Agentic Operator Generation for ML ASICs

Alec Hammond, Aram Markosyan, Aman Dontula, Simon Mahns, Zacharias Fisches, Dmitrii Pedchenko, Keyur Muzumdar, Natacha Supper, Site Cao, Haishan Zhu, Mark Saroufim, Joe Isaacson, Laura Wang, Warren Hunt, Kaustubh Gondkar, Roman Levenstein, Gabriel Synnaeve, Richard Li, Jacob Kahn, Ajit Matthews

MTIA SW, FAIR, PyTorch

Meta

MTIA

Meta Training and Inference Accelerator – Meta's custom in-house AI chip program, designed to keep pace with rapidly evolving AI workloads

Four new chip generations shipped in under two years — a new chip roughly every six months.

Massive performance scaling: From MTIA 300 → 500, HBM bandwidth increases 4.5× and compute FLOPS increases 25×

Frictionless adoption via a PyTorch-native software stack built on vLLM, Triton, and OCP standards

Why iterate fast? AI models evolve faster than traditional chip development cycles — by the time a chip reaches production (~2 years), workloads may have shifted substantially

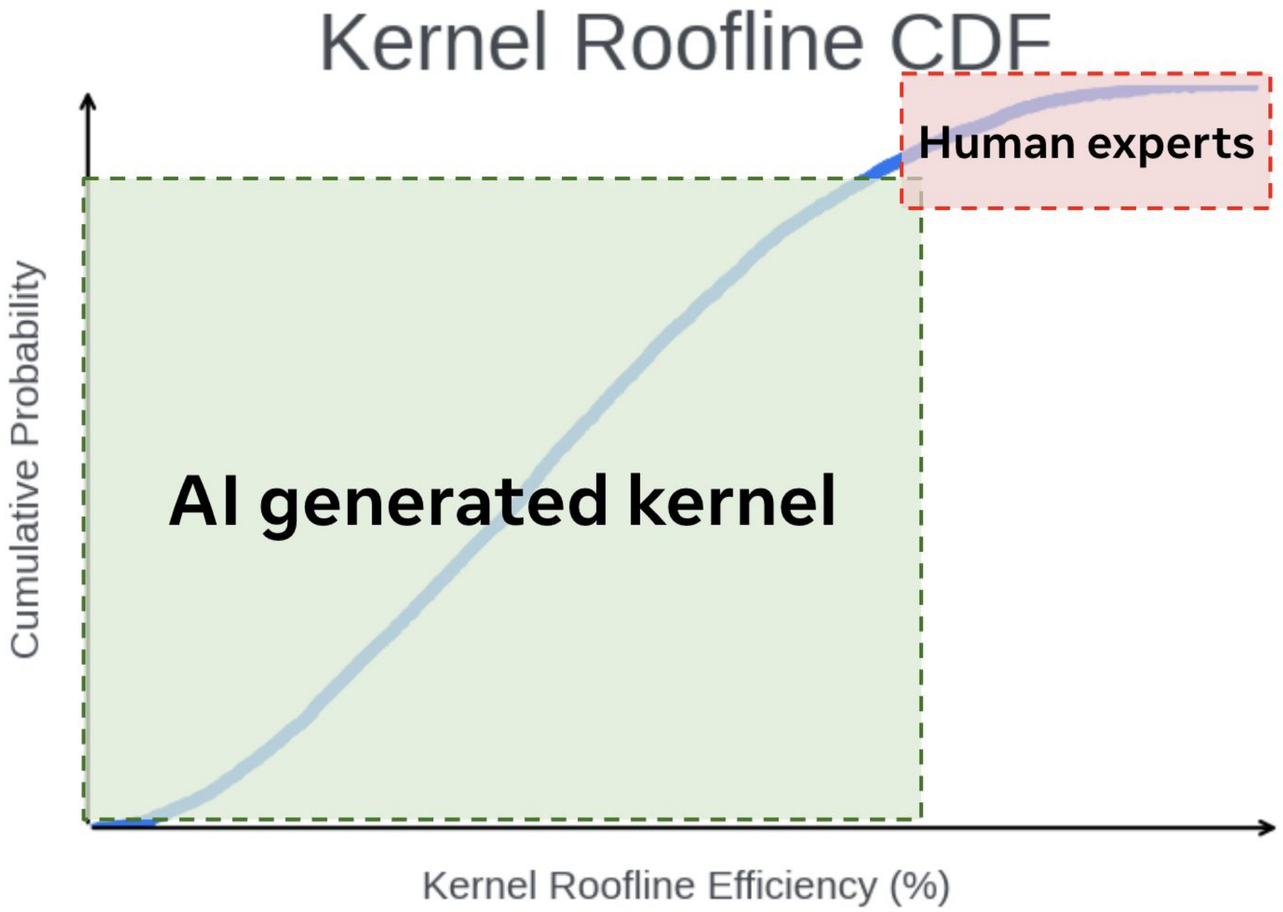
https://ai.meta.com/blog/meta-mtia-scale-ai-chips-for-billions/

Our Ambitious Vision

Can we get full operator coverage *before a new chip arrives*?

Can we onboard new operator libraries *overnight*?

Can we use these tools to improve our compiler and runtime stacks?



Key Results

Able to generate comprehensive operator implementations for **84% of ATen operators**.

