Automating Array Program Partitioning with PartIR

Dominik Grewe, Tamara Norman, Adam Paszke, Dougal Maclaurin, Norman Rink,
Michael Schaarschmidt, Georg Schmid, Tina Jia, Dimitrios Vytiniotis

DeepMind & Google Brain

November 2021

Programming with nd-arrays

Instructions manipulate nd-arrays, e.g. dot, conv2d, etc.

Origins in APL, Fortran, Matlab. Popular for numerical computing (numpy) and Deep Learning (Jax, PyTorch, TF)

Reasons for popularity: naturality/habituality, with notable exceptions trying to break out of the paradigm (e.g. Dex)

This talk: We take nd-array programming model as *given*, and focus on distributing programs written in this model (*)

(*) we will use semantic information not expressible in the model

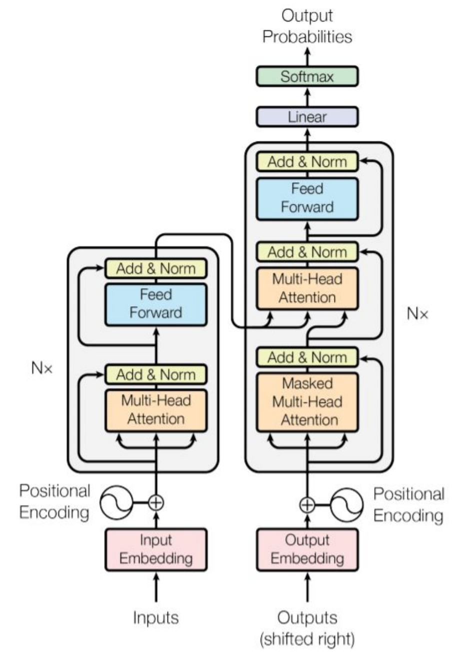


Figure 1: The Transformer - model architecture.

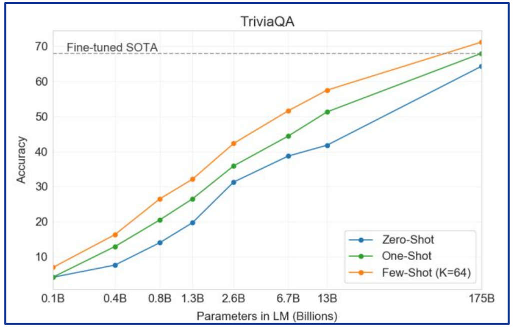
Deep Learning: a case for extreme scaling of array programs

Bigger DL models perform better

- GPT-3: 175B parameters, 3.6PFLOPS-days to train
- Many research scientists aim to scale up their models, need more memory and flops

Training on a single accelerator device (e.g. Google TPU, NVIDIA GPU) not sufficient for research. Hence accelerators typically come in system configurations with custom interconnect networks. e.g:

- Google TPU pods
- NVIDIA DGX POD and SuperPOD



Given the needs and the available HW systems, partitioning is key for:

- research velocity
- hardware ROI

yet remains an expensive and challenging task …

Scaling research workloads to multiple devices

1. Multi-device programming model e.g. JAX `pmap()`, `jit()`, collective communication, explicit device transfers.
    - Programmers responsible for performance and correctness
2. Single-device model with programmer-supplied “sharding annotations” driving compiler passes (e.g. GShard, GSPMD and `pjit()`, JAX `xmap()`)
    - Programmers responsible for performance
3. Automated search-based solutions e.g. Flexflow.

Require either known sharding strategy (e.g. Megatron) or often expensive rounds of annotation and profiling

We also aim for automation; but our motivation and design has differences

Our setting: partitioning for researchers

Support fast paced research program, diverse accelerator stacks

- Rich set of array programs (constrained only by a de-facto set of nd-array combinators)
- Platform independent (i.e. compiler & runtime, GPU/TPU etc.)

