High-Performance FPGA-Based QR Decomposition

Huy Nguyen, James Haupt, Michael Eskowitz, Birol Bekirov, Jonathan Scalera, Thomas Anderson, Michael Vai, and Kenneth Teitelbaum

HPEC 2005

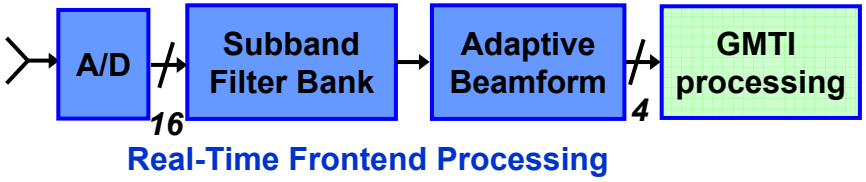
September 20, 2005

*** This work is sponsored by the Department of the Air Force under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.**

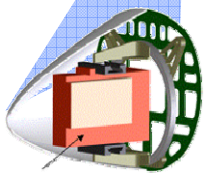
Outline

- QR Computation
- C-Language Implementation on FPGA
- Pipelined Linear Array on FPGA
- Summary

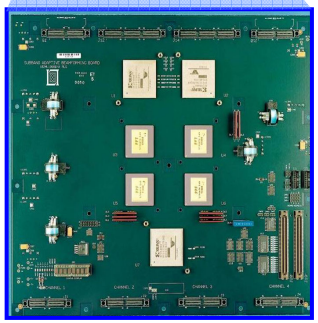
Adaptive Beamforming in Radar Processing



Wideband Active Radar

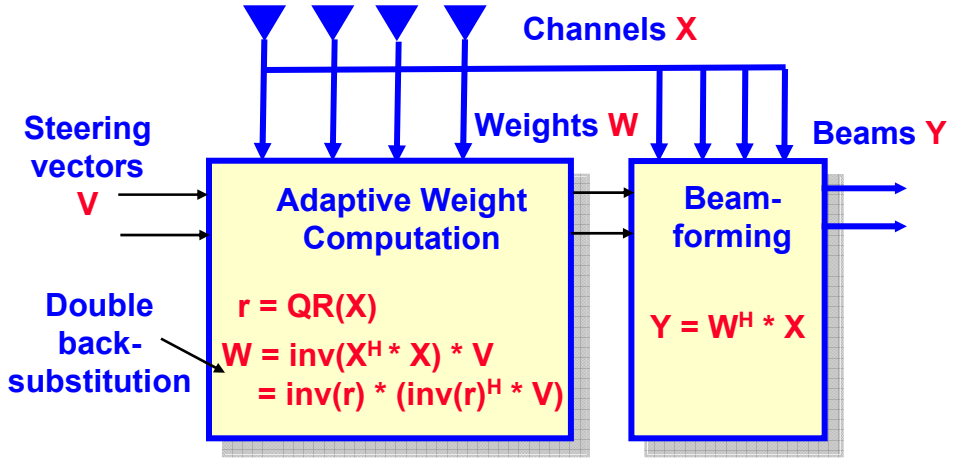


Airborne Electronically Steered Array

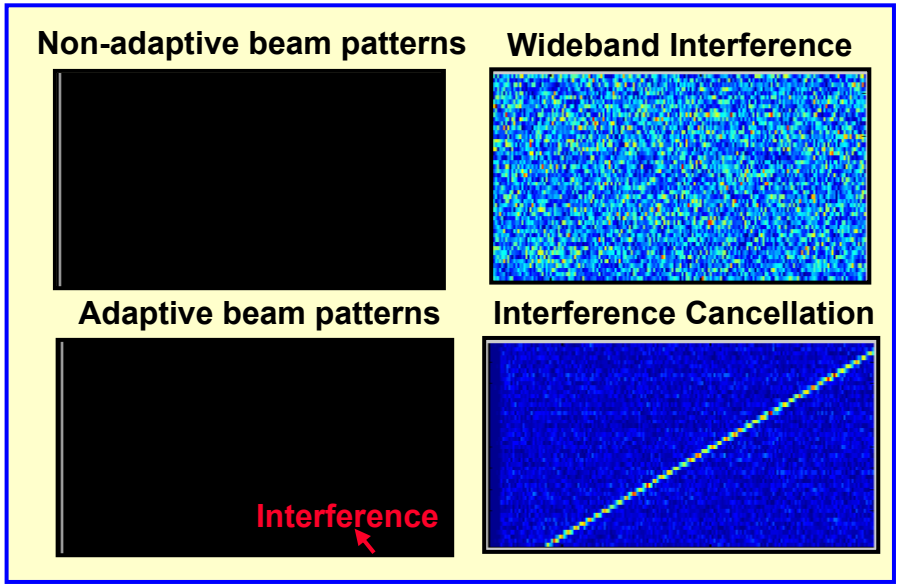


Processor
Subband Filtering and
Adaptive Beamforming

Backend Processing & Recording

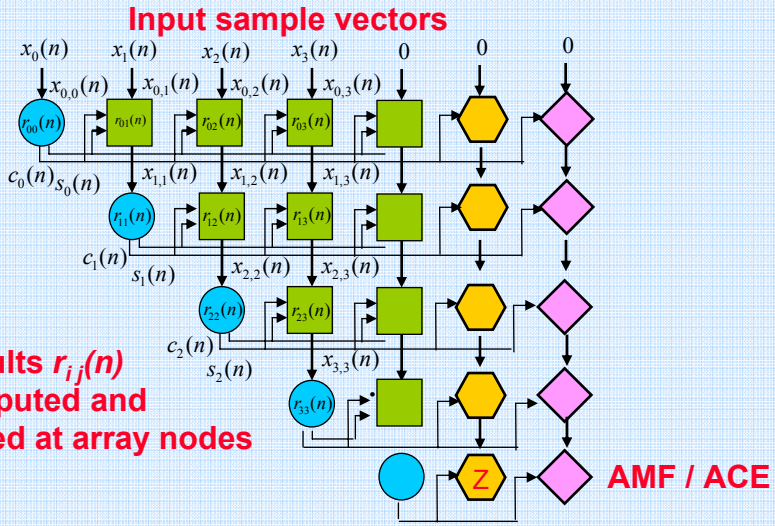


Adaptive Beamforming



Adaptive Weight Computation

McWhirter Array



$$r_{ij}(0) = \begin{cases} \sqrt{\alpha} & i = j \\ 0 & i \neq j \end{cases}$$

$$\begin{cases} r_{ii}(n) = \sqrt{r_{ii}^2(n-1) + |x_{i,i}(n)|^2} \\ c_i(n) = \dfrac{r_{ii}(n-1)}{r_{ii}(n)} \\ s_i(n) = \dfrac{x_{i,i}(n)}{r_{i,i}(n)} \end{cases}$$

$$\begin{cases} r_{ij}(n) = c_i r(n-1) + s_i^* x_{i,j}(n) \\ x_{i+1,j}(n) = -s_i r_{ij}(n-1) + c_i x_{i,j}(n) \end{cases}$$

$$sv = s \quad when\ control = 1$$
$$x\_out = x\_in + s \bullet conj(sv)$$

$Z = x^H r^{-1} r^{-1H} v$

$$x\_out = x\_in + s \bullet conj(s)$$

$AMF = v^H r^{-1} r^{-1H} v$

$ACE = x^H r^{-1} r^{-1H} x$

Desirable Properties of McWhirter Array

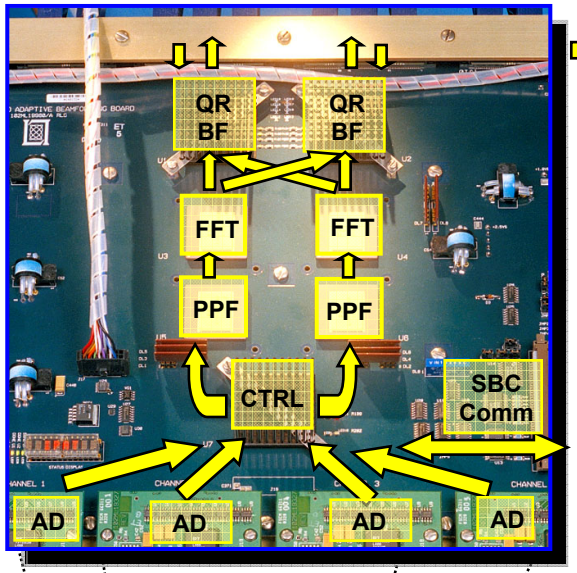
- Systolic, high-speed neighbor communication
- QR, Back-substitution, Beamforming can be performed using the same array
- Givens rotation in "voltage-domain" requires smaller word size than "power-domain"

