Hardware-Oblivious SIMD Parallelism for In-Memory Column-Stores

Template Vector Library — General Approach and Opportunities

Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Alexander Krause, Dirk Habich, Wolfgang Lehner
Database Systems Group
Technische Universität Dresden
Dresden, Germany
firstname.lastname@tu-dresden.de

Erich Focht
High Performance Computing Europe
NEC Deutschland GmbH
Stuttgart, Germany
erich.focht@emea.nec.com

ABSTRACT

Vectorization based on the *Single Instruction Multiple Data (SIMD)* parallel paradigm is a core technique to improve query processing performance especially in state-of-the-art in-memory column-stores. In mainstream CPUs, vectorization is offered by a large number of powerful SIMD extensions growing not only in vector size but also in terms of complexity of the provided instruction sets. However, programming with vector extensions is a non-trivial task and currently accomplished in a hardware-conscious way. This implementation process is not only error-prone but also connected with quite some effort for embracing new vector extensions or porting to other vector extensions. To overcome that, we present a *Template Vector Library (TVL)* as a hardware-oblivious concept in this paper. We will show that our single source hardware-oblivious implementation runs efficiently on different SIMD extensions as well as on a pure vector engine. Moreover, we demonstrate that several new optimization opportunities are possible, which are difficult to realize without a hardware-oblivious approach.

1. INTRODUCTION

To satisfy the requirements of high query throughput and low query latency, database systems constantly adapt to novel hardware features, but usually in a *hardware-conscious* way [6, 9, 11, 22, 26, 27]. That means, the implementations of, for example, query operators is very hardware-specific and any change or development of the underlying hardware leads to significant implementation and maintenance activities [12]. To overcome this issue, a very promising and upcoming research direction will focus on the development of *hardware-oblivious* or abstraction concepts. For example, Heimel et al. [12] already proposed a *hardware-oblivious* parallel library for query operators, so that these operators can be mapped to a variety of parallel processing architectures like many-core CPUs or GPUs. Another recent *hardware-oblivious* approach is the data processing interface DPI for modern networks [2].

In addition to parallel processing over multiple cores, vector processing according to the Single Instruction Multiple Data (SIMD) parallel paradigm is also a heavily used hardware-driven technique to improve the query performance in particular for in-memory column-store database systems[1] [10, 14, 21, 26, 27]. On mainstream CPUs, this vectorization is done using SIMD extensions such as Intel's SSE (Streaming SIMD extensions) or AVX (Advanced Vector Extensions). In the last years, we have seen great advancements for this vectorization feature in form of wider vector registers and more complex SIMD instructions sets. Moreover, NEC Corporation recently released a strong vector engine as a co-processor called SX-Aurora TSUBASA, which operates on vector registers multiple times wider than those of recent mainstream processors [16, 24].

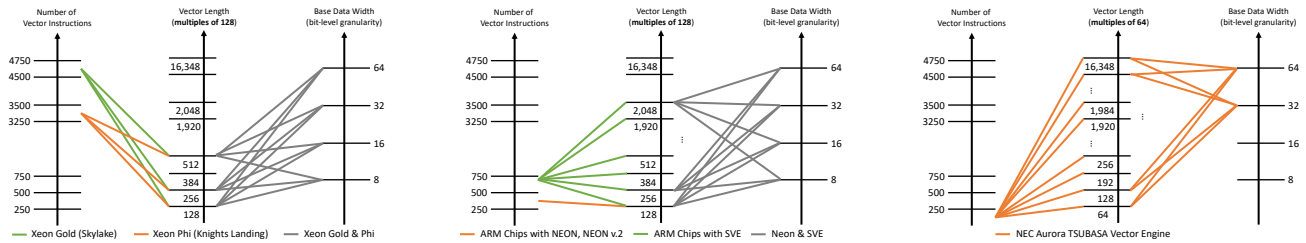
Our Contribution and Outline

To use the evolving variety of vector processing units in a unified way for a highly vectorized query processing in in-memory column-stores, we developed a *Template Vector Library (TVL)* as a novel hardware-oblivious approach. The unique properties of *TVL* are: (i) we provide a well-defined, standardized, and abstract interface for vectorized query processing, (ii) query operators have to be vectorized *only once* using *TVL*, (iii) this single set of query operators can be mapped to all vector processing units from different SIMD extensions up to vector engines at compile time, and (iv) based on our *hardware-oblivious* vectorization concept, we enable new optimization opportunities. To support our claims, we make the following contributions in this paper:

- We start with a motivation why we need a hardware-oblivious vectorization concept and present related work in Section 2.
- Then, we introduce our developed *Template Vector Library (TVL)* as a hardware-oblivious approach by describing the library interface and the mapping to dif-

[1]Without loss of generality, we will mainly focus on in-memory column-stores in this paper. Nevertheless, our concepts should be applicable in other contexts as well.

10th Annual Conference on Innovative Data Systems Research (CIDR '20) January 12-15, 2020, Amsterdam, Netherlands.



(a) Intel (b) ARM (c) NEC Vector Engine

Figure 1: Diversity in SIMD hardware landscape.

ferent vector processing units in Section 3.

- Based on *TVL*, we developed a novel in-memory column-store *MorphStore* [10]. Using that system, we empirically prove that our single hardware-oblivious implementation can efficiently run on different vector processing units in Section 4. Moreover, we discuss possible directions for future research in Section 5.

