PartIR

Declarative abstractions for tensor program partitioning

Dimitrios Vytiniotis
dvytin@google.com

Dominik Grewe, Michael Schaarschmidt, James Molloy, Dan Belov
DeepMind
Adam Paszke, Dougal Maclaurin, Nicolas Vasilache
Google Brain

PPDP’20, September 2020

An invited talk about work-in-progress?!?

What is a machine learning model?

- Typically "beefy" multi-dimensional array programs: *tensor programs*
- Accept inputs and parameters:
  - Wild variation on input batch size, input spatial dimensions, and parameter size
- Parameters + compute grow:
  - GPT-3 is a 175B param transformer model
- Ever-growing body of work to reduce parameters and compute:
  - Exploit sparsity [not the topic of this talk]
  - Novel ways to *partition these tensor programs* to exploit parallelism and make them fit memory constraints

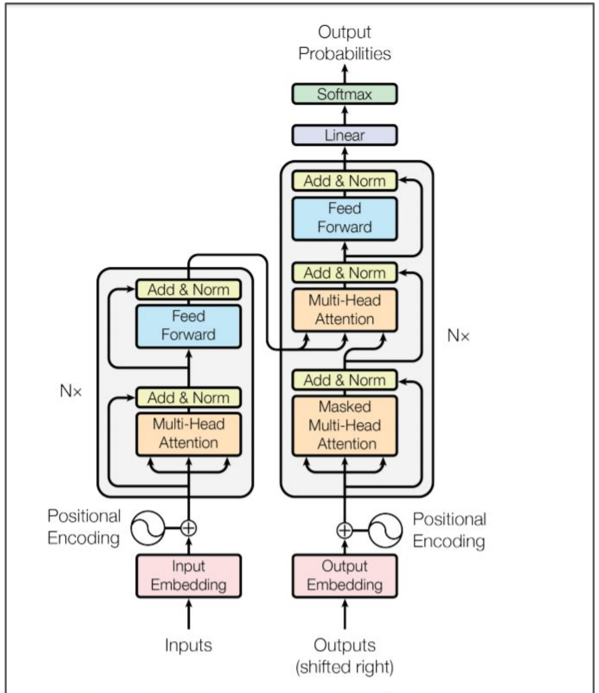
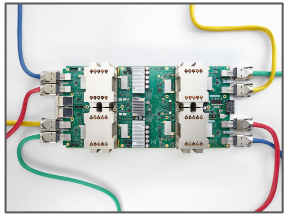


Figure 1: The Transformer - model architecture.

Machine learning hardware: from tensor cores to systems

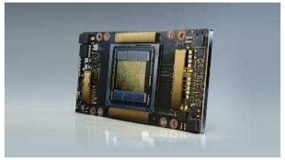
Cannot run a model with 175B parameters on a single machine!

Accelerator trend for scaling: chips => systems of multiple chips => data-centers

