IR Design for Heterogeneity: Challenges and Opportunities

CC — February 22, 2020

Albert Cohen, Google, Paris

A New Golden Age for Computer Architecture

John Hennessy and David Patterson's
ISCA 2018 Turing Award Lecture

"We believe the deceleration of performance gains for standard microprocessors, the opportunities in high-level, domain-specific languages and security, the freeing of architects from the chains of proprietary ISAs, and (ironically) the ending of Dennard scaling and Moore's law will lead to another Golden Age for architecture"

A New Golden Age for *Optimizing Compilers*

"We live in a heterogeneous world of domain-specific languages and accelerators, freeing programming language and computer architects from the chains of general-purpose, one-size-fits-all designs."

→ A call to action for **compiler construction**

A New Golden Age for *Optimizing Compilers*

“We live in a heterogeneous world of domain-specific languages and accelerators, freeing programming language and computer architects from the chains of general-purpose, one-size-fits-all designs.”

What to expect in the next 45mn

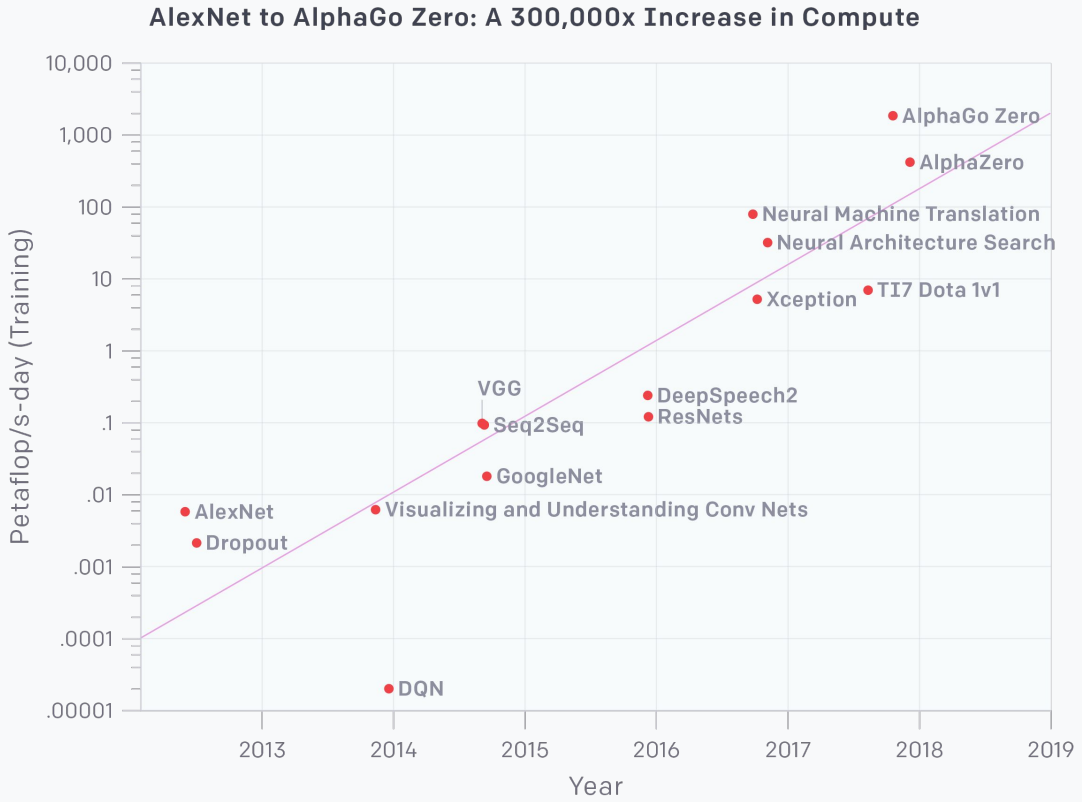
1. Some HPC and ML context
   opportunities for compilers
2. Heterogeneity in action
   focus on lowering tensor algebra to tiled, vector accelerators
3. IR design directions and research
   industry perspective, academic perspective

A Detour Through ML Applications

Google AI

Models are growing and getting more complex

- **Model Size:** larger models require more multiply accumulate operations.
- **Model Complexity:** as model complexity increases it becomes harder to fully utilize hardware.
- **Much faster than Moore's law**



Source: OpenAI - AI & Compute

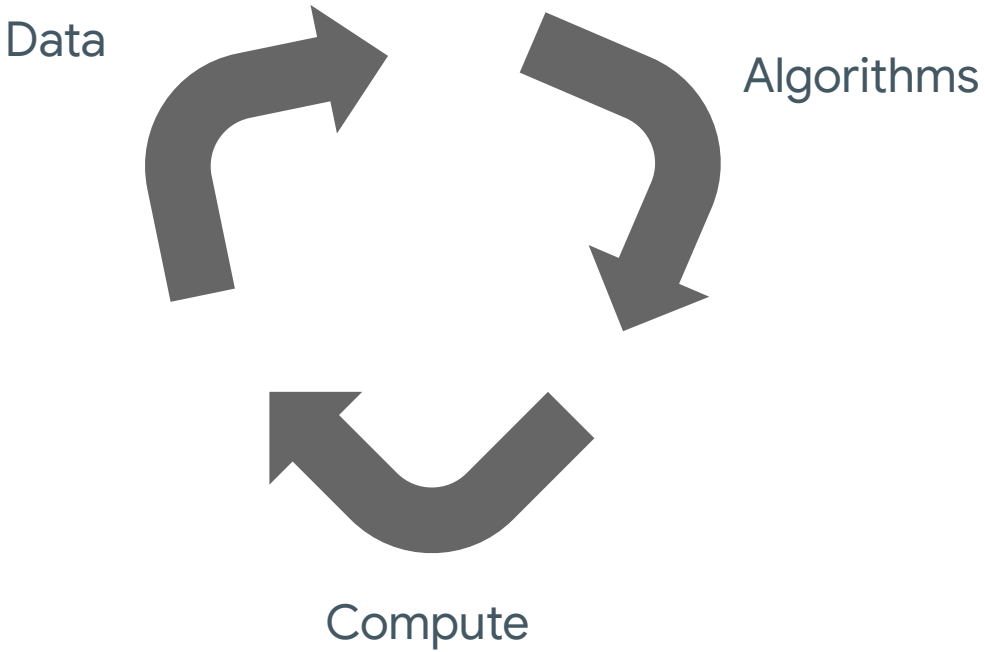
A Detour Through ML Applications

ML is: **data + algorithms + compute**

~ Data drives the continuous improvement cycle for ML models

~ Researchers provide new algorithmic innovations unlocking new techniques and models

