Duplicate Detection on GPUs

Benedikt Forchhammer[1], Thorsten Papenbrock[1], Thomas Stening[1], Sven Viehmeier[1], Uwe Draisbach[2], Felix Naumann[2]

Hasso Plattner Institute
14482 Potsdam, Germany
[1]firstname.lastname@student.hpi.uni-potsdam.de
[2]firstname.lastname@hpi.uni-potsdam.de

Abstract: With the ever increasing volume of data and the ability to integrate different data sources, data quality problems abound. Duplicate detection, as an integral part of data cleansing, is essential in modern information systems. We present a complete duplicate detection workflow that utilizes the capabilities of modern graphics processing units (GPUs) to increase the efficiency of finding duplicates in very large datasets. Our solution covers several well-known algorithms for pair selection, attribute-wise similarity comparison, record-wise similarity aggregation, and clustering. We redesigned these algorithms to run memory-efficiently and in parallel on the GPU. Our experiments demonstrate that the GPU-based workflow is able to outperform a CPU-based implementation on large, real-world datasets. For instance, the GPU-based algorithm deduplicates a dataset with 1.8m entities 10 times faster than a common CPU-based algorithm using comparably priced hardware.

1. Introduction

Duplicate detection (also known as *entity matching* or *record linkage*) is the task of identifying multiple representations of the same real-world entities [NH10]. It is an integral part of data cleansing and an important component of every ETL process. Duplicate detection is typically performed by applying similarity functions to pairs of entries in datasets: Some algorithm carefully selects promising pairs of records. If the values of two records are sufficiently similar, they are assumed to be duplicates. Due to the large number of comparisons and the ever-increasing size of many databases, duplicate detection is a problem that is hard to solve efficiently. However, in most approaches the comparisons of record pairs are independent from one another – the problem is highly parallelizable. In this paper, a selection of duplicate detection algorithms and similarity measures are described and adapted in the context of General Purpose Computation on Graphics Processing Units (GPGPUs).

General purpose GPU programming has gained much appreciation in the past few years. Unlike *Single Instruction, Single Data* (SISD) CPU architectures, *Single Instruction, Multiple Data* (SIMD) GPU computing allows the execution of one set of operations on large amounts of data in a massively parallel fashion. This parallelization can provide immense speedups in applications that focus on highly data-parallel problems.

Currently, there are only few frameworks for GPGPU development. For our prototype, we use the OpenCL 1.0 framework, as it allows development for both ATI and NVIDIA graphics cards. The framework allows the execution of so-called kernels, which are written in a variant of ISO C99. OpenCL kernels can be executed on different devices; usually the device is a graphics card, but other devices, in particular the CPU, are also possible if respective hardware drivers are available. Devices execute kernels as work items. A work item is a set of instructions that are executed on specific data by one thread. Further work items are grouped into work groups.

When developing applications for GPUs, memory management is a key factor: GPUs have four types of memory with different capacities and different access speeds: *Global memory* is slow but has the highest capacity; *local memory* is faster but has a far smaller capacity; *private memory* is only usable by one operating unit; and *constant memory* is the fastest but not writable by the graphics card. An additional difficulty lies in the fact that it is not possible to allocate memory dynamically on the GPU. We address these memory challenges and opportunities in the next sections. Concerning the execution units, the graphics card executes a number of threads (usually 32) in so-called *warps*. All threads within a warp execute the same instructions on different data. If one thread of a warp takes a longer execution time, all the others wait. Moreover, conditions in the program flow are serialized; each thread waits until the complete warp finishes an `if`-statement, before starting with an `else`-statement. After an `else`-statement the threads are synchronized as well. Hence, we avoid divergent branching as far as possible.

Our main contribution is a complete duplicate detection workflow that utilizes the resources of the GPU as much as possible. First, we describe how each algorithm can be parallelized to utilize a very high amount of GPU cores. Second, we propose algorithm specific data-partitioning structures and memory access techniques to organize data in the NUMA architecture of GPUs. Finally, we present experiments that evaluate the performance of the presented workflow based on different CPU and GPU hardware. For comparison reasons, we optimized the algorithmic parameters for high precision and recall values (not for speed) and used real world data sets as input data.

In the following Sec. 2, we highlight related work for the areas of duplicate detection and GPGPU programming. Section 3 introduces the individual components of the duplicate detection workflow. Section 4 describes our adaptations for two popular pair-selection methods for the GPU environment. In Sec. 5 we adapt algorithms for popular similarity measures, as well as for the aggregation of different result lists and clustering. Section 6 evaluates the components of the workflow on various hardware platforms. The last section summarizes our results and discusses future work.

2. Related Work

Duplicate detection has been researched extensively over the past decades. Recent surveys [EIV07,NH10] explain various techniques for duplicate detection and methods for improving effectiveness and efficiency. Common approaches to improve the efficiency of duplicate detection are blocking and windowing methods, such as the Sorted Neighborhood method [HS95], which reduce the number of comparisons. Another approach to reducing execution time is parallelization, i.e., splitting the problem into smaller parts and distributing them onto multiple computing resources. Our approach combines both the Sorted Neighborhood method and parallelization.

Parallelization has been proven to be effective by various authors. One of the first approaches to parallelizing duplicate detection is the Febrl system [CCH10], which is implemented in Python and parallelized via the well-known Message Passing Interface (MPI) standard. Kim and Lee presented a match/merge algorithm for cluster computing based on distributed Matlab [KL07]. Kirsten et al. developed a parallel entity matching strategy for a service-based infrastructure [KKH10]. They evaluate both the Cartesian product as well as a blocking approach, and demonstrate that parallelization can be used to reduce execution time significantly. Kolb et al. explored *map-reduce* to bring duplicate detection onto a cloud infrastructure [KTR11]. They focus on parallelizing the Sorted Neighborhood method and their experiments show nearly linear speedup for up to 4 and 8 cores. While these papers present effective approaches to the problem of parallelizing duplicate detection, they all require multiple CPUs or PC clusters for parallelization. This limits the level of parallelization that can be achieved, e.g., Kirsten et al. use up to 4 nodes and 16 CPUs for evaluation. Compared to what is possible with GPUs, the respective level of parallelization is low.

Katz and Kider worked on parallelizing transitive closure, i.e., the step of transforming a list of duplicate pairs into duplicate clusters [KK08]. In contrast to other papers on this topic [AJ88,To91] which only use CPUs for parallelization, Katz and Kider's approach utilizes graphics cards. Their algorithm is, however, not scalable for a large number of input pairs, as it is limited by the amount of memory available on the GPU. Our prototype builds on their work and solves this scalability issue.

GPGPU programming has received an increasing amount of attention over past years. Recent surveys show that applications for GPGPU can be found in a wide area of fields including database and data mining applications [ND10,OLG07]. For duplicate detection, however, most approaches have been targeted at distributed infrastructure and do not consider the unique challenges presented by GPUs. To the best of our knowledge, we are first to evaluate a complete duplicate detection workflow on GPUs.