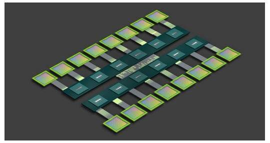


TPUs: few tensor cores, ~10-100GBs of accelerator memory

TPU pods: thousands of TPU "cores" in custom interconnects



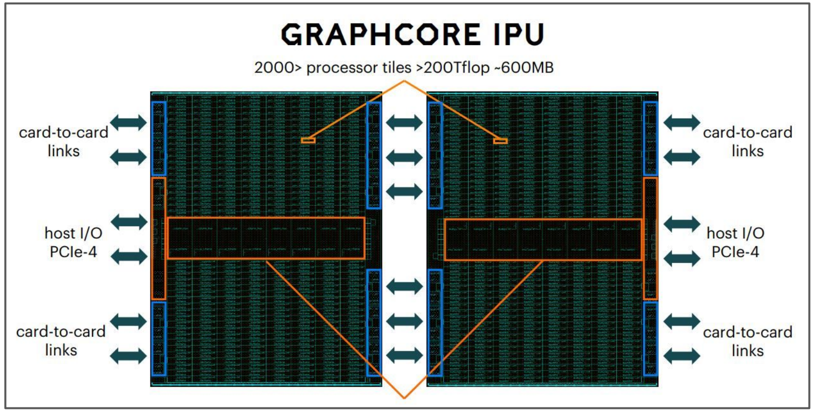
NVIDIA A100 GPU

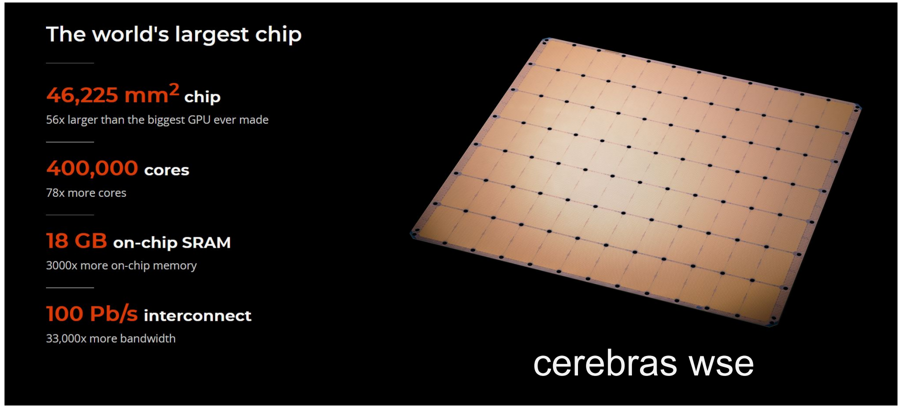


nvswitch + nvlink for data-center scale

Accelerator scaling trend: lots of wimpy cores

New and cheaper ways to scale compute and memory than monolithic MXU cores




The world's largest chip

46,225 mm^2 chip
56x larger than the biggest GPU ever made

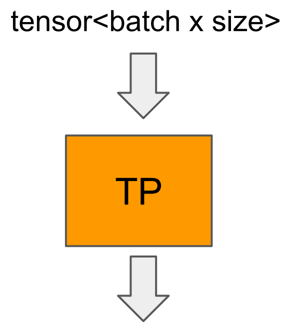
400,000 cores
78x more cores

18 GB on-chip SRAM
3000x more on-chip memory

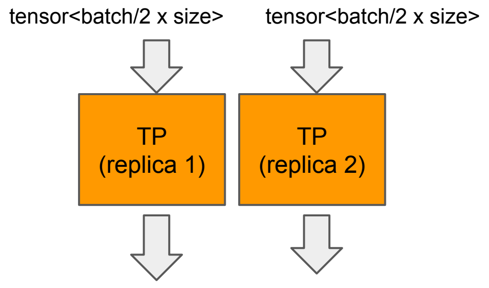
100 Pb/s interconnect
33,000x more bandwidth

cerebras wse

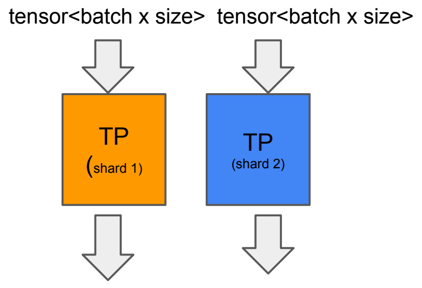
Multiple ways to partition a tensor program



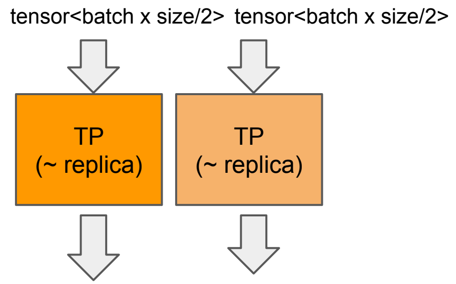
- Partitioning a model is a *program transformation*
- Often need to additionally mix-and match different forms in the same model
- Beyond Data and Model Parallelism for Deep Neural Networks, SysML'19



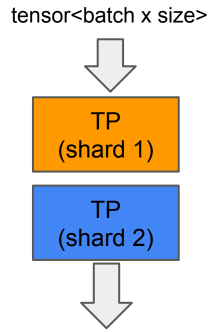
Data Parallelism



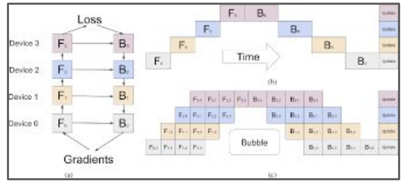
Param sharding (v1)



Input (feature) parallelism



Param sharding (v2)



Pipelining variant (gPipe)

Sounds simple? Still SW in a BAD place for partitioning

- We use data parallelism to reduce input (activation) sizes and scale the available compute flops, *assuming model parameters fit on available cores*
  - OK for TPUs and small models (~ 8-32GB/core), but new hardware may come with few GBs per device (e.g. Graphcore IPU). How does one map a large transformer model?
- We bake in system assumptions in library code
  - Typically batch parallelism, no model parallelism
- Engineering **years** on specialized partitions (papers: Megatron, gPipe).
  - New partitioning strategy ~> new research paper!
- System design: manual back-of-the-envelope-calculations for mapping models on new systems. Hard to support a mixture of partitioning strategies

THIS IS MADNESS!!!!!

