High-Performance Probabilistic Record Linkage via Multi-Dimensional Homomorphisms

Ari Rasch
University of Münster, Germany
a.rasch@wwu.de

Richard Schulze
University of Münster, Germany
r.schulze@wwu.de

Waldemar Gorus
University of Münster, Germany
w.gorus@wwu.de

Jan Hiller
Krebsregister NRW, Germany
Jan.Hiller@krebsregister.nrw.de

Sebastian Bartholomäus
Krebsregister NRW, Germany
Sebastian.Bartholomaeus@krebsregister.nrw.de

Sergei Gorlatch
University of Münster, Germany
gorlatch@wwu.de

ABSTRACT

Probabilistic Record Linkage (PRL) identifies data records referring to the same real-world entity, e.g., in a database. PRL is increasingly used in epidemiology centers, intelligence agencies, and universities. However, PRL is a time-consuming task, which limits its applicability for large data sets in real-world applications.

We address the problem of accelerating PRL by parallelizing it for modern high-performance architectures, such as multi-core CPU and many-core GPU. Our approach relies on the formalism of *Multi-Dimensional Homomorphisms (MDHs)* – a class of functions with a generic parallel implementation in OpenCL. The schema allows for automatic optimization for a particular target hardware architecture by exploiting the auto-tuning approach. Our experiments show that we achieve significantly better performance on both CPU and GPU – speedups of up to 80 times – as compared to the parallel implementation of PRL that is currently used by EKR – the largest cancer registry in Europa.

CCS CONCEPTS

• **Applied computing**; • **Computing methodologies → Parallel computing methodologies**;

ACM Reference Format:
Ari Rasch, Richard Schulze, Waldemar Gorus, Jan Hiller, Sebastian Bartholomäus, and Sergei Gorlatch. 2019. High-Performance Probabilistic Record Linkage, via Multi-Dimensional Homomorphisms. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3297280.3297330

1 MOTIVATION AND RELATED WORK

Record Linkage (RL) [2] (a.k.a. *merge/purge processing*, *fuzzy matching*, *duplicate detection*, or *database hardening*) is the problem of identifying data records, e.g., in a database, that belong to the same real-world entity. RL is used in many important areas such as the management of: epidemiology centers, hospitals, universities, and intelligence agencies. For example, in the Epidemiologisches Krebsregister (EKR) of North Rhine-Westphalia, Germany – the largest cancer registry in Europe – record linkage is used to avoid adding duplicate entities to the patient data base. Duplicates can occur when the same patient is accidentally registered at EKR by different registration offices under different names; for example, Mary Smith (office 1) and Marie Smith (office 2).

A high-performance RL algorithm is easily implementable when entities have unique identifiers (a.k.a. deterministic record linkage [2]), e.g., numerical ids: two entities are considered as equal (i.e., they are *linked*) when their ids coincide. However, in most real-world applications, entities do not have unique identifiers. For example, in the EKR, patients are represented by 14 non-unique attributes, so-called *Quasi-IDentifiers (QIDs)*, including patient's first and last name, date of birth, and address. This makes duplicate detection challenging, e.g., because of typos in names (as described above) and/or changing addresses.

Probabilistic Record Linkage (PRL) [4] is the most prominent approach to record linkage on QID-represented entities. In PRL, a so-called *matching weight* is computed for comparing two records; the weight indicates: 1) *link* if it exceeds an *upper threshold*, 2) *non-link* if it is below a *lower threshold*, and 3) *possible-link* which has to be reviewed by a human if it is between the lower and upper threshold. The basic idea of PRL is to compute the matching weights based on probabilities – the probabilities are used for comparing QIDs on the basis of frequency ratios. For example, Mary Dijkstra and Marie Dijkstra have a higher probability to match than Mary Smith and Marie Smith, because surname Dijkstra has a lower frequency than Smith.

PRL has proven to be effective for many application areas [3]. However, PRL is a time-consuming approach, which limits its applicability for large real-world data sets: e.g., more than 5.4 million patient records are contained in the EKR's database, and in the research on interval carcinoma and early cancer diagnose, up to 1 million records have to be merged with the EKR's data base, requiring more than 7 days computation time when using EKR's current parallel PRL implementation for multi-core CPUs in JAVA.

The demand for high-performance PRL implementations has been recently identified as a research challenge [19]. There exist efficient approaches that bring PRL to the cloud [20, 21], but they miss the full performance potential of modern clouds that are increasingly equipped with multi-core CPU and many-core Graphics

Processing Units (GPU). Papers [5, 12, 17] present PRL implementations in OpenCL – an emerging standard for uniformly programming a wide range of parallel architectures such as multi-core CPU and GPU. However, these OpenCL implementations are optimized to achieve a high average performance over different architectures while missing the full performance potential of individual architectures. Approach [1] presents a CUDA-based implementation of PRL optimized for NVIDIA GPUs, but it is not applicable for devices from other vendors, e.g., Intel, AMD, or ARM.

In this paper, we develop an OpenCL implementation for PRL that – in contrast to the related work which provide only average good performance – provides high performance for each specifically target hardware architecture by being automatically tunable. For this, we exploit *Multi-Dimensional Homomorphisms (MDHs)* [16] – a recently defined formalism of parallelizable functions. Applications that are expressed as MDH can be implemented as efficient OpenCL code according to [16]. To enable implementation's automatic optimization, the MDHs' OpenCL code is parameterized with performance-critical parameters of the OpenCL's platform model – the number of threads (a.k.a. *work-item* in OpenCL) and the number of thread groups (a.k.a. *work-group)* – and thus, optimized values of these parameters can be determined for the target architecture by auto-tuning [15]. MDHs' OpenCL implementation provides high performance for important linear algebra routines (BLAS) on CPU and GPU [16].

We make the following new contributions:

(1) we show (in Section 3) that PRL can be expressed as MDH and thereby demonstrate the generality of the MDH approach which has so far been used for linear algebra routines;
(2) we show (in Section 4) that PRL can be efficiently executed on multi-core CPU and many-core GPU by designing a parallel, auto-tunable OpenCL implementation for PRL that is based on the MDHs' OpenCL implementation schema of [16];
(3) we experimentally evaluate (in Section 5) our parallel PRL OpenCL implementation on both Intel multi-core CPU and NVIDIA GPU by comparing it to the EKR's currently used parallel implementation of PRL for multi-core CPUs in JAVA.

