An End-to-End Workflow for Data-Driven GPU Optimization with LLVM

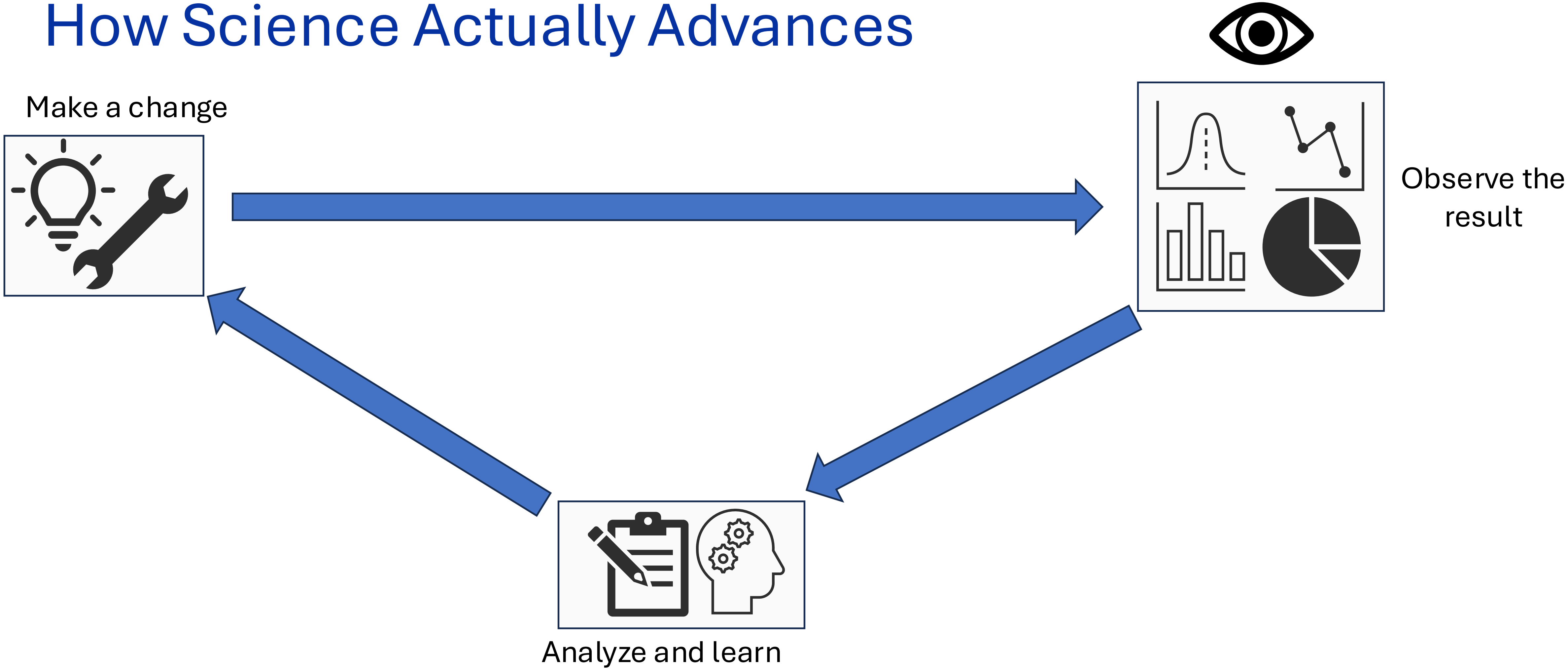
LLVM @ CGO26

Jan. 31st 2026

Konstantinos Parasyris
Center for Advanced Scientific Computing (CASC), LLNL

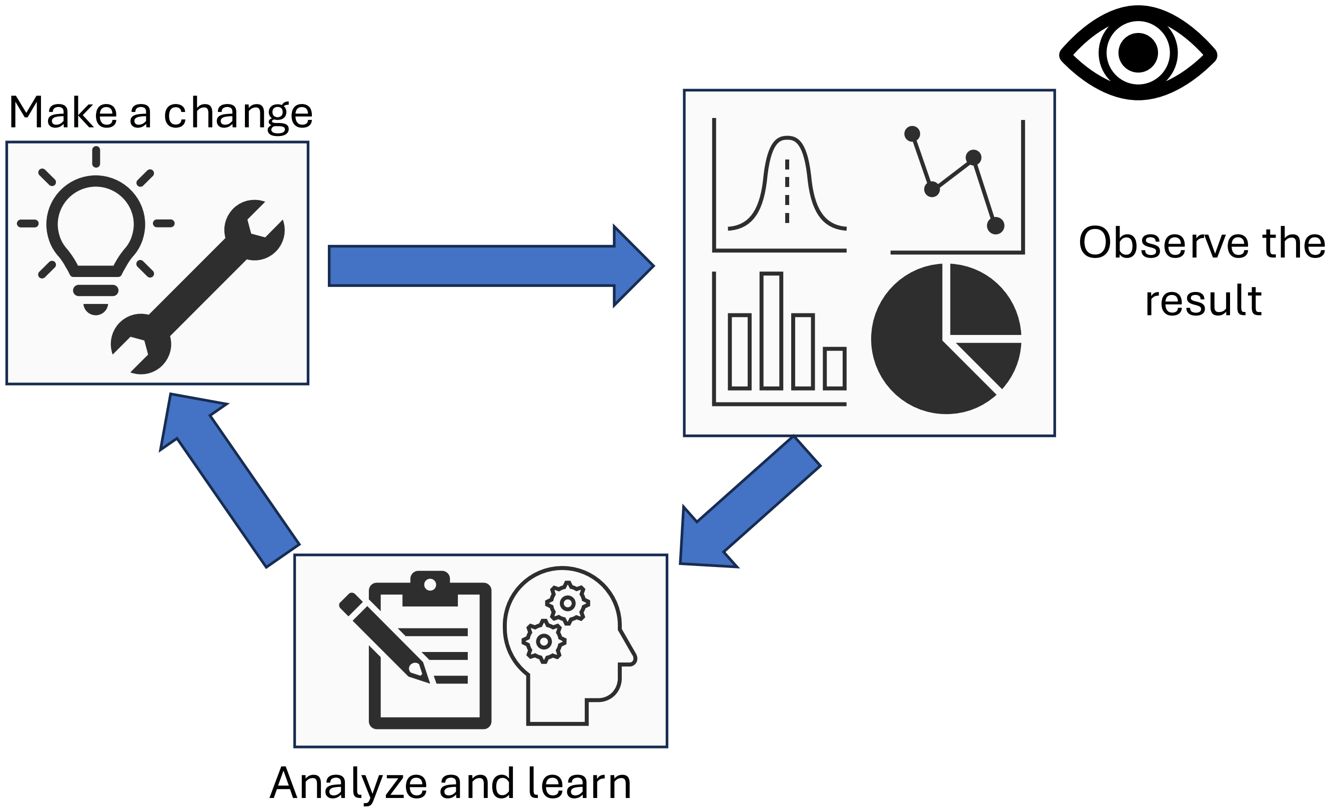
Prepared by LLNL under Contract DE-AC52-07NA27344.

How Science Actually Advances



Progress comes from *fast, controlled iteration* and *trustworthy observation*.

How Science Actually Advances