PartIR aspiration: generic partitioning compiler infra

PartIR offers:

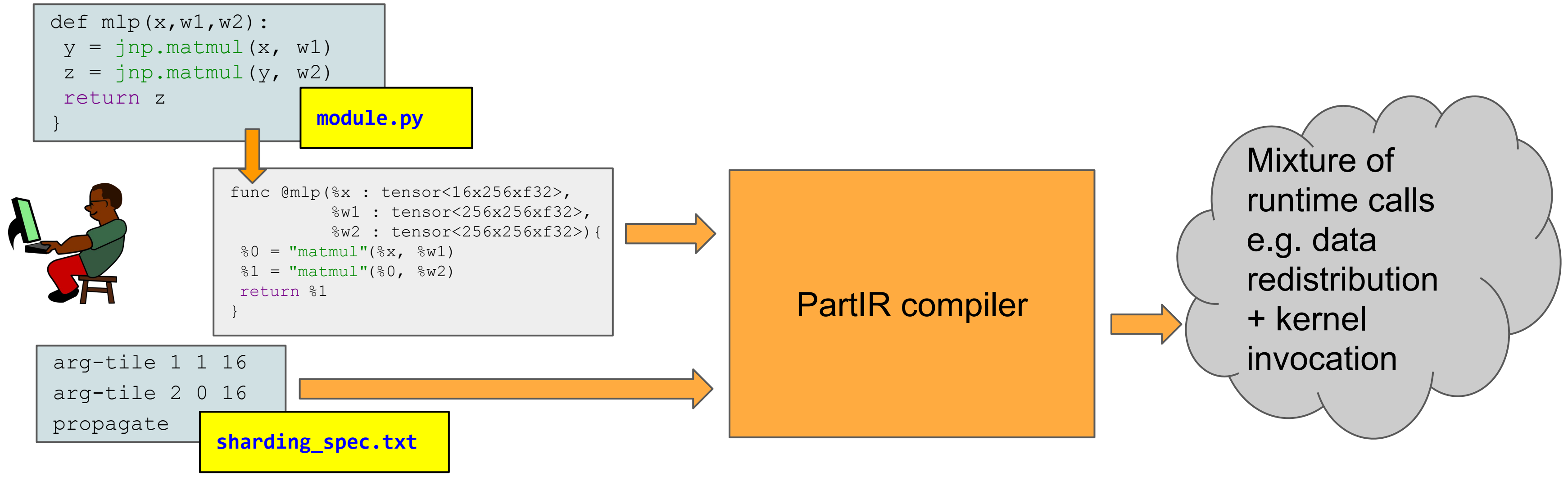
- Declarative abstractions to express partitioned models on accelerator systems
- Declarative transformation rules for partitioning actions and fusion
- Vision: driven by sharding annotations, or interactive tactics, or search + RL (non-exclusive)

Other features:

- POC lowering pass to dataflow graphs of SPMD ops (beyond a single SPMD op)
- Solves the "tiling-on-tensors" problem i.e. a functional specification of tiling
- Useful to explain semantics of related efforts (e.g. XLA sharding propagation, Mesh TF); eventually may increase expressivity (e.g. dataflow and control flow of SPMD ops, pipeline parallelism)
- Close relatives: Dex, Linalg, F-smooth [ICFP'19]
- Relies on some existing tensor op backend compiler and runtime

PartIR value; for ML researchers

Reusable math-focused (*vs. systems-focused*) ML libraries



```
def mlp(x,w1,w2):
 y = jnp.matmul(x, w1)
 z = jnp.matmul(y, w2)
 return z
}
```

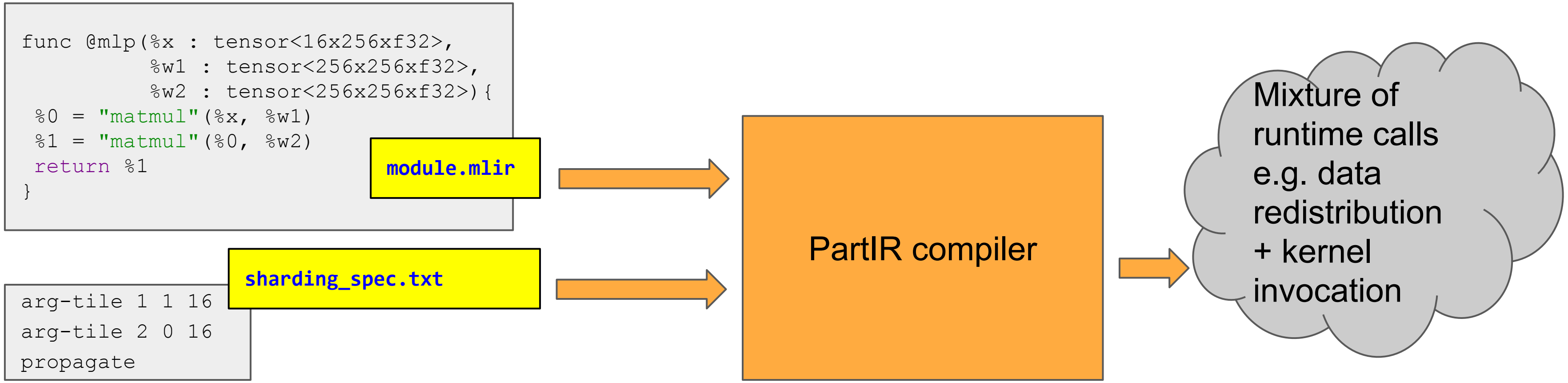
```
func @mlp(%x : tensor<16x256xf32>,
          %w1 : tensor<256x256xf32>,
          %w2 : tensor<256x256xf32>){
 %0 = "matmul"(%x, %w1)
 %1 = "matmul"(%0, %w2)
 return %1
}
```

```
arg-tile 1 1 16
arg-tile 2 0 16
propagate
```

