Mastering the NEC Vector Engine Accelerator for Analytical Query Processing

Annett Ungethüm, Lennart Schmidt, Johannes Pietrzyk, Dirk Habich, Wolfgang Lehner
Database Systems Group, *TU Dresden*, Dresden, Germany
{annett.ungethuem,lennart.schmidt1,johannes.pietrzyk,dirk.habich,wolfgang.lehner}@tu-dresden.de

***Abstract*—NEC Corporation offers a vector engine as a specialized co-processor having two unique features. On the one hand, it operates on vector registers multiple times wider than those of recent mainstream x86-processors. On the other hand, this accelerator provides a memory bandwidth of up to 1.2TB/s for 48GB of main memory. Both features are interesting for analytical query processing: First, vectorization based on the Single Instruction Multiple Data (SIMD) paradigm is a state-of-the-art technique to improve the query performance on x86-processors. Thus, for this accelerator we are able to use the same programming, processing, and optimization concepts as for the host x86-processor. Second, this vector engine is an optimal platform for investigating the efficient vector processing on wide vector registers. To achieve that, we describe an approach to master this co-processor for analytical query processing using a column-store specific abstraction layer for vectorization in this paper. We also detail on selected evaluation results to show the benefits and shortcomings of our approach as well as of the co-processor compared to x86-processors. We conclude the paper with a discussion on interesting future research activities.**

***Index Terms*—Vectorization, Query Processing, Abstraction**

I. Introduction

Vectorization based on the *Single Instruction Multiple Data (SIMD)* parallel paradigm has become a core technique to improve analytical query processing performance especially in state-of-the-art in-memory column-stores [1]–[5]. The main goal of SIMD is to increase the single-thread performance by executing an identical operation on multiple data elements in a vector register simultaneously (data parallelism) [6]. Such SIMD capabilities are common in today's mainstream x86-processors using specific SIMD instruction set extensions. A current hardware trend in this context can be seen in a growth of these extensions not only in terms of complexity of the provided instructions but also in the size of the vector registers [7]. To tackle that SIMD-specific heterogeneity, we introduced a novel abstraction layer called *Template Vector Library (TVL)* for in-memory column-stores [7]. Using that abstraction layer, we are able to implement hardware-oblivious vectorized query operators, which can be specialized to different SIMD instruction sets at query compile-time [7].

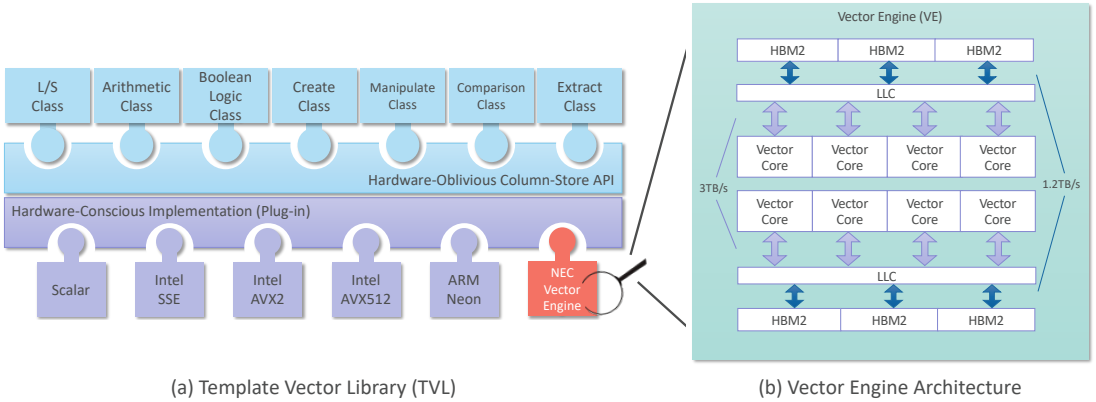
Furthermore, hardware is also shifting from homogeneous x86-processors towards heterogeneous systems with different computing units [8]. In this context, NEC Corporation introduced a novel heterogeneous hardware system called SX-Aurora TSUBASA consisting of a recent x86-processor as host and one or multiple strong vector engines as co-processors [9]. Each vector engine operates on vector registers of size 16,384-bit which is multiple times wider than that of common x86-processors and provides a total memory bandwidth of 1.2TB/s for a maximum of 48 GB of high bandwidth memory. By applying our *TVL* approach to SX-Aurora TSUBASA, we are able to use the same programming, processing, and optimization concepts for the host x86-processor as well as for this accelerator. Additionally, this co-processor is an optimal platform for investigating the vector processing on extremely wide vector registers.

