ML Compiler on Heterogenous Computer Architecture

Tsung Tai Yeh
Department of Computer Science
National Yang-Ming Chiao Tung University

Acknowledgements and Disclaimer

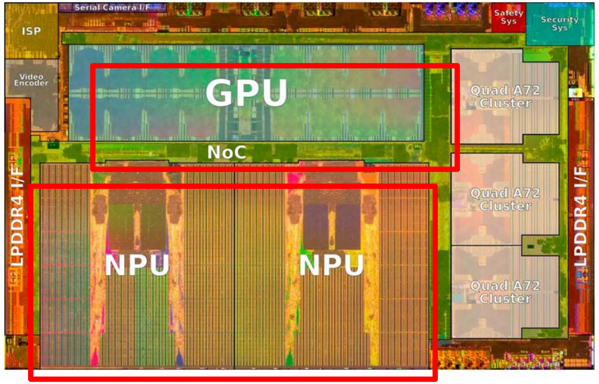
- Slides was developed in the reference with
  CS 15-779, Advanced Topics in Machine Learning Systems (LLM Edition), CMU, 2025
- AI System: https://github.com/Infrasys-AI/AISystem

Outline

- Heterogeneous Computer Architecture
  - CPU+GPU
  - CPU+ASIC
- ML Compiler
  - MLIR
  - IREE
- MegaKernel + Mirage on GPU
  - Domain-Specific Language

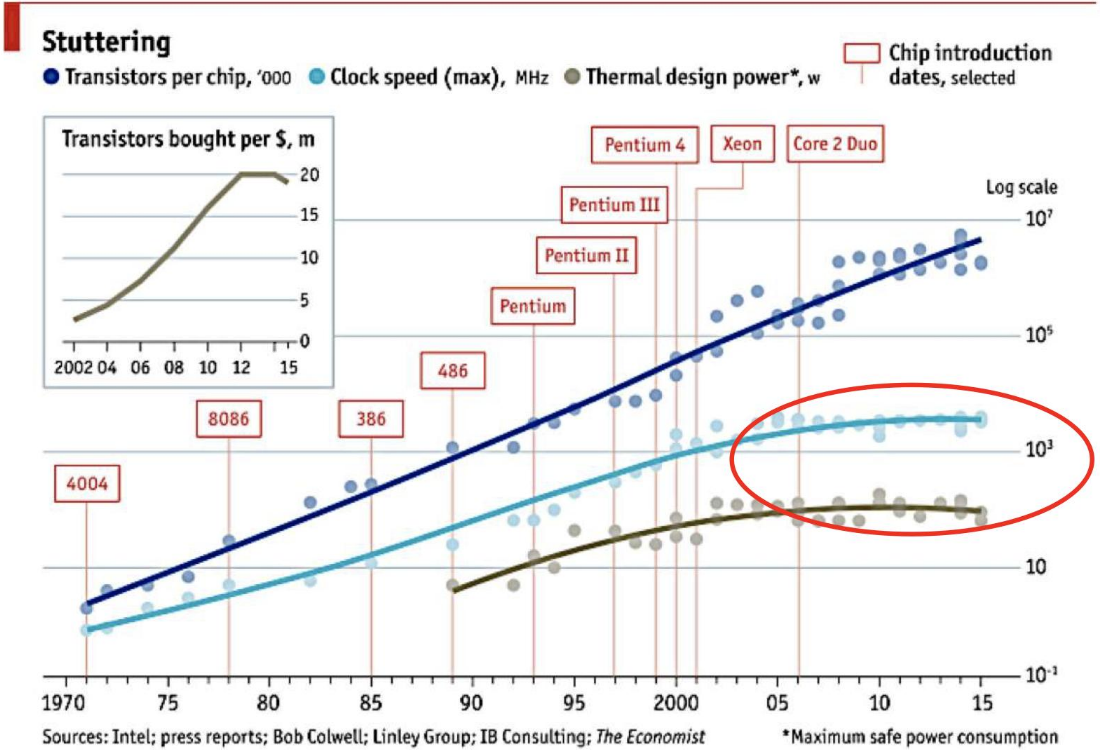
What is heterogeneous SoC?

- **Heterogenous computer architecture**
  - A chip contains CPU and multiple specialized functional units



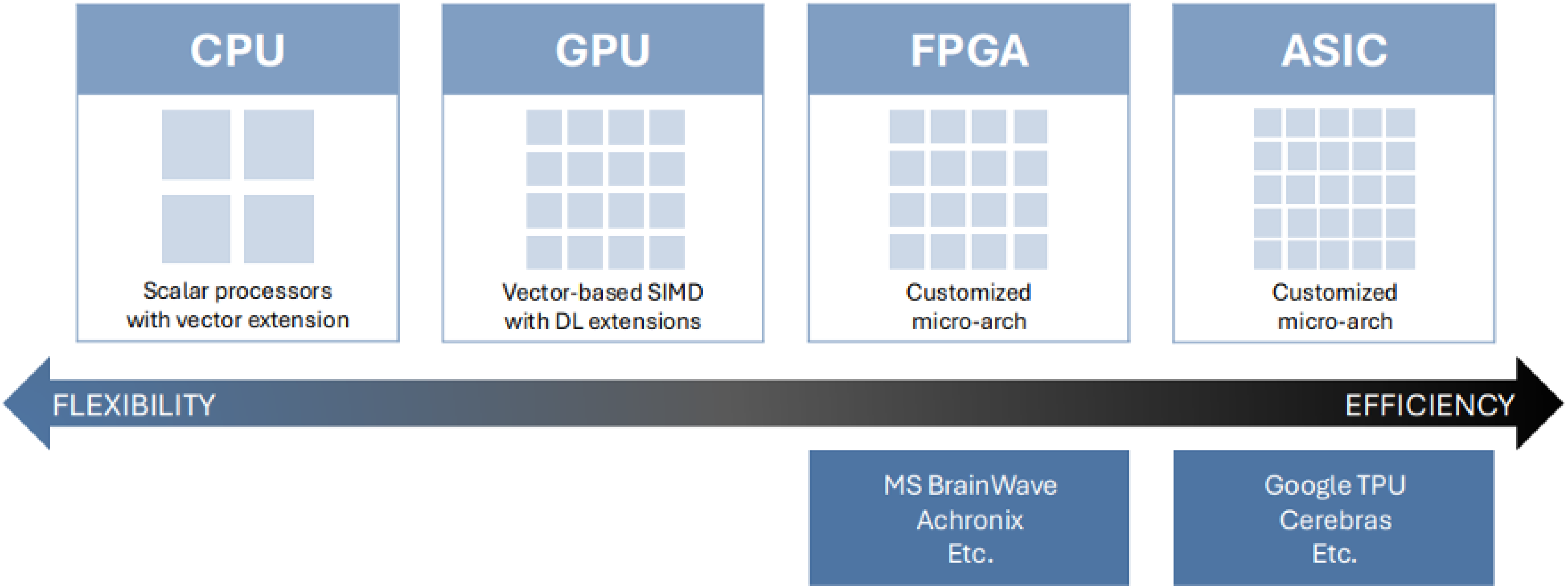
Chip	Tesla - FSD Chip	Qualcomm - Snapdragon 865 (Galaxy S20, March 6 2020)
Technology Node	Samsung 14 nm process	TSMC's advanced 7nm (N7P)
CPU	3x (4-core) Cortex-A72	4x Cortex-A77, 4x Cortex-A55 (4 high power, 4 low power)
GPU	Custom GPU, 0.6 TFLOPS @ 1 Ghz	Adreno 650, 1.25 TFLOPS @ 700 MHz -ish
NPU (AI accelerator)	2x Tesla NPU, each 37 TOPS (total 74 TOPS)	Hexagon 698 @ 15 TOPS
Memory (Cache)	2x 32MB SRAM for NPUs	1 MB L2, 4 MB L3, and 3 MB system wide cache
Memory (RAM)	8GB LPDDR4X, 2x 64-bit, Bandwidth 111 GB/s	16GB LPDDR5, 4x 16-bit , Bandwidth 71.30 GB/s
ISP (Image signal processor)	24-bit? 1 billion pixels per second	Spectra 480, dual 14-bit CV-ISP 2 Gpixel/s, H.265 (HEVC)
Secure Processing Unit	"Security system", verify code has been signed by Tesla.	Qualcomm SPU230, EAL4+ certified

Why heterogeneous computer architecture?

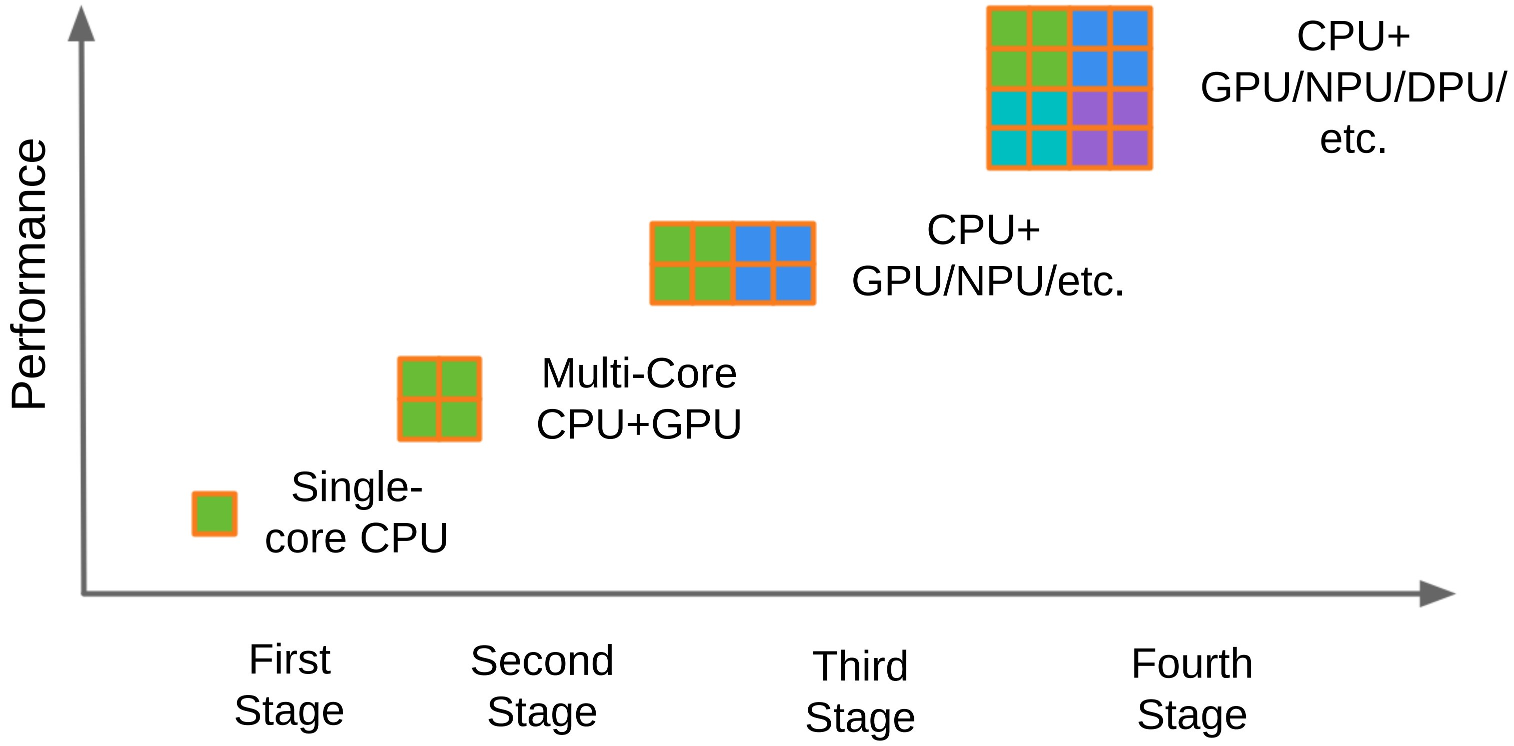


General purpose processor is not getting faster and power-efficient because of **Slowdown of Moore's Law and Dennard Scaling**

Why heterogeneous computer architecture?

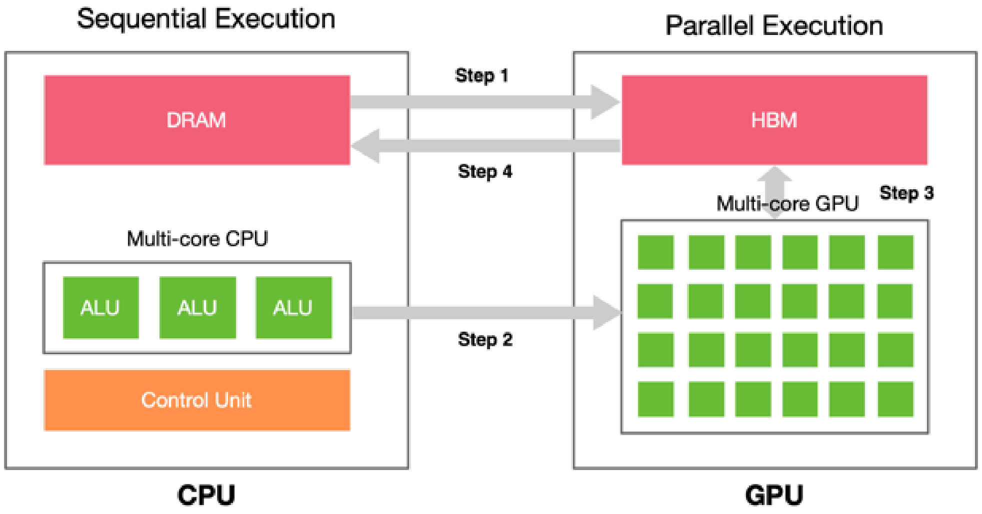


Evolution of Computer Architecture



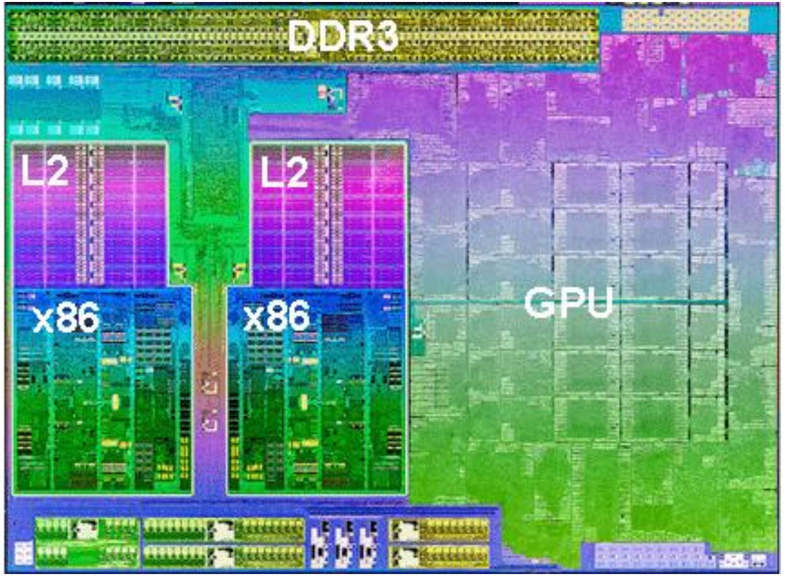
Hetero Computer Architecture (CPU + GPU)

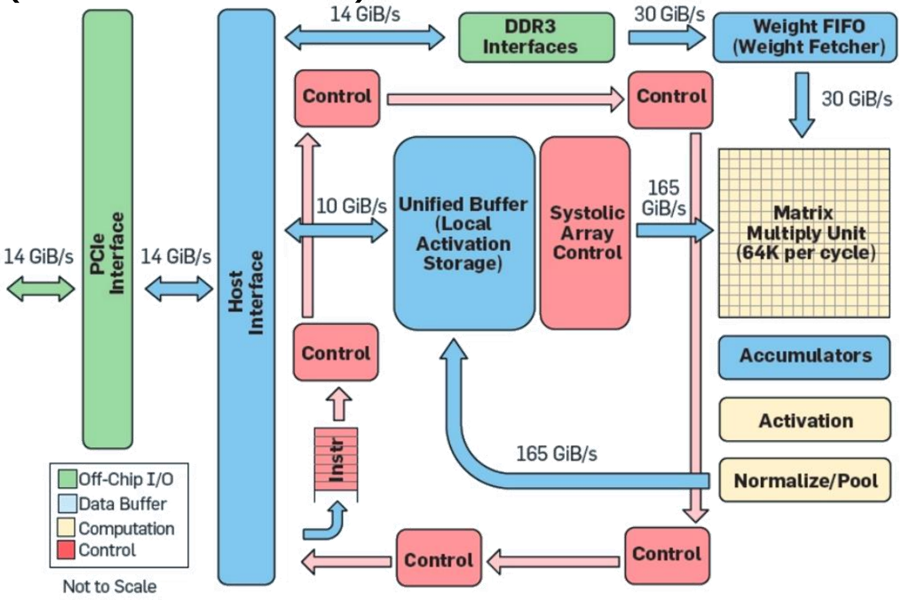
- Step 1: CPU sends data from its host memory to device memory
- Step 2: CPU asks GPU to begin the execution
- Step 3: GPU sends results back to the CPU
- What are advantages when using this hetero. architecture?



