Co-Utilizing SIMD and Scalar to Accelerate the Data Analytics Workloads

Zewen Sun†, Zhifang Li†, Chuliang Weng*
School of Data Science and Engineering
East China Normal University, Shanghai, China
{zewensun, zhifangli}@stu.ecnu.edu.cn, clweng@dase.ecnu.edu.cn

2023 IEEE 39th International Conference on Data Engineering (ICDE) | 979-8-3503-2227-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDE55515.2023.00387

***Abstract*—The increasing capacity and reducing cost of the main memory made in-memory data analytics systems widely deployed as they could provide higher throughput and lower latency. Since the data resides in memory, computational throughput becomes a crucial factor in the performance of these systems rather than disk accesses. Single instruction multiple data (SIMD) is an effective mechanism to improve computational performance, which has been well studied to accelerate data analytics systems. However, the state-of-the-art methods focus on using SIMD more efficiently while neglecting scalar execution units.**

In this paper, we present the hybrid execution framework (HEF) to co-utilize SIMD and scalar execution units for the data analytics workload. We also extend the concept of *pack* to eliminate the data dependency between adjacent instructions, achieving shorter instruction execution intervals. Experimental results show that the hybrid execution achieves up to 2.38× and 1.45× better performance compared with the purely scalar and SIMD implementation on the star schema benchmark (SSB) queries, respectively. Besides, HEF performs better than the state-of-the-art system Voila for a majority of queries in SSB under all data scales.

***Index Terms*—hybrid execution, SIMD, microarchitecture, data analytics**

I. Introduction

As memory has become cheaper and its capacity has grown to the terabyte level, we could enable data analytics systems to reside in memory. In-memory data analytics systems exploit this trend to satisfy the ever-increasing performance demands. By eliminating costly disk I/Os in traditional data analytics systems, disk accesses are no longer the bottleneck of in-memory data analytics systems. Instead, computational throughput is a dominating factor in the performance of memory-resident data analytics systems. SIMD is a ubiquitous instruction set of current mainstream processors, achieving higher computational performance via data parallelism.

State-of-the-art approaches to accelerating data analytics systems with SIMD concentrate on using more SIMD instructions and utilizing them more efficiently, which has been demonstrated to be an effective way. The most popular method is leveraging SIMD to optimize stand-alone operators in data analytics systems, including scan [46] [24] [2] [16], and join [30] [11] [20]. Besides, there are some other research efforts [26] [33] [14] exploiting SIMD to implement vectorized execution engine instead of individual operators.

It is noted that existing solutions do not get the utmost out of execution units in a processor, since they overlook the scalar execution units. In data analytics systems, the data are primarily integer rather than floating-point, due to the fact that users often convert floating-point numbers into integers when the precision is sufficient for the application [9]. Meanwhile, it is observed that for the mainstream processor, SIMD shares arithmetic logic units (ALU) and registers with the floating-point unit (FPU), but it has independent integer scalar ALUs and registers. In response, we leverage SIMD and scalar execution units simultaneously to improve the throughput and reduce the response time of the data analytics systems. As a result, the hybrid execution of SIMD and scalar instructions could exploit both SIMD and scalar execution units in parallel.

However, there are some challenges in applying the hybrid execution of SIMD and scalar instructions to data analytics systems. First, the vectorized execution implemented with SIMD cannot support the same properties as the scalar implementation, such as register-resident execution while maintaining data parallelism [33]. Next, different microarchitectures contain different amounts of decoders and schedulers, micro-operations (uops) issue ports, ALUs, and various scheduling strategies. Even the different processors based on the same microarchitecture may have a different number of execution units. For example, Intel®Xeon®Silver series processors have only one AVX-512 execution unit, but Intel®Xeon®Gold or higher specifications have two AVX-512 execution units, even though both of them are based on Skylake [6]. Thus, a fixed combination of the SIMD and scalar statements is not reasonable. Further, data analytics systems contain various operators, which are implemented with disparate instructions and have distinguishing characteristics. As a result, we have to determine the optimal combination of SIMD and scalar statements for an operator.

To solve these issues, we present the hybrid execution framework (HEF) to co-utilize SIMD and scalar units simultaneously, achieving higher throughput and lower latency for the data analytics workload. We introduce HEF as a plug-in component, rather than building a new system or rewriting the current systems, and then it could be deployed along with existing systems at a low cost. Specifically, to unify the implementation of SIMD and scalar, we propose the hybrid intermediate description, which is an intermediate representation of SIMD and scalar statements, and used similarly as intrinsic SIMD functions. Meanwhile, it is hardware platform independent, since instruction set architectures (ISAs) usually

2375-026X/23/$31.00 ©2023 IEEE
DOI 10.1109/ICDE55515.2023.00387

provide equivalent interface for the same operation. Though not explicitly evaluated, HEF could be applied to other ISAs with vector support, such as Neon [5] and RVV [44].

With HEF, the optimal combination of SIMD and scalar statements is related to system microarchitectures. A straightforward approach is to determine the values with the case-by-case method, however, it is impractical to find an accurate result directly with the information of processors due to the complexity of modern processors. Thus, we introduce a test-based approach to get the optimal implementation. If all possible implementations are implemented and tested, it will take a long time. Moreover, they also have to be repeated on platforms with different vector ISAs. To avoid the tedious development work, we introduce the translator as a component of HEF. It translates the hybrid intermediate description into specific code statements based on the description table, which stores the map of the hybrid intermediate description to SIMD and scalar statements. Besides, we employ a pruning search mechanism to reduce the translation and testing overhead. In light of the existing test results, it could reduce overhead by avoiding generating and testing unnecessary implementations. Meanwhile, we provide an initial input with a smaller search space by exploiting the available information compared to a random or fixed initial input.

In addition, we achieve shorter instruction execution intervals by changing the number and arrangement of code statements in the generated code. We observe that the latency of many SIMD and scalar instructions are significantly less than their throughput [7]. Instruction latency is the number of clock cycles required for the execution unit to complete all micro-operations that form the instruction [6]. The instruction throughput refers to the clock cycles required to wait before the issue ports are free to accept the same instruction again [6]. The execution interval between adjacent instructions becomes the instruction throughput rather than the instruction latency by packing multiple isomorphic SIMD and scalar instructions together. We treat the isomorphic code statements executed in parallel as a *pack*, which was proposed in superword level parallelism (SLP) [21] to achieve auto-vectorization, and then extend it to optimize operators in data analytics systems.

To summarize, we make the following contributions in this paper:

- We analyze state-of-the-art approaches to accelerating data analytics systems with SIMD instructions, and observe that they do not achieve peak performance due to the neglect of scalar execution units.
- We propose HEF, a hybrid execution framework, to co-utilize the SIMD and scalar execution units in parallel. Furthermore, we extend the concept of *pack* to reduce the execution intervals between adjacent instructions.
- We introduce a test-based approach for implementing HEF with a pruning search mechanism, and evaluate HEF under the SSB (Star Schema Benchmark) and synthetic benchmarks to show benefits over the traditional implementation.

II. Background

This section introduces the relevant background information that helps to understand our motivations.

A. Processor Microarchitecture

In general, mainstream processors are superscalar that each physical core could issue and execute multiple instructions per cycle. Each physical core consists of three parts, including the front-end, execution engine, and memory subsystem. Fig. 1 shows a simplified block diagram of a modern processor core. *Front-End* denotes the first part of the processor core which is responsible for fetching instructions and preparing them for the subsequent pipeline stages. Within this phase, cache lines are fetched from the memory subsystem, then parsed into instructions, and finally decoded into micro-operations by multiple decoders in parallel. Subsequently, the micro-operations are delivered to the out-of-order buffer of the execution engine. *Execution Engine* is the back-end of the processor core. At this stage, micro-operations are reordered in the out-of-order buffer, executing in an out-of-order manner. In this way, micro-operations can execute as soon as the input operands are ready, enabling the processors to maximize the utilization of execution units (e.g., arithmetical units and load/store units). Usually, there are several execution units and issue ports in the execution engine, allowing the parallel execution of multiple instructions per cycle. *Memory Subsystem* is in charge of the load and store operations, including the L1 data cache (L1D), the unified L2 cache, and the system on a chip (SoC) interconnect. The SoC interconnect is connected to the L2 cache, and is used to access the out-of-core cache and main memory when missing all level caches.

