

Accelerating Large Language Model Inference with Allo

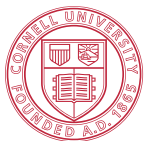
Hongzheng Chen

PI: Zhiru Zhang

Cornell University

UW SAMPL

05/31/2024









Cornell University



The Era of Large Language Models (LLMs)

- ▶ Conventional wisdom suggests that LLMs are MatMul dominated, and GPUs are the optimal choice



Devices	Cost	Energy
 A100 GPU (7nm)	 ~\$20K	 ~250W
 U280 FPGA (16nm)	 ~\$8K	 ~30W

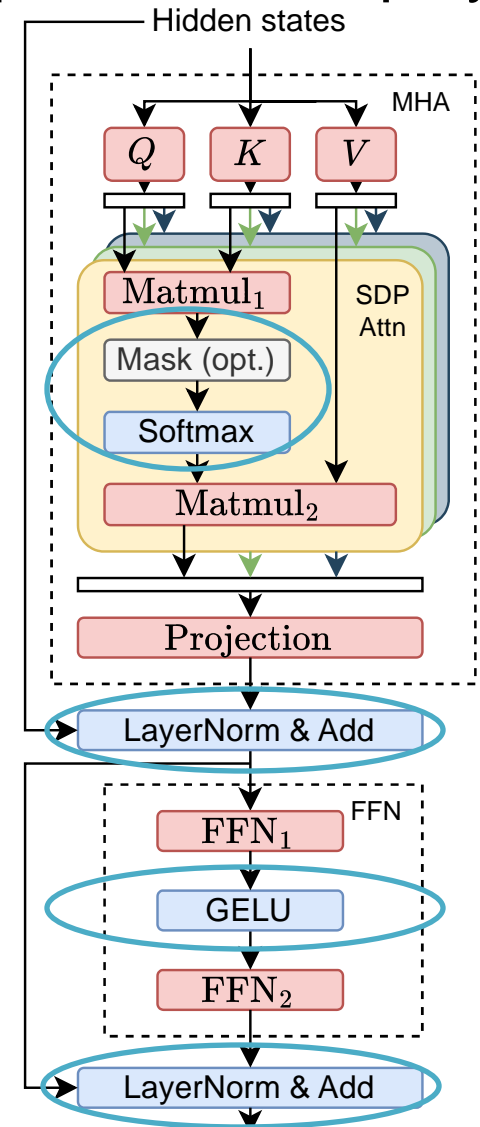
Is FPGA suitable for efficient LLM inference?

Transformer Execution Breakdown on GPU

- ▶ LLMs aren't just about MatMul; non-linear & element-wise operators also play a significant role
 - Low compute-to-memory ratio
 - High kernel launch overheads

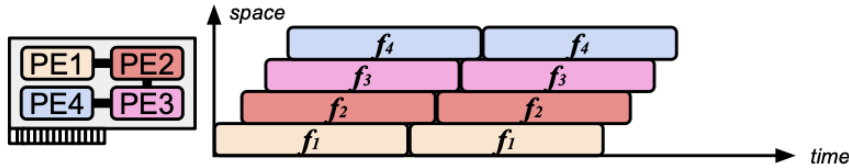
Operator Class	Representative Op	% FLOP	% Run Time
Tensor contraction	GEMM, GEMV	99.80	61.0
Stat. normalization	softmax, layernorm	0.17	25.5
Element-wise	bias, dropout	0.03	13.5

Proportions for operator classes in the BERT model
(implemented in PyTorch, profiled with an NVIDIA V100 GPU)



Opportunities for FPGAs

Model-specific acceleration



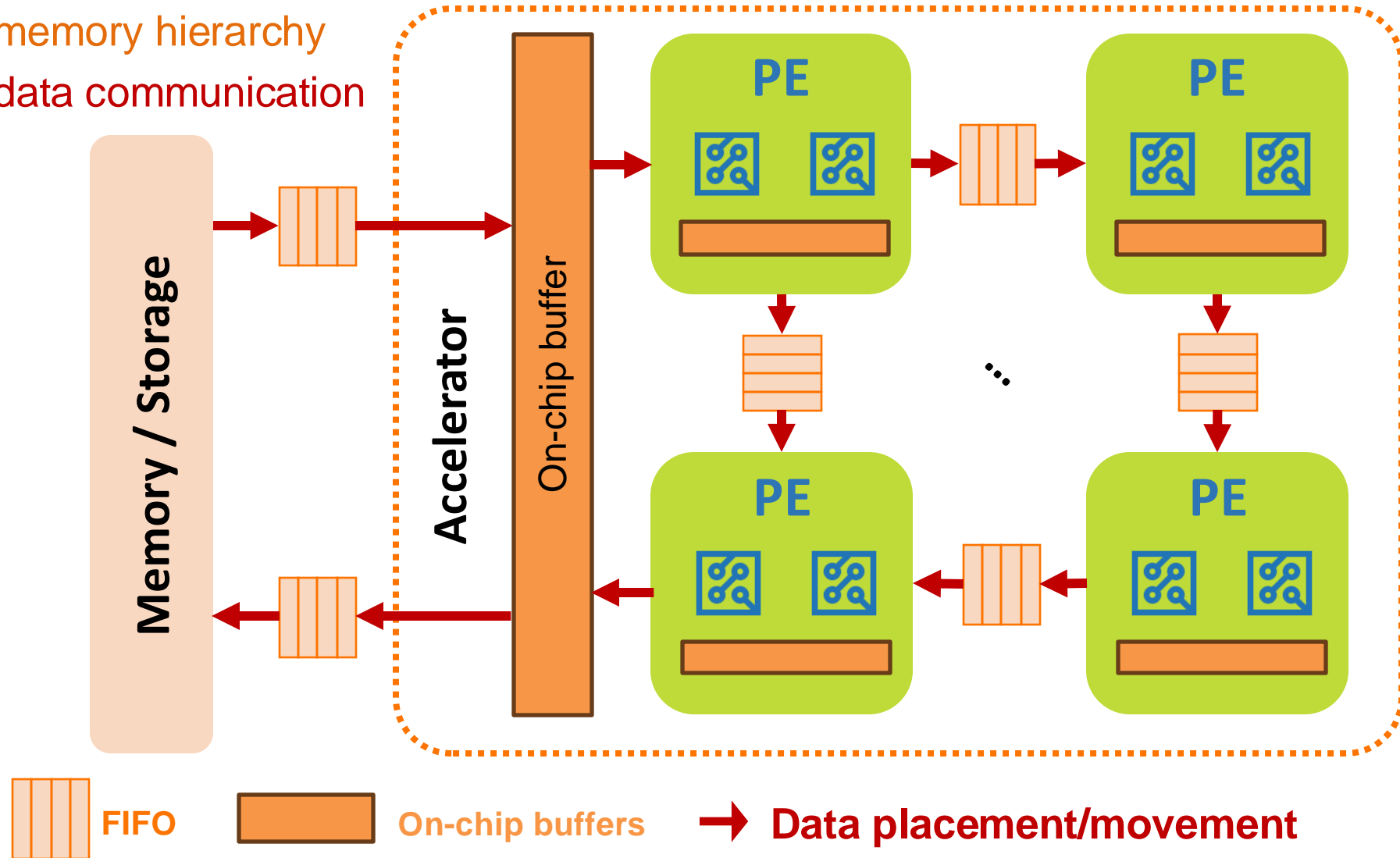
Spatial/Dataflow architecture
Distinct PEs connected with FIFOs

Development cost?

- ✓ Reduced memory access
 - Activation tensors are directly streamed to the next operator
- ✓ High performance & Low energy consumption
 - Standard Transformer building blocks
- ✓ One-time compilation overheads
 - Acceptable overheads compared to long training time

Complexity in Specialized Accelerator Design

- ▶ Accelerator design is different from programming on general processors
 - Custom processing engines (PEs)
 - Custom memory hierarchy
 - Custom data communication



Challenge 1: Balancing Manual Control & Compiler Optimization

```
void systolic_tile(int8_t A_tile[2][768],
int8_t B_tile[768][2],
int8_t C_tile[2][2]) {
#pragma dataflow
#pragma partition variable=A/B/C_tile complete dim=1
hls::stream<int8_t> A_fifo[2][3], B_fifo[2][3];
#pragma stream variable=A/B_fifo depth=3
for (int k4 = 0; k4 < 768; k4++) {
for (int m = 0; m < 2; m++) {
int8_t v105 = A_tile[m][k4];
A_fifo[m][0].write(v105);}
// ... write B_fifo
}}
```

```
for (int Ti = 0; Ti < 2; ++Ti) {
#pragma HLS unroll
for (int Tj = 0; Tj < 2; ++Tj) {
#pragma HLS unroll
```

Custom compute
(Loop tiling & unrolling)

```
// ... load A/B_fifo
PE_kernel(A_in, A_out, B_in, B_out, C, Ti, Tj);
}}}
```

```
void systolic(int8_t A[512][768], int8_t B[768][768],
int8_t C[512][768]
) {
int8_t local_A[2][768], local_B[768][2], local_C[2][2];
for (int mi = 0; mi < 256; mi++) {
for (int ni = 0; ni < 384; ni++) {
// ... load A, B
systolic_tile(local_A, local_B, local_C);
}}}
```

Vanilla GEMM (<1% theoretical peak)

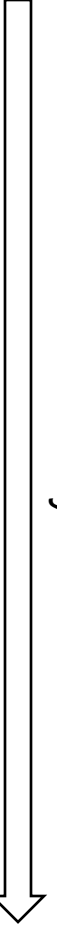
→ Compute optimization (20% peak)

+ Loop tiling

+ Loop unrolling

+ Loop pipelining

Difficulty



Challenge 1: Balancing Manual Control & Compiler Optimization

```
void systolic_tile(int8_t A_tile[2][768],
int8_t B_tile[768][2],
int8_t C_tile[2][2]) {
#pragma dataflow
#pragma partition variable=A/B/C_tile complete dim=1
hls::stream<int8_t> A_fifo[2][3], B_fifo[2][3];
#pragma stream variable=A/B_fifo depth=3
for (int k4 = 0; k4 < 768; k4++) {
for (int m = 0; m < 2; m++) {
int8_t v105 = A_tile[m][k4];
A_fifo[m][0].write(v105);}
// ... write B_fifo
}}
for (int Ti = 0; Ti < 2; ++Ti) {
#pragma HLS unroll
for (int Tj = 0; Tj < 2; ++Tj) {
#pragma HLS unroll
// ... load A/B_fifo
PE_kernel(A_in, A_out, B_in, B_out, C, Ti, Tj);
}}}
```

**Custom memory
(Caching & streaming)**

**Custom compute
(Loop tiling & unrolling)**

```
void systolic(int8_t A[512][768], int8_t B[768][768],
int8_t C[512][768]
) {
int8_t local_A[2][768], local_B[768][2], local_C[2][2];
for (int mi = 0; mi < 256; mi++) {
for (int ni = 0; ni < 384; ni++) {
// ... load A, B
systolic_tile(local_A, local_B, local_C);
}}}
```

Vanilla GEMM (<1% theoretical peak)

→ Compute optimization (20% peak)

+ Loop tiling

+ Loop unrolling

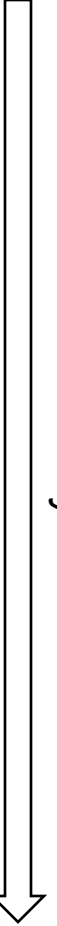
+ Loop pipelining

→ Memory optimization (50% peak)

+ Multi-level caching

+ Memory partitioning

Difficulty



Challenge 1: Balancing Manual Control & Compiler Optimization

```
void systolic_tile(int8_t A_tile[2][768],  
int8_t B_tile[768][2],  
int8_t C_tile[2][2]) {
```

```
#pragma dataflow
```

```
#pragma partition variable=A/B/C_tile complete dim=1
```

```
hls::stream<int8_t> A_fifo[2][3], B_fifo[2][3];
```

```
#pragma stream variable=A/B_fifo depth=3
```

Custom comm.

```
for (int k4 = 0; k4 < 768; k4++) {
```

```
for (int m = 0; m < 2; m++) {
```

```
int8_t v105 = A_tile[m][k4];
```

```
A_fifo[m][0].write(v105);
```

```
// ... write B_fifo
```

```
}}
```

Custom memory
(Caching & streaming)

```
for (int Ti = 0; Ti < 2; ++Ti) {
```

```
#pragma HLS unroll
```

```
for (int Tj = 0; Tj < 2; ++Tj) {
```

```
#pragma HLS unroll
```

```
// ... load A/B_fifo
```

```
PE_kernel(A_in, A_out, B_in, B_out, C, Ti, Tj);
```

```
}}}
```

Custom compute
(Loop tiling & unrolling)

```
void systolic(int8_t A[512][768], int8_t B[768][768],
```

```
int8_t C[512][768]
```

```
) {
```

```
int8_t local_A[2][768], local_B[768][2], local_C[2][2];
```

```
for (int mi = 0; mi < 256; mi++) {
```

```
for (int ni = 0; ni < 384; ni++) {
```

```
// ... load A, B
```

```
systolic_tile(local_A, local_B, local_C);
```

```
}}}
```

Vanilla GEMM (<1% theoretical peak)

→ Compute optimization (20% peak)

+ Loop tiling

+ Loop unrolling

+ Loop pipelining

→ Memory optimization (50% peak)

+ Multi-level caching

+ Memory partitioning

→ Dataflow optimization (95% peak)

+ Function pipelining

+ Data streaming

+ Data packing (vectorization)

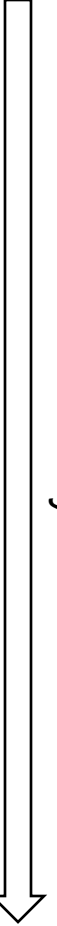
+ Memory coalescing

+ Daisy chaining

~500 lines of HLS code for a 2x2 systolic array

Unproductive, target-specific, hard to maintain

Difficulty



Challenge 1: Balancing Manual Control & Compiler Optimization

```
void systolic_tile(int8_t A_tile[2][768],
int8_t B_tile[768][2],
int8_t C_tile[2][2]) {
```

```
#pragma dataflow
```

```
#pragma partition variable=A/B/C_tile complete dim=1
```

```
hls::stream<int8_t> A_fifo[2][3], B_fifo[2][3];
```

```
#pragma stream variable=A/B_fifo depth=3
```

Custom comm.

```
for (int k4 = 0; k4 < 768; k4++) {
    for (int m = 0; m < 2; m++) {
        int8_t v105 = A_tile[m][k4];
        A_fifo[m][0].write(v105);
        // ... write B_fifo
    }
}
```

Custom memory
(Caching & streaming)

```
for (int Ti = 0; Ti < 2; ++Ti) {
    #pragma HLS unroll
    for (int Tj = 0; Tj < 2; ++Tj) {
        #pragma HLS unroll
```

Custom compute
(Loop tiling & unrolling)

```
// ... load A/B_fifo
```

```
PE_kernel(A_in, A_out, B_in, B_out, C, Ti, Tj);
```

```
}}}
```

```
void systolic(int8_t A[512][768], int8_t B[768][768],
int8_t C[512][768]
) {
```

```
int8_t local_A[2][768], local_B[768][2], local_C[2][2];
```

```
for (int mi = 0; mi < 256; mi++) {
```

```
    for (int ni = 0; ni < 384; ni++) {
```

```
        // ... load A, B
```

```
        systolic_tile(local_A, local_B, local_C);
```

```
    }}
```

Vanilla GEMM (<1% theoretical peak)

→ Compute optimization (20% peak)
+ Loop tiling **Existing HLS compiler**
+ Loop unrolling **e.g., ScaleHLS [HPCA'22]**
+ Loop pipelining

→ Memory optimization (50% peak)
+ Multi-level caching
+ Memory partitioning

→ Dataflow optimization (95% peak)
+ Function pipelining
+ Data streaming
+ Data packing (vectorization)
+ Memory coalescing
+ Daisy chaining

~500 lines of HLS code for a 2x2 systolic array
Unproductive, target-specific, hard to maintain

