QcLab: A Framework for Query Compilation on Modern Hardware Platforms

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r
I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Henning Funke

Dortmund

2022

Tag der mündlichen Prüfung:	25.03.2022
Dekan/Dekanin:	Prof. Dr.-Ing. Gernot Fink
Gutachter/Gutachterinnen:	Prof. Dr. Jens Teubner
	Prof. Dr. Thomas Neumann

Abstract

As modern in-memory database systems achieve higher and higher processing speeds, the performance of memory becomes an increasingly limiting factor. Although there has been significant progress, the bottleneck only has shifted. While earlier systems were optimized for *memory latencies*, current systems are rather affected by the limited *memory bandwidth*.

Query compilation is a proven technique to address bandwidth limitations. It translates queries via *Just-In-Time compilation* to native programs for the target hardware. The compiled queries execute with very high efficiency and only with a bare minimum of communication via memory. Despite these important improvements, the benefit of query compilation in certain scenarios is limited.

On the one hand query compilers typically use standard compiler technology with relatively long compilation times. Therefore the overall execution time can be prolonged by the additional compilation time. On the other hand, not all emerging database technology is compatible with the approach. Query compilation uses a tuple-at-a-time processing style that departs from the column-at-a-time or vector-at-a-time approaches that in-memory systems typically use. Especially data-parallel processing techniques, e.g. SIMD or coprocessing-techniques, are challenging to use in combination with the approach.

This work presents *QCLab*, a framework for query compilation on modern hardware platforms. The framework contains several new query compilation techniques that allow us to address the mentioned shortcomings and ultimately to extend the benefit of query compilation to new workloads and platforms. The techniques cover three aspects: *compilation*, *communication*, and *processing*. Together they serve as basis for building highly efficient query compilers. The techniques make efficient use of communication channels and of the large processing capacities of modern systems. They were designed for practical use and enable efficient processing, even when workload characteristics are challenging.

Contents

1 Introduction

Increasing main memory capacities have led to the development of *in-memory database systems*. Instead of using hard disk drives (*disks*) for storage, in-memory database systems keep their entire database in main memory. This approach has removed the disk access bottleneck, which previously was the strongest limiting factor for database processing. Systems that are optimized for in-memory processing, however, show a much greater potential [34]. In-memory technology has increased the relevance of database systems in existing [27] and new domains, e.g. data science [92]. A major benefit of such optimized systems are the high peformance gains that can be achieved.

An important factor for the performance of in-memory systems are *memory latencies* [15]. Memory latencies cause *memory stalls* when query execution waits for memory requests to be processed. In many cases the CPU caches help to reduce the number of memory stalls by providing cached data with shorter access times. Not all workloads, however, can benefit from the caches to a similar extent. Especially scan and hash-based techniques, that are typical for database workloads, have insufficient memory access locality for an effective use of the (relatively small) CPU caches. To address the impact of memory latencies, substantial research and development work has been performed by the database community. This work includes processing techniques for joins [15, 8], aggregation [68, 103], and projection [60]. The techniques typically increase the locality of data access, for instance with partitioning mechanisms, to achieve a more effective use of the CPU caches. Additional work leverages CPU prefetching features to *hide* memory latencies [23].

More recently another performance factor has gained importance. *Memory bandwidth* characterizes the data volume that can be serviced by memory over time. Workloads that are memory-intensive, such as database processing, are bounded in

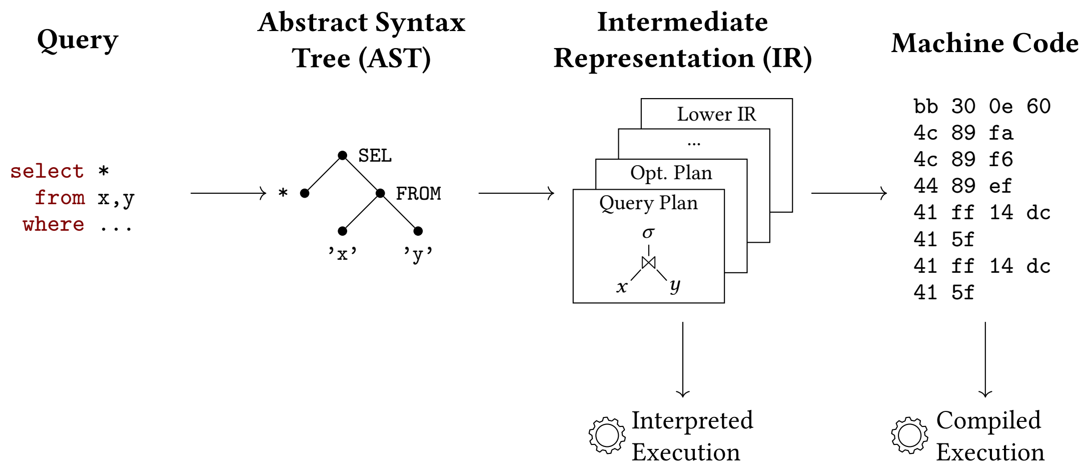


Figure 1.1: Translation and execution of a query by a database system.

throughput by the available bandwidth. Several techniques were proposed to reduce the bandwidth demand for database processing. Column-based technology [15] uses homogeneous data layouts to increase the efficiency of memory loads and stores. Vector-at-a-time processing [16] improves the cache-utilization for the communication between operations. However, technological improvements in hard- and software enable processing speeds, that make it more and more challenging to provide sufficient bandwidth for the I/O operations during processing. While some technology exacerbates the issue (e.g. multi-core processors have limited per-core bandwidth), other may provide potential solutions (e.g. parallel coprocessors with high-bandwidth memory). This thesis deals with processing techniques that can be used to address bandwidth limitations during query processing. We take a look at different existing techniques and focus on query compilation as a promising approach.

1.1 Query Translation and Execution

Queries are data processing tasks that run against the database to extract or to modify information. From a more general perspective, however, *a query is a program* and executing queries has many commonalities with executing other (general purpose) programs.

Figure 1.1 illustrates how database systems typically execute queries. The process resembles the way compilers work and a similar process (without the database specifics) is used by general purpose compilers, e.g. to compile C++ or Java programs. The query statement on the left is translated to a series of different representations to prepare it for execution (further on the right). The first step translates the query to the *abstract syntax tree* (AST). This step is performed by the parser. It applies the syntax rules of the SQL language to derive the query statement from the language. The AST is the result of these rules and represents the *query syntax*. The query planner then maps the syntactic elements of the AST to relational operators (e.g. join $\bowtie$, selection σ) that form the *query plan*. The query plan is a tree of relational operators that represents the *query semantics*.

The query plan is the first of several *intermediate representations* (IRs). Figure 1.1 shows the IRs with multiple layers that result from successive translation steps. After the query plan, several other IRs, e.g. the optimized query plan, follow. Each IR allows the compiler to use a well-suited representation when performing complex tasks such as resource allocation, optimization, or translation to a lower (machine-near) level.

After translation to the desired representation queries are executed. When an intermediate representation is used, queries are executed with *interpreted execution*. When machine code is used, queries are executed with *compiled execution*. In the following we discuss the implications of both modes.

Interpreted Execution Traditionally database systems have used interpreted execution, e.g. Volcano [36], to execute queries. The *interpreter* takes an intermediate representation of the query and executes its elements step by step. For instance, it traverses the query plan's operators and for each traversal step, it executes the operator's implementation. Some systems, e.g. PostgreSQL [93], interpret the query plan directly. Other systems, e.g. MonetDB [15] or SQLite3 [91], translate the query plan to an imperative IR with *commands* and execute those. For simplicity we call all executed IR elements commands. Commands may (and typically will) read tables from the database, e.g. when executing scan operators. The rows resulting from interpretation of the IR form the query result.

Interpreted execution includes an additional cost for performing the interpretation itself. Every command that is executed has to be *decoded* first. During decoding the interpreter reads a command and jumps to the code location with its implementation. To perform this decoding work, the CPU executes additional instructions that add to the workload of operator processing. For in-memory techniques, where disk access no longer dominates execution times, this quickly becomes a limiting factor. Especially interpretation techniques that perform decoding for every row of a table are strongly affected. The accumulating interpretation work can dwarf the actual processing work and has been identified as *interpretation overhead* [16].

Compiled Execution Query compilation is a processing technique that solves the problem of interpretation overheads. As illustrated on the right-hand side of Figure 1.1, the approach takes query translation one step further and translates queries to machine code before their execution. This enables highly efficient query processing and makes the approach very attractive for addressing the performance demands of modern in-memory systems. At the same time, database systems are compilers anyhow and are therefore well suited for applying compilation techniques. For these reasons, we make query compilation the focus of this work

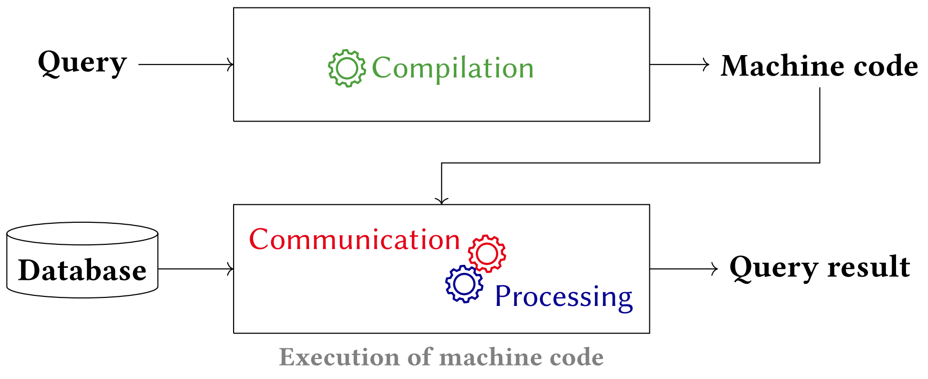


Figure 1.2: Illustration of compiled query execution. Three central concepts that are addressed in this thesis are visualized with gear symbols.

in the following we present three concepts with a central role.

Compiled query execution is illustrated in Figure 1.2. The figure shows three central concepts with gear symbols. The top part of the figure illustrates *compilation*, the bottom part *execution*. In the top part Compilation performs translation of the query to a machine code representation, the *compiled query*. This translation is a form of Just-In-Time (JIT) compilation because queries are compiled immediately when the database is queried. It therefore relies on short compile times. To perform compilation, database systems translate the query plan (or other representations of the query) to a low-level IR, e.g. LLVM IR [55], which supports translation to machine code. The IR library is then used to translate the IR to the compiled query, which is executed afterwards.

Query execution (Figure 1.2, bottom) is initiated by pointing the CPU to the compiled query (e.g. with a function call). The CPU proceeds with execution of the compiled query. During execution Processing concerns the execution of relational operators. The operators of the query are stringed together in the machine code and are therefore executed by the CPU coherently. There is no decoding work required as the CPU steps from operator to operator without interventions being necessary. This makes processing very efficient and removes the interpretation overhead entirely. Communication services processing. This is illustrated by the engaging gears (Figure 1.2, bottom). It provides data access and data passing methods that are used for sharing data between operators and for accessing relations. By compiling several relational operators to the same binary function, query compilation has the opportunity for a very efficient way of sharing data between operators. The operators can share data simply by accessing the same CPU registers. This enables highly efficient use of memory bandwidth and improves the communication speed over techniques that materialize tuples in memory or in caches.

Through the compilation to machine code, compiled execution has the potential to reach *optimal* execution performance. To reach this optimum, however, a costly compilation process would be required to determine the corresponding machine code representation. The two goals of low execution times and low compilation oppose each other in the goal for overall short response times. A good solution for this tradeoff is an important design decision for query compilers and can also depend on workload characteristics. The techniques presented in this work address different aspects of this tradeoff and represent new solutions for challenging workloads.

1.2 Open Challenges

We discuss several open challenges for query compilation in relation to the previously introduced concepts. Some of the open challenges result from the mismatch between interpretation-based database technology and compiled query exeution. Other challenges stem from requirement differences between query compilers and classical compilers.

Compilation During compilation, compilers go through several translation steps and yield an intermediate representation (IR) of the program. The IRs help compilers to handle translation complexity and to apply optimizations. Query compilers have used existing compiler infrastructure for intermediate representations. However, the generality of such solutions limits their benefit. Queries have a simpler and more well-defined form than general purpose programs and may therefore be translated with less effort. Query processing represents new incentives and opportunities for the development of compiler techniques that are specifically made for database workloads.

Communication During query processing data moves from their storage locations into processing units and back. For these communication processes different hardware channels, e.g. bus links and main-memory access, with varying transfer speeds are used. Especially for coprocessor systems this can become a strong limiting factor as they rely on additional communication channels during computations. The issue is amplified by processing techniques that perform multiple passes over the input, which are typical on massively parallel processing devices.

Processing Data-parallel processing devices handle multiple elements simultaneously. This increases the processing capacity to an otherwise impossible extent. The performance gains can be high, especially for massively parallel processing units that handle thousands of tasks simultaneously. However, to achieve the desired benefit, an effective distribution of work is essential. Otherwise processing units may run empty while others operate. Especially with processors that support fine-grained parallelism, such as those supporting multi-element instructions, handling these asymmetries is a challenge.

1.3 Contributions and Outline

This thesis focuses on query compilers for modern hardware platforms. We present a number of analyses and techniques that address the previously motivated open challenges. The contributions and correspondingly the chapters of this thesis are structured according to the three components *compilation* (Chapter 2), *communication* (Chapter 3), and *processing* (Chapter 4). Together the presented techniques provide the basis for building highly efficient query compilers.

The thesis starts in Chapter 2 with a compilation technique that is designed for database workloads. The technique was first introduced in

[29] H. Funke, J. Mühlig, and J. Teubner. Efficient generation of machine code for query compilers. In *DaMoN workshop*, pages 1–7, 2020

and includes a new intermediate representation that enables tailoring of the machine code generation steps used by query compilers to the demands database workloads. We show that this simplifies the compilation process to the benefit of low JIT-compilation times. A follow-up publication

[33] H. Funke and J. Teubner. Low-Latency Compilation of SQL Queries to Machine Code. *PVLDB*, 14(12):2691–2694, 2021.

showcases the compilation of SQL statements to machine code in a more comprehensive setting. We use this as basis for an in-depth evaluation of compilation and execution performance. Finally

[30] H. Funke, J. Mühlig, and J. Teubner. Low latency query compilation. *The VLDB Journal*, 2022

extends the compilation approach to additional use cases and shows how it can be used in a comprehensive system.

Chapter 3 contains a thorough analysis of the communication behavior of coprocessor systems during query execution. We show that existing processing techniques are affected by bandwidth limitations on multiple levels and that compilation techniques can help to alleviate them. However, massively parallel coprocessors typically use processing techniques with multiple passes over the input. We introduce a new technique that was published along with the communication analysis in

[28] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. Pipelined Query Processing in Coprocessor Environments. In *SIGMOD*, pages 1603–1618. ACM, 2018.

The technique only uses a single pass over the input relations and we show experimentally that this leads to a distinct reduction of the bandwidth demand to the benefit of efficient communication.

Chapter 4 addresses the problem of imbalances during compiled execution on data-parallel architectures. We present two balancing techniques that integrate with the single-pass processing approach [28]. The techniques restore processing efficiency when imbalances otherwise leave processing units unused. This enables effective use of the large processing capacities of parallel coprocessors even for workoads that are challenging to parallelize. The techniques were published in

[31] H. Funke and J. Teubner. Data-parallel query processing on non-uniform data. *PVLDB*, 13(6):884–897, 2020

and the follow-up publication

[32] H. Funke and J. Teubner. Like water and oil: with a proper emulsifier, query compilation and data parallelism will mix well. *PVLDB*, 13(12):2849–2852, 2020

presents a monitoring tool that helps to better understand the balancing effects. We conclude the chapter with an experimental analysis that shows the high processing speeds that can be achieved with our techniques. Finally Chapter 5 wraps up the thesis. In the development of our techniques, we emphasized the practical feasibility and built several prototype systems. To leverage the insight from these projects, we add a discussion on the engineering aspects in each chapter.

Additional Work During the writing and prior, the author contributed to other publications that are not part of this thesis. This work includes the following:

[18] S. Breß, H. Funke, and J. Teubner. Robust query processing in co-processor-accelerated databases. In *SIGMOD*, pages 1891–1906. ACM, 2016.

[19] S. Breß, B. Köcher, H. Funke, S. Zeuch, T. Rabl, and V. Markl. Generating custom code for efficient query execution on heterogeneous processors. *The VLDB Journal*, 27(6):797–822, 2018.

[70] S. Noll, H. Funke, and J. Teubner. Energy efficiency in main-memory databases. *Datenbank-Spektrum*, 17(3):223–232, 2017.

The Author's Contributions According to §10 (2) of the doctoral regulations of the computer science dapartment at TU Dortmund University from August 29, 2011, the author should indicate their own contributions to the results of collaborations that are used. The author is principal author of the articles [28, 29, 31, 32, 33] and of all contents from the articles that are used in chapters of this thesis. He is responsible for the concepts, the implementations, the presentation, and the analyses.

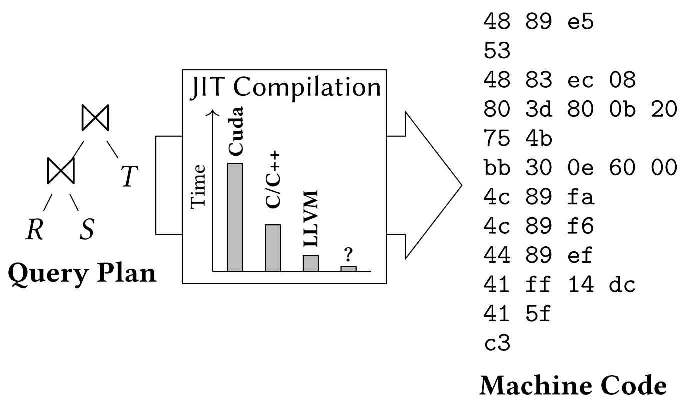
2 Compilation

By compiling queries to machine code, query compilation enables very high processing speeds and reduces the bandwidth demands during processing to a bare minimum. This makes the technique very attractive for modern in-memory systems. The technique, however, introduces *compilation* as an additional step to query processing. This causes a compilation overhead that adds to the response times of queries. The overhead is relatively high for queries with a short execution time and for queries with high complexity.

This chapter introduces *Flounder IR*, a new lightweight intermediate representation for query compilation to address the problem of compilation overheads. *Flounder IR* is close to machine assembly and adds just that set of features that is necessary for efficient query compilation: virtual registers and function calls ease the construction of the compiler front-end; database-specific extensions enable efficient pipelining in query plans; more elaborate IR features are intentionally left out to maximize compilation speed.

We present and motivate *Flounder IR* and showcase its capabilities with the prototype system *ReSQL* that uses the IR for compilation. *ReSQL* employs similar compilation techniques as existing query compilers, but leverages the capabilities of *Flounder IR* and the *Flounder* library to achieve low compilation times. We demonstrate with micro-benchmarks and for real world workloads that our approach significantly reduces query compilation times. We show reductions in compilation times up to two orders of magnitude over LLVM and show improvements in overall execution time for TPC-H queries up to 5.5× over state-of-the-art systems.

Parts of this chapter are contained in published articles [29, 33, 30].



```
48 89 e5
53
48 83 ec 08
80 3d 80 0b 20
75 4b
bb 30 0e 60 00
4c 89 fa
4c 89 f6
44 89 ef
41 ff 14 dc
41 5f
c3
```

Figure 2.1: Effect of different intermediate representation levels on JIT query processing performance.

2.1 Introduction

Query compilation is a technique for query execution with extremely high efficiency. It uses *just-in-time* (JIT) compilation to generate custom machine code for the execution of every query. The approach leverages a compiler stack that first translates the query from a relational query plan to an *intermediate representation* (IR), and then from the IR to *native machine code* for the target machine. The execution-efficiency of the compiled code is very high compared to standard interpretation-based backends. However, by using compilation the technique adds a step to query execution, which introduces translation cost. Especially short-running queries and queries with high complexity experience a relatively high translation cost, which ultimately extends query response times.

When using query compilation for queries on smaller datasets, the relative cost of compilation increases. The query engine spends most of its time on compilation before entering execution only for a very short time. Further, complex queries can have particularly long compilation times due to complexity of algorithms used in JIT machine code translation [80]. Approaches to mitigate the impact of compilation time on response time have been proposed previously [53]. However, these typically rely on *both* an interpretation-based and a compilation-based backend at a high implementation cost.

2.1.1 Intermediate Representation Levels

The intermediate representation is an important design choice for query compilers. Figure 2.1 illustrates the effect of the IR choice on JIT compile times. Query compilers with high-level IRs, such as C/C++ [51, 89, 25] or OpenCL and Cuda [19, 31, 28, 79] generally have longer compilation times than query compilers that generate lower-level IRs such as LLVM IR [69, 74]. Existing work on JIT compilers, however, shows the feasibility of much shorter compile times [6, 4] than those of LLVM. In fact non-database JIT compilers reach break-even points for dynamic compilation versus static compilation already for thousands of records [6]. By contrast, state-of-the-art LLVM-based query compilers have compilation times of tens of milliseconds [69], which is sufficient time to process queries on millions of tuples [16]. This raises the question illustrated by the bar '**?**' in Figure 2.1: How can such short compilation times be adopted for database systems that perform query compilation?

LLVM IR is general purpose and was designed to serve as backend for the translation of high-level language features [55]. Being general purpose, LLVM is relatively heavyweight and devises a translation stack that is "overkill" for relational workloads. The code for relational queries typically consists of tight loops with conditional code mainly to drop non-qualifying tuples. This plain structure offers potential for much simpler translation techniques than those used by general purpose translators, which leverage complex code analysis and register allocation algorithms.

2.1.2 Contributions

In this work, we present the intermediate representation *Flounder IR* and the *ReSQL* database system, which represent our new approach to query compilation with low compilation latencies.

Flounder IR We propose Flounder IR as a lightweight domain-specific IR that is designed for fast compilation of database workloads. Flounder IR is based on machine assembly and adds several features for efficient use by query compilers: virtual registers enable efficient handling of attribute data; function calls allow interfacing with the database library; database-specific register allocation enables efficient pipelining. The IR features are designed to be lightweight and we avoid the use of more elaborate features to allow for fast translation. Along with the IR, we show the techniques for translation of Flounder IR to machine code used by the Flounder library.

ReSQL The ReSQL database system was developed as a showcase for low-latency query compilation with Flounder IR. ReSQL provides a full translation stack from SQL to machine code and supports a variety of queries. We discuss the interaction of ReSQL's translation components with Flounder IR and use the system to perform an experimental evaluation on TPC-H benchmark [13] workloads. The analysis shows that our query compilation approach reduces compilation times while preserving high processing speeds. We show with speedups up to 5.5× over a state-of-the-art LLVM-based query compiler, Hyper, that our approach achieves better tradeoffs between compilation and execution time.

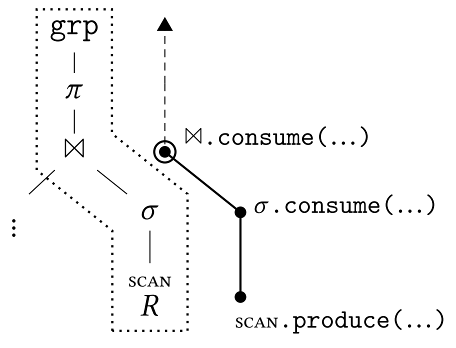


Figure 2.2: Translation of the probe-side pipeline of a query plan.

2.1.3 Outline

This chapter is structured as follows: The next Section 2.2, illustrates how query compilers use Flounder IR for query translation. Section 2.3 then details the design of Flounder IR. Section 2.4 shows the translation of Flounder IR to machine code. Section 2.5 discusses further improvements and applications of our approach. Section 2.6 evaluates the approach experimentally and Section 2.7 discusses future work. In Section 2.8, we discuss the engineering aspects of the techniques and finally Section 2.9 wraps-up the chapter with a summary.

2.2 Query Translation

Query compilation typically involves one step that translates relational queries to an *intermediate representation* (IR) and another step that translates the IR to machine code. In the following, we give an overview of how both steps are realized for query compilation with our intermediate representation Flounder IR.

2.2.1 Query Plan to IR

The first translation step traverses the query plan and builds an intermediate representation of the query functionality. A common way to do this is the produce/consume model [69], which emits code for operator functionality either in `produce` or `consume` methods. We call these methods *operator emitters*. Figure 2.2 illustrates the operator emitters that are executed during translation of a sample query. The *build pipeline* on the left of the join populates the join hash table. It was translated previously. The *probe pipeline*, surrounded by the dotted line, accesses the join hash table. We look at its translation in detail.

The code to scan *R* was already emitted by SCAN`.produce(...)`. It contains a loop that iterates over the table *R* and reads its tuples. The code for selection was emitted by σ`.consume(...)` and now the hash join follows with $\bowtie$`.consume(...)`. The implementation of the method is shown in Figure 2.3, which uses a notation

```
TRANSLATE HASH JOIN OPERATOR TO IR
Function ⋈.consume(attributes, caller):
 1  if caller is ⋈.left:                                   /* build-side */
 2      ht ← createHashtable(...)
 3      emit entry ← ht_ins (ht, ⋈.buildKey)               /* get bucket */
 4      emit materialize (entry, attributes)               /* write to ht */
 5      $a_l$ ← attributes
 6  if caller is ⋈.right:                                  /* probe-side */
 7      emit entry ← null                                  /* initialize */
 8      emit while (true):                     /* loop over join matches */
            /* probe hash table to get next matching entry */
 9          emit entry ← ht_get (ht, ⋈.probeKey, entry)
10          emit if entry is null:                       /* check result */
11              emit break                              /* no more match */
12          emit dematerialize (entry, $a_l$)           /* read to regs */
13          ⋈.parent.consume ($a_l$ ∪ attributes, ⋈)       /* next ops */
```

Figure 2.3: Operator emitter of the hash join operator. We underlined the functionality that is placed in the JIT query.

similar to Kersten et al. [49]. Code lines following an **emit** statement are underlined to emphasize that this code is not executed immediately but instead placed in the JIT query. For instance `createHashtable(..)` is not underlined (line 2) and is therefore executed during translation. By contrast `ht_ins(..)` is underlined (line 3) and is therefore placed in the compiled code. This leads to repeated execution of the line for every tuple of the scanned table.

In the example `⋈.consume(...)` is called from its right child and therefore the probe-side code is produced (lines 7–13). The code first initializes the variable *entry*, which holds hash probe results (line 7) and then loops over the hash join matches (lines 8–13). In the loop, we first call `ht_get(...)` to retrieve the next match (line 9) and then perform a check to exit when no more matches exist (lines 10–11). To process *join matches*, we read the attributes of the match to registers (line 12) and then the join's parent operators place their code by calling `consume(...)` (line 13).

The resulting intermediate representation is shown in Figure 2.4 (a)[1]. It performs the described probe functionality. We briefly describe the resulting IR here and provide a detailed description of the used Flounder IR features in Section 2.3.

The attribute values are held in `{r_a}`, `{s_a}`, and `{s_b}` and the locations of hash table entries in `{entry}`. The `hash_get(...)` call is realized with `mcall` and the loop over the probe matches with a combination of compare (`cmp`) and two

[1]We use an `nasm`-style assembler notation with destination operand on the left and source operand on the right.

```
 [...] ;child code
 vreg {entry}
 mov {entry}, 0
;while head
loop_headN:
 ;ht_get(..) call
 mcall {entry},{ht_get},
   {ht},{r_a},{entry}
 ;break when entry=NULL
 cmp {entry}, 0
 je loop_footN
 ;dematerialize ht entry
 vreg {s_a}
 vreg {s_b}
 mov {s_a}, [{entry}]
 mov {s_b}, [{entry}+8]
 [...] ;parent.consume(..)
 clear {s_a}
 clear {s_b}
 ;loop foot
 jmp loop_headN
loop_footN:
 clear {entry}
 [...] ;child code
```

Flounder IR
(in-memory)
(a)

```
 [...] ;child code
 mov  r11, 0; init entry
loop_headN: ;while head
 mov  [rsp-8],  r8 ;caller-
 mov  [rsp-16], r9 ;save
 mov  [rsp-24], r10
 mov  rdi, 0x25cac0 ;call
 mov  rsi, r9       ;params
 mov  rdx, r11
 sub  rsp, 24 ;adjust stack
 mov  rax, 0x42fa10
 call rax ;ht_get call
 add  rsp, 24 ;restore stack
 mov  r8,  [rsp-8] ;restore
 mov  r9,  [rsp-16] ;caller-
 mov  r10, [rsp-24] ;save
 mov  r11, rax ;return value
 cmp  r11, 0 ;break condition
 je   loop_footN
 mov  r12, [r11]   ;demate-
 mov  r13, [r11+8] ;rialize
 [...] ;parent.consume(..)
 jmp  loop_headN ;next probe
loop_footN:
 [...] ;child code
```

`x86_64` **assembly**
(in-memory)
(b)

Figure 2.4: Intermediate representation of hash join probe functionality (a) and corresponding machine assembly (b).

jumps (`jmp`, `je`). To read attributes from a hash table entry (dematerialize), we use `mov` from a memory location in brackets `[]` to e.g. `{s_a}`.

2.2.2 IR to Machine Code

The next step translates the query's intermediate representation to machine code. The machine code needs to follow the application binary interface (ABI) of the execution platform. In this work, we use the target architecture x86_64 [61].

The Flounder IR emitted by the hash join is translated to the machine assembly shown in Figure 2.4 (b). Several abstractions that were used during IR generation are now replaced by machine-level concepts. E.g. the machine assembly uses processor registers such as `r12` instead of `{s_a}`. Further, the machine assembly uses additional `mov` instructions to transfer values between registers and the stack, e.g. `mov r8,[rsp-8]`. The translation process from Flounder IR to machine code needs to manage machine resources such as registers and stack memory and find

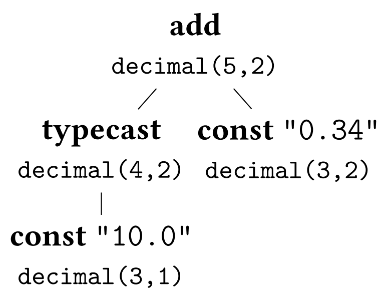


Figure 2.5: Typed expression tree for the expression `10.0` + `0.34`.

an efficient way for their use during JIT query execution.

2.2.3 ReSQL Translation Mechanisms

For a more comprehensive picture, we present two more essential translation mechanisms that are used for the translation of queries to Flounder IR by ReSQL. We first discuss the translation of scalar expressions, which are used in SQL statements, e.g. for selection or join criteria. Then we discuss handling of tuples in the implementation of operator emitters.

Expression Translation To illustrate expression translation, we use the expression `10.0` + `0.34`, a sum of two decimal constants, as example. ReSQL uses 64 bit integers for decimal arithmetics and thus represents the values as `100` and `34` along with the *base* and *precision*. The precision is the number of digits in total and base is the number of digits following the decimal point.

For JIT-based evaluation, the expression translator performs two steps. The first step is type resolution, a standard procedure that derives the result type of each expression node. The leaf types `decimal(3,1)` and `decimal(3,2)` are given by the constants. The expression translator applies type rules to derive the typed expression tree shown in Figure 2.5. One typecast was inserted to maintain the same base for the add. Then the second step emits Flounder IR for the expression tree. Starting with the leaf expressions, code for the evaluation of each node is emitted. The resulting Flounder IR to evaluate the expression is shown in Figure 2.6. The code uses, e.g. `vreg {x}` and `clear {x}` to indicate the validity range of `{x}`. First both constants are loaded. Then the typecast for `{dec_const1}` is evaluated by multiplying with `10`. Finally the add is evaluated and the result is stored in `{add_res0}`. The IR-code is inserted into the code frame of the query and translated to machine code along with the query.

