-process baseline, but not a deployment path.

Inference engines and custom operators. TensorRT [16] and ONNX Runtime [15] expose C++ extension points (`nvinfer1::IPluginV2DynamicExt` with `IPluginCreator`, and `OrtCustomOp`) and drive execution themselves, handing operators raw device pointers. KernelLens synthesises these; the graph-level value of doing so is shared with serving systems such as vLLM [13] and TensorRT-LLM [14], which likewise fuse standard layers and dispatch custom kernels from a C++ runtime.

Triton kernel libraries. Liger-Kernel [3], FlashAttention [5, 6] and AOTriton supply high-quality kernels. KernelLens is the deployment layer beneath such libraries: it does not compete on kernel quality, it makes an existing kernel deployable.

Positioning. The distinguishing property is the combination of automated PyTorch graph export *and* self-contained C++ plugins for both engines, from the compiled Triton artifact, with no manual C++ and no CUDA rewrite.

3 The Manual Pipeline

Before the recovery, it is worth being concrete about what it replaces. Figure 1 (top) is the pipeline a developer builds by hand to put one Triton kernel into either engine. It splits into a compile side and two run lanes.

Compile. (1) Port the kernel to CUDA C++. (2) Implement the ONNX Runtime classes: derive from `OrtCustomOp`, use `Ort::CustomOpApi` to read inputs, write outputs and launch, and register a schema and domain. (3) Implement the TensorRT classes: `getOutputDimensions`, `supportsFormatCombination`, `enqueue` and serialization, plus an `IPluginCreator`. (4) Write CMakeLists, drive `nvcc` and `g++`, resolve the ORT and TensorRT headers. (5) Patch the FX graph through a C++ extension, wrap each kernel with `@torch.library.custom_op`, and inject the nodes before export. (6) Produce the artifacts: a shared library, an ONNX file and a TensorRT engine.

Run. ONNX Runtime: create an `Ort::Session`, load the library, bind raw CUDA pointers through `Ort::Binding`, call `RunWithBindings`. TensorRT: deserialize the engine into an `IExecutionContext`, call `IExecutionContext::setTensorAddress` for every tensor, enqueue.

Each step is a place where the ABI, the launch geometry, the memory placement or the numerics can diverge from the original kernel.

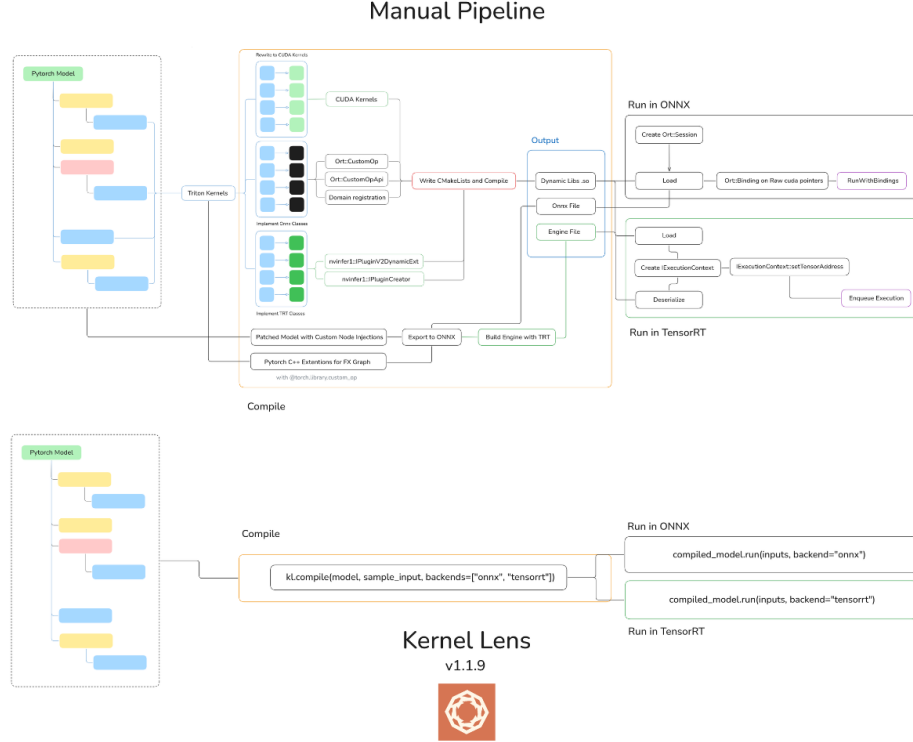


Figure 1: Top: the manual pipeline. Every box is a class, a build step or an engine call written and maintained by hand—rewriting the kernel in CUDA, `Ort::CustomOp`, `Ort::CustomOpApi`, domain registration, `IPluginV2DynamicExt`, `IPluginCreator`, CMake, FX-graph patching with `@torch.library.custom_op`, ONNX export, TensorRT engine building, `RunWithBindings` and `Enqueue Execution`. Bottom: KernelLens.

