Bridge of Life Education

Advanced SOC Design

Architecture Design Examples

Jiin Lai

Topics

- Architecture Framework
- Architecture Design Examples
  - Dynamic Pipeline
  - Large Matrix-Matrix Multiplication
  - Convolution 2D -> 1D
  - Optimal Architecture for Stencil Application
  - Systolic Array – Smith-Waterman DNA alignment algorithm
  - Interconnect Issues

Architecture Framework

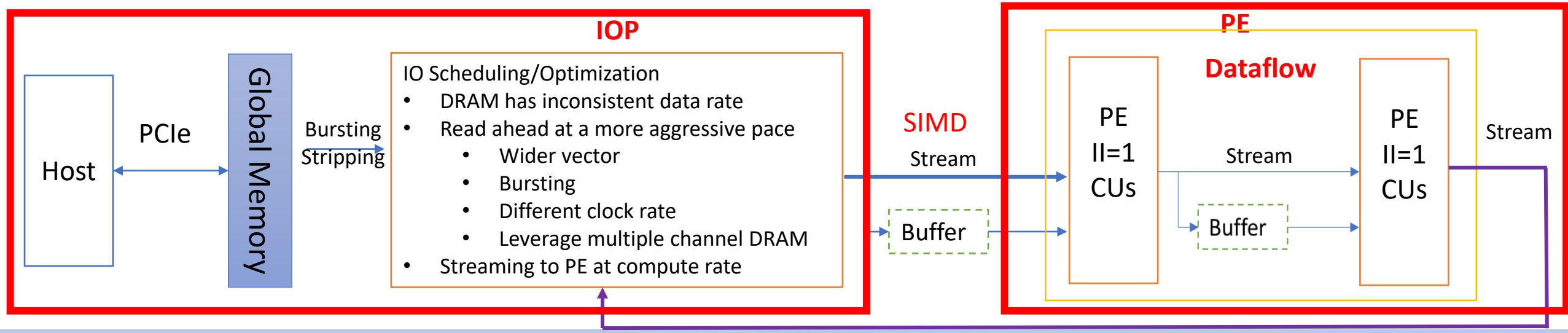
Three Optimization Goals

- Enable Pipelining with II = 1 (Tasks parallelism)
  - Ensure all pipelines run at maximum throughput
- Scaling/folding (Data Parallelism)
  - SIMD
- Memory / Data movement efficiency
  - Compute bound – saturate pipelines with data from memory
  - Memory bound – maximize bandwidth utilization
  - Roofline Model

Architecture Framework

Back

- Separate data movement (IOP) from Computation (PE)
- Optimize data movement first – use pseudo-design PE with II = 1
- Memory access extraction/transformation/optimization (**Polyhedral Analysis**)
- Streaming with Buffering/Cache for non-sequential access, e.g. window/line buffers
- Processing Stage (PE)
  - Multiple CUs (degree of folding, maximize data bandwidth SIMD)
  - Pipeline within PE (II = 1)
  - Dataflow among PEs
  - Systolic Array (Pipeline + Dataflow) - connectivity
- Design for Scalability (parameterized)



Best Practices for Designing with M_AXI Interfaces
https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Best-Practices-for-Designing-with-M_AXI-Interfaces

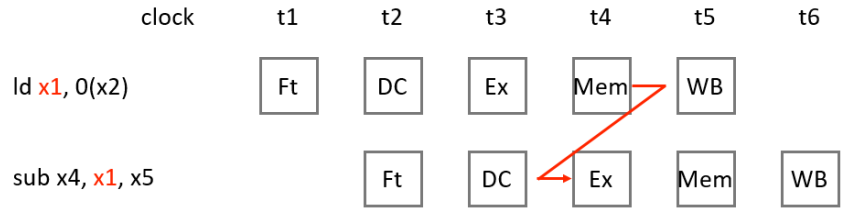
Dynamic Pipeline

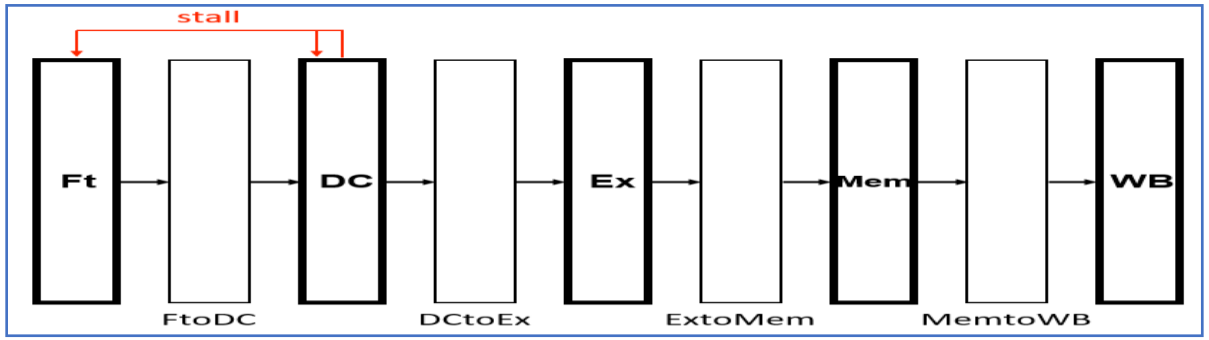
Dynamic Pipeline Techniques

- Dynamic Schedule – Example from RISC-V
- Speculative Loop Pipeline
- Loops with uncertain dependencies, Non-uniform dependencies

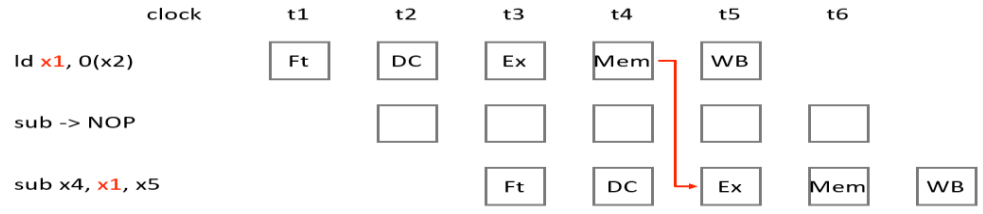
Dynamic Scheduling – RISC-V Stall

- Data hazards occur when an instruction needs the result of previous instruction, while the result of previous instruction is not yet available





- Data hazards can be solved by inserting NOP instruction(s) between the two instruction, which is so-called Pipeline Stall



Stall - From DC

```
void DC(struct DCtoEx* dctoEx,
        CORE_UINT(1) * dctoDc_stall,    // the self signal being sent to
                                        // DC() stage in the next cycle
        CORE_UINT(1) & dctoFt_stall)    // the signal being sent to Ft()
                                        // stage in the next cycle
{
    ...
    CORE_UINT(1) _dctoDc_stall_in = 0;
    CORE_UINT(1) _dctoFt_stall = 0;

    if (*prev_opCode == RISCV_LD &&   Check if load-use occur
        (dctoEx->rs1 == *prev_dest || dctoEx->rs2 == *prev_dest)) {
        _dctoDc_stall_in = 1;
        _dctoFt_stall = 1;            If load-use occur, the stall signals should be set to 1

        dctoEx->pc = 0;
        dctoEx->dataa = 0;            All the data of dctoEx should be set to 0,
        dctoEx->datab = 0;            as if the instruction is an NOP
        dctoEx->datac = 0;
        ...
    }
```

Stall - to Ft

```
void Ft(CORE_UINT(32) * pc,
        CORE_INT(32) ins_memory[MEM_SIZE / 4],
        struct FtoDC* ftoDC,
        CORE_UINT(1) _dctoFt_stall)
{
    CORE_UINT(32) next_pc = *pc;

    if (!_dctoFt_stall) {
        next_pc += 4;     If a stall occur, the pc should not be added
        *pc = next_pc;
        (ftoDC->instruction).SET_SLC(0, ins_memory[(*pc & 0x0FFFF) / 4]);
        ftoDC->pc = *pc;
    }
}
```

Intel HLS: speculative_iterations

Example: while (m*m*m < N) { m += 1; }

II can not less than the latency to calculate exit condition

Can not speculative if there is side effect, e.g. external memory access

Figure 19. Loop Orchestration Without Speculative Execution

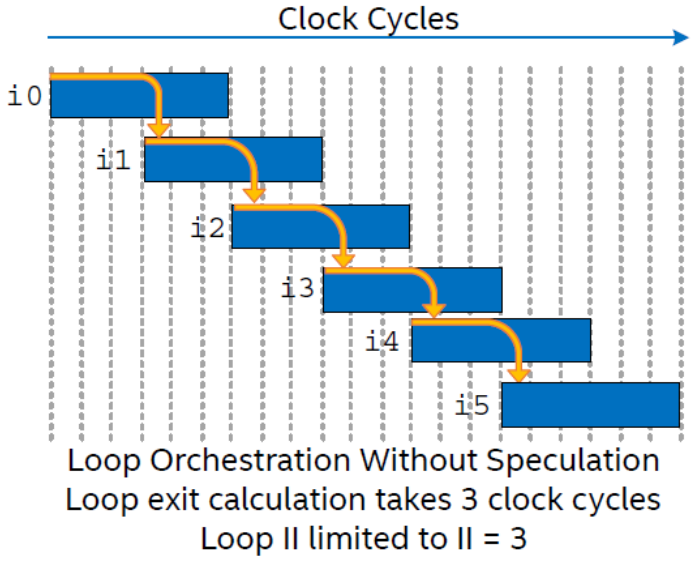
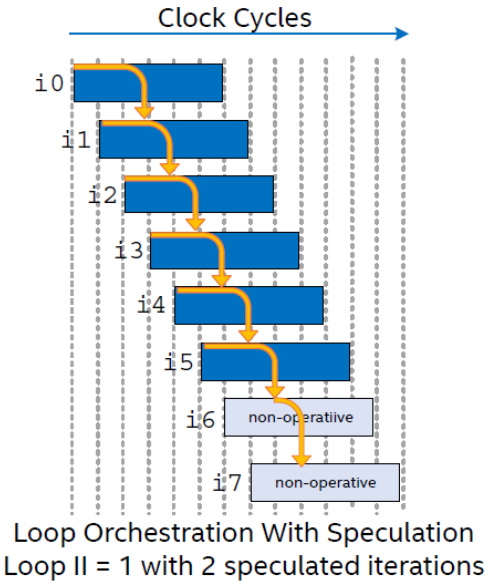


Figure 20. Loop Orchestration With Speculative Execution



Intel https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/hls/ug-hls-best-practices.pdf

Reference Manual: https://www.intel.com/content/www/us/en/programmable/documentation/ewa1462824960255.html#sfz1565797370740

Speculative Loop Pipeline

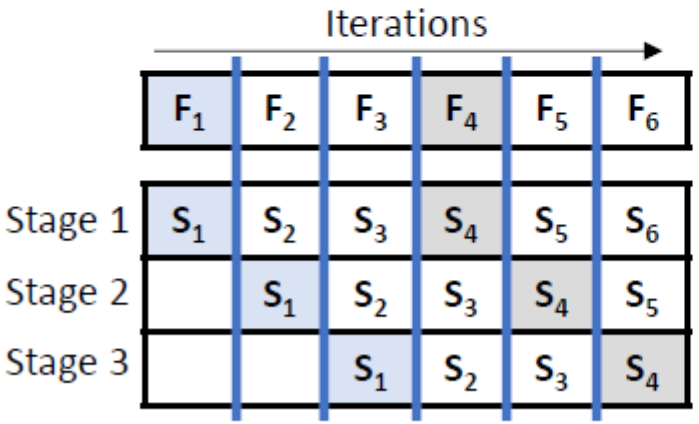
- Function S – 3 stages pipeline
- Function F – 1 stage pipeline
- What is latency of iteration?
- What is the II?

RecMII = max [Latency(c)/Distance(c)]

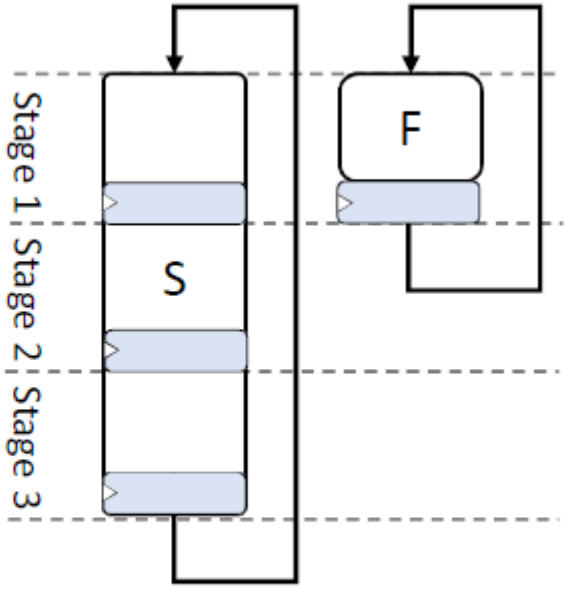
S: Latency = 3, Distance = 3
F: Latency = 1, Distance = 1
➔ II = 1

```
for(i=3;i<…;i++) {
  x[i]= S(x[i-3]);
  y[i]= F(y[i-1]);
}
```

Pipelined loop kernel



Pipelined execution trace



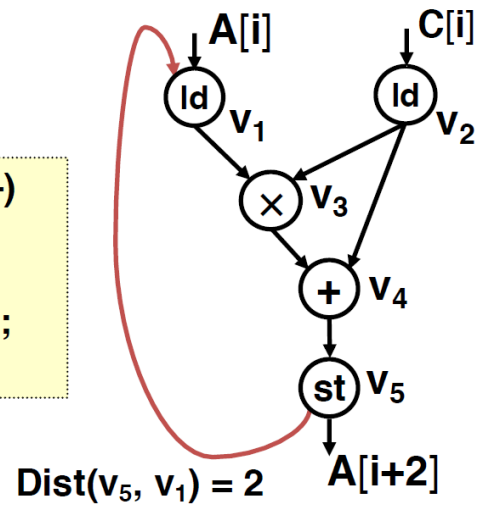
Pipelined datapath sketch

Speculative loop pipeline: https://hal.archives-ouvertes.fr/hal-02949516/document

Ref: Calculate II

- Dependency Graph
- Latency(c): sum of operation latencies along c
- Distance(c) = iterations separating the two dependent operations
- Ops(r) = # of operation for operator r
- Resource(r) = # of operator available
- ResMII – Minimum II due to Resource Limits **(Structure Hazard)**
  - ResMII = max [Ops(r)/Resource(r)]
  - r = operator
- RecMII – Minimum II due to Recurrences **(Data Hazard)**
  - **RecMII = max [Latency(c)/Distance(c)]**
  - **Optimize by reduce Latency(c), or increase Distance(c)**
- Min-II = max(ResMII, RecMII)

```
for (i = 0; i < N-2; i++)
{
    B[i] = A[i] * C[i];
    A[i+2] = B[i] + C[i];
}
```



Latency ->

Iteration->

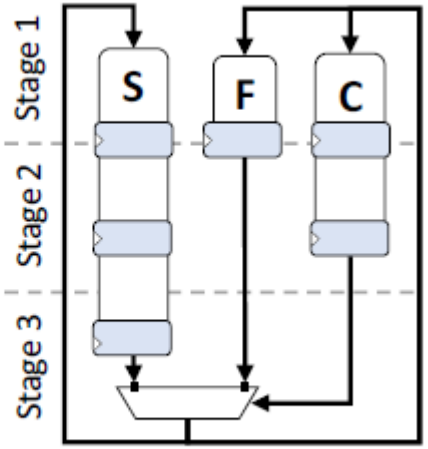
I/A0	*	*	+	s/A2	
	I/A1	*	*	+	s/A3
		I/A2	*	*	+

Loop with data-dependent control flow

- Control flow is data-dependent
- Actual execution path depends on the value of x, which only known at run time
- Application
  - Graph analytics,
  - Network packet inspection
  - Algorithm operating on sparse data structure

C() : latency = 2
S() : Latency = 3
F() : Latency = 1

```
do {
  if(C(x)) {
    // slow
    x = S(x);
  } else {
    // fast
    x = F(x);
  }
} while (!x)
```

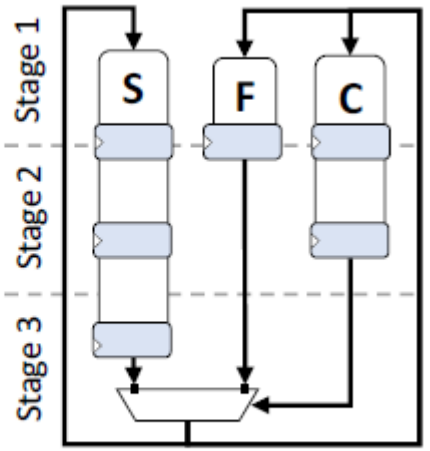


(a) Loop with data-dependent control flow.

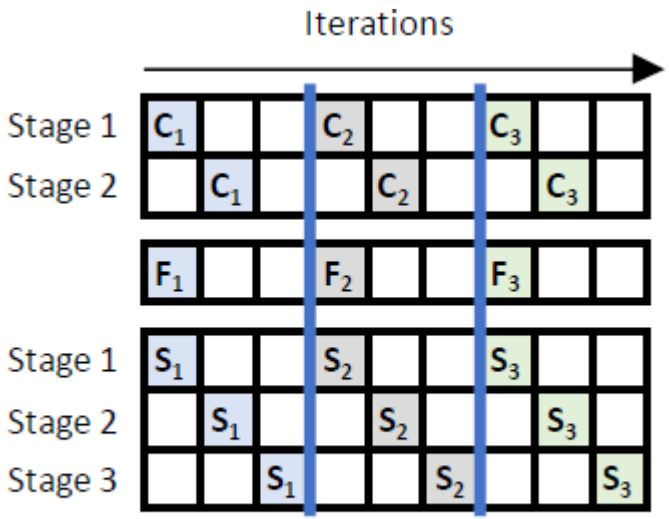
Loop with data-dependent control flow

C() : latency = 2
S() : Latency = 3
F() : Latency = 1

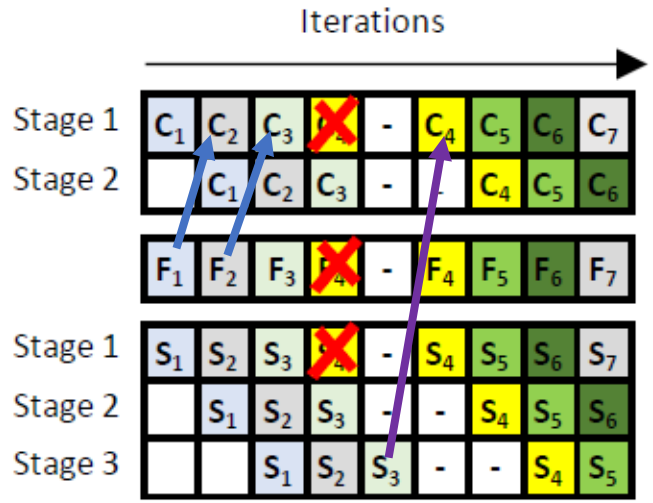
```
do {
  if(C(x)) {
    // slow
    x = S(x);
  } else {
    // fast
    x = F(x);
  }
} while (!x)
```



(a) Loop with data-dependent control flow.



(b) Static loop pipelining (II=3).



(c) Speculative pipelining (average II=1.4). Assume 20% miss-speculation rate

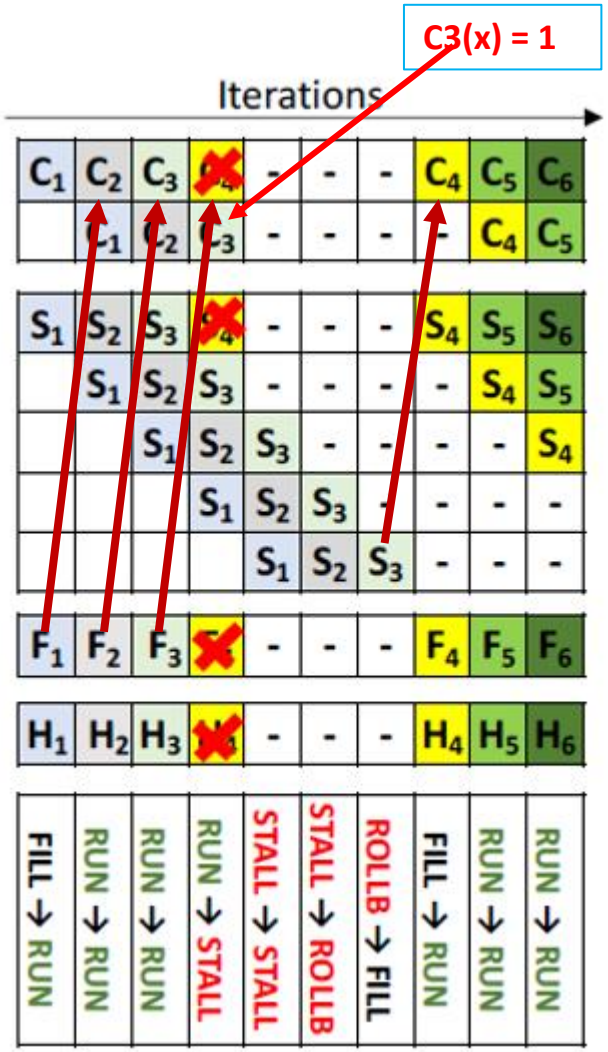
3 level of pipelining

- Static : II = 3
- Dynamic pipelining with stall – II = 2 (C(x) latency). If C(x) is true, stall one cycle to get S(x) result. II = 2.x
- Speculative pipelining – II = 1.x

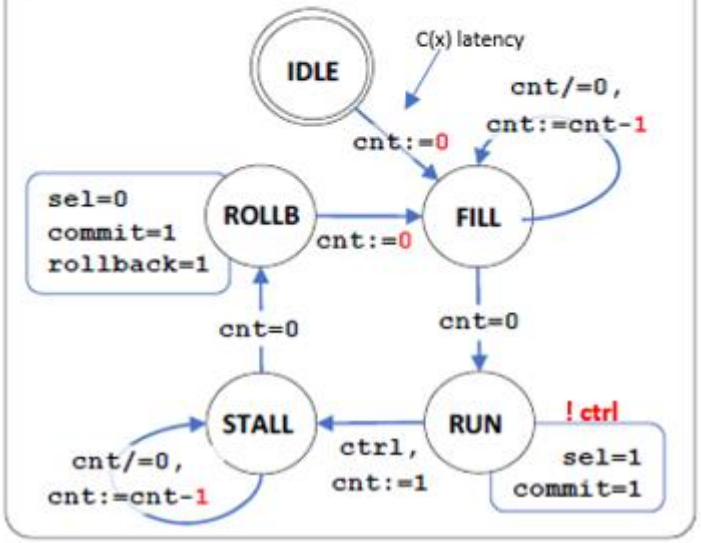
Speculative Pipeline Code Transformation

```
do {
  tmp=x;
  if(C(x)) {
    // slow
    x = S(tmp);
  } else {
    // fast
    x = F(tmp,y);
  }
  y = H(tmp,y)
} while (!x)
```

Latency estimates	
C	2
S	5
F	1
H	1




```
enum tstate = {IDLE, FILL, … }

struct fsm {
   int3 cnt;
   tstate cs;
   bool commit,rollback,sel;
} cs;
```

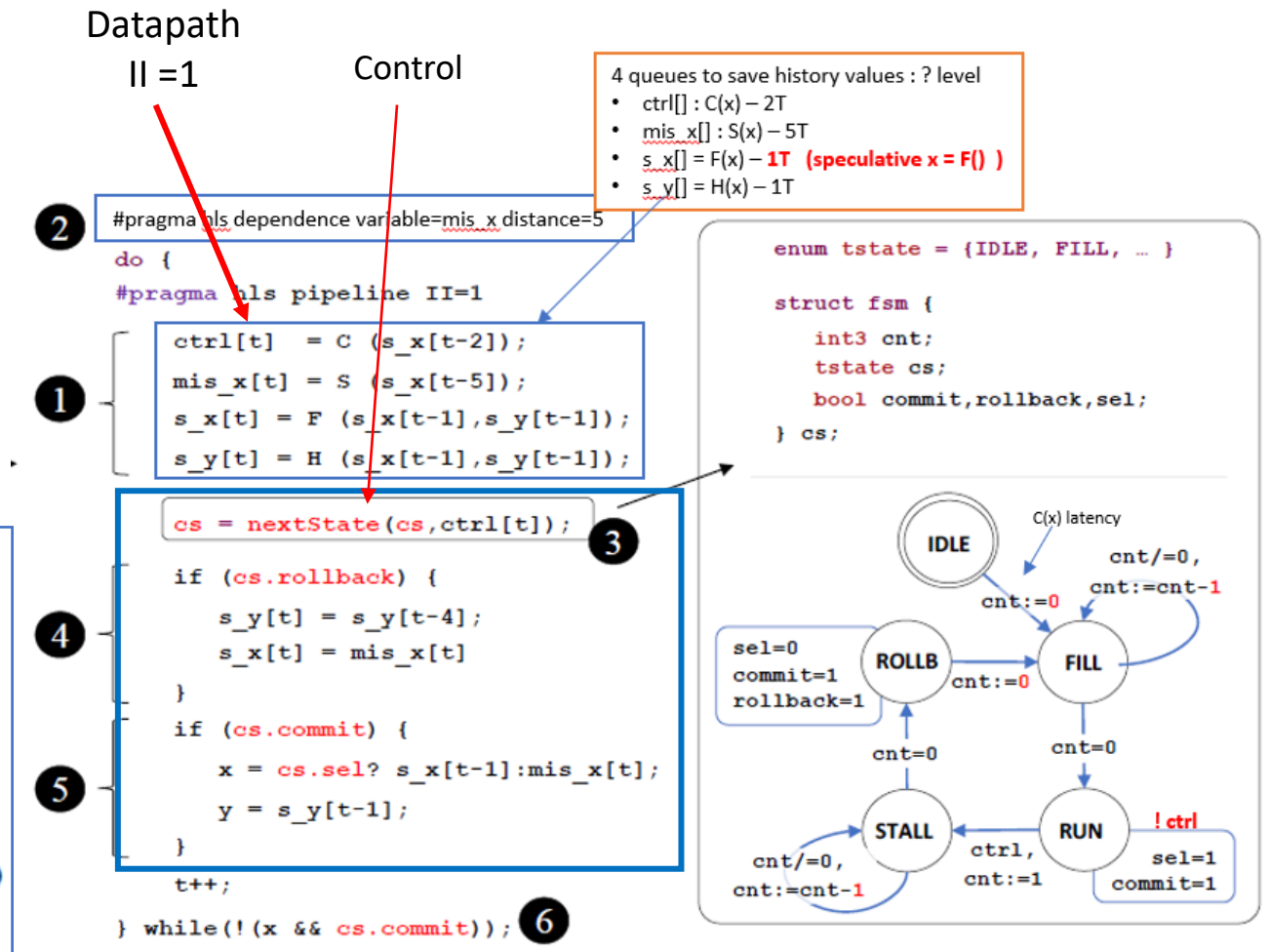


RUN -> STALL * 2 -> ROLLB

Speculative Pipeline Code Transformation

- Decouple the control logic from the datapath in the source code
- Increase the dependence distance exposed in the datapath
  - S uses s_x[t-5], S takes five cycles to produce mis_x[t]
- Use circular buffer, its level is determined by pipeline latency
- State machine
  - FILL: Startup, C-function Latency
  - RUN: Speculative Pipeline Ready
  - STALL: mis-speculation recovery
  - ROLLB: recover states

```
do {
  tmp=x;
  if(C(x)) {
    // slow
    x = S(tmp);
  } else {
    // fast
    x = F(tmp,y)
  }
  y = H(tmp,y)
} while (!x)
```