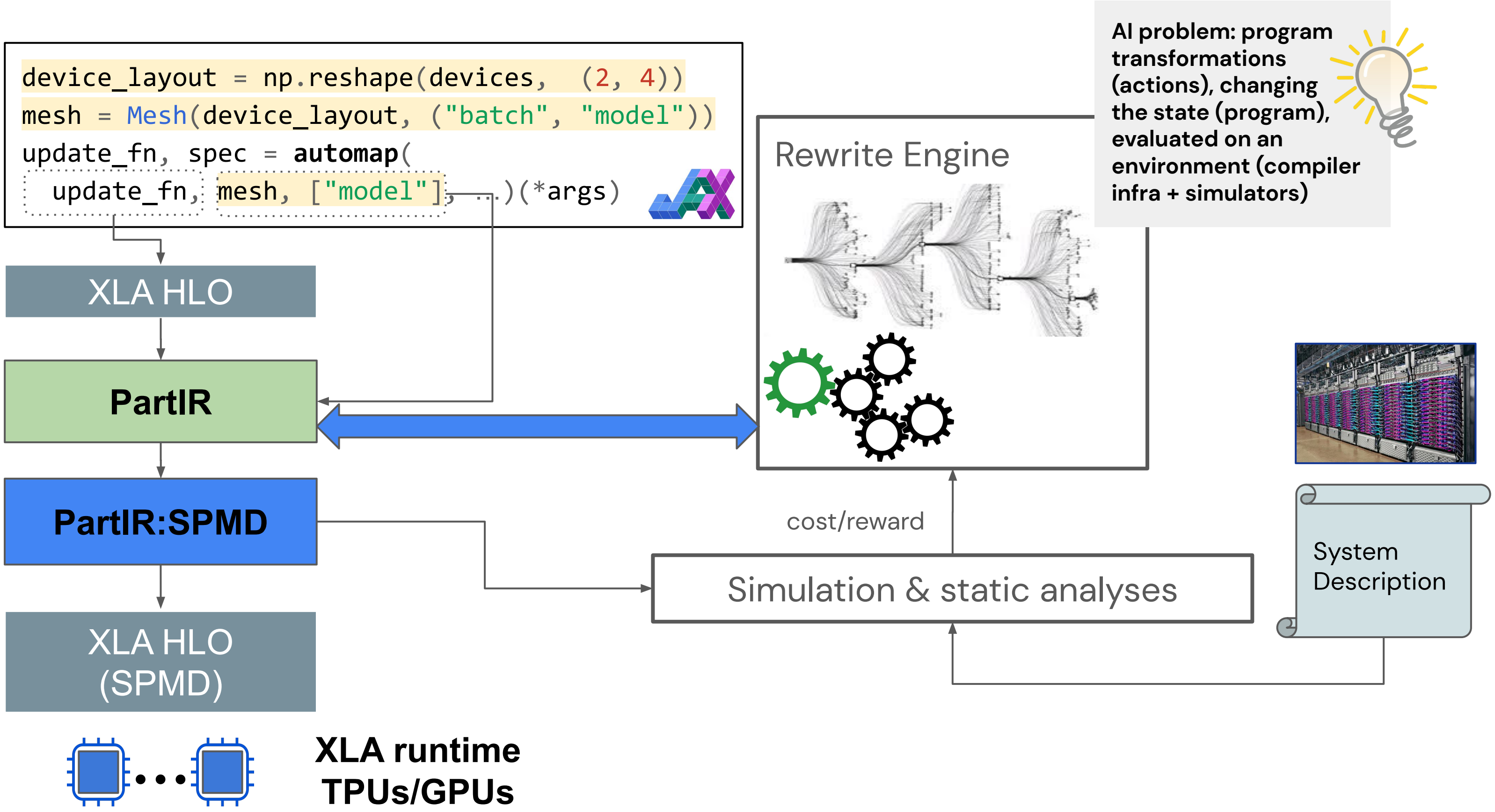
Solution must enable speedy R&D cycle (vs. production needs)

- Few minutes to a good-enough solution, scale with more resources
- Scalability-target: HLO programs of 50k-300k instructions > 1k chips

Minimize annotation burden and maximize composability

- Compose with other JAX APIs, such as pmap(), xmap(), pjit()
- Eliminate need to annotate user code with sharding annotations

