Catalyzing Accelerator Design and Programming for Machine Learning

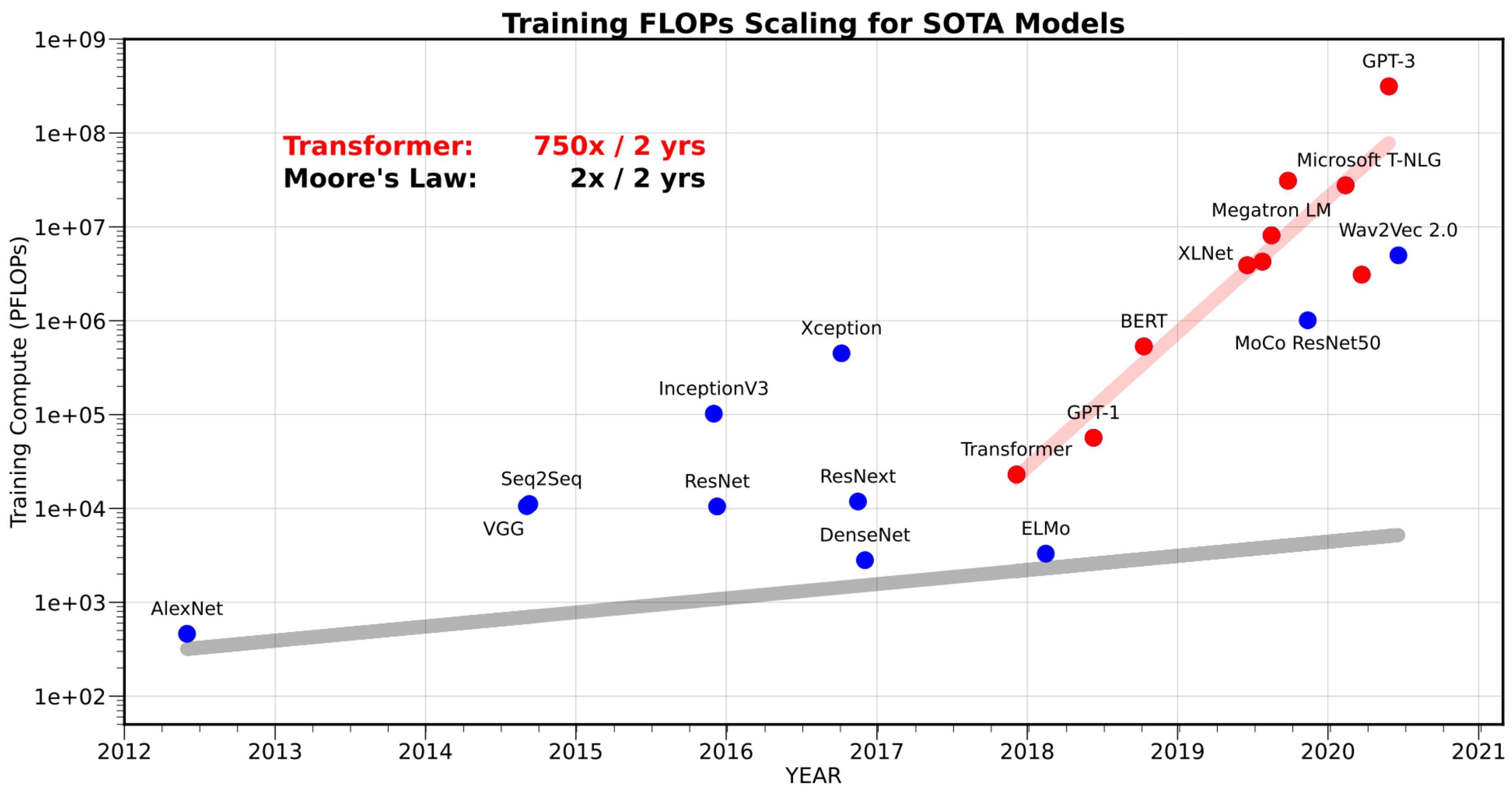
Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhiru Zhang

Cornell University

C4ML workshop @ CGO’25

Cornell University

Deep Learning Models Become Larger and Larger…

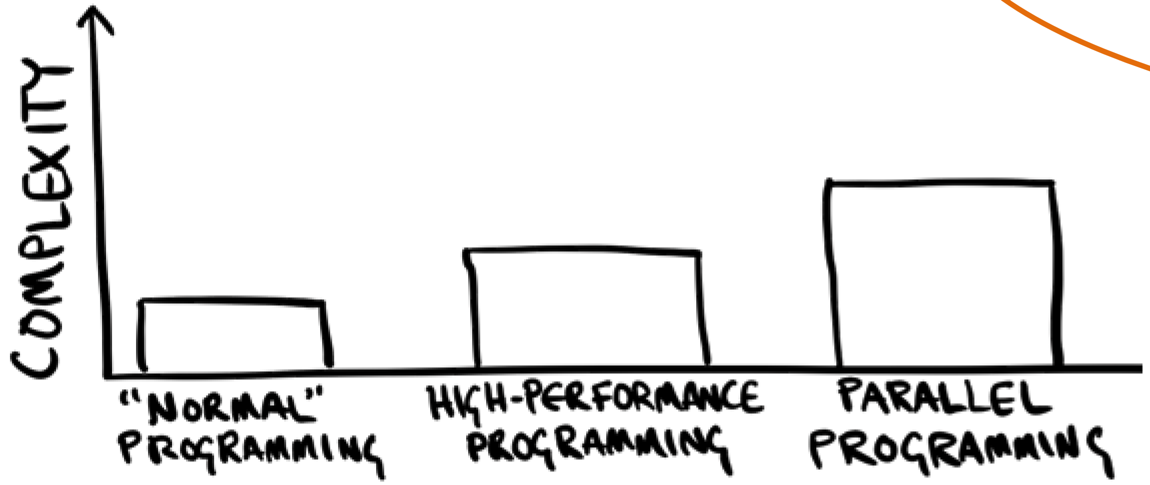


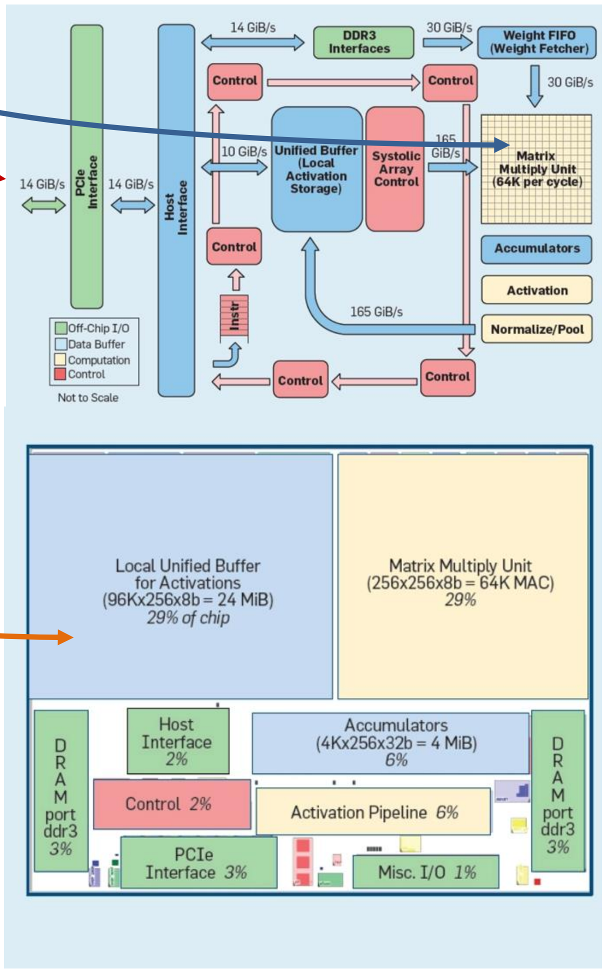
- The computational demands of large models have outgrown the capabilities of existing hardware
- Specialized accelerators are essential for speeding up model training and inference

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

Complexity in Specialized Accelerator Design

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





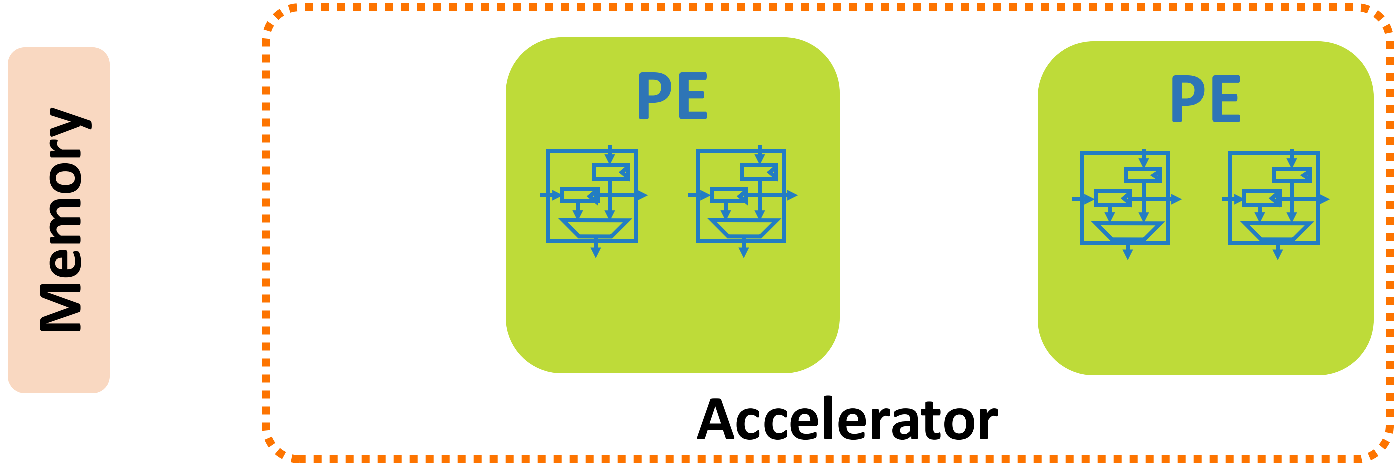
An AI accelerator example

Fig credit: Adrian Sampson, "From Hardware Description Languages to Accelerator Design Languages"

Challenge: 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
 hls::stream<int8_t> A_fifo[2][3], B_fifo[2][3];
 #pragma stream variable=A/B_fifo depth=3
 #pragma partition variable=A/B/C_tile complete dim=1
 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);
}}}

void matmul(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);
}}}
```

Compute Customization



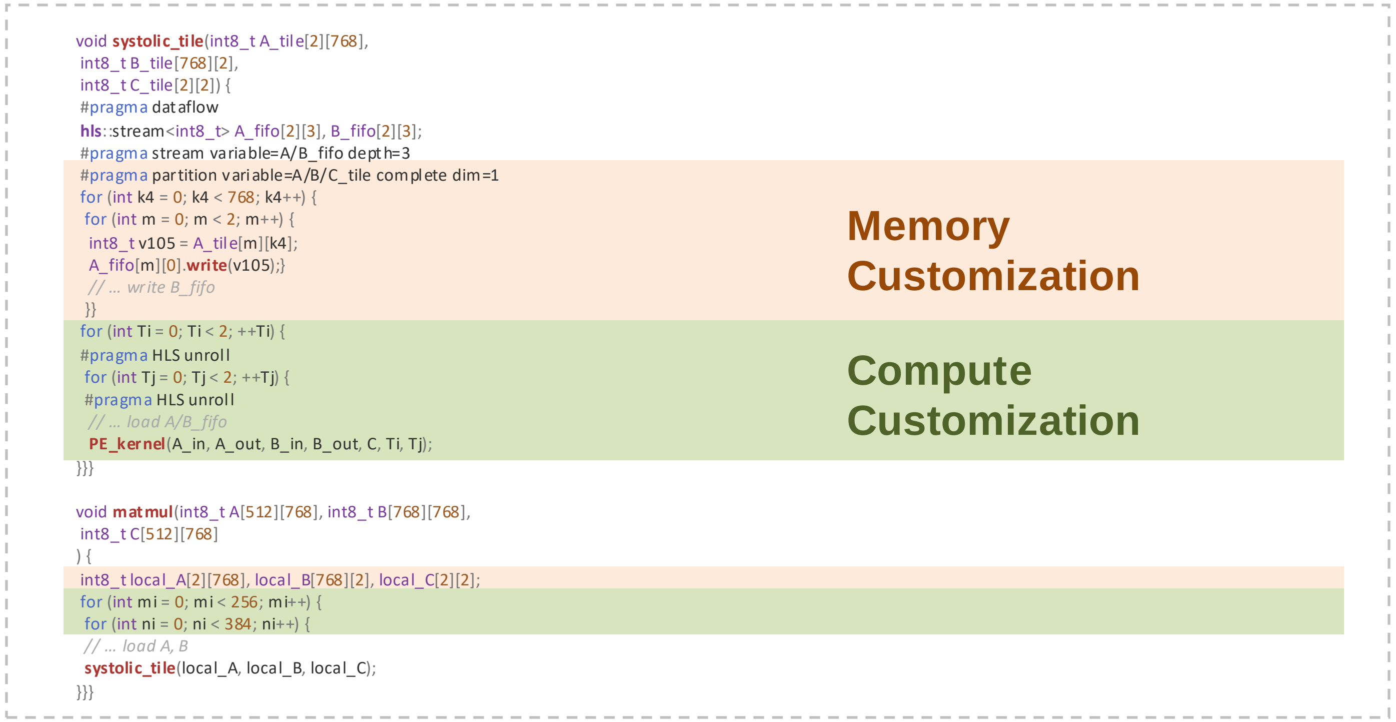
Vanilla Matmul (1% theoretical peak perf.)

+ Compute customization (~30% peak)

- + Loop tiling
- + Loop unrolling
- + Loop & function pipelining

Increasing programming effort

Challenge: Balancing Manual Control & Compiler Optimization

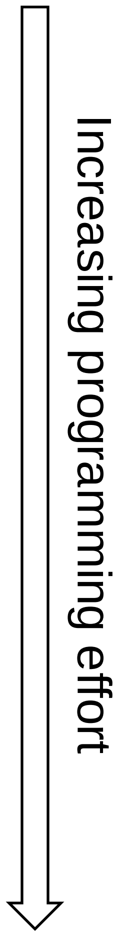


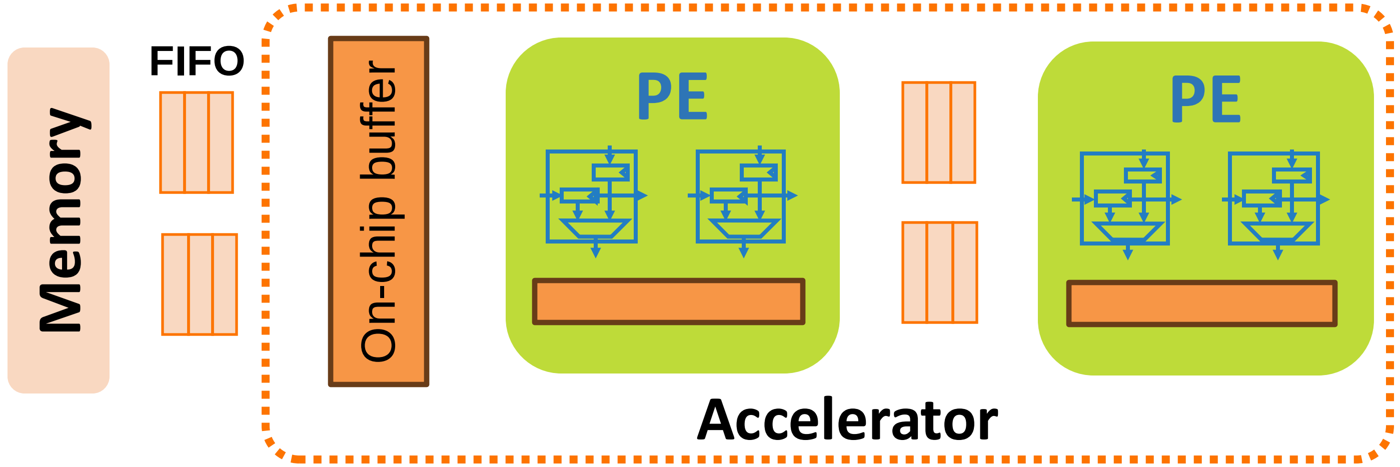
```
void systolic_tile(int8_t A_tile[2][768],
 int8_t B_tile[768][2],
 int8_t C_tile[2][2]) {
 #pragma dataflow
 hls::stream<int8_t> A_fifo[2][3], B_fifo[2][3];
 #pragma stream variable=A/B_fifo depth=3
 #pragma partition variable=A/B/C_tile complete dim=1
 for (int k4 = 0; k4 < 768; k4++) {
  for (int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

void matmul(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 Matmul (1% theoretical peak perf.)

