Large Language Model Acceleration with Allo

Hongzheng Chen

PI: Zhiru Zhang

Cornell University

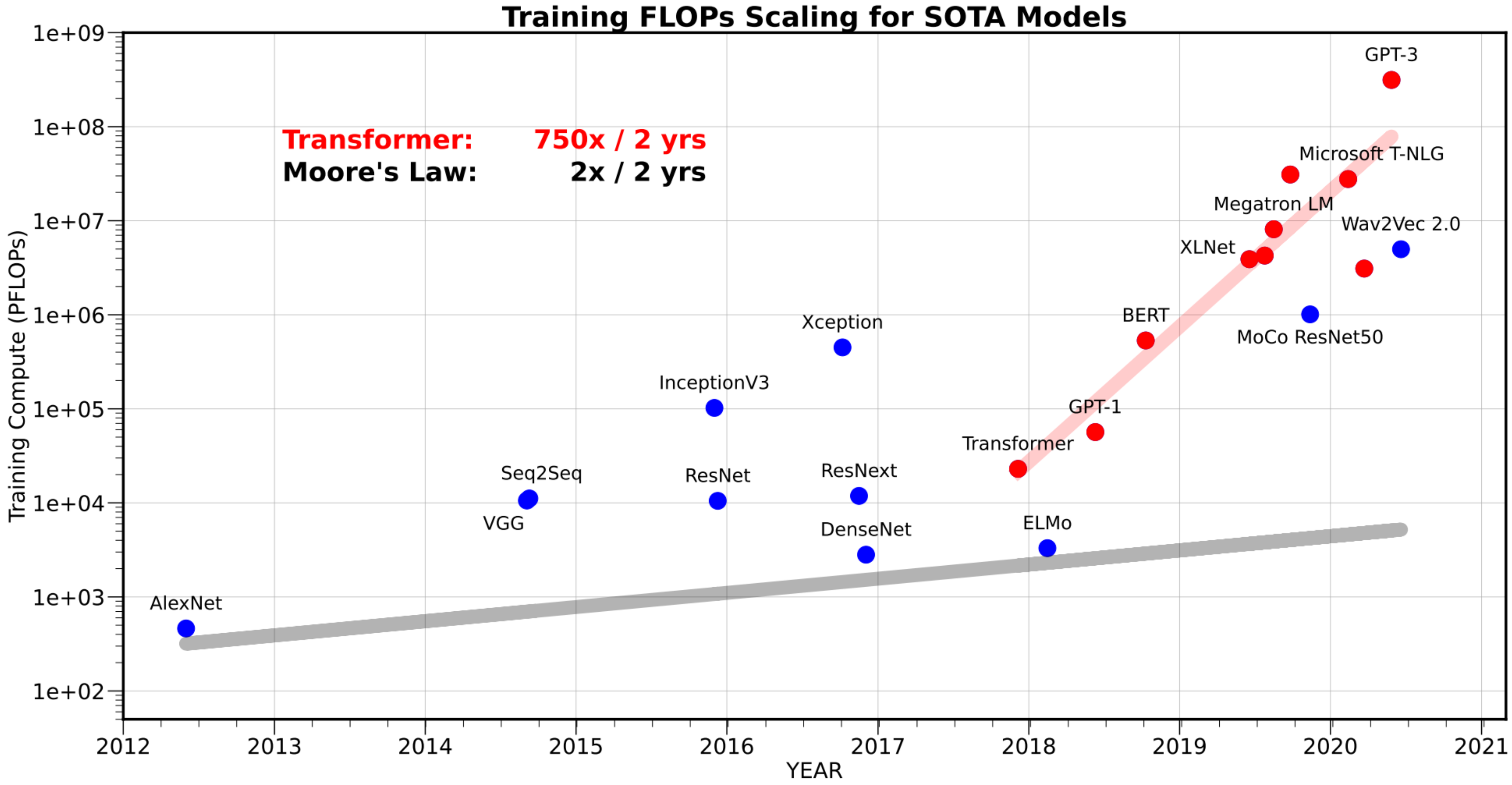
UIUC-HACC

04/10/2024

Cornell University

ZHANG

The Era of Large Language Models (LLMs)

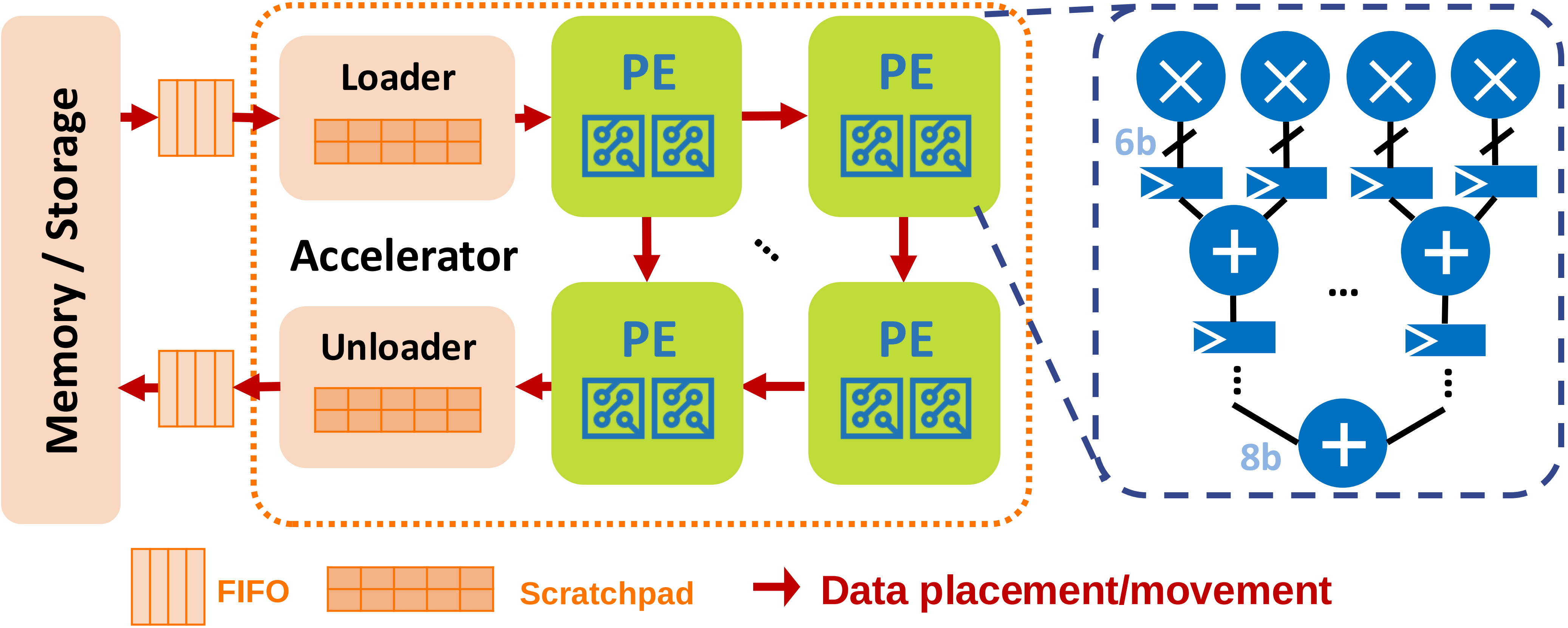


- The growth in computation demands outpaces the increase in compute power offered by current hardware
- Special-purpose hardware accelerator for large deep learning models (e.g., Google TPU, AWS Inferentia)

Fig source: Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, Kurt Keutzer, “AI and Memory Wall”, IEEE MICRO, 2024.

Specialized Accelerator Design

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



Challenge 1: Balancing Manual Control & Compiler Optimization

(a) Manual optimization

😃 Fine-grained optimizations, high performance

□ Rewrite the application code, hard to maintain

```
void conv1(...) {
 #pragma HLS array_partition variable=Filter dim=0
 hls::LineBuffer<3, N, ap_fixed<8,4> > buf;
 hls::Window<3, 3, ap_fixed<8,4> > window;
 for(int y = 0; y < N; y++) {
  for(int xo = 0; xo < N/M; xo++) {
   #pragma HLS pipeline II=1
   for(int xi = 0; xi < M; xi++) {
    int x = xo*M + xi;
    ap_fixed<8,4> acc = 0;
    ap_fixed<8,4> in = Input[y][x];
    buf.shift_up(x);
    buf.insert_top(in, x);
    window.shift_left();
    for(int r = 0; r < 2; r++)
     window.insert(buf.getval(r,x),
                 i, 2);
    window.insert(in, 2, 2);
    if (y >= 2 && x >= 2) {
     for(int r = 0; r < 3; r++) {
      for(int c = 0; c < 3; c++) {
       acc += window.getval(r,c) * Filter[r][c];
     }}
     Out[y-2][x-2] = acc;
}}}}}
```

Custom compute (Loop tiling)

Custom data type (Quantization)

Custom data placement (Reuse buffers)

(b) Compiler optimization

😃 Fully automatic, least manual effort

□ No control on memory/communication, not general, hard to debug

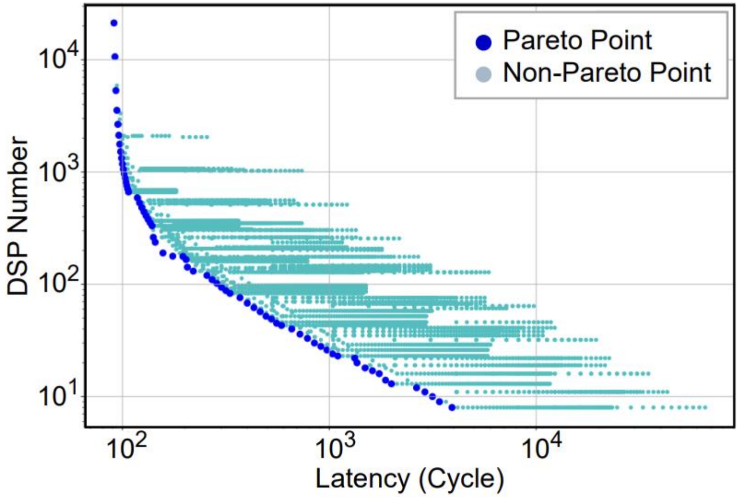
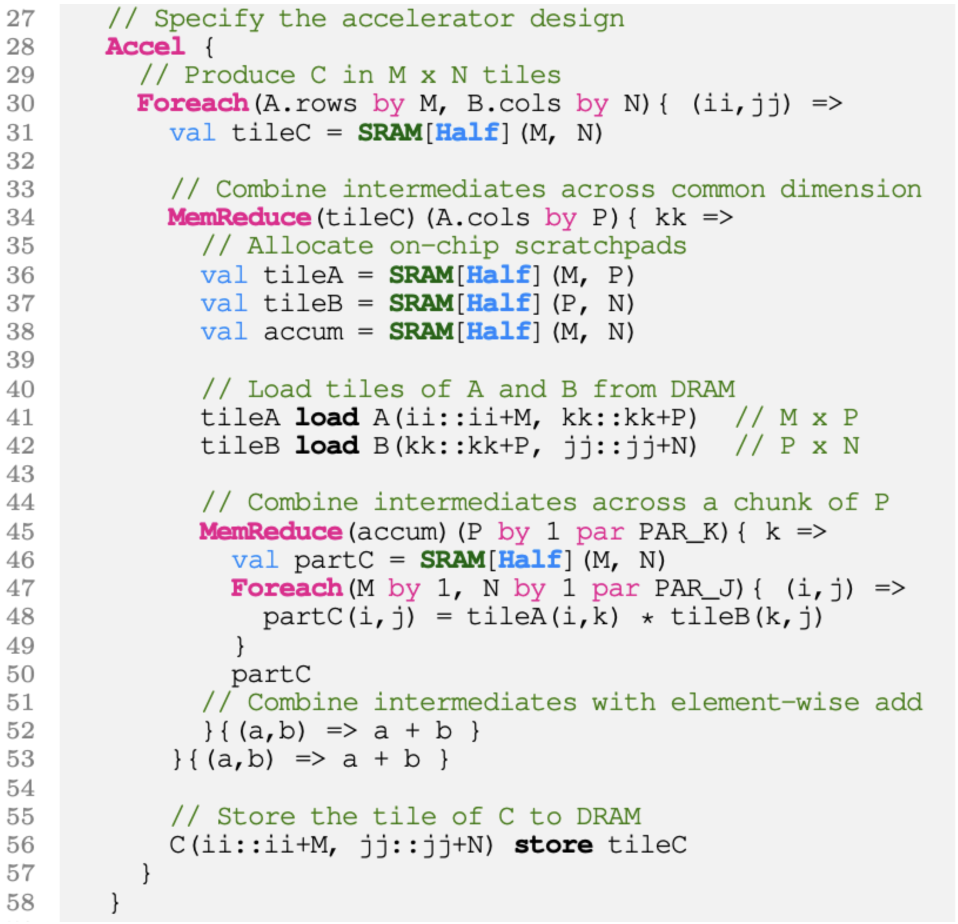


Fig source: ScaleHLS [HPCA'22]

- DSE only searches for hyperparams (e.g., tiling factors) but not program transformations (e.g., buffering)
- No transformation correctness guarantees

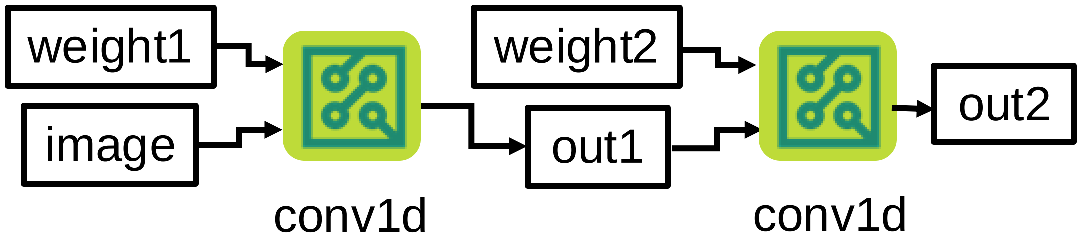
Challenge 2: Bridging the Gap from Single-Kernel to Multi-Kernel Design

- Existing accelerator design languages (ADLs) only consider opt. inside a kernel
  - e.g., HeteroCL[FPGA'19], Spatial[PLDI'18], Dahlia[PLDI'20]



```
// Specify the accelerator design
Accel {
  // Produce C in M x N tiles
  Foreach(A.rows by M, B.cols by N){ (ii,jj) =>
    val tileC = SRAM[Half](M, N)

    // Combine intermediates across common dimension
    MemReduce(tileC)(A.cols by P){ kk =>
      // Allocate on-chip scratchpads
      val tileA = SRAM[Half](M, P)
      val tileB = SRAM[Half](P, N)
      val accum = SRAM[Half](M, N)

      // Load tiles of A and B from DRAM
      tileA load A(ii::ii+M, kk::kk+P)  // M x P
      tileB load B(kk::kk+P, jj::jj+N)  // P x N

      // Combine intermediates across a chunk of P
      MemReduce(accum)(P by 1 par PAR_K){ k =>
        val partC = SRAM[Half](M, N)
        Foreach(M by 1, N by 1 par PAR_J){ (i,j) =>
          partC(i,j) = tileA(i,k) * tileB(k,j)
        }
        partC
      // Combine intermediates with element-wise add
      }{(a,b) => a + b }
    }{(a,b) => a + b }

    // Store the tile of C to DRAM
    C(ii::ii+M, jj::jj+N) store tileC
  }
}
```

(a) Code snippet in Spatial



Host-accelerator

```
void blur(DTYPE* input0, ..., DTYPE* input6,
DTYPE* output0, ..., DTYPE* output6) {
 #pragma HLS interface port=input0 bundle=g0 burst=32
 #pragma HLS interface port=input1 bundle=g1 burst=32
 stream<DTYPE> fifo_in[8], fifo_out[8];
 input_io_schedule(fifo_in, input0, ..., input6);
 #pragma HLS dataflow
 #pragma HLS stream var=fifo_inter[0] depth=32
 #pragma HLS stream var=fifo_inter[1] depth=32
 conv1(fifo_in, fifo_inter);
 conv2(fifo_inter, fifo_out);
 output_io_schedule(fifo_out, output0, ..., output6);
}
```

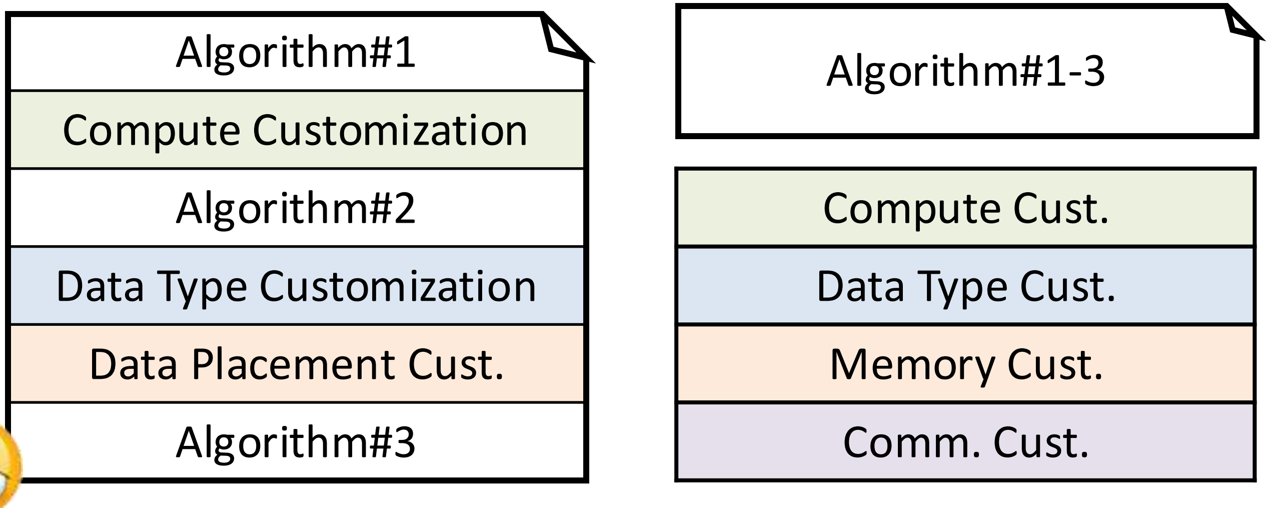
Inter-kernel

(b) Code snippet in C++ HLS

Allo: A Programming Model for Composable Accelerator Design [PLDI'24]

Challenge 1: Manual vs compiler opt.