~ Compute allows it all to scale as datasets get larger and algorithms need to scale on that accordingly

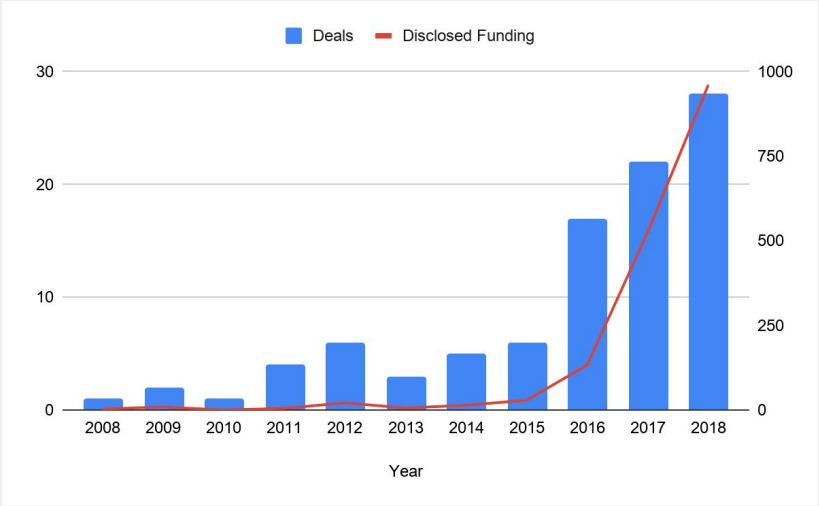


Explosion in Hardware Startups

Google AI

Significant growth in deals and funding

ML semiconductors global funding history
($M, # of deals)



Explosion in Cloud and HPC Accelerators

Google AI

Chip Manufacturers:

Volta, Vega, Ampere

Nervana

Graphcore

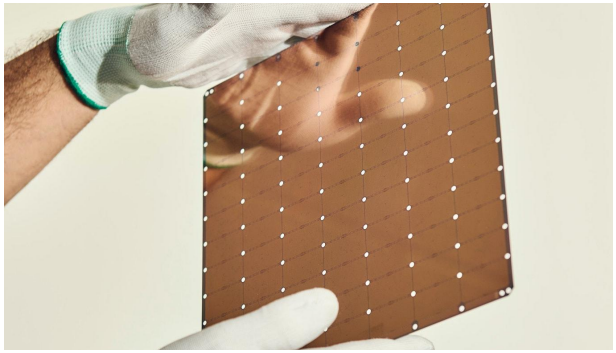
Habana

Cerebras Systems

... and many more

Habana

Intel



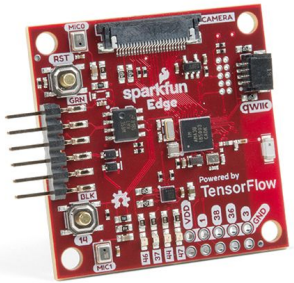
Cerebras Systems

Graphcore

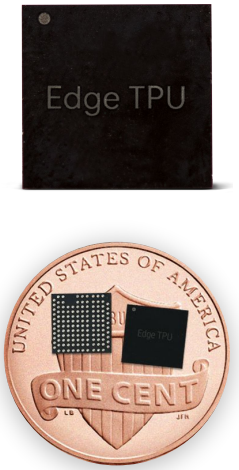
Explosion in Embedded, Mobile, Edge Hardware

Google AI

~5.5B Mobile Phones



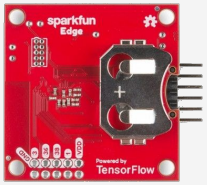
250B+ Microcontrollers



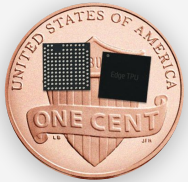
Edge TPUs

CPU GPU DSP NPU

Heterogeneous Compute



Microcontrollers



Edge TPUs

With increasingly complexity

~ Heterogeneous hardware is now the norm

~ Scaling from phones down to microcontrollers

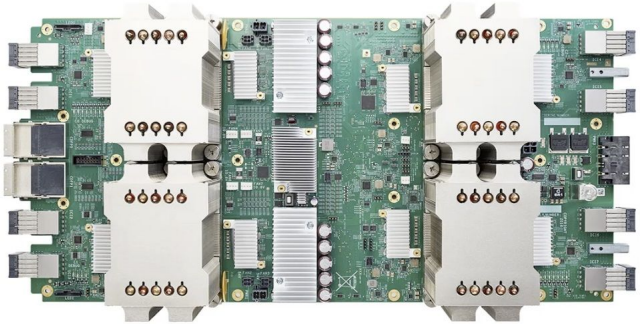
~ Memory, energy, performance and latency constraints become paramount

More Hardware... More Complexity...

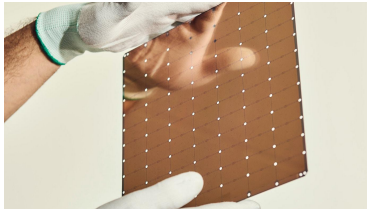
~ Many different hardware accelerators focused on ML

~ Many different types and architectures: 4-bit, 16-bit, 32-bit...

~ Inability to quickly scale up and down hardware consistently and varying levels of abstractions

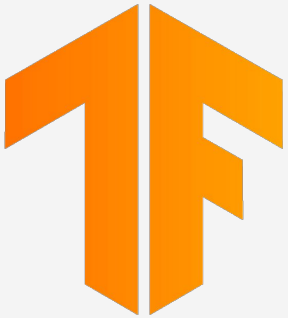


TPU's



Cerebras Systems

Graphcore



PyTorch

CNTK

mxnet

ONNX

HW is not just to blame here

ML Software Explosion too...

~ Many frameworks

~ Many different graph implementations

~ Each framework is trying to gain a usability and performance edge over each other

Google AI

None of this is scaling

Because

~ Systems that don't interoperate

~ Cannot handle all these operators and types consistently on all hardware

~ Poor developer usability and debuggability across hardware

~ No generalizable standard for ensuring software and hardware scales together

Google AI

Any relief from programming languages?

Investment in a new software infrastructure?

