# Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware

Holger Pirk
MIT CSAIL
holger@csail.mit.edu

Oscar Moll
MIT CSAIL
orm@csail.mit.edu

Matei Zaharia
MIT CSAIL
matei@csail.mit.edu

Sam Madden
MIT CSAIL
madden@csail.mit.edu

## ABSTRACT

In-memory databases require careful tuning and many engineering tricks to achieve good performance. Such database performance engineering is hard: a plethora of data and hardware-dependent optimization techniques form a design space that is difficult to navigate for a skilled engineer – even more so for a query compiler. To facilitate performance-oriented design exploration and query plan compilation, we present *Voodoo*, a declarative intermediate algebra that abstracts the detailed architectural properties of the hardware, such as multi- or many-core architectures, caches and SIMD registers, without losing the ability to generate highly tuned code. Because it consists of a collection of declarative, vector-oriented operations, Voodoo is easier to reason about and tune than low-level C and related hardware-focused extensions (Intrinsics, OpenCL, CUDA, etc.). This enables our Voodoo compiler to produce (OpenCL) code that rivals and even outperforms the fastest state-of-the-art in memory databases for both GPUs and CPUs. In addition, Voodoo makes it possible to express techniques as diverse as cache-conscious processing, predication and vectorization (again on both GPUs and CPUs) with just a few lines of code. Central to our approach is a novel idea we termed *control vectors*, which allows a code generating frontend to expose parallelism to the Voodoo compiler in a abstract manner, enabling portable performance across hardware platforms.

We used Voodoo to build an alternative backend for MonetDB, a popular open-source in-memory database. Our backend allows MonetDB to perform at the same level as highly tuned in-memory databases, including HyPeR and Ocelot. We also demonstrate Voodoo's usefulness when investigating hardware conscious tuning techniques, assessing their performance on different queries, devices and data.

## 1. INTRODUCTION

Increasing RAM capacities on modern hardware mean that many OLAP and database analytics applications can store their data entirely in memory. As a result, a new generation of main-memory-optimized databases, such as HyPeR [18], Legobase [14] and TupleWare [9], have arisen. These systems are designed to operate close to memory bandwidth speed by ad-hoc generating CPU-executable code.

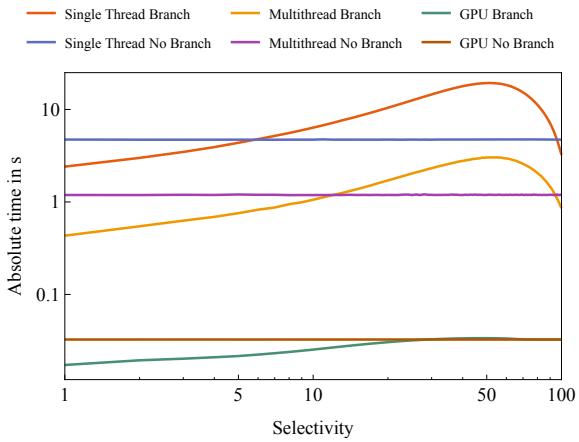



Figure 1: Performance of branch-free selections based on cursor arithmetics [28] (a.k.a. predication) over a branching implementation (using *if* statements)

However, code generation is complex, and as a result most systems were designed to generate code for a specific hardware platform (and sometimes a specific dataset or workload), and require substantial changes to target an additional platform. This is because different architectures employ very different techniques to achieve performance, ranging from SIMD instructions to massively parallel co-processors such as GPUs or Intel's Xeon Phi to asymmetric chip designs such as ARM's big.LITTLE. Exploiting such hardware properly is tricky because *the benefit of most machine code optimizations is data as well as hardware dependent.*

To illustrate the complexity of these tradeoffs, Figure 1 shows the impact of predicate selectivity and architecture on in-memory selections (over one billion single-precision floats). Here, we filter a list of values with a predicate of variable selectivity, using one of two methods: conventional `if`-statements that evaluate the predicate on every item, and a branch-free approach where every input is copied to the output, but the address for the next output is computed by adding the outcome of the predicate (1 or 0) for the current value (so values that don't satisfy the predicate are overwritten by the next value). The branch-free implementation executes more instructions but avoids potentially expensive branch mispredictions. On GPUs, the branching implementation is often better and never significantly


*Proceedings of the VLDB Endowment,* Vol. 9, No. 14

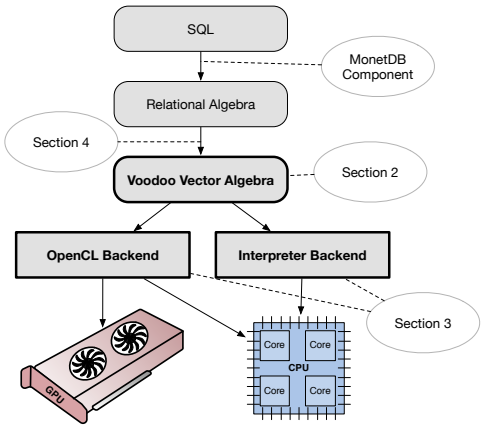


Figure 2: The Voodoo Query Processing System

worse; on CPUs the branch-free implementation can sometimes be up to 4x better in the single-threaded case (where costs are strongly branch-dominated) and 2.5x better in the multi-threaded case (which is less branch-dominated). This example shows why generating high-performance code for modern hardware is hard: even straight-forward optimizations such as the elimination of branch mispredictions must be applied with knowledge of both hardware and data!

Unfortunately, implementing transformations like this in existing code generation engines is hard because it requires encoding knowledge about the hardware (GPU/CPU) and data (selectivities) into the code generator. Furthermore, many such optimizations require *cross-cutting* changes across operators and code components. As a result of this, none of the aforementioned engines implement hardware-specific or data-driven optimizations such as the one shown in Figure 1.

To address this complexity, we developed a new intermediate algebra, called *Voodoo* as the compilation target for query plans. *Voodoo* is both *portable* to a range of modern hardware architectures, including GPUs and CPUs and expressive enough to easily capture most of the optimizations proposed for main-memory query processors in the literature – a property we will call *tunability* in the rest of this paper. For example, it can express different layouts (column vs row) [26, 34], materialization strategies [1], sharing of common subexpressions [6, 14], vectorization [25], parallelization, predication (the above example) [28] as well as loop fusion and fission [32]. While supporting all of these optimizations, Voodoo maintains the efficiency of hand-written C-code by just-in-time generating executable code.

Of course, Voodoo is not the first system to generate efficient executable code for modern hardware. However, existing systems occupy particular design points in the space, implementing architecture-specific techniques to achieve bandwidth and CPU efficiency. Table 1 displays the design choices of several well-known systems (we benchmark against some in our experiments). Porting any of these to a new hardware architecture (with a new set of data-specific bottlenecks) would involve a nearly complete rewrite of the system. We aim to develop an abstraction layer that makes it easy to obtain performance from new hardware architectures.

We termed the key innovation that enables such "tunability" *declarative partitioning*, which allows a code generating frontend to provide information about the desired parallelism of operations in a hardware-independent fashion. This is specifically embodied in Voodoo by a novel technique, called *controlled folding*, where virtual attributes (which we term *control vectors*) are attached to data vectors. By tuning the value of these virtual attributes, frontends can create more or fewer partitions (yielding more or less parallelism) to adapt to hardware with different lane-widths, cache sizes, and numbers of execution units.

| System | Bandwidth Efficiency Technique | CPU Efficiency Technique | Hardware Target |
|---|---|---|---|
| HyPeR [18] | Pipelining | Compilation | CPU-only |
| MonetDB [7] | – | Bulk-Processing | CPU-only |
| Vectorwise [33] | Cache-friendly Partitioning | | CPU-only |
| Tupleware [9] | Pipelining | Compilation | CPU-only |
| Ocelot [13] | – | Bulk-Processing | GPU-Optimized |
| **Voodoo** | **Tunable** | | |

Table 1: Techniques Used in Existing In-Memory DBMSs

Specifically, we make the following contributions:

- We present the Voodoo algebra and describe our implementation of it. Our implementation compiles into OpenCL, and serves as an alternative physical plan algebra and backend for MonetDB [7], an existing high-performance query processing engine that did not previously generate machine code.
- We present a set of principles that guide the design of Voodoo that allow it to efficiently adapt to a range of hardware. These include the use of a minimal collection of declarative operators that can be efficiently executed on different hardware platforms.
- We describe the design and implementation of controlled folding as well as a compiler that generates efficient OpenCL code from Voodoo programs.
- We show that our implementation is performance-competitive with existing database engines designed for specific hardware platforms and hand-optimized code, using simpler and more portable code, and that it can capture several existing and new architecture-specific optimizations.