Our Contribution: Thus, we present an approach to master the NEC SX-Aurora TSUBASA vector engine in our developed SIMD abstraction layer *TVL* in this paper. This mastering requires novel concepts to represent and to process masks for the wide vector registers. Masks are an important part of vectorization, because they are used for marking which elements in a vector register the operation has to be applied to [3]–[5]. Then, we show selected results to highlight the benefits and shortcomings of the vector engine compared to SIMD extensions on Intel x86-processors in our evaluation. Moreover, we discuss interesting research challenges for an efficient analytical query processing on wide vector registers.

Outline: The remainder of the paper is structured as follows: In Section II, we briefly introduce our SIMD abstraction layer *TVL* and describe the NEC vector engine. Thereafter in Section III, we present two major steps to master the vector engine in our TVL. Based on that, we present selected evaluation results for micro-benchmarks as well as for an end-to-end evaluation using the Star-Schema-Benchmark in Section IV. Then, we summarize the paper in Section V.

II. Preliminaries

Analytical queries usually access a small number of columns, but a high number of rows and are, thus, most efficiently processed by column-store systems [2], [10]. In those systems, all values of every column are encoded as a sequence of integer values, so that the whole query processing is done on these integer sequences [2], [10]. To increase the query performance, vectorization is a state-of-the-art optimization technique nowadays, because all recent x86-processors offer powerful SIMD extensions [1]–[5]. To achieve the best performance, explicit vectorization using SIMD intrinsics is still the best way [4], [7], whereas intrinsics are functions wrapping the underlying machine calls. However, these SIMD extensions are increasingly diverse in terms of (i) the number of available vector instructions, (ii) the vector length, and (iii)



(a) Template Vector Library (TVL) (b) Vector Engine Architecture

Fig. 1. Architectures of the Template Vector Library (TVL) and the NEC Vector Engine (VE).

the granularity of the bit-level parallelism, i.e., on which data widths the vector instructions are executable [7]. To hide this heterogeneity, we developed a specific abstraction layer called *Template Vector Library (TVL)* for column-stores [7].

A. Template Vector Library

Our abstraction approach follows a separation of concerns concept as shown in Fig. 1(a). On the one hand, it offers hardware-oblivious but column-store specific primitives, which are similar to intrinsics. In that sense, the state-of-the-art vectorized programming approach does not change, but the explicit vectorization can be done in a hardware-independent way. Moreover, we organized the necessary primitives in seven classes like load/store or an arithmetic class for a better organization including a unified interface per class [7]. On the other hand, our *TVL* is also responsible for mapping the provided hardware-oblivious primitives to different SIMD extensions. For this mapping, our *TVL* includes a plug-in concept and each plug-in has to provide a hardware-conscious implementation for all primitives.

From an implementation perspective, our abstraction concept is realized as a header-only library, where the hardware-oblivious primitives abstract from SIMD intrinsics. These primitives are generic functions representing a unified interface for all SIMD architectures. In addition to the primitives, we introduced generic datatypes:

base_t: The base type can be any scalar type.

vector_t: The vector type contains one or more values of the same base type.

mask_t: A mask is a scalar value, which is large enough to store one bit for each element in a vector.