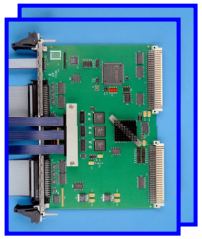
Challenges

- Entire array does not fit in FPGA
- Efficiency at most 50%
- QR and BackSolve need different dynamic ranges
- 1/sqrt in boundary nodes difficult to implement
- Op-count higher than Squared Givens and House-Holder methods

- AMF: Adaptive Matched Filter
- ACE: Adaptive Coherence Estimate

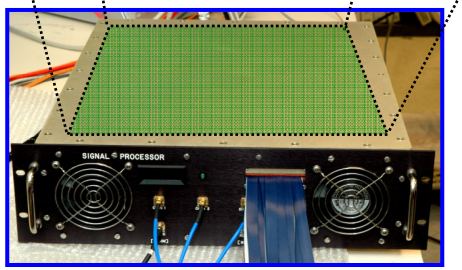
Computation Platform





MEMIO

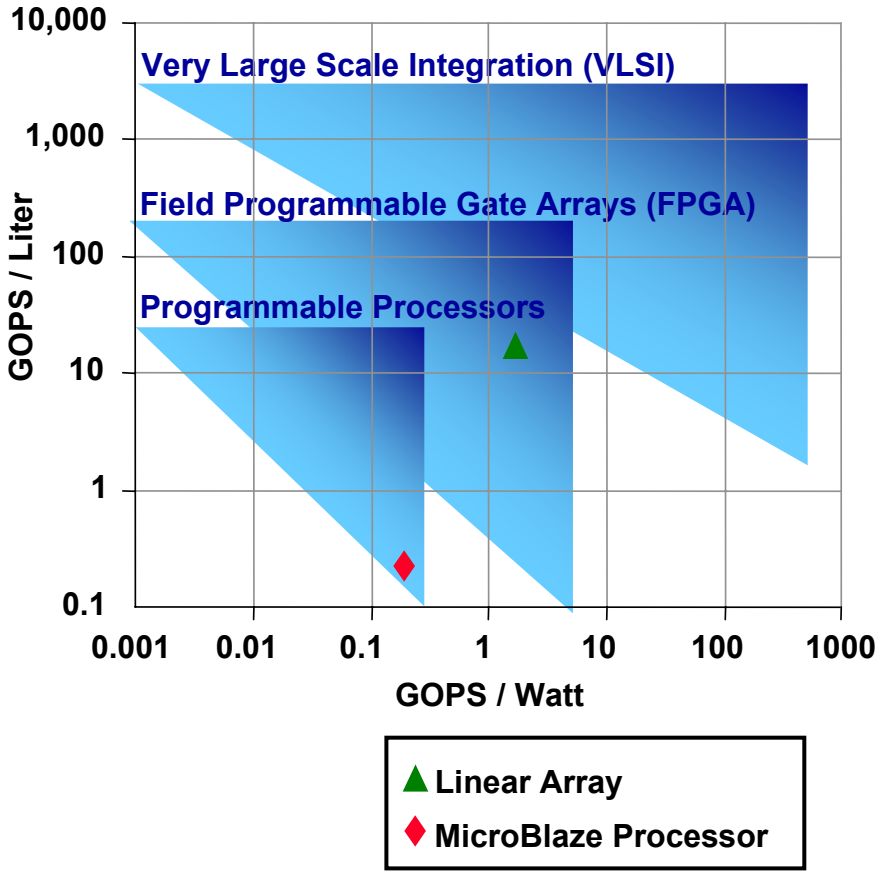
Single Board Computer



Processor Board
A/D Modules

Processing	200 Gops
Internal bw	15 GBps
Digital i/o	5 GBps dplx
SBC comm	50 MBps dplx
Scalable	x 4

- On-board computation eliminates I/O latency
- FPGAs enable high computational throughput and power efficiency

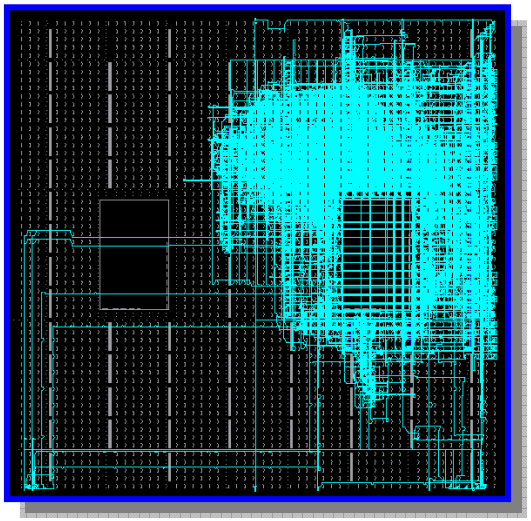


Outline

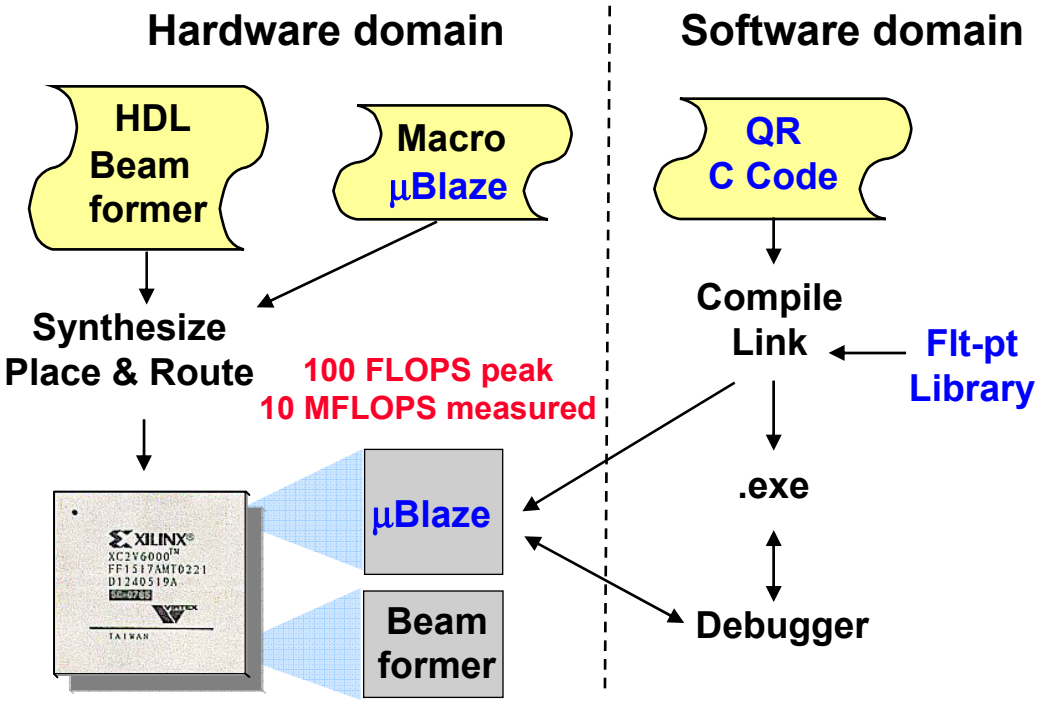
- QR Computation
- C-Language Implementation on FPGA
  - Soft-macro Embedded Processor inside FPGA
  - Measured Performance
- Pipelined Linear Array on FPGA
- Summary