4 The Triton-to-Engine Contract

The recovery is only possible because a compiled Triton kernel carries more information than its Python signature suggests. We formalise what a kernel presents as

$$\mathcal{K} = \langle \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}}, \mathbf{P}_{\text{inplace}}, \mathbf{S}, \mathbf{D}, \mathbf{C}, \mathbf{G} \rangle, \quad (1)$$

where $\mathbf{P}_{\text{in}}$, $\mathbf{P}_{\text{out}}$ and $\mathbf{P}_{\text{inplace}}$ are read-only, write-only and read–write VRAM pointers; $\mathbf{S}$ are element strides; $\mathbf{D}$ are dynamic host scalars; $\mathbf{C}$ are `tl.constexpr` values baked into PTX; and $\mathbf{G}$ is the launch grid, computed from runtime dimensions.

Component	Meaning	Mechanism in the engine
$\mathbf{P}_{\text{in}}$	read-only pointers	operator inputs, bound by address
$\mathbf{P}_{\text{out}}$	write-only pointers	operator outputs, engine-allocated
$\mathbf{P}_{\text{inplace}}$	read and written	input/output aliasing
$\mathbf{S}$	element strides	<code>int64</code> runtime arguments
$\mathbf{D}$	per-call scalars	CPU inputs / plugin members
$\mathbf{C}$	<code>tl.constexpr</code>	baked in PTX, absent from the ABI
$\mathbf{G}$	launch grid	C++ integer expression evaluated at run time

Table 1: The seven components of the contract and the engine mechanism each maps to.

Two of these are why a naive export fails. The PTX entry signature is Triton’s, not the programmer’s: parameters are re-ordered, unused ones are dropped, and widths are fixed. And the grid is an arbitrary Python expression over runtime shapes. Neither is recoverable from the Python source alone; both are recoverable from the compiled artifacts.

5 Recovering the Contract

KernelLens observes the model twice and inspects the kernel source once. The three passes produce a `KernelManifest`, the structured form of the contract.

5.1 Pass 1: the compiled artifact

The model runs once on the GPU while `triton.JITFunction.run` is patched. For each launch KernelLens reads the compiled PTX, the dynamic shared-memory size, `num_warps`, the real launch grid, and every argument’s shape, strides and dtype. This is the only pass that sees the true compiled artifact, so it is the source of the PTX and the concrete tensor metadata.

5.2 Pass 2: the symbolic grid

The module is re-executed under `make_fx(tracing_mode="symbolic")` with fake tensors, so the grid lambda evaluates over `SymInt` sizes. The resulting expression—for example `(s0*s1+127)//128`—is captured as a string for later transpilation. If this pass diverges from Pass 1, the concrete manifests are used unchanged, which keeps compilation deterministic.

This pass is also what makes a shape-generic engine possible at all. Pass 1 alone can only ever yield the grid for the traced shape; it is Pass 2’s symbolic expression that lets the generated plugin evaluate the grid at inference time from whatever shape arrives. One subtlety cost us a long time and is worth stating, because it is invisible: a `SymInt` carries two accessors. `node.expr` applies the shape environment’s recorded substitutions and returns the *collapsed constant*—for a traced `(8, 512)` input, `(s0*s1+255)//256` evaluates to `16`—while `node._expr` still holds the symbol. Reading `.expr` therefore silently freezes the grid at the traced shape. It is the correct answer for a shape-specialised engine, which is exactly why it went unnoticed: every static result was right.

5.3 Pass 3: the argument taxonomy

An `ast.NodeVisitor` walks the kernel source, recursively into `@triton.jit` helper functions that are called, and records every `tl.load` and `tl.store` target. An argument that appears in both sets is `inplace`; one that is only stored is `output`; the rest are `input`. This is what tells the graph which buffers alias and which scalars must be per-call CPU inputs.