For instance, Intel Skylake [8], AMD Zen [40], and ARM Neoverse [28] are the most representative server processor microarchitectures. All processor cores based on these microarchitectures support fetching at least 16 bytes of data from the instruction cache per cycle. Furthermore, these cores provide more than 4-way decoders which could decode multiple instructions in parallel. Skylake and Neoverse have a unified scheduler for the integer scalar and vector micro-operations. On the contrary, Zen separates the scheduler into several parts. For Skylake, part of the issue ports are shared between vector and scalar micro-operations, while Zen and Neoverse have separate issue ports for vector and scalar

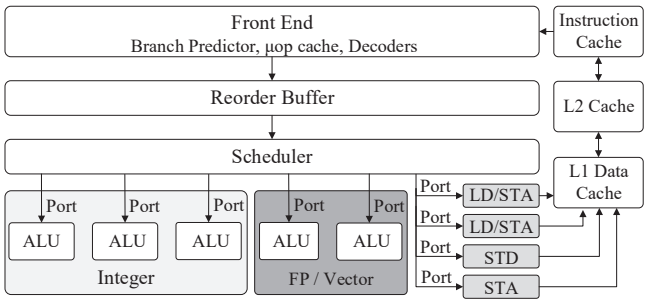

Fig. 1. Block diagram of a mainstream processor core.

micro-operations. It should be noted that for scalar pipelines, different operations have different numbers of execution units. For example, Skylake has four addition operation pipelines but only one multiplication pipeline.

B. Superword Level Parallelism

SLP [21] is fine-grained parallelism based on loop unrolling, basic block vectorization, and SIMD. It was proposed to exploit auto-vectorization for compilers. The original algorithm packs isomorphic scalar instructions that perform the same operation but with different data into a pack, and then compiles the scalar code within the pack to SIMD instructions. A *pack* is an n-tuple, $<s_1, s_2, ..., s_n>$, where $s_1, s_2, ..., s_n$ are independent isomorphic statements in a basic loop block. The workflow of SLP is presented in Fig. 2. In Fig. 2(c), all the $b_k = a[i+k]$ $(k = 0, 1, 2, 3)$ constitute a pack, the same as $c_k = 10 + k$ and $d_k = c_k + b_k$.

C. Motivation and Challenges

Data analytics systems mainly handle integer data instead of floating-point in the computation, which provides an opportunity to leverage the scalar and SIMD units simultaneously. However, the existing data analytics systems utilize either scalar or SIMD instead of leveraging them in parallel. Meanwhile, mainstream processors are superscalar processors equipped with multiple SIMD and scalar pipelines per physical core. Under data analytics workloads, part of the pipelines are idle due to data dependencies between adjacent code statements. Hybrid execution will benefit from the front-end and execution engine of the processor core, and we could leverage all the pipelines in parallel instead of one part of them by changing the number of isomorphic code statements within *pack*. Correspondingly, we get virtual vectorized instructions with wider lanes, which handles more elements at a time than SIMD.

With packing the isomorphic statements together to avoid data dependencies, we further achieve shorter execution intervals. Taking SIMD instruction *vpgatherqq* as an example, it is widely used in data analytics systems for assembling data from discontinuous addresses. Its latency is 26 cycles, and the throughput is only 5 cycles [7], and then the subsequent instruction consumes the data produced by *vpgatherqq* has to wait for 26 cycles due to the data dependency. During the period, the resources of the processor, including the instruction issue ports and ALUs, are idle. By executing multiple isomorphic SIMD and scalar statements, the data dependency between adjacent instructions is avoided, reducing the waiting time from instruction latency to instruction throughput, that is from 26 to 5 cycles. As a result, the fine-grained parallelism of instructions with long latency but short instruction throughput is achieved. Fig. 3 illustrates the execution of different implementations.

However, it seems to be a non-trivial challenge to benefit from the hybrid execution and the optimization of *pack*. The number of SIMD and scalar statements in the hybrid execution should be adapted to operator and platform to get optimal performance. It is necessary to find the applicable combination for diverse operators that execute on processors with different numbers of ALUs, which run in an out-of-order manner and have multiple execution pipelines, equipped with a complex data access system. Intuitively, we have to implement operators with all combinations of SIMD and scalar, and test all these implementations against different scenarios to obtain the optimal implementation. Assuming the number of vector and scalar statements ranges from 0 to v and 0 to s, respectively, and the value of *pack* ranges from 1 to p, then the search space size is expressed as:

$$space = \begin{cases} \sum_1^s 1,\ v = 0,\ s \neq 0 \\ \sum_1^v 1,\ s = 0,\ v \neq 0 \\ \sum_1^v \sum_1^s \sum_1^p 1 + v + s, v * s \neq 0 \end{cases} \quad (1)$$

Reducing them to one, we arrive at the following equation:

$$space = v * s * (p - 1) + v + s - 1,\ v + s \geq 1 \quad (2)$$

According to (2), its time complexity is $O(v * s * p)$, which will introduce tedious work to implement and seek various operators constructed with all possible combinations of SIMD and scalar statements. Although repetitive development work could be avoided by the code generation techniques, it still takes a lot of testing to determine the optimal implementation.

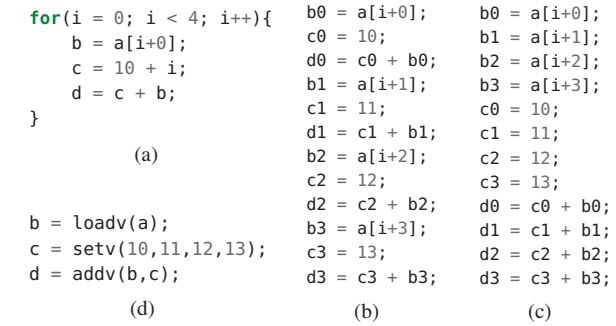

Fig. 2. The workflow of SLP. (a) The original code. (b) Loop unrolled. (c) Packing the isomorphic statements together. (d) Generated target SIMD instructions.

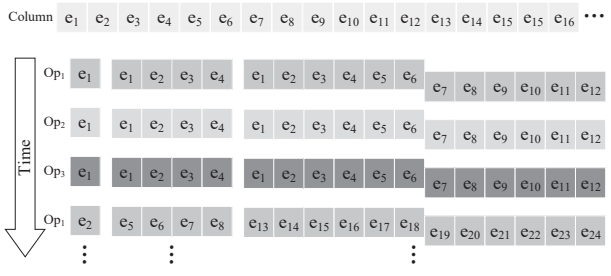

Fig. 3. The execution of purely scalar, SIMD with 4 lanes, and the implementation of hybrid execution description with a 4-lane SIMD and two scalar statements, and the size of *pack* is two.

III. Hybrid Execution Framework

To address these challenges, we propose and implement HEF. The core idea is to conceal the details behind the concepts of the hybrid intermediate description, that is, it shields the details of the concrete implementation and provides a unified interface to the upper layers. HEF translates the hybrid intermediate description to concrete implementations with SIMD and scalar statements, and finds the optimal implementation from a set of implementations.

A. Framework Architecture and Workflow

The architecture and procedure of HEF are shown in Fig. 4. It works in the offline phase to get the optimal implementation of hybrid execution operators, through a heuristic approach with the pruning method. In addition, HEF relies on the preprocessing phase, which contains manual implementation-dependent tasks to provide necessary input files. After getting the dependency files, we could get the optimal implementation of the target operators with HEF. Once we get the optimal implementation of hybrid execution operators, we could use them to implement various queries directly without further training.

Preprocess. In this phase, we generate dependency files, including SIMD and scalar description tables, and the configuration of processors. Specifically, description tables store the mapping relation of the hybrid intermediate description to SIMD and scalar statements. In addition, we also prepare additional information, including predefined test queries and operator templates implemented with the hybrid intermediate description.

Front-end. There are two components in the front-end, i.e., the candidate generator and the translator. The candidate generator generates the initial candidate node with the information of the processor, instructions, and operator templates. The candidate node contains the number of SIMD and scalar statements, and the value of *pack*, and then the translator generates the target code with this information. Formally, the number of SIMD and scalar statements is v and s, respectively, and the value of *pack* is p, the implementation generated by the translator is a set $<n_{vsp}, n_{(v+1)sp}, n_{v(s+1)p}, n_{vs(p+1)}, n_{(v-1)sp}, n_{v(s-1)p}, n_{vs(p-1)}>$, and these implementations are sent to the optimizer.

Optimizer. First, the optimizer compiles predefined test queries and concrete implementations of the target operators to executable programs. Then, it executes these executable programs and collects the execution times. Next, the optimizer classifies these implementations into two categories: winner and loser. If the execution time of the current implementation is shorter than the current node n_{vsp}, it is the winner and then is appended to the candidate list. Otherwise, the current implementation is added to the end list, and the variants derived from it will not be generated and tested.

B. Hybrid Intermediate Description

The hybrid intermediate description is an abstract intermediate representation of SIMD and scalar statements, and variables. The interface of the hybrid intermediate description is similar to the SIMD provided as an intrinsic function. Each hybrid intermediate description is a function, and its parameter types include the hybrid intermediate description, C/C++ built-in, and SIMD types. In the case that the processor does not support the specific SIMD instruction, we use multiple scalar instructions or a combination of other SIMD instructions to achieve the purpose of interface consistency. Take the SIMD instruction *gather* as an example, it is not supported by Neon currently, so the underlying implementation is scalar statements.

