gSSJoin: a GPU-based Set Similarity Join Algorithm

Sidney R. Junior, Rafael D. Quirino, Leonardo A. Ribeiro, Wellington S. Martins
1 Introduction
2 Background
3 The gSSJoin Algorithm
4 Experiments
5 Related Work
6 Conclusion and Future Work
Set similarity join returns all pairs of similar sets from a dataset. Sets are considered similar if the value returned by a set similarity function for them is not less than a given threshold.
Applications

- Integration
- Cleaning
- Plagiarism identification
- Data Mining
Most state-of-the-art algorithms use a technique called *prefix filtering* to prune dissimilar sets by inspecting only a fraction of them.
Limitations

Prefix filtering is only effective at high threshold values. When the threshold value is low, the fraction needed to be inspected is bigger and the remaining sets needed to be verified after the pruning step is much higher.
Contributions

- A fine-grained parallel algorithm for both indexing the data and performing the set similarity joins.
- A highly threaded GPU implementation that takes advantage of intensive occupation, hierarchical memory, and coalesced memory access.
- A scalable multi-GPU implementation that exploits both data parallelism and task parallelism.
- Extensive experimental work with standard real-world textual datasets.
Data Representation

We map the strings to sets of tokens called *q-grams*. *q-grams* are sub-strings of length *q* obtained by sliding a window over the characters of the input string.

3-grams of "gSSJoin": {"gSS", "SSJ", "SJo", "Joi", "oin"}
Given two set collections $\mathcal{R}$ and $\mathcal{S}$, a set similarity function $\text{sim}$, and a threshold $\tau$, the set similarity join between $\mathcal{R}$ and $\mathcal{S}$ returns all scored set pairs $\langle (r, s), \tau' \rangle$ s.t. $(r, s) \in \mathcal{R} \times \mathcal{S}$ and $\text{sim}(r, s) = \tau' \geq \tau$. 
A set similarity function $sim(r, s)$ returns a value in $[0, 1]$ to represent their similarity. A greater value indicates that $r$ and $s$ have higher similarity.
Similarity Function

Jaccard Similarity: \( J(r, s) = \frac{|r \cap s|}{|r \cup s|} \)
A predicate involving the Jaccard similarity and a threshold $\tau$ can be equivalently rewritten into a set overlap constraint:

$$J(r, s) \geq \tau \iff |r \cap s| \geq \frac{\tau}{1+\tau}(|r| + |s|).$$
Prefix

For a threshold $\tau$, we can identify all candidate matches of a given set $r$ using a prefix of length $|r| - \lceil |r| \cdot \tau \rceil + 1$. 
Algorithm 1: General set similarity join algorithm.

\begin{algorithm}
\textbf{Input}: A sorted set collection $C$, a threshold $\tau$
\textbf{Output}: A set $S$ containing all pairs $(r, s)$ s.t. $\text{sim}(r, s) \geq \tau$

1. $I_1, I_2, \ldots I_{|U|} \leftarrow \emptyset$, $S \leftarrow \emptyset$
2. \textbf{foreach} $r \in C$ \textbf{do}
3. \hspace{1em} \textbf{foreach} $t \in \text{pref}_\beta(r)$ \textbf{do}
4. \hspace{2em} \textbf{foreach} $s \in I_t$ \textbf{do}
5. \hspace{3em} \text{if not} $\text{filter}(r, s)$
6. \hspace{4em} $S \leftarrow S \cup \text{refine}(r, s)$
7. \hspace{3em} $I_t \leftarrow I_t \cup \{r\}$
8. \textbf{return} $S$
\end{algorithm}
Limitations of Current Algorithms

- High threshold
- Low threshold
- Prefix size

threshold effect
Limitations of Current Algorithms

![Bar chart showing the number of candidates for different thresholds.](chart.png)
Limitations of Current Algorithms

The diagram shows the number of candidates at different q-gram sizes. The x-axis represents the q-gram size, and the y-axis shows the number of candidates. The q-gram sizes are 2, 3, and 4, and the number of candidates is denoted as $10^9$, $10^8$, and $10^7$, respectively.
The gSSJoin Algorithm