3. Duplicate Detection Workflow

This section presents a complete duplicate detection workflow, which combines common duplicate detection algorithms with the computation capacities of modern graphics cards. Figure 1 gives an overview of the workflow with the following steps:

Parsing converts the input data, e.g., a CSV file, into an internal character array with all values concatenated. To allow values of different lengths, an additional array containing the starting indices of the individual attribute values is needed. This format is essential, because GPU-kernels can only handle basic data types and arrays with known sizes.

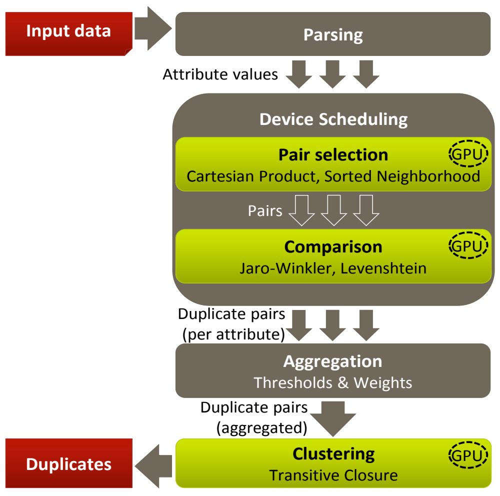


Figure 1: The duplicate detection workflow

Pair Selection selects record pairs for comparison. We adapt the *Cartesian product* and the *Sorted Neighborhood* algorithms to run on the GPU. To generate a sorting key for the *Sorted Neighborhood* algorithm, we present a simple *key-generation function* and an adapted *Soundex* algorithm, both running on the GPU.

Comparison: The selected record pairs are compared for similarity: We process each attribute value individually and return a normalized similarity value for each pair of attribute values. We describe the computation of two edit-based similarity measures on the GPU: *Levenshtein* and *Jaro-Winkler.*

Aggregation: The attribute similarities are aggregated to an overall record pair similarity, which is used to decide whether the two records are duplicates or not. We calculate a weighted average and check similarity values before and after the aggregation against predefined thresholds.

Clustering: The result of a pairwise duplicate detection process may not contain all transitively related record pairs. Thus, we calculate the transitive closure to obtain a complete list of duplicate clusters.

4. Pair Selection

Next to the Cartesian product, the literature knows several algorithms that select a subset of candidate pairs for comparison to avoid the complexity of comparing all pairs; a popular representative is the Sorted Neighborhood Method [HS95].

Regardless of the used algorithm, to completely utilize the parallel potential of GPUs, each work item compares exactly one selected pair of attribute values. This leads to a higher amount of work items than the GPU has processors, and, therefore, allows the GPU to use optimization techniques like memory latency hiding.

Since the memory of graphics cards is limited, it cannot fit all values of a large dataset. Thus, we cannot execute all comparisons at once and, instead, have to perform multiple comparison rounds. Each round consist of the following steps: Copy a subset of attribute values from the host to the GPU, execute the comparisons on those values, and finally copy the results back from the GPU to the host. We describe two approaches to divide the input values into blocks of data and select the comparisons for each round.

4.1 Cartesian product

The simplest method to select pairs is the Cartesian product. It selects every possible combination of input values. This leads to high recall, but also to a high number of comparisons. In general, the set of pairs must be split into chunks that fit into memory. This split can be performed easily with CPU and main memory due to dynamic memory allocation. But on the GPU, memory allocations must be done *before* the GPU executes the kernel code. Especially, different lengths of input values lead to different memory requirements for each comparison.

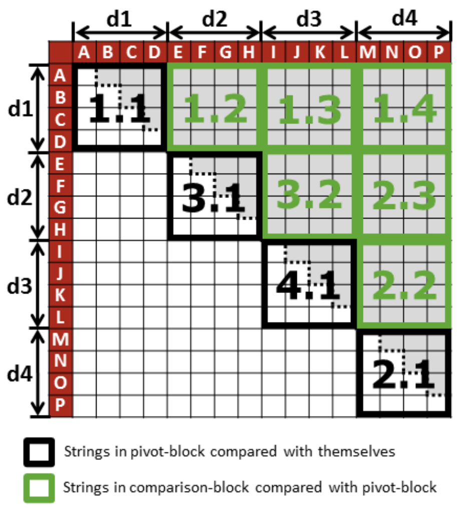

d1
d2
d3
d4
A B C D E F G H I J K L M N O P
1.1
1.2
1.3
1.4
3.1
3.2
2.3
4.1
2.2
2.1
Strings in pivot-block compared with themselves
Strings in comparison-block compared with pivot-block

Figure 2: Cartesian product pair selection

For an optimal usage of GPU-resources two requirements must be met: First, the transfer of data between main memory and graphics cards should be minimized, i.e., data on the GPU should be reused as much as possible. Second, the entire available memory should be used to fully utilize the parallel potential of the GPU. To fulfill these goals, we establish two blocks of GPU memory of about the same size: The first block is the *pivot-block*, which is kept on the graphics card until all comparisons with its values are finished. The second block is the *comparison-block*, whose content is exchanged in each round.

Figure 2 shows which blocks of input data are compared. The x- and y-axes represent the input values; each cell represents a comparison between a value from the x- and a value from the y-axis. The comparisons under and on the diagonal (white cells) are never performed, because we assume symmetric comparison measures.

First, the pivot-block contains data `d1` and is compared with itself in round 1.1. Then in rounds 1.2 to 1.4, the comparison-block is filled with input data `d2` to `d4` and compared with the pivot-block. The data in the last comparison-block is then kept on the graphics card and used as the new pivot-block. The selection of the last comparison-block as the new pivot-block can lead to very small pivot-blocks, which in turn leads to fewer comparisons. To avoid this effect, the algorithm pre-calculates the optimal size of the new pivot-block based on the current pivot-block. We call it the *candidate-block,* and compare it with the pivot-block after all other comparison-blocks have been processed.

Assuming that the pivot-block contains p elements and the comparison-block contains c elements, we can do $p * c$ comparisons in parallel and thus maximally utilize the parallel potential of the GPU. The comparisons in rounds x.1 are exceptions, because they compare the pivot-block with itself, with $p * \frac{p-1}{2}$ comparisons in parallel. Every kernel has to calculate the memory addresses of the values that it should compare. To unify the calculation and to avoid branches, we increase the number of comparisons to $p * \left\lceil\frac{p-1}{2}\right\rceil$. Now, $p/2$ work items always compare the same string to one of the following $p/2$ strings – continuing at the beginning of the value array if its end is reached. This generates duplicate results if p is an even number, but the subsequent aggregation algorithm (see Sec. 5.4) filters them out.

We process input values with different lengths. Thus, we cannot use blocks of fixed size. Instead, the sizes of pivot- and comparison-blocks have to be determined based on input data and any additional memory required by a specific comparison algorithm. Additionally, the block-sizes are limited by the GPU-memory. This leads to the formula:

$$Memory \geq Strings + AlgorithmData + Results \qquad (1)$$

where *Strings* represents the size in bytes of the strings in both blocks (including the index arrays), *AlgorithmData* represents the individual requirements of a comparison algorithm, and *Results* is the size of the array that contains the calculated similarity values. Furthermore, the two goals of using the entire available memory and minimizing the data transfer have to be fulfilled by the value selection.

