18-643 Lecture 10: FPGA Memory Architecture

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today: think about memory from a spatial (non-monolithic) computing perspective

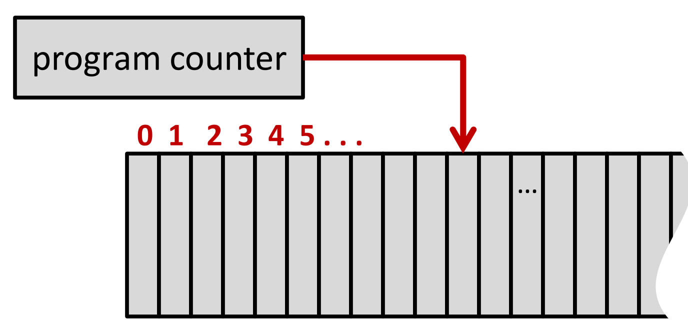
  (I assume you have taken a comp arch course)

- Notices
  - Handout #5: lab 2, due noon, 10/9
  - Project status report due each Friday
- Readings: (see lecture schedule online)
  - Intel® FPGA SDK for OpenCL™ Pro Edition: Best Practices Guide
  - Compute Express Link™ 2.0 White Paper

Stored Program Architecture
a.k.a. von Neumann

- Memory holds both program and data
  - instructions and data in a linear memory array
  - instructions can be modified as data
- Sequential instruction processing
  1. program counter (PC) identifies current instruction
  2. fetch instruction from memory
  3. update some state (e.g. PC and memory) as a function of current state according to instruction
  4. repeat

What I mean by Monolithic Abstraction



Dom… paradigm since its invention

from 18-447

Necessary Awareness #1: Fast means Small and Expensive

- Bigger is slower
  - SRAM 512 Bytes @ sub-nsec
  - SRAM KByte~MByte @ nsec
  - DRAM GByte @ ~50 nsec
  - SSD TByte @ msec
  - Hard Disk TByte @ ~10 msec
- Faster is more expensive (dollars and chip area)
  - SRAM ~$10K per GByte
  - DRAM ~$10 per GByte
  - “Drives” ~$0.1 per GByte

Treat the values as ×/÷ 3x

Necessary Awareness #2: Locality is Good

- Temporal: after accessing **A**, how many other distinct addresses before accessing **A** again

not how many ***times***?

- Spatial: after accessing **A**, how many other distinct addresses before accessing a <u>near-by</u> **B**

not what you think it means

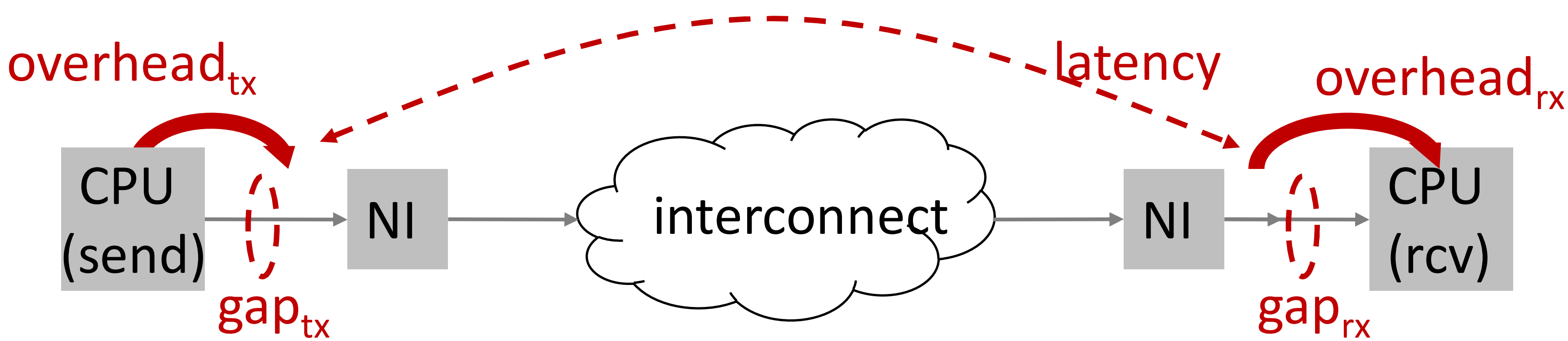
- Good locality implies
  - easier to cover more of working-set with a small (fast) memory
  - lower BW and/or more efficient use of BW to large (slow) memory

MMM::good; sparse MMM::bad; streaming::1 of 2

Necessary Awareness #3: Data Movement not Free

- Latency: transit time between source and dest
- Overhead: dead time spent in the act of sending or receiving <u>not overlapped with concurrent compute</u>
- Gap: wait time in between successive send's or receive's due to limited transfer BW

see LogP [Culler, et al., PPoPP93]



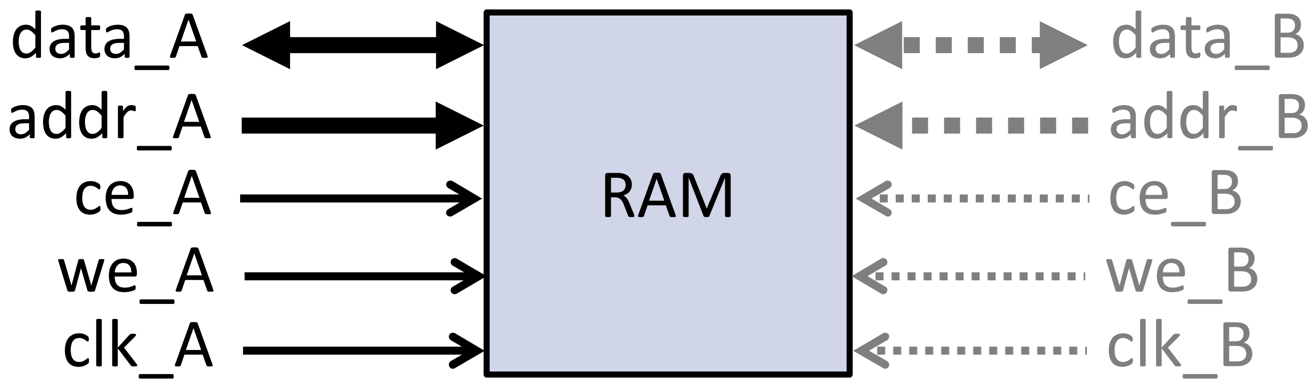
Necessary Awareness #3.1: Memory Perf Matters

- An algorithm has a cost in terms of operation count
  - $\text{runtime}_{\text{compute-bound}}$ = # operations / FLOPS
- An algorithm also has a cost in terms of number of bytes communicated (ld/st or send/receive)
  - $\text{runtime}_{\text{BW-bound}}$ = # bytes / BW
- Which one dominates depends on
  - ratio of FLOPS and BW of platform
  - ratio of ops and bytes of algorithm
- Average Arithmetic Intensity (AI)
  - how many ops performed per byte accessed
  - # operations / # bytes

accounting for communication cost with memory

DRAM, SRAM and all that

On-Chip Fast/Small Memory



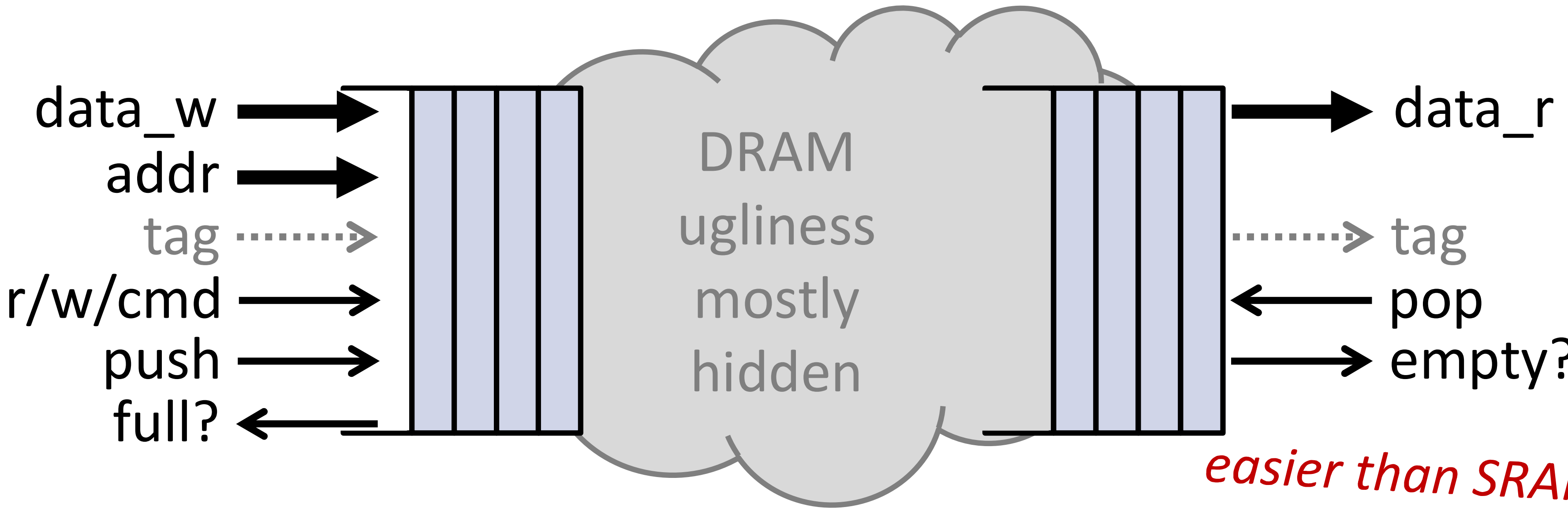
- LUT-RAM: 64x1b or 32x2b
  - dual-port: [sync write/async read] + async read
  - also good as one 32b or two 16b shift-regs
- BRAM: 18Kb, variable aspect ratio 1~36b wide
  - true dual-port: 2x[sync read/write], separate clocks
  - fast-enough to be double-pump'ed
- Become FF array if infeasible as hard macro