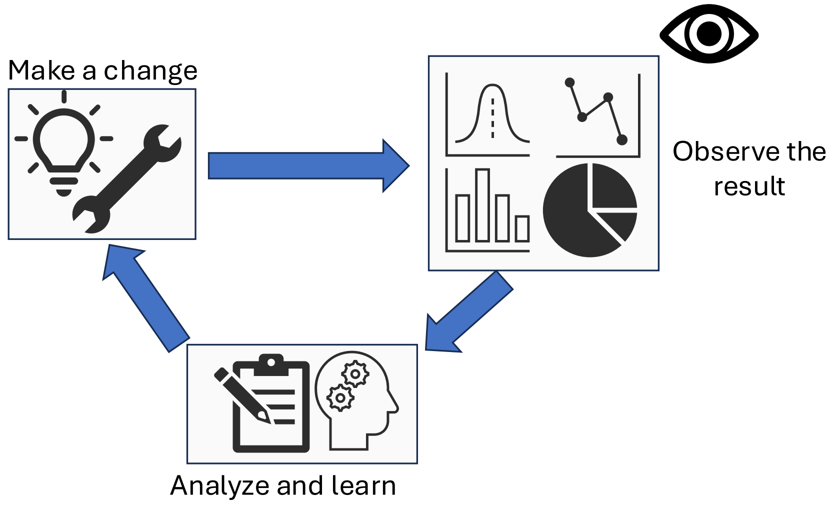


Reducing the complexity and time of this cycle is a strong indicator of continuous scientific progress

How Science Actually Advances

Challenges

- Data quality and representativeness
  - Noisy, biased data, or detached from real workloads, lead to non generalizable conclusions.
- Iteration cost
  - Costly experiment taking hours because of:
    - Compilation cost
    - Fragile Scripts
  - Limit our exploration capabilities