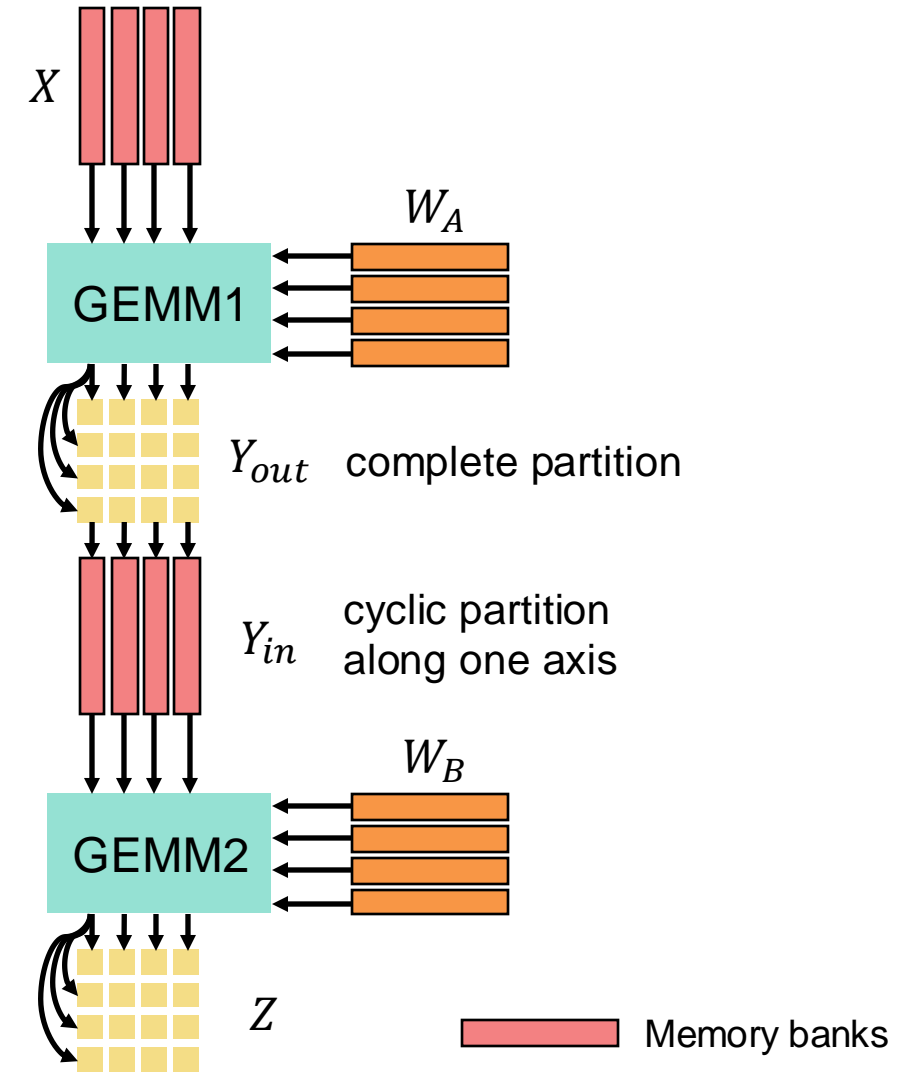
Difficulty

Challenge 2: Bridging the Gap from Single-Kernel to Multi-Kernel Design

```
void GEMM1(int8_t A1[512][768], int8_t B1[768][3072],
           int8_t C1[512][3072]) {
    #pragma partition var=A1 cyclic factor=16 dim=0
    // ... matmul computation
}

void GEMM2(int8_t A2[512][3072], int8_t B2[3072][768],
           int8_t C2[512][768]) {
    #pragma partition var=A2 cyclic factor=16 dim=0
    // ... matmul computation
}

void top(int8_t X[512][768], int8_t W_A[768][3072],
         int8_t W_B[3072][768], int8_t Z[512][768]) {
    int8_t Y[512][3072];
    GEMM1(X, W_A, Y);
    GEMM2(Y, W_B, Z);
}
```



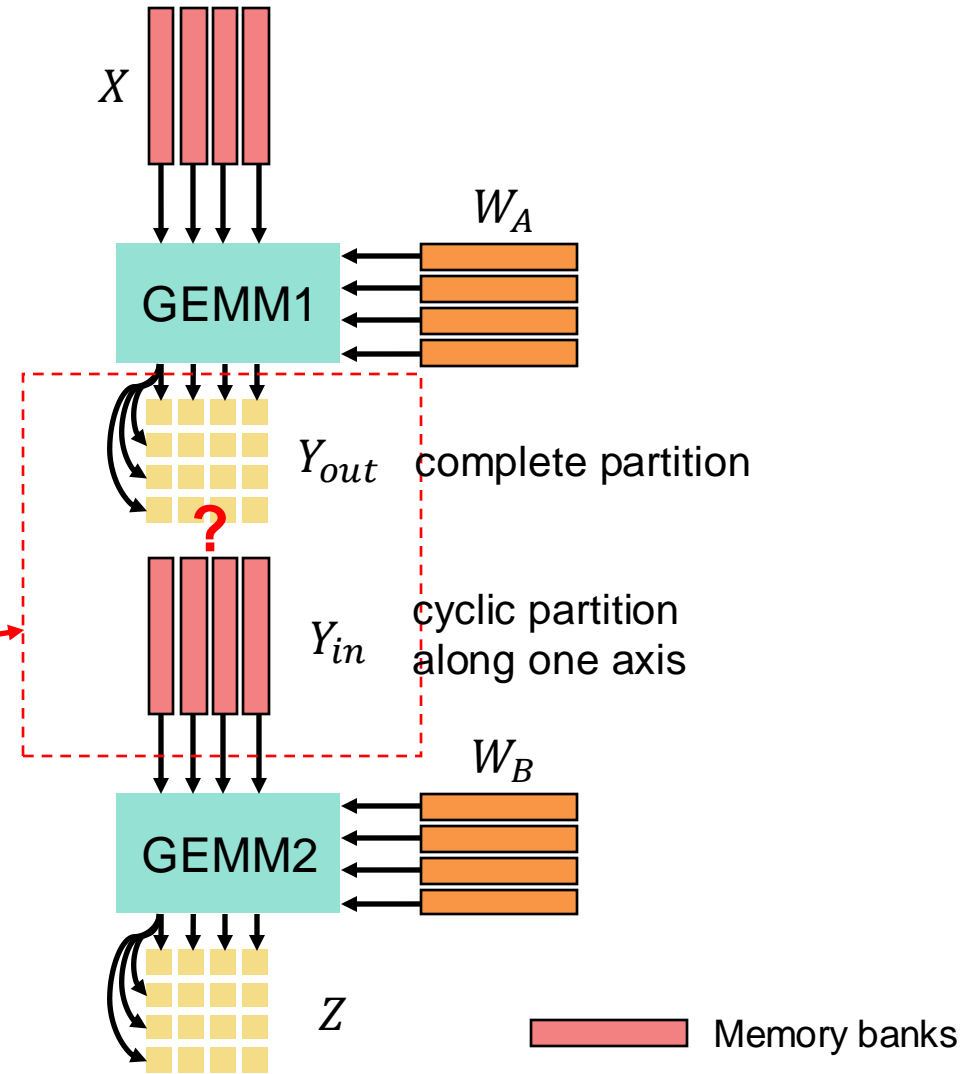
Challenge 2: Bridging the Gap from Single-Kernel to Multi-Kernel Design

```
void GEMM1(int8_t A1[512][768], int8_t B1[768][3072],
          int8_t C1[512][3072]) {
  #pragma partition var=A1 cyclic factor=16 dim=0
  // ... matmul computation
}

void GEMM2(int8_t A2[512][3072], int8_t B2[3072][768],
          int8_t C2[512][768]) {
  #pragma partition var=A2 cyclic factor=16 dim=0
  // ... matmul computation
}

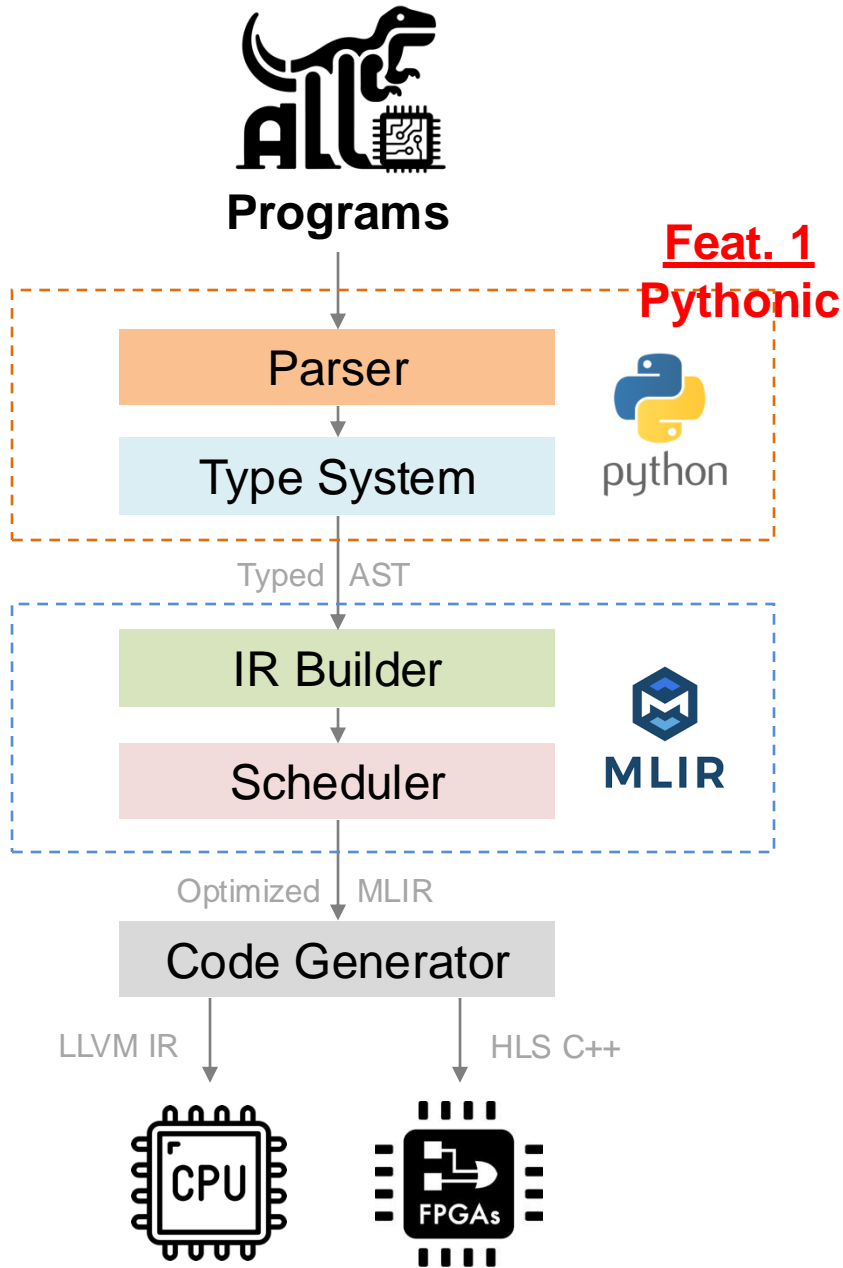
void top(int8_t X[512][768], int8_t W_A[768][3072],
         int8_t W_B[3072][768], int8_t Z[512][768]) {
  int8_t Y[512][3072];
  GEMM1(X, W_A, Y);
  GEMM2(Y, W_B, Z);
}
```

Conflicting layouts



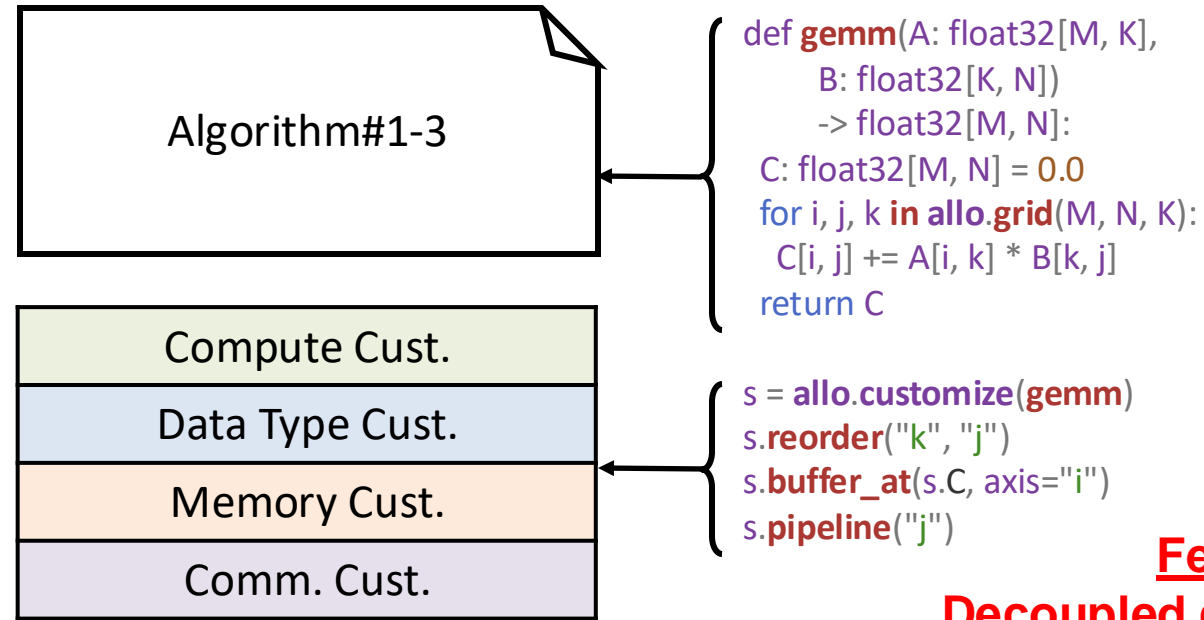
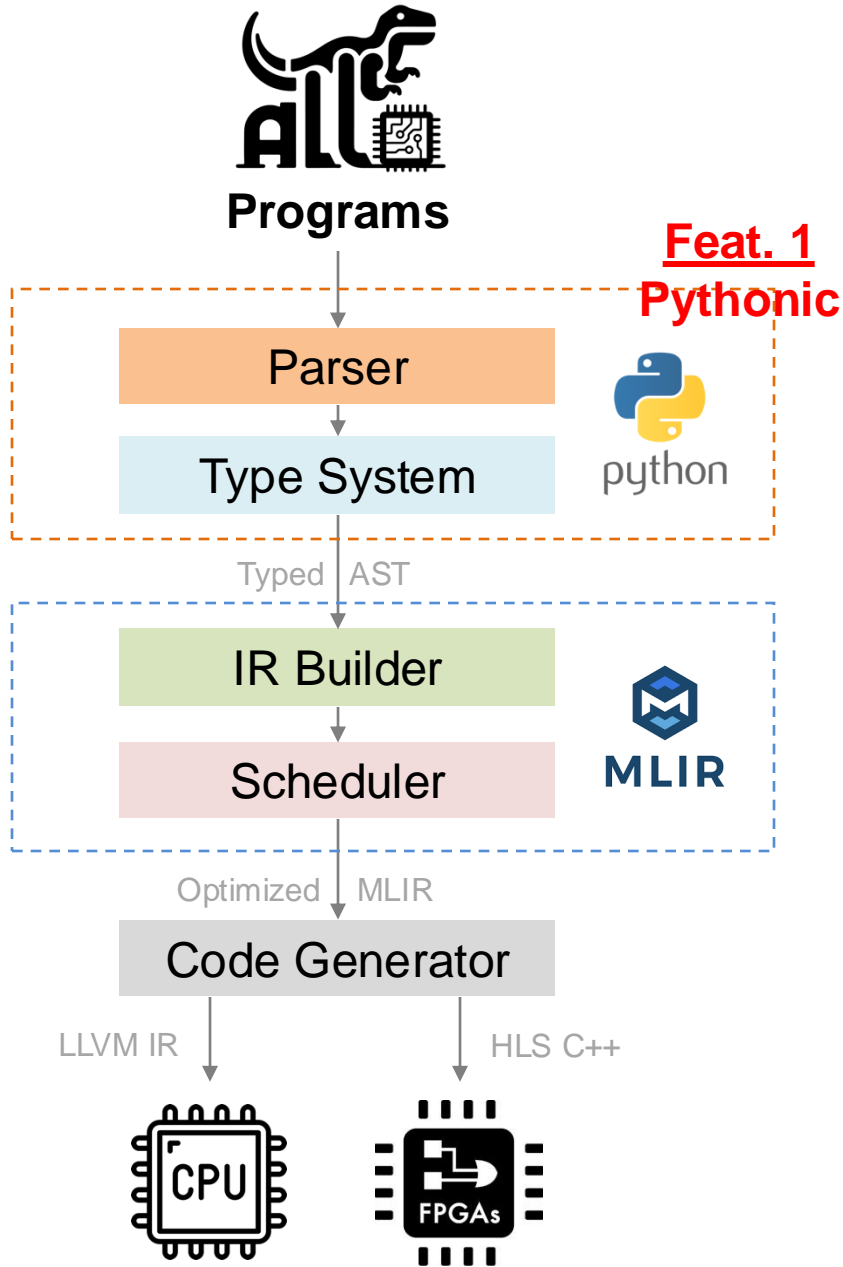
Inconsistent function interface leads to unexpected performance degradation
(e.g., layout transformation overheads)

