The New Frontiers of Compilers in the AI Era

Giuseppe Tagliavini

Department of Computer Science and Engineering (DISI)

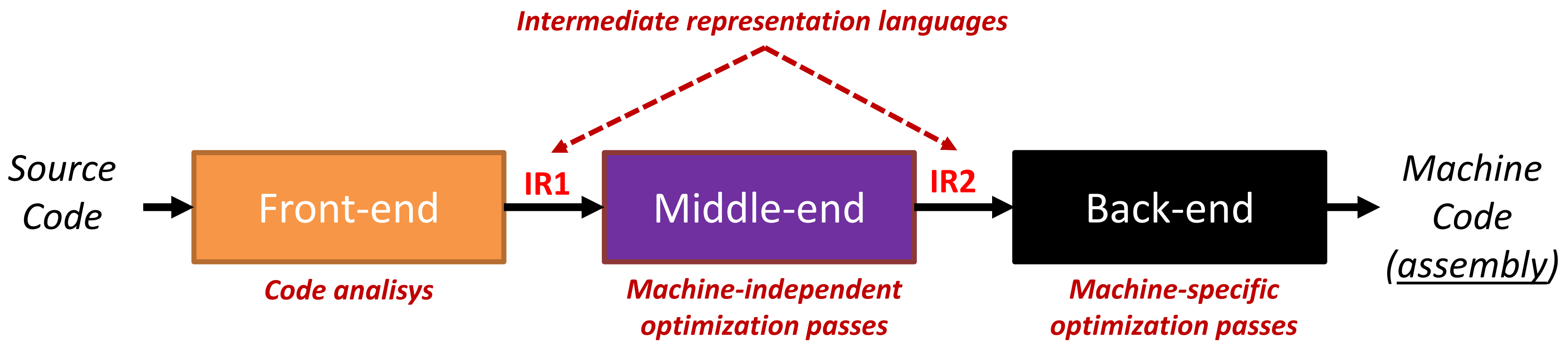
ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Outline

- **Introduction:** ***A quick recap of the compiler architecture***
- Compilers serving AI tasks: *A new era for compilers*
- AI tasks serving compilers: *A look at the future*
- Conclusion: *Do we really trust these technologies?*

Anatomy of a compiler

- High-level design of modern compilers (e.g., GCC and LLVM):



- **Intermediate representation (IR)** languages → programming languages (e.g., C or Python) are too high-level, assembly is too low-level
  - Language-independent
  - Machine-independent (as long as possible!)
  - Easy to optimize

LLVM compilation flow: From C to Assembly (1/4)

```c
void mul_sum(int *a, int *b, int
*result, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += a[i] * b[i];
    }
    *result = sum;
}
```

C code (7 lines)

```llvm
define dso_local void @mul_sum(ptr noundef %a, ptr noundef %b, ptr noundef %result,
i32 noundef %n) #0 {
entry:
  %a.addr = alloca ptr, align 4
  %b.addr = alloca ptr, align 4
  %result.addr = alloca ptr, align 4
  %n.addr = alloca i32, align 4
  %sum = alloca i32, align 4
  %i = alloca i32, align 4
  store ptr %a, ptr %a.addr, align 4
  store ptr %b, ptr %b.addr, align 4
  store ptr %result, ptr %result.addr, align 4
  store i32 %n, ptr %n.addr, align 4
  store i32 0, ptr %sum, align 4
  store i32 0, ptr %i, align 4
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %0 = load i32, ptr %i, align 4
  %1 = load i32, ptr %n.addr, align 4
  %cmp = icmp slt i32 %0, %1
  br i1 %cmp, label %for.body, label %for.end
...
}
```

Initial LLVM IR (49 lines)

LLVM compilation flow: From C to Assembly (2/4)

```
define dso_local void @mul_sum(ptr nocapture noundef readonly %a, ptr nocapture noundef readonly %b, ptr nocapture noundef writeonly
%result, i32 noundef %n) local_unnamed_addr #0 {
entry:
  %cmp6 = icmp sgt i32 %n, 0
  br i1 %cmp6, label %for.body, label %for.cond.cleanup

for.cond.cleanup:                                 ; preds = %for.body, %entry
  %sum.0.lcssa = phi i32 [ 0, %entry ], [ %add, %for.body ]
  store i32 %sum.0.lcssa, ptr %result, align 4, !tbaa !4
  ret void

for.body:                                         ; preds = %entry, %for.body
  %i.08 = phi i32 [ %inc, %for.body ], [ 0, %entry ]
  %sum.07 = phi i32 [ %add, %for.body ], [ 0, %entry ]
  %arrayidx = getelementptr inbounds i32, ptr %a, i32 %i.08
  %0 = load i32, ptr %arrayidx, align 4, !tbaa !4
  %arrayidx1 = getelementptr inbounds i32, ptr %b, i32 %i.08
  %1 = load i32, ptr %arrayidx1, align 4, !tbaa !4
  %mul = mul nsw i32 %1, %0
  %add = add nsw i32 %mul, %sum.07
  %inc = add nuw nsw i32 %i.08, 1
  %exitcond.not = icmp eq i32 %inc, %n
  br i1 %exitcond.not, label %for.cond.cleanup, label %for.body, !llvm.loop !8
}
```

Optimized LLVM IR (23 lines)

LLVM compilation flow: From C to Assembly (3/4)

```
name:            mul_sum
body:
    bb.0.entry:
        %11:gpr = COPY $x13
        %10:gpr = COPY $x12
        %22:gpr = COPY $x11
        %21:gpr = COPY $x10
        BLT $x0, %11, %bb.2

    ...

    bb.3.for.cond.cleanup:
        SW %23, %10, 0 :: (store (s32) into %ir.result, !tbaa !4)
        PseudoRET
    bb.4.for.body:
        %17:gpr = LW %21, 0 :: (load (s32) from %ir.lsr.iv2, !tbaa !4)
        %18:gpr = LW %22, 0 :: (load (s32) from %ir.lsr.iv1, !tbaa !4)
        %19:gpr = nsw MUL %18, %17
        %23:gpr = nsw ADD %19, %23
        %22:gpr = ADDI %22, 4
        %21:gpr = ADDI %21, 4
        BEQ %22, %0, %bb.3
        PseudoBR %bb.4
```