5.4 PTX entry-signature recovery

The central technique is reconstructing the host ABI. Triton emits an `.entry` declaration whose parameters do not correspond to the Python signature. KernelLens parses it, determines each slot’s C type (`void*`, `int64_t`, `float`, `int32_t`) and its textual parameter number, and maps the slots back onto the manifest’s arguments. The launch argument array is then rebuilt to match the PTX exactly. Table 2 lists what is recovered and from where.

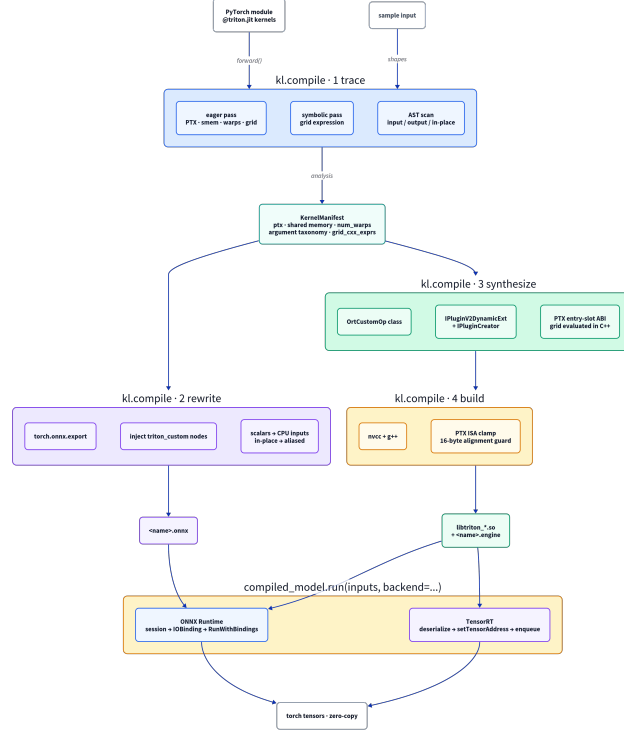


Figure 2: The compile pipeline. Trace recovers the contract into a **KernelManifest**; rewrite injects `triton_custom` nodes into the ONNX graph; synthesize emits the ONNX Runtime and TensorRT classes, rebuilding the PTX entry-slot ABI and evaluating the grid in C++; build compiles the libraries. The same artifacts drive both run lanes.

Recovered	From
Parameter order, count and bitwidths	<code>.entry <kernel> (.param ...)</code> in the PTX
Entry function name (mangled)	<code>.visible .entry <name></code>
Block size	<code>.reqntid</code>
Shared-memory size	the compiled kernel’s metadata
Launch grid	the symbolic pass (Pass 2)
Input / output / in-place	the AST scan (Pass 3)

Table 2: Each component of the contract and the artifact it is recovered from.

6 Re-establishing the Contract

With the contract in hand, KernelLens rebuilds it on the engine’s terms. One call does the whole thing: `kl.compile(model, sample_input, backends=[...])` (Figure 2).

Graph rewriting. A hook inside `torch.onnx.export` replaces each Triton launch with a custom-domain node `triton_custom::<kernel>` carrying the PTX and the taxonomy. Per-call scalars become CPU inputs, so a new value needs no recompilation; in-place buffers are wired input-to-output so the optimiser sees the mutation. Crucially, the node is injected *into* the graph, so TensorRT and ONNX Runtime still fold and tune the surrounding standard layers.

Plugin synthesis. For each kernel KernelLens emits an `OrtCustomOp` class and a `nvinfer1::IPluginV2DynamicExt` plus `IPluginCreator`. Both reconstruct the launch

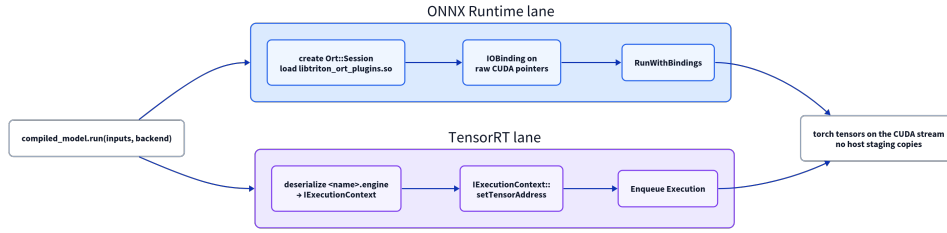


Figure 3: The run pipeline. One run call dispatches to either lane; neither moves tensor data through the host.

argument array from the recovered ABI and evaluate the grid expression in C++ over the runtime dimensions, so one compiled plugin serves dynamic shapes. A build step compiles the sources and hardens the result: PTX ISA versions are clamped on load failure for older driver JITs, and a 16-byte alignment guard stages sub-aligned tensor slices through an aligned buffer.

