Otto-von-Guericke University Magdeburg

Faculty of Computer Science

Master Thesis

A Framework for Adaptive Reprogramming Using a JIT-Compiled Domain Specific Language for Query Execution

Author:

Paul Blockhaus

March 11, 2022

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
Research Group Databases and Software Engineering
Otto-von-Guericke University Magdeburg

Dr.-Ing. David Broneske
Department of Infrastructures and Methods
German Centre for Higher Education Research and Science Studies

Blockhaus, Paul:
A Framework for Adaptive Reprogramming Using a JIT-Compiled Domain Specific Language for Query Execution
Master Thesis, Otto-von-Guericke University Magdeburg, 2022.

Abstract

With the evolving heterogeneous many-core age, hardware becomes more and more specialized to increase performance, despite the problems of dark silicon. This poses a new category of challenges for modern database design by having to make the best use of available resources. In the last decade, new algorithms have been designed and optimized to utilize not only CPUs but also GPUs for database query processing. However, it has also been shown that straight-forward-algorithms alone will not achieve the goal, as they are sensitive to the query characteristics and dependent on the optimization of the underlying hardware. Current state-of-the-art database systems tend to solve this problem with expensive hand-optimization of the algorithms, but with the increasing heterogeneity this becomes a Sisyphean task.

To this end, we propose a novel framework for adaptive reprogramming, an adaptivity technique which automatically optimizes code to the query and hardware characteristics. The framework is based upon the MLIR framework, a framework for domain-specific languages built upon LLVM. To allow for a large degree of variability without trading off expressiveness, we choose to adapt the query execution language Voila as an MLIR dialect. MLIR enables us to use a common infrastructure to simplify the implementation of our high-performance code and allows for simple adaptation for execution on heterogeneous hardware. Building upon this dialect, we implement a compilation pipeline with fully 7 configurable optimizations and many more optimizations built-in per default into MLIR. From this set of optimizations, we are able to automatically generate at least 200 different variants with the goal to optimize it for specific query and hardware characteristics.

To validate our approach and show its adaptivity, we conduct an extensive evaluation on three different machines using a subset of 120 variants built from 5 of our optimizations. To show the performance in the context of real-life workloads, we use a subset of the TPC-H benchmark. In the first part, we analyze the impact of our optimizations on the variant run times, as well as the impact of optimizations on each other. Furthermore, we show that even on our three architecturally similar evaluation machines, the optimizations showed different effects. Following the variability analysis, we show that our approach is able to outperform statically compiled code of general-purpose compilers in the second part of our analysis. To conclude our evaluation, we compare the performance of our generated variants to state-of-the-art, hand-optimized representatives for the vectorized and compiled query execution engines Typer and Tectorwise.

Contents

List of Figures ix

List of Tables xiii

List of Code Listings xv

1 Introduction 1

2 Background 7
2.1 Principles and Architectures for Parallel Computing . . . 7
2.1.1 Flynn's Taxonomy . . . 7
2.1.2 CPU Architecture . . . 10
2.2 Query Execution . . . 13
2.3 Voila . . . 15
2.4 MLIR . . . 15
2.4.1 Language . . . 16
2.4.2 Dialects . . . 17
2.4.3 Pass Infrastructure . . . 20
2.4.3.1 The Pass Manager . . . 21
2.4.3.2 Bufferization . . . 21

3 Related Work 23

4 Voila as MLIR Dialect 27
4.1 Architecture Overview . . . 27
4.2 Voila as MLIR Dialect . . . 29
4.2.1 Parsing Voila . . . 29
4.2.1.1 Functions . . . 29
4.2.1.2 LOOPs . . . 30
4.2.1.3 Statements . . . 30
4.2.1.4 Expressions . . . 31
4.2.2 Type System . . . 33
4.2.3 Representing Voila's Vectors in MLIR . . . 33
4.2.4 Type and Shape Inference . . . 34
4.2.5 SSA Semantics for Voila . . . 36
4.3 Lowering MLIR to LLVM . . . 36
4.3.1 Linalg Lowering . . . 37
4.3.2 Affine Lowering . . . 37

4.3.3 Runtime Wrappers . . . 39
4.4 Executing the Query . . . 39

5 Rewrite Rules **41**
5.1 Function Inlining . . . 41
5.2 Operation Predication for Lazy Materialization . . . 43
5.3 Operator Fusion . . . 45
5.4 Loop Tiling . . . 48
5.4.1 Linalg Tiling . . . 49
5.4.2 Affine Tiling . . . 50
5.5 Linalg Lowering to Affine Dialect . . . 51
5.6 Affine Vectorization . . . 52
5.7 Parallelization . . . 54
5.7.1 Affine Parallelization . . . 54
5.7.2 Parallel Loops To Async Lowering . . . 56

6 Evaluation **59**
6.1 Setup . . . 59
6.2 Dataset & Queries . . . 60
6.3 Variability Benchmark Results . . . 62
6.3.1 Overview . . . 62
6.3.2 Parallelization . . . 64
6.3.3 Vectorization . . . 65
6.3.4 Tiling . . . 66
6.3.5 Unrolling . . . 68
6.3.6 Selection Forwarding . . . 69
6.3.7 Statistic Correlations of Optimizations with Run Time . . . 71
6.3.8 Summary . . . 72
6.4 JIT vs. Baseline . . . 73
6.4.1 Time . . . 74
6.4.2 Impact of Compression and Normalization . . . 75
6.5 Voila vs. Typer and Tectorwise . . . 76
6.6 JIT Compile Time . . . 78
6.7 Threads to Validity . . . 80
6.8 Summary . . . 81

7 Conclusion **83**
7.1 Summary . . . 83
7.2 Future Work . . . 84

Appendix **87**
A.1 Modified TPC-H Queries . . . 87
A.1.1 Influence of Tiling on Other Optimizations . . . 88
A.1.1.1 Vectorization . . . 88
A.1.1.2 Parallelization . . . 88
A.1.1.3 Selection Forwarding . . . 89
A.1.1.4 Unrolling . . . 89
A.1.2 Influence of Vectorization on Other Optimizations . . . 89

A.1.2.1 Tiling . . . 89
A.1.2.2 Parallelization . . . 90
A.1.2.3 Selection Forwarding . . . 90
A.1.2.4 Unrolling . . . 90
A.1.3 Influence of Unrolling on Other Optimizations . . . 91
A.1.3.1 Tiling . . . 91
A.1.3.2 Parallelization . . . 91
A.1.3.3 Selection Forwarding . . . 91
A.1.3.4 Vectorization . . . 92
A.1.4 Influence of Parallelization on Other Optimizations . . . 92
A.1.4.1 Tiling . . . 92
A.1.4.2 Unrolling . . . 92
A.1.4.3 Selection Forwarding . . . 93
A.1.4.4 Unrolling . . . 93
A.1.5 Influence of Selection Forwarding on Other Optimizations . . . 93
A.1.5.1 Tiling . . . 93
A.1.5.2 Unrolling . . . 94
A.1.5.3 Parallelization . . . 94
A.1.5.4 Vectorization . . . 94
A.1.6 Compile Times for Machines . . . 95

Bibliography **97**

List of Figures

1.1 Trend of Memory Bandwidth 1

1.2 Trend of Microprocessor Performance 2

1.3 Influence of Query Characteristics on Algorithm Runtime for Different Selection Mechanisms 3

2.1 Example Illustration for SISD Processing on the Von-Neumann Architecture 8

2.2 Example of MISD Data and Instruction Flow on the Von-Neumann Architecture 9

2.3 Example Illustration for SIMD Processing on the Von-Neumann Architecture 9

2.4 Conceptual Illustration of MIMD Data and Control Flow on the Von-Neumann Architecture 10

2.5 Simplified Blockdiagram of CPU Cores Connected With Bus and Caches 11

2.6 Example of Strided Memory Access 12

2.7 Example Pipeline with a Depth of 8 and three Cycle Pipeline Stall . . 13

2.8 Overview of In-Tree Dialects in MLIR 18

4.1 Architecture of our Adaptive Reprogramming Framework 28

4.2 Type Hierarchy of Voila 34

5.1 Schematic Array Processing with and Without Tiling 48

6.1 Heatmaps Showing the Run Times of the Query Variant Space 63

6.2 Stripplots of Speed-up for Parallelized Variants vs. non-parallelized . 64

6.3 Stripplot Showing the Influence of Vectorization on Parallelization for Machine 2 65

6.4 Stripplot Showing the Influence of Vectorization on Tiling for Machine 2 66

6.5 Stripplot Showing the Influence of Tiling on Vectorization for Machine 1 67

6.6 Stripplot Showing the Influence of Tiling on Selection Forwarding for Machine 2 67

6.7 Stripplot Showing the Influence of Unrolling on Selection Forwarding for Machine 2 68

6.8 Stripplot Showing the Influence of Unrolling on Vectorization for Machine 2 69

6.9 Stripplot Showing the Influence of Selection Forwarding on Vectorization for Machine 2 70

6.10 Comparison of Baseline Implementation With Best Performing Serial Variants at Scale Factor 1 74

6.11 Influence of Compression and Denormalization on Query Runtime . . 76

6.12 Comparison of Best Performing Variants with Typer and Tectorwise Single Threaded and Multi Threaded at Scale Factor 100 77

6.13 Compile Times of Machine 2 79

6.14 Comparison of Typer and Tectorwise compiled with GCC-8 vs. Clang 13.0.1 Single Threaded at Scale Factor 100 81

A.1 Influence of Tiling on Vectorization 88

A.2 Influence of Tiling on Parallelization 88

A.3 Influence of Tiling on Selection Forwarding 89

A.4 Influence of Tiling on Unrolling 89

A.5 Influence of Vectorization on Tiling 89

A.6 Influence of Vectorization on Parallelization 90

A.7 Influence of Vectorization on Selection Forwarding 90

A.8 Influence of Vectorization on Selection Forwarding 90

A.9 Influence of Unrolling on Tiling 91

A.10 Influence of Unrolling on Parallelization 91

A.11 Influence of Unrolling on Selection Forwarding 91

A.12 Influence of Unrolling on Vectorization 92

A.13 Influence of Parallelization on Tiling 92

A.14 Influence of Parallelization on Unrolling 92
A.15 Influence of Parallelization on Selection Forwarding 93
A.16 Influence of Parallelization on Vectorization 93
A.17 Influence of Selection Forwarding on Tiling 93
A.18 Influence of Selection Forwarding on Parallelization 94
A.19 Influence of Selection Forwarding on Parallelization 94
A.20 Influence of Selection Forwarding on Vectorization 94
A.21 Compile Times of Queries on Machine 1 95
A.22 Compile Times of Queries on Machine 3 95

List of Tables

4.1 Voila Statements to Voila-MLIR Dialect Operations Mapping With Descriptions 31

4.2 Voila Expressions to Voila-MLIR Dialect Operations Mapping With Descriptions 32

4.3 Voila Voila-MLIR Data Types and Mapping to Standard Types . . . 33

6.1 Benchmark Machine Specifications 59

6.2 Spearman Rank-Order Correlation on Performance of Single Variant . 72

6.3 Fastest Variants with Values for the Parameters 73

List of Code Listings

2.1 Example of Smaller Selection in Voila 15

2.2 Example MLIR IR 16

2.3 Example Linalg Generic Operation with Reduction Iterator Representing a Horizontal Sum Operation on Tensors 19

2.4 Example Affine Loop Operation with Reduction Variable Representing a Horizontal Sum Operation on Buffers 19

4.1 Example Voila-MLIR Function Definition 29

4.2 Example Voila Function Translation To MLIR 29

4.3 Example Voila Loop 30

4.4 Example Voila Loop Translation To MLIR 30

4.5 Voila Variable Reassignment 36

4.6 Variable Versioning for Reassignment in SSA form 36

4.7 Voila-MLIR Add Operation 37

4.8 Lowered Voila-MLIR Add 37

4.9 Voila-MLIR Insert Operation 38

4.10 Lowered Voila-MLIR Insert 38

5.1 Example Functions that block further optimizations 42

5.2 Range Query in Which the Between Function Will be Inlined 43

5.3 Resulting Fused Linalg Loops 43

5.4 Sum Aggregation of a Selection 44

5.5 Predicated Sum Aggregation with Elided Selection 44

5.6 Linalg Loops for Conjunction of Comparison Result Predicates Before Fusion 46

5.7 Resulting Fused Linalg Loops 46
5.8 Affine Loops for Smaller-Than Comparison and Selection Before Fusion 47
5.9 Resulting Fused Affine Loops 47
5.10 Linalg Generic Loop for Sum Aggregation Before Tiling 49
5.11 Resulting Linalg Tiled Loop with Generic After Tiling and Peeling With Tile Size of Eight 49
5.12 Affine Loop of a Smaller-Than Comparison Before Tiling 51
5.13 Resulting Tiled and Peeled Affine Loops 51
5.14 Example of Linalg Reduction 52
5.15 Reduction Lowered to Affine Dialect Using Iter Args 52
5.16 Example of Affine Reduction Loop 53
5.17 Affine Reduction Loop Vectorized Through Super-Vectorization . . . 53
5.18 Example of Vectorized, Tiled Affine Reduction Generated From Previous Transformations 56
5.19 Parallelized Affine Loop With Reduction Through Summation 56
5.20 Example of Vectorized, Tiled Affine Reduction Generated From Previous Transformations 57
5.21 Transformed Parallel Loop in a Separate Function called Asynchronously 57

6.1 Query with Joins 61
6.2 Query on Wide Table 61
6.3 Modified Q9 With Compression and Wide Table Selections 61
6.4 Example Linalg Generic Operation with Reduction Iterator Representing a Horizontal Sum Operation on Tensors 70

A.1 Q1 87
A.2 Q3 87
A.3 Q6 87
A.4 Q18 88

1. Introduction

A decade ago, advances of main memory technology were finally able to overcome the classical I/O bottleneck of classical database systems. It was now possible to store terabytes of data directly in main memory, shifting the focus to the new bottleneck of main-memory: I/O and CPU speed. While advances in cache-friendly algorithms allow to optimally exploit the available memory bandwidth [MBK00], the bandwidth also increases continuously as of today. We show this trend in Fig. 1.1, which depicts the development of memory bandwidth in GB/s from 1995 to 2020.

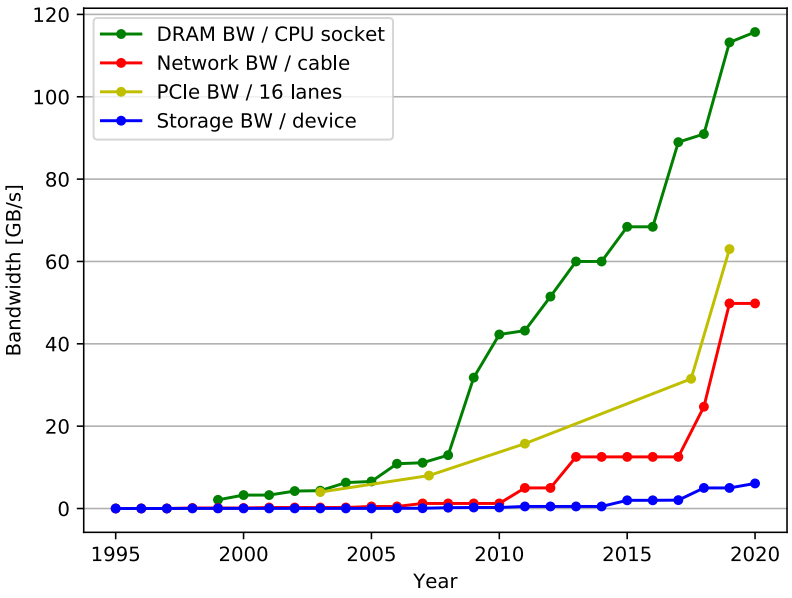


Figure 1.1: Trend of Memory Bandwidth by [MHA18]

However, such advances are not achieved for CPU performance. As of around 2006, scaling of processors in terms of frequency was no longer possible, since for higher

frequencies, it is necessary to have an ever-decreasing of the transistors. However, the power consumption of the transistors does no longer keep up with the size scaling, leading to an ever-increasing heat dissipation on a shrinking surface. Around 2006, this led to microprocessors dissipating heat per cm^2 equivalent of a hotplate (in Fig. 1.2 we see a typical power consumption of around 200 Watts).
To keep up the scaling of transistors and further increase the performance, it became necessary to power-off parts of the processors if not used. This phenomenon is commonly known under the term dark silicon. In order to keep increasing performance, microprocessors started to become heterogeneous. The first developments in this direction were done by using multiple cores in the processor (cf. Fig. 1.2) to achieve a combined higher processing performance, while allowing for a stagnating performance of the individual cores, which we also depict in Fig. 1.2, where the single thread performance determined by the *SpecINT* benchmarks is stagnating.

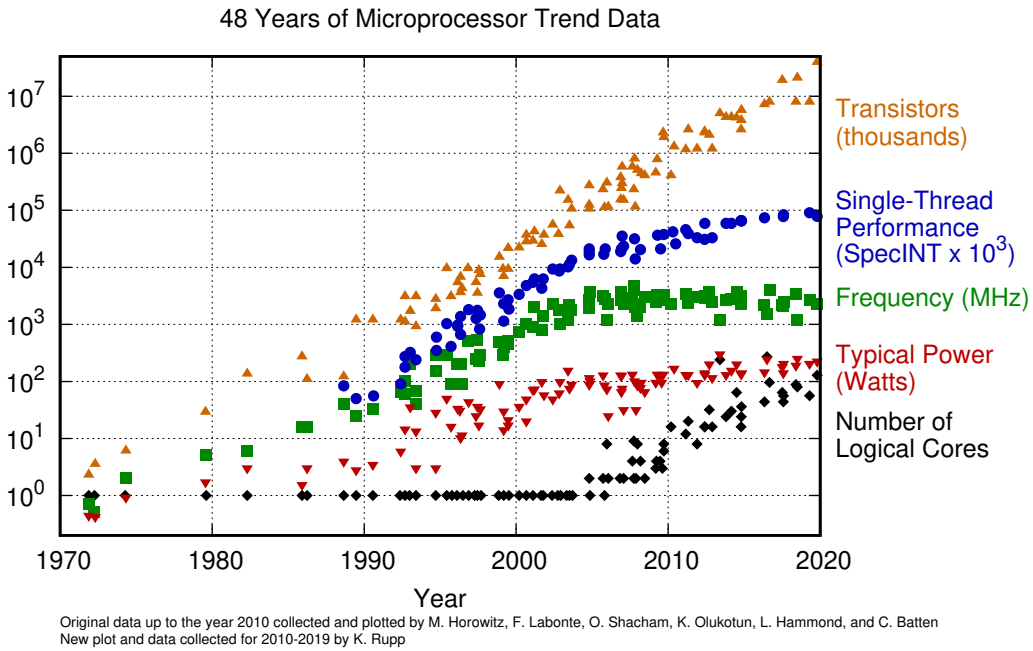


Figure 1.2: Trend of Microprocessor Performance[1]

The database community responded with high-performance parallel algorithms to exploit the new possibilities given by multiple processing cores [Bal+13; DS13]. However, early on it was clear that this approach only works to a certain degree, as formulated by [Amd67]. Since certain parts of every algorithm are of an inherently serial nature and can not be parallelized, scaling in terms of more processing cores is not optimal for every use case. Additionally, due to the emerging end of multicore scaling through the dark silicon problem[Esm+11], hardware became more and more specialized for certain use cases. The first steps in this direction already started with the outsourcing of graphic processing on GPUs, or the adoption of single-instruction multiple-data (SIMD) instruction sets in modern CPUs. With the time, the vector size of these instructions increased and the instruction set became more and more versatile. A more recent example of such specialization is the development of separate tensor processing instruction units within in the processor and as separate systems.

[1] https://github.com/karlrupp/microprocessor-trend-data

They are optimized for inference of neuronal networks with smaller data types and special data flow architectures to achieve a more efficient use of resources. As database operations cover a wide variety of workloads roughly categorized in online analytical processing (OLAP) and online transaction processing (OLTP), many workloads profited greatly from the use of these heterogeneous processing capabilities [He+08; He+09; Wil+09]. Conversely, this posed a problem known as operator placement, as choosing the wrong processing device for the workload could also easily lead to performance drops of several orders of magnitude [Kar+14; BS17].
For static workloads, compilers do a great job optimizing the algorithms to the underlying architecture. This idea is adapted using JIT compilation to optimally exploit compilers' ability to specialize single queries for best performance. Together with operator placement strategies, it is possible to run database queries efficiently on heterogeneous hardware [Bre14]. However, modern compilers do not only use heuristics during compilation, but are capable of profile guided optimization (PGO). PGO uses collected runtime performance data of the compiled binary to track back suboptimal optimization decisions made by the compiler. These profiled data can then be used to guide the compilers' optimization decisions to produce a better optimized binary for the profiled use cases. Even though this optimization technique is able to greatly improve the performance of programs, this idea is not easily applicable to database systems, as their workload varies greatly from query to query and even small differences in the executed instructions can lead to drastically changed performance. This was i.a. demonstrated by Broneske, Breß, and Saake. They showed, that for example the execution of a selection using branching has a significantly different behavior than a selection using prediction, on the same hardware.[BBS13] We show one of their results in Fig. 1.3. The graph illustrates the response time in relation to the selectivity factor of the branching and predicated selections. It can be noticed that predicated selections are faster for selectivities of 10% to 90%, whereas for higher or lower selectivity factors, the branching selection is faster. As a result, the static optimization of code is not sufficient to achieve

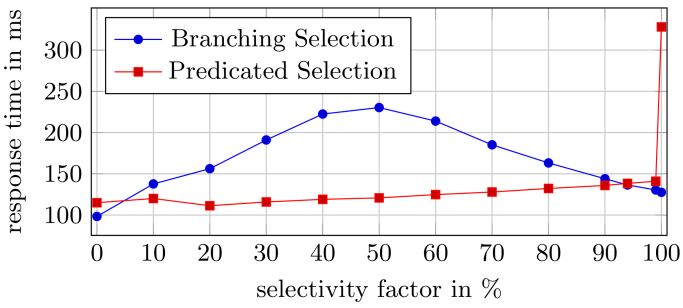


Figure 1.3: Influence of Query Characteristics on Algorithm Runtime for Different Selection Mechanisms [BBS13]

the best performance for every workload on a database system. Therefore, David Broneske proposed *Adaptive Reprogramming*[Bro15] as a technique to dynamically reprogram the executed database operators until an optimal runtime is achieved. The concept is based on a domain-specific language (DSL), which is specifically designed to reassemble hardware-oblivious database operations, and a learning approach to

find the best operator variant. The hardware-obliviousness solves two issues. First, it allows for independence of the type of hardware, which allows the operators to be generated for heterogeneous hardware. Second, it achieves the independence of the operator description in the DSL from actual operator, which gives huge flexibility in the variants that can be generated from the operation. Generated variants can reach from fine-grained variants such as branching vs. predicated selection to coarse-grained variants choosing between different algorithms such as hash-join vs. nested loop join. The level of variance is largely dependent on the design of the DSL. However, since very good heuristics on on coarse-grained variant selection already exist, the design of the DSL should typically focus on a large, fine-grained variant space.
Voila [GB21] is a language designed for such rather fine-grained, hardware-oblivious database operations. The language does not contain a set of hardware-oblivious database operators, but instead allows describing the operators on a high level using a language entirely based on vector data types. This imposes the flexibility to rapidly implement different algorithms in Voila, but keeps the operations abstract enough to have a large pool of execution variants. Voila also comes with a set of backends to generate machine code with, but they are limited in their capabilities to generate code for heterogeneous hardware, and to control over fine-grained variant generation e.g. loop unrolling, branchless code generation, etc. In order to gain these capabilities, we will adapt Voila as a dialect of the Multi Language Intermediate Representation (MLIR) [Lat+21], which is part of the LLVM framework. This allows us to gain full control over the compilation process while at the same time simplifying the machine code generation. The MLIR framework is structured in multiple dialects that represent different paradigmatic views on the code. Based upon this representation, we introduce variability in the lowering infrastructure to achieve adaptive reprogrammability[2]. Subsequently, we demonstrate the effects of our introduced variability and compare our approach to state-of-the-art, handwritten execution engines using the TPC-H benchmark[3].

Contributions of this Thesis

In this thesis, we propose an adaptive reprogramming approach based on the Voila-MLIR Dialect. Therefore, we make the following contributions:

1. We introduce the Voila-MLIR dialect as bridge between Voila as DSL and the MLIR intermediate representation of programs.
2. We build the compilation pipeline to generate JIT compiled executables with LLVM.
3. We extend several MLIR dialects with additional transformation patterns in order to generate a diverse set of variants from the same query execution program.
4. We adapt the code generation pipeline to adaptively apply optimizations that lead to optimized queries.
5. Following the implementation, we demonstrate the variability achieved through our optimizations and analyze the influence of the optimizations on the overall runtime, as well as on each other.

[2]The source code of our framework can be found at: https://git.iti.cs.ovgu.de/blockhau/voila_mlir
[3]Our evaluation results can be found at: https://git.iti.cs.ovgu.de/blockhau/master_thesis

6. To conclude our work, we compare the performance of our framework against Typer and Tectorwise, two handwritten mock-ups of query execution engines for vectorized and compiled query execution.

Structure of this Thesis

Chapter 2 - Background In this chapter, we introduce important prerequisites of our following work.

Chapter 3 - Related Work In this chapter, we give a concise overview over existing approaches for adaptive reprogramming in database systems and related fields such as stream data processing.

Chapter 4 - Voila as MLIR Dialect In this chapter, we introduce Voila-MLIR as a high-level MLIR dialect to bridge the gap between Voila as DSL and MLIR as intermediate Representation. Furthermore, we describe the lowering pipeline we use to produce JIT-compiled machine code and execute the query programs.

Chapter 5 - Rewrite Rules In this chapter, we introduce a set of transformations based upon various MLIR dialects to achieve variability and consequentially adaptivity of the Voila programs.

Chapter 6 - Evaluation In this chapter, we evaluate our approach in several aspects and examine the validity of our evaluation results.

Chapter 7 - Conclusion In this chapter, we wrap up this thesis and its contributions and give a brief overview over future work in this field.

2. Background

In this chapter, we introduce the foundations we utilize and upon which our thesis is build. We start by introducing types of parallel computation, which modern CPUs typically realize and how they differ from each other. Afterwards, we briefly describe common query execution strategies found in modern query engines. Then, we give an overview of heterogeneous database systems and their architecture. In the context of heterogeneous query execution using compiled queries, we introduce the domain-specific language Voila and MLIR, a framework for domain-specific languages with native support for heterogeneous hardware.

2.1 Principles and Architectures for Parallel Computing

In the following, we will give a brief overview of the classification of parallel processing, explain the architecture of modern CPUs and how parallel processing is implemented on modern CPUs.

2.1.1 Flynn's Taxonomy

One of the most common classifications of concurrency in computer architectures is Flynn's taxonomy[Fly66]. While it may not be the most precise and complete taxonomy, it is still a good overview of most of today's parallel computing architectures.

The Flynn taxonomy is based on two aspects:

1. How data are processed
2. How instructions are executed

In specific, the taxonomy considers how data and instructions are streamed for execution. This can be either in single element at-a-time or multiple elements at-a-time per processing step. Originally, Flynn's taxonomy is independent from the

location of the data and instructions, but as all processors follow the Von-Neumann architecture, where data and instructions share the same memory. Subsequently, data and instructions are both streamed from the same memory to the processors. From the two aspects of Flynn's taxonomy, the following four categories of parallel architectures[FR96] can be derived:

Single Instruction Stream - Single Data Stream (SISD)

This architecture represents the classic uniprocessor, in which each instruction processes a single data item, as we visualize in Fig. 2.1. In this architecture, instruction-level parallelism is often introduced by splitting the instruction in a sequence in smaller steps (μOps). These can be performed by different units of the processor simultaneously and keep these units busy by independently working in parallel.

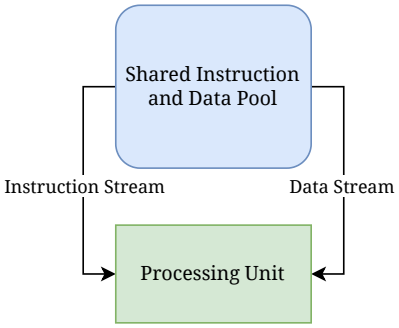


Figure 2.1: Example Illustration for SISD Processing on the Von-Neumann Architecture

This technique is called pipelining, a concept we explain later in the context of CPUs in Section 2.1.2.

Multiple Instruction Stream - Single Data Stream (MISD)

In this class of parallel computation, multiple instructions are executed concurrently on the same data stream. Those architectures recently became of large interest again, as systolic arrays are very efficient in solving neuronal network inference in tensor processing units (TPUs), as for example systolic arrays can process matrix multiplications very efficiently by letting data flow through the fixed matrix-multiply circuit. Similar to our Fig. 2.2, where we show that the same data is fed to all instructions executed in parallel.

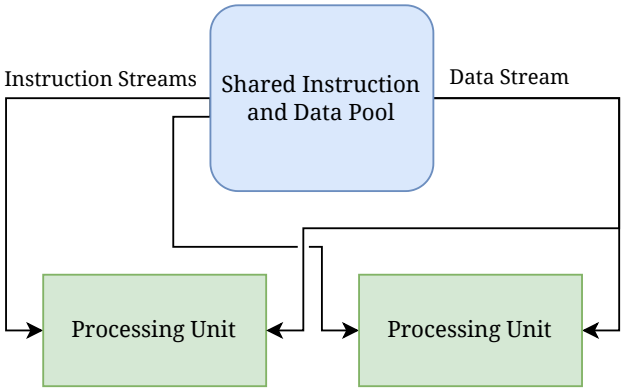


Figure 2.2: Example of MISD Data and Instruction Flow on the Von-Neumann Architecture

MISD architectures use the same data for multiple different processing steps at the same time, and while this could lead to very efficient calculations for certain problems, this architecture is bound to a rather small domain of problems.

Single Instruction Stream - Multiple Data Stream (SIMD)

In this architecture, a single instruction is executed simultaneously on all data items (cf. Fig. 2.3). For example, two four-element vectors are added element-wise with a single add-instruction executed on all data items in parallel.

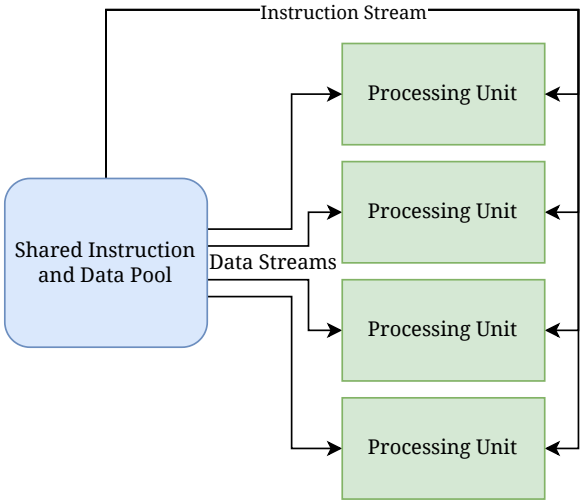


Figure 2.3: Example Illustration for SIMD Processing on the Von-Neumann Architecture

A common implementation of SIMD instructions on x86 CPUs is the AVX instruction-set extension. In addition to CPU extensions, compute units of GPUs are designed as SIMD units to achieve highly parallel computations on large vector data types.

Multiple Instruction Stream - Multiple Data Stream (MIMD)

MIMD architectures allow execution of multiple instructions independently on multiple data in parallel (cf. Fig. 2.4) and thus, split the input problem into smaller sub-problems which then can be solved in parallel as, for example, a parallel sum.