+ Compute customization (~30% peak)
    + Loop tiling
    + Loop unrolling
    + Loop & function pipelining

+ Custom memory hierarchy (~50% peak)
    + Tiling & data reuse buffers
    + Memory banking/partitioning





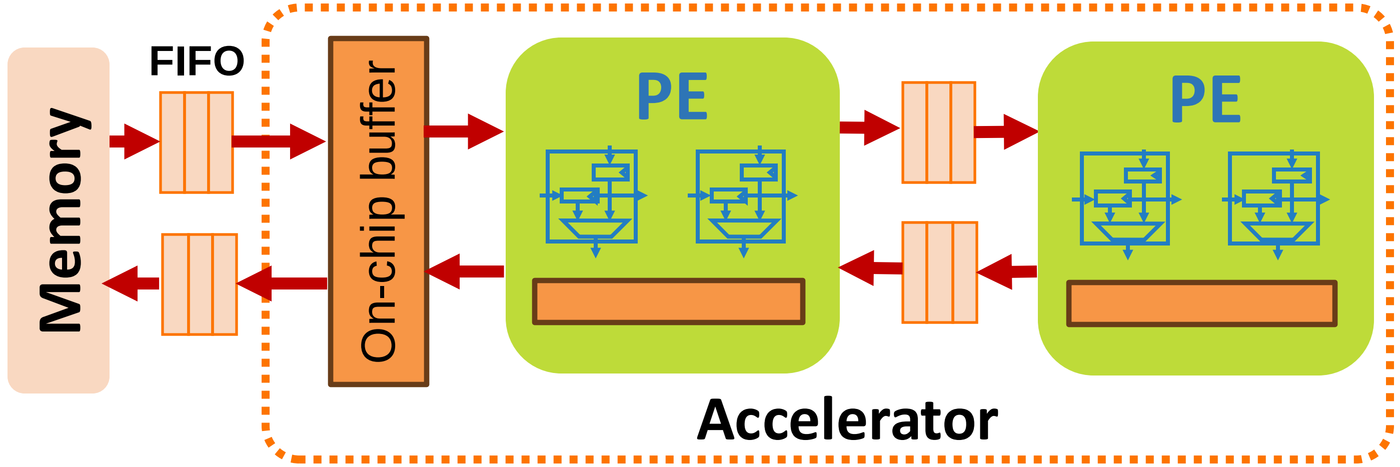
Challenge: Balancing Manual Control & Compiler Optimization

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

void matmul(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);
}}}
```

Communication Customization

Memory Customization

Compute Customization



Vanilla Matmul (1% theoretical peak perf.)

+ Compute customization (~30% peak)
  + Loop tiling
  + Loop unrolling
  + Loop & function pipelining

+ Custom memory hierarchy (~50% peak)
  + Tiling & data reuse buffers
  + Memory banking/partitioning

+ Data movement optimization (~95% peak)
  + Data streaming
  + Data packing (vectorization)
  + Memory coalescing
  + Systolic communication

Increasing programming effort

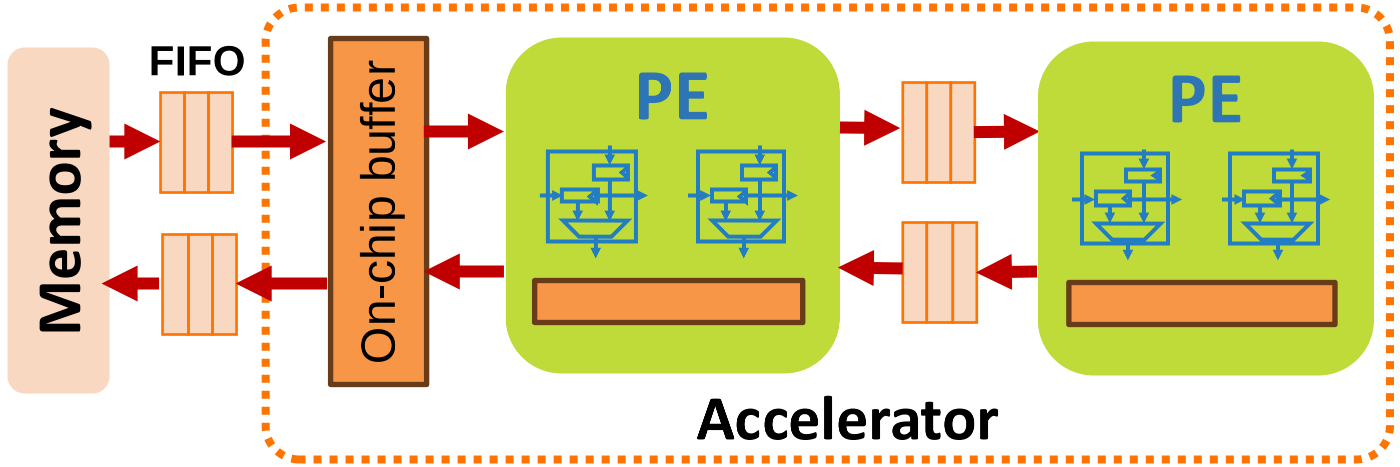
Challenge: Balancing Manual Control & Compiler Optimization

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

void matmul(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);
}}}
```

Communication Customization

Memory Customization

Compute Customization



Vanilla Matmul (1% theoretical peak perf.)

+ Compute customization (~30% peak)
  + Loop tiling
  + Loop unrolling
  + Loop & function pipelining

Existing HLS compiler e.g., ScaleHLS [HPCA'22]

+ Custom memory hierarchy (~50% peak)
  + Tiling & data reuse buffers
  + Memory banking/partitioning

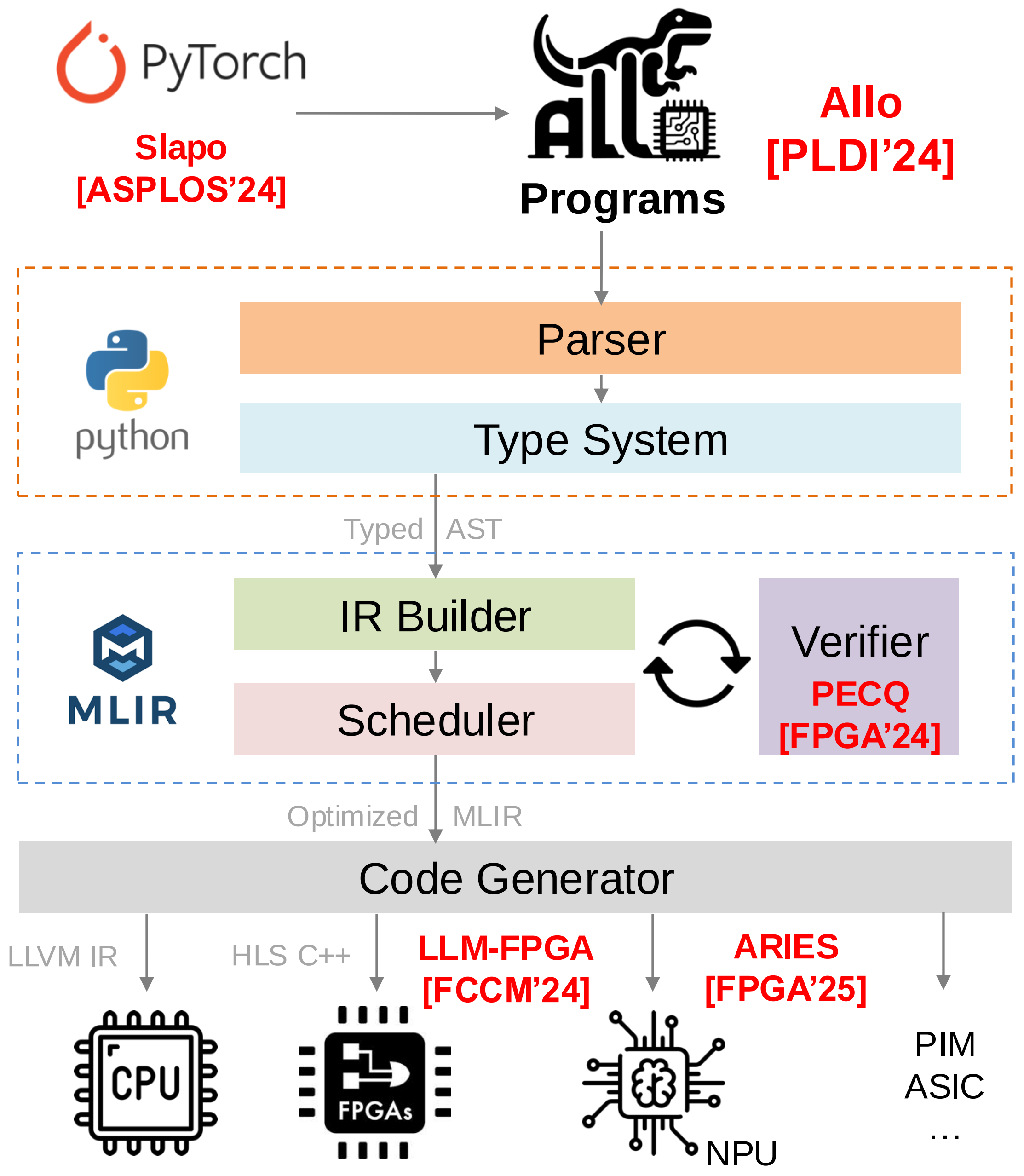
+ Data movement optimization (~95% peak)
  + Data streaming
  + Data packing (vectorization)
  + Memory coalescing
  + Systolic communication

Increasing programming effort

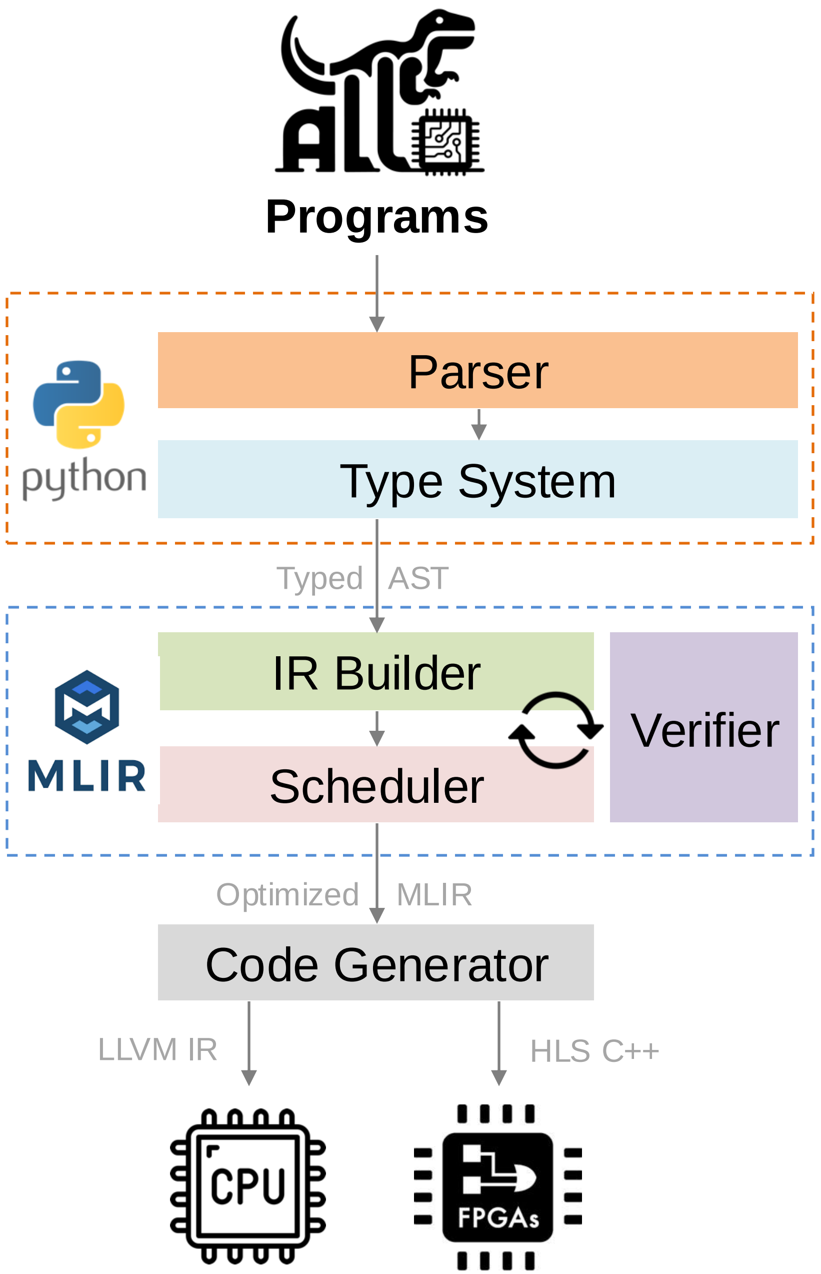
~500 lines of HLS code for a GEMM systolic array

vendor-specific, hard to maintain & reuse

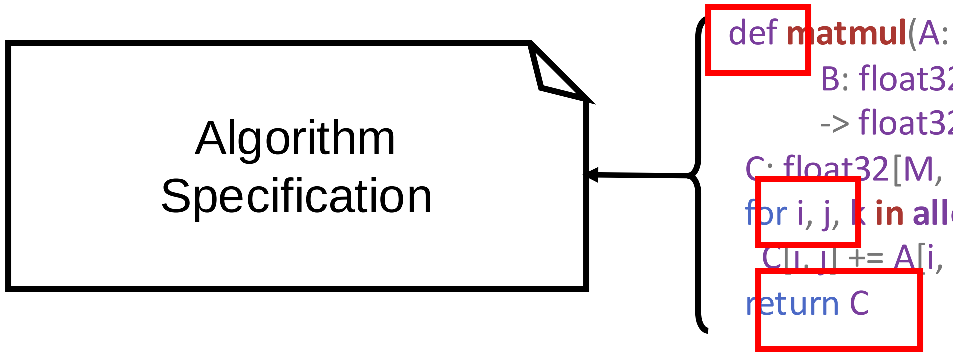
Allo Accelerator Design Language (ADL) and Compiler



