Exploiting GPUs in Frequent Itemset Mining

Claudio Silvestri, Salvatore Orlando

Department of Environmental Sciences, Informatics and Statistics

Data-Intensive and SCalable Information Systems (DISCIS) Lab
Data Mining and Frequent Itemset Mining
The DCI algorithm
Opportunities for parallelization
gpuDCI: an overview
Different parallelization strategies
Building blocks
Some results
Conclusions
Frequent Itemset Mining
Apriori principle

If an itemset is frequent, then all of its subsets must also be frequent.

Pruned supersets

Found to be Infrequent
DCI: Direct Counting & Intersecting

- **Level-wise** (BFS) algorithm
- **Hybrid** method for determining the supports of frequent itemsets
  - **Counting-based** during early iterations
    - Effective pruning of horizontal dataset (the dataset is stored ‘per-transaction’)
  - **Intersection-based** when database fits into the main memory ⇒ resource-aware
    - Horizontal-to-Vertical transformation (the dataset is stored ‘per-item’)

---

*Università Ca’ Foscari Venezia*
DCI: counting-based phase

- **Direct Counting** of candidate supports
  - *Apriori*-like generation of $C_k$ from $F_{k-1}$
  - counting based on a directly accessible data structures to efficiently access counters associated with candidates
  - efficient for short patterns only

- **Transaction pruning**
  - A pruned database $D_k$ is written to the disk at each iteration
  - Fewer and shorter transactions entail less computation to perform
When the pruned database fits into the main memory, DCI builds on-the-fly an *in-core bit-vector vertical dataset*. Due to the effectiveness of dataset pruning, this usually occurs at early iterations (2nd or 3rd iter).
DCI: intersection-based phase

- Still level-wise behavior
  - $C_k$ generated from $F_{k-1}$ (but without pruning)
- Candidate supports is computed on-the-fly, by intersecting the bit-vectors that corresponds to items in the candidate
  - $C_k$ no longer needs to be stored in-core
  - Vertical dataset kept in-core
- k-way bitvector intersection, with caching of partial intersections
  - exploiting efficient bitwise machine operations
Cache: Tidlist Intersection

- **k-way intersections**
  - intersect tidlists associated with single items
  - low memory requirements, but too many intersections!

- **2-way intersections**
  - start from tidlists associated with frequent \((k-1)\)-itemsets
  - huge memory requirements, but less intersections!

- DCI $\Rightarrow$ tradeoff between 2-way and k-way
  - is based upon k-way intersections of bitvectors,
  - BUT caches all the partial intersections corresponding to the various prefixes of the current candidate itemset

**Cache size:** \(k-2\) bitvectors of \(n_k\) bits
Cache: Tidlist Intersection

Buffer of \((k-2)\) vectors of \(n_k\) bits used for caching intermediate intersection results

Current candidate

Reuse of this cached intersection
Cache: No. of AND operations
Opportunities for parallelization

- **Direct count**: small part of total running time, random memory access

- **Candidate generation**: negligible cost

- **Intersection**: significant cost, sequential memory access
**gpuDCI: an overview**

- Direct count phase: CPU
- As soon as possible:
  - Move the shrinked dataset to GPU global memory
  - Use CPU for candidate generation (and housekeeping)
  - Use GPU for candidate support computation
- Goals:
  - Minimize memory transfers
  - Keep cores busy
  - Ensure memory access coalescing
  - Limit global synchronizations
Parallelization strategies

- **Transaction-wise:**
  all threads work on the same intersection operation

- **Intersection-wise:**
  would create dependences due to partial result reuse

- **Candidate-wise:**
  the threads in the same block\(^*\) works on the same intersection operation. Different blocks deals with different candidates (far wrt lexicographical order)

\(^*\) Threads in the same block will be scheduled at the same time on the same multiprocessor, thus having access to the same shared memory
Transaction-wise parallelization
Candidate-wise parallelization

(a) Intersection

(b) Count
Building blocks

- **Intersection:**
  - Bit-wise AND of an array of integer
  - Read from shrinked dataset and partial results
  - Write to partial results

- **Count:**
  - Integrated with last intersection
  - Local reduction in shared memory
  - Global reduction in global memory (transaction-wise parallelization)

- **Batches of operations:**
  - Encoded in constant cache
  - Directly refers to operations and operands:
    \{operation,op1,op2,destination\}
Comparison of Strategies

- **Transaction-wise:**
  - straightforward to implement
  - count-reduction in two phases
  - requires one global synchronization for each count
  - One copy of partial result cache
  - **Full occupancy only for very large datasets**

- **Candidate-wise**
  - just local count reduction
  - batches of operations stored in constant cache
  - different blocks performs different operations (but referred to the same index in the command buffer)
  - **One copy of partial result cache per block!**
Overall running time

Note: includes CPU direct count phase
Intersection running time

Runtime vs pattern length
Dataset: accidents

- gpuDCI$_{CW}$ (30 blocks)
- gpuDCI$_{TW}$ (30 blocks)
- DCI
- # candidates

Time (s) vs Pattern length
Scalability: dataset size

Scalability
T10 dataset - ms = 5%

Time (s)

# transactions (x 10^6)

gpuDCI_{CW} (30 blocks)

gpuDCI_{TW} (30 blocks)

DCI

Arrow pointing to the left on the graph.
Scalability: thread blocks
Conclusions

- **gpuDCI**: two parallelization strategies, reuse of partial results
- Clear advantage wrt CPU-only
  - Tie in case of small number of transactions and large number of candidates (eg: kosarak dataset)
- Candidate-wise (CW) parallelization is the best choice, but require more memory for large number of transactions. However….
- …. Transaction-wise (TW) parallelization equals CW for large number of transactions
Future works

- Hybrid: multi-CPU + multi-GPU
- Port of DCI multi-strategy optimizations to GPU
- Frequent Closed itemset mining