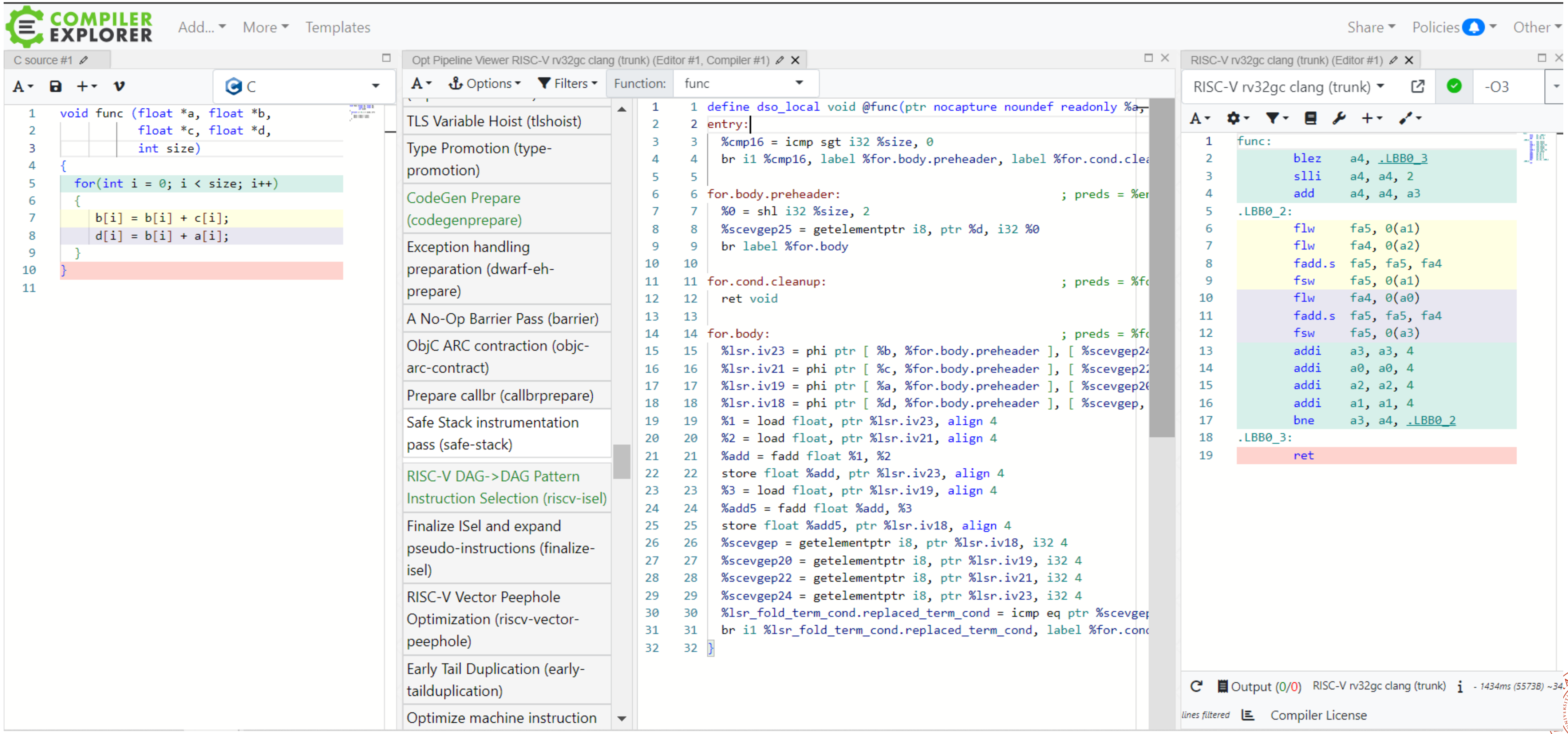
MIR before regalloc (31 lines)

LLVM compilation flow: From C to Assembly (4/4)

```
mul_sum:
li      a4, 0
blez    a3, .LBB0_3
# %bb.1:   # %for.body.preheader
slli    a3, a3, 2
add     a6, a1, a3
.LBB0_2:   # %for.body
lw      a5, 0(a0)
lw      a3, 0(a1)
mul     a3, a3, a5
add     a4, a4, a3
addi    a1, a1, 4
addi    a0, a0, 4
bne     a1, a6, .LBB0_2
.LBB0_3:   # %for.cond.cleanup
sw      a4, 0(a2)
ret
```

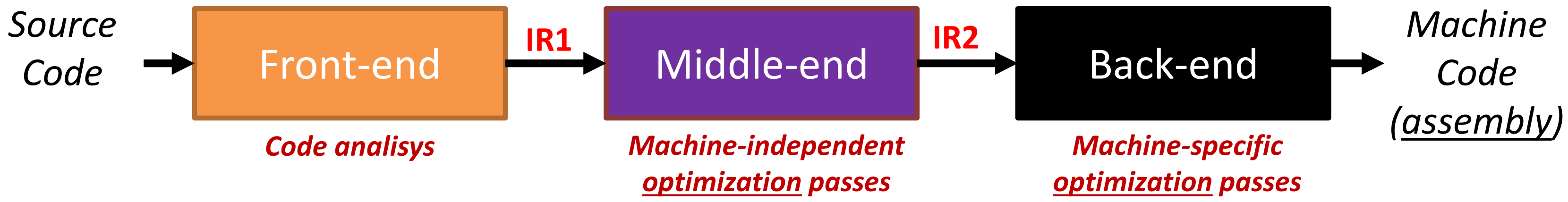
RV32 assembly LLVM IR (17 lines)

Compiler Explorer: Democratizing Compiler Optimization Passes!



Anatomy of a compiler

- High-level design of modern compilers (e.g., GCC and LLVM):



What do we really mean for «optimization»?
From a compiler toolchain perspective, <u>optimizing</u> means <u>transforming</u> one program representation into another

Outline

- Introduction: *A quick recap of the compiler architecture*
- **Compilers serving AI tasks: *A new era for compilers***
- AI tasks serving compilers: *A look at the future*
- Conclusion: *Do we really trust these technologies?*