2 PROBABILISTIC RECORD LINKAGE

Probabilistic Record Linkage (PRL) was introduced by Fellegi and Sunter [4]. In the following, we recap the theoretical foundation of PRL, and we show how PRL is used in the EKR cancer registry.

Let A and B be the two sets, e.g., of new patient records (A) and existing patient records (B), whose intersection of duplicates has to be identified. Records may be duplicates (i.e., refer to the same real-world entity) but differ in their QID values; for example, when there is a typo in patient's forename: Mary vs. Marie. We denote the set of duplicate records as M and the corresponding set of non-duplicates as U:

$$M = \{\, (a,b) \in A \times B \mid a \equiv b \,\}$$

$$U = \{\, (a,b) \in A \times B \mid a \not\equiv b \,\}$$

Here, $A \times B = \{(a,b) \mid a \in A,\ b \in B\}$ is the cartesian product of A and B, and $a \equiv b$ means that records a and b refer to the same real-world entity. Note that $a \equiv b$ does not necessarily require that a and b coincide in their QID values (e.g., because of typos in names, etc.).

PRL compares QID values by using *matching weights* that are based on *matching/unmatching probabilities*. A matching probability m_i^x is a function that for records a and b yields the probability that these records coincide in their i-th QID value and that both records are equal to value x, given that $(a,b) \in M$ (i.e., the records refer to the same real-world entity):

$$m_i^x(a,b) = \mathbf{P}(\, a_i = b_i = x \mid (a,b) \in M \,)$$

The unmatching probability u_i^x is defined as the probability that both records coincide in their i-th QID value and are equal to value x (as before) but, in contrast to the matching probability's definition, the records refer to different entities, i.e., $(a,b) \in U$:

$$u_i^x(a,b) = \mathbf{P}(\, a_i = b_i = x \mid (a,b) \in U \,)$$

In EKR, the matching/unmatching probability functions m_i^x and u_i^x are defined by value tables, i.e., EKR provides for each function a table that assigns to a combination (x,i,a,b) the probability $m_i^x(a,b)$ or $u_i^x(a,b)$, correspondingly. EKR determines these probabilities empirically by analyzing existing real-world data bases.

Using the matching/unmatching probabilities, matching weight w_i of a and b in the i-th QID is a real number that is calculated as:

$$w_i(a,b) = \begin{cases} log(\frac{m_i^x(a,b)}{u_i^x(a,b)}) & : a_i = b_i \wedge x = a_i \\ log(\frac{1-m_i^x(a,b)}{1-u_i^x(a,b)}) & : a_i \neq b_i \ \wedge x = a_i \end{cases}$$

The matching weight w of a and b is then the sum of the $w_i(a,b)$:

$$w(a,b) = \sum_{i=1}^{N} w_i(a,b)$$

Here, N is the number of QIDs, e.g., $N = 14$ in case of EKR's patient records (the EKR's 14 QIDs in Table 1 are discussed later). The matching weight is defined to be high when QID values coincide that have a low frequency in EKR's data base. For example, surname Dijkstra has quite a low frequency, therefore, its matching probability $m_{\texttt{surname}}^{\texttt{Dijkstra}}$ is high, and its unmatching probability $u_{\texttt{surname}}^{\texttt{Dijkstra}}$ is low; both contribute to a high matching weight of records with surname Dijkstra. In contrast, surname Smith has a high frequency – and thus, as compared to surname Dijkstra, it has a lower matching probability and a higher unmatching probability – causing a lower matching weight for records with surname Smith. Note that in the definition of $w_i(a,b)$, in case of $a_i \neq b_i$, it would also be possible to set $x = b_i$ instead of $x = a_i$. In [8], it is shown that both variants usually lead to similar results.

The matching weight w is used to categorize a pair $(a,b) \in A \times B$ as either i) link, ii) non-link, or iii) possible-link; for this, an upper and a lower threshold value are used which are in case of EKR: 15 (lower) and 45 (upper), correspondingly. If a record pair's matching weight i) exceeds the upper threshold, then the weight indicates a link, and the records are considered as duplicates; ii) is below the lower threshold value, then a non-link is indicated, and the records are considered as non-duplicates; iii) is between the upper and lower threshold, then a possible-link is indicated, and the records have to be manually checked by a human. The thresholds are usually determined empirically and must be chosen carefully. For example, setting the upper and lower threshold values close to each other minimizes the costly manual checking of patient records by a human, but it may cause wrong link/no-link predictions.

As weight calculations are costly, PRL usually uses a so-called *blocking strategy* [2] to reduce them: a record $a \in A$ is only compared to – and thus matching weights are only calculated for – those $b \in B$ for which a and b exactly coincide in a pre-defined set of QIDs. For example, EKR uses a 7-staged blocking strategy. In Stage 1, the matching weight of $a \in A$ is only calculated for those $b \in B$ for which the following QIDs of a and b coincide: surname, forename, and birthday; the set of QIDs is selected by the EKR experts, based on empirical data. If a record $b \in B$ is found for which $(a, b) \in M$, i.e., a and b are considered as duplicates, a is not compared to the further $b \in B$ (i.e., which differ from a in the surname, forename, and/or birthday) and thus, costly weight calculations are omitted. If no duplicate is found in Stage 1, a is compared in Stage 2 to those $b \in B$ which coincide with a in the QIDs: surname, forename and address. This procedure continues in Stages 3-7; each stage is characterized by a selection of QIDs that is determined empirically by the EKR experts. After Stage 7, if no duplicate $b \in B$ is found, record a is considered to have either no duplicates in B if the highest computed matching weight of a in Stages 1-7 – denoted as $w(a, b_{\texttt{max\_a}})$ for the corresponding $b_{\texttt{max\_a}} \in B$ – is below the lower threshold, and a and $b_{\texttt{max\_a}}$ are considered as a possible-link if matching weight $w(a, b_{\texttt{max\_a}})$ is between lower and upper thresholds.