Stepping Back: Three Dimensions of Heterogeneity

Languages
Source, Domain-Specific

- Source
  system language, infrastructure, application programming, script, managed memory...
- Domain-Specific
  DSL or embedded DSL, lifting APIs into DSLs, cross-domain

Abstractions
Nature, Level, Gray Box

- Nature
  types and logic, data structures, control structures, concurrency
- Level
  tensor/matrix vs. array/pointer, loops vs. linearized control flow, security property vs. hardening
- Gray Box
  interaction, static analysis, debug, traceability

Targets
Framework, ISA, Primitive Block

- Framework
  execution environment (OpenMP or TensorFlow runtime), data and communication abstraction
- ISA
  scalar, vector, memory spaces, memory model, DMA, NVRAM
- Primitive Block
  HW block, native library API

Challenges: **soundness** and **performance portability**

separation? composition? sharing? system-level? compiler construction?

MLIR

What is MLIR?

~ An extensible representation for types and operations, control & compute

~ Driven by ML training and inference, scaling from **mobile** to **cloud**

~ Best in class programming models and compiler technology

~ Independent of the execution environment

~ Modular, reusable components

~ Enabling the progressive lowering of higher level abstractions

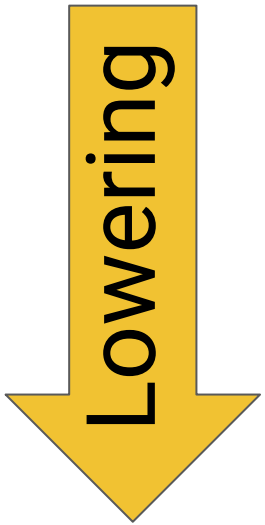
Industry Adoption

95% of the world's data-center accelerator hardware

4 billion mobile phones
countless IoT devices

Governance moved to LLVM
https://mlir.llvm.org

MLIR — Compute Graphs to Instructions in One Slide



TensorFlow

```
%x = "tf.Conv2d"(%input, %filter)
          {strides: [1,1,2,1], padding: "SAME", dilations: [2,1,1,1]}
     : (tensor<*xf32>, tensor<*xf32>) -> tensor<*xf32>
```

XLA HLO

```
%m = “xla.AllToAll"(%z)
          {split_dimension: 1, concat_dimension: 0, split_count: 2}
     : (memref<300x200x32xf32>) -> memref<600x100x32xf32>
```

LLVM IR

```
%f = llvm.add %a, %b
     : !llvm.float
```

And many more abstractions and levels: TF-Lite, structured linear algebra operations, nested control flow, affine loops, quantized operations, GPU, etc.

Mix and Match in one IR

MLIR — Modeling TensorFlow Control & Concurrency

Control flow and dynamic features of TensorFlow 1, TensorFlow 2

- Conversion from control to data flow
- Both lazy and eager evaluation modes

Concurrency

- Sequential execution in blocks
- Distribution
- Offloading
- Concurrency in tf.graph regions
  Implicit **futures** to capture asynchronous task parallelism within SSA and CFG graph representations

MLIR — GPU Acceleration

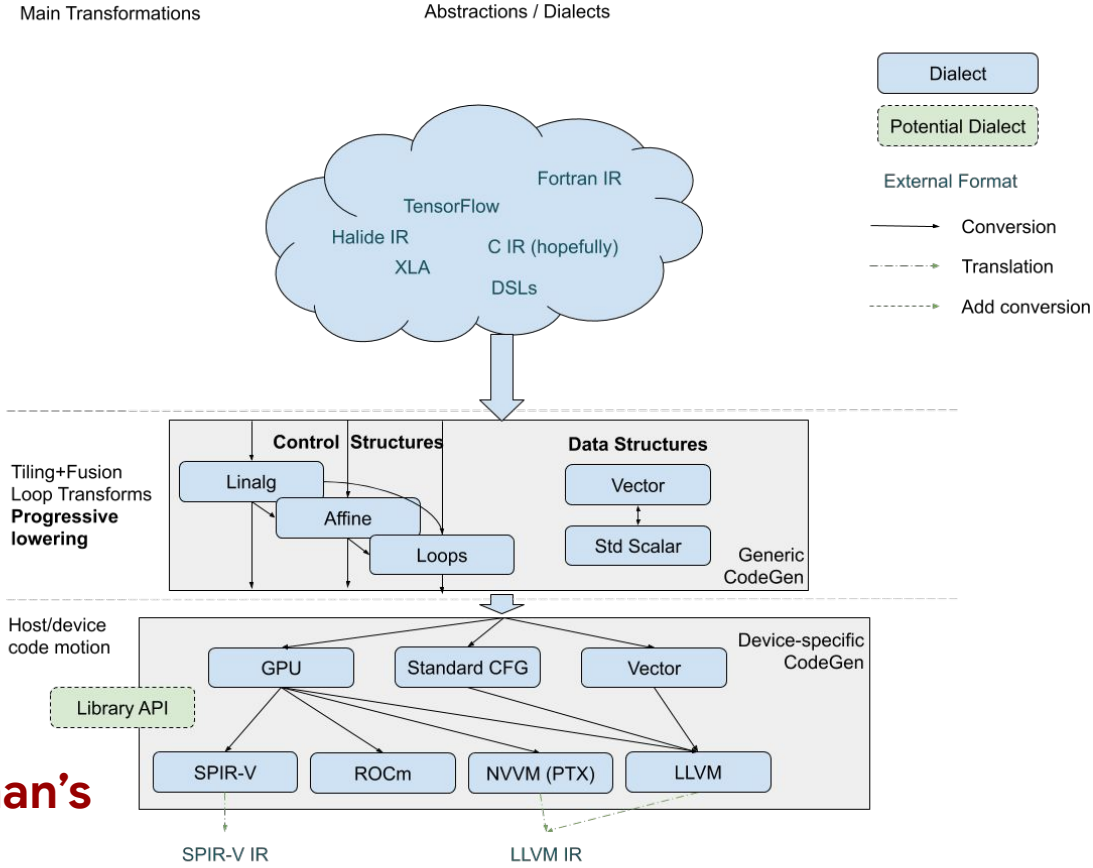
MLIR Open Design Meeting
December 12, 2019

And many more dialects, projects

See also:

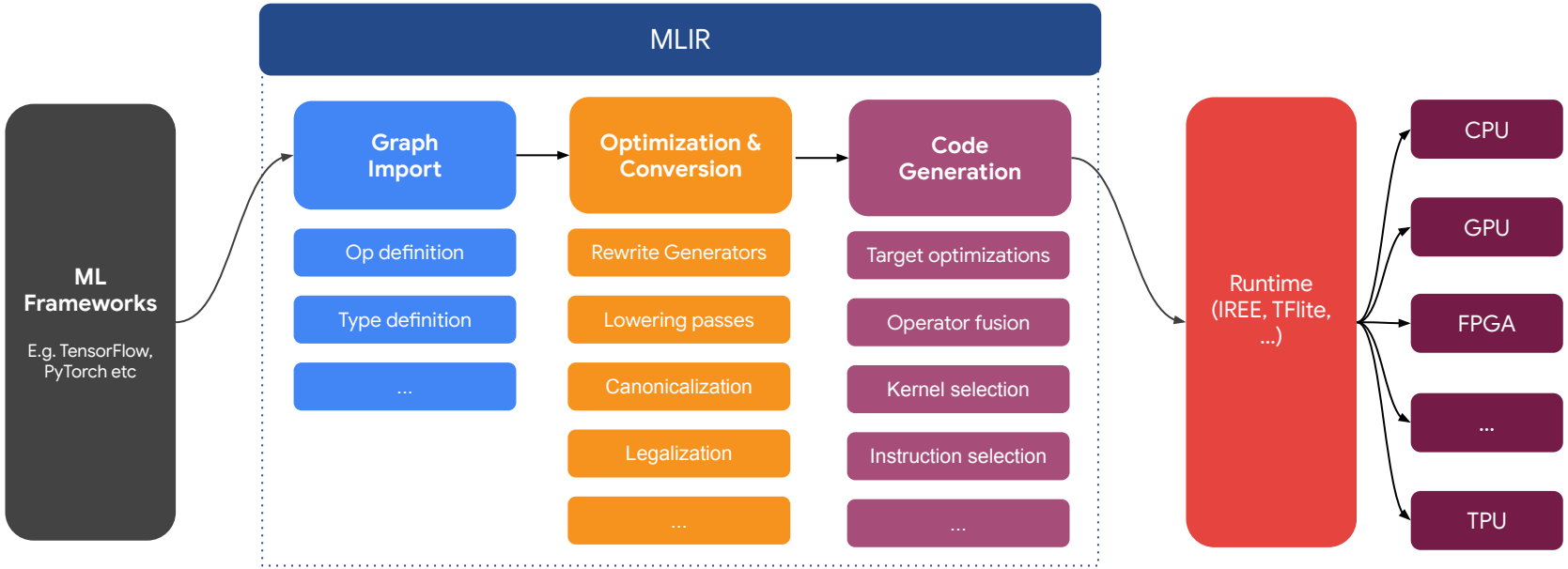
Jacques Pienaar's Sunday C4ML presentation

Chris Lattner and Tatiana Shpeisman's Wednesday keynote



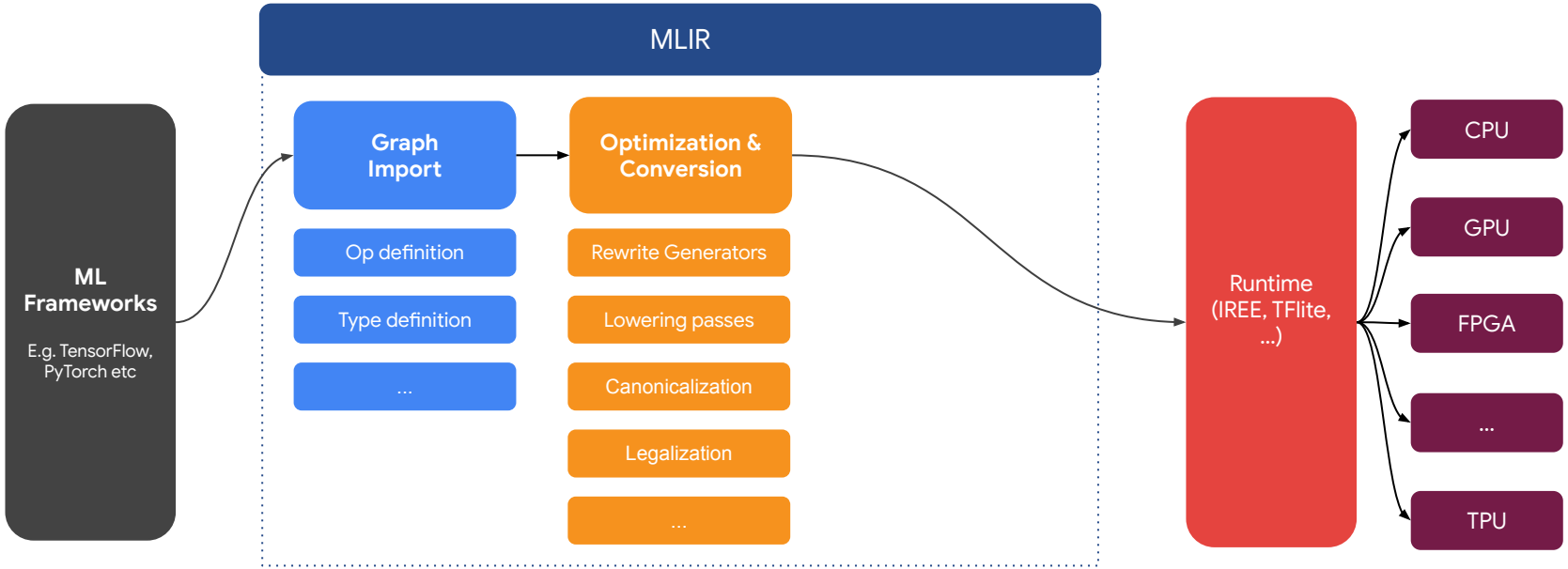
MLIR Compiler Infrastructure

A common graph representation and legalization framework,
a common set of optimization and conversion passes and code generation pipeline