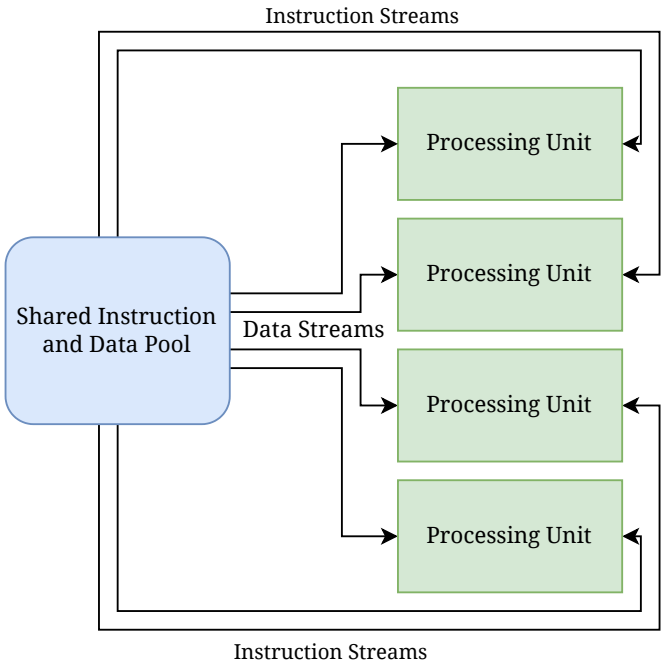


Figure 2.4: Conceptual Illustration of MIMD Data and Control Flow on the Von-Neumann Architecture

MIMD is one of the most commonly used parallel architectures in the form of multicore processors. However, it is not limited to a single processor. Many-core-processor systems such as Intel's Xeon Phi, distributed systems and superscalar processors also count as MIMD architectures.

Each of the four computation types has its domain where it works best and therefore there is no one-size-fits-all solution and different computation architectures exist for different workloads. In the following, we give a brief overview over CPU architectures.

2.1.2 CPU Architecture

To build fast database query engines, it is important to understand how CPUs function and how to optimize for their architecture. Central processing units (CPUs) are optimized to provide good performance for a wide variety of algorithms. They

have to be capable to do any general purpose calculations, manage I/O and dedicated devices. Therefore, the actual execution units are rather small and large parts of the CPU are dedicated to support and increase the performance of those general purpose operations. As the memory bandwidth of CPUs is rather limited in favor of allowing a large main memory, a complex cache hierarchy consisting of multiple stages are needed to hide the main memory access latency and bandwidth limits. In our example block diagram in Fig. 2.5, the CPU block has a common third level cache connected to all cores. It is followed by an L2 and L1 cache per core with a separated L1 cache for data and instructions.

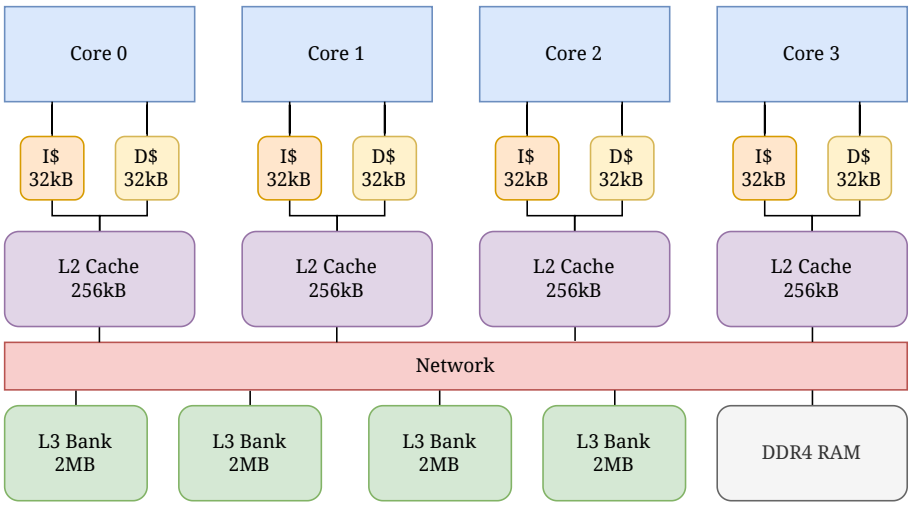


Figure 2.5: Simplified Blockdiagram of CPU Cores Connected With Bus and Caches

In order to achieve a high CPU performance, it is necessary to use algorithms that have data access patterns that allow for the caches to be optimally used. When data are fetched from main memory, they are loaded into all caches from L3 to L1 data and instruction cache. For that reason, frequently reused data have a high chance of being cache resident and therefore reduce comparatively slow main memory accesses. Furthermore, the lower the cache level is, the higher is the bandwidth of the cache, the smaller the cache size and the nearer it is to the execution unit (EU). Because caches are organized hierarchically, all data that are contained in level n must also be stored in level $n + 1$. As modern CPUs usually consist of multiple cores, this could lead to consistency problems, if multiple cores work on the same cached data. To prevent coherency issues arising from this, protocols that can flush those parts of the caches to maintain consistency are implemented.

Most modern CPUs support a form of MIMD parallelism through multiple cores. In our example, the CPU consists of four cores. When multiple cores access data in parallel, they should access data using a strided access pattern (cf. Fig. 2.6), as coalesced memory accesses optimizes the cache efficiency. CPUs usually not only fetch the requested memory into the caches, but also do prefetching to anticipate common data access patterns, for example a sequential scan over a continuous memory region.

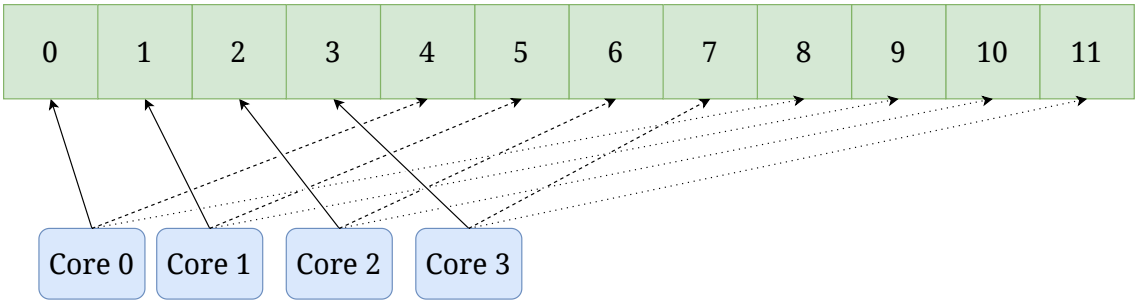


Figure 2.6: Example of Strided Memory Access

Through strided memory access, a single data block can be fetched into the cache and can be used for all cores. The pattern is also sequential and thus profits greatly from the hardware prefetching.

When multiple threads work concurrently, it often is necessary to synchronize their access to shared data resources to avoid data races. For example, the sum of an array has to be calculated in parallel. Each thread sums up a slice of the array, but in the end the partial sums have to be summed up again, to get a correct final result. Therefore, one can wait for all threads to finish and afterwards sum up the partial sums serially. This approach is very simple, but the serialization still has an extra overhead over the parallel sum. Most CPUs have a set of atomic operations implemented, to overcome this problem. Atomic instructions are data-race free, meaning that they can be executed in parallel without interfering with other accesses to the data. In our parallel sum example, this means that after each thread calculated its partial sum, they can use an atomic read-modify-write operation to add the local to the final result. Unfortunately, atomic instructions can be very expensive when not used carefully, as atomic updates lead to a cache invalidation of the data on all cores. For this reason, our parallel sum example should at first calculate a local sum, instead of constantly adding to the global result using atomic instructions.

Another type of parallelism commonly implemented in modern CPUs are SIMD instructions. A popular representative for SIMD instructions is the AVX instruction set. With AVX, it is possible to execute the same instruction on a vector of data, instead of single elements. With AVX 2 and subsequently AVX-512, the vector size has increased of up to 512 bits per vector and gather and scatter instructions have been implemented. These instructions allow fetching data from non-adjacent memory locations into an SIMD vector. Together with the masking feature of AVX-512 that allows conditional execution on elements according to a mask, SIMD instructions became usable for a wide area of applications. However, vector instructions are very expensive in terms of energy requirements, making it necessary to decrease the overall processor frequency to not exceed the available power budget and overheat. This relativizes the performance gains through vectorization and can lead to performance loss for applications that use a mix of vector and normal instructions[BS17].

Alongside instruction and data parallelism, CPUs optimize for SISD parallelism. Therefore, CPUs typically implement their instructions as a sequence of smaller μ-Operations which run within one CPU cycle to realize instruction-level parallelism.

This allows the separate parts of the CPU to be better utilized and kept busy. For example, the fetch-unit of a CPU is only used once per instruction, but with pipelining it is possible to fetch multiple data from memory within a single cycle, reducing the idle time of the fetch unit.

Instruction \Cycle	1	2	3	4	5	6	7	8
I_1	IF	ID	EX	Mem	WB			
I_2		IF	ID	EX	Mem	WB		
I_3			IF	ID	EX	Mem	WB	
I_4			–	–	–	IF	ID	EX

Figure 2.7: Example Pipeline with a Depth of 8 and three Cycle Pipeline Stall

Phases: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Fetch Operands from Memory (Mem), Write Results Back (WB), Stall (–)

However, instruction pipelining has the limitation of not being able to pipeline instructions that have control or data dependencies. This is called pipeline conflict and leads to stalling of the pipeline, as we show exemplary in Fig. 2.7. Stalling in our example is caused by I_4 that depends on the results of I_1. This imposes an ordering constraint and the pipeline has to be filled with no-ops to wait until the result of I_1 is written and can be read again by I_4. One main factor of these pipeline conflicts are branch instructions. Therefore, modern CPUs use large parts of their available chip surface to implement complex branch prediction units and out of order execution. Branch prediction units try to predict which branch is possibly taken and thus allow to speculatively execute this branch allowing deeper pipelining, but introduce a penalty when the branch is mispredicted. Out-of-order execution allows to reorder the order of executed instructions to reduce stall by executing non-related instructions.

2.2 Query Execution

The classic view of database query execution is based on the iterator pattern. SQL-queries are translated to a sequence of operator calls, the data are fed to the operators in an iterator-based fashion. The first operator opens the data input stream and fetches the data one-by-one using the iterator `next()`-calls and forwards its result to the next operator. This minimizes memory needed for temporary results and is called pipelining. There are three major strategies when it comes to details in executing queries:

Tuple-at-a-Time Often implemented as volcano style execution[Gra90], Tuple-at-a-Time is an execution strategy following the iterator pattern, emitting one tuple at each `next()`-call. This results in very small amounts of memory used for intermediate results, but can have large overhead for the `next()` function calls, since the operators have to be called for every tuple. This is mostly a problem for OLAP queries[BZN05]. Therefore, techniques exist to reduce the overhead of repeated `next()`-calls.

Operator-at-a-Time The operator-a-a-time execution strategy is on the other end of the spectrum for query execution. Operations are not called per tuple, but instead all data are processed by the operation and the result is passed to the `next()`-operation. In this strategy, function call overhead is nearly eliminated, because all data is given to the operator at once. While this approach eliminates additional costs from function calls, it generates potentially large intermediate results that do not fit into the cache and lead to slow execution times because of the memory I/O bottleneck. A popular representative of this execution strategy is the MonetDB[Bon+02] database system.

Vector-at-a-Time The vector-at-a-time execution strategy is a trade-off of the function call overhead and intermediate result size. The `next()`-calls returns chunks of data, instead of just single tuples. This reduces the number of `next()`-calls and their overhead, but leads to bigger intermediate results. However, all modern CPUs have large caches and as long as the intermediate results fit into these, no performance penalty occurs. With increased vector size, the performance improvement is proportional to the chunk size on which the operators work, so this allows for optimization to the cache size[BZN05].

All three techniques provide their advantages, depending on the workload. However, different existing realizations of the execution strategies impact their performance as well:

Interpreted Execution For interpreted query execution, the operator pipeline is built as a series of function calls to the operator implementation, through which the data is passed. Even though its performance combined with tuple-at-a-time execution strategy is inferior to most other implementations' performance-wise, it is still often used in classical database management systems such as MySQL, PostgreSQL, etc. because of its simplicity and the negligible overhead of function calls for traditional OLTP queries. A modern approach that speeds up interpreted OLAP queries is the vector-style execution which also allows the combination with JIT-compilation of the vector operations running on the chunks. A popular representative of this processing strategy is Vectorwise[ZVB12].

Compiled Execution This approach generates a compiled version of the query pipeline. This has the advantage of eliminated function call overhead, as those are optimized out during compilation. Furthermore, multiple operators processing the same data can be combined within a single loop, which allows better optimization of the operators in terms of pipelining and cache optimization.

The disadvantage is the added compilation overhead, which should not dominate the total query execution time, and while this is mostly true for longer running OLAP queries, it usually poses a problem for OLTP workloads. For our evaluation, we will use Typer, a mock-up of HyPer[KN11] as a representative of state-of-the-art DBMS with compiled query execution.

2.3 Voila

Domain-specific languages (DSLs) are a commonly used tool to solve problems in a way that is specifically tailored for the domain of the problem itself. Therefore, they are able to solve the problems more efficiently than general purpose languages. Over the time, query execution languages with different levels of instruction granularity have been proposed (cf. [Bon+02; Pir+16]). A language that finds the right level of abstraction for our use case is *Voila*. Voila is a DSL specialized for database operator description at algorithmic level without being too low-level [GB21]. Voila data types are automatically inferred vector data types of dynamic size. All operations are executed on those vectors and offer the feature of masking vectors with boolean predicates. This allows to store temporary results of a selection as bit-vectors and only materialize full intermediate tables when needed, saving memory and processing time.

```
1  select_smaller(input, value)
2  {
3      condition = le(input, value)
4      emit select(input, condition)
5  }
```

Listing 2.1: Example of Smaller Selection in Voila

In Section 2.3 we show the example code of a smaller selection in Voila. The `input` vector and selection `value` are input parameters of the function. Their size and type is dynamically determined and if necessary can be converted. In Section 2.3, a bit vector is based on the result of the smaller comparison of `input` and `value`. Afterwards, in Section 2.3 the result of the selection is materialized and returned.

Additionally to the features of a predicated vectorized languages, Voila allows using different query execution strategies on the granularity of single operators. Therefore, Voila comes with multiple backends lowering to CLite[4], a simplified C dialect and generating data-centric compiled, vectorized-interpreted or mixed code with its flexible FUJI backend. We will use our custom, compiling backend for Voila, which we will base on the following framework.

2.4 MLIR

The MLIR (multi-language intermediate representation) project[5] is a framework to simplify the design and implementation of domain-specific languages. To achieve this goal, MLIR provides a generic intermediate representation (IR), a declarative system to declare IR dialects and common infrastructure to reduce implementation effort[Lat+21]. Furthermore, MLIR is part of the LLVM framework[LA04] and allows to generate native code for a variety of heterogeneous hardware. The key components of MLIR are:

[4]https://myslu.stlawu.edu/~ehar/Spring10/364/clite_grammar.html
[5]https://mlir.llvm.org

- Dialects
- Transformations

Dialects express common computation patterns and data structures efficiently, whereas transformation patterns are available in form of passes to optimize and translate dialects to executable code. Therefore, the syntax of the IR itself is rather simple and represents a textual form of a directed acyclic graph (DAG).

Additional to dialects with their operations, there exist traits that can be assigned to operations to signal that they have certain properties. Example for those traits are the `NoSideEffect`-trait or the `Commutative`-trait. Those traits can be used to enable additional optimizations, for example in common sub-expression elimination (CSE). Additionally, operations can implement interfaces to allow them to be used for a common functionality the interface implements. In the next chapter, we will discuss the `InlinerInterface` as one example.

2.4.1 Language

The MLIR IR has a single static assignment semantic (SSA)[RWZ88], which means that variables can only be assigned once and before its first use. This simplifies and unifies optimization and rewriting of the IR without the need of complex alias analysis[ASU13].

```
1         module  {
2             func @simple(i64, i1) → i64 {
3                 ^bb0(%a: i64, %cond: i1): // Code dominated by ^bb0 may refer to %a
4                 cond_br %cond, ^bb1, ^bb2
5
6                 ^bb1:
7                 br ^bb3(%a: i64)    // Branch passes %a as the argument
8
9                 ^bb2:
10                %b = arith.addi %a, %a : i64
11                br ^bb3(%b: i64)    // Branch passes %b as the argument
12
13                // ^bb3 receives an argument, named %c, from predecessors
14                // and passes it on to bb4 along with %a. %a is referenced
15                // directly from its defining operation and is not passed through
16                // an argument of ^bb3.
17                ^bb3(%c: i64):
18                br ^bb4(%c, %a : i64, i64)
19
20                ^bb4(%d : i64, %e : i64):
21                %0 = arith.addi %d, %e : i64
22                return %0 : i64   // Return is also a terminator.
23            }
24        }
```

Listing 2.2: Example MLIR IR[6]

[6]Taken from https://mlir.llvm.org/docs/LangRef/

MLIR consists of operations, as well as types and attributes and each dialect can declare own types, attributes and operations. Only a basic set of scalar types in addition to tensors is built-in. Furthermore, only a generic, yet expressive syntax that can be overridden by the dialects at need, in addition to some structuring elements are provided. Listing 2.2 is a simple example of MLIR's syntax and structure, it computes either `%a * 2` or `%a * 3`, depending on the boolean value of `%cond`. The `module` operation in Line 1 is the top-level instruction and necessary to define the start of an IR container. In Line 2 follows a function definition. It contains a name, list of input parameters and list of output parameters. In our example in Line 2, the function `simple` takes a 64-bit integer (i64) and a single bit (i1) and returns a i64. The function signature is followed by the function body represented as a *region*. *Regions* in MLIR are built from *blocks* and have no own semantics. Instead, the semantics of a *region* are defined by the operation associated with the *region*. They also define a hierarchical encapsulation, restricting referencing into other *regions* and provide scoping of values.

Blocks on the other hand have a defined semantic. A *block* is a sequence of operations that are executed in order and have a terminator which can return values. The terminator can be omitted for a single *block* in a *region*, or if no value is returned. An example terminator is `return` in Line 22. *Blocks* in MLIR can take a list of arguments, which are bound to values within the *region*, for example, `^bb0` in Line 3 has the arguments `%a` and `%cond`. The introduction of *block arguments* allows control-flow dependent value definition without special cases or *phi-values* that are typically necessary in SSA syntax. In our example, this property is demonstrated in Line 17, where `%c` is defined depending on where it was branched to, either Line 7 or Line 11.

2.4.2 Dialects

Dialects are the main reason for the expressiveness of MLIR. They allow for an efficient implementation of components in their application-specific semantics with the best suiting level of abstraction and thus allow them to perfectly fit their purpose. The possibility to combine multiple dialects for different purposes within a single common IR allows reuse of concepts and efficient implementation and optimization of the involved dialects. For example, affine loops can be mixed with vector types and arithmetic operations inside the loop to provide polyhedral optimizations to vector types on which arbitrary arithmetic operations are able to operate. For this reason, MLIR comes with a diverse collection of dialects on different abstraction levels (cf. Fig. 2.8) that can be used to implement a custom dialect which relies on already implemented functionalities.

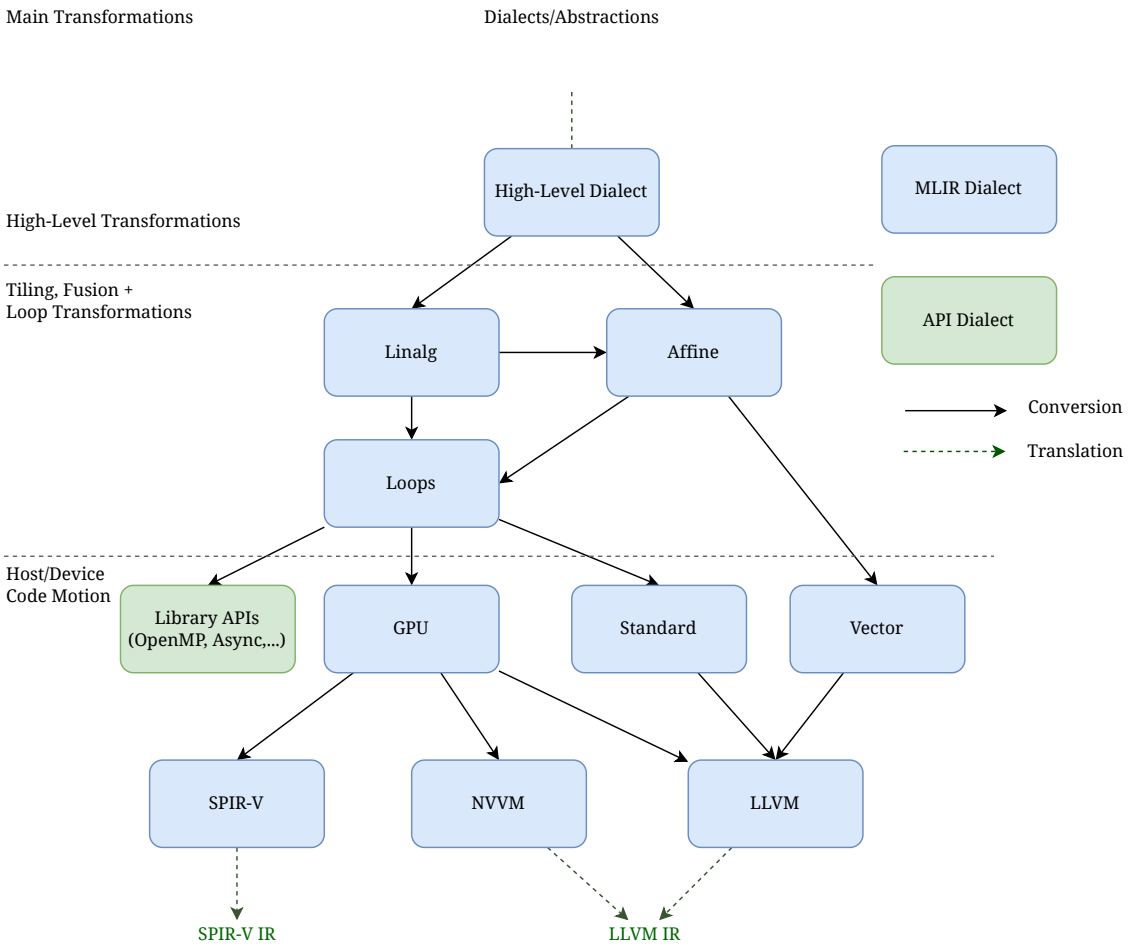


Figure 2.8: Overview of In-Tree Dialects in MLIR

Linalg Dialect

One commonly used dialect of MLIR is the `linalg` dialect. This dialect is a rather abstract dialect working with iterator-based loop-like structures with a range of different iterator types e.g. `parallel`, `reduction` or `window`. Aside from the `linalg.generic`-loop, `linalg` also provides special tiled loops and named specializations of generic operations such as matrix multiplication or convolution. In general, linalg loops can operate on abstract tensors as well as memref types and represent perfect loop-nests with loop bodies that are side-effect free. This is the main restriction of linalg-based loops, but allows easy parallelization, analysis and optimization. In Listing 2.3, we show an example code for a `linalg` generic loop returning the sum of a tensor.

Line 1 and Line 2 define the mapping of the input data dimensions in the loop iterations. `#map0` is an identity mapping and `#map1` is the mapping of the output element to no element, as the return value of Line 9 is the output value of the next iteration. In Line 3, the initial value of the iteration parameter is defined, in this case zero, the neutral element of the addition. Afterwards, in Line 4 the indexing maps and iterator types, in this case only one respectively, for the input dimensions are specified and input as well as output parameters are passed to the loop operation(cf. Line 5). Afterwards, in the region of the loop, the region which will be executed for

```
1  #map0 = affine_map<(d0) → (d0)>
2  #map1 = affine_map<(d0) → ()>
3  %cst = arith.constant dense<0> : tensor<i64>
4  %sum = linalg.generic {indexing_maps = [#map0, #map1], iterator_types = ["reduction"]}
5  ins(%arg0 : tensor<100×i64>) outs(%cst : tensor<i64>)
6  {
7    ^bb0(%arg1: i64, %arg2: i64):
8    %2 = arith.addi %arg1, %arg2 : i64
9    linalg.yield %2 : i64
10 } → tensor<i64>
```

Listing 2.3: Example Linalg Generic Operation with Reduction Iterator Representing a Horizontal Sum Operation on Tensors

the specified index mapping is described. In this case, each element of the input is added to the current output element in Line 8 and the accumulated value is returned as the next output element for the next iteration in Line 9.

Affine Dialect

Another dialect we will use for the large part of our transformations is the `affine` dialect. The `affine` dialect is based upon the polyhedral description of loops. Thus, the dialect offers nested loop structures and transformations. The loops allow for affine buffer access that can be easily analyzed and based upon this, non-affine side effects can be restricted, allowing for efficient transformation on buffered loops. In Listing 2.4, we show the reduction example of the `linalg` dialect written as affine loop.

```
1          %arg0 = ...
2          %cst = arith.constant dense<0> : tensor<i64>
3          %sum = affine.for %i = 0 to 100
4          iter_args(%sum_iter = %cst) -> (i64) {
5              %t = affine.load %arg0[%i] : memref<100xi64>
6              %sum_next = arith.addi %sum_iter, %t : i64
7          // yield current iteration sum to next iteration %sum_iter or to %sum
8          // if final iteration.
9              affine.yield %sum_next : f32
10         }
```

Listing 2.4: Example Affine Loop Operation with Reduction Variable Representing a Horizontal Sum Operation on Buffers[7]

The loop bounds are derived from the size of the buffer whose elements are added (cf. Line 3). The buffer itself is accessed through an affine load instruction in Line 5 that is restricted to affine induction values like the loop iteration counter. The sum is carried as additional loop induction value that is initialized before the loop and updated with every iteration through the yield instruction. When the last iteration finishes, the final result is returned from the loop.

[7]Taken from https://mlir.llvm.org/docs/Dialects/Affine/#affinefor-mliraffineforop

SCF Dialect

The `scf` dialect represents arbitrary control-flow structures, such as loops and conditionals. At this level, the control flow is still structured and not only conditional branch instructions which have lost their structured information, but there are no guarantees regarding the semantic. With `scf`, arbitrary side effects can be modeled, which makes transformations on this dialect rather complicated. This dialect can be understood as an intermediate layer between the high-level dialects and low-level branch-based control flow. Therefore, we do not apply optimizations on this level, except for the parallelization in the `async` dialect which is based upon the `scf.parallel` loop.

LLVM Dialect

The `llvm`-dialect in MLIR is a one-to-one mapping of a subset of LLVM IR which allows for a fast translation between MLIR and LLVM. Both follow a very similar SSA structure and for translation only the blocks with parameters have to be transformed to branch labels with ϕ-nodes if the block is conditionally executed. As we do not apply any optimizations on this level yet, this dialect is only relevant for lowering out of MLIR and compiling an executable program.

2.4.3 Pass Infrastructure

In MLIR, rewrite-patterns are used to translate code between different dialects and apply optimizations. Multiple rewrite-patterns can be combined to a pass which can be called during the compilation. There exist passes that translate a language into an MLIR dialect, convert operations from one dialect into another, or just optimization passes that increase the performance of the code. Passes can be classified in the three following categories:

Operation-Specific Passes are passes that work specifically on a single operation type, for example the `linalgTiledLoopsToSCF` pass converts `linalg.tiled_loop` operations to `scf.loops` with explicit counter variable and step size.

Operation-Agnostic Passes are passes operating on the level of the pass managers' operation type. These passes are more general, mostly applying inter-operation transformations such as inlining or CSE.

Analysis Passes are passes that work like their transforming counterparts, but are prohibited to apply transformations to the code. Instead, they apply computations on the code and collect analytic information. Furthermore, they are not implemented as normal passes, but as freestanding classes that are called only on-demand and their results are cached in order to avoid costly re-computations of already applied analysis.

2.4.3.1 The Pass Manager

The management of pass application, as well as related options such as application scope, order and benefit are managed by the integrated pass manager. Pass managers are collections of passes which run on operations of a specific type. The highest level of operation on which pass managers can run is the module level. If lower-level operations, such as `FuncOp` are used as entry point, a new pass manager can be nested within the pipeline of the upper level one. This results in a nesting of passes corresponding to structural nesting within regions on the IR level.

The nesting and specification on which operations the passes of the pipeline run, allow concurrent execution of passes on multiple operations independently. Aside of their main task, pass managers support detailed statistics on executed passes such as timings, execution order and information provided by run passes. Furthermore, pass managers provide a set of instrumentations which allow running code between certain passes in the pipeline, as well as enable failure detection and reproduction utilities.

2.4.3.2 Bufferization

Bufferization is the translation of tensors in the side-effect free domain to the buffer domain with memory references. The bufferization in MLIR spans through multiple dialects and translating all dialects from a tensor semantic to a buffer semantic is a very involved task. During the time of the writing of this thesis, the bufferization workflow of MLIR changed slightly and it can be expected that the information of this section is outdated. Therefore, we only briefly describe the process of bufferization for our framework. To correctly bufferize all operations in MLIR, each dialect has its own bufferization pass. Some dialects just allow to mix buffering semantic with tensor semantic, like the `linalg` dialect. Other passes rely on casting of tensors or `memrefs` to the correct type on which the operations can operate. Later on, the operations are lowered to operations that work in the buffer domain and the casts can be eliminated.

Additionally to the dialect-specific bufferization passes, there is a comprehensive bufferization pass that relies on the definition of bufferization steps for the operations it encounters. On the other hand, it has a set of optmizations such as buffer hoisting, scalar replacements and automatic deallocation implemented to remove duplicate buffers, increase buffer reuse and proper deallocation.

For bufferization in our pipeline, we use the comprehensive bufferization approach without deallocation, as the deallocation is error-prone and does not corretly support returning buffers from functions. Thus, we run the separate deallocation pass after the bufferization.

3. Related Work

In this chapter, we describe previous work relevant in the context of this thesis, to show the novelty of our approach and help contextualize our work. At first, we give an overview over some state-of-the-art frameworks for adaptive query execution, following a brief digression into the field of execution adaptivity and discuss some common techniques to achieve adaptivity. We close this chapter with an overview of related domain-specific query processing languages, stencil computation and rewrite specifications for optimization adaptivity, as well as compiler techniques for adaptivity.

Query Execution Adaptivity

Răducanu, Boncz, and Zukowski introduced "Micro Adaptivity in Vectorwise"[RBZ13], which is a vector-at-a-time database management system. Their approach bases on the observation that depending on the processor, the compiler and the query, a static generated operator can never execute optimally under all circumstances. Their solution is a variant pool of handwritten operator flavor templates, such as branch-less selections, loop unrolling, fission, etc. that are compiled with different compilers and template instantiations and thus generate differently performing operators. During query generation, for each operator a flavor is selected out of the flavor pool and the flavors are combined into entire queries. As the volcano-style query execution is an interpreted execution, the flavors are maintained as different symbols in different flavor libraries and are loaded at need. Therefore, the selection of operation flavors and composition of operations is just a call to the function pointer of the selected flavor. To find the best flavor for each operation, the authors formulate the problem as a multi-armed bandit learning approach. This approach allows for learning of the best combination, while queries can still be executed, thus no offline-learning or training data set is necessary. To mitigate the problem that the system is always choosing the same flavor and gets stuck at a single flavor, the ϵ-greedy strategy is used to make the learning non-stationary resistant.

With the rise of heterogeneous databases, Breß et al. proposed the Hawk-Framework[Bre+18] that is not limited to adaptivity on CPUs. In contrast to the previous approach,

this approach relies on JIT-compiled tuple-at-a-time query execution. For adaptivity, operation templates are defined representing not only different variants of the same algorithm, but partially also different algorithms, e.g. different hash table implementations. Those operations are part of the pipeline programs that are generated by the query translation and represent a hardware-oblivious query blueprint. Additional to the operation variants, the overall pipeline program is configurable by for instance the type of parallelism that is used to allow optimization to different heterogeneous targets. This blueprint takes a configuration assigned from the variant optimizer. This configuration defines which variant of the operators is generated for the later target processor. To select an efficient variant, the variant optimizer learns the optimal variant configuration for the processor on a training datasets by executing all possible options for each variant and collecting the best parameters before the query execution starts. To reduce the learning time, Hawk also uses some additional heuristics to prune the exploration space. After the query is lowered to target dependent code, the code is JIT-compiled and optimized by OpenCL and afterwards executed.

