Exploiting automatic vectorization to employ SPMD on SIMD registers

Stefan Sprenger
Department of Computer Science
Humboldt-Universität zu Berlin
Berlin, Germany
sprengsz@informatik.hu-berlin.de

Steffen Zeuch
Intelligent Analytics for Massive Data
German Research Center for Artificial Intelligence
Berlin, Germany
steffen.zeuch@dfki.de

Ulf Leser
Department of Computer Science
Humboldt-Universität zu Berlin
Berlin, Germany
leser@informatik.hu-berlin.de

***Abstract*—Over the last years, vectorized instructions have been successfully applied to accelerate database algorithms. However, these instructions are typically only available as intrinsics and specialized for a particular hardware architecture or CPU model. As a result, today's database systems require a manual tailoring of database algorithms to the underlying CPU architecture to fully utilize all vectorization capabilities. In practice, this leads to hard-to-maintain code, which cannot be deployed on arbitrary hardware platforms. In this paper, we utilize ispc as a novel compiler that employs the Single Program Multiple Data (SPMD) execution model, which is usually found on GPUs, on the SIMD lanes of modern CPUs. ispc enables database developers to exploit vectorization without requiring low-level details or hardware-specific knowledge. To enable ispc for database developers, we study whether ispc's SPMD-on-SIMD approach can compete with manually-tuned intrinsics code. To this end, we investigate the performance of a scalar, a SIMD-based, and a SPMD-based implementation of a column scan, a database operator widely used in main-memory database systems. Our experimental results reveal that, although the manually-tuned intrinsics code slightly outperforms the SPMD-based column scan, the performance differences are small. Hence, developers may benefit from the advantages of SIMD parallelism through ispc, while supporting arbitrary hardware architectures without hard-to-maintain code.**

I. Introduction

Vectorization allows to process multiple data items with one instruction. According to Flynn's taxonomy [9], this execution model is called Single Instruction Multiple Data (SIMD). Database developers can exploit data-level parallelism using SIMD instructions through programming libraries, typically available as intrinsics [3]. However, intrinsics depend on the available instruction set, which is specific to the underlying CPU architecture, and processed data type. This leads to poor developer productivity, hard-to-maintain programs, and hinders code deployments on arbitrary hardware architectures.

Over the last years, many approaches have shown that database systems benefit from vectorization. Researchers used SIMD instructions to accelerate column scans [20], index structures [10], [17], [21], and regular expression matching [16]. In the context of recent many-core CPUs such as Intel®'s Xeon Phi, modern CPUs achieve a similar degree of parallelism (DOP) as state-of-the-art GPUs when combining multi-threading with SIMD instructions[1]. For these reasons, vectorization is essential for the performance of database systems on modern CPU architectures.

Although modern compilers, like GCC [2], provide auto vectorization [1], typically the generated code is not as efficient as manually-written intrinsics code. Due to the strict dependencies of SIMD instructions on the underlying hardware, automatically transforming general scalar code into high-performing SIMD programs remains a (yet) unsolved challenge. To this end, all techniques for auto vectorization have focused on enhancing conventional C/C++ programs with SIMD instructions. However, in 2012, Intel® proposed a different approach, the Intel® SPMD Program Compiler (ispc) [14]. ispc enables developers to utilize the parallel capabilities of vectorization without requiring low-level intrinsics programming. Therefore, ispc uses the execution model of modern GPUs, Single Program Multiple Data (SPMD). ispc deploys the same program on multiple processing units, i. e., on different SIMD lanes of a CPU. Additionally, ispc exploits multi-threading to increase the DOP. Fortunately, ispc supports all modern CPU architectures and instruction set architectures[2], which allows to deploy programs on arbitrary hardware. Finally, ispc combines the convenience of CPU programming with the high parallelism of GPUs without requiring hardware-specific knowledge.

