



MLIR : an Agile Infrastructure for Building a Compiler Ecosystem

LLVM Compiler Infrastructure in HPC @ SuperComputing 2020

Mehdi Amini, Google

Hi, I'm Mehdi Amini and I'd like to thank the organizer for inviting me for this talk. Today, I'll talk about MLIR, but beyond the core infrastructure that we implemented in the LLVM project, I'd like to push forward a vision for the next decade around the need for agility in compiler development, and the potential we have to build a strong ecosystem around the MLIR infrastructure, in the LLVM Project.

High-Performance Computing

Do we have a common definition for HPC? Some online definitions:

- *“the use of parallel processing for running advanced application programs efficiently, reliably and fast”*
- *“the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer in order to solve large problems in science, engineering, or business.”*
- Wikipedia redirects to “SuperComputer”? What about the “edge”?



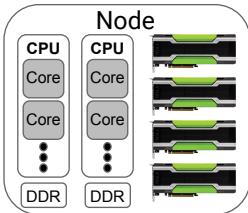
I'm glad to have the opportunity to look a bit more in HPC today. In order to prepare for this talk I wanted to refresh myself on the state of the art HPC environment, and looked up some definition of HPC. The first one describes it as “the use of parallel processing for running advanced application programs efficiently, reliably and fast”, this is my favorite one as it is fairly general, and we can include a lot of application space “at the edge”: in embedded environment or even in everyone's pocket with your smartphone.

Wikipedia is more old-school there and: “High Performance Computing” page redirects to “Supercomputer”.

Let's look into a “typical” HPC setup.

Your Typical HPC Setup

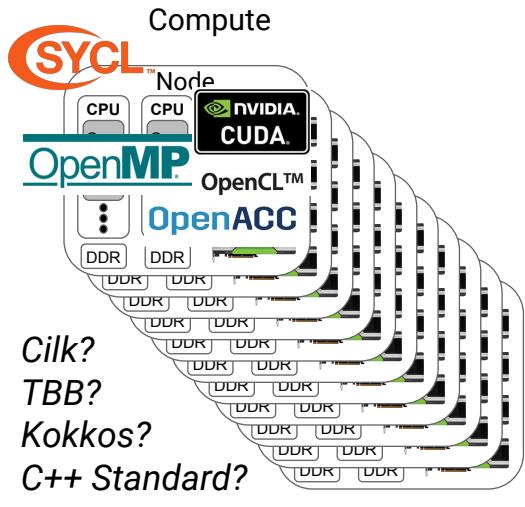
Compute



In general you start with a machine with many CPU cores, the amount varies: for example a single node in the current fastest supercomputer, the Japanese system Fugaku, has a single ARM CPU with 48 cores and 32 GB HBM. The former champion, IBM Summit has two 28 cores Power9 and 512 GB DDR.

An important change in the large decade is that the use of HW accelerators is now common, mostly has because general purpose GPUs are ubiquitous. For Summit it accounts for 6 GPUs per node and 96 GB extra HBM, coherent with the CPU. At this point, let's have a look at the commonly available programming abstractions.

Your Typical HPC Setup



Multi-core can be targeted by various APIs, a common one in HPC may be OpenMP: the programmer is given control over C/C++/Fortran programs mostly with directives expressed as pragmas instructing the compiler transformations which are fairly limited. The control is still in the hands of the programmer.

For GPUs, the de-facto programming model is Cuda. OpenCL is the Khronos counter-part to CUDA, intended to be more widely available, but it is likely not taking advantage of the most recent features of Nvidia GPUs the way CUDA does. Both of these solutions leaves fairly little room to the compiler in practice.

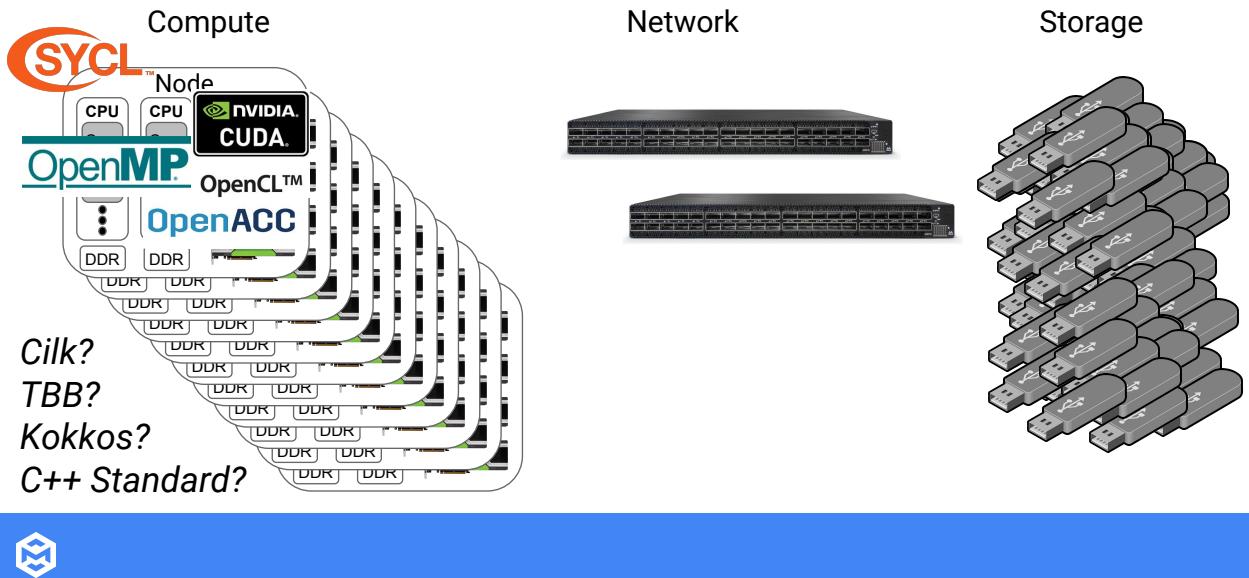
SYCL is a more recent Khronos standard, and can be seen as taking advantage of modern C++ feature to provide higher-level programming model for OpenCL. Intel is particularly involved with it and SYCL is present under the *Intel One API*. It remains a C++ language extension, where the role of the compiler is fairly limited.

OpenACC is a directive based approach like OpenMP, however it makes different tradeoffs and the programmer use the directives to instruct the compiler about properties of the program (a loop is parallel, buffers consumed and produced) and let more responsibility to the compiler to transform the program.

We can't be exhaustive here, there are also many library based approaches that are popular: Cilk, Thread Building Block, Kokkos, and even the C++ Standard since C++17! These all have limited compiler involvement though.

So this was all about a single node, that's already a lot to play with, but surely there is a limit to what you can do here. That's why you may want to scale this up!

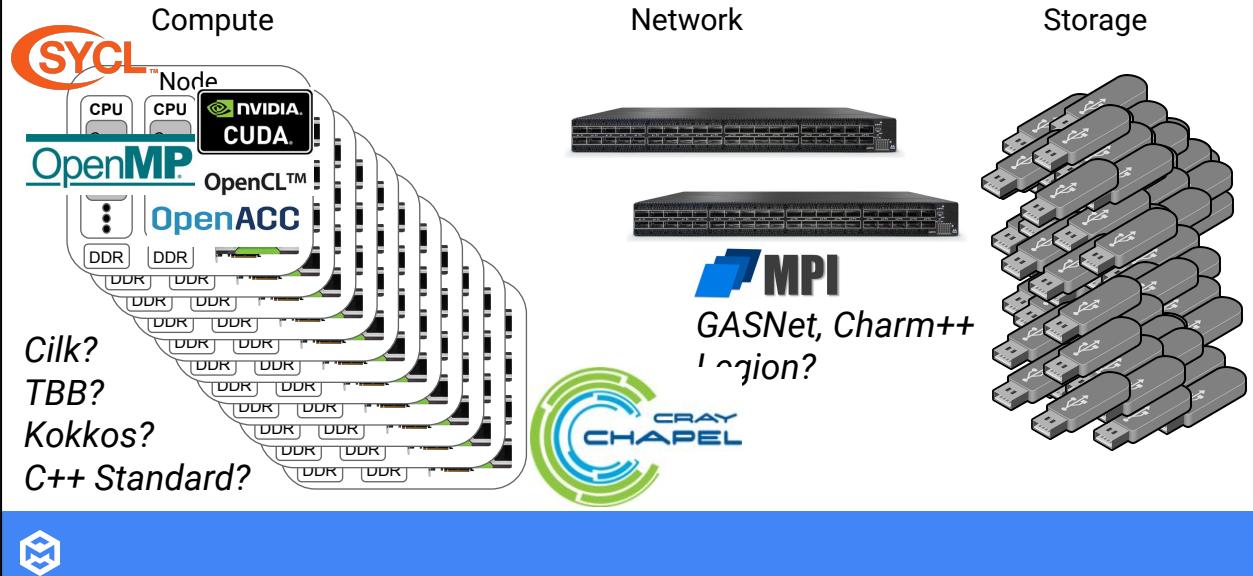
Your Typical HPC Setup



OK, now that we scaled up the machines, we need also some network, possibly something like Infiniband which is super fast and fancy (with things like RDMA).

Finally we won't get far without a lot of storage globally accessible from our nodes. Now that we have to go through the network, the programming model becomes more challenging.

Your Typical HPC Setup



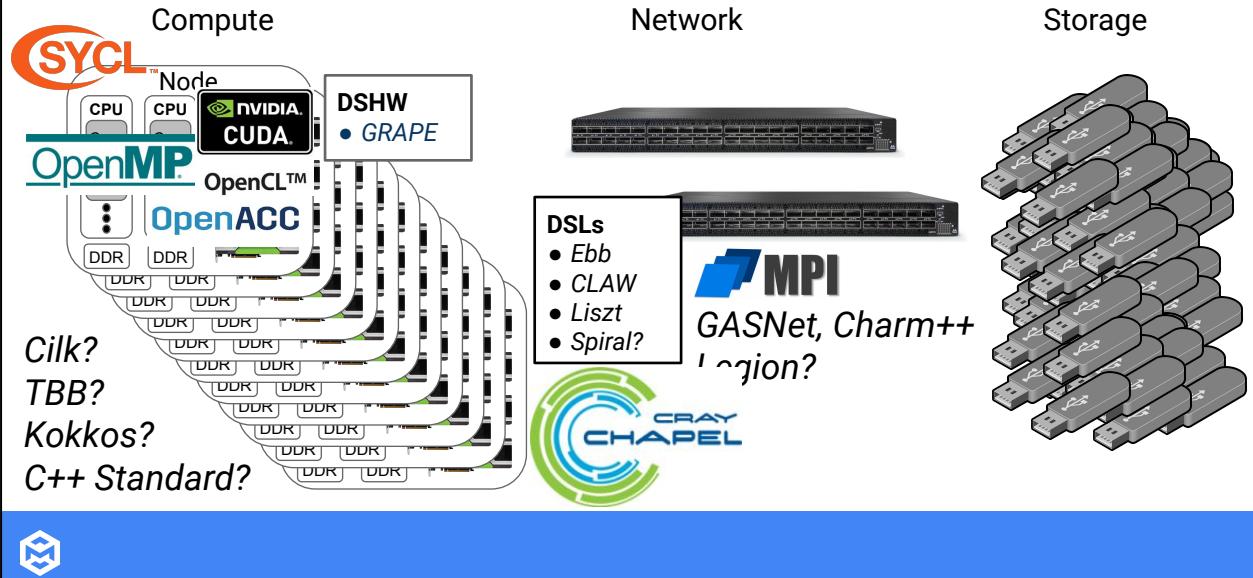
The “big fish” in this domain is still MPI. This is kind of the “assembly language” of distributed computing, but somehow it is still heavily used directly.

Some alternatives may be GASNet and Charm++, and at a higher level, the Legion runtime. These are all libraries approaches, and GASNet is used as a target by other high-level projects (language or frameworks).

One of them is Chapel: the last one of the PGAS still actively developed. This is also my favorite, but I am biased: I discovered Chapel with a full-day tutorial from Brad Chamberlain in person in 2009. I found the the kind of productivity boost a compiler can bring to be so amazing that I left my job manually writing OpenMP, Cuda and MPI and started a PhD on compilers.

A more standard approach may be coarray Fortran, which are now in the standard. But I’m less familiar with it.

Your Typical HPC Setup

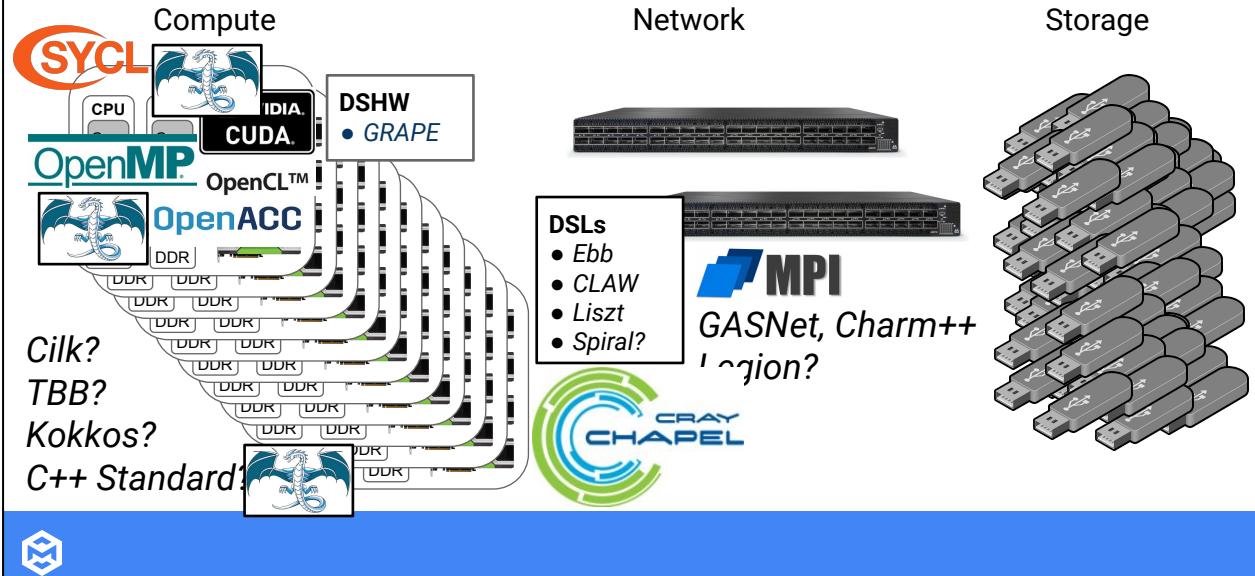


What about DSLs though? It seems like the perfect solution! Scientists express their problem in a programming model that captures the essence of their mental model and the compiler can optimize it at a high-level and applies various strategies to map it to the target system.

