GPU-accelerated Keyword Matching and Expression Evaluation for Real-time Text Search

Brendan Wood
MTS, Software Engineering

brendan.c.wood
@brendanwood1986
in/brendancalebwood

Background

- Salesforce Marketing Cloud provides the following:
  - Social listening
  - Social content
  - Engagement
  - Workflow and automation
  - Social advertisements
  - Measurement
- Social listening is provided by our Radian6 technology, acquired in 2011
  - We target the entire social web for monitoring and engagement

Background

- Keyword-matching
  - Central to all of our listening capabilities
  - Forms the basis of all our search functionality
- Use cases
  - Content acquisition
    - Check all incoming content for relevancy
  - Content search
    - Search content in our database for items that match a particular topic profile

Background

- A keyword is a string of characters defining a discrete word or phrase that is to be matched
- Eg., “apple”, “orange”, “justin bieber”
- For a keyword to be matched, it must be bounded by whitespace (not a substring of a longer word/phrase)
- Keyword expressions are sets of keywords logically combined with Boolean operators
- Eg., “mango” AND (“horrible” OR “delicious”)

salesforce marketing cloud

Scale of Content Acquisition

- Documents
  - Need to handle about 450M incoming tweets per day
    - And growing...
  - Also need to handle every other type of content we acquire
- Keywords
  - Over 1.6M search expressions defined in our topic profiles
    - This is growing quickly with our customer base
  - Expressions are becoming more expensive to evaluate
    - Each expression averages about 12 inclusive keywords
    - Customers are learning that more keywords can give better specificity for listening

salesforce marketing cloud™

Old Approach

- Apache Lucene
  - Java library which provides comprehensive indexing and search capabilities for Boolean keyword expressions
  - Easy to use and integrate into a tech stack
- Twitter workers
  - Build an index for a large batch of tweets, then query every expression against the index
  - Low throughput, high latency

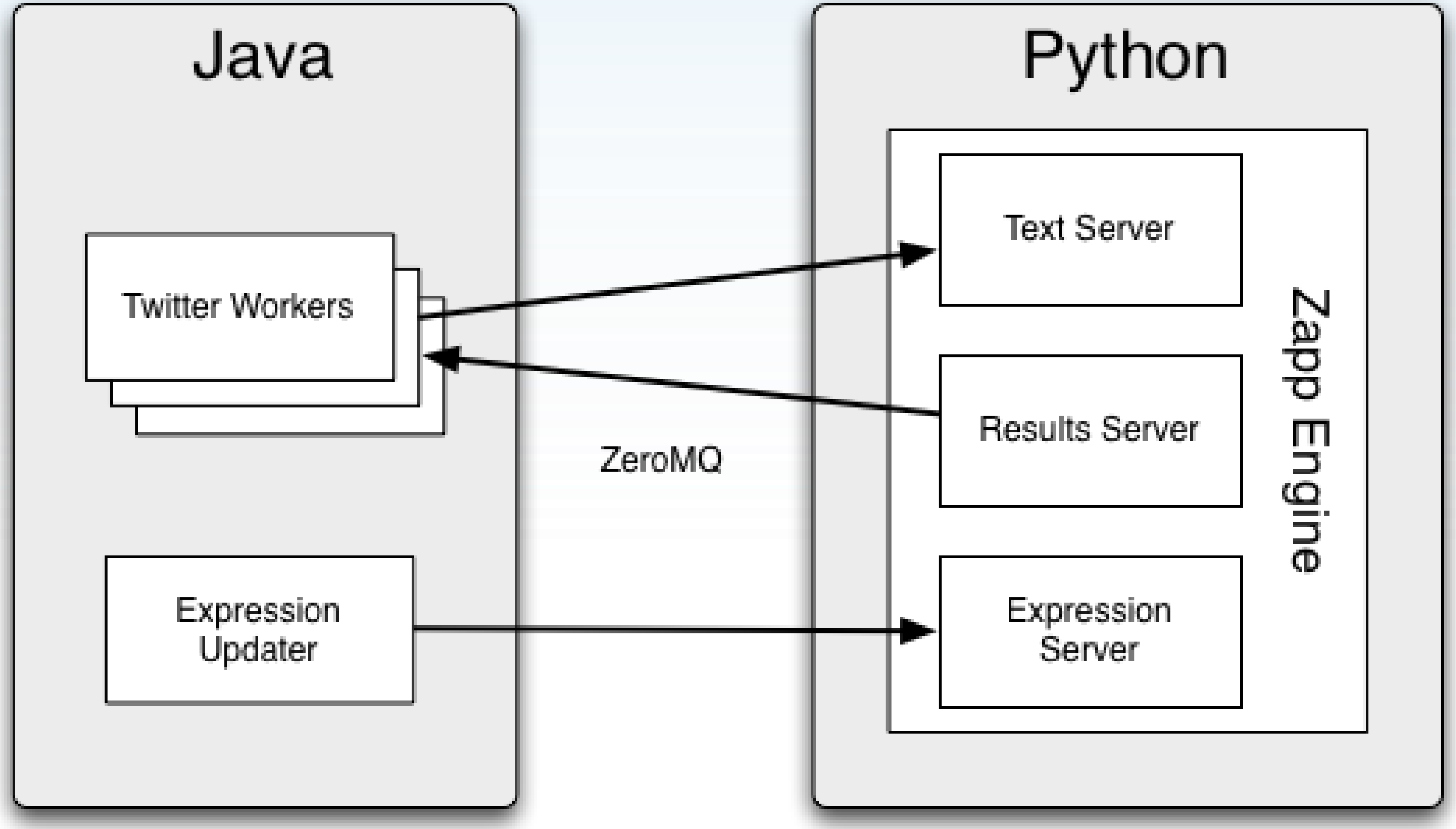
What's the problem?