Finally, the paper concludes with a short summary of our findings in Section 6.

2. NECESSITY OF AN ABSTRACTION

Hardware vendors continuously improve their individual hardware systems by providing an increasingly high degree of parallelism [5]. This results in higher hardware core counts (MIMD parallelization) and improved vector units (SIMD parallelization). In MIMD (*Multiple Instruction Multiple Data*), multiple processors execute *different* instructions on *different* data elements in a parallel manner. This parallelization concept is heavily exploited in database systems [1, 3, 15, 20, 23] and Heimel et al. [12] already proposed a *hardware-oblivious* parallel library to easily map to different MIMD processing architectures. In contrast to that, multiple processing elements perform the *same* operation on multiple data elements simultaneously for SIMD parallelization (often called vectorization). This vectorization concept is always provided by modern x86-processors using specific SIMD instruction set extensions.

2.1 SIMD Parallelism Diversity

This SIMD parallel paradigm has received a lot of attention to increase query performance, especially in the domain of in-memory column-stores [10, 14, 21, 26, 27]. Without claim of completeness, current approaches mainly focus on vectorizing isolated operators, compression, partitioning, or on completely new processing models [7, 8, 14, 19, 21, 26, 27]. However, the vectorization of the mentioned aspects is normally done in a *hardware-conscious* way via hand-written code using SIMD intrinsics providing an understandable interface for the vector instructions [29]. This code will usually result in the best performance, but the code will probably only run on the given architecture and porting to a new architecture requires a high effort.

The reason is that the SIMD hardware landscape is increasingly diverse as illustrated in Figure 1. For this diversity, three metrics are important: (i) the number of available vector instructions[2], (ii) the vector length, and (iii) the granularity of the bit-level parallelism, i.e., on which data widths the vector instructions are executable. Figure 1(a) shows the metric values for two recent Intel architectures: *Xeon Skylake* and *MIC Knights Landing*. Generally, Intel offers several SIMD extensions such as SSE (Streaming SIMD Extensions), AVX (Advanced Vector Extensions), AVX2, or AVX-512. Both architectures in Figure 1(a) offer SIMD functionality up to Intel's latest extension AVX-512, resulting in a very high number of vector instructions with three different vector lengths of 128-, 256-, and 512-bit. The SIMD processing can be done on 64-, 32-, 16-, and 8-bit data elements on both architectures. As depicted, the number of vector instructions differs between both Intel architectures, because not all available SIMD extensions are meant to be supported by all architectures.

In contrast to that, ARM, for example, pursues a completely different approach as highlighted in Figure 1(b). Instead of providing a high number of vector instructions, ARM supports much wider vector lengths. While the ARM NEON extension (available in ARMv7-A and ARMv8-A) was limited to a vector length of 128-bit, the Scalable Vector Extension (SVE) (available in ARMv8-A AArch64) aims at supporting much more vector lengths from 128 to 2,048 bits, in 128-bit increments [28]. In all cases, the SIMD processing can be done on 64-, 32-, 16-, and 8-bit data elements. Moreover, Figure 1(c) shows the metric values for a recently released pure vector engine from NEC Corporation [16, 24]. This vector engine operates on vector registers multiple times wider than those of recent SIMD extensions of modern x86-processors, whereas the engine supports vector lengths from 64 to 16,384 bits, in 64-bit increments. Moreover, the number of available vector instructions is lower than with the SIMD extensions and the instructions can only be executed on 64- and 32-bit data elements.

2.2 Abstraction Guidelines

With this diversity in mind, we believe that developing a *hardware-oblivious* or abstraction concept will become equally important as achieving optimal performance. From our point of view, the *hardware-oblivious* concept should provide the following core aspects:

Portability and Extensibility: The vectorized code written in a *hardware-oblivious* way should be easily portable between vector processing units with different SIMD capabilities. In addition, the implementation effort for the integration of new SIMD functionalities offered by a specific vector processing unit should be manageable. Of course, this also means that the *hardware-oblivious* approach should be extensible with new functions that are necessary for the explicit vectorization of application logic.

Enabling Explicit Vectorization: To achieve the best performance, explicit vectorization of application

[2]We counted the instructions in the hardware vendor intrinsic guides.

logic is still the best way [17, 27]. Even if a given code can be auto-vectorized implicitly, this poses new challenges to the overall system as described in [17]. Thus, a *hardware-oblivious* concept should enable an explicit vectorization for the diversity in SIMD hardware landscape (see Figure 1) with the following properties. On the one hand, the diversity characteristics of SIMD functionality, vector length and the granularity of the bit-level parallelism should be treated independently of each other. This is the best way to represent diversity in a meaningful way. On the other hand, the *hardware-oblivious* concept should allow a separation between application logic implementation and specification of the three diversity characteristics at compile- or run-time. This enables the highest degree of flexibility in mapping of application logic to a specific SIMD hardware.