PartIR and its user facing API



PartIR and its user facing API

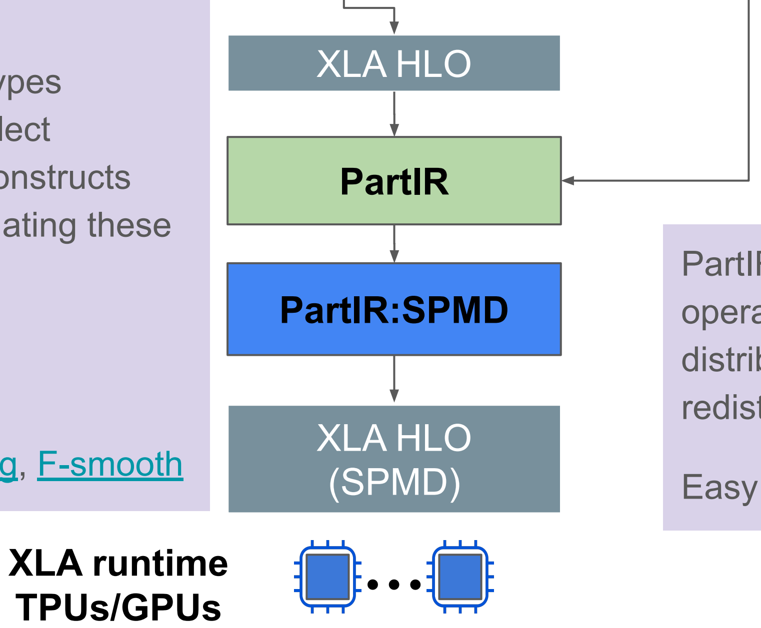
PartIR is an MLIR dialect layered on top of another array *dialect compiler*, and *runtime*.

Includes:
- Statically shaped array types shared with the array dialect
- Iteration and reduction constructs
- Rewrite rules for manipulating these

Nothing really to do with partitioning!

Close relatives: Dex, Linalg, F-smooth

```
device_layout = np.reshape(devices,  (2, 4))
mesh = Mesh(device_layout, ("batch", "model"))
update_fn, spec = automap(
  update_fn, mesh, ["model"], ...)(*args)
```



PartIR:SPMD: No native array operations either, types that express distribution and replication, explicit redistribution instructions, SPMD ops

Easy to reason about cost.

PartIR basics and rewrite system

* NB: we will ignore the meshes for this part of the talk and we will return later

PartIR constructs I: range values and slicing

Range type range<n> denotes set 0..(n-1). Use range values to slice:

```
slice d x[r]
```

A dimension index (attribute)

A tensor-typed value

A range-typed value

```
x : tensor<64x32x64xf32>
r : range<4>

slice 0 x[r] : tensor<16x32x64xf32>
slice 2 x[r] : tensor<64x32x16xf32>
```

PartIR constructs II: tiling a dimension with a range

PartIR introduces a higher-order loop-like expression for this:

A constant integer, which dimension to slice, here d=1

```
y = tile d (\(r : range<k>) -> expr)
```

```
y : tensor<32xnx16xf32>
```

```
expr : tensor<32x(n/k)x16xf32>
```

Semantics: a generator expression for the slices of a bigger array. It can be given either parallel or sequential semantics (depends on lowering)

PartIR constructs III: reductions

A similar higher-order operator:

```
y = sum (\(r : range<k>) -> expr)
```

```
y : tensor<32x16xf32>
```

```
expr : tensor<32x16xf32>
```

Semantics: sum together the k chunks of size `<32x16xf32>` to a single tensor of the same shape. Can be implemented with all-reduce in a distributed setting (see later)

Program equivalences in PartIR

Each tensor op in the underlying dialect is registered with information describing the equivalences of this op with tiling or reduction loops. Example:

```
x : tensor<nxmxf32>, y : tensor<mxoxf32>

matmul(x, y) == tile 0 (\r -> matmul(slice 0 x[r], y))
matmul(x, y) == tile 1 (\r -> matmul(x, slice 1 y[r]))
matmul(x, y) == sum (\r -> matmul(slice 1 x[r], slice 0 y[r]))
```

```
{ tile_mappings = [{1 -> (none, 1), 0 -> (0, none)], sum_mappings = [(1, 0)] }
```

Note: Data structure encodes information not present in the array programming model, but would be visible e.g. in Dex or Linalg.