Both approaches implement their variability using a template mechanic to generate variants in a high-level language that is potentially expensive to compile. Their runtime performance is still entirely dependent on optimizations of the underlying compiler and can only be influenced within certain boundaries. The template system has the inherent disadvantage that more complex optimizations that require analysis of the control or data flow are not possible to implement without large effort. In contrast, our approach uses a lightweight DSL that is independent of a particular implementation and interweaves the compilation process with the variability generation. This allows for faster compile times and greater flexibility.

Execution Adaptivity in Other Fields

Aside from adaptive database query engines, there exists an adaptive stream processing engine, called Grizzly[Gru+20]. It has a similar learning approach as "Micro Adaptivity in Vectorwise" with continuous variant variation, but operates on streams compiling a variant for a query window and evaluating the variants' performance. The query begins with a generic variant that only utilizes static knowledge like constant comparison optimization. During the execution of the first variant, data and execution statistics are collected. After the first executed window, the collected profiling information from the previous execution are used to generate a new, optimized variant that is executed in the next step. If in an executed window a deoptimization takes place, i.e. an assumption on the data is incorrect, or the performance of the generated variant degrades, a switch to the generic variant is done and the variant learning is redone.

Another high-level framework for GPU auto-tuning[Gra+12] has been proposed by Grauer-Gray et al. The programs are written in the C language from which a set of variants is inferred, which are explored and evaluated in an extensive search. Afterwards, lower level CUDA/OpenCL code is generated and annotations for the best variant are generated.
Another approach which directly integrates the adaptivity in the program is Adaptive

Code Refinement[SHB17]. The program code is annotated with pragmas that provide domain-specific approximation information on the loops in the program. These pragmas are used in the following step to reason about the iteration domain and optimize the program using polyhedral techniques. The optimization itself can be done static on compilation or dynamic on runtime by using multiple threads that monitor the performance, execute the variants or generates new variants.

Domain-Specific Languages

In the field of domain-specific languages for query processing, the Voodoo Language [Pir+16] has been proposed to efficiently use a vector algebra for query processing on heterogeneous hardware. The language bases on high-level vector primitives and thus allows hardware-independent expression of vector programs. These programs are lowered to OpenCL and thus can be executed on a variety of heterogeneous processors. With the LIFT-Language[Ste+15], Steuwer et al. proposed a functional high-level language tailored for the expression of program rewrite rules for optimization and implementation of data-parallel processing. As a follow-up, [Hag+20] proposed RISE and ELEVATE[Hag+20]. The RISE language is used to define high-level computations in a functional way, while ELEVATE can be used to describe different implementations of the RISE language constructs. Together, they can generate code specially designed for the target processor on which the program runs. The PATUS framework[CSB11] is a framework for automatic optimization of stencil computations. It uses its own domain-specific language based on the C language as well, as domain-specific knowledge for auto-optimization.

4. Voila as MLIR Dialect

In this chapter, we introduce the architecture of our adaptive reprogramming framework and its DSL implementation, with which we will be able to generate variable code which is individually optimized for hardware and query properties. Therefore, we start with an overview of all components, followed by the translation of Voila to MLIR. This needs several adaptions to bring Voila into the form of an MLIR dialect. Therefore, we explain the most important steps of parsing, single static assignment form, as well as type and shape inference. Afterwards, we explain which dialects of MLIR we use to generate executable code and how we actually run this code.

4.1 Architecture Overview

In Fig. 4.1, we give an overview of the components in our framework. As the goal of our framework is to enable adaptive query execution, we focus on the generation of operators that are already translated from SQL to Voila. Therefore, we ignore the translation of SQL in this thesis and instead assume an appropriate implementation of SQL operations. Gubner and Boncz already demonstrated that an implementation of SQL operators is definitely feasible and we already showed in an example that selections can be simply implemented as shown in Section 2.3. And while an efficient translation of SQL to Voila will undoubtedly take some effort, the challenge seems to be more on the technical side and thus out of scope for our thesis.

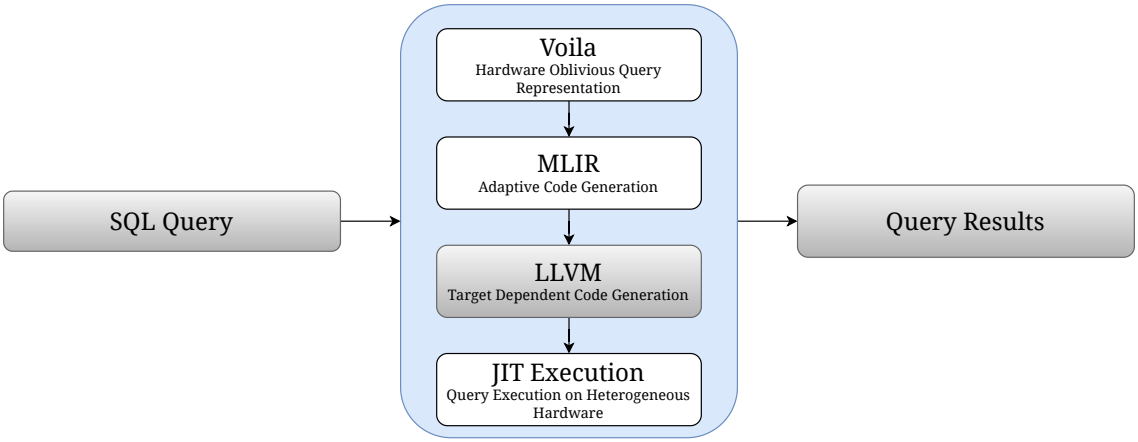


Figure 4.1: Architecture of our Adaptive Reprogramming Framework

The framework itself starts at the translated query plan in Voila that represents a hardware-oblivious description of the operator algorithms to execute. From there-on, we translate Voila to the Voila-MLIR dialect. This dialect represents the Voila language in an intermediate language syntax within the constraints of MLIR. This includes type inference and single static assignment (SSA) semantics, as well as transformation of operations to become side-effect free and thus better automatically optimizable.

With the translated Voila-MLIR dialect, it is now possible to gradually lower the instructions through different MLIR-internal dialects to more hardware sensitive instructions and at the same time optimize for the underlying hardware. This MLIR with its dialects and transformations represents the core of our framework and the source of our variability. The end result of the lowering passes is a module that contains only the LLVM dialect, as well as other hardware-near dialects such as SPIR-V, NVIDIA PTX, Vulkan or ARM SVE.

In the next step, the LLVM-MLIR dialect is translated to LLVM IR and an interface function that makes the code callable from C code is generated. Afterwards, the code is compiled to executable code. In this step, no major algorithmic optimizations happen. In this step, they are restricted to generate efficient platform-dependent code that makes the best use of the available hardware, e.g. by choosing the right instruction set, optimizing register usage and code size. Therefore, we do not consider this optimization knob for our approach, as it leads to very fine-grained adaptivity with rather small potential of increasing performance in comparison to the heuristics already used by compilers.

The compiled code can now be called by an execution engine that ensures correct invocation of the newly compiled function, along with correct linking and parameter passing. In our framework, those engines are wrapped in a program that completely hides the invocation, parameter passing, and result reading complexity. Furthermore, the execution engine features a binary code cache to avoid recompiling the program on repeated execution. We also integrate a profiler in the execution wrapper that is able to profile the executed code using the available performance counters on the

hardware. This ensures low measurement overhead and high precision of the profiling results.

4.2 Voila as MLIR Dialect

The Voila-MLIR Dialect has the goal to represent the Voila language as a dialect in MLIR's intermediate representation. It is designed to be similar to the original language, while fulfilling the restrictions of MLIR. This allows for a simple and efficient translation of Voila to MLIR, the use of MLIR for high-level optimizations on the semantic of Voila, and also simple lowering within MLIR to other dialects to enable adaptivity.

4.2.1 Parsing Voila

In order to translate Voila to MLIR, we have to parse the code using Voila's language specification. As there exists no full specification of Voila, we derived our own grammar based on the available specification and examples in the original Voila paper. Based on this grammar, we used RE/flex[8], a flex-like lexer generator with support for Unicode to generate a lexer for tokenization and Bison[9], a Look-Ahead LR parser (LARL) parser generator[DeR69] to generate a parser which transforms the tokenized input to an abstract syntax tree (AST). Both, the generated parser and lexer are re-entrant, making it possible to concurrently parse multiple Voila files.

Voila's building blocks are functions, blocks, expressions and statements. Voila expressions are defined as function applications that produce a result and have no side effect, whereas statements produce a side effect and no results. This mapping is not possible in MLIR, as there exist no such constructs as statements and expressions. Instead, those properties for the operations are specified by traits. In the following, we describe the mapping of Voila's language constructs to MLIR's IR.

4.2.1.1 Functions

Functions usually implement an operator, while the main function combines the operators in a pipeline to form a query. In our dialect, this query pipeline is specified in the `main`-function, which acts as entry point for query execution.

```
1 def example_func(int_para)
2 {
3     emit add(int_para,1);
4 }
```

Listing 4.1: Example Voila-MLIR Function Definition

```
1 func  @example_func(%int_para:i64) : i64
2 {
3  %const = 1 : i64
4  %inc = voila.add %int_para, %const
5  voila.emit %inc
6 }
```

Listing 4.2: Example Voila Function Translation To MLIR

[8] https://re-flex.sourceforge.io/
[9] https://www.gnu.org/software/bison/

We translate Voila functions one-to-one to MLIR's built-in function operations with a region containing a single block of statements, as we show in Listings 4.1 and 4.2. The function name is the same, but as MLIR is strong typed, its signature is extended by the types of the parameters and results. We do also introduce the `emit`-Statement as an optional block terminator, if the block returns any values (cf. Line 5). Blocks have their own value scope, and isolate their contained values from accesses from outside the block. Those blocks can contain an arbitrary number of operations that are executed in the order of the control flow. The operations are also explicitly typed and all usages of variables or operation results are brought in SSA form.

4.2.1.2 LOOPs

Loops are the only explicit control-flow statement of Voila. They iterate on a supplied predicate, until all values in this predicate are false. We map this operation to a new custom MLIR operation with a block containing the loop operations and a parameter as loop induction variable, cf. Listings 4.3 and 4.4.

```
1 cond = 100;
2 val = 0;
3 val = LOOP | cond
4 {
5     val = add(val,1);
6     cond = sub(cond, 1);
7     emit val;
8 }
```

Listing 4.3: Example Voila Loop

```
1 %cond = 100 : i64
2 %val = 0 : i64
3 %val = voila.loop
4  (%cond : i64) → i64
5  {
6  %val = voila.add(%val, 1) : i64
7  %cond = voila.sub(%cond, 1) : i64
8  voila.emit %val : i64
9 }
```

Listing 4.4: Example Voila Loop Translation To MLIR

This operation functions like a while-loop, checking the supplied predicate on each iteration and executes the associated operation block until the predicate is false. As LOOPs have a statement-semantic, they do not return values, but instead their body has to be handled as a side effect, because the predicate has to be modified in the body. Otherwise, this would lead to an infinite loop. As MLIR per default assumes that operations can have side effects, we do not have to take extra steps to restrict possible rewrites in MLIR.

4.2.1.3 Statements

In contrast to Voila, which also allows expressions in a block during parsing, we wrap those expressions in a statement to simplify parsing and have the same effect as the containing expression. However, MLIR as an intermediate representation does not differentiate between statements and expressions. In MLIR they are both operations and operations can generally have side effects or return an arbitrary number of results. For this reason, Voila statements can simply be mapped to MLIR operations of the Voila dialect. In Table 4.1, we show the statements that are mapped to MLIR-operations.

Voila-MLIR Operation	Voila Operation	Description
`LOOP |`	`LOOP`	Control flow structure executing its assigned code block until condition is false
`=`	`assign`	Assignment function
`emit`	`emit`	Function terminator signalling return values

Table 4.1: Voila Statements to Voila-MLIR Dialect Operations Mapping With Descriptions

While most operations can be simply mapped to MLIR, most operations would have to expose side effects that can only be represented using memory references. This restriction would mean that many transformations that rely on an SSA semantic without side effects can no longer be applied. Furthermore, the optimizations would have to prove that transformations do not modify the semantics of the program due to side effects. To avoid these problems, most of the Voila statements are implemented as expressions returning their results. This changes the semantics of the operations slightly, but allows for better optimization and analysis capabilities through their property of being side-effect free and following the SSA-semantics of MLIR. Therefore, we list those operations in the expression table (cf. Table 4.2).

4.2.1.4 Expressions

Expressions in Voila are side-effect free function applications. We make MLIR aware of this property by implementing the built-in `NoSideEffect`-Trait for all expression operations. As for the statements, we list all Voila operations and their counterparts in the Voila-MLIR dialect in Table 4.2. Additionally, we include Voila statements that we reimplemented as side-effect free operations in MLIR.

Most of those operations are classical functions, taking a fixed amount of operations and returning a single result. However, we have modified some Voila operations to take a variadic amount of arguments or return multiple results. This is especially useful for some operations related to hash tables. For example, the hash operations can hash multiple columns. Therefore, it can take multiple arguments and returns a single result. Another example is the insert operation. This operation is a statement from Voila modified to be an expression. The operation takes a vector of hash values, as well as the corresponding value vectors, and inserts the columns into the hash tables, which are then returned. The rationale for this lies in the design of our type system.

Voila-MLIR Operation	Voila Operation	Description
Comparison/arithmetic/logic		
`eq(x, y)`	`eq(x, y)`	Elementwise Equality comparison
`ne(x, y)`	`ne(x, y)`	Elementwise Inequality comparison
`leq(x, y)`	`leq(x, y)`	Elementwise Smaller-equals comparison
`and(x, y)`	`and(x, y)`	Elementwise Conjunction
`not(x, y)`	`not(x, y)`	Elementwise Negation
`add(x, y)`	`add(x, y)`	Elementwise Addition
`div(x, y)`	`div(x, y)`	Elementwise Division
...		
Hash Table		
`hash(c...)`	`hash(c)`	Variadic hash function, in Voila only non-variadic
`insert(h, c...)`	–	Variadic insert function building a hash table for hashes h and values
`lookup(h, c..., ht...)`	`bucket_lookup(ht,b)`	Variadic lookup of indices for supplied hashes, Voila Version looks up buckets instead of values
Filter		
`select(c, p)`	`scan(c,p)`	Create new Column from values `c` where `p` is true
`gather(c, i)`	`gather(c, i)`	Read Values as Indices and Return a new, compressed Column
Aggregates		
`aggr(type, x)`	–	Scalar aggregation
`aggr(type, ht, i)`	`aggr_*(c,i,v)`	Grouped Aggregation
Variables and Constants		
`x,y,z`	`x,y,z`	Variables
`false,1,2.0`	`x,y,z`	Constants
`sum,avg, count, min, max`	–	Aggregation Types

Table 4.2: Voila Expressions to Voila-MLIR Dialect Operations Mapping With Descriptions

4.2.2 Type System

As Voila is a weakly typed language and types are inferred at compile-time, there is no predefined type system. However, as Voila is lowered to CLite[10], for type inference, standard C data types are used. MLIR allows for more fine-grained choice of data types. For example, using an integer of specified size in bits. For our implementation, we only use data types that are used for standard C types by LLVM, except for boolean values. In C, the smallest addressable unit is a byte, in LLVM it is possible to control single bits.

Standard C Type	Voila-MLIR Type
int	i32
long	i64
float	f32
double	f64
char	i8
bool[11]	i1

Table 4.3: Voila Voila-MLIR Data Types and Mapping to Standard Types

As strings typically consist of multiple characters and a single character alone potentially can only be represented by multiple bytes, those strings could be represented by integer vectors or an own dialect modelling a custom string type and its operations. Like in the original design of Voila, we ignore modelling string data types. In the long term, the MLIR framework offers the flexibility to extend this type model quickly through integers of nearly arbitrary size, fixed precision numbers or even custom data types. However, for the scope of this thesis we kept the type system relatively simple.

4.2.3 Representing Voila's Vectors in MLIR

One of the core features of Voila is the representation of data as vectors. Now, that we defined the scalar types that we use, we define our representation of Voila vector types in the Voila-MLIR dialect. With the vector dialect, MLIR has direct support for vector data types that could be used to map Voila's vectors to MLIR. The major drawback for this approach is that the semantics of MLIR vectors are designed as a rather low-level dialect to map to hardware vector representations, thus they are designed for rather small vectors that are directly stored in registers and for which shuffle maps and constants can directly be stored in code. Using this dialect for entire columns would result in longer compilation times, as well as register spilling and very large final executables. Another approach is the mapping of Voila vectors to MLIR `memrefs`. `Memrefs` represent blocks of memory, they are more generic than vectors and do not suffer from the same modeling restrictions, but as they represent memory addresses, they can be aliased and certain optimizations

[10] https://cs.colby.edu/courses/F19/cs333/notes/3.Syntax(5).pdf
[11] Actually, bool is just a macro in C and points to the C99 _Bool type

are not available, because modifications of `memrefs` are inherent side effects in an SSA semantic. The third option is the use of the high-level construct of tensors. Tensors are n-dimensional collections of a single data-type and have no concrete memory representation. Operations on tensors are side-effect free, generating a new tensor upon modification. As the SSA semantics is free of side effects, more optimizations are possible on tensors. On the other hand, this poses a limitation on certain algorithms such as hash table build, which relies on consecutive insertions with linear probing. Furthermore, tensors have the limitation of being homogeneous and are only allowed to contain a single, scalar data type. Therefore, it is not possible to store a collection of tensors in another tensor. This poses restrictions on the layout of our relation and table representation. However, the alternatives of using vectors or `memrefs` also have the limitation of heterogeneous collections, making them equally unsuitable in this regard. In order to have the most optimization options, we choose to represent Voila vectors as MLIR tensors and we solve the problem of collection of vectors by using multiple decomposed 1-D tensors that are only logically grouped.

4.2.4 Type and Shape Inference

One of the core properties of the Voila language is that it is a weakly typed language, which makes algorithms largely type-independent. To infer the correct types in the program, only the input parameter types are needed, everything else can be deduced from a type hierarchy and the knowledge of operation input and output constraints.

Type Hierarchy

For our type system, we decided to use a hierarchy similar to C's type conversion conventions. We depict our type hierarchy in Fig. 4.2.

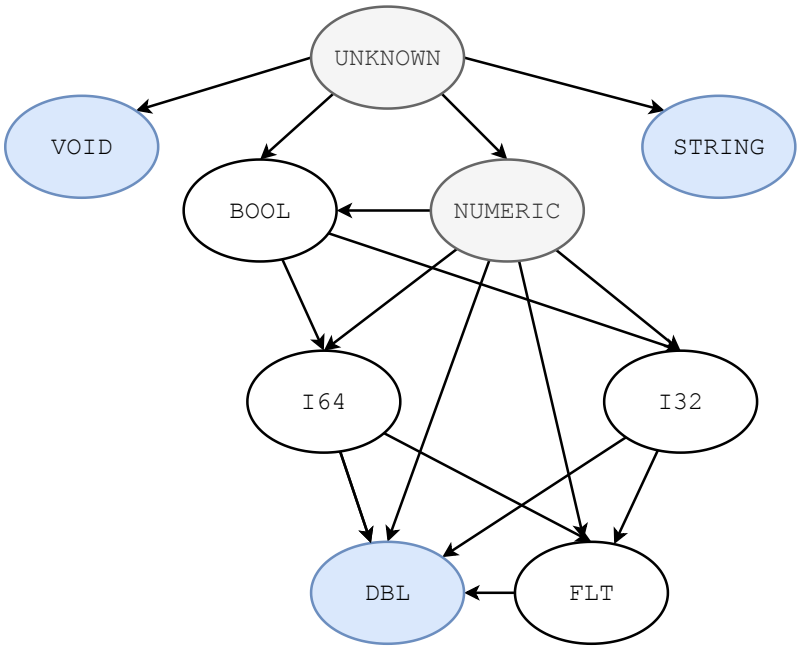


Figure 4.2: Type Hierarchy of Voila

For our hierarchy, the most general type is the `UNKNOWN`-type, which is an abstract type used only for placeholder purposes. If we are unable to infer any other type than `UNKNOWN`, our inference has been unsuccessful and the program is not valid. The most general placeholder type `UNKNOWN` is specialized to physical types such as the `VOID`-type, which is a type indicating that no value exists, the `BOOL` type, `STRING`-type or the conceptual `NUMERIC` type. The `NUMERIC` type is a group of types on which calculations can be done and it has to be specialized to a concrete subtype before it can be used. There exist integer and floating-point types, between which the order of specialization is given towards the next larger type and from integer types to floating-point types. Additionally, the `BOOL`-type can be interpreted as an integer type with a value of either zero or one. It is important to note, that this hierarchy exists only as a logical level to determine whether variables passed to the operations are compatible with the type constraints set on the parameters. This does not mean that for example the `BOOL`-type is implemented as an 64-bit integer in hardware, but it has a known conversion and can be interpreted as an `NUMERIC` (eg. `false` = 0 and `true` = 1). To find the best matching data type for the context in which the value is used, we use a type inference algorithm, which we discuss later. Additionally to the types that have to be inferred, we also have to infer whether the variables are scalar or tensor types and try to determine a static size for the tensor. Therefore, we define that shape conversion from scalar to tensor types is only possible by broadcasting. Tensor to scalar conversion is only allowed by operation that produce scalar values.

Operator Constraints

As mentioned before, the types are not only inferred by the type hierarchy, but also from the operations that are executed on the variables, as well as their results. Therefore, each operation in form of an implicit signature with the types and shapes the operation can operate on and the result types and shapes it returns. For example, the `add`-operator takes two `NUMERIC`-operands and returns a `NUMERIC` result. Additionally, the addition can handle scalars and tensor shapes. For tensor addition, we restrict the tensor shape to be the same for both operands, or one of them being a scalar that is then broadcasted to the correct shape.

Inference Algorithm

Our algorithm is loosely based on the Hindley–Milner type inference algorithm [DM82] and is a stand-alone pass after AST generation and before MLIR translation. We start out with our input-parameter types and traverse the parsed Voila AST, deducing variables in a top-down, depth-first fashion. The types of the input parameters have to be completely deduced and are not allowed to be changed, as this would change the types that have to be passed from outside into the JIT execution environment. Therefore, these types are our start point for the inference. The types of the variables can only change according to the type hierarchy. When two variables have a known, but different types, they are unified by finding the most general common type. For example, an `add` operations has the parameters `%a` with type `DOUBLE` and `%b` with type `NUMERIC`, then both variables need the same type, as add operates only on operands of the same type. Therefore, `%b` is deduced to type `DOUBLE`. The result of the add operations is again a `DOUBLE`, as the `NUMERIC` type is not allowed to be converted

to, from `DOUBLE`. In parallel to the type inference, the shape inference takes place. Similar to the type unification, shape inference has a unification too. Unification for shapes works as follows: all variables begin with an undefined shape that can either be deduced to be a scalar shape, or a shaped tensor, although the shape can still be undefined. However, if the shape is defined once, it can not change anymore. Consider the `add` operation again, with the variables `%a` and `%b`. Furthermore, let `%a` be of tensor shape with a size of 100 elements and `%b` be of scalar type. In this case, the two shapes can not be unified, but this is not necessary, as the `add` operation allows a scalar variable to be broadcasted. The result of the `add` operation will have the same shape as `%a`. After the AST is traversed, all variables have a deduced type and shape. If any variable has an abstract type or a type conflict occurs where two types can not be unified, the algorithm aborts and an error is raised.

4.2.5 SSA Semantics for Voila

In contrast to MLIR IR, Voila has no restrictions regarding assignments. In Voila, variables can be assigned multiple times and can change their assigned values depending on control flow, e.g. `LOOP`-statements. This behavior is not allowed in an SSA semantic. Therefore, we have to transform such behavior to legal assignments. We perform this step during the generation of MLIR code. This allows us to associate the used variables in the AST to values already in SSA. To handle assignments in these situations, we only have to track the most current use of the Voila variable and its corresponding MLIR SSA value. If we visit an assignment node in the AST, we just replace the SSA value in the mapping with the new SSA value. For side effects we do not have to introduce extra transformations, as those side effects work on memory contents, while the references themselves are by definition SSA values. As we also have no control-flow that would make ϕ-nodes or similar constructs necessary, the result of this mapping is the Voila-MLIR dialect in SSA form. In Section 4.2.5, we will show an example lowering of Voila (cf. Listing 4.5) to Voila-MLIR (cf. Listing 4.6) in SSA form. During the translation of the Voila AST to Voila-MLIR operations,

```
1 ...
2 i = 1
3 i = add(i,i)
4 i = add(i,i)
5 ...
```

Listing 4.5: Voila Variable Reassignment

```
1 ...
2 %i1 = 1 : i64
3 %i2  = add(%i1, %i1)
4 %i3  = add(%i2, %i2)
5 ...
```

Listing 4.6: Variable Versioning for Reassignment in SSA form

for all variables in scope a hash table with the current SSA-value is stored. When a reassignment occurs in Voila, for example in Line 3, then the entry for this value is updated to point to the new SSA. In this way, we achieve a versioning of the variables and our mapping always points to the SSA that contains the newest version of the variable.

4.3 Lowering MLIR to LLVM

To lower the generated Voila-MLIR program, we use passes provided by MLIR to successively lower to lower level dialects until the `llvm` dialect is reached. In order

to allow for as many optimizations as possible, we lower to the highest level dialect allowed for the operations. Generally, all operations are lowered to loops, iterating over the input tensor operands of the operation and apply calculations element-wise.

4.3.1 Linalg Lowering

For operations that have either no loop dependencies or can be implemented as a reduction on the complete input, these loops are implemented in the `linalg` dialect as `linalg.generic` operations. For example, in Listings 4.7 and 4.8 we show the lowering of the `voila.add` operation to the `linalg` dialect. The input arguments are mapped to `linalg.generic` inputs (cf. Line 8) and corresponding to the operation type, the iterator type is set. As a simple element-wise addition of two vectors can be done in parallel, so the `parallel` iterator type is specified in Line 6. For other operations, such as aggregations which operate horizontally on vectors, we use the `reduction` iterator type. Additionally to the iterator type, further attributes can be added to the operation. Another mandatory attribute supplied are the indexing maps (cf. Line 6). Those maps describe an affine mapping of input values to output values, while the mapping is an identity mapping. Other possible mappings are reduction mappings, which map a dimension to a single value, and strided memory accesses, mapping input dimensions with additional stride specification to output dimensions. In contrast to Voila operations, the `linalg.generic` operation does

```
1 %0 = ... : tensor<100×i64>
2 %1 = ... : tensor<100×i64>
3 %2 = voila.add(%0, %1 :
4   tensor<100×i64>, tensor<100×i64>)
5    → tensor<100×i64>
```

Listing 4.7: Voila-MLIR Add Operation

```
1  %0 = ... : tensor<100×i64>
2  %1 = ... : tensor<100×i64>
3  %2 = linalg.init_tensor<100×i64>
4  #map = affine_map<(d0)→(d0)>
5  linalg.generic {indexing_maps = [#map],
6   iterator_types = ["parallel"]}
7  ins(%0, %1: tensor<100×i64>,
8   tensor<100×i64>)
9  outs (%2 : tensor<100×i64>)
10 {
11   ^bb0(%arg0: i64, %arg1: i64, %arg3: i64):
12   %3=std.add(%arg0,%arg1) : i64
13   linalg.yield %3: i64
14 }
```

Listing 4.8: Lowered Voila-MLIR Add

not directly return its results. Instead, it uses output operands (cf. Line 9) in which the result is stored. Those output operands are destructively built during the execution of the body. In our example, parallel computations never read from output parameters. Therefore, we do not need to supply initialized tensors to the `linalg.generic` operations, we just pass empty tensors that are filled by the loop. However, this changes when we want to do reductions. In those cases, the output parameter acts as accumulator and has to be initialized, e.g. to the neutral element of the reduction operation.

4.3.2 Affine Lowering

Other operations, such as hash table insertions with probing, can not be represented in the `linalg` dialect, as they carry dependencies within the loop body. To translate

them to MLIR in-tree dialects, we have to use a dialect which allows side effects. For our approach, we decided to use the highest-level dialect which allows the most operations while having a buffer semantic that allows for side effects. The next high-level dialect that fulfills these requirements is the `affine` dialect, which represents affine loop nests and modifies data using affine memory accesses. In Listings 4.9 and 4.10, we show the lowering of the insert operation to the affine dialect. In preparation to access the tensors in the affine loop, all input tensors are casted to `memrefs` (cf. Lines 3 and 5). Afterwards, in Line 9, the result buffer is allocated

```
1 %keys = ... : tensor<100×i64>
2 %values = ... : tensor<100×i64>
3 %ht = voila.insert(%keys, %values :
4    tensor<100×i64>, tensor<100×i64>)
5    → tensor<128×i64>
```

Listing 4.9: Voila-MLIR Insert Operation

```
1  %keys = ... : tensor<100×index>
2  %values = ... : tensor<100×i64>
3  %val_mem = to_memref %keys :
4    memref<100×index>
5  %key_mem = to_memref %values :
6    memref<100xi64>
7  %values_size = ... : index
8  %ht_size = constant.index(128):index
9  %res = memref.alloc(%ht_size) :
10  memref<128×i64>
11 affine.for %arg0 = 0 to %values_size
12 {
13  %val = affine.load %val_mem[%arg0] :
14   memref<100×i64>
15  %key = affine.load %key_mem[%arg0] :
16   memref<100×i64>
17  %insert_idx = (probing) ...
18  memref.store %val, %res[%insert_idx] :
19   memref<100×i64>
20 }
21  %ht = to_tensor(%res) :
22  tensor<128×i64>
```

Listing 4.10: Lowered Voila-MLIR Insert

and the affine loop iterates over all key-value pairs using the induction value `%arg0` and affine memory accesses in Line 15. Within each loop iteration, probing over the hash table is performed and one value inserted (cf. Line 18). Afterwards, when all values are inserted, in Line 21 the buffer is casted back to a tensor to make the lowering interoperable with other operations still using tensor semantics. We use this early lowering directly after the `voila` dialect to `linalg` lowering, to be able to exploit canonicalizations and CSE on a side-effect free SSA with tensors. Instead of bufferizing first, then lower voila to affine operations and having to deal with more complex analysis.

After all voila operations are translated to MLIR's internal dialects, we successively lower from one dialect to the next lower-level dialect, until we reach `llvm` code. In the first pass, we translate all tensors to memory references with corresponding allocations and deallocations. This pass is called bufferization and is automatically handled by MLIR. Afterwards, we lower all `linalg` operations to `affine` loops, followed by lowering to the `scf` dialect representing control-flow primitives such as

loops and if-statements. As the last step, we translate the `scf` dialect to the `llvm` dialect.

4.3.3 Runtime Wrappers

Besides the lowering towards LLVM, there are certain dialects that we use for optimizations and that are not directly translated to LLVM. Those dialects are the async and openmp dialects. Both dialects are lowered to function calls that act as wrapper around additional libraries. These wrappers are functions that are compiled to llvm bitcode, mostly C/C++ functions. With this technique, it is possible to use arbitrary code in the voila dialect through function calls. The only condition is that the functions call interface is compatible with the function call policy of LLVM, which is the case when the code was compiled with LLVM. We use these runtime wrappers for the async dialect, a small wrapper around LLVM coroutines to control dynamically generated thread groups and for the OpenMP runtime that is linked in the JIT executable after compilation.

4.4 Executing the Query

To make the generated MLIR code in `llvm` dialect executable, we have to convert it a last from MLIR IR to LLVM IR. We do this by passing the JIT execution engine an optimizing transformer that not only transforms the MLIR to LLVM, but also applies optimizations to the LLVM IR. Those optimizations are canonicalizations and peephole-optimizations that can be applied, but also the `mem2reg`-pass that detects and replaces memory allocations by register uses, if applicable. Alongside the translation, the execution engine also adds interface functions for every MLIR function to make them callable from host-code. This interface function takes a pointer to a byte array, which extracts the parameters to supply to the MLIR function and then calls the function with the correct parameters. When the transformation is finished, the code is JIT-compiled to target-dependent executable code. Afterwards, it is possible to call MLIR functions using the execution engine. To pass parameters to the newly created functions, those have to be formatted to obey the parameter format of the interface. The interface itself does only take a void pointer in which all pointers to the parameters have to be stored. Scalar parameters are passed as simple pointer to value, while arrays must be passed in a format corresponding to the format of `memrefs`. The data layout of a `memref` type passed as parameter to the interface function is a pointer to a struct containing a base pointer, a data pointer in the base pointer memory block, where the data are stored, followed by an offset from the data pointer where the data are stored. Furthermore, the struct also specifies a pointer to the memory sizes and the stride sizes. For our one-dimensional columns that are stored continuously, these size and stride arrays have a size of 1 with the size containing a pointer to the size of the column and the stride set to 1. As the columns are stored in an aligned memory, the base and data pointer are the same and the offset can be set to zero. To read the results returned from the query, the buffer matching the return types of the called function is generated on the fly and appended as the last argument to the interface function. This array has to be built on the fly, as the function return type is only known at runtime and currently there exists

no way to dynamically generate new data types on runtime in neither C++, nor LLVM. To extract the results from the dynamic buffer, the caller of the JIT-compiled function has to specify the desired result type and position in the result set. The corresponding buffer is then transformed to the correct data type and the result returned for further use.