Software Approach to QR Decomposition

- C program runs on Xilinx's 32-bit MicroBlaze processor macro core
- Software performs Givens-based QR and back substitution to compute the weights
- QR program code resident on FPGA (450 lines of C code)
- Maximum clock rate 100 MHz on Virtex-II
- Floating-point library and floating-point unit available
- Development time 2 – 3 months, flexible, suitable for rapid prototyping



One MicroBlaze takes about 10% of Virtex-II 8000



Measured Performance

- Optimization results in 100x speedup for the µBlaze
  - Inclusion of floating-point unit
  - 1/sqrt with Newton's method rather than library call
  - In-line code roll up
  - Integer-to-fltpt conversion and vice versa
  - Next power of 2
- Performance measurement
  - C code with compiler optimization ON
  - Execution time measured on µBlaze, G4, and Pentium 4
  - Effectively 10 to 20 clocks per "op"
  - Performance scales with clock rate
  - Multiple µBlaze processors needed to meet system time budget

Effective "ops" per sec

Time per QR

Processors to meet system requirements

Training 5 x chans	µBlaze 100 MHz	G4 733 MHz	Pentium 4 3.6 GHz
4 Chans (4 x 20)	759 µs, 5 MFLOPS, 3 units	106 µs, 38 MFLOPS, 1 unit	17 µs, 200 FLOPS, 1 unit
8 Chans (8 x 40)	3,833 µs, 7.5 MFLOPS, 12 units	602 µs, 48 MFLOPS, 3 units	90 µs, 300 FLOPS, 1 unit
16 Chans (16 x 80)	22,180 µs, 9.8 MFLOPS, NA	3,890 µs, 56 MFLOPS, 12 units	550 µs, 400 FLOPS, 2 units

- Verification was simple, most time spent on speed optimization
  - Excellent visibility into variable space inside the FPGA
  - Would be excellent wrapper for verifying / customizing high-performance IP cores

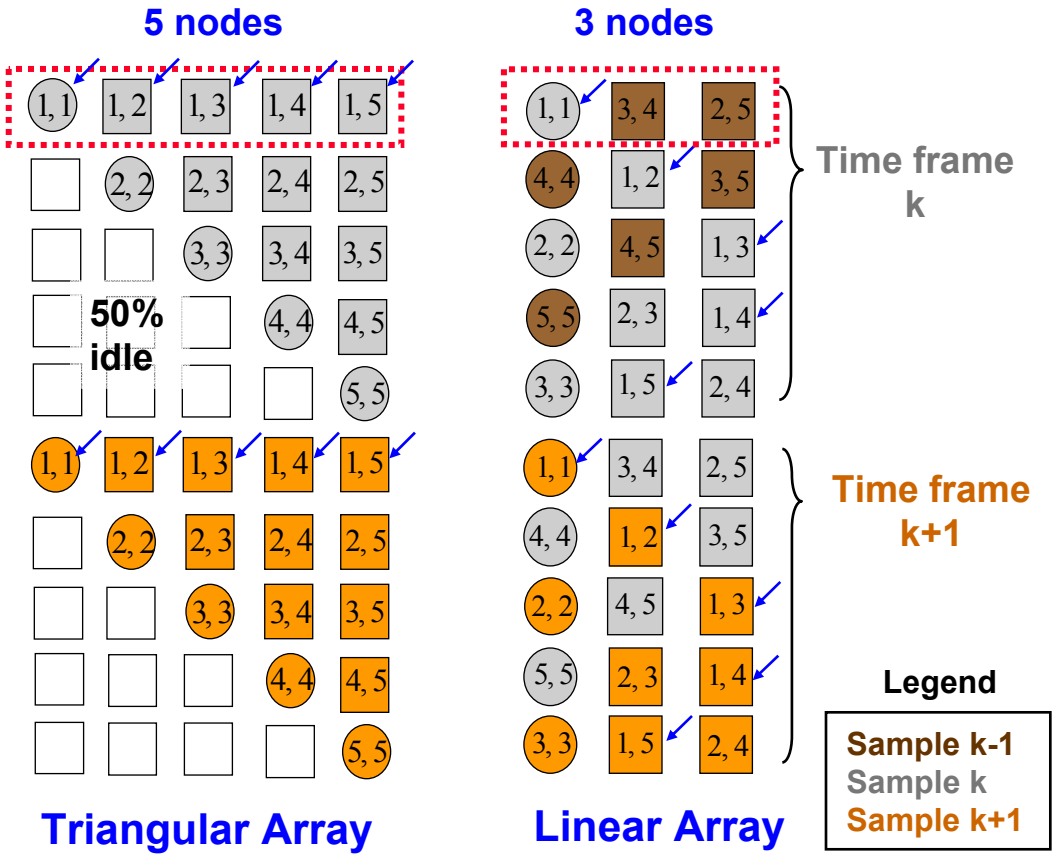
*QR program not hand-coded for G4 Altivec or Pentium MMX or MicroBlaze

Outline

- QR Computation
- C-Language Implementation on FPGA
- Pipelined Linear Array on FPGA
  - Linear array for Givens computation
  - Weight computation
  - Threshold Floating-point
  - 1/sqrt() using polynomial approximation
  - Pipelining
- Summary

Linear Array Structure

- Linear Array is obtained by folding McWhirter array [Walke et al]
- One row of schedule is implemented in the FPGA
- 100% efficiency, less hardware resources



Highlights

Walke et al

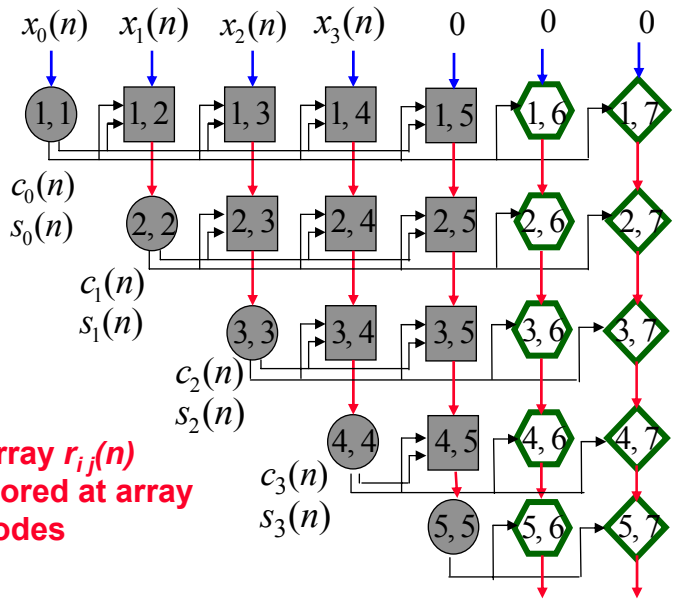
- Squared Givens
- Difficult op: Division
- Floating-point
- No weight computation

