Program your (custom) SIMD instruction set on FPGA in C++

Johannes Pietrzyk
TU Dresden
Dresden, Germany
johannes.pietrzyk@tu-dresden.de

Alexander Krause
TU Dresden
Dresden, Germany
alexander.krause@tu-dresden.de

Christian Färber
Intel Corporation
Munich, Germany
christian.faerber@intel.com

Dirk Habich
TU Dresden
Dresden, Germany
dirk.habich@tu-dresden.de

Wolfgang Lehner
TU Dresden
Dresden, Germany
wolfgang.lehner@tu-dresden.de

ABSTRACT

Field Programmable Gate Arrays (FPGAs) are more and more becoming a viable option for implementing data processing pipelines as their computing capacity as well as the access bandwidth between host and device memory continue to increase. Unfortunately, nowadays hardware description languages are still mainly used for programming FPGAs which implies major limitations. To tackle this issue, our paper shows that the general-purpose parallel processing architecture SIMD (Single Instruction Multiple Data) is a perfect match for FPGAs. With this specific architecture, we are able to consider an FPGA as SIMD processing unit and the necessary SIMD instruction set can now be implemented in C++. As we will present, this offers a lot of advantages if both software (SIMDified query processing) and hardware can be written consistently in C++.

1 INTRODUCTION

Generally, it can be stated that the relationship between hardware and software has reached a crucial point in time. While software has benefited for many years from higher clock cycles of modern CPUs, hardware components are still advancing at an incredible speed, providing a rich bouquet of novel techniques. For example, in the area of processing units, the core count has been increased and internal techniques like sophisticated SIMD (Single Instruction Multiple Data) instruction set extensions, pre-fetching, or branch prediction have been dramatically improved within modern CPUs. Moreover, alternative processing unit approaches such as GPGPUs (General Purpose Computation on Graphics Processing Unit) or FPGAs (Field Programmable Gate Arrays) have been developed. These alternative processing units are becoming more and more accessible and with the emerging interconnect Compute Express Link (CXL) [25] connectivity will improve dramatically. However, with these emerging opportunities, there also arises a plethora of new challenges regarding application implementation and integration on these hardware platforms.

In particular, FPGAs are an increasingly viable option for implementing efficient data processing pipelines [6, 14, 20], as they have recently gained increasing computational capacity and high-bandwidth access to device and host memory. FPGAs are integrated circuits that are configurable after being manufactured at any time and being internally composed of programmable logic blocks, a collection of on-chip memories, and arithmetic units (DSPs). To create a custom hardware module for an FPGA, a hardware description language (HDL), e.g., Verilog, is usually used to describe the operation mode of a specific application logic (state-of-the-art). This description is then translated via several steps to an implementation for an FPGA. Then, this implementation behaves like an application-specific integrated circuit. The main advantage of HDLs is that they equip the developers with fine-grained control to define a suitable data processing architecture leading to performant as well as resource-efficient FPGA designs. However, this fine-grained control makes programming more difficult and is therefore very different from higher-level languages such as C/C++, which are mainly used, e.g., for developing database query processing models.

Our Contribution: In this paper, we argue that the general-purpose parallel data processing paradigm SIMD is a perfect match for FPGAs. The SIMD architecture is characterized by the fact that the same operation is simultaneously applied on multiple data elements within a single instruction [11]. For more than a decade, modern CPUs have already supported SIMD processing by vendor-specific SIMD instruction set extensions, which are (i) explicitly provided in higher-level programming languages such as C++ and (ii) extensively used in the database domain as well [21, 22]. Therefore, in order to consider an FPGA as a flexible SIMD processing unit, we need a proper SIMD-oriented instruction set extension, e. g., just like SSE or AVX for Intel® CPUs. To achieve this, we show within this paper that the SIMD instruction set for an FPGA can be implemented in C++ using Intel® oneAPI. We chose this approach, since being able to build both software and the underlying architectural foundation in the same higher-level language yields many benefits, with a slimmer code base being just one example. The following aspects are covered in this paper:

- In Section 2, we introduce Intel®'s new cross-architecture language *Data Parallel C++ (DPC++)*. Moreover, we show (i) that the DPC++ compiler is able to synthesize arbitrary C++ code and (ii) to improve the performance a data parallel approach may be the way to go.
- In Section 3, we discuss how to program a SIMD instruction set for FPGAs in C++ and introduce various instruction examples including micro-benchmark results.

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution, provided that you attribute the original work to the authors and CIDR 2024. 14th Annual Conference on Innovative Data Systems Research (CIDR '24). January 14-17, 2024, Chaminade, USA.

- Based on this, we present a use case study in Section 4 and show how to port existing SIMD application code to FPGAs using two examples. One of these examples is the lightweight integer compression algorithm *Binary Packing*. The state-of-the-art SIMDification of this algorithm has some disadvantages that can be circumvented with custom SIMD instruction as introduced in Section 5. This clearly demonstrates the flexibility of our concept with regard to a HW/SW co-design.

Finally, we close the paper with a summary of learned best practices in Section 6, a discussion of related work in Section 7, and a short summary in Section 8. Our entire source code for this paper is available on GitHub[1,2].

2 FPGA PROGRAMMING IN C++

To overcome the complexity of HDLs to describe the operation mode of a specific application logic in FPGAs, manufacturers of FPGAs are also actively developing tools that enable programming of FPGAs in higher-level languages such as C/C++. Most recently, Intel® has released a new and powerful software development kit called oneAPI providing a unified programming model for diverse architectures such as CPUs, GPUs, and FPGAs [23]. In particular, Intel®'s oneAPI features a new cross-architecture language Data Parallel C++ (DPC++) as an implementation of SYCL™ [15], which is a heterogeneous programming framework built on top of modern C++ [4, 23]. DPC++ is open-source [13] and based on Clang/LLVM.

Moreover, DPC++ extends SYCL™ by adding additional features to enable peak application performance as well as to increase programmer productivity [4]. One of these extensions is *Unified Shared Memory (USM)*, which defines a pointer-based alternative to the buffer abstraction in SYCL™ [4]. Instead of using specific SYCL™ buffer objects for memory allocations, USM provides a familiar pointer-based approach to the regular C++ pointers. Based on that, USM defines three types of memory allocations: (i) device, (ii) host, and (iii) shared. While device allocations are only directly accessible through dereferencing a pointer on the actual device (FPGA or GPU), host or shared allocations are accessible on both the host (CPU) as well as on the device. The difference between host and shared allocations is that host allocations remain in host main memory, while shared allocations migrate to where they are being used without programmer intervention. Contents of the device-allocated memory have to be either explicitly populated or moved back to the host by, e. g., a SYCL™ `memcpy` operation from the host.

The foundation of Intel®'s oneAPI for FPGA is a *Board Support Package (BSP)* describing all hardware interfaces to the FPGA, like PCIe and DDR4. It also provides a shell design for these interfaces for faster application logic integration and synthesis to the FPGA. In particular, USM is part of the BSP, which enables us to let the data transfers be managed by the FPGA itself. This functionality frees the host CPU from the data transfer management. Moreover, the CPU is only involved in creating input and output buffers on the host memory and the FPGA gets the data as needed by the oneAPI application logic. The virtual-to-physical address translation for

```
1 template<typename T>
2 auto aggregate(T const * data, size_t elCount) {
3   T result = 0;
4   for (; data!=data + elCount; ++data) {
5     result += *data;
6   }
7   return result;
8 }
```

Figure 1: Aggregation - Naïve scalar approach.

data transfers is handled by the BSP. Throughout this paper, we conduct experiments on different FPGA cards:

(1) Agilex: A BittWare® IA-840f card, which is equipped with an Intel® Agilex® 7 AGF027 FPGA (BSP 2022.2) and 4x 16 GB DDR4. The interface to the host (Intel® XEON® Platinum 8351N) is a PCIe 4.0 with 16 lanes.

(2) Stratix 10: An Intel® D5005 PAC card equipped with an Intel® Stratix® 10 SX 2800 FPGA (DCP 2.0.1) and 4x 16 GB DDR4. The interface to the host (Intel® XEON® Gold 6238R) is a PCIe 3.0 with 16 lanes.

From here on, we refer to the hardware either as Agilex or Stratix 10 for brevity.