Note that we *do not* address the problem of programatically generating *optimal* Voodoo code. Instead, we show that Voodoo can be used to express a variety of sophisticated hardware-conscious optimizations, some novel and some previously proposed, and argue that these could *eventually* be chosen via an optimizer that generates Voodoo code.

We structure the rest of this paper around the architecture of Voodoo (shown in Figure 2): after a brief discussion of our design goals in the rest of this section we present the Voodoo algebra that is used to encode query plans in Section 2. Section 3 provides an in-depth discussion of the Voodoo OpenCL backend. In Section 4 we describe how we integrated the Voodoo kernel with MonetDB to accelerate the query processing engine. We evaluate the complete system in Section 5, discuss relevant related (Section 6) and future (Section 6) work and conclude in Section 8.

## 2. THE VOODOO ALGEBRA

To introduce the Voodoo algebra, we start with an example that illustrates its key features and our design principles. After that, we go on to present Voodoo's data model and operators in more detail.

```
1 input := Load("input") // Single Column: val
2 ids := Range(input)
3 partitionSize := Constant(1024)
4 partitionIDs := Divide(ids, partitionSize)
5 positions := Partition(partitionIDs)
6 inputWPart := Zip(input, partitionIDs)
7 partInput := Scatter(inputWPart, positions)
8 pSum := FoldSum(partInput.val, partInput.partition)
9 totalSum := FoldSum(pSum)
```

Figure 3: Multithreaded Hierarchical Aggregation in Voodoo

```
1 3,4c3,4
2 < partitionSize := Constant(1024)
3 < partitionIDs := Divide(ids, partitionSize)
4 ---
5 > laneCount := Constant(2)
6 > partitionIDs := Modulo(ids, laneCount)
```

Figure 4: Multithreading to SIMD in Voodoo (textual diff)

Figure 3 shows a Voodoo program (in *Static single assignment form*) to perform a hierarchical summation: data is first partially summed on N processors, and then the N partial aggregates are themselves summed. Line 1 loads an input vector with a single column "val". Line 2 creates a vector of ids, ranging from $1 \ldots |\text{input}|$. Line 3 and 4 create a vector that maps each tuple to a *partition* by integer-dividing it by the *partitionSize* (1024 tuples in the example). Line 5 computes the output position for each tuple, based on its partition. Line 6 attaches the generated partition ids to the input tuples. Line 7 partitions the input according to the positions computed in line 5 (note that this partitioning is purely logical – meaning it just causes the generated code to loop over the specified number of partitions – unless explicitly materialized). Finally, line 8 performs the per-partition aggregation, and line 9 performs the global aggregation. This example illustrates several key properties of Voodoo and how they enable portability and performance.

**Vector Oriented**: The algebra consists of a small set of *vector operations* like `Scatter` and `FoldSum`, which can be parallelized on modern architectures. Vector instructions enable both portability and performance, as they can be parallelized on many hardware platforms while also yielding to straightforward implementations on any hardware platform. The specific operators in the algebra were chosen to be familiar to compiler-designers reflecting the design of vector machines, SIMD instruction sets, and functional languages, and expressive enough to capture a wide variety of new and previously proposed techniques for optimizing main-memory analytics. For example, in addition to the example in Figure 4, Voodoo is expressive enough to capture the different implementations shown in Figure 1, and compact enough that each implementation is just a few lines of code.

**Declarative**: Voodoo describes dataflow rather than explicit behavior. In particular, the operators only define how the output depends on the inputs – not how the outputs are produced. This allows Voodoo to, e.g., implement lane-wise parallelism (as in `FoldSum`) using SIMD-instructions on CPUs and work-groups/warps on GPUs (which are conceptually very similar). This declarative property is also important for portability in Voodoo, as operators don't describe specific properties of hardware that they rely on.

In addition, it makes the program shorter and simpler than the equivalent C++ program using, e.g., Intel's Threading Building Blocks (see Figure 5) while providing equivalent expressive power. Simplicity arises because it employs a single concept: vector operations. In contrast, TBB involves `blocked_ranges` (line 4), functional `lambdas` (lines 6 and 13) using lexical scoping and a `reducer` (line 13).

```
1  auto input = load("input");
2  auto totalssum =
3   parallel_deterministic_reduce(
4     blocked_range<size_t>(0, input.size,
5                           input.size / 1024),
6   0, [&input](auto& range, auto partsum) {
7         for(size_t i = range.begin();
8             i < range.end(); i++) {
9      partsum += input.elements[i].constant;
10    }
11    return partsum;
12   },
13  [](auto s1, auto s2) { return s1 + s2; });
```

Figure 5: Multithreaded Hierarchical Aggregation in TBB

```
1  auto input = load("input");
2  typedef int v4i __attribute__((vector_size(16)));
3  auto vSize = (sizeof(v4i) / sizeof(int));
4  v4i sums = {};
5  for(size_t i = 0; i < input.size / vSize; i++)
6   sums += ((v4i*)input.elements)[i];
7  int* scalarSums = (int*)&sums;
8  auto totalsum = 0l;
9  for(size_t i = 0; i < 4; i++)
10  totalsum += scalarSums[i];
```

Figure 6: Hierarchical Aggregation using SIMD Intrinsics

Finally, declarative operators allow us to avoid materializing many of the intermediate vectors in a Voodoo program. For example, in the program in Figure 3, most of the vectors (except `input`, `parts`, and `total`) are never stored. These vectors are simply used to control the degree of parallelism in the generated code, as described in Section 3.1 below.

**Minimal**: Voodoo consists of non-redundant, stateless operators. (an example of - hidden - state would be an internal hash table when computing an aggregate). By keeping the API simple, frontends are able to effectively influence the generation and usage of intermediate data structures which may or may not be beneficial for performance. Hidden data structures are also problematic with respect to portability because they can be unbounded in size (again, hash-tables come to mind). Since most co-processors do not efficiently support the (re-)allocation of memory at runtime, unbounded data structures impede portability. Simple, fine-grained operators also lend themselves to fine-grained cost models such as the one we defined in earlier work [21]. Effective reasoning about cost is an integral part of *tunability*.

Non-redundancy in the operator set has three distinct advantages: a) improvements in one operator can improve many queries b) it increases the number opportunities for common subexpression elimination c) it simplifies backend implementation and maintenance.

**Deterministic**: Voodoo programs do not contain runtime control statements such as `if` or `for` that decide if an operator is executed. Such determinism enables efficient execution on architectures with no, or only expensive execution control, such as GPUs and SIMD units, and also improves CPU performance by allowing the CPU to effectively speculate during program execution. It also simplifies cost-modeling.

Determinism does come at a price: the lack of dynamic decisions prevent the frontend from influencing dynamic execution strategies such as load-balancing, garbage collection, memory re-allocation or running loops of which the number of iterations is unknown at compile time. However, this does not imply that generated code cannot make decisions about what data to load (e.g., which is the next node in a tree index), as long as the operations on the data are known at compile time[1]. Operations on trees, e.g., are expressible as long as the depth of the tree is (reasonably) bounded/balanced. To remove the need for such a bound, we plan to (re-)integrate dynamic decisions into Voodoo and will explore the impact of control-statements in future work.

**Explicit**: As much as possible, every Voodoo program has exactly one implementation on each underlying hardware platform. Explicitness is important for tunability, because it means a code generating frontend (or the developer thereof) can reason clearly about what a particular program will do on a particular hardware platform.