- Latency
  - It takes roughly 5 minutes for a Twitter worker to process a batch of ~8k tweets against the full expression set
  - Sometimes can take up to 12 minutes
  - For Twitter content that is supposed to be real-time, some of our customers found this unacceptable
- Throughput
  - Roughly 80 multithreaded Twitter workers were needed to keep up with the Twitter Firehose of Public Tweets
  - Spikes on the Firehose caused backups in our queues, introducing more latency and requiring extra workers to remove the backlog

New Approach – Zapp

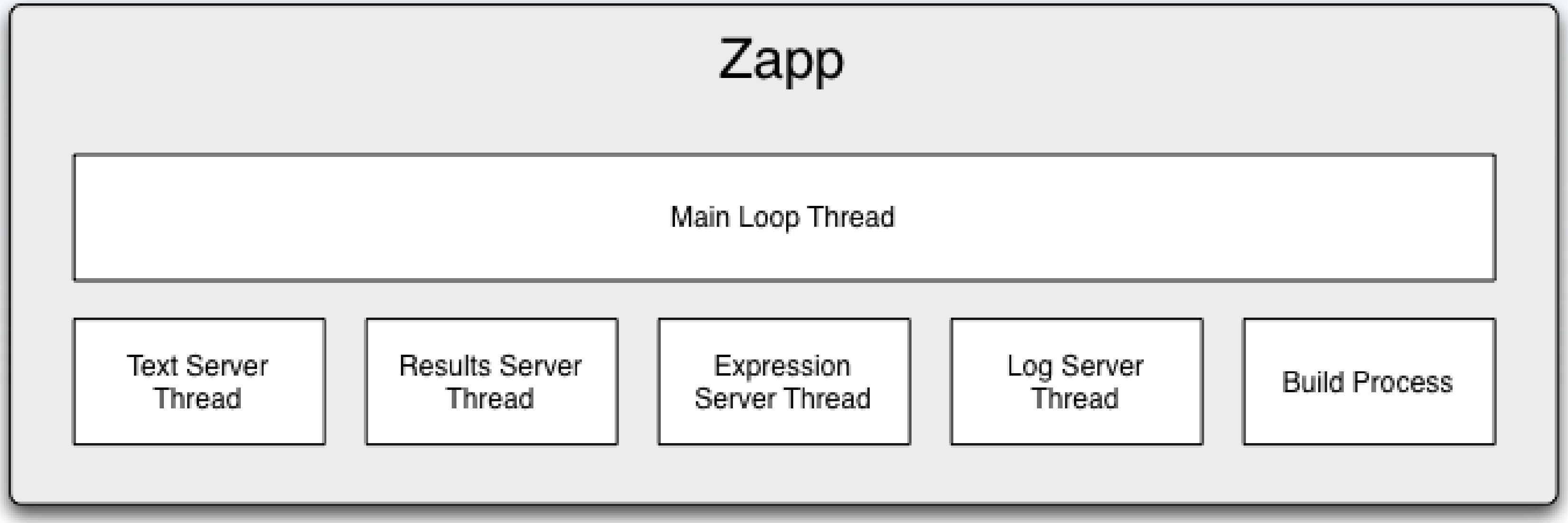
- Accelerated hardware
  - High-throughput GPUs
  - Run keyword-matching controller on CPU and push compute-intensive tasks to GPU
- Better algorithm
  - Search keywords simultaneously in parallel, then evaluate keywords against Boolean expressions
  - Prune the expression search space to reduce the number of expression evaluations