Handling of Tuples In JIT-based execution, the individual values of a tuple are distributed across registers. For the implementation of operator emitters, however, it is still useful to handle tuples as a single entity [49]. ReSQL provides several

```
;const "0.34"
vreg   {dec_const0}
mov    {dec_const0}, 34
;const "10.0"
vreg   {dec_const1}
mov    {dec_const1}, 100
;typecast [decimal(3,1) to decimal(4.2)]
vreg   {cast_res0}
mov    {cast_res0},  {dec_const1}
clear  {dec_const1}
imul   {cast_res0},  10
;add
vreg   {add_res0}
mov    {add_res0},   {dec_const0}
clear  {dec_const0}
add    {add_res0},   {cast_res0}
clear  {cast_res0}
;[...] work with add_res0
clear  {add_res0}
```

Figure 2.6: Flounder IR produced for the expression `10.0 + 0.34`.

`tup = Values::evaluate(expr);`
Evaluate the list of expressions `expr`.

`tup = Values::dematerialize(loc, schm);`
Scan a tuple with schema `schm` from location `loc`.

`hash = Values::hash (tup);`
Hash the tuple `tup`.

`flag = Values::checkEquality(tup1, tup2);`
Check tuples `tup1` and `tup2` for equality.

`Values::materialize(tup, loc);`
Write tuple `tup` to location `loc`.

Figure 2.7: Tuple-based code generation methods allow us to handle lists of attribute registers as if they were coherent tuples.

code generation functions in the `Values` namespace for this purpose. These are shown in Figure 2.7. To evaluate the projection expressions from a `select`-clause, for example, we use `tup=Values::evaluate(projs)`. The result `tup` is a list of virtual registers that hold the expression results, ultimately a tuple. Similarly, lists of virtual registers are used to hold tuples after scanning them or when applying a hash function.

2.3 Lightweight Abstractions

Flounder IR is similar to x86_64 assembly, but it adds several *lightweight abstractions*. The abstractions are designed with the interface to the query compiler *and* with the resulting machine code in mind. In this way, Flounder IR passes just the right set of information into the compilation process. For operator emitters, the IR provides independence of machine-level concepts, which allows similar code generation as is typically performed with LLVM. For translation to machine code, the abstractions are sufficiently lightweight to avoid the use of compute-intensive algorithms. Additionally, the IR contains information about the relational workload that enables efficient tuning of the machine code.

In the following, we present the lightweight abstractions. They add several pseudo-instructions, i.e. `vreg`, `clear`, and `mcall` to x86_64 assembly and use additional tokens, which are shown in braces, e.g. {param1}.

2.3.1 Virtual Registers

An unbounded number of *virtual registers* is a common abstraction in compilers [9]. Query compilers use them to handle attributes without the restrictions of machine registers. When replacing virtual registers with machine registers for execution, general purpose compilers perform live-range analysis [2]. This is rather expensive because compilers consider all execution-paths that lead to a register usage.

Query workloads use virtual registers in a much simpler way than general purpose code. They hold attribute data within a pipeline and the pipeline's execution path only consists of tight loops. This allows query compilers to use a simpler approach that skips live-range analysis. In Flounder IR, operator emitters mark the validity range of virtual registers. The `vreg` pseudo-instruction marks the start of a virtual register usage, e.g. by using

```
;start virtual register use
vreg {vreg_nameN}
```

and the `clear` pseudo-instruction marks the end of the usage, e.g. with

```
;finish virtual register use
clear {vreg_nameN} .
```

We use these markers in a way similar to scopes in higher-level languages. For instance the Flounder IR in Figure 2.4 (a) marks the range of the probe attributes {s_a} and {s_b} to reach around the operators that are contained in the probe loop.

2.3.2 Function Calls

Being able to access pre-compiled functionality is important for query compilers. It reduces compile times and avoids the implementation cost of code generation for every SQL feature. To this end Flounder IR provides the mcall pseudo-instructions to specify function calls in a simple way. For instance

```
;function call to ht_ins
mcall {res} {ht_ins} {param1} ... {paramN}
```

represents a function call to ht_ins(...) with parameters param1 to paramN and the return value is stored in {res}. A pointer to the function code is provided as an address constant via {ht_ins}. This pseudo-instruction is later replaced with an instruction sequence that realizes the calling convention.

2.3.3 Constant Loads

Large constants, e.g. 64 bit, can not be used as *immediate operands* (imm) on current architectures. To use large constants, they have to be placed in machine registers. The *constant load* abstraction in Flounder IR, allows using such constants without restrictions. For instance the following instruction

```
;load from 64 bit address with offset
mov {attr} [{0x7fff5a8e39d8} + {offs}]
```

loads data from the address {0x7fff5a8e39d8}+{offs} to the virtual register {attr}. During translation to machine assembly, the address constant will be placed in a machine register.

2.3.4 Transparent High-Level Constructs

We use *transparent high-level constructs* that mimic high-level language features such as loops and conditional clauses. They are used to generate Flounder IR instructions in operator emitters. For example operator emitters can generate a while loop with the condition {tid} < {len} by using the methods While(...), close(...), and isSmaller(...) as shown below on the left.

```
// Produce code for while loop (C++)
wl = While(isSmaller(tid,len)); {
    [...]
} wl.close();
```

$\longrightarrow$

```
loop_headN:
 cmp {tid},{len}
 jge loop_footN
 ;loop body
 [...]
 jmp loop_headN;
loop_footN:
 ;after loop
 [...]
```

This generates the Flounder IR code shown on the right, that realizes the loop functionality. The start of the loop is marked with the label loop_headN. The cmp instruction then evaluates the loop condition and jge jumps to the loop_footN-label at the loop end, if the condition evaluates to false. Otherwise, the loop body is executed and after it, the loop starts over by executing the jump instruction jmp loop_headN, which redirects control flow to the loop head.

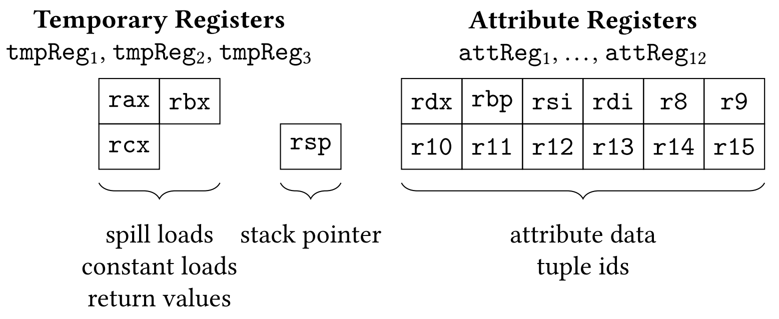


Figure 2.8: Usage of machine registers by the translator.

2.4 Machine Code Translation

In the translation from Flounder IR to `x86_64` machine code, the abstractions that facilitated code generation in the previous step are now replaced with machine concepts. A key challenge here is to replace virtual registers with machine registers and to manage spill memory locations for cases of insufficient registers. Finding optimal register allocations is an NP-hard problem and even the computation of approximations is expensive [21]. In the context of JIT compilers, linear scan has been proposed as a faster algorithm [80] and was adopted by LLVM. However, linear scan register allocation is still relatively expensive due to live range computations and increasing numbers of registers.

The following presents a much simpler technique that benefits from the explicit usage ranges marked in Flounder IR. We first show the machine register configuration used by the translator and then show the algorithm for translation of the lightweight abstractions.

2.4.1 Register Layout

We use a specific register layout for the machine code generated from Flounder IR. The layout is shown in Figure 2.8. We split the 16 integer registers of the `x86_64` architecture into three categories.

We use twelve attribute registers $\texttt{attReg}_1, \ldots, \texttt{attReg}_{12}$ to carry attribute data and tuple ids. We use three temporary registers `tmpReg1`, `tmpReg2` and `tmpReg3`, which are-multi purpose for accessing spill registers and constant loads. Lastly, we use the stack pointer `rsp` to store the stack offset. The stack base pointer `rbp` is repurposed for attribute data and not used for the stack.

2.4.2 Translation Algorithm

The translation algorithm translates Flounder IR to `x86_64` assembly in one sequential pass over the code. It replaces the Flounder abstractions with machine

```
TRANSLATE FLOUNDER IR TO MACHINE ASSEMBLY
1  $a \leftarrow 0$                          /* attribute registers in use */
2  foreach instruction $i$ in input:
3      $t \leftarrow 0$                      /* temporary registers in use */
4      if $i$ is vreg {$v$}:          /* allocate pseudo-instruction */
5          if $a$ < number attribute registers:
6              allocate free attReg$_k$            /* machine register */
7              $a \leftarrow a + 1$
8          else allocate spill location                     /* spill */
9      elseif $i$ is clear {$v$}:  /* deallocate pseudo-instruction */
10         if any attReg$_k$ holds $v$:
11             release attReg$_k$                /* free machine reg */
12             $a \leftarrow a - 1$
13     elseif $i$ is mcall (...): /* function call pseudo-instr.  */
14         emit call-convention code
15     else:                              /* other instructions */
16         foreach virtual register operand $v$ in $i$:
17             if $v$ is spilled:
18                 emit spill code for $v$ to tmpReg$_t$          /* spilled */
19                 replace $v$ with tmpReg$_t$
20                 $t \leftarrow t + 1$
21             else replace $v$ with attReg$_k$      /* machine register */
22         foreach constant load operand $c$ in $i$:
23             emit load $c$ to tmpReg$_t$        /* place c in temp reg */
24             replace $c$ with tmpReg$_t$ in $i$
25             $t \leftarrow t + 1$
26         emit $i$                   /* output native instruction */
```

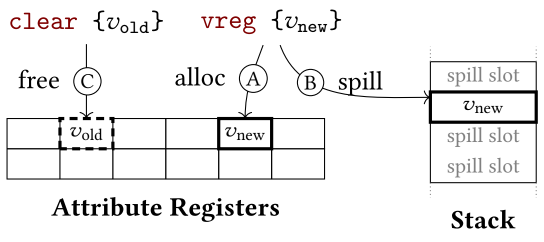
Figure 2.9: Pseudocode for the translation of Flounder IR to machine assembly. The code is translated in one pass.

instructions, machine registers, and stack access. The algorithm is shown in Figure 2.9.

When iterating over the IR elements, the algorithm keeps track of a, the number of in-use attribute registers (line 1), and t, the number of temporary registers per instruction (line 3). We describe the translation in three parts. The first part is register allocation, then the replacement of virtual operands with machine operands in instructions, and finally function calls.

Register Allocation Register allocation is used to decide which virtual registers are stored in machine registers and which virtual registers are stored on the stack.

Register allocation does not produce code directly, but it sets the allocation state for spill code and operand replacement. The procedure is illustrated below.



When a vreg {v_{new}} pseudo-instruction is encountered (line 4), there are two options. In case Ⓐ there are sufficient machine registers available and we assign one of them to v_{new} (lines 5-7). In case Ⓑ all machine registers are occupied and we assign a spill slot on the stack (line 8). For vreg {v_{old}}, illustrated by Ⓒ, any machine registers assigned to v_{old} are freed (line 11).

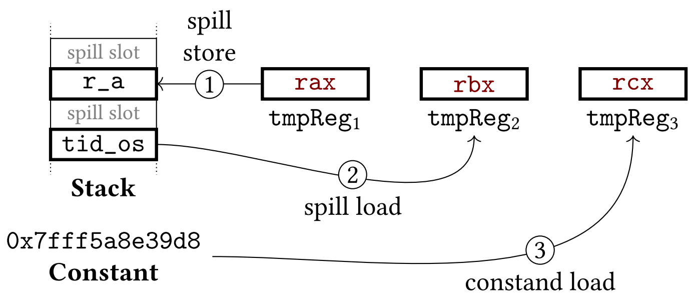
This assignment procedure has the effect that spilled virtual registers remain spilled. However, this happens only when the pipeline requires to hold more than 12 attributes simultaneously. As query compilers typically choose pipeline boundaries such that the data volume per tuple fits into the processor registers, this technique is a perfect match for query compilation.

Spill Code and Operand Replacement For each instruction, operands that use *constant loads* or *virtual registers* have to be replaced with machine-compatible operands. Virtual registers that were assigned with machine registers are simply swapped (line 21). For the other cases, the algorithm uses $\texttt{tmpReg}_1$ to $\texttt{tmpReg}_3$ to hold values temporarily per instruction. Three registers are sufficient for this purpose as this is the highest numner of non-immediate operands per instruction. As an example, we look at the following instruction.

```
mov {r_a}, [{0x7fff5a8e39d8}+{tid_os}]
```

It reads an 8 byte value with the offset `{tid_os}` from the memory address `0x7f...` and stores it in `{r_a}`. The address is too large for an immediate operand and we assume for illustration purposes that both virtual registers `{r_a}` and `{tid_os}` are spilled.

The translator assigns temporary registers to each operand and emits *spill code* that exchanges values between spill slots and temporary registers. This is performed in pseudocode lines 16–26 and illustrated below.



The algorithm enumerates the virtual register accesses (lines 16-21) and the constant loads (lines 22-25) from the instruction. It assigns one of the temporary registers tmpReg$_1$ to tmpReg$_3$ to each. In step ① the translator assigns tmpReg$_1$ (rax) to the operand {r_a}. This is the only output operand of the instruction and the operator emits a store to {r_a}'s spill slot on the stack. Step ② assigns tmpReg$_2$ (rbx) to the operand {tid_os}. The translator emits a load to retrieve the value from its spill slot. Step ③ assigns tmpReg$_3$ (rcx) to the constant load of address 0x7f... . The translator emits a load for the constant. This results in the following machine code sequence, which includes the original mov instruction with replaced operands.

```
mov rbx, [rsp-24]       ;load spill tid_os
mov rcx, 0x7fff5a8e39d8 ;load constant
mov rax, [rcx+rbx]      ;instruction
mov [rsp-8], rax        ;store spill r_a
```

Calling Conventions During translation the mcall IR-instruction is replaced with a machine code sequence that performs the function call. To this end, a *calling convention* is applied, which specifies rules for the execution of function calls on a given hardware platform. It specifies the way registers are preserved across the call, how parameters are passed, and how the stack frame is adjusted. For the x86_64 calling convention, the calling function preserves up to 7 integer registers (*caller-save registers*) and passes up to 6 parameters in integer registers before using the stack for parameter passing [61].

The call translation is initiated in line 14 of the Flounder IR translation algorithm (Figure 2.9). The machine register allocation to the point of the call is known. This allows us to generate a call sequence that is tailored to the current register usage.

The mcall translation algorithm is specified in Figure 2.10 and explained in the following. We use the call to ht_get(..) from a previous example (Figure 2.4).

```
mcall {entry}, {ht_get}, {ht}, {r_a}, {entry}
```

It has the return value {entry}, the function address {ht_get}, and the parameters {ht}, {r_a} and {entry}. To derive the call-convention instruction sequence, the translator first replaces these operands with the already allocated machine operands (lines 1–3).

```
TRANSLATE mcall ret, func, $p_0$, ..., $p_n$
1  foreach $p$ in {ret, $p_0$,...,$p_n$}: /* replace virtual registers */
2      if $p$ is virtual register:        /* and use machine operands */
3          replace $p$ with attribute register or stack location
4  $R_{\text{caller-save}}$ = {rsi, rdi, r8, r9, r10, r11}    /* A caller-save */
5  foreach register $r$ in $R_{\text{caller-save}}$
6      if $r$ is allocated:                            /* check use */
7          emit save $r$ to stack
8  $R_{\text{param}}$ = {rdi, rsi, rdx, rcx, r8, r9}     /* B set parameters */
9  foreach parameter $p_i$ in $p_0$, ..., $p_n$:
10     src ← $p_i$
11     if $p_i$ was overwritten:                  /* handle overwrites */
12         src ← stack backup of $p_i$
13     emit mov $R_{\text{param}_i}$, src
14 stackOffset ← total stack usage           /* C boilerplate call */
15 emit sub rsp, stackOffset
16 emit mov rax, func
17 emit call rax
18 emit add rsp, stackOffset
19 foreach register $r$ in $R_{\text{caller-save}}$:  /* D restore caller-save */
20     if $r$ is allocated:
21         emit restore $r$ from stack
22 emit mov ret, rax                        /* get return value (C) */
```

Figure 2.10: Translate `mcall` IR-instruction to call-convention code.

```
mcall r11, 0x42fa10, 0x25cac0, r9, r11
```

Then the translator generates code that performs the following four steps:

A Save caller-save registers that are in-use on the stack. These are `r8`, `r9`, `r10` in the example (lines 4-6).

B Assign parameter registers in the order specified by the ABI (lines 7-12). We assign `0x25cac0` to `rdi`, `r9` to `rsi`, and `r11` to `rdx`.

C Place boiler-plate code to modify the stack frame, jump into the function, and to retrieve the return value (lines 13-17,21).

D Restore caller-save registers (lines 18-20).

This results in the instruction sequence shown in Figure 2.11 that realizes the call in machine assembly. The instructions are annotated with **A** to **D** to indicate the step that generated them.

```
mov  [rsp-8],  r8     ;A save caller-save
mov  [rsp-16], r9
mov  [rsp-24], r10
mov  rdi, 0x25cac0    ;B assign parameters
mov  rsi, r9
mov  rdx, r11
sub  rsp, 24          ;C boilerplate call
mov  rax, 0x42fa10
call rax
add  rsp, 24
mov  r8,  [rsp-8]     ;D caller-save restore
mov  r9,  [rsp-16]
mov  r10, [rsp-24]
mov  r11, rax         ;(C get return value)
```

Figure 2.11: Instruction sequence for the example function call.

2.5 Getting More Out of Flounder

Flounder IR is a near-hardware representation for database processing functionality. This property enables additional uses and benefits for the IR. We present ideas on taking the IR's database specialization further by adding additional domain knowledge to the language. Then we show prefetching as an example of utilizing such domain knowledge. Finally, we discuss the use of Flounder IR as compilation vehicle for higher-level IRs.

2.5.1 Utilizing Additional Database Knowledge

The domain specialization makes Flounder IR receptive to utilizing particular *database knowledge*. This idea can be extended in the way Flounder IR uses types. Currently it only uses machine datatypes. Alternatively, we can add SQL types to the IR. This simplifies the translation from SQL to Flounder because operator translators can directly emit instructions on SQL types. At the same time the responsibility of implementing SQL types and their special type characteristics moves down one level to the IR translation. This may open up interesting new ways for handling `NULL`-logic or types with multi-register representations (e.g. 128 bit decimals). The translator has the opportunity to apply simpler or unified logic to handle such characteristics.

Many database operators have optimized implementations that leverage hardware features, e.g. sort and hash-based operators [8]. Specifically applying vectorization techniques (e.g. AVX) has proven to be beneficial [98]. Flounder IR is a good match for such techniques because it gives explicit control over the instructions that are used. This helps to clearly express the way hardware optimizations are applied, which can be difficult with high-level languages that abstract hardware details. Similar to passing specific implementation aspects, additional hints about the database or about database statistics may be used. For instance information about relation and tuple sizes can be leveraged by the compiler for loop unrolling and prefetching. Hints about predicate selectivities are beneficial in estimating which branches are likely to be taken.

2.5.2 Higher-Level IRs

Other IRs that describe data processing on a higher level than Flounder IR are frequently used. They are used as translation step for a specific query processing paradigm. For instance MonetDB uses MAL [14] for its column-style processing approach and SQLite uses a (high-level) bytecode representation for its bytecode interpreter [91]. Alternatively higher-level IRs can be used as an abstraction layer. As such they enable database systems to target different parallel hardware architectures [19, 79] or to handle multiple processing paradigms. The IR Voila [38], for instance, provides a representation that is suitable for compiled and interpreted execution. We take Voila's scatter operation as example to illustrate how Flounder can be leveraged for compiled execution of this IR. The scatter operation is used by hash-based operators to write values to the hash table. For example

```
// Voila scatter operation: Write key to HT
scatter ( ht.k1, new_pos |can_scatter, t[0] )
```

scatters the value `t[0]` to the hash table key location `k1` of the bucket `new_pos`. The scatter is executed conditionally depending on the flag `can_scatter`. Translation to Flounder IR can implemented as a operator emitter, similar to Section 2.2.1. The Voila operation translates a short sequence of Flounder instructions:

```
;Scatter op in Flounder IR
 cmp {can_scatter}, 0
 je  afterScatter
 mov [{new_pos}+4], {t0}
afterScatter:
```

The `cmp` and `je` instructions evaluate `{can_scatter}` to skip processing if necessary. Then `mov` performs the actual write of `{t0}` to the hash bucket with base address `{new_pos}` and an exemplary offset +4.

2.6 Evaluation

This section evaluates our approach of using a simple IR for query compilation that is specialized to relational workloads over using a general purpose IR. We use the micro prototype of a query compiler to evaluate the characteristics of different IR's along with their translation libraries. Then we use the ReSQL database system that was built on top of Flounder IR to evaluate the real world performance against other state-of-the-art systems.

Micro Prototype We use a smaller query compiler prototype that supports translation of query plans to *both* Flounder IR and LLVM IR. This allows us to evaluate the performance of both IRs on the same system. The prototype is used to execute the workloads from Figure 2.12. Flounder emits the binary representation of compiled queries with the `AsmJit` library [52] to avoid the overhead of running external assemblers, e.g. `nasm`. For LLVM IR, the machine code is generated by the LLVM library's JIT functionality. We use `O0` and `O3` optimization levels for tradeoffs between compilation time and code quality.

Database Systems We built the JIT-compiling database system ReSQL, which uses Flounder IR during compilation and has the ability to run various SQL queries. This allows us to evaluate the real world performance by executing TPC-H benchmark [13] queries. For comparison, we use one compilation-based system Hyper [69] and one interpretation-based system DuckDB [84]. We use Hyper version `v0.5-222`, which executes queries by JIT compiling via LLVM. We use DuckDB version `v0.2.5`, which executes queries with vector-at-a-time processing [16] for cache-efficiency. In its current development state, ReSQL only supports single-threaded execution. We configured all systems to run single-threaded for a fair comparison. Furthermore, ReSQL's query planner does not yet support sub-queries. Therefore we only use benchmark queries that do not contain sub-queries.

Design of Characteristic Workloads We use four query templates that allow us to evaluate different query characteristics. The templates are specified in Figure 2.12 in an SQL-form that uses additional integer parameters. The parameter l varies the data size in $\mathbf{Q}_{\text{📊}}$. Parameters p, j, and s vary query complexity in $\mathbf{Q}_{\pi}$, $\mathbf{Q}_{\bowtie}$, and $\mathbf{Q}_{\sigma}$ respectively. The attribute data is generated from uniform random distributions with the following relation sizes: $\mathbf{Q}_{\text{📊}}$ has l tuples for r an s, $\mathbf{Q}_{\pi}$ has 1 M tuples, $\mathbf{Q}_{\bowtie}$ has 10 K tuples per join relation, and $\mathbf{Q}_{\sigma}$ has 1 M tuples.

Execution Platform We use a system with Intel(R) Xeon E5-1607 v2 CPU with 3.00 GHz and 32 GB main memory. The experiments run in one thread. We use operating system Ubuntu 18.04.4 and clang++ 6.0.0 to compile the query compiler and the library for JIT queries. The LLVM backend uses LLVM 6.0.0.

2.6.1 Compilation Times

We compare the machine code compilation times for LLVM and Flounder for $\mathbf{Q}_{\pi}$ and $\mathbf{Q}_{\bowtie}$. We use $\mathbf{Q}_{\pi}$ with values of p to project 50 to an extreme case 500 attributes (filter with selectivity 1%). We use $\mathbf{Q}_{\bowtie}$ with values of j to join 2 to 100 relations. We show the results for **Flounder**, **llvm-O0**, and **llvm-O3** in Figure 2.13.

Observations For all techniques, the compilation times increase with the query complexity. The compilation times for $\mathbf{Q}_{\bowtie}$ are higher (up to 657 ms) than for $\mathbf{Q}_{\pi}$(up

```
SELECT AVG(r.e)
  FROM r,s --len(r)=len(s)=l
 WHERE r.b = s.d
   AND r.c BETWEEN 40 AND 50
```

$\mathbf{Q}_{\bowtie 0}$: Vary relation lengths (l).

```
SELECT r.a1, r.a2, ..., r.ap
  FROM r
 WHERE r.a1 < c
```

$\mathbf{Q}_{\pi}$: Vary projection complexity (p).

```
SELECT r1.a, r2.a, ..., rj.a
  FROM r1, r2, ..., rj
 WHERE r1.a = r2.a
    ...
   AND rj-1.a = rj.a
```

$\mathbf{Q}_{\bowtie}$: Vary join complexity (j).

```
SELECT r.a
  FROM r
 WHERE r.a != c1
   AND r.a != c2
    ...
   AND r.a != cs
```

$\mathbf{Q}_{\sigma}$: Vary selection complexity (s).

Figure 2.12: Query templates used to vary query characteristics.

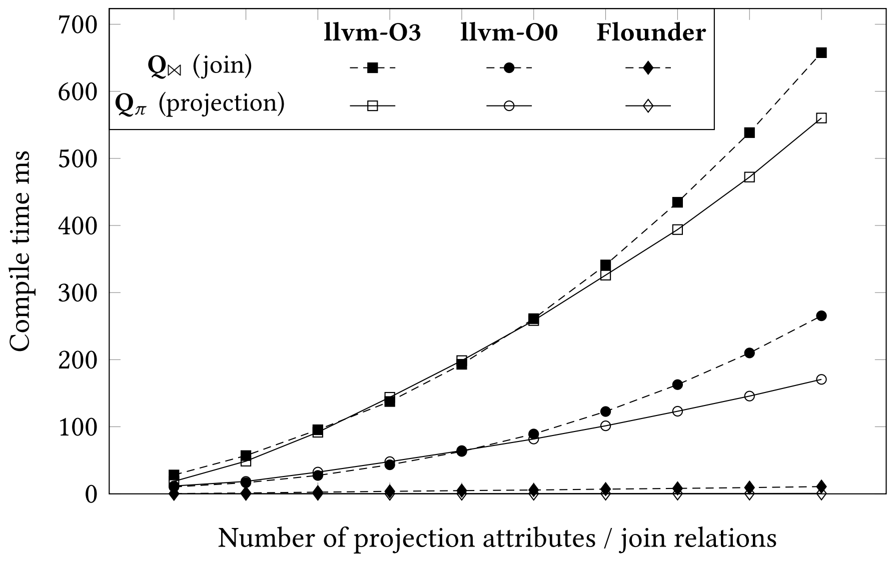


Figure 2.13: Effect of query complexity on compilation times for different query compilation techniques.

to 560 ms) and we look in detail at $\mathbf{Q}_{\bowtie}$. With O0 optimization LLVM has compilation times between 10 ms up to 265 ms. With O3 compilation times range from 28 ms up to 657 ms. For both levels, the graphs show super-linear growth of compilation times with query complexity. **Flounder** shows lower compilation times that scale linearly between 0.3 ms to 10.8 ms. The highest factor of improvement is 24.6x over **llvm-O0**. and 60.9x over **llvm-O3** (both for 100 join relations). For $\mathbf{Q}_{\pi}$ **Flounder** has very low compilations times ranging from 0.1 ms (50 attributes) to 0.6 ms (500 attributes). This leads to factors of improvement up to 933x over **llvm-O3**. We attribute this to the time LLVM spends on register allocation. This is due to the large number of virtual registers used to carry attributes for this workload.

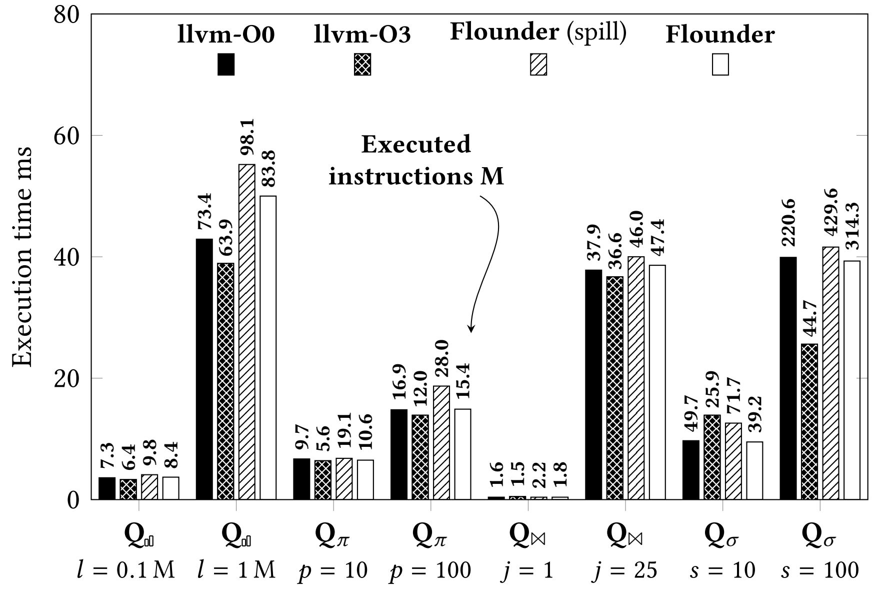


Figure 2.14: Time and instruction count for execution of machine code from different query compilation techniques.

2.6.2 Machine Code Quality

To evaluate machine code quality, we execute two configurations of each query template and measure the *execution time* and the number of *executed instructions*. The results are shown in Figure 2.14. The bars show the execution time in milliseconds and the number on top shows the executed instructions in millions.

Register Allocation We analyze the effect of our register allocation strategy on machine code quality. To this end, we look at the techniques **Flounder** (spill) and **Flounder**. The former uses spill access for every virtual register use. The latter allocates machine registers with the translation algorithm. We observe that register allocation reduces the number of executed instructions by factors between 1.2× and 1.8× (with one exception). This shows that our register allocation strategy effectively reduces the amount of executed spill code. We explain the lack of improvement for $\mathbf{Q}_\bowtie$ $j = 25$ with a large number of hash table operations, which execute invariant library code. The results show that the register allocation technique reduces execution times for all queries by factors between 1.02× to 1.35×. The factors are not as high as the factors between L1 access and register access. This is because the memory access for reading relation data limits throughput (as is typical for database workloads). The improvements shown by the experiment are due to faster machine register access and execution of less spill code.

Comparison with LLVM Next we compare the machine code quality of Flounder and LLVM (cf. Figure 2.14). On average **llvm-O0** executes 1.4× fewer instructions than **Flounder**. The execution times, however, are similar and are longer for **Flounder** only by an average factor of 1.01×. With regard to execution times, the machine code quality resulting from Flounder is similar to **llvm-O0**. We attribute the small time difference despite the higher instruction count to memory bound execution.

The technique **llvm-O3** executes 2.2× fewer instructions than **Flounder** on average. The average factor between the execution times of 1.05× is still low. However, especially queries on larger datasets benefit from the optimizations applied by **llvm-O3**. E.g. the larger variant $\mathbf{Q}_{▯▯}$ 1 M executes 1.3× faster. We conclude that despite the much shorter translation times, our compilation strategy produces code with competitive performance to the machine code generated by LLVM.

2.6.3 Post-Projection Optimizations

The workload $\mathbf{Q}_\pi$benefits from post-projection optimizations. For increasing numbers of projection attributes p, it is preferable to read attributes a_2 to a_p only for tuples that pass the filter (1% of the relation) instead of performing a full scan. We analyze how the code generation strategies handle post-projection optimization by executing $\mathbf{Q}_\pi$with $p = \{10, 50, 100\}$. We use the llvm-based techniques, **Flounder** (naive), and **Flounder** (p.proj). The technique **Flounder** (p.proj) produces IR with explicit post-projection; the other techniques produce IR with full scans.

Observations The experiment results are shown in Figure 2.15. We observe that **Flounder** (naive) has execution times between 8.2 ms and 79.7 ms, and **Flounder** (p.proj) has lower execution times between 6.6 ms and 15.0 ms. Adding post-projection reduces execution times by factors up to 5.3x. The LLVM-based techniques have execution times between 6.4 ms and 14.8 ms. Despite not using post-projection explicitly, LLVM has similar execution performance as the post-projection strategy. We explain this by LLVM adding a similar optimization during machine code generation.