Allo Accelerator Design Language (ADL) and Compiler



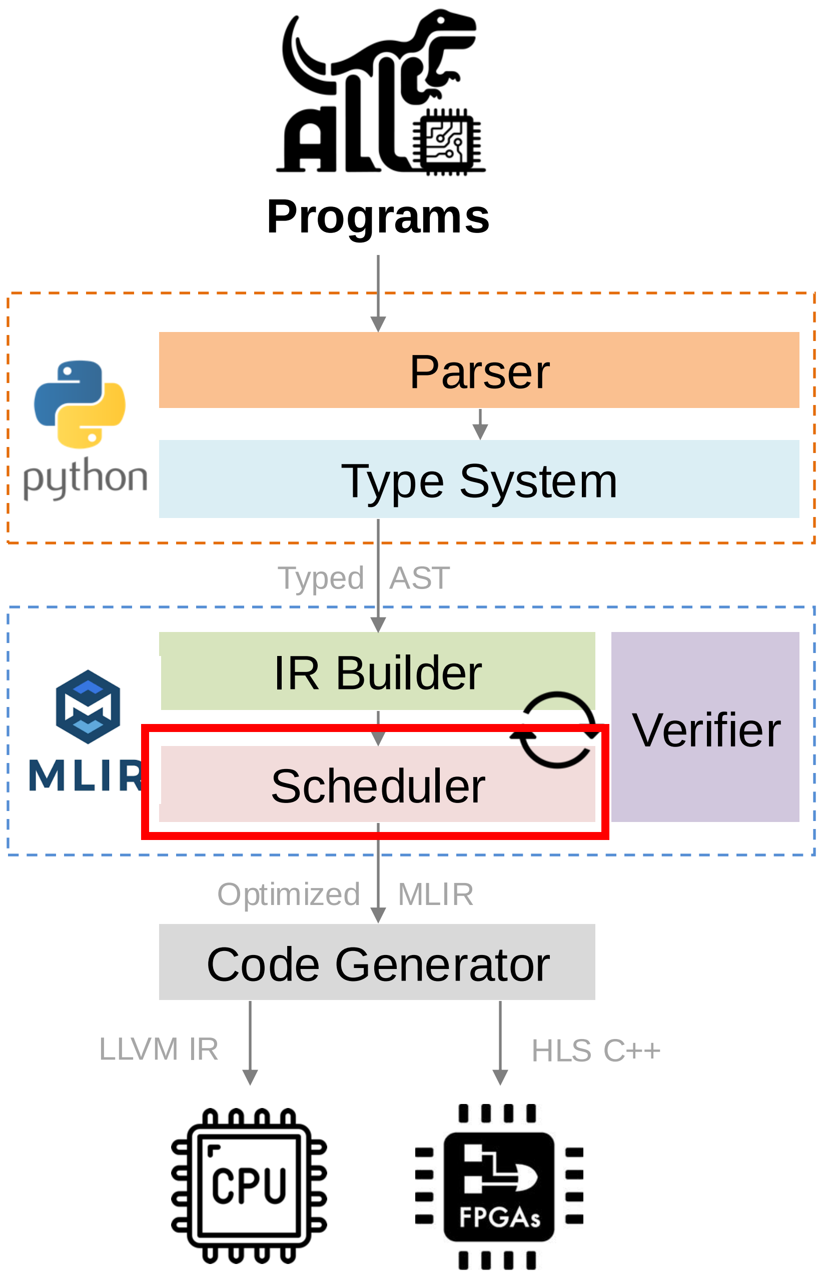
Pythonic: No need to learn a new DSL!



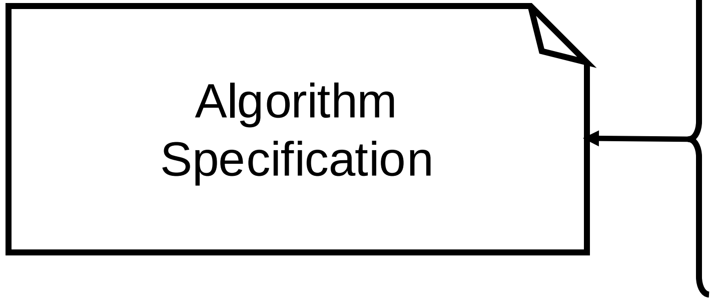
```
def matmul(A: float32[M, K],
           B: float32[K, N])
           -> float32[M, N]:
  C: float32[M, N] = 0.0
  for i, j, k in allo.grid(M, N, K):
    C[i, j] += A[i, k] * B[k, j]
  return C
```

- Free-form imperative programming
- Strong type system

Allo Accelerator Design Language (ADL) and Compiler

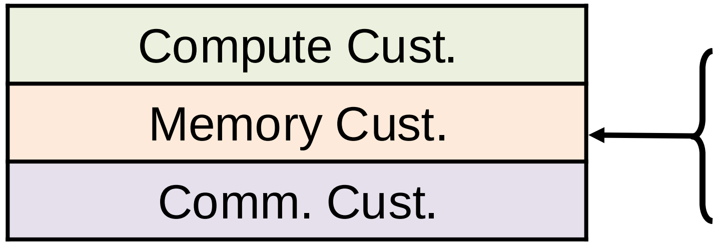


Pythonic: No need to learn a new DSL!



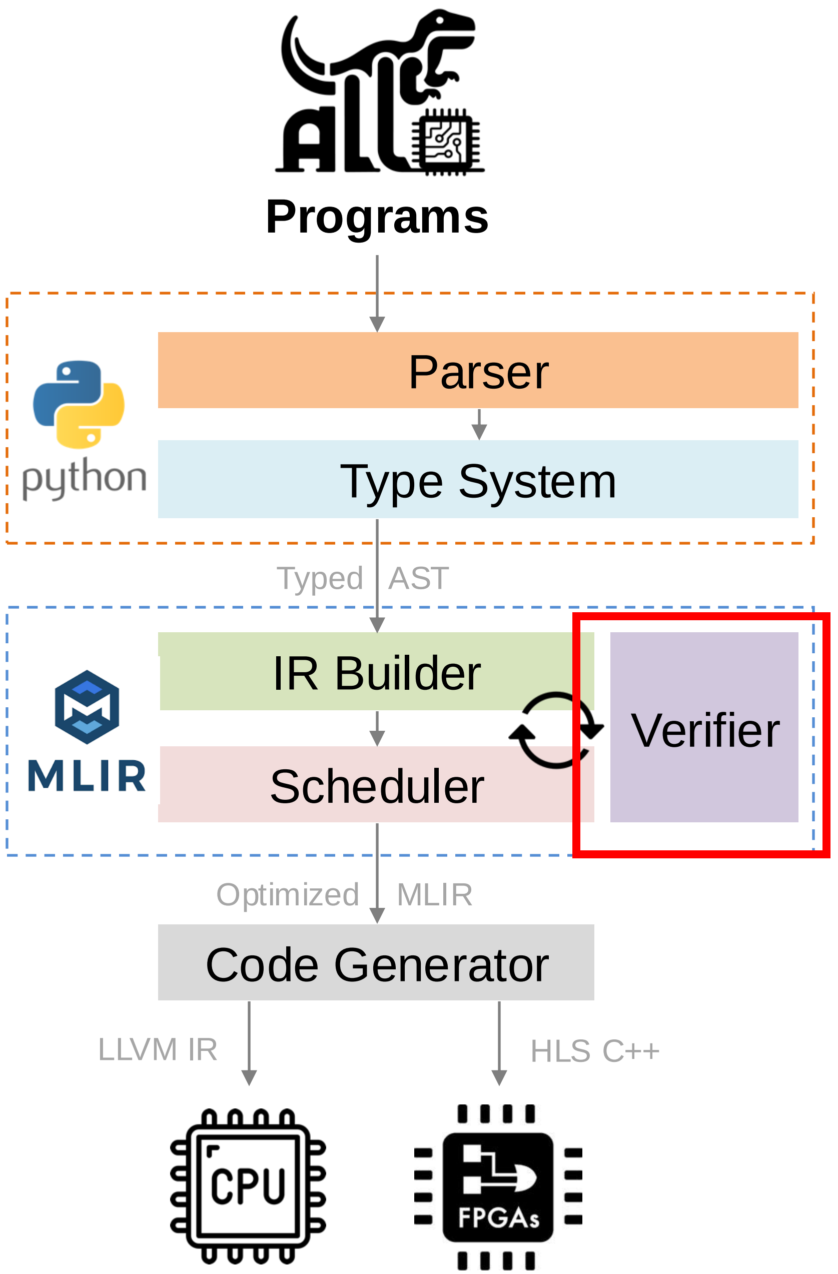
```
def matmul(A: float32[M, K],
           B: float32[K, N])
           -> float32[M, N]:
  C: float32[M, N] = 0.0
  for i, j, k in allo.grid(M, N, K):
    C[i, j] += A[i, k] * B[k, j]
  return C
```

Decoupled customization



```
s = allo.customize(matmul)
s.reorder("k", "j")
s.buffer_at(s.C, axis="i")
s.pipeline("j")
```

Allo Accelerator Design Language (ADL) and Compiler



Pythonic: No need to learn a new DSL!

Algorithm Specification

```
def matmul(A: float32[M, K],
           B: float32[K, N])
           -> float32[M, N]:
  C: float32[M, N] = 0.0
  for i, j, k in allo.grid(M, N, K):
    C[i, j] += A[i, k] * B[k, j]
  return C
```

Stepwise verifiable rewrites

```
s = allo.customize(gemm)
s.reorder("k", "j")
print(s.module)
```

```
module {
func.func @gemm(%arg0: memref<1024x1024xf32>, %arg1: memref<1024x1024xf32>) ->
memref<1024x1024xf32> {
 %alloc = memref.alloc() {name = "C"} : memref<1024x1024xf32>
 affine.for %arg2 = 0 to 1024 {
  affine.for %arg3 = 0 to 1024 {
   affine.for %arg4 = 0 to 1024 {
    ...
   } {loop_name = "k"}
  } {loop_name = "j"}
 } {loop_name = "i", op_name = "S_i_j_0"}
 return %alloc : memref<1024x1024xf32>
}}
```

```
s.buffer_at(s.C, axis="i")
print(s.module)
```

MLIR

```
module {
func.func @gemm(%arg0: memref<1024x1024xf32>, %arg1: memref<1024x1024xf32>) ->
memref<1024x1024xf32> {
 %alloc = memref.alloc() {name = "C"} : memref<1024x1024xf32>
 affine.for %arg2 = 0 to 1024 {
  %alloc_0 = memref.alloc() : memref<1024xf32>
  affine.for %arg3 = 0 to 1024 {
   ...
  } {buffer, loop_name = "j_init", pipeline_ii = 1 : i32}
  affine.for %arg3 = 0 to 1024 {
   affine.for %arg4 = 0 to 1024 {
    ...
   } {loop_name = "j"}
  } {loop_name = "k", op_name = "S_k_0", reduction}
  affine.for %arg3 = 0 to 1024 {
   ...
  } {buffer, loop_name = "j_back", pipeline_ii = 1 : i32}
 } {loop_name = "i", op_name = "S_i_j_0"}
 return %alloc : memref<1024x1024xf32>
}}
```

```
s.pipeline("j")
print(s.module)
```

```
module {
func.func @gemm(%arg0, %arg1) -> memref<1024x1024xf32> {
 %alloc = memref.alloc() {name = "C"} : memref<1024x1024xf32>
 affine.for %arg2 = 0 to 1024 {
  affine.for %arg3 = 0 to 1024 {
   affine.for %arg4 = 0 to 1024 {
    ...
   } {loop_name = "j", pipeline}
  } {loop_name = "k", op_name = "S_k_0", reduction}
  ...
 } {loop_name = "i", op_name = "S_i_j_0"}
 return %alloc : memref<1024x1024xf32>
}}
```

Transforming GEMM to Systolic Array

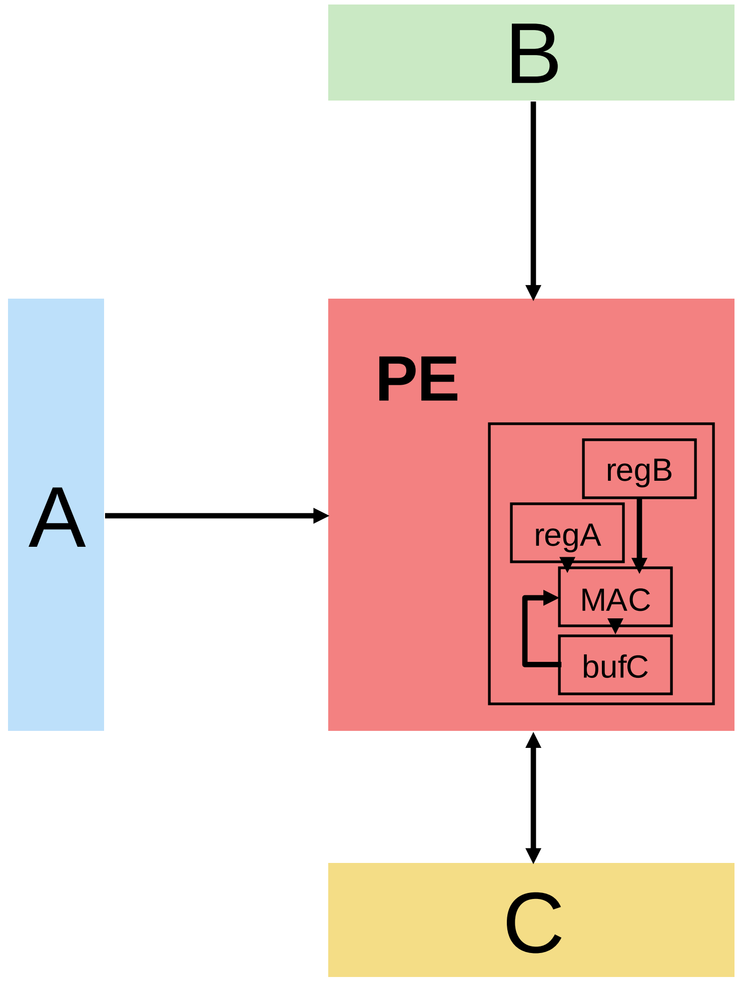
➔ Algorithm specification

```
def matmul(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(matmul)
```