Our approach dynamically calculates the block's memory requirements depending only on the current lengths of the strings in the input data: First, it calculates the size of the pivot-block, which also depends on the strings in the comparison block, by increasing its size continuously. Since the strings of the comparison-block are not known at this time, the content of the comparison-block must be estimated. We assume that the comparison-block contains one string with average length for every string in the pivot-block. This approach fulfills the goal of maximizing the number of comparisons in each round. The pivot-block is filled with strings until the memory is too small to contain the pivot-block, the estimated comparison-block, and the additional memory for the comparison algorithm. Then the pivot-block is transferred to the GPU and compared with itself. Since the size and the content of the pivot-block are now fixed, the content of the comparison-block can be calculated based on the input data and the pivot-block. As the strings have different lengths, the comparison-block can contain more or fewer strings than the pivot-block. Our experiments show that usually both contain nearly the same number of strings, because of the average length estimation.

The Levenshtein algorithm for comparing attribute values needs additional memory: each comparison of a string from the pivot-block with a string from the comparison-block requires two times the size of the string from the pivot-block. Thus, it needs the size of the pivot-block times the number of strings in the comparison-block as additional memory (see Sec. 5.1 for more details). In the best case, the pivot-block contains many short strings while the comparison-block contains few long strings. Then, the Levenshtein algorithm will only need a small amount of memory and more strings can be placed on the graphics card. In the worst case, the pivot-block contains few long strings and the comparison-block contains many short strings. In this case, the Levenshtein algorithm needs more memory for the comparisons. To avoid the worst case, one could always compare the shorter with the longer string, but in this case the calculation of memory addresses in the kernel becomes overly complex. Jaro-Winkler does not need additional memory for its comparisons; it uses the complete GPU-memory for attribute values. Thus, the number of comparisons does not depend on the contents of the blocks.

4.2 Sorted Neighborhood

The Sorted Neighborhood Method (SNM) [HS95] greatly reduces the number of comparisons compared to the Cartesian product. It consists of three phases: First, a sorting key is generated; then the records are sorted according to that key in the hope that duplicates have similar sorting keys and thus end up close to each other; and finally a fixed-size window is slid over the sorted records and all records within the same window are compared to each other.

For *key generation* we propose a simple hashing algorithm as well as the Soundex code (see Sec. 5.3): The simple hashing approach uses the string length and the first letter of the attribute to be compared, to create a sort key: $1{,}000 \cdot |String| + firstLetterCharValue$ results in a sorting primarily according to the length and secondly according to the first letter. Sorting by length leads to comparisons of strings of roughly same length, which reduces branch divergence – an advantage in GPU processing. Further, the first letter is often the same for duplicate strings, e.g., because spelling mistakes are less likely to be made here [YF83], and because abbreviations start with the same letter. The computation of the simple sort key requires no branching, and therefore all GPU threads can run in parallel. The downside of this simple key generation is that it produces poor results if there are many strings with the same length, which results in many similar keys. In this case, Soundex (described in Sec. 5.3) produces better sort keys, because it focuses on phonetic characteristics instead of the string length. Altogether, sorting key generation is well suited for GPUs, because each calculation only depends on one string, and therefore can be easily computed in parallel.

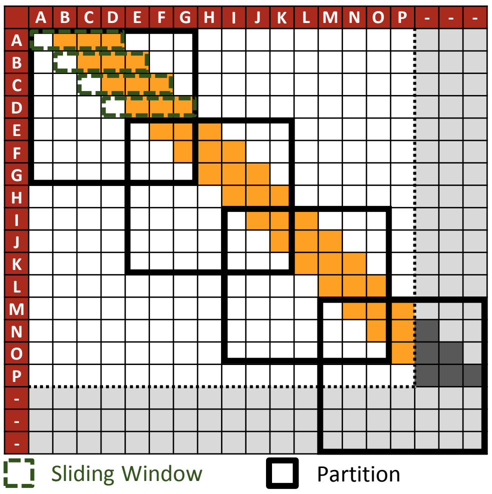

A B C D E F G H I J K L M N O P - - -
A B C D E F G H I J K L M N O P - - -
Sliding Window
Partition

Figure 3: Sorted Neighborhood pair selection

For the sorting step, we choose the GPU-based merge sort algorithm of [SKC10]. We did not improve this sorting algorithm; therefore, we do not cover it in this paper.

Figure 3 shows the comparison matrix for SNM. The colored cells on the diagonal denote the comparisons that are actually performed by SNM. To compare the attribute values efficiently, we have to determine the maximum amount of strings that can be transferred to the GPU at a time. In Fig. 3, a partition visualizes the comparisons that can be done with the strings on the GPU in one execution. We approximate the number of the strings that can be copied to the GPU based on the average string length. The amount of memory required for one string depends on the length of the string, the number of strings it is compared with, and the comparison algorithm.

The Sorted Neighborhood approach compares each string with the next $w-1$ strings where w is the window size (see the sliding window, Fig. 3). This leads to $\left((w-1)\cdot |AllStringsInPartition|\right)-(w-1)^2$ comparisons in one round on the GPU. In this formula, $(w-1)^2$ denotes the comparisons that are postponed to the next partition, due to the window sliding out of the partition boundaries.

With this number of comparisons and the average string length, we approximate the maximum number of strings that can be copied to the GPU. Since the input strings have different length, we iteratively calculate the required memory based on the approximation, until the maximum number of strings that can be computed on the graphics card is determined. The calculation benefits from the internal data format, produced during the parsing step (see Sec. 3). It allows the computation of the string lengths just by inspecting the index array with the starting indices of the strings.

Once the data is copied onto the GPU, each string within a partition is compared with the next $w-1$ strings. The last strings in a partition cannot be compared, because they are not followed by $w-1$ strings. Therefore, partitions have to overlap by $w-1$ strings to ensure that no comparisons are missed. To execute the comparisons in the last partition efficiently, the index array is expanded by $w-1$ dummy string entries (see header-cells labeled with "-" in Fig. 3). These dummy strings prevent branching, because the last strings of the final partition can be treated like any other string without needing conditional checks. Furthermore, the dummies are empty, so their respective comparisons can easily be omitted by the kernels. Thus, they do not negatively impact computation time.

5. Similarity Classification

This section describes implementations of methods to classify record pairs as duplicate or non-duplicate. In particular, we present two edit-based and one phonetic (Soundex) similarity measure to calculate the attribute similarity on graphics cards. Then we describe how different attribute similarities are aggregated to record similarities and how we cluster results using GPUs.