However, these optimization capabilities of LLVM come at the cost of high compilation times (up to 56.7 ms compared to 0.2 ms for **Flounder**). Although **Flounder** does not apply post-projection optimizations automatically, explicit control over post-projections is preferable for DBMSs, which typically use decision mechanisms for projection strategies.

2.6.4 Overall Performance for Characteristic Workloads

We show a table with the overall performance for each technique in Figure 2.16. The workloads are the same as in Section 2.6.2 with two configurations for each

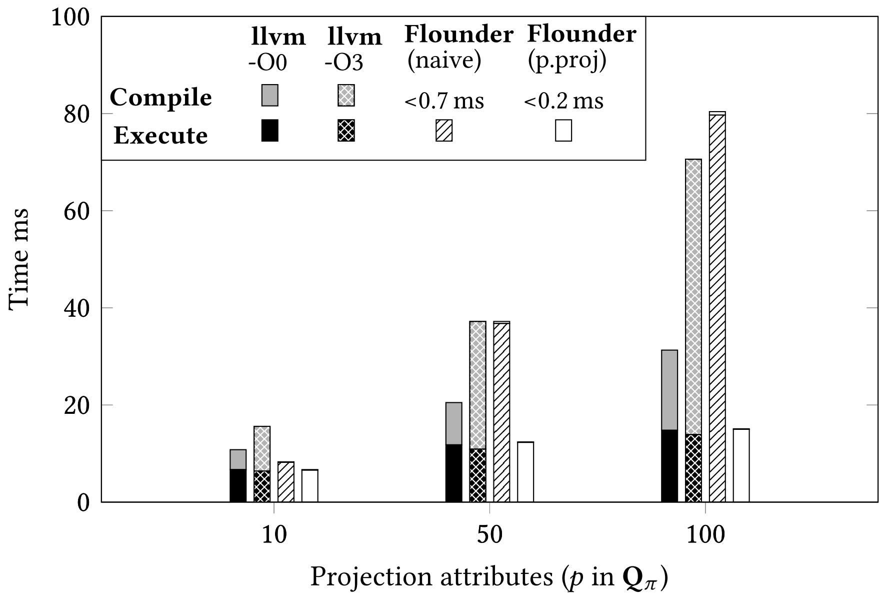


Figure 2.15: Processing the projection workload varying compilation and projection techniques.

		llvm-O0			llvm-O3			Flounder		
		cmpl	exec	**total**	cmpl	exec	**total**	cmpl	exec	**total**
Q📊	l = 0.1 M	4.9	3.5	**8.5**	9.9	3.3	**13.2**	0.1	3.6	**3.8**
Q📊	l = 1 M	4.7	43.6	**48.4**	9.7	38.9	**48.7**	0.1	50.1	**50.2**
$\mathbf{Q}_\pi$	p = 10	4.0	6.5	**10.6**	9.2	6.4	**15.7**	0.1	6.4	**6.4**
$\mathbf{Q}_\pi$	p = 100	15.9	14.0	**29.9**	56.7	13.9	**70.7**	0.1	14.0	**14.1**
$\mathbf{Q}_\bowtie$	j = 1	4.9	0.3	**5.3**	10.9	0.5	**11.4**	0.1	0.3	**0.4**
$\mathbf{Q}_\bowtie$	j = 25	36.8	38.1	**74.9**	105.2	36.7	**141.9**	2.8	39.1	**42.0**
$\mathbf{Q}_\sigma$	s = 10	3.8	9.7	**13.5**	7.8	13.9	**21.7**	0.1	9.5	**9.6**
$\mathbf{Q}_\sigma$	s = 100	10.3	40.0	**50.3**	18.5	25.6	**44.2**	0.2	39.0	**39.2**

Figure 2.16: Overall performance for two configurations of each characteristic workloads (values shown are in milliseconds).

template. The relation sizes range from 10 K to 1 M tuples with total attribute numbers between 2 and 100.

Observations The technique **Flounder** has overall execution times between 0.4 ms and 50.2 ms and **llvm-O0** between 5.3 ms and 74.9 ms. For **llvm-O0**, compilation makes up 46% of the execution on average. For **Flounder** the average is 5%. This leads to better performance of **Flounder** for 7 of 8 queries. For **Q**📊 $l = 1\,M$ compilation times are generally low; thus **llvm-O0** achieves a slightly shorter overall time due to 1.15× faster execution. The technique **llvm-O3** has execution times between 11.4 ms and 141.9 ms, which is longer than the other techniques for 7 of 8 queries. The compilation times make up a high percentage of 62% of the overall on average. The highest factor of improvement of **Flounder** over **llvm-O0** is 10.7x. The highest factor of improvement over **llvm-O3** is 23.2x.

2.6.5 Real World Performance

To evaluate the real world performance of our approach, we execute TPC-H benchmark queries with ReSQL, Hyper, and DuckDB. The relative benefit of lowering compilation latencies depends on the size of the processed data. To this end, we evaluate a smaller database with scale factor 100 MB and another database with scale factor 1 GB. We execute those TPC-H queries that are compatible with ReSQL and report compilation and execution times. We do not show compilation times for DuckDB as it is an interpretation-based engine. The results of the experiment are shown in Figure 2.17 (100 MB database) and in Figure 2.18 (1 GB database).

Observations 100 MB Database Excluding compilation times, the JIT-based engines have shorter execution times than the interpretation-based engine DuckDB. DuckDB's execution times range from 7 ms to 82 ms. Hyper's execution times are shorter by 8.3× on average (1 ms to 12 ms). ReSQL's execution times are shorter by 2.3× on average (7 ms to 32 ms). Including compilation times, however, Hyper is slower than DuckDB for 6 out of 8 queries. This is because Hyper's compilation with LLVM takes up to 117 ms. ReSQL has much lower compilation times than Hyper by up to 106.3× (Q5). The highest compilation time of ReSQL is only 3 ms. This makes ReSQL's faster than DuckDB (up to 3.8×) for all but one query (near even for Q6) and faster than Hyper for all queries (up to 5.5×).

Observations 1 GB Database For the larger database, the execution times (excluding compilation) increase compared to the 100 MB database by 8.9× on average for DuckDB, 13.7× for Hyper, and 10.2× for ReSQL. The compilation times, however, remain unchanged and thus now make up a smaller portion of the overall time for the JIT-based systems. This makes Hyper's overall execution faster than ReSQL for 6 out of 8 queries (1.5× faster on average). Compared to DuckDB, however, ReSQL remains faster by the average factor 1.9x (The factor was 2.0× for the 100 MB database). This is because ReSQL's compilation times have such a small contribution to the overall processing time.

The results show that the simple yet fast compilation approach of ReSQL and Flounder leads to a drastic reduction of compilation times. This leads to faster overall execution times for smaller database sizes (e.g. 100 MB) than state-of-the-art systems. The execution times after compilation of both JIT-based systems are lower than those of the vector-at-a-time engine. This shows that Flounder and ReSQL, despite using simpler translation, can leverage the fast processing speeds of the query compilation approach.

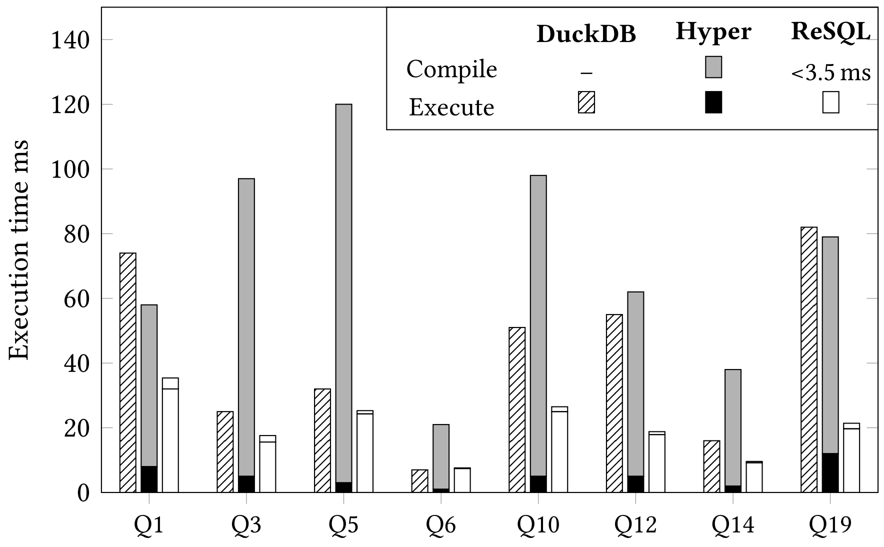


Figure 2.17: Executing TPC-H queries with different approaches on 100 MB database.

2.7 Future Work

Flounder IR is highly specialized. It is designed specifically for database workloads and for one hardware architecture only. This was a no-compromise decision for simplicity and translation performance. In the future, it will be interesting to see which *generalizations* can be applied, e.g. targeting multiple hardware architectures, without adding substantial translation cost. In the following, we discuss several aspects of generalization.

2.7.1 Domain-Specific Processing

Previous work has shown the benefits of combining database processing with other domains, such as data science [92, 66]. For Flounder IR, the addition of such other domains would make the IR suitable for use in a wider range of applications. One way to tackle this idea would be to split IR code into database specific parts and domain specific parts. This would allow it to apply Flounder's capabilities (e.g. scope-based register allocation) to the database specific code and use other compiler techniques (e.g. LLVM's register allocator) to the domain specific parts. A challenging aspect of this direction is defining a good interface between both parts, which should allow efficient interaction (e.g. sharing registers) and specialized compilation (e.g. separate basic blocks).

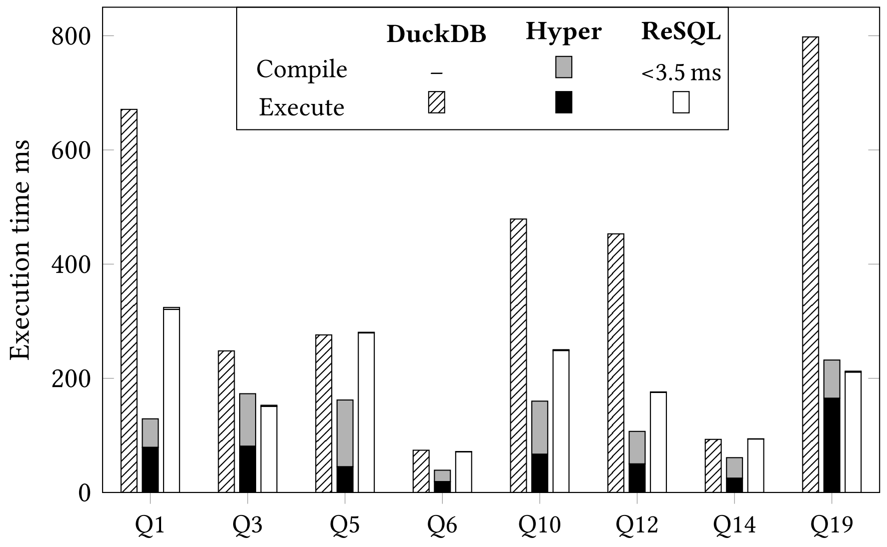


Figure 2.18: Executing TPC-H queries with different approaches on 1 GB database.

2.7.2 Hardware Architectures

A straightforward way of supporting other hardware architecture targets is to take Flounder's approach and apply it to a new target. While some aspects may differ, it seems reasonable that the key technique of scope-based register allocation is applicable for most forms of target machine code. By rewriting the IR, however, the emitter functions from the query compiler have to be rewritten aswell. A more sustainable IR design should therefore include the abstraction of most machine-specific concepts and then offer translation for multiple targets from the same IR. It remains an open question how much translation cost such capabilities would add. An early prototype for Intel and ARM architectures showed promising results so far with compilation times similar to Flounder's for basic IR programs.

2.8 Engineering Query Compilers

From this work there are several lessons learned for the design of query compilers. We have shown that low-level query compilation is feasible with other IRs than LLVM. While there are conceptually no reasons against other IRs, most related work has restricted their IR choices to LLVM. By deviating from this choice, we realize a different compilation process and moreover by designing a new IR, we point out that specializations for database processing are feasible.

Our approach is designed for the x86_64 hardware architecture. As discussed in Section 2.7, using a single IR to target multiple architectures is beneficial because otherwise every architecture would need a separate implementation of the operator translation. Our specialized approach, however, omits such functionality for the benefit of simplicity. As pointed out in the discussion, it is likely that similar CPU architectures could be supported without significant overhead.

However, when it comes to architectures with an entirely different processing model, e.g. data-parallel coprocessors such as GPUs, there are more far-reaching differences. LLVM, for instance, supports a variety of architectures including GPUs. Although LLVM supports the representation of such data-parallel code, the actual IR code for GPUs is different from the IR code for CPU targets. Therefore query compilers that support both CPUs and GPUs have to generate specialized IR code for each architecture. In this sense the benefit of using the same IR for both architectures is limited. Despite using the same IR, the architectural differences necessitate multiple implementation of the operator translators.

2.9 Summary

We showed a query compilation technique that includes all machine code generation steps in the query compiler. The technique uses the intermediate representation Flounder IR that enables *simple translation* of query plans to IR and *fast translation* from IR to machine code. While the translation of query plans to IR is similar to existing approaches, the next step, translation to machine code, is much simpler than in existing techniques. Compared to established low-level query compilers, our approach achieves much shorter compilation times with competitive machine code quality.

The approach of Flounder IR has several applications, which include the query compiler of the database system ReSQL, compilation of other higher-level IRs to machine code, and tuning of machine code for specific hardware architectures. The explicit control over the machine code sequence also makes the approach a good candidate for targeting database-specific architectures [1, 5].

The ReSQL database system was built on top of Flounder IR and uses several translation mechanisms that enable translation from SQL to machine code. We use ReSQL to showcase that the advantages of Flounder's compilation approach carry over to real world workloads.

3 Communication

Another approach that is typically not considered in combination with query compilation are query processing techniques for massively parallel processing devices (e.g. GPUs). With this approach specialized hardware with high bandwidth memory and native support for data-parallelism is used as a promising solution for the design limitations of standard systems. The effective use of GPU-style coprocessors during query processing, however, has proven to be challenging. The processing capacities are so high (e.g. teraflops in one device) that even high-bandwidth memory cannot provision enough data for a reasonable utilization. The throughput is limited by the movement of data and the additional communication channels that are used by such systems exacerbate the issue.

The ability of query compilation to increase the memory efficiency would provide much desired improvements. However, the inherent tuple-at-a-time processing style of the approach does not suit the massively parallel execution model of GPU-style coprocessors. When trying to adapt query compilation to the parallel exeution model, the improvements in efficiency can easily be compromised.

This chapter shows how query compilation and GPU-style parallelism can be made to play in unison nevertheless. We describe a compiler strategy that merges multiple operations into a single GPU kernel, thereby significantly reducing bandwidth demand. Compared to operator-at-a-time, we show reductions of memory access volumes by factors up to 7.5× resulting in shorter kernel execution times by factors up to 9.5×.

Parts of this chapter are contained in published articles [28].

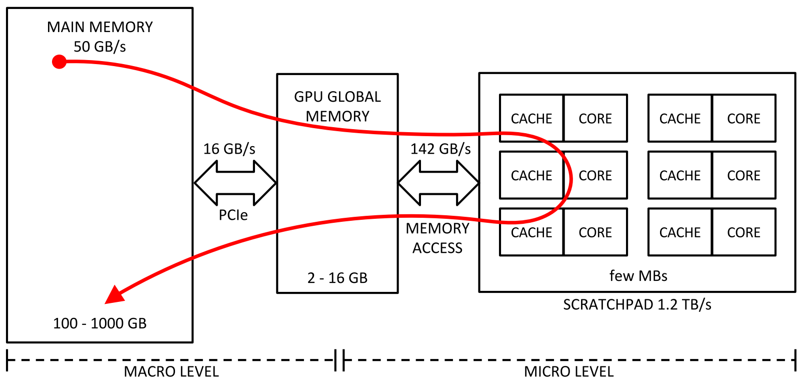


Figure 3.1: The path of a tuple through the memory levels of a coprocessor environment.

3.1 Introduction

GPUs are frequently used as powerful accelerators for query processing. As the arithmetic throughput of the coprocessor peaks in the teraflop range, it becomes a challenge to provision enough data. For this reason, hardware vendors equip graphics cards with high bandwidth memory that has read and write rates of hundreds of GB/s. Still, memory intensive applications such as query processing fall behind regarding the cost of data movement for different reasons. Figure 3.1 shows the path of relational data through the hierarchical memory levels in a typical coprocessor system. Along the path, several bandwidth and capacity constraints need to be considered to achieve scalability and performance:

PCIe / OpenCAPI / NVLink A widely-acknowledged problem is the data transfer bottleneck between the host system and the coprocessor [37], typically via PCIe. Due to the coprocessor's limited memory capacity, data transfers are necessary *during* computations. With an order of magnitude between internal and external memory bandwidth, database developers are challenged with data locality-aware algorithms that efficiently use inter-processor communication. Recent technologies, i.e., OpenCAPI and NVLink, increase the bandwidth over PCIe, shifting the bottleneck towards GPU global memory.

GPU Global Memory The fine-grained data parallelism of a GPU typically requires that kernels perform additional passes over the data. Performing multiple passes, however, can significantly inflate memory loads and can cause a bandwidth bottleneck especially for random memory accesses.

Main-Memory A recent development are integrated GPU-style coprocessors that can directly access the memory of the host CPU. Such an *Accelerated Processing Unit* (APU) allows to use massively parallel processing without additional data transfers. However, the available memory bandwidth is lower than that of a dedicated GPU (30 GB/s vs. hundreds of GB/s).

Scratchpad Memory[1] Scratchpad memory is located on-chip and placed next to each compute unit of a GPU. It can be controlled as an explicit cache for low-level computations and offers a very high bandwidth. However, the capacity is limited to 16 KB – 96 KB per core which makes it challenging to use it for large-scale computations.

3.1.1 Contributions and Outline

In this chapter, we present our GPU-based query compiler HorseQC. We designed HorseQC to account for the hierarchical memory structure of coprocessor environments and for the inherent bandwidth limitations. Our main contribution is to show how various existing techniques can be combined and extended to build an efficient query processing engine on coprocessors.

1. We *analyze the bandwidth limitations* in several execution models (cf. Section 3.2).
2. We show a way to *integrate query compilation* into a coprocessor-accelerated DBMS (cf. Sections 3.3 and 3.4).
3. We present solutions to efficiently process all processing steps of a pipeline in a *single pass over the data* (cf. Sections 3.5 and 3.6).
4. We describe how these parts play together in an overall system (Section 3.7) and evaluate our proposed concepts (cf. Section 3.8).
5. We discuss our results (cf. Section 3.9) and related work (cf. Section 3.10). Then we discuss engineering aspects of our approach (cf. Section 3.11) and finally we conclude (cf. Section 3.12).

GPU-accelerated database systems have used different *macro execution models* in the past. Orthogonally, our work describes a *micro execution model* that can be integrated with different existing macro execution models.

3.2 Macro Execution Model

We first analyze macro execution models that various systems have used in the past. To evaluate a relational query operator, state-of-the-art systems will select a number of primitives and execute the corresponding kernel sequence on the GPU. To feed the kernels with data, the macro execution model defines how data

[1]We use the term *scratchpad memory* to disambiguate *shared memory* for CUDA and *local memory* for OpenCL.

```
Run-to-finish – input: R, output: P
move R Host → GPU
tmp ← op1(R)      /* invoke first GPU kernel */
P ← op2(tmp)     /* invoke second GPU kernel */
move P GPU → Host
```

Figure 3.2: Run-to-finish execution of two successive kernels.

transfers will be interleaved with kernel executions. Here, the data movement from kernel to kernel may result in additional bandwidth demand as compared to conventional systems. To understand the effect, we study the implications that existing macro execution models have on the use of bandwidth at multiple levels (PCIe, GPU global memory, etc.). As a poster child, we profiled the execution of Query 3.1 from the star schema benchmark (SSB) [75]. The query was executed at scale factor 10 with CoGaDB [18] on a NVidia GTX970 GPU[2] In the following, we discuss three macro execution models: *run-to-finish*, *kernel-at-a-time* and *batch processing*.

3.2.1 Run-To-Finish (Not Scalable)

A straightforward way to execute a sequence of kernels is to first transfer all input, execute the kernels, and finally transfer all output. The approach, illustrated in Figure 3.2, has the advantage that intermediate data remains in GPU global memory in-between kernel executions and no significant PCIe transfers are necessary. However, run-to-finish has the disadvantage that it only works if *all* input, output, and intermediate data is small enough to fit in GPU memory. Run-to-finish macro execution models are used, e.g., by Ocelot [40] and CoGaDB [18]. The *lack of scalability* leads us to evaluate the following execution models.

3.2.2 Kernel-At-A-Time

To process large data on coprocessors, we can execute each kernel on blocks of data. The pseudocode of this approach is shown in Figure 3.3. Processing blocks of data requires algorithm choices that can deal with partitioned inputs. Joins or aggregations, for instance, can only be processed in this mode if their internal state (e.g. a hash table) can fit in GPU global memory.

We analyze the data movement of kernel-at-a-time for SSB Query 3.1. Blocks are first moved via PCIe from the host to the coprocessor and then read by the kernel from GPU global memory (output passes both levels vice-versa). In this way, the data volumes for GPU global memory accesses equal the data volume

[2]We measured 146.1 GB/s GPU global memory bandwidth in a host system with 16 GB/s bidirectional PCIe bandwidth.

```
KERNEL-AT-A-TIME – **input:** R, **output:** P
foreach $r_i$ in R=$r_1 \cup \cdots \cup r_m$ do
    move $r_i$ Host → GPU
    $m_i$ ← op1($r_i$)    /* invoke first GPU kernel */
    move $m_i$ GPU → Host (assemble into M)
foreach $m_j$ in M=$m_1 \cup \cdots \cup m_n$ do
    move $m_j$ Host → GPU
    $p_j$ ← op2($m_j$)   /* invoke second GPU kernel */
    move $p_j$ GPU → Host (assemble into P)
```

Figure 3.3: Kernel-at-a-time achieves scalability by transferring I/O for each kernel through PCIe.

transferred via PCIe, plus the cost to build up the hash tables in GPU global memory (0.4 GB here). Figure 3.5a shows the resulting data movement.

In the figure, the arrows annotated with data volumes represent PCIe transfers and GPU global memory accesses. We aggregated the data volumes by kernel types (e.g. scan, gather) and show only the most important kernels responsible for 88.2% of the memory traffic. Given a PCIe bandwidth of 16 GB/s, all PCIe transfers together require at least 350 ms to complete. This exceeds the aggregate time for GPU global memory access by a factor of **5.8x**. For kernel-at-a-time processing *the PCIe link is clearly the bottleneck.*

Kernel-at-a-time processing is used to scale out individual operators [47]. Unified virtual addressing (UVA) produces the same low-level access pattern, albeit transparent to the system developer.

3.2.3 Batch Processing

We can alleviate PCIe bandwidth limitations by rearranging the operations of kernel-at-a-time. Instead of running kernels until a column is processed, we can short-circuit the transfer of intermediate results to the host. Batch processing achieves this by reusing the output of the previous operation (op1) as input for the next operation (op2) instead of transferring to the host. This is applicable whenever intermediate batch results can be kept within GPU global memory. The corresponding pseudocode is shown in Figure 3.4.

We analyze the data movement cost with the example of SSB Query 3.1. The GPU global memory load is the same as for kernel-at-a-time processing, because each kernel reads and writes I/O to GPU global memory. We obtain the PCIe transfer cost using the transfer volumes of input columns of the query and output of the final result. Figure 3.5b shows the resulting data movement cost. Batch processing reduces the amount of PCIe transfers by a factor of **8.8x**. This shows that transferring data in blocks *and* performing multiple operators per block allows

```
BATCH PROCESSING – input: R, output: P
foreach $r_i$ in R=$r_1 \cup \cdots \cup r_m$ do
    move $r_i$ Host $\rightarrow$ GPU
    tmp$_i$ $\leftarrow$ op1($r_i$)    /* invoke first GPU kernel */
    $p_i$ $\leftarrow$ op2(tmp$_i$)  /* invoke second GPU kernel */
    move $p_i$ GPU $\rightarrow$ Host (assemble into P)
```

Figure 3.4: Batch processing executes multiple kernels for each block that is transferred via PCIe.

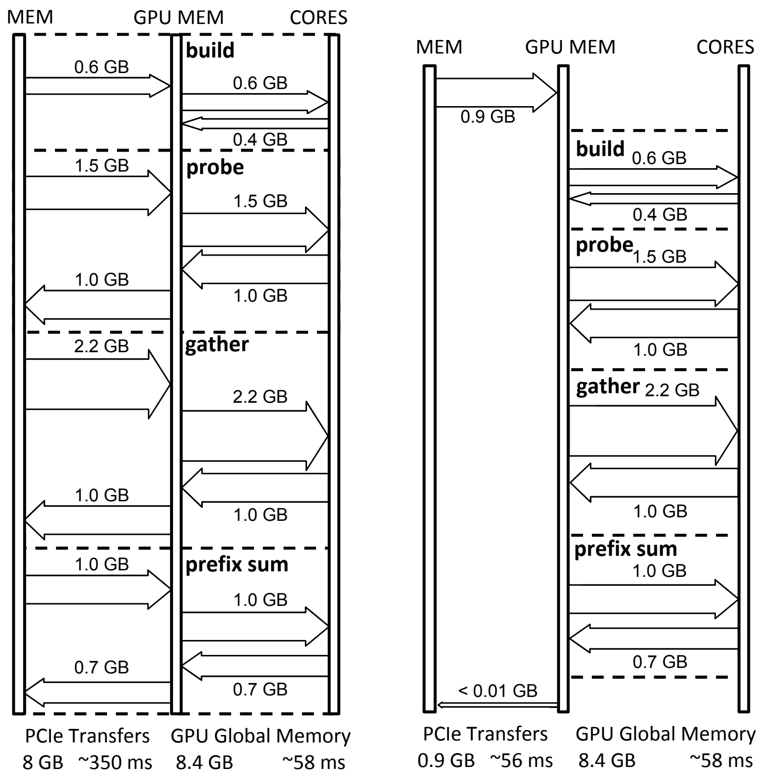
scalability and increases the efficiency compared to kernel-at-a-time.

Batch processing macro execution models have been used for coprocessing by GPUDB [104] and Hetero-DB [105]. Wu et al. [101] describe the concept as *kernel fission* and detect opportunities to omit PCIe transfers automatically.

Limitations The lower amount of PCIe traffic can expose GPU global memory bandwidth as the next limitation. Batch processing reduces the PCIe transfer cost, but the amount of GPU global memory access remains unaffected. The memory access volume inside the device is now an order of magnitude larger which, despite the high bandwidth, takes longer to process than the PCIe bus transfers (Figure 3.5b). For this reason, batch processing SSB Query 3.1 is *not* limited by PCIe transfers, but by accesses to the (high-speed) GPU global memory. Since in typical query plans, I/O and hashing operations both address the same GPU global memory, the situation, in fact, may arise frequently in real-world workloads.

Other Queries A limiting amount of global memory access can easily occur when many kernels are executed one after another. Karnagel et al. [46] show that a simple query with one selection and one aggregation operator already uses 13 kernels for processing. To determine the prevalence of GPU global memory bandwidth limitations, we profiled several queries from the TPC-H [13] and SSB benchmark [75] sets.[3] We look at the ratio of memory access to PCIe traffic as number of passes to assess the load on memory and bus links. Table 3.1 shows the number of passes for queries from the TPC-H and SSB benchmarks. With a symmetric memory load, we can afford $\frac{146\,\text{GB/s}}{2\cdot 16\,\text{GB/s}} \sim 4$ to 5 passes before being limited by GPU global memory. While memory can adapt to asymmetric read and write loads, PCIe can service each direction with at most 16 GB/s. This changes the number of affordable passes for asymmetric workloads to $\frac{146\,\text{GB/s}}{16\,\text{GB/s}} \sim 9$ in the worst case. Queries that require more than 9 passes are always limited by memory bandwidth before being affected by the PCIe bottleneck. In Table 3.1 this is the case for 9 out of 24 queries, which indicates that it is crucial to reduce the GPU global memory load.

[3]Note that CoGaDB does not support all TPC-H queries.



(a) kernel-at-a-time (b) batch processing

Figure 3.5: Data movement for processing SSB Query 3.1. While the throughput of (a) is limited by PCIe transfers, (b) exposes GPU global memory access as the next bottleneck.

3.3 Micro Execution Model

Tuning the macro level helps to remove the main bottleneck for scalability: data transfers over PCIe. However, the macro level change exposes a new bottleneck: the memory bandwidth of GPU global memory (cf. Section 3.2.3). To utilize the GPU global memory bandwidth more efficiently, we need to apply additional micro-level optimizations using *micro execution models* and combine them with the macro execution model (batch processing) to achieve scalability *and* performance.

Existing micro-level optimizations such as *vector-at-a-time* processing [16] and *query compilation* [69] utilize memory bandwidth more efficiently by leveraging pipelining in on-chip processor caches. Therefore, both techniques are promising candidates for opening up the bottleneck of limited GPU global memory bandwidth. However, vector-at-a-time processing and query compilation are designed in the context of CPUs. While it is highly desirable to apply both techniques in the context of GPUs, mapping the techniques from CPU to GPU is challenging, which we discuss in the following.

Query	Passes	Query	Passes	Query	Passes
ssb11	7.5	ssb34	2.2	tpch5	7.2
ssb12	6.9	ssb41	7.4	tpch6	6.2
ssb13	6.7	ssb42	3.9	tpch7	**9.0**
ssb21	**9.6**	ssb43	3.5	tpch9	**9.0**
ssb22	**9.2**	tpch1	**15.5**	tpch10	5.8
ssb23	**9.1**	tpch2	**14.5**	tpch15	6.3
ssb31	**11.0**	tpch3	5.2	tpch18	**38.5**
ssb32	7.9	tpch4	6.6	tpch20	**10.5**
ssb33	7.5				

Table 3.1: Number of passes for benchmark queries. Out of 25 queries, 9 are definitely limited by GPU global memory.

3.3.1 Vector-At-A-Time

To mediate the interpretation overhead of Volcano and the materialization overhead of operator-at-a-time, vector-at-a-time uses batches that fit in the processor caches. First, this reduces the number of `getNext()` calls from one per tuple to one per batch. Second, this makes materialization cheap because operators pick up the cached results of previous operators. On CPUs, vector-at-a-time benefits from batch sizes that are large enough to limit the function call overhead and small enough to fit in the CPU caches.

On GPUs, the compromise between tuple-at-a-time and full materialization strategies is not a sweet spot, however. Kernel invocations are an order of magnitude more expensive than CPU function calls. Furthermore, GPUs need much larger batch sizes to facilitate over-subscription and out-of-order execution. This leads to the problem that batches, which fit in the GPU caches, are too small to be processed efficiently. Alternatively, more recent GPUs support *pipes* to move a local execution context from one kernel to another. This has been used by GPL [77] for query processing. However, this technique still introduces an overhead for switching the execution context. In addition, it is limited to a depth of 2–32 kernels depending on the microarchitecture.

3.3.2 Query Compilation

Query compilation is a commonplace tool for avoiding excessive memory transfers during query processing. Compiling code for incoming queries becomes feasible with low-level code generation and achieves performance close to hand-written code. The compilation strategy of Neumann [69] keeps intermediate results in CPU registers and passes data between operators without accessing memory at all. The generated code processes full relations or blocks of tuples using a sequential tight loop.