Xilinx universe

FPGA Memory Peculiarities

- Quantized, disjoint memory options
  - jumps between FF-based vs. LUT-RAM vs. BRAMs
  - choose from fixed menu of sizes and aspect ratios
  - subtle limitations (comb-read, multi-write)
- Large memory (BRAM) abnormally fast and “free” until your run-out
- Must manage RAM usage
  - don’t waste BRAM on small buffers
  - tune buffer sizes to natural granularities, e.g., zero incremental cost to go from 1KB to 2KB
  - pack buffers to share same physical array

Off-Chip Memory: DRAM



easier than SRAMs since coarser, further away from "action"

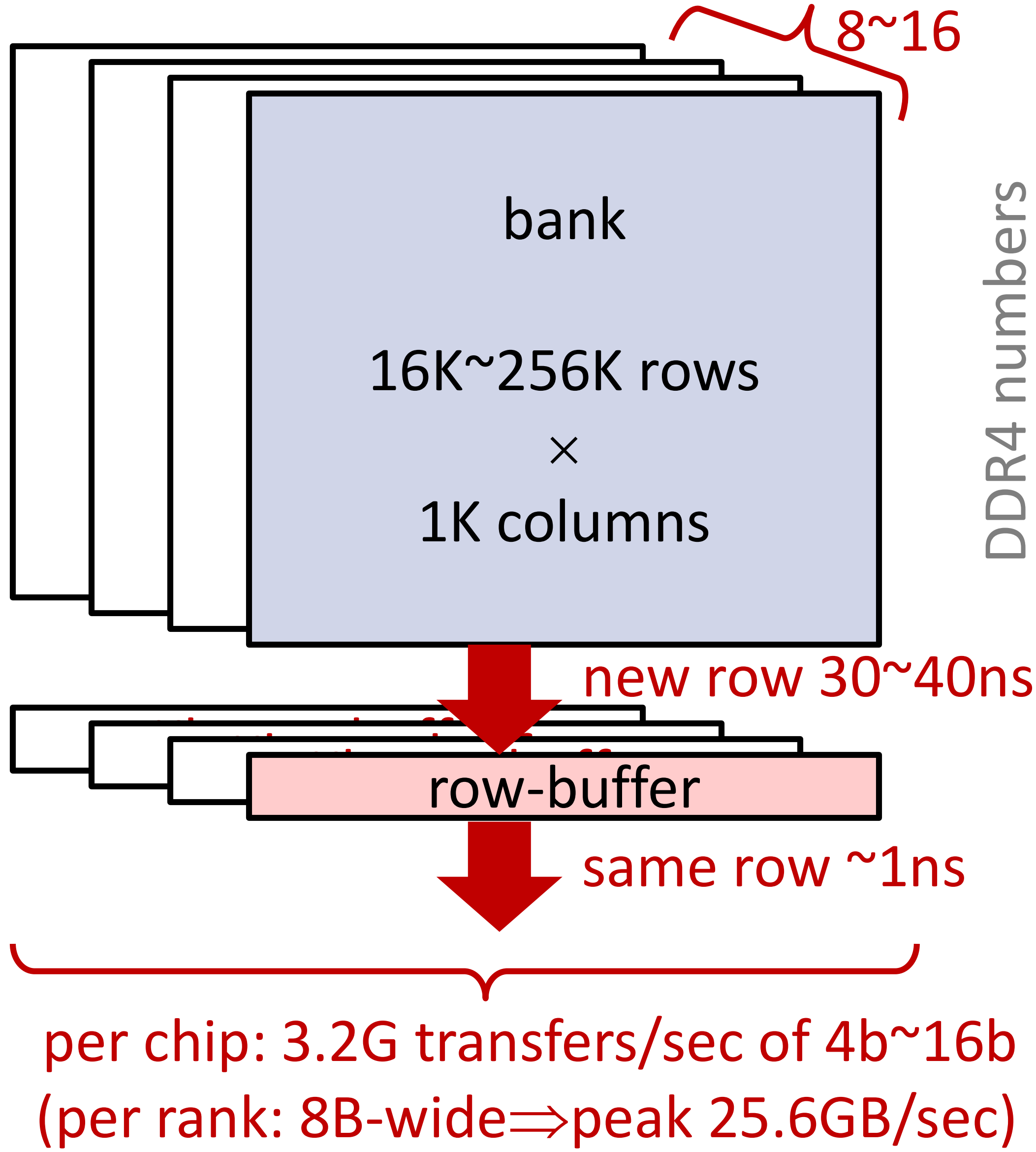
- Simple asynchronous request/reply queues
  - in-order or out-of-order (need tags)
  - multiple queues, separate read/write queues
- Need very wide data word for bandwidth

4B@200MHz is only 0.8GB/sec

- Long and variable <u>but predictable</u> latency

Access Pattern affects Latency and BW

- DRAM organization
  - (multiple ranks per DIMM)
  - (multiple chips per rank)
  - multiple banks per chip
- Per bank
  - long delay to new row
  - very fast to same row
- Chip/package/board add to total latency
- Rows refreshed every 64ms
  - bank unavailable for 30~40ns at a time
  - avg. ~1% unavailability



FPGAs gaining more SRAM capacity and DRAM bandwidth

- Interest in FPGA computing sets new "balance"
- 10 years ago
  - single-digit GB/sec to DRAM
  - single-digit Mbit SRAM
- New FPGAs (more than Moore increase)
  - GPU-level memory bandwidth (HBM, HMC, ...)
  - 10s MByte SRAM
  - also add GPU-level FP throughput

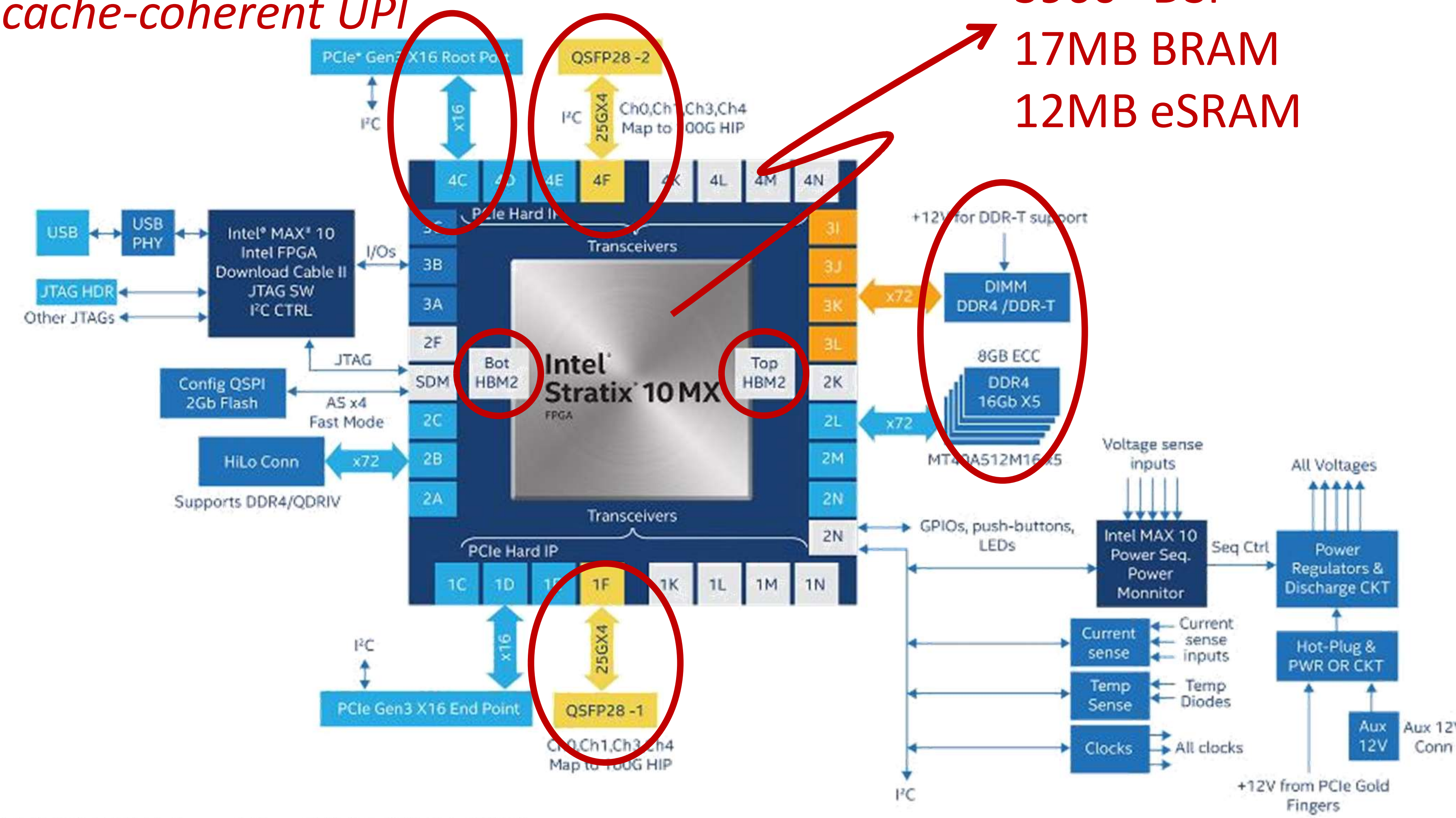
No free lunch in Watt/Perf though

- Will we see hardened memory hierarchy?

