Ocelot/HyPE: Optimized Data Processing on Heterogeneous Hardware

Sebastian Breß[♯⋆1] Max Heimel[†2] Michael Saecker[+3]
Bastian Köcher[†4] Volker Markl[†5] Gunter Saake[⋆6]

[♯] *TU Dortmund University,* [†] *Technische Universität Berlin,* [+] *Parstream GmbH,* [⋆] *University of Magdeburg*
[1]sebastian.bress@tu-dortmund.de [2,5]firstname.lastname@tu-berlin.de [3]michael.saecker@parstream.com
[4]bastian.koecher@campus.tu-berlin.de [6]gunter.saake@ovgu.de

ABSTRACT

The past years saw the emergence of highly heterogeneous server architectures that feature multiple accelerators in addition to the main processor. Efficiently exploiting these systems for data processing is a challenging research problem that comprises many facets, including how to find an optimal operator placement strategy, how to estimate runtime costs across different hardware architectures, and how to manage the code and maintenance blowup caused by having to support multiple architectures.

In prior work, we already discussed solutions to some of these problems: First, we showed that specifying operators in a hardware-oblivious way can prevent code blowup while still maintaining competitive performance when supporting multiple architectures. Second, we presented learning cost functions and several heuristics to efficiently place operators across all available devices.

In this demonstration, we provide further insights into this line of work by presenting our combined system Ocelot/HyPE. Our system integrates a hardware-oblivious data processing engine with a learning query optimizer for placement decisions, resulting in a highly adaptive DBMS that is specifically tailored towards heterogeneous hardware environments.

1. INTRODUCTION

In the last few years, it became apparent that the traditional performance drivers in processor design – frequency and parallelism – are increasingly hitting limitations. This forces hardware manufacturers to rely to a greater extent on designing specialized hardware that is tuned towards certain types of computations [2]. The effect of these developments – which can already be seen today and is expected to significantly increase in the future – is a move towards highly heterogeneous server hardware. Accordingly, in order to keep up with the performance requirements of the modern information society, tomorrow's database systems will need to exploit and embrace this increased heterogeneity.

Proceedings of the VLDB Endowment, Vol. 7, No. 13

Efficiently running data operators on highly specialized hardware such as graphics cards, many-core accelerator cards, modern NUMA architectures, or FPGAs is a highly active research area, and several authors came up with specifically tailored algorithms and data structures for this task [6, 11]. However, integrating those findings into a complete system that can fully exploit highly diverse hardware environments still poses several challenges.

In prior work, we discussed solutions for two of those challenges. First, we demonstrated how *hardware-oblivious* operators can help to manage the inevitable code blowup that is caused by having to develop and maintain multiple device-specific variants of each operator. The general idea behind this approach is to avoid device-specific optimizations and instead implement operators in a generic and abstract form that can be automatically mapped to a specific device at runtime. We demonstrated the feasibility of this approach through our hardware-oblivious engine *Ocelot* [8]. Second, we discussed the so-called *operator placement problem*, which deals with selecting a suitable device for each operation. Solving this problem is challenging, and multiple authors have suggested solutions [6, 10, 13]. We build on our optimizer library *HyPE*, which tackles this problem by utilizing learning cost functions and a set of optimization and scheduling heuristics that take multiple factors – including the estimated operator runtime, the current device load, the current data placement, etc. – into account [4].

In this demonstration, we present our combined system *Ocelot/HyPE*, which integrates these prior projects. This allows our system to a) run efficient data operators on a vast number of different architectures, b) learn device-specific cost functions for these operators, and c) automatically find optimized operator placements across the available devices. In summary, it is a major step towards our goal of building a fully-fledged DBMS that can efficiently exploit highly heterogeneous environments.