Progressive hardware customization



Algorithm#1

Compute Customization

Algorithm#2

Data Type Customization

Data Placement Cust.

Algorithm#3

Algorithm#1-3

Compute Cust.

Data Type Cust.

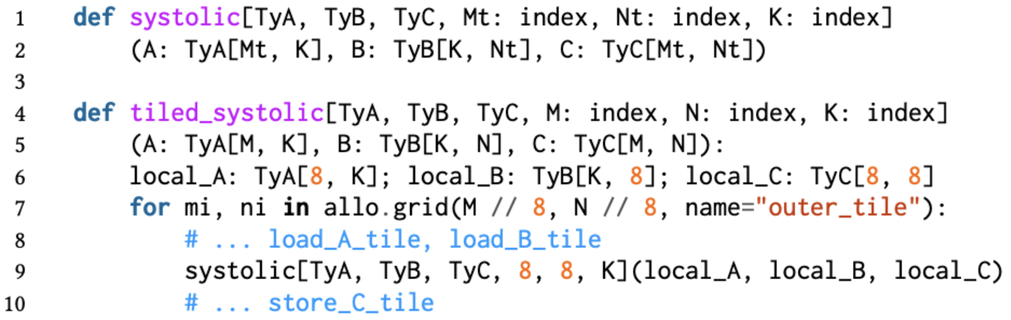
Memory Cust.

Comm. Cust.

Entangled algorithm specs and customization schemes

Fully decoupled customization schemes (i.e., schedule)

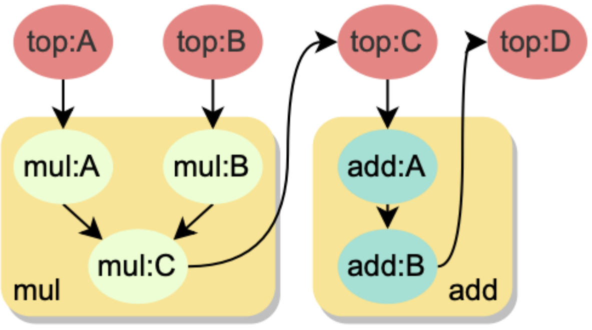
Reusable parameterized kernel templates



```
def systolic[TyA, TyB, TyC, Mt: index, Nt: index, K: index]
    (A: TyA[Mt, K], B: TyB[K, Nt], C: TyC[Mt, Nt])

def tiled_systolic[TyA, TyB, TyC, M: index, N: index, K: index]
    (A: TyA[M, K], B: TyB[K, N], C: TyC[M, N]):
    local_A: TyA[8, K]; local_B: TyB[K, 8]; local_C: TyC[8, 8]
    for mi, ni in allo.grid(M // 8, N // 8, name="outer_tile"):
        # ... load_A_tile, load_B_tile
        systolic[TyA, TyB, TyC, 8, 8, K](local_A, local_B, local_C)
        # ... store_C_tile
```

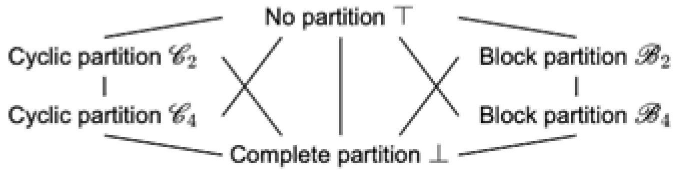
Challenge 2: Single -> Multi-kernel

Composable schedules

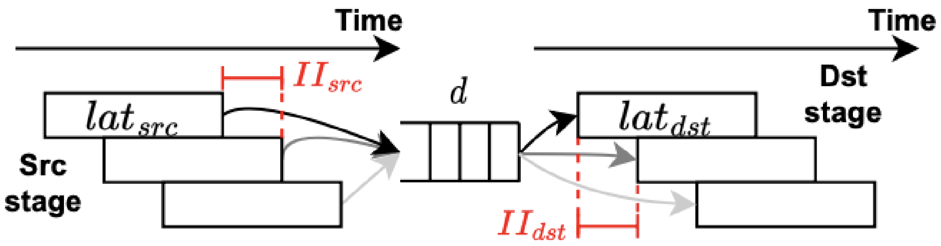


Hierarchical dataflow graph

Type-safe composition

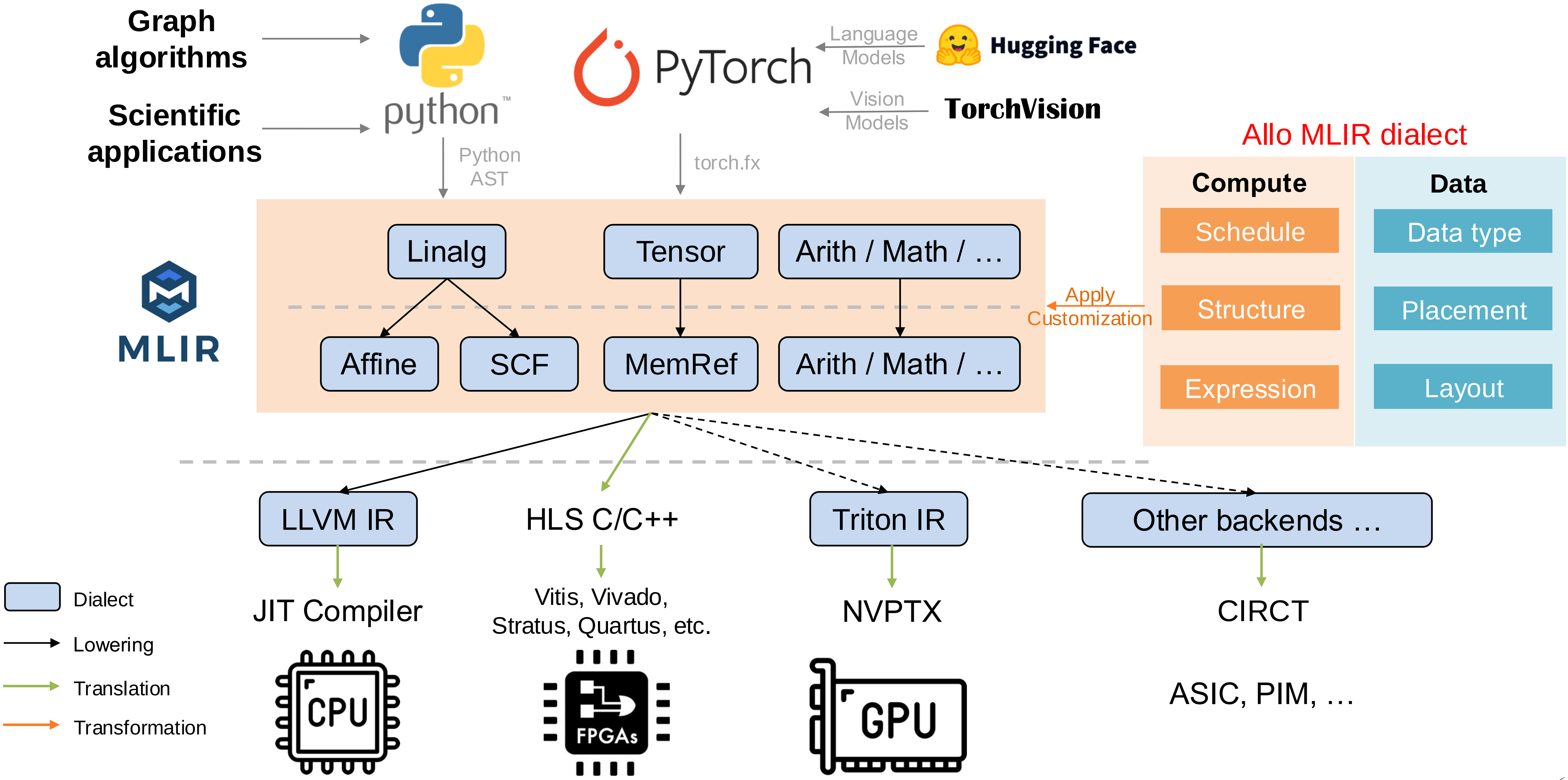

No partition $\top$
Cyclic partition $\mathscr{C}_2$
Cyclic partition $\mathscr{C}_4$
Block partition $\mathscr{B}_2$
Block partition $\mathscr{B}_4$
Complete partition $\bot$

Holistic dataflow optimizations



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

Overview of Allo ADL and Compilation Stack

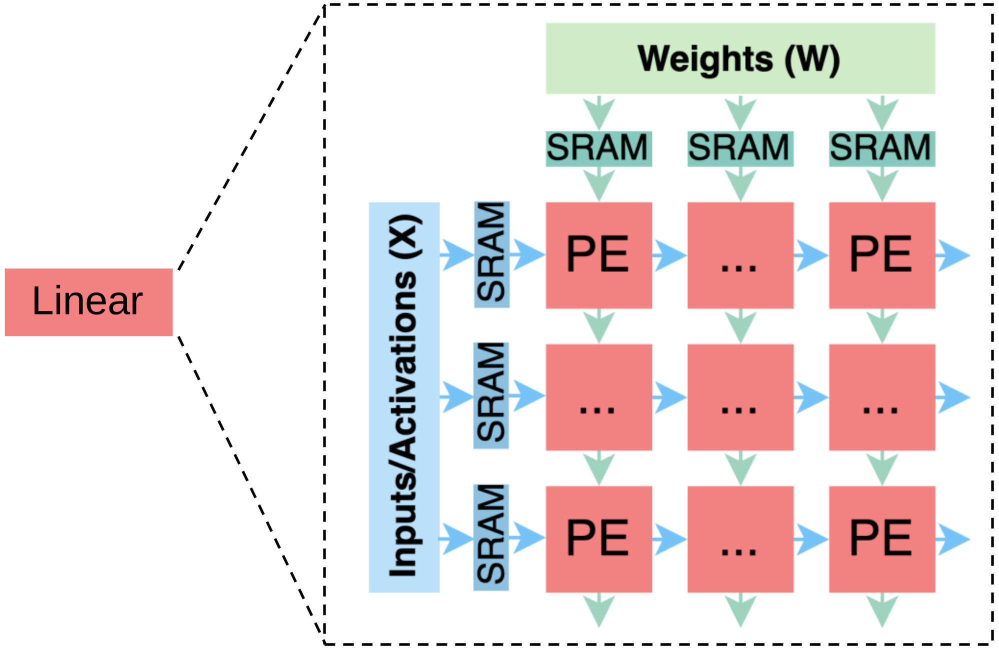


Goal: Design a High-Performance LLM Accelerator

Step 1: Construct building blocks

- Performance
- Correctness
- Reusability

- Linear operators

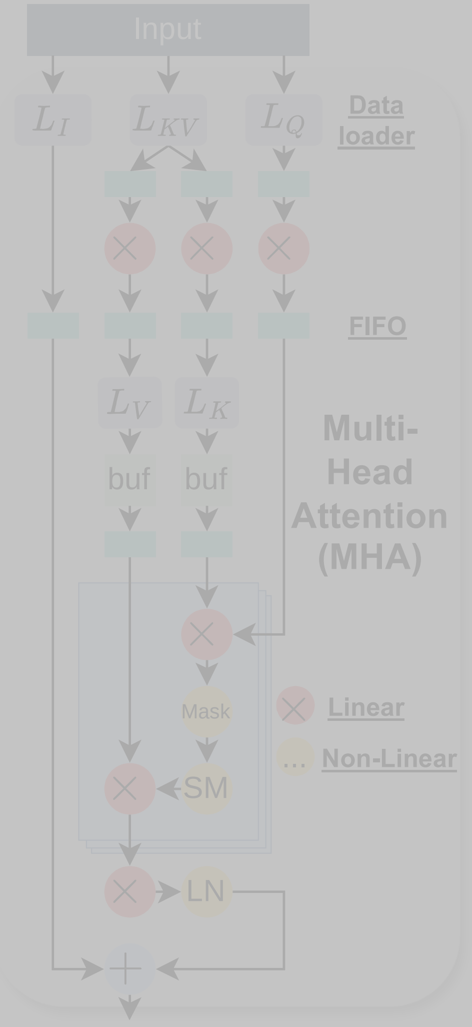


- Non-linear operators

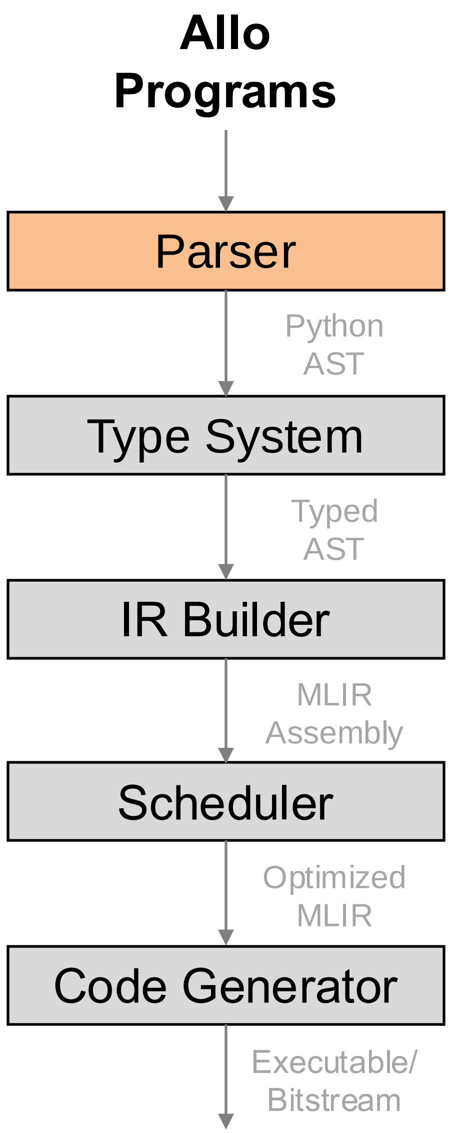
Softmax LayerNorm GELU

Step 2: Ensemble into a complete design

- Connect different operators bottom-up



Allo ADL Compilation Stack



Allo Programs

Parser

Python AST

Type System

Typed AST

IR Builder

MLIR Assembly

Scheduler

Optimized MLIR

Code Generator

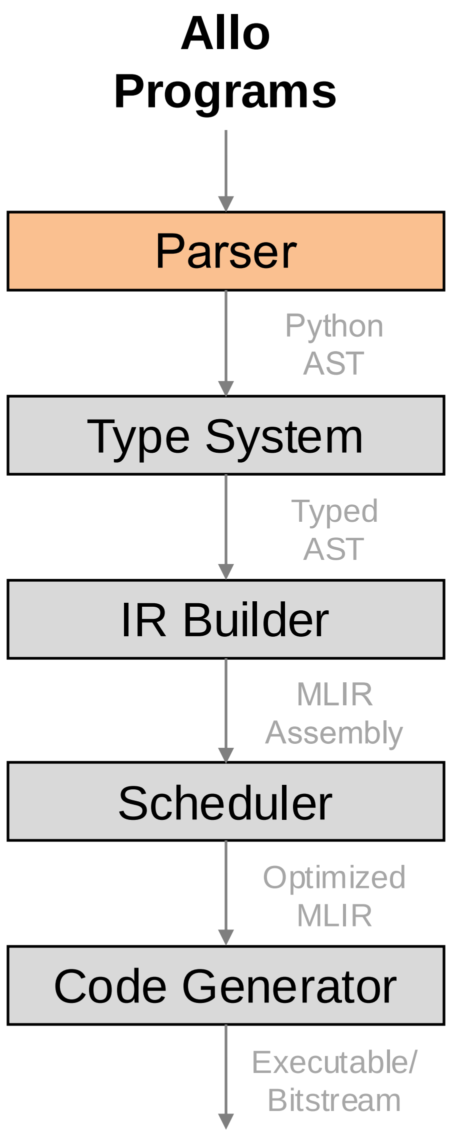
Executable/ Bitstream

Pythonic: No need to learn a new DSL!