I suspect that DSLs unfortunately require a very large investment, as one not only need to design a solution tailored to a specific domain, but also need to build the entire toolchain all the way down to MPI (or GASNet) and CUDA. The barrier to entry is far too large: there are no abstractions that you can compose and reuse while writing your DSL compiler. We're coming a bit to the thesis behind this presentation: software libraries are composable and reusable, compilers abstractions aren't as easy to compose.

Finally in the accelerator domain, it seems the GPU have been so powerful that we aren't seeing domain specific accelerator for HPC: I haven't found much more references since the Gravity Pipe (GRAPE), which is an accelerator for gravitational model.

Your Typical HPC Setup

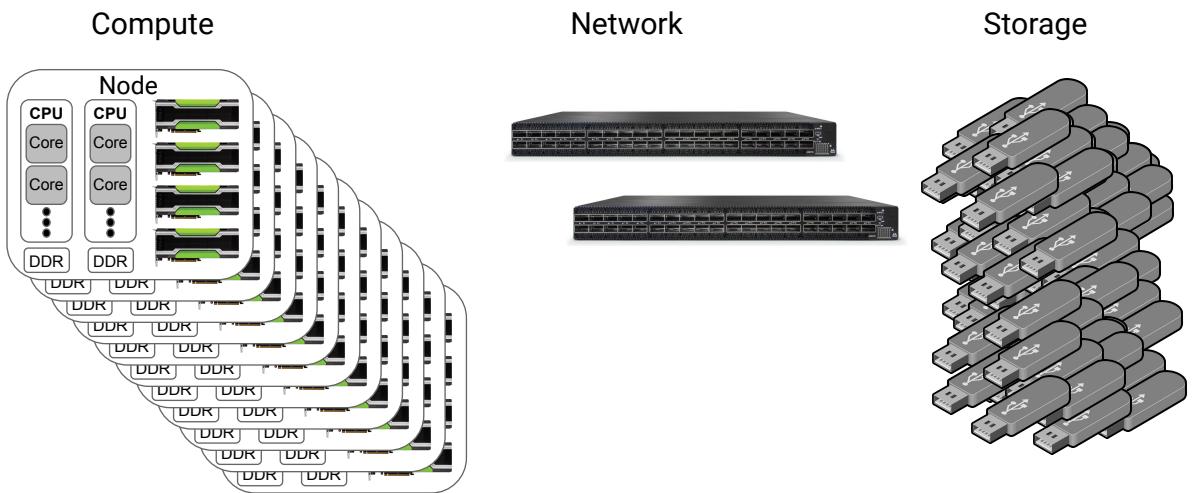


So what about the LLVM project and the LLVM community: it seems that we have a good solution for CPUs. LLVM IR is the common language here and has been successful as **the** compiler abstraction for targeting single-core CPU. LLVM has support for OpenMP and OpenCL in Clang, but these are mainly supported as language feature exposed to the programmer. The OpenMP IR Builder have been refactored from Clang into LLVM for the purpose of sharing these with Flang and MLIR, but this is still fairly limited.

Clang supports CUDA, but LLVM IR when used to target GPU only models the stream of execution of a single GPU “thread”.

Finally of course LLVM with libc++ is involved in the C++ standard library support for parallel primitives. So it seems as a community that there is a very large space for the compiler to be present and bring solutions. We believe MLIR is the way for the LLVM project to start building and offering the kind of reusable abstractions that are need for assembling a compiler in such a complex space.

Similarity and Contrast with Deep Learning



Let contrast the “traditional” HPC environment with what is happening in deep learning.

To begin with for some serious learning, you need compute, a lot of compute! Probably a very similar configuration our HPC cluster.

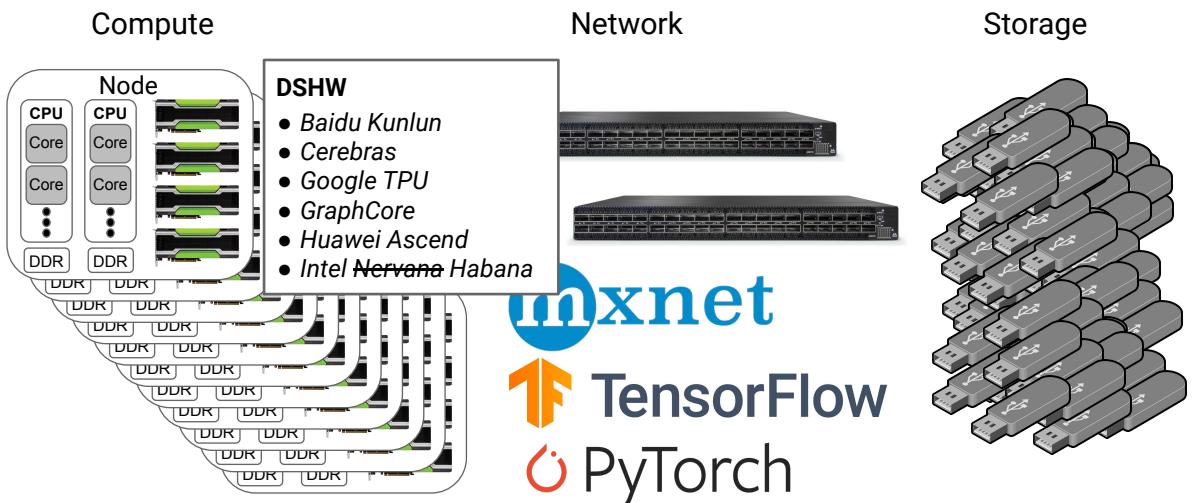
Then you need some superfast network: you need to feed these nodes with data and training involves a lot of communication as well!

Finally, you will likely want a very fast storage backend to keep your compute nodes busy with all the data to process.

All in all we have a very similar system. What is interesting is that deep learning is a much more recent field that really exploded the last 5 years. It does not have the baggage associated with HPC, it does not have the millions of line of Fortran libraries to carry over.

To some extent deep learning may give us some insights into how we may develop programming model and compilers for HPC if we started over from scratch today. Indeed in this domain it is all about DSLs.

Similarity and Contrast with Deep Learning



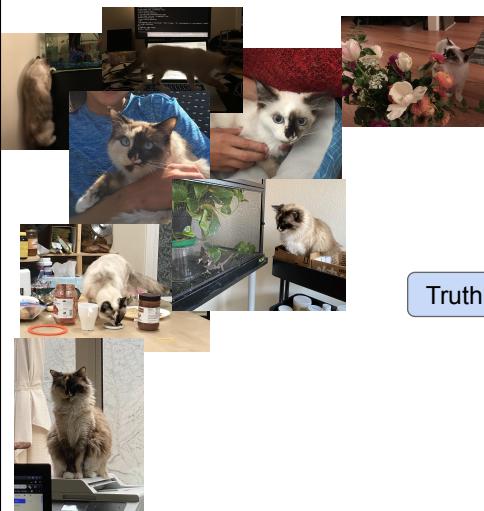
The big players from the last 5 years are MXNet, TensorFlow, and PyTorch. But there is a myriad of other frameworks out there.

A common theme though is to meet the scientists where they are: i.e. mostly in data science language like Python or Julia. There is virtually no one who would target a heterogeneous cluster with C++ or Fortran and MPI in order to train or deploy machine learning models.

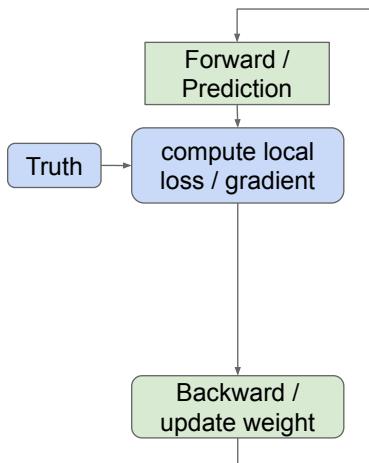
Another trend is the development of custom accelerators: the large companies like Intel and Google are present of course, but there is also a large number of startup in the field.

Let's look a little bit more into how deep learning training work and I'll zoom a bit into how Google TPUs are setup, and how the compiler is a center-piece to the scaling.

Distributed Deep Learning Training 101



Straightforward approach:
process one image at a time



Alright so we first need a task to learn, and preferably something really useful. In general this involves pictures of cats, and we need a lot of them to train our model.

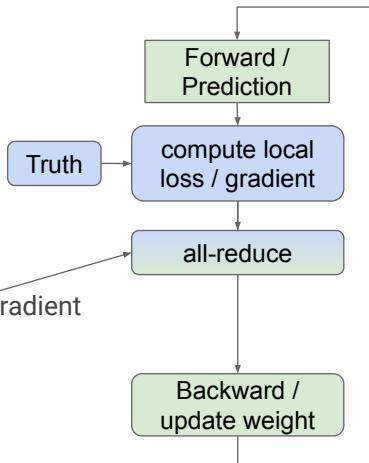
Then we'll run the forward pass for our model which will compute a prediction, in general very incorrect at the beginning. At this point we need to provide feedback to the system, in general with an external source of truth (manual labelling of the initial set of picture for example) and compute the loss, i.e. how much incorrect the prediction was. Then we process with the backward pass which adjust all the coefficient (or weights) involved.

Now this representation is a bit incorrect, even in theory we're not supposed to process images one at a time but all at once because some parts of the algorithm need to normalize for the dataset. In practice we perform "batched training".

Distributed Deep Learning Training 101



Batched process:
“Loop Vectorization” - N images per iteration

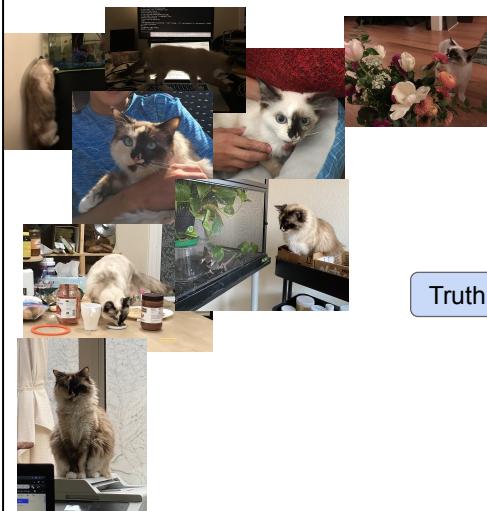


The batch process consist in “vectorizing” the loop, and taking the “average” of the evaluation of these iteration together. The forward pass also take advantage of the batch to perform some normalization across the “vector” of inputs.

In practice the size of this batch (or “minibatch”) is part of the hyperparameters of the training: the experts will manually tune this by trial and error. The ideal value can vary based on the model and the system.

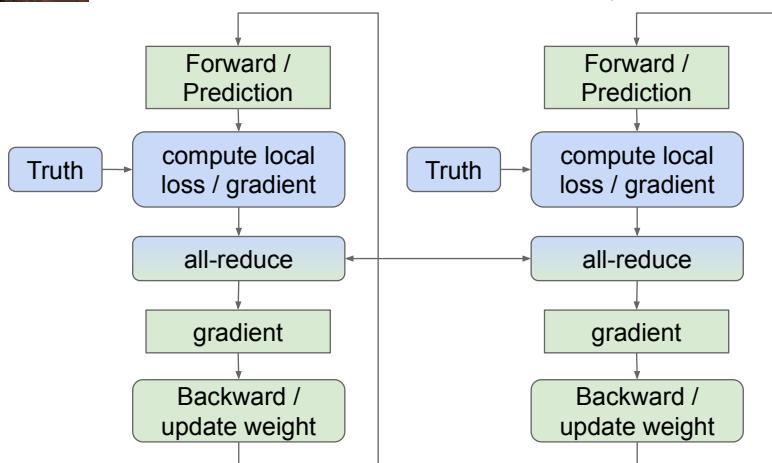
This is all good, not how do we scale this up to tens or hundreds of nodes?

Distributed Deep Learning Training 101



Multi-Batched process: N images per iteration x M nodes

Node 1



This is fairly simple, we just distribute different “minibatch” to different nodes, they can perform the prediction independently and communicate only for averaging the loss before applying the backward pass. Their weights always have the same value: this is synchronous training. We’re not gonna get into asynchronous training today, we’ll show how we map synchronous training to a distributed cluster in practice, using Google TPUs.