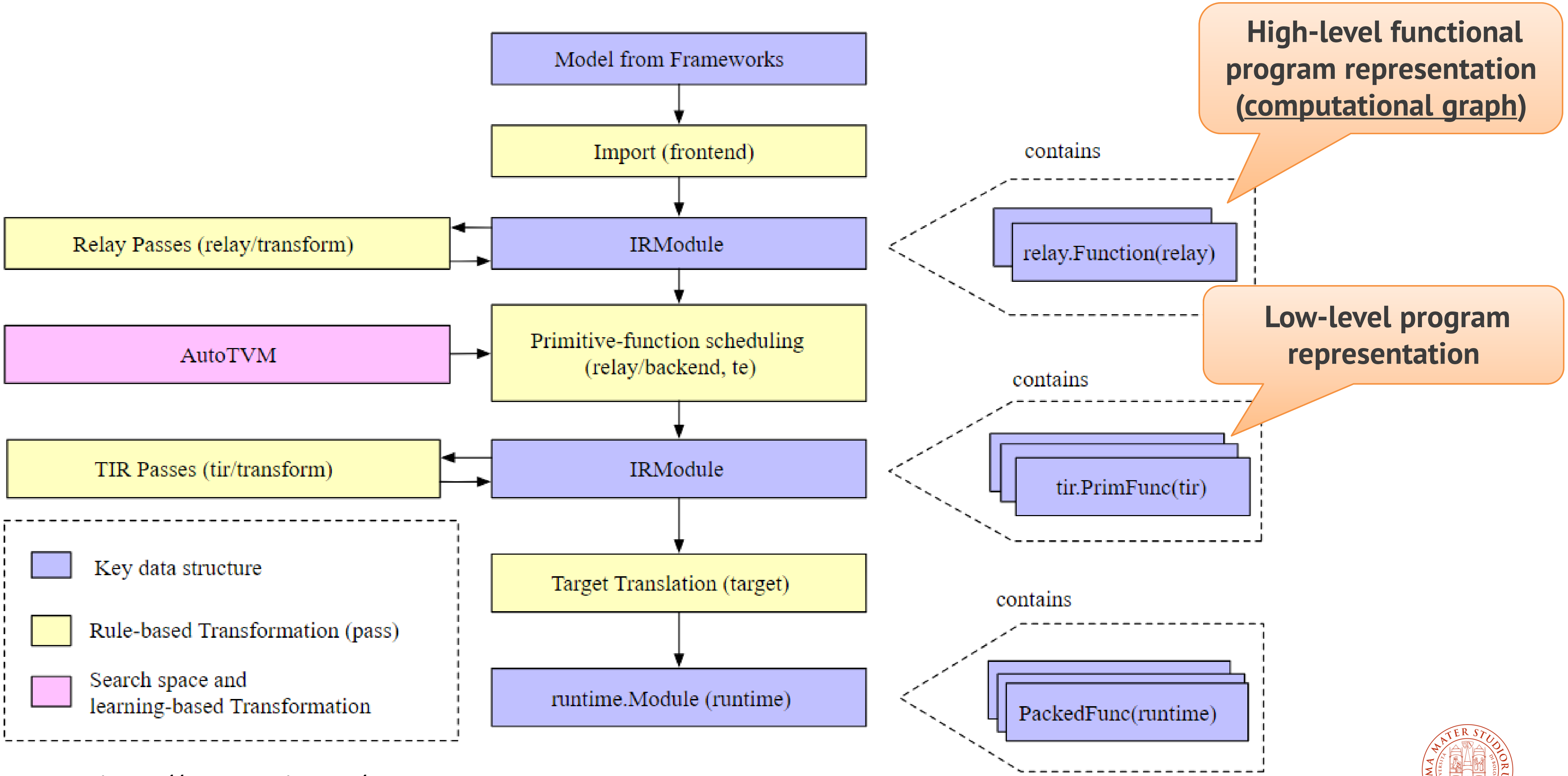
Deep Learning Compilers

- Current DL frameworks (e.g., TensorFlow, MXNet, Caffe, and PyTorch) rely on **graph-level optimizations**
  - constant folding, common subexpressions elimination (CSE), redundant control edge removal, algebraic simplifications, …

- **Operator-level optimizations** are critical for efficient support of diverse hardware targets
  - Standard approach consists of adopting *manually optimized operator libraries*

- A **DL compiler** takes a high-level specification of a DL model and generates low-level optimized code for different hardware targets
  - TVM (Apache Foundation), XLA (Google), Glow (Facebook), …

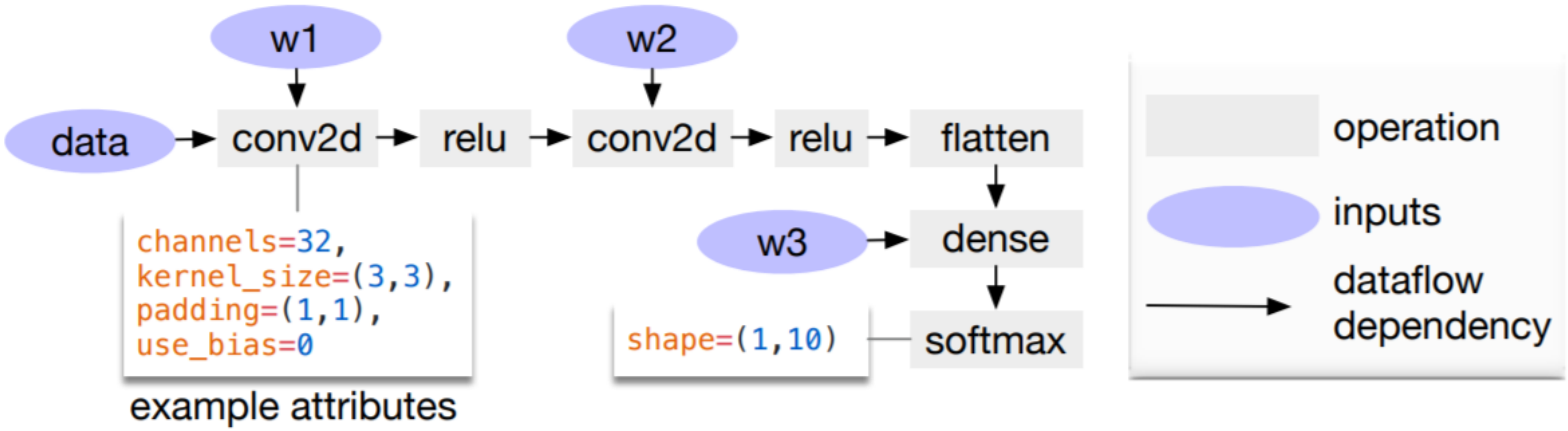
TVM compilation flow



Source: https://tvm.apache.org/

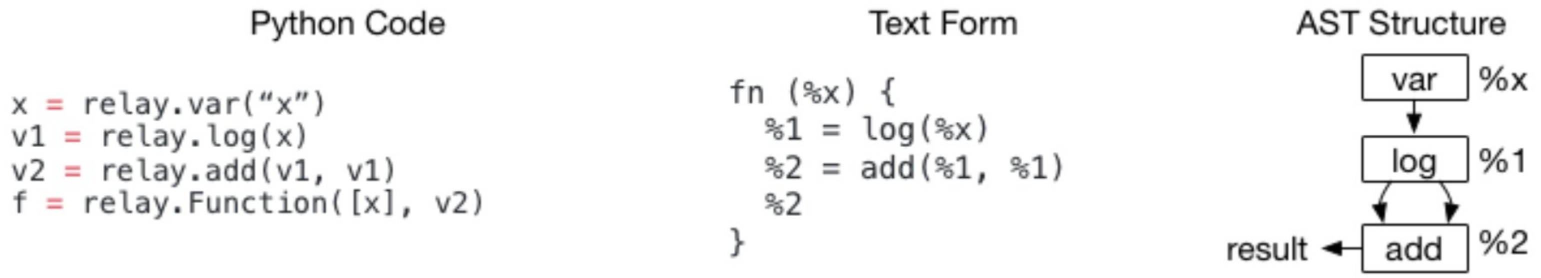
Computational graph

- **Computational graphs** provide a global view of operators without providing implementation details
- Difference w.r.t. "traditional" IRs → intermediate data items are **multi-dimensional tensors** and nodes are **high-level operators**
- Optimization pass → **transforming a computational graph into a different computational graph that is functionally equivalent**