Registering array ops mostly easy but can get hairy …

XLA Scatter Op

Scatter

The XLA scatter operation generates a result which is the value of the input array `operand`, with several slices (at indices specified by `scatter_indices`) updated with the values in `updates` using `update_computation`.

See also `XlaBuilder::Scatter`.

`scatter(operand, scatter_indices, updates, update_computation, index_vector_dim, update_window_dims, inserted_window_dims, scatter_dims_to_operand_dims)`

Arguments	Type	Semantics
`operand`	`XlaOp`	Array to be scattered into.
`scatter_indices`	`XlaOp`	Array containing the starting indices of the slices that must be scattered to.
`updates`	`XlaOp`	Array containing the values that must be used for scattering.
`update_computation`	`XlaComputation`	Computation to be used for combining the existing values in the input array and the updates during scatter. This computation should be of type `(T, T) -> T`.
`index_vector_dim`	`int64`	The dimension in `scatter_indices` that contains the starting indices.
`update_window_dims`	`ArraySlice<int64>`	The set of dimensions in `updates` shape that are *window dimensions*.
`inserted_window_dims`	`ArraySlice<int64>`	The set of *window dimensions* that must be inserted into `updates` shape.
`scatter_dims_to_operand_dims`	`ArraySlice<int64>`	A dimensions map from the scatter indices to the operand index space. This array is interpreted as mapping `i` to `scatter_dims_to_operand_dims[i]`. It has to be one-to-one and total.
`indices_are_sorted`	`bool`	Whether the indices are guaranteed to be sorted by the caller.

In the absence of formal semantics we have tests to guarantee that our registration is correct!

Rewriting: dumb-tiling actions

```
x : tensor<64x32xf32>

// dumb-tile(value=x,dim=0,range=16).
x ~~> tile 0 (\r:range<16> -> slice 0 x[r])

// dumb-tile(value=x,dim=1,range=16).
x ~~> tile 1 (\r:range<8> -> slice 1 x[r])
```

Propagation I: pushing forward

```
let x = tile 0 (\r -> e) in C[matmul(x,y)]
  ~~> let x = tile 0 (\r -> e) in C[tile 0 (\r -> matmul(slice 0 x[r], y))]

let x = tile 1 (\r -> e)) in C[matmul(x,y)]
  ~~> let x = tile 1 (\r -> e) in C[tile 1 (\r -> matmul(x, slice 1 y[r]))]

let x = tile 1 (\r -> e1) in … in
let y = tile 0 (\r -> e2) in C[matmul(x,y)]
  ~~> let x = tile 1 (\r -> e1) in … in
      let y = tile 0 (\r -> e2) in C[sum (\r -> matmul(slice 1 x[r], slice 0 y[r]))]
```

Propagation II: pushing backward

```
let x = matmul(a,b) in C[slice 0 x[r]]
  ~~> let x = tile 0 (\r -> matmul(slice 0 a[r], b) in C[slice 0 x[r]]

let x = matmul(a,b) in C[slice 1 x[r]]
  ~~> let x = tile 0 (\r -> matmul(a, slice 1 b[r]) in C[slice 1 x[r]]
```

Propagation III: propagate sideways

```
let x = tile 1 (\r -> e) in C[matmul(x,y)]
  ~~(and y is not a ‘tile 0’-op)~~>
let x = tile 1 (\r -> e) in
let y’ = tile 0 (\r -> slice 0 y[r]) in C[matmul(x,y’)]
```

Dumb-tiles operands based on some other operands or results being tiled (in our implementation we call this “inference”)

Propagation IV: Fusion

Happens once a tile def meets a slice use (*and we are allowed to inline)

```
let x = tile 0 (\r:range<k> -> expr) in C[slice 0 x[s]]
  ~~(s:range<k>)~~>
let x = ... in C[expr{s/r}]
```