Table 1 shows the 14 QIDs that are currently used by EKR for representing a patient. The QIDs for the names consist of three parts, in order to cover patients with multiple names, e.g., two forenames. The birth date is given by the day/month/year of birth, and a patient's address is represented by its municipality key.

No.	QID	m_i
1	Surname 1	0.975
2	Surname 2	0.975
3	Surname 3	0.975
4	Forename 1	0.975
5	Forename 2	0.975
6	Forename 3	0.975
7	Birth name 1	0.975
8	Birth name 2	0.975
9	Birth name 3	0.975
10	Day of birth	0.99
11	Month of birth	0.99
12	Year of birth	0.99
13	Gender	0.999
14	Municipality key	0.9

Table 1: The 14 QIDs used by EKR for representing patients, and their averaged matching probabilities m_i.

Table 1 also shows the QIDs' *averaged matching probabilities* m_i, which EKR uses instead of the matching probabilites m_i^x. An averaged matching probability m_i represents the matching probability m_i^x averaged over the x value. For example, $m_{\texttt{forename}}$ represents the probability that two records referring to the same real-world entity have the same forename, where – according to the averaged matching probability's definition – the forename is arbitrary (i.e., it does not have to be equal to a concrete forename x). EKR uses the averaged matching probability m_i because data bases containing duplicate records are rare but are required for determining matching probabilities (as discussed before); as averaged matching probabilities, EKR uses pre-computed values [8]. The highest probability – and thus the highest impact on the matching weight – has the gender QID (see Table 1): the gender usually does not change for a person, and its identifying for a person is usually not error-prone. The unmatching probabilities u_i^x are pre-computed by EKR based on the patient data base. As QIDs may have many possible values x, e.g., the forename QID, EKR does not provide a pre-computed u_i^x for each possible x; for the missing u_i^x, EKR uses a default probability. The unmatching probabilities u_i^x are not stated in Table 1 for brevity – for each QID, they are defined once per each (of the many) possible QID values x.

3 PROBABILISTIC RECORD LINKAGE AS A MULTI-DIMENSIONAL HOMOMORPHISM

Multi-dimensional homomorphisms (MDHs) are a class of parallelizable functions introduced in [16] – they extend the traditional (one-dimensional) homomorphisms on lists [6]. In this section, we first demonstrate that PRL can be expressed as an MDH. Based on this, we show in Section 4 how the MDHs' implementation schema in [16] can be used to efficiently parallelize PRL.

MDHs operate on multi-dimensional arrays and are defined as follows. Let T and T' be two arbitrary data types, and let further $T[\,N_1\,]\ldots[\,N_d\,]$ be the set of d-dimensional arrays with elements in T and size N_i in dimension i. A function $h : T[\,N_1\,]\ldots[\,N_d\,] \to T'$ on d-dimensional arrays is a *multi-dimensional homomorphism (MDH)* iff there exist *combine operators* $\circledast_1, \ldots, \circledast_d : T' \times T' \to T'$, such that for each $1 \le i \le d$ and arbitrary, concatenated input array $a +\!\!+_i\, b$ in dimension i, the *homomorphism property* is satisfied:

$$h(\,a +\!\!+_i\, b\,) \;=\; h(a) \circledast_i h(b)$$

In words: the value of h on a concatenated array in dimension i can be computed by applying h to the array's chunks a and b and combining the results by using the combine operator $\circledast_i$. Since the computations of $h(a)$ and $h(b)$ are independent of each other, they can be performed in parallel.

In [16], it is proved that every MDH h can be computed as:

$$h(\,a\,) = \underset{1 \le i_1 \le N_1}{\circledast_1} \;\ldots\; \underset{1 \le i_d \le N_d}{\circledast_d} \; f(\,a[\,i_1\,]\ldots[\,i_d\,]\,)$$

Here, $a \in T[N_1]\ldots[N_d]$, and $a[\,i_1\,]\ldots[\,i_d\,]$ is the element in a that is accessed by the indices $i_1, \ldots, i_d$. Function f describes the behavior of h on scalar values, i.e., $f(\,a[0]\ldots[0]\,) = h(a)$ for each d-dimensional array a comprising only one element (i.e., a has size 1 in each of its d dimensions). Intuitively, in the formula, we apply f to each of the input array's scalar values $a[\,i_1\,]\ldots[\,i_d\,]$, and we combine the obtained results, step by step, in the dimensions $1, \ldots, d$ using the combine operators $\circledast_1, \ldots, \circledast_d$; combine operators can be applied in any arbitrary order [16]. Concluding, each MDH is completely determined by its combine operators $\circledast_1, \ldots, \circledast_d$ and its behavior f on scalar values. This enables expressing h also as:

$$h = \texttt{md\_hom}(\,f,\; (\circledast_1, \ldots, \circledast_d)\,)$$

Note that `md_hom` can be computed in parallel – function f can be applied independently to the scalar values, and the combination of the results can also be performed in parallel using parallel reduction [13] – thus the `md_hom` function can be viewed as a *parallel pattern* (a.k.a. *algorithmic skeleton* [7]) as described in detail in [16].

The PRL problem can be expressed using the `md_hom` pattern:

$$\texttt{PRL} = \texttt{md\_hom}(\,\texttt{weight},\ (\mathbin{+\!\!+},\ \texttt{max}_{\texttt{mw}}\,)\,) \circ \texttt{cart} \qquad (1)$$

where $\circ$ denotes functional composition applied from right to left. Intuitively, in (1), we first form the set of all pairs $(a, b) \in A \times B$ by applying function `cart` to A and B. Then, function `weight` computes for each pair (a, b) the matching weight $w(a, b)$ (according to our formula in Section 2). For each record a, we combine the computed weights $w(a, b)$ for the different $b \in B$ via function $\texttt{max}_{\texttt{mw}}$. Thus, we obtain for each $a \in A$ the maximum matching weight, i.e., $w(a, b_{\texttt{max\_a}})$ for the corresponding $b_{\texttt{max\_a}} \in B$ that leads to the highest weight of a. As we need the maximum weights for all $a \in A$, we straightforwardly concatenate their computed and combined weights using $\mathbin{+\!\!+}$. Note that function `weight` represents PRL's behavior f on scalar values, and functions $\mathbin{+\!\!+}$ and $\texttt{max}_{\texttt{mw}}$ are PRL's combine operators $\circledast_1$ and $\circledast_2$, correspondingly. Therefore, as discussed before, we can apply function `weight` in parallel to the scalar values (in this case, pairs of patient records), and we can combine function `weight`'s results in parallel by its combine operators – our parallel implementation of PRL is focus of Section 4.