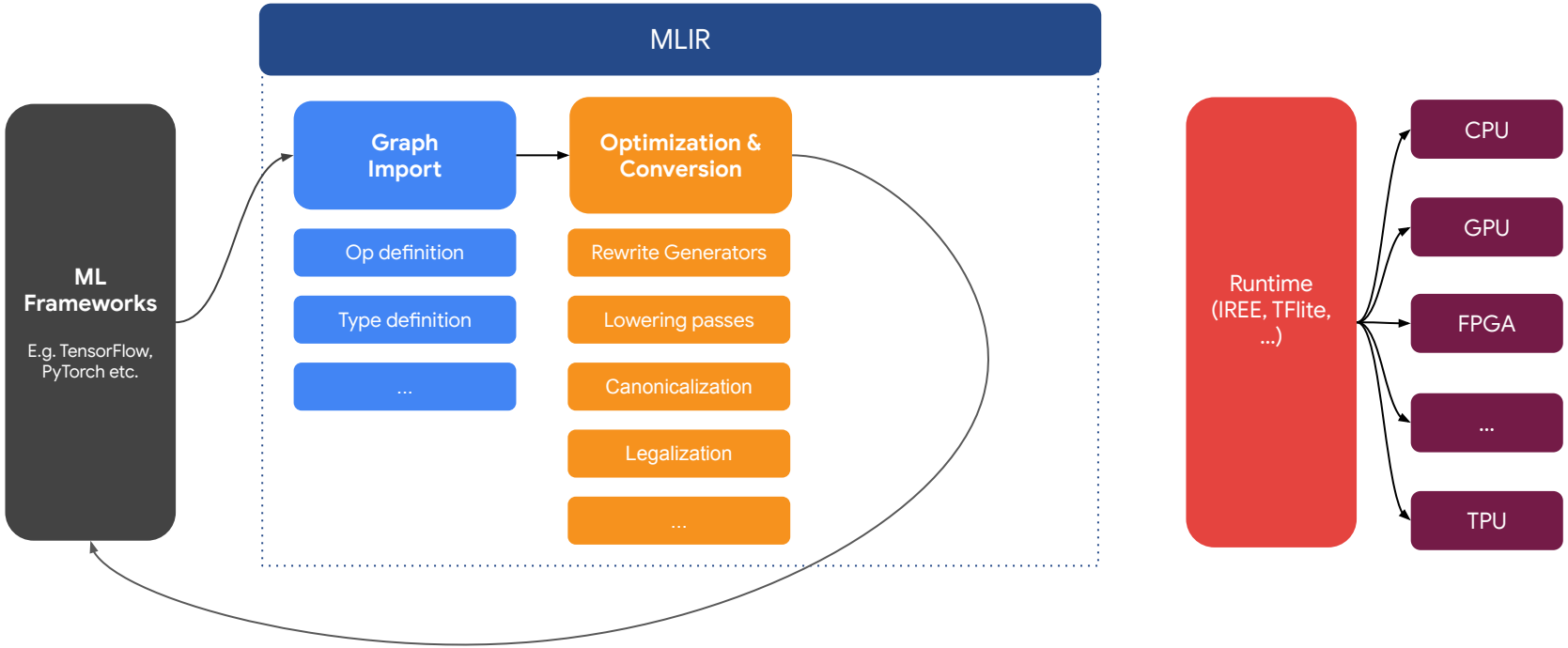
It's Unopinionated

Utilize different components of the system as needed



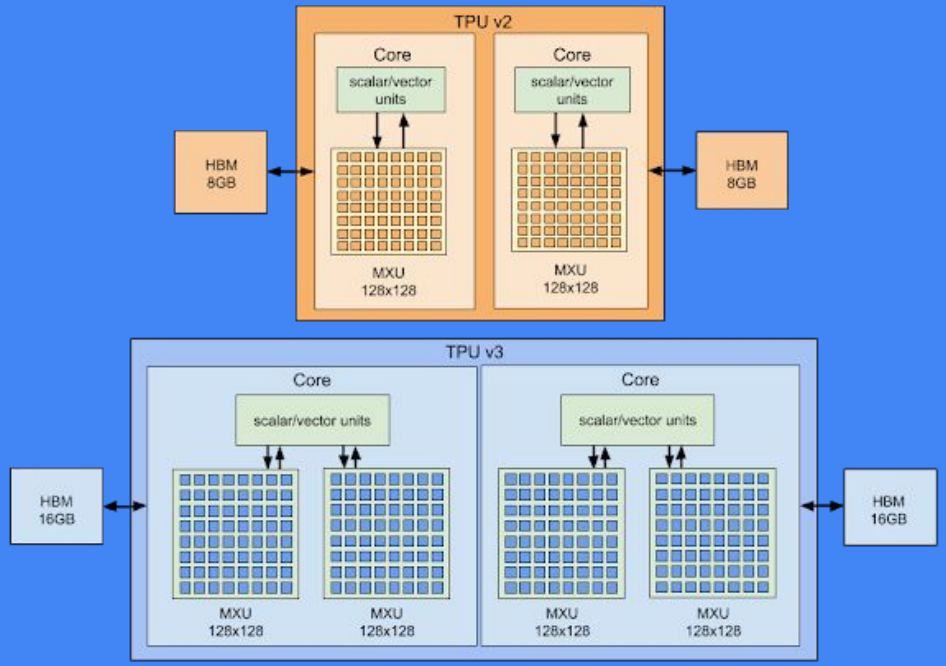
One Size Fits None

MLIR can also be modularized as a graph rewriting tool, e.g. for TensorFlow Lite



MLIR

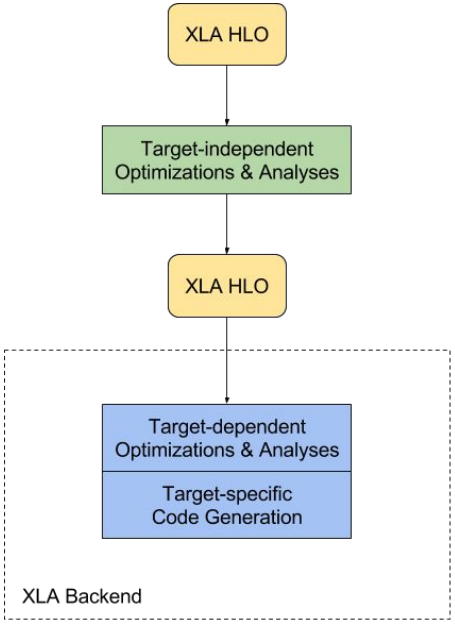
Focus: Programming Tiled SIMD Hardware

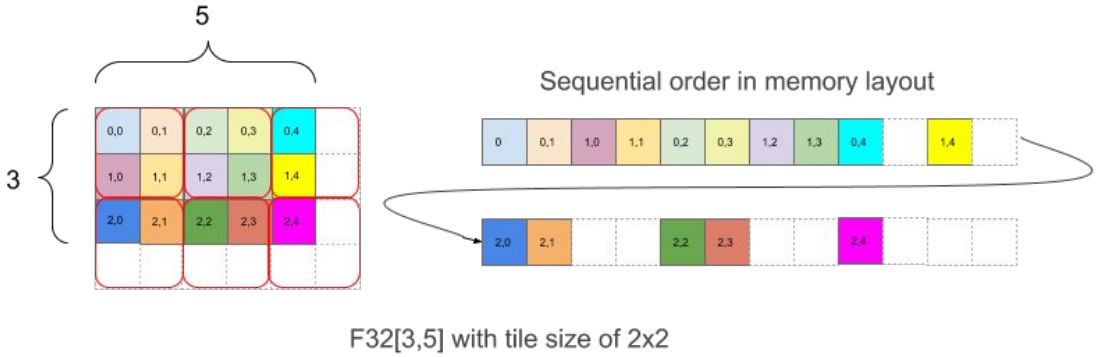


Google

Tiles Everywhere

1. Hardware
2. Data Layout





F32[3,5] with tile size of 2x2

Example: XLA compiler, Tiled data layout

Repeated/Hierarchical Tiling
e.g., BF16 (bfloat16)
on Cloud TPU
(should be 8x128 then 2x1)

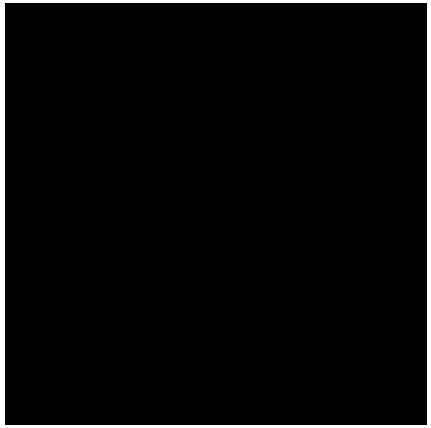
Tiles Everywhere

1. Hardware
2. Data Layout
3. Control Flow
4. Data Flow
5. Data Parallelism

Example: Halide for image processing pipelines

https://halide-lang.org

Meta-programming API and domain-specific language (DSL) for loop transformations, numerical computing kernels



Tiling in Halide

Tiled schedule:
strip-mine (a.k.a. split)
permute (a.k.a. reorder)

Vectorized schedule:
strip-mine
vectorize inner loop

Non-divisible bounds/extent:
strip-mine
shift left/up
redundant computation
(also forward substitute/inline operand)

Tiles Everywhere

1. Hardware
2. Data Layout
3. Control Flow
4. Data Flow
5. Data Parallelism

Example: Halide for image processing pipelines

https://halide-lang.org

And also TVM for neural networks

https://tvm.ai

TVM example: scan cell (RNN)

```
m = tvm.var("m")
n = tvm.var("n")
X = tvm.placeholder((m,n), name="X")
s_state = tvm.placeholder((m,n))
s_init = tvm.compute((1,n), lambda _,i: X[0,i])
s_do = tvm.compute((m,n), lambda t,i: s_state[t-1,i] + X[t,i])
s_scan = tvm.scan(s_init, s_do, s_state, inputs=[X])

s = tvm.create_schedule(s_scan.op)

// Schedule to run the scan cell on a CUDA device
block_x = tvm.thread_axis("blockIdx.x")
thread_x = tvm.thread_axis("threadIdx.x")
xo,xi = s[s_init].split(s_init.op.axis[1], factor=num_thread)
s[s_init].bind(xo, block_x)
s[s_init].bind(xi, thread_x)
xo,xi = s[s_do].split(s_do.op.axis[1], factor=num_thread)
s[s_do].bind(xo, block_x)
s[s_do].bind(xi, thread_x)
print(tvm.lower(s, [X, s_scan], simple_mode=True))
```

Challenge: Compile to Learn → Learn to Compile