Google Tensor Processing Units (TPUv3)



420 TFLOPS, 128 GB HBM



TPU Pod: 1024 chips; 32x32 torus topology;
100+ PFLOPS



4096 chips; 128x32; mesh topology;
400+ PFLOPS

* **TPU operates on BF16 format**



<https://cacm.acm.org/magazines/2020/7/245702-a-domain-specific-supercomputer-for-training-deep-neural-networks/fulltext>

TPUs, or Tensor Processing Units are publicly available on Google Cloud in their second and third generation. A single tray in a machine has 4 chips / 8 cores and 128 GB HBM, for 420 TFLOPS, not counting the CPUs in the attached host.

TPUs are assembled in PODs, with 1024 chips / 2048 cores connected with a dedicated inter-connect network joining the TPUs in a 2D torus topology: no host is involved in the communications.

More recently, Google published how multiple pods can be chained together and showed scaling to 4 PODs, with over 400 PFLOPS available! Keep in mind that TPUs operate at peak on BF16 format.

TPUs are Supercomputers!

Rank	Name	RMax (PFlops)
1	Summit	148
2	Sierra	95
3	Sunway TaihuLight	93
4	Tianhe-2A	61
5	Frontera	24
6	Piz Daint	21
7	Trinity	20
8	AI Bridging Cloud Infrastructure	20

TPU Pod: 1024 chips; 32x32 torus topology;
100+ PFLOPS



4096 chips; 128x32; mesh topology;
400+ PFLOPS

* TPU operates on BF16 format

15

Source: <https://www.top500.org/lists/2019/11/>

Here's a list of top 10 supercomputers from the top500 supercomputer tracker. Although TPUs operate on BF16 format, the order of magnitude deserve the comparison.

Looking at this and at the amount of investment in supporting Deep Learning infrastructure: some people can see it as a new driving force for high performance computing.

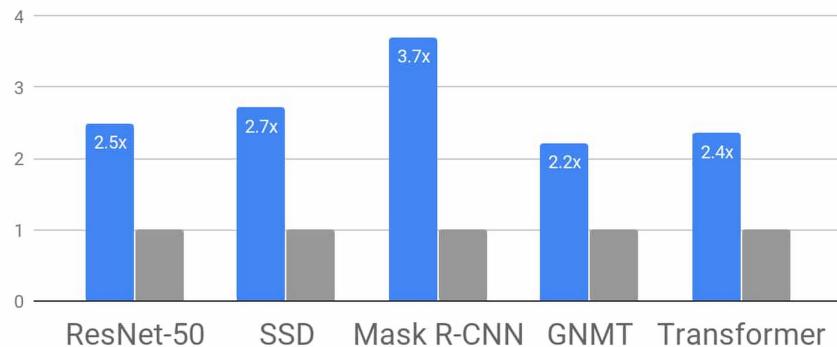
This list is from last year, but TPUv3 is also not the latest...

TPUs are Supercomputers!

TPU v4 Speedups over TPU v3

All comparisons at 64-chip scale

■ TPU v4 in MLPerf Training v0.7 ■ TPU v3 in MLPerf Training v0.6



<https://cloud.google.com/blog/products/ai-machine-learning/google-breaks-ai-performance-records-in-mlperf-with-worlds-fastest-training-supercomputer>

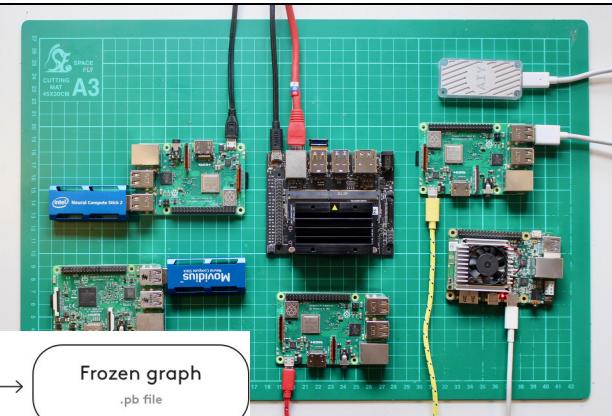
16

Just this spring in the most recent round of MLPerf, which is a benchmark suite and a competition where academics and industry are invited to submit their best score at training some models, Google announced their TPUs and show impressive improvements over a year!

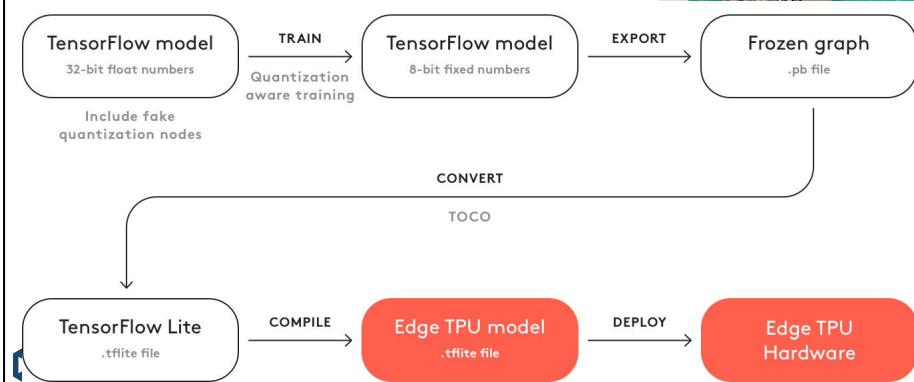
From Supercomputing to Embedded HPC

Highly specialized hardware

e.g. Google Edge TPU



Edge and embedded computing zoo



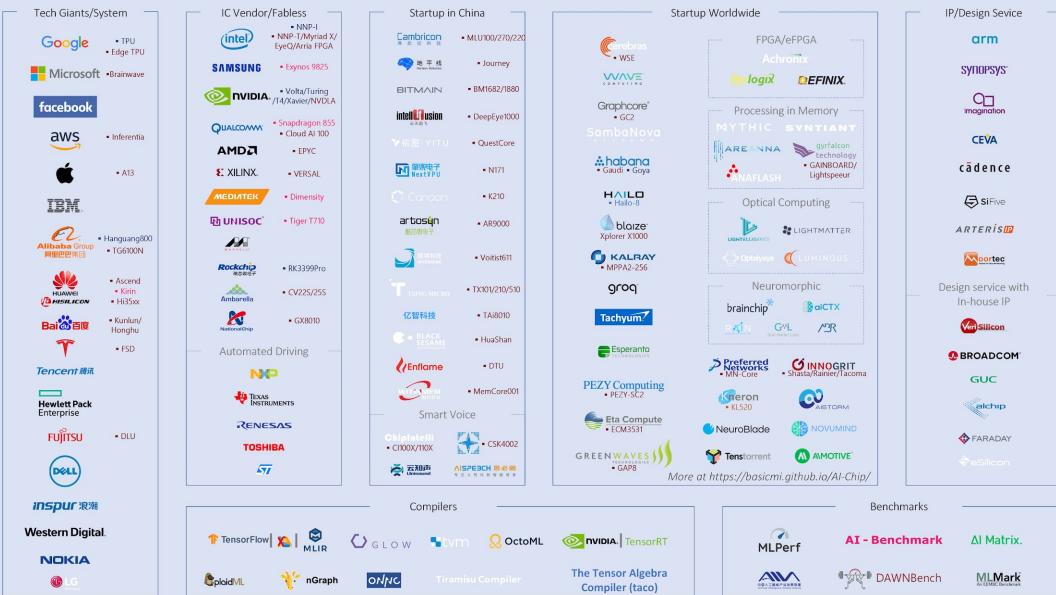
Machine learning is also very present also on the edge, most mobile vendors are including accelerator designed for deep learning workloads. Google produces the “Edge TPU” to this end.

• MLPerf results available • AI-Benchmark results available

AI Chip Landscape

V0.7 Dec., 2019

S.T.



All information contained within this infographic is gathered from the internet and periodically updated, no guarantee is given that the information provided is correct, complete, and up-to-date.



<https://basicmi.github.io/AI-Chip/>

Tensor Processing Units



- Under the hood:
 - Systolic arrays → allow high re-use of intermediate values
 - Parallel processing units, configured statically
 - 1970s computer architecture concept* *Kung, H.T. and Leiserson, C.E., 1979. Systolic arrays (for VLSI). In Sparse Matrix Proceedings 1978 (Vol. 1, pp. 256-282). Society for Industrial and Applied Mathematics.
 - Use them for the most fundamental linear algebra operation, namely, matrix multiplication
- See also [Cloud TPU: Codesigning Architecture and Infrastructure](#) (HotChips 2019)
- Problems: [lot of constraints\(alignment, padding, no lcache, etc.\)](#), [hard to program \(8-way VLIW\)](#), even more when managing multiple TPUs at once!

SOLUTION:



Deep learning involves a lot of linear algebra and in particular stresses the need for optimizing matrix multiplications. So without surprise, a TPU Core include a 128x128 matrix-multiply unit, with a not-revolutionary design since it borrow the systolic arrays concept from the 70s. The TPU also has independent vector and scalar units.

While very powerful, the TPU is difficult to program at a low-level and imposes many constraints around memory transferts to scratchpad, padding and alignment, VLIW packing, etc.

What's interesting is that from the beginning, the chosen path to address the programmatically challenge has been to rely on the compiler, and only expose only a high-level programming model for the TPU.

XLA: Accelerated Linear Algebra

- [XLA](#) HLO IR : High-level (mostly) linear algebra operations
 - Examples:
 - Dense linear algebra : matmul, dot, convolutions, cholesky
 - Control : While, Conditional
 - Data-ordering manipulations : reshape, transpose, sort
 - Sparse operations : gather, scatter
- Operations designed with deep learning in mind
- The XLA compiler represents these operations as a dataflow graph-based IR. Edges represent data (tensor) flow, nodes represent an operation.
- Input tensors are statically bounded-shape: the compiler computes an entire static memory layout.



More recent competitors: [Glow](#), [TVM](#).



XLA is the only way to target the TPU: it exposes a programming model relying on operators manipulating tensors (multi-dimensional array) that are assembled in a mostly-pure dataflow graph (communications primitives require some ordering).

The operators are general linear algebra operator, but with also many operator suitable for what we commonly find in deep learning. A design principle in general is to keep the operators orthogonal to each other.

A limitation of the IR is that every tensor has to be entirely statically shaped: this may seem overly restrictive but it also allows simplify the compiler and provide the ability to perform a lot of necessary optimization for TPU, in particular layout optimization to account for the padding and alignment constraint.

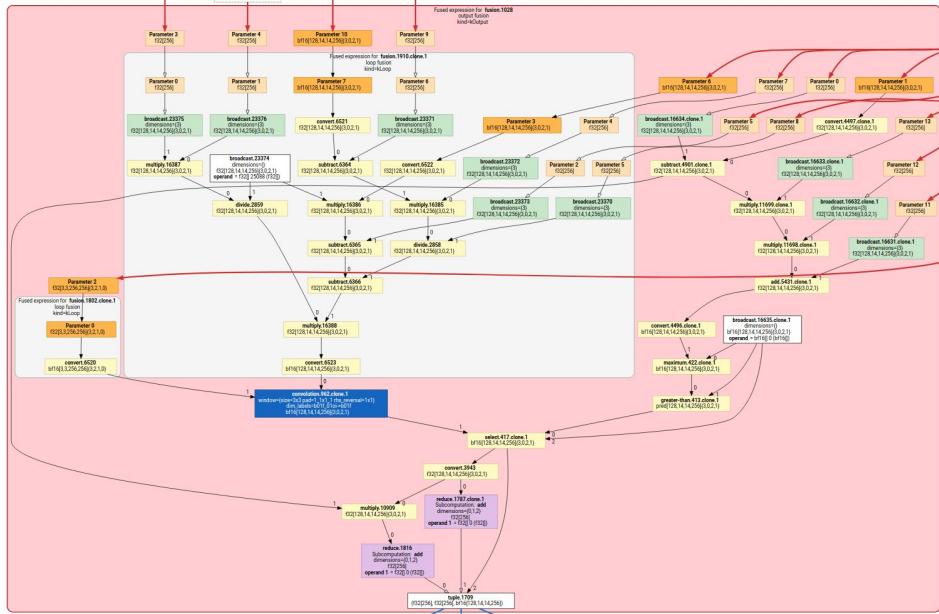
Ultimately the entire program is operating on statically laid out and memory bounded: there is no dynamic memory allocation involved in the execution of the program.

Because this is fundamentally a compiler technology, the operators are “fused” during codegen: even though in the dataflow graph a sequence of element-wise operation would appear as if there is a temporary array materialized between each operator.

XLA codegen can also make use of libraries, for example when targeting GPUs it will use cuDNN primitives when appropriate.

There are some competitors in this field, for example Glow and TVM. However XLA has some unique features.

XLA Example

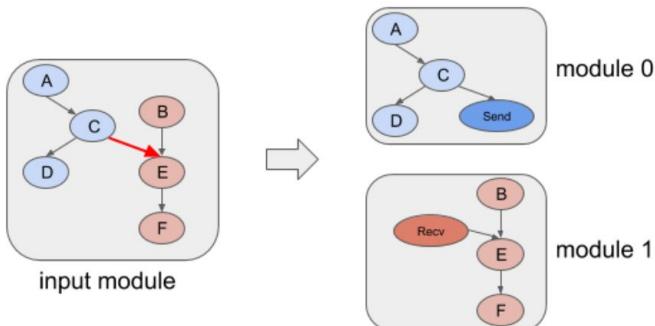


21

This is an example of a fusion of operator: a fusion of operators is handled as single unit by the codegen.

XLA Scaling: Multi TPUs

- Model Parallelism: multiple device can be represented in the graph with automatic partitioning and communication insertions.



