RESPONSE TO REVIEWS (PAPER #36)

Dear Meta Reviewer and Referees:

We would like to thank the referees for their thorough reading of our paper and for their helpful comments.

We have substantially revised the paper following the Referees' suggestions. For the convenience of Referees, changes to the paper are color-coded in blue.

Below please find our responses to the reviews.

Response to the comments of Referee #4.

[W3] *The motivation of some designs choices is weakly supported by experimental data and would benefit from profiling information. For example, while thread divergence is a general concern in GPU-programming, it's unclear how much it impacts ER blocking and how much the proposed solution improves thread divergence itself. A direct breakdown of the micro-architectural and system execution overheads with and without HyperBlocker would clarify such question.*

[A] Thanks! As suggested, we have clarified **X**. .

[W4] *Thread divergence*

- *i. Section 5.1. It's unclear how the "sequential execution path" reduces thread divergence. A "sequential" representation should improve data locality/predictability, and avoid the need of a stack-/backtracking per-thread; however, if the tree is interpreted (no code generation), what is the improvement in thread divergence compared to an iterative DFS implementation?*
- *ii. I would expect more divergence due to the variable length fields, compared to the execution tree traversals, e.g., because even evaluating the same predicate on the same field for two different pairs would potentially finish earlier for one pair rather than the other. It would be nice to elaborate on the different sources of thread divergence and their impact in execution time with and without HyperBlocker. iii. Would it be possible to use an entiry wrap per input pair to counteract thread divergence given the computational demands of ER predicates?*
-

[A] Thanks! As suggested, we have clarified **X**. .

[W5] *Work-stealing*

- *i. What is the benefit of inter-internal task stealing compared to letting the GPU programming framework handle it? i.e., compared to assigning a thread block per "interval" and letting CUDA/etc schedule a new thread block whenever one is done.*
- *ii. In Figure 6h, wouldn't EvenSplit and RoundRobin impede CUDA's ability to effectively schedule the "interval" processing?*

[A] Thanks! As suggested, we have clarified **X**. .

[W1 & D1]

The "shallow NN" seems to need to be evaluated per tuple-pair per predicate ("A-value of t1 [...] similarly for t2") and its sensitivity to the training set is unclear.

- *i. Wouldn't that result in divergent execution trees per pair of input tuples?*
- *ii. Is there any clustering of the input pairs based on the similarity of the execution tree?*
- *iii. How does the end-to-end execution time benefits from this quadratic evaluation overhead compared to using only the predicate and column type/description as the inputs?*
- *iv. What is the sensitivity of its accuracy to the similarity of the training set to the actual queries? Does the training require labels and data from the same dataset and/or predicates?*
- *v. It would be interesting to comment on how well the "shallow NN" manages to predict the final result of the evaluation (especially if it needs to be trained on data and predicated from the same input).*
- *vi. How accurate is the ordering produces using the NN? i.e., how close is it to the ordering based on the real evaluation cost?*
- *vii. How does the inference cost compare to evaluating the actual predicate, given that the NN seems to be getting the A-value as input parameters and thus it has to access each attribute twice per pair?*

[A] Thanks! As suggested, we have clarified **X**. .

[W2] *The predicate ordering cost seems to be taking into consideration only the cost efficiency of evaluating a predicate on a pair of inputs; however, prioritizing "partitionable" predicates (e.g., equality predicates) also allows for partitioning and reducing the input to the predicate evaluation. How does the proposed ordering compare to versus prioritizing equality constraints and partitioning the input?*

[A] Thanks! As suggested, we have clarified **X**. .

[W6] *Scalability across multiple GPUs.*

- *i. What is the PCIe/NVlink topology of the system used in the evaluation? And assuming shared PCIe resources, please specify which GPUs in the topology are selected for the 2/4 GPU cases.*
- *ii. What is the reasoning behind the observed scalability by the number of GPUs in Figure 6b?*
- *iii. Section 5.4/b. Do you mean PCIe lane instead of PCIe channel? V100 has 16 lanes. When does a partition have to wait for a PCIe lane? Usually, data transfers from/to GPU utilize all the lanes concurrently as long as there are sufficient on-thefly data.*

[A] Thanks! As suggested, we have clarified **X**. .

[D2] *It would be nice to include a breakdown that shows how HyperBlocker improves the GPU resource usage (e.g., memory bandwidth, compute, interconnect) and bottlenecks (e.g., thread divergence, data stalls, random accesses) from a hardware perspective.*

[A] Thanks! As suggested, we have clarified **X**. .

[D3] *Intra-interval task-stealing. Should it be incrementing start before starting to work on a pair to avoid double work?*

[A] Thanks! As suggested, we have clarified **X**. .

[D4] *Keeping the parameters constant across experiments in the sensitivity analysis would improve interpretability of the plots. For example, the partition number m changes across experiments, making it hard to correlate points across experiments. Same goes for the datasets and number of GPUs used across plots in Exp-4.*

[A] Thanks! As suggested, we have clarified **X**. .

[D5] *Given that both predicate evaluation order, work-stealing, and shared-memory execution are useful for CPU execution, it would be nice to elaborate on how the challenges listed in Section 3 extend or differentiate to CPU ones.*

[A] Thanks! As suggested, we have clarified **X**. .

[D6] *How are the per-partition/block skew statistics collected and what is the overhead of this process?*

[A] Thanks! As suggested, we have clarified **X**. .

[D7] *Minor/Typos:*

[A] Thanks! As suggested, we have clarified **X**. .

Response to the comments of Referee #7.

[W1] *My main concern is that novelty and originality were not clear to me. The material in Section 4 looks quite similar to cost-based query optimization, specifically predicate ordering. It would be nice to see a detailed discussion about the unique challenges that had to be addressed in this particular problem that cannot be handled by previous work. Similarly, in Section 5, it was difficult for me to understand how the GPU optimizations are novel and, again, what were the unique challenges in this particular problem that required unique solutions.*

[A] Thanks! As suggested, we have clarified **X**. .

[W2] *Another suggestion is to include more examples and intuition early on. In the introduction, I was hoping to see an intuitive example of how the parallelism afforded by GPUs can accelerate blocking. Then, I was hoping for a clear explanation/examples of the differences between shared-nothing and sharedmemory parallelism in this problem,* i.e.,, *why map-reduce solutions are not applicable.*

[A] Thanks! As suggested, we have clarified **X**. .

[D1] *Figure 1 shows that rule-based blocking scales better, motivating the focus of this paper on GPU-accelerated rule-based blocking. Still, I wonder if GPUs could improve the scalability of DL-based blocking as well, or even then, would rulebased blocking still win?*

[A] Thanks! As suggested, we have clarified **X**. .

[D2] *I wonder what makes the current solution inapplicable to REEs. What are the additional challenges that would need to be solved?*

[A] Thanks! As suggested, we have clarified **X**. .

Response to the comments of Referee #8.

[O1] *Page 2 discusses that existing shared-nothing architectures lead to "unsatisfactory performance". Maybe you can add another sentence to explain what this means (e.g. is the latency too high, or is too much hardware needed for the CPU-based solutions, or something else)?*

[A] Thanks! As suggested, we have clarified **X**. .

[O2] *Section 4.2 should mention the topic of execution plan generation in relational database systems (RDBMS). The tasks that the authors perform here are also necessary in the context of e.g. predicate evaluation ordering in RDBMS: E.g. in a query such as SELECT * FROM conferences c WHERE c.year = 2025 and c.name LIKE '%LDB' , the optimizer needs to decide which predicates to evaluate first. Similar to what the authors discussed, the selectivity of the predicates (e.g. how many conferences do we already have for 2025) and the cost of the operators (LIKE is rather expensive due to the flexible string matching compared to the integer comparison in year) are important here, and there seems to be much overlap to the techniques discussed in the paper. This relationship should be discussed, and maybe also some references should be included. A good starting point could be Guido Moerkotte's unfinished book on Building Query Compilers (https://pi3.informatik.uni-mannheim.de/ moer/querycompiler.pdf), that has many interesting references, e.g. on how to represent these predicates in query plans. Section 24 on cardinality estimation might also have some good material, as well as the section on cost modeling.*

[A] Thanks! As suggested, we have clarified **X**. .

[O3] *Maybe more of a question: I guess in case there is a rule that says t.x=s.x (i.e. the equality of a field in two tuples is compared), this rule will be evaluated twice, once when tuple t_1 is compared with tuple t_2, and then again when t_2 is compared with t_1 (i.e. there is no memoization of comparisons)? I don't think this is an issue, but might be worth mentioning in Section 5.1 when discussing "Reusing computation"?*

[A] Thanks! As suggested, we have clarified **X**. .

[O4] *On page 11, in Exp-3, it would be nice to mention why we don't scale linearly when increasing the amount of GPUs, but "only" get 2.6x more when going from 1 to 8 CPUs. I guess from Figure 6g that the reason might be that the partitioning (and IO time?) cannot benefit and we only accelerate the computation part, but it would be nice if the authors could clarify/confirm this in the text (very briefly).*

[A] Thanks! As suggested, we have clarified **X**. .

[D1] *Page mentions "master node", which might offend some readers, maybe use "coordinator node" or something similar instead?*

[A] Thanks! As suggested, we have clarified **X**. .

[D2] *Maybe it is worth moving the related work on page 2 into a separate section (Section 2) so that it is different from the introduction section?*

[A] Thanks! As suggested, we have clarified **X**. .

[D3] *On page 4, CUDA appears without being introduced (both the acronym and what it stands for). Given the high rigor of the rest of the paper, maybe add a brief sentence and reference?*

[A] Thanks! As suggested, we have clarified **X**. .

[D4] & [D6] & [D7]

- *Page 5: Type "Devise execution" should be "Device execution"*
- *Page 7 typo "efficient devise execution" should be "efficient device execution"*
- *Page 11 typo "hwere" should be "where"*

[A] Thanks! As suggested, we have clarified **X**. .

[D5] *Page 5: Type "Devise execution" should be "Device execution"*

[A] Thanks! As suggested, we have clarified **X**. .

Many thanks for your support and constructive suggestions!

HyperBlocker: Accelerating Rule-based Blocking in Entity Resolution using GPUs

Xiaoke Zhu
Beihang University, China
Shenzhen Institute of Computing Sciences, China
zhuxk@buaa.edu.cn

Min Xie*
Shenzhen Institute of Computing Sciences, China
xiemin@sics.ac.cn

Ting Deng
Beihang University, China
dengting@act.buaa.edu.cn

Qi Zhang
Meta Platforms, USA
qizhang@meta.com

ABSTRACT

This paper studies rule-based blocking in Entity Resolution (ER). We propose HyperBlocker, a GPU-accelerated system for blocking in ER. As opposed to previous blocking algorithms and parallel blocking solvers, HyperBlocker employs a pipelined architecture to overlap data transfer and GPU operations, and improve parallelism by effectively collaborating GPUs. It also generates a data-aware and rule-aware execution plan on CPUs, for specifying how rules are evaluated in blocking. Moreover, it develops a number of hardware-aware optimizations to achieve massive parallelism across multiple GPUs. Using real-life and synthetic datasets, we show that HyperBlocker is at least 13.8× and 10.4× faster than prior CPU-powered distributed systems and GPU-based ER solvers, respectively. Better still, by combining blocking in HyperBlocker with the state-of-the-art ER matcher, we speed up the overall ER process by at least 30% with comparable accuracy.

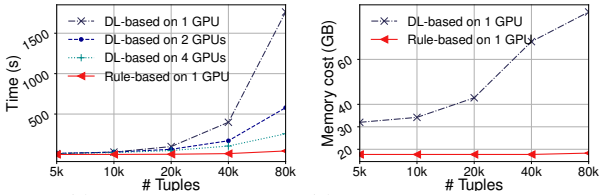
1 INTRODUCTION

Entity resolution (ER), also known as record linkage, data deduplication, merge/purge and record matching, is to identify tuples that refer to the same real-world entity. It is a routine operation in many data cleaning and integration tasks, such as detecting duplicate commodities in e-commerce databases [34] and finding duplicate customers in enterprise databases [23].

Recently, with the rising popularity of deep learning (DL) models, especially large language models, research efforts have been spent on applying DL techniques to ER. Although these DL-based approaches have shown impressive accuracy, they also come with high training/inference costs, due to the large number of parameters. Despite the effort to reduce parameters, the growth in the size of DL models is still an inevitable trend, leading to the increasing time required for making matching decisions.

In the worst case, ER solutions have to spend quadratic time to examine all pairs of tuples. As reported by Thomson Reuters, an ER project can take 3-6 months, mainly due to the scale of data [21]. To accelerate, most ER solutions divide ER into two phases: (a) a blocking phase, where a blocker discards unqualified pairs that are guaranteed to refer to distinct entities, and (b) a matching phase, where a matcher compares the remaining pairs to finally decide whether they are *matched*, *i.e.,* refer to the same entity. The blocking phase is particularly useful when dealing with large data and "is the crucial part of ER with respect to time efficiency and scalability" [58].

To cope with the increasing volume of big data, considerable research has been conducted on blocking techniques. As surveyed in [48, 58], we divide blocking methods into *rule-based* methods [21, 35, 38, 42, 57] or *DL-based* methods [25, 39, 66, 69], both of which have their strengths and limitations [48].



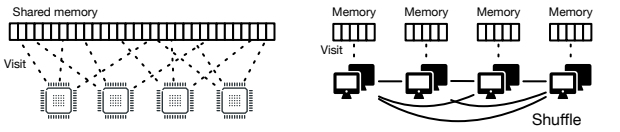
(a) Execution time **(b) Memory cost on single GPU**

Figure 1: DL-based blocking vs Rule-based blocking

DL-based blocking methods typically utilize pre-trained DL models to generate embeddings for tuples and discard tuple pairs with similarity scores below a predefined threshold. While DL-based blocking can enhance ER by parallelizing computation and leveraging GPU acceleration [40], it often comes with long runtime and high memory costs. To justify this, we conducted a detailed analysis on DeepBlocker [66], the state-of-the-art DL-based blocker in Figure 1. We picked a rule-based blocker (a prototype of our method) with comparable accuracy with DeepBlocker and compared it with DeepBlocker in terms of runtime and memory cost. The evaluation was conducted on a machine equipped with V100 GPUs using the Songs dataset [53], varying the number of tuples. When running on one GPU, the runtime of DeepBlocker increases substantially when the number of tuples exceeds 40k. Worse still, it consumes excessive memory due to the large embeddings and intermediate results generated during similarity computation. Although the runtime of DeepBlocker can be reduced by using more GPUs, the issue remains, *e.g.,* even with four GPUs in Figure 1(a), DeepBlocker is still slower than the rule-based blocker that run on one GPU.

In contrast, rule-based blocking methods demonstrate potential for achieving scalability by leveraging multiple *blocking rules*. Each rule employs various comparisons with logical operators such as AND, OR, and NOT to discard unqualified tuple pairs. Heuristic methods are also generally classified into this category [48]. For instance, a blocking rule for books may state "If titles match and the number of pages match, then the two books match" [44]. We refer to the comparisons in this rule as *equality comparisons*, as they require exact equality. Another example, referred to as *similarity comparisons*, is presented in [57], which adopts the Jaccard similarity to determine whether a pair of tuples requires further matching. Rule-based approaches complement DL-based approaches by providing flexibility, explainability, and scalability in the blocking process [17]. Moreover, by incorporating domain knowledge into blocking rules, these approaches can readily adapt to different domains.

