The Operator Variant Selection Problem on Heterogeneous Hardware

Viktor Rosenfeld, Max Heimel, Christoph Viebig, Volker Markl

Technische Universität Berlin, Germany

firstname.lastname@tu-berlin.de

ABSTRACT

With the ongoing trend towards increased hardware heterogeneity, database systems will need to support many different processor architectures in order to fully exploit all available hardware configurations. However, different hardware architectures typically require different code optimizations, and the lack of performance portability in programming frameworks like OpenCL requires developers to hand-tune operator implementations. Even when only supporting a few architectures, these manual optimizations can drastically inflate the source code, resulting in extensive development and maintenance costs. Ideally, the database should be able to automatically generate and select optimal operator implementations from a single codebase.

In this paper, we discuss this *operator variant selection problem on heterogeneous hardware*, demonstrating that even for such simple operations as selection and aggregation, we can already generate thousands of variants. We provide an extensive experimental evaluation, demonstrating that picking the optimal variant is non-trivial and strongly dependent on the specific device. Finally, we discuss how to automatically select good variants at runtime and provide heuristic selection algorithms that work well in practice.

1. INTRODUCTION

The last decade has brought a tremendous diversification in the hardware landscape. Modern machines include multi-core CPUs, highly parallel GPUs, and sometimes even further co-processors, such as accelerator cards or FPGAs. This trend towards increased heterogeneity is only expected to increase in the future [4, 6]. Database systems will need to embrace and exploit this increased heterogeneity in order to keep up with the ever-growing performance requirements of the modern information society [13].

Traditionally, database systems targeting heterogeneous hardware have relied on hand-written implementations of database operators for each target architecture [5, 12, 15, 19]. Hand-written code can be highly optimized, but it also incurs a maintenance and development cost as the number of supported architectures increases. An alternative option is to avoid hardware-specific code, and implement the operators using a heterogeneous parallel programming framework such as OpenCL [29], offloading the creation of device-specific code to a vendor-provided driver. The feasibility of this *hardware-oblivious* approach has been demonstrated by Ocelot [13], an extension of MonetDB [2] that achieves competitive performance across devices from a single, abstract OpenCL codebase.

A major problem of hardware-oblivious databases is the missing performance portability: to reach peak performance in OpenCL, devices typically require very specific code optimizations which do not translate well to others [26]. This leads to a problem: in order to build a high-performance, hardware-oblivious database engine, we need to manually develop and maintain several different *operator variants*, each targeting different device properties. This approach obviously contradicts the initial goal of reducing development overhead and is therefore undesirable.

A potentially better solution is to employ modern software engineering techniques to create operator implementations that are tailored to a specific hardware architecture from a single, expressive codebase [8]. Even then, given the wide range of devices and their particular characteristics, it is necessary to automatically fine-tune operator implementations to the specific device they are running on. Finding this combination of different automatic mechanisms to achieve performance-portable, hardware-oblivious database operators is what we call the *operator variant selection problem on heterogeneous hardware*.

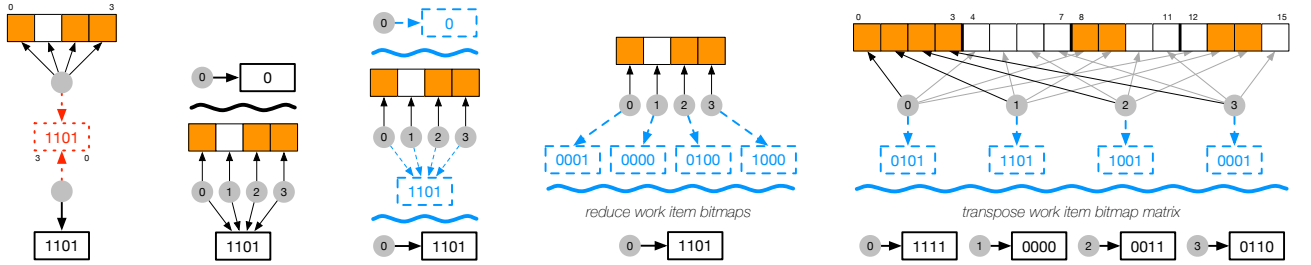
In this paper, we experimentally motivate this problem, demonstrate it based on the concrete example of the selection and aggregation operators, and provide initial results towards solving it. Specifically, we make the following contributions:

- We perform an extensive experimental evaluation of different implementations for two database operators, selection and aggregation, to illustrate the diversity of hardware architectures with regard to performance optimizations (Section 2).
- We extend Micro Adaptivity [24] to handle a large variant space, and use it to automatically pick a nearly optimal operator variant for a specific device (Section 3).

Finally, we conclude the paper with a discussion of related work in Section 4, and present current challenges as well as our next steps towards tackling the operator variant selection problem in Section 5.

 Articles from this volume were invited to present their results at ADMS'15, a workshop co-located with the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.

.



(a) Sequential: independent work items

(b) Interleaved: `atomic_or` on global memory

(c) Interleaved: `atomic_or` on local memory

(d) Interleaved: reduction of bitmaps in local memory

(e) Interleaved: matrix transpose in local memory

Figure 1: Selection kernel variants used in the experiments. The row of boxes denote the input column; filled boxes evaluate to true. Circles denote threads. Arrows leading upwards from threads illustrate reads, arrows leading downwards or to the right illustrate writes. Line styles indicate storage locations: solid is global memory, dashed is local, and dotted are thread-private registers. Wavy lines illustrate memory barriers.

In order to simplify repeating our experiments and to facilitate a better exchange with other researchers, we made the source code for all of our experiments, including the variant generator, the variant selection algorithms, and all benchmarks, available at `http://goo.gl/zF8U1W`.

2. OPERATOR VARIANTS

In this section, we describe how a large number of database operator implementations can be derived from a small number of basic code templates, by changing certain implementation details which do not change the semantics of the algorithm. We present two use cases, a selection kernel and an aggregation kernel. Throughout the paper, these serve as examples to illustrate performance characteristics of different devices, and to evaluate the influence of different implementation parameters on the performance of database operators on heterogeneous hardware.

2.1 Use Case 1: Selection

Our first use case is a selection kernel, which evaluates a predicate on a column and returns a bitmap to indicate which tuples satisfy the predicate.

2.1.1 Basic Variants

In Figure 1, we illustrate the primary distinctive features of the different variants: how they access memory and construct elements of the result bitmap. Note that while input data is read from left to right, bitmaps are written from right to left. For brevity, we only illustrate the case of four-bit bitmaps.

Figure 1a shows the simplest possible variant, which we call *sequential*: each thread evaluates the predicate for a few consecutive values, creates the resulting bitmap element in a register, and writes it out to the corresponding global memory address. In contrast to this, the other variants use an interleaved access pattern, where neighboring threads evaluate the predicate on neighboring values. Depending on the underlying hardware, this pattern can be more efficient than a sequential one.

A straightforward interleaved variant is shown in Figure 1b: threads in a work group create the output bitmap by directly setting the corresponding bits in parallel using atomic operations. In order to ensure correctness, the work group has to first zero-initialize the result memory, requiring synchronization via a global memory barrier. Figure 1c shows a slightly modified version of this variant, where the threads construct the bitmap in local memory. After synchronizing via a local memory barrier, the work group then copies the result to its final position in global memory. We call these two variants *interleaved-atomic-global* and *interleaved-atomic-local*. A major problem of these variants is their reliance on atomic operations. For predicates that are satisfied by a large percentage of the input data, this results in severe thread contention, as all atomic operations on the same address are serialized by the hardware.