5.1 Levenshtein similarity

The Levenshtein distance is defined as the minimum number of character insertions, deletions, and replacements necessary to transform a string s_1 into another string s_2 [NH10]. To compute the Levenshtein distance $LevDist(s_1, s_2)$ on a GPU, we use a dynamic programming approach [MNU05] and extend this approach to optimize its memory usage. The comparison of two strings requires a matrix M of size $(|s_1|+1)\times(|s_2|+1)$, where $|s|$ denotes the length of string s. A value in the i-th row and j-th column of M is defined by $M_{i,j}$, where $0 \leq i \leq |s_1|$ and $0 \leq j \leq |s_2|$. We initialize the first row $M_{0,j}$ and column $M_{i,0}$ as:

$$M_{0,j} = j \qquad\qquad M_{i,0} = i \quad (2)$$

The algorithm then iterates from the top left to the bottom right cell of the matrix. It recursively computes each value $M_{i,j}$ in the matrix as:

$$M_{i,j} = \begin{cases} M_{i-1,j-1} & \text{if } s_{1,i} = s_{2,j} \\ 1 + min(M_{i-1,j}, M_{i,j-1}, M_{i-1,j-1}) & \text{otherwise} \end{cases} \quad (3)$$

where $s_{k,i}$ denotes the i-th letter in the string s_k. In the end, matrix cell $M_{|s_1|,|s_2|}$ delivers the Levenshtein distance between s_1 and s_2.

Because dynamic memory allocation is not possible from inside a GPU kernel in OpenCL, we pre-allocate the needed memory for each comparison. To reduce memory consumption, we use only two matrix rows for each comparison, because calculation of row i depends only upon the current row i and the previous row $i - 1$ (see Equation 3). Thus, we can swap the current and previous row and calculate row $i + 1$ by overwriting the values of row $i - 1$, without affecting performance. We analyzed that the average string length in our test collection is 14 characters, which results in an average matrix size of $(14 + 1) \cdot (14 + 1) = 225$ cells. By using only two rows, we can greatly reduce the average required cells in our test collection to $(14 + 1) \cdot 2 = 30$, which is only 13% of the whole matrix.

To calculate the amount of required memory for the matrix rows, the algorithm can use a simple formula that takes the arbitrary length of each string $|s_k|$ into account. Let n be the number of strings that should be compared and c be the number of strings to which each of the n strings is compared to. Then the overall memory in byte that is required for the matrix rows can be calculated as:

$$c \cdot sizeof(int) \cdot 2 \cdot \sum_{k=1}^{n} (|s_k| + 1) \quad (4)$$

Within a comparison of two strings, one string defines the length of the two matrix rows. Therefore, one comparison requires $sizeof(int) \cdot 2 \cdot (|s_k| + 1)$ bytes of memory for the matrix rows. Our pair selection algorithms are designed to compare each of the n strings to $c > 0$ other strings, so that each string defines c times the length of the matrix rows. For example, the Sorted Neighborhood algorithm sets $c = w - 1$ and the Cartesian product defines $c = \left\lceil \frac{n-1}{2} \right\rceil$. To calculate the overall amount of matrix memory, the algorithm sums up all n string specific row lengths $|s_i| + 1$.

As usual, to transform the Levenshtein distance into a normalized similarity measure, we finally normalize the distance by dividing by the length of the longer string and subtract the result from 1.

5.2 Jaro-Winkler similarity

Jaro-Winkler similarity was originally developed for the comparison of names in U.S. census data. The measure is comprised of the Jaro distance [Ja89] and additions by Winkler [WT91]. The Jaro distance $JaroDist(s_1, s_2)$ combines the number of common characters m between two strings s_1 and s_2, the number of transpositions t between the two strings of matching characters, and the lengths of both strings:

$$JaroDist(s_1, s_2) = \frac{1}{3} \cdot \left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m} \right) \quad (5)$$

Common characters are only searched for within a range of size w:

$$w = \frac{max(|s_1|, |s_2|)}{2} - 1 \quad (6)$$

Winkler's main modification of the Jaro distance is the inclusion of the length of the common prefix ℓ into the formula (see Eq. 7), which improves similarity scores for names starting with the same prefix. The common prefix is limited to ≤ 4 and is weighted by the factor p, for which Winkler's default value is 0.1.

$$JaroWinkler(s_1, s_2) = JaroDist(s_1, s_2) + \Big(\ell p\big(1 - JaroDist(s_1, s_2)\big)\Big) \quad (7)$$

The original algorithm calculates the number of transpositions t by first calculating the two strings of matching characters and then comparing them character by character. For each pair of strings being compared, matching characters are stored in two temporary variables of length $l_{min} = min(|s_1|, |s_2|)$. For graphics cards, these variables pose difficulties: First, the amount of fast private/local memory is limited, which restricts the amount of work items that can be executed in parallel. In order to still achieve high levels of parallelism, global memory needs to be used, which is slower to access but also several magnitudes larger. Second, since GPU memory cannot be allocated dynamically within a kernel at runtime, we would have to wastefully pre-allocate the worst-case amount of memory or perform additional comparisons as in [HYF08].

Our approach focuses on reducing memory consumption with the goal of achieving high levels of parallelism while primarily using fast private/local memory. This comes at the cost of increased kernel-time complexity. Instead of pre-computing the strings of matching characters, our algorithm (see Alg. 1) computes the number of matched characters and transpositions by iterating the input strings twice. The first iteration (lines 3-9) finds and counts the number of matching characters m. It also keeps track of which characters have been matched already (array of matched characters $mc_{1,x}$). The second iteration (l. 12-25) calculates the number of half-transpositions t and the length of the common prefix ℓ. The $find(c, s, mc)$ function (l. 4 and l. 13) tries to find a character c in the given string s, without matching any characters that were previously matched. This is done by checking that the respective position in the mc array is not set to 1, and by respectively updating the array once a matching character has been found.

The $find$ function returns a Boolean value indicating whether a match was found, and the offset at which it was found. Internally, the function also considers the window size w (see Eq. 6) to match only characters within the allowed range of $i \pm w$.

The $countUnmatched(i, mc)$ function (l. 15) counts how many characters in s_2 up to position i cannot be matched to a character in s_1 (by inspecting the mc array). A half-transposition exists if a character can be matched without an offset, while ignoring all unmatched characters (l. 16). The prefix counter ℓ is only increased for the first 4 characters, if the current character has been matched without an offset, and all characters on earlier positions do also match respectively (l. 19).

