> *:warning: **This is an experimental feature***

Static Runtime

The premise of this approach is that a small subset of neural networks are well represented by a completely flattened dataflow graph. TorchScript supports a far more feature programming paradigm, so many models will not work out of the box.

Assumptions

This is a list of current assumptions for use with this feature.

- Inference only execution, CPU only
- Static input dtypes
- Static input shapes (the runtime supports dynamic shapes, but excessive dynamic shapes may degrade performance)

Threading model

Static runtime supports two execution modes.

Mode 1: single-threaded with no parallelism except for intra-op parallelism. For this mode, you can do either:

```
// m is the TorchScript module
auto runtime = StaticRuntime(m, opts);
auto output = runtime.run(args, kwargs);
```

or

```
auto mod = PrepareForStaticRuntime(m);
auto runtime = StaticRuntime(mod, opts);
auto output = runtime.run(args, kwargs);
```

Mode 2: similar to data parallelism, run the same model for different inputs on different threads at the same time. In this case, run `PrepareForStaticRuntime` to prepare the graph for Static Runtime. You should have one InferenceModule instance per model, and one Static Runtime instance per running thread. To avoiding creating StaticRuntime on the fly, use a synchronized stack (i.e. `boost::lockfree::stack`) to cache all the Static Runtime instances in your code.

```
// initialization
auto mod = PrepareForStaticRuntime(m);
// 128 is good for most cases. Pick a number that works for you
boost::lockfree::stack<std::shared_ptr<StaticRuntime>,
  boost::lockfree::fixed_sized<true>> pool(128);

// inference
std::shared_ptr<StaticRuntime> runtime = nullptr;
pool.pop(runtime);
if (!runtime) {
  runtime = std::make_shared<StaticRuntime>(mod, opts);
}
auto output = runtime->run(args, kwargs);
pool.push(runtime);
```

In both modes, `StaticRuntime` may not be used after its associated `StaticModule` is destructed!

Memory Planning

Static runtime's memory planner does two things:

1. Coalesces internal allocations for tensor storage
2. Does static analysis to figure out how to efficiently re-use memory.

For (2), there are two algorithms used. Specify which algorithm with the `memory_planner_algorithm` field in `StaticModuleOptions`. The algorithms are briefly described below:

Standard Resizing (default)

Static runtime will record the space required for each intermediate managed tensor it sees on the first inference iteration. An intermediate tensor is *managed* if two conditions are satisfied:

1. The op that produces it has an out variant. Out variants are wrappers around ops that conceptually transform the op's signature from `Tensor some_op(const Tensor& some_arg)` into `void some_op(Tensor& output, const Tensor& some_arg)`. Out variants are registered with static runtime via the `REGISTER_OPERATOR_FUNCTOR` macro; see "Registering Ops" for more info.

2. The tensor does not alias a graph output. Output tensors are handled separately by the memory planner, see "Managed Output Tensors" for details.

With this algorithm, static analysis is used to group the tensors in `StorageGroup`s. Tensors in the same storage group share memory, and two tensors can be in the same storage group if their lifetimes do not overlap.

On the subsequent iterations, static runtime allocates the tensor buffer at the start of the run. The amount of memory allocated is `sum([max(tensor.size()) for tensor in storage_groups])`.

If a tensor needs to be bigger than the allocated space on subsequent runs, a dynamic allocation will occur. This is why dynamic shapes will degrade performance. With the standard resizing strategy, static runtime will record the new largest tensor size in each storage group at the end of the iteration and allocate a buffer that is possibly bigger on the next iteration.

Precomputed Offsets Memory Planner (experimental)

This algorithm is based on arXiv:2001.03288, section 5.2 "Greedy by Size for Offset Calculation".

The paper describes the algorithm in detail, but the key considerations are:

1. This algorithm will tend to be more efficient with respect to maximum memory usage
2. This algorithm will *not* resize the tensor buffer since recomputing offsets is a quadratic operation. Therefore, to avoid performance degradation, the model should be warmed up with the largest possible inputs.

Managed Output Tensors

`StaticRuntime` can optionally manage output tensors via the `manage_output_tensors` option in `StaticModuleOptions`. When this flag is turned on, we coalesce allocations for output tensors together. Note that the buffer containing output tensors is separated from the one containing intermediate tensors. The former needs to live past the end of the inference run, but the latter needs deallocated at the end of the run.

Under the hood, we store a refcounted pointer to the output arena in each returned `Tensor`. The arena is destroyed only when all output tensors are destroyed.

```
auto output = runtime(args);
auto& elems = output.toTupleRef().elements();
auto tensor_1 = elems[0].toTensor();
auto tensor_2 = elems[1].toTensor();

tensor_1 = at::empty({0}); // Output buffer not deallocated yet!
tensor_2 = at::empty({0}); // This call deallocates the output buffer.
```

Registering Ops

Static runtime has three op execution modes:

1. Out variants: ops that return tensors which we may be able to manage. See "Memory Planning" for more details. Out variants are registered via the `REGISTER_OPERATOR_FUNCTOR` macro in `ops.h`.

```
REGISTER_OPERATOR_FUNCTOR(
  aten::op_name,
  aten_op_name, // This macro generates a struct, this field names it
  [](torch::jit::Node* n) -> SROperator {
    // This mechanism lets us support a subset of schemas
    if (n->matches(some_schema)) {
      return some_overload;
    } else if (n->matches(another_schema)) {
      return another_overload;
    }
    return nullptr;
  })
```

A `SROperator` is a type alias for `std::function<void(ProcessedNode*)>`. See "Implementation Details" for more details on `ProcessedNode`.