A bit like beta-reduction (cf. also the Dex paper), also like in F-smooth [ICFP’19]

Towards lowering – device meshes

Options for lowering and executing PartIR programs

PartIR does not commit to a mode of execution, not even partitioning. Many options:

- **Option 1**: Lower `tile d (\r -> e)` and `sum (\r -> e)` to serial loops. May help lower peak memory on a single device ("micro-batching")

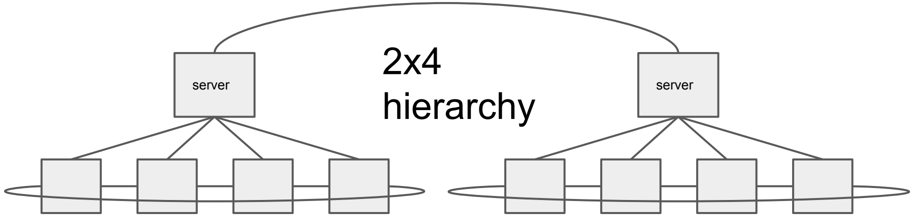
- **Option 2:** Lower to some form of fork-join parallelism on multicore machines

We are interested in (**Option 3**) SPMD parallelism: a highly-performant, well-supported and widely used model for systems of accelerators (e.g. dominant model in JAX/XLA)

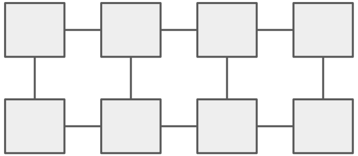
The first step is to introduce the concept of *meshes*.

Meshes: logical organization of devices as nd-arrays

```python
device_layout = np.reshape(devices,  (2, 4))
mesh = Mesh(device_layout, ("batch", "model"))
update_fn, spec = automap(
  update_fn, mesh, ["model"], ...)(*args)
```



2x4
hierarchy



2x4 grid

Similar concepts found in Mesh TF, JAX xmap(), and more …

Mesh-aware PartIR

PartIR iterations/reductions always have an associated mesh axis:

```
tile d axis (\(r : range<k>) -> expr)

sum axis (\r: range<k> -> expr)
```

A few well-formedness restrictions:

- Cannot double-nest the same axis
- Range type value must be equal to the corresponding axis size

Rewrite rules become stricter to ensure axes match and do not introduce non-well-formed programs.

Consequences of making the IR mesh-aware

1. Device assignment problem becomes trivial since each loop comes already annotated with a mesh axis.

2. Impose a strong prior/structure on the search space: rewrites will not need introduce entirely arbitrarily sized and wildly nested loops. Only:
    a. Loops associated with one of (a few) axes
    b. Nesting depth only up to the mesh rank

Note that (2) makes the search *largely independent of the number of actual available devices*, only dependent on the rank of the mesh

Example of mesh-aware rewriting

```
func @mlp(%x: tensor<16x256xf32>, %w: tensor<256x256xf32>, %u: tensor<256x256xf32>)
     attributes {batch:2, shard:2} {
   %0 = partir.tile 0 “batch” (%r : !partir.range<2>) {
     %1 = partir.slice 0 %x[%r]
     %2 = matmul(%1, %w)
     %3 = matmul(%2, %u)
     partir.yield %3
   }
   return %0
}
```

```
func @mlp(%x: tensor<>, %w: tensor<256x256xf32>,
         %u: tensor<256x256xf32>) attributes {batch:2, shard:2} {
   %0 = partir.tile 0 “batch” (%r : !partir.range<2>) {
     %1 = partir.slice 0 %x[%r]
     %2 = partir.sum “shard” (%s : !partir.range<2>) {
       %3 = partir.slice 1 %w[%s]
       %4 = matmul(%1, %3)
       %5 = partir.slice 0 %u[%s]
       %6 = matmul(%4, %5)
       partir.yield %6 }
     partir.yield %2 }
   return %0
}
```

Actions:
tile(arg=%w,dim=1,axis=”shard”)
infer-propagate

PartIR:SPMD

An IR suitable as target for lowering PartIR

- Distributed types that express replication or distribution
- Explicit redistribution commands (type casts)
- Reduction instructions along given mesh axes
- Explicit SPMD ops consisting of base-dialect (non-distributed) computations

