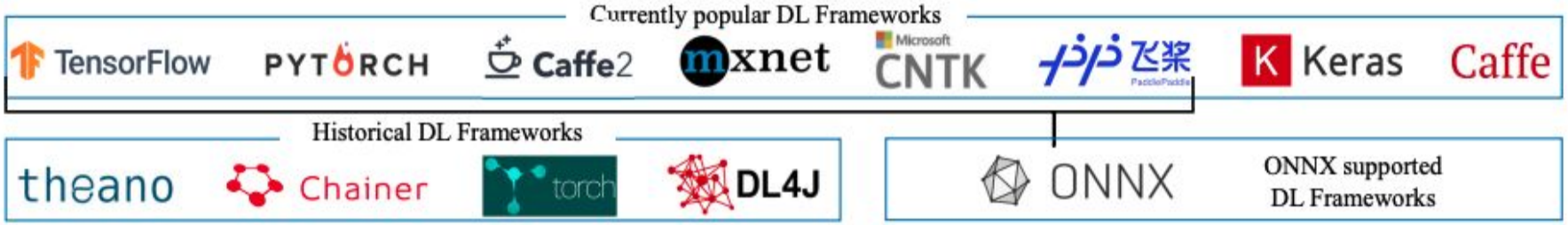
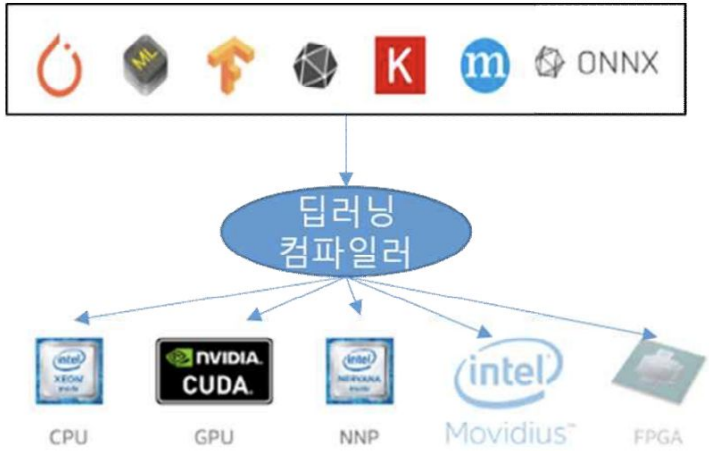
# The Deep Learning Compiler

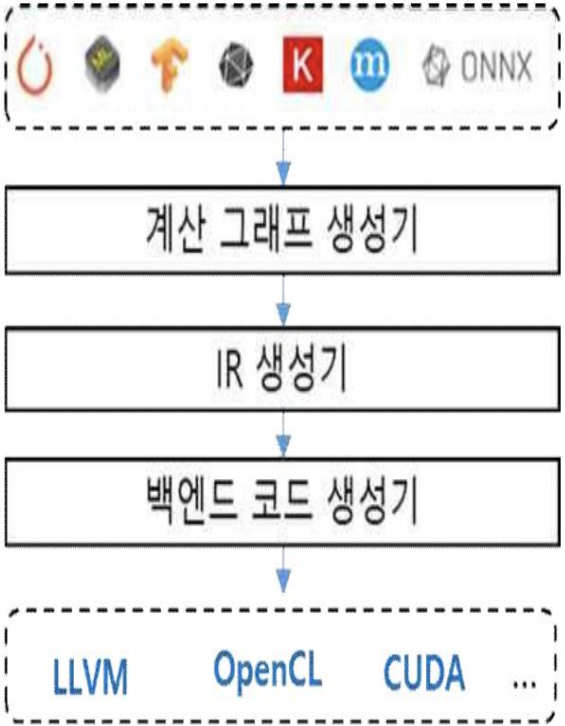
Tae Young Lee

# DL Framework landscape



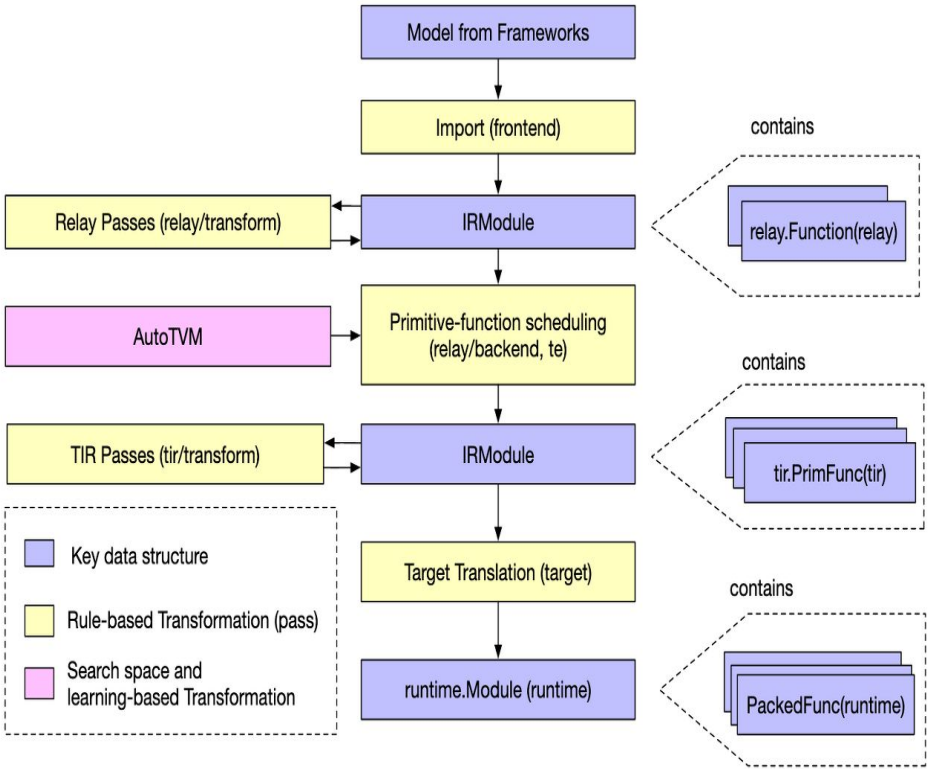




- 딥러닝 컴파일러는, 다양한 딥러닝 플랫폼에서 학습된 딥러닝 모델을 입력으로 받아,
- 특정 하드웨어에서 동작 가능한 머신 코드(또는, 백엔드 코드)를 자동으로 생성합니다.
- 최근 제안된 XLA, TVM, Glow 와 같은 딥러닝 컴파일러들은,
- TensorFlow, Pytorch, MxNet, ONNX 등의 프레임워크로 작성된 모델을 입력으로 하여,
- CPU 및 GPU 용 백엔드 코드를 생성하고 있습니다.
    - CPU 용 백엔드 코드 : llvm
    - GPU 용 백엔드 코드 : CUDA, OpenCL, Metal 등




