LINQits: Big Data on Little Clients

Eric S. Chung, John D. Davis, Jaewon Lee

Presentation by Tie Chen and Parker Hill
Motivation

- Many emerging applications:
  - Continuous speech recognition
  - Interactive personal agents
  - Augmented reality

- High computational requirements:
  - Bad for battery life
  - Can exceed device’s abilities

- Hard to accelerate:
  - Expensive design process
  - Difficult for application programmers
Goal

• Build a framework that:
  ○ Seamlessly works with current software
  ○ Can accelerate a substantial fraction of the application
  ○ Provides order of magnitude speed up / energy efficiency improvements
  ○ Has enough flexibility to work on many applications
Language-Integrated Query (LINQ)

- Many common patterns on collections
  - `int[] a=new int[]{1,2,3,4}
  - `a.Select(x=>x*x) → [1,4,9,16]
  - `a.GroupBy(value=>value%2, (key,group)=>group.Max()) → [3,4]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Where</strong></td>
<td>(Filter) Keep all values satisfying a given property.</td>
</tr>
<tr>
<td><strong>Select</strong></td>
<td>(Map) Apply a transformation to each value in the collection.</td>
</tr>
<tr>
<td><strong>Aggregate</strong></td>
<td>(Fold, Reduce) Combine all values in the collection to produce a single result (e.g., max).</td>
</tr>
<tr>
<td><strong>GroupBy</strong></td>
<td>Create a collection of collections, where the elements in each inner collection all have a common property (key).</td>
</tr>
<tr>
<td><strong>SelectMany</strong></td>
<td>Generates a collection for each element in the input (by applying a function), then concatenates the resulting collections.</td>
</tr>
<tr>
<td><strong>Join</strong></td>
<td>Combine the values from two collections when they have a common property.</td>
</tr>
</tbody>
</table>
LINQits Overview

- LINQ → build graph → translate to HW → add to binary
- Everything else → compile normally
LINQits Design

- Input data → pre-core → organize → post-core → output
  - a. GroupBy(value=>value%2, (key, group)=>group.Max())
Challenging Expressions — GroupBy

- Very little locality
  - Not enough on-chip memory for all groups
  - High DRAM latency

![GroupBy Diagram]

<table>
<thead>
<tr>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>Keys</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

`Key = f(Val)`
GroupBy — Partitioning

- Make multiple passes → reduce number of groups per pass

Partition(Key) = Key % 2

GroupBy with 1 level of partitioning
Partitioning Hardware

\[ qid = (\text{key} \gg \text{pass #} \times \log_2(\text{numQ})) \mod \text{numQ} \]
GroupBy — Example

- GroupBy without partitioning

Input (Memory) - 0 loads

Spill (Memory) - 0 stores

Queue

Queue

Queue

Queue
GroupBy — Example

- 16 unique keys

Input (Memory) - 0 loads

Spill (Memory) - 0 stores

Queue

Queue

Queue

Queue
GroupBy — Example

- Use first free queue

Input (Memory) - 1 load

Queue
- 1

Spill (Memory) - 0 stores

Queue

Queue

Queue
GroupBy — Example

- Use remaining free queues

Input (Memory) - 4 loads

Spill (Memory) - 0 stores

Queue

Queue

Queue

Queue
GroupBy — Example

- No more queues → spill to memory

Input (Memory) - 5 loads

| 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 |

Spill (Memory) - 1 store

| 5 |

Queue 1

Queue 2

Queue 3

Queue 4
GroupBy — Example

Input (Memory) - 16 loads

Spill (Memory) - 12 stores

Queue
1
Queue
2
Queue
3
Queue
4
GroupBy — Example

- Make another pass

Input (Memory) - 16 loads

Spill (Memory) - 12 stores

Queue

Queue

Queue

Queue
GroupBy — Example

- 2 more passes later...

Input (Memory) - 36 loads

Spill (Memory) - 24 stores

Queue
Queue
Queue
Queue
GroupBy — Example

- Done (64 memory operations)

Input (Memory) - 40 loads

Spill (Memory) - 24 stores
GroupBy — Example

- Now with partitioning

  Partitions (Memory) - 0 stores
  Queue
  Queue
  Queue
  Queue

  Input (Memory) - 0 loads
  Queue
  Queue
  Queue
  Queue

  Spill (Memory) - 0 stores
  Queue
  Queue
  Queue
  Queue
GroupBy — Example

- Partition = key % 4
GroupBy — Example

- Queue on partition

Parititions (Memory) - 0 stores

Input (Memory) - 4 loads

Spill (Memory) - 0 stores

Queue

Queue

Queue

Queue
GroupBy — Example

- Same partition → same queue

Partitions (Memory) - 0 stores

Input (Memory) - 5 loads

Spill (Memory) - 0 stores

Queue

- 5
- 1

Queue

- 2

Queue

- 3

Queue

- 4
GroupBy — Example

Partitions (Memory) - 0 stores

Input (Memory) - 16 loads

Spill (Memory) - 0 stores

Queues:
1. Queue: 13, 9, 5, 1
2. Queue: 14, 10, 6, 2
3. Queue: 15, 11, 7, 3
4. Queue: 16, 12, 8, 4
GroupBy — Example