We will illustrate key concepts of lowering PartIR to PartIR:SPMD

Lowering step 1: introduce SPMD op + lift free variables

```
%x : tensor<32x8xf32>
%y : tensor<8x16xf32>
%0 = tile 0 “xaxis” (%r:range<2>) { yield matmul(slice 0 %x[%r], %y)
```

Lift free variables %x and %y via replication, naively

Introduce a generic SPMD op on multiple arguments

```
%y1 = distribute(%y) : dist_tensor<[“xaxis”:range<2>],[8,16]>
%x1 = distribute(%x) : dist_tensor<[“xaxis”:range<2>],[32,8]>
%1 = spmd(%x1, %y1) [“xaxis”] (%r:range(2),
                    %x_arg : tensor<32x8xf32>,
                    %y_arg : tensor<8x16xf32>) {
       %2 = matmul(slice 0 %x_arg[%r], %y_arg);
       yield %2
     } : dist_tensor<[stacked “xaxis”:range<2>],[16,16]>
%2 = tile_stacked_tensor 0 (%1) :dist_tensor<[“xaxis”:range<2>], [32{0}, 16]>
%3 = undistribute(%2) : tensor<32x16xf32>
```

SPMD op returns a “stacked” tensor

Tile (i.e. rearrange stacked tensor into distributed tensor

Explicitly un-distribute to preserve original type

Lowering step 2: transform replication to distribution

```
%y1 = distribute(%y) : dist_tensor<[“xaxis”:range<2>],[8,16]>
%x1 = distribute(%x) : dist_tensor<[“xaxis”:range<2>,[32,8]>
%1 = spmd_op(%x1, %y1) [“xaxis”] (%r:range<2>,
                        %x_arg : tensor<32x8xf32>,
                        %y_arg : tensor<8x16xf32>) {
       %2 = matmul(slice 0 %x_arg[%r], %y_arg);
       yield %2
     } : dist_tensor<[stacked “xaxis”:range<2>], [16,16]>
%2 = tile_stacked_tensor 0 (%1) : dist_tensor<[“xaxis”:range<2>],[32{0}, 16]>
%3 = undistribute(%2) : tensor<32x16xf32>
```

```
%y1 = distribute(%y) : dist_tensor<[“xaxis”:range<2>],[8,16]>
%x1 = distribute(%x) : dist_tensor<[“xaxis”:range<2>],[32,8]>
%x1 = distribute(%x) : dist_tensor<[“xaxis”:range<2>],[32{0},8]>
%1 = spmd_op(%x1, %y1) [“xaxis”] (%r:range<2>,
                        %x_arg : tensor<16x8xf32>,
                        %y_arg : tensor<8x16xf32>) {
       %2 = matmul(%x_arg, %y_arg);
       yield %2
     } : dist_tensor<[stacked “xaxis”:range<2>],[16,16]>
%2 = tile_stacked_tensor 0 (%1) : dist_tensor<[“xaxis”:range<2>, [32{0}, 16]>
%3 = undistribute(%2) : tensor<32x16xf32>
```

More lowering details

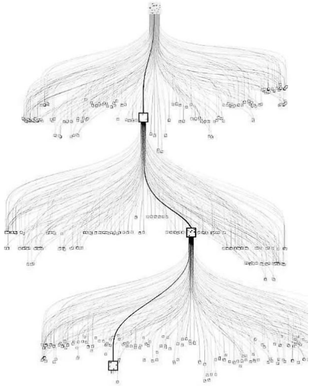
- For translating sum we introduce spmd op + reduction over relevant axis
- Supported: nested partir.tile and partir.sum, including non-perfect nests
- Fusion of distribution operators:
  - `undistribute(distribute[τ](%x) ~> %x`
  - `distribute[τ](undistribute(%x) ~> %x`
    `when type(%x) == τ`
  - `distribute[τ](undistribute(%x) ~> redistribute %x`
    `when globalType(type(%x) == globalType(τ)`
- Final pass to convert functions to receive/return distributed types by removing initial `distribute()` calls and final `undistribute()` calls