2. Native functions: just like out variants, except their outputs cannot be managed. This is because the op's return type is not a tensor or it is a view op (returns a tensor alias instead of a new tensor). Registration is done with `REGISTER_NATIVE_OPERATOR_FUNCTOR`. This macro is used in the same way as `REGISTER_OPERATOR_FUNCTOR`.

3. JIT fallback: static runtime has no implementation for this op, so the implementation that the JIT interpreter uses is selected instead.

When loading a model, ops are selected for each `torch::jit::Node` in the graph as follows:

1. If an out variant is registered, pass the node to the function that prodcues the `SROperator`. If the result is not `nulltpr`, use that op.
2. If a native function is registered, pass the node to the function that prodcues the `SROperator`. If the result is not `nulltpr`, use that op.
3. Use the JIT implementation. Static runtime will throw an exception if it does not exist.

Implementation Details

Structure and Lifetime Details

The following diagram shows the core data structure. An arrow from `A` to `B` means that `A` stores a reference to `B`. If the reference is unowned, `A` may not out live `B` or anything that `B` stores a reference to (directly or indirectly). If the reference is owned, the lifetimes of `A` and `B` are the same.

```
                     IValue array◄─────────────────┐┌────────────────────────────────────────┐
                          ▲                        │              Owns                      │
Owns
                          │                        │
┌───────────────────────────────►ProcessedNode───────►BlockRunner
                          │Owns                    │  │                                      │
│
                          │         Owns           │  │  Owns                                │
│
```

```
StaticModule◄──────────────StaticRuntime─────────────►BlockRunner──────────►MemoryPlanner │ ▼ │ │ │ │ │ ...
Owns│ │ │ │ │ ▼ │ │ │ │
BlockInfo◄ ├──────────────────────────────────────────────────┘ ─────────────────────┘ │ │ │
Owns│ │ ▼ │ ProcessedFunction
◄──────────────────────────────────────────────────────────────────────────────────────────┘
```

Each class is described in detail below.

`StaticModule` and `StaticRuntime`

`StaticModule` s are constructed from `torch::jit::Module` s and can be used to construct `StaticRuntime` instances. Each `StaticModule` caches exactly one `StaticRuntime` instance - it is lazily initialized when you access it via `runtime()` .

`StaticModule::operator()` can be used directly to make predictions. Under the hood, this method just forwards to the cached runtime's `StaticRuntime::operator()` . One upshot of this behavior is that `StaticModule::operator()` is not thread-safe.

The way to use static runtime in a multi-threaded context is to give each thread its own `StaticRuntime` instance. New runtime instances can be created directly (`StaticRuntime(static_module)`) or `clone()` 'd from an existing runtimes.

`StaticModule` takes a set of options that control the behavior of the runtime instances that it spawns; see `StaticModuleOptions` for more details.

Internally, `StaticRuntime` owns an array of `IValue` s that is referenced from all `BlockRunner` s and `ProcessedNode` s. All values that are generated at runtime are stored in this array.

`BlockRunner`

A `BlockRunner` represents a single sub-block in the graph. Every graph has at least one `BlockRunner` corresponding to the top-level block, and `StaticRuntime` starts its inference run by invoking `(*top_level_block)(args, kwargs)` . Each `BlockRunner` has its own `MemoryPlanner` and set of `ProcessedNode` s. Special nodes that have sub-blocks (like `prim::If`) might own `BlockRunner` s. The op implementations are responsible for invoking `BlockRunner` s corresponding to sub-blocks.

`MemoryPlanner`

See the "Memory Planning" section. `MemoryPlanner` is an abstract base class. Each sub-class implements a different memory planning algorithm.

In addition to the memory planning we do for tensors, `MemoryPlanner` encapsulates a few other optimizations.

- Managed output tensors (see "Managed Output Tensors")
- Borrowed `IValue` s; ops that just unpack their inputs (e.g. `dict_unpack`) might produce weak-references to avoid refcount bumps, the `MemoryPlanner` needs to destroy these borrows appropriately.

`ProcessedNode` and `ProcessedFunction`

`ProcessedNode` is our abstraction for a single op. Each `ProcessedNode` stores an unowned reference to `StaticRuntime` 's `IValue` array. It knows how to map input/output indices to indices in this array (so `processed_node->output(i)` returns a reference to `ivalue_array[some_set_of_indices[i]]`)

Each `ProcessedNode` stores a `ProcessedFunction` , which represents the actual op to execute. `ProcessedFunction` s are initialized upon `StaticModule` construction according to the out variant/native/JIT fallback lookup rules described in "Registering Ops". **Note that all `ProcessedFunction` s are shared amongst all runtime instances**, so all `ProcessedFunction` s must be thread-safe.