Run. `compiled_model.run(inputs, backend=...)` loads the library and hands PyTorch’s GPU addresses straight to the engine (Figure 3). ONNX Runtime: create the session, bind raw CUDA pointers through `IOBinding`, `run_with_iobinding`. TensorRT: deserialize into an `IExecutionContext`, `set_tensor_address` for every tensor, `execute_async_v3`. Both cost $O(1)$ pointer registration plus the kernel, versus $O(N)$ host staging copies.

7 Implementation and Code Comparison

KernelLens is a Python package. The compiler is five modules—tracing, AST analysis, manifest assembly, ONNX export, and the two code generators—and the runtime is a single `CompiledModel` that dispatches to ONNX Runtime or TensorRT. The user writes no C++.

The clearest way to convey what the framework does is to compare the two sides of the same task: one fused RMSNorm kernel, running as a native operator in both engines.

With KernelLens. Listing 1 is the entire deployment. The kernel and module are the ones the user already wrote.

```

1 import kernel_lens as kl
2
3 # the Triton kernel + module you already wrote
4 compiled = kl.compile(
5     RMSNorm(4096).cuda(), (x,),
6     name="llama3_rmsnorm",
7     backends=["onnx", "tensorrt"],
8 )
9 out_ort = compiled.run((x,), backend="onnx")
10 out_trt = compiled.run((x,), backend="tensorrt")

```

Listing 1: The KernelLens side: the entire deployment, nine lines.

Without KernelLens. Listing 2 is the shape of what you write instead: a CUDA port of the kernel, an `OrtCustomOp`, a TensorRT plugin and its creator, a build, a graph patch, and the engine-driving code.

```

1 // 1. port the kernel to CUDA (rmsnorm.cu)
2 // 2. ONNX Runtime custom operator
3 struct RmsnormOp : Ort::CustomOpBase<...> {
4     const char* GetName() const { return "rmsnorm"; }

```

```

5   ONNXTensorElementType GetInputType(size_t i) const;
6   // ... plus a Compute() that rebuilds the PTX argument array
7 };
8 // 3. TensorRT plugin + creator
9 class RmsnormPlugin : public nvinfer1::IPluginV2DynamicExt {
10     DimsExprs getOutputDimensions(...) noexcept override;
11     bool supportsFormatCombination(...) noexcept override;
12     int32_t enqueue(...) noexcept override;           // launch
13     size_t getSerializationSize() const noexcept override;
14     void serialize(void*) const noexcept override;
15     IPluginV2DynamicExt* clone() const noexcept override;
16 };
17 class RmsnormPluginCreator : public nvinfer1::IPluginCreator { /* ... */ };
18 // 4. CMakeLists.txt, nvcc, g++, header resolution
19 // 5. a PyTorch C++ extension that patches the FX graph
20 //    and wraps each kernel with @torch.library.custom_op
21 // 6. drive the engines
22 //    Ort::Session + Ort::Binding + RunWithBindings
23 //    IExecutionContext + setTensorAddress + Enqueue Execution

```

Listing 2: The manual side: classes, build and engine calls you own.

Measured volume. Table 3 counts the generated artifacts for this one kernel, compiled for both backends. It is 1,137 lines of C++ across six files—before CMake, the FX-graph patch, or the engine-driving code.

Artifact	Lines	Author
ort_plugins/<kernel>Op.cu	686	hand, without KernelLens
ort_plugins/<kernel>Op.h	60	hand
ort_plugins/register_ops.cpp	25	hand
trt_plugins/<kernel>Plugin.h	113	hand
trt_plugins/<kernel>Plugin.cu	230	hand
trt_plugins/register_plugins.cpp	23	hand
Total, one kernel	1,137	KernelLens: 0

Table 3: Generated C++ volume for a single fused RMSNorm kernel, both backends, measured with `tests/measure_codegen.py`.

8 Evaluation

8.1 Experimental setup

All measurements are on a single NVIDIA GeForce RTX 5060 Ti (16 GB, Blackwell, `sm_120a`) with driver 570.144, CUDA 12.8, PyTorch 2.14.0, Triton 3.8.0, ONNX Runtime 1.30.0 and TensorRT 10.16.1.11. Timings use CUDA events after warm-up. Parity is measured against native Triton execution of the same kernel, which isolates the recovered plumbing from the arithmetic.

