Generating Custom Code for Efficient Query Execution on Heterogeneous Processors

Sebastian Breß · Bastian Köcher · Henning Funke · Steffen Zeuch · Tilmann Rabl · Volker Markl

Abstract Processor manufacturers build increasingly specialized processors to mitigate the effects of the power wall in order to deliver improved performance. Currently, database engines have to be manually optimized for each processor which is a costly and error prone process. In this paper, we propose concepts to adapt to and to exploit the performance enhancements of modern processors automatically. Our core idea is to create processor-specific code variants and to learn a well-performing code variant for each processor. These code variants leverage various parallelization strategies and apply both generic and processor-specific code transformations. Our experimental results show that the performance of code variants may diverge up to two orders of magnitude. In order to achieve peak performance, we generate custom code for each processor. We show that our approach finds an efficient custom code variant for multi-core CPUs, GPUs, and MICs.

1 Introduction

Over the last decade, the main memory capacity has grown into the terabyte scale. Main memory databases exploit this trend in order to satisfy the ever-increasing performance demands. Thus, they store data primarily in main-memory to eliminate disk IO as the primary bottleneck [3,19]. As a result, memory access and data processing have become the new performance bottlenecks for in-memory data management [35]. Alleviating these bottlenecks has received significant attention in the database community and thus CPU and cache-efficient algorithms [1,5,35], data structures [1,33,49], and database systems [16,31,47] have been proposed.

Current designs of main-memory database systems assume that processors are homogeneous, i.e., with multiple identical processing cores. However, todays hardware vendors break with this paradigm of homogeneous multi-core processors in order to adhere to the fixed energy budget per chip [8]. This so-called *power wall* forces vendors to explore new processors to overcome the energy limitations [15]. As a consequence, hardware vendors integrate heterogeneous processor cores on the same chip, e.g., combining CPU and GPU cores as in Intel's processors with HD Graphics and AMD's Accelerated Processing Units (APUs). Another trend is *specialization*: processors are optimized for certain tasks, which already have become commodity in the form of *Graphics Processing Units* (GPUs), *Multiple Integrated Cores* (MICs), or *Field-Programmable Gate Arrays* (FPGAs). These accelerators promise large performance improvements because of their additional computational power and memory bandwidth. As a direct consequence of the power wall, current machines are built with a set of heterogeneous processors. Thus, from a processor design perspective, the *homogeneous many core age* ends [8,61]. The upcoming *heterogeneous many core age* provides an opportunity for database systems to embrace processor heterogeneity for peak performance.

Previous solutions either focused on generating highly efficient code for a single processor [40,58] or

Sebastian Breß
DFKI GmbH and TU Berlin E-mail: sebastian.bress@dfki.de

Bastian Köcher
TU Berlin E-mail: bastian.koecher@tu-berlin.de

Henning Funke
TU Dortmund E-mail: henning.funke@tu-dortmund.de

Steffen Zeuch
DFKI GmbH E-mail: steffen.zeuch@dfki.de

Tilmann Rabl
TU Berlin and DFKI GmbH E-mail: rabl@tu-berlin.de

Volker Markl
TU Berlin and DFKI GmbH E-mail: volker.markl@dfki.de

allowed database operators to run on multiple processors using the same operator code [24,64]. These code generation approaches were restricted to a single processor by generating low-level machine code (e.g., using LLVM [40]). In contrast, hardware-oblivious approaches experienced limited performance portability [51]. As of now, we need to manually adapt the database system to every new processor (e.g., for Intel's MIC architecture) for peak performance.

The methods we propose in this paper empower database systems to automatically generate efficient code for any processor without *any a priori* hardware knowledge, thus making database systems fit for the heterogeneous many-core age. To achieve this goal, we propose Hawk[1], a novel hardware-tailored code generator, which produces variants of generated code. By executing code variants of a compiled query, Hawk adapts to a wide range of different processors without any manual tuning. Hawk achieves low compilation times and runs queries on a wide range of processors.

In this paper, we make the following contributions:

1. We present the architecture of Hawk, a hardware-tailored code generator (cf. Section 3).
2. We introduce *pipeline programs*, a new form of a physical query plan. Pipeline programs store operations and global parameters of a pipeline and are the basis of our code generation (cf. Section 4).
3. We discuss the dimensions in which Hawk changes *pipeline programs* to tailor generated code to a processor (cf. Section 5).
4. We explain how Hawk produces target code from pipeline programs (cf. Section 6).
5. We present a learning strategy which automatically derives an efficient variant configuration for each processor. We incorporate the results into an optimizer for pipeline programs (cf. Section 7).
6. We show the potential of a database system that rewrites its code until it runs efficiently on the underlying heterogeneous processors (cf. Section 8).
7. We provide an implementation that leverages OpenCL as code compilation target to showcase Hawk's hardware-tailored code generation.

2 Background

In this section, we provide an overview of heterogeneous computing. We start by describing the processors that are currently supported by Hawk. Then, we discuss the challenge of abstraction from the hardware. Finally, we explain programming techniques in detail that are necessary to capture the intricacies of each processor.

[1] `https://github.com/TU-Berlin-DIMA/Hawk-VLDBJ`

2.1 Overview of Heterogeneous Processors

Multicore CPUs. CPUs are designed to achieve good performance for general purpose applications [25]. CPUs use large cores with complex control logic for features such as pipelining and out-of-order execution. They use caches to avoid access latencies to main memory and make use of multiple-cores to parallelize computations. Modern CPUs typically consist of up to tens of cores. Note that for parallelization, one thread is used per (physical) core. We refer to this one-to-one mapping of threads to cores as *coarse-grained parallelism.*

General Purpose GPUs. GPUs are designed for compute and data-intensive tasks that are highly parallelizable [25]. They consist of hundreds or more of very simple cores without complex control logic for features such as pipelining or out-of-order execution of individual instructions. In contrast to CPUs, GPUs process threads in blocks and execute them on so called streaming multiprocessors. GPUs leverage different techniques to improve efficiency compared to CPUs, most noteworthy by parallel computation and a memory bandwidth in the order of several hundred GB/s. Furthermore, GPUs hide stalls by memory access and costly computations by continuously switching context between thread blocks (thread block scheduling). Thus, GPUs hide memory accesses by other computations, instead of avoiding them like CPUs. Finally, modern GPUs offer several thousand light-weight cores and for each core, a number of threads must be spawned to fully exploit the thread block scheduling. Thus, GPUs need a N:1 mapping of threads to cores (so-called thread oversubscription), which we refer to as *fine-grained parallelism.*

MICs. MICs were designed to accelerate scientific, engineering, and graphics applications [46]. Their design is inspired by features of CPUs and GPUs. MICs consists of simple in-order cores with four hardware threads and a dedicated vector processing unit which supports 512-bit vector operations. Furthermore, they provide a memory bandwidth in the order of several hundred GB/s and avoid latencies to their memory by using caches. Overall, MICs provide up to hundreds of hardware threads. The original MIC (Knights Corner) is a co-processor connected to the CPU via a PCI-E bus. However, the new designs (Knights Landing) use the MIC as a special socket with the same access speed to memory as a regular CPU.

2.2 Hardware Abstraction without Regret

Programmers typically provide redundant implementations, i.e., one implementations for each processor, and work with a variety of APIs, e.g., OpenCL, CUDA,

```
__kernel void vecAdd(__global float* A, __global float* B, __global float* OUT){
 uint64_t ID = get_global_id(0);
 OUT[ID] = A[ID]+B[ID];
}
```

Fig. 1 Example OpenCL kernel for vector addition.

OpenMP, to leverage heterogeneous processors. This leads to high implementation effort, code complexity, and development cost.

The idea of *abstraction without regret* provides abstraction layers that simplify the code without causing a performance penalty [29]. One example is the LegoBase system [28], which compiles queries in a high-level language (Scala) to low-level languages (e.g., C). This allows to translate queries to optimized code just-in-time. Translating a general purpose language to co-processors requires automatic parallelization, which is not feasible for all programs. However, the problem is tractable for relational queries, which are the focus of this paper. We provide an abstraction layer (pipeline programs, cf. Section 4) that decouples the logic of queries from the programming concepts of the specific co-processor designs. This enables us to support different processors within a single code generator.

2.3 Programming Heterogeneous Processors

Next, we introduce the kernel programming model, as it is the basis for Hawk's code generation. Then, we present code transformations needed for programming heterogeneous processors.

2.3.1 Kernel Programming Model

Popular programming frameworks such as CUDA or the *Open Compute Language* (OpenCL) center around the kernel programming model, which has the goal to specify highly parallel computations [24].

Kernels are specialized functions which express computations with respect to individual data elements [24]. By launching a kernel with a specific number of threads, the execution is run in parallel and each thread performs work on its share of the input and writes to its share of the output. We show a simple OpenCL kernel for an addition of attributes A and B in Figure 1. Here, each thread gets its global id from the runtime and accesses the input and output arrays based on this id. The kernel programming model maps to different types of processors such as Multi-Core CPUs and GPUs [24]. On CPUs, multiple instances of a kernel for a range of input elements are spawned, potentially making use of SIMD (see coarse-grained parallelism in Section 2.1). On GPUs, threads are grouped in blocks and

```
int* output;
int count=0; /* num result tuples */
for(id=0;id<num_rows;id+=1){
  if(lo_quantity[id] < 25){
   output[count++] = lo_revenue[id];
  }
}
```

Branched Evaluation

```
int* output; int count=0;
bool result_increment;
for(id=0;id<num_rows;id+=1){
  result_increment=(lo_...[id] < 25);
  output[count] = lo_revenue[id];
  count+=result_increment;
}
```

Predicated Evaluation

Fig. 2 Example for branched and predicated evaluation.

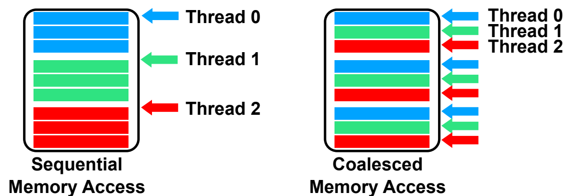

Fig. 3 Visualizing different memory access strategies.

these blocks are executed on a multi-processor (see fine-grained parallelism in Section 2.1). However, highly-tuned algorithms have fundamental differences that are not covered by current frameworks, such as granularity of parallelism and the memory access pattern [55].

In this paper, we use OpenCL as a target for our code generator, because it supports processors with different architectures including CPUs, GPUs, and MICs. However, all our concepts are also applicable to CUDA and other parallel programming frameworks.

2.3.2 Code Transformations

Research on heterogeneous processors showed that software predication and memory access pattern are the most impacting code transformations [24, 53].

Software Predication. Query predicates are often evaluated using if statements (e.g., if($x < 10$)). If the selectivity of a predicate is close to 50%, modern processors run in performance problems due to branch misprediction (CPUs, MICs) or branch divergence penalties (GPUs). Software predication is a technique to avoid such penalties. It transforms a control flow in a data flow by storing the result of a predicate evaluation in a variable, which we refer to as *result_increment* in this paper. After writing a database tuple to the output buffer, the result_increment variable is added to the tuple counter of the output buffer. In case the predicate matched, result_increment is one, and the tuple counter is incremented. Otherwise, if result_increment is zero, the output is overwritten by the next matching tuple. We illustrate the difference of branched evaluation and predicated evaluation in Figure 2. In Hawk, the *predication mode* is a parameter that specifies whether a program uses branched or predicated evaluation.

Memory Access Pattern. Different processors prefer different ways of accessing data in memory during

parallel processing [24, 55]. Using sequential access, each thread processes a continuous chunk of tuples. This access pattern is superior on CPUs. In contrast, using coalesced memory access, every thread reads a neighbored location relative to other threads. This access pattern represents the most efficient access pattern on GPUs. We illustrate both access patterns in Figure 3.

3 Hawk Architecture

In this section, we provide an overview of Hawk's architecture (cf. Figure 4) and Hawk's role in the process of executing an SQL query. The SQL parser translates queries into relational query plans. After that, the query optimizer rewrites the query plan by applying common optimizations to obtain a query execution plan. Examples for such optimizations are predicate push down, join ordering, and algorithm selection. We refer to the process of translating a database query into a query execution plan as *query translation*. On the next layer, Hawk provides the code generation backend. Thus, Hawk performs *query compilation*, which compiles query execution plans just-in-time into machine code of a target processor [40].

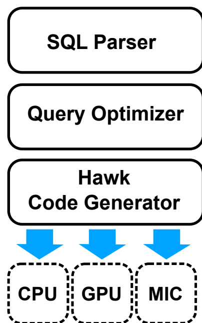

Fig. 4 Role of Hawk in DBMS architecture.

Hawk's key feature is the generation of efficient code for processors of different architectures. Our approach follows the principles of query compilation [40] as opposed to vector-at-a-time processing [6], because query compilation has the largest potential of applying processor-specific optimizations. Next, we discuss Hawks architecture and its hardware-tailored code generation.

3.1 Overview

In the following, we discuss Hawk's three-step compilation process: 1) query segmentation, 2) variant optimization, and 3) code generation (see Figure 5). In Table 1, we summarize the notions we introduce in this paper and that are required to describe this process. In general, Hawk receives a query plan as input and outputs optimized code for the underlying processors. This process centers around *pipelines*, i.e., non-blocking data flows. In particular, all operations in a pipeline are fused into one operator. The individual steps are as follows.

Query Segmentation. Hawk first segments query execution plans into pipelines using the produce/consume model [40] (Step (1) in Figure 5). During this step, Hawk creates for each pipeline a *pipeline program*, which is the intermediate representation for a pipeline. A pipeline program consists of simple operations such as loop, filter, and hash probe and establishes the start point for optimization and target code generation.