2.1 Naïve C++ programming

Generally, the DPC++ compiler is capable of synthesizing arbitrary C++ code into an FPGA design. To evaluate this capability for data processing, Figure 1 exemplarily shows a straightforward scalar aggregation method in C++ that sums up the individual element values of a column (represented via the pointer `data`). The data type is a template parameter and the pointer to the column (`data`) as well as the number of elements (`elCount`) are input parameters. The summation is done by a simple loop over all elements adding each element value to the variable `result`. At the beginning, the variable `result` is initialized with zero. Figure 3a depicts the achieved throughputs for host allocated `uint64_t` data for both FPGA cards and for different data sizes. In general, our achieved throughputs with increasing data volumes remain very low compared to the maximum possible bandwidth. There are many reasons for that, e. g., the DPC++ compiler may have difficulties parallelizing the code due to the data dependency in the loop. In addition, only individual data elements are requested and probably exclusively transferred via PCIe, which does not exhaust the potential bus width.

2.2 Data Parallel C++ programming.

To improve the throughput, a data-parallel approach based on the SIMD parallel processing paradigm may be a good candidate, as it represents a well known model from CPUs. Figure 2 shows the rewritten code with data parallelism in mind, where the number of elements processed in parallel (`VL`) is an additional template parameter. The inner loop (lines 7-9) performs an elementwise addition for VL elements, while the outer loop now has a step size of VL elements (line 5). To ensure that the DPC++ compiler properly detects the data parallel processing opportunity, the inner loop is annotated with the pre-processor directive `pragma unroll` (line 6). In the end, the individual intermediate sums still have to be added up (lines 11-13), yet the overall contribution of that part to the execution time is negligible. Figure 3b depicts the achieved

[1] Paper: https://github.com/db-tu-dresden/CIDR24-SIMD-FPGA
[2] SIMD-Abstraction Library: https://github.com/db-tu-dresden/TSL

throughputs for host allocated `uint64_t` data for increasing data level parallelism. As visible, the throughput on both FPGA cards increases with growing data parallelism reaching nearly interface speed for an inner loop count of 16 elements (1024 bits) on the Stratix 10, and 8 elements (512 bits) on the Agilex, respectively.

2.3 Using specialized data types

A common trend in modern programming frameworks are specialized data types that allow the compiler to better optimize the code, e. g., through using hardware-provided counterparts. For Intel® FPGAs, *Algorithmic C (AC) Datatypes* [12] constitute such optimized data types. *AC-Types* are class templates representing arbitrary-length integers, fixed-point as well as floating-point data, and complex datatypes. Those types bring in two remarkable advantages. First, they enable bit-accurate calculations and access to the underlying data. Second, when using *AC-Types*, the compiler can create narrower data paths, which may increase the maximum frequency ($f_{\max}$) of the resulting block design, and are thus increasing the algorithm's performance. Figure 4 depicts the adjusted code from Figure 2, but instead of using plain *C* types, we are utilizing an *ac_int* (AC-Type for integers) with a bitwidth corresponding to the given plan *C* integer type. In general, the code looks very similar with two exceptions:

(1) the base type for the register array is changed, and
(2) the initialization of the register array is done using a member method of *ac_int* rather than an implicit cast.

While the algorithm does not directly benefit from the strenghts of the AC-Types, like bit-accurate slicing and shifting, we will show in the following that better designs can be achieved even for such a scenario.

2.4 Analyzing FPGA resources

In the previous sections, we have already presented experimental throughput results for the variants with the plain C-types. For our simple aggregation example, the optimization with the specialized *ac_int* data types does not lead to a throughput improvement. Nevertheless, the usage of the specialized AC-Types has a resource-saving effect as illustrated in Figure 3c. The table in Figure 3c shows the used FPGA resources for our aggregation on Stratix 10 depending on the number of elements processed in parallel (VL). For comparison purposes, the FPGA resources for scalar execution are also displayed as well. While all variants run with the same $f_{\max}$ of 432.0Mhz, we observe differences in the necessary resources for a given design and the overall latency. For every tested register size (VL), using *ac_int* leads to lower latencies than the variants using plain C-types. With regards to computing resource demands, the *ac_int* variants need less adaptive look-up tables (ALUTs) and flip-flops (FF) overall. In general, each variant uses less than 1% of the available resources of our Stratix 10. The same applies for the Agilex FPGA card. As our investigated aggregation kernel is relatively small and has only a single state that has to be maintained across loop iterations, the necessary memory resource requirements are generally negligible.

```
1  template<typename T, size_t VL>
2  auto aggregate(T const * data, size_t elCount) {
3    T result = 0;
4    T tmp[VL] = {0};
5    for (; data!=data + elCount; data+=VL) {
6      #pragma unroll
7      for (auto i=0; i<VL; ++i) {
8        tmp[i] += data[i];
9      }
10   }
11   for (auto i=0; i<VL; ++i) {
12     result += tmp[i];
13   }
14   return result;
15 }
```

Figure 2: Aggregation - Using a data parallel approach.

2.5 Summary

From our above presented investigation, we may conclude that the DPC++ compiler is able to transform data parallel written code to efficiently executable FPGA designs in a resource-efficient way.

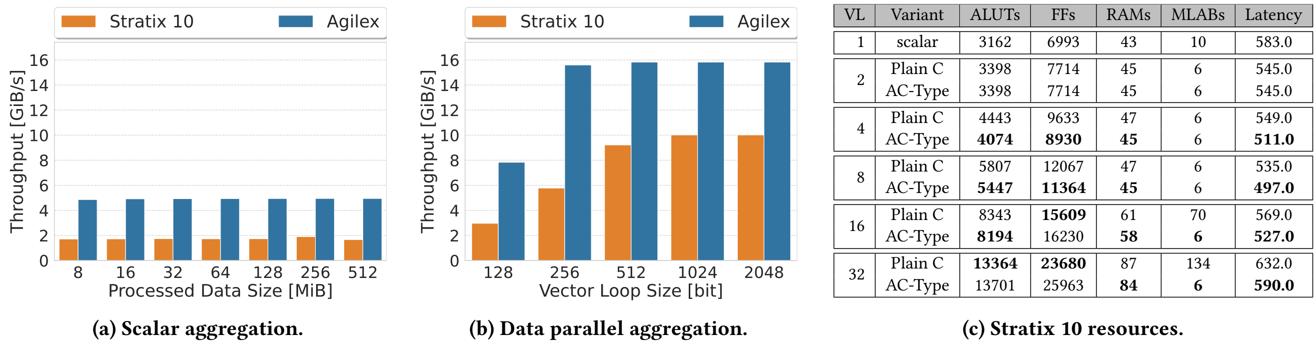
3 PROGRAMMING SIMD INSTRUCTION SET

To leverage an FPGA as a regular SIMD-based processing unit such as state-of-the-art CPU extensions (AVX or SVE) in C++, a specific SIMD instruction set has to be defined, consisting of (i) SIMD registers, (ii) instructions to load or store data, and (iii) actual data processing capabilities. The advantage of our approach is that the necessary FPGA SIMD instruction set can now be implemented in DPC++, as we will show in the remainder of this section.

3.1 Register Definition

To specify our FPGA SIMD instruction set, we first need definitions of the relevant data types, such as a SIMD register and a mask data type. As shown in Section 2, a regular array is sufficient to enable data parallel processing on the FPGA. The underlying data type for the array can be any arithmetic type, e. g., *uint32_t* or *double*. Our prototype uses a fixed-width SIMD register instead of a fixed-element count for better comparability with Intel® SIMD ISAs. Figure 5 summarizes our advanced DPC++ implementation of variable-length SIMD registers and mask types based on arrays. To calculate the number of elements within our array, we divide the target register size by the bit width of the data type (Figure 5 lines 6-7). With this helper function, we can define a type *register_t* as an alias for our SIMD register. To help the compiler recognize the data parallelism – also called vectorization — potential, we use the OpenCL-specific memory attribute *register*[3] (line 9). SYCL™ provides a variety of existing classes and types that can be used to improve the resulting design and, thus, the overall performance. As described in the previous section, *AC-Types* are one prime example. While it is conceivable to use the desired vector length (*VLb*) as the number of bits in an *AC-Type* to represent a SIMD register, actual calculations are predominantly executed on C-Type granularity for our use cases. An exception for the granularity is the category of scalar-integer instructions like binary logic operations, like *AND*,

[3]SYCL™, as an high-level abstraction layer on top of OpenCL defines the [[intel::fpga_register]] attribute. However, with the current compiler version, this attribute could not be used for type definitions.



(a) Scalar aggregation.

(b) Data parallel aggregation.

