# Towards Porting Hardware-Oblivious Vectorized Query Operators to GPUs

Johannes Fett, Annett Ungethüm, Dirk Habich, Wolfgang Lehner
Database Systems Group, Technische Universität Dresden, Dresden, Germany
{johannes.fett,annett.ungethuem,dirk.habich,wolfgang.lehner}@tu-dresden.de

## ABSTRACT

Nowadays, query processing in column-store database systems is highly tuned to the underlying (co-)processors. This approach works very well from a performance perspective, but has several shortcomings from a conceptual perspective. For example, this tuning introduces high implementation as well as maintenance cost and one implementation cannot be ported to other (co-)processors. To overcome that, we developed a column-store specific abstraction layer for hardware-driven vectorization based on the Single Instruction Multiple Data (SIMD) parallel paradigm. Thus, we are able to implement vectorized query operators in a hardware-oblivious manner, which can be specialized to different SIMD instruction set extensions of modern x86-processors. To soften the limitation to x86-processors, we describe our vision to integrate GPUs in our abstraction layer by interpreting GPUs as virtual vector engines in this paper. Moreover, we present some initial evaluation results to determine a reasonable virtual vector size. We conclude the paper with an outlook on our ongoing research in that direction.

## 1. INTRODUCTION

Analytical database queries typically access a small number of columns, but a high number of rows and are, thus most efficiently answered by column-store database systems [1]. Since the amount of data is still growing, these systems constantly adapt to novel hardware features to satisfy the requirements of high query throughput and low query latency [2, 3, 4]. From the hardware perspective, we see that Moore's Law is still valid and the transistors on a chip double about every two years [5]. Unfortunately, we also see an end of Dennard scaling, so that not all transistors can be active due to power constraints [5]. To deal with that, vectorization, parallelization, specialization and heterogeneity are key approaches for hardware designers [5].

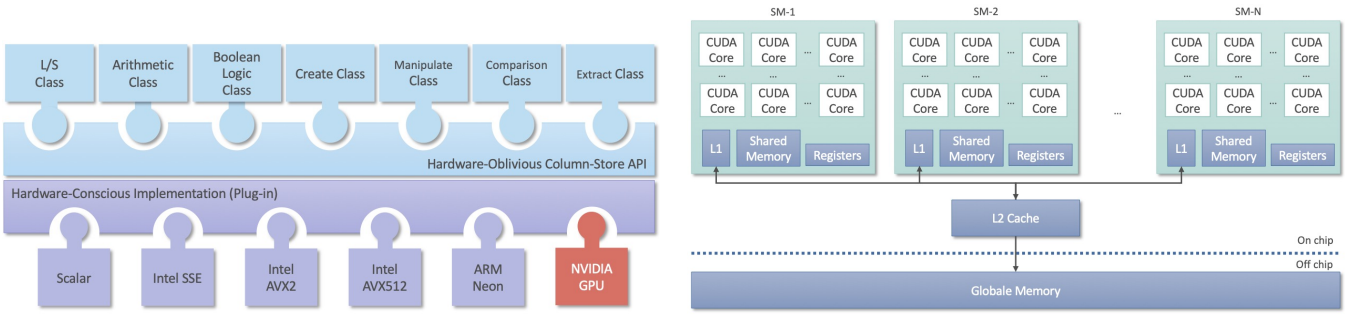
For this reason, *vectorization* based on the *Single Instruction Multiple Data (SIMD)* parallel paradigm has established itself as a core query optimization technique in column-store systems [1, 6, 7]. SIMD increases the single-thread performance by executing a single operation on multiple data elements in a vector register simultaneously (data parallelism) [8]. Such SIMD capabilities are common in today's mainstream x86-processors using specific SIMD instruction set extensions, whereas a current hardware trend is that these extensions are growing not only in terms of complexity of the provided instructions but also in the size of the vector registers (number of data elements in parallel). To tackle the evolving SIMD-specific diversity, we developed a novel abstraction layer called *Template Vector Library (TVL)* for in-memory column-stores [6]. Using that *TVL*, we are able to implement hardware-oblivious vectorized query operators, which can be specialized to different SIMD instruction set extensions at query compile-time [6].