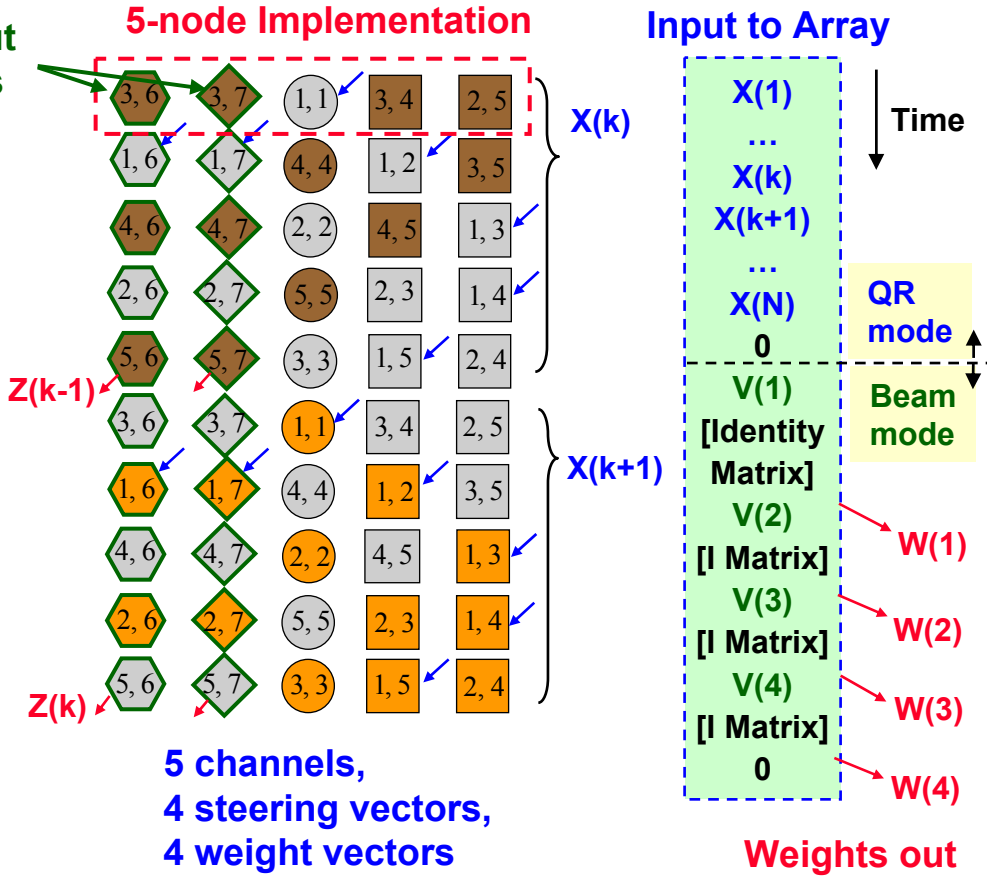
This Presentation

- Givens rotation
- Difficult op: 1/sqrt()
- "Threshold-fltpt"
- Weight computation

Weight Computation

- Weights, beam outputs, AMF, ACE can be computed by adding a *Beamforming mode* and two additional output nodes
- In *Beamforming mode*, array *r* is used in computation but not updated





$sv = s \quad when\ control = 1$
$x\_out = x\_in + s \bullet conj(sv)$

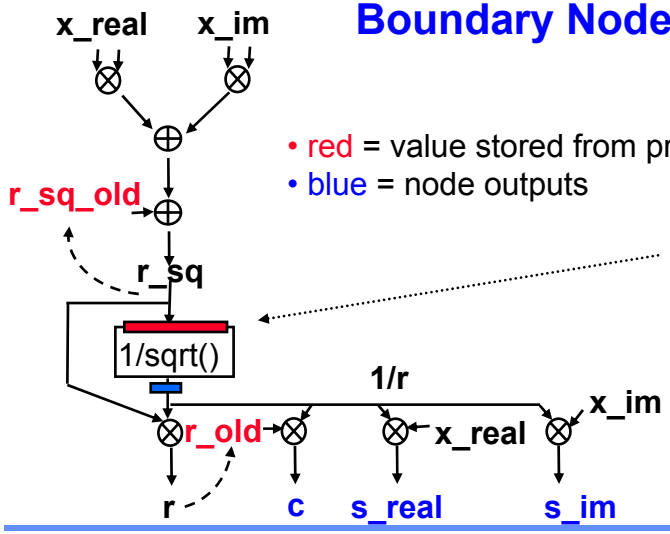
$x\_out = x\_in + s \bullet conj(s)$

Z = $x^H r^{-1} r^{-1H} v$
Weights, Beam = Z

AMF = $v^H r^{-1} r^{-1H} v$
ACE = $x^H r^{-1} r^{-1H} x$

Threshold Floating Point

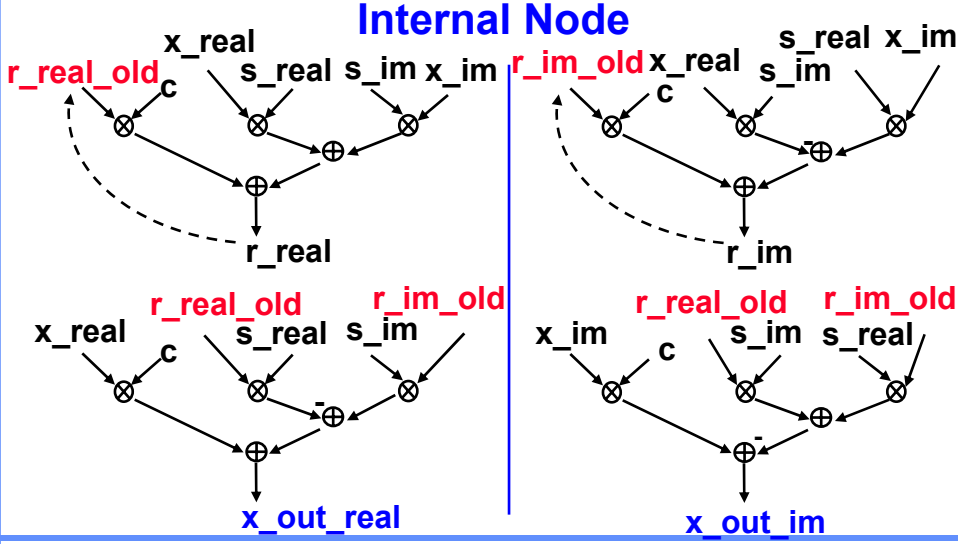
Boundary Node



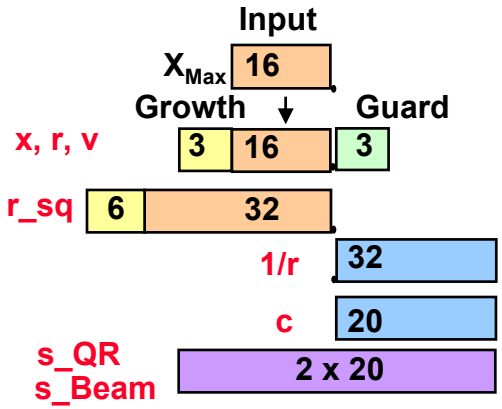
- red = value stored from previous QR iteration
- blue = node outputs