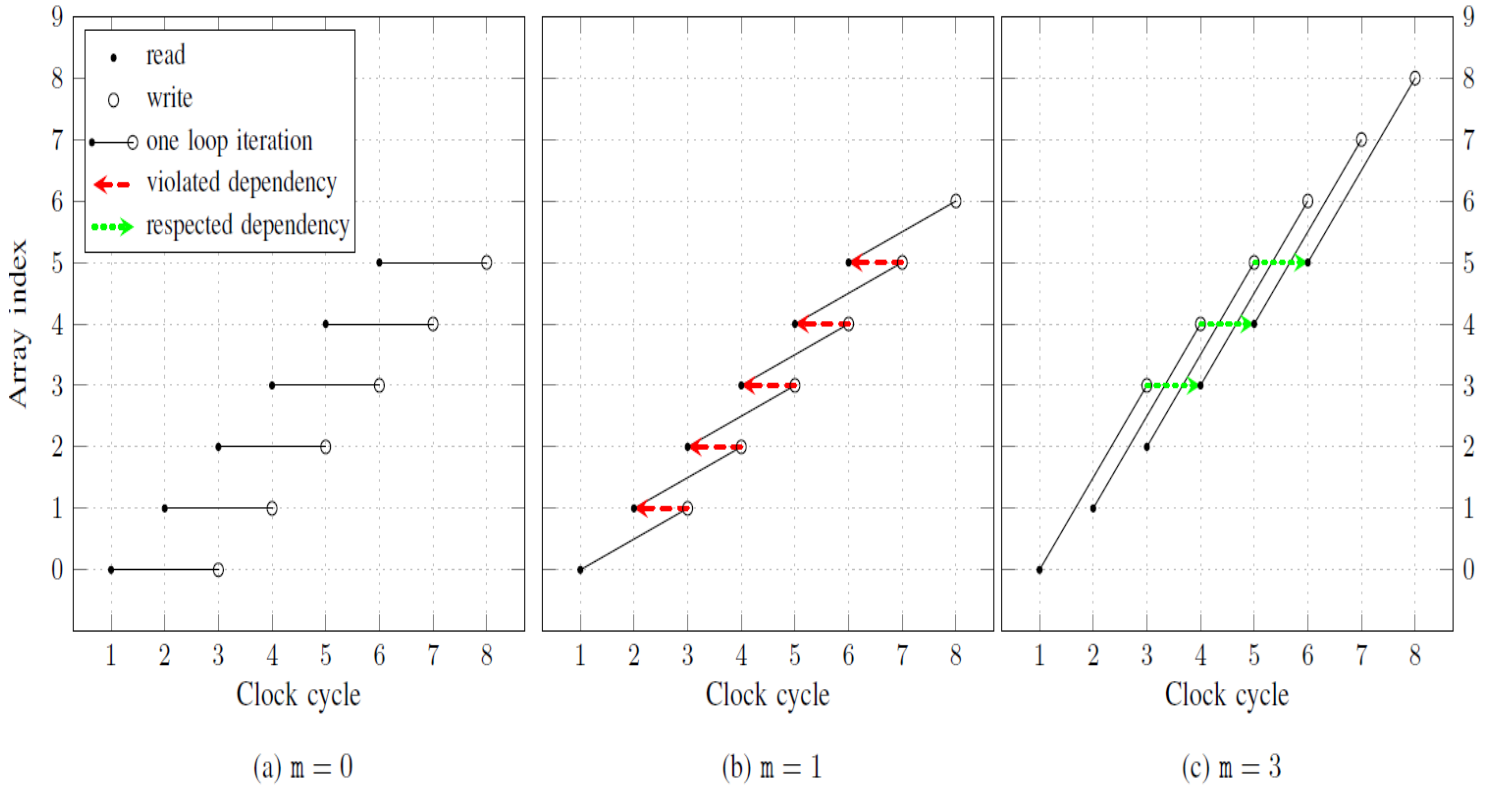


Uncertain and non-uniform memory dependencies

- Uncertain Dependency

```
//uncertain dependency
for (i=0; i<N; i++)
   A[i+m] = A[i] + 0.5f;
```

- Value of m is not known at compile time
- Break the pipeline execution at iteration (i=m) to resolve the RAW conflict
- Example: Matrix decomposition, triangular matrix computation



Iteration latency = 3

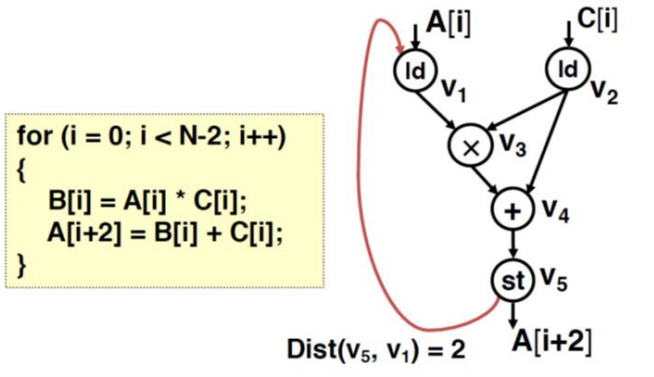
- Non-uniform dependency

```
// non-uniform dependency
for (i=0; i<N; i++)
A[2*i] = A[i] + 0.5f;
```

Polyhedra-based Dynamic Loop Pipelining: https://johnwickerson.github.io/papers/looppipelining_TCAD17.pdf

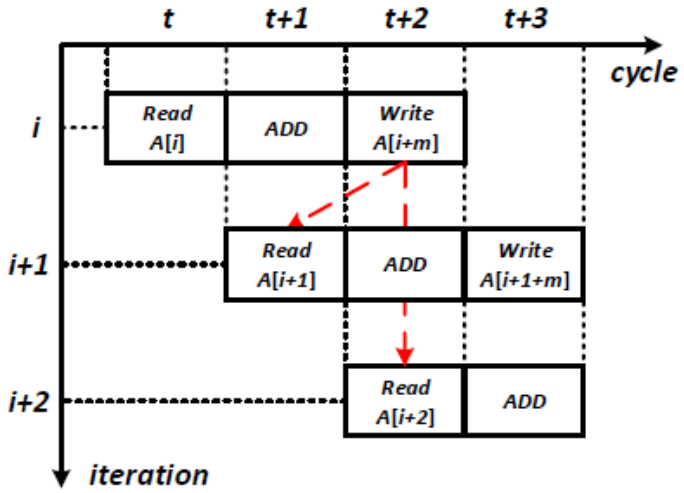
- L : period when the execution of the dependent read access will violate the iteration memory dependency
- d(m, i): dependence iteration distance – smallest number of iteration between the execution of two such dependent data access

$$1 \leq d(\mathtt{m}, \mathtt{i}) \leq \left\lceil \frac{L}{II} \right\rceil - 1$$

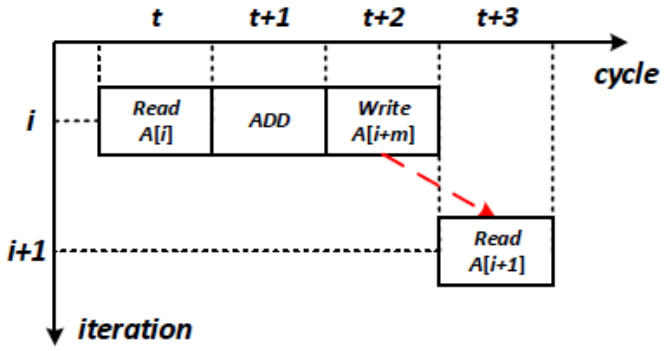


RecMII = max [Latency(c)/Distance(c)]

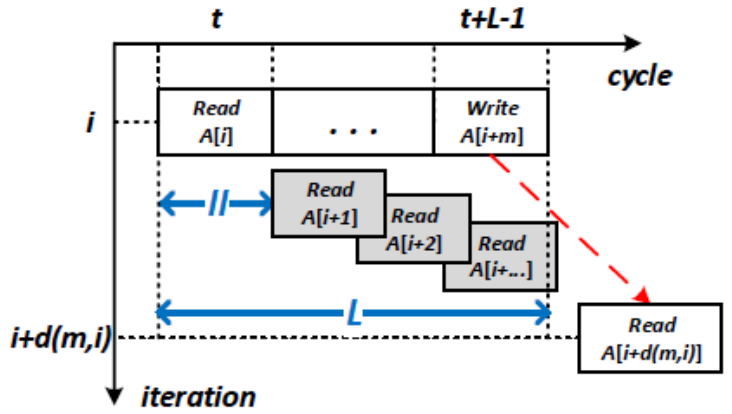
- L = 3; II = 1 => 1<= m <= 2



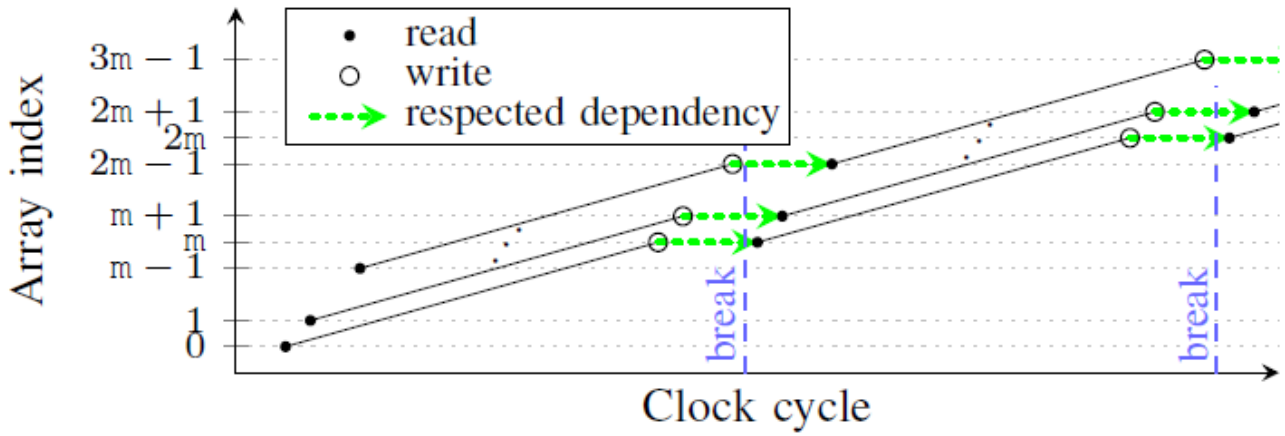
(a) Unsafe II.



(b) Conservative and safe II.



(c) The conflict region of $d(\mathtt{m}, \mathtt{i})$.



Breaking the pipeline execution at (m+1)th and (2m+1) iteration

```
//uncertain dependency
for (i=0; i<N; i++)
   A[i+m] = A[i] + 0.5f;
```

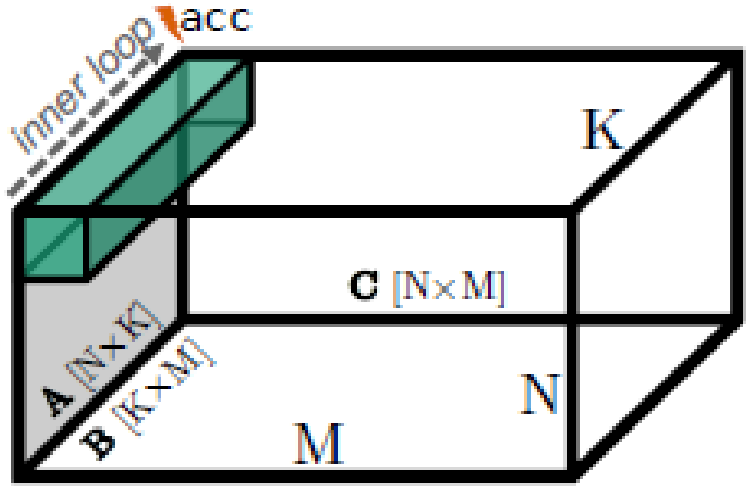
```
// Conflict region detection with L=3
if ( m >= 1 && m <= 2) {
   // Split execution
   for (k=0; k<N; k=k+m)
      // inner loop: force pipelining with II=1
      for (i=k; i<=min(N-1,k+m-1); i++)
         A[i+m] = A[i] + 0.5f;
}
else {
   // Fast execution
   // force pipelining with II=1
   for (i=0; i<N; i++)
      A[i+m] = A[i] + 0.5f;
}
```

Matrix-Matrix-Multiplication – MMM (Large-size)

Problem Statement: Large MMM

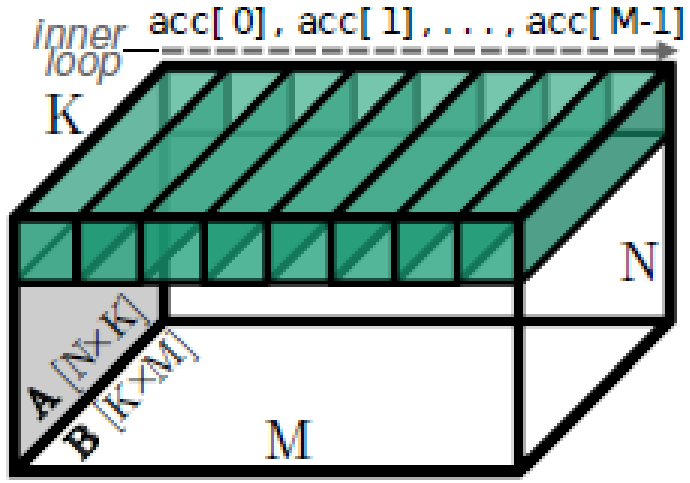
- Loop-carried dependency
  - 64-bit double operation for multiple/add takes multiple cycles to complete ($L_{+,double} > 1$)
  - ➢ Loop Transformation + Interleaving
- Memory IO & on-chip buffer limitation
  - Poor OI (operation intensity)
  - Can not fit row/column in on-chip buffer
  - ➢ IO scheduling + Cache (block/striping/tiling)
- Resources – Routing
  - N degree of parallelism -> 1-to-N and N-to-1 connection
  - ➢ Dataflow (Systolic Array)

```
// A[N][K] x B[K][M] ; A,B,C : double
for (int n = 0; n < N; ++n)
  for (int m = 0; m < M; ++m) {
     double acc = C[n][m];
     for (int k = 0; k < K; ++k) {
       #pragma PIPELINE
       acc += A[n][k] * B[k][m]; }
     C[n][m] = acc;
}  }
```



Loop Transformation – Accumulation Interleave

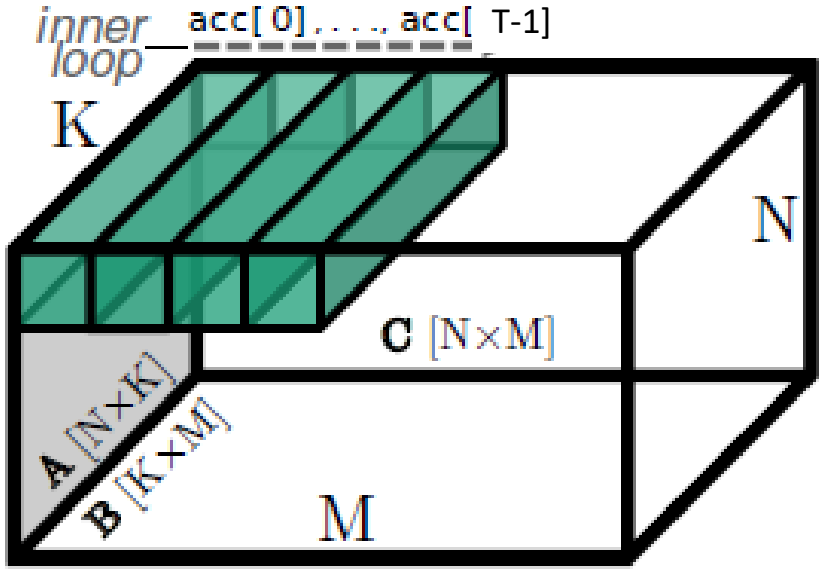
- Accumulation buffer ***acc[M]***
- Loop-carried dependency resolved - each location is only update every M cycle (M > L)
- A, B, C are all read in a contiguous fashion (assume row-major memory layout) – what if column-major ?)
- Each element of A is read exactly once
- No need to initialize Accumulation buffer




```
for (int n = 0; n < N; n++) {
   double acc[M];   // Uninitialized
   for (int k = 0; k < K; k++) {
      double a = A[n][k];  // only read once
      for (int m = 0; m < M; m++) {
         #pragma HLS PIPELINE
         double prev = (k == 0) ? C[n][m] : acc[m];
         acc[m] = prev + a * B[k][m];
      }
   }
   for (int m = 0; m < M; m++)  // write out
      c[n][m] = acc[m];
}
```

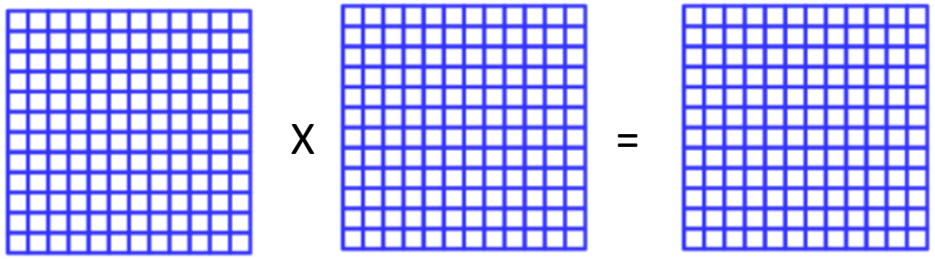
1D Tiled Accumulation Interleave

- For M >>, to save accumulation buffer
- Choose Tile T >= L
- A[n][k] need to read M/T times
- Adding an additional inner loop over the tile




```
for (int n = 0; n < N; n++) {
   for (int m = 0; m < M; m += T) {
      double acc[T]; //Tile of size T
      for (int k =0; k < K; k++) {
         double a = A[n][k]        // M/T read
         for (int t = 0; t < T; t++) {
            #pragma HLS PIPELINE
            double prev = (k == 0) ? C[n][m + t] : acc[t];
            acc[t] = prev + a * B[k][m + t];
      }  }
      for (int t = 0; t < T; t++)  // write out
        C[n][m + t] = acc[t];
}  }
```

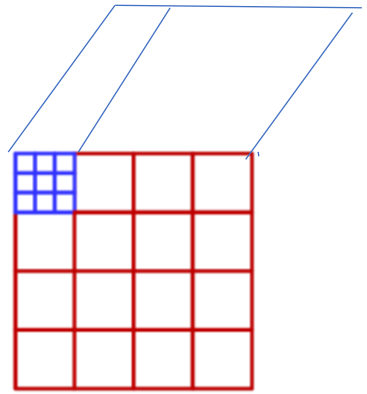
Memory IO – Poor Operation Intensity



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

- N is large, s.t. 1 row/col too large for on-chip
- Operation count: N^3 mul, N^3 add => 2 * N^3
- External memory access (4-byte int)
  - 2*N^3 4-byte read (A,B) from DRAM)
  - .. N^2 4-byte write (C) to DRAM
- Operation Intensity ~ 2*N^3 / (4 * 2 * N^3)=**1/4**

```
for(i0=0; i0<N; i0+=Nb)
  for(j0=0; j0<N; J0+=Nb)
    for(k0=0; k0<N; k0+=Nb) {
BlockMatrix: {
      for(i=i0; i< i0+Nb; i++)
        for(j=j0; j<j0+Nb; j++)
          t=C[i][j];  // RAW dependency
          for(k=k0; k<k0+Nb; k++)
            t += A[i][k] * B[k][j];
          C[i][j] = t;
}     }
```

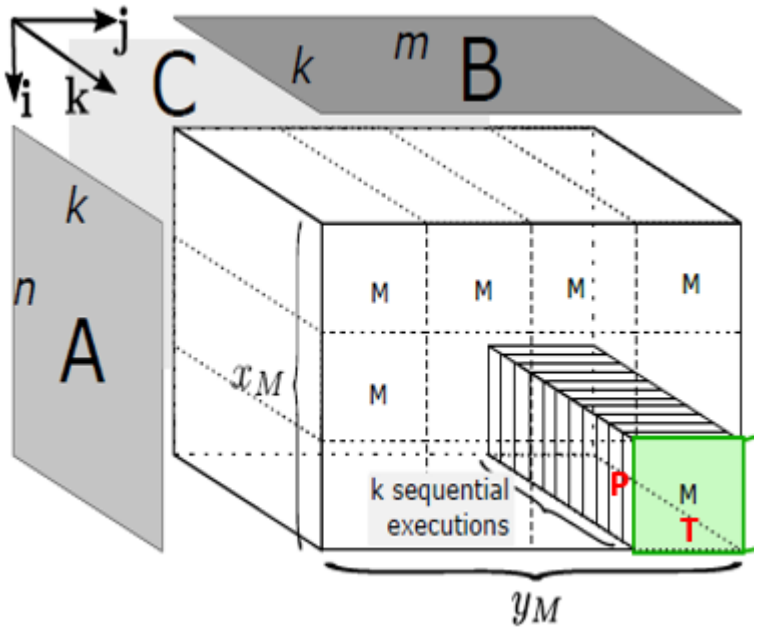


Block Matrix

- Array of Nb can fit into on-chip
- Opeartion count: Nb^3 mul, Nb^3 add => 2 * Nb^3
- Exernal memory access (4-byte int)
  - 2*Nb^3 4-byte on-chip read (A,B) – fast ignore
  - 3Nb^2 4-byte off-chip DRAM read (A, B, C) – slow
  - Nb^2 4-byte off-chip DRAM write (C)
- Operation Intensity ~ 2*Nb^3 / (4 * 4 Nb^2)=**Nb/8**

Vertical Folding

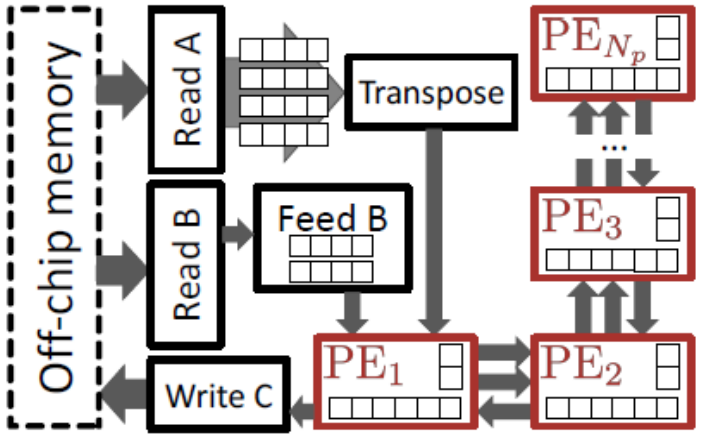
- P - # of PE, T - Tile size
- 2D accumulation buffer: acc[T][P]
- Separate A array reading, loop reading A, concurrently with B




```
for (int n = 0; n < N; n += P) {  // unrolling factor P
  for (int m = 0; m < M; m+=T) { // Tiling T
    double acc[T][P];  // Is now 2D
    // ... initialize acc from C ...
    for (int k = 0; k < K; ++K) {
      double a_buffer[P];          // Buffer multiple elements
      for (int p = 0; p < P; p++) {    // combine with incoming
        #pragma HLS PIPELINE       // values of
        a_buffer[p] = A[n + p][k];  } // B in parallel
      for (int t = 0; t < T; t++) { // stream tile of B
        for(int p = 0; p < P; p++) { // P-fold vertical unrolling
          #pragma HLS UNROLL
          acc[t][p] += a_buffer[p] * B[k][m + t];
      } }
} }
```

Memory/IO Scheduling + Dataflow/Systolic

- Separate I/O scheduling and Compute
- Scalable (parameterized) – Subject to FPGA Compute & Memory Resource
  - Nc – Number of Compute Unit
  - Nb – Number of BRAM
- Fit FPGA connection by dataflow/systolic architecture
- Double buffer to parallelize compute and data movement



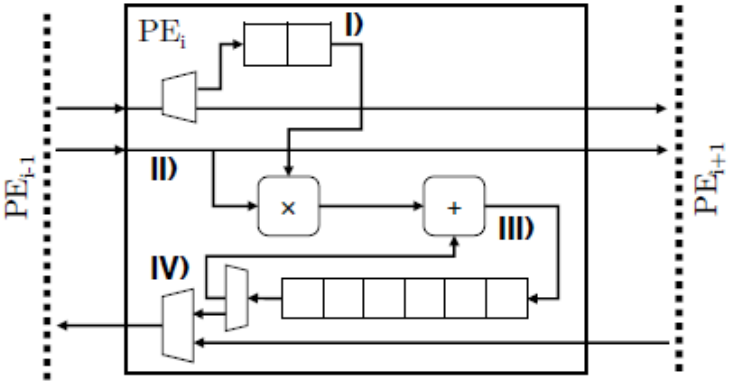
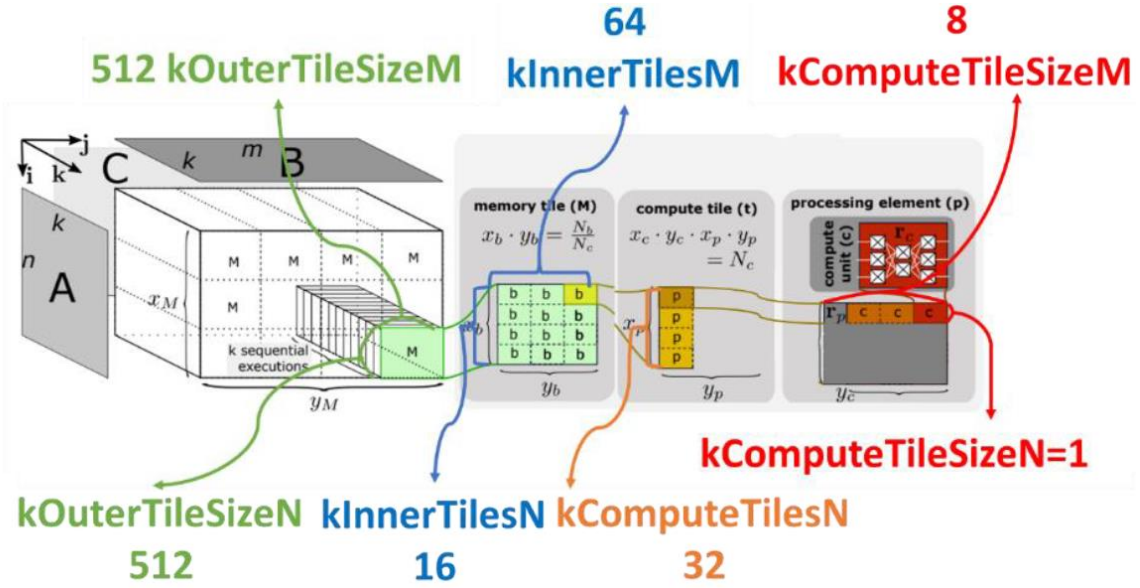


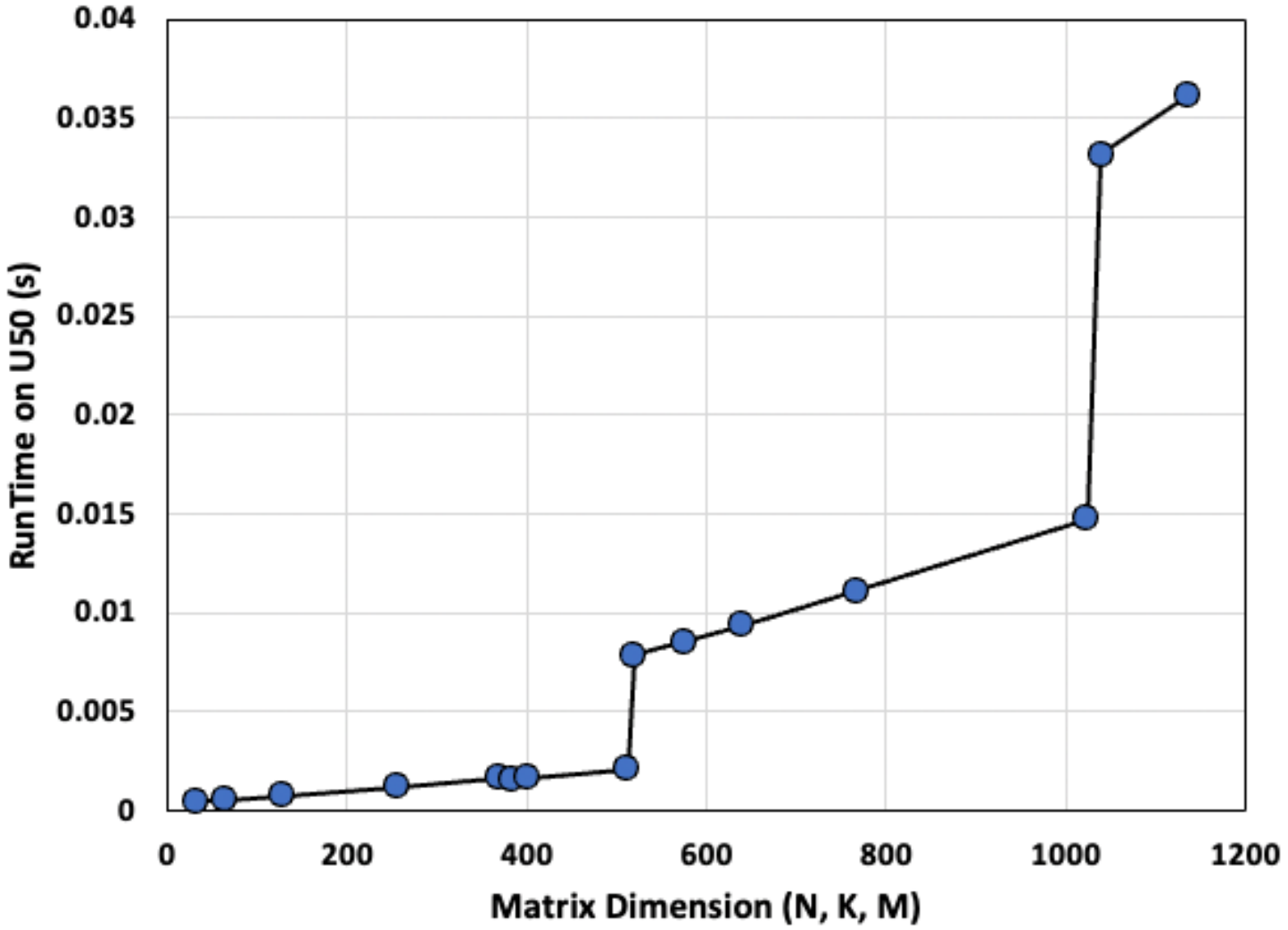
Figure 6: Architecture of a single PE.



Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis: https://arxiv.org/pdf/1912.06526
Github: https://github.com/spcl/gemm_hls
PPT: https://github.com/bol-edu/2022-spring-nthu/blob/main/Final/gemm_hls/Final/Group8_Final_Presentation.pptx

Empirical Result - Runtime

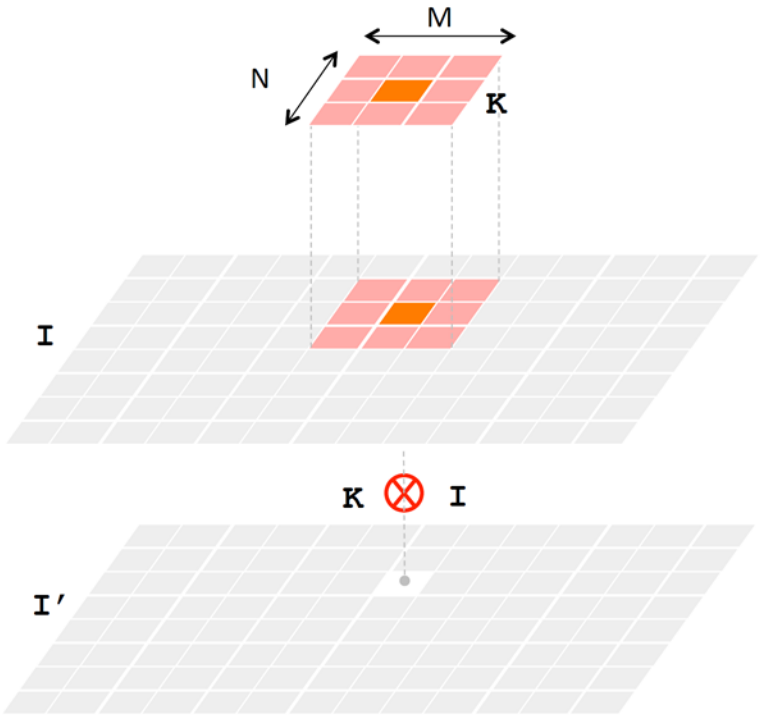
Back



Convolution

2D Convolution with Line Buffer

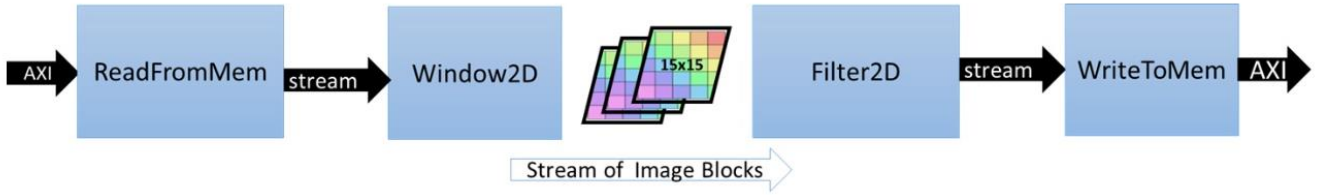
- Output image I' = K x I, sliding the kernel window K over all the input image
- CNN has multiple layers of convolution filters
- Techniques:
  - Minimize data input read – use local cache
  - Minimize assess arrays – use local cache to hold results and write the final result to the array
  - Perform conditional branching inside pipelined tasks rather than conditionally execute tasks
  - Minimize output write
  - Using hls::stream enforce good coding practices

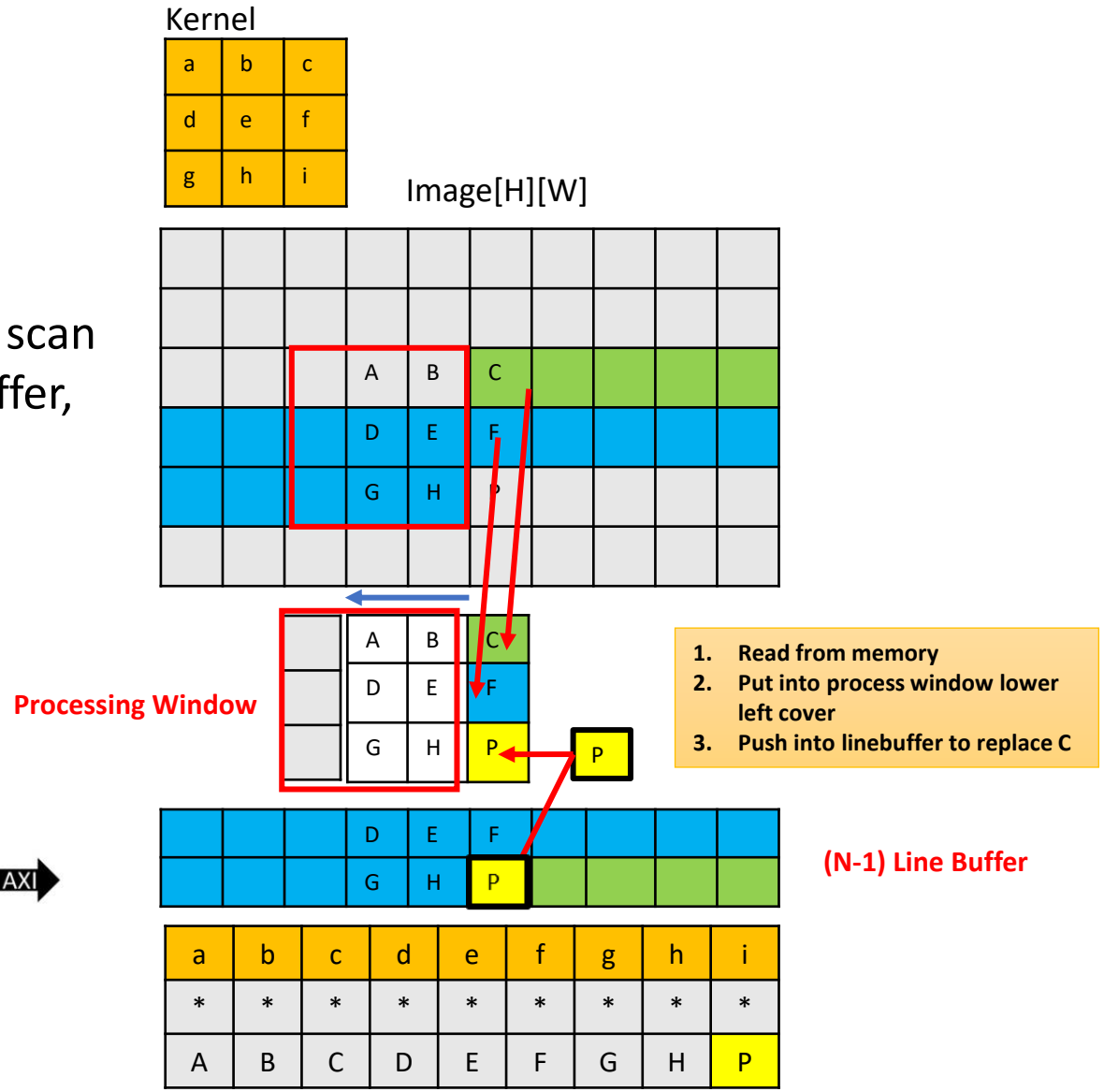


https://github.com/Xilinx/HLS-Tiny-Tutorials/tree/master/algorithm_2D_convolution_linebuffer
https://casr.eee.hku.hk/publication/swim-fpt17/swim-fpt17.pdf

2D convolution

- Operation
  - Read input pixel (prefetched, and stream-in) – raster scan
  - Shift processing window with N-1 value from line-buffer, and input pixel
  - Update line-buffer
  - Dot product
  - Output update

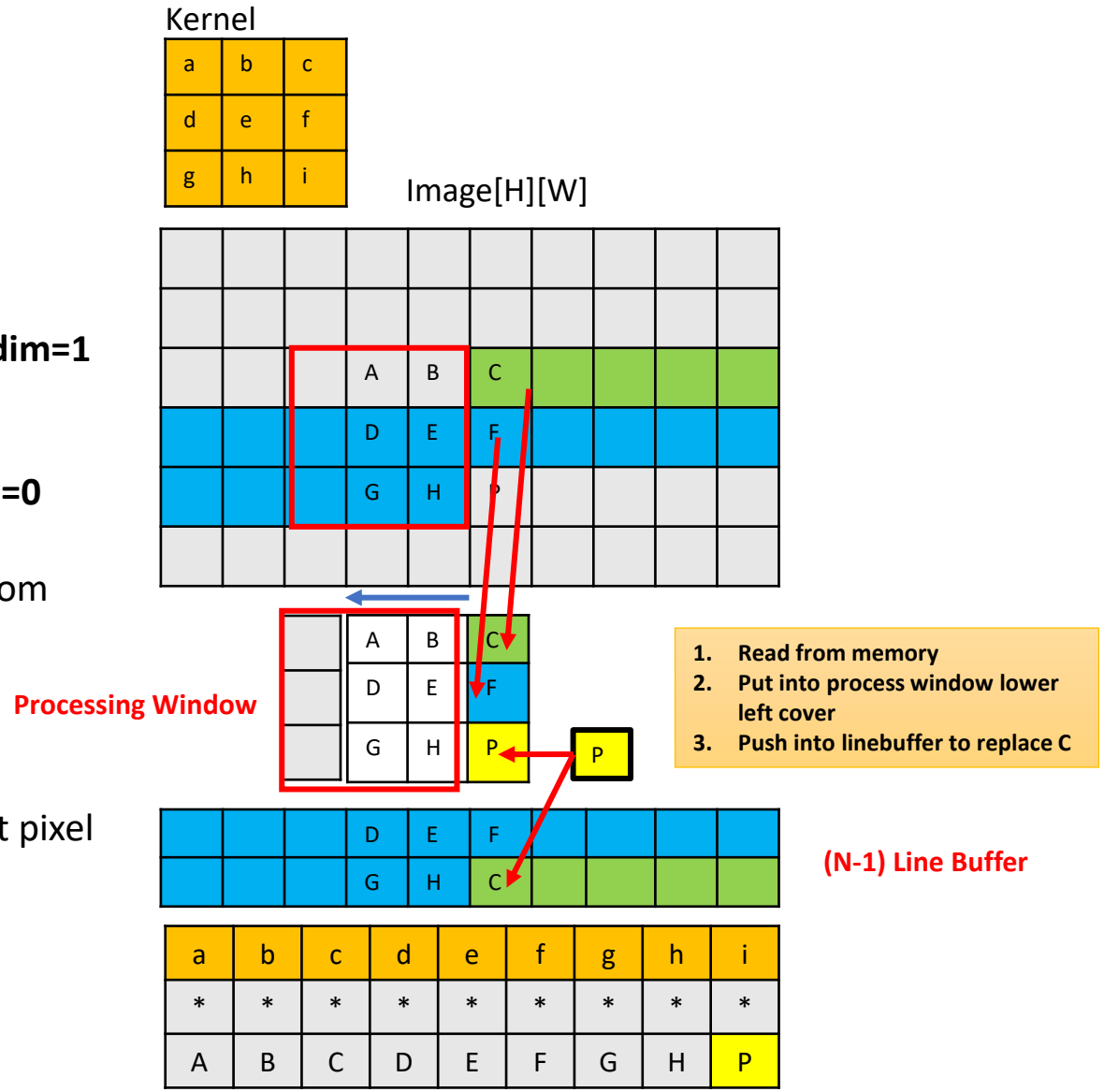




https://github.com/Xilinx/Vitis-Tutorials/blob/2021.2/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial/src/filter2d_hw.cpp

2D convolution

- Image[H][W] (off-chip memory)
- Kernel [N][M] (on-chip, constant memory)
  - #pragma HLS ARRAY_PARTITION variable=kernel complete dim=0
- Line_buffer [N-1][W] (on-chip)
  - **#pragma HLS ARRAY_PARTITION variable=line_buffer compelte dim=1**
  - (N-1) lines, (N-1) BRAM or (N-1) port
- Processing window [N][M] (on-chip)
  - **#pragma HLS ARRAY_PARTITION variable=window complete dim=0**
  - Shift registers
  - Right-most column filled with (N-1) Line + Current pixel (P) read from memory)
- Operation
  - Read input pixel (prefetched, and stream-in)
  - Update line-buffer
  - Shift processing window with N-1 value from line-buffer, and input pixel
  - Dot product
  - Output update

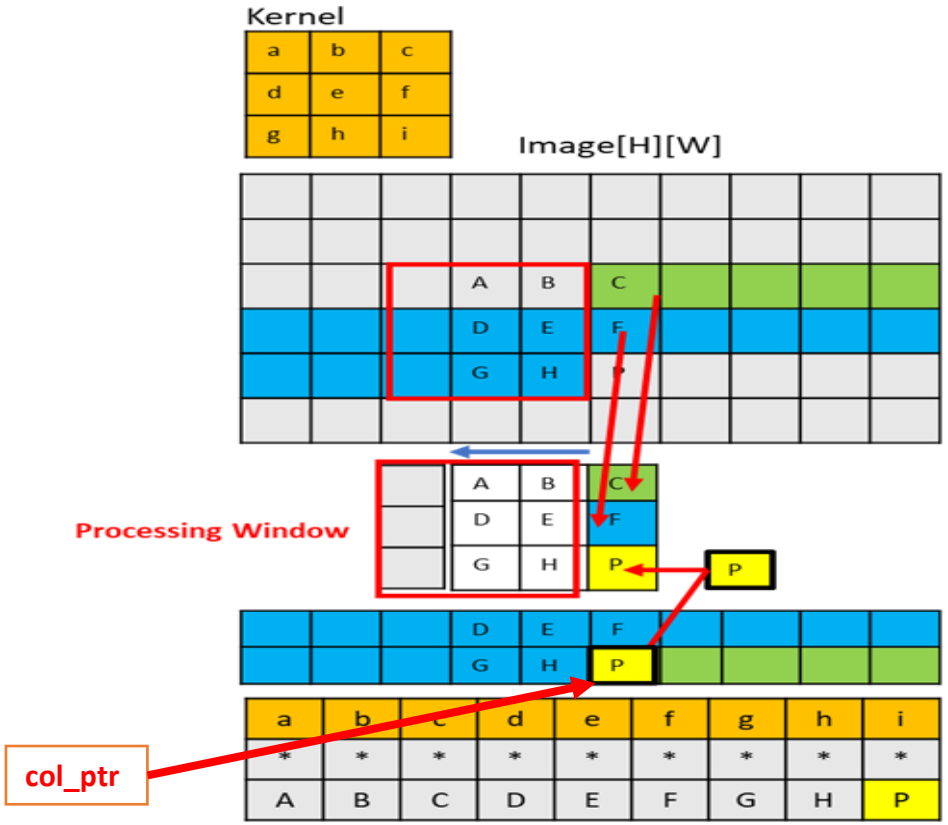


https://github.com/Xilinx/Vitis-Tutorials/blob/2021.2/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial/src/filter2d_hw.cpp

Window_2D

```
struct window {
   U8 pix[FILTER_V_SIZE][FILTER_H_SIZE];
};
```

```
// Line buffers - used to store [FILTER_V_SIZE-1] entire lines of pixels
   U8 LineBuffer[FILTER_V_SIZE-1][MAX_IMAGE_WIDTH];
#pragma HLS ARRAY_PARTITION variable=LineBuffer dim=1 complete
```




```
 update_window: for (int n=0; n<num_iterations; n++)  {
#pragma HLS PIPELINE II=1
      // Read a new pixel from the input stream
      U8 new_pixel = (n<num_pixels) ? pixel_stream.read() :

      // Shift the window and add a column of new pixels from the line buffer
      for(int i = 0; i < FILTER_V_SIZE; i++) {
         for(int j = 0; j < FILTER_H_SIZE-1; j++) {
            Window.pix[i][j] = Window.pix[i][j+1];    // shift left
         }
         Window.pix[i][FILTER_H_SIZE-1] = (i<FILTER_V_SIZE-1) ?
                              LineBuffer[i][col_ptr] : new_pixel;   // append new pixel
      }

      // Shift pixels in the column of pixels in the line buffer, add the newest pixel
      for(int i = 0; i < FILTER_V_SIZE-2; i++) {
         LineBuffer[i][col_ptr] = LineBuffer[i+1][col_ptr];  // shift up one line
      }
      LineBuffer[FILTER_V_SIZE-2][col_ptr] = new_pixel;  // append new pixel

      window_stream.write(Window);
}
```


https://github.com/Xilinx/Vitis-Tutorials/tree/2021.2/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial

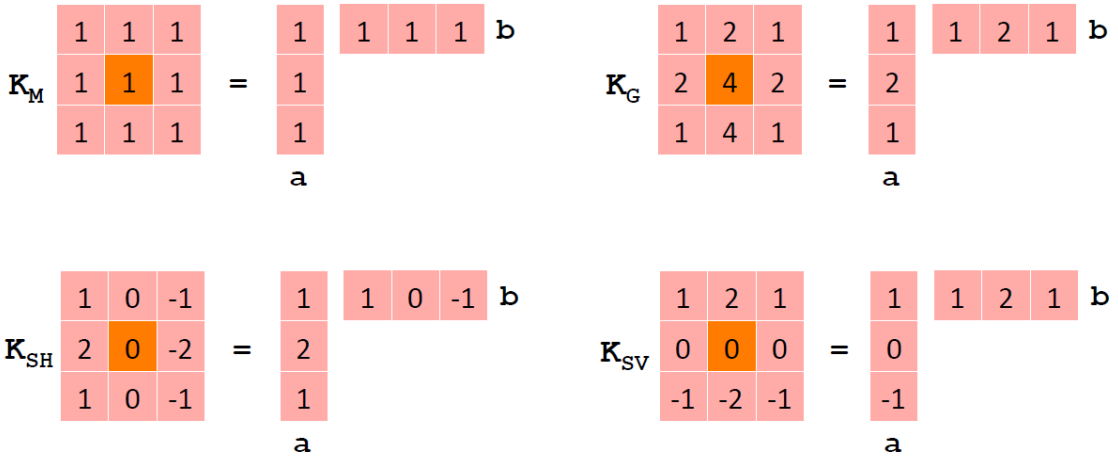
Filter_2D

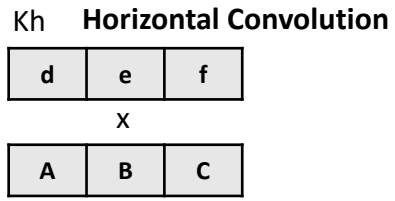
```
apply_filter: for (int y = 0; y < height; y++)
  {
    for (int x = 0; x < width; x++)
    {
#pragma HLS PIPELINE II=1
      // Read a 2D window of pixels
      window w = window_stream.read();

      // Apply filter to the 2D window
      int sum = 0;
      for(int row=0; row<FILTER_V_SIZE; row++)
      {
        for(int col=0; col<FILTER_H_SIZE; col++)
        {
          unsigned char pixel;
          int xoffset = (x+col-(FILTER_H_SIZE/2));
          int yoffset = (y+row-(FILTER_V_SIZE/2));
          // Deal with boundary conditions : clamp pixels to 0 when outside of image
          if ( (xoffset<0) || (xoffset>=width) || (yoffset<0) || (yoffset>=height) ) {
            pixel = 0;
          } else {
            pixel = w.pix[row][col];
          }
          sum += pixel*(char)coeffs[row][col];
        }
      }
      unsigned char outpix = MIN(MAX((int(factor * sum)+bias), 0), 255);
      pixel_stream.write(outpix);   // // Write the output pixel
```

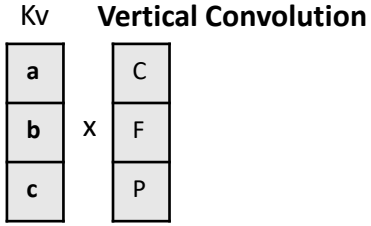
```
struct window {
   U8 pix[FILTER_V_SIZE][FILTER_H_SIZE];
};
```

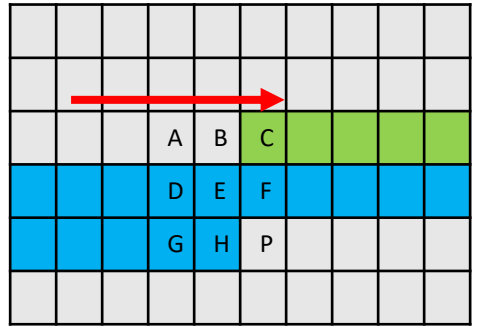
Convert 2D Convolution to 1D Convolution

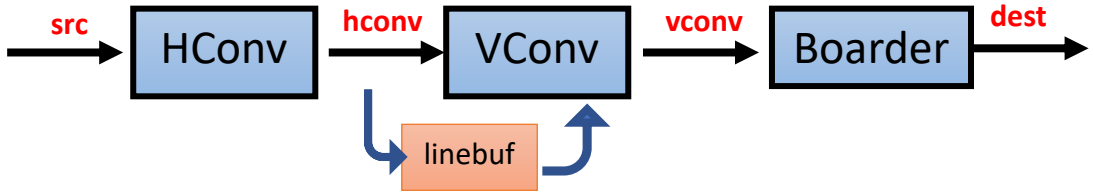
- Separable – 2D filter K = v * h
- Associativity : (v * h) * I = v * (h * I)
- Computation Reduction
  I[W][H], K = [M][N], K * I: WxHxMxN
  v * (h * I) = (WxHxM) + (WxHxN)
  Reduction = (M+N)/(MxH) if 5x5 => 2/5
- Stream interface: Horizontal Conv (strm hconv) -> Vertial Conv (strm vconv) -> Border
- Not always equivalent result, simple and efficient
- May not be faster than 2D if not implement correctly





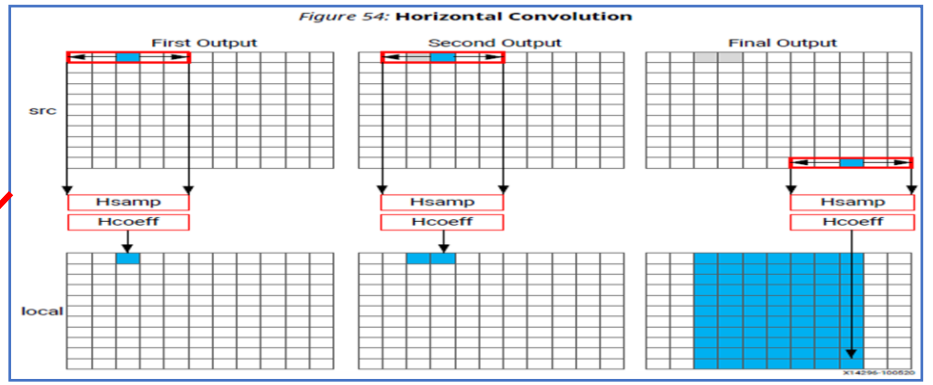


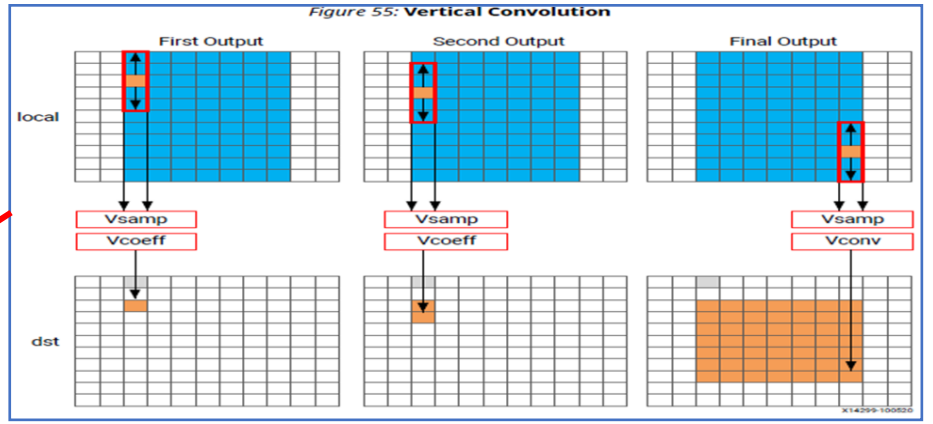


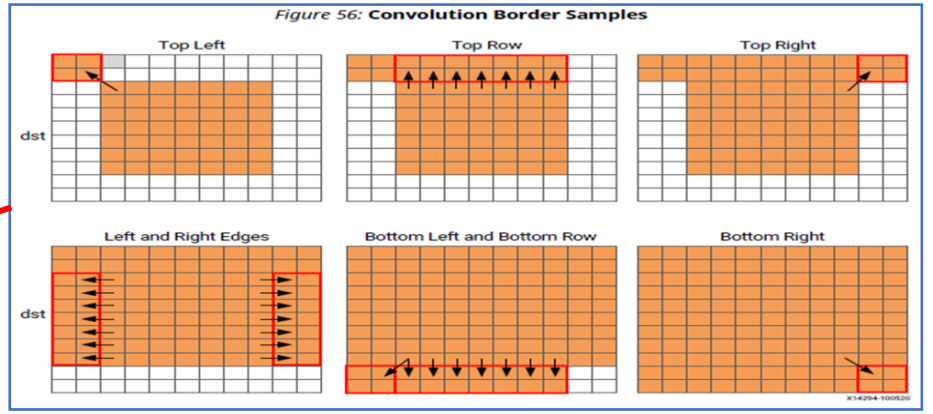


1D Convolution Code Example - Naive

```
template<typenae T, int K>
static void convolution_orig(
            int width, int height,    // picture dimension
            const T *src,  T *dst,    // input, output picture array
            const T *hcoeff, *vcoeff );  // kernel  {
      const int conv_size = K;
      const int border_width = int(conv_sie/2);

T local[MAX_IMG_ROWS*MAX_IMG_COLS];

// Horizontal convolution
      HConvH:for(int row = 0; rol < height; row++) {
        HconvW:for(int col = border_width; col < width - border_width; col++) {
            Hconv:for(int i = -border_width; i <= border_width; i++) { .. } } }
// Vertical convolution
      VconvW:for(int col = border_width; col < width - border_width; col++) {
        VconvH:for(int row = bolder_width; row < height - border_width; row++) {
            Vconv:for(int i = -border_width; i <= border_width; i++) { … } } }

// Border pixels
      // 8-loops for ur-right, upper, up_left, right, left, bot-left, bot-, bot-right
```







https://github.com/jefflieu/HLS-Tiny-Tutorials/tree/master/algorithm_2D_convolution_linebuffer