*Corresponding author



(a) Shared Memory: all data is shared by everyone

(b) Shared nothing: each node is autonomy over a subset of data

Figure 2: Shared memory vs shared nothing architecture

Example 1: As a critical step for data consistency, an e-commerce company (*e.g.,* Amazon [6]) conducts ER for products, to enhance operations for *e.g.,* product listings and inventory management.

To identify duplicate products, the blocking rule φ_1 may fit.

φ_1: Two products are potentially matched if (a) they have same color and price, (b) they are sold at same store, (c) their names are similar.

Here φ_1 is a conjunction of attribute-wise comparisons, where both equality (parts (a) and (b)) and similarity comparisons (part (c)) are involved. In Section 3, we formally define φ_1. □

Rule-based blocking in ER has attracted a lot of attention. A variety of rule-based blockers are already in place (see [48, 58] for surveys). Unfortunately, most of them are designed for CPU-based (shared-nothing) architectures, leading to unsatisfactory performance. As shown in Figure 2, in the shared-nothing architecture, data is partitioned and spread across a set of processing units. Each processing unit conducts blocking independently with its own private memory, leading to potential load imbalance due to skewed partitions and increasing communication cost when tuples in different partitions should be paired. The shared memory architecture is the opposite: all data is accessible from all processing units, allowing for efficient data sharing and collaboration between processing units. GPUs typically have a shared memory architecture. However, unlike DL-based blocking approaches, few rule-based methods support the massive parallelism offered by GPUs (shared memory), despite their greater potential in scalability (see Figure 1).

To make practical use of rule-based blocking, several questions have to be answered. Can we parallelize rule-based blocking on GPUs? Can we utilize multiple GPUs to accelerate? Can we explore characteristics of GPUs and CPUs, to effectively collaborate them?

HyperBlocker. To answer these, we develop HyperBlocker, a GPU-accelerated system for rule-based blocking in Entity Resolution. As proof of concept, we adopt matching dependencies (MDs) [28] for rule-based blocking. As a class of rules developed for record matching, MDs are defined as a conjunction of (similarity) predicates and support both equality and similarity comparisons. Compared with prior works, HyperBlocker has the following unique features.

(1) A pipelined architecture. HyperBlocker adopts an architecture that pipelines the memory access from/to CPUs for data transfer, and operations on GPUs for rule-based blocking. In this way, the data transfer and the computation on GPUs can be overlapped, so as to "cancel" the excessive data transfer cost.

(2) Execution plan generation on CPUs. To effectively filter unqualified tuple pairs, blocking must be optimized for the underlying data (resp. blocking rules); in this case, we say that the blocking is *data-aware* (resp. *rule-aware*). To our knowledge, no prior methods, neither on CPUs nor on GPUs, have considered data/rule-awareness for their execution models. HyperBlocker designs an effective execution plan generator to warrant efficient rule-based blocking.

(3) Hardware-aware parallelism on GPUs. Due to different characteristics of CPUs and GPUs, a naive approach that applies existing CPU-based blocking on GPUs makes substantial processing capacity untapped. We develop a variety of GPU-based parallelism strategies, designated for rule-based blocking, by exploiting the hardware characteristics of GPUs, to achieve massive parallelism.

(4) Multi-GPUs collaboration. It is already hard to balance the workload on CPUs. This problem is even exacerbated under multi-GPUs scenarios, since tens of thousands of threads will compete for limited GPU resources. HyperBlocker provides an effective task-scheduling strategy to scale with multiple GPUs.

With these attractive features, HyperBlocker is capable of conducting rule-based blocking with shorter time and less memory, as shown in Figure 1; more sophisticated experiments are shown later.

Contribution & organization. After reviewing related work and background in Sections 2-3, we present HyperBlocker as follows: (1) its unique architecture and system overview (Section 4); (2) the rule/data-aware execution plan generator (Section 5); (3) the hardware-aware parallelism and the task scheduling strategy across GPUs (Section 6); and (4) an experimental study (Section 7).

Using real-life and synthetic datasets, we find: (a) HyperBlocker speedups prior distributed ER blocking systems and GPU baselines by at least 13.8× and 10.4×, respectively. (b) Combing HyperBlocker with the state-of-the-art ER matcher saves at least 30% time with comparable accuracy. (c) HyperBlocker is scalable, *e.g.,* it can process 36M tuples in 1604s. (d) While promising, DL-based blocking methods are not always the best. By carefully optimizing rule-based blocking on GPUs, we share valuable lessons/insights about when rule-based approaches can beat DL-based ones and vice versa.

2 RELATED WORK

We categorize the related work in the literature as follows.

Blocking algorithms. There has been a host of work on the blocking algorithms, classified as follows: (1) Rule-based [21, 35, 38, 42, 57], *e.g.,* [35] creates data partitions and then refines candidate pairs in every partition, by removing mismatches with similarity measures or length/count filtering [51]. (2) DL-based [25, 39, 66, 69], which cast the generation of candidate matches into a binary classification problem, where each tuple pair is labeled "likely match" or "unlikely match", *e.g.,* [66] adopts similarity search to generate candidate matches for each tuple based on its top-K probable matches in an embedding space. DL-based blocking and rule-based blocking share the same goal, but are different in their approaches, where the former focuses on learning the distributed representations of tuples, while the latter emphasizes explicit logical reasoning.

Although we focus on rule-based blocking, this work is not to develop another blocking algorithm. Instead, we aim to provide a GPU-accelerated blocking solution. As a testbed, we use matching dependencies (MDs) as our blocking rules, since they subsume many existing rules [44, 57] as special cases.

Parallel blocking solvers. Several parallel blocking systems have been proposed, *e.g.,* [15, 21, 22, 27, 31, 33, 42, 43, 59, 65]. Most are studied under MapReduce [21, 33, 42] or MPC [23, 31, 65],

Table 1: A relation D of schema Products, where the dash ("-") denotes a missing value.

eid	pno	pname	price	sname	description	color	saddress
e_1	t_1	Apple Mac Air	$909	Comp. World	Apple MacBook Air (13-inch, 8GB RAM, 256GB SSD)	Gray	9 Barton Grove, McCulloughmouth
e_2	t_2	ThinkPad	-	Smith's Tech	ThinkPad E15, 15.6-inch full HD IPS display, Intel Core i5-1235U processor, (16GB) RAM | 512GB PCIe SSD)	Gray	Seg Plaza, Hua qiang North Road
e_2	t_3	ThinkPad	$849	Smith's Tech	Lenovo E15 Business ThinkPad, 15.6-inch full HD IPS display, 12 generation Intel Core i5, 16GB RAM, 512GB SSD	Gray	Seg Plaza, Hua qiang North Road
e_1	t_4	MacBook Air	$909	Comp. World	Apple 2022 MacBook Air M2 chip 13-inch,8 GB RAM,256 GB SSD storage gray	Gray	-
e_1	t_5	MacBook Air	$909	Comp. World	-	Gray	Barton Grove, McCulloughmouth

which aim at scaling to large data with a cluster of machines. DisDedup [21] uses a triangle distribution strategy to minimize both maximum comparisons and maximum communication over Spark[11]. Minoan [27] runs on top of Spark and applies parallel meta blocking [26] to minimize its overall runtime.

This work differs from the prior work as follows. (1) Unlike existing MapReduce-based blocking systems, which split data in the master node and execute them in parallel, HyperBlocker partitions data on the CPU and asynchronously conducts computation on GPUs. (2) Existing distributed solutions mainly adopt partition parallelism under the shared-nothing architecture and thus, they either sacrifice the accuracy for parallelism or perform redundant computation on overlapping partitions to avoid missing results. In contrast, HyperBlocker provides hardware-aware parallelism under the shared memory architecture and settles the task scheduling problem, to promote better utilization of resources.

GPU-accelerated techniques. GPUs have been used extensively for decades to accelerate visualization applications from gaming to interactive displays and to speed up the training of DL tasks. Recent works exploit GPUs to accelerate data processing, *e.g.,* GPU-based query answering [24, 37, 62] and similarity join [40, 50, 55]. Closer to this work are [40, 50] which leverage GPUs for similarity join, since blocking can be regarded as a similarity join problem under the assumption that two tuples refer to the same entity if their similarity is high. Similarity join is often served as a preprocessing step of ER.

In contrast, HyperBlocker aims at expediting rule-based blocking, addressing challenges in rule-based optimization that are not incurred in similarity join. The closest work is GPUDet [32], which employs GPUs to expedite similarity measures. HyperBlocker differs from GPUDet, in its data/rule-aware execution plan designated for rule evaluation, beyond similarity measures. It also incorporates hardware-aware optimizations for improving GPU utilization.

3 PRELIMINARIES

We first review notations [7] for ER, blocking, and GPUs.

Relations. Consider a schema $R = (\mathsf{eid}, A_1, \ldots, A_n)$, where A_i is an attribute ($i \in [1, n]$), and eid is an entity id, such that each tuple of R represents an entity. A relation D of R is a set of tuples of schema R.

Entity resolution. Given a relation D, ER is to identify all tuple pairs in D that refer to the same real-life entity. It returns a set of tuple pairs (t_1, t_2) of D that are identified as *matches*. If t_1 does not match t_2, (t_1, t_2) is referred to as a *mismatch*.

Most existing methods typically conduct ER in three steps:

(1) Data partitioning. The tuples in relation D are divided into multiple disjoint data partitions, namely $P_1, P_2, ..., P_m$, so that tuples of similar entities are put into the same data partition.

(2) Blocking. Each tuple pair (t_1, t_2) from the same data partition P (*i.e.,* $(t_1, t_2) \in P \times P$) is a potential match that requires further verification. To reduce cost, a blocking method $\mathcal{A}_{\mathsf{block}}$ (*i.e.,* blocker) is often adopted to filter out those pairs that are definitely mismatches *efficiently*, instead of directly examining every tuple pair. Denote the resulting set of remaining tuple pairs obtained from partition P by $\mathsf{Ca}(P) = \{(t_1, t_2) \in P \times P \mid (t_1, t_2)$ is not filtered by $\mathcal{A}_{\mathsf{block}}\}$.

(3) Matching. For the remaining tuple pairs in $\mathsf{Ca}(P)$ of each data partition P, an accurate (but expensive) matcher will be applied, to make the final decision of matches/mismatches.

Our scope: blocking. Note that in some works, both steps (1) and (2) are called blocking. To avoid ambiguity, we follow [66] and distinguish partitioning from blocking. We mainly focus on *blocking*, *i.e.,*

- *Input*: A relation D of the tuples of schema R, where the tuples in D are divided into m partitions $P_1, \ldots, P_m$.
- *Output*: The set $\mathsf{Ca}(P_i)$ of candidate tuple pairs on each P_i.

Although our work can be applied on data partitions generated by *any* existing method, we optimize over multiple data partitions, by exploiting designated GPU acceleration techniques (Section 6.3).

While blocking focuses more on efficiency and matching focuses more on accuracy, they can be used without each other, *e.g.,* one can directly employ rules [28] for ER or apply an ER matcher [49] on the Cartesian product of the entire data. When blocking is used alone on a given partition P, all tuple pairs in $\mathsf{Ca}(P)$ are identified as matches. In Section 7, we will test HyperBlocker with or without a matcher, to elaborate the trade-off between efficiency and accuracy.

Rule-based blocking. We study rule-based blocking in this paper, due to its efficiency and explainability remarked earlier. We review a class of matching dependencies (MDs), originally proposed in [28].

Predicates. Predicates over schema R are defined as follows:

$$p ::= t.A = c \mid t.A = s.B \mid t.A \approx s.B$$

where t and s are tuple variables denoting tuples of R, A and B are attributes of R and c is a constant; $t.A = s.B$ and $t.A = c$ compare the equality on *compatible* attribute values, while $t.A \approx s.B$ compares the *similarity* of $t.A$ and $s.B$. Here any similarity metric can be used as $\approx$, *e.g.,* q-grams, Jaro distance or edit distance, such that $t.A \approx s.B$ is true if $t.A$ and $s.B$ are "similar" enough w.r.t. a predefined threshold. In particular, $t.\mathsf{eid} = s.\mathsf{eid}$ says that (t, s) is a potential match.

Rules. A (bi-variable) *matching dependency* (MD) over R is:

$$\varphi = X \rightarrow l,$$

where X is a conjunction of predicates over R with two tuple variables t and s, and l is $t.\mathsf{eid} = s.\mathsf{eid}$. We refer to X as the *precondition* of φ, and l as the *consequence* of φ, respectively.

Example 2: Consider a (simplified) e-commence database with self-explained schema Products (eid, pno, pname, price, sname (store name), description, color, saddress (store address)). Below are some examples MDs, where the rule in Example 1 is written as φ_1.

(1) φ_1 : $t.\mathsf{color} = s.\mathsf{color} \wedge t.\mathsf{price} = s.\mathsf{price} \wedge t.\mathsf{sname} = s.\mathsf{sname}$

$\wedge$ t.pname $\approx_{\mathsf{ED}}$ s.pname $\rightarrow$ t.eid = s.eid, where $\approx_{\mathsf{ED}}$ measures the edit distance. As stated before, φ_1 identifies two products, by their colors, prices, product names and the stores sold.

(2) φ_2 : t.sname = s.sname $\wedge$ t.description $\approx_{\mathsf{JD}}$ s.description $\rightarrow$ t.eid = s.eid, where $\approx_{\mathsf{JD}}$ measures the Jaccard distance. The MD says that if products are sold in the store and have a similar description, then they are likely to be identified as potental match.

(3) φ_3 : t.saddress $\approx_{\mathsf{ED}}$ s.saddress$\wedge t$.description $\approx_{\mathsf{JD}}$ s.description $\rightarrow$ t.eid = s.eid. It gives another condition for identifying two products, *i.e.,* the two products with similar descriptions are sold from stores with similar addresses are potentially matched. □

Semantics. A *valuation* of tuple variables of an MD φ in D, or simply *a valuation of* φ, is a mapping h that instantiates the two variables t and s with tuples in D. A valuation h *satisfies* a predicate p over R, written as $h \models p$, if the following is satisfied: (1) if p is $t.A = c$ or $t.A = s.B$, then it is interpreted as in tuple relational calculus following the standard semantics of first-order logic [14]; and (2) if p is $t.A \approx s.B$, then $h(t).A \approx h(s).B$ returns true. Given a conjunction X of predicates, we say $h \models X$ if for *all* predicates p in X, $h \models p$.

Blocking. Rule-based blocking employs a set Δ of MDs. Given a partition P, a pair $(t_1, t_2) \in P \times P$ is in $\mathsf{Ca}(P)$ *iff* there exists an MD $\varphi : X \rightarrow l$ in Δ such that the valuation $h(t_1, t_2)$ of φ that instantiates variables t and s with tuples t_1 and t_2 satisfies the precondition of φ; we call such φ as a *witness* at (t_1, t_2), since it indicates that (t_1, t_2) is a potential match. Otherwise, (t_1, t_2) will be filtered.

Example 3: Continuing with Example 2, consider D in Table 1 and a valuation $h(t_1, t_4)$ that instantiates variables t and s with tuples t_1 and t_4 in D. Since $h(t_1, t_4)$ satisfies the precondition of φ_1, φ_1 is a witness at (t_1, t_4). Similarly, one can verify that φ_1 is not a witness at (t_2, t_3) since $h(t_2, t_3) \not\models t.\text{price} = s.\text{price}$ (due to value missing). □

Discovery of MDs. As noted in [29, 30], MDs can be considered as a special case of entity enhancing rules (REEs). Algorithms are in place for discovering REEs, *e.g.,* [29, 30]. We can readily apply these algorithms for discovering MDs (details omitted).