- **계산 그래프 생성기**
  - 계산 그래프 생성기는, 다양한 딥러닝 플랫폼에서 생성된 딥러닝 모델을 로딩하여, 그래프 구조로 재구성한 뒤,
  - 모델에서 명시한 연산자들을, 프리미티브 연산자로 변환하여, 계산 그래프를 생성합니다.
  - 이후, 불필요한 노드 제거, 연산자 결합(Operator Fusion), 양자화(Quantization), 상수 폴딩(constant-folding)과 같은, 그래프 최적화를 기법들을 적용하여,
  - 최종적으로 최적화 된 계산 그래프를 생성합니다.
- **IR 생성기**
  - **계산 그래프 생성기로부터 생성된 그래프는, 하드웨어에 독립적인 정보만 표현 할 수 있습니다.**
    - ex) 연산자 간 입출력 관계, 연산자의 입출력 데이터 값, etc
  - **그러나 컴파일러는, 딥러닝 모델의 실행 성능을 높이기 위해,**
  - **모델이 동작할 하드웨어의 아키텍처를 고려한, '하드웨어 의존적 최적화'를 필수적으로 수행해야합니다.**
  - **따라서, 계산 그래프를 '하드웨어 의존적 최적화'에 적용 가능한 형태인,**
  - **중간 표현 (IR: Intermediate Representation)으로 변환하는 과정이 필요합니다.**
    - 참고로, IR 은 입출력 변수(또는 Tensor)들에 대한 선언과,
    - 변수들을 사용하는 연산식에 대한 정보를 포함하고 있습니다.
  - **컴파일러는, 생성된 IR 을 바탕으로 메모리 할당, 병렬화, 연산식 실행 순서 결정 등의 '하드웨어 의존적 최적화' 작업을 수행한 뒤, 코드를 생성합니다.**
  - **요약하면, IR 생성기는 컴파일러가 하드웨어 의존적 최적화를 용이하게 수행하도록, 계산 그래프를 IR 로 변환하는 역할을 합니다.**
- **백엔드 코드 생성기**
  - **백엔드 코드 생성기는, IR 을 기반으로, 딥러닝 모델 워크로드가 배치될 타겟 하드웨어 (CPU, GPU, TPU 등)의 아키텍처에 최적화 된 백엔드 코드를 생성합니다.**
  - 즉, 타겟 하드웨어 별로, 병렬로 처리 가능한 데이터의 크기, 캐시 크기, 효율적으로 수행 가능한 딥러닝 오퍼레이터의 타입, 실행 순서 등을 참조하여,
  - 스케쥴 정보를 생성한 뒤, 해당 하드웨어에 최적화 된 백엔드 코드를 생성합니다.

# TVM 실행 흐름




- **DL framework로부터 생성한 모델을 import 한 뒤, relay 를 통해, 해당 모델을 TVM 의 IRModule 로 변환합니다.**
  - IRModule에는 모델을 나타낼 수 있는 다양한 함수들이 포함되어 있습니다.
- **IRModule 을 점점 Lowering 하는 Transformation 과정을 거칩니다.**
  - IRModule 이 원하는 하드웨어(메모리 구조, 데이터 타입 등)에 알맞게 최적화될 수 있도록 Tensorize 시킵니다.
  - 이를 위해 Quantization, Graph-Packing 등, 다양한 작업들이 포함되며,
  - 이때, AutoTVM 이 사용됩니다.
- **타깃 하드웨어에서 잘 작동하는 코드를 생성하기 위해, Translation 과정을 거칩니다.**
  - 예를 들어, 타깃 하드웨어가 GPU 라면 [CUDA/OpenCL/Metal 등], CPU라면 [llvm] 코드를 생성합니다.
  - 이는, TVM 의 runtime.Module 에 encapsulate 되어있고,
  - 이를 통해, 타깃 하드웨어에 쉽게 export / load / execute 시킬 수 있습니다.
- **runtime.Module 를 로드하여, 타깃 하드웨어에서 최적화 된 모델을 동작시킵니다.**

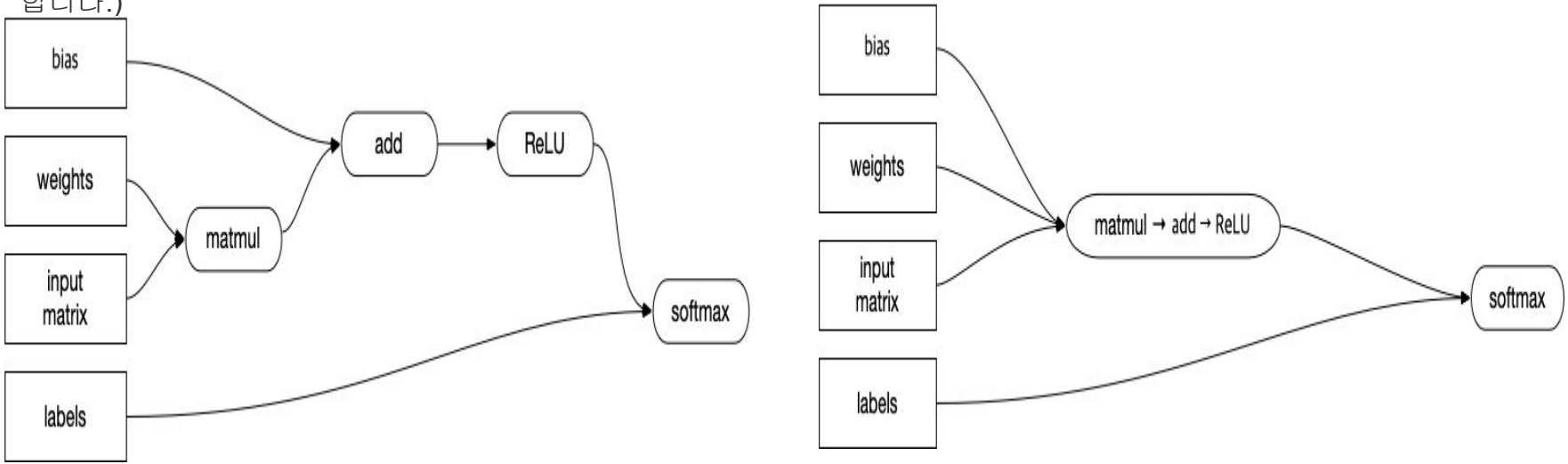
# TFLite, TensorRT, TVM의 차이

- **TFLite**
  - TensorFlow(Keras)의 모델만 최적화가 가능합니다.
  - 여러 하드웨어에서 추론이 가능합니다.
  - 타겟 하드웨어에 따른 추가적인 최적화는 없습니다.
- **TensorRT**
  - 다양한 프레임워크의 모델 최적화가 가능합니다.
  - NVIDIA GPU 만 추론 하드웨어로 사용 가능합니다.
  - 추론에 사용하려는 NVIDIA GPU 종류에 따라 추가적으로 최적화할 수 있습니다.
- **TVM**
  - 다양한 프레임워크의 모델 최적화가 가능합니다.
  - 여러 하드웨어서 추론이 가능합니다.
  - 추론에 사용하려는 하드웨어 종류에 따라 추가적으로 최적화할 수 있습니다.