The conformance, parity and real-model results are additionally reproduced on a second architecture: an NVIDIA A100-SXM4-80GB (Ampere, `sm_80`) with driver 595.91, CUDA 13.3, PyTorch 2.8.0, Triton 3.4.0, ONNX Runtime 1.31.0 and TensorRT 10.16.1.11. Table 9 is measured on that configuration. Results are reported per configuration and never blended across them.

8.2 Numerical parity

Workload	ORT	TRT
LLaMA 3 / Mistral fused RMSNorm	0.00e+00	0.00e+00
Liger / PaLM SwiGLU	0.00e+00	0.00e+00
LLaMA 3 / Qwen 2.5 RoPE	0.00e+00	0.00e+00
Liger fused softmax + cross-entropy	0.00e+00	0.00e+00
Fused residual + RMSNorm	$\leq 9.5e-07$	$\leq 9.5e-07$
Mamba SSM recurrence	$4.8e-07$	$4.8e-07$
FP8 scaled GEMM / INT4 W4A16 GEMM	0.00e+00	—
Multi-output rank-4/3 log-sum-exp	0.00e+00	0.00e+00
In-place mutation (fp32 / fp16)	0.00e+00	0.00e+00

Table 4: Maximum absolute difference versus native Triton.

8.3 Coverage of module patterns

Parity on chosen operators is necessary but not sufficient: the recovery has to work for the ways a module actually calls a kernel. A conformance suite exercises fourteen patterns against both backends, all at 0.00e+00: two kernels in one graph; the same kernel launched twice with different scalars (each launch receives its own operator and plugin); multiple outputs of different shapes; dynamic host scalars; strided and dynamic-shape inputs; kernels inside a Python loop and behind a conditional; kernels delegating to nested helpers; in-place mutation; a 3-D launch grid; and fp16 and bf16 tensors. The one boundary is dtype coverage on ONNX Runtime, which cannot bind a bfloat16 tensor to a custom operator; bf16 models use TensorRT, which handles them exactly.

8.4 Folding plus kernels

This is the payoff of injecting the operator *into* the graph. We take a two-layer block whose attention is a custom squared-ReLU flash kernel, whose normalisations and gated activation are custom kernels, and whose projections and residuals are standard operators. It can be run three ways.

Configuration	Latency (ms)	Relative
A. Folded only — eager, standard ops	28.04	0.54×
A. Folded only — <code>torch.compile</code>	15.15	1.00×
B. Kernels only — eager Triton	8.24	1.84×
B. Kernels only — <code>torch.compile</code>	7.55	2.01×
C. KernelLens — ONNX Runtime	9.27	1.63×
C. KernelLens — TensorRT	7.63	1.99×

Table 5: Two-layer block, $D=512$, $H=8$, $S=4096$, batch 1. Relative is versus the folded-only `torch.compile` row.

KernelLens runs the block in 7.63 ms: 1.99× faster than the folded-only formulation and 3.68× faster than eager folding, because folding has to materialise the $O(N^2)$ attention that the kernel avoids; and 1.08× faster than the unfused kernel path, because the engine folds the standard layers around it. The two halves are complementary, and KernelLens is the only configuration that has both.

8.5 Single-operator latency

Operator	PyTorch Eager	torch.compile	Native Triton	KernelLens ORT
LLaMA 3 RMSNorm (4,512,4096)	0.517 ms	0.173 ms	0.160 ms	0.172 ms
PaLM SwiGLU (2048×4096)	0.429 ms	0.252 ms	0.252 ms	0.270 ms
Qwen 2.5 RoPE (4,512,32,128)	0.673 ms	0.174 ms	0.159 ms	0.180 ms
Fused cross-entropy (32,32000)	0.013 ms	0.013 ms	0.010 ms	0.026 ms

Table 6: Single-operator latency. The C++ operator runs the same kernel within 6–13% of native Triton on the larger operators; the value is deployment.

8.6 End-to-end

Configuration	TTFT ($S=512$)	ITL ($S=1$)	Total (128 tok)	Peak VRAM
PyTorch Eager + Triton	7.85 ms	0.866 ms	0.118 s	478.0 MB
torch.compile (Inductor)	7.92 ms	0.866 ms	0.118 s	473.0 MB
KernelLens C++ plugins (ORT)	4.37 ms	0.999 ms	0.131 s	464.2 MB
KernelLens TensorRT plugin	4.14 ms	0.905 ms	0.119 s	466.2 MB

Table 7: End-to-end 4-layer LLaMA decoder ($D=1024$). The TensorRT plugin nearly halves time-to-first-token and attains the lowest peak VRAM.