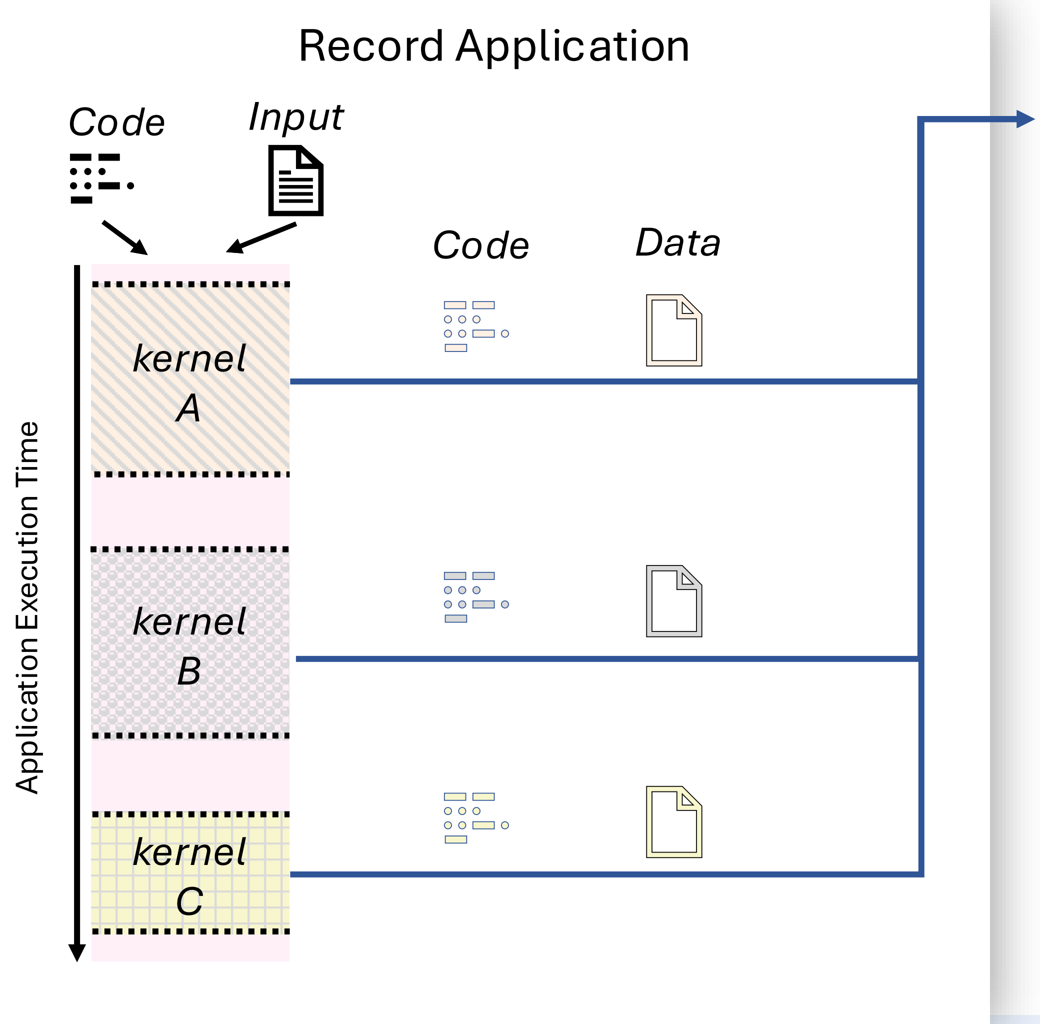


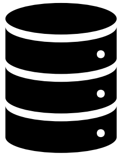
LLVM’s Strength — and Its Accessibility Problem

Mneme is a tool providing a core infrastructure to enable data driven research on GPU LLVM research