Besides vectorization, hardware also shifts from homogeneous x86-processors towards heterogeneous systems with different computing units (CU) [5]. In this context, there is already a huge number of research works that deal with the use of different CUs such as GPUs or FPGAs for an efficient analytical query processing [2, 3, 4, 9, 10, 11]. In general, these works have shown the great potential, but all these approaches have a common shortcoming. For each CU, a separate hardware-conscious and hand-tuned codebase for query operators has to be implemented and maintained using different programming concepts. From a performance perspective, this approach works very well, but the efforts for implementation and maintenance often outweigh the benefits. To overcome that, our overall vision is to enhance our SIMD abstraction layer *TVL* to cover different heterogeneous CUs as well.

**Our Contribution and Outline.** In this paper, we describe our vision and present some initial steps to integrate GPUs in our abstraction layer by interpreting GPUs as virtual vector engines. Generally, GPUs use a *Single Instruction Multiple Thread (SIMT)* execution model which can be also interpreted as SIMD combined with multi-threading. Based on that, GPUs seem like a perfect match for our *TVL* to soften the limitation to x86-processors. Thus, our contributions are the following in this paper:

1. We start with an introduction in our SIMD abstraction layer TVL as well as with an architectural description of NVIDIA GPUs in Section 2.
2. Then, we present our general idea of SIMDization of GPUs in Section 3. To interpret a GPU as virtual vector engine, we have to determine a reasonable virtual vector size, which can be most efficiently processed in parallel. The vector size is important because it is an

$32^{nd}$ GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), September 01-03, 2021, Munich, Germany.




(a) Template Vector Library (TVL)

(b) NVIDIA GPU Architecture

**Figure 1: Architectures of the *Template Vector Library (TVL)* and of NVIDIA GPUs.**

essential part of SIMD and an integral part of our *TVL*. For that, we present some *TVL*-oriented experiments. Based on the results, we derive a virtual vector size.

3. In Section 4, we summarize our lessons learned and describe our ongoing activities in that direction.
4. We close the paper with related work in Section 5 and a short summary in Section 6.

## 2. BACKGROUND

In this section, we introduce our *Template Vector Library* as SIMD abstraction layer for column-stores. Moreover, we briefly describe the GPU architecture and execution model.

### 2.1 Template Vector Library

Vectorization is a state-of-the-art query optimization technique in in-memory column-stores, because all recent x86-processors offer powerful SIMD extensions [1, 7, 12, 13, 14]. SIMD provides data parallelism by executing a single instruction on multiple data elements simultaneously [8]. To achieve the best performance, explicit vectorization using SIMD intrinsics is still the best way [6, 13, 14], 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) the granularity of the bit-level parallelism, i.e., on which data widths the vector instructions are executable [6]. To hide this heterogeneity, we developed a specific abstraction layer called *Template Vector Library (TVL)* for column-stores [6].

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. The primitives are derived from state-of-the-art vectorized columnar query operators. We organized these primitives in seven self-descriptive classes like load/store (L/S) or an arithmetic class for a better organization including a unified interface per class [6]. 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 data types:

**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 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 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.

Figure 2 illustrates how a hardware-oblivious vectorized operator using our TVL looks like and how it can be called. We show a simple aggregation (summation) operator consisting of four TVL primitives:

`set1:` fills a vector register with a given value

`load:` loads multiple consecutive data elements into a vector register

`add:` executes an element-wise addition on data elements in two vector registers

`hadd:` executes an horizontal addition on data elements in one vector register

The aggregation operator assumes that the number of data elements is a multiple of the number of elements per vector and works as follows: One vector register called `resVec` is filled with zeros. Afterwards, the operator iterates over the input column `in` and loads a number of consecutive data elements into a second vector register called `dataVec`. Then, the data elements in both vector registers are added element-wise and the result is stored in `resVec`. When all elements of the input column have been processed, the horizontal aggregation `hadd` is carried out to determine the final sum result.