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

```
def gemm(A: int8[M, K],
         B: int8[K, N]) -> int8[M, N]:
  C: int8[M, N] = 0
  for i, j, k in allo.grid(M, N, K):
    C[i, j] += A[i, k] * B[k, j]
  return C

s = allo.customize(gemm)
```

Allo ADL Compilation Stack



Allo Programs

Parser

Python AST

Type System

Typed AST

IR Builder

MLIR Assembly

Scheduler

Optimized MLIR

Code Generator

Executable/ Bitstream

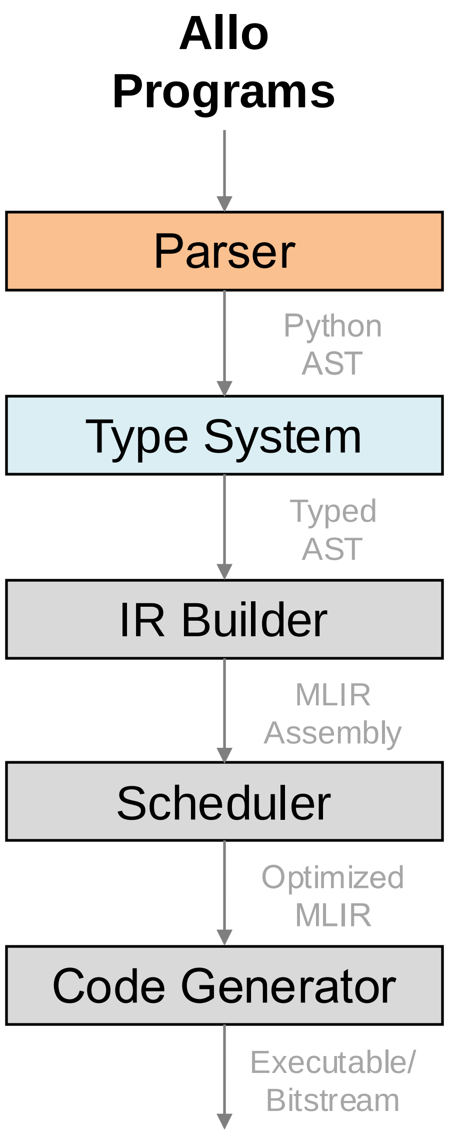
Pythonic: No need to learn a new DSL!

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

```
def gemm(A: int8[M, K],
         B: int8[K, N]) -> int8[M, N]:
  C: int8[M, N] = 0
  for i, j, k in allo.grid(M, N, K):
    C[i, j] += A[i, k] * B[k, j]
  return C

s = allo.customize(gemm)
```

```
T ::= eleTy | eleTy [ shape ]
shape ::= dim | dim , shape
dim   ::= [0-9]+ | constvar | expr
expr  ::= dim binop dim
binop ::= + | - | * | /
eleTy ::= intTy | floatTy | fixedTy
intTy ::= int[0-9]+ | uint[0-9]+
floatTy ::= float16 | float32 | float64
fixedTy ::= fixed(width, frac)
          | ufixed(width, frac)
constvar ::= [a-zA-Z_][a-zA-Z0-9_]*
width ::= [0-9]+
frac  ::= [0-9]+
```

Allo ADL Compilation Stack

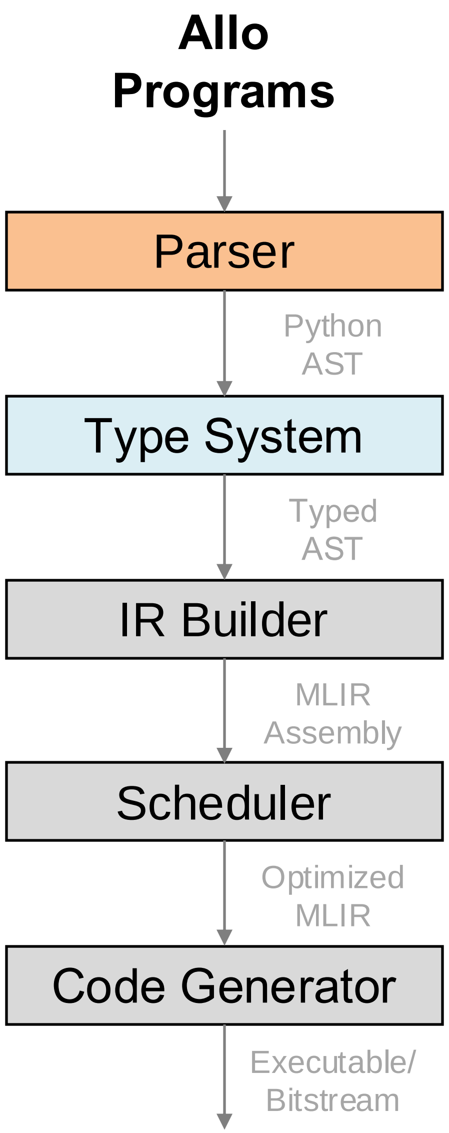


Type System

- Type checking: Inferred type different from annotated
- Type conversion: Type inference and implicit casting
- Shape propagation: Array broadcasting

```
M, N, K = 32, 32, 32

def gemm(A: int8[M, K], B: int8[K, N]) -> int8[M, N]:
  C: int8[M, N] = 0
  for i, j in allo.grid(M, N):
    v: int14 = 0
    for k in range(K):
      v += A[i, k] * B[k, j]
    C[i, j] = v
  return C
```

Allo ADL Compilation Stack



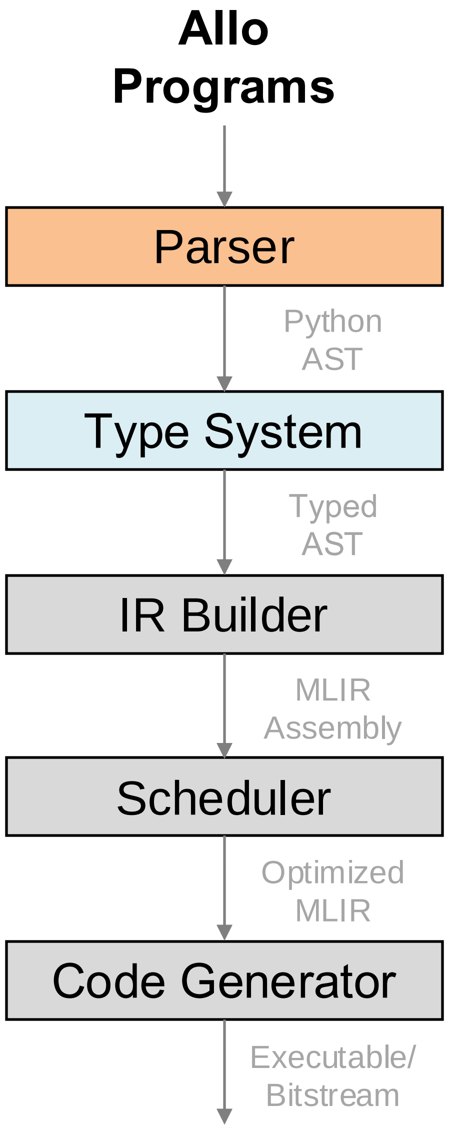
Type System

- Type checking: Inferred type different from annotated
- Type conversion: Type inference and implicit casting
- Shape propagation: Array broadcasting

Type inling & constant folding

```
def gemm(A: int8[32, 32], B: int8[32, 32])
        -> int8[32, 32]:
  C: int8[32, 32] = 0: int8 -> int8[32, 32]
  for i: index, j: index in allo.grid(M, N):
    v: int14 = 0: int14
    for k: index in range(K):
      v: int14 += (A[i, k]: int8
              * B[k, j]: int8) -> int16 -> int14
    C[i, j]: int8 = v: int14 -> int8
  return C: int8[32, 32]
```

Allo ADL Compilation Stack

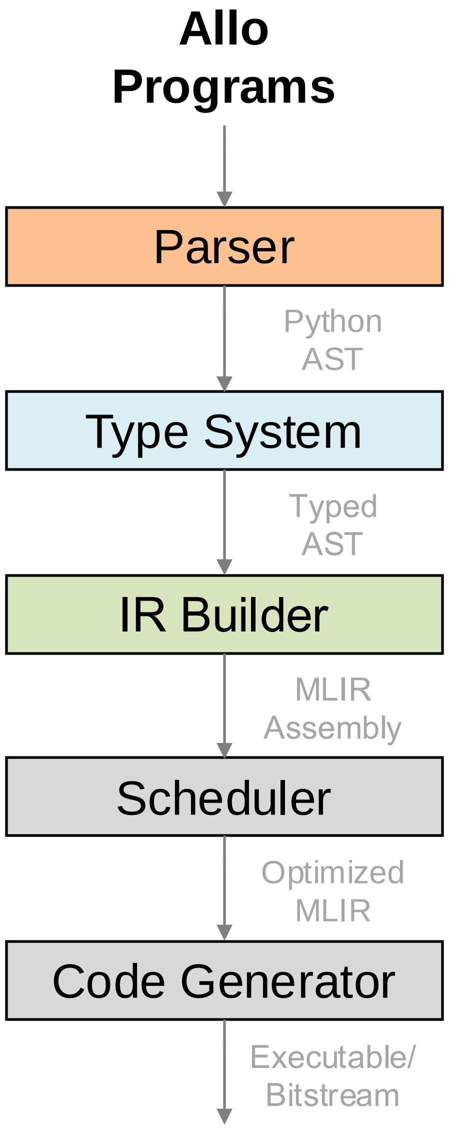


Type System

- Type checking: Inferred type different from annotated
- Type conversion: Type inference and implicit casting
- Shape propagation: Array broadcasting
- Quantization: Automatic fixed point to int conversion

```
T_IN, T_OUT = Fixed(7, 1), Fixed(15, 4)

def gemm(A: T_IN[M, K], B: T_IN[K, N]) -> T_OUT[M, N]:
    C: T_OUT[M, N] = 0
    for i, j, k in allo.grid(M, N, K, name="C"):
        C[i, j] += A[i, k] * B[k, j]
    return C

s = allo.customize(gemm)
```

Allo ADL Compilation Stack



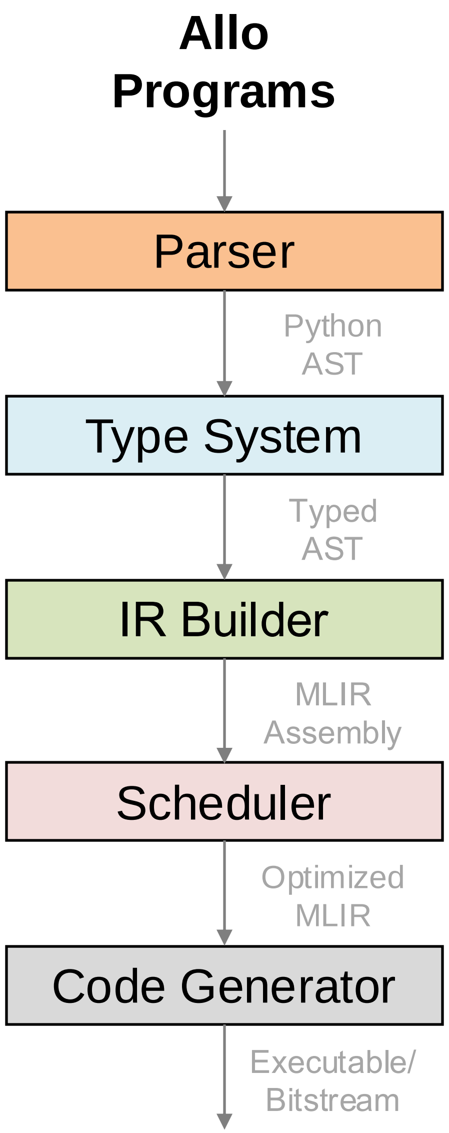
IR Builder

- MLIR builtin dialects
- One-to-one mapping, strictly follow program structure

```
def gemm(A: int8[M, K],
         B: int8[K, N])
         -> int8[M, N]:
  C: int8[M, N] = 0
  for i, j, k in allo.grid(M, N, K):
    C[i, j] += A[i, k] * B[k, j]
  return C

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

```
module {
 func.func @gemm(%arg0: memref<32x32xi8>,
                 %arg1: memref<32x32xi8>)
                   -> memref<32x32xi8> {
 %0 = memref.alloc() {name = "C"} : memref<32x32xi8>
 %c0_i8 = arith.constant 0 : i8
 linalg.fill ins(%c0_i8 : i8) outs(%0 : memref<32x32xi8>)
 affine.for %arg2 = 0 to 32 {
  affine.for %arg3 = 0 to 32 {
   affine.for %arg4 = 0 to 32 {
     ...
   } {loop_name = "k"}
  } {loop_name = "j"}
 } {loop_name = "i", op_name = "S_i_j_k_0"}
 return %0 : memref<32x32xi8>
}}
```

Allo ADL Compilation Stack



Allo Programs

Parser

Python AST

Type System

Typed AST

IR Builder

MLIR Assembly

Scheduler

Optimized MLIR

Code Generator

Executable/ Bitstream

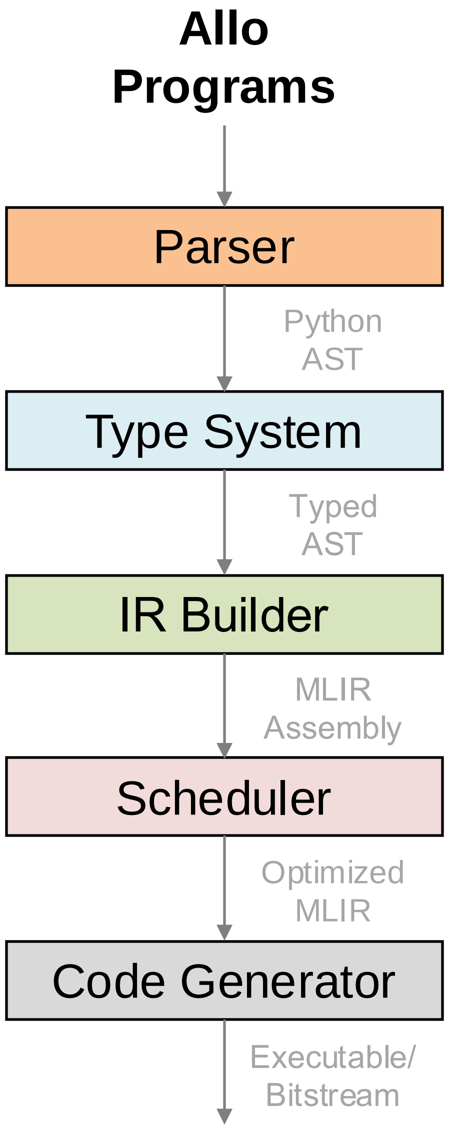
Scheduler

- Decoupled customizations
- Real-time transformation