➔ Architectural Diagram



* Schedule: A sequence of customization primitives

Transforming GEMM to Systolic Array

➔ Algorithm specification

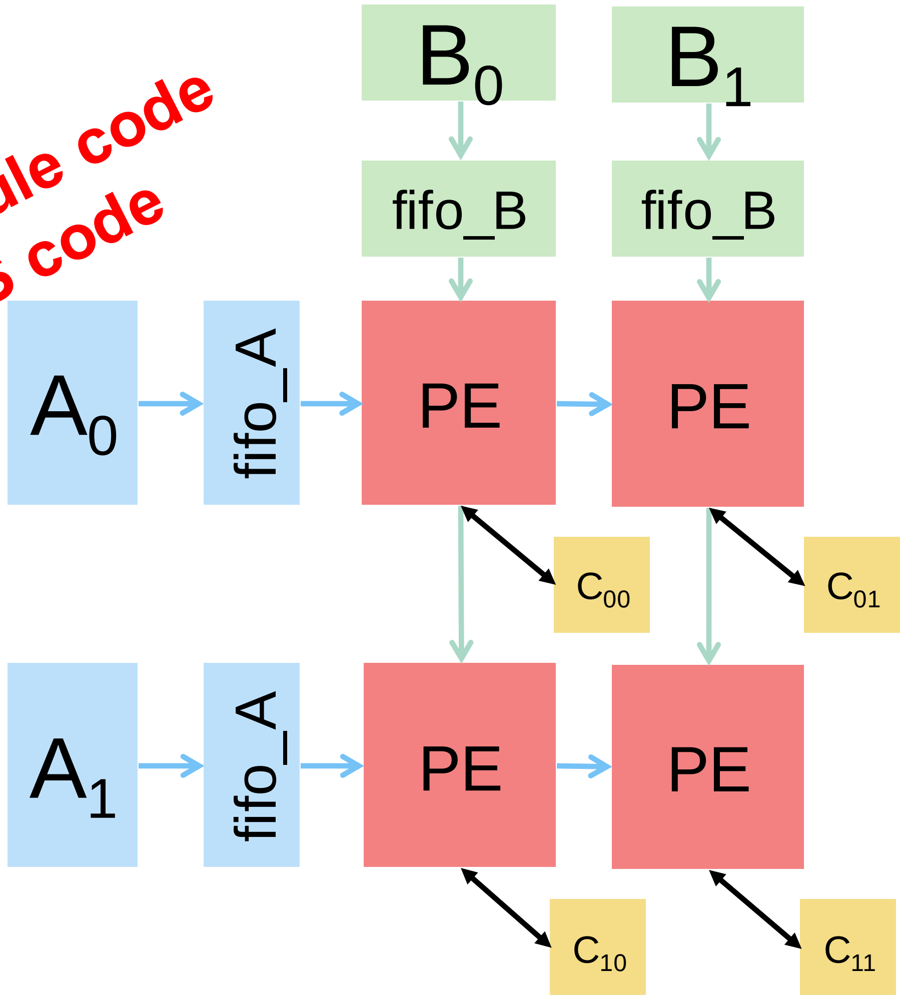
```
def matmul(A: int8[M, K],
           B: int8[K, N],
           C: int16[M, N]):
  for i, j in allo.grid(M, N, "PE"):
    for k in range(K):
      C[i, j] += A[i, k] * B[k, j]
```

➔ Schedule construction

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

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

➔ Architectural Diagram



* Schedule: A sequence of customization primitives

Transforming GEMM to Systolic Array

➔ Algorithm specification

```
def matmul(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(matmul)
buf_A = s.buffer_at(s.A, "j")
buf_B = s.buffer_at(s.B, "j")
pe = s.unfold("PE", axis=[0, 1],
              factor=[M, N])
s.partition(s.A, dim=0)
s.partition(s.B, dim=1)
s.partition(s.C, dim=[0, 1])
s.to(buf_A, pe, axis=0, depth=M + 1)
s.to(buf_B, pe, axis=1, 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)
```

➔ Bitstream generation

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

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

Transforming GEMM to Systolic Array

➔ Algorithm specification

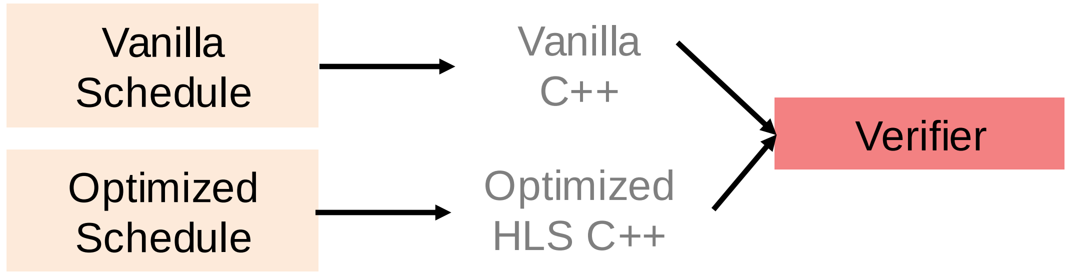
```
def matmul(A: int8[M, K],
           B: int8[K, N],
           C: int16[M, N]):
  for i, j in allo.grid(M, N, "PE"):
    for k in range(K):
      C[i, j] += A[i, k] * B[k, j]
```

➔ Schedule construction

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

➔ **Formal Verification** [FPGA'24 Best Paper*]

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



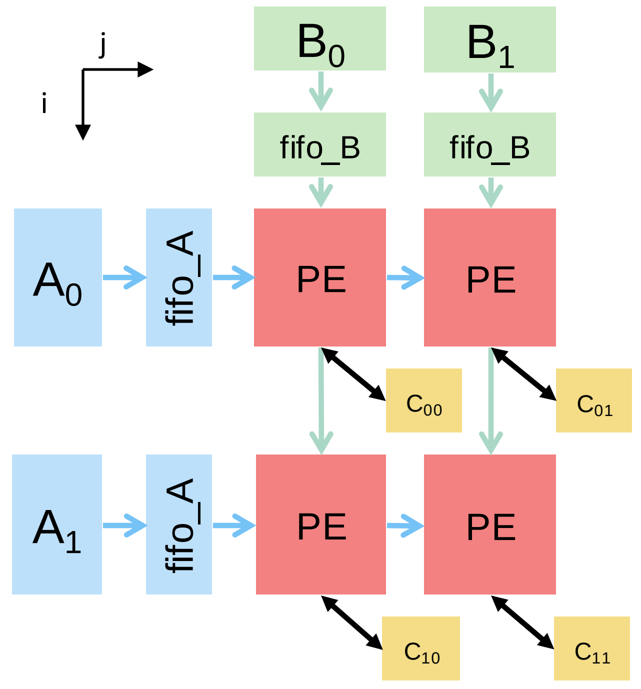
- A **formal equivalence checker** of source-to-source HLS transformations via symbolic execution
  - Support statically interpretable control-flow (SICF)
- Verification in time/space linear w.r.t. OPs executed
  - ~500K Ops/sec in verification throughput
  - A complex **64x64 systolic array verified in 16 minutes**

* L.-N. Pouchet et al., "Formal Verification of Source-to-Source Transformation for HLS", FPGA 2024 (Best Paper Award)

Transforming GEMM to Systolic Array

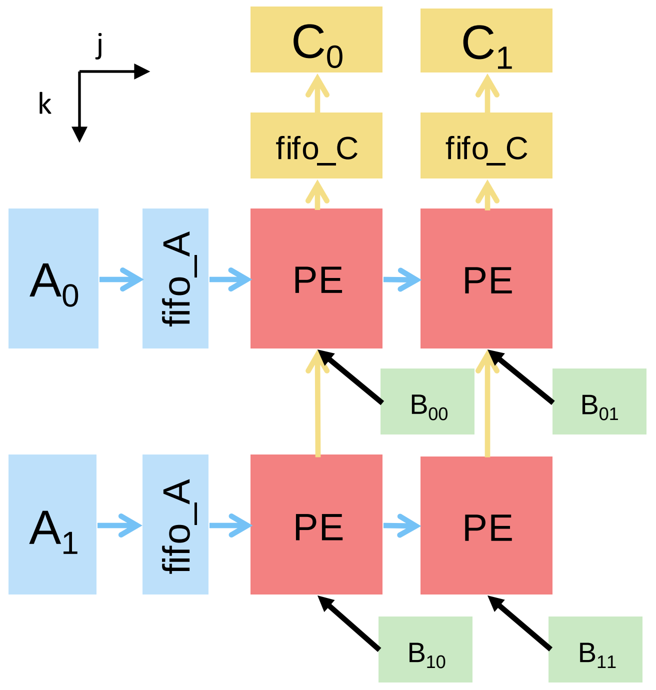
➔ Spatial loop: i, j

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



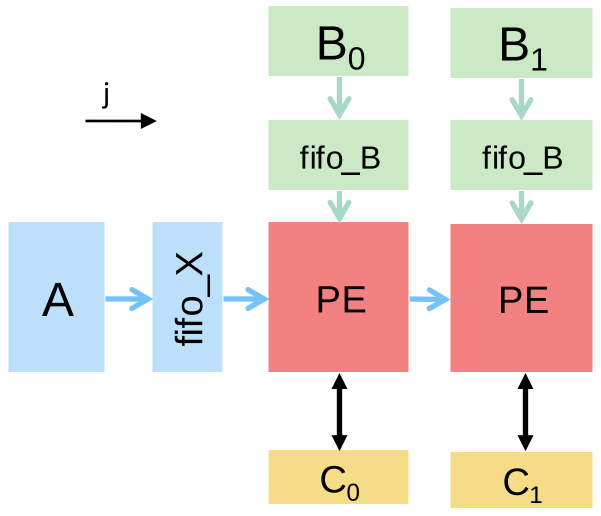
➔ Spatial loop: k, j

```
s = allo.customize(matmul)
s.reorder("k", "j", "i")
buf_A = s.buffer_at(s.A, "j")
buf_C = s.buffer_at(s.C, "j")
pe = s.unfold("PE", axis=[0, 1],
              factor=[K, N])
s.partition(s.A, dim=0)
s.partition(s.B, dim=[0, 1])
s.partition(s.C, dim=1)
s.to(buf_A, pe, axis=0)
s.to(buf_B, pe, axis=1)
```



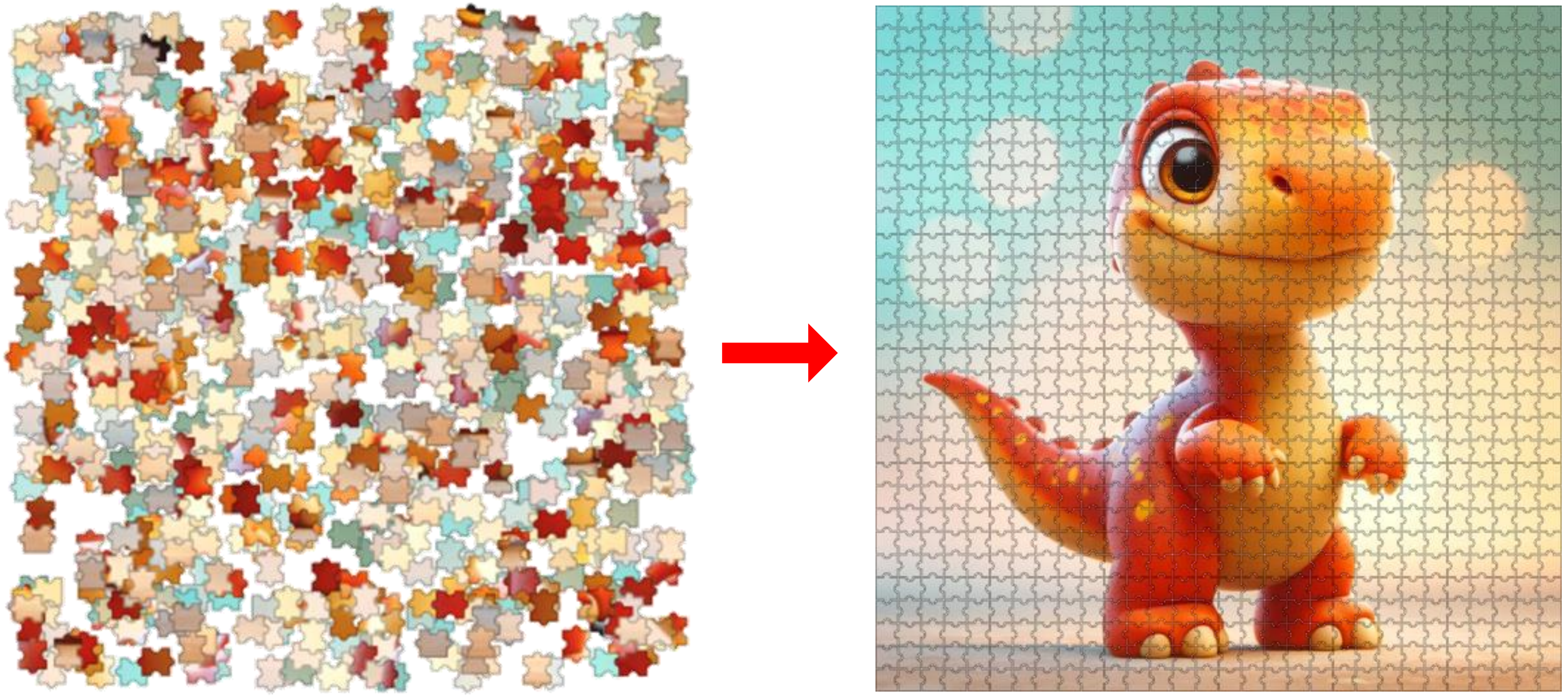
➔ Spatial loop: j

```
s = allo.customize(matmul)
buf_A = s.buffer_at(s.A, "j")
buf_B = s.buffer_at(s.B, "j")
pe = s.unfold("PE", axis=1,
              factor=M)
s.partition(s.B, dim=1)
s.partition(s.C, dim=1)
s.to(buf_A, pe, axis=0)
s.to(buf_B, pe, axis=1)
```



Realize different dataflows with minimal schedule code

How to Compose Optimized Kernels into Complete Accelerator?