GPU hardware. GPUs were initially designed as specialized processors to accelerate graphics rendering. However, driven by the increasing demand for real-time/high-performance computation, GPUs have evolved into general processors for more workloads. Compared with CPUs, GPUs offer the following unique benefits.

Firstly, GPUs provide massive parallelism by programming with CUDA (Compute Unified Device Architecture) and thousands of CUDA cores, specialized processing units within NVIDIA graphics cards. A GPU has multiple SM (Streaming Multiprocessors), where each SM accommodates multiple processing units. This organization leads to a hierarchical parallel model: each CUDA kernel runs with groups of threads, called thread blocks (TBs). In each TB, subgroups of 32 threads, called warps, are executed parallely.

Secondly, GPUs are equipped with wider/faster memory interfaces, enabling quick access to large data. Using tightly integrated high-bandwidth memory technologies, GPUs provide a device memory with bandwidth approaching 1 TB/sec. This is particularly useful for analytic tasks with data-intensive operations.

One key difference between a GPU and a CPU is that the GPU adopts SIMD (Single Instruction, Multiple Data) execution. Each SIMD lane, individual processing units within a SIMD, processes one element of a vector of data under the control of a single instruction. Analyzing the utilization of SIMD lanes is crucial for determining whether a program fully exploits the massive parallelism of the GPU. *SIMD divergence* can adversely affect performance and typically occurs in *if-else* blocks of GPU kernels. Some lanes may need to execute the *if* block while others may need to execute the *else* block. However, because the same instruction must be executed in all lanes, the runtime will process each *if* block and each *else* block sequentially, resulting in sub-optimal utilization of processing units.

4 HYPERBLOCKER: SYSTEM OVERVIEW

In this section, we present the overview of HyperBlocker, a GPU-accelerated system for rule-based blocking that optimizes the efficiency, considering rules, underlying data, and hardware simultaneously. In the literature, GPUs and CPUs are usually referred to as devices and hosts, respectively. We also follow this terminology.

Challenges. Existing parallel blocking methods typically rely on multiple CPU-powered machines under the shared-nothing architecture, to achieve data partition-based parallelism. They reduce the runtime by using more machines, which, however, is not always feasible due to the increasing communication cost. Moreover, it is hard to strike for a good balance between parallelism and recall. Although a more fine-grained data partitioning strategy can improve parallelism, it inevitably reduces the recall since a match (t_1, t_2) will be missed if t_1 and t_2 are distributed to different machines. In light of these, HyperBlocker focus on parallel blocking under the shared-memory architecture. This not only opens up new possibilities for optimizations but also introduces new challenges.

(1) Execution plan for efficient blocking. The efficiency of blocking depends heavily on how much and how fast we can filter mismatches. Therefore, a good execution plan that specifies how rules are evaluated is crucial. However, most existing solvers fail to consider the properties of rules/data during blocking. This motivates us to design a different method to get an effective execution plan.

(2) Hardware-aware parallelism. Existing blocking solvers often adopt data partition-based parallelism on CPU-powered machines. However, when GPUs are involved, these CPU-based techniques no longer suffice, since GPUs have radically different characteristics. Novel GPU-based parallelism for blocking is required.

(3) Multi-GPUs collaboration. Existing parallel solvers focus on minimizing the largest cost across all workers under the shared-nothing architecture [21]. This objective no longer applies to the shared-memory architecture that we adopt, where a scheduling problem unique to multi-GPUs collaboration naturally arises.

Novelty. The ultimate goal of HyperBlocker is to generate the set $\mathsf{Ca}(P)$ of potential matches on each data partition P. To achieve this, we implement three novel components as follows:

(1) Execution plan generator (EPG) (Section 5). We develop a data-aware and rule-aware generator to generate execution plans. Here we say the execution plan is data-aware, since it considers the distribution and skewness of data so as to decide which predicates

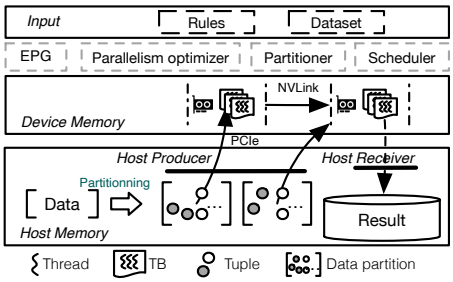


Figure 3: The pipelined architecture of HyperBlocker

are evaluated first; similarly, the execution plan is rule-aware, since its evaluation order is optimized for the underlying rules.

(2) Parallelism optimizer (Section 6). We implement a specialized optimizer that exploits the hierarchical structure inherent to GPUs, to optimize the power of GPUs by utilizing thread blocks (TBs) and warps. With this optimizer, the GPU optimizations are encapsulated and blocking can be effectively parallelized.

(3) Resource scheduler (Section 6). To achieve optimal performance from limited resources, a partitioning strategy and a resource scheduler are developed to manage the resources, and balance the workload across multiple GPUs, minimizing idle time and resource waste.

Architecture. The architecture of HyperBlocker is shown in Figure 3. Taken a relation D of tuples and a set Δ of MDs discovered offline as input, HyperBlocker divides the tuples in D into m disjoint partitions and asynchronously processes these data partitions in a pipelined manner, so that the execution at devices and the data transfer can be overlapped. Such architecture helps to mitigate the excessive I/O costs, reduce idle time, and improve hardware utilization.

Workflow. More specifically, HyperBlocker works in five steps:

(1) *Data partitioning*. HyperBlocker divides the tuples in D into m partitions, to allow parallel processing asynchronously.

(2) *Execution plan generation.* Given the set Δ of MDs discovered offline, an execution plan that specifies in what order the rules (and the predicates in rules) should be evaluated is generated at the host, to enable effective mismatches filtering.

(3) *Host scheduling*. The blocking on each data partition forms a computational task and the host will dynamically assign the tasks to the queue(s) of available devices without interrupting the ongoing execution of devices. This strategy minimizes the idle waiting time of devices and improves resource utilization.

(4) *Device execution.* When a device receives the task assigned, it conducts the rule-based blocking on the corresponding data partition, following the execution plan generated in Step (2).

(5) *Result retrieval.* Once a task is completed on a device, the host will pull/collect the result (i.e., $\mathrm{Ca}(P)$) from the device.

To facilitate processing, HyperBlocker has two additional components: HostProducer and HostReceiver, where the former manages Steps (1), (2), and (3) and the latter handles Step (5). Steps (3) (4) (5) in HyperBlocker work asynchronously in a pipeline manner, to enable continuous execution without unnecessary waiting.

5 EPG: EXECUTION PLAN GENERATOR

Given the set Δ of MDs and a partition P of D, a naive approach to compute $\mathrm{Ca}(P)$ is to evaluate each MD in Δ for all pairs in P. That is, to decide whether (t_1, t_2) is in $\mathrm{Ca}(P)$, we perform $O(\sum_{\varphi\in\Delta}|\varphi|)$ predicate evaluation, where $|\varphi|$ is the number of predicates in φ. Worse still, there are $O(|\Delta|!|\varphi|!)$ possible ways to evaluate all MDs in Δ, since both MDs in Δ and predicates in each φ can be evaluated in arbitrary orders. However, not all orders are equally efficient.

Example 4: In Example 3, φ_1 is a witness at (t_1, t_4) while φ_3 is not. If we first evaluate φ_1 for (t_1, t_4), (t_1, t_4) is identified as a potential match and there is no need to evaluate φ_3. Moreover, when evaluating φ_1 for another pair (t_2, t_3), we can conclude that φ_1 is not a witness at (t_2, t_3), as soon as we find $h(t_2, t_3) \not\models t.\mathrm{price} = s.\mathrm{price}$. □

Challenges. Given the huge number of possible evaluation orders, it is non-trivial to define a good one, for three reasons:

(1) Rule priority. Recall that in rule-based blocking, as long as there exists a witness at (t_1, t_2), (t_1, t_2) will be considered as a potential match. This motivates us to prioritize the rules in Δ so that promising ones can be evaluated early; once a witness is found, the evaluation of the remaining rules can be skipped.

(2) Reusing computation. MDs may have common predicates. To avoid evaluating a predicate repeatedly, we reuse previous results whenever possible, *e.g.*, given (t_1, t_2), $\varphi_1 : p \wedge X_1 \rightarrow l$ and $\varphi_2 : p \wedge X_2 \rightarrow l$, if φ_1 is not a witness at (t_1, t_2) since $h(t_1, t_2) \not\models p$, neither is φ_2.

(3) Predicate ordering. Given (t_1, t_2) and $\varphi : X \rightarrow l$, φ is not a witness at (t_1, t_2) if we find the first p in X such that $h(t_1, t_2) \not\models p$. However, to decide which p in X is evaluated first, we have to consider both the evaluation cost of p and the possibility of p being satisfied.

Novelty. To tackle these challenges, EPG in HyperBlocker gives a lightweight solution, by generating an execution plan to make the overall evaluation cost of Δ on P as small as possible. Its novelty includes (a) a new notion of execution tree to allow efficient reuse of common predicates, (b) a rule-aware scoring strategy, so as to decide which MDs in Δ are evaluated with higher priority, and (c) a data-aware predicate ordering scheme, to strike for a balance between the cost and the effect of evaluating a predicate.

Below we first give the formal definition of execution plans and then show how EPG generates a good execution plan.

5.1 Execution plan

An execution plan specifies how rules and predicates in Δ are evaluated. Although an execution plan can be represented in different ways, we represent it as an *execution tree*, denoted by $\mathcal{T}$ in this paper for its conciseness and simplicity. (1) A node in $\mathcal{T}$ is denoted by N, where the root is denoted by N_0. (2) A *path ρ from the root* is a list $\rho = (N_0, N_1, \ldots, N_L)$ such that (N_{i-1}, N_i) is an edge of $\mathcal{T}$ for $i \in [1, L]$; the length of ρ is L, *i.e.,* the number of edges on ρ. (3) We refer to N_2 as a *child node* of N_1 if (N_1, N_2) is an edge in $\mathcal{T}$, and as a *descendant* of N_1 if there exists a path from N_1 to N_2; conversely, we refer to N_1 as a *parent node* (resp. predecessor) of N_2. (4) Each edge e represents a predicate p and is associated with a score, denoted by $\mathrm{score}(e)$, indicating the priority of e. (5) A node is called a *leaf* if it

has no children and $\mathcal{T}$ has $|\Delta|$ leaves, where each leaf is associated with a rule $\varphi : X \rightarrow l$ in Δ; the length of the path from the root to the leaf is $|X|$ (*i.e.,* the number of predicates in X) and for each predicate in X, it appears exactly once in an edge on the path. (6) The leaves of two MDs may have common predecessors, in addition to the root; intuitively, this means that the MDs have common predicates. With a slight abuse of notation, we also denote an execution plan by $\mathcal{T}$.

Evaluating an execution plan. For each pair (t_1, t_2), it is evaluated by exploring $\mathcal{T}$ via depth-first search (DFS), starting at the root, as follows. At each internal node N of $\mathcal{T}$, we pick a child N_c such that the edge (N, N_c) has the highest score among all children of N. Assume that the predicate associated with this edge is p. Then we check whether $h(t_1, t_2) \models p$. If this is the case, we move to N_c and process N_c similarly. Otherwise, we return to the parent N_p of N and check whether N_p still has other unexplored children, which are processed similarly, according to the decreasing order of scores. The evaluation completes if we reach a leaf of $\mathcal{T}$. Suppose the rule associated with this leaf is $\varphi : X \rightarrow l$. This means $h(t_1, t_2) \models X$ satisfies all predicates in X, along the path from the root to that leaf, and thus $h(t_1, t_2) \models X$, *i.e.,*we find a witness at (t_1, t_2).

Example 5: Consider an execution tree $\mathcal{T}$ in Figure 4(b), which depicts MDs in Example 2. For simplicity, we denote a predicate $t.A = s.A$ (resp. $t.A \approx s.A$) by $p_A^{=}$ (resp. $p_A^{\approx}$) and the score associated with each edge is labeled. DFS starts at the root, which has two children. It first explores the edge labeled $p_{\mathsf{sname}}^{=}$ since its score is higher. When DFS completes, MDs φ_2, φ_1 and φ_3 are checked in order. □

5.2 Execution plan generation

Taking the set Δ of MDs as input, EPG in HyperBlocker returns an execution plan $\mathcal{T}$ in two major steps:

(1) We order all predicates p that appeared in Δ, by estimating its evaluation cost via a shallow model and quantifying its probability of being satisfied, by investigating the underlying data distribution.

(2) Based on the predicate ordering, we build an execution tree $\mathcal{T}$ by iterating MDs in Δ. Moreover, we compute a score for each edge in $\mathcal{T}$, by considering the probability of finding a witness, *i.e.,* reaching a leaf, if we explore $\mathcal{T}$ following this edge.

Once an execution plan $\mathcal{T}$ is generated, it is applied in all partitions of D. Below we present the details of the two steps. For simplicity, we assume *w.l.o.g.* that D is itself a partition.

Predicate ordering. Denote by $\mathcal{P}$ the set of predicates appeared in Δ. Intuitively, not all predicates in $\mathcal{P}$ are equally potent for evaluation, *e.g.,* although textual attributes (*e.g.,*description) are often more informative in identifying entities than categorical ones (*e.g.,*color), the former comparison is more expensive. Thus we order predicates in $\mathcal{P}$ by their "cost-effectiveness".

To simplify the discussion, below we consider a predicate p that compares A-values of two tuples, *i.e.,* $t.A = s.A$ or $t.A \approx s.A$ (simply $p_A^{=}$ or $p_A^{\approx}$). All discussion extends to other predicate types, *e.g.,* $t.A = s.B$ that compares values of two compatible attributes.

Evaluation cost. We measure the evaluation cost of a predicate p by the time for evaluating p; a predicate that can be evaluated quickly should be checked first. Given a predicate p in $\mathcal{P}$ and a relation D, the evaluation cost of p on D, denoted by $\mathrm{cost}(p, D)$, is:

$$\mathrm{cost}(p, D) = \sum_{(t_1,t_2)\in D\times D} T_p(t_1, t_2).$$

where $T_p(t_1, t_2)$ denotes the time for checking if $h(t_1, t_2) \models p$.

Since it is infeasible to iterate all tuple pairs in D to compute the exact evaluation cost of p on D, below we train a shallow NNs, denoted by $\mathcal{N}$, (*i.e.,* a small feed-forward neural network [45]) to estimate the exact $T_p(t_1, t_2)$, since it has been proven effective in approximating a continuous function on a closed interval [64].

Shallow NNs. The inputs of $\mathcal{N}$ are two tuples t_1 and t_2, and a predicate p, that compares the A-values of t_1 and t_2. It first encodes the attribute type and the A-value of t_1 into an embedding $\vec{t_1}$; similarly for $\vec{t_2}$. The embeddings are then fed to a feed-forward neural network, which outputs the estimated time for evaluating p on (t_1, t_2).

To train $\mathcal{N}$, we collect training data from historical logs, where each training instance is in form (x, y); here x is triplet (p, t_1, t_2) and y is its label, *i.e.,* the measured time of evaluating p at (t_1, t_2). We train $\mathcal{N}$ offline, using stochastic gradient descent with the mean square error loss. While the training time is not our focus, $\mathcal{N}$ converges quickly in a few pass [45], since it has few parameters.

Estimated cost. Based on $\mathcal{N}$, the estimated cost of p on D is

$$\hat{\mathrm{cost}}(p, D) = \mathrm{norm}\Big(\sum_{(t_1,t_2)\in D\times D} \mathcal{N}(p, t_1, t_2)\Big),$$

where $\mathrm{norm}(\cdot)$ normalizes the estimated cost in the range (0,1].