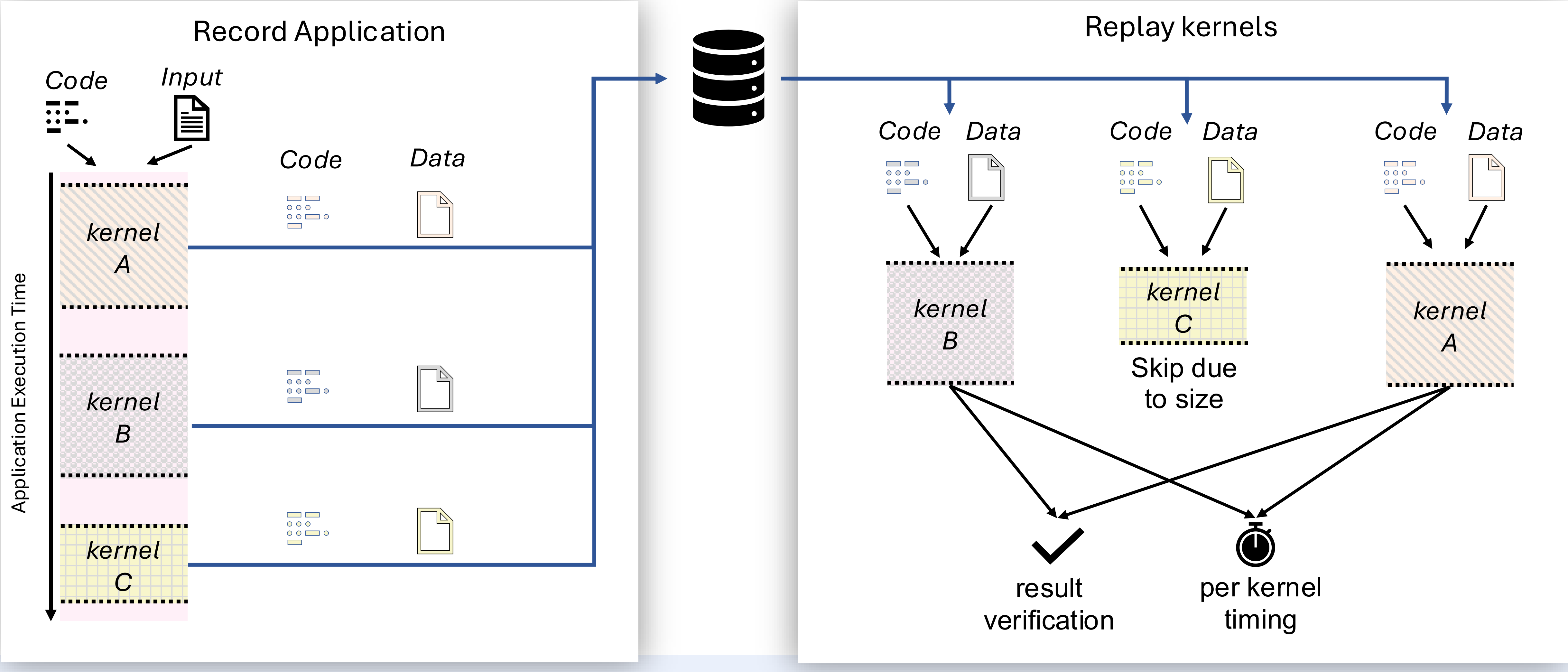
Mneme

What is Record Replay?