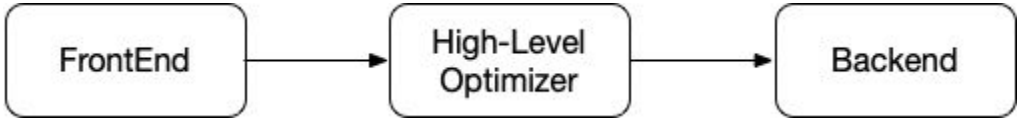
# XLA 란?

우선 XLA (Accelerated Linear Algebra)는 Tensorflow의 서브 프로젝트로 그래프 연산의 최적화 / 바이너리 사이즈의 최소화 등을 목적으로 하는 컴파일러입니다.

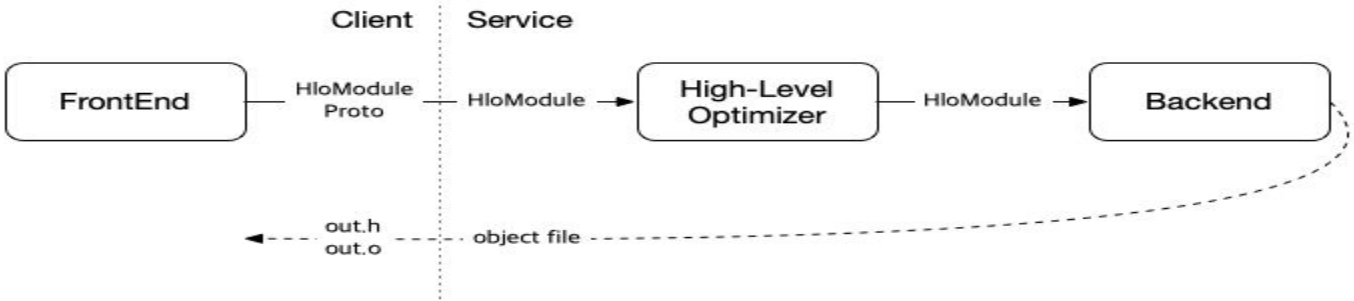
결과를 먼저 말씀드리자면, XLA를 활용하는 것을 통해 그래프 연산 과정에 필요한 임시 버퍼도 덜 사용할 수 있고, 그래프 연산 속도도 단축시킬 수 있습니다. (구글 측 자료에 의하면 Nvidia GPU를 사용할 때 50%까지도 성능 향상이 있었다고 합니다.)



# XLA




XLA도 일종의 컴파일러이기 때문에 일반적인 컴파일러 들이 동작하는 것처럼 FrontEnd (Parser) / MiddleEnd (High-level Optimizer) / BackEnd (Target Code Generator)의 구조를 가지고 있으며, 주어진 타겟 플랫폼에 맞춰 그래프를 최적화 한 후 실행가능한 형태의 무언가를 생성하는 역할을 수행
BackEnd에서는 LLVM을 사용하고 있으며, LLVM과 마찬가지로 JiT compilation과 AoT compilation 모두를 지원하는게 특징입니다. 두가지 중 어떤 방식을 사용하느냐에 따라 실제 사용 시나리오가 달라지게 되며, AoT compilation을 사용하는 경우 아래와 같이 컴파일의 결과로 object file 및 이를 사용하기 위한 helper class에 대한 header file이 생성됩니다.