Allo Accelerator Design Language (ADL) and Compiler

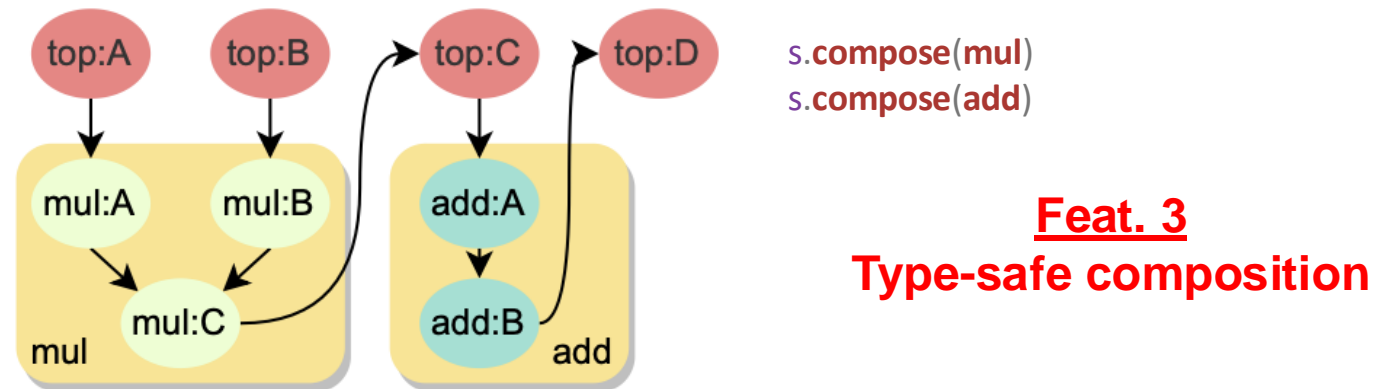
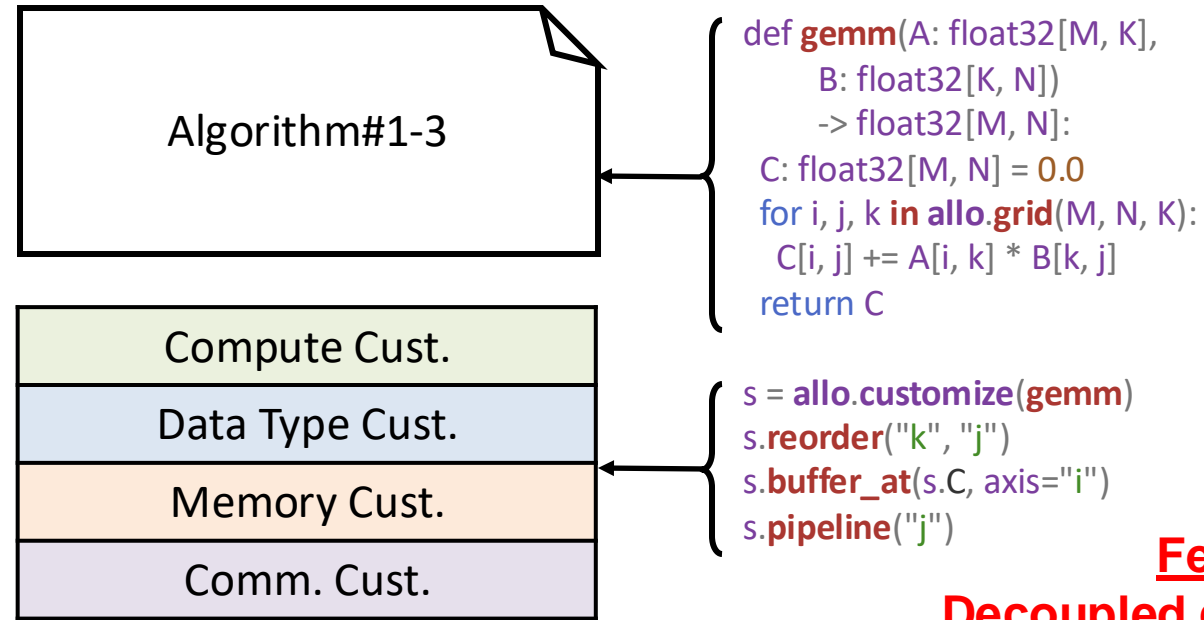
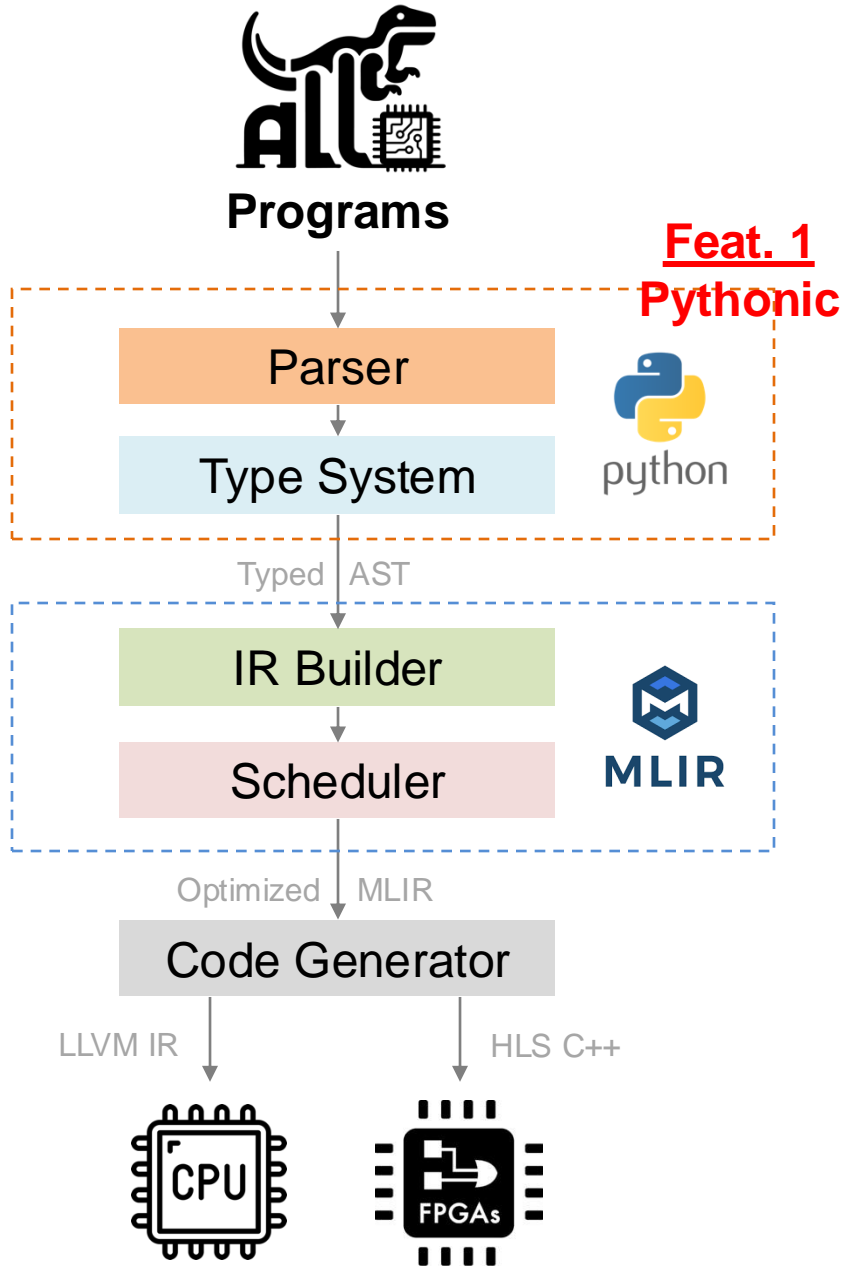


* Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, Zhiru Zhang, “Allo: A Programming Model for Composable Accelerator Design”, PLDI, 2024.

Allo Accelerator Design Language (ADL) and Compiler



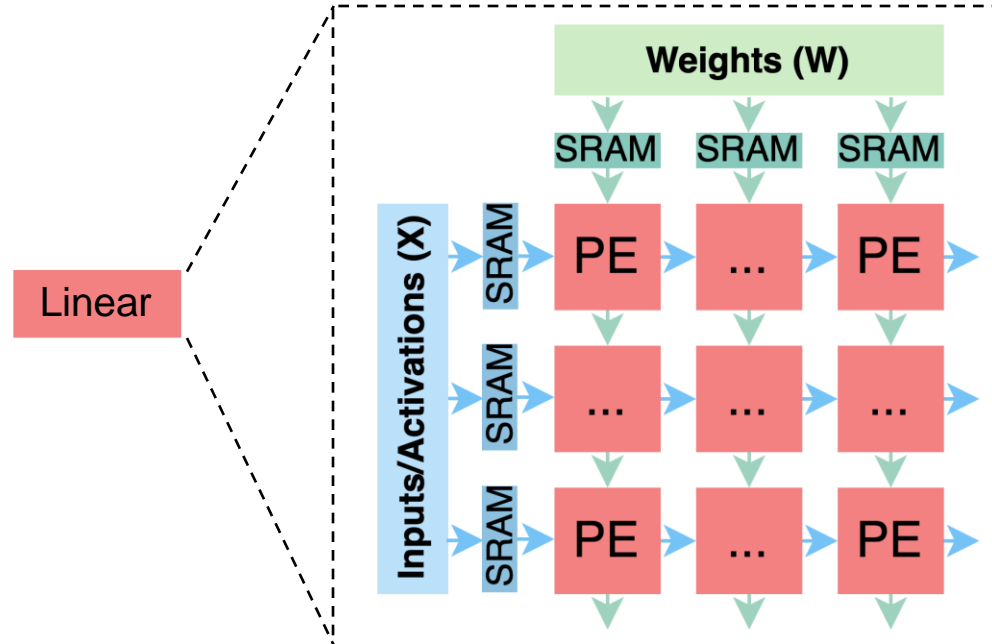
Allo Accelerator Design Language (ADL) and Compiler



Goal: Design a High-Performance LLM Accelerator

Step 1: Construct building blocks

- ▶ Performance
- ▶ Correctness
- ▶ Reusability
- Linear operators



- Non-linear operators

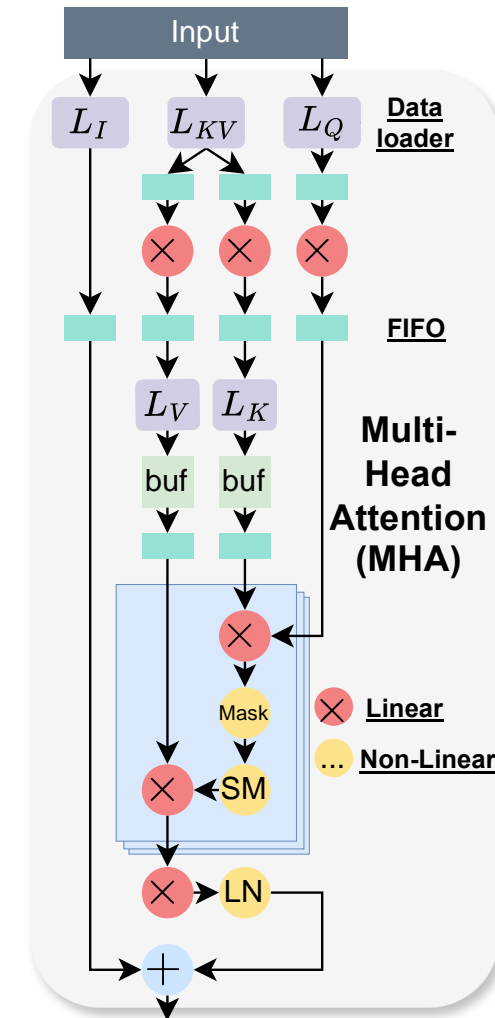
Softmax

LayerNorm

GELU

Step 2: Ensemble into a complete design

- ▶ Connect different operators bottom-up



Transforming GEMM to Systolic Array

→ Algorithm specification

```
def gemm(A: int8[M, K],  
         B: int8[K, N],  
         C: int16[M, N]):  
    for i, j in allo.grid(M, N, "PE"):  
        for k in range(K):  
            C[i, j] += A[i, k] * B[k, j]
```



Pythonic: No need to learn a new DSL!

- Free-form imperative programming
- Python native keywords (e.g., for, if, else)

Transforming GEMM to Systolic Array

→ Algorithm specification

```
def gemm(A: int8[M, K],  
        B: int8[K, N],  
        C: int16[M, N]):  
    for i, j in algo.grid(M, N, "PE"):  
        for k in range(K):  
            C[i, j] += A[i, k] * B[k, j]
```



Pythonic: No need to learn a new DSL!

- Free-form imperative programming
- Python native keywords (e.g., for, if, else)
- Explicit type annotation

Declarative programming

e.g., TVM TE [OSDI'18], HeteroCL [FPGA'19]

```
k = te.reduce_axis((0, K), "k")  
A = te.placeholder((M, K), name="A")  
B = te.placeholder((K, N), name="B")  
C = te.compute((M, N), lambda x, y: \  
    te.sum(A[x, k] * B[k, y], axis=k), \  
    name="C")
```



Not straightforward, hard to express control flow

Transforming GEMM to Systolic Array

→ Algorithm specification

```
def gemm(A: int8[M, K],  
        B: int8[K, N],  
        C: int16[M, N]):  
    for i, j in allo.grid(M, N, "PE"):  
        for k in range(K):  
            C[i, j] += A[i, k] * B[k, j]
```



→ Schedule construction

```
s = allo.customize(gemm)  
print(s.module)
```

Transforming GEMM to Systolic Array

→ Algorithm specification

```
def gemm(A: int8[M, K],  
         B: int8[K, N],  
         C: int16[M, N]):  
    for i, j in allo.grid(M, N, "PE"):  
        for k in range(K):  
            C[i, j] += A[i, k] * B[k, j]
```



→ Schedule construction

```
s = allo.customize(gemm)  
print(s.module)
```

→ Real-time MLIR Module Inspection

```
module {  
  func.func @gemm(%arg0: memref<1024x1024xi8>,  
                  %arg1: memref<1024x1024xi8>,  
                  %arg2: memref<1024x1024xi16>) {  
    affine.for %arg3 = 0 to 1024 {  
      affine.for %arg4 = 0 to 1024 {  
        affine.for %arg5 = 0 to 1024 {  
          %0 = affine.load %arg0[%arg3, %arg5]  
          %1 = affine.load %arg1[%arg5, %arg4]  
          %2 = arith.extsi %0 : i8 to i16  
          %3 = arith.extsi %1 : i8 to i16  
          %4 = arith.muli %2, %3 : i16  
          %5 = affine.load %arg2[%arg3, %arg4]  
          %6 = arith.addi %5, %4 : i16  
          affine.store %6, %arg2[%arg3, %arg4]  
        } {loop_name = "k", op_name = "Sk"}  
      } {loop_name = "j"}  
    } {loop_name = "i", op_name = "PE"}  
    return  
  }  
}
```