We will provide detailed insights into the inner workings of our system, present how to integrate new operators, demonstrate how the system learns cost functions, and discuss the operator placement routines. In order to demonstrate the adaptivity of our system, we will allow users to issue queries on a heterogeneous server containing a multi-core CPU and two graphics cards. The user will also be able to interactively assign operators from a query plan to study the performance behavior of database operators on different processors.

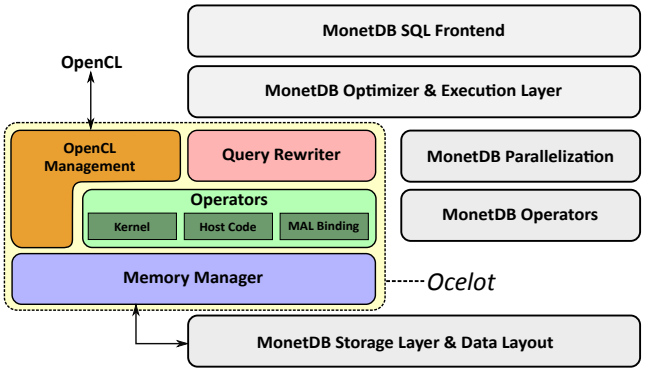

Figure 1: Ocelot's Architecture, taken from [8].

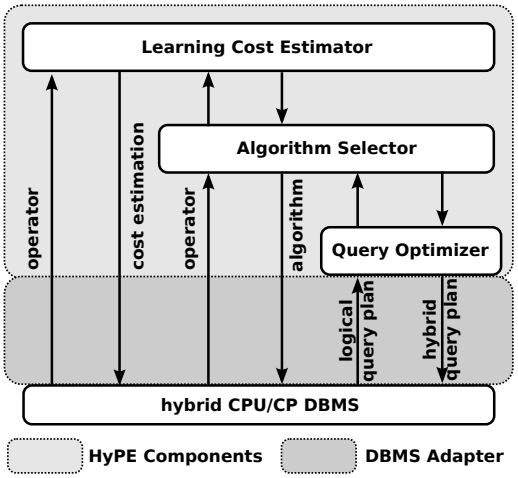

Figure 2: HyPE's Architecture, taken from [3].

2. BACKGROUND

In this section, we provide an overview over our previous systems Ocelot and HyPE, and also discuss the related work for this demonstration.

2.1 Ocelot

Providing efficient data processing in highly heterogeneous environments requires database vendors to implement several device-specific variants of each operator. This is a very challenging and resource-intensive task that effectively limits the number of different architectures a single engine can reasonably support. In prior work [8], we introduced the notion of a hardware-oblivious database as a way to cope with this overhead. The general idea behind this approach is to avoid hand-tuned implementations and instead rely on hardware abstractions and self-tuning to generate device-specific operators at runtime.

In order to demonstrate that a hardware-oblivious approach is feasible and can offer competitive performance, we contributed Ocelot, a prototypical hardware-oblivious database engine integrated into the in-memory column-store MonetDB [1]. Figure 1 illustrates the system architecture of Ocelot: The central part is a set of hardware-oblivious relational operators, which were implemented using the abstract parallel programming library OpenCL[1]: OpenCL is an open programming standard by the Khronos Group that is supported across a wide variety of platforms from all major hardware vendors – including CPUs, iGPUs, dGPUs, accelerators such as Xeon Phis, and even FPGAs. Furthermore, Ocelot includes a memory manager that tracks data placements across devices to avoid unnecessary transfers, and a set of APIs to manage the interaction with the OpenCL runtime.

2.2 HyPE

HyPE is a learning query optimization framework for heterogeneous hardware environments. It is composed of three layers: The cost estimator, the device- and algorithm selector, and the hybrid query optimizer [3]. Figure 2 summarizes this architecture.