Optimized Kernels

High-performance Accelerator

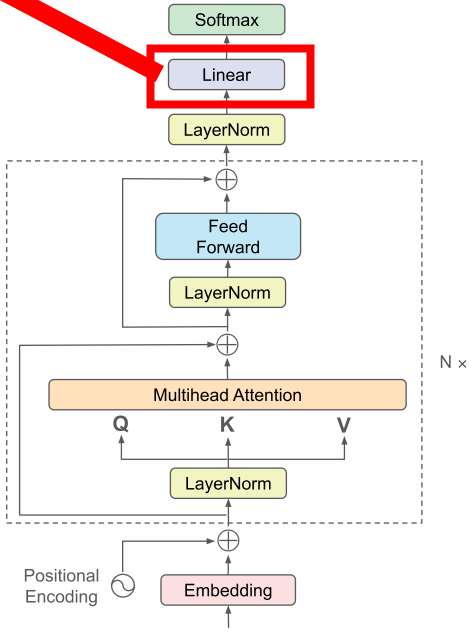
Composable Schedules

➔ Given different optimized kernel implementations

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

➔ Algorithm specification (Hierarchical)

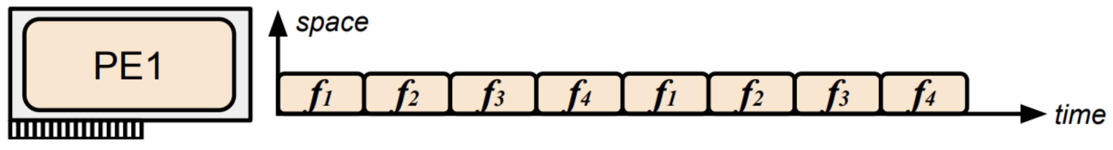
```
def top(X: int8[M, K],
        W_A: int8[K, N],
        W_B: int8[N, K]):
  Y: int8[M, N] = 0
  Z: int8[M, K] = 0
  matmul(X, W_A, Y)
  matmul(Y, W_B, Z)
  return Z
```



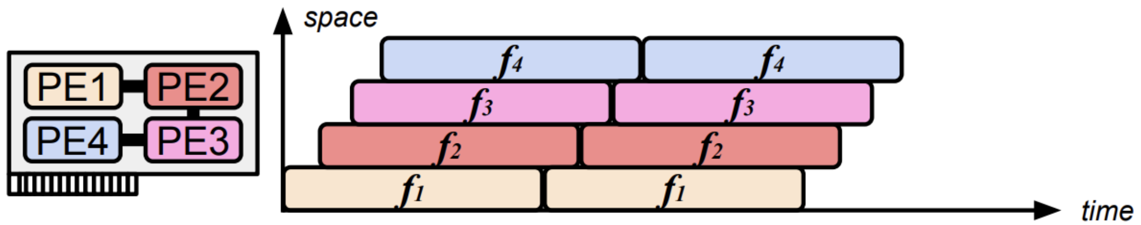
➔ Schedule composition

```
# Previous customizations for matmul
s_matmul = allo.customize(matmul)
# ...

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

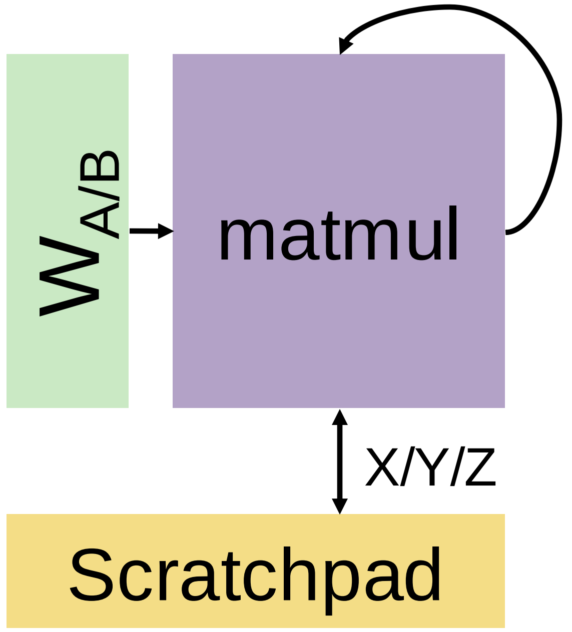
Interconnection between Kernels



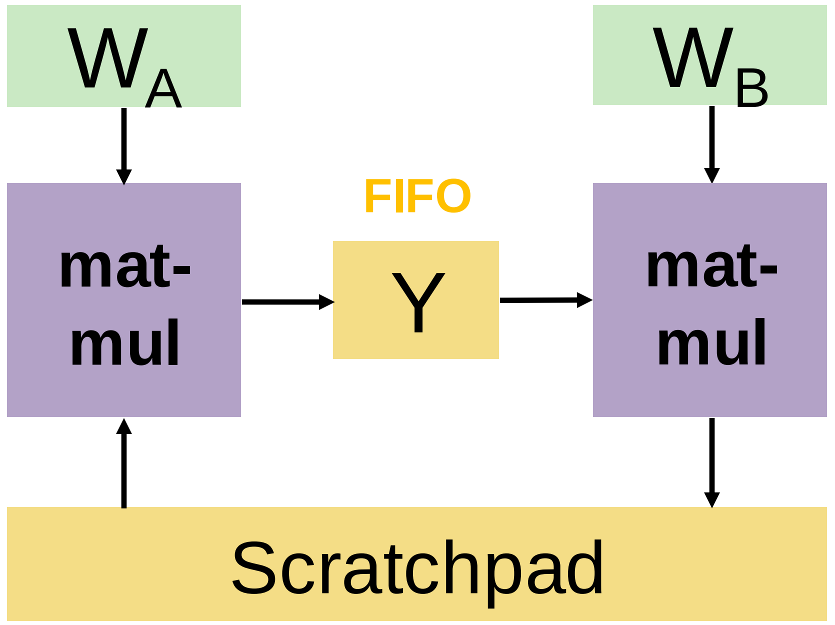
Temporal Composition



Spatial Composition



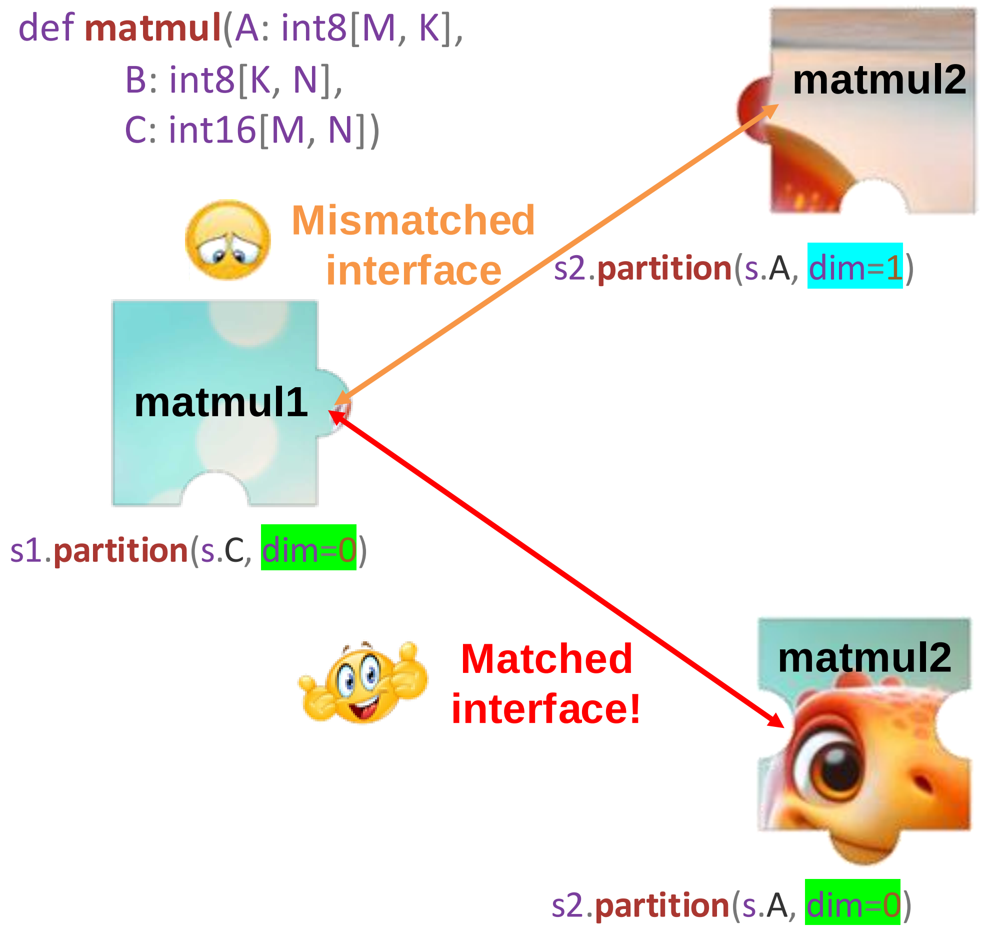
```
def top(
  X: int8[M, K],
  W_A: int8[K, N],
  W_B: int8[N, K]):
  Y: ?
  Z: int8[M, K] = 0
  matmul(X, W_A, Y)
  matmul(Y, W_B, Z)
  return Z
```



Temporal Composition

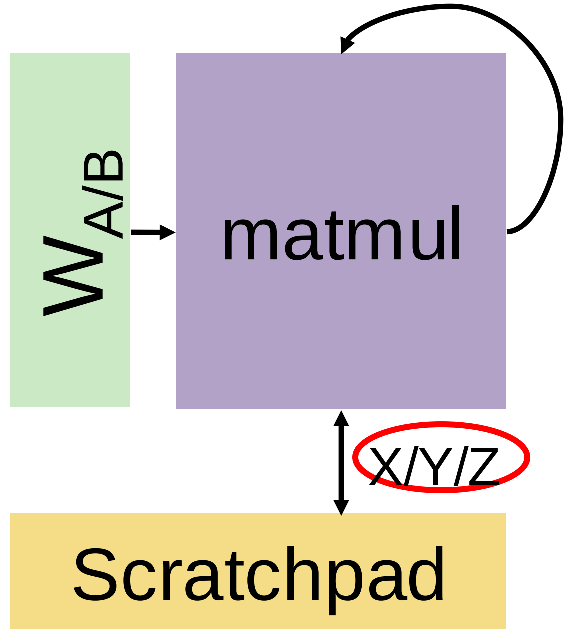
➔ Given optimized kernel implementations

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



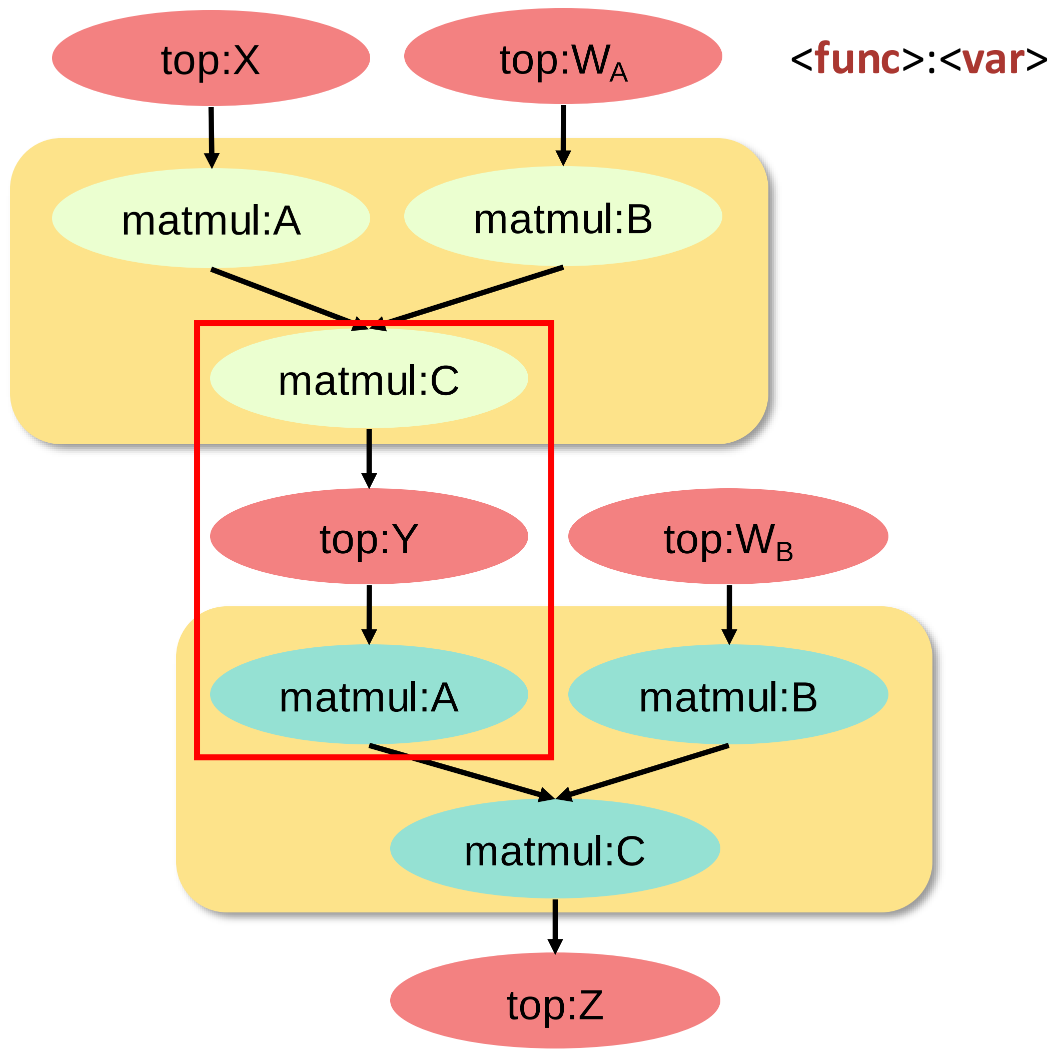
➔ Algorithm specification (Hierarchical)

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



Need to ensure interface consistency

Temporal Composition



Hierarchical use-def 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
  matmul(X, W_A, Y)
  matmul(Y, W_B, Z)
  return Z
```