Transforming GEMM to Systolic Array

→ Algorithm specification

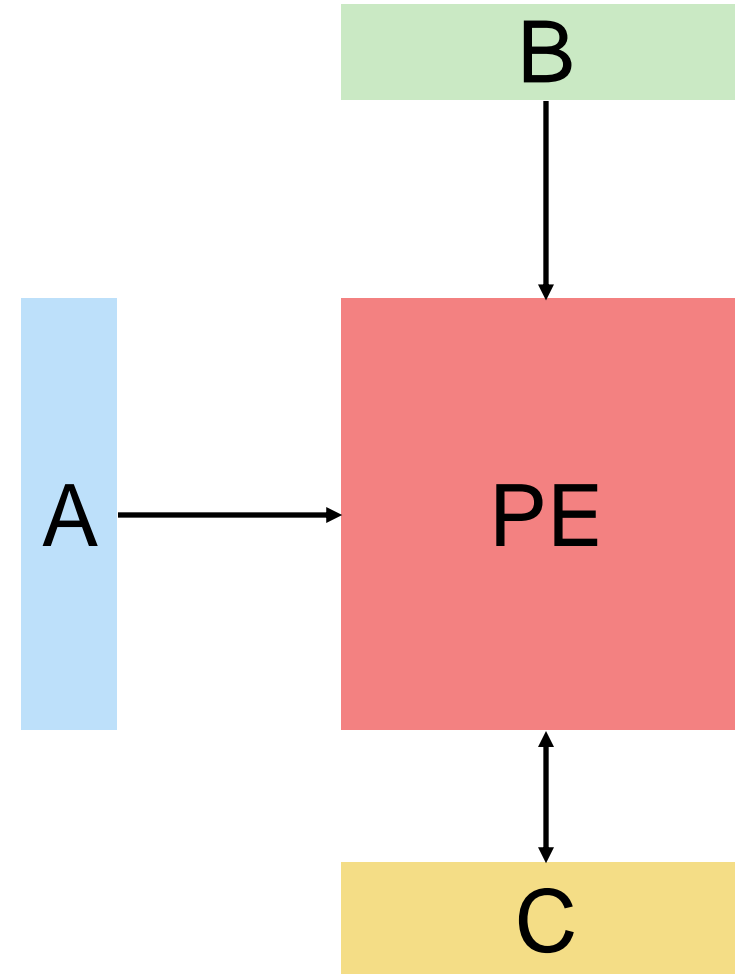
```
def gemm(A: int8[M, K],  
        B: int8[K, N],  
        C: int16[M, N]):  
    for i, j in allo.grid(M, N, "PE"):  
        for k in range(K):  
            C[i, j] += A[i, k] * B[k, j]
```



→ Schedule construction

```
s = allo.customize(gemm)
```

→ Architectural Diagram



Transforming GEMM to Systolic Array

→ Algorithm specification

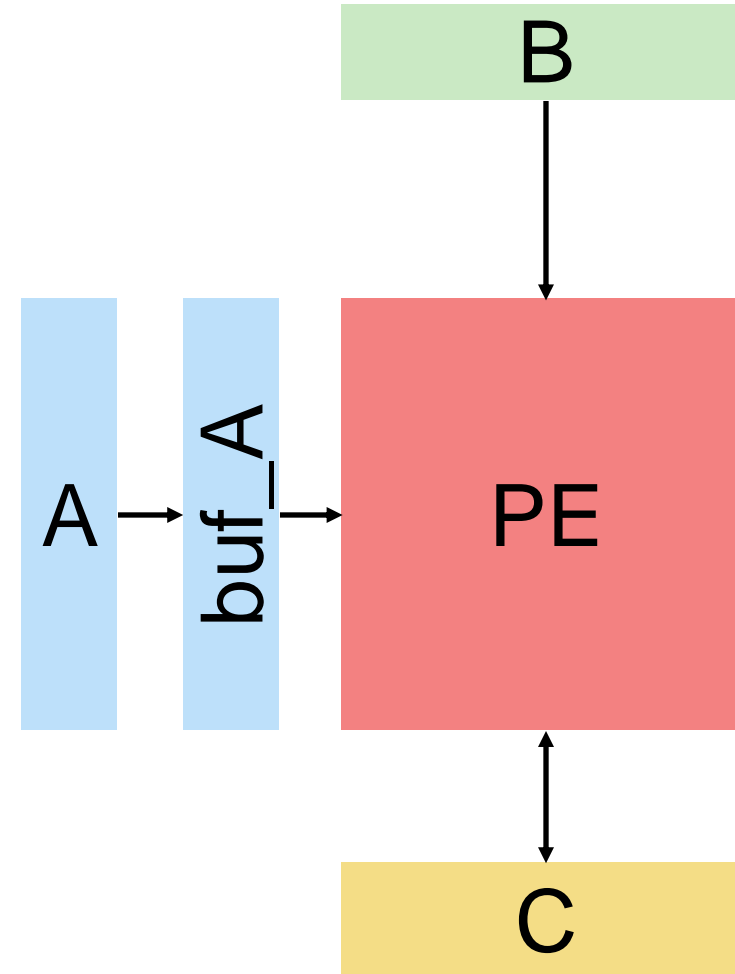
```
def gemm(A: int8[M, K],  
        B: int8[K, N],  
        C: int16[M, N]):  
    for i, j in allo.grid(M, N, "PE"):  
        for k in range(K):  
            C[i, j] += A[i, k] * B[k, j]
```



→ Schedule construction

```
s = allo.customize(gemm)  
buf_A = s.buffer_at(s.A, "j")
```

→ Architectural Diagram



Transforming GEMM to Systolic Array

→ Algorithm specification

```
def gemm(A: int8[M, K],
        B: int8[K, N],
        C: int16[M, N]):
    for i, j in allo.grid(M, N, "PE"):
        for k in range(K):
            C[i, j] += A[i, k] * B[k, j]
```



→ Schedule construction

```
s = allo.customize(gemm)
buf_A = s.buffer_at(s.A, "j")
print(s.module)
```

✓ Progressive rewrite

Each step inspectable and verifiable



✗ Monolithic lowering

e.g., TVM TE, HeteroCL



Hard to debug, no correctness guarantee

→ Real-time MLIR Module Inspection



```
module {
  func.func @gemm(%arg0: memref<1024x1024xi8>,
                 %arg1: memref<1024x1024xi8>,
                 %arg2: memref<1024x1024xi16>) {
    affine.for %arg3 = 0 to 1024 {
      affine.for %arg4 = 0 to 1024 {
        %alloc = memref.alloc() : memref<1xi8>
        affine.store %c0_i8, %alloc[%c0]
        affine.for %arg5 = 0 to 1024 {
          %1 = affine.load %alloc[%c0]
          %2 = affine.load %arg1[%arg5, %arg4]
          %3 = arith.extsi %1 : i8 to i16
          %4 = arith.extsi %2 : i8 to i16
          %5 = arith.muli %3, %4 : i16
          // ... store back C
        } {loop_name = "k", op_name = "Sk"}
      } {loop_name = "j"}
    } {loop_name = "i", op_name = "PE"}
    return
  }
}
```

Transforming GEMM to Systolic Array

→ Algorithm specification

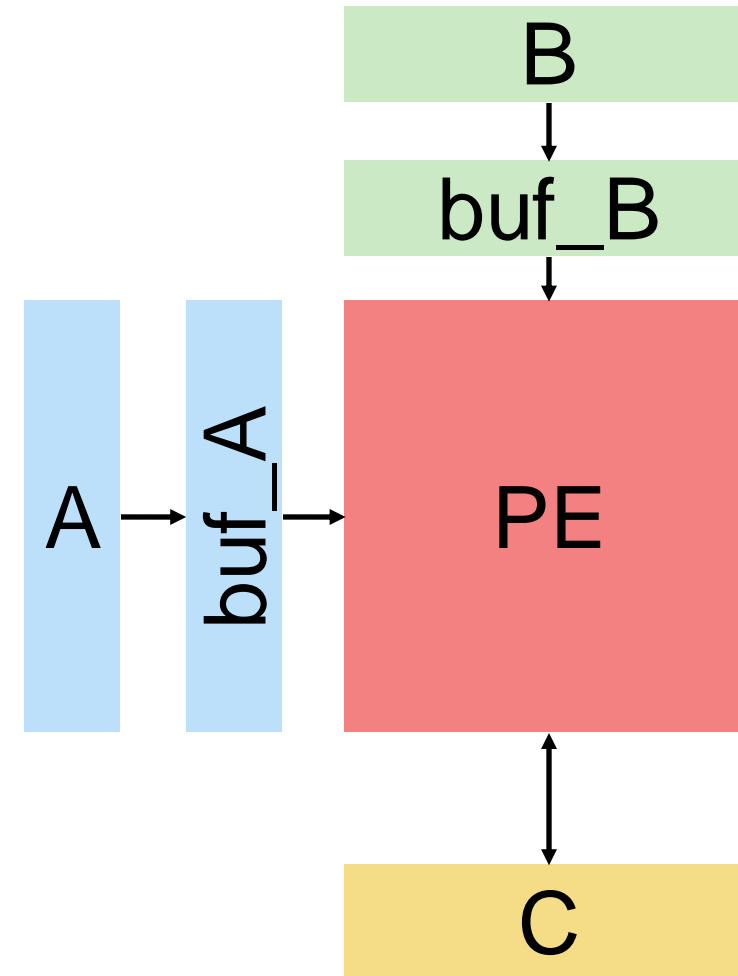
```
def gemm(A: int8[M, K],  
        B: int8[K, N],  
        C: int16[M, N]):  
    for i, j in allo.grid(M, N, "PE"):  
        for k in range(K):  
            C[i, j] += A[i, k] * B[k, j]
```



→ Schedule construction

```
s = allo.customize(gemm)  
buf_A = s.buffer_at(s.A, "j")  
buf_B = s.buffer_at(s.B, "j")
```

→ Architectural Diagram



Transforming GEMM to Systolic Array

→ Algorithm specification

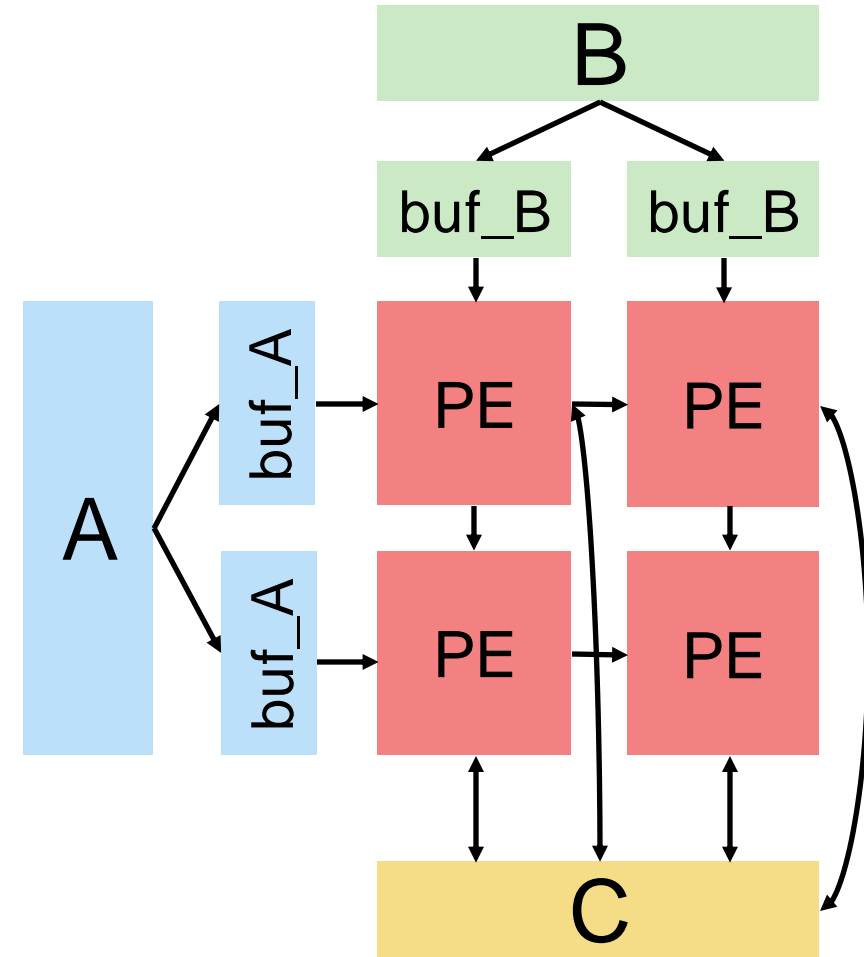
```
def gemm(A: int8[M, K],  
        B: int8[K, N],  
        C: int16[M, N]):  
    for i, j in allo.grid(M, N, "PE"):  
        for k in range(K):  
            C[i, j] += A[i, k] * B[k, j]
```



→ Schedule construction

```
s = allo.customize(gemm)  
buf_A = s.buffer_at(s.A, "j")  
buf_B = s.buffer_at(s.B, "j")  
pe = s.unfold("PE", axis=[0, 1])
```

→ Architectural Diagram



Transforming GEMM to Systolic Array

→ Algorithm specification

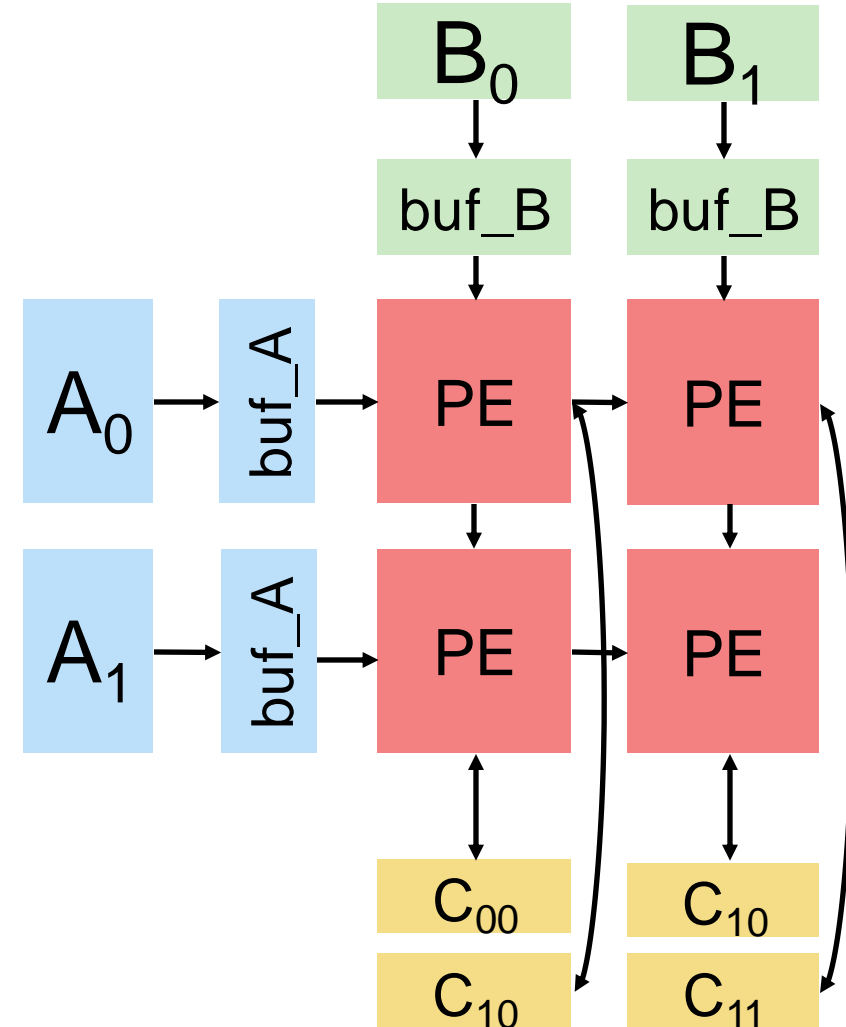
```
def gemm(A: int8[M, K],  
        B: int8[K, N],  
        C: int16[M, N]):  
    for i, j in allo.grid(M, N, "PE"):  
        for k in range(K):  
            C[i, j] += A[i, k] * B[k, j]
```



→ Schedule construction

```
s = allo.customize(gemm)  
buf_A = s.buffer_at(s.A, "j")  
buf_B = s.buffer_at(s.B, "j")  
pe = s.unfold("PE", axis=[0, 1])  
s.partition(s.A, dim=0)  
s.partition(s.B, dim=1)  
s.partition(s.C, dim=[0, 1])
```