Using the provided primitives and the data types, we can implement columnar query operators in a hardware-oblivious way. For the hardware-conscious mapping, we use template metaprogramming requiring hardware-conscious implementations for all primitives and for all underlying SIMD extensions. This function template specialization has to be implemented, whereby the implementation depends on the available functionality of the SIMD extension. In the base case, we can directly map a *TVL* primitive to an SIMD intrinsic. However, if the necessary SIMD intrinsic is not available, we are able to implement an efficient workaround in a hardware-conscious way. This implementation is independent of any query operator and must be done *only once* for a specific SIMD extension.

B. NEC Vector Engine

NEC Corporation developed a new vector engine (VE) as co-processor and the architecture of this VE is illustrated in Fig. 1(b). The VE consists of 8 vector cores, 6 banks of HBM2 high speed memory, and only one shared last-level cache (LLC) of size 16MB. The LLC is on both sides of the vector cores, and it is connected to each vector core through a 2D mesh network-on-chip with a total cache bandwidth of 3TB/s [9]. This design provides a memory bandwidth of up to 1.2TB/s per vector engine [9]. Each vector core consists of three core units: (i) a scalar processing unit (SPU), (ii) a vector processing unit (VPU), and (iii) a memory addressing vector control and processor network unit (AVP). The SPU has almost the same functionality as modern processors such as fetch, decode, branch, add, and exception handling, but the main task is to control the status of the vector cores.

The VPU has three vector fused multiply add units, which can be independently executed by different vector instructions, whereby each unit has 32 vector pipelines consisting of 8 stages [9]. The vector length of the VPU is 256 elements. One vector instruction executes 256 arithmetic operations within eight cycles [9]. The major advantage, compared to wider SIMD functionalities e.g., in Intel processors like AVX-512, is that the operations are not only executed spatially parallel, but also temporally parallel, which hides memory latency better [9]. Each VPU has 64 vector registers and each vector register is 2Kb in size (32 pipeline elements with 8 Byte per element). Thus, the total size of the vector registers is 128Kb per vector core, which is larger than an L1 cache in modern x86-processors. To fill these large vector registers with data, the LLC is directly connected to the vector registers with a bandwidth of roughly 400GB/s per vector core [9].

In [11], we presented a comprehensive experimental evaluation of this VE for analytical queries using selective in-memory column-store operators. In particular, we showed the benefits of this VE compared to regular SIMD extensions in single-threaded as well as multi-threaded environments.

```
__vr vy = _vel_vsfa_vvssl(p_vec, 3, reinterpret_cast<uint64_t>(p_DataPtr), element_count);
return _vel_vgt_vvssl(vy, 0, 0, element_count);
```

Fig. 2. The *gather* implementation for 64-bit values on the NEC vector engine involves two intrinsics and an additional vector register. In a first step, *element_count* addresses are determined, from which data is read in the second step. *P_vec* is a function parameter containing the index offsets. *P_DataPtr* is also a function parameter pointing to an address in memory, from where the data should be gathered.

III. Mastering NEC Vector Engine

The focus of this paper is to present an approach to fully support the NEC VE in our *TVL* as highlighted in Fig. 1. For that, we created a VE-specific hardware-conscious implementation of the hardware-oblivious interface. Apart from mapping to abstract datatypes as explained above, this requires (i) a hardware-conscious realization of the provided primitives and (ii) additional primitives to work with large masks.

A. Hardware-Conscious Primitive Implementation

For the hardware-conscious implementation, we have to distinguish three different main groups of primitives across all *TVL* classes [12]: (i) load/store primitives, (ii) element-wise primitives, and (iii) horizontal primitives. The hardware-specific implementation for them on the VE can be done by using a limited number of intrinsics supporting only 32-bit and 64-bit data widths. An overview of these intrinsics and their asm mapping can be found on the developer's github page[1].