Hetero Computer Architecture (CPU + ASIC)

- Two types of heterogeneous computer architecture
  - Discreated CPU+ASIC (separated DRAM)
  - Integrated CPU+ASIC (shared DRAM)



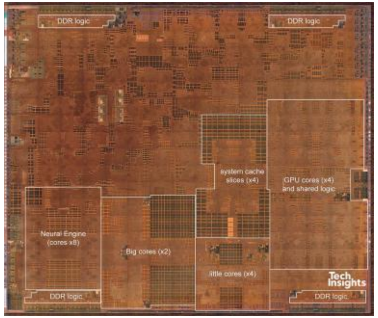


Hetero Computer Architecture (CPU+ASIC)

- **Post-Moore era and dark silicon**
  - A suite of accelerators on chip are rising
  - Applications will only use a subset of processors/accelerators at a time
  - Such a heterogeneous architecture is compatible with dark silicon

2010 Apple A4
65 nm TSMC 53 mm^2
4 accelerators

2014 Apple A8
20 nm TSMC 89 mm^2
28 accelerators

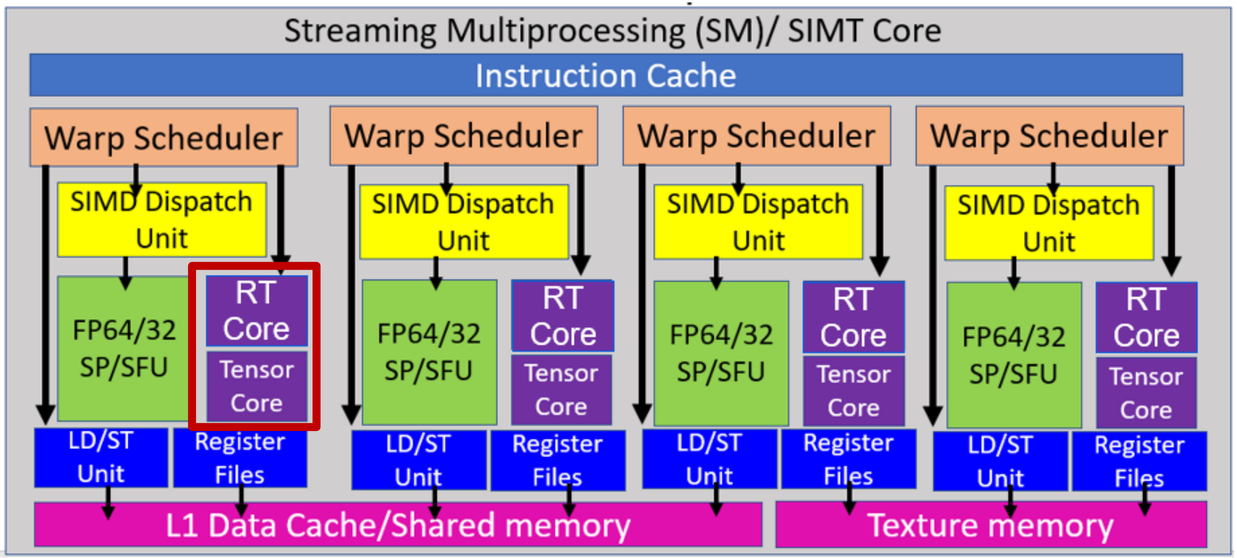


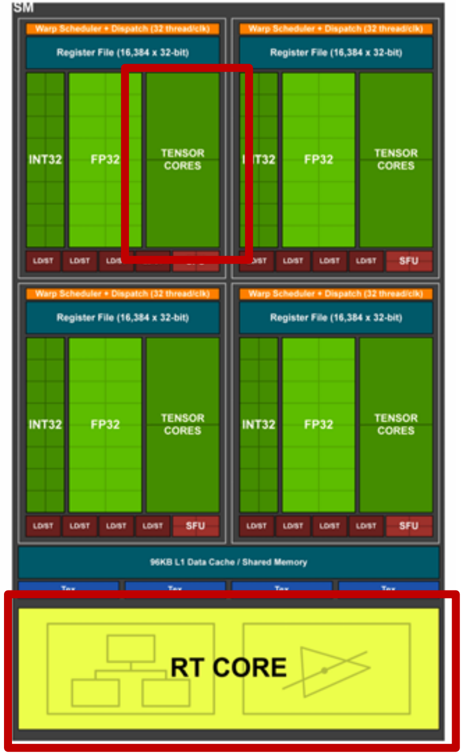
2019 Apple A12
7 nm TSMC 83 mm^2
42 accelerators

https://edge.seas.harvard.edu/files/edge/files/alp.pdf

Hetero Computer Architecture (GPU)

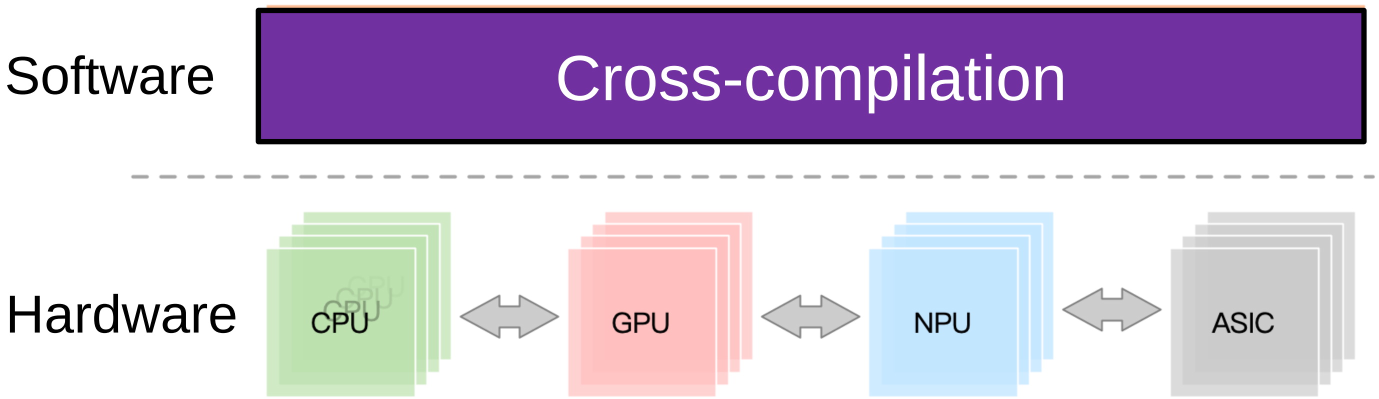
- GPU includes FP, SFU (Special Functional Unit), Ray Tracing (RT) Core, and Tensor Core





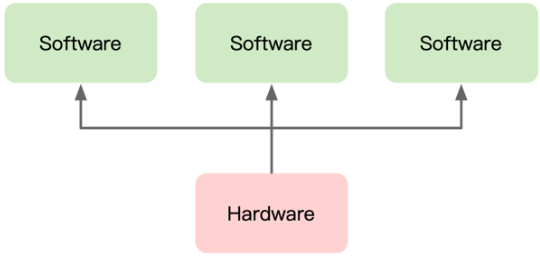
Challenges of Hetero. Computer Architecture?

- Program Compilation
  - Programming model?
  - Data/Kernel mapping/partition?
  - Concurrent execution?

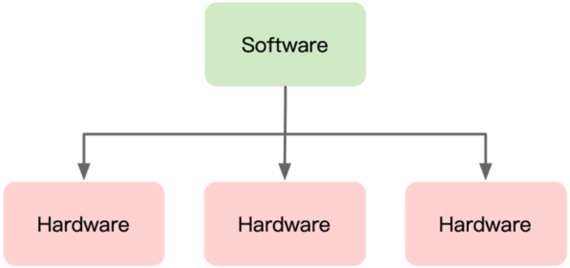


Challenges of Hetero. Computer Architecture?

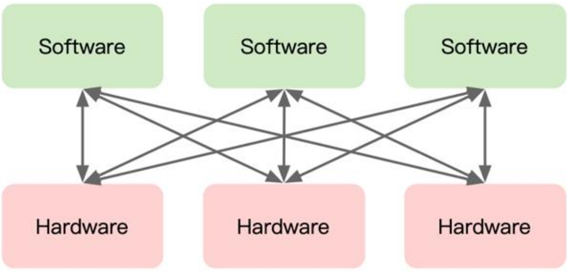
- Program Compilation



Hardware defines software



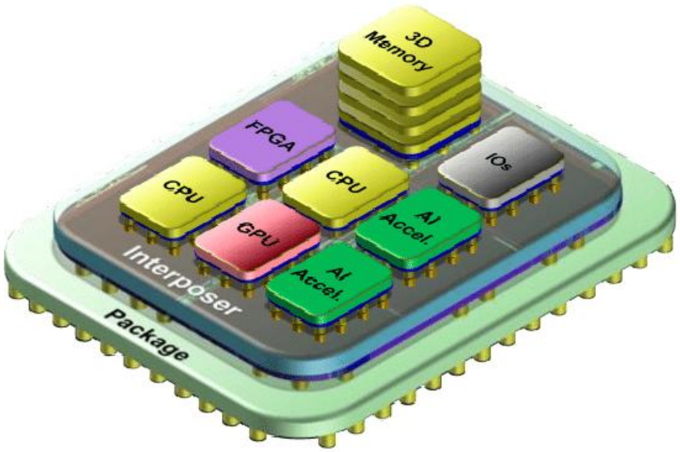
Software defines hardware

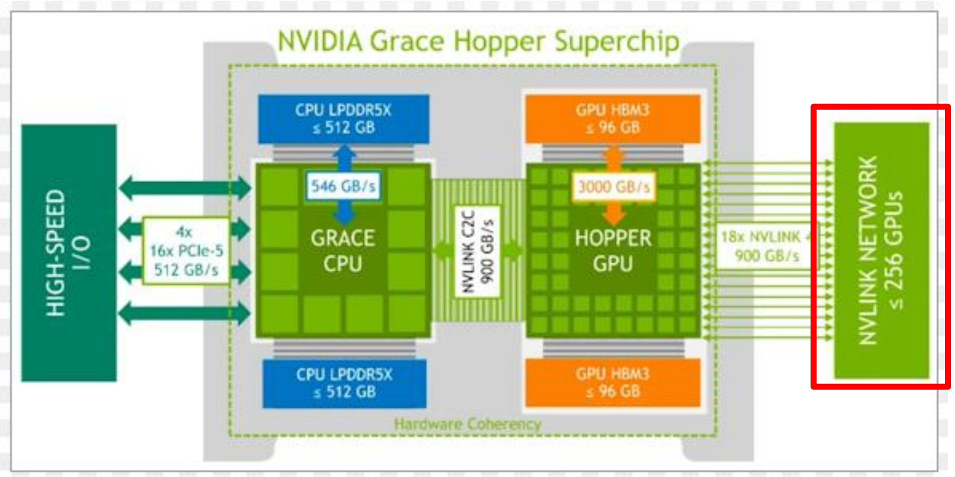


Hardware-Software Co-Design

Challenges of Hetero. Computer Architecture?

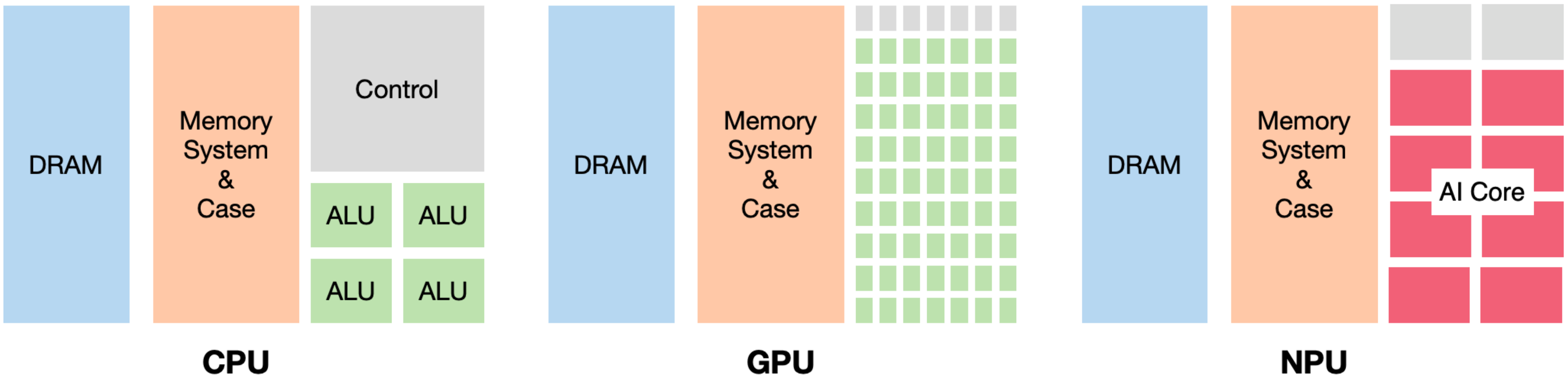
- Hardware
  - Packaging on Chiplet
  - Network-on-Chip (NoC)
    - Photonic Integrated Circuit





Challenges of Hetero. Computer Architecture?

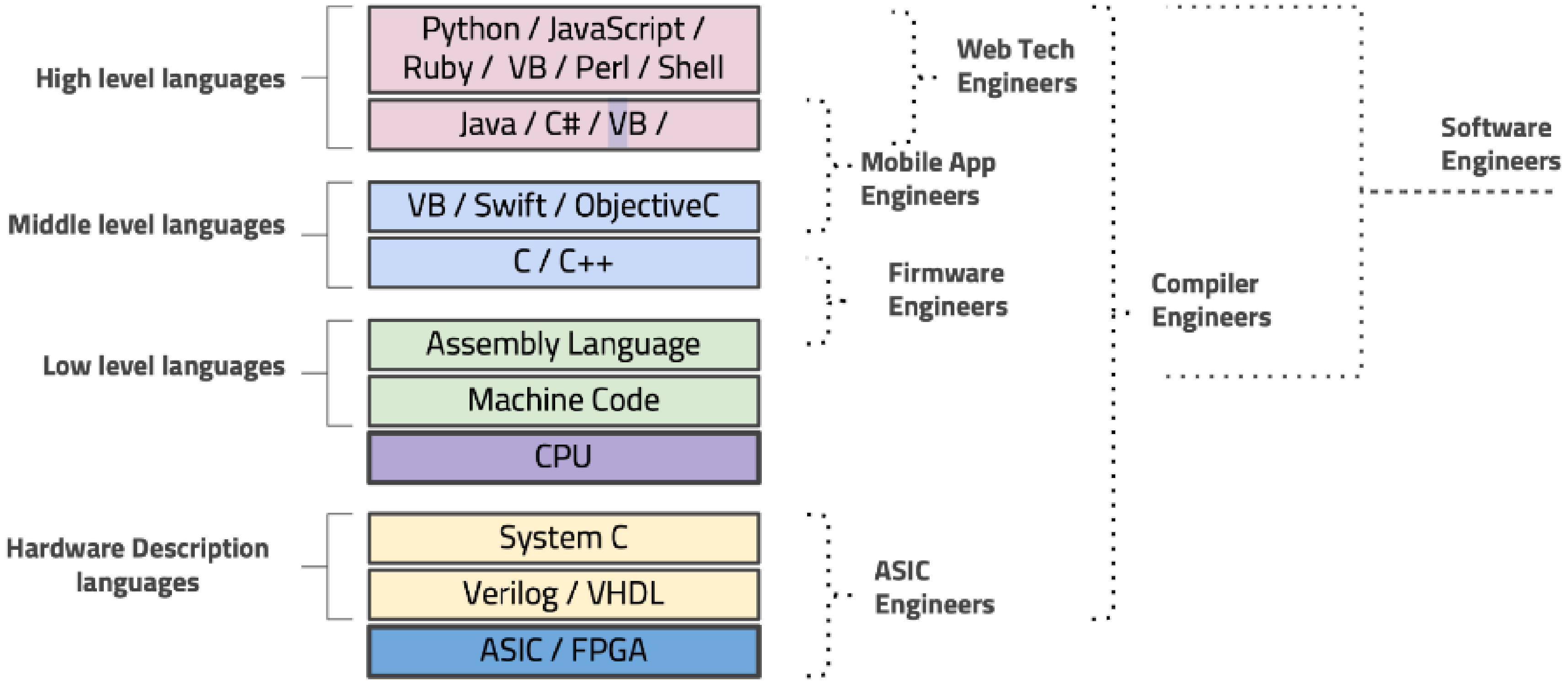
- Trade-off the performance and flexibility



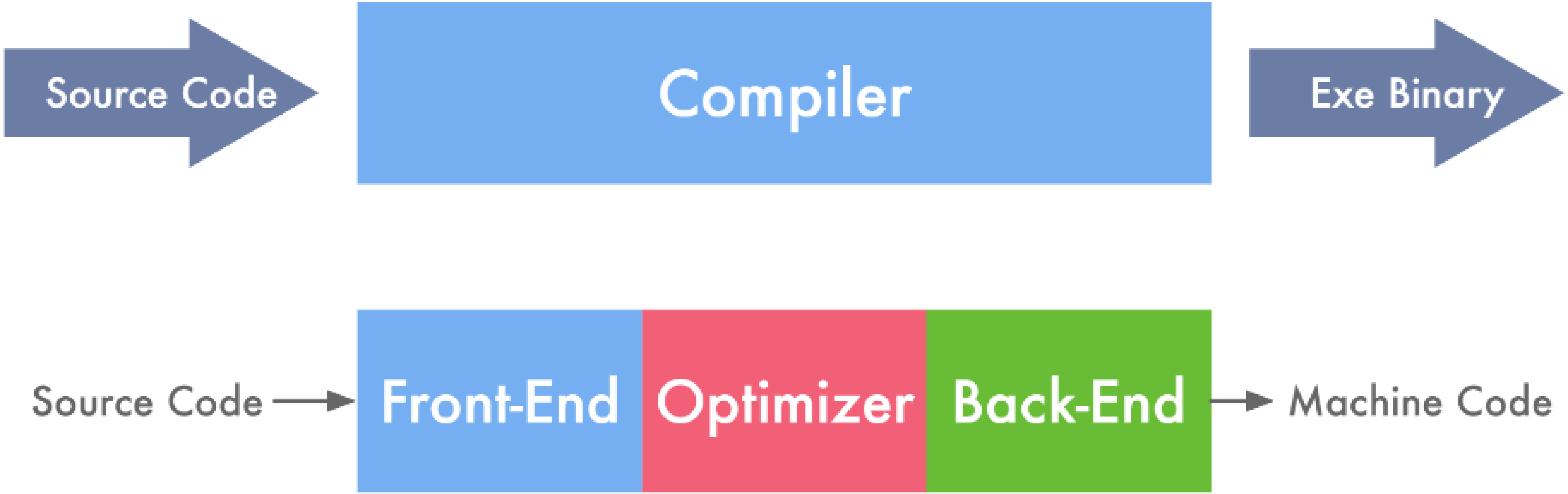
Takeaway Questions

- How to improve the performance of processor?
  - (A) Increase the size of cache
  - (B) Add specialized engines in the processor
  - (C) Utilize high bandwidth memory (HBM)
- What are benefits of heterogeneous computer architecture?
  - (A) Improve energy efficiency of the processor
  - (B) Facilitate parallel computing
  - (C) Reduce memory access latency