In the following, we discuss in detail the building blocks of our PRL `md_hom` expression (1) – `cart`, `weight`, $\mathbin{+\!\!+}$, and $\texttt{max}_{\texttt{mw}}$ – and we present their implementations which we use in Section 4 in our OpenCL implementation of the `md_hom` expression (1).

Function `cart`. MDHs take as input multi-dimensional arrays. However, in case of PRL, the input are two sets A and B – for which the common duplicate patient records have to be identified. We use function `cart` for computing the cartesian product of A and B:

$$\texttt{cart}(A, B)[\,i\,][\,j\,] = (\,A(i),\ B(j)\,)$$

It takes as input the sets A and B, and it yields a 2-dimensional array $cart(A, B)$ comprising pairs $(\,A(i),\ B(j)\,)$, where $A(i)$ is the i-th patient record in set A, and $B(j)$ the j-th record in set B.

We implement function `cart` in OpenCL as a straightforward preprocessor macro, as follows:

```
#define cart( A,B , i,j, c ) ( (c==0) ? A(i) : B(j) )
```

The macro takes as input the sets A and B, the indices i and j, and the desired component of pair $(\,A(i),\ B(j)\,)$, i.e., $c = 0$ for $A(i)$ and $c \neq 0$ for $B(j)$. As preprocessor macros in OpenCL are resolved at compile time, function `cart` has no impact on runtime performance. Function `cart`'s impact on compile time is also negligible: it only performs the simple check `c==0`.

Function `weight`. Function `weight` computes the matching weight of two patients a and b (according to Section 2) – it represents the behavior of f on scalar values in our PRL `md_hom` expression (1). It takes as input pairs of patient records in $A \times B$ – the result of function `cart` – and it yields matching weights as real numbers.

Listing 1 shows the pseudocode of our OpenCL implementation of the `weight` function. We implement `weight` as a sequential OpenCL code which is called as helper function in our parallel

```
1  result_t weight( const patient  a,
2                   const patient  b,
3                   const prob     up,
4                   const prob     a_mp )
5  {
6    result_t res = { a,b , 0, NON_MATCH };
7    // blocking strategy
8    bool match = false;
9    // 1st stage
10   if (!match) {
11     match =    eq_str( a.surname ,  b.surname  )
12             && eq_str( a.forename , b.forename )
13             && a.birth_day   = b.birth_day
14             && a.birth_month = b.birth_month
15             && a.birth_year  = b.birth_year;
16   }
17   /* Stages 2-6  */
18   // 7th stage
19   if (!match) {
20     match = /* ... */
21   }
22   // check blocking result
23   if (!match) return res;
24    // compute weight
25   double weight = 0.0;
26   weight += /* compute matching weight 1  */
27   // ...
28   weight += /* compute matching weight 14 */
29   res.matching_weight = weight;
30   // categorize weight
31   if( weight > UPPER_THRESHOLD )
32     res.match_status = MATCH;
33   else if ( weight < LOWER_THRESHOLD )
34     res.match_status = NON_MATCH;
35   else
36     res.match_status = POSSIBLE_MATCH;
37   return res;
38 }
```

Listing 1: OpenCL implementation (pseudocode) of function `weight` for computing two patients' matching weight.

OpenCL implementation which we introduce in Section 4. The input of function `weight` are two patient records a and b for which the matching weight should be computed (Listing 1, lines 1 and 2). Moreover, `weight` takes as input the input records' corresponding unmatching and averaged matching probabilities (lines 3 and 4) which are required for computing the matching weight (see Section 2).

In the first part of our implementation (lines 7-23), we check if a and b belong to the same block in terms of one of the 7 stages of EKR's blocking strategy described in Section 2. For example, in the first stage (lines 9-16), records a and b belong to the same block if they coincide in their QIDs for the surname (line 11), forename (line 12), and birthday (lines 13-15). If a and b do not belong to the same block in one of the blocking strategy's 7 stages, the costly computation of the matching weight for a and b is omitted (line 23), and a and b are considered as non-link with a matching weight of 0 (lines 23 and 6). If a and b belong to the same block for one of the stages, their matching weight is computed – this is done by computing and summing up the QIDs' matching weights $w_1(a, b),\ \ldots,\ w_{14}(a, b)$ for each of the EKR's 14 QIDs (lines 24-28), according to our formula for the matching weight $w(a, b)$ in Section 2; the computed weight is written to result element `res` (line 29). We also compute and add to `res` the information if a and b are considered as a link, non-link or possible-link by comparing the computed weight to the upper and lower thresholds (lines 30-36).

Concatenation ($+\!\!+$). Function $+\!\!+$ is used as the first combine operator in our `md_hom` expression for PRL (1), i.e., $\circledast_1 = +\!\!+$. It takes as input two one-dimensional arrays, and it yields the concatenation of these two arrays. In our OpenCL kernel for (1) (introduced in Section 4), we use concatenation only on arrays whose content is computed. Thus, concatenation means writing the computed array's content consecutively to memory (as we demonstrate later), i.e., it does not require an additional computation effort.

Comparison Function ($\mathtt{max}_{\mathtt{mw}}$). Function $\mathtt{max}_{\mathtt{mw}}$ is used as the second combine operator of our `md_hom` expression (1) for PRL, i.e., $\circledast_2 = \mathtt{max}_{\mathtt{mw}}$. It takes as input two return values of function `weight`: (a, b, w, L) and (a', b', w', L'), where w is the computed matching weight for patients a and b, and w' is the weight for a' and b', correspondingly; L and L' indicate a link, non-link or possible link between the patients. The result of $\mathtt{max}_{\mathtt{mw}}$ is (a, b, w, L), if $w \geq w'$, and (a', b', w', L') otherwise. We refrain from presenting our OpenCL implementation for function $\mathtt{max}_{\mathtt{mw}}$ as it is straightforward.

