FPGA-based Multithreading for In-Memory Hash Joins

Robert J. Halstead,

Ildar Absalyamov,

Walid A. Najjar,

Vassilis J. Tsotras

University of California, Riverside

Outline

- Background
  - What are FPGAs
  - Multithreaded Architectures & Memory Masking
- Case Study: In-memory Hash Join
  - FPGA Implementation
  - Software Implementation
- Experimental results

What are FPGAs?

- Reprogrammable Fabric
- Build custom application-specific circuits
  - E.g. Join, Aggregation, etc.
- Load different circuits onto the same FPGA chip

- Highly parallel by nature
  - Designs are capable of managing thousands of threads concurrently

Memory Masking

- Multithreaded architectures
  - Issue memory request & stall the thread
  - Fast context switching
  - Resume thread on memory response

- Multithreading is an alternative to caching
  - Not a general purpose solution
    - Requires highly parallel applications
    - Good for irregular operations (i.e. hashing, graphs, etc.)
  - Some database operations could benefit from multithreading

- SPARC processors, and GPUs offer limited multithreading
- FPGAs can offer full multithreading

Case Study: In-Memory Hash Join

- Relational Join
  - Crucial to any OLAP workload
- Hash Join is faster than Sort-Merge join on multicore CPUs [2]
- Typically FPGAs implement Sort-Merge join [3]

- Building a hash table is non-trivial for FPGAs
- Store data on FPGA [4]
  - Fast memory accesses, but small size (few MBs)
- Store data in memory
  - Larger size, but longer memory accesses

- **We propose the first end-to-end in memory Hash Join implementation with a FPGAs**

[2] Balkesen, C. et al. Main-memory Hash Joins on Multi-core CPUs: Tuning to the underlying hardware. *ICDE'2013*
[3] Casper, J. et al. Hardware Acceleration of Database Operations. FPGA'2014
[4] Halstead, R. et al. Accelerating Join Operation for Relational Databases with FPGAs. FPGA'2013

FPGA Implementation

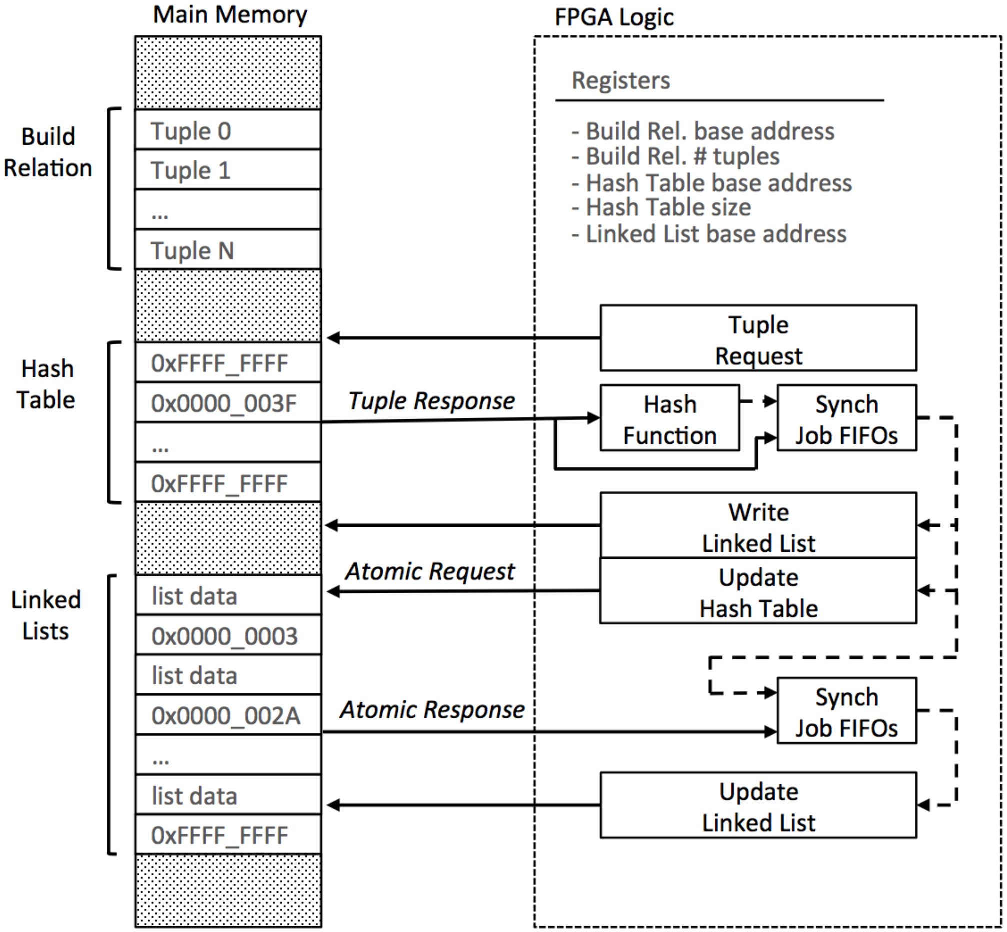
- All data structures are maintained in memory
  - Relations, Hash Table, and the Linked Lists
  - Separate chaining with linked list for conflict resolution