In this paper, we use ispc's SPMD-on-SIMD approach to accelerate database algorithms. In particular, we implement an operator that is widely used in main-memory database systems, the column scan. We examine if ispc can compete with manually-tuned intrinsics code. In a comprehensive evaluation, we compare a SPMD-based variant with a scalar and an intrinsics-based column scan. Our experimental results indicate that the performance of programs written in ispc are almost as good as manually-tuned intrinsics code. In contrast to intrinsics code, ispc code is not specific to the underlying architecture. Thus, employing the SPMD execution model on the SIMD registers of modern CPUs is a viable alternative to vectorize database algorithms.

Stefan Sprenger is funded by the Deutsche Forschungsgemeinschaft through graduate school SOAMED (GRK 1651).

[1]For instance, the Intel® Xeon Phi 7290 CPU features 72 cores and 512-bit wide SIMD registers offering a theoretical DOP of $72 * 16 = 1152$, when processing integers, because one SIMD register can hold 16 32-bit integer values.

[2]https://ispc.github.io/ispc.html#selecting-the-compilation-target

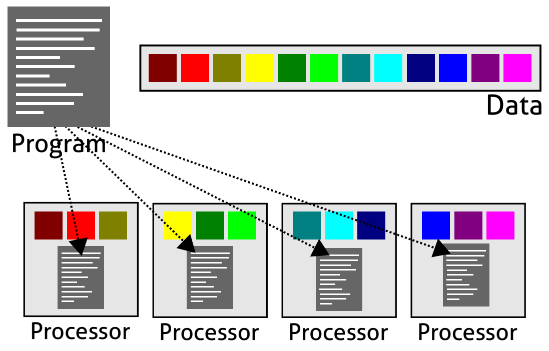

Fig. 1. The SPMD execution model.

II. Background and Foundations

A. Single Program Multiple Data (SPMD)

SPMD is an execution model for parallel programs, where a sequential, non-parallel program is deployed onto multiple processing units. In particular, each instance of the program is executed concurrently on different data. This model was proposed in 1988 by Darema et al. [7] as an extension to Flynn's taxonomy [9] and has been applied to various research areas for parallel programming [13], [18]. Today, SPMD is mainly found in parallel graphics computing [13].

Figure 1 illustrates the parallel execution of a sequential program on four processing units, where each program instance processes a different subset of the data. In SPMD, developers write sequential code that is automatically parallelized by initializing multiple instances of the program. This is in contrast to other execution models, like SIMD, which require developers to explicitly specify parallelization in the program code.

Overall, SPMD leads to a high developer productivity, easy-to-maintain code, and reduced code complexity.

B. Intel SPMD Program Compiler (ispc)

The Intel® SPMD Program Compiler (ispc) [14] is an open-source compiler, which translates sequential C/C++ code into a highly parallel program that is executed on the SIMD registers of modern CPUs. ispc code is mostly compliant with standard C/C++, but introduces few keywords and concepts that are used for automatic vectorization, e. g., parallel for loops. ispc supports different instruction set architectures, e. g., SSE2, SSE4, AVX, AVX2, and AVX512, and many modern CPU architectures, e. g., Haswell, Sandy Bridge, Ivy Bridge, and Nehalem. In addition to exploiting SIMD registers, ispc can also deploy programs onto multiple CPU cores, which enables an even higher DOP.

Functions written in ispc can be directly called from C/C++, thus ispc can be used to extend existing code without introducing fundamental changes. The ispc compiler creates object files that can be included in conventional C/C++ programs. Listing 1 shows an ispc function named *square_arr* that squares all elements of an *input* array of size *count* and stores the results in an *output* array.

Listing 1. Exemplary ispc function that can be directly called from conventional C/C++ code.

```
1  export void square_arr(uniform float input[],
2                         uniform float output[],
3                         uniform int count) {
4    foreach (i = 0 ... count) {
5      output[i] = input[i] * input[i];
6    }
7  }
```

The keyword *export* is used to mark a function that can be called from external C/C++ code. In this case, *square_arr* can be called like a regular C/C++ function. In ispc, *uniform* is used to mark parameters and variables that have identical values for all concurrently executed instances of the function. Using the parallel *foreach* looping construct, we can concurrently iterate over *count* values. The degree of parallelism is determined by the width of the SIMD registers and the size of the used data types. For instance, given 256-bit SIMD registers are available, eight iterations of the *foreach* loop (see Lines 4-6 of Listing 1) can be executed concurrently, because 32-bit floating-point values are used as data type.