exponent normalization and realignment

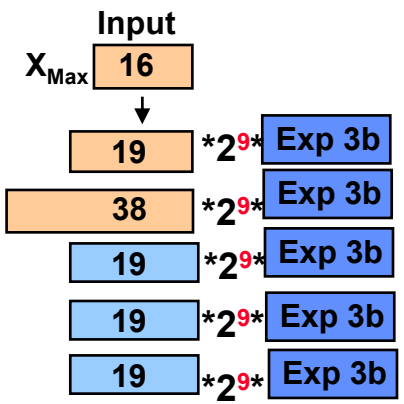
Internal Node



Fixed-pt



Threshold fltpt

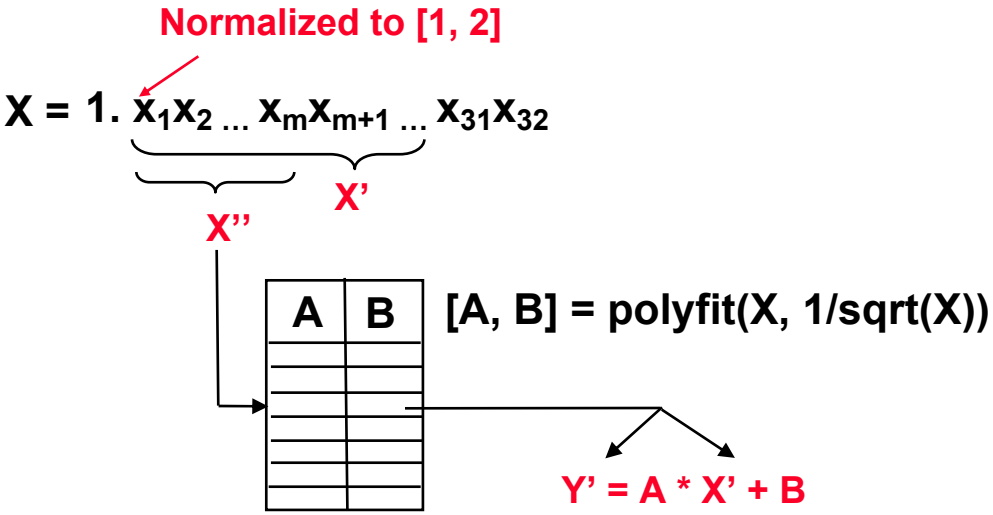


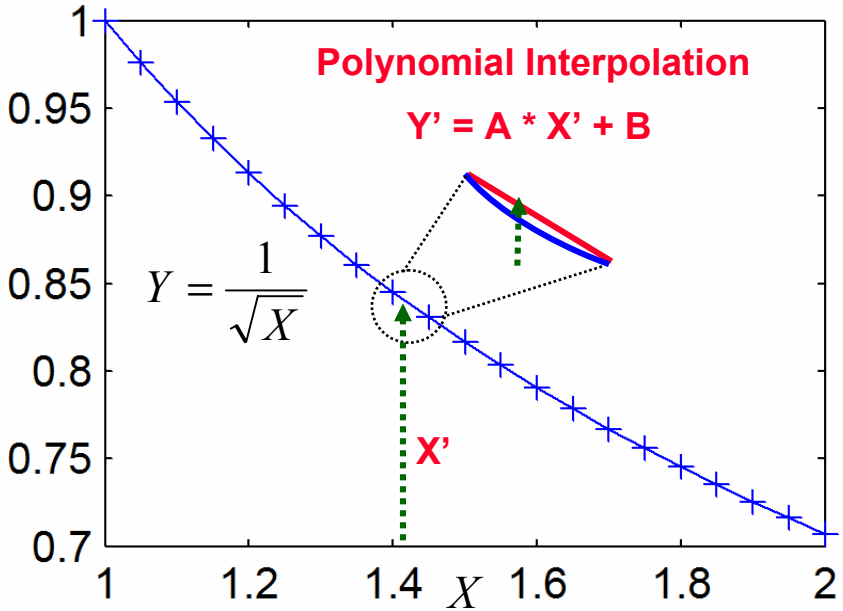
- Dual-use variable 's' has different dynamic ranges in QR mode and Beam mode.

- Threshold floating-point shifts the window by 9 bits when register content crosses threshold value(s).
- The number of thresholds is determined by shift size. Binary threshold (1/2 word size) results in normalizing logic simpler than regular floating-point implementation.
- Word size is a function of *dynamic range* and application *required precision*. Most of the time, it is smaller than fixed-pt implementation.

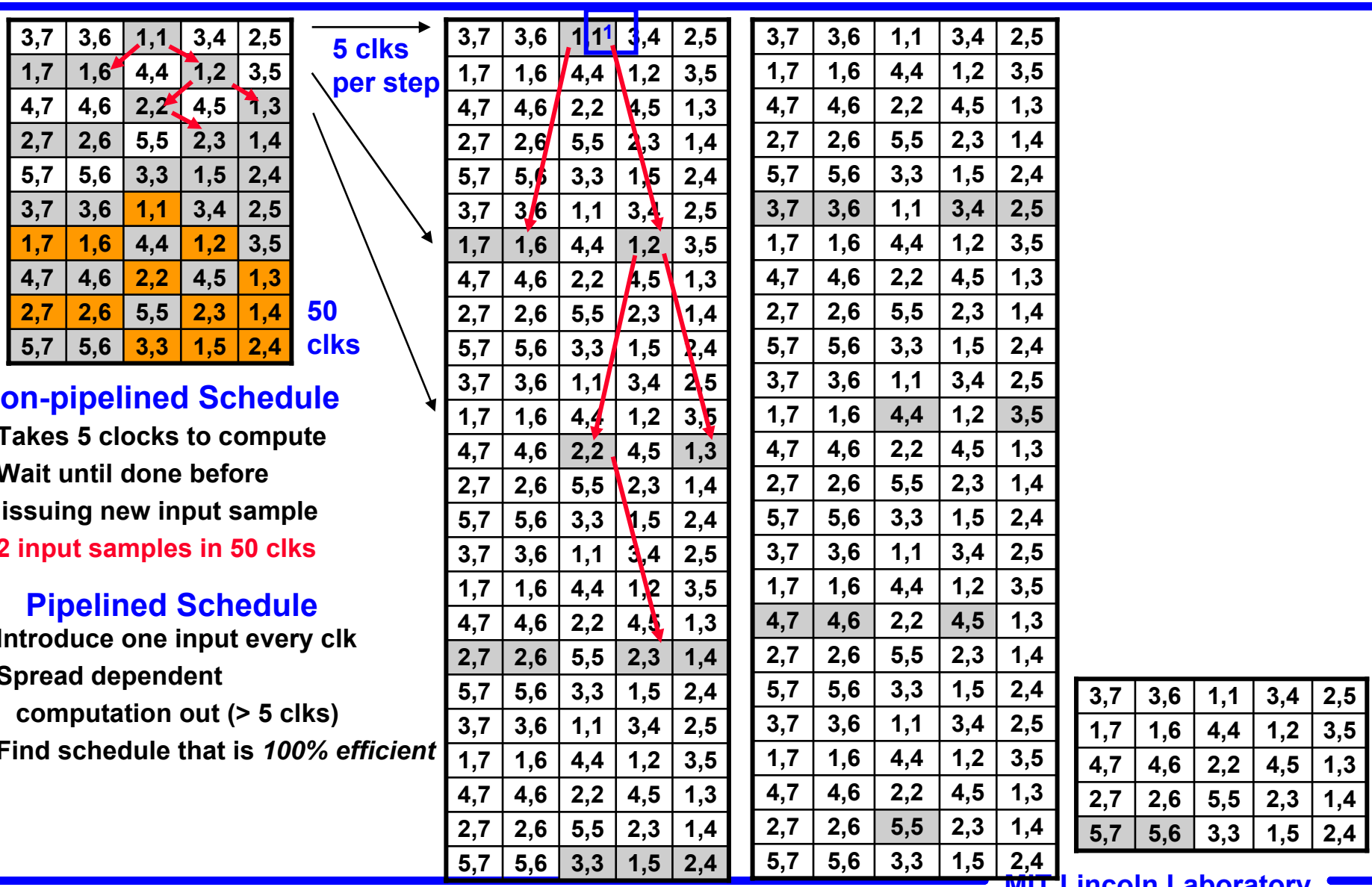
Polynomial Approximation to 1/sqrt()

- First-order approximation works well for 1/sqrt in [1, 2]
- Match with double precision computation within +/- lsb/2
  - 16-bit output: need two 256 x 18 tables, one 18 x 18 multiplier, and one 18b adder
  - 24-bit output: need two 4k x 26 tables, one 26 x 26 multiplier, and one 26b adder
- Table sizes appropriate for FPGA implementation
- No iteration required





Pipelining Illustrated for 5 Levels



Non-pipelined Schedule

- Takes 5 clocks to compute
- Wait until done before issuing new input sample
- 2 input samples in 50 clks