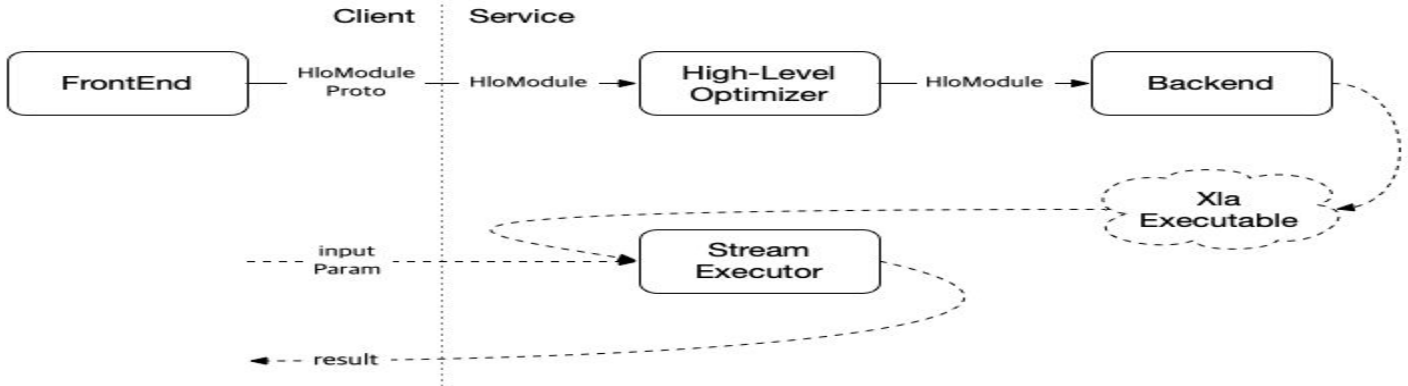


# AOT

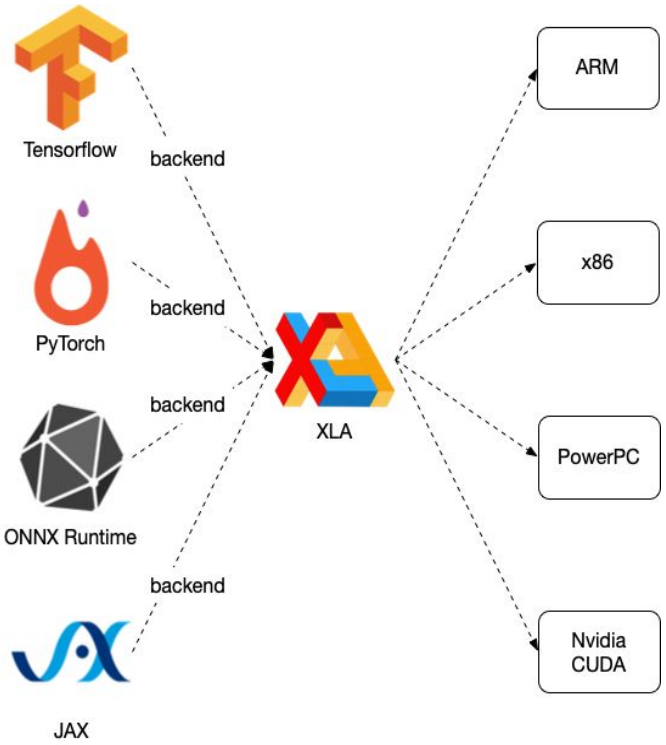
tfcompile은 Ahead-of-Time(AOT)이 TensorFlow 그래프를 실행 가능한 코드로 컴파일하는 독립 실행형 도구입니다. 이를 통해 총 바이너리 크기를 줄이고 일부 런타임 오버헤드를 방지할 수 있습니다. tfcompile은 일반적으로, 추론 그래프를 모바일 기기용 실행 코드로 컴파일하는 데 사용됩니다.

실행 바이너리를 만들기 위해 XLA 라이브러리나 Tensorflow 라이브러리 등을 포함할 필요가 없기 때문에 최종 타겟 바이너리 사이즈를 굉장히 작게 만들 수 있으나 기본 CPU backend의 경우 convolution, matmul 등의 연산에 대해 XLA runtime function을 호출하는 코드를 생성하도록 되어 있기 때문에 일부 XLA runtime function들과 링크되어야 할 수도 있습니다.

JiT compilation을 사용하는 경우는 compile의 결과로 XlaExecutable 데이터가 생성되며, StreamExecutor를 통해 input parameter를 feeding하고 그래프를 실행시킬 수 있는 형태입니다.