VL	Variant	ALUTs	FFs	RAMs	MLABs	Latency
1	scalar	3162	6993	43	10	583.0
2	Plain C	3398	7714	45	6	545.0
	AC-Type	3398	7714	45	6	545.0
4	Plain C	4443	9633	47	6	549.0
	AC-Type	**4074**	**8930**	**45**	6	**511.0**
8	Plain C	5807	12067	47	6	535.0
	AC-Type	**5447**	**11364**	**45**	6	**497.0**
16	Plain C	8343	**15609**	61	70	569.0
	AC-Type	**8194**	16230	**58**	**6**	**527.0**
32	Plain C	**13364**	**23680**	87	134	632.0
	AC-Type	13701	25963	**84**	**6**	**590.0**

(c) Stratix 10 resources.

Figure 3: FPGA programming in C++ – evaluation results.

```
1  template<typename T, size_t VL>
2  auto aggregate(T const * data, size_t elCount) {
3    using ac_t =
4      ac_int<sizeof(T)*CHAR_BIT, std::is_signed_v<T>>;
5    using register_t = std::array<ac_t, VL>;
6    register_t tmp;
7    T result = 0;
8    #pragma unroll
9    for (auto i=0; i<VL; ++i) {
10     tmp[i].template set_val<AC_VAL_0>();
11   }
12   for (; data!=data + elCount; data+=VL) {
13     #pragma unroll
14     for (auto i=0; i<VL; ++i) {
15       tmp[i] += data[i];
16     }
17   }
18   for (auto i=0; i<VL; ++i) {
19     result += tmp[i];
20   }
21   return result;
22 }
```

Figure 4: Aggregation - Using a data parallel approach with the specialized data type *ac_int*.

```
1  #include <sycl/ext/intel/ac_types/ac_int.hpp>
2  #include <array>
3  #include <climits>
4  template<typename T, size_t VLb>
5  struct oneAPIsimd {
6    constexpr static size_t VL() {
7      return (VLb/(sizeof(T)*CHAR_BIT));
8    }
9    using register_t =
10     __attribute__((register)) std::array<T, VL()>;
11   using si_register_t =
12     std::conditional_t<
13       std::is_integral_v<T>,
14       ac_int<VLb, false>, register_t
15     >;
16   using mask_t = ac_int<VL()), false>;
17 };
18 template<typename T, size_t VLb>
19 using reg_t = typename oneAPIsimd<T, VLb>::register_t;
20 template<typename T, size_t VLb>
21 using si_register_t =
22   typename oneAPIsimd<T, VLb>::si_reg_t;
23 #define INLINE __attribute__((always_inline)) inline
```

Figure 5: Variable-length (VL) SIMD registers and mask types.

or *XOR*. Those instructions can be carried out on the given data without regard to the underlying type since only the respective bits contribute to the result. In some scenarios, shift operations should be carried out holistically on all bits of the vector, resulting in complex logic if we use an array of smaller types. Consequently, we defined an *si_register_t* that aliases an *ac_int* with *VLb*[4] bits if the underlying type is integral, e. g., short or long. In any other case, the type is just an alias for the *register_t* since we argue that bit logic should not be directly executed on floating point values.

With the advent of Intel AVX-512 and ARM SVE, a specific SIMD mask type was introduced, representing the validity of every element within a given SIMD register. For Intel AVX-512, this mask type translates to an unsigned integer value that contains as many bits as there are values in a register. The n-th bit indicates the validity of the n-th element, e. g., if the third bit is set to one, the third element in the register is valid, and vice versa if the bit is set to zero. However, with arbitrary-length SIMD registers, corresponding register-mask types must also be of arbitrary length. Considering a register size of 1024 bits, holding 8-bit wide data elements, up to 128 values can be kept and processed within a single register. Consequently, a corresponding mask would exceed the maximum bit width of *unsigned long long* and, thus, must be represented as a vector of two 64-bit values. For such a scenario, *ac_int* seems to be a perfect match. Not only is the bit width programmatically defined, but accessing a specific bit or a slice of bits is supported and especially well translatable into a digital circuit.

3.2 Instruction Definition

Now that we have defined the necessary fundamental types to execute data parallel operations on vectors of data with an arbitrary degree of parallelism, we can define relevant instructions. For compatibility reasons, we followed the schema of the SIMD instruction set extensions from Intel and ARM and identified four instruction categories: (i) load/-store, (ii) element-wise, (iii) horizontal, and (iv) scalar-integer instructions.

[4]The *ac_int* is always unsigned to avoid a sign extension.

```
1 template<typename T, size_t VLb>
2 INLINE auto load(T const * memory) -> reg_t<T,VLb> {
3   reg_t<T,VLb> reg{};
4   #pragma unroll
5   for (size_t i=0; i< oneAPIsimd<T,VLb>::VL(); ++i) {
6     reg[i] = memory[i];
7   }
8   return reg;
9 }
10 template<typename T, size_t VLb>
11 INLINE void store(T * memory, reg_t<T,VLb> const & reg) {
12   #pragma unroll
13   for (size_t i=0; i<oneAPIsimd<T,VLb>::VL(); ++i) {
14     memory[i] = reg[i];
15   }
16 }
```

Figure 6: SIMD-register load-/store functions (VL).

Load-/Store Primitives. Contrary to a scalar processing, (i) SIMD registers must be explicitly populated with data elements from main-memory and (ii) data in SIMD registers must be explicitly written to the main memory. Appropriate instructions must be available within the SIMD instruction set. We have implemented two DPC++ functions assuming that the memory is directly accessible (for example, via USM) with the corresponding names for both aspects (see Figure 6). The implementation is straightforward and follows the general scheme of Figure 2. A scalar loop iterates over the memory or register and copies either data into a new register or manifests the content of a register in the memory (Figure 6 lines 5-7 and 13-15).

Element-wise Register Instructions. Generally, two classes of SIMD register processing instructions can be distinguished: (i) *element-wise* and (ii) *horizontal instructions*. Element-wise instructions are processing operations that are independently applied to every element within a given register representing an easy target for a data parallel, dependency-free processing. Such instructions also usually output a register with updated or manipulated elements. Well-known examples are calculation operations (e. g., *add*, *subtract*, *multiply*), and binary operations (e. g., *shifting*, *OR*, and *AND*). While unary instructions exist, e. g., counting the leading zeroes of every element, most instructions are binary. The latter ones take either a scalar value as the second operand, which is then used for executing the operation element-wise (e. g., *shifting* by a constant, or a *modulo* operation), or a complete register, where corresponding (i. e., same-position) elements from both registers are the input for the operation.

```
1 template<typename T, size_t VLb>
2 INLINE auto modulo(reg_t<T,VLb> const & a, T const m) {
3   reg_t<T,VLb> result{};
4   #pragma unroll
5   for (size_t i=0; i<oneAPIsimd<T,VLb>::VL(); ++i) {
6     result[i] = a[i] % m;
7   }
8   return result;
9 }
```

Figure 7: Element-wise modulo instruction.

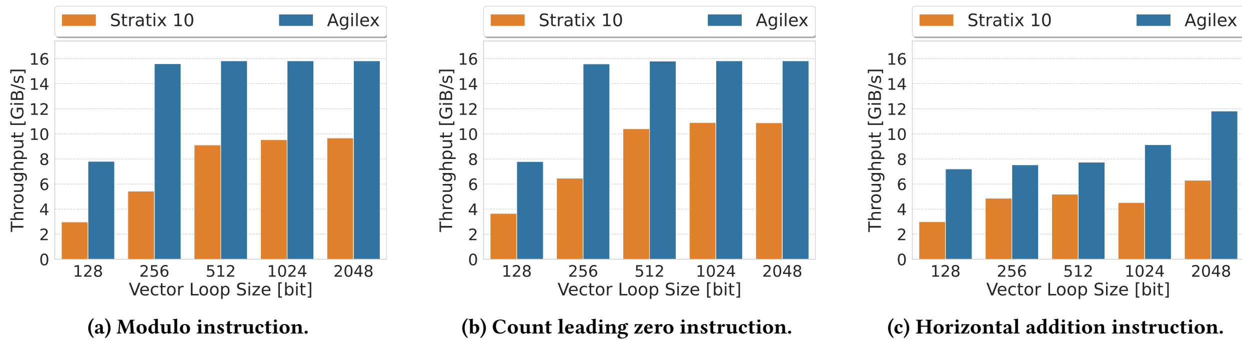
```
1 template<typename T, size_t VLb>
2 INLINE auto countLeadingZeroes(reg_t<T,VLb> const & a) {
3   reg_t<T,VLb> result{};
4   auto const bitCount = sizeof(T)*CHAR_BIT;
5   #pragma unroll
6   for (size_t i=0; i<oneAPIsimd<T,VLb>::VL(); ++i) {
7     T v = a[i];
8     T clz = bitCount+1;
9     T cmp = (T)1<<(bitCount-1);
10    #pragma unroll
11    for (size_t j=0; j<bitCount; ++j, cmp>>=1) {
12        clz = (((v&cmp) != 0) && (j<clz)) ? j : clz;
13    }
14    result[i] = clz;
15  }
16  return result;
17 }
```

Figure 8: Element-wise count leading zeroes instruction.