```
def gemm(A: int8[M, K],
         B: int8[K, N])
         -> int8[M, N]:
 C: int8[M, N] = 0
 for i, j, k in allo.grid(M, N, K):
   C[i, j] += A[i, k] * B[k, j]
 return C

s = allo.customize(gemm)
i_out, i_in = s.split("i", 8)
print(s.module)
```

```
#map = affine_map<(d0, d1) -> (d0 + d1 * 8)>
module {
 func.func @gemm(%arg0: memref<32x32xi8>,
                 %arg1: memref<32x32xi8>)
                   -> memref<32x32xi8> {
 %0 = memref.alloc() {name = "C"} : memref<32x32xi8>
 %c0_i8 = arith.constant 0 : i8
 linalg.fill ins(%c0_i8 : i8) outs(%0 : memref<32x32xi32>)
 affine.for %arg2 = 0 to 4 {
  affine.for %arg3 = 0 to 8 {
   affine.for %arg4 = 0 to 32 {
    affine.for %arg5 = 0 to 32 {
     ...
   } {loop_name = "k"}
  } {loop_name = "j"}
 } {loop_name = "i.inner"}
} {loop_name = "i.outer", op_name = "S_i_j_k_0"}
return %0 : memref<32x32xi8>
}}
```

Allo ADL Compilation Stack



Allo Programs

Parser

Python AST

Type System

Typed AST

IR Builder

MLIR Assembly

Scheduler

Optimized MLIR

Code Generator

Executable/ Bitstream

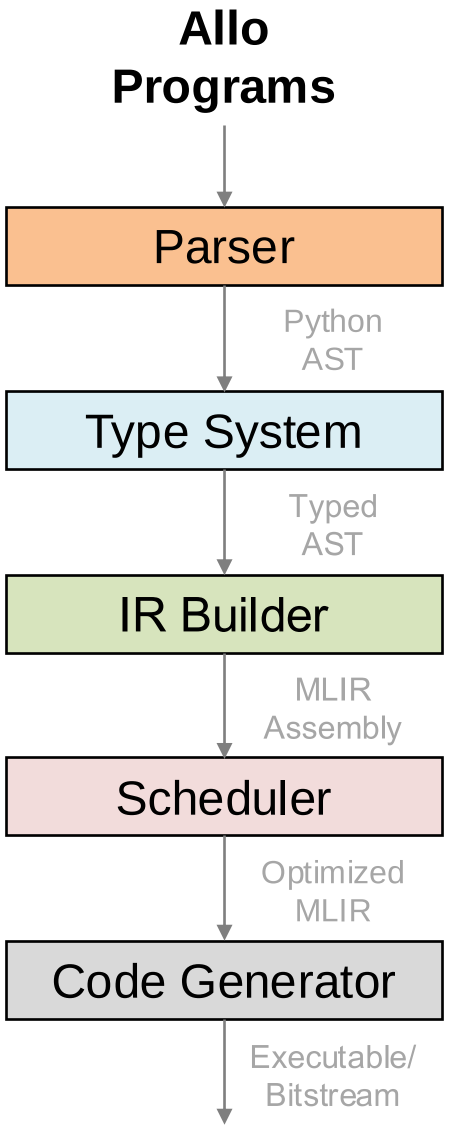
Scheduler

- Decoupled customizations
- Real-time transformation

```
def gemm(A: int8[M, K],
         B: int8[K, N])
         -> int8[M, N]:
  C: int8[M, N] = 0
  for i, j, k in allo.grid(M, N, K):
    C[i, j] += A[i, k] * B[k, j]
  return C

s = allo.customize(gemm)
i_out, i_in = s.split("i", 8)
print(s.module)
j_out, j_in = s.split("j", 8)
print(s.module)
```

```
#map = affine_map<(d0, d1) -> (d0 + d1 * 8)>
module {
 func.func @gemm(%arg0: memref<32x32xi8>,
                 %arg1: memref<32x32xi8>)
                   -> memref<32x32xi8> {
 %0 = memref.alloc() {name = "C"} : memref<32x32xi8>
 %c0_i8 = arith.constant 0 : i8
 linalg.fill ins(%c0_i8 : i8) outs(%0 : memref<32x32xi8>)
 affine.for %arg2 = 0 to 4 {
  affine.for %arg3 = 0 to 8 {
   affine.for %arg4 = 0 to 4 {
    affine.for %arg5 = 0 to 8 {
     affine.for %arg6 = 0 to 32 {
      ...
     } {loop_name = "k"}
    } {loop_name = "j.inner"}
   } {loop_name = "j.outer"}
  } {loop_name = "i.inner"}
} {loop_name = "i.outer"}
return %0 : memref<32x32xi8>}}
```

Allo ADL Compilation Stack



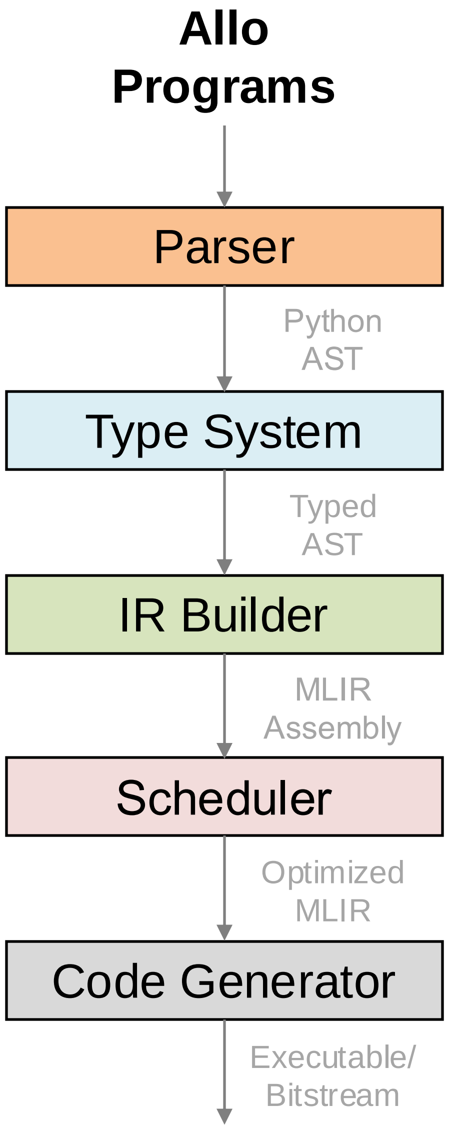
Scheduler

- Decoupled customizations
- Real-time transformation

```
def gemm(A: int8[M, K],
         B: int8[K, N])
         -> int8[M, N]:
 C: int8[M, N] = 0
 for i, j, k in allo.grid(M, N, K):
   C[i, j] += A[i, k] * B[k, j]
 return C

s = allo.customize(gemm)
i_out, i_in = s.split("i", 8)
print(s.module)
j_out, j_in = s.split("j", 8)
print(s.module)
s.reorder(i_out, j_out, i_in, j_in)
print(s.module)
```

```
#map = affine_map<(d0, d1) -> (d0 + d1 * 8)>
module {
 func.func @gemm(%arg0: memref<32x32xi8>,
                 %arg1: memref<32x32xi8>)
                   -> memref<32x32xi8> {
 %0 = memref.alloc() {name = "C"} : memref<32x32xi8>
 %c0_i8 = arith.constant 0 : i8
 linalg.fill ins(%c0_i8 : i8) outs(%0 : memref<32x32xi8>)
 affine.for %arg2 = 0 to 4 {
  affine.for %arg3 = 0 to 8 {
   affine.for %arg4 = 0 to 4 {
    affine.for %arg5 = 0 to 8 {
     affine.for %arg6 = 0 to 32 {
      ...
     } {loop_name = "k"}
    } {loop_name = "j.inner"}
   } {loop_name = "i.inner"}
  } {loop_name = "j.outer"}
 } {loop_name = "i.outer"}
 return %0 : memref<32x32xi8>}}
```

Allo ADL Compilation Stack



Allo Programs

Parser

Python AST

Type System

Typed AST

IR Builder

MLIR Assembly

Scheduler

Optimized MLIR

Code Generator

Executable/ Bitstream

Codegen

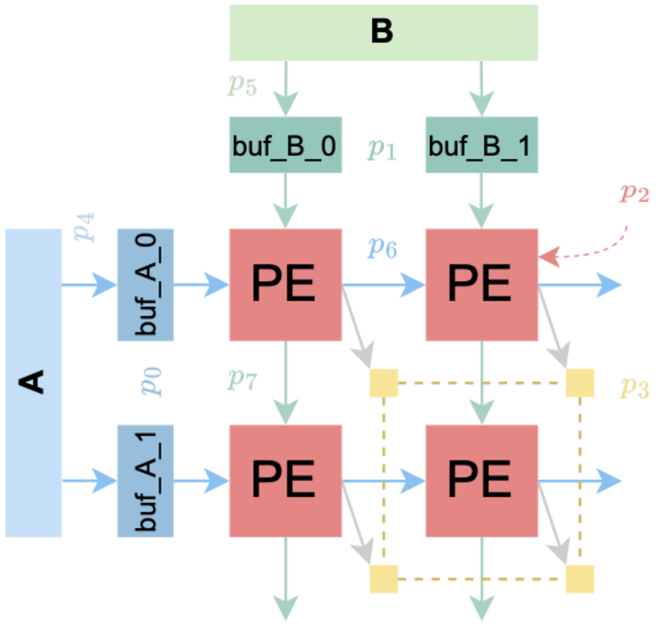
- Default: CPU backend for simulation testing

```
def gemm(A: int8[M, K],
         B: int8[K, N])
         -> int8[M, N]:
 C: int8[M, N] = 0
 for i, j, k in allo.grid(M, N, K):
   C[i, j] += A[i, k] * B[k, j]
 return C

s = allo.customize(gemm)
# ... customizations (omitted)
X = np.random.randint(-8, 8, size=(M, K)).astype(np.int8)
A = np.random.randint(-8, 8, size=(K, N)).astype(np.int8)
np_outs = gemm(X, A)

f = s.build(target="llvm")
outs = f(X, A)
np.testing.assert_allclose(outs, np_outs, atol=1e-3)
```

Allo ADL Example - Systolic Array

- Transform a vanilla GEMM implementation into a high-performance systolic array
- Only 8 lines of schedule code is needed!

```
# 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")            # p0
buf_B = s.buffer_at(s.B, "j")            # p1
pe = s.unfold("PE", axis=[0, 1])         # p2
s.partition(s.C, dim=[0, 1])             # p3
s.partition(s.A, dim=0)                  # p4
s.partition(s.B, dim=1)                  # p5
s.relay(buf_A, pe, axis=1, depth=M + 1)  # p6
s.relay(buf_B, pe, axis=0, depth=N + 1)  # p7
```



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

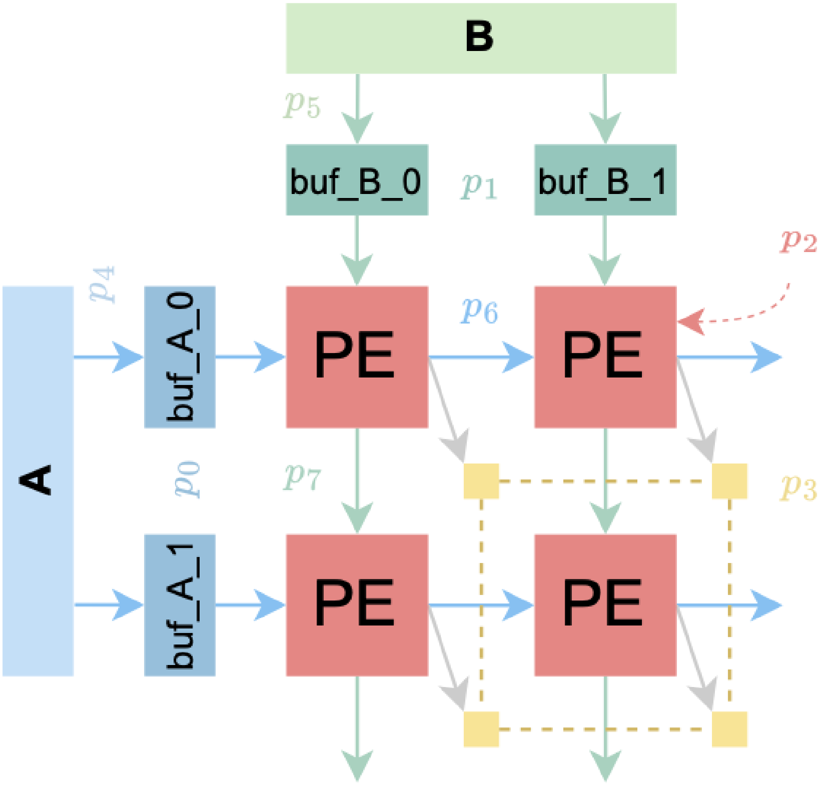
	Latency (ms)	BRAM	DSP	FF	LUT
Ours (w/o DSP packing)	15.73	0 (0%)	256 (2%)	88284 (3%)	168190 (12%)
Ours (w/ DSP packing)	15.73	0 (0%)	**128 (1%)**	79969 (3%)	244439 (18%)
AutoSA [75]	15.71	514 (12%)	256 (2%)	100138 (3%)	244032 (18%)

These schedules can indeed generate high-performance designs

Verifiable Schedule [FPGA'24 Best Paper]

- s.verify(orig_sch, new_sch)
  - Integrated CDAG verifier
  - Able to verify statically interpretable control-flow (SICF) programs

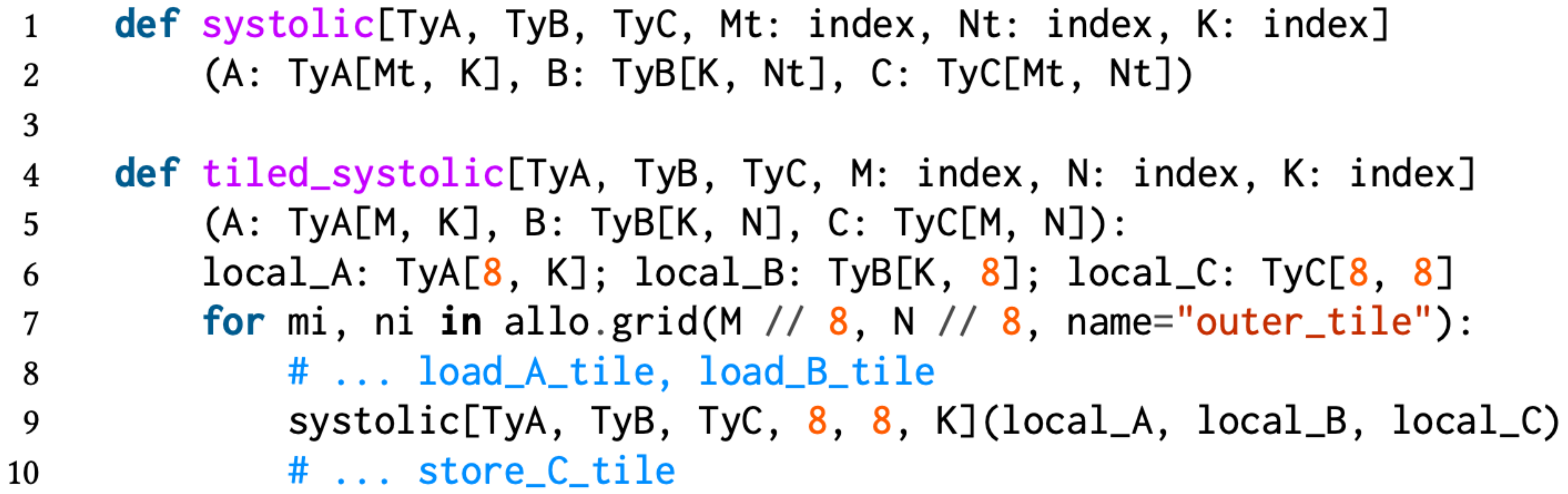
```
# Algorithm specification
def gemm(A: int8[M, K], B: int8[K, N],
         C: int16[M, N]):
  for i, j in allo.grid(M, N, "PE"):
    for k in range(K):
      C[i, j] += A[i, k] * B[k, j]

