Efficient GPU-accelerated Join Optimization for Complex Queries

Vasilis Mageirakos[1†], Riccardo Mancini[2], Srinivas Karthik[3#], Bikash Chandra[3], Anastasia Ailamaki[3,4]
[1]University of Patras [2]Scuola Superiore Sant'Anna [3]EPFL [4]RAW Labs SA
[†]vasilis.mageirakos@gmail.com [2]rickyman7@gmail.com [#]skarthikv@gmail.com [3]{*firstname.lastname*}@epfl.ch

***Abstract*—Analytics on modern data analytic and data warehouse systems often need to run large complex queries on increasingly complex database schemas. A lot of progress has been made on executing such complex queries using techniques like scale out query processing, hardware accelerators like GPUs and code generation techniques. However, optimization of such queries remains a challenge. Existing optimal solutions either cannot be effectively parallelized, or are inefficient while doing a lot of unnecessary work.**

In this demonstration, we present our system, GPU-QO, which aims to demonstrate query optimization techniques for large analytical queries using GPUs. We first demonstrate Massively Parallel Dynamic Programming (`MPDP`) – a novel query optimization technique that can run on GPUs to generate optimal plans in a (massively) parallel and efficient manner. We then showcase IDP$_2$-MPDP and UnionDP – two heuristic techniques, again using GPUs, that can even optimize queries containing 1000s of joins. Furthermore, we compare our techniques with current state-of-the-art solutions, and demonstrate how our techniques can reduce optimization time for optimal solutions by nearly two orders of magnitude and produce much better query plans for heuristics (up to 7x).

***Index Terms*—Parallel query optimization, GPU query optimization, Dynamic programming**

I. Introduction

Complex analytical queries are becoming increasingly important in a world where information derived from the data are key drivers for policy decisions, clinical research and business strategies. With the growing pace of data and query complexities, certain query processing modules have evolved to keep pace with it. For instance, modern data warehouses use sophisticated data layouts, scale-out and/or scale-up query processing, hardware accelerators (GPUs, FPGAs), just-in-time (JIT) code generation, and approximate query processing to execute complex queries in real-time. However query optimization may not benefit from these and remains a challenge.

In order to efficiently run such large analytical queries, query optimizers must choose an optimal query execution plan. One of the crucial aspects for choosing an optimal query plan is choosing an optimal join order which is known to be NP-hard. For instance, PostgreSQL takes as much as around 160 secs to find the optimal plan even for a complex 21-join query. Hence, systems fall back to heuristics beyond a certain threshold number of relations (e.g. 12 rels in PostgreSQL).

* This work was partially funded by the EU H2020 project SmartDataLake (825041).

† Work done partially while at EPFL

Existing optimal join order optimization techniques can be categorized under *vertex-based or edge-based enumeration*. The shortcomings of these techniques are: a) vertex-based techniques are inefficient and may generate up to orders of magnitude more *intermediate states* than required (detailed in [1]); b) edge-based enumeration solutions cannot be efficiently parallelized, thus, having limited performance gains from parallelization using multicore CPUs and GPUs [2], [3]. As for heuristics, prior solutions either produce low-quality plans or cannot scale to 1000s of relations. The existence of such large queries and the inability of existing optimizers to handle them is described in [4].

Our objective is to improve the performance of query optimizers for complex queries, i.e. large join queries ($\geq$ 10 rels), using the massive parallelism offered by GPUs. Specifically, we aim to 1) Reduce the query optimization time of optimal solutions; 2) Improve the quality of heuristic techniques (that can scale to 1000s of rels) while being cheap to compute.

Contributions: We propose a new dynamic programming (DP) based join ordering algorithm, `MPDP` (massively parallel DP), that can efficiently use the massive parallelism offered by GPUs. Also, to overcome the limitations of inefficient vertex-based enumeration, we develop a novel plan enumeration technique that evaluates close to the optimal number of intermediate states. MPDP uses a hybrid of vertex and edge based enumeration, each enumerating on carefully chosen relation subsets. `MPDP` on GPU for around 25 relations, is up to 3 orders of magnitude faster than PostgreSQL. It is also up to 80x faster than the state-of-the-art parallel CPU algorithm, and up to 19x than the state-of-the-art parallel GPU algorithm [1] in our setup.