The key to memory efficiency with this algorithm lies in the arrays $mc_{1,x}$ and $mc_{2,x}$ which store only Boolean values and thus can be represented at the level of single bits. Our implementation uses two 8-byte variables allowing comparisons of strings up to length 64. Using the original approach we would need two 64-byte variables to compare strings of the same length.

```
01 m ← 0, t ← 0, $\ell$ ← 0
02 $mc_{1,x}$ ← 0
03 for i = 1 to |$s_1$| do
04  [match,offset] ← find($s_{1,i}$,$s_2$,$mc_1$)
05  if match = True then
06   m ← m + 1
07   $mc_{1,i}$ ← 1
08  end if
09 end for
10 $mc_{2,x}$ ← 0
11 $uc_1$ ← 0
12 for i = 1 to |$s_1$| do
13  [match,offset] ← find($s_{1,i}$,$s_2$,$mc_2$)
14  if match = True then
15   $uc_2$ ← countUnmatched(i+offset,$mc_1$)
16   if offset + $uc_1$ ≠ $uc_2$ then
17    t ← t + 1
18   end if
19   if offset = 0 and $\ell$ = i - 1
        and i ∈ [1,4] then
20    $\ell$ ← $\ell$ + 1
21   end if
22  else
23   $uc_1$ ← $uc_1$ + 1
24  end if
25 end for
26 t ← t/2
27 return m, t, $\ell$
```

Algorithm 1: Jaro-Winkler: computation of matching characters **m**, transpositions **t** and common prefix ℓ for two strings $\mathbf{s_1}$ and $\mathbf{s_2}$

5.3 Soundex

Soundex is a phonetic algorithm for identifying words that are pronounced similarly but spelled differently [USN07]. The algorithm produces 4-letter codes, which match for similar sounding words, e.g., Robert and Rupert are both represented by the code R163. Soundex is good for finding misspelled names but it produces many false positives as well as false negatives [PS01].

Our implementation consists of two kernels: One for generating Soundex codes for a set of input strings, and one for comparing pairs of Soundex codes. For comparison we minimize memory operations by leaving generated Soundex codes on the graphics card for the comparison phase. To generate Soundex codes we walk through the letters of a given term and build up the Soundex code by either coding the current letter or moving on to the next one. The compare-kernel uses lists of previously generated Soundex codes to create pairs of terms that have the same code. Unlike other similarity measures, this results in similarity values of either 0 or 1.

5.4 Aggregation

To classify whether two records are a duplicate or not, we aggregate the attribute similarities to an overall record similarity. The comparators described in the previous sections return lists of pairs with similarity values above attribute-specific thresholds. The aggregated similarity value is a weighted average of all similarity values for the specific pair. In order to increase the precision of results, merged pairs with a similarity value below a manually defined overall threshold are removed.

For efficient aggregation each list is first sorted by a unique identifier that represents the compared data records. This approach reduces the search time for corresponding pairs in the result lists; additionally, duplicate entries that may have been produced by the Cartesian product (see Sec.4.1) can be removed easily.

While the sorting part is suited for the GPU, the merging part is not: First, merging on the CPU can be a simple Sort-Merge join that requires linear time, so the additional time required for copying the data to the GPU does not pay off (see Sec. 6.2). On the GPU, the retrieval of corresponding pairs is more complex, because each kernel instance would merge one combination of pairs and corresponding pairs in different lists cannot be found at the same defined places. Pairs can be missing in some lists, due to attribute-specific thresholds and different comparisons that are triggered by the Sorted Neighborhood method. Thus, a GPU variant would either need a complex kernel with slow branching, or additional preprocessing of all lists. When iterating all lists, the computation of the weighted average would only produce little to no computational overhead. This invalidates the point of using the GPU for merging, so we sort the lists on the GPU and merge them on the CPU.

5.5 Clustering

As we use pairwise comparisons to find duplicate records, our result may not be transitively closed (e.g. pairs ⟨A,B⟩ and ⟨B,C⟩ are classified duplicates, but not ⟨B,C⟩). We calculate the transitive closure using the tiled Floyd-Warshall (FW) algorithm by Katz and Kider [KK08] and adapt it to the specific task of clustering real-world duplicate pairs. We first present the tiled FW algorithm in a condensed form. Afterwards, we describe how the algorithm can be extended to optimize its efficiency and scalability in computing extremely large amounts of data.

The tiled FW extends the original FW [Wa62] in order to run efficiently on the GPU. It uses dynamic programming and is based on a directed graph, represented by an adjacency matrix M. In the design of the tiled FW, Katz and Kider assume that the entire matrix for n vertices can be loaded into the GPU's global memory at once, but not into local memory. Therefore, they propose to load all data into global memory first and then split the computation of the transitive closure into many sub-tasks that can be executed sequentially using maximal local memory in each step. After loading M into global memory, the tiled FW algorithm partitions M into sub-matrices of size $s \times s$ with $s \leq n$. Size s must be chosen small enough so that three sub-matrices can be loaded into local memory at once. M then consists of $m \times m$ sub-matrices with $m = \lceil n/s \rceil$. Afterwards, the algorithm uses an iterative execution strategy for the Floyd-Warshall algorithm (see Fig. 4). It needs m stages to calculate the complete transitive closure. Each stage consists of the following three phases:

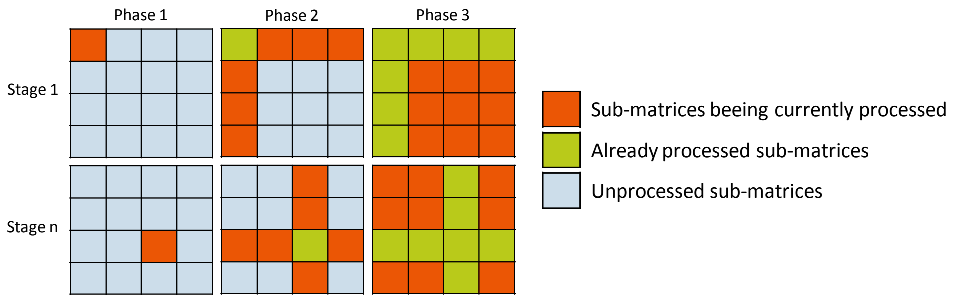

Phase 1
Phase 2
Phase 3
Stage 1
Stage n
Sub-matrices beeing currently processed
Already processed sub-matrices
Unprocessed sub-matrices

Figure 4: Stages and phases of the tiled FW algorithm introduced by Katz and Kider [KK08].

1. Start one work group: The work group loads the submatrix (i, i) as pivot matrix into local memory, where i is the current stage number. Then, only one thread in this work group calculates the transitive closure for this matrix using the original Floyd-Warshall algorithm.
2. Start $(m-1) \cdot 2$ work groups: Each work group loads the pivot matrix (i, i) and a second sub-matrix (j, i) or (i, j) into local memory. Now the second submatrix is located in the same row or column as the pivot matrix. Its calculation depends only upon itself and the pivot matrix. Within a work group, each value in the second sub-matrix can be computed in parallel by an own thread executing a part of the Floyd-Warshall algorithm (for more details see [KK08]).
3. Start $(m-1)^2$ work groups: Each work group loads two sub-matrices that have been processed in phase 2 and a third sub-matrix into local memory. The third matrix for two previously processed matrices (k, i) and (i, j) is placed at (k, j) and only depends upon their values and itself in this step. Again, all values of the third matrix can be processed in parallel by an own thread executing a part of the Floyd-Warshall algorithm.

In the following, we adapt the approach of Katz and Kider to the specific task of clustering duplicate pairs and add some modifications to improve the algorithm's efficiency and scalability.