Pipelined Schedule

- Introduce one input every clk
- Spread dependent computation out (> 5 clks)
- Find schedule that is *100% efficient*

Pipelining Illustrated for 5 Levels

3,7	3,6	1,1	3,4	2,5
1,7	1,6	4,4	1,2	3,5
4,7	4,6	2,2	4,5	1,3
2,7	2,6	5,5	2,3	1,4
5,7	5,6	3,3	1,5	2,4
3,7	3,6	1,1	3,4	2,5
1,7	1,6	4,4	1,2	3,5
4,7	4,6	2,2	4,5	1,3
2,7	2,6	5,5	2,3	1,4
5,7	5,6	3,3	1,5	2,4

5 clks per step

50 clks

Non-pipelined Schedule

- Takes 5 clocks to compute
- Wait until done before issuing new input sample
- 2 input samples in 50 clks

Pipelined Schedule

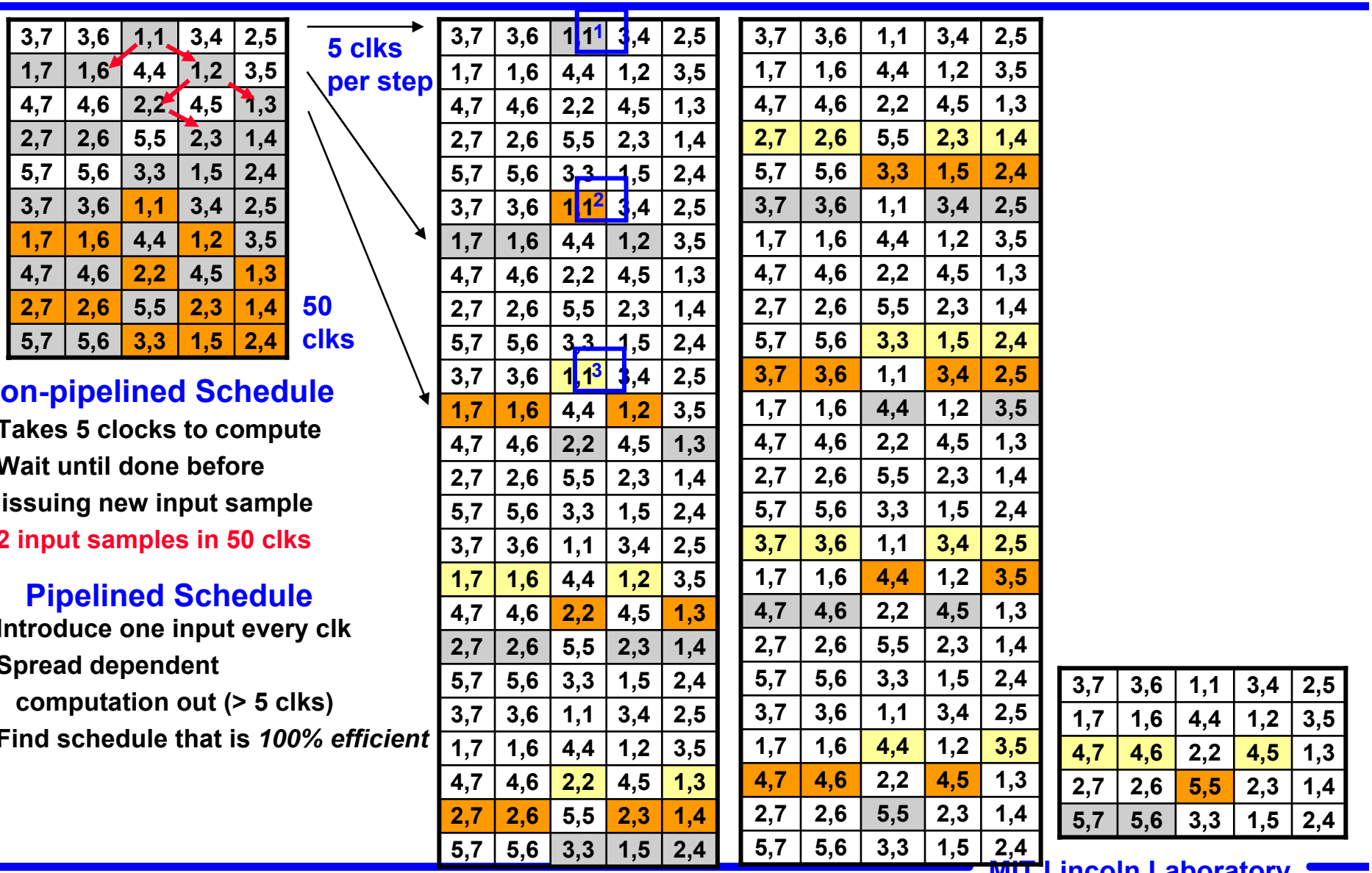
- Introduce one input every clk
- Spread dependent computation out (> 5 clks)

3,7	3,6	1,1[1]	3,4	2,5
1,7	1,6	4,4	1,2	3,5
4,7	4,6	2,2	4,5	1,3
2,7	2,6	5,5	2,3	1,4
5,7	5,6	3,3	1,5	2,4
3,7	3,6	1,1[2]	3,4	2,5
1,7	1,6	4,4	1,2	3,5
4,7	4,6	2,2	4,5	1,3
2,7	2,6	5,5	2,3	1,4
5,7	5,6	3,3	1,5	2,4
3,7	3,6	1,1	3,4	2,5
1,7	1,6	4,4	1,2	3,5
4,7	4,6	2,2	4,5	1,3
2,7	2,6	5,5	2,3	1,4
5,7	5,6	3,3	1,5	2,4
3,7	3,6	1,1	3,4	2,5
1,7	1,6	4,4	1,2	3,5
4,7	4,6	2,2	4,5	1,3
2,7	2,6	5,5	2,3	1,4
5,7	5,6	3,3	1,5	2,4
3,7	3,6	1,1	3,4	2,5
1,7	1,6	4,4	1,2	3,5
4,7	4,6	2,2	4,5	1,3
2,7	2,6	5,5	2,3	1,4
5,7	5,6	3,3	1,5	2,4

3,7	3,6	1,1	3,4	2,5
1,7	1,6	4,4	1,2	3,5
4,7	4,6	2,2	4,5	1,3
2,7	2,6	5,5	2,3	1,4
5,7	5,6	3,3	1,5	2,4
3,7	3,6	1,1	3,4	2,5
1,7	1,6	4,4	1,2	3,5
4,7	4,6	2,2	4,5	1,3
2,7	2,6	5,5	2,3	1,4
5,7	5,6	3,3	1,5	2,4
3,7	3,6	1,1	3,4	2,5
1,7	1,6	4,4	1,2	3,5
4,7	4,6	2,2	4,5	1,3
2,7	2,6	5,5	2,3	1,4
5,7	5,6	3,3	1,5	2,4
3,7	3,6	1,1	3,4	2,5
1,7	1,6	4,4	1,2	3,5
4,7	4,6	2,2	4,5	1,3
2,7	2,6	5,5	2,3	1,4
5,7	5,6	3,3	1,5	2,4
3,7	3,6	1,1	3,4	2,5
1,7	1,6	4,4	1,2	3,5
4,7	4,6	2,2	4,5	1,3
2,7	2,6	5,5	2,3	1,4
5,7	5,6	3,3	1,5	2,4

3,7	3,6	1,1	3,4	2,5
1,7	1,6	4,4	1,2	3,5
4,7	4,6	2,2	4,5	1,3
2,7	2,6	5,5	2,3	1,4
5,7	5,6	3,3	1,5	2,4