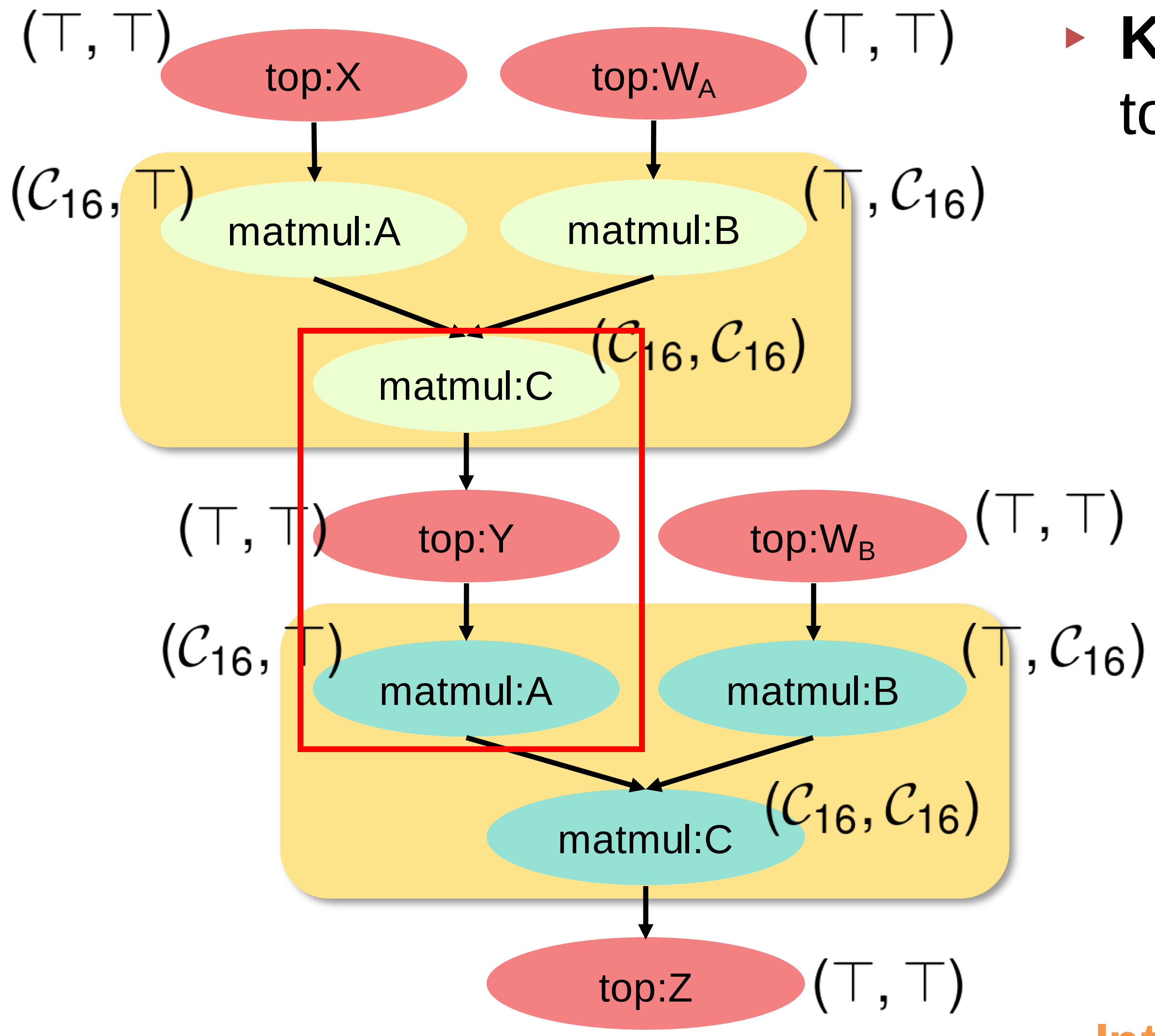
← **Caller definition**

➔ Callee definition

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

Need to ensure interface consistency

Temporal Composition

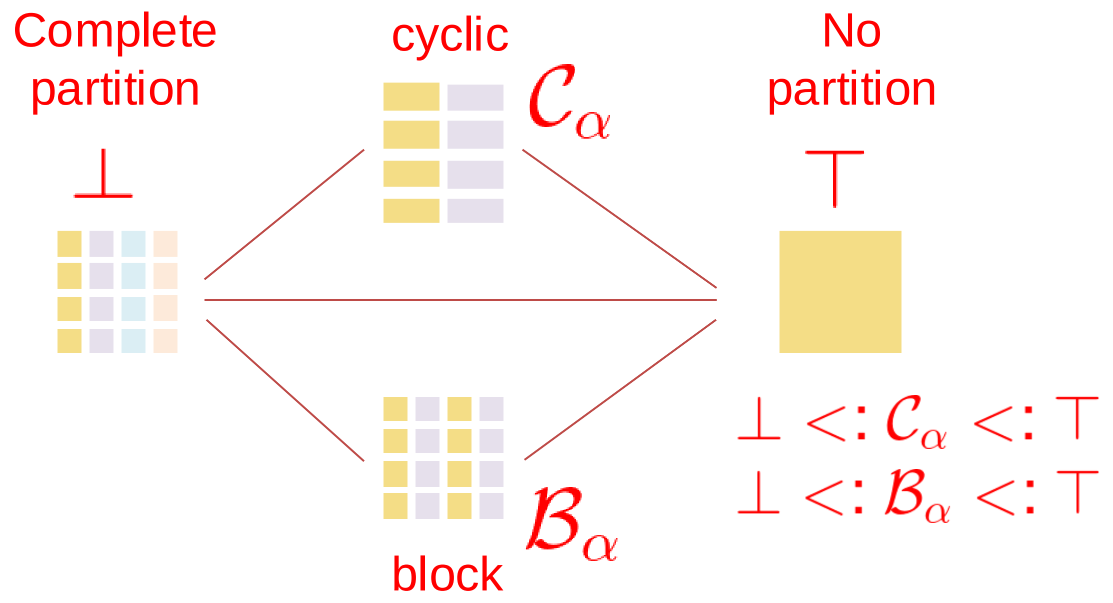


$(\top, \top)$
top:X
top:W_A
$(\top, \top)$
$(\mathcal{C}_{16}, \top)$
matmul:A
matmul:B
$(\top, \mathcal{C}_{16})$
matmul:C
$(\mathcal{C}_{16}, \mathcal{C}_{16})$
$(\top, \top)$
top:Y
top:W_B
$(\top, \top)$
$(\mathcal{C}_{16}, \top)$
matmul:A
matmul:B
$(\top, \mathcal{C}_{16})$
matmul:C
$(\mathcal{C}_{16}, \mathcal{C}_{16})$
top:Z
$(\top, \top)$

Hierarchical use-def graph

- **Key idea: Model memory banking as a type** to ensure kernel interfaces are consistent

Subtyping relation forms a **lattice**!

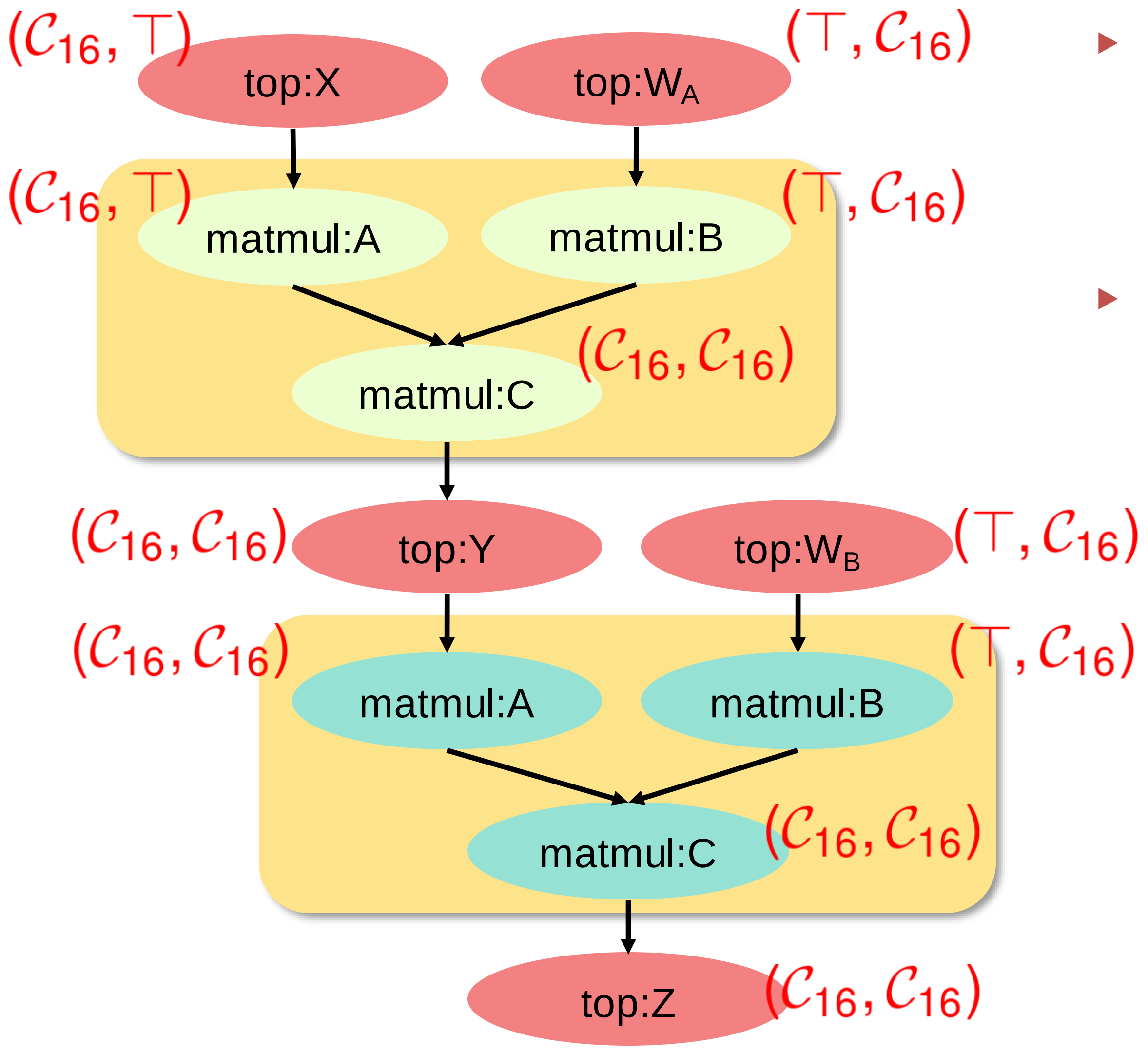


$\perp <: \mathcal{C}_\alpha <: \top$

$\perp <: \mathcal{B}_\alpha <: \top$

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

Temporal Composition



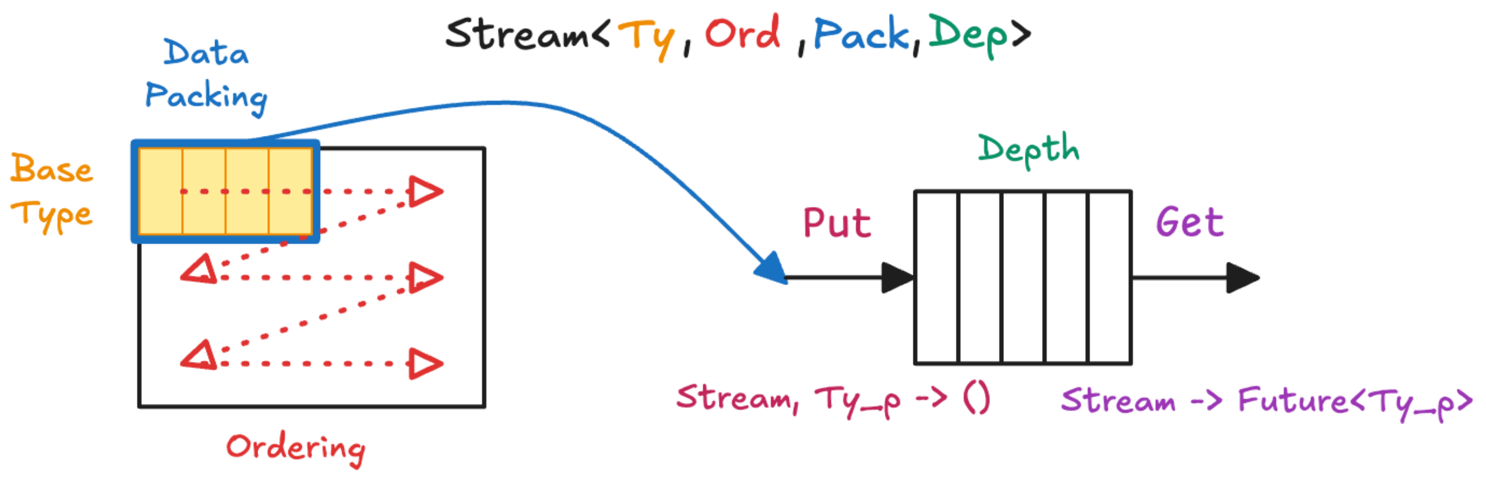
$(\mathcal{C}_{16}, \top)$
top:X
$(\top, \mathcal{C}_{16})$
top:W_A
$(\mathcal{C}_{16}, \top)$
matmul:A
$(\top, \mathcal{C}_{16})$
matmul:B
$(\mathcal{C}_{16}, \mathcal{C}_{16})$
matmul:C
$(\mathcal{C}_{16}, \mathcal{C}_{16})$
top:Y
$(\top, \mathcal{C}_{16})$
top:W_B
$(\mathcal{C}_{16}, \mathcal{C}_{16})$
matmul:A
$(\top, \mathcal{C}_{16})$
matmul:B
$(\mathcal{C}_{16}, \mathcal{C}_{16})$
matmul:C
top:Z
$(\mathcal{C}_{16}, \mathcal{C}_{16})$

Hierarchical use-def graph

- **Key idea: Model data layout as a type** to ensure the kernel interfaces are consistent
- Lattice-based subtyping relation permits **linear-time layout inference** through Worklist algorithm