Search design and initial results

Search design

Three components in our design:

- **Rewrite actions** to partition function arguments given a mesh
- Rules that **propagate** these actions throughout the program.
  - Frequently results in partitioning other arguments accordingly (e.g. parameter -> opt. state for this param)
- **Cost models** based on memory and/or runtime estimation



Key insights:

- Users decide on mesh in advance (number and axes to partition over)

  ⇒ search space becomes independent of the # of devices

- Mimic expert human partitioning by propagating argument decisions

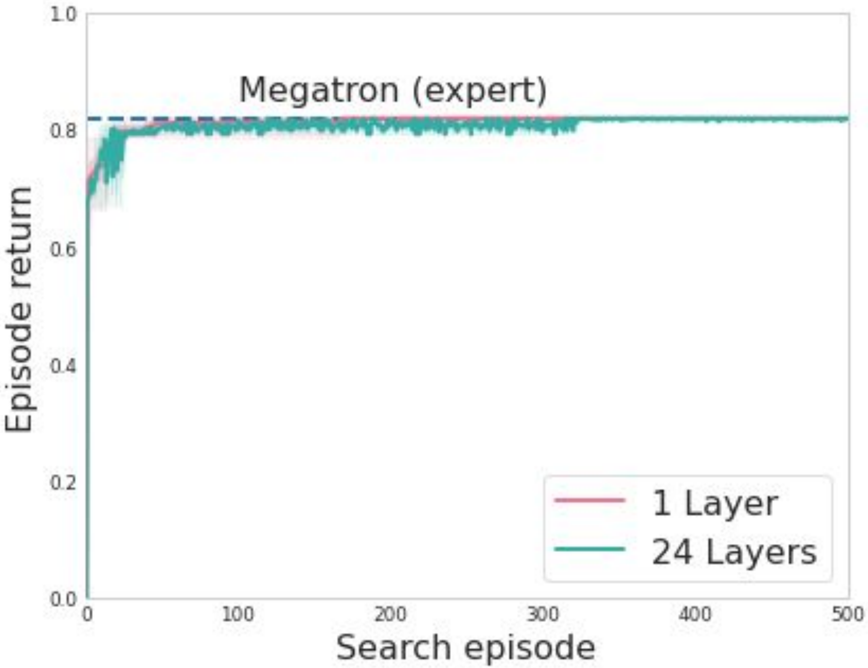
  ⇒ search space largely independent of the # of total ops

Transformers

Setup: GPT-3 style transformers of different sizes (e.g. 27 GB initial memory for 24 layers, device = TPU v3)

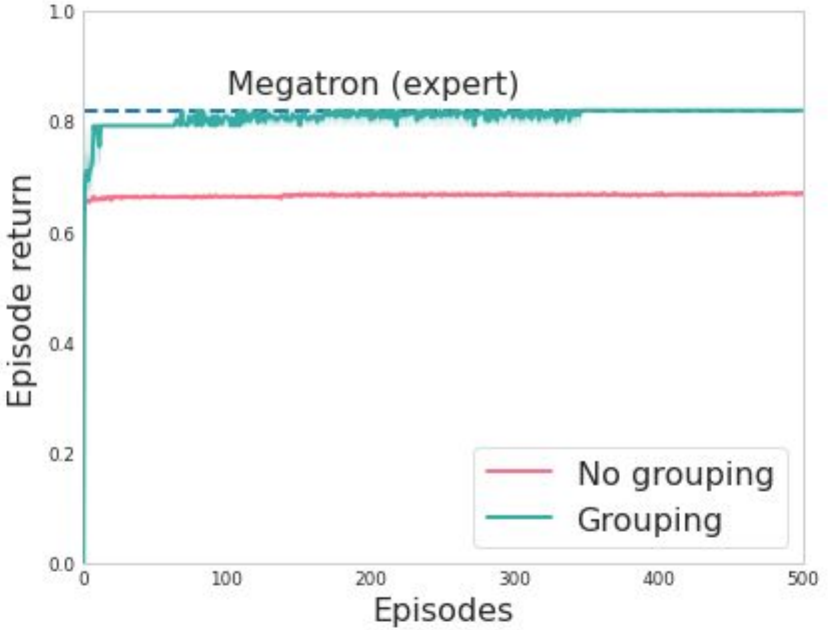
Known expert strategy: Megatron sharding

Status: Can achieve Megatron reliably (100% of 25 seeds in nightly benchmarking) within <1k episodes



Performance analysis: layer grouping

ML models often have **regular structure** with the same blocks of layers repeated (e.g. Transformer, ResNet etc). High-level layer libraries (eg Haiku) maintain this structural information enabling us to detect repeated blocks.