- Exploits massive parallelism instead of filtering
- Builds an inverted index on the pre-processing phase
- Performs a set similarity search for each set
Algorithm 2: $Data\text{Indexing}(E)$

\begin{verbatim}
input : token-set pairs in $E[0..|E|−1]$.  
output: $count$, $index$, $cardinality$, $invertedIndex$.  
1 array of integers $count[0..|V|−1] // count array, initialized with zeros.  
2 array of integers $index[0..|V|−1]$.  
3 array of integers $cardinality[0..|S|−1]$.  
4 $invertedIndex[0..|E|−1] // the inverted index$  
5 Count the occurrences of each token in parallel on the input and accumulates in $count$.  
6 Perform an exclusive parallel prefix sum on $count$ and stores the result in $index$.  
7 Access in parallel the pairs in $E$, with each processor performing the following tasks:  
8 \hspace{1em} begin $\hspace{2em}$  
9 \hspace{2em} Contribute on the cardinality computation of each set in $cardinality[s]$.  
10 \hspace{2em} Store in $invertedIndex$ the entries corresponding to pairs in $E$, according to $index$.  
11 \hspace{2em} end $\hspace{2em}$  
12 Return the arrays: $count$, $index$, $cardinality$ and $invertedIndex$.
\end{verbatim}
Algorithm 3: \texttt{SimilaritySearch}(invertedIndex, s)

\begin{algorithm}
\begin{algorithmic}[1]
\Statex \textbf{input}: invertedIndex, count, index, cardinality, threshold, inputset \texttt{s}[0..|\texttt{V}_s| - 1].
\Statex \textbf{output}: Jaccard similarity array \texttt{jac_sim}[0..|\texttt{S}| - 1] initialized with zeros.
\State array of integers \texttt{count}_s[0..|\texttt{V}_s| - 1] initialized with zeros
\State array of integers \texttt{index}_s[0..|\texttt{V}_s| - 1]
\For{each token \texttt{t}_i \in \texttt{s}, in parallel}
\State \texttt{count}_s[\texttt{t}_i] = \texttt{count}[\texttt{t}_i];
\State \texttt{index}_s[\texttt{t}_i] = \texttt{index}[\texttt{t}_i];
\EndFor
\State Perform an inclusive parallel prefix sum on \texttt{count}_s and stores the results in \texttt{index}_s
\For{each processor \texttt{p}_i \in P}
\For{\texttt{x} \in [\left[ \frac{|E_s|}{|P|} \right], \min((i + 1)\left[ \frac{|E_s|}{|P|} \right] - 1, |E_s| - 1)]}
\State // Map \texttt{x} to the correct position \texttt{indInvPos} of the invertedIndex
\State \texttt{pos} = \texttt{min}(i : \texttt{index}_s[i] > \texttt{x});
\If{\texttt{pos} = 0}
\State \texttt{p} = 0; \texttt{offset} = \texttt{x};
\Else
\State \texttt{p} = \texttt{index}_s[\texttt{pos} - 1]; \texttt{offset} = \texttt{x} - \texttt{p};
\EndIf
\State \texttt{indInvPos} = \texttt{start}_s[\texttt{pos}] + \texttt{offset}
\State uses \texttt{s}[\texttt{pos}] and invertedIndex[\texttt{indInvPos}] in the partial computation of the intersection
\State between \texttt{s} and the set associated to invertedIndex[\texttt{indInvPos}]
\EndFor
\For{\texttt{x} \in [\left[ \frac{|S|}{|P|} \right], \min((i + 1)\left[ \frac{|S|}{|P|} \right] - 1, |S| - 1)]}
\State // Jaccard similarity calculation for \texttt{S} sets using \texttt{P} processors
\State uses intersection and union (through cardinality) to compute the Jaccard similarity
\State flag sets with Jaccard similarity above the threshold
\EndFor
\State \textbf{end}
\State \textbf{end}
\State Perform an exclusive parallel prefix sum on the flagged sets to compact the selected sets
\State \textbf{Return} the array: \texttt{jac_sim} with the selected jaccard similarities.
\end{algorithmic}
\end{algorithm}
Multi-GPU