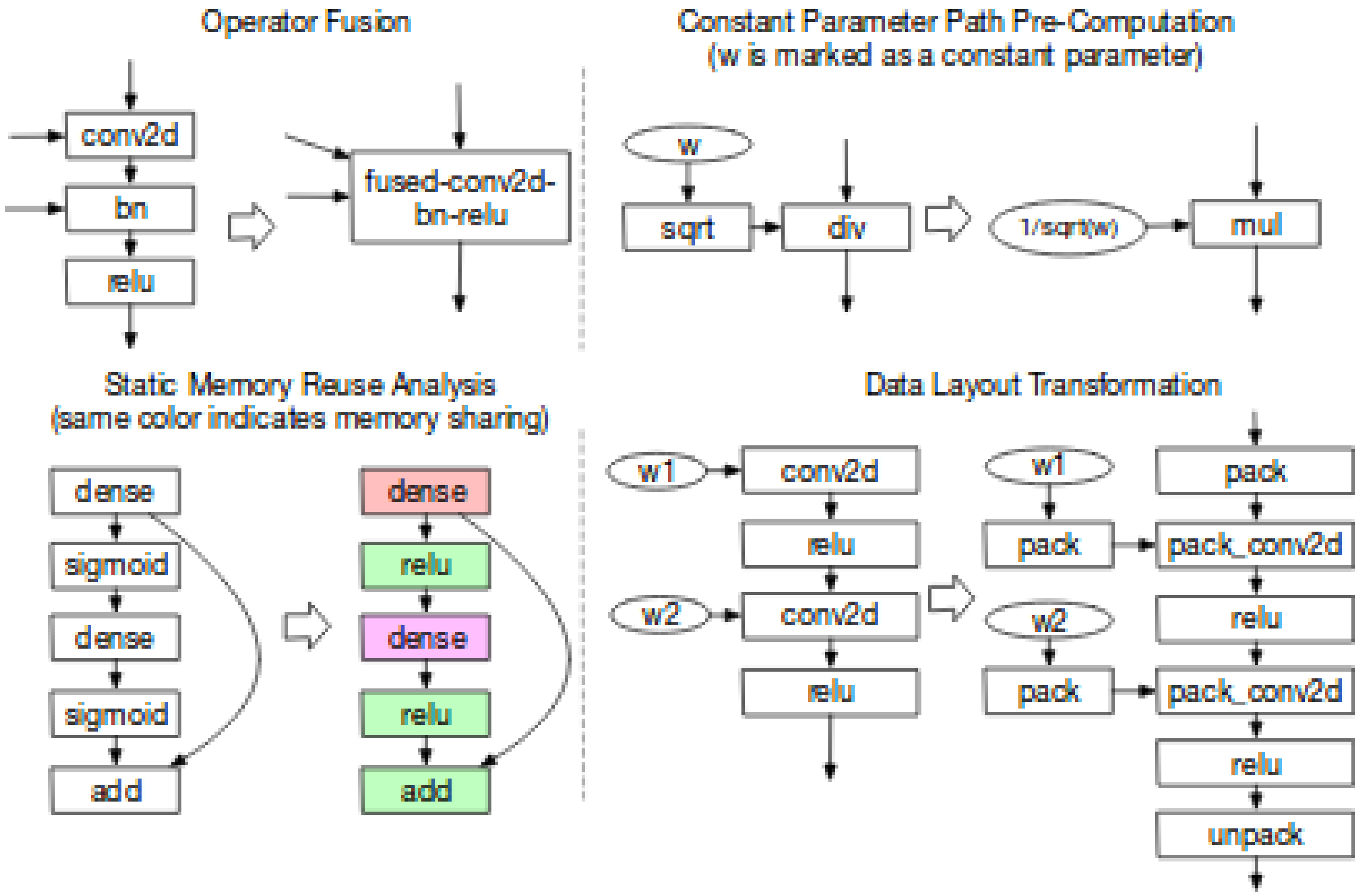
Relay IR

- **Relay** is the IR used to build computational graphs in TVM
- Its authors defines Relay a "***statically typed, purely functional, differentiable IR***"
- This example describes a graph including two nodes:



- Relays uses **functions** to represent graphs
- The text form is similar to "traditional" middle-end IRs

High-level optimizations in TVM



Tensor Expression language: Compute

- VM introduces a **tensor expression language** to support code generation for the graph nodes (kernels)
- Unlike high-level computation graph languages, where the implementation of tensor operations is opaque, each operation is described in an **index formula expression language**
  - Programmers have to specify how each output element (e.g., out[i]) is computed
  - symbolic variables are created by specifying their shapes, and define how the program will be computed

```
import tvm

A = tvm.te.placeholder((n,), name='a')
B = tvm.te.placeholder((n,), name='b’)
C = tvm.te.compute(A.shape, lambda i: A[i] + B[i], name='c')
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Tensor Expression language: Scheduling

- Create high-performance implementations of a tensor expression for each hardware platform is extremely challenging
- An optimized low-level program is always the result of different combinations of **scheduling strategies**
- TVM adopts the **principle of decoupling compute descriptions from schedule optimizations.**
  - **Schedules are the specific rules that lower compute descriptions down to back-end-optimized implementations.**