For the heuristic solution, we integrate `MPDP` into an existing heuristic, IDP$_2$ [5], and propose a novel join graph topology conscious heuristic, *UnionDP*. Due to the algorithmic efficiency and high parallelizability of `MPDP`, as well as systematic exploration by IDP$_2$ and UnionDP, we get better quality plans. Our techniques can scale to 1000s of relations while producing up to 7x (UnionDP) better plans compared to the state-of-the-art solution.

In this paper, we demonstrate our system, GPU-QO, which is an *efficient GPU-accelerated query optimizer* built inside PostgreSQL. The system can handle large analytical queries, and our efficient GPU implementation achieves significant performance benefits (as described in Section II-B). *To the best of our knowledge, GPU-QO is the first GPU-accelerated*

query optimizer in a widely used database system.

We present MPDP and its GPU implementation architecture in Section II. Then, in Section III, we discuss our two heuristic solutions. Details of our optimal and heuristic techniques are discussed in [1]. We provide an overview of the interactive demonstration in Section IV along with the user interface and some examples. The demonstration would help users understand the query optimization challenges for large join queries. They will be able to get insights on the impact of join order, along with improvements our techniques provide in comparison with the state-of-the-art solutions.

II. Parallel Query Optimizer

A. Optimal Solutions

Dynamic programming (DP) techniques are one of the standard ways to find the best join order. Systems such as PostgreSQL and IBM DB2 adopt DPSIZE [6] – a *vertex-based enumeration* using DP. DPSIZE explores the search space in increasing sub-relation sizes, while another *vertex-based enumeration* DPSUB [7], enumerates the power-set of relations in subset *precedence* order. Assuming no cross joins – a widely used rule of thumb – these algorithms do not scale well with large joins, since they evaluate a significant number of *invalid* Join-Pairs. A valid Join-Pair is a pair of relation subsets to be joined which does not result in cross joins [1].

In contrast DPCCP [3], an *edge-based enumeration* algorithm, outperforms DPSIZE and DPSUB, by evaluating only valid Join-Pairs. Such edge-based enumeration algorithms are difficult to parallelize. DPE [2] proposes a framework that can parallelize DPCCP. More recently, Meister et al. [8] proposed GPU versions of DPSIZE and DPSUB algorithms.

Existing optimization techniques thus are either efficient (do not evaluate invalid join pairs like DPCCP) or parallelize well (like DPSUB and DPSIZE), *but not both*. The only optimization technique that is both efficient and parallelizable is DPE, but that has limited parallelizability. *In order to minimize the optimization time, our goal is to design an efficient optimization algorithm that is highly parallelizable, while minimizing the evaluation of invalid* Join-Pairs.

Our key contribution is a novel enumeration technique that combines *vertex-based* and *edge-based* enumeration systematically. This enables us to leverage the power of massive parallelism offered by GPUs (hardware benefits), while evaluating only a few invalid Join-Pairs (software benefits). This is achieved by identifying blocks (or biconnected components) in the join graph. We perform a vertex-based enumeration within the blocks to create Join-Pairs at a block-level. Then, we create a Join-Pair for the whole graph, by edge enumeration, using the block Join-Pairs as the seed nodes. Since the expensive vertex-based enumeration is just limited to blocks, it is more efficient and highly parallelizable. When evaluated on MusicBrainz real-world dataset, MPDP is nearly two orders of magnitude faster compared to parallel state-of-the-art techniques for large analytical queries. Thus, in PostgreSQL, the heuristic-fall-back limit increases from 12 to 25 rels with the same time budget.

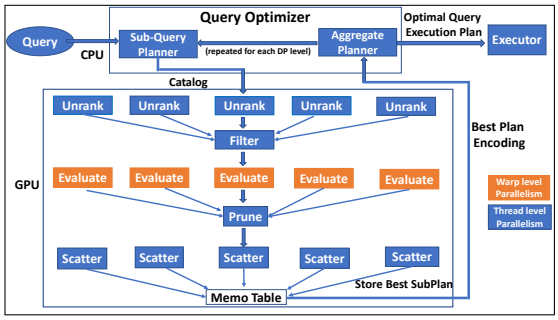

Fig. 1: MPDP Architecture on GPUs

B. MPDP Architecture

Figure 1 shows the architecture of our GPU-accelerated optimizer, MPDP. We build on the design mooted in [8]. The DP based optimization happens at several levels, where at each level i, the best sub-plan for all possible i relations will be found. Leveraging the principle of optimality helps us get the best plan at the final level.

