Hardware-Oblivious Parallelism
for In-Memory Column-Stores

Max Heimel
Technische Universität Berlin
max.heimel@tu-berlin.de

Michael Saecker
ParStream GmbH
michael.saecker@
parstream.com

Holger Pirk
CWI Amsterdam
holger.pirk@cwi.nl

Stefan Manegold
CWI Amsterdam
stefan.manegold@cwi.nl

Volker Markl
Technische Universität Berlin
volker.markl@tu-
berlin.de

ABSTRACT

The multi-core architectures of today's computer systems make parallelism a necessity for performance critical applications. Writing such applications in a generic, hardware-oblivious manner is a challenging problem: Current database systems thus rely on labor-intensive and error-prone manual tuning to exploit the full potential of modern parallel hardware architectures like multi-core CPUs and graphics cards. We propose an alternative design for a parallel database engine, based on a single set of hardware-oblivious operators, which are compiled down to the actual hardware at runtime. This design reduces the development overhead for parallel database engines, while achieving competitive performance to hand-tuned systems.

We provide a proof-of-concept for this design by integrating operators written using the parallel programming framework OpenCL into the open-source database MonetDB. Following this approach, we achieve efficient, yet highly portable parallel code without the need for optimization by hand. We evaluated our implementation against MonetDB using TPC-H derived queries and observed a performance that rivals that of MonetDB's query execution on the CPU and surpasses it on the GPU. In addition, we show that the same set of operators runs nearly unchanged on a GPU, demonstrating the feasibility of our approach.

1. INTRODUCTION

The modern hardware landscape is getting increasingly diverse. Today, a single machine can contain several different parallel processors like multi-core CPUs or GPUs. This diversity is expected to grow further in the coming years, with micro-architectures themselves diverging towards highly parallel and heterogeneous designs [8]. We believe that making database engines ready to exploit the capabilities of this diverse landscape of parallel processing platforms will be one of the major challenges for the coming decade in database research.

Unfortunately, implementing parallel data operators is a tedious and error-prone task that usually requires extensive manual tuning. Most systems are therefore designed with a certain hardware architecture in mind: they are *hardware-conscious*. Extending those systems to new architectures usually requires the developer to implement an additional set of hardware-specific operators, adding significant development and maintenance overhead in the process.

Instead of maintaining multiple sets of operators, we believe that a parallel database engine can be designed in a *hardware-oblivious* manner, i.e., without any inherent reliance on a specific architecture: All knowledge is encapsulated into a library that adheres to a standardized interface and is provided by the manufacturer of the respective hardware components. The system is designed around a single set of operators, which can be mapped to a variety of parallel processing architectures at runtime. We also argue that existing systems can be extended to become hardware-oblivious. To support these claims, we make the following contributions:

1. We present Ocelot[1], a hardware-oblivious extension of the open-source column-store MonetDB. Ocelot uses a standardized interface provided by OpenCL to map operations to any supported parallel processing architecture.

2. We demonstrate that a single hardware-oblivious implementation of the internal MonetDB operators can efficiently run on such dissimilar devices like CPUs and GPUs.

3. We evaluate our approach against the hand-tuned query processor of MonetDB and show that Ocelot can compete with MonetDB's performance when running on a CPU, and outperform it when using the graphics card.

The paper is structured as follows: In the next section, we motivate and discuss the concept of hardware-oblivious database designs. We also give an introduction to the kernel programming model, and motivate why we chose this model for our prototype. In Section 3, we give an overview of the design of Ocelot, with further implementation details being discussed in Section 4. Section 5 presents our evaluation of Ocelot and discusses the results, Section 6 presents related work. In Section 7, we discuss possible directions for future research. Finally, the paper is concluded by Section 8, which summarizes our findings.

---

[1]The Ocelot source code is available at: `goo.gl/GHeUv`.
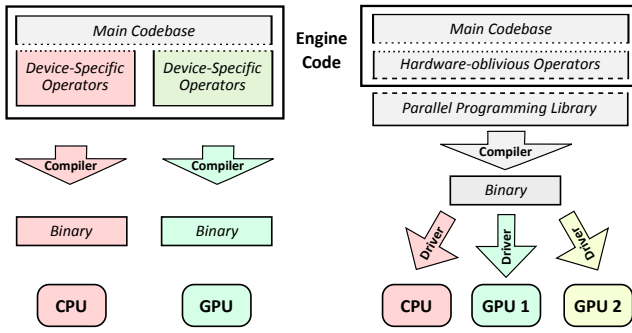
Figure 1: Hardware-Conscious and Hardware-Oblivious database designs: The left side illustrates a hardware-conscious database engine that uses distinct sets of operators for each supported architecture. The right side shows a hardware-oblivious design, which relies on a parallel programming library to provide abstract operator implementations that are compiled down at runtime to the actual hardware by a device driver.

2. MOTIVATION

In this section, we motivate the idea of building a highly portable, hardware-oblivious database engine to cope with the increasingly diverse hardware landscape. We also provide a short introduction to the kernel programming model, and motivate why we chose it to implement hardware-oblivious operators for our prototype.

2.1 The Need for Hardware Abstraction

The increasingly diverse hardware landscape warrants the question how to design a database engine that can efficiently use all available compute resources. So far, this question has mostly been discussed in the context of data processing on specialized hardware, like graphics cards and FPGAs. The usual approach relies on using a separate set of hand-tuned *hardware-conscious* database operators for each supported device type [19, 23, 28]. The left side of Figure 1 illustrates this approach.

Using hand-tuned database operators will usually result in the best performance for any given architecture. However, from a development perspective, several problems arise when the number of supported architectures grows:

- A significant increase in code volume and complexity is expected, incurring high development and maintenance costs. Since development resources are limited, this forces the vendor to focus on a few selected architectures.

- Adding support for "the next big thing" is quite expensive, since it requires implementing and fine-tuning a completely new set of operators. This means that the database vendor is always chasing after the latest hardware developments. Furthermore, this high entry burden hinders the adoption of specialized hardware for data processing.

- The database vendor has to build up expertise outside of its core competences: Each supported architecture typically requires at lest one specialized developer that focuses solely on it. Especially for modern architectures, this can be a risky investment that ultimately might not pay off. Furthermore, acquiring developers specialized in programming for a specific hardware architecture is a challenging task.

Even when we discard the notion of supporting specialized hardware, the outlined problems will still become imminent in the near future: Experts expect a shift towards heterogeneous and highly parallel micro-architectures that feature several different parallel processing elements on a single chip, each with different instruction sets and performance characteristics [8]. With these developments in mind, we believe that building a portable engine will become equally important as achieving optimal performance.

2.2 A Hardware-Oblivious Database Design

Instead of maintaining multiple code-paths for several architectures, our suggested design is built around a single set of hardware-oblivious parallel database operators. The operators are implemented in a highly abstract fashion against a parallel programming library, without any inherent reliance on a particular hardware architecture. At runtime, a vendor-supplied compiler – or driver – maps this representation to the actual hardware, performing all device-dependent optimizations. The right side of Figure 1 illustrates this design.

Because of the involved abstraction, we expect that a hardware-oblivious database would likely not perform as well as hand-tuned operators. However, it provides a much more portable and maintainable engine. Besides the obvious advantage of reducing code volume and complexity, it would also lead to a separation of concerns during development: The expertise of finding the most efficient execution strategy lies solely with the specialists at the hardware vendor, while the database vendor can focus all of its development resources on implementing the respective processing and storage model. A hardware-oblivious engine would also make it easy to add support for novel architectures - as long as the specific hardware vendor provides a suitable implementation of the parallel programming interface. This would accelerate both the adoption and the acceptance of using specialized hardware like graphics cards for data processing.

At the same time, database users would profit from the freedom to choose among a broader range of devices, allowing them to exploit the processing power of either existing or newly acquired systems to their full extent. Additionally, the support for many architectures lessens the burden of upgrading the hardware of existing machines, which can be a tedious task when migrating systems.