C. Related Work

Previous work related to this paper can be divided into two groups: (1) approaches exploiting SIMD instructions to accelerate database algorithms, and (2) modern implementations of the column scan.

Zhou and Ross [22] proposed early approaches that applied SIMD instructions to database systems. They vectorized scan operations and index structures. FAST [10], ART [11], and CSSL [17] are examples for index structures that exploit SIMD instructions to speedup index search. Recently, Polychroniou et al. [15] presented advanced techniques for vectorizing algorithms of a database system, e. g., column scans, hash tables, or partitioning. They use modern extra-wide SIMD registers and operations found in recent instruction sets, e. g., scatter, and gather. All these approaches, which exploit SIMD instructions to accelerate database algorithms, rely on explicit intrinsics. In contrast, we focus on automatic vectorization and employ the SPMD execution model on SIMD registers.

Broneske et al. [6] study different techniques to accelerate column scans on modern hardware, e. g., elimination of branches, vectorized instructions, multi-threading, or loop unrolling. Willhalm et al. [20] propose a table scan variant, the SIMD-Scan, which uses vectorized instructions to iterate over compressed data of an in-memory column store. They also provide a framework to exploit SIMD instructions in column scans with complex predicates [19]. SAP HANA employs the SIMD-Scan algorithm [8]. Finally, BitWeaving [12] represents an advanced approach to scanning a database column. It processes multiple bits of database columns in a single CPU cycle by exploiting the full word width of modern CPU instructions, which leads to intra-cycle parallelism. In contrast, we contribute a portable column scan implementation that exploits vectorized instructions without requiring complex code constructs, like SIMD intrinsics.

III. SPMD-based Column Scan

In this section, we present four flavors of a SPMD-based column scan (SBCS):

1) **B-ST-SBCS**: *Branching, single-threaded* implementation. This flavor solely relies on automatic vectorization.
2) **BF-ST-SBCS**: *Branch-free, single-threaded* implementation.
3) **B-MT-SBCS**: *Branching, multi-threaded* implementation. This flavor exploits multiple threads in addition to vectorized instructions to further the DOP.
4) **BF-MT-SBCS**: *Branch-free, multi-threaded* implementation.

A. The Column Scan

In this paper, we implement a column scan that selects all keys from a column that satisfy a given search predicate. In particular, we investigate range queries which are defined by two boundaries. As a result, we obtain a bitmask that indicates which keys of the column match the search predicate.

Algorithm 1 presents a basic variant of the column scan. The function $SCAN$ requires the following five input parameters. The integer value *count* equals to the size of the to-be-scanned column. The array *column* holds all values of the to-be-scanned column in a possibly unsorted order. The array *result*, which has the same size as *column*, is used to store the scan results. The value *lower* defines the lower boundary of the search query, and the value *upper* defines the upper boundary of the search query.

Algorithm 1 The Column Scan

```
function SCAN(count, column, result, lower, upper)
    for (i = 0; i < count; ++i) do
        if (column[i] $\geq$ lower && column[i] $\leq$ upper) then
            result[i] = 1;
        else
            result[i] = 0;
```

Using a *for* loop, the algorithm iterates over all elements of *column* and determines for each element if it matches the search predicate defined by *lower* and *upper*. The results are stored in an array called *result*. For each value in the column, the result array holds the value 1 at position i if the i'th element of the column matches the query, i. e., $lower \leq column[i] \leq upper$, and 0 if it does not match.

B. SPMD-based Implementation

The implementation of the SPMD-based column scan is very similar to conventional C/C++ code and uses only few constructs introduced by ispc, e. g., $uniform$, and $export$ (see Section II-B for a description).