Let's see how do we scale to make efficient use of our TPU Pods.

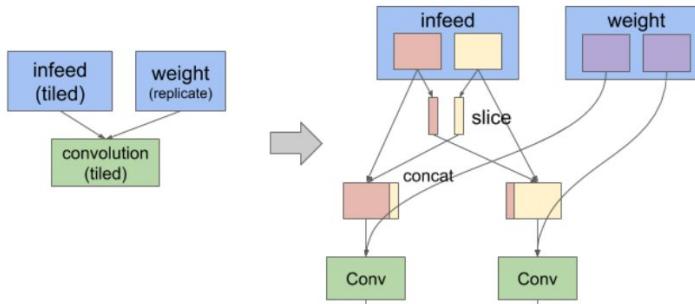
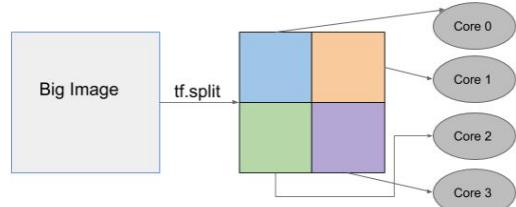
First in an XLA computation, nodes can be assigned to different devices. XLA will ultimately partition the graph and insert communication primitives. The implementation of these communication primitives depends on the target system, on TPU systems it'll involve DMA using the private inter-connect network. On Nvidia GPU it would likely involve NCCL.

This technique can also reduce the memory limit for the model by splitting it across two devices, making use of more HBM available.

XLA Scaling: Multi TPUs

Partitioning annotations can also be placed on the inputs, the compiler shard the computation accordingly:

```
tpu_config=tpu_config.TPUConfig(  
    iterations_per_loop=100,  
    num_cores_per_replica=4,  
    input_partition_dims=[[1, 2, 2, 1], None]])
```



23

Another way to use multiple TPUs for a single computation is *spatial partitioning*: the user only has to indicate how to shard the input and the compiler will take care of the rest.

For example here is a TensorFlow API to target TPUs: the user indicates that they would like to use 4 TPU cores per replica for their models. They also specify how to shard the dimension of the input. For example here the input images will be split in 4 pieces and distributed to the 4 TPUs.

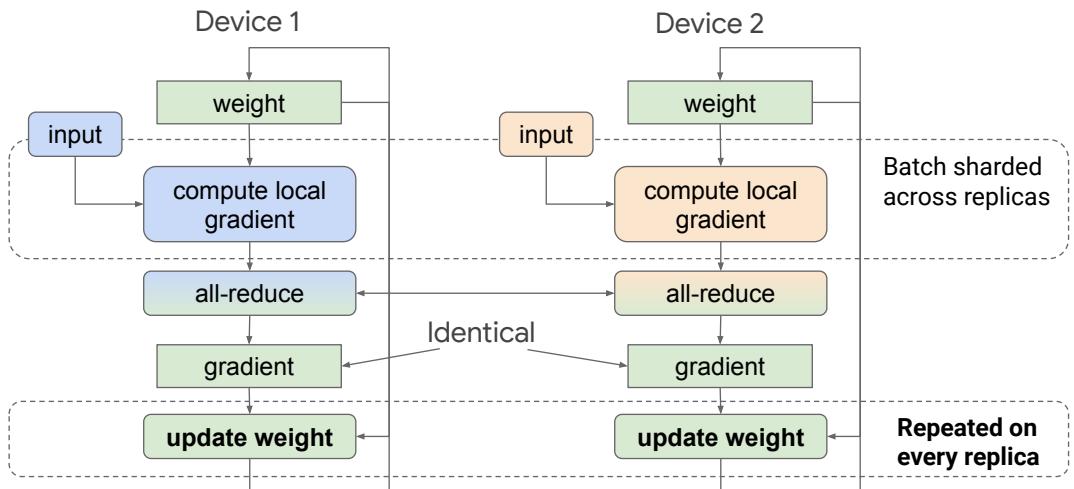
The compiler is then responsible to manage the partitioning of the graph, managing communication of the halos or redundant computations as needed.

This can improve the performance, but also again it also increases the amount of HBM memory available.

The user interface is minimal and it is all automated by the compiler.

XLA Scaling: Multi TPUs

Weight Update Sharding: eliminate redundancy across device in backprop



Another optimization implemented in XLA is “weight update sharding”.

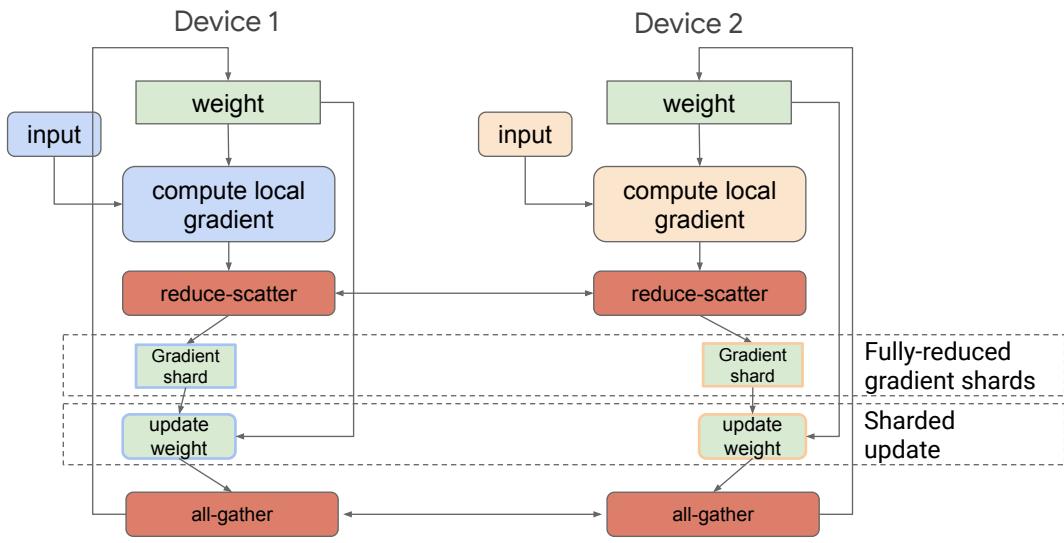
Each node processes different data, compute the local gradient, which are then averaged across all nodes before being back-propagated on each node.

Note that after averaging the gradient, the computation is identical on every nodes.

This can be a large part of the process, up to 45% of the whole time on some models!
XLA can recognize this situation and automatically shard this computation, including adding the extra communication.

XLA Scaling: Multi TPUs

Weight Update Sharding: eliminate redundancy across device in backprop



Instead of averaging the gradients so that every node has the entire copy, the nodes will only get a shard of the gradient, for example here only half, and perform the backpropagation on this shard and actually communicate their shard of the update weight to the other nodes.

We're trading off an extra communication for much less compute and potentially memory.

XLA and TPU: More Resources

[XLA: Accelerated Linear Algebra](#)

[Automated GPU Kernel Fusion with XLA](#)

[Scale MLPerf-0.6 models on Google TPU-v3 Pods](#)

[GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism](#)

[GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding](#)

[Automatic Cross-Replica Sharding of Weight Update in Data-Parallel Training](#)

[Mesh-TensorFlow: Deep Learning for Supercomputers.](#)

JAX is Autograd and XLA, brought together for high-performance machine learning research.



<https://github.com/google/jax>



26

Here are some references if you're interested in getting more depth into this topic.

Also, a rising star in the domain of Deep Learning framework is JAX: this is a project coming out of Google research that started as a direct thin wrapper on top of XLA. If you want to play with XLA capability without all the layers of complexity that comes with a large project like TensorFlow, JAX is a very elegant solution.

So I showed a sample of the capabilities of XLA, showcasing the kind of things that can be achieved by compiler co-designed with a HPC system. A key point is that we achieve very advanced optimizations which allow Machine Learning scientists to stay in Python, Julia, Swift, or any high-level language and never have to see any Fortran or MPI. Optimization like weight-update sharding are not for everyone to be implemented manually either.

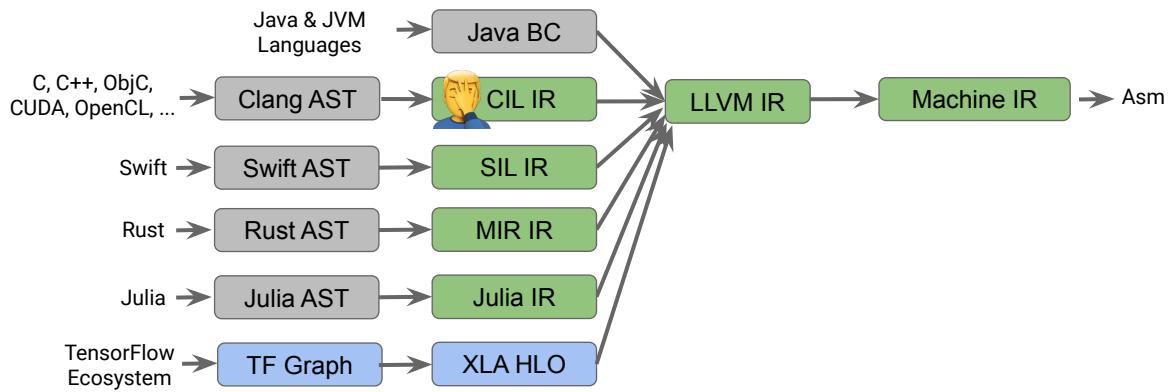
Yet, these users manage to make use of a supercomputer? Can this be the future for HPC in general? What kind of compiler capabilities do we need to get there?

MLIR Genesis



Alright, let me come back to the idea behind MLIR first and why it may be a game changer for the LLVM community.

From Programming Languages to the TensorFlow Compiler



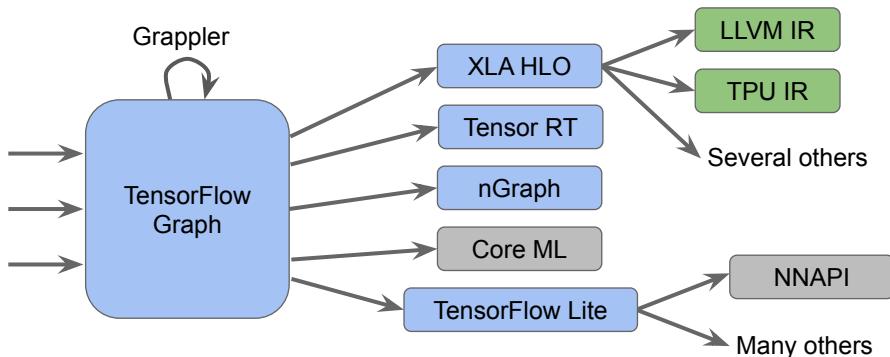
- Domain specific optimizations, progressive lowering
- Common LLVM platform for mid/low-level optimizing compilation in SSA form



LLVM managed to achieve the “hourglass” model of providing a unified target for CPU. However modern languages also redefine their own IR, for example optimizing the Swift refcounting is much easier at the SIL level where you can capture the high-level semantics. Similarly Rust borrow-checker would be difficult to implement in LLVM, and Rust has its own IR (MIR) that enables this.

Many frameworks in the machine learning world are targeting LLVM. They are effectively defining higher level IRs in the tensor domain, and lowering to LLVM for CPUs and GPUs. This is structurally the same thing as any other language frontend.

The TensorFlow Compiler Ecosystem



Many “Graph” IRs, each with challenges:

- Similar-but-different proprietary technologies: not going away anytime soon
- Fragile, poor UI when failures happen: e.g. poor/no location info, or even crashes
- Duplication of infrastructure at all levels



Zooming on the TensorFlow ecosystem, at the top is the XLA path that we talked about extensively. However TensorFlow supports many other systems. Most of them are fairly similar conceptually and all these arrows are complicated “bridges” that try to integrate these projects together. They rarely lead to a good user experience though, they are fragile, rarely complete, and hard to maintain.

In general there is poor reuse and a lot of redundancy across all these projects.

MLIR: A toolkit for representing and transforming “code”

Represent and transform IR 

Represent [Multiple Levels](#) of [IR](#) at the same time

- tree-based IRs (ASTs)
- data-flow graph IRs (TF Graph, SSA)
- control-flow graph IRs (TF Graph, SSA)
- target-specific parallelism (CPU, GPU, TPU)
- machine instructions

While enabling

Common compiler infrastructure

- location tracking
- richer type system(s)
- common set of conversion passes
- LLVM-inspired infrastructure

And much more



MLIR is at its Core a generic infrastructure for representing and transforming “code”. It provides a framework to create an IR and manipulate it. The project is heavily inspired by the LLVM infrastructure and engineering practices in general. Since MLIR allows to create new IRs, it also provides facilities for multiple IRs to cohabitiate together and a framework for converting one to another, or a mix of others.

MLIR Story

1. The right abstraction at the right time
2. Progressive conversion and lowering
3. Extend and reuse
4. Industry standard



We listen & learn as we go

AMD

arm

cerebras

Google

GRAPHCORE

habana

intel

MEDIATEK

NVIDIA

Qualcomm

SambaNova
SYSTEMS

SAMSUNG

mi

XILINX



The idea is that this capability can be leveraged to easily add new abstractions. This incentivizes the compiler engineers to favor very progressive lowering of the abstraction level, which is convenient in terms of design and testing of the compiler components, but also maximizes the reuse.

This approach has been successful so far, and convinced enough partners in the industry that the best place for MLIR governance and ensuring a good collaboration was the LLVM project.