E.g., Stratix-10 MX

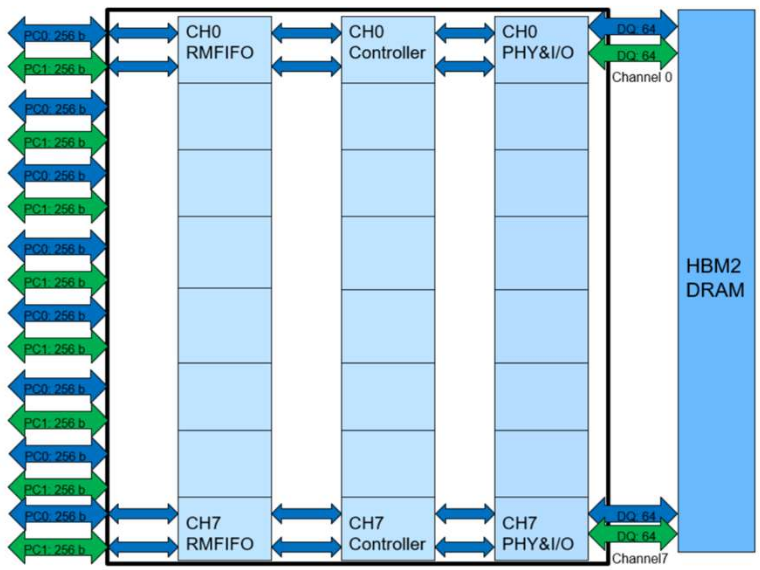
Stratix-10 DX has cache-coherent UPI

2M "Logic Elements"
3960 "DSP"
17MB BRAM
12MB eSRAM



HBM2 on Stratix-10 MX

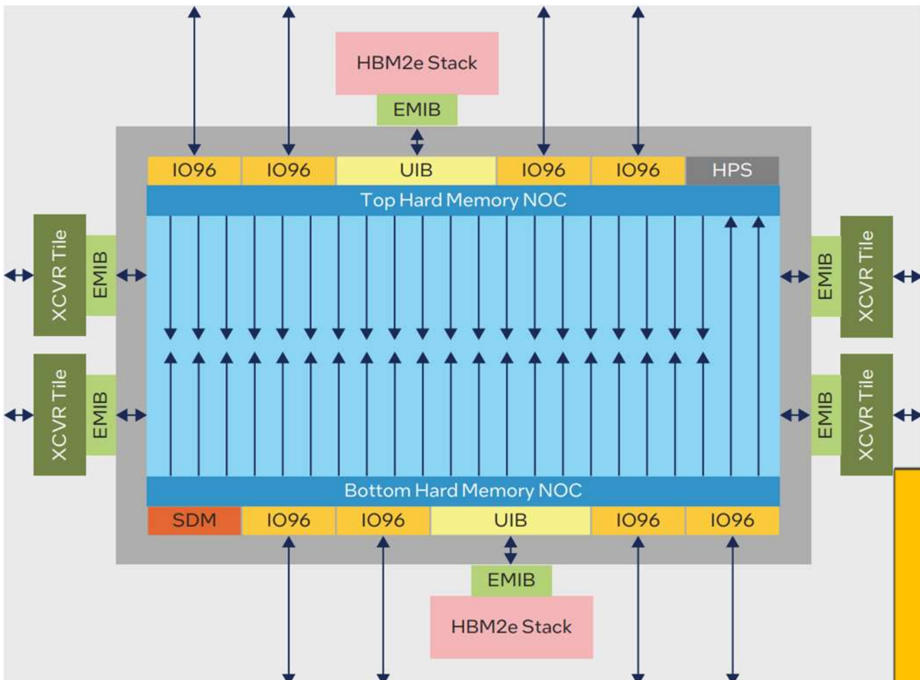
- Not a monolithic structure or abstraction
- 32 x “pseudo” channels (32B wide upto 500MHz)
  - each channel to a separate DRAM array
  - 16GB/sec/chnl if bursting consecutive addresses
- 512GB/sec if algorithm
  - partition data 32 ways
  - issue 32 concurrent,
    burst accesses wo. conflict
  - digest 32 x 32B per cyc
  - run at 500MHz



[Intel Stratix 10 HBM2 Architecture]