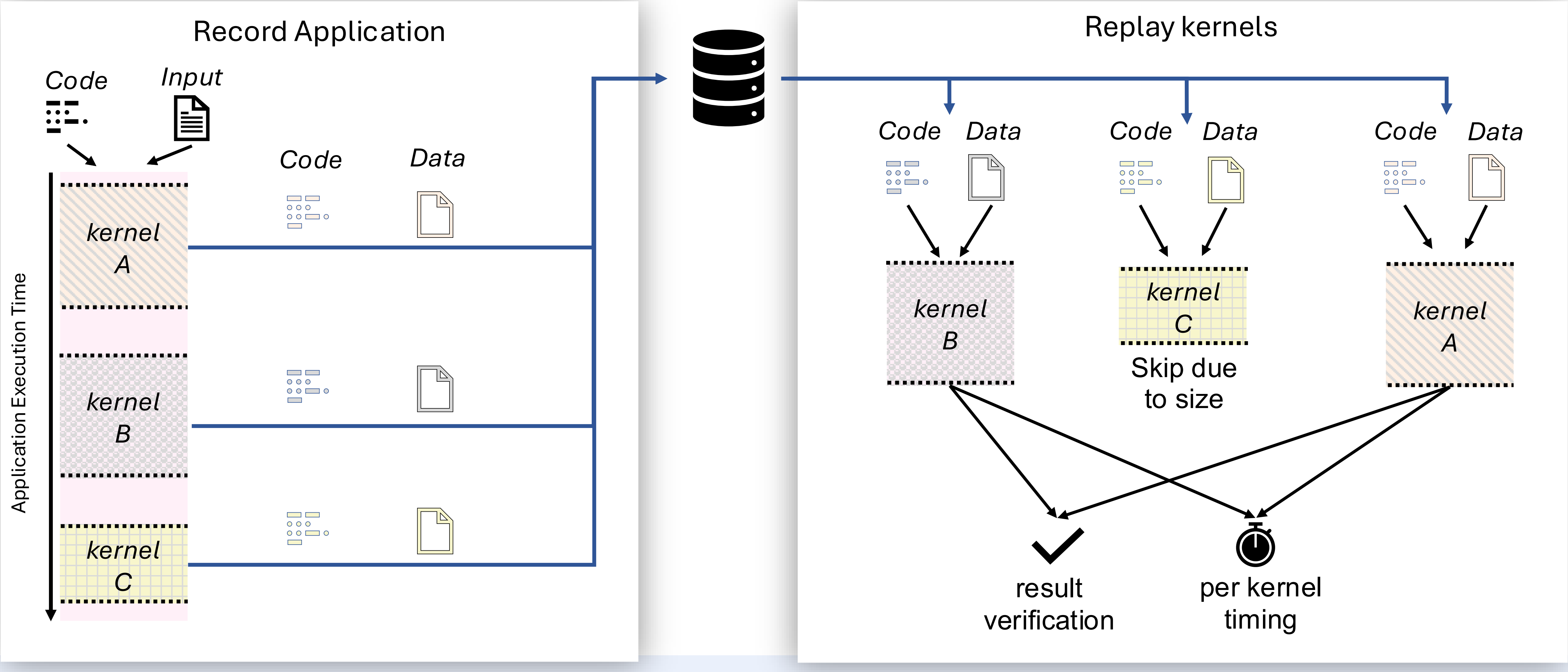




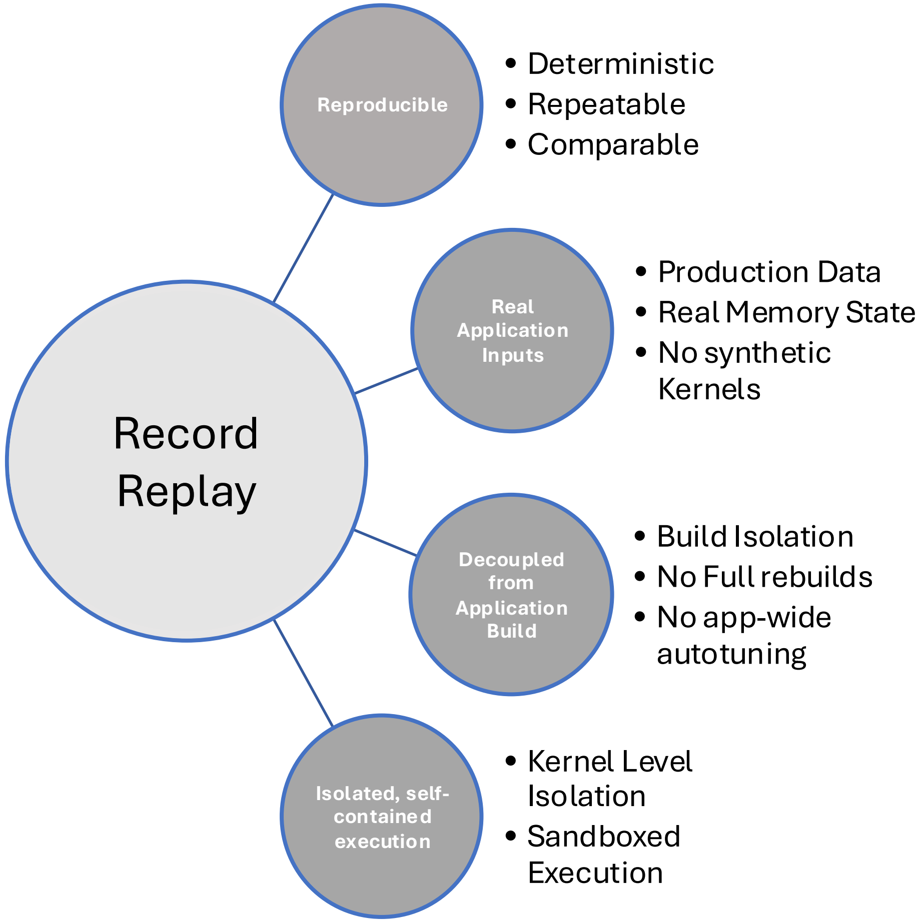
What is Record Replay?



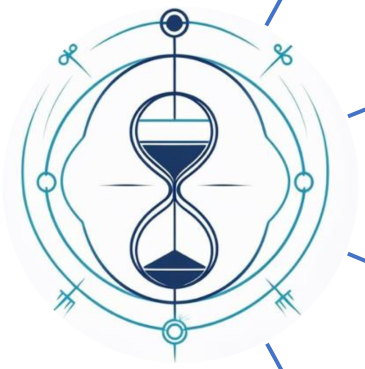
What is Record Replay?



Record Replay can provide



Mneme: Record–Replay for Scalable Optimization and analysis



Reproducible
- Deterministic
- Repeatable
- Comparable

Real Application Inputs
- Production Data
- Real Memory State
- No synthetic Kernels

Decoupled from Application Build
- Build Isolation
- No Full rebuilds
- No app-wide autotuning

Isolated, self-contained execution
- Kernel Level Isolation
- Sandboxed Execution

- Implements record–replay for GPU
- Decouples tuning from application dependencies
- Integrates with LLVM
  - Python accessors to Functions, Blocks, Instructions etc.
    - Similar to numba/llvmlite
  - Proteus is the execution engine and applies optimizations
- Exposes replayed kernels to Python ecosystem
- Enables autotuning, analysis, and experimentation

High Level Of Execution Phases

Build "**Mneme**"

- export LLVM_INSTALL_PATH=${ROCM_PATH}
- `pip install https://github.com/Olympus-HPC/Mneme`

Create a "**recordable executable**"

- Apply instrumentation pass to the code

Record the execution of an application

- Check the generated artifacts

Replay a single Kernel

- Verify outputs
- Create your own autotuner

Create a “**recordable executable**”