# Schedule construction
s = allo.customize(gemm)
buf_A = s.buffer_at(s.A, "j")                # p0
buf_B = s.buffer_at(s.B, "j")                # p1
pe = s.unfold("PE", axis=[0, 1])             # p2
s.partition(s.C, dim=[0, 1])                 # p3
s.partition(s.A, dim=0)                      # p4
s.partition(s.B, dim=1)                      # p5
s.relay(buf_A, pe, axis=1, depth=M + 1)      # p6
s.relay(buf_B, pe, axis=0, depth=N + 1)      # p7
```



[*] Louis-Noël Pouchet, Emily Tucker, Niansong Zhang, Hongzheng Chen, Debjit Pal, Gabriel Rodríguez, Zhiru Zhang, "Formal Verification of Source-to-Source Transformation for HLS", FPGA, 2024.

Parameterized Kernel Template

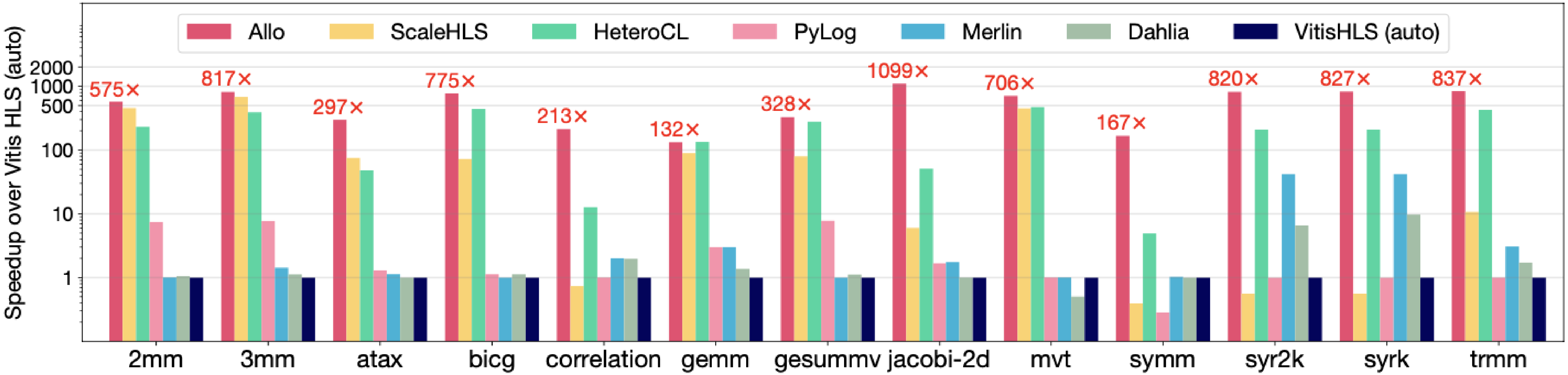
- Reusability: Parameterized templates
  - Parameter types: New feature in Python 3.12
  - Works like C++ template
  - Parameterizable shapes and types
  - Build kernel libraries in Python



```python
def systolic[TyA, TyB, TyC, Mt: index, Nt: index, K: index]
    (A: TyA[Mt, K], B: TyB[K, Nt], C: TyC[Mt, Nt])

def tiled_systolic[TyA, TyB, TyC, M: index, N: index, K: index]
    (A: TyA[M, K], B: TyB[K, N], C: TyC[M, N]):
    local_A: TyA[8, K]; local_B: TyB[K, 8]; local_C: TyC[8, 8]
    for mi, ni in allo.grid(M // 8, N // 8, name="outer_tile"):
        # ... load_A_tile, load_B_tile
        systolic[TyA, TyB, TyC, 8, 8, K](local_A, local_B, local_C)
        # ... store_C_tile
```

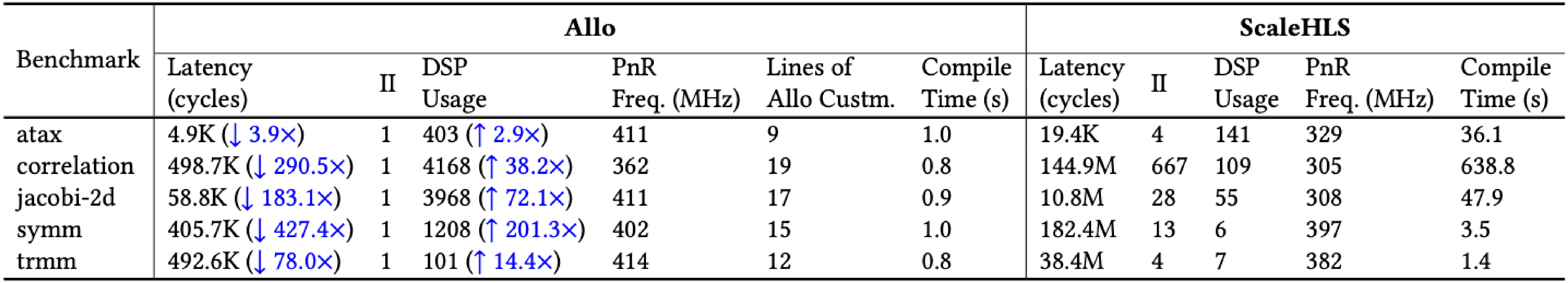
Single-Kernel Evaluation

- Normalized against VitisHLS auto baseline (no pragma inserted)
- Automated DSE: ScaleHLS, Merlin
- Human designed customizations: Allo, HeteroCL, PyLog, Dahlia



Single-Kernel Evaluation

- Compared to ScaleHLS DSE results, Allo achieves:
  - Lower II, II=1
  - More efficient use of computation resources
  - Higher PnR frequency due to better pipelined designs



Benchmark	Allo						ScaleHLS				
	Latency (cycles)	II	DSP Usage	PnR Freq. (MHz)	Lines of Allo Custm.	Compile Time (s)	Latency (cycles)	II	DSP Usage	PnR Freq. (MHz)	Compile Time (s)
atax	4.9K (↓ 3.9×)	1	403 (↑ 2.9×)	411	9	1.0	19.4K	4	141	329	36.1
correlation	498.7K (↓ 290.5×)	1	4168 (↑ 38.2×)	362	19	0.8	144.9M	667	109	305	638.8
jacobi-2d	58.8K (↓ 183.1×)	1	3968 (↑ 72.1×)	411	17	0.9	10.8M	28	55	308	47.9
symm	405.7K (↓ 427.4×)	1	1208 (↑ 201.3×)	402	15	1.0	182.4M	13	6	397	3.5
trmm	492.6K (↓ 78.0×)	1	101 (↑ 14.4×)	414	12	0.8	38.4M	4	7	382	1.4

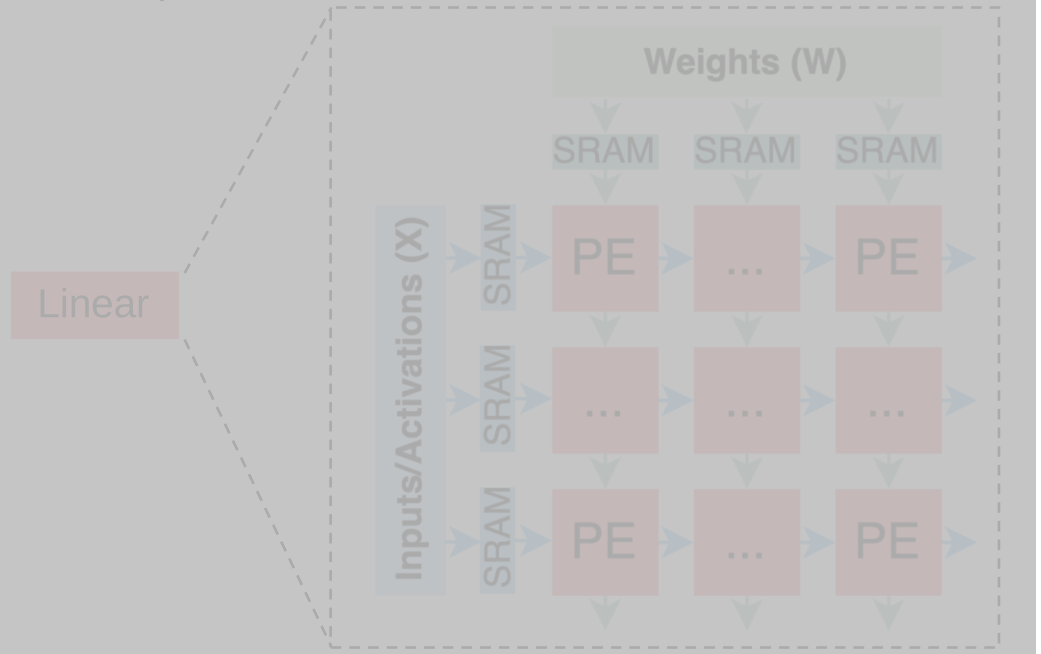
Memory & Communication customization can introduce larger design space leading to better performance

Goal: Design a High-Performance LLM Accelerator

Step 1: Construct building blocks

- Performance
- Correctness
- Reusability

- Linear operators

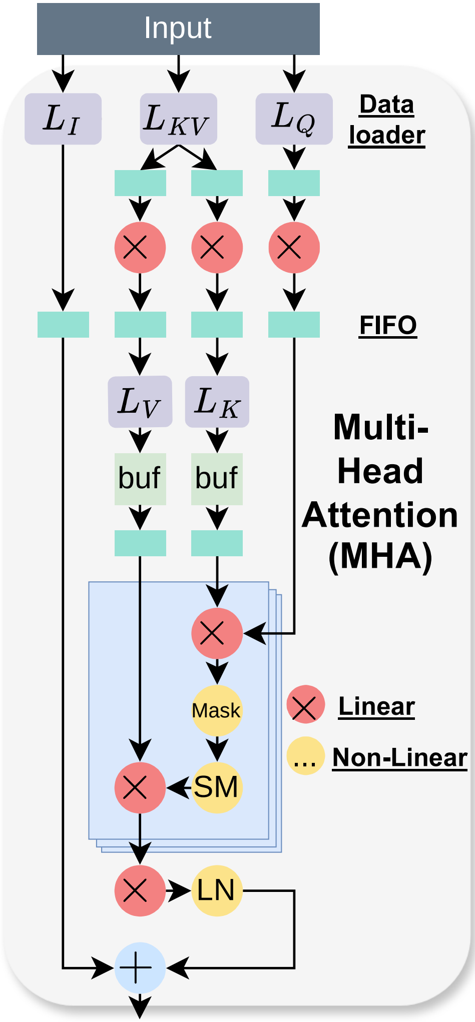


- Non-linear operators

Softmax LayerNorm GELU

Step 2: Ensemble into a complete design

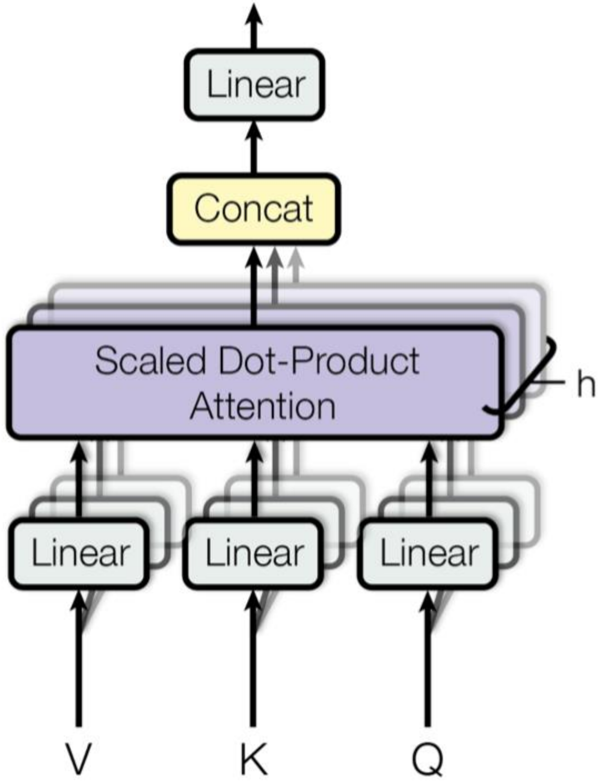
- Connect different operators bottom-up



We have the kernel library, but how to compose them to form a Transformer layer?

Allo ADL Example - Transformer Kernel (Alg. Specification)

- Allo ADL
  - High-level tensor-based operations
  - NumPy-like interface

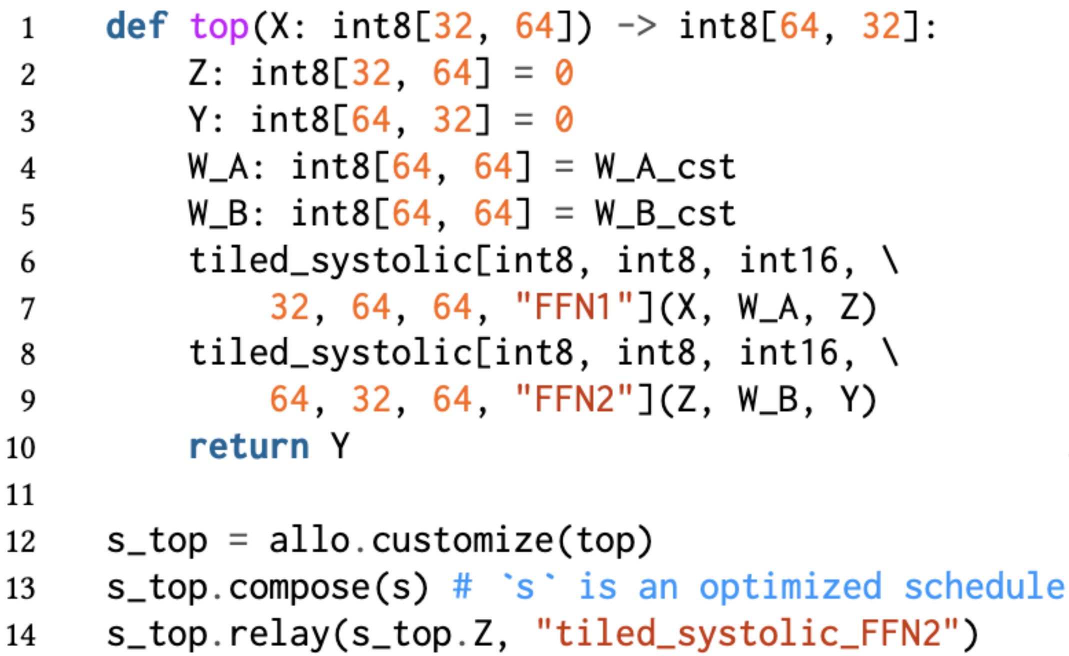


Each kernel has its own schedule, so how to glue them together?

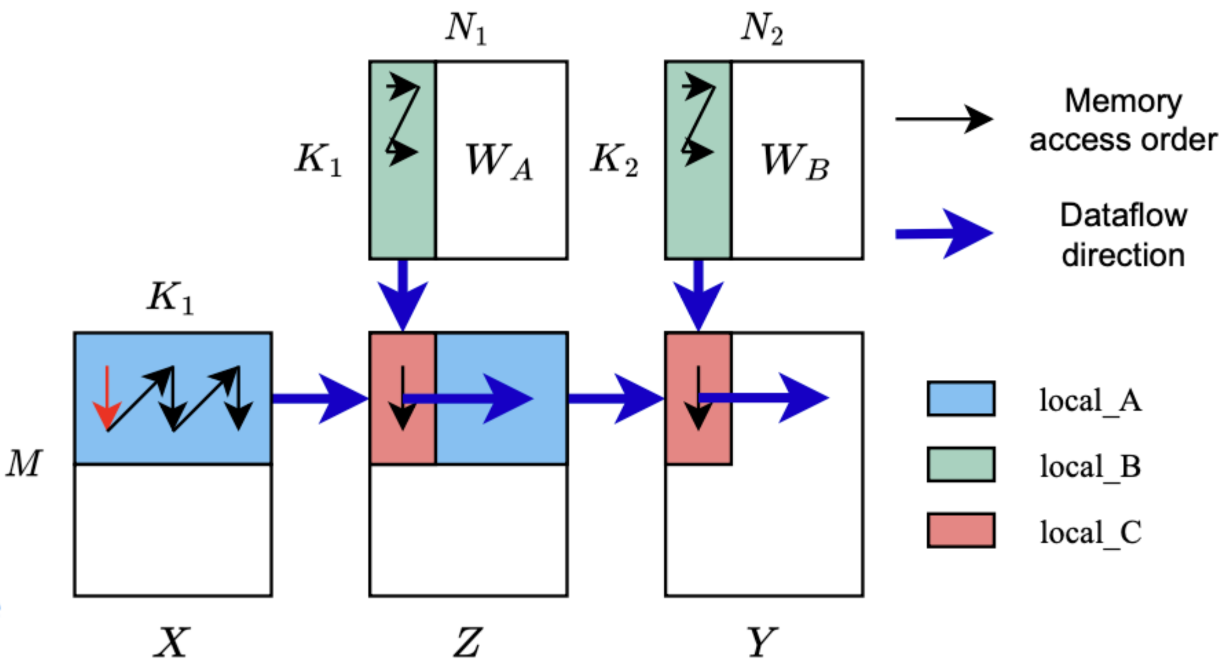
```
def attention(hidden_states: float32[2, 512, 768])
                        -> float32[2, 512, 768]:
  q_proj_weight: float32[768, 768] = global_q_proj_weight
  q_proj_bias: float32[768] = global_q_proj_bias
  k_proj_weight: float32[768, 768] = global_k_proj_weight
  k_proj_bias: float32[768] = global_k_proj_bias
  v_proj_weight: float32[768, 768] = global_v_proj_weight
  v_proj_bias: float32[768] = global_v_proj_bias
  q_proj = allo.linear(hidden_states, q_proj_weight, q_proj_bias)
  reshape = allo.reshape(q_proj, (2, 512, 12, 64))
  permute = allo.transpose(reshape, (0, 2, 1, 3))
  k_proj = allo.linear(hidden_states, k_proj_weight, k_proj_bias)
  reshape_1 = allo.reshape(k_proj, (2, 512, 12, 64))
  permute_1 = allo.transpose(reshape_1, (0, 2, 1, 3))
  v_proj = allo.linear(hidden_states, v_proj_weight, v_proj_bias)
  reshape_2 = allo.reshape(v_proj, (2, 512, 12, 64))
  permute_2 = allo.transpose(reshape_2, (0, 2, 1, 3))
  transpose = allo.transpose(permute_1, (-2, -1))
  matmul = allo.matmul(permute, transpose)
  truediv = matmul / 8.0
  softmax = allo.softmax(truediv)
  matmul_1 = allo.matmul(softmax, permute_2)
  permute_3 = allo.transpose(matmul_1, (0, 2, 1, 3))
  reshape_3 = allo.reshape(permute_3, (2, 512, 768))
  return reshape_3