Since the metadata usually resides in CPU main memory, in MPDP, the CPU calls functions in the GPU to find all the best sub-plans for level i. This is repeated for every level in the DP lattice to get the final plan. In the GPU, the optimal sub-plans at each level is found using the following phases:

1) *Unrank*: All possible sets of relations of size i, $\mathcal{S}_i$, are unranked using a combinatorial scheme [8]. These are stored in a temporary contiguous memory allocation for use in subsequent phases.
2) *Filter*: All subsets in $\mathcal{S}_i$, that do not form a valid join are filtered. This results in compacting the temporary array via `thrust::remove` in GPU.
3) *Evaluate*: This phase corresponds to evaluating (i.e. costing) $\mathcal{S}_i$ using warp-level parallelism. The warp first finds all the blocks in the relation subset, and then each thread works on different Join-Pairs. Later, each thread unranks the blocks, checks the validity and computes its cost.
4) *Prune:* The best Join-Pair for each $\mathcal{S}_i$ is chosen in parallel using a classical warp reduction.
5) *Scatter:* All the key-value pairs $(\mathcal{S}_i, bestJoin(\mathcal{S}_i))$ are stored in a memo table for use in subsequent levels. This phase is a parallel store on the GPU hash table.

Finally, the best plan, using a plan encoding, is sent to the aggregate plan builder in the CPU from the GPU memo table. The decoded plan is finally executed by the engine to return the results to the user. We also enhance the above design by (a) *kernel fusion* of prune and scatter phases to reduce global memory writes; (b) *avoiding branch divergence* due to 'If' condition by using *collaborative context collection* [9].

III. Approximation Heuristics

Join Ordering is an NP-Hard problem, and as the number of relations further increase, say beyond 30 rels, we need to resort to heuristics. PostgreSQL uses GE-QO, a genetic optimization algorithm. GOO [10], proposes a greedy heuristic that constructs bushy join trees by ordering joins such that it

minimizes intermediate join cardinalities. While IKKBZ [11] limits the search space to left-deep join trees. More recently, LinDP [4], proposes an adaptive optimization technique as a function of the number of relations in the input query. They use either DPCCP [3], linearized DP or IDP$_2$ [5] with GOO and linearized DP depending on the query size. Linearized DP optimizes the left-deep plan found by IKKBZ, and can scale to 1000s of relations. The main issue with heuristic techniques is that they produce low quality plans. *Our objective is to use GPU parallelism to systematically explore a larger search space, and hence produce better quality plans.*

We propose two heuristics, IDP$_2$-MPDP and UnionDP, both of which can scale with queries containing 1000s of rels. The key idea is to decompose the problem into smaller chunks of size k such that each chunk can be optimized with MPDP (using GPUs), and finally combine these (sub)solutions to get the final plan. The value of k represents the maximum number of relations for which MPDP can optimize within a timeout.

1) IDP$_2$-MPDP: We build on IDP$_2$ [5] for this heuristic solution. The idea in IDP$_2$ is to first generate a valid join plan and then greedily improve the solution until convergence. Specifically, in each iteration, it finds the costliest sub-tree of size k, optimizes it using an optimal *DP* algorithm, and replaces the sub-tree if it generated a cheaper sub-plan. Using MPDP as the *DP* algorithm results in much better plans due to its massive parallelizability and efficient enumeration.

However, the performance of IDP$_2$, on certain join graphs, due to a possibly weak initial plan and its greedy nature, may get stuck on a poor local optima. This results in sub-optimal join order choices (details in [1]).

2) UnionDP: To address the above issue, we propose, UnionDP, wherein the key contribution is to *develop a graph topology - conscious heuristic*. In order to generate tractable subproblems, the idea is to partition the graph based on the following sub-goals which trade-off with each other: (a) Minimize the number of partitions/chunks (with size $\leq k$) as partitions misses join order exploration across them; (b) maximize the sum of weights of cut-edges across partitions (edge weights assigned using PostgreSQL cost model) – this way costlier joins are optimized as late as possible.

UnionDP uses the Union-Find data structure to efficiently keep track of partition information. First, it initializes the join graph with unit partition sizes, then the *union* operation is applied to neighbouring partitions in a *minimum union-size order*. When a tie exists, the edge with minimum weight is used to break it. Once union operation can no longer be applied, these partitions are individually optimized by MPDP. Finally, the above procedure is recursively applied until the entire graph has been optimized. At 200 rels, query plans produced by UnionDP are around 7x cheaper compared to LinDP.