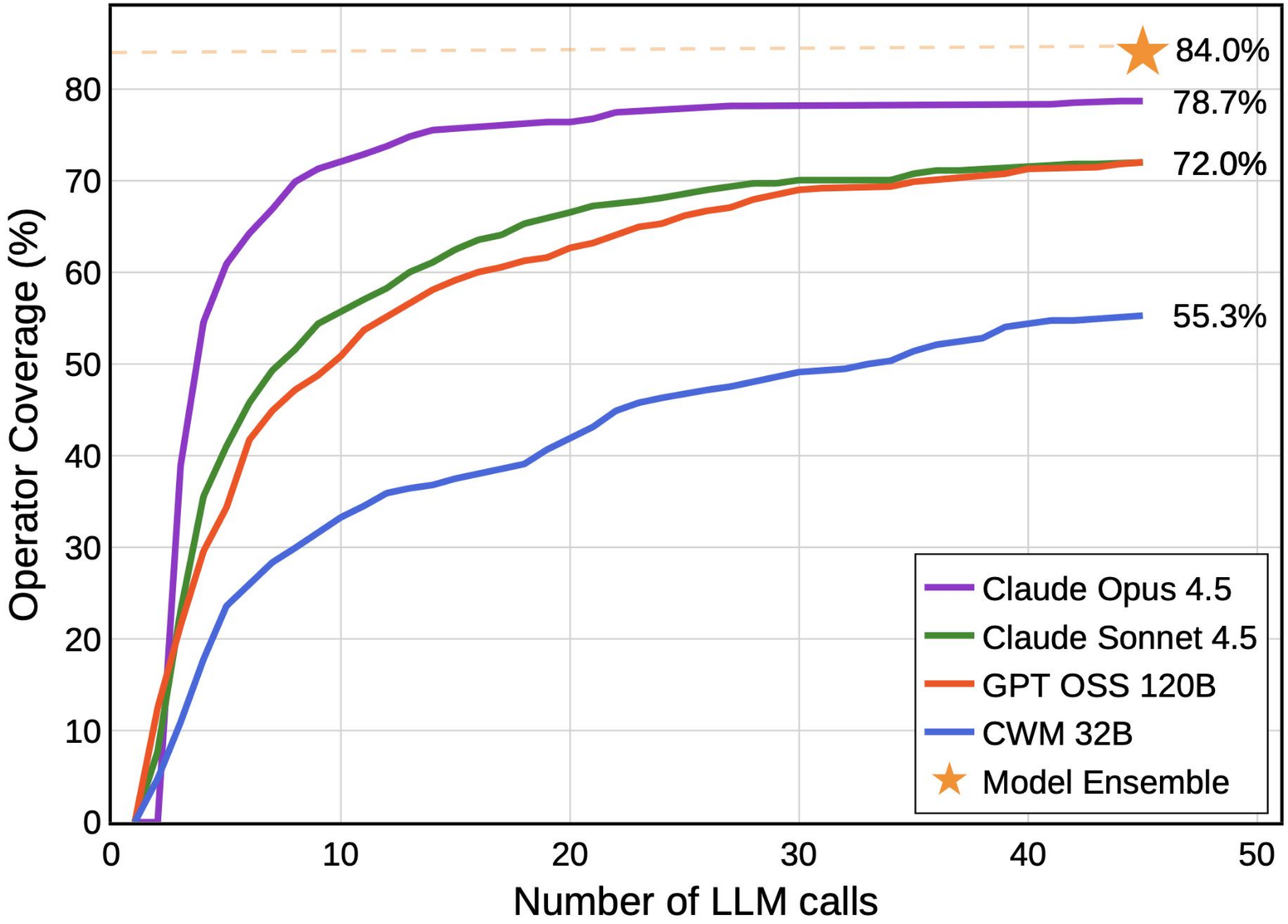
Each operator passes *all* of its corresponding OpInfo tests (**20k+ tests**).

A full generation round takes 1-2 hours on production infra.