1) Load/Store Primitives: Load and store primitives are required to get data either into or out of vector registers, whereas the source or destination can be main memory or a scalar value on the stack. The primitives for loading or storing data sequentially can directly be mapped to a single intrinsic. The same holds true for broadcasting a scalar value into a vector register. However, primitives including random memory access, require a workaround. For instance, the gather primitive uses two intrinsics and an additional vector register. The corresponding source code for 64-bit values is shown in Fig. 2. Because of the different interfaces and data locations, we separated the Load/Store primitives into 3 different *TVL* classes as shown in Fig. 1:

- **Load/Store (L/S)** For all primitives involving main memory access, e.g., *load, store, gather*.
- **Create** For filling a vector with immediate values or the contents of variables, e.g., *set sequence*.
- **Extract** For extracting scalar values from a vector. This class only contains a single primitive: *extract value*.

2) Element-wise Primitives: Element-wise primitives are characterized by the feature that they do not introduce dependencies between the elements of the same vector register, e.g., element-wise arithmetic, comparisons, or boolean logic. For arithmetic operations or boolean logic, the primitives return another vector, while comparisons return a bitmask. Those primitives returning a vector, can directly be mapped to an intrinsic. However, primitives returning a bitmask involve at least two intrinsics: (i) a *maskable function*, which fills a vector register with a positive value, 0, or a negative value, depending on the result of an element-wise vector comparison, and (ii) a function creating a mask out of this result. According to the different interfaces and purposes of the element-wise primitives, we divided them into 3 *TVL* classes: **arithmetic, logic,** and **comparison**.

3) Horizontal Primitives: Horizontal primitives do not treat the elements of a vector independently. Examples are (i) the *horizontal addition*, (ii) the *compress-store* storing selected vector elements sequentially, or (iii) a *rotation* of the vector elements. All of these primitives require a workaround on the NEC VE since they are not supported natively. The workaround for the *horizontal addition* uses a first intrinsic to sum up all elements in a vector register, and a second intrinsic to extract the result of this operation out of the vector register. While the *horizontal addition* fits into our arithmetic *TVL* class, and the *compress-store* is realized as a part of our load/store class, the *rotation* is a part of the **manipulate** class. This class contains primitives, which change the sequence of the elements within a vector register.

As expected, the hardware-conscious implementation is more or less straightforward. However, the limited number of vector instructions exposed by the intrinsics leads to the fact that some primitives can only be realized via workarounds.

B. Enhanced Mask Primitives

When implementing vectorized column-store operators using primitives or intrinsics, bitmasks (or masks in short) often have to be used and combined, e.g., by shifting them or using boolean logic [3]–[5]. A simple example is a variant of the vectorized intersection of two index lists [12]:

1) An initial bitmask is set to 0.
2) Then, the outer loop broadcasts the current value of one index list into a vector register A.
3) The inner loop sequentially loads values from the second index list into a vector register B.
4) Registers A and B are compared. The result is a bitmask.
5) If a match was found, the inner loop stops, the initial bitmask is shifted by one bit, and a logical OR is performed on the shifted initial mask and the result mask.
6) When there are no more values left in the second index list and there has still been no match, the result mask remains 0.
7) Finally, the next iteration of the outer loop starts with an incremented pointer to the data of the first index list. The pointer to the data of the second index list is reset to the position of the last match.
8) The steps 2-7 are repeated until the initial bitmask was shifted completely, e.g., until a 64-bit mask was shifted 63 times. Then the outer loop stops and the values from the first relation are stored according to the bits set in the initial bitmask. This is done using the *compress-store* primitive.

[1]https://sx-aurora-dev.github.io/velintrin.html. A more thorough insight into the instruction set may be found in the manual [13].

Class	L/S			Arithmetic			Boolean Logic			Create			Manipulate			Comparison			Extract		
	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)
AVX512	4	0	0	9	2	0	4	0	0	2	1	0	1	1	0	9	0	0	0	0	1
Tsubasa Aurora	2	2	0	8	2	1	4	0	0	2	0	1	1	0	1	2	7	0	1	0	0

(a) 1:1 mapping to intrinsic/native scalar code on masks | (b) Workaround using intrinsics | (c) Workaround using significant amount of scalar code

Fig. 3. Comparing the hardware-conscious implementations of AVX-512 with NEC VE in terms of mappings.