5.5.1 Optimizing transitive closure efficiency

The adjacency matrix defines a directed graph, whereas our result graph is undirected, as we assume a symmetric duplicate relation between different records. Thus, all values in the adjacency matrix are mirrored across the matrix's diagonal axis. An obvious optimization approach is to remove redundant edges and hence reduce both the matrix size and the necessary computation steps in Phases 2 and 3 of the tiled FW algorithm. In Phase 2, for example, the algorithm could compute only the sub-matrices (i, j) with $j > i$ and (j, i) with $j < i$. Nevertheless, the overall performance would decrease for two reasons: First, the computation of the edge position in the matrix becomes more complex. Whenever the algorithm needs to read an edge value from the redundant (and therefore not existing) half of the matrix, it must mirror the edge's coordinates to find the corresponding value, which is a complex operation especially in Phases 2 and 3. Second, Warshall's algorithm might write the edges (x_i, x_j) and (x_j, x_i) at the same time. To guarantee consistent write operations, the kernels would need locking mechanisms, which decrease performance and restrains parallelism. Thus, we retain the original matrix and store each duplicate pair as two directed edges in the adjacency matrix.

The original tiled FW represents each value in the adjacency matrix as a single numerical value. To reduce the physical size of the matrix in memory, our implementation of the algorithm encodes these values as bitmasks: Each bitmask contains 32 edge values, because common GPUs address 32 bits at once. This technical optimization reduces the required memory by 1/32 compared to integers. However, this compression also impacts the structure of the algorithm: While computing the transitive closure, Floyd-Warshall's algorithm iterates over multiple rows and columns of the matrix. Each read operation returns 32 edge values. A horizontal iteration over a row containing bitmasks of edge values can be done very fast, because it needs $\lceil n/32 \rceil$ read operations to receive n edge values. In contrast, a vertical iteration over a column of the matrix still needs n read operations for n edges and returns $31 \cdot n$ not required values. This becomes a drawback for the performance, if we execute the Floyd-Warshall algorithm on a bit-compressed graph matrix. Warren's algorithm [Wa75], which extends the Floyd-Warshall algorithm, solves this problem by just iterating horizontally in the adjacency matrix. So we use this approach instead of Warshall's algorithm to calculate Phase 1 without iterating vertically. In Phases 2 and 3, the algorithm can use the redundant edges in the adjacency matrix to avoid vertical iterations. Each column c in the matrix has a corresponding row r that is mirrored across the matrix's diagonal axis and contains the same bit values. Therefore, all iterations over c can be replaced by iterations over r.

Using bitmasks to encode the matrix also affects the granularity of parallelization. In Phases 2 and 3 the algorithm can no longer compute the value of each single edge in parallel. To guarantee consistent writes, each bitmask must be processed by one GPU thread. However, by using bitwise *OR* operations for the comparison of two bitmasks, each thread computes all 32 values at once.

Figure 5 shows how all previously described modifications of the tiled FW work together in Phase 2. In this phase, each work group loads the pivot and a second submatrix into local memory. Then, all bitmasks in the second sub-matrix are computed in parallel.

Let $(x_k, y_i) - (x_k, y_j)$ be a bitmask b in the second submatrix. The thread that processes b iterates over row x_k in the pivot matrix and analyses each bit. If a bit (x_k, x_l) is 1, the thread loads the bitmask $(x_l, y_i) - (x_l, y_j)$ from the second matrix and then compares it to b using the bitwise *OR* operation. After analyzing the whole row x_k in the pivot matrix, the thread writes the new values for *b* into the second matrix. This algorithm also works for Phase 3. In this phase, three sub-matrices are loaded into local memory. To compute the bitmask *b* in the third matrix, a thread iterates over the corresponding row in the horizontally deferred second matrix and loads bitmasks for the comparison from the vertically deferred second matrix.

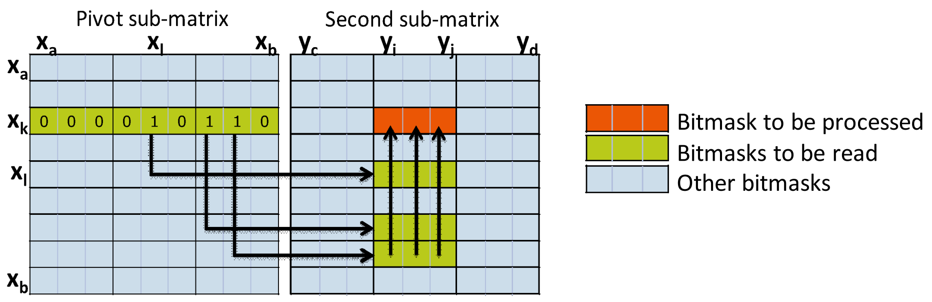


Figure 5: Optimized calculation of Phase 2 using bitmask encoding, horizontal iteration and bitwise OR comparison.

5.5.2 Achieving scalability

The algorithm of Katz and Kider assumes that the entire adjacency matrix fits into the GPU's global memory. Given a GPU with 1GB of global memory, this assumption limits the maximum number of nodes in the result graph to 92,672 even if bitwise encoding is used. Assuming 5% duplicates as result size, this is not enough to analyze datasets with 2 million records or more. Therefore, we need an additional partitioning of the matrix between the host's main memory and the GPU's global memory. We achieve this partitioning by using the same stage-wise execution strategy of the tiled FW again to pre-partition the global adjacency matrix G into smaller, quadratic matrices M_i on the host. The algorithm has to ensure that all matrices M_i are equally large and that three matrices M_i fit into the global memory at once. We call this approach the double tiled FW algorithm. It uses the same stages and phases of loading matrices M_i into global memory like the original tiled FW loads sub-matrices into the local memory. In Phase 1, only one pivot sub-matrix M_i resides in global memory. The GPU processes this matrix by executing the already known tiled FW. In Phase 2, the algorithm loads the pivot and a second sub-matrix into global memory. All bitmasks in the second sub-matrix are then processed in parallel like in Stage 2 of the tiled FW (see Fig. 5). Afterwards, the same procedure is used for Phase 3, which needs one pivot and two previously processed second sub-matrices.

6. Evaluation

We evaluated performance and accuracy of our workflow using real-world data sets. In addition, the execution time of each component is evaluated on different hardware.

6.1 Experimental setup

We evaluated on four different graphics cards, two from NVIDIA and two from ATI. As ATI's OpenCL drivers also allow the execution of OpenCL kernels on CPUs, we additionally evaluated our implementation on two Intel CPUs (see Tab. 2 for specifics of all six devices).

We used a subset of 1.792 million music CDs extracted from freedb.org for the performance evaluation of our algorithms. This dataset contains attributes artist, title, genre, year of publication, and multiple tracks. The DuDe Duplicate Detection Toolkit [DN10] provides a gold-standard for a randomly selected subset of 9,763 CDs (http://www.tinyurl.com/dude-toolkit), which we used to measure the accuracy of our results. Furthermore, we calculated the similarity of two records based on the values of four attributes that contain strings of variable length, namely Artist, Title, Track01, and Track02. This selection is based on our experience with that database.