**Tunable/Transformable**: The final key property of Voodoo, as noted in the introduction, is that it is *easy to tune* to various hardware platforms using a single abstraction. Since Voodoo already follows a declarative operator model, it is natural to extend this model to create a declarative approach to tuning. In addition to compatiblity with the algebra, this approach has an appealing property: it encodes conceptually similar techniques into structurally similar programs. To illustrate this, consider the example of parallelization using either multiple cores or multiple SIMD-lanes of a modern multicore CPU. This is a non-trivial difference in C: Figure 6 shows code that is equivalent to Figure 5 but uses SIMD intrinsics instead of TBB multithreading. These are almost entirely different programs: the only lines that are shared are the loading of the input (line 1) and most of the output declaration (lines 2 and 9, respectively). In contrast, the changes to the Voodoo program are minimal (see Figure 4 for a textual diff): the constant in line 3 now encodes the number of lanes rather than the size of a partition and the generation of successive runs has been replaced with the generation of circular lane-ids in line 4. This change causes the records to be scattered in a round-robin pattern in lines 5–7, which naturally maps to SIMD instructions. This example shows how an laborious code change is made simple by Voodoo. This simplicity is what we mean by tunable. The previous example also introduces the concept of *controlled folding*, which we describe in more detail later on.

In summary, the design of Voodoo is driven by two primary goals: *portability* and *tunability*. These goals are often contradictory as exemplified by the case of platform-specific extensions of C such as SIMD intrinsics: they can improve performance if the hardware efficiently supports them but hurt portability and sometimes even performance if they are not or only badly implemented [24]. Voodoo alleviates these problems by providing a layer of abstraction that a) can be translated into efficient code for a variety of hardware platforms, b) allows easy and fine-grained control over the generated code while keeping the abstraction simple in order to c) allow reasoning about a program, both in terms of semantics as well as cost (given a hardware platform).

In the rest of this section we describe the data model and operators that allow Voodoo to achieve these properties.

[1]Note that, since we generate code, we have information about factors such as datasizes at compile time

## 2.1 Data: Structured Vectors

The Voodoo data model is based on integer-addressable vectors. We chose this because virtually all hardware platforms implement some kind of integer-addressable memory. Consequently, Voodoo stores data using a thin layer of abstraction over such integer-addressable memory: a model we term *Structured Vectors*. A Structured Vector is an ordered collection of fixed size data items, all of which conform to the same schema. For convenience, we allow data items to contain (nest) other structured data items. Structured Vectors are equivalent to one-dimensional arrays of structs in ANSI C and can, thus, be mapped naturally to native code in a C-derived language such as OpenCL C. We currently only allow scalar types and nested structs as fields (but may, in the future, add fixed size arrays as a convenience feature). To illustrate this, Figure 7 shows two vectors: an input (on top) and an output vector (bottom). The input vector has two attributes (`.fold` and `.value`) and 8 elements.

To address an attribute of a vector in Voodoo code, we use *Keypaths* to "navigate" the nested structures. In notation, keypaths are marked with a preceding dot: for example, the path `.value` designates the *value* component of every tuple in Figure 7. Since structures can be nested, keypaths can have more than one component (e.g., `.input.value`).

**Pointers and NULL values.** Voodoo has no notion of pointers. References to tuples of vectors can be represented similar to foreign-keys: integer values encoding a position in another vector. We provide primitives to resolve these references (the `gather` operation). Regarding `NULL` values, we decided to not impose a specific way but enable common design patterns to deal with `NULL`s such as bitmaps (implemented in, e.g., HyPeR), reserved values (MonetDB) or sparse attributes (Postgres) in Voodoo. In the rest of this paper, we implement `NULL` values using MonetDB's scheme.

Voodoo does have the notion of an "empty" field value which we denote as $\epsilon$ (e.g., in Figure 7). Empty slots occur if, e.g., values are not set in a `scatter` or not selected in a `foldSelect`. We illustrate the usefulness of this concept for efficient code generation in Section 3.

## 2.2 Controlled Folding

A key challenge in Voodoo is providing declarative operators that still give control over tuning parameters (such as parallelism). To address this challenge, we developed a concept we term *Controlled Folding*, which is used to express aggregation (producing a single value) as well as partition-wise selections (producing a sequence of tuple ids). The basic idea of *Controlled Folding* (illustrated for the case of aggregation in Figure 7, as well as in the use of `Divide` and `Modulo`, respectively in Figures 3 and 4) is similar to fold operations in functional languages (e.g., Haskell): reduce a sequence of values into a single value using a binary function. Controlled fold operators in Voodoo are a generalization of functional folds: In addition to the vector of values to fold

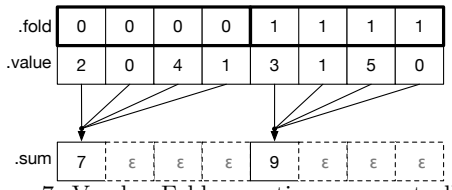


Figure 7: Voodoo Fold operations are controlled

| | Operator | Explanation |
|---|---|---|
| Maint. | `Load(.keypath)` | Load a vector identified by `keypath` from persistent storage |
| | `Persist(.keypath, V)` | Persist vector `V`, making it available from persistent storage under `.keypath` |
| Data Parallel | `BitShift(.out, V1, .kp1, V2, .kp2)` | Shift the value of each item in `V1.kp1` by `V2.kp2` |
| | `LogicalAnd/Or(.o, V1, .p1, V2, .p2)` | Logical Operations |
| | `Add, Subtract, Multiply, Divide, Modulo` | Arithmetic Operations |
| | `Greater, Equals` | Comparison Operations |
| | `Zip(.out1, V1, .kp1, .out2, V2, .kp2)` | Create new vector with substructure `V1.kp1` as `.out1` and `V2.kp2` as `.out2` |
| | `Project(.out, V, .kp)` | Create new vector with substructure `V.kp` as `.out` |
| | `Upsert(V1, .out, V2, .kp)` | Copy `V1` and replace or insert attribute `.out` with value `V2.kp` |
| | `Scatter(V1, V2, .kp2, V3, .pos)` | Create a new vector of size `V2`. Fill the slots of the new vector by placing each value of `V1` into position `V3.pos`. Values are overwritten on conflict. Scatters are performed in order within a value-run in `V2.kp2` — runs have no order guarantees with respect to each other. |
| | `Gather(V1, V2, .pos)` | Create a vector of size `V2` filling it by resolving position `V2.pos` in `V1`. Out of bounds positions result in empty slots. |
| | `Materialize(V1, V2, .kp2)` | Materialize vector `V1` in memory. Materialize chunks of sizes according to runs in `V2.kp2` (X100-style [33] processing) |
| | `Break(V1, V2, .kp)` | Break up `V1` into segments according to the runs in `V2.kp` (pure tuning hint) |
| Fold | `Partition(.out, V1, .v, V2, .pv)` | Generate a scatter position vector to partition `V1.v` according to the list of pivots `V2.pv` |
| | `FoldSelect(.out, V1, .fold, .s)` | Generate a vector of positions of slots in `V1` that have `.s` set to non-zero. Align the output to value runs in `.fold` (see Figure 7) |
| | `FoldMax/Min/Sum(.out, V1, .fold, .agg)` | Calculated Max/Min/Sum for every run in `.fold`. Align output value with start of run. |
| | `FoldScan(.out, V1, .fold, .s)` | Prefix-Sum the values of `V1 .s` (start of new run in `.fold` starts new sum) |
| Shape | `Range(.kp, fromI, [vInt\|v], stepI)` | Generate a vector with the same size as v with values starting from `from` increasing by `step` |
| | `Cross(.kp1, v1, .kp2, v2)` | Generate the cross product of the positions of `v1` and `v2` |

Table 2: Voodoo Operators

they accept a second vector that we term the *control vector* of the operation. The control vector effectively provides the partition ids for values being folded.

The effect of the control vector on the output of an operator is that, when sequentially traversing the values, the operator also traverses the aligned control-sequence. As it does this, it folds all adjacent tuples that have the same partition id into a single output value – the beginning of a new run of partition ids causes the fold to start a new result. The result of each sub-fold is written to the output cell at the start of the partition. The slots between one fold result and the next are padded with empty values. The padding avoids the need for synchronization when processing runs in parallel (we describe how to eliminate the storage overhead of this in Section 3.1.) Controlled folding is a key abstractions in Voodoo that allows us obtain parallel performance from multiple hardware platforms. As we show in the following, it is a very powerful abstraction because it allows declaratively specification of the output of partition-wise operations.