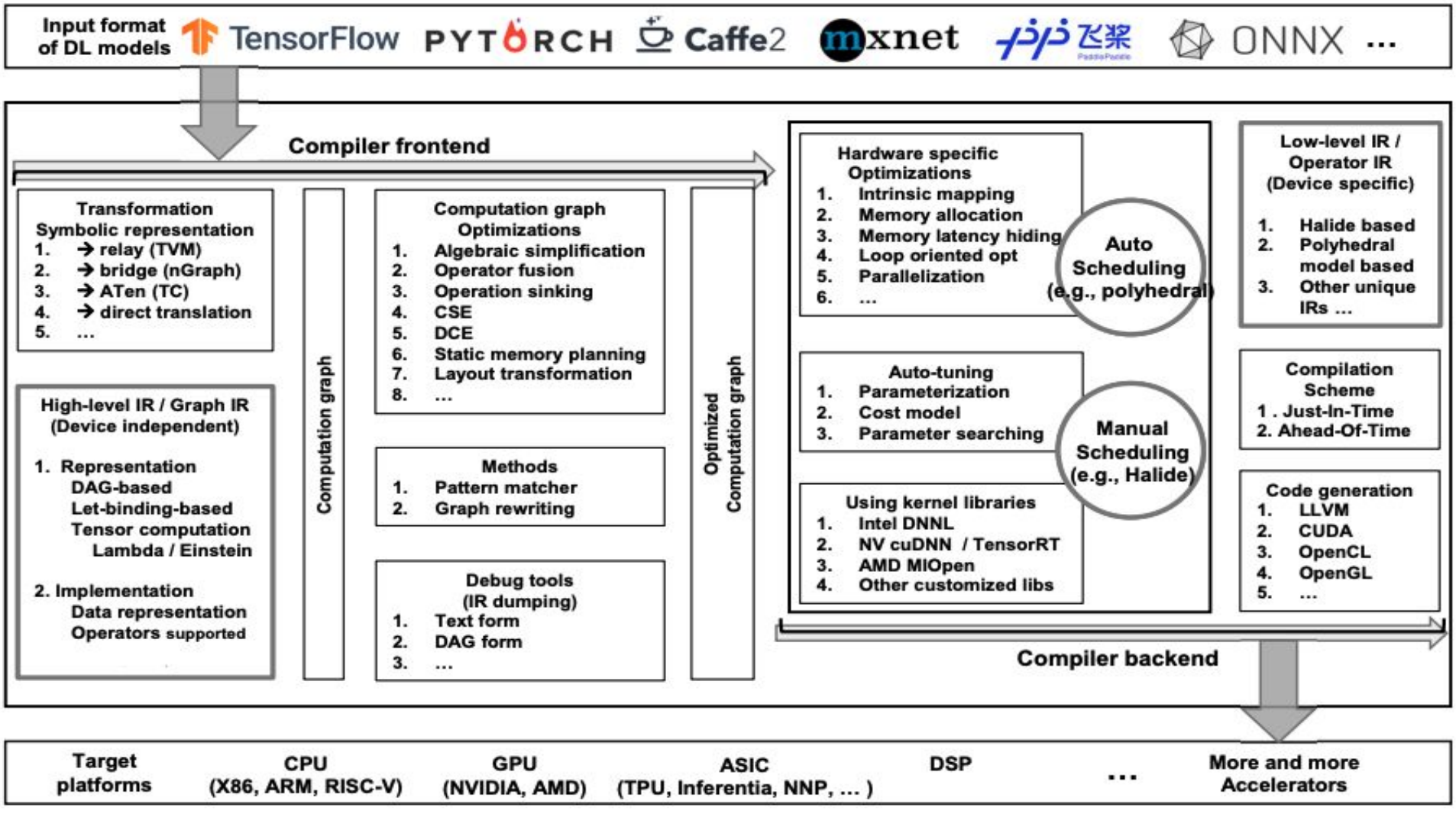
# JitCompilation

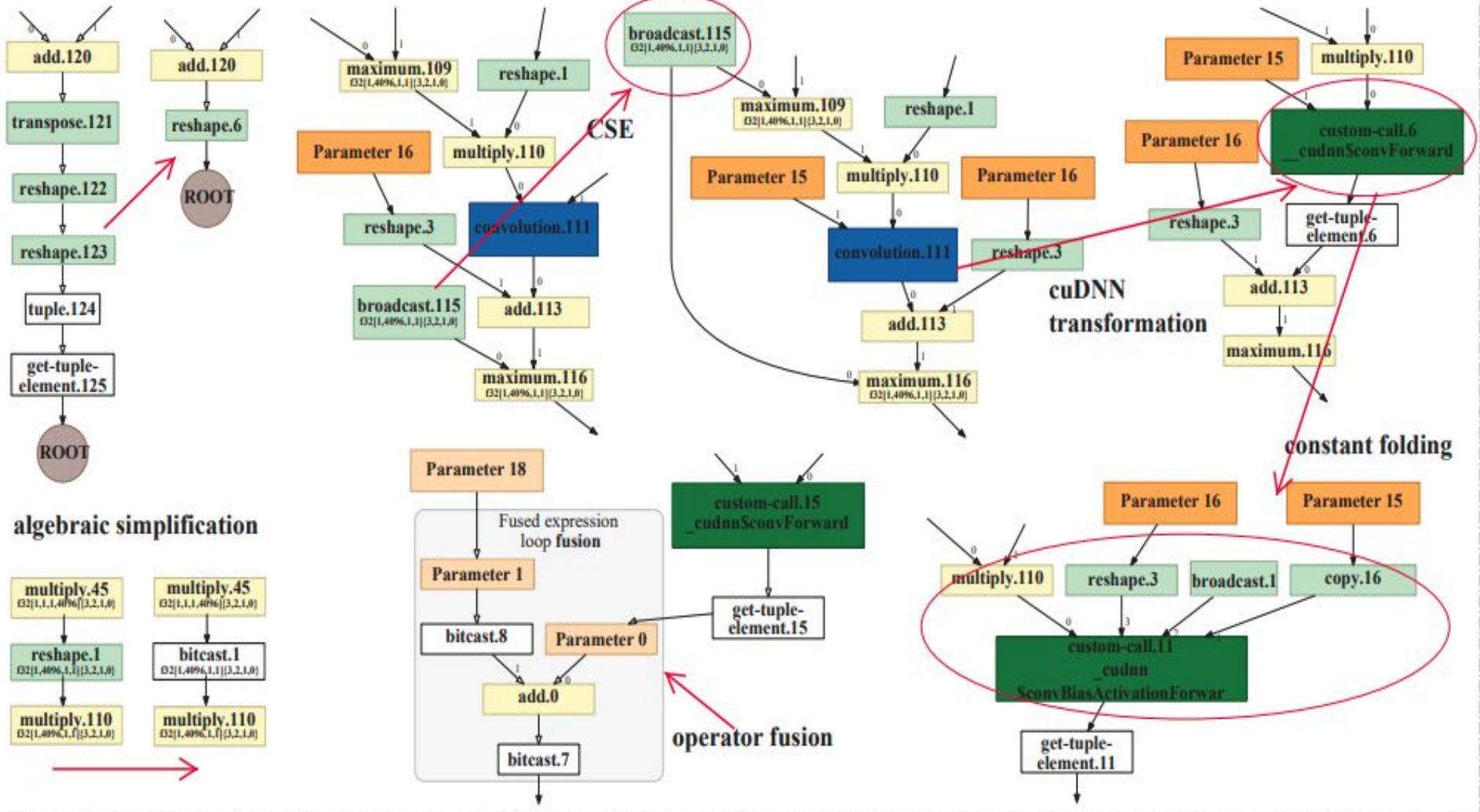


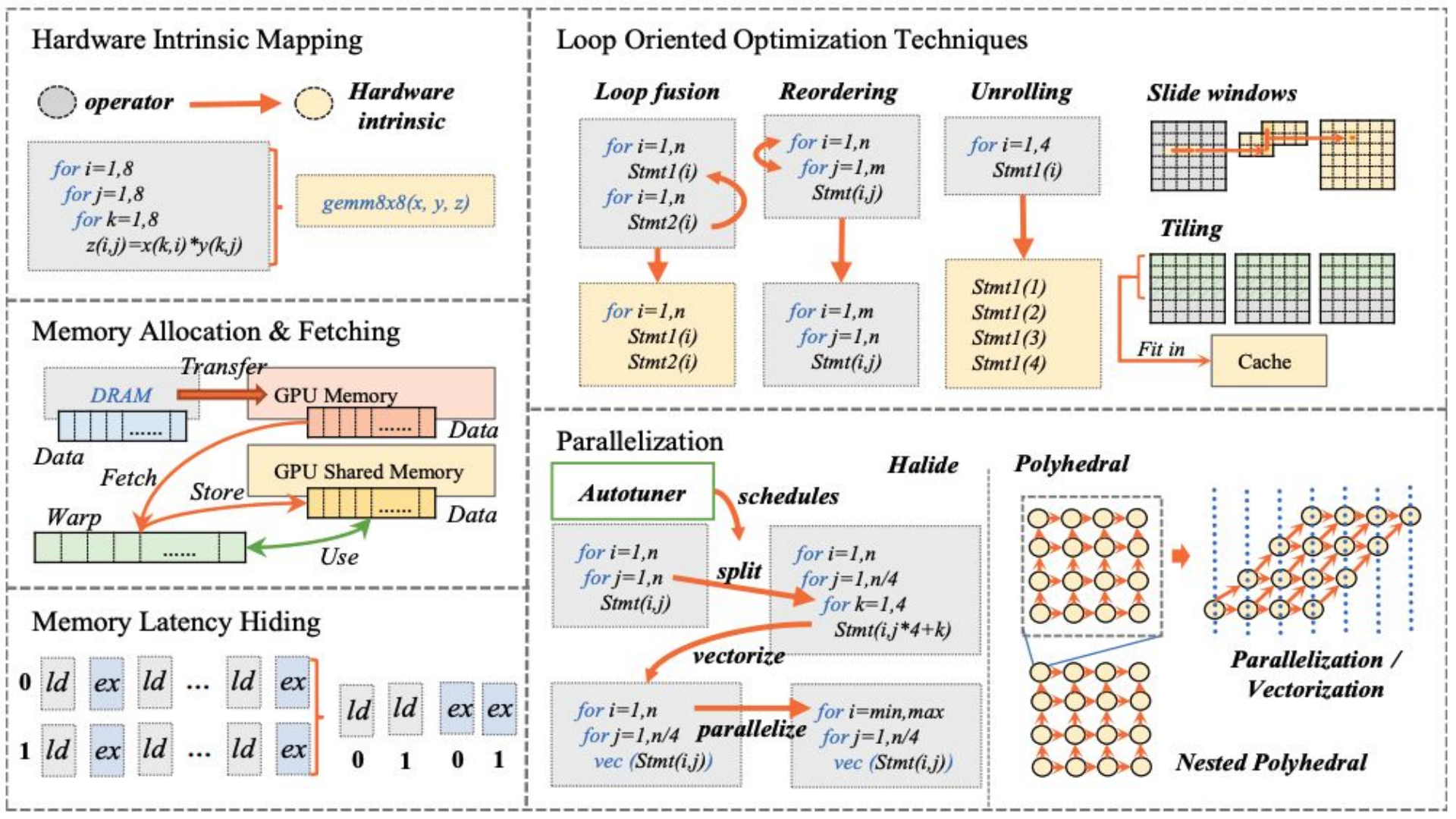

JitCompilation을 사용하는 경우 런타임에 그래프를 입력으로 받아 실행할 수 있다는 장점이 있으나 애플리케이션이 XLA 런타임을 포함해야하기 때문에 eigen, protobuf, LLVM 등등 의존성들이 줄줄이 엮인다는 단점이 있습니다. (타겟 플랫폼에 해당 라이브러리들을 줄줄이 포팅해야할 수도 있습니다.)