→ Architectural Diagram



Transforming GEMM to Systolic Array

→ Algorithm specification

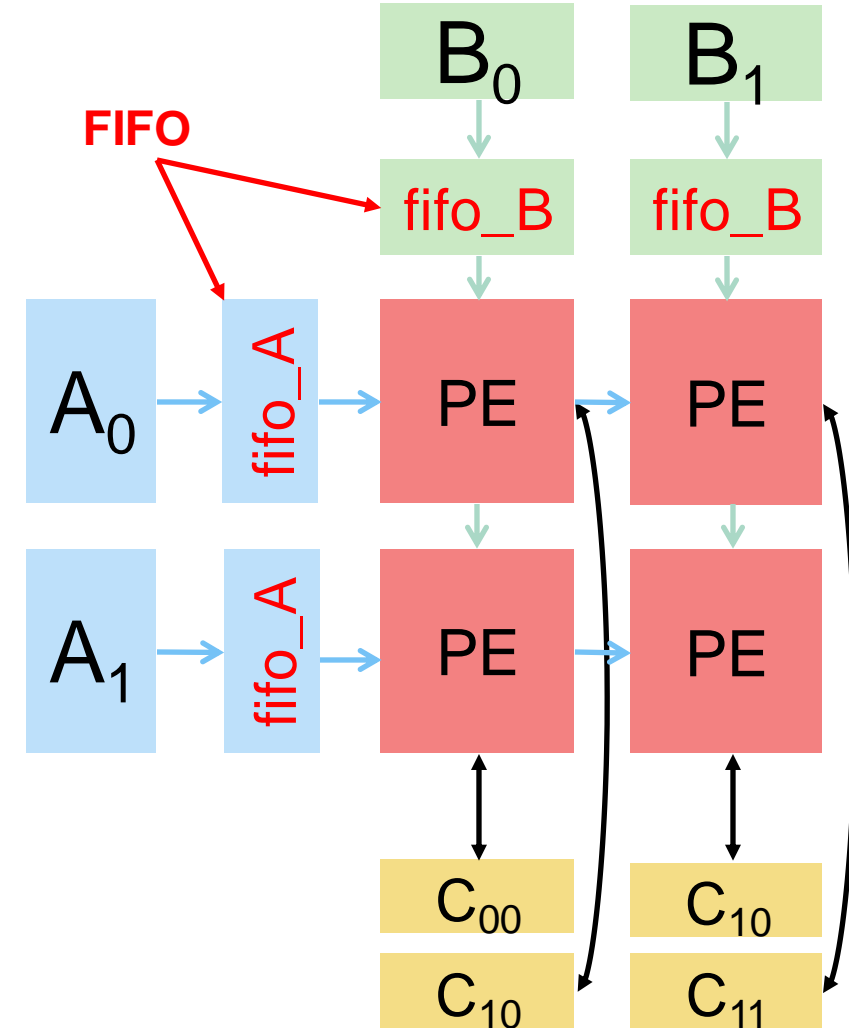
```
def gemm(A: int8[M, K],  
        B: int8[K, N],  
        C: int16[M, N]):  
    for i, j in allo.grid(M, N, "PE"):  
        for k in range(K):  
            C[i, j] += A[i, k] * B[k, j]
```



→ Schedule construction

```
s = allo.customize(gemm)  
buf_A = s.buffer_at(s.A, "j")  
buf_B = s.buffer_at(s.B, "j")  
pe = s.unfold("PE", axis=[0, 1])  
s.partition(s.A, dim=0)  
s.partition(s.B, dim=1)  
s.partition(s.C, dim=[0, 1])  
s.relay(buf_A, pe, axis=1, depth=M + 1)  
s.relay(buf_B, pe, axis=0, depth=N + 1)
```

→ Architectural Diagram



Transforming GEMM to Systolic Array

→ Algorithm specification

```
def gemm(A: int8[M, K],  
        B: int8[K, N],  
        C: int16[M, N]):  
    for i, j in allo.grid(M, N, "PE"):  
        for k in range(K):  
            C[i, j] += A[i, k] * B[k, j]
```

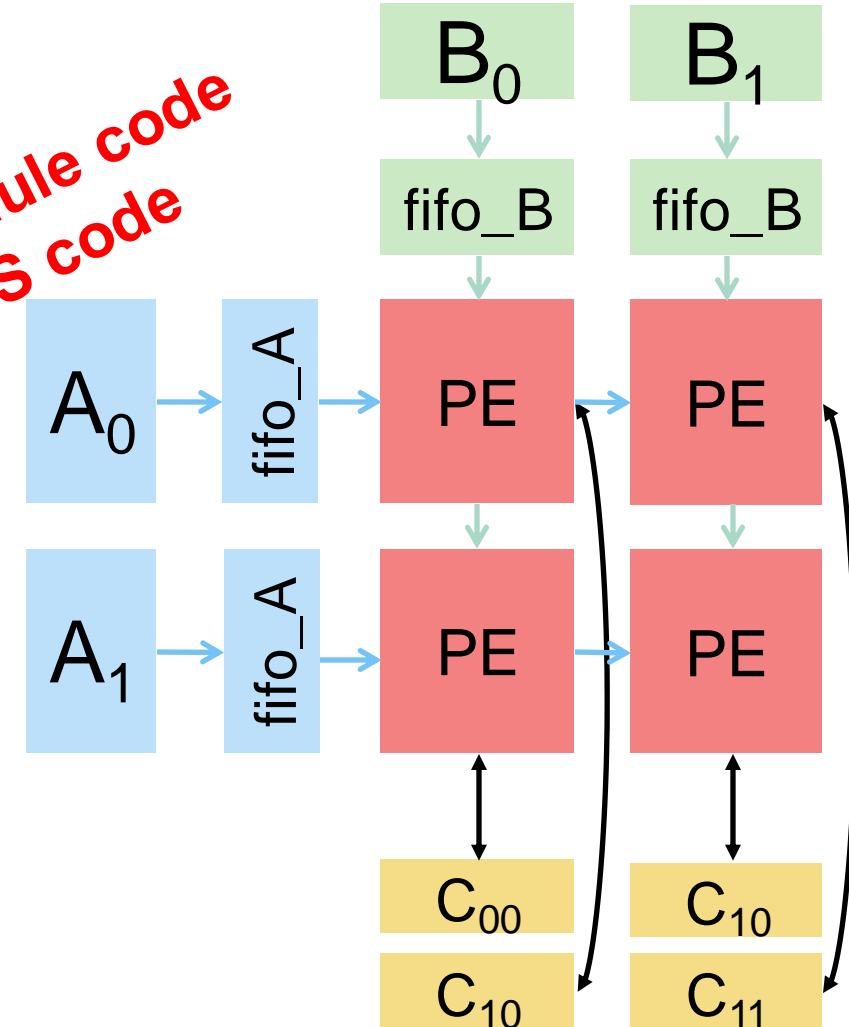


Only 8 lines of schedule code
vs ~500 lines HLS code

→ Schedule construction

```
s = allo.customize(gemm)  
buf_A = s.buffer_at(s.A, "j")  
buf_B = s.buffer_at(s.B, "j")  
pe = s.unfold("PE", axis=[0, 1])  
s.partition(s.A, dim=0)  
s.partition(s.B, dim=1)  
s.partition(s.C, dim=[0, 1])  
s.relay(buf_A, pe, axis=1, depth=M + 1)  
s.relay(buf_B, pe, axis=0, depth=N + 1)
```

→ Architectural Diagram



Transforming GEMM to Systolic Array

→ Algorithm specification

```
def gemm(A: int8[M, K],
        B: int8[K, N],
        C: int16[M, N]):
    for i, j in allo.grid(M, N, "PE"):
        for k in range(K):
            C[i, j] += A[i, k] * B[k, j]
```



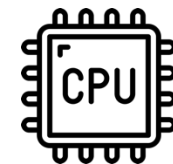
→ Schedule construction

```
s = allo.customize(gemm)
buf_A = s.buffer_at(s.A, "j")
buf_B = s.buffer_at(s.B, "j")
pe = s.unfold("PE", axis=[0, 1])
s.partition(s.A, dim=0)
s.partition(s.B, dim=1)
s.partition(s.C, dim=[0, 1])
s.relay(buf_A, pe, axis=1, depth=M + 1)
s.relay(buf_B, pe, axis=0, depth=N + 1)
```

→ CPU Simulation

```
mod = s.build(target="llvm")

A = np.random.randint(-8, 8, size=(M, K)) \
    .astype(np.int8)
B = np.random.randint(-8, 8, size=(K, N)) \
    .astype(np.int8)
C = np.zeros((M, N), dtype=np.int16)
mod(A, B, C)
np.testing.assert_allclose(C, A @ B, atol=1e-3)
```



Transforming GEMM to Systolic Array

→ Algorithm specification

```
def gemm(A: int8[M, K],  
        B: int8[K, N],  
        C: int16[M, N]):  
    for i, j in allo.grid(M, N, "PE"):  
        for k in range(K):  
            C[i, j] += A[i, k] * B[k, j]
```

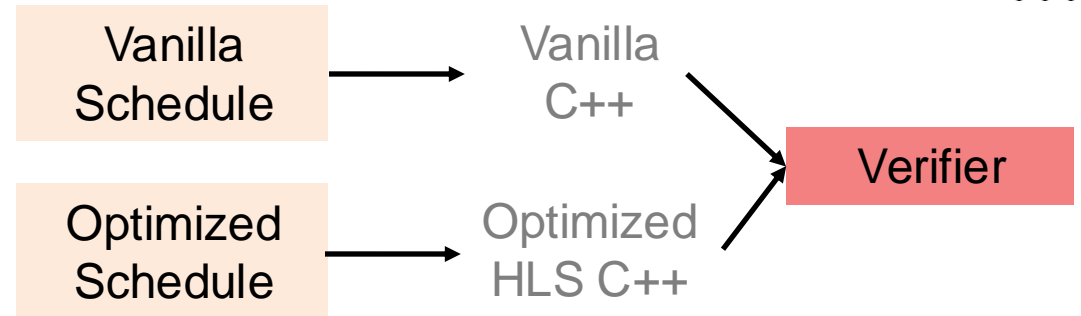
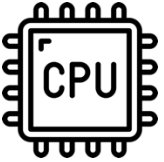


→ Schedule construction

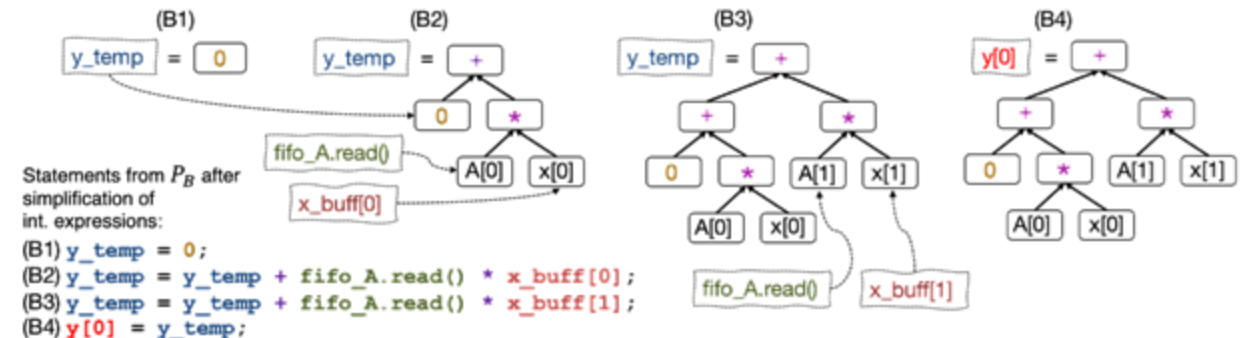
```
s = allo.customize(gemm)  
buf_A = s.buffer_at(s.A, "j")  
buf_B = s.buffer_at(s.B, "j")  
pe = s.unfold("PE", axis=[0, 1])  
s.partition(s.A, dim=0)  
s.partition(s.B, dim=1)  
s.partition(s.C, dim=[0, 1])  
s.relay(buf_A, pe, axis=1, depth=M + 1)  
s.relay(buf_B, pe, axis=0, depth=N + 1)
```

→ CPU Verification [FPGA'24 Best Paper]

```
mod = s.verify()
```



- Formally verify the equivalence of two C++ programs
- Support statically interpretable control-flow (SICF)
- Can be invoked for each transformation step
- Minute-scale verification!



Transforming GEMM to Systolic Array

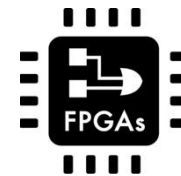
→ Algorithm specification

```
def gemm(A: int8[M, K],  
        B: int8[K, N],  
        C: int16[M, N]):  
    for i, j in allo.grid(M, N, "PE"):  
        for k in range(K):  
            C[i, j] += A[i, k] * B[k, j]
```



→ Bitstream generation

```
mod = s.build(target="vitis_hls",  
              mode="hw",  
              project="systolic.prj")  
  
# Automatically invoke the Vitis toolchain  
mod(A, B, C)
```



First FFN layer in BERT-base (512, 768)x(768, 3072) w/ 16x16 SA

	Latency (ms)	BRAM	DSP	FF	LUT
Allo	15.73	0 (0%)	128 (1%)	79969 (3%)	244439 (18%)
AutoSA	15.71	514 (12%)	256 (2%)	100138 (3%)	244032 (18%)

Same level of performance but much lower resource usage

Transforming GEMM to Systolic Array

→ Algorithm specification

```
def gemm(A: int8[M, K],  
        B: int8[K, N],  
        C: int16[M, N]):  
    for i, j in allo.grid(M, N, "PE"):  
        for k in range(K):  
            C[i, j] += A[i, k] * B[k, j]
```



→ Bitstream generation

```
mod = s.build(target="vitis_hls",  
              mode="hw",  
              project="systolic.prj")
```

```
# Automatically invoke the Vitis toolchain  
mod(A, B, C)
```



→ Parameterized template

```
def gemm[TyA, TyB, TyC, Mt: index, Nt: index, Kt: index] \  
    (A: TyA[Mt, Kt], B: TyB[Kt, Nt], C: TyC[Mt, Nt])
```

```
def tiled_gemm[TyA, TyB, TyC, M: index, N: index, K: index, S0: index, S1: index] \  
    (A: TyA[M, K], B: TyB[K, N], C: TyC[M, N]):  
    local_A: TyA[S0, K]  
    local_B: TyB[K, S1]  
    local_C: TyC[S0, S1]  
    for mi, ni in allo.grid(M // S0, N // S1, name="outer_tile"):  
        # ... load_A_tile, load_B_tile  
        gemm[TyA, TyB, TyC, S0, S1, K](local_A, local_B, local_C)  
        # ... store_C_tile
```