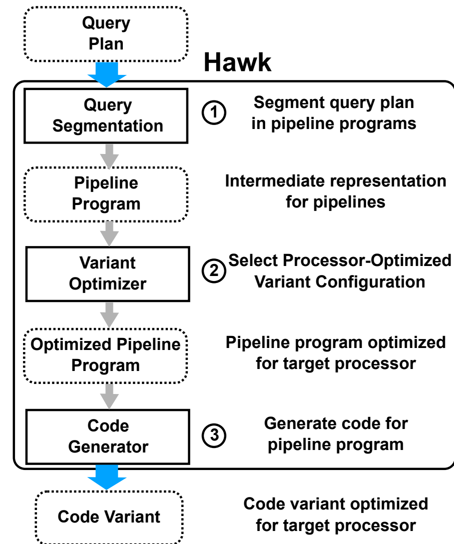

Fig. 5 Core concepts of Hawk and their role in the system.

Variant Optimizer. The initial pipeline program represents a hardware-oblivious blue print as a starting point for processor-specific optimizations. Based on that, Hawk produces hardware-tailored code by applying modifications to the pipeline programs. A *modification* is a change to a pipeline program, which conserves its semantic but changes the generated code (e.g., memory access pattern). A *variant configuration* captures all modifications of a pipeline program and thus provides a value for each supported modification. The set of all modifications defines the code generated by Hawk. The variant optimizer selects an efficient variant configuration for each pipeline program on a target processor (Step (2) in Figure 5). Note that Hawk automatically determines a variant configuration for each target processor without the need for manual tuning. In sum, Hawk applies the modifications specified in the variant configuration to the input pipeline program and returns an optimized pipeline program.

Code Generator. The code generator takes the optimized pipeline program as an input and produces the target code (Step (3) in Figure 5). We refer to the compilation result as *code variant*.

3.2 Overview of Hawk's Code Generation

We now discuss the modifications supported by Hawk and the code generation steps required for creating code variants in Section 3.2.1 and 3.2.2, respectively.

Term	Description
pipeline	a non-blocking data flow
pipeline program	intermediate representation for a pipeline, consists of operations such as loop, filter, and probe
modification	a change to a pipeline program, conserves the semantic but changes the generated code
variant configuration	provides value for each supported modification, defines the generated code
code variant	compilation result of a pipeline program

Table 1 Summary of important terms.

3.2.1 Dimensions of Code Modifications

Hawk captures hardware properties in a generic way. To this end, we structure modifications to pipeline programs in two dimensions: parallelization strategies and code transformations.

Parallelization Strategy. The parallelization strategy defines how parallelism is implemented in a pipeline program. The optimal strategy depends on the processor and has a strong impact on performance (cf. Section 5.1). Thus, a hardware-tailored code generator needs to cope with different parallelization strategies.

Code Transformations. Optimizing code for certain processors usually involves many low-level code transformations. For example, Hawk needs to decide on the optimal memory access pattern, the predication mode, and the hash table implementation [50]. Therefore, a hardware-tailored code generator should be flexible enough to apply a certain subset of code transformations to the generated code (cf. Section 5.2).

We discuss how Hawk applies various modifications to pipeline programs in Section 5. Furthermore, we describe how we keep modifications freely composable. For example, the parallelization strategy should not depend on the hash tables or memory access pattern used.

3.2.2 Code Variant Generation

Hawk generates code variants from pipeline programs, which allows Hawk to adapt to different heterogeneous processors. The code generation of a pipeline program proceeds in two steps: transformation and code generation. At first, Hawk inspects the variant configuration from the optimizer to determine which modifications need to be applied to the pipeline program. We illustrate this process in Figure 6.

In the *transformation step*, Hawk executes a sequence of transformation passes (Step (1) in Figure 6), where each pass reflects one modification to the pipeline program. These passes can adjust the pipeline program

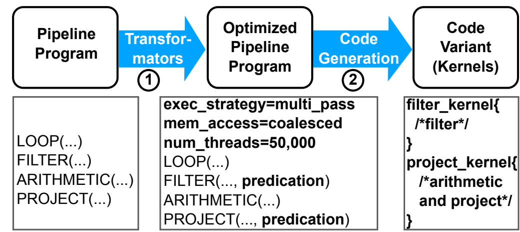

Fig. 6 Code Variant Generation.

in two ways. First, they set the global parameters of the pipeline program (e.g., the parallelization strategy). Second, they (re-)configure individual pipeline operations (e.g., set the hash table).

In the *code generation step*, we instantiate the selected parallelization strategy, which serves as a *fragment assembler* (Step (2) in Figure 6). The fragment assembler traverses the pipeline program and calls the code generator for each pipeline operation to obtain code fragments. These code fragments are combined by the fragment assembler, which generates one or more kernels depending on the parallelization strategy. We discuss target code generation in detail in Section 6.

Note that our code generation algorithm allows Hawk to freely combine modifications in the same pipeline program. For example, it is possible to generate a variant that uses coalesced memory access, software predication, with a fine-grained multi-pass parallelization strategy. Thus, it represents a flexible mechanism, which keeps the possible modifications freely composable.

4 Intermediate Representation

In this section, we introduce *pipeline programs*, Hawk's novel intermediate representation. Pipeline programs define the semantics of a pipeline and serves as a basis for query transformation and code generation. By transforming a pipeline program, Hawk generates different code variants, which is the key requirement for a hardware-tailored code generator. First, we discuss the general design of pipeline operations, which are the building blocks of pipeline programs. Second, we provide an overview of the pipeline operations supported in Hawk. Third, we show how Hawk translates relational operators to pipeline programs.

4.1 Design of Pipeline Operations

The generated code of pipeline operations depends on their parameter settings. In Hawk, pipeline operations accept two categories of parameters:

1. **Regular parameters:** These parameters encode the semantics of the operation, such as the table scanned or the filter predicates applied. We format these parameters *italic*.
2. **Code generation modes:** These parameters define which code variant is generated by the operation, such as the hash table implementation used. We format these parameters **bold**.

Hawk currently uses the following code generation modes, which are sufficient to cover all code transformations supported (cf. Section 3.2.1). Note that, a change of the code generation mode modifies the target code without affecting the semantic.

1. *Predication Mode* **m**: This parameter defines how filter conditions are evaluated, either by using an if-statement or by using software predication.
2. *Hash Table* **h**: This parameter defines the hash table implementation used. Hawk supports hash tables based on linear probing and Cuckoo hashing.
3. *Hash Table Parameters* **p**: This parameter defines specific parameters of a hash table, e.g., Cuckoo hashing requires the number of hash functions used.
4. *Element Access Offset* **o**: This parameter defines an offset relative to the current tuple position. It is required for transformations such as loop unrolling.

Furthermore, pipeline programs contain *global parameters* in addition to pipeline operations. These parameters are related to the whole pipeline program, such as the parallelization strategy or the number of threads.

In the next sections, we define pipeline operations as a central building block (cf. Section 4.2) and code generation rules for relational operators (cf. Section 4.3).

4.2 Overview of Pipeline Operations

In the following, we provide an overview of available pipeline operations for pipeline programs in Hawk.

LOOP(T; **step**, **s**, **e**). LOOP iterates over all input tuples of a table T and makes them available for following operations using a loop increment of **step**, and a loop start index **s** and end index **e**. Note that every valid pipeline program needs to have at least one LOOP statement as first operation. Consecutive LOOP operations in the same pipeline program result in nested for loops in the generated code. For instance, two LOOP operations perform a nested loop join.

PROJECT(A; **m**, **o**). PROJECT materializes tuples to the output relation projecting attributes of attribute set A. Thus, no operation may succeed a PROJECT operation in a valid pipeline program. The code generation needs to take two parameters into account: the predication mode **m** and the element access offset **o**. The predication mode is needed because the code for materializing the result depends on it (cf. Section 6.2.1).

FILTER(F_σ; **m**, **o**). FILTER selects input tuples that fulfill condition F_σ and passes them to the next operation. For code generation, FILTER requires a predication mode **m** and an element access offset **o**.

HASH_PUT(A; **h**, **p**). HASH_PUT inserts tuples into a hash table for attribute set A using hash table **h** with parameters **p**. Note that HASH_PUT is a pipeline breaking primitive. Thus, the next operation in the pipeline program must be a PROJECT operation, which writes the result and ends the pipeline program.

HASH_PROBE(A, f_{probe}, F_σ; **h**, **p**, **m**, **o**). The HASH_PROBE performs a lookup for each input tuple in a hash table for attribute set A and passes matching tuples to the next operator. In case the query provides an (optional) arbitrary filter condition F_σ, the HASH_PROBE passes only tuples to the next operator that meet the condition. In general, HASH_PROBE evaluates join conditions of the form $f_{probe} \wedge F_\sigma$. f_{probe} is a conjunction of equal or unequal expressions applied during the lookup in the hash table and F_σ is an arbitrary filter condition. F_σ is required to correctly support semi joins with multiple join conditions. Consecutive HASH_PROBE operations will be nested into each other. In case the build attribute is not guaranteed to be unique, the HASH_PROBE will loop over all matching entries of the hash table for the current tuple. The HASH_PROBE uses hash table **h** with parameters **p**, predication mode **m**, and element access offset **o**.

ARITHMETIC(f; **o**). ARITHMETIC performs a computation $f : A \times B \rightarrow C$ of attributes A, B, C using element access offset **o**. Note that we perform more complex computations by consecutive ARITHMETIC operations, which refer to attributes computed earlier in the pipeline program.

HASH_AGGREGATE(G,F; **h**, **p**, **m**, **o**). The HASH_AGGREGATE performs an aggregation with grouping attributes G using the aggregation expression $F = (f_1, f_2, \cdots, f_n)$. Each f_i consists of an aggregation function on an atomic attribute reference. Thus, Hawk needs to compute arithmetic expressions by ARITHMETIC operations that precede the aggregation. The generated code uses hash table **h** with parameters **p**, predication mode **m**, and element access offset **o**.

AGGREGATE(F; **m**, **o**). The AGGREGATE operation handles the special case of non-grouping aggregations, where Hawk directly aggregates into a local variable instead of a hash table. AGGREGATE evaluates the aggregation expression $F = (f_1, f_2, \cdots, f_n)$. The generated code depends on the predication mode **m** and element access offset **o**.

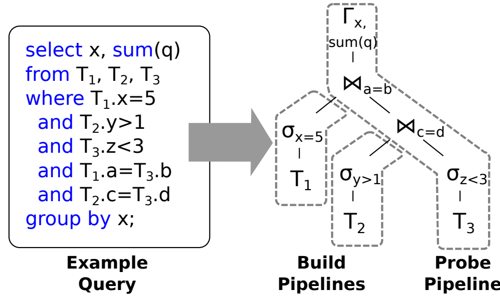

Fig. 7 Example for produce/consume model. We segment the query plan into three pipelines, two for building join hash tables, and one for probing both join hash tables.

4.3 Translating Relational Algebra to Pipeline Programs

Next, we show how Hawk translates relational database operations into pipeline programs. We build on the produce/consume model for code generation [40], as it fuses all operations in the same pipeline. Each operator provides a produce and a consume function. The *produce* function traverses the query plan top down from the root operator and creates a new pipeline for every pipeline breaking operator. If produce reaches a scan, it calls the consume function of succeeding operators bottom up and generates the code for each operator in the current pipeline. After that, we generate the code for the next pipeline. Thus, the produce functions essentially segment the query plan into pipelines, whereas the consume functions fill the pipelines with operators and generate the code. We illustrate the query segmentation of a query into pipelines in Figure 7. The query contains two hash joins, which results in a new pipeline for each hash table build and one probe pipeline.

In the following, we present the translation of each relational operator into pipeline programs by Hawk.

Scan(T, F_σ). The scan operator iterates over all tuples of a table T and passes all tuples, which fulfill the selection condition F_σ, to the next operator. Therefore, Hawk first inserts a LOOP operation into the pipeline program, followed by a FILTER operation:

LOOP(T; **step=1**, **s=0**, **e=numTuples(**T**)**)
FILTER(F_σ; **m=branched**, **o=0**)

The scan is a non-pipeline breaking operation, which continues the pipeline by notifying it's parent operator.

Projection(A). Projections either reference attributes or contain computational expressions (X+Y). Let $K \subseteq A$ be the subset of attribute references from A and let $F \subseteq A$ be expressions from A ($A = K \cup F$). If F is not empty, we generate for each $f \in F$ a set of ALGEBRA operations to compute expression f, assuming f is computable by arithmetic operations $f_1 \cdots f_n$:

ARITHMETIC(f_1; **o=0**) .. ARITHMETIC(f_n; **o=0**)

We denote the set of atomic attribute references to computed attributes by F'. After Hawk processed all computational expressions F, it generates the final PROJECT, consisting of the attributes from K and F':

PROJECT($K \cup F'$; **m=branched**, **o=0**)

Join(T_1, T_2, F_σ). We consider two implementations for joins, i.e., the nested-loop join, which is capable of handling any join conditions, and the hash join.

Nested-loop join. Hawk implements a *nested-loop join* by traversing the left and right sub-trees. Each scan operator adds its LOOP operation to the pipeline program, which creates a nested loop for each scan in the plan. Then, Hawk adds a FILTER operation that evaluates the join condition F_σ.