1 Include Mneme on build process

```
> cat CMakeLists.txt
...
find_package(HIP REQUIRED)
find_package(mneme REQUIRED)
add_executable(tutorial.exe tutorial.hip)
add_mneme(tutorial.exe)
...
```

2 Configure & Build

```
> cmake -B BUILD -S SRC_PRJ \
        -DCMAKE_C_COMPILER=$(mneme config cc) \
        -DCMAKE_CXX_COMPILER=$(mneme config cxx) \
        -DCMAKE_PREFIX_PATH=$(mneme config cmakedir)
> cmake --build BUILD/
```

The executable carries its own compiler IR

What:
- Embedded LLVM IR

Why it matters:
- Enables **post-mortem analysis and recompilation**
- No need to recover IR from build system or source tree

All kernel executions become observable and interceptable

What:
- Kernel launches go through **Proteus API**
- Vendor launch APIs are not invoked directly

Why it matters:
- Intercept kernel launches, arguments, launch configurations etc.
  - These can be “tunable parameters” at replay time

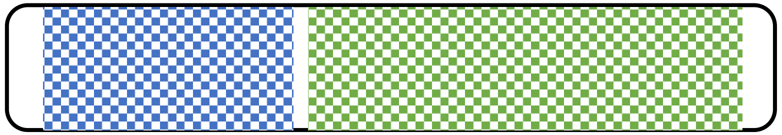
Record the execution of an application

1 Wrap “recordable executable” execution with mneme

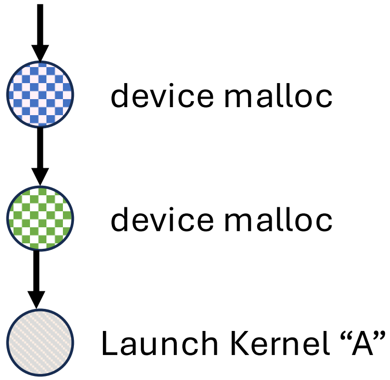
```
> mneme record -rdb record-db-dir/ -vass X \
              -- <recordable-executable> \
                   <arguments>
```

Address Space
Managed by Mneme
(|HighAdress – LowAdress| = “vass”)

Low Address High Address



Trace of host-device events



1) Store Mneme Memory to persistent storage (prologue)

2) Query proteus for LLVM IR of the kernel and store into storage

3) Launch Kernel (synchronously)

4) Store Mneme Memory to persistent storage (epilogue)

Record the execution of an application

1 Wrap “recordable executable” execution with mneme

```
> mneme record -rdb record-db-dir/ -vass X \
                -- <recordable-executable> \
                   <arguments>
```

2 Recording artifacts are stored under “record-db-dir”

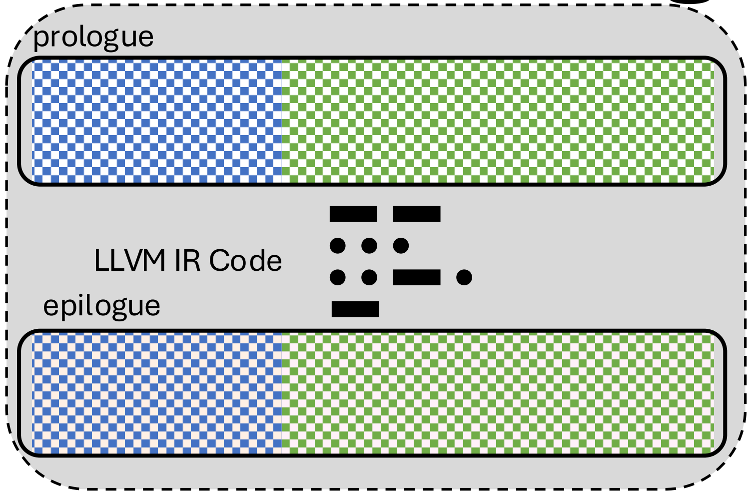
```
> tree record-db-dir/
── <static-hash>.json
── DeviceState.epilogue.<static-hash>.<dynamic-hash>.mneme
── DeviceState.prologue.<static-hash>.<dynamic-hash>.mneme
── RecordedIR_<static-hash>.bc
```

Replay a single Kernel

1 Replay a single kernel invocation

```
> mneme replay \
    -rdb record-example-dir/<static-hash>.json \
    -rid <dynamic-hash> "default<O3>"
```

record-example-dir/



Trace of host-device events

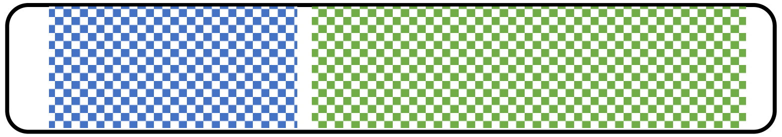
- Instantiate Device Memory Space
- Initialize Memory
- Compile and execute code through Proteus
- Compare device memory with epilogue