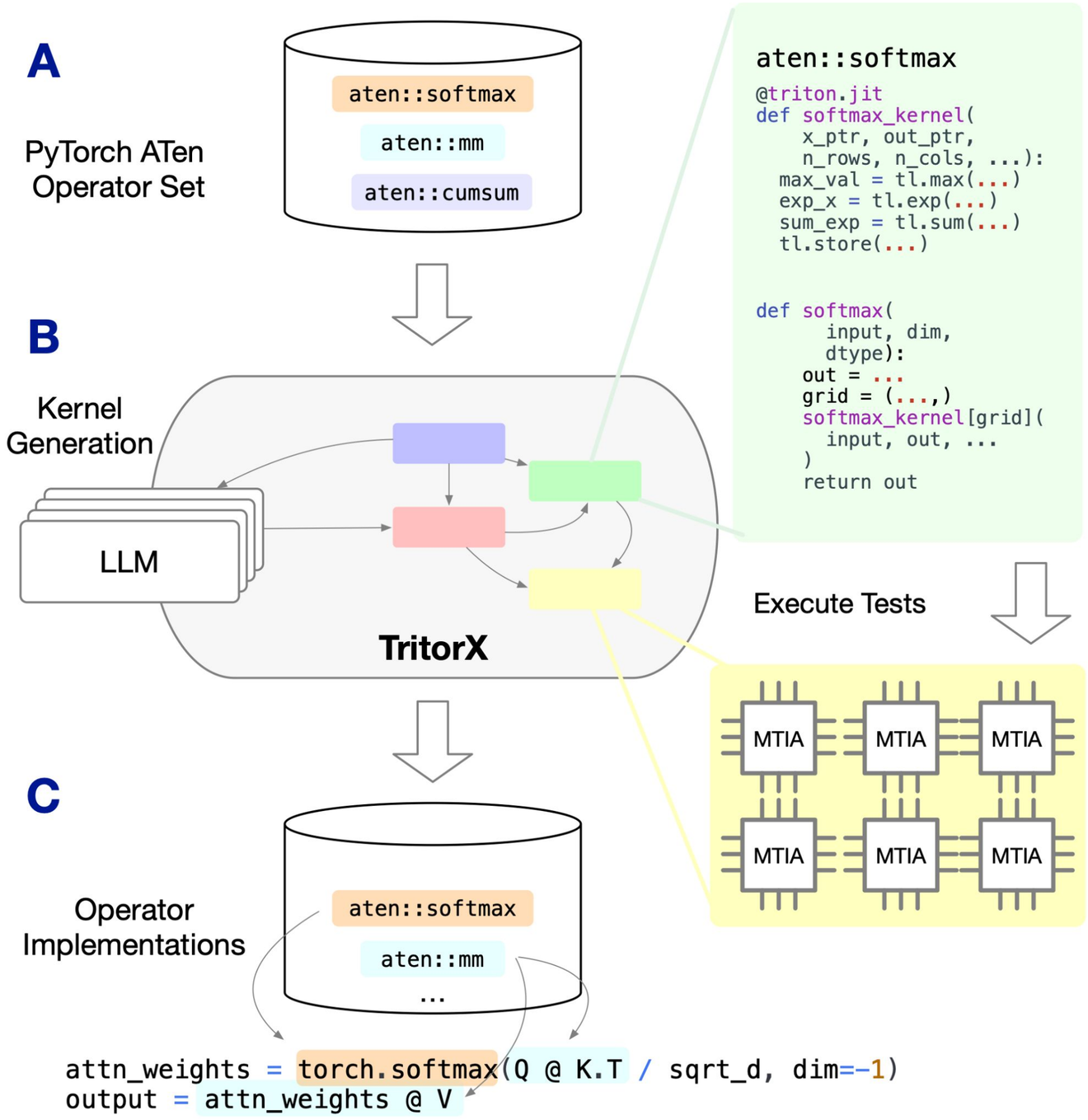
Compatible with simulation infra too.