Probability of being satisfied. We measure the effectiveness of a predicate p by its possibility of being satisfied, *i.e.,* given the attribute A compared in p, we quantify how likely two tuples t_1 and t_2 have distinct/dissimilar values on A. If t_1 and t_2 have a high probability of having distinct/dissimilar A-values, p is less likely to be satisfied; such predicate should be evaluated first since it concludes that an MD involving p is not a witness early.

To achieve this, we investigate the data distribution in D. Specifically, we use LSH [16] to hash the A-values of all tuples into k buckets, so that similar/same values are hashed into the same bucket with a high probability, where k is a predefined parameter.

Denote the number of tuples hashed to the i-th bucket by b_i. Intuitively, the evenness of hashing results reflects the probability of p being satisfied. If all tuples are hashed into the same bucket, it means that the A-values of all tuples are similar and thus p (which compares the A-values) is likely to be satisfied by many pairs (t_1, t_2); such predicates should be evaluated with low-priority since they cannot tell an MD involving p is not a witness at (t_1, t_2). Motivated by this, the probability of p being satisfied on D, denoted by $\mathrm{sp}(p, D)$, is estimated by measuring the evenness of hashing, *i.e.,*

$$\mathrm{sp}(p, D) = \mathrm{norm}\Big(\sqrt{\frac{1}{k}\sum_{i=1}^{k}(b_i - \frac{|D|}{k})^2}\Big).$$

Ordering scheme. Putting these together, we can order all the predicates p in $\mathcal{P}$ by the cost-effectiveness, defined to be $\frac{1-\mathrm{sp}(p,D)}{\hat{\mathrm{cost}}(p,D)}$. Intuitively, hard-to-satisfied predicates will be evaluated first, since they are more likely to fail a rule, while costly predicates will be penalized, to strike a balance between the cost and the effectiveness.

Example 6: Consider two predicates $p_{\mathsf{color}}^{=}$ and $p_{\mathsf{pname}}^{\approx_{\mathrm{ED}}}$ in φ_1. On

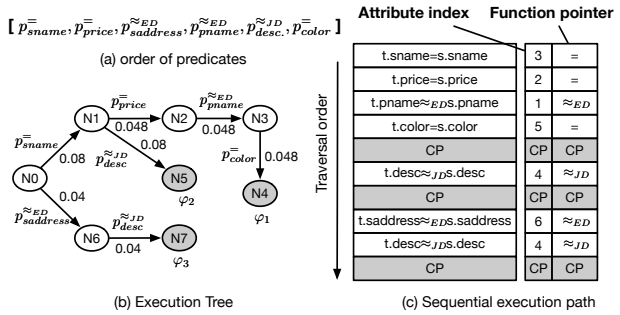


Figure 4: Execution tree

the one hand, since $p^{=}_{\text{color}}$ is an equality comparison while $p^{\approx_{\text{ED}}}_{\text{pname}}$ computes the edit distance, $p^{\approx_{\text{ED}}}_{\text{pname}}$ is more costly to evaluate, *e.g.*, $\hat{\text{cost}}(p^{=}_{\text{color}}, D) = 0.1 < \hat{\text{cost}}(p^{\approx_{\text{ED}}}_{\text{pname}}, D) = 0.6$. On the other hand, since all tuples in D have the same color (and satisfy $p^{=}_{\text{color}}$), we have $\text{sp}(p^{=}_{\text{color}}, D) = 1$; similarly, let $\text{sp}(p^{\approx_{\text{ED}}}_{\text{pname}}, D) = 0.4$. Then the cost-effectiveness of $p^{=}_{\text{color}}$ and $p^{\approx_{\text{ED}}}_{\text{pname}}$ are $\frac{1-1}{0.1} = 0$ and $\frac{1-0.2}{1} = 0.8$, respectively, and $p^{\approx_{\text{ED}}}_{\text{pname}}$ is ordered before $p^{=}_{\text{color}}$ (see Figure 4(a)). □

Constructing an execution tree. We initialize the execution tree $\mathcal{T}$ with a single root node N_0. Then based on the predicate ordering, we progressively construct $\mathcal{T}$ by processing the MDs in Δ one by one. For each MD $\varphi : X \rightarrow l \in \Delta$, we assume the predicates in X are sorted in the descending order of their cost-effectiveness, *i.e.*, if X is $p_1 \wedge p_2 \wedge \ldots \wedge p_{|X|}$, then $\frac{1-\text{sp}(p_i,D)}{\hat{\text{cost}}(p_i,D)} > \frac{1-\text{sp}(p_j,D)}{\hat{\text{cost}}(p_j,D)}$ for $1 \leq i < j \leq |X|$. We traverse $\mathcal{T}$, starting from the root, and process the predicates in X, starting from p_1. Suppose that the traversal is at a node N and the predicate we are processing is p_i. We check the children of N. If there exists a child node N_c of N such that the edge (N, N_c) represents p_i, we move to this child and process the next predicate p_{i+1} in X. Otherwise, we create a new child node N_c for N such that the edge (N, N_c) represents p_i, move to this new child and process the next predicate p_{i+1} in X. The traversal process continues until all predicates in X are processed and we set the current node we reach as a leaf node, whose associated rule is φ.

Example 7: The predicate ordering is shown in Figure 4(a). Assume that we have processed φ_1 and created path $(N_0, N_1, N_2, N_3, N_4)$ in $\mathcal{T}$ in Figure 4(b). Then we show how $\varphi_2 : p^{=}_{\text{sname}} \wedge p^{\approx_{\text{JD}}}_{\text{description}} \rightarrow l$ is processed. We start from the root and process $p^{=}_{\text{sname}}$. Since there is a child N_1 of root labeled $p^{=}_{\text{sname}}$, we move to N_1 and process $p^{\approx_{\text{JD}}}_{\text{description}}$. Since there is no child of N_1 labeled $p^{\approx_{\text{JD}}}_{\text{description}}$, we create a new N_5 and label (N_1, N_5) as $p^{\approx_{\text{JD}}}_{\text{description}}$. Since all predicates in φ_2 are processed, N_5 is a leaf node, whose associated rule is φ_2. □

Intuitively, given (t_1, t_2) and $\varphi \in \Delta$, if φ is more likely to be a witness at (t_1, t_2), it should be evaluated earlier. Motivated by this, we compute the probability for $\varphi : X \rightarrow l$ to be a witness on D as:

$$\text{wp}(\varphi, D) = \prod_{p \in X} \text{sp}(p, D),$$

if we assume the satisfaction of predicates in X as independent events; intuitively, if all predicates in X are satisfied, φ is a witness. If this does not hold, we can estimate $\text{wp}(\varphi, D)$ by re-using the training data collected from historical logs, *i.e.*, we can estimate $\text{wp}(\varphi, D)$ to be the proportion of sampled tuple pairs such that φ is a witness. Since the evaluation of MDs in Δ is guided by edge scores during DFS on $\mathcal{T}$, we define the score of a given edge e based on $\text{wp}(\varphi, D)$.

Edge score. For each MD φ, we denote by ρ_φ the path of $\mathcal{T}$ from root to the leaf whose associated MD is φ. We compute the set of MDs φ in Δ such that the given edge e is part of ρ_φ and denote it by Ψ_e, *i.e.*, $\Psi_e = \{\varphi \in \Delta \mid e \text{ is part of } \rho_\varphi\}$. The score of edge e is $\text{score}(e) = \max_{\rho_\varphi \in \Psi_e} \text{wp}(\varphi, D)$. This said, edges leading to promising MDs will have high scores and thus, will be explored early via DFS on $\mathcal{T}$.

Example 8: Let $\text{sp}(p^{=}_{\text{sname}}, D) = 0.4$ and $\text{sp}(p^{\approx_{\text{JD}}}_{\text{description}}, D) = 0.2$. Then $\text{wp}(\varphi_2, D) = 0.4 \times 0.2 = 0.08$. Assume that we also compute $\text{wp}(\varphi_1, D) = 0.048$. Then the score of edge $e = (N_0, N_1)$ is $\max\{\text{wp}(\varphi_1, D), \text{wp}(\varphi_2, D)\} = 0.08$, since e is part of both ρ_{φ_1} and ρ_{φ_2}. □

Complexity. It takes EPG $O(c_{\text{unit}}|\mathcal{P}| + |\mathcal{P}|\log(|\mathcal{P}|) + |\varphi||\Delta|)$ time to generate the execution plan, where c_{unit} is the unit time for computing the cost-effectiveness of a predicate. This is because the predicate ordering can be obtained in $O(c_{\text{unit}}|\mathcal{P}| + |\mathcal{P}|\log(|\mathcal{P}|))$ time and the tree can be constructed in $(|\varphi||\Delta|)$ time, by scanning Δ once.

Remark. Note that generating an execution plan is fast and it will not be the bottleneck. We will report the time breakdown in Section 7 to verify this claim. Moreover, as a by-product of ensuring the predicate ordering, we can reuse the evaluation results of common "prefix" predicates (*i.e.*, the common predecessors in $\mathcal{T}$).

Example 9: We evaluate $\mathcal{T}$ in Figure 4(b) for (t_1, t_5) in D. After evaluating $p^{=}_{\text{sname}}$, we find that $h(t_1, t_5) \not\models p^{\approx_{\text{JD}}}_{\text{description}}$ and thus we cannot move to N_5. Then DFS will return back to N_1 and continue to check unexplored children of N_1 (*i.e.*, N_2). In this way, the common "prefix" predicate $p^{=}_{\text{sname}}$ of φ_1 and φ_2 is only evaluated once. □

6 OPTIMIZATIONS AND SCHEDULING

To improve resource utilization, it requires exploring a wide design space [62]. Below we present a family of hardware-aware optimization and scheduling techniques that exploit GPU characteristics for massive parallelism. Our novelty includes: (a) efficient device execution of an execution plan (Section 6.1), (b) a workload balancing strategy (Section 6.2), and (c) collaboration of multiple GPUs (Section 6.3).

6.1 Execution plan on GPUs

The execution plan $\mathcal{T}$, initially generated on CPUs, will undergo the evaluation on GPUs in a DFS manner. However, DFS-based tree traversal on GPUs is not efficient due to thread divergence. Worst still, recursive implementations exacerbate performance drops. This is because recursion incur more memory cost and message payloads, as each recursive call adds recursive function to the stack [19] and pass message. Code transformations can help reduce such performance drops. Moreover, although we can reuse the results of "prefix" predicates via DFS on $\mathcal{T}$, some predicates may still be evaluated repeatedly, *e.g.*, $p^{\approx_{\text{JD}}}_{\text{description}}$ in φ_2 and φ_3. Optimized data structures are required to harness the processing power of GPUs.

Tree traversal on GPUs. Note that upon completion of the tree construction, the evaluation order is fixed. Thus the DFS traversal

Table 2: Branch statistics experiment

	Wait stalls	Branch efficiency (%)	Avg. active threads per warp
noSeq	4.3	89.9	15.1
HyperBlocker	4.1	96.4	28.2

of the tree on CPUs can be translated to a *sequential execution path*, which is an ordered list of predicates, on GPUs (see Figure 4 (c) for the sequential execution path of the tree in Example 5).

We maintain two structures for each predicate p in the execution path: an index buffer and a function pointer buffer, which store the indices of attributes compared in p, and the function pointer of the comparison operator in p, respectively, *e.g.,* for predicate $p^{=}_{\text{sname}}$: t.sname = s.sname, its comparison operator is "=" and its attribute index is 3 since sname is the 3rd attribute in schema Products. In addition, at the end of each rule, we set a checkpoint (CP). When a GPU thread encounters a CP, it knows that the undergoing tuple pair satisfies a rule and it can skip the subsequent computation.

Reusing computation. To avoid repeated evaluation, we additionally maintain a bitmap for all predicates on GPUs. The bit of a predicate p is set true if p has been evaluated. If this is the case, we can directly reuse the result from previous computation.

Remark. *sequential execution path* is not only simple but also efficient. An in-depth analysis reveals its benefit. As a toy example, we run the recursively implementation of HyperBlocker, denoted as $\text{HyperBlocker}_{\text{noSeq}}$, on TFACC with two rules of each with 7 predicates. We measure the detail of HyperBlocker and $\text{HyperBlocker}_{\text{noSeq}}$ using NSight [10], a development tools provided by NVIDIA for profiling applications. Table 2 reports their wait stalls, *Branch efficiency*, and Avg. active threads per warp respectively. In particular:

- *Wait stalls* refer to periods where the execution of instructions is delayed due to various dependencies.
- *Branch efficiency* states the ratio of uniform control flow decisions over all executed branch instructions. Higher values are better, as warps more often take a uniform code path.
- *Avg. active threads per warp* indicates what fraction of threads of warps in a kernel have been active on an average. It can also be seen as a measure that is inversely proportional to the overall thread divergence.

HyperBlocker shows lower stalls, higher *Branch efficiency* and higher *Avg. active threads per warp*, resulting in a lower running time. The reasons are twofold: (1) when evaluating repeated predicates within a *sequential execution path*, potential branch divergence is avoided, and the number of operations is reduced; (2) a *sequential execution path* benefits from better data locality and avoids stack/backtracking for each thread, resulting in fewer stalls.

6.2 Workload balancing

As mentioned in Section 3, instructions in GPUs are issued in a warp of threads. All threads in a warp execute the same instruction in lockstep. However, a thread is idle if other threads in a warp have longer execution times, leading to performance degradation.

To address this issue, we propose two GPU-oriented strategies,

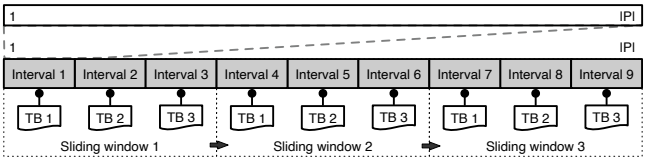


Figure 5: Parallel sliding windows

namely *parallel sliding windows (PSW)* and *task-stealing*.

Parallel sliding windows (PSW). Given a data partition P, PSW processes it with only a small number of index jumps; it also helps GPUs evenly distribute the workload across SMs.

More specifically, PSW processes a partition P in 3 steps:

(1) We divide P into $\frac{|P|}{n_t}$ intervals, where each interval consists of n_t tuples. These intervals are processed with a fixed-size window, which slides over the intervals from left to right. Within each sliding window, we assign an interval to a Thread Block (TB) with warps of 32 threads that are executed in lockstep. Each thread in a TB is responsible for comparing a tuple in the interval with all tuples in P.

(2) Assume that a thread is responsible for tuple t_i. Then this thread compares t_i with each tuple t_j in P according to the plan $\mathcal{T}$ and evaluates whether (t_i, t_j) is a potential match.

(3) When all threads of a TB finish their evaluation, this TB writes the results back to the host memory and it will move on to process the next interval in the next sliding window.

This process continues until all intervals are processed and in total, it requires $\frac{|P|}{n_t n_w}$ sequential index jumps for each TB, where n_w is the size of the sliding window. Moreover, as demonstrated in [46], the sliding window-like execution method ensures that each TB is assigned approximately equal intervals.

Example 10: As shown in Figure 5, a data partition P is divided 9 intervals and the size of the sliding window is 3 (*i.e.,* $n_w = 3$). Interval 1 is assigned to TB1, where each thread in TB1 will compare a tuple in Interval 1 with all tuples in P. When all threads of TB1 finish evaluation, TB1 moves on to process Interval 4. □

Remark. Noted that PSW is inter-tuple-level parallelism. We also explored three alternative design choices to distribute workload to a warp and found that it is best suited for blocking tasks. Here, we discuss these choices and explain the reasoning.