The vectorized intersection contains 3 operations on masks. In step 0, the mask is initialized. In step 5, a mask is shifted and then combined with another mask via a logical OR. These operations are common on scalar values, but a scalar value can only hold up to 64 bits. Thus, when a vector register contains more than 64 values, a scalar is not sufficient anymore.

For NEC VE holding up to 256 elements, the mask contains 256 bit, which are stored in dedicated mask registers. The scalar operations on these masks are not defined, but there are already primitives performing these operations on vector registers. A logical step to add the missing functionality in our *TVL*, is to explicitly create these primitives for mask registers allowing a direct use instead of the standard operators.

The complexity of the hardware-specific implementations of these mask primitives varies. For all masks containing 64 bit or less, it is a simple fallback to the scalar operators. With the 256-bit mask on the NEC VE, direct mappings to intrinsics are sometimes possible. For instance, in our example, we could directly map the initialization of the mask (step 0) and the logical OR (step 5) to intrinsics. However, shifting left by one bit across 8 Byte lane boundaries requires a workaround involving the following steps:

1) Copy the mask into a 4 x 64 bit array A.
2) Copy the mask into a second array B, but increase the destination index by 1, such that the least significant value remains empty.
3) Set the least significant 64 bit of array B to 0.
4) Shift all elements in array A left by 1 bit.
5) Shift all elements in array B right by 63 bit (carry).
6) Perform a logical OR on array A and array B.
7) Write the result from step 6 back into a mask register.

Obviously, there are more general solutions for shifts by larger distances, but since the shift by 1 is by far the most widely used in our operator implementations, this specialized solution makes sense. For completeness, the source code for the mentioned mask primitives is shown in Fig. 4.

```
//Bitwise OR on masks
return _vel_orm_mmm(p_mask1, p_mask2);
```

```
//Initialize mask with 0
return _vel_vfmklaf_ml(element_count);
```

```
//Shift mask by 1 bit
uint64_t masks[4]; uint64_t masks_carry[4];
masks_carry[3] = 0;
for (unsigned i=0; i<3; i++){
  masks_carry[2-i] = _vel_svm_sms(p_mask,i);
  masks_carry[2-i] = masks_carry[2-i] >> 63;}
for (unsigned i=0; i<4; i++){
  masks[3-i] = _vel_svm_sms(p_mask,i);
  masks[3-i] = masks[3-i] << 1;
  masks[3-i] |= masks_carry[3-i];}
__vm256 final_mask =_vel_vfmklaf_ml(256);
for (unsigned i=0; i<4; i++){
  final_mask =
    _vel_lvm_mmss(final_mask, i, masks[3-i]);}
return final_mask;
```

Fig. 4. The hardware-conscious implementation of different mask primitives on the NEC VE. While some primitives map directly to an intrinsic, others require workarounds.

C. Summary

To fully support the NEC VE in our *TVL*, we (i) implemented a hardware-conscious realization of the provided hardware-oblivious primitives and (ii) added additional primitives to work with large masks. Fig. 3 compares the NEC VE with the AVX-512 hardware-conscious realization for 64-bit data widths in terms of mapping quality. For example, the four load/store TVL primitives can be directly mapped to SIMD intrinsics in the case of AVX-512, while in the case of NEC VE, two primitives require workarounds using intrinsics. The same behavior is observable for the other TVL classes with an exception for the extract class. Here, the included primitive requires a workaround with a significant amount of scalar code for AVX-512, i.e., a switch-case statement, while it can be mapped to an intrinsic for NEC VE. However, our NEC VE hardware-conscious implementation requires more workarounds than the AVX-512 implementation.