In detail, each hybrid intermediate description corresponds to a set of concrete implementations that contain different numbers of isomorphic SIMD and scalar statements. The different implementation handles different numbers of arguments, but users do not need to care about this. They could specify hybrid intermediate description types, and HEF will automatically complete the type conversion. Table I presents the interfaces and descriptions of the different hybrid intermediate descriptions, including load, mul, and add. In addition, the supported variable types are shown in Table II, including 16-bit, 32-bit, and 64-bit integers.

IV. Implementation

The detailed procedure of HEF is shown as Fig. 5. This section gives a detailed description of components, including Candidate Generator, Translator, and Optimizer.

A. Candidate Generator

Although the test-based approach could find the optimal implementation, it requires generating and testing all possible cases if without guidance. The candidate generator generates an initial candidate node by leveraging the information about the SIMD and scalar pipelines, and the latency and throughput of the instructions used to implement the operator. Compared with the random and fixed initial input, the candidate generator utilizes the information of the processor, instructions, and

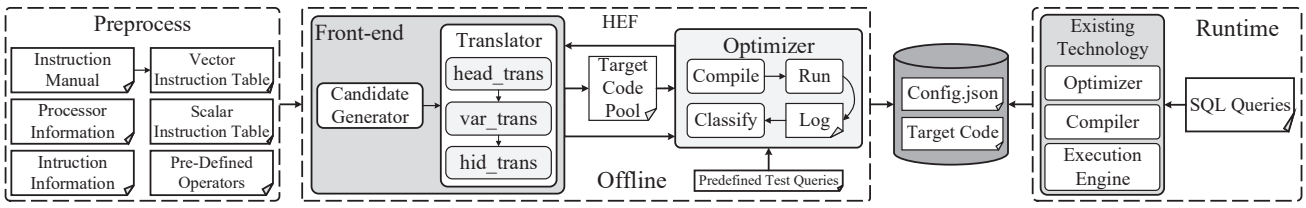

Fig. 4. Hybrid Execution Framework

TABLE I
PARTIAL HYBRID INTERMEDIATE DESCRIPTION TO DIVERSE VECTOR INSTRUCTION SET ARCHITECTURES

Op	Description	Scalar	AVX2	AVX-512
Add	a = hi_add_epi64(b, c)	for(...){a_i = b_i + c_i}	a = _mm256_add_epi64(b, c)	a = _mm512_add_epi64(b, c)
Mul	a = hi_mul_epi64(b, c)	for(...){a_i = b_i * c_i}	a = _mm256_mullo_epi64(b, c)	a = _mm512_mullo_epi64(b, c)
And	a = hi_and_epi64(b, c)	for(...){a_i = b_i & c_i}	a = _mm256_and_epi64(b, c)	a = _mm512_and_epi64(b, c)
Store	hi_store_epi64(a, b)	for(...){*b_i = a_i}	_mm256_store_epi64(a, b)	_mm512_store_epi64(a, b)
Load	a = hi_load_epi64(b)	for(...){a_i = *b_i}	a = _mm256_load_epi64(b)	a = _mm512_load_epi64(b)
Gather	a=hi_gather_epi64(b, c)	for(...){a_i = *b_i}	a=_mm256_i64gather_epi64(b, c)	a=_mm512_i64gather_epi64(b, c)
...	...	...	...	...

TABLE II
VARIABLES OF HYBRID INTERMEDIATE DESCRIPTION TO DIVERSE VECTOR INSTRUCTION SET ARCHITECTURES

Type	Description	AVX-512	AVX-2
16-bit integer	vint16, uint16	_mm512i	_mm256i
32-bit integer	vint32, vuint32	_mm512i, _mm512u	_mm256i, _mm256u
64-bit integer	vint64, vuint64	_mm512i, _mm512u	_mm256i, _mm256u
32-bit float	vfloat	__m512	__m256
64-bit float	vdouble	__m512d	__m256d

operators, and then could obtain the initial input closer to the optimal result, reducing the overhead of code generation and testing.

Particularly, we use the candidate generator to generate an appropriate initial input instead of the precise value due to the complexity of the modern processor. We pay attention to the number of the SIMD and scalar pipelines, and the latency and throughput of instructions while ignoring the influence including the out-of-order buffer, and the fetch and decode bandwidth. We overlook these factors because we could not intuitively capture the relationship between these factors and the efficiency of hybrid execution. Although we neglect some information, it is possible to get the optimal implementation in the end, because we continue to search the optimal implementation in the following phase.

Specifically, to make use of the information in a simple and efficient way, we utilize it with a two-stage model. In the first stage, we focus on the information of the processor, especially the number of the SIMD and scalar pipelines, and overlook the information of instructions and operators. Then, the candidate generator decides the number of SIMD and scalar statements within *pack* according to the pipelines. For pipelines shared with SIMD and scalar, we treat such pipelines as SIMD exclusive because SIMD is more efficient than scalar in most cases under the data analytics workload. In the second stage, we determine the value of *pack* with the latency and throughput of the instruction. We find the instruction with the maximum $\frac{latency}{throughput}$ from the operator template. Then, we get the SIMD instruction that contains the most parameters, and the number of the parameter is $argc$. Finally, we determine the value of *pack* with the equation $min\{\frac{latency}{throughput}, \frac{32}{max\{s\times 3, v\times argc\}}\}$. The rationale behind it is most scalar instructions use three registers at a time, while Skylake has 32 general purpose scalar and vector registers respectively. According to this observation, the candidate generator could reduce the execution intervals as much as possible without data exchanging between registers and cache.

It is noted that we ignore the different latency to access data from the different level cache. The instruction latency given in the user manual [7] is the latency to access data from the L1 cache. Meanwhile, accessing data from the L2 cache, last level cache (LLC) and memory, has higher latency than the L1 cache. In practice, the data access from the L1 cache usually is the main factor affecting the performance.

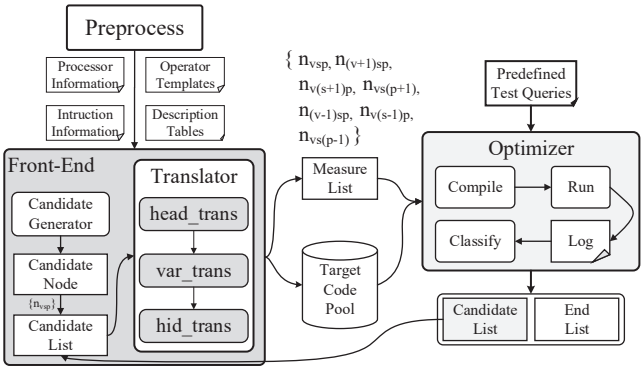

Fig. 5. The process of hybrid intermediate descriptions to target codes.

B. Translator

As the core component of HEF, the translator translates the hybrid intermediate description to the target code. Algorithm 1 presents the detailed process of translating the predefined operators to the target code. The translator depends on the operator template file, and the vector and scalar description tables. The template of the operator is a string stored in the operator template file, and it stores an operator list and an operator dictionary. The list contains all the operators that need to be generated, and the dictionary stores the map between the operator name and the implementation. To add a new operator, users could write the operator template with the hybrid intermediate description, and then add it to the list and dictionary.

```
hash(const uint64_t *val,const uint64_t size,
 const uint64_t batch_size, uint64_t *out){
 uint64_t ofs=0; const uint32_t r = 47; ...
 hi_uint64 data; hi_uint64 k; hi_uint64 dst;
 for(...){data=hi_load_i64(val+ofs);
  k=hi_mullo_i64(data,m);kr=hi_srli_i64(k,r);
  kr=hi_xor_i64(kr,k); kr=hi_mullo_i64(kr,m);
  hval = hi_xor_i64(h, kr); ... ofs = i;}}
```

(a)

```
hash(const uint64_t *val,const uint64_t size,
 const uint64_t batch_size, uint64_t *out){
 ... __m512i kr_v0_p0; int64_t kr_s0_p0;
int64_t kr_s1_p0; int64_t kr_s2_p0; ...
for(...){data_v0_p0=_mm512_loadu_epi64(val+ofs+0);
  data_s0_p0=*(val+ofs+8); data_s1_p0=*(val+ofs+9);
  data_s2_p0 = *(val + ofs + 10);
  data_v0_p1=_mm512_loadu_epi64(val+ofs+11);...}}
```

(b)

```
hash(const uint64_t *val,const uint64_t size,
 const uint64_t batch_size, uint64_t *out){
 for(...){ data_v0_p0=_mm512_loadu_epi64(val+ofs+0);
  data_v1_p0 = _mm512_loadu_epi64(val + ofs + 8);
 data_s0_p0=*(val+ofs+16);data_s1_p0=*(val+ofs+17);
 ata_s2_p0 = *(val + ofs + 18);
  data_v0_p1 = _mm512_loadu_epi64(val + ofs + 19);
  data_v1_p1=_mm512_loadu_epi64(val+ofs+27);...}}
```