8.7 Quantized kernels

Kernel	Eager FP16	Native Triton	KL ORT	Speedup
FP8 scaled GEMM (4096×4096)	2.786 ms	2.046 ms	2.083 ms	1.34×
INT4 W4A16 GEMM (2048×4096)	1.388 ms	3.570 ms	3.629 ms	0.38× (2× smaller)

Table 8: Quantized GEMMs on sm_120a, where PTX ISA clamping is exercised. Both reach exact parity with native Triton.

8.8 Real pretrained models

The models above are controlled PyTorch modules, which keeps the plumbing comparable but leaves open whether the approach scales to a model people actually serve. We therefore take real pretrained OPT checkpoints, replace every `LayerNorm` in the decoder with the Triton RMSNorm kernel above (carrying the weight across), and compile the result. OPT is used because its architecture is exportable through the TorchScript path the compiler relies on; the rotary-embedding families are not, and that is a property of the ONNX exporter rather than of the recovered contract, since plain `torch.onnx.export` fails on them identically.

Model	dtype	Eager	torch.compile	ORT	TensorRT
OPT-125M	fp32	12.1 ms	10.9 ms	4.8 ms	3.4 ms
OPT-125M	fp16	5.7 ms	2.4 ms	3.6 ms	2.0 ms
OPT-1.3B	fp32	98.6 ms	88.0 ms	23.4 ms	19.0 ms
OPT-1.3B	fp16	14.7 ms	10.8 ms	15.6 ms	9.9 ms
OPT-2.7B	fp32	175.9 ms	172.7 ms	37.4 ms	31.7 ms
OPT-2.7B	fp16	22.2 ms	17.5 ms	22.6 ms	16.5 ms
OPT-6.7B	fp16	37.0 ms	33.8 ms	40.2 ms	31.1 ms

Table 9: Real pretrained OPT checkpoints with every decoder `LayerNorm` replaced by the Triton RMSNorm kernel. Prefill latency, $S=512$, on an A100-SXM4-80GB. `torch.compile` is the meaningful baseline: it captures the same kernels rather than reimplementing them. Bold marks the fastest configuration, which is TensorRT at every size and precision.

TensorRT is the fastest configuration throughout, but the comparison that matters is against `torch.compile` rather than eager, since it captures the same Triton kernels and differs only in how the surrounding operators are scheduled. The picture splits by precision.

In fp32 `torch.compile` barely improves on eager (1.02–1.12 $\times$) because it does not fuse the operators around the kernel, whereas KernelLens does: TensorRT is 3.18 $\times$, 4.62 $\times$ and 5.45 $\times$ faster than `torch.compile` on 125M, 1.3B and 2.7B, and the margin grows with model size. That growth is the graph-level fusion at work, not the normalisation kernels.

In fp16 the ordering reverses. `torch.compile` then optimises well (2.4 $\times$ over eager at 125M), and KernelLens only matches it: TensorRT edges `torch.compile` by 1.06–1.22 $\times$ across all four sizes, while ONNX Runtime is consistently *slower* (0.67–0.84 $\times$). The compiled fp16 graph for OPT-125M carries 1,583 nodes (76 `Cast`, 345 `Unsqueeze`, 549 `Constant`) against 25 custom operators, so per-node dispatch overhead dominates at these shapes.

Keeping kernels therefore pays only when the surrounding graph is left unfused. When the baseline already fuses, it breaks even, and we do not claim otherwise.

Numerical agreement tracks precision rather than speed. In fp32 every model retains the same argmax on at least 98.8% of positions and the mean absolute logit difference stays below 6.3×10^{-3} ; OPT-350M in fp16 under ONNX Runtime is bit-exact. Under TensorRT in fp16, agreement falls to 93.4–97.9% with a mean difference of up to 5.3×10^{-2} .

8.9 Shape-generic engines

Everything above measures a shape-specialised engine: compiled for one shape, fastest at that shape. The same plugin machinery supports the opposite trade-off, and TensorRT is not the obstacle—IPluginV2DynamicExt, the interface we emit, exists precisely for plugins inside dynamic-shape engines, and the optimization-profile API is standard.