To ensure a realistic assessment of the workflow efficiency, we first evaluated its effectiveness. We calculated precision (proportion of retrieved real duplicates), recall (proportion of identified real duplicates), and F-measure (harmonic mean of precision and recall) for different configurations: Sorted Neighborhood (SNM) and Cartesian product (CP) for pair selection combined with Levenshtein (L) and Jaro-Winkler (JW) as comparison algorithms. Table 1 lists the configuration parameters that delivered the best F-measure, showing similar results compared to other duplicate detection tools [DN10]. In Sec. 6.2, we use these configuration parameters to test the performance of our algorithm

For SNM, we tested window sizes between 10 and 500. We observed that any value above 20 has only minimal effect on the F-measure (at best 2 percentage points increase). Therefore, all experiments used a window size of 20. For the SNM's sort key generation, we tested two different generating algorithms. As already mentioned in Sec. 4.2, the Soundex algorithm generates the best sort keys for attributes whose values have similar lengths, which is true for the artist and track attributes. The values of the title attribute, however, vary considerably in length. As a result, our own key generation algorithm performs better for these attributes.

We tried multiple thresholds to determine whether a pair with a certain similarity is classified as a duplicate. The thresholds are first applied to attribute pairs during comparison and afterwards to record pairs during aggregation. The aggregation step additionally uses a set of weights to sum up the single attribute similarities. We evaluated various sets of thresholds and weights and settled on the values in Tab. 1.

Method	**Thresholds**				**Weights**			**Precision**	**Recall**	**F-Measure**
	Overall	**Artist**	**Title**	**Tracks**	**Artist**	**Title**	**Tracks**			
SNM + L	0.6	0.6	0.6	0.5	20%	30%	25%	95.2%	80.3%	87.1%
SNM + JW	0.66	0.6	0.67	0.87	20%	30%	25%	95.2%	79.6%	86.7%
CP + JW	0.66	0.78	0.75	0.87	20%	30%	25%	92.2%	86.6%	89.3%

Table 1: Configurations and results

ID	Type	Device Name	Clock	Memory	Cores	System	Price (August 2011, http://www.alternate.de)
G1	GPU	Nvidia GeForce GTX 570	732 MHz	1280 MB GDDR5	480 CUDA	Win64	279 Euro
G2	GPU	Nvidia Tesla C2050	1147 MHz	3071 MB GDDR5	448 CUDA	Linux64	2,149 Euro
G3	GPU	ATI Radeon HD 5700	850 MHz	1024 MB GDDR5	800 SP	Win64	91 Euro
G4	GPU	ATI Mobility Radeon HD 5650	450 MHz	1024 MB GDDR3	400 SP	Win64	*unknown*
C1	CPU	Intel Core i5 750	2.67 GHz	8192 MB DDR3	4	Win64	185 Euro
C2	CPU	Intel Core i5 M560	2.67 GHz	8192 MB DDR3	2	Win64	200 Euro

Table 2: Evaluation devices

6.2 Algorithmic complexity

To evaluate the performance of the duplicate detection workflow, we analyzed the execution times of its individual components. All tests were executed on the NVIDIA GeForce GTX 570 (G1), because our experiments in Sec. 6.3 show that this device performs best.

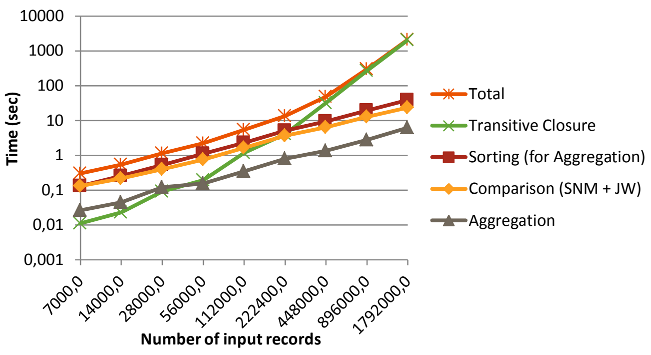


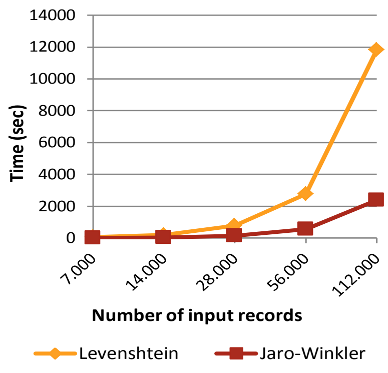
Figure 6: Execution times of different components

Figure 6 shows the execution times of the different components as parts of the complete workflow for various input sizes n. We used Jaro-Winkler for comparison and Sorted Neighborhood for pair selection. In the following, l denotes the longest list of found attribute-wise duplicates after the comparison, and m denotes the number of record-wise duplicates after the aggregation step. Since we observed that l and m increase linearly in proportion to n, the complexities of the subsequent algorithms can be defined in relation to the input size n.

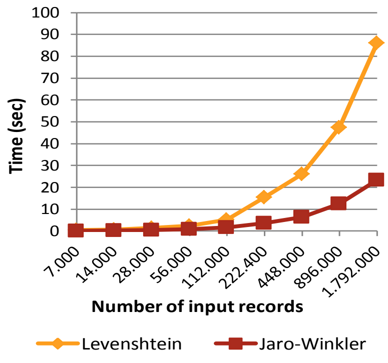
The diagram shows that with an increasing amount of data, and thus an increasing amount of duplicates, the execution time of the transitive closure becomes the dominant part of the workflow. Note that for a complete result one cannot omit this last step and that its complexity is hardly dependent on the total number of previously found duplicates, but rather on the number of disjoint records in the duplicates. The execution time of the transitive closure increases fastest, because its complexity is $O(n^3)$, whereas the other complexities are $O(n^2)$ for the Cartesian product, $O(n \log(n))$ if $w \leq \log(n)$ or otherwise $O(w \cdot n)$ for the Sorted Neighborhood, and $O(n \log(n))$ for the aggregation.

The sorting, as an aggregation preprocessing step, has the second highest time; more advanced algorithms [SKC10] might improve this value. The comparisons also have high execution times, because a string comparison is the most complex calculation on the GPU. The aggregation step itself has the smallest execution time and thus has only little impact on the workflow's overall execution time.

Figure 7 shows the execution times for the comparators only. We observe that Jaro-Winkler has a much lower execution time for both pair-selection algorithms for three reasons: First, Levenshtein performs more accesses to global GPU-memory. Second, Levenshtein performs more comparison rounds due to the higher memory consumption; these rounds need additional time to be triggered by the host. Third, Jaro-Winkler creates more work items allowing the GPU to use memory latency hiding to optimize the execution.