Figure 7 shows our DPC++ implementation of the well-known modulo function, i. e., it computes the remainder of the division of every element in the register with a constant scalar value (m). The code shows remarkable similarities to the load operation with the difference that the results of the individual lanes depend on the inputs. Another example of an element-wise instruction, namely a leading zero count, is depicted in Figure 8. For every element, the number of leading zeroes is counted and written to the corresponding position of the result. This is done by a nested loop that is carried out for every element in the register. The loop iterates over every bit position, starting at the most significant bit, and checks whether the bit is set to one. If so, the result value (*clz*) is set to the current position in the loop (line 12). As we only want to get the highest position of a bit set to 1, we check whether the current value of *clz* is greater than the current position. If so, we have yet to find a 1-bit. While this implementation looks rather cumbersome from a *CPU*-centric point of view, such a pattern can be highly beneficial for FPGAs if the loop can be fully rolled out and thus forms a pipeline.

Figures 9a and 9b show results of micro-benchmarks of both presented element-wise instructions. The micro-benchmarks are executed on Stratix 10 as well as on Agilex with different register sizes ranging from 128 bit up to 2048 bit. As base type, we used the plain C type `uint32_t` for the leading zero count, and `uint64_t` for the modulo benchmark. The following conclusion can be drawn from these results. First, the Agilex yields higher performance than the Stratix 10, which follows the same trend compared to CPUs: Algorithms can benefit freely from newer hardware. Second, as long as we do not reach interface speed, wider SIMD registers lead to higher throughput. Third, despite the fact that both instructions are considerably complex from a logical circuit perspective, even with only 2 elements being processed in parallel the measured throughput comes close to full interface speed on the Agilex. On the Stratix 10, only the leading zero count could achieve such a high throughput with at least 8 values being processed in parallel.

Horizontal Register Instructions. In contrast to element-wise instructions, horizontal register instructions do not treat the elements of a register independently. While these types of instructions



(a) Modulo instruction. (b) Count leading zero instruction. (c) Horizontal addition instruction.

Figure 9: Micro-benchmark results for selected SIMD instructions on FPGA.

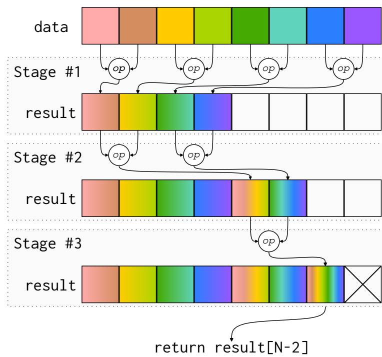


Figure 10: Implementation of reduction using an addition.

contradict the data-parallel, dependency-free processing paradigm, they represent an essential class of instructions. We can distinguish two different types of horizontal functions, namely (i) reduction and (ii) transformation instructions. The former reduces a register to a scalar value, e. g., aggregation-sum, and the latter transforms every element depending on the other elements within the register. In the following, we will examine how those instructions can be implemented on an FPGA through DPC++.

Reducing instructions basically carry out a register aggregation. All elements within a single register contribute to the overall result, e. g., horizontal addition or maximum. Consequently, data dependencies during execution can not be avoided. A naïve implementation would initialize a result (r) and loop over all elements ($[0, n)$) within the register (v), applying the operation (e. g., addition) with the result and the element $r = v[i] + r$. If the operation is not commutative, the loop can not be parallelized, and every stage in the pipeline must wait for the preceding stages to finish. While we cannot entirely eliminate the existing data dependencies, we are able to reduce the impact drastically by using a divide-and-conquer approach. Instead of maintaining a single result for the whole loop, we create $n-1$ intermediate results (t). In the first pipeline stage, we calculate the $n/2$ intermediate results as the sum of disjoint pairs of elements from our input as illustrated in Figure 10. In the next stage, we repeat the procedure but use the results from the previous state as input. We repeat this until the last two intermediate results are used to calculate the final result. In total, we will execute $ld(n)$ stages that can be perfectly pipelined since the result of every stage serves as the input of every subsequent stage. This general scheme is also called an adder tree. Figure 9c depicts the throughput for this approach on both tested FPGA platforms processing 500 MiB synthetically generated 64-bit integer data. As the results clearly show, we can reach a notable performance yet do not reach interface speed. This behavior is expected since our benchmark calculates a running aggregate for a chunk of data. If the compiler does not detect the pipeline friendliness of the given instruction, the execution path must be stalled whenever a chunk of data is processed. However, such horizontal operations occur only sporadically and thus should not significantly impact the overall performance of a given algorithm.

A further prominent example of a *horizontal transformation instruction* is the in-register conflict detection, that was introduced by Intel® with AVX512. It became crucial through the newly added capability of random access store operations (i. e., *scatter* instruction to store register elements back to multiple memory locations at once). Specifically, an offset register that is used by a *scatter* operation to write register elements to random memory locations could contain equal values. This would lead to multiple elements being written to the same memory address and thus, multiple values could be overwritten with only one value being actually materialized. Conflict detection ensures the uniqueness of the values within any register – i. e., our offset register in this example – and, thus, eliminate potential write collisions. Each element in the register is compared with all its predecessors. In the case of equality, the bit is set at the position corresponding to the position of the duplicate. Unique elements or elements without an equal predecessor will have the value 0 in the result register after performing the conflict detection. To mimic this intrinsic on an FPGA, a nested loop would be sufficient to achieve this behavior with simple C++ means. However, the compiler failed to roll it out properly for the FPGA in our experiments. For this reason, we used template metaprogramming to force the compiler to unroll the loop. Processing for a specific lane is done using the function *conflict_single* (Figure 11a lines 5-9). Each element at position $[0, Idx)$ is compared to that at position *Idx* (Figure 11a line 6). If they are equal, the result bit at the respective position is set to 1 shifted by *Idx*. This "inner loop" is called by *conflict_impl* for each element at position $[1, max(Idx)]$ (Figure 11b lines 7-8). Finally, the proper *conflict* instruction just invokes *conflict_impl*.

Since all loops have been unrolled by template metaprogramming, we only need to create the result register and pass it along with the data register into the appropriate function. The depicted

```
1 template<typename T, size_t VLb, size_t Idx, size_t... Idxs>
2 INLINE T conflict_single(
3   reg_t<T,VLb> const & data, std::index_sequence<Idxs...>
4 ) {
5   return ((T)0 |...| (
6     data[Idx+1] == data[Idxs]
7     ? ((T)1 << (T)Idxs)
8     : (T)0)
9   );
10 }
```

(a) Single comparison using fold expressions.

```
1 template<typename T, size_t VLb, size_t... Idxs>
2 INLINE void conflict_impl(
3   reg_t<T,VLb> & result, reg_t<T,VLb> const & data,
        std::index_sequence<Idxs...>
4 ) {
5   (
6     (result[Idxs+1] =
7       conflict_single<T,VLb,Idxs>(data,
8         std::make_index_sequence<Idxs+1>{})
9     ), ...
10   );
11 }
```

(b) Unrolled comparison of an element with all its predecessors.

Figure 11: Horizontal conflict detection.

```
1 template<typename T, size_t VLb, size_t Out, size_t In=0>
2 INLINE si_reg_t<T,VLb> append_si(
3   si_reg_t<T, VLb> source, si_reg_t<T, VLb> data
4 ) {
5   constexpr size_t data_read_count{
6     ((VLb-Out) <= (VLb-In))
7     ? (VLb-Out)
8     : (VLb-In)
9   };
10   using slice_t = si_reg_t<T, data_read_count>;
11   slice_t slice = data.template slc<data_read_count>(In);
12   source.set_slc(Out, slice);
13   return source;
14 }
```

Figure 12: Filling up an *ac_int*.

code will trigger the compiler to generate an unrolled version. Yet, the current version of the DPC++ builds suboptimal results.

We manually unrolled the implementation for 64-bit integers for our benchmark using 512-bit wide FPGA-SIMD registers. Nevertheless, the discussed conflict detection instruction is a corner case for horizontal operations since not only do all elements within a SIMD register contribute to a single result, but the value of all values up to position n are relevant for finding the conflicting elements at position $n + 1$. As we already showed that a horizontal addition with its inherent data dependencies does not reach interface speed, it comes as no surprise that the conflict detection does not perform well on both cards, reaching a maximum of 3.12GiB/s on the Agilex, 1.75GiB/s on the Stratix 10 respectively. To sum up, while it is generally possible to realize horizontal operations on an FPGA, the investigated techniques do not use the currently existing hardware and its associated compiler framework to the maximum extent.