- Move past handwritten heuristics
  - NP complete problems
  - Cost models that are hard or infeasible to characterize
  - Hardware explosion, model diversity, problem diversity
- Autotuning, search and caching
  - Separate algorithms from policy
  - Exploit structure in the search space

Telamon: Commutative Optimizations on Partially Specified Implementations

with Ulysse Beaugnon, Basile Clément and Andi Drebes, Nicolas Tollenaere
ENS, Inria, Google

CC 2017: *Optimization space pruning without regrets*
Ulysse Beaugnon, Antoine Pouille, Marc Pouzet, Jacques Pienaar, Albert Cohen

arXiv preprint: *On the Representation of Partially Specified Implementations and its Application to the Optimization of Linear Algebra Kernels on GPU*
Ulysse Beaugnon, Basile Clément, Nicolas Tollenaere, Albert Cohen

Problem Statement

Context: “superoptimizing” loop nests in numerical kernels
Challenge: finding best implementation/optimization decisions is hard

- Optimizations do not compose well, they may enable or disable others
- Cannot infer precise performance estimation from intermediate compilation steps

Yet… optimizing compilation never seems to catch up

... new hardware, optimization tricks

… witnessing a widening performance portability gap

Telamon Approach

Candidates as Partially Specified Implementations

- Optimizations as independent, commutative decisions
  e.g., tile? unrolling? ordering?
- Vector of choices, listed upfront, decisions taken in any order
  defer any interference to search
- Synthesize imperative code from fixed/complete decision vectors
  e.g., infer buffers, control flow

Constraint Programming for Semantics and Resource Modeling

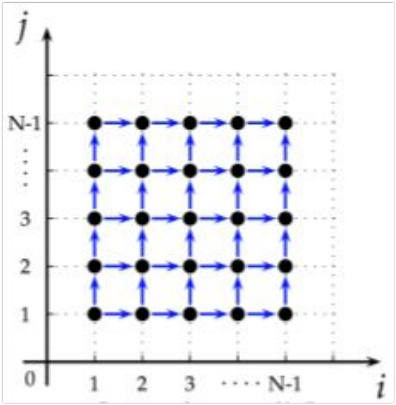
- Control structure
  e.g., loop nesting
- Semantics of the kernel
  e.g., def-use, array dependences
- Optimization interactions
  e.g., enabling transformations
- Resource constraints
  e.g., local memory

Branch-and-Bound- and MCTS-enabled **Search**

- Lower bound derived from orthogonal resource modeling
  inspired by roofline modeling
- Lower bound for a candidate = ideal performance for a set of potential implementations
- Empowered by structured, decision vector and CSP-based implementation space

Candidates?