(c)

Fig. 6. The hash value computation template implemented with the hybrid intermediate descriptor and its corresponding concrete implementations. (a)The hash value computation template. (b)The corresponding implementation with one SIMD statement and three scalar statements, and the value of *pack* is two. (c)The corresponding implementation with two SIMD statements and three scalar statements, and the value of *pack* is two.

```
1 vmovdqu64 (%rbx,%rax,8),%zmm4
2 vpmullq %zmm2,%zmm4,%zmm1
3 vpsrlq $47,%zmm1,%zmm0
4 vpxorq %zmm1,%zmm0,%zmm0
5 vpmullq %zmm2,%zmm0,%zmm0
6 vpxorq %zmm3,%zmm0,%zmm0
7 vpmullq %zmm2,%zmm0,%zmm0
8 vpsrlq $47,%zmm0,%zmm1
9 vpxorq %zmm0,%zmm1,%zmm0
10 vpmullq %zmm2,%zmm0,%zmm0
11 vpsrlq $47,%zmm0,%zmm1
12 vpxorq %zmm0,%zmm1,%zmm0
13 vmovdqu64 %zmm0,(%r12,%rax,8)
14 addq $8, %rax
```

(a)

```
1 vmovdqu64 (%rbx,%r10,8),%zmm4
2 leaq 8(%r10), %rcx
3 leaq 16(%r10), %rsi
4 vmovdqu64 (%rbx,%rcx,8),%zmm14
5 leaq 24(%r10), %rdi
6 leaq 32(%r10), %r8
7 vpmullq %zmm0, %zmm4, %zmm2
8 vpmullq %zmm0, %zmm14, %zmm15
9 vpsrlq $47, %zmm2, %zmm1
10 vpxorq %zmm2, %zmm1, %zmm5
11 vpsrlq $47, %zmm15, %zmm4
12 vpmullq %zmm0, %zmm5, %zmm6
13 vpxorq %zmm15, %zmm4, %zmm2
14 vpmullq %zmm0, %zmm2, %zmm1
15 vpxorq %zmm3, %zmm6, %zmm7
16 vpmullq %zmm0, %zmm7, %zmm8
17 vpxorq %zmm3, %zmm1, %zmm5
18 vpmullq %zmm0, %zmm5, %zmm6
19 vpsrlq $47, %zmm8, %zmm9
20 vpxorq %zmm8, %zmm9, %zmm10
21 vpsrlq $47, %zmm6, %zmm7
22 vpmullq %zmm0,%zmm10,%zmm11
23 vpxorq %zmm6, %zmm7, %zmm8
24 vpmullq %zmm0, %zmm8, %zmm9
25 vpsrlq $47, %zmm11, %zmm12
26 vpxorq %zmm11, %zmm12, %zmm13
27 vmovdqu64 (%rbx,%rsi,8),%zmm12
28            ...
```

(b)

```
1 movq 64(%r13,%r14,8), %rcx
2 movq 72(%r13,%r14,8), %rdx
3 leaq 0(,%r14,8), %rbx
4 vmovdqu64 0(%r13,%r14,8),%zmm4
5 movq 80(%r13,%r14,8), %r9
6 imulq %rax, %rcx
7 vmovdqu64 88(%r13,%rbx),%zmm7
8 movq 152(%r13,%rbx), %r8
9 imulq %rax, %rdx
10 vpmullq %zmm3,%zmm4,%zmm6
11 movq 160(%r13,%rbx),%rdi
12 movq 168(%r13,%rbx),%rsi
13 imulq %rax, %r9
14 vpmullq %zmm3,%zmm7,%zmm5
15 movq %rcx, %r10
16 imulq %rax, %r8
17 shrq $47, %r10
18 imulq %rax, %rdi
19 xorq %r10, %rcx
20 imulq %rax, %rsi
21 movq %rdx, %r10
22 vpsrlq $47,%zmm6,%zmm2
23 shrq $47, %r10
24 imulq %rax, %rcx
25 xorq %r10, %rdx
26 movq %r9, %r10
27 vpxorq %zmm6,%zmm2,%zmm2
28       ...
```

(c)

Fig. 7. The AVX-512 and hybrid implementation compiled with different parameters. (a) The AVX-512 implementation compiled with "-O3 -mavx512f -mavx512dq -fno-tree-vectorize". (b) The AVX-512 implementation compiled with "-O3 -mavx512f -mavx512dq -fno-tree-vectorize -funroll-all-loops". (c) The hybrid implementation with one vector, three scalar statements, and the value of *pack* is two, and compiled with "-O3 -mavx512f -mavx512dq -fno-tree-vectorize".

There are two issues when translating the hybrid intermediate description to target hybrid statements. On the one hand, the operator template contains various types of statements, including declarations and initialization of the variable, memory access, and computational statements, and these miscellaneous statements require different conversion rules. On the other hand, these statements contain kinds of variables, and different types of variables require distinct transition guidelines. Specifically, for the built-in types of C/C++ and SIMD types, these variables do not need to be unrolled, but the hybrid intermediate description variables should be unrolled to multiple variables. The number of unrolling is related to the number of vector and scalar statements, and the value of *pack*. Besides, if the variable is a constant, it would be unrolled to a scalar and an SIMD variable, but not in relation to v, s, and the value of *pack*.

To address the first issue, we specify the rules of the operator template, where the declaration, definition, and initialization of variables must precede the memory load and other computation instructions. Then, inside the code generator, we implement the corresponding parts separately, including the declaration, definition, and initialization of variables. Specifically, the code generator first converts the header of the operator, and then generates the code about the declaration, definition, and initialization of scalar, SIMD, and hybrid execution description variables. Finally, it translates the memory access and computational hybrid intermediate description to the target code.

To tackle the second issue, we create three lists, two of them to store scalar and SIMD variables, and the third to store constant variables. During the variable expansion phase, we determine whether the variable is stored in lists. If it exists in the scalar or SIMD list, no expansion is performed, and it is directly filled in the corresponding position. Otherwise, if it is stored in the constant list, the variable is converted to a scalar and a vector variable. Finally, if the variable is neither in the scalar or SIMD list nor in the constant variable list, it is converted to multiple scalar and SIMD variables according to the corresponding parameters. For variable name conflicts, we solve them by adding a suffix to the variable, while considering whether the current variable is a scalar or a vector and the current number of layers unrolled.

By changing the number of vector and scalar statements, and the value of *pack*, HEF could generate a wide variety of codes that cover all the regular order combinations. Fig. 6 illustrates a hash computation with the hybrid intermediate descriptor and its corresponding implementation, and Fig. 7 shows its assembly code with different implementations or different compiler options. Since the compiler performs a rearrangement of instructions during the compilation phase, the resulting assembly instructions are not arranged exactly the way the code is generated, we do not produce irregular sequential combinations. As shown in Fig. 6(b) and Fig. 7(c), the vector instructions precede scalar instructions in the source code, however, scalar instructions precede vector instructions in the final generated assembly code. We manually rearrange the assembly code, in our experiments, the performance difference between the compiler-produced code and the best performing handwritten code is less than 2%.

Algorithm 1: The algorithm of generating the target code according to operator templates and description tables.

Input: Op : The input operator template.
T_v, T_s : The vector and scalar instruction table.
v, s : Vector and scalar statements within pack.
p : The value of *pack*.
Output: x : Target code strings.

$vt \leftarrow T_v$[variable] , $vi \leftarrow T_v$[instruction]
$st \leftarrow T_s$[variable] , $si \leftarrow T_s$[instruction]
$V_t \leftarrow init\_vt()$, $S_t \leftarrow init\_st()$
$const\_list \leftarrow \emptyset$, $scalar\_list \leftarrow \emptyset$, $simd\_list \leftarrow \emptyset$
$code\_lines \leftarrow split(op)$
// header initialization
$vhead\_init(code\_lines[0], vt, st,$
$const\_list, scalar\_list, simd\_list)$
for $j \leftarrow 1$ **to** $len(code\_lines)$ **do**
  // variables initialization
  if $is\_var\_init(code\_lines[j])$ **then**
    $scalar\_list, simd\_list,$
    $const\_list, x \leftarrow var\_init(code\_lines[j])$
  // instructions translation
  else if $is\_hid(code\_lines[j])$ **then**
    $hi\_ins, argus \leftarrow split(code\_lines[j])$
    $ins_v \leftarrow vi[hi\_ins]$, $ins_s \leftarrow si[hi\_ins]$
    for $argu \in argus$ **do**
      $argu_s, argu_v, argu_c \leftarrow arg\_unroll($
      $scalar\_list, simd\_list, const\_list, argu)$
    for $i \leftarrow 0$ **to** $p-1$ **do**
      for $k \leftarrow 0$ **to** $v-1$ **do**
        $x \leftarrow ins_v + argu_{v\_k\_i} + argu_c$
      for $n \leftarrow 0$ **to** $s-1$ **do**
        $x \leftarrow ins_s + argu_{s\_n\_i} + argu_c$
return x

