VarVE: Bringing SIMD Performance to Variable-width Values

Chen Zou
Google LLC†
Bellevue, WA, United States
chenzou@google.com

Andrew A. Chien
University of Chicago and Argonne National Laboratory
Chicago, IL, United States
achien@cs.uchicago.edu

***Abstract*—Processor datapaths grew from 4 to 512 bits via Single-Instruction-Multiple-Data (SIMD) parallelism. SIMD applies the same operation to multiple values, which increases performance and reduces the instruction count. However, these evolvements do not provide support for variable-width values, so programmers 'pad' values to align to outliers, wasting the upper bits with zeros in both registers and datapath.**

We propose VarVE, a vector instruction set extension built upon the state-of-the-art vector-length agnostic SIMD instruction set: ARM SVE. VarVE provides native support for variable-width values within a vector, avoiding padding waste, thus making better use of the SIMD datapath. VarVE's design enables a flexible strip mining model with a variety of optimizations.

Evaluation of VarVE shows 60x speedup over ARM in kernels with element packing and unpacking, and 1.3x - 5.4x speedup over SVE for pure-compute filtering in TPC-H benchmarks. VarVE also achieves 2x speedup on a neural network inference task. All these results exemplify VarVE's general ability to improve datapath and memory system efficiency.

***Index Terms*—SIMD, VLA SIMD, Variable-width values**

I. Introduction

Datapaths in major microprocessors evolved from 4 bits [1] to 512 bits [2] over the past semicentury, driven by the continuous need of performance improvement. The evolvement includes both the growth of bitwidth for each value from 4 bits [1] to 64 bits [3], and the growth of the number of values processed at a time, enabled by the Single-Instruction-Multiple-Data (SIMD) [4], [5].

However, 64, the number of datapath bits assigned for each value, is mostly arbitrary. A real-world value is more diverse than just 64-bit or 32-bit. The diversity usually results from the biases, precisions and ranges during digitization. For example, temperature measured by the mercury thermometer or silicon bandgap sensor [6] can be represented as 7-bit or 12-bit one-fraction-digit decimals. Credit card limits and transactions could be represented as 24-bit integers, with one being a cent. More than half of the datapath and registers are wasted on zeros when computing on these data.

There has been little effort or innovation to provide efficient compute support for variable-width values to reduce the datapath waste. Here, by variable-width values compute support, we are referring to the mechanism to compute over a set of values and consume for each value only the number of bits needed to represent the value. This should not be confused with datatype innovations, e.g. bfloat16 [7], which make better use of a fixed number of bits.

This work considers new compute supports dubbed VarVE, a SIMD **E**xtension that natively computes on **Var**iable-width **V**alues. The goal is to achieve higher performance and memory efficiency by allowing more values to be packed into each datapath cycle or memory operation. And we build VarVE on top of ARM SVE, a state-of-the-art vector-length agnostic (VLA) SIMD ISA. The ISA innovatively answers how to load variable-width values in diverse coding to SIMD vectors, and how to provide flexible and high-efficiency compute without leading to the instruction set explosion that burdens the programmer and the compiler.

The specific contributions of the paper are:

- Characterization and analysis of low datapath efficiency in existing architectures, failing to adapt to diverse data.
- VarVE: A novel SIMD instruction set extension and its associated programming model, which provides native support for variable-width values without padding wastes aligning to a large outlier, largely improving compute datapath efficiency on diverse data.
- Evaluation of VarVE using various workloads shows it achieves 1.3x - 5.4x speedup over ARM SVE, the state-of-the-art SIMD ISA, with the same datapath width. Implementation shows that VarVE can be implemented with similar timing to fixed-width SIMD.

The rest of the paper is as follows. We cover VLA SIMD background and related work in Section II. Our motivation is then detailed in Section III, investigating the problems and challenges of existing SIMDs. The VarVE ISA design is discussed in Section IV, which we further evaluate in Section V. The challenges of hardware implementation are outlined and addressed in Section VI. We summarize and identify directions for future work in Section VII.

II. Background

A. Vector-length agnostic SIMD

Vector-length agnostic (VLA) SIMD ISAs like SVE [5] and RVV [8], [9] are proposed to address the instruction duplication issue from SIMD ISA sets with various datapath

†Work done while a Ph.D. student at the University of Chicago

```
void sve_baxpy(int N, uint8_t a, uint8_t* output,
    const uint8_t* xdata, const uint8_t* ydata) {
  for (int i = 0; i < N; i += svcntb()) {
    svbool_t pred = svwhilelt_b8(i, N);
    svuint8_t x = svld1_u8(pred, xdata);
    svuint8_t y = svld1_u8(pred, ydata);
    svuint8_t z = svmul_m_u8(pred, x, a);
    svst1_u8(pred, output, svadd_u8_m(pred, z, y));
    xdata += svcntb();
    ydata += svcntb();
    output += svcntb();
  }
}
```

Fig. 1: aX+Y for byte vectors in SVE: an example of VLA

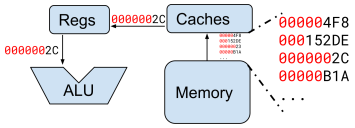

Fig. 2: Wastes (red zeros) from value-datapath mismatch

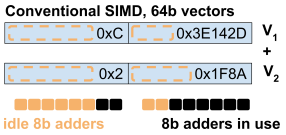

Fig. 3: Padding wastes both the VecReg and datapath

widths