To use query compilation on GPUs, we must integrate fine-grained data-parallelism into compiled queries. The parallelization strategy of Hyper [56], however, uses a coarse-grained approach, which allows it not to break with the

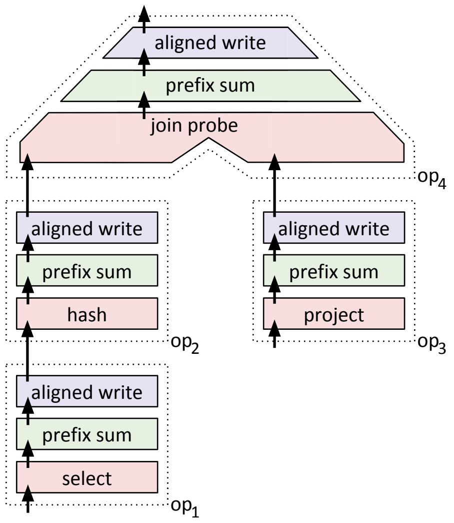


Figure 3.6: Operator-at-a-time

concept of tight loops. In fact, Hyper does not use SIMD instructions [69] and thus omits fine-grained data-parallelism. Even on CPUs with a moderate degree of parallelism in SIMD instructions, database researches are challenged with integrating query compilation and SIMD [90, 63].

In summary, using a micro-level technique for efficient on-chip pipelining on GPUs remains a challenge. Applying any of the commonplace techniques makes it necessary to combine at least three things that are hardly compatible: fine-grained data-parallel processing, extensive out-of-order execution, and deep operator pipelines. To achieve our goal of mitigating the GPU global memory bottleneck, we need to develop a new micro execution model which we build up step by step in the following sections.

3.4 Data-Parallel Query Compilation

In the following, we show a micro-level execution strategy that reduces GPU global memory access volumes by means of pipelining in on-chip memory. To this end, we show the approach of our query compiler HorseQC and its integration with the operator-at-a-time execution engine of CoGaDB [18].

3.4.1 Fusion Operators

HorseQC extends the operator-at-a-time approach with the concept of *fusion operators*, operators that embrace multiple relational operations. A fusion operator replaces a sequence of conventional operators in the physical execution plan with a micro-level-optimized pipeline. The data movement within a fusion operator can be improved by applying different micro level execution models.

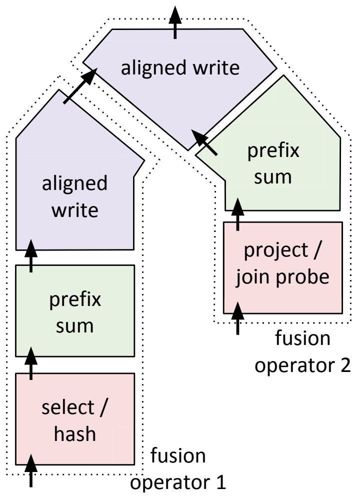


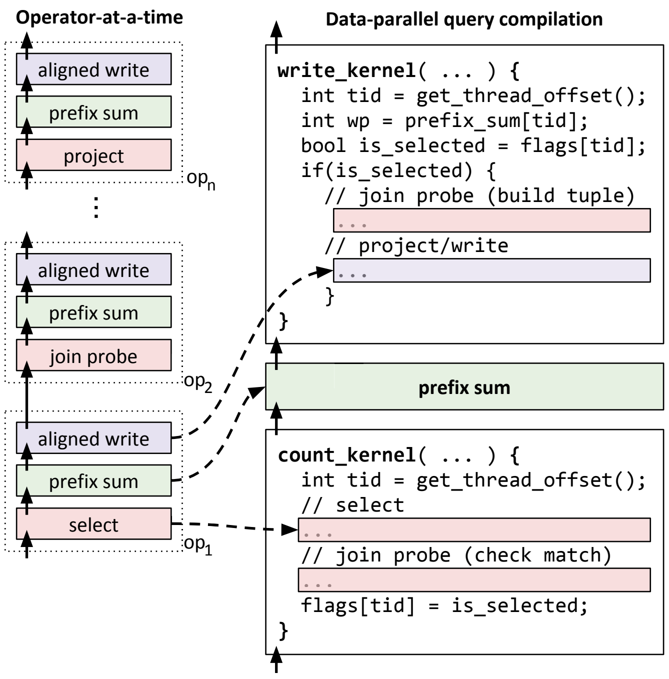
Figure 3.7: Multi-pass query compilation

3.4.2 Micro-Level Pipeline Layout

To keep matters simple, we first apply query compilation with the operator-at-a-time primitives described by He et al. [39]. This choice is not limiting as other data-parallel primitives may be used instead. However, a commonality of different primitive sets is that they use *relational primitives* with relational functionality (e.g. select) and *threading primitives* with thread coordination functionality (e.g. map, prefix sum, gather).

State-Of-The-Art We look at a query with two input tables and a total of four relational operators $op_1, \cdots, op_4$. Operator-at-a-time runs three primitives per operator (cf. Figure 3.6): The first pass executes the relational primitive (e.g. select, project) and counts the number of outputs of each thread. The second pass computes a *prefix sum* to obtain unique per-thread write positions. The third pass performs an *aligned write*. This means that the output values are written into a dense array and may include executing the relational primitive for a second time to produce the output values. Thus, the query is processed in twelve operations with separate GPU global memory I/O.

Multi-Pass Query Compilation By grouping operations that are applied to the same input table, the query may be processed with two fusion operators. Within each fusion operator, we apply the following query compilation strategy (cf. Figure 3.7): We extract the prefix sum from the operators and execute it only once between all relational primitives and all aligned writes. The relational primitives are then compiled into one kernel called `count`, which is executed before the prefix sum. The aligned writes are compiled into one kernel called `write`, which is executed after the prefix sum. In this way, we apply *kernel fusion* [96] to the four relational primitives and to the four aligned writes. The same query is processed with six



```
write_kernel( ... ) {
  int tid = get_thread_offset();
  int wp = prefix_sum[tid];
  bool is_selected = flags[tid];
  if(is_selected) {
    // join probe (build tuple)
    ...
    // project/write
    ...
  }
}
```

```
count_kernel( ... ) {
  int tid = get_thread_offset();
  // select
  ...
  // join probe (check match)
  ...
  flags[tid] = is_selected;
}
```

Figure 3.8: Transforming data-parallel operator-at-a-time into compiled execution. The functionality of each operator maps to designated positions in the generated kernels.

operations and the operations in compiled kernels communicate through on-chip memory instead of GPU global memory.

3.4.3 Instancing Relational Code Templates

We briefly describe the process used by HorseQC to compile OpenCL code for the `count` and `write` kernels by an example of the projection operation (similar to [19]). Each primitive, except for prefix sum, is mapped to a designated position in the `count` kernel or in the `write` kernel (cf. Figure 3.8). The query compiler receives a C++ object that describes the primitive's functionality (e.g. a tree for an arithmetic expression) and maps the semantics to fragments of OpenCL. To illustrate, $\pi_{\texttt{revenue}\leftarrow\texttt{price*discount}}$ would compile to

```
revenue[wp] = price[tid] * discount[tid];
```

The global index `tid` is used to access the input columns and the write position `wp` is used for the output columns.

The instantiated code is placed in a code frame, which has several invariant features, e.g., thread offset computations, a surrounding loop, as well as managed features such as a parameter list. Projection is positioned in a conditional clause of the `write` kernel that is entered by all threads with a positive `is_selected` flag

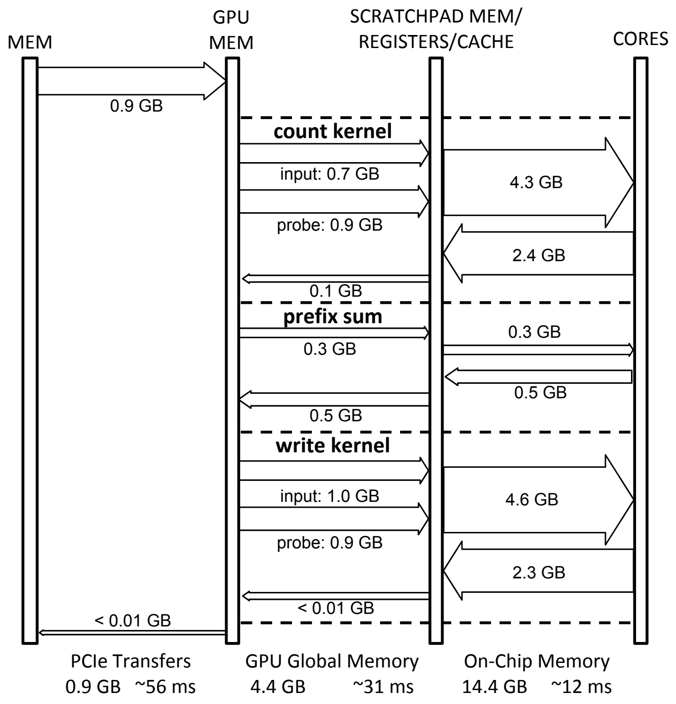


Figure 3.9: Data movement for data-parallel query compilation with three phases.

(cf. Figure 3.8). Other operations may include function calls for reductions or hash table operations.

3.4.4 Memory Access and Limitations

In Figure 3.9, we illustrate the bandwidth characteristics of our example query when using code generation with three phases. The figure shows the behavior of the three-phase micro execution model described above with the batch processing macro execution model. To analyze the implications of forwarding intermediate results in the generated kernels through registers and scratchpad memory, we extended the illustration with an additional GPU-internal layer of memory.

GPU global memory access has previously been the bottleneck for query execution. Here the `count` kernel accesses 1.7 GB in GPU global memory, the prefix sum computation 0.8 GB, and the `write` kernel 1.9 GB respectively. This is a reduction by factor **1.9x** compared to batch processing. In the generated kernels, a substantial amount of memory traffic has moved to on-chip memory. In on-chip memory, the access volume of 14.4 GB is not a limiting factor due to the extremely high bandwidth of 1.2 TB/s of scratchpad memory.

Although the reduced GPU global memory traffic may suggest that the approach eliminates the bottleneck, real world queries still experience limitations. In fact, Section 3.8.3 shows that compilation with three phases can still not saturate PCIe for 9 out of 12 SSB queries. This is because the query complexity prevents the

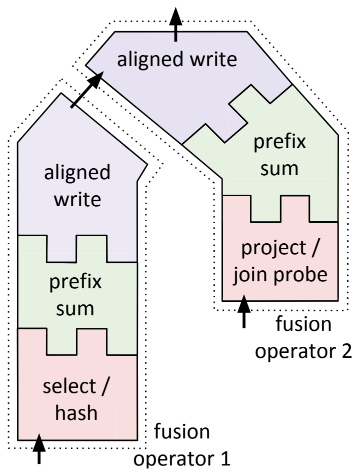


Figure 3.10: Compound kernel

strategy from utilizing the full GPU global memory bandwidth. Therefore, we investigate ways to further increase the processing efficiency in the next section.

3.5 Processing Pipelines in One Pass

The previous execution model relied on a typical programming concept of GPUs that executes operations with multiple kernels. The kernels that execute the actual work for the operation are interleaved with kernels that execute prefix sum computations. To further improve the processing efficiency, we have to break with this concept. With a new micro execution model, we avoid round trips to GPU global memory, which are caused by multi-pass implementations. This enables us to radically reduce GPU global memory traffic and lift the bandwidth bottleneck.

Compound Kernel Kernel fusion brought reduction operations (e.g. prefix sum) as boundaries into the spotlight. Previously, we computed the prefix sum *between* two generated kernels to obtain write positions. Instead of two separate kernels, we now generate only one *compound kernel* that integrates the prefix sum computation (cf. Figure 3.10) and this eliminates multiple passes. Computing write positions *within* a generated kernel makes it possible to process pipelines in one pass without intermediate materialization. In this way, each fusion operator is executed by a single compound kernel. In the following, we look at implementation strategies for reduction operations that enable fully pipelined processing.

3.5.1 Pipelining Data-Parallel Reductions

Reductions are a poster child for data-parallel algorithms [42] and have been investigated in detail regarding complexity, efficient implementations, and their applications. In the context of database systems, they are especially relevant in the context of prefix sums, sorting, and aggregations [3, 35]. The latter involves two techniques: Simple reductions aggregate to a single tuple and segmented reductions compute grouped aggregates on sorted data [87]. As reductions have inherent parallel dependencies, they are typically implemented in a hierarchical structure that involves running multiple kernels in sequence. This approach is applied in state-of-the-art coprocessor database systems such as Ocelot [40], CoGaDB [18], GPUDB [104], Kernel Weaver [99] and Voodoo [79].

Atomic Prefix Sum The separation into multiple reduction kernels with intermediate materialization is an obstruction for pipelining. To introduce a pipelined implementation, let us look at a very simple sequential prefix sum at first:

```
for(i=0; i<n; i++)
    if(flags[i]) prefix_sum[i] = sum++;
```

The sequential prefix sum loops through the array `flags` while writing *and* incrementing `sum` for every valid entry. Figure 3.11a illustrates the use of the prefix sum for a dense write of selected input elements. When parallelizing the `for`-loop, this implementation runs into the issue of many threads trying to increment `sum` at the same time. To resolve this parallel dependency, atomic operations can be used to isolate parallel modifications of the same memory address. Atomic operations ensure a consistent state, yet are executed in an undefined order. The following code executes an *atomic prefix sum* to compute unordered, dense write positions:

```
if(is_selected) wp = atom_add(&sum, 1);
```

Threads contribute an offset of `1` to the sum at address `&sum` by executing the expression conditionally. Each `atomic_add(..)` returns the previous state of `sum`. Thus, threads immediately obtain a unique global write offset as `wp` in register. This is illustrated in Figure 3.11b.

The use of atomic operations causes a break with the semantic of the prefix sum because the result has *no defined order*. For the relational semantic, however, only the *uniqueness* of output positions is critical. Output permutations only lead to non-aligned GPU global memory access where adjacent threads do not necessarily write to adjacent memory addresses. The impact on write throughput, however, is limited, because the filter semantics lead to non-aligned access for separate prefix sums too.

3.5.2 Code Generation for Compound Kernels

Computing write positions within a generated kernel allows us to contract the three phases within a fusion operator into one *compound kernel*. This simplifies code generation for two reasons (cf. Figures 3.8 and 3.12): First, selection flags and write offsets remain in registers and do not have to be passed between kernels through materialization. Second, relational primitives that occur both in the `count`

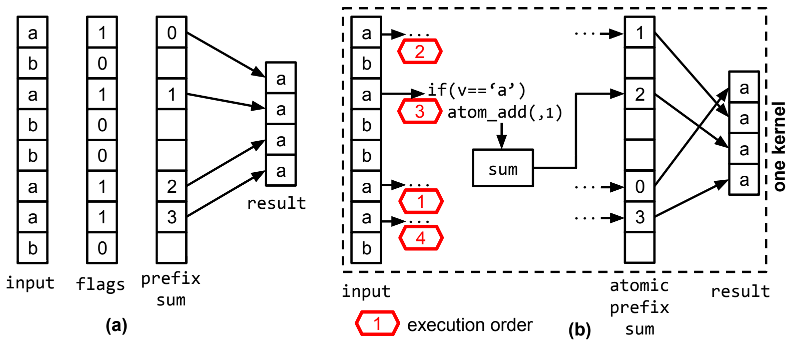
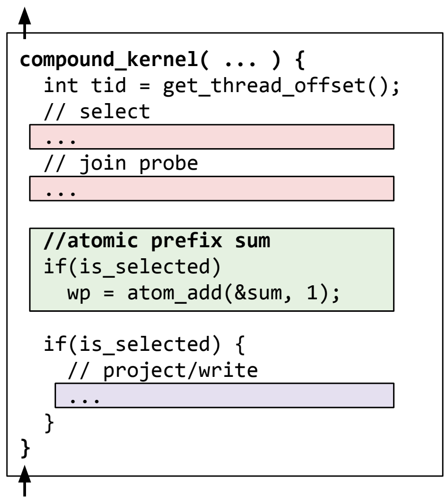


Figure 3.11: The computation of a prefix sum for writing selected elements to a dense array (a) can be parallelized using atomic operations (b).



```
compound_kernel( ... ) {
  int tid = get_thread_offset();
  // select
  ...
  // join probe
  ...

  //atomic prefix sum
  if(is_selected)
    wp = atom_add(&sum, 1);

  if(is_selected) {
    // project/write
    ...
  }
}
```

Figure 3.12: The compound kernel integrates all three pipeline phases into one kernel.

and in the `write` kernel are executed only once in the compound kernel, e.g., we probe the hash table to check the number of matches and keep the payload in registers for projection. This becomes possible in the compound kernel as the register content remains valid until projection.

To instantiate relational primitives, we follow a similar procedure as previously described, but now we use only one kernel code frame: All relational primitives that affect the number of outputs are placed before the atomic prefix sum and all relational primitives that produce output after it. The atomic prefix sum is instantiated from an invariant code template that takes the `is_selected` flag as input and assigns the write position `wp` as output. Both the input flag and the write position are available in registers.

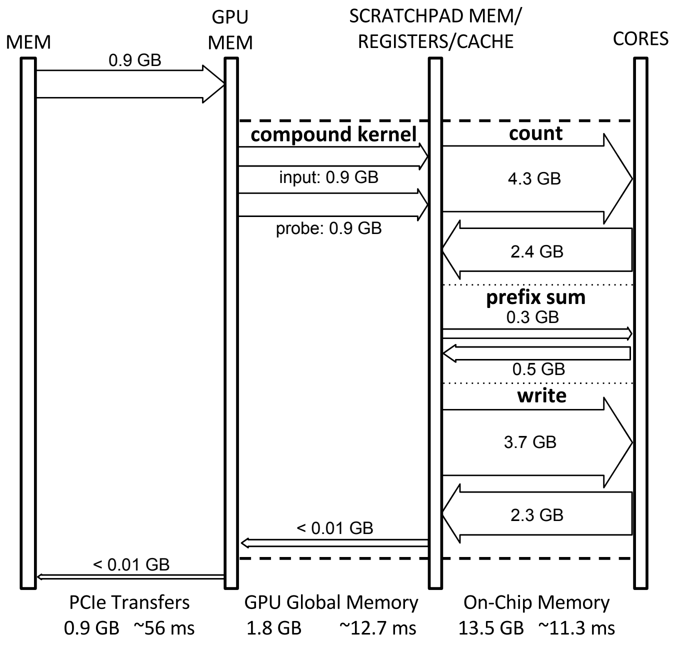


Figure 3.13: Data movement for query compilation with one pass. The compound kernel reduces data movement by 4.7x.

3.5.3 Memory Access and Limitations

The compound kernel micro execution model further reduces GPU global memory access by a factor of 2.4x to 1.8 GB (see Figures 3.9 and 3.13). Compared to operator-at-a-time, this is a reduction by a factor of **4.7x**. Pipelining the prefix sum avoids round trips to GPU global memory that are necessary in the three-phase micro execution model. The compound kernel has only a minimal GPU global memory access volume for input, output and hash table access. Now the on-chip traffic is balanced with the GPU global memory traffic when relating each memory volume to the available bandwidth.

The described approach heavily relies on atomic operations. This has the disadvantage to cause limitations for parallelism. Although the execution order is undefined, the operations *are* sequentialized and reducing n values takes $O(n)$ parallel steps. However, Egielski et al. [26] show that recent hardware support can make atomic operations competitive to parallel algorithms. Still, the integrated prefix sum puts a significant pressure on the atomic functional units, which prevents pipeline kernels from utilizing full GPU global memory bandwidth. In the following, we address this issue and show how the efficiency of parallel reductions in compound kernels can be increased.

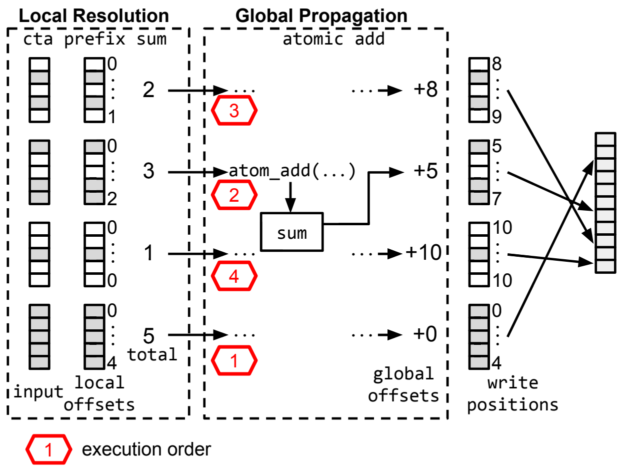


Figure 3.14: Computing write positions with local resolution (local offset), global propagation (global offset).

3.6 Efficient Pipelined Reductions

Previously, we showed a way to pipeline reductions in generated kernels using atomic operations. This benefits the memory efficiency, but at the same time exposes the atomic functional units of a GPU as the bottleneck. This is especially critical because several operations that are combined in the compound kernel rely on atomic isolation as well, i.e., state-of-the-art implementations of hash joins and hash aggregations [47] use atomic operations to isolate hash table inserts.

This section addresses performance bottlenecks that occur when utilizing atomic reductions to pipeline relational operators. We show a new technique *local resolution, global propagation*, that is used by HorseQC to pipeline prefix sums, single tuple aggregation and grouped aggregation efficiently. The approach reduces the pressure on atomic functional units and offers tunability regarding hardware and thread group granularity. We describe the approach in the following.

3.6.1 Local Resolution, Global Propagation

Similar to other efficient GPU implementations as in CUB [64], local resolution with global propagation consists of two levels of reductions. In contrast to other techniques, local resolution, global propagation always uses pipelined techniques on *both* levels. Local resolution is an additional pre-reduction step, computed by a local thread group, whereas global propagation is the same atomic reduction as described in Section 3.5. We use the term *thread group* for the threads that collaborate during local resolution. Thread groups can either represent the workgroups (AMD) or thread-blocks (NVidia) executed by the GPU or work on subgroups of them (e.g. warp).

The following code, illustrated by Figure 3.14, executes an atomic prefix sum using local resolution, global propagation:

```
local_offset = group_prfx ( flags, &group_total ); //local resolution
if ( thread_group_idx == 0 )
    global_offset = atom_add( &sum, group_total ); //global propagation
write_pos = local_offset + global_offset;
```

First, each thread group executes `group_prfx` to compute a local prefix sum on `flags`. This is the local resolution step. We implement `group_prfx` e.g. with SIMD reductions (cf. Intra-Warp Scan Algorithm by Sengupta et al. [86]). The function returns the local offset `local_offset` and the sum of all flags assigned to the thread group `group_total`. Second, one thread of each thread group adds `group_total` atomically to a global counter `sum`. This is the global propagation step. The call to `atom_add` returns the global offsets `global_offset`. Finally, the write position `wp` is the sum of `local_offset` and `global_offset`.

Compared to the simple atomic prefix sum, we now add pre-aggregates instead of 1/0 flags to `sum`. Therefore, each atomic add obtains ranges of output indices instead of a single index. Effectively, this *allocates* segments of output memory to thread groups. The order of the allocations, however, is undefined (see execution order in Figure 3.14). This leads to output that is ordered *within segments* and permuted *between segments*. Further investigation revealed that, due to the GPUs stream processing engine, the permutations exhibit locality, leading to semi-ordered output data.

3.6.2 Local Resolution Mechanisms

The mechanisms used for local resolution are interchangeable. This enables tuning the reduction technique and to apply them in different operations. Figure 3.15 illustrates different types of reductions used in local resolution before contributing to global propagation via atomic operations. Figure 3.15 (a) shows work-efficient reductions [12] and Figure 3.15 (b) shows SIMD reductions [86]. Both are used for pipelined prefix sums and for aggregations. The work-efficient reduction computes one pre-aggregate per thread group, whereas the SIMD reduction computes four. By picking the right technique, we can adapt to the hardware parallelism of different coprocessors.

Figure 3.15 (c) illustrates segmented reductions [87] as local resolution technique. Segmented reductions are used to compute pre-aggregates for grouped aggregations. A similar approach PLAT [103] aggregates frequent grouping keys in a table local to each CPU core. During The ability to control scratchpad memory opens up a new design space for grouping algorithms in pipelined computations (e.g. handling frequent items).

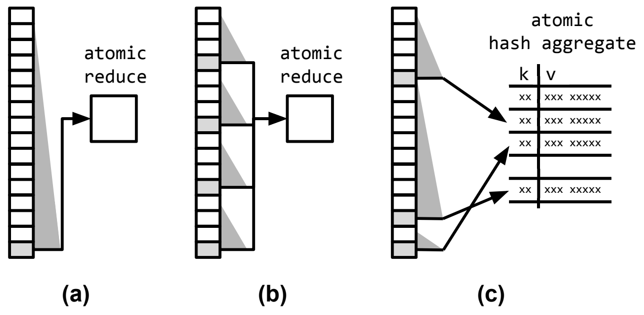


Figure 3.15: Local resolution mechanisms: (a) Work-efficient reduction (b) SIMD reduction (c) segmented reduction.

3.7 DBMS Integration

We integrated our query compiler HorseQC into the open source DBMS CoGaDB, leveraging the built-in code generator [19]. The DBMS uses a columnar data layout and processes full columns operator-at-a-time on GPUs and CPUs. We use the front-end and the storage layer of CoGaDB, and HorseQC adds a new compiler-based execution engine.

We added two components to the DBMS: 1. a query compiler that compiles fusion operators to GPU code (cf. Section 4) and 2. a *translation layer* that identifies fusion operators and drives the query compiler. Currently, there are two different workflows for the translation layer:

1. CoGaDB parses the SQL code for a query and generates a query plan. The translation layer applies the produce/consume model [69] to the query plan to determine fusion operators. We use this approach for the SSB queries and TPC-H Q6.

2. The translation layer parses a JSON file that describes the query plan including the fusion operators. This enables us to process queries when (1) cannot handle the queries via SQL (e.g. correlated subqueries or automatic unnesting). This is used for the other TPC-H queries.

When the fusion operators are defined, the translation layer drives the query compiler to compile and execute. Finally, decompression of dictionary compressed columns and sorting are executed by CoGaDB's original execution engine.

3.8 Evaluation

Section 3.2.1 showed that query coprocessing in existing macro execution models is sensitive to memory bandwidth bottlenecks on various hierarchical levels. We proposed several micro execution models that allow to remove memory indirections to achieve a more efficient use of bandwidth. In this section, we evaluate our approaches and carefully assess bandwidth and throughput to show several benefits.

The experimental study is structured as follows: First, we evaluate the *micro execution models*. Therefore, we execute specific queries to analyze the *reduction performance* of the proposed techniques in the first two experiments. Then, we evaluate the micro execution models for the SSB and TPC-H benchmarks. After this, we analyze the *integration* of our micro execution model with the batch processing *macro execution model*. Finally, we analyze the *real-world benefits* of our approach with a comparison of end-to-end performance and a scalability analysis. Note that all experiments (except *Scalability*), were executed with scale factor 10.

Processing Techniques We use three micro execution models from HorseQC and Operator-at-a-time. The goal of our micro execution models is to use them within macro execution models to improve performance. We show the benefit of our approaches by comparing them to an operator-at-a-time micro execution model. In this way, we analyze the benefit of moving data transfers between relational operators from the memory to the on-chip level.

- Multi-pass. The first approach separates reductions from JIT-compiled kernels, which leads to an execution in multiple passes (Section 3.4). Each reduction is executed on materialized data using the `boost::compute` library.

- Pipelined. The second approach integrates reductions into a fully pipelined kernel using atomic operations (Section 3.5). By using atomic operations for each reduction input, the approach is an instance of local resolution, global propagation that has no local resolution step.

- Resolution. The third approach increases the efficiency of pipelined reductions with local resolution (e.g. pre-aggregation, Section 3.6). We differentiate between local resolution implementations using Resolution:SIMD and Resolution:WE (work-efficient).

- Operator-at-a-time. We use CoGaDB 0.4.1, which processes full columns of data in each operator with CUDA kernels. It features a run-to-finish macro execution model and an operator-at-a-time micro execution model.

Baselines We use two baselines to assess our results. The *PCIe transfer time* for transferring input and output data between the host's main-memory and GPU global memory. It is the target time for micro execution models for balancing throughput and PCIe bandwidth. The PCIe transfer time is shown in each graph with a dashed line (- - -).

The *GPU global memory bound execution time* is the time for reading the input data and for writing the output data. The baseline is a lower bound on the kernel execution time. We indicate it with a solid line (——) in each graph.

Model	Type	Archi-tecture	Cores	Scratch pad (KB)	B/W (GB/s)
GTX970 (NV)	GPU	Maxwell	13	96	146.1
GTX770 (NV)	GPU	Kepler	8	48	167.6
RX480 (AMD)	GPU	Ellesmere	32	32	104.9
A10 (AMD)	APU	Godavari	8	32	18.7

Table 3.2: Coprocessors used in the evaluation.

```
select lo_extprice * lo_discount + lo_tax as revenue
from   lineorder
where  lo_quantity between 25 - x and 25 + x
```

Figure 3.16: Select/Project query inspired by star schema benchmark.

System Configuration For the experiments, we use three dedicated GPUs with PCIe gen 3.0 links and one APU that accesses main-memory directly. Table 3.2 specifies the GPU models and shows hardware properties. The amount of scratch-pad is available *per core*. The reported bandwidth refers to GPU global memory for the GPUs and to main-memory for the APU. It was measured using on-GPU `memcpy` of 1 GB data. We measured bidirectional PCIe transfers between CPU and GPU as 12.1 GB/s.

Both NVidia GPUs GTX770 and GTX970 run in a system with an Intel Xeon E5-1607 CPU. We use the NVidia 364.19 driver and CUDA Toolkit 7.5 with OpenCL drivers. The AMD RX480 GPU is placed in a separate system with the A10-7890K APU. We use the AMDGPU-Pro 16.40 driver for the GPU and the fglrx 15.201 driver for the APU. Each system is running Ubuntu 14.04 and uses the boost library 1.61.

We used `nvprof 2.0.28` and `CodeXLGpuProfiler V4.0.511` for profiling kernel execution times, PCIe transfers, and GPU global memory access. For the measurements of kernel execution times, we profilers to measure individual kernels and sum up the kernel execution times if multiple kernels are involved.

3.8.1 Pipelined Prefix Sum

We compare several pipelined prefix sum techniques to one non-pipelined technique for a query that filters and projects one table. This allows us to analyze the benefit of integrating prefix sum computations into single-pass kernels. We execute the Select/Project-query (shown in Figure 3.16) and vary the selectivity in the range [0, 1] using `x`. By running the experiment on four GPUs, we aim to assess the best local resolution mechanisms for a given hardware. Figure 3.17 shows the results.

