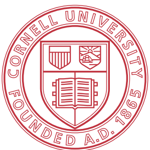
ECE 6775
High-Level Digital Design Automation
Fall 2025

Domain-Specific Programming



Cornell University

Announcements

- HW 2 due tomorrow
  - 1% late penalty; Tue Oct 21 is the hard deadline

- Lab 4 (NN acceleration) will be posted today
  - TWO students per group
  - **Team up by Monday Oct 20**

- 2nd paper reading session next Tuesday
  - C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks", FPGA 2015.

- Prelim on Tuesday 10/28
  - In class, 75 mins
  - Open notes, open book, closed Internet
  - More info will be shared soon

Pipelining: True or False?

- MII = max{ResMII, RecMII} is a lower bound for II, it may not be always achievable

- To pipeline a loop with an II=1, all loop-carried dependencies must have a distance greater than 1

- Not all loop nests can be mapped onto a systolic array

Course Roadmap

- **Background**
  - Introduction
  - Hardware specialization
  - Algorithm basics
- **High-level synthesis**
  - C-based synthesis for FPGAs
  - Front-end compilation
  - Scheduling
  - Resource sharing
  - Pipelining
- **Advanced topics**
  - Domain-specific programming
  - Deep learning acceleration

Agenda

- Motivation for domain-specific languages (DSLs)
  - A brief intro to Halide, a schedule language that decouples algorithm description from optimizations
- Two representative DSLs for accelerator design
  - SuSy: A DSL for constructing systolic arrays
  - Allo: A Python-based programming model for composable accelerator design

Donald Knuth on Multicore Architectures

Q: Vendors of multicore processors have expressed frustration at the difficulty of moving developers to this model. As a former professor, what thoughts do you have on this transition and how to make it happen?

> I might as well flame a bit about my personal unhappiness with the current trend toward multicore architecture. To me, **it looks more or less like the hardware designers have run out of ideas, and that they're trying to pass the blame for the future demise of Moore's Law to the software writers by giving us machines that work faster only on a few key benchmarks!** I won't be surprised at all if the whole multithreading idea turns out to be a flop, worse than the "Itanium" approach that was supposed to be so terrific — until it turned out that the wished-for compilers were basically impossible to write.

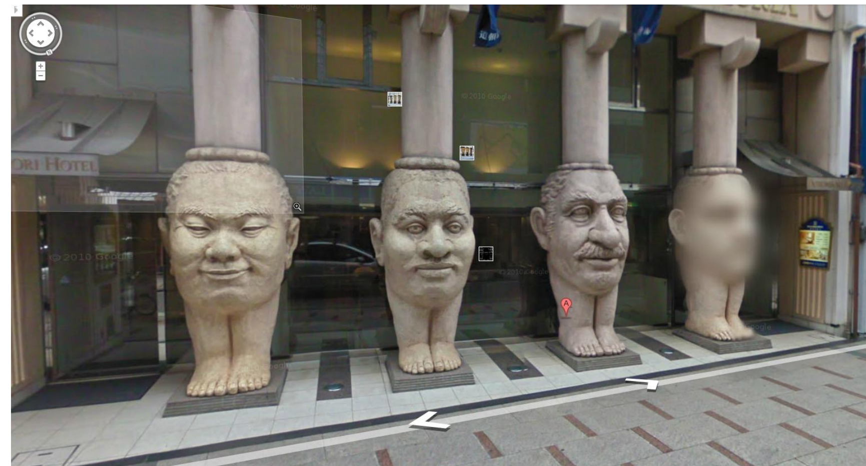
Source: https://news.ycombinator.com/item?id=3366360, 2008

Blur Filter: Original C++ Code

```
void blur_filter_3x3(const Image &in, Image &blury) {
  Image blurx(in.width(), in.height()); // allocate blurx array
  for (int x = 0; x < in.width(); x++)
        for (int y = 0; y < in.height(); y++)
      blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

  for (int x = 0; x < in.width(); x++)
        for (int y = 0; y < in.height(); y++)
      blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
}
```

The blurred face of KFC Mascot



Blur Filter: Optimized C++ Code for Multicore

```
void blur_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
        __m128i *blurxPtr = blurx;
        for (int y = -1; y < 32+1; y++) {
                const uint16_t *inPtr = &(in[yTile+y][xTile]);
                for (int x = 0; x < 256; x += 8) {
                        a = _mm_loadu_si128((__m128i*)(inPtr-1));
                        b = _mm_loadu_si128((__m128i*)(inPtr+1));
                        c = _mm_load_si128((__m128i*)(inPtr));
                        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                        avg = _mm_mulhi_epi16(sum, one_third);
                        _mm_store_si128(blurxPtr++, avg);
                        inPtr += 8;
      }}
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
                a = _mm_load_si128(blurxPtr+(2*256)/8);
                b = _mm_load_si128(blurxPtr+256/8);
                c = _mm_load_si128(blurxPtr++);
                sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                avg = _mm_mulhi_epi16(sum, one_third);
                _mm_store_si128(outPtr++, avg);
}}}}}
```

11X faster on quad core x86 processor

+ Tiling
+ Vectorization
+ Multithreading

Less is More – Domain-Specific Languages (DSLs)

- Programming languages that are tailored for a specific application domain
  - More accessible and productive for domain experts
  - Restricted expressiveness facilitates more automated optimization and verification
  - Examples: SQL, MATLAB, OpenGL, HTML, …
- Embedded DSLs (eDSLs)
  - A DSL built on a host, typically general-purpose language
  - Examples: Halide (in C++), Chisel (in Scala), PyTorch (in Python), …

Case Study: Halide, an eDSL for Image Processing

Main Idea: Separate algorithm (what to compute) from schedule (how to compute it)

```
// Algorithm of Blur Filter
blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
```

```
// Schedule
blurx.compute_at(blur_y, y).unroll(x);
blury.tile(x, y, xi, yi, 256, 32);
```

Algorithm
- Write and test once
- Portable across platforms

Schedule
- Specify optimizations
- Explore combinations
- Target different back-ends

Scheduling functions encode common program transformations, e.g.

tile: loop tiling
unroll: loop unrolling
compute_at: change order of computation

[1] J. Ragan-Kelley et al. Halide: a Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. PLDI'2013.