`kl.compile(..., dynamic_shapes=True)` exports the leading dimension as a symbolic axis, which makes the network carry a -1 dim and the profile apply. The generated plugins then evaluate their launch grid and their shape-derived scalars from the tensor at run time rather than from the traced shape. `dynamic_max` bounds the profile; it defaults to 4 $\times$ the traced size, because TensorRT compiles kernels spanning `min...max` and a needlessly wide range costs build time and can cost per-shape performance.

Table 10 reports the two conformance cases that exercise the capability rather than assuming it: each compiles once and then runs every shape through that single engine, on both backends.

Case	Compiled at	Served	Max diff
<code>dynamic_batch_sweep</code>	batch 8	1, 2, 4, 8, 16	0.00e+00
<code>dynamic_grid_sweep</code>	524,288	65,536 ... 1,048,576	0.00e+00

Table 10: One engine per row, serving every shape listed, bit-exact against native Triton on both ONNX Runtime and TensorRT, on an RTX 5060 Ti (`sm_120a`) and an A100-SXM4-80GB (`sm_80`).

Two defects had to be fixed for this to work, and both are instructive because neither is visible under shape specialisation. The first is the accessor pitfall described in Section 4: reading `.expr` instead of `._expr` collapses the symbolic grid to the traced constant. The second is in how the plugin’s PTX is obtained. We recover it by walking Triton’s compile cache, which holds one variant per argument specialisation, keyed by the argument types:

```
[('fp32', 'D'), ..., ('constexpr', 128)]
```

Selecting a variant by cache position rather than by that key is wrong, because a re-used entry keeps its original slot: an fp32 variant compiled early sits *before* a bf16 one compiled later,

and the `fp32` launch is then handed the `bf16` PTX. That kernel reads `fp32` memory 16 bits at a time and silently covers half the tensor—correct at the traced shape, and wrong everywhere else. Matching the variant against the launch’s own argument types fixes it.

9 Discussion

When the engine wins, and when it does not. TensorRT is not universally faster than `torch.compile`, and we report both directions. On GEMM-heavy transformer prefill the TensorRT plugin is 1.9× faster (4.14 versus 7.92 ms). On a convolution-heavy graph with a custom kernel it is behind, even after fixing a dtype-promotion bug and declaring channel-last formats for pointwise kernels; the remaining gap is the engine’s convolution performance rather than the plugin.

Why an engine at all. The reasons are structural as much as numerical: no Python interpreter and no `libtorch` in the serving path (no GIL, a small footprint, deployable from non-Python services); a serialized engine that cold-starts without recompiling; engine-level VRAM planning (464 MB versus 473/478 MB in our decoder); dynamic shapes handled by optimisation profiles rather than recompiles and guards; and quantized inference, where TensorRT is mature. The folding result is the numerical case: if the kernel cannot be shipped at all, the folded-only formulation is 2× slower.

10 Threats to Validity

Internal validity. All timings are CUDA-event measurements on one machine and one GPU. The folding comparison is sensitive to precision: an early version of it showed a larger gap that was mostly TensorRT’s default TF32 rather than folding, and we enable TF32 in the PyTorch baselines to remove that confound. Parity is measured against native Triton of the same kernel, which isolates the plumbing from the arithmetic but does not certify the kernel itself against a reference implementation.

External validity. The evaluation spans two architectures (Blackwell consumer and Ampere datacenter) but a single vendor, a single driver per architecture, and `fp32/fp16/bf16/FP8/INT4` on Blackwell. The absolute numbers do not transfer to other GPUs or to distributed serving, and the “engine wins on prefill” conclusion is specific to the shapes tested.

Construct validity. “Coverage” is demonstrated by fourteen module patterns, which is a strong sample but not a proof of universality; genuinely exotic cases (custom `autograd.Function` kernels, CUDA-graph capture, collectives inside a kernel’s forward) are untested.

11 Limitations and Future Work

ONNX Runtime cannot bind `bfloat16` to a custom operator, so `bf16` models use TensorRT. TensorRT’s tensor path required routing `INT8` and in-place-aliased operators to the exact ONNX Runtime plugin. The generated ONNX Runtime registrar exports a fixed symbol, so several compiled models that share a kernel base name can interfere when loaded in one process. Future work: auto-tuning dispatch (folding `@triton.autotune` variants into one plugin with a shape-keyed table); CUDA-graph capture of the launch sequence; channel-last formats for non-pointwise kernels; distributed execution with embedded NCCL handles; and a Blackwell backend (cluster-launch descriptors, `cp.async.bulk`, NVFP4).