Observations Pipelined techniques perform better than Multi-pass in most cases. Integrating the prefix sum computation into single-pass kernels reduces the kernel execution times by factors up to **6.3**×. While processing with Multi-pass takes up to 328.6% of the PCIe time, Resolution:SIMD uses only 101.3% of the PCIe time in

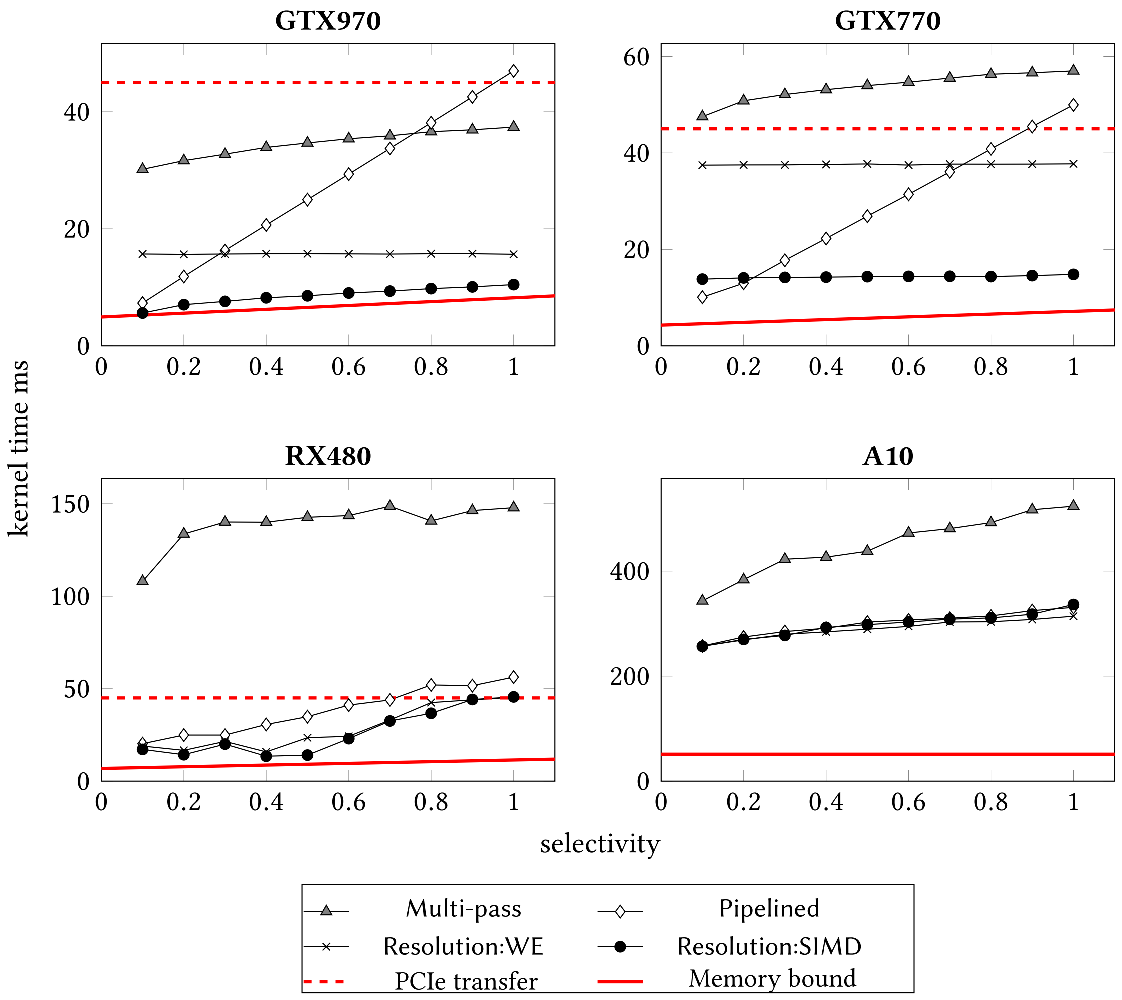


Figure 3.17: Executing the Select/Project-query with different techniques. We achieve the fastest execution times when pipelining the prefix sum computation with local resolution.

the worst case (selectivity 1.0, RX480). This shows that the approach can saturate the bus bandwidth for a variety of configurations. On the A10 there are no PCIe transfers and Resolution:SIMD increases the overall throughput by factors up to **1.6**× over Multi-pass.

The results show that the local resolution step reduces the performance impact of atomic operations. This becomes visible for higher selectivity factors: Pipelined has higher executions times because the strategy executes one atomic addition per output. Resolution:SIMD and Resolution:WE however show good performance across all selectivities due to local resolution.

Resolution:SIMD achieves the shortest kernel execution times in most cases and allows memory bound processing on the GTX970. On the GTX770, lowering the output size down to 0 does not affect the execution time. We conclude that

```
select sum(lo_extendedprice), lo_orderkey % x
from lineorder
group by lo_orderkey % x
```

Figure 3.18: Group By-query inspired by star schema benchmark.

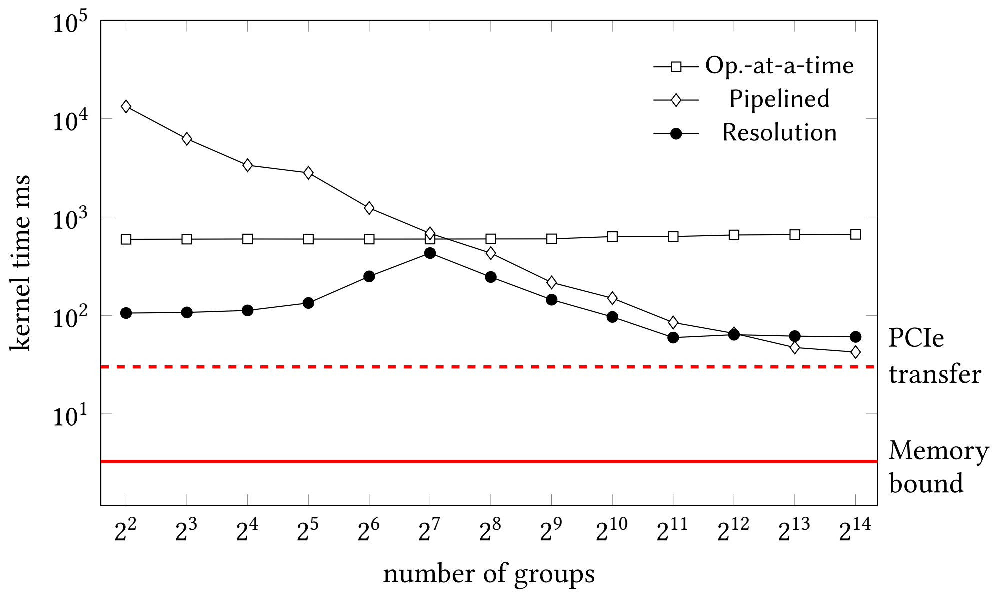


Figure 3.19: Performance of grouped aggregations.

the GTX770 is compute-bound earlier than the GTX970. The higher memory bandwidth of the GTX770 leads to an increased throughput for atomic operations and Pipelined can outperform Resolution:SIMD for selectivities below 10%. On the RX480 and on the A10 there is no definite advantage for one of the reduction techniques. In the following, we only use Resolution:SIMD and skip the other techniques for a clear presentation.

3.8.2 Pipelined Group By

We evaluate the effect of pipelined GROUP BY aggregations using different techniques. We execute the Group By-query (shown in Figure 3.18) with Operator-at-a-time, Pipelined and Resolution. The query groups all tuples of lineorder according to the computed attribute lo_orderkey%x into sums. We vary the number of groups by increasing x from 2 to 16384. We show the results of the experiment on a GTX970 GPU in Figure 3.19.

Observations The execution times of Operator-at-a-time do not depend on the group size. The main cost factor is sorting the input columns. Pipelined shows up to **11.1**× lower execution times but only for larger group sizes. For group sizes below 64, we observe high execution times. This is caused by heavy contention of parallel aggregation hash table inserts.

The bottleneck is resolved by Resolution which uses pre-aggregations to reduce the contention. The results show that execution times reduce by factors of up to **126×**. However, the local pre-aggregations have a limited effect on larger group numbers. This explains the spike at 128 groups, where both pre-aggregation and contention have an effect. While the approaches cannot saturate PCIe when aggregating a full table, filters reduce the cost of grouping for real-world queries.

3.8.3 Star Schema Benchmark

The previous experiments showed that pipelining specific reduction operations helps to increase the throughput of query processing. In this experiment, we analyze whether this behavior carries over to real-world situations. To this end, we execute the SSB Queries[4] on the GTX970 GPU.

We use Operator-at-a-time and two variants of our query compiler. HorseQC: Multi-pass uses pipeline breaking implementations for reductions. HorseQC: Fully pipelined integrates all pipeline operations in one kernel. We show the results of the experiment in Figure 3.20.

Observations The bandwidth analysis in Section 3.2.1 showed that 4 out of 12 queries are limited by GPU global memory access in operator-at-a-time processing.

- The kernel execution times of Operator-at-a-time show that compute and latencies further increase the problem. While PCIe would allow execution times between 60.6*ms* to 90.9*ms*, the kernel execution times take longer for 10 out 12 queries with up to 295.5%.
- HorseQC: Multi-pass improves over Operator-at-a-time and uses only 50.5% of the PCIe bandwidth transfer time in the best case and 215.5% in the worst case. This shows that without efficient pipelining of reduction operations, the benefit of query compilation is limited.
- HorseQC: Fully pipelined lowers all kernel execution times to a level that is consistently lower than PCIe transfer times. This shows that compiling pipelines into one kernel with local resolution, global propagation provides an execution approach with sufficient throughput. Processing takes 9.7% of the PCIe transfer time in the best case and 78.1% in the worst case. For Queries 1.1, 1.2 and 1.3 kernel execution is memory bound by GPU global memory access.

[4]We could not process SSB Query 2.2 as we do not support range predicates on dictionary compressed columns yet.

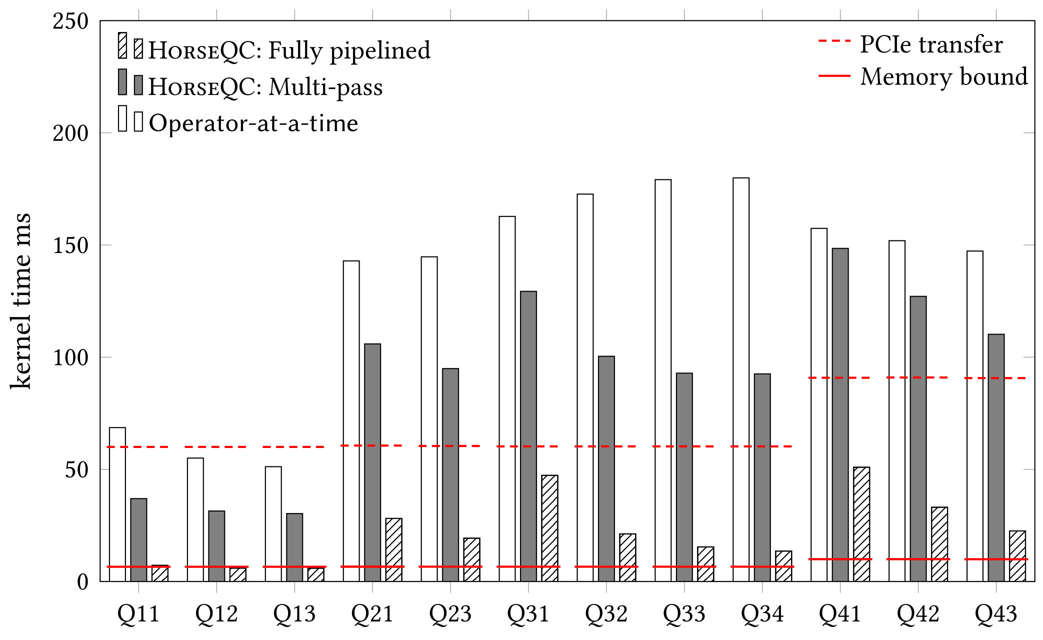


Figure 3.20: Performance of SSB queries.

3.8.4 TPC-H Queries

We execute and profile queries from the TPC-H benchmark to show the effect when relaxing the specific assumptions of the star schema benchmark (e.g. using one centralized table). We select a subset of queries based on the work by Boncz et al. [13] to capture challenging aspects of the TPC-H benchmark, i.e., Q1, Q4, Q13, and Q21 contain heavy aggregation, Q9, and Q18 contain heavy joins, and Q4, Q19, and Q21 contain parallelism bottlenecks. We modified 4 queries, because HorseQC currently does not support all operations.[5] The results of the experiment are shown in Figure 3.21. For Q1, there is no result for HorseQC: Multi-pass, because the strategy ran out of GPU memory. The results shown for Operator-at-a-time are for all TPC-H queries supported by the DBMS.

Observations The PCIe and memory bound baselines show larger variations than for the SSB benchmark. This is mainly caused by the join structure, e.g., Q13 joins three small tables, while Q17, Q18, and Q21 join multiple instances of the largest `lineitem` table.

The kernel execution times show that HorseQC can improve over operator-at-a-time by factors of up to **8.6x**. For Q1, Q4, and Q9, there are cases where Operator-at-a-time has shorter kernel execution times than compiled strategies.

[5]We kept seven TPC-H queries (1, 4, 5, 6, 7, 18, 19) unchanged and slightly modified four (9, 13, 17, 21). E.g. we replaced `like`-predicates, with simple predicates and preserved the query's original selectivity.

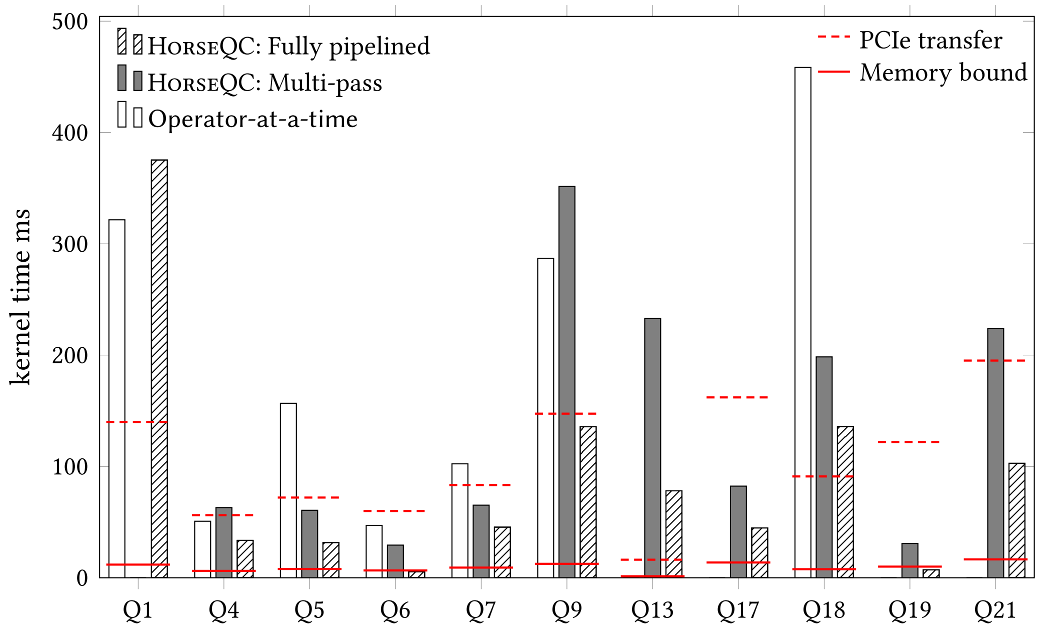


Figure 3.21: Performance of TPC-H queries.

Further investigation showed that in these cases Operator-at-a-time moves some operators to the CPU, therefore the measurements cover a limited amount of operations.

Comparing the variants of the query compiler, we observe that HorseQC: Fully pipelined consistently improves over HorseQC: Multi-pass by factors of up to **5.4**×. HorseQC: Fully pipelined achieves lower execution times than PCIe transfer times for 8 out of 11 queries. For Q1, Q13, and Q18 the PCIe bandwidth cannot be fully saturated. This is because the queries contain grouped aggregations of unfiltered columns (cf. Pipelined Group By). The execution times of HorseQC: Fully pipelined take 5.6% of the PCIe transfer time in the best case and 268.1% in the worst case.

3.8.5 Scalability

Due to the deeply integrated storage layer implementations of the host DBMS CoGaDB, we were not able to build a fully scalable version of HorseQC. For this reason, we perform a separate experiment that integrates the Resolution micro execution model with the batch processing macro execution model for the star join from SSB Query 3.1. Decoupling this experiment allows us to apply the rules for coprocessor data management by Yuan et al. [104] and to measure end-to-end performance for larger datasets.

The star join recombines three dimension tables and one fact table with an overall selectivity of 3.4%. We build hash tables for the dimension tables in GPU

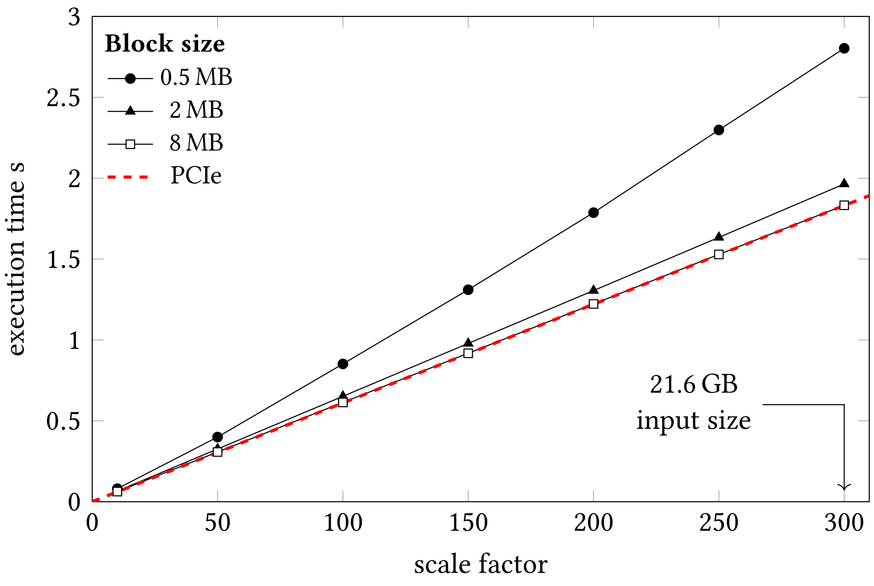


Figure 3.22: End-to-end performance of star join computation for different scale factors.

global memory. The fact table resides in pinned host memory and each column is partitioned into blocks of 0.5 MB, 2 MB or 8 MB. The blocks are transferred asynchronously via PCIe into an inner kernel that computes the star join by probing each dimension hash table.

Figure 3.22 shows the end-to-end execution times for each block size when executing the experiment. We observe that execution times grow linearly with increasing scale factors and that block sizes larger than 2 MB can saturate the PCIe bandwidth. The computation does not become a bottleneck for the examined scale factors. With a block size of 4 MB and scale factor 300, the size of intermediate data in GPU global memory is only 473 MB. Therefore, we expect the approach to scale to even larger databases with linear performance.

3.8.6 End-to-End Performance

To make a comparison to other database systems, we execute the TPC-H queries with different database systems and measure end-to-end performance. We compare MonetDB5 Dec2016-SP3 executed on CPUs, and CoGaDB 0.41 and HorseQC executed on GPUs. Both competitors feature an operator-at-a-time approach. We perform the measurements with warm caches. MonetDB runs on a workstation-class system with an Intel Xeon E5-1607 CPU and 32 GB RAM. CoGaDB and HorseQC run on the GTX970. The results are shown in Figure 3.23.

Observations For the supported queries, HorseQC is up to **5.8**× faster than CoGaDB. While CoGaDB uses GPU global memory as a cache for frequently used columns, HorseQC does not cache data between queries. This shows that HorseQC uses memory and interconnects more efficiently. For Q6 there is no improvement, because query execution is PCIe bound.

HorseQC has lower execution times than MonetDB by factors of up to **26.9×**. Despite moving data through the PCIe bottleneck, the additional bandwidth resources of GPU global memory offer an acceleration. For Q19 MonetDB has a lower execution time than HorseQC. This shows that for queries with a low complexity, it is more effective to process data directly than moving it over PCIe.

3.9 Discussion

In these experiments, we evaluated our new approaches to query compilation on coprocessors. Across all experiments, we were able to show improvements of query compilation over operator-at-a-time processing. Operator-at-a-time has a low memory efficiency due to large materialization volumes and repetitive operations. The approach therefore cannot utilize the memory systems surrounding the coprocessor efficiently.

While naive compilation techniques increase the memory efficiency, reductions and prefix sums split operator pipelines into multiple passes. In this way, the approach inherits the drawbacks of operator-at-a-time. This becomes visible because kernel execution times frequently exceed PCIe transfer times.

This chapter shows a query compilation technique that merges the operators of a pipeline into one compound kernel. When combined with efficient reduction techniques, the compound kernel achieves substantial advantages over other processing approaches. With upcoming OpenCAPI and NVLink interconnects, these improvements to GPU-local processing are essential to benefit from increased bandwidth of the new hardware. In the evaluation setting, the PCIe bandwidth can be saturated for all SSB queries. For the TPC-H benchmark, the approach improves over operator-at-a-time and naive compilation, but saturates PCIe only 8 out of 11 queries. We conclude that the compound kernel works particularly well with star join queries.

3.10 More Related Work

Combining multiple kernels for query processing on GPUs has been used in related work. Wu et al. [99] analyze query plans to automatically fuse kernels with matching I/O data. Li et al. [57] use pre-fabricated kernels that recombine several operators.

Our approach to pipeline the computation of write positions produces data this is not strictly ordered but still contains locality. Such partially ordered data has been examined in the context of the Diag-Join by Helmer et al. [41].

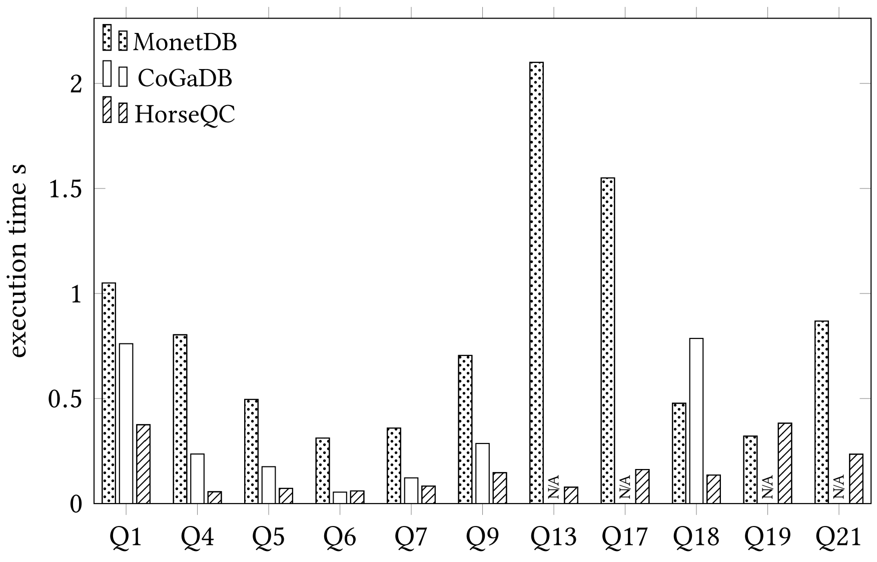


Figure 3.23: End-to-end performance of TPC-H queries.

Query compilation can be applied in higher-level languages for programmability [51] or in lower-level languages for low compilation times [69]. Similarly, on GPUs lower-level PTX or SPIR code may be used or higher-level languages may help to abstract hardware details.

With the end of frequency scaling, it has become increasingly important to exploit hardware parallelism. Power et al. [83] show that especially integrated GPUs can achieve better processing efficiency than CPUs.

In related work, two ways to compute single-pass prefix scans have been proposed. They are similar to local resolution, global propagation with different approaches to pipeline global propagation. First, Yan et al. [102] serialize the computation of local prefix sums with memory barriers. Second, Merrill et al. [65] propose a dynamic look-back mechanism that recomputes unavailable partial sums. In contrast, we use atomic operations to avoid re-computations of long pipelines and to facilitate out-of-order execution.

3.11 Engineering Query Compilers

As in the previous chapter, we discuss the lessons learned for the design of query compilers. In this section, we address balancing of compute and memory resources. We consider a compiled pipeline that is used for query execution as example. The pipeline has a data volume that is accessed in memory and a compute workload that is processed by the processor. If the data volume and the compute workload are balanced (i.e. at the tip of a roofline diagram [73]) both memory and compute resources work at their highest capacity. This is a beneficial situation as none of the resources delays the other.

If we find a way to use memory resources more efficiently, the data volume reduces. However, with the same compute workload there is no advantage to the processing speed, because compute limits throughput. This shows that resource efficiency should not be the only goal. Rather to achieve significant improvements, it is important to improve on the resource that is the most limiting factor. This is the predominant approach we have persued in this chapter. For instance in Section 3.2.3, we found that standard processing techniques are limited by GPU global memory bandwidth and introduced compilation techniques in Section 3.4 and 3.5 to lower the accessed data volumes in this memory.

In some cases, however, there are no viable options to improve on the limiting resource directly. Then it can be possible to spend additional work on a resource with open capacities to the benefit of the limiting resource. This approach was used in Section 3.6, where the atomic functional units of the GPU limited performance. As GPUs have sufficient compute capacity, we were able to add computations that take away work from the atomic functional units. This served as a beneficial way of balancing resources.

3.12 Summary

In this chapter, we showed query processing techniques that help to balance the data movement cost and the compute throughput on GPU-style coprocessors. We measured the data transfer volumes in different scalable processing approaches to assess bandwidth bottlenecks. While naive scalable execution techniques are limited by PCIe bandwidth, batch processing is limited by GPU-local throughput. To address the bottleneck, we proposed micro execution models that benefit from on-chip pipelining. Naive query compilation techniques allow simple code generation but inherit the memory-intensity of operator-at-a-time. We introduced compound kernels that merge several pipeline phases into one efficient kernel.

4 Processing

The previous chapter introduced techniques to transfer the query compilation approach to data-parallel coprocessors, such as GPUs. The presented techniques largely improve the communication efficiency. However, the changes in the processing model also have a strong effect on the processing efficiency. Generally query compilation affects the processing efficiency positively: it eliminates interpretation work, it reduces the amount of executed memory instructions, and it improves the ability to apply compiler optimizations. By switching to the data-parallel processing model, however, there are also negative effects on the processing efficiency. Every GPU instruction handles several elements simultaneously (e.g. 32 elements). Therefore tuples can no longer be processed independently and variations in the amount or type of processing per tuple lead to execution with reduced efficiency. This effect, called *control flow divergence*, is well known, but it has a particularly strong effect on query compilation techniques. This is because divergence effects are amplified during the execution of fused pipeline operators.

In this chapter, we identify two types of control flow divergence—*filter divergence* and *expansion divergence*—that frequently occur in real world workloads. We quantify the problem for two poster cases and propose techniques to balance these divergence effects. By balancing divergence effects, our approach is able to restore processing efficiency even when pipelines contain heavily skewed operations. Our query compiler DogQC has a wider range of functionality than other query coprocessors *and* achieves performance improvements. We observe shorter execution times for TPC-H benchmark queries by factors up to 4.51× compared with existing GPU query compilers and by factors up to 4.54× compared with CPU-based systems.

Parts of this chapter are contained in published articles [31, 32].

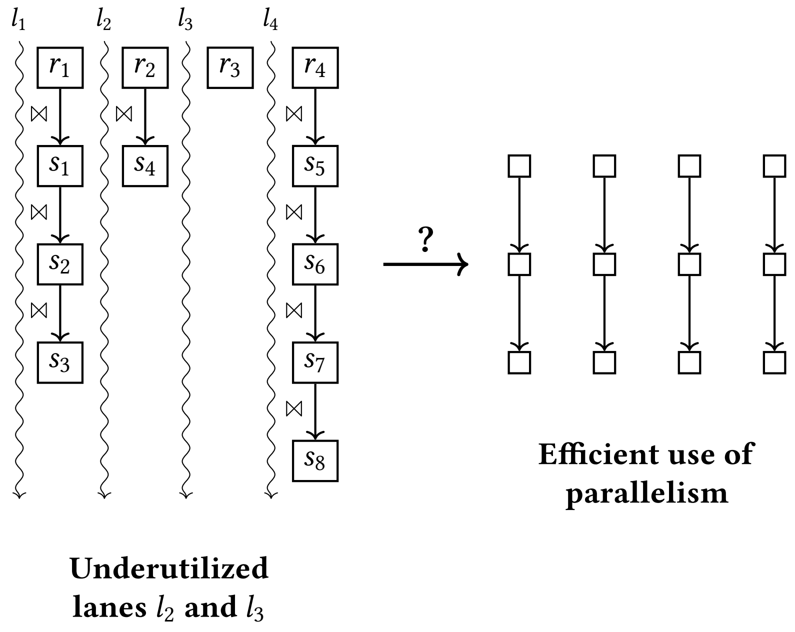


Figure 4.1: Data-parallel computation of $R \bowtie S$ with inefficient use of compute resources due to non-uniform distribution of S.

4.1 Introduction

Data-parallelism is frequently used for efficient query processing (e.g. SIMD, co-processors). As means of specialization, it is a way to overcome the *power wall* that limits the design of modern multiprocessors [17]. Instead of dedicating chip resources to control flow management, data-parallel architectures target throughput. For instance, executing an instruction for 32 fields at a time reduces control flow management work by 32× compared to scalar execution.

Leveraging data-parallelism in a beneficial way can be challenging. While uniform data can be processed naturally, irregular data and computation patterns may compromise the benefits. In the uniform case, it is sufficient to package data into parallel lanes and then to run an instruction sequence. Non-uniform data, however, cannot easily be packaged into a fixed number of fields and the instruction sequences may *diverge*. Consequently, for irregular problems, data-parallel operations execute with reduced efficiency.

Figure 4.1 illustrates the problem for a database join operation. While rows r_1 and r_4 find three/four join partners, there is only a single join partner for r_2 and none for r_3. A naive data-parallel execution, therefore, will leave execution lanes l_2 and l_3 underutilized. In real-world problems, unfortunately, such irregularities are the norm, rather than the exception, e.g.

Variable Length Data The size of an attribute may vary across different entities (e.g. strings).

Skewed Distributions Skewed data distributions lead to divergence during recombination tasks (e.g. joins).

Computation Divergence As a secondary effect of data properties, divergence may occur during computations (e.g. hash collisions).

4.1.1 State of the Art

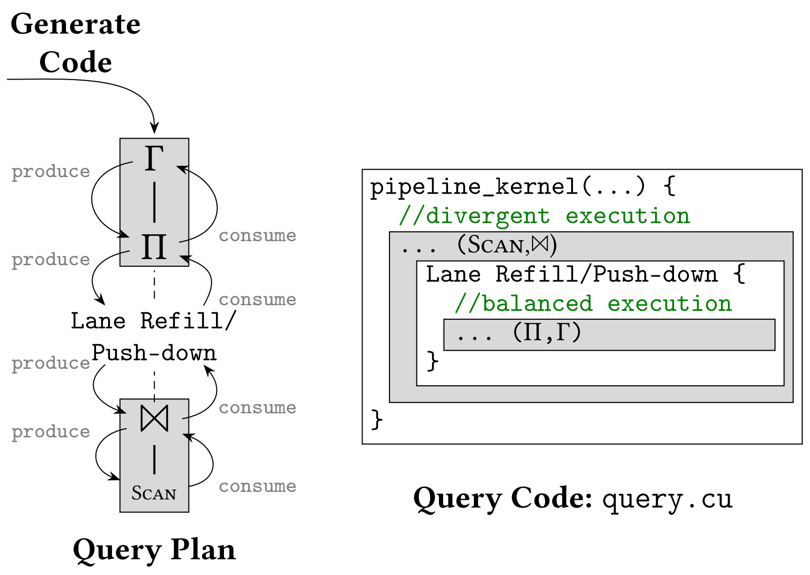
Non-uniformness can be particularly harmful to parallel *query compilation* approaches. Query compilation closely entwines sequences of operators (*pipelines*) into native code. Thus non-uniform effects that occur in the data-parallel execution of one operator may be amplified during the execution of successive operators. In CPU-based systems, the problem of data-parallelism in compiled pipelines has been addressed by database researchers [56, 90]. A promising approach by Lang et al. [54] refills inactive SIMD lanes with buffered elements from previous low-activity iterations.

By contrast, in the context of data-parallel accelerators—such as GPUs—existing systems tend to circumvent the problem of non-uniformity at a high price. E.g., they use string dictionaries [67, 11, 18, 40, 28], specialized joins [78], materialization barriers [99, 40, 18], or bit-packed keys [18, 19] to provide a uniform surrogate. The surrogate, however, usually has limited expressivity, and query coprocessing engines struggle to match the same range of operations supported by their CPU counterparts.

4.1.2 System: DogQC

We use our query compiler *DogQC* to illustrate our techniques to cope with non-uniformity on data-parallel processing devices. DogQC performs *Just-in-Time compilation* (JIT) with standard template-based code generation [69], which we apply to GPUs with the techniques established in Chapter 3. The mechanisms to cope with non-uniformity are *orthogonal* to other GPU-based query processors.