5. Rewrite Rules

In this chapter, we present the rewrite passes that we use to lower from high-level MLIR to LLVM. Therefore, we will discuss how and which optimization passes of MLIR we use, as well as our adaptions to certain passes, in order to lower to better optimizable code in the end. The rewrite rules in this chapter are presented in the order of application. The application order is determined by the concepts of the dialect on which the transformations are applied to achieve as many optimizations from the passes as possible. We start out with inlining of functions to enable subsequent passes to apply transformations without being blocked by function boundaries. Afterwards, we fuse the tight loops characterizing single voila operations in the `linalg` dialect to reduce the number of intermediate results produced by the vector semantic. To optimize the resulting loops for better cache usage and future parallelization, we transform the fused loops to tiled loops. In the next step, we lower those tiled loops to affine loop nests, from which we vectorize the inner loop and afterwards parallelize the outer loop with parallelization passes.

5.1 Function Inlining

Directly after translation of Voila into MLIR, the query can contain multiple calls to other functions, to better structure the code. For example, selections or group-by operations could be separate functions written from Voila building blocks. While those functions allow for better code reuse and code structure, they also impose more complexity in the optimization. Intra-procedural optimization is more complex, as the context of all callers have to be considered. Especially in the case of our implementation of operations as tight loops, this leads to limitations of optimizations, because nearly all of our applied optimizations can be applied in a very restrictive manner. In Listing 5.1 we show an example of a program structured in multiple functions. While it is easy to comprehend, broken into multiple functions, many optimizations could only be applied on a per-function level. Thus, `between` would fuse to a loop generating the condition vector and `sel_and_sum` would be fused to a single loop, but both loops are not fused together because they are separated

```
1   func @between(%col, %lb, %ub) → tensor<100×i1>
2   {
3       %is_le = voila.leq(%col, %lb)
4       %is_ge = voila.geq(%col, %lb)
5       %pred = voila.and(%is_le, %is_ge)
6       voila.emit %pred
7   }
8
9   func @sel_and_sum(%col, %pred) → i64
10  {
11      %to_sum = voila.select(%col, %pred)
12      %sum = voila.aggr_sum(%to_sum)
13      voila.emit %sum
14  }
15
16  func @main(%col, %lb, %ub : ( tensor<100×i64>, i64, i64)) → i64
17  {
18          %pred = call @between(%col, %lb, %ub)
19          %res = call @sel_and_sum(%col, %pred)
20          voila.emit %res
21  }
```

Listing 5.1: Example Functions that block further optimizations such as loop fusion, and common sub-expression elimination because the scope of optimization is limited to intra-procedural optimizations

by different function regions and thus materializations on function boundaries are necessary. Those materializations are very expensive considering that every operation is a tight loop generating a materialized result and only through the fusion of those loops an efficient program without needless materializations can be generated.

To overcome these fusion boundaries, we replace function calls by their function body, a technique known as function inlining. We achieve this by using the built-in inliner pass of MLIR., which can automatically replace function calls with the body while replacing the function terminator with a value definition for the results and replace the function parameters by values. In Listings 5.2 and 5.3, we show a simplified example of inlining the `between` function into the `main` function.

In the `voila` dialect, `emit` operations are specified as region terminators, signaling the end of a function and its results returned to the caller. For further information on which operations can be safely inlined, we implement the `InlinerInterface` for the voila dialect. As all operations in Voila are safe for inlining, the applicability function always returns true. With this implementation, the MLIR inliner pass can automatically rewrite `voila` operations. In our example, the pass inlines the `between` function (cf. Line 1). This is done by replacing the function call with the function body. To connect the inserted body with the rest of the function, the return values of the terminator parameters are replaced with assignments and the terminator is erased. Furthermore, the input parameters are replaced with values of the calling function in Lines 4 to 9. This replacement also enables inference of shape information for the values used in the `between` function. In this example, the tensor size of the first dimension is set to 100, as a tensor of this size was passed to the function, before

```
func @between(%arg0: tensor<?×i64>,
  %arg1: 64, %arg2 : i64)
  → tensor<?×i64>
{
 %0 = voila.geq(%arg0, %arg1) :
   tensor<?×i1>
 %1 = voila.leq(%arg0, %arg2) :
  tensor<?×i1>
 %2 = voila.and(%0, %1) :
  tensor<?×i1>
 %3 = voila.select(%arg0, %2) :
  tensor<?×i64>
 voila.emit %3
}

func @main(%arg0 : tensor<100×i64>,
  %arg1 :i64 , %arg2 :i64) → i64
{
 %0 = call @between
   (%arg0, %arg1, %arg2) :
   tensor<?×i64>
 %1 = voila.aggr_sum(%0) : i64
 voila.emit %1
}
```

Listing 5.2: Range Query in Which the Between Function Will be Inlined

```
func @main(%arg0 : tensor<100×i64>,
  %arg1 :i64 , %arg2 :i64) → i64
{
 %0 = voila.geq(%arg0, %arg1) :
  tensor<100×i1>
 %1 = voila.leq(%arg0, %arg2) :
  tensor<100×i1>
 %2 = voila.and(%0, %1) : tensor<100×i1>
 %3 = voila.select(%arg0, %2) :
  tensor<?×i64>
 %4 = voila.aggr_sum(%3) : tensor<?×i64>
 voila.emit %4
}
```

Listing 5.3: Resulting Fused Linalg Loops

it was inlined. In the last step, the now unused function `between` is deleted to safe compilation time later on.

We usually apply this pass as the first optimization pass in the `voila` dialect, since this gives the highest potential of optimization later when lowering to other dialects and applying optimizations such as loop fusion.

5.2 Operation Predication for Lazy Materialization

In contrast to the original proposal of Voila, our implementation does not support predication of operations. The reason for this is the nature of predication in a vector programming language for query execution. Predications are similar to actual selections, just without doing a full computation of the selection results with compression. The predicates can be inferred from selections and another reason against explicit predication is that most operations on the high-level vectors lower to low-level vector instructions with the same latency of predicated and unpredicated instructions. This makes predication by the user more costly in terms of parsing, restricting the freedom of the framework towards adapting the query and also making the language more complex for the user by adding a hinting that has only positive effects in certain non-trivial situations. Those situations can be at least equally good handled by an adaptive framework that recognizes those situations and optimizes accordingly. Moving towards more heterogeneous processing, predication could also lead to worse performance, for example for rather complex operations that could

not be lowered to predicated hardware vector instructions but instead introduce a branch that could be mispredicted or lead to branch divergence in the case of GPUs. Therefore, we choose to not expose predication as a language element, but instead implement a separate pass which predicates operations implicitly and only in cases where it allows for lazy result materialization i.e. on aggregations, or even the elision of entire selections. To achieve this goal, we identify all operations where results of a selection have to be materialized and introduce a `PredicationInterface` which handles predication of those operations. These operations are aggregations, `gather` and `scatter`, as well as the hash table operations insert and lookup. For these operations, we implement an additional lowering with predication. The predicate is an additional tensor of bits and their values represent the condition predicate in the same way as in the select operation. This allows reuse of the selection predicate during predication. The predication itself is implemented as a branch that executes the operations conditionally. This allows to meld an operation with a selection and since the operations that are predicated all have an implementation with multiple instructions that are not easily vectorizable and thus hardware predicatable, their execution latency benefits from predication.

In the following, we describe the predication pass using the example transformation of a sum aggregation of a selection in Listing 5.4 to a predicated version in Listing 5.5.

```
1  func @main(%arg0: tensor<100×i64>,
2   %arg1:i64) → i64
3  {
4    %0 = voila.geq(%arg0, %arg1) :
5     tensor<100×i1>
6    %1 = voila.select(%arg0, %0) :
7     tensor<?×i64>
8    %2 = voila.mul(%1, %1) :
9     tensor<?×i64>
10   %3 = voila.aggr_sum(%2) : i64
11   voila.emit %3
12 }
```

Listing 5.4: Sum Aggregation of a Selection

```
1  func @main(%arg0: tensor<100×i64>,
2  %arg1:i64) → i64
3  {
4      %0 = voila.geq(%arg0, %arg1) :
5      tensor<100×i1>
6      %2= voila.mul(%arg0, %arg0) :
7      tensor<100×i64>
8      %3 = voila.aggr_sum(%2, %0) : i64
9      voila.emit %3
10 }
```

Listing 5.5: Predicated Sum Aggregation with Elided Selection

The pass consists of two steps, the analysis step in which it is analyzed whether the selection in Line 6 can be replaced by predication and the replacement step based upon the analysis result.

Selection Matching

To determine, if a selection is eligible for being replaced with predication, we transitively traverse the DAG of all uses of the selection result. During the traversal, we check for direct or indirect uses of the selection result in an emit statement, our predicatable operations, or other operations. If the selection result is used in an emit statement, the result has to be materialized in the selection as emit statements indicate a form of result from a function call over whose boundaries we cannot forward predicates. Furthermore, this pass runs after the inlining pass descibed in

Section 5.1 meaning, that any emit statement present in the code signals a result return outside of the JIT environment. In case of non-predicatable operations such as the multiplication in Line 8 we keep traversing their uses recursively, as these operations typically represent hardware vectorized instructions that have the same latency with and without predication. For predicatable operations, such as `voila.aggr_sum` in Line 10, we store them as possible predication candidates in the later replacement phase. As the results of predicatable operations already contain the selection semantic, we cut off traversal at these instructions and continue the traversal on another use. We use a depth-first traversal in our implementation, but the traversal order does not matter, as long as it is ordered, since we are only searching for emit statements and predicatable operations. When the traversal finishes without finding an emit operation, the rewrite phase takes place.

Selection Rewrite

In this phase, all stored operations are predicated with the selection predicate, and their use of the selection result is replaced with the unpredicated input tensor of the selection. In our example, only the aggregation operation in Line 10 has to be predicated by replacing the selection result `%1` with the input argument `%arg0`. Furthermore, the selection predicate `%0` is attached as predicate to the aggregation. (cf. Line8) . When all candidates are predicated, the select operation is checked for any uses left. If there are none left, the operation is erased, which is also the case in our example. Otherwise, the predication pass was able to delay the selection to operations where the selection results are needed, but failed to completely elide the selection.

5.3 Operator Fusion

With the inlining and linalg lowering (cf. Section 4.3), the code contains mostly tight loops operating on a single vector at-a-time and often producing large intermediate results. These results are materialized by each operation and passed to the next operation, just to be eventually modified and materialized again for the next operation. This is very ineffective, as this is an explicit producer-consumer relationship and thus, the results produced by one operation are used by the next to generate the next result and not reuse the input. To remove those unnecessary intermediate results in producer-consumer operation chains, we utilize the linalg-loop fusion pass built-in into MLIR to fuse those tight `linalg.generic` loops within a single loop with multiple element-wise operations in the body. This helps to effectively remove intermediate vectors and instead pass a single value through all operations. Aside of unneeded intermediate results, there also exist calculations without data-flow dependencies on other operations. Those operations can also be fused with parallel running computations to increase the locality of computations. This fusion is called sibling fusion[SS03]. In Listings 5.6 and 5.7, we show an example of sibling and producer-consumer fusion. The two comparisons in Line 3 and 12 are independent of each other and are fused as a sibling fusion. The conjunction of the two comparison results in Line 21 is a producer-consumer relationship and can also be fused, as the operation is applicable element-wise. The fusion is done gradually, fusing two neighboring loops at a time until no more loops can be fused.

```
//variable initialization
...
linalg.generic
ins(%0, %1;:
 tensor<100×i64>, tensor<100×i64>)
outs (%2 : tensor<100×i1>)
{
 ^bb0(%arg0: i64, %arg1: i64, %arg2: i64):
  %3=arith.cmpi "sle", %arg0, %arg1 :i64
  linalg.yield %3: i1
}
linalg.generic
ins(%1, %4:
 tensor<100×i64>, tensor<100×i64>)
outs (%5 : tensor<100×i1>)
{
 ^bb0(%arg0: i64, %arg1: i64, %arg2: i64):
  %6=arith.cmpi "sgt", %arg0, %arg1 :i64
  linalg.yield %4: i1
}
linalg.generic
ins(%2, %5:
 tensor<100×i1>, tensor<100×i1>)
outs (%7 : tensor<100×i1>)
{
^bb0(%arg0: i1, %arg1: i1, %arg2: i1):
  %8=arith.andi %arg0,%arg1: i1
  linalg.yield %5: i1
}
```

Listing 5.6: Linalg Loops for Conjunction of Comparison Result Predicates Before Fusion

```
//variable initialization
...
linalg.generic
ins(%0, %1, %2:
 tensor<100×i64>, tensor<100×i64>
 ,tensor<100×i64>)
outs (%3 : tensor<100×i1>)
{
 ^bb0(%arg0: i64, %arg1: i64, %arg2: i64,
  %arg3: i64):
  %4=arith.cmpi "sle", %arg0, %arg1 :i64
  %5=arith.cmpi "sge", %arg1, %arg2 :i64
  %6=arith.andi %3,%4 : i1
  linalg.yield %6: i1
}
```

Listing 5.7: Resulting Fused Linalg Loops

In the first step, the two comparison loops are matched to the loop-fusion rule. As there are no operations with potential side effects on values participating in the loop operations between the loops, the rewrite as fused loops does not change the semantics of the program. To determine how to fuse the loops, a potential producer-consumer relationship has to be determined. In the example, this is not the case, as the output tensor of the first comparison is not used as input parameter of the second loop. Therefore, we apply sibling fusion, by merging the input and output arguments of both loops and merge both loop bodies. This step is not depicted in our example, as there is still another loop that is fusible with the newly created fused loop. The next fusion is a producer-consumer fusion. The two output parameters of the first loop are used as input parameters in the second loop. To fuse both loops, the input tensors of the first loop become input tensors of the new, fused loop and the output parameters of the fused loop are the output parameters of the second loop. When fusing the two loop bodies, operations of the second loop body which have parameters with a producer-consumer relationship to operation results in first loop, directly use these results, effectively eliminating any intermediate materializations.

Naturally, this approach is not applicable to all operations. There are operations, that need the complete relation in order to do a calculation. This is the case for hash table lookup of one column in the hash table build from another column. Those two operations have no producer-consumer relationship in the loop-body and also have no sibling relationship, making them non-fusible. Those operations are known as pipeline blockers[Dur+19] and act as synchronization points in our query. As those operations also have to be implemented in another dialect allowing side effects, those operations are not fused with the element-wise loop, instead the materialized intermediate result of the fused loop is passed to those operations. This reduces the number of materializations to only a one per pipeline until the next pipeline breaker, but is still more than needed. To eliminate the left materialization before the pipeline breaker, we rely on later loop-fusion in the affine dialect. To fuse affine loops, the analysis is somewhat more involved, as producer-consumer analysis is more complex due to memory use analysis. Furthermore, affine-loop fusion is done on a best-effort basis and not guaranteed to be applicable to any pipeline. However, most of the time this allows to remove all unneeded materializations of intermediate results. In Listings 5.8 and 5.9, we exemplary show the fusion of affine loops, fusing a smaller-equals comparison lowered to the `affine` dialect with a selection.

```
//variable initialization
...
affine.for %i = 0 to %N
{
  %3 = affine.load %0[%i] : i64
  %4 = arith.cmpi "sle", %3, %cst : i64
  affine.store %1[%i], %4
}
affine.for %i = 0 to %N {%cur_idx = %c0}
{
  %cond = affine.load %1[%i]
  %next_idx = scf.if (%cond)
  {
    %5 = affine.load %0[%i] : i64
    memref.store %2[%cur_idx], %5
    %inc = arith.add %cur_idx, %c1
    scf.yield %inc
  }
  {
    scf.yield %cur_idx
  }
  affine.yield %next_idx
}
```

Listing 5.8: Affine Loops for Smaller-Than Comparison and Selection Before Fusion

```
//variable initialization
...
affine.for %i = 0 to %N {%cur_idx = %c0}
{
  %3 = affine.load %0[%i] : i64
   %cond = arith.cmpi "sle", %3, %cst
  %next_idx = scf.if (%cond)
  {
     memref.store %2[%cur_idx], %3
     %inc = arith.add %cur_idx, %c1
     scf.yield %inc
   }
   {
     scf.yield %cur_idx
   }
  affine.yield %next_idx
}
```

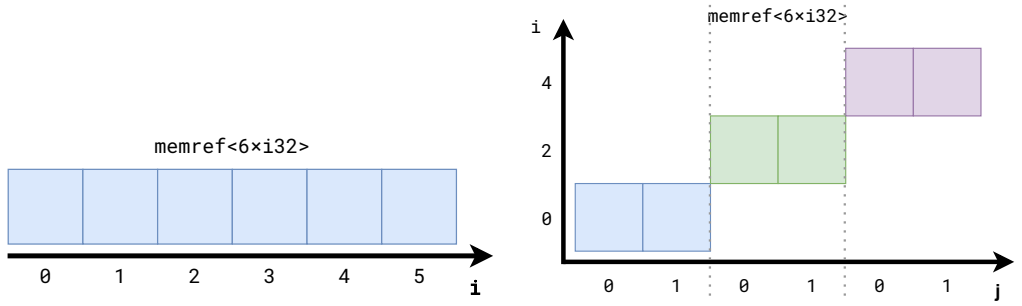
Listing 5.9: Resulting Fused Affine Loops

The first affine loop compares all values of `%0` with the smaller-than relation and stores the results in the selection predicate vector `%1`. The next loop selects values depending on those predicate and stores them compressed in a new vector. To indicate the position on which the next value is stored, an iterator argument is

used to carry the current insert position in the loop. As the second loop uses the first loop's results, there is a producer-consumer relationship between both loops. Furthermore, both loops have the same loop bounds and no further side effects on the consumed values. This means, all preconditions are fulfilled to fuse both loops. For the affine fusion, no input and output arguments are used, which simplifies merging as the loop head can just be replicated. On the other hand, producer-consumer relationships are more difficult to handle because they are implicit through memory accesses. To correctly fuse the two loop bodies, it is required to replace all loads of the produced values by the result value before it is stored and thus replace the producer-consumer relationship between entire results of the loops by a relationship of scalar values in the loop body.

5.4 Loop Tiling

Usually, loop tiling is used to optimize cache residency of data used in loop-nests[HS17]. This works through chunking of the loop range and thus operating on a block of the arrays on which the loop operates. When choosing an appropriate size matching the available cache sizes on different loop nests, this leads to an optimal caching behavior with a minimal amount of cache misses. While it is also relevant to optimize



(a) Schematic Description of Iteration over an Array in a Single Loop Without Tiling

(b) Schematic Description of Iteration over an Array with Tiling in two Nested Loops

Figure 5.1: Schematic Array Processing with and Without Tiling

cache efficiency for our use case, we primarily use loop tiling as preparation for parallelization. We tile loops to achieve a new outer loop matching the number of available threads in iterations. This allows subsequent passes to simply transform the outer loop for parallelization and the inner loop for vectorization. In the schematics in Fig. 5.1, the untiled loop (cf. Fig. 5.1a) would be blocked into a 2-D nested loop (cf. Fig. 5.1b), where `i` becomes the outer, parallel loop and `j` the inner, vectorized loop.

Most of the time, the number of loop iterations is not evenly divisible by the desired tile size. To still be able to tile those loops, the loop is split into two parts, the evenly divisible and the remainder. The evenly divisible part is then tiled and afterwards, the second loop is executed. This technique is called loop peeling, and we use this transformation to achieve a higher resource utilization in the simplified loops. In MLIR, there are two dialects in which loop tiling can occur, `linalg` and `affine`.

MLIRs affine loop tiling pass splits affine loops, as described above. In contrast to this, the built-in linalg tiling pass of MLIR has several options. Linalg tiling can produce special `linalg.tiled_loop` operations representing a tiled loop nest, but the pass can also generate affine, parallel or `scf` loops. The main advantage of the `linalg.tiled_loop` operation is that the tiling transformation is made explicit and can be used later on to allow better optimization. For example, `linalg` tiled loops have an attribute to specify how parallelization of the tiling can be done, regarding the mapping of iterations to execution units. In order to best use the capabilities of all dialects, we use the `linalg` tiling pass to generate tiled loops exclusively. As for the loop fusion, we use both to enable tiling for all operations. In the following, we will show example transformations for both, `linalg` tiling (cf. Listings 5.10 and 5.11) and `affine` tiling (cf. Listings 5.12 and 5.13).

5.4.1 Linalg Tiling

```
1  //variable initialization
2  ...
3  linalg.generic
4   ins(%0: tensor<100xi64>)
5   outs (%1 : tensor<i64>)
6   iterators("reduction")
7  {
8   ^bb0(%arg0: i64, %arg1: i64):
9     %2=arith.addi %arg0, %arg1 :i64
10    linalg.yield %2: i64
11 }
```

Listing 5.10: Linalg Generic Loop for Sum Aggregation Before Tiling

```
1  //variable initialization
2  ...
3  linalg.tiled_loop (%i) = (%c0)
4   to (%c96) step (%c8)
5  ins(%0: tensor<100xi64>)
6  outs(%out : tensor<i64>)
7  iterators("reduction") {
8   %sub = tensor.extract_slice
9    %lhs[%i] [%c8] [1]
10   : tensor<100xi64> to tensor<8xi64>
11
12   %sum = linalg.generic
13    ins(%sub: tensor<8xi64>)
14    outs (%1 : tensor<i64>)
15    iterators("reduction")
16   {
17    ^bb0(%arg0: i64, %arg1: i64):
18     %2=arith.addi %arg0, %arg1 :i64
19     linalg.yield %2: i64
20   }
21   linalg.yield %sum : tensor<i64>
22 }
23
24 //peeled loop
25 linalg.tiled_loop (%i) = (%c96)
26  to (%c100) step (%c8)
27  ins(%0: tensor<100xi64>)
28  outs(%out : tensor<i64>)
29  iterators("reduction") {
30    ...
31 }
```

Listing 5.11: Resulting Linalg Tiled Loop with Generic After Tiling and Peeling With Tile Size of Eight

To introduce tiling on tensors using `linalg.tiled_loops`, the respective loop is wrapped into the tiled loop in Lines 3-22, which handles tiling and merging of the

tensors applied to the loop. In each iteration of the tiled loop, the original loop is called with a tile of its original input parameters. In each iteration of the tiled loop, a tile of the input parameters is extracted, starting from the current offset of the loop, as we show in Line 8. After the loop is executed, the computation results are merged back into the tiled loop's output operands. Depending on the iterator type, this can look differently. For this example, the loop computes a reduction, therefore leading only a single result per iteration in Line 21. The same counts for a block of iterations, and thus the result of the loop can just be yielded to the next tile. However, this is different for parallel iterators, which are just element-wise mappings of inputs to outputs, which means that the results have to be merged together with the results from previous iterations. This is done by inserting the result slices to the output parameters of the tiled loops on the specified offset used for this tile, just the reverse of the slice extract operation done before the loop. As in our example, the size of the input tensors and thus loop iterations is not evenly divisible by 8, we peel off the last iterations of the loop into a second loop. This enables the first loop to have a fixed tile size of 8 and thus enables better reasoning on the loop body and better optimization. The second loop, in our example from Line 25 to 31, just executes the remaining iteration in the same fashion as the other tiled loop, but with reduced tile size.

5.4.2 Affine Tiling

For affine loop tiling, no special loop type is available. Instead, the built-in affine loop tiling pass of MLIR, builds an affine loop nest around the loop (cf. Listing 5.13), the outer loop contains the loop tiles and iterates in steps of the tile size. Then the inner loop bounds are adapted to iterate the tile-size number of iterations, starting at the offset of the outer loop's induction variable. In contrast to `linalg`-loop tiling, tiling on affine loops does not slice the input data, as all modifications are made implicitly through memory references. To peel affine loops, MLIR has an extra loop peeling pass.

The tiling semantic of the linalg dialect works well for parallel iterators, and those linalg tiled loops can simply be parallelized by executing multiple tiled blocks in parallel. This is possible because each iteration is independent of the others. This is also the case for the tiling and merge phases, as they work on distinct areas. However, this is no longer true for reduction iterators, which we frequently use for several operations. In those loops, we can still tile the input operands independently, but the output operand is the same and access to it has to be serialized, which blocks the parallelization of reductions. As of now, there is no implemented solution for this, but there are discussions[12] to split the `tiled_loop` operation into regions. The first region would execute operations in parallel on each tile and the second region specifies the merging of those independently calculated tiles. This would allow generating parallel tree-merges and further speed-up the generated code. For now, we do not adapt `tiled_loop` operations to this newly proposed semantic and instead rely on a canonicalization applied during lowering to the affine loops for better parallelization. However, affine tiling requires a more complex loop analysis to tile reductions, as it

[12] https://llvm.discourse.group/t/rfc-changes-to-linalg-tiledloopop-to-unblock-redutions/3890

```
//variable initialization
...
// implicitly steps by 1
affine.for %i = 0 to %N
{
  %3 = affine.load %0[%i] : i64
  %4 = arith.cmpi "sle", %3, %cst : i64
  affine.store %1[%i], %4
}
```

Listing 5.12: Affine Loop of a Smaller-Than Comparison Before Tiling

```
//variable initialization
...
affine.for %i = 0 to %N
step 10
 iter_args(%cur_idx = %c0)
{
//simplification, normally
//an affine map application
//would be used to statically
//determine the upper bound
  %next_idx = affine.for %i = %i
  to %i+10
  iter_args(%cur_idx2 = %cur_idx)
  {
    %3 = affine.load %0[%i] : i64
    %cond = arith.cmpi "sle", %3, %cst
    %next_idx = scf.if (%cond)
    {
      memref.store %2[%cur_idx2], %3
      %inc = arith.add %cur_idx2, %c1
      scf.yield %inc
    }
    {
      scf.yield %cur_idx2
    }
    affine.yield %next_idx
  }
  affine.yield %next_idx
}
```

Listing 5.13: Resulting Tiled and Peeled Affine Loops

operates on memory references and thus the transformation can only be applied if there are no side effects in the loops. Therefore, we use linalg tiling where possible to reduce the run time costs caused by affine tiling.

5.5 Linalg Lowering to Affine Dialect

To lower `linalg` operations such as `linalg.generic` and `linalg.tiled_loop`, we convert them into loops of the `affine` dialect. These loops represent affine loop nests in combination with memory accesses, instead of tensors. Lowering to the `affine` dialect allows for an efficient vectorization and parallelization in subsequent passes. MLIR already comes with a pass to lower `linalg` operations to affine loops, but at the time of our writing, this feature is not fully functional and only able to lower linalg generic operations. The resulting affine loops are semantically correct, but for reduction loops, the lowering produces loops not optimally translated to the affine dialect, as they do not use the induction variables for the reductions. Reductions in the affine dialect are done via additional loop induction variables in the `iter_args` argument. Later parallelization and vectorization transformations rely on this representation. In order to generate better affine loops, we adapt the

rewrite pattern for `linalg` generics and add a lowering pass for `linalg` tiled loops. Together with the vectorization pass, we describe in Section 5.6, we are able to work around the reduction blocking issue of `linalg` tiled loops. In Listings 5.14 and 5.15, we exemplarily explain the lowering of a `linalg` generic using the reduction iterator to an affine loop with induction variable.

```
1  //variable initialization
2  ...
3  linalg.generic
4  ins(%0: memref<100×i64>)
5  outs (%1 : memref<i64>)
6  iterator("reduction")
7  {
8   ^bb0(%arg0: i64, %arg1: i64):
9    %2=arith.addi %arg0, %arg1 :i64
10   linalg.yield %2: i64
11 }
```

Listing 5.14: Example of Linalg Reduction

```
1  //variable initialization
2  ...
3  %init = memref.load %1[] : i64
4  // implicitly steps by 1
5  %2 = affine.for %i = 0 to %N
6    iter_args(%acc = %init)
7  {
8    %3= affine.load %0[%i] : i64
9    %4 = arith.addi  %1, %acc : i64
10   affine.yield %4 : i64
11 }
12 memref.store %4, %1[] : memref<i64>
```

Listing 5.15: Reduction Lowered to Affine Dialect Using Iter Args

In our example, we use a buffered version of the tiled loop example from the last section with a tile size of 10 to omit the peeled loop for clarity. To transform the example into affine loops, the loop bounds have to be inferred through the parameter sizes and their mappings. For our example, the input `memref` in Line 4 has a single dimension with 100 elements. As the outer loop (not shown in the example) operates on complete tiles, the loop bounds for the affine loop are a number of elements in the input operands with a step size equivalent to the tile size. For the inner loop operating on the tiles, the bounds are the tile size. As the iterator type for the `linalg` operations is a reduction, we introduce an iterator variable with the type of the output parameter on which the reduction in the affine loops takes place. This value is initialized to the initial value of the output parameter from the `linalg` operations loaded in Line 3, to achieve the same behavior of the operation when the output value is used. Additionally, in Line 12 a store of the iterator value is issued after the loop operation to store the loop results back to the output for later use. As this leads to redundant loads and stores in the tiled loop, those operations can be folded away and the loop result can be used directly. To fill the loop bodies, for all input operands, an affine load with a mapping to the `linalg` output variable is created. For output iterators, the same is done with affine stores. Afterwards, all operations are copied to the body and the uses of `linalg` block arguments are replaced with the mappings.

5.6 Affine Vectorization

The goal of the vectorization passes, which we utilize in our framework, is to optimize our code for the use of the hardware's SIMD processing capabilities. In the built-in affine vectorization pass of MLIR, affine loops are vectorized through a technique called super-vectorization. In contrast to classical vectorization, where loops must be

proved to be vectorizable, this pass can rely on the affine properties of the loops and memory operations. There also exists a `linalg` vectorization pass, which rewrites entire linalg loops into a sequence of vector operations. The vector size then has to be controlled through tiling. However, we do not use this pass in our pipeline, as it would prematurely lower entire structures into the vector dialect, skipping the affine dialect and thus also skipping several operations we natively lower to affine that could be potentially vectorized. For this reason, we decide to use the super-vectorization pass of the affine dialect, which also keeps intact the affine loop structure and thus allows further optimizations such as loop fusion.