## 2.3 Operators

In this section, we briefly describe the core operators of Voodoo. The goal is to provide an intuition for the kinds of operators we support. A detailed description is provided in Table 2. Voodoo's operators fall into four categories:

1. **Maintenance Operations** manipulate the persistent state of the database. They `import` data, `persist` data to the database and `load` it back.
2. **Data-parallel Operations** operate on aligned tuples in two input vectors. They include standard operations such as arithmetic or logical expressions and `Zips`. `Gather` and `Materialize` also fall into this category because they are trivial to parallelize. The size of the output of these operators is the size of the smaller input. `Scatter` takes a third parameter that specifies the size of the output and technically involves a consistent write to memory. However, as virtually all hardware platforms implement such a primitive, we consider `Scatter` data parallel.
3. **Fold Operations** are operations that require some level of synchronization. In general, this includes all operations where the value at a position in the output depends on more than one value of the input vector. Naturally, this holds for aggregations. However, it also holds for `foldSelect` because it is order preserving: consequently, the position of a tuple in the output depends on the number of qualifying tuples that precede it. `Partition` also takes a vector of pivots as a second input. The size of the output is the size of the input vector (not the pivot vector for partitioning).
4. **Shape Operations** are operations that create vectors with values that are not based on the data values of other vectors but only on their size. Shaped operations do not take any input keypaths because they do not process any attribute values. While we use these operators to generate constants, their most important use is instead to create run-control vectors for fold operations. By generating appropriate fold attribute values and maintaining metadata that describes the runs, we can declaratively control the degree of parallelism in the Voodoo program. Consequently, we call such generated attributes *Control Attributes*. We describe the functioning of these attributes in more detail in the next section.

# 3. VOODOO BACKENDS

The declarative nature of the Voodoo operators makes it easy to provide different backend implementations. While

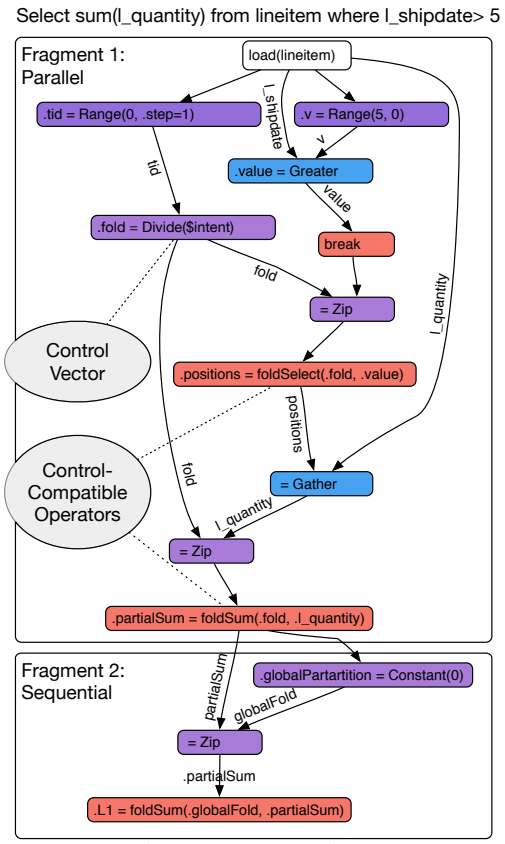


Figure 8: Select & Hierarchically Aggregate in Voodoo

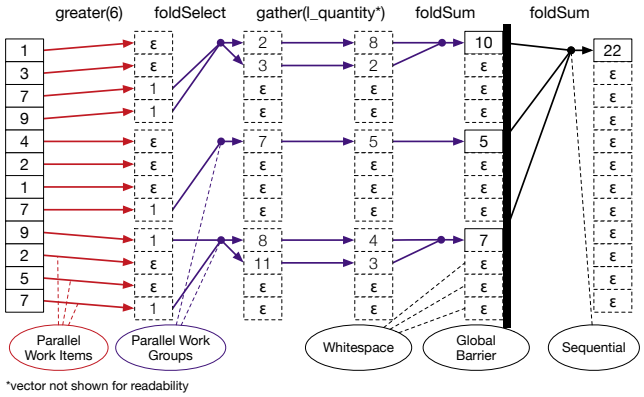


Figure 9: Select & Hierarchically Aggregate in OpenCL

we foresee implementations in different contexts such as distributed processing, we focus our efforts on single-node (multi-core) backends in this paper. We have implemented two backends: an interpreter on top of C++ standard library containers classes as well as a backend that compiles Voodoo code to highly efficient OpenCL kernels. We will start this section with an in-depth discussion of the design of the OpenCL backend and finish with a brief discussion of the interpreter.

## 3.1 The OpenCL Compiler

The purpose of the OpenCL compiler is to generate highly efficient, parallel code that avoids unnecessary data materialization. It does so by generating fully inlined, function-call-free OpenCL kernels from sequences of multiple Voodoo operators. The generation of the kernels is strongly inspired by the code generation process in HyPeR [18], and, following their nomenclature, we will refer to a generated piece of code as a fragment. Result materialization to memory only occurs at the seams between fragments. As in HyPeR's query compiler, the Voodoo to OpenCL compiler traverses the plan in a dependency order and appends statements to a fragment. However, the code generation process in Voodoo is more involved than in HyPeR for two reasons: First, we generate data parallel code whenever possible and only generate code with reduced parallelism when necessary and second Voodoo query plans are DAGs rather than trees (see Figure 8) to enable sharing of intermediate results.

In the following, we illustrate the code generation process taking these complicating factors into account. As a running example, we use a simplified version of TPC-H Query 1 (See Figure 8). The query is:

```
SELECT SUM(l_quantity) FROM lineitem
GROUP BY l_returnflag
```

### 3.1.1 Code Generation

As in earlier in-memory data management systems such as Vectorwise/MonetDB X100 [33] and HyPeR [18], we aim to avoid the materialization of intermediate results. Similar to HyPeR, we generate code by traversing the execution-DAG in a linearized order (top to bottom in Figure 8).

**Controlling Parallelism.** Most Voodoo programs contain fully parallel as well as controlled fold operations. The OpenCL backend needs to efficiently map both to kernels with the appropriate degree of parallelism. To this end, Voodoo assigns an *Extent* and an *Intent* to each generated code fragment. The *Extent* is the degree of (data) parallelism (roughly equivalent to the global work size in OpenCL or the number of parallel threads working on a vector) while the *Intent* is the number of sequential iterations per parallel work-unit. A fragment of Extent 1 is fully sequential while a fragment of Intent 1 is fully parallel. Before describing how we derive these parameters from the *Control Vectors*, we first discuss how they affect a generated program.

The DAG property of our query plans forces us to maintain multiple active fragments at the same time. To illustrate this, consider the case of compiling a program in which a number of fully parallel statements are interrupted by a run-controlled sequential one. In this case, Voodoo creates a sequential fragment in addition to the already active parallel fragment — neither is executed until needed.