The task of the cost estimator is to produce accurate runtime estimates for a given algorithm invocation on a given device. Instead of going the traditional way of using analytical cost models, we leverage query feedback and machine learning techniques for this task. Figure 3 visualizes this approach. The advantage of using query feedback is that we arrive at very flexible cost models that do not require any upfront knowledge about the underlying hardware or algorithms. In particular, our system trains (L2) linear regression models in multi-dimensional feature spaces using a mini-batch online learning approach. These models produce highly accurate estimates with virtually no impact on the query performance. Based on the models from the cost estimator, the device- and algorithm selector chooses a suitable processor and algorithm for a given database operator. This decision can happen according to a list of potential heuristics. In particular, HyPE integrates heuristics that optimize for response time and throughput. For further details on these heuristics, please refer to [5].

Finally, the hybrid query optimizer is build on top of the other two components and uses them to identify a suitable operator placement for a given logical query plan. The optimizer has two different strategies: It can either traverse a set of potential plans and use the cost estimator to select the cheapest one, or it can utilize an adaptive strategy that requests a processing device for each operator at runtime from the processor allocator.

2.3 Related Work

In a prior demonstration [7], we already discussed parts of this ongoing work. In particular, we demonstrated the internals of the learning cost models and how to utilize them to dynamically pick the optimal algorithm variant for a given device. However – in contrast to this work –, we specifically did not cover the multi-device case, the operator placement problem and the system integration aspects.

The major distinction of our work compared to other comparable approaches is our use of dynamic cost models. He et. al. [6] – and more recently Karnagel et. al. [10] and Yuan et. al. [13] – suggested the use of analytical cost models that have to be calibrated for the given architecture by adjusting certain constants. This approach requires extensive calibration before system start-up, which can require a fairly significant amount of time. Furthermore, dynamic models are more flexible than their analytical counterparts, given that they can automatically capture even fairly complex relations between input features and runtime, and that they can easily adapt to changes in the system environment and the user behaviour.

[1] www.khronos.org/opencl/

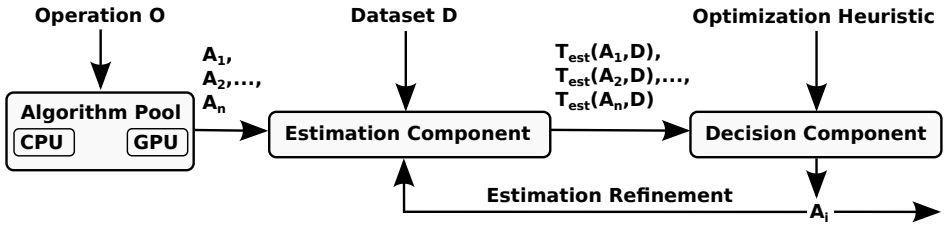

Figure 3: HyPE's underlying decision model.

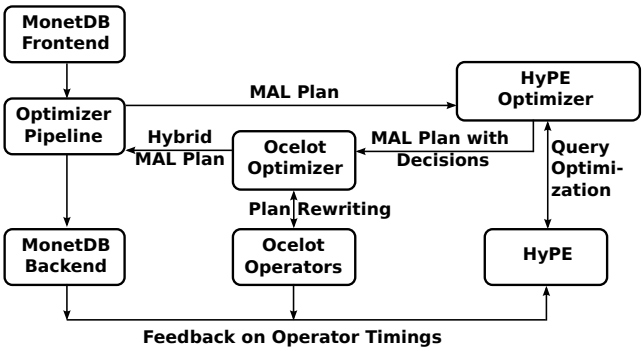

Figure 4: Overview over the Ocelot/HyPE system.