IV. Demonstration

We will showcase our system that demonstrates our techniques discussed in this paper as well as existing state-of-the-art solutions. All algorithms are implemented and run in the PostgreSQL 12 [12] engine with a timeout of 1 min. Our demonstration setup will run on a server with dual Intel Xeon E5-2650Lv3 CPU (12 cores / 24 threads on each socket) and an Nvidia GTX 1080 GPU. For the virtual demonstration, we will use the video conference software used by ICDE, and permit users to access our machine on which GPU-QO run. The access will allow them to interact with GPU-QO.

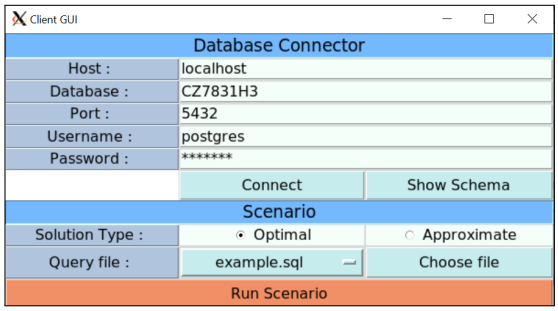

Fig. 2: Input Window

A. Database and Query Selection

The user will first be greeted with an input window. In this window, as shown in Figure 2, is where the user can change the database connection parameters and choose between optimal or heuristic solutions for optimizing selected queries. The schema of the query would be taken from the database connection details. Users can choose to upload a custom schema or modify the existing schema. Since the demonstration targets optimization of large queries, input queries can be selected by selecting an appropriate file that contains the query. Some example query files will be made available for the demo. Also, we would share sample scripts which can generate large database schemas and queries. Users can choose to modify these scripts, queries or upload new queries.

B. Optimal Solution Scenario

If the user selects the optimal solution scenario, our system will use the input query and run different types of query optimization algorithms. We compare the optimization times of MPDP with: a) the default PostgreSQL optimizer; b) DPE-DPCCP (state-of-the-art CPU parallel); and c) COMB-GPU version of DPSUB [8] (state-of-the-art GPU parallel). This part of the demonstration will help users understand and visualize the selected join order and see the improvements in optimization time that our techniques can bring.

Figure 3 shows an example output window for a 20 relation query on the real-world MusicBrainz [13] dataset. In the top half of the window, the user will see a visualization of the optimal join order depicted as a join tree. In the bottom half of the window, a bar plot shows the optimization time (in ms) in the log scale. For the example query with the optimal plan being a bushy tree, MPDP is around 3x faster than DPSUB (GPU), 30x faster than DPE-DPCCP, and more than 3 orders of magnitude faster than PostgreSQL plan.