```
#Generate a schedule
s = tvm.te.create_schedule(C.op)
# Execution plan (C pseudo-code)
tvm.lower(s, [A, B, C], simple_mode=True)
```

Tensor Expression language: Build and execute

- After defining computation and schedule, we can generate an object (called **module**) for execution

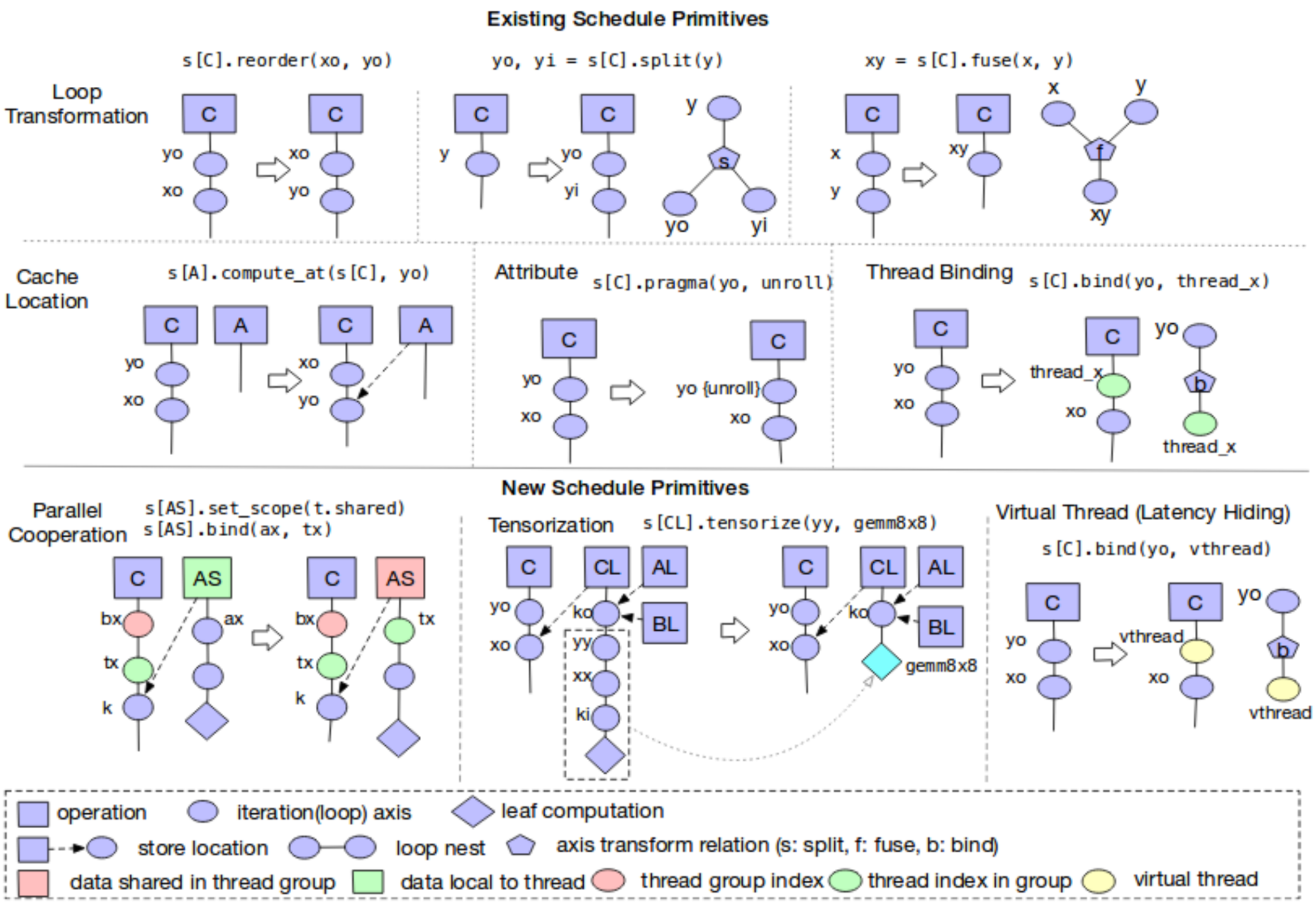
```
# Build the executable module from the schedule
mod = tvm.build(s, [A, B, C])

# Execute the module
mod(a, b, c)

# Save the module
mod.export_library('vector-add.tar’)

# Load the module
loaded_mod = tvm.runtime.load_module('vector-add.tar')
```

Schedule primitives



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Outline

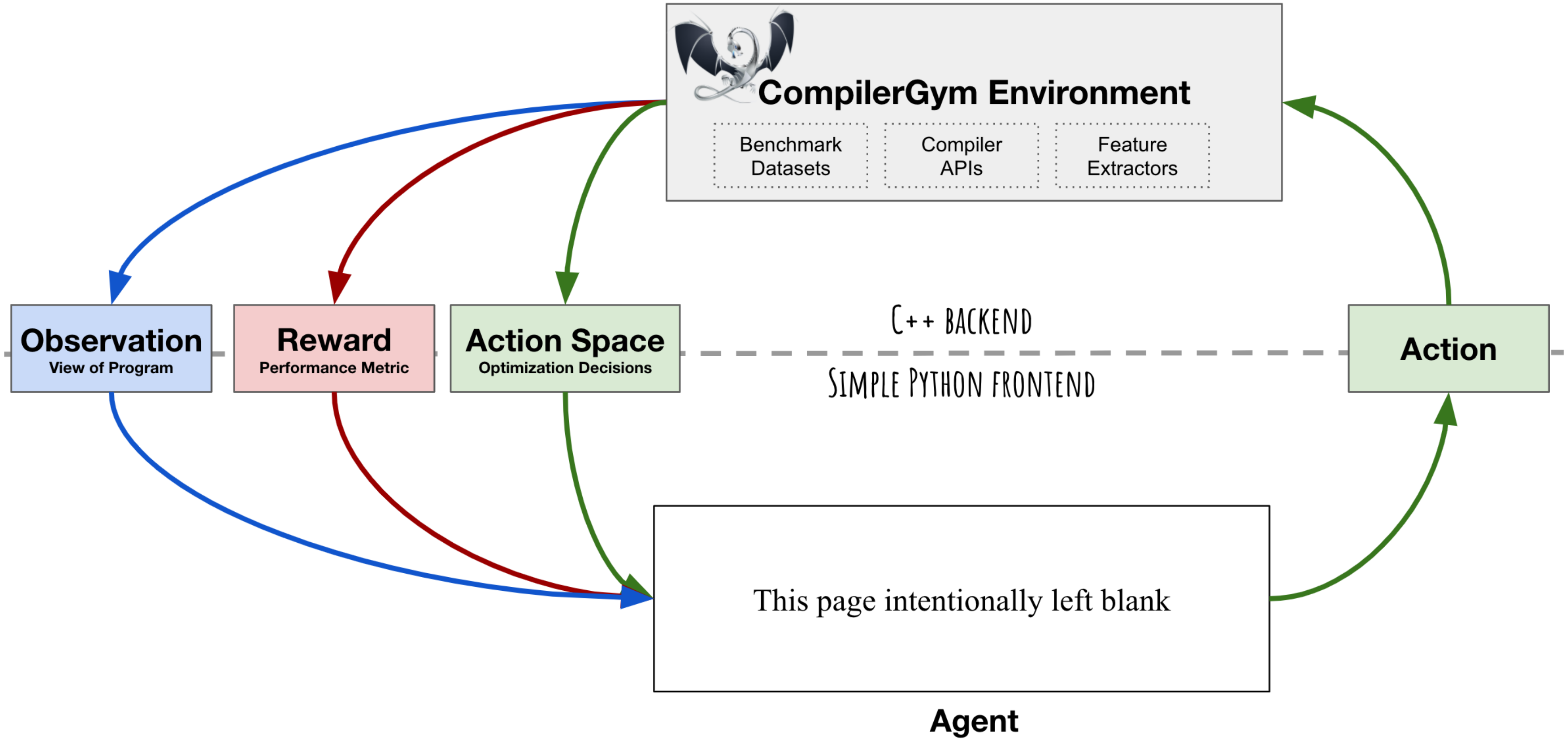
- Introduction: *A quick recap of the compiler architecture*
- Compilers serving AI tasks: *A new era for compilers*
- **AI tasks serving compilers: *A look at the future***
- Conclusion: *Do we really trust these technologies?*

Compilers to optimize DL ←→ DL to optimize compilers

- DL compilers aim at adapting compiler principles to the DL domain, but …
- … **could we also use DL to optimize compiler optimization passes???**

- **CompilerGym** is a toolkit for applying reinforcement learning to compiler optimization (https://compilergym.com/)
  - Built by Facebook AI
  - Based on OpenAI Gym framework
  - three compiler problems: phase ordering using LLVM, flag tuning using GCC, and loop nest generation using CUDA

CompilerGym



An interesting case study: Phase ordering

- **Phase ordering** is the selection of the right set of compiler optimizations for a particular code segment
  - **This is a hard problem!**
- In LLVM, an agent can choose an optimization pass to run next from a set of **100+ distinct optimizations**
- LLVM environments provided by CompilerGym support optimizing for *runtime*, *compiler binary size*, and *instruction count*

- Testing environment provided by Facebook AI: https://colab.research.google.com/github/facebookresearch/CompilerGym/blob/development/examples/getting-started.ipynb?fbclid=IwAR0zbD4MzfcErpVzRwJrb61d4sAq33HBqfOpNvr1h0Xjev_rOh9jSA-LbfM

Can we compile a program using an LLM?

- A prompt for GPT-4:

```
I want to compile a C file for a RISC-V target using the LLVM toolchain.
The clang flags are: -O3 --target=riscv32 -march=rv32imfc -mabi=ilp32f -S -
emit-llvm
Generate the LLVM IR for this file and provide me only the resulting LLVM IR
code with no explanation:
***
{code}
***
```

A classic example: Matrix multiplication

```
void matmul(float* out, float* in1, float* in2, int m, int n, int o) {
    // Initialize the output matrix with zeros
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < o; j++) {
            out[i * o + j] = 0.0f;
        }
    }

    // Perform matrix multiplication
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < o; j++) {
            for (int k = 0; k < n; k++) {
                out[i * o + j] += in1[i * n + k] * in2[k * o + j];
            }
        }
    }
}
```

Answer provided by GPT-4

```
define void @matmul(float* %out, float* %in1, float* %in2, i32 %m, i32 %n, i32 %o) {
entry:
  %0 = alloca float*, align 4
  %1 = alloca float*, align 4
  %2 = alloca float*, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store float* %out, float** %0, align 4
  store float* %in1, float** %1, align 4
  store float* %in2, float** %2, align 4
  store i32 %m, i32* %3, align 4
  store i32 %n, i32* %4, align 4
  store i32 %o, i32* %5, align 4
  ...
```

????

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Code generated using the LLVM 18 toolchain

```
define dso_local void @matmul(ptr nocapture noundef %out, ptr nocapture noundef readonly
%in1, ptr nocapture noundef readonly %in2, i32 noundef %m, i32 noundef %n, i32 noundef
%o) local_unnamed_addr #0 {
entry:
  %cmp63 = icmp sgt i32 %m, 0
  %cmp261 = icmp sgt i32 %o, 0
  %or.cond = and i1 %cmp63, %cmp261
  br i1 %or.cond, label %for.cond14.preheader.lr.ph.split.us, label %for.cond.cleanup11

...
```

Can we modify the compiler toolchain using LLMs?

```
(define_insn "abssf2"
  [(set (match_operand:SF 0 "register_operand"          "=d")
        (abs:SF (match_operand:SF 1 "register_operand"   "d")))]
  "TARGET_USE_FPU"
  {
    static const char * asmcode[] =
    {
      "insert\t%0, %1, 0, 31, 1",
      "abs.f\t%0,%1"
    };
    if (TRIC_18UP) return asmcode[1]; else return asmcode[0];
  }
  [(set_attr "pipe" "fp2")])
```

This is a pattern in the GCC backend for the Tricore architecture:
'''{code}'''
Can I programmatically change the value of the "pipe" attr based on the same condition defined for the asmcode choice in the code snippet?
Consider the following "pipe" values associated to instructions (format is "instruction name -> attr value"):
- insert -> ip
- abf.f -> fp2
...

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

```
(define_insn "abssf2"
  [(set (match_operand:SF 0 "register_operand"          "=d")
        (abs:SF (match_operand:SF 1 "register_operand"   "d")))]
  "TARGET_USE_FPU"
  {
    static const char * asmcode[] =
    {
      "insert\t%0, %1, 0, 31, 1",
      "abs.f\t%0,%1"
    };
    if (TRIC_18UP) return asmcode[1]; else return asmcode[0];
  }
  [(set_attr "pipe"
        (if_then_else (symbol_ref "TRIC_18UP")  ;; Check if TRIC_18UP is defined
                      (const_string "ip")
                      (const_string "fp2")))])
```

```
(define_insn "abssf2"
  [(set (match_operand:SF 0 "register_operand"          "=d")
        (abs:SF (match_operand:SF 1 "register_operand"   "d")))]
  "TARGET_USE_FPU"
  {
    static const char * asmcode[] =
    {
      "insert\t%0, %1, 0, 31, 1",
      "abs.f\t%0,%1"
    };
    if (TRIC_18UP) return asmcode[1]; else return asmcode[0];
  }
  [(set (attr "pipe")
        (if_then_else (symbol_ref "TRIC_18UP")  ;; Check if TRIC_18UP is defined
                      (const_string "ip")
                      (const_string "fp2")))])
```

Is GPT-4 aware of the difference between these constructs?

Key Differences:

Feature	`set_attr "type"`	`set (attr "type")`
Purpose	Sets an attribute outside of RTL as a static rule	Sets an attribute dynamically as part of RTL
Usage	For global/static instruction attribute setting	For dynamic per-instruction attribute setting
Context	Typically outside the RTL as part of the instruction pattern	Inside the RTL to modify attributes dynamically
Example Usage	`(set_attr "type" "arith")`	`(set (attr "type") (const_string "arith"))`

Summary:

- `set_attr` is used to globally define attributes for an instruction outside its RTL, useful for the instruction's type, scheduling, and pipeline behavior.
- `set (attr "type")` is used dynamically within the instruction's RTL to set or modify attributes as part of the operation during instruction selection or scheduling.

Phase Ordering with LLM (1/2)

- Investigated by Meta AI [1]
- Model architecture: Llama 2 7B
- Dataset: 1,000,000 LLVM IR functions (373M tokens)
- Training: for 30,000 steps on 64 V100s for (620 GPU days)

- Comparison with the state-of-the-art:

	additional compilations	functions improved	functions regressed	instructions saved	instructions regressed	overall improvement
Autotuner [3]	2,522,253,069	6,764	0	30,948	0	5.03%
AutoPhase [2]	4,500,000	1,558	8,400	6,522	32,357	-3.85%
Coreset-NVP [3]	442,747	3,985	6,072	16,064	28,405	-1.88%
Our Approach	0	4,136	526	21,935	3,095	3.01%

[1] Cummins, Chris, et al. "Large language models for compiler optimization." arXiv preprint arXiv:2309.07062 (2023).
[2] Haj-Ali, Ameer, et al. "AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning." Proceedings of Machine Learning and Systems 2 (2020): 70-81.
[3] Liang, Youwei, et al. "Learning Compiler Pass Orders using Coreset and Normalized Value Prediction." *International Conference on Machine Learning*. PMLR, 2023.

Phase Ordering with LLM (2/2)