- The inverted index is created in all GPUs
- The search sets are divided between the GPUs
Algorithm 4: \textit{MultiGPUSearch}(E)

\begin{algorithmic}
\State \textbf{input} : token-set pairs in $E[0 \ldots |E| - 1]$. \\
\State \textbf{output}: A list of the most similar, one for each set.
\State \begin{align*}
1 & \text{for each } i \in g, \text{ in parallel do} \\
2 & \quad \text{set.gpu.device}(i); \\
3 & \quad \text{DataIndexing}(E); \\
4 & \text{end}
\end{align*}
\State Allocate memory for the biggest set;
\State \begin{align*}
5 & \text{for each } j \in g, \text{ in parallel do} \\
6 & \quad \text{set.gpu.device}(j); \\
7 & \quad \text{for each set } s \in (m/g) \text{ in parallel do} \\
8 & \quad \quad \text{SimilaritySearch}(invertedIndex, s); \\
9 & \quad \text{end}
\end{align*}
\State \textbf{end}
\State \textbf{Return} A list of the most similar sets, one for each set.
\end{algorithmic}
Experimental Setup

We implemented gSSJoin using the CUDA Toolkit version 7.5. The experiments were conducted on a machine running CentOS 7.2.1511 64-bits, with 24 Intel Xeon E5-2620, 16GB of ECC RAM, and four GeForce Zotac Nvidia GTX Titan Black, with 6GB of RAM and 2,880 CUDA cores each.
Databases

<table>
<thead>
<tr>
<th>Database</th>
<th>Number of sets</th>
<th>Max</th>
<th>Mean</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>100k</td>
<td>205</td>
<td>70</td>
<td>23.9</td>
</tr>
<tr>
<td>Netflix</td>
<td>200k</td>
<td>54</td>
<td>30.54</td>
<td>8.2</td>
</tr>
</tbody>
</table>
Performance Results

Figure: DBLP dataset using 3-grams.
Performance Results

Figure: Netflix dataset using 3-grams.
Performance Results

Figure: DBLP dataset using 2-grams.
Performance Results

**Figure**: Netflix dataset using 2-grams.
### Performance Results

#### Table: Execution Time (secs) Using Multiple GPUs, Threshold = 0.5

<table>
<thead>
<tr>
<th>Number of GPUs:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DBLP 2gram</strong></td>
<td>27.56</td>
<td>14.37</td>
<td>10.4</td>
<td>7.2</td>
</tr>
<tr>
<td><strong>DBLP 3gram</strong></td>
<td>16.73</td>
<td>8.77</td>
<td>6</td>
<td>4.47</td>
</tr>
<tr>
<td><strong>Netflix 2gram</strong></td>
<td>41.10</td>
<td>21.63</td>
<td>14.37</td>
<td>11.2</td>
</tr>
<tr>
<td><strong>Netflix 3gram</strong></td>
<td>23.52</td>
<td>12.69</td>
<td>8.5</td>
<td>6.5</td>
</tr>
</tbody>
</table>
Related Work


- A distributed memory MapReduce framework to perform similarity joins was proposed by [Vernica et al. 2010].

- Parallel solutions such as [Cruz et al. 2015] perform approximate set similarity joins using MinHash and Locality Sensitive Hashing.
Conclusion

The gSSJoin algorithm has a much better performance than the state-of-the-art CPU-based algorithms when the threshold is low and the database is more uniform.

In future work, we plan to incorporate candidate filtering techniques to obtain even greater speed-ups.
Questions?

email: srjsoftware@gmail.com
[Cruz et al. 2015]

[Bayardo et al. 2007]

[Chaudhuri et al. 2006]

[Ribeiro and Härder 2011]

[Sarawagi and Kirpal 2004]
Sarawagi, S. and Kirpal, A. Efficient Set Joins on Similarity Predicates. In SIGMOD.
References

[Vernica et al. 2010]

[Xiao et al. 2011]

[Wang et al. 2012]
GPU Architecture

Introduction
Background
The gSSJoin Algorithm
Experiments
Related Work
Conclusion and Future Work