To specialize this hardware-oblivious operator implementation during query compile-time, we use three template parameters called `processingStyle (ps)`: (i) the vector extension (e.g., SSE, AVX, NEON, or scalar), (ii) the vector size in bit, and (iii) the base data type with bit granularity (e.g., int8, int64, float). The definition of `ps` is shown in Figure 2, which is used to call the aggregation-operator.

### 2.2 GPUs

```
// 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::v256<uint64_t>>; // for example
size_t count = 1024;
uint64_t * array = generate_data(count);
uint64_t sum = agg<ps>(array, elemCount);
```

**Figure 2: A simple sum-aggregation operator using the TVL [6].**

Graphics Processing Units (GPUs) are increasingly used for large-scale query processing in database systems [2, 3, 4, 15]. Specifically, their hardware parallelism and memory access bandwidths contribute to considerable speedups. Figure 1(b) depicts a simplified architecture of an NVIDIA GPU. Generally, a modern GPU consists of (i) a large global main memory with a memory bandwidth of up to 1.2 TB/s and (ii) a number of compute units called *Streaming Multiprocessors (SMs)*. Each SM has a number of simplistic cores, a fixed set of registers, and shared memory. This shared memory serves as scratchpad and can be accessed by all cores in the SM. Moreover, the GPU has an on-chip L2 cache, which is shared across all SMs and optionally, each SM may have a local L1 cache. The number of SMs, and cores per SM, the size of global memory, the size of the L2 cache, etc. varies across GPU products.

The execution model of GPUs is called *Single Instruction Multiple Threads (SIMT)*, whereas SIMT is very similar to SIMD. While multiple data are processed by a single instruction in SIMD, multiple threads are processed by a single instruction in lock-step. That means, each thread in SIMT executes the same instruction, but on different data. While a thread switch is very costly on CPUs, GPUs can handle thread switching with more ease. GPUs feature a fast thread switching based on groups of 32 threads called warps. The warp scheduler issues instructions to warps available on an SM beyond the number of physical cores to hide latency [16]. Thus, it is encouraged to create more threads than are available as physical cores. Then, this overload can be used to schedule threads for execution, while others wait for a memory transfer. This is especially important since databases are more likely I/O-bound, not CPU-bound, and it is thus one of the most important features for implementing query operators on GPUs [4].

For the GPU implementation, general purpose parallel programming models such as CUDA [17] or OpenCL [18] have to be used, whereas CUDA generate better performance results. The CUDA programming model consists of two code parts: host code and GPU code. The host code runs on a CPU process and is responsible for setting up the environment, memory transfers between CPU and GPU and executing kernels on the GPU. On the GPU, a function called kernel is executed in parallel with a number of threads and blocks. Blocks consist of a number of threads and are assigned to one streaming multiprocessor. Threads are assigned to a block. Thus, executing a kernel requires at least 2 parameters. The first parameter defines how many blocks will be spawned. The second parameter sets the number of threads per block. The total number of GPU threads is the product of both parameters. Depending on the data size and the number of GPU threads, the number of elements processed per threads can vary.

## 3. GPU-SIMDIZATION EXPERIMENTS

Our overall vision is to fully support GPUs in our *TVL* as shown in Figure 1(a). For that, we have to create a GPU-specific hardware-conscious implementation of the hardware-oblivious interface. To achieve that, we interpret GPUs as virtual vector engines and call this interpretation *SIMDization*. For an optimal SIMD processing, we have to find a reasonable vector size. This vector size is virtual because we want to interpret regular arrays as virtual vector registers. For example, on Intel x86-processors, the different SIMD instruction set extensions either have a vector length of 128-, 256- or 512-bit. If we assume 64-bit per data element, we can simultaneously process 2 elements in a vector register of the size 128-bit. The wider the vector registers, the more data can be processed in parallel. We are not aware of any work that has ever determined a vector size for GPUs.