Scalar-Integer Instructions. In some situations, considering a register as a single value is advantageous to execute specific instructions, such as logical or shift operations. The *AC-Types* classes provided by SYCL™ overload all relevant arithmetic and logical operations and can, therefore, be called directly with the corresponding types. Slicing is a unique feature of the AC integer and fixed-float types. An arbitrary (compile-time constant) number (N) of neighboring bits can be read from a start index and written to a new N-bit integer or fixed-float type. Correspondingly, such a type can be inserted into another type at any position. For example, this capability enables inserting new values into partially filled registers or overwriting them. A possible implementation for such an insertion is shown in Figure 12. First, the number of bits to be read is calculated from *data* (see lines 5-9). If fewer or precisely as many bits are written to *source* as can be read from *data* based on *In*, *data_read_count* is set to the number of bits to be overwritten in *source*. Otherwise, *data_read_count* is set to the number of remaining bits in *data*. Now that the number of bits to be read from *data* is known, we define a *ac_int* with exactly this size (see line 10). We then create a slice of *data* starting from *In* (line 11) and write the corresponding bits to *source*, starting from bit position *Out* (line 12).

3.3 Comparing with RTL kernels

Writing ordinary C/C++-Code to describe an algorithm and letting a specialized compiler framework do the heavy lifting of translating the code into a digital circuit (and fitting the design into a given FPGA hardware) is desirable for various reasons. On the one hand, such a workflow can drastically decrease the development time. On the other hand, a programmer can benefit from the existing type system and features like template-based code generation. However, the question of how well the generated result performs compared to a hand-tuned VHDL code remains. We took an optimized Leading-Zero-Count VHDL-code from the industry to gain an impression and compared the throughput with our solution from Figure 8. Through the oneAPI toolchain, we can associate a VHDL kernel with a C function name. Consequently, we can call a VHDL kernel from C/C++ and thus use it directly within our framework. To assess our generic implementation, we compared it against the optimized VHDL version. Table 1 depicts the differences in the achieved throughput using the generic version as the baseline. As we compare the throughput, positive differences indicate that the VHDL version was faster than the generic variant and vice versa for negative differences. Surprisingly, the differences between the investigated variants of the leading zero count are negligible, ranging from −0.551% to +1.3%. Taking into consideration that the generic version works for any integral datatype and SIMD register size, the results speak volumes about the potential of the setup.

Table 1: Throughput differences of industry-strength Verilog CLZ kernel compared to a generic oneAPI Implementation.

	Vector Loop Size [bit]				
	128	256	512	1024	2048
Stratix 10	+ 1.3%	-0.738%	-0.136%	-0.551%	-0.0092%
Agilex	+0.118%	+0.0019%	+0.019%	+0.0025%	±0.0%

4 USE CASE STUDIES

With our approach, we are able to fully implement a SIMD instruction set for an FPGA in DPC++ that behaves similarly to the state-of-the-art SIMD instruction set extensions of modern CPUs. Based on this, any SIMD extension behavior such as Intel®'s AVX512 or ARM Neon can now be made available on the FPGA and therefore any existing SIMD application code can be seamlessly ported to FPGAs. To reduce the porting effort, SIMD abstraction libraries such as TVL [26] should be used, as we have already envisioned in [9]. Another nice and cost-free side effect of our approach is that a single-source SIMDified code can now be concurrently executed on both CPU and FPGA without having to consider FPGAs in isolation (co-processing with one single source code for an algorithm).

4.1 FilterCount

To highlight the applicability of our approach, we implemented a SIMDified `filter-count` operation counting the number of values in a given range and dispatched this operation concurrently to the host CPU and the Agilex FPGA. Figure 13 shows the achieved bandwidth on both the host and the FPGA with USM-allocated data on the host [9]. The host code is executed using AVX512 and the FPGA is parameterized with a reasonable SIMD register size. As expected, more concurrently working threads on the host memory lead to a decreased per-thread bandwidth. However, the simultaneously running FPGA is able to constantly achieve its peak performance.

4.2 Binary Packing

A common technique to tackle the limitations of the memory subsystem in database systems relies on compressing base- and even intermediate data [1, 8]. One popular example of such compression algorithms is *Binary Packing (BP)* [17, 24]. The fundamental idea is to divide data into fixed-sized blocks, determine the maximum amount of bits to represent every single value within the block (1st stage), and encode all values with this bit width (2nd stage).

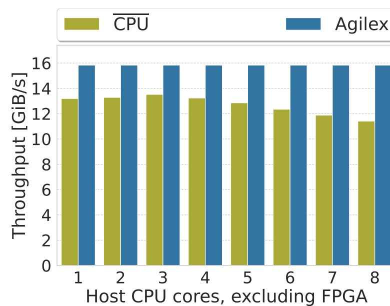


Figure 13: Co-processing bandwidth for `filter-count`.

```
1  auto bp_compress_block_vlb512_ui32_16b(
2    uint32_t * out, uint32_t const * data
3  ) {
4    using T = uint32_t; constexpr size_t VLb = {512};
5    auto i = 0, o = 0;
6    #pragma unroll
7    for (; i < VLb; i += 32, o += 16) {
8      auto decmpr_1 = load<T, VLb>(&data[i]);
9      auto decmpr_2 = load<T, VLb>(&data[i+16]);
10     auto uncompr_sl = shift_left<T, VLb>(decmpr_2, 16);
11     auto cmpr = binary_or<T, VLb>(decmpr_1, uncompr_sl);
12     store<T, VLb>(&out[o], cmpr);
13   }
14 }
15 auto bp_decompress_block_vlbVLb_ui32_16b(
16   uint32_t * out, uint32_t const * data
17 ) {
18   using T = uint32_t; constexpr size_t VLb = {512};
19   auto i = 0, o = 0;
20   auto mask = set1<T, VLb>(0xffff);
21   #pragma unroll
22   for (auto i = 0; i < VLb; i += 16, o += 32) {
23     auto cmpr = load<T, VLb>(&data[i]);
24     auto decmpr_1 = binary_and<T, VLb>(cmpr, mask);
25     store<T, VLb>(&out[o], decmpr_1);
26     auto decmpr_2 = shift_right<T, VLb>(cmpr, 16);
27     store<T, VLb>(&out[o+16], decmpr_2);
28   }
29 }
```

Figure 14: Implementation of Binary Packing (Type = uint32_t, Blocksize = 512, Effective Bitwidth = 16).

Without further encoding, this algorithm can benefit unsigned integer values (without the sign bit in signed integers and floating point types). The performance of *Binary Packing* mainly depends on the data properties [7, 17]. Higher effective bit counts lead to worse compression rates and thus decrease the positive effect on the memory bus. However, the compression as well as decompression rountines can be SIMDified very well, as shown in Listing 15 [7, 17].

For example, with an effective bitwidth of 16-bit, an unsigned int data type, and a blocksize of 512 elements, the compression algorithm consists of a sequence of loading adjacent data into two SIMD registers (lines 3 and 4), shifting the second one by 16 to the left (line 5), applying a binary *OR* on the first register and the shifted register (line 6), and storing the result into memory (line 7). The corresponding decompression algorithm reverses the compression by loading a single SIMD register (line 20) and masks out the upper 16-bit of every 32-bit lane to retrieve the first chunk of uncompressed data (line 21) and, consequently, shifts the loaded register right by 16 to restore the second chunk (line 23). As the number of bits within the base type is a multiple of the compressed bitwidth, everything adds up correctly. If the bitwidth of the underlying type isn't a multiple of the compressed bitwidth, an additional step has to be added for the situation when a carry exists.

A significant drawback of this algorithm lies in its sensitivity to (only sparsely occurring) bit-width deviations, as a single value within a block may contribute disproportionately to the block's encoded bit-width [10]. For a uniform value distribution, this characteristic translates to lower compression rates for bigger block

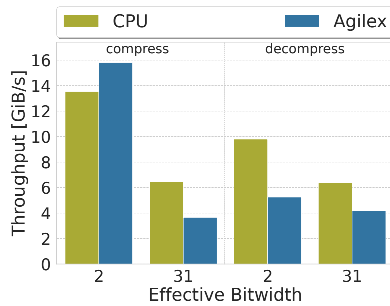


Figure 15: State-of-the-art *Binary Packing* (constant bitwidth with 512-bit wide registers) on CPU (AVX512) and FPGA.

sizes. To minimize the SIMD load- and store operations by constantly processing a complete SIMD register independent of the effective bitwidth, the blocksize is set to the number of bits within a SIMD register. Consequently, wider SIMD registers lead to bigger block sizes and thus may diminish the positive effects of higher data parallelism. A variation of *Binary Packing* was introduced to address this issue, which changes the internal data format and thus reduces the block size drastically [10]. However, in order to properly function, the adapted *Binary Packing* heavily relies on (fast) SIMD-random access [10].