We provide a branching and a branch-free variant of SBCS to investigate the impact of branch mispredictions on the performance of queries with different selectivity.

Listing 2. Branching variant of the SPMD-based column scan.

```
 1  export void sbcs(uniform KEY_TYPE column[],
 2                   uniform KEY_TYPE result[],
 3                   uniform KEY_TYPE lower,
 4                   uniform KEY_TYPE upper,
 5                   uniform int count) {
 6    foreach (index = 0 ... count) {
 7      KEY_TYPE val = column[index];
 8      if (lower <= val && upper >= val)
 9        result[index] = 1;
10      else
11        result[index] = 0;
12    }
13  }
```

Listing 3. Branch-free variant of the SPMD-based column scan.

```
 1  export void sbcs_bf(uniform KEY_TYPE column[],
 2                      uniform KEY_TYPE result[],
 3                      uniform KEY_TYPE lower,
 4                      uniform KEY_TYPE upper,
 5                      uniform int count) {
 6    foreach (index = 0 ... count) {
 7      KEY_TYPE val = column[index];
 8      result[index] = (lower <= val &&
 9                       upper >= val);
10    }
11  }
```

Listing 2 presents the branching variant of SBCS, B-ST-SBCS. The function *sbcs* (see Lines 18-30) is marked with *export* and can thus be called from C/C++.

The *foreach* loop (see Lines 6-12) iterates over all column data and compares each value with the search predicate defined by *lower* and *upper*. It is automatically vectorized by ispc; hence multiple column values are processed in parallel.

Our implementation is able to work with different data types, e. g., integers, or floating-point values, because *KEY_TYPE* is defined as a macro.

For example, given that *KEY_TYPE* is set to 32-bit integers and the underlying CPU provides 256-bit wide SIMD registers, eight instances of the *foreach* loop are executed in parallel.

Listing 3 presents the branch-free implementation of SBCS, BF-ST-SBCS. Following the approach of Zhou and Ross [22], this implementation omits branches and stores the result of the predicate evaluation into the result array (see Lines 8-9). We also evaluate multi-threaded implementations of SBCS, B-MT-SBCS, and BF-MT-SBCS. Therefore, we use ispc's built-in task parallelism[3] to launch multiple threads. Enabling multi-threading does not introduce any major changes to the algorithms, but furthers the DOP.

IV. Evaluation

The main objective of our evaluation is to investigate whether ispc's SPMD-on-SIMD approach can compete with manually-tuned intrinsics code. To this end, we compare four flavors of a SPMD-based column scan (single-threaded, multi-threaded, branching, branch-free) with different implementations of a non-vectorized and an intrinsics-based column scan.

[3]https://ispc.github.io/ispc.html#task-parallel-execution

A. Experimental Setup

1) Hardware: We executed all experiments on a server equipped with two Intel® Xeon E5-2620 CPUs (2 GHz clock rate), each featuring six cores, 12 virtual cores through hyper-threading, and 16 256-bit wide SIMD registers (AVX [4]). The server features 32 GB main memory.

2) Software: The code was compiled with GCC 4.8.4 and ispc 1.9.1. For GCC, we use the optimization flag *-O3*. When evaluating non-vectorized code, we disable GCC's auto vectorization by using the compiler flag *-fno-tree-vectorize*. Additionally, we enable the following options for the ispc compiler: *-O3* (optimization level), *--arch=x86-64* (target architecture), and *--target=avx* (target instruction set architecture). We measured CPU performance counters with PAPI [5].

3) Methodology: Our experiments measure speedup or throughput as an indicator for performance. Throughput is measured in GB/s and defines the amount of data each approach can process/scan per second. In contrast, speedup (as a factor) is used to compare the throughput of two variants w.r.t. a baseline approach. For instance, we may use throughput when comparing the scan performance of a vectorized and a non-vectorized column scan. In contrast, we may use speedup when comparing the performance of two different multi-threaded column scan implementations w.r.t. a single-threaded implementation.