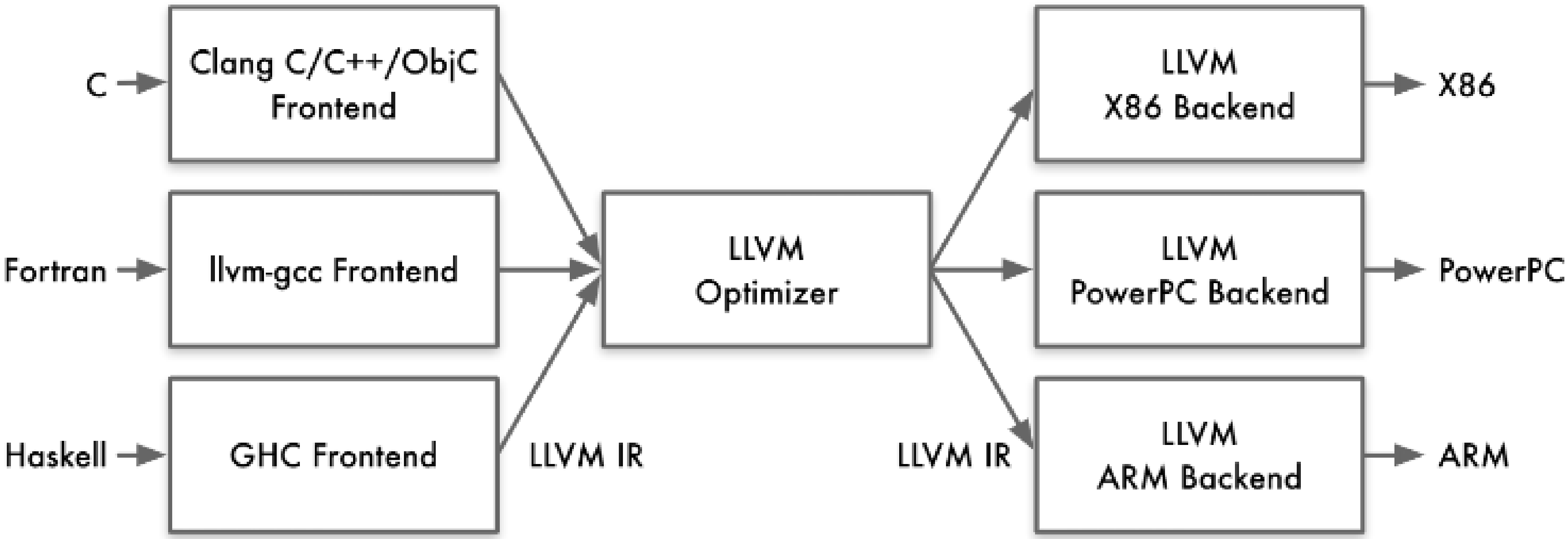
Computer Language Stacks



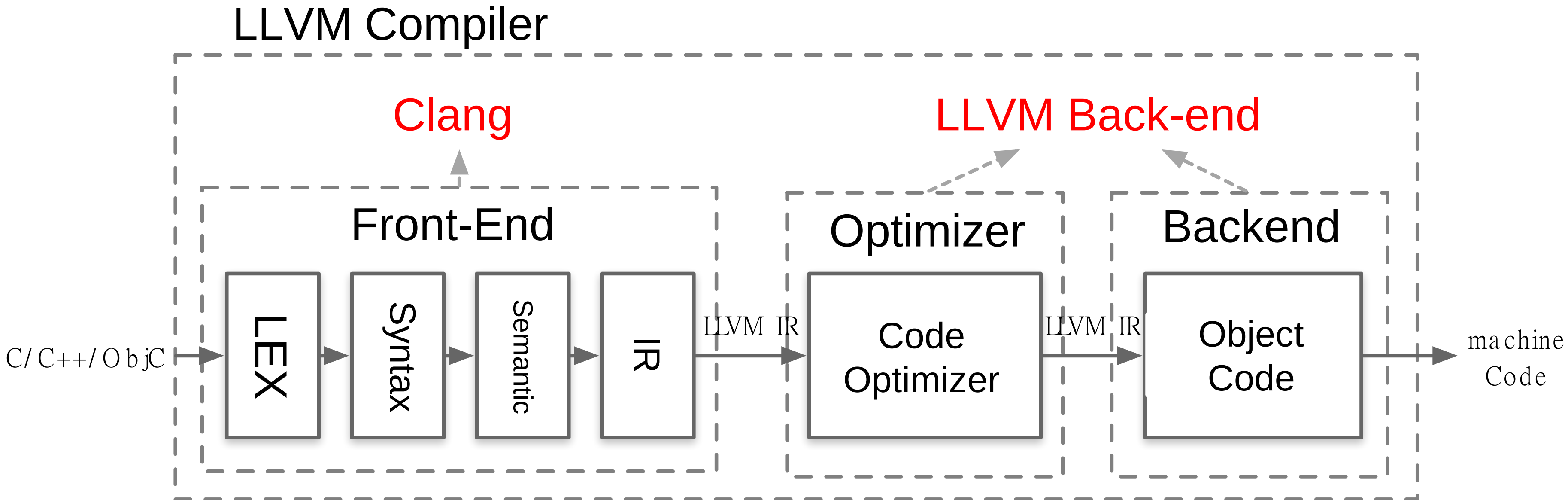
Compiler Basics



LLVM Compiler Architecture

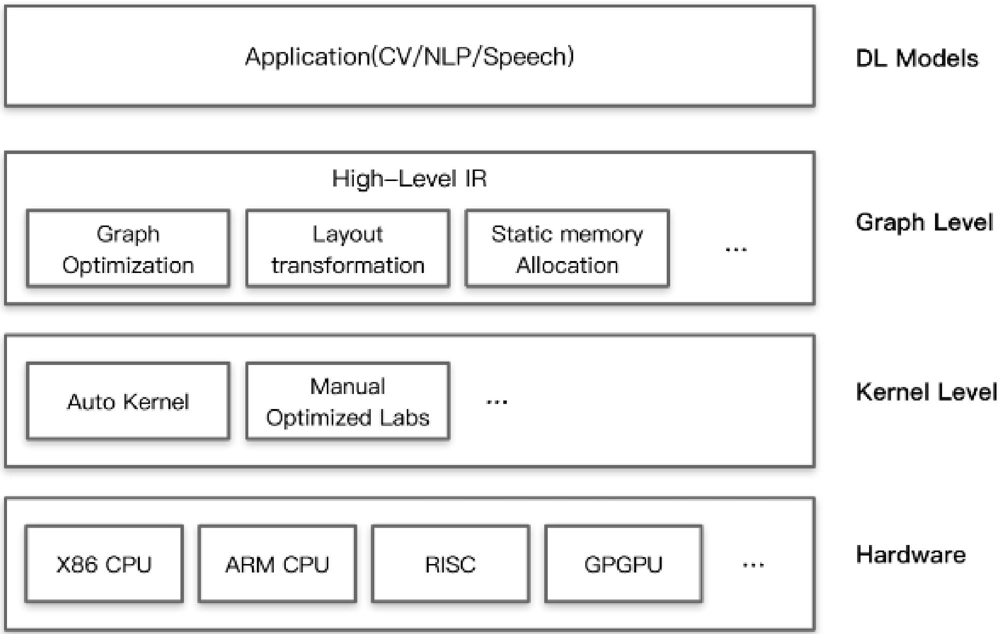


LLVM Compiler Architecture

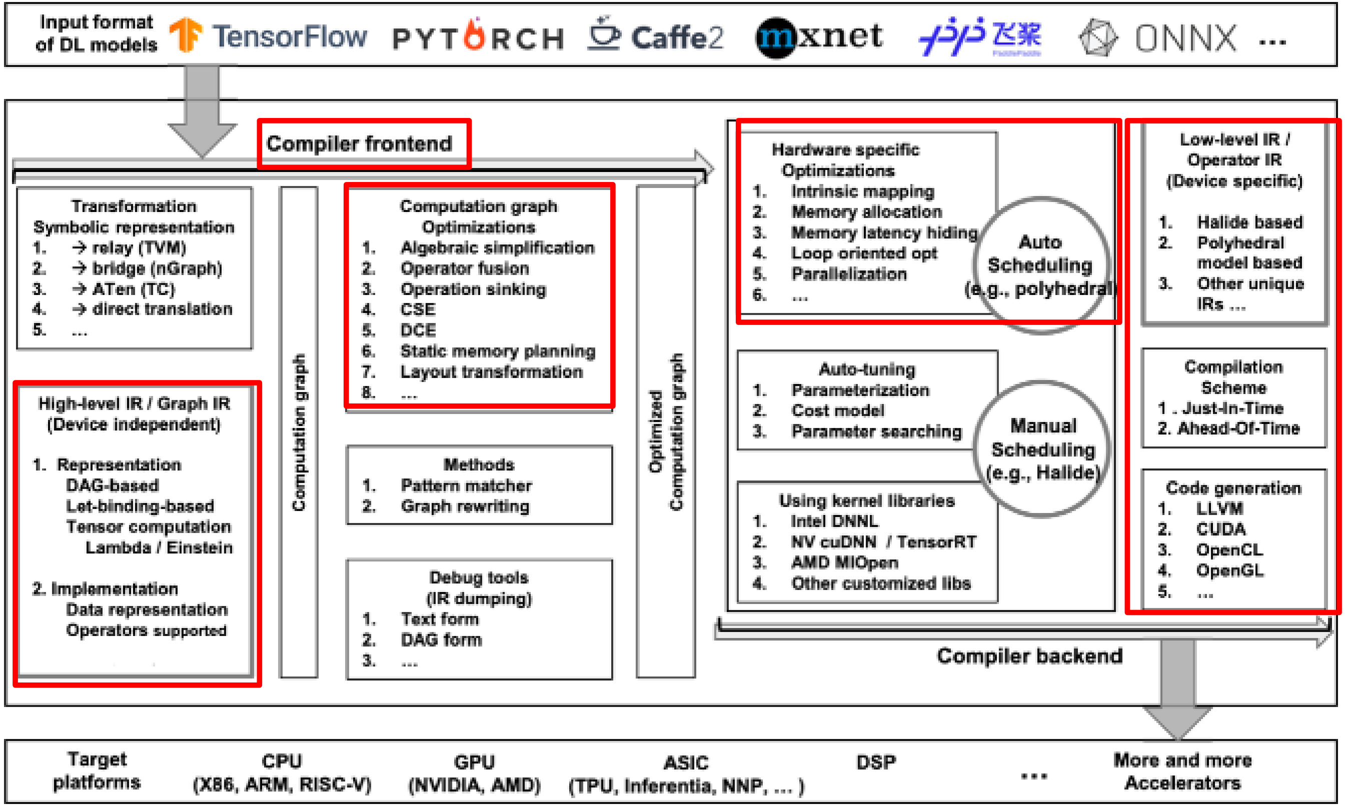


What is AI Compiler?

- Translate the operators of ML models to hardware

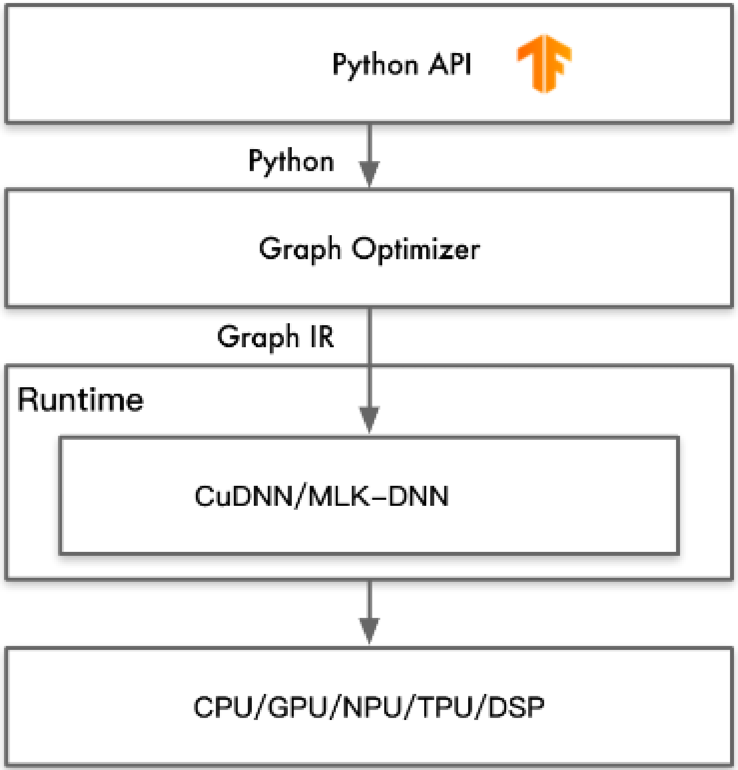


What is AI Compiler?



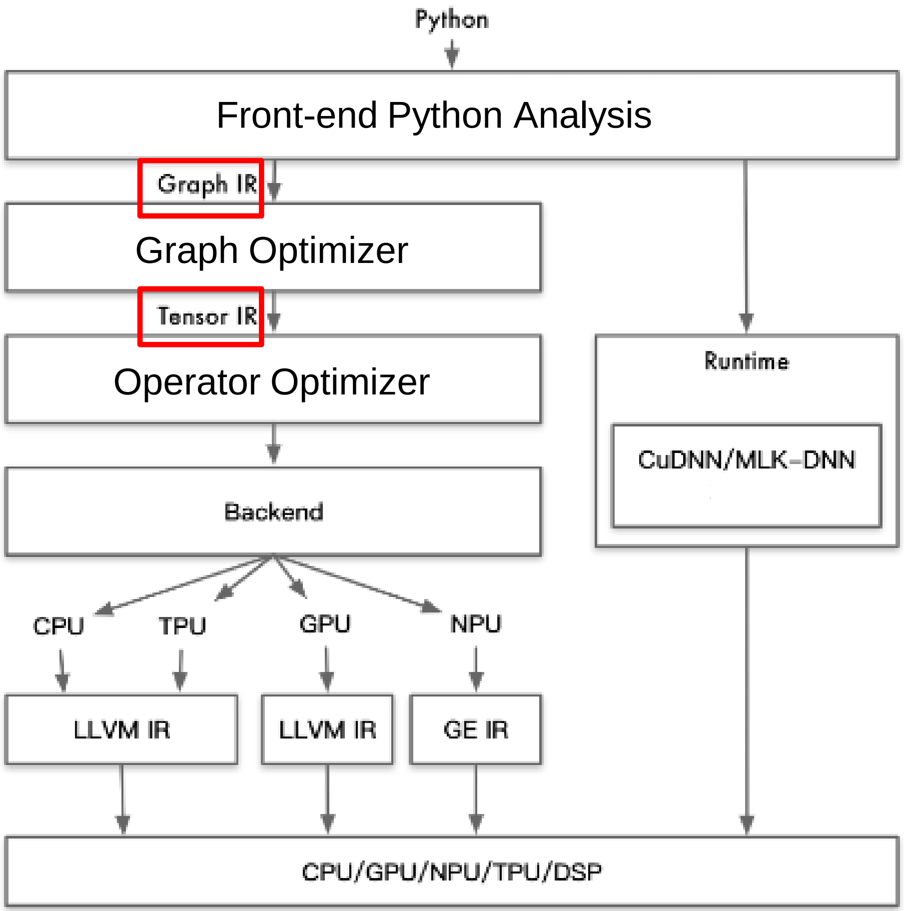
AI Compiler: Stage I

- ML Model graph
  - Static model graph Python->Onnx
  - Graph rewrite/Optimizer
- Performance
  - Op kernel libraries (cuDNN, CMSIS-NN …)
  - More performance improve using Op scheduling, tiling, fusion



AI Compiler: Stage II

- ML Model graph
  - Transforms PyTorch expression into IR
  - Optimizes Tensor IR
- Performance
  - Operator lowering
  - Inter-op optimization
  - Static/dynamic graphs
  - Not only rely on the customized Op Lib



AI Compiler Frontend

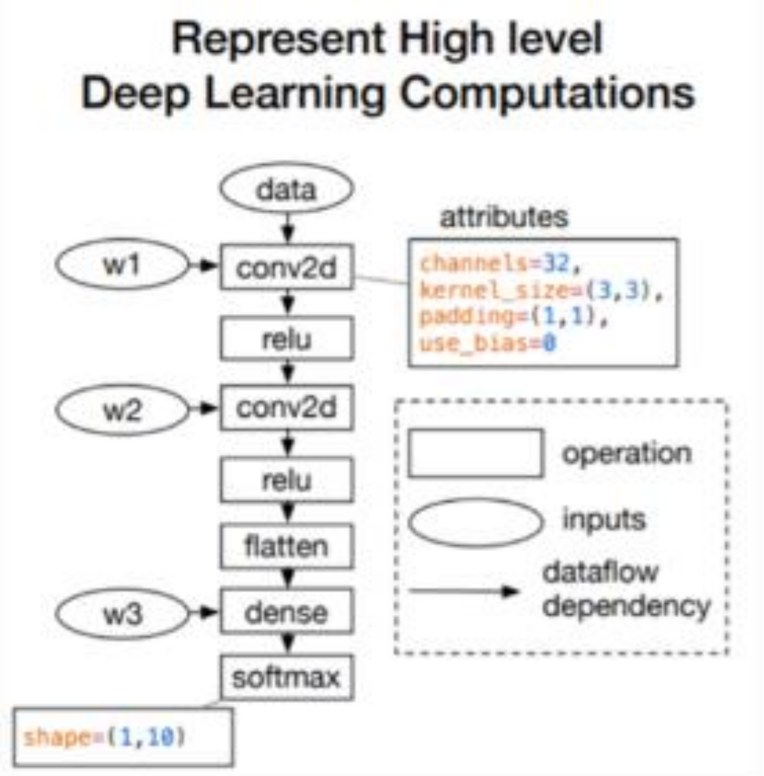
- **Front-end compilation**
  - **Goal**
    - Parse model graphs from different AI system frameworks
    - Transforms model graphs into IR
  - **Tasks**
    - Input format of ML models ((TensorFlow, PyTorch, ONNX …)
    - Transformation: transform model into united expression
      - TVM Relay, PyTorch Aten (TorchScript)
    - High-level IR/Graph IR
      - Hardware independent
      - Operator/Tensor expression

AI Compiler Frontend

- **Front-end compilation**
  - **Tasks**
    - Computational Graph Optimizations
      - Algebraic simplification
      - Operator Fusion
      - Operator Sinking
      - Static memory planning
      - Tensor Layout transformation

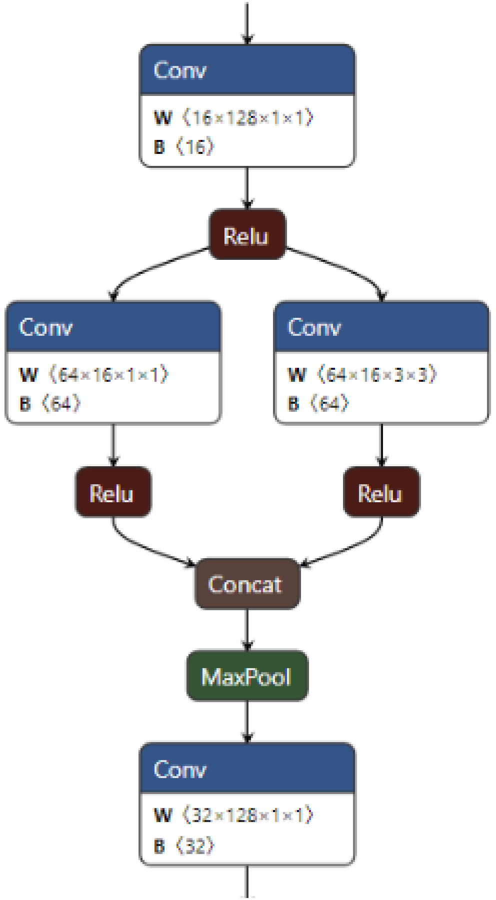
AI Compiler High-Level IR

- Layer-level IR
  - Express ML model structure as a calculation graph
  - High-level abstraction
  - Optimization
    - DSE, operator fusion..
  - Cross-platform

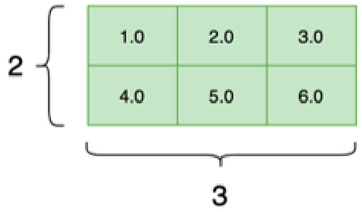


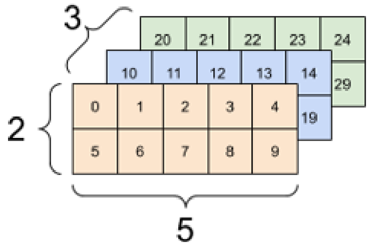
AI Compiler Graph IR

- Graph IR
  - Express ML model as a computation graph
  - Tensor
  - Operator
  - Dependency



AI Compiler Graph IR



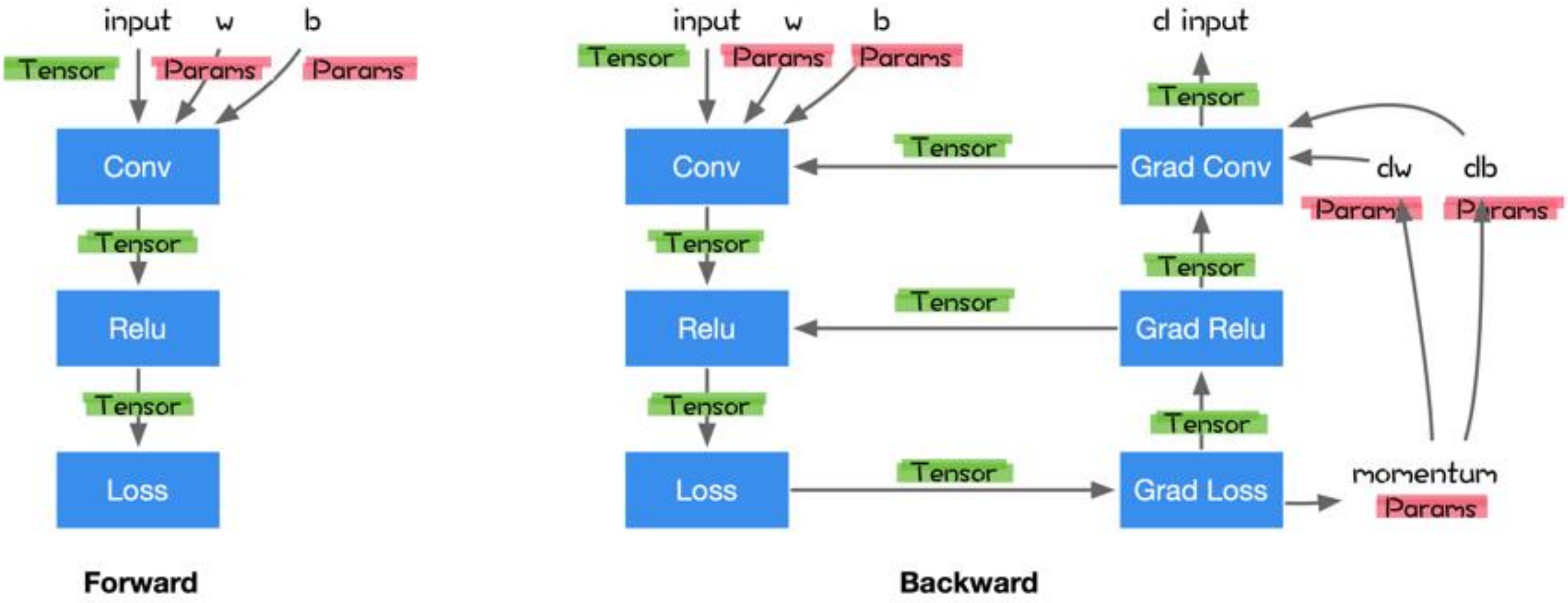


- Tensor
  - Shape [2, 3, 4, 5]
  - [N, C, H, W] [N, H, W, C]
  - Type [int, float, string, …]
- Operator
  - Algebra operator
  - Pre-defined operators

Add	Log	While
Sub	MatMul	Merge
Mul	Conv	BroadCast
Div	BatchNorm	Reduce
Relu	Loss	Map
Floor	Sigmoid	…..