Argument grouping: Deciding once per repeated block is key to scale to large depths.

Supported by automap via **grouping hints.**

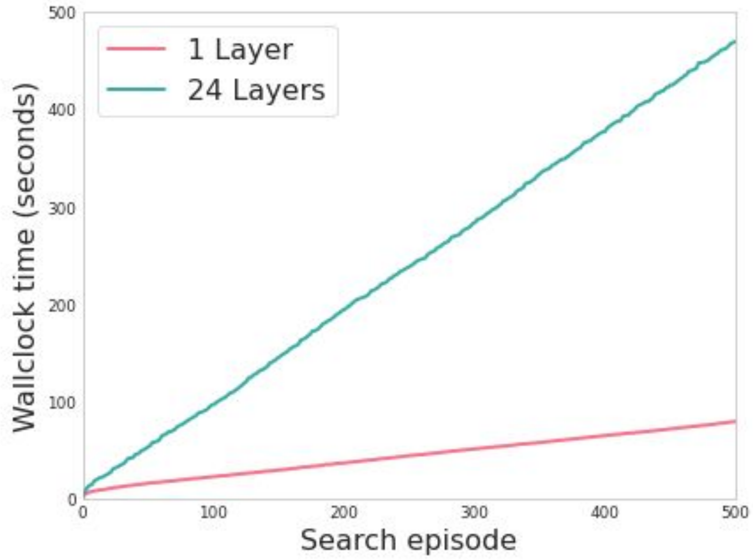
```
argument_grouping = argument_grouping_utils.get_repeating_params_haiku(
    abstract_params, 'transformer_block')
... = automap(..., partir_hints={'argument_grouping': argument_grouping}
```

Performance analysis: search time

Search performance: Search time scales with model size; ongoing work to speed up step time

Default automap setting: Use host resources, multi-threaded search

Caching search results: reuse across preemptions (and experiments)



Very slow search step time!
few episodes ~ sec

Recap and Status on Search and APIs

- ✓ A functioning JAX API
- ✓ Reach expected performance on a variety of models
- ✓ Early integration of automap with pilot users
- Work In Progress: Not yet battle-tested
- Work In Progress: User hints on model structure required to reach good performance
- Work In Progress: Missing features (e.g. control flow support)

Also have an active foray into learnt policies for controlling the MCTS search, tune-in for the coming NeurIPS 2021 ML for Systems workshop for a presentation by Michael Schaarschmidt.

Automap: Towards Ergonomic Automated Parallelism for ML Models

https://nips.cc/Conferences/2021/ScheduleMultitrack?event=21866#collapse35261

What’s Next and Conclusions

Very active area of work in our team

Current and planned work:

- Tiling through control flow constructs (almost ready)
- Padding design for non-divisible dimension by axes sizes
- Optimize search performance, introduce learnt policies and costs
- Revisit rewrite engine to understand effects of “races”
- More efficient ways to express choices without eager rewriting
- MCTS optimizations (caching of intermediate states vs recalculation)
- More expressive forms of parallelism (e.g. pipelining)
- APIs for distributed data loading and checkpointing
- …

Thank you!

- Need abstractions for partitioning in our compiler stacks that are platform-independent and flexible
- PartIR offers a principled approach to partitioning via semantics preserving sequences of really simple transforms, rooted in deforestation and fusion ideas from declarative programming
- Lower level type system that can reason about data redistribution
- Real workbench to explore program transformation through search, constraint solving, super-optimization, ML/RL

The work is ramping up, keen to engage and collaborate!