The process of super-vectorization in the affine dialect is rather straightforward: scalar affine memory operations in the loop are replaced with vector load operations. Additionally, the loop step size is adjusted to the vector size and all operations of the loop body also have to support vector-type operands. This is verified through a check of all operations implementing the `Vectorizable` trait. If this is the case, the loop can be vectorized through super-vectorization. If any operation in the loop body is not vectorizable, the loop is ignored for vectorization, which makes the vectorization a best-effort application. In the following, we explain the rewrites applied by super-vectorization on the example vectorization in Listings 5.16 and 5.17 with a vector size of 8.

```
1  //variable initialization
2  ...
3  %init = memref.load %1[] : i64
4  // implicitly steps by 1
5  %2 = affine.for %i = 0 to %N
6   iter_args(%acc = %init) → (i64)
7  {
8    %3= affine.load %0[%i] : i64
9    %4 = arith.addi  %1, %acc : i64
10   affine.yield %4 : i64
11 }
12
13 memref.store %4, %1[] : memref<i64>
```

Listing 5.16: Example of Affine Reduction Loop

```
1  //variable initialization
2  ...
3  %2 = affine.for %i = 0 to %N
4   step 8 iter_args(%acc = %c0)
5    → (vector<8xi64>)
6  {
7    %3= vector.transfer_read %0[%i], %c0 :
8      vector<8xi64>
9    %4 = arith.addi  %1, %acc :
10     vector<8xi64>
11    affine.yield %4 : vector<8xi64>
12 }
13
14 %init = memref.load %1[] : i64
15 %5 = vector.reduction "add" %2, %init :
16    vector<8xi64> into i64
17 memref.store %5, %1[] : memref<i64>
```

Listing 5.17: Affine Reduction Loop Vectorized Through Super-Vectorization

In this example, the affine loop calculates the sum of all values in `%4` using an iteration argument that is updated in each iteration (cf. Section 5.6). The super-vectorization pass is able to detect such reductions and vectorize them. However, as of now this is only possible for simple reductions. A reduction is simple, if the reduction variable is carried as a loop `iter_arg`, the reduction result is not used for a side effect and the reduction operation is an arithmetic or logical operation (e.g. `add`, `mul`, `and`, `or`, etc.). In addition to the restriction to simple reductions, currently only 1-D vectors and affine loops with unit step size are supported[13]. In order to rewrite the affine

[13] https://reviews.llvm.org/D100694

loop into a vectorized form, the loop step size is adjusted to match the vector size and to have no overlapping computations in the loop body (cf. Section 5.6). The next step is detection and rewriting of simple reductions. This is done by replacing the `iter_arg` `%acc` with a vector type, whose values are initialized with the neutral element of the reduction operation – the same would be done for all `iter_args`. This leads to a vectorized, independent horizontal reduction of the input `memref` to the size of the used vector, but still lacks the last reduction step from a vector to a single scalar value, as the affine loop result is now a vector and not a scalar. The horizontal vector reduction is added below the loop body. Therefore, in Section 5.6, the loop result vector is horizontally reduced with the loop's reduction operation using the initial value of the `iter_arg` as accumulator. As all supported reduction operations are commutative, it is guaranteed that the vectorized reduction result is equivalent to the previous loop. At last, all scalar load and store operations in the loop are replaced with vectorized counterparts (cf. Section 5.6). Currently, simple `vector` load/store operations are used, instead of the `affine` vector-load/store operations that support better memory liveness analysis. However, these operations, in contrast to the affine operations, have the capability to allow masking and permutation of vector elements, which could be needed for irregular access patterns in the general case. If the loop upper bound is not evenly divisible by the vector size, the vectorized load and store instructions use masks to mask out garbage out-of-bounds calculations. For the sake of simplicity, we omit this part in the example and instead assume that the upper bound is evenly divisible by 8. Consequently, all usages of these values are now vectorized, which is only one operation in the example, the addition in Section 5.6.

Besides plain vectorization of single loops, the affine super-vectorization pass is able to vectorize entire loop-nests. For those nests, there are different strategies of loop vectorization, like innermost, outermost or intermediate loop vectorization: each has certain benefits depending on the use case[NZ08]. To decide which loop to vectorize, the pass has the built-in capability to test different techniques to find the most promising approach. Therefore, the pass takes into account the number of operations that can be vectorized, as well as the effect it takes on the number of loops and what gives the best loop layout for fusion.

5.7 Parallelization

5.7.1 Affine Parallelization

The last major optimization is the parallelization pass. We use this pass to apply loop parallelization on the outer loops that we produced during tiling. We run this pass rather late in the optimization pipeline, to ensure all previous optimizations that work on higher dialects, or are independent of MIMD parallelism could be applied. To parallelize our code, we transform serial `affine.for` loops to `affine.parallel` loops. The goal of this technique is to enable our optimized and vectorized code to take advantage of MIMD capabilities of the executing hardware, for example multithreading. This transformation is straight forward for affine loops without induction variables, as affine for operations operate on a single affine range that can

be directly mapped to the representation of `affine.parallel` loops that operate on affine bands. Those bands can be multidimensional, meaning that a single parallel loop can represent multiple parallel loop nests. Nonetheless, in our approach we only have to deal with non-nested loops resulting in one-dimensional bands. These simple transformations however, are only applicable, if the loop does not contain side effects carried out over a single loop iteration, as in these cases, parallel execution of the loop body would generally lead to race conditions. For this reason, loop parallelization transformations are restricted to simple loops and loops with side effects representable as reductions over the loop's `iter_args`. Additionally, only a subset of reduction functions in the form of single, scalar arithmetic or logic instructions that can be executed atomically is allowed. In principle, any reduction function could be used for the parallel reduction, but more complex functions require synchronization during the reduction step in order to prevent race conditions. This would make transformation more complex, involving additional dialects for loop synchronization, while having no influence on the parallelizability of our code, as reductions available in our Voila-MLIR dialect are already restricted to simple aggregate functions in form of single instructions in the reduction step.

In the following, we exemplary show the transformation of an affine for loop (cf. Listing 5.18) with a reduction on an induction variable, as generated by the previous optimization passes to a parallel loop (cf. Listing 5.19). As parallelization is the last major pass of our optimization pipeline, we assume that reductions are generally formulated idiomatically using `iter_args`, as also generated by our affine lowering and vectorization passes. In the first step, of MLIR's built-in affine parallelization pass, side effects of the loop body are examined. The analysis is rather simple and conservative, as it considers all affine loops as not parallelizable that contain store or load instructions that are not using affine induction values as indices, as well as affine load/stores that use other affine maps than identity or strided map layout. Furthermore, the analysis fails, if the loop body contains non-affine regions. As this is only the case for hash insert and selection operations, which we implemented in an inherently serial fashion for simplicity, these restrictions do not prevent the transformation of any relevant operation. In our example, the inner vectorized reduction loop in line 5,which we omitted, uses affine vector load operations to load the tile's data from memory and aggregates them, similarly to our example of super-vectorization in Listing 5.17. Since the vector load is the only side effect and uses an affine induction variable as index, there are no side effects that could lead to data races.

In the next step, potential induction variables and their usages have to be checked for side effects, as those could also easily lead to data races, for example in the case that multiple threads write to the `iter_arg` at the same time. In this case, the `iter_arg` could end up containing an incorrect value. To rule out the possibility of such effects, induction variables are only allowed to be used by a single side-effect free operation followed by the use in the terminator, which also applies in our example. Hence, the loop is parallelizable and can be transformed into a parallel loop. This is done by constructing the loop head, copying the loop lower and upper bounds, as well as its step size. In the next step, the reduction type of the induction variable is determined. Therefore, the algorithm inspects the last use of `iter_arg` and

```
%2 = affine.for %i = 0 to %N step 64
 iter_args(%res = 0)
{
// vectorized inner reduction loop tile
  %vec_red = ...
  %5 = vector.reduction "add" %vec_red :
   vector<8xi64> into i64

  %sum = arith.add %5, %res : i64
  affine.yield %res
}
```

Listing 5.18: Example of Vectorized, Tiled Affine Reduction Generated From Previous Transformations

```
%2 = affine.parallel (%i)
 from (0) to (%N)
 step (8) reduce "addi" → i64
{
//vectorized inner reduction loop tile
  %vec_red = ...

  %5 = vector.reduction "addi" %2 :
   vector<8xi64> into i64
  affine.yield %5
}
```

Listing 5.19: Parallelized Affine Loop With Reduction Through Summation

which atomic reduction operation it matches. The supported reduction kinds are all types of `atomic_rmw` operations. In our example, the only use of the `iter_arg` `%res` is an addition in Line 9, which can be made atomic through the use of the atomic `fetch_and_add` instruction. So, access to `%res` in the loop could be proved to be side-effect free. Subsequently, the add operation can be removed and the loop induction variable annotated with the reduction type `addi` (cf. Line 3). As the last step, the serial loop body is copied to the parallel loop, mapping the old loop variables and removing the explicit induction variable. This results in the parallelized loop we show in Listing 5.19.

5.7.2 Parallel Loops To Async Lowering

In contrast to lowering to `openmp`, the `async` dialect does not have parallel loop constructs. The async dialect is a dialect generalizing the concept of parallel code block execution and is more of a wrapper for parallelizing entire functionality opposed to parallelization of control flow such as openmp. Therefore, lowering parallel loops towards using the async dialect for parallelism is more involved than just wrapping the parallel code in a parallel structure. In the following, we will describe the process of lowering parallel loops into the async dialect. For a better overview of the process, we use the parallel reduction loop example previously used to show parallelization of reductions (cf. Listing 5.19).

As there is a preexisting pass implementing parallelization of loops in the structured control flow dialect, we adapted and extended this pass to also allow async reductions, instead of writing an entirely new pass lowering from the affine dialect. Consequentially, our example in Listing 5.21 is not written in the affine domain, but the `scf`-domain. In `scf`, loop reductions are expressed as separate blocks in the loop bodies. Hence, the example has a different syntax, but same semantic as the affine example in Listing 5.19.

```
1  func @main(%input : memref<i64x128>)
2   → i64 {
3   %res = scf.parallel (%iv) = (0) to
4    (128) step (64) init (%neutral)
5    → i64 {
6    //load and reduce vectorized locally
7    %to_reduce = ...
8     scf.reduce(%to_reduce) : i64 {
9     ^bb0(%lhs : i64, %rhs: i64):
10     %res = arith.addf %lhs, %rhs : i64
11     scf.reduce.return %res : i64
12    }
13   }
14   return %res
15  }
```

Listing 5.20: Example of Vectorized, Tiled Affine Reduction Generated From Previous Transformations

```
1  func @async_reduce
2   (%Input, %start, %stop, %res :
3  (memref<128×i64>, index,index,memref<i64>))
4  {
5  //load and reduce vectorized locally range
6  %to_reduce = ...
7  atomic_rmw "addi" %to_reduce, %res :
8   (i64, memref<i64>) → i64
9  }
10 func @dispatch
11 (%group, %Input, %start, %stop, %res)
12 {
13 //split the range
14 //and call @dispatch recursively
15 //add all threads to the %group
16 ...
17 call async_reduce
18 (%Input, %start, %stop, %res)
19 }
20 func @main
21 (%input : memref<i64x128>) → i64{
22 %res = memref.alloc(0) : memref<i64>
23 %threads = ... : index
24 %group = async.create_group
25  %threads : async.group
26 %token = call dispatch
27  (%group, %start, %stop, %res)
28 async.add_to_group %group, %token
29 async.await_all %group
30     return %res
31 }
```

Listing 5.21: Transformed Parallel Loop in a Separate Function called Asynchronously

To parallelize the loop with the async dialect, the loop body is extracted and a new function with a loop over the iteration sub-range is generated, taking all used values of the loop body as parameters. For our example, the result of this transformation can be seen in Lines 1-9. For parallel loops, this is a straightforward process, as each loop iteration is independent from the others and thus no data dependencies exist between loop iterations. For reductions, this process is more involved, as each loop iteration updates the same memory reference. However, the reduction operations are well separated from parallel operations through a separate reduction region which contains all operations which generate data dependencies. As we only need simple operations, e.g. additions, to support common database aggregation operations, we are only focusing on parallelizing these and leave more complex reductions with multiple operations subject to future implementation. Considering that modern processors usually support atomic operations for the subset of operations we need, we add the reduction as a single atomic read-modify-write operation after the loop (cf. Line 7). The atomic instruction takes a memory reference to the final result and adds the local result after the local reduction loop finishes. As the same memory

reference is passed to every asynchronous call of the function, the final result can be read from this location after all parallel computations finished in Line 30.

As an alternative, it would be also possible to pass every reduction instance a separate memory location in an array and after all threads have finished, to sequentially reduce the local temporary results to a final result. This approach would be also suitable for more complex reductions, but has an additional overhead that is inherently serial. For atomic operations, depending on the number of concurrently running reductions, the overhead should be smaller, even though not necessarily entirely negligible, as with a growing number of concurrent workers, the risk of cache invalidation rises. This is an inherent disadvantage of atomic operations, but because the atomic operations are entirely implemented in hardware, the cost is nevertheless rather small. As another alternative, the asynchronous dialect allows the use of asynchronous values, which are essentially promises of the return values of asynchronously called functions. It is possible to wait on these promises until the return value of the function is available, which is then returned from the promise. This approach is similar to the approach of an array with local reduction result and has similar drawbacks, it needs a second, serial reduction stage. Therefore, we decided to use atomic instruction as global synchronization of the reduction results. To call the newly generated function asynchronously, the transformation replaces the parallel loop with calls to the asynchronous runtime to start multiple asynchronous instances, each running a local reduction in parallel. To start up all instances, there are two possible strategies available. Either, the main thread starts all threads sequentially, or the main thread starts a separate dispatch thread that by itself spawns a further dispatch threads until the final number of instances is started. When this is the case, all dispatch threads call the working function. This strategy is called asynchronous, recursive work-splitting. The asynchronous recursive dispatch could lead to a considerably faster start-up of the workers, as it parallelizes the otherwise sequential process of starting up multiple threads. Therefore, this strategy is the default strategy for asynchronous start-up.

6. Evaluation

In this chapter, we evaluate the performance of Voila-MLIR. Our goal is to determine the impact of the introduced variability on the performance, as well as the performance compared to existing state-of-the-art approaches. Therefore, we will start by describing our execution environment and the structure of our benchmarks. Afterwards, we discuss the results of our variability experiments and continue with the comparison of our approach to other approaches. To conclude our evaluation, we contemplate possible threats to the validity of our analysis.

6.1 Setup

In the context of performance variability, we do not only have to consider different query characteristics, but also the hardware characteristics. To embrace this even if only on the small scale, we run our experiments on three different machines, listed in Table 6.1.

Machine Name	Processor	RAM	OS	SIMD
Machine 1	Intel Xeon E-2286M@2.4GHz	128GB	Arch Linux	AVX 2
Machine 2	Intel Xeon Gold 6130@2.1GHz	346GB	Ubuntu 18.04	AVX-512
Machine 3	Intel Xeon E5-2630 v3@2.4GHz	1008GB	CentOS 7	AVX 2

Table 6.1: Benchmark Machine Specifications

These machines belong to three different architecture generations, reaching from one of the first generation of AVX2 CPUs in Machine 3 to Machine 2 with two AVX-512 execution engines. Even though, these machines have different hardware characteristics, the software, such as operating system, compiler and libraries have significant impact on the performance as well. While we were not able to use the same operating system and kernel on all three machines, we did use the same compiler on all three systems. For our experiments, we used Clang[14] at version 13.0.1, the

[14]https://clang.llvm.org/

most recent version we found working on all of the test machines. As compile flags we used parameters commonly used for highly optimized code `-march=native -O3 -flto`. `march=native` ensures that the compiler generates code tailored to the architecture using its features where available. These features include, for example AVX instructions. In order to effectively use these, we compile our experiments with the highest optimization level (`O3`). At this level, the compiler aggressively tries to vectorize code, as well as apply other expensive optimizations. `flto` enables link-time optimization, which consists mostly of inter-procedural optimizations as, for example, inlining, dead code elimination and devirtualization and can lead to better performance. Unfortunately, on Machine 3, the OpenMP, as well as the Intel TBB libraries were too old to conduct experiments with them. Therefore, we will have no OpenMP variants for Machine 3 and no comparison to the state of the art for this machine. Furthermore, we were unable to run the experiments with the large data sets on Machine 1, due to lack of RAM. Aside of these drawbacks, we were able conduct all tests. For the testing, we deactivated frequency scaling on all machines to reduce the noise introduced by sudden frequency changes due to the operating system detecting changing load and throttling the CPU.

6.2 Dataset & Queries

We decided to use the TPC-H [Cou14] benchmark as workload for our experiments. The reason behind that is on the one hand, because the data set, as well as the queries are designed to represent a variety of real life OLAP workloads, and on the other hand, because the benchmark is used widely in the community, making it well understood [BNE13; Dre+20] and making the experiments based on the benchmark comparable to other research. However, we do not use the complete benchmark, but only a representative subset of typical OLAP queries also chosen for evaluation by the authors of our competitors Typer and Tectorwise. The selected queries are Q1, Q3, Q6, Q9 and Q18, whose key properties we list in the following:

Q1: Multiple grouped aggregation on decimal data types with small group size

Q3: Large joins with push-down selection

Q6: Aggregation on selection with multiple predicates

Q9: Large joins with string matching

Q18: Large joins with high-cardinality aggregation

We also modified the queries to match the feature we currently support in our framework. First, we omitted `ORDER BY`-clauses in the queries. On a real-world database, those statements would test the query optimizer to select an index or join variant efficiently, but our framework does not have built-in high-level query optimization, as this is no use case for our framework. Thus, sorting is just a separate, constant cost factor after the actual query was executed and we argue that the sorting performance is not relevant for our evaluation

Due to time limitations, we were not able to implement joins and string support in our framework. Therefore, we have to adapt the benchmark according to these limitations. To work around the missing join support, we use denormalized tables as proposed by the WideTable approach [LP14]. This allows us to express joins as selections with the same predicates as the join on the join columns, which we demonstrate on an example in Listings 6.1 and 6.2.

```
1  SELECT *  FROM
2  customer join orders
3  on (c_custkey = o_custkey)
4  join lineitem
5  on (o_orderkey = l_orderkey);
```

Listing 6.1: Query with Joins

```
1  SELECT *  FROM
2   customer_orders_lineitem
3  WHERE c_custkey = o_custkey AND
4   o_orderkey = l_orderkey;
```

Listing 6.2: Query on Wide Table

Our wide tables are generated by full outer joins and we did no additional changes. However, these outer joins lead to significant blow-up to the data TPC-H data set of a factor of around x5, thus our benchmarks are restricted to a maximum scale factor of 100. As Li and Patel have shown, the denormalization can have a major performance impact on our evaluation[LP14]. Therefore, we will evaluate the impact separately in Section 6.4.2, to interpret our results with regard to the modifications.

Another modification we use is the dictionary compression of the data set to allow our framework to work with tables that contain string data. Therefore, we processed the wide table data, compressing all columns that contain strings to integers while preserving lexicographical order. This allows for potential range queries on the compressed data, if needed. Other columns with integer or decimal data types are used as is, only date strings are imported as integral types. As a consequence, we had to replace the pattern matching in Q9 with an `IN` condition. The resulting query looks as follows in Listing 6.3.

```
1   SELECT nation, o_year, sum(amount) as sum_profit
2   FROM (
3   SELECT n_name AS nation, extract(year from o_orderdate) AS o_year,
4   l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity AS amount
5   FROM part_supplier_lineitem_partsupp_orders_nation
6   WHERE
7   s_suppkey = l_suppkey
8   AND ps_suppkey = l_suppkey
9   AND ps_partkey = l_partkey
10  AND p_partkey = l_partkey
11  AND o_orderkey = l_orderkey
12  AND s_nationkey = n_nationkey
13  AND p_name IN  ([DICTIONARY VALUES OF %COLOR%])
14  ) AS profit
15  GROUP BY nation, o_year;
16  --ORDER BY omitted
```

Listing 6.3: Modified Q9 With Compression and Wide Table Selections

The other queries also have to be adjusted according to the modification of the data set. For Q3 and Q18, we replace the joins with wide table selections and additionally for all tables, we replace string and date comparison with integer comparisons, mapping the date data types to integers. The queries can also be found in the Appendix in Listings A.1 to A.4. As the compression does also have effects on the runtime of our experiments we also evaluate and discuss this in Section 6.4.2.

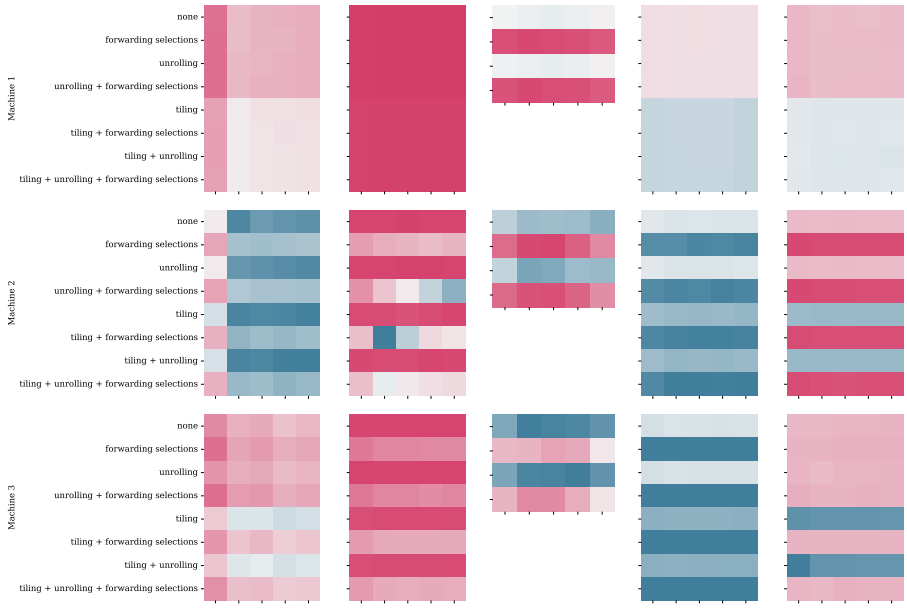
6.3 Variability Benchmark Results

To get an overview of the performance characteristics and bottlenecks of our framework, we conduct a three-fold empirical evaluation. With the next section, we start with experiments determining the influence of optimizations on the runtime of the TPC-H queries for different machines. Afterwards, in Section 6.4, we quantify the influence of the modifications on queries and data set on the runtime of our framework. We achieve this through the use of a pure C++ baseline implementation of the queries with an implementation based on the implementation of the frameworks algorithms. At last, but not least, in Section 6.5, we compare the quality of our best generated variants compared to the state-of-the-art vector-at-a-time and compiled query execution engines Tectorwise and Typer. To get statistically robust results and at least partially cancel out noise during the measurements, we repeated all benchmarks 100 times and use the median query run time without compilation time over all iterations as robust average, unless explicitly stated otherwise.

6.3.1 Overview

To analyze the influence of our optimizations and their combination to variants, we run an extensive subset of all possible variants. As subset we choose the optimizations tiling, unrolling, vectorization, parallelization and selection forwarding as they are larger, commonly applied optimizations[BBS13; RBZ13; Ros+15; Bre+18]. For tiling, we let the framework itself infer a tile size, which is typically the number of loop iterations divided by the number of executing threads to parallelize the computation of a single tile to a single thread each. For unrolling, we decided to use a constant unroll factor of 16. While this factor is larger compared to commonly used unroll factors of 4 or 8 [HP11], it should have no negative effect on the run time and only influences the compile time, which we will analyze later on. Generally, we apply all optimizations greedily on the complete query. This also holds for additional optimizations that we use as base for the other optimizations, which is inlining, loop fusion, as well as common canonicalizations and common sub-expression elimination (CSE). With our framework it is generally possible to apply optimizations on the level of single operations, but this would generate even more variability whose handling would be out of scope. Even for our rather small set of optimizations, we generate 120 variants to evaluate. To keep the overall runtime of the variants within an acceptable time frame, we run the variants only on the data set at scale factor 1 which corresponds roughly to 1 GB of data processed.

(a) Variants Performance for all Machines Without Parallelization



(b) Variants Performance for all Machines With Coroutines

(c) Variants Performance for Machines 1 and 2 With OpenMP

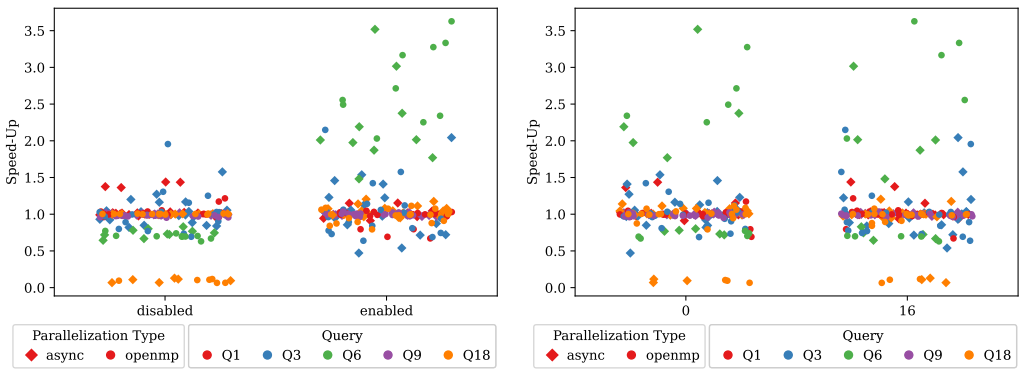
Figure 6.1: Heatmaps Showing the Run Times of the Query Variant Space

To get an overview of the performance space of the generated variants, we use heat maps showing the distribution of runtime over the variant space in Figs. 6.1a to 6.1c. Each block of heat maps characterizes the performance of all machines with one kind of parallelization. For each block, each row represents one machine from Machine 1 to Machine 3. In the heat-maps themselves, darker colors represent lower runtime, while each column represents the vector size used in vectorization from vector size 1=off to a vector size of 16 in steps of times 2. The rows represent combinations of the left optimizations. Unfortunately, variants with tiling that involve vectorization could not be successfully generated for Q6. Therefore, we decided to exclude variants with tilling entirely for Q6. We discuss the reason for this failure in detail in Section 6.3.4.

6.3.2 Parallelization

Additionally to the qualitative comparison of the influence of parallelization, we use a qualitative analysis to identify the effects of parallelization more in-depth. Therefore, we use heatmaps that show the speed-up of parallelized variants over the same non-parallelized variants. Additionally, we vary an additional optimization, and split the variants on the x-axis. In the following, we want to show the influence of parallelization on the optimizations of selection forwarding (cf.Fig. 6.2a) and unrolling (cf. Fig. 6.2b). On the x-axis, the options for the optimizations are shown and each point in the plot represents the speed-up of one variant, either with ,or without the optimization.

Observing the behavior of parallelization among the Figs. 6.1a to 6.1c, positive effects are visible for Q1 on machines 1 and 2 for variants without vectorization. Additionally, parallelization leads to a speed-up of Q6 for variants with selection forwarding, as we show in Fig. 6.2a, but without unrolling (cf. Fig. 6.2b). For selection forwarding, there is an especially noticeable speed-up for Q6 and also for Q18 to reach break-even. The other queries have a largely unchanged speedup scattered around 1.0. For unrolling, there is no such change visible between unrolling off and on, the queries show essentially the same behavior, except for three to four outliers of Q3. However,



(a) Influence of Parallelization on Selection Forwarding for Machine 2

(b) Influence of Parallelization on Unrolling for Machine 2

Figure 6.2: Stripplots of Speed-up for Parallelized Variants vs. non-parallelized. Note, that points are randomly jittered.

for Q18 there is also a negative impact on the performance for all machines, especially

for variants without vectorization. Between the parallelization variants of llvm coroutines (async) and openmp, we can see that for very good performing variants, the run time is roughly equivalent, while async seems to accelerate more and also non-optimal variants for e.g. Q1 and Q3 on Machine 2. Overall, the implementation of parallelism does seem to have rather minor effects on certain variants, but not on all of them. This contradicts our expectations that all queries should at least be parallelizable on a level that shows positive effects on every query. Since this is not the case, we conclude, that our parallelization pass was not able to parallelize all loops necessary to achieve a runtime improvement and furthermore, has blocked other optimizations from being applied. As an example of this occurrence, we select Q18, which shows a generally higher run time with than without parallelization.

6.3.3 Vectorization

When it comes to vectorization, we can observe clearly positive effects only on the run time for query Q6, and there are also small positive effects visible for Q3 and Q18. At the same time, for Q1, Q3 and Q18, vectorization leads to a major slow-down compared to non-vectorized variants. Overall, the vectorization does not seem to be influenced by any other optimizations than parallelization (cf. Appendix A.1.2 and Fig. 6.3). On parallelized variants, the effect of vectorization is generally smaller, which is expected, as most queries are memory and not compute bound and vectorization can only lead to significant speed-ups for compute-bound algorithms. Consequentially, if the parallelization increases the memory throughput, but the

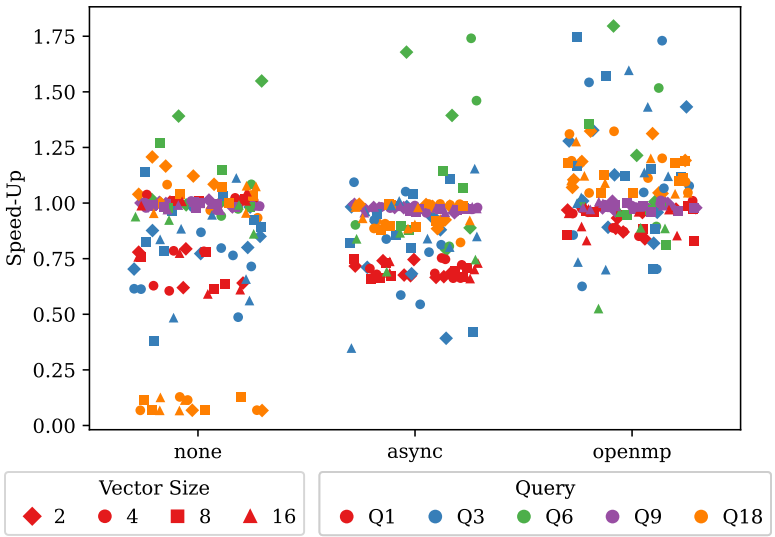


Figure 6.3: Stripplot Showing the Influence of Vectorization on Parallelization for Machine 2

arithmetic density stays very low, the speedup of vectorization compared to serial execution decreases. Unfortunately, we do not see any increase in performance for Q1, which is the query that benefits most from vectorization [BNE13]. We attribute this to our hash table implementation that has a probing algorithm that is currently not vectorizable due to the conservative nature of the vectorization algorithm that only vectorizes loops whose memory access consists entirely out of affine operations.

Interestingly, tiling, even though showing overall bad performance, is able to increase the performance of vectorized variants of Q3 on Machine 2, a fact that we also visualize in Fig. 6.4.

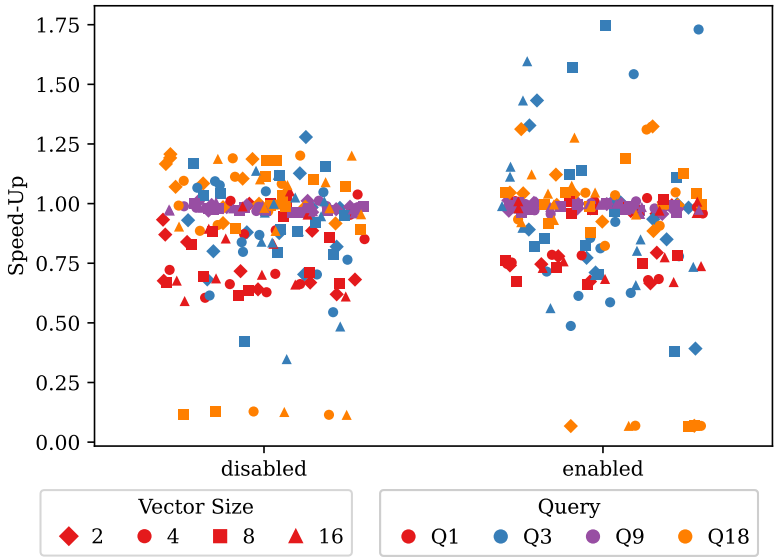


Figure 6.4: Stripplot Showing the Influence of Vectorization on Tiling for Machine 2

This effect is only visible for very specialized variants, for instance the *tiling+forwarding selection+unrolling* variant on Machine 2 with async parallelization and does not represent the best variants that are still all untiled. Therefore, we conclude that tiling is not the main factor for the relative speed-up, but rather a combination of all these optimizations.

6.3.4 Tiling

We were unable to apply the tiling optimization to Q6, because the generated parallel loop used a vector type induction variable for reduction, but since parallel reductions are only allowed for scalar types that support atomic read-modify-write operations, those variants could not be lowered successfully. The solution would be to lift the restriction of scalar types and also allow vector types for reduction, which would change parallel loop lowering to ensure that the reduction variable can still be changed atomically. This could be done through thread local variables and an extra serial reduction step or mutexes. Unfortunately, at the time of the evaluation we did not have the time to implement this solution and thus we exclude Q6 entirely from the tiling analysis. For tiling, we observed largely negative effects for most variants that do not already have a low run time compared to other variants. This behavior was largely independent of parallelization and vectorization and is consistent for all machines. In Fig. 6.5, we exemplary show the speedups of tiling for different vector sizes, while plots for the other machines can be found in the appendix Figs. A.1a and A.1b.