A variant that does not use atomics is *interleaved-reduce*, shown in Figure 1d. Each thread evaluates the predicate for one value, creates a bitmap element with the corresponding bit set, and writes it to local memory. The work group then cooperates to create the result bitmap by using a parallel aggregation algorithm, as described by Horn [14]. A drawback of *interleaved-reduce* is its wastefulness with regard to local memory: each thread only sets one bit, but stores a full bitmap element.

The final two variants, which are summarized in Figure 1e, work similarly but utilize resources more efficiently. Like in *sequential*, each thread first evaluates the predicate for multiple tuples, creating a bitmap element in a register. However, due to the interleaved memory access, the bit pattern in these elements will be interleaved as well, forcing us to restore the correct bit order before writing to global memory. Essentially, the intermediate bitmaps can be interpreted as a matrix which we need to transpose, and the variants differ in how this happens. In *interleaved-collect*, each thread builds one element of the result in a register by subsequently collecting the required bits using bit masks and bit shifts. This scales linearly with the number of matrix elements and does not require memory barriers. In contrast, *interleaved-transpose* uses the full work group to cooperatively transpose increasingly larger tiles of the interleaved bitmap elements in local memory. While this algorithm scales logarithmically, it also requires additional memory barriers.

2.1.2 Implementation Parameters

The basic kernel variants described in the previous sections should be seen as "code templates" that can be modified along the following dimensions to fine-tune the implementation for a given hardware architecture:

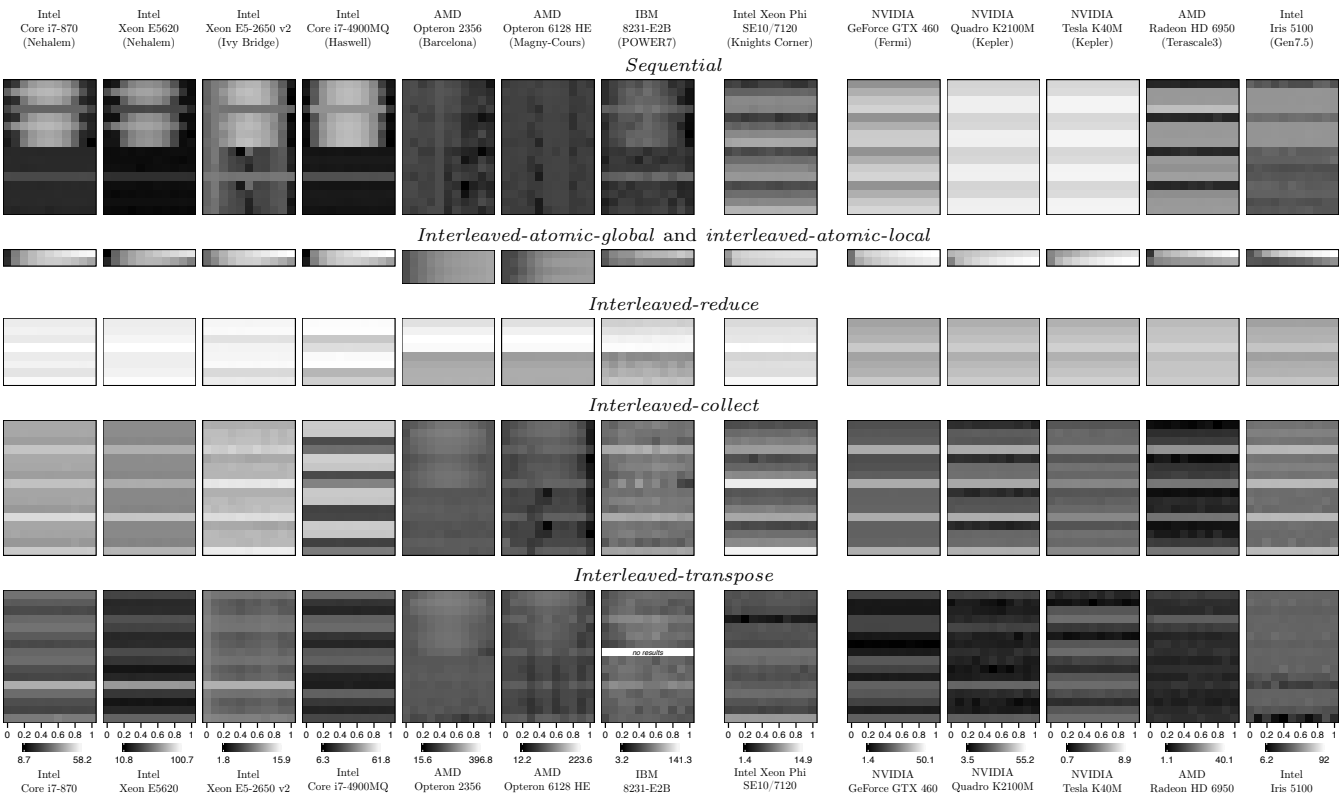


Figure 2: Heatmaps illustrating the relative performance of selection kernel variants across a range of selectivities, plotted on the x-axis, on different devices. Each block represents a basic variant, with the interleaved-atomic-global and interleaved-atomic-local variants combined into one block. Within a block, each row represents a combination of the parameters result type, loop unrolling, and predication. For example, the top row in a block shows a 8-bit branched kernel, whereas the bottom row shows an 64-bit unrolled, predicated kernel. Row pixels represent the runtime of the fastest combination of the parameters work group size and elements per thread. Darker colors indicate faster execution. Runtimes are normalized for each device, i.e., for each device, black indicates the fastest kernel. At the bottom, the range of absolute runtimes is shown in milliseconds.

Result type While we illustrated the base variants for bitmap elements containing four bits, in reality we create bitmaps at the granularity of 8, 16, 32, or 64 bits.

Loop unrolling We optionally remove for-loops entirely by replicating the loop body the required number of times.

Predication In order to avoid branch mispredictions, we can set bits unconditionally using predication when evaluating the predicate.

Furthermore, we can use the following two parameters to vary the workload of each thread:

Work group size The work group size (or local size) determines how many threads are active in each group. Tuning this parameter allows us to better utilize the parallelism of the underlying hardware.

Elements per thread In order to reduce the overhead for each thread, the variants can be modified to produce more than one result bitmap element by sequentially processing multiple input values.

Another possible tuning parameter is to use SIMD capabilities of CPUs. However, for the selection kernel, we rely on auto-vectorization by the OpenCL compiler.

Based on the six base variants and these parameters we wrote a parameterized code generator that can generate over six thousand different variants for the selection operator. The interested reader can find source examples of the basic variants in Appendix A.

2.1.3 Selection Kernel Performance

In a first experiment, we ran a simple range selection over an array of 32 million random integers. Figure 2 summarizes the runtimes of the different selection variants at different selectivity values, normalized for each device, on a number of CPUs and GPUs from different manufacturers, and a Xeon Phi accelerator card. Note that interleaved-atomic kernels are only supported for a bitmap granularity of 32 bits and, for the AMD CPUs, of 64 bits. Furthermore, there are no predicated or unrolled interleaved-atomic kernels. Similarly, there are no branched interleaved-reduce variants. Also note that there are no results for the unrolled 64-bit interleave-transpose kernel on the IBM CPU because this variant did not evaluate the predicate correctly.