IV. Evaluation

Our entire evaluation is based on *MorphStore*, which is an in-memory columnar analytical query engine completely implemented using TVL [3]. All experiments were conducted on a NEC-SX Aurora TSUBASA machine equipped with (i) an Intel Xeon Gold 6126 processor with 96GB of main memory as vector host (VH) and (ii) a single vector engine (VE) with 24GB of main memory with a maximum memory bandwidth of 750GB/s. While the VH features all Intel SIMD extensions SSE (128-bit), AVX2 (256-bit) and AVX-512 (512-bit), the VE operates on vector registers of size 16,384-bit. Since the main goal of vectorization is to increase the single-thread performance, all experiments were executed single-threaded on unsigned 64-bit integers and happened entirely in-memory.

A. Micro-Benchmarks

Our micro-benchmarks investigate single query operators and simple queries on synthetically generated data.

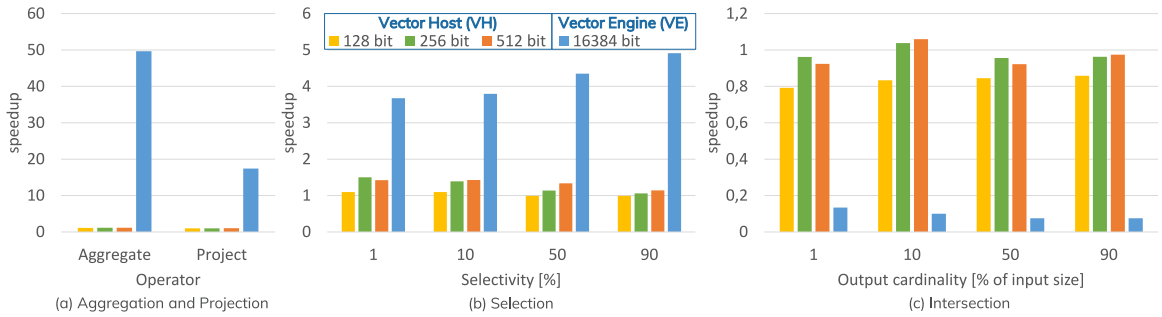


Fig. 5. Speedups for different operators. The benefit of vectorization depends on the operator.

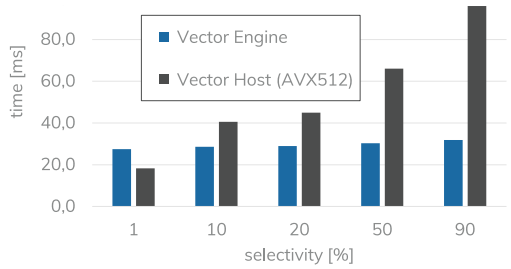


Fig. 6. Runtime comparison of a simple aggregation-query between VE and VH (AVX-512). The selectivity of the `select` operator is varied.

1) ***Operators:*** In a first set of experiments, we compared the effect of the vectorized solution to a scalar solution on an operator level. For that, we ran four different operators: `Aggregation`, `Projection`, `Selection`, and `Intersection` on fixed input sets. Each operator was built for all available vector sizes, and for scalar processing on the VE and VH. On the VE, the scalar code runs on the SPU instead of the VPU. After running the operators, we computed the speedup for each vector size over its native scalar processing method. We expect that those operators involving mostly sequential memory access and no workarounds for the hardware-specific implementations benefit the most from vectorization. An operator with these properties is the `Aggregation` as depicted in Fig. 5(a). While it already benefits from smaller vector registers on the VH (speedup between 1.1 and 1.2), the VE fully exploits the potential of SIMD processing with a speedup of almost 50 compared to a scalar processing. A similar effect can be observed for the `Projection` (Fig. 5(a)), whereby this operator involves random memory read access but the write access is sequential.

In Fig. 5(b), the speedup of the `Select` operator is shown for different selectivities. This operator requires to remove unmasked elements within a register before storing them. This primitive is called *compressstore* being a costly operation not natively supported by any of the used SIMD instruction sets, except for AVX512. Thus, the speedup on VE is not as high as for the previous operators. However, there is still a significant speedup, which is increased along with the vector size.