All speedups are below one, meaning that all tiling variants behave worse than their non-tiling counterpart and as all clusters are basically located equally on the y-axis,

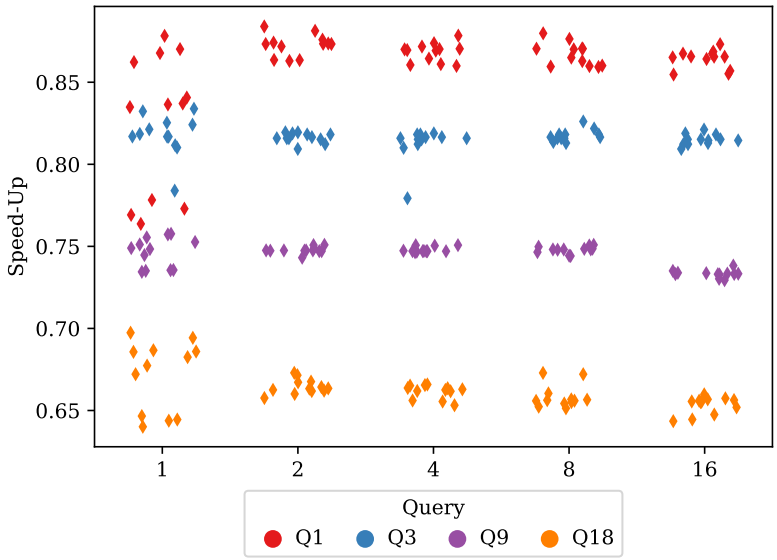


Figure 6.5: Stripplot Showing the Influence of Tiling on Vectorization for Machine 1

independent of the vector size, SIMD has no influence on the run time if tiling is enabled. Furthermore, we can see that tiling has the least performance impact on Q1, followed by Q3, Q9 and the largest on Q18. From the other optimizations, only selection forwarding seems to make the variants gradually independent from the impact of tiling such that they reach a speed-up of 1 (cf. Fig. 6.6).

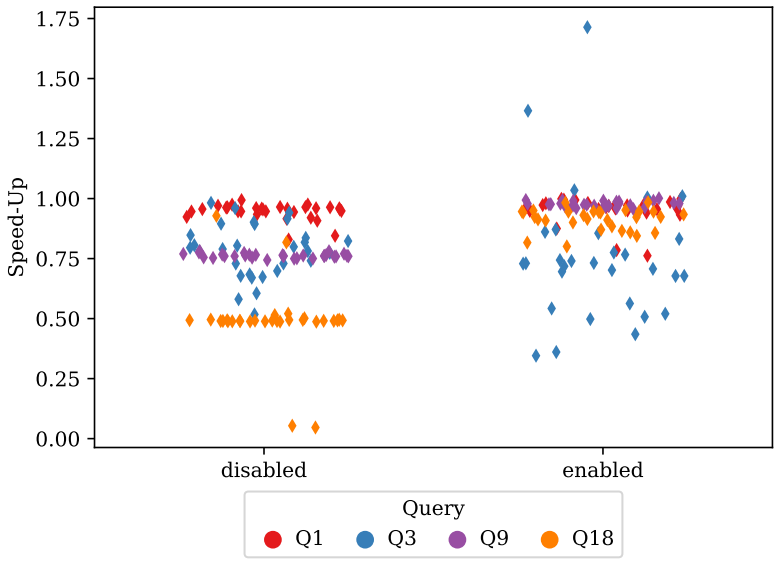


Figure 6.6: Stripplot Showing the Influence of Tiling on Selection Forwarding for Machine 2

A single exception to this behavior builds only Q3 on Machine 2. Here, Q3 becomes generally slower with selection forwarding enabled, but two variants appear to undergo a larger speed-up. However, this behavior is unlikely to have any major implications as forwarding selections in general show a bad performance for this query. We did not

expect tiling to have large positive effects, as for algorithms with deeply nested loops that are not cache-conscious [HS17]. On the other hand, we did not expect tiling to under-perform so badly, as we used it mainly to tile loops for better parallelization afterwards. We could not observe this behavior, partially because the generated loops are not as trivially parallelizable as we expected, which means that our parallelization pass was not able to accelerate these loops and partially because algorithms seem to already have good memory access patterns for 1-dimensional data and therefore tiling has shown itself to be largely unneeded.

6.3.5 Unrolling

Unrolling appears to have overall only minor effects on the variants. We observe a barely visible influence on Q3 and Q6 in the heat map. Dependent on the vector size, the performance varies (mostly decreases for vectorization). This matches with our expectations, as unrolling is a rather small optimization and mostly used to reduce control flow overhead on loops. For a deeper understanding of how unrolling interacts with other optimizations, we compare the speed-up of variants with a feature turned on vs. of for unrolling constantly enabled. The full results can be found in Appendix A.1.3.

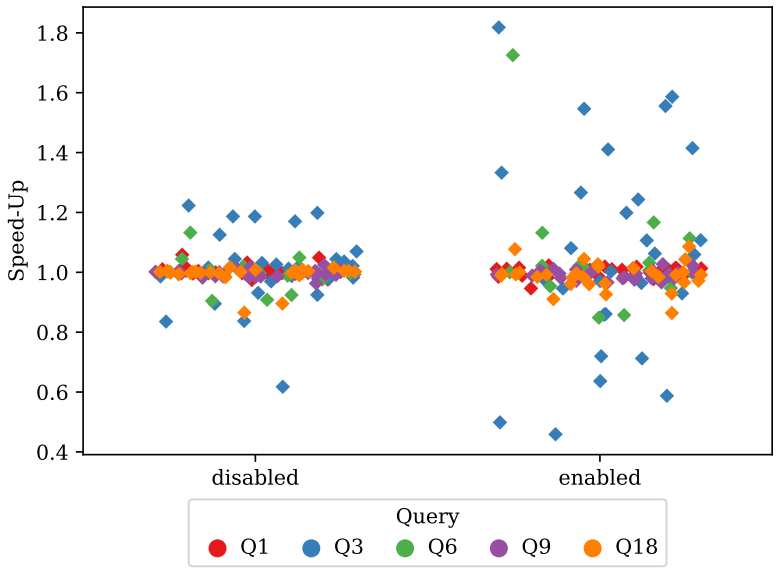


Figure 6.7: Stripplot Showing the Influence of Unrolling on Selection Forwarding for Machine 2

It turns out, that while the influence of unrolling is barely visible it still has some influence on the performance. As most well performing variants are combined from unrolling and selection forwarding, in Fig. 6.7 we show the effect of unrolling on selection forwarding on Machine 2, standing as a representative for all other machines. While we have no visible effect of unrolling on Q1 and Q9, unrolling shows distorting effects on the performance of Q3 and Q6. Some variants have a speed-up of up to factor 1.8, others become slower by up to 50%. As the effects have shown to be rather chaotic, we conclude that unrolling has no major effect on selection forwarding alone,

but possibly multiple optimizations together. To confirm this thesis, we show the behavior of vectorization together with unrolling in Fig. 6.8, as this chaotic behavior was also visible, especially on suboptimal variants with unrolling alone in the heat maps.

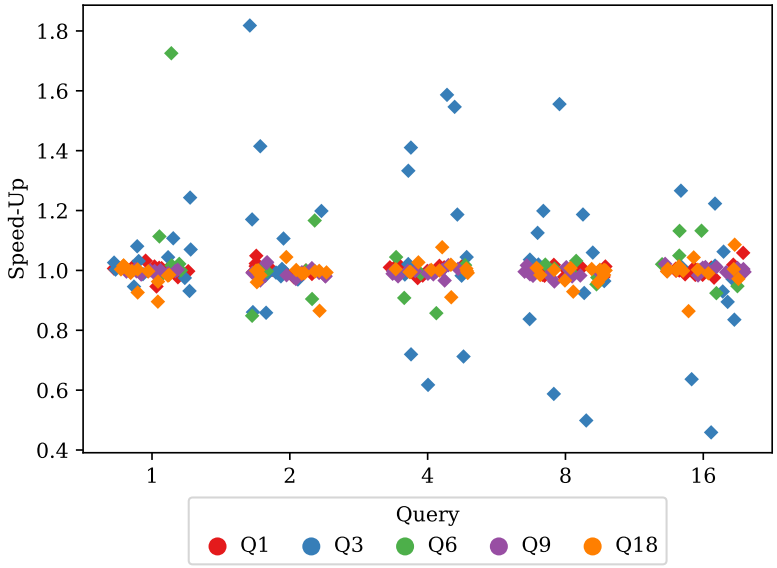


Figure 6.8: Stripplot Showing the Influence of Unrolling on Vectorization for Machine 2

For vectorization, we can again see distorting effects on Q3 and Q6, which match with the previous observations for unrolling and selection forwarding. Thus, the speed-up we observe from unrolling is likely very minor and most of the effect comes from the selection forwarding in combination with vectorization. In the next subsection, we will investigate in this further.

6.3.6 Selection Forwarding

One major influence on the performance of our variants has shown to be the selection forwarding optimization. Independent of the machine, parallelization library or vector size, it decreases the run time for queries Q1, Q6 and Q18. For Q3 and Q9, it has the complete opposite effect of a significant slow-down. This slow-down arises from the way, selections are forwarded to our hash table implementation. For our hash table, we use the highest 64-bit value as an invalid entry but hide this implementation detail from the dialect for simplicity. When it comes to lookup from the hash table, we have to check the selection condition on the lookup index. This produces a conditional branch that prevents the lookup loop from fusion with the following operations, as the current loop-fusion algorithm is rather simple in its nature and does not fuse loops with non-affine regions such as the region of the conditional branch. In Listing 6.4, we show a small example of this problem.

The two affine loops have the same bounds and their bodies can be executed element-wise one after another without interfering. The affine analysis of the loop regions has

```
%cond = ...
 affine.for %ji= 0 to %N {
    //calculation of selection condition
    ...
}
//simplified selection loop
affine.for %j = 0 to %N {
    %p = affine.load %cond[%i]
    scf.if (%p)
    {
        //selection in non-affine then region
        ...
    }
}
```

Listing 6.4: Example Linalg Generic Operation with Reduction Iterator Representing a Horizontal Sum Operation on Tensors

to prove that this is indeed the case. Therefore, all loops have to be affine loops that only contain affine regions and affine memory accesses. Unfortunately, this is not the case for the second loop, because it contains an if operation and the corresponding region to an if operations is not affine. Therefore, the affine analysis assumes the loops are not fusible. To overcome this problem, a more powerful side effect analysis has to be implemented, so it can analyze non-affine regions and take into account branching. As this can become very complex, we can implement an annotation that can be transitively forwarded through the lowering pipeline and indicates that loops can be safely fused.

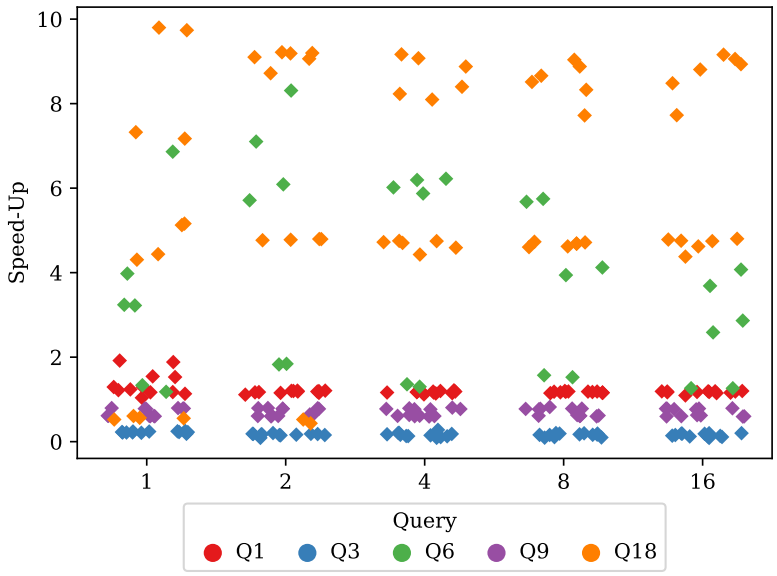


Figure 6.9: Stripplot Showing the Influence of Selection Forwarding on Vectorization for Machine 2

In Fig. 6.9 we show an example of the influence of selection forwarding on vectorization for Machine 2. The effects on the other machines and optimizations are generally

similar and are shown in Appendix A.1.5. As already visible from the heat maps, forwarding selection has independently of the vectorization strong effects on the run time. On Machine 2, there is a speed-up of nearly x10 for Q18. On Machine 1, the effect is more restricted to Q6, but also Q3 and Q9 are less negatively influenced. However, with larger vector size, we see a decrease in peak performance for Q6 and Q18, while at the same time more variants profit from vectorization. As our main goal is not the plateau of good overall run time, we are more interested in the variants mostly influenced by selection forwarding. Unfortunately, these are still the unvectorized versions, which should not be the case. Our expectation was, to have selection forwarding speed-up vectorization through a more regular access pattern. This shows that selection forwarding does not have as high an influence as expected. Nevertheless, we have shown that selection forwarding is one of the most influential optimizations. Considering its nature as a more algorithmic optimization, this is an understandable effect.

6.3.7 Statistic Correlations of Optimizations with Run Time

Additionally to the qualitative overview through the heat maps and the in-depth view of pairwise influence between two optimizations, we want to find correlations between optimizations and the run time of the variants. This is often done using the Pearson correlation coefficient, which can detect correlations between two continuous sets of data and gives the correlation on a range between $[-1, 1]$, where -1 is a strong negative correlation, 0 no correlation at all and 1 a strong positive correlation. However, to get meaningful results of the Pearson correlation metric, variables of the two sets have to be approximately normal distributed[KZ99]. This property does not hold for our data set. In fact, for the optimization categories, we always have a uniform distribution as our subset of variants is extensive regarding our optimizations. This means that for example, the tiling optimization with the possible values on and off is exactly half the times of our 120 variants on and the other half off, hence the optimizations are equally distributed over all variants.

Therefore, we use the Spearman rank-order correlation algorithm[Zar05] provided by the *scipy* framework[15]. This algorithm does not depend on the distribution of data and also gives the result of a statistical test describing how large the probability is that the correlation could also be generated by a random data set. However, even the Spearman rank-order correlation is only really meaningful for continuous values, which is not satisfied for our values of the optimizations that are categorical. Nevertheless, we are confident that the results are applicable for our case, as we used the correlation results conservatively and only to recognize trends. Furthermore, we can verify the results with our previous analysis. To get the correlations for each optimization, we choose the optimization parameter value and tested the correlation with the median run times of the variants. The resulting correlation data are listed in Table 6.2. For values with $p \geq 0.05$, we indicate the correlation value with *. However, as we use partial categorical values and only a rather small data set for correlation, the p-value is not very reliable and we only include this as an indication. Overall, we do not see strong correlations for most of the optimizations, as some

[15] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html

Machine	Query	Tiling	Vectorization	Unrolling	Parallelization	Selection Forwarding
Machine 1	Q1	0.766	-0.05*	-0.047*	-0.093	-0.006*
	Q3	0.871	-0.146*	-0.056*	0.174*	0.0467*
	Q6	–	0.086*	0.026*	0.051*	-0.866
	Q9	0.866	0.129*	0.125*	0.115*	0.022*
	Q18	0.758	0.031*	0.008*	0.204	-0.048*
Machine 2	Q1	0.348	0.308	-0.021*	-0.062*	-0.727
	Q3	0.356	0.044*	-0.013*	0.012*	0.866
	Q6	–	0.124*	0.015*	0.01*	-0.866
	Q9	0.388	0.139*	0.069*	0.09*	0.866
	Q18	0.373	0.167*	0.001*	0.148*	-0.693
Machine 3	Q1	0.668	0.358	-0.028*	-0.0547*	-0.515
	Q3	0.433	0.0637*	0.008*	0.0476*	0.866
	Q6	–	0.17*	-0.0173*	0.0303*	-0.779
	Q9	0.433	0.0538*	-0.011*	0.1*	0.866
	Q18	0.316	0.124*	0.001*	0.294*	-0.668

(*) not significant with $p \geq 0.05$. Values range from $[-1, 1]$ where -1 is strong negative correlation, 1 is strong positive correlation and 0 is no correlation

Table 6.2: Spearman Rank-Order Correlation on Performance of Single Variant

of our optimizations are likely micro optimizations that naturally do not lead to significant changes in the run time. Nevertheless, we are able to see some correlations recurring from the heat maps. One major correlation going through all queries and machines is the influence of tiling on the run time. This correlation is positive, meaning that with tiling on, the runtime also increases. This is consistent with the behavior we have observed on the heat maps. Furthermore, for nearly all queries, selection forwarding showed major effects on the run times, which also is indicated by the negative correlation for Q1, Q6 and Q18, and positive for Q3 and Q9, except for Machine 1, where only Q6 showed a correlation.

6.3.8 Summary

To conclude our variant analysis, we have built up a hierarchy of optimizations that fit best for each query and respectively their characteristic. Selection forwarding has shown to be most effective for queries except Q3 and Q9. Based on selection forwarding, vectorization showed some benefits and together with parallelization variants achieved peak performance. Unrolling had only minor effects on both directions and tiling always led to a worse performance. We have also identified the four largest problems regarding single optimizations not being applied as expected:

1. Parallelization does not always show an effect on the query run time and thus is likely not always to be applied successfully.

2. Tiling shows unexpectedly bad run times, mostly because parallelization was not effective.
3. In most cases, vectorization had a smaller effect than anticipated, which is related to the parallelization problem, as both passes share a similar transformation logic.
4. Affine loop fusion does not fuse loops as expected.

In Table 6.3, we list the optimizations that led to the on average best optimized queries.

Machine	Query	Tiling	Vectorization	Unrolling	Parallelization	Selection Forwarding
Machine 1	Q1	✗	1	✓	async	✗
	Q3	✗	1	✗	none	✗
	Q6	✓	2	✗	openmp	✓
	Q9	✗	1	✓	none	✗
	Q18	✗	1	✗	none	✗
Machine 2	Q1	✗	1	✗	none	✓
	Q3	✗	1	✓	none	✗
	Q6	✓	2	✓	openmp	✓
	Q9	✗	1	✗	none	✗
	Q18	✗	2	✗	none	✗
Machine 3	Q1	✗	1	✗	none	✓
	Q3	✗	4	✗	async	✗
	Q6	✓	2	✓	async	✓
	Q9	✗	1	✗	none	✗
	Q18	✗	1	✗	none	✗

Table 6.3: Fastest Variants as Visualized in Fig. 6.10 with Values for the Parameters

Note, that we have not been able to completely analyze all relationships between optimizations. We have ignored complex relationships between multiple optimizations, as those are even more complex to reason about and due to their shier amount makes them a good candidate for learning algorithms [Bro15].

6.4 JIT vs. Baseline

To evaluate how our approach compares to static compiled code and to estimate the impact of our modification on the TPC-H queries and data set on the run time, we implemented a basic version of the algorithms used in our framework as baseline in C++. For the implementation we only hand-fused all loops, as this amount of loop fusion is not done automatically by the compiler. In the following we show how this statically compiled baseline compares to the dynamic Voila-MLIR framework variants, we found to have the best run time in the previous analysis, and what is the estimated impact of denormalization and compression of tables on the run time of our framework.

6.4.1 Time

For comparison of the baseline with our variants, we choose the best performing serial variants we identified throughout the experiments in the previous section and compare their run times for the queries at SF 1. The results are plotted in Fig. 6.10.

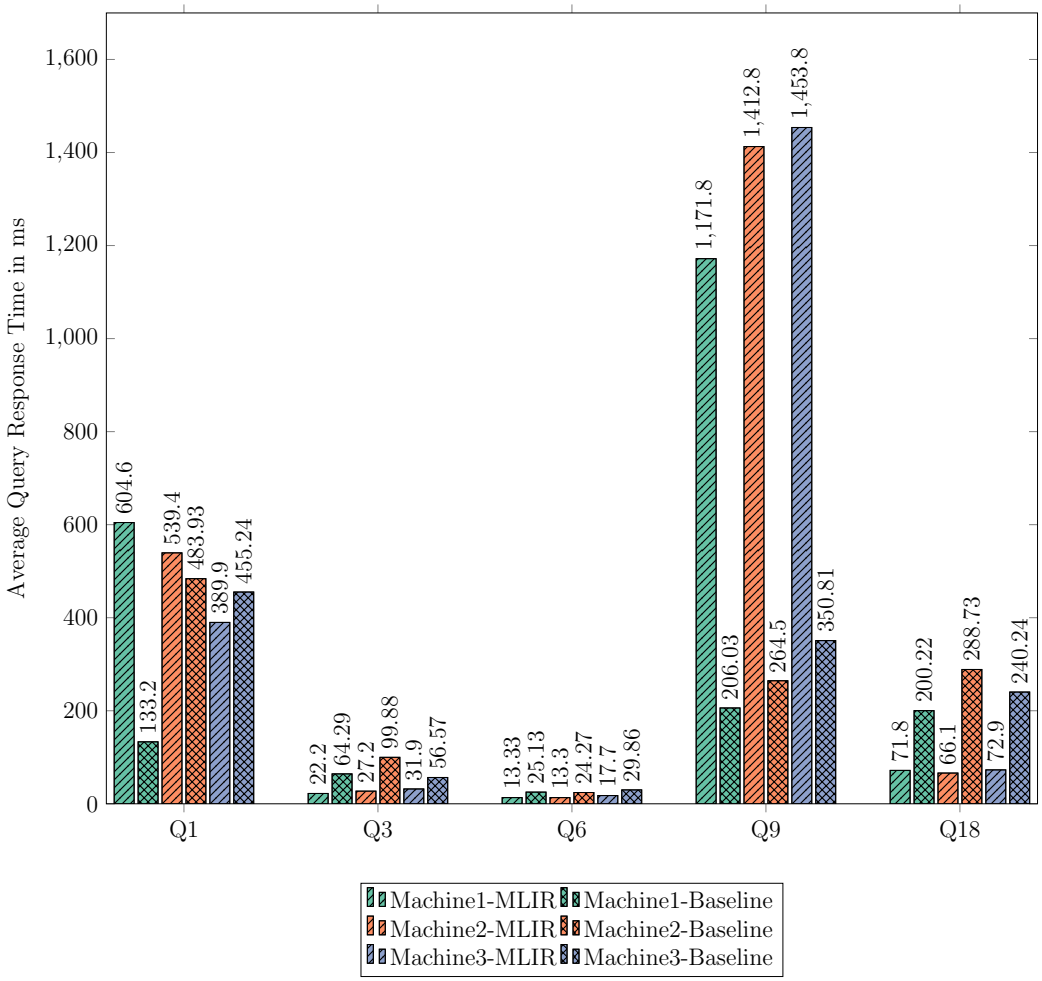


Figure 6.10: Comparison of Baseline Implementation With Best Performing Serial Variants at Scale Factor 1

At first, we observe from the experiment on different machines that the results are largely consistent between the baseline and the framework. The only outlier at this point is Machine 3 for Q1, where the baseline implementation shows a higher run time than the framework one, which is the exact opposite for the other two machines for this query. We also observed this behavior previously, in Fig. 6.1a, where Machine 3 was significantly faster than the other machines. For Q3, Q6 and Q18, we see smaller run times for the framework's variants than the baseline. This lets us conclude, that for those variants we have reached a better optimized code than the compiler could statically generate. This is especially important, because with this results, we show that our approach is able to automatically generate faster code than general purpose compilers can without a considerable amount of guided optimizations through hand-optimized or annotated code. On the other hand,Q1 and Q9 are slower by a factor of up to $\times 5$ when executed with the framework than with the baseline

implementation. This indicates that for those queries, our framework is currently not able to generate better performing variants due to some properties of the query leading to unoptimized code. We have already identified some blocker optimizations in the variant analysis that have these effect on other optimizations. However, when looking at the variants that are best performing, it is not alone their influence, as almost no optimizations are applied to those queries. Thus, the largest difference between the baseline implementation and the framework's is the hand-written loop fusion, which we expected to be always applicable to the framework generated code. As it turns out, most of the hash table related functionality could not be fused into the loops around them, which leads to a serialization of the code-flow and consequently a pipeline break at those operations. Additionally to those synchronizations, this generates intermediate results, as loop variables that are used before and after the hash table operation have to be materialized. To mitigate this performance problem in the future, the loop fusion passes have to be redesigned to do deeper memory access analysis to also fuse loops with nested conditional regions and non-affine memory accesses.

6.4.2 Impact of Compression and Normalization

Following the comparison of the baseline implementation with our framework generated query variants, we analyze the impact of the modifications we employed on the TPC-H data set and queries. Therefore, we compare our baseline representing a C++-based version of our framework's queries to modified versions which work with joins instead of denormalized tables and uncompressed data respectively. While the baseline has shown to not exactly match the performance of the framework generated code, the algorithms are still similar and thus the insights of the baseline comparisons are applicable to the performance of the framework, despite the compiler being able to optimize the C++ code better than the MLIR code. To give an impression on the performance impact, we calculated the speedup of the baseline variant with all adaptions compared to the other two variants with joins on compressed data and joins on uncompressed data at scale factor 1. For the implementation of the joins, we used an adapted version of the non-partitioning hash join[Bal+13]. Furthermore, the denormalization approach replaces the join order problem with a selection predicate order problem and to get the best performance of the queries in both cases the optimal order has to be chosen. As this is not the goal of our thesis, we choose the join order as well as the predicate evaluation order to be the same as in the query description in the TPC-H benchmark. While this does not lead to the best run times possible, we hope to gain comparable results. We show the results for Q3,Q9 and Q18 in Fig. 6.11. The run times of Q1 and Q6 were identical to the run times of the baseline variant, since those queries operate on a single table and not using any string comparisons and since we decided to use a column store format for our data, we can ignore string columns entirely as having no influence on our queries. Therefore, we omitted Q1 and Q6 on the plot. Note that while there are columns with single characters they are technically strings, we just encode them as integers by their ASCII value.

For Q3, we can see high speed-ups of one to two magnitudes of the modified versions compared to joins with or without compression. The compression seems to have

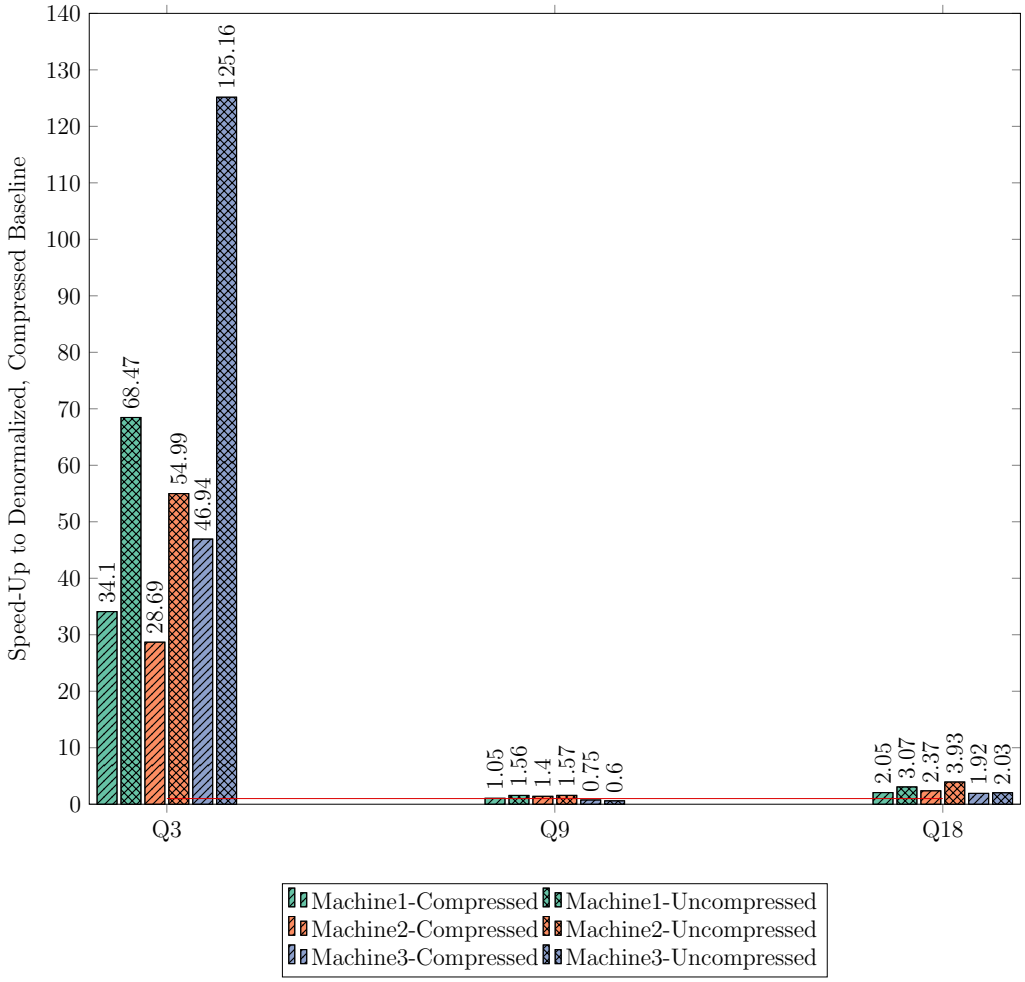


Figure 6.11: Influence of Compression and Denormalization on Query Runtime

an additional speed-up of factor 2× compared to the uncompressed join versions on all machines. The influence of denormalization however, is much higher with a speed-up factor of ×28 to ×54 depending on the machine, which is consistent with the results of the WideTable approach [LP14]. The same counts for Q18, where the speed-up of compression is far less than for Q3, but still within reasonable bounds, as we do not suffer from the same store fetch slow-down as WideTable. Interestingly, Q9 was on tie with the denormalized, compressed variants and on Machine 3, while the modified versions are even slower than the joins on original TPC-H tables. The reason therefore lies in the high run time of the baseline version caused by the hash table lookup for the compressed string matching in the selection condition.

6.5 Voila vs. Typer and Tectorwise

For the qualitative comparison of our approach with state-of-the-art, hand-tuned query execution engines, we selected the best performing variants at scale factor 1 (cf. Table 6.3). Our goal for this experiment is to see how far away is our automatic optimization from being competitive with hand tuned code. As competitors we choose adapted variants of vectorwise [ZVB12] and Hyper [KN11], named Tectorwise and Typer that were used to show the differences in performance for vectorized

vs. compiled execution [Ker+18]. Typer and Tectorwise have also been compiled with Clang 13.0.1, but with the default parameters set in the replication package[16]. We executed the queries on Machine 2 at scale factor 100 and show the results in Fig. 6.12. Unfortunately, we were not able to build the project on Machine 3, as the dependencies were outdated and we were also unable to execute the benchmark on Machine 1, as it does not have enough RAM to execute the benchmark at scale factor 100.