The heatmaps clearly show the diversity in the behavior of different devices. In particular, they show diversity:

Between device classes CPUs generally prefer a predicated sequential kernel, whereas on GPUs and, surprisingly, on the Xeon Phi, the interleaved-transpose or interleaved-collect kernels are faster.

Between different manufacturers The heatmaps of the sequential, interleave-collect, and interleave-transpose

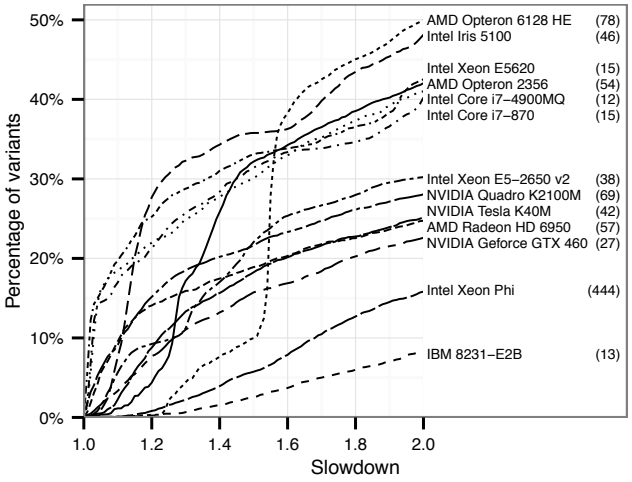


Figure 3: Cumulative distribution of the runtime of selection kernel variants at selectivity 0.5. Runtimes are expressed as the slowdown compared to the fastest variant for each device, capped at a factor of two. The number in parentheses after each device shows the maximum slowdown.

kernels on AMD and IBM CPUs are almost uniform, although they do show a small performance slowdown due to branch mispredictions for some variants. Conversely, on Intel CPUs, the performance of these kernels strongly depends on the parameter combination.

Between individual devices Even devices of the same architecture behave differently. For example, the Tesla K40M strongly prefers a branched 16-bit interleaved-transpose kernel. On the Quadro K2100M, also a Kepler device, other kernel variants are competitive, including 8-bit interleaved-collect kernels.

For many of the devices we tested, there are quite a number of competitive kernels. These can be kernels that have different code parameters, i.e., from different rows in Figure 2, or different workload parameters, i.e., from the same row in Figure 2. In Figure 3 we show the percentage of variants that are at most two times slower than the fastest variant for each device at selectivity 0.5. Especially for the Intel Xeon E5620, the Core i7-4900MQ, and the Core i7-870, there are many very competitive kernel variants. More than 10% are only minimally slower, i.e., about 3%, than the fastest variant. Additionally, for the Intel Iris 5100, the NVIDIA Quadro K2100M, and the AMD Radeon HD 6950 more than 10% of the kernels are less than 20% slower than the fastest. On the other hand, for the Intel Xeon Phi and the IBM CPU there are only few competitive kernels. Of particular note is also the large increase in the middle of curve for the AMD Opteron 6128 HE. Excluding the Xeon Phi, for which the largest slowdown is quite large, the maximum slowdown is in the range between 12, for the Core i7-4900MQ, and 78, for the Opteron 6128 HE, with a median of 40.

Furthermore, we can identify for most devices a kernel variant that performs particularly well. For example, 12 different kernels outperform the others in the 44 experiments at different selectivities on the four Intel CPUs. However, as Table 1 shows, a particular variant, the unrolled, predicated

Table 1: Fastest selection kernels on Intel CPUs and how often they outperform other variants. P stands for a predicated, U for an unrolled kernel.

Variant	Core i7-870	Xeon E5620	Xeon E5-2650 v2	Core i7-4700MQ	Total
64-bit sequential PU	8	6	4	9	27
64-bit sequential U	3			1	4
8-bit sequential P			3		3
32-bit sequential			2		2
eight other variants		*5*	*2*	*1*	*8*

Table 2: Fastest selection kernels on GPUs and Xeon Phi.

Device	Variant	Elements per thread	Local size
GeForce GTX 460	32-bit transpose U	1	256
Quadro K2100M	16-bit transpose (P/PU)	1/2/4	128
Tesla K40M	16-bit transpose (U)	1	128
Radeon HD 6950	8-bit collect (U)	1	128
Iris 5100	64-bit transpose P/PU	1024	64/128
Xeon Phi SE10/7120	64-bit transpose	1/2/4	64/128

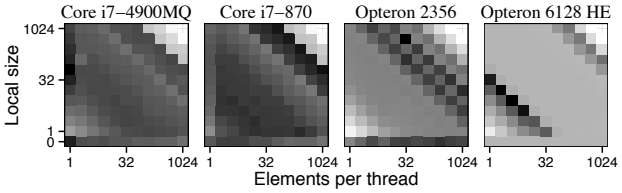


Figure 5: Normalized runtime of the unrolled, predicated 64-bit sequential kernel on CPUs for different combinations of the parameters work group size and elements per thread.

64-bit sequential kernel, is fastest more than 60% of the time. Furthermore, the Intel CPUs are more or less sensitive to the work group size and the number of elements processed by each thread, and some CPUs prefer a particular combination of these parameters, as shown by the black hotspots in the heatmaps in Figure 5. On the Core i7-4900MQ, a workload that processes only one element per thread at a work group size set by the OpenCL runtime, i.e., 0, or at around 64, is the fastest in most cases. However, the Core i7-870 does not perform well when the work group size is set by the OpenCL runtime. Instead, it prefers a combination lying on the darkly shaded line in the upper right half of its heatmap. AMD processors are similarly sensitive to the kernel workload as their heatmaps in Figure 5 show.

Finally, the behavior of the GPUs and the Xeon Phi accelerator, which we summarize in Table 2, is strikingly consistent. In our experiments, a single base kernel variant, with a particular result type and a narrow range of the workload parameters, outperforms all other variants on each device. The only variation we observe is that sometimes a predicated and/or unrolled variant is fastest. On the GeForce GTX 460, the same kernel variant is selected across the entire selectivity range. Note that the Intel Iris 5100 prefers a drastically different local size than the other GPUs.

While the absolute differences between fast kernel variants are very small, for some devices, they are also quite stable. We therefore believe that these parameter configurations indeed constitute a performance sweet spot.