- An FPGA engine is a digital circuit
  - Separate Engines for the Build & Probe Phase
  - Reads tuples, and updates hash table and linked list
  - Handle multiple tuples concurrently
  - Engines operate independent of each other
  - Many engines can be placed on a single FPGA chip

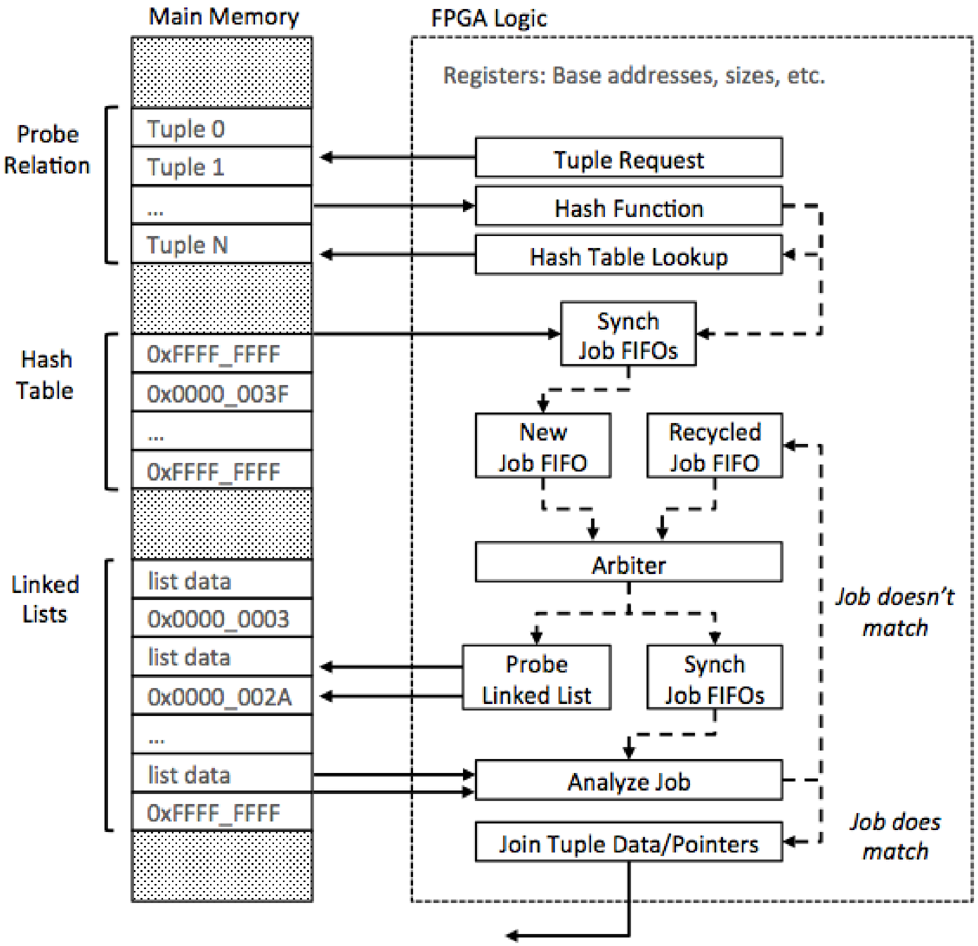
FPGA Implementation: Build Phase Engine

- Every cycle a new tuple enters the FPGA engine

- Every tuple in R is treated as a unique thread:
  - Fetch tuple from memory
  - Calculate hash value
  - Create new linked list node
  - Update Hash Table
    - Has to be synchronized via atomic operations
  - Insert the new node into the linked list