### 3.1 Vector Length Evaluation

To determine a reasonable virtual vector size for our *SIMDization* approach for GPUs, several experiments were executed. A first set of experiments were conducted on an NVI-

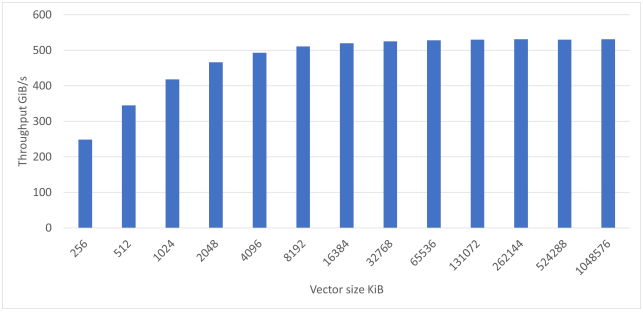



**Figure 3: Evaluation of element-wise primitive add.**

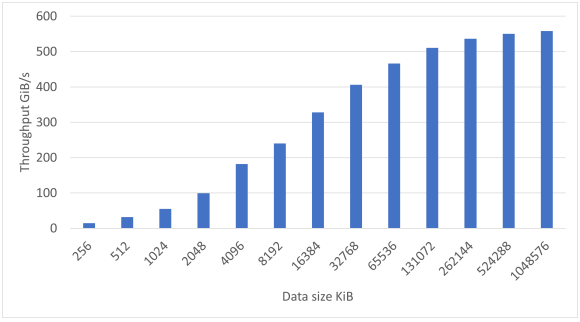



**Figure 4: Evaluation of horizontal primitive hadd.**

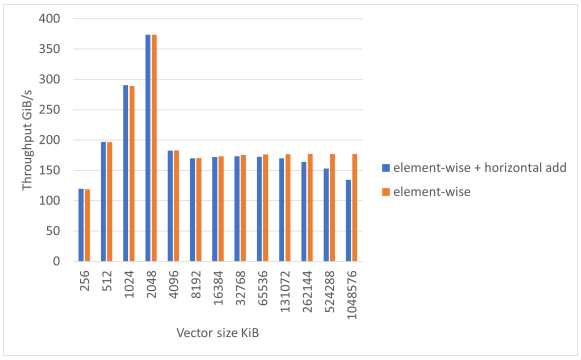



**Figure 5: Evaluation of Operator Aggregation.**

DIA RTX Quadro 8000 GPU. This GPU has a global main memory size of 48 GiB, a memory bandwidth of 672 GB/s, an L2 cache with a size of 6 MiB, and 72 SMs. Each SM has 64 cores resulting in a total of 4,608 cores.

For the hardware-conscious implementation, we can distinguish three different main groups of hardware-oblivious SIMD primitives across all *TVL* classes: (i) load/store primitives, (ii) element-wise primitives, and (iii) horizontal 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. In contrast to that, horizontal primitives do not treat the elements of a vector independently. An example is the *horizontal addition*. As shown in our example sum-aggregation operator (cf. Figure 2), different primitives from all groups are used to implement a vectorized query operator. In this operator, we use an element-wise and a horizontal add primitive.

In our first experiments, we evaluated the element-wise and horizontal addition on the GPU. For that, we implemented simple vectorized CUDA kernels and executed these kernels using different vector sizes. For the horizontal addition, we use the function `cub::DeviceReduce` from the CUDA SDK 11.2 as most efficient implementation as foundation. Since our GPU has 4,608 cores and usually more threads than cores are used for good performance, we investigated vector sizes in the range from 256 KiB to 1 GiB. In terms of number of elements, we evaluated the range from $2^{13}, .., 2^{27}$ number of 64-bit data elements. For each vector size, different CUDA configurations with blocks and threads per block are possible and we tested a large number of different configurations in a systematical way.

**Element-wise Addition:** This kernel was performed on two input columns A and B. Each column was a sequence of unsigned randomly generated 64-bit integer values with a size of 1 GiB. The result was written back into column B. Within a loop, our vectorized kernel is called until the whole columns have been processed. Figure 3 shows the best achieved throughput over all configurations for each vector size. As we can see, small vector sizes negatively affect the performance. For a vector size of 256 KiB, a throughput of 248.87 GiB/s is achieved. The best performance is gained by using a vector size of 1 GiB. This results in a throughput of 530.99 GiB/s. A vector size of 8 MiB is 3.8% slower than a vector size of 1 GiB. We conclude, higher vector sizes lead to higher throughput and in the best case, the vector size corresponds to column size.