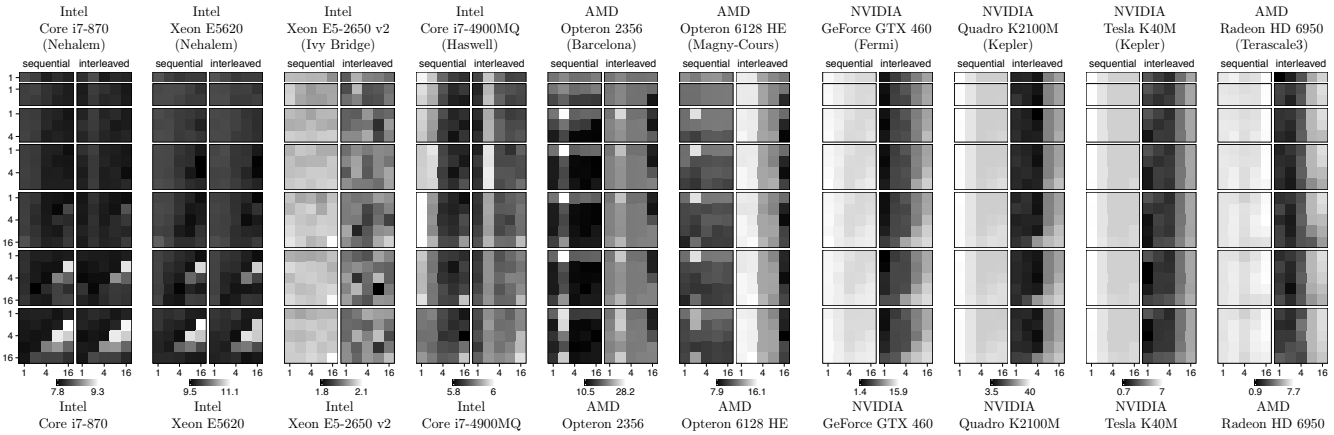


Figure 4: Heatmaps illustrating the relative performance of sequential and interleaved aggregation kernel variants. From top to bottom, each box represents a doubling of the unroll count from one to 64. Within a box, each row represents a doubling of the attainable instruction level parallelism, as indicated on the y-axis. The boxes have different heights because the instruction level parallelism can only be increased together with the loop unroll count. Each column represents a doubling of the vector width, as indicated on the x-axis. Each pixel represents the fastest combination of the parameters work group size and number of work groups, with darker colors indicating a faster runtime. Runtimes are normalized for each device, i.e., for each device, black indicates the fastest kernel. At the bottom, the range of absolute runtimes is shown in milliseconds.

2.2 Use Case 2: Aggregation

Our second use case is an aggregation kernel. Whereas the selection kernel maps each input value to an output value in the result bitmap, and is therefore trivial to parallelize, the aggregation kernel reduces an input column to a single value. Consequently, although we can aggregate distinct parts of the input column in parallel, the final aggregation of the intermediate values needs be performed by a single work group. Here, we focus on the first part that can be parallized and limit the number of intermediate results produced to a few dozen.

2.2.1 Basic Algorithm

A basic parallel reduction strategy for GPUs, in which the threads of a work group cooperate to produce an intermediate result, is described by Horn [14]. The number of values that can be processed by a work group is limited by the maximum work group size. On the devices we tested, this size ranges from 256 threads, for the AMD Radeon HD 6950, to 8192 threads, for the Intel CPUs. Consequently, even for moderately sized input columns, e.g., containing more than 65k values when evaluated on the AMD Radeon HD 6950, this strategy requires multiple passes. Each pass has to be setup and controlled by the host code, which is why we want to limit the computation of intermediate values to a single pass. We can achieve this by locally aggregating a number of input values in each thread.

Specifically, our aggregation kernel implements the following basic algorithm. The input column is partitioned hierarchically into two levels: top-level partitions are aggregated by work groups and second-level partitions by individual threads. Each thread writes its intermediate result into a buffer in local memory. Afterwards, the threads cooperate to produce an intermediate result for each work group, using a parallel reduction strategy [14]. Finally, the intermediates are aggregated into a single value using atomic operations.

2.2.2 Implementation parameters

We can modify this basic algorithm along the following dimensions to fine-tune it:

Memory access In the *sequential* variant, each thread aggregates a consecutive partition of the input column. Conversely, in the *interleaved* variant, the partitions processed by the threads of a work group are interleaved with each other to exploit coalesced memory transfers on GPUs.

Loop unrolling Instead of completely eliminating for-loops, as we do in the selection kernel, we replicate the loop body a configurable number of times and modify the loop condition accordingly.

Instruction level parallelism (ILP) Aggregating the values of a partition in multiple thread-private registers reduces data dependencies between loop iterations. CPUs can then use out-of-order execution to increase instruction throughput.

Vector width We can exploit SIMD instructions to aggregate multiple values at once by packing them in OpenCL's vector data types.

Furthermore, we vary the workload of each thread in the same way as for the selection kernel, by changing the work group size and the size of the partition processed by each thread. Because we want to limit the number of intermediate results, we fix the number of work groups to a low multiple of the device's compute units and derive the partition size instead of setting it explicitly. Note that when we set the work group size to one, the algorithm reverts to a straightforward sequential aggregation, regardless of the memory access pattern. Similarly, if we set the partition size to one, the algorithm reverts to a parallel reduction strategy [14].

Based on these parameters, we currently generate up about 15000 different kernels.

2.2.3 Aggregation Kernel Performance

Again, we can use the different aggregation kernels to illustrate the diversity in the hardware landscape. Figure 4 summarizes the runtime of various aggregation kernels that

Table 3: Fastest aggregation kernels with values for the parameters unroll count (U), ILP (I), vector width (V), work groups (W), and local size (L).

Device	Kernel	U	I	V	W	L
Intel Core i7-870	sequential	32	8	2	8	1
Intel Xeon E5620	sequential	8	4	16	8	1
Intel Xeon E5-2650 v2	interleaved	32	8	8	128	1
Intel Core i7-4900MQ	sequential	64	4	16	8	4
AMD Opteron 2356	sequential	32	8	8	8	8
AMD Opteron 6128 HE	interleaved	64	4	16	32	4
NVIDIA Geforce GTX 460	interleaved	32	1	1	128	512
NVIDIA Quadro K2100M	interleaved	4	1	4	8	256
NVIDIA Tesla K40M	interleaved	64	4	1	128	512
AMD Radeon HD 6950	interleaved	1	1	1	64	256

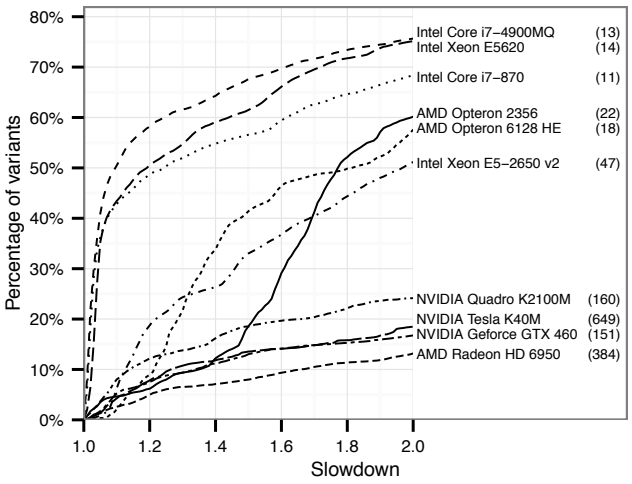


Figure 6: Cumulative distribution of the runtime of aggregation kernel variants. Runtimes are expressed as the slowdown compared to the fastest variant for each device, capped at a factor of two. The number in parentheses after each device shows the maximum slowdown.

compute the sum of a column of 32 million integers. For each device, the fastest variant is shown in Table 3.

Similarly to the selection kernel heatmaps in the previous section, we can make out patterns separating devices of different classes, manufacturers, and architectures. For example, the heatmaps for the two Intel Nehalem CPUs are almost identical, and the heatmaps for the two AMD CPUs also show a similar pattern, especially for sequential kernels.

In general, CPUs prefer a sequential kernel with high values for the code parameters unroll count, ILP, and vector width, and low values for the workload parameters. The Xeon E5-2650 v2 and the Opteron 6128 HE appear to be exceptions, both prefering an interleaved kernel. However, for the Xeon E5-2650 v2, the local size is one, which turns this variant into a sequential kernel as described above. Note that the minimum number of workloads in this experiment was eight because the CPUs have at least eight cores; the Xeon E5-2650 v2 has 32 and the Opteron 6128 HE has 16.