Xilinx ug1399: chapter 16: Examples of Hardware Efficient C++ Code

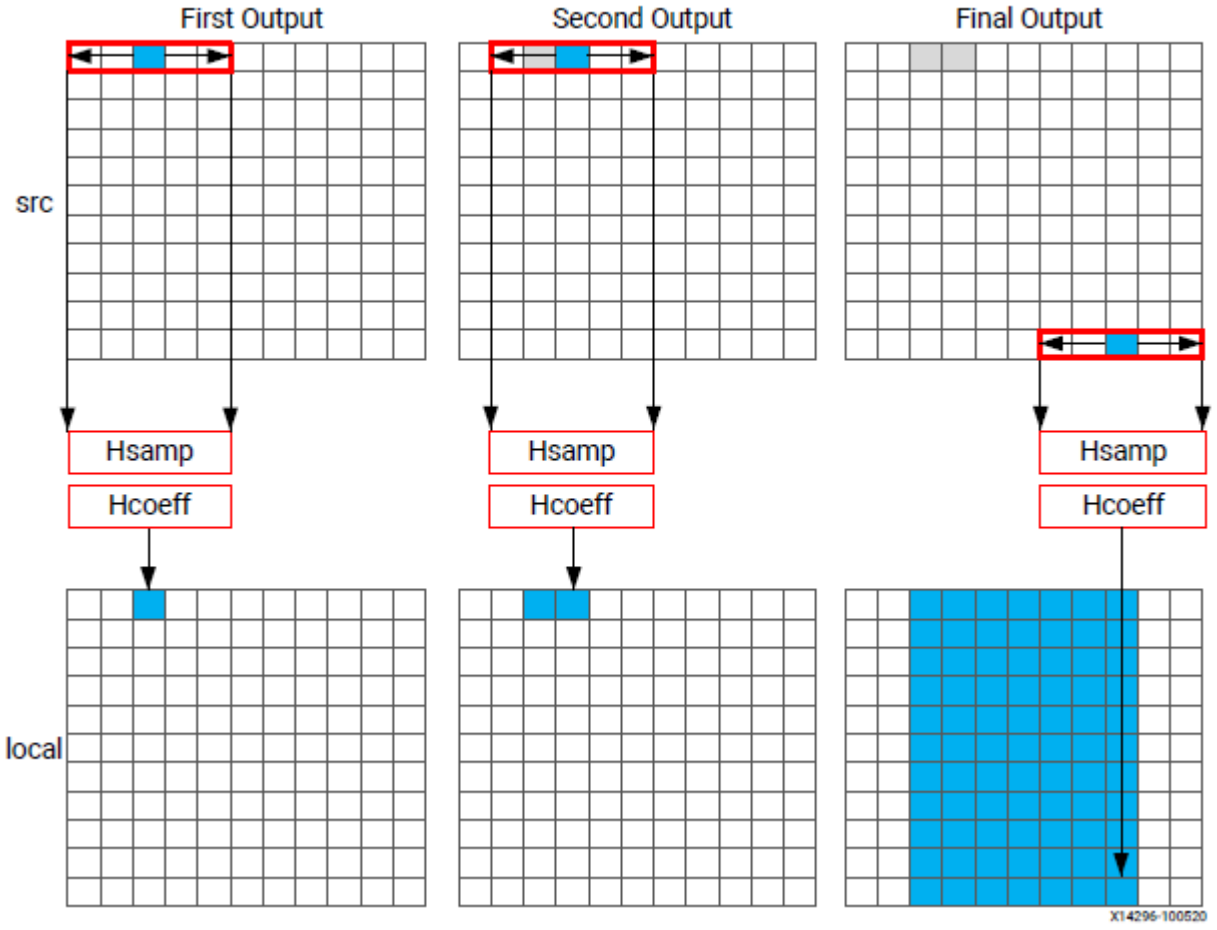
Naive Implementation – Horizontal Convolution

```
Clear_Local: for(int i = 0; i < height * width; i++)      local[i] = 0;

// Horizontal convolution
HConvH:for(int row = 0; rol < height; row++) {
  HconvW:for(int col = border_width; col < width - border_width; col++) {
    int pixel = row * width + col;
    Hconv:for(int i = -border_width; i <= border_width; i++) {
      local[pixel] += src[pixel + i] * hcoeff[i + border_width]; }   }   }
```

Issues:

- Need *local* array, size for 1920 * 1080 ~ 8MB (exceed on chip memory storage) – can we remove it?
- *Clear_Local* loop – needs 2M cycles
- Throughput is limited by
  - Input *src* is repeated read.
  - *src* is the argument of top-level function
- Hconv-loop entry/exit overhead
- The code prevents data streamed from PS using a DMA operation

Figure 54: **Horizontal Convolution**



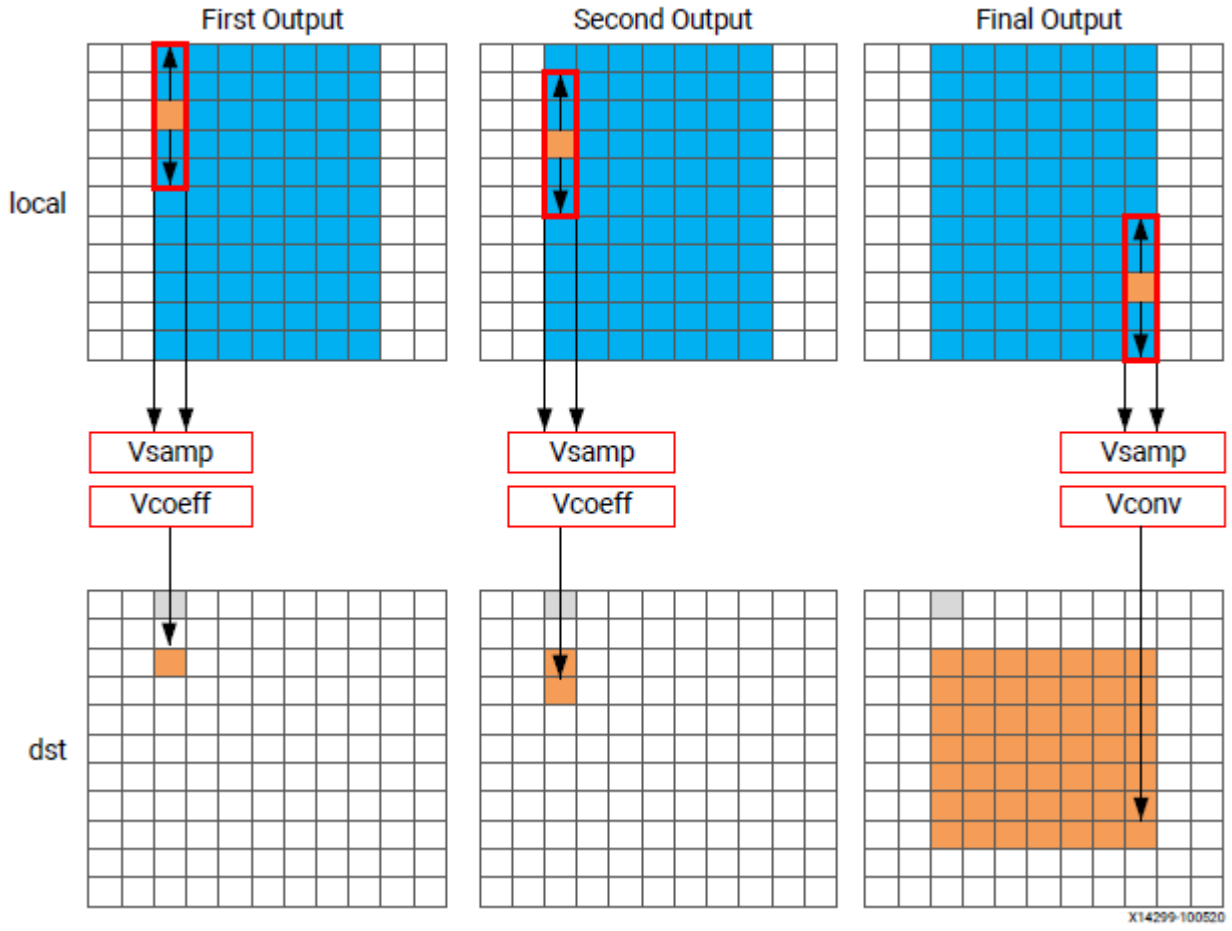
Naive Implementation – Vertical Convolution

```
Clear_Dst:for(int i = 0; i < height * width; i++)  dst[i] = 0;

// Vertical convolution
VconvW:for(int col = border_width; col < width - border_width; col++) {
  VconvH:for(int row = border_width; row < height - border_width; row++) {
    int pixel = row * width + col;
    Vconv:for(int i = -border_width; i <= border_width; i++) {
      int offset = i * width;
      dst[pixel] += local[pixel + i] * vcoeff[i+ border_width]; } } }
```

Issues:

- *Clear_Dst* loop – 2M cycles to clear output image *dst*
- Multiple read per pixel from array *local*
- Multiple write per pixel to output port *dst*
- Pixels are processed down the rows -> data could not stream out of array *local*
- If DATAFLOW optimization is used, this requires a ping-pong buffer. Double the memory requirement (4M pixels storage)

Figure 55: **Vertical Convolution**



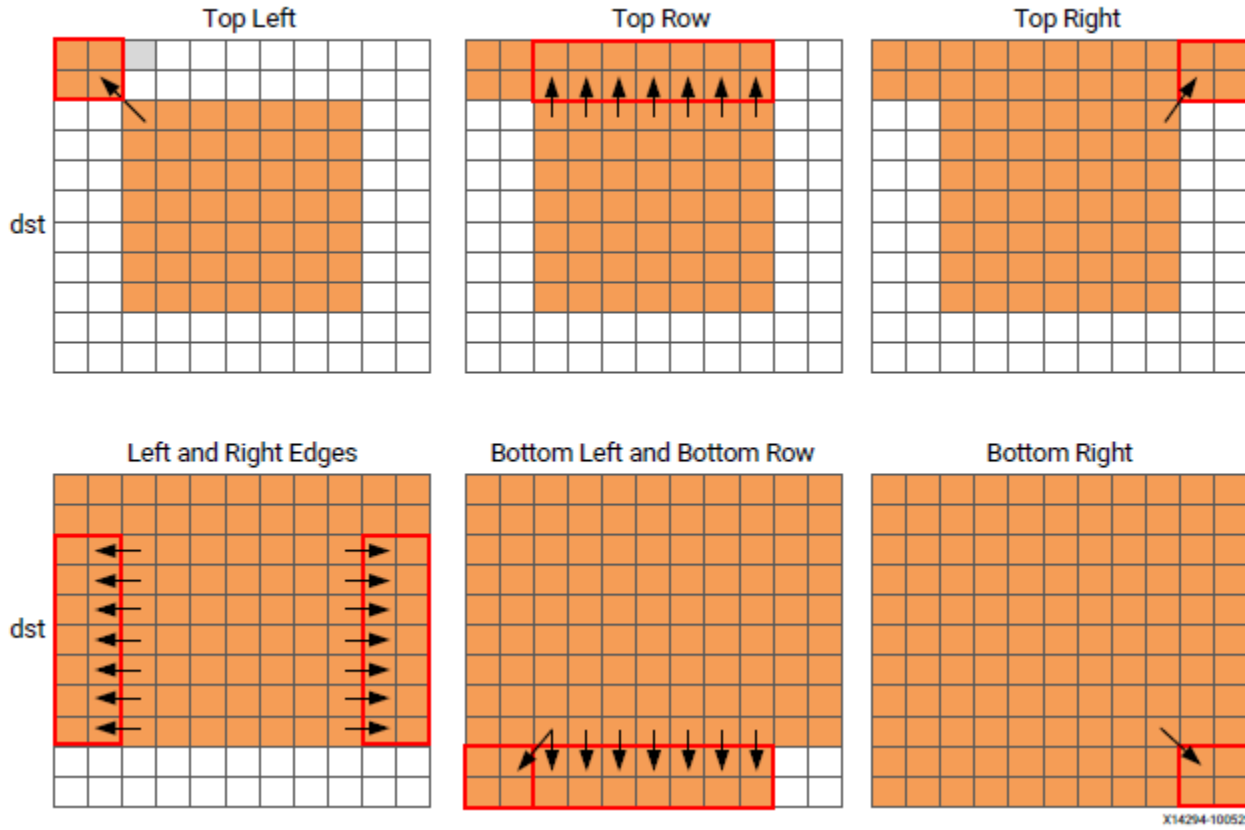
Naive Implementation - Borders

```
int border_width_offset = border_width * width;
int border_height_offset = (height - border_width -1) * width;
    // Border pixels
    Top_Border:for(int row = 0; row < border_width; row++) {
        for(int col = 0; col < border_width; col++) { … }   // upper-left corner
        for(int col = border_width; col < width - border_width; col++) { … } // top
        for(int col = width - border_width; col < width; col++) { … } // upper-right corner
    }
    Side_Border:for(int row = border_width; row < height - border_width; row++) {
       for(int col = 0; col < border_width; col++) { … }  // left side
       for(int col = width - border_width; col < width; col++) { … }  // right-side
    }
    Bottom_Border:for(int row = height - border_width; row < height; row++) [
       for(int col = 0; col < border_width; col++) { … }   // bottem-left corner
       for(int col = border_width; col < width - border_width; col++) { … }  // bottom
       for(int col = width - border_width; col < width; col++) { … } // bottom-right corner
```

Issues:

- Multiple write per pixel to output port *dst*
- *A for-loop for each condition*
- *Could not pipeline top-level loops – sub-loop has variable bound*
- Accessing data in an arbitrary or random access manner requires the data to be stored locally in arrays.

Figure 56: **Convolution Border Samples**



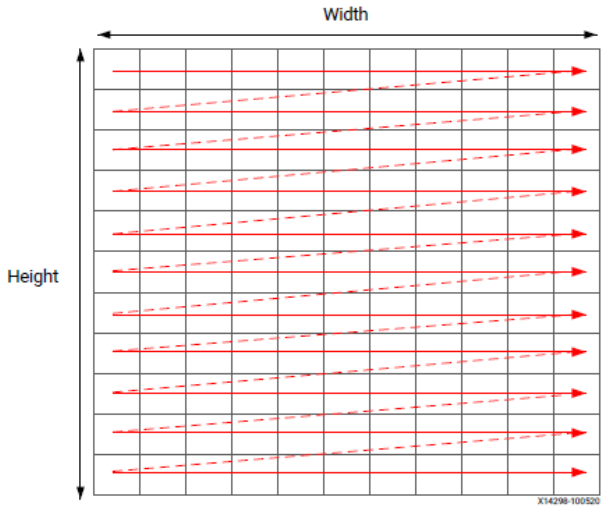
Ideally, we want

- No local buffer for image
- *src* image only read once
- *dst* image only write once
- Streaming datapath => pipelining frame
- Minimum latency and high throughput

Convolution by Stream

- Input/Output transferred in the Raster Scan Order
- FPGA data from/to CPU transferred in streaming manner
- Ensure continuous flow of data and data reuse
- Eliminate array local, instead, using stream FIFO (with depth 1)
- If the data from an hls::stream is required again, it must be cached (window-buffer, line-buffer)
- Use hls::stream to enforce good coding style
- Assertions are used to specify the loop bounds, allows HLS to report on the latencies of variable bounded loops

Figure 57: **Raster Scan Order**



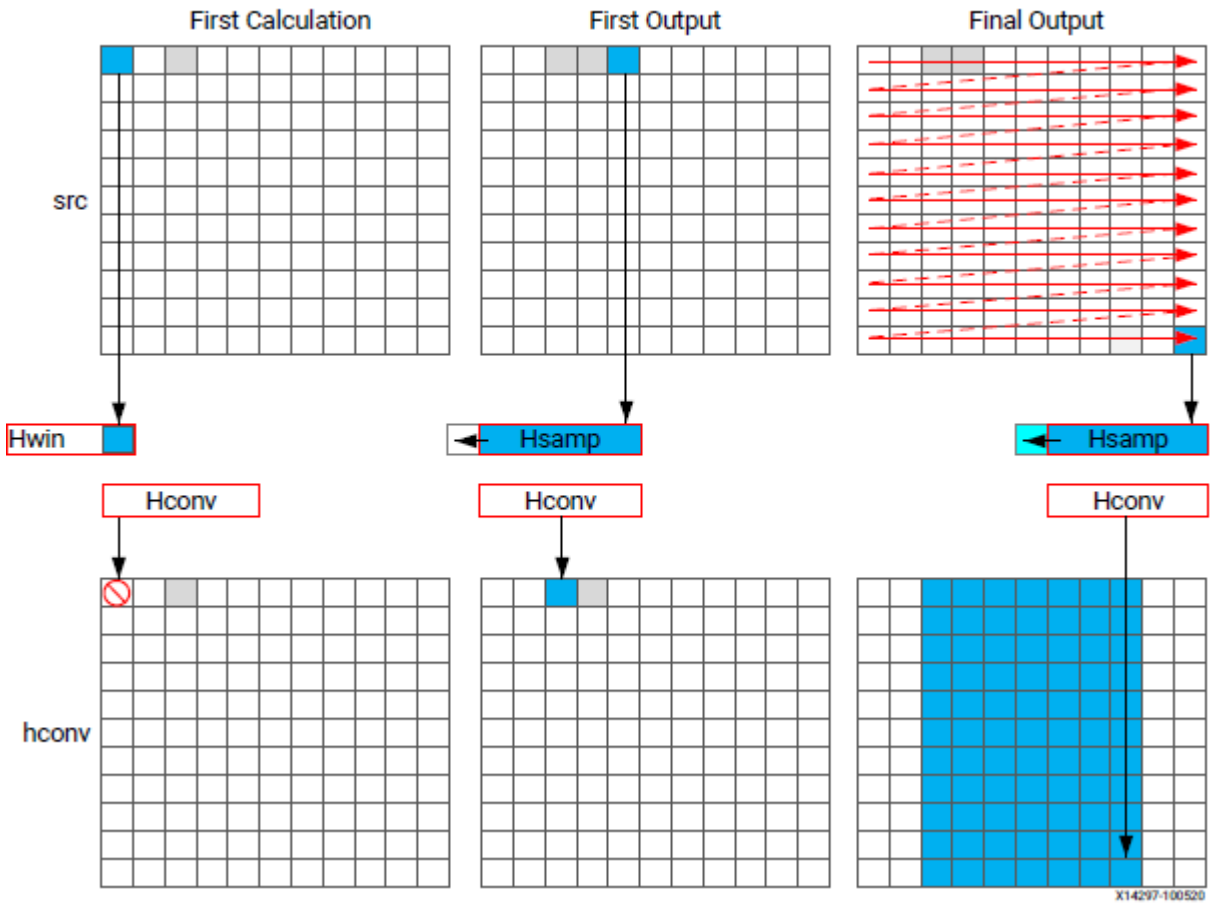

```
template<typenae T, int K>
static void convolution_orig(
            int width, int height,    // picture dimension
            hls::stream<T> &src,   //  source picture
            hls::stream<T> &dst,  // destination picture
            const T *hcoeff, *vcoeff );  // kernel  {
#pragma HLS DATAFLOW
#pragma HLS ARRAY_PARTITON variable=linebuf dim=1 complete
      const int conv_size = K;
      const int border_width = int(conv_size/2);
      const int vconv_xlim = width - (K-1));
      hls::stream<T> hconv("hconv");
      hls::stream<T> vconv("vconv");
// These assertion let HLS know the upper bounds of loops
      assert(height < MAX_IMG_ROWS);
      assert(width < MAX)IMG_COLS);
      assert(vconv_xlim < MAX_IMG_COLS - (K-1));
 // Horizontal convolution
      HConvH:for(int row = 0; rol < height; row++) {
         HconvW:for(int col = 0; col < width; col++) {
            T inv_val = src.read();
            Hconv:for(int i = 0; i < K; i++) { ..  }
            hconv << out_val;   } }
// Vertical convolution
      T line_buf[K-1][vconv_xlim];
      VconvW:for(int row = 0; row < height; rol++) {
         VconvH:for(int col = 0; col < vconv_xlim; col++) {
            T in_val = hconv.read();
            Vconv:for(int i = 0; i < K; i++) {  … }
            vconv << out_val;  }  }
T borderbuf[vconv_limit];
      // Border pixels
      Border:for(int row = 0; row < height; row++) {
         for(int col = 0; col < width; col++) {
            pix_in = vconv.read();
            dst << pix_out  }   }
```

Convolution by Stream – Horizontal Convolution

```
// Horizontal convolution
      HConvH:for(int row = 0; rol < height; row++) {
         HconvW:for(int col = 0; col < width; col++) {
#pragma HS PIPELINE
            T inv_val = src.read();
            T out_val = 0;
            Hconv:for(int i = 0; i < K; i++) {
               hwin[i] = i < K - 1 ? hwin[i+1]: in_val;  // shift
               out_val += hwin[i] * hcoeff[i];
            }
            if( col >= border_width && col < width - border_width) {
               hconv << out_val;
            }
         }
      }
```

- *hwin* is used to cache repeated used data (src data)
- Variable *out_val* used to calculate convolution, and initialized in the loop
- Conditional operation is used in the loop (no overhead as oppose to CPU architecture)

Figure 58: **Streaming Horizontal Convolution**

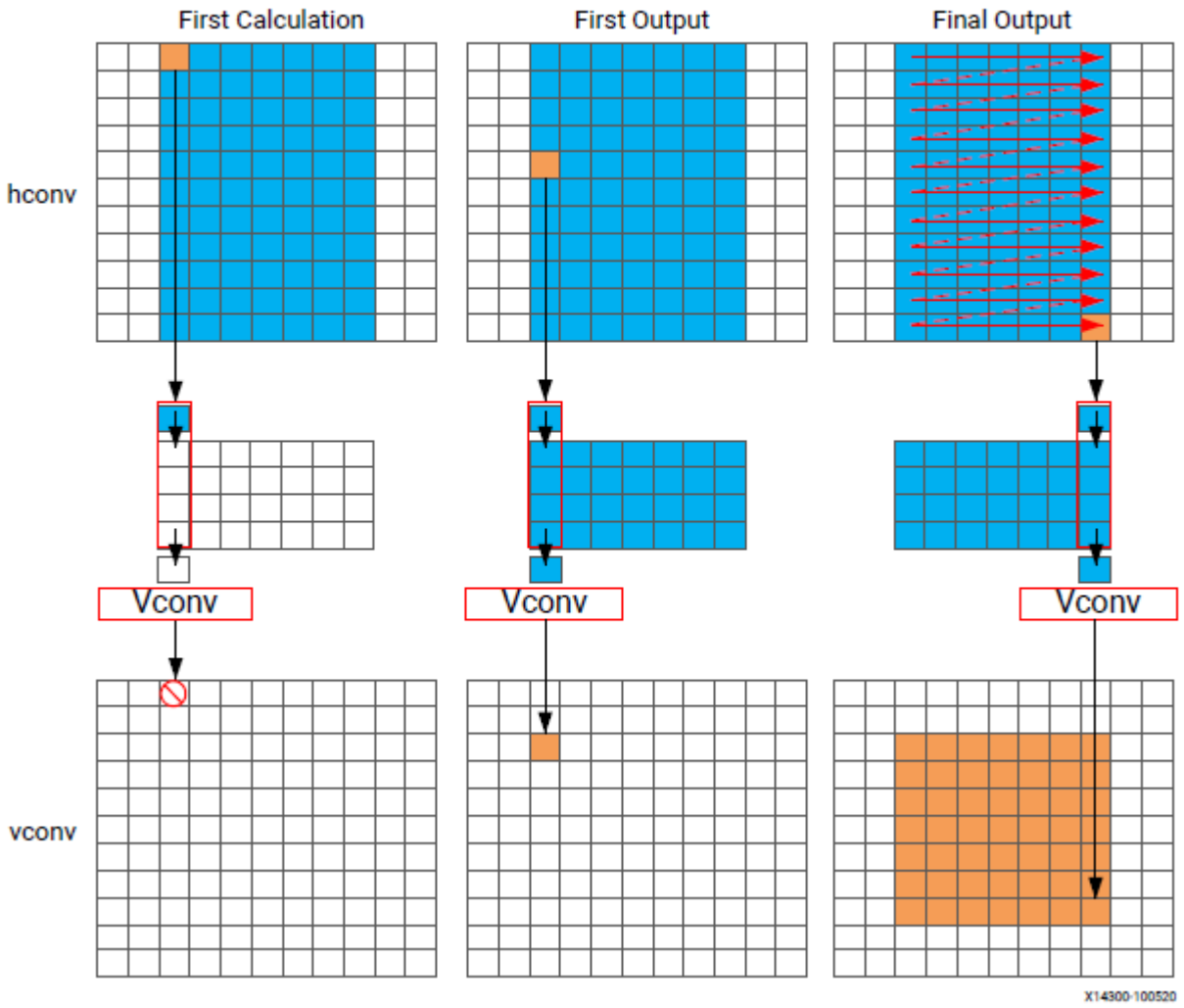


Convolution by Stream – Vertical Convolution

```
// Vertical convolution
    T line_buf[K-1][vconv_xlim];
    VconvW:for(int row = 0; row < height; rol++) {
      VconvH:for(int col = 0; col < vconv_xlim; col++) {
        #pragma HLS DEPENDENCE variable=linebuf inter false
        #pragma HLS PIPELINE
        T in_val = hconv.read();
        T out_val = 0;
        Vconv:for(int i = 0; i < K; i++) {
          T vwin_val = i < K -1 ? linebuf[i][col] : in_val;
          out_val += vwin_val * vcoeff[i];
          if(i > 0)
            linebuf[i-1][col] = vwin_val;
        }
        if( row >= K - 1)
          vconv << out_val;
        } } }
```

- ***line_buf (K-1 lines)*** to allow column access
- The first output on the Kth line is read
- Calculation before the Kth line is discarded
- The hconv.read is in the same sequence as hconv.write
  - if(col >= border_width && col < width - border_width) hconv << out_val;
  - VconvH:for(int col = 0; col < vconv_xlim; col++) { T in_val = hconv.read();

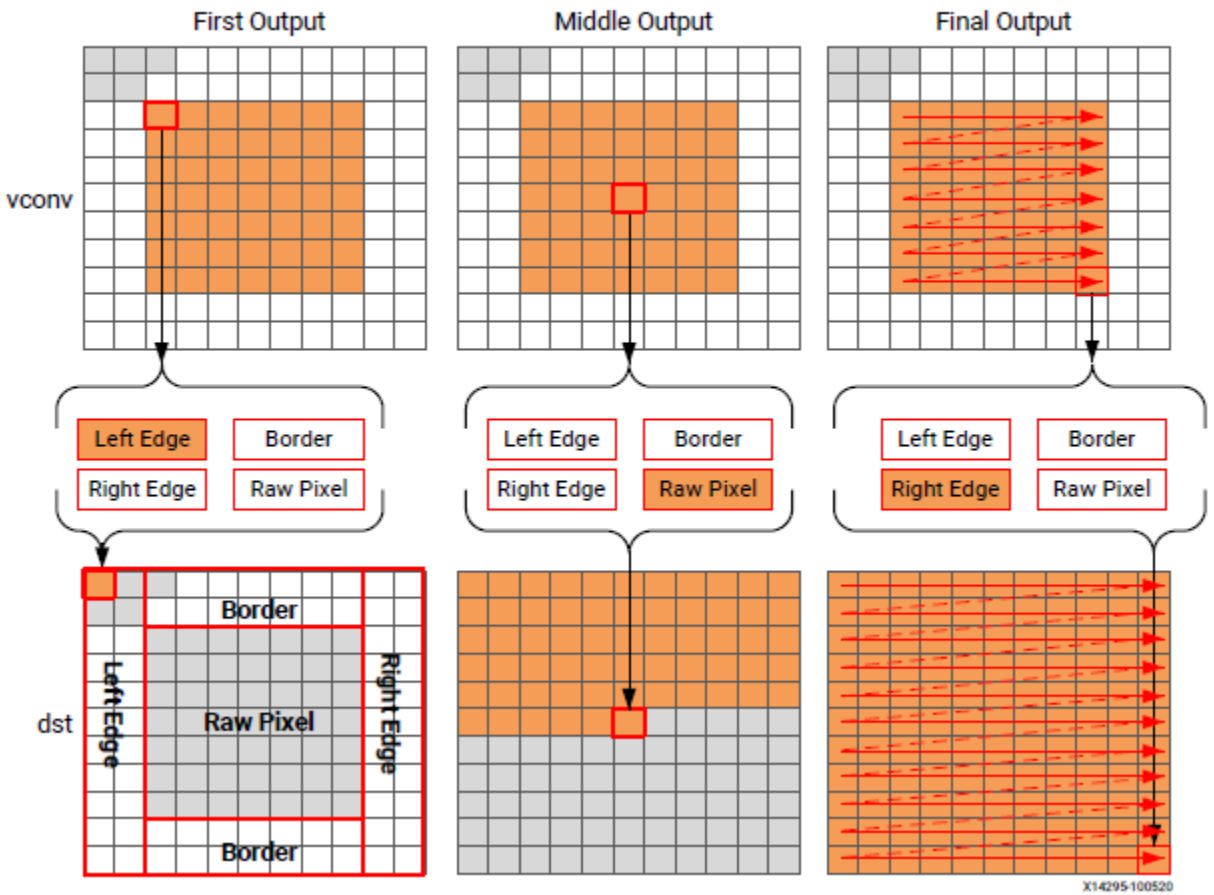
Figure 59: **Streaming Vertical Convolution**