FPGA Implementation: Probe Phase Engine



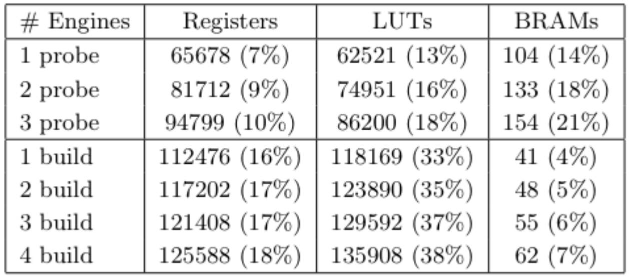
- Every tuple in R is treated as a unique thread:
  - Fetch tuple from memory
  - Calculate hash value
  - Probe Hash Table for linked list head pointer
    - Drop tuples if the hash table location is empty
  - Search linked list for a match
    - Recycle threads through the data-path until they reach the last node
  - Tuples with matches are joined
- Stalls can be issued between New & Recycled Jobs

FPGA Area & Memory Channel Constraints

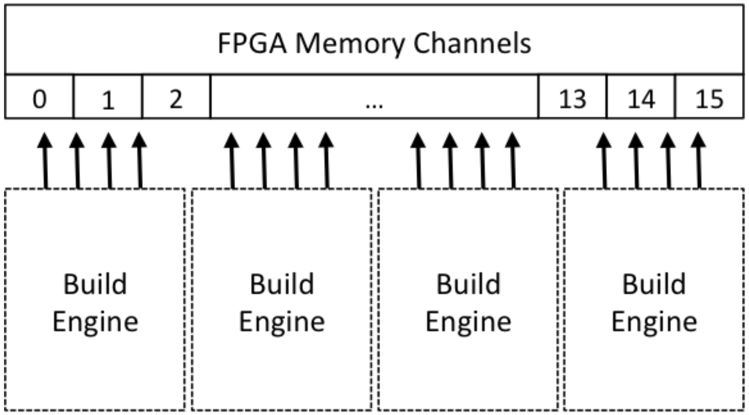
- Target platform: Convey-MX
  - Xilinx Virtex 6 760 FPGAs
  - 4 FPGAs
  - 16 Memory channels per FPGA

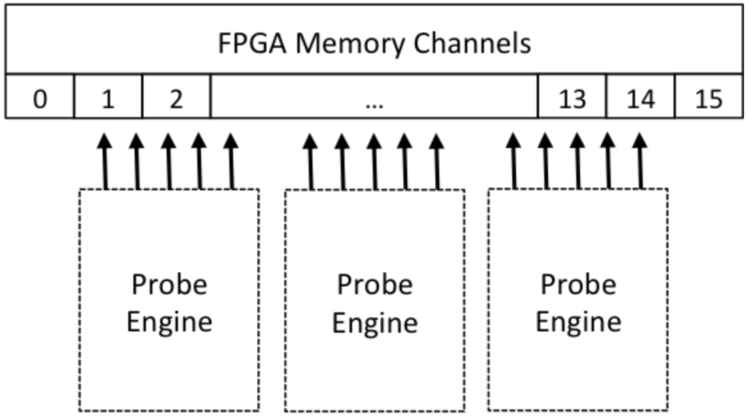
- Build engines need 4 channels
- Probe engines need 5 channels

- Designs are memory channel limited



# Engines	Registers	LUTs	BRAMs
1 probe	65678 (7%)	62521 (13%)	104 (14%)
2 probe	81712 (9%)	74951 (16%)	133 (18%)
3 probe	94799 (10%)	86200 (18%)	154 (21%)
1 build	112476 (16%)	118169 (33%)	41 (4%)
2 build	117202 (17%)	123890 (35%)	48 (5%)
3 build	121408 (17%)	129592 (37%)	55 (6%)
4 build	125588 (18%)	135908 (38%)	62 (7%)





Software Implementation

- Existing state-of-the art multi-core software implementation was used [5].
- Hardware-oblivious approach
  - Relies on hyper-threading to mask memory & thread synchronization latency
  - Does not require any architecture configuration
- Hardware-conscious approach
  - Performs preliminary Radix partitioning step
  - Parameterized by L2 & TLB cache size (to determine number of partitions & fan-out of partitioning algorithm)
- Data format, commonly used in column stores – two 4-byte wide columns:
  - Integer join key
  - Random payload value