Conversely, GPUs prefer an interleaved kernel with low values for the parameters ILP and vector width, and high values for the workload parameters. The unroll count appears to have little influence, except for the Radeon HD 6950, which prefers an unroll count of one.

Finally, in Figure 6 we show the percentage of those aggregation kernels that are at most two times slower than the fastest variant on each device. Similarly to the selection use case, there are many competitive kernels on the Intel CPUs. Indeed, on the Core i7-4900MQ, the Xeon E5620, and the Core i7-870, more than 40% are just 10% slower than the fastest variant. Conversely, on the GPUs we tested, the number of competitive kernels is fairly small, and only between 10% and 25% of them are at most two times slower than the fastest.

3. VARIANT SELECTION

As we have seen in the previous section, we can easily generate thousands of variants for simple database operators by modifying a few implementation or workload parameters. The large number of possible parameter combinations leads to an interesting question: How do we select the best, or at least a near-optimal, variant for a given device?

A straightforward approach is to run a training phase during database setup in which we evaluate all possible variants to identify the fastest one. However, this method is not feasible for a number of reasons. First, it would be prohibitively time-consuming. The exploration of just one operator, e.g., our exhaustive evaluation of a few thousand selection variants in Section 2, can easily take hours on a single device. Second, assuming we can search the full variant space reasonably fast, we still face potential data and workload dependencies. Therefore, the initial exploration should be based on a representative query workload, which is often hard to facilitate. Furthermore, even for a single query, there might not exist a single optimal operator variant [24]. Third, in cloud-based database-as-a-service applications, the hardware running the database might change at any given moment because of machine migrations. Therefore, we require a flexible strategy that allows us to quickly adapt the selected variant to the new environment.

Given these limitations of offline strategies, we decided to investigate online methods, which rely on performance feedback generated during normal operations to select operator variants. In this section, we report on our first steps towards achieving this task. In particular, we introduce a generic, hardware-oblivious operator variant learning framework and compare a variety of different search strategies to efficiently explore the vast universe of possible operator implementations. While these strategies do not guarantee finding the optimal variant, we found that in practice even fairly simple strategies typically converge quickly towards close-to-optimal operator variants within only a handful of queries.

3.1 Micro Adaptivity

Our learning framework builds upon Micro Adaptivity, an online learning strategy to choose optimal operator implementations proposed by Raducanu et al. [24]. Micro Adaptivity assumes a vectorized, or block-at-a-time, in-memory query processor [3], i.e., the query data is split into cache-sized chunks and a function implementing a database operator is called repeatedly on the chunks to evaluate the query. Furthermore, each database operator exists in multiple variants, which implement the same algorithm but differ in their implementation details, e.g., loop unrolling or vectorization. To find the optimal variant, Micro Adaptivity uses the *vw-greedy* algorithm [24]. When a database operator is called

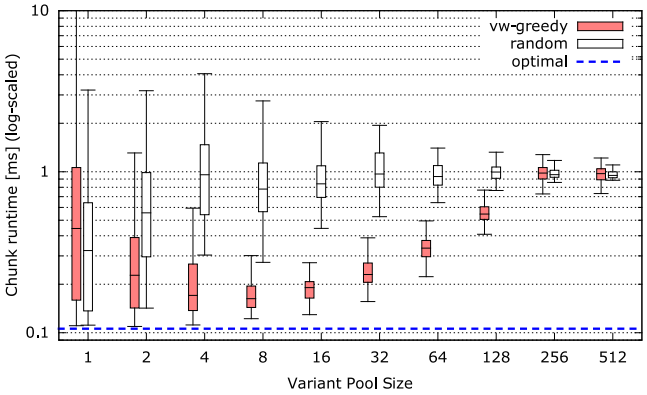


Figure 7: Query performance depending on pool size, using vw-greedy to pick kernel variants at runtime compared to using a random variant from the pool for each chunk.

for the first time, i.e., at the beginning of the first query, each variant is called on a number of chunks to learn about its performance. The algorithm then enters the *exploitation* phase, choosing for each of the remaining chunks the fastest variant to process them. Over time, the knowledge about the performance of variants that are not chosen becomes stale and, as the data and workload changes, a chosen variant might no longer be optimal. To adapt, vw-greedy periodically enters an *exploration* phase, choosing a random variant and evaluating its current performance on a small number of chunks. Afterwards, it uses its updated knowledge about variant performance to choose optimal variants in the next exploitation phase.

vw-greedy has three beneficial properties which make it suitable as a foundation for our learning framework. First, it has very low overhead because its book-keeping costs are amortized over the tuples contained in a chunk. Second, each chunk provides performance information about the currently chosen variant and the periodic exploration phase allows it to update its knowledge about variants that were not chosen recently. Consequently, vw-greedy can quickly adapt to data and workload changes. Third, slow variants will only affect the performance of a few chunks during exploration instead of slowing down the entire query.

3.2 Variant Pool Size Limitation

By itself, vw-greedy is only able to handle cases where the number of existing variants is comparatively small, as shown in Figure 7. For this experiment, we evaluated a selection query over a 16 GB column partitioned into 1024 chunks of 16 MB each, using (randomly selected) variant pools of increasing sizes. For each pool size, we repeat the experiment 300 times, and, for each run, we construct a new variant pool by randomly selecting variants from the universe of about 6000 selection kernels described in Section 2.

The box plots show the distribution of the average runtime per chunk using two different strategies. In the *random* strategy, we simply pick a random variant from the pool for each chunk. A pool of size one can contain either a fast, a slow, or an average variant, resulting in the large spread of the measured chunk runtimes. As the pool size increases, the pool will contain a mix of fast, slow, and average variants. Thus, the spread of the runtime distribution contracts and converges towards the mean runtime of all the variants in the universe.

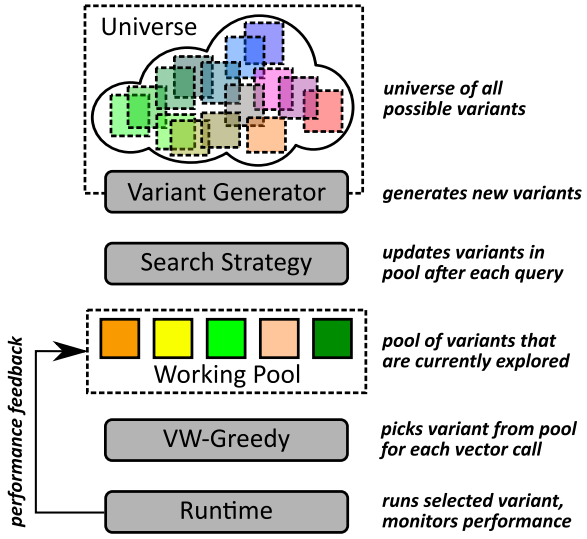


Figure 8: Overview of the Variant Learning Framework.

For the other strategy, we use vw-greedy to select an optimal variant for each chunk. We can make two observations: First, vw-greedy reduces the influence of bad variants in the pool, i.e., the spread of the runtime distribution is contracted even further than using the random strategy with increasing pool sizes. Consequently, the average runtime approaches the performance of the optimal variant, as indicated by the dashed line. However, vw-greedy also limits the practical size of the variant pool to around eight to sixteen variants. With larger pool sizes, the initial exploration phase, used to determine the performance of each variant in the pool, dominates query execution time. Indeed, in this experiment, we use two chunks to measure variant performance. Consequently, at a pool size of 512, the initial exploration phase comprises all of the 1024 chunks, and there are no chunks left for the exploitation phase.