2.3 The Kernel Programming Model

Originating from stream programming, the *kernel programming model* became the de-facto standard for a variety of different GPU programming frameworks like OpenCL [35], CUDA [29], and DirectCompute. In this model, programs are expressed through *kernels*, which describe the operation on a single element of the data. Within the terminology[2] of the model, kernels are running on the *device*, and are scheduled and controlled by the *host*[3]. Kernels are scheduled across the complete input in a lock-free, data-parallel fashion – they can thus be seen as the body of a loop over the input data. Listing 1 shows a simple OpenCL kernel. The call to global_id in line four returns a unique identifier for each invocation of the kernel, controlling the input - and output - elements it operates on. This call can be compared to accessing the loop counter variable within the body of a loop.

Conceptually, the kernel programming model assumes a shared-memory architecture: Each kernel invocation can access any *global memory* address without restrictions. Global memory is not assumed to be addressable by the host, requiring memory transfer

---

[2]While the discussed concepts are valid for all implementations, we will primarily use terminology from OpenCL.

[3]Note that host and device can refer to the same physical device, e.g., when running on a multi-core CPU.

Listing 1: A simple OpenCL kernel.

```
1    __global T* res,
2    __global const T* inp, T cnst) {
3    res[global_id()] = inp[global_id()] + cnst;
4  }
```

– or mapping – between host & device. Offering further abstraction, kernel invocations are partitioned into work-groups, with all threads within a work-group sharing access to a distinct amount of *local memory*. Finally, each thread invocation has some *private memory*, which is used to store local variables.

Due to the highly abstract specification, programs written in the kernel programming model can be matched to a wide variety of parallel and sequential hardware architectures: On a single-core CPU, the kernel can be invoked sequentially within a loop, potentially introducing SIMD instructions to merge neighboring invocations. On a multi-core CPU, a thread can be scheduled for each invocation, mapping the threads of a single work-group onto the same core. If the architecture supports it, local memory can be mapped to directly control the L2 cache. On a GPU, work-groups are mapped to multi-processors, which each can run a few hundred invocations in parallel and have access to a small amount of fast on-chip memory. There has also been work on mapping programs written in the kernel programming model to FPGAs [34].

Summarizing, the kernel programming model is abstract enough to build highly portable code across a wide variety of architectures. At the same time, it is expressive enough to implement all required operations: All major database operators – including aggregation [24], selection [19], sorting [16, 22, 30, 31], joins [20], hashing [2, 3, 14] and string operations [10] –, have been shown to be efficiently implementable within the constraints of the model. We therefore believe that the kernel programming model is a good choice to form the basis of a hardware-oblivious parallel database engine.

3. OCELOT: A HARDWARE-OBLIVIOUS DATABASE ENGINE

In this section, we present an overview of Ocelot, our prototypical hardware-oblivious parallel database engine. Ocelot is integrated into the in-memory column-store MonetDB [7] and uses OpenCL [35] to offer operators that are agnostic of the underlying hardware. To the best of our knowledge, Ocelot is the first attempt at designing a hardware-oblivious database engine for modern parallel hardware architectures.

3.1 System Overview

The primary design goal of Ocelot is to demonstrate the feasibility of a hardware-oblivious database design based on the kernel programming model. In order to quickly arrive at a a working prototype, we limited our scope to supporting the main relational operators – selection, projection, join, aggregation, and grouping – on four-byte integer and floating point data types. On the technical side, we chose to implement our operators using OpenCL, since it is supported across a wide variety of platforms and by all major hardware vendors.

From an architectural perspective, we implemented Ocelot as a light-weight extension for MonetDB. This allowed us to reuse several major components, including data layout, storage management and query execution engine. We also made sure to model our operators as drop-in replacements for MonetDB's query operators,
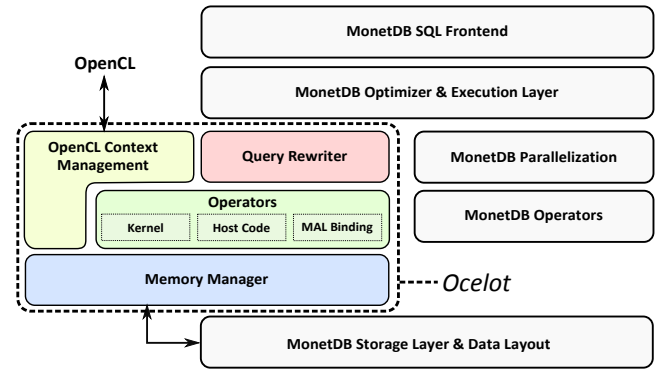

Figure 2: The architecture of Ocelot.

allowing us to recycle the plans generated by MonetDB's query optimizer. Due to the shared architecture, both systems can complement each other, with MonetDB running operations that Ocelot does not support.

Figure 2 shows the architecture of Ocelot and highlights its four major components: The *Operators* are the central part and the workhorse of Ocelot. Each operator implements a drop-in replacement of a particular MonetDB operator using the kernel programming model. The *Memory Manager* is used to abstract away details of the memory architectures from the operators by transparently handling device memory management. The *Ocelot Query Rewriter* adjusts MonetDB query plans for Ocelot by rerouting operator calls to the corresponding Ocelot implementations. The *OpenCL Context Management* initializes the OpenCL runtime, triggers kernel compilation, manages kernel scheduling, and offers access to important OpenCL data structures.

3.2 Operators

Ocelot's operators are advertised to MonetDB via a *MonetDB Assembly Language (MAL) binding*, describing the interface and entry function. The entry function – also called the *operator host-code* – checks input parameters, sets up in- and output resources using the Memory Manager, initializes the required kernels, and schedules them for execution using the Context Management. It also handles error cases, ensuring that all held resources are released upon encountering an unrecoverable error. It should be noted, that host-code is written completely device-independent. All device-dependent decisions are abstracted away by the kernel programming model, the OpenCL framework, the Memory Manager and the Context Management. Further details about the operators, including which implementations were chosen, can be found in Section 4.

3.3 Memory Manager

The Memory Manager acts as a storage interface between Ocelot and MonetDB, hiding details of the device memory architecture from the operator host-code. MonetDB operates on so-called *Binary Association Tables* (BATs), which reside in host memory. The OpenCL kernels, however, can only operate on cl_mem buffers, which reside on the device. Consequently, we have to transform each BAT into an OpenCL buffer object before operating on it. This transformation is handled by the Memory Manager.

Internally, the Memory Manager keeps a registry of OpenCL buffers for BATs. When a BAT is requested, the corresponding buffer object is returned from this registry. If there is no corresponding entry, a new buffer is allocated and registered. When

Ocelot is running on a device that operates in host memory – e.g., on the CPU –, this is a zero-copy operation. Things get more complicated for devices like graphics cards that operate on discrete storage. These devices require a data transfer between host and device to copy the BAT content. In order to avoid these expensive transfers, the Memory Manager acts as a device cache, keeping copies on the device as long as possible.

All resource requests from operators are piped through the Memory Manager. If a request cannot be fulfilled due to insufficient device storage, resources are automatically freed up. This happens through evicting cached BATs in LRU order. Once all cached BATs are evicted, the Memory Manager resorts to offloading result and intermediate buffers to the host[4]. This also happens in LRU order, giving preference to auxiliary data structures like hash-tables before offloading result buffers. The Memory Manager uses reference counting to prevent evicting buffers that are currently in use. By manually increasing the reference count of a BAT, this mechanism can be used to pin frequently accessed BATs permanently to the device.

The Memory Manager also plays an important role in transferring data between operators. Since we need to stay compatible with MonetDB's calling interface, we cannot directly pass cl_mem objects. Instead, our operators return a newly created BAT, and use the Memory Manager to link it with the generated result buffer.