s = allo.customize(attention)
mod = s.build()
example_inputs = [np.random.randn(2, 512, 768).astype(np.float32)]
out = mod(*example_inputs)
```

Composable Schedules

- s.compose(<new_schedule>, <id>)
  - <id>: Distinguish between different function calls
  - Can also compose external HLS IP modules

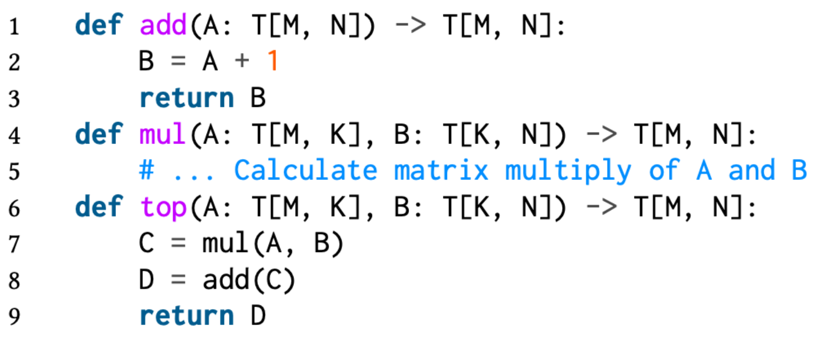


```python
def top(X: int8[32, 64]) -> int8[64, 32]:
    Z: int8[32, 64] = 0
    Y: int8[64, 32] = 0
    W_A: int8[64, 64] = W_A_cst
    W_B: int8[64, 64] = W_B_cst
    tiled_systolic[int8, int8, int16, \
        32, 64, 64, "FFN1"](X, W_A, Z)
    tiled_systolic[int8, int8, int16, \
        64, 32, 64, "FFN2"](Z, W_B, Y)
    return Y

s_top = allo.customize(top)
s_top.compose(s) # `s` is an optimized schedule
s_top.relay(s_top.Z, "tiled_systolic_FFN2")
```

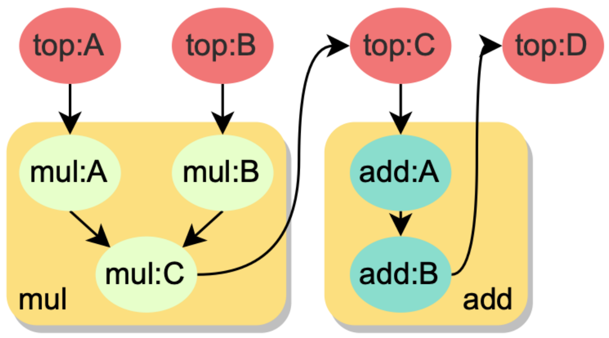


Composable Schedules: Implementation

- Hierarchical Dataflow Graph
  - Traditional DFG are flattened
  - Missing function boundaries info, less optimization opportunity



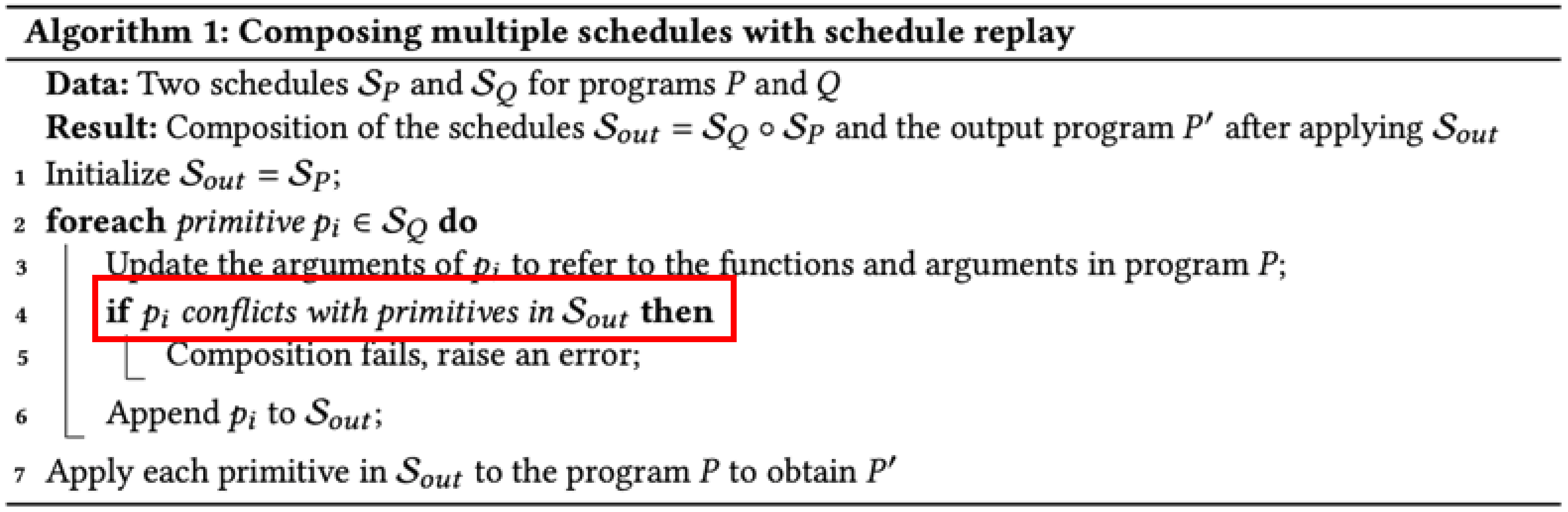
```
def add(A: T[M, N]) -> T[M, N]:
    B = A + 1
    return B
def mul(A: T[M, K], B: T[K, N]) -> T[M, N]:
    # ... Calculate matrix multiply of A and B
def top(A: T[M, K], B: T[K, N]) -> T[M, N]:
    C = mul(A, B)
    D = add(C)
    return D
```



How to resolve conflicts?

e.g., mul:C is partitioned in mul, but mul wants to be composed into top, where top:C is NOT partitioned

- Schedule Replay
  - Replays customizations on design modules after composition



Algorithm 1: Composing multiple schedules with schedule replay

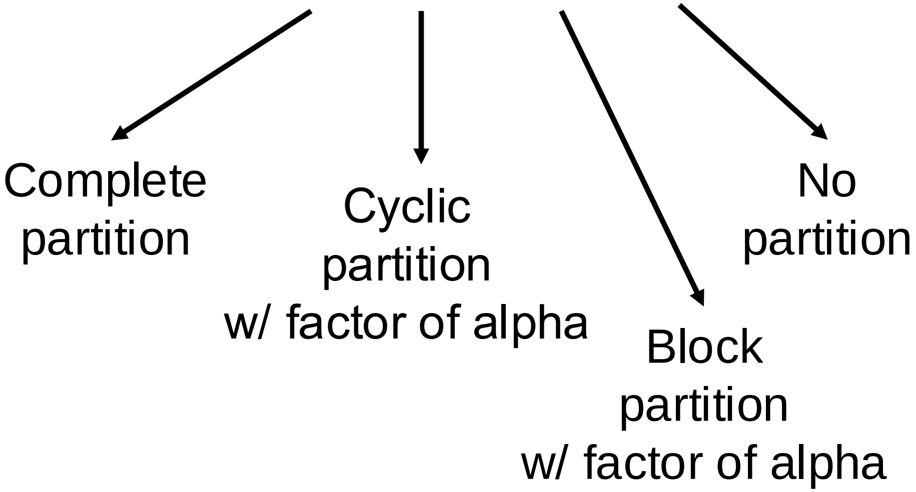
```
Data: Two schedules $\mathcal{S}_P$ and $\mathcal{S}_Q$ for programs $P$ and $Q$
Result: Composition of the schedules $\mathcal{S}_{out} = \mathcal{S}_Q \circ \mathcal{S}_P$ and the output program $P'$ after applying $\mathcal{S}_{out}$
Initialize $\mathcal{S}_{out} = \mathcal{S}_P$;
foreach primitive $p_i \in \mathcal{S}_Q$ do
    Update the arguments of $p_i$ to refer to the functions and arguments in program $P$;
    if $p_i$ conflicts with primitives in $\mathcal{S}_{out}$ then
        Composition fails, raise an error;
    Append $p_i$ to $\mathcal{S}_{out}$;
Apply each primitive in $\mathcal{S}_{out}$ to the program $P$ to obtain $P'$
```

Memory Layout Composition

- Goal: Ensure the layouts of function arguments are consistent
- **Key idea: Model data layout as a type**
- Data layout propagation -> type inference
  - N-D array layout (composite type): $\tau := (\hat{\tau}_1, \ldots, \hat{\tau}_N)$
  - Base type for each dimension:

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

$$\hat{\tau} := \bot \mid \mathcal{C}_\alpha \mid \mathcal{B}_\alpha \mid \top$$



Complete partition

Cyclic partition w/ factor of alpha

Block partition w/ factor of alpha

No partition

Subtyping relation: **X<:Y**
The code expecting a memory with partition type Y is also compatible with a memory with partition type X

e.g., $\bot <: \mathcal{C}_2$ since complete partitioning already partition the array into cyclic with a factor of 2

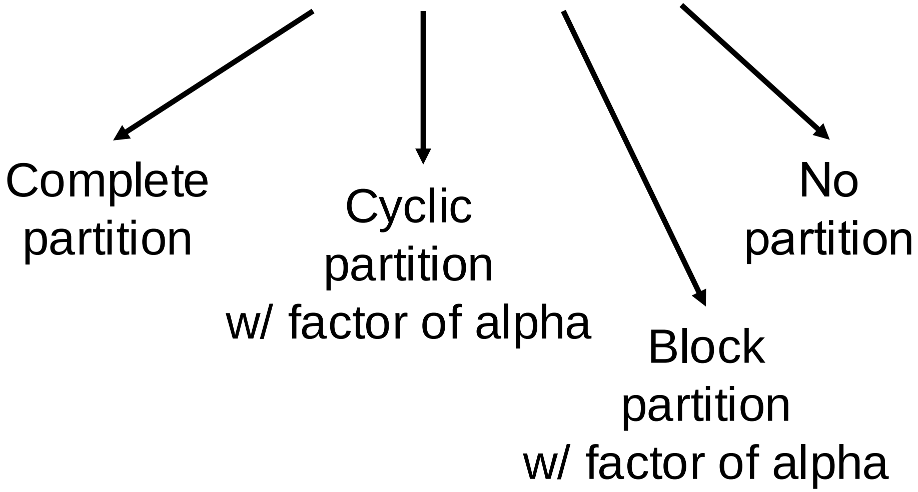
Intuition:
You can supply more read/write parallelism, but not less!

Memory Layout Composition

- Goal: Ensure the layouts of function arguments are consistent
- **Key idea: Model data layout as a type**
- Data layout propagation -> type inference
  - N-D array layout (composite type): $\tau := (\hat{\tau}_1, \ldots, \hat{\tau}_N)$
  - Base type for each dimension:

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

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



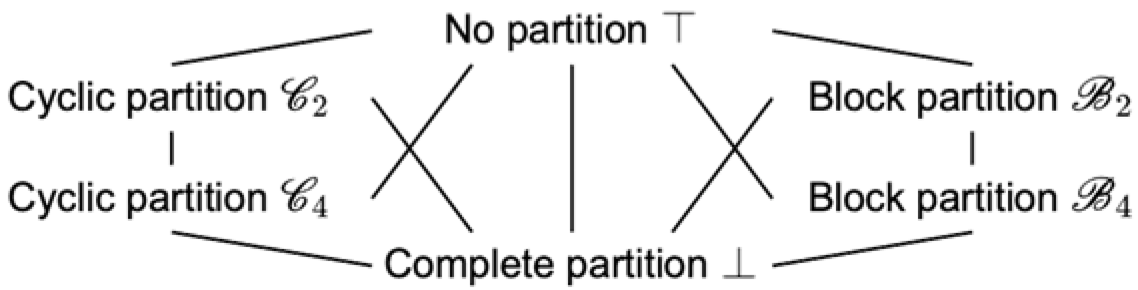
Complete partition

Cyclic partition w/ factor of alpha

Block partition w/ factor of alpha

No partition

Subtyping relation construct a **lattice**!



No partition $\top$

Cyclic partition $\mathscr{C}_2$

Block partition $\mathscr{B}_2$

Cyclic partition $\mathscr{C}_4$

Block partition $\mathscr{B}_4$

Complete partition $\bot$

An example lattice of partition types for a 1D array of shape (8,)

Memory Layout Composition

- Goal: Ensure the layouts of function arguments are consistent
- **Key idea: Model data layout as a type**
- Data layout propagation -> type inference
  - We check if layout is correctly composed with typing rules

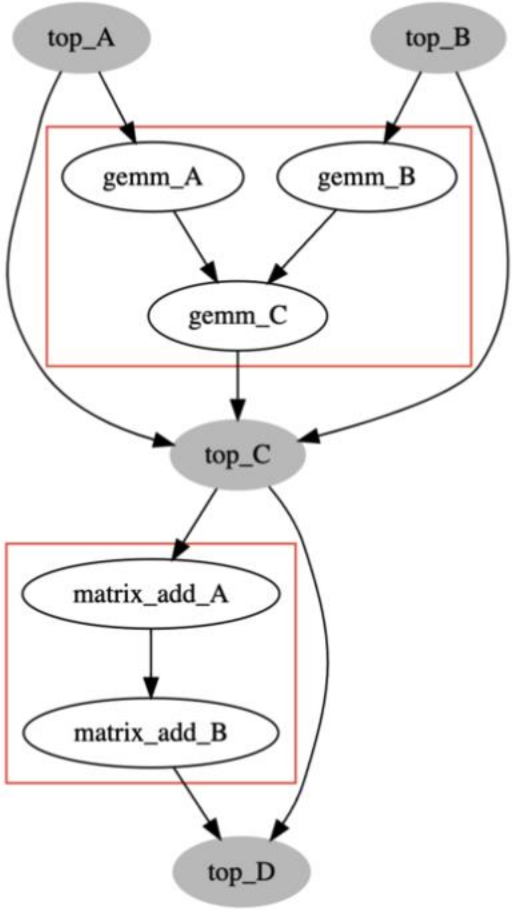
$$\textsc{S-Bottom-C}\quad \frac{}{\bot <: \mathcal{C}_{\alpha}} \qquad \textsc{S-Bottom-B}\quad \frac{}{\bot <: \mathcal{B}_{\alpha}} \qquad \textsc{S-Cyclic}\quad \frac{\alpha_2 \equiv 0 (\operatorname{mod} \alpha_1)}{\mathcal{C}_{\alpha_2} <: \mathcal{C}_{\alpha_1}} \qquad \textsc{S-Block}\quad \frac{\alpha_2 \equiv 0 (\operatorname{mod} \alpha_1)}{\mathcal{B}_{\alpha_2} <: \mathcal{B}_{\alpha_1}} \qquad \textsc{S-Top-C}\quad \frac{}{\mathcal{C}_{\alpha} <: \top} \qquad \textsc{S-Top-B}\quad \frac{}{\mathcal{B}_{\alpha} <: \top}$$

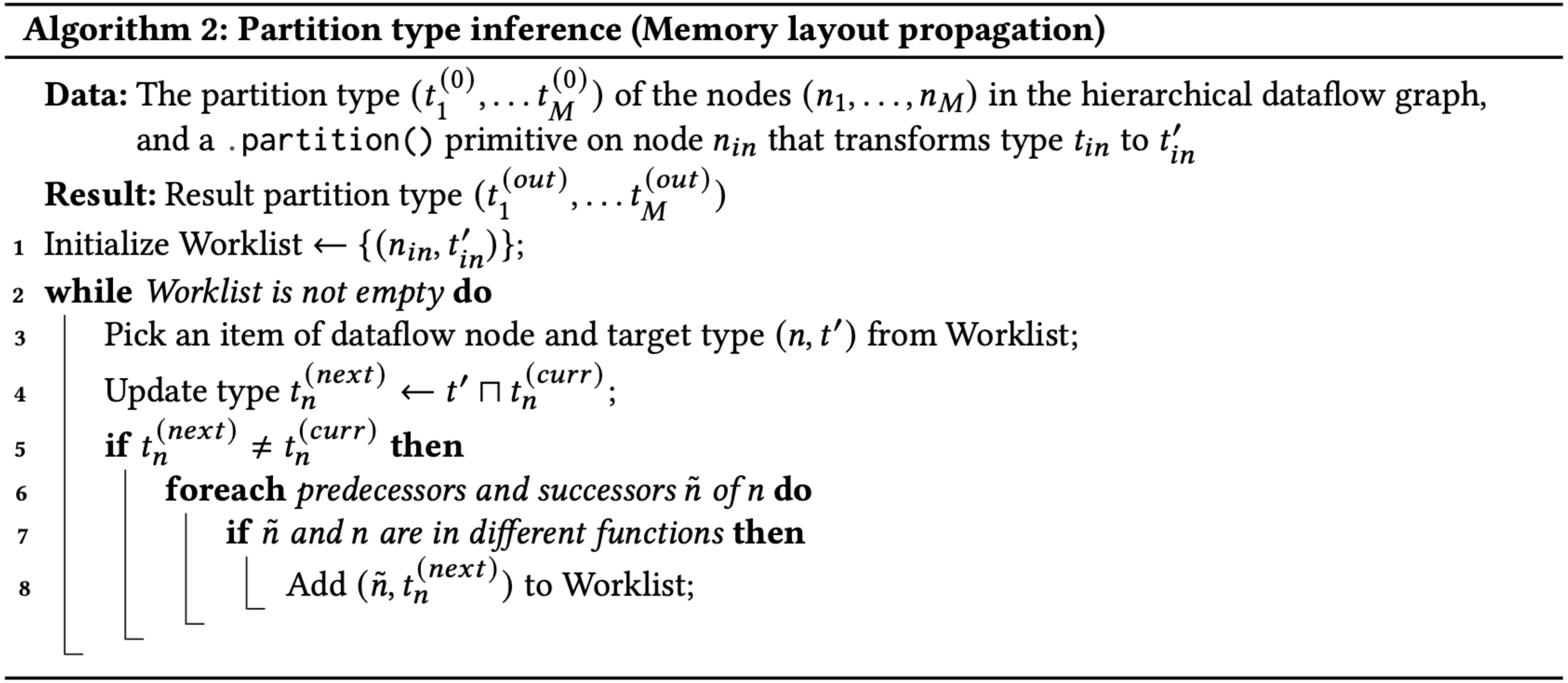
$$\textsc{S-Array}\quad \frac{\exists i \in \{1, \ldots, N\} : \ \hat{\tau}_i <: \hat{\tau}'_i}{(\hat{\tau}_1, \ldots, \hat{\tau}_N) <: (\hat{\tau}'_1, \ldots, \hat{\tau}'_N)} \qquad \textsc{FuncApp}\quad \frac{\Gamma \vdash f : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e : \tau_3 \quad \tau_3 <: \tau_1}{\Gamma \vdash fe : \tau_2}$$

Memory Layout Composition

- Unification algorithm in functional programming does not work for type systems with subtypes
- Lattice has good property! We can directly use dataflow analysis for type inference (Worklist algorithm)

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





Algorithm 2: Partition type inference (Memory layout propagation)

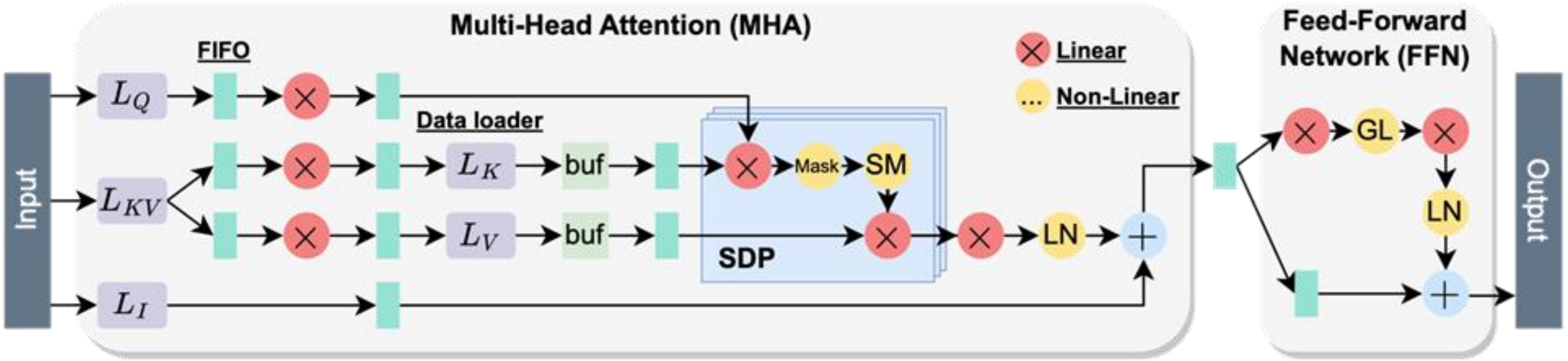
Data: The partition type $(t_1^{(0)}, \ldots t_M^{(0)})$ of the nodes $(n_1, \ldots, n_M)$ in the hierarchical dataflow graph, and a `.partition()` primitive on node n_{in} that transforms type t_{in} to t'_{in}

Result: Result partition type $(t_1^{(out)}, \ldots t_M^{(out)})$