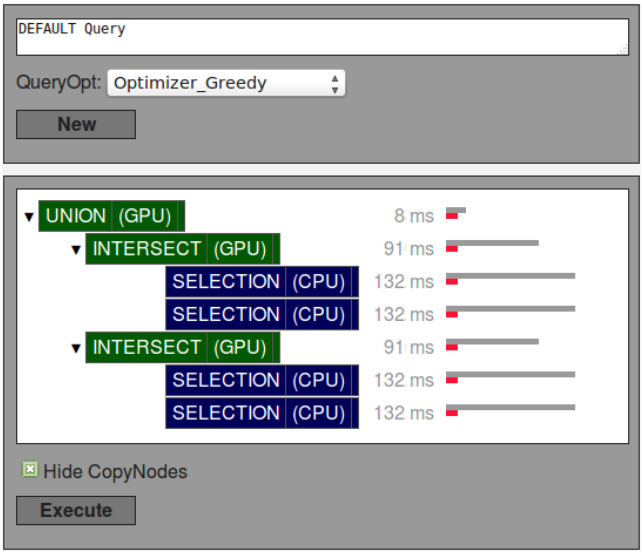

Figure 5: User interface for the demonstration.

3. SYSTEM OVERVIEW

In this section, we provide details on the integration aspects in our combined system.

3.1 Motivation

Due to its flexible engine, Ocelot already offers the possibility to place operators across all available devices. However, this placement has to be specified manually, meaning the user has to annotate a specific device for each operator in the query plan. While this approach might be feasible for recurring queries that are specified ahead of time, it is impractical for ad-hoc scenarios. In these cases, it becomes mandatory to decide the placement automatically and without user interaction. Otherwise, we will not be able to fully exploit the heterogeneity of the machine. Accordingly, integrating a specialized optimizer such as HyPE to automate the placement decisions is the logical next step towards our goal of building a DBMS for heterogeneous systems.

3.2 Integration

HyPE is implemented in a very straightforward modular fashion, which allowed us to integrate it fairly seamlessly. Basically, there were only two required steps: First, we needed to register Ocelot's operators to HyPE, which was easily accomplished using HyPE's APIs. Second, we needed a mechanism to convey the placement decisions from HyPE to Ocelot, and subsequently feed query runtimes from Ocelot back to HyPE. This step was implemented by directly integrating HyPE into the query optimization pipeline of MonetDB.

The query optimizer of MonetDB is structured as a list of sequential optimizer stages, where each stage transforms a plan into a more efficient, but equivalent one. The final optimized plan is then executed by MonetDB's plan interpreter [9]. This modular design allowed us to easily integrate HyPE's decision logic into the query optimizer. In particular, we only had to add a new optimizer stage that runs before Ocelot injects its operations into the MonetDB plan. This newly added optimizer stage works as follows:

1. Before executing the query, we transform the given MonetDB query plan into the internal representation used by HyPE and hand it over.
2. HyPE chooses a physical plan according to a query optimization heuristic which is specified by the user (See Section 2.2 for a list of available heuristics).
3. Then, the HyPE optimizer step retrieves the scheduling decisions from the resulting HyPE plan and assigns them to their corresponding MAL operators.
4. Afterwards, the Ocelot optimizer step replaces a MAL operator with the respective Ocelot operator and sets the processing device decided by HyPE.
5. Finally, MonetDB executes the query plan. After the query has finished, Ocelot retrieves the measured execution times of all operators and sends them as feedback to HyPE. This information is then used to adjust the learned cost models.

In general, HyPE optimizes the plan ahead of query execution. However, depending on the chosen heuristic, HyPE can also rewrite the query plan during execution, and hence, re-optimize the plan at runtime (e.g., in case cardinality estimates exceed an error threshold).

4. DEMONSTRATION SETUP

During the demonstration, we will use remote access to a server, which runs the combined Ocelot/HyPE system. The actual configuration of this server is unclear as of yet, but

will contain at least two graphics cards and a multi-core CPU. Based on this setup, we will demonstrate and explain the following aspects:

Hardware Obliviousness: The demonstration will show how a database can be designed and implemented in a hardware-oblivious manner from the operator level up to the query optimizer. Additionally, we will prepare a poster with further details of the system's architecture and will illustrate the process of query processing in Ocelot.