**Horizontal Additon:** This kernel adds all elements in a vector register together and the kernel is based on the `cub::DevideReduce` function being shipped as part of the CUDA SDK. For our evaluation, we generated different columns where the column size corresponds to the vector size. Again, the data elements are 64-bit unsigned integer values. Figure 4 shows the best achieved throughput over all configurations for each vector size. Similar to the *element-wise addition*, higher vector sizes lead to higher throughput.

**Aggregation Operator:** Based on these evaluations, we could draw the conclusion that the best vector size should be the column size. To validate our hypothesis, we evaluated the sum-aggregation operator consisting of both primitives as next. For this evaluation, we generated a single column with a size of 1 GiB and varied the vector size from 256 KiB to 1 GiB. As illustrated in Figure 5, we obtain a completely different result. The throughput increases up to a vector size of 2 MiB. Then, the throughput decreases and stabilizes at a low level. The main difference to our previous experiments is that a single vector register or array is now the main driver of the processing. As shown in Figure 2, one vector register `resVec` is filled with zeros at the beginning of the aggregation operators. Afterwards, we repeatedly load into a second vector $dataVec$ of the column and conduct an element-wise add between both. The result is stored in `resVec` vector. This `resVec` vector is frequently accessed, and thus, should be kept in cache.

To evaluate the cache-fitting in more detail, we slightly modified our experiment. We generated a new input column of size 1,5 GiB containing 64-bit unsigned integers and varied the vector size according to the on-chip L2 cache size: 1/4, 1/3, 1/2, 1, 2 times of the cache size. Figure 6 depicts the resulting throughputs. As we can see, we obtain the best performance when our vector size is a third of the L2 cache size. Larger vector sizes lead to a lower throughput.

## 3.2 Comparing with Native CUDA

As shown above, we are able to determine a reasonable

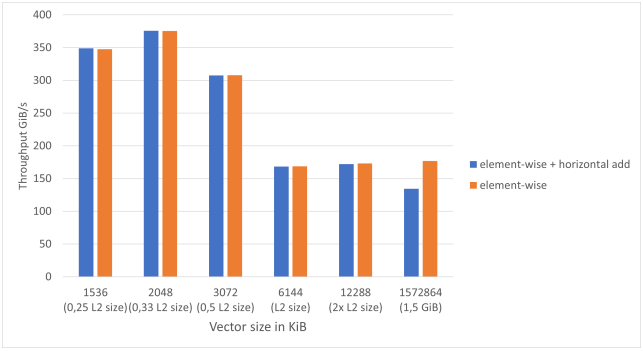



**Figure 6: Evaluation of cache size-fitting.**

virtual vector size providing the best performance for our vectorized aggregation on the GPU. In this section, we present evaluation results comparing the vectorized aggregation with the native CUDA aggregation. For that, we generated various columns—sequences of 64-bit integer values—with increasing sizes from 2 MiB to 8 GiB. For the vectorized aggregation, we applied the best performing vector size of 2 MiB in all experiments. For the native CUDA aggregation, we used the function `cub::DevideReduce` from the CUDA SDK as already done for our *horizontal aggregation*.

The results are depicted in Figure 7. The throughput of the native CUDA aggregation is slightly higher as for our vectorized aggregation. This is especially true for large columns, which is not particularly surprising. Nevertheless, the result is promising to specialize our hardware-oblivious vectorized query operators to GPUs and to obtain a reasonably good performance. We hope to increase the throughput of our vectorized execution with additional GPU-specific optimization techniques as discussed in Section 4.