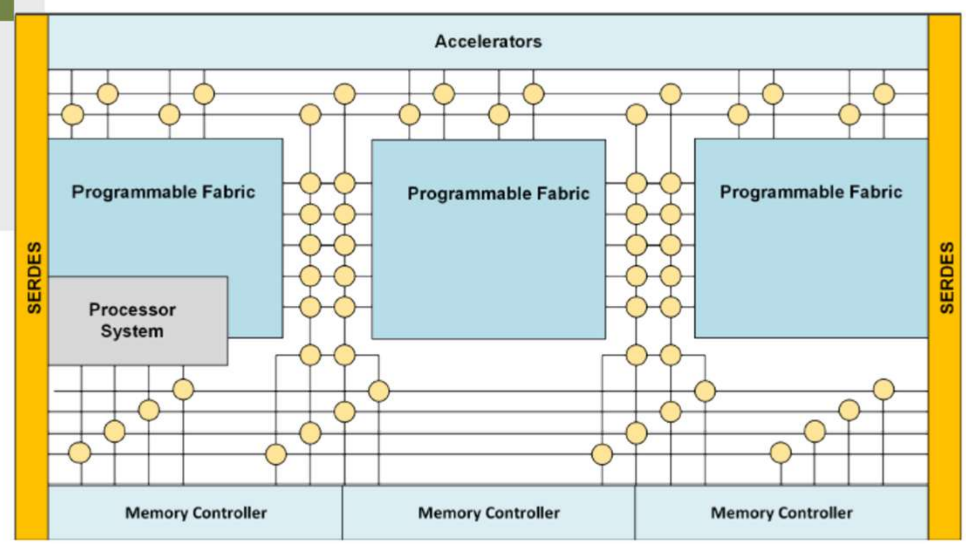
No walk-in-the-park on GPUs either

Hard NoC: making memory BW usable



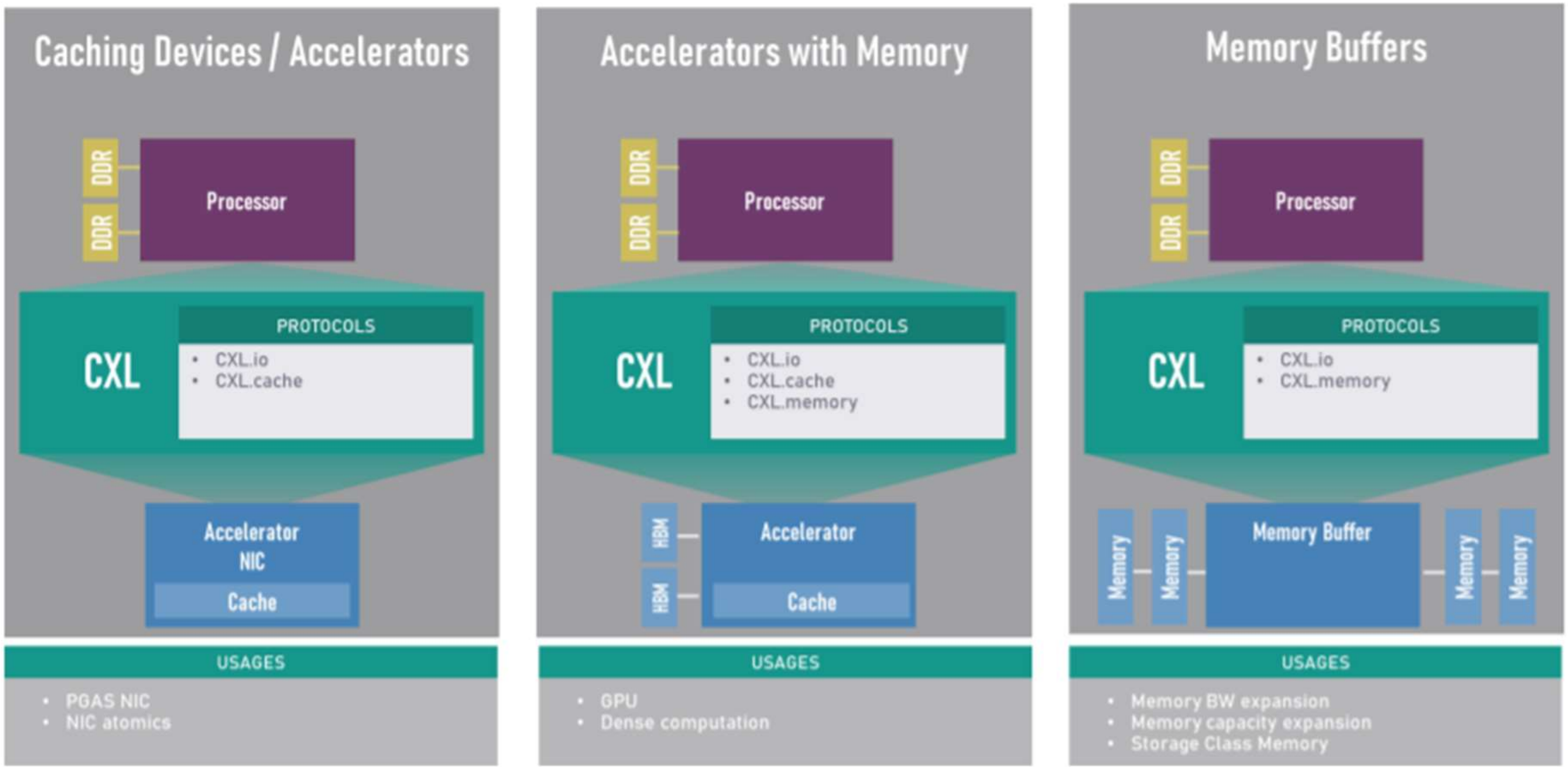
Intel Agilex M

- Hardened NoC to distribute memory BW into fabric
- Area and energy prohibitive as soft logic



Xilinx Versal

Compute Express Link (CXL)



[Compute Express Link™ 2.0 White Paper]

Device can cache host memory coherently

Host/Device symmetric and coherent

Host can cache device memory

In PCI, host cannot cache device space; device can read host space coherently with host cache but cannot cache

Memory Organization as Design Optimization

Classic Memory Hierarchy 101
(pre-multicore)

- Memory hierarchy level i has access time of t_i
- Perceived access time T_i is longer than t_i
  - a chance (hit-rate h_i) you find what you want $\Rightarrow t_i$
  - a chance (miss-rate m_i) you don't find it $\Rightarrow t_i+T_{i+1}$
  - $h_i + m_i=1.0$
- In general

$$T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})$$

$$T_i = t_i + m_i \cdot T_{i+1}$$

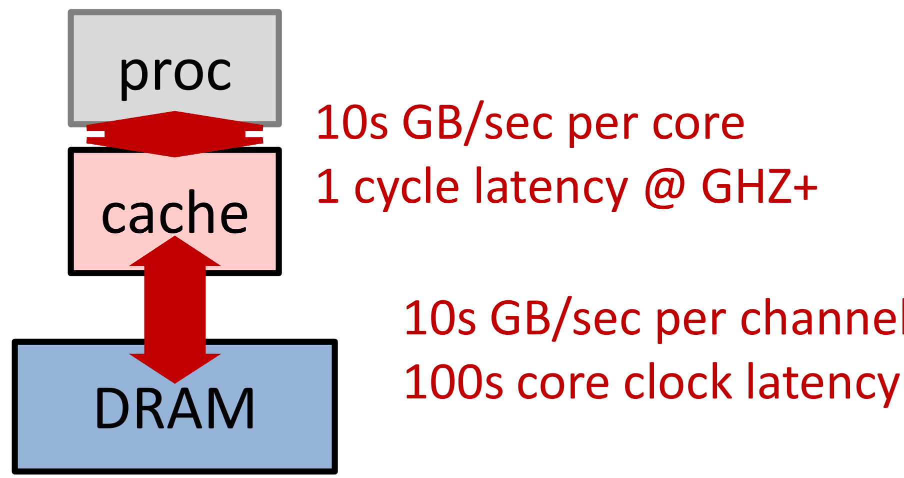
think this of as "miss penalty"

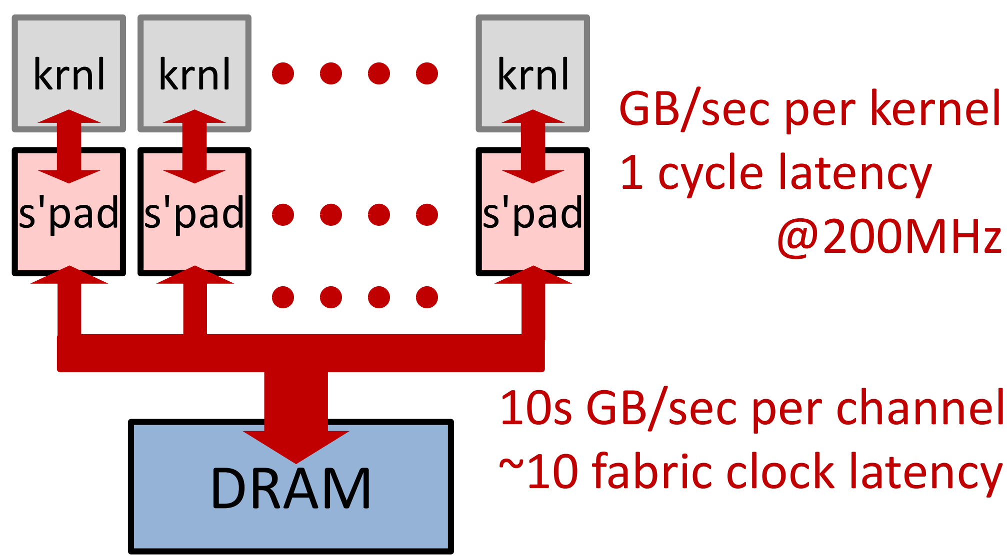
Note: h_i and m_i are of the references missing at level $i-1$

$h_{bottom\text{-}most}=1.0$

FPGA hierarchy used differently

- 200MHz soft-logic cache
  - a miss to DRAM is not too many cycles away
  - if 4-byte access, 100% hit-rate only 0.8 GB/sec
- Remember to think ***SPATIAL***
  - distributed concurrent **¡bandwidth!** for spatial kernels
  - reduce off-chip memory bandwidth and **¡power!**





Cache vs Scratchpad

- Manual scratchpad is easy for regular/structured locality
  - per-kernel scratchpad more opportunity and benefit in specialization
  - HW management does not lengthen code
  - prefetching can hide memory latency completely
  - 95% of the time: streaming or double-buffering

  These easy cases actually against cache heuristics

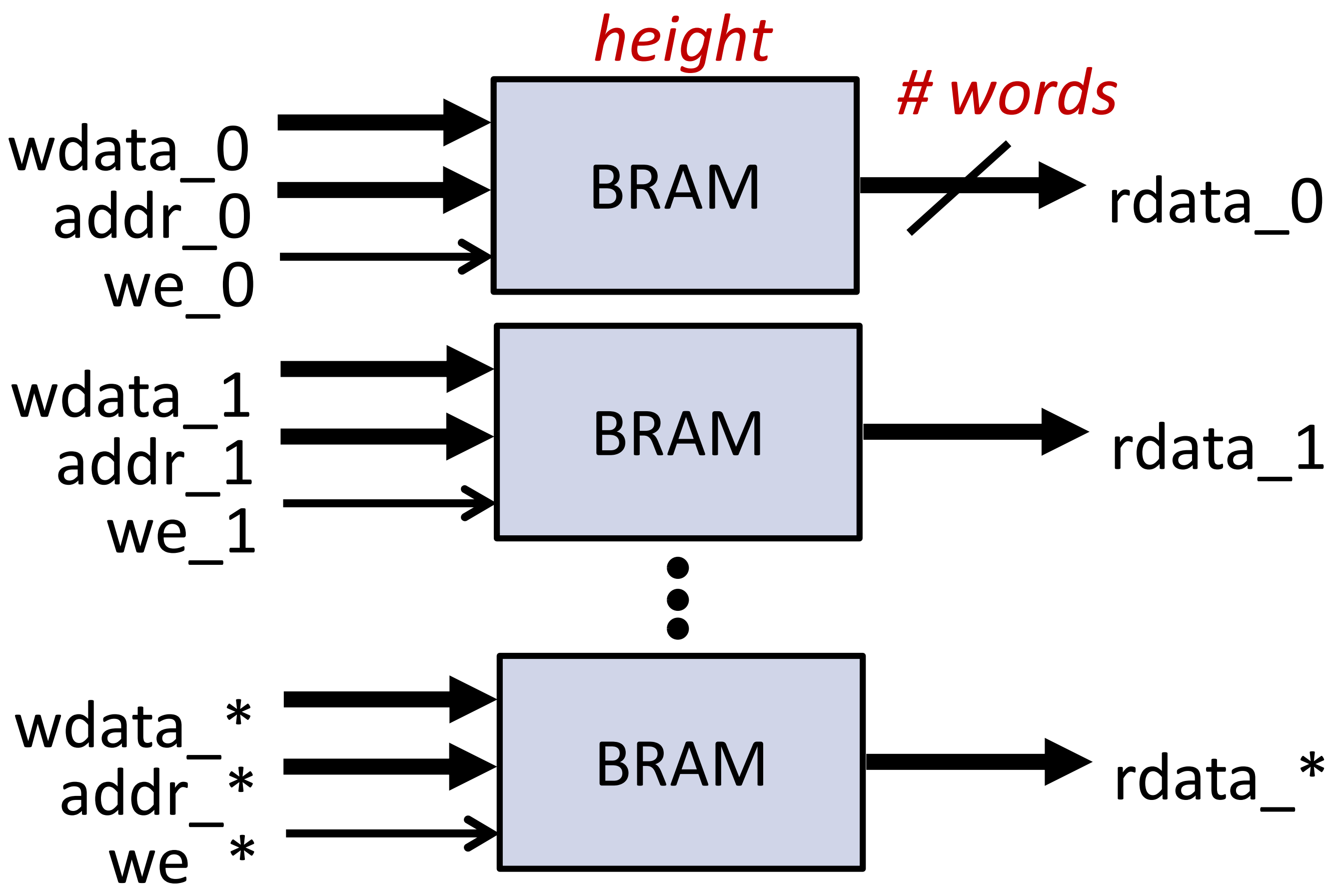
- Cache is useful when locality is not predictable ahead of time