Thus, the main challenge our learning framework needs to solve is how to select a comparatively small variant pool from a universe that can potentially contain thousands of possible variants.

3.3 Learning Framework Overview

Figure 8 gives an overview of our learning framework. A parameterized *variant generator* is used to produce callable variants of an operator for a given device. Individual operator variants are identified through a *genome*, which is a pre-defined collection of variables, and allowed values for them, that modify the behavior of the code generator. For example, Listing 1 shows the genome of the selection operator from Section 2.

```
kernel_type: { sequential, interleaved_reduce,
 interleaved_transpose, interleaved_atomic_global,
 interleaved_atomic_local, interleaved_collect }
result_type: { char, short, int, long }
branched: { true, false }
unrolled: { true, false }
lements_per_thread: { 1, 2, 4, 8, 16, 32, 64, 128, 256, ... }
local_size: { 0, 1, 2, 4, 8, 16, 32, 64, ... }
```

Listing 1: Genome definition for the selection kernel.

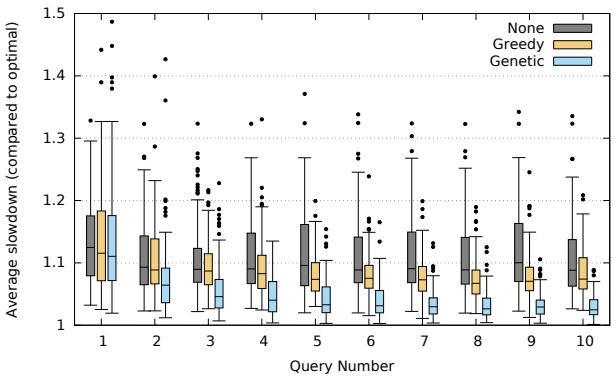


Figure 9: Influence of different search strategies on the Intel Xeon E5620, a device with many competitive variants.

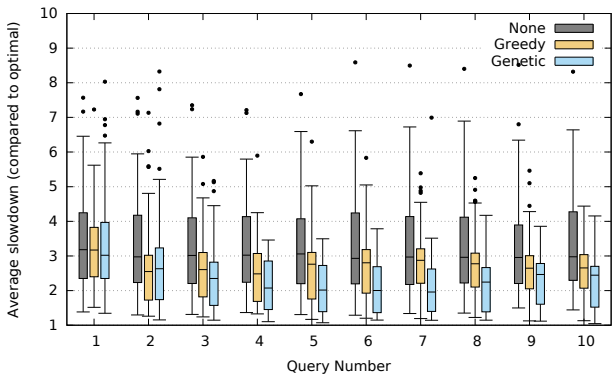


Figure 10: Influence of different search strategies on the Intel Xeon Phi, a device with few competitive variants.

Instead of instantiating the full universe of all possible implementations, we only use a small *working pool* of around eight to sixteen active variants during the evaluation of a query. Queries are evaluated using a vectorized runtime [3] which allows us to make fine-grained performance measurements of the variants, and then use vw-greedy [24] to pick an optimal variant for each chunk from the working pool.

In between queries, or after a fixed number of queries, we use a *search strategy* to update the pool based on the collected performance feedback. The search strategy replaces badly performing variants in the pool by newly selected ones. This process continuously improves the quality of the variants in the working pool, bringing the overall performance of the system closer to the optimum in each step. Our learning framework is fairly flexible, allowing us to plug-in different search strategies that differ in how they decide which variants to pick next.

3.4 Search Strategies

As discussed in the previous section, the goal of a search strategy is to periodically update the active variant based on collected performance feedback by replacing underperforming variants with newly selected ones. At the moment, we use the following two strategies in our framework:

Greedy Keep the two fastest variants of the current pool. Replace all other variants by newly selected random ones. Random variants are generated by randomly picking values for all variables in the genome. This is our simplest strategy, essentially corresponding to a random walk through the variant universe.

Genetic Keep the two fastest variants of the current pool. Replace all other variants by following a genetic propagation protocol that generates new variants by combining the genetic features of two parents from the current pool. Parents are selected randomly, with probability proportional to their observed performance, i.e., faster variants have a higher chance of passing on their configurations. In order to add genetic diversity and avoid getting stuck in local minima, we also introduce mutations to the new genomes with low probability.

Both search strategies are initialized with a pool consisting of randomly chosen variants.

3.5 Experimental Evaluation

To evaluate our learning framework, we ran a series of selection kernels and measured how the query runtime changes over time. In particular, each experimental series consists of ten random selection queries with a fixed selectivity of 0.5. The queries scan a 16 GB column in 1024 chunks of 16 MB each, and we measure the average runtime per chunk for each query. After each query, we use the selected search strategy to evolve the current working pool, which is set to a size of eight variants. Each experimental series is repeated 100 times to control for randomization effects, resetting the working pool before each repetition. In addition to the two search strategies, we also use the following baseline: in the *None* strategy, the working pool is initialized at random and then remains constant for all ten queries.

We picked two devices to run the experiment on: the Intel Xeon E5620 CPU and the Intel Xeon Phi accelerator card. We chose these two devices because they have very different performance distributions across the variants. Looking at Figure 3, we can see that for the Xeon E5620, around 40% of all variants fall within a factor of two of the optimal kernel variant, while on the Xeon Phi only 15% of all variants fall within this bound. This means that the E5620 is a much easier target to optimize for, since we have a 40% chance of picking a good kernel at random. Therefore, we expect to clearly see different learning behaviour between the two devices. Figure 9 shows the results for the Xeon E5620 and Figure 10 for the Xeon Phi. The runtimes are reported as the relative slowdown compared to the respective fastest variant, which we determined in Section 2.

The experiment shows a few interesting things. Let us first take a closer look at the Xeon E5620. Even without using any search strategy, i.e., the None baseline, vw-greedy alone produces a very competitive performance. This behavior is caused by the performance distribution of the selection variants for the Xeon. As discussed, 40% of all variants fall within a factor two of the optimal one, and 30% fall within a factor of 1.5. Consequently, a randomly initialized work pool of size eight, as we use in this experiment, has a 94% chance of containing a variant that is at most 1.5 times slower than the optimal one. Since vw-greedy masks the occurrence of slow variants in the working pool, even the None strategy produces competitive results.

Building upon the behavior of vw-greedy during the execution of a single query, the Greedy search strategy improves

query runtimes even further. Figure 3 shows that about 15% of the variants are within a factor of 1.1 of the fastest variant on the Xeon. After each query, six of the eight variants in the pool are replaced, and we have a chanche of about 60% to choose one of those fast variants. As a result, after five queries, the working pool contains a variant that is within a factor of 1.1 of the optimal variant in 75% of the query series in our experiment. The Genetic search strategy achieves an even stronger convergence towards the optimum. Here, after two queries the runtime is within a factor of 1.1 of the optimum in 75% percent of the time, and after seven queries we reach a factor of 1.05.

On the Xeon Phi, vw-greedy alone cannot achieve as good a performance as on the Xeon CPU. Here, only about 6% of all variants fall within a factor of 1.5 of the optimal one, meaning there is only around a 40% chance that a random, eight-variant pool will contain a good variant. Furthermore, the worst variant on the Xeon Phi is 444 times slower than the optimal. Compared to other processors, there are not only few good variants but also some *very* bad variants, which are likely to cause a strong performance degradation during the exploration phase.

