Vforce:
Aiding the Productivity and Portability in Reconfigurable Supercomputer Applications via Runtime Hardware Binding

Nicholas Moore, Miriam Leeser

Department of Electrical and Computer Engineering
Northeastern University, Boston MA

Laurie Smith King
Department of Mathematics and Computer Science
College of the Holy Cross, Worcester MA

Outline

- COTS Heterogeneous systems: Processors + FPGAs
- VSIPL++ & the VForce framework
- Runtime hardware binding and runtime resource management
- Results:
  - FFT on Cray XD1:
    - Measuring VForce overhead
  - Beamforming on
    - Mercury VME system
    - Cray XD1
- Future directions

Heterogeneous Reconfigurable Systems

Cray XD1

www.cray.com/products/xd1/

- Xilinx Virtex 2s paired with AMD Opteron nodes
- RapidArray interconnect

Mercury PowerStream

http://www.mc.com/products/

- Interchangeable PPC and FPGA daughtercards housed in chassis
- Race++, RapidIO interconnect

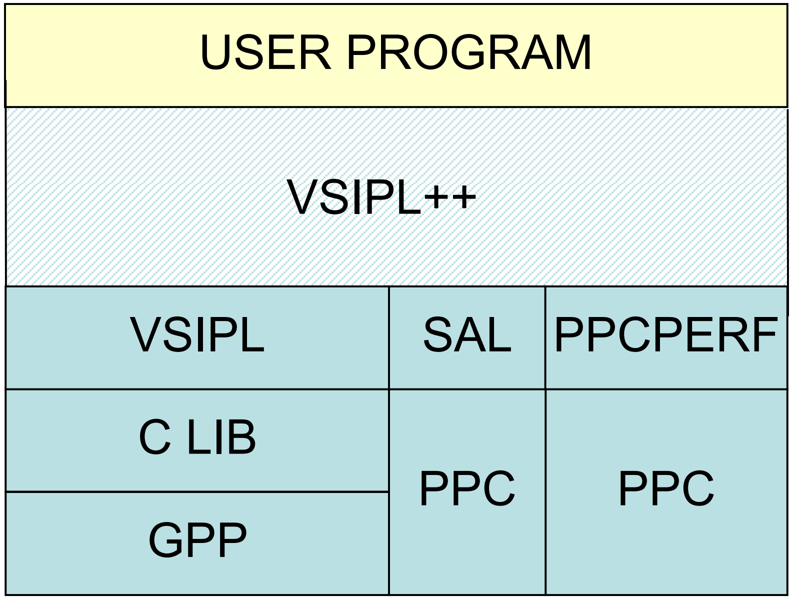
Portability for Heterogeneous Processing

- All systems contain common elements:
  - Microprocessors
  - Distributed memory
  - Special-purpose computing resources
    - FPGAs are our focus
    - also GPUs, DSPs, Cell ...
  - Communication channels
- Currently no support for application portability across different platforms
- Redesign required for hardware upgrades, move to new architecture
- Focus on commonalities, abstract away differences
- Deliver performance

What is VSIPL++ ?

- An open API standard from HPEC-SI
- A library of common signal processing functions
  - Data objects interact intuitively with processing objects
  - High level interfaces ease development

- Implementation can be optimized for a given platform
  - run-time performance depends on implementation
  - Different implementations of VSIPL++ are available from different vendors



USER PROGRAM

VSIPL++

VSIPL | SAL | PPCPERF

C LIB | PPC | PPC

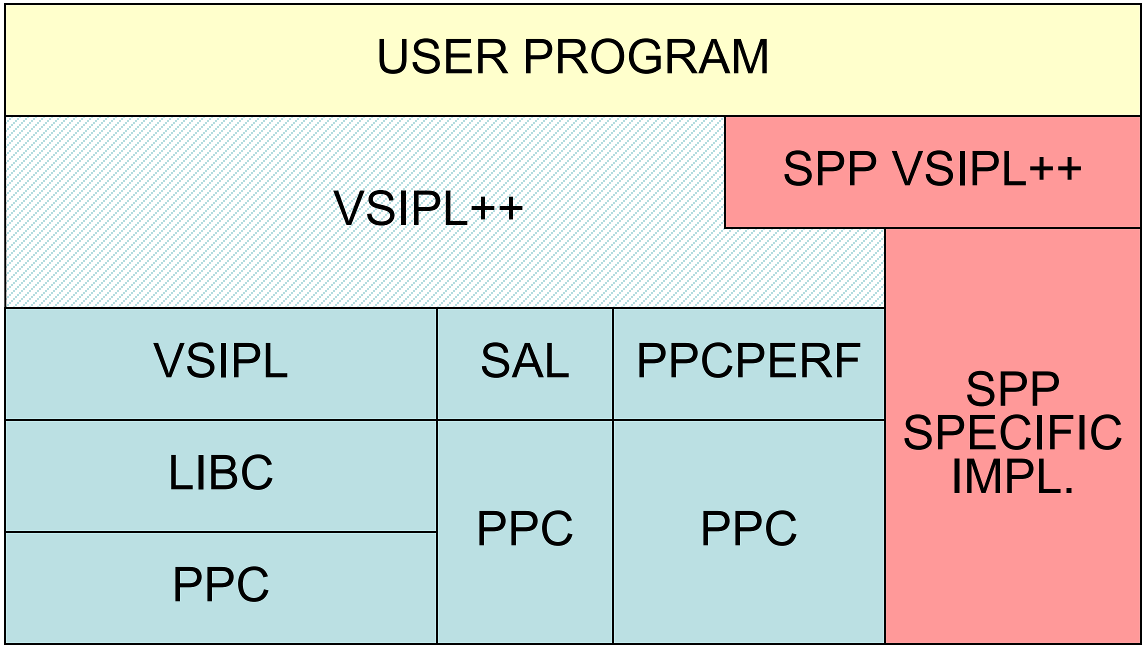
GPP

VForce: Extending VSIPL++

- VForce: a middleware framework
- adds support for special purpose processors (SPPs) to VSIPL++
  - Currently FPGAs
- Programmer uses VSIPL++ processing and data objects
  - Custom processing objects utilize a Generic Hardware Object (GHO) that interacts with VForce
  - Run time resource manager uses SPP implementations when available (defaults to software)
- Standard API between processing objects and hardware resources

VFerce: Extending VSIPL++

- Sits on top of VSIPL++
  - Implementation Independent
- Custom processing objects:
  - Overload a subset of VSIPL++ functions
  - Add new higher level functions → SPP's strength



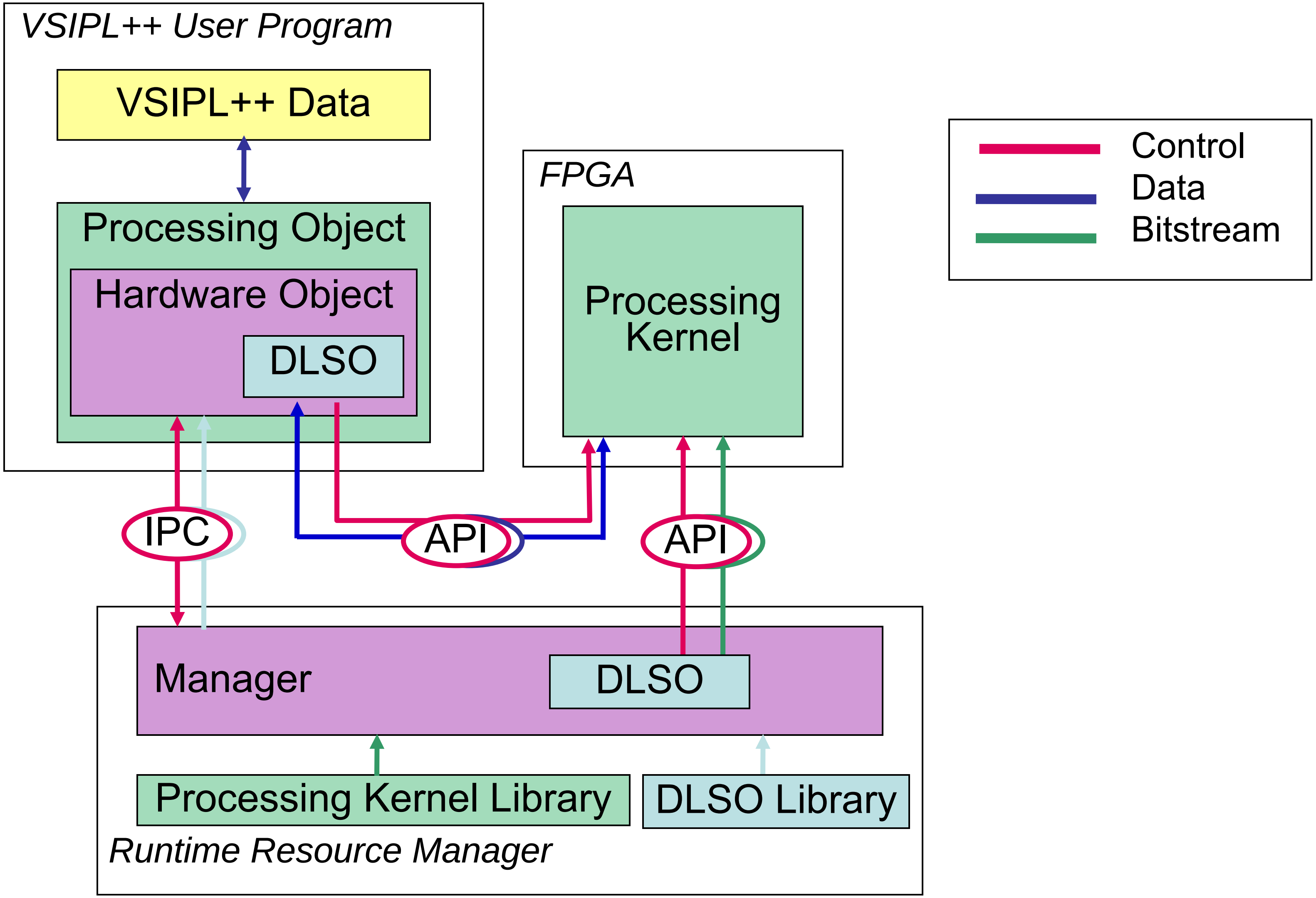
VForce API

- Generic Hardware Object (GHO) implements a standard API:
  - Move data to and from the Special Purpose Processor (SPP)
  - Configure algorithm parameters
  - Initialize and finalize SPP kernels
  - Start processing
  - Check interrupt status
- A processing object uses these hardware functions to interact with the SPP