For our use-case study, we implemented the necessary SIMD instructions for FPGA. We ran a benchmark with bit widths of 2 (an example for an integer denominator) and 31 (as an example for a more complex situation). As *Binary Packing* is a compression algorithm, the amount of bytes that must be read differs from the number of bytes that must be written. To calculate the reported throughput, we divide the amount of uncompressed data by the required time to execute the compression or decompression. Furthermore, as we treat the FPGA as a co-processing device, we placed the data that should be read within the USM-allocated data on the host and write the result directly to the device. All experiments were executed with 512MiB of integer data with a constant maximum bitwidth, i. e., if the effective bitwidth equals 2, the uncompressed data consists of values in the range of $[0, 3]$, for 31 bit, the range is $[0, (1 << 32) - 1]$. The results are shown in Figure 15. Working with smaller bit widths leads to an increased throughput. This is expected since, on the one hand, the amount of data that has to be written for the compression part and read for the decompression part, respectively, becomes lower with a smaller effective bitwidth. On the other hand, 2 is an integer denominator of 32, and consequently, no expensive overflow handling needs to take place. While the compression strategy on the FPGA with 2 bits outperforms its CPU counterpart, traditional *Binary Packing* falls short for all other combinations compared to the CPU. To sum up, we could show that, in general, we can successfully port a CPU-SIMD algorithm to FPGA with negligible effort. However, if we adopt the general processing scheme to utilize the opportunities of FPGAs to the maximum extent, we may benefit from optimization potential.

5 CUSTOM SIMD INSTRUCTIONS

The essence of *Binary Packing* as described above relies on packing a fixed-sized number of elements densely together on the bit level. Setting specific bits in a value or a stream of bits is necessary.

```
1 auto bp_fpga_compress_block32_vlb512_ui32_16b(
2   uint32_t * out, uint32_t const * data
3 ) {
4   using T = uint32_t; constexpr size_t VLb = {512};
5   auto cmpr_1 = packed_load_si<T, VLb, 31>(data);
6   auto cmpr_2 = packed_load_si<T, VLb, 31>(&data[16]);
7   auto result_1 =
8     append_si<T, VLb, 496, 0>(cmpr_1, cmpr_2);
9   store_si<T, VLb>(out, result_1);
10  auto result_2 =
11    shift_right_si<T, VLb, 16, 16, 496>(cmpr_2);
12  packed_store_si<T, VLb, 480>(&out[16], result_2);
13 }
14 auto bp_fpga_decompress_block32_vlb512_ui32_16b(
15   uint32_t * out, uint32_t const * data
16 ) {
17   using T = uint32_t; constexpr size_t VLb = {512};
18   auto cmpr_1 = load_si<T, VLb, 31>(data);
19   auto decmpr_1 = unpack_si<T, VLb, 31, 0>(cmpr_1);
20   store_si<T, VLb>(out, decmpr_1);
21   auto decmpr_2_0 = unpack_si<T, VLb, 31, 496>(cmpr_1);
22   auto cmpr_2 = load_si<T, VLb>(&data[16]);
23   auto decmpr_2_1 =
24     unpack_merge_si<T, 31, 16>(decmpr_2_0, cmpr_2);
25   store_si<T, VLb>(decmpr_2_1);
26 }
```

Figure 16: Adjusted implementation of Binary Packing (Type = uint32_t, Blocksize = 512, Effective Bitwidth = 16) using custom SIMD instructions.

While this is not supported directly by general-purpose CPUs like x86-64, it is not a concern when designing a custom circuit on an FPGA. A Flip-Flop represents a single bit and can be directly set to 1 or 0. Given this peculiarity, we can rewrite the state-of-the-art *Binary Packing* as shown in Figure 21. To demonstrate the flexibility of our approach, we show a 32-element block-wise packing of 32-bit values with 31-bits per value. Thus, when packing $16 \cdot 31$ bits together, the underlying scalar integer with a size of 512-bit has 16 bits left unused. This remainder can be filled with the lower 16 bits from the 17th data element before writing 512 bits to memory. When processing the remaining 16 elements within the block, we must omit the first 16 bits of the 17th element. For the decompression, the existing boundary overflow must be considered. In the following, we will describe the used custom SIMD instructions one by one except for *load_si* and *store_si* as they behave similarly to the load/store instructions shown in Figure 6.

Packed Load. The *packed_load_si* instruction iterates over a constant number of bits (*VLb*) of a given type (*T*) of data (see Figure 17). At first (lines 3 and 4), a *VLb*-sized *ac_int* is created and initialized with 0 (following the official documentation, this is highly recommended to avoid undetermined behavior). From every value, the least significant *Bitwidth* bits are sliced (line 14) and inserted into the result *ac_int* (lines 14 and 15). The insertion position is calculated by the current position of the data element multiplied by the bandwidth, leading to densely packed values inside the *ac_int*.

Unpack. The reverse operation of *packed_load_si* is *unpack_si* (see Figure 18). The function gets an *ac_int* with densely packed bits

```
1 template<typename T, size_t VLb, size_t Bitwidth>
2 INLINE si_reg_t<T,VLb> packed_load_si(T const * memory) {
3   constexpr unsigned ElementSizeBit = sizeof(T)*CHAR_BIT;
4   static_assert(Bitwidth <= ElementSizeBit,
5     "Bitwidth > element size in bits.");
6   si_reg_t<T, VLb> result;
7   result.template set_val<AC_VAL_0>();
8   using ElementT = ac_int<CHAR_BIT*sizeof(T), false>;
9   using SliceT = ac_int<Bitwdith, false>;
10  #pragma unroll
11  for (
12    auto idx = 0; idx < oneAPIsimd<T,VLb>::VL(); ++idx
13  ) {
14    ElementT element = memory[idx];
15    SliceT slice = element.template slc<Bitwidth>(0);
16    result.set_slc((unsigned)(idx*Bitwidth), slice);
17  }
18  return result;
19 }
```

Figure 17: Implementation of a packed load (packing N bits per value into an *ac_int*).

as input and returns an *ac_int* containing the restored values. A single *ac_int* can contain the packed values of multiple *VLb*-sized uncompressed data, the bitwidth and starting offset of the current relevant bits are provided as a non-type template argument. Based on the compile-time constants, the number of fully encoded values (lines 4-9) and the amount of partially encoded bits from an overflow value (lines 10-14) are computed at compile time. The remainder of the function works like a reversed *packed load*. For every fully encoded value, *Bitwidth* bits are sliced from the input value and inserted into a resulting *VLb*-sized value at the corresponding bit position, a multiple of the element size in bits (lines 17-26). If an overflow occurs in the packing stage, the partial bits from that value are inserted at the corresponding position (lines 27-34), and the result is returned. The remainder of bits within a subsequent *ac_int* is merged and unpacked through *unpack_merge_si* (see Figure 19). First, the number of necessary bits from the overflowing value is calculated (line 5) and inserted into the *source*, containing the unpacked values from the previous stage (lines 6-10). If *source* is not filled yet, the remaining bits are set according to the regular *unpack_si* with the exception that it is assumed that no carry exists in the remainder (see lines 11-22).