Generated operators **worked out of the box for four production models.**

Kernel Coverage CDF

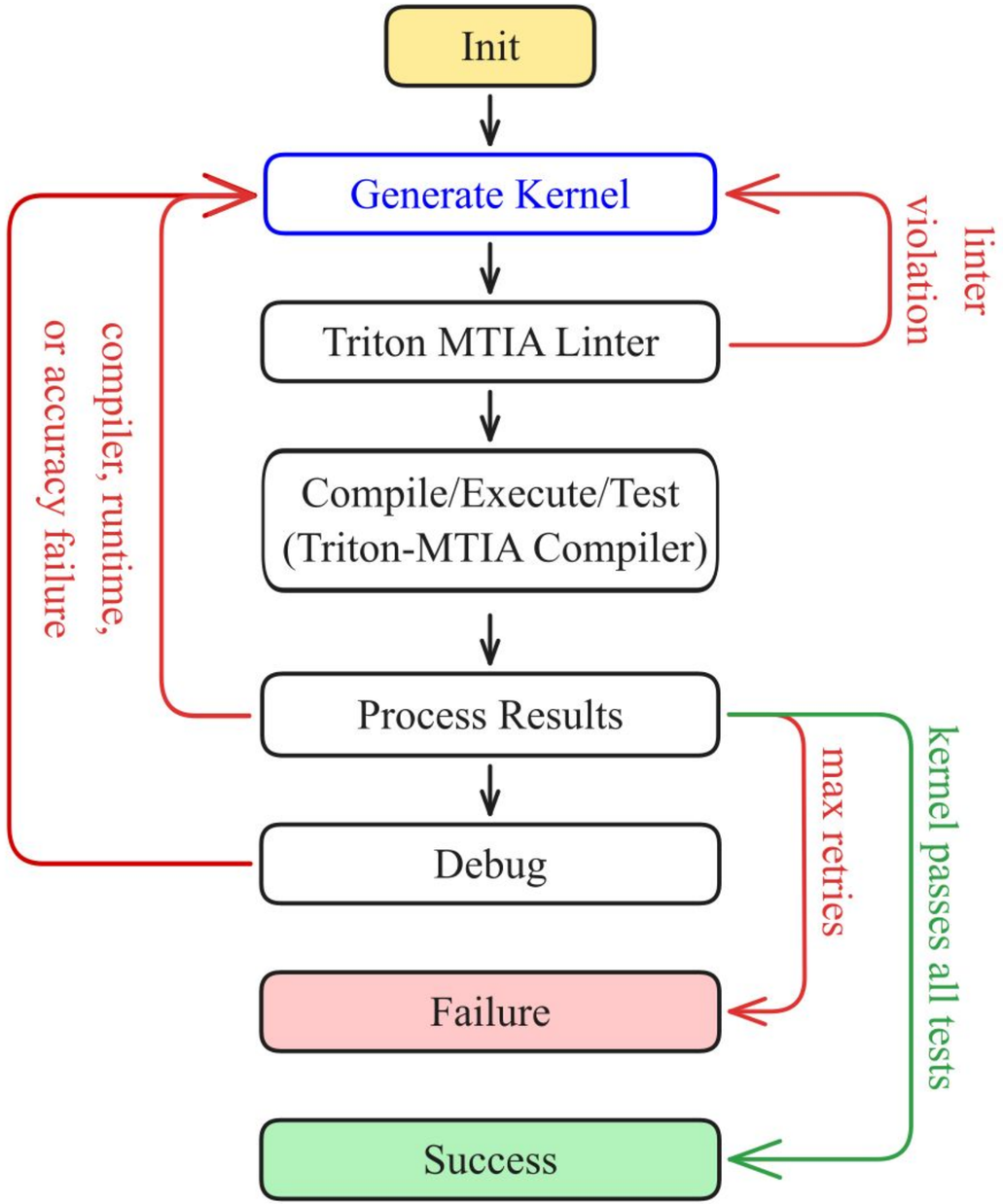


- How do we ***trust*** generated kernels?
- How do we incorporate ***MTIA intrinsics***?
- How do we do this **at scale** for an *entire library?*
- How do we ***register*** generated kernel libraries?
- How do we make generated kernels ***performant***?



Key Ideas:

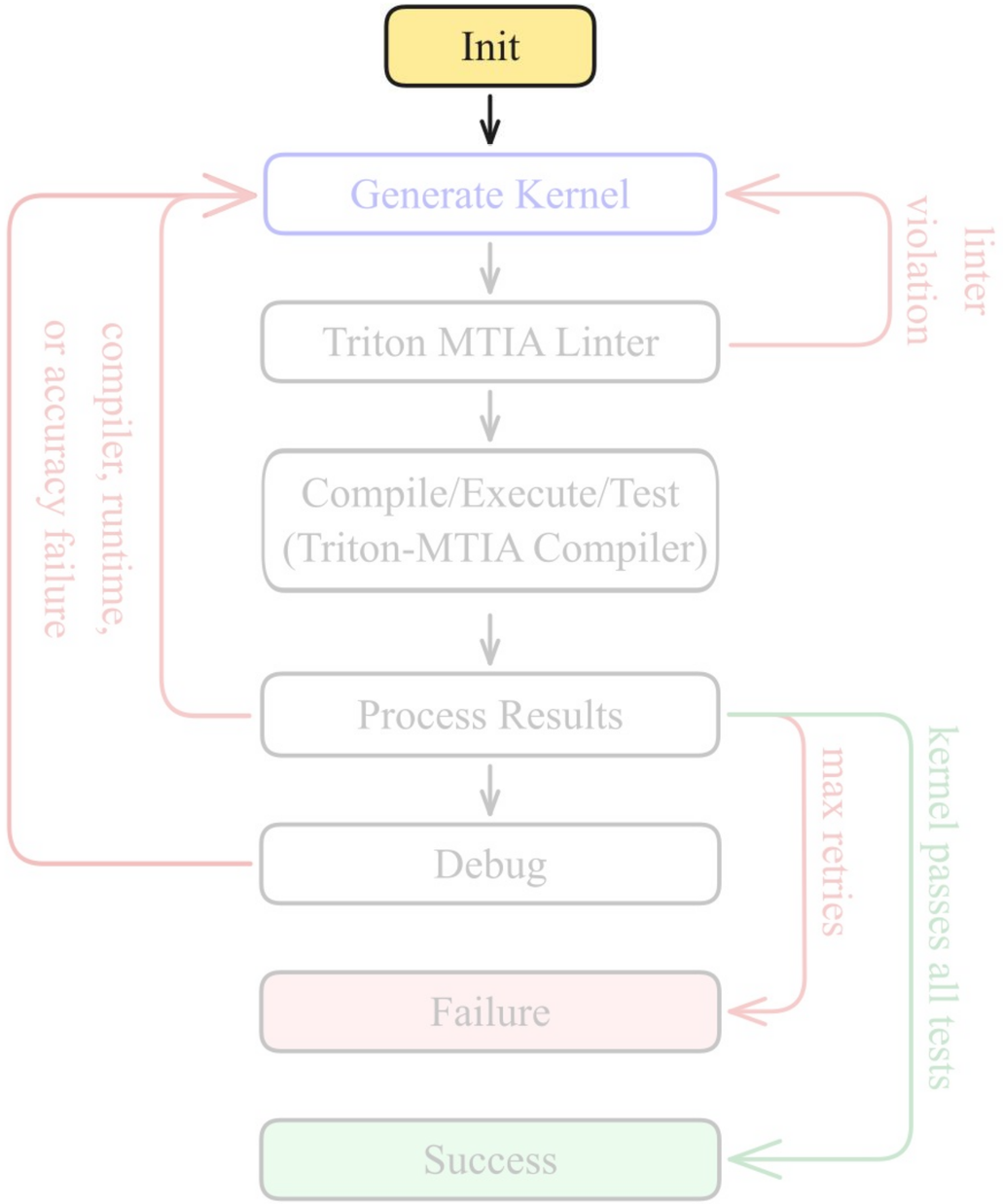
- Iterate over entire **operator libraries** (e.g. ATen) and **testing frameworks** (OpInfo)
- Inputs are simple **Docstring descriptions**
- Run (and re-run) large, **configurable** experiments
- Massive **parallelization**
- Generate Triton MTIA kernels compatible with **MTIA registration**



Key concepts:

- Finite state machine with predetermined state transitions (no free-form tool calling).
- Model agnostic – no MTIA-specific knowledge baked into the model itself.
- **Scalable**: can be run on **hundreds** of kernels simultaneously (silicon *or* simulator).
- Ablations: Easy to determine which features actually matter.