Zapp Overview



marketing cloud

Zapp Engine



salesforce marketing cloud

Algorithm Pseudo-code

```
preprocess expressions
while true:
    get document batch from text server
    for each document:                                  First CUDA kernel
        find all keyword matches
    for each document:                                  Second CUDA kernel
        find all keyword expressions which evaluate as true
    push results to results server
```

salesforce marketing cloud

Keyword-matching Kernel

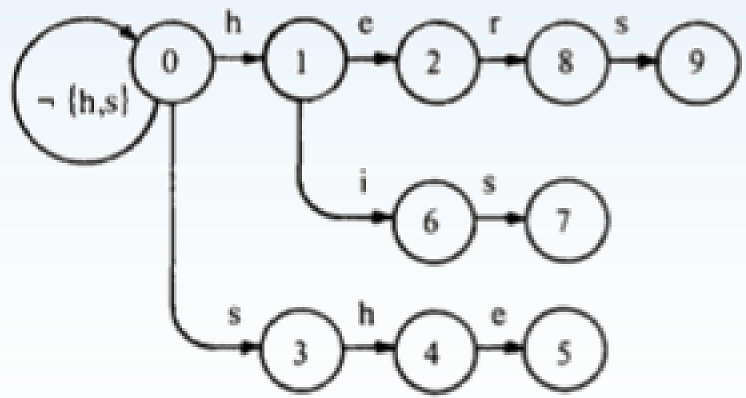
- Implements parallel Aho-Corasick
  - Goto, failure, and output functions are pre-computed on CPU
- Functions are serialized as a state transition table
- Documents enter as linear buffer with an index marking the document boundaries
- Text is divided into chunks of a fixed number of bytes
  - Each chunk maps to a CUDA thread
- Matches are stored as binary flags in a bitmap

Aho-Corasick

- Construct a goto function representing all unique keywords from all expressions
- Define failure and output functions for each node tree
- Step through a text stream, byte-by-byte, walking through the tree according to functions

Shelly had a husband. His name was Herbert, but she didn't like his cooking and he found her bothersome.