2.3 Related Work

As already mentioned, a very promising and upcoming research direction will focus on the development of *hardware-oblivious* concepts to tackle the ever-increasing diversity or heterogeneity in hardware. For example, Heimel et al. [12] presented a hardware-oblivious extension for MonetDB by using the parallel programming framework OpenCL. As they have shown, they can map single-source query operators implemented in OpenCL to different parallel processing architectures like multi-core CPUs or GPUs. Another hardware-oblivious approach in this direction is Voodoo, which is a declarative intermediate algebra abstracting the detailed architectural properties [25]. The Voodoo compiler also produces OpenCL code to support multi-core CPUs and GPUs. Moreover, Voodoo includes a small set of vector operations like Scatter. However, the main bottleneck of both approaches is OpenCL from a vectorization performance perspective. Behrens et al. [4] have clearly shown that, e.g., vectorized hashing based on SIMD intrinsics outperforms OpenCL-based hashing.

There are already approaches for *hardware-oblivious* vectorization in the form of vector libraries, which avoid additional layers like OpenCL and reduce the additional overhead to a minimum, e.g., VC [17] and Sierra [18]. Both approaches automatically translate a generic vector type to the largest available vector size and overload standard operators to work with this vector type. This reduces the translation of the written code to exactly one vector size and instruction set, even if the system offers more, and potentially faster, variants. Additionally, very specialized functions, e.g., a stream store, which can enhance the performance in some cases but have no equivalent standard operator, cannot be used. Hence, VC and Sierra work well in plain mathematical scenarios, but they do not satisfy our abstraction guidelines as presented in the previous section.

3. TEMPLATE VECTOR LIBRARY

In this section, we introduce TVL, our developed hardware-oblivious template vector library approach for in-memory column-store systems. To the best of our knowledge, TVL is the first approach in this direction satisfying our defined abstraction guidelines.

3.1 TVL Overview

Figure 2 illustrates our library architecture according to

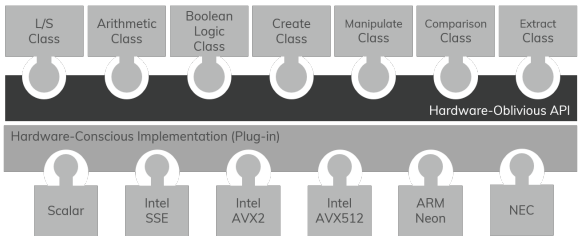


Figure 2: Architecture of our Template Vector Library (TVL).

a separation of concerns concept with the help of template metaprogramming in C++. That means, our TVL offers hardware-oblivious, but column-store specific primitives as generic functions. This explicitly enables database systems programmers to implement each query operator in a vectorized, but hardware-independent fashion, on the one hand. On the other hand, the TVL is also responsible for mapping the provided hardware-oblivious primitives to different SIMD hardware. For this mapping, our TVL includes a plug-in concept, where each plug-in has to implement each provided hardware-oblivious primitive for a specific SIMD hardware in a hardware-conscious manner.

3.2 Hardware-Oblivious Primitives

To enable a hardware-oblivious approach without sacrificing the performance, our TVL primitives abstract from SIMD intrinsics in such a way that there is the same interface and the same data types for all SIMD architectures. Our data types are `vector_t` referring to a vector, `base_t` referring to the type of the elements in a vector, and `mask_t` referring to a bitmask. While scalar or plain mathematical computations are always a combination of L/S operations, comparisons, arithmetic calculations, and boolean logic, vectorized query processing in in-memory column-stores requires additional functionality, e.g., permutation of vector elements as discussed in the literature [7, 8, 14, 19, 21, 26, 27]. Based on this observation, we define 7 different classes of TVL primitives as illustrated in Figure 2.

Load/Store Class: The *Load/Store Class* contains different load and store primitives. The most obvious members of this class are sequential load and store primitives. Random memory access is realized by gather and scatter primitives. A special primitive for selectively storing the elements of a vector is called compressstore. A compressstore takes a bitmask (`mask_t`), a vector (`vector_t`), and a pointer as arguments. Then, it stores all elements of the vector, which have a corresponding set bit in the bitmask, consecutively into memory without gaps for non-selected elements. Furthermore, data can be aligned or unaligned in memory, and there are instruction sets, which explicitly support streaming, also known as non-temporal memory access. To differentiate between these cases, a template parameter is used, which can have the values `ALIGNED, STREAM`, and `UNALIGNED` for alternative implementations. To determine the granularity of any operation, another template parameter is used. For instance, if a gather is supposed to read 64-bit integers, the granularity template parameter is set to `64`.

Arithmetic Class: The *Arithmetic Class* provides different unary and binary function primitives. Currently, the unary function primitives contain the aggregation of all elements of a vector by summing them up, and the change of

```
//a) AVX-512
_mm512_mask_compressstoreu_epi64(dataPtr, mask, vec);
//b) SSE
switch (mask){
    case 0: return; //store nothing
    case 1: _mm_storeu_si128(dataPtr, vec);
            return; //store everything, unnecessary second vector element is overwritten
                 later
    case 2: vec = _mm_shuffle_epi8(vec, _mm_set_epi8
        (7,6,5,4,3,2,1,0,15,14,13,12,11,10,9,8));
            //exchange lanes before storing
            _mm_storeu_si128(dataPtr, vec);
            return;
    case 3: _mm_storeu_si128(dataPtr, vec);
            return; //store everything
}
//c) NEON
switch (mask){
    case 0: return; //store nothing
    case 1: vst1q_lane_u64(dataPtr, vec, 0); return; //store 1st lane
    case 2: vst1q_lane_u64(dataPtr, vec, 1); return; //store 2nd lane
    case 3: vst1q_u64(dataPtr, vec); return; //store everything
}
```

Figure 3: Different vectorized *compressstore* specializations.

the sign of a number. The currently provided binary function primitives are the four basic arithmetical operations, modulo, and shift operations, where each element of a vector is shifted by a defined number of bits. The result returned by arithmetic primitives is always a vector (`vector_t`), regardless of whether there are one or two input vectors. The granularity of bit-level parallelism of the operations is again provided by a template parameter.