MLIR: Under the hood



Let's see quickly what is under the hood and explore the basic principles of MLIR.

MLIR Core Concepts

Very few core-defined aspects, MLIR is generic and favor extensibility:

- **Region:** a list of basic blocks chained through their terminators to form a CFG.
- **Block:** a sequential list of Operations. They take arguments instead of using phi nodes.
- **Operation:** a generic single unit of “code”.
 - takes individual Values as operands,
 - produces one or more SSA Values as results.
 - A terminator operation also has a list of successors blocks, as well as arguments matching the blocks.

There aren't any hard-coded structures or specific operations in MLIR:

even Module and Function are defined just as regular operations!



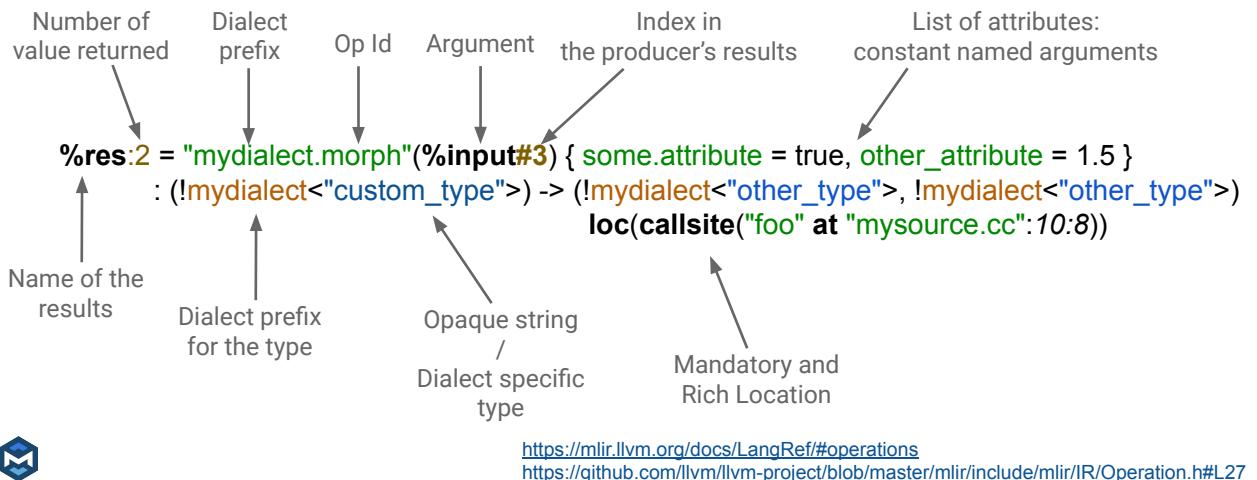
MLIR core concepts are fairly simple. The IR is organized around three main data structure:

- Region: ...
- Block: ...
- Operation: ...

The important part to remember is that there aren't any hard-coded structure or operations in MLIR. Even the top-level Module and the definitions of Function are just modeled as any other operation.

Operations, Not Instructions

- No predefined set of instructions
- Operations are like “opaque functions” to MLIR



In MLIR, everything is about Operations, not Instructions: we put the emphasis to distinguish from the LLVM view. In LLVM you have a fixed list of instructions, which all are defined with their own class which defines members and storage. It isn't the case in MLIR: there is one opaque C++ class and it defines the storage in a generic way for any possible operation.

Operations can be coarse grain (perform a matrix-multiplication, or launch a remote RPC task) or can directly carry loop nest or other kind of nested “regions”, we'll show some examples later.

Let's starts with the anatomy of an operation.

What you see on the screen with a lot of color is the generic assembly format for MLIR. Just like LLVM has a textual output, any MLIR operation can be represented in this generic format. This makes serialization and deserialization really simple.

So what are the elements that define an operation? There is an isomorphic relation between the in-memory representation and the generic format, let me walk you through this.

First, what uniquely identify an operation is its name, you have the operation ID, prefixed by the dialect name. Together this provides a unique name for an operation.

An operation produces SSA results. An LLVM instruction produces only one SSA value at most, in MLIR they can generate many. This operation for example defines 2 SSA value as results. The textual IR here uses a single name for the SSA value, and an index to differentiate the two values.

In parentheses, you have the list of operands for the operation. This is a comma-separated list of SSA values. You can see here the name of the SSA value but also an optional index. Here we'll use the fourth result of the operation producing the "%input" result.

After the list of operands is a dictionary of *Attributes*, which can be seen as extras operands with are restricted to be constant literal values, they can't refer to other SSA value.

On the second line is the type of the operation. We're using a functional notation, so after the colon you have the types of the operands in parenthesis.

The type after the bang is the name of a dialect, followed in angle brackets by the custom serialization of the type defined by the dialect, it is opaque to MLIR. After the arrow is the type for the results, here this operation defines two results so we have two types.

Finally on the last line is the location for the operation. We often elide it from the debug print, but it is always present in memory. Locations are rich: here we can represent that it corresponds to a particular call site of a function at a given place in the source.

Alright this what an Operation is made for, and something to keep in mind is that when you define an operation you really can't add more state or storage to an operation. When you define an operation in MLIR you just actually put restriction on what is valid for the operation: for example can it return a result? Multiple? What are the restrictions on the types? What attributes are allowed? Actually there is one more thing though, let's look at regions.

Recursive nesting: Operations -> Regions -> Blocks

```
%results:2 = "d.operation"(%arg0, %arg1) ({
    // Regions belong to Ops and can have multiple blocks.
    ^block(%argument: !d.type):
        %value = "nested.operation"() ({
            // Ops can contain nested regions.
            "d.op"(): () -> ()
        }) : () -> (!d.other_type)
        "consume.value"(%value) : (!d.other_type) -> ()
    ^other_block:
        | "d.terminator"() [^block(%argument : !d.type)] : ()
    -> ()
}) : () -> (!d.type, !d.other_type)
```

The diagram illustrates the recursive nesting of regions and blocks. It shows three nested scopes represented by light blue boxes. The outermost box is labeled 'Region', the middle box is labeled 'Block', and the innermost box is also labeled 'Region'. This visualizes how operations can contain multiple regions, which in turn contain basic blocks.

- Regions are list of basic blocks nested inside of an operation.
 - Basic blocks are a list of operations: **the IR structure is recursively nested!**
- Conceptually similar to function call, but can reference SSA values defined outside.
- SSA values defined inside don't escape.

<https://mlir.llvm.org/docs/Tutorials/UnderstandingTheIRStructure/>
<https://mlir.llvm.org/docs/LangRef/#high-level-structure>



On top of the previous introduced element, another important property of an operation is that it can hold a list of “region”. The concept of region does not have an equivalent in LLVM IR. The best analogy is to look at LLVM functions, these are first class structure in LLVM which hold a body in the form of a CFG. The CFG is a control flow graph which is hold as chained list of basic blocks. In MLIR, everything is an operation: even a function is an operation. Operations optionally have one or multiple regions attached, and a region is nothing else than a list of blocks which may represent a CFG. This is how functions are modelled in MLIR: an operation with a region that models the body of the function.

Since a region is a list of basic blocks, which themselves are a list of operations: the structure is recursively nested! This is a whole new dimension in the IR which opens up design possibilities. Regions are commonly used in MLIR and very powerful to express the structure of the IR, we'll come back to this with multiple examples. And with this simple structure you can understand almost everything in MLIR.

Dialects: Defining Rules and Semantics for the IR

A MLIR dialect is a logical grouping including:

- A prefix (“namespace” reservation)
- A list of custom types, each its C++ class.
- A list of operations, each its name and C++ class implementation:
 - Verifier for operation invariants (e.g. `toy.print` must have a single operand)
 - Semantics (has-no-side-effects, constant-folding, CSE-allowed,)
- Passes: analysis, transformations, and dialect conversions.
- Possibly custom parser and assembly printer

<https://mlir.llvm.org/docs/LangRef/#dialects>

<https://github.com/llvm/llvm-project/blob/master/include/mlir/IR/Dialect.h#L37>

<https://mlir.llvm.org/docs/Tutorials/CreatingADialect/>



You will hear a lot about “Dialects” in the MLIR ecosystem. A Dialect is a bit like a C++ library: it is at minima a namespace where you can group a set of types, a set of operations that operate on these types (or types defined by other dialects), and a set of custom attributes. So just like a C++ library where you define classes, methods, etc. The Dialect is your own IR library: by defining this set of types/attributes/operations you can define a closed set that has a well defined semantics that you can manipulate.

A dialect is loaded inside the `MLIRContext` and extends MLIR using various hooks, like for example to the IR verifier: it will enforce invariants on the IR (just like the LLVM verifier).

Dialects are cheap abstraction: you create one like you create a new C++ library. There are roughly 20 dialects that come bundled with MLIR, but many more have been defined by MLIR users: our internal users at Google have defined over 60 so far!

Something else that is important to know before looking at examples of MLIR, is that the IR does not always look like the generic format we’ve seen previously. This is because Dialect authors can also customize the printing/parsing of Operations and Types to make the IR more readable. Dialect IR are more like DSLs: you may need to read the documentation to interpret them correctly. You can always disable the custom printing and have a generic print of the IR though.

Example: Affine Dialect

```
func @test() {
    affine.for %k = 0 to 10 {
        affine.for %l = 0 to 10 {
            affine.if (d0) : (8*d0 - 4 >= 0, -8*d0 + 7 >= 0)(%k) {
                // Dead code, because no multiple of 8 lies between 4 and 7.
                "foo"(%k) : (index) -> ()
            }
        }
    }
    return
}

#set0 = (d0) : (d0 * 8 - 4 >= 0, d0 * -8 + 7 >= 0)
func @test() {
    "affine.for"() {lower_bound: #map0, step: 1 : index, upper_bound: #map1} : () -> () {
        ^bb1(%i0: index):
        "affine.for"() {lower_bound: #map0, step: 1 : index, upper_bound: #map1} : () -> ()
    {
        ^bb2(%i1: index):
        "affine.if"(%i0) {condition: #set0} : (index) -> () {
            "foo"(%i0) : (index) -> ()
            "affine.terminator"() : () -> ()
        } { // else block
        }
        "affine.terminator"() : () -> ()
    }
    ...
}
```

With custom parsing/printing: affine.for operations with an attached region feels like a regular for!

Extra semantics constraints in this dialect: the if condition is an affine relationship on the enclosing loop indices.

Same code without custom parsing/printing:
isomorphic to the internal in-memory representation.

<https://mlir.llvm.org/docs/Dialects/Affine/>



Here is an example of nice syntax and advanced semantics modelling at the same using regions is shown here with the Affine Dialect.

The affine dialect is modeling polyhedral loop nests (and a bit more), we see that here you have a function with nested loops and inside the innermost loop you have a conditional with some sort of linear equation describing the condition. This is important for polyhedral tools because it ensures that the loop nest can be analyzed and transformed within a mathematical framework for correctness.

<walkthrough the IR modeling>

This affine.for loops are pretty and readable, but the generic form really show the actual implementation.

<walkthrough the IR modeling>

LLVM as a dialect

```
%13 = llvm.alloca %arg0 x !llvm.double : (!llvm.i32) -> !llvm.ptr<double>
%14 = llvm.getelementptr %13[%arg0, %arg0]
      : (!llvm.ptr<double>, !llvm.i32, !llvm.i32) -> !llvm.ptr<double>
%15 = llvm.load %14 : !llvm.ptr<double>
llvm.store %15, %13 : !llvm.ptr<double>
%16 = llvm.bitcast %13 : !llvm.ptr<double> to !llvm.ptr<i64>
%17 = llvm.call @foo(%arg0) : (!llvm.i32) -> !llvm.struct<(i32, double, i32)>
%18 = llvm.extractvalue %17[0] : !llvm.struct<(i32, double, i32)>
%19 = llvm.insertvalue %18, %17[2] : !llvm.struct<(i32, double, i32)>
%20 = llvm.constant(@foo : (!llvm.i32) -> !llvm.struct<(i32, double, i32)>) :
      !llvm.ptr<func<struct<i32, double, i32> (i32)>>
%21 = llvm.call %20(%arg0) : (!llvm.i32) -> !llvm.struct<(i32, double, i32)>
```



More intro to MLIR: <https://mlir.llvm.org/docs/Tutorials/Tutorial/>

Another example of a dialect with a custom printer is the LLVM IR itself. Indeed the LLVM IR can be modeled as a dialect, and actually is implemented in MLIR! You'll find the LLVM instructions and types, prefixed with the `llvm.` dialect namespace.

The LLVM dialect isn't feature-complete (inline assembly, block addresses, ...), but defines enough of LLVM to support the common need of DSL-oriented codegen. There are also some minor deviation from LLVM IR: for example because of MLIR structure, *constants* aren't special and are instead modeled as regular operations.

For more details into the MLIR infrastructure, feel free to lookup the website for documentation, and in particular the Toy tutorial which can walk you through a practical example.

MLIR: an Ecosystem



Alright so that was the basics of the MLIR infrastructure. But while the infrastructure alone is already a boost to get started writing a compiler, a large part of the value proposal here is the vision of the ecosystem we can grow in MLIR.

Changing the paradigm for compiler design

- Software libraries are reusable, composable, ...
=> software development is agile!
- Can we have compiler IR and abstractions that are easily reusable and composable?
=> MLIR Dialects can make **heterogeneous compiler development agile!**
No silver bullet:
 - composable is never perfect, assembling an entire toolchain is still **work**,
 - But just like assembling a large project by reusing libraries is!