R

Key	Payload
r_1	…
..	…
r_n	…

$\bowtie_{R.key=S.key}$

S

Key	Payload
s_1	…
..	…
s_m	…

[5] Balkesen, C. et al. Main-memory Hash Joins on Multi-core CPUs: Tuning to the underlying hardware. *ICDE'2013*

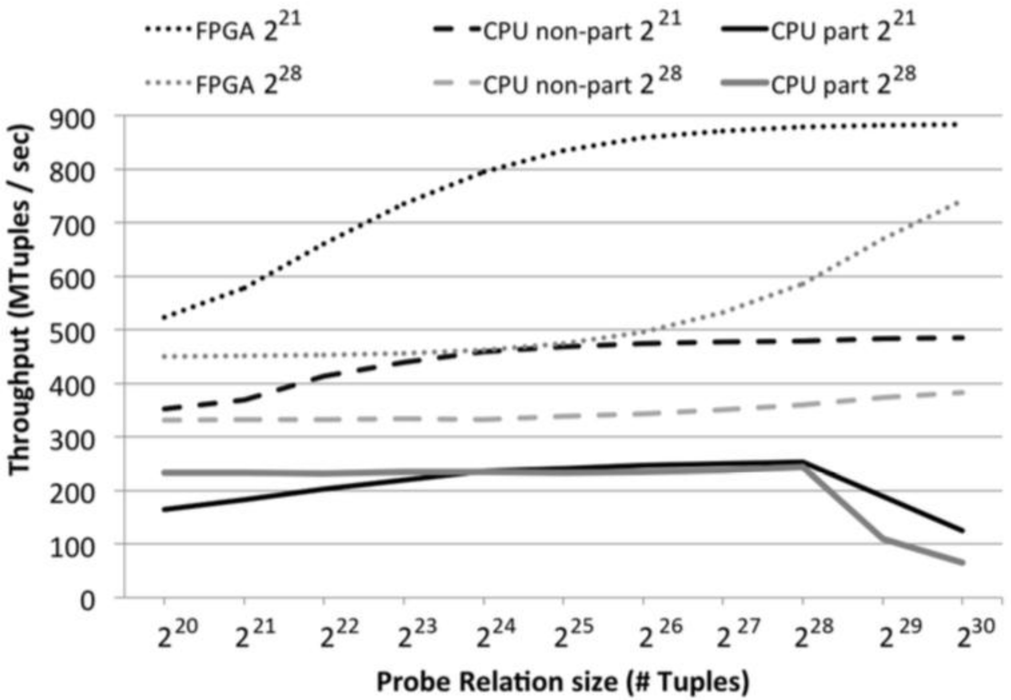
Experimental Evaluation

- Four synthetically generated datasets with varied key distribution
  - Unique: Shuffled sequentially increasing values (no-repeats)
  - Random: Uniformly distributed random values (few-repeats)
  - Zipf: Skewed values with skew factor 0.5 and 1
- Each dataset has a set of relation pairs (R&S) ranging from 1M to 1B tuples
- Results were obtained on Convey-MX heterogeneous platform

Hardware Region	
FPGA board	Virtex-6 760
# FPGAs	4
Clock Freq.	150 MHz
Engines per FPGA	4 / 3
Memory Channels	32
Memory Bandwidth (total)	76.8 GB/s

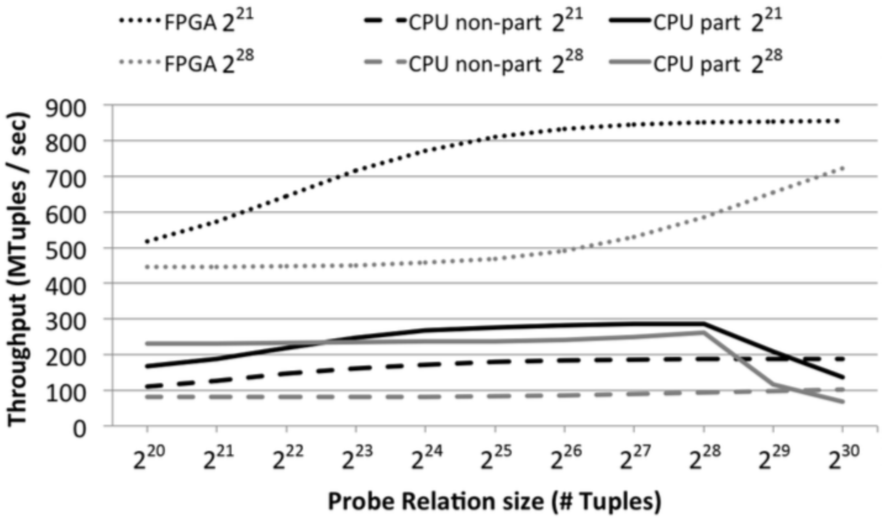
Software Region	
CPU	Intel Xeon E5-2643
# CPUs	2
Cores / Threads	4 / 8
Clock Freq.	3.3 GHz
L3 Cache	10 MB
Memory Bandwidth (total)	102.4 GB/s