C. Optimizer

The diverse concrete implementations of the hybrid execution description vary in three dimensions, i.e., the number of the scalar and SIMD statements, and the *pack* size. It is observed that the trend of the running time of the different implementations is regular. Specifically, if the runtime of the n_{vsp} is less than $n_{(v+1)sp}$, the runtime of the $n_{(v+2)sp}$ being longer than $n_{(v+1)sp}$ is identified. It is noted that the case applies not only to v but also to s and p. This is deterministic when the running time is increasing, but indeterminate when the running time is decreasing. If the runtime of the $n_{(v+1)sp}$ is less than the n_{vsp}, it is possible that the runtime of the $n_{(v+2)sp}$ is greater or less than the n_{vsp}, but we could confirm that the current optimal implementation is $n_{(v+1)sp}$. With the information, we prune the node that has undergone a transformation and has long runtime than the current node. Taking an example to explain, assuming n_{vsp} as current node, and the nodes that have undergone a transformation being a set $<n_{(v+1)sp}, n_{v(s+1)p}, n_{vs(p+1)}, n_{(v-1)sp}, n_{v(s-1)p}, n_{vs(p-1)}, n_{vsp}>$, we append the node to the candidate list with shorter runtime than n_{vsp}, and the node with longer runtime is pruned, and their variants do not need to generate and test anymore. The rationale behind the pruning is as follows. On the one hand, before reaching the optimum performance, the processor utilization could be improved by increasing the number of instructions within the pack, and a shorter execution time could be obtained. On the other hand, after achieving the peak performance, continuing to add instructions increases execution time due to the limit on the number of registers, resulting in register and cache data swapping. On both sides of the optimal implementation, the change in execution time is monotonically increasing or decreasing.

Algorithm 2: The search strategy for getting the optimal implementation of the predefined operators.

Input: $node\_list$: Nodes need to be tested.
$tested\_list$: The tested nodes list.
Output: end_list : Nodes do not need be generated.
$candidate\_list$: Nodes with better performance.

$exe \leftarrow node\_list[0]$
$base\_time \leftarrow run(exe)$
for $node \leftarrow node\_list[1 \ldots n]$ **do**
  $exe \leftarrow compile(impl(node))$
  $time \leftarrow run(exe)$
  if $(base\_time > time)$ **then**
    $candidate\_list \leftarrow node$
  else
    $end\_list \leftarrow node$
return $candidate\_list$, end_list

The pruning algorithm is presented as Algorithm 2, and we integrate it into the framework as the optimizer. The pruning algorithm is introduced to cut out those unnecessary branches, while we could still get the optimal performance implementation, since the relationship between candidate nodes is a strongly-connected graph. For example, assuming the optimal initial candidate node is n_{132} ($v=1, s=3, p=2$), and the target node is n_{113} ($v=1, s=1, p=3$), and the variants n_{133}, n_{142}, n_{232}, n_{032} have longer runtime than n_{132}, then they are pruned and appended to the end list instead of the candidate list. Although n_{133} is transformed twice $<n_{123}, n_{113}>$ to get the optimal implementation and it has been pruned, we could get the target node n_{113} with the path $<n_{132}, n_{122}, n_{112}, n_{113}>$ or $<n_{132}, n_{122}, n_{123}, n_{113}>$ in the end. With the available test results, the pruning algorithm could reduce the overhead by avoiding generation and test of unnecessary implementations.

V. Evaluation

To evaluate the case of analytical workloads, we adopt SSB [27], which has been widely used in various data analytics research studies [43] [13] [22]. SSB includes 13 SQL queries that perform filter, join, aggregation, and group-by operators upon the main fact table of commercial data along with at most four reference tables. The scale factor of SSB is set to SF10

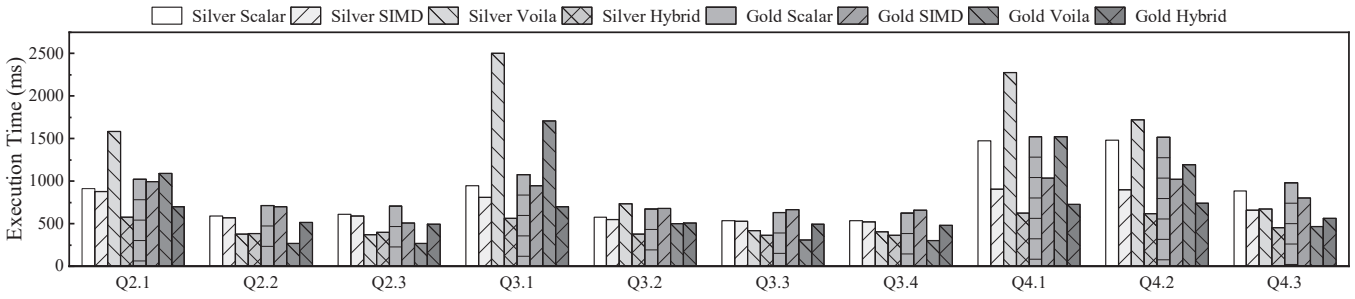

Fig. 8. Small-scale Workload (SSB SF10).

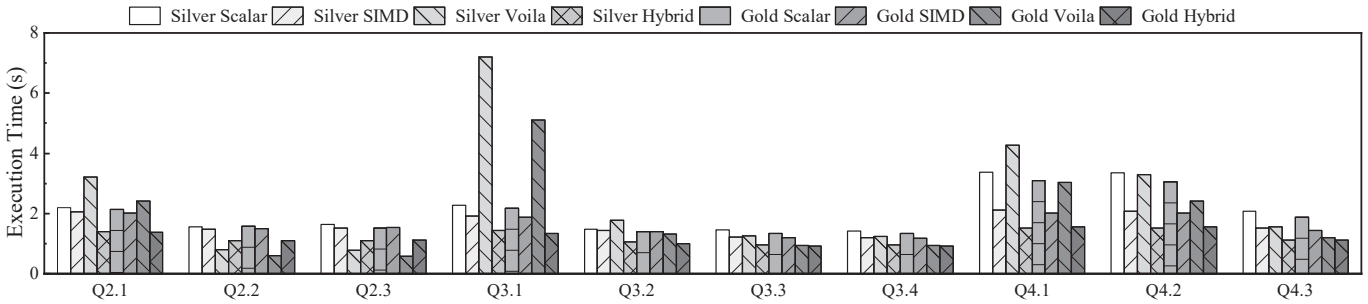

Fig. 9. Medium-scale Workload (SSB SF20).

(roughly 10 GB) to simulate the small-scale workloads, SF20 (roughly 20 GB) to simulate the media-scale workloads, and SF50 (roughly 50 GB) to simulate the large-scale workloads. In this paper, we focus on making better use of the processor's resources, so we do not select the queries which bottleneck lies in memory bandwidth. In addition, we apply a large linear hash table for hash join to reduce the conflicts and avoid data access becoming the bottleneck.

A. Setup and Metrics

Our experiments leverage two types of server settings. One of the servers is equipped with the Intel®Xeon®Silver 4110 CPU, which has only one fused AVX-512 unit per core. It has 160 GB of main memory and the operating system is CentOS (version 7.5.1804), and the version of Perf is 3.10.0-957.12.2.el7.x86_64.debug. The other server is equipped with the Intel®Xeon®Gold 6240R CPU, which contains two AVX-512 units per core. It has 256 GB of main memory and the operating system is Ubuntu (version 18.04.6 LTS), and the version of Perf is 5.4.157. We use GCC 10.2.0., and invoke GCC with -march=native -O3 -mavx512f -mavx512dq -fno-tree-vectorize -funroll-all-loops on the overall performance evaluation. In addition, we invoke GCC with -march=native -O3 -mavx512f -mavx512dq -fno-tree-vectorize on the synthetic benchmark evaluation.

B. Overall Performance

To evaluate the overall performance under data analytics workloads, we compare the implementation based on the HEF with purely scalar and purely SIMD implementations, and the state-of-the-art system Voila [14].

As a baseline, we use the operator, pipeline, and the materialization strategy described in VIP [33], which applied pipeline execution for the SIMD implementation. We adopt the same configuration for queries implemented with HEF. Moreover, we apply the linear probe hash table in our experiments instead of the hash table mentioned in VIP, due to their optimization method being oriented to 32-bit integers rather than 64-bit integers. The optimal implementation of the hybrid execution descriptor in HEF has one SIMD statement and one scalar statement, and the value of *pack* is three, where SIMD instruction set is AVX-512.