Automatic Operator Placement: We will provide an interactive user interface – illustrated by Figure 5 –, in which a user can either choose a query from a list of queries from the *Star Schema Benchmark* [12] or issue own ad-hoc queries. The user interface will forward the query to our system and display the resulting query plan, visualizing the placement decisions made by HyPE. The user can then interactively modify these decisions to explore the effects on query response time. Furthermore, the interface also allows the user to choose between the different query optimization heuristics (c.f. Section 2.2) and compare their impact on the placement decisions.

Performance of Query Processing. In the demonstration, we will also display the run-time of each database operator in the executed query plan to provide the user with detailed performance information. This allows the user to gradually adjust the query plan, finding iteratively the optimal processor for the most performance critical operations. Finally, we will show the performance differences of the combined Ocelot/HyPE system compared to Ocelot and vanilla MonetDB running on a single device.

5. SUMMARY

It is still an open problem to optimize database queries for heterogeneous CPU/co-processor systems. The two main issues are (1) that the analytical cost models used so far do not scale for an increasing number of heterogeneous (co-)processors and (2) that today's query optimization strategies do not consider properties of co-processors sufficiently (e.g., limited memory or (un-)suitability for certain operations).

In this demonstration, we show an alternative database design, which learns the cost models during query processing and performs query optimization in consideration of data locality and the load condition across (co-)processors. The combined Ocelot/HyPE system demonstrates that it is possible to perform cost-based query optimization without detailed knowledge of the hardware by learning the performance behavior of database operators. By using execution times as cost measure, we can also keep track of the computational load across all (co-)processors, which enables us to steer the parallelism between CPUs and co-processors.

In future work, we plan to evaluate our system on other co-processors (e.g., APUs or Xeon Phis) and compare them with other co-processor-accelerated DBMSs, such as OmniDB [14] and CoGaDB [3].

Acknowledgements

This work was partly funded by the German Federal Ministry of Education and Research under funding code 01IS12056. Furthermore, we thank Jens Teubner from TU Dortmund University for the fruitful feedback and discussions.

6. REFERENCES

[1] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[2] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[3] S. Breß. Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS. *The VLDB PhD workshop, PVLDB*, 6(12):1398–1403, 2013.

[4] S. Breß, F. Beier, H. Rauhe, K.-U. Sattler, E. Schallehn, and G. Saake. Efficient co-processor utilization in database query processing. *Information Systems*, 38(8):1084–1096, 2013.

[5] S. Breß, N. Siegmund, L. Bellatreche, and G. Saake. An operator-stream-based scheduling engine for effective GPU coprocessing. In *ADBIS*, pages 288–301. Springer, 2013.

[6] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query co-processing on graphics processors. In *ACM Trans. Database Syst.*, volume 34. ACM, 2009.

[7] M. Heimel, F. Haase, M. Meinke, S. Breß, M. Saecker, and V. Markl. Demonstrating self-learning algorithm adaptivity in a hardware-oblivious database engine. In *EDBT*, pages 616–619, 2014.

[8] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9):709–720, 2013.

[9] S. Idreos et al. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[10] T. Karnagel, M. Heimel, M. Hille, M. Ludwig, D. Habich, W. Lehner, and V. Markl. Demonstrating efficient query processing in heterogeneous environments. In *SIGMOD*. ACM, 2014. to appear.

[11] R. Mueller, J. Teubner, and G. Alonso. Data processing on FPGAs. *PVLDB*, 2(1):910–921, 2009.

[12] T. Rabl, M. Poess, H.-A. Jacobsen, P. O'Neil, and E. O'Neil. Variations of the star schema benchmark to test the effects of data skew on query performance. In *ICPE*, pages 361–372. ACM, 2013.

[13] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on GPU devices. *PVLDB*, 6(10):817–828, 2013.

[14] S. Zhang, J. He, B. He, and M. Lu. OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures. *PVLDB*, 6(12):1374–1377.