- Partitions sent back to memory

Partitions (Memory) - 16 stores

Input (Memory) - 16 loads

Spill (Memory) - 0 stores

Queue

Queue

Queue

Queue
GroupBy — Example

- Bring first partition back to HW

Partitions (Memory) - 16 stores

Input (Memory) - 16 loads

Spill (Memory) - 0 stores
GroupBy — Example

- Queue on key

![Diagram showing GroupBy example with partitions, input, and spill memories]

Partitions (Memory) - 16 stores

Input (Memory) - 20 loads

Spill (Memory) - 0 stores

Queue

1

5

9

13
GroupBy — Example

- Bring the next partition...

Partitions (Memory) - 16 stores

Input (Memory) - 20 loads

Spill (Memory) - 0 stores
GroupBy — Example

- Continue...

Partitions (Memory) - 16 stores

Input (Memory) - 26 loads

Spill (Memory) - 0 stores

Queue

Queue

Queue

Queue
GroupBy — Example

- Done (48 memory ops vs 64)

Partitions (Memory) - 16 stores

Input (Memory) - 32 loads

Spill (Memory) - 0 stores
GroupBy — Value of Partitioning

<table>
<thead>
<tr>
<th></th>
<th>No Partitioning</th>
<th>1-pass Partition</th>
<th>2-pass Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01% Unique Keys</td>
<td>2739118</td>
<td>2000000</td>
<td>3000000</td>
</tr>
<tr>
<td>0.1% Unique Keys</td>
<td>8311241</td>
<td>2002989</td>
<td>3000000</td>
</tr>
<tr>
<td>1% Unique Keys</td>
<td>78238488</td>
<td>2766866</td>
<td>3001120</td>
</tr>
</tbody>
</table>
Evaluation — Experimental Setup
# Evaluation — FPGA Resources

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Operators</th>
<th>Total Area</th>
<th>Pre-Core Area</th>
<th>Post-Core Area</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GroupBy</strong></td>
<td>Partition, GroupBy</td>
<td>21 KLUTs (39%), 90 BRAMs (62%)</td>
<td>1.1 KLUTs (2%), -</td>
<td>-</td>
</tr>
<tr>
<td><strong>Join</strong></td>
<td>Partition, GroupBy, Join</td>
<td>27 KLUTs (51%), 94 BRAMs (67%)</td>
<td>2.2 KLUTs (4%), -</td>
<td>-</td>
</tr>
<tr>
<td><strong>Black-Scholes</strong></td>
<td>Select</td>
<td>29 KLUTs (54%), 54 BRAMs (34%)</td>
<td>3.9 KLUTs (7%), 55 DSPs (24%)</td>
<td>-</td>
</tr>
<tr>
<td><strong>KeyCount</strong></td>
<td>Partition, GroupByAccum</td>
<td>38 KLUTs (71%), 98 BRAMs (65%)</td>
<td>635 LUTs (1%), 747 LUTs (1%)</td>
<td>-</td>
</tr>
<tr>
<td><strong>K-means</strong></td>
<td>GroupByAccum</td>
<td>28 KLUTs (51%), 90 BRAMs (60%)</td>
<td>1.7 KLUTs (3%), 14 DSPs (6%), 1 BRAM (1%)</td>
<td>6.7KLUTs (13%), 8 DSPs (4%)</td>
</tr>
</tbody>
</table>
Evaluation — Software Platform

- LINQ and LINQits:
  - C# using Mono runtime
  - Same code for software and hardware versions

- C implementation:
  - Written without LINQ
  - Multithreaded code
Evaluation — Speedup and Energy Efficiency

- 10.7x - 38.1x speedup over multithreaded C
- 8.9x - 30.6x energy efficiency over multithreaded C
Conclusion

● LINQits seamlessly accelerates LINQ expressions

● Wide applicability due to flexibility of LINQ

● Up to 38.1x speedup + 30.6x energy efficiency
Questions
Discussion

- Do pre- and post-cores in hardware make sense? (especially for ASICs)
- Can LINQits be scaled to real applications?
- Is the mobile domain a good area for LINQits?
- Is this a good use of hardware real estate?
Partition Linked List

- **Free List**
- **Partition Meta**
- **Data Arrays**

**Partition Header**
- Key
- Next
- DataRoot
- Size

**Next Partition**

- **Data Array**
- Next

Approximately 1-4KB
Discussion

- Could LINQits be used in a multi-process situation?
- Are the constraints on code too strict?
- Are programming costs low enough?