Spatial Composition

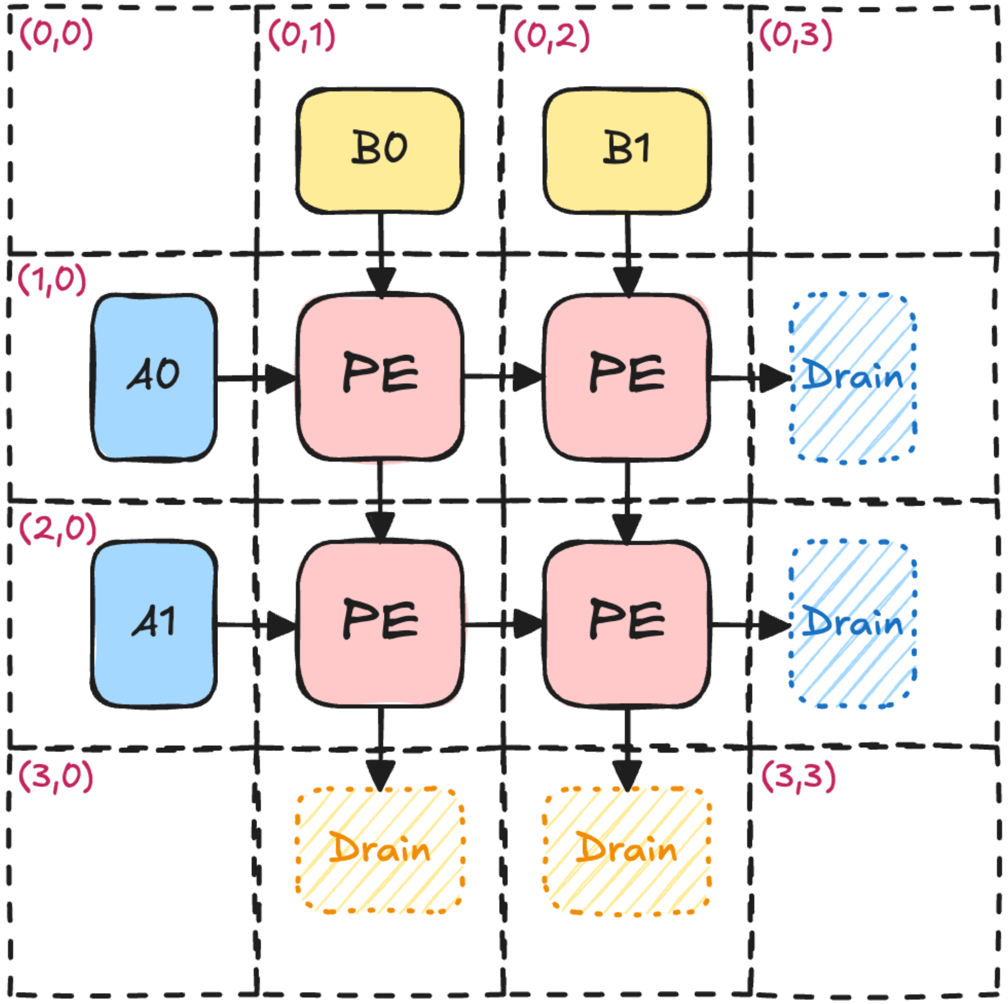
- **Model data streaming as a type**



```
@df.region
def top():
  pipe: Stream[Ty[M, N],
               "R", 1, 4]

  @df.kernel(mapping=[1])
  def producer(A: Ty[M, N]):
   for i, j in allo.grid(M, N):
    # send data
    pipe.put(A[i, j])

  @df.kernel(mapping=[1])
  def consumer(B: Ty[M, N]):
   for i, j in allo.grid(M, N):
    # receive data
    B[i, j] = pipe.get() + 1
```

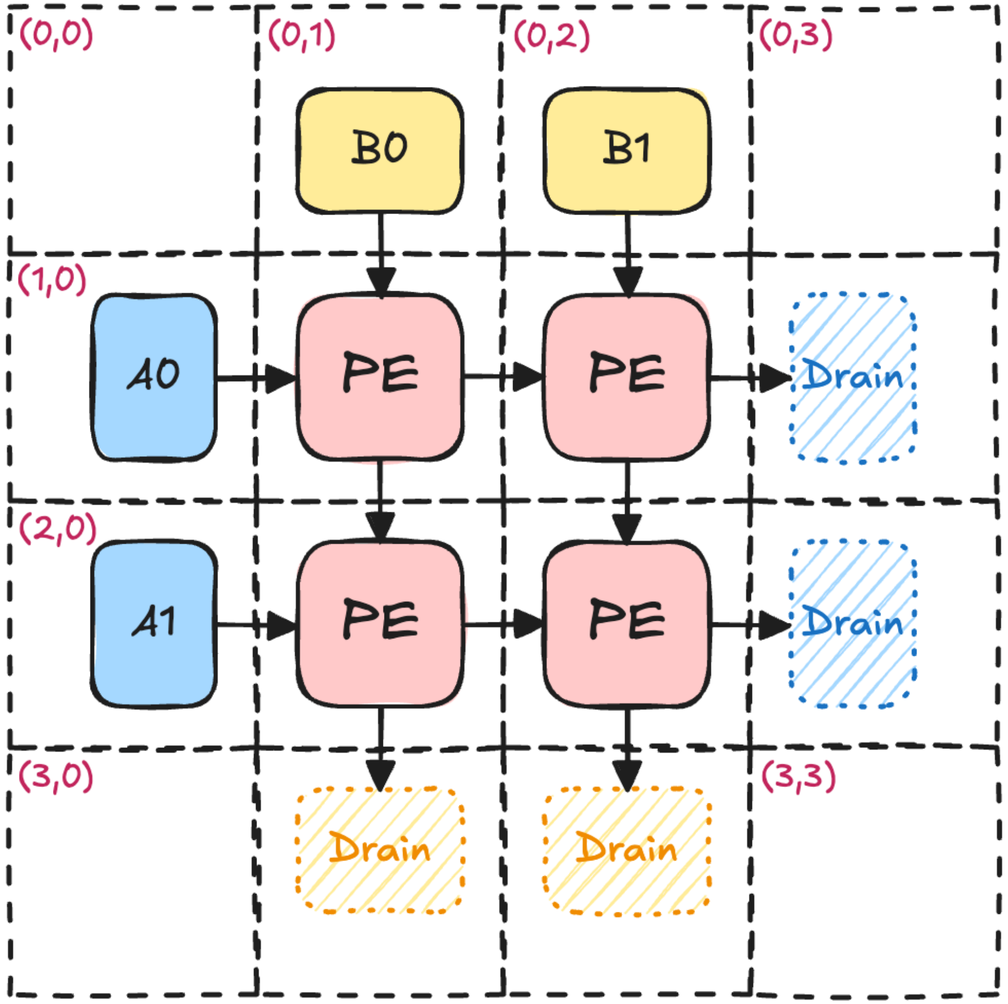
Spatial Composition

```
@df.region()
def top():
 fifo_A = df.array(df.pipe(dtype=float32, depth=4), shape=(P0, P1))
 fifo_B = df.array(df.pipe(dtype=float32, depth=4), shape=(P0, P1))

 @df.kernel(mapping=[P0, P1])
 def gemm(A: float32[M, K],
          B: float32[K, N],
          C: float32[M, N]):
  i, j = df.get_pid()
  # peripheral kernels
  with allo.meta_if(i in {0, M + 1}
              and j in {0, N + 1}):
    pass
  with allo.meta_elif(j == 0):
   # i > 0
   for k in range(K):
    fifo_A[i, j + 1].put(A[i - 1, k])
  with allo.meta_elif(i == 0):
   # j > 0
   for k in range(K):
    fifo_B[i + 1, j].put(B[k, j - 1])
```



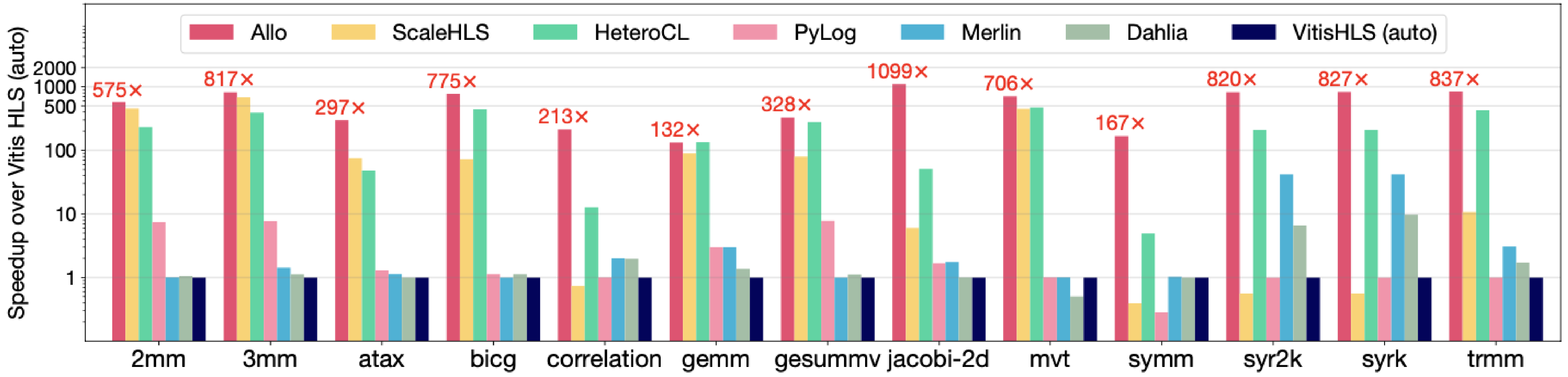
Spatial Composition

```
@df.region()
def top():
  fifo_A = df.array(df.pipe(dtype=float32, depth=4), shape=(P0, P1))
  fifo_B = df.array(df.pipe(dtype=float32, depth=4), shape=(P0, P1))


# drain
with allo.meta_elif(i == M + 1 and j > 0):
 for k in range(K):
  b: float32 = fifo_B[i, j].get()
with allo.meta_elif(j == N + 1 and i > 0):
 for k in range(K):
  a: float32 = fifo_A[i, j].get()
# main body
with allo.meta_else():
  c: float32 = 0
  for k in range(K):
   a: float32 = fifo_A[i, j].get()
   b: float32 = fifo_B[i, j].get()
   c += a * b
   fifo_A[i, j + 1].put(a)
   fifo_B[i + 1, j].put(b)
   C[i - 1, j - 1] = c
```



Single-Kernel Evaluation

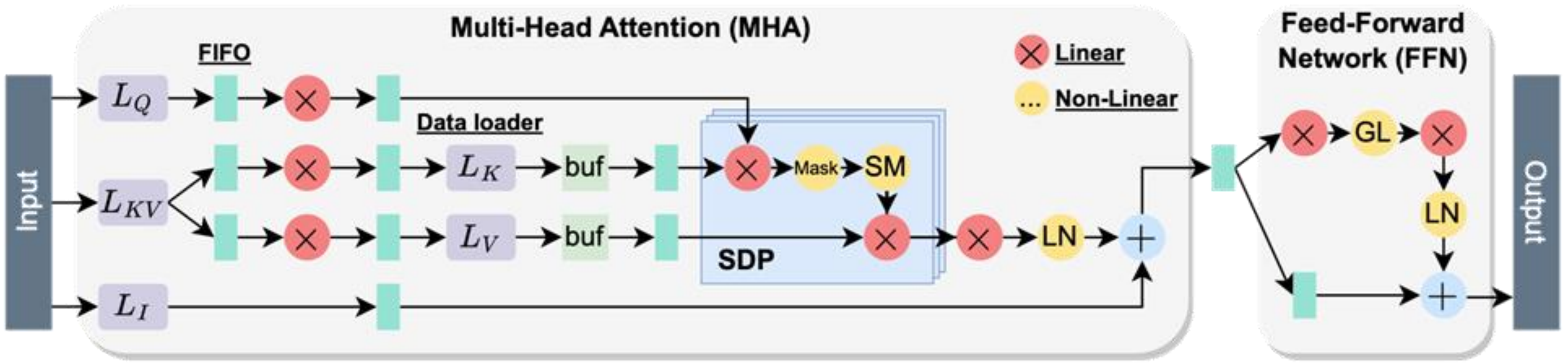
- Benchmarks from PolyBench; Target hardware: AMD U280 FPGA
- Normalized against AMD VitisHLS-auto (pragmas automatically inserted)
- Other baselines
  - ADLs: HeteroCL [FPGA'19], Dahlia [PLDI'20], PyLog [TC'21]
  - Automated DSE for HLS: ScaleHLS [HPCA'22], Merlin [TRETS'22]



Allo achieves (much) higher performance by optimizing data placement with a custom memory hierarchy

Multi-Kernel Evaluation: A Complete LLM Accelerator

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



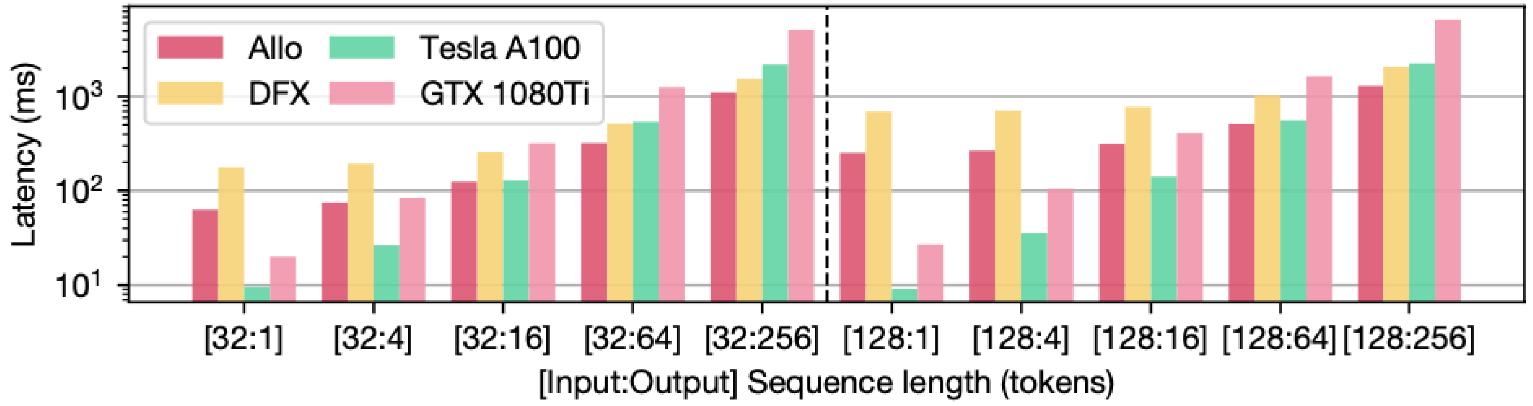
```
def GPT_layer(inp: float32[L, D], ...)
            -> float32[L, D]:
  # 1. Multi-Head Attention (MHA)
  Q = Linear_layer_qkv(inp, Wq, Bq)
  K = Linear_layer_qkv(inp, Wk, Bk)
  V = Linear_layer_qkv(inp, Wv, Bv)
  attn_sf_outp = Self_attention(Q, K, V)
  # ...
  # 2. Feed Forward Network (FFN)
  #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

Compose all the schedules together