Dynamically Loaded Shared Objects (DLSO)

- Generic Hardware Object (GHO) is hardware independent
- Use dynamically loaded shared objects(DLSO) to control a specific SPP
- Each type of SPP requires a pre-compiled DLSO that converts the standard VForce API into vendor specific calls
- Separation of hardware concerns from user code and from binary until run time
- Which DLSO and device?
  - Determined by a Run Time Resource Manager (RTRM)

Control and Data Flow



Northeastern University

VForce Framework Benefits

- VSIPL++ code easily migrates from one hardware platform to another
- Specifics of hardware platform encapsulated in the manager and DLSOs
- Handles multiple CPUs, multiple FPGAs efficiently
- Programmer need not worry about details or availability of types of processing elements
- Resource manager enables run time services:
  - fault-tolerance
  - load balancing

Extending Vforce

- Adding Hardware Support:
  - Hardware DLSO
  - Processing Kernels
    - Can be generated by a compiler or manually
- Adding Processing Objects:
  - Write a new processing class
    - Use GHO to interface with hardware
    - Include software failsafe implementation
  - Corresponding processing kernel
- One to many mapping of processing objects to kernels

Vforce FFT Processing Object

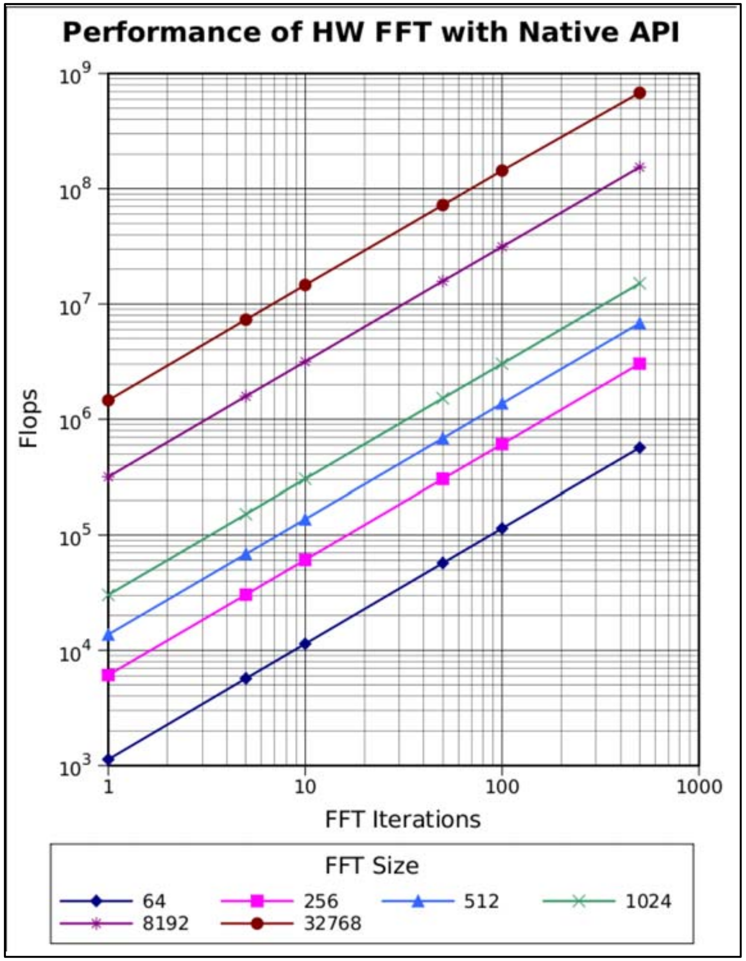
- Matches functionality of the FFT class within VSIPL++
  - Uses VSIPL++ FFT for SW implementation
- Cray XD1 FPGA Implementation
  - Supports 8 to 32k point 1D-FFT
  - Scaling factor and FFT size adjustable after FPGA configuration
  - Uses parameterized FFT core from Xilinx Corelib
  - Complex single precision floats in VSIPL++ converted to fixed point for computation in hardware (using NU floating point library)
  - Dynamic scaling for improved fixed point precision

Vforce FFT Overhead on the Cray XD1

- Two situations examined:
  1) Vforce HW vs. Native API HW
     - Currently the default operation when SPP present, even if the CPU is faster
     - Run times include data transfer and IPC
  2) Vforce SW vs. VSIPL++ SW
     - Vforce SW the fall back mode on SPP error or negative response from RTRM
     - Includes IPC
- In both cases Vforce delays instantiation of the VSIPL++ FFT until it is used

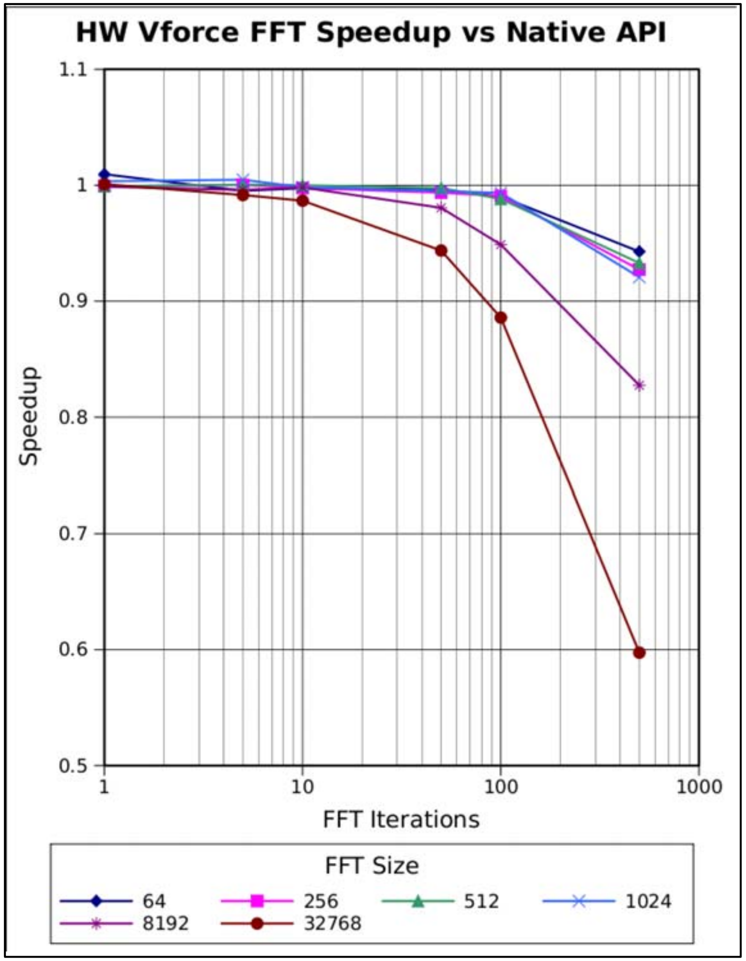
1) Native HW Performance

- Includes one FPGA configuration & multiple uses:
  - Configuration time amortized over number of iterations
- Time of an individual iteration is dominated by communication time (control setup)



1) HW Overhead

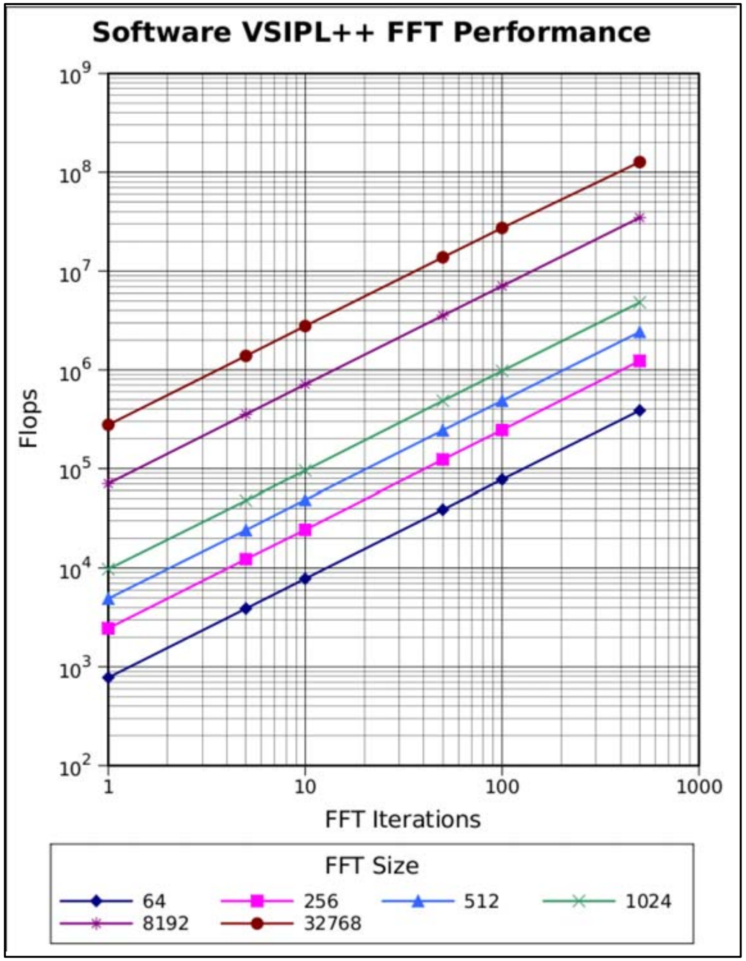
- Can see the effects of data copying
  - XD1 needs page aligned DMA buffers
  - Assuming VSIPL++ views opaque:
    - There is one copy from a view into a DMA-able block
    - We plan to look at using user admitted views
- No concurrent processing
- Current DLSO sets up DMA every time