[2] https://halide-lang.org

Halide Example Code

- Example: 3D convolution with N inputs and N K-by-K filters

```
// C++
for (int i = 0; i < N; i++)
 for (int y = 0; y < Rows; y++)
  for (int x = 0; x < Cols; x++)
   for (int ry = 0; ry < K; ry++)
    for (int rx = 0; rx < K; rx++)
     out(x, y) += wt(rx, ry, i) * in(x+1-rx, y+1-ry, i);
```

Imperative syntax emphasize in C++

```
// Halide
RDom r(0,K, 0,K, 0,N);
out(x, y) += filter(r.x, r.y, r.z) * in(x+1-r.x, y+1-r.y, r.z);
```

Halide's functional syntax

Halide Example Code Optimization

- Example: 3D convolution with N inputs and N K-by-K filters

```
// C++
for (int yo = 0; yo < Rows/TR; yo++)
for (int xo = 0; xo < Cols/TC; xo++)
 for (int yi = 0; yi < Rows; yi++)
 for (int xi = 0; xi < Cols; xi++)
  for (int i = 0; i < N; i++)
   for (int ry = 0; ry < K; ry++)
    for (int rx = 0; rx < K; rx++)
     out(xo*TC+xi, yo*TR+yi) += wt(rx, ry, i) *
    in(xo*TC+xi+1-rx, yo*TC+yi+1-ry, i);
```

New loops

Loop Reordering

Rewrite indices

Optimizing C++

- Significant and intrusive code changes
- Must retest new code in case bugs are introduced
- Rewriting and retesting code for each new design point is slow

```
// Halide
RDom r(0,K, 0,K, 0,N);
out(x, y) += filter(r.x, r.y, r.z) * in(x+1-r.x, y+1-r.y, r.z);
out(x, y).tile(x,y, xo, xi, yo, yi, TR, TC)
        .reorder(xo, yo, xi, yi, i);
```

New code

Optimizing Halide

- Algorithm code is unchanged
- Scheduling functions encode common optimizations
- Productive design exploration

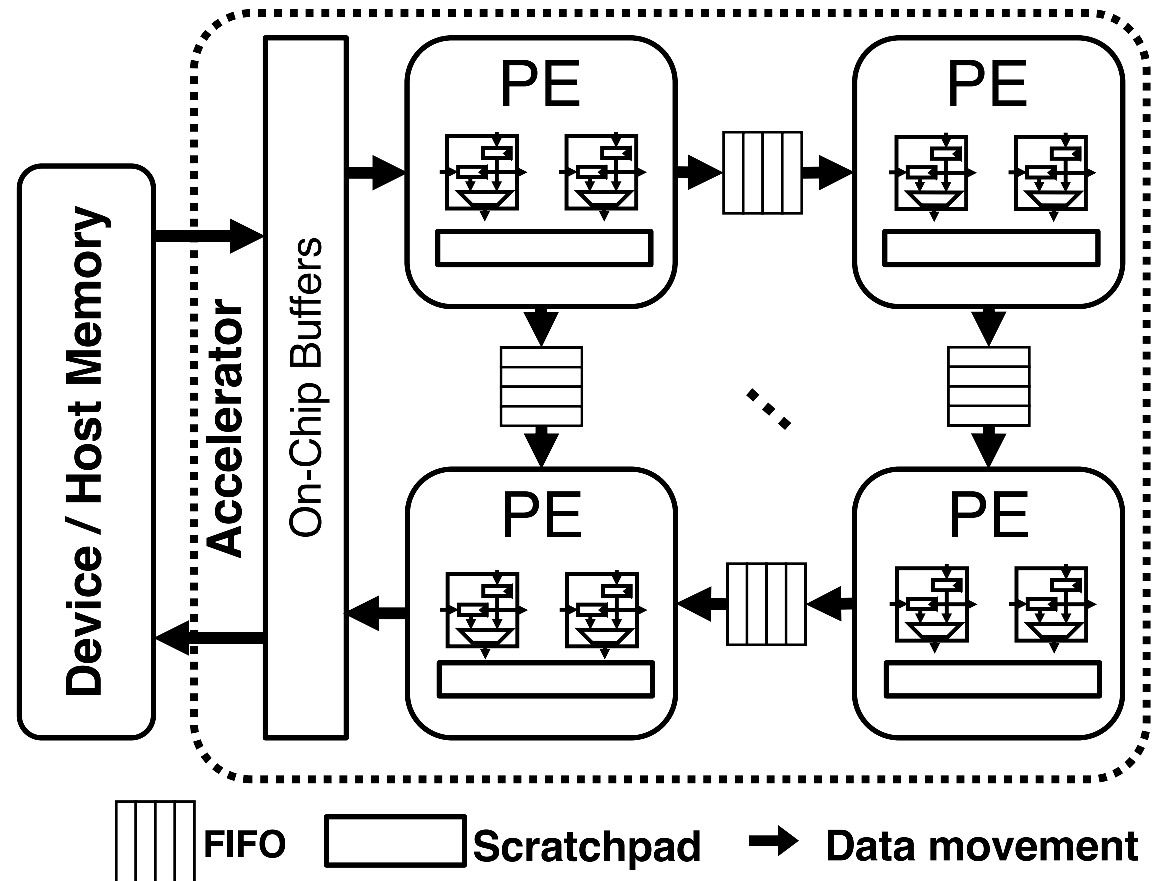
What about Accelerator Design?

Custom Compute Units:
Use complex "instructions" to amortize overhead

Custom Numeric Types:
Trade off accuracy and efficiency with low-bitwidth integer or fixed-point types

Custom Memory Hierarchy:
Exploit data access patterns with reuse buffers, etc.

Custom Communication Architecture: Tailor on-chip networks to data movement patterns



Accelerators are even more challenging to design and program!

Building Accelerator with HLS C/C++

```
void mm_tile ( int8 X_tile[2][768],  int8 W_tile[768][2], int8 Z_tile[2][2] )
{
 #pragma dataflow
 hls::stream<int8> X_fifo[2][3], W_fifo[2][3];
 #pragma stream variable=X/W_fifo depth=3
 #pragma partition variable=X/W/Z_tile complete dim=1
 for (int k4 = 0; k4 < 768; k4++) {
  for (int m = 0; m < 2; m++) {
   int8 v105 = X_tile[m][k4];
   X_fifo[m][0].write(v105);}
   // ... write W_fifo
  }}
 for (int Ti = 0; Ti < 2; ++Ti) {
 #pragma HLS unroll
  for (int Tj = 0; Tj < 2; ++Tj) {
  #pragma HLS unroll
   // ... load X/W_fifo
   PE(X_in, X_out, W_in, W_out, Z, Ti, Tj);
}}}

void matmul ( int8 X[512][768], int8 W[768][768],  int8 Z[512][768] ) {
 int8 local_X[2][768], local_W[768][2], local_Z[2][2];
 for (int mi = 0; mi < 256; mi++) {
  for (int ni = 0; ni < 384; ni++) {
  // ... load X, W
   mm_tile(local_X, local_W, local_Z);
}}}
```

Communication Customization

Memory Customization

Compute Customization

~500 lines of HLS code for a small systolic array (vendor-specific; hard to maintain & reuse)

Vanilla MatMul (1% theoretical peak perf.)