Comparison Class: This class provides element-wise comparisons between vectors, e.g. for equality or greater/less than. The input parameters are the same as for the binary arithmetic primitives, but the output is a bitmask (`mask_t`) instead of a vector. This is especially useful, when the result of a comparison is stored. Instead of storing the bitmask, the corresponding values can be stored by using the compressstore of the *Load/Store Class*.

Bitwise Logic Class: In this class, boolean algebra is treated. Currently, a bitwise AND and a bitwise OR are provided. Compared to the interface of the binary arithmetic primitives, the template parameter, which indicates the granularity, is not necessary, because the operations are always performed bitwise.

Create Class: Sometimes, the elements of a vector are not loaded from memory, but computed at runtime, or loaded from immediate values or constants. These cases are treated by the *Create Class*. This class poses the challenge, that setting the elements of a vector to different values is an operation whose interface depends on the number of elements of a vector, i.e. depending on the number of elements per vector, the number of input parameters varies. This is incompatible with the concept of code which is portable to different vector lengths. For this reason, we provide two additional primitives in this class: `set1` and `set_sequence`. The first one sets all elements of a vector to the same value. The second one fills a vector with a sequence of numbers, where the first number (`base_t`) and the distance between the following numbers are provided as parameters. This is especially useful for the initial creation of record indexes.

Extract Class: Whenever it is necessary to get a single element out of a vector, the `extract` primitive is used. This is the only member of the *Extract Class*. The same effect can be achieved by using a `compressstore` (Load/Store Class) with a single bit set in the bitmask, but `extract` avoids the roundtrip to the main memory, if a corresponding instruction is available on the targeted architecture.

```
template<...>
   void
   compressstore(
      typename base_t * dataPtr,
      typename vector_t vec,
      typename mask_t mask
   );
```

Figure 4: Compressstore template declaration in C++.

Manipulate Class: The last class is the *Manipulate Class*, which takes care of vector manipulations on the element-level, i.e. permutations of the vector elements. Currently, a single primitive is provided, the `rotate` primitive. The rotate primitive rotates the elements in a vector by one element. If there are only two elements in a vector, e.g. two double values in a 128-bit register, this results in swapping these two elements.

These classes and primitives are specific for an efficient vectorized query processing in in-memory column stores. An example TVL primitive from the `L/S Class`, which is used in selective database operators, is the `compressstore` function template as depicted in Figure 4. This specific function takes a pointer (`dataPtr`) to a memory location, a vector (`vec`), and a bitmask (`mask`) as input parameters. Then, the task is to store the vector register (`vec`) in such a way that the vector elements with a corresponding set bit in the bitmask are stored consecutively to memory. How this function is realized is not the subject of this template *declaration*.

3.3 Hardware-Conscious Specialization

From a template metaprogramming perspective, our TVL primitives are generic interfaces to implement in a vectorized way, but for the execution, we require a hardware-conscious

```
//let outPtr, mask, and vec be local variables
//a) Using AVX-512 intrinsics directly, no TVL
_mm512_mask_compressstoreu_epi64(outPtr, mask, vec);
//b) Naive implementation using TVL
tvl::compressstore < avx512<v512<int64_t>> , tvl::UNALIGNED, 64>(outPtr, vec, mask);
//c) Fully portable implementation using TVL
using processingStyle = avx512<v512<int64_t>>;
tvl::compressstore <processingStyle, tvl::UNALIGNED, processingStyle::base_t_size_bit>
  (outPtr, vec, mask);
```

Figure 5: Example usage of the TVL.

function template specialization for the underlying SIMD hardware. This function template specialization has to be implemented, whereby the implementation depends on the available functionality of the SIMD hardware. However, this is independent from the query operators and must be done *only once* by a domain expert for a specific SIMD hardware.

In the best case, we can directly map a TVL primitive to a SIMD intrinsic. However, if the necessary SIMD intrinsic is not available, we are able to implement a work-around in a hardware-conscious way. To illustrate this specialization, Figure 3 shows the implementation of the *compressstore* primitive for three different SIMD instruction sets assuming a 64-bit base data type.[3] To the best of our knowledge, Intel's AVX-512 is the only instruction set, which contains an intrinsic doing exactly what a *compressstore* is meant to do, and we map directly to this intrinsic. For architectures without AVX-512, a work-around has to be implemented. In this work-around, we have to treat all possible values of the bitmask manually. If no bit in the bitmask is set, the function returns without storing anything. If all bits are set, the whole register is stored. If only a subset of bits is set, there are different ways to store the corresponding vector elements. In NEON, there is an intrinsic to store selected vector elements. In SSE, we could either use an intrinsic to extract values into a scalar register and then write them to memory, or shuffle the according elements to the beginning of the register before writing it to memory. We decided for the latter because of a higher compatibility with different base data types.

To eliminate the overhead of a function call when using primitives, we inlined all primitives with `inline __attribute__((always_inline))`. This ensures that the overhead over using intrinsics is negligible as we show in Section 4.