4 PROBABILISTIC RECORD LINKAGE IN OPENCL VIA THE MDH APPROACH

OpenCL is currently a de-facto standard for uniformly programming modern parallel devices such as multi-core CPU and many-core GPU. OpenCL provides to the programmer a 2-layered thread hierarchy: in the first layer, the programmer distributes computations to *work-groups (WGs)*, and in the second layer, the computations of each WG are further distributed to *work-items (WIs)* – the OpenCL term for thread. Usually, on a multi-core CPU, WGs are scheduled to the CPU's cores, and WIs are scheduled to the cores' Single Instruction Multiple Data (SIMD) vector processing units [9]; on a GPU from NVIDIA, WGs are processed by so-called *Streaming Multiprocessors* (a.k.a. *SMX*) and WIs are processed by *CUDA cores* which are part of the SMX [14].

The numbers of WGs and WIs are parameters that have to be set manually by the OpenCL programmer and are crucial for achieving high performance [15]: a too low number of WIs and WGs do not appropriately utilize the hardware, while too high numbers may cause a high parallelization overhead. Optimal numbers of WGs and WIs have to be chosen specifically for the target hardware, because hardware differs significantly in its characteristics: e.g., a GPU has many more cores than a multi-core CPU. For providing high performance on various hardware, we use auto-tuning [15] to automatically choose appropriate numbers of WGs and WIs.

Figure 1 outlines our OpenCL implementation for PRL which is based on the MDHs' implementation schema in [16]. According to Section 3, our PRL `md_hom` expression (1) can be computed as:

$$\mathtt{PRL}(\ \mathtt{cart}(A,B)\ ) = \mathop{+\!\!+}_{i_A \in [1,N_A]} \ \mathop{\mathtt{max}_{\mathtt{mw}}}_{i_B \in [1,N_B]} \ \mathtt{weight}(\ \mathtt{cart}(A,B)[\ i_A\ ][\ i_B\ ]\ )$$

where N_A and N_B denote the number of patient records in sets A and B, correspondingly. In words: function `weight` is applied to each pair of patient records in the 2-dimensional input array $cart(A, B)$, and the results of computing `weight` are combined in dimension 2 by function $\mathtt{max}_{\mathtt{mw}}$ and by concatenation $+\!\!+$ in dimension 1. For illustration, we use in Figure 1 input sizes N_A and N_B both of 8 elements. For each of the the input array $\mathtt{cart}(A, B)$'s two dimensions, we consider 2 WGs, each containing 2 WIs, i.e., we use in total 4 WGs, each comprising 4 WIs.

The input array $\mathtt{cart}(A, B)$ (marked with ① in Figure 1) comprises pairs of patient records, denoted by symbol $\times$. The pairs are evenly partitioned and distributed to the $2*2$ WGs (see ②), i.e., each WG processes (in parallel to the other WGs) a chunk of $4*4$ patient pairs, as shown in ③. For computing a WG-chunk, it is further partitioned and distributed to the $2*2$ WIs of the WG, as illustrated in ④. We distribute the WG-chunks to the WIs in a strided fashion (see ③ and ④), thereby enabling high performance: a strided distribution causes consecutive WIs to access consecutive memory regions, enabling either a higher memory bandwidth on GPUs (a.k.a. *memory coalescing* [14]) or efficient SIMD-parallelization on multi-core CPUs [9]. The WIs process simultaneously chunks of size $2*2$ (see ⑤) by applying the PRL `md_hom` expression (1) to their chunks, i.e.: each WI applies (in parallel to the other WIs) the `weight` function to the patient pairs of its chunk, and it combines the results of function `weight` in both dimensions by using the combine operators $\mathtt{max}_{\mathtt{mw}}$ and $+\!\!+$ as illustrated in ⑥. We obtain a 1-dimensional array per WI-chunk (see ⑦), containing the two combined and concatenated matching weights, each denoted by symbol $\circ$ in Figure 1. The processed WI-chunks – shown in ⑧ – are then further combined cooperatively by the WIs in parallel – using the combine operators $\mathtt{max}_{\mathtt{mw}}$ and $+\!\!+$ – to one result per WG (see ⑨). Finally, the WGs' results in ⑩ have to be combined, analogously to before, in both dimensions using again our two combine operators $+\!\!+$ and $\mathtt{max}_{\mathtt{mw}}$, leading to the final result ⑪. Note that our implementation is correct because of the MDHs' homomorphic property: it ensures correctness when applying the MDH to smaller chunks and then combining the obtained results by the MDH's combine operators – this is formally proven in [16].

We enable our implementation to be auto-tunable by making it parameterized in the number of WGs and WIs, the optimized parameter values are determined via auto-tuning [15]. For example, another valid configuration in Figure 1 consists of 16 WGs, each containing 1 WI; it achieves high performance on a 16-core CPU that contains no SIMD-units for processing multiple WIs.

Listing 2 is the pseudocode of our OpenCL implementation (a.k.a. *kernel* in OpenCL terminology) for the PRL `md_hom` expression (1) which we have illustrated in Figure 1. This kernel code is executed by the individual WIs in a Single-Program Multiple-Data (SPMD) manner. Each WI processes its corresponding WI-chunk (step ⑥ in Figure 1) in parallel to the other WIs (Listing 2, lines 4-12); the WI-chunk is part of the WI's corresponding WG-chunk – the partitioning is shown in steps ①-⑤ of Figure 1. The WG- and WI-chunks are obtained by straightforwardly computing indices (lines 9-10) based on offsets and the ids of the WIs and WGs; the indices are used to access the input array $cart(A, B)$ (line 11). We access the array in a strided fashion by multiplying `NUM_WI_B` to `WI_itr_B` in line 10; this leads to higher performance as discussed before. After the WI-chunks are processed (step ⑦), the WIs copy their results (line 13) to *local memory* – an OpenCL memory region