Mixture of runtime calls e.g. data redistribution + kernel invocation

PartIR value; for compiler infrastructure teams

Reusable partitioning compiler infra; across tensor dialects, different systems of accelerators



```
func @mlp(%x : tensor<16x256xf32>,
          %w1 : tensor<256x256xf32>,
          %w2 : tensor<256x256xf32>){
 %0 = "matmul"(%x, %w1)
 %1 = "matmul"(%0, %w2)
 return %1
}
```

```
arg-tile 1 1 16
arg-tile 2 0 16
propagate
```

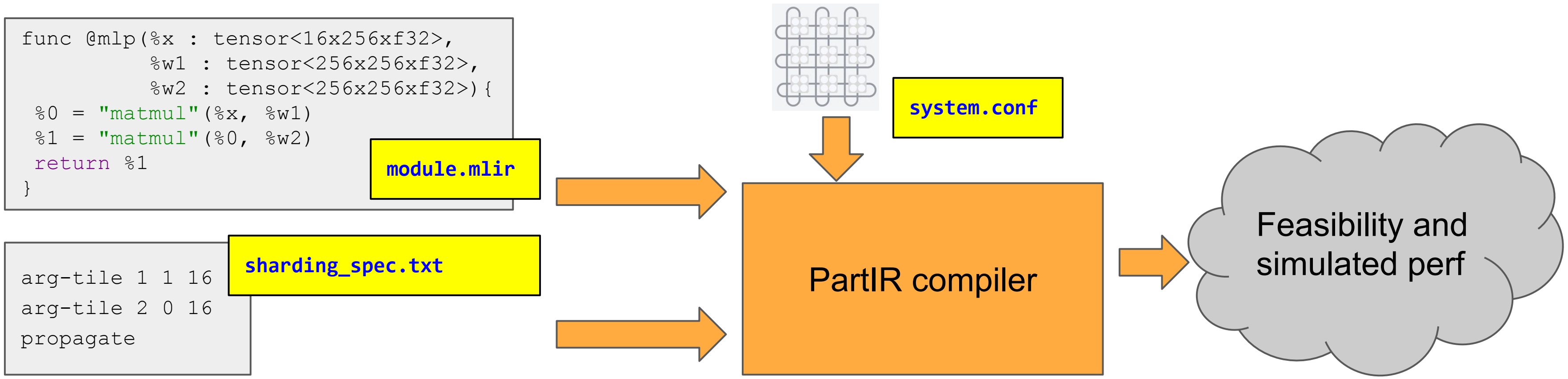
module.mlir
sharding_spec.txt
PartIR compiler
Mixture of runtime calls e.g. data redistribution + kernel invocation

Do not invent a new IR from scratch: we have mature tensor IRs for different needs (e.g. XLA, LinAlg), need common infra to partition across accelerator systems. PartIR is just a generic partitioning framework on top: *it relies on a backend compiler and a runtime, for a given target (e.g. XLA for TPUs)*

PartIR value; to speed up systems design

Systems Design questions: what if 2x memory/core? 2x cores with ½ mem? 2x faster mesh? 2x FLOPS/sec/core?

How can we quickly experiment with partitioning and mapping important models in different ways? At the moment it's months of engineering time but a ton of boilerplate.



```
func @mlp(%x : tensor<16x256xf32>,
          %w1 : tensor<256x256xf32>,
          %w2 : tensor<256x256xf32>){
 %0 = "matmul"(%x, %w1)
 %1 = "matmul"(%0, %w2)
 return %1
}
```

```
arg-tile 1 1 16
arg-tile 2 0 16
propagate
```

A technical tour of PartIR

PartIR sits on top of some other tensor dialect, e.g. XLA:

- Using the MLIR (multi-level IR, https://mlir.llvm.org/) framework this is technically easy to express
- MLIR is infrastructure for representing multiple IRs (*dialects*). Allows programs to span operators from different dialects and features generic binding trees facilities, generic representations of modules, functions, SSA blocks, rewriting APIs, an extensible type system and more
- XLA is a dialect in MLIR and PartIR simply adds a few constructs as a separate dialect

PartIR constructs:

- A range type (partir.range) whose values represent contiguous non-overlapping intervals
- 3 operators specific to partitioning:
  - An op to slice a dimension (partir.slice)
  - An op to tile a dimension (partir.tile)
  - An op to do a tiled reduction (partir.sum)

Not discussed today: a handful of extra operators that allow tensor *building* (as opposed to *partitioning*): out of scope for today's talk, but very useful for e.g. Dex. Such operators were first introduced in the F-smooth paper.