애플리케이션을 배포하기 전 단계에 타겟 플랫폼이나 네트워크 형태에 대해 미리 알고 있는 경우에는 AoT compilation을 활용하는 방안을 고려해볼 수 있으며, 애플리케이션은 네트워크를 모르는 상태로 배포되어야 하는 환경이라면 JiT compilation을 사용해야합니다.

Tensorflow 외에도 다양한 머신러닝 프레임웍의 백엔드로 XLA를 활용할 수 있으며, PyTorch를 위해선 PyTorch XLA, ONNX (Open Neural Network Exchange )를 위해선 onnx xla 프로젝트가 개발되고 있습니다.


Input format of DL models
TensorFlow
PYTORCH
Caffe2
mxnet
飞桨
ONNX
...
Compiler frontend
Transformation
Symbolic representation
1. → relay (TVM)
2. → bridge (nGraph)
3. → ATen (TC)
4. → direct translation
5. …
High-level IR / Graph IR
(Device independent)
1. Representation
DAG-based
Let-binding-based
Tensor computation
Lambda / Einstein
2. Implementation
Data representation
Operators supported
Computation graph
Computation graph
Optimizations
1. Algebraic simplification
2. Operator fusion
3. Operation sinking
4. CSE
5. DCE
6. Static memory planning
7. Layout transformation
8. …
Methods
1. Pattern matcher
2. Graph rewriting
Debug tools
(IR dumping)
1. Text form
2. DAG form
3. …
Optimized Computation graph
Hardware specific Optimizations
1. Intrinsic mapping
2. Memory allocation
3. Memory latency hiding
4. Loop oriented opt
5. Parallelization
6. …
Auto Scheduling (e.g., polyhedral)
Auto-tuning
1. Parameterization
2. Cost model
3. Parameter searching
Manual Scheduling (e.g., Halide)
Using kernel libraries
1. Intel DNNL
2. NV cuDNN / TensorRT
3. AMD MIOpen
4. Other customized libs
Low-level IR / Operator IR
(Device specific)
1. Halide based
2. Polyhedral model based
3. Other unique IRs …
Compilation Scheme
1 . Just-In-Time
2. Ahead-Of-Time
Code generation
1. LLVM
2. CUDA
3. OpenCL
4. OpenGL
5. …
Compiler backend
Target platforms
CPU (X86, ARM, RISC-V)
GPU (NVIDIA, AMD)
ASIC (TPU, Inferentia, NNP, … )
DSP
...
More and more Accelerators

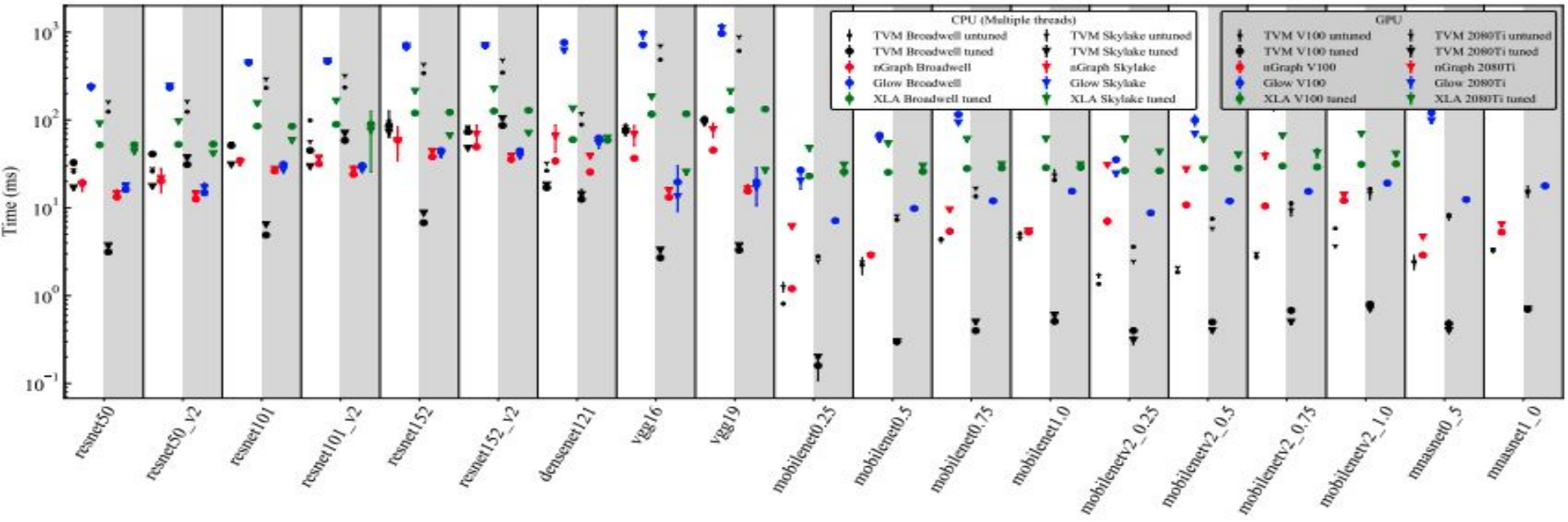

add.120
transpose.121
reshape.122
reshape.123
tuple.124
get-tuple-element.125
ROOT
add.120
reshape.6
ROOT
algebraic simplification
multiply.45
reshape.1
multiply.110
multiply.45
bitcast.1
multiply.110
maximum.109
reshape.1
Parameter 16
multiply.110
reshape.3
convolution.111
broadcast.115
add.113
maximum.116
CSE
broadcast.115
maximum.109
reshape.1
Parameter 15
multiply.110
Parameter 16
convolution.111
reshape.3
add.113
maximum.116
cuDNN transformation
Parameter 15
multiply.110
Parameter 16
custom-call.6 __cudnnSconvForward
reshape.3
get-tuple-element.6
add.113
maximum.116
constant folding
Parameter 18
Fused expression loop fusion
Parameter 1
bitcast.8
Parameter 0
add.0
bitcast.7
operator fusion
custom-call.15 __cudnnSconvForward
get-tuple-element.15
Parameter 16
Parameter 15
multiply.110
reshape.3
broadcast.1
copy.16
custom-call.11 __cudnn SconvBiasActivationForwar
get-tuple-element.11