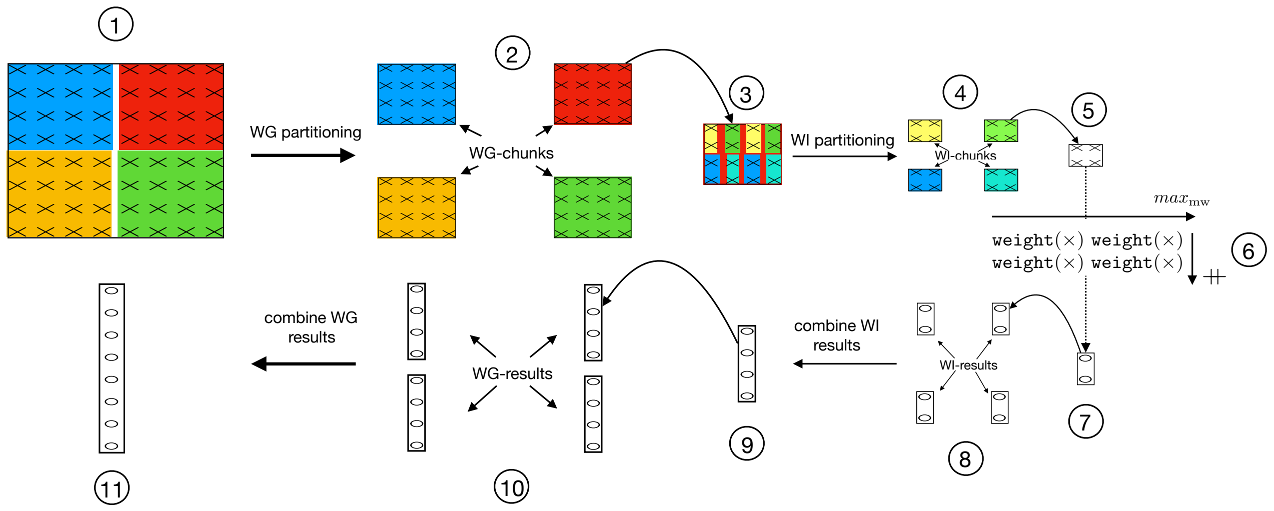

Figure 1: Schema of our OpenCL implementation for the PRL `md_hom` expression. We illustrate the case of: $|A| = |B| = 8$, and 2 WGs and 2 WIs in each of the two dimensions.

```
1   __kernel void prl( /* ... */ )
2   {
3     /* ... initialization ... */
4     // "md_hom( weight , (...) )" on WI-chunk
5     for( int WI_itr_A = 0 ; WI_itr_A < WI_CHUNK_SIZE_A ; ++
          WI_itr_A )
6     {
7       for( int WI_itr_B = 0 ; WI_itr_B < WI_CHUNK_SIZE_B ;
            ++WI_itr_B )
8       {
9         int index_A = WG_offset_A( WG_ID_A ) + WI_offset_A(
              WI_ID_A ) +  WI_itr_A * N_B;
10        int index_B = WG_offset_B( WG_ID_B ) + WI_offset_B(
              WI_ID_B ) +  WI_itr_B * NUM_WI_B;
11        res[ WI_itr_A ] max_mw= f( cart(A,B)[ index_A ][
            index_B ] );
12      }
13      res_lcl[ WI_ID_A ][ WI_ID_B ] = res;
14      barrier( CLK_LOCAL_MEM_FENCE );
15      // parallel reduction of WI results
16      for( int stride = NUM_WI_B / 2 ; stride > 0 ; stride
            /=2 )
17      {
18        if( WI_ID_B < stride )
19          res_lcl[ WI_ID_A ][ WI_ID_B ] max_mw= res_lcl[
              WI_ID_A ][ WI_ID_B + stride ];
20        barrier( CLK_LOCAL_MEM_FENCE );
21      }
22      /* ... store WGs' results ... */
23    }
24  }
```

Listing 2: OpenCL implementation (pseudocode) of our PRL `md_hom` expression (1).

that can be accessed by all WIs within a WG. The individual WIs' results (step ⑧) are then combined cooperatively by all WIs in parallel to one result per WG (lines 15-21; step ⑨ in Figure 1). For this, WIs' results are combined in the second dimension by using function $\mathtt{max}_{\mathtt{mw}}$ (line 19), and we write the combined results consecutively to memory. When writing consecutively to memory, we implicitly perform concatenation of the computed results – and thus the combination by ++ in the first dimension. Therefore, concatenation does not require an additional computation effort. The obtained results (step ⑩) – one per WG – have to be combined further to the final result (step ⑪). Since OpenCL does not support synchronization between WGs, we start a second kernel that takes first kernel's result as input. The second kernel combines the first kernel's WG-results (depicted in step ⑩), analogously to before (lines 15-21), in both dimensions using our two combine operators; we thus refrain from presenting the second kernel's pseudocode. We avoid synchronization between WGs in the second kernel by starting only one WG in the second dimension, which has an only negligible impact on performance: since first kernel's number of WGs in the second dimension is usually rather low – for example, on our target system, an optimal number of WGs is < 512 for both CPU and GPU – the second kernel combines only a low number of WG results and thus, its runtime is negligible as compared to the runtime of the first kernel (< 0.6%).

5 EXPERIMENTAL RESULTS

In the following, we experimentally study the efficiency of our PRL OpenCL implementation – presented in Section 4 – by comparing it to the EKR's current implementation that uses *JAVA multi-threading* for parallelization on multi-core CPUs. In contrast, our implementation is based on OpenCL and thus is executable on a broad range of modern processors, e.g., multi-core CPUs and also many-core GPUs (Graphics Processing Units) of different vendors.