In this section I'd like to bring back the parallel between how software development is agile: you can reuse other people's libraries and compose them, and how this is missing in compiler design. The idea is that MLIR Dialects may be getting us closer to have this capability for IR design. In particular for heterogeneous compilers where the paradigms are various and we can't come up with a single IR like LLVM achieved on CPU, we need to be agile and have flexibility.

This isn't a silver bullet though: assembling a toolchain like XLA for a heterogeneous system in a particular domain is still intrinsically a lot of work. But just like the availability of libraries like boost aren't making software development trivial either. In this section I'd like to talk about these compiler IR abstractions, and develop a narrative that would surface the value there is to have all of these composable as needed in MLIR.

Example: Affine Dialect for Polyhedral Compilation

```
func @test() {
  affine.for %k = 0 to 10 {
    affine.for %l = 0 to 10 {
      affine.if (d0) : (d0 - 1 >= 0, -d0 + 8 >= 0)(%k) {
        // Call foo except on the first and last iteration of %k
        "foo"(%k) : (index) -> ()
      }
    }
  }
  return
}
```

<https://mlir.llvm.org/docs/Dialects/Affine/>



The first abstraction is one I mentioned before: the Affine dialect opens the door to polyhedral optimization. This can be a very powerful tool to have at hand when your problem can fit the framework.

Example: Affine Dialect for Polyhedral Compilation

...AND WHY IT WAS WORTH IT

MLIR is a community, providing a shared understanding and implementation of infrastructure of legible pieces built to be integrated

The MLIR community can benefit from what we develop (and hopefully improve it!): affine.parallel, autotiling, aliasing analysis, llvm.atomic_rmw, ...

And we can benefit from what others have developed: LLVM (dialect & emitter), CSE, SPIR-V, canonicalization, ...

All developed as MLIR dialects, providing clear and legible boundaries for integration.



Using PlaidML for Affine Parallel Optimizations in MLIR (Intel) C4ML Workshop (In conjunction with CGO 2020)



This abstraction has already been leveraged and adopted, for example at Intel who presented their early experience with MLIR and the affine dialect during the “Compiler for Machine Learning” Workshop earlier this year.

The affine dialect, and the in-tree path to LLVM can boost not only the development of such tools, but also compiler research in this domain.

Example: The Tensor Linear Algebra Compiler (TACO)

<http://tensor-compiler.org/index.html>

Particularly interesting for its flexibility in sparse code generation.

Current collaboration to reimplement it in MLIR!

<https://llvm.discourse.group/t/sparse-tensors/2020>

```
1 Format csr({Dense,Sparse});
2 Tensor<double> A({{64,42}}, csr);
3
4 Format csf({Sparse,Sparse,Sparse});
5 Tensor<double> B({{64,42,512}}, csf);
6
7 Format svec({Sparse});
8 Tensor<double> c({512}), svec;
9
10 B.insert({0,0,0}, 1.0);
11 B.insert({1,2,0}, 2.0);
12 B.insert({1,2,1}, 3.0);
13 B.pack();
14
15 c.insert({0}, 4.0);
16 c.insert({1}, 5.0);
17 c.pack();
18
19 IndexVal i, j, k;
20 A(i,j) = B(i,j,k) * c(k);
21
22 A.compile();
23 A.assemble();
24 A.compute();
```

Fig. 12. Computing tensor-times-vector with the taco C++ library.

```
$taco "A(i,j) = B(i,j,k) * c(k)" -f=A:ds -f=B:sss -f=c:s
// ...
int pA2 = A2_pos[0];
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
    int i = c1_idx[pB1];
    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1+1]; pB2++) {
        int j = B2_idx[pB2];
        double tk = 0.0;
        int pB3 = B3_pos[pB2];
        int pc1 = c1_pos[0];
        while ((pB3 < B3_pos[pB2+1]) && (pc1 < c1_pos[1])) {
            int kB = B3_idx[pB3];
            int kc = c1_idx[pc1];
            int tk = min(kB, kc);
            if (kB == k && kc == k) {
                tk += B_vals[pB3] * c_vals[pc1];
            }
            if (kB == k) pB3++;
            if (kc == k) pc1++;
        }
        A_vals[pA2] = tk;
        pA2++;
    }
}
```

Fig. 13. Using the taco command-line tool to generate C code that computes tensor-times-vector. The output of the command-line tool is shown after the first line. Code to initialize tensors is elided.

*Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe.
The tensor algebra compiler. Proc. ACM Program. Lang. 1, OOPSLA, Article 77 (October 2017)*



Another example of compiler abstraction is what has been demonstrated by the TACO compiler, and in particular in the domain of codegen for sparse linear algebra. TACO is a fantastic standalone tool, but it is likely not straightforward to integrate and reuse in your project. For example if we were to add support to sparse code generation to a project like XLA, using TACO would probably be through rigid interface built for the purpose of the integration. It isn't clear if the current implementation of TACO would fit in the deployment flow of XLA either. These hurdles in general lead to reimplementing custom solutions from scratch. Luckily here, my colleague Aart is currently bringing TACO's ideas into MLIR! That means that our ecosystem is growing with two different abstractions, for polyhedral codegen and for sparse codegen, in an infrastructure intended for making them compose together in the same project.

Example: Heterogeneous Compiler IR



We mentioned before that HPC is frequently heterogeneous nowadays, in particular GPUs are ubiquitous. If I start a DSL compiler to support HPC users I likely want solid abstraction to target accelerators.

Unified Accelerator and Host Representation

```
llvm.mlir.global internal @global(42 : i64) : !llvm.i64

func @some_func(%arg0 : memref<?xf32>) {
  %cst = constant 8 : index
  gpu.launch blocks(%bx, %by, %bz) in (%grid_x = %cst, %grid_y = %cst,
                                         %grid_z = %cst)
    threads(%tx, %ty, %tz) in (%block_x = %cst, %block_y = %cst,
                                %block_z = %cst) {
      gpu.call @device_function() : () -> ()
      %0 = llvm.mlir.addressof @global : !llvm<"i64*"!
      gpu.return
    }
  return
}

gpu.func @device_function() {
  gpu.call @recursive_device_function() : () -> ()
  gpu.return
}

gpu.func @recursive_device_function() {
  gpu.call @recursive_device_function() : () -> ()
  gpu.return
}
```



MLIR has already in tree the capability to represent a unified view of the project across the host and the accelerator.

For example here you have a `gpu.launch` operation that delimit a region that will execute on the accelerator. The code on the GPU is then able to call GPU functions directly. We can make use of the LLVM IR dialect for the host and the device side.

Nested Module -> Split Host/Device Code in the Same IR

```
module attributes {gpu.container_module} {
  func @some_func(%arg0: memref<?xf32>) {
    %c8 = constant 8 : index
    gpu.launch_func(%c8, %c8, %c8, %c8, %c8, %c8)
      {kernel = "function_call_kernel", kernel_module = @function_call_kernel}
      : (index, index, index, index, index, index) -> ()
    return
  }
  gpu.module @function_call_kernel attributes {gpu.kernel_module} {
    func @function_call_kernel() attributes {gpu.kernel} {
      %0 = gpu.block_id() {dimension = "x"} : () -> index
      ...
      %3 = gpu.thread_id() {dimension = "x"} : () -> index
      ...
      call @device_function() : () -> ()
      %12 = llvm.mlir.addressof @global : !llvm<"i64*>
      return
    }
    func @device_function() {
      call @recursive_device_function() : () -> ()
      gpu.return
    }
    llvm.mlir.global internal @global(42 : i64) : !llvm.i64
    func @recursive_device_function() {
      call @recursive_device_function() : () -> ()
      gpu.return
    }
  }
}
```

Existing transformations can be reused to further split the IR with a nested `gpu.module` to group the code that has to be compiled for the accelerator. Again this is all in the same unified IR that preserve the ability to model the entire program across the host-accelerator boundary.

A basic flow is already implemented in MLIR to JIT this and execute it on MLIR GPU, using separate compilation module at the LLVM level to build the PTX and embed it in the CPU module that can be executed.

Nested Module -> Split Host/Device Code in the Same IR

SPIR-V dialect: an example

add %lhs, %rhs: tensor<f32>

```
spv.module Logical GLSL450 requires #spv.vce<
    v1.0, [Shader],
    [SPV_KHR_storage_buffer_storage_class]> {
    spv.EntryPoint "GLCompute" @add
    spv.ExecutionMode @add "LocalSize", 1, 1, 1
    spv.globalVariable @_resource_var_0_0 bind(0, 0) :
        !spv.ptr<!spv.structc
            !spv.array<1 x f32, stride=4> [0], StorageBuffer>
    spv.globalVariable @_resource_var_0_1 bind(0, 1) : ...
    spv.globalVariable @_resource_var_0_2 bind(0, 2) : ...

    spv.func @add() "None" {
        %0 = spv.constant 0 : i32
        %1 = spv._address_of @_resource_var_0_2 : ...
        %2 = spv._address_of @_resource_var_0_0 : ...
        %3 = spv._address_of @_resource_var_0_1 : ...
        %4 = spv.AccessChain %2[%0, %0] : ...
        %5 = spv.Load "StorageBuffer" %4 : f32
        %6 = spv.AccessChain %3[%0, %0] : ...
        %7 = spv.Load "StorageBuffer" %6 : f32
        %8 = spv.FAdd %5, %7 : f32
        %9 = spv.AccessChain %1[%0, %0] : ...
        spv.Store "StorageBuffer" %9, %8 : f32
        spv.Return
    }
}
```

[SPIR-V Dialect and Conversions \(MLIR Open Meeting Tech Talk\)](#)



MLIR also supports SPIRV and Vulkan, we have in-tree a SPIRV dialect allowing to both import SPIRV binaries but more importantly target SPIRV/Vulkan platforms from the GPU abstractions. This may be less common in HPC at the moment, but Vulkan is very common in mobile platforms.

Example: MLIR PatternMatch Execution

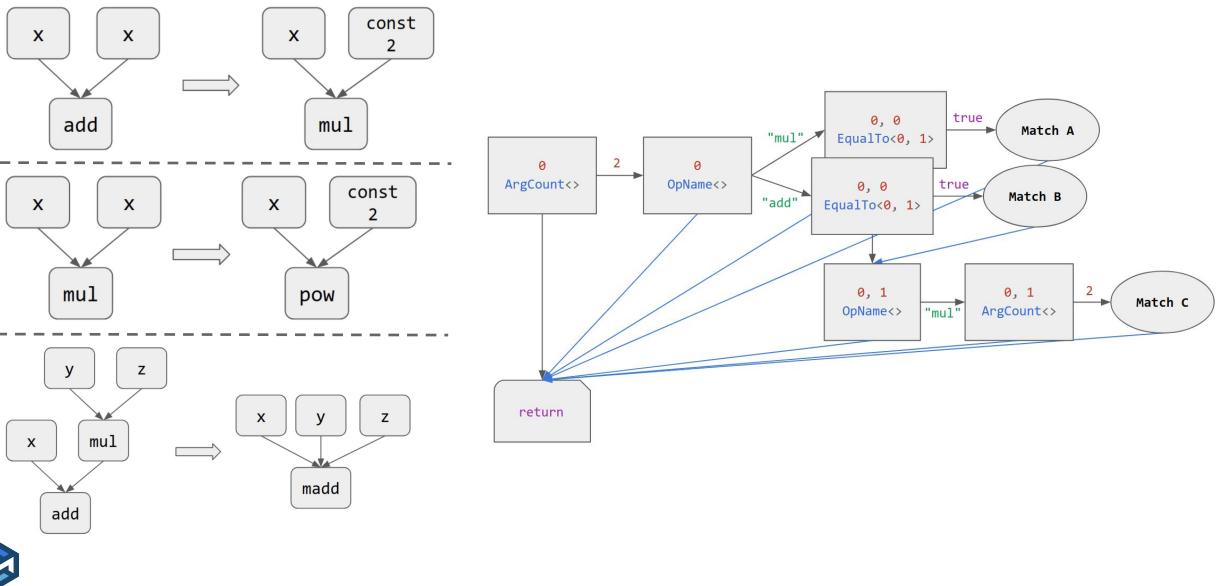
Meta-level: MLIR applied to MLIR internals!



This example may be less interesting from an ecosystem point of view, but it shows an interesting meta-level aspect and I find it too interesting technically to leave it out.

MLIR Pattern Matching and Rewrite

~ Instruction Selection problem.



The idea is to create a dialect to manipulate MLIR IR generically. Starting from rewrites on the IR approaching the instruction selection problem, we can model the available rewrites as a finite state machine, and then generate the code that will actually perform the rewrite. However how to optimize this state machine? Well we implemented a dialect for this!