3.4 Query Execution Model

Ocelot follows the operator-at-a-time model of MonetDB. Conceptually, each operator consumes its complete input and materializes its output before the next operator is started. However, contrary to MonetDB, we employ a lazy evaluation model, as our operators only schedule kernel invocations and data transfers, they do not wait for them to finish.

The execution model of Ocelot is built upon OpenCL's event model. In OpenCl, events are associated to specific device operations, like a kernel invocation or a memory transfer. When scheduling a new operation, the user can pass a wait-list of events, which have to finish execution before the operation will start. In Ocelot, we use this mechanism to pass scheduling information to the device driver, allowing it to potentially reorder operations to improve performance. Internally, we maintain a registry of events for each buffer, keeping producer events – tied to operations writing the buffer – and consumer events[5] – tied to operations reading it. When kernels – or data transfers – are scheduled, we pass the producer events of all consumed buffers to OpenCL, ensuring that the operation will only execute once all of its inputs are ready. Afterwards, we register the new operation's event both as a producer for its result and as a consumer for its input buffers.

Due to the different scheduling models, we had to define strict data ownership rules for interactions between Ocelot and MonetDB. Every BAT that is generated by an Ocelot operator is owned by Ocelot, if MonetDB operates on a BAT that is owned by Ocelot, results are undefined. We added an explicit synchronization operator that hands ownership of a BAT back to MonetDB. Internally, this operator waits on the producer events of the buffer associated with the BAT and – depending on the architecture – transfers or maps the buffer back to the host. Our query rewriter automatically inserts this operation when required, e.g., before returning the result set or before calling MonetDB operators.

---

[4]We cannot simply drop these buffers, as they contain computed content. Instead, we offload them to the host and copy them back when needed.

[5]Maintaining consumer events is important to decide whether it is safe to discard a buffer, e.g., when freeing up device storage.

Figure 3 illustrates an exemplary query execution schedule in Ocelot. The lower half shows the sequence of operators on the host, while the upper half shows the scheduled kernels and allocated memory buffers on the device. Note that BATs $t1$, $t2$, and $t3$ are never owned by MonetDB – they are solely used to pass references to device buffers between operators. A sync operation occurs after the $\bowtie$-kernel, triggering the result transfer of BAT r to the host. The figure also illustrates two cases where an OpenCL device driver might reorder operations: First, data transfer t_b is independent of kernels σ_2, σ_3, and $\vee$ - it could be moved forward, hiding transfer latency by interleaving it with any of those kernels. Second, kernels σ_2 and σ_3 are independent of each other. Depending on device load, the driver might decide to interleave those kernels to achieve higher throughput.

4. IMPLEMENTATION DETAILS

In this section, we take a look under the hood of Ocelot. In particular, we will discuss which implementations were chosen for our operators, give an example demonstrating the limitations of our approach, and list the changes we made to MonetDB for our integration.

4.1 Operator Details

We based most of the implementations of our operators on existing work from the area of GPU-accelerated databases. In particular, several operators are based on work of Bingshen He et al. [18, 19, 20].

4.1.1 Selection

Our selection implementation follows the approach outlined in [37]: We encode the selection result as a bitmap, with each thread evaluating the predicate on a small chunk of the input. We found that evaluating the predicate on eight four-byte values – generating one byte of the result bitmap per thread – gave the best results across architectures. Using bitmaps as intermediate results of the selection operator allows us to efficiently evaluate even complex predicates by combining multiple bitmaps using bit operations. Note that, to ensure compatibility with MonetDB's selection operator, bitmaps are never exposed in the interface and are only passed via Memory Manager references. The system transparently materializes bitmaps into lists of qualifying tuple IDs if MonetDB operators access them.

4.1.2 Projection

Conceptually, the projection operation in a column-store is a join between a list of tuple IDs and a column. Practically, since the tuple IDs directly identify the join partner, it can be implemented by directly fetching the projected values from the column. We use a parallel gather primitive to implement this operation efficiently [18]. If the left input is a bitmap – e.g., when projecting on a selection result –, we first have to transform it into a list of tuple IDs by materializing the list of set bits. This materialization requires two steps: First, we compute a prefix sum [33] over bit counts to get unique write offsets for each thread. Then, each thread writes the positions of set bits within its assigned bitmap chunk to its corresponding offset.

4.1.3 Sorting

We use a binary radix sort implementation following the ideas of Satish et al. [31, 32]. In a first step, we generate local histograms of the current radix for each work-group. Afterwards, we shuffle the histograms to ensure that all buckets for the same radix are laid out consecutively in memory, using a prefix sum to calculate the offset
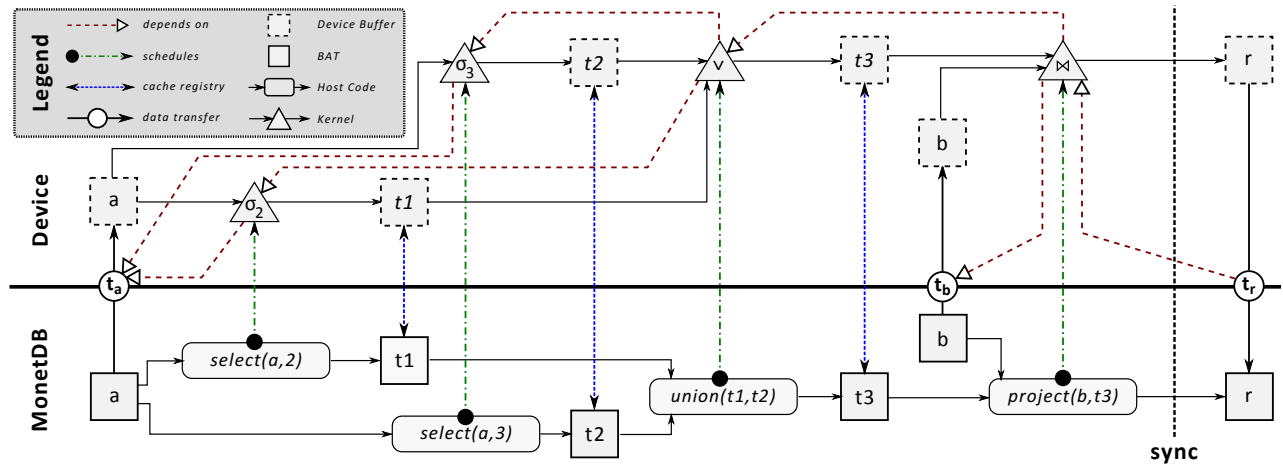
Figure 3: Illustration of the execution schedule for the query: "SELECT b FROM ... WHERE a IN (2,3)".

for each value and all work-groups. Finally, we reorder the values according to the offsets in the global histogram. We repeat this procedure until the complete key was processed. Our actual implementation is based on the work of Helluy [22] with minor modifications to handle arbitrary input sizes and negative values. We currently do not support sorting over multiple columns. Due to the nature of the radix sort, sorting by multiple columns increases the size of the keys, requiring multiple passes over the data. Therefore, for multi-column sorting we would require a different comparator-based implementation to stay competitive.

4.1.4 Hashing

Our parallel hashing algorithm builds on ideas from [2, 3, 14]. It begins with an optimistic round, letting each thread insert its keys without any form of synchronization. If a collision occurs, this will result in keys being overwritten. We test for this case in a second round, where each thread checks whether its keys ended up in the hash table. If the test failed for at least one key, we start a pessimistic round, that uses re-hashing and atomic compare-and-swap operations to re-insert failed keys. We found that in practice, a probing strategy that re-hashes with six strong hash functions before reverting to linear probing gave us a good balance of achieved load factors and hashing cost.

In contrast to prior work, we do not use a stash for failed elements, as we did not observe any noteworthy improvements from using one. Instead, if the pessimistic approach fails for at least one key, we restart with an increased table size. Since restarting is expensive, we try to avoid it by picking an adequate initial table size. In particular, we observed that our hash tables have a filling rate of around 75% and consequently over-allocate the hash table by a factor of 1.4.