```
s = allo.customize(GPT_layer)
s.compose([s_qkv, ..., s_gelu])
```

LLM Accelerator Evaluation

- GPT2: single-batch, low-latency, generative inference settings
  - AMD U280 FPGA (16nm), 250MHz
  - **2.2x speedup in prefill stage** compared to DFX [MICRO'22] (an FPGA-based overlay)
  - **1.9x speedup for long output** sequences and **5.7x more energy-efficient** vs. NVIDIA A100 GPU (7nm)
  - Fewer than 50 lines of schedule code in Allo



	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%)

H. Chen et al., "Understanding the Potential of FPGA-Based Spatial Acceleration for Large Language Model Inference", ACM TRETS (FCCM Journal Track), 2024.

High-Level PyTorch Frontend

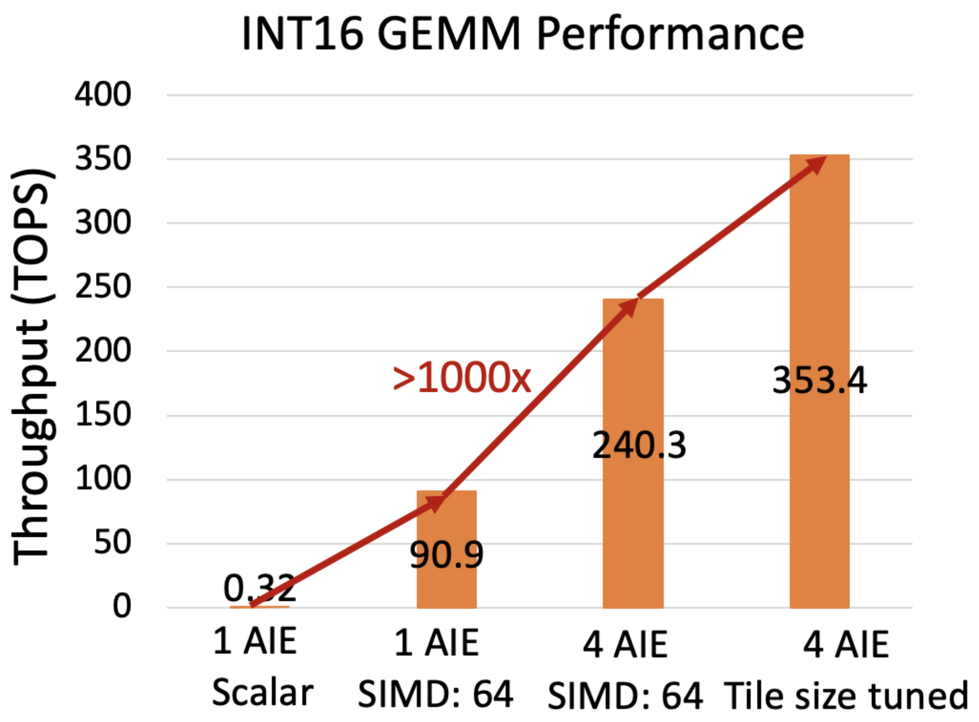
- Predefined schedules for commonly used NN operators
- Can directly import model from PyTorch and build optimized xcel design
  - Through TorchDynamo and torch.fx

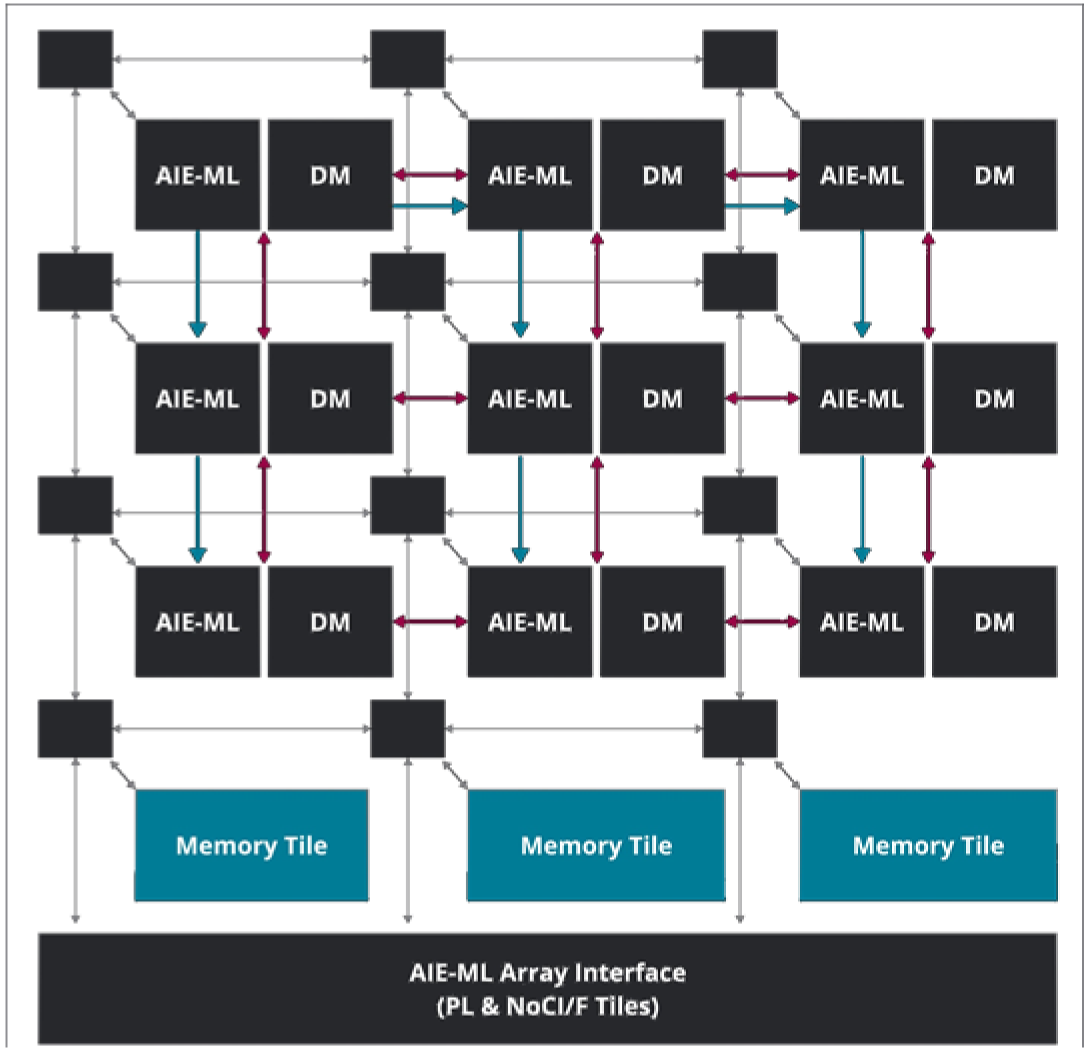
```
import torch
import allo
import numpy as np
from transformers import AutoConfig
from transformers.models.llama.modeling_llama import LlamaModel

bs, seq, hs = 1, 512, 1024

example_inputs = [torch.rand(bs, seq, hs)]
config = AutoConfig.from_pretrained("meta-llama/Meta-Llama-3-8B")
module = GPT2Model(config).eval()
mlir_mod = allo.frontend.from_pytorch(
  module,
  example_inputs=example_inputs,
)
```

Other Backends: AMD AI Engine Evaluation

- GEMM on AMD Ryzen 7940 SoC (CPU + AIE NPU)
  - Orders of magnitude improvement using the streaming interface

```
mod = df.build(target="aie")
```



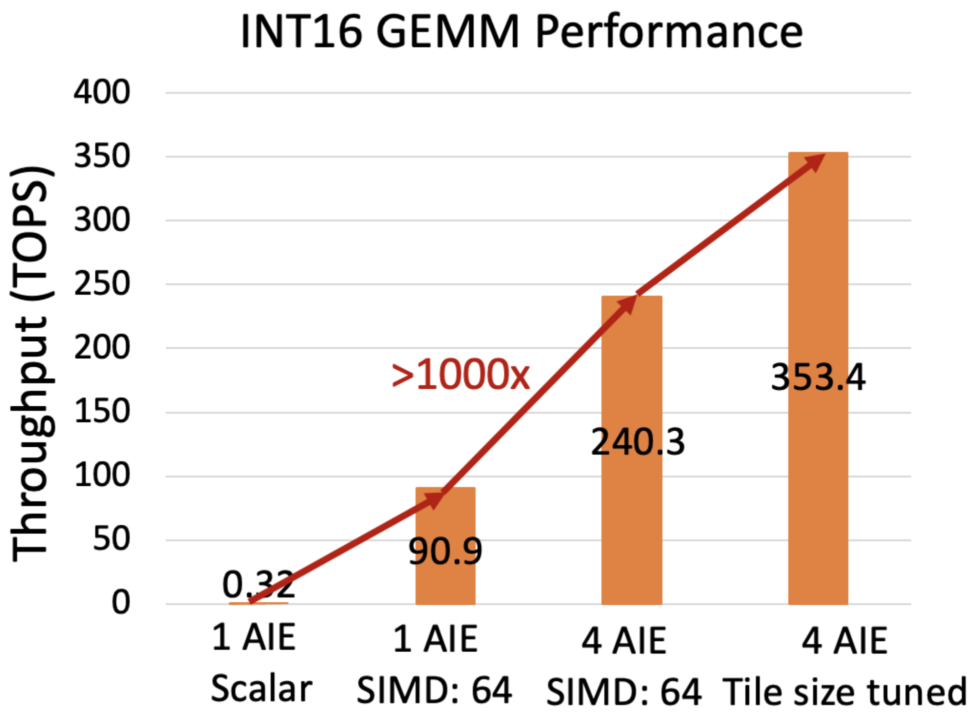


Jinming Zhuang et al., "An Agile MLIR-Based Compilation Flow for Reconfigurable Devices with AI Engines", FPGA, 2025.

Other Backends: AMD AI Engine Evaluation

- ResNet on on AMD Ryzen 7940 SoC (CPU + AIE NPU)
  - 23x speedup compared to AMD Riallto

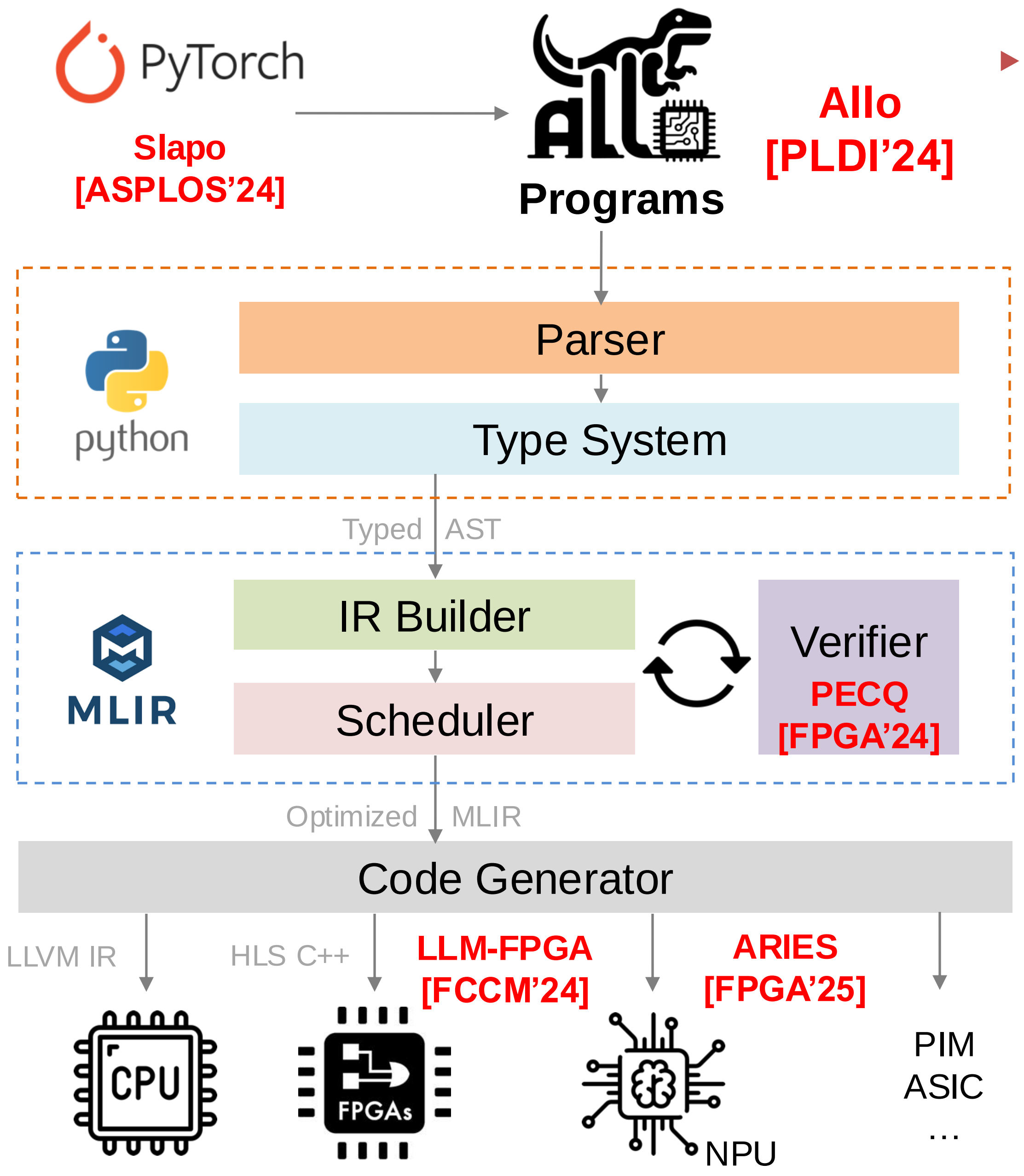
- LLaMA models being evaluated



Designs	Tile	SL	IP	Util (%)	RT (ms)	Speedup
D1 scalar	[16, 16]	1	No	15	57.40	1.24
D2 +vectorized	[16, 16]	4	No	15	16.63	4.29
D3 +vectorized	[16, 16]	8	No	15	9.82	7.27
D4 +conv3×3-2core	[16, 16]	8	No	20	9.18	7.77
D5 +inst-pipeline	[16, 16]	8	Yes	20	8.44	8.45
D6 +opt-tile-size	[32, 16]	8	Yes	20	5.72	13.70
D7 +opt-tile-size	[32, 32]	8	Yes	20	5.21	12.48
D8 +opt-tile-size	[32, 64]	8	Yes	20	4.99	14.30
D9 +more-cores	[32, 64]	8	Yes	40	3.16	22.58
Riallto / RAI-SW[20]	-	-	-	20	71.36	1x

J. Zhuang et al., "An Agile MLIR-Based Compilation Flow for Reconfigurable Devices with AI Engines", FPGA, 2025.

Summary



- Features of Allo ADL
  - Pythonic
  - Decoupled & verifiable customizations
  - Composability

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