Hardware Intrinsic Mapping
operator
Hardware intrinsic
for i=1,8
for j=1,8
for k=1,8
z(i,j)=x(k,i)*y(k,j)
gemm8x8(x, y, z)
Memory Allocation & Fetching
Transfer
DRAM
GPU Memory
Data
Data
Fetch
Store
GPU Shared Memory
Data
Warp
Use
Memory Latency Hiding
0 ld ex ld ... ld ex
1 ld ex ld ... ld ex
ld ld ex ex
0 1 0 1
Loop Oriented Optimization Techniques
Loop fusion
for i=1,n
Stmt1(i)
for i=1,n
Stmt2(i)
for i=1,n
Stmt1(i)
Stmt2(i)
Reordering
for i=1,n
for j=1,m
Stmt(i,j)
for i=1,m
for j=1,n
Stmt(i,j)
Unrolling
for i=1,4
Stmt1(i)
Stmt1(1)
Stmt1(2)
Stmt1(3)
Stmt1(4)
Slide windows
Tiling
Fit in
Cache
Parallelization
Autotuner
schedules
Halide
for i=1,n
for j=1,n
Stmt(i,j)
split
for i=1,n
for j=1,n/4
for k=1,4
Stmt(i,j*4+k)
vectorize
for i=1,n
for j=1,n/4
vec (Stmt(i,j))
parallelize
for i=min,max
for j=1,n/4
vec (Stmt(i,j))
Polyhedral
Parallelization / Vectorization
Nested Polyhedral

| | | TVM | nGraph | TC | Glow | XLA |
|---|---|---|---|---|---|---|
| | Developer | Apache | Intel | Facebook | Facebook | Google |
| Frontend | Programming | Python/C++<br>Lambda expression | Python/C++<br>Tensor expression | Python/C++<br>Einstein notation | Python/C++<br>Layer programming | Python/C++<br>Tensorflow interface |
| | ONNX support | ✓<br>tvm.relay.frontend<br>.from_onnx (built-in) | ✓<br>Use ngraph-onnx<br>(Python package) | × | ✓<br>ONNXModelLoader<br>(built-in) | ✓<br>Use tensorflow-onnx<br>(Python package) |
| | Framework support | tvm.relay.frontend<br>.from_* (built-in)<br>tensorflow/tflite/keras<br>pytorch/caffe2<br>mxnet/coreml/darknet | tensorflow<br>paddlepaddle<br>(Use *-bridge,<br>act as the backend) | (Define and optimize<br>a TC kernel, which<br>is finally called by<br>other frameworks.)<br>pytorch/other DLPack<br>supported frameworks | pytorch/caffe2<br>tensorflowlite<br>(Use built-in<br>ONNXIFI interface) | Use tensorflow<br>interface |
| | Training support | ×<br>Under developing<br>(Support derivative<br>operators now) | ✓<br>Only on NNP-T<br>processor | ✓<br>(Support auto<br>differentiation) | ✓<br>(Limited support) | ✓<br>Use tensorflow<br>interface |
| | Quantization support | ✓<br>int8/fp16 | ✓<br>int8 (include training) | × | ✓<br>int8 | ✓<br>int8/int16 (Use<br>tensorflow interface) |
| IR | High-/low-level IR | Relay/Halide | nGraph IR/None | TC IR/Polyhedral | Its own high-/low-level IR | HLO (Both<br>high- and low- level) |
| | Dynamic shape | ✓<br>(*Any*) | ✓<br>(*PartialShape*) | × | × | ✓<br>(*None*) |
| Optimization | Frontend opt | Hardware independent optimizations (refer to Section 4.3)<br>Hardware specific optimizations (refer to Section 4.4)<br>And hybrid optimizations | | | | |
| | Backend opt | | | | | |
| | Autotuning | ✓<br>(To select the best<br>schedule parameters) | ×<br>(Call optimized kernel<br>libraries, no need) | ✓<br>(To reduce JIT<br>overhead) | ×<br>(Additional info is<br>already provided in IR) | ✓<br>(On default<br>convolution and gemm |
| | Kernel libraries | ✓<br>mkl/cudnn/cublas | ✓<br>eigen/mkldnn/cudnn/<br>Others | × | × | ✓<br>eigen/mkl/<br>cudnn/tensorrt |
| Backend | Compilation methods | JIT<br>AOT (experimental) | JIT | JIT | JIT<br>AOT (Use built-in<br>executable bundles) | JIT<br>AOT (Generate<br>executable libraries) |
| | Supported devices | CPU/GPU/ARM<br>FPGA/Customized (<br>Use VTA) | CPU/Intel GPU/NNP<br>GPU/Customized (<br>Use OpenCL support<br>in PlaidML) | Nvidia GPU | CPU/GPU<br>Customized (<br>Official docs) | CPU/GPU/TPU<br>Customized (<br>Official docs) |

Table 2. The hardware configuration.

| | CPU | GPU |
|---|---|---|
| Platform a | Broadwell E5-2680v4 *2 (28 physical cores, 2.4GHz) | Tesla V100 32GB (15.7TFlops, FP32) |
| Platform b | Skylake Silver 4110 *2 (16 physical cores, 2.1GHz) | Turing RTX2080Ti 11GB (13.4TFlops, FP32) |