Keywords: he, she, his, hers



(a) Goto function.

i	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	0	1	2	0	3	0	3

(b) Failure function.

i	*output* (i)
2	{he}
5	{she, he}
7	{his}
9	{hers}

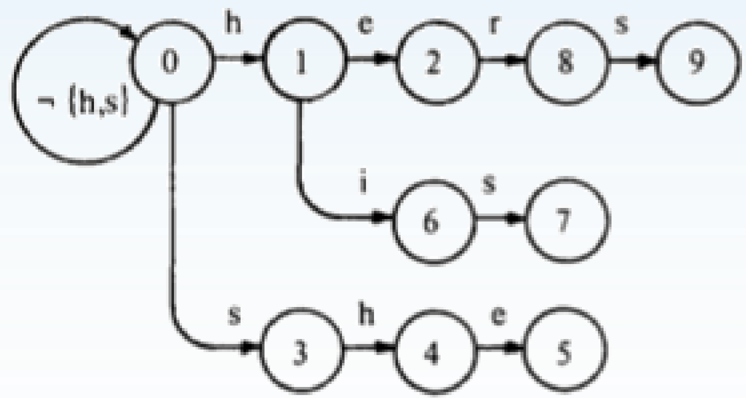
(c) Output function.

Expression Evaluation Kernel

- Evaluate Boolean expressions against binary matching flags from the bitmap
- Each CUDA thread maps to one comparison (a unique pair of document and expression)
- Expressions can be nested
- Expressions are sorted by size and grouped into CUDA blocks for more efficient execution
- Possible to heavily optimize this process

Expression Evaluation

Keywords: he, she, his, hers



(a) Goto function.

i	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	0	1	2	0	3	0	3

(b) Failure function.

i	*output* (i)
2	{he}
5	{she, he}
7	{his}
9	{hers}

(c) Output function.

Shelly had a husband. His name was Herbert, but she didn't like his cooking and he found her bothersome.

"husband" AND "cooking" AND ("delicious" OR "like")

"delicious" AND "cooking" AND ("husband" OR "wife")

Improvement

- Keyword-matching happens very quickly, but expression evaluation is slow
  - Checking each expression against each document is effectively an exhaustive search, taking roughly 95% of the runtime
  - Will become increasingly expensive with increasing expressions
- Time to evaluate an individual expression is independent of document size or keywords matched
- If we can guarantee that all expressions are inclusive (no NOT operators), then we can prune the expressions search space

Expression Space Pruning

- Perform a Boolean OR reduction for all keyword matches across a batch of documents
- Evaluate the reduced keyword matches against all expressions
- Expressions which evaluate as true form the pruned expression set
  - May have false positives
  - Guaranteed no false negatives
- Perform fine-grained matching of each document in the document batch against each of the pruned expressions
  - Eliminates false positives

Expression Space Pruning

- Two stages
  - Coarse-grained pruning to remove the vast majority of expressions from the search space (for each document batch)
  - Fine-grained to eliminate false positives and decide exactly which expressions match which documents
- Performance depends on a number of characteristics
  - Most are data-dependent
- Can be generalized to a multi-stage approach for improved performance, but we use two for simplicity
- Let’s model the performance