A nice side effect of our overall concept is that we are also able to map to a scalar specialization. In this case, the vector length can be 8-, 16-, 32-, or 64-bit and we map our TVL primitives to the corresponding scalar instructions.

3.4 Interplay

In order to connect our hardware-oblivious primitives with the different hardware-conscious specializations during query compile-time, we decided to use three template parameters and call a combination of these parameters a *processing style*. The template parameters are derived from the description of the SIMD variety in Section 2: (1) the vector extension (e.g., SSE, AVX, NEON, or scalar), (2) the vector size in bit, and (3) the base data type with bit granularity (e.g., int8, int64, float). This enables us to define the exact mapping in a very fine-grained and flexible way. That means, each primitive within a query operator can be called with its own processing style. An example of how exactly the processing styles are used, is shown in Figure 5. In a), the intrinsics provided by AVX-512 are used directly. This code will only work on 512-bit registers on Intel architectures providing this instruction. Snippet b) uses the TVL in a very naïve way. The primitive *compressstore* is called with a processing style, that maps to AVX-512. The second template parameter indicates that the data does not need to be aligned. To make the code fully portable to other SIMD architectures, derived constants, such as `base_t_size_bit`, can be used, which are also provided by our TVL as shown in snippet c) in Figure 5.

Finally, Figure 6 illustrates how a query operator implemented using TVL can look like and how it can be called. In particular, we show a simple aggregation operator, which assumes that the number of data elements is a multiple of the number of elements per vector.

[3]Note that, for the sake of simplicity, the function headers are not shown.

4. EVALUATION

Our TVL is a core component of *MorphStore*, a proper in-memory column-store with a novel highly vectorized compression-aware query processing concept [10]. The source code of MorphStore including the TVL is available on GitHub.[4] Since the contribution of this paper is on hardware-oblivious vectorization using TVL, our evaluation mainly focuses on vectorized query processing of uncompressed 64-bit data in MorphStore. For this evaluation, we implemented template specializations for scalar processing and different SIMD extensions such as Intel SSE, AVX2, and AVX-512, a NEC vector engine, and ARM NEON.

4.1 Microbenchmarks

For an initial evaluation of the TVL, we run some microbenchmarks on different hardware. All microbenchmarks consist of isolated query operators. We use the same code base on every system for every supported instruction set. Only the processing style is changed, which affects one line of code.

Runtime Overhead Consideration

In a first series of experiments, we compared the runtimes of hand-written query operator code using SIMD intrinsics with runtimes of TVL-enabled query operators. To keep the results comparable, both variants use the functionalities offered by Intel AVX2. For all these microbenchmarks, the complete processed data fit at least into the L3-cache of the Xeon Gold 5120 (Skylake, max. core frequency: 3.2 GHz) and the benchmarks ran single-threaded.

[4]https://github.com/MorphStore/Engine

```
// Aggregation operator definition.
template<class ps> // processing style
base_t agg(const base_t * in, size_t elCount) {
  // For simplicity, we assume that elCount is a multiple of the number of data elements
  // per vector register.
  const size_t vecCount = elCount / ps::vector_element_count;

  // Initialize running sum to zero.
  vector_t resVec = tvl::set1<ps, ps::vector_base_t_granularity>(0);

  // Add all input data elements to running sum.
  for(size_t i = 0; i < vecCount; ++i) {
    vector_t dataVec = tvl::load<ps, tvl::ALIGNED, ps::vector_size_bit>(in);
    resVec = tvl::add<ps, ps::vector_base_t_granularity>(resVec, dataVec);
    in += ps::vector_element_count;
  }

  // Calculate final sum using horizontal summation of the vector elements.
  return tvl::hadd<ps, ps::vector_base_t_granularity>(resVec);
}

// Calling the operator.
using ps = tvl::avx2<tvl::vec256<uint64_t>>; // for example
size_t count = 1024;
uint64_t * array = generate_data(count);
uint64_t sum = agg<ps>(array, elemCount);
```

Figure 6: A simple summation operator using the TVL.

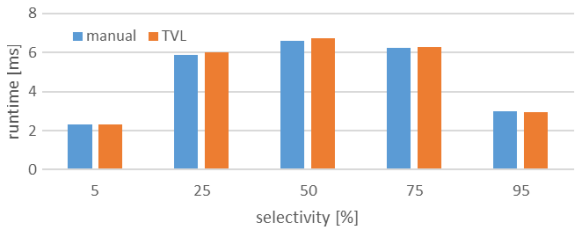


Figure 7: Runtime overhead for column scan. The dataset consisted of about 18.5 MB of integers.

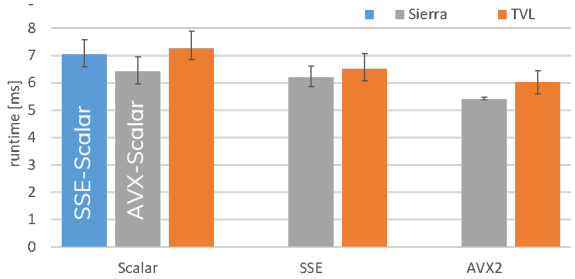


Figure 8: Runtime comparison of an aggregation between Sierra (blue and gray) and the TVL (orange). The dataset contained 10^7 integers.