```
Initialize Worklist ← $\{(n_{in}, t'_{in})\}$;
while Worklist is not empty do
    Pick an item of dataflow node and target type $(n, t')$ from Worklist;
    Update type $t_n^{(next)} \leftarrow t' \sqcap t_n^{(curr)}$;
    if $t_n^{(next)} \neq t_n^{(curr)}$ then
        foreach predecessors and successors $\tilde{n}$ of $n$ do
            if $\tilde{n}$ and $n$ are in different functions then
                Add $(\tilde{n}, t_n^{(next)})$ to Worklist;
```

A Complete LLM Accelerator

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



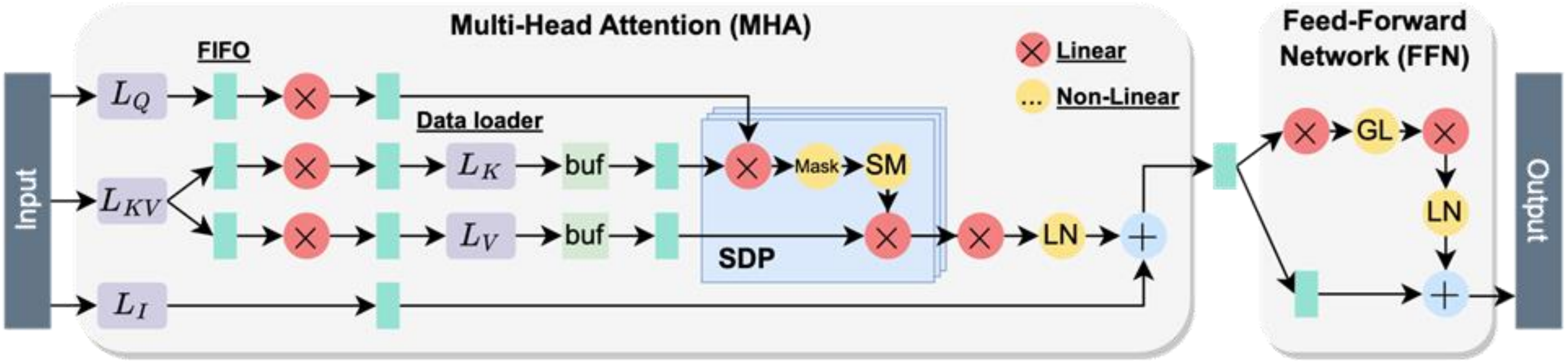
```
def GPT_layer(inp: float32[inp_num, inp_len], ...)
          -> float32[inp_num, inp_len]:
 # 1. Multi-Head Attention (MHA)
 Q = Linear_layer_qkv(inp, Wq, Bq)
 K = Linear_layer_qkv(inp, Wk, Bk)
 V = Linear_layer_qkv(inp, Wv, Bv)
 attn_sf_outp = Self_attention(Q, K, V)
 # ...
 # 2. Feed Forward Network (FFN)
 # ...
 return ffn_res_outp
```

Compose all the schedules together

```
s = allo.customize(GPT_layer)
s.compose(s_qkv)
s.compose(s_sfa)
s.compose(s_ds0)
s.compose(s_res)
s.compose(s_ln)
s.compose(s_ds1)
s.compose(s_ds2)
s.compose(s_gelu)
```

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!

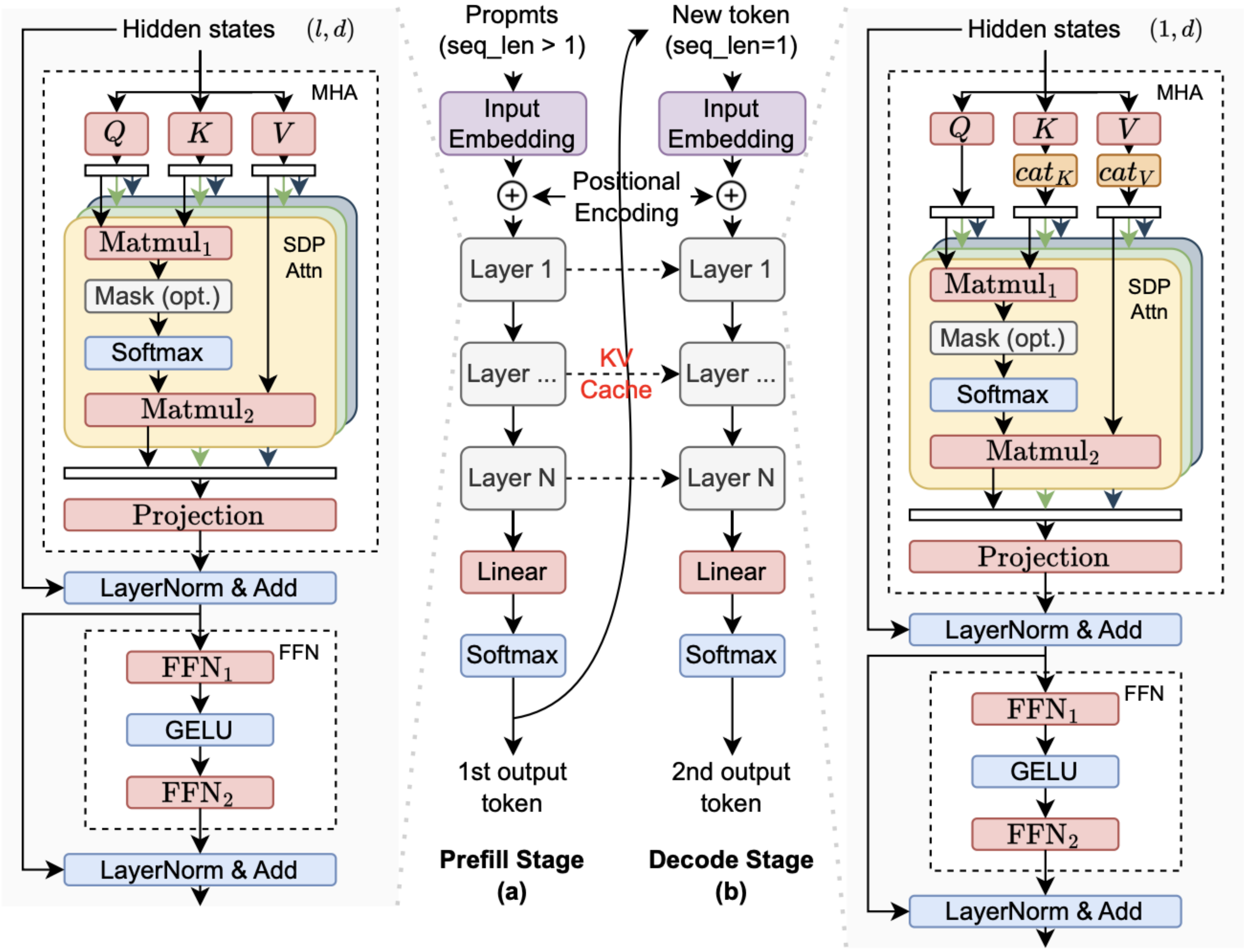
Compose all the schedules together