Easy to adjust SA size

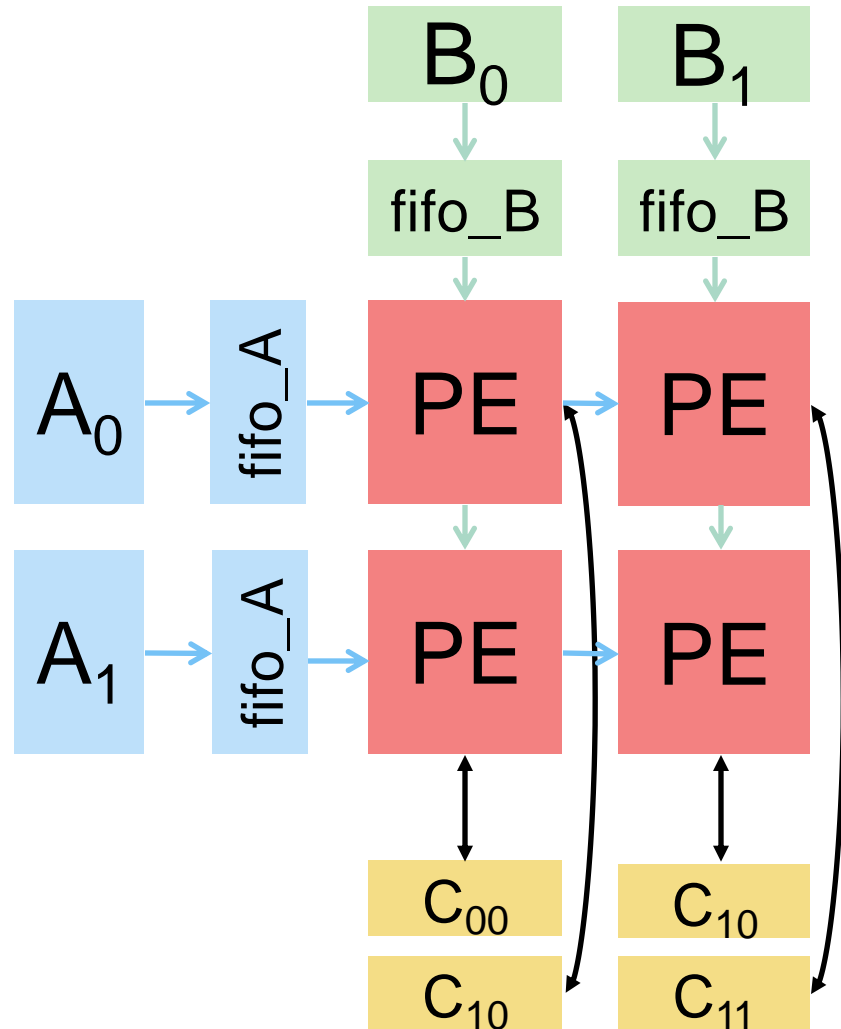


Highly customizable parameters



Composable Schedules

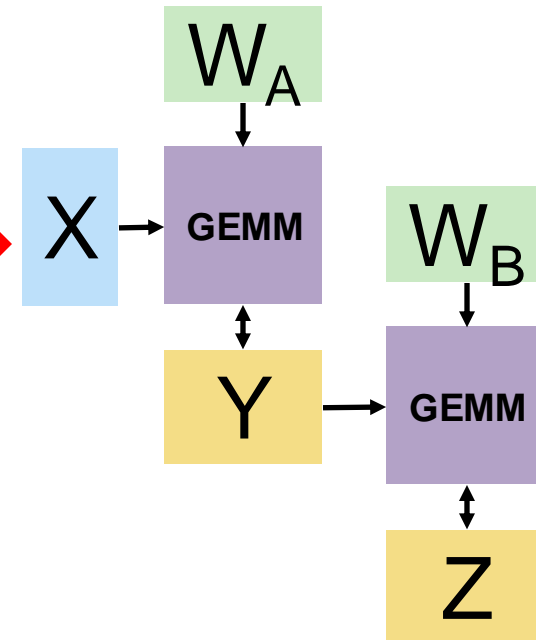
→ Given optimized kernel implementation



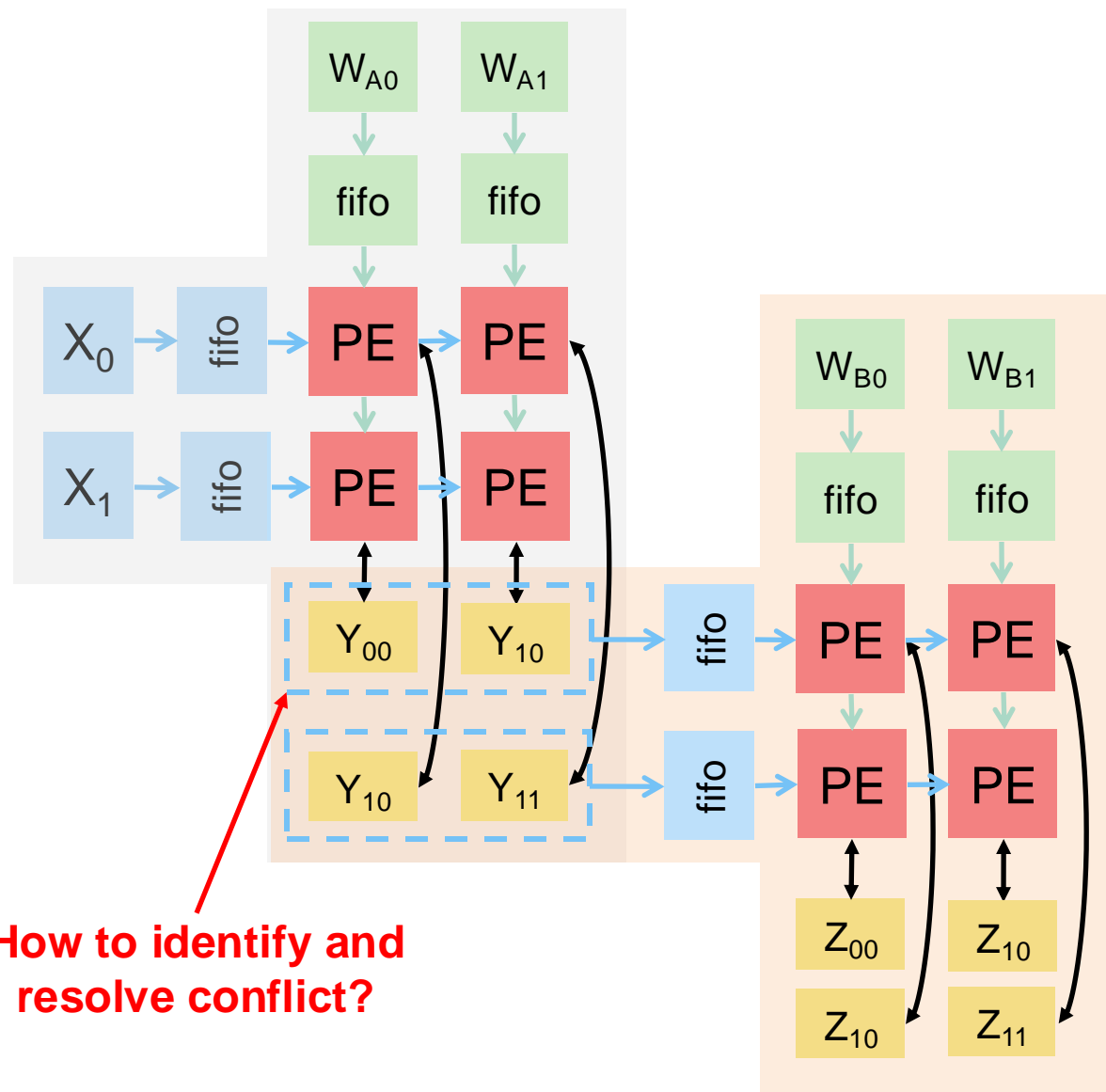
→ Algorithm specification (Hierarchical)

```
def top(X: int8[M, K], W_A: int8[K, N],  
        W_B: int8[N, K], Y: int8[M, K]):  
    Y: int8[M, N] = 0  
    Z: int8[M, K] = 0  
    gemm(X, W_A, Y)  
    gemm(Y, W_B, Z)  
    return Z
```

How to leverage?



Composable Schedules



→ Algorithm specification (Hierarchical)

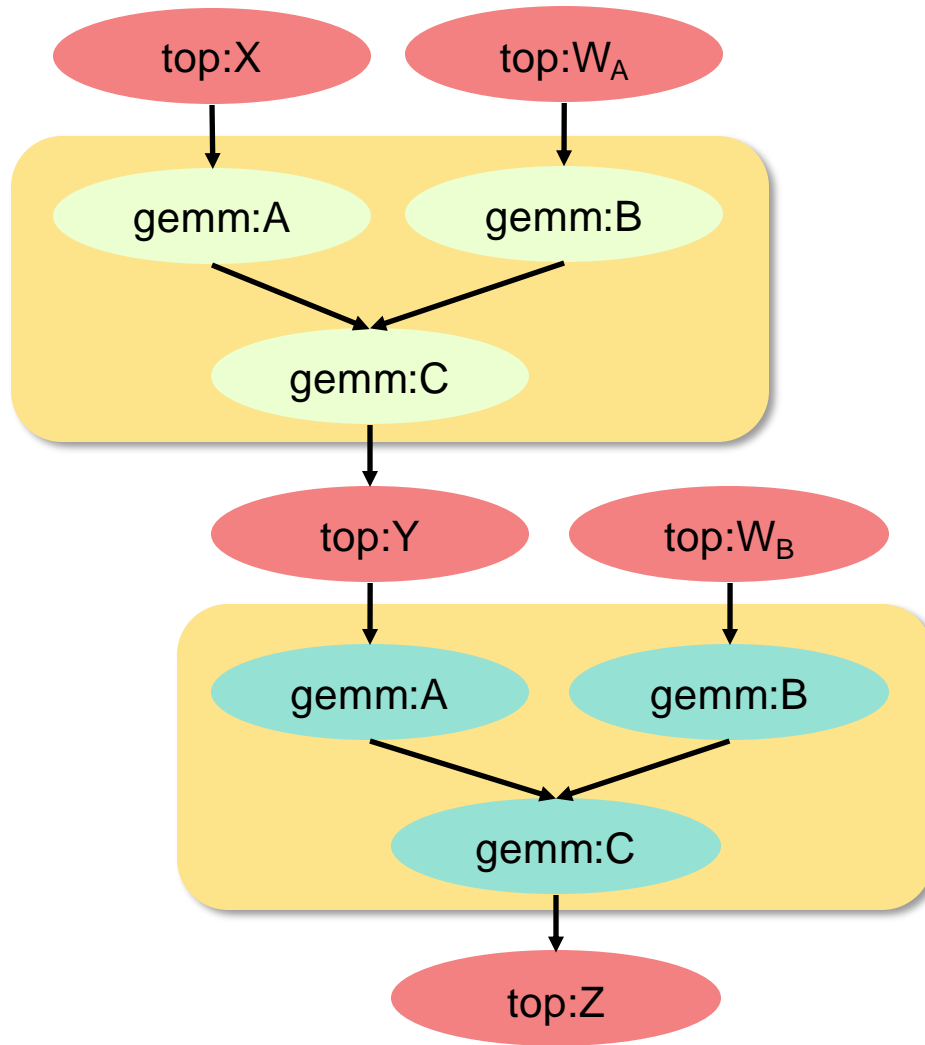
```
def top(X: int8[M, K], W_A: int8[K, N],
        W_B: int8[N, K], Y: int8[M, K]):
    Y: int8[M, N] = 0
    Z: int8[M, K] = 0
    gemm(X, W_A, Y)
    gemm(Y, W_B, Z)
    return Z
```

→ Schedule composition

```
# Previous customizations for GEMM
s_gemm = allo.customize(gemm)
# ...

s_top = allo.customize(top)
s_top.compose(s_gemm)
```

Composable Schedules



Hierarchical dataflow graph

→ Algorithm specification (Hierarchical)

```
def top(X: int8[M, K], W_A: int8[K, N],  
        W_B: int8[N, K], Y: int8[M, K]):  
    Y: int8[M, N] = 0  
    Z: int8[M, K] = 0  
    gemm(X, W_A, Y)  
    gemm(Y, W_B, Z)  
    return Z
```

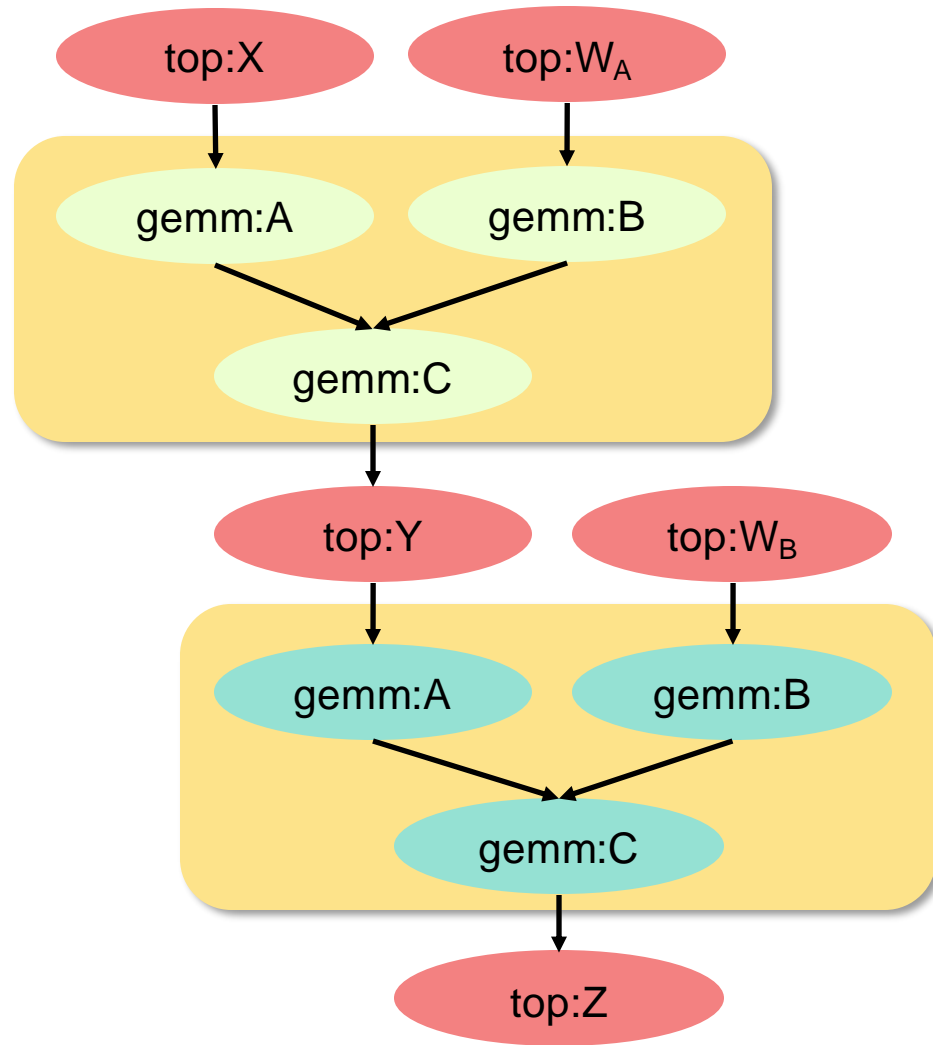
← Caller definition

→ Callee definition

```
def gemm(A: int8[M, K],  
        B: int8[K, N],  
        C: int16[M, N])
```

Goal: Model function arguments in both **callers** and **callees** and make sure the layouts are consistent

Composable Schedules



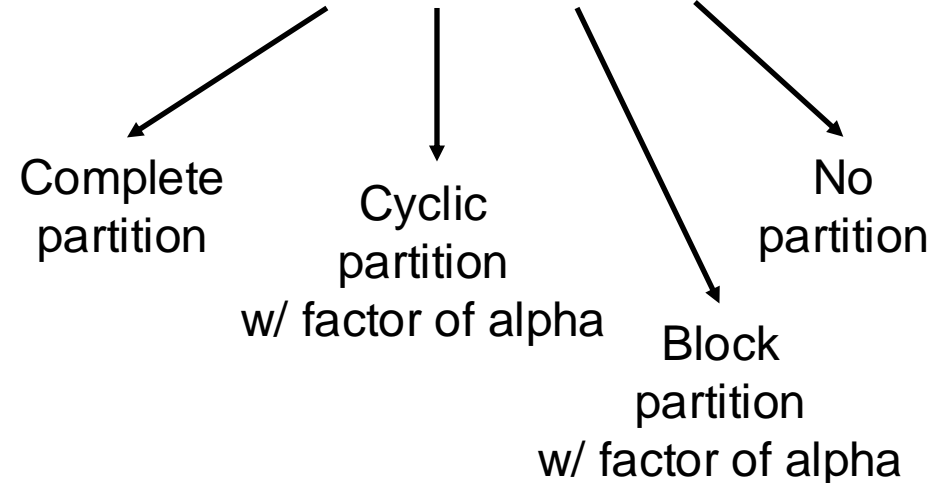
Hierarchical dataflow graph

- ▶ **Goal:** Ensure the layouts of function arguments are consistent
- ▶ **Key idea: Model data layout as a type**