+ Custom compute (~30% peak)
    - + Loop tiling
    - + Loop unrolling
    - + Loop pipelining
    - + Function pipelining

+ Custom memory hierarchy (~50% peak)
    - + Tiling/double buffering
    - + Reuse buffer insertion
    - + Memory banking/partitioning

+ Custom communication (95% peak)
    - + Data streaming
    - + Data vectorization & coalescing
    - + Systolic communication

SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs

Yi-Hsiang Lai[1], Hongbo Rong[2], Size Zheng[3], Weihao Zhang[4], Xiuping Cui[3], Yunshan Jia[3], Jie Wang[5], Brendan Sullivan[1], Zhiru Zhang[1], Yun Liang[3], Youhui Zhang[4], Jason Cong[5], Nithin George[2], Jose Alvarez[2], Christopher Hughes[2], Pradeep Dubey[2]

[1]Cornell University, [2]Intel, [3]Peking University, [4]Tsinghua University, [5]UCLA

International Conference On Computer Aided Design (ICCAD)

Recap: Uniform Recurrence Equations (UREs)

- Any **systolic algorithm** can be described by a set of UREs
  - i.e., an n-dimensional loop nest where the recurrences (inter-iteration dependences) must have constant distances

$\mathbf{c} = \mathbf{A} * \mathbf{b}$

```
for (int i = 0; i < N; i++)
  c[i] = 0;
  for (int j = 0; j < N; j++)
    c[i] += A[i, j] * b[j]
```

Matrix Vector Multiplication (MV) in UREs

$$Z[i, 0] = 0$$
$$Z[i, j] = Z[i, j-1] + A[i, j] \cdot x[j], \qquad 0 < j < N$$
$$c[i] = Z[i, N-1]$$