Inspired From Polyhedral Compilation



- Polyhedral compilation
  - Affine scheduling
    e.g., ILP-based
  - Code generation
    from affine schedules to nested loops

- Meta-programming array processing code
  - Halide / TVM specific combinators and scheduling/mapping primitives
  - URUK, CHiLL
    with automatic schedule completion

TVM example: scan cell (RNN)

```
m = tvm.var("m")
n = tvm.var("n")
X = tvm.placeholder((m,n), name="X")
s_state = tvm.placeholder((m,n))
s_init = tvm.compute((1,n), lambda _,i: X[0,i])
s_do = tvm.compute((m,n), lambda t,i: s_state[t-1,i] + X[t,i])
s_scan = tvm.scan(s_init, s_update, s_state, inputs=[X])
s = tvm.create_schedule(s_scan.op)
// Schedule to run the scan cell on a CUDA device
block_x = tvm.thread_axis("blockIdx.x")
thread_x = tvm.thread_axis("threadIdx.x")
xo,xi = s[s_init].split(s_init.op.axis[1], factor=num_thread)
s[s_init].bind(xo, block_x)
s[s_init].bind(xi, thread_x)
xo,xi = s[s_do].split(s_do.op.axis[1], factor=num_thread)
s[s_do].bind(xo, block_x)
s[s_do].bind(xi, thread_x)
print(tvm.lower(s, [X, s_scan], simple_mode=True))
```

Constraints?

Inspired From Program Synthesis and Superoptimization

- Program synthesis
  - Start from denotational specification, possibly partial (sketching), or (counter-)examples
    Telamon ≈ Domain-specific denotations
  - Guess possible implementations by (guided) sampling lots of random ones
    Telamon ≈ Guess efficient implementations by (guided) sampling lots of stupid ones
  - Filter correct implementations using SMT solver or theorem prover
    Telamon ≈ Constraint programming to model both correctness and hardware mapping
- Superoptimization
  - Typically on basic blocks, with SAT solver or theorem prover and search
  - Architecture and performance modeling
    Telamon ≈ Operate on loop nests and arrays

Search?

Inspired From Adaptive Libraries and Autotuning

- Feedback-directed and iterative compiler optimization, lots of work since the late 90s
- Adaptive libraries
  - SPIRAL: *Domain-Specific Language (DSL) + Rewrite Rules + Multi-Armed Bandit or MCTS* http://www.spiral.net
  - ATLAS, FFTW, etc.: *hand-written fixed-size kernels + micro-benchmarks + meta-heuristics*
  - Pouchet et al. (affine), Park et al. (affine and CFG): *Genetic Algorithm, SVM, Graph Kernels*
- Telamon
  - vs. SPIRAL, FFTW: better structured, independent/commutative choices, branch-and-bound
  - vs. Pouchet and Park: finite space, bounded vectors

Candidates

Partially Instantiated Vector of Decisions

- Every choice is a decision variable
- Taking a decision = restricting a domain
- Fully specified implementation ⇔ All decision variables assigned a single value

```
- order(a, b) ∈ { Before, After }
- order(a, c) ∈ { Before }
- …
```

Candidates and Constraints

Kernel

```
%yr=%d0dd%X[0]216 {
%yr=%d0d=%0,d4X[0]{
for %y0==a0dd%X[0]42
    %z = add %y, %d0
}
```

Decisions

```
order(%x, %d0) ∈ { Before, Inner } <- decision
order(%x, %y)  ∈ { Before }
order(%y, %d0) ∈ { Before, Inner } <- constraint propagation
...
```

(In the first and third lines, "Before" is struck through.)

Enforce coherent decisions with constraints

```
order(x, d0) = Inner && order(x, y) = Before => order(y, d0) ∈ { Inner, After }
```

Enabling Better Search Algorithms

Well Behaved Set of Actions

- Commute
- All decisions known upfront
- Constraint propagation almost never backtracks in practice

Flat, Fixed Sized, Ideal Environment for Reinforcement Learning (RL)

- Extract features from the decision vector
- Global heuristics, aware of all potential optimizations
- Infer all possible decisions (actions) and/or estimate performance

Constraint Satisfaction Problem (CSP)

Find an Assignment for Functions

```
kind: Dimension -> { Loop, Unrolled, Vector, Thread, Block }

order: Statements x Statements -> { Before, After, Inner, Outer, Fused }
```

That Respects Constraints

```
∀ a, b ∈ Dimension. order(a, b) = Fused => kind(a) = kind(b)
```

(a.k.a. typed fusion)

CSP Without the Optimization Aspect

- Finding an implementation is easy: random decisions + constraint propagation
- Use CSP to represent, not to solve the problem
- No analytical objective function
  - Analytical functions cannot model the complexity of the hardware
  - Hardware details are proprietary
  - Rely on actual performance evaluations
  - And external heuristics

Telamon on GPU

Generic loop nest and array optimizations + GPU-specific optimizations

Supported Decisions

- Strip mining factor
- Loop interchange
- Loop fusion
- Software pipelining
- Statement Scheduling
- Rematerialization
- Memory layout
- Copy to local memories
- Double buffering
- Vectorization

Example: Vector Addition

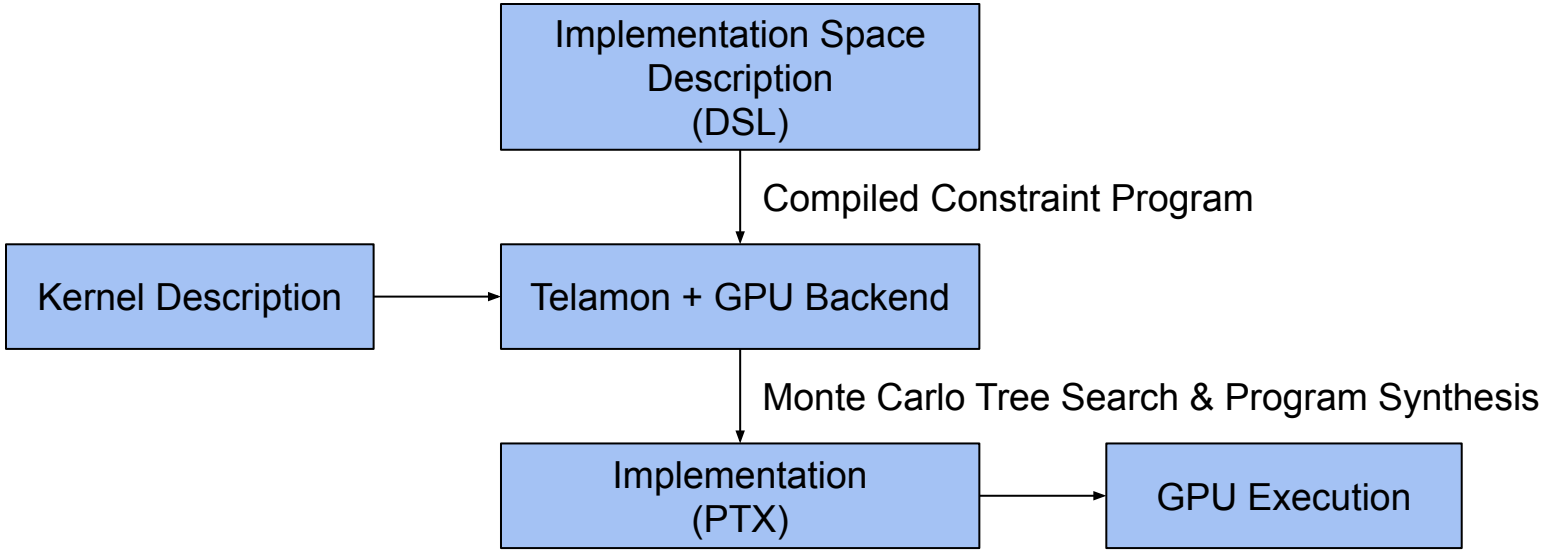
Computes Z = X + Y

```
- load X
- load Y
- add X and Y into Z
- store Z
```