AI Compiler Graph IR

- Directed Acyclic Graph (DAG)
  - Operator, Tensor, control flow (For/While), dependency

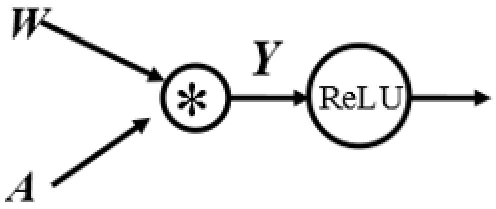


AI Compiler Graph IR

- Static Computational Graph
  - AI system framework (e.g. TensorFlow) parses API used to describe ML model
  - Fixed before execution
  - Use static data structure to describe model graph topology

```python
class Network(nn.Cell):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.dense_relu_sequential = nn.SequentialCell(
            nn.Dense(28*28, 512),
            nn.ReLU())

    def construct(self, x):
        x = self.flatten(x)
        logits = self.dense_relu_sequential(x)
        return logits
```

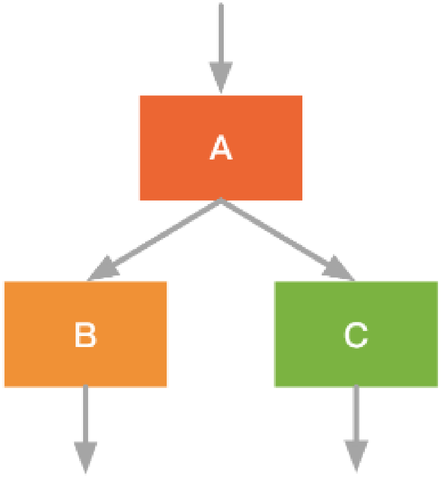


AI Compiler Graph IR

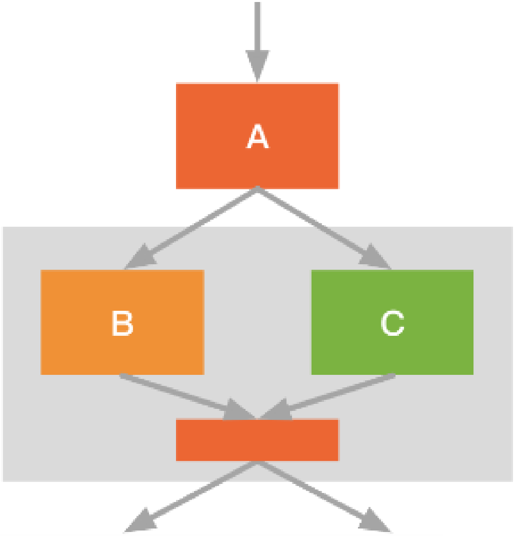
- Dynamic Computational Graph
  - Built on-the-fly as operations are performed
  - Define-by-run offers greater flexibility
    - Good for handling complex and variable-structured data
      - Time-series data: audio
      - Graph data: social networks
      - Multi-modal data: combinations of different variable-structured data types

AI Compiler Graph IR

- Operator fusion



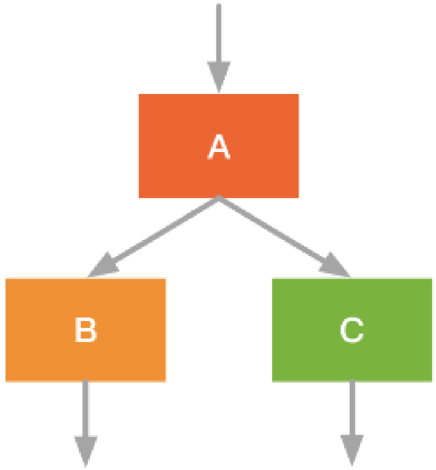
A is called twice



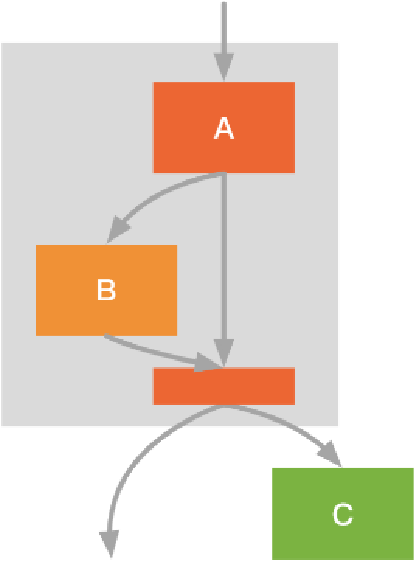
A is called once
Buffer A's output

AI Compiler Graph IR

- Operator fusion



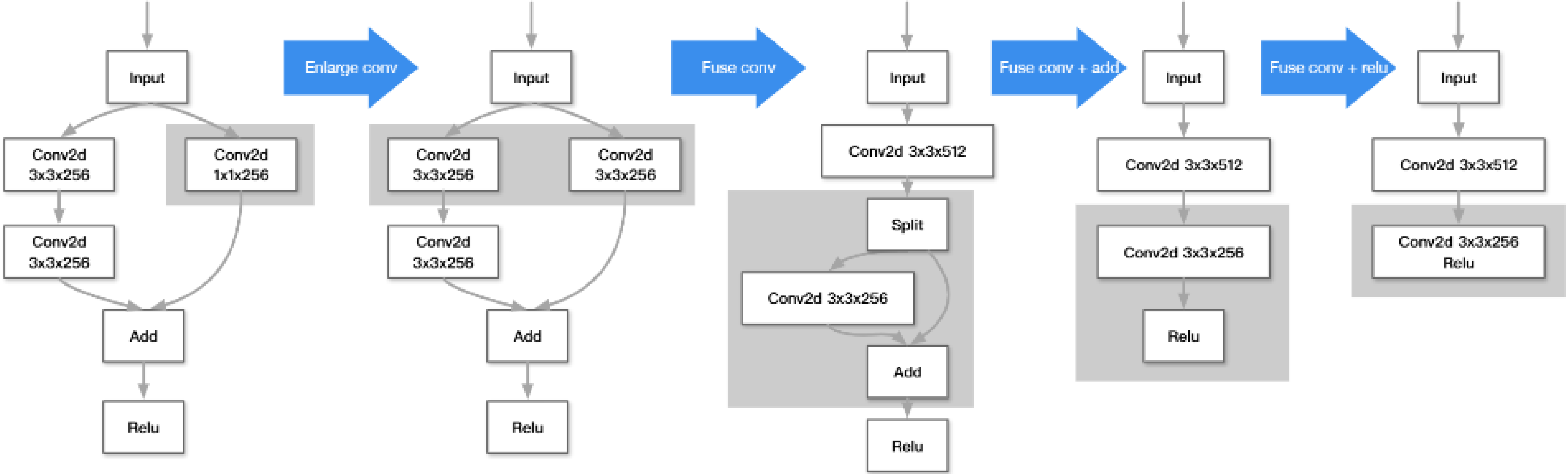
Three kernel calls
(A, B, C)



Reuse the intermediate
data buffer

AI Compiler Graph IR

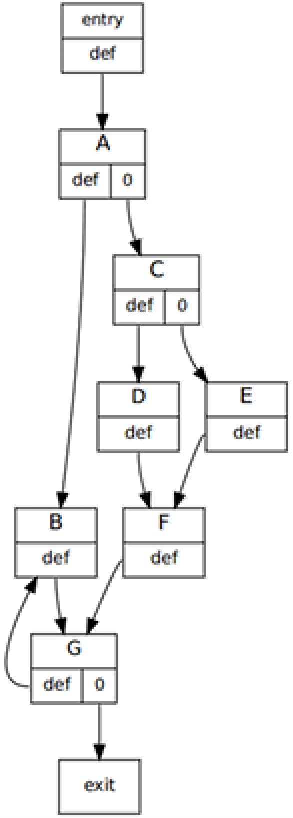
- Operator fusion
  - Reduce memory RW of intermediate tensors



AI Compiler Graph IR

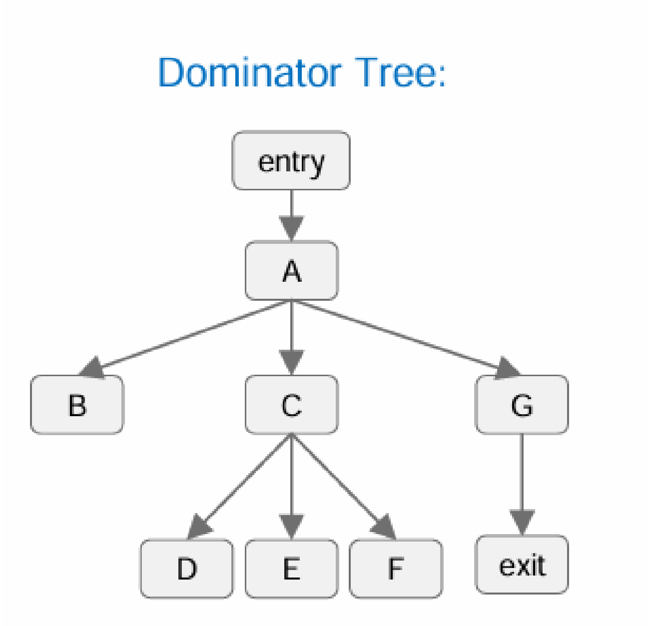
- How to fuse operators ?
  - TVM dominator tree (In a DAG)
  - Dominator
    - <u>Node X dominates node Y iff all paths from the entry to Y go through X.</u>
    - Node A dominates node C (A dom C)

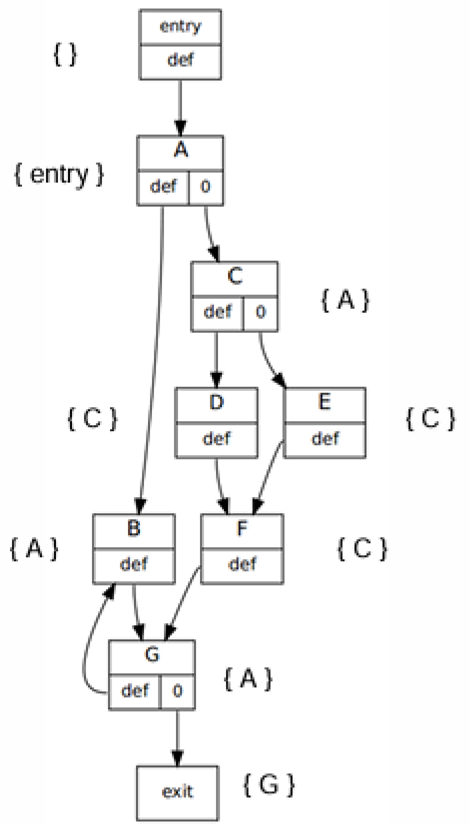
CFG (Control Flow Graph)



AI Compiler Graph IR

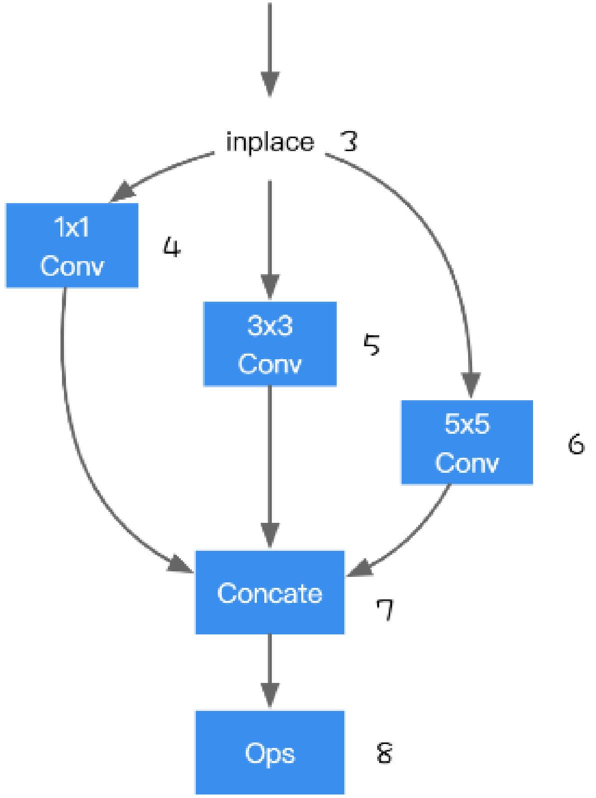
- How to fuse operators ?
  - TVM dominator tree (In a DAG)





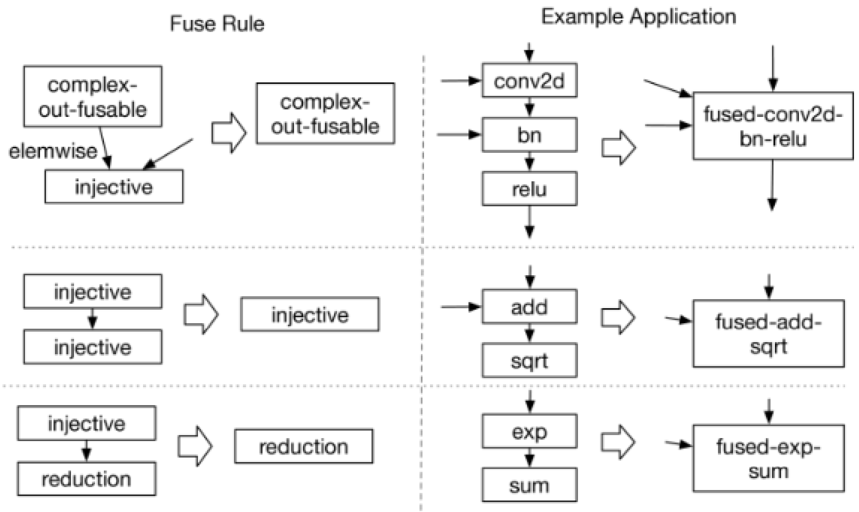
AI Compiler Graph IR

- The purpose of dominator tree
  - Check the path of each node to dominator node
  - Fuses the node that does not affect the rest of nodes
  - How to create a dominate tree?
    - Create DFS tree based on DAG
    - Create DOM (dominator) tree
    - Examine a group of nodes to check if multiple nodes can be fused



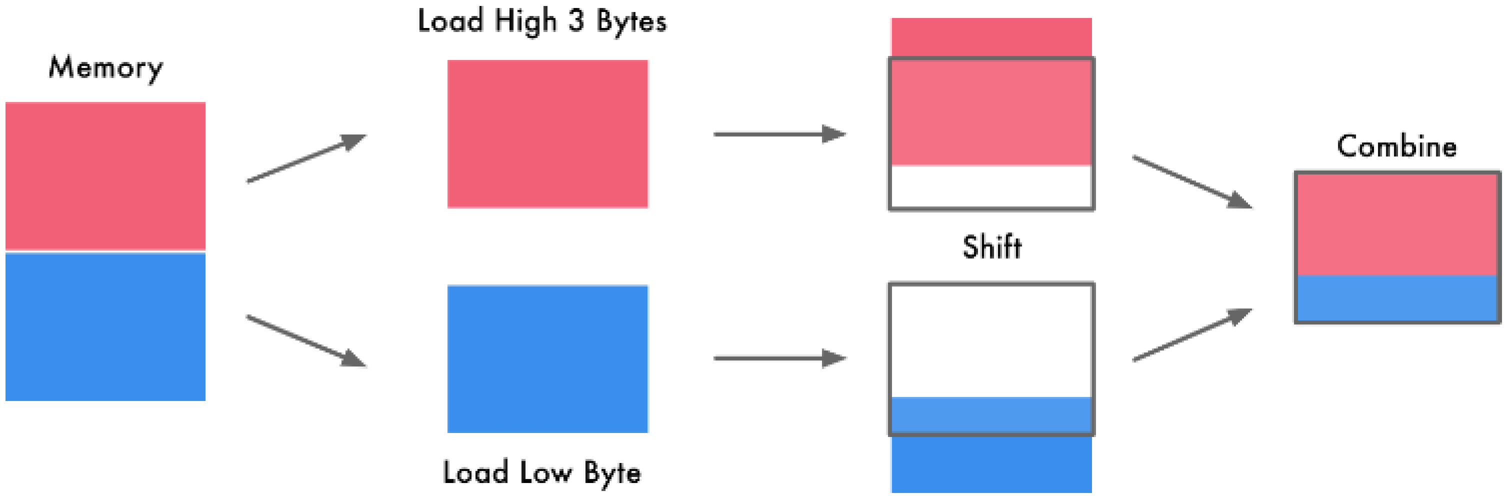
AI Compiler Graph IR

- Rule of operator fusion
  - **Injective (one-to-one map) :** Add, pointwise
  - **Reduction:** sum/max/min
  - **Complex-out-fusable** : conv2D
  - **Opaque** (cannot be fused): sort



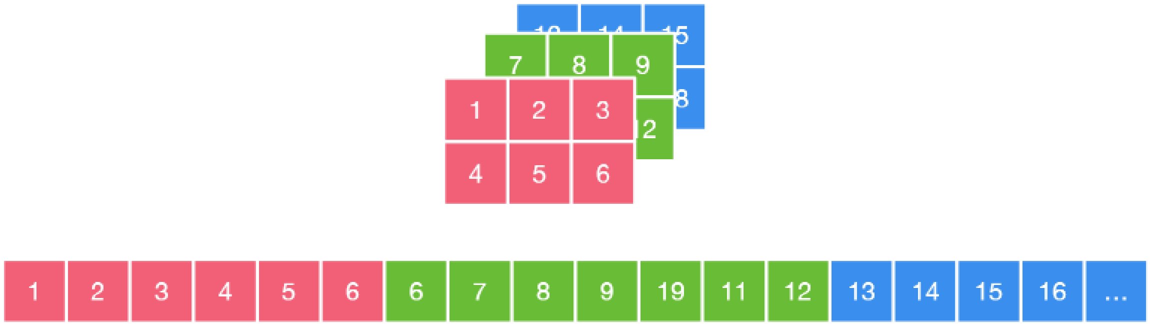
AI Compiler Graph IR

- Data layout alignment
  - Unaligned tensor data will increase the memory transactions