4) Experimental Data: For all experiments, we use 1 GB synthetically generated keys in random order. Given that n defines the number of keys, we generate queries by using a randomly selected key as lower boundary and add $n * selectivity$ to determine the upper boundary. Before each experiment, we conduct one warming run and start actual measurements afterwards (hot caches). All experiments are executed three times and we report the average value.

B. Experiments

In our evaluation, we investigate the speedup of single-threaded SBCS over sequential non-vectorized scans depending on selectivity (see Section IV-B1), key sizes (see Section IV-B2), and data types (see Section IV-B3). In Section IV-B4, we analyze the scalability of multi-threaded SBCS depending on the number of used CPU threads. Finally, Section IV-B5 compares single-threaded SBCS with single-threaded, manually-written intrinsics code.

1) Query Selectivity: In this experiment, we investigate the speedup that ispc achieves over a scalar execution of the identical code on a conventional server CPU. We considered a branch-free as well as a branching implementation of the column scan when scanning over 1GB of 32-bit integer keys ($n = 268,435,456$) to evaluate selection queries of varying selectivity. We run ispc in single-threaded mode to determine the pure speedup of employing the SPMD execution model on the SIMD lanes of the CPU. Note that we present multi-threading experiments in Section IV-B4.

We investigated query selectivities ranging from 0% (low selectivity) to 100% (high selectivity). While the speedup for the branch-free variant is rather constant, speedup for the branching variant highly depends on query selectivity (see Figure 2). On average, the branch-free variant of the SBCS achieves a speedup factor of 1.5, while the branching variant achieves an average speedup factor of 3.8.

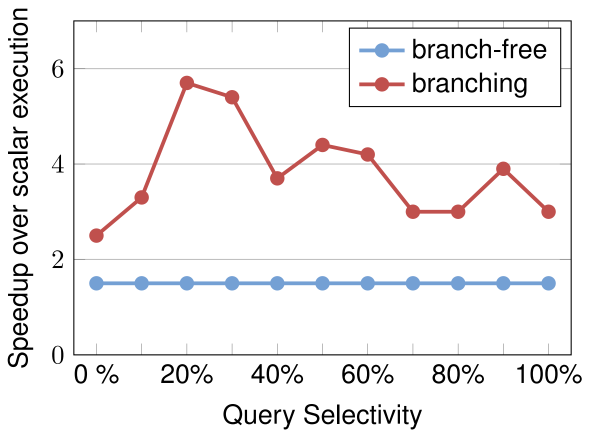

Fig. 2. Speedup over scalar execution of the column scan depending on query selectivity (data type = 32-bit unsigned integer).

Perf. Counter	scalar	scalar (branch-free)	SBCS	SBCS (branch-free)
Instructions	2,004M	2,953M	892M	738M
Branch Mispr.	93,917k	1	15,647k	2
LLC Misses	1,698k	2,427k	2,168k	3,178k
TLB Misses	3,634	2,197	4,737	3,067

Fig. 3. Performance counters per scan on 1GB 32-bit unsigned integer keys (10 % query selectivity).

Figure 3 shows selected performance counters per selection query with a selectivity of 10 %. The measured performance counters reveal that the performance gains of SBCS over the scalar scan are achieved by a tremendous reduction of executed instructions. By exploiting SIMD registers for query evaluation, SBCS needs to conduct less comparisons. This also results in six times less branch mispredictions for the branching variant. In contrast, the branch-free variants produce no branch mispredictions by design (the measured branch mispredictions are caused by our experimental setup that executes batches of queries using a *for* loop to determine the average throughput).

2) Key Sizes: ispc employs SPMD on multiple SIMD lanes. As a result, the performance benefits depend on the used key size. In the following experiment, we analyze the speedup of SBCS over a scalar implementation on four different key sizes: 8 bits, 16 bits, 32 bits, and 64 bits. We use a query selectivity of 10% and unsigned integer values.