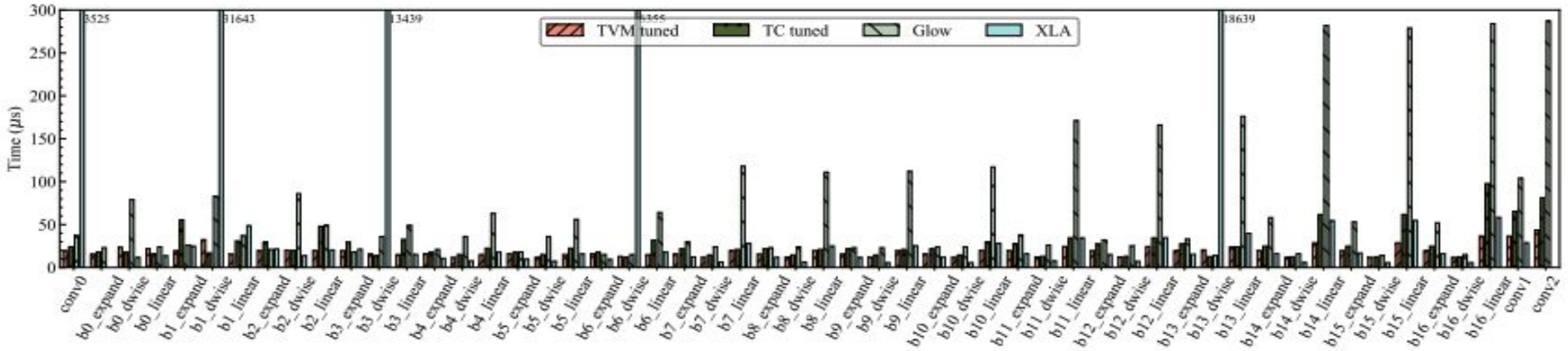
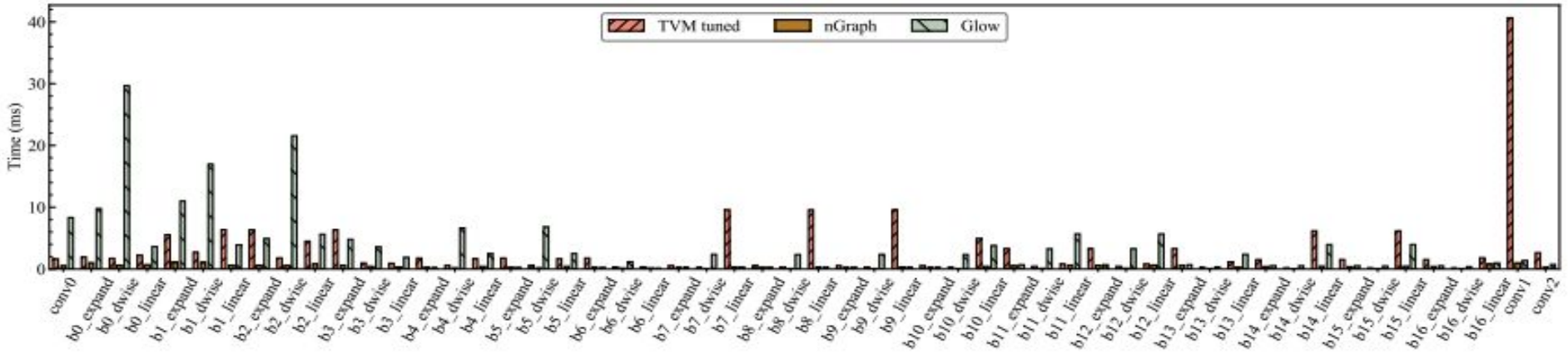



Fig. 6. The performance comparison of convolution layers in `MobileNetV2_1.0` across TVM, TC, Glow and XLA on V100 GPU.



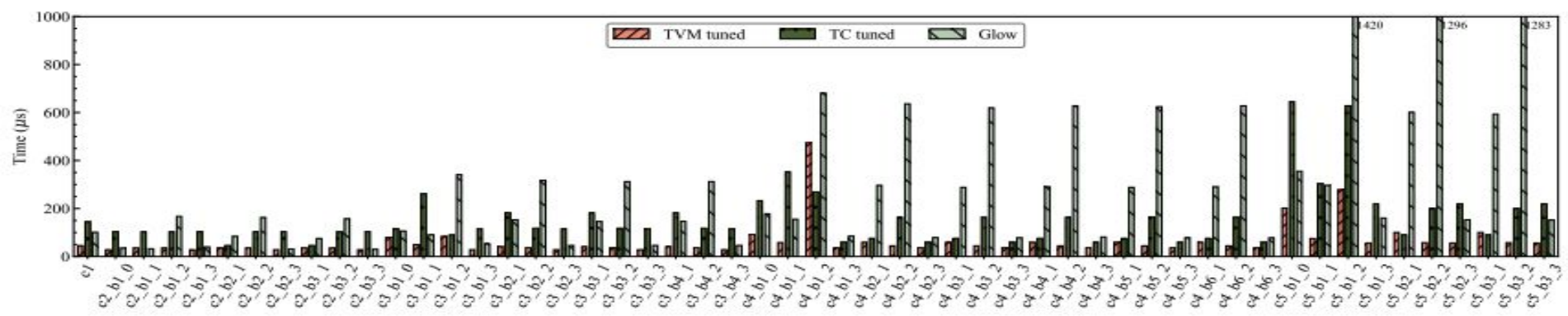



Fig. 8. The performance comparison of convolution layers in ResNet50 across TVM, TC and Glow on V100 GPU.

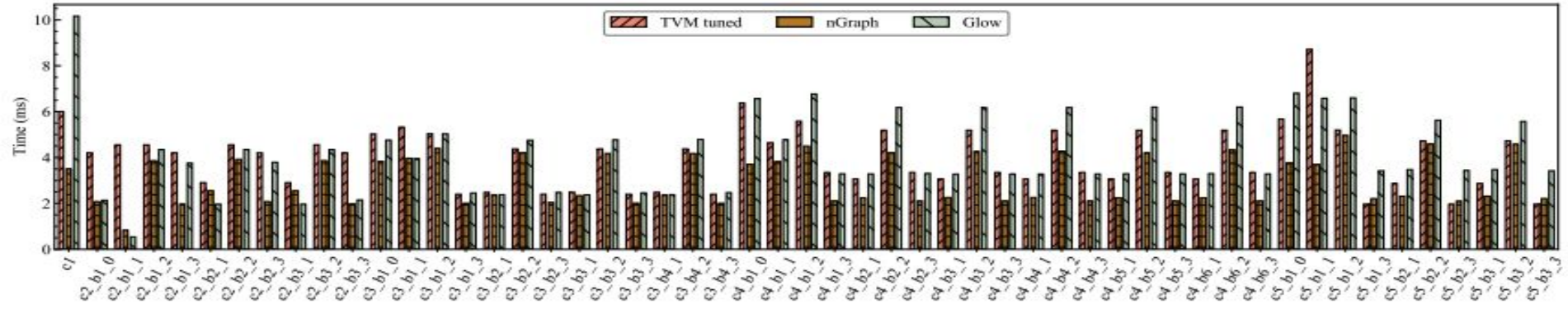



Fig. 9. The performance comparison of convolution layers in ResNet50 across TVM, nGraph and Glow on Broadwell CPU.

Table 3. The number of the clustered and non-clustered convolutions of XLA on V100 GPU and Broadwell CPU.

| | MobileNetV2_1.0 | | ResNet50 | |
|---|---|---|---|---|
| | Clustered | Non-clu- | Clustered | Non-clu- |
| V100 | 5 | 47 | 0 | 53 |
| Broadwell | 17 | 35 | 53 | 0 |