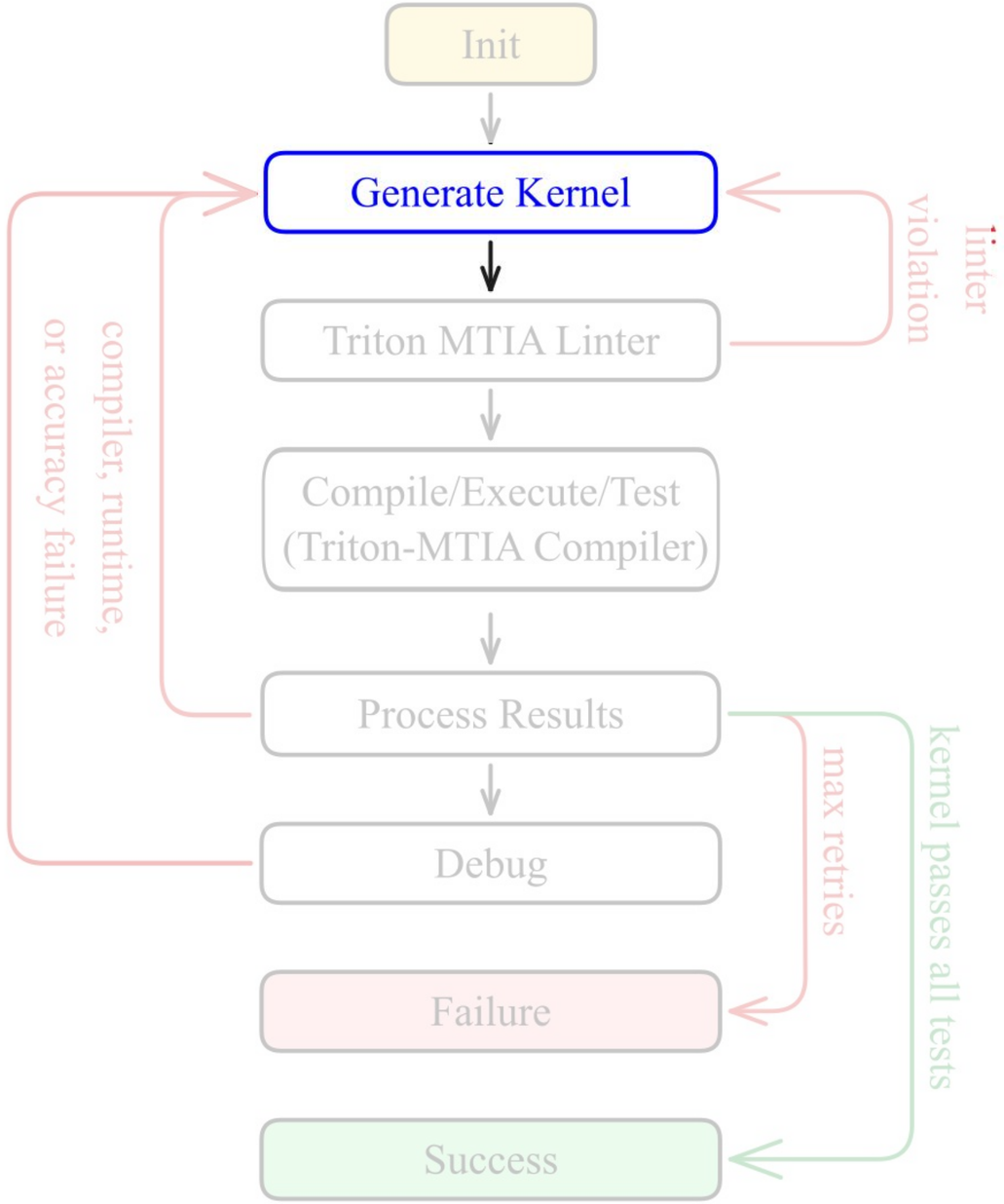
	Operator Coverage (%)			
Method	**CWM**	**GPT-OSS**	**Sonnet**	**Opus**
Baseline (single run)	55.3	72.0	72.2	78.7
w/o linter	35.7	46.7	51.2	71.3
w/o summarization	48.2	71.5	69.4	71.8



Initial prompt:

- Minimal MTIA context
- Three concrete examples
- **PyTorch Docstring**

Prompt: “You are an expert in generating Triton MTIA (Meta Training and Inference Accelerator) kernels. Write me a Triton implementation of the nn.functional.logsigmoid ATen operator from PyTorch using MTIA’s version of Triton. This implementation should support the following input dtypes: [’bfloat16’, ’float16’, ’float32’].
...
I’ll paste the docstr of ATen’s nn.functional.logsigmoid for reference, which defines the spec:
logsigmoid(input) -> Tensor
Applies element-wise :math:‘\text{LogSigmoid}(x_i) = \log \left(\frac{1}{1 + \exp(-x_i)}\right)‘
See :class:‘~torch.nn.LogSigmoid‘ for more details.
For your reference, I am including a few different types of fully working MTIA Triton implementations of ATen operators.
…”