Customized the cache if you use one!!!

Banking

not the same as DRAM chip banks

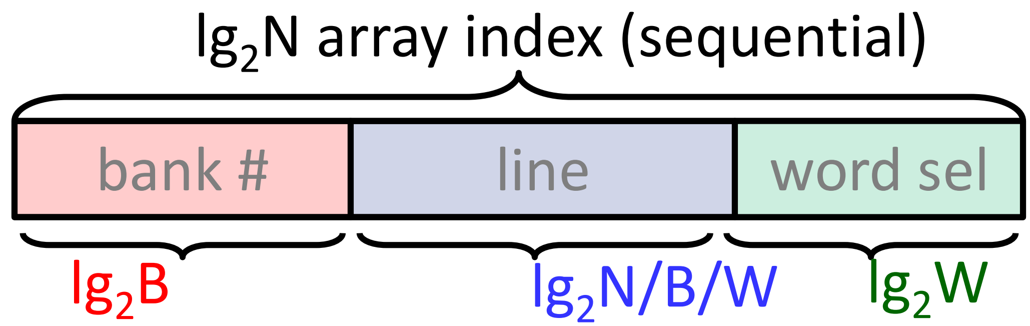
- Partition storage onto multiple structures
- More BW for parallel, non-conflicting accesses



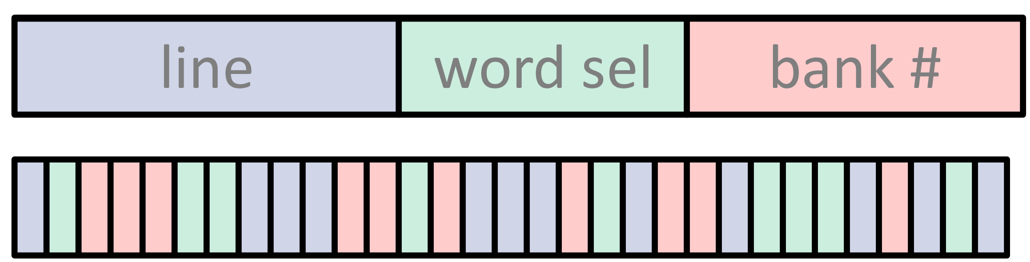
applicable to SRAM, DRAM or disk "storage"

Control over Data Layout

- An array of N words; index is lg_2N bits



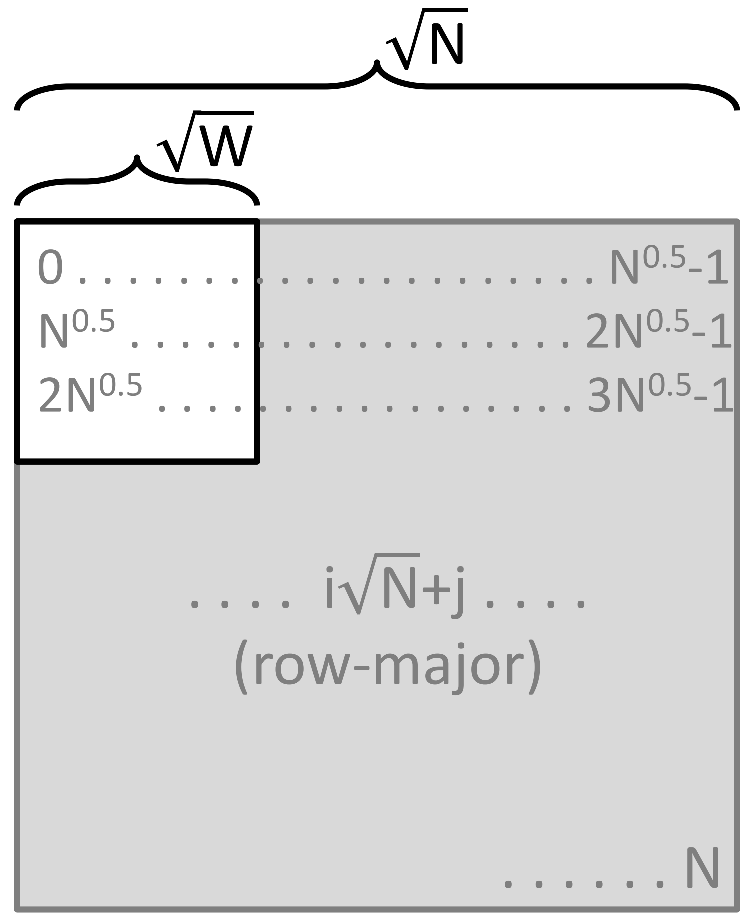
- N-word total storage
  - divided into B banks; bank number is lg_2B bits
  - each bank is W-word wide; word-select is lg_2W bits
  - line index within bank is $lg_2(N/B/W)$ bits
- Assign bank #, word select and index to maximize
  - spatial locality in word select
  - “entropy” in bank #

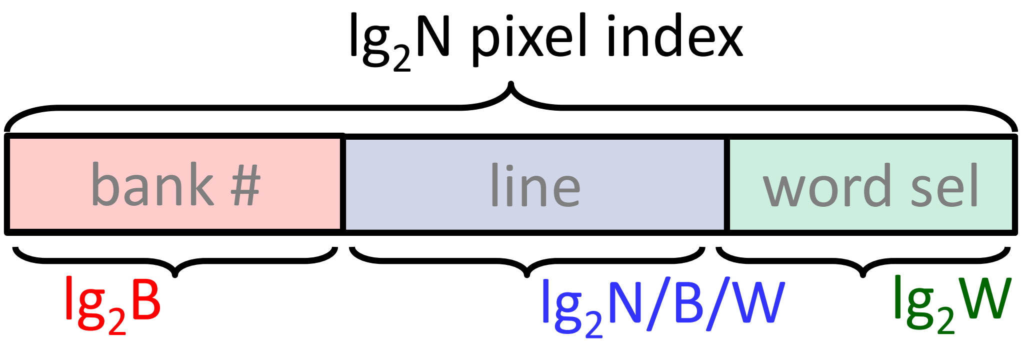


In general interleaved & reordered

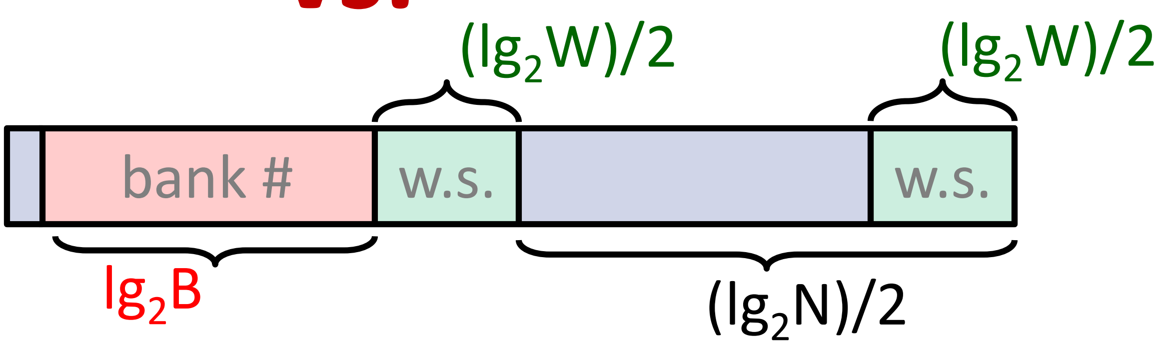
Example: Image Frame

- N pixels in $\sqrt{N}$-by-$\sqrt{N}$ frame
- Spatial locality in $\sqrt{W}$-by-$\sqrt{W}$ tiles
- Parallelism across same-column tiles





vs.