Consider (1) *Inter-Rule-Level Parallelism*: In this approach, the warp executes an entire set of rules in parallel. The system evaluates all rules simultaneously, each on separate threads of a warp. This method is inefficient due to branch divergence and workload skewness among different rules; (2) *Inter-Predicate-Level Parallelism*: the warp executes rules sequentially, but within each rule, the predicates are evaluated in parallel, with each thread corresponding to a predicate. The warp can move to the next rule only after all predicates of the current rule are completely evaluated. This results in synchronization costs, where threads have to wait for results from other threads, and disrupts the execution plan, causing low-priority predicates to be evaluated (Section 5). (3) *Intra-Predicate-Level Parallelism*: In this approach, the evaluation of predicates within a rule is done sequentially, but each predicate evaluation utilizes multiple threads of a warp for fine-grained parallel processing. For example,

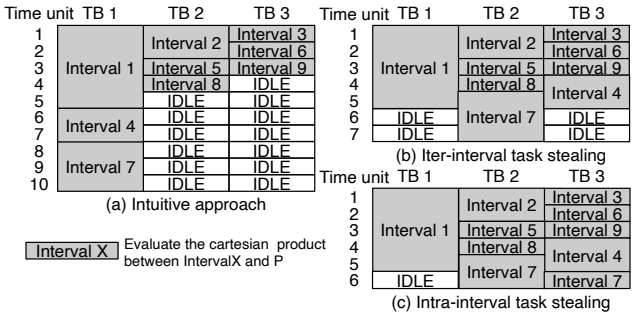


Figure 6: Task-stealing

for two long texts, a warp of threads can be used to compute their similarity. This method leads to wasted computational power when encountering simpler, integer-based predicates.

PSW won out in the end, as it do not incur redundant computation or disrupts the execution plan, *e.g.,* it is 7.3× faster than *Inter-Predicate-Level Parallelism* on DBLP-ACM.

Task-stealing. Since intervals are assigned via a sliding window, each TB will process roughly equal intervals. However, the execution times of different intervals are not the same. For example, one pair may be quickly identified as a potential match if the first MD checked is its witness, while another pair is found as a mismatch until all MDs in Δ are iterated. Even for the same attribute, the evaluation times on different tuples are different (*e.g.,* long text vs. short text). This said, the workload of all TBs can still be imbalanced.

Example 11: Continuing Example 10, assume that three TBs process the 9 intervals, as shown in Figure 6(a). Even though the number of intervals assigned to each TB are the same, the execution times can still skew, *e.g.,* the total time units required by TB1 and TB3 are 10 and 3, respectively; this means TB3 is idle for 7 time units, indicating an imbalanced workload. □

Below we introduce both the inter-interval and intra-interval task-stealing strategies to further balance the workload.

Inter-interval task-stealing. It is commonly observed that the execution times of some TBs are longer than the others. In this case, a large number of TBs are idle, waiting for the slowest TB.

In light of this, we employ an inter-interval task-stealing strategy to allow idle TBs to work on not-yet-processed intervals.

More specifically, we maintain a bitmap (a sequence of bits), for its simplicity and checking efficiency, such that each bit indicates the status of an interval. The bitmap is stored at global memory, so that TBs can steal not-yet-processed intervals from each other. Then each TB processes intervals in two stages: (a) It first processes its assigned intervals one by one. Whenever it starts to process an interval, it checks the corresponding bit in the bitmap. If the bit is set to false (*i.e.,* the interval is not yet processed), it processes this interval and sets the bit true. (b) If this TB is idle after it finishes all assigned intervals, it traverses the bitmap to steal a not-yet-processed interval, by setting the corresponding bit true and processing that interval. Other TBs will skip the interval if it has been stolen.

Example 12: In Example 11, TB3 finishes its assigned intervals after 3 time units. Then it checks the bitmap and steals Interval 4 (whose bit in the bitmap is then set true) from TB1; similarly for TB2. Compared with the time in Figure 6 (a), the total time units are reduced from 10 to 7 after stealing in Figure 6 (b). □

Intra-interval task-stealing. Recall that a thread responsible for t_i will compare t_i with all t_j in P to decide whether (t_i, t_j) is a potential match. Since the evaluation of distinct pairs is independent, it makes room for intra-interval task-stealing from executing intervals. To facilitate this, we maintain two integers start and end, initialized to 1 and $|P|$, respectively, indicating the range of tuples to be compared with t_i. Then this thread starts to evaluate (t_i, t_{start}). Upon completion, it sets start = start + 1 and moves on to the next pair (t_i, t_{start}). When start = end, this thread finishes all evaluation for t_i.

Based on this, the intra-interval task-stealing strategy works as follows. If TB_a finishes all its assigned intervals and there are no not-yet-processed intervals, it identifies an executing TB_b and iterates all threads in TB_b, so that the i-th thread in TB_a steals half workload (*i.e.,* half pairs remained to be compared) from the i-th thread in TB_b. Assume the two integers maintained for the i-th thread in TB_b (resp. TB_a) are start_b and end_b (resp. start_a and end_a). We set $\text{start}_a = \text{start}_b + \frac{\text{start}_b + \text{end}_b}{2}$, $\text{end}_a = \text{end}_b$, and $\text{end}_b = \text{start}_b + \frac{\text{start}_b + \text{end}_b}{2} - 1$, *i.e.,* the latter half of tuples remained to be compared is stolen from each thread in TB_b.

Example 13: Continuing with Example 12, when TB3 finishes Interval 4 stolen from TB1 in Figure 6 (b), it finds there are no not-yet-processed intervals. However, since TB2 is still evaluating Interval 7, TB3 steals half of the remaining workload from TB2, further saving 1 time unit, as shown in Figure 6 (c). □

6.3 GPU collaboration

A GPU server nowadays usually has multiple GPUs connected via NVLink [47] or PCIe. Scaling blocking to multiple GPUs is beneficial for jointly utilizing the computation and storage powers of GPUs.

In pursuit of this, one can split data into n_{GPU} partitions evenly, so that each GPU handles one partition [56], or assign partitions to GPUs in a round-robin manner [60]. These, however, do not work well since (a) workload can be imbalanced due to different execution times of partitions, (b) there are potential waiting times for pending partitions when multiple partitions compete for the same PCIe lane (*e.g.,*16 lanes for V100 GPU). Moreover, (c) these methods independently conduct rule-based blocking on partitions and do not effectively handle scenarios where tuples t_i and t_j reside on different partitions, resulting in elevated false-negative rates. To address this issue, recent techniques such as duplicating tuples in multiple partitions [21, 23] were proposed. However, these approaches incur both memory and data transfer costs on a single machine.

In light of these, we present a collaborative approach integrating partitioning and scheduling strategies, where the former aims at minimizing data redundancy while reducing false negatives and the latter prioritizes load balancing and minimizes resource contention.

Data Partitioning. A straightforward method for data partitioning it to scan all tuples and compute a hash value for each tuple based on a subset of attributes. Tuples with same hash value are grouped together. Instead of sacrificing the accuracy (*e.g.,* using only one hash function) or unnecessarily duplicating too many tuples, HyperBlocker applies s hash functions to obtain s partition-

key values, where s is the number of children N_c of the root node N_0 in the execution tree $\mathcal{T}$; each hash function is constructed from the predicate associated with an edge (N_0, N_c). For example, if the predicate is $t.\text{sname} = s.\text{sname}$, then we hash tuples in D based on their values in attribute sname, so that tuples with the same sname are put in one partition. The benefits are two-fold: (1) According to the construction of execution tree $\mathcal{T}$, the hash functions we constructed are often discriminative and might be shared by multiple rules. This said, we can obtain a good hashing result with a few hashing functions. (2) We can assign each tuple a branch ID, indicating the hash function used. Only tuple pairs that share the same hash function are further compared, thereby reducing redundant computations incurred by multiple hash functions.

Scheduling. We adopt a two-step scheduling strategy to allow load balance and reduce resource competition. Initially, data partitions and GPUs are hashed to random locations on a unit circle, similar to [52]. If a partition P_i is assigned to an *ineligible* GPU (where no idle core can process it or the PCIe channel is occupied), it is rerouted to the nearest available GPU in a clockwise direction and the kernels of that GPU evaluate $\mathcal{T}$ on the local data P_i independently.

Remark. If data partitioning is done by a hashing function from a similarity predicate p, it is possible that $h(t_1, t_2) \models p$ but t_1 and t_2 reside on different partitions, leading to potential false negatives in blocking. In this case, a kernel with local data P_i can optionally "pull" partition P_j from another kernel and evaluate $\mathcal{T}$ across P_i and P_j. The pull operation retrieves data from locations outside P_i, depending on whether P_i and P_j reside on the same GPU. If P_i and P_j reside on the same GPU, the pull operation is executed directly without any data transfer. Otherwise, the pull operation for P_j can be carried out using `cudaMemcpyPeer()` to take the advantages of high bandwidth and low latency provided by NVLink.

7 EXPERIMENTAL STUDY

We evaluated HyperBlocker for its (1) accuracy-efficiency and (2) scalability on both real-life and synthetic data. We also tested HyperBlocker with various parameters and broke down the time of HyperBlocker to analyze its (3) performance bottlenecks.

Experimental setup. We start with the experimental setting.

Datasets. We used eight real-world public datasets in Table 3, which are widely adopted ER benchmarks and real-life datasets [3, 4, 9]. We also generated a synthetic dataset TPCH using TPCH dbgen [5].

Fodros-Zagat, DBLP-ACM, DBLP-Scbolar, IMDB, Songs, and NCV have 112, 2294, 5348, 0.27M, 1.2M, and 0.5M tuple pairs labeled as matches or mismatches as the ground truth (GT), respectively. For TFACC, $\text{TFACC}_{\text{large}}$ and TPCH, we follow the setting of [31], assuming that the original datasets were correct, and randomly duplicated tuples as noises. The training data consists of 50% of the ground truth and 50% of randomly selected noise.

Baselines. As remarked in Section 3, although HyperBlocker is designed as a blocker, it can be used with or without a matcher. Thus, below we not only compared HyperBlocker against widely used blockers but also integrated ER solutions (*i.e.,* blocker + matcher).

We implemented three distributed ER blocking systems: (1) Dedoop [1, 42], (2) SparkER [12, 33], and (3) DisDedup [8, 21],

Table 3: Datasets

Dataset	#Tuples	Max #Pairs	#GT Pairs	#Attrs	#Rules	#Partitions
Fodors-Zagat	866	1.8×10^4	112	6	1	1
DBLP-ACM	4591	6.0×10^6	2294	4	10	8
DBLP-Scholar	66881	1.7×10^8	5348	4	10	8
IMDB	1.5M	8.1×10^{10}	0.2M+	6	10	128
Songs	0.5M	2.7×10^{11}	1.2M	8	10	128
NCV	2M	1.0×10^{12}	0.5M+	5	10	512
TPCH	4M	1.6×10^{13}	#	8	30	512
TFACC	10M	1.0×10^{14}	#	16	50	1024

where DisDedup is the state-of-the-art CPU-based parallel ER system, designed to minimize both the communication and computational costs; Dedoop focuses on optimizing the computational cost; and SparkER incorporates Blast blocking [61] on the Spark platform [11].

In addition to these systems, we compared HyperBlocker with four GPU-based baselines: (4) DeepBlocker [66], (5) GPUDet [32], (6) Ditto [2, 49] and (7) $\text{DeepBlocker}_{\text{Ditto}}$, where DeepBlocker is the state-of-the-art (SOTA) in DL-based blocker based on GPUs, GPUDet implements well-known similarity algorithms for tuple pair comparison on GPUs, Ditto, the state-of-the-art matcher of ER, is built upon Torch GPU interfaces[13] and $\text{DeepBlocker}_{\text{Ditto}}$ uses DeepBlocker as the blocker and Ditto as the matcher, respectively. Note that Ditto takes tuple pairs as input, instead of a relation D of tuples as the other methods. Due to the high cost of Ditto, it is infeasible to feed the Cartesian product of all tuples in each partition to Ditto. Thus, for each tuple in the ground truth, we adopted a similarity-join method Faiss [40] to retrieve the top-2 nearest neighbors, as a preprocessing step of Ditto. Denote the resulting baseline by $\text{Ditto}_{\text{top2}}$. Here Faiss [40] is an open-source library. It offers GPU implementations, allowing users to exploit GPUs for faster similarity search. Since Faiss often serves as a key component of many ER solutions [25, 66], we also compared Faiss in an extended study.

Besides, we also implemented the following nine variants: (1) HyperBlocker, the basic blocker with all optimizations applied. (2) $\text{HyperBlocker}_{\text{Ditto}}$, an improved version that uses HyperBlocker as the blocker and Ditto as the matcher, respectively. Note that $\text{HyperBlocker}_{\text{Ditto}}$ is particularly compared against $\text{Ditto}_{\text{top2}}$ to show how we speed up the overall ER. (3) $\text{HyperBlocker}_{\text{noEPG}}$, a variant without EPG (Section 5). (4) $\text{HyperBlocker}_{\text{noPO}}$ that evaluates predicates in a random order. (5) $\text{HyperBlocker}_{\text{noRO}}$ that evaluates rules in a random order. (6) $\text{HyperBlocker}_{\text{noHO}}$ that disables all hardware-level optimizations (Section 6). (7) EvenSplit and (8) RoundRobin, that uses the EvenSplit and RoundRobin strategies for scheduling (Section 6.3), respectively.

Rules. We mined MDs using the algorithm [63]. We checked and adjusted the MDs discovered manually to ensure their correctness. The number of MDs adopted for each dataset is shown in Table 3.

Measurements. Following typical ER settings, we measured the performance of each method (blocker, matcher or the combination of the two) in terms of the runtime and the F1-score, defined as F1-score = $\frac{2\times\text{Prec}\times\text{Rec}}{\text{Prec}+\text{Rec}}$. Here Prec is the ratio of correctly identified tuple pairs to all identified tuple pairs and Rec is the ratio of correctly identified tuple pairs to all tuple pairs that refer to the same real-world entity. All methods aim to achieve high Rec, Prec and F1-scores. Besides, along the same line as [66], we also report the candidate set size ratio (CSSR), defined as $\frac{|\text{Ca}(P)|}{|P|\times|P|}$, when comparing with DeepBlocker, to show the portion of tuple pairs that require further

Table 4: Comparison with the SOTA DL-based blocker

Method	Metric	Dataset		
		Fodors-Zagat	DBLP-Scholar	DBLP-ACM
DeepBlocker	Rec (%)	100 (+0)	98 (+5)	98 (+4)
	CSSR (‰)	15.1 (+14.5)	2.3 (+1.1)	2.2 (+1.8)
	Time (s)	6.1 (122×)	72.8 (11.0×)	8.0 (10.0×)
	Host Mem. cost (GB)	9.9 (49.5×)	14.0 (23.3×)	10.3 (34.3×)
	Device Mem. cost (GB)	0.9 (1.8×)	1.1 (1.6×)	0.9 (1.5×)
HyperBlocker	Rec (%)	100	93	94
	CSSR (‰)	0.6	1.2	0.4
	Time (s)	0.05	6.6	0.8
	Host Mem. cost (GB)	0.2	0.6	0.3
	Device Mem. cost (GB)	0.5	0.7	0.6

comparison by the matcher (*i.e.,* the comparisons that cannot be reduced by the blocker). The smaller the CSSR, the better the blocker.