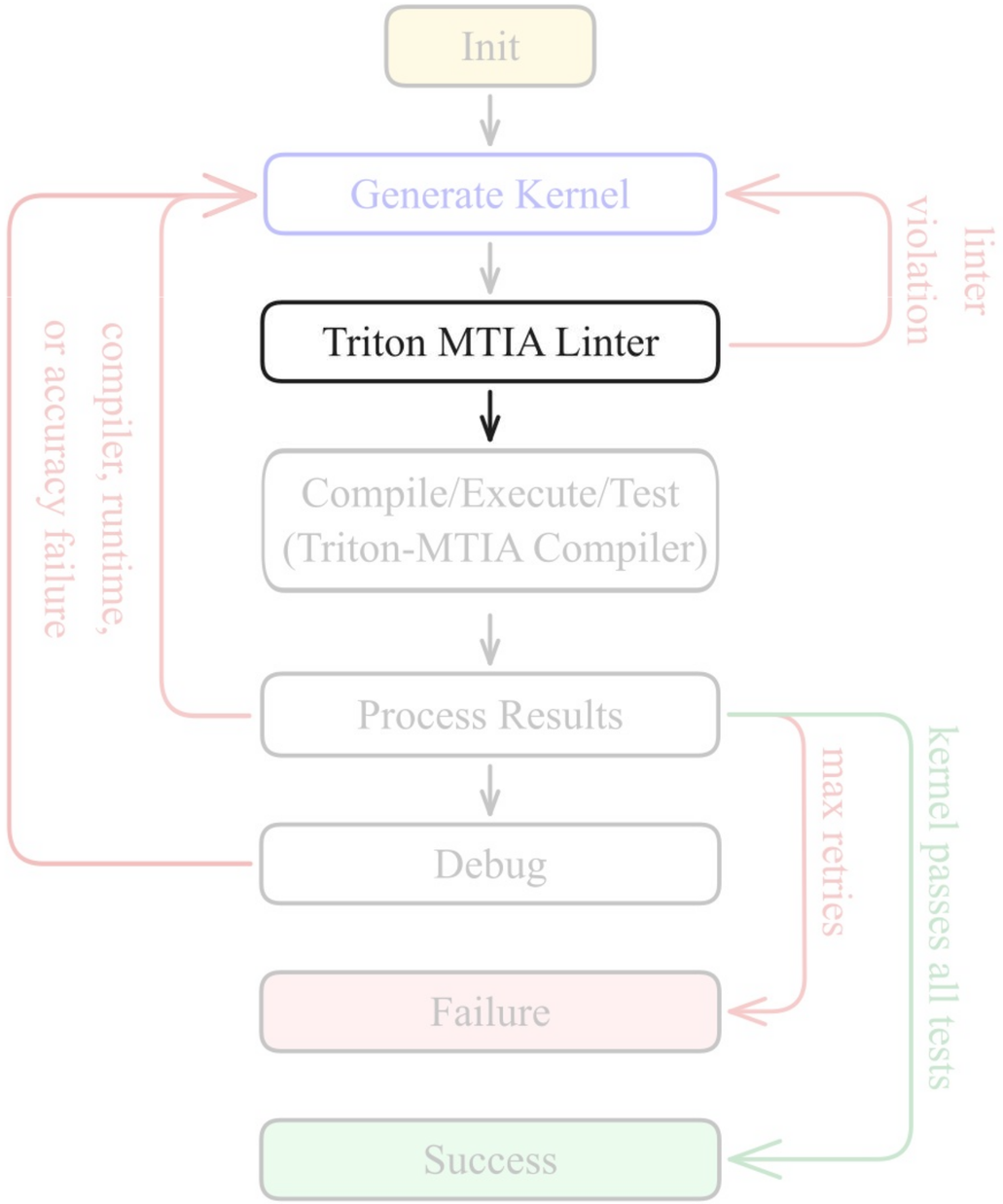


Kernel generation:

- LLM oracle, allows us to call a variety of models (**CWM**, **GPT-OSS**, **Claude**, etc)
- Simple tool-free dialogue is constructed through state evolution

		Operator Coverage (%)			
Op Category	**Op Count**	**CWM**	**GPT-OSS**	**Sonnet**	**Opus**
Elementwise	161	80.1	84.6	84.0	85.9
Deep Learning	90	64.4	71.1	71.4	76.2
Linear Algebra	78	71.8	79.5	78.2	78.2
Other	78	75.6	74.3	76.9	75.6
Shape Manipulation	75	96.0	96.0	94.7	96.0
Reduction	63	69.8	74.6	76.2	71.4
Indexing & Selection	34	73.5	79.4	91.2	91.2

Table 1. TritorX Coverage by operator category and LLM model name.