We also compare HEF with the state-of-the-art system Voila [14], and all the queries run with the parameter "–optimized –default_blend computation_type = vector(1024), concurrent_fsms = 1, prefetch = 1". It indicates that Voila uses vectorized execution and the batch size is 1024, with prefetching and a state-machine, and the compiler optimization option O3 is enabled.

First, we compare HEF with purely scalar and purely SIMD implementations. Fig. 8 reports execution times of the 10 queries of SSB under the small-scale SF10 workload on the Intel®Xeon®Silver 4110 CPU and Intel®Xeon®Gold 6240R CPU. For Q2.x (three joins), HEF achieves up to 1.58× speedups than the purely scalar, and 1.53× than the purely SIMD implementation. Furthermore, for Q3.x (three joins), HEF achieves up to 1.69× speedups than the purely scalar implementation, and it achieves up to 1.44× speedups than the purely SIMD implementation. For Q4.x (four joins), HEF outperforms the purely scalar implementation with 2.38×, and it outperforms the purely SIMD implementation with 1.45×. The experimental results of medium-scale SF20 workload and

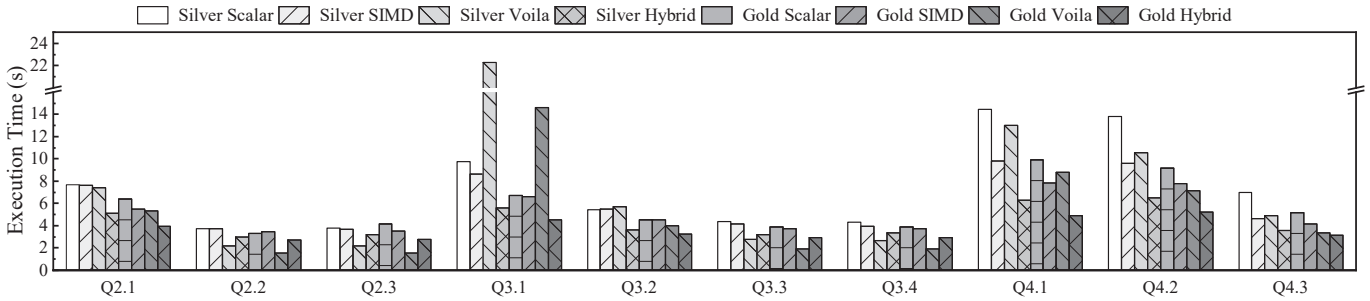

Fig. 10. Large-scale Workload (SSB SF50).

TABLE III
THE DETAILED INFORMATION OF THE DIFFERENT IMPLEMENTATIONS OF Q3.3 UNDER THE SMALL-SCALE SF10 WORKLOAD, AND ON THE INTEL®XEON®SILVER 4110 CPU.

Attributes	Scalar	SIMD	Voila	Hybrid
Instructions (10^8)	23.3	8.6	11.8	10.1
LLC-misses (10^6)	18.47	18.19	4.18	18.03
IPC	1.19	0.46	1.46	0.70
Frequency	2.97	2.86	1.77	2.84
Time (ms)	657	653	433	510

TABLE IV
THE DETAILED INFORMATION OF THE DIFFERENT IMPLEMENTATIONS OF Q2.3 UNDER THE MEDIUM-SCALE SF20 WORKLOAD, AND ON THE INTEL®XEON®SILVER 4110 CPU.

Attributes	Scalar	SIMD	Voila	Hybrid
Instructions (10^8)	49.2	23.4	22.3	25.1
LLC-misses (10^6)	33.3	32.8	8.4	32.8
IPC	1.06	0.54	1.63	0.75
Frequency	2.93	2.83	1.78	2.86
Time (ms)	1579	1519	763	1164

large-scale SF50 workload are shown in Fig. 9 and Fig. 10 respectively. As a result, HEF performs better than purely scalar and purely SIMD implementations on all queries with all the data scales.

It is observed that HEF performs better on the Intel®Xeon®Silver 4110 CPU than on the Intel®Xeon®Gold 6240R CPU under the small-scale SF10 workload. The reason is that HEF executes more instructions on the latter, and the instruction number is 2.38×10^9, but only 2.0×10^9 instructions on the former. However, under the medium-scale SF20 workload and large-scale SF50 workload, HEF executes roughly the same number of instructions on the two kinds of processors.

Meanwhile, we notice that HEF achieves different speedup ratios on the different scale workloads. This is because the size of the hash table generated during hash join differs from different scale workloads. The different size hash tables are stored in different levels of cache, and the data access from different levels of cache has caused an effect on the execution time. Although HEF gets lower IPC than the purely scalar implementation, it has fewer instructions than the purely scalar implementation. Meanwhile, compared with the purely SIMD implementation, HEF has more instructions than the purely SIMD implementation, but it achieves higher IPC. Correspondingly, compared with the purely scalar and SIMD implementation, HEF requires the least number of clock cycles.

Next, we turn our attention to the performance comparison between HEF and Voila, the results are also shown as Fig. 8, Fig. 9, and Fig. 10. HEF achieves up to 2.75× speedups than Voila on Q2.1, but Voila achieves up to 1.02× and 1.07× than HEF on Q2.2 and Q2.3 respectively. HEF outperforms Voila on Q3.1 and Q3.2, but it performs worse than Voila on Q3.3 and Q3.4, owing to the selectivity of Q3.3 and Q3.4 being very high (less than 1%). HEF has better performance than Voila on Q4.1 and Q4.2, but similar performance on Q4.3. We observe that Voila and HEF perform quite differently on different queries. Compared with Voila, HEF has better performance when the selectivity is low but performs worse while the selectivity is high, this is because Voila uses a different hash join and caching strategy compared with HEF.

To further figure out the reason why different implementations perform differently on the same dataset, we use *perf_event* to capture the detailed runtime information of the different implementations, including the number of instructions, IPC, and CPU frequency. The results are presented in Table III, Table IV, and Table V, respectively. The performance of Voila is particularly poor in some cases, because it applies state machines and data prefetching in the synthesized code, and it caches more intermediate results,

TABLE V
THE DETAILED INFORMATION OF THE DIFFERENT IMPLEMENTATIONS OF Q2.1 UNDER LARGE-SCALE SF50, AND ON THE INTEL®XEON®GOLD 6240R CPU.

Attributes	Scalar	SIMD	Voila	Hybrid
Instructions (10^9)	12.5	5.18	17.0	5.74
LLC-misses (10^7)	10.4	10.78	4.78	10.83
IPC	0.68	0.34	1.27	0.50
Frequency	3.20	3.05	2.49	3.06
Time (s)	6.1	4.8	5.4	4.1

which introduces enormous instructions when the selectivity is low. As shown in Table V, the synthesized code generated by Voila achieves the highest IPC as it has the highest LLC cache hit rate, but it is slower than HEF since the synthesized code contains more instructions than HEF. The instruction number of Q2.1 implemented with Voila is more than the purely scalar implementation, as shown in Table V. However, when the selectivity is high, such as Q2.3, the filter rate is 0.1% in the first join, Voila has the best performance in our experiments. This is because, with the rapid increase of selectivity, the instruction generated by Voila decrease rapidly, and the instruction number is less than the purely SIMD implementation, as shown in Table IV.

HEF performs better than Voila for a majority of queries in SSB under all data scales. Especially, under the small-scale SF10, HEF performs better than Voila on most queries and has similar performance to Voila on other queries, when tested on the Intel®Xeon®Silver 4110 CPU. In addition, HEF has a more stable performance on different queries compared with Voila.

C. Synthetic Benchmarks

In order to figure out how the hybrid execution with the optimization *pack* improves performance, we conduct two specific experiments. For the hybrid execution of SIMD and scalar, we adopt MurmurHash [1] computation as the benchmark. In addition, for the optimization *pack*, we apply the CRC64 [19] as the benchmark. We compute the hash value of 10^9 64-bit integer elements with MurmurHash and CRC64.

MurmurHash. The reason that we chose MurmurHash is that it is widely used in data analytics, and its bottleneck is computation, which contains multiplication, addition, shift, and logic operations.

It is noted that the optimal implementation on the Intel®Xeon®Silver 4110 CPU we get from HEF has one SIMD statement, three scalar statements, and the value of *pack* is two. This is because the Intel®Xeon®Silver 4110 CPU is equipped with one fused AVX-512 pipeline and four scalar pipelines, in which one of the scalar pipelines shares the issue port with the AVX-512. Although the Intel®Xeon®Gold 6240R CPU has two AVX-512 pipelines, it has the same optimal implementation. Since the processor works in a lower frequency when the implementation has two SIMD and scalar statements, it has a longer execution time, compared to the implementation with one SIMD and three scalar statements.

TABLE VI
THE EXECUTION TIME AND IPC OF MURMURHASH WITH DIFFERENT IMPLEMENTATIONS ON THE INTEL®XEON®SILVER 4110 CPU.