Convolution by Stream – Border

```
T borderbuf[vconv_limit];
    // Border pixels
    Border:for(int row = 0; row < height; row++) {
      for(int col = 0; col < width; col++) {
#pragma HLS PIPELINE
        T pix_in, l_edge_pix, r_edge_pix, pix_out;  j=0;
        #pragma HLS PIPELINE
        if(row == 0 || (row > border_width && row < height - border_width)) {
          if( col < width - (K - 1)) {
            pix_in = vconv.read();
            borderbuf[j++] = pix_in;
          }
          if( col == 0) {
            l_edge_pix = pix_in;
          }
          if(col == width - K) {
            r_edge_pix = pix_in;
          }
        }
        if( col <= border_width) {
          pix_out = l_edge_pix;
        } else if( col >= width - border_width -1) {
          pix_out = r_edge_pix;
        } else {
          pix_out = borderbuf[col - border_width];
        }
        dst << pix_out
      } // loop_col
    } // loop_row
```

Figure 60: **Streaming Border Samples**



- 4 type of caches: left_edge, right_edge, borderbuf, pix_in
- Extensive use of conditional operations in the loop, does not impact pipeline

Summary

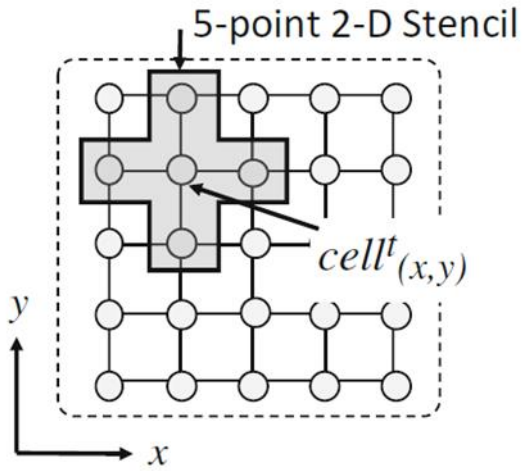
- No local buffer for image
- *src* image only read once
- *dst* iage only write once
- Streaming datapath => pipelining frame
- Minimum latency and high throughput

- Minimize data input reads
- Minimize accesses to arrays
- Use small localized cache (hwin, line-buffer) to hold result
- Perform conditional branching inside pipelined tasks rather than conditionally execute tasks
- Minimize output writes
- Using hls::stream to enforce good coding practices

Optimal Architecture for Stencil Application

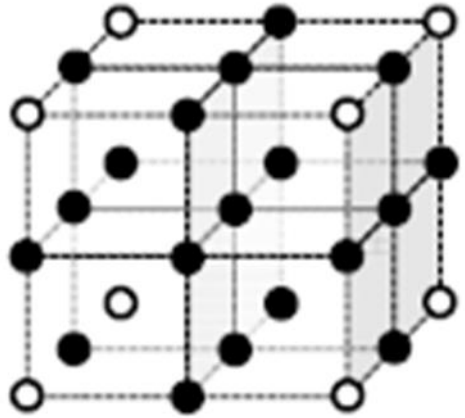
https://cadlab.cs.ucla.edu/beta/cadlab/sites/default/files/publications/42_3.pdf
https://github.com/miltosmac/TCAD

Stencil Computation Application



Important class of kernels in

1. Scientific computation, e.g. fluid dynamics, simulation, electromagnetic simulation
2. image processing,
3. constituent kernels in multigrid methods, and
4. partial differential equation solvers.
5. ……



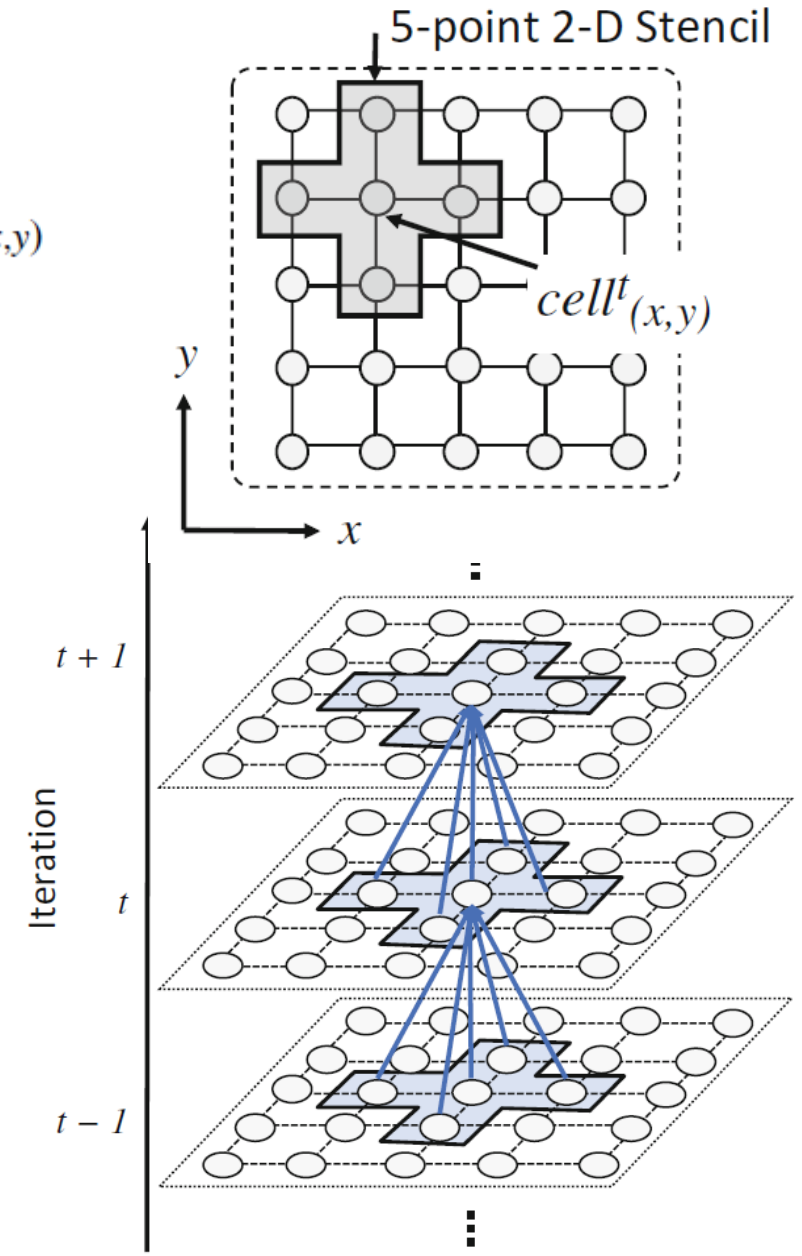
19-point stencil
'SEGMENTATION_3D'

Stencil Computation

$$cell^{t+1}_{(x,y)} = k_1 \times cell^{t}_{(x,y-1)} + k_2 \times cell^{t}_{(x-1,y)} + k_3 \times cell^{t}_{(x,y)}$$
$$+k_4 \times cell^{t}_{(x+1,y)} + k_5 \times cell^{t}_{(x,y+1)}.$$

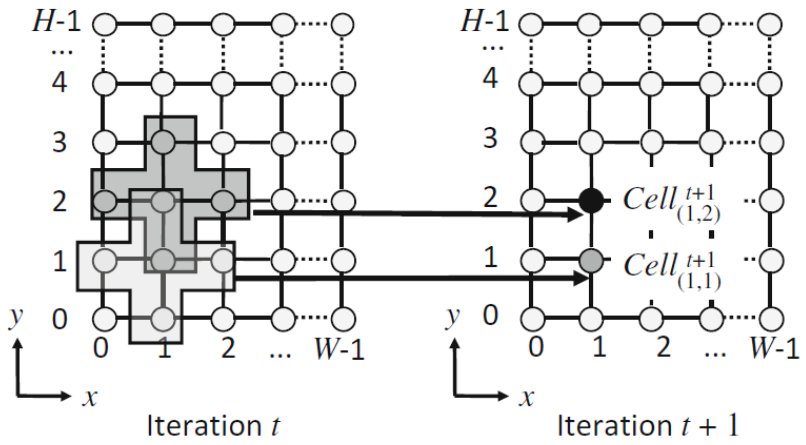
```
void CPU_iteration(float din, float dout) {
  for(int y=1; y<H-1; y++) {
    for(int x=1; x<W-1; x++) {
      // Compute one stencil according to Eq.(6.1)
      dout[y*W + x] = k1* din[(y-1)*W + x] +
                      k2*din[(y)*W + x-1] +
                      k3*din[(y)*W + x] +
                      k4*din[(y)*W + x+1] +
                      k5*din[(y+1)*W + x];
} } }
```

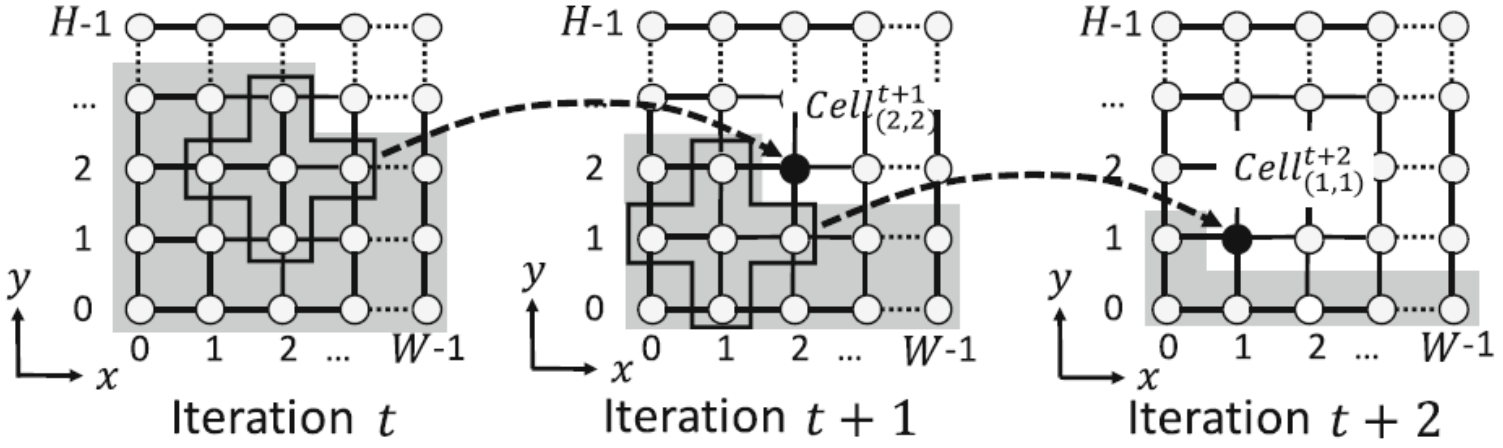
```
int main() {
  for(int i=0; i<ITERATION; x++) {
      if((i%2)==0) CPU_stencil(buf0, buf1);
      else CPU_stencil(buf1, buf0);
} }
```



Design of FPGA-Based Computing Systems with OpenCL Chap 6.1. https://link.springer.com/book/10.1007/978-3-319-68161-0

Parallelism





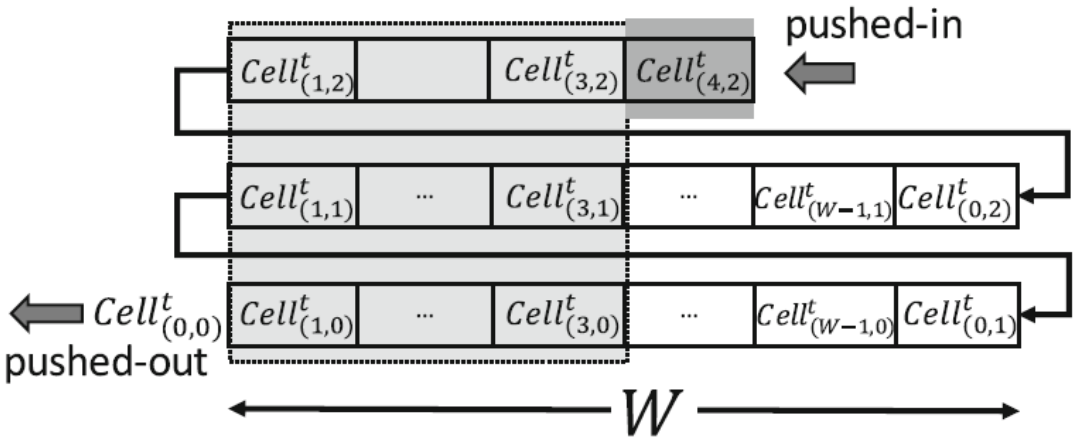
Stencil-parallel computation

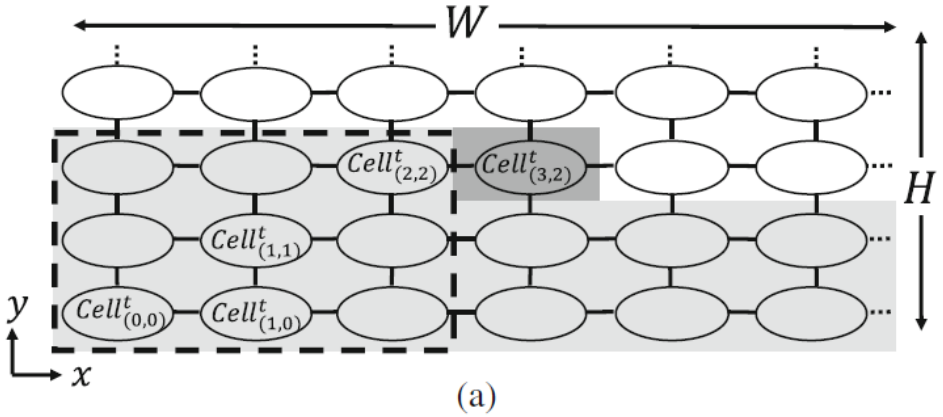
- Require memory-bandwidth

Iteration-parallel computation

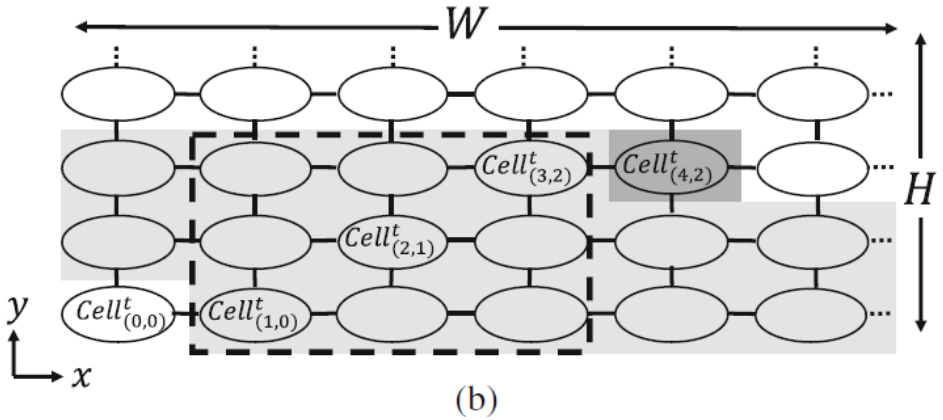
Data Access Sequence and Lifetime

- Raster Scan order
- To compute cell(2,1) need to load cell(3,2), then cell(0,0) is no longer required
- Lifetime of data kept in buffer = 2W + 4 (distance of cell(3,2), and cell(0,0)



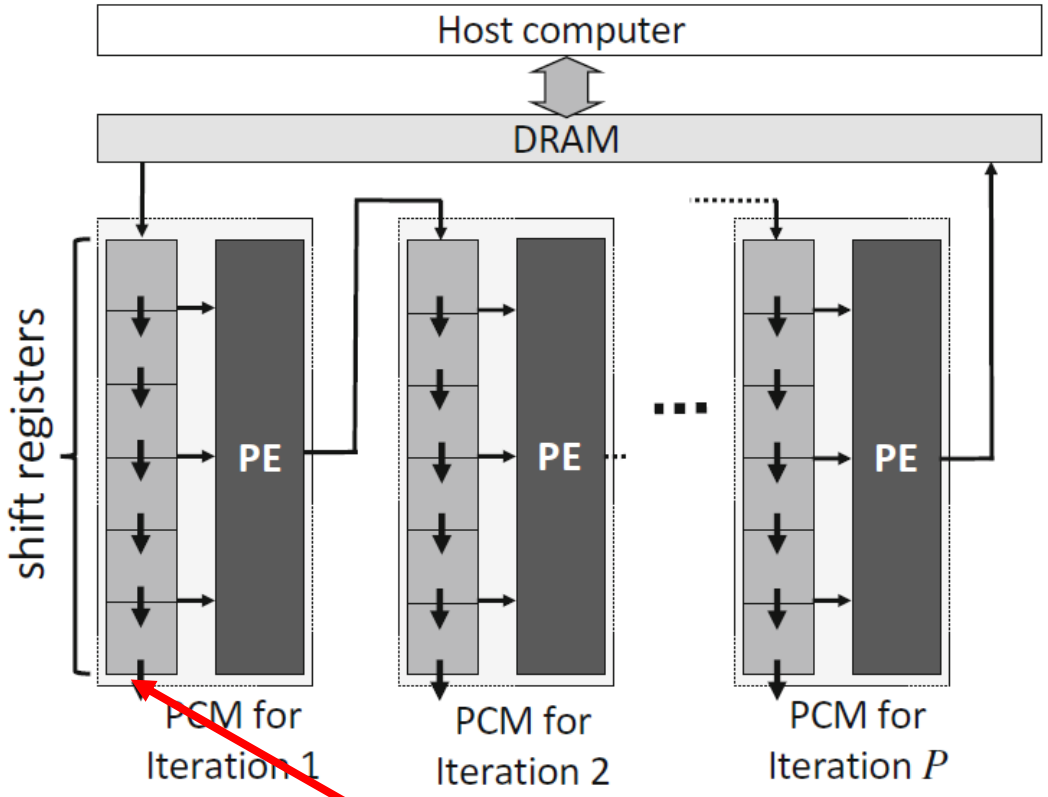


(a)



(b)

Iteration Parallelism



Length of shift register = lifetime = 2W + 4
Use too much shift register, optimize the resource usage

Code Listing & Explained

```
void stencil( const float * din, float *dout ) {
  float shiftreg[P][2*W+4];
  const int loop_iteration = P*(W+2) + W*H;
  int count = 0;
  while (count != loop_iteration) {
    for (int i = 2*W+3; i>0; --i)  {
      #pragma unroll
      for (int j=0; j<P; j++ ) {
          #pragma unroll
          shiftreg[j][i] = shiftreg[j][i-1];
    } ]
shiftreg[0][0] = (count < W*H) ? din[count] : 0.0f;
```

Shifting

```
float result;
float ce, le, ri, to, bo;
for (int j=0; j<P-1; j++)  {
        #pragma unroll
        to = shiftreg[j][2*W+2];
        le = shiftreg[j][1*W+3];
        ce = shiftreg[j][1*W+2];
        ri = shiftreg[j][1*W+1];
        bo = shiftreg[j][0*W+2];
  if( count >= ((j+1)*(W+2)+W) && (((count-
(j+1)*(W+2)) % W) != 0) &&
((count-(j+1)*(W+2)) % W) != (W-1) && count <
((j+1)*(W+2)+W*(H-1)) )
  result = k1*to + k2*le + k3*ce + k4*ri + k5*bo;
else
   result = 0.0f;

shiftreg[j+1][0] = result;
}
```

Stencil Calculation

```
int j = P-1;
        to = shiftreg[j][2*W+2];
        le = shiftreg[j][1*W+3];
        ce = shiftreg[j][1*W+2];
        ri = shiftreg[j][1*W+1];
        bo = shiftreg[j][0*W+2];

if(count >= (j+1)*(W+2)) {
  if( count >= ((j+1)*(W+2)+W) &&  (((count-
(j+1)*(W+2)) % W) != 0) &&
((count-(j+1)*(W+2)) % W) != (W-1) && count <
((j+1)*(W+2)+W*(H-1)) )
  result = k1*to + k2* le + k3*ce + k4*ri +
k5*bo;
else
  result = 0.0f;

dout[count-(j+1)*(W+2)] = result;
 }
count++;
} }
```

Last iteration output

Non-Uniform Partitioning of Data Reuse Buffers

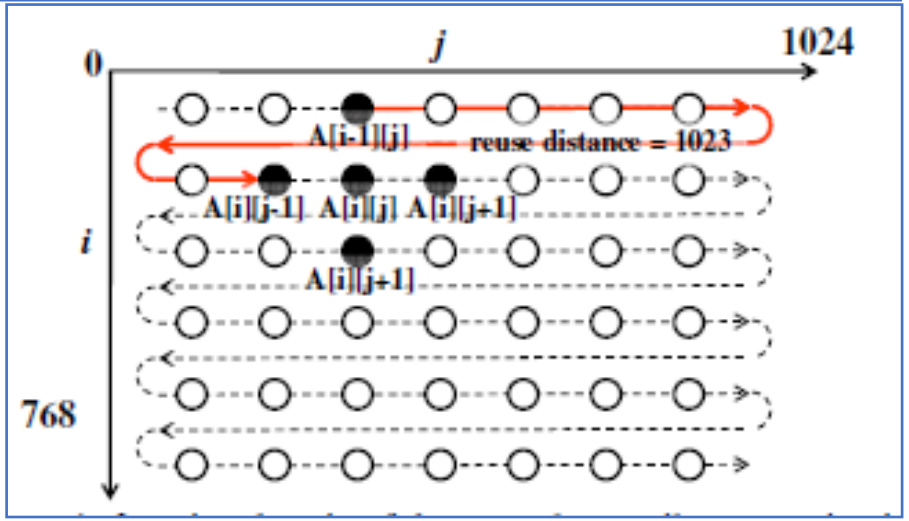
- Each data accessed for multiple time as time goes

➔ Data reuse via on-chip memory

- Multiple data access at each time point

➔ Partition of on-chip memory

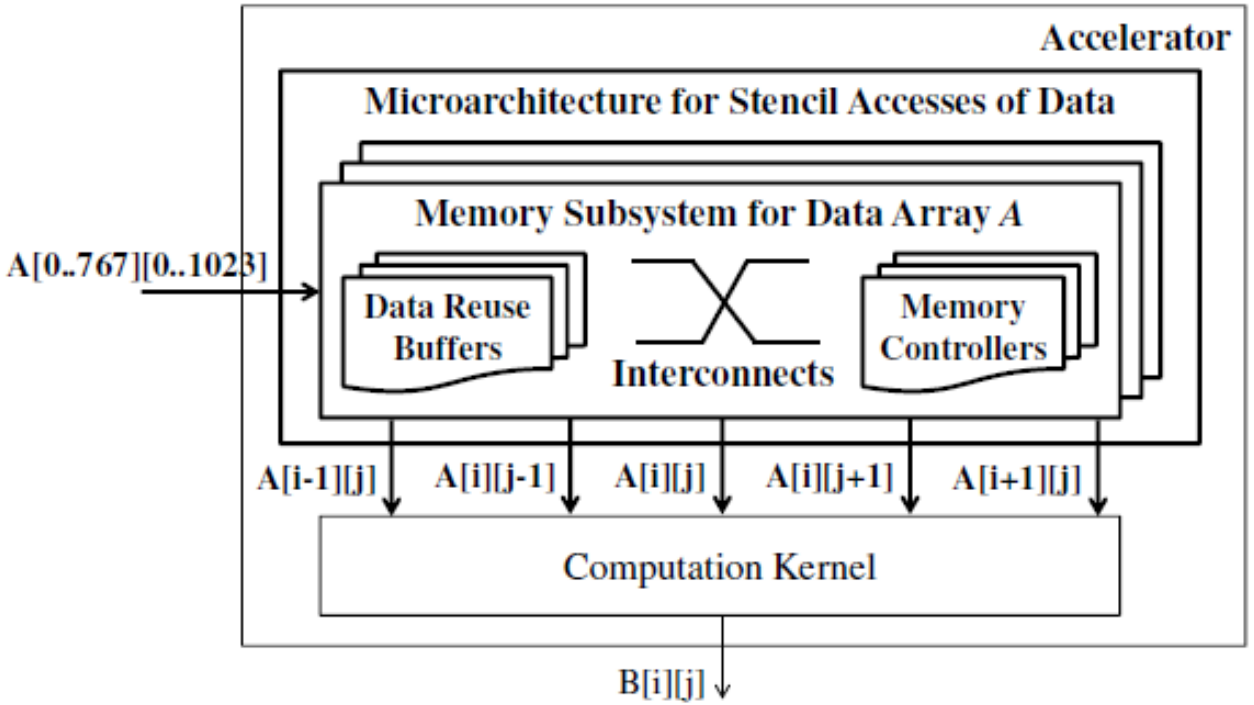
```
void denoise2D( float A[768][1024],
                float B[768][1024] ) {
  for(int i = 1; i < 767; i++ )
    for(int j = 1; j < 1023; j++)
      B[i][j] = pow(A[i][j] - A[i][j-1], 2) +
                pow(A[i][j] - A[i][j+1], 2) +
                pow(A[i][j] - A[i-1][j], 2) +
                pow(A[i][j] - A[i+1][j], 2);
}
```




```
to = shiftreg[j][2*W+2];
le = shiftreg[j][1*W+3];
ce = shiftreg[j][1*W+2];
ri = shiftreg[j][1*W+1];
bo = shiftreg[j][0*W+2];
```

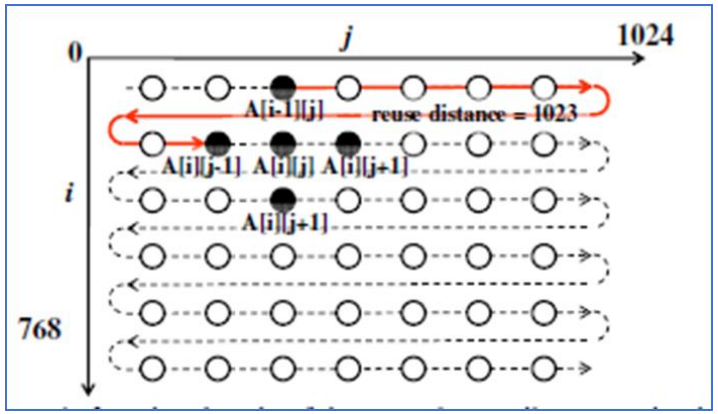
https://cadlab.cs.ucla.edu/beta/cadlab/sites/default/files/publications/42_3.pdf

Microarchitecture for Stencil Applications

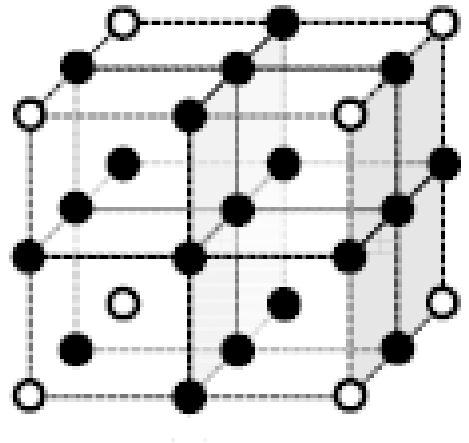



```
void denoise2D_kernel(
            volatile float  * a0_ptr,
            volatile float * a1_ptr,
            volatile float * a2_ptr,
            volatile float * a3_ptr,
            volatile float * a4_ptr,
            volatile float * b_ptr )  {
  for( int i =  1; i < 767; i++ )
    for( int j = 1; j < 1023; j++ ) {
        #pragma AP pipeline II=1
        a0 = *a0_ptr;
        *b_ptr = pow(a0 - *a1_ptr, 2) +
                 pow(a0 - *a2_ptr, 2) +
                 pow(a0 - *a3_ptr, 2) +
                 pow(a0 - *a4_ptr, 2);
} } }
```