Throughput Results: Unique dataset

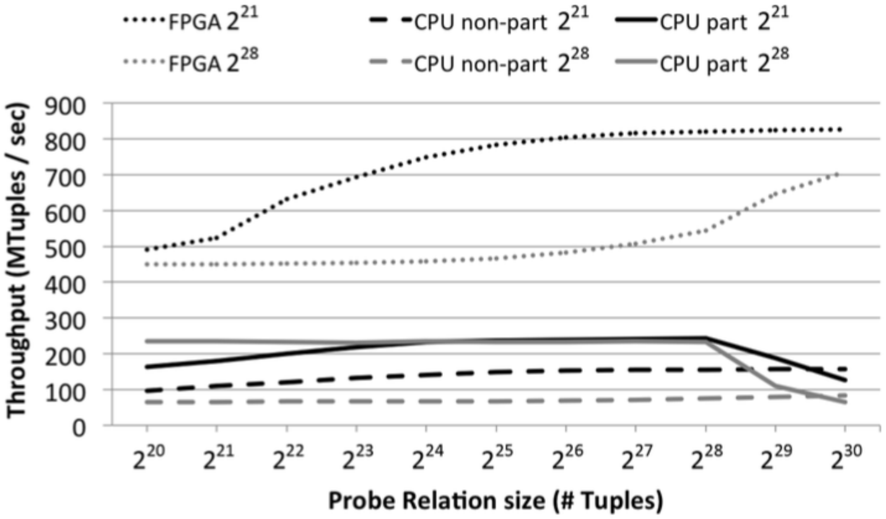


- 1 CPU (51.2 GB/s)
  - Non-partitioned CPU approach is better than partitioned one, since each bucket has exactly one linked list node
- 2 FPGAs (38.4 GB/s)
  - 900 Mtuples/s when Probe Phase dominated
  - 450 Mtuples/s when Build Phase dominated
  - 2x Speedup over CPU