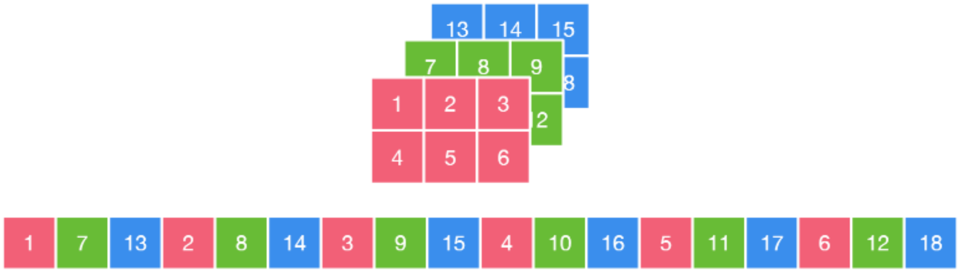
AI Compiler Graph IR

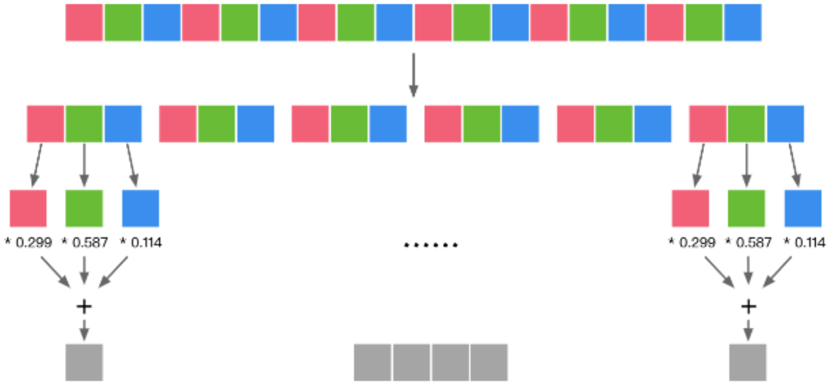
- Data layout (N, C, H, W)
  - N: batch; N: Height; W: width; C: Channels
  - NCHW: arrange data in the same channel in the a consecutive memory space
  - Good for the computations of GPU (data parallel)



AI Compiler Graph IR

- Data layout (N, H, W, C)
  - NHWC: arrange the data having the same location in different channels in a consecutive memory space e.g. Conv1x1





AI Compiler Graph IR

- Data layout (N, C, H, W)
  - PyTorch on NPU/GPU uses **NCHW** data layout
  - TensorFlow use **NHWC** data layout

AI Compiler Graph IR

- Memory optimization
  - Attention memory usage for a deep Transformer (64 layer and 4 heads), recomputed during the backward pass
  - BERT (768 hidden layers) and needs **73GB** memory when the batch size is 64

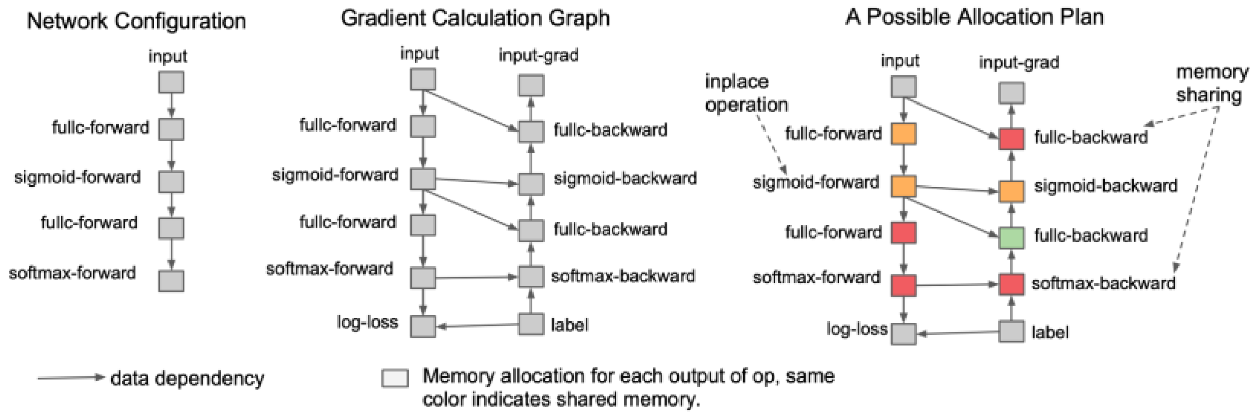
Data type	Stored	Recomputed
1024 text tokens (several paragraphs)	1.0 GB	16 MB
32×32×3 pixels (CIFAR-10 image)	9.6 GB	151 MB
64×64×3 pixels (Imagenet 64 image)	154 GB	2.4 GB
24,000 samples (~2 seconds of 12 kHz audio)	590 GB	9.2GB

AI Compiler Graph IR

- Memory optimization
  - **Static memory allocation**
    - Parameters, constant, output
    - Allocate memory in the model initialization stage
  - **Dynamic memory allocation**
    - Output tensor, workspace tensor (intermediate tensor)
    - Allocate memory (dynamic: varying batch size, static: fixed batch size)

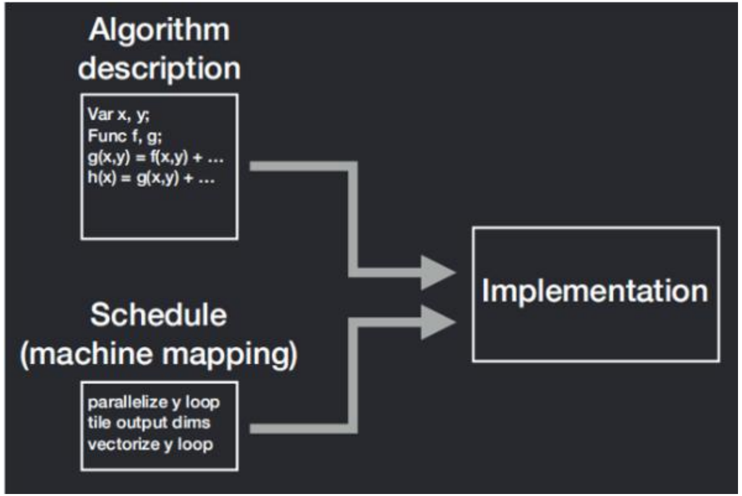
AI Compiler Graph IR

- Memory optimization
  - **Inplace operation:** overwrite when the next operator is element-wise operator
  - **Memory sharing:** the size of both operators is the same and no data dependency in these two operators



AI Compiler Low-Level IR

- Low-level IR
  - Describes the computation of a ML model in a more fine-grained representation than that in high-level IR
  - Enable the target-dependent optimization
  - Halide-based IR
    - Separation of comp. and schedule
    - Choose the best schedule to specific target platform

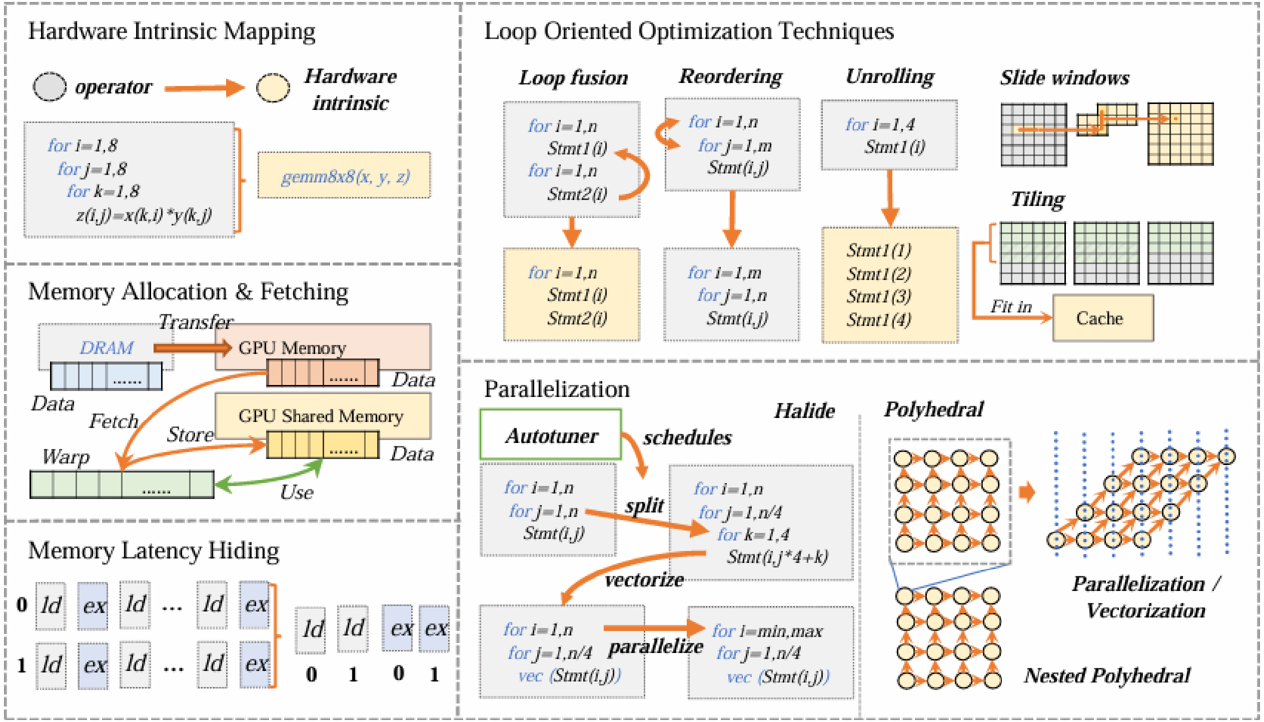


AI Compiler Backend

- Back-end compilation
  - Goal
    - Transform ML graph to specific hardware
    - Code generation: LLVM/CUDA/OpenCL …
  - Tasks
    - Hardware Specific Optimization
      - Memory allocation
      - Parallelization
    - Scheduling
      - Auto Scheduling: polyhedral, Halide

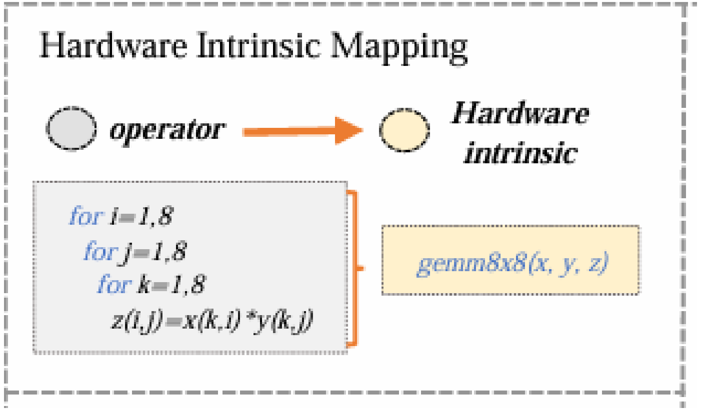
AI Compiler Backend

- Hardware-specific optimizations



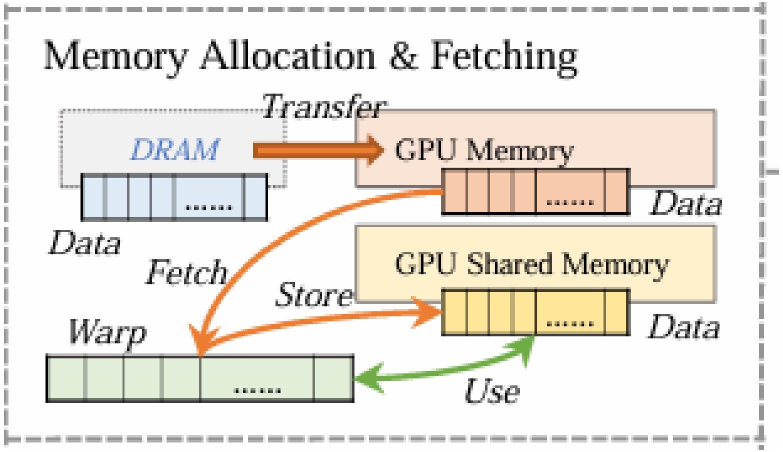
AI Compiler Backend

- Hardware intrinsic mapping
  - Transform a certain set of low-level IR to kernels
  - TVM extensible tensorization
    - Declare the behavior of hardware intrinsic and lowering the rule for intrinsic mapping
    - Enable compiler backend apply optimized micro-kernels to a specific pattern of operations



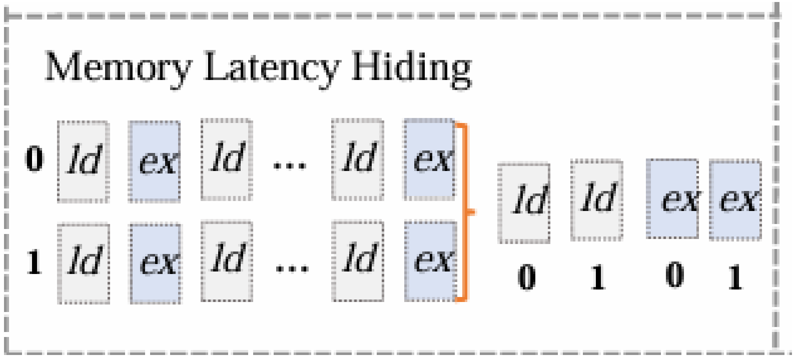
AI Compiler Backend

- Memory allocation and fetching
  - E.g. GPU memory hierarchy requires efficient memory allocation and fetching techniques for improving data locality
  - TVM memory scope
    - Tag a compute stage as shared or thread-local
      - Shared: generates code with shared memory allocation
      - Properly insert memory barrier



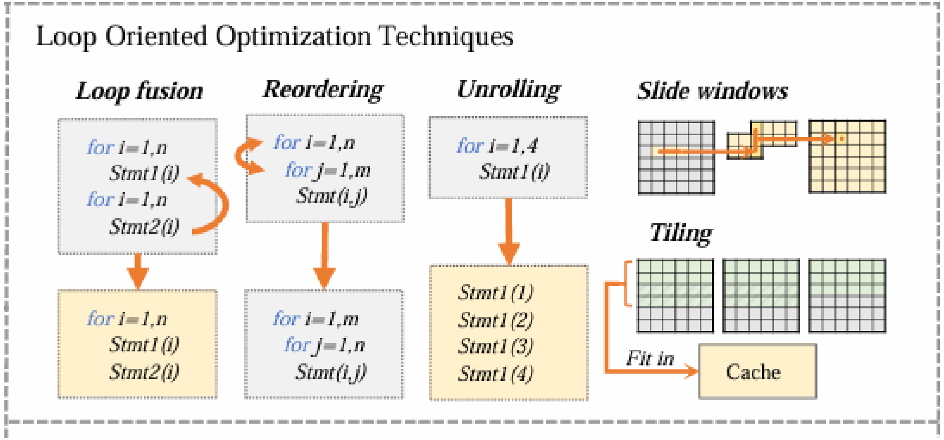
AI Compiler Backend

- Memory latency hiding
  - Reordering the execution pipeline
  - In TPU-Accel with decoupled access-execute (DAE)
    - Backend needs to perform scheduling and fine-grained sync to produce the correct and efficient code
  - TVM virtual threading schedule primitive
    - Virtually parallelized threads
    - Barriers + operations = a single instruction stream



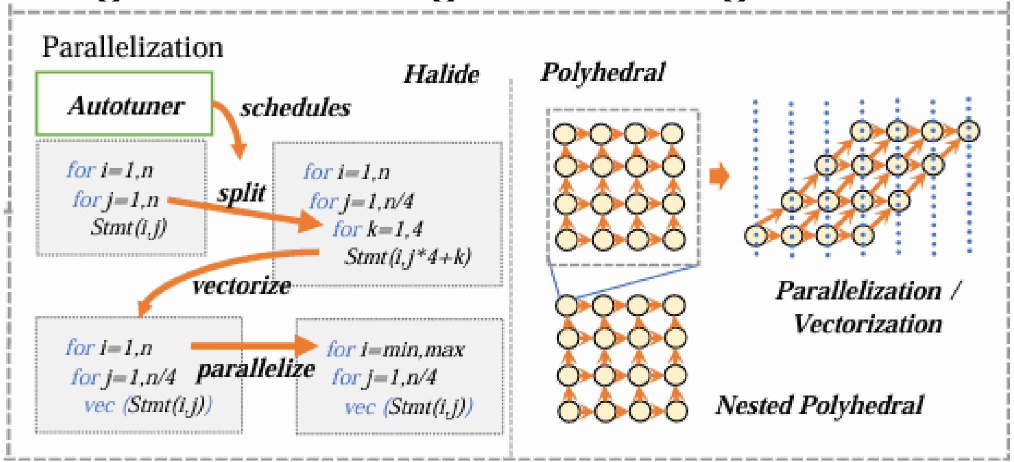
AI Compiler Backend

- Loop oriented optimization
  - Loop fusion
    - fuse loops with the same boundaries for better data reuse
  - Sliding window
    - Compute values when needed
    - Store them for data reuse until they no longer required



AI Compiler Backend

- Parallelization
  - Halide uses a <u>schedule primitive called parallel</u>
    - Specify the parallelized dimension of the loops
  - <u>Nested polyhedral model</u> – detect hierarchy parallelization among levels of tiling and striding



AI Compiler Backend

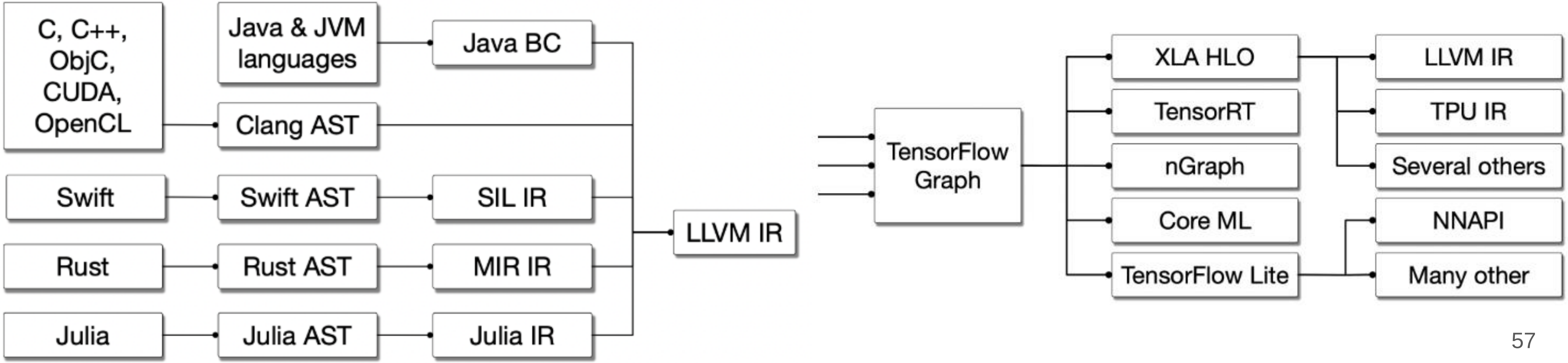
- Back-end compilation
  - Tasks
    - Auto-tuning
      - Parameterization cost model
    - Using kernel libraries
      - NVIDIA cuDNN/TensorRT, AMD MIOpen
    - Low-level IR/ Operator IR
      - Halide IR
    - Compilation scheme
      - Just-In-Time (JIT), Ahead-Of-Time (AOT)