Right Shift. While *AC-Types* support logical left- and right shifts that, if the shift-by value is a compile-time constant, translate to plain routing without additional logic, we implemented a specialized right shift that can only be applied to a specific part of the whole *ac_int* to demonstrate its flexibility. The code is shown in Figure 20. Initially, the positions of the relevant bits (sub-value) are normalized. Suppose the sub-value starts at a position smaller than the value it should be shifted to the right. In that case, the bits at the position up to the difference between the *ShiftValue* and *ReadOffset* will not contribute to the overall result. They thus can be omitted by increasing the *ReadOffset* by the delta (lines 7-10). Such a change in the *ReadOffset* decreases the amount of *EffectiveBits* (lines 11-15). With the help of the normalized *ReadOffset* and the normalized

```
1 template<typename T, size_t VLb,
2   unsigned Bitwidth, unsigned ReadOffset>
3 INLINE si_reg_t<T,VLb> unpack_si(si_reg_t<T,VLb> data) {
4   constexpr unsigned FullReadCount = {
5     (VLb - ReadOffset) >=
6       (Bitwdith * (VLb/(CHAR_BIT*sizeof(T))))
7     ? VLb/(CHAR_BIT*sizeof(T))
8     : (VLb - ReadOffset)/Bitwidth
9   };
10  constexpr unsigned PartialReadBits = {
11    (VLb - ReadOffset) >=
12      (Bitwdith * (VLb/(CHAR_BIT*sizeof(T))))
13    ? 0 : VLb - (FullReadCount * Bitwidth + ReadOffset)
14  };
15  si_reg_t result;
16  result.template set_val<AC_VAL_0>();
17  if constexpr (FullReadCount > 0) {
18    #pragma unroll
19    for (auto idx = 0; idx < FullReadCount; ++idx) {
20      ac_int<Bitwidth, false> element =
21        data.template
22          slc<Bitwidth>((unsigned)(idx*Bitwidth +
                ReadOffset));
23      result.set_slc(
24        (unsigned)(idx*CHAR_BIT*sizeof(T)), element);
25    }
26  }
27  if constexpr (PartialReadBits > 0) {
28    ac_int<PartialReadBits, valse> element =
29      data.template
30        slc<PartialReadBits>(
31          (unsigned)FullReadCount*Bitwidth + ReadOffset);
32    result.set_slc(
33      (unsigned)(FullReadCount*CHAR_BIT*sizeof(T)),
           element);
34  }
35  return result;
36 }
```

Figure 18: Implementation of unpacking values from an *ac_int*.

EffectiveBits, a corresponding slice can be used to fill the resulting value (lines 18-22).

Evaluation

We evaluated our adjusted *Binary Packing* using custom SIMD instructions on the two given platforms (see Figure 21). Following the findings of the state-of-the-art experiment (see Figure15), the effective bitwidth was set to 2 bits. The introduced custom instructions not only allow us to sustain the existing data layout but also allow us to vary the block size. As long as the block size is a multiple of the size in bits of the underlying base type (32 in our case), the algorithm can process an arbitrary number of elements. Those smaller block sizes can dramatically increase the compression rate for real-life data [10]. Smaller block sizes have another advantage over bigger ones. The smaller the block, the fewer instructions per block must be executed and, consequently, the smaller the integrated circuit. This benefits the possible f_{max} that will increase

```
1  template<typename T, size_t VLb, unsigned Bitwidth,
       unsigned WriteOffset>
2  INLINE si_reg_t<T,VLb> unpack_merge_si(
3    si_reg_t<T,VLb> source, si_reg_t<T,VLb> source data
4  ) {
5    constexpr unsigned CarryBitsCount = {WriteOffset %
         (CHAR_BIT*sizeof(T))};
6    if constexpr (CarryBitsCount != 0) {
7      ac_int<CarryBitsCount, false> carry_element =
8        data.template slc<CarryBitsCount>((unsigned)0);
9      source.set_slc((unsigned)(WriteOffset),
           carry_element);
10   }
11   constexpr unsigned ReadCount = {
12     (VLb - (WriteOffset + CarryBitsCount)) /
           (CHAR_BIT*sizeof(T))
13   };
14   constexpr unsigned NormalizedWriteOffset = {WriteOffset
         + CarryBitsCount};
15   if constexpr (ReadCount > 0) {
16     #pragma unroll
17     for (auto idx = 0; idx < ReadCount; ++idx) {
18       ac_int<Bitwidth, false> element =
19         data.template slc<Bitwidth>((unsigned)(idx *
               Bitwidth + CarryBitsCount));
20       source.set_slc((unsigned)(idx * CHAR_BIT*sizeof(T)
             + NormalizedWriteOffset), element);
21     }
22   }
23   return source;
24 }
```

Figure 19: Implementation of a combined unpack and merge instruction on FPGA.

the overall performance. Surprisingly, the compression throughput of 512-element blocks on both cards was on par with its 32 elements counterpart. We argue that the compression's pipeline-friendly (stall-free) processing scheme absorbs the adverse effects of an increased circuit size. Until the 512 bits are entirely filled, no store operation has to be executed (that may be a pipeline breaker if the compiler assumes a data dependency of the store location). In contrast, multiple stores are executed per load instruction when decompressing, significantly harming the overall throughput for big blocks. Furthermore, for all remaining combinations, our implementation could reach interface speed. The presented results show that FPGAs are an excellent match for traditional SIMD processing and open up a whole new design space with exciting speedup potential since it enables data parallel hardware-software co-design.

6 DPC++ BEST PRACTICES

Through our work we found several pitfalls that a C++-programmer may stumble upon. As described above, loops are a crucial and basic building block of nearly every instruction/algorithm. A key factor for performant code is to have a compile-time constant amount of iterations, which allows the compiler to unroll them properly. However, some specific algorithms like, e. g., hashing with bucket chaining, require loops with dynamic iteration count (`while(!found) {...}`) to recompute data placement for overflowing buckets. Such

```
1  template<typename T, size_t VLb,
2    unsigned ShiftValue, unsigned ReadOffset,
3    unsigned EffectiveBits>
4  INLINE si_reg_t<T,VLb> shift_right_si(
5    si_reg_t<T,VLb> data
6  ) {
7    constexpr unsigned NormalizedReadOffset = {
8      (ReadOffset < ShiftValue)
9      ? ReadOffset + (ShiftValue - ReadOffset) : ReadOffset
10   };
11   constexpr unsigned NormalizedEffectiveBits = {
12     (ReadOffset != NormalizedReadOffset)
13     ? EffectiveBits - (ShiftValue - ReadOffset)
14     : EffectiveBits
15   };
16   constexpr unsigned WriteOffset =
17     {NormalizedReadOffset - ShiftValue};
18   ac_int<NormalizedEffectiveBits, false> slice =
19     data.template
           slc<NormalizedEffectiveBits>(NormalizedReadOffset);
20   si_reg_t result;
21   result.template set_val<AC_VAL_0>();
22   result.set_slc(WriteOffset, slice);
23   return result;
24 }
```

Figure 20: Implementation of a scalar-integer logical right-shift across C type bit boundaries on FPGA.

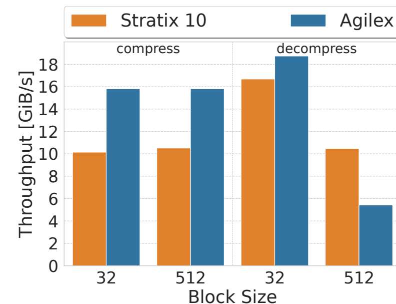


Figure 21: Specialized *Binary Packing* of data with effective bitwidth of 2 using different block sizes on FPGA.

algorithms should be refactored to avoid such loops as much as possible through, e. g., a fixed number of tries and dedicated overflow buckets.

Following that, we observed that determinism is another important factor. That is, branching or conditional access is generally discouraged. Consider an algorithm that conditionally accesses a value: `for (..) {idx = data[i]*3; myVal += temp[idx]};` The access to `temp` is dependent on the value calculated from `data`, which introduces a data dependency. This in turn leads to stalling of the pipeline, since the computation has to be carried out prior to the memory access. Thus, this is detrimental for the overall performance. Also in that context, the tail latency of an implementation is based off its longest path. That is, if an algorithm has several nested `if` statements, the FPGA image will create multiple paths for all possible combinations. Yet, the result of every shorter path has to wait for the longest path to be finished, e. g., to meet global pipeline timing requirements, which again means stalling the pipeline.

As a combination of the previous two factors, determinism and data dependencies, we saw that the explicit usage of `#pragma unroll` is crucial for the compiler to understand that this is an actually unrollable loop, whose iterations might be executed on multiple elements in parallel. Further, the generated hardware synthesis report notified us occasionally that a data dependency was detected (and thus pipeline stalling occurs) because we accessed adjacent array indices, even if there was actually no dependent read or write. The usage of `#pragma ivdep` can help alleviate this issue, as it tells the compiler to ignore assumed data dependencies[5].

Further, issues arise when working with commonly used C++ templates, which are generally employed to reduce the size of the code base. Heavy template usage and their eventual instantiation for many different types can lead to the compiler using more logic array blocks (MLAB) and thus an image that is too large to be fit onto the FPGA. Secondly, template recursion is often used to unroll recursive functions at compile time and, thus, avoid costly function calls during runtime. Our development cycle showed, that in its current state, the DPC++ compiler generates well-performing FPGA images from compile-time constant template recursion unreliably[6].