$\mathbf{C} = \mathbf{A} * \mathbf{B}$

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    C[i, j] = 0;
    for (int k = 0; k < N; k++)
      C[i, j] += A[i, k] * B[k, j]
```

Matrix Matrix Multiplication (MM) in UREs

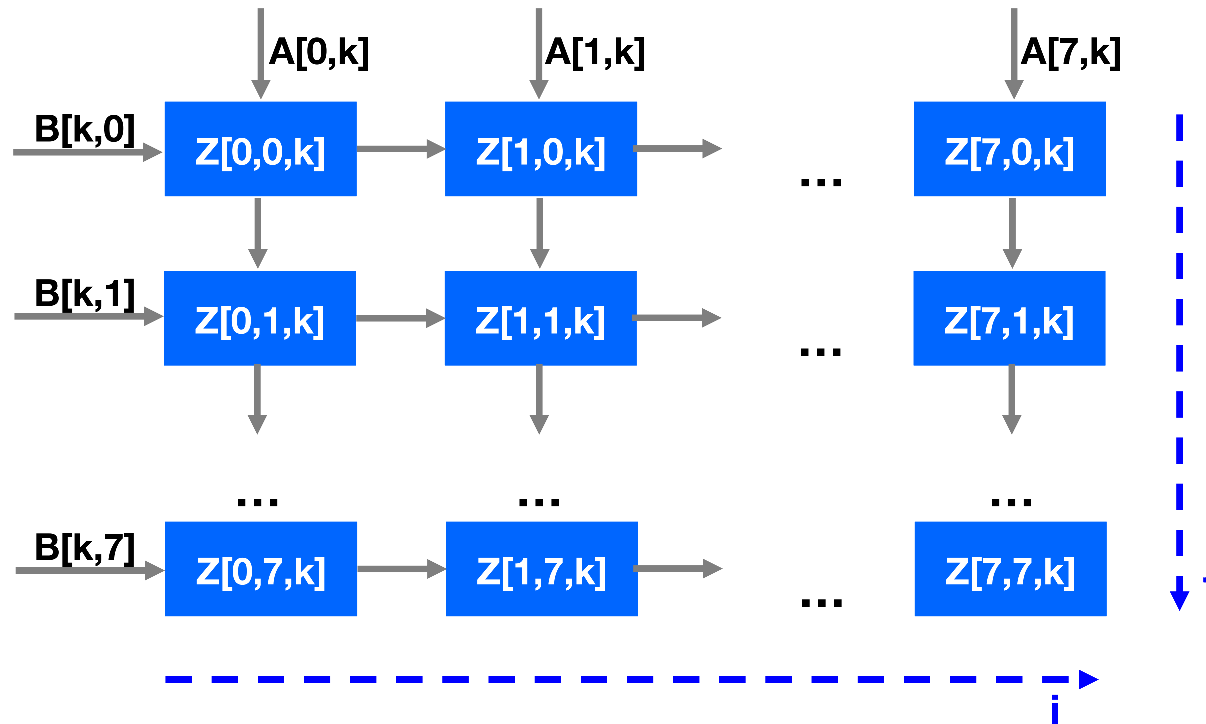
$$Z[i, j, 0] = 0$$
$$Z[i, j, k] = Z[i, j, k-1] + A[i, k] \cdot B[k, j], 0 < k < N$$
$$C[i, j] = Z[i, j, N-1]$$

Recap: Mapping MM to a Systolic Array

- <u>One way</u> to map MM into a physical array of PEs

$$\mathbf{C} = \mathbf{A} * \mathbf{B}$$

$$Z[i,j,0] = 0$$
$$Z[i,j,k] = Z[i,j,k-1] + A[i,k] \cdot B[k,j], \qquad 0 < k < N$$
$$C[i,j] = Z[i,j,N-1]$$



Space-Time Transform: A Generalized View of Mapping Strategies

- A **space-time transform** is a function that determines where (in space: which PE) and when (in time: at which cycle) each operation in a loop nest occurs
  - e.g., (i, j, k) → (PE coordinates, time step), which captures a mapping between a 3-level loop nest and the systolic array that executes it

- Each space-time transform can be represented by a **transformation matrix**

$$T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \hdashline 1 & 1 & 1 \end{pmatrix}$$

Π **Spatial mapping** $\Pi \times (i, j, k)^T = (i, j)$

τ **Temporal schedule** $\tau \times (i, j, k)^T = i + j + k$

Example: transformation matrix of MM for mapping to a 2D systolic array

- **Spatial mapping:** the first two rows determine *where* each operation is mapped in a 2D PE array
- **Temporal schedule:** the last row defines *when* that operation is executed

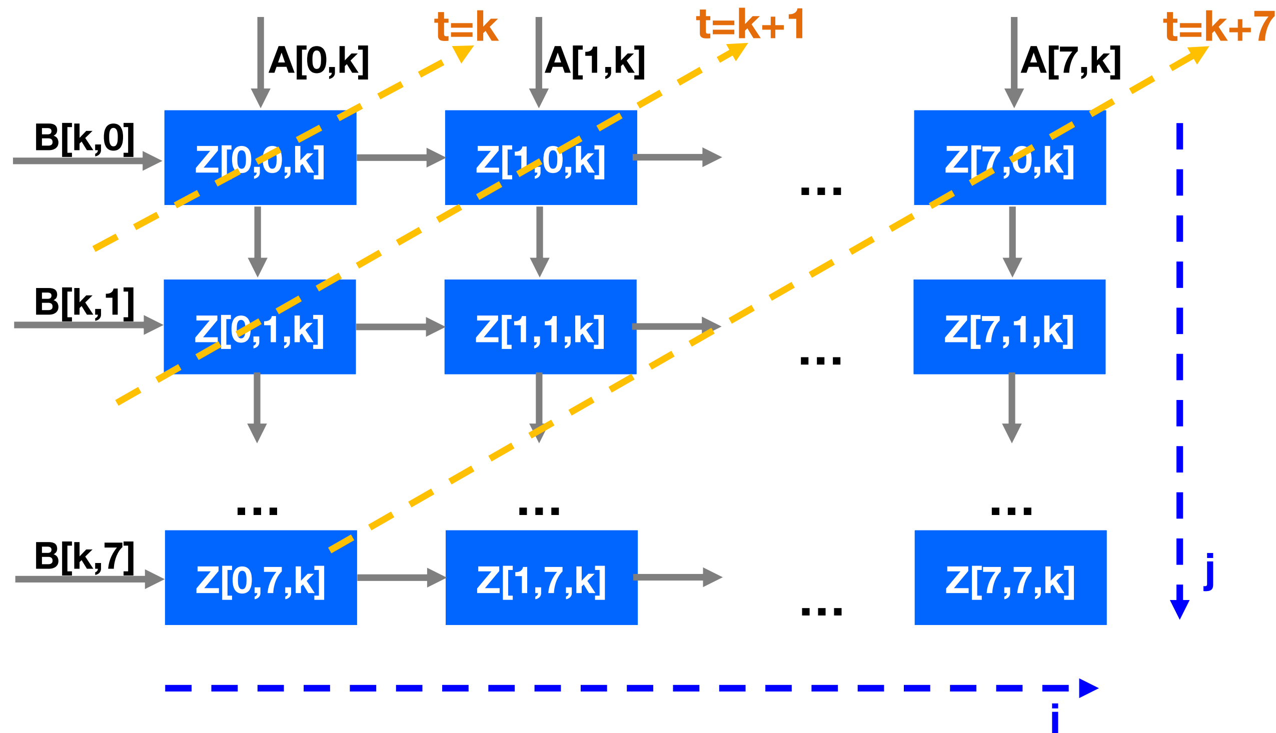
Example: Space-Time Transform of MM

$$T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

Transformation matrix

Π **Spatial mapping** $\Pi \times (i, j, k)^T = (i, j)$

τ **Temporal schedule** $\tau \times (i, j, k)^T = i + j + k$

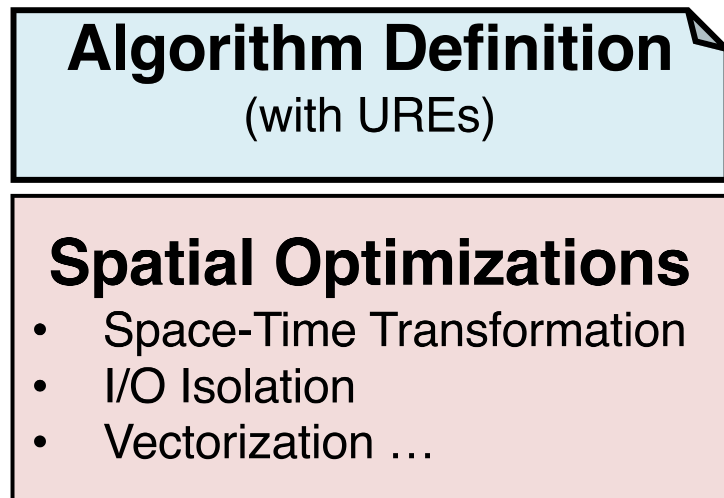


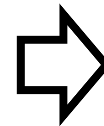
An eDSL for Constructing Systolic Arrays

A programming model for accelerating systolic algorithms

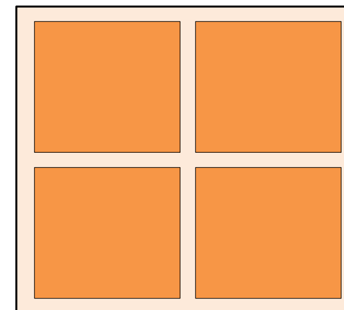
- Decoupled algorithm definition and spatial optimizations
- Explicitly represent optimizations such as **s**pace-time transformation
- Concisely describe a systolic algorithm with **U**REs

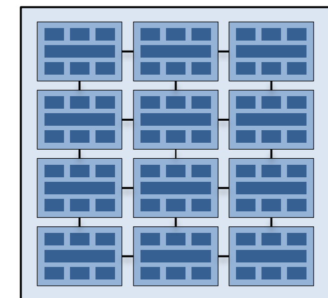
SuSy





Processors + Accelerators





Algorithm Specification with UREs

- Any systolic algorithm can be described by a set of UREs
  - i.e., an n-dimensional loop nest where the recurrences (inter-iteration dependences) must have constant distances

$\mathbf{C} = \mathbf{A} * \mathbf{B}$

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    C[i, j] = 0;
    for (int k = 0; k < N; k++)
      C[i, j] += A[i, k] * B[k, j]
```