Based on this general hashing scheme, we built a multi-stage hash lookup table for joins and grouping operation as described in [19].

4.1.5 Join

We use parallel implementations of two join algorithms, based on work from [20]: A nested loop join is used for theta-joins, equi-joins are handled using a hash join[6]. Both nested-loop and hash join use a two-step approach to avoid thread synchronization: In a

first step, each thread counts the number of result tuples it will generate. From these counts, unique write offsets into a result buffer are computed for each thread using a prefix sum. In the second stage, the join is actually performed, with each thread storing its result tuples at its respective offset. Opposed to [20], we only use this two-step procedure if the number of join partners is unknown. In several common cases – for instance when joining against a key column –, the number of results – or at least a tight upper bound of it – is known beforehand. In those cases, we execute the join directly, omitting the additional overhead.

4.1.6 Group-By

The grouping operator in MonetDB produces a column that assigns a dense group ID to each tuple. Ocelot uses two different implementations for this operation. If the input is sorted, we identify group boundaries by having each thread compare its value with its successor. Then, a prefix sum operation is used to generate dense group IDs. If the input is unsorted, we use a hash table to generate dense group IDs. Afterwards, we build the assignment table via hash look-ups. Multi-column grouping is implemented by recursively calling the group-by operation on the combined IDs from two group assignment columns.

4.1.7 Aggregation

Ungrouped aggregation is implemented using a parallel binary reduction strategy [18]. Grouped aggregation uses a hierarchical scheme, scheduling multiple work-groups on disjunct data partitions to build intermediate aggregation tables using atomic operations[7] in local memory. Afterwards, a single thread is scheduled per group to compute the final aggregate. As discussed in Section 4.2, we found that scheduling one work-group per processor gave us the best performance across architectures. When aggregating values for just a small number of groups, we observed significant performance losses due to synchronization overhead. This overhead is introduced by the atomic operations frequently accessing the same memory address simultaneously. In order to reduce this number of concurrent accesses, we distribute the aggregation evenly across multiple memory addresses within each work-group: The values for each group are aggregated across multiple accumulators, with

---

[6]A special case are PK-FK joins, which are precomputed by MonetDB. These joins only require a projection against the join index.

[7]Since the current version of the OpenCL standard does not provide atomic operations on floating point data, we had to emulate those through atomic compare-and-swap operations on integer values.
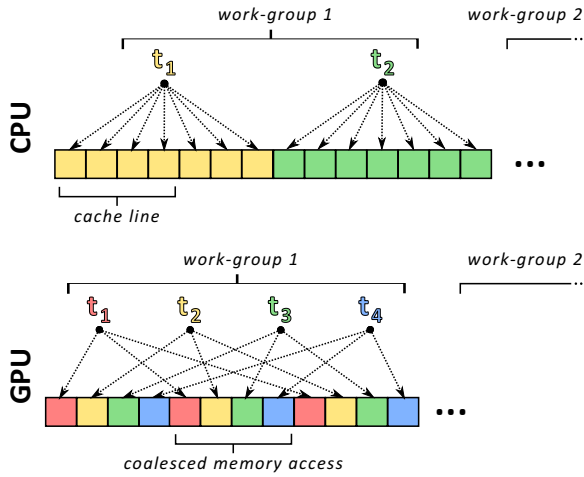
Figure 4: Architecture-dependent scheduling of threads and work-groups.

the number of accumulators per group being chosen inversely proportional to the number of groups. If the aggregation table does not fit into local memory, we fall back to using the same scheme in global memory.

4.2 Kernel Scheduling Strategy

The majority of architecture-dependent code in Ocelot is used to abstract away hardware properties from the operators. One example for this is the Memory Manager, another one occurs when scheduling a kernel. The default OpenCL scheduling parameters – e.g., the size of the work-groups – are often suboptimal, and usually require manual tuning to the architecture. Through trial-and-error, we found a device-dependent scheduling scheme that seems to give robust performance across architectures. For a compute device with n_c cores, where each core features n_a compute units, we schedule one work-group per core, with each work-group being roughly of size $4 \times n_a$[8]. Figure 4 illustrates this scheduling model for both a multi-core CPU (small number of compute units per core) and a GPU (multiple compute units per core).

Within this model, each kernel is invoked exactly $4 \times n_c \times n_a$ times, meaning each invocation has to have a sequential part that operates on $\lceil \frac{n}{4 \times n_c \times n_a} \rceil$ elements. This sequential part leads to another architecture-dependent problem: Graphics cards and CPUs use different methods to increase memory bandwidth. While graphics cards try to coalesce multiple neighboring accesses from different threads into a single operation, multi-core CPUs rely on prefetching and caching. This results in different optimal memory access patterns for the threads within a work-group. Graphics cards prefer that neighboring threads access neighboring locations in memory, since this pattern can easily be coalesced. On multi-core CPUs on the other hand, a single thread should access consecutive locations in main memory, since this pattern will lead to optimal caching behavior. This is also shown in Figure 4.

We introduce these different access patterns into our kernels by injecting the current architecture as a pre-processor constant into the kernel build process. The access patterns are then chosen within the kernel based on the value of this constant.

---

[8]Choosing a higher number of invocations than there are compute units allows the device to hide transfer latencies by swapping out invocations that wait for I/O.

4.3 Modifications to MonetDB Code

Integrating Ocelot into MonetDB proved to be surprisingly lightweight. In particular, we only needed to make four small changes to existing MonetDB code parts to fully integrate Ocelot:

- We added a flag to MonetDB's BAT descriptor to indicate whether a given BAT is currently owned by Ocelot.

- In order to make Ocelot aware of MonetDB's resource management decisions, we added callbacks to our Memory Manager when BATs are deleted or recycled. The Memory Manager uses this information to directly remove buffers for unused and deleted BATs from the device cache.

- We added a new optimizer pipeline that includes the Ocelot query rewriter. This pipeline is based on the sequential optimizer pipeline, which is identical to the default MonetDB optimizer pipeline, minus parallelization.

- Since the Intel OpenCL SDK makes extensive use of SSE operations, which only operate on 128-byte aligned memory, we modified MonetDB's memory management to return 128-byte aligned memory chunks.

5. EVALUATION

In this section, we provide an analysis of the performance of Ocelot on various hardware configurations. We investigate the performance and scaling characteristics of single operators through microbenchmarks and demonstrate how our system behaves for complex SQL statements by running experiments using a modified[9] TPC-H [36] benchmark.

5.1 Experimental Setting

In our evaluation, we compare the following four configurations:

Sequential MonetDB This configuration marks the baseline for our comparisons. We run MonetDB without any parallelism on the CPU to get an understanding of the performance when running on a single CPU core.

Parallel MonetDB In this configuration, we use the Mitosis and Dataflow optimizers of MonetDB to achieve efficient intra-operator parallelism [25]. This configuration demonstrates the performance that is achievable by hand-tuning operators for a multi-core CPU.

Ocelot on CPU For this configuration, we run Ocelot on a multi-core CPU, demonstrating how it compares with MonetDB's hand-tuned operators.

Ocelot on GPU The final configuration runs Ocelot on an off-the-shelf graphics card to demonstrate that our system is indeed hardware-oblivious.

Note, that when running on the GPU, Ocelot features the same functionality as on the CPU. However, due to limited device memory, the scope is restricted to smaller input sizes. We thus see the GPU component of Ocelot as a way to quickly answer queries on a small hot set of the data, which can be kept resident in the device's global memory.

Due to limited resources, we tested our prototype only on an nVidia graphics card and an Intel x86 CPU. In general, Ocelot should however run on a much wider variety of devices. In particular, OpenCL has been ported to several device classes, such as

---

[9]For details, see Appendix Section A.

FPGAs [4], APUs [1] and the IBM Cell and Power processor line. We believe that more vendors will start to support this programming framework in the future, making our implementation portable even to future devices.