Automated verification

Address Space
Managed by Mneme
(|HighAdress – LowAdress| = "vass")

Low Address

High Address



Replay a single Kernel

1 Replay a single kernel invocation

```
> mneme replay \
   -rdb record-example-dir/<static-hash>.json \
   -rid <dynamic-hash> "default<O3>"
```

2 Execution emits a key-value dictionary describing various metrics

```
"Replay-config": {
  "grid": {
    "x": 40000,
    "y": 1,
    "z": 1
  },
  "block": {
    "x": 256,
    "y": 1,
    "z": 1
  },
  "shared_mem": 0,
  "specialize": false,
  "set_launch_bounds": false,
  "max_threads": null,
  "min_blocks_per_sm": 0,
  "specialize_dims": false,
  "passes": "default<O3>",
  "codegen_opt": 3,
  "codegen_method": "serial",
  "prune": true,
  "internalize": true
},
```

```
"Result": {
  "preprocess_ir_time": 9.2298723757267e-06,
  "opt_time": 0.006206092890352011,
  "codegen_time": 0.01226967596448958,
  "obj_size": 4792,
  "exec_time": [
    84040,
    82561,
    81761,
    83360,
    76520
  ],
  "verified": true,
  "executed": true,
  "failed": false,
  "start_time": "",
  "end_time": "",
  "gpu_id": 0,
  "const_mem_usage": -1,
  "local_mem_usage": 0,
  "reg_usage": 12,
  "error": ""
}
```

Replay a single Kernel

1 Replay a single kernel invocation

```
> mneme replay \
   -rdb record-example-dir/<static-hash>.json \
   -rid <dynamic-hash> "default<O3>"
```

These parameters can be modified

By forming valid configuration ranges of these parameters one can search the space and tune the application in respect to some quantity of interest

2 Execution emits a key-value dictionary describing various metrics

```
"Replay-config": {
  "grid": {
    "x": 40000,
    "y": 1,
    "z": 1
  },
  "block": {
    "x": 256,
    "y": 1,
    "z": 1
  },
  "shared_mem": 0,
  "specialize": false,
  "set_launch_bounds": false,
  "max_threads": null,
  "min_blocks_per_sm": 0,
  "specialize_dims": false,
  "passes": "default<O3>",
  "codegen_opt": 3,
  "codegen_method": "serial",
  "prune": true,
  "internalize": true
},
```

```
"Result": {
  "preprocess_ir_time": 9.2298723757267e-06,
  "opt_time": 0.006206092890352011,
  "codegen_time": 0.01226967596448958,
  "obj_size": 4792,
  "exec_time": [
    84040,
    82561,
    81761,
    83360,
    76520
  ],
  "verified": true,
  "executed": true,
  "failed": false,
  "start_time": "",
  "end_time": "",
  "gpu_id": 0,
  "const_mem_usage": -1,
  "local_mem_usage": 0,
  "reg_usage": 12,
  "error": ""
}
```

Replay a single Kernel

1 Replay a single kernel invocation

```
> mneme replay \
    -rdb record-example-dir/<static-hash>.json \
    -rid <dynamic-hash> "default<O3>"
```

These parameters can be modified

By forming valid configuration ranges of these parameters one can search the space and tune the application in respect to some quantity of interest

Several quantity of interest are supported

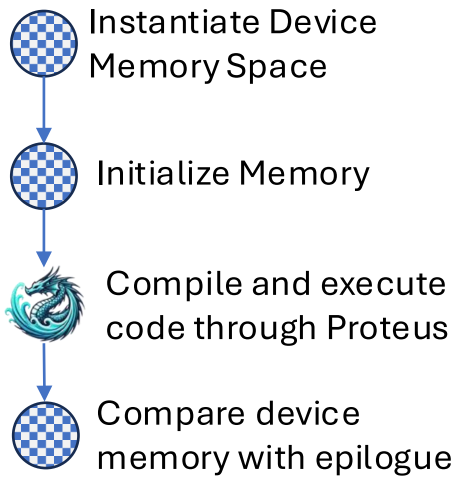
- Execution Time (exec_time)
- Register Usage (reg_usage)
- Binary Size (obj_size)