Throughput Results: Random & Zipf_0.5



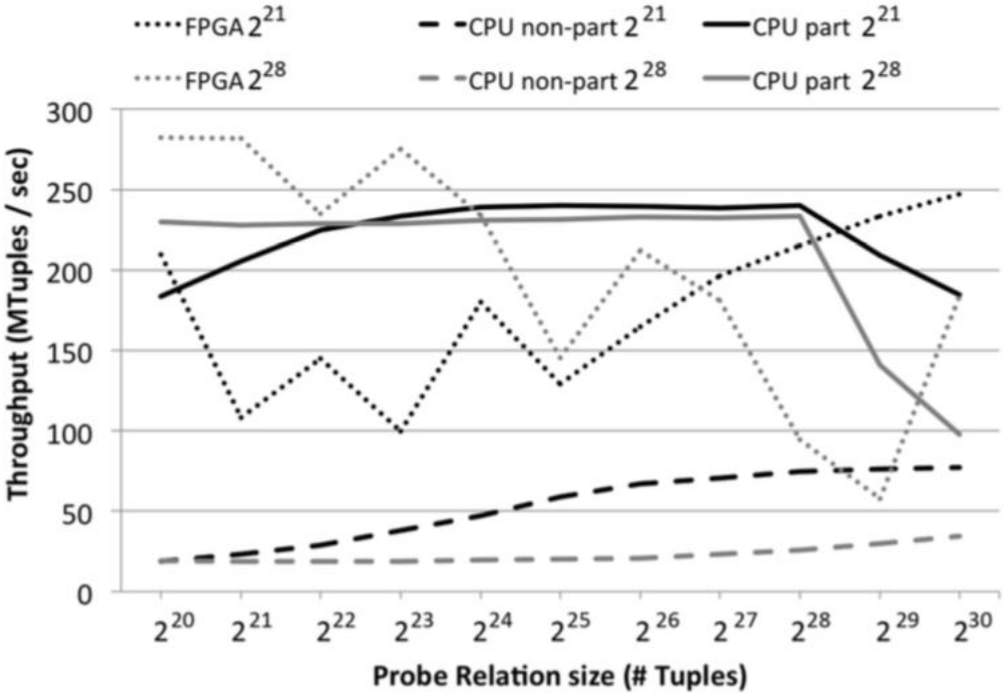
(b) *Random* dataset



(c) *Zipf_0.5* dataset

- As the average chain length grows from one non-partitioned CPU solution is outperformed by partitioned one
- FPGA has similar throughput, speedup ~3.4x

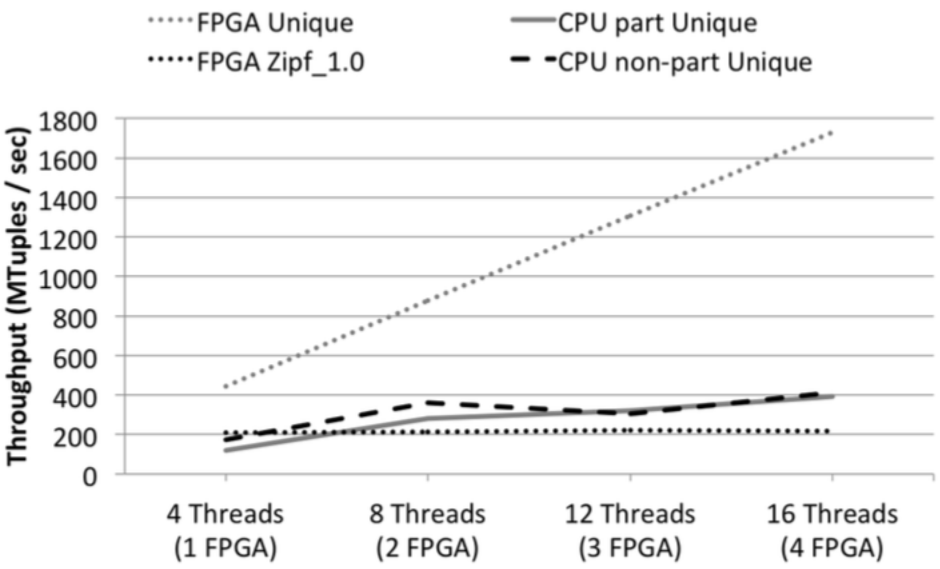
Throughput Results: Zipf_1.0 dataset



- FPGA throughput decreases significantly due to stalling during probe phase

Scale up Results: Probe-dominated

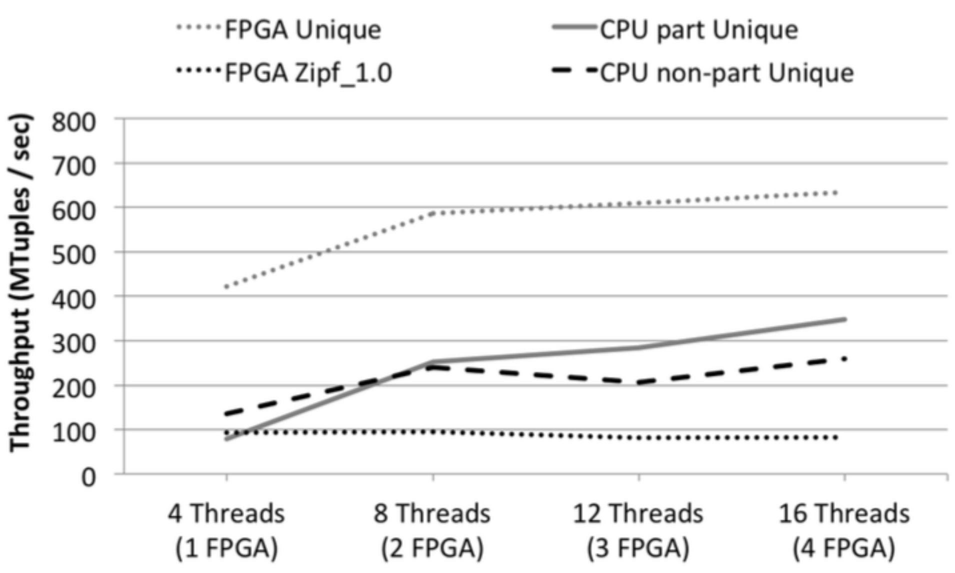
- Scale up: each 4 CPU threads are compared to 1 FPGA (roughly matches memory bandwidth)
- Only Unique dataset is shown, Random & Zipf_0.5 behave similarly
- FPGA does not scale on Zipf_1.0 data



