Hardware Accelerated Apriori Algorithm for Data Mining

Zachary K. Baker and Viktor K. Prasanna
University of Southern California, Los Angeles, CA, USA
zbaker@halcyon.usc.edu, prasanna@ganges.usc.edu

Abstract

The Apriori algorithm is a popular correlation-based datamining kernel. However, it is a computationally expensive algorithm and the running times can stretch up to days for large databases, as database sizes can extend to Gigabytes.

Through the use of a new extension to the systolic array architecture, time required for processing can be significantly reduced. Our array architecture implementation on a Xilinx Virtex-II Pro 100 provides a performance improvement that can be orders of magnitude faster than the state-of-the-art software implementations. The system is easily scalable and introduces an efficient "systolic injection" method for intelligently reporting unpredictably generated mid-array results to a controller without any chance of collision or excessive stalling.

1 Introduction

Data mining is a rapidly advancing field of strategies for finding connections between elements in large databases. The Apriori algorithm is a popular and foundational member of the correlation-based data mining kernels used today. However, it is a computationally expensive algorithm and running times can stretch to days for large databases, as database sizes can reach from Gigabytes and computation requires multiple passes.

The Apriori algorithm operates by progressively building frequent sets over multiple generations. Each generation is composed of three sections: the candidate generation, candidate pruning, and candidate support steps. Each generation provides a set of candidates that is expanded in the next generation. The support information is fed back into the candidate generator and the cycle continues until the final candidate set is determined.

Candidate generation is the process in which one generation of candidates are built into the next generation. This operation is as follows:

$\forall\, f_1, f_2 \in F_k$ do
with $f_1 = (i_1, ..., i_{k-1}, i_k)$ and $f_2 = (i_1, ..., i_{k-1}, i_k^*)$
and $i_k < i_k^*$
$f := f_1 \cup f_2 = (i_1, ..., i_{k-1}, i_k, i_k^*)$

Another phase of the algorithm is the support calculation. It is by far the most time consuming and data intensive part of the hardware implementation, as it requires the database to be streamed into the system. Each potential candidate's support, or number of occurrences over the database set, is determined by comparing each candidate with each transaction in the database. If the items in a candidate are a subset of the transaction, the support count for that candidate is incremented, as follows:

$\forall t \in T$ do
$\forall c \in C$ do
if $(c \in t)$ then support(c)++

The main problem with the Apriori algorithm is this data complexity. Each candidate must be compared against every transaction data, and candidate generation must see the entire database transaction set. This gives a large running time for a single generation, $O(|T||C||t|)$, assuming the subset function can be implemented in constant time $|t|$. However, the parallelism contained in the loops allows for some interesting acceleration in hardware, particularly when implemented as a systolic array.

1.1 Our Contributions

The architecture we have designed utilizes a systolic architecture to allow for the uni-directional streaming of candidate and transaction data through the hardware accelerator device. Due to our streaming implementation, the candidate generation phase of the algorithm require orders of magnitude less time than the support calculation, and overall requires roughly 25% of the time required by the fastest non-supercomputer implementations of Bodon, Borgelt, and Goethals. The off-chip memory required is negligible beyond the size of the database and the bandwidth between memory and the systolic array is only 250MB/s.

While the architecture could easily be implemented in a custom ASIC – in fact, the simple units that make up the systolic array are designed explicitly for ease of ASIC implementation – the use of FPGA allows the user to utilize parameterized designs which allow for variable size item descriptors as well as optimized memory sizes for a particular problem. As well, FPGAs allow the design to be scaled upward easily as process technology allows for ever-larger gate counts.

The candidate generation phases are based on the availability of a method to inject results into the datastream. Injection is important in this sort of application because any or all of the items in the array can produce data at any time over consecutive cycles. Unfortunately, this is not as simple as inserting some sort of "shadow" register to provide a delay given a downstream stall, although this is part of the solution. Without some method of controlling the data, this could result in a tangle of data collisions. Consider the candidate generation phase; $(|C_m|)^2$ comparisons are made, and were each of the c_a units to produce a result (a worst case that is not actually possible) and attempt to forward it in some way, the final unit in the chain would either have c_a collisions, or we would have to find some way to buffer out c_a elements within the units. Candidate generation and pruning is an unpredictable, entirely data-controlled operation. However, we can stall the pipeline and cause the pipeline itself to act as an efficient buffer. This is possible in data mining whereas it is less attractive in applications such as string matching because we have some flexibility as to when data is sent into the array.

As the support calculation takes an order of magnitude more time than any other of the functional segments in our implementation, it is a good approximation of the overall system performance. With one large FPGA device we can beat the much faster clocked dual device Xeon machine by at least 4x, and often by far greater margins.

[1]Supported by the United States National Science Foundation under award No. ACI-0325409 and in part by an equipment grant from the Xilinx and HP Corporations.