When processing an instruction, the compiler chooses a code fragment that has the same *extent* as the current statement. This process is applied in a straightforward manner for data-parallel, maintenance, and shape operators in Table 2. For fold operators, we distinguish three cases: a) if the runs are of length 1 (the extent is $n$ and the intent 1), the fold-operation is fully data-parallel and can be appended to a code fragment of the right extent b) if there is only a single run of length $n$ (the intent is $n$ and the extent 1), the fold-operation is fully sequential and can only be appended to a sequential fragment of the right intent c) if the length of the runs is less than or equal to the supported partition size, values are written to the beginning of the partition. In that case, we do not need to introduce a global synchronization point, i.e., create a new fragment. This means the fold can

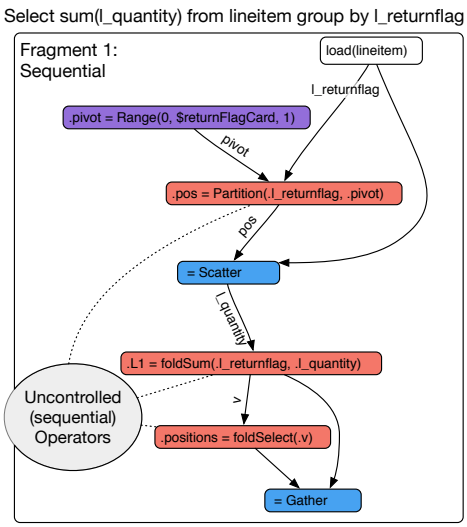



Figure 10: Group & Aggregate

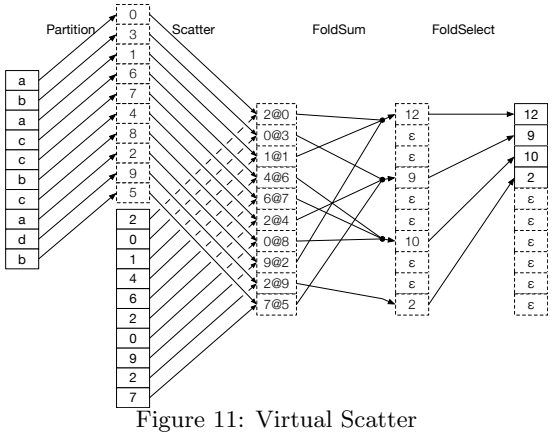



Figure 11: Virtual Scatter

apply to any fragment that has the same extent. Specifically this can be achieved by synchronizing and sequentially processing the input using a subset of the active data items by disabling some of the cores in OpenCL.

If no fragment can be found that has the same extent as the current operator, a new fragment is created.

**Example.** To illustrate this process, consider the Fragment 1 in Figure 8. It is easiest to understand this diagram in terms of the red operators, which either partially or completely interrupt the pipelining of the plan. Starting from the top, the `break` operator causes the greater than expression to be computed in a full-data parallel manner, and the result of this expression to be materialized in memory (or cache). Then, the first fold (`.position = foldSelect`) computes the positions in the `lineitem` table where the predicate on `l_shipdate` is satisfied. The parallelism of this operator is controlled by the `.fold` control vector. In computing `.fold`, the $\$intent$ variable encodes the length of the runs allowing to transition from fully parallel ($\$intent = 1$) to fully sequential ($\$intent = n$). This `foldSelect` operator results in one loop per fold (the extent is $\|input\|/\$intent$); each loop in this case just loops over the boolean values materialized at the `break` operator and emits the non-zero positions. Finally, the third red operator (`foldSum`) computes the partial summation of each fold. It uses the same parallelism as the `foldSelect`, so no additional materialization is necessary, and they can be pipelined into the same loop instance.

There are a few additional things to note about this diagram. First, the parallelism of each operator is determined by the parallelism of its first red ancestor, as the Voodoo compiler aggressively inlines operators between the red pipeline-breaking operations. Second, the purple operators (and associated vectors) are "virtual" in the sense that they simply control the parallelism of the generated program but are not computed (or generated) at runtime.

Figure 9 illustrates the resulting OpenCL program assuming $\$grainsize$ is set to 4. The figure illustrates that the `predicate` is evaluated fully data-parallel. The select as well as the first sum are evaluated using locally reduced parallelism but without the need for a global barrier. The final sum (Fragment 2 in Figure 8) is entirely sequential and requires a global barrier (in the form of a new OpenCL kernel).

Note that, to keep the example concise, the `foldSelect` and the first `foldSum` share the fold attribute (stemming from the same *Control Vector*). This is not necessary in practice, as they are independently tunable. Naturally, selecting the optimal parallelization strategy is hard and not the focus of this paper. The example does, however, show how a complex optimization decision can be encoded into a (set of) integer constant(s). However, it hinges on an effective way to keep control vector metadata.

**Maintaining Run Metadata.** While control vectors allow us to specify the degree of parallelism in an abstract manner, we still have to generate appropriate code from that specification in the backend. For that purpose, the Voodoo compiler maintains descriptive metadata about each generated vector attribute: the start, a step factor and a modulo cap. Attributes values generated by range operators can, thus, be calculated using the following equation:

$$v[i] = from + \lfloor i * step \rfloor \mod cap$$

Dividing a vector by a constant $x$ is equivalent to dividing $step$ by $x$. A modulo by $x$ is setting the $cap$ to $x$. When combining (e.g., through addition) a control vector with a data vector (to encode parallel grouped aggregation), we keep the values of the control vector in addition to the data values. We expect the frontend to ensure that the bits of the control vector do not conflict with the data bits and throw an exception when this assumption is violated.

### 3.1.2 Empty Slot Suppression

As described in Section 2.2, the outputs of all operations are of statically known size and are padded with empty slots. While this seems wasteful at first, it makes it possible to efficiently execute Voodoo on massively parallel hardware without the need for expensive write conflict handling. However, by applying control vector metadata knowledge, we can drastically reduce the memory consumption. To illustrate this, consider the `foldSum` steps in Figure 9. The folding creates a predictable number of empty cells in the vector. To reduce the memory footprint, slots that can be guaranteed to never be filled with values (e.g., when folding all values of a work group into a local sum) can simply not be allocated. We suppress empty slots by allocating a smaller buffer, appropriately modifying the generated write cursor

and annotating the vector with a metadata field to keep track of the empty slots.

### 3.1.3 Virtual Scatter

Another case in which we exploit compile-time knowledge to avoid runtime materialization is the case of `scatter`. A naïvely implemented scatter would write all input values to a slot in the output causing a break in the fragment and substantial random memory traffic. This may, however, be unnecessary if the scattered vector is only created to be used, e.g., as an input to an aggregation. Figure 10 depicts this very common case. To avoid the intermediate materialization, we can drop the assumption that OpenCL work items (i.e., positions in the input vector) and data items (i.e., positions in the output vector) are aligned.

This is illustrated in Figure 11: we tag a vector with a *scatter position* (denoted with an @ in the third vector). That scatter position is a per work-item local variable (split into partition start and tuple offset) that will be used to determine the position of each output tuple if the vector is ever materialized. Since most vectors are never materialized, scatter can become a very cheap operation, that is paid for only when and if it is fully materialized. Figure 11 illustrates how we exploit this in the case of a (single-partition) grouped aggregation: the partition generates a partition id which is work-item local. The scatter creates a (virtual) vector that contains the input values annotated with the scatter path. The vector is read by the foldCount (a macro on top of foldSum) which generates the (partition aligned) counts. Finally, these counts are compacted into a contiguous memory region for the result.

## 3.2 The Interpreter

The interpreter mainly serves as a reference implementation; it uses vectors of maps to represent data as well as control vectors. The interpreter materializes all intermediate vectors and is, in that respect, a classic bulk-processor. However, since it stores all data in `vectors` of `maps`, it uses virtual function calls to retrieve values from tuples. The combination of full materialization and virtual function calls means that this backend is not designed for high performance. It is, rather, a small reference implementation that is useful for debugging and verification because all intermediates are materialized and, thus, inspectable.

# 4. A RELATIONAL FRONTEND

To demonstrate Voodoo's effectiveness as a database kernel as well as to simplify experimentation, we implemented a prototype of a relational query processing engine on top of the Voodoo algebra. Due to its easy extensibility and open source, we chose to integrate Voodoo as an alternative execution engine into MonetDB. However, by replacing MonetDB's execution engine and physical optimizer, we effectively reduced its role to data loading and query parsing. To illustrate the effectiveness of the resulting system, let us briefly walk through the aspects of storage, query processing and optimization.

**Loading.** MonetDB exposes its internal catalog information, including pointers towards the backing files, through a queryable SQL interface. We exploit this to directly load data from the filesystem bypassing the query processor. Upon startup, Voodoo loads data from the internal catalog of MonetDB. We directly copy data from disk into the processing device, using the same storage format MonetDB uses: binary column-wise using dictionary encoding for strings.