Design Objectives



- Full pipelining
  - Provide all the data in the stencil window to kernel
- Minimum data reuse buffer size
  - A data element is fully reused. Stay in on-chip memory from its first access until its last access
  - Element A[i][j] is accessed by A[i+1][j] (first time), and A[i-1[[j] for the last time. Data reuse buffer A size is 2048.
- Minimum number of reuse buffer banks
  - All data elements in stencil window should be in different bank
  - n=5 need at least 4 memory banks (assuming one element is from external memory)



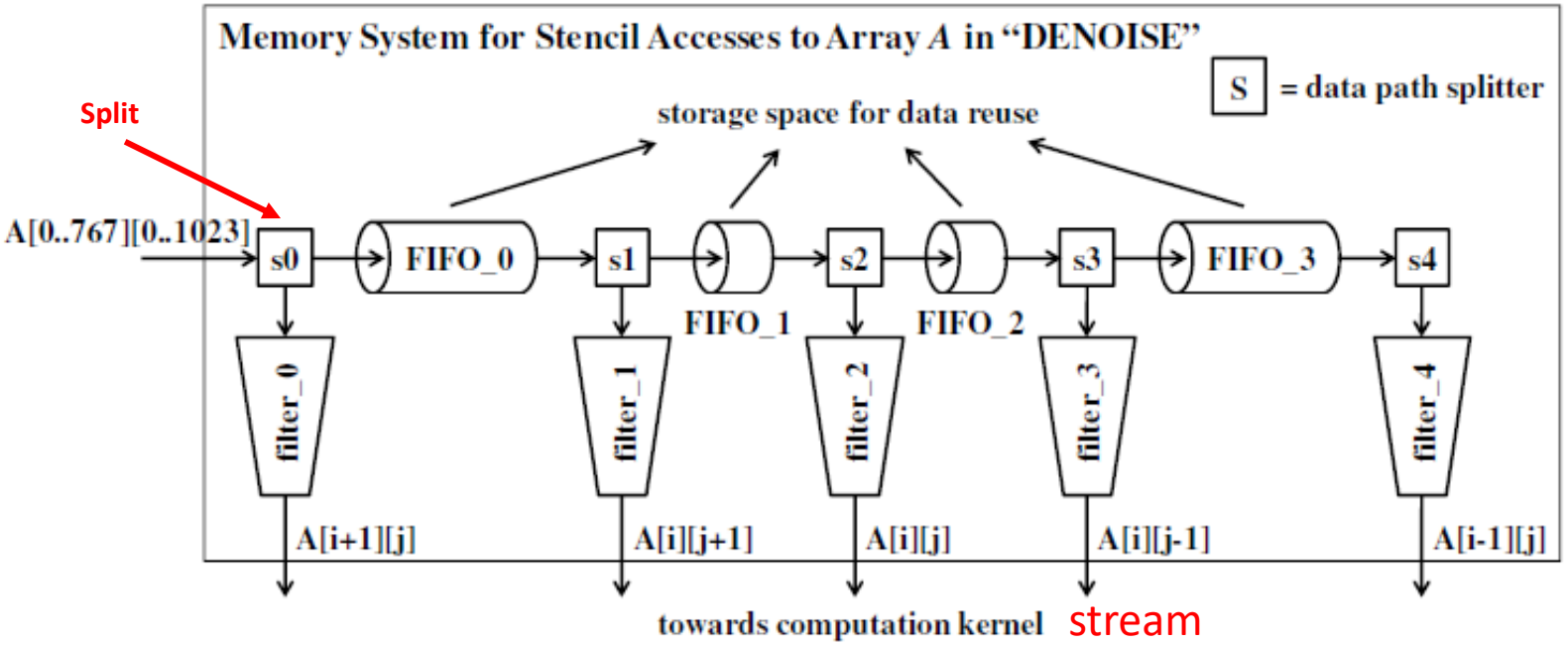
19-point stencil
'SEGMENTATION_3D'

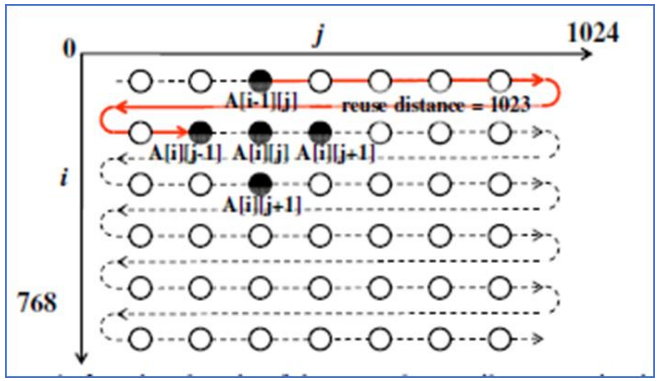
Microarchitecture of Memory Subsystem

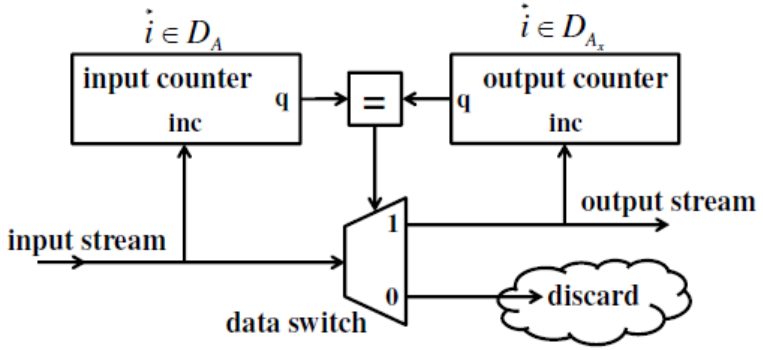
Back

- Non-uniform reuse buffer – customized to the shape of the stencil window
- Each data reuse buffer has different size
- Distributed controller (filter) based on streaming
- Optimal Design
  - Minimum Reuse Buffer size - 2048
  - Minimum Number of Buffer Banks - 4 buffer banks

FIFO ID	precedent/successive references	FIFO size	physical impl.
FIFO 0	$A[i+1][j] \rightarrow A[i][j+1]$	1023	BRAM
FIFO 1	$A[i][j+1] \rightarrow A[i][j]$	1	register
FIFO 2	$A[i][j] \rightarrow A[i][j-1]$	1	register
FIFO 3	$A[i][j-1] \rightarrow A[i-1][j]$	1023	BRAM







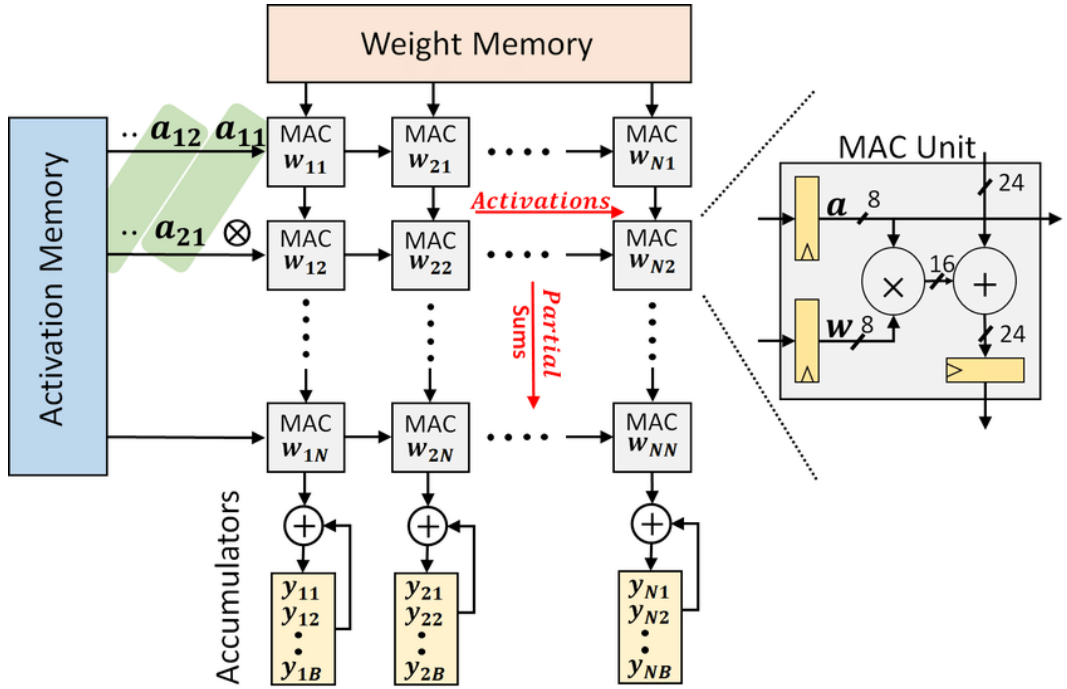
Data Filter

Smith Waterman - Systolic Array

https://past.date-conference.com/proceedings-archive/2017/pdf/0231.pdf
https://github.com/necst/coursera-sdaccel-practice

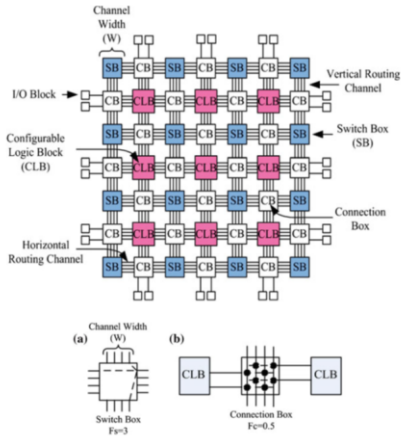
Why Systolic Array Architecture

- Characteristic
  - Homogenous parallel / pipeline computing
  - Synchronous evaluation
  - Spatial / temporal locality
  - Scalability
- Advantage
  - High degree of parallelism, compact, robust and efficient
  - Streaming dataflow
  - Good for FPGA architecture – local connectivity



Island-Style Routing

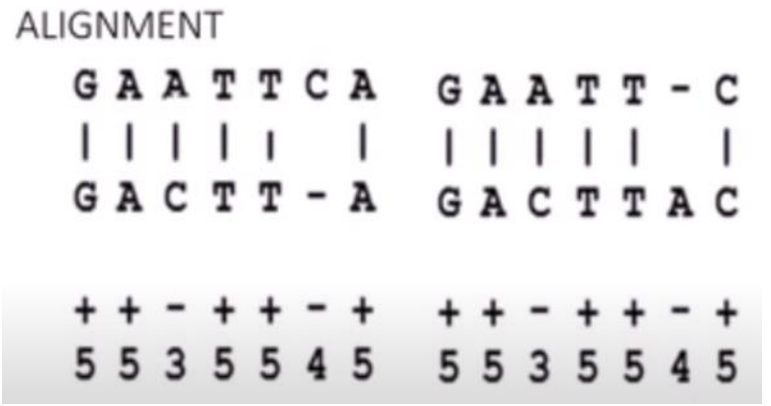
- Routing network occuping 80-90% area
- 98% memory bits are for routing configure
- CLB growth is O(N^2), Interconnect is O(N)
- CLB go unused for insufficient interconnect
- Cascade Memory/DSP uses hard wiring instead of interconnect fabric
- 2.5D SSI with > 10K connections between slices



Fs - connections offered per incoming wire. This is also called S block flexibility.
Fc - number of channel wires connected to each logic pin. This is also called C block flexibility

Computational Biology – Smith-Waterman

- Local sequence alignment between two nucleotides or proteins
- Dynamic programming algorithm
- Highly Compute Intensive
- Applications
  - Similar document retrieval
  - Near duplicate web page detection
  - Fingerprint
  - Deep packet inspection.



Smith-Waterman Examples

query [N]

Database [M]

Similarity matrix – S[i,j]

S[i,j] = max { S[i-1,j-1] + score (i, j)

S[i-1, j] – R_del

S[i, j-1] – R_ins

0

}

Traceback Matrix (N, NW, W)

From the highest score traceback to until 0

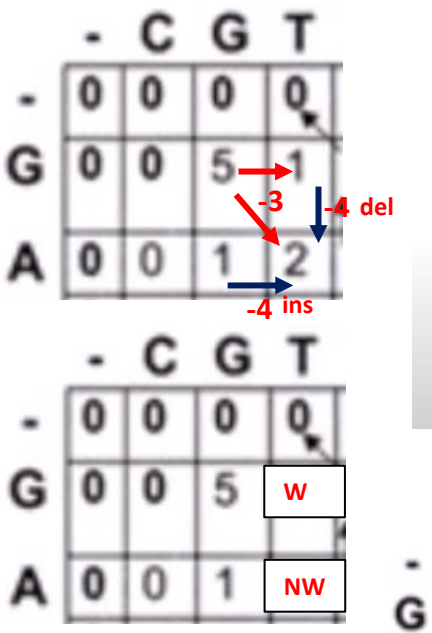
score-matrix

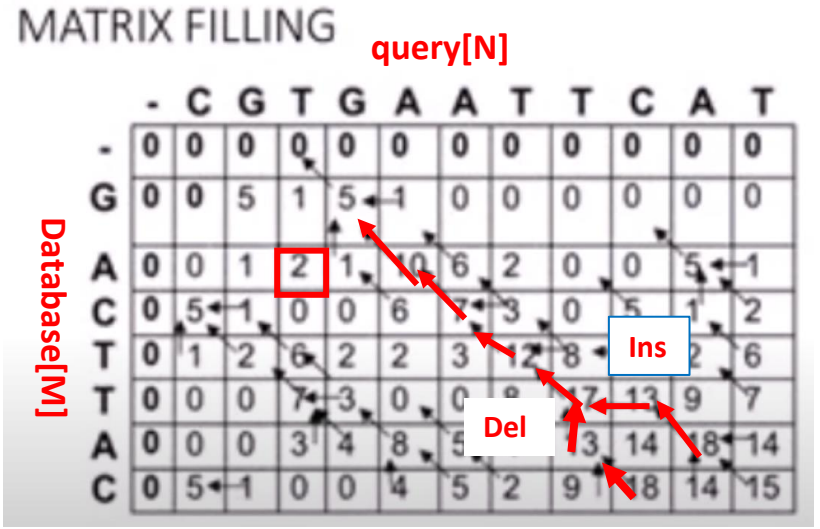
Match = +5

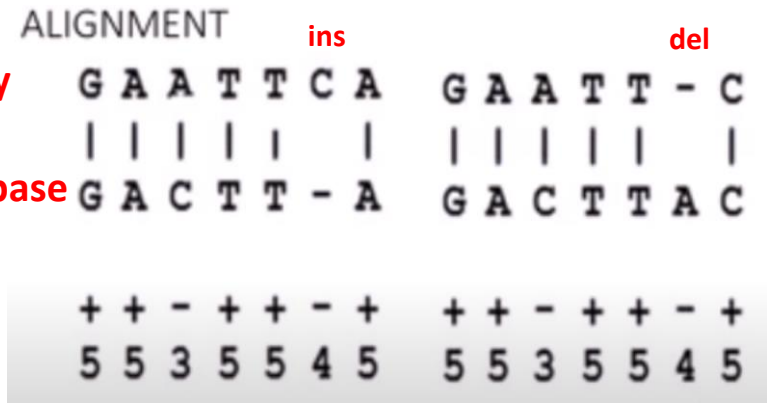
Mismatch = -3

Delete = -4

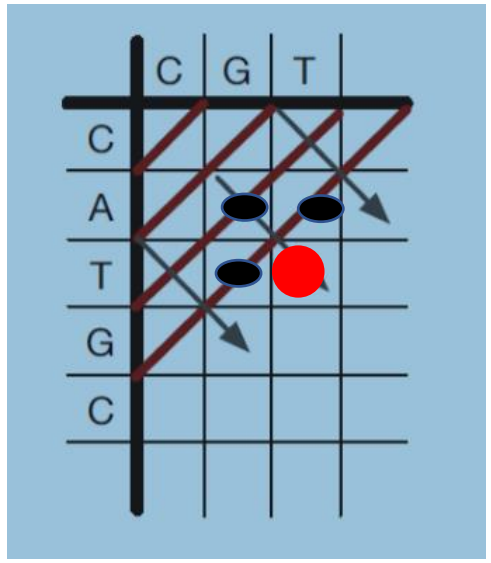
Insert = -4



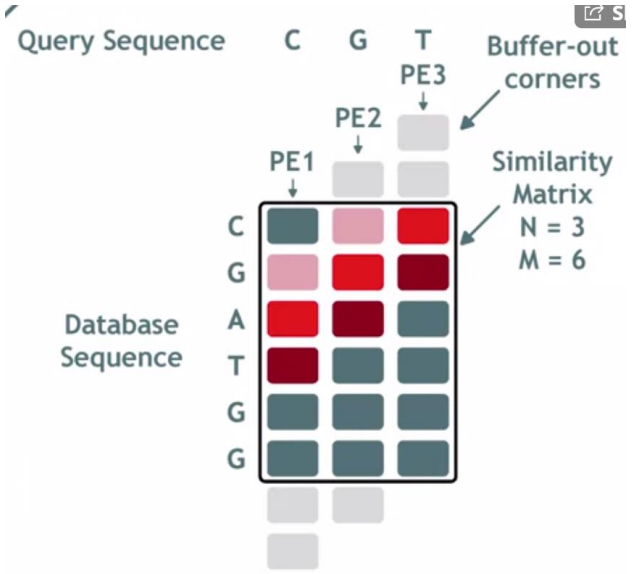




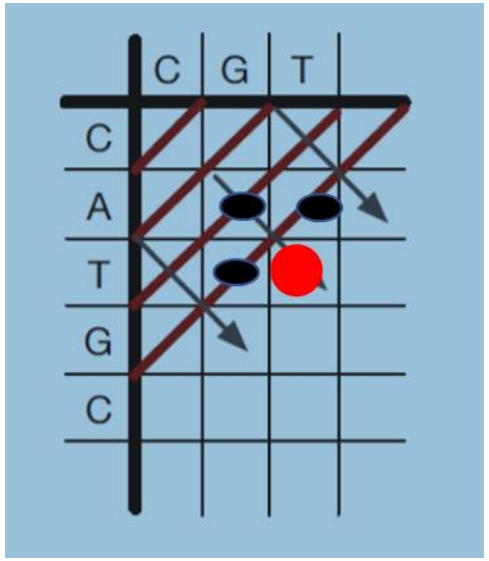
Systolic Array



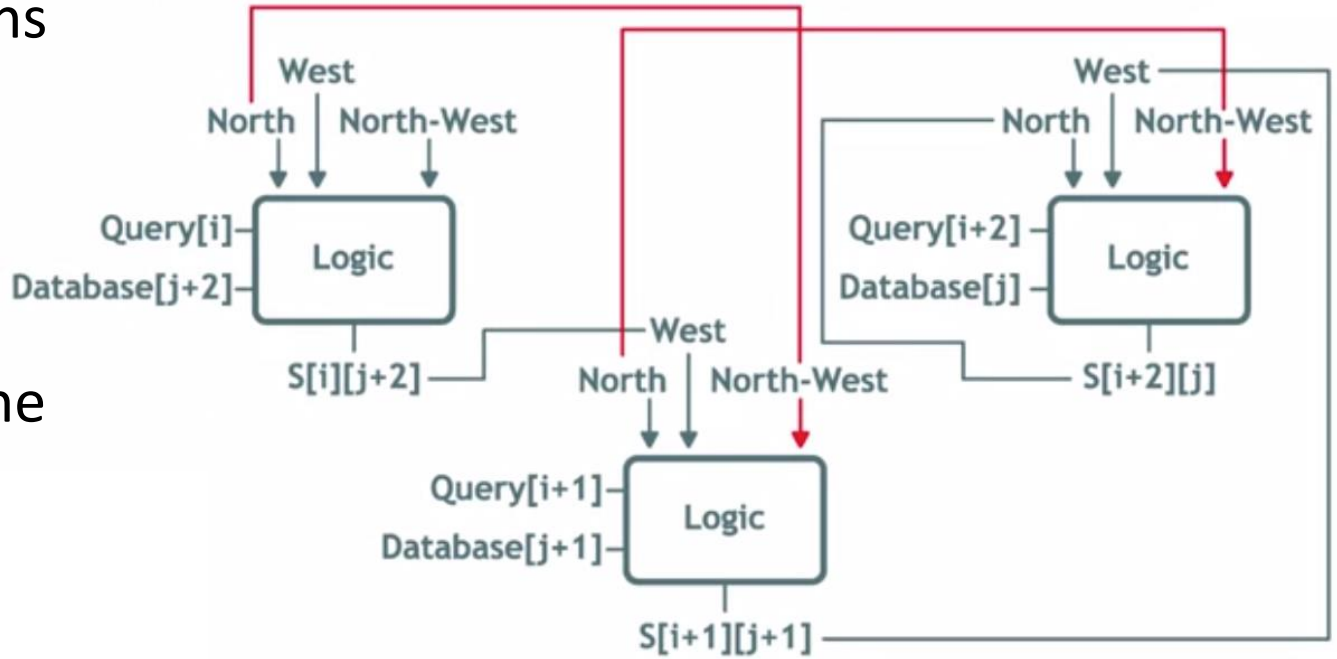
- Anti-Diagonal is dependency free
- Use values from two previous anti-diagonal iterations
- Use N PE (query[N]) – compute an anti-diagonal each cycle. It takes (M+N-1) cycles
- Each PE performs same function



Systolic Array – PE operation



- Each PE receive data from neighbor PE
  - Query
  - Database
  - Dependency from previous iterations
    - West/North/North-West value
- Output to next iteration
  - West of successive PE
  - North of the same PE
  - Its North becomes North-West of the successive PE



Code Structure

```
void compute_matrices( char *string1_g, char *string2_g, ap_uint<512>
*direction_matrix_g)
{
char string1[N];
#pragma HLS ARRAY_PARTITION variable=string1 complete dim=1
char string2[DATABASE_SIZE];
#pragma HLS ARRAY_PARTITION variable=string2 complete dim=1
memcpy(string1, string1_g, N*sizeof(char));
memcpy(string2, string2_g, DATABASE_SIZE * sizeof(char));

int north[N+1];
#pragma HLS ARRAY_PARTITION variable=north complete dim=1
int west[N+1];
#pragma HLS ARRAY_PARTITION variable=west complete dim=1
int northwest[N+1];
#pragma HLS ARRAY_PARTITION variable=northwest complete dim=1

num_diag_for: for(int num_diagonals = 0; num_diagonals < N + M - 1;
num_diagonals++){
#pragma HLS inline region recursive
#pragma HLS PIPELINE

calculate_diagonal( ..... );
}
```

Why inline & pipeline ?

```
void calculate_diagonal(int num_diagonals,
        char string1[N],   // query string
        char string2[DATABASE_SIZE],  // database
        int northwest[N + 1],
        int north[N + 1],
        int west[N + 1],    ...  ) {
calculate_diagonal_for: for(int index = N - 1; index >= 0; index --){
        int val = 0;
        unsigned int q = string1[index];
        unsigned int db = string2[databaseLocalIndex];

        const short match = (q == db) ? MATCH : MISS_MATCH;
        const short val1 = northwest[index] + match;
        const short val2 = north[index] + GAP_d;
        const short val3 = west[index] + GAP_i;

        // calculate val = max of (val1, val2, val3)
        northwest[index + 1] = north[index];
        north[index] = val;
        west[index + 1] = val;
} }
```

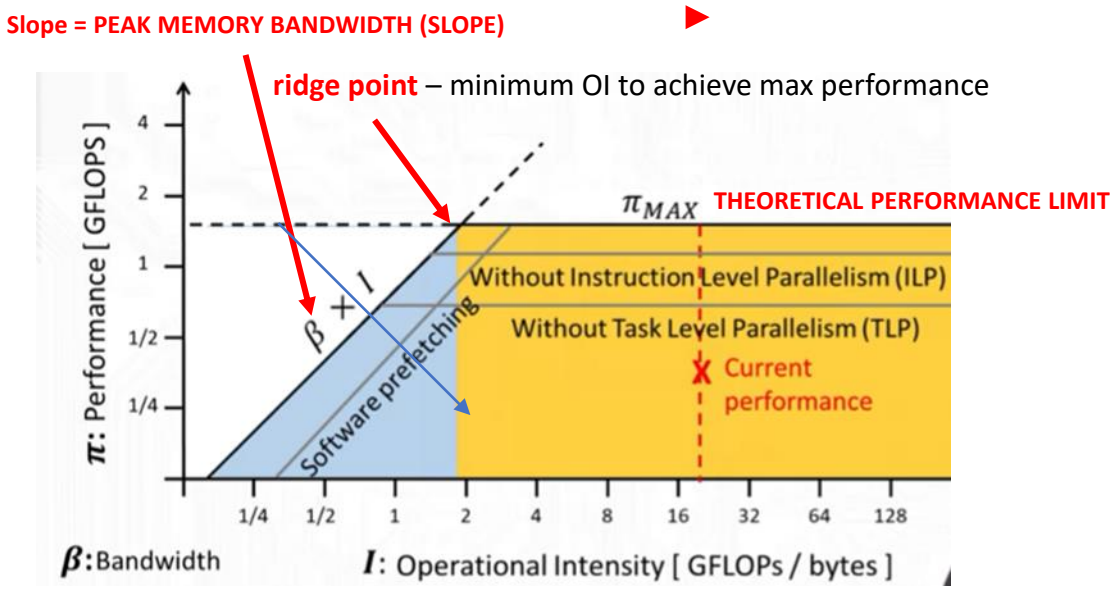
Why loop index from N-1 to 0

Output to next iteration
- West of successive PE
- North of the same PE
- Its North becomes North-West of the successive PE

Roofline Model

Auto-tuning Performance on Multicore Computers
Samuel Webb William December 17, 2008

- Method for “Bound and Bottleneck Analysis”
- Assumption: off-chip memory bandwidth will be a bottleneck
- Peak performance & Memory bandwidth are performance upper-bound – architecture specific
- Operation Intensity (OI) = W (# of operations) / M (off-chip bytes transferred)
- If Memory-Bound => memory width, memory ports
- Improve OI
  - by reduce M -> use Local memory
  - Increase W -> parallelism
- OI estimation first by static code analysis



Auto-Tuning Performance on Multicore Computers: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.418.178&rep=rep1&type=pdf
Performance Modeling for FPGAs: Extending the Roofline Model with High-Level Synthesis Tools http://downloads.hindawi.com/journals/ijrc/2013/428078.pdf

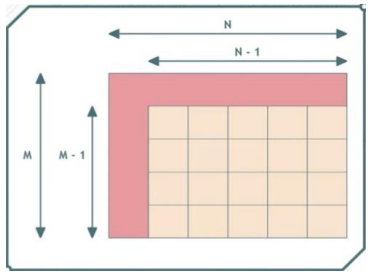
Operation Intensity Calculation

Calculating Operations

- Arithmetic Operation
- Comparison
- Indexing

```
#define N 256
#define M 2048
#define MATRIX_SIZE N * M
void compute_matrices(
char *string1, char *string2, int *max_index, int *similarity_matrix,
short *direction_matrix)  {
max_index[0] = 0;
for(index = N; index < MATRIX_SIZE; index++) {
  dir = CENTER;
  val = 0;

  i = index % N; // column index
  j = index / N; // row index

  if(i == 0) {  // first column
    west = 0;
    northwest = 0;
  } else {      // all columns but first
```

N-1




```
else {      // all columns but first
    north = similarity_matrix[index - N];
    match = ( string1[i] == string2[j] ) ? MATCH : MISS_MATCH;

    test_val = northwest + match;
    if(test_val > val){
      val = test_val;
      dir = NORTH_WEST;   }

    test_val = north + GAP_d;
    if(test_val > val){
      val = test_val;
      dir = NORTH;   }

    test_val = west + GAP_i;
    if(test_val > val) {
      val = test_val;
      dir = WEST;  }

    similarity_matrix[index] = val;
    direction_matrix[index] = dir;
    west = val;
    northwest = north;
    if(val > max_value) {
      max_index[0] = index;
      max_value = val;
  } } } }
```

(N-1)*(M-1)

Arithmetic Operations:

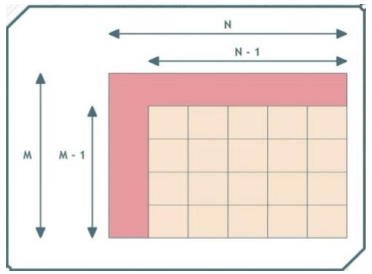
N*(M-1) * 3 : for loop

(N-1)*(M-1) * 4 : else

=》 N*(M-1) * 3 + (N-1)*(M-1) * 4

```
#define N 256
#define M 2048
#define MATRIX_SIZE N * M
void compute_matrices(
char *string1, char *string2, int *max_index, int *similarity_matrix,
short *direction_matrix)  {
max_index[0] = 0;
for(index = N; index < MATRIX_SIZE; index++) {
  dir = CENTER;
  val = 0;

  i = index % N; // column index
  j = index / N; // row index

  if(i == 0) {  // first column
    west = 0;
    northwest = 0;
  } else {      // all columns but first
```

N-1




```
else {      // all columns but first
    north = similarity_matrix[index - N];
    match = ( string1[i] == string2[j] ) ? MATCH : MISS_MATCH;

    test_val = northwest + match;
    if(test_val > val){
      val = test_val;
      dir = NORTH_WEST;   }

    test_val = north + GAP_d;
    if(test_val > val){
      val = test_val;
      dir = NORTH;   }

    test_val = west + GAP_i;
    if(test_val > val) {
      val = test_val;
      dir = WEST;  }

    similarity_matrix[index] = val;
    direction_matrix[index] = dir;
    west = val;
    northwest = north;
    if(val > max_value) {
      max_index[0] = index;
      max_value = val;
  } } } }
```

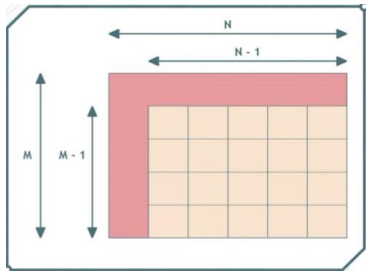
(N-1)*(M-1)

Comparison:

N*(M-1) * 2 + (N-1)*(M-1) * 5

```
#define N 256
#define M 2048
#define MATRIX_SIZE N * M
void compute_matrices(
char *string1, char *string2, int *max_index, int *similarity_matrix,
short *direction_matrix)  {
max_index[0] = 0;
for(index = N; index < MATRIX_SIZE; index++) {
  dir = CENTER;
  val = 0;

  i = index % N; // column index
  j = index / N; // row index

  if(i == 0) {  // first column
    west = 0;
    northwest = 0;
  } else {      // all columns but first
```

N-1



Index

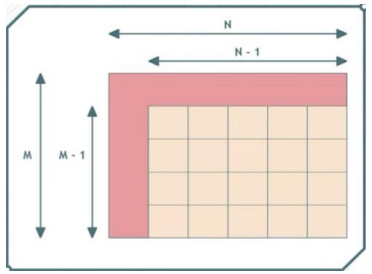
1+ (N-1)*(M-1) * 6

```
else {      // all columns but first
  north = similarity_matrix[index - N];
  match = (string1[i] == string2[j] ) ? MATCH : MISS_MATCH;

  test_val = northwest + match;
  if(test_val > val){
    val = test_val;
    dir = NORTH_WEST;   }

  test_val = north + GAP_d;
  if(test_val > val){
    val = test_val;
    dir = NORTH;   }

  test_val = west + GAP_i;
  if(test_val > val) {
    val = test_val;
    dir = WEST;  }

  similarity_matrix[index] = val;
  direction_matrix[index] = dir;
  west = val;
  northwest = north;
  if(val > max_value) {
    max_index[0] = index;
    max_value = val;
 } } } }
```

(N-1)*(M-1)

```
#define N 256
#define M 2048
#define MATRIX_SIZE N * M
void compute_matrices(
char *string1, char *string2, int *max_index, int *similarity_matrix,
short *direction_matrix)  {
max_index[0] = 0;
for(index = N; index < MATRIX_SIZE; index++) {
  dir = CENTER;
  val = 0;

  i = index % N; // column index
  j = index / N; // row index

  if(i == 0) {  // first column
    west = 0;
    northwest = 0;
  } else {      // all columns but first
```

N-1



Memory

1+ (N-1)*(M-1) * 6

```
else {      // all columns but first
  north = similarity_matrix[index - N];
  match = (string1[i] == string2[j] ) ? MATCH : MISS_MATCH;

  test_val = northwest + match;
  if(test_val > val){
    val = test_val;
    dir = NORTH_WEST;   }

  test_val = north + GAP_d;
  if(test_val > val){
    val = test_val;
    dir = NORTH;   }

  test_val = west + GAP_i;
  if(test_val > val) {
    val = test_val;
    dir = WEST;  }

  similarity_matrix[index] = val;
  direction_matrix[index] = dir;
  west = val;
  northwest = north;
  if(val > max_value) {
    max_index[0] = index;
    max_value = val;
 } } } }
```

(N-1)*(M-1)

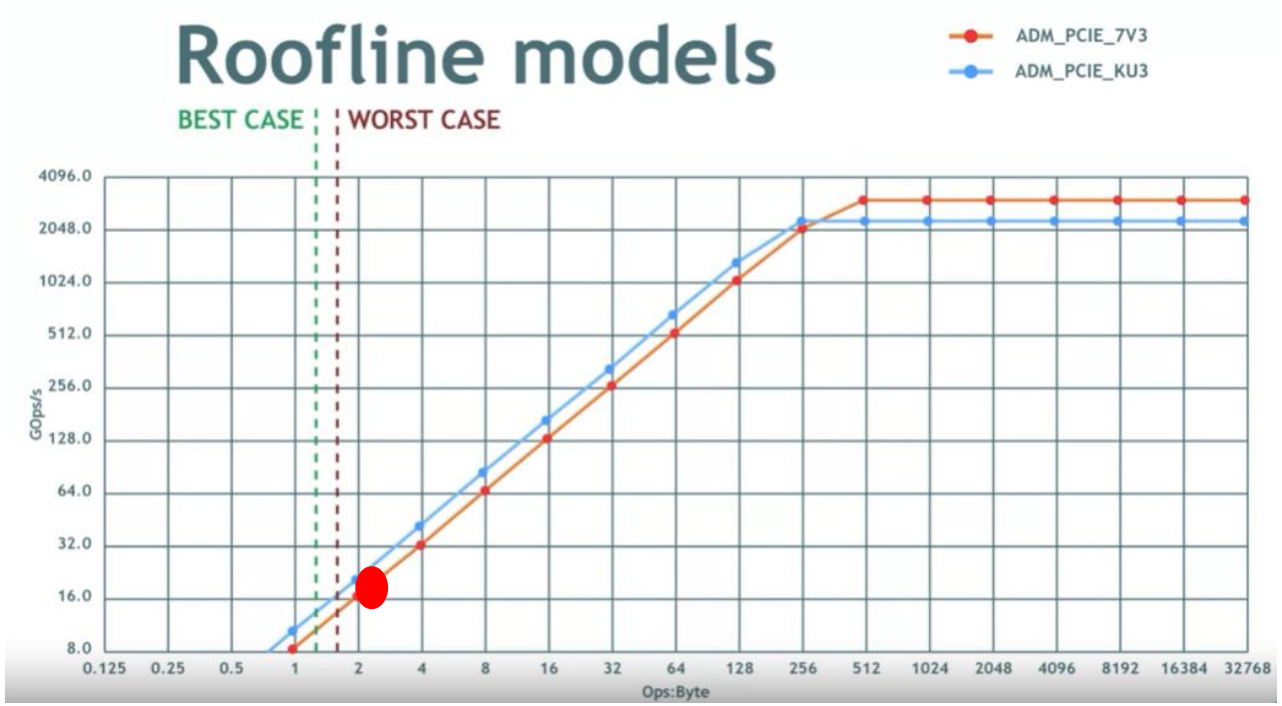
Operation Intensity

Arithmatic Operations:
N*(M-1) * 3 : for loop
(N-1)*(M-1) * 4 : else
=》 N*(M-1) * 3 + (N-1)*(M-1) * 4

Comparison:
N*(M-1) * 2 + (N-1)*(M-1) * 5

Index
1+ (N-1)*(M-1) * 6

Memory
1+ (N-1)*(M-1) * 6



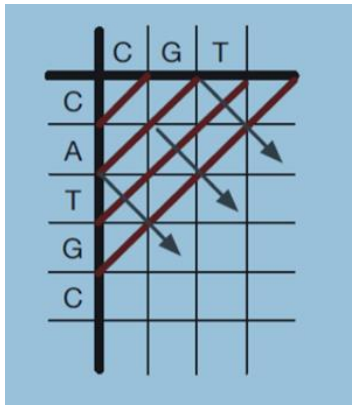
OI = (Arithemetic + Comparison + Index) / Memory

For N = 256, M = 2048

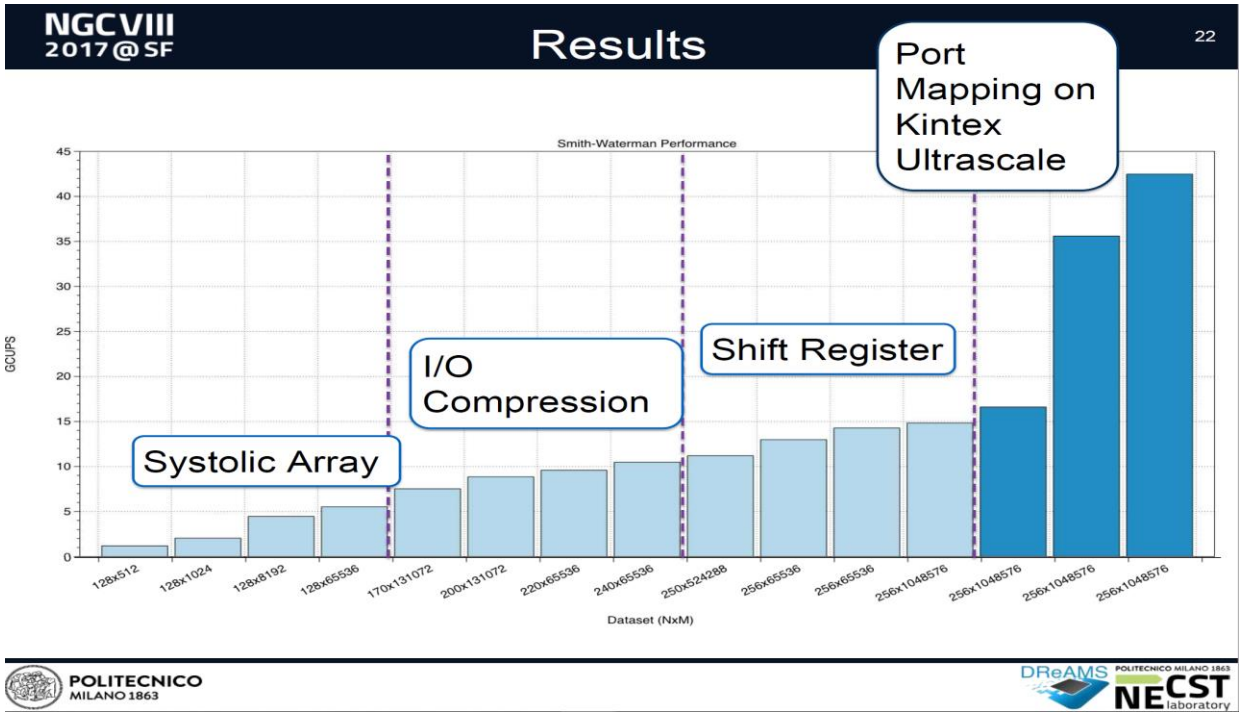
OI = 2.16

It shows it is a memory bound

Improvement



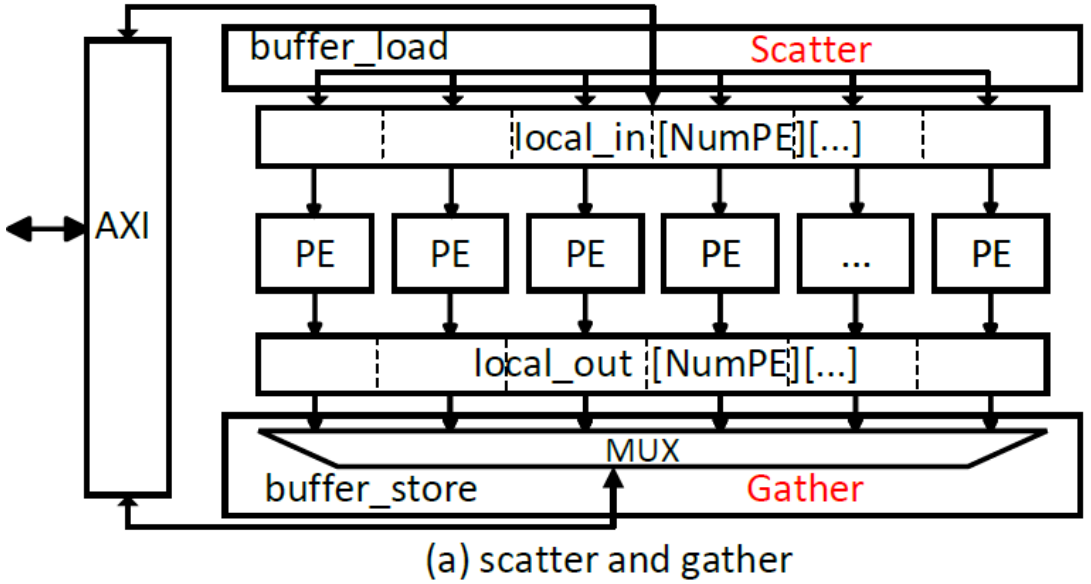
- Roofline model suggests it is memory bound
- Need to use two previous anti-diagonal value
- Use N PE (query[N])
- Parallelism & optimization
  - Systolic array
  - Memory bound -> input (A,G,C,T)/output (NW,W,N,C) compression uses 2-bits
  - Shift registers – slide through database
  - Use 2 ports of DRAM



Interconnect Issues

Common Interconnect Patterns in Accelerator

- Interconnect patterns causes timing degradation
- Data access pattern (MPI – Message Passing Interface)
  - Scatter
  - Gather
  - Broadcast
  - Reduce



(a) scatter and gather

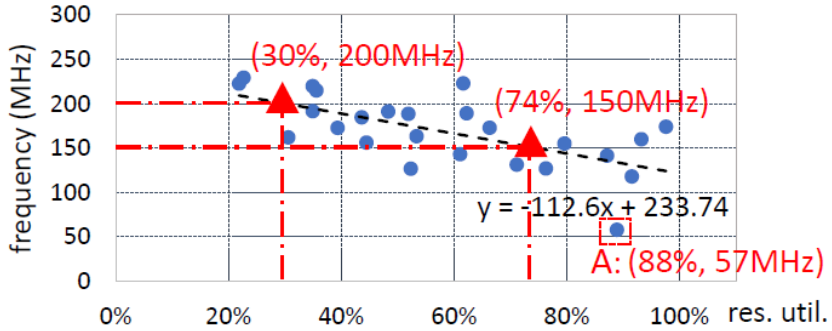
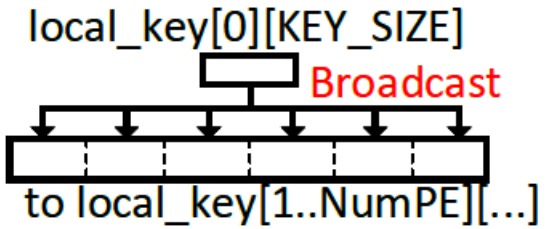
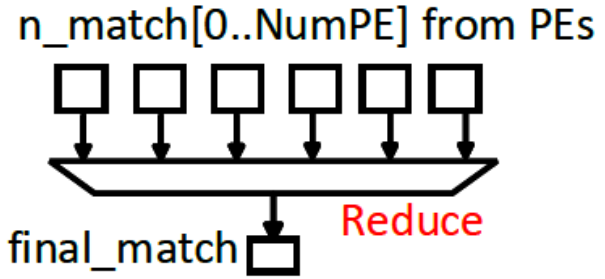


Fig. 1: Frequency vs. area: Freq decreases as design size scales out.



(b) broadcast



(c) reduce

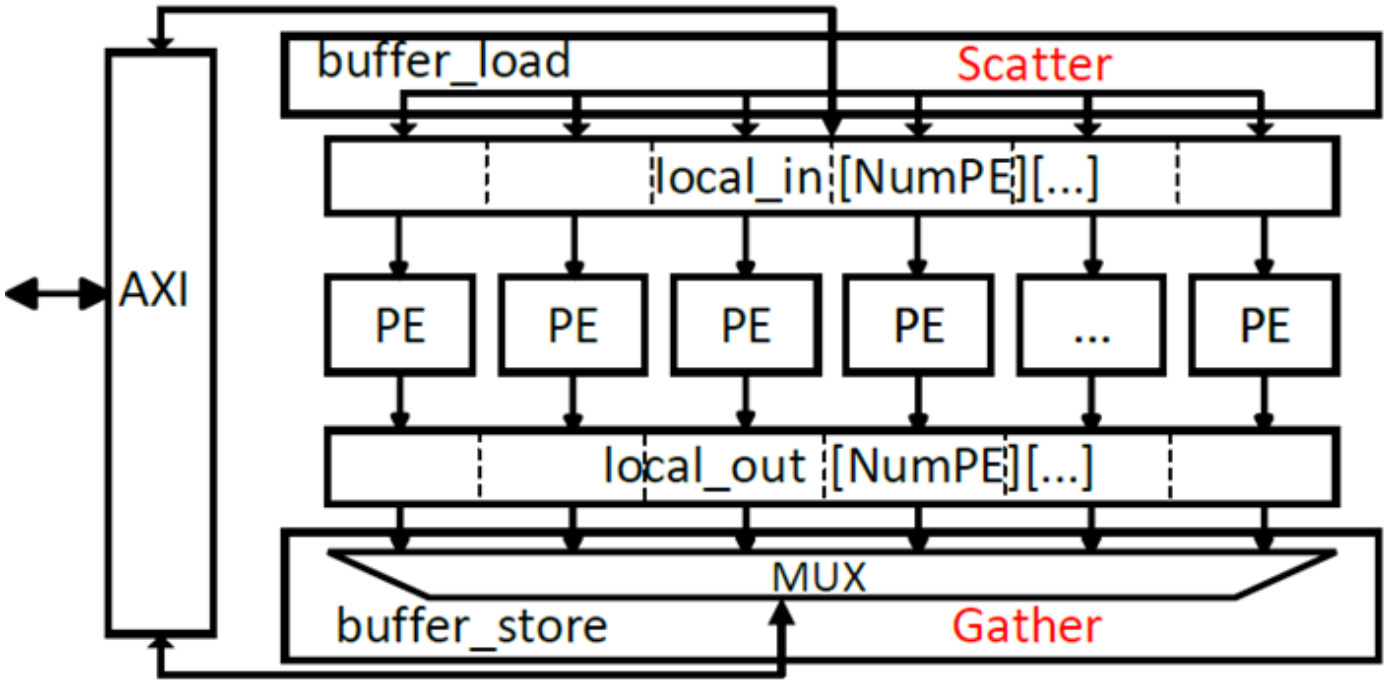
Latte: Locality Aware Transformation for High-Level Synthesis https://vast.cs.ucla.edu/sites/default/files/publications/fccm18-latte.pdf

Data Patterns in Application

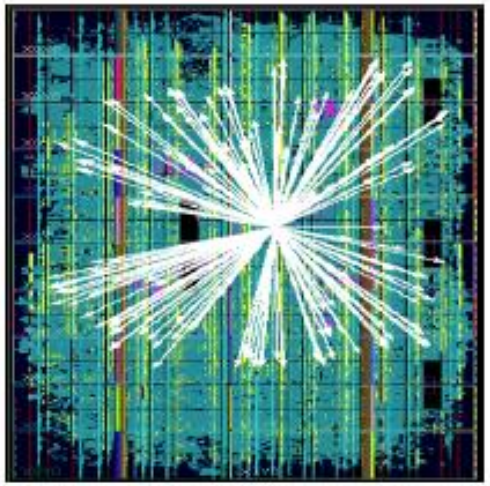
Benchmark	Domain	Scatter	Gather	Broadcast	Reduce
AES	Encryption	✓★	✓★	✓	
FFT	Signal	✓	✓★		
GEMM	Algebra	✓	✓★	✓★	
KMP	String	✓		✓	✓★
NW	Bioinfo.	✓	✓		
SPMV	Algebra	✓	✓★	✓	
STENCIL	Image	✓	✓★	✓	
VITERBI	DP	✓	✓		

Checkmark ✓ represents the design has the pattern.
A star ★ represents that a critical path lies in the pattern.
For broadcast, GEMM uses bc_in_compute while others use bc_by_copy.

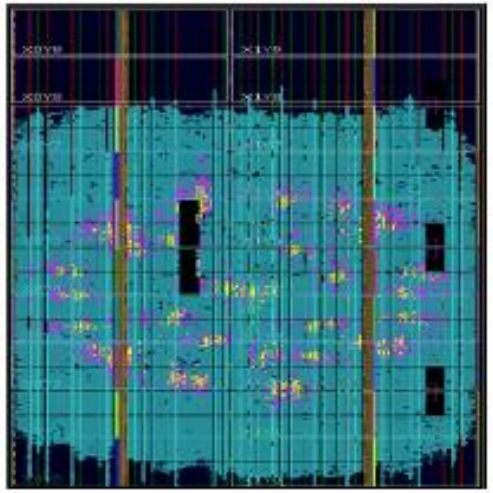
Layout



(a) scatter and gather



(a) Scatter pattern



(b) Gather pattern

Pipeline Transfer Controller PTC – Reduce interconnect loading Similar to Clock-Tree Routing

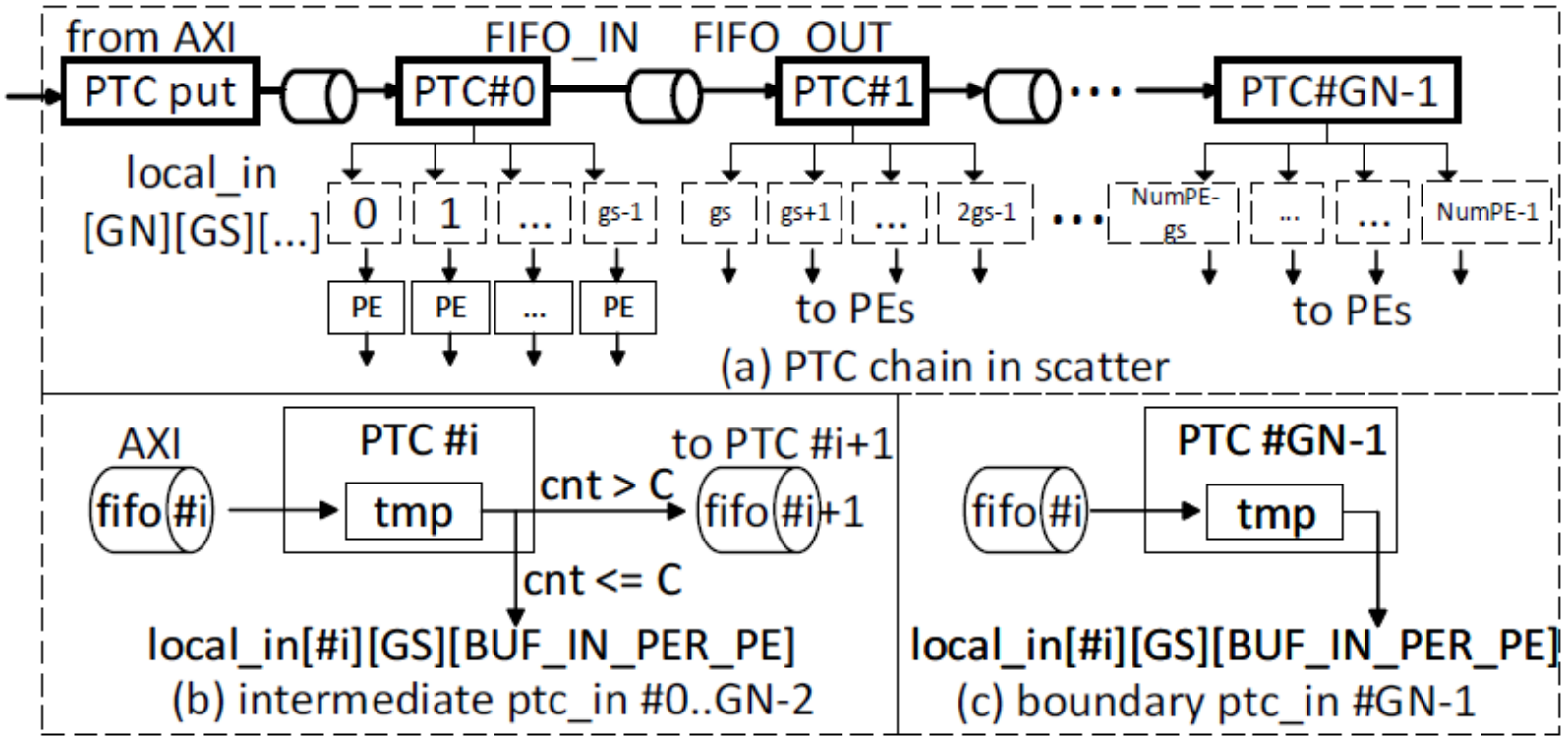


Fig. 7: Microarchitecture of PTC in scatter.

```
#include <hls_stream.h>
int local_in[GN][GS][BUF_IN_PER_PE];  // redef.
void PTC_load(
                    int local_in[GN][GS][],
                    int* global_in) {
#pragma HLS dataflow
  hls::stream<int> fifo[GN];    // FIFOs, in Fig. 7a
  ptc_put(global_in, fifo[0]);
  for(int i = 0; i < GN-2; i++){
    ptc_in(fifo[i], fifo[i+1], local_in[i], GN-1-i);}
  ptc_in(fifo[GN-1], local_in[GN-1]);
}
```

```
void ptc_put(int* global_in,
                    stream<int> & fifo)  {
   for (int i=0; i<NumPE; i++)
      for(int j = 0; j < BUF_IN_PER_PE; j++){
         #pragma HLS pipeline
         fifo << global_buf[i*BUF_IN_PER_PE+j];}
}
```

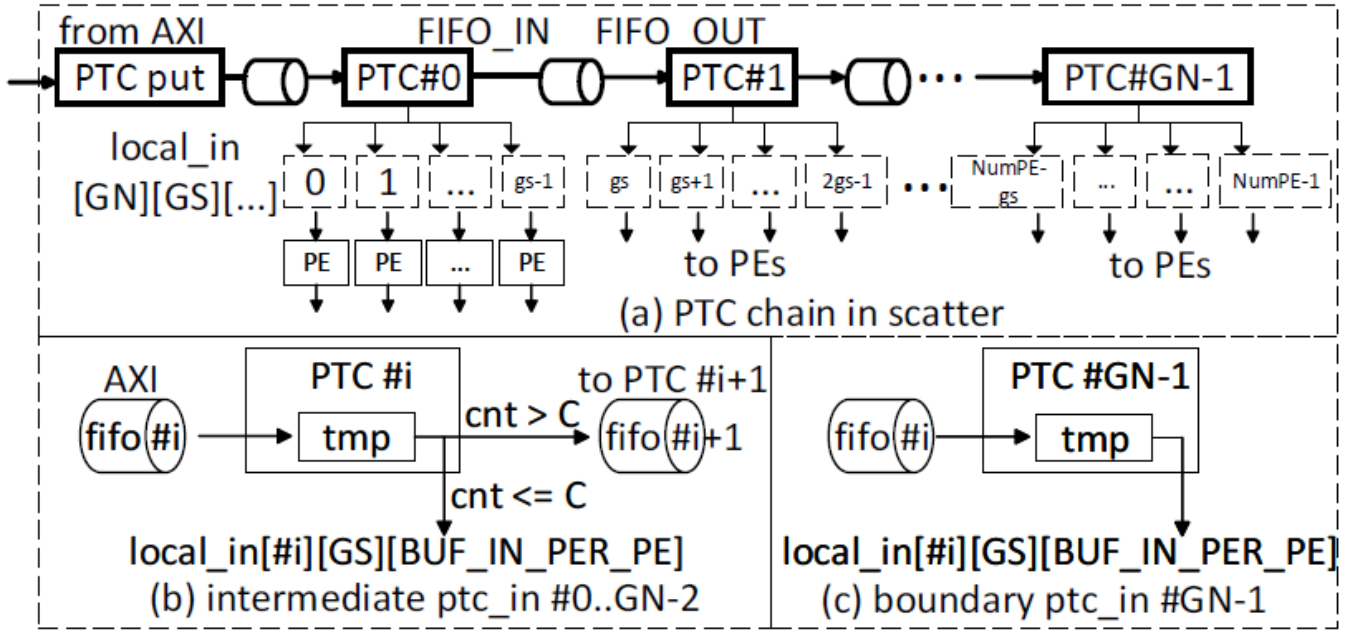
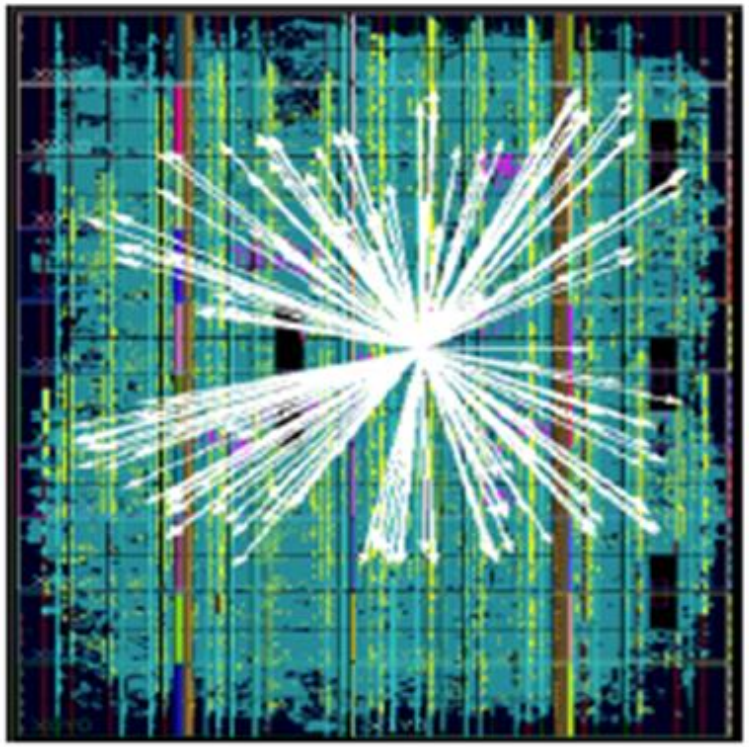