Figure 2 illustrates the runtime of our OpenCL implementation in Listing 2 on Intel Xeon E3-1240 v2 4-core CPU (dark gray line) and NVIDIA Tesla V100-SXM2-16GB GPU (light gray line) as compared to the runtime of EKR's parallel implementation of PRL in JAVA (black line) which can only be executed on the multi-core CPU. We automatically optimize our OpenCL implementation for

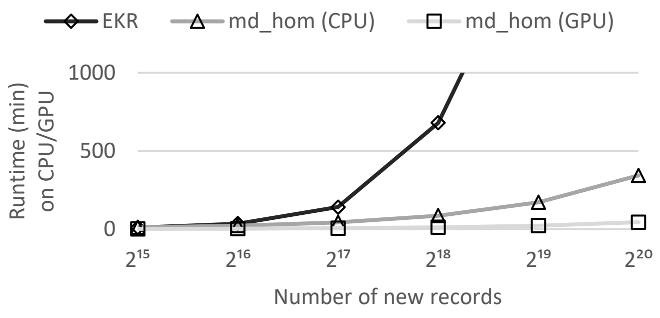

Figure 2: Runtime (in minutes) of our PRL OpenCL implementation (in Listing 2) on Intel multi-core CPU (dark gray line) and NVIDIA many-core GPU (light gray line) as compared to the EKR's parallel JAVA implementation (black line) which targets multi-core CPUs only. We show the median time of 30 runs. Our implementation provides significantly better performance on both CPU and GPU as compared to the EKR's implementation on the CPU.

the target device by choosing optimized numbers of WIs and WGs via auto-tuning (as discussed in Section 4). We use the *Auto-Tuning Framework (ATF)* [15] which has proven to be effective for OpenCL; it requires less than 24 hours of runtime for auto-tuning our PRL implementation. This additional overhead for auto-tuning is required only once per hardware device, i.e., the auto-tuned numbers of WGs and WIs can be reused for each execution of PRL on the same device.

In Figure 2, we show the results for merging $5.4 * 10^6$ patient records — the current amount of records in the EKR's data base – with increasing numbers of new patient records. As input data, we use real-world patient records provided by the EKR. For a fair comparison to JAVA, we measure for our OpenCL implementation the runtime of both, our two PRL kernels (see Section 4) and the OpenCL C++ *host code* [18] which is required for executing OpenCL kernels; the host code performs data transfers between main and device's memory, and it also just-in-time compiles the kernels.

We observe that our OpenCL implementation provides significantly better performance than EKR's JAVA implementation – with a speedup of 8 on the CPU and 80 on the GPU even for a small input size of 2^{18}. This is because on the CPU, we efficiently utilize its SIMD vector units: we access data in a strided fashion (see Section 4), and we automatically choose an optimized number of WIs via auto-tuning – 8 WIs in case of our CPU – enabling OpenCL to efficiently vectorize our code according to Intel's recommendation [9]. In contrast, the EKR's JAVA implementation parallelizes only for the CPU's cores, without exploiting SIMD units. Moreover, the EKR's implementation is not efficient in handling memory: it uses convenient high-level data structures such as JAVA's `ArrayList` which perform costly memory reallocations transparently from the user, while we rely on explicitly managed low-level OpenCL buffers which avoid reallocations.

On the GPU, performance is better because GPUs provide a significantly higher number of cores than the multi-core CPU: 5376 cores (GPU) vs. 4 cores (CPU). We thus reach on the GPU better results than both, the EKR's JAVA implementation and also our OpenCL implementation when optimized and executed on the CPU. Note that even though the GPU provides about 10^3 more cores than the CPU, our implementation on the GPU is on average (only) 10 times faster as compared to executing it on the CPU. This is because we highly optimize our code for the CPU via auto-tuning which determines optimized number of WIs and WGs to efficiently utilize the CPU's powerful cores – 3.4 Ghz (CPU) vs 1.5 Ghz (GPU) – and these optimized numbers enable exploiting CPU's SIMD vector units, which diminishes the expected (high) performance gap between CPU and GPU [11]. Furthermore, in contrast to the GPU, the CPU's architecture is better optimized for complex calculations – such as PRL's weight calculations (see Section 2) – e.g., by performing branch prediction, which also significantly contributes to a high performance of PRL on the CPU.

In Figure 3, we demonstrate the adverse effect of optimizing PRL for an only average high performance over hardware architectures – the usual approach of the related work which provide OpenCL implementations that cannot be auto-tuned. For this, we compare the runtime of our implementation when auto-tuned specifically for the target CPU/GPU against the OpenCL implementation for PRL that is optimized for an only average high performance over CPU and GPU. As averaged implementation, we use our auto-tunable

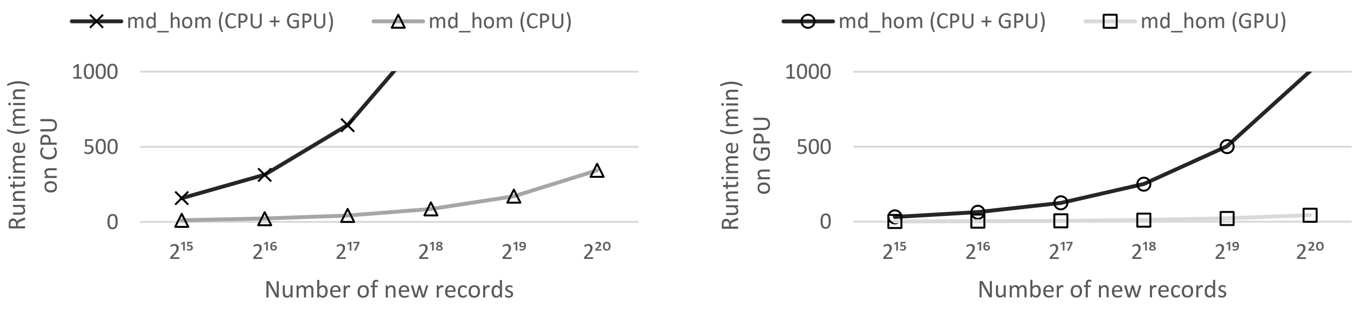

Figure 3: Runtime (in minutes) of our PRL OpenCL implementation (Listing 2) on Intel multi-core CPU (left) and NVIDIA many-core GPU (right) using average-good numbers of WIs and WGs for both CPU and GPU (black lines), as compared to our implementation with optimized numbers of WGs/WIs for either CPU (gray line, left figure) or GPU (gray line, right figure). We show the median time of 30 runs. Optimizing the numbers of WGs and WIs specifically for the target hardware (gray lines) provides significantly better performance as compared to using only average-good numbers of WGs and WIs (black lines).