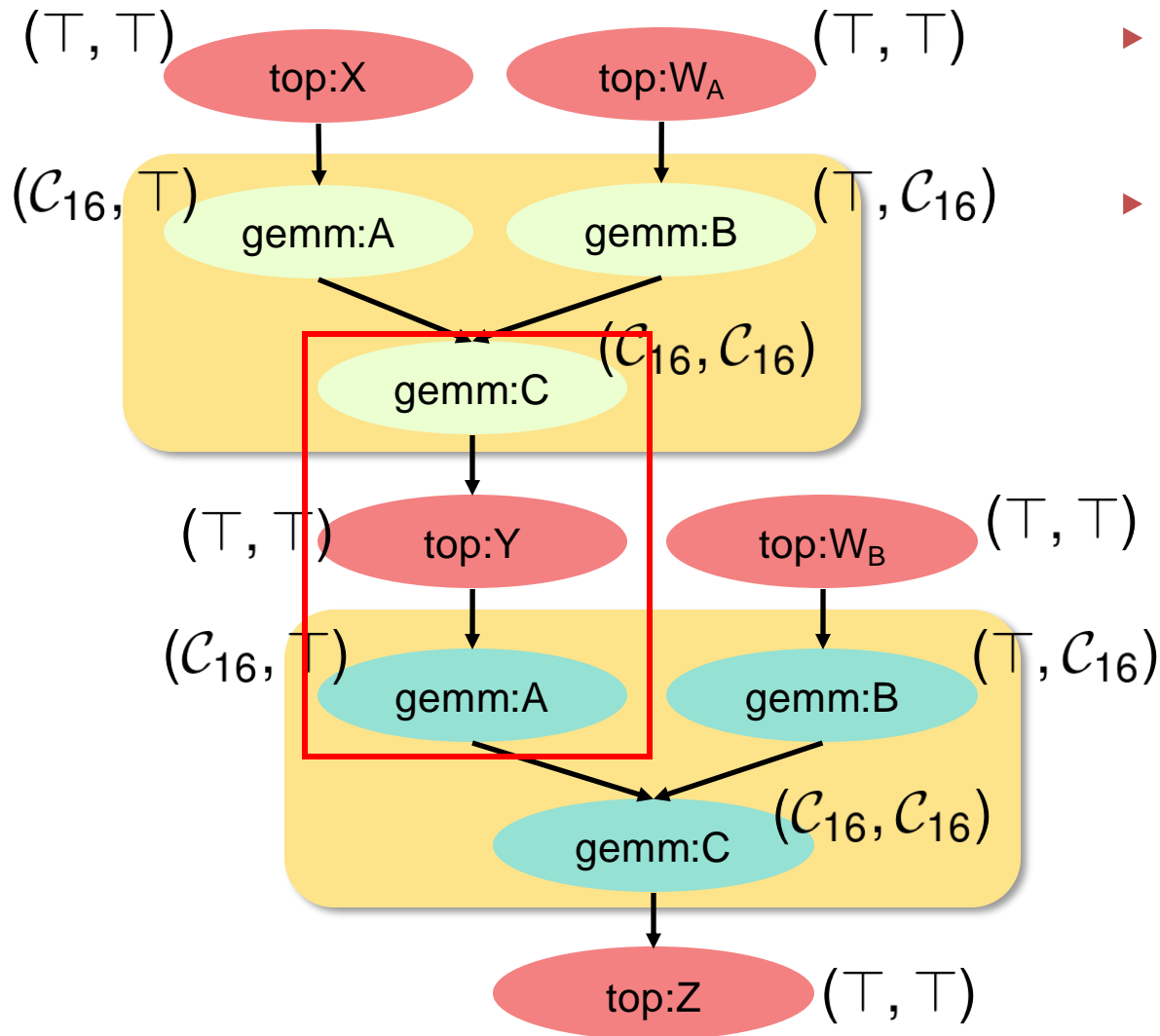
- N-D array layout (composite type):
 $\tau := (\hat{\tau}_1, \dots, \hat{\tau}_N)$
- Base type for each dimension:

$$\alpha := \mathbb{N}$$

$$\hat{\tau} := \perp \mid C_\alpha \mid \mathcal{B}_\alpha \mid \top$$



Composable Schedules



Hierarchical dataflow graph

- ▶ **Goal:** Ensure the layouts of function arguments are consistent
- ▶ **Key idea: Model data layout as a type**

- N-D array layout (composite type):
 $\tau := (\hat{\tau}_1, \dots, \hat{\tau}_N)$
- Base type for each dimension:

$$\alpha := \mathbb{N}$$

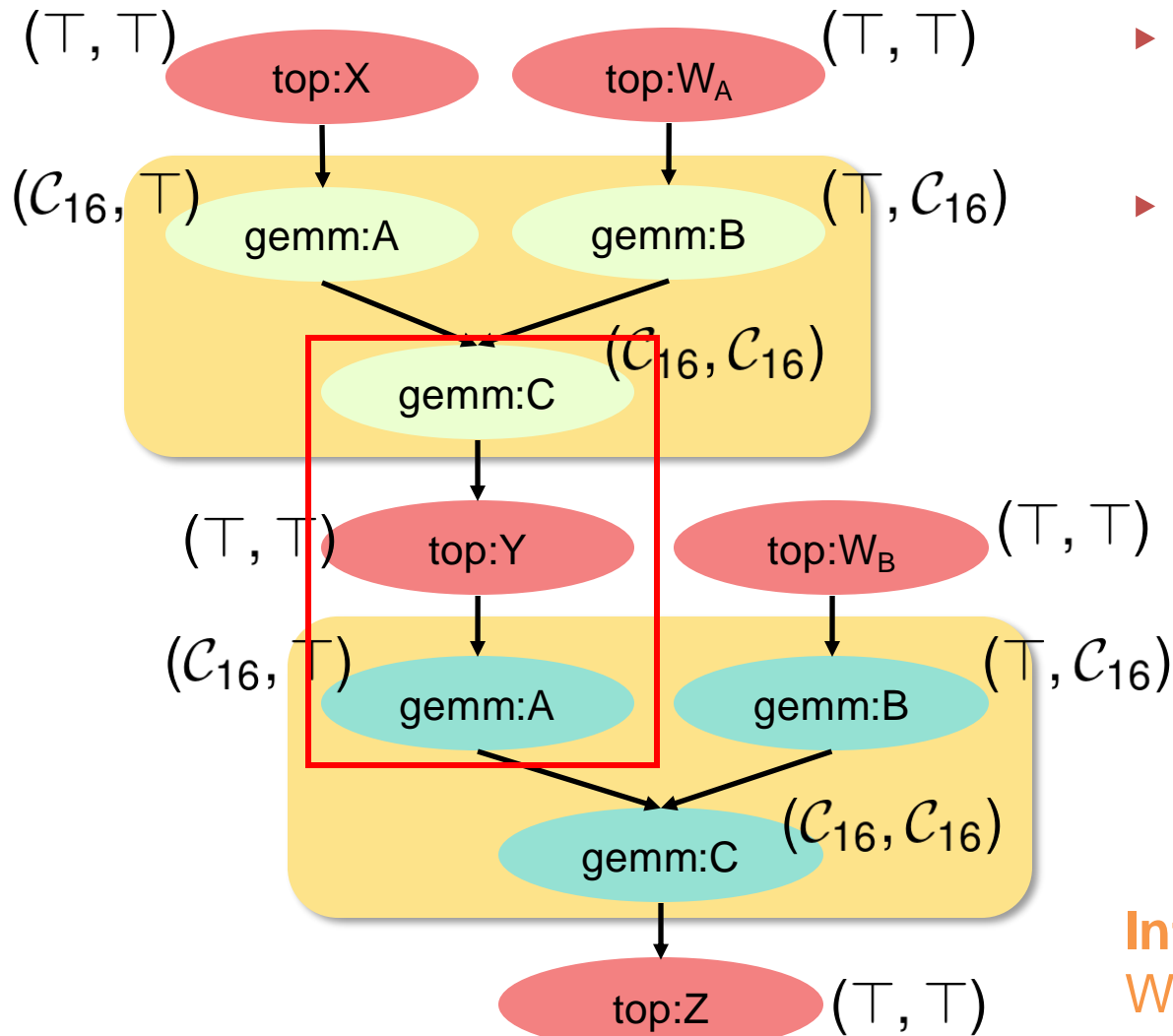
$$\hat{\tau} := \perp \mid C_\alpha \mid \mathcal{B}_\alpha \mid \top$$

Complete
partition

Cyclic
partition
w/ factor of alpha

No
partition
Block
partition
w/ factor of alpha

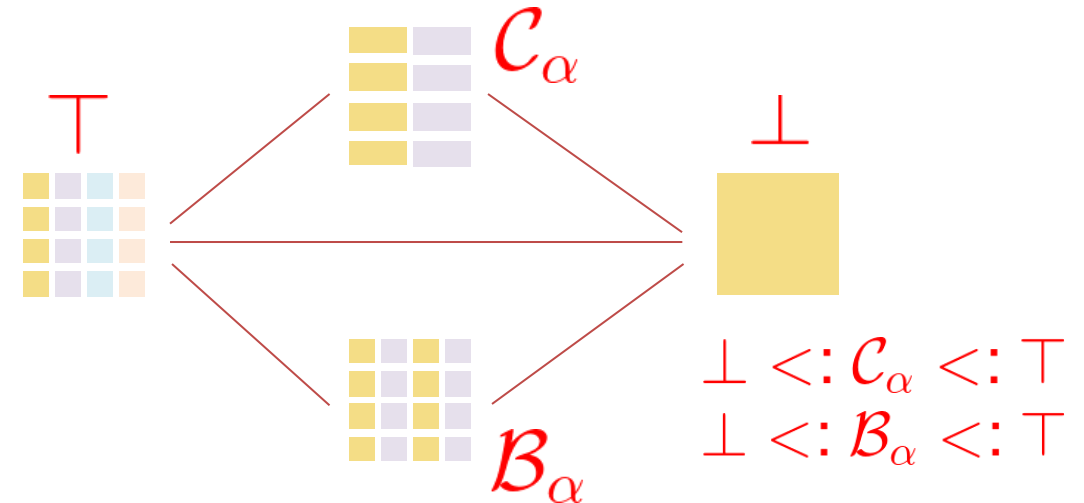
Composable Schedules



Hierarchical dataflow graph

- ▶ Goal: Ensure the layouts of function arguments are consistent
- ▶ Key idea: Model data layout as a type

Subtyping relation forms a **lattice**!



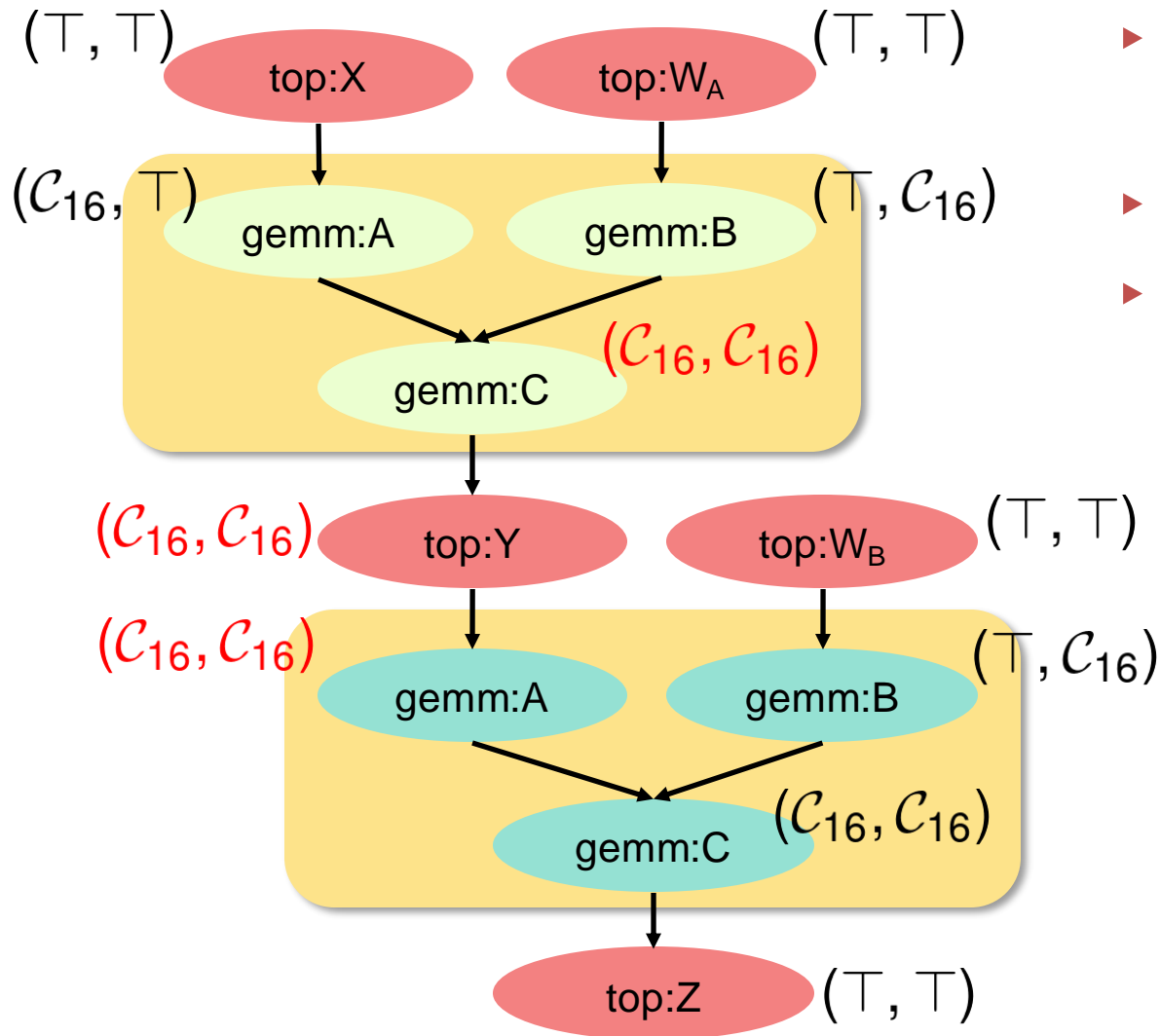
Intuition:

We can supply more read/write parallelism, but not less!

Subtyping relation: $X <: Y$

The code expecting a memory with partition type Y is also compatible with a memory with partition type X

Composable Schedules



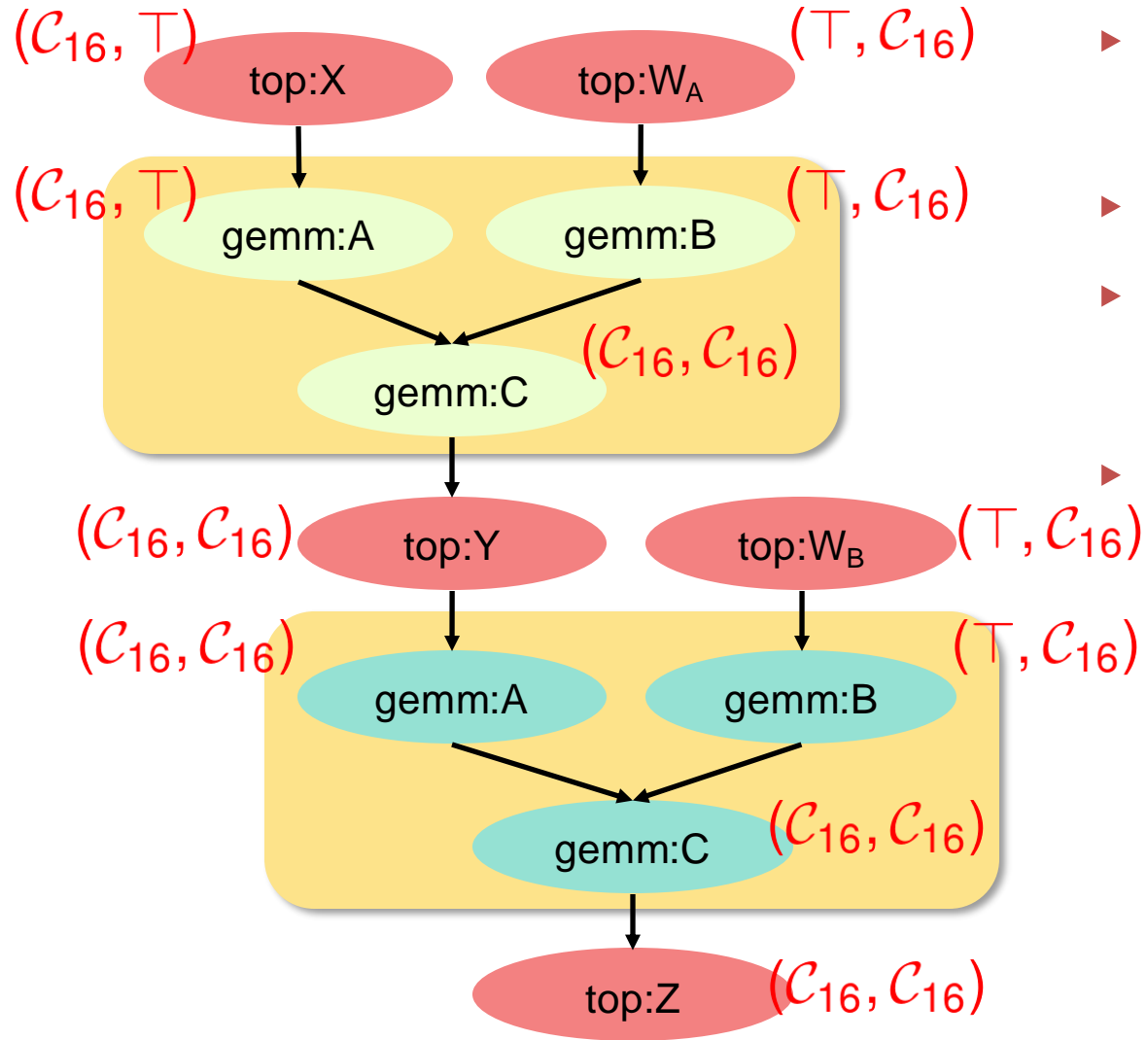
Hierarchical dataflow graph

- ▶ Goal: Ensure the layouts of function arguments are consistent
- ▶ **Key idea: Model data layout as a type**
- ▶ Data layout propagation \rightarrow type inference

$$\top \sqcap \mathcal{C}_{16} = \mathcal{C}_{16}$$

Create the “least common” number of memory banks

Composable Schedules



Hierarchical dataflow graph

- ▶ Goal: Ensure the layouts of function arguments are consistent
- ▶ **Key idea: Model data layout as a type**
- ▶ Data layout propagation → type inference

▶ Worklist algorithm for type inference

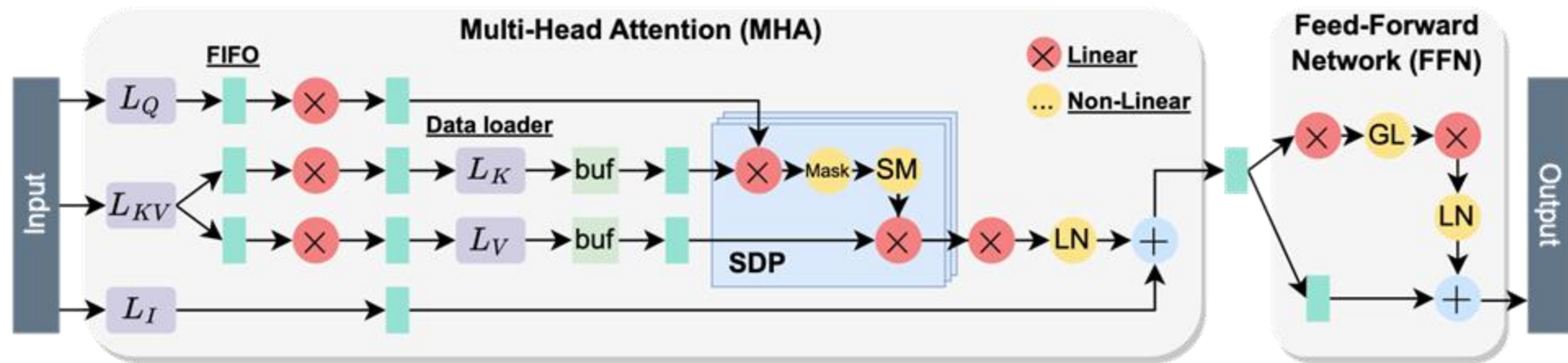
Guaranteed to terminate in **linear time** by the *Fixed-Point Theorem* if

- (1) the structure is a finite lattice and,
- (2) the transfer function is monotonic

Proof in the supplementary material!