The translation process of a C++ program to the final block design, carried out by the quartus fitter, was found to be not completely robust. Based on a random seed, the fitting process starts to place circuits on the virtual board and tries to find the locally optimal placement. However, there are constraints to be considered, e.g. distances between circuits, local and global timing limits, or just plain resource utilization. Depending on the starting seed, a fitting process may or may not succeed. In the latter case, the process could fail after several days with the error "The fitter failed to successfully route the design." or the execution crashes.

The very long fitting times also limit the use case scenarios for this approach. A priori known variants can be easily compiled for operator-at-a-time or vector-at-a-time processing styles during system build time. However, generating an operator pipeline for tuple-at-a-time-based processing via just-in-time compilation is, from our point of view, simply not feasible.

Lastly, the fitting process also influences a key indicator for the performance of the FPGA-based code: the maximum frequency $f_{\max}$. Multiple layouts of the same algorithm stemming from different seeds can exhibit a difference in the resulting $f_{\max}$. The key question resides if there exists a generally well-performing seed for the fitting process or if it is at all possible to somehow predict potential frequencies, based on the required resources or algorithmic properties. This effect becomes even more severe if multiple algorithmic circuits are placed within the same image. Because the FPGA only features a single global clock, all circuits are limited to the lowest $f_{\max}$ of all used layouts. This requires the programmer to leverage a multistep process when synthesizing the FPGA code: First, the estimated $f_{\max}$ has to be collected for all algorithms and second, algorithms with a low deviation in their $f_{\max}$ are to be grouped together, while still adhering to the overall available resources like MLABs or ALUTs.

[5]Use with care, since even actual data dependencies will be ignored as well.

[6]When we substituted our templated code with a manually unrolled version, i. e., the anticipated compiler output, we could observe a notable increase in the achieved throughput

7 RELATED WORK

Leveraging FPGAs is commonly considered as a non-trivial task, since it traditionally involves maintaining a second code base just for the FPGA accelerator code. However, its general potential for empowering modern database systems has already been recognized [20]. In this paper, we focus on Intel® oneAPI and its DPC++ compiler to generate FPGA-specific images for data-parallel code. Generally, OpenCL™ is an alternative dialect to submit C-style code to CPUs, GPUs, or FPGAs. The utilization of OpenCL™ for FPGAs has been already investigated [16, 18, 19]. For example, [18] achieved a 4X better performance for a specific sort-merge algorithm implemented in HDL than in OpenCL™. However, we are not aware of any work investigating OpenCL for general-purpose data-parallel code for FPGA including the definition of a SIMD instruction set. We decided to use DPC++, since USM offers a shared memory programming model that significantly improves upon the shared virtual memory (SVM) model defined in OpenCL™ [4].

SIMD is a well-established and crucial paradigm for in-memory database query processing, e. g., as investigated in [21]. Combining SIMD and FPGAs is an age-old idea, which was already considered in 1994 [5]. In their work, the authors elaborate on the differences and similarities between SIMD-processors and FPGAs and further propose a hybrid model called Dynamically Programmable Gate Array (DPGAs). With the continuous advancement of hardware capabilities, we can now investigate actual implementations of SIMDified algorithms on FPGA hardware in our paper. Moreover, the general idea of building a dedicated ISA on FPGAs is not entirely new, as it has already been investiged by [3]. The authors leveraged a Tensilica LX4 prototyping core to implement set instructions for database operators. Our work builds upon this idea by further investigating the application of general SIMD processing methods and how to avoid performance pitfalls.

In FastLanes, the customizability of FPGAs is used to create SIMD-like registers of an arbitrary size, e. g., 1024 bit in this case [2]. This work considers load/store operations as well as bitwise manipulations like shifts, (X)OR, addition and set operations. While these basic instructions are suitable to represent certain algorithms, we consider even more complex operations to allow for the portability of complete algorithms.

8 CONCLUSION

This paper explored the possibilities of porting a SIMD-like instruction set, inspired by modern CPUs, to FPGA boards. We leveraged oneAPI and the modern language extension DPC++ from Intel® to implement and synthesize our intrinsic-like draft to two contemporary FPGA boards. Our microbenchmarks show, that both naïve and sophisticated methods achieve runnable FPGA code without the necessity of using low-level HDLs, but with varying performance.

ACKNOWLEDGMENTS

This work was partly funded by (1) the European Union's Horizon 2020 research and innovation program under grant agreement No 957407 (DAPHNE) and (2) the German Research Foundation (DFG) via a Reinhart Koselleck-Project (LE-1416/28-1).

REFERENCES

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 671–682. ACM, 2006.

[2] A. Afroozeh and P. A. Boncz. The fastlanes compression layout: Decoding >100 billion integers per second with scalar code. *Proc. VLDB Endow.*, 16(9):2132–2144, 2023.

[3] O. Arnold, S. Haas, G. P. Fettweis, B. Schlegel, T. Kissinger, and W. Lehner. An application-specific instruction set for accelerating set-oriented database primitives. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 767–778. ACM, 2014.

[4] B. Ashbaugh, A. Bader, J. C. Brodman, J. R. Hammond, M. Kinsner, S. J. Pennycook, R. Schulz, and J. Sewall. Data parallel C++: enhancing SYCL through extensions for productivity and performance. In *IWOCL '20: International Workshop on OpenCL, Virtual Event / Munich, Germany, April 27-29, 2020*, pages 7:1–7:2. ACM, 2020.

[5] M. Bolotski et al. Unifying fpgas and simd arrays. In *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, 1994.

[6] X. Chen, Y. Chen, R. Bajaj, J. He, B. He, W. Wong, and D. Chen. Is FPGA useful for hash joins? In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

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

[8] P. Damme, A. Ungethüm, J. Pietrzyk, A. Krause, D. Habich, and W. Lehner. Morphstore: Analytical query engine with a holistic compression-enabled processing model. *Proc. VLDB Endow.*, 13(11):2396–2410, 2020.

[9] D. Habich, A. Krause, J. Pietrzyk, C. Faerber, and W. Lehner. Simplicity done right for simdified query processing on CPU and FPGA. In *Proceedings of the 1st Workshop on Simplicity in Management of Data, SiMoD@SIGMOD 2023, Bellevue, WA, USA, 23 June 2023*, pages 3:1–3:5. ACM, 2023.

[10] J. Hildebrandt, D. Habich, and W. Lehner. BOUNCE: memory-efficient SIMD approach for lightweight integer compression. *Distributed Parallel Databases*, 41(3):439–466, 2023.

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

[12] Intel. Variable-precision data type support. https://www.intel.com/content/www/us/en/docs/oneapi-fpga-add-on/developer-guide/2024-0/var-prec-fp-sup.html, 2023. Accessed: 05.12.2023.

[13] Intel Corporation. Intel data parallel c++ compiler. https://github.com/intel/llvm.

[14] W. Jiang, D. Korolija, and G. Alonso. Data processing with fpgas on modern architectures. In *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023*, pages 77–82. ACM, 2023.

[15] Khronos SYCL Working Group. Sycl specification 2020. https://www.khronos.org/sycl/.

[16] R. Lasch, M. Moghaddamfar, N. May, S. S. Demirsoy, C. Färber, and K. Sattler. Bandwidth-optimal relational joins on fpgas. In *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*, pages 1:27–1:39. OpenProceedings.org, 2022.

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

[18] M. Moghaddamfar, C. Färber, W. Lehner, and N. May. Comparative analysis of opencl and RTL for sort-merge primitives on FPGA. In *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*, pages 11:1–11:7. ACM, 2020.

[19] M. Mohsen, N. May, C. Färber, and D. Broneske. Fpga-accelerated compression of integer vectors. In *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*, pages 9:1–9:10. ACM, 2020.

[20] R. Müller and J. Teubner. Fpgas: a new point in the database design space. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, volume 426 of *ACM International Conference Proceeding Series*, pages 721–723. ACM, 2010.

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

[22] O. Polychroniou and K. A. Ross. VIP: A SIMD vectorized analytical query engine. *VLDB J.*, 29(6):1243–1261, 2020.

[23] J. Reinders et al. *Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL*. Springer Nature, 2021.

[24] F. Silvestri and R. Venturini. Vsencoding: efficient coding and fast decoding of integer lists via dynamic programming. In J. X. Huang, N. Koudas, G. J. F. Jones, X. Wu, K. Collins-Thompson, and A. An, editors, *Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010, Toronto, Ontario, Canada, October 26-30, 2010*, pages 1219–1228. ACM, 2010.

[25] The CXL Consortium. Compute express link. https://www.computeexpresslink.org/, 2019.

[26] A. Ungethüm, J. Pietrzyk, P. Damme, A. Krause, D. Habich, W. Lehner, and E. Focht. Hardware-oblivious SIMD parallelism for in-memory column-stores. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.