We conducted our experiments on a custom-built server with the following specification:

- Intel Xeon E5620, 64-bit, four cores running at 2.4GHz, 12MB Cache.

- 32GB of DDR-3 RAM, clocked at 1333MHz.

The server is equipped with a middle-class NVIDIA GTX460 graphics card, sitting in a PCIexpress 2.1 x16 slot. The graphics card has the following specification:

- NVIDIA Fermi GF104 core:
  - Seven multiprocessors, each having 48 compute units.
  - 48KB of local device memory, 64KB of constant buffer per compute unit.

- 2GB of DDR4 graphics memory, clocked at 1800MHz.

The experiments were conducted on a 64-bit Scientific Linux 6.2 (Linux kernel 2.6.32-220.7.1.e16.x86_64). The graphics card was controlled with the NVIDIA 310.32 driver for 64-bit Linux systems. We used Intel's SDK for OpenCL Applications 2013 XE Beta to run our operators on the CPU.

5.2 Microbenchmarks

We use microbenchmarks to get an understanding of the performance characteristics and scaling behaviour of single operators. Each microbenchmark was created by piping a simple SQL query containing the operation of interest through MonetDB's EXPLAIN command. The resulting plan was then manually stripped of unnecessary operators to focus on the relevant parts. Unless otherwise noted, all microbenchmarks were run on synthetic, uniformly distributed test data.

We ran each benchmark ten times, measuring the average runtime across those invocations. Single timings were measured with millisecond accuracy using MonetDB's mtime.msec() function. For the GPU configuration, measurements do not include data transfer to or from the device, as we are only interested in the actual operator performance. The results of our experiments are depicted in Figure 5. In the figures, CPU & GPU denote the runtime of our Ocelot operators on the respective configurations, MS denotes MonetDB's sequential performance, and MP MonetDB's performance when utilizing all cores. If a line for GPU measurements ends midway, we reached the device memory limit for this operator.

5.2.1 Selection

Figure 5(a) shows the scaling behavior for a range[10] selection with .05 selectivity. As expected, the operation scales linearly with the input size on all configurations. It is interesting to note that Ocelot is faster on the CPU than parallel MonetDB. This is caused by Ocelot's selection operator generating bitmaps as a result, while MonetDB returns the list of qualifying oids – which is simply larger. Figure 5(b) shows the impact of predicate selectivity for a range selection on a 400MB column. Since Ocelot returns bitmaps, the runtime stays constant, while MonetDB has to materialize the list of qualifying oids, which gets more expensive as the result set grows.

---

[10]We only measured range selections, as point selections use a hash selection in MonetDB, which Ocelot does not support yet.

5.2.2 Left Fetch Join

The left fetch join is one of the most frequently used operators in MonetDB/Ocelot. Its main task is to merge two columns, for instance when running a projection, building a result set, or running a PK-FK join using a join index. Figure 5(c) shows how the runtime of the left fetch join changes with increasing input size. For this benchmark, two columns of the same relation were joined via their row identifiers, i.e., the left fetch join performs a projection of two columns. As expected, all configurations scale linearly with the input size. The relative positioning of the four configurations meets the expectations: Ocelot is as fast[11] as the parallel – and faster than the sequential – MonetDB instance when running on the CPU, and is clearly the fastest option, when running on the GPU.

5.2.3 Aggregation

Figure 5(d) shows the runtimes of the minimum aggregation operator. While the operation scales linearly with the input size for all configurations, it is interesting to note that parallel MonetDB is roughly 30% faster than Ocelot on the CPU. This is somewhat surprising, as aggregation is an operation that is very easy to parallelize and should therefore not introduce a significant performance penalty. We believe that this issue is likely caused by either a compiler or a runtime problem of the used Intel OpenCL SDK, which was still a beta version when we ran the experiment.

5.2.4 Hash Table Creation

As discussed in Section 4, our hashing algorithm uses atomic operations to build a hash table in parallel. Especially on the CPU, these atomic operations lead to a drastic loss in performance, as can be seen in Figure 5(e), which illustrates the time it takes to build a hash table for a column with 100 distinct values. While our algorithm scales linearly with the input, it is clearly slower than the sequential hash table creation used by MonetDB. Figure 5(f) shows how the number of distinct values affects the time to build a hash table. Since resolving a hash collision in parallel is rather costly, hashing in Ocelot becomes more expensive as the number of distinct values – and thus hash collisions – grows. In contrast to the other configurations, the hashing performance of Ocelot on the CPU actually increases, due to the higher overhead caused by atomic operations frequently accessing the same memory address. Interestingly, the GPU does not show this pattern.

Looking at these results, it is obvious that hashing is one of the major shortcomings of Ocelot. This is partly caused by the inherent difficulty of building a global hash table in parallel – an operation that usually requires either partitioning or extensive synchronization efforts. In order to make Ocelot fully competitive, it will be vital to invest further research into a more efficient, portable hashing strategy that does not rely extensively on atomic operations. This could for instance build on work from hashing in main-memory database systems, which use partitioning to avoid building a single global hash table [26].

5.2.5 Grouping

Figure 5(g) illustrates how the group-by operator scales with increasing input size on a column with uniformly distributed values from 1 to 100. As can be seen, the operator scales linearly for all configurations. Figure 5(h) shows how the group-by operator scales when we increase the number of groups, fixing the size of

---

[11]Note that we excluded the time required to merge the final result for the parallel MonetDB configuration. This final step introduced significant overhead in our test scenario, increasing the runtime by about one order of magnitude.
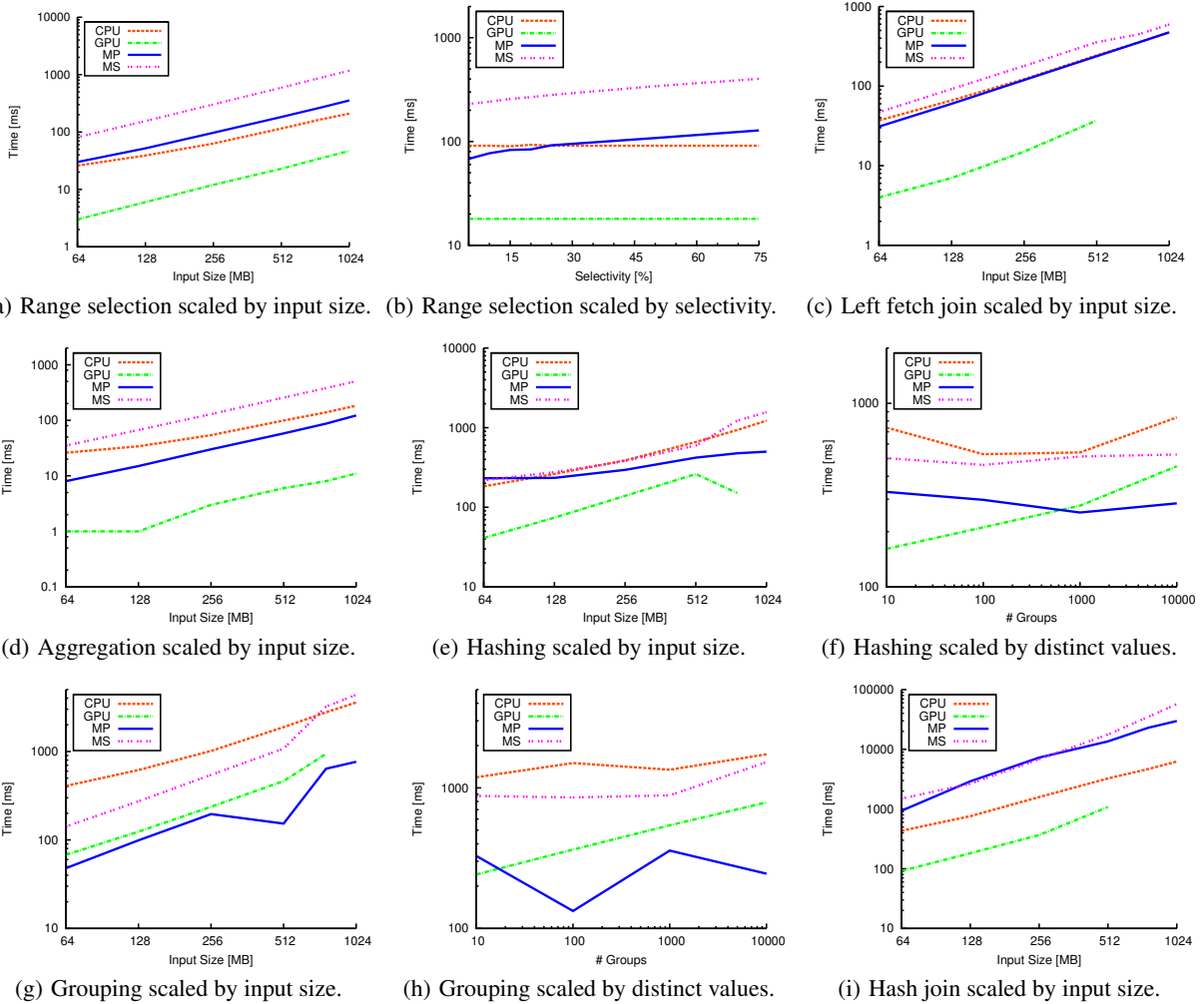
(a) Range selection scaled by input size.   (b) Range selection scaled by selectivity.   (c) Left fetch join scaled by input size.