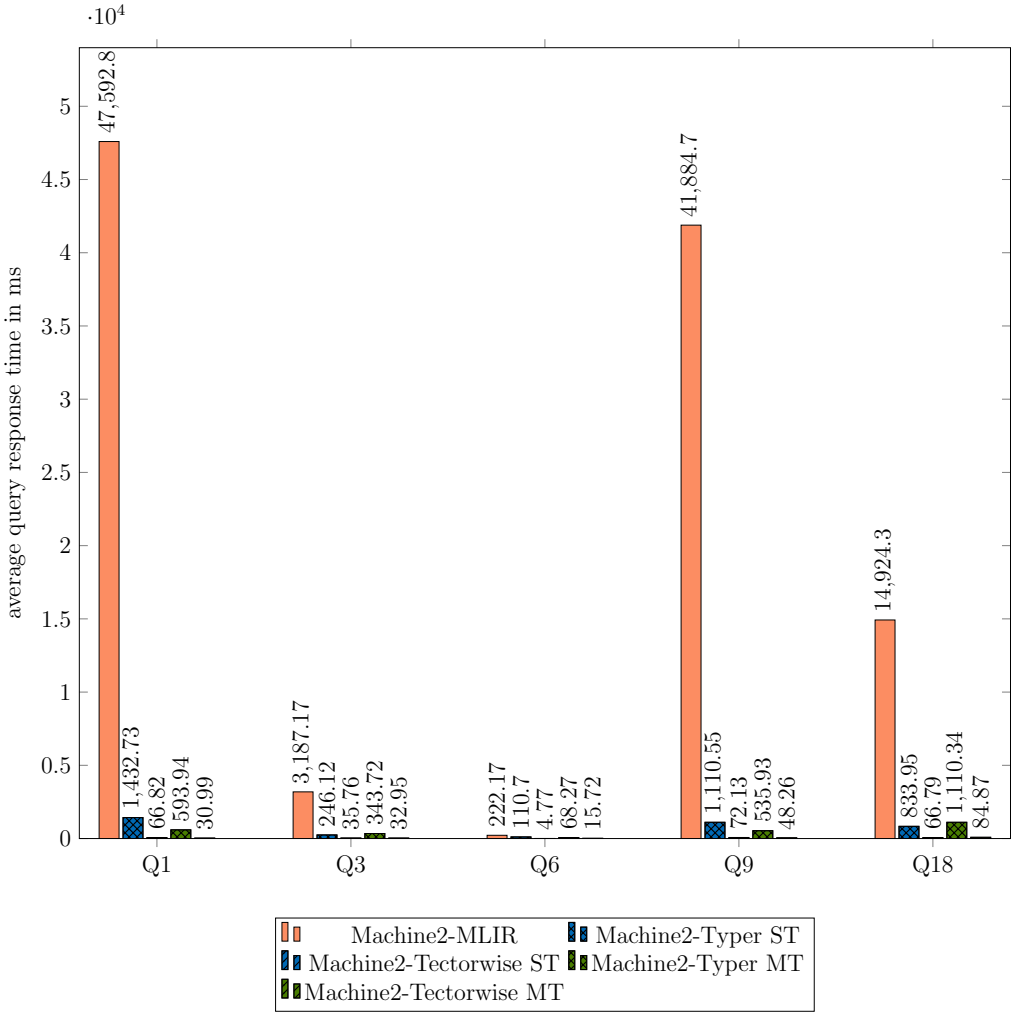


Figure 6.12: Comparison of Best Performing Variants with Typer and Tectorwise Single Threaded and Multi Threaded at Scale Factor 100

Against our expectations, we have not been able to compete with Typer and Tectorwise for queries Q3, Q6 or Q18. For all queries, even the best variants are slower than the single threaded versions of Typer and Tectorwise. However, Q6 the variant at which most optimizations have been shown to achieve a speed-up, is only slower by a factor of 2×. This verifies once more that our optimizations indeed speed up the run time. The longer run time can be explained through the missing use of AVX-512 for our parallel reduction. With AVX 2 there were only half as many elements processed per loop iteration, which leads to an exact 2x difference in the run

[16]https://github.com/TimoKersten/db-engine-paradigms

time we observe. This is probably an issue with the JIT environment not generating AVX 512 instructions, likely due to the general tendency of compilers to not generate 512 bit wide SIMD instructions[17] because of the high impact on processor frequency and resulting lower performance for most workloads[18]. For the other queries, the slow-down turns out to be between a factor of 13 for Q3 to 38 for Q9. We could also verify the gap between Q3 and Q18 compared to Q1 and Q9, which we observed previously, during the comparison with our baseline implementation. We also observe that Q3 and Q18 scale super-linear with the scale factor of the benchmark. This should not be the case, as the number of tuples to process also scales linear. As we observed from the MLIR code generated after the loop fusion pass, those queries suffered from the same restriction of the loop fusion and hash table algorithms that block further optimizations and impose unneeded materializations.

6.6 JIT Compile Time

As the last step, we analyze the run time of the query generator pipeline, as this is a limiting factor for database systems using compiled queries [Vog+18]. The goal for those systems is to achieve a compile time that is lower than the run time to have the compile time not dominate the overall execution time of the query. This is usually true for OLAP workloads that are rather long-running, but becomes problematic for OLTP workloads if they are very short running. As competitive compile time was not a major goal of our approach, we did not optimize for this. However, in the following, we will give a brief analysis of the compile times for our selected TPC-H queries that are all OLAP workloads. We show the results of all queries run at scale factor 1 in a box plot in Fig. 6.13. The center of the box plot marks the median time of the median times of all variants of a single query. The boxes are the quartiles and whiskers include the 1.5 inter-quartile-range. Values outside this range are outliers marked with diamonds. The scale factor does not have any influence on the compile time, as the generated code is the same just with different loop bounds, therefore we mention it just for convenience to the curious reader.

[17] https://reviews.llvm.org/D67259
[18] https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/

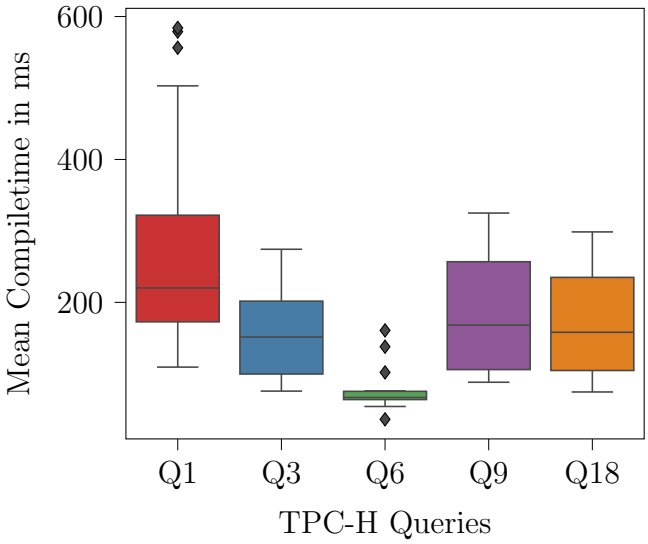


Figure 6.13: Compile Times of Machine 2

As expected, the compile time is not very low, since our compilation pipeline contains many passes that are not necessarily needed, such as canonicalizations or CSE after every lowering. Furthermore, we also activated potentially costly but no-op passes on the LLVM lowering, as we have not yet investigated on which LLVM passes are strictly necessary and which are just low-level pedants of the MLIR passes. We also link in large libraries, such as the OpenMP library at compile time. This can also have a large impact on the compilation time, but the effects can be reduced with techniques like GhostLinkage or building the query program as library and load it into the dependent libraries.[19] However, the compile times for our queries are well within the limits expected from comparison of C++ Hyper and LLVM hyper [Neu11]. Actually, it fits perfectly in between, as our framework has the compilation time of LLVM and additionally the compilation time of MLIR, which is faster than C++, because C++ itself has a high compilation and parsing complexity, while the MLIR compilation is designed to allow parallel compilation. The variation of compile times is somewhat proportional to their complexity, a behavior stable throughout all our machines (cf. Figs. A.21 and A.22). We also see a deviation of the boxes and the whiskers towards the higher compilation time. We explain this with the observation that most of the variants have a rather lower, consistent run time, even though there are some variants with certain optimizations or combinations thereof that have a large impact on the compilation time. This hypothesis is supported by the outliers at Q1 and Q6 that largely lean in the direction of higher compile times. When it comes to a break-even of compilation vs. run time, Q3, Q6 and Q18 are dominated by the compilation time, Q1 and Q9 not. As we have determined a roughly linear performance scaling with growing scale factor previously, we can project the break even for all queries at around a scale factor of 10.

[19] https://llvm.org/devmtg/2008-08/Begeman_EfficientJIT.pdf

6.7 Threads to Validity

Even though we carefully designed our experiments to reduce possible validity threads, there are still several points left that could impose unknown effects on the validity of our results and possible make them less generalizable to other related research. First and foremost, there are the typical problems of benchmarks representing the behavior in reality[Vog+18]. In our implementation, we did not implement any kind of fixed-point arithmetic as it would be the case on real databases and is also the case on Typer and Tectorwise. While this is allowed by the TPC-H benchmark, fixed-point arithmetic can have large influences on the performance and correctness, depending on the operations. For the comparison of our approach with the competitors, we used their entire framework. This made the set-up very convenient, but also makes it more complex to compare the approaches. For example, the Typer and Tectorwise framework uses constant query parameters of the qualification queries of TPC-H. This can seriously change the performance of the benchmark, as modern compilers are able to do extensive constant expression evaluation and possibly eliminate entire subsets of the query algorithms. For our baseline implementation for instance, we had to actively block optimizations of the running loop to stop the compiler from optimizing the entire running loop away. To mitigate this problem and to be more comparable to the TPC-H benchmark, we implemented our query parameters as specified with uniform random selected parameters. Nevertheless, the run times are very sensitive to the abilities of the compiler to optimize the code accordingly, as well as the underlying data structures, algorithms and libraries used. For example, Kersten et al. use hardware comparable to ours for their evaluation of Typer an Tectorwise. However, they use an older version of the GCC-compiler and their results are partially an order of magnitude slower for the single threaded version. We also reproduces this result to a certain degree with GCC-8, single threaded in Fig. 6.14. Thus, the results of those benchmarks are highly dependent on how far the compiler can optimize the code. In this case, especially auto-vectorization of the compiler appears to have significantly improved. With the results of GCC, our variants would be more competitive performance-wise to most of the other approaches, but the comparison would not be nearly fair, as the outdated GCC would not be able to remotely optimize programs as far, as our automated approach supported by the state-of-the-art could do.

We have found our variants to be unable to compete with our selected state-of-the-art competitors. This has the direct consequence that even our best optimized variants are still far away from bare-metal speed and as those we have to consider the generalizability of our variant analysis. While our optimizations show certain connections, their behavior may change when we come closer to the architectural limits in terms of run time. Therefore, our experiments can show relations between the optimizations, but we cannot be sure that the behavior might not change drastically rendering our results incorrect for high-performance applications. For the variability analysis, we also had only a narrow set of machines available to determine the effects of the optimizations. All our machines have modern Intel CPUs with a similar monolithic architecture and deep out-of-order pipelines. Other hardware, such as AMD's Zen architecture, embedded ARM/RISC-V processors or heterogeneous processors may show drastically different behavior for our optimizations. Especially

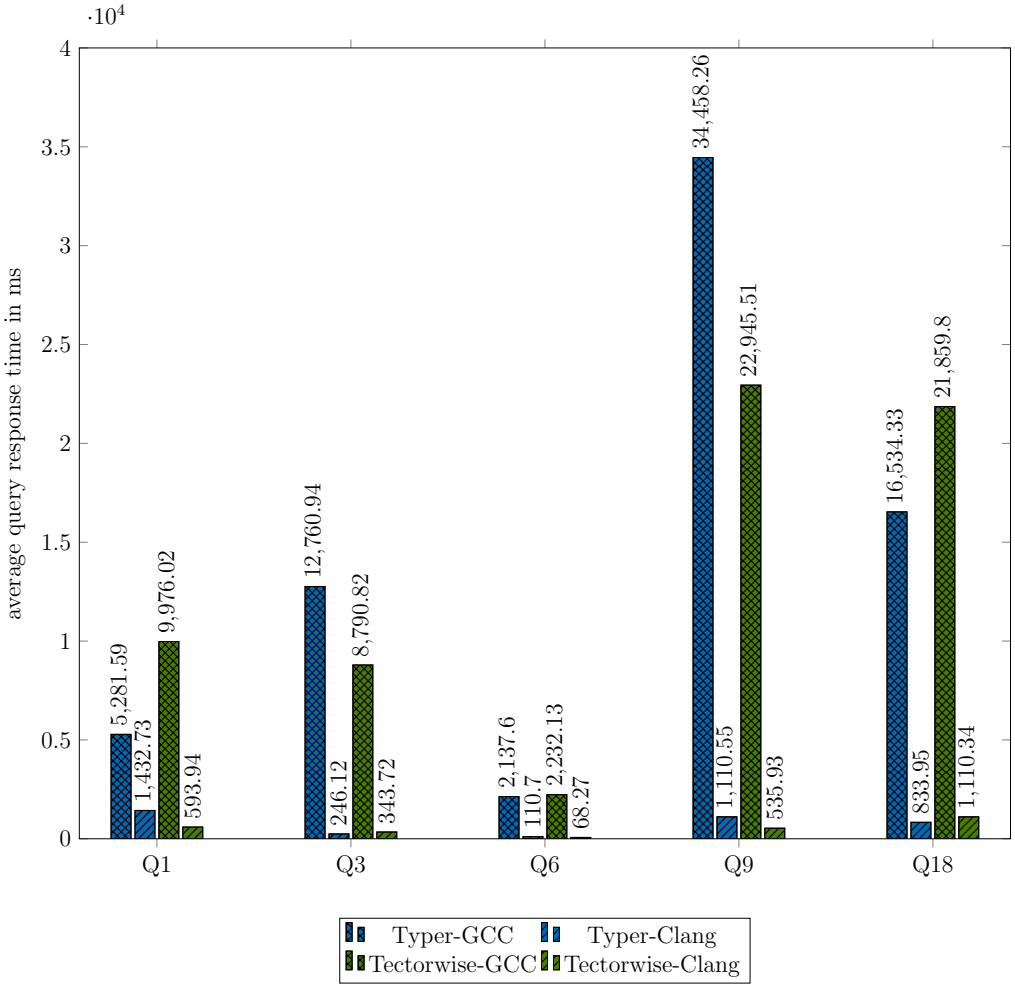


Figure 6.14: Comparison of Typer and Tectorwise compiled with GCC-8 vs. Clang 13.0.1 Single Threaded at Scale Factor 100

for loop unrolling, it is known that in-order CPUs profit more of the optimization, as unrolling removes control flow dependencies and allows for better pipelining which does not play such a role for out-of-order CPUs. Also, vectorization with larger vector sizes could have better effects for GPUs or other coprocessors. Thus, our results can so far only be seen valid for modern monolithic Intel CPUs.

6.8 Summary

In our evaluation, we have at first evaluated the variety in performance for an extensive subset of our variant space. We found that the most influential optimization was selection forwarding influencing every other optimization. For Q1, Q6 and Q18, forwarding selection led to a major speed-up, but it had a largely negative impact on the queries Q3 and Q9. For the tiling optimization, we found that it had a negative impact on the performance for all queries, as soon as the performance of the variant increased over the base variant with our optimizations. For our parallelization and vectorization optimizations, we found that they were not always applied successfully due to limitations in loop fusion, as well as the parallelization and vectorization algorithms.

For our comparison of our approach against a C++ implementation compiled with a general-purpose compiler, we were able to find that our approach outperforms the baseline implementation for Q3, Q6 and Q18 only being slower for Q9 by a factor of 5×. Therefore, we were able to show that our approach can adapt to the query and hardware specifics and outperform state-of-the-art optimizing compilers.

In preparation for our comparison with state-of-the-art database query engines, we evaluated the impact in performance of the modifications we applied to the TPC-H data set. Our results show that the impact is only really relevant for Q3, where compression had a speed-up of 2x and denormalization a speed-up of 24× over the original query on uncompressed, normalized tables. For Q9, we even saw a negative impact of the modification and on Q18 a moderate speed-up of up to factor 2×. For completeness reasons, we have to consider that queries with selection or joins are always sensitive to the order of joins and selections. Therefore, these observations only hold for the default ordering in which the queries are written.

Finally, we compared our approach to the competitors Typer and Tectorwise and have not been able to complete directly. Q1 and Q9 are still well beyond a one order of magnitude slower with our best variant. Only for Q6 we were able to come as near as factor 2× slower. This is due to our LLVM backend during JIT compilation does not emit AVX-512 instructions that could process double the amount of AVX 2 instructions our variants use. However, these results vary greatly with the used compiler for Typer and Tectorwise. The compiler we used for our benchmark is far more capable of vectorization and other optimizations than the compilers installed per default on most operating systems. In a brief comparison, we have seen a speed-up of factor 5× to 50×, depending on the query. Under these circumstances, our variants are at least comparable in performance.

To quantity the influence of the JIT-compilation overhead for query execution, we measured the range of the mean compile times. We found that there is a slight connection between query complexity and compile time, with more complex queries taking longer to compile. Furthermore, we found that more optimizations increase the compile time and the more complex they are, the longer they take. Especially tiling was an expensive optimization. Overall, the compile times were between the compile times of high-level languages such as C++ but higher than direct compilation of LLVM.

7. Conclusion

7.1 Summary

In this thesis, we proposed a novel approach of a domain-specific language framework specifically tailored for adaptive query execution on heterogeneous systems. To achieve this final goal, we first adapted Voila as a vector-based query language for basic syntax and semantic of our language. Upon this language definition, we developed a language front end for our language based on the MLIR framework. The MLIR framework is designed to represent language dialects within a single intermediate representation to reuse common algorithms and the strengths of specialized dialects to generate high-performance programs for heterogeneous hardware. These dialects and transformations also allow for a comparably simple, yet very effective mechanism of variability, as optimizations can be directly implemented in the compiler on a level perfectly suited for the transformation. Following this idiom, we implemented a complete lowering pipeline from the Voila-MLIR dialect down to LLVM IR, from which the code is compiled to the target architecture and afterwards executed. In this pipeline, we used several built-in MLIR dialects to gradually lower the level of abstraction and apply transformations where best suited. Directly at the level of Voila-MLIR, we implemented a technique we call selection forwarding to eliminate unneeded temporary result materialization during a query. Following this optimization, we implemented a lowering of all Voila-MLIR operations to MLIR's built-in dialects. For side-effect-free loops, we lower to the linalg dialect, and for for-loops with side effects such as hash table operations or selections, we choose the affine dialect. From thereon, we gradually applied optimizations such as loop fusion, tiling, unrolling, vectorization and parallelization to generate a program-variant specifically optimized to the underlying hardware. Due to the largely missing support for reduction optimization in MLIR, we extended most of the passes to also properly handle aggregations, a major factor in OLAP.

To get an in-depth understanding of the capabilities of our framework in a real-world context and to show that it is capable of generating variable behavior suited for query optimization, we evaluate our approach with the TPC-H benchmark. As

our framework is unable to represent the complete relational algebra to execute all kind of queries, we selected only a set of queries that represent typical OLAP work loads. Additionally, we transformed strings in the TPC-H data sets to an integer representation (i.e., dictionary compression) to execute queries on strings without direct support of a string representation. Furthermore, we denormalized the TPC-H database tables to support queries without the capability of joins. On this modified benchmark, we choose a subset of our variant space to explore in depth. We generated 120 variants and ran the queries on three different machines. In the following analysis, we found that we are able to positively influence the performance of the queries and most of our optimizations had a positive effect, with some of them demonstrating a trend towards the queries properties. Furthermore, we found that certain combinations of optimizations sum up to achieve peak performance and not all optimizations had the same effect and magnitude on all benchmark machines, even though they were all architecturally similar. Subsequently, we evaluated if we are able to outperform a state-of-the-art optimizing compiler with our optimizations and additionally identify the influence of our modifications on the data set on the run time of our queries using a baseline implementation of our algorithms in C++. We showed, that our approach can outperform modern compilers on Q3,Q6 and Q18. We also quantified the possible influence of our modifications to be only considerable for Q3, for the other queries it had a moderate influence with a speed-up of up to factor 3×. For Q9, our modifications even showed negative effects on the run time. To conclude our analysis, we compared our approach to state-of-the-art implementations of compiled and vectorized query engines, Typer and Tectorwise. Our experiments have shown that our approach can not yet keep up with the competitors, which is expected, considering that our main goal was to demonstrate the ability of our framework to generate queries with variability and achieve a speed-up compared to queries only optimized by a general purpose compiler.

7.2 Future Work

In this thesis, we laid the groundwork for a state-of-the-art adaptive reprogramming framework for heterogeneous hardware, but we are not there yet. In the following, we give an outlook for topics we want to investigate further.

Build the Bridge for Heterogeneous Adaptivity

Due to time constraints, we were unable to implement lowering to heterogeneous processors, such as GPUs. We want to change this to exploit an entire new branch of adaptivity with our framework. MLIR already supports transformations to GPU dialects, including optimizations to NVIDIA and AMD's own GPU programming frameworks. We seek to add the missing transformations for reduction transformations and make the lowering passes less strict to enable state-of-the-art heterogeneous adaptivity on the granularity of single operations.

Get Competitive

During our analysis, we found that optimizations are not always applied as expected. The main problem we identified, was the loop-fusion pass which blocked further

optimizations. Furthermore, we found that even though we partially come near to the performance of our selected competitors, we do not achieve a performance near to the bare-metal speed. This is also an issue when estimating the influence of optimizations to the overall performance of a query variant, as with a performance nearing the architectural limits of the hardware, the behavior of optimizations could change. To investigate if this is actually the case and foremost, to show that our approach is able to outperform other approaches, we want to optimize the framework further by correcting identified flaws and adding more optimizations to finally achieve bare-metal speed.

Runtime Adaptivity and Self Optimization Through Learning

With the current state of our framework, we are able to generate different variants of the same query, but there is no mechanism for any kind of directed selection of optimizations to generate high-performance variants. As this kind of adaptivity at runtime, dependent on the hardware and query characteristics, is the ultimate goal of adaptive reprogramming, we want to implement such mechanisms in the future. Therefore, we plan to use machine learning approaches and gradually refine the approach until we reach this goal.

Tackling the JIT-Compilation Overhead

One large drawback of query compilation is the compile-time overhead. For our approach, we determined that the overhead is well below the overhead of compilation from a high-level general purpose programming language, but still above the compile time of LLVM IR. To overcome this problem, we want to investigate several approaches. At first, we want to include the compile-time overhead of optimizations into the best variant calculation of queries and thus reduce compile time by weighting the optimization speed-up vs. the compile time. Another approach we want to investigate is the capability of our framework to emit LLVM IR that can be interpreted. From this capability, we want to switch between interpretation and compilation based query execution similar to [KLN18]. In a next step, we plan to use JIT-Compilation together with interpretation and switch seamlessly from interpretation to compiled query execution similar to the approach of [Gur+20], but using re-entrant code to allow switching at arbitrary points in the query pipeline.

Appendix

A.1 Modified TPC-H Queries

```
SELECT  l_returnflag,  l_linestatus,
SUM(l_quantity) AS sum_qty,
SUM(l_extendedprice) AS sum_base_price,
SUM(l_extendedprice*(1-l_discount)) AS sum_disc_price,
SUM(l_extendedprice*(1-l_discount)*(1+l_tax)) AS sum_charge,
AVG(l_quantity) AS avg_qty,
AVG(l_extendedprice) AS avg_price,
AVG(l_discount) AS avg_disc,
COUNT(*) AS count_order
FROM  lineitem
WHERE l_shipdate <=  [19981201 -DELTA]
GROUP BY l_returnflag, l_linestatus;
--ORDER BY omitted
```

Listing A.1: Q1

```
SELECT l_orderkey,
SUM(l_extendedprice*(1-l_discount)) AS revenue,
o_orderdate, o_shippriority
FROM customer_orders_lineitem
WHERE c_mktsegment = [SEGMENT]
AND c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND o_orderdate < [DATE]
AND l_shipdate > [DATE]
GROUP BY l_orderkey, o_orderdate, o_shippriority;
--ORDER BY omitted
```

Listing A.2: Q3

```
SELECT SUM(l_extendedprice*l_discount) AS revenue
FROM lineitem WHERE
l_shipdate >= [DATE]
AND l_shipdate < [DATE] + 10000
AND l_discount BETWEEN [DISCOUNT] - 0.01 AND [DISCOUNT] + 0.01
AND l_quantity < [QUANTITY];
```

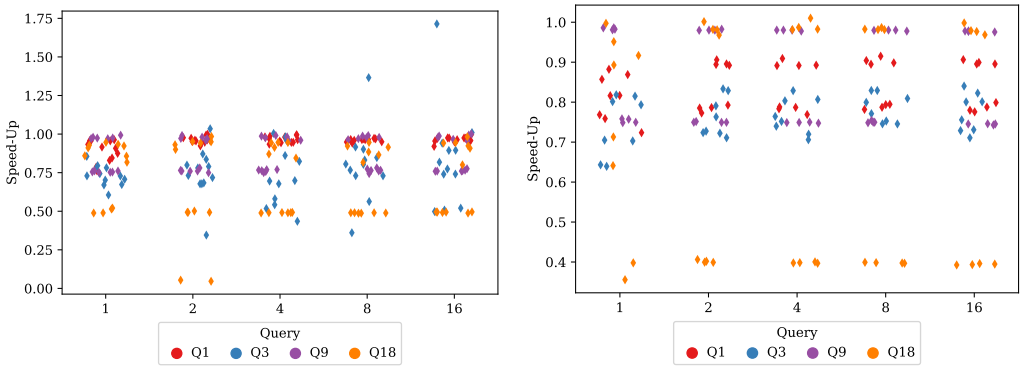
Listing A.3: Q6

```
SELECT c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice, SUM(l_quantity)
FROM customer_orders_lineitem
WHERE o_orderkey IN (
SELECT l_orderkey FROM lineitem GROUP BY l_orderkey
HAVING SUM(l_quantity) > [QUANTITY]
)
AND c_custkey = o_custkey
AND o_orderkey = l_orderkey
GROUP BY c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice;
--ORDER BY omitted
```