While the search strategies are able to improve the performance on the Xeon Phi, it is not as impressive as on the Xeon CPU. After three queries, the Greedy strategy is at most three times slower than the optimal variant in 75% of the queries series. The Genetic strategy shows a slightly stronger benefit and is able to improve upon the None baseline by about 40%. Interestingly, the best median performance of Genetic strategy, a slowdown of factor two, is achieved after four queries, but becomes worse after eight queries. This is most likely due to the performance degradation caused by very bad variants on the Xeon Phi.

4. RELATED WORK

Auto-tuning has been used prominently to create highly-optimized linear algebra libraries, e.g., ATLAS [30]. During installation, a large space of auto-generated variants of basic linear algebra subprograms (BLAS) is evaluated in order to find optimal implementations for a particular environment. Similarly, MAGMA [1] uses auto-tuning to not only find optimal implementations on CUDA GPUs but also to keep up with rapidly changing hardware characteristics [17].

Fabeiro et al. employ a genetic algorithm to adapt the configuration parameters of parameterized matrix multiplication kernels written in OpenCL for various device architectures [10]. They note the importance of optimizing multiple parameters simultaneously as setting a single parameter to the learned optimal value can actually decrease performance.

Seo et al. investigate the influence of the work group size on OpenCL kernel performance and describe a model-based algorithm to dynamically pick a work group size that minimizes cache misses and improves load balancing on multi-core CPUs [27].

Due to its simplicity and relative importance, the table scan is probably one of the most extensively studied relational operators [7, 25, 28, 18, 23]. Optimizing its execution on modern hardware can lead to significant performance benefits because it is the first operator executed in a query plan and processes large amounts of data.

Ross developed a cost model to decide between branched or predicated evaluation of compound select conditions in order to reduce branch misprediction on CPUs [25]. Similarly, Sitaridi et al. studied this problem for GPUs where the main challenge is reduced memory bandwidth due to thread divergence [28].

Broneske et al. evaluated the impact of combining different optimization techniques to the scan operator on multiple machines, showing that the best combination depends both on the selectivity of the workload and the targeted processor [7]. They argue that a cost model for multiple optimizations is not feasible because complex interactions between different optimizations make it difficult, if not impossible, to infer an optimal combination for a processor.

BitWeaving [18] is a CPU-based scan algorithm that uses bit-packing to evaluate a predicate on multiple input values at once; BitWarp [23] extends this technique to GPUs, treating the data accessed concurrently by multiple work group threads as a single wide word. Polychroniou et al. study the influence of vectorization on database operations the Xeon Phi and CPUs [22]. Pirk et al. devise a set of micro benchmarks to analyze and compare the performance of common data management operations on GPUs and the Xeon Phi [21].

5. CONCLUSION & FUTURE WORK

In this paper we exemplified the operator variant selection problem on heterogeneous hardware for two simple operators: selection and aggregation. We demonstrated that even for such simple operations, we can easily generate thousands of different code variants. Based on an extensive experimental evaluation across multiple different hardware architectures, we showed that selecting the optimal variant out of those large number of alternatives is non-trivial and strongly dependent on the current device. Furthermore, we discussed how to algorithmically approach the variant selection problem. Our primary findings are that a vectorized database runtime [3] together with vw-greedy, a multi-armed bandit algorithm to select optimal operator implementations [24], is a good starting point to select optimal algorithm variants for a given device. However, we also showed that we have to limit the pool size given to vw-greedy to achieve good results and demonstrated how simple search strategies, which periodically evolve the current set of selected variants based on collected performance feedback, can improve the selection quality even further.

This paper is a starting point in the investigation of the variant selection problem for heterogeneous hardware. Besides the obvious next step of extending our evaluation to more operators (join, sort, etc.), there are other major problems that have to be solved. Clearly, the biggest missing puzzle piece is how to automatically generate implementation variants for arbitrary operators in an efficient manner. For this paper, we used a hand-written code generator that uses string concatenations to produce different variants. Obviously, such an approach does not scale and contradicts our main goal of using a single performance-portable operator implementation in the database.

Ideally, we want to define data processing operators in an abstract fashion, and derive and generate possible variants algorithmically. A promising approach is to use a combination of high-level DSLs, as suggested by Broneske et al. [8], and code generation, as used in HyPer [20], Legobase [16], Delite [9], and the work by Haensch et al. [11].

Another important aspect for future work is to keep improving the variant selection framework and reduce the gap to the optimal variant. While our current methods already achieve fairly good results, there is obviously a lot of room left for improvement, especially for devices with few competitive variants. For the evaluation in this paper, we set the working pool size to eight variants, based on optimal results for the NVIDIA Tesla K40M GPU. We are currently exploring if the pool size should also be adjusted based on the number of competitive variants for each device.

At the moment, our methods are completely hardware-oblivious and do not assume any prior knowledge about the hardware environment. An almost self-evident alternative is to investigate if we can use simple and fast micro benchmarks to learn the characteristics of the underlying hardware, e.g., the preferred memory access pattern and vector width, and use this information to seed and drive our search strategies. Another interesting direction is to explore advanced search heuristics that help to further restrict the variant search space. This will be particularly important once we move to more complex operations, where the combinatorial blow-up of the number of potential variants will be much more pressing.

6. REFERENCES

[1] M. Baboulin et al. Enhancing the performance of dense linear algebra solvers on GPUs [in the MAGMA project]. *Poster at Super Computing*, 2008.

[2] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.

[3] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: hyper-pipelining query execution. In *CIDR*, 2005, pp. 225–237.

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

[5] S. Breß and G. Saake. Why it is time for a HyPE: a hybrid query processing engine for efficient GPU coprocessing in DBMS. *PVLDB*, 6(12):1398–1403, 2013.

[6] S. Breß et al. GPU-accelerated database systems: survey and open challenges. *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, 15:1–35, 2014.

[7] D. Broneske, S. Breß, and G. Saake. Database scan variants on modern CPUs: a performance study. In *IMDM*, 2015, pp. 97–111.

[8] D. Broneske et al. Toward hardware-sensitive database operations. In *EDBT*, 2014, pp. 229–234.

[9] H. Chafi et al. A domain-specific approach to heterogeneous parallelism. *ACM SIGPLAN Notices*, 46(8):35–46, 2011.

[10] J. Fabeiro et al. Writing self-adaptive codes for heterogeneous systems. In *Euro-Par*, 2014, pp. 800–811.

[11] C.-P. Haensch et al. Plan operator specialization using reflective compiler techniques. In *BTW*, 2015, pp. 363–382.

[12] B. He et al. Relational query coprocessing on graphics processors. *ACM TODS*, 34(4):21:1–21:39, 2009.

[13] M. Heimel et al. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9):709–720, 2013.

[14] D. Horn. Stream reduction operations for GPGPU applications. *GPU gems*, 2:573–589, 2005.

[15] T. Kaldewey et al. GPU join processing revisited. In *DaMoN*, 2012, pp. 55–62.

[16] I. Klonatos et al. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[17] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. *Computational Science*, 5544:884–892, 2009.

[18] Y. Li and J. M. Patel. Bitweaving: fast scans for main memory data processing. In *SIGMOD*, 2013, pp. 289–300.

[19] R. Mueller, J. Teubner, and G. Alonso. Data processing on FPGAs. *PVLDB*, 2(1):910–921, 2009.