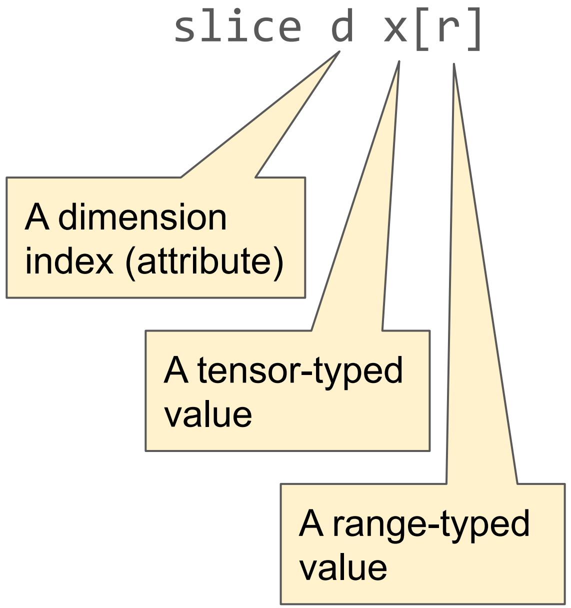
PartIR range type and its values: `range<n,w>`

A range type `range<n,w>` denotes the set of contiguous non-overlapping intervals of a range `n`, each interval being of width `w`.

- Assume `(x:range<16,8>)`. Then x is one of: [0..8), [8..16)
- Non-overlapped and **perfectly divisible**, for now. (n `mod` w == 0)
- We can nest ranges in a monoid fashion (will skip for this talk)

PartIR constructs: slicing a dimension with a range value

Key idea: use range values directly in slicing



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

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

slice 0 x[r] : tensor<4x32x64xf32>
slice 1 x[r] : TYPE-ERROR
slice 2 x[r] : tensor<64x32x4xf32>
```

PartIR constructs: 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<n,w>) -> expr)
```

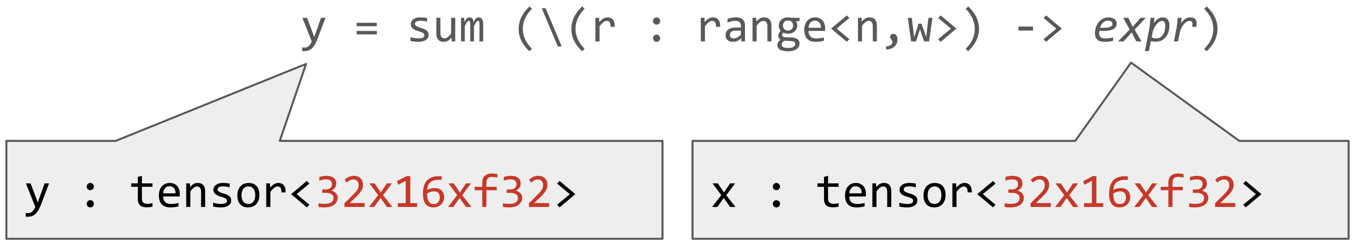
`y : tensor<32xnx16xf32>`

`expr : tensor<32xwx16xf32>`

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: reductions

A similar higher-order operator:



```
y = sum (\(r : range<n,w>) -> expr)
```

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

```
x : tensor<32x16xf32>
```

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

Partitioning = progressive application of rewrite rules

```
x : tensor<nxmxf32>, y : tensor<mxoxf32>
matmul-tile-0:
   matmul(x, y) <~~> tile 0 (\r:range(n,nw) -> matmul(slice 0 x[r], y))
matmul-tile-1:
   matmul(x, y) <~~> tile 1 (\r:range(o,ow) -> matmul(x, slice 1 y[r]))
matmul-sum:
   matmul(x, y) <~~>
      sum (\r:range(m,mw) -> matmul(slice 1 x[r], slice 0 y[r]))
```

- Rewrite rules preserve types and semantics
- Each tensor operator from our base dialect is equipped with annotations that inform how each dimension could be partitioned, and how this propagates to argument slicing.
- Opportunity: given the mathematical definition of an op or even a sub-program search for valid rewrites + validate through a theorem prover. Reminiscent of TASO [SOSP'19])

Producer-consumer fusion = when slice met tile

```
s : range<64,32>

fuse:
  slice 0 (tile 0 (\r:range<64,32> -> expr)[s] ~~> expr{s/r}
```

Analogous to the build-slice fusion rule from F-smooth paper [ICFP’19]

Slicing function arguments or values (aka “dumb tiling”)

```
x : tensor<64x32xf32>

x <~~> tile 0 (\r:range(64,4) -> slice 0 x[r])
x <~~> tile 1 (\r:range(32,4) -> slice 1 x[r])
```

Propagation tactics - push slicing **in**

```
slice 0 (matmul(x, y))[r] ~~> matmul(slice 0 x[r], y)

slice 1 (matmul(x, y))[r] ~~> matmul(x, slice 1 y[r])
```

Intuition: do not compute a big matmul only to take a slice out of it! Instead, directly compute the slice. Also note: these are derived rules from fusion + the specialized matmul ops, but useful on their own!