Lastly, we run an `Intersection` as explained in Section III-B. The described algorithm for vectorized intersections is only effective for very low output cardinalities ($\ll 1\%$ of the input size). For larger cardinalities, more unnecessary values are read from the main memory the larger the vector register is. Additionally, workarounds for mask operations are involved. For these reasons, the intersection (Fig. 5(c)) does not benefit from vectorization. Moreover, there is a significant performance loss for large registers.

2) ***Simple Queries:*** As we have shown, there are operators that benefit largely from extremely wide vectors. We used three of these operators to generate a simple aggregation query for the next micro-benchmark: a selection, a projection, and an aggregation. For this query template, we generated two input columns and varied the selectivity of the selection predicate. We ran this query on the VH and on the VE. Fig. 6 shows the execution time for different selectivities on both components. The graph shows, that the VE offers a robust behavior while the runtime on the VH increases if the selectivity grows. Moreover, for large selectivities, the VE outperforms the VH by a multiple of 3. Note that the runtime on the VE is also growing linearly, but the rise is too small to be visible.

B. End-to-End Evaluation

In our end-to-end evaluation, we investigated the Star Schema Benchmark (SSB) [14]. In this paper, we only show the results for query Q1.1 executed on the VE, whereby we ran this query on the SPU (scalar execution) and on the VPU (vectorized execution). All benchmark data (scale factor 1) was placed beforehand on the VE. Besides the overall query runtime, we also measured the runtimes of each single query operator. Fig. 7 shows the results as absolute numbers (left) and as a percentage of the overall runtime of the query (right). As shown above, some operators benefit from vectorization on the VE, while others suffer from a performance loss, which is also observable here. While the `selections` take only a small fraction of the query runtime in the vectorized query, the `intersections` and the `join` operator require significantly more time than in the scalar query execution. Vice versa, during scalar query execution, the `selections` consume a significant fraction of the overall execution time. All other operators, i.e., `projections`, `calculations`, and `aggregations`, are faster on the VPU than on the SPU. However, because of their comparatively short runtime, they are barely visible in Fig. 7.

To get the best of both worlds, we combined vectorized and scalar execution, which can be easily done using our *TVL* approach by assigning different template arguments to

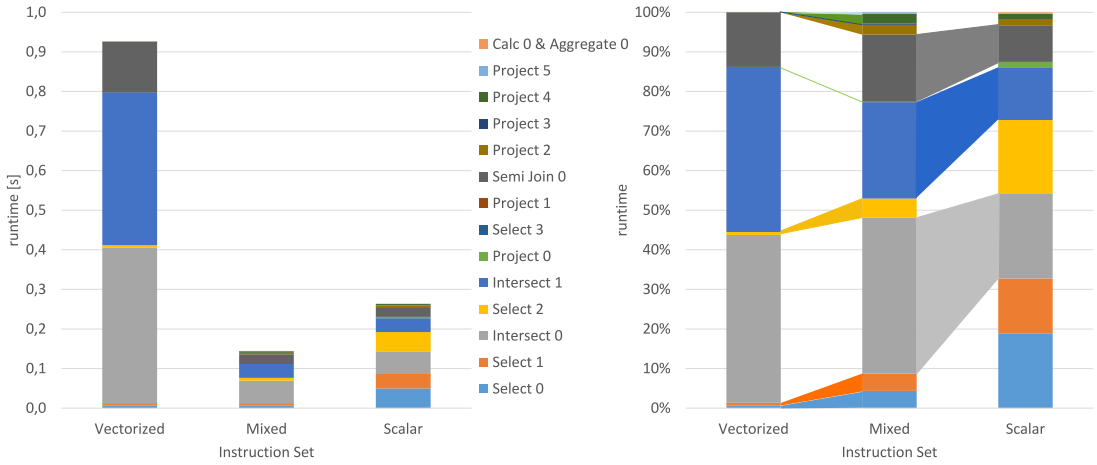


Fig. 7. Runtime breakdown of SSB Query 1.1 on the VE: **Left:** Absolute time for each operator with different instruction sets. **Right:** Runtime of each operator relative to overall query runtime. For better identification, each operator has an index. The first query operator, which is executed, is *Select 0*. The last executed query operators are *Calc 0* and *Aggregate 0*.

the operator calls. This way, individual query operators are mapped to different TVL backends during query compile time. The result of this experiment is shown in the *mixed* bar of Fig. 7. With this mixed execution, we are able to decrease the overall query runtime compared to a full vectorized as well as to a full scalar execution. We observed the same behavior for all other queries as well.