Figure 4 shows the results. Both variants of the SBCS, branch-free and branching, show a similar speedup w.r.t. different key sizes. For 16-bit keys, both variants achieve the highest speedup (branch-free: 3.95X, branching: 6.24X). In general, the speedup increases when key size decreases, except for very small keys, like 8-bit keys. As noted in the ispc performance guide [4], code generated for small integer types is usually less efficient.

[4]https://ispc.github.io/perfguide.html

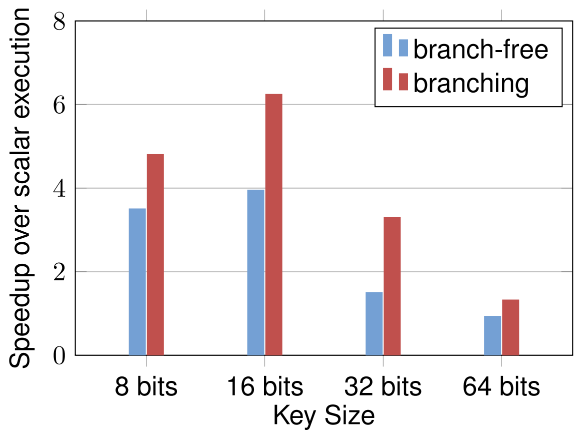

Fig. 4. Speedup over scalar execution of the column scan depending on key size (selectivity = 10%).

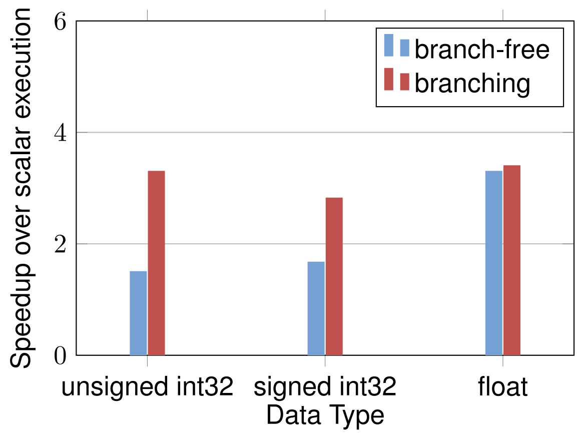

Fig. 5. Speedup over scalar execution of the column scan depending on data type (selectivity = 10%).

3) Data Types: In our evaluation, we use AVX [4] as target instruction set architecture. AVX, opposed to AVX2, supports only operations on floats and doubles but not integers. Hence, integer values need to be converted to floating-point values, which is done at compile time and should not impact performance according to [3].

In this experiment, we analyze the speedup of SBCS over scalar code depending on the used data type. We consider three different data types of the same size (32 bits): unsigned integers, signed integers, and floats. As shown in Figure 5, SBCS achieves the highest speedup for floats (branching: 3.4X, branch-free: 3.3X). For unsigned integers, branching SBCS achieves a slightly higher speedup than the branch-free variant (3.3X vs. 2.6X), whereas for signed integers, branch-free SBCS achieves a slightly higher speedup than the branching variant (1.7X vs. 1.5X).

Overall, the branching implementation of SBCS induces no large performance gaps between the considered data types given that they have the same size. In contrast, the branch-free implementation achieves the highest speedup for floating-point values.

4) Deployment on Multiple Cores: In addition to SPMD parallelism, which solely exploits SIMD lanes, ispc provides task parallelism at the granularity of CPU threads. Figure 6 shows the throughput of the multi-threaded SBCS variants depending on the number of used software threads. Note that the dotted vertical line indicates the number of available physical cores (12). We used 32-bit unsigned integer keys and executed selection queries with a selectivity of 10 %.

Branching as well as branch-free SBCS implementations benefit from multi-threading up to a particular barrier, which is established by the number of physical cores. As shown, hyper-threading does not provide additional performance improvements, as multiple hyper-threads share the same SIMD registers on a certain CPU core. The throughput of SBCS scales almost linearly with the number of threads until all physical cores are used (approx. 12 used threads). Beyond this point, only minor performance improvements can be achieved. In particular, the branch-free variant achieves a throughput of 14.8 GB/s (branching 13.8 GB/s) using 12 threads, and a throughput of 15.7 GB/s (branching 14.1 GB/s) using 24 threads.