(d) Aggregation scaled by input size.   (e) Hashing scaled by input size.   (f) Hashing scaled by distinct values.

(g) Grouping scaled by input size.   (h) Grouping scaled by distinct values.   (i) Hash join scaled by input size.

Figure 5: Microbenchmarks.

the grouping column to 400MB. There are some interesting observations: First, Ocelot on the CPU is clearly the slowest option among the tested configurations. Second, even when running on the GPU, Ocelot is only as fast as parallel MonetDB. This can be explained by the shortcomings of Ocelot's parallel hashing algorithm, which is extensively used by the grouping operator.

5.2.6 Hash Join

We tested our hash join implementation with a primary key - foreign key (PK-FK) join scenario. In the experiment, we increased the size of the probing table, while keeping the build side fixed to 100 keys. Looking at Figure 5(i), we observe[12] linear scaling with the input size for all configurations. Seemingly, once the hash-table is built, the actual look-up is highly efficient in Ocelot, clearly outperforming both parallel and sequential MonetDB. Since building hash-tables is highly expensive compared to actually using them, we maintain a cache of all built hash tables of base tables in the Memory Manager.

5.2.7 Sort

Figure 6 shows the performance of our sorting operator. As discussed in Section 4, Ocelot uses a binary radix sort implementation [22], which requires a constant number of passes over the whole data set. The number of passes depends on the size of the key and the chosen radix. For the CPU implementation, we use a radix of eight bits, for the GPU a radix of four bits. We observe linear sorting performance with an increasing input size for all configurations. With this sort strategy, both on CPU and GPU Ocelot outperforms MonetDB's sort algorithm, which is based on quick- and merge-sort.

5.3 TPC-H

In this Section, we demonstrate the performance of Ocelot when executing complete SQL queries. For these experiments, we chose the TPC-H benchmark [36], modifying it slightly to match the feature set of Ocelot. Details about the modifications can be found in Appendix A[13]. While these modifications change the size and shape of the final result sets, they do not impact performance com-

---

[12]Note that the measurements for this experiment do not include the time it takes to build the hash table.

[13]Note, while the workload in the Appendix contains query 18, we had to skip it due to problems with MonetDB.
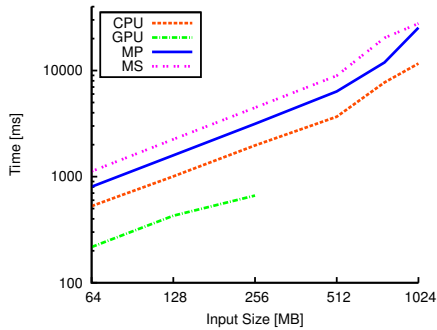
Figure 6: Performance of the sort operation.

parisons with MonetDB. We used TPC-H in scale-factors 1, 2, 4, 8 & 50 for our experiments.

Each query was run directly from the MonetDB SQL interface, using MonetDB's optimizer and Ocelot's query rewriter to transparently generate a correct query plan that uses Ocelot's operators. For each query, we computed the average runtime over five runs, using the timing reported by the SQL front-end of MonetDB as our measurements. The collected measurements can be seen in Figure 5.3.

All experiments were performed using a hot cache, i.e., we ran each query multiple times and only started measuring with the second run. For the GPU measurements, this means that parts of the input were already cached on the device when measurement began. However, the measured times still contain all data transfer of uncached input data and the complete result transfer time.

5.3.1 Small Data Set

For the first test series, we ran our workload on a TPC-H data set with a scale factor of one. The results of this experiment are depicted in Figure 7(a). A few interesting observations: First, Ocelot on the CPU clearly offers the worst performance. The runtimes are often multiple times slower than even those of sequential MonetDB. In fact, there is not a single query where any other configuration is slower than Ocelot on the CPU. Second, Ocelot on the GPU offers competitive performance to MonetDB: For most queries, we outperform the parallel MonetDB configuration. In particular, the GPU is only significantly slower for query 21, which contains several joins that require hashing. In summary, for small data-sets, Ocelot offers very good performance on the GPU. However, the CPU implementation fails to meet the expectations for this data set.

5.3.2 Intermediate Data Set

For the next test series, we increased the data volume slightly to a scale-factor of eigth. Figure 7(b) shows the measurements for this experiment. Looking at the figure, we see a much more balanced picture than in the last case. The performance difference between parallel MonetDB and Ocelot on the CPU are much less dramatic, with Ocelot offering clearly competitive runtimes for several queries. However, there are still a few queries where Ocelot is much slower than MonetDB, in particular queries 10, 11, 17, and 21. Taking a closer look at these queries, we again found that the primary offender for this drop in performance were hash-join operations. When running the intermediate-sized scenario on the GPU, Ocelot still offers very good performance. However, the performance lead over parallel MonetDB is visibly smaller than in the last scenario. This is caused by device memory limitations on

the graphics card: The scale-factor eight TPC-H instance was the largest one we could run on the graphics card. Internally, Ocelot had to continuously swap data in and out of the device memory to free up resources during query execution. This incurred high data transfer costs, that ate up the performance lead of Ocelot over MonetDB.

Looking at the results of this experiment, it seems that Ocelot on the CPU seems to have a better scaling behaviour than MonetDB. While Ocelot was clearly outmatched for the small scenario, it can provide competitive performance when the data volume is increased. We decided to take a closer look at the scaling behaviour of Ocelot with increasing data volume. For this evaluation, we took query 01 of TPC-H and measured its runtime on TPC-H instances with increasing scale-factors. Figure 7(d) illustrates the results of this experiment. There are a few interesting observations. First, as expected, all configurations scale linearly with the input size. However, on the GPU, there is a clear non-linear drop in performance for the larger scale-factor, which is caused by increased data transfer due to swapping operations. Second, by extrapolating the measurements in Figure 7(d) to an empty dataset, we can see that Ocelot has roughly a one second overhead when running on the CPU, explaining the bad performance for the small input size. For all other configurations, this extrapolated overhead estimate is close to zero. Note that this overhead is neither caused by Ocelot itself, nor by a fundamental limitation of the programming model, as in those cases the GPU trend should show a similar overhead. Instead, we believe that this indicates framework overhead that is introduced by the Intel OpenCL SDK. We hope that future releases of the SDK will improve performance and help to remove this overhead, so that Ocelot becomes competitive even for small data sets when running on the CPU.