MLIR Pattern Matching and Rewrite

An MLIR dialect to manipulate MLIR IR!

```
func @matcher(%0 : !Operation) {
^bb0:
  CheckArgCount(%0) [^bb1, ^ex0] {count = 2}
    : (!Operation) -> ()
^bb1:
  CheckOpName(%0) [^bb2, ^bb5] {name = "add"}
    : (!Operation) -> ()
^bb2:
  %1 = GetOperand(%0) {index = 0} : (!Operation) -> !Value
  %2 = GetOperand(%0) {index = 1} : (!Operation) -> !Value
  ValueEqualTo(%1, %2) [^rr0, ^bb3] : (!Value, !Value) -> ()
^rr0:
  // Save x
  RegisterResult(%1) [^bb3] {id = 0} : (!Value) -> ()
^bb3:
  %3 = GetDefiningOp(%2) : (!Value) -> !Operation
  CheckOpName(%3) [^bb4, ^bb5] {name = "mul"}
    : (!Operation) -> ()
^bb4:
  CheckArgCount(%3) [^rr1, ^bb5] {count = 2}
    : (!Operation) -> ()
```

```
^rr1:
  // Save x, y, and z
  %4 = GetOperand(%3) {index = 0} : (!Operation) -> !Value
  %5 = GetOperand(%4) {index = 1} : (!Operation) -> !Value
  RegisterResult(%1, %4, %5) [^bb5] {id = 1}
    : (!Value, !Value, !Value) -> ()
^bb5:
  // Previous calls are not necessarily visible here
  %6 = GetOperand(%0) {index = 0} : (!Operation) -> !Value
  %7 = GetOperand(%0) {index = 1} : (!Operation) -> !Value
  ValueEqualTo(%6, %7) [^bb6, ^ex0] : (!Value, !Value) -> ()
^bb6:
  CheckOpName(%0) [^rr2, ^ex0] {name = "mul"}
    : (!Operation) -> ()
^rr2:
  // Save x
  RegisterResult(%6) [^ex0] {id = 2} : (!Value) -> ()
^ex0:
  return
}
```

[Interpreted Pattern Match Execution](#) (MLIR Open Meeting Tech Talk)
[High level pattern definition dialect PDL Documentation](#)
[Interpreted Pattern Execution Dialect Documentation](#)



This the meta-level I mentioned before, the dialect can describe manipulation of the IR as a program that can be understood, and optimized. For example CSE to eliminate redundant checks on the IR. This also provide a way for dynamically injecting rewrites into the compiler, using a plugin system for example, and optimizing the state machine at runtime.

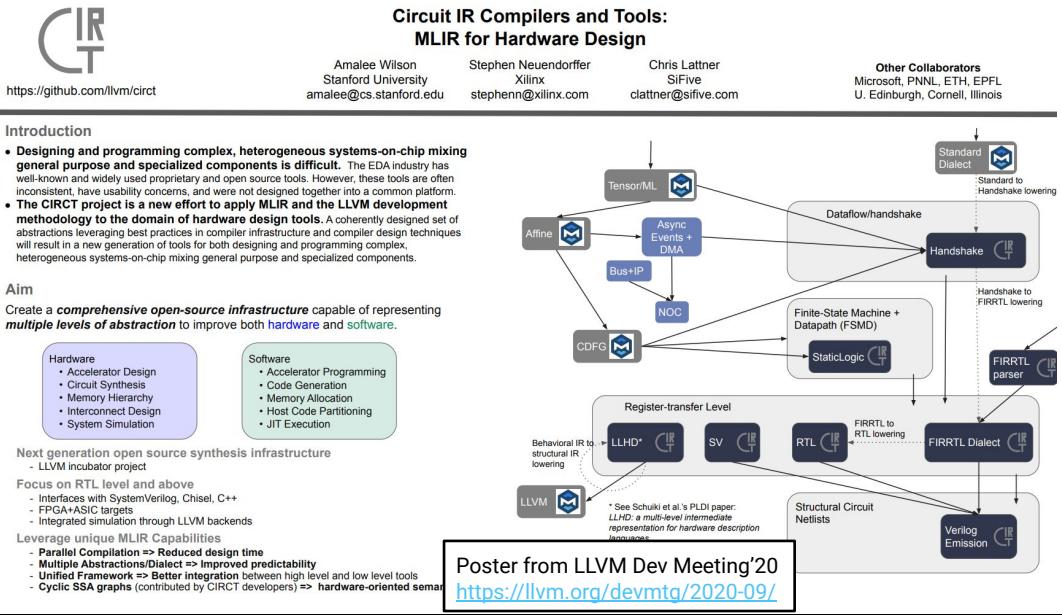
Example: MLIR for HW design



Another example that showcase the wide applicability of the infrastructure is the use of MLIR for HW design.

Example: CIRCT Project

Apply MLIR and the LLVM development methodology to the domain of hardware design tools



In particular the LLVM project recently accepted in its incubator the CIRCT project which aims to apply MLIR and the LLVM development methodology to the domain of hardware design tools. You can see on the schematic on the right points of integration with the ecosystem (with the MLIR logo), and in dark the newly introduced abstractions.

These abstractions may not be interesting to you if you're not in custom HW, but I can directly see how this may help people targeting FPGA for example.

Example: Runtime Abstractions

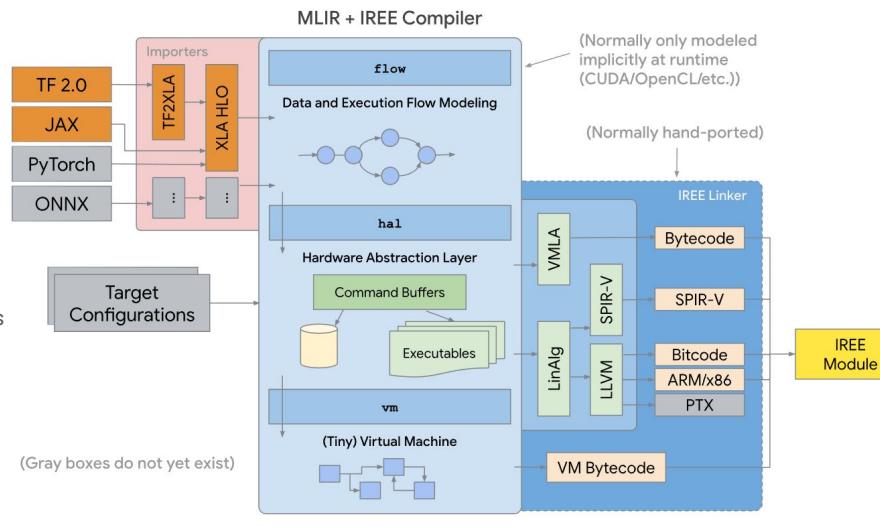


Another category of abstractions is about runtime systems.

IREE: Intermediate Representation Execution Environment

holistic approach towards ML model compilation: the IR produced contains both the *scheduling* logic, required to communicate data dependencies to low-level parallel pipelined hardware/API like Vulkan, and the *execution* logic, encoding dense computation on the hardware in the form of hardware/API-specific binaries like SPIR-V.

<https://google.github.io/iree/>

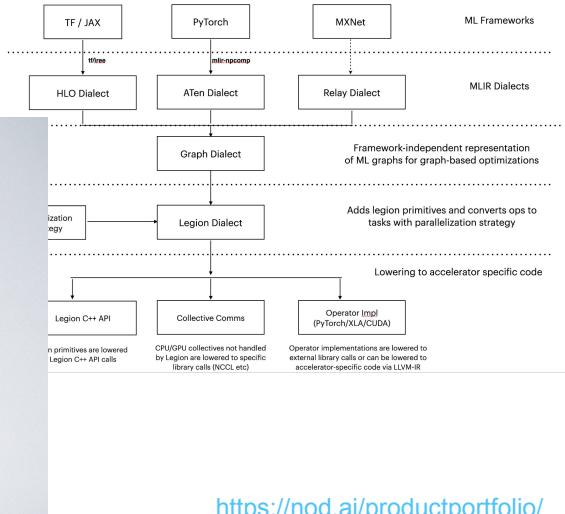


IREE is already the perfect example of leveraging the ecosystem and integrate their ideas into it. They built a low-level runtime system, starting from the principles that drives the Vulkan API, and built multiple levels of abstractions above this: all in MLIR. In this picture that represents IREE, most abstractions have a matching Dialect: `flow`, `hal`, `vmla`. We also find reuse of upstream dialects like `linalg`, `spirv` and `llvm`. The use of the `linalg` (linear algebra) dialect is even abstracting most of the complexity with targeting CPUs or GPUs from the system.

NOD

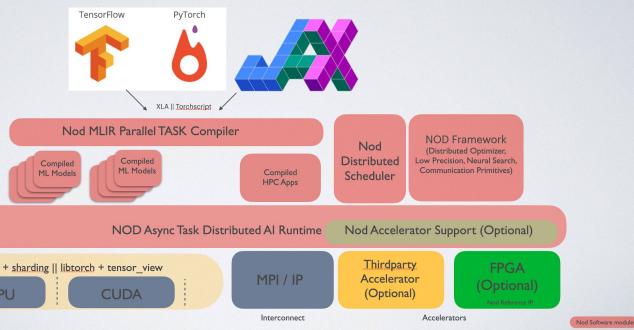


NOD Parallel Compiler



NOD DISTRIBUTED RUNTIME

Asynchronous fine-grained parallel task based distributed runtime built on Legion



<https://nod.ai/productportfolio/>



Another example is the status NOD, which maps some machine learning workload on a distributed runtime system. Their compiler stack on the right introduces a dialect for each level of abstraction, as such their raised the abstraction exposed by the Legion runtime into a dialect that they can expose to their compiler and reason about.

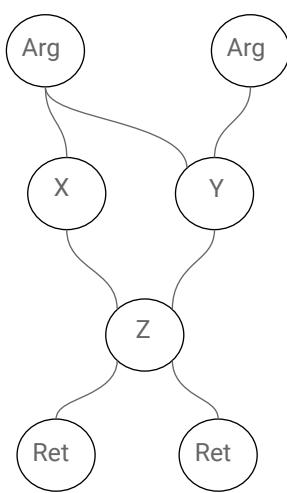
Example: TensorFlow in MLIR

Computational data-flow graphs,
and modeling control flow, asynchrony



TensorFlow itself is making use of MLIR to model its internals.

TensorFlow in MLIR – Computational Graph Dialect



```
func @foo(%arg0: tensor<i1>, %arg1: tensor<...>) ... {
    %X = tf.X %arg0 : tensor<...>
    %Y = tf.Y %arg0, %arg1 : tensor<..., tensor<...>
    %Z:2 = tf.Z %X, %Y : tensor<..., tensor<...>
    return %Z#0, %Z#1 : tensor<..., tensor<...>
}
```



TensorFlow is like XLA modeling its computation using tensors and operators.

We map it to an SSA IR with a topological sort.

We already have some TensorFlow product using MLIR this way, for example to deploy on mobile, TensorFlow users have to invoke a conversion step to target TFLite. This is implemented using dialect conversions in MLIR.

Example: Stencils Computation

MLIR for
accelerating
climate modelling

Open Climate Compiler Initiative



ETH zürich



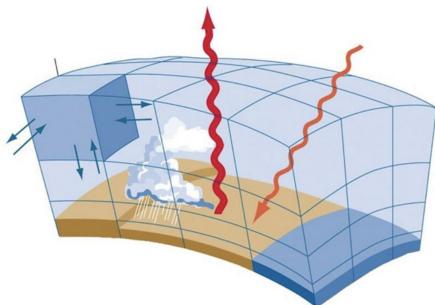
A Compiler Intermediate Representation for Stencils

JEAN-MICHEL GORIUS, TOBIAS WICKY, TOBIAS GROSSER, AND TOBIAS GYSI

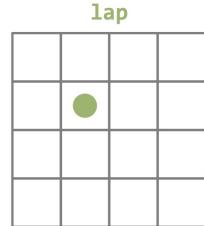
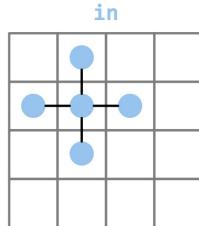
Domain-Science vs Computer-Science

- solve PDE
- finite differences
- structured grid

- element-wise computation
- fixed neighborhood



```
lap(i,j) = -4.0 * in(i,j) +  
          in(i-1,j) + in(i+1,j) +  
          in(i,j-1) + in(i,j+1)
```

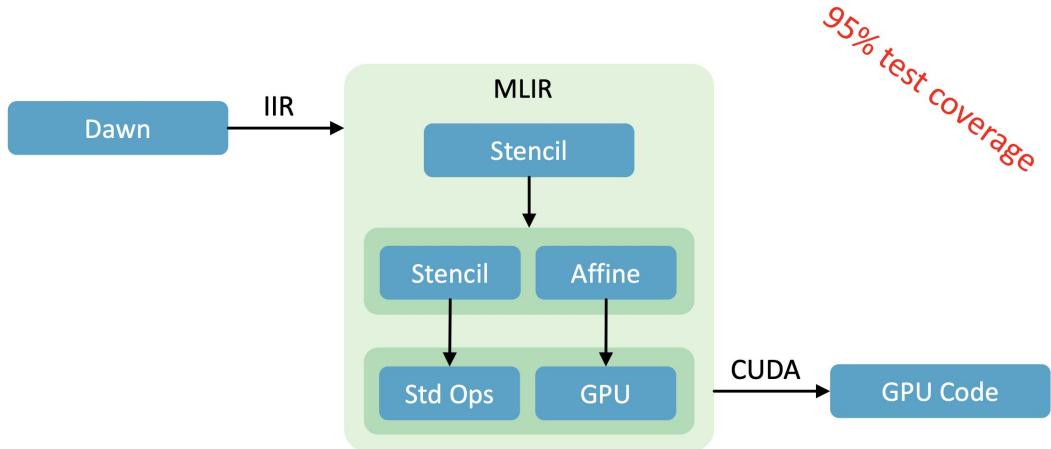


Going back towards a more HPC focus, here is another use case for MLIR: it is a DSL for stencils computation suitable to solve PDE modeling climate and weather. The goal is to map the high level DSL to cluster of multi-GPUs machines.

A Compiler Intermediate Representation for Stencils

JEAN-MICHEL GORIUS, TOBIAS WICKY, TOBIAS GROSSER, AND TOBIAS GYSI

Our Current Toolchain



You can see the value proposal for the MLIR ecosystem here, the Dawn project can focus on the language semantics, and MLIR provides the infrastructure to create their high-level stencil IR, and reuse other components to map it to various targets: accelerators and runtime.