Can you tell the compiler (through C) this is what you want?

A Small Concrete Example: N=16, W=4

pixel idx = $a_3a_2a_1a_0$

col idx = a_1a_0

row idx = a_3a_2

tile idx = a_3a_1

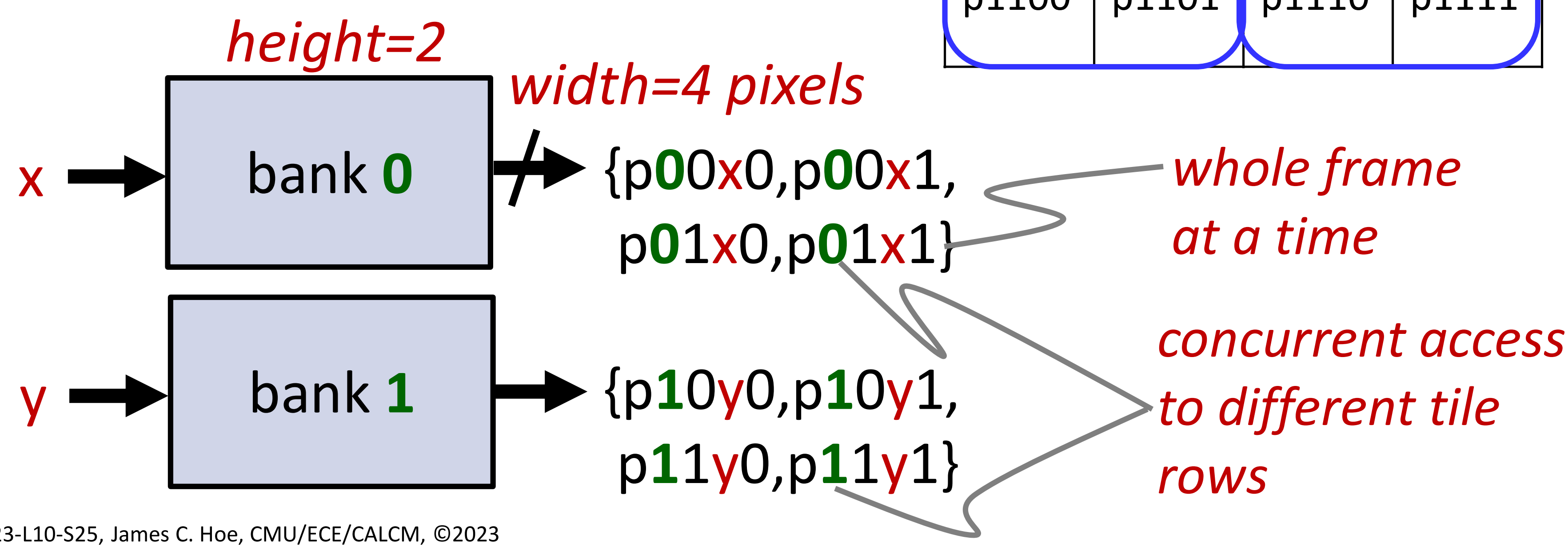
---

word sel = a_2a_0

bank offset = a_1

bank # = a_3

T00 p0000	p0001	T01 p0010	p0011
p0100	p0101	p0110	p0111
T10 p1000	p1001	T11 p1010	p1011
p1100	p1101	p1110	p1111



Blocked CNN Kernel on Local Memory

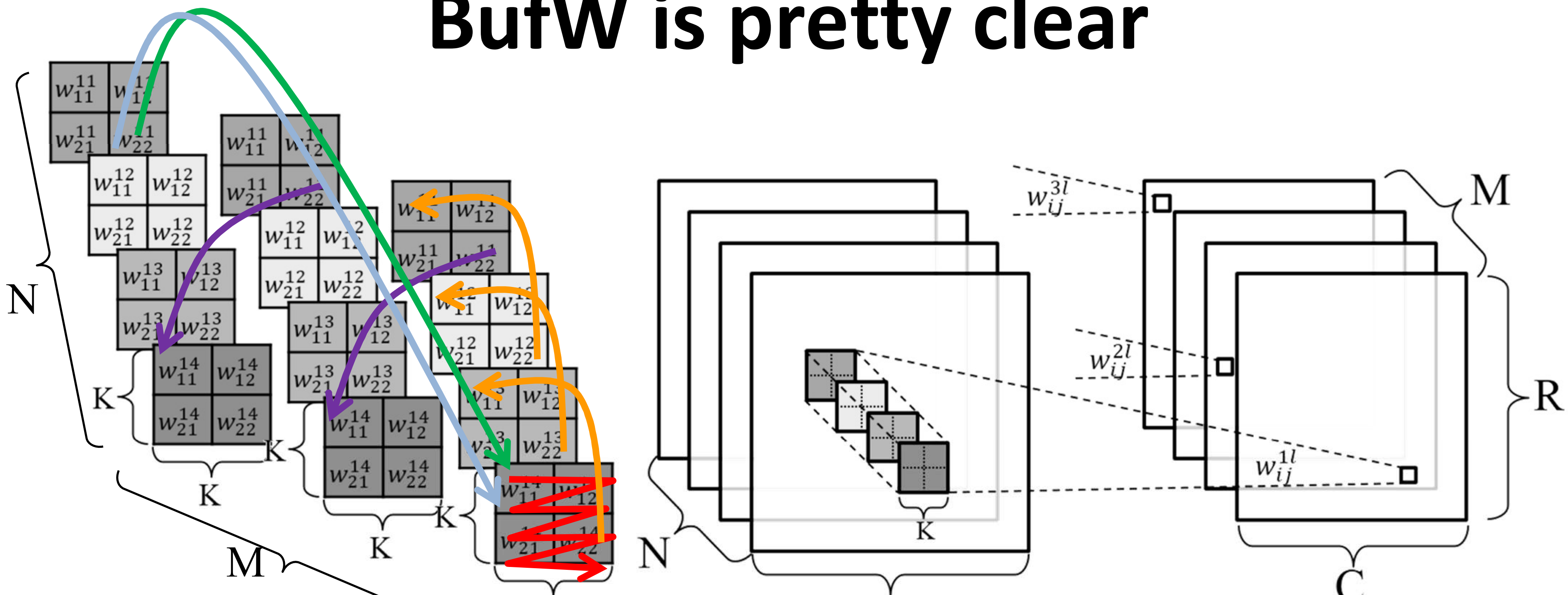
```
local cnndata_t BufI[_Tn][_Tr*_S_wts+_K_wts-1][_Tc*_S_wts+_K_wts-1];
local cnndata_t BufO[_Tm][_Tr][_Tc];
local cnndata_t BufW[_Tm][_Tn][_K_wts][_K_wts];

for(row_b=0;row_b<_Tr;row_b++){
  for(col_b=0;col_b<_Tc;col_b++){
    for(to_b=0;to_b<_Tm;to_b++){
      for(ti_b=0;ti_b<_Tn;ti_b++){
        for(i=0;i<_K_wts;i++){
          for(j=0;j<_K_wts;j++){
            BufO[to_b][row_b][col_b]+=
              BufW[to_b][ti_b][i][j]*
              BufI[ti_b][_S_wts*row_b+i]
                        [_S_wts*col_b+j];
}}}}}}
```

- Assuming sequential execution, what is the access sequence on **BufW** and **BufI**?