Mathematical Model

$$c_{total} = c_1 + c_2$$

$$c_1 = \frac{d \times e}{b}$$

$$c_2 = d \times E(b)$$

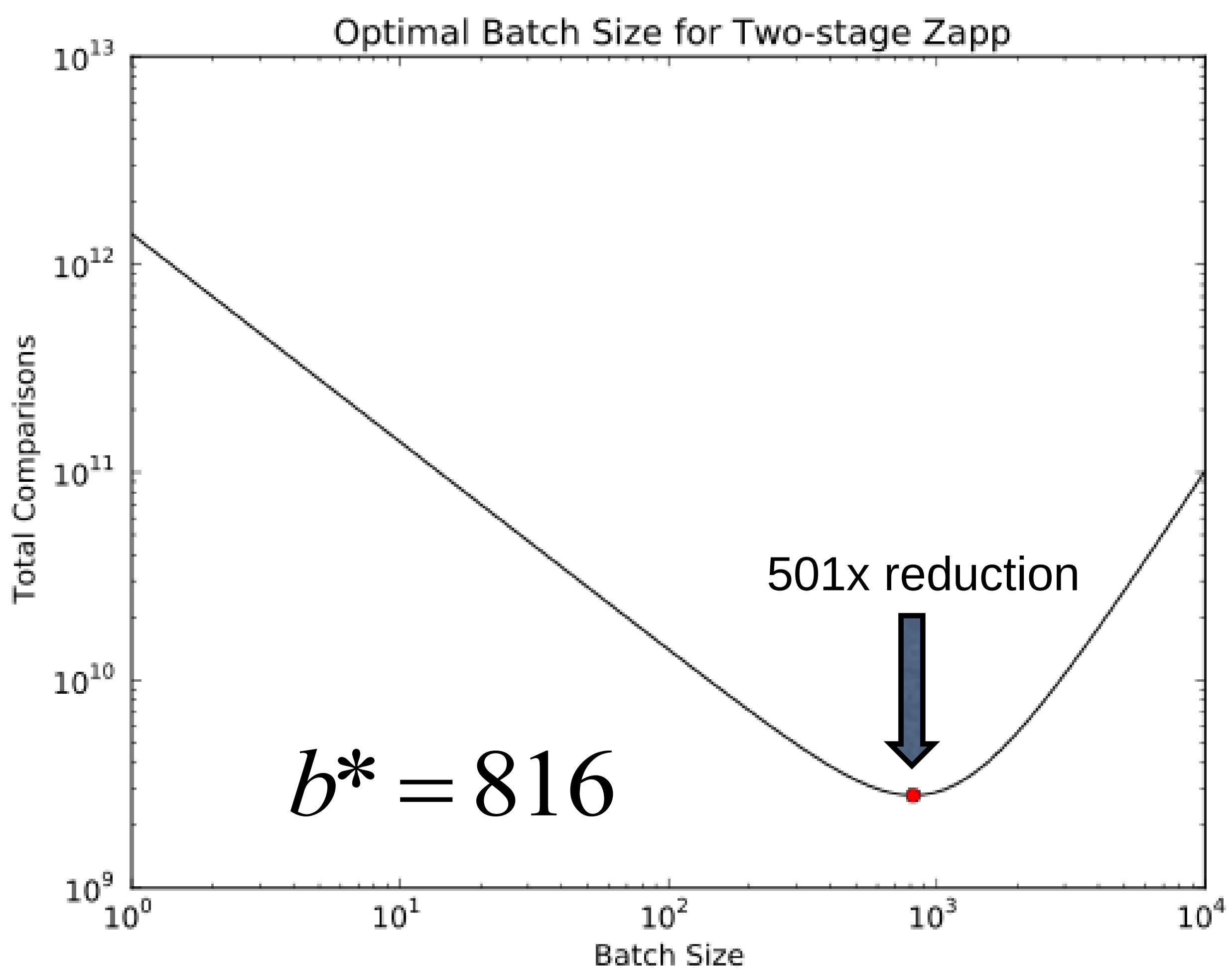
c : comparisons
e : expressions
d : documents
b : batch size
$E(b)$: expected matches

$E(b)$ is the average number of expressions we would expect to match a document batch of size b. This is determined empirically and is best modeled with a quadratic function.