All micro-benchmarks were executed 900 times to minimize the impact of time measurement. Figure 7 exemplary shows the different average runtimes for a column scan with different range predicate selectivities. We used these selectivities to vary the runtime behavior. The selectivities are plotted on the x-axis while the y-axis shows the complete runtime of the operator in milliseconds (lower is better). On the one hand, the variant using TVL performs slightly better than the variant using SIMD intrinsics for selectitivies of 5% and 95%, respectively. On the other hand, for selectivities of 25% and 50%, the operator variants behave inversely, resulting in an average runtime overhead for our TVL approach of around 1.02% for this operator. For the other query operators, we observed a similar overhead resulting in a neglectable overhead. Hence, we conclude that our TVL offers high flexibility at virtually no performance cost.

Comparison with Sierra

In a second series of experiments, we compared the runtime performance of the TVL to another vector library, namely Sierra [18]. Since the functionality of Sierra is limited, we did not implement any selective operator. Instead, we implemented an aggregation, which is trivially vectorizable.

To not measure the efficiency of an auto-vectorizer, but of Sierra's SIMD mode and the TVL primitives respectively, we turned off auto-vectorization. To compile the implementation for Sierra, we used the sierra fork of clang (version 3.3). For the TVL, we used the regular clang compiler (version 7). All experiments were run single-threaded on an Intel Xeon Gold 6130 CPU (max. core frequency: 3.7 GHz). Fig. 8 shows the runtimes for SSE, AVX, and scalar processing. In addition to the median of 10 runs, the range of the runtimes is shown. In Sierra, there are two different versions of scalar processing. The first one, *SSE-Scalar*, is compiled for SSE, but with a vector length of 1, i.e. every vector contains only one element but the compiler is allowed to use SSE functionality. The second one, *AVX-Scalar*, also has a vector length of 1, but is compiled for AVX. For the TVL, we used scalar, SSE, and AVX2 processing styles with the native vector lengths of the corresponding instruction set, e.g. 128-bit registers for SSE. The graph shows that in this use-case, the size of registers does not scale proportionally with the performance. This is because the workload mainly consists of memory read access, which is bandwidth limited. The ranges of the runtimes for Sierra and the TVL are overlapping in all cases except for AVX2. However, Sierra performs slightly better than the TVL. We suppose that this is because of the highly SIMD-optimized Sierra compiler. A six times lower cache-reference number and 30% more instructions per CPU cycle, with slightly slower CPU cycles, for the

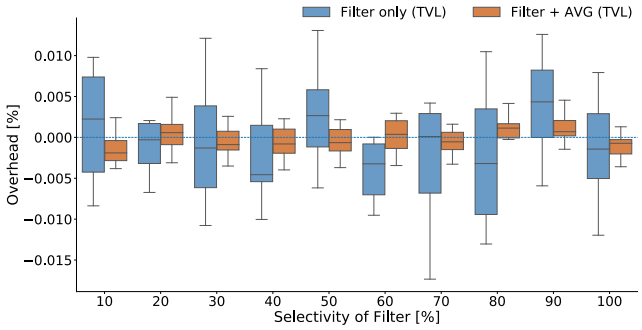


Figure 9: Runtime overhead for a FILTER and a FILTER with a subsequent AGGREGATION on NEC SX-Aurora TSUBASA.

Sierra implementation supports this assumption. Considering, that Sierra puts tight limits on the usable functionality of any vector extension, especially when it comes to selective operations, it is not applicable for a database system despite the small performance gain compared to the TVL.

TVL on a Vector Engine

As another part of our evaluation, we investigated the applicability of TVL on the novel hardware accelerator card SX-Aurora TSUBASA. On the one hand, this vector processor, manufactured by NEC Corp., offers a very high sustained memory throughput of up to 1TB/s. This is made possible by an HBM packaging technology, which has been designed specifically for this accelerator [16]. On the other hand, SX-Aurora TSUBASA provides comparably large vector registers containing up to 256 double words (16,484 Bit). While the vector card is shipped with a tailored compiler providing auto-vectorization but no intrinsics support, our experiments were conducted using an LLVM-VE backend [13] (version 1.7). Since intrinsics are provided by the backend, we could expand the TVL to support SX-Aurora TSUBASA quite easily using the already discussed separation of concerns.

To measure the potential overhead of the TVL we implemented a FILTER operator which filters out all elements from a given dataset which are less than or equal to a constant number. The result is materialized as a bitvector, where every set bit represents an element from the initial dataset which satisfies the given predicate. In addition, we implemented an AGGREGATE operator, which takes a dataset as well as a bitvector as input and calculates the average of all elements from the given dataset, while only considering elements with their corresponding bit set within the bitvector. The operators were implemented twice, once using the intrinsics defined by LLVM-VE and once using the extended TVL primitives. The experiments where executed by a single thread on a single maschine, consisting of an Intel(R) Xeon(R) Gold 6126 CPU and a SX-Aurora TSUBASA 10C with a frequency of 1.4 Ghz, 24 GB HBM2 providing a maximum memory bandwith of 750.0 GB. On the host system a CentOS Linux release 7.7.1908 was used. The vector engine is operated using veos 2.1.0. To build the experiments LLVM-VE (1.7.0) was used with the platform flag *-target ve-linux* and optimization flags *-O3 -fno-vectorize -fno-slp-vectorize*, whereby the latter two disable auto-vectorization. The experiments were executed completely on the vector card utilizing veos. To measure the performance of the operators, the user clock cycles on the vector engine were retrieved using inline assembly.