Attributes	Scalar	SIMD	Hybrid
Time (ms)	3305.64	3351.83	2646.68
IPC	3.31	1.25	2.08

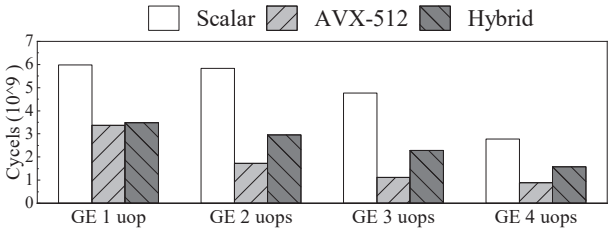

Fig. 11. Micro-operations executed in one cycle on the Intel®Xeon®Silver 4110 CPU, GE means greater or equal.

TABLE VII
THE EXECUTION TIME AND IPC OF MURMURHASH WITH DIFFERENT IMPLEMENTATIONS ON THE INTEL®XEON®GOLD 6240R CPU.

Attributes	Scalar	SIMD	Hybrid
Time (ms)	4956.76	4756.61	4134.97
IPC	1.28	0.86	0.91

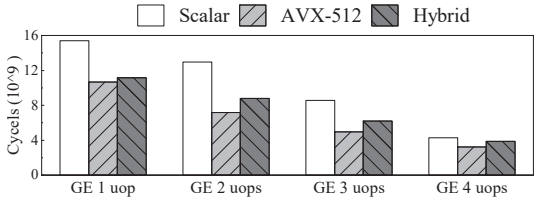

Fig. 12. Micro-operations executed in one cycle on the Intel®Xeon®Gold 6240R CPU.

The hybrid execution achieves higher IPC, and has the smaller execution time compared with the SIMD implementations, as shown in Table VI and Table VII. This is because HEF achieves higher micro-operations parallelism than the purely AVX-512 implementation on the different processors, as presented in Fig. 11 and Fig. 12. Compared with the purely scalar implementation, it achieves a lower IPC but has fewer instructions. Specifically, HEF achieves up to 1.25× than the purely scalar and SIMD implementation on the Intel®Xeon®Silver 4110 CPU, as shown in Table VI. It achieves up to 1.2× than the purely scalar implementation, and 1.15× than the purely SIMD implementation on the Intel®Xeon®Gold 6240R CPU, as presented in Table VII.

In addition, we notice that the scalar implementation has similar execution times to the SIMD implementation on the Intel®Xeon®Silver 4110 CPU. Because the computation is mainly shifting and logical operations rather than multiplication, the Intel®Xeon®Silver 4110 CPU has multiple scalar shift and logical pipelines but only one fused AVX-512 pipeline, so the scalar implementation achieves a higher IPC than the SIMD implementation. In contrast, the purely SIMD implementation achieves higher micro-operations parallelism on the Intel®Xeon®Gold 6240R CPU, since the processor has more AVX-512 pipelines.

CRC64. Cyclic redundancy check (CRC) is a hash function to detect or check possible errors in data transmission or storage, such as Redis [38]. The bottleneck of its computation [19] is the L1 cache access. To the SIMD implementation, the main cost instruction is *vpgather*. It is a befitting workload to test the impact of the *pack*. The optimal implementation of the

TABLE VIII
THE EXECUTION TIME AND IPC OF CRC64 WITH DIFFERENT IMPLEMENTATIONS ON THE INTEL®XEON®SILVER 4110 CPU.

Attributes	Scalar	AVX-512	Hybrid
Time (ms)	1064.09	910.22	380.01
IPC	2.72	0.38	1.26

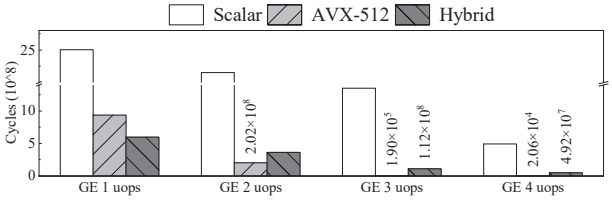

Fig. 13. Micro-operations executed in one cycle on the Intel®Xeon®Silver 4110 CPU.

hybrid descriptor we get from HEF has eight SIMD statements without scalar statements.

Table VIII and Table IX show the testing results of implementations on the Intel®Xeon®Silver 4110 CPU and Intel®Xeon®Gold 6240R CPU, respectively. We compared HEF with the purely scalar and SIMD implementation, and its instruction number is closer to the purely SIMD implementation, but it has a higher IPC (up to 3.33×) than SIMD, so it uses fewer cycles than the SIMD implementation. It then has a smaller execution time than the purely SIMD (by 44%) and scalar (by 37%) implementation.

Besides, HEF has fewer stalled cycles from the aspect of memory accessing and micro-operations issuing. Specifically, HEF enables three or more micro-operations to be executed simultaneously in 19% of the total cycle, and the processor executes two or more micro-operations in parallel in 61% of the total cycle. To the purely SIMD implementation, the processor executes two or more micro-operations simultaneously in 10% of the total cycle. On the Intel®Xeon®Gold 6240R CPU, HEF behaves basically the same as on the Intel®Xeon®Silver 4110 CPU. Compared to the purely SIMD implementation, it has fewer cycles with more than three micro-operations, and it has a higher IPC with half cycles. Subseqently, HEF achieves higher instruction parallelism proportionally, and the results are shown in Fig. 13 and Fig. 14.

TABLE IX
THE EXECUTION TIME AND IPC OF CRC64 WITH DIFFERENT IMPLEMENTATIONS ON THE INTEL®XEON®GOLD 6240R CPU.

Attributes	Scalar	SIMD	Hybrid
Time (ms)	1177.27	1042.67	533.42
IPC	1.98	0.42	0.78

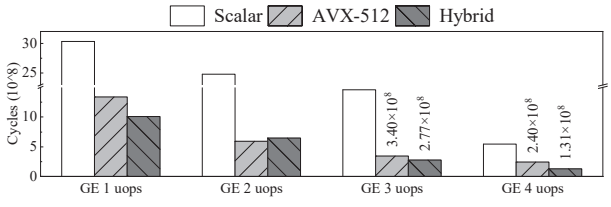

Fig. 14. Micro-operations executed in one cycle on the Intel®Xeon®Gold 6240R CPU.

VI. RELATED WORK

SIMD in data analytics systems. Since [46] applying SIMD to optimize the operation of data analytics systems, SIMD has been widely studied in data analytic systems. With SIMD, higher computational efficiency could be achieved, which can reduce the overhead of the computational operators, including hash joins, hash-based aggregations, and bloom filters. Besides, it could reduce branch misses [15], and provide convenient instructions to access data, such as *gather* and *scatter*. Frequently-used operations, such as scans [4] [23] [12] [17], index scans [24], joins [18] [20] [2] [10], aggregations [31], indexing and sorting [16] are implemented with SIMD [46] [39] [30]. These research efforts exploited SIMD instructions to accelerate the individual operators, while ROF [26] and VIP [33] [32] proposed and implemented a vectorized execution engine. IMV [11] uses SIMD to optimize hash join from a different perspective. It proposed an efficient way to utilize SIMD to implement hash joins by reducing cache misses and branch misses with interleaved execution of multiple instances of vectorized code. Voila [14] proposes a framework that implements the synthesis of state-of-the-art paradigms, including data-centric compilation and vectorized execution. Micro adaptivity [37] uses the vw-greedy algorithm to choose the most promising flavor potentially for each function call to increase performance robustness and save the time spent in query optimization.

In addition, there are several other efforts that used the vector unit of the processors to accelerate data processing. [41] introduced *template vector library* (TVL) as a hardware-oblivious SIMD parallelism concept for in-memory column-stores, since they think programming with vector extensions is a non-trivial task. [30] and IMV [11] applied their work not only on the processor but also on the many-integrated-cores (MIC) architecture. [29] evaluated the performance of the data analytics workloads with the vector supercomputer SX-Aurora TSUBASA [45], which is equipped with a strong vector engine as a (co-)processor. [42] uses the NEC vector engine as an accelerator to speedup the analytical query processing.

Auto-vectorization with SLP. There are many works based on SLP to implement auto-vectorization compilers. GoSLP [25] used an integer linear programming (ILP) solver to discover vectorization opportunities in high-level language code instead of heuristics. PSLP [35] proposed a novel vectorization algorithm that could vectorize codes containing non-isomorphic instruction sequences. It injects a near-minimal number of redundant instructions into the code, and transforms non-isomorphic sequences into isomorphic ones. SuperGraph-SLP [34] proposed an improved vectorization algorithm to further vectorize instructions. VW-SLP [36] proposed a novel algorithm with adjusting the vector width at an instruction granularity, adapting to the SIMD parallelism characteristics.

VeGen [3] used SLP to pack the scalar instructions to non-SIMD vector instructions systematically instead of ad-hoc.