Matrix Matrix Multiplication (MM) in UREs

$$Z[i,j,0] = 0$$
$$Z[i,j,k] = Z[i,j,k-1] + A[i,k] \cdot B[k,j], 0 < k < N$$
$$C[i,j] = Z[i,j,N-1]$$

Algorithm Definition in SuSy

```
// Iteration space
Var i, j, k;
// UREs
Z(i, j, k) = select(k==0, 0, Z(i, j, k-1)) + A(i, j, k) * B(i, j, k);
C(i, j) = select(k == N-1, Z(i, j, k));
```

Declarative Programming (builds on Halide)

Supported Spatial Optimizations

Optimizations for custom I/O

F.merge_ures(U_1, U_2, ..., U_n)	Define the set of UREs F, U_1, U_2, ..., U_n to optimize.
F.space_time_transform(space, tau)	Specify the space-time transformation that will be applied to F, where space is the set of space loops, and tau is the scheduling vector.
F.vectorize(var)	Vectorize the specified loop variable var of F.
F.reorder(var_1, var_2, ..., var_n)	Reorder the loop nest for F according to the specified order, starting from the innermost level.
F.isolate_producer({E_1, E_2, ...}, P)	Isolate a list of expressions {E_1, E_2, ...} (usually inputs) in F to a separate producer kernel P.
F.isolate_consumer(E, C)	Isolate an expression E (usually an output) in F to a separate consumer kernel C.
F.remove(var)	Remove loop var of F.
F.buffer(E, v, mode)	Insert a reuse buffer at loop v for expression E with mode (either Buffer::Single or Buffer::Double).
F.scatter(E, var)	Reduce data communication overhead (i.e., data broadcast) by scattering the expression E to the consumer along loop var.
F.gather(E, var)	Reduce data communication overhead (i.e., data broadcast) by gathering the expression E from the producer along loop var.

https://github.com/IntelLabs/t2sp

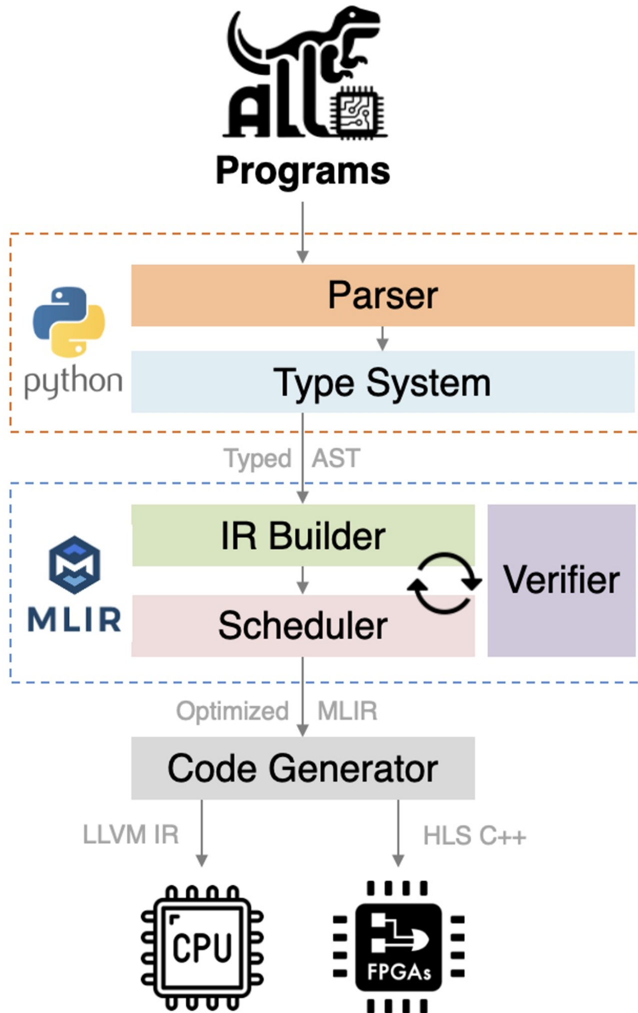
Allo: A Programming Model for Composable Accelerator Design

Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang

Cornell University

ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2024

Allo Accelerator Design Language (ADL)

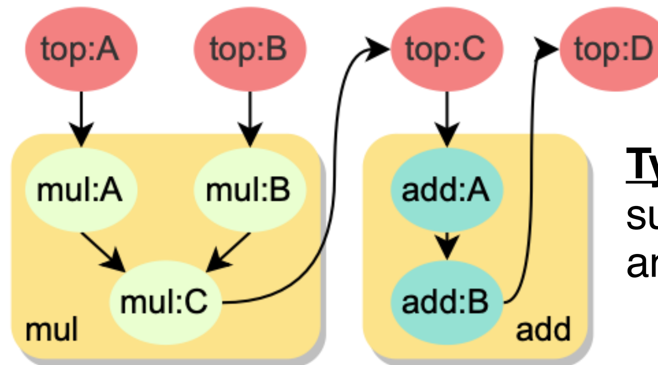



```
def matmul(A: int8[M, K], B: int8[K, N])
        -> int16[M, N] :
  C: int16[M, N] = 0
  for i, j, k in allo.grid(M, N, K):
    C[i, j] += A[i, k] * B[k, j]
  return C
```

Pythonic: easier to pick up and interoperates with other high-level DSLs

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

Verifiable customizations: productive, progressive, and portable optimizations



Type-safe composition: supports both behavioral and structural styles

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

Decoupled Customization (1)

- Allo separates algorithm from various code transformations for hardware customizations

Algorithm

Allo code

```
def conv( image: T[N, N],
          kernel: T[3, 3] )
      -> T[N - 2, N - 2]:
  out: T[N - 2, N - 2] = 0
  for y, x, r, c in allo.grid(N, N, 3, 3):
    out[y, x] += image[y + r, x + c] * kernel[r, c]
  return out
```

Imperative programming

Corresponding HLS code in C

```
for (int y = 0; y < N; y++)
 for (int x = 0; x < N; x++)
  for (int r = 0; r < 3; r++)
   for (int c = 0; c < 3; c++)
    out[y, x] += image[y+r, x+c] * kernel[r, c]
```

Decoupled customization

```
s = allo.customize(conv)
s.split("x", factor=M)
s.reorder("x.outer", "x.inner", "y")
```

```
for (int xi = 0; xi < M; xi++)
 for (int xo = 0; xo < N/M; xo++)
  for (int y = 0; y < N; y++)
   for (int r = 0; r < 3; r++)
    for (int c = 0; c < 3; c++)
     out[y, xi+xo*M] +=
      image[y+r, xi+xo*M+c] * kernel[r, c]
```