A Complete LLM Accelerator

- ▶ GPT2 model (the only open-source LLM in the GPT family)
 - 355M parameters, 24 hidden layers, 16 heads
 - W4A8 quantization



Is it the end?

No! We need to determine how many resources are allocated to each operator!

Compose all the schedules together

```
s = allo.customize(GPT_layer)
s.compose(s_qkv)
s.compose(s_mha)
s.compose(s_ds0)
...
```


Analytical Model for LLMs [FCCM/TRETS'24]

Linear Layer	Abbreviations	Input Matrices	Prefill	Decode
Q/K/V linear	q, k, v	XW_Q, XW_K, XW_V	$3ld^2$	$3d^2$
Matmul ₁	a_1	QK^T	l^2d	$(l+1)d$
Matmul ₂	a_2	$X_{sm}V$	l^2d	$(l+1)d$
Projection	p	$X_{sdp}W_{Proj}$	ld^2	d^2
FFN ₁	f_1	$X_{mha}W_{FFN_1}$	ldd_{FFN}	dd_{FFN}
FFN ₂	f_2	$X_{act}W_{FFN_2}$	ldd_{FFN}	dd_{FFN}

- **Compute resource:** M is compute power in MACs/cycle and C is the # of layers per FPGA

$$\sum M_i C < M_{tot}, i \in \{q, k, v, a_1, a_2, p, f_1, f_2\}$$

- **Memory capacity:** S is buffer size

$$S_{param} C < DRAM_{tot},$$

$$\sum S_i C < SRAM_{tot}, i \in \{tile, KV, FIFO\}$$

- **Memory port:** s is tensor size and b is bitwidth

$$R_i = \left\lceil \frac{s_i b_{BRAM}}{M_i / r_i \times S_{BRAM}} \right\rceil \times \frac{M_i / r_i}{k}$$

$$\sum_i C R_i + 2C(R_{a_1} + R_{a_2}) < SRAM_{tot}, i \in \{q, k, v, p, f_1, f_2\}$$

- **Memory bandwidth:** B is bandwidth

$$\sum_i C B_i < B_{tot}, i \in \{q, k, v, p, f_1, f_2\}$$

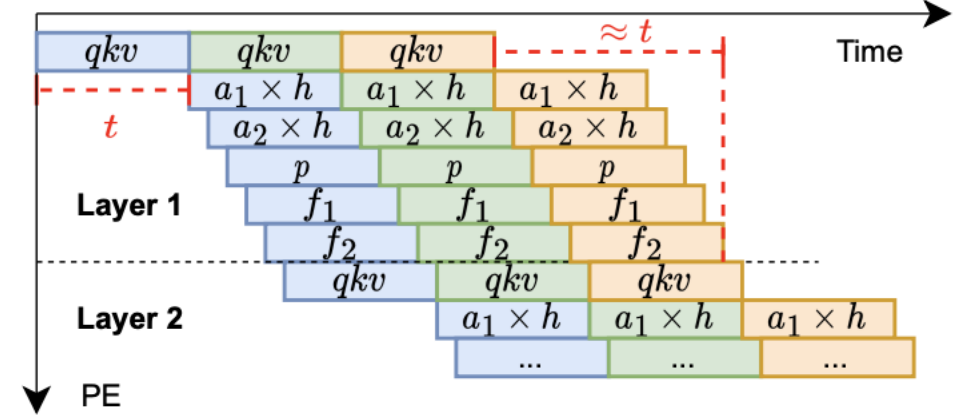


Figure : Pipeline diagram. Different colors stand for different input samples. Different blocks stand for different linear operators which also constitute the pipeline stages. h is the number of attention heads.

$$T_{prefill} = \frac{1}{freq} \frac{N}{C} \left(\frac{ld^2}{M_k} + C_{max} \left(\frac{ld^2}{M_k}, \frac{l^2d}{M_{a_1}}, \frac{ldd_{FFN}}{M_{f_1}}, T_{mem} \right) \right)$$

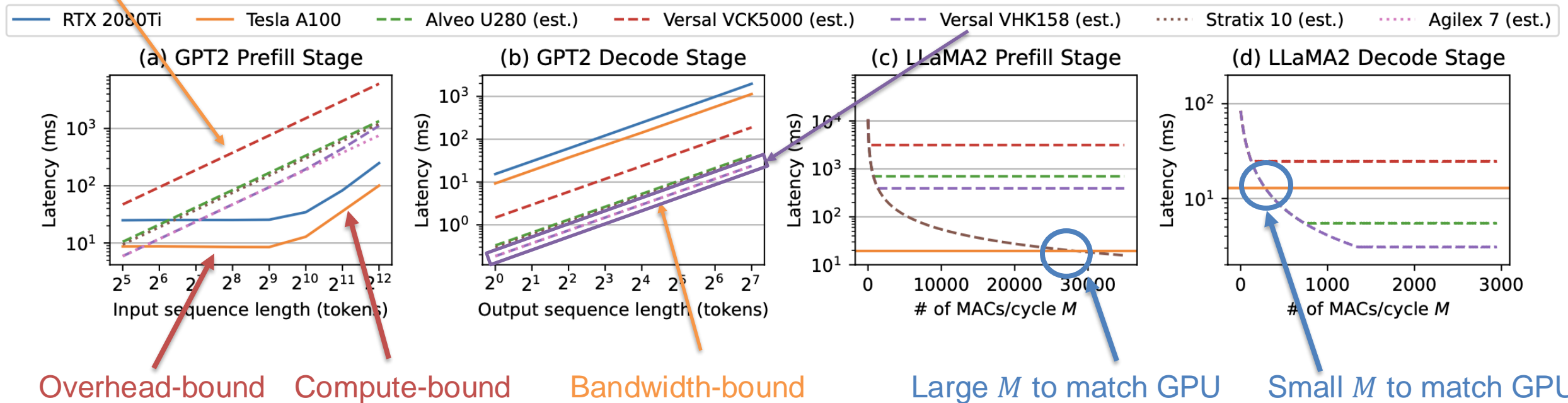
$$T_{decode} = \frac{1}{freq} \frac{N}{C} \left(\frac{d^2}{M_k} + C_{max} \left(\frac{d^2}{M_k}, \frac{(l_{max}+1)d}{M_{a_1}}, \frac{dd_{FFN}}{M_{f_1}}, T_{mem} \right) \right)$$

Estimation on Different Models and Devices

► Q: Is FPGA suitable for efficient LLM inference?

- Latency estimation of GPT2 and LLaMA2 on different FPGAs
- GPU results are obtained through actual profiling

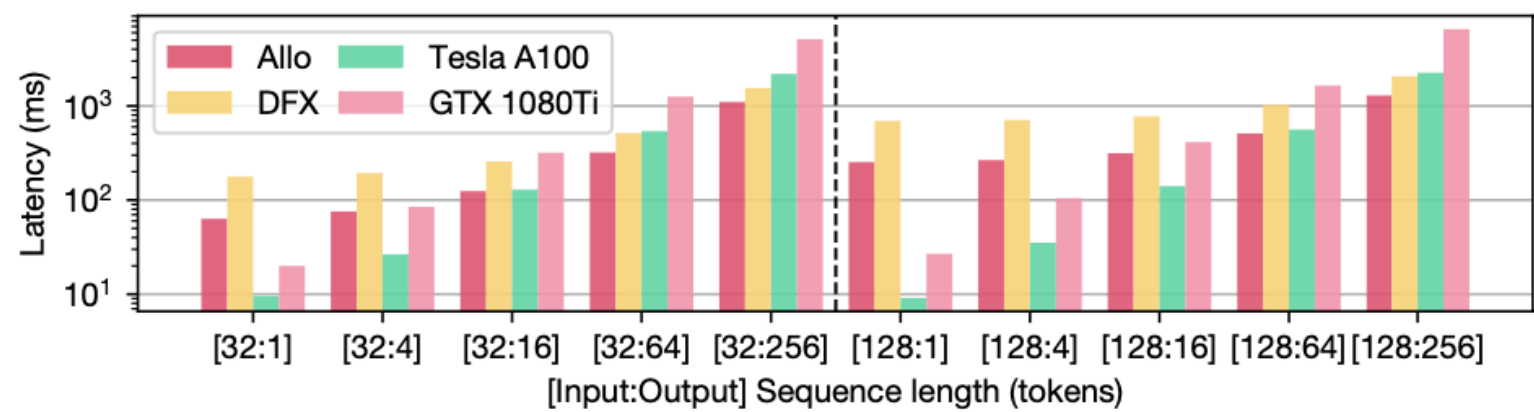
Compute-bound



Existing FPGAs are inferior in the compute-intensive prefill stage but can outperform GPUs in the memory-intensive decode stage.

LLM Accelerator Evaluation

- ▶ GPT2: single-batch, low-latency settings, adjust input/output token numbers
 - U280 FPGA (16nm), 250MHz
 - 2.2x speedup in prefill stage compared to DFX (an overlay FPGA-based xcel)
 - 1.7x speedup for long output sequences and 5.4x more energy-efficient vs A100
 - < 50 lines of schedule code



	Allo	DFX
Device	U280	U280
Freq.	250MHz	200MHz
Quant.	W4A8	fp16
BRAM	384 (19.0%)	1192 (59.1%)
DSP	1780 (19.73%)	3533 (39.2%)
FF	652K (25.0%)	1107K (42.5%)
LUT	508K (39.0%)	520K (39.9%)

* Hongzheng Chen, Jiahao Zhang, Yixiao Du, Shaojie Xiang, Zichao Yue, Niansong Zhang, Yaohui Cai, Zhiru Zhang, “Understanding the Potential of FPGA-Based Spatial Acceleration for Large Language Model Inference”, ACM Transactions on Reconfigurable Technology and Systems (TRETs), 2024.

High-Level PyTorch Frontend

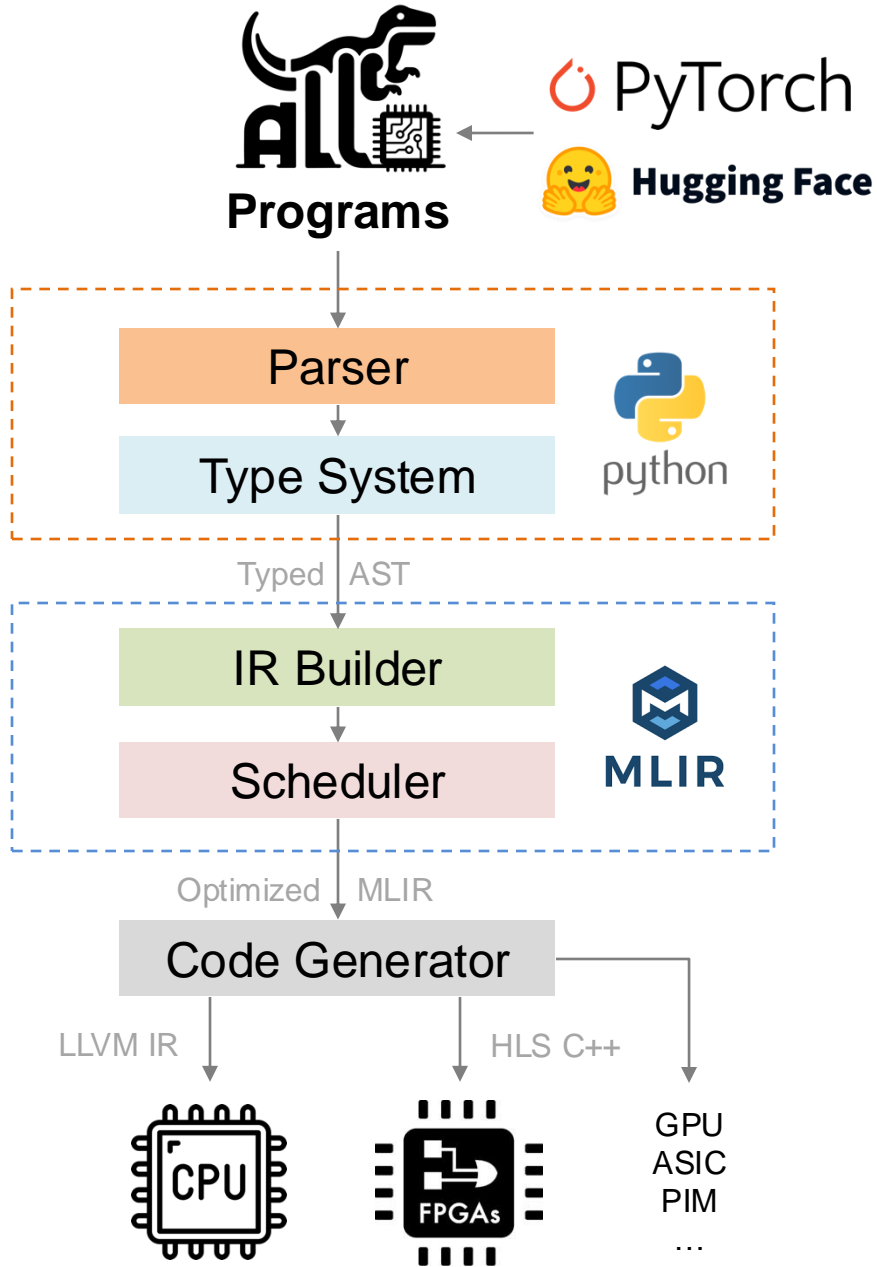
- ▶ Predefined schedules for commonly used NN operators
- ▶ Can directly import model from PyTorch and build optimized xcel design
 - Through TorchDynamo and torch.fx

```
import torch
import allo
import numpy as np
from transformers import AutoConfig
from transformers.models.gpt2.modeling_gpt2 import GPT2Model

bs, seq, hs = 1, 512, 1024

example_inputs = [torch.rand(bs, seq, hs)]
config = AutoConfig.from_pretrained("gpt2")
module = GPT2Model(config).eval()
mlir_mod = allo.frontend.from_pytorch(
    module,
    example_inputs=example_inputs,
)
```

Summary



- ▶ Features of Allo ADL
 - Pythonic
 - Decoupled customizations
 - Composability

- ▶ Single-kernel
 - High-performance
 - Verifiable
 - Reusable

- ▶ Multi-kernel
 - First time to leverage an ADL to design a large-scale LLM accelerator

- ▶ Future work
 - Autoscheduling
 - Build system



<https://github.com/cornell-zhang/allo>



Contact: hzchen@cs.cornell.edu