VII. Discussion

HEF utilizes the SIMD and scalar execution units in parallel and achieves short execution intervals under the data analytics workloads. We discuss the limitations of HEF and several directions to continue optimizing HEF.

We implement a group of commonly used operators at present, and users could use the hybrid intermediate description to implement additional customized operators. Adding new operators will be optimized in the offline phase, in order to achieve performance improvement. Although HEF provides a candidate generator to generate a small search space, it overlooks some not-so-important factors, including fetch, decoding bandwidth, and the size of the out-of-order buffer. In further work, we plan to apply machine learning methods to get an initial node closer to the ultimate result by considering all the factors.

We implement queries by assembling the operators with the pre-tested performance in this paper. Although the current method has higher performance than the purely scalar and SIMD implementations, it may not be the optimal implementation for the whole query. In the further, we will enable HEF to support the function of dynamic selection, which makes it dynamically select operators with different implementations according to queries, in order to achieve higher throughput and shorter response time.

VIII. Conclusion

We present HEF to co-utilize SIMD and scalar units under the data analytics workload, achieving higher throughput and shorter response time than purely scalar and purely SIMD implementations. Furthermore, we propose the hybrid intermediate description to unify the implementation of SIMD and scalar. It is the core conception of HEF, shielding the underlying implementation details.

We implement the translator that implements the same operator with various combinations of SIMD and scalar statements. As a core component of HEF, it could automatically translate hybrid intermediate descriptions to the target code. Meanwhile, we introduce two approaches to reduce the translation and testing overhead. HEF generates the initial input close to the final result by using the information of processors, instructions, and operators. HEF reduces the search space by pruning candidate nodes with the acquired test results.

In conclusion, HEF enables SIMD and scalar instructions to execute in parallel, and achieves shorter instruction execution intervals. The hybrid execution of SIMD and scalar has inspired a new way to use processor resources more efficiently. It could be applied not only to data analytics systems but also to deep learning systems, especially in the inference stage. Moreover, it provides a new idea for compiler optimization and processor design.

Acknowledgment

This research was supported by the National Natural Science Foundation of China (No. 62141214, 62272171, and 61732014). The authors would like to thank the shepherd and anonymous reviewers for their valuable feedback and guidance. Chuliang Weng (clweng@dase.ecnu.edu.cn) is the corresponding author.

References

[1] Austin Appleby. MurmurHash3, 2012. *https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp*, 2012.

[2] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, Main-memory Joins: Sort vs. Hash Revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[3] Yishen Chen, Charith Mendis, Michael Carbin, and Saman P. Amarasinghe. Vegen: A Vectorizer Generator for SIMD and Beyond. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 902–914. ACM, 2021.

[4] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, 2008.

[5] ARM Corporation. "Introducing NEON Development Article". https://developer.arm.com/documentation/dht0002/a/Introducing-NEON/NEON-architecture-overview?lang=en.

[6] Intel Corporation. "Intel 64 and IA-32 Architectures Optimization Reference Manual". https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[7] Intel Corporation. Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[8] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro*, 37(2):52–62, 2017.

[9] Wenbin Fang, Bingsheng He, and Qiong Luo. Database Compression on Graphics Processors. *Proceedings of the VLDB Endowment*, 3(1-2):670–680, 2010.

[10] Zhuhe Fang, Zeyu He, Jiajia Chu, and Chuliang Weng. SIMD Accelerates the Probe Phase of Star Joins in Main Memory Databases. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 476–480, 2019.

[11] Zhuhe Fang, Beilei Zheng, and Chuliang Weng. Interleaved Multi-Vectorizing. *Proceedings of the VLDB Endowment*, 13(3):226–238, 2019.

[12] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 31–46, 2015.

[13] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*, pages 1603–1618, 2018.

[14] Tim Gubner and Peter Boncz. Charting the Design Space of Query Execution using VOILA. *Proceedings of the VLDB Endowment*, 14(6):1067–1079, 2021.

[15] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. Faster Set Intersection with SIMD Instructions by Reducing Branch Mispredictions. *Proceedings of the VLDB Endowment*, 8(3):293–304, 2014.

[16] Hiroshi Inoue and Kenjiro Taura. SIMD and Cache-friendly Algorithm for Sorting an Array of Structures. *Proceedings of the VLDB Endowment*, 8(11):1274–1285, 2015.

[17] Hao Jiang and Aaron J Elmore. Boosting Data Filtering on Columnar Encoding with SIMD. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, pages 1–10, 2018.

[18] Peng Jiang and Gagan Agrawal. Efficient SIMD and MIMD Parallelization of Hash-based Aggregation by Conflict Mitigation. In *Proceedings of the International Conference on Supercomputing*, pages 1–11, 2017.

[19] David T Jones. An Improved 64-bit Cyclic Redundancy Check for Protein Sequences. *University College London*, 2009.

[20] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, 2009.

[21] Samuel Larsen and Saman Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. *ACM SIGPLAN Notices*, 35(5):145–156, 2000.

[22] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *Proceedings of the VLDB Endowment*, 9(14):1647–1658, 2016.

[23] Yinan Li and Jignesh M Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 289–300, 2013.

[24] Jianyuan Lu, Ying Wan, Yang Li, Chuwen Zhang, Huichen Dai, Yi Wang, Gong Zhang, and Bin Liu. Ultra-Fast Bloom Filters Using SIMD Techniques. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):953–964, 2018.

[25] Charith Mendis and Saman Amarasinghe. goSLP: Globally Optimized Superword Level Parallelism Framework. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[26] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. Relaxed Operator Fusion for In-memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *Proceedings of the VLDB Endowment*, 11(1):1–13, 2017.

[27] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. The Star Schema Benchmark and Augmented Fact Table Indexing. In Raghunath Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking*, pages 237–252, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[28] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, Jamshed Jalal, Mark Werkheiser, and Anitha Kona. The ARM Neoverse N1 platform: Building Blocks for the Next-gen Cloud-to-edge Infrastructure SoC. *IEEE Micro*, 40(2):53–62, 2020.

[29] Johannes Pietrzyk, Dirk Habich, Patrick Damme, and Wolfgang Lehner. First Investigations of the Vector Supercomputer SX-Aurora TSUBASA as a Co-Processor for Database Systems. *BTW 2019–Workshopband*, 2019.

[30] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508, 2015.

[31] Orestis Polychroniou and Kenneth A Ross. High Throughput Heavy Hitter Aggregation for Modern SIMD Processors. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, pages 1–6, 2013.

[32] Orestis Polychroniou and Kenneth A Ross. Towards Practical Vectorized Analytical Query Engines. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pages 1–7, 2019.

[33] Orestis Polychroniou and Kenneth A Ross. VIP: A SIMD Vectorized Analytical Query Engine. *The VLDB Journal*, 29(6):1243–1261, 2020.

[34] Vasileios Porpodas. SuperGraph-SLP Auto-Vectorization. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 330–342, 2017.

[35] Vasileios Porpodas, Alberto Magni, and Timothy M Jones. PSLP: Padded SLP Automatic Vectorization. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 190–201, 2015.

[36] Vasileios Porpodas, Rodrigo CO Rocha, and Luís FW Góes. VW-SLP: Auto-vectorization with Adaptive Vector Width. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–15, 2018.

[37] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. Micro Adaptivity in Vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1231–1242, 2013.

[38] Salvatore Sanfilippo. "CRC64 in Redis 5.0.". https://github.com/redis/redis/blob/5.0/src/crc64.c.

[39] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. Compilation in Query Execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, pages 33–40, 2011.

[40] David Suggs, Mahesh Subramony, and Dan Bouvier. The AMD "Zen 2" Processor. *IEEE Micro*, 40(2):45–52, 2020.

[41] Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Alexander Krause, Dirk Habich, Wolfgang Lehner, and Erich Focht. Hardware-oblivious SIMD Parallelism for In-memory Column-stores. In *Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR)*, 2020.

[42] Annett Ungethüm, Lennart Schmidt, Johannes Pietrzyk, Dirk Habich, and Wolfgang Lehner. Mastering the NEC Vector Engine Accelerator for Analytical Query Processing. In *Proceedings of the 2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*, pages 60–65, 2021.

[43] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. Concurrent Analytical Query Processing with GPUs. *Proceedings of the VLDB Endowment*, 7(11):1011–1022, 2014.

[44] Andrew Waterman and Krste Asanovic. "RISC-V "V" Vector Extension". https://github.com/riscv/riscv-v-spec.

[45] Yohei Yamada and Shintaro Momose. Vector Engine Processor of NEC's Brand-New Supercomputer SX-Aurora TSUBAS. In *Proceedings of A Symposium on High Performance Chips (HOT CHIPS)*, pages 19–21, 2018.

[46] Jingren Zhou and Kenneth A Ross. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 145–156, 2002.