Tiled loop

Reorder loops

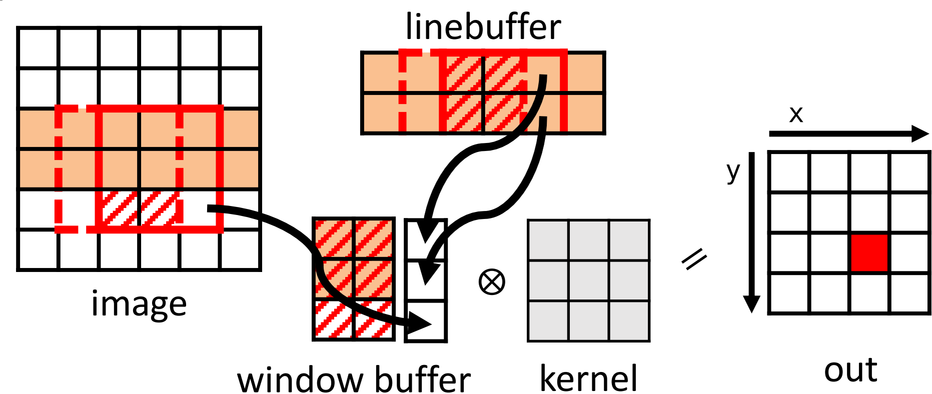
Decoupled Customization (2)

- Inferring custom on-chip storage with .reuse_at()

```
def conv(image: T[N, N],
               kernel: T[3, 3])
           -> T[N - 2, N - 2]:
  out: T[N - 2, N - 2] = 0
  for y, x, r, c in allo.grid(N, N, 3, 3):
    out[y, x] += image[y + r, x + c] * kernel[r, c]
  return out
```

```
for (int y = 0; y < N; y++)
  for (int x = 0; x < N; x++)
    for (int r = 0; r < 3; r++)
      for (int c = 0; c < 3; c++)
        out[y, x] += image[y+r, x+c] * kernel[r, c]
```

```
s = allo.customize(conv)
linebuf = s.reuse_at(s.image, "y")
winbuf = s.reuse_at(linebuf, "x")
```

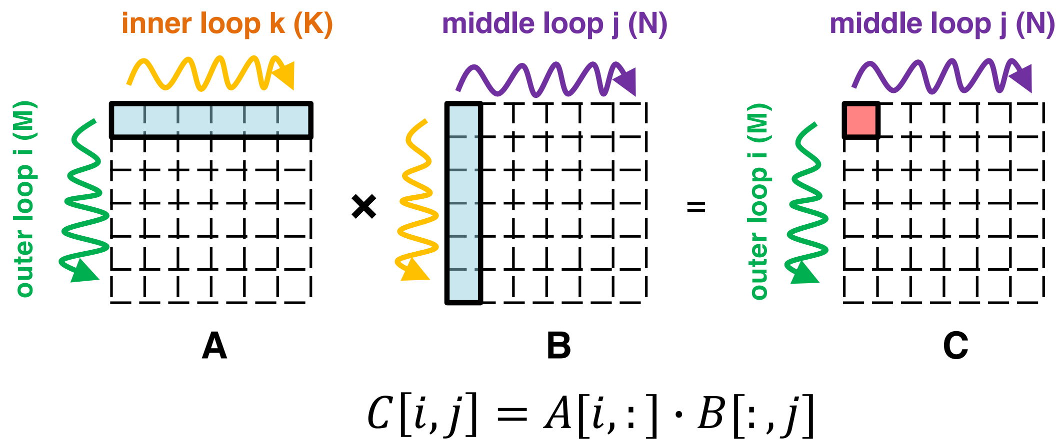


Customization Primitives in Allo (a subset)

Compute Customizations		Memory Customizations	
`s.split(i,v)`	Split loop `i` into a two-level nested loop with `v` as the bound of the inner loop.	`s.buffer_at(A,i)`	Create an intermediate buffer at loop `i` to store the results of array `A`.
`s.fuse(*l)`	Fuse multiple sub-loops `l` in the same nest loop into one.	`s.reuse_at(A,i)`	Create a buffer storing the values of array `A`, where the values are reused at loop `i`.
`s.reorder(*l)`	Switch the order of sub-loops `l` in the same nest loop.	`s.partition(A,d,v)`	Cyclic/Block partition dimension `d` of array `A` with a factor `v`.
`s.compute_at (Op1,Op2,i)`	Merge loop `i` of the operation `Op1` to the corresponding loop level in operation `Op2`.	`s.pack(A,i,v)`	Pack dimension `i` of array `A` into words with a factor `v`.
`s.unroll(i,v)`	Unroll loop `i` by factor `v`.		
`s.unfold(i)`	Unfold loop `i` as hardware instances.	**Communication Customizations**	
`s.pipeline(i,v)`	Schedule loop `i` in a pipeline manner with a target initiation interval `v`.	`s.relay(A,Dst,v)`	Connect array `A` to destination `Dst` with a FIFO of depth `v`.

Case Study: Matrix Multiplication (MM)

- A vanilla MM implementation performs inner product to produce *one output element*
  - **Floating-point accumulation** introduces carried dependency, slowing down the pipeline (II>1)



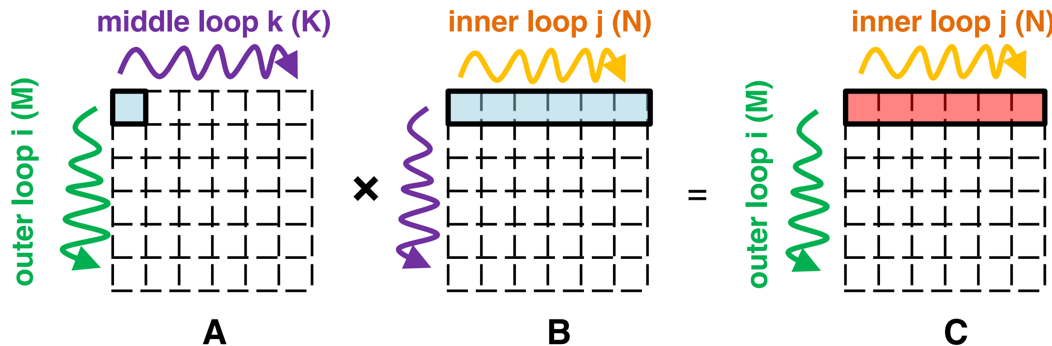
$$C[i,j] = A[i,:] \cdot B[:,j]$$

```
for (int i = 0; i < M; i++) {
 for (int j = 0; j < N; j++) {
  C[i, j] = 0;
  for (int k = 0; k < K; k++) {
   #pragma pipeline II=??
   C[i, j] += A[i, k] * B[k, j]
  }
 }
}
```