2 Execution emits a key-value dictionary describing various metrics

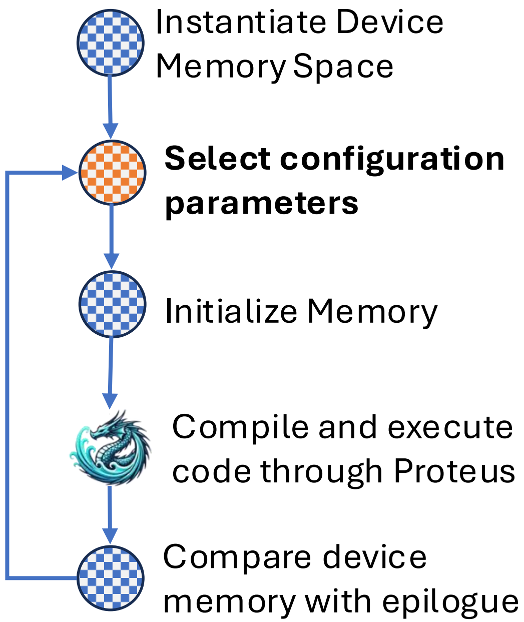
```
"Replay-config": {
  "grid": {
    "x": 40000,
    "y": 1,
    "z": 1
  },
  "block": {
    "x": 256,
    "y": 1,
    "z": 1
  },
  "shared_mem": 0,
  "specialize": false,
  "set_launch_bounds": false,
  "max_threads": null,
  "min_blocks_per_sm": 0,
  "specialize_dims": false,
  "passes": "default<O3>",
  "codegen_opt": 3,
  "codegen_method": "serial",
  "prune": true,
  "internalize": true
},
```

```
"Result": {
  "preprocess_ir_time": 9.2298723757267e-06,
  "opt_time": 0.006206092890352011,
  "codegen_time": 0.01226967596448958,
  "obj_size": 4792,
  "exec_time": [
    84040,
    82561,
    81761,
    83360,
    76520
  ],
  "verified": true,
  "executed": true,
  "failed": false,
  "start_time": "",
  "end_time": "",
  "gpu_id": 0,
  "const_mem_usage": -1,
  "local_mem_usage": 0,
  "reg_usage": 12,
  "error": ""
}
```

How do you go from a single replay to a feedback loop (autotune)?



How do you go from a single replay to a feedback loop (autotune)?



Execution With Multiple Workers

1. Create the sampling strategy
2. Invoke submit that returns a future
3. Get results (blocking call)

```python
SS = ExhaustiveSamplingStrategy(space)

for i, config in enumerate(SS):
    if not config.is_valid():
        continue
    futures.append((config, executor.submit(config)))

for i, (config, future) in enumerate(futures):
    val = future.result()
```

Mneme is extremely efficient in performing the feedback loop

Traditional Benchmark-Centric Measurement

- Full **host + device compilation** per experiment
- Whole-application execution (fork/exec, runtime overheads)
- Repeated **device memory initialization**
- Heavy **I/O and data marshaling**
- Output validation via **separate runs / subprocesses**

This Breaks Data-Driven Optimization

- Iteration cost measured in **minutes**
- Experiment throughput Repeated **device memory initialization**
- Exploration space collapses prematurely
- Decisions become **sample-starved**

Data-driven approaches don't need hundreds of samples —
they need thousands samples on 100s of benchmarks/kernels

Mneme is extremely efficient in performing the feedback loop

Traditional Benchmark-Centric Measurement

- Full **host + device compilation** per experiment
- Whole-application execution (fork/exec, runtime overheads)
- Repeated **device memory initialization**
- Heavy **I/O and data marshaling**
- Output validation via **separate runs / subprocesses**

Mneme Approach

- Compile **only the device code** of the kernel
- Execute only the kernel under investigation
- Single device memory initialization
- Minimal I/O
  - User should use the robust python ecosystem for persistent storage
- Automated bit wise exact validation

Data-driven optimization is fundamentally throughput-limited

Simple results on MiniFE

- Time to build miniFE:
  - 10s clean build, 4 seconds modifying single source file
- Time to execute miniFE:
  - No Recording : 36 seconds
  - With Recording: 38 seconds
    - This cost is paid once
    - Size the GPU snapshot and speed of IO define slowdown
- Back-of-the-envelope calculation:
  - To run 200 experiments and optimize a single kernel, we would need roughly:
    1. Run MiniFE : 200 * (4 (compile-time) + 7 (number of experiments to reduce noise) * 36) = **51200 seconds = 0.004 observation/second**
    2. Use sub process + standalone replay tool: 38 + 200 * (7 seconds) = **1438 seconds = 0.13 observations/second** ← SC-23
    3. Use python mneme interface (single worker): 38 + 120 (seconds) = **158 seconds = 1.26 observations/second** ← Mneme

Conclusions

- **LLVM enables deep GPU optimization — but experimentation cost limits exploration**
  - Traditional benchmark-centric workflows are too slow for data-driven optimization
  - Experiment throughput — not compiler capability — becomes the bottleneck
- **Proteus makes LLVM JIT specialization practical**
  - Low-overhead, programmable LLVM JIT for device code
  - Specialization and optimization close to the compiler pipeline
- **Mneme enables scalable, data-driven GPU optimization**
  - Record–replay decouples kernels from full applications
  - Orders-of-magnitude faster optimization feedback loops
  - Python-driven autotuning and analysis at scale

👉 High-throughput record–replay + LLVM JIT turns GPU optimization into a data-driven workflow