A Compiler Intermediate Representation for Stencils

JEAN-MICHEL GORIUS, TOBIAS WICKY, TOBIAS GROSSER, AND TOBIAS GYSI

Low-level Dialect (IIR)

```
stencil.iir {
    stencil.stencil(%arg0: !stencil<"field:f64">, %arg1: !stencil<"field:f64">) {
        stencil.multi_stage "Parallel" {
            stencil.stage {
                stencil.do_method [0, 0, 60, 0] {
                    %0 = stencil.field_access %arg1 [0, 0, 0] : !stencil<"ptr:f64">
                    %1 = stencil.field_access %arg0 [0, 0, 0] : !stencil<"ptr:f64">
                    %2 = stencil.get_value %0 : f64
                    %3 = stencil.get_value %1 : f64
                    %4 = addf %2, %3 : f64
                    %cst = constant 4.000000e+00 : f64
                    %5 = mulf %4, %cst
                    stencil.write %0, %5 : f64
                }
            }
        }
    }
}
```



Here is a sample of what the stencil dialect they implemented looks like. It is intended to be capturing their computation at a higher-level, retaining the important semantics of the DSL, and as such allowing some specific optimizations. It can be progressively lowered to lower level abstractions and refined depending on the system targeted.

Example: COMET: A Domain-Specific Compilation of High-Performance Computational Chemistry

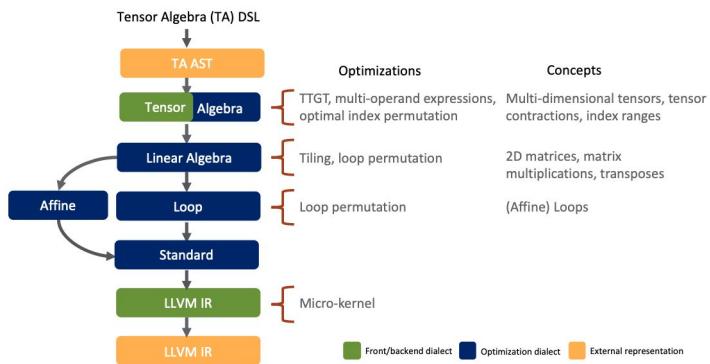


Fig. 1: COMET execution flow and compilation pipeline

*LCPC 2020: COMET: A Domain-Specific Compilation of High-Performance Computational Chemistry
Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar and Gokcen Kestor
Pacific Northwest National Laboratory, The College of William & Mary, Google*



Another recent example is COMET. This is a publication from last month at LCPC: this is a DSL for computational chemistry. Again it fits nicely in the framework and the ecosystem. As the capabilities of MLIR increases, it'll be easy for the author of COMET to benefit from improved optimizations, or support for multi-GPUs or other needs they may have.

```

1 def main(){
2   #IndexLabel Declarations
3   IndexLabel [a, b, c, d, e, f] = [72];
4
5   #Tensor Declarations
6   Tensor<double> A[a, e, b, f];
7   Tensor<double> B[d, f, c, e];
8   Tensor<double> C[a, b, c, d];
9
10  #Tensor Fill Operation
11  A[a, e, b, f] = 1.0;
12  B[d, f, c, e] = 1.0;
13  C[a, b, c, d] = 0.0;
14
15  #Tensor Contraction
16  C[a, b, c, d] = A[a, e, b, f] * B[d, f, c, e];
17 }

#map0 = affine_map<(d0, d1, d2, d3, d4, d5) -> (d0, d4, d1, d5)>
#map1 = affine_map<(d0, d1, d2, d3, d4, d5) -> (d3, d5, d2, d4)>
#map2 = affine_map<(d0, d1, d2, d3, d4, d5) -> (d0, d1, d2, d3)>

module {
  func @main() {
    %c0 = constant 0 : index
    %c72 = constant 72 : index
    %c1 = constant 1 : index
    %0 = "ta.index_label"(%c0, %c72, %c1) : (index, index, index) -> !ta.range
    %1 = "ta.index_label"(%c0, %c72, %c1) : (index, index, index) -> !ta.range
    %2 = "ta.index_label"(%c0, %c72, %c1) : (index, index, index) -> !ta.range
    %3 = "ta.index_label"(%c0, %c72, %c1) : (index, index, index) -> !ta.range
    %4 = "ta.index_label"(%c0, %c72, %c1) : (index, index, index) -> !ta.range
    %5 = "ta.index_label"(%c0, %c72, %c1) : (index, index, index) -> !ta.range
    %6 = "ta.tensor_decl"(%0, %4, %1, %5) : (!ta.range, !ta.range, !ta.range, !ta.range)
      -> tensor<72x72x72x72xf64>
    %7 = "ta.tensor_decl"(%3, %5, %2, %4) : (!ta.range, !ta.range, !ta.range, !ta.range)
      -> tensor<72x72x72x72xf64>
    %8 = "ta.tensor_decl"(%0, %1, %2, %3) : (!ta.range, !ta.range, !ta.range, !ta.range)
      -> tensor<72x72x72x72xf64>
    "ta.fill"(%6) {value = 1.000000e+00 : f64} : (tensor<72x72x72x72xf64>) -> ()
    "ta.fill"(%7) {value = 1.000000e+00 : f64} : (tensor<72x72x72x72xf64>) -> ()
    "ta.fill"(%8) {value = 0.000000e+00 : f64} : (tensor<72x72x72x72xf64>) -> ()
    "ta.tc"(%6, %7, %8) {indexing_maps = [#map0, #map1, #map2]} :
      (tensor<72x72x72x72xf64>, tensor<72x72x72x72xf64>,
       tensor<72x72x72x72xf64>) -> ()
    "ta.return"() : () -> ()
  }
}

```



(a) DSL TA language

(b) TA dialect

This is showing the COMET DSL on the left, and the matching dialect in MLIR before it gets lowered to other abstractions and optimized for a given system.

Example: Fortran IR

Flang: the LLVM Fortran Fortrend



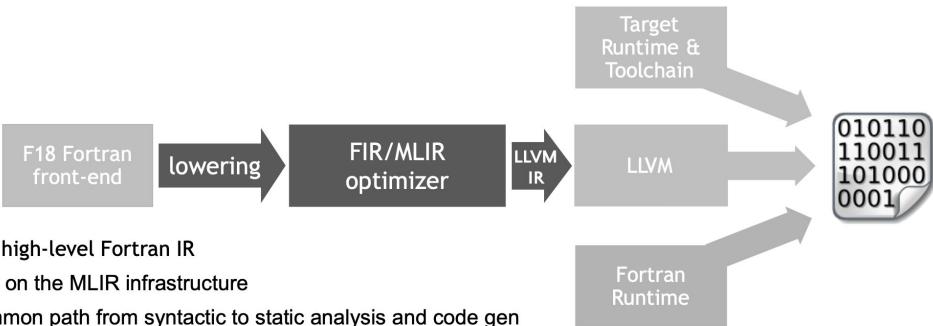
We may get really into the heart of HPC now with Flang: the LLVM Fortran Compiler.

An MLIR Dialect for High-Level Optimization of Fortran

Eric Schweitz (NVIDIA)

FLANG

The LLVM Fortran compiler



3

NVIDIA

The design for the Flang IR is based on MLIR. This follows a design similar to Rust or Swift.

[An MLIR Dialect for High-Level Optimization of Fortran](#)

Eric Schweitz (NVIDIA)

LOOPS

An example of loop optimization

```
// subroutine convolution(r, f, g)
func @convolution(%r : !fir.box<!fir.array<?:f32>>, %f : !fir.box<...>, %g : !fir.box<...>) {
    %uf:3 = fir.box_dims %f, 0 : (!fir.box<...>, index) -> (index, index, index) ... // and %ug:3
    fir.loop %n = 1 to %ug#1 {
        fir.loop %k = 1 to %ug#1 {
            %2 = subi %n, %k : index
            %3 = fir.coordinate_of %f, %2 : (!fir.box<...>, index) -> !fir.ref<f32>
            %4 = fir.load %3 : !fir.ref<f32> ... // and likewise %6 = load g[k]
            %7 = mulf %6, %4 : f32 ... // and likewise %9 = load r[n]
            %10 = addf %9, %7 : f32
            fir.store %10 to %8 : !fir.ref<f32>
    }
}
```



Here is how the dialect may look like: just like for the previous DSL it is intended to capture the Fortran specific semantics and enable accurate analyses and transformations.

[An MLIR Dialect for High-Level Optimization of Fortran](#)

Eric Schweitz (NVIDIA)

OBJECT-ORIENTED PROGRAMMING

FIR: Devirtualization

```
// dispatch table for type(u)
fir.dispatch_table @dtbl_type_u {
    fir.dt_entry "method", @u_method
}

%uv = fir.alloca !fir.type<u> : !fir.ref<!fir.type<u>>
fir.dispatch "method"(%uv) : (!fir.ref<!fir.type<u>>) -> ()
```



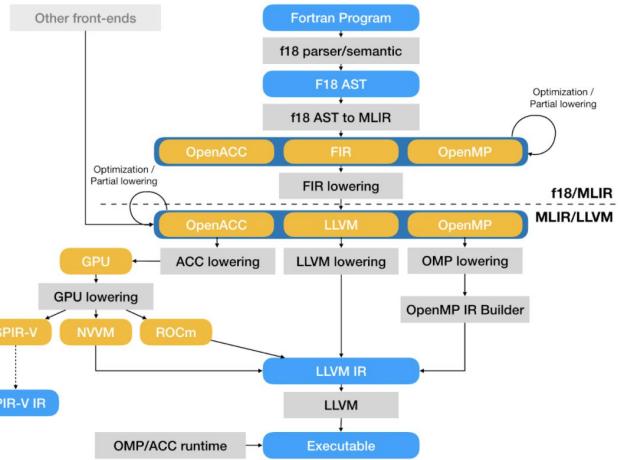
The kind of thing that are easier to recover with the language semantics than when you end up at the LLVM level can be devirtualization: by representing virtual tables and virtual calls as first class concept you can leverage the guarantees of the language to devirtualize calls.

Leveraging LLVM Ecosystem to Meet a Critical ECP (community) need : FORTRAN

- Fortran support continues to be an ongoing requirement
- Flang project started in NNSA funding NVIDIA/PGI to open source compiler front-end into LLVM ecosystem
- SOLLVE is improving OpenMP dialect, implementation, and core optimizations
- PROTEAS-TUNE is creating OpenACC dialect and improving MLIR
- ECP projects are contributing many changes upstream to LLVM core, MLIR, etc
- Many others are contributing: backends for processors, optimizations in toolchain, ...
 - Google contributed MLIR

ECP Projects: Flang, SOLLVE, PROTEAS-TUNE
 Many other contributors: NNSA, NVIDIA, ARM, Google,

...



Finally, during the LCPC keynote last month, Jeffrey Vetter from Oak Ridge National Lab captured the picture accurately for Flang. You can really see how components from MLIR, below the dotted line are leveraged to provide to Flang features like OpenMP for multi-processor or OpenACC to target GPUs. This means that targeting GPUs from Fortran with Flang could use the same optimization and codegen path as XLA used from TensorFlow: this consolidation of effort in the LLVM project represents one of the goal of MLIR.

Beyond this picture implementing the Fortran standards and some extensions, the fact that Flang is being implemented itself with a set of Dialects means that the Fortran internal abstractions may be open and reusable. This could enable DSL authors in a HPC context to design their language and tools with a close interaction with Fortran libraries (instead of going down to C-like FFI, which is fairly rigid).



MLIR

This concludes this section, and it is now time to wrap up!

MLIR : Reusable Compiler Abstraction Toolbox, and More!

MLIR provides all the infrastructure to build IR and transformations:

- Same infra at each abstraction level
- Investment in toolings has compounding effects

IR design involves multiple tradeoffs

- Iterative process, constant learning experience
- MLIR makes compiler design “agile” (and fun!)

Building an ecosystem around **dialects**:

- Abstractions like “software libraries”
- Every “library API” becomes an IR construct: may be composed, understood, transformed, ...
- New “compiler blocks” that compose and lower the cost of writing a new toolchain
- LLVM IR unified the CPU abstraction layer for every frontend, MLIR embraces the many abstractions in a end-to-end stack.
- Large span: from DSLs to runtime to HW design.



With the benefit of hindsight here are some takeaways. The impedance mismatch between LLVM IR and programmers gave rise to *many* systems and countless rewrites of similar infrastructure, with varying quality. MLIR breaks away with the “one-size-fits-all” approach that LLVM IR pushed forward for compilers targeting CPUs.

Our experience with MLIR is that it makes compiler development more agile, more iterative, but also more importantly: it involves a lot of fun!

Finally my angle today was to advocate for the ecosystem effect. I tried to draw an analogy with the reuse and composability we get from software library with the Dialect abstractions that MLIR provides. The LLVM community can grow and continue to be the place to collaborate on defining these abstractions, and possibly this decade will bring to HPC users the same level of usability and productivity that ML practitioners enjoy.

MLIR and the associated ecosystem has the potential to impact compiler research, reducing the startup cost for new project, making it easy to experiment with new ideas by maximizing reuse of existing flow, and finally reduce the path from research to production.



Thank you!

MLIR

Questions?

<https://mlir.llvm.org/>

Join the community:

[Discourse Forums](#)

[Discord Chat](#)

Weekly open meeting

[Biweekly newsletter](#)

Finally I'd like to point out that we have a weekly open meeting with tech talks that are recorded and published on the MLIR website. The community is mainly on the LLVM Discourse forums, and if you prefer live chat we're on Discord. Finally we publish a bi-weekly newsletter to stay tuned into the latest developments.