5.3.3 Large Data Set

For the final test series, we tried to minimize the impact of the Intel OpenCL framework overhead on our measurements by scaling the test up to a very large data set. We decided to pick a scale-factor of 50 for this experiment. Figure 7(c) shows the results of for sequential & parallel MonetDB, and Ocelot on the CPU. Due to the small amount of device memory, we could not use the graphics card for this experiment. The results clearly show that Ocelot can compete with the parallel MonetDB implementation for large data sets. In fact, apart from three queries, Ocelot is on par – or even outperforms – MonetDB. This confirms our belief, that the Intel OpenCL SDK we used has some performance issues with small data sets.

5.4 Summary of Results

In this paper, we wanted to demonstrate that a hardware-oblivious database engine design is feasable and can provide competitive performance to hand-tuned parallel database operators. To achieve this goal, we ran a series of experiments comparing our hardware-oblivious prototype Ocelot against MonetDB.

While our experiments showed comparable – or even superior – performance in most scenarios, there were also multiple cases for which our prototype was clearly beaten by MonetDB. However, we believe that future iterations of Ocelot will close this performance gap. There are multiple reasons for this: First, Ocelot is at an early development stage and still has several opportunities to improve performance, e.g. by fine-tuning parameters, introducing new algorithms, and optimizing existing ones. Second, existing OpenCL frameworks and compilers are still in their infancy, and often have bugs and performance problems. We believe that as OpenCL becomes more mature, and as vendors become more familiar with it, we will automatically see performance improvements. Third, we
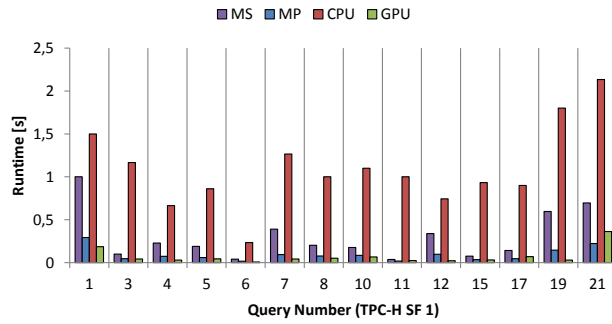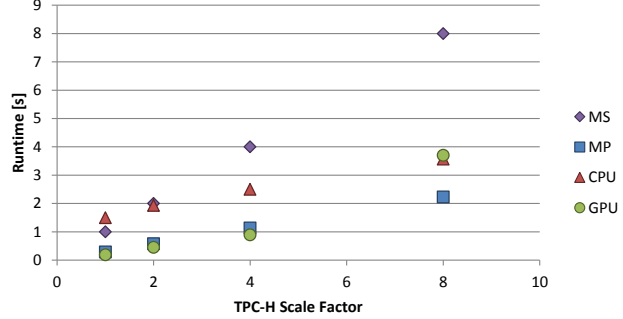
(a) TPC-H performance, Scale Factor 1.
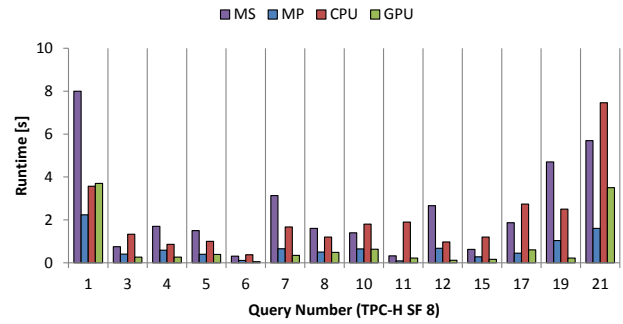
(b) TPC-H performance, Scale Factor 8.
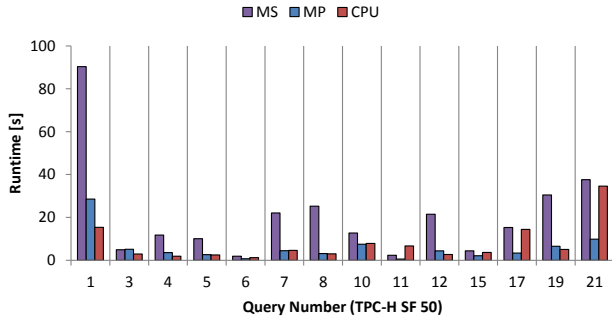
(c) TPC-H performance, Scale Factor 50.

(d) TPC-H Q1 scaling performance

Figure 7: TPC-H Measurements.

were reusing MonetDB's query plans, which were generated by a query optimizer that is tuned towards CPU-based architectures. Given that different compute devices have vastly different capabilities and characteristics, it is likely that they also require different query plans to provide optimal perfomance. Ideally, an "hardware-oblivious optimizer" would understand these differences and take them into account during plan generation.

Taking these considerations into account, and given that Ocelot offered competitive performance in most experiments, we achieved our goal of demonstrating that a hardware-oblivious database design is not only feasible, but is indeed a solid potential choice when building a parallel database engine for tommorrow's hardware.

6. RELATED WORK

Already back in 1978, DeWitt proposed to use co-processors to accelerate data processing in DIRECT [13]. However, at the time, these co-processors were rapidly overtaken by the fast development of single-core CPUs. With the wide-spread availability of graphics cards, the interest in data co-processing grew anew. Govindaraju et al. presented their work on relational operators on graphics adapters in [17]. He et al. investigated relational query processing on GPUs by implementing a complete set of relational operators for the GPU and a distinct set for the CPU in their custom-tailored database system GPUDB [19]. In general, all of those systems were limited by the small amount of device memory, and the comparably slow PCIe bus. In [21], Heimel et al. suggested circumventing this problem by using a GPU to assist during query optimization instead. Besides graphics cards, there is a plethora of work on data processing on other non-traditional architectures: The work of Mueller et al. explained how to express database operations us-

ing the building blocks of FPGAs [28]. Gold et al. analyzed data processing on network processors [15] and Heman et al. addresses data processing on Cell processors [23].

A common point among these research papers is the focus on a single architecture. They analyze the suitability of a certain device for data processing considering performance. While this is a valid and important aspect on it's own, our approach differs greatly. We focus on the maintainability and complexity of a system incorporating more than just CPUs for data processing and show that through using hardware abstraction, we arrive at a highly portable system whose performance is competitive to hand-tuned implementations.

The benefit of aiming for hardware-oblivious operations has been demonstrated before, for instance by Balkesen et al. in [5], who focus on auto-tuning a hash-join operator to different multi-core architectures. There has also been work on designing programming models that can easily be mapped to multiple architectures. An example for this is the data-parallel Haskell project [11], which has been demonstrated to be easily mappable to both CPU and GPUs [27]. However, to the best of our knowledge, we provide the first analysis of designing a hardware-oblivious database engine that targets such diverse architectures as CPUs and GPUs.

7. FUTURE WORK

At the moment, Ocelot uses the exact same algorithm on all devices, which is probably overly optimistic: It will thererefore be interesting to take a closer look at the limits of hardware-oblivious designs. As a first step, we plan to provide a set of alternative algorithms for each operator, with the optimizer selecting the best-fitting algorithm for the given device. This will require an automatic understanding of the performance characteristics of the given

hardware, which could – for instance –, be obtained by automatically generating a device profile from standardized benchmarks.

Another restriction of Ocelot is that it only uses one device at a time. Reasonably supporting multiple devices would call for automatic operator placement. As a prerequisite, this requires an understanding of specific hardware properties, which could also be based on automatically generated device profiles. Once the cost model is defined, a hardware-aware query optimizer strategy is required to decide on the actual placement. This work could build on existing work on self-tuning cost models and search strategies for CPU/GPU hybrids [9, 19]. Given the heterogeneous environments that Ocelot targets, it would also be highly interesting to investigate non-traditional cost-metrics for query optimization, such as energy consumption or (monetary) cost per result tuple.