## 3.3 Validation

We also executed all our experiments on a second NVIDIA GPU namely an NVIDIA GTX 1070 Ti. This GPU provides a global main memory size of 8 GiB, a memory bandwidth of 256 GB/s, an L2 cache of 2 MB size, and 19 SMs. Each SM has 128 cores resulting in a total of 2,432 cores. In general, we observed a behavior similar as with the NVIDIA Quadro RTX 8000. In particular, we executed the cache-fitting experiment to determine the best virtual vector size for the aggregation operator. In contrast to the RTX 8000 GPU, we obtain the best performance when our vector size is a fourth of the L2 cache (512 KiB) size as shown in Figure 8.

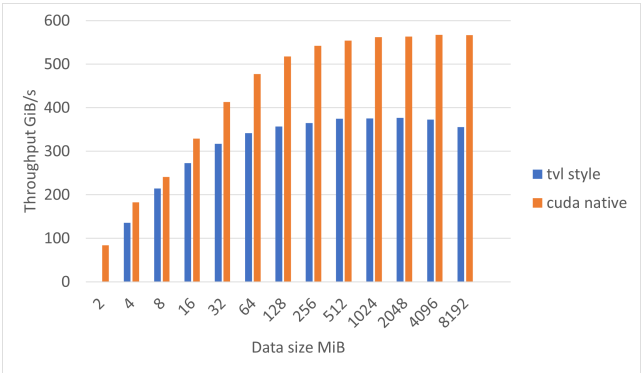



**Figure 7: Comparing vectorized versus CUDA native aggregation.**

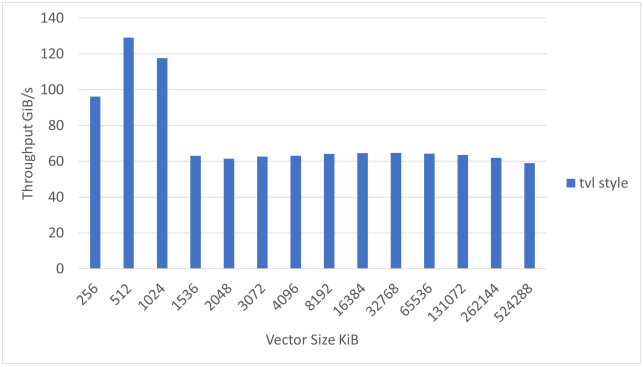



**Figure 8: Validation on NVIDIA RTX 1070 Ti.**

# 4. FUTURE WORK

Based on the previous section, we conclude that SIMD processing using a virtual vector model is generally possible on a GPU. Choosing the right vector size and configuration is critical to achieve a good performance. Sub-optimal configurations reduce the performance by more than one order of magnitude. Overall, we obtain good performance which is slower than native CUDA implementations. For the aggregation operator using the ideal vector size, our approach is 37.4% slower in the worst case for 8 GiB data size, and 10.8% slower in the best case for 8 MiB data size than a native CUDA approach. By validating our approach on a different GPU, we have also shown that our conclusions are more generally applicable beyond a single GPU model. Moreover, our hardware-oblivious query operators can be specialized to SIMD extensions as well as to GPUs in a unified way.

To summarize, our results are promising and our ongoing research in that direction will focus on the following aspects:

**(1) Virtual Vector Size:** To extend our work, we are looking forward to evaluate other query operators. We seek to explore if optimal vector sizes and configurations depend on the query operators.

**(2) Implementation:** Completing the GPU TVL requires to implement all primitives. To optimize the hardware-conscious implementations, we want to explore the usage of shared memory, registers, and persistent caching.

**(3) Optimization:** Besides an optimal hardware-conscious implementation for GPUs, we want to investigate more mapping strategies for a broader optimization. Currently, the TVL has a 1:1 mapping of hardware-oblivious primitives to hardware-conscious implementations. As shown in our experiments, the CUDA native aggregation outperforms the vectorized approach. To improve the performance of our vectorized approach, an idiom-based mapping strategy could be helpful. By replacing a vectorized computation by a semantically equivalent but more performant code at query compile-time, a speedup can possibly be achieved. For that, it is necessary to identify often used idioms that lose performance by vectorization and replace them by idiomatic implementations.