Propagation tactics - pulling tiling **out**

```
matmul(tile 0 (\r:range(n,nw) -> expr), y) ~~>
               tile 0 (\r:range(n,nw) -> matmul(expr, y))

matmul(x, tile 1 (\r:range(o,nw) -> expr)) ~~>
               tile 1 (\r:range(o,ow) -> matmul(x, expr))
```

Intuition: try to expose looping constructs at the top-level

But needs to be done carefully: we are pulling “y” **into** the tiling construct, hence likely to increase the local memory requirements on every compute core!

A space of user-controlled actions + propagation

```
 func @mlp(%x: tensor<16x256xf32>,
           %w: tensor<256x256xf32>,
           %u: tensor<256x256xf32>)
-> tensor<16x256xf32> {
   %0 = matmul(%x, %w)
   %1 = matmul(%0, %u)
   return %1
 }
```

arg-tile 0 0 8.

```
 func @mlp(%x: tensor<16x256xf32>, %w: tensor<256x256xf32>,
          %u: tensor<256x256xf32>) -> tensor<16x256xf32> {
   %0 = partir.tile 0 (%r : !partir.range<16,8>) {
     %3 = partir.slice 0 %x[%r]
     partir.yield %3
   }
   %1 = matmul(%0, %w)
   %2 = matmul(%1, %u)
   return %2
 }
```

Data parallelism

propagate.

```
func @mlp(%x: tensor<16x256xf32>, %w: tensor<256x256xf32>, %u: tensor<256x256xf32>)
-> tensor<16x256xf32> {
   %0 = partir.tile 0 (%r : !partir.range<16,8>) {
     %1 = partir.slice 0 %x[%r]
     %2 = matmul(%1, %w)
     %3 = matmul(%2, %u)
     partir.yield %3
   }
   return %0
}
```

arg-tile 1 1 128;
arg-tile 2 0 128;
propagate.

```
func @mlp(%x: tensor<>, %w: tensor<256x256xf32>,
          %u: tensor<256x256xf32>) -> tensor<16x256xf32> {
  %0 = partir.tile 0 (%r : !partir.range<16,8>) {
    %1 = partir.slice 0 %x[%r]
    %2 = partir.sum (%s : !partir.range<256,128>) {
      %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
}
```

Data parallelism
+
parameter sharding

Lowering PartIR

Since many accelerators offer facilities for:

- data transfers between devices and re-distribution
- SPMD-style ops across a number of available cores

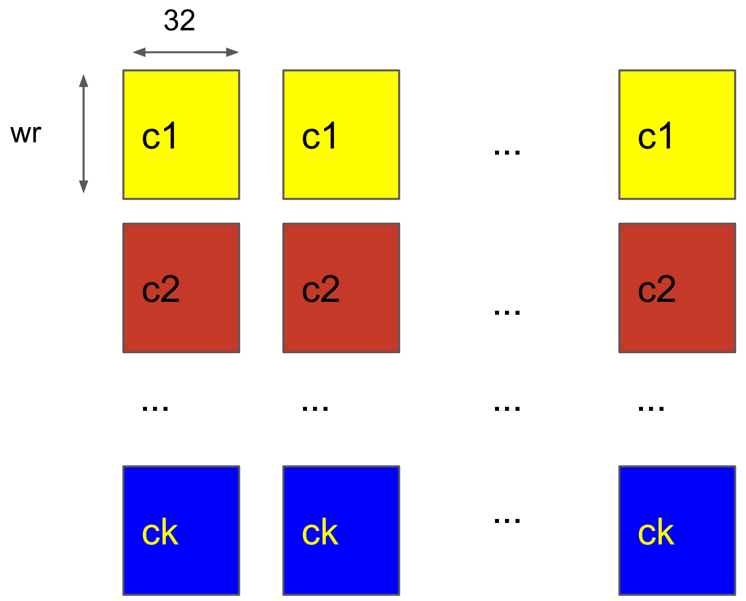
we sketch out a translation of PartIR programs to such an SPMD dialect, which we call PartIR:SPMD. This entails several steps:

- Introducing the type system of PartIR:SPMD
- Introducing the ops of PartIR:SPMD
- Introducing progressive lowering to PartIR:SPMD

PartIR:SPMD Distributed tensor types

```
distributed_tensor<(r:range(nr,wr), s:range(ns,ws)) -> [r,32]>
```