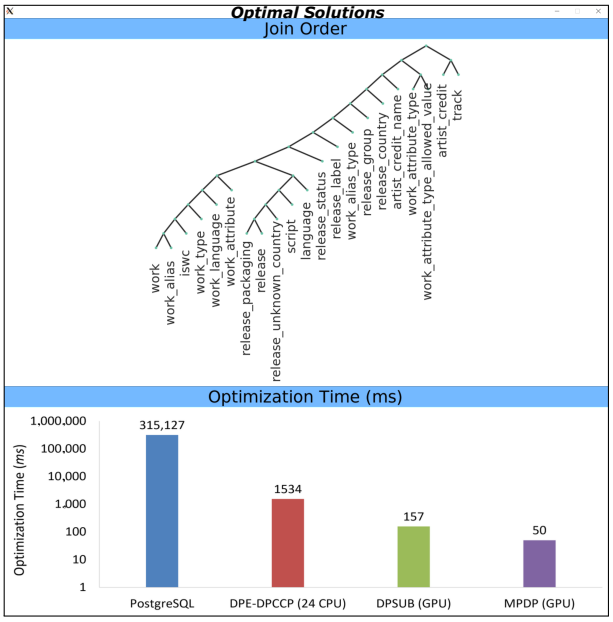

Fig. 3: Optimal Solution Window

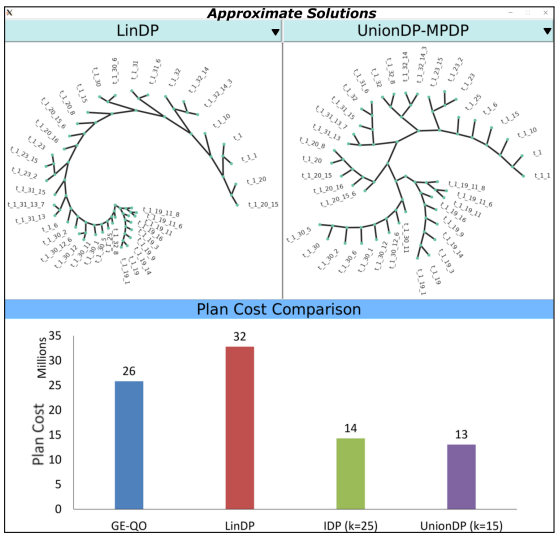

Fig. 4: Heuristic Solution Window

C. Approximation Solution Scenario

This scenario is meant to optimize very large queries that may run into even 1000s of relations. In this part of the demonstration, the user will be able to see the query plans and understand how different heuristic techniques generate different join orders. They will also be able to visualize how the cost of the different join orders (selected by plans) and corresponding optimization time would be different. Our system will compare the plans generated by our two heuristics with: a) GE-QO, the default heuristic used by PostgreSQL, and b) `LinDP` [4] (state-of-the-art heuristic).

An example of the output window shown to the user is captured in Figure 4. For example, a snowflake schema has been used, and the user has submitted a query with 40 relations to find the heuristic plans (even though our techniques can handle much larger join sizes). The output window shows the plan cost, as estimated by PostgreSQL, for each of the heuristic techniques for the input query. Two drop down menus allow the users to select which technique's plan they want to visualize and compare. In our example, the `LinDP` and UnionDP plans are shown.

A circular visualization is used for the selected join plan since there are too many relations for the traditional join tree format. The table names in the example show the depth and the path from the current table to the central fact table in the following way: $t\_level1\_level2\_level3\_level4$. For example, $t\_1\_32\_8$ is a third level table, with t_1_32 as its parent, and the central fact table t_1 as its grandparent. As we can see from the figure, UnionDP has produced a query plan that is 2x cheaper than GE-QO, and about 2.5x cheaper than `LinDP`[1].

V. Conclusion

We develop efficient optimization techniques that leverage the massive parallelism offered by modern hardware (GPU). This can significantly reduce the optimization time of optimal algorithms, and improve the quality of heuristic solutions. As a future work, with increasing optimization parameters in modern times such as cloud analytics, graph analytics and big data systems, one can leverage our GPU-accelerated optimization framework to get efficient solutions – more so, with the advent of declarative machine learning.

References

[1] R. Mancini, S. Karthik, B. Chandra, V. Mageirakos, and A. Ailamaki, "Efficient massively parallel join optimization for large queries," in *SIGMOD*, 2022.

[2] W. Han and J. Lee, "Dependency-aware reordering for parallelizing query optimization in multi-core CPUs," in *SIGMOD*, 2009, pp. 45–58.

[3] G. Moerkotte and T. Neumann, "Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products," in *VLDB*, 2006, pp. 930–941.

[4] T. Neumann and B. Radke, "Adaptive optimization of very large join queries," in *SIGMOD*, 2018, pp. 677–692.

[5] D. Kossmann and K. Stocker, "Iterative dynamic programming: a new class of query optimization algorithms," *ACM TODS*, vol. 25, no. 1, pp. 43–82, 2000.

[6] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *SIGMOD*, 1979, pp. 23–34.

[7] K. P. Bennett, M. C. Ferris, and Y. E. Ioannidis, "A genetic algorithm for database query optimization," in *Proc. of the 4th Intl. Conf. on Genetic Algorithms*, 1991, pp. 400–407.

[8] A. Meister and G. Saake, "GPU-accelerated dynamic programming for join-order optimization," 2020. [Online]. Available: https://www.inf.ovgu.de/inf_media/TechnicalReport+02_2020-p-8268.pdf

[9] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Efficient warp execution in presence of divergence with collaborative context collection," in *MICRO*, 2015, pp. 204–215.

[10] L. Fegaras, "A new heuristic for optimizing large queries," in *DEXA*, 1998, pp. 726–735.

[11] R. Krishnamurthy, H. Boral, and C. Zaniolo, "Optimization of nonrecursive queries," in *VLDB*, 1986, pp. 128–137.

[12] "Postgresql," https://www.postgresql.org/docs/12/index.html, 2021.

[13] *MusicBrainz - The Open Music Encyclopedia*, 2021. [Online]. Available: https://musicbrainz.org/

[1] IDP_2-MPDP and UnionDP have similar performance in this case.