Implementation space

- Each instruction in its own loop
- Strip-mined 3 times
- Can choose strip-mining factors
- Can fuse, interchange and unroll loops
- Can reorder instructions
- Can coalesce transfers across memory spaces

Telamon System Overview



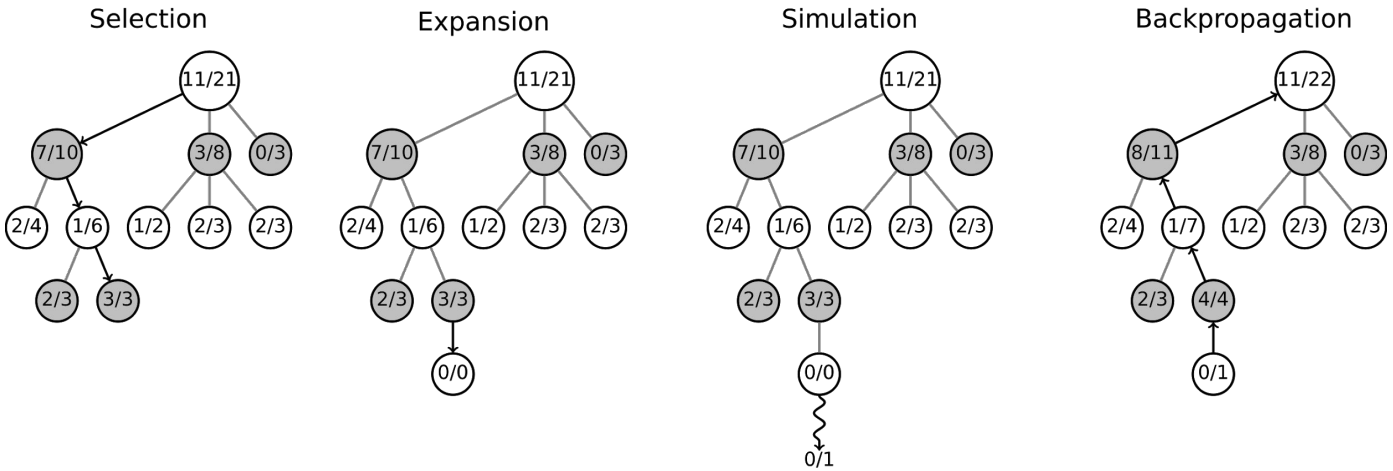
Branch and Bound + Monte Carlo Tree Search (MCTS)

Performance model of a lower bound on the execution time

$$\forall x \in S.\ \mathrm{Model}(S) \leq \mathrm{Time}(x)$$

- Enables Branch & Bound, with feedback from real executions
  - Reduces the search space by several orders of magnitude
  - Prunes early in the search tree (75% in the first two levels for matmul on GPU)
- Possible because it is aware of potential future decisions
- GPU model of block- and thread-level performance features, as well as single-thread microarchitecture
  - No cache and registers model (yet)
  - Coarse-grain model of the interaction between bottlenecks

Zooming in the MCTS-Based Search



Source: Wikipedia

Search Issues (Ongoing Research)

- **Match or outperform the best domain-specific generators**
- **High variance of the search time (stuck in suboptimal areas)**
- **Lots of dead-ends**
  - Mostly due to performance model
  - ~20x more dead-ends than implementations
- **Non-stationary distribution due to cuts**
  - Somewhat intrinsic to MCTS
  - Branch & bound strategy makes it trickier

MLIR

Call to Action: *Extensibility* & *Hackability* & *Research*

Heterogeneity ⇒ need for a **super-extensible = super-reusable** system
catalyzing next-generation accelerator adoption and research

- domain-specific languages as first-class constructs
- domain-specific hardware constructs as first-class operations
- lowering and mixing language and hardware abstractions
- type systems: novel numerics, sparse tensors, logic properties, dependent
- concurrency & parallel constructs, memory modeling
- model and carry debug information, traceability, security properties
- model structured search spaces of program transformations:
  algorithmic and graph rewriting, polyhedral, synthesis

Compiler Construction Design for Diversity

We are hiring!
mlir-hiring@google.com

Convergent Needs of ML and Scientific HPC

~ Embarrassingly parallel dimensions, e.g., batching

~ Data parallelism and “model” (task, streaming) parallelism

~ Fast single-sided communication abstractions

~ Highest possible bisection bandwidth for fast all-reduce

~ Numerical libraries for single-threaded or single-node performance

~ Growing model complexity (data flow and control flow)

~ Towards “differentiable programming”, coupling with other computations

→ Well aligned with HPC practices and trends

Yet ML Also Pushes for Changes in HPC

~ Automation across the board: architecture search, code generation

~ Python, C++, domain-specific languages, more diverse user populations

~ Workloads: diverse, unstable, virtualized, growing share of inference

~ Open source infrastructure and algorithms, but seldom build-from-source

~ Non-standard data representation: sparse, low-precision fp variants

~ Built-in fault tolerance and elasticity, data center and edge

~ More diverse dimensions of / opportunities for approximation

→ Disrupting HPC practices