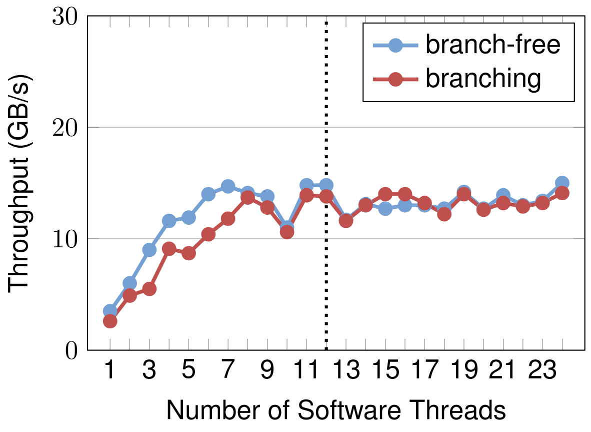

Fig. 6. Throughput of the SPMD-based column scan depending on the number of used software threads (data type = 32-bit unsigned integer, selectivity = 10%). The vertical dotted line indicates the number of physical cores.

By combining multi-threaded execution with employing SPMD on the SIMD lanes, ispc enables the efficient exploitation of the enormous computing power of modern CPUs. Taking recent many-core CPUs, like Intel®'s Xeon Phi family, into account, ispc achieves DOP similar to modern GPUs.

5) ispc vs. intrinsics: In this experiment, we compare the performance of single-threaded SBCS with single-threaded intrinsics-based column scans. Intrinsics are provided by a dedicated library that supports operations such as comparing, shuffling, or computations. We manually implemented a vectorized column scan that makes use of intrinsics code. We highlight the performance differences between ispc-based SBCS and the intrinsic scans. In particular, ispc automatically vectorizes code, whereas intrinsics require developers to manually tune their code for using SIMD parallelism. We also provide the throughput of a scalar, non-vectorized version of the column scan, which does not make use of SIMD.

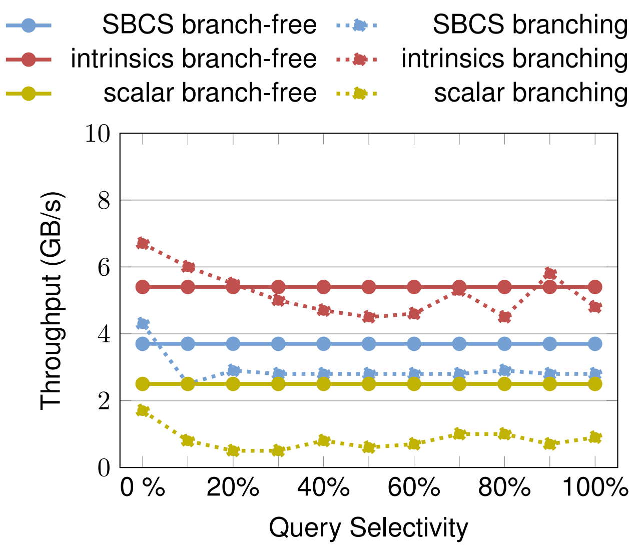

Fig. 7. Throughput of SPMD-based column scan, intrinsics-based column scan, and scalar column scan using single-threading depending on query selectivity (data type = 32-bit unsigned integer).

Figure 7 shows the throughput of the approaches when executing selection queries of varying selectivity. As in previous experiments, we execute randomly generated range queries on a data set of 32-bit unsigned integers.

For both implementation variants, branching and branch-free, the intrinsics code achieves the highest throughput. On average, the branch-free intrinsics-based scan is 1.5X faster than SBCS (1.8X for branching). This speedup can be explained by the fact that the intrinsics-based scan was carefully tuned to the underlying hardware, while SBCS was automatically vectorized by the ispc compiler.