**Queries.** We use MonetDB's SQL to relational algebra compiler to parse SQL queries and remove logical concepts such as nested subqueries. From the relational algebra representation, we generate Voodoo plans, thus bypassing MonetDB's physical optimizer. Voodoo plans are similar in complexity to those that MonetDB uses for physical optimization and query evaluation.

*Optimization.* Since Voodoo compiles MonetDB's logical plans, it inherits the logical optimizations that MonetDB applied (join-order, query-unnesting, etc.). Beyond that, the physical optimizer has a number of optimization flags that enable hardware-specific optimizations: cache-conscious partitioning, predication, parallelization strategy (for selections and aggregations) as well as the targeted device.

While we use these flags for the microbenchmarks in Section 5.3, we disable them when comparing the macrobenchmark results (Section 5.2). This allows a comparison of the efficiency of the generated code to that of HyPeR (which also does not apply such optimizations). However, we enable the generation of parallel plans by specifying appropriate control vectors. Beyond that, we use identity hashing on open hashtables and derive their size from the input domain (using only min and max). A detailed per-query discussion of the query plans can be found in our upcoming technical report [23].

# 5. EVALUATION

To evaluate our approach, we study the system with respect to our two design goals: *portability* and *tunability*. The portability experiments in Section 5.2 show that Voodoo not only runs on different hardware platforms but matches (and even outperforms) existing systems tailored to specific hardware architectures. Specifically, we compare against HyPeR [18] and MonetDB/Ocelot [13], which target CPU and GPU architectures, respectively. We conduct the evaluation using a subset of the TPC-H benchmark that was used to evaluate these systems when they were presented.

The tunability experiments in Section 5.3 demonstrate how Voodoo facilitates the implementation and examination of various hardware-conscious optimizations on different platforms. We highlight a number of hardware and data dependent trade-offs that Voodoo allows us to explore.

## 5.1 Setup

We ran our CPU, as well as all of our GPU-experiments, on a Dell Server with single Intel Skylake Xeon E3-1270v5 running Ubuntu Linux 15.10 (Kernel 4.2.0-27) at 3.60GHz with 64 GB of RAM. The GPU was a GeForce GTX TITAN X with 12 GB of global memory using CUDA version 7.5.

All micro-experiments were compiled using Intel ICC 16. For both CPUs and GPUs, we only counted the execution time once the data was loaded into their respective memories and ignore costs for result output. We do not address the PCI bottleneck.

## 5.2 Baseline Portability: TPC-H

We ran a significant subset of the TPC-H queries on a scale factor 10 dataset using Voodoo, HyPeR and, where applicable, Ocelot (Ocelot does not actually support all of

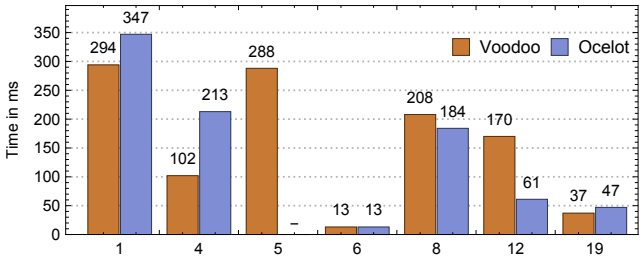


Figure 12: TPC-H Performance on GPU

the queries we evaluated). These queries cover most of the relational operators (selections, aggregates, group-by and joins) exposing standard problems in relational query processing [6]. Note that our goal is to demonstrate the capability of the Voodoo system to generate efficient code — not the quality of the relational query frontend and optimizer, which are still quite basic. In terms of the optimizations of the generated code, our implementation is roughly equivalent to the code generation that is implemented in HyPeR (no vectorization, no manual SIMD instructions). However, we aggressively exploit available metadata (min, max, FK-constraints) which, in many cases, allows us to bypass operations such as hashing or collision management which HyPeR has to perform.

The results on the CPU (Figure 13) and GPU show that, as an abstraction, Voodoo imposes little overhead. Generally, performance is comparable to HyPeR's. Voodoo performs better for compute and lookup-intensive queries (such as 5, 6, 9 and 19) because of the aggressive exploitation of metadata and the use of SIMD instructions by the OpenCL compiler. HyPeR performs better for order-by/limit queries since it evaluates these using priority queues which avoid expensive materialization of the full result (in Voodoo, the order-by/limit clauses were ommitted).

When comparing to Ocelot, the benefit of executable code generation becomes clear: where HyPeR and Voodoo avoid expensive materialization of intermediate results, Ocelot pays a high price for doing so. In particular the low-selectivity (high output cardinality) queries such as query 1 expose this problem. This is an indication that Ocelot was developed for an architecture with a high memory-bandwidth such as GPUs. When evaluating on the GPU, with its 300GB/s memory bandwidth, we see that Ocelot suffers significantly less from that design decision (see Figure 12).

## 5.3 Tunability

We now turn our attention to the ability of Voodoo to experiment with different hardware-aware optimizations, focusing on ease of implementation and the performance tradeoffs of these different techniques.

**Just-in-time layout transformations.** Most in-memory databases store data using a fixed physical schema (usually row-wise or column-wise). However, it has been shown that performance can be increased by transforming the layout before or even during query execution [34]. As an operation that can benefit from such an optimization, we consider the evaluation of an indexed foreign-key join (essentially a positional lookup) on multiple columns of the same table.

We consider three possible implementations of this operation in C and Voodoo: a single traversal of the index-column with lookups into both columns (termed *Single Loop*), two consecutive traversals of the positions, each resolving the keys into one of the target columns (termed *Separate Loops*) and the transformation of the target table from column- to row-wise storage, followed by a single loop over the positions and their resolution (termed *Layout Transform*).

As shown Figure 14a, the best implementation is dependent on the lookup pattern into the target table: if the lookups are sequential, locality is always good and the *Single Loop* implementation performs best. If the lookups are random and the target table small (4MB) the best implementation is to resolve the keys in two *Separate Loop*s. If the lookups are random and the target table large (128MB), a *Layout Transform* pays off: because the values of both projected columns are co-located this optimization reduces the number of random cache misses by two.

Expressing these optimizations in Voodoo involves a `break` operator between the two gathers to switch from *Single Loop* to *Separate Loops* and a `zip` and `materialize` to switch to the *Layout Transform* implementation. As displayed in Figure 14b, Voodoo accurately matched the performance of the C implementation on the CPU.

Figure 14c shows that the performance on the GPU is similar but the *Separate Loops* version is outperformed by the *Layout Transform* implementation in all cases. This experiment illustrates how the lack of large per-core caches on the GPU penalize random accesses earlier than on a CPU. Still, the optimization ports reasonably well.

**Selective Aggregation.** Since selections are one of the most frequently used operators, an efficient implementation is one of the cornerstones of good in-memory performance. The main problem with the default (branching) implementation is the penalty for branch mispredictions. As shown in Figure 1, avoiding these branch mispredictions via a branch-free *predication* technique can be beneficial as it trades memory traffic for fewer mispredictions, although the benefit depends on selectivity. To avoid the additional memory traffic of predication, the processing can be *vectorized*: divided into cache-sized chunks, where for each chunk, a position list is generated using a branch-free implementation. This position list is then traversed and processed in a second (cache-sized) loop. We implemented these three design alternatives in C and Voodoo and show the selectivity-dependent results in Figure 15a: we see the characteristic behavior of speculative execution with worst-case performance at 50% selectivity. In contrast the branch-free implementation shows flat performance that outperforms the branching implementation for mid-range selectivities. The vectorized version performs significantly better than the branch-free implementation and, for selectivities above 1%, outperforms the branching version. Note that the C-code of these versions looks very different: two loops and an additional buffer vs. a single loop with no buffering. In Voodoo, we achieve virtually identical performance (Figure 15b) but only need to insert a single additional operator: a materialize with a control-vector to encode the intermediate buffer size.

Running the Voodoo code on the GPU creates a different picture (see Figure 15c): since the GPU does not speculatively execute code, the predicated version only adds additional memory traffic without any benefit. Surprisingly, the vectorized implementation hurts performance: the additional position buffer causes additional memory traffic and, even worse, is filled sequentially, which limits the degree of parallelism that can be used to hide latencies. We conclude that this optimization does not port well to GPUs.

