An Innovative High-Performance Architecture for Vector and Matrix Math Algorithms

Presented by: Tim Olson, Architect

HPEC 2002 – September 24, 2002

Authors: Veeraraghavan Anantha, Ph.D.; Christophe Harlé, Ph.D.; Tim Olson, George Yost, Ph.D.

Intrinsity FastMATH™ Vector and Matrix Math Processor

Optimized for real-time and adaptive signal processing needs:

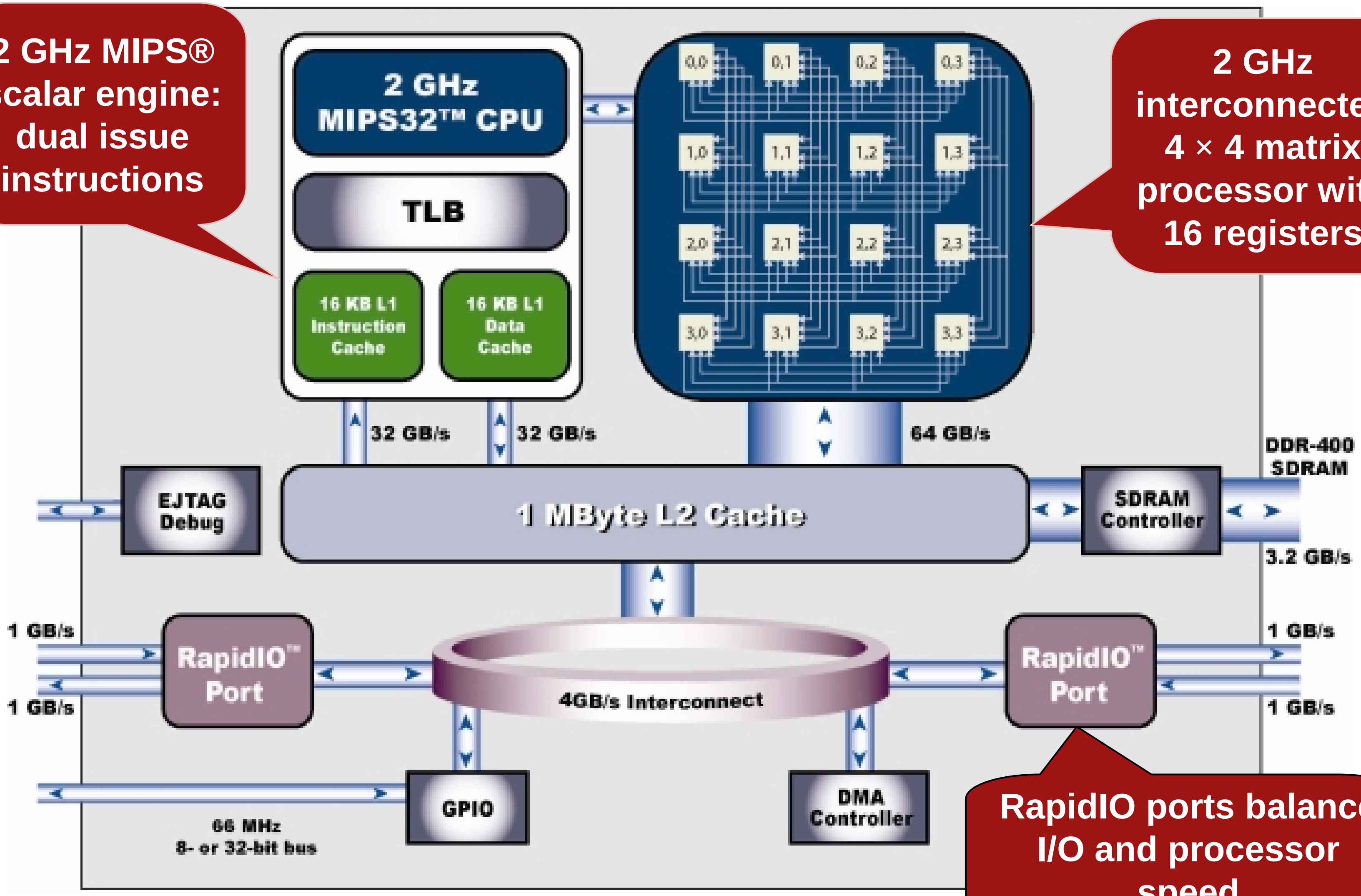
Innovative architecture:

- *2 GHz SIMD 4 × 4 matrix engine with multiprocessor scalability due to high bandwidth RapidIO™ interfaces*
- Fixed-point math
- High-level (e.g., C) language programmable
  - Compiler built-in matrix intrinsics
  - Vector/matrix library

- On-chip matrix coprocessor and MIPS32™ ISA RISC core
- 4 × 4 array of processors, each with sixteen 32-bit registers, two 40-bit MACs
- 64 GOPS (peak)
- Matrix and vector math native instructions: 1-, 8-, 16-, 32-bit support; convenient complex math
- Descriptor-based DMA controller
- 1 Mbyte on-chip cache-coherent L2 cache

Speed *plus* an architecture designed for parallel computations

Intrinsity FastMATH Vector and Matrix Math Processor



2 GHz MIPS® scalar engine: dual issue instructions

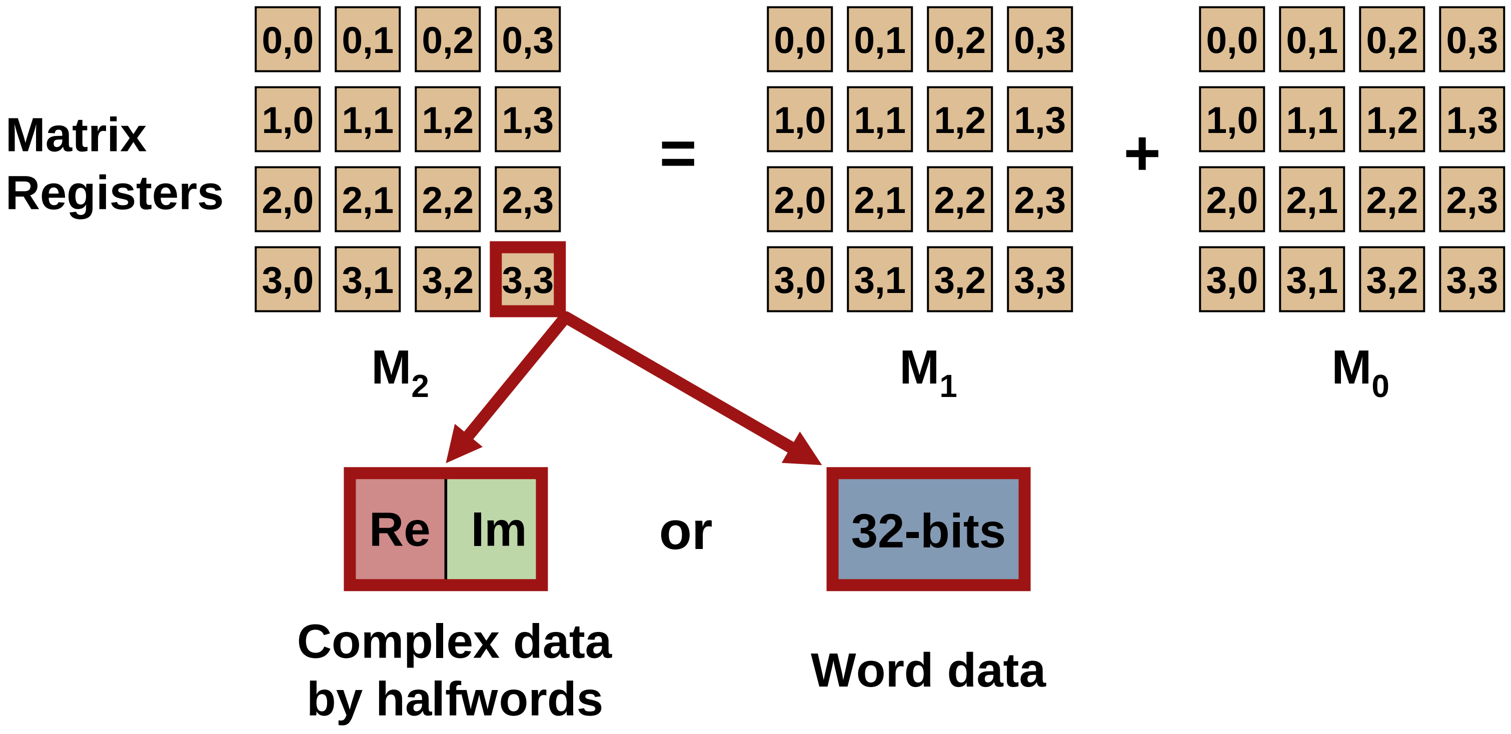
2 GHz interconnected 4 × 4 matrix processor with 16 registers

RapidIO ports balance I/O and processor speed

Matrix Register Arithmetic: Element-by-Element

The matrix engine has 16 matrix registers, each with 16 32-bit values. Halfword and word arithmetic is supported.

Single instruction, element-wise addition of two 4 × 4 matrices



Matrix Register Arithmetic: Matrix Multiplication

Matrix-multiply of two 4 × 4 submatrices by halfword, for example to support 16-bit complex arithmetic

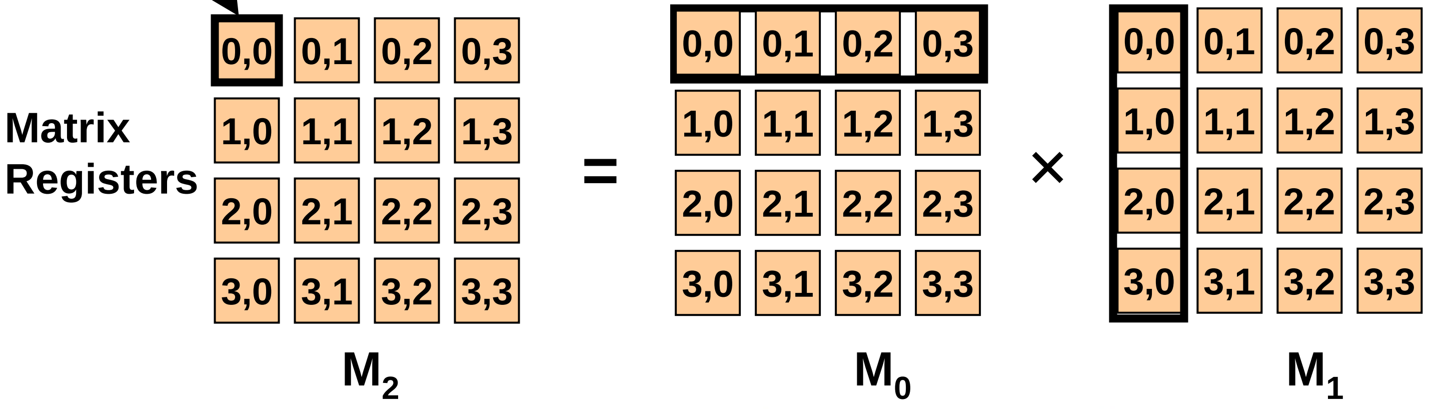
One instruction → **matmulhh.m.m   M2,M0,M1**

- **Four cycles (2 ns @ 2 GHz)**
- **128 operations**

$$\sum_{k=0}^{3} M0^h(0,k) \times M1^h(k,0)$$

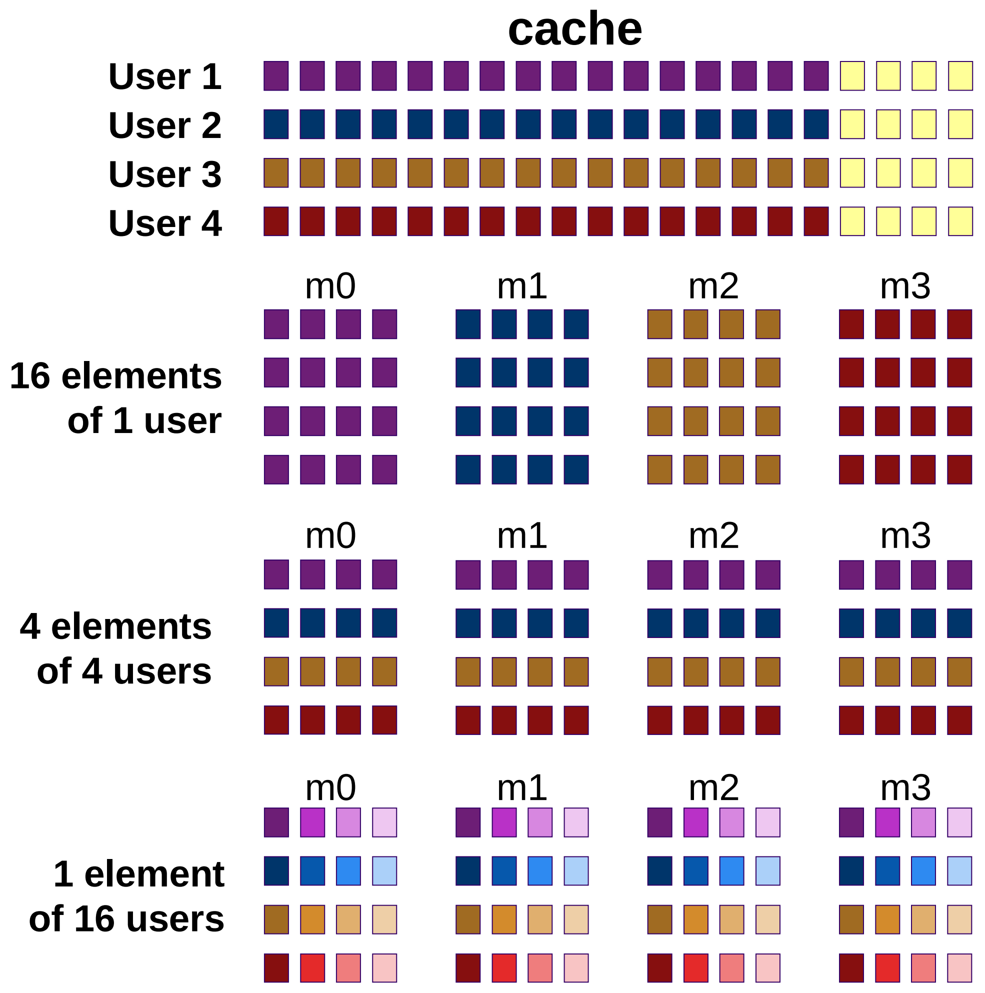
```
for i = 0 to 3
   for j = 0 to 3
      sum = 0
      for k = 0 to 3
          sum = sum + $M0^h(i,k) \times M1^h(k,j)$;
      $M2^h(i,j)$ = sum;
```

High-high halfword multiply, e.g., re × re



Can subdivide large matrices into 4 × 4 parts for multiplication

Matrix Register Arithmetic: Block Rearrangement for Parallelism



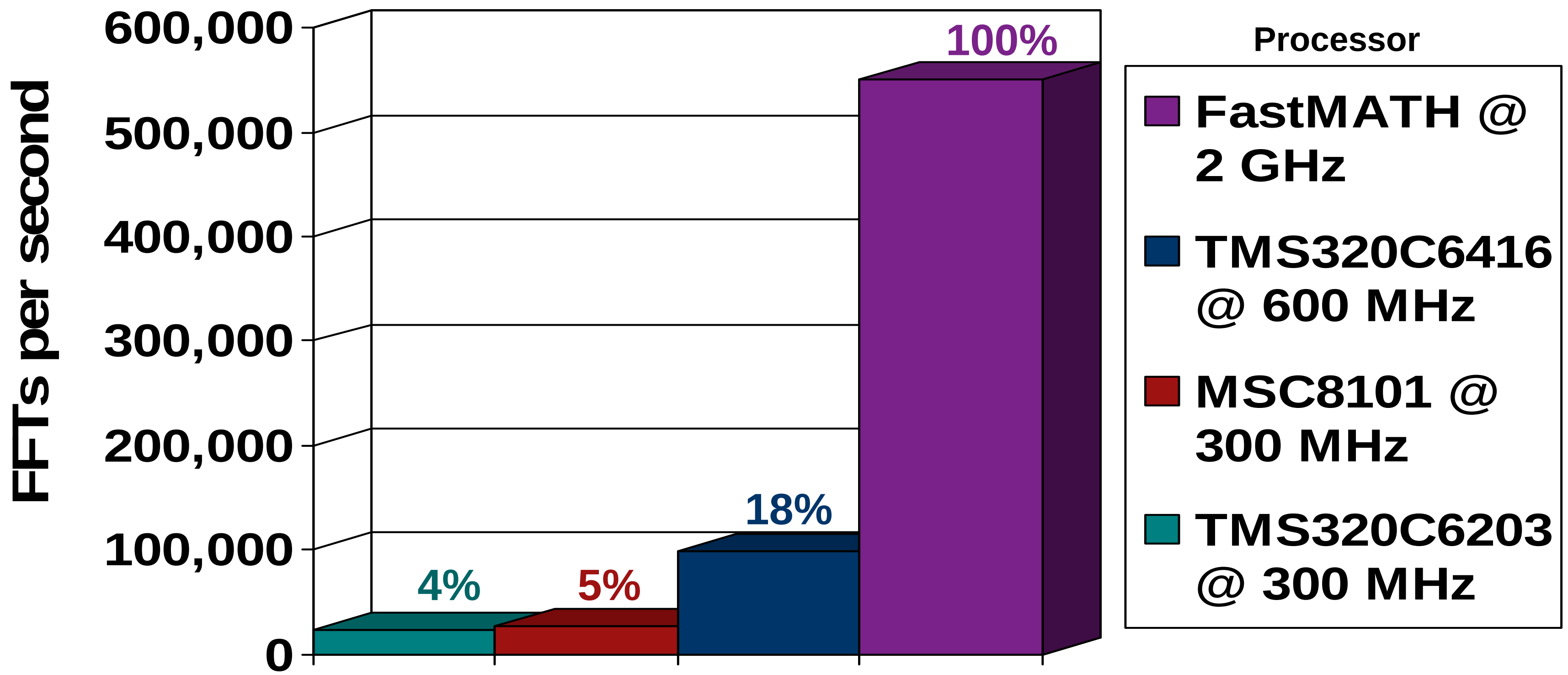
Load 4 or 16 data streams (users) and re-block for SIMD parallel processing

- **Original register `load` instructions**
- **`block4` (four cycles): matrix operations on four streams**
- **For SIMD operations on *16 parallel data streams*: continue rearrangement with block data movement instructions—70 cycles (35 ns) total**

FastMATH Performance Example: Fast Fourier Transform

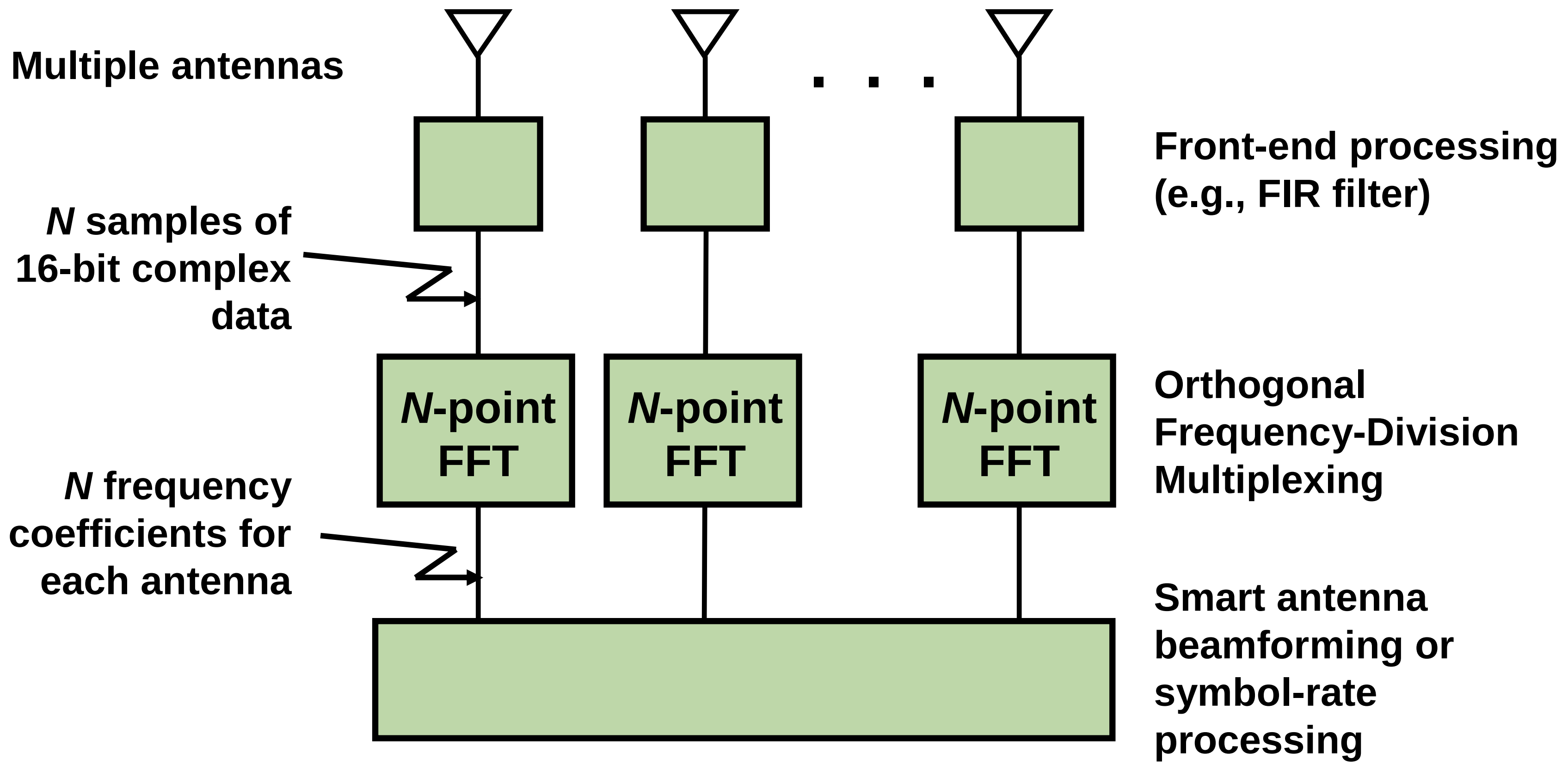
Matrix architecture plus cycle speed combine approximately equally for advantage on this key benchmark

1 K Radix-4 FFT, 16-bit complex data



Notes: Competitive data from published benchmarks
Competitive clock rates are highest announced

FastMATH Performance Example: FFT to Implement OFDM

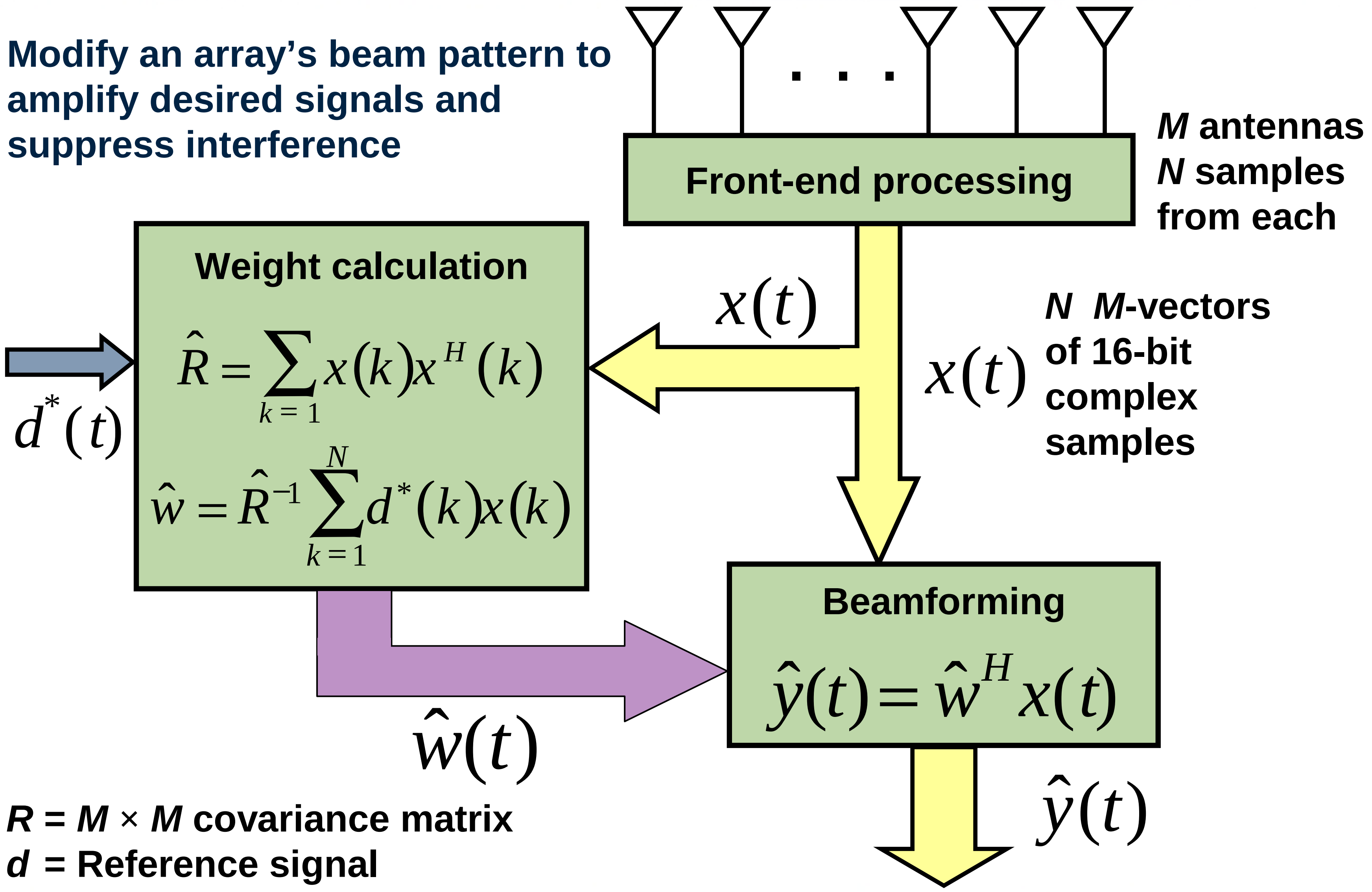


Multiple antennas
. . .
Front-end processing (e.g., FIR filter)
N samples of 16-bit complex data
N-point FFT
N-point FFT
N-point FFT
Orthogonal Frequency-Division Multiplexing
N frequency coefficients for each antenna
Smart antenna beamforming or symbol-rate processing

Example results:
for 8 antennas, 10 Msamples per second, 1024-pt complex FFT: requires *14.4% FastMATH processor*

FastMATH Performance Example: Smart Antennas

Modify an array's beam pattern to amplify desired signals and suppress interference



Front-end processing
M antennas
N samples from each
$x(t)$
$x(t)$
N *M*-vectors of 16-bit complex samples
Weight calculation
$\hat{R} = \sum_{k=1} x(k) x^H(k)$
$\hat{w} = \hat{R}^{-1} \sum_{k=1}^{N} d^*(k) x(k)$
$d^*(t)$
$\hat{w}(t)$
Beamforming
$\hat{y}(t) = \hat{w}^H x(t)$
$\hat{y}(t)$

R **=** ***M*** **×** ***M*** **covariance matrix**
d **= Reference signal**

FastMATH Performance Example: Smart Antennas

Background

- More users than antennas $\Rightarrow$ orthogonal beams not possible
- No a priori information about signal directions $\Rightarrow$ need real-time adaptation
- Input stream is 16-bit complex data

FastMATH Implementation

- Covariance matrix calculated by *complex matrix-matrix multiplications on 4 × 4 submatrices*, then re-assembling full matrix
- Covariance matrix inverted by Cholesky decomposition; use *block matrix manipulation* instructions to rearrange input into blocks for SIMD parallelization
- Beamforming using matrix-matrix multiplications; more efficient than simple vector math

WCDMA Example Results

- With 64 voice users and 16 antennas, 4 rake fingers per user, weights updated every slot: 0.73 FastMATH processors

Scaled Multiprocessor Example: CDMA Multi-User Detection

Algorithms

- ***Mitigate interference between users in CDMA***
- **Solve for estimators for correct symbols, beginning with user-user correlation matrix *R* and user input vector *y***
- **Difference equation for interference on symbol m of desired user from near-by symbols of all other users:**

$$y_m = \sum_{k=-K}^{K} R_{m-k} \hat{b}_{m-k}$$

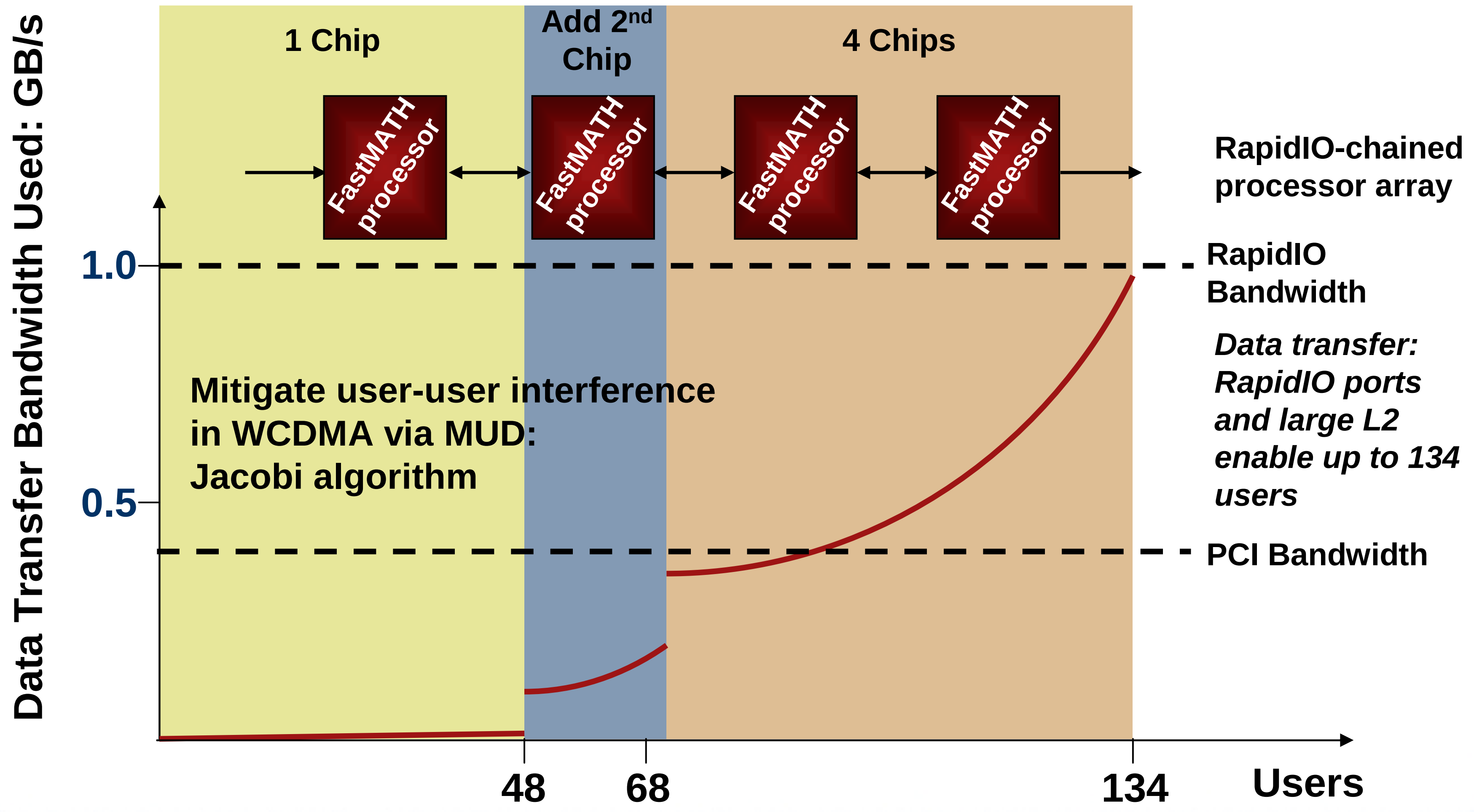
- $\hat{b}$ **is desired estimator vector for symbol *m* of *N* users to be found**

Implementation

- ***Jacobi iteration*: Solve for matrix *B* of *M* symbols for *N* users. Perform matrix-matrix multiplications distributed over processors**
- **Calculate correlation matrices *R* on chip; large capacity L2 cache reduces data transfer**
- **At each iteration exchange partial results over RapidIO port via DMA**
- **RapidIO interfaces work in background in parallel with computations – data transfer time efficiently hidden**

Scaled Multiprocessor Example: WCDMA Short Code Multi-User Detection

- Data transfer in parallel with computation
- Scalable multiprocessor system distributing tasks and results over RapidIO interface via coherent L2 cache


Data Transfer Bandwidth Used: GB/s
1 Chip
Add 2nd Chip
4 Chips
FastMATH processor
FastMATH processor
FastMATH processor
FastMATH processor
RapidIO-chained processor array
RapidIO Bandwidth
Data transfer: RapidIO ports and large L2 enable up to 134 users
Mitigate user-user interference in WCDMA via MUD: Jacobi algorithm
PCI Bandwidth
Users