binding occurrences

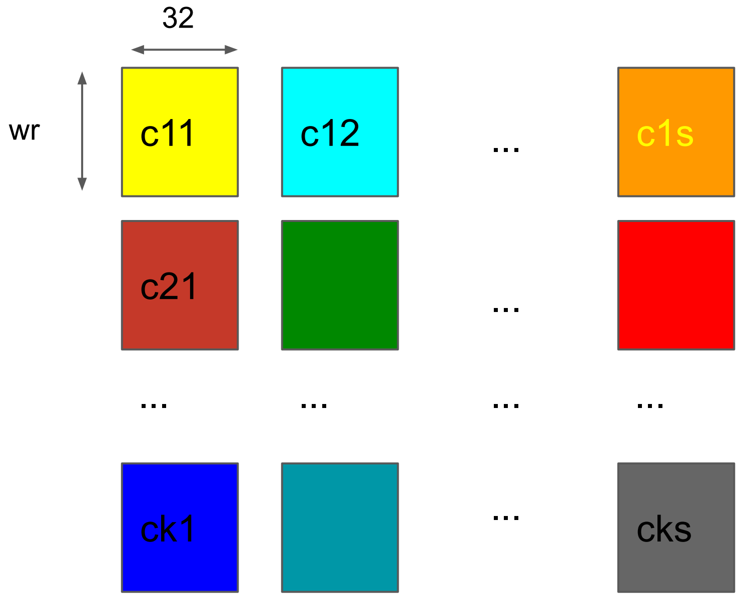


k = nr / wr

horizontal replication = ns/ws

PartIR:SPMD Stacked tensor types

```
distributed_tensor<(r:range(nr,wr), @stacked s:range(ns,ws)) -> [r,32]>
```



k = nr / wr
s = ns / ws

Summary: `distributed_tensor<(rs) -> shape>`

- Chunks = Π(`|r.size/r.tilesize| for r in rs)`
- `stacked r ∉ rangeVars(shape)` => stacking
- `non stacked r ∉ rangeVars(shape)` => replication

Lowering step 1: introduce SPMD op + lift free variables

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

```
%y1 = distribute(%y) : distributed_tensor<(%r:range(32,16)) -> [8,16]>
~~%x1 = distribute(%x) : distributed_tensor<(%r:range(32,16)) -> [32,8]>~~
%x1 = distribute(%x) : distributed_tensor<(%r0:range(32,16)) -> [%r0,8]>
%1 = spmd_op(%x1, %y1) (%r:range(32,16),
                        %x_arg : tensor<16x8xf32>,
                        %y_arg : tensor<8x16xf32>) {
       %2 = matmul(%x_arg, %y_arg);
       yield %2
     } : distributed_tensor<(@stacked %r:range(32,16)) -> [16,16]>
%2 = tile_stacked_tensor 0 (%1) : distributed_tensor<(r:range(32,16)) -> [r, 16]>
%3 = undistribute(%2) : tensor<32x16xf32>
```

More lowering details (known and unknown)

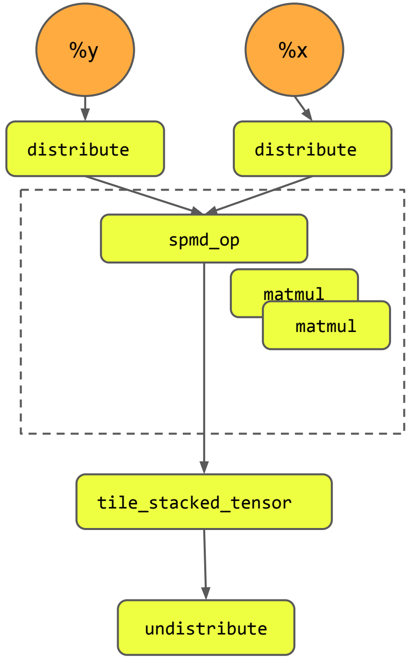
- For translating partir.sum we introduce spmd + partir.all_reduce op
- Nested partir.tile and partir.sum
- Non-perfectly nested code inside partir.tile/partir.sum
  - Need to choose between creating distributed versions of intermediates, or inlining in inner loops and replicating computation. Expose option to programmers
- 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)
    ```

Execution

```
%y1 = distribute(%y) : distributed_tensor<(%r:range(32,16)) -> [8,16]>
%x1 = distribute(%x) : distributed_tensor<(%r0:range(32,16)) -> [%r0,8]>
%1 = spmd_op(%x1, %y1) (%r:range(32,16),
                        %x_arg : tensor<16x8xf32>,
                        %y_arg : tensor<8x16xf32>) {
       %2 = matmul(%x_arg, %y_arg);
       yield %2
     } : distributed_tensor<(@stacked %r:range(32,16)) -> [16,16]>
%2 = tile_stacked_tensor 0 (%1) : distributed_tensor<(r:range(32,16)) -> [r, 16]>
%3 = undistribute(%2) : tensor<32x16xf32>
```



- Compile all bodies of spmd_op using the backend compiler
- Literally execute the data-flow graph using the backend runtime (must support the redistribution operators and SPMD execution)

Range widths that do not divide the dimension?

I.e. let the last interval have a smaller width

Proposal: use dependent types

- Semantics: Range(16, 5) = [0..4), [5..10),[10..15), [15,16)
- Introduce dependent types:
  - `slice d %x[%r] : <n1, n2, .., %r, .., n_k>`
- Type partir.tile using dependent types:

BEFORE

```
y : tensor<32xnx16xf32>                x : tensor<32xwx16xf32>
```

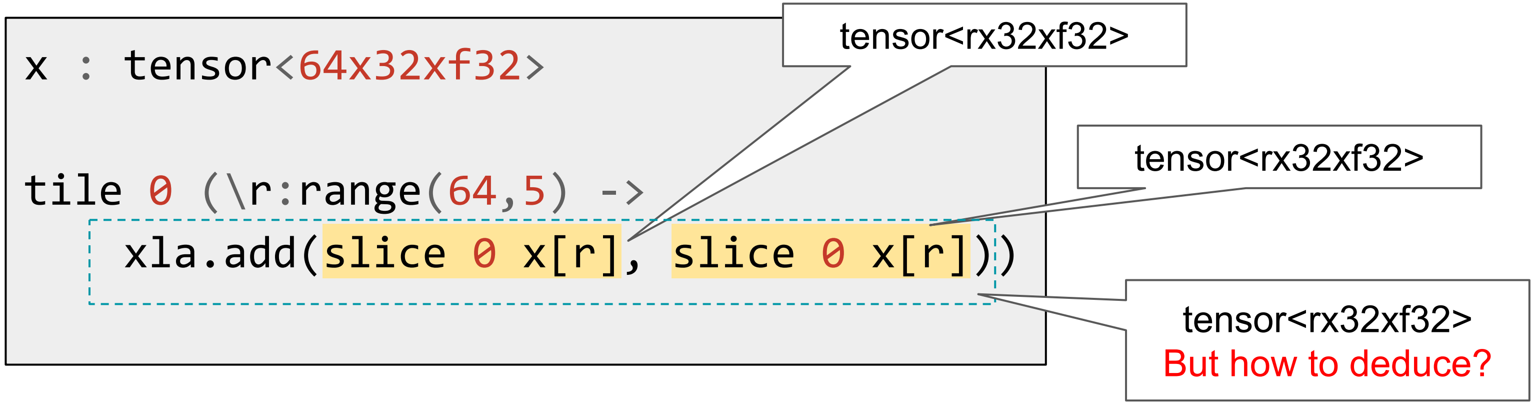
```
%y = tile d (%r : range<n,w>) { … stuff … ; yield %x }
```

```
y : tensor<32xnx16xf32>                x : tensor<32x%rx16xf32>
```

NOW

More expressive (dependent) types annoyance

- XLA dialect only has static shape: not dependent types on range values!
- How to type check a PartIR + XLA program with dependent types?
- Solution: specialization
  - range<32,5> has values with in-effect 2 widths: 5 and 2
  - type-check existing XLA ops for the two different assignments of widths
  - can lead to exponential blow-up for deeply nested tiles/sum, but we don't really expect a nesting level of > 2-3



```
x : tensor<64x32xf32>

tile 0 (\r:range(64,5) ->
   xla.add(slice 0 x[r], slice 0 x[r]))
```

An active research & engineering effort in DeepMind

- Testing strengths and weaknesses of PartIR on a set of mission-important models (XLA generated from Tensorflow or Jax high-level programs)
- Introduced “tiling specifications” for a substantial subset of of XLA ops
- Working through lowering and execution for TPU/GPU environments
- Started work on exposing the state of a partir program and an action space as a reinforcement-learning environment to tap onto the big pool of ML researchers in DeepMind. Example actions:
    - Application of a rewrite rule
    - Navigation
    - Effective inlining of a tile op inside another
    - Pushing tiling out
- Work towards approximate cost models for evaluating solutions (e.g. minimize data redistribution, subject to memory fit) and learnt cost models from data (Ithemal, [ICML’19])
- Identifying architectural modifications to MLIR pass and rewriting APIs needed for RL

What we do not know: how much declarative is enough?

- How to best express even higher-level abstractions, most notably: pipelining
- Input (“feature”) partitioning when we can’t exactly partition in the boundaries:
  - halo exchanges and convolutions require extra data to be communicated
- Is a tile-based IR on top of a tensor IR already too low level? Do we need higher-level combinators -- for instance “map over data” or “map over parameters” or “pipeline”?
- How to enhance numpy-style tensor programs with PartIR concepts:
  - Expose PartIR constructs directly, let people program with those (not portable code)
  - Numpy + interactive environment to partition a program (nice but also niche!)
  - Numpy + sharding annotations on variables (a bit better but limits library usability)
  - Numpy + sharding annotations on inputs and outputs only + ML/search for intermediates?

Thank you!

- We desperately need abstractions for partitioning in our compiler stacks
    - Types, Operators, Transformations
- PartIR offers a principled approach to partitioning via semantics preserving sequences of really simple transforms, rooted in deforestation and fusion ideas from declarative programming
- Great value for understanding the partitioning problem and having an executable semantics of partitioned programs
- But many things we do not know and need to learn; e.g. what has the HPC/MPI community been doing in this space?

The work is really just starting, keen to engage and collaborate!

Extra material

Notation convention: SSA vs. expressions

We use expression-based notation instead of SSA cause it's shorter.

```
tile d (\r:range<n,w> -> expr)
```

expr here can be thought of as the body of a region that yields a value.

NOTATION

```
tile 0 (\r:range(64,4) -> matmul(slice 0 x[r],y))
```

ACTUAL MLIR

```
tile 0 (%r:range(64,4)) {
   %0 = slice 0 %x[%r];
   %1 = matmul(%0, %y);
   yield %1 }
```