Listing A.4: Q18

A.1.1 Influence of Tiling on Other Optimizations

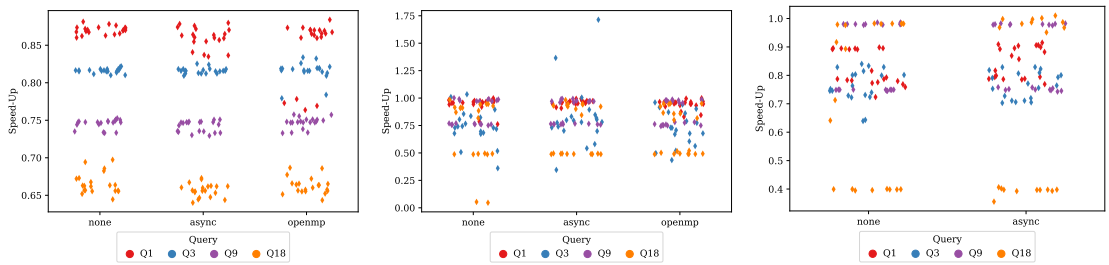
A.1.1.1 Vectorization



(a) Machine 2 **(b)** Machine 3

Figure A.1: Influence of Tiling on Vectorization

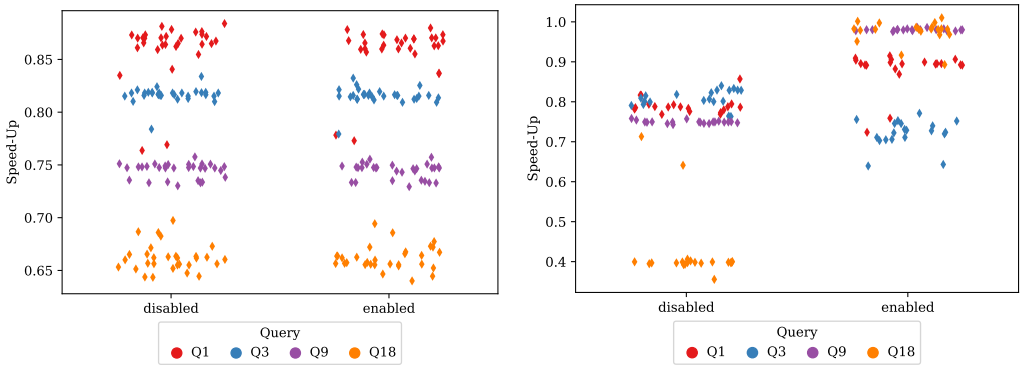
A.1.1.2 Parallelization



(a) Machine 1 **(b)** Machine 2 **(c)** Machine 3

Figure A.2: Influence of Tiling on Parallelization

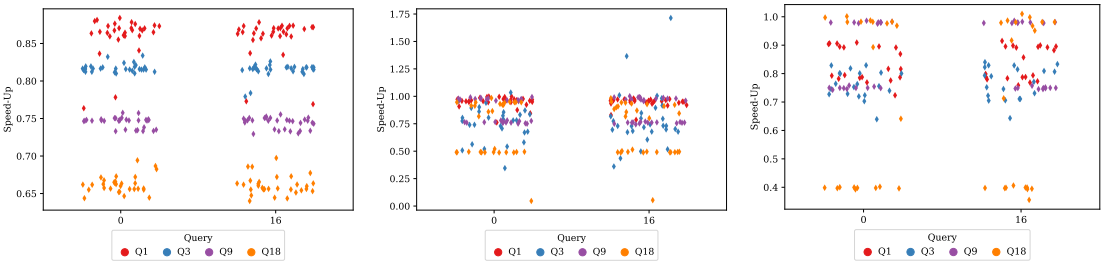
A.1.1.3 Selection Forwarding



(a) Machine 1 **(b)** Machine 3

Figure A.3: Influence of Tiling on Selection Forwarding

A.1.1.4 Unrolling

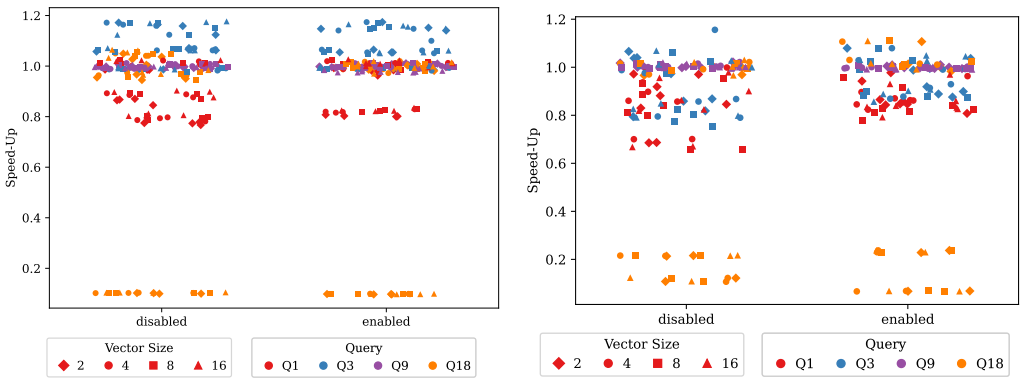


(a) Machine 1 **(b)** Machine 2 **(c)** Machine 3

Figure A.4: Influence of Tiling on Unrolling

A.1.2 Influence of Vectorization on Other Optimizations

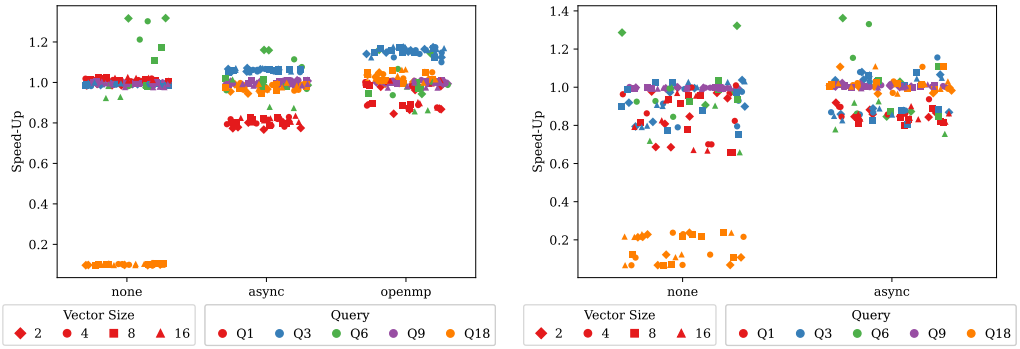
A.1.2.1 Tiling



(a) Machine 1 **(b)** Machine 3

Figure A.5: Influence of Vectorization on Tiling

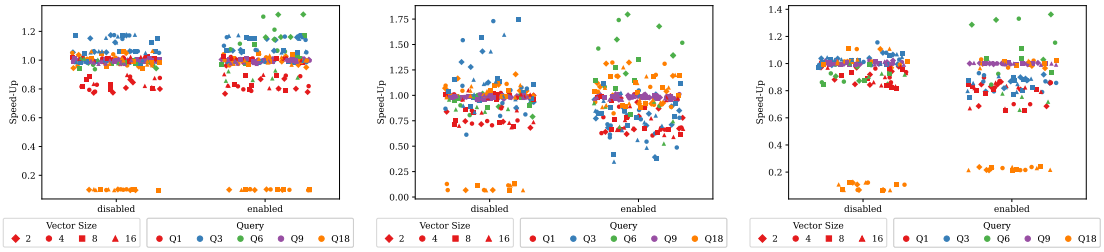
A.1.2.2 Parallelization



(a) Machine 1 **(b)** Machine 3

Figure A.6: Influence of Vectorization on Parallelization

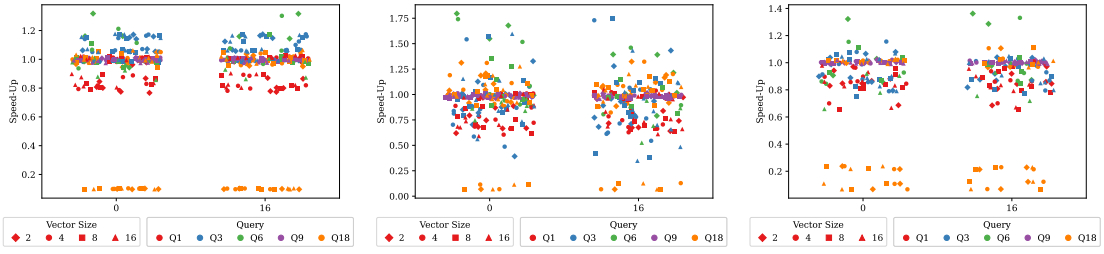
A.1.2.3 Selection Forwarding



(a) Machine 1 **(b)** Machine 2 **(c)** Machine 3

Figure A.7: Influence of Vectorization on Selection Forwarding

A.1.2.4 Unrolling

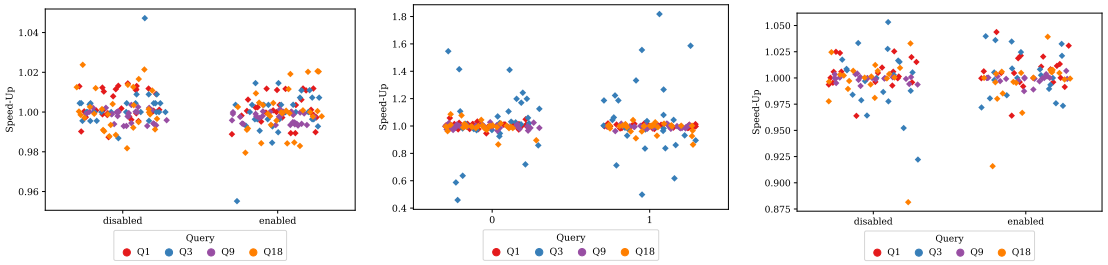


(a) Machine 1 **(b)** Machine 2 **(c)** Machine 3

Figure A.8: Influence of Vectorization on Selection Forwarding

A.1.3 Influence of Unrolling on Other Optimizations

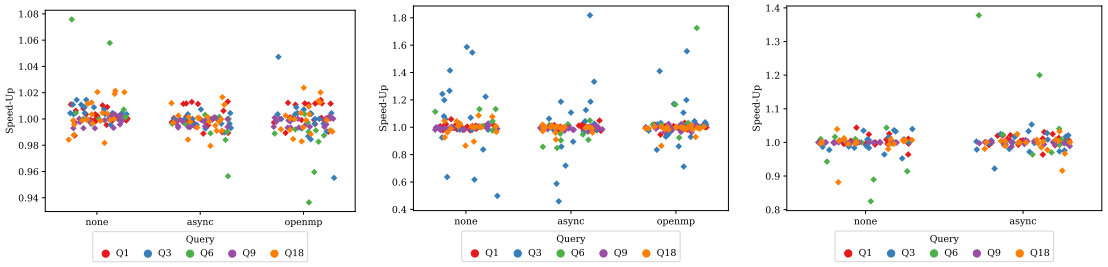
A.1.3.1 Tiling



(a) Machine 1 **(b)** Machine 2 **(c)** Machine 3

Figure A.9: Influence of Unrolling on Tiling

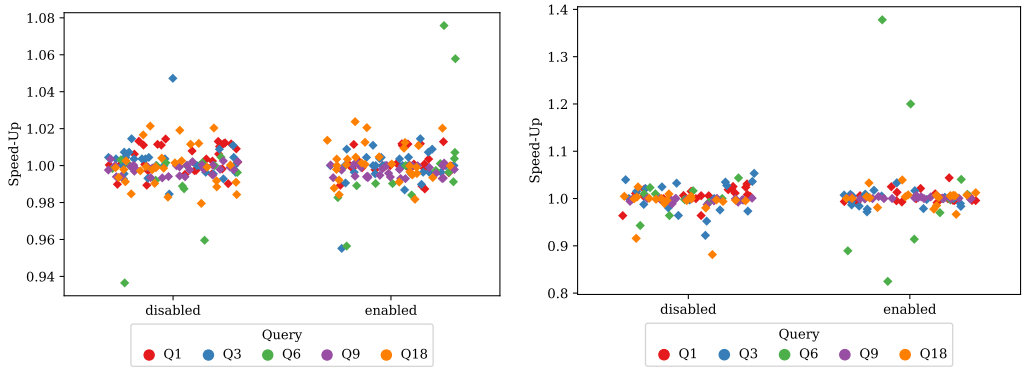
A.1.3.2 Parallelization



(a) Machine 1 **(b)** Machine 2 **(c)** Machine 3

Figure A.10: Influence of Unrolling on Parallelization

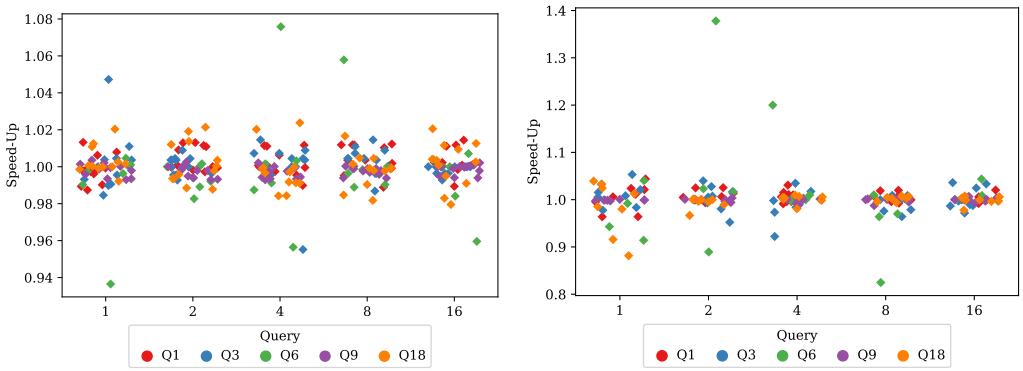
A.1.3.3 Selection Forwarding



(a) Machine 1 **(b)** Machine 3

Figure A.11: Influence of Unrolling on Selection Forwarding

A.1.3.4 Vectorization

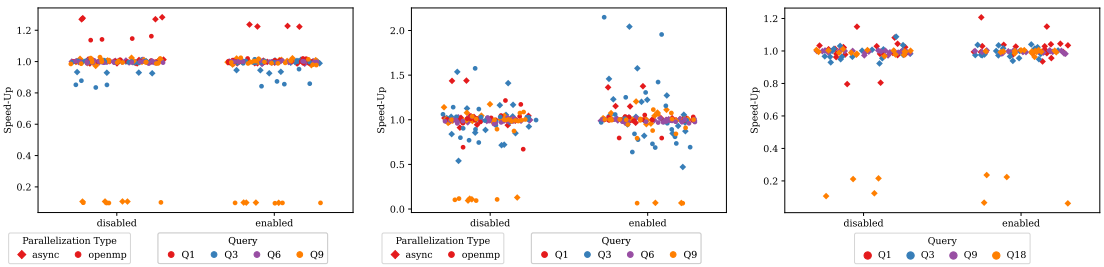


(a) Machine 1 **(b)** Machine 3

Figure A.12: Influence of Unrolling on Vectorization

A.1.4 Influence of Parallelization on Other Optimizations

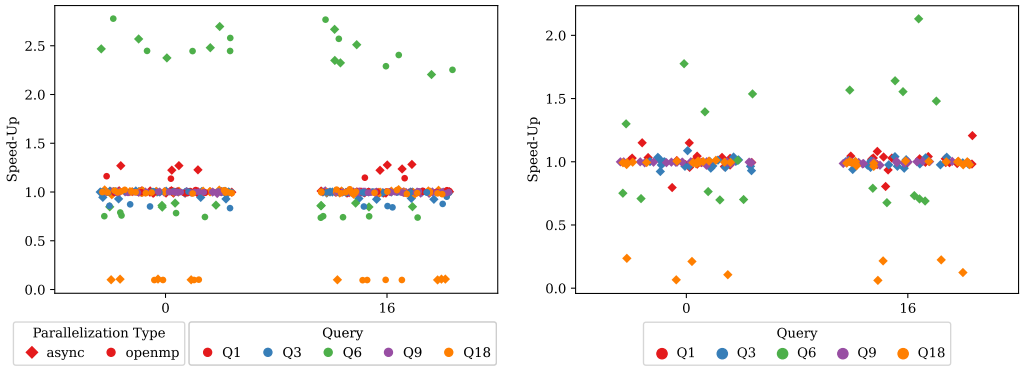
A.1.4.1 Tiling



(a) Machine 1 **(b)** Machine 2 **(c)** Machine 3

Figure A.13: Influence of Parallelization on Tiling

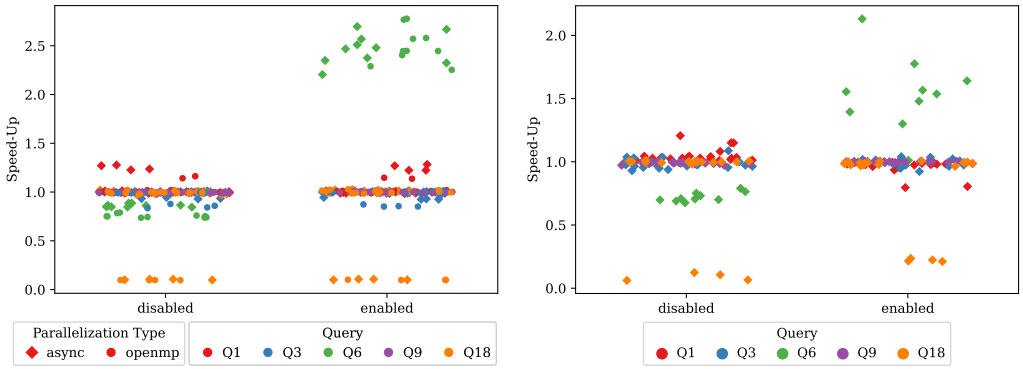
A.1.4.2 Unrolling



(a) Machine 1 **(b)** Machine 3

Figure A.14: Influence of Parallelization on Unrolling

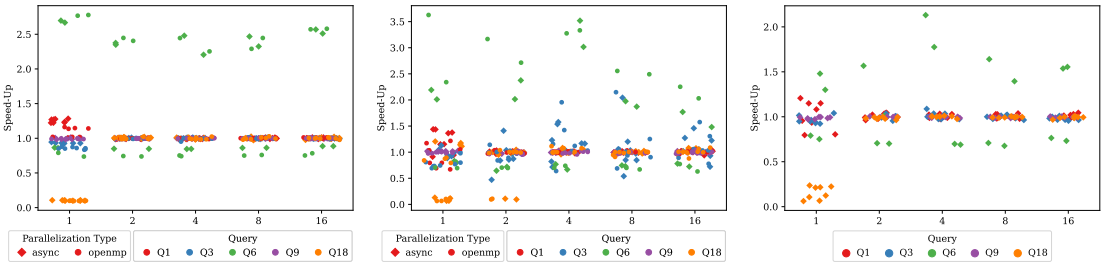
A.1.4.3 Selection Forwarding



(a) Machine 1 **(b)** Machine 3

Figure A.15: Influence of Parallelization on Selection Forwarding

A.1.4.4 Unrolling

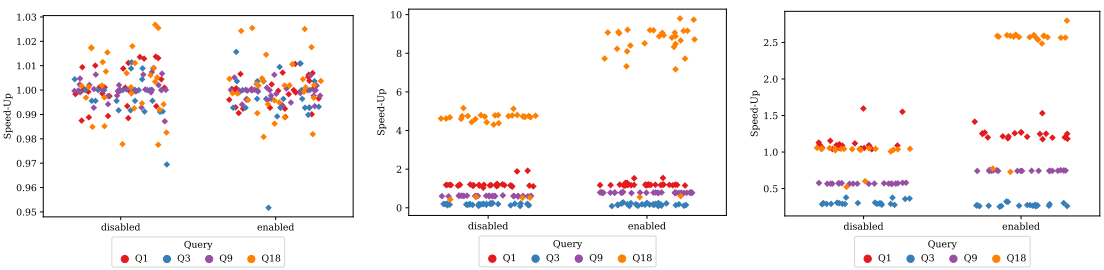


(a) Machine 1 **(b)** Machine 2 **(c)** Machine 3

Figure A.16: Influence of Parallelization on Vectorization

A.1.5 Influence of Selection Forwarding on Other Optimizations

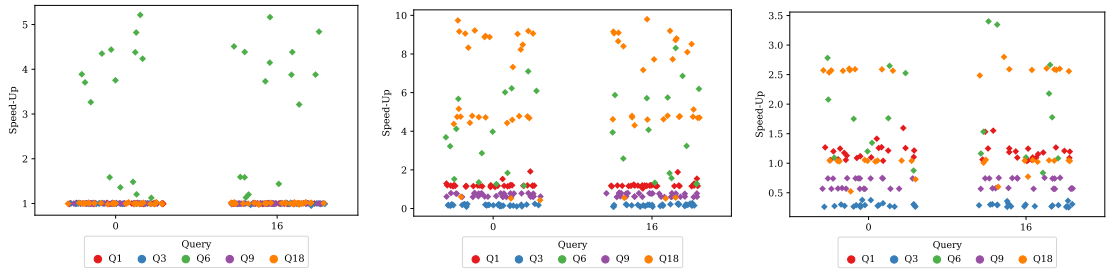
A.1.5.1 Tiling



(a) Machine 1 **(b)** Machine 2 **(c)** Machine 3

Figure A.17: Influence of Selection Forwarding on Tiling

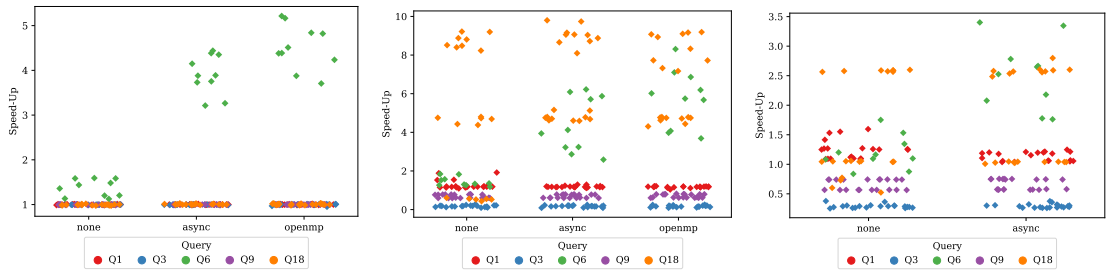
A.1.5.2 Unrolling



(a) Machine 1 **(b)** Machine 2 **(c)** Machine 3

Figure A.18: Influence of Selection Forwarding on Parallelization

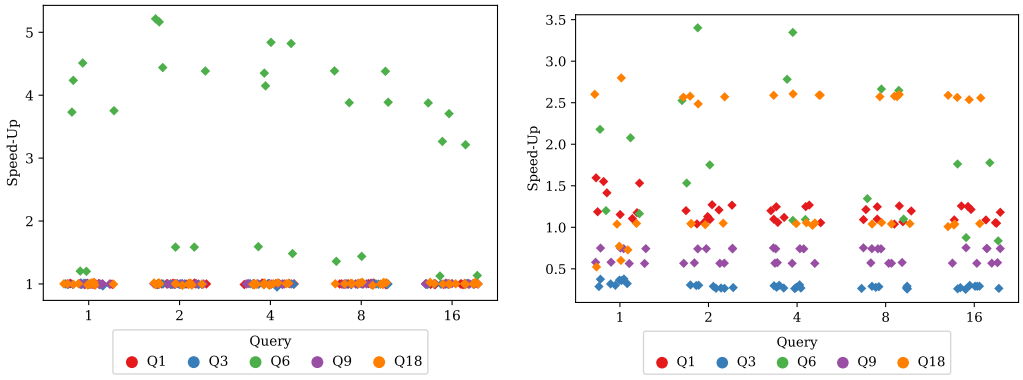
A.1.5.3 Parallelization



(a) Machine 1 **(b)** Machine 2 **(c)** Machine 3

Figure A.19: Influence of Selection Forwarding on Parallelization

A.1.5.4 Vectorization



(a) Machine 1 **(b)** Machine 3

Figure A.20: Influence of Selection Forwarding on Vectorization

A.1.6 Compile Times for Machines

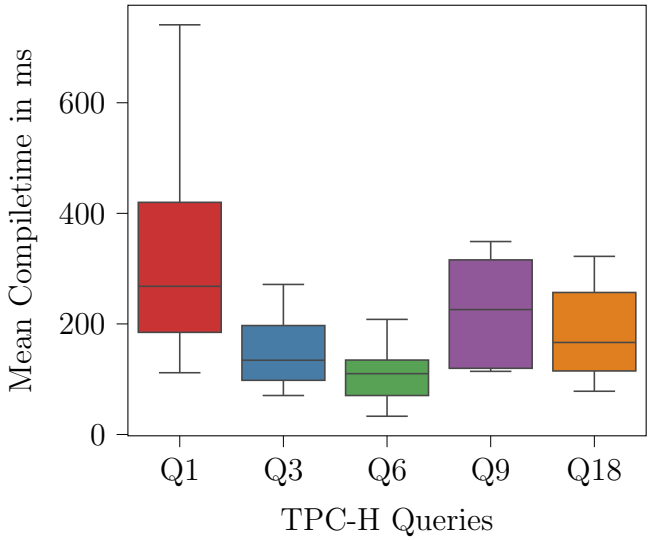


Figure A.21: Compile Times of Queries on Machine 1

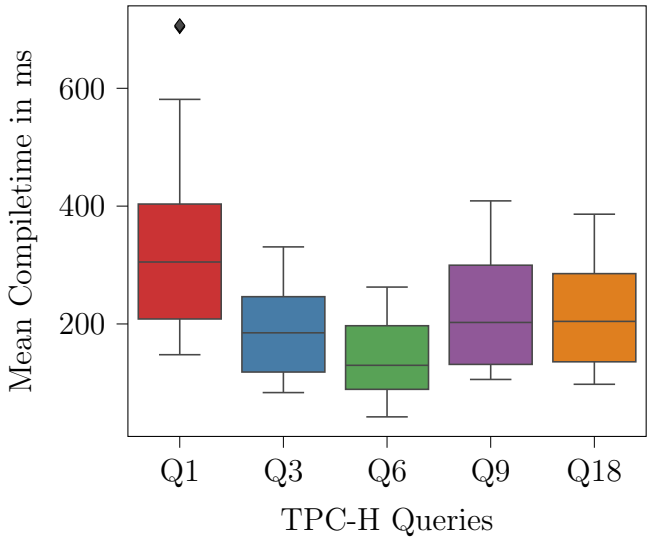


Figure A.22: Compile Times of Queries on Machine 3

Bibliography

[Amd67] Gene M Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the Spring Joint Computer Conference*. 1967, pp. 483–485. DOI: 10.1145/1465482.1465560 (cit. on p. 2).

[ASU13] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2013. ISBN: 0201100886 (cit. on p. 16).

[Bal+13] Cagri Balkesen et al. "Multi-core, main-memory joins: Sort vs. hash revisited". In: *Proceedings of the VLDB Endowment* 7.1 (2013), pp. 85–96. DOI: 10.14778/2732219.2732227 (cit. on pp. 2, 75).

[BBS13] David Broneske, Sebastian Breß, and Gunter Saake. "Database Scan Variants on Modern CPUs: A Performance Study". In: *In Memory Data Management and Analysis*. Springer, 2013, pp. 97–111. DOI: 10.1007/978-3-319-13960-9_8 (cit. on pp. 3, 62).

[BNE13] Peter Boncz, Thomas Neumann, and Orri Erling. "TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark". In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2013, pp. 61–76. DOI: 10.1007/978-3-319-04936-6_5 (cit. on pp. 60, 65).

[Bon+02] Peter Alexander Boncz et al. "Monet: A next-generation DBMS kernel for query-intensive applications". 2002 (cit. on pp. 14, 15).

[Bre+18] Sebastian Breß et al. "Generating custom code for efficient query execution on heterogeneous processors". In: *The VLDB Journal* 27.6 (2018), pp. 797–822. DOI: 10.1007/s00778-018-0512-y (cit. on pp. 23, 62).

[Bre14] Sebastian Breß. "The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS". In: *Datenbank-Spektrum* 14.3 (2014), pp. 199–209. DOI: 10.1007/s13222-014-0164-z (cit. on p. 3).

[Bro15] David Broneske. "Adaptive Reprogramming for Databases on Heterogeneous Processors". In: SIGMOD '15 PhD Symposium. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 51–55. DOI: 10.1145/2744680.2744685 (cit. on pp. 3, 73).

[BS17] David Broneske and Martin Schäler. “Single Instruction Multiple Data–Not Everything is a Nail for this Hammer”. In: *Proceedings of the International Workshop on Failed Aspirations in Database Systems (FADS)*. 2017 (cit. on pp. 3, 12).

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes. “MonetDB/X100: Hyper-Pipelining Query Execution”. In: *Second Biennial Conference on Innovative Data Systems Research (CIDR)*. Vol. 5. 2005, pp. 225–237 (cit. on pp. 13, 14).

[Cou14] Transaction Processing Performance Council. *TPC benchmark H (decision support)*. Tech. rep. 2014. URL: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf (cit. on p. 60).

[CSB11] Matthias Christen, Olaf Schenk, and Helmar Burkhart. “PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures”. In: *International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 676–687. DOI: 10.1109/IPDPS.2011.70 (cit. on p. 25).

[DeR69] Franklin Lewis DeRemer. *Practical translators for LR (k) languages*. Tech. rep. USA, 1969 (cit. on p. 29).

[DM82] Luis Damas and Robin Milner. “Principal Type-Schemes for Functional Programs”. In: *Symposium on Principles of Programming Languages (POPL)*. Ed. by Richard A. DeMillo. ACM Press, 1982, pp. 207–212. DOI: 10.1145/582153.582176 (cit. on p. 35).

[Dre+20] Markus Dreseler et al. “Quantifying TPC-H choke points and their optimizations”. In: *Proceedings of the VLDB Endowment* 13.8 (2020), pp. 1206–1220. DOI: 10.14778/3389133.3389138 (cit. on p. 60).

[DS13] Jonathan Dees and Peter Sanders. “Efficient many-core query execution in main memory column-stores”. In: *29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 350–361. DOI: 10.1109/ICDE.2013.6544838 (cit. on p. 2).

[Dur+19] Kayhan Dursun et al. “A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores.” In: *Proc. VLDB Endow.* 12.12 (2019), pp. 2218–2229. URL: http://dblp.uni-trier.de/db/journals/pvldb/pvldb12.html#DursunBCSG19;%20https://doi.org/10.14778/3352063.3352137;%20https://www.bibsonomy.org/bibtex/2c6cd4627723e030a8af1df9bd8fcf620/dblp (cit. on p. 47).

[Esm+11] Hadi Esmaeilzadeh et al. “Dark Silicon and the End of Multicore Scaling”. In: *38th Annual International Symposium on Computer Architecture (ISCA)*. Ed. by Ravi Iyer, Qing Yang, and Antonio González. ACM, 2011, pp. 365–376. DOI: 10.1145/2000064.2000108 (cit. on p. 2).

[Fly66] Michael J. Flynn. “Very High-Speed Computing Systems”. In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909. DOI: 10.1109/PROC.1966.5273 (cit. on p. 7).

[FR96] Michael J Flynn and Kevin W Rudd. "Parallel Architectures". In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 67–70. DOI: 10.1145/234313.234345 (cit. on p. 8).

[GB21] Tim Gubner and Peter Boncz. "Charting the Design Space of Query Execution Using VOILA". In: *Proceedings of the VLDB Endowment* 14.6 (2021), pp. 1067–1079. DOI: 10.14778/3447689.3447709 (cit. on pp. 4, 15, 27).

[Gra+12] Scott Grauer-Gray et al. "Auto-tuning a High-Level Language Targeted to GPU Codes". In: *Innovative Parallel Computing (InPar)*. 2012, pp. 1–10. DOI: 10.1109/InPar.2012.6339595 (cit. on p. 24).

[Gra90] Goetz Graefe. "Encapsulation of Parallelism in the Volcano Query Processing System". In: *SIGMOD Record* 19.2 (1990), pp. 102–111. DOI: 10.1145/93605.98720 (cit. on p. 13).

[Gru+20] Philipp M. Grulich et al. "Grizzly: Efficient Stream Processing Through Adaptive Query Compilation". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 2487–2503. DOI: 10.1145/3318464.3389739 (cit. on p. 24).

[Gur+20] Bala Gurumurthy et al. "When Vectorwise Meets Hyper, Pipeline Breakers Become the Moderator". In: *Eleventh International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)* (2020) (cit. on p. 85).

[Hag+20] Bastian Hagedorn et al. "Achieving High-Performance the Functional Way: A Functional Pearl on Expressing High-Performance Optimizations as Rewrite Strategies". In: *Proceedings of the ACM on Programming Languages* 4.ICFP (2020). DOI: 10.1145/3408974 (cit. on p. 25).

[He+08] Bingsheng He et al. "Relational Joins on Graphics Processors". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 511–524. DOI: 10.1145/1376616.1376670 (cit. on p. 3).

[He+09] Bingsheng He et al. "Relational query coprocessing on graphics processors". In: *ACM Transactions on Database Systems (TODS)* 34.4 (2009), pp. 1–39. DOI: 10.1145/1620585.1620588 (cit. on p. 3).

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X (cit. on p. 62).

[HS17] Emna Hammami and Yosr Slama. "An Overview on Loop Tiling Techniques for Code Generation". In: *14th International Conference on Computer Systems and Applications (AICCSA)*. IEEE/ACS, 2017, pp. 280–287. DOI: 10.1109/AICCSA.2017.168 (cit. on pp. 48, 68).

[Kar+14] Tomas Karnagel et al. "Heterogeneity-Aware Operator Placement in Column-Store DBMS." In: *Datenbank-Spektrum* 14.3 (2014), pp. 211–221. DOI: 10.1007/s13222-014-0167-9 (cit. on p. 3).

[Ker+18] Timo Kersten et al. “Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask”. In: *Proceedings of the VLDB Endowment* 11.13 (2018), pp. 2209–2222. DOI: 10.14778/3275366.3284966 (cit. on pp. 77, 80).

[KLN18] André Kohn, Viktor Leis, and Thomas Neumann. “Adaptive Execution of Compiled Queries”. In: *34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 197–208. DOI: 10.1109/ICDE.2018.00027 (cit. on p. 85).

[KN11] Alfons Kemper and Thomas Neumann. “HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots”. In: *27th International Conference on Data Engineering*. IEEE. 2011, pp. 195–206. DOI: 10.1109/ICDE.2011.5767867 (cit. on pp. 14, 76).

[KZ99] Stephen Kokoska and Daniel Zwillinger. *CRC Standard Probability and Statistics Tables and Formulae*. CRC Press, 1999. DOI: 10.1201/9780367802417 (cit. on p. 71).

[LA04] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665 (cit. on p. 15).

[Lat+21] Chris Lattner et al. “MLIR: Scaling Compiler Infrastructure for Domain Specific Computation”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE/ACM. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308 (cit. on pp. 4, 15).

[LP14] Yinan Li and Jignesh M. Patel. “WideTable: An Accelerator for Analytical Data Processing”. In: *Proceedings of the VLDB Endowment* 7.10 (2014), pp. 907–918. DOI: 10.14778/2732951.2732965 (cit. on pp. 61, 76).

[MBK00] Stefan Manegold, Peter A Boncz, and Martin L Kersten. “Optimizing database architecture for the new bottleneck: memory access”. In: *The VLDB Journal* 9.3 (2000), pp. 231–246. DOI: 10.1007/s007780000031 (cit. on p. 1).

[MHA18] Y Mulder, HP Hofstee, and Ir Z Al-Ars. “Feeding high-bandwidth streaming-based FPGA accelerators”. PhD thesis. Master’s Thesis, Delft University of Technology, 2018 (cit. on p. 1).

[Neu11] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware.” In: *Proc. VLDB Endow.* 4.9 (2011), pp. 539–550. URL: http://dblp.uni-trier.de/db/journals/pvldb/pvldb4.html#Neumann11 (cit. on p. 79).

[NZ08] Dorit Nuzman and Ayal Zaks. “Outer-Loop Vectorization: Revisited for Short SIMD Architectures”. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT ’08. Toronto, Ontario, Canada: Association for Computing Machinery, 2008, pp. 2–11. DOI: 10.1145/1454115.1454119 (cit. on p. 54).

[Pir+16] Holger Pirk et al. “Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware”. In: *Proceedings of the VLDB Endowment* 9.14 (2016), pp. 1707–1718. DOI: 10.14778/3007328.3007336 (cit. on pp. 15, 25).

[RBZ13] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. “Micro Adaptivity in Vectorwise”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. SIGMOD ’13. New York, New York, USA: Association for Computing Machinery, 2013, pp. 1231–1242. DOI: 10.1145/2463676.2465292 (cit. on pp. 23, 24, 62).

[Ros+15] Viktor Rosenfeld et al. “The Operator Variant Selection Problem on Heterogeneous Hardware.” In: *ADMS@VLDB*. Ed. by Rajesh Bordawekar et al. 2015, pp. 1–12 (cit. on p. 62).

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. “Global Value Numbers and Redundant Computations”. In: *Symposium on Principles of Programming Languages*. Ed. by Jeanne Ferrante and P. Mager. ACM Press, 1988, pp. 12–27. DOI: 10.1145/73560.73562 (cit. on p. 16).

[SHB17] Maxime Schmitt, Philippe Helluy, and Cidric Bastoul. “Adaptive Code Refinement: A Compiler Technique and Extensions to Generate Self-Tuning Applications”. In: *24th International Conference on High Performance Computing (HiPC)*. IEEE, 2017, pp. 172–181. DOI: 10.1109/HiPC.2017.00028 (cit. on p. 25).

[SS03] YN Srikant and Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2003. ISBN: 084931240X. DOI: 10.1201/9781315219967 (cit. on p. 45).

[Ste+15] Michel Steuwer et al. “Generating Performance Portable Code using Rewrite Rules. From High-Level Functional Expressions to High-Performance OpenCL Code”. In: *ACM SIGPLAN Notices* 50.9 (2015), pp. 205–217. DOI: 10.1145/2784731.2784754 (cit. on p. 25).

[Vog+18] Adrian Vogelsgesang et al. “Get Real: How Benchmarks Fail to Represent the Real World”. In: *Proceedings of the Workshop on Testing Database Systems*. 2018, pp. 1–6. DOI: 10.1145/3209950.3209952 (cit. on pp. 78, 80).

[Wil+09] Thomas Willhalm et al. “SIMD-Scan: Ultra Fast in-Memory Table Scan Using on-Chip Vector Processing Units”. In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 385–394. ISSN: 2150-8097. DOI: 10.14778/1687627.1687671 (cit. on p. 3).

[Zar05] Jerrold H Zar. “Spearman Rank Correlation: Overview”. In: *Encyclopedia of Biostatistics* 7 (2005). DOI: 10.1002/9781118445112.stat05964 (cit. on p. 71).

[ZVB12] Marcin Zukowski, Mark Van de Wiel, and Peter Boncz. “Vectorwise: A vectorized analytical DBMS”. In: *28th International Conference on Data Engineering*. IEEE. 2012, pp. 1349–1350. DOI: 10.1109/ICDE.2012.148 (cit. on pp. 14, 76).

I herewith assure that I wrote the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

Magdeburg, 11th March 2022