# 5. RELATED WORK

To address the portability of code across heterogeneous computing units, OpenCL [18] and Intel's OneAPI [19]general purpose parallel programming language approaches. However, the genericity is a major drawback from a perfor-

mance point of view. In contrast, Pirk et al. [20] presented a more database-specific approach called Voodoo to execute single-source query operators on different (co-)processors. Voodoo is a declarative intermediate algebra that abstracts the detailed architectural properties of the hardware, without losing the ability to generate highly tuned code. The proposed algebra consists of a collection of declarative and vector-oriented operations. Operators described in this algebra are compiled to OpenCL. The drawbacks of this approach are: (i) operators have to be described with a new algebra, (ii) a specialized compiler is required, and (iii) the overhead of OpenCL. Another abstraction concept was proposed by Heimel et al. [21]. They developed 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. However, the approach is mainly based on OpenCL and they do not support SIMD on CPUs.

In a recent work, Shanbhag et al. [15] introduced a new processing model for an efficient query processing on GPUs called *tile-based execution model*. This processing model extends the SIMD-based processing on CPUs where each thread processes a vector at a time to GPU. Based on that SIMD extension concept, they introduced a CUDA-like library called *Crystal* consisting of data processing primitives that can be composed in order to implement queries on the GPU. Thus, this approach has a lot in common with our idea, but they are limited to GPUs. It would be interesting to implement our *TVL* hardware-conscious plug-in for GPUs using the *Crystal* library.

# 6. SUMMARY

In this paper, we evaluated the integration of GPUs in our SIMD abstraction layer *TVL* by interpreting GPUs as virtual vector engines. By conducting a number of experiments, we have shown that our approach is promising. While the observed throughput does not outperform CUDA native implementations, it allows developers to use CUDA-based fast GPU primitives without requiring knowledge of GPU implementations. Our vectorized approach still achieves reasonable performance that is not an order of magnitude slower than native CUDA implementations. However, tuning the primitives and operators by choosing the right virtual vector size and configuration is critical for achieving good performance.

## Acknowledgments

This work was partly funded by the German Research Foundation (DFG) within the RTG 1907 (RoSI).

# 7. REFERENCES

[1] 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.

[2] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner, "Pipelined query processing in coprocessor environments," in *SIGMOD*, 2018, pp. 1603–1618.

[3] T. Karnagel, D. Habich, and W. Lehner, "Adaptive work placement for query processing on heterogeneous computing resources," *Proc. VLDB Endow.*, vol. 10, no. 7, pp. 733–744, 2017.

[4] Y. Yuan, R. Lee, and X. Zhang, "The yin and yang of processing data warehousing queries on GPU devices," *Proc. VLDB Endow.*, vol. 6, no. 10, pp. 817–828, 2013.

[5] 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.

[6] 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.

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

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

[9] M. Xue, Q. Xing, C. Feng, F. Yu, and Z. Ma, "Fpga-accelerated hash join operation for relational databases," *IEEE Trans. Circuits Syst. II Express Briefs*, vol. 67-II, no. 10, pp. 1919–1923, 2020.

[10] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh, "Improving main memory hash joins on intel xeon phi processors: An experimental approach," *Proc. VLDB Endow.*, vol. 8, no. 6, pp. 642–653, 2015.

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

[13] 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.

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

[15] A. Shanbhag, S. Madden, and X. Yu, "A study of the fundamental performance characteristics of gpus and cpus for database analytics," in *SIGMOD*, 2020, pp. 1617–1632.

[16] Turing Tuning Guide: CUDA Toolkit documentation, https://docs.nvidia.com/cuda/turing-tuning-guide/index.html.

[17] CUDA C Programming Guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[18] OpenCL, https://www.khronos.org/opencl/.

[19] OneAPI, https://www.oneapi.com.

[20] H. Pirk, O. R. Moll, M. Zaharia, and S. Madden, "Voodoo - A vector algebra for portable database performance on modern hardware," *Proc. VLDB Endow.*, vol. 9, no. 14, pp. 1707–1718, 2016.

[21] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl, "Hardware-oblivious parallelism for in-memory column-stores," *Proc. VLDB Endow.*, vol. 6, no. 9, pp. 709–720, 2013.