Takeaway Questions

- What are jobs of AI compiler?
  - (A) Handle tensor memory allocation
  - (B) Reorder the execution of the DL operators
  - (C) Generate assembly codes
- How does AI compiler improve the data reuse on the local memory?
  - (A) Use the NCHW data layout
  - (B) Operator fusion
  - (C) Operator lowering

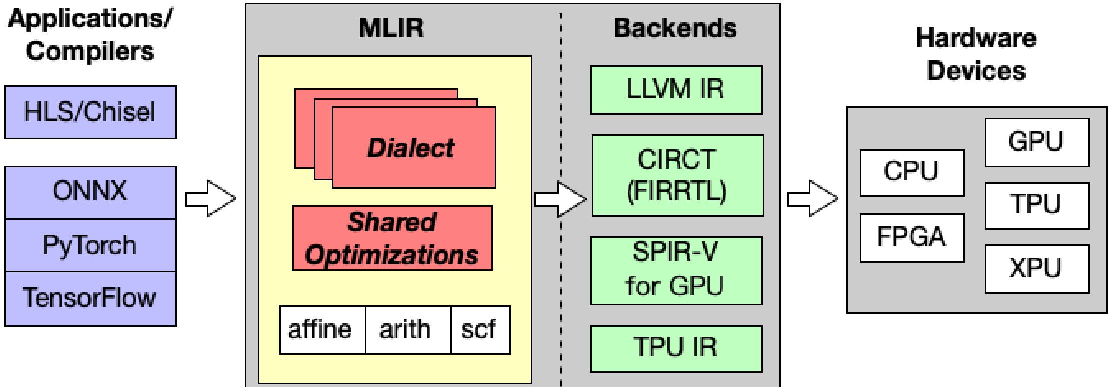
MLIR – Multi-Level Intermediate Representation

- Most high-level languages have their own AST
- ML graphs compilation process is fragmented
- MLIR allows developers to use a unified codebase/framework to do their optimizations and develop some optimizations for multiple inputs



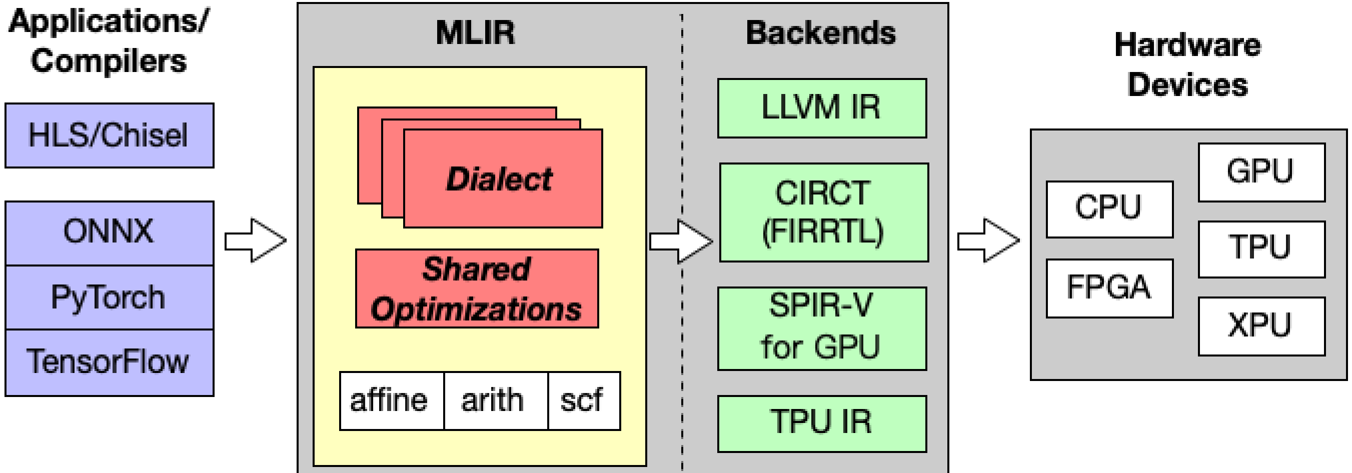
MLIR – Multi-Level Intermediate Representation

- MLIR's input
  - applications, compilers, C program, etc.
- Within MLIR
  - Implement multiple Dialects for distinct inputs
  - Use Dialect to deal with tensors



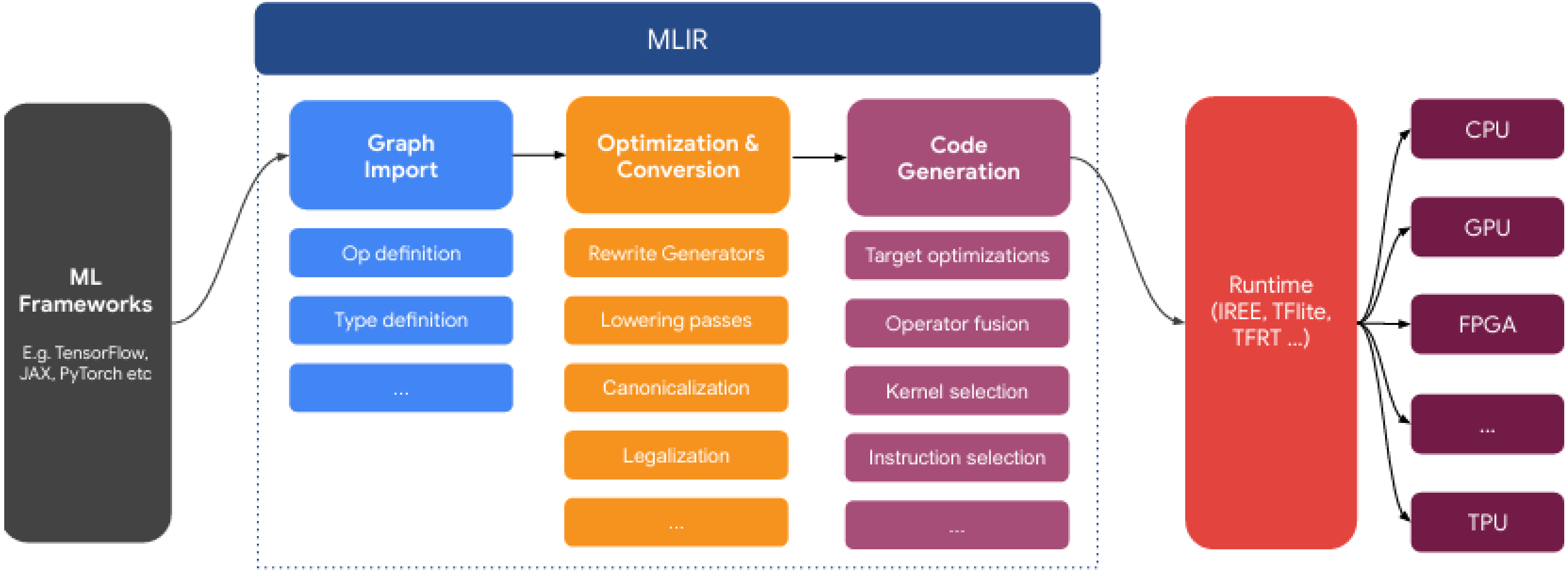
MLIR – Multi-Level Intermediate Representation

- Once we have an optimal IR
  - MLIR can lower it onto the backends such as LLVM for CPU …
  - If the targeting hardware is FPGA, TPU, need vendor-tools for final compilation



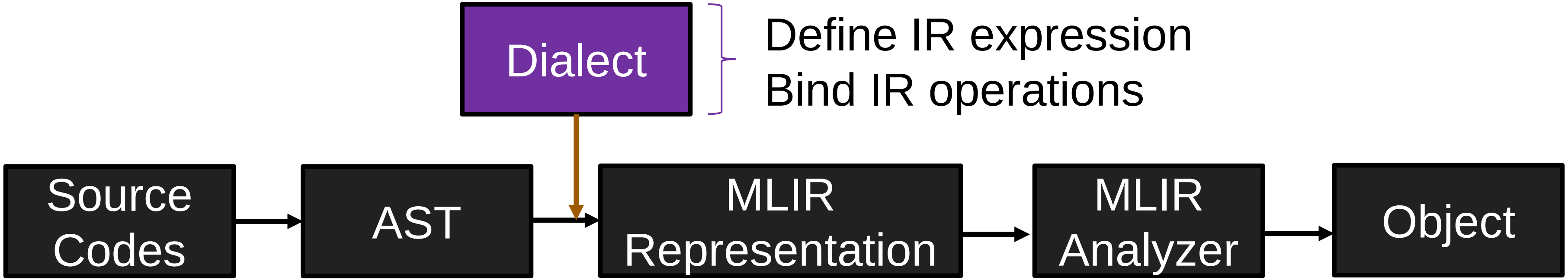
MLIR – Multi-Level Intermediate Representation

- MLIR Compiler Infrastructure
  - A set of optimization/code conversion/code generation pipeline



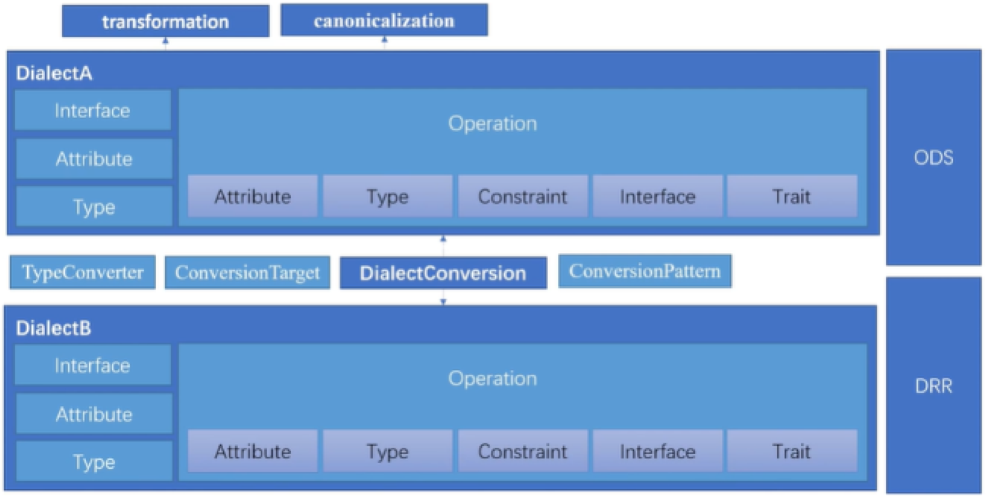
MLIR – Multi-Level Intermediate Representation

- MLIR Dialect
  - One way to express IR from other specific IRs
  - Every IR can be transformed in the corresponding MLIR dialect
  - Each programming language's dialect (Tensor dialect, HLO dialect, LLM IR dialect) is inherent from **mlir::Dialect**
  - AST (Abstract Syntax Tree)



MLIR – Multi-Level Intermediate Representation

- MLIR Dialect
  - DRR(Dynamic Reconstructed Radiography) - transform different dialect
  - ODS(Operation Definition System) – define operation



- 'acc' Dialect
- 'affine' Dialect
- 'amdgpu' Dialect
- 'amx' Dialect
- 'arith' Dialect
- 'arm_neon' Dialect
- 'arm_sve' Dialect
- 'ArmSME' Dialect

MLIR – Multi-Level Intermediate Representation

- Operations
  - gpu.all_reduce (gpu::AllReduceOp)
  - gpu.alloc (gpu::AllocOp)
  - gpu.barrier (gpu::BarrierOp)
  - gpu.binary (gpu::BinaryOp)
  - gpu.block_dim (gpu::BlockDimOp)
  - gpu.block_id (gpu::BlockIdOp)
  - gpu.cluster_block_id (gpu::ClusterBlockIdOp)

- MLIR Operation
  - Output: %tensor
  - Operation: toy.transpose
  - Input: %tensor
  - Transform tensor <2x3xf64> to tensor<3x2xf64>
  - The location of transpose is in “example/file/path”, line 12, 1st word

```
%t_tensor = "toy.transpose"(%tensor) {inplace = true} : (tensor<2x3xf64>) ->
tensor<3x2xf64> loc("example/file/path":12:1)
```

MLIR – Multi-Level Intermediate Representation

- Simple Matmul Kernel

```
M = 2        # Rows in arg0
K = 2816     # Columns in arg0, Rows in arg1
N = 1280     # Columns in arg1

# Matrix multiplication with f16 -> f32 promotion
for i in range(M):
   for j in range(N):
       acc = 0.0  # float32 accumulator
       for k in range(K):
           a = float(arg0[i][k])  # f16 -> f32
           b = float(arg1[k][j])  # f16 -> f32
           acc += a * b
       result[i][j] = acc  # store result as float32
```

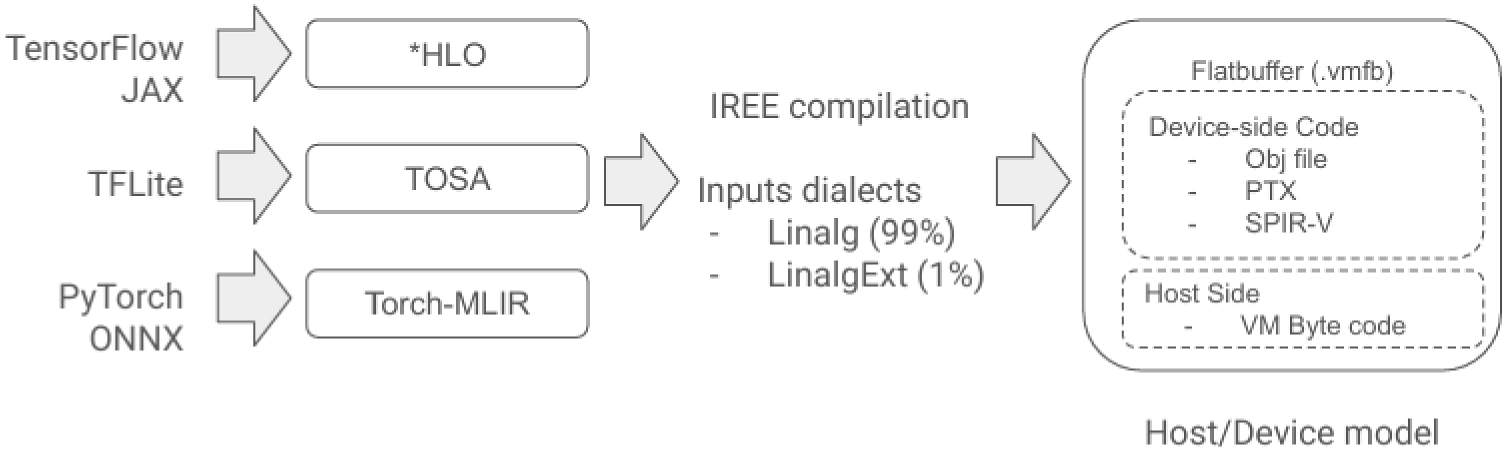
MLIR – Multi-Level Intermediate Representation

- matmul.mlir

```
#map = affine_map<(d0, d1, d2) -> (d0, d2)>
#map1 = affine_map<(d0, d1, d2) -> (d2, d1)>
#map2 = affine_map<(d0, d1, d2) -> (d0, d1)>
func.func @matmul(%arg0: tensor<2x2816xf16>, %arg1: tensor<2816x1280xf16>) -> tensor<2x1280xf32> {
 %cst = arith.constant 0.000000e+00 : f32
 %0 = tensor.empty() : tensor<2x1280xf32>
 %1 = linalg.fill ins(%cst : f32) outs(%0 : tensor<2x1280xf32>) -> tensor<2x1280xf32>
 %2 = linalg.generic {indexing_maps = [#map, #map1, #map2], iterator_types = ["parallel", "parallel",
"reduction"]} ins(%arg0, %arg1 : tensor<2x2816xf16>, tensor<2816x1280xf16>) outs(%1 : tensor<2x1280xf32>) {
 ^bb0(%in: f16, %in_0: f16, %out: f32):
   %3 = arith.extf %in : f16 to f32
   %4 = arith.extf %in_0 : f16 to f32
   %5 = arith.mulf %3, %4 : f32
   %6 = arith.addf %out, %5 : f32
   linalg.yield %6 : f32
 } -> tensor<2x1280xf32>
 return %2 : tensor<2x1280xf32>
}
```

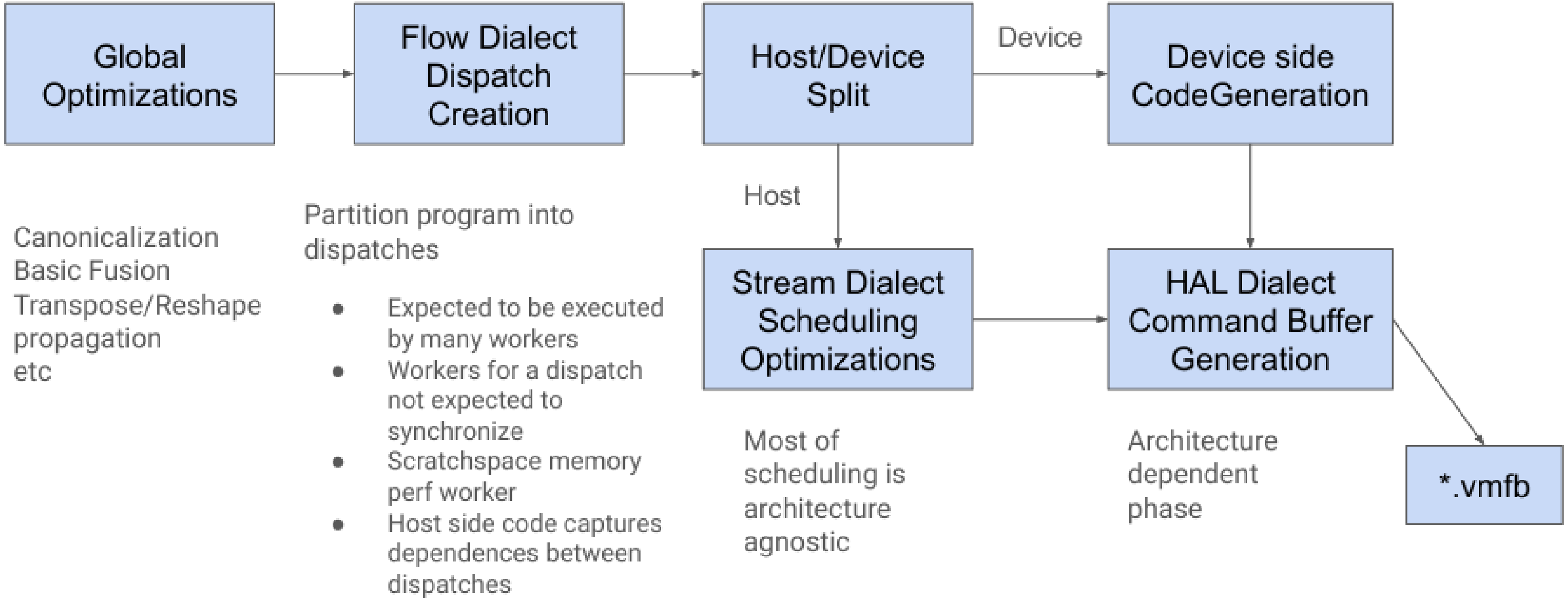
IREE – Intermediate Representation Execution Environment

- IREE
  - A MLIR-based compiler for ML programs
  - Takes ML workloads from various frontends (PyTorch ..) and execute on different backends (x86, Arm, NVIDIA GPUs, AMD GPUs ..)