While the kernel programming model offers a suitable abstraction for hardware-oblivious parallel operators, it is rather low-level and has a steep learning curve. This makes it a bad choice for a user-facing programming model: Designing or adopting a different model will be essential to support more complex user-defined functions. This could build on existing work on programming models for large-scale data processing like MapReduce [12] or PACT [6], or programming languages for multi-core architectures like data-parallel Haskell [11]. In order to fully support such a model, we would also require a code generation module that maps the model to OpenCL code.

8. CONCLUSION

This paper motivates the idea of designing database engines in a hardware-oblivious manner to cope with the increasingly diverse hardware landscape. Our proposed design centers around a single set of operators, which are implemented against an abstract hardware representation. At runtime, a vendor-provided driver maps this abstract representation down to the actual hardware. This approach reduces the development overhead for database systems that support multiple architectures.

We demonstrated the feasibility of such a design by presenting Ocelot[14], our hardware-oblivious extension of the open-source column-store MonetDB. Ocelot uses OpenCL to implement a set of hardware-oblivious drop-in replacements for MonetDB's operators. Through experimental evaluation against MonetDB via microbenchmarks and a TPC-H-derived workload, we demonstrated that a hardware-oblivious design can achieve competitive performance when running on a multi-core CPU. Furthermore, we could show that – if the problem fits onto the device memory – we can outperform MonetDB by running Ocelot on a graphics card.

9. REFERENCES

[1] Advanced Micro Devices. OpenCL Zone. `http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/`, January 2013.

[2] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Real-time parallel hashing on the gpu. In *ACM SIGGRAPH Asia 2009 papers*, SIGGRAPH Asia '09, pages 154:1–154:9, New York, NY, USA, 2009. ACM.

[3] D. A. F. Alcantara. *Effcient Hash Tables on the GPU*. PhD thesis, University of California, Davis, 2011.

[4] Altera Corporation. OpenCL for Altera FPGAs: Accelerating Performance and Design Productivity.

[5] C. Balkesen, J. Teubner, G. Alonso, and M. T. Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. *ETH Zurich, Systems Group, Tech. Rep*, 2012.

[6] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 119–130. ACM, 2010.

[7] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking The Memory Wall In MonetDB. *Communications of the ACM*, 51(12):77 – 85, December 2008.

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

[9] S. Breß, F. Beier, H. Rauhe, E. Schallehn, K.-U. Sattler, and G. Saake. Automatic selection of processing units for coprocessing in databases. In *Advances in Databases and Information Systems*, pages 57–70. Springer, 2012.

[10] N. Cascarano, P. Rolando, F. Risso, and R. Sisto. infant: Nfa pattern matching on gpgpu devices. *SIGCOMM Comput. Commun. Rev.*, 40(5):20–26, Oct. 2010.

[11] M. M. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18. ACM, 2007.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] D. J. DeWitt. Direct - a multiprocessor organization for supporting relational data base management systems. In *Proceedings of the 5th annual symposium on Computer architecture*, ISCA '78, pages 182–189, New York, NY, USA, 1978. ACM.

[14] I. García, S. Lefebvre, S. Hornus, and A. Lasram. Coherent parallel hashing. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, SA '11, pages 161:1–161:8, New York, NY, USA, 2011. ACM.

[15] B. Gold, A. Ailamaki, L. Huston, and B. Falsafi. Accelerating database operators using a network processor. In *Proceedings of the 1st international workshop on Data management on new hardware*, DaMoN '05, New York, NY, USA, 2005. ACM.

[16] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM.

[17] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 215–226, New York, NY, USA, 2004. ACM.

[18] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 46:1–46:12, New York, NY, USA, 2007. ACM.

---

[14]Ocelot is open source and can be downloaded from: `http://goo.gl/GHeUv`.

http://www.altera.com/products/software/opencl/opencl-index.html, January 2013.

[19] B. He, M. Lu, K. Yang, R. Fang, N. Govindaraju, Q. Luo, and P. Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)*, 34(4):21, 2009.

[20] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 511–524. ACM, 2008.

[21] M. Heimel and V. Markl. A first step towards gpu-assisted query optimization. *ADMS*, 2012.

[22] P. Helluy. A portable implementation of the radix sort algorithm in opencl.

[23] S. Héman, N. Nes, M. Zukowski, and P. Boncz. Vectorized data processing on the cell broadband engine. In *Proceedings of the 3rd international workshop on Data management on new hardware*, page 4. ACM, 2007.

[24] D. Horn. *GPU Gems 2nd Edition*, chapter Stream reduction operations for GPGPU applications. Addision Wesley, 2005.

[25] M. Ivanova, M. Kersten, and F. Groffen. Just-in-time data distribution for analytical query processing. In *Advances in Databases and Information Systems*, pages 209–222. Springer, 2012.

[26] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: fast join implementation on modern multi-core cpus. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, 2009.

[27] S. Lee, M. M. Chakravarty, V. Grover, and G. Keller. Gpu kernels as data-parallel array computations in haskell. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*, 2009.

[28] R. Mueller, J. Teubner, and G. Alonso. Data processing on fpgas. *Proc. VLDB Endow.*, 2(1):910–921, Aug. 2009.

[29] C. Nvidia. Compute Unified Device Architecture Programming Guide. *NVIDIA: Santa Clara, CA*, 83:129, 2007.

[30] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[31] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[32] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 351–362, New York, NY, USA, 2010. ACM.

[33] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[34] D. Singh and S. P. Engineer. Higher level programming abstractions for fpgas using opencl. In *Workshop on Design Methods and Tools for FPGA-Based Acceleration of Scientific Computing*, 2011.

[35] The Khronos Group Inc. OpenCL - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/, May 2011.

[36] Transaction Processing Performance Council. TPC-H. http://www.tpc.org/tpch/default.asp, May 2011.

[37] R. Wu, B. Zhang, M. Hsu, and Q. Chen. Gpu-accelerated predicate evaluation on column store. In *Proceedings of the 11th international conference on Web-age information management*, WAIM'10, pages 570–581, Berlin, Heidelberg, 2010. Springer-Verlag.

APPENDIX

A. MODIFICATIONS TO TPCH-H

Due to the limited scope of our implementation, we had to make some changes to TPC-H [36] for the evaluation. While these modifications change the size and content of the final result sets, they do not impact our performance comparison with MonetDB. The schema modifications were relatively straightforward: Since Ocelot does not support operations on data types that are larger than four bytes, we replaced all DECIMAL fields by REAL fields.

The query modifications were more involved since we needed to remove unsupported features. In particular, Ocelot does not support operations on strings beside equality comparisons, joins between eight-byte columns, multi-column sorting, and an efficient top-k operator for limit operations. Note that those missing operations are not caused by any fundamental restriction of the kernel programming model, and could be integrated with moderate overhead.

A.1 Modified Workload

In total we omitted seven queries (2, 9, 13, 14, 16, 20 and 22), and modified six queries (1, 3, 7, 10, 18 and 21). The remaining nine queries were not modified. The detailed list of changes is given next:

Q1 Removed the sorting clause for l_linestatus.

Q2 Omitted, since it requires both a string like expression and a join between eight-byte columns in the MonetDB plan.

Q3 Removed the sorting clause for o_orderate. Removed limit expression.

Q7 Removed the sorting clauses for supp_nation and l_year.

Q9 Omitted, since it requires a like expression on p_name.

Q10 Removed limit expression.

Q13 Omitted, since it requires a like expression on p_comment.

Q14 Omitted, since it requires a like expression on p_type.

Q16 Omitted, since it requires a like expression on p_type.

Q18 Removed the sorting clause for o_orderdate. Removed limit expression.

Q20 Omitted, since it requires a like expression on p_name.

Q21 Removed the sorting clause for s_name.

Q22 Omitted, since it requires substring expressions on c_phone.