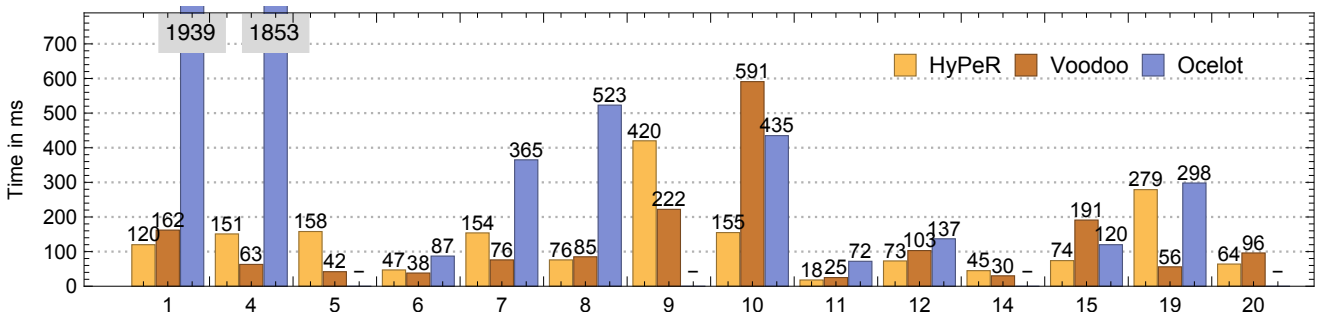
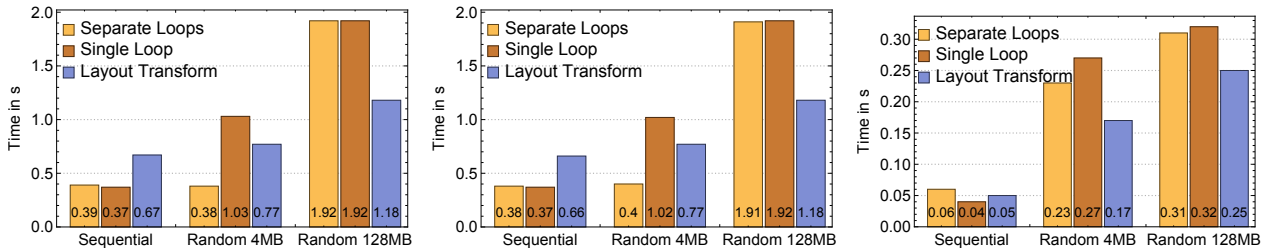


Figure 13: TPC-H Performance on CPU



(a) Implemented in C (b) Voodoo on CPU (c) Voodoo on GPU

Figure 14: Just-in-time layout changes.

**Branch-Free Foreign-Key Joins.** In the last microbenchmark, we present a novel tuning technique that illustrates the interplay between data access and processing. We consider a table-scan, the application of a selection and an indexed foreign-key join into a single, large target table with subsequent aggregation. The SQL-query is:

```
SELECT sum(target.v) FROM fact, target
WHERE facts.target_fk = target.pk
AND facts.v < $1
```

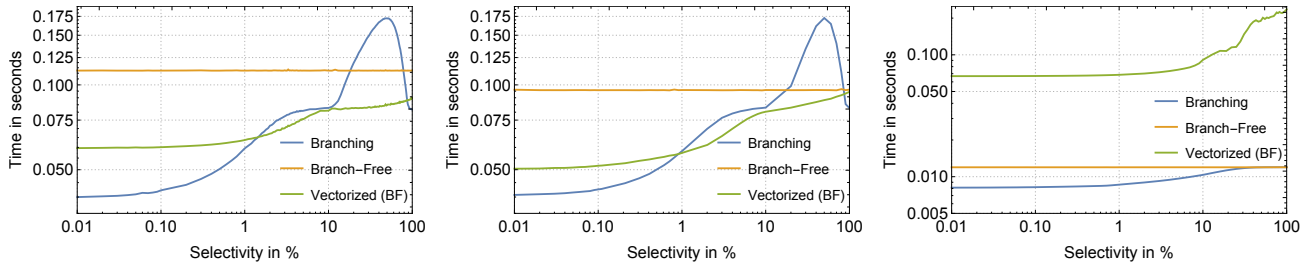
The straightforward approach is *Branching*: a sequential scan of the selection column (`facts.v`), the evaluation of the predicate and a lookup and aggregation of qualifying tuples. A branch-free alternative is to unconditionally perform the lookups and multiply the resulting values with the outcome of the predicate (0 or 1) before aggregation (labeled *Predicated Aggregation* in Figure 16). The selectivity-dependent results are displayed in Figure 16a: the branching version exhibits the typical bell-shaped curve that indicates bad speculation. The branch-free variant is significantly more expensive, due to the high number of random cache misses that result from the unconditional lookups. To address the bad cache behavior, we devised an optimization we term *Predicated Lookups*: before performing the lookup, we multiply the position with the outcome of the selection predicate. This way, all non-qualifying lookups hit the same address (position zero) which will be held in one "very hot" cache line. This addresses the bad cache behavior but causes an extra arithmetic operation (note that the looked-up values still need to be predicated to ensure correctness). The result (*Predicated Lookups* in Figure 16a) is an implementation that performs significantly better than the branch-free version and outperforms the branching version for much of the parameter space. Voodoo matches this result very accurately on the CPU (see Figure 16b).

On the GPU, Voodoo exposes different performance tradeoffs: the *Branching* implementation shows the best performance over most of the parameter space and is only outperformed by the *Predicated Lookups* version for selectivities above 80%. This result exhibits another GPU design decision: the sacrifice of integer arithmetic for floating point performance . Since the *Predicated Lookups* performs two integer arithmetic operations, performance is dominated by that – this optimization also does not port well.

## 6. RELATED WORK

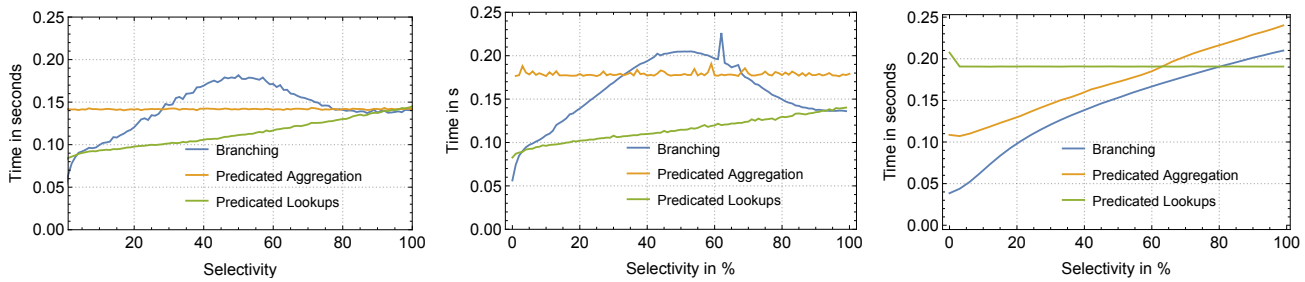
The Voodoo project was inspired by the disparity of the large number of optimization techniques in the literature and the small number of such techniques that are actually used systems. Before concluding, let us provide a overview over some techniques and systems that try to use them.

**Low level optimizations for modern hardware.** Most of the work in the literature addresses specific hardware components in isolation. Among these are techniques addressing hierarchical caches [3], branch predictors [28] and SIMD [25] registers . Much of this prior work shows that each of these techniques can lead to orders of magnitude performance improvements. However, these techniques were not studied in the context of a full data management system stack or generalized into a full operator model. red The most recent of these, Polychroniou et al. [25] develop a set of algorithms that employ advanced techniques for SIMD-enabled processing. Most of these can be translated directly into equivalent Voodoo code (see our upcoming technical report [23] for details). The exception to this are the cases in which data structures are not write-once: when filling hash-table slots with unique marker values (to recognize conflicts) and subsequently overwriting them with data values and when swapping values through a cuckoo-table until an empty slot is found. The former can be implemented by using a second (logical) buffer to hold the marker values. The later can only be approximated in Voodoo because each cuckoo iteration needs to (logically) create a new data structure. While the memory overhead can be removed at compile-time, the program grows linearly with the number of cuckoo-iterations. This bounds the number of possible iterations to a (reasonably small) constant.