Fig. 7: Microarchitecture of PTC in scatter.

```
void ptc_in(        // #0..GN-2 ptc_in, in Fig. 7b
        stream<int>&fifo_in,
        stream<int> &fifo_out,
        int local_set[GS][BUF_IN_PER_PE],
        int todo)  {
  int i, j, k; int tmp;
   for(i= 0; i < GS; i++){          // to local first
      for(j = 0; j < BUF_IN_PER_PE; j++){
          tmp = fifo_in.read();
          local_set[i][j] = tmp;
   } }
   for(k=0; k < todo; k++)       // to next ptc
        for(i= 0; i < GS; i++){
          for(j = 0; j < BUF_IN_PER_PE; j++){
              tmp = fifo_in.read();
              fifo_out.write(tmp);
        } }
}
```

Distributed Wire Load to Improve Timing



(a) Scatter pattern

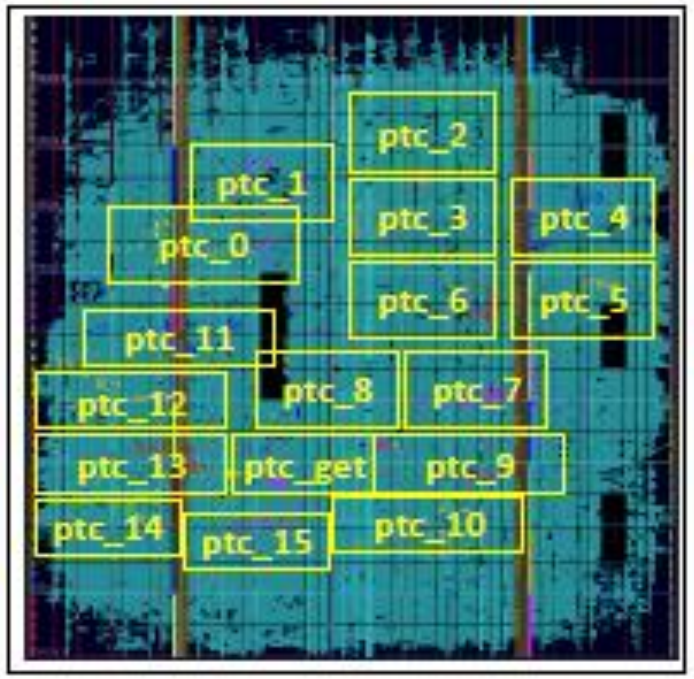


Fig. 13: PTC layout in FFT (N=64,GS=4,GN=16).

Result: Improve the timing of baseline HLS designs by 1.50x with only 3.2% LUT overhead on average

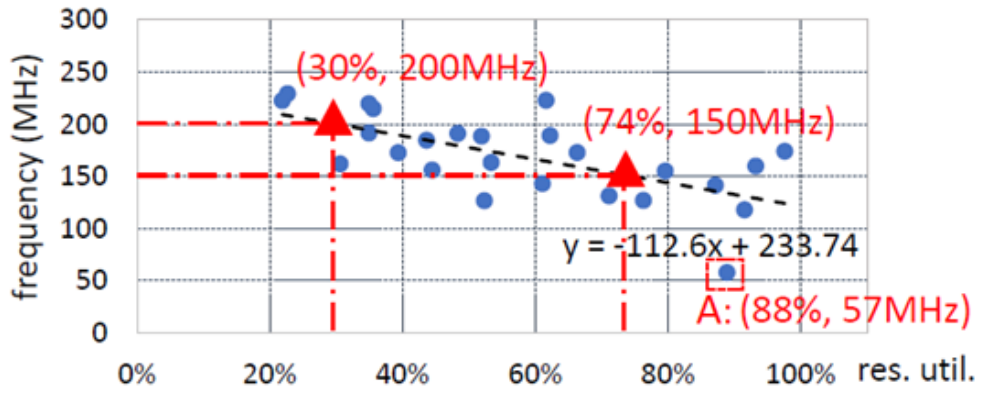


Fig. 1: Frequency vs. area: Freq decreases as design size scales out.

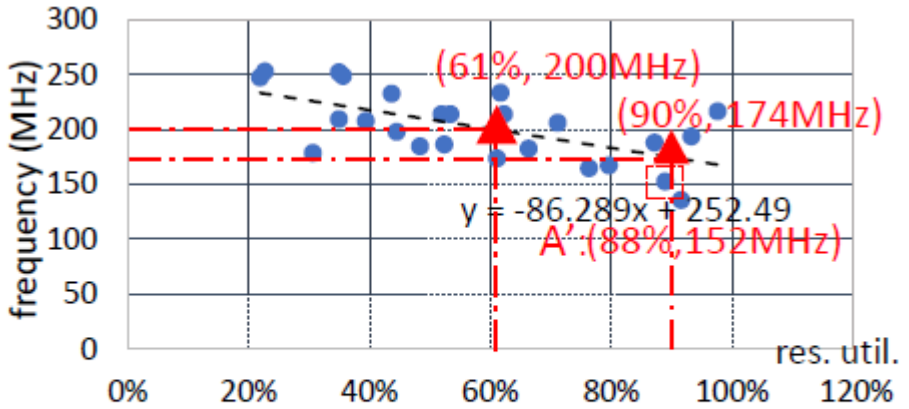


Fig. 12: Freq. degradation much less severe in Latte optimized designs.

Thank You

Speculative Pipeline Code Transformation

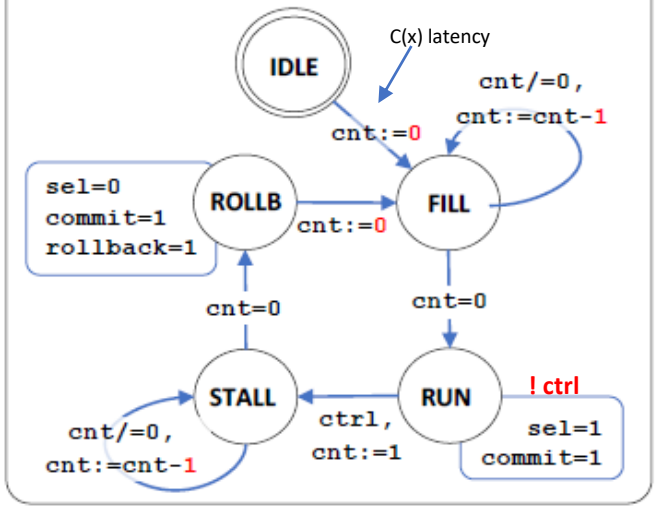
```
do {
  tmp=x;
  if(C(x)) {
    // slow
    x = S(tmp);
  } else {
    // fast
    x = F(tmp,y);
  }
  y = H(tmp,y)
} while (!x)
```

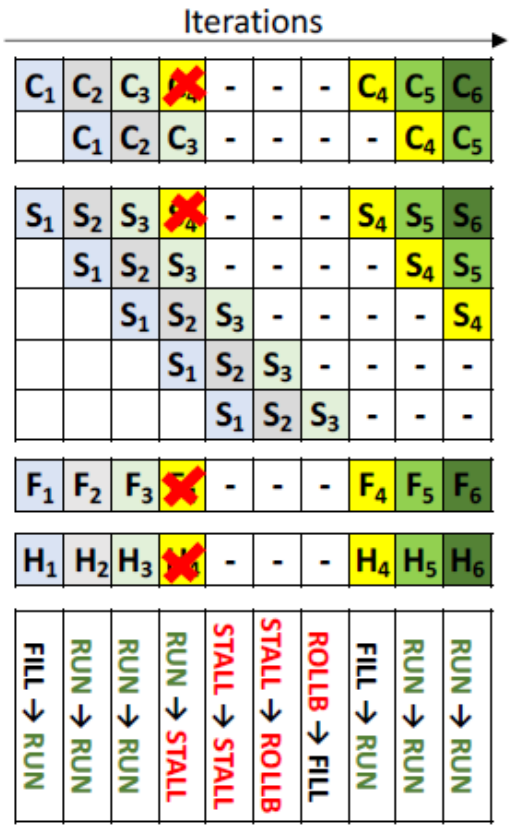
Latency estimates	
C	2
S	5
F	1
H	1

4 queues to save history values : ? level
- ctrl[] : C(x) – 2T
- mis_x[] : S(x) – 5T
- s_x[] = F(x) – **1T** **(speculative x = F())**
- s_y[] = H(x) – 1T

```
2  #pragma hls dependence variable=mis_x distance=5
   do {
   #pragma hls pipeline II=1
1    ctrl[t]  = C (s_x[t-2]);
     mis_x[t] = S (s_x[t-5]);
     s_x[t] = F (s_x[t-1],s_y[t-1]);
     s_y[t] = H (s_x[t-1],s_y[t-1]);

3    cs = nextState(cs,ctrl[t]);

4    if (cs.rollback) {
       s_y[t] = s_y[t-4];
       s_x[t] = mis_x[t]
     }
5    if (cs.commit) {
       x = cs.sel? s_x[t-1]:mis_x[t];
       y = s_y[t-1];
     }
     t++;
   } while(!(x && cs.commit)); 6
```

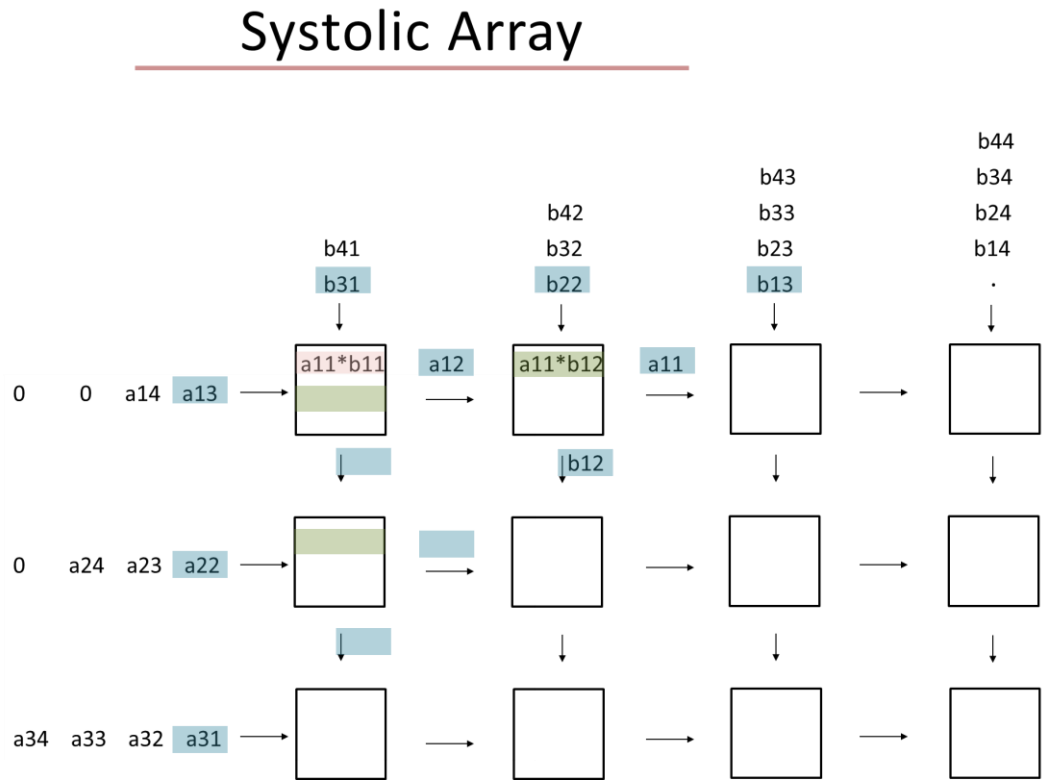
```
enum tstate = {IDLE, FILL, … }

struct fsm {
   int3 cnt;
   tstate cs;
   bool commit,rollback,sel;
} cs;
```





Vitis Tutorial – MM Systolic Array

```
systolic1:
    for (int k = 0; k < a_col; k++) {
#pragma HLS LOOP_TRIPCOUNT min = c_size max = c_size
    systolic2:
        for (int i = 0; i < MAX_SIZE; i++) {
#pragma HLS UNROLL
        systolic3:
            for (int j = 0; j < MAX_SIZE; j++) {
#pragma HLS UNROLL
                // Get previous sum
                int last = (k == 0) ? 0 : localC[i][j];

                // Update current sum
                // Handle boundary conditions
                int a_val = (i < a_row && k < a_col) ? localA[i][k] : 0;
                int b_val = (k < b_row && j < b_col) ? localB[k][j] : 0;
                int result = last + a_val * b_val;

                // Write back results
                localC[i][j] = result;
            }
        }
    }
```

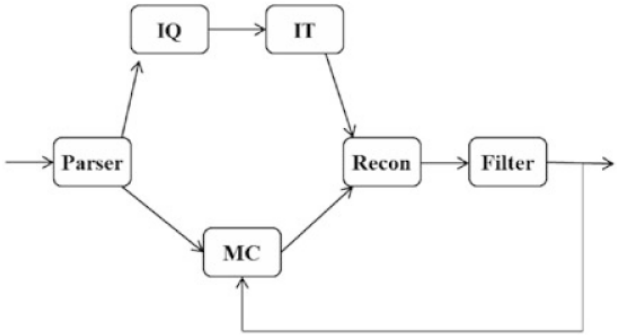


Issues with System-Level Design

- Scheduling of Submodules
  - Execution order for parallelism and data locality
  - Pipeline or sequential execution
  - Data granularity for the pipeline
- Resource Trade-offs between submodules:
  - Design option for each submodule (performance/area)
  - # of parallel duplications for each submodules
- Memory allocation for data
  - On-chip buffer or off-chip
  - Folding/unroll and the address transformation
- Access efficiency
  - Partition patterns in memory partition for on-chip access parallelism
  - Prefetcing buffer and address sequence optimization

- Submodule design space exploration
  - HLS does not handle it for you.

```
for (m = 0; m < N; m++){
    Parser(I → B, A);
    for (i = 0; i < 6; i++) {
        IQ(A → D); IT(D → E);
    }
    MC(B → C);
    Recon(C, E → O)
    Filter(C, E → O)
}
```



Architecture Exploration – An Example

a
```
for (y=0; y<8;y++)
  for (x = 0; x<8; x++)
    for (t=0; t < 8; t++)
      out[y,x]+=in[y,t]*c[t,x];
```

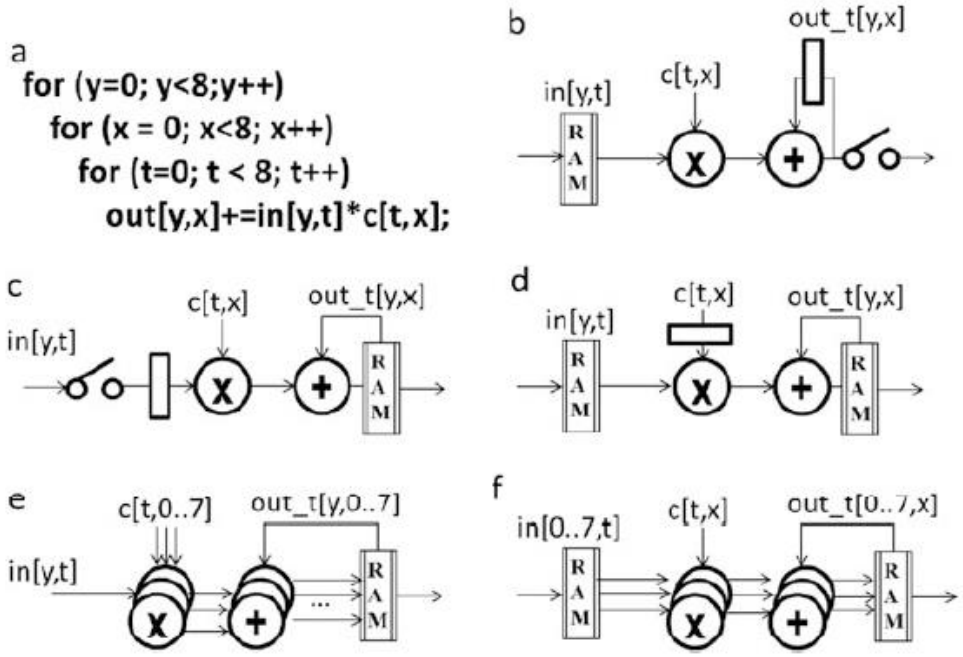


Fig. 8.2 Microarchitecture options for IDCT. (a) Original code of IDCT. (b) Single pipeline along t. (c) Single pipeline along x. (d) Single pipeline along y. (e) Parallel version of (c). (f) Parallel version of (d)

a
```
for (y=0; y<8;y++)
  for (x=0; x<8; x++)
    for (t=0; t<8; t++)
      out[y,x]+=in[y,t]*c[t,x];
```

b
```
for (t=0; t<8; t++)
  for (y=0; y<8;y++)
    for (x=0; x<8; x++)
      out[y,x]+=in[y,t]*c[t,x];
```

c
```
for (t=0; t<8; t++)
  for (y=0; y<8;y++)
    for (x=0; x<8; x++) {
      #pragma HLS pipeline
      #pragma HLS unroll
      out[y,x]+=in[y,t]*c[k,t];
    }
```

d
```
for (t=0; t<8; t++)
  for (y=0; y<8;y++)
    for (x=0; x<8; x++) {
#pragma HLS pipeline
#pragma HLS unroll
#pragma HLS partition out dim=2 factor=8
#pragma HLS partition c dim=2 factor=8
      out[y,x]+=in[y,t]*c[k,t];
    }
```

e
```
data_type out_t[8][8];
for (t=0; t<8; t++)
  for (y=0; y<8;y++) {
    data_type in_t = *in;
    for (x=0; x<8; x++) {
#pragma HLS pipeline
#pragma HLS unroll
#pragma HLS partition out_t dim=2 factor=8
#pragma HLS partition c dim=2 factor=8
      out_t[y,x]+=in_t*c[k,t];
      if (t==7) *out = out_t[y,x];
    }
  }
```

Fig. 8.3 Code rewrite for the implementation of Fig. 8.2e. (a) Original code. (b) Execution order change. (c) Micro-architecture pragmas. (d) Memory partitioning. (e) Data reuse and interface handling

https://link.springer.com/book/10.1007/978-3-319-26408-0

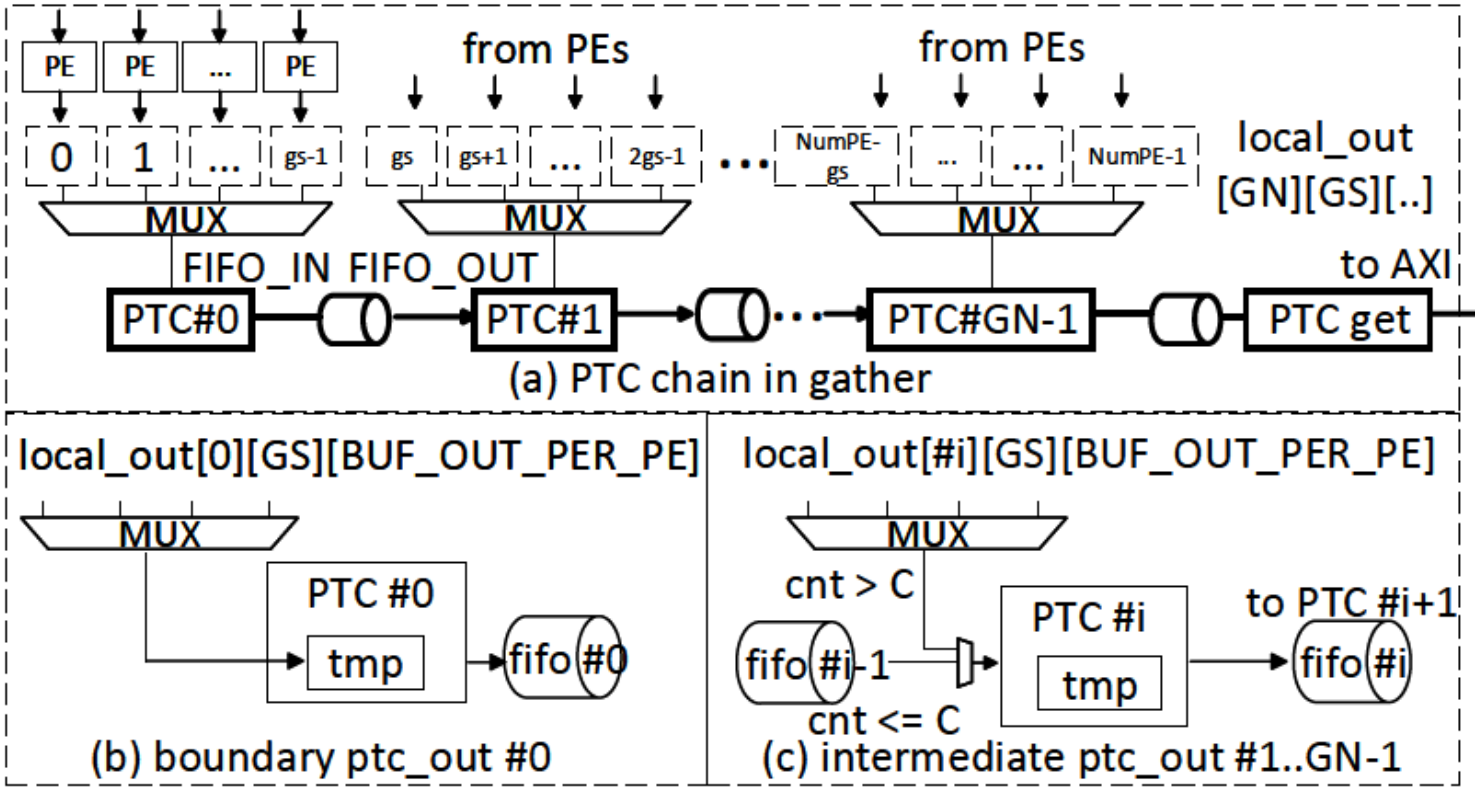


Fig. 9: Microarchitecture of PTC in gather.

MMM: Optimize Memory Access

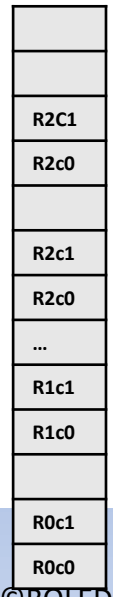
- Cache & use DRAM row burst access
- Reorder the loop k->i->j to explore locality
- SIZE=6, Block_SIZE = 2
- Matrix A[6][6]
  - Access DRAM in col
  - Cache Nb rows
  - Permute Data, stream by column
- Matrix B[6][6]
  - Burst on row access
- Cache AB block for repeated accumulation

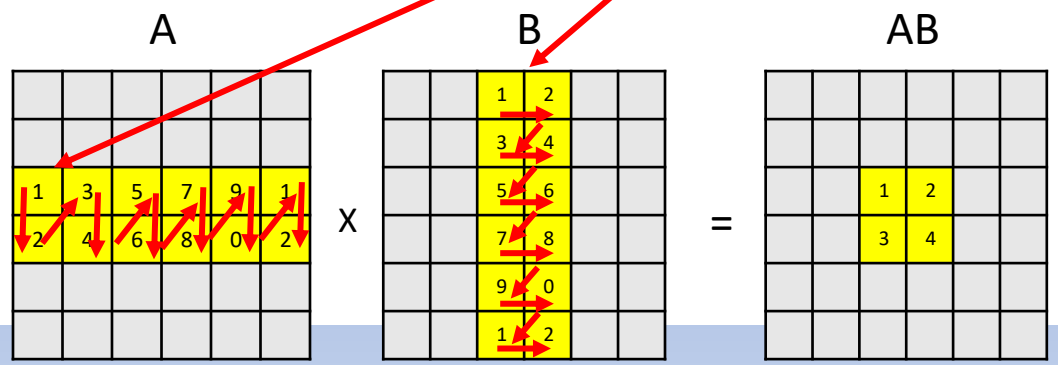
```
for(int it1 = 0; it1 < SIZE; it1 = it1 + BLOCK_SIZE) {
    for(int it2 = 0; it2 < SIZE; it2 = it2 + BLOCK_SIZE) {
        row = it1; //row + BLOCK_SIZE * factor_row;
        col = it2; //col + BLOCK_SIZE * factor_col;

        for(int k = 0; k < SIZE; k++) {
            for(int i = 0; i < BLOCK_SIZE; i++) {
                if(it % (SIZE/BLOCK_SIZE) == 0) strm_matrix1_element.a[i] = A[row+i][k];
                strm_matrix2_element.a[i] = B[k][col+i];
            }
            if(it % (SIZE/BLOCK_SIZE) == 0) strm_matrix1.write(strm_matrix1_element);
            strm_matrix2.write(strm_matrix2_element);
        }
        blockmatmul(strm_matrix1, strm_matrix2, block_out, it);

        for(int i = 0; i < BLOCK_SIZE; i++)
            for(int j = 0; j < BLOCK_SIZE; j++)
                matrix_hwout[row+i][col+j] = block_out.out[i][j];
        it = it + 1;
    }
}
```

Host / IOP

```
#pragma HLS DATAFLOW
int counter = it % (SIZE/BLOCK_SIZE);
static DTYPE A[BLOCK_SIZE][SIZE];
if(counter == 0){ //only load the A rows when necessary
    loadA: for(int i = 0; i < SIZE; i++) {
        blockvec tempA = Arows.read();
        for(int j = 0; j < BLOCK_SIZE; j++) {
            #pragma HLS PIPELINE II=1
            A[j][i] = tempA.a[j];
        }
    }
}
DTYPE AB[BLOCK_SIZE][BLOCK_SIZE] = { 0 };
partialsum: for(int k=0; k < SIZE; k++) {
    blockvec tempB = Bcols.read();
    for(int i = 0; i < BLOCK_SIZE; i++) {
        for(int j = 0; j < BLOCK_SIZE; j++) {
            AB[i][j] = AB[i][j] + A[i][k] * tempB.a[j];
        }
    }
}
writeoutput: for(int i = 0; i < BLOCK_SIZE; i++) {
    for(int j = 0; j < BLOCK_SIZE; j++) {
        ABpartial.out[i][j] = AB[i][j];
    }
}
}
```

Memory floorplan – row-major





Need to cache the full rows of A array => Not scalable

Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency

- https://ieeexplore.ieee.org/document/9218718

GraphLily: Accelerating graph linear algebra on HBMequipped FPGAs.

Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between FPGA and GPU