```
define i32 @f1(i8 %0) {
 %2 = alloca i32, align 4
 %3 = alloca i8, align 1
 store i8 %0, i8* %3, align 1
 %4 = load i8, i8* %3, align 1
 %5 = zext i8 %4 to i32
 %6 = icmp sge i32 %5, 65
 br i1 %6, label %7, label %15

7:
 %8 = load i8, i8* %3, align 1
 %9 = zext i8 %8 to i32
 %10 = icmp sle i32 %9, 90
 br i1 %10, label %11, label %15

11:
 %12 = load i8, i8* %3, align 1
 %13 = zext i8 %12 to i32

<snip 21 lines...>

33:
 %34 = load i32, i32* %2, align 4
 ret i32 %34
}
```

Input → 39 instructions

```
define i32 @f1(i8 %0) {
 %2 = zext i8 %0 to i32
 %.off = add i8 %0, 191
 %3 = icmp ult i8 %.off, 26
 br i1 %3, label %4, label %6

4:
 %5 = add nsw i32 %2, 191
 br label %10

6:
 %.reload16.off = add nsw i32 %2, 159
 %7 = icmp ult i32 %.reload16.off, 26
 br i1 %7, label %10, label %8

8:
 %9 = icmp eq i8 %0, 32
 %. = select i1 %9, i32 26, i32 1
 br label %10

10:
 %.0.reg2mem.0 = phi i32 [%5, %4],
  [%., %8], [%.reload16.off, %6]
 ret i32 %.0.reg2mem.0
}
```

Autotuner → 14 instructions
`-reg2mem -instcombine -Os -O1`
<u>**Result after testing 26k different pass orderings**</u>

```
define i32 @f1(i8 %0) {
 %2 = zext i8 %0 to i32
 %.off = add i8 %0, 191
 %3 = icmp ult i8 %.off, 26
 br i1 %3, label %6, label %._crit_edge

._crit_edge:
 %.off24 = add i8 %0, 159
 %4 = icmp ult i8 %.off24, 26
 br i1 %4, label %6, label %._crit_edge9

._crit_edge9:
 %5 = icmp eq i8 %0, 32
 %spec.select = select i1 %5,
  i32 26, i32 1
 ret i32 %spec.select