(a) Implemented in C (b) Voodoo on CPU (c) Voodoo on GPU

Figure 15: `select sum(v2) from facts where v1 between $1 and $2`




(a) Implemented in C (b) Voodoo (Run on CPU) (c) Voodoo (Run on GPU)

Figure 16: Selective Foreign-Key Join Performance

Another line of research targets the use of programmable GPUs with its diverse tuning techniques [11, 12, 22]. GPU programming textbooks (e.g., [20]) contain a number of platform specific heuristics to choose the right set of techniques given a problem. Unfortunately, the low-level programming paradigm of frameworks such as OpenCL or CUDA makes this kind of optimization hard, often requiring a substantial rewrite of the program for each architecture or optimization.

**High Performance Computing (HPC).** The HPC community has aimed to create easy to use abstractions over highly parallel hardware for more than three decades. The most prominent artifact of this is the BLAS standard with several implementations. Aimed at linear algebra, however, BLAS implementations such as Intel's MKL [31], cuBLAS [4], MAGMA [2] or OpenBLAS solve a restricted and, most importantly, data-independent tuning problem. Hence, tuning is usually done by the developers of the library, rather than generating data-dependent code when the application runs. Compiler frameworks such as Delite [30] or Dandelion [29] are designed to facilitate this process but it still remains with the library developer and, thus, static.

General purpose compile-time abstractions such as Intel's Array Building Blocks (ABB) [19] inherited this problem. ABB specifically was abandoned in favor of "tunable" approaches such as Cilk [5], Threading Building Blocks [27] and OpenMP [10]. While these offer high tunability, they are ill-suited to automatic code generation at runtime (see Section 2). The approach closest to ours is ArrayFire [17] which provides abstract vector operations backed by multiple hardware specific backends (CUDA, OpenCL and C++). ArrayFire even generates code at runtime but only for arithmetic expressions applied using a map operator.

**State of the art systems for in-memory analytics.** Some of the techniques used by this work involve bulk-processing [7] vector processing and just-in-time compilation. MonetDB/x100 [33] (a.k.a. JIT-compiling) [15]. HyPeR [18], employs a simple direct translation of relational algebra to LLVM assembler which is then executed. HyPeR aims to run both analytic and transactional workloads in the same system. Legobase [14], in addition to generating LLVM or C code from SQL, lets database developers express internal database data structures and algorithms using a high level language (Scala) and then have them compiled down to low level with the rest of the query. TupleWare [9] aims to handle a larger class of computations including iterative computations and UDFs, and employs the code generation technique to efficiently integrate framework code with the UDFs. Ocelot aims to port MonetDB to exploit GPUs [13]. Voodoo is complimentary to HyPeR and Legobase in that Voodoo can be used as a lower layer for such systems.

# 7. FUTURE WORK

We believe Voodoo to be useful as a foundation for much future work in high- as well as low-level optimization of database queries. The machine-friendly design of Voodoo lends itself to automatic exploration of the database design space. Specifically an automatic, incremental, runtime re-optimization system is enabled by the design of Voodoo. Such a system could employ current and future low-level optimizations. However, it will still have to handle the large design space and may require new abstractions. We believe that declarative optimizers [8, 16] can be effectively combined with Voodoo to handle this complexity.

The current Voodoo design deliberately omits control-statements (`for`, `if`, `while`, ...). While these are not necessary to implement relational algebra, they enable runtime optimizations such as load balancing, dynamic resizing of data structures (e.g., hash tables) or data structures that have complex behavior (such as priority queues). However,

these are exactly the kinds of optimizations that are difficult to port to massively parallel architectures. A solution to this problem is likely to be based on the co-operation of multiple devices (CPUs for control, GPUs for data processing). We plan to develop such a solution in the near future.

We also plan to expand the current algorithms to add non-relational operations. An example of this is supporting regular expression matching (which has efficient massively parallel solutions) but also support for graphs or arrays.

# 8. CONCLUSION

Implementing efficient in-memory databases is challenging, and often requires being aware of both the characteristics of the workload, data and the specific architectural properties of the hardware. In this work, we proposed *Voodoo*, a novel unifying framework to explore, implement and evaluate a range of hardware-specific tuning techniques that can capture all of these aspects of efficient database design. Voodoo consists of an intermediate vector algebra that abstracts away hardware specifics such as hierarchical caches, many-core architectures or SIMD instruction sets while still allowing frontend developers to optimize for them. Central to our approach is a novel technique called *control vectors* that expose parallelism in the program and data to the compiler, without the use of hardware-specific abstractions.

We showed that Voodoo can be used as an alternative backend for an existing system (MonetDB), and that it can match and even outperform previously proposed highly optimized in-memory databases. We also demonstrated that Voodoo makes it easy to tune programs and explore design alternatives for different hardware architectures.

# 9. REFERENCES


[1] D. Abadi, D. Myers, D. DeWitt, and S. Madden. Materialization strategies in a column-oriented dbms. In *ICDE 2007*. IEEE, 2007.

[2] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, 2009.

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

[4] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Orti. Evaluation and tuning of the level 3 cublas for graphics processors. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1), 1996.

[6] P. Boncz, T. Neumann, and O. Erling. Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark. In *TPC-TC*. Springer, 2013.

[7] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *CACM*, 12 2008.

[8] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita raced: metacompilation for declarative networks. *PVLDB*, 1(1), 2008.

[9] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Cetintemel, and S. Zdoni. Tupleware:" big" data, big analytics, small clusters. In *CIDR*, 2015.

[10] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1), 1998.

[11] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *ISPASS '11*. IEEE, 2011.

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

[13] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *VLDB*, 2013.

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

[15] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[16] M. Liu, Z. G. Ives, and B. T. Loo. Enabling incremental query re-optimization. In *SIGMOD*, 2016.

[17] J. Malcolm, P. Yalamanchili, C. McClanahan, V. Venugopalakrishnan, K. Patel, and J. Melonakos. Arrayfire: a gpu acceleration platform. In *SPIE Defense, Security, and Sensing*, 2012.

[18] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9), 2011.

[19] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, et al. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *CGO*, 2011.

[20] H. Nguyen. *Gpu gems 3*. Addison-Wesley Professional, 2007.

[21] H. Pirk et al. Cpu and cache efficient management of memory-resident databases. In *ICDE*, 2013.

[22] H. Pirk, S. Manegold, and M. L. Kersten. Waste not...efficient co-processing of relational data. In *ICDE 2014*, pages –. IEEE, April 2014.

[23] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo - portable database performance on modern hardware. Technical report, MIT CSAIL, 2016.

[24] H. Pirk, E. Petraki, S. Idreos, S. Manegold, and M. Kersten. Database cracking: fancy scan, not poor man's sort! In *DaMoN*. ACM, 2014.

[25] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking simd vectorization for in-memory databases. In *SIGMOD 2015*. ACM, 2015.

[26] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, et al. Db2 with blu acceleration: So much more than just a column store. *PVLDB*, 6(11), 2013.

[27] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[28] K. A. Ross. Selection conditions in main memory. *ACM Trans. Database Syst.*, 29(1), Mar. 2004.

[29] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *SOSP*. ACM, 2013.

[30] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM TECS*, 13:134, 2014.

[31] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*. Springer, 2014.

[32] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. Optimizing data warehousing applications for gpus using kernel fusion/fission. In *IPDPSW*. IEEE, 2012.

[33] M. Zukowski, P. Boncz, N. Nes, and S. Héman. Monetdb/x100-a dbms in the cpu cache. *IEEE Data Engineering Bulletin*, 1001:17, 2005.

[34] M. Zukowski, N. Nes, and P. Boncz. DSM vs. NSM: CPU Performance Tradeoffs in Block-oriented Query Processing. In *DaMoN 08*, 2008.