The approach is illustrated in Figure 4.2, which shows an operator with `Lane Refill/Push-down` for divergence balancing that is placed between standard relational operators (shown in gray). During JIT-compilation the query plan on the left is translated to the query code on the right. For the balancing operator, DogQC instantiates a code template that is weaved into the code for relational processing. The mechanism resolves imbalances in the code of the operators with divergence effects (outer gray box) before continuing with the execution of succeeding operators (inner gray box). In this way, the succeeding operators are executed with increased processing efficiency.



```
pipeline_kernel(...) {
  //divergent execution
  ... (SCAN,⋈)
    Lane Refill/Push-down {
      //balanced execution
      ... (Π,Γ)
    }
}
```

Figure 4.2: Injecting divergence balancing into query code generation.

By balancing divergence effects, DogQC targets efficient query processing without assumptions about the uniformness of workloads. This allows DogQC to achieve a larger range of functionality and to avoid expensive preprocessing steps that are typically used to harmonize data.

4.1.3 Contributions and Outline

Our work is the first to pinpoint the problem of divergence in the context of GPU-accelerated database processing (Section 4.2). We identify two flavors of divergence: *expansion divergence* (Section 4.3) and *filter divergence* (Section 4.4). With *Push-down Parallelism* (Section 4.3.2) and *Lane Refill* (Section 4.4.2), we provide novel and effective mechanisms to counter the two divergence effects. In an extensive set of experiments (Section 4.5), we demonstrate how Push-down Parallelism and Lane Refill can speed up query processing by more than a factor of two for realistic benchmarks. To round up the chapter, we discuss related work in Section 4.6, discuss engineering aspects of our approach in Section 4.7, and summarize in Section 4.8.

4.2 Non-Uniform Pipelines

Data-parallel processing of non-uniform data encounters the following problem: Some data elements need a different amount or kind of processing than others. Consequently, parallel lanes need to diverge to follow their tuples' processing path. Due to this effect, called *control flow divergence*, (short: divergence) the affected lanes may idle, or unmatched execution paths are sequentialized. The advantage of data-parallelism to reduce the amount of control flow work is compromised.

Control flow divergence is particularly harmful in kernel-programs[1] that execute operator sequences (e.g. $\mathrm{op}_1 \ldots \mathrm{op}_n$) as they are typical in compiled query pipelines [28]. If the operator op_i introduces divergence, the subsequent operators $\mathrm{op}_{i+1} \ldots \mathrm{op}_n$ may suffer from it as well. For example, a tuple that is filtered out should be disregarded by the following operators, leaving the respective lane idle throughout.

In the following, we take the TPC-H benchmark [13] and analyze the divergence effects that occur in actual query pipelines. We differentiate between two types of control flow divergence, called *filter divergence* and *expansion divergence.* Their difference is based on properties of the operation they originate from.

4.2.1 Lane Activity

Data-parallel processors execute instructions on multiple lanes at a time, e.g. GPUs execute instructions in warps of 32 lanes. Starting with scan, each warp reads the attribute data for 32 tuples into an on-chip register file [94, 72]. Each of the warp lanes is responsible for one scanned tuple and we call the lane *active* when it holds at least one tuple to pass on to the next operator. In subsequent operators, lanes may resign from their tuple, e.g. by applying a filter. However, warp instructions will still compute a value for these *passive* lanes, but the result is discarded. Passive lanes do not contribute to the computation, but cause dissipation of chip resources for register allocation and instruction execution. To achieve a high execution efficiency, it is important to minimize the number of passive lanes.

4.3 Expansion Divergence

Expansion divergence occurs in operators such as string comparisons and joins, where parallel lanes need to process varying amounts of work items depending on data properties. Expansion divergence can lower the execution-efficiency due to divergence in the operator itself (e.g. comparisons of short strings finish early) and due to divergence in subsequent operators. The latter occurs when the expansion process creates a varying amount of new tuples, e.g. join matches.

4.3.1 Poster Case 1

TPC-H query 10 contains a join between the `orders` and `lineitem` tables. Both tables are filtered, therefore optimizers may decide on `orders` $\bowtie$ `lineitem` or `lineitem` $\bowtie$ `orders`. For the latter DogQC computes a hash join with `lineitem` as build relation and `orders` as probe relation. During probe, the tuples from `orders` have varying numbers of matches, which correspond to the items in an

[1]Parallel GPU procedures, called kernels in short.

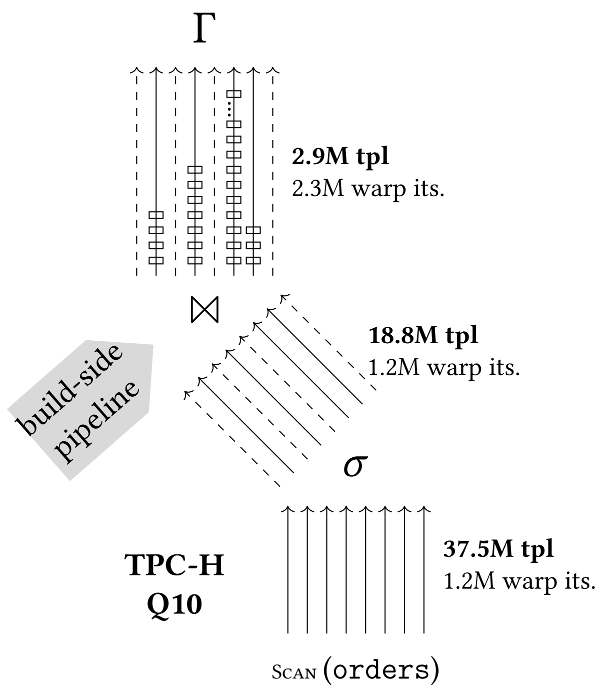


Figure 4.3: Analytic benchmark query with expansion divergence in join operator. Varying numbers of join matches cause more warp iterations for fewer tuples.

order. Producing the matches is a process with expansion divergence. To analyze the execution efficiency, we execute the query with DogQC and look at two metrics at each pipeline stage: The number of tuples and the number of *warp iterations*. The number of warp iterations indicates how many times a warp of 32 lanes goes through an operation. If at least one element is active, the full warp performs the iteration. However, each iteration can process up to 32 elements.

Figure 4.3 illustrates the compiled pipeline.[2] First, a scan of 37.5 M tuples from `orders`, then selection leaving 18.8 M tuples active, and then join probe producing 2.9 M match tuples. The scanned `orders`-tuples are evenly parallelized and thus processed in $37.5\,\text{M}/32 \approx 1.2\,\text{M}$ warp iterations. Selection has the same number of warp iterations because almost all warps have remaining tuples. The following join probe produces a lower number of 2.9 M tuples but requires a higher number of 2.3 M warp iterations. Each lane iterates through varying match numbers and only $2.9\,\text{M}/2.3\,\text{M} \approx 1.3$ lanes per warp are active on average. In an ideal setting only $2.9\,\text{M}/32 \approx 0.1\,\text{M}$ warp iterations would be sufficient. Expansion divergence that occurs in the join probe operator causes a low execution efficiency.

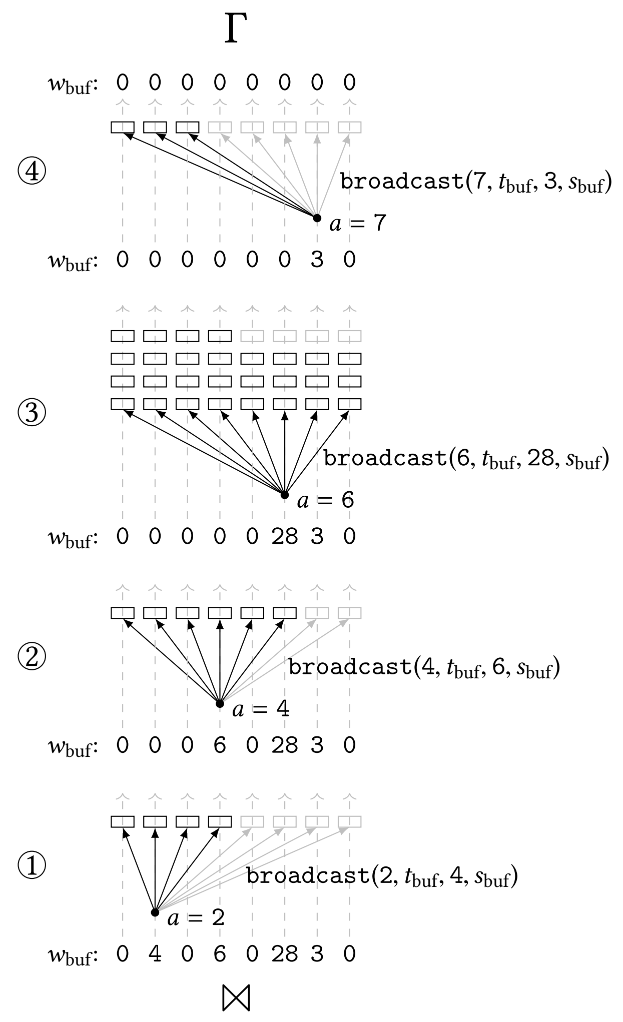
[2]Figures 4.3, 4.4, 4.6, and 4.7 use a lower number of 8 warp lanes for illustration purposes. The actual hardware in this work uses 32 warp lanes.

4.3.2 Push-down Parallelism

Existing query compilers [56, 19, 28] parallelize over the scanned table. *Within* each parallelization unit, expansion processes are executed *sequentially*. For example in the join $R \bowtie S$, where $r \in R$ is part of the scanned table, all join matches of r with S are produced by the same thread. This causes inefficiency as lanes diverge along the distribution of join matches. In the worst case the operators op_i to op_n are executed sequentially when all tuples with matches are processed by the same lane.

Push-down Parallelism has the ability to prevent this effect by changing the parallelization strategy *within the pipeline*. For operators with expansion properties, it pushes parallelization down one level to the expansion process. E.g. for joins, the parallelization level moves from parallelizing over the scanned tuples of R to parallelizing over the join matches with S. This is achieved with *broadcast operations* that redistribute parallel work.

We describe how Push-down Parallelism redistributes work to prevent imbalance caused by join expansion. Figure 4.4 illustrates this and we formalize the mechanism as pseudocode in Figure 4.5. Before applying Push-down Parallelism, warps have gone through the previous operators op_1 to op_{i-1} (lines 1–5). Now, op_i is a hash probe that expands varying numbers of join matches per probe. Push-down parallelism performs the following steps. First, the number of join matches in each lane $w = [0, 4, 0, 6, 0, 28, 3, 0]$ is determined (line 6). Next, the state of each lane consisting of w, the tuple t, and data structure state s is written to local buffer variables w_{buf}, t_{buf}, and s_{buf} (lines 7 and 8). Then Push-down Parallelism enters a sequence of broadcast operations ① to ④ (lines 9 to 16) that finishes when no lane has remaining expansion items. For broadcast ①, Push-down Parallelism selects lane $a = 2$ with $w_{buf} = 4$ join matches as source •. The broadcast takes the values w_{buf}, t_{buf}, and s_{buf} and propagates them from lane 2 to the other lanes of the warp (line 11). Thus all warp lanes retrieve the probe-side tuple with it's current state. The build-side tuples are now retrieved from the hash bucket. The join matches ▭ are processed in a loop with adjacent hash buckets offsets for adjacent lanes (cf. lines 12 to 14). Push-down parallelism performs one loop iteration with the hash bucket offsets $e = [0, 1, 2, 3, \text{x}, \text{x}, \text{x}, \text{x}]$. During the iteration the subsequent operators op_{i+1} to op_n are executed (line 14). Now the hash probes from lane $a = 2$ are finished. This is marked by updating $w_{\text{buf}} = [0, 0, 0, 6, 0, 28, 3, 0]$ (lines 15 and 16). Push-down Parallelism continues with broadcast ②, which starts with the selection of lane $a = 4$ with 6 matches (line 10). The remaining broadcast procedure is unchanged and finishes by updating $w_{\text{buf}} = [0, 0, 0, 0, 0, 28, 3, 0]$ (lines 15 and 16). Broadcast ③ processes lane $a = 6$ with 28 matches. Here the larger number of matches necessitate 4 iterations of the loop in lines 12 to 14. The iterations process 8, 8, 8, and 4 matches. The broadcast finishes by updating $w_{\text{buf}} = [0, 0, 0, 0, 0, 0, 3, 0]$ and leaves 3 matches in lane $a = 7$ for broadcast ④. The join matches are processed, $w_{\text{buf}} = [0, 0, 0, 0, 0, 0, 0, 0]$ is updated, and the loop from line 9 exits. The pipeline starts over with fresh tuples.



Γ
w_{buf}: 0 0 0 0 0 0 0 0
④
broadcast(7, t_{buf}, 3, s_{buf})
$a = 7$
w_{buf}: 0 0 0 0 0 0 3 0
③
broadcast(6, t_{buf}, 28, s_{buf})
$a = 6$
w_{buf}: 0 0 0 0 0 28 3 0
②
broadcast(4, t_{buf}, 6, s_{buf})
$a = 4$
w_{buf}: 0 0 0 6 0 28 3 0
①
broadcast(2, t_{buf}, 4, s_{buf})
$a = 2$
w_{buf}: 0 4 0 6 0 28 3 0
$\bowtie$

Figure 4.4: Illustration of Push-down Parallelism that expands the join matches of four warp lanes one after another.

Each broadcast takes the join matches from an individual lane and spreads them out across the warp. As consequence warps parallelize over the join matches instead of parallelizing over the scan table. This balances expansion processes *and* increases the memory efficiency for hash bucket reads. Push-down Parallelism yields preferable *coalesced memory access* patterns, which means that adjacent lanes access adjacent memory locations [43], whereas the standard approach uses slower sequential memory access.

4.3.3 Implementation

We implement Push-down Parallelism in DogQC by adding a code generation template to the query compiler. The implementation uses *warp primitives* via intrinsics, which allow lanes to exchange data and to perform collaborative compu-

```
PUSH-DOWN PARALLELISM
1  foreach warp of 32 lanes in parallel do
2      $lane_{ix} \leftarrow [1, \dots, 32]$
3      while more inputs do
4          $t \leftarrow$ scan 32 tuples                /* $op_1$ */
5          [...]                          /* $op_2$ - $op_{i-1}$ */
6          $w \leftarrow$ number of work items
             after expansion in $op_i$
7          $s \leftarrow$ data structure state $op_i$
8          $t_{buf}, w_{buf}, s_{buf} \leftarrow t, w, s$
9          while warp_any($w_{buf} > 0$) do
10             $a \leftarrow$ select_leader($w_{buf}$)
11             $t, w, s \leftarrow$ broadcast($a, t_{buf}, w_{buf}, s_{buf}$)
12             for $e \leftarrow lane_{ix}$ to $w$ by 32 do
13                 process $op_i$ expansion item $e$
14                 [...]                  /* $op_{i+1}$ - $op_n$ */
15             if $lane_{ix} = a$ then
16                 $w_{buf} \leftarrow 0$
```

Figure 4.5: Pseudocode for a pipeline that applies Push-down Parallelism to op_i. The strategy expands op_i with another level of parallelism.

tations [71]. E.g. `__sfhl_sync(..)` performs lane index-based data exchange and `__ballot_sync(..)` computes a predicate bitmask across a warp. We describe the implementation of lane buffering, leader selection, and broadcast operations with these intrinsics in the following.

Buffering Active Lanes Lanes that receive work items during broadcast may already have an active tuple in register. To switch to a new work item, it is necessary to postpone processing of that tuple. This is done by buffering active tuples (line 8) before broadcast and leader selection. The buffer operation is local to each lane (i.e., lanes postpone only their own tuple). Consequently, buffering is as simple as writing each attribute value to a local buffer variable.

Leader Selection During leader selection (line 10), Push-down Parallelism picks one lane as broadcast source and provides its lane index a to the other warp lanes. This is implemented with the following expression using only two warp intrinsics:

```
// select broadcast source lane
a = __ffs(__ballot_sync(w_buf>0,ALL));
```

The first primitive `__ballot_sync(..)` builds a bitmask of lanes that have remaining work items and shares it with all lanes. The second primitive `__ffs(..)`

computes the index of the first 1-bit of the bitmask. The lane with index a is selected for broadcast.

Broadcast Operation The broadcast operation (line 11) takes the buffered data from one lane a and distributes it to the other warp lanes. The following values are broadcasted: the attributes of the tuple $t_{\text{buf},a}$, the number of expansion items $w_{\text{buf},a}$, and the data structure state $s_{\text{buf},a}$, e.g., the hash bucket offset. The following code performs the broadcast for a tuple with two attributes and the hash bucket offset using *warp shuffle primitives*.

```
// gather w_buf, t_buf, and s_buf from lane a
w = __shfl_sync(w_buf,a);                            } w
o_orderdate = __shfl_sync(o_orderdate_buf,a);        }
o_orderkey = __shfl_sync(o_orderkey_buf,a);          } t
c_acctbal = __shfl_sync(c_acctbal_buf,a);            }
bucket_offs = __shfl_sync(bucket_offs_buf,a);        } s
```

The `__shfl_sync(...)` intrinsic takes the payload as first parameter and the source lane as second parameter. All lanes of the warp execute the instruction and obtain data from lane a. After the broadcast, each lane processes a distinct expansion work item (lines 12–14). E.g., hash bucket entries are obtained by adding the expansion index e to the base address of the hash bucket. In this way, warps consume the tuples from the hash bucket in coalesced iterations.

4.3.4 Planning for Push-down Parallelism

In DogQC, we select hash join operators for the application of Push-down Parallelism based on the build attributes. If the build is performed on other attributes than primary keys, hash buckets can contain multiple elements per key. We choose Push-down Parallelism to balance expansion processes during hash probes. For primary key build attributes, matching probes will retreive exactly one tuple. We choose the standard hash join as there is no expansion.

A fully-fledged system will include Push-down Parallelism in cost-based optimization as an alternative join operator. Similar to our current implementation, cost estimates can be based on build attribute statistics.

4.3.5 Usage Scenarios

Push-down Parallelism allows efficient execution of operators with expansion processes. The expansion may produce new tuples as the join in the previous example. Alternatively, expansions can be local and the operator passes on only one tuple, e.g., when processing the characters of string-typed attributes. For the latter case line 14 of the pseudocode in Figure 4.5 moves behind the **for**-loop.

By taking the parallelization level to the same level as the expansion process, Push-down Parallelism gives two main benefits. First, non-uniform distributions of the number of expansion items no longer cause expansion divergence. Second, memory accesses that are performed during expansion are transformed from sequential memory access to coalesced memory access. In the following, we discuss several scenarios for the application of Push-down Parallelism.

Joins Joins between tables with varying key distributions are a poster child for the application of Push-down Parallelism. Existing GPU-based techniques restrict functionality by limiting the number of join matches, join conditions, or attributes stored in the hash table [45, 78, 85]. The restrictions limit divergence effects, but also lack support for important query plan options. DogQC handles varying key distributions, multi-predicate joins, and different payload sizes gracefully by using Push-down Parallelism to balance expansion work.

(Anti-) Semi Joins Push-down Parallelism applies to (anti-) semi-joins with multiple match candidates (e.g., for combinations of equality and inequality predicates). The technique helps to balance the parallel evaluation of match candidates. However, the parallelization can prevent join strategies from early exit once the first match is found.

String Equality Equals operations on string datatypes cause expansion divergence due to varying numbers of characters in the strings. Push-down Parallelism expands the string characters across lanes and compares the characters in parallel. This reduces divergence effects from varying string lengths and increases memory efficiency by loading the string data using coalesced access.

Graph Processing The node degree of real world graphs follows skewed distributions, e.g., power law [20]. Consequently, parallel graph algorithms are challenged by varying amounts of traversal work per node. Existing GPU techniques address these imbalances with node partitioning [58], edge partitioning [24], and compression [88]. Push-down parallelism naturally applies to the problem for relational graph representations.

4.4 Filter Divergence

Filter divergence occurs in operators that inactivate some of the parallel lanes, for example filters and primary key-foreign key joins. The subsequent operations experience a lowered execution efficiency due to lane inactivity. This problem has been addressed by *stream compaction* [10] earlier; however, existing solutions are not suitable for compiled query pipelines because of their use of global synchronization barriers.

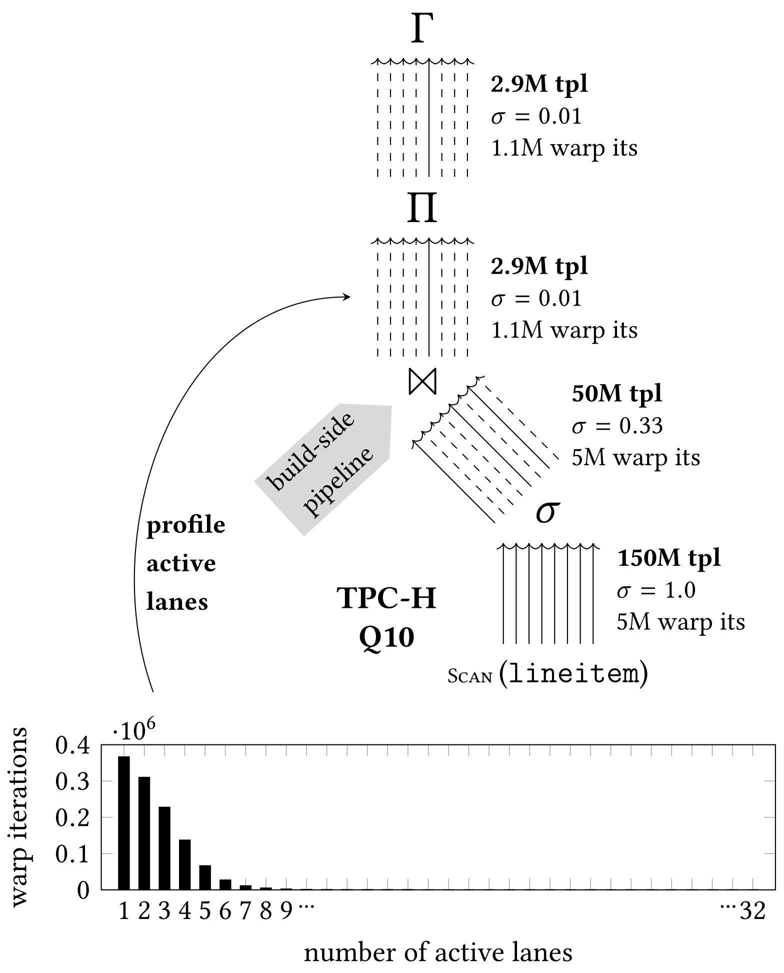


Figure 4.6: Analytic benchmark query with heavy filter divergence. After the filtering join operator most warp iterations have few active lanes.

4.4.1 Poster Case 2

TPC-H Query 10 contains two selective operations on tuples from the `lineitem` table: a selection `l_returnflag = 'R'` and a sparse foreign key join with the condition `l_orderkey = o_orderkey`. Figure 4.6 illustrates a pipeline that scans `lineitem` and then performs selection, join probe, projection, and aggregation. Compared to Section 4.3.1, the pipeline contains an additional projection for `l_extendedprice * (1-l_discount)`. The previous plan performed the projection in the build pipeline favoring a smaller hash table payload.

Again, we look at the number of *warp iterations* (cf. Section 4.3.1) in each pipeline stage to analyze the effect of the filters on execution efficiency. Starting with scan, the pipeline parallelizes 150 M `lineitem` tuples evenly across lanes. This requires 150 M/32 = 5 M warp iterations. The following filter with $\sigma = 0.33$ is likely to leave elements active in each warp. Consequently, the number of 5 M warp

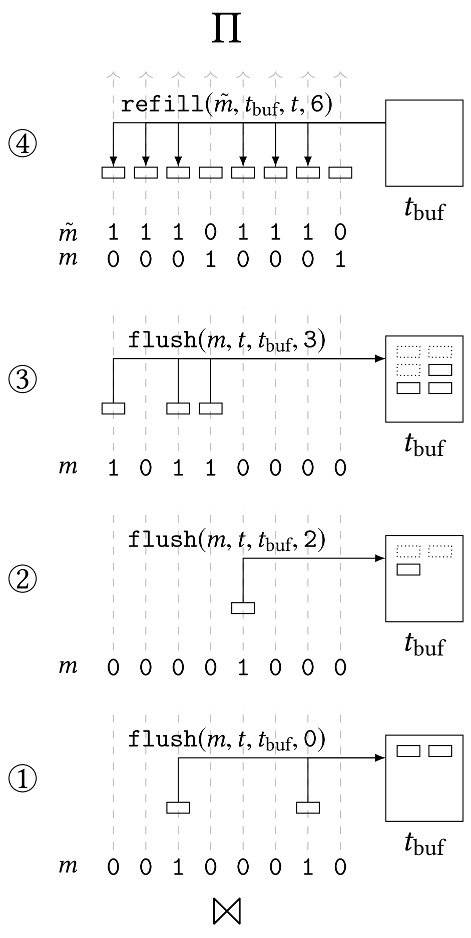

Π
④
refill($\tilde{m}$, t_{buf}, t, 6)
$\tilde{m}$ 1 1 1 0 1 1 1 0
m 0 0 0 1 0 0 0 1
t_{buf}
③
flush(m, t, t_{buf}, 3)
m 1 0 1 1 0 0 0 0
t_{buf}
②
flush(m, t, t_{buf}, 2)
m 0 0 0 0 1 0 0 0
t_{buf}
①
flush(m, t, t_{buf}, 0)
m 0 0 1 0 0 0 1 0
t_{buf}
⋈

Figure 4.7: Illustration of Lane Refill that postpones processing of three low-activity iterations for full lane activity in the fourth iteration.

iterations remains constant. Subsequently, the (single match) join probe produces 2.9 M tuples that are processed in 1.1 M warp iterations. Due to the selectivity of $\sigma = 0.01$ most lanes in the pipeline have become inactive and the remaining tuples are spread across warps. The histogram at the bottom of Figure 4.6 shows a profile of this pipeline stage, illustrating how many active lanes we measured in the 1.1 M executed warp iterations. Only few lanes are active in each warp causing a low execution efficiency that is carried through the subsequent projection and aggregation operators. Ideally, both operators would be processed with only $2.9\,\text{M}/32 = 90\,\text{K}$ warp iterations.

4.4.2 Lane Refill

Selective filters or sparse foreign key joins that trigger filter divergence situations are commonplace in analytic workloads [13]. The *Lane Refill* technique is a nat-

```
LANE REFILL
foreach warp of 32 lanes in parallel do
    $n_{\text{buf}} \leftarrow 0$
    $t_{\text{buf}} \leftarrow$ empty
    while more inputs do
        $t \leftarrow$ scan 32 tuples          /* op_1 */
        [...]          /* op_2 - op_{i-1} */
        $m \leftarrow$ bitmask of active lanes
        $n_{\text{active}} \leftarrow$ popcount($m$)
        while $n_{\text{buffer}} + n_{\text{active}} > \mathcal{T}$ do
            if $n_{\text{active}} < \mathcal{T}$ then
                $n_{\text{buf}} \leftarrow$ refill($\tilde{m}, t, t_{\text{buf}}, n_{\text{buf}}$)
            execute op_i
            [...]          /* op_{i+1} - op_n */
            $m \leftarrow$ bitmask of active lanes
            $n_{\text{active}} \leftarrow$ popcount($m$)
        if $n_{\text{active}} > 0$ then
            $n_{\text{buf}} \leftarrow$ flush($m, t, t_{\text{buf}}, n_{\text{buf}}$)
```

Figure 4.8: Pseudocode for a pipeline with Lane Refill between op_{i-1} and op_i. The control flow proceeds with op_i only for lane activities above threshold T.

ural match to counter the imbalances caused by such operations. The technique we describe here resembles the mechanism proposed by Lang et al. [54] as *consume everything* strategy for SIMD processing. A similar idea was introduced by Polychroniou et al. [81] for a sequence of Bloom-filter bitmasks.

Lane Refill introduces *buffering operators* that control the lane activity during pipeline execution. The buffering operator is designed to work with a given *threshold*. If the lane activity drops below threshold there are two options:

1. There are insufficient buffered tuples. Active lanes are buffered and the pipeline starts over with fresh tuples.

2. There are sufficient buffered tuples to reach threshold and the tuples are reactivated in empty lanes.

This strategy ensures that the operators succeeding the buffering operator always start with a lane activity above threshold. It is worth noting that one element buffer space for each lane is sufficient for any given threshold.

We show the pseudocode for the technique in Figure 4.8 and illustrate it in Figure 4.7. As an example, we assume a Lane Refill operator with threshold 7 (out of 8 lanes) that is placed after the sparse join of TPC-H Query 10. Figure 4.7 shows four iterations ① to ④ of the same warp receiving tuples from the sparse join.

The boxes ▭ represent active lanes holding tuples. The first iteration receives two tuples from the join (pseudocode lines 1–6). Activity lies below threshold and the tuples are flushed to the buffer (lines 9 and 17). The pipeline starts over and the Lane Refill operator receives new tuples from the join. The following two iterations are flushed as well because the highest possible acitivity is 6 (out of 8) for three tuples from join plus three buffered tuples. In iteration ④, there are two fresh tuples and six buffered tuples. The empty lanes are refilled (lines 10–11) and the pipeline proceeds to the following operators with full lane activity. In the following, we show how Lane Refill is implemented in compiled query pipelines on GPUs.

4.4.3 Implementation

We implement Lane Refill in DogQC by introducing a buffering operator with the semantics shown in pseudocode Figure 4.8. The buffering operator is code generation-based, similar to the other operators in DogQC. The main challenges in adapting the approach by Lang et al. [54] are efficient GPU implementations for the balancing operations `flush` and `refill` and the application of warp parallelism.

The previous implementation of Push-down Parallelism performed lane communication via warp shuffles. This was possible because the only communication pattern used were gather operations. Lane Refill, however, uses scatter operations aswell. This is unsupported by warp shuffles, and therefore *shared memory* with communication via array-style indexing is better suited here.

Although shared memory and shuffle registers are both located on-chip, shared memory can perform slower when multiple lanes access the same memory bank [62]. However, further investigation of using warp shuffles only for the gather communication of lane refill showed no significant benefit over using solely shared memory.

Flush to Buffer The `flush` operation is executed when the number of active lanes is below threshold and there are not enough buffer elements to restore sufficient activitiy. The remaining active lanes are written to empty buffer slots. `flush` takes a bitmask of active lanes m, the tuples t, the buffer t_{buf}, and the buffer count n_{buf} as input. Then `flush` computes the buffer destination `dest` that specifies the buffer position for each lane to write its active tuple to. This is done with the following code:

```
// warp prefix sum on active lanes
dest = __popc((m) & (pre_lanes)) + n_buf;
```

We look at an example with 8 lanes and lane activity m = `[0,1,0,0,1,1,0,0]`. The bitmask `pre_lanes` marks all preceding lanes, e.g. lane 4 has `pre_lanes` = `[1,1,1,0,0,0,0,0]`. With the population count intrinsic `__popc(..)`, we count the set bits on preceding lanes. This gives us an exclusive prefix sum of the warp. With $n_{\text{buf}} = 2$ previously buffered elements, the destinations are `dest` = `[x,2,x,x,3,4,x,x]`.

Next, `flush` writes the tuples t from active lanes to the buffer t_{buf} at their respective destinations `dest`. This is done by scattering the tuple's attributes to shared memory, e.g.

```
// scatter to shared memory
l_extprice_buf[dest] = l_extprice;
o_orderdate_buf[dest] = o_orderdate;
```

Refill from Buffer The `refill` operation is executed when the lane activity is below threshold *and* there are sufficient buffered tuples to reach threshold. The operation takes tuples from the buffer and reactivates them in passive lanes. `refill` receives the bitmask of passive lanes $\tilde{m}$, the tuples t, the buffer tuples t_{buf}, and the buffer count n_{buf} as input. To always maintain dense adjacent buffer elements, we push and pop the buffer content like a stack. To this end, we first compute the number of remaining buffer elements `n_remain` based on the buffer count and the number of empty lanes. Then we compute the buffer source index `src` with a warp prefix sum, similar to `flush`.

```
// warp prefix sum on passive lanes
src = __popc((inv_m) & (pre_lanes)) + n_remain;
```

After computing the buffer source index `src`, we can refill passive lanes from the buffer as shown below.

```
// gather from shared memory
if(src < n_buf) {
    l_extprice = l_extprice_buf[src];
    o_orderdate = o_orderdate_buf[src];
}
```

The code reads the attributes of buffered tuples from shared memory locations and stores them in registers by executing assignments to local variables. Note that we only load tuples from the buffer for the first n_{buf} passive lanes to account for the number of buffer elements.