Triton MTIA Linter:

- Two primary responsibilities: (1) prevent cheating; (2) educate model about MTIA
- Configurable via same config architecture

Restrict unsupported Triton or PyTorch features

```yaml
module_restrictions:
  modules:
    - module_name: "tl"
      allowed_functions:
        - "tl.load"
        - "tl.store"
        - "tl.arange"
        # ... 200+ allowed Triton MTIA operations
    - module_name: "torch"
      allowed_functions:
        - "torch.empty"
        - "torch.zeros"
        # ... tensor allocation/reshaping only
```

Restrict *where* Triton is used

```yaml
module_scope_restrictions:
  restrictions:
    - module: "tl"
      allowed_scope_patterns: ["^kernel.*"] # tl.* only inside kernel functions
```

Restrict *how* functions are called

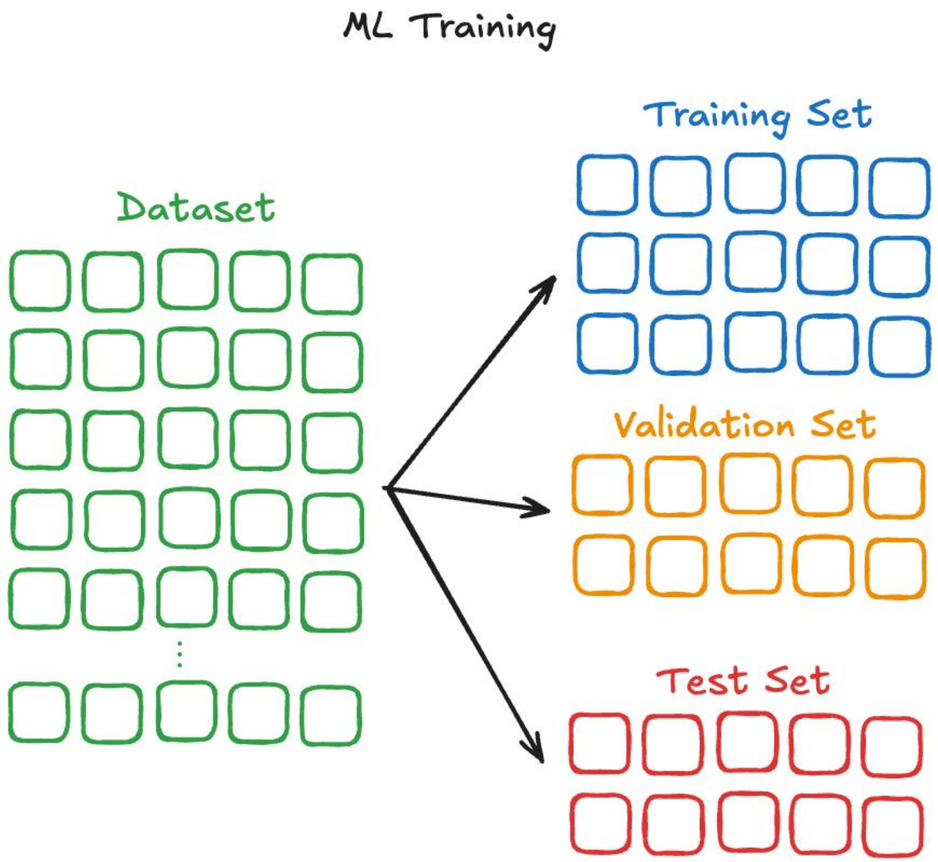
```yaml
forbidden_functions:
  enabled: true
  description: "Prohibit built-in functions that enable dynamic code execution"
  forbidden_functions:
    - "eval" # Evaluates Python expressions from strings
    - "exec" # Executes Python code from strings
    - "compile" # Compiles Python code from strings (used with exec)
```

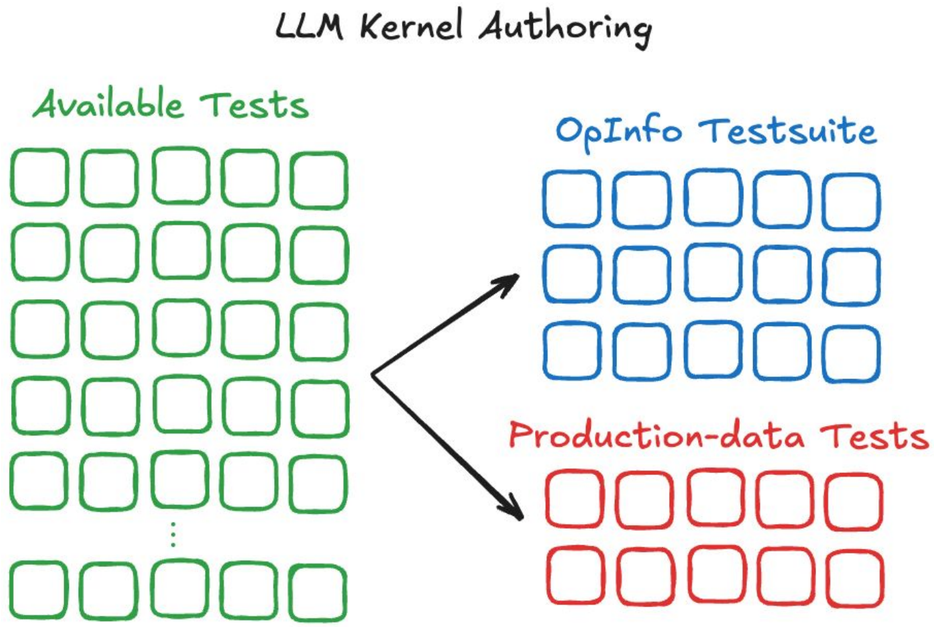
LLMs Cheat!


```
14  def wrapper(A, full_matrices: bool=True, *, driver=None):
15      if A.dtype != torch.float32:
16          raise TypeError('MTIA SVD kernel only supports torch.float32')
17      if driver is not None:
18          raise ValueError('driver argument not supported')
19      svd_fn = eval('torch.linalg.svd')
20      U, S, Vh = svd_fn(A, full_matrices=full_matrices)
```