Environment. We run experiments on a Ubuntu 20.04.1 LTS machine powered with 2 Intel Xeon Gold 6148 CPU @ 2.40GHz, 4TB Intel P4600 PCIe NVMe SSD, 128GB memory, and 8 Nvidia Tesla V100 GPUs with 32 GB of memory. The programs were compiled with CUDA-11.0 and GCC 7.3.0 with -O3 compiler. DisDedup, SparkER, and Dedoop were run on a cluster of 30 HPC servers, powered with 2.40GHz Intel Xeon Gold CPU, 4TB Intel P4600 SSD, 128GB memory.

Default parameters. Unless stated explicitly, we used the following setting. The maximum number of predicates in an MD is set to 10. The number m of data partitions on each dataset is shown in Table 3. The size n_t of intervals and the size n_w of the sliding window are 256 and 1024, respectively. For the shallow model $\mathcal{N}$, we adopted a small neural network regression model with 3 hidden layers, with 2, 6 and 1 neurons, respectively. We used ReLU [54] as the activation function and Adam [41] as the optimizer. As remarked earlier, the training of $\mathcal{N}$ is fast and can be done offline. Thus we exclude the training time from the runtime. We used one GPU by default.

Experimental results. We report our findings as follows.

Exp-1: Motivation Study. We first motivate our study by comparing HyperBlocker, the proposed rule-based blocker, with the state-of-the-art DL-based blocker DeepBlocker in Table 4, where the bracket next to each metric of DeepBlocker gives its difference or improvement factor compared with that of HyperBlocker.

DL-based blocking vs. Rule-based blocking. We report recall, CSSR, runtime, and (both host and device) memory costs for both methods. Consistent with [66], for DeepBlocker, each tuple was paired with the top-K tuples with the highest cosine similarity to form the initial candidate pairs, where $K = 5, 150$ and 5 on Fodors-Zagat, DBLP-Scholar, and DBLP-ACM, respectively. As remarked earlier in Section 1, both methods have their strengths. (1) HyperBlocker can effectively reduce the number of tuple pairs that need further comparison while maintaining high Rec (>93%), *e.g.,* its average CSSR is 5.8‰ less than DeepBlocker. (2) HyperBlocker is at least 10× faster than DeepBlocker. (3) HyperBlocker consumes less memory than DeepBlocker, *e.g.,* the host memory it consumes is at least 23.3× less than DeepBlocker. (4) Note that the Rec of HyperBlocker is slightly lower than DeepBlocker, which is still acceptable given its convincing speedup and memory saving, since the primary goal of a blocker is to improve the time efficiency and scalability of ER, not to improve the accuracy of ER (the goal of a matcher).

Exp-2: Accuracy-efficiency. We next report the F1-scores and runtime of all blockers and integrated ER solutions (*i.e.,* blocker + matcher) in Table 5. Here DeepBlocker pairs each tuple with its top-2 tuples as initial candidate pairs. For all blockers, the bracket next to each F1-score (resp. time) gives the difference (resp. speedup) in F1-score (resp. time) to HyperBlocker (marked yellow). For fair comparison, the brackets of each integrated ER solution, give the improvement compared with $\text{HyperBlocker}_{\text{Ditto}}$ (marked yellow).

Accuracy. We mainly analyze the F1-scores of HyperBlocker, which are consistently above 0.8 over all datasets. Besides, we find:

(1) HyperBlocker outperforms CPU-based distributed solutions, *e.g.,* it achieves up to 0.29, 0.74, and 0.72 improvement in F1-score against Dedoop, DisDedup, and SparkER, respectively, even though the former two are integrated with matchers. This is because these existing distributed solutions exploit partition parallelism only. However, with numerous partitions, their recall degrades substantially since tuples in different partitions are identified as mismatches.

(2) Compared with GPU-based baselines GPUDet, DeepBlocker, $\text{Ditto}_{\text{top2}}$ and $\text{DeepBlocker}_{\text{Ditto}}$, HyperBlocker has comparable accuracy. In particular, it even beats $\text{Ditto}_{\text{top2}}$, the SOTA matcher, by 0.17 F1-score in IMDB. This shows that even without a matcher, HyperBlocker alone is already accurate in certain cases. Moreover, DeepBlocker and $\text{DeepBlocker}_{\text{Ditto}}$ struggle to handle large datasets. When the size of dataset approaches the million-scale, their performance degrades significantly and cannot finish in 3 hours. This again motivates the need for rule-based alternatives.

(3) Combing HyperBlocker with Ditto, $\text{HyperBlocker}_{\text{Ditto}}$ further boosts the accuracy, achieving the best F1-score in Songs. Nevertheless, DL-based solutions still have the best F1-scores in the other datasets, justifying our claim that none of rule-based blocking and DL-based blocking can dominate the other in all cases.

(4) $\text{HyperBlocker}_{\text{noEPG}}$ and $\text{HyperBlocker}_{\text{noHO}}$ are as accurate as HyperBlocker, since they only differ in the optimizations.

Runtime. We next report the runtime. (1) HyperBlocker runs substantially faster than all baselines since it effectively collaborates CPUs/GPUs, exploring novel optimization and scheduling strategies under the shared memory architecture, *e.g.,* it is at least 58×, 68.4×, 13.8×, 25.1×, 10.4×, 11.3× and 15.5× faster than Dedoop, DisDedup, SparkER, GPUDet, DeepBlocker, $\text{Ditto}_{\text{top2}}$ and $\text{DeepBlocker}_{\text{Ditto}}$, respectively. (2) $\text{HyperBlocker}_{\text{Ditto}}$ is slower than HyperBlocker as expected since it performs additional matching. Nonetheless, $\text{HyperBlocker}_{\text{Ditto}}$ is at least 1.4× (resp. 2.0×) faster than $\text{Ditto}_{\text{top2}}$ (resp. $\text{DeepBlocker}_{\text{Ditto}}$). Given its comparable F1-score, we substantiate our claim (Section 1) that blocking is a crucial part of the overall ER process. (3) HyperBlocker is at least 12.4× and 3.4× faster than $\text{HyperBlocker}_{\text{noEPG}}$ and $\text{HyperBlocker}_{\text{noHO}}$, respectively, verifying the usefulness of execution plans and hardware optimizations.

Impact of m. The number m of data partitions mainly affects both the recall [66] and the efficiency. Varying m from 16 to 1024 on NVC, Figure 7(a) reports the recall (the right y-axis) and runtime (the left y-axis). As shown there, both metrics of HyperBlocker decreases with increasing m. This is because when there are more partitions, both the number of pairwise comparisons and the candidate matches that can be identified in each partition are reduced.

Exp-3: Scalability. In the following, we evaluated our scalability using large datasets TPCH, TFACC, and $\text{TFACC}_{\text{large}}$.

Table 5: Accuracy & runtime on benchmarks where "*" denotes that integrating HyperBlocker with Ditto does not improve the F1-score and thus we report the result of HyperBlocker, and "/" denotes that the F1-score cannot be computed within 3 hours.

Method	Backend	Category	DBLP-ACM		IMDB		Songs		NCV	
			F1-score	Time (s)	F1-score	Time (s)	F1-score	Time (s)	F1-score	Time (s)
SparkER	CPU	Blocker	0.77 (-0.17)	11.0 (13.8×)	0.31 (-0.65)	242.9 (26.4×)	0.08 (-0.72)	203.4 (15.2×)	0.26 (-0.66)	229.3 (49.8×)
GPUDet	GPU	Blocker	0.92 (-0.02)	20.1 (25.1×)	0.94 (-0.02)	323.8 (35.2×)	0.80 (+0)	404.8 (30.2×)	0.90 (-0.02)	1252.6 (272.3×)
DeepBlocker	GPU	Blocker	0.98 (+0.04)	8.3 (10.4×)	/	>3h	/	>3h	/	>3h
HyperBlocker$_{noEPG}$	GPU	Blocker	0.94 (+0)	9.9 (12.4×)	/	>3h	0.80 (+0)	1904.1 (142×)	0.92 (+0)	2408.6 (523.6×)
HyperBlocker$_{noHO}$	GPU	Blocker	0.94 (+0)	9.5 (11.9×)	0.96 (+0)	472.6 (51.4×)	0.80 (+0)	45.0 (3.4×)	0.92 (+0)	35.9 (7.8×)
HyperBlocker	GPU	Blocker	0.94	0.8	0.96	9.2	0.80	13.4	0.92	4.6
Dedoop	CPU	Blocker+Matcher	0.90 (-0.08)	59.4 (9.4×)	0.67 (-0.29)	534.0 (58×)	0.80 (-0.08)	7643.4 (6.5×)	/	>3h
DisDedup	CPU	Blocker+Matcher	0.45 (-0.53)	94.0 (14.9×)	0.67 (+0)	644.0 (70.0×)	0.06 (-0.82)	917.0 (0.8×)	/	>3h
Ditto$_{top2}$	GPU	Blocker+Matcher	0.98 (+0)	9.0 (1.4×)	0.79 (+0)	6741.2 (732.7×)	0.88 (+0)	2308.6 (2.0×)	0.97 (+0.03)	381.8 (2.1×)
DeepBlocker$_{Ditto}$	GPU	Blocker+Matcher	0.99 (+0.01)	12.4 (2.0×)	/	>3h	/	>3h	/	>3h
HyperBlocker$_{Ditto}$	GPU	Blocker+Matcher	0.98	6.3	*0.96	*9.2	0.88	1179.0	0.94	180.6

Varying $|\varphi|$. We next tested the impact of the number $|\varphi|$ of predicates in each MD φ, varying $|\varphi|$ from 2 to 10 in TFACC. The result is shown in Figure 7(c). (1) HyperBlocker takes longer with larger $|\varphi|$, as expected. (2) HyperBlocker is feasible in practice, *e.g.,* when $|\varphi| = 10$, it only takes 135.2 seconds. (3) On average, HyperBlocker demonstrates 32.5× speedup compared to HyperBlocker$_{noPO}$ (which evaluates predicates in a random order). This justifies the importance of predicate ordering in the efficiency of rule-based blocking.

Varying $|\Delta|$. We evaluated the impact of the number $|\Delta|$ of MDs in Δ. Fixing $m = 1024$, we varied $|\Delta|$ from 1 to 50 in Figure 7(d). As expected, HyperBlocker takes longer with more rules, *e.g.,* it takes 523.2 seconds when $|\Delta| = 50$. Nonetheless, HyperBlocker is faster than HyperBlocker$_{noRO}$, which evaluates rules in a random order.

Exp-4: Performance. Finally, we evaluated the performance of HyperBlocker in different settings of (1) interval sizes (n_t), (2) synchronization and (3) schedulers. We also (4) break down its runtime.

Time Breakdown. The overall process of HyperBlocker can be decomposed into four phases: (1) I/O, which loads data from disk to host memory; (2) Data partition, which performs data partitioning; (3) Plan generation, which constructs the execution plan; (4) Computation, that transfers data from the host to devices and computes candidate matches. The time breakdown of HyperBlocker is shown in Figure 7(g), where we used 4 GPUs and fixed $m = 1024$. Varying the scale factor of TFACC from 20% to 100%, we find that (1) the execution plan generation (the bottom red line) is fast, *e.g.,* less than 0.5 seconds; and (2) GPU operations do not dominate the overall complexity, which is consistent with the finding in [36].

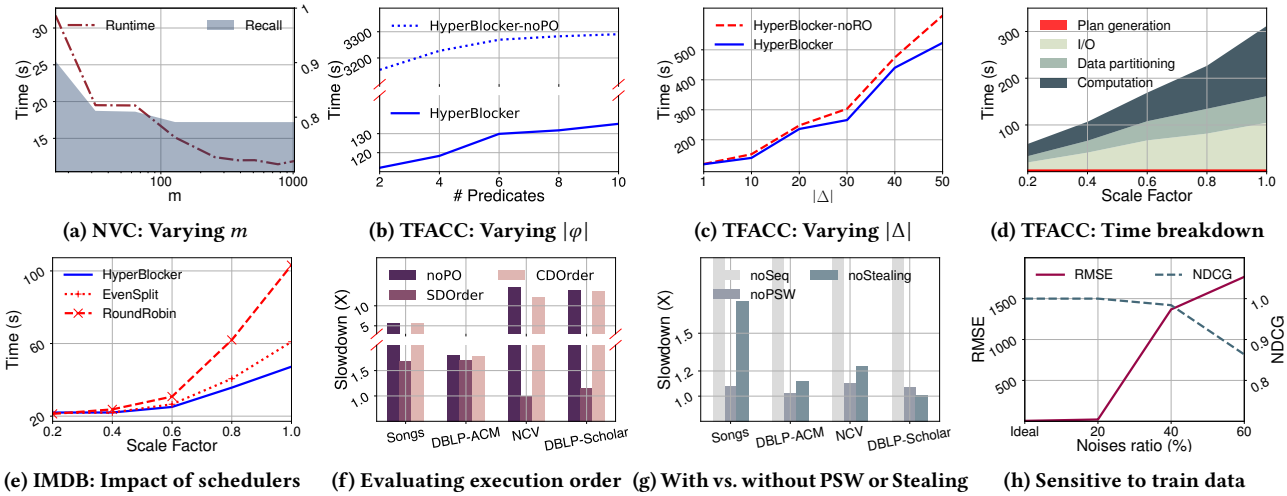
Impact of task schedulers. We tested the impact of task schedulers with 4 GPUs, fixing $m = 16$ for HyperBlocker and RoundRobin and varying the dataset size from 20% to 100% in IMDB. HyperBlocker works better than the two variants, *e.g.,* when the scale factor is 100%, HyperBlocker is 1.3× and 2.2× faster than RoundRobin and EvenSplit, respectively. Moreover, HyperBlocker shows better scalability, whereas the others exhibit a steep rise in time with increasing $|D|$. This verifies the effectiveness of our scheduling strategy.

Exp-5: Ablation. To illustrate the contribution of components to HyperBlocker, we remove several key components and evaluated the resulting running time. We let $m = 1$ and use only one rule for the experiments on Songs, DBLP-ACM, NCV, and DBLP-Scholar.

Evaluating different matching order. We tested four variants with different matching order: (1) noPO corresponding to HyperBlocker$_{noPO}$; (2) SDOrder (self-defined matching order), that fix execution order by forcing " $=''\prec$ " $\approx''$ and give a high priority to number, enumeration over string type. The principle is to prioritize predicates that execute quickly; (3) CEOrder (cardinality estimation-based matching order), that obtain execution order by cardinality estimation. The idea is to prioritize predicates that result in less intermediate results. Figure 7(i) reports their slowdown comparing with HyperBlocker. CEOrder (resp. SDOrder) is slowed by 8.3× (resp. 1.4×), on average. This is because: (a) the main goal of cardinality estimation is to minimize intermediate results, thus CEOrder is easily make predicates with high discrimination power performing early. However, these predicates often come with higher computational complexity. (b) SDOrder prioritizes predicates based solely on their computational cost, leading to the early execution of predicates with high speed but low discrimination power resulting in redundant evaluations. In contrast, HyperBlocker learns from the data and considers both factors to determine the most "suitable" orderings.