Although ispc promises to generate highly efficient SIMD code, it does not (yet) achieve the same performance as intrinsics code that was carefully implemented by experienced developers. Nonetheless, ispc's SPMD-on-SIMD approach represents an interesting approach to use SIMD parallelism without low-level programming or hardware-specific intrinsics code.

V. Conclusions and Future Work

In this paper, we presented a SPMD-based column scan and analyzed it in a comprehensive evaluation. Our goal was to investigate whether ispc's SPMD-on-SIMD approach can compete with intrinsics code manually-written by experienced developers. We showed that SBCS is almost as fast as manually-written intrinsics code, although it is not specific to the underlying CPU architecture or used data type. Also, it does not require low-level programming constructs. As a result, we identify a trade-off between the maintainability/developer productivity and performance of ispc vs. intrinsics code. Considering the rise of many-core CPUs with extra-wide SIMD registers, our evaluation results indicate that ispc's SPMD-on-SIMD approach will become even more valuable in the next years. ispc allows to combine automatic vectorization with multi-threading and hence achieves DOP similar to modern highly-parallel GPUs. Furthermore, Intel® will strive continuously to improve the code generation of ispc. In future work, we plan to run experiments on massively parallel systems to compare the performance of the SPMD-on-SIMD approach when deployed on many-core CPUs, like Intel®'s Xeon Phi, with highly-parallel GPUs. We also intend to investigate other database operations beyond column scans, e. g., joins, hash tables, or bloom filters.

References

[1] Auto-vectorization in GCC. https://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[2] GCC, the GNU Compiler Collection. https://gcc.gnu.org/.

[3] Intel®Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide.

[4] Introduction to Intel®Advanced Vector Extensions. https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions.

[5] PAPI. http://icl.cs.utk.edu/papi/.

[6] D. Broneske, S. Breß, and G. Saake. Database Scan Variants on Modern CPUs: A Performance Study. In *Proc. of the 2nd Int. Workshop on In Memory Data Management and Analytics*, pages 1–15, 2014.

[7] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, 1988.

[8] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[9] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Computers*, 21(9):948–960, 1972.

[10] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proc. of the ACM Int. Conf. on Management of Data*, pages 339–350, 2010.

[11] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *29th IEEE Int. Conf. on Data Engineering*, pages 38–49, 2013.

[12] Y. Li and J. M. Patel. BitWeaving: fast scans for main memory data processing. In *Proc. of the ACM Int. Conf. on Management of Data*, pages 289–300, 2013.

[13] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[14] M. Pharr and W. R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing*, pages 1–13. IEEE, 2012.

[15] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. In *Proc. of the ACM Int. Conf. on Management of Data*, pages 1493–1508, 2015.

[16] E. A. Sitaridi, O. Polychroniou, and K. A. Ross. SIMD-accelerated regular expression matching. In *Proc. of the 12th Int. Workshop on Data Management on New Hardware*, pages 8:1–8:7, 2016.

[17] S. Sprenger, S. Zeuch, and U. Leser. Cache-Sensitive Skip List: Efficient Range Queries on Modern CPUs. In *ADMS/IMDM@VLDB*, 2016.

[18] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W. W. Hwu. Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs. In *Proc. of the 8th Int. Symposium on Code Generation and Optimization*, pages 111–119, 2010.

[19] T. Willhalm, I. Oukid, I. Müller, and F. Faerber. Vectorizing Database Column Scans with Complex Predicates. In *Int. Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, pages 1–12, 2013.

[20] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB*, 2(1):385–394, 2009.

[21] S. Zeuch, J. Freytag, and F. Huber. Adapting Tree Structures for Processing with SIMD Instructions. In *Proc. of the 17th Int. Conf. on Extending Database Technology*, pages 97–108, 2014.

[22] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proc. of the ACM Int. Conf. on Management of Data*, pages 145–156, 2002.