C. Lessons Learned and Future Work

Based on our evaluation, we conclude the following two aspects. On the one hand, extremely wide vector registers can help to improve the query/operator processing performance. Unfortunately, that does not hold for all vectorized operators in a straightforward way. This means that new specialized processing methods have to be developed to be able to use extremely wide vector registers efficiently. Important operators would be intersect or join, because the current implementations for both operators do not really benefit from larger vectors.

On the other hand, different vector register sizes and instruction sets can be present on the same system with the scalar execution as special case with a vector length of one. As shown in our evaluation, the mixed execution is an effective optimization approach to decrease the overall query runtime. This optimization becomes even more interesting in the context of ARM SVE [15], where the vector length is an implementation choice ranging from 128 to 2048 bits, in increments of 128 bits. Thus, this optimization should be investigated in more detail, for this, our *TVL* provides the right foundation.

V. Conclusion

NEC Corporation offers a vector engine as a specialized co-processor having interesting features for an efficient query processing perspective. To use this vector engine, we described the mastering using a column-store specific abstraction layer for vectorization in this paper. As we have shown in the evaluation, the vector engine has some benefits and shortcomings compared to SIMD extensions on Intel x86-processors and our approach opens up a rich bouquet of opportunities for future research activities.

References

[1] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column oriented database systems," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1664–1665, 2009.

[2] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, "The design and implementation of modern column-oriented database systems," *Found. Trends Databases*, vol. 5, no. 3, pp. 197–280, 2013.

[3] P. Damme, A. Ungethüm, J. Pietrzyk, A. Krause, D. Habich, and W. Lehner, "Morphstore: Analytical query engine with a holistic compression-enabled processing model," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 2396–2410, 2020.

[4] O. Polychroniou and K. A. Ross, "VIP: A SIMD vectorized analytical query engine," *VLDB J.*, vol. 29, no. 6, pp. 1243–1261, 2020.

[5] J. Zhou and K. A. Ross, "Implementing database operations using SIMD instructions," in *SIGMOD*, 2002, pp. 145–156.

[6] C. J. Hughes, *Single-Instruction Multiple-Data Execution*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2015.

[7] A. Ungethüm, J. Pietrzyk, P. Damme, A. Krause, D. Habich, W. Lehner, and E. Focht, "Hardware-oblivious SIMD parallelism for in-memory column-stores," in *CIDR*. www.cidrdb.org, 2020.

[8] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *IEEE Micro*, vol. 32, no. 3, pp. 122–134, 2012.

[9] 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 *SC*, 2018, pp. 54:1–54:12.

[10] F. Faerber, A. Kemper, P. Larson, J. J. Levandoski, T. Neumann, and A. Pavlo, "Main memory database systems," *Found. Trends Databases*, vol. 8, no. 1-2, pp. 1–130, 2017.

[11] 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*, vol. 19, no. 3, pp. 183–197, 2019.

[12] B. Schlegel, R. Gemulla, and W. Lehner, "Fast integer compression using SIMD instructions," in *DaMoN@SIGMOD*, 2010, pp. 34–40.

[13] NEC Corporation, *SX-Aurora TSUBASA Architecture Guide, revision 1.1*, 2018. [Online]. Available: https://www.hpc.nec/documents/guide/pdfs/Aurora_ISA_guide.pdf

[14] J. Sanchez, "A review of star schema benchmark," *CoRR*, vol. abs/1606.00295, 2016.

[15] 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*, vol. 37, no. 2, pp. 26–39, 2017.