Case Study: Optimized MM to Lower II

- The row-wise product approach performs a sequence of scalar-vector products to produce *one output row*
  - An additional buffer is added to store the intermediate results (i.e., c_vec)

MatMul via row-wise product



$$C[i,:] = \sum_k A[i,k] \cdot B[k,:]$$

```
for (int i = 0; i < M; i++) {
  float C_vec[N];
  for (int j = 0; j < N; j++)
    C_vec[j] = 0.0;

  for (int k = 0; k < K; k++) {
    for (int j = 0; j < N; j++) {
      #pragma pipeline II=??
      C_vec[j] += A[i, k] * B[k, j];
    }
  }
  for (int j = 0; j < N; j++)
    C[i, j] = C_vec[j];
}
```

Case Study: Optimized MM in Allo

- Optimizations via decoupled primitives
  - .reorder() swaps the order of the k and j loops
  - .buffer_at() creates an intermediate buffer at a given axis
  - Algorithm code stays unchanged

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

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

	II	Latency (cycles)	Speedup
Vanilla MM	8	4295M	1x
Optimized MM	1	539M	**7.97x**

Constructing a Systolic Array for MM

➔ Algorithm specification

python

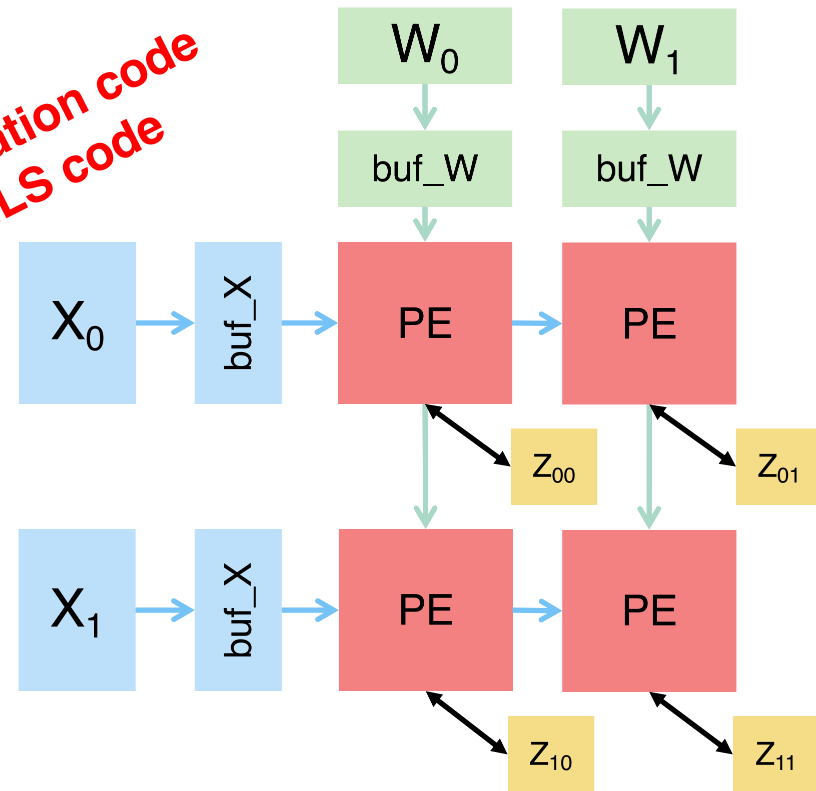
```
def matmul( X: int8[M, K], W: int8[K, N],
            Z: int16[M, N] ) :
  for i, j in allo.grid(M, N, "PE") :
    for k in range(K):
      Z[i, j] += X[i, k] * W[k, j]
```

➔ Customization

```
s = allo.customize(matmul)
buf_X = s.buffer_at(s.X, "i")
buf_W = s.buffer_at(s.W, "j")
pe = s.unfold("PE", axis=[0, 1])
s.partition(s.X, dim=0)
s.partition(s.W, dim=1)
s.partition(s.Z, dim=[0, 1])
s.relay(buf_X, pe, axis=0)
s.relay(buf_W, pe, axis=1)
```

8 lines of customization code vs ~500 lines HLS code

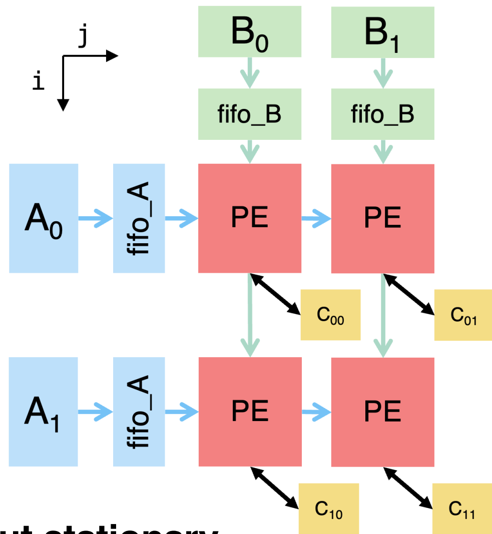
➔ Architectural Diagram



Varying Space-Time Mapping Strategies

➔ **Spatial loop: i, j**

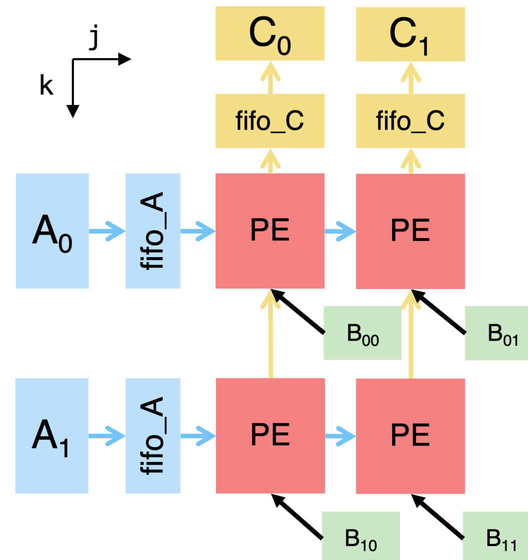
```
s = allo.customize(matmul)
buf_A = s.buffer_at(s.A, ”i")
buf_B = s.buffer_at(s.B, "j")
pe = s.unfold("PE", axis=[0, 1],
                    factor=[M, N])
s.partition(s.A, dim=0)
s.partition(s.B, dim=1)
s.partition(s.C, dim=[0, 1])
s.relay(buf_A, pe, axis=0)
s.relay(buf_B, pe, axis=1)
```