Cartesian Product



Sorted Neighborhood

Figure 7: Execution times of comparison algorithms in combination with different pair selectors

6.3 Comparison of hardware

As discovered in Sec. 6.2, the most efficient configuration uses Sorted Neighborhood in combination with the Jaro-Winkler comparison algorithm. Figure 8 shows that the best results are indeed achieved on GPUs. The fastest GPU G1 (see Tab. 2) takes 35 minutes (2,095 seconds) to process 1.792 million entries; this is about 10 times faster than the fastest CPU C2, which takes 335 minutes. However, Tab. 2 shows that the CPUs in our experimental setup are cheaper than the used GPUs. To compare them in a fair way, we placed the execu-

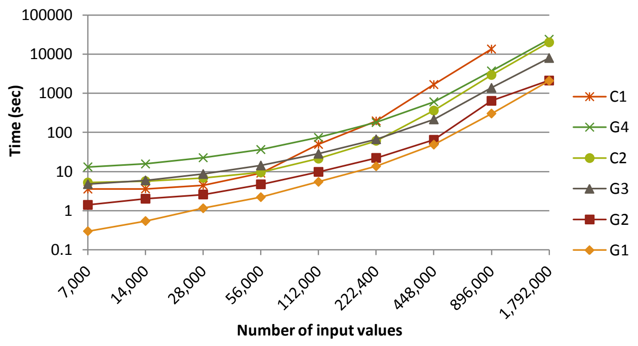


Figure 8: Performance of Sorted Neighborhood with Jaro-Winkler on different devices

tion times in relation to the prices by multiplying the price (in Euro) and the execution time (in minutes). This gives us a measure for the price-performance ratio, which assigns lower numbers to better devices. Since a GPU cannot be operated without a CPU, we add the price for the cheapest CPU. Still under this measure, the GPUs perform better than the CPUs: Again, for 1.792 million entries, G1 has the best results with a value of $(279 + 185) \cdot 35 = 16{,}240\ EuroMins$ compared to the best CPU C2 with a value of $200 \cdot 335 = 67{,}000\ EuroMins$; this is a 4-fold better price-performance ratio for the GPU.

7. Conclusion

We have presented and evaluated a complete duplicate detection workflow that uses graphics cards to speed up execution. The workflow uses either the Cartesian product or the Sorted Neighborhood approach for pair selection, and calculates the similarity of a record pair using Levenshtein, Jaro-Winkler, and Soundex. The evaluation of our workflow shows that modern GPUs can execute the duplicate detection workflow faster than modern CPUs. It has also been shown that the workflow and algorithms are scalable and can process large datasets.

The experiments also show that the access of global memory on graphics cards is indeed a bottleneck and has great impact on the performance of our algorithms. Profiling has shown that reads and writes are mostly non-coalesced and therefore very slow. To solve this problem in the future, all strings could be interlaced, which is a complicated task when using strings of variable lengths. Also, the use of local memory could further speed up execution. More optimizations concerning concrete hardware devices are possible and could be applied to a concrete usage of the workflow [FTP11]. Currently, only the comparisons of different attributes are distributed over all available devices. Thus, other algorithms, especially the computation of the transitive closure, could be further optimized to scale out on multiple devices. The implementation and evaluation of more similarity measures, e.g., token-based approaches, would allow the processing of real-world data with different properties and make the workflow more adaptable.

Acknowledgments: This research was supported by the HPI Future SOC Lab and the German Research Society (DFG grant no. NA 432). We thank Frank Feinbube (HPI) for his support.

References

[AJ88] R. Agrawal and H. V. Jagadish. Multiprocessor transitive closure algorithms. In *Proceedings of the first international symposium on Databases in parallel and distributed systems* (DPDS), 56-66, Los Alamitos, 1988.

[CCH10] P. Christen, T. Churches, and M. Hegland. Febrl - a parallel open source data linkage system. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining* (PAKDD), 638-647, Sydney, 2004.

[DN10] U. Draisbach and F. Naumann. DuDe: The duplicate detection toolkit. In *Proceedings of* *the International Workshop on Quality in Databases* (QDB), Singapore, 2010.

[EIV07] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering* (TKDE), 19(1):1-16, Piscataway, 2007.

[FTP11] F. Feinbube, P. Tröger, and A. Polze. Joint Forces: From Multithreaded Programming to GPU Computing. *IEEE Software*, 28(1):51-57, 2011.

[HS95] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. *In Proceedings of the ACM International Conference on Management of Data* (SIGMOD), 127-138, San Jose, 1995.

[HYF08] B. He, K. Yang, R. Fang, M. Lu, N.K. Govindaraju, Q. Luo, P.V. Sander: Relational joins on graphics processors. *In Proceedings of the ACM International Conference on Management of Data* (SIGMOD), 511-524, Vancouver, Canada, 2008.

[Ja89] M. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida. *Journal of the American Statistical Association*, 84(406):414-420, 1989.

[KK08] G. Katz and J. Kider Jr. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the ACM Symposium on Graphics Hardware* (SIGGRAPH), 47-55, Los Angeles, 2008.

[KKH10] T. Kirsten, L. Kolb, M. Hartung, A. Groß, H. Köpcke, and E. Rahm. Data partitioning for parallel entity matching. *Proc. of the VLDB Endowment*, 3(2), Singapore, 2010.

[KL07] H. Kim and D. Lee. Parallel linkage. *In Proceedings of the International Conference on Information and Knowledge Management* (CIKM), 283-292, Lisbon, 2007.

[KTR11] L. Kolb, A. Thor, and E. Rahm. Parallel sorted neighborhood blocking with mapreduce. In *Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik* (BTW), 45-64, Kaiserslautern, 2011.

[MNU05]V. Makinen, G. Navarro, and E. Ukkonen. Transposition invariant string matching. *Journal of Algorithms*, 56(2):124-153, 2005.

[ND10] J. Nickolls and W. Dally. The GPU computing era. *Micro*, IEEE, 30(2):56-69, 2010.

[NH10] F. Naumann and M. Herschel. *An Introduction to Duplicate Detection*. Morgan & Claypool, 2010.

[OLG07] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80-113, 2007.

[PS01] F. Patman and L. Shaefer. Is Soundex good enough for you? On the hidden risks of Soundex-based name searching. *Language Analysis Systems*, Inc., Herndon, 2001.

[SKC10] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proceedings of the ACM International Conference on Management of Data* (SIGMOD), 351-362, Indianapolis, 2010.

[To91] A. Toptsis. Parallel transitive closure computation in highly scalable multiprocessors. *Advances in Computing and Information* (ICCI), 197-206, Ottawa, 1991.

[USN07] The U.S. National Archives and Records Administration. The Soundex indexing system, May 2007. URL: http://www.archives.gov/research/census/soundex.html. Retrieved on Sept. 1, 2011.

[Wa62] S. Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9(1):11-12, 1962.

[Wa75] H. Warren Jr. A modification of Warshall's algorithm for the transitive closure of binary relations. *Communications of the ACM*, 18(4):218-220, 1975.

[WT91] W. E. Winkler and Y. Thibaudeau. An application of the Fellegi-Sunter model of record linkage to the 1990 U.S. decennial census. In *U.S. Decennial Census. Technical report, US Bureau of the Census*, 11-13, 1991.

[YF83] E. J. Yannakoudakis and D. Fawthrop. The rules of spelling errors. *Information Processing and Management*, 19(2):87-99, 1983.