All operators were measured with different data sizes as well as data distributions, which lead to different selectivities for the FILTER. Figure 9 shows the results for a data size of a single FILTER operator on 4GB of data and a FILTER on a 4GB buffer followed by an AGGREGATE on another 4GB buffer, resulting in a combined data size of 8GB. The maximum measured overheads were -0.017% for a selectivity of 70% and 0.012% for a selectivity of 50%. Apart from that, the overhead of the TVL is usually within the range of -0.005% to 0.005% and is, thus, negligible. Hence, we conclude that the TVL can also be extended to support specialized hardware at no significant performance costs.

4.2 Star Schema Benchmark (SSB)

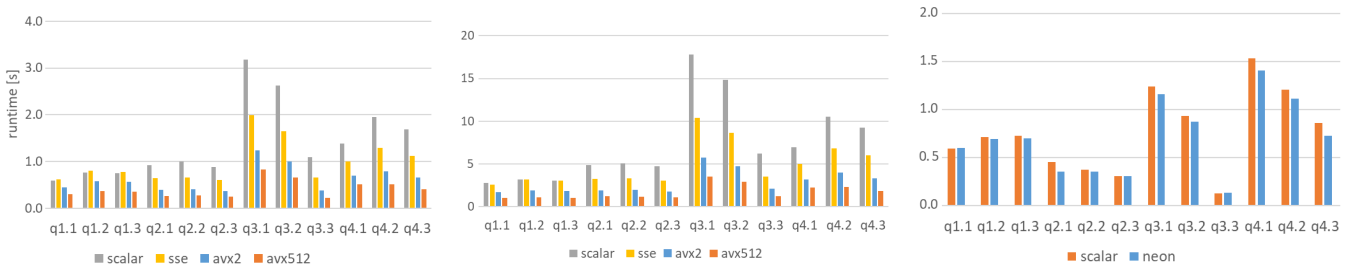
For a more sophisticated evaluation to show the applicability and portability of our approach, we ran the SSB using one code facilitating the TVL with different *processing styles* to use different SIMD extensions. Our systems were: a) an Intel Xeon Gold 5120 (Skylake, max. core frequency: 3.2 GHz), b) an Intel Xeon Phi 7250 (max. core frequency: 1.6 GHz), and c) an Odroid-C2, a single-board computer equipped with 4 ARM Cortex-A53 cores (max. core frequency: 1.5 GHz). While the Intel machines support SSE, AVX2 and AVX-512, the ARM core only supports the NEON instruction set. For a better comprehensibility, we used register sizes according to the instruction set, e.g., AVX2 uses only 256-bit registers, not 128-bit registers, and the queries are executed by a single thread. Figure 10 shows the individual query runtimes for all systems and instruction sets for a scale factor of 1. At larger scale factors, the data would not fit into the Odroid's 2 GB of RAM. The results on the Intel machines show that the performance benefits from wider registers. Note that this improvement can be accomplished by changing only the single line of code which defines the *processing style*. However, on the Odroid, vectorization does reduce the runtime, but not as much as expected. One reason for this is the high amount of work-arounds resulting from the low number of available vector instructions.

Overall, our evaluation empirically proves that our single hardware-oblivious operators with TVL can efficiently run on different vector SIMD extensions, including different hardware vendors, and that the query performance can be increased with the vector width.

5. ONGOING RESEARCH DIRECTIONS

Our ongoing research activities are manifold. On the one hand, we are currently completing our integration of the instructions of the vector engine SX-Aurora into our TVL. This will also give us the opportunity to use a common code base to investigate the interplay of the vector engine with its Intel Skylake host processor for an efficient query processing. That means, we want to leverage the wide vector registers as well as the high memory throughput of the vector engine while compensating the shortage of processing logic (see Section 2) with the Intel Skylake.

On the other hand, the different available instruction sets on one hardware system, for example on Intel with SSE, AVX2, and AVX-512 or on ARM with a configurable vector length in SVE, provide a huge optimization potential. That means, we want to optimize the overall performance on a very fine-grained level. To illustrate that, Figure 11



(a) Intel Xeon Gold (b) Intel Xeon Phi (c) ARM Cortex-A53

Figure 10: Star Schema Benchmark results on different systems using different *Processing Styles*.

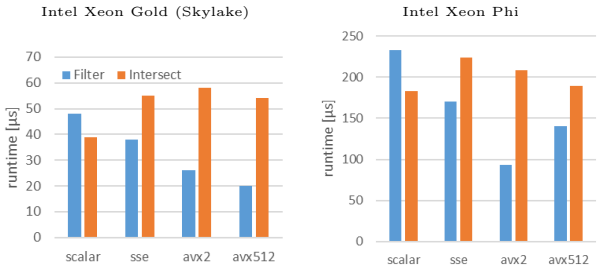


Figure 11: Runtime of two subsequent operators of SSB query 3.3 on two different Intel systems for different instruction sets and register sizes.

shows two subsequent operators of SSB query 3.3. As we can see, the same instruction set behaves differently on different systems, such that for example, AVX-512 is not always the best choice on Intel systems. Moreover, the prioritization of queries gets another dimension. For instance, cores executing low-priority queries can be executed using scalar code or SSE, while high-priority queries are executed with AVX-512. This allows for less frequency down-scaling on the whole system, because AVX2 and AVX-512 scale down the core frequency if multiple cores are used.