ignore **BufO** for now

BufW is pretty clear

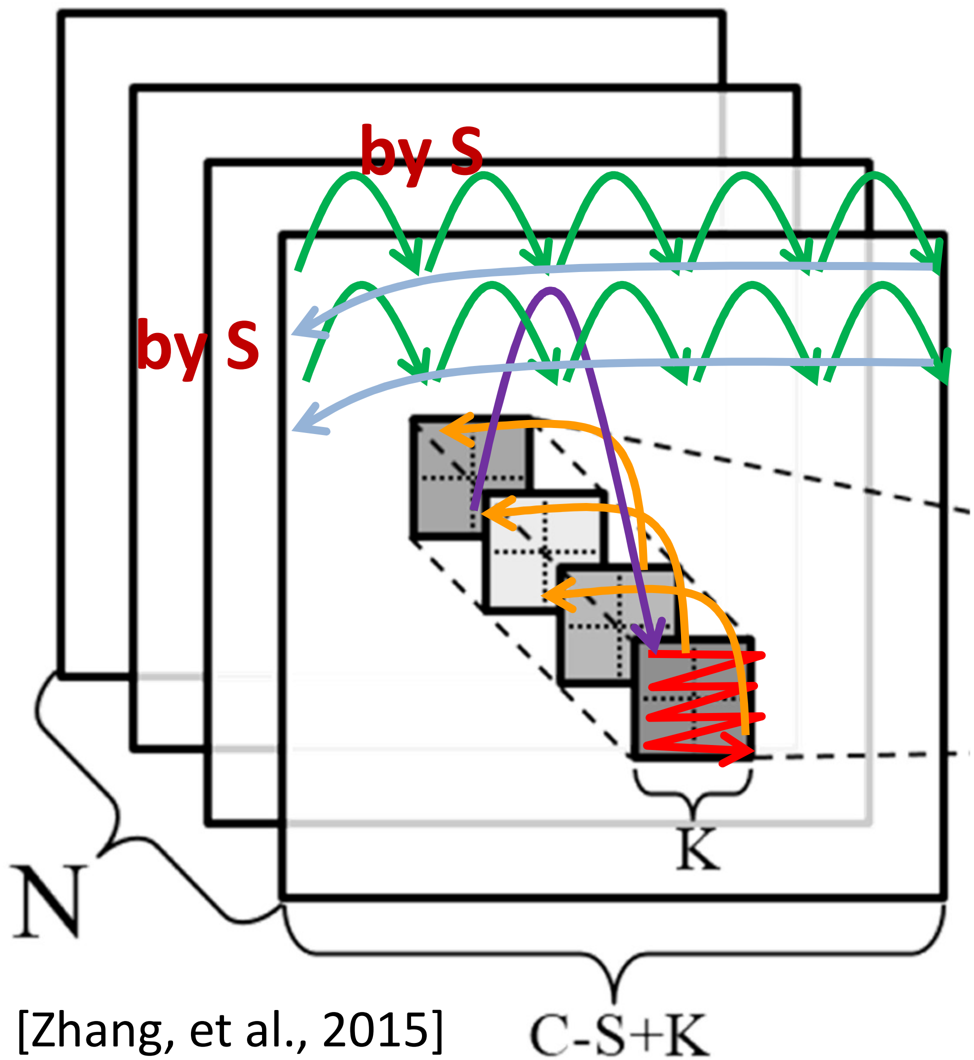


[Zhang, et al., 2015]

```
for(row_b=0;row_b<_Tr;row_b++){
 for(col_b=0;col_b<_Tc;col_b++){
  for(to_b=0;to_b<_Tm;to_b++){
   for(ti_b=0;ti_b<_Tn;ti_b++){
    for(i=0;i<_K_wts;i++){
     for(j=0;j<_K_wts;j++){
      BufO[to_b][row_b][col_b]+=
        BufW[to_b][ti_b][i][j]*
        BufI[ti_b][_S_wts*row_b+i]
                  [_S_wts*col_b+j];
}}}}}}
```

BufI more challenging

```
for(row_b=0;row_b<_Tr;row_b++){
 for(col_b=0;col_b<_Tc;col_b++){
  for(to_b=0;to_b<_Tm;to_b++){
   for(ti_b=0;ti_b<_Tn;ti_b++){
    for(i=0;i<_K_wts;i++){
     for(j=0;j<_K_wts;j++){
      BufO[to_b][row_b][col_b]+=
       BufW[to_b][ti_b][i][j]*
       BufI[ti_b][_S_wts*row_b+i]
                 [_S_wts*col_b+j];
}}}}}}
```



[Zhang, et al., 2015]

What should happen?

```
for(row_b=0;row_b<_Tr;row_b++){
  for(col_b=0;col_b<_Tc;col_b++){
    for(to_b=0;to_b<_Tm;to_b++){
      for(ti_b=0;ti_b<_Tn;ti_b++){
        #pragma HLS_PIPELINE
        for(i=0;i<_K_wts;i++){
          for(j=0;j<_K_wts;j++){
            BufO[to_b][row_b][col_b]+=
              BufW[to_b][ti_b][i][j]*
              BufI[ti_b][_S_wts*row_b+i]
                        [_S_wts*col_b+j];
}}}}}}}
```

initiated sequentially into pipelined body

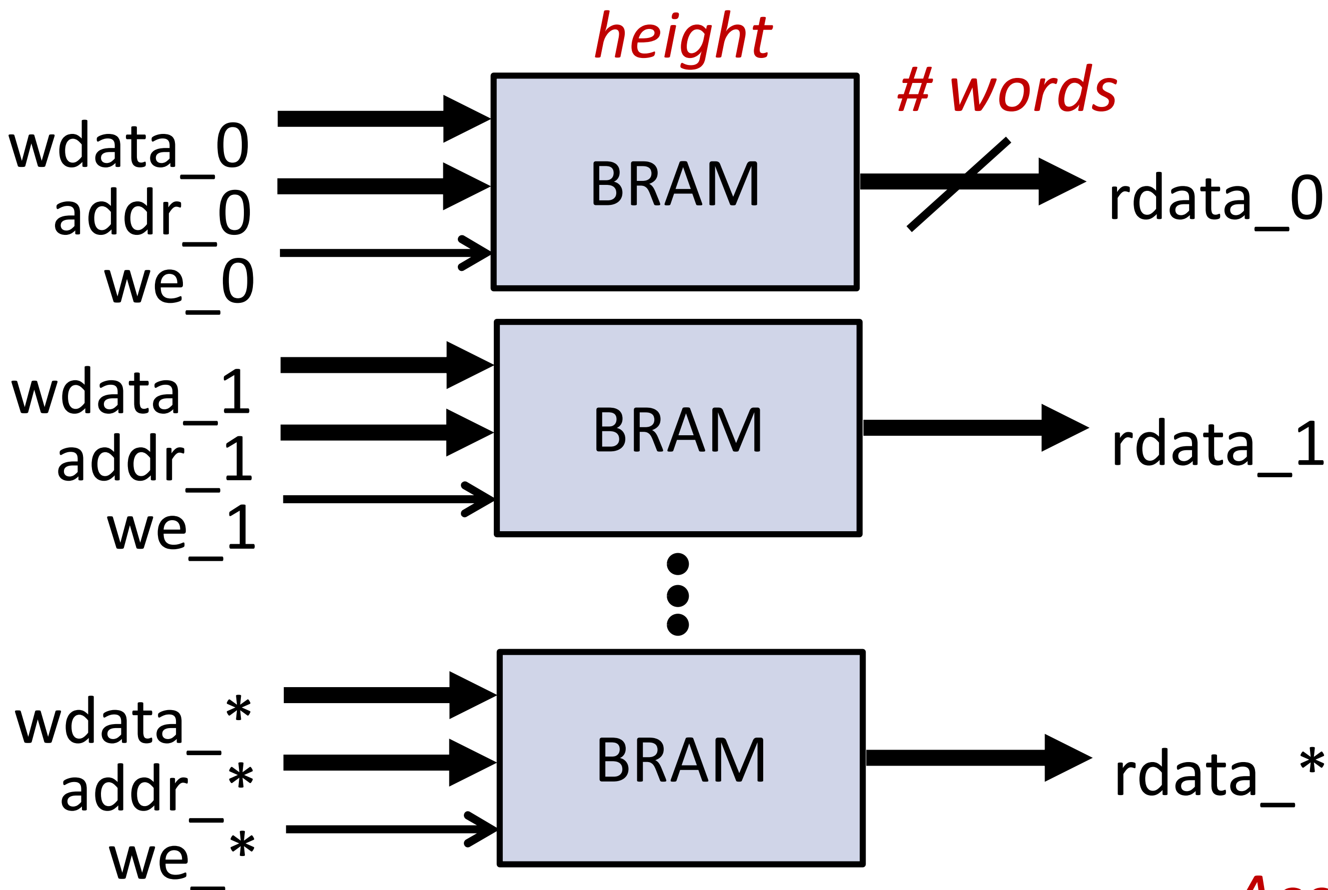
i and j loops fully unrolled; and pipelined

How many '*'/cycle possible?

What to read concurrently in 1 cycle?