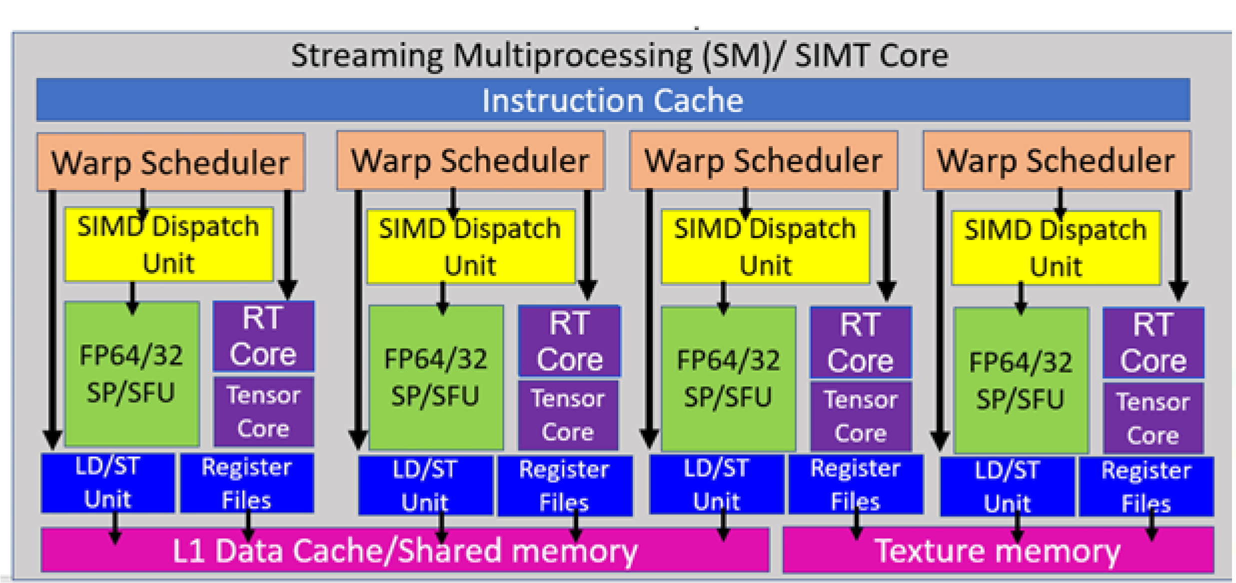
IREE – Intermediate Representation Execution Environment

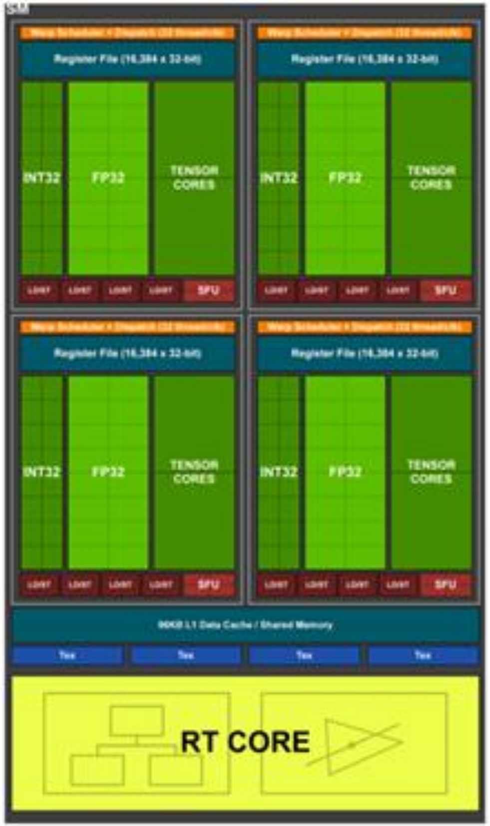
- IREE Compiler Design



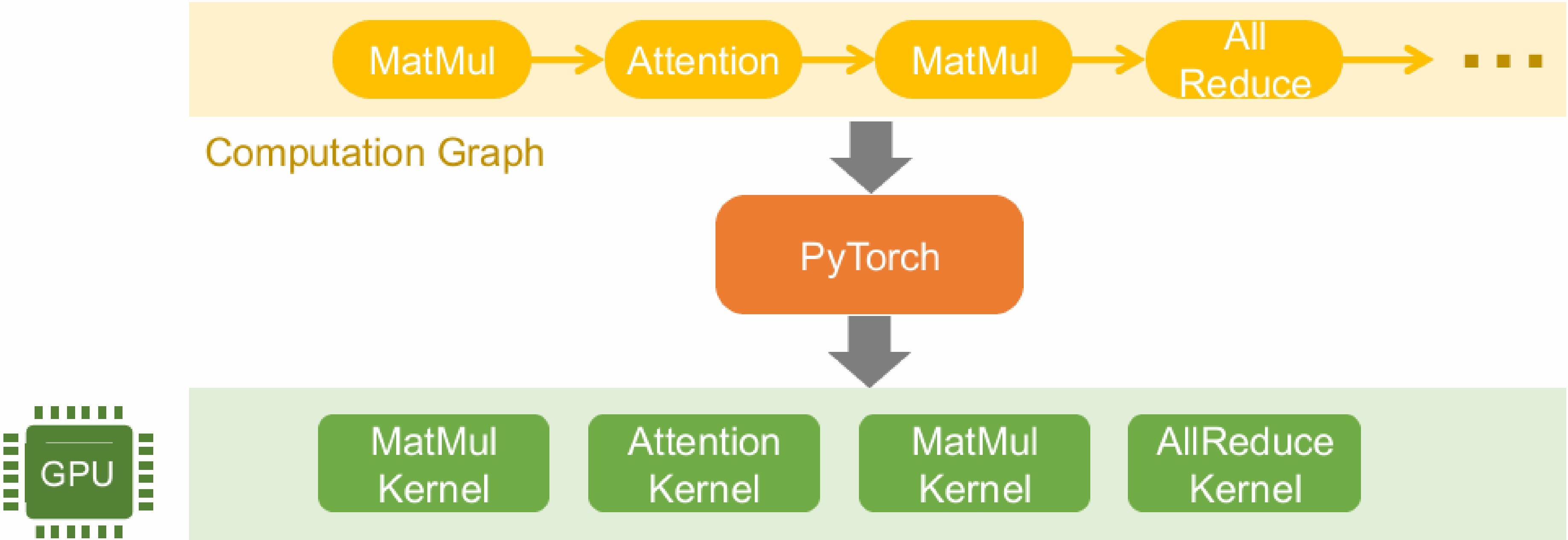
Mega-Kernel on the GPU

- GPU includes multiple specialized engines (CUDA core, Tensor core ..)





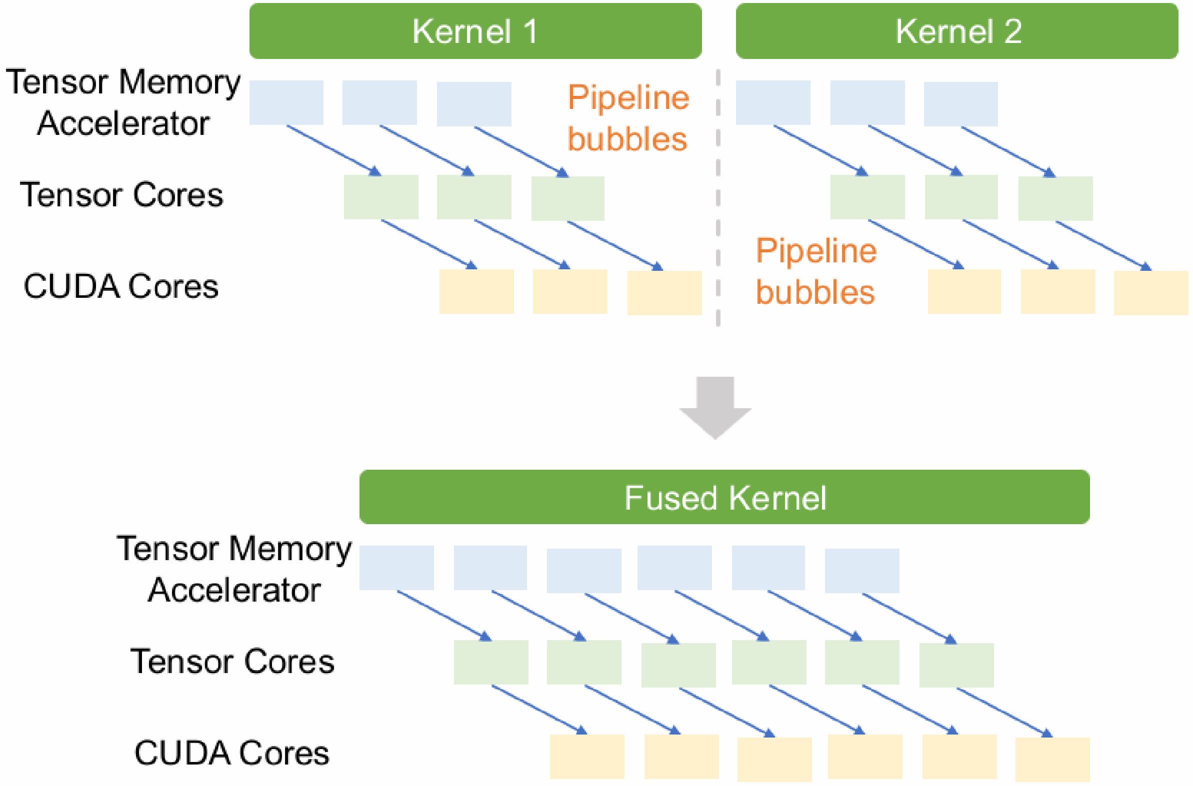
Existing Kernel-Per-Operator Approach



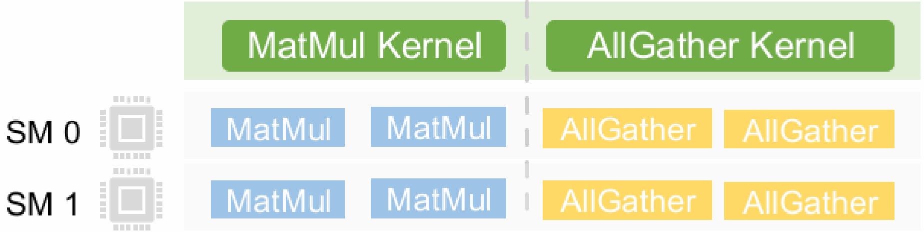
Limitations

No Inter-Layer Pipelining

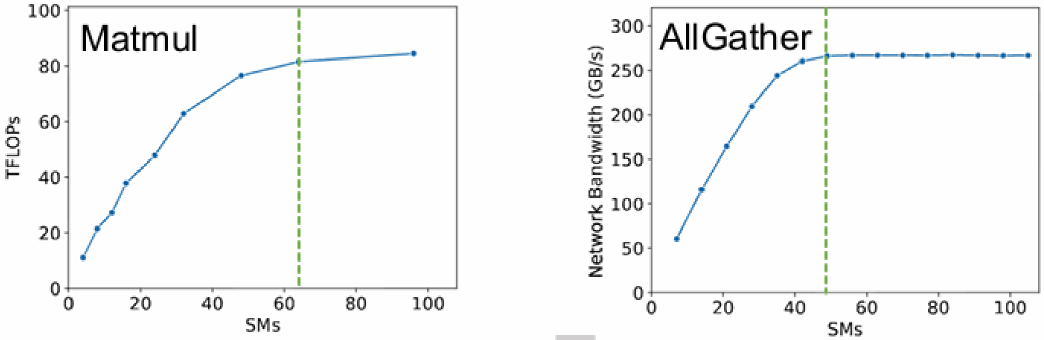
Kernel barriers prevent inter-layer pipelining

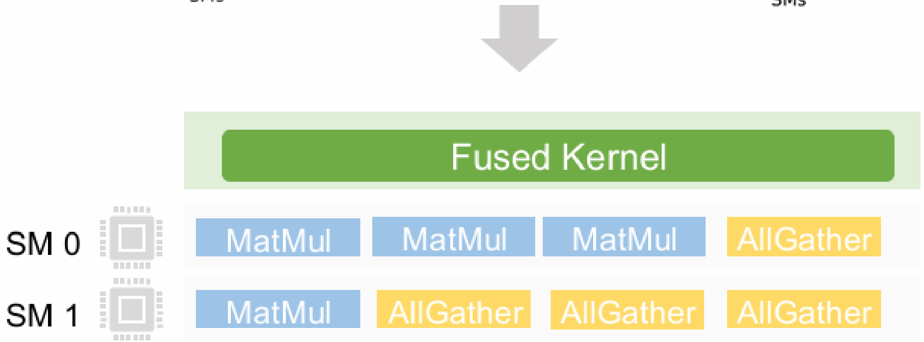


Limitations

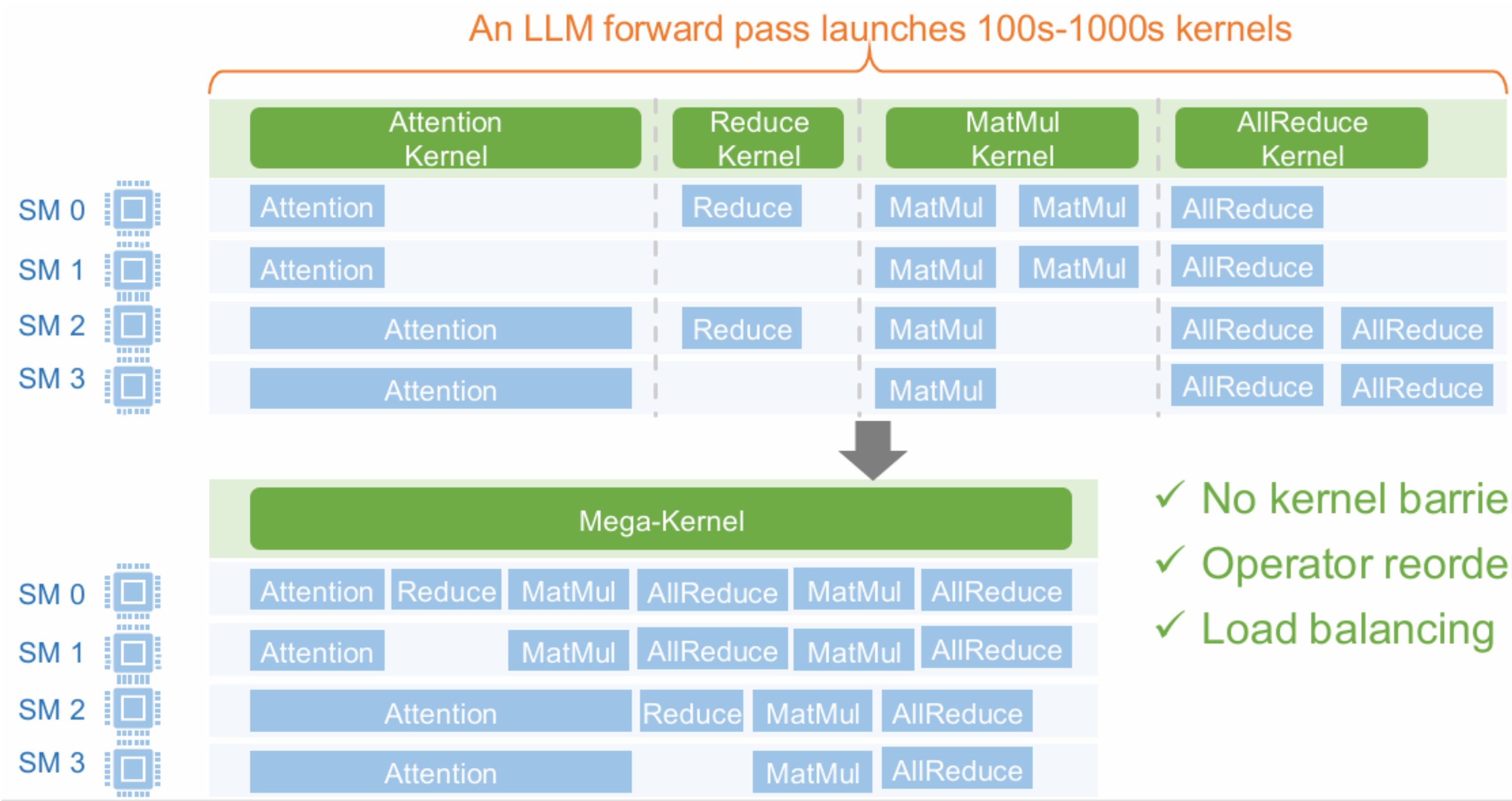


No Overlapping
Coarse-grained dependency prevents comp. & comm. overlap





Kernel-Per-Operator v.s. Mega-Kernel



- ✓ No kernel barriers
- ✓ Operator reordering
- ✓ Load balancing

Key Challenges of Mega-Kernel

1. How to manage dependency?

No kernel barriers in mega-kernel

Task Graph

2. How to handle dynamism?

Continuous batching, prefill/decode,
paged/radix attention, speculative decoding

In-Kernel
Parallel Runtime

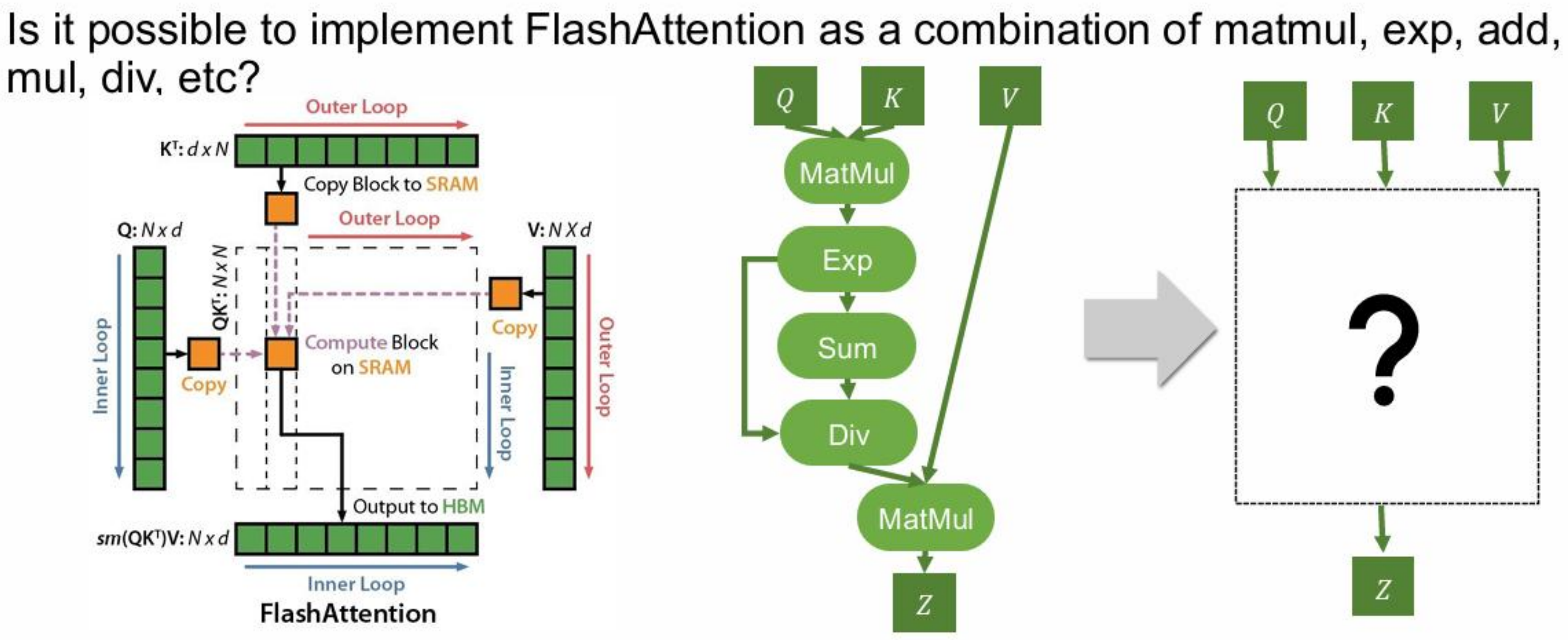
3. How to optimize performance?