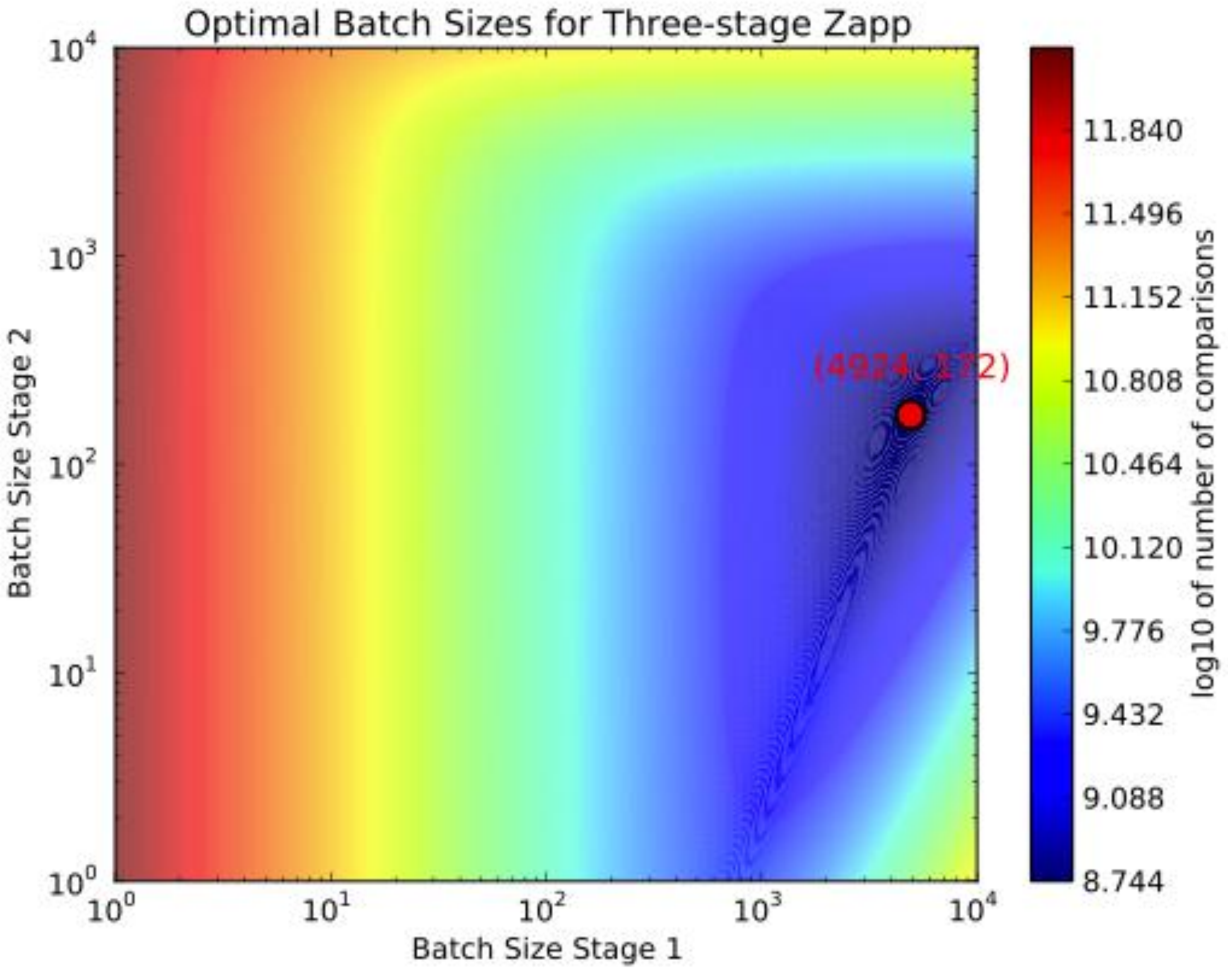
Optimization

- We wish to minimize the number of expression evaluations (i.e., document-expression comparisons)
- Free variable: documents per batch
  - Too few will require many comparisons in the first stage
  - Too many will produce increase the number of expressions matched by the batch, requiring many comparisons in the second stage
- Since the expected matches function is quadratic, we are guaranteed a unimodal objective function (even for the multi-stage generalization).

Two-stage Optimization



Three-stage Optimization



2522x reduction

Results

- We have built a cost-effective, scalable search engine powered by CUDA
- Need only two GTX580 GPUs to handle all tweets during peak load times
  - Frees up a great deal of hardware and personnel resources for other purposes
- Computational expense no longer increases linearly with number of expressions
  - Now closer to $O(n^{2/3})$ instead of $O(n)$
  - Difficult to pin down a firm figure due to model complexity and empirical assumptions

Thank you!