How to Layout BufW

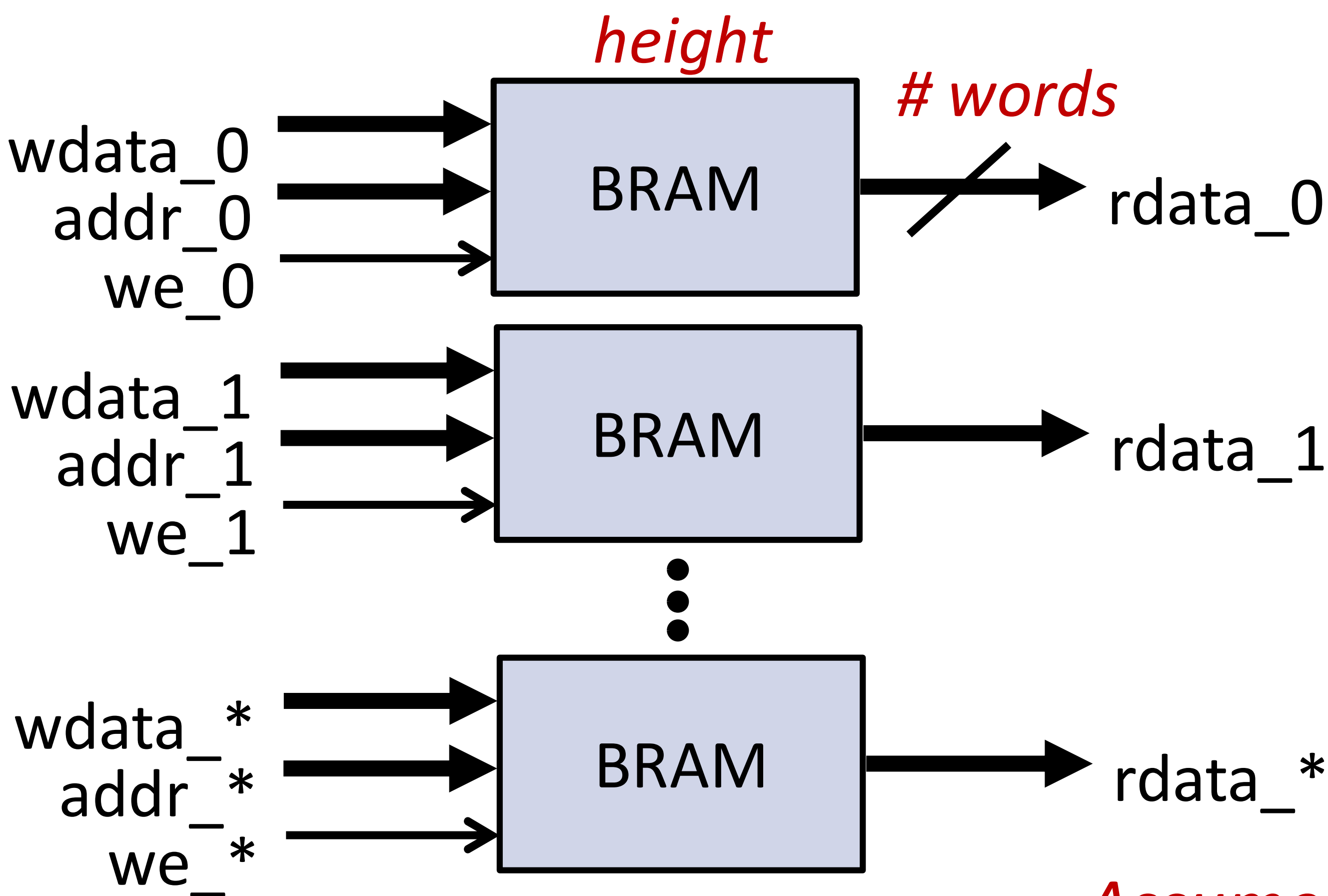
- # of bank, width of bank *(height is derived)*
- bank number, line index, word select



Assume K=4

How to Layout BufI

- # of bank, width of bank *(height is derived)*
- bank number, line index, word select



Assume S=2, K=4

What should happen?

```
for(row_b=0;row_b<_Tr;row_b++){
  for(col_b=0;col_b<_Tc;col_b++){
    for(to_b=0;to_b<_Tm;to_b++){
      #pragma HLS_PIPELINE
      for(ti_b=0;ti_b<_Tn;ti_b++){
        for(i=0;i<_K_wts;i++){
          for(j=0;j<_K_wts;j++){
            BufO[to_b][row_b][col_b]+=
              BufW[to_b][ti_b][i][j]*
              BufI[ti_b][_S_wts*row_b+i]
                    [_S_wts*col_b+j];
}}}}}}}
```

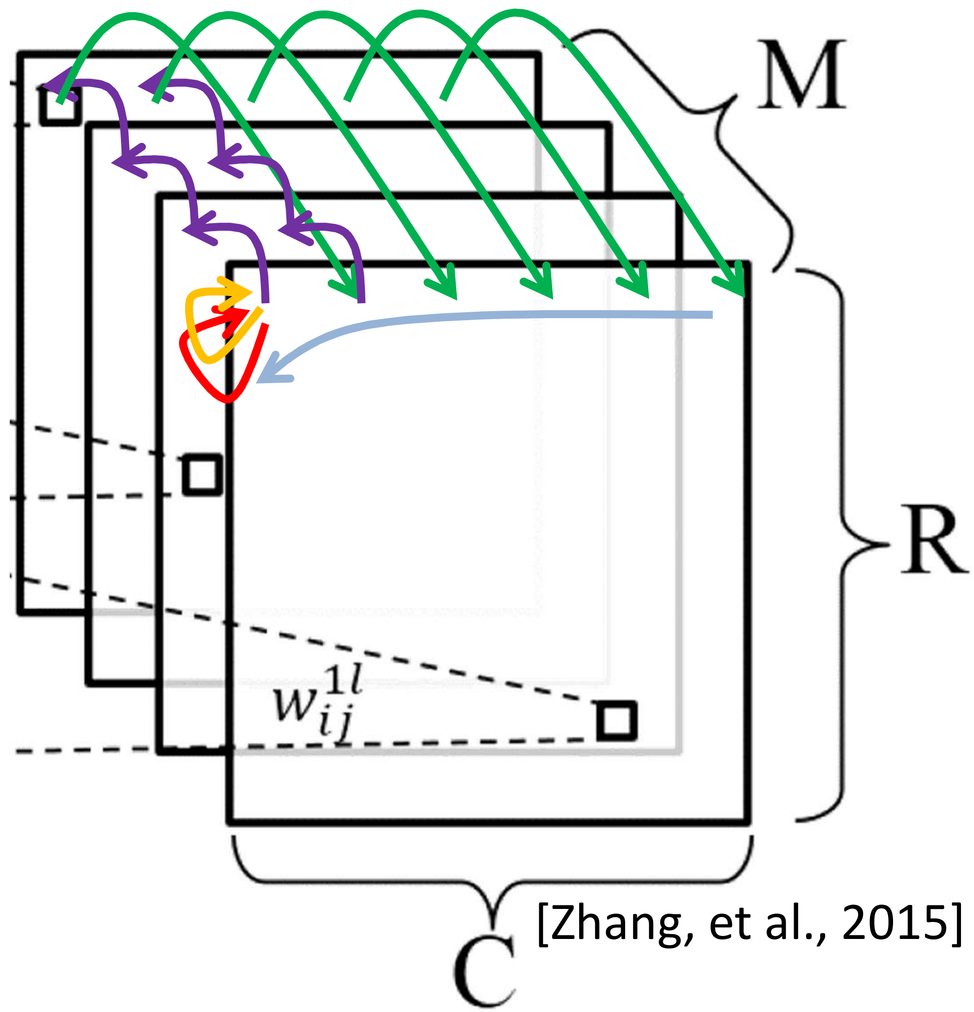
initiated sequentially into pipelined body

ti, i and j loops unrolled and pipelined

How many ‘*’/cycle possible?

What about BufO?

```
for(row_b=0;row_b<_Tr;row_b++){
 for(col_b=0;col_b<_Tc;col_b++){
  for(to_b=0;to_b<_Tm;to_b++){
   for(ti_b=0;ti_b<_Tn;ti_b++){
    for(i=0;i<_K_wts;i++){
     for(j=0;j<_K_wts;j++){
       BufO[to_b][row_b][col_b]+=
         BufW[to_b][ti_b][i][j]*
         BufI[ti_b][_S_wts*row_b+i]
                   [_S_wts*col_b+j];
}}}}}}
```



[Zhang, et al., 2015]

Some Hints on Lab 2

- Optimize tiled kernel throughput out-of-context
  – maximize arithmetic op/cyc
  – buffers need to match arithmetic concurrency
  – stay within Ultra96 resources *(don't run out of LUT, DSP, or BRAM)*

Don't worry about full layer size and data reuse

- Degrees of freedom
  – tile size
  – loop ordering *(affects access pattern)*
  – loopnest level to pipeline *(higher perf but also cost)*

Buffer design and data layout tied to above choices

Start from the Shallow End

```
for(row_b=0;row_b<_Tr;row_b++){
  for(col_b=0;col_b<_Tc;col_b++){
    for(to_b=0;to_b<_Tm;to_b++){
      for(ti_b=0;ti_b<_Tn;ti_b++){
        for(i=0;i<_K_wts;i++){
          for(j=0;j<_K_wts;j++){
#pragma HLS_PIPELINE
            BufO[to_b][row_b][col_b]+=
              BufW[to_b][ti_b][i][j]*
              BufI[ti_b][_S_wts*row_b+i]
                        [_S_wts*col_b+j];
}}}}}}}
```

Play with the Design Space

```
for(i=0;i<_K_wts;i++){
  for(j=0;j<_K_wts;j++){
    for(ti_b=0;ti_b<_Tn;ti_b++){
      for(to_b=0;to_b<_Tm;to_b++){
        for(row_b=0;row_b<_Tr;row_b++){
          for(col_b=0;col_b<_Tc;col_b++){
            #pragma HLS_PIPELINE
            BufO[to_b][row_b][col_b]+=
              BufW[to_b][ti_b][i][j]*
              BufI[ti_b][_S_wts*row_b+i]
                   [_S_wts*col_b+j];
}}}}}}
```

Parting Thoughts

- Memory architecture and memory performance important to computing
- Spatial FPGA computing needs adjusted intuition
  - wider, more concurrent access
  - slow clock ticks
  - amenable to extreme specialization
- Look forward to
  - more SRAM, faster DRAM
  - more data movement BW within fabric
  - cache-coherence, better integration
  - hardwired, native memory architecture?