OpenCL kernel in Listing 2 with fixed numbers of WIs and WGs (i.e., we do not auto-tune these numbers for the specific target device), according to the related work. We choose the number of WGs and WIs following the Intel's and NVIDIA's optimization guides [9, 14]. For the number of WIs on the CPU, Intel recommends to choose it as a multiple of the SIMD vector length – 8 in case of our CPU – to enable SIMD vectorization. NVIDIA recommends for its GPUs to use a multiple of the so-called *warp size* of 32 as the number of WIs. For the number of WGs, both guides recommended to choose it as (at least) one WG per core (a.k.a. SMX in NVIDIA terminology). Following these recommendations of Intel and NVIDIA, we choose as number of WIs 32 as this is a multiple of both the CPU's SIMD vector length of 8 and the GPU's warp size of 32, and we use 80 WGs as this is the maximum number of cores of both architectures – the Intel CPU has 4 cores and the NVIDIA GPU has 80 SMX.

We observe in Figure 3 that even though the number WIs and WGs are chosen according to vendors' optimization guides, we reach with them only poor results on both CPU (left) and GPU (right) as compared to our OpenCL kernel auto-tuned for the specific target device. This is because, for complex applications such as PRL, non-intuitive numbers of WIs and WGs often lead to best performance [10] due to further influencing factors, e.g., the number of used registers per WI/WG and/or cache utilization. For example, on the CPU, we have determined via auto-tuning the optimal number WGs of 512, even though the CPU provides only 4 cores, and on the GPU, the optimal number of WIs is 16, although the NVIDIA documents recommend for this number a multiple of 32.

6 CONCLUSION

We present a high-performance, parallel OpenCL implementation of Probabilistic Record Linkage (PRL) that targets both multi-core CPU and GPU. Our implementation can be automatically optimized for the target hardware architecture by auto-tuning. For this, we use the approach of Multi-Dimensional Homomorphisms (MDHs) and their OpenCL implementation [16]: we show how PRL is expressed as an MDH, and we use the MDHs' implementation schema to elaborate a high-performance OpenCL implementation that is automatically optimized for different target hardware architectures.

We show experimentally that our implementation for PRL provides significantly better performance (up to 80 times) on both CPU and GPU as compared to the currently used PRL implementation of the EKR – the largest cancer registry in Europa – which uses JAVA threads on multi-core CPUs. Our PRL implementation is generic and can be used in further application scenarios.

ACKNOWLEDGMENT

We are grateful to the anonymous reviewers for very helpful comments. This work has been supported by the BMBF project HPC2SE and the DFG Cluster CiM. We thank NVIDIA Corp. for donating the hardware used in our experiments.

REFERENCES

[1] Murilo Boratto, Pedro Alonso, Clicia Pinto, Pedro Melo, Marcos Barreto, and Spiros Denaxas. 2018. Exploring hybrid parallel systems for probabilistic record linkage. *The Journal of Supercomputing* (2018).

[2] Peter Christen. 2012. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer.

[3] A. K. Elmagarmid et al. 2007. Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 16 pp.

[4] Ivan P. Fellegi et al. 1969. A Theory for Record Linkage. *J. Amer. Statist. Assoc.*, 1183–1210.

[5] Benedikt Forchhammer et al. 2013. Duplicate Detection on GPUs. *HPI Future SOC Lab* 70, 3 pp.

[6] Sergei Gorlatch. 1999. Extracting and implementing list homomorphisms in parallel program development. *Science of Computer Programming* 33, 27 pp. https://doi.org/10.1016/S0167-6423(97)00014-2

[7] Sergei Gorlatch and Murray Cole. 2011. Parallel Skeletons. *Encyclopedia of Parallel Computing*, 1417–1422.

[8] K Hentschel et al. 2008. Das Krebsregister-Manual der Gesellschaft der epidemiologischen Krebsregister in Deutschland e.V. Zuckschwerdt Verlag.

[9] Intel. 2014. OpenCL Optimization Guide. https://software.intel.com/sites/default/files/managed/72/2c/gfxOptimizationGuide.pdf

[10] Kazuhiko Komatsu et al. 2010. Evaluating Performance and Portability of OpenCL Programs. In *Workshop on Automatic Performance Tuning*. 15 pp.

[11] Victor W. Lee et al. 2010. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 451–460.

[12] Axel-Cyrille Ngonga Ngomo et al. 2013. When to Reach for the Cloud: Using Parallel Hardware for Link Discovery. In *Extended Semantic Web Conference*. Springer, 275–289.

[13] John Nickolls et al. 2008. Scalable Parallel Programming with CUDA *(ACM SIGGRAPH)*. 14 pp.

[14] NVIDIA. 2009. NVIDIA OpenCL Best Practices Guide. https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

[15] Ari Rasch and Sergei Gorlatch. 2018. ATF: A Generic, Directive-Based Auto-Tuning Framework. *Concurrency and Computation: Practice and Experience*, 16 pp. https://doi.org/10.1002/cpe.4423

[16] Ari Rasch and Sergei Gorlatch. 2018. Multi-dimensional Homomorphisms and Their Implementation in OpenCL. *International Journal of Parallel Programming* (2018), 101–119. https://doi.org/10.1007/s10766-017-0508-z

[17] Ziad Sehili et al. 2015. Privacy Preserving Record Linkage with PPJoin . In *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*. Gesellschaft für Informatik e.V., 85–104.

[18] John E Stone et al. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*.

[19] Dinusha Vatsalan et al. 2017. *Privacy-Preserving Record Linkage for Big Data: Current Approaches and Research Challenges*. Springer, 851–895.

[20] X. Zhang et al. 2013. A Privacy Leakage Upper Bound Constraint-Based Approach for Cost-Effective Privacy Preserving of Intermediate Data Sets in Cloud. *IEEE Transactions on Parallel and Distributed Systems*, 1192–1202.

[21] Xuyun Zhang et al. 2013. An efficient quasi-identifier index based approach for privacy preservation over incremental data sets on cloud. *J. Comput. System Sci.*, 542 – 555.