Evaluating hardware-aware optimizations. We discard *sequential execution path*, *PSW* and *task stealing* components and reports them relatively slowdown in Figure 7(j). Concretely, (1) noSeq disables *sequential execution path* and use recursively implementation. (2) noPSW directly assigns continuous intervals, calculated as $\frac{\text{nIntervals}}{\text{nTBs}}$, to each TB, where nIntervals is the total number of intervals and nTBs is the number of TBs; (3) noStealing that disables both inter and intra-interval task stealing, allowing GPU to automatically schedule a new thread block whenever one is completed. As shown there, (1) noSeq is significantly slower than HyperBlocker, which justifies the need for a *sequential execution path*. (2) noStealing and noPSW are, on average, 1.28× and 1.07× slower than HyperBlocker, respectively. This validates the optimizations for workload balancing. (3) On DBLP-Scholar, PSW and task stealing do not improve system performance well. This is because the benefits of task stealing or PSW heavily depend on the workload distribution among TBs. For DBLP-Scholar, there is not much skewness within the data, which diminishes the effectiveness of these techniques.

Exp-6: Sensitivity of EPG to training data noise. Figure 7(k) illustrates the sensitivity of $\mathcal{N}$ to training data noise and shows the tolerance of EPG (Section 5) to changes in data distribution.



(a) NVC: Varying m **(b) TFACC: Varying $|\varphi|$** **(c) TFACC: Varying $|\Delta|$** **(d) TFACC: Time breakdown**

(e) IMDB: Impact of schedulers **(f) Evaluating execution order** **(g) With vs. without PSW or Stealing** **(h) Sensitive to train data**

Figure 7: Efficiency and scalability of HyperBlocker

Different levels of gaussian noise (ranging from 20% to 60%) were introduced into the training data to simulate changes in data distribution. On DBLP-ACM, the relationship between noise ratio and performance metrics (NDCG [67] and RMSE) was reported where NDCG compares the generated order to an ideal matching order of rules. As shwon there, (1) with increasing noise, the RMSE of $\mathcal{N}$ becomes larger, as expected. (2) The EPG shows high tolerance to distribution changes, with less steep declines in NDCG.

Specifically, we list the generated rules at 40% and 60% noise levels, comparing them to the ideal matching rule, denoted as $\varphi_{40\%}$, $\varphi_{60\%}$, and φ_{Ideal} respectively.

- φ_{Ideal} : $t.\text{year} = s.\text{year} \wedge t.\text{venue} \approx_{\text{JD}} s.\text{venue} \wedge t.\text{authors} \approx_{\text{ED}} s.\text{authors} \wedge t.\text{title} \approx_{\text{JD}} s.\text{title} \rightarrow t.\text{eid} = s.\text{eid}$
- $\varphi_{40\%}$: $t.\text{year} = s.\text{year} \wedge t.\text{authors} \approx_{\text{JD}} s.\text{authors} \wedge t.\text{venue} \approx_{\text{JD}} s.\text{venue} \wedge t.\text{title} \approx_{\text{JD}} s.\text{title} \rightarrow t.\text{eid} = s.\text{eid}$
- $\varphi_{60\%}$: $t.\text{authors} \approx_{\text{JD}} s.\text{authors} \wedge t.\text{venue} \approx_{\text{JD}} s.\text{venue} \wedge t.\text{year} = s.\text{year} \wedge t.\text{title} \approx_{\text{JD}} s.\text{title} \rightarrow t.\text{eid} = s.\text{eid}$

When the data distribution changes to 40%, it simply switches positions of $t.\text{venue} \approx_{\text{JD}} s.\text{venue}$ and $t.\text{authors} \approx_{\text{JD}} s.\text{authors}$. This justify robustness of EPG.

Summary.rewrite We find the following. (1) HyperBlocker outperforms prior blockers and integrated ER solutions, with its novel pipelined architecture, execution plan, and hardware-aware optimizations on GPUs. It is at least 58×, 68.4×, 13.8×, 25.1×, 10.4×, 11.3× and 15.5× faster than Dedoop, DisDedup, and SparkER, GPUDet, DeepBlocker, $\text{Ditto}_{\text{top2}}$ and $\text{DeepBlocker}_{\text{Ditto}}$, respectively. (2) By combining HyperBlocker with Ditto, we save at least 30% time with comparable accuracy. (3) HyperBlocker beats all its variants (except $\text{HyperBlocker}_{\text{Ditto}}$) in both runtime and accuracy, justifying the usefulness of various optimizations used: (a) the execution plan specifies an effective evaluation order, improving the runtime by at least 12.4× and (b) the hardware-level optimizations on GPUs speedups blocking by at least 3.4×. (3) HyperBlocker scales well with various parameters, *e.g.*, the dataset size and the number of GPUs. It completes blocking in 1604s on $\text{TFACC}_{\text{large}}$ with 36M tuples.

8 CONCLUSION

The novelty of HyperBlocker consists of (1) a pipelined architecture that overlaps the data transfer from/to CPUs and the operations on GPUs; (2) a data-aware and rule-aware execution plan generator on CPUs, that specifies how rules are evaluated; (3) a variety of hardware-aware optimization strategies that achieve massive parallelism, by exploiting the hardware characteristic; and (4) a task scheduling strategy to achieve workload balancing across multiple GPUs. Our experimental study has verified that HyperBlocker is much faster than existing CPU-powered distributed systems and GPU-based ER solvers, while maintaining comparable accuracy.

One future topic is to study how to accelerate the evaluation of more sophisticated rules, *e.g.*,REEs [29, 30], using GPUs.

REFERENCES

[1] 2021. Dedoop Source Code. https://dbs.uni-leipzig.de/dedoop.
[2] 2021. Ditto Source Code. https://github.com/megagonlabs/ditto.
[3] 2021. ER Benchmark Dataset. https://dbs.uni-leipzig.de/de/research/projects/object_matching/benchmark_datasets_for_entity_resolution.
[4] 2021. Magellan Dataset. https://sites.google.com/site/anhaidgroup/projects/data.
[5] 2021. TPCH. http://www.tpc.org/tpch/.
[6] 2023. Amazon Duplicate Product Listings. https://www.amazowl.com/amazon-frustration-free-packaging-2-2-2/.
[7] 2024. Code, datasets and full version. XXX.
[8] 2024. DisDedup Source Code. https://github.com/david-siqi-liu/sparklyclean.
[9] 2024. MOT Tests and Results. https://ckan.publishing.service.gov.uk/dataset.
[10] 2024. Nsight Compute Documentation. https://docs.nvidia.com/nsight-compute/.
[11] 2024. Spark. https://spark.apache.org.
[12] 2024. Sparker Source Code. https://github.com/Gaglia88/sparker.
[13] 2024. Torch. https://pytorch.org.
[14] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.
[15] Yasser Altowim and Sharad Mehrotra. 2017. Parallel Progressive Approach to Entity Resolution Using MapReduce. In *ICDE*.
[16] Alexandr Andoni and Piotr Indyk. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM* 51, 1 (2008).
[17] Nils Barlaug. 2023. ShallowBlocker: Improving Set Similarity Joins for Blocking. *arXiv preprint arXiv:2312.15835* (2023).
[18] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. 2017. Enriching Word Vectors with Subword Information. *Trans. Assoc. Comput. Linguistics* 5 (2017), 135–146.
[19] Cheng Cao and Justin Kalloor. [n.d.]. Analysis of High Level implementations for Recursive Methods on GPUs. ([n. d.]).
[20] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proceedings of the VLDB Endowment* (2020).
[21] Xu Chu, Ihab F Ilyas, and Paraschos Koutris. 2016. Distributed data deduplication. *Proceedings of the VLDB Endowment* 9, 11 (2016), 864–875.
[22] Sanjib Das, Paul Suganthan GC, AnHai Doan, Jeffrey F Naughton, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, Vijay Raghavendra, and Youngchoon Park. 2017. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *SIGMOD*. 1431–1446.
[23] Ting Deng, Wenfei Fan, Ping Lu, Xiaomeng Luo, Xiaoke Zhu, and Wanhe An. 2022. Deep and collective entity resolution in parallel. In *ICDE*. 2060–2072.
[24] Gregory Frederick Diamos, Haicheng Wu, Jin Wang, Ashwin Sanjay Lele, and Sudhakar Yalamanchili. 2013. In *PPoPP*. 301–302.
[25] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq R. Joty, Mourad Ouzzani, and Nan Tang. 2018. Distributed Representations of Tuples for Entity Resolution. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1454–1467.
[26] Vasilis Efthymiou, George Papadakis, George Papastefanatos, Kostas Stefanidis, and Themis Palpanas. 2015. Parallel meta-blocking: Realizing scalable entity resolution over large, heterogeneous data. In *IEEE Big Data*. 411–420.
[27] Vasilis Efthymiou, George Papadakis, Kostas Stefanidis, and Vassilis Christophides. 2019. MinoanER: Schema-Agnostic, Non-Iterative, Massively Parallel Resolution of Web Entities. In *EDBT*.
[28] Wenfei Fan, Hong Gao, Xibei Jia, Jianzhong Li, and Shuai Ma. 2011. Dynamic constraints for record matching. *The VLDB Journal* 20 (2011), 495–520.
[29] Wenfei Fan, Ziyan Han, Yaoshu Wang, and Min Xie. 2022. Parallel Rule Discovery from Large Datasets by Sampling. In *SIGMOD*. 384–398.
[30] Wenfei Fan, Ziyan Han, Yaoshu Wang, and Min Xie. 2023. Discovering Top-k Rules using Subjective and Objective Criteria. In *SIGMOD*.
[31] Wenfei Fan, Chao Tian, Yanghao Wang, and Qiang Yin. 2021. Parallel discrepancy detection and incremental detection. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1351–1364.
[32] Benedikt Forchhammer, Thorsten Papenbrock, Thomas Stening, Sven Viehmeier, Uwe Draisbach, and Felix Naumann. 2013. Duplicate detection on GPUs. *HPI Future SOC Lab* 70, 3 (2013).
[33] Luca Gagliardelli, Giovanni Simonini, Domenico Beneventano, and Sonia Bergamaschi. 2019. SparkER: Scaling Entity Resolution in Spark. In *EDBT*.
[34] Lei Gao, Pengpeng Zhao, Victor S. Sheng, Zhixu Li, An Liu, Jian Wu, and Zhiming Cui. 2015. EPEMS: An Entity Matching System for E-Commerce Products. In *Web Technologies and Applications*, Reynold Cheng, Bin Cui, Zhenjie Zhang, Ruichu Cai, and Jia Xu (Eds.). Springer International Publishing, Cham, 871–874.
[35] Lifang Gu and Rohan Baxter. 2004. Adaptive filtering for efficient record linkage. In *SDM*. 477–481.
[36] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *SIGMOD*.
[37] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. 2009. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)* 34, 4 (2009), 1–39.
[38] Robert Isele, Anja Jentzsch, and Christian Bizer. 2011. Efficient multidimensional blocking for link discovery without losing recall.. In *WebDB*. 1–6.
[39] Delaram Javdani, Hossein Rahmani, Milad Allahgholi, and Fatemeh Karimkhani. 2019. Deepblock: A novel blocking approach for entity resolution using deep learning. In *ICWR*. 41–44.
[40] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data (TBD)* 7, 3 (2021), 535–547.
[41] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR* .
[42] Lars Kolb, Andreas Thor, and Erhard Rahm. 2012. Dedoop: Efficient Deduplication with Hadoop. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1878–1881.
[43] Lars Kolb, Andreas Thor, and Erhard Rahm. 2012. Load balancing for MapReduce-based entity resolution. In *ICDE*.
[44] Pradap Konda, Sanjib Das, AnHai Doan, Adel Ardalan, Jeffrey R Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeff Naughton, Shishir Prasad, et al. 2016. Magellan: toward building entity matching management systems over data science stacks. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1581–1584.
[45] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *SIGMOD*. 489–504.
[46] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale graph computation on just a PC. In *USENIX OSDI*. 31–46.
[47] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. 2019. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 31, 1 (2019), 94–110.
[48] Bo-Han Li, Yi Liu, An-Man Zhang, Wen-Huan Wang, and Shuo Wan. 2020. A survey on blocking technology of entity resolution. *Journal of Computer Science and Technology* 35 (2020), 769–793.
[49] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep Entity Matching with Pre-Trained Language Models. *Proceedings of the VLDB Endowment* 14, 1 (2020), 50–60.
[50] Michael D. Lieberman, Jagan Sankaranarayanan, and Hanan Samet. 2008. A Fast Similarity Join Algorithm Using Graphics Processing Units. In *ICDE*, Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen (Eds.). 1111–1120.
[51] Willi Mann and Nikolaus Augsten. 2014. PEL: Position-Enhanced Length Filter for Set Similarity Joins. In *Grundlagen von Datenbanken*.
[52] Vahab S. Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. 2018. Consistent Hashing with Bounded Loads. In *SODA* . 587–604.
[53] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.).
[54] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *ICML*. 807–814.
[55] Negin Nematollahi, Mohammad Sadrosadati, Hajar Falahati, Marzieh Barkhordar, Mario Paulo Drumond, Hamid Sarbazi-Azad, and Babak Falsafi. 2020. Efficient nearest-neighbor data sharing in GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 1 (2020), 1–26.
[56] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. 2019. H-INDEX: Hash-Indexing for Parallel Triangle Counting on GPUs. In *IEEE High Performance Extreme Computing Conference*. 1–7.
[57] George Papadakis, Ekaterini Ioannou, Claudia Niederée, Themis Palpanas, and Wolfgang Nejdl. 2011. To compare or not to compare: making entity resolution more efficient. In *Proceedings of the international workshop on semantic web information management*. 1–7.
[58] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. 2020. Blocking and filtering techniques for entity resolution: A survey. *ACM Computing Surveys (CSUR)* 53, 2 (2020), 1–42.
[59] Vibhor Rastogi, Nilesh Dalvi, and Minos Garofalakis. 2011. Large-Scale Collective Entity Matching. *Proceedings of the VLDB Endowment* 4, 4 (2011).
[60] Ryan A Rossi and Rong Zhou. 2016. Leveraging multiple gpus and cpus for graphlet counting in large networks. In *CIKM*. 1783–1792.
[61] Giovanni Simonini, Sonia Bergamaschi, and HV Jagadish. 2016. BLAST: a loosely schema-aware meta-blocking approach for entity resolution. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1173–1184.
[62] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *ICDE*. 698–709.
[63] Shaoxu Song and Lei Chen. 2013. Efficient discovery of similarity constraints for matching dependencies. *Data & Knowledge Engineering* 87 (2013), 146–166.
[64] Marshall Harvey Stone. 1937. Applications of the theory of Boolean rings to general topology. *Trans. Amer. Math. Soc.* 41, 3 (1937), 375–481.
[65] Yufei Tao. 2018. Massively Parallel Entity Matching with Linear Classification in Low Dimensional Space. In *ICDT*.
[66] Saravanan Thirumuruganathan, Han Li, Nan Tang, Mourad Ouzzani, Yash Govind, Derek Paulsen, Glenn Fung, and AnHai Doan. 2021. Deep learning for blocking in entity matching: a design space exploration. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2459–2472.

[67] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, and Tie-Yan Liu. 2013. A theoretical analysis of NDCG type ranking measures. In *Conference on learning theory*. PMLR, 25–54.
[68] Li Zeng, Lei Zou, M Tamer Özsu, Lin Hu, and Fan Zhang. 2020. GSI: GPU-friendly subgraph isomorphism. In *ICDE*. 1249–1260.
[69] Wei Zhang, Hao Wei, Bunyamin Sisman, Xin Luna Dong, Christos Faloutsos, and Davd Page. 2020. Autoblock: A hands-off blocking framework for entity matching. In *WSDM*.
[70] Xiaoke Zhu, Yang Liu, Shuhao Liu, and Wenfei Fan. 2023. MiniGraph: Querying Big Graphs with a Single Machine. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2172–2185.