Engine coverage beyond ONNX. The recovered contract is engine-agnostic: it describes how a compiled Triton kernel maps onto an engine’s tensors, and only the injection step is engine-specific. TensorRT-LLM is the natural next target, being the standard way to serve LLMs on NVIDIA hardware. Its plugins *are* TensorRT plugins — the same `IPluginV2DynamicExt` and `IPluginCreator` interfaces generated here — so the plugin generator carries over almost unchanged, and TensorRT-LLM ships no automatic plugin generator of its own. The work is consequently in the graph injection rather than the plugin: TensorRT-LLM builds its engine from a checkpoint or from its own module API rather than from an arbitrary ONNX graph, so the rewriting pass has no direct equivalent and a new injection point is required. The reachable scope is non-attention kernels — normalisations, activations, fused elementwise — because TensorRT-LLM’s attention plugin owns the paged KV cache, the grouped-query mapping and in-flight batching, a contract a user kernel cannot re-enter without reimplementing it.

12 Conclusion

A compiled Triton kernel is a self-contained artifact, but the interface between it and a C++ engine is implicit. We showed that the interface can be recovered automatically—the PTX entry signature, the symbolic launch grid, and the argument taxonomy—and re-established as native TensorRT and ONNX Runtime operators injected into the graph. The composition is exact on the research kernels and fourteen module patterns, it replaces 1,137 lines of hand-written C++ per kernel with nine lines of Python, and it pays off: keeping the kernel while letting the engine fold the rest is $1.99\times$ faster than the folded-only formulation and $1.08\times$ faster than the unfused kernel path.

13 Reproducibility Statement

KernelLens is released as a Python package on PyPI (`pip install kernel-lens`) and on GitHub (https://github.com/Just1truc/kernel_lens). The evaluation scripts are in `tests/`: `test_conformance.py` (the fourteen module patterns), `test_research_kernels.py` (parity), `benchmark_folding_vs_triton.py` (the folding comparison), `benchmark_research_kernels.py` and `benchmark_e2e_llama.py` (latency and end-to-end), `test_fp8_gemm.py` and `test_int4_dequant_gemm.py` (quantized), and `measure_codegen.py` (the code-volume measurement). Raw measurements are committed as JSON under `docs/assets/data/`. The complete source tree, the evaluation scripts and the raw measurement JSON are provided as supplementary material (`kernel_lens_supplementary.zip`). All results were produced on the single RTX 5060 Ti configuration described in §8.

References

- [1] P. Tillet, H. Kung, and D. Cox, “Triton: an intermediate language and compiler for tiled neural network computations,” in *ARRAY*, 2019.
- [2] AI@Meta, “The Llama 3 Herd of Models,” *arXiv:2407.21783*, 2024.
- [3] LinkedIn Corp., “Liger-Kernel: Efficient Triton Kernels for LLM Training,” 2024.
- [4] Qwen Team, “Qwen2.5 Technical Report,” *arXiv:2412.15115*, 2024.
- [5] T. Dao et al., “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness,” in *NeurIPS*, 2022.
- [6] T. Dao, “FlashAttention-2,” in *ICLR*, 2024.

- [7] A. Paszke et al., “PyTorch: An imperative style, high-performance deep learning library,” in *NeurIPS*, 2019.
- [8] J. Ansel et al., “PyTorch 2.0: TorchDynamo and TorchInductor,” in *ASPLOS*, 2024.
- [9] PyTorch Engineering Team, “PyTorch AOTInductor,” 2023.
- [10] T. Chen et al., “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning,” in *OSDI*, 2018.
- [11] C. Lattner et al., “MLIR: Scaling Compiler Infrastructure for Domain Specific Architectures,” in *CGO*, 2021.
- [12] J. Ragan-Kelley et al., “Halide: A Language and Compiler for Optimizing Parallel Image Processing Pipelines,” in *PLDI*, 2013.
- [13] W. Kwon et al., “Efficient Memory Management for LLM Serving with PagedAttention,” in *SOSP*, 2023.
- [14] NVIDIA Corporation, “TensorRT-LLM,” <https://developer.nvidia.com/tensorrt>, 2023.
- [15] Microsoft Corporation, “ONNX Runtime,” <https://onnxruntime.ai>, 2021.
- [16] NVIDIA Corporation, “NVIDIA TensorRT,” <https://developer.nvidia.com/tensorrt>, 2023.