LOOP(T_1; **step=1**, **s=0**, **e=numTuples(**T_1**)**)
LOOP(T_2; **step=1**, **s=0**, **e=numTuples(**T_2**)**)
FILTER(F_σ; **m**, **o=0**)

Hash join. Next, we present the translation scheme for *hash joins*, which consist of two phases: build and probe. In the build phase, Hawk creates a hash table on the intermediate result of the left sub-tree. Thus, a hash join first introduces a new pipeline program P_{build}. After that, Hawk traverses the left sub-tree down (using the produce function) to add all operations of the left sub-tree to the pipeline program P_{build}. Then, Hawk adds the HASH_PUT and PROJECT operations to P_{build}, compiles, and executes P_{build}. Note that attribute set J contains all attributes required by the probe pipeline program, which is passed to PROJECT:

HASH_PUT(A; **h**, **p**)
PROJECT(J; **m=branched**, **o=0**)

In the probe phase, Hawk traverses the right-subtree and adds operations to the current pipeline program (P_{probe}). Then, Hawk adds the HASH_PROBE to P_{probe}:

HASH_PROBE(A, f_{probe}, F_σ; **h**, **p**, **m**, **o**)

As the probe is a not a pipeline breaker, Hawk calls the consume function of the parent operator, which adds it's operations to the current pipeline program.

Aggregation(G,F). Hawk handles aggregations with grouping attributes G using the aggregation expression $F = (f_1, f_2, \cdots, f_n)$ as follows. Each f_i is either an aggregation function on a single attribute (e.g., SUM(A)), or it contains an expression (e.g., SUM(A+B)). In case of an expression, Hawk adds ARITHMETIC operations to the pipeline program in the same way as in the relational projection. If G is not empty, Hawk adds the hash aggregate operation to the pipeline program:

HASH_AGGREGATE(G,F; ...)

Table 2 Pipeline programs created for example query plan from Figure 7. Each pipeline program belongs to one pipeline.

Build Pipeline 1	Build Pipeline 2	Probe Pipeline
LOOP(T_1, ..)	LOOP(T_2, ..)	LOOP(T_3, ..)
FILTER(x=5, ..)	FILTER(y>1, ..)	FILTER(z<3, ..)
HASH_PUT(a, ..)	HASH_PUT(b, ..)	HASH_PROBE(a=c, ..)
PROJECT(a, x, ..)	PROJECT(b, ..)	HASH_PROBE(b=d, ..)
		HASH_AGGREGATE(x, sum(q), ..)

In case G is empty, Hawk uses the more efficient operation for non-grouped aggregation, because it directly aggregates into a local variable instead of a hash table:

AGGREGATE(F; $\mathbf{m}$, $\mathbf{o}$)

Example. We illustrate the translation process in Hawk using the query in Figure 7 and show the pipeline programs produced in Table 2. The query contains two hash joins, leading to a query plan with three pipeline programs. The build pipeline programs iterate over their input tables (T_1 and T_2), apply their filters, insert the matching keys into a hash table, and materialize the result on the required attributes. The probe pipeline program iterates over table T_3, applies it's filter, probes the hash tables, and performs the aggregation.

5 Dimensions of Code Modifications

In Section 4, we discussed Hawk's intermediate representation – pipeline programs. In this section, we discuss the code modifications which are needed by a hardware-tailored code generator to produce efficient code. We also explain how Hawk transforms pipeline programs to produce hardware-tailored target code. We classify code modifications in two dimensions. We first discuss *parallelization strategies* in Section 5.1 and then explain several *code transformations* in Section 5.2.

5.1 Parallelization Strategies

One of the major parameters of pipeline programs is the parallelization strategy, which defines how we parallelize query execution. Efficient code generation for heterogeneous processors needs to trade-off two basic design dimensions: the degree of parallelism and synchronization overhead. As discussed in Section 2.1, different processors require different parallelization. CPUs require coarse-grained parallelism, i.e., spawning one thread per processor core. Co-processors such as GPUs require fine-grained parallelism (i.e., ten thousand and more threads) to exploit all available SIMD lanes in a streaming multi-processor and having enough thread blocks to hide memory latencies.

In the following, we discuss parallelization strategies that reflect different degrees of parallelism and synchronization cost. We differentiate between single-pass and multi-pass strategies, which reflect the number of times we need to read the input data. Single-pass strategies read data once but require synchronization. Multi-pass strategies avoid synchronization by incrementally building a data structure that contains unique write positions for all threads.

The parallelization strategy depends on the type of pipeline program. As a last step, a pipeline program either projects output tuples to a temporary relation (projection pipeline) or aggregates tuples (aggregation pipeline). Next, we present parallelization strategies for projection pipelines and aggregation pipelines.

5.1.1 Projection Pipelines

A projection pipeline is a pipeline program that materializes result tuples into an output buffer (i.e., does not perform aggregations). We show a simple query that creates a single projection pipeline in Listing 1. It consists of one filter predicate and projects three attributes. In the following, we show how a projection pipeline is implemented on CPU and GPUs/MICs.

Listing 1 Projection Query 1.

```
select lo_linenumber, lo_quantity,
    lo_revenue
from lineorder where lo_quantity<25;
```

Single-Pass Strategy. On CPUs, it is common to generate a single for-loop per pipeline [40]. This loop processes all input tuples and writes result tuples to the output buffers.. This single-pass strategy parallelizes query processing by concurrently executing the same pipeline on different chunks of the input relation [34]. Since one thread per core is launched, the single-pass strategy uses coarse-grained parallelism.

Multi-Pass Strategy. On processors with many light-weight cores (e.g., GPUs or MICs), the coarse-grained parallelization used by the single-pass strategy cannot utilize all cores (cf. Section 2.1). In this case, we apply a multi-pass strategy to achieve fine-grained parallelism. Algorithms that use fine-grained parallelism avoid latching at all cost and are typically multi-pass strategies, consisting of three phases [21,20]. In the first phase, the operator is executed and all matching tuples are marked in a flag array. In the second phase, per-thread write positions are computed using an exclusive prefix sum. Finally, the operator is executed again, but this time, the threads can utilize globally unique write positions to write their result. In Hawk, we implement this three-step processing technique in pipeline

Table 3 Impact of parallelization strategies on Projection Query 2 (cf. Listing 2). We show execution times in seconds.

Parallelization Strategies	**CPU**	**dGPU**	**MIC**
Single-Pass Strategy	0.05	3.12	0.56
Multi-Pass Strategy	0.14	0.02	0.18

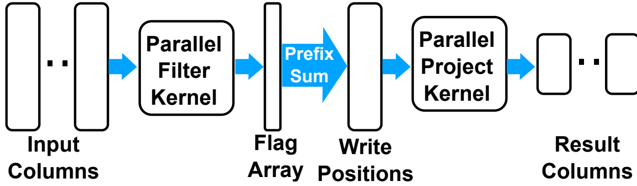

Fig. 8 Fine-grained parallelism. We generate two kernels that are executed massively parallel.

Listing 2 Projection Query 2.

```
select lo_linenumber, lo_quantity,
    lo_revenue
from lineorder where lo_quantity<25 and
    lo_discount<=3 and lo_discount>=1 and
    lo_revenue>4900000;
```

programs as follows. We generate two kernels, a filter and a projection kernel. In the first step, the *filter kernel* performs all operations that reduce the number of result tuples. These are essentially filter and hash probes (e.g., to conduct joins). All matching tuples are marked in a flag array. The second step computes the write positions for each thread by performing a prefix sum on the flag array. In the third step, the *projection kernel* repeats the hash probes to obtain the payload of matching join tuples. The projection kernel also performs arithmetic instructions and writes the result to the computed write positions. We illustrate this algorithm in Figure 8.

We illustrate the trade-off between the single-pass and the multi-pass strategy in Table 3, where we execute Projection Query 2 (cf. Listing 2) with the single-pass and the multi-pass strategy on different processors. We describe our detailed experimental setup in Section 8.1. The single-pass strategy outperforms the multi-pass strategy on CPUs by a factor of 2.8. The multi-pass strategy outperforms the single-pass strategy on a GPU by a factor of 148 and on a MIC by a factor of 3.19.

5.1.2 Aggregation Pipelines

An aggregation pipeline is a pipeline where the last operator is an aggregation operator. In this case, we materialize the result in a hash table and, therefore, we do not need to compute write positions in an output buffer. Depending on the number of result groups, we use different aggregation strategies.

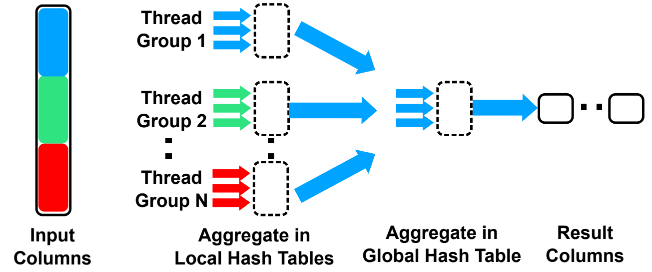

Fig. 9 Local hash table parallelization strategy for aggregation pipelines. For each of the N hash tables, M threads perform the aggregation.

Local Hash Table Aggregation. If we expect few result groups, we perform the aggregation in two steps [27,59]. First, we pre-aggregate the result in parallel in multiple local hash tables. Second, we merge the local hash tables into a global result hash table. We call this *local aggregation* and illustrate the principle in Figure 9. For each of the N hash tables, M threads perform the aggregation. The number of hash tables and threads per hash table are thus important tuning parameters (cf. Section 9). We synchronize concurrent operations on the aggregates using OpenCL's atomics.

Global Hash Table Aggregation. If we expect many result groups, we aggregate into a single global hash table [27,59]. In this case, synchronization overhead is small and cost for merging large partial results high. We refer to this as *global aggregation*, which is a special case of local aggregation with a single local hash table. Thus, we only need to tune the number of threads per hash table.

Parallelism. We implement *coarse-grained parallelism* by using the local hash table aggregation with one thread per hash table. Furthermore, we set the number of hash tables to the number of OpenCL compute units (e.g., the number of CPU cores). Each thread group is responsible for one hash table. Thus, if we increase the number of threads per thread group, we achieve *fine-grained parallelism*.

5.2 Code Transformations

In this section, we discuss how Hawk captures different hardware (and query) properties at the level of traditional code transformations. Hawk considers exchanging the memory access pattern, the predication mode, and the hash table implementation.

Adjusting the memory access pattern. The optimal memory access pattern is processor dependent [24, 55]. We show the performance impact of the memory access pattern in Table 4. On a CPU, sequential access outperforms coalesced memory access by a factor

Table 4 Impact of memory access pattern on Projection Query 2 (cf. Listing 2). We show execution times in seconds.

Access Pattern	CPU	dGPU	MIC
Sequential	0.14	0.04	0.18
Coalesced	0.22	0.02	0.18

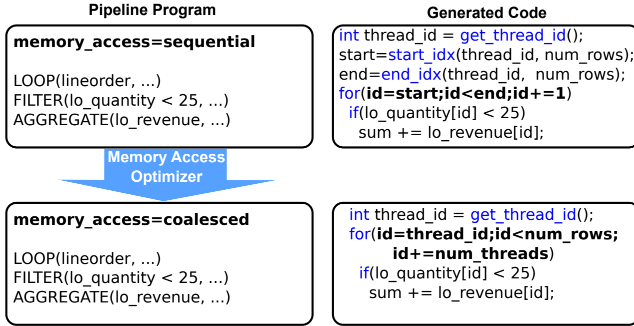

Fig. 10 Effect of memory access pattern on generated code.

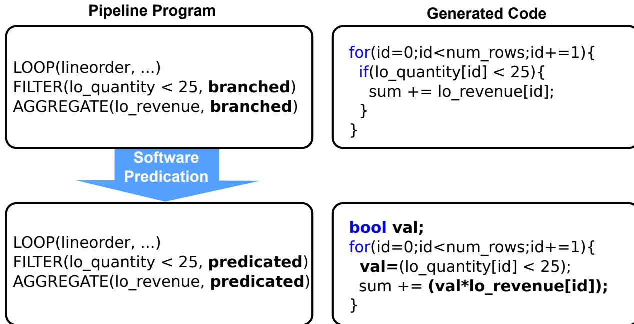

Fig. 11 Applying software predication to a pipeline program.

of 1.6. On a GPU, coalesced memory access outperforms sequential memory access by a factor of 1.8. On a MIC, we measure no significant difference between access patterns. In Hawk, we rewrite the memory access pattern in a pipeline program by setting the memory access property. We show the impact on the generated code in Figure 10.

Applying software predication. Software predication is a common technique to avoid branch misprediction penalties [11,53]. To support predication, each pipeline operation has a flag that determines whether code with branching (if statements) or with predication should be generated. In the predicated mode, the result of predicate evaluations is stored in a result value. This value is either added to the variable storing the result size (projection pipeline) or multiplied to the input values before an aggregation (aggregation pipeline). We illustrate the principle in Figure 11, where we apply predication to a simple aggregation pipeline. Note that all pipeline operations have to be in the same predication mode. Otherwise, an AGGREGATE or PROJECT operation after a FILTER operation would incorrectly

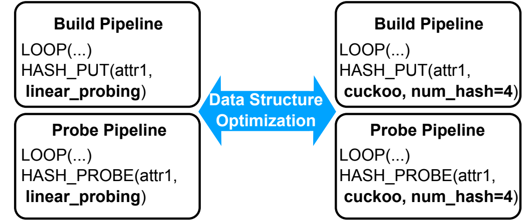

Fig. 12 Exchanging hash table implementations.

include filtered out tuples in the result computation. This is because the code templates of these operations depend on the predication mode (cf. Section 6.4).

Selecting hash table implementations. Besides a well-selected parallelization strategy, high performance implementations require optimized data structures. A prominent example in the database context are hash tables. In particular, different implementations are optimal for different data and query characteristics [50]. The Hawk code generator allows us to select the hash table implementation on a per-query basis.

In Hawk, we set the hash table implementation using a transformation pass during the transformation step of code generation (cf. Section 3.2.2). The transformation pass iterates over the pipeline program and configures each HASH_PUT and each HASH_PROBE operation with a hash table and its parameters, as we illustrate in Figure 12. For example, we exchange the collision resolution strategy by using Cuckoo hashing instead of linear probing (e.g., because we expect a sparse data distribution and want to achieve a high fill factor [50]). Cuckoo hashing uses n hash functions that provide for each key n possible insertion locations in the hash table. Thus, Cuckoo hashing avoids situations where a large number of entries of the hash table need to be inspected during a probe. However, Cuckoo hashing might require expensive rehashing if the insertion of a key fails. Note that we can also change the parametrization of a hash table, e.g., we can set the number of hash functions of Cuckoo hashing to tune performance.

The important constraint that has to be satisfied is that the build and probe pipeline operations need to work with the same hash table and same parametrization. This introduces a dependency between pipeline programs. Thus, the query processor needs to ensure that corresponding HASH_PUT and HASH_PROBE operations use the same data structure.

Other optimizations. We can also apply more complex code transformations, such as loop unrolling or vectorization in Hawk. We exemplary show how loop unrolling can be supported by pipeline programs in Figure 13. Loop unrolling affects the original pipeline program beyond the choice of code generation parameters.

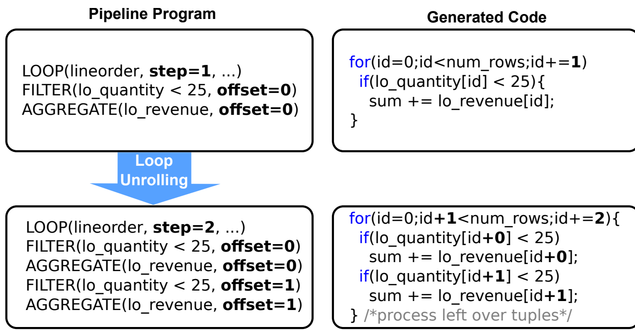

Fig. 13 Applying loop unrolling to a pipeline program.

Table 5 Query compilation times in milliseconds of a simple Projection Query (cf. Listing 2).

HyPer	Hawk (OpenCL)				
	CPU (Intel)	CPU (AMD)	MIC (Intel)	GPU (Nvidia)	GPU (AMD)
13	25.2	39.3	96.7	81.5	55.2

However, loop unrolling does not limit the combinations with other modifications.

In sum, a pipeline program is a highly flexible representation, which stores low-level code transformations that are hard to represent in a physical query plan.

6 Target Code Generation

In the previous section, we discussed different ways of how Hawk modifies pipeline programs to produce hardware-tailored code. In this section, we discuss the target code generation of Hawk for pipeline programs. We outline why we use OpenCL as target language and discuss how Hawk generates kernels by fragment generation and assembly. Then, we discuss how the code generator can be extended by new data structures and algorithms and present implementation details of Hawk.

6.1 Target Code: OpenCL Kernel

The drawback of generating high-level code is usually high compilation time [30, 40]. By compiling pipeline programs to OpenCL kernels, Hawk benefits from the JIT compilation capabilities and the performance portability of OpenCL. The latter allows Hawk to run any code variant on any OpenCL-capable processor.

In Table 5, we show query compilation times for a simple query (cf. Listing 2) for compiling OpenCL kernels for an Intel CPU, an AMD CPU, an Intel MIC, a NVIDIA GPU, and an AMD GPU. As reference, we also show the compilation time of HyPer [40] (v0.5-222-g04766a1), a state-of-the-art system for query compilation. Compiling OpenCL kernels for CPUs is in the same order of magnitude (slower by a factor of 2 to 3 for Intel and AMD OpenCL SDKs) as the LLVM IR query compilation used by HyPer [40]. Furthermore, we observe that compilation for GPUs and MICs is up to a factor of 4.3 to 7.4 slower compared to LLVM IR query compilation. The compilation times are consistently below 100ms. Thus, we conclude that query compilation using OpenCL is sufficiently efficient for compiling database queries to support interactive querying on GPUs and MICs. Note that the OpenCL compilation times can be further reduced. For example, we could disable certain optimization passes and trading off runtime and optimization time, similar to optimization levels in some commercial database engines.

6.2 Fragment Generation and Assembly

We now discuss how we generate code for projection and aggregation pipelines from pipeline programs. The code generation follows a two step approach: *fragment generation*, followed by *fragment assembly*.

6.2.1 Fragment Generation

A code fragment (in short *fragment*) consists of six segments: host variable declarations, host initialization code, host cleanup code, kernel variable declarations, kernel code top, and kernel code bottom. These fine-grained separations allow us to route fragments into different kernels. Each pipeline operation produces a fragment that implements its semantic. We retrieve the fragment for each pipeline operation to create all fragments. Each operation can generate code for any part in the target source code, e.g., body of the for-loop, declarations, or cleanup operations. Furthermore, the fragment produced by a pipeline operation depends on the code generation modes. These modes are special parameters, which define the code variant generated by the operation, but do not change the semantic. Code generation modes enable Hawk to adapt the fragment

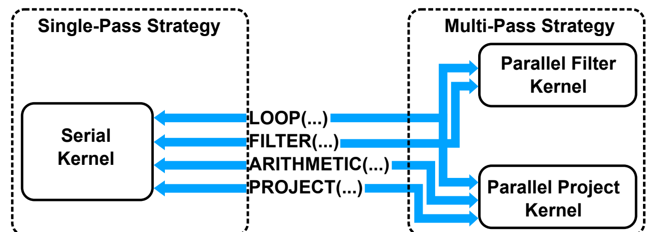

Fig. 14 Supporting multiple parallelization strategies. Each strategy acts as a fragment assembler for a pipeline program. A fragment assembler combines code fragments of each pipeline operation into one or more kernels.

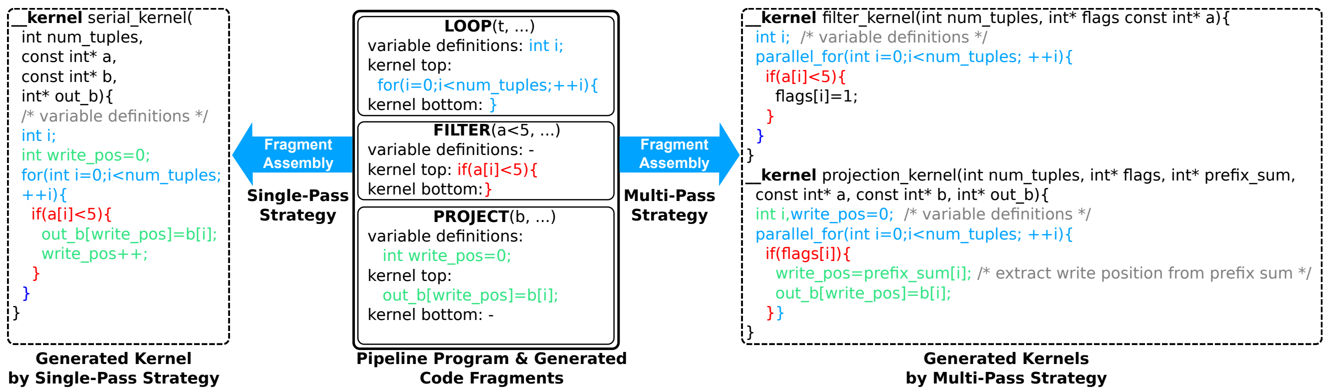

Fig. 15 Example for fragment generation and fragment assembly: Each pipeline operation generates fragments, which are then assembled to kernels. The single-pass strategy generates one kernel that includes all operations from the fragments. The multi-pass strategy generates a filter and a projection kernel which include different fragments.

by re-parameterizing the pipeline operations or global parameters of the pipeline program. Using this code generation approach, it is straightforward to create code variants of a pipeline program to adapt to the underlying hardware (cf. Section 6.4).

6.2.2 Fragment Assembly

We combine fragments by assembling them into a single fragment. Note that this *fragment assembly* is essentially a string concatenation of code segments. Our guiding idea is as follows. We provide a *fragment assembler* for pipeline-programs for each parallelization strategy. Each fragment assembler knows how many kernels are required for the strategy. The fragment assembler assigns the fragments, depending on the pipeline operation, to one or more kernels. We illustrate this process in Figure 14. For the single-pass strategy, all fragments belong to the same kernel. In contrast, the multi-pass strategy routes fragments from different pipeline operations to different kernels. Thus, a fragment can be part of multiple kernels, e.g., LOOP or HASH_PROBE.

For each kernel used by the parallelization strategy, the fragment assembler combines all fragments assigned to the kernel to a result fragment. We create the final kernel from this result fragment. Note that Hawk's code generator is conceptionally not limited to OpenCL kernels. Thus, Hawk could also produce code for frameworks such as CUDA. Since we implement parallelization strategies as fragment assemblers, we can apply different strategies to pipeline programs. Our design keeps the parallelization strategies composable with any other modification on the pipeline program.

6.3 Example: Fragment Generation and Assembly

We now present an example that illustrates the code generation process. Consider the query *select b from t where a<5*, which will result in a pipeline program with three operations: LOOP, FILTER, and PROJECT. We show the generated fragments of the pipeline operations in Figure 15. The generated fragments can add code to two parts of the kernel: the variable declaration and initialization code block, and the for-loop. Code can be inserted into a for-loop at two positions: at the top position we insert the actual code; at the bottom position we insert closing brackets and perform operations after an iteration, e.g., increasing counters. Generated code of succeeding operations is nested inside the brackets of previous operations. For example, the final projection is nested in the generated code of the filter operation.

6.4 Fragment Generation and Assembly Algorithms

We now introduce algorithms for fragment generation. We show pseudo code for each algorithm and highlight generated code by surrounding it with angle brackets and by coloring the background (<generated code>). We also highlight entry points for code templates of succeeding pipeline operations (<entry point>).

Loop. The LOOP operation generates code that iterates over every input tuple of a table in parallel. We can either iterate sequentially or interleaved over the tuples, which leads to sequential or coalesced memory access (cf. Listing 3). In case of sequential access, we compute the start and end offset of the partition that each thread processes. In case of coalesced access, each thread starts the iteration on its unique thread identifier and advances by adding the number of threads to the loop variable *id*.

Listing 3 Loop Fragment Generation:
LOOP(table, memory_access_pattern).

```
<thr_id = get_thread_id()>
if(memory_access_pattern==SEQUENTIAL){
  <start=start_idx(thr_id,num_rows)>
  <end=end_idx(thr_id,num_rows)>
  <for(id=start;id<end;id+=1){>
    <insert code of next operation>
  <}>
}else if(memory_access_pattern==COALESCED)
    {
  <for(id=thr_id;id<num_rows;id
  +=num_threads){>
    <insert code of next operation>
  <}>
}
```

Listing 4 Filter Fragment Generation:
FILTER(condition, predication_mode).

```
if(predication_mode==BRANCHED){
  <if(condition){>
    <insert code of next operation>
  <}>
}else if(predication_mode==PREDICATED){
  <result_increment=(condition)>
  <insert code of next operation>
}
```

Generated Code: <code>

Listing 5 Project Fragment Generation:
PROJECT(proj_attributes, predication_mode).

```
<declare variable write_pos=0>
for(attribute in proj_attributes){
  <copy value of attribute to result
  column at position write_pos>
}
if(predication_mode==BRANCHED){
  <write_pos++>
}else if(predication_mode==PREDICATED){
  <write_pos+=result_increment>
}
```

Listing 6 Join Fragment Generation:
HASH_BUILD(attr, hash_table) and
HASH_PROBE(attr, hash_table).

```
HASH_PUT(A;h,p){
/* declare and init ht in host code */
 <ht = createHashTable(A, h,p);>
/* inside the for loop for each tuple t */
 <ht.insert(π_A(t));>
 <insert code of next operation>
}
HASH_PROBE(A, f, F; h,p){
/* get hash table of previous pipeline */
 <ht = getHashTable(A,h,p);>
/* inside the for-loop for each tuple t */
 <res = ht.lookup(t.A, f);>
 <if(res.match && F(t,res.payload)){>
  <tuple t'=res.payload;>
    <insert code of next operation>
 <}>
}
```

Listing 7 Aggregate Fragment Generation:
AGGREGATE(attr, SUM, predication_mode).

```
<declare variable aggregate=0>
if(predication_mode==BRANCHED){
  <aggregate+=attr[id]>
}else if(predication_mode==PREDICATED){
  <aggregate+=(attr[id]*result_increment)>
}
```

Generated Code: <code>

Filter. The FILTER operation generates code that evaluates a selection predicate. It either generates an if-statement (no predication) or stores the result of the predicate evaluation in the variable *result_increment* (predication), as we illustrate in Listing 4.

Project. The PROJECT operation generates code that copies the values of each projected attribute and writes them to the write position *write_pos* in the projection attribute's output array (cf. Listing 5). The generated code depends on the predication mode. If predication is disabled, we know the tuple passed all previous filters. Thus, we increment the write position after writing the tuple into the output buffer. If predication is enabled, we always write the result tuple but add the variable *result_increment* to *write_pos*. If the tuple passed all previous filters, *result_increment* is one and the write position is advanced by one row. In case the tuple did not match all filters, *result_increment* is zero and the write position is not changed, which discards the current tuple.

Hash. The HASH_PUT and HASH_PROBE operations generate code that insert/lookup tuples into/from a certain hash table (cf. Listing 6). HASH_PROBE first probes the hash table using attributes A and then applies the generic filter condition F to the joined tuple. The code of consecutive HASH_PROBE operations is nested into each other. We omit the detailed code for the sake of brevity.

Aggregate. The AGGREGATE operation generates code that computes the aggregates. The generated code depends on the predication mode. If predication is disabled, we evaluate the aggregate expression without any changes. In case of enabled predication, we need to take special care to not include a filtered out tuple in the aggregation. Therefore, we need to ensure that the aggregate is not changed in case the variable *result_increment* is zero. In case of count or sum aggregation functions, we multiply the tuple value with the

result_increment before applying the aggregation function (cf. Listing 7). This way, the aggregate remains unchanged if and only if *result_increment* is zero.

6.5 Extending Hawk's Code Generator

We now discuss how we can extend the code generator to support new data structures and algorithms.

6.5.1 Support for Different Data Structures

For each supported data structure, Hawk's code generator first requires the code templates for initializing, accessing, and modifying the data structure. These code templates may also be generated at run-time, e.g., to adapt the hash function depending on the data properties. Second, we need to add a new operation for the pipeline program if required, e.g., an index scan. The operations of pipeline programs encapsulate the code generation for different data structures. For example, the HASH_PUT operation generates code for the hash table specified in its parameter. This variability allows Hawk to select different hash table implementations depending on certain data characteristics [50]. For example, to add a robin hood hash table, we need to extend the existing HASH_PUT and HASH_PROBE operations by the respective code templates.

6.5.2 Support for Different Algorithms

Adding a new algorithm to Hawk starts by extending the code generator with its required data structure (if not present). Then, the algorithm needs to be registered to the code generator. We either add a new pipeline operation or include the algorithm in an existing pipeline operation. Finally, we provide the respective code templates. For example, we can extend Hawk with the scan of Zhou and Ross [65], which uses SIMD instructions to check the predicate of multiple tuples at once. To support this SIMD scan in Hawk, we need to add a new code generation mode to the FILTER operation. The mode parameter allows Hawk to select either the scalar or the SIMD code template. Furthermore, the code template for the SIMD scan has to be added to the FILTER operation. The same procedure applies for SIMD support for other operations supported by Hawk.

6.6 Hawk Implementation Details

We implemented Hawk as a prototype that targets column-oriented main-memory database engines. We show the viability of our approaches on the example of CoGaDB [9,10], as it fulfills our requirements and resulted in the smallest integration effort for us. Note that there is no inherent limitation to apply our concepts to other in-memory database systems. The main changes to the database engine consists of the extension of the query plan interface by the produce/consume code generation along with our proposed approach for code variant generation. Furthermore, the execution engine has to be replaced by a run-time for the compiled queries.

Hawk dictionary compresses strings and rewrites comparisons on strings to comparisons on dictionary compressed keys if possible. Hawk keeps only dictionary compressed keys in co-processor memory to reduce the memory footprint. In Hawk, we do not consider executing multiple queries in parallel. We studied the challenges of query concurrency in another publication [10].

Hawk supports all pipeline operations discussed in Section 4.1, which allows for producing code for selections, projections, joins, and aggregations. Aggregations are currently limited to distributive and algebraic aggregation functions (e.g., holistic aggregation functions such as the median are currently not supported).

6.7 Applying Hawk's Concepts to other Systems

Next, we discuss how the concepts of Hawk can be transferred to other database systems. In general, there are two fundamental ways of executing a query: query interpretation [1,6] and query compilation [30,40].

Other interpretation-based systems. Query interpretation calls a particular function for every operator in a query execution plan. Pipeline programs and our concepts are applicable for individual operators as well. In this case, we create for every relational operator a pipeline program and use Hawk's concepts to produce optimized code variants for every target processor for each operator. These operators can either consume the whole input, i.e., implementing bulk processing (e.g., MonetDB), or parts of the input, i.e., implementing vector-at-a-time processing (e.g., VectorWise). Thus, Hawk's concepts integrate with most processing models of main-memory databases [1].

Other compilation-based systems. Pipeline programs and our concepts for code variant generation can be applied to other compilation-based database systems as well. The developer first needs to integrate pipeline programs as intermediate layer between query plans and code generation. Second, the code generator needs to use pipeline programs as the source of compilation. As a result, all concepts of Hawk become applicable.

Other code generation targets. Hawk leverages OpenCL as code compilation target to showcase it's hardware-tailored code generation. However, Hawk is not limited to OpenCL, as pipeline programs allow us to abstract from programming languages. In fact, Hawk is also capable of generating code for C, and in earlier versions of the prototype, we also supported CUDA. Furthermore, we experimented with code generation based on the LLVM framework [32]. In particular, the generated fragments of LLVM's intermediate representation could be combined using LLVM's inliner. Thus, Hawk's architecture supports code generation based on code templates (e.g., C, CUDA, OpenCL) and based on intermediate representations of a compiler (e.g., LLVM).

7 Optimizing Pipeline Programs

Hawk is able to generate a large number of code variants to adapt to different processors. We refer to the set of all variant configurations as *variant space*. The size of the variant space is the cross product of all possible parameter values for each modification supported. We discretize numeric parameters such as the number of threads and the number of work groups to not unnecessarily bloat the number of variant configurations. However, Hawk still faces a large variant search space. Exploring the entire search space is very expensive for two reasons. First, Hawk pays query compilation cost for each generated code variant. Second, the execution time of some code variants may be significantly slower than the optimal code variant. In particular, if Hawk explores code variants that are very slow on a certain processor (e.g., a serial implementation on a GPU), the impact on performance can be significant.

In this section, we discuss how Hawk automatically finds a fast-performing variant configuration for each processor for a given query workload.

7.1 Navigating the Optimization Space

Hawk explores the search space for a processor *offline* by executing a workload of representative test queries. Hawk compiles code variants of each query and explores which modifications are most efficient on a particular processor. We present our strategy in Algorithm 1.

Algorithm 1 Learning an efficient variant configuration for a processor.

```
Input: dimensions of modifications: D = {D_1, ..., D_n}
Input: workload of k queries: W = {Q_1, ..., Q_k}
Output: variant configuration v
 1: v = (v_1, ..., v_n) ∈ D_1 × ... × D_n
 2: for (iter = 0; iter < q; iter++) do
 3:     last_variant=v
 4:     for D_i ∈ D do
 5:         execution_time=∞
 6:         best_dimension_value=∅
 7:         for d ∈ D_i do
 8:             v' = v;
 9:             v'_i = d;
10:             execution_time' = executeQueries(W, v');
11:             if execution_time'<execution_time then
12:                 execution_time=execution_time';
13:                 best_dimension_value=d;
14:             end if
15:         end for
16:         /* Update configuration v in-place */
17:         v_i =best_dimension_value;
18:     end for
19:     if v ==last_variant then
20:         return v;
21:     end if
22: end for
23: return v
```

Core algorithm. Initially, we have no knowledge about the performance behavior of the processor. We start from a base configuration (Line 1), which we initialize with the first parameter value in each variant dimension. In the following, we change one parameter at a time (Line 4–10) and select the parameter value with the best performance (Line 11–14). We perform this step for every variant dimension (e.g., parallelization strategy or memory access pattern). The best parameter values are stored in the variant configuration (Line 16–17, see Section 3.1).

Handling performance dependencies. Note that different modifications may influence each other. For example, depending on the number of threads, a different number of work groups is optimal. This means that a previously optimal parameter value of a modification may be sub-optimal in the new configuration. To make sure that our algorithm finds a fast performing variant configuration, we repeat the core of the algorithm (Line 4–18) iteratively. Note the update to v in Line 17 with the best found dimension value. This makes sure that the outer loop (Line 2) continues with the best found variant configuration from the previous iteration. The algorithm terminates in case we have not found any faster variant configuration (Line 3, 19–21) or reach a maximum number of iterations q (Line 2).

Complexity. Let D_i be a modification of all supported modifications D. Let $|D_i|$ be number of parameter values available for modification D_i and n the number of modifications supported. Then, our learning algorithm has a search complexity of $O(|D_1| + |D_2| + \cdots + |D_n|)$ per iteration. Note that a naive algorithm that explores all variant configurations in the variant space has a complexity of $O(|D_1| \cdot |D_2| \cdot ... \cdot |D_n|)$.

7.2 Reducing Variant Optimization Time

As the variant exploration requires Hawk to execute code variants, very slow code variants increase exploration time significantly. In Hawk, we reduce the exploration time by applying *early termination*, *feature ordering*, and *nested modifications*.

Early Termination: Our learning strategy gains knowledge over the entire variant space. We can terminate the search early, if we reach a local optimum during an iteration. This *early termination* reduces exploration time, but we may not reach the global optimum.

Feature Ordering: If we take the most impacting modifications into account, we can further accelerate the search in the variant space. In this case, we explore the parameter values of the most critical modifications first to find an efficiently performing code variant faster. In our experiments, we identified the following critical modifications: the parallelization strategy, the number of threads, and the memory access pattern. Different modifications can potentially influence each others performance (e.g., number of threads and number of work groups). Thus, the convergence rate of our search algorithm depends on the order in which we explore modifications. By optimizing the dimension order, we also improve the convergence rate of the search algorithm. In our evaluation, our search algorithm usually terminated after 3 iterations.

Handling nested modifications: Some modifications introduce additional *nested modifications* that Hawk needs to apply to a pipeline program. For example, the local hash table aggregation introduces the number of work groups as additional modification to a pipeline program. The global hash table aggregation does not require this parameter. Thus, it would unnecessarily prolong the search if we always include a nested modification. Therefore, Hawk explores nested modifications (e.g., number of work groups) only if the respective parent modification uses a particular parameter value (e.g., local aggregation).

7.3 Building a Heuristic Query Optimizer

Hawk builds a heuristic optimizer using the learned variant configurations. However, Hawk learns variant configurations for a representative query workload. Thus, the resulting variant configuration is a *heuristic* that performs well for a workload. While the heuristic delivers good performance for the given queries, it may become sub-optimal for different query-dependent parameters. To avoid high overhead during query processing, we execute the learning algorithm before query processing starts. Thus, Hawk uses the best found variant configuration (heuristic) of a processor to produce a custom code variant as discussed in Section 3.1.

Optimizing for a particular processor. Different models of the same processor type may require a custom variant configuration. For example, on GPUs the optimal number of threads or the efficiency of synchronization depends on the particular processor. Thus, Hawk learns for each specific processor model a custom variant configuration to achieve peak performance.

Considering Query Dependencies

Parameters such as the predication mode are also query-dependent [53] (i.e., the optimal parameter depends on query characteristics). Thus, we discuss how Hawk can support query-dependent tuning. The variant configurations optimized for each target processor serve as a starting point for further tuning. We introduce a set of heuristics that trigger a rewrite of the pipeline program if a certain condition is met. Such *heuristic-based rewrites* set a query-dependent modification to another parameter value (e.g., number of threads or predication mode), if we expect a performance improvement (i.e., anticipated by known cost models). We discuss two examples for heuristic-based rewrites on pipeline programs in the following: Adjusting the predication mode and choosing the hash table implementation.

Software Predication. The common wisdom for software predication is that it is particularly efficient for selections with a selectivity around 50% [11,53]. In pipeline programs, selections are represented by FILTER operations. Thus, Hawk can use classic selectivity estimation to predict whether the FILTER operations selectivity is close to 50% and switch to software predication or branched evaluation accordingly.

Hash Table Selection. Richter and others evaluated many different hash table implementations for various workloads [50]. One of their key results was a classification under which circumstances which hash table implementation is optimal. Hawk can exploit the results of such studies to derive a set of heuristic-based rewrites for a particular query.

8 Evaluation

In this section, we show our experimental setup and design, present our results on hardware adaption, and discuss the implications of these results.

8.1 Experimental Setup

In our evaluation, we use two machines that have several heterogeneous processors installed. In total, we con-

Table 6 Processors used in evaluation.

Processor	Short	Architecture	Vendor
A10-7850K (CPU)	CPU	Kaveri	AMD
A10-7850K (GPU)	iGPU	CGN 1.1	AMD
Tesla K40M	dGPU	Kepler	Nvidia
Xeon Phi 7120	MIC	Knights Corner	Intel

Listing 8 Grouped Aggregation Query 1

```
select lo_shipmode, sum(lo_quantity) from
    lineorder group by lo_shipmode;
```

Listing 9 Grouped Aggregation Query 2

```
select lo_partkey, sum(lo_quantity) from
    lineorder group by lo_partkey;
```

sider four different processor types with varying architectures: a CPU, an integrated GPU (iGPU), a dedicated GPU (dGPU), and a MIC, as shown in Table 6. All machines run Ubuntu 16.04 LTS (64bit). Depending on the processor's vendor, we have to use a certain OpenCL SDK and driver to compile and run our kernels. For CPU and iGPU from AMD, we use the AMD APP SDK version 3.0. For the MIC, we use the Intel OpenCL SDK Version 4.5.0.8. For the dGPU from Nvidia, we use the CUDA SDK version 8.

In the experiments using processors with dedicated main memory, we cache all input data before running the code variants to avoid biased observations because of PCIe transfers. Our goal is to evaluate the performance of queries on heterogeneous processors, rather than bottlenecks in current interconnects. The interconnect bottleneck is reduced by two factors. First, query compilation uses memory bandwidth more efficiently by keeping intermediate results on the coprocessor. This reduction in data movement among processors diminishes the impact of bandwidth bottlenecks [18]. Second, many modern coprocessors access CPU main memory with the same bandwidth as regular CPUs (e.g., Xeon Phi Knights Landing or when CPU and GPU are connected via NVLink). We run each code variant of a pipeline program five times and report the mean and the standard deviation. We prune the variant space if we detect a very slow code variant (execution time greater than one second) to keep the run-time of the benchmark in a reasonable time frame. The rationale behind the pruning is that a code variant that is orders of magnitude slower than other code variants is not a candidate for the optimal code variant. Repeating measurements to obtain a reliable average unnecessarily prolongs the benchmark's run-time. Since we measure a hot system, we exclude the possibility of measurement artifacts by accessing uncached data.

We use the Star Schema Benchmark [41] and the TPC-H Benchmark [7] dataset to run our experiments. We use Scale Factor 1 for the experiments including a full exploration of all code variants. With larger scale factors, inefficient code variants would not finish in reasonable time. As OpenCL does not provide any mechanism to abort a kernel, we have to wait until the kernel finishes. For all other experiments, we use a scale factor of 10. The main memory of the iGPU usable by OpenCL is limited to 2.2GB and thus, we can not use a larger scale factor.

8.2 Experimental Design

We now present our evaluation workload and the code variants we generate for our test queries.

8.2.1 Queries

All SQL queries can be split in a series of projection and aggregation pipelines. Thus, we evaluate our approaches for code variant generation and optimization on simple queries representing a single pipeline. These queries allow for unbiased observation of hardware adaption using pipeline programs. Additionally, we validate the results on complex benchmark queries.

Projection Pipelines. We take as representatives for projection pipelines one query with 50 % selectivity (Projection Query 1, cf. Listing 1) and one filter query with very high selectivity (<0.01 %, Projection Query 2, cf. Listing 2). While Projection Query 1 is read and write intensive, Projection Query 2 is read intensive.

Aggregation Pipelines. As representatives for aggregation pipelines, we use one query with few result groups (Aggregation Query 1, cf. Listing 8) and one query with many result groups, e.g., several hundred thousand (Aggregation Query 2, cf. Listing 9). The first query is common for the final group by in an OLAP query. The second query is common in sub-queries using a group by. Having so many groups, Aggregation Query 2 is bound by memory latency.

TPC-H Q1 and SSB Q4.1. We perform a full code variant exploration for TPC-H Q1 and SSB Q4.1. The TPC-H query is a compute intensive query. It consists of a single pipeline with a FILTER, several ALGEBRA operations and a grouped aggregation with multiple aggregation functions. The SSB query is a join dominated query (four joins), consisting of four projection pipelines and one aggregation pipeline. The projection pipelines build the hash tables, whereas the aggregation pipeline probes each hash table.

Other Queries. We also evaluate the performance of other Star Schema Benchmark and TPC-H queries. Due to current implementation restrictions of our prototype system (e.g., a missing LIKE operator), we limit the evaluation queries to a representative subset. The star schema benchmark showcases a data warehouse workload, where one central fact table is connected to four dimensions tables [41]. The query consists of four query groups, which vary in the number of joins involved in the queries. We selected Query Groups 1, 3, and 4 as representants, where Query Group 1 has the lowest join intensity and Query Group 4 has the highest join intensity. The TPC-H benchmark loosens the assumptions of the star schema benchmark (e.g., using one centralized table) and provides more computational intensive queries. We select a subset of queries (Q1, Q4, Q5, Q6, Q7, Q19) that cover challenging aspects [7]. Q1 and Q4 are representatives for queries with heavy aggregation. Q5, Q7, Q19 run into parallelization bottlenecks. Additionally, Q1 and Q19 compute many temporary values prior to aggregation. Q4, Q5, and Q7 have problems with data locality (i.e., correlations between the orders and lineitem table) [7].

8.2.2 Variant Space of Generated Code Variants

We now discuss the variant space for our experiments. The total number of code variants multiply with each new variant dimension. We encode the number of code variants in brackets [x variants]. For all pipeline types, we vary the memory access pattern (sequential and coalesced) and the predication mode (branched predicate evaluation and software predication) [2x2 variants].

Projection Pipelines. For projection pipelines, we additionally vary the parallelization strategy (single pass for coarse-grained parallelism and multi pass for fine-grained parallelism) [2 variants]. For the single-pass strategy, we set the number of parallel running pipelines to the number of *maximal compute units* of the OpenCL device [1 variant]. We change the memory access pattern and the predication mode [2x2 variants]. Thus, we generate 4 variants which use the single-pass strategy. The multi-pass strategy uses a multiplier (1, 8, 64, 256, 1024, 16384, 65536) that is multiplied with the number of *maximal compute units* of the OpenCL device to calculate the number of threads [7 variants]. We choose as multipliers powers of two, which allows for a convergence with a logarithmic number of steps in order to find the right order of magnitude. We generate 28 variants that use the multi-pass strategy. In total, we generate 32 variants for a projection pipeline.

Aggregation Pipelines. For aggregation pipelines, we additionally vary the aggregation parallelization strategy, the hash table implementation, and the hash function. For the hash table implementation, we vary between linear probing and Cuckoo hashing [2 variants]. The hash function is either Murmur hashing or Multiply-Shift hashing [2 variants]. For the parallelization strategy, we vary between local and global aggregation. In case of a local aggregation, we optimize the number of hash tables (1, 8, 64, 256, 1024, 16384, 65536) as a multiplier of the number of *maximal compute units* of the OpenCL device to test different levels of thread oversubscription. We also optimize the number of threads per hash table (16, 32, 64, 128, 256, 512, 1024) to find the best configuration between high parallelism and synchronization overhead [7x7 variants]. In case of global aggregation, we optimize the number of threads per hash table, which configurations are identical to local aggregation [7 variants]. We generate [2x2x2x2x7x7 variants] for the local aggregation and [2x2x2x2x7 variants] for the global aggregation. In total, we generate 896 code variants for an aggregation pipeline.

8.3 Results

We validate our concepts as follows. First, we evaluate kernel compilation times for all generated kernels. Second, we evaluate all code variants on representative queries: two projection queries, two aggregation queries, TPC-H Query 1, and SSB Query 4.1. We determine the optimal code variant of a pipeline program by performing a full search. This means that we generate all possible code variants for a pipeline and execute them multiple times. The in average fastest code variant is reported in the plots as CPU, iGPU, dGPU, and MIC optimized. Furthermore, we evaluate our learning strategy for automatic hardware adaption. We report the learned variant configurations and the query execution times on different processors.

8.3.1 Compilation Times

In Figure 16, we show for each of our evaluation queries and processors the compilation time of all code variants in a box plot. The boxes include 50 % of the observations, whereas the upper and lower whiskers mark a 99 % confidence interval. As shown, all compilation times for kernels for the projection query are below 70ms for CPU, iGPU, and dGPU, and below 100 ms for the MIC processor. For the aggregation query, we observe that except for the MIC processor, kernel compilation times are either 100ms, or below. As we need to generate more code for aggregation pipelines compared to projection pipelines, the compilation time increases.

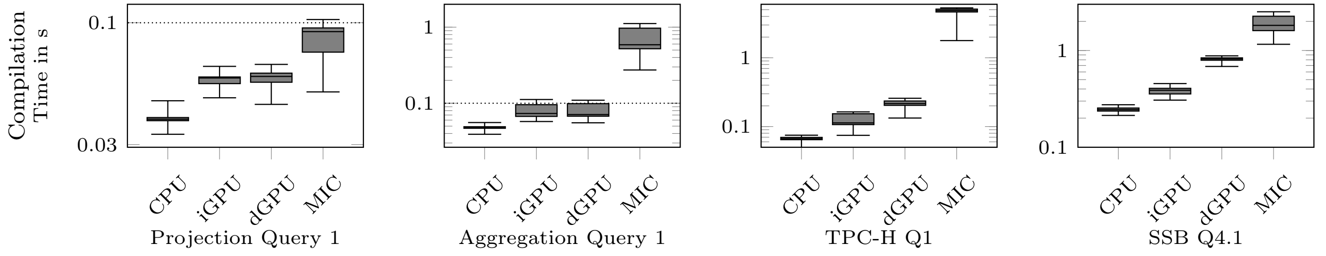

Fig. 16 Compilation times for all generated kernel variants for each processor and query pipeline. Most kernels can be compiled in less than 100ms, which allows for fast query compilation.

# Pipeline Programs	1	2	3	4	5
Compilation Time in ms	40	85	145	192	238

Table 7 Kernel compilation times depending on number of pipeline programs produced for a query.

Compiling TPC-H Query 1 takes longer compared to the aggregation queries. This is because the TPC-H query results in a larger kernel due to many additional computations. We observe 66ms on the CPU, 113ms on the iGPU, 216ms on the dGPU and 4.9s on the MIC. Compiling SSB Query 4.3 is even more time intensive, as we have to compile four projection and one aggregation pipelines. We observe 245ms on the CPU, 380ms on the iGPU, 818ms on the dGPU and 1.8s on the MIC. Note that we can compile multiple pipelines in parallel to reduce the compilation time.

Compiling for the MIC is very expensive and may take longer than a second, even for a single pipeline. However, this is the only processor where we observed this behavior. We repeated our experiments on other machines using NVIDIA GPUs and Intel CPUs, and measured similar kernel compilation times reported here for CPU, iGPU, and dGPU. Thus, we assume that the high compilation time for the MIC is an implementation artifact, which we expect will be resolved in future versions of the Intel OpenCL SDK.

Impact of query complexity. In general, the query compilation time depends on the number of pipeline programs produced during segmentation of a query plan. The number of pipeline programs depends on the number of joins and aggregations involved in the query plan. A query consists of at least one pipeline program. Each join and aggregation computation in the query plan increments the number of pipeline programs (including semi joins produced by IN or EXISTS clauses). We perform a simple microbenchmark, where we measure the compilation time of all generated kernels depending on the number of pipeline programs. We show the result in Table 7. As expected, the compilation time grows linearly with the number of pipeline programs.

Table 8 Execution times (in seconds) of code variants optimized for CPU, iGPU, dGPU, and MIC for different queries, executed on all processors.

	exec.	opt. CPU	opt. iGPU	opt. dGPU	opt. MIC	Learned
Proj. Q1	**CPU**	0.04	1.0	1.2	0.14	0.04
	iGPU	4.7	0.06	0.06	0.25	0.06
	dGPU	6.9	0.03	0.03	0.06	0.03
	MIC	0.48	0.16	0.25	0.15	0.27
Proj. Q2	**CPU**	0.05	0.58	0.61	0.69	0.05
	iGPU	4.4	0.04	0.04	0.04	0.04
	dGPU	3.8	0.01	0.01	0.01	0.01
	MIC	0.24	0.17	0.10	0.10	0.14
Agg. Q1	**CPU**	0.02	0.40	0.61	0.19	0.03
	iGPU	19.4	0.03	0.04	0.19	0.03
	dGPU	11.2	0.02	0.01	0.08	0.01
	MIC	0.6	0.19	0.19	0.03	0.08
Agg. Q2	**CPU**	0.40	0.71	0.70	0.65	0.43
	iGPU	14.9	0.16	0.16	0.18	0.16
	dGPU	8.3	0.09	0.09	0.11	0.10
	MIC	2.5	0.11	0.11	0.11	0.12
TPC-H Q1	**CPU**	0.10	2.07	2.06	0.95	0.11
	iGPU	51.1	0.32	0.81	2.35	0.32
	dGPU	22.6	0.17	0.16	1.5	0.15
	MIC	1.2	3.09	3.09	0.20	0.23
SSB Q4.1	**CPU**	0.09	0.8	0.77	0.57	0.10
	iGPU	16.1	0.10	0.10	0.52	0.13
	dGPU	7.0	0.04	0.03	0.3	0.05
	MIC	0.8	0.21	0.39	0.10	0.20

8.3.2 Full Code Variant Exploration

We show the run-time of all code variants optimized for a particular processor and query. We show these code variants for the following queries: the projection queries (Listing 1 and 2), the aggregation queries (Listing 8 and 9), TPC-H Query 1, and SSB Query 4.3.

Observations Projection Query 1. We show in Table 8, Projection Query 1 that the CPU-optimized code variant outperforms the code variants optimized for the iGPU, dGPU, and MIC by a factor of 25, 30, and 3.5, respectively. However, we see that the same implementation performs more slowly compared to optimized code variants on the iGPU, dGPU, and MIC by a factor of up to 81, 237, and 3.1, respectively. The large performance difference between CPU and the other processors is mainly due to the parallelization strategy: CPUs prefer the single-pass strategy using coarse-grained parallelism, whereas GPUs, and MICs prefer the multi-pass strategy using fined-grained parallelism. On the iGPU, we observe that the code variant optimized for iGPU outperforms the code variant optimized for MIC by a

Table 9 Code variant exploration times for SSB Q4.1 on SF1. Our learning strategy outperforms the backtracking search by up to two orders of magnitude.

Processor	Backtracking (in seconds)	Feature-Wise (in seconds)	Factor Improved
CPU	197,139	479	411
iGPU	78,388	1,219	64.3
dGPU	52,897	1,036	51
MIC	177,914	3,390	52.5

factor of 4.3. For the dGPU optimized code variant on the iGPU, we observe that the performance is equal to the iGPU optimized code variant. The main difference among the code variants optimized for iGPU, dGPU, and MIC is in the optimal number of threads. Furthermore, the MIC prefers sequential memory access, similar to CPUs, whereas the GPU prefers coalesced memory access. Our learning strategy found a configuration that performs closely to the optimal variant on CPU, iGPU and dGPU. On the MIC, the found variant is by a factor of 1.8 slower than the optimum.

Observations Projection Query 2. In Table 8, Projection Query 2, we make the same basic observation for Projection Query 2 (very high selectivity, <0.001 %) as for Projection Query 1 (50 % selectivity). The CPU-optimized code variant outperforms code variants optimized for iGPU, dGPU, and MIC by a factor of 12, 12, and 14, respectively. On the other processors, the CPU-optimized code variant is slower by a factor of 118, 345, 2.4 on the iGPU, dGPU, and MIC, respectively. Our learning strategy found a configuration that performs closely to the optimal code variant on CPU, iGPU, dGPU, and MIC.

Observations Aggregation Query 1. We show in Table 8, Aggregation Query 1 that on the CPU the CPU-optimized code variant outperforms code variants optimized for iGPU, dGPU, and MIC by a factor of 16, 24.4, and 7.6, respectively. However, we see that the same implementation performs significantly slower compared to optimized variants on the iGPU, dGPU, and MIC by a factor of up to 606, 861, and 24, respectively. We also see significant differences between the code variants optimized for iGPU, dGPU, and MIC: On the iGPU, the iGPU-optimized code variant outperforms the code variants optimized for dGPU and MIC by a factor of 1.2 and 6, respectively. On the dGPU, the dGPU-optimized code variant outperforms code variants optimized for iGPU and MIC by a factor of 1.2 and 6. On the MIC, the MIC-optimized code variant outperforms code variants optimized for iGPU and dGPU by a factor of 7.6 and 7.6, respectively. Our learning strategy found a configuration that performs closely to the optimal code variant on CPU, iGPU and dGPU. On the MIC, the found code variant is by a factor of 3.4 slower than the optimal code variant.

Observations Aggregation Query 2. We make the same basic observation as for Aggregation Query 1 (cf. Table 8): On the CPU, the CPU-optimized code variant outperforms code variants optimized for iGPU, dGPU, and MIC by a factor of 1.7, 1.7, and 1.6, respectively. The same CPU-optimized code variant is significantly slower compared to code variants optimized for iGPU, dGPU, and MIC by a factor of up to 94, 94, and 23. Note that for this query, the optimal code variant of iGPU and dGPU is the same, thus we will report numbers only once (GPU). On the GPUs, the GPU-optimized code variant outperforms the MIC by a factor of 1.1 (iGPU) and 1.2 (dGPU). On the MIC, the MIC-optimized code variant achieves the same performance as the GPU-optimized code variant. Our learning strategy found a configuration that performs closely to the optimal code variant on CPU, iGPU, dGPU and MIC.

Observations on Complex Queries. We show that the variant exploration has the same impact on more complex queries. We present the result of the variant exploration for two OLAP queries: TPC-H Q1 and SSB Q4.1 (cf. Table 8). For every processor, we observe similar factors between the optimal code variant and the other code variants. On the MIC, we observed a high variance of execution time for some code variants.

8.3.3 Optimization Time

We now investigate how long the variant exploration itself takes. We show in Table 9 the time to explore the best code variant for SSB Query 4.1. We compare backtracking (executing every possible code variant and choosing the fastest) with our learning strategy. We observe that our strategy improves the search time by up to a factor of 411. While the longest exploration took more than two days, our strategy finished within an hour. Thus, we can run our calibration benchmarks offline (e.g., as part of the database installation process).

8.3.4 Hardware Adaption on Full Queries

In this experiment, we evaluate Hawk and our learning algorithm on complex benchmark queries. To this end, we investigate whether our results carry over to real-world workloads using a representative subset of the star schema and TPC-H benchmarks (cf. Section 8.2.1). First, we derive efficient variant configurations for each evaluation processor. Second, we measure performance to assess the efficiency of our variant configurations.

Learned Variant Configurations. We derive processor-specific optimizers using our learning strategy

Table 10 Per processor variant configurations identified by the variant learning strategy for the SSB and TPC-H workload.

Modifications	Learned Optimizers		
	cpu-o	dgpu-o	mic-o
Parallelization Strategy (Projection)	single-pass	multi-pass	multi-pass
Parallelization Strategy (Aggregation)	local hash table	local hash table	local hash table
Memory Access Pattern	sequential	coalesced	coalesced
Hash Table Implementation	linear probing	Cuckoo hashing	Cuckoo hashing
Predication Mode (Query Dependent)	branched	branched	branched
Thread Multiplier (Projection)	1	16384	65536
Thread Multiplier (Aggregation)	1	1	1
Work Group Size (Aggregation)	256	1024	64

from Section 7. We explore the same variant space as the full exploration, which we discuss in Section 8.2.2. As training workload, we use the Query Groups 1, 3, and 4 of the Star Schema Benchmark and Queries 5, 6, 7 from the TPC-H benchmark. In this experiment, we use a scale factor of 10 for both benchmarks. We show the learned variant configurations optimized for the CPU (cpu-o), for the dGPU (dgpu-o), and for the MIC (mic-o) in Table 10.

The learning strategy correctly identifies that for projection pipelines, CPUs prefer single-pass strategies with coarse-grained parallelism, whereas the GPUs and the MIC prefer multi-pass strategies with fine-grained parallelism. For aggregation pipelines, the CPU, GPU, and MIC prefer local hash table aggregation. The learning strategy also found that the CPU prefers sequential memory access, whereas the GPUs and MIC prefer coalesced memory access. The preferred aggregation hash table for CPUs is linear probing, whereas the GPUs and the MIC are more efficient when using Cuckoo hashing. For the query workload, the learning strategy found that the evaluation using if-statements outperforms code variants that use software predication.

We implement the number of threads as a multiplier of the number of OpenCL compute units ("cores"), as the multiplier quantifies the degree of over-subscription required for a processor. CPUs prefer no over-subscription (one thread per core), whereas the GPUs and the MIC need a large multiplier (over-subscription) to have enough thread blocks ready to hide memory access latencies. Additionally, we need to specify the work group size for aggregation pipelines, which also strongly differs between the different processors.

Performance. We execute for each query a code variant optimized for CPU, dGPU, and MIC and measure the execution times on CPU, dGPU, and MIC without compilation times. Note that each code variant is optimized for a complete workload (cpu-o, dgpu-o, and mic-o). We call these variants *per-workload* code variants. We include measurements of a per-query optimized code variant for each query (q-o) to show additional optimization potential compared to the per-workload code variants. We show the results in Table 11 and include measurements of HyPer (v0.5-222-g04766a1) with the same queries on the same dataset on the CPU.[2] We observe that the code generated by Hawk on a CPU is in the same order of magnitude as an optimized state-of-the-art query compiler.

Most queries are executed faster when we use the per-workload code variant of the target processor. On the CPU, the performance of a CPU-optimized code variants outperforms GPU and MIC-optimized variants by up to a factor of 5.5 (SSB Query 3.4). On the GPU, the GPU-optimized code variant outperforms the other per-workload code variants by up to a factor of 9 (SSB Query 3.2). On the MIC, the MIC-optimized code variant outperforms the other per-workload code variants by up to a factor of 1.12 (SSB Query 3.2). The reason for this low factor is that GPU code variants are typically also fast on a MIC (but not the other way around). However, we can still improve the performance with a custom code variant for the MIC. We occasionally observe a better performance of another code variant for some queries, such as TPC-H Q5 and Q6 for the CPU. The reason for this is that a code variant optimized for several queries may be sub-optimal for a particular query. We conclude that we achieve the best performance when we use a processor-tailored code variant.

If we additionally tune the code variant to a particular query, we observe for our workload speedups on the CPU by up to a factor of 1.02, on the GPU by up to a factor of 27 (TPC-H Query 5), and on the MIC by up to a factor of 1.43 (SSB Query 3.2). The per-query code variants differ mainly in the thread multipliers, as different degrees of parallelism are optimal for different queries on GPU and MIC. Additionally, some queries are faster with enabled predication or prefer global instead of local hash table aggregation specific queries, such as TPC-H Q5 on GPUs and MICs. We conclude that the optimal code variant is query-dependent.

[2] Note that this comparison is not intended to be an end-to-end measurement of system performance.

Table 11 Execution times in seconds of variants optimized for CPU (cpu-o), dGPU (dgpu-o), and MIC (mic-o) for selected queries of the star schema and TPC-H benchmark (Scale Factor 10), executed on a CPU, a dGPU, and a MIC processor.

	HyPer	Executed on CPU				Executed on dGPU				Executed on MIC			
	(CPU)	cpu-o	dgpu-o	mic-o	per-q	cpu-o	dgpu-o	mic-o	per-q	cpu-o	dgpu-o	mic-o	per-q
Q1.1	0.149	0.186	0.441	0.342	0.189	0.067	0.015	0.015	0.015	0.055	0.062	0.057	0.057
Q1.2	0.099	0.113	0.271	0.272	0.114	0.052	0.047	0.047	0.013	0.046	0.061	0.06	0.064
Q1.3	0.092	0.111	0.25	0.248	0.109	0.052	0.022	0.023	0.013	0.047	0.056	0.049	0.051
Q3.2	0.200	0.21	2.258	0.885	0.206	56.697	0.191	1.724	0.138	5.021	0.247	0.221	0.155
Q3.3	0.146	0.115	1.467	0.61	0.114	53.543	0.073	0.472	0.052	3.781	0.14	0.14	0.114
Q3.4	0.146	0.111	1.468	0.615	0.114	53.646	0.073	0.471	0.053	3.795	0.132	0.128	0.109
Q4.1	0.654	0.567	2.186	1.559	0.567	77.701	0.743	5.188	0.25	11.146	0.423	0.397	0.417
Q4.2	0.588	0.444	1.758	1.272	0.45	78.042	0.523	1.704	0.111	8.552	0.341	0.322	0.351
Q4.3	0.316	0.195	2.421	1.073	0.212	58.718	0.764	4.816	0.286	4.709	0.435	0.415	0.343
Q1	0.423	1.428	19.40	10.85	0.995	287.7	2.131	14.50	0.880	15.376	1.220	1.240	1.210
Q4	0.524	0.791	3.680	1.629	0.675	75.513	0.124	9.406	0.0739	7.707	0.339	1.043	0.347
Q5	0.857	0.934	5.105	4.095	1.033	73.572	7.091	10.605	0.261	10.874	0.905	0.907	0.838
Q6	0.147	0.185	0.257	0.258	0.195	0.063	0.009	0.009	0.011	0.033	0.037	0.036	0.036
Q7	0.611	1.293	5.539	2.255	1.097	72.816	5.008	21.147	0.800	11.435	1.080	7.051	1.020
Q19	0.756	0.205	0.420	0.330	0.198	1.509	0.029	1.494	0.029	0.219	0.155	0.200	0.142

Summary. In all of our experiments, we observed that our strategy reliably identified the correct parameters for all hardware-dependent modifications, such as the parallelization strategy (single-pass strategy for CPU, multi-pass strategy for coprocessors). Furthermore, the optimal code variant is query dependent, e.g., number of threads, predication mode [53], and hash table implementation [50]. This is reflected in Table 11, were we contrast the performance of per-query tuned variant configurations and per-workload tuned variant configurations. The query dependencies can be handled by using our heuristics or by adding a run-time optimizer that performs per-query variant optimization, similar to the work of Raducanu [53] and Zeuch [62]. For processors of the same category, the optimal code variant mainly differs in the number of threads and threads per block.

8.4 Discussion

In our experiments, we observed that most compilation times for single pipelines are very fast (< 100 ms). OpenCL could compile even complex queries in several hundred milliseconds, if we disregard vendor-specific artifacts. We conclude that efficient query compilation is possible using OpenCL. Thus, Hawk allows for interactive querying despite using query compilation. Furthermore, we observe large performance differences among code variants optimized for a CPU, a GPU, and a MIC by up to two orders of magnitude. Thus, we conclude that a hardware-tailored code generator is able to achieve high performance gains. This is because it can optimize for various processors of different architectures with previously unknown performance behavior *without any manual tuning*. The diversity of the optimized code variants shows that we need to support the modifications discussed. Finally, we showed that our learning strategy detected all major preferences of all processors. Our strategy derived efficient per-processor code variants without having to explore all code variants.

9 Related Work

In this section, we discuss related work on query compilation, data processing on heterogeneous processors, and automatic optimization of variants.

9.1 Query Compilation

Query compilation goes back to System R [13]. With the upcoming of main-memory databases, query compilation received new attention as reducing main-memory traffic and executed CPU instructions became increasingly important. Rao and others generated query-specific code using the just-in-time compilation capabilities of Java [48]. Krikellas and others used a template-based code generation approach to compile queries to C code, which was then compiled by a C compiler to machine code [30]. Neumann introduced the produce/consume model, which provides a systematic way to generate code that allows for data-centric query processing by fusing all operators in an operator pipeline. Additionally, Neumann proposed to generate LLVM IR code instead of C code to achieve low compilation times [40]. The key difference between the produce/consume model and Hawk is that the produce/consume model generates code directly for each operator. In contrast, Hawk extends the produce/consume model to create a pipeline program for each pipeline in the query plan. These

pipeline programs serve as basis for generating code variants, which is not supported by the original produce/consume model without extensions. Leis and others proposed the morsel framework, which introduces NUMA-aware parallelization of compiled pipelines [34].

Sompolski and others carefully studied vectorized and compiled execution [56]. They observe that compilation is not always superior to vectorization and conclude that compilation should always be combined with block-wise query processing. Dees and Sanders compiled the 22 TPC-H queries by hand to C code and showed large performance potentials for query compilation [14]. Nagel and others investigated query compilation in the context of language-integrated queries in managed run-times [39]. Amad and others developed DBToaster, which uses code generation to compile view maintenance queries to efficient machine code [2]. Query compilation also found its way into commercial products such as Hekaton [17] and Impala [57].

Weld is a run-time that efficiently executes data-intensive applications [42]. The key idea is to compile code to a common intermediate representation. Weld removes data movement between functions in a workflow and generates efficient parallel code for CPUs. In contrast to Hawk, Weld cannot generate custom code for different heterogeneous processors. However, Welds code-generation backend can be enriched by the code variant generation concepts introduced in this paper to efficiently support GPUs and MICs.

9.2 Query Compilation for CPUs and GPUs

Wu and others proposed Kernel Weaver, a compiler framework that can automatically fuse the kernels of relational operators and kernels of other domains [58]. In contrast to kernel weaver, Hawk uses our concept of parallelization strategies to generate a minimal number of kernels. We see Kernel Fusion as a complementary building block. Another key difference is that Kernel Weaver targets GPUs only, whereas Hawk executes efficiently on CPUs, GPUs, and MICs.

A new line of research called abstraction without regret focuses on writing database systems in a high-level language [29]. The LegoBase system uses generative programming to generate efficient low-level C code for a database implementation in a high-level language [28]. Shaikhha and others further refine this principle in DBLAB [54] by introducing a stack of multiple Domain Specific Languages (DSLs) that differ in the levels of abstraction. In DBLAB, high-level code is progressively lowered to low-level code by compiling code in multiple stages, where each stage compiles to a DSL of lower abstraction level, until the final code is generated.

The key difference to Hawk is that Hawk uses a intermediate representation designed to capture all major modifications required to generate code variants for different heterogeneous processors. Thus, Hawk provides an abstraction layer for a variety of coprocessor designs to benefit the evaluation of relational query languages. Finally, Hawk automatically derives an efficient code variant for a processor with unknown performance characteristics. Thus, the concepts of Hawk are complementary to the vision of abstraction without regret.

Pirk and others propose Voodoo, a framework which consists of an intermediate algebra representation based on vectors and a code generator for OpenCL [45]. Based on the algebra, Voodoo is capable of generating code for different processors, including CPUs and GPUs. Hawk's pipeline programs are more high level and allow for modifications not easily expressible in a more low-level representation such as the Voodoo Algebra. In particular, Hawk applies different parallelization strategies (e.g., multi-pass strategy), hash table implementations, and memory access patterns. Voodoo is better suited for the task of vectorizing code and focuses on more low-level optimizations. Pipeline programs and Voodoo algebra can complement each other as consecutive intermediate representations in a query compiler, which also benefits the vision of abstraction without regret. As another key difference to Voodoo, Hawk has shown its capability of producing and automatically deriving processor-optimized code variants.

In summary, existing query compilation approaches generate efficient code for a single processor. Hawk is the first hardware-tailored code generator that produces code variants to run efficiently on different processors.

9.3 Compilers

Brown and others developed Delight, a framework that allows to build, compile, and execute DSLs which enable users to program at a high-abstraction level [12]. The key idea is to compile domain-specific languages to a common intermediate representation. From the intermediate representation, Delight generates code for CPUs and GPUs. However, Delight does not optimize for heterogeneous processors to the degree Hawk does, e.g., changing parallelization strategies. The concepts of Delight and Hawk complement each other.

Dandelion is a general purpose compiler based on .NET LINQ that compiles data-parallel programs to multiple heterogeneous processors, such as CPUs, GPUs, and FPGAs and automatically distributes data processing on different processors, be it in a single machine or a cluster [52]. While Dandelion uses cross compilation to

support GPUs, Hawk profits from the functional portability of OpenCL. This allows Hawk to run code on any OpenCL-capable processor and to generate code for different processors using the same code generator.

9.4 Databases on Heterogeneous Hardware

Balkesen and others studied efficient hash joins [4] and sort-merge joins on multi-core CPUs [5]. He and others developed efficient algorithms for joins [20, 22] and other relational operators [21] on GPUs. He and others also studied efficient co-processing on APUs [23]. Pirk and others studied common database operations on the Intel Xeon Phi (MIC) [44]. Jha and others investigated hash joins on the Intel Xeon Phi [26].

Paul and others investigated the effect of pipelining between multiple GPU kernels using the channel mechanism provided by OpenCL 2.0 pipes [43]. Meraji and others implemented support for GPU acceleration into DB2 with BLU acceleration and observed significant performance gains using GPUs for query processing [36]. Karnagel and others analyzed hash-based grouping and aggregation on GPUs [27]. This work was the basis for Hawk's parallelization strategies for grouped aggregation. Müller and others studied database query processing on FPGAs [38] and developed Glacier, a query compiler that generates logic circuits for queries to accelerate stream processing [37]. Many database prototypes were developed to study query processing on GPUs, such as GDB [21], GPUDB [60], OmniDB [64], Ocelot [24], CoGaDB [10], and HeteroDB [63].

Heimel and others showed the feasibility of building a database engine in OpenCL, which allows to run a database engine with the same operator code base on any OpenCL-capable processor [24]. The core difference between Ocelot and Hawk is that Ocelot facilitates the same operator implementations for each processor. Ocelot only modifies the memory access pattern using explicit knowledge about the processor to use sequential access on CPUs and coalesced access on GPUs. In contrast, Hawk supports more code modifications (e.g., parallelization strategy and predication mode), automatically derives custom per-processor variant configurations, and supports operator fusion.

9.5 Variant Optimization

Raducanu and others propose Micro Adaptivity, a framework that provides alternative function implementations called *flavors* (equivalent to our term code variant) [53]. Micro Adaptivity exploits the vector-at-a-time processing model and can exchange a flavor at each function call, which allows for finding the best implementation for a certain query and data distribution. Rosenfeld and others showed for selection and aggregation operations that many operator variants can be generated and that different code transformations are optimal for a particular processor [51]. Zeuch and others exploit performance counters of modern CPUs for progressive optimization. They introduce cost models for cache accesses and branch mispredictions and derive selectivities of predicates at query run-time to re-optimize predicate evaluation orders [62]. The techniques for variant optimization from Raducanu [53], Rosenfeld [51], and Zeuch [62] are orthogonal to the code variant generation in this paper.

10 Summary

In this paper, we describe a hardware-tailored code generator that customizes code for a wide range of heterogeneous processors. Through hardware-tailored implementations, our code generator produces fast code *without manual tuning* for a specific processor.

Our key findings are as follows. Our abstraction of pipeline programs allows us to flexibly produce code variants while keeping a clean interface and a maintainable code base. Code variants optimized for a particular processor can result in performance differences of up to two orders of magnitude on the same processor. Therefore, it is crucial to optimize the database system to each processor. Consequently, we proposed a learning strategy that automatically derives an efficient variant configuration for a processor. Based on this algorithm, we derived efficient variant configurations for three common processors. Finally, we incorporated the variant configurations into a heuristic query optimizer.

Acknowledgments We thank Tobias Behrens, Tobias Fuchs, Martin Kiefer, Manuel Renz, Viktor Rosenfeld and Jonas Traub from TU Berlin for helpful feedback. This work was funded by the EU projects SAGE (671500) and E2Data (780245), DFG Priority Program Scalable Data Management for Future Hardware (MA4662-5) and Collaborative Research Center SFB 876, project A2, and the German Ministry for Education and Research as BBDC (01IS14013A).

References

1. D. Abadi et al. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.
2. Y. Ahmad and C. Koch. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB*, 2(2):1566–1569, 2009.
3. A. Ailamaki. Database architecture for new hardware. In *VLDB*, page 1241, 2004.
4. C. Balkesen et al. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.

5. C. Balkesen et al. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.
6. P. Boncz et al. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.
7. P. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *TPCTC*, pages 61–76. Springer, 2014.
8. S. Borkar and A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.
9. S. Breß. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum*, 14(3):199–209, 2014.
10. S. Breß et al. Robust query processing in co-processor-accelerated databases. In *SIGMOD*. ACM, 2016.
11. D. Broneske et al. Database scan variants on modern CPUs: A performance study. In *IMDM@VLDB*, 2014.
12. K. Brown et al. A heterogeneous parallel framework for domain-specific languages. In *PACT*. IEEE, 2011.
13. D. Chamberlin et al. A history and evaluation of System R. *Commun. ACM*, 24(10):632–646, 1981.
14. J. Dees et al. Efficient many-core query execution in main memory column-stores. In *ICDE*. IEEE, 2013.
15. Esmaeilzadeh et al. Dark silicon and the end of multicore scaling. In *ISCA*, pages 365–376. ACM, 2011.
16. F. Färber et al. The SAP HANA database – an architecture overview. *Data Eng. Bull.*, 35(1):28–33, 2012.
17. C. Freedman et al. Compilation in the microsoft SQL server hekaton engine. *Data Eng. Bull.*, 37(1):22–30, 2014.
18. H. Funke et al. Pipelined query processing in coprocessor environments. In *SIGMOD*. ACM, 2018.
19. S. Harizopoulos et al. OLTP through the looking glass, and what we found there. In *SIGMOD*. ACM, 2008.
20. B. He et al. Relational joins on graphics processors. In *SIGMOD*, pages 511–524. ACM, 2008.
21. B. He et al. Relational query co-processing on graphics processors. In *TODS*, volume 34. ACM, 2009.
22. J. He et al. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. *PVLDB*, 6(10), 2013.
23. J. He et al. In-cache query co-processing on coupled CPU-GPU architectures. *PVLDB*, 8(4):329–340, 2014.
24. M. Heimel et al. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9):709–720, 2013.
25. J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 5th edition, 2011.
26. S. Jha et al. Improving main memory hash joins on Intel Xeon Phi processors: An experimental approach. *PVLDB*, 8(6):642–653, 2015.
27. T. Karnagel et al. Optimizing GPU-accelerated group-by and aggregation. In *ADMS*, pages 13–24, 2015.
28. Y. Klonatos et al. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.
29. C. Koch. Abstraction without regret in database systems building: a manifesto. *Data Eng. Bull.*, 37(1):70–79, 2014.
30. K. Krikellas et al. Generating code for holistic query evaluation. In *ICDE*, pages 613–624. IEEE, 2010.
31. P.-A. Larson et al. Real-time analytical processing with SQL Server. *Proc. VLDB Endow.*, 8(12):1740–1751, 2015.
32. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–86. IEEE, 2004.
33. V. Leis et al. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. IEEE, 2013.
34. V. Leis et al. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754. ACM, 2014.
35. S. Manegold et al. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB Journal*, 9(3):231–246, 2000.
36. S. Meraji et al. Towards a hybrid design for fast query processing in DB2 with BLU acceleration using graphical processing units: A technology demonstration. In *SIGMOD*, pages 1951–1960. ACM, 2016.
37. R. Müller et al. Streams on wires - A query compiler for FPGAs. *PVLDB*, 2(1):229–240, 2009.
38. R. Müller, J. Teubner, and G. Alonso. Data processing on FPGAs. *PVLDB*, 2(1):910–921, 2009.
39. F. Nagel et al. Code generation for efficient query processing in managed runtimes. *PVLDB*, 7(12), 2014.
40. T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.
41. P. O'Neil, E. J. O'Neil, and X. Chen. The star schema benchmark (SSB), 2009. Revision 3, `http://www.cs.umb.edu/~poneil/StarSchemaB.PDF`.
42. S. Palkar et al. Weld: A common runtime for high performance data analytics. In *CIDR*, 2017.
43. J. Paul et al. GPL: A GPU-based pipelined query processing engine. In *SIGMOD*. ACM, 2016.
44. H. Pirk et al. By their fruits shall ye know them: A data analyst's perspective on massively parallel system design. In *DaMoN*, pages 5:1–5:6. ACM, 2015.
45. H. Pirk et al. Voodoo - a vector algebra for portable database performance on modern hardware. *PVLDB*, 9(14):1707–1718, 2016.
46. R. Rahman. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, 2013.
47. V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11), 2013.
48. J. Rao et al. Compiled query execution engine using JVM. In *ICDE*. IEEE, 2006.
49. J. Rao and K. Ross. Making B+- trees cache conscious in main memory. In *SIGMOD*, pages 475–486. ACM, 2000.
50. S. Richter, V. Alvarez, and J. Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB*, 9(3):96–107, 2015.
51. V. Rosenfeld et al. The operator variant selection problem on heterogeneous hardware. In *ADMS@VLDB*, 2015.
52. C. Rossbach et al. Dandelion: A compiler and runtime for heterogeneous systems. In *SOSP*. ACM, 2013.
53. B. Răducanu et al. Micro adaptivity in Vectorwise. In *SIGMOD*, pages 1231–1242. ACM, 2013.
54. A. Shaikhha et al. How to architect a query compiler. In *SIGMOD*, pages 1907–1922. ACM, 2016.
55. J. Shen et al. Performance traps in OpenCL for CPUs. In *PDP*, pages 38–45, 2013.
56. J. Sompolski et al. Vectorization vs. compilation in query execution. In *DaMoN*, pages 33–40. ACM, 2011.
57. S. Wanderman-Milne and N. Li. Runtime code generation in Cloudera Impala. *Data Eng. Bull.*, 37(1):31–37, 2014.
58. H. Wu et al. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *MICRO*, pages 107–118. IEEE, 2012.
59. Y. Ye et al. Scalable aggregation on multicore processors. In *DaMoN*, pages 1–9. ACM, 2011.
60. Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on GPU devices. *PVLDB*, 6(10):817–828, 2013.
61. M. Zahran. Heterogeneous computing: Here to stay. *Commun. ACM*, 60(3):42–45, 2017.
62. S. Zeuch et al. Non-invasive progressive optimization for in-memory databases. *PVLDB*, 9(14):1659–1670, 2016.
63. K. Zhang et al. Hetero-DB: Next generation high-performance database systems by best utilizing heterogeneous computing and storage resources. *J. Comput. Sci. Technol.*, 30(4):657–678, 2015.
64. S. Zhang et al. OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures. *PVLDB*, 6(12):1374–1377, 2013.
65. J. Zhou and K. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*. ACM, 2002.