(a) Build Relation has 2^{21}, Probe has 2^{28} tuples

- Partitioned CPU solution scales up, but at much lower rate than FPGA

Scale up Results: |R|=|S|



(b) Build and Probe Relations both have 2^{28} tuples

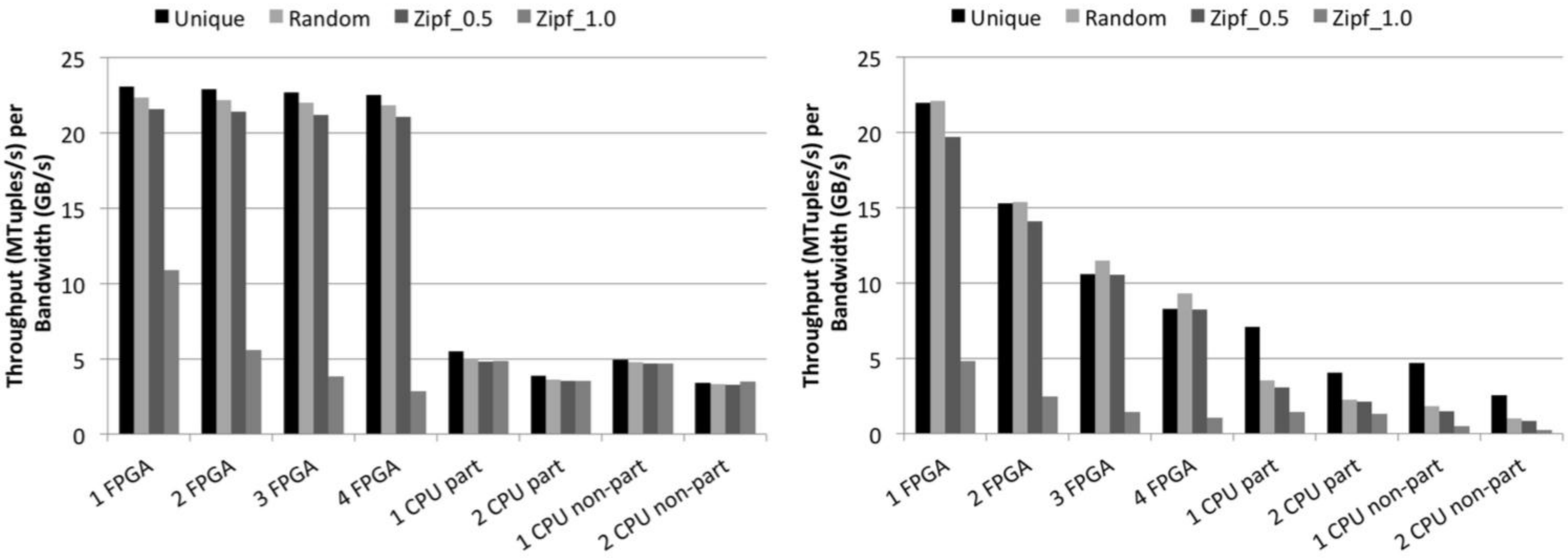
- FPGA does not scale better than partitioned CPU, but it is still ~2 times faster

Conclusions

- Present first end-to-end in-memory Hash join implementation on FPGAs

- Show memory masking can be a viable alternative to caching
  - FPGA multi-threading can achieve 2x to 3.4x over CPUs
  - Not reasonable for heavily skewed datasets (e.g. Zipf 1.0)

Normalized throughput comparison



(a) Build Relation has 2^{21}, Probe has 2^{28} tuples

(b) Build and Probe Relations both have 2^{28} tuples

- Hash join is memory-bounded problem
- Convey-MX platform gives advantage to multicore solutions in terms of memory bandwidth
- Normalized comparison shown that FPGA approach achieves speedup up to 6x (Unique) and 10x (Random & Zipf_0.5)