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

[21] H. Pirk, S. Madden, and M. Stonebraker. By their fruits shall ye know them: a data analyst's perspective on massively parallel system design. In *DaMoN*, 2015, 5:1–5:6.

[22] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*, 2015, pp. 1493–1508.

[23] J. Power et al. Implications of emerging 3D GPU architecture on the scan primitive. *SIGMOD Record*, 44(1):18–23, 2015.

[24] B. Raducanu, P. A. Boncz, and M. Zukowski. Micro adaptivity in Vectorwise. In *SIGMOD*, 2013, pp. 1231–1242.

[25] K. A. Ross. Selection conditions in main memory. *ACM TODS*, 29(1):132–161, 2004.

[26] S. Rul et al. An experimental study on performance portability of OpenCL kernels. In *SAAHPC*, 2010.

[27] S. Seo et al. Automatic OpenCL work-group size selection for multicore CPUs. In *PACT*, 2013, pp. 387–398.

[28] E. A. Sitaridi and K. A. Ross. Optimizing select conditions on GPUs. In *DaMoN*, 2013, 4:1–4:8.

[29] J. E. Stone, D. Gohara, and G. Shi. OpenCL: a parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, 2010.

[30] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

APPENDIX

A. SELECTION VARIANT SOURCES

In this appendix, we list the OpenCL source code for the basic selection kernel variants described in Section 2. All kernels use branched evaluation and construct bitmaps with a granuliary of eight bits, except the *interleaved-reduce* kernel in Listing 5, which uses predication, and the atomic kernels in Listing 3 and Listing 4, which use 32-bit elements.

```
kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void scan(global int* in, global uchar* out, int cmp) {
  size_t rpos = get_global_id(0) * 1 * 8;
  uchar res = 0;
  for (uint i=0; i<8; ++i) {
    if (in[rpos++] < cmp) res |= ((uchar) 0x1) << i;
  }
  out[(rpos - 1)/8] = res;
}
```

Listing 2: Basic *sequential* kernel.

```
kernel __attribute__((reqd_work_group_size(32, 1, 1)))
void scan(global int* in, global uint* out, int cmp) {
  size_t group = get_group_id(0) + get_global_offset(0) / 32;
  size_t pos = group * 1 + get_local_id(0);
  for(;pos<(group + 1) * 1; pos += 1) {
    out[pos] = 0;
  }
  barrier(CLK_GLOBAL_MEM_FENCE);
  pos = group * 32 + get_local_id(0);
  if (in[pos] < cmp)
    atomic_or(&out[pos / 32], ((uint)0x1) << (pos % 32));
}
```

Listing 3: Basic *interleaved-global-atomic* kernel.

```
kernel __attribute__((reqd_work_group_size(32, 1, 1)))
void scan(global int* in, global uint* out, int cmp) {
  size_t group = get_group_id(0) + get_global_offset(0) / 32;
  local uint scratch[1];
  size_t pos = group * 32 + get_local_id(0);
  size_t li = get_local_id(0);
  if (li < 1) scratch[li] = 0;
  barrier(CLK_LOCAL_MEM_FENCE);
  if (in[pos] < cmp)
    atomic_or(&scratch[li / 32],
              ((uint)0x1) << (pos % 32));
  barrier(CLK_LOCAL_MEM_FENCE);
  event_t evt = async_work_group_copy(&out[group * 1],
                                      scratch, 1, 0);
}
```

Listing 4: Basic *interleaved-local-atomic* kernel.

```
kernel __attribute__((reqd_work_group_size(32, 1, 1)))
void scan(global int* in, global uchar* out, int cmp) {
  size_t group = get_group_id(0) + get_global_offset(0) / 32;
  size_t li = get_local_id(0);
  local uchar scratch[32];
  scratch[li] = ((uchar)(in[get_global_id(0)] < cmp))
                << (get_global_id(0) % 8);
  barrier(CLK_LOCAL_MEM_FENCE);
  for (uint stride = 1; stride < 4; stride *=2) {
    if (li % (2*stride) == 0)
      scratch[li] |= scratch[li + stride];
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (li % 8 == 0)
    out[group * 4 + li / 8] = scratch[li] | scratch[li + 4];
}
```

Listing 5: Basic *interleaved-reduce* kernel.

```
kernel __attribute__((reqd_work_group_size(32, 1, 1)))
void scan(global int* in, global uchar* out, int cmp) {
  size_t group = get_group_id(0) + get_global_offset(0) / 32;
  size_t li = get_local_id(0);
  local uchar scratch[32];
  size_t pos = group * 256 + li;
  uchar res = 0;
  for (uint i=0; i<8; ++i) {
    if(in[pos] < cmp) res |= ((uchar)0x1) << i;
    pos += 32;
  }
  scratch[li] = res;
  barrier(CLK_LOCAL_MEM_FENCE);
  uchar mask = ((uchar)(0x1)) << (li / 4);
  pos = 8 * (li % 4);
  uchar rot = 8 - ( li / 4);
  res = 0;
  for (uint i=0; i<8; ++i) {
    res |= rotate((uchar)(scratch[pos++] & mask),
          (uchar)(rot++));
  }
  out[group * 32 + li] = res;
}
```

Listing 6: Basic *interleaved-collect* kernel.

```
kernel __attribute__((reqd_work_group_size(32, 1, 1)))
void scan(global int* in, global uchar* out, int cmp) {
  size_t group = get_group_id(0) + get_global_offset(0) / 32;
  size_t li = get_local_id(0);
  local uchar scratch[32];
  size_t pos = group * 256 + li;
  uchar res = 0;
  for (uint i=0; i<8; ++i) {
    if(in[pos] < cmp) res |= ((uchar)0x1) << i;
    pos += 32;
  }
  scratch[li] = res;
  barrier(CLK_LOCAL_MEM_FENCE);
  if (li < 16) {
    pos = (li / 1) * 2 + li % 1;
    uchar t1 = scratch[pos];
    uchar t2 = scratch[pos + 1];
    scratch[pos] = (t1 & ((uchar)0x55)) |
                   ((t2 & ((uchar)0x55)) << 1);
    scratch[pos + 1] = ((t1 & ((uchar)0xAA)) >> 1) |
                       (t2 & ((uchar)0xAA));
  }
  barrier(CLK_LOCAL_MEM_FENCE);
  if (li < 16) {
    pos = (li / 2) * 4 + li % 2;
    uchar t1 = scratch[pos];
    uchar t2 = scratch[pos + 2];
    scratch[pos] = (t1 & ((uchar)0x33)) |
                   ((t2 & ((uchar)0x33)) << 2);
    scratch[pos + 2] = ((t1 & ((uchar)0xCC)) >> 2) |
                       (t2 & ((uchar)0xCC));
  }
  barrier(CLK_LOCAL_MEM_FENCE);
  if (li < 16) {
    pos = (li / 4) * 8 + li % 4;
    uchar t1 = scratch[pos];
    uchar t2 = scratch[pos + 4];
    scratch[pos] = (t1 & ((uchar)0x0F)) |
                   ((t2 & ((uchar)0x0F)) << 4);
    scratch[pos + 4] = ((t1 & ((uchar)0xF0)) >> 4) |
                       (t2 & ((uchar)0xF0));
  }
  barrier(CLK_LOCAL_MEM_FENCE);
  out[group * 32 + li] = scratch[(li % 4) * 8 + li / 4];
}
```

Listing 7: Basic *interleaved-transpose* kernel.