Northeastern UNIVERSITY

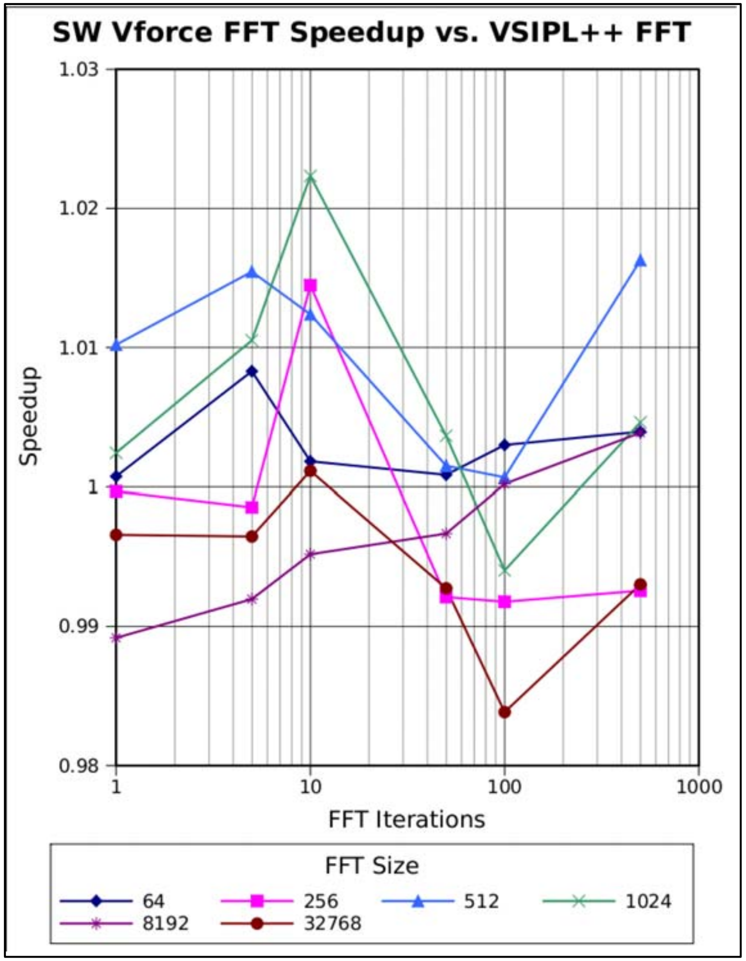
2) VSIPL++ FFT Performance

- VSIPL++ Reference Version 1.01
- Large setup time
  - Same for VSIPL++ SW FFT and Vforce FFT



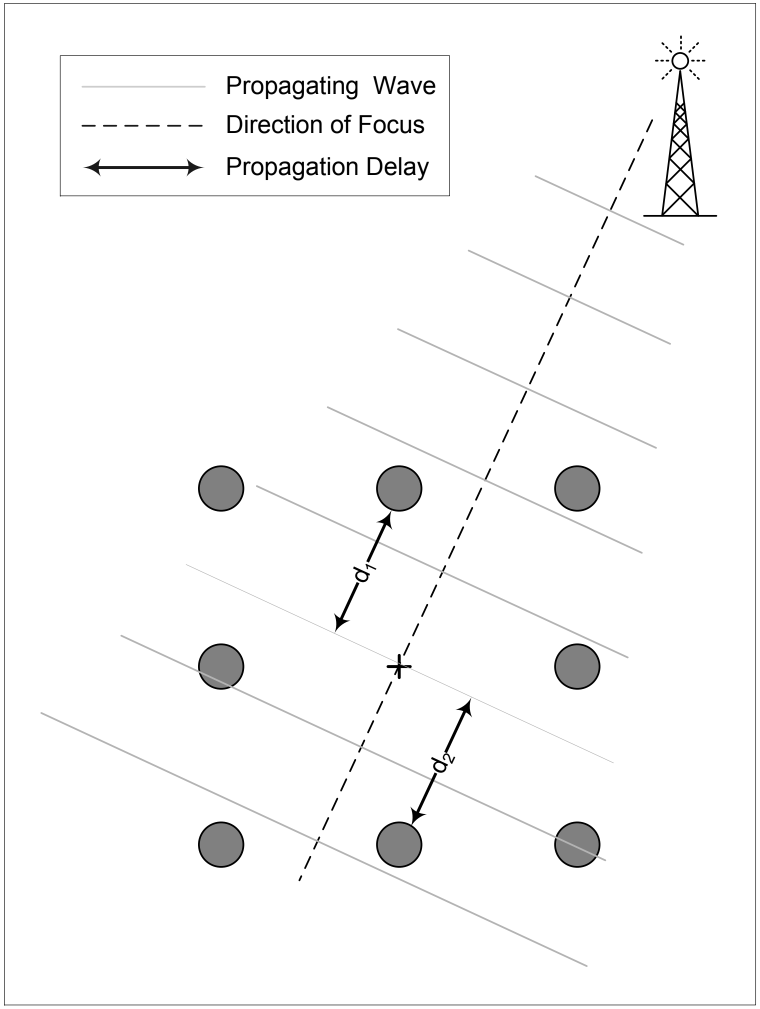
3) Vforce SW Overhead

- Vforce FFT uses the VSIPL++ FFT
- Difference in performance is overhead
  - RTRM running, always no hardware available
  - Defaults to SW
  - Only check once for available hardware
- Both versions show approx same speedup
  - Variation due to measurement error



Beamforming: Spatial Filter

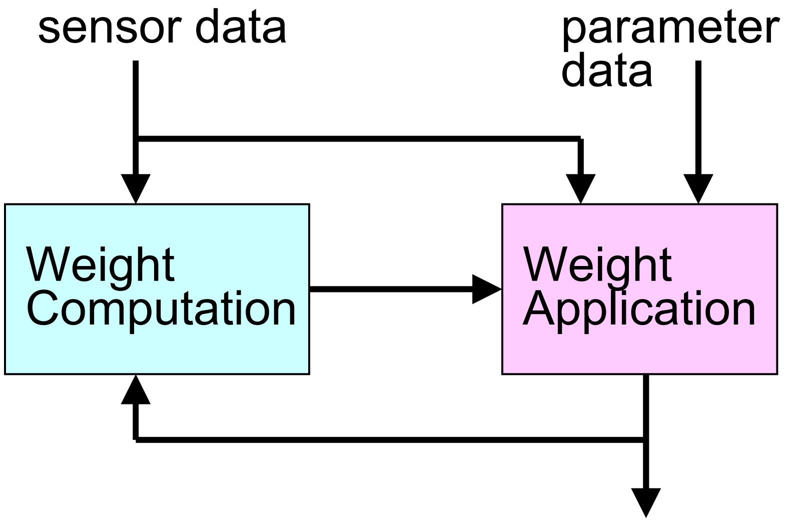
- A collection of techniques used to steer an array of sensors and form beam patterns to null interference
- Applications in radar, sonar, medical imaging, wireless communication



Beamformer Implementation

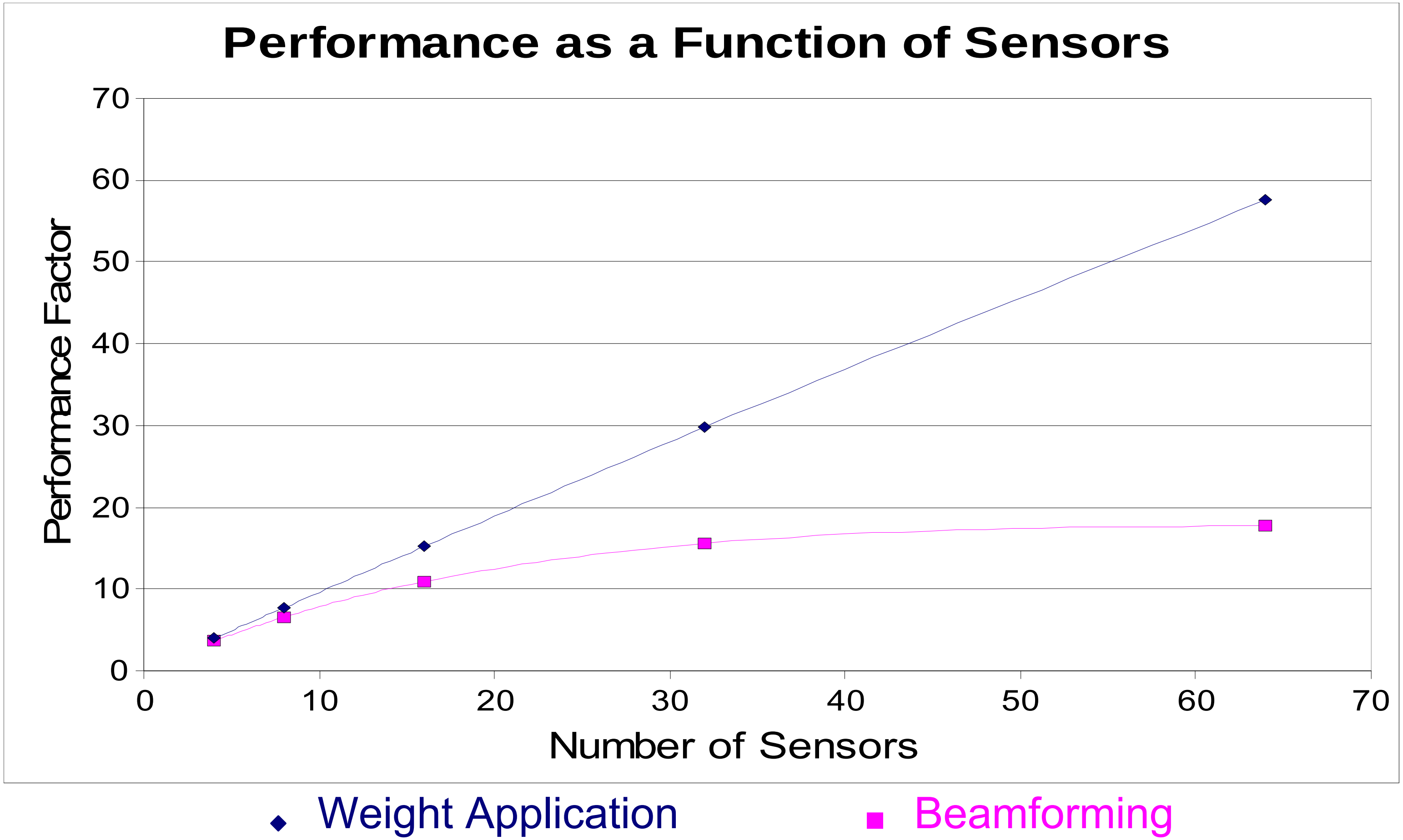
- 3-D time-domain beamformer, adaptive weights
- Single precision floating point operators at every stage
- Hybrid hardware/software implementation

- Weights computed periodically
  - in software
- Weights applied (multiply accumulate)
  - in hardware or software
- Implemented on
  - Mercury 6U VME (version 1.1)
  - Cray XD1 (version 1.2)



Beamformer: MCS 6U VME

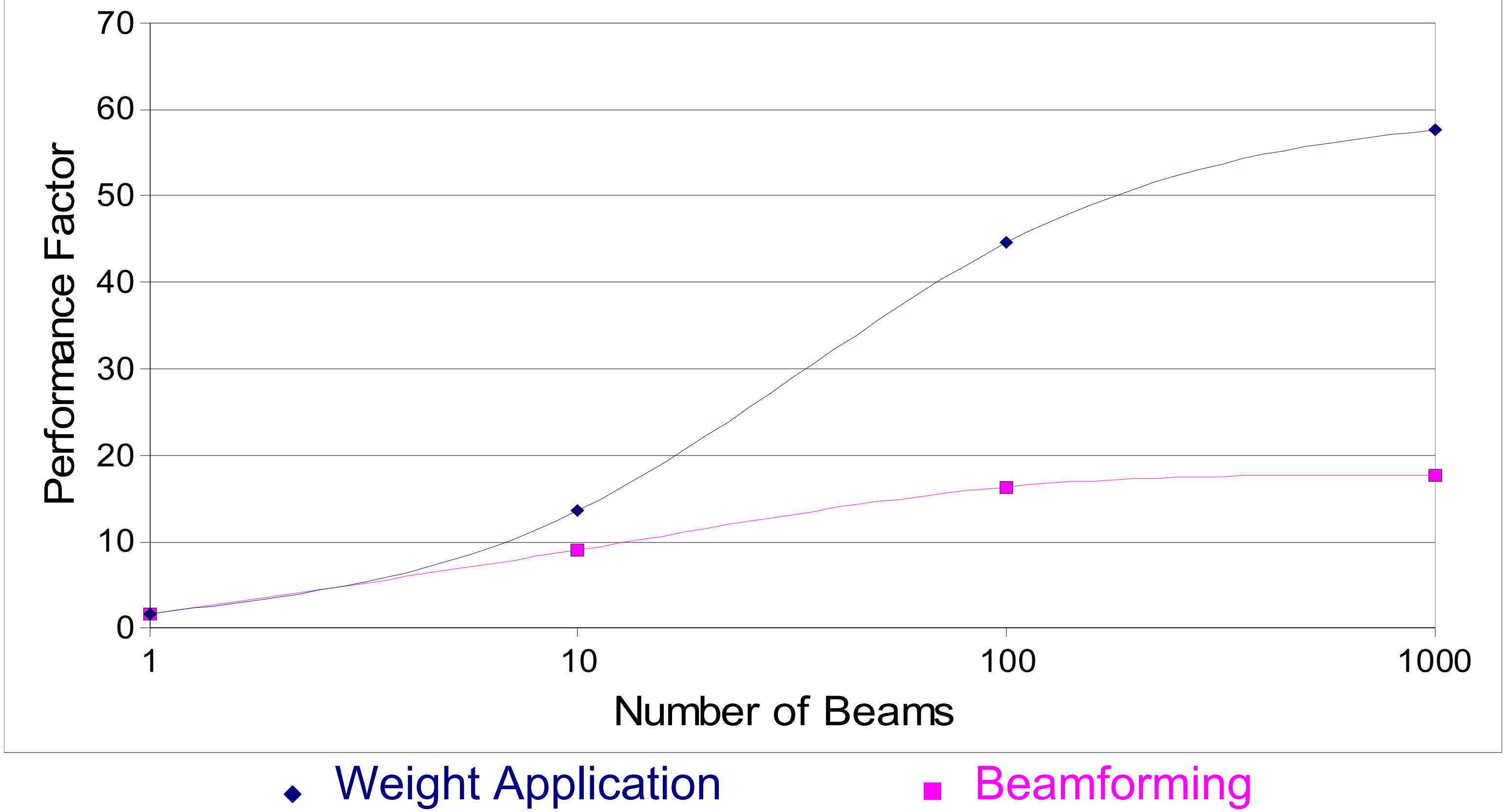
- Data transfer dominates beamforming
- Implemented before non-blocking data transfer implemented
  - – Vforce version 1.1
  - – Now Vforce version can run weight computation while data is being transferred to FPGA
- Overall speedup ranges from ~1.2 to >200
  - – Largest speedups on unrealistic scenarios
    - Many beams, few sensors



Exp	Sen	Eqns	Period	Beams
4	4	1024	256k	1000
8	8	512	128k	1000
12	16	256	64k	1000
16	32	128	32k	1000
20	64	64	16k	1000

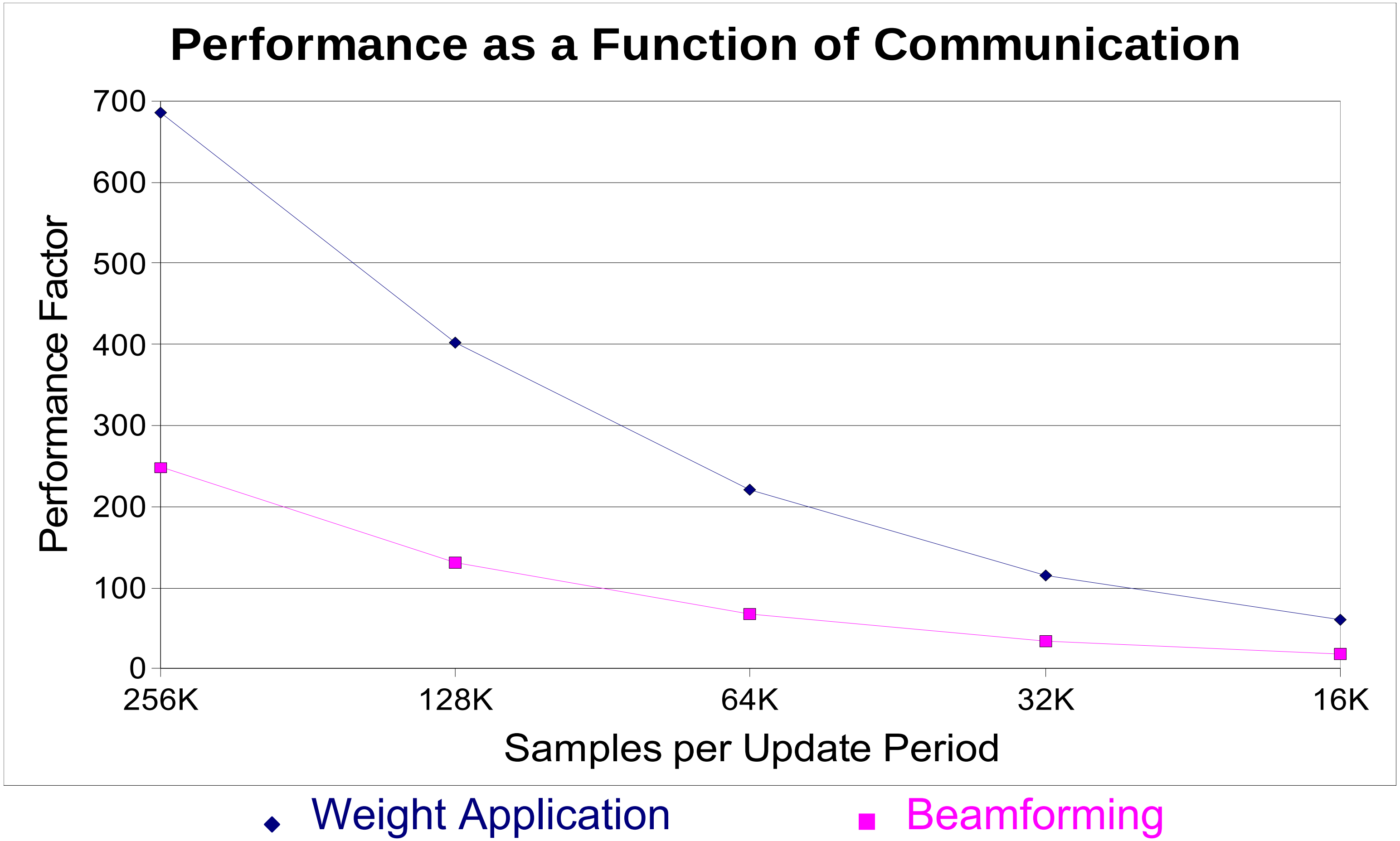
- Weight application performance improves linearly as a function of sensors
- Weight computation run-time increases as a function of sensors
- At 1000 beams, weight application run-time limited by result data transfer

Performance as a Function of Beams



Exp	Sen	Eqns	Period	Beams
17	64	64	16k	1
18	64	64	16k	10
19	64	64	16k	100
20	64	64	16k	1000

- As the number of beams increases, more processing is performed per block of sensor data
- Performance gain increases as a function of beams
- The gain is limited by the time required to transfer results



Exp	Sen	Eqns	Period	Beams
21	64	64	16k	10000
22	64	64	32k	10000
23	64	64	64k	10000
24	64	64	128k	10000
25	64	64	256k	10000

- Performance gain is heavily impacted by the rate at which weights are computed and result data is transferred
- More data transfers when the update period is small

Beamformer: Cray XD1

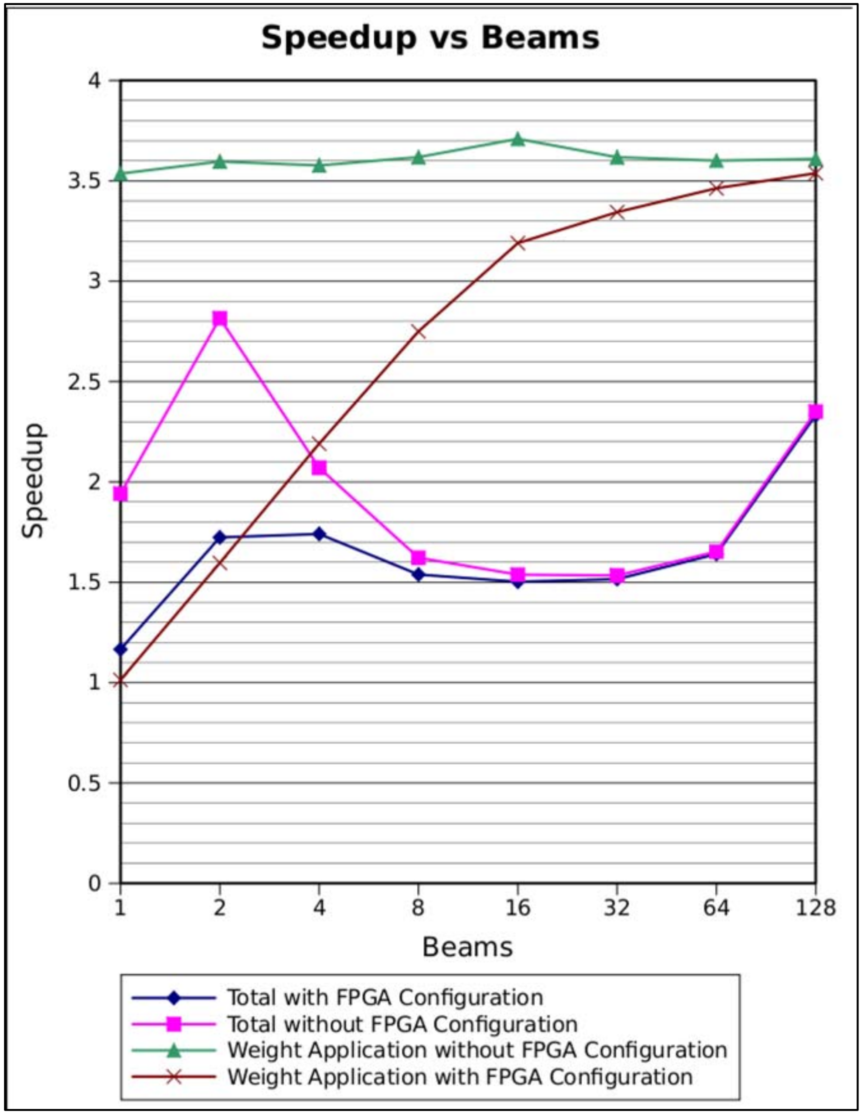
- Uses Vforce 1.2 with non-blocking data transfers
  - Double buffer incoming sensor data
  - Stream results back to CPU as produced
  - Much higher levels of concurrency
  - Data transfer almost completely hidden
    - Don't get the same performance hit with smaller update periods that the Mercury implementation did
- Smaller update periods on XD1
  - More powerful CPU on XD1 allows for more frequent weight computation
- Different hardware accumulator
  - Not as fast as the one used in the Mercury beamformer

Cray XD1 Test Scenarios

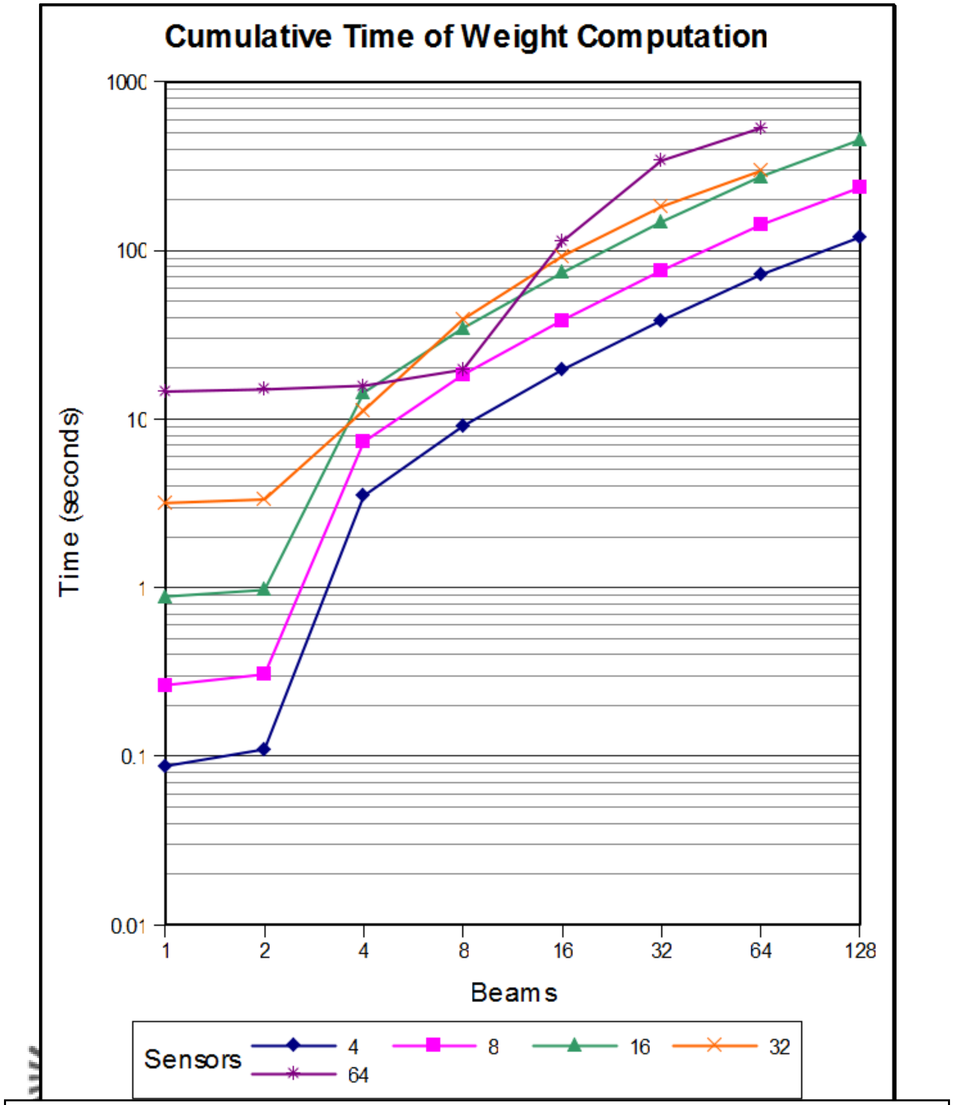
- All combinations (powers of 2) of the following:
  - 4 to 64 sensors
  - 1 to 128 beams
  - 1024 to max allowed time steps per update period, limited by 4 MB RAM banks (varies with sensors)
  - Weight computation history of 5 consecutive powers of 2 ending with half the update period
- Speedup of 1.22 to 4.11 for entire application
  - Excluded extreme values (i.e. 10,000 beams)
  - Much smaller update periods balance CPU/FPGA computation time but limit maximum speedup

Performance vs Beams

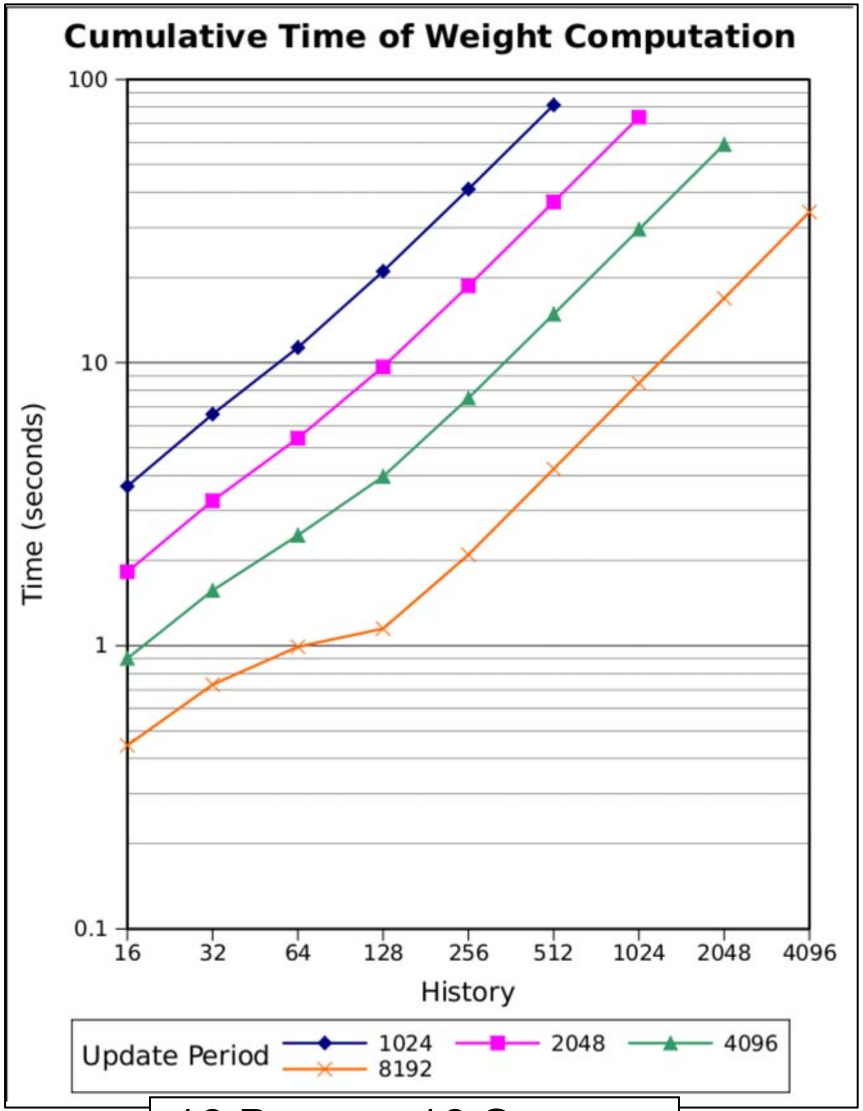
- HW provides a relatively constant ~3.5x speedup for weight application
- Irregularity in weight computation causes initial unpredictability
- This example:
  - 32 sensors
  - 2048 time steps update period
  - 1024 past time steps used in weight computation



SW Weight Computation Performance



2048 update period; 1024 past time steps

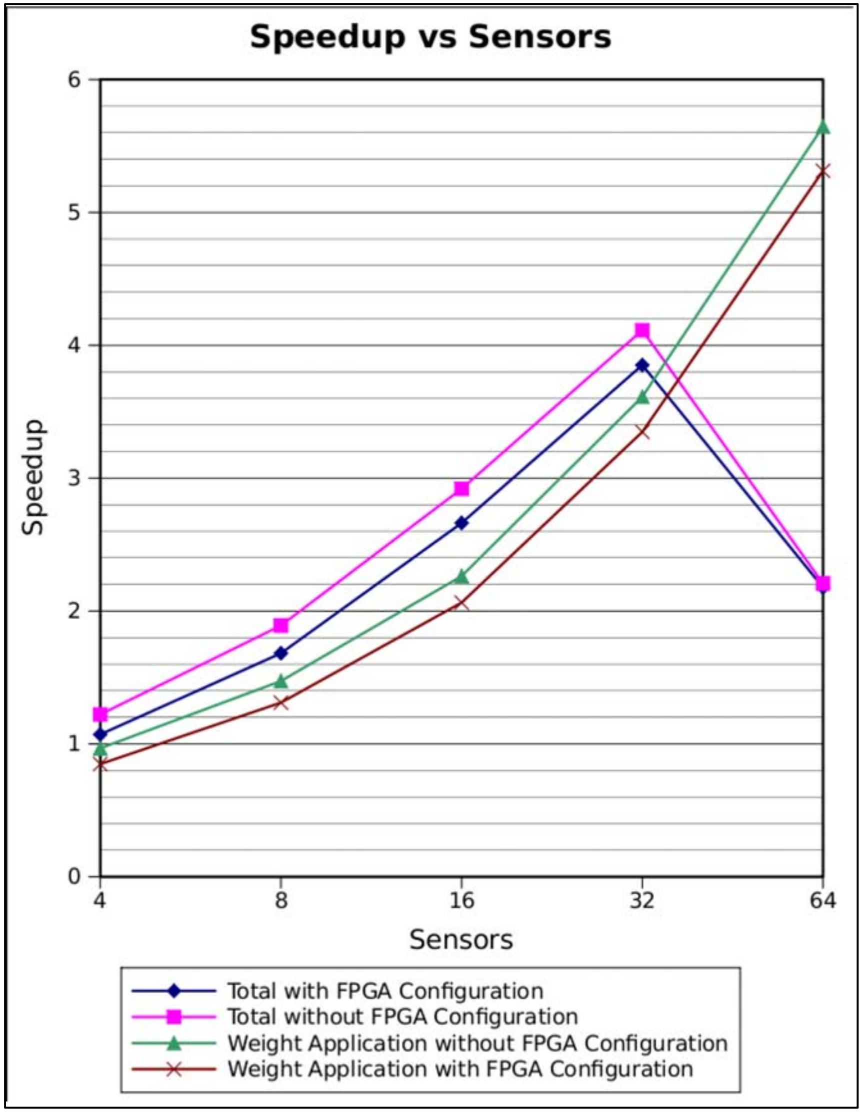


16 Beams; 16 Sensors

Performance vs Sensors

- Large jump in weight computation at 64 sensors
  - Weight computation runtime dominates
  - Limits possible speedup

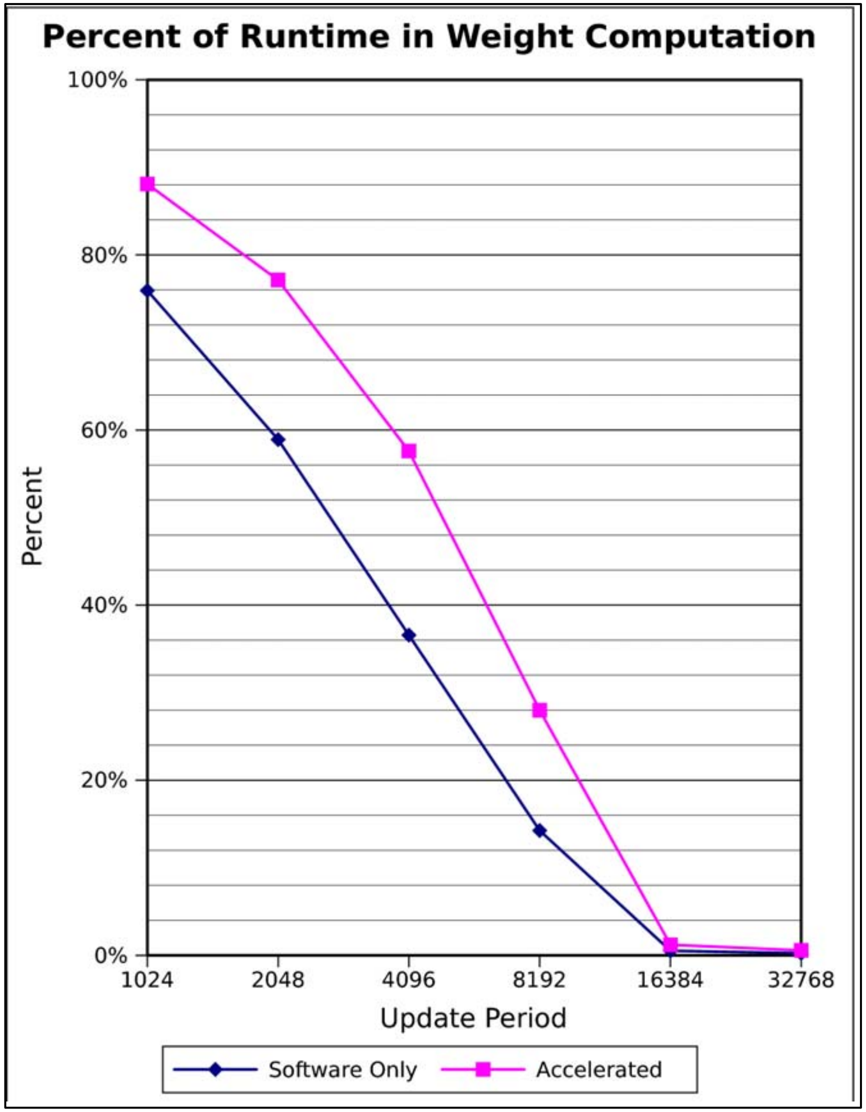
- This scenario:
  - 32 beams
  - 4096 time steps update period
  - 256 past time steps used in weight computation



Northeastern
UNIVERSITY

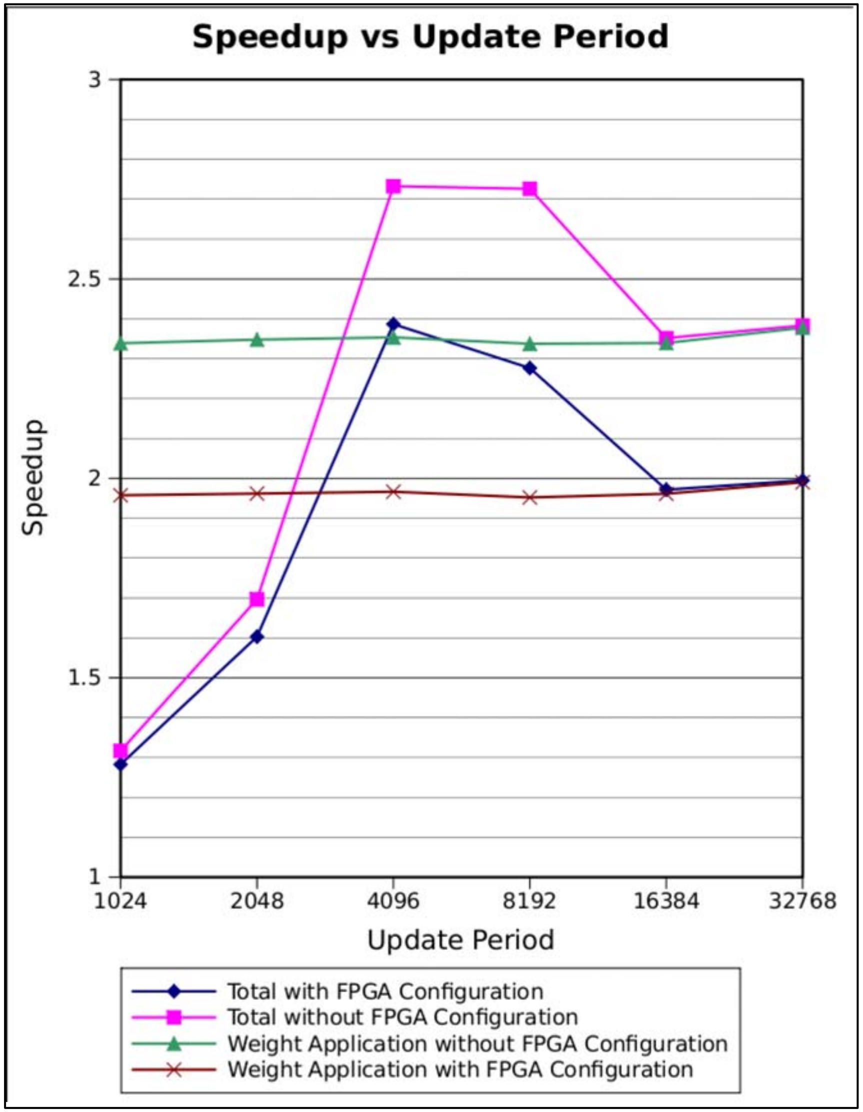
Performance vs Update Period

- Larger update period corresponds to fewer weight computations
  - Makes up less of total runtime
  - Ratio doesn't have a consistent relationship



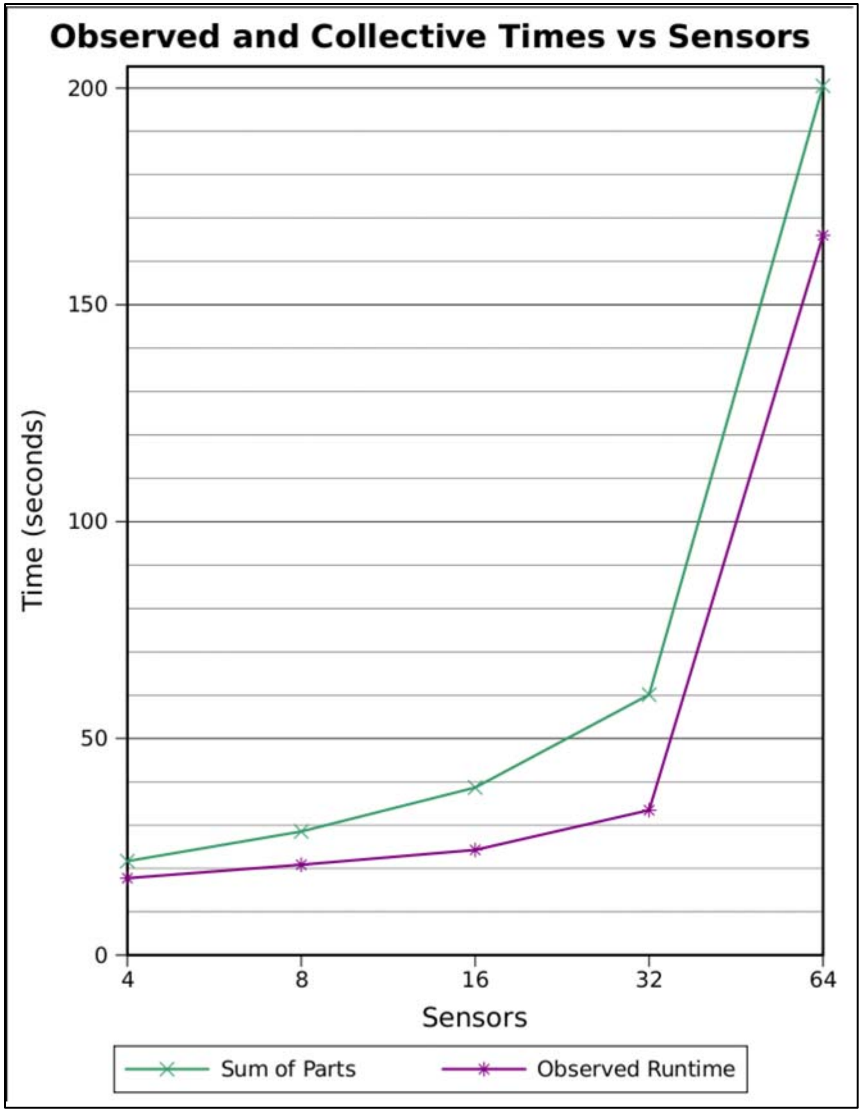
Performance vs Update Period

- Results in irregular total application speedup
- Weight application times are relatively constant
  - Smaller update times don't negatively impact performance (as they did in the Vforce 1.1version)
- This scenario
  - 16 beams
  - 16 sensors
  - 512 past time steps used in weight computation

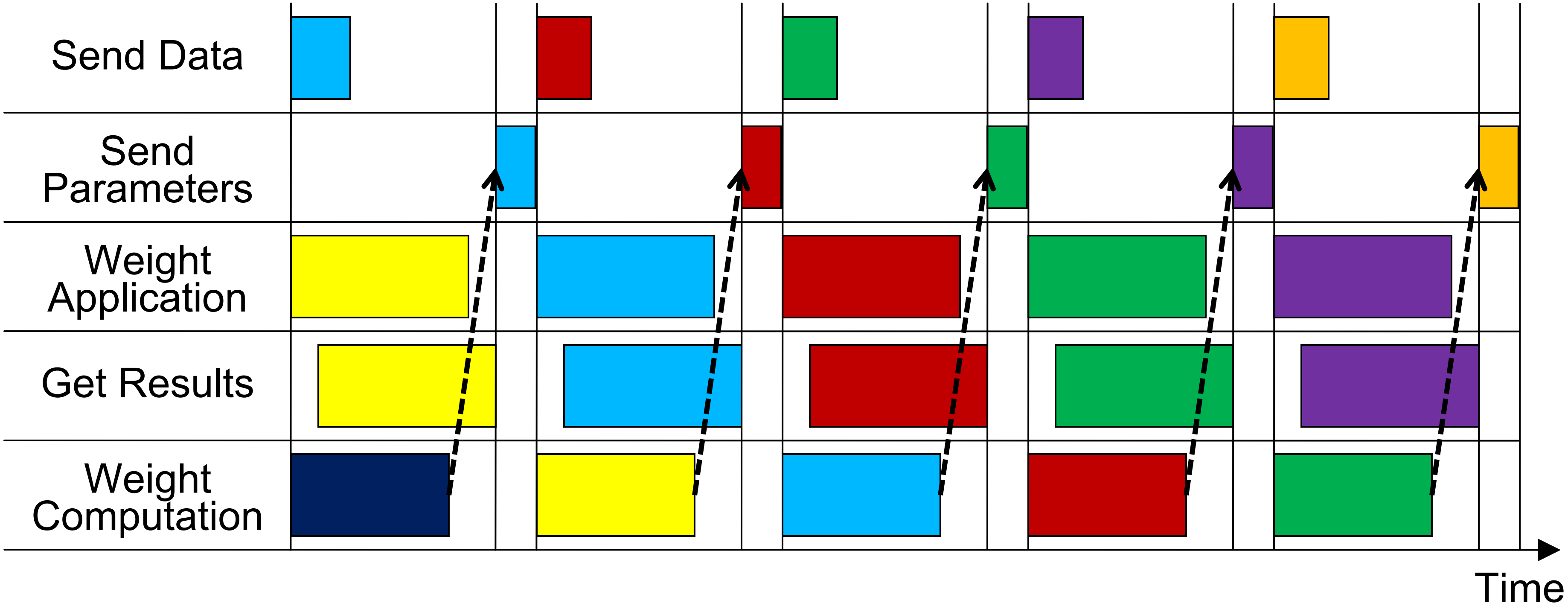


Concurrency

- Extension of VSIPL++ serial specification
- Generally hide the shorter of weight computation and weight application
- Speedups better than Amdahl's law normally allows due to overlapping of operations
- Take advantage of HW



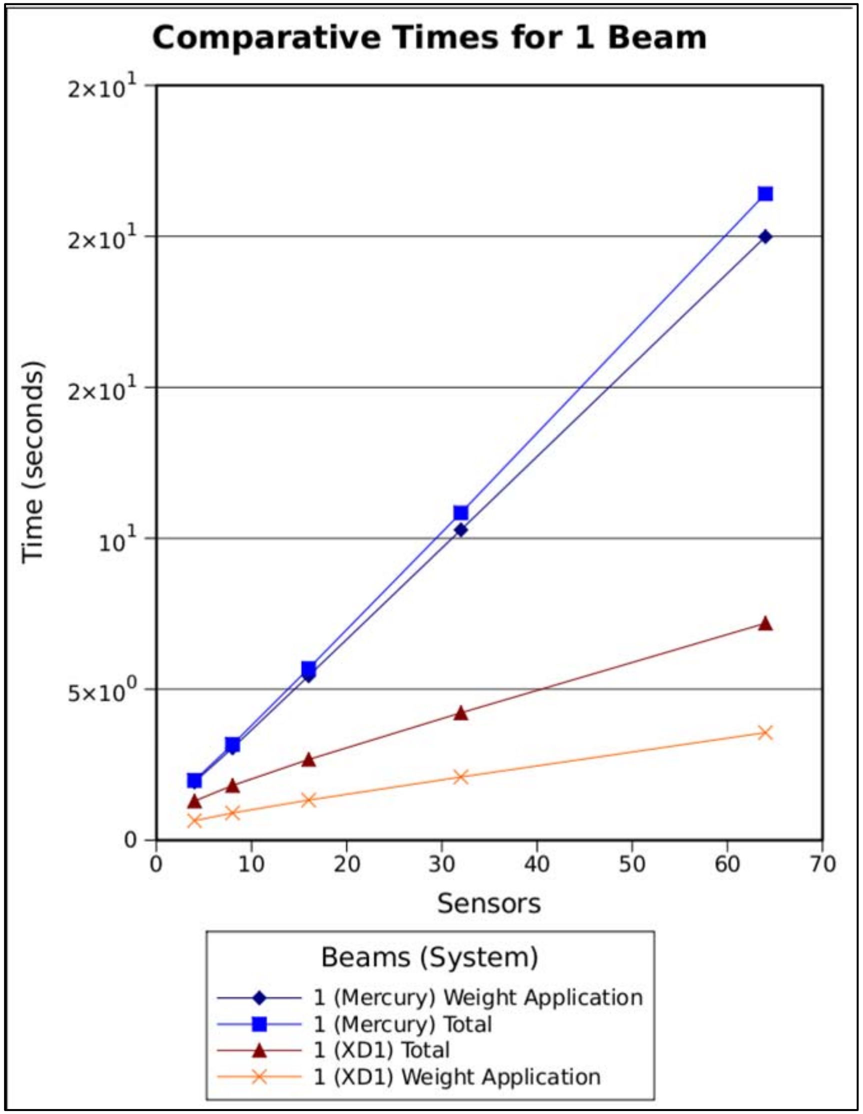
Concurrency



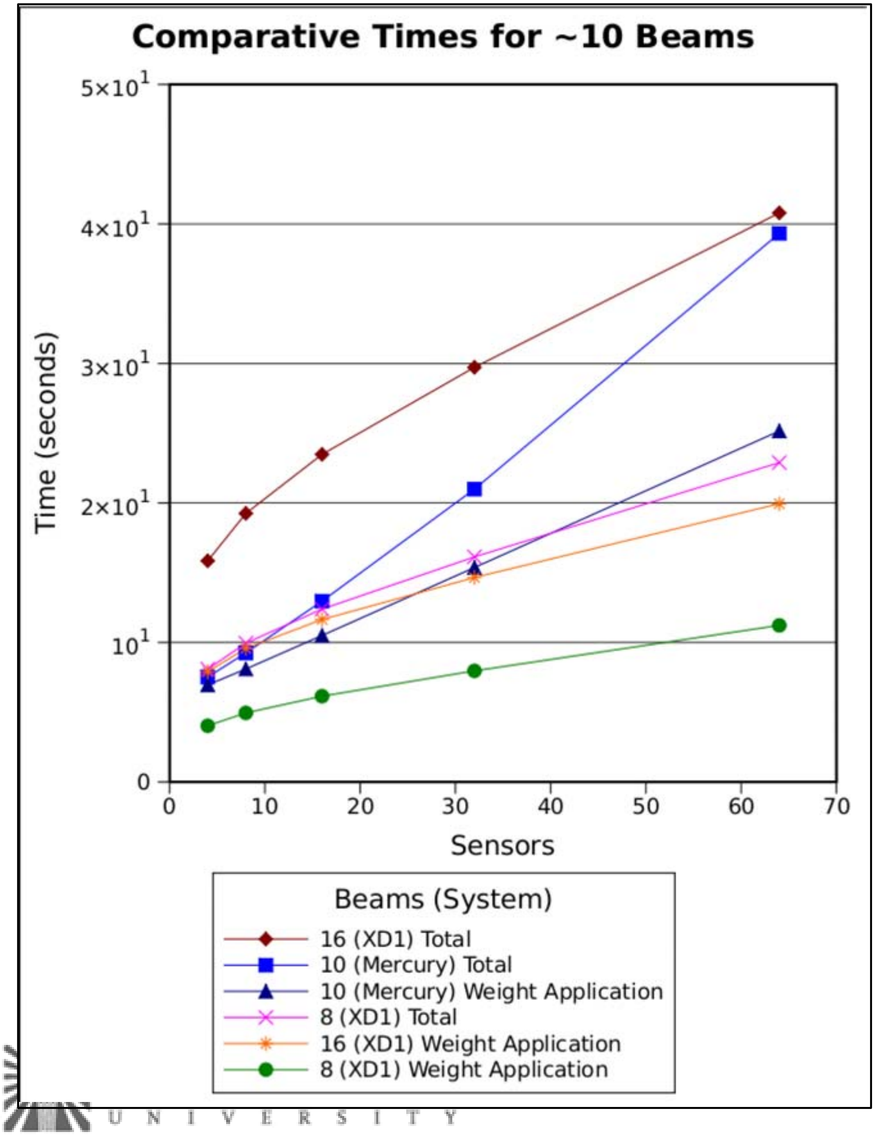
- Not to scale
- Everything overlapped except sending parameters
  - Not double buffered

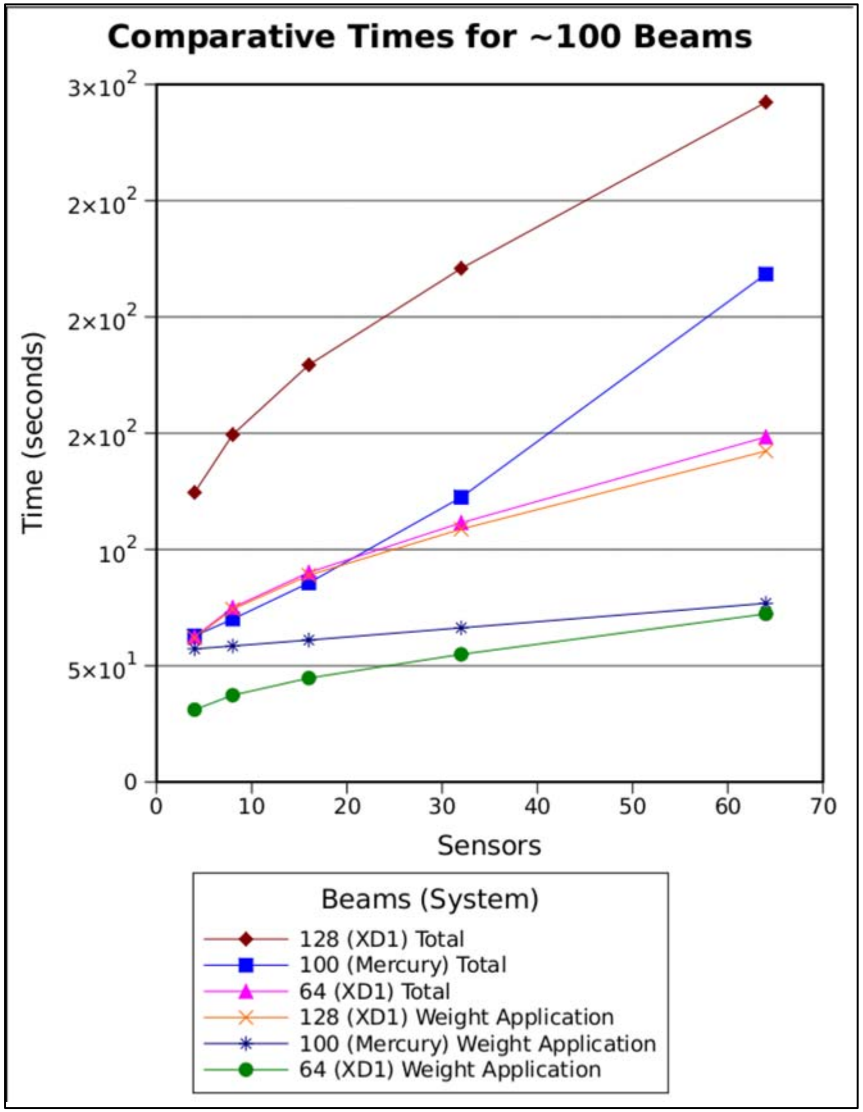
Implementation Comparison

- Even with the much shorter update period, the XD1 version exhibits comparable performance
  - Cray has constant update period of 8192 time steps
  - Mercury update period varies from 256K to 16K
  - Sensors & history match



Results Comparison





Future Directions

- Support for new platforms
  - SGI RASC
  - SRC SRC-7
  - GPUs, DSPs, CELL SPEs
- Move beyond master-slave model of processing and communication
  - FPGA to FPGA communication not currently implemented
- Implement more complex processing kernels, applications
- Improve performance
  - Identified possible mechanisms to remove the extra data copy

Conclusions

- VForce provides a framework for implementing high performance applications on heterogeneous processors
  - Code is portable
  - Support for a wide variety of hardware platforms
  - VSIPL++ programmer can take advantage of new architectures without changing application programs
  - Small overhead in many cases
  - Unlocks SPP performance improvements in VSIPL++ environment

May Vforce be with you

May Vforce be with you

May Vforce be with you

May Vforce be with you

Contact: mel@coe.neu.edu

VForce:

http://www.ece.neu.edu/groups/rcl/project/vsipl/vsipl.html