Existing compilers target individual kernels

Mirage Superoptimizer*

Mirage: A SuperOptimizer for ML

- Can we represent FlashAttention as a graph optimization?

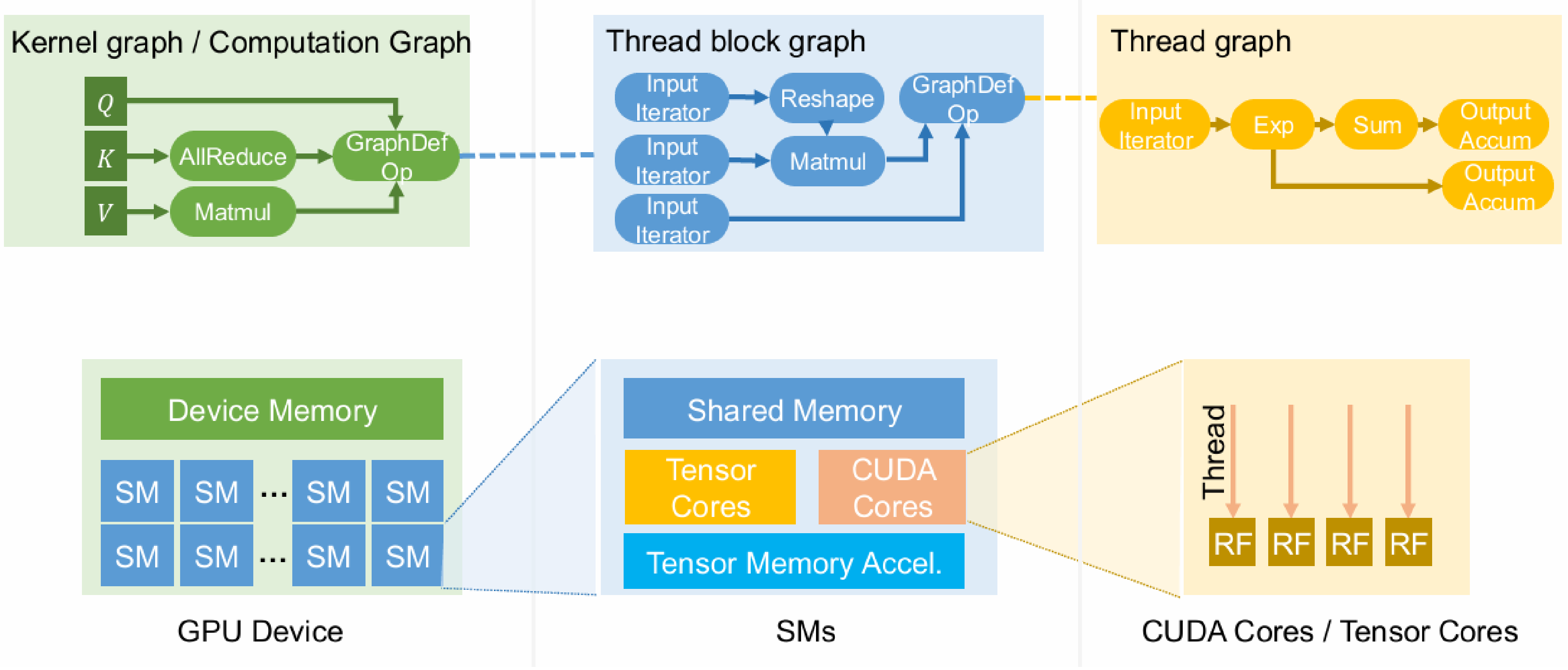


Mirage: A SuperOptimizer for ML

- Key idea: automatically generate highly-optimized GPU kernels for DNNs

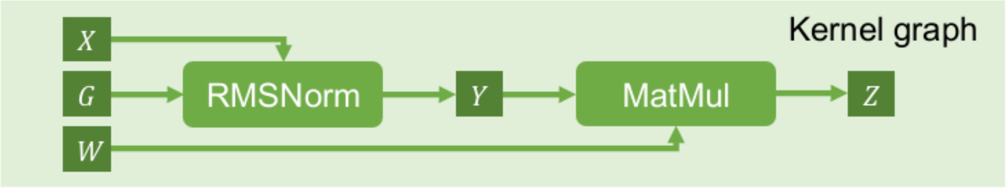
- **Less engineering effort**: thousands lines of CUDA code → a few lines of Python code in Mirage
- **Better performance**: outperform existing systems by 1.1-3.5x
- **Faster adaptation:** day-0 support for new models; no manual effort

Hierarchical Graph Representation



Example: RMSNorm & MatMul in LLMs

- Existing systems launch two kernels since Y does not fit in shared memory



$$y_i = \frac{x_i g_i}{\sqrt{\frac{1}{N}\sum_j x_j^2}}$$

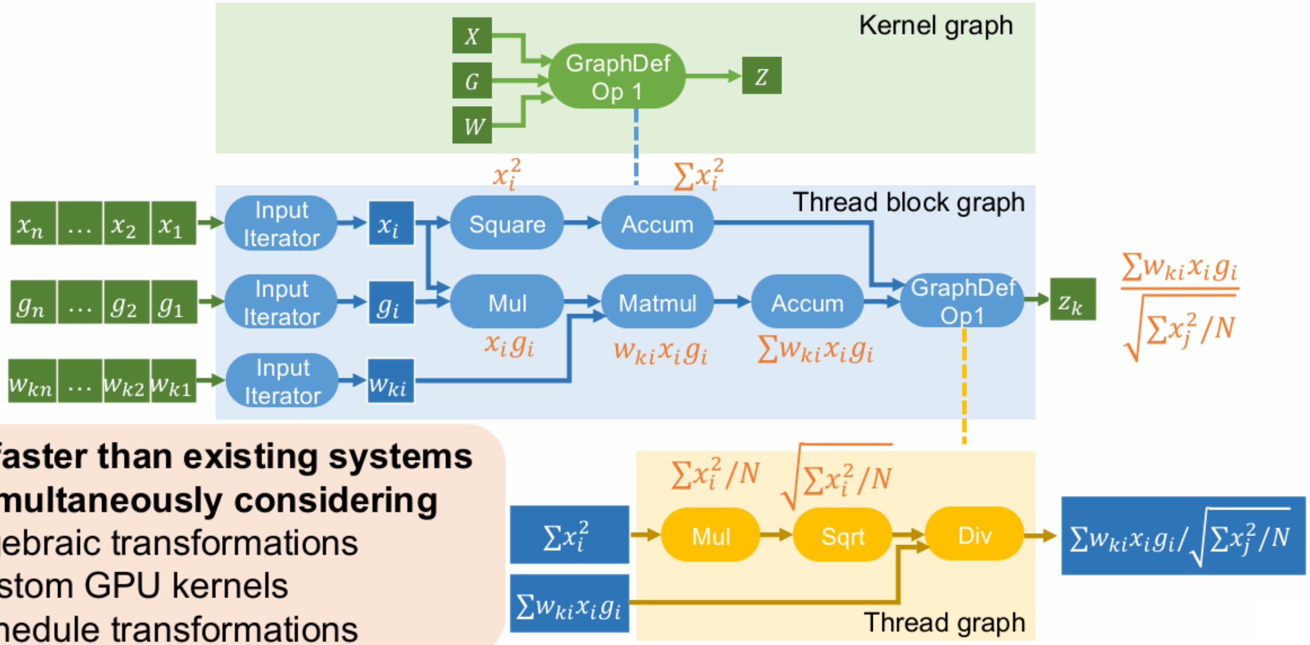
$$z_i = \sum_k w_{ik} y_k$$

Performance issues:

1. No shared memory reuse
2. Kernel launch overhead

μGraph for RMSNorm & MatMul

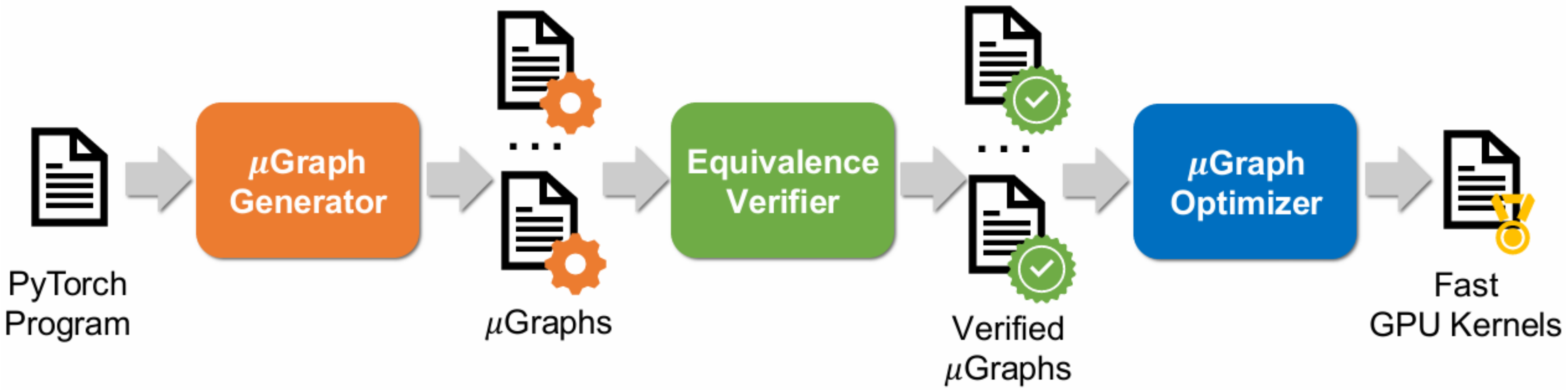
- Existing systems launch two kernels since Y does not fit in shared memory



2.2x faster than existing systems by simultaneously considering
- Algebraic transformations
- Custom GPU kernels
- Schedule transformations

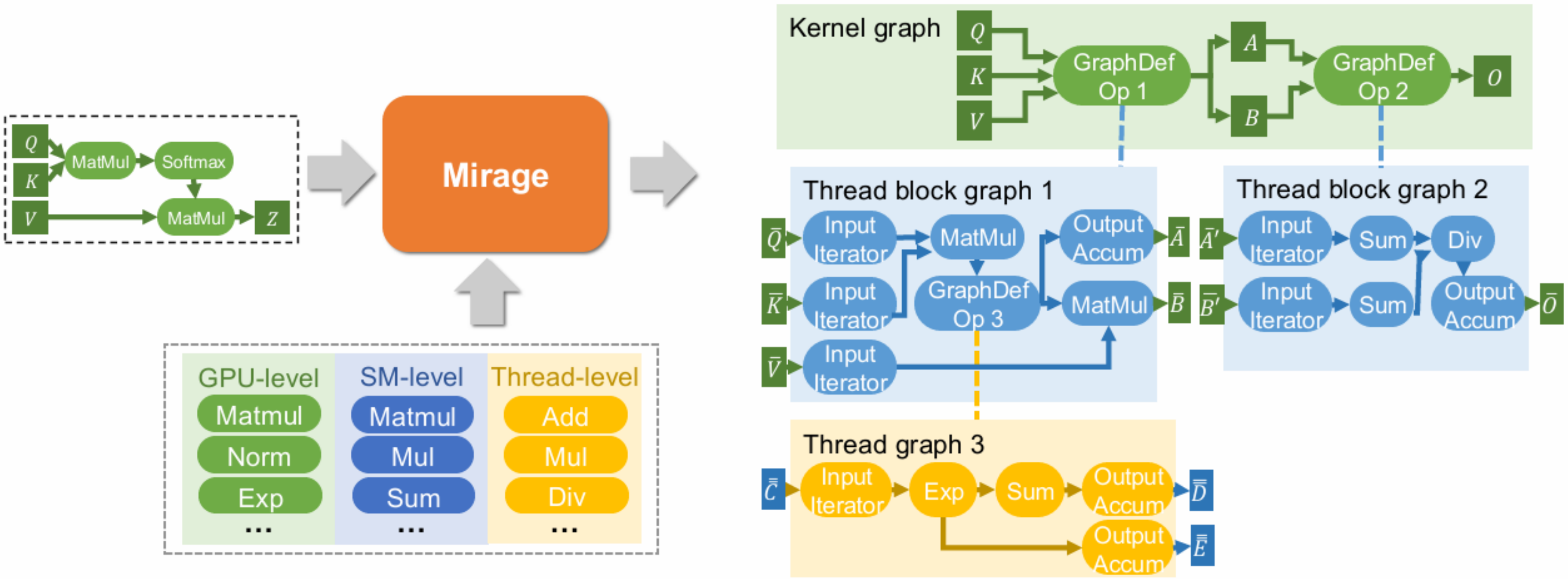
High-Performance μGraph

- Key Challenges to discover High-performance μGraph
  - How to generate potential μGraph?
  - How to verify their correctness?
  - Mirage system



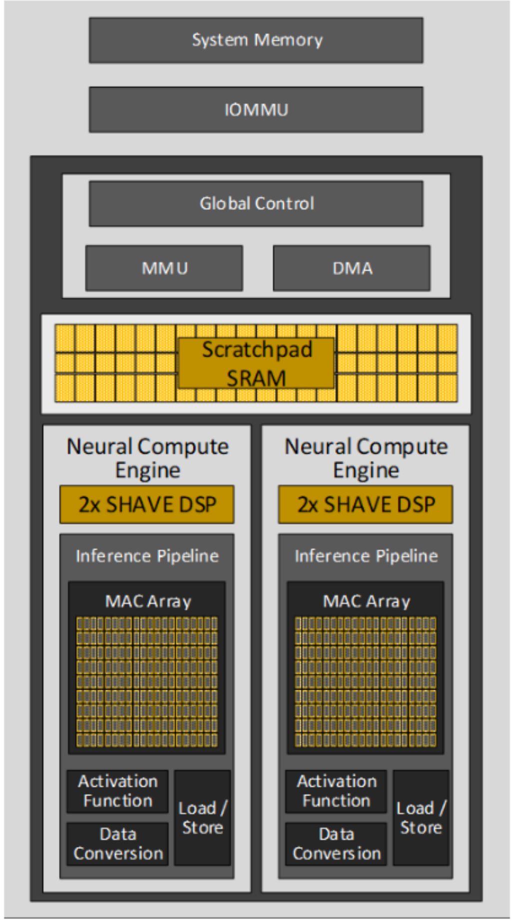
Hardware-Customized μGraphs

- Find μGraphs similar to expert-written implementations for attention on NVIDIA A100 GPU



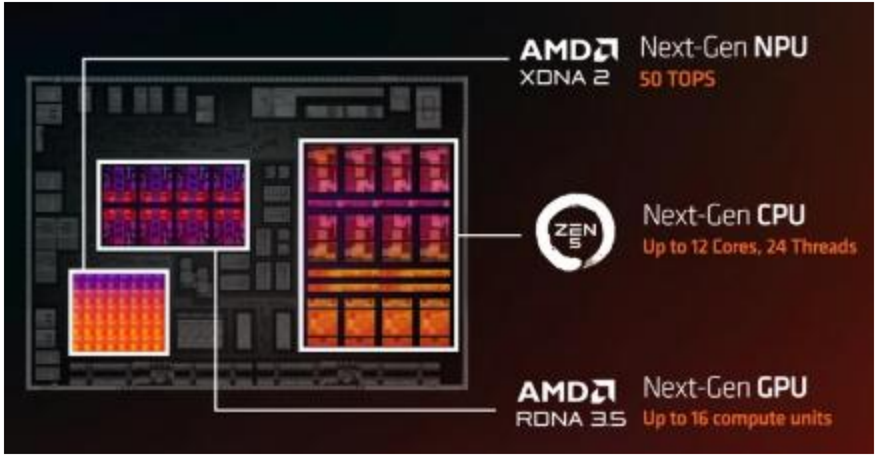
Neural Processing Unit (NPU)

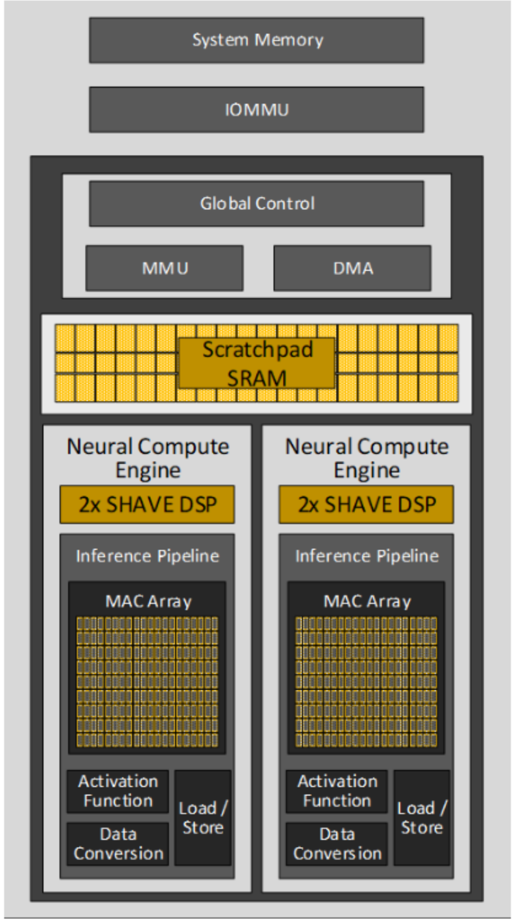
- Intel NPU
  - Hardware Acceleration Blocks
    - Handle GEMM,CONV …
  - Streaming Hybrid Architecture Vector Engines (SHAVE)
    - Perform parallel computing for general needs
  - DMA Engines
    - Moving data between DRAM and software-managed cache



Heterogeneity on ASIC

- NPU
  - MAC engine + Specialized engines
- CPU on AI PC
  - AMD Ryzen AI Pro 300 (CPU+NPU+GPU)





Takeaway Questions

- What are jobs of MLIR?
  - (A) Operator definition
  - (B) Operator lowering
  - (C) Instruction selection
- What are benefits of MegaKernel?
  - (A) Overlapping GPU specialized engine execution
  - (B) GPU register reuse
  - (C) Decrease the kernel launch overhead

Future of AI Compiler

- Future of AI compiler
  - In model inference
    - Ahead-of-Time (AoT) compilation
  - In model training
    - Just-in-Time (JIT) compilation
- The form of IR
  - Need one IR that can support diverse programming language and ML frameworks
  - Good for cross-platform

Future of AI Compiler

- Auto-parallelization
  - Automatic execute ML models through different parallelization approaches
  - Distributed computing (Model training)
  - Parallel computing in one chip
- Auto Code/kernel generation
  - Not only Domain-Specific Language (DSL)
  - Match diverse hardware platforms