EE482C Project

Stream Cache Architectures for Irregular Stream Applications

Timothy Knight
Arjun Singh
Jung Ho Ahn
Motivation

Irregular stream applications
- Perform a data-dependent traversal of an arbitrary graph
- Pointer chasing

Index streams
- Method for traversing irregular streams on a `traditional’ stream architecture (i.e.) Imagine
- Inefficient
  - Wasted SRF space
  - Wasted memory bandwidth

Architectural enhancements
- Stream cache
- Indexed SRF
- ...?
Stream Cache

- Amplifies memory bandwidth in presence of temporal locality of reference
- Avoids repeated memory accesses for same address
- Allowing clusters to issue memory requests through a cluster cache avoids replicating data in the SRF.
Applications Studied

Application 1

- All updates committed after all computations done
- Pseudo code:

```plaintext
// Computation phase
for each vertex v:
    v.newdata = kernel (v.data, v.neigh.data)

// Update phase
for each vertex v:
    v.data = v.newdata
```
Applications Studied (Cont.)

Application 2

- Updates committed as soon as computed
- Conceptually an advancing wave-front of computation in a DAG
- Chosen to create coherence issues
- Pseudo code:

```plaintext
repeat until all vertices are updated:
    for each vertex v with valid predecessors:
        v.data = kernel (v.neigh.data)
```
Architectures Modeled

1. No cache (baseline)
   - 16 clusters
   - 8 banked memory
2. Cache accessible from SRF
3. Dedicated cache accessible from clusters
4. Cache accessible from clusters and SRF

All cache models:
- Have 8 banks, matching the memory
- Have 1 word per cache line
- Are not coherent
Hardware Costs

• **Cache in memory system**
  - 8 SRAM banks, associated logic and buffers
  - Need 1 word of tag storage per address cached

• **Enabling cluster memory access**
  - Reorder buffer: 16x 2/3*-ported register files
  - Requests FIFO / AG
  - 8 new buses between memory and clusters

• **Dedicated cluster cache (additional costs)**
  - 2 full 8x8 crossbars
Simulation Environment

Implemented a ‘cycle-by-cycle performance simulator’:

- Not functionally cycle-accurate
- Modeled throughputs, latencies, and resource constraints of architectures
- Applications coded in `macrocode`
- Average sim: 5,000,000 cycles in 15 minutes of real time

Parameters:
- **Data sets**: record size, graph size, connectivity, locality
- **Apps**: kernel computation time, strip size, cache use
- **Model**: throughputs, latencies, mem. sizes, number of nodes, cache organization
Results

Speedup vs. K (app1)

- Line 1
- Line 2
- Line 3
- Line 4

Speedup

K (cycles)
Results (cont.)

**Speedup vs. K (app2)**

- **Speedup**
- **K (cycles)**

Graph showing speedup versus K (cycles) for different labels 1, 2, 3, and 4.
Results (cont.)

Speedup vs. ROB length (app2)

- Speedup
- ROB length

Graph showing speedup versus ROB length for different configurations labeled as 1, 3, and 4.
Results (cont.)

Other results observed:
- Increase of speedup with increasing degree
- Relationship between speedup and number of words of data per record
- Constant speedup with increasing number of nodes for small data sets; large data sets blow up our simulator with many nodes
- Constant speedup with increasing number of vertices
Conclusion

- A stream cache can improve performance on an irregular stream app.
  - Speedups of up to 3.5 were observed in our experiments
- A cache is most useful when the kernel performs little computation on each record.
- Various pros and cons between different cache architectures
  - The architecture which performs best is determined by the choice of application and data set
Limitations on speedup from a stream cache:
- Amdahl’s Law: only some of the memory requests exhibit temporal locality
- Bank conflicts
- Dependencies in application
- Available locality
- ‘Preprocessing’ overhead
Future Work

- Study cache performance on real applications in ISIM.
Questions?