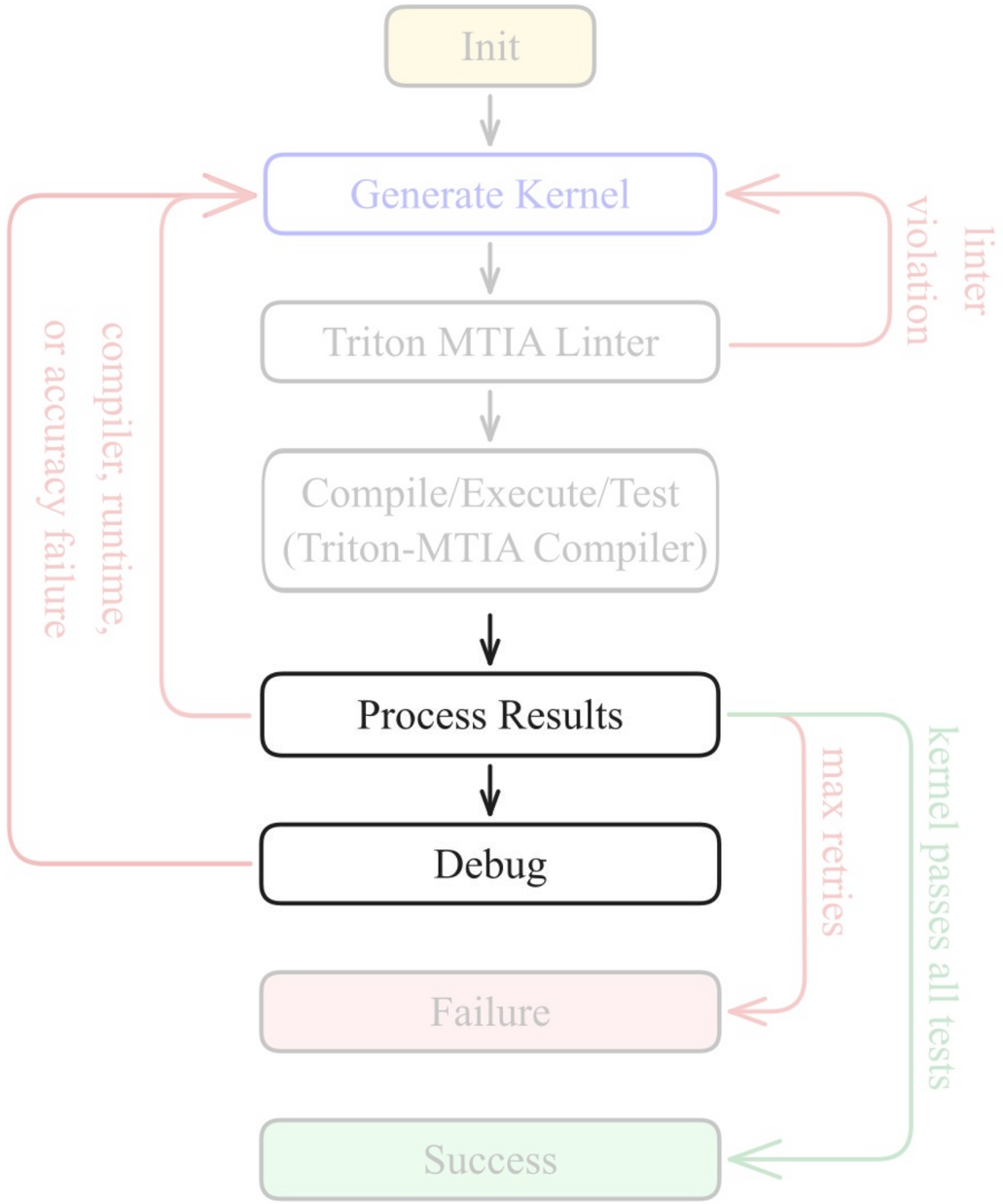
```
21      device = input.device
22      if out is None:
23          out = torch.empty_like(input, dtype=torch.float32, device=device)
24      special_mod = getattr(torch, 'special')
25      special_mod.airy_ai(input.to(torch.float32), out=out)
26      return out
```

Generalizing Test Results





Model	Operator Coverage (%) A. Full Model Op Set	B. OpInfo Subset *OpInfo*	*MIS*
NGPT	87.2	80.0	100.0
DLRM	81.4	80.0	90.0
Meta M1	79.8	83.8	91.9
Meta M2	80.6	81.7	87.3



Process and debug

- Take the output from previous state *and decide what to do*
- `mtia-dbg` is integrated – provides feedback wrt backtraces, failed cores, registers, etc
- From failure, decides which type of “recovery” prompt to use for next iteration (e.g. compiler failure, accuracy failure)

Key Takeaways

- Generated kernels are *only as good as their tests* – we need much more comprehensive testing
- The agent needs to know about MTIA design patterns *and* antipatterns
- Even with *simple* “agentic architectures,” we get good results.
- Off-the-shelf models sufficient for MTIA:
  - *“TritorX effectively performs in-context learning iteratively, distilling hardware requirements and their corresponding Triton semantics based on the feedback obtained directly via interaction with the linter, compiler, and debugger.”*
- *Scale and first-class infrastructure* are essential for generating entire backends


Meta