```
s = allo.customize(GPT_layer)
s.compose(s_qkv)
s.compose(s_sfa)
s.compose(s_ds0)
s.compose(s_res)
s.compose(s_ln)
s.compose(s_ds1)
s.compose(s_ds2)
s.compose(s_gelu)
```

Two-Stage Generative Inference for LLM

- **Stage 1: Prefill**
  - Take in user prompts and generate the 1st token
  - seq_len > 1
  - GEMM

- **Stage 2: Decode**
  - Take in previous generated token and generate new tokens one at a time in an auto-regressive way
  - seq_len = 1
  - GEMV



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	$X_{\text{sm}}V$	l^2d	$(l+1)d$
Projection	p	$X_{\text{sdp}}W_{\text{Proj}}$	ld^2	d^2
FFN$_1$	f_1	$X_{\text{mha}}W_{\text{FFN}_1}$	ldd_{FFN}	dd_{FFN}
FFN$_2$	f_2	$X_{\text{act}}W_{\text{FFN}_2}$	ldd_{FFN}	dd_{FFN}

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

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

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

$$S_{\text{param}} C < \text{DRAM}_{\text{tot}},$$

$$\sum S_i C < \text{SRAM}_{\text{tot}}, i \in \{\text{tile,KV,FIFO}\}$$

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

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

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

- **Memory bandwidth**: B is bandwidth

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

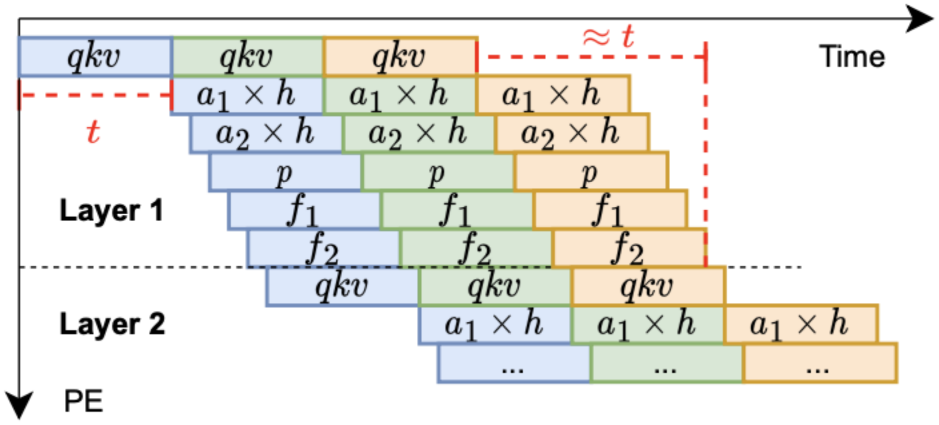


Figure : Pipeline diagram. Different colors stand for different input samples. Different blocks stand for different linear operators which also constitute the pipeline stages. h is the number of attention heads.

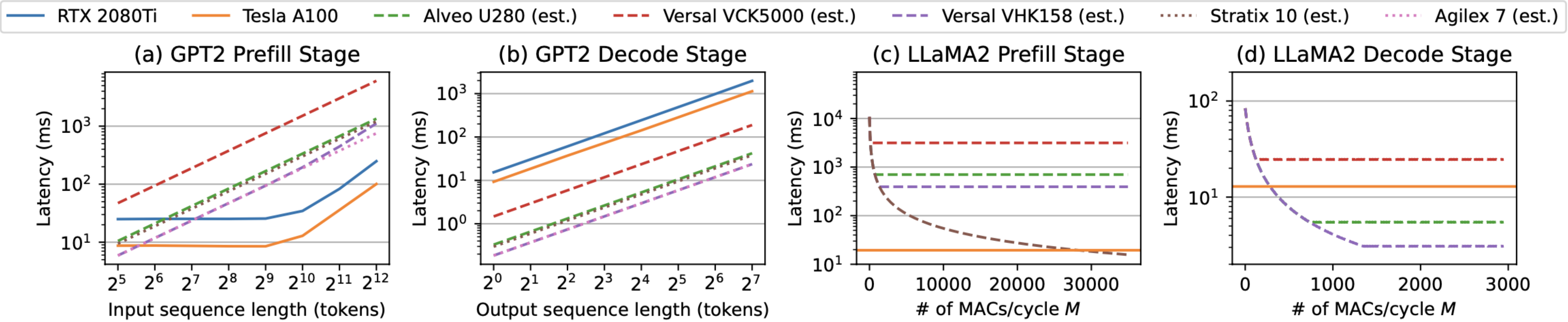
$$T_{\text{prefill}} = \frac{1}{freq}\frac{N}{C}\left(\frac{ld^2}{M_k} + C\max\left(\frac{ld^2}{M_k}, \frac{l^2d}{M_{a_1}}, \frac{ldd_{\text{FFN}}}{M_{f_1}}, T_{\text{mem}}\right)\right)$$

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

* 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”, TRETS, 2024. (FCCM’24 Journal Track)

Analytical Model for LLMs

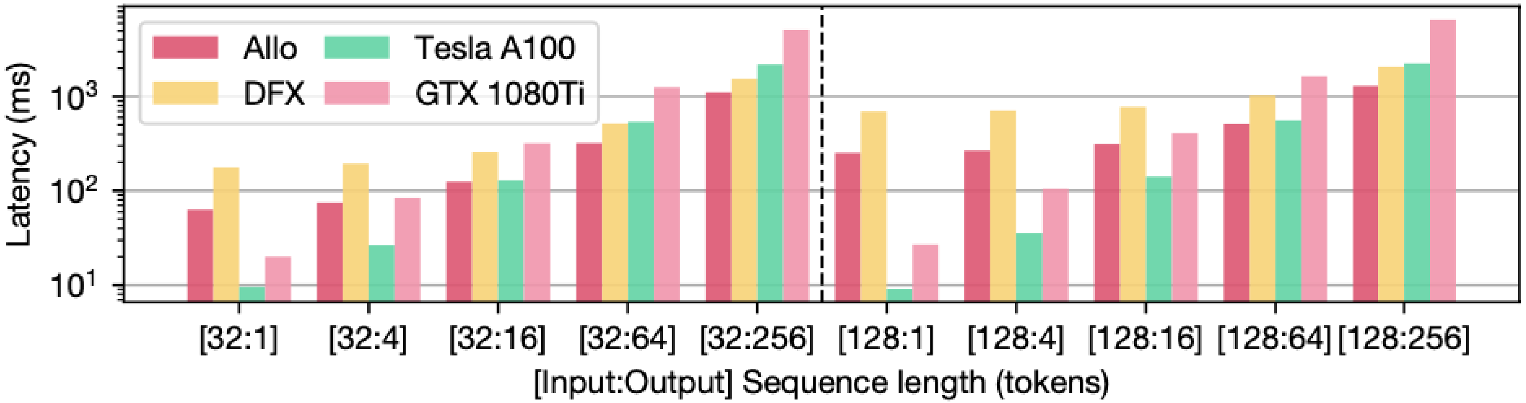
	AMD Xilinx FPGA			Intel FPGA		Nvidia GPU	
	Alveo U280 [83]	Versal VCK5000 [86]	Versal VHK158 [90]	Stratix 10 NX 2100 [37]	Agilex 7 AGM039 [27]	GeForce RTX 2080 Ti	Tesla A100
Process Node	TSMC 16nm	TSMC 7nm	TSMC 7nm	Intel 14nm	Intel 7nm	TSMC 12nm	TSMC 7nm
Release Date	2018	2022	2023	2020	2022	2018	2021
Thermal Design Power	225W	225W	180W	225W	225W	250W	300W
Peak Throughput	24.5 INT8 TOPS	145 INT8 TOPS	56 INT8 TOPS	143 INT8 TOPS	88.6 INT8 TOPS	14.2 TFLOPS	312 TFLOPS
Specialized Blocks	-	400× AI Engine	-	3960× AI Tensor Block	-	544× Tensor Cores	432× Tensor Cores
DSP/CUDA Cores	9024	1968	7392	-	12300	4352	6912
BRAM18K/M20K	4032	967	5063	6847	18960	-	-
URAM/eSRAM	960	463	1301	2	-	-	-
On-chip Memory Capacity	41MB	24MB	63.62MB	30MB	46.25MB	5.5MB	40MB
Off-chip Memory Capacity	8GB HBM2 & 32GB DDR	16GB DDR	32GB HBM2e & 32GB DDR	16GB HBM2	32GB HBM2e	11GB DDR	80GB HBM2e
On-chip Memory Bandwidth	460GB/s & 38GB/s	102.4GB/s	819.2GB/s & 102.4GB/s	512GB/s	820GB/s	616GB/s	1935GB/s

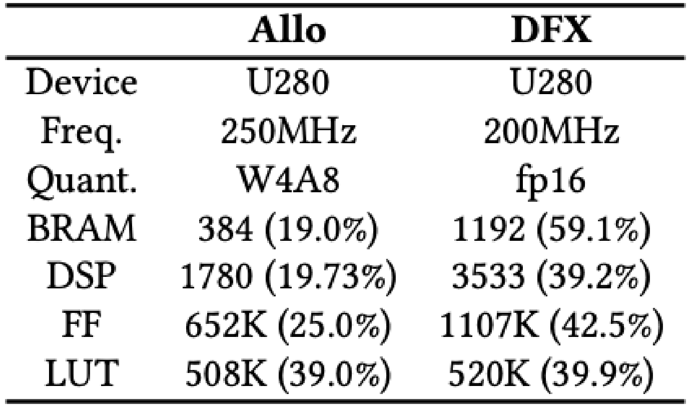


Existing FPGAs are inferior in the compute-intensive **prefill** stage
but can outperform GPUs in the memory-intensive **decode** stage.

LLM Accelerator Evaluation

- GPT2: single-batch, low-latency settings, adjust input/output token numbers
  - Use Allo to implement a design point in the analytical model (M=256)
  - 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%)

Composing Operators into Complete Design

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

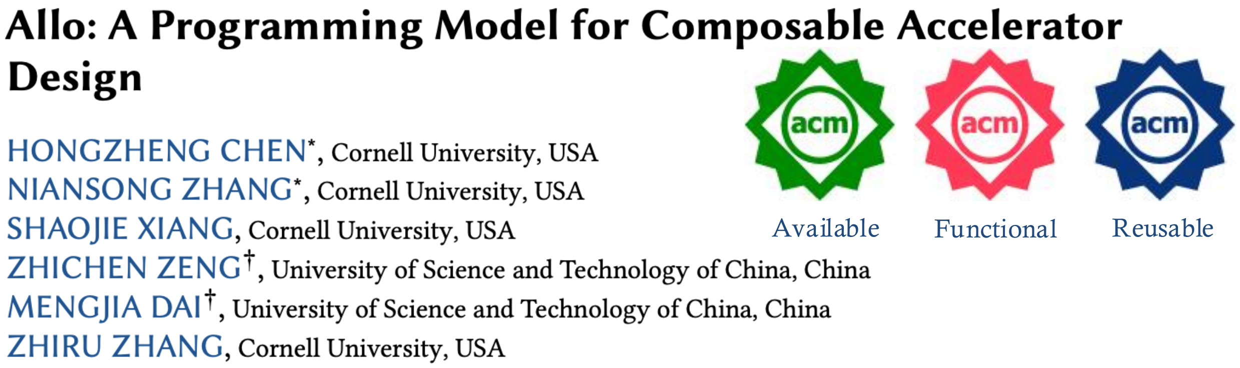
```
import torch
import allo
import numpy as np
from transformers import AutoConfig
from transformers.models.gpt2.modeling_gpt2 import GPT2Model

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

Accpeted to PLDI’24

https://arxiv.org/abs/2404.04815



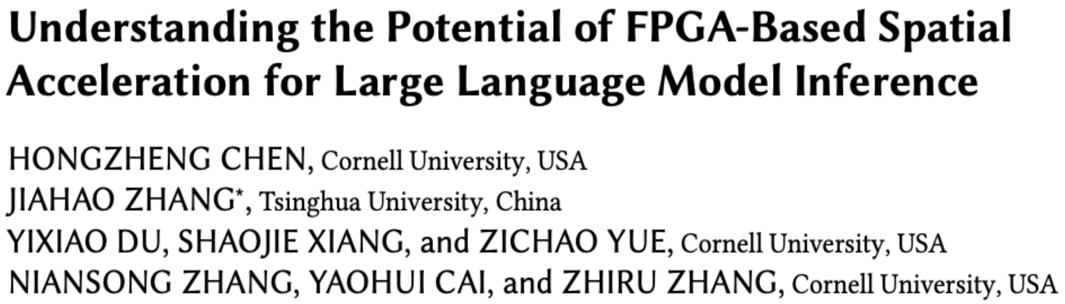
Allo: A Programming Model for Composable Accelerator Design

HONGZHENG CHEN*, Cornell University, USA
NIANSONG ZHANG*, Cornell University, USA
SHAOJIE XIANG, Cornell University, USA
ZHICHEN ZENG†, University of Science and Technology of China, China
MENGJIA DAI†, University of Science and Technology of China, China
ZHIRU ZHANG, Cornell University, USA

Available Functional Reusable

Accpeted to FCCM’24 Journal Track (TRETS)

https://arxiv.org/abs/2312.15159



Understanding the Potential of FPGA-Based Spatial Acceleration for Large Language Model Inference

HONGZHENG CHEN, Cornell University, USA
JIAHAO ZHANG*, Tsinghua University, China
YIXIAO DU, SHAOJIE XIANG, and ZICHAO YUE, Cornell University, USA
NIANSONG ZHANG, YAOHUI CAI, and ZHIRU ZHANG, Cornell University, USA

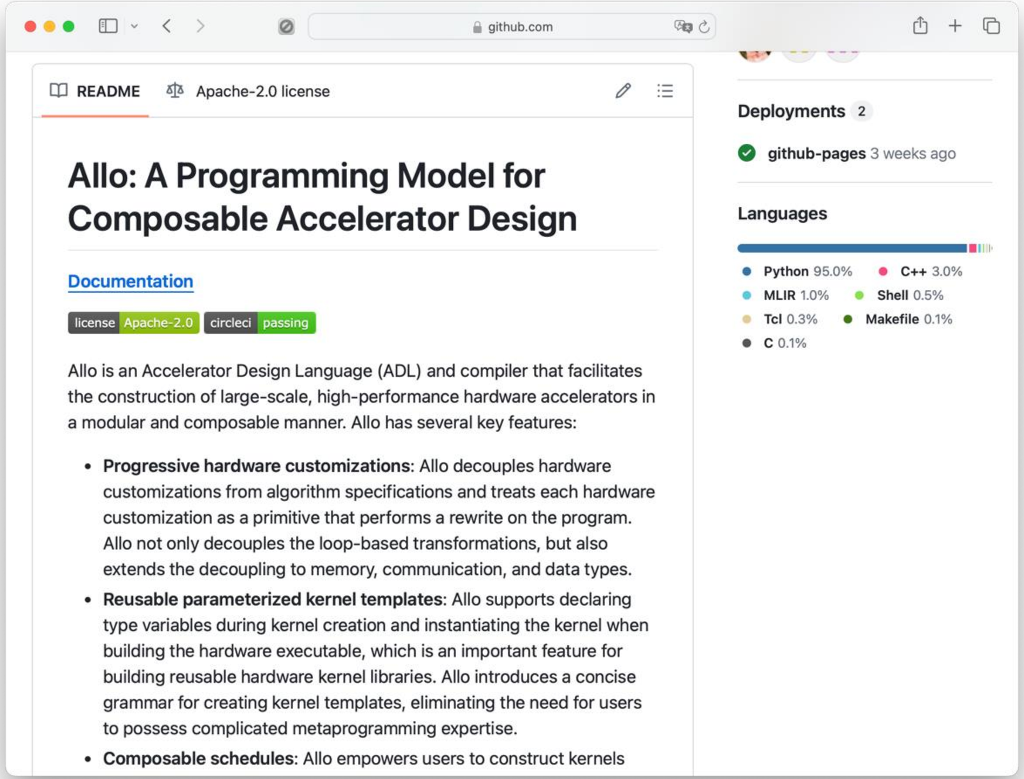
https://github.com/cornell-zhang/allo

Acknowledgements

Contributors & Collaborators

- **Cornell**: Shaojie Xiang, Andrew Butt, Matthew Hofmann, Jie Liu, Zhongyuan Zhao
- **UIUC**: Hanchen Ye
- **UCLA**: Licheng Guo, Jason Lau, Yuze Chi, Jason Cong
- **UIC**: Debjit Pal
- **CSU**: Louis-Noël Pouchet, Emily Tucker
- **AWS**: Yi-Hsiang Lai
- **Tsinghua**: Jiahao Zhang, Wenbo Zhu
- **Peking**: Haiyu Wang
- **USTC**: Zhichen Zeng, Mengjia Dai
- **Intel**: Jeremy Casas, Pasquale Cocchini, Zhenkun Yang, Jin Yang, Hongbo Rong

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



Allo: A Programming Model for Composable Accelerator Design

Documentation

license Apache-2.0 circleci passing

Allo is an Accelerator Design Language (ADL) and compiler that facilitates the construction of large-scale, high-performance hardware accelerators in a modular and composable manner. Allo has several key features:

- **Progressive hardware customizations**: Allo decouples hardware customizations from algorithm specifications and treats each hardware customization as a primitive that performs a rewrite on the program. Allo not only decouples the loop-based transformations, but also extends the decoupling to memory, communication, and data types.
- **Reusable parameterized kernel templates**: Allo supports declaring type variables during kernel creation and instantiating the kernel when building the hardware executable, which is an important feature for building reusable hardware kernel libraries. Allo introduces a concise grammar for creating kernel templates, eliminating the need for users to possess complicated metaprogramming expertise.
- **Composable schedules**: Allo empowers users to construct kernels

Deployments 2

✓ github-pages 3 weeks ago

Languages

● Python 95.0%	● C++ 3.0%
● MLIR 1.0%	● Shell 0.5%
● Tcl 0.3%	● Makefile 0.1%
● C 0.1%	

Sponsors

DARPA NSF Semiconductor Research Corporation ACE CENTER FOR EVOLVABLE COMPUTING intel AMD XILINX