Pipelining Illustrated for 5 Levels



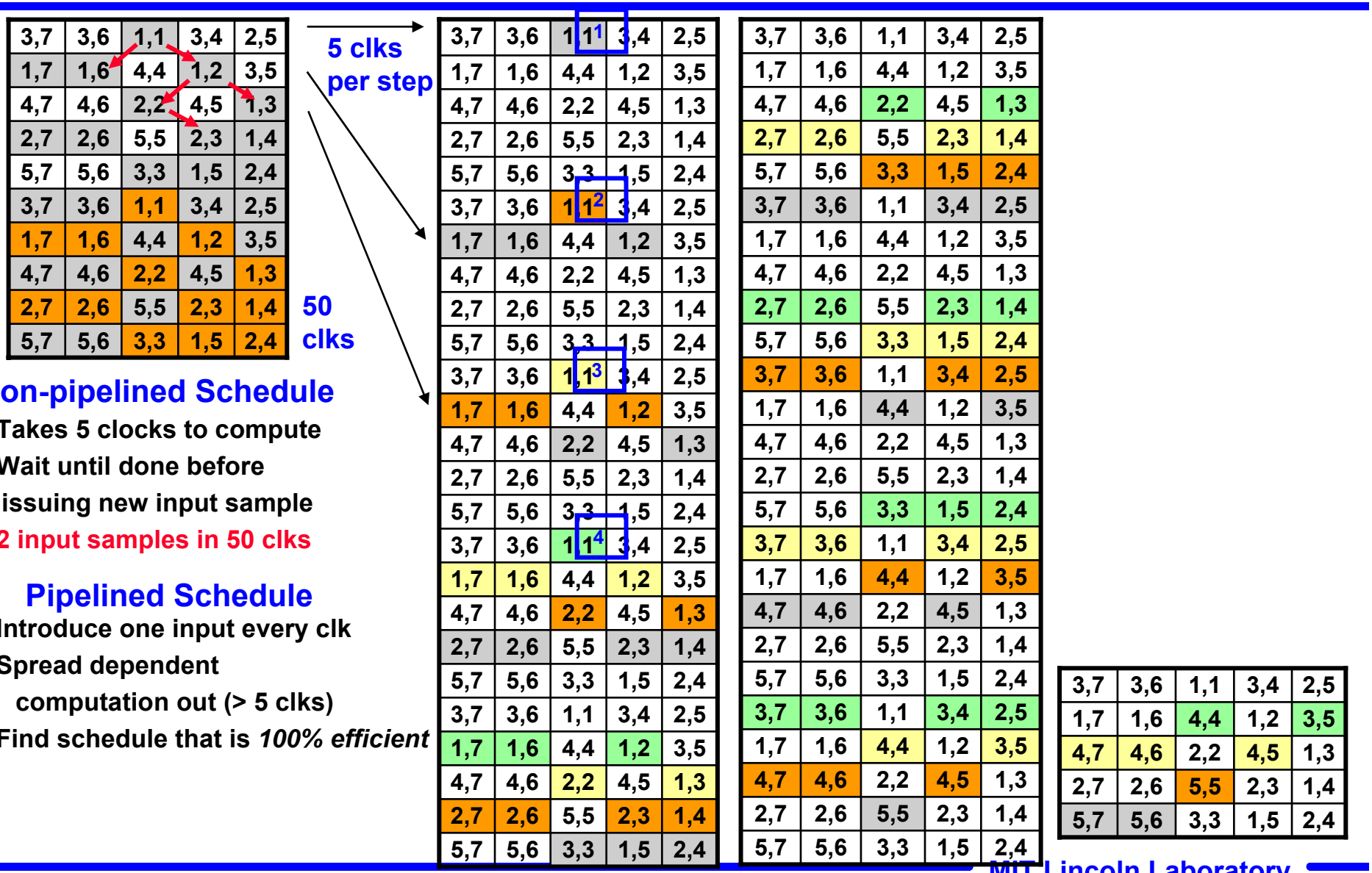
Non-pipelined Schedule

- Takes 5 clocks to compute
- Wait until done before issuing new input sample
- 2 input samples in 50 clks

Pipelined Schedule

- Introduce one input every clk
- Spread dependent computation out (> 5 clks)
- Find schedule that is *100% efficient*

Pipelining Illustrated for 5 Levels



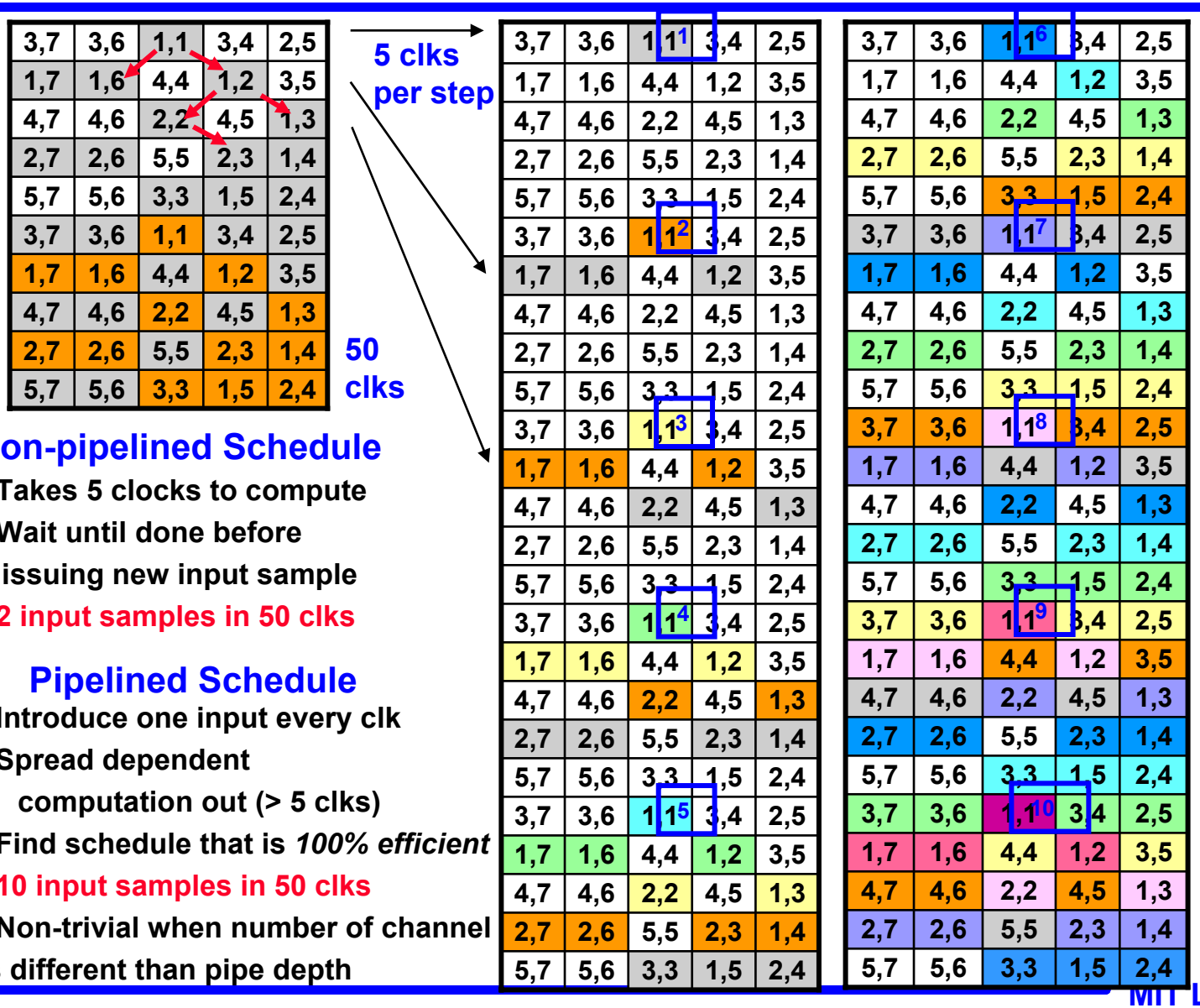
Non-pipelined Schedule

- Takes 5 clocks to compute
- Wait until done before issuing new input sample
- 2 input samples in 50 clks

Pipelined Schedule

- Introduce one input every clk
- Spread dependent computation out (> 5 clks)
- Find schedule that is *100% efficient*

Pipelining Illustrated for 5 Levels



Non-pipelined Schedule

- Takes 5 clocks to compute
- Wait until done before issuing new input sample
- 2 input samples in 50 clks

Pipelined Schedule

- Introduce one input every clk
- Spread dependent computation out (> 5 clks)
- Find schedule that is *100% efficient*
- 10 input samples in 50 clks
- Non-trivial when number of channel is different than pipe depth

Array 100% efficient after 10 * 5 = 50 clocks

Array Performance

Training 5 x channels Weight computation for 1 Beam		Virtex-II 8000 80 MHz clock (not fully optimized)	
		No pipeline	Pipeline by 16
4 Chans (4 x 20)	300 ops per sample 20 samples for QR 6 samples flushing per beam	1 Msps 300 MOPS 26 us refresh 25% Resources	[16 Msps estimated] [4.8 GOPS] [1.7 us] 35% Resources
8 Chans (8 x 40)	900 ops per sample 40 samples per QR 10 samples flushing per beam	0.5 Msps 500 MOPS 90 us refresh 35% Resources	[8 Msps] [8 GOPS] [6 us] 45% Resources
16 Chans (16 x 80)	3060 ops per sample 80 samples per QR 18 samples flushing per beam	0.3 Msps 900 MOPS 333 us refresh 60% Resources	[6 Msps] [14.4 GOPS] [21 us] 70% Resources

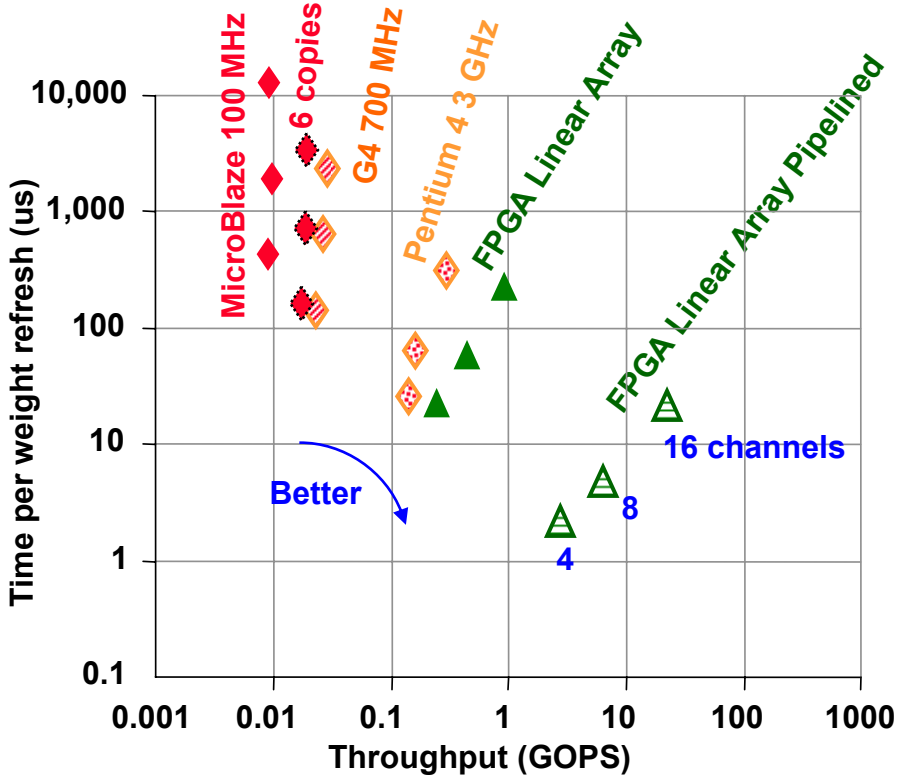
*Op counts are for 5-chan, 9-chan, and 17-chan arrays with 1 weight computation node
*Large multipliers are implemented with 17-bit built-in multipliers and logic slices

- Non-pipelined refresh rate is 26 us for 4 channels, and 333 us for 16 channels
- Non-pipelined 16-channel array is 1.7x faster than 3.6 GHz Pentium4, 12x 700 MHz G4
- Pipelined version expected to be 16 times faster

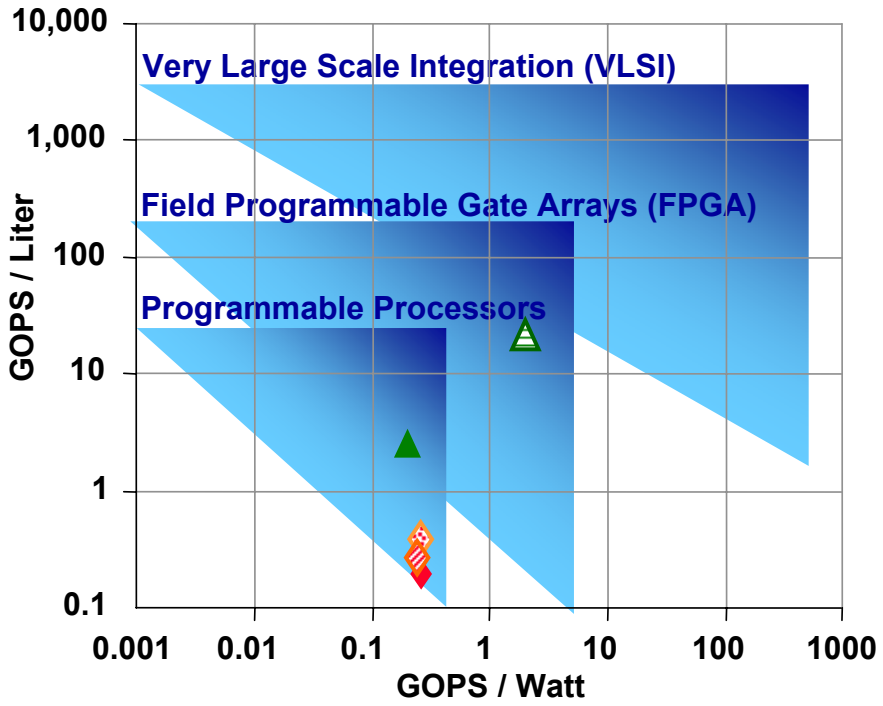
Outline

- QR Computation
- C-Language Implementation on FPGA
- Pipelined Givens Linear Array on FPGA
- Summary

QR & Weight Computation Summary



*6 copies of MicroBlaze can be used in parallel if application permits



*Assume 6 MicroBlaze processors in one FPGA V2-8000
*Assume minimum volume for packaging and cooling:
- FPGA, G4: 10 cm x 10 cm x 3 cm per chip ~ .3 L
- Pentium 4: 16 cm x 20 cm x 3 cm per chip ~ 1 L

- Linear array throughput increases with # channels due to available capacity on chip
- Linear array has best computational density (GOPS/L) and efficiency (GOPS/W)
- All processors are close in computational density and efficiency

*QR program not hand-coded for G4 Altivec or Pentium MMX or MicroBlaze

Summary

- **QR Decomposition is widely used in scientific applications.**

  Radar adaptive beam-forming requires high-speed QR

- **Software-based MicroBlaze Embedded Microprocessor**
  - **Single-chip implementation with multiple cores can outperforms G4**
  - **Similar computational density and efficiency as G4 and Pentium4**

  ***QR program not hand-coded for G4 Altivec or Pentium MMX or MicroBlaze**

- **Linear array for Givens computation**
  - **Two orders of magnitude better than programmable processors in computational density**
  - **One order of magnitude better than programmable processors in computational efficiency**