6:
 %.sink = phi i32 [191, %1],
  [159, %._crit_edge]
 %7 = add nsw i32 %.sink, %2
 ret i32 %7
}
```

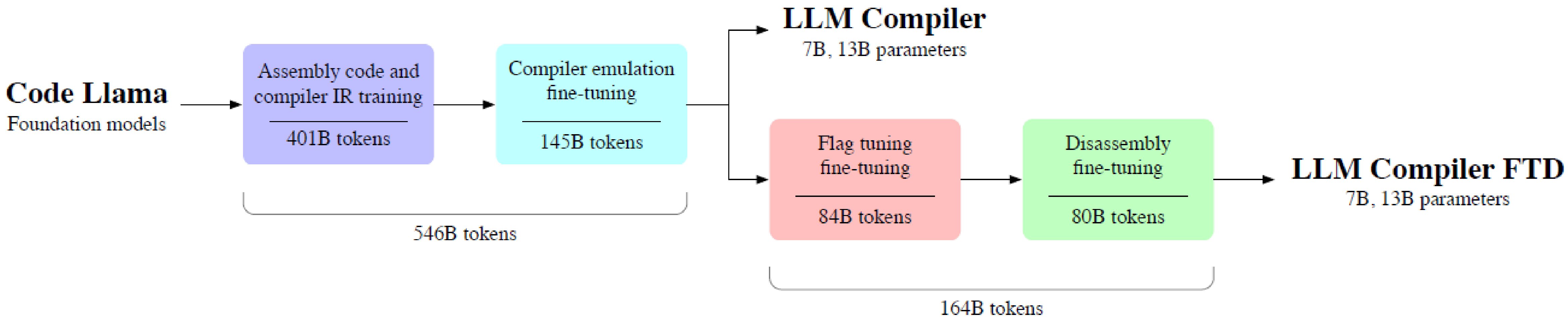
LLM →13 instructions
`-reg2mem -simplifycfg -mem2reg -jump-threading –Os`
<u>**This pass list appears 5 times in the training set**</u>

ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

LLM Compiler: The Holy Grail (?) of LLM compiler research

- LLM Compiler is a family of foundation models that have already been trained to understand the semantics of compiler IRs and assemblies and to emulate the compiler
- At the same time, it simplifies fine-tuning with minimal data for specific downstream compiler optimization tasks

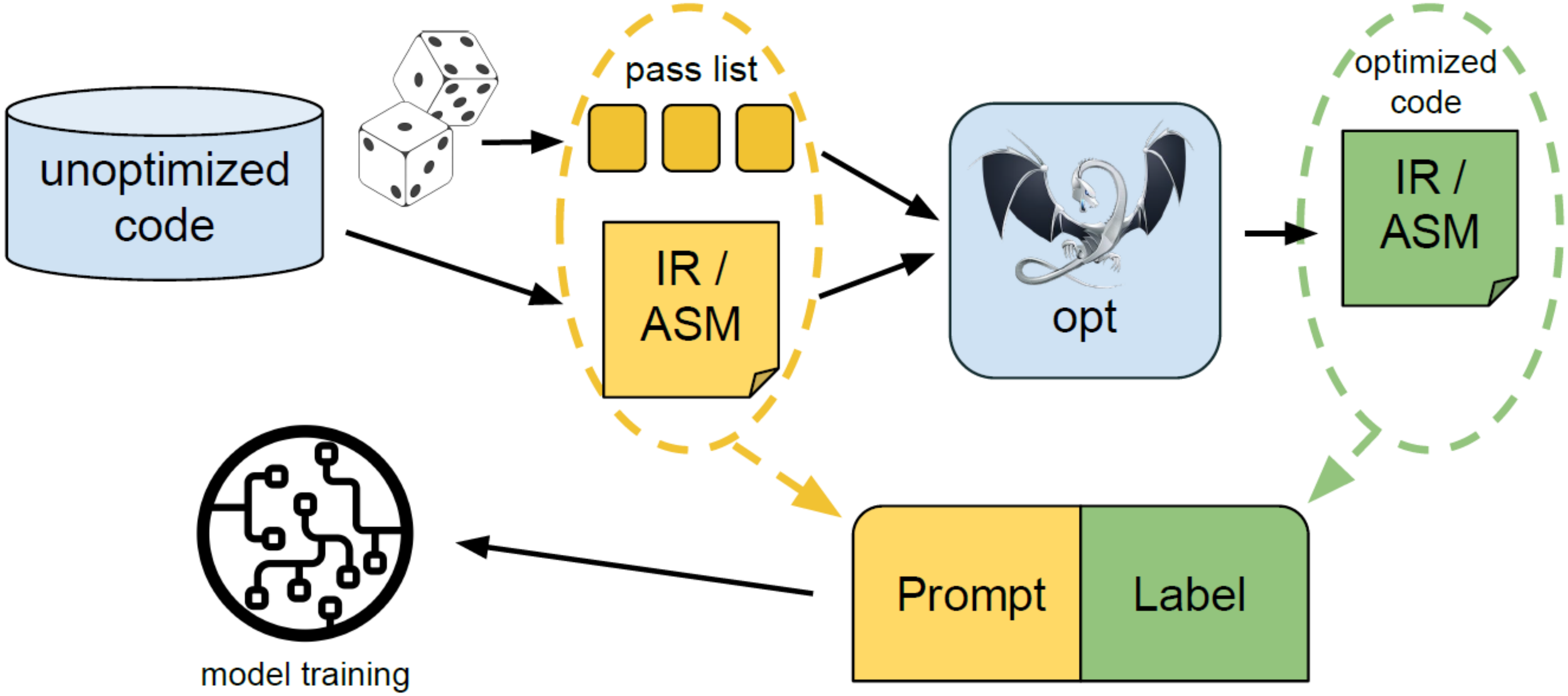
- Trained on IR and assembly code generated by LLVM version 17.0.6.



Cummins, Chris, et al. "Meta Large Language Model Compiler: Foundation Models of Compiler Optimization." *arXiv preprint arXiv:2407.02524* (2024).

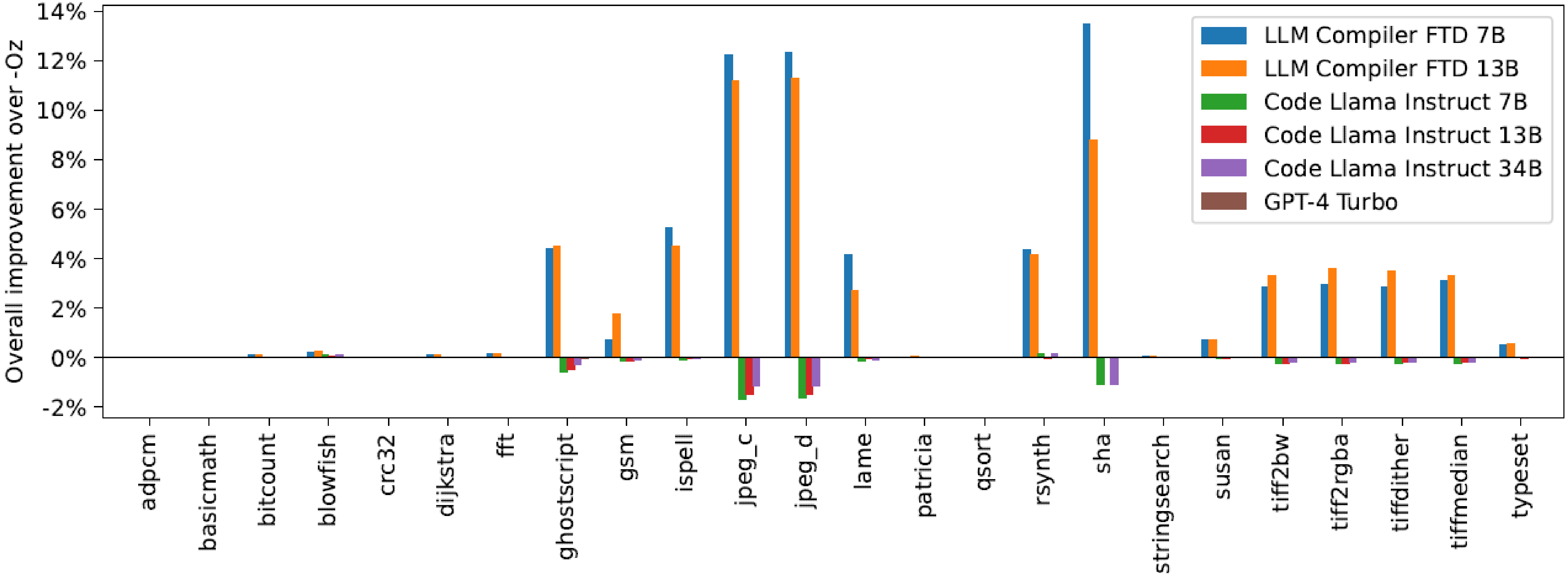
Compiler emulation fine-tuning

- Compiler emulation dataset → applying random lists of between 1 and 50 optimization passes to unoptimized programs



Cummins, Chris, et al. "Meta Large Language Model Compiler: Foundation Models of Compiler Optimization." *arXiv preprint arXiv:2407.02524* (2024).

Reducing the binary size (baseline –Oz)



Cummins, Chris, et al. "Meta Large Language Model Compiler: Foundation Models of Compiler Optimization." *arXiv preprint arXiv:2407.02524* (2024).

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Outline

- Introduction: *A quick recap of the compiler architecture*
- Compilers serving AI tasks: *A new era for compilers*
- AI tasks serving compilers: *A look at the future*
- **Conclusion: *Do we really trust these technologies?***

Conclusion

- Most of these papers are published on arXiv → Even if the quality is good and code is provided, a peer-review process is missing
- What are the real advantages of using LLMs?
- Can we ask really people to send their IPs to external servers?

THANK YOU FOR YOUR ATTENTION!!!!

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA