IBM T. J. Watson Research

Accelerating Stream Joins using the Cell Processor

Buğra Gedik
Philip S. Yu
Rajesh Bordawekar

Outline

- **Motivation**
- **Cell overview**
- **Stream joins overview**
- **Design choices**
  - Join program structure
  - Column oriented memory
  - Unit blocks and SIMD
- **PPE-side operation**
  - Dynamic window partitioning
  - Batch tuple processing
  - Asynchronous result handling
- **SPE-side operation**
  - Optimal basic window size
  - Taking advantage of SIMD
  - Optimizing the join code
- **Hash-based Equi-joins and M-way joins**
- **Experimental Results**
- **Conclusions**

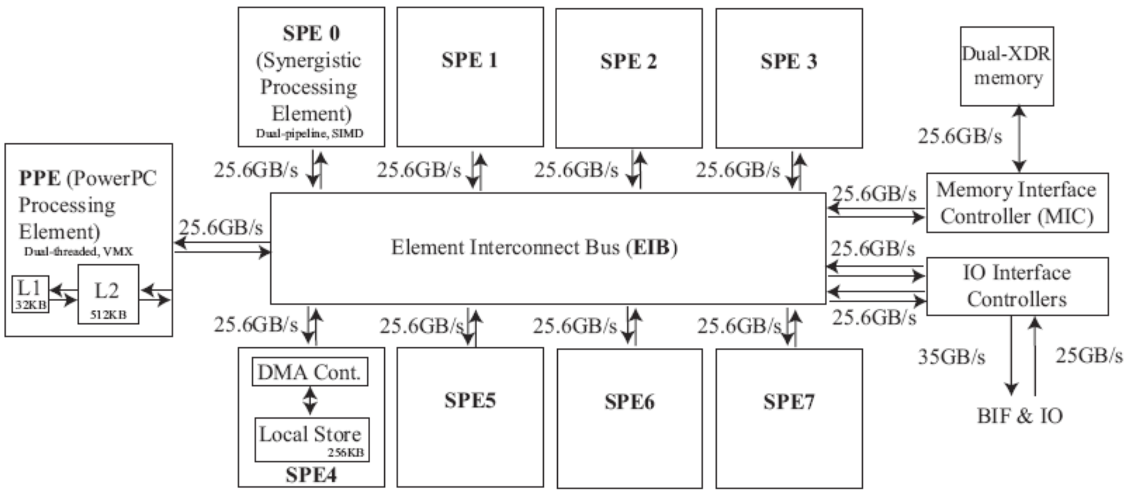
Motivation

- **Key requirements of DSMSs**
  - *Low-latency* and *high-throughput* processing
- **Multi-core processor architectures**
  - Provide high aggregate processing capacity
  - Perfect match for executing costly DSMS operators
- **STI Cell processor**
  - Heterogeneous multi-core architectures, plenty of parallelism
- **Windowed stream joins**
  - Fundamental and costly operations in DSMSs
    - Need low latency and high-throughput processing
  - Representative of a broader class of stream applications
    - Search / correlate dynamic views on streams

Cell Processor Overview

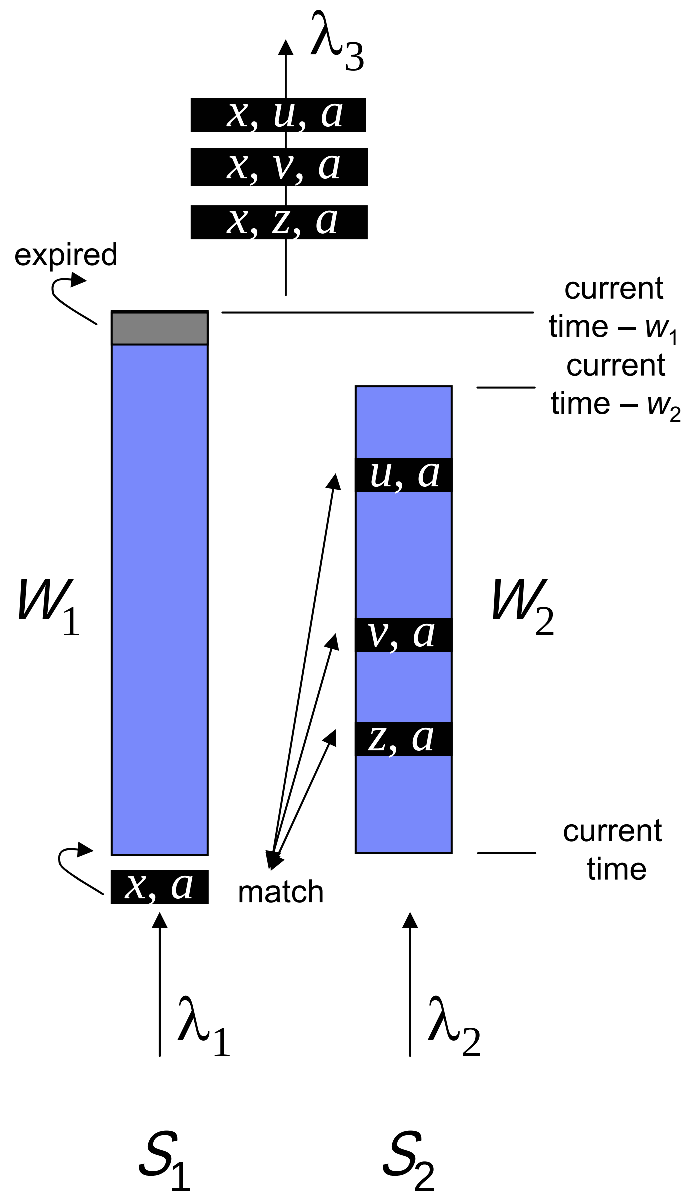
- **Heterogeneous multi-core architecture**
  - 1 Power Processing Element (PPE) for control tasks
  - 8 Synergistic Processing Elements (SPE) for data intensive processing
  - High-bandwidth Element Interconnect Bus (EIB)
- **Each SPE has**
  - Synergistic Processor Unit
  - 256 KB Local Store
  - Memory Flow Controller
- **Lots of parallelism!**
  - Asynchronous DMAs
  - 128-bit SIMD per SPE
  - Two-way ILP per SPE

No branch prediction hardware on the SPUs



Stream Join Overview

- **User-defined windows over unbounded streams**
  - Time-based windows
  - Count-based windows
- **Tuple insertion/expiration**
- **Variable stream rates**
- $\boldsymbol{S_1 \bowtie W_2}$ **and** $\boldsymbol{S_2 \bowtie W_1}$
- **Different join conditions**
  - Ex: Band joins
    - $X_l \leq S_1.A - S_2.B \leq X_u$
- ***NLJ* processing, extensions to hash-joins**

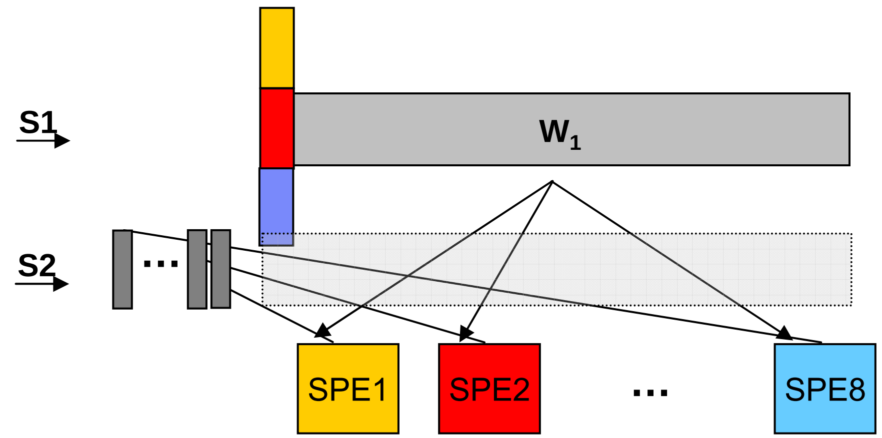


Outline

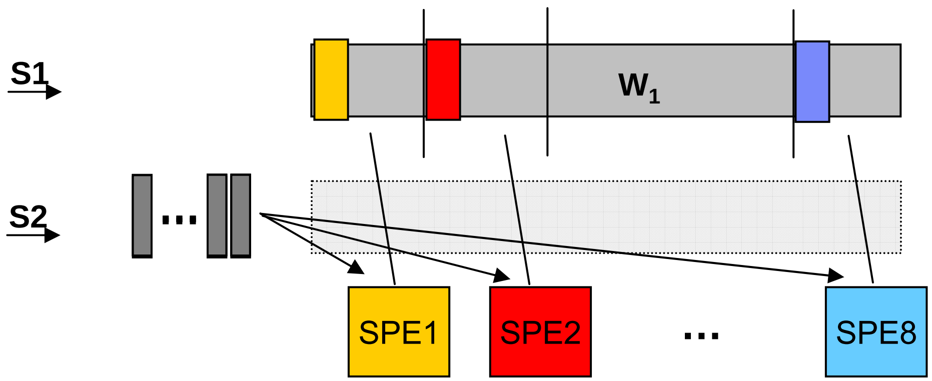
- **Motivation**
- **Cell overview**
- **Stream joins overview**
- **Design choices**
  - Join program structure
  - Column oriented memory
  - Unit blocks and SIMD
- **PPE-side operation**
  - Dynamic window partitioning
  - Batch tuple processing
  - Asynchronous result handling
- **SPE-side operation**
  - Optimal basic window size
  - Taking advantage of SIMD
  - Optimizing the join code
- **Hash-based Equi-joins and M-way joins**
- **Experimental Results**
- **Conclusions**

Parallelizing the join

- **Store join windows in main memory, not in local stores**
  - Arbitrary join window sizes
  - Transparent to # of SPEs
- **Replication vs. Distribution**
  - Replicate tuples, distribute window
  - Distribute tuples, replicate window
- **Disadvantages of Option *1***
  - ~8 x memory bw. consumption
    - At maximum throughput, we consume 3.35 GB/sec
    - 3.35 * 8 = 26.8 GB/sec vs. 25.6 GB/sec available
  - Average tuple delay



Option 1: replicate window, distribute tuples

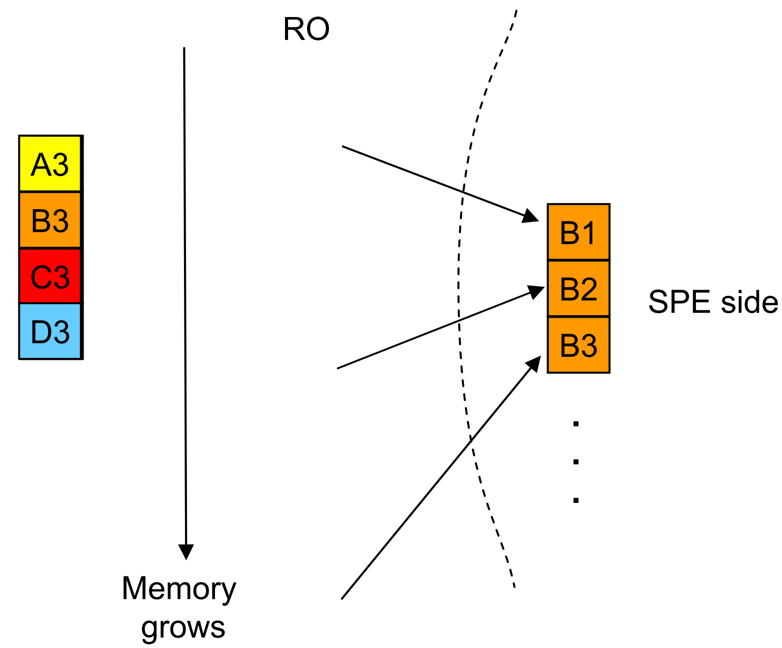


Option 2: replicate tuple, distribute window

Column-oriented memory

- **Memory organization for storing tuples in the join windows**
  - row-oriented (tuple-oriented) / array of structures
  - column-oriented (attribute-oriented) / structure of arrays

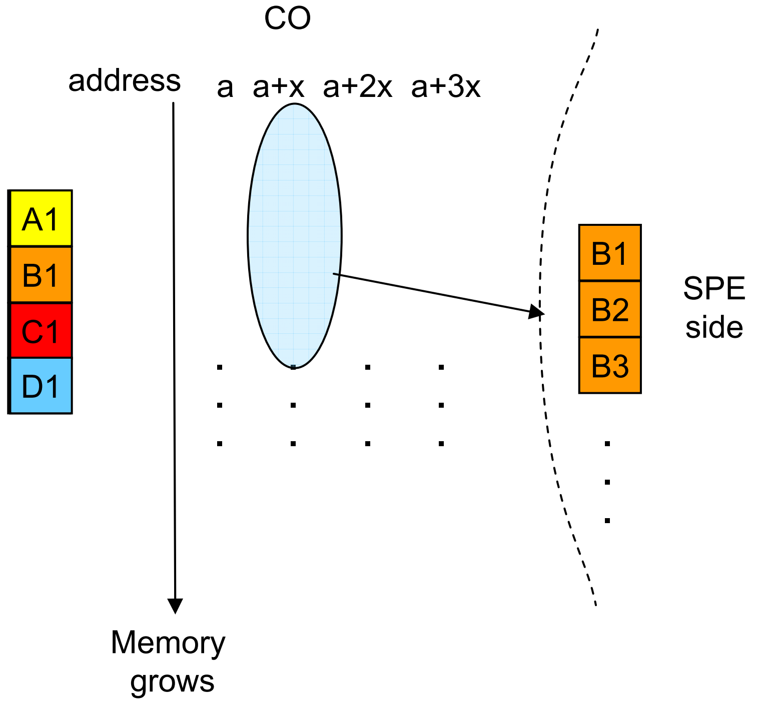
- **With column-oriented**
  - Easily transfer only the join attributes
  - No need for shuffling to take advantage of SIMD
  - No need for scatter/gather transfers (DMA-list commands)



Column-oriented memory

- **Memory organization for storing tuples in the join windows**
  - row-oriented (tuple-oriented) / array of structures
  - column-oriented (attribute-oriented) / structure of arrays

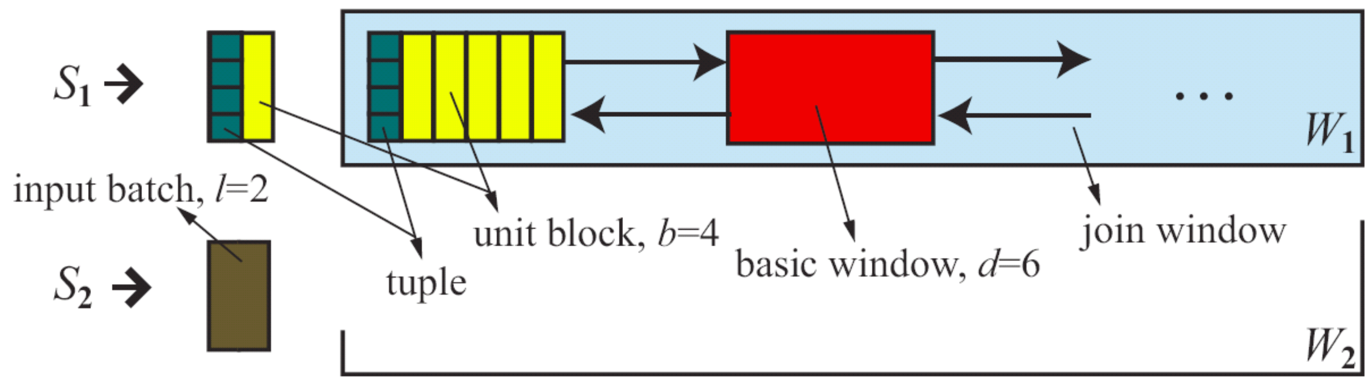
- **With column-oriented**
  - Easily transfer only the join attributes
  - No need for shuffling to take advantage of SIMD
  - No need for scatter/gather transfers (DMA-list commands)



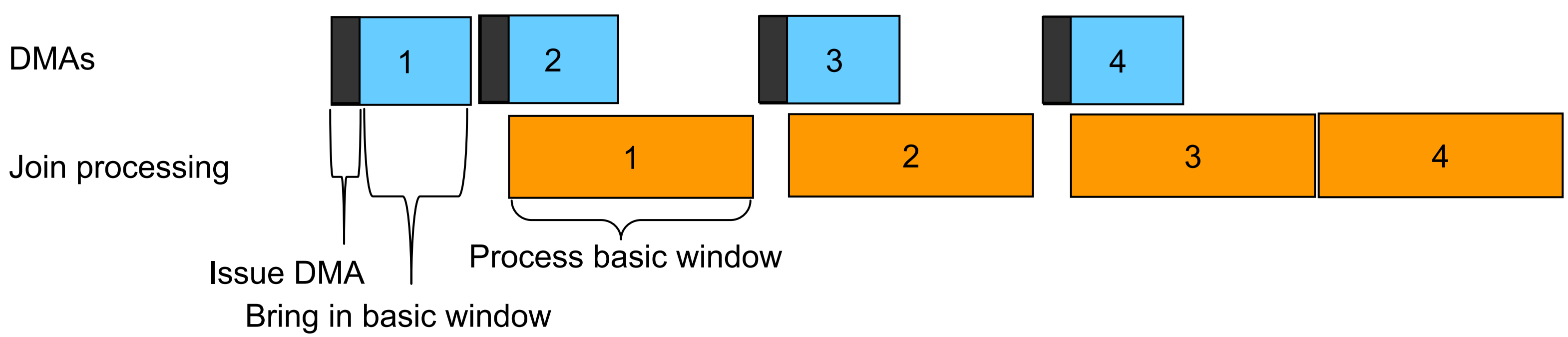
Outline

- **Motivation**
- **Cell overview**
- **Stream joins overview**
- **Design choices**
  - Join program structure
  - Column oriented memory
  - Unit blocks and SIMD
- **PPE-side operation**
  - Dynamic window partitioning
  - Batch tuple processing
  - Asynchronous result handling
- **SPE-side operation**
  - Optimal basic window size
  - Taking advantage of SIMD
  - Optimizing the join code
- **Hash-based Equi-joins and M-way joins**
- **Experimental Results**
- **Conclusions**

Join Window Organization

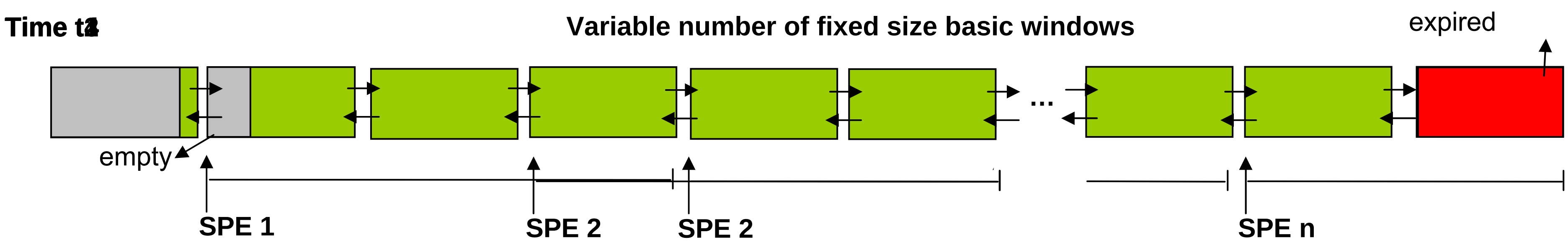


- ***Basic Window***: **Single unit of memory transfer for an SPE**
- **Used to hide memory transfer delays (double buffering)**



Dynamic window partitioning

- **Since join windows are sliding we need to maintain a dynamic partitioning over the join windows**
- **Assign consecutive basic windows to each SPE**

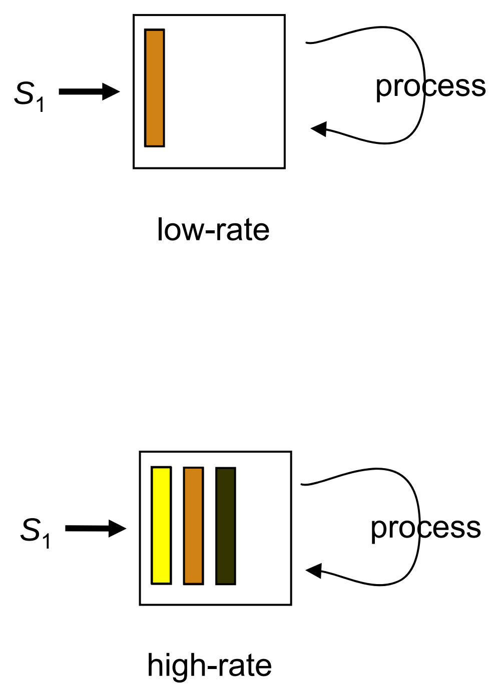


- **Use pointer shifting to update the partitioning in *O*(# of SPEs) time, independent of the # of basic windows**
- **Needed only when there is a basic window to insert (this first window is full) or remove (the last basic window is expired)**

Batch tuple processing

- **Batching is effective**
  - Transfer overheads are incurred once per batch
- **Average tuple delay components**
  - Time waited on the batch
  - Time to process a batch / batch size
- **Batch size trade-off**
  - Low stream rates -> small batches
  - High stream rates -> large batches
- **Our solution**
  - Rate-aware, dynamic tuple batching

Dynamic tuple batching



Result handling

- **Result handling includes**
  - Conversion of matching tuples into output tuples
    - SPEs only know about join attributes
    - PPE post-processes the results
- **Do result and join processing in parallel**
    - SPEs accumulate results in their buffers (2 per SPE)
    - SPEs notify the PPE when their buffer is full
      - continue join processing with an alternate buffer
    - Upon an SPE notification, PPE dispatches the job of fetching and processing result entries to a result thread
      - main PPE thread continues to service other SPEs

Outline

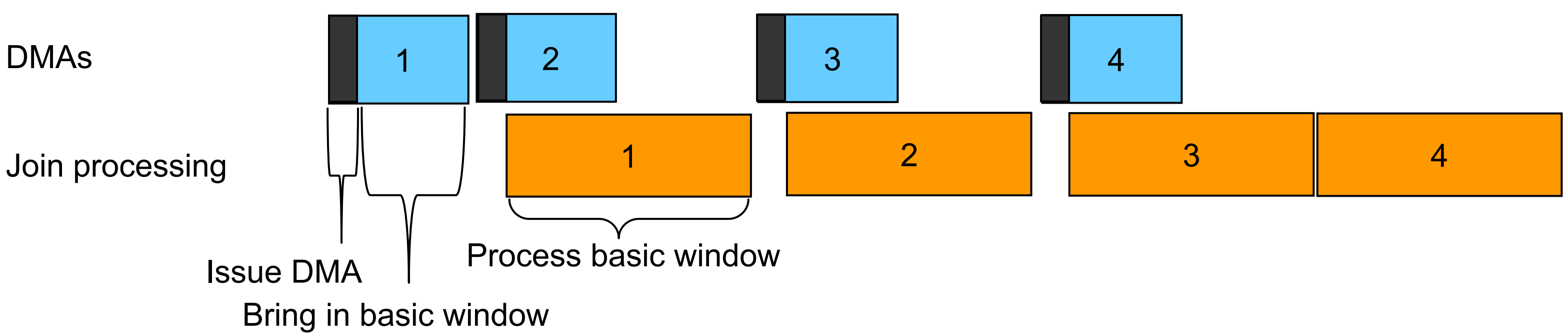
- **Motivation**
- **Cell overview**
- **Stream joins overview**
- **Design choices**
  - Join program structure
  - Column oriented memory
  - Unit blocks and SIMD
- **PPE-side operation**
  - Dynamic window partitioning
  - Batch tuple processing
  - Asynchronous result handling
- **SPE-side operation**
  - Join processing
  - Taking advantage of SIMD
  - Optimizing the join code
- **Optimal basic window size**
- **Hash-based Equi-joins and M-way joins**
- **Experimental Results**
- **Conclusions**

Join processing and asynchronous DMAs

- **Memory transfer delays with double buffering**
  - α_m: time to issue DMA commands
  - $f_m(X)$: time to bring in a basic window of size X
    - For simplicity assume linear: $\beta_m * X$
  - $f_n(X)$: time to process a basic window of size X
    - $f_n(X) = \alpha_n + \beta_n * X$ (linear function)
  - Total delay: $(\alpha_m * L + \beta_m * H) + L * (\alpha_n + \beta_n * H)$
    - L: # of basic windows, H: basic window size
- **The optimal value of *d* (basic window size)**

$$d \propto \sqrt{\frac{\alpha_m + \alpha_n}{\beta_m} \cdot \lambda \cdot w}$$

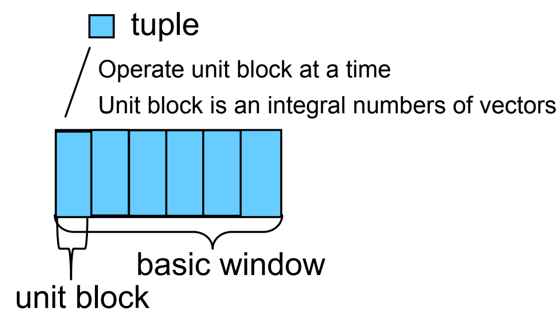
Larger rates & join window lengths require larger basic windows: max DMA size 16KB



Taking advantage of SIMD

- **Single-instruction, multiple-data**
  - Can operate on 128-bits at once, either in the form of 16 bytes, 8 shorts, 4 ints/floats, or 2 doubles/longs
- **For a band join, we can use 5 SIMD instructions to evaluate one join condition for 4 tuples**
  - Compared to 16 instructions required without SIMD

initially $a = a_1$,	$v_b = <b_1, b_2, b_3, b_4>$
SIMD replicate a,	$v_a = <a_1, a_1, a_1, a_1>$
SIMD subtract v_a from v_b	$v_d = <\cdots, a_1 - b_i, \cdots>$
SIMD compare v_d with X_l	$v_l = <\cdots, (a_1 - b_i) \geq X_l, \cdots>$
SIMD compare X_u with v_d	$v_u = <\cdots, X_u \geq (a_1 - b_i), \cdots>$
SIMD and v_l and v_u	$v_r = <\cdots, X_u \geq (a_1 - b_i) \geq X_l, \cdots>$



Optimizing the join code

- **CPI: Cycles per instruction**
  - SPEs have dual pipelines (instruction-level parallelism)
    - Independent instructions in a sequential code can be executed in parallel using the two pipelines
    - Each pipeline executes only certain types of instructions
  - Best CPI that can be reached: 0.5
- **Do loop unrolling to operate on a larger number of vectors within the body of the innermost NLJ loop**
  - Help the compiler to execute loads/stores in parallel with some of the comparison operators in the join core
  - No benefit from loop unrolling after all registers are used up (128x 128-bit registers on the SPE side)

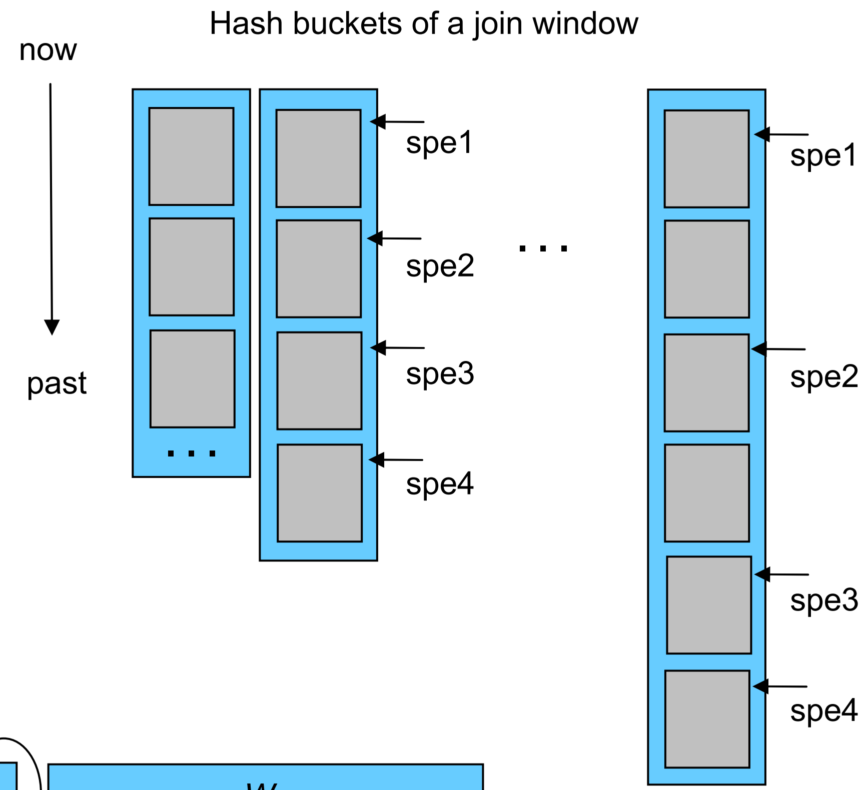
Outline

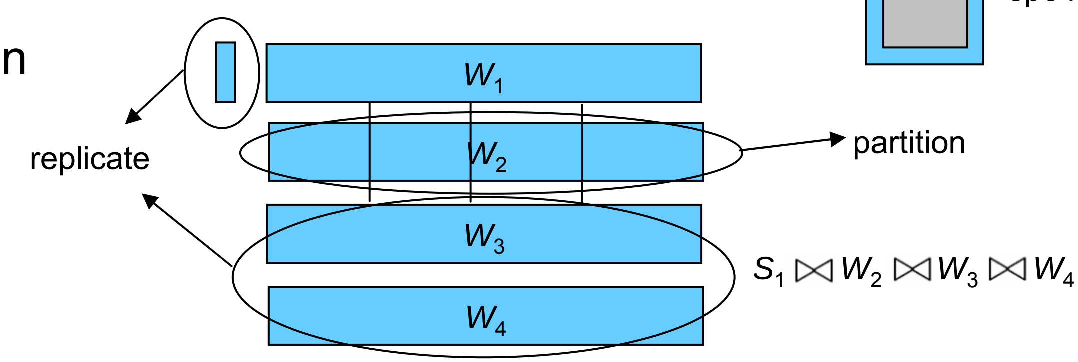
- **Motivation**
- **Cell overview**
- **Stream joins overview**
- **Design choices**
  - Join program structure
  - Column oriented memory
  - Unit blocks and SIMD
- **PPE-side operation**
  - Dynamic window partitioning
  - Batch tuple processing
  - Asynchronous result handling
- **SPE-side operation**
  - Optimal basic window size
  - Taking advantage of SIMD
  - Optimizing the join code
- **Hash-based Equi-joins and M-way joins**
- **Experimental Results**
- **Conclusions**

Hash-based Equi-joins and M-way Joins

- **Hash-based equi-joins**
  - Use hash buckets
    - Each bucked is organized like our previous join windows
    - Maintain partitioning for each bucket
  - Dynamic window partitioning
    - Same time complexity
    - Increased space complexity

- **M-way joins**
  - Replicate the tuple
  - Partition the first join window in the current join order
  - Replicate the remaining windows of the join order





Outline

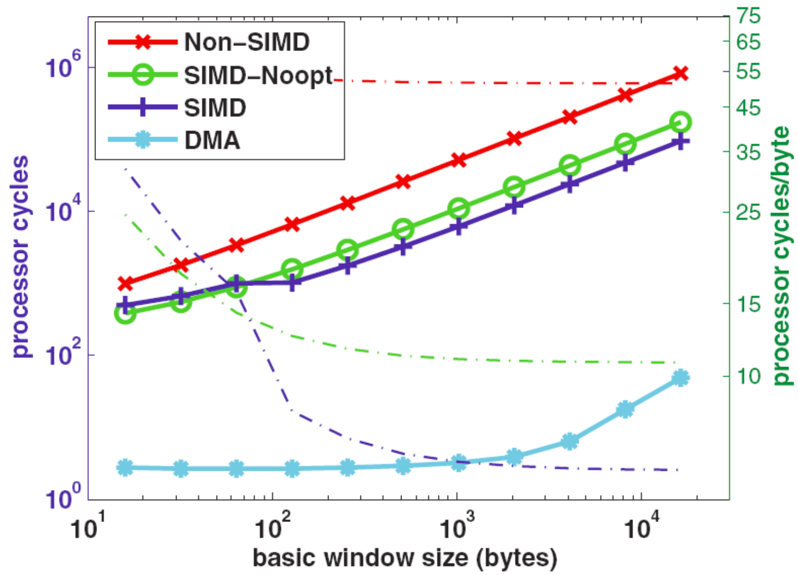
- **Motivation**
- **Cell overview**
- **Stream joins overview**
- **Design choices**
  - Join program structure
  - Column oriented memory
  - Unit blocks and SIMD
- **PPE-side operation**
  - Dynamic window partitioning
  - Batch tuple processing
  - Asynchronous result handling
- **SPE-side operation**
  - Optimal basic window size
  - Taking advantage of SIMD
  - Optimizing the join code
- **Hash-based Equi-joins and M-way joins**
- **Experimental Results**
- **Conclusions**

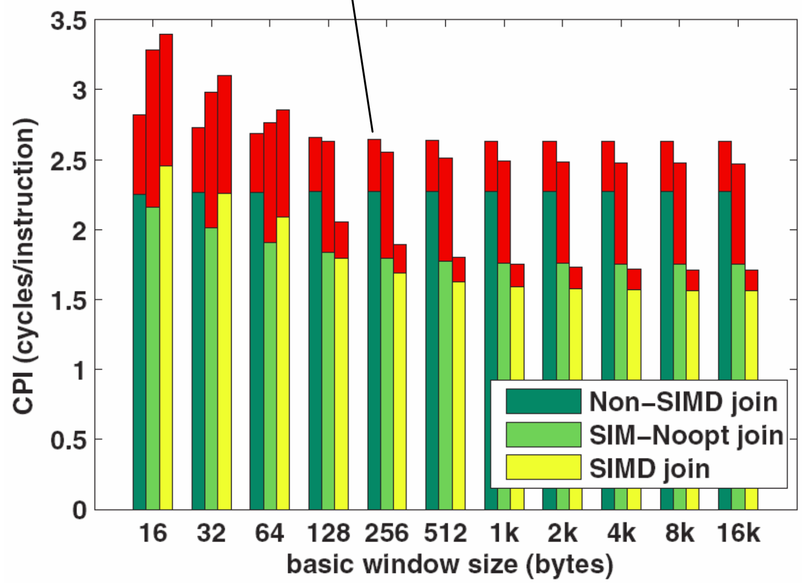
Experimental Setup

- **Two platforms are used**
  - IBM Full-scale Cell System Simulator
    - Used to measure join performance on a single SPE
    - Band join on a single int attribute
    - Measures: cycles/byte, cycles/instruction
  - 3.2 GHz IBM Cell Processor
    - Used to measure overall performance
    - Band join on two attributes: one float, one int
    - Measures: output rate, drop rate, avg. tuple processing time
- **Compared approaches:**
    - Non-SIMD, SIMD-NoOpt, SIMD using 1-8 SPEs
    - Conventional windowed stream join on Intel Xeon 3.4Ghz (no SSE optimizations applied)

Experimental Results I

Effective CPI: noops are excluded
Smaller CPI is better





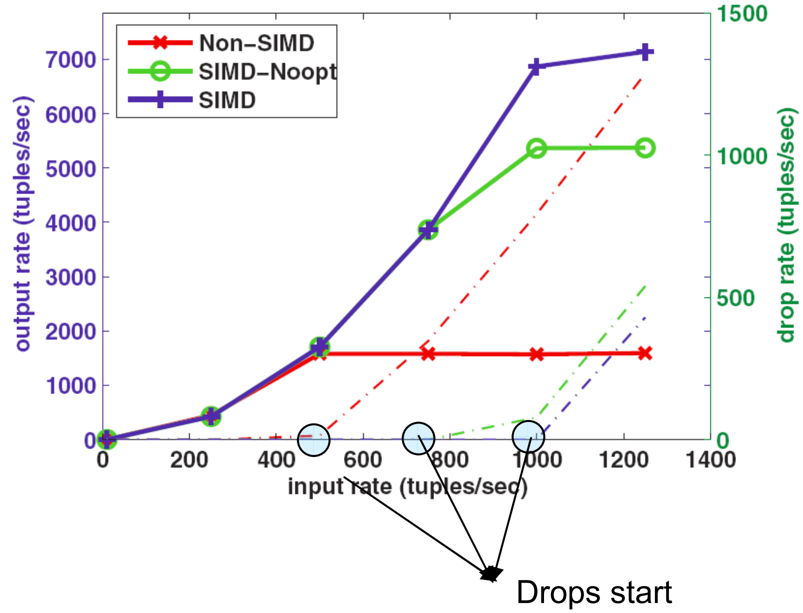
Non-SIMD: 10 times # of cycles of SIMD
SIMD-Noopt: 1.8 times # of cycles of SIMD

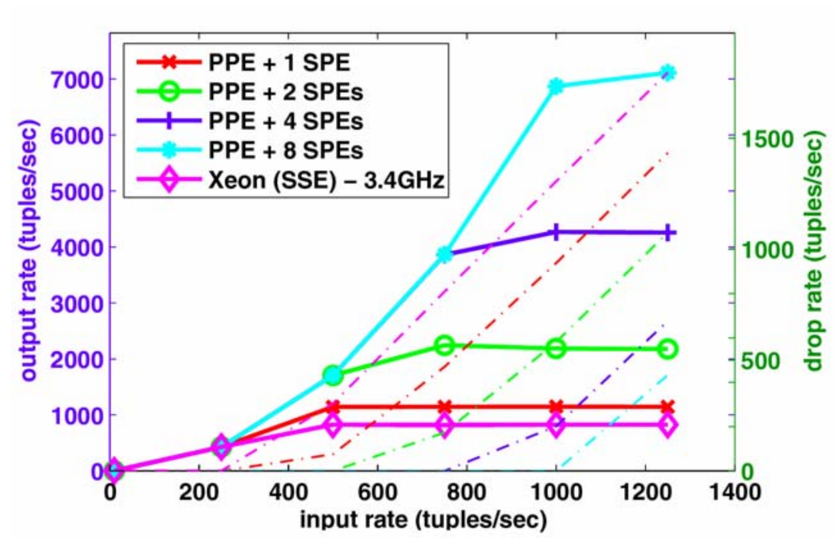
Non-SIMD: 54% higher effective CPI
SIMD-Noopt: 45% higher effective CPI

Far from optimal value of 0.5
Due to branchy nature of join

Experimental Results II

15 minutes join windows, 8KB basic windows, no batching



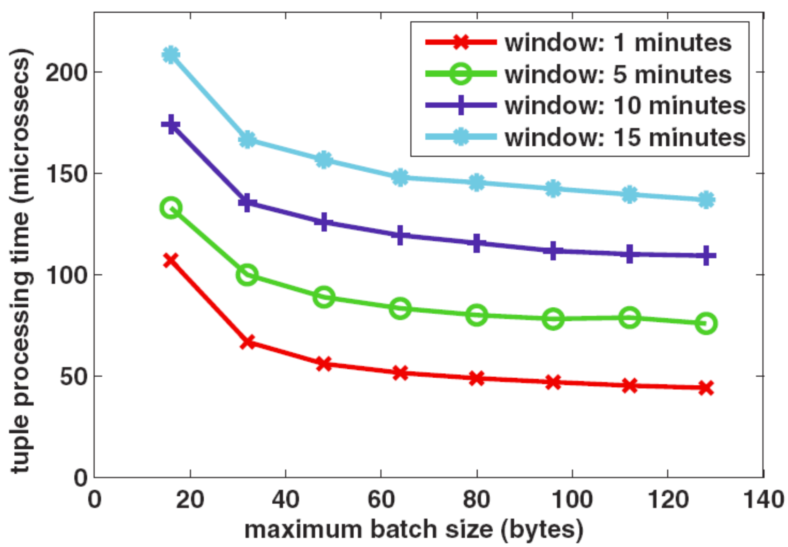


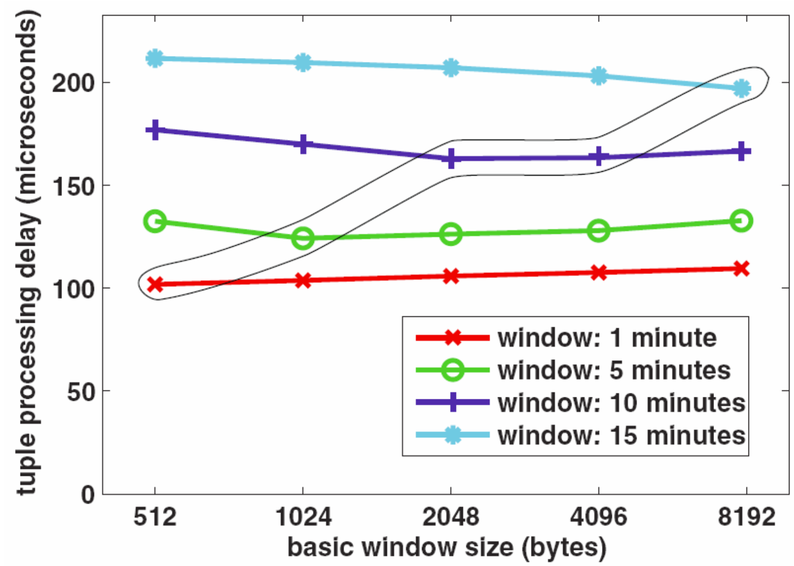
SIMD: Drop starts at 500 tuples/sec
Output rate 250% higher than Non-SIMD
33% higher than Non-SIMD
SIMD-Noopt: Drop starts at 750 tuples/sec
Non-SIMD: Drop starts at 1000 tuples/sec

Linear scalability with number of SPEs
8.6 x higher output rate compared to Xeon 3.4Ghz

At 1000tuples/sec, the join window processing rate is 13.4 GB/sec

Experimental Results III





Batching can cut down processing time by as much as 50%
Increasing batch size bring diminishing returns

Using a suboptimal basic window size can cause up to 10% increase in tuple processing time

Conclusions

- **Developed concepts and techniques to execute stream joins on heterogeneous multicore processor architectures in a scalable manner**
- **Concepts such as**
  - column-oriented memory organization, and
  - dynamic window partitioning

  enable us to better exploit multicore parallelism
- **Techniques such as**
  - delay-optimized double buffering,
  - rate-aware dynamic batch processing,
  - SIMD-optimized join code

  together lead to high throughput and low latency processing
- **Experimental results show**
  - up to 30 times better performance compared to an Intel Xeon 3.4Ghz processor
  - perfect scalability (linear) with the number of SPEs used
  - zero drop rate up to combined input rate of 2000 tuples/sec with 15 minutes join windows, resulting in a join window processing rate of 13.4 GB/sec

Questions

- **Thank You!**