Output stationary

➔ **Spatial loop: k, j**

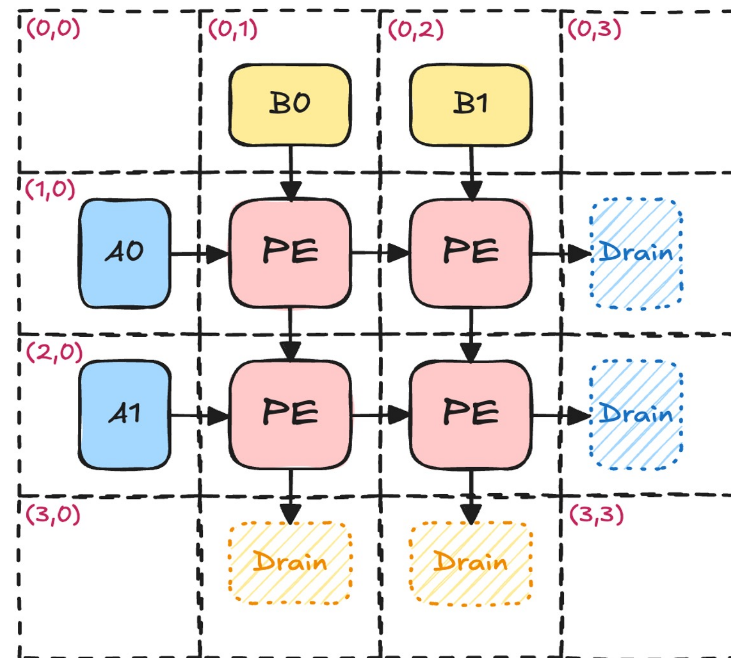
```
s = allo.customize(matmul)
s.reorder("k", "j", "i")
buf_A = s.buffer_at(s.A, ”i")
buf_C = s.buffer_at(s.C, "j")
pe = s.unfold("PE", axis=[0, 1],
                    factor=[K, N])
s.partition(s.A, dim=0)
s.partition(s.B, dim=[0, 1])
s.partition(s.C, dim=1)
s.relay(buf_A, pe, axis=0)
s.relay(buf_B,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



Weight stationary

Structural Composition

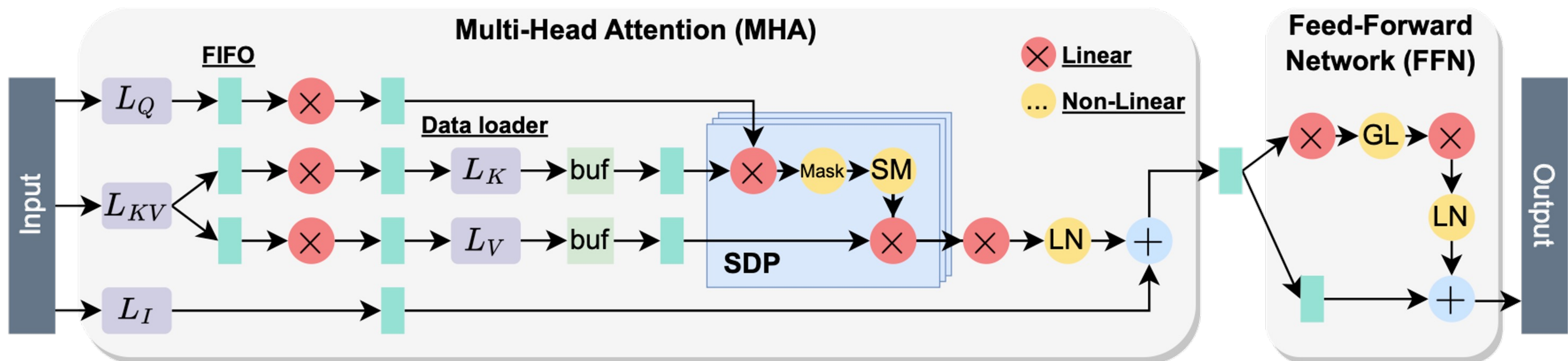
```
pipe_A: Stream[int8, 4][P0, P1]
pipe_B: Stream[int8, 4][P0, P1]
@df.kernel(mapping=[P0, P1])
def matmul(A: int8[M, K],
        B: int8[K, N],
        C: int16[M, N]):
  i, j = df.get_pid()
  with allo.meta_elif(i == M+1 and j > 0):
   for k in range(K):  # drain
    b: int8 = pipe_B[i, j].get()
  with allo.meta_elif(j == N + 1 and i > 0):
   for k in range(K):
    a: int8 = pipe_A[i, j].get()
  with allo.meta_else():  # main processing
   c: int16 = 0
   for k in range(K):
    a: int8 = pipe_A[i, j].get()
    b: int8 = pipe_B[i, j].get()
    c += a * b
    C[i - 1, j - 1] = c
    pipe_A[i, j + 1].put(a)
    pipe_B[i + 1, j].put(b)
```



- Type-safe structural composition using data layout types (Stream is a special case)
- Automatic type checking for deadlock, layout consistency, etc.

Constructing A Complete LLM Accelerator

- GPT2: single-batch, low-latency settings
  - U280 FPGA (16nm), 250MHz
  - 2.2x speedup in prefill stage compared to DFX [MICRO'22] (an FPGA-based overlay)
  - **In decode stage, 1.7x speedup and 5.4x more energy-efficient vs. A100** (7nm)
  - Fewer than 50 lines of schedule code in Allo



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

Acknowledgements

- This lecture contains/adapts materials developed by authors of the following papers
  - Allo: A Programming Model for Composable Accelerator Design (PLDI’24)
  - SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs (ICCAD’20)
  - HeteroCL: A Multi-Paradigm Programming Infrastructure for Sofware-Defined Reconfigurable Computing (FPGA’19)

Next Lecture

- Deep learning acceleration on FPGAs
  - Complete this reading assignment