4.4.4 Planning for Lane Refill

In the current version of DogQC, Lane Refill operators are placed manually into query plans. We insert balancing operators into pipelines with heavy filter divergence *if* there are multiple succeeding operators that can benefit from balanced processing. Section 4.5.4 specifies the queries where Lane Refill was applied in the context of the TPC-H benchmark.

In a fully-fledged system, optimizers will select insertion points for Lane Refill operators based on selectivity estimation. As the balancing operations can be executed with a low overhead, the negative impact of estimation errors is small. Optimizers that consider interesting orders, are affected by order changes due to balancing. Such optimizers, may leverage their ability to consider both plans with and without interesting orders during optimization.

4.4.5 Usage Scenarios

Lane Refill restores balanced lane activity in sequences of operators with filter divergence. The technique can be used after an operator that leaves execution in divergent stage (e.g. selection) before continuing with the next operator. Alternatively, Lane Refill can be used in succeeding iterations of the same operator (e.g. character comparisons in string equality) to restore lane activity between iterations. For the latter application, Lane Refill has the beneficial property to *preserve sequential order* of the iterations. This property is contrary to Push-down parallelism which parallelizes iterations. The sequential order can be leveraged by operators, such as regular expression matching with automata, where each iteration is dependent on the previous iterations. In the following we discuss several usage scenarios for Lane Refill.

Selection Selection operators are a poster child for filter divergence. Database systems usually perform selection push-down to reduce workload sizes early. However, in data-parallel pipelines, the early selection does not reduce the workload size. Unless the full warp exits, lanes with filtered-out tuples still allocate the same processing resources. By filling the gaps with useful work, Lane Refill scales processing with the workload size.

Filter Join Sparse foreign key joins occur in *both* normalized database workloads and in de-normalized star schema workloads. They recombine relations with only few matches for the join condition and filter out the majority of one of the relations. In the TPC-H benchmark join selectivities are usually around 0.1 and frequently go even lower [13]. Typically join filters go into effect during the join probe and leave lanes idle that have no match. By placing Lane Refill operators after a join probe, we can reactivate these idle lanes and allow them to perform more useful work.

String Pattern Matching Database systems support string pattern matching with `LIKE`-predicates and regular expression (`regexp`) predicates. Most GPU-based systems, however, have very limited pattern matching capabilities, likely because of divergence effects [7, 18, 19, 28, 40]. Still there is existing work on GPU-based pattern matching. There is work on NFA-based `regexp` matchers [106], which parallelize over the states of the automaton. Albeit this parallelization strategy collides with per-tuple parallelization of GPU query engines. Other work on DFA-based matchers [95] uses per-string parallelism, which appears more suitable for query engines. During pattern matching, however, non-matching strings reach rejecting states of the DFA early. Lane Refill can be used to reactivate those lanes with new tuples to make string pattern matching efficient. The property of Lane Refill to preserve sequential order is essential for following state transitions through DFAs.

Index Traversal Index traversals are used to find tuples that match predicates. The hierarchical index structure is traversed from coarse-grained ranges to more fine-grained ranges to localize matching tuples. For regions with sparse population, traversal paths are often shorter than for densely populated regions. This leads to filter divergence during concurrent traversals. While B-Trees have relatively uniform path lengths, other index structures, e.g., for geospatial data [50], show more variation. To support such datatypes efficiently on GPUs, Lane Refill can be used to address these divergence effects during traversal.

4.5 Evaluation

In this section, we evaluate the proposed techniques. We first evaluate the effect of Push-down Parallelism for expansion divergence. Then we evaluate the effect of applying Lane Refill to filter divergence. Next, we contrast Push-down Parallelism and Lane Refill when being applied to the same operation. Finally, we evaluate the overall performance of the divergence-optimized system against other state-of-the-art systems on CPU and GPU. In all experiments except for end-to-end performance (Figure 4.19) the execution times refer to GPU-resident data.

Query Processor We evaluate the presented approach in the GPU-based query compiler DogQC.[3] DogQC follows an orthogonal approach to other GPU query processors. Instead of tuning *operator*-implementations for efficient GPU utilization, DogQC constructs pipelines from relatively simple operators and then applies tuning on the *pipeline level.* This two-step approach makes it more feasible to achieve both functionality *and* performance.

In the evaluation we use two versions of DogQC. The first version executes queries after the first step, which can cause heavy divergence during query processing. We call this version **DogQC naive**. The second version **DogQC opt** executes queries after the second step, which adds divergence balancing to increase processing efficiency on the GPU.

System As experimentation platform, we use an NVidia RTX2080 GPU with 46 Streaming Multiprocessors (SMs) and 8 GB GPU Memory. We use Cuda 10.0 and `nvcc` V10.0.130 for JIT-compilation in all experiments but Figure 4.19, which uses `clang++9.0` to compile Cuda code. When not indicated differently, we use grid configurations of 80 warps per Streaming Multiprocessor (117,760 threads). This choice is due to sufficiently large grid sizes showing only small performance variations (cf. Figure 4.16). The GPU is placed in a workstation-class host system with 32 GB main-memory, operating an Intel Core i7-9800X CPU with Ubuntu 18.04 as operating system.

[3]The source code of DogQC is available at `https://github.com/henning1/dogqc` and at `http://dbis.cs.tu-dortmund.de`

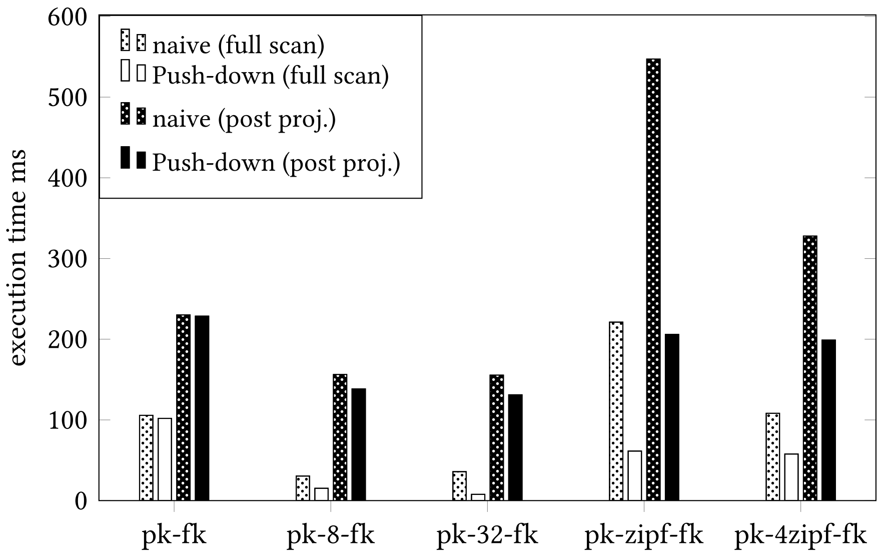


Figure 4.9: Divergence balancing for hash join with different build distributions. Push-down achieves robustness against skew and improves performance.

4.5.1 Effect of Push-down Parallelism

We first evaluate the benefit of Push-down Parallelism for expansion divergence. We execute a query that scans two relations and joins them with different join key distributions. We use a synthetic dataset where one relation has a dense primary key distribution and the other has one of the following key distributions:

pk-fk	Uniform distribution of foreign keys.
pk-8-fk	Each foreign key occurs 8 times.
pk-32-fk	Each foreign key occurs 32 times.
pk-zipf-fk	Foreign keys sampled from Zipfian distribution with $z = 0.75$ and $n = 10^7$.
pk-4zipf-fk	Foreign keys sampled from four Zipfian distributions with $z = 0.75$ and $n = 10^7$.

We generate join workloads for each of the distributions with 100 M build tuples and also 100 M result tuples. The first three workloads have an even number of 1 to 32 join matches per probe. Probes for pk-fk have exactly one match, probes for pk-8-fk have 8 matches, and 32 for pk-32-fk. With even match numbers we expect performance differences mainly due to the access method to hash buckets. The latter two workloads are non-uniform and the number of matches follows Zipfian distributions. The heaviest skew is for pk-zipf-fk with one probe matching the most frequent key 452 K times. For pk-4zipf-fk, the four frequent keys occur 112 K times.

We show the results in Figure 4.9. The Figure reports execution times of the probe pipeline with the naive approach and with Push-down Parallelism for two

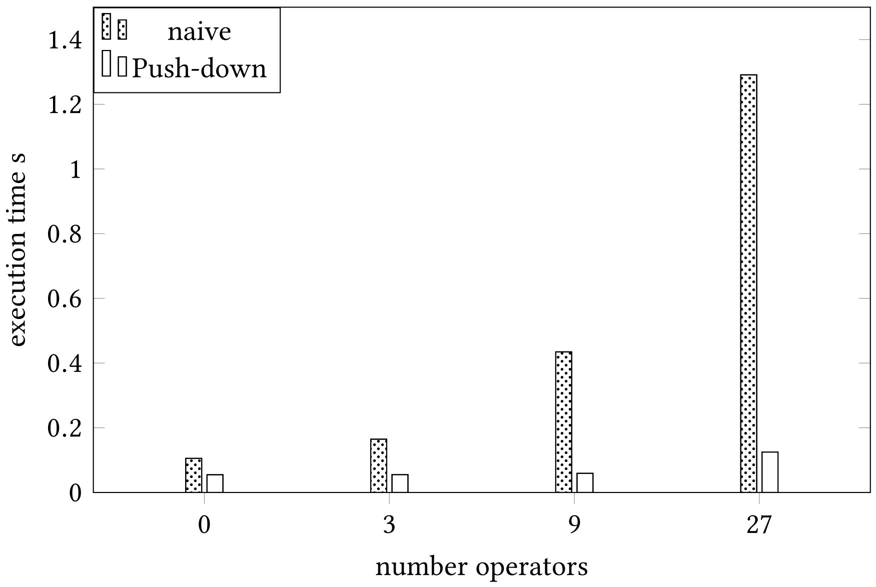


Figure 4.10: Varying numbers of operators after expansion. Without push-down parallelism divergence effects are amplified by the following operators.

different projection strategies. *Full scan* reads all attributes into registers during scan. *Post-proj* performs tuple-id based post projection.

We observe that Push-down Parallelism reduces execution times for all examined workloads by factors up to 4.2×. We discuss two effects that explain these improvements. The first effect is better load balancing across threads, which becomes visible when comparing pk-zipf-fk to pk-4zipf-fk. The workloads have different levels of skew that affect the execution times of naive. Push-down parallelism achieves even execution times for both distributions.

The second effect is due to memory access patterns. Although the probes for pk-32-fk and pk-8-fk do not provoke load imbalance, the execution times improve. We attribute this to coalesced memory access, which means that adjacent lanes access adjacent memory locations in the hash buckets. This pattern is preferable on GPUs [43]. With Push-down Parallelism probes perform coalesced memory access with 8 or 32 lanes when executing pk-8-fk and pk-32-fk.

While we did not expect to observe an improvement for pk-fk, Push-down Parallelism reduces the execution times by 4%. We explain this with an increased efficiency when handling hash collisions.

Looking at the two projection strategies, we observe that Push-down Parallelism provides benefits for both. Push-down parallelism improves by factors up to 2.7× for post-proj and by factors up to 4.2× for full scan. We attribute the higher benefit for full scan to the way Push-Down Parallelism channels tuple data to lanes with new join tuples. For post-proj only the tuple-id communicated via warp shuffles and other attributes are read from memory.

Varying Numbers of Operators We evaluate the effect of varying the workload size that follows expansion divergence. This allows us to assess the impact of processing in divergent or in consolidated state. We append different numbers of projection operators to the pk-4zipf-fk workload and execute with the naive approach and with Push-down Parallism. We use configurations up to 27 operators to evaluate settings with high compute intensity. Figure 4.10 shows the experiment results.

Push-down parallelism improves throughput by factors that increase with the number of operators up to 10.3×. Further investigation showed that increasing the number of operators even further does not lead to higher factors. We attribute this effect to the compute load becoming the dominant part of the workload. The magnitude of the factor appears to be distribution dependent.

Poster Case 1 In Section 4.3.1, we discussed a query pipeline from TPC-H Query 10 with expansion divergence. Here we evaluate the effect of applying Push-down Parallelism in this pipeline to counter expansion divergence. We measure the execution time of the pipeline for a benchmark database with scale factor 25. We use a pipeline with the naive approach that has heavy expansion divergence in the join and we compare it to a pipeline that applies Push-down Parallelism in the join operator to counter expansion divergence. Figure 4.11 shows the experiment results. The naive approach has an execution time of 26.4 ms. Adding Push-Down Parallelism to the join operator of the pipeline reduces the execution time by 1.9× to 13.8 ms.

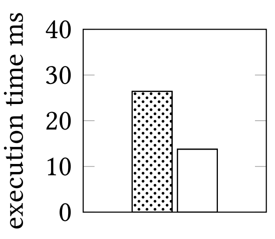


Figure 4.11: Push-down parallelism in Poster Case 1.

4.5.2 Effect of Lane Refill

We evaluate Lane Refill to counter filter divergence. The workload is a query that scans the TPC-H tables `lineitem` and `part`. The `lineitem` relation is filtered on `l_quantity` with *varying selectivities*, and then joined with `part` and aggregated. For Lane Refill, we place a balancing operator after the filter to restore lane activity. GPUs typically *oversubscribe* the number of warps to the number of streaming multiprocessors (SMs). This ability allows GPUs to hide divergence effects to some extent. To understand the way Lane Refill works, we first suppress effects from oversubscription by using only *one warp per SM*. After that we perform another experiment with *multiple warps per SM*.

One Warp per SM Figure 4.12 shows the results of the experiment with one warp per SM. If we set the filter to leave all tuples in the result (selectivity 1.0), we observe an execution time of 423 ms for the naive approach. The query becomes

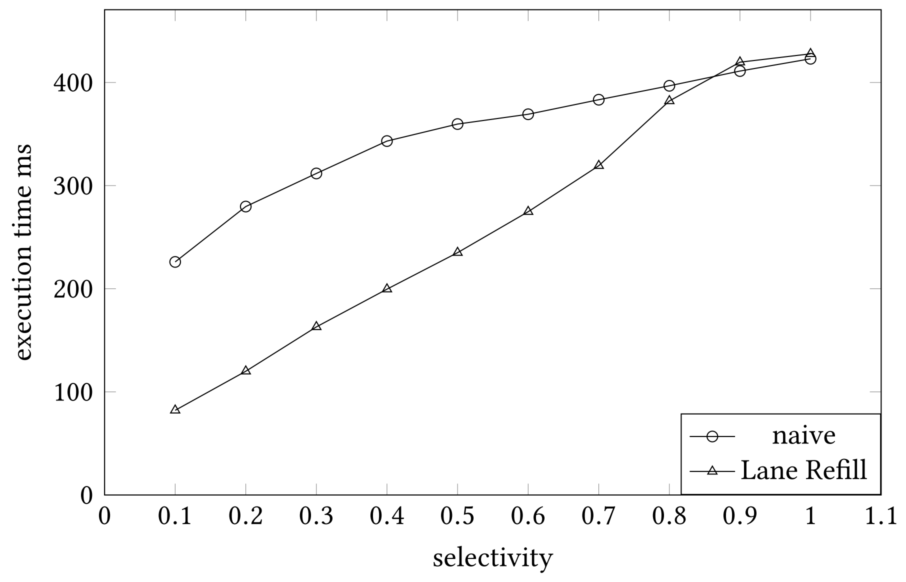


Figure 4.12: Effect of Lane Refill on filter divergence workload with one warp per SM. Execution times scale with the workload size when using Lane Refill.

faster as we make the filter predicate more restrictive. For the naive strategy, that benefit is small, however: setting the selectivity to 0.4 improves performance by only 19% (343 ms). Only for very selective predicates, execution time noticeably drops, as shown in the graph for selectivity 0.1 (226 ms). This is because the naive approach can only benefit from filtering when *full warps* become inactive, but not if only subsets of the 32 lanes get filtered out.

Lane Refill, by contrast, benefits from restrictive predicates more directly and to a stronger extent. As we see in Figure 4.12, Lane Refill shows the desired linear scaling. For selectivity 0.1, execution time drops by 81% compared to a selectivity of 1.0. Compared to naive execution, this is a 2.8-fold improvement. Only for high selectivities (0.9 and 1.0) the balancing work introduces a small overhead up to 2%. We conclude that Lane Refill successfully prevents the GPU from working on inactive lanes and thus improves the processing efficiency.

Multiple Warps per SM Figure 4.13 shows results for the same experiment, but we let the system overcommit and assign 4 and 8 warps to each SM. With 46 SMs on the RTX2080 GPU, this corresponds to 5,888 and 11,776 threads. As expected, overcommitting can hide some of the divergence effect that we saw in the previous experiment. Still, Lane Refill can better utilize the available resources, resulting in an performance advantage of 2.6× for the 4-warp configuration (70 ms vs. 27 ms) and 2.2× for the 8-warp configuration (40 ms vs. 18 ms). The balancing work causes a small overhead up to 3.5% for high selecitivities.

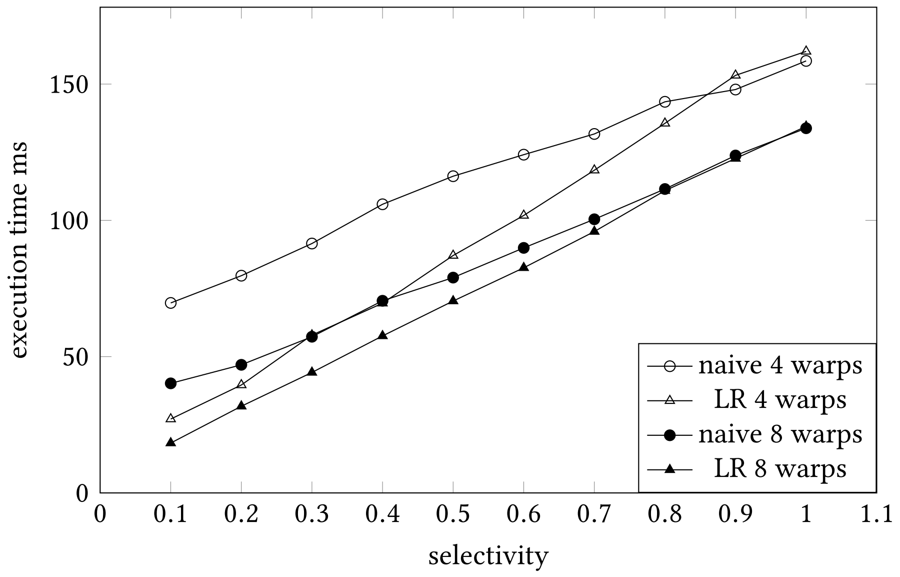


Figure 4.13: Effect of Lane Refill on filter divergence workload with multiple warps per SM. Lane Refill improves run-times for configurations with high degrees of warp-parallelism.

Varying Numbers of Operators We evaluate the effect of varying the workload size that follows filter divergence. This allows us to assess the impact of processing in divergent or in consolidated state. We use a filter divergence workload and append low to high intensity compute loads by adding up to 27 projection operators. The workload scans 1.5 B tuples from the TPC-H table `lineitem` and filters on `l_quantity > 45` with selectivity 0.1. Figure 4.14 shows the experiment results.

For increasing numbers of operators the execution times of the naive approach go from 51 ms up to 799 ms. Lane Refill reduces the execution times to at most 91 ms (27 operators). We observe factors of improvement up to 8.8×, which corresponds to the lane utilization being raised from 0.1 (after the filter) close to 1.0 (after balancing). Further investigation of workloads with selectivity 0.2 support this explanation showing no better improvements than 5×. We suspected to observe a performance penalty caused by the lane refill computation for the data point with 0 operators. We attribute the absence of this to the high scan volume, which leaves compute resources available while servicing memory loads.

Poster Case 2 In Section 4.4.1 we presented a query pipeline from TPC-H Query 10 with filter divergence. Here we evaluate the effect of applying Lane Refill in this pipeline. We measure the execution time of the pipeline for a benchmark database with scale factor 25. We use the naive approach with filter divergence originating from the selection operator and from the sparse join operator.

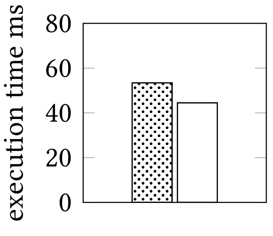


Figure 4.15: Lane Refill in Poster Case 2.

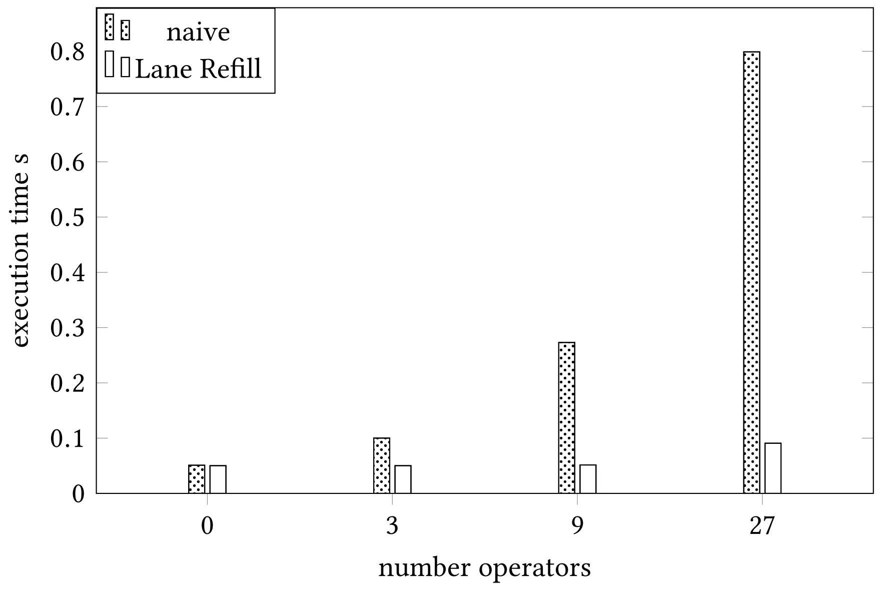


Figure 4.14: Varying numbers of operators that follow a filter. Without Lane Refill the negative divergence effects increase with the number of operators.

Then we compare the performance to a pipeline that adds a Lane Refill operator after the sparse join. Figure 4.15 shows the experiment results. The pipeline with the naive approach has an execution time of 53.4 ms. Adding the Lane Refill operator improves the execution time of the pipeline by 1.2x to 44.5 ms.

4.5.3 Push-down Parallelism vs. Lane Refill

In this experiment, we apply Push-down Parallelism and Lane Refill to the same divergence problem. This allows us to determine whether each technique is best-suited for its respective divergence domain or if one technique may work for most cases. Expansion divergence can be viewed as filter divergence that occurs in steps of the same operation. E.g., when iterating through join matches, lanes with fewer expansion items act like filtered-out lanes in the current iteration. For the experiment, we use the workload **pk-zipf-fk** from Section 4.5.1, which joins a dense primary key with a Zipf-distributed foreign key. We use the naive approach, Lane Refill, and Push-down Parallelism for the join.

Observations Figure 4.16 shows the results of the experiment. The figure shows execution times for different numbers of warps per Streaming Multiprocessor. The execution times for Lane Refill are split into regular work and pipeline flush. Pipeline flush represents work that is performed when all tuples are already scanned and only one remaining lane is active. We observe that Lane Refill can not improve over naive with regard to the best performing warp/SM configuration. For 1

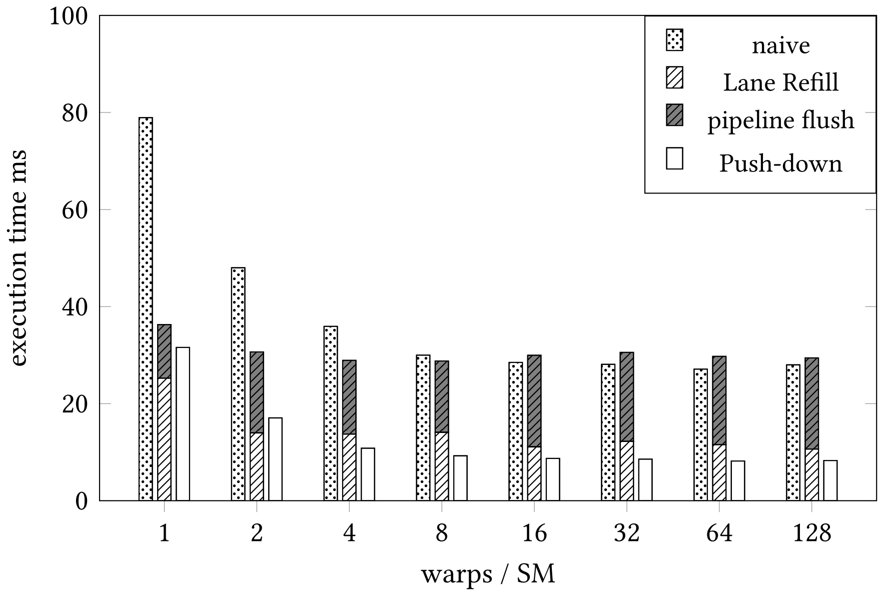


Figure 4.16: Push-down vs. Lane Refill when joining a Zipfian distribution. Push-down Parallelism is effective while Lane Refill suffers from pipeline flush.

warp/SM Lane Refill performs better than naive, but for larger warp/SM configurations, Lane Refill suffers from growing amounts of flush work. To achieve high performance, GPUs need many warps in flight. Therefore it is likely that heavy hitting tuples are isolated in warps. This prevents Lane Refill from performing effective balancing operations. Push-down Parallelism does not run into this problem because its balancing approach is effective even when one tuple per warp is remaining. Push-down Parallelism improves over naive by 3.3x for the workload.

4.5.4 Overall Performance

This section evaluates the benefit of the proposed techniques when applied in an overall system. We analyze the impact on data imports, the additional resource demand, and the query performance for realistic workloads. We compare DogQC against two other systems: OmniSci [74], which uses GPU-based query compilation and MonetDB [15], which uses operator-at-a-time processing on CPUs. The experiments were performed with OmniSci 4.8.1 and MonetDB 11.33.3.

The analysis of query performance splits into two parts to address effects of divergence optimizations and end-to-end performance separately. The workloads for both parts are the TPC-H benchmark queries on a scale factor 25 GB database.

Import Cost of Dictionary Encoding The approach used by DogQC works directly on variable length string data instead of building string dictionaries. This saves the cost of building string dictionaries during import. We quantify the cost

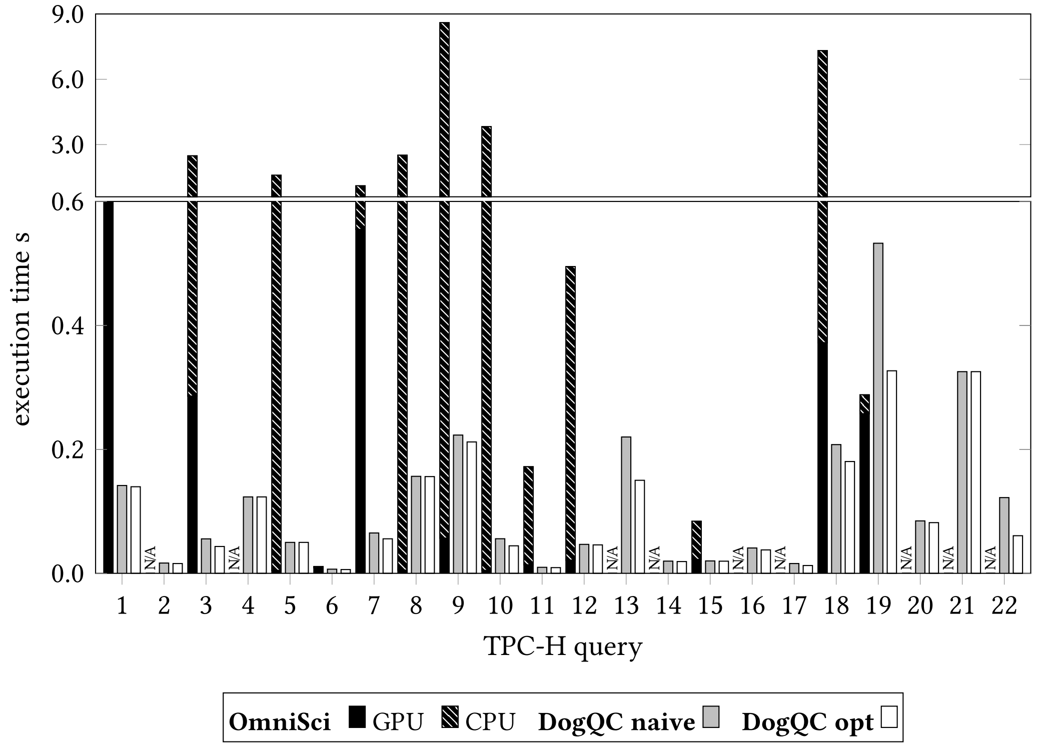


Figure 4.17: Execution times of DogQC for TPC-H benchmark queries (scale factor 25). The divergence optimizations improve query performance.

	Registers per thread	SMEM per block	Instruction footprint
Expansion naive	19	0 KB	2.36 KB
Expansion Push-down	29	0 KB	5.23 KB
Filter naive	36	0 KB	1.52 KB
Filter Lane Refill	33	1 KB	5.89 KB
Available capacity	255	64 KB	16 KB (L0)

Figure 4.18: Resource Consumption.

with an experiment that uses OmniSci's parallel importer. We import a sequence of 100 M numeric values with 9 digits. One time the data is interpreted as `INTEGER` and another time the data is interpreted as `VARCHAR(10)`. While the string-based import takes 16.16 s, the numeric import takes only 2.98 s. Importing the data with the use of a dictionary takes 5.4× longer. The shorter import times without dictionary encoding make a strong case for the processing approach of DogQC with variable length strings.

Resource Consumption We analyze the additional demand for resources of the GPU cores when using divergence balancing. This allows us to assess whether the addition of divergence balancing to query pipelines causes bottlenecks during query execution. We profile the use of shared memory (SMEM), registers per thread, and the instruction footprint for the experiments from Sections 4.5.1 and 4.5.2. Figure 4.18 shows the results along with the available capacity.

Overall, we observe a low resource consumption. The highest relative demand with divergence balancing is 5.89 KB of the 16 KB L0-level instruction cache. The L0-level, however, is backed by three larger cache-levels [44]. We conclude that the balancing techniques have a low resource demand and there are sufficient open resources for complex queries.

Divergence Optimizations We analyze the effect of applying divergence optimizations when processing realistic query workloads on the GPU. To this end, we analyze the execution times of the GPU systems DogQC and OmniSci. We use **DogQC naive** and for **DogQC opt** we add the following divergence optimizations: We replaced all join operator thats do hash builds on non-primary key attributes with Push-down Parallelism join operators. Additionally we added eight Lane Refill balancing operators to the query plans. One to each of the Queries 4, 5, 7, 10, 15, 17, 19, and 20. We show the experiment results in Figure 4.17.

OmniSci was only able to execute 13 out of 22 queries. The execution times are split into GPU work and CPU work and range from 11 ms to 8599 ms. The highest time on GPU is 604 ms and 8542 ms on CPU. For nine of the supported queries, the CPU execution takes the majority of processing time. DogQC performs all processing on the GPU and was able to execute all TPC-H queries. **DogQC naive** has execution times between 7 ms and 532 ms and **DogQC opt** has execution times between 7 ms and 327 ms. Divergence optimizations reduced execution times by more than 5% for 10 out of 22 queries. The highest factors of improvement are 2.0× (Query 19) and 1.6× (Query 16). DogQC currently adds divergence balancing into query plans after optimization (cf. Sections 4.3.4 and 4.4.4). A future divergence-aware optimizer may find plans with a higher benefit.