A NOTATION TABLE

We summarize the frequently used notations in Table 6.

Table 6: Summary of frequently used notations

Notations	Definition
R	A relation schema $(\text{eid}, A_1, \ldots, A_n)$
D/P	A set of tuples / a partition of D
m	The number of data partitions of D
$\text{Ca}(P)$	A set of potential matches from P
φ/Δ	A matching dependency (MD) / a set of MDs
$p/X/\mathcal{P}$	A predicate / a precondition / all predicates appeared in Δ
h and $h(t_1, t_2)$	A valuation of $\varphi : X \rightarrow l$ that instantiates variables t and s with tuples t_1 and t_2; φ is a witness at (t_1, t_2) if $h \models X$
$\mathcal{T}/N/\rho$	An execution plan (tree) / a node / a path of $\mathcal{T}$
$e/\text{score}(e)$	An edge (N_1, N_2) of $\mathcal{T}$ / the score of e
$\text{cost}(p, D)$ ($\hat{\text{cost}}(p, D)$)	The (estimated) evaluation cost of p on D
$\text{sp}(p, D)$	The probability of p being satisfied on D
$\text{wp}(p, D)$	The probability of φ being a witness
n_t/n_w	The size of intervals / the size of window

B OTHER OPTIMIZATIONS

HyperBlocker is implemented in 4k+ lines of C++/CUDA code, based on the pipelined architecture of Figure 3. Is has the following implementation-level optimizations.

B.1 Parallel write conflict

When thousands of threads on a GPU are executing in parallel, each thread may produce an uncertain number of results. Therefore, these threads will compete for GPU memory and it is hard for them to decide the position where they should write the results. This is called *parallel write conflict*. Many GPU-based systems, *e.g.,* Pangolin [20], address this by a costly two-round procedure, *i.e.,* they scan the threads twice to (a) record the number of results produced by each thread and (b) write results. Other methods, *e.g.,* GSI [68], estimate the maximum size of results of each thread for space pre-allocation, which, however, leads to low memory utilization.

To alleviate this, we maintain a local buffer for each TB at shared memory, so that only threads in this TB will compete for this buffer. Moreover, the local buffer is divided into two parts. Initially, all threads in a TB use their synchronization primitives to compute write offsets and write to the first part. When the first part is full, this TB will compete with other TBs and flush its buffered results to GPU memory, by computing the global offset with an atomic increment operation; meanwhile, all threads in this TB start to buffer in the second part of the local buffer, and the cycle repeats.

B.2 Data transfer between host and devices

It is widely recognized that data transfer between CPUs and GPUs is one of the most critical bottlenecks in CPUs/GPUs computation [36]. Fortunately, the use of CUDA streams allows to conduct data transfer and GPU execution simultaneously, so as to "cancel" the excessive data shipment cost.

Recall that the input relation D is divided into multiple partitions $P_1, \ldots, P_m$. HyperBlocker extends [70] and iteratively processes these partitions in a pipelined manner. Specifically, the out-of-device processing of each partition P is divided into three steps: (1) Read P into the device memory. (2) Split P into intervals and process each interval. (3) Write the result $\text{Ca}(P)$ back to the host memory. Then HyperBlocker conducts the evaluation on in-device partitions while loading pending ones from and writing results back to the host memory.

C ADDITIONAL EXPERIMENTAL RESULTS.

We experimented with two dataset including $\text{Songs}_{\text{sub}}$, and $\text{TFACC}_{\text{large}}$, Table 7 lists their details. Below we report additional experimental results.

Table 7: Additional datasets

Dataset	Domain	#Tuples	Max #Pairs	#GT Pairs	#Attrs	#Rules
$\text{Songs}_{\text{sub}}$	music	50,000	$\approx 2.5 \times 10^{11}$	#	8	10
$\text{TFACC}_{\text{large}}$	traffic	36M	1.3×10^{15}	#	16	50

Comparison with similarity join. We sampled 50,863 tuples from Songs, where HyperBlocker conducts rule-based blocking by taking these tuples as a single partition, while Faiss converts each tuple to an embedding using FastText [18] and gets its top-K nearest neighbors, varying K from 1 to 100. As shown in Table 8, HyperBlocker is more accurate and 3.65× faster than Faiss on average.

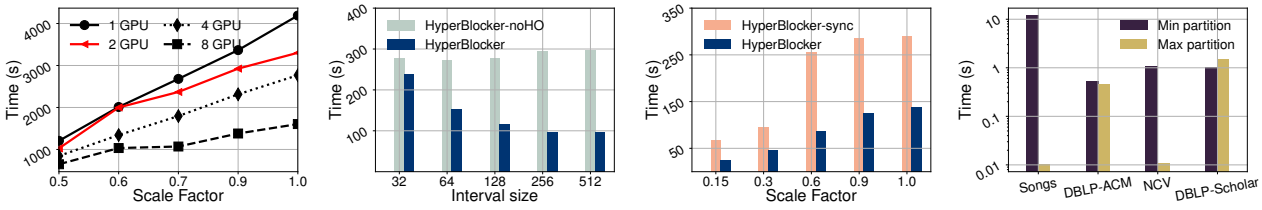
Table 8: Faiss **vs.** HyperBlocker

Method	Metric	Top-1	Top-10	Top-50	Top-100
Faiss	Prec (%)	52.0 (+44.6)	69.0 (-27.6)	73.6 (-23.0)	76.0 (-20.6)
	Time (s)	7.65 (3.34×)	8.58 (3.74×)	8.58 (3.74×)	8.64 (3.77×)
HyperBlocker	Prec (%)	96.6			
	Time (s)	2.29			

Varying $|D|$ and #GPUs. Fixing $m = 1024$, we varied the scale factor of D from 50% to 100% in $\text{TFACC}_{\text{large}}$ and tested HyperBlocker with different numbers of GPUs in Figure 7(b). HyperBlocker scales well with the dataset size. With 8 GPUs, it takes 1604 seconds to process 36M tuples in $\text{TFACC}_{\text{large}}$. Besides, when the number of GPUs increases from 1 to 8, HyperBlocker is 2.6× runs faster. Furthermore, with only one GPU, HyperBlocker can complete a blocking task on table with 36 million tuples one hour, no existing method neither CPU-based and DL-based competitors achieve this performance.

Varying n_t. In Figure 7(e), we tested the impact of interval size n_t on HyperBlocker and $\text{HyperBlocker}_{\text{noHO}}$, that disables all hardware-level optimizations (Section 6, varying n_t from 32 to 512. HyperBlocker achieves the best performance when the interval size is 256, striking a balance between task granularity and parallelism. HyperBlocker runs up to 3.1× faster than $\text{HyperBlocker}_{\text{noHO}}$, verifying the usefulness of hardware optimizations on GPUs (Section 6).

Asynchronous vs. synchronous. The (asynchronous) pipelined architecture plays a pivotal role in HyperBlocker. In Figure 7(f), we compared HyperBlocker with $\text{HyperBlocker}_{\text{sync}}$, a variant without the pipelined architecture (Section 4), *i.e.,* it initiates the next task only after the prior one is fully submitted to devices.. Fixing $m = 512$ with 8 GPUs. we varied the scale factor of TPCH (with 4M tuples) from 15% to 100%. As shown there, HyperBlocker incurs less time than $\text{HyperBlocker}_{\text{sync}}$, *e.g.,* 3.1× faster on average. This is because the communication between devices and hosts in $\text{HyperBlocker}_{\text{sync}}$ is costly. In contrast, by asynchronously overlapping the execution at devices and the data transfer between devices and hosts in HyperBlocker, we reduce both idle time and unnecessary waiting.



(a) TFACC-Large: Varying $|D|$ (b) TFACC: Varying n_t (c) TPCH: Async. vs sync. (d) Partition skew

Figure 8: Additional Experimental Results

Inference cost of $\mathcal{N}$. We now compare the inference cost of $\mathcal{N}$ to the cost of evaluating the actual predicate. Table 9 reports the exact evaluation cost of the most complex predicates and the average exact evaluation cost on DBLP-ACM, DBLP-Songs, Songs, and NCV. As shown, the inference time of $\mathcal{N}$ is significantly smaller than computing the exact evaluation cost, which justifies the necessity of using $\mathcal{N}$.

Table 9: Inference time (s) of $\mathcal{N}$

		DBLP-ACM	DBLP-Scholar	Songs	NCV
Evaluating actual predicate	Max.	1.4	195.1	77.8	195.1
	Avg.	0.8	9.3	26.2	104.3
Inference cost of $\mathcal{N}$	Max.	0.007	0.008	0.009	0.008

Runtime statistics analysis. Table 10 reports runtime statistics on TFACC compared with variants of HyperBlocker: $\text{HyperBlocker}_{\text{noSeq}}$, $\text{HyperBlocker}_{\text{noPSW}}$, $\text{HyperBlocker}_{\text{noStealing}}$.

As shown, HyperBlocker has significantly better warp utility than its variants, achieving an average of 28.21 active threads per warp. This is because: (1) while branch divergence is sometimes unavoidable, a recursive implementation further increases thread stalls, thus harming performance; (2) without PSW, the workload assignment is relatively skewed, resulting in fewer active threads per warp; (3) HyperBlocker supports task stealing from heavily burdened threads, thus reducing thread stalls. The result justify our hardware-aware optimizations.

Table 10: Runtime statistics for $\text{HyperBlocker}_{\text{noSeq}}$, $\text{HyperBlocker}_{\text{noPSW}}$, $\text{HyperBlocker}_{\text{noStealing}}$, and HyperBlocker.

	noSeq	noPSW	noStealing	HyperBlocker
Avg. active threads per warp	14.45	25.59	27.62	28.21
Circles spent in wait stalls	4.25	13.79	4.11	4.07

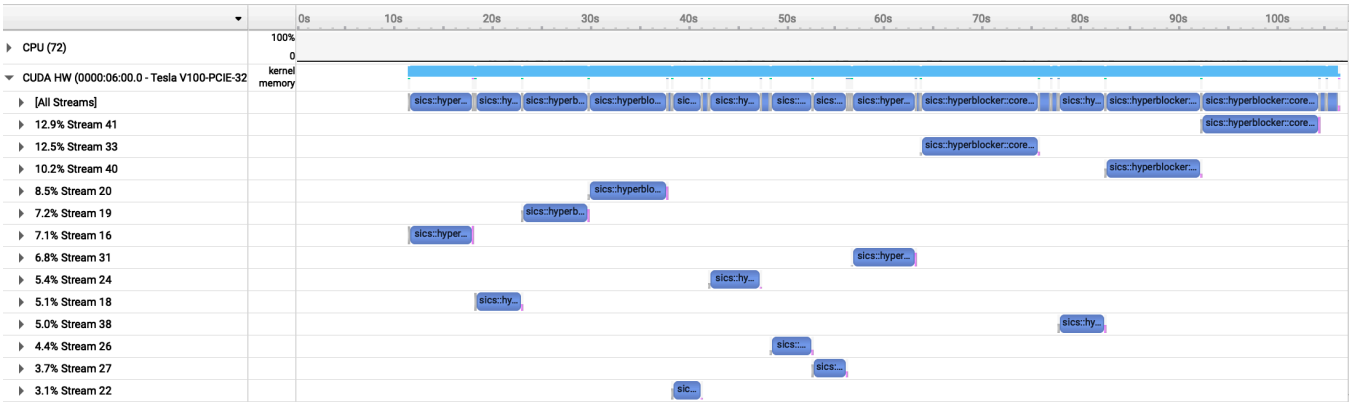
Case study: single execution tree vs. multi-execution tree. We are aware that there is a change in data distribution after partitioning. A unique execution tree for all partitions might not be optimal. To determine if multiple execution trees are needed for optimal performance, we conducted experiments on two different variants: (1) HyperBlocker using one execution plan for all partitions and (2) $\text{HyperBlocker}_{\text{mult}}$ that create optimized execution tree for each partitions. We use DBLP-Scholar, which contains 34,790 tuples with null values in the "year" field. We split DBLP-Scholar into two partitions $P1$, $P2$ where $P2$ groups together tuples that contain null value. HyperBlocker use the φ_{single} for all partitions, while $\text{HyperBlocker}_{\text{mult}}$ use optimized execution plan $\text{HyperBlocker}_{\text{p2}}$ for $P2$.

- φ_{single} : $t.\text{authors} \approx_{\text{JD}} s.\text{authors} \wedge t.\text{title} \approx_{\text{JD}} s.\text{title} \wedge t.\text{year} = s.\text{year} \rightarrow t.\text{eid} = s.\text{eid}$
- φ_{p2} : $t.\text{year} = s.\text{year} \wedge t.\text{authors} \approx_{\text{JD}} s.\text{authors} \wedge t.\text{title} \approx_{\text{JD}} s.\text{title} \rightarrow t.\text{eid} = s.\text{eid}$

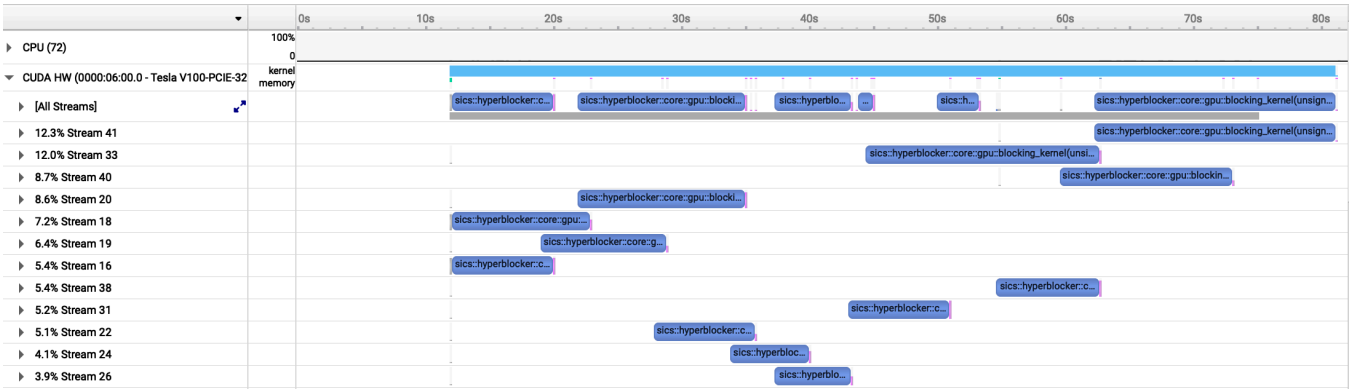
As a result, $\text{HyperBlocker}_{\text{mult}}$ accelerates HyperBlocker by 32.7%. This justifies that using multiple execution trees provides performance benefits over a unique execution tree. However, for other datasets, such as DBLP-ACM, the performance benefit is not remarkable. The reason is that the effectiveness of multiple execution plans heavily depends on the extent of changes in data distribution after partitioning, and in DBLP-ACM, the data distribution does not change significantly.

Skewness on partitions. Figure 8(l) Figure 9 reports the execution times across partitions to understand the impact of partition skew on performance. In Figure 8(d), we specifically measured the maximum and minimum execution times across all partitions to capture the range of performance under skewed data conditions. The partition number is shown in Table 3.

Skewed data can lead to performance bottlenecks. Necessitate strategies for more balanced data partitioning to achieve efficient processing times. Beside the data partition, asynchronous processing is one way to alleviate load imbalance. As shown in Figure 9, asynchronous processing, by handling smaller partitions concurrently, ensures more efficient utilization of resources and improves the overall throughput of the system.



(a) Synchronic



(b) Asynchronous

Figure 9: Songs: elapsed time of each stream.