Generally, our TVL concept enables new research directions on a fine-grained level which are hardly realizable without a hardware-oblivious concept.

6. CONCLUSION

In this paper, we introduced a *Template Vector Library (TVL)* as a hardware-oblivious SIMD parallelism concept for in-memory column-stores. The TVL can represent the vector extension, the vector width, and the granularity of the bit-level parallelism, as the most important dimensions of the variety of the SIMD parallelism paradigm, effectively. As we have shown, we are able to map a single set of operators to different SIMD architectures without sacrificing the performance compared to hardware-conscious implementations. Moreover, TVL is already a core component of a new in-memory column-store called *MorphStore* [10].

7. ACKNOWLEDGMENT

This work was partly funded (1) by the German Research Foundation (DFG) within the CRC 912 (HAEC), RTG 1907 (RoSI) as well as by an individual project LE-1416/26-1 and (2) by NEC Corporation.

8. REFERENCES

[1] A. Ailamaki, E. Liarou, P. Tözün, D. Porobic, and I. Psaroudakis. *Databases on Modern Hardware: How to Stop Underutilization and Love Multicores.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2017.

[2] G. Alonso, C. Binnig, I. Pandis, K. Salem, J. Skrzypczak, R. Stutsman, L. Thostrup, T. Wang, Z. Wang, and T. Ziegler. DPI: the data processing interface for modern networks. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[3] C. Barthels, G. Alonso, T. Hoefler, T. Schneider, and I. Müller. Distributed join algorithms on thousands of cores. *PVLDB*, 10(5):517–528, 2017.

[4] T. Behrens, V. Rosenfeld, J. Traub, S. Breß, and V. Markl. Efficient SIMD vectorization for hashing in opencl. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, pages 489–492, 2018.

[5] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, 2011.

[6] P. Chrysogelos, P. Sioulas, and A. Ailamaki. Hardware-conscious query processing in gpu-accelerated analytical engines. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[7] P. Damme, D. Habich, J. Hildebrandt, and W. Lehner. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 72–83, 2017.

[8] P. Damme, A. Ungethüm, J. Hildebrandt, D. Habich, and W. Lehner. From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms. *ACM Trans. Database Syst.*, 44(3):9:1–9:46, 2019.

[9] F. Faerber, A. Kemper, P. Larson, J. J. Levandoski, T. Neumann, and A. Pavlo. Main memory database systems. *Foundations and Trends in Databases*, 8(1-2):1–130, 2017.

[10] D. Habich, P. Damme, A. Ungethüm, J. Pietrzyk, A. Krause, J. Hildebrandt, and W. Lehner. Morphstore - in-memory query processing based on morphing compressed intermediates LIVE. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1917–1920, 2019.

[11] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled CPU-GPU architectures. *PVLDB*, 8(4):329–340, 2014.

[12] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9):709–720, 2013.

[13] K. Ishizaka, K. Marukawa, E. Focht, S. Moll, M. Kurtenacker, and S. Hack. NEC SX-Aurora - A Scalable Vector Architecture. US LLVM Developers' Meeting(2018), 2018. http://compilers.cs.uni-saarland.de/papers/nec_poster_llvmdev18.pdf [Online, last acessed 12-16-2019].

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

[15] T. Kiefer, T. Kissinger, B. Schlegel, D. Habich, D. Molka, and W. Lehner. ERIS live: a numa-aware in-memory storage engine for tera-scale multiprocessor systems. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 689–692, 2014.

[16] K. Komatsu, S. Momose, Y. Isobe, O. Watanabe, A. Musa, M. Yokokawa, T. Aoyama, M. Sato, and H. Kobayashi. Performance evaluation of a vector supercomputer sx-aurora TSUBASA. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*, pages 54:1–54:12, 2018.

[17] M. Kretz and V. Lindenstruth. Vc: A C++ library for explicit vectorization. *Softw., Pract. Exper.*, 42(11):1409–1430, 2012.

[18] R. Leißa, I. Haffner, and S. Hack. Sierra: a SIMD extension for C++. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing, WPMVP 2014, Orlando, Florida, USA, February 16, 2014*, pages 17–24, 2014.

[19] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.*, 45(1):1–29, 2015.

[20] F. Liu, L. Yin, and S. Blanas. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 48–63, 2017.

[21] P. Menon, A. Pavlo, and T. C. Mowry. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB*, 11(1):1–13, 2017.

[22] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes. Memory management techniques for large-scale persistent-main-memory systems. *PVLDB*, 10(11):1166–1177, 2017.

[23] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1):928–939, 2010.

[24] J. Pietrzyk, D. Habich, P. Damme, E. Focht, and W. Lehner. Evaluating the vector supercomputer sx-aurora TSUBASA as a co-processor for in-memory database systems. *Datenbank-Spektrum*, 19(3):183–197, 2019.

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

[26] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1493–1508, 2015.

[27] O. Polychroniou and K. A. Ross. Towards practical vectorized analytical query engines. In *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019*, pages 10:1–10:7, 2019.

[28] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Prémillieu, A. Reid, A. Rico, and P. Walker. The ARM scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017.

[29] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, pages 145–156, 2002.