In comparison of OmniSci and DogQC, we observe that OmniSci frequently falls back to slower CPU processing. This causes significantly higher execution times. DogQC's processing times were faster than OmniSci's by factors up to 68× for **DogQC naive** and by factors up to 86× **DogQC opt** for the divergence-optimized version. This shows that a fallback strategy for functionality that may be considered unsuitable for GPUs is disadvantageous. DogQC shows that it is preferable to include operations into compiled pipelines even when they cause heavy divergence. The highest benefit is achieved with additional divergence balancing.

End-to-End Performance We evaluate the end-to-end performance of DogQC in comparison with MonetDB and OmniSci. OmniSci and DogQC are JIT compilation-based GPU systems. Therefore their end-to-end performance is affected by *two additional factors* that are *orthogonal* to divergence optimizations: The data transfer time between main memory and GPU, and the JIT-compilation time to generate

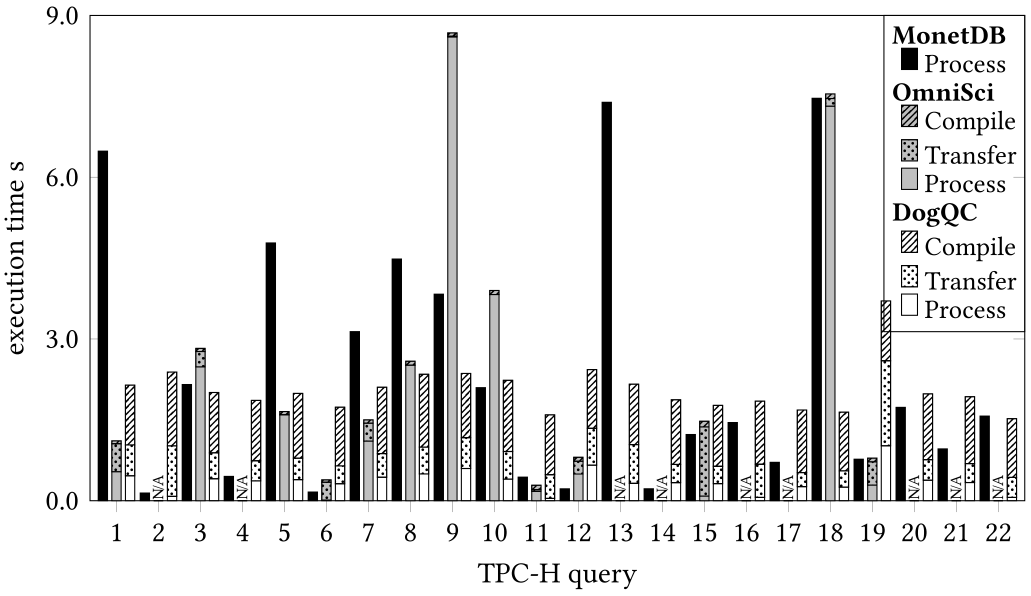


Figure 4.19: End-to-end performance for TPC-H benchmark queries with MonetDB (CPU) and OmniSci (GPU), and DogQC (GPU).

machine code. Systems that keep the entire database in GPU memory only transfer the query results. DogQC is compatible with this mode of operation, but uses sequential input data transfers here. Asynchronous techniques that pipeline data transfers and processing [104, 28] can further reduce the transfer cost.

MonetDB runs on a two-socket server with Intel Xeon E5-2695 v2 CPUs and 256 GB main memory and the GPU hardware platform remains unchanged. For DogQC, we use the version **DogQC opt**.

The experiment results are shown in Figure 4.19. MonetDB's end-to-end execution times range from 142 ms up to 7464 ms. DogQC's end-to-end execution times range from 1188 ms to 3705 ms and were shorter than MonetDB's for 12 out of 22 queries. DogQC is faster only for longer running queries, where the lowered processing times outweigh the cost of data transfers and compilation. OmniSci was able to execute only 13 out of 22 queries. The end-to-end execution times range from 56 ms up to 8662 ms. The previous experiment showed that DogQC has lower processing times than OmniSci for all but one TPC-H query. End-to-end execution, however, is shorter for 8 of the 13 queries with OmniSci. The measurement shows that this effect is due to JIT compilation times. The current version of DogQC is not optimized for low compilation times and generates high-level Cuda code. Extending DogQC with a low-level code generator, such as LLVM, would reduce JIT-compilation times. The highest factors of improvement, that we observe in this experiment are 4.54× over MonetDB and 4.51× over OmniSci.

4.5.5 Usage Scenario: String Pattern Matching

In this chapter, we proposed several usage scenarios for the presented divergence balancing techniques. To study their applicability, we exemplarily evaluate one of them. The workload is the test for a prefix of 50 characters in a dataset with titles of computer science articles. The dataset stems from DBLP[4] and was scaled up by 5× by repitition. Matching prefixes were scattered into random positions. We use two datasets with match rates of 1% and 32%, each has 21.5 M entries and an average title length of 76 characters. We process the workload with one warp per SM and each warp lane is responsible for one title at a time. We apply Lane Refill to reactivate lanes with rejected titles. The results of the experiment are shown in Figure 4.20. The workload with 1% matches has lower run-times than the workload with 32% matches. For lower match rates many strings are rejected early reducing the overall processing volume. We observe that divergence optimizations improve the performance of string prefix tests. For the 1% workload Lane Refill improves performance by 2.5× from 38 ms down to 15 ms. For the 32% workload the improvement is 1.7× from 102 ms down to 59 ms.

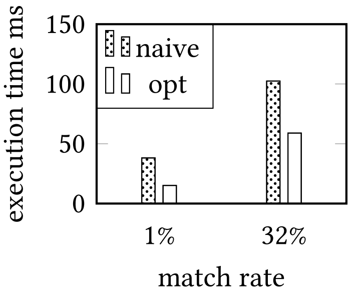


Figure 4.20: Balancing parallel prefix test computations.

4.6 More Related Work

In this section, we relate our approach to work that was not mentioned in one of the other sections. First we discuss work in the database context that uses the GPU feature *dynamic parallelism* to balance the use of parallel resources. Second we discuss other related GPU query processing techniques.

Dynamic Parallelism Dynamic parallelism is a feature that allows GPUs to start new kernels from within a kernel [71]. The number of threads for the inner kernels can be chosen dynamically. Rui et al. [85] apply dynamic parallelism for sort-merge joins. Wang et al. [97] evaluate the feature for joins based on binary search and for regular expression matching. Liu et al. [59] propose the implementation of a MapReduce framework for GPUs with dynamic parallelism. Similar to Push-down Parallelism, dynamic parallelism adapts parallel resources to the characteristics of sub-problems. The main advantage of the approach is programmability. The downside, however, are costs for context switching. Chen et al. report overheads of up to 21× [22].

[4] `https://dblp.org`

Pipelined GPU Query Processing This work targets GPU query engines that implement pipelining via just-in-time compilation. In related work other means of pipelining have been proposed, such as in-cache processing [77] and kernel fusion [99]. Other related work that performs pipelining via just-in-time compilation [19, 100] may be susceptible to the presented divergence optimizations.

4.7 Engineering Query Compilers

As in the previous chapters, we discuss the lessons learned for the design of query compilers. Here we discuss the application of data-parallelism. In the design of query engines, data-parallelism (via instructions) and query compilation are somewhat contradicting techniques. When using data-parallel instructions to handle multiple input elements in a compiled pipeline, the quality of parallelism deteriorates over the course of the pipeline (cf. Sections 4.3 and 4.4). There is no simple mapping of processing lanes to the processed elements as the structure of the processed elements changes over the course of the compiled pipeline (e.g. a tuple finds multiple join matches). This issue does not affect data-parallelism via multi-threading [56], because it uses scalar instructions in independent partitions.

At a first glance, data-parallel instructions can be applied much more easily in column-based processing as in operator-at-a-time and vector-at-a-time engines. For instance when evaluating a filter predicate to set a bit-mask, we can use data-parallel instructions where each procesing lane is responsible for the predicate evaluation on one element. At a closer look, however, a similar problem as we have seen for compiled pipelines already affects column-based processing techniques. Operations such as join probes or selections can already produce varying result numbers in a single operation. To account for this, data-parallel implementations typically use reorganization operations, e.g. `compress`, `permute`, `scatter`, `gather` [82], that restore the alignment of data elements to processing lanes. Using these balancing operations, however, comes at a cost and careful design choices are necessary not to pay an additional cost that exceeds the benefits of using data-parallel instructions in the first place.

We can leverage some insights from this work to address the issue. We have seen that it is worthwhile to tolerate some degree of imbalance. Our evaluation showed benefits for data-parallel processing already with few balancing operations per compiled pipeline and also *without* balancing operations. Query compilation and data-parallelism can work together most beneficially with a convervative use of balancing work. Approaches that aim to avoid imbalances by replacing data with a uniform surrogate (e.g. dictionary encoding) are often affected by limitations in functionality. Especially on GPUs, it is beneficial perform use the non-uniform data for processing and to allow a degree of imbalance.

4.8 Summary

In this chapter, we put the processing capabilities of data-parallel coprocessors for non-uniform database workloads to the test. DogQC introduces techniques, that allow us to gracefully align parallel processing units with work items, even when problems are heavily skewed. The evaluation analyzes different query processing scenarios with distinct workload imbalances. We observed that the techniques Lane Refill and Push-down Parallelism are able to increase processing efficiency for these non-uniform workloads.

Existing query coprocessors typically avoid imbalances by working on a uniform surrogate (e.g. dictionary keys, materialization barriers). This has led to the perception that GPUs have limited capabilities of processing irregular problems. DogQC conversely avoids the overhead of maintaining such additional data-structures and instead restores balance during non-uniform processing. This approach achieves a bigger functionality range and better performance than other query coprocessing engines. This is shown by support of the full set of TPC-H benchmark queries with best-in-class performance.

5 Conclusions

This thesis dealt with query processing techniques on modern hardware platforms. To account for limitations in today's systems, we looked at new ways to apply query compilation techniques that increase the processing efficiency. Query compilation compiles queries to machine code via JIT compilation, which removes interpretation overheads and reduces the amount of data movement. The latter is essential to account for the large data volumes that are moved through communication channels with limited bandwidth during processing. In the course of this thesis, we addressed several open challenges for query compilation in the chapters: Compilation, Communication, and Processing.

Chapter 2 ("*Compilation*") dealt with the translation of queries to low-level IRs and machine code. With query compilation, the execution of queries, once translated, is usually fast. However, the JIT compilation process itself extends the query response times. In existing approaches, the compiler infrastructure used was designed for general purpose programming languages. In this way the translation process does not account for the specific properties of database workloads. We presented a new technique that leverages database domain knowledge in a tailored machine code generation process. This enabled a simplification of the compilation processes and led to significant reduction of compilation times. While the compilation approach is simple, it does not compromise the benefits in processing speed gained from query compilation.

Chapter 3 (*"Communication"*) dealt with query processing techniques on parallel GPU-style coprocessors. These coprocessors represent a solution for the design limitation of standard processors. However, the communication infrastructure that is used by coprocessor systems is more complex than in standard systems. We performed a bandwidth analysis that clearly exposes the limiting factors. We showed that in naive approaches, the bus-link between the hosting system and the coprocessor quickly becomes a bottleneck. However, when leveraging existing techniques to use the bus-links more efficiently, the on-coprocessor GPU global memory becomes the new limiting factor. To address these limitations, we showed a way to combine (the previously unrelated) GPU query processing techniques with query compilation techniques. This reduces the amount of GPU global memory access by a significant factor and leads to lower overall execution times.

By transfering the query compilation approach to GPU-style coprocessors, we have introduced a drastic change to the processing model. Chapter 4 (*"Processing"*) deals with the effect that this has on the efficiency of data-parallel processing. Data-parallel instructions perform the same operation across several elements. Therefore variations in the workloads lead to processing inefficencies. We introduced balancing techniques to address this problem, which has a particularly strong effect on query compilation techniques that closely combine several consecutive operations in data-parallel execution. We showed that the techniques are well-suited to restore balanced parallelism during compiled execution and to recover from distinct workload imbalances.

In the following, we add concluding remarks to several overarching topics from the chapters. This also allows us to relate several individual results. We first discuss observations for the execution model, then we continue with observations for different hardware platforms. Finally we provide a higher-level perspective on the research results.

5.1 Compilation vs. Interpretation

Compiled execution provides major improvements over classical interpretation techniques such as Volcano. However, interpretation techniques that are optimized for in-memory processing, e.g. vector-at-a-time, achieve similar benefits to query compilation (i.e. high processing speeds and bandwidth efficiency). In several parts of this thesis, we compared compilation-based techniques with interpretation-based techniques. We discuss several results here for a concluding comparison.

In Chapter 2, we compared our compilation approach in ReSQL with the vector-at-a-time system DuckDB. Both approaches perform processing on standard CPUs. For the supported TPC-H queries, ReSQL achieved higher processing speeds by an average factor of 1.9×. In Chapter 3 we showed additional results on GPUs. Our compilation approach from HorseQC achieved higher processing speeds than the operator-at-a-time engine CoGaDB by an average factor of 2.6×.

Across our experiments, the best performing compilation-based techniques have performed significantly better than the best interpretation-based techniques. In other work [90, 48] more even performance has been observed in the comparison of query compilation and vector-at-a-time. Our results, however, are in line with the results from Leis et al. [56] that compare complete systems rather than micro-benchmarks. The authors show an advantage of 2.7× for HyPer (compilation) over Vectorwise (vector-at-a-time).

5.2 Hardware Platforms for Query Processing

Standard CPUs are the most prevalent hardware platform used for query processing. Alternative platforms, such as GPUs, FPGAs, clusters of smaller systems, or custom hardware have been proposed. While some projects show practical use [74], it is unclear whether such approaches will play a major role in future database systems. We revisit results and observations from this thesis to contribute to this discussion.

The results in Chapters 3 and 4 show very high processing speeds for compiled execution on GPUs. The processing speeds are particularly high when all data is held in coprocessor memory. For instance the results for DogQC (Figure 4.17) show an execution time of 44 ms for TPC-H Query 10 (25 GB database, single GPU). By comparison Leis et al. show an execution time of 610 ms for Hyper (100 GB database, server with 4 CPUs). For the GPU approach data is 4× smaller but the execution is nearly 14× faster despite only using a single GPU. While a fair comparison is difficult with varying hardware and database configurations, this still suggests a large potential for GPU-based query processing.

In practice the assumption of holding entire databases in GPU memory is usually not feasible. GPU memory is smaller than main-memory, it is more expensive, and not upgradable or replaceable. When including data transfers with efficient transfer techniques (Chapter 3), the comparison between CPU and GPU boils down to two main factors: The bandwidth of the bus link for GPU processing (e.g. 16 GB/s PCIe per GPU) and the bandwidth of main-memory for CPU processing (e.g. 50 GB/s per CPU socket). In fact these bandwidths can be used to approximate the processing speeds of compiled execution, because relations are transfered through these channels just once (and no intermediate data).

From this perspective GPUs are at the disadvantage as the bandwidth of the bus link is typically lower than the main memory bandwidth. Workloads that are processing-intensive (rather then data intensive) can benefit more easily from GPUs. These, however, are not so typical for query processing, but may appear during query processing on domain-specific data (e.g. geographical data). Future architectures that combine the GPU's data-parallel processing approach with classical DRAM would significantly increase the benefit of GPU query processing techniques.

5.3 Impact of the Covered Research

During the work on this thesis, we emphasized the practical usability of our approaches. Several groups from research and industry have showed interest in applications of the techniques. To this end we published the source code of our prototype systems DogQC[1] and ReSQL[2] publicly. Paul et al. [76] have already used DogQC for the evaluation of their query processing approach. Despite the practical

[1]`https://github.com/Henning1/dogqc/`

[2]`https://github.com/Henning1/resql/`

orientation of the projects, the conceptual contributions have also lead to several citations in other research work. At the time of writing, the author's work has been cited in 70 articles according to the Association of Computing Machinery (ACM) and in 186 articles according to Google Scholar.

Acknowledgements

The work on this thesis has been very demanding. At the same time it was—personally and professionally—an enlightening experience. Finishing it would not have been possible without many great people. I want to express my sincere gratitude to the people who encouraged and supported me. Everyone who gave their advice, or who participated in collaborations, discussions and feedback eventually helped me more than they know. I want to thank several people in the following. This, however, is not conclusive and I would like to say thanks to everyone for inspiring talks, conversations, blog posts, and more.

First and foremost I want to express my sincere gratitude to *Jens Teubner* who inspired me to participate in database research and gave me the opportunity to join the DBIS group at TU Dortmund. I very much appreciate the opportunity and the well-thought-out advice that I frequently got.

When I first joined the DBIS group as a student, I worked together with *Sebastian Breß*. In this time I gained first practical experience with the development of database systems and later the collaboration grew into several research projects and publications. In the following years there were several other collaborations with *Jan Mühlig*, *Stefan Noll*, *Steffen Zeuch*, *Volker Markl*, *Bastian Köcher* and *Tilmann Rabl*. I really appreciate these collaborations and they were as much fun as they were instructional.

During my PhD programme, I worked together with several colleagues. The everyday exchange and also the in-depth discussions made this a great experience. Thanks to my colleagues from the DBIS group *Roland Kühn*, *Max Berens*, *Florian Grieskamp*, *Marcel Preuß*, *Michael Kußmann*, *Thomas Lindemann*, *Lea Schönberger*, *Alexander Lochmann* and also *Stefan Dißmann* as director of the computer science department. I also spent half a year working at Oracle Labs in California during an internship. It was an exciting and instructive time working with the guys from *Nipun*'s team including *Sanjay Jinturkar*, *Renato Marroquin*, *Sandeep Agrawal*, *Michael deLorimier*, *Mark Esguerra*, *Sam Idicula*, *Shrikumar Hariharasubrahmanian* and *Venkatanathan Varadarajan*.

Next I would like to thank my mentor *Günter Rudolph* and the PhD thesis committee for taking the time and effort. Many thanks to *Thomas Neumann* for reviewing the thesis, and to *Erich Schubert* and *Johannes Fischer* for taking part in the defense.

Last but not least I want to thank my family and friends, my brothers and my parents. Thanks to my personal friends *Florian*, *Martin*, and *Romano* for proof-reading and also for distracting me when it counts. My special appreciation and thanks go to my loved ones *Jenni* and my son *Erik*.

Bibliography

[1] S. R. Agrawal, S. Idicula, A. Raghavan, E. Vlachos, V. Govindaraju, V. Varadarajan, C. Balkesen, G. Giannikis, C. Roth, N. Agarwal, et al. A many-core architecture for in-memory data processing. In *MICRO*, pages 245–258. ACM, 2017.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers, principles, techniques. *Addison Wesley*, 7(8):9, 1986.

[3] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *PVLDB*, 5(10):1064–1075, 2012.

[4] P. Andrade, P. Bonzini, I. Piumarta, M. Flatt, L. Michel, and L. Courtes. GNU lightning, 2022. URL: `https://www.gnu.org/software/lightning/`.

[5] O. Arnold, S. Haas, G. P. Fettweis, B. Schlegel, T. Kissinger, T. Karnagel, and W. Lehner. HASHI: An Application Specific Instruction Set Extension for Hashing. In *ADMS workshop*, pages 25–33, 2014.

[6] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. *SIGPLAN Notices*, 31(5):149–159, 1996.

[7] P. Bakkum and S. Chakradhar. Efficient data management for GPU databases. *High Performance Computing on Graphics Processing Units*, 2012.

[8] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, pages 362–373. IEEE, 2013.

[9] J. Bharadwaj, W. Y. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, and J. Pierce. The Intel IA-64 compiler code generator. *MICRO*, 20(5):44–53, 2000.

[10] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *Conference on high performance graphics*, pages 159–166. ACM, 2009.

[11] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, pages 283–296. ACM, 2009.

[12] G. E. Blelloch. Prefix Sums and Their Applications. Technical report, Carnegie Mellon University, 1990.

[13] P. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 61–76. Springer, 2013.

[14] P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. *The VLDB Journal*, 8(2):101–119, 1999.

[15] P. A. Boncz, S. Manegold, M. L. Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *PVLDB*, volume 99, pages 54–65, 1999.

[16] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, volume 5, pages 225–237, 2005.

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

[18] S. Breß, H. Funke, and J. Teubner. Robust query processing in co-processor-accelerated databases. In *SIGMOD*, pages 1891–1906. ACM, 2016.

[19] S. Breß, B. Köcher, H. Funke, S. Zeuch, T. Rabl, and V. Markl. Generating custom code for efficient query execution on heterogeneous processors. *The VLDB Journal*, 27(6):797–822, 2018.

[20] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer Networks*, 33(1-6):309–320, 2000.

[21] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[22] G. Chen and X. Shen. Free launch: optimizing gpu dynamic kernel launches through thread reuse. In *MICRO*, pages 407–419, 2015.

[23] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *TODS*, 32(3):17–es, 2007.

[24] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel GPU methods for single-source shortest paths. In *International Parallel and Distributed Processing Symposium*, pages 349–359. IEEE, 2014.

[25] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*, pages 1243–1254. ACM, 2013.

[26] I. J. Egielski, J. Huang, and E. Z. Zhang. Massive Atomics for Massive Parallelism on GPUs. *SIGPLAN Notices*, 49(11):93–103, 2015.

[27] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD*, 40(4):45–51, 2012.

[28] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. Pipelined Query Processing in Coprocessor Environments. In *SIGMOD*, pages 1603–1618. ACM, 2018.

[29] H. Funke, J. Mühlig, and J. Teubner. Efficient generation of machine code for query compilers. In *DaMoN workshop*, pages 1–7, 2020.

[30] H. Funke, J. Mühlig, and J. Teubner. Low latency query compilation. *The VLDB Journal*, 2022.

[31] H. Funke and J. Teubner. Data-parallel query processing on non-uniform data. *PVLDB*, 13(6):884–897, 2020.

[32] H. Funke and J. Teubner. Like water and oil: with a proper emulsifier, query compilation and data parallelism will mix well. *PVLDB*, 13(12):2849–2852, 2020.

[33] H. Funke and J. Teubner. Low-Latency Compilation of SQL Queries to Machine Code. *PVLDB*, 14(12):2691–2694, 2021.

[34] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *Transactions on knowledge and data engineering*, 4(6):509–516, 1992.

[35] S. Geffner, D. Agrawal, A. El Abbadi, and T. Smith. Relative Prefix Sums: An Efficient Approach for Querying Dynamic OLAP Data Cubes. In *ICDE*, pages 328–335. IEEE, 1999.

[36] G. Graefe. Volcano — an extensible and parallel query evaluation system. *Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[37] C. Gregg and K. Hazelwood. Where Is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *ISPASS*, pages 134–144. IEEE, 2011.

[38] T. Gubner and P. Boncz. Charting the design space of query execution using voila. *PVLDB*, 14(6):1067–1079, 2021.

[39] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Query Coprocessing on Graphics Processors. *Transactions on Database Systems*, 34(4):21, 2009.

[40] 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.

[41] S. Helmer, T. Westmann, and G. Moerkotte. Diag-Join: An Opportunistic Join Algorithm for 1:N Relationships. *PVLDB*, 1998.

[42] W. D. Hillis and G. L. Steele Jr. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[43] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *Transactions on Parallel and Distributed Systems*, 22(1):105–118, 2010.

[44] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza. Dissecting the NVidia Turing T4 GPU via Microbenchmarking. *arXiv preprint 1903.07486*, 2019.

[45] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU join processing revisited. In *DaMoN workshop*, pages 55–62. ACM, 2012.

[46] T. Karnagel, D. Habich, and W. Lehner. Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. *PVLDB*, 10(7):733–744, 2017.

[47] T. Karnagel, R. Mueller, and G. M. Lohman. Optimizing GPU-Accelerated Group-By and Aggregation. In *ADMS workshop*, pages 13–24, 2015.

[48] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB*, 11(13):2209–2222, 2018.

[49] T. Kersten, V. Leis, and T. Neumann. Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra. *The VLDB Journal*, 30, 2021.

[50] A. Kipf, H. Lang, V. Pandey, R. A. Persa, P. Boncz, T. Neumann, and A. Kemper. Approximate geospatial joins with precision guarantees. In *ICDE*, pages 1360–1363. IEEE, 2018.

[51] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[52] P. Kobalicek. Asmjit Library, 2022. URL: `https://asmjit.com`.

[53] A. Kohn, V. Leis, and T. Neumann. Adaptive execution of compiled queries. In *ICDE*, pages 197–208. IEEE, 2018.

[54] H. Lang, A. Kipf, L. Passing, P. Boncz, T. Neumann, and A. Kemper. Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. In *DaMoN workshop*, page 5. ACM, 2018.

[55] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86. IEEE, 2004.

[56] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754. ACM, 2014.

[57] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *PVLDB*, 9(14):1647–1658, 2016.

[58] H. Liu and H. H. Huang. Enterprise: breadth-first graph traversal on GPUs. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[59] L. Liu, Y. Zhang, M. Liu, C. Wang, and J. Wang. A-MapCG: an adaptive MapReduce framework for GPUs. In *International Conference on Networking, Architecture, and Storage*, pages 1–8. IEEE, 2017.

[60] S. Manegold, P. Boncz, N. Nes, and M. Kersten. Cache-conscious radix-decluster projections. In *PVLDB*, pages 684–695, 2004.

[61] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. System V application binary interface. *AMD64 Architecture Processor Supplement, Draft v0*, 99, 2013.

[62] X. Mei and X. Chu. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2016.

[63] P. Menon, A. Pavlo, and T. C. Mowry. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *PVLDB*, 11(1):1–13, 2017.

[64] D. Merrill. CUB v1.7.0: CUDA Unbound, a Library of Warp-Wide, Block-Wide, and Device-Wide GPU Parallel Primitives, 2017.

[65] D. Merrill and M. Garland. Single-Pass Parallel Prefix Scan with Decoupled Look-Back. Technical report, NVidia Corporation, 2016.

[66] I. Müller, R. Marroquín, D. Koutsoukos, M. Wawrzoniak, S. Akhadov, and G. Alonso. The collection Virtual Machine: an abstraction for multi-frontend multi-backend data analysis. In *DaMoN workshop*, pages 1–10, 2020.

[67] I. Müller, C. Ratsch, F. Faerber, et al. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *EDBT*, pages 283–294, 2014.

[68] I. Müller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber. Cache-efficient aggregation: Hashing is sorting. In *SIGMOD*, pages 1123–1136. ACM, 2015.

[69] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[70] S. Noll, H. Funke, and J. Teubner. Energy efficiency in main-memory databases. *Datenbank-Spektrum*, 17(3):223–232, 2017.

[71] NVidia Corporation. NVidia's Next Generation CUDA Compute Architecture: Kepler™ GK110/210. *NVidia White Paper*, 2014.

[72] NVidia Corporation. NVidia Turing GPU Architecture. *NVidia White Paper*, 2018.

[73] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel. Applying the roofline model. In *International Symposium on Performance Analysis of Systems and Software*, pages 76–85. IEEE, 2014.

[74] OmniSci Incorporated. OmniSciDB, 2022. URL: `https://www.omnisci.com`.

[75] P. E. O'Neil, E. J. O'Neil, and X. Chen. The star schema benchmark (SSB). *Pat*, 200(0):50, 2007.

[76] J. Paul, B. He, S. Lu, and C. T. Lau. Improving execution efficiency of just-in-time compilation based query processing on gpus. *PVLDB*, 14(2):202–214, 2020.

[77] J. Paul, J. He, and B. He. GPL: A GPU-based pipelined query processing engine. In *SIGMOD*, pages 1935–1950. ACM, 2016.

[78] H. Pirk, S. Manegold, M. L. Kersten, et al. Accelerating Foreign-Key Joins using Asymmetric Memory Channels. In *ADMS workshop*, pages 27–35, 2011.

[79] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo-a vector algebra for portable database performance on modern hardware. *PVLDB*, 9(14):1707–1718, 2016.

[80] M. Poletto and V. Sarkar. Linear scan register allocation. *Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[81] O. Polychroniou and K. A. Ross. Vectorized Bloom filters for advanced SIMD processors. In *DaMoN workshop*, page 6. ACM, 2014.

[82] O. Polychroniou and K. A. Ross. VIP: A SIMD vectorized analytical query engine. *The VLDB Journal*, 29(6):1243–1261, 2020.

[83] J. Power, Y. Li, M. D. Hill, J. M. Patel, and D. A. Wood. Toward GPUs Being Mainstream in Analytic Processing: An Initial Argument Using Simple Scan-Aggregate Queries. In *DaMoN workshop*, page 11. ACM, 2015.

[84] M. Raasveldt and H. Mühleisen. DuckDB: an embeddable analytical database. In *SIGMOD*, pages 1981–1984. ACM, 2019.

[85] R. Rui and Y.-C. Tu. Fast Equi-Join Algorithms on GPUs: Design and Implementation. In *International Conference on Scientific and Statistical Database Management*, page 17. ACM, 2017.

[86] S. Sengupta, M. Harris, and M. Garland. Efficient Parallel Scan Algorithms for GPUs. *NVidia, Tech. Rep. NVR-2008-003*, (1):1–17, 2008.

[87] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU Computing. In *Graphics Hardware*, volume 2007, pages 97–106, 2007.

[88] M. Sha, Y. Li, and K.-L. Tan. GPU-based Graph Traversal on Compressed Graphs. In *SIGMOD*, pages 775–792. ACM, 2019.

[89] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch. How to architect a query compiler. In *SIGMOD*, pages 1907–1922. ACM, 2016.

[90] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *DaMoN workshop*, pages 33–40, 2011.

[91] SQLite Developers. SQLite3, 2022. URL: `https://www.sqlite.org`.

[92] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The architecture of SciDB. In *International Conference on Scientific and Statistical Database Management*, pages 1–16. Springer, 2011.

[93] The PostgreSQL Global Development Group. PostgreSQL, 2022. URL: `https://www.postgresql.org`.

[94] P. K. Tiwari, V. V. Menon, J. Murugan, J. Chandrasekaran, G. S. Akisetty, P. Ramachandran, S. K. Venkata, C. A. Bird, and K. Cone. Accelerating x265 with intel® advanced vector extensions 512. *Intel White Paper*, 2018.

[95] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 265–283. Springer, 2009.

[96] M. Wahib and N. Maruyama. Scalable Kernel Fusion for Memory-Bound GPU Applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 191–202. IEEE, 2014.

[97] J. Wang and S. Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured GPU applications. In *International Symposium on Workload Characterization*, pages 51–60. IEEE, 2014.

[98] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[99] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *MICRO*, pages 107–118. IEEE, 2012.

[100] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. Red fox: An execution environment for relational query processing on gpus. In *International Symposium on Code Generation and Optimization*, page 44. ACM, 2014.

[101] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. Optimizing Data Warehousing Applications for GPUS Using Kernel Fusion/-Fission. In *Parallel and Distributed Processing Symposium*, pages 2433–2442. IEEE, 2012.

[102] S. Yan, G. Long, and Y. Zhang. StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization. In *SIGPLAN Notices*, volume 48, pages 229–238. ACM, 2013.

[103] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *DaMoN workshop*, pages 1–9, 2011.

[104] 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.

[105] K. Zhang, F. Chen, X. Ding, Y. Huai, R. Lee, T. Luo, K. Wang, Y. Yuan, and X. Zhang. Hetero-DB: Next Generation High-Performance Database Systems by Best Utilizing Heterogeneous Computing and Storage Resources. *Journal of Computer Science and Technology*, 30(4):657–678, 2015.

[106] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong. GPU-based NFA implementation for memory efficient high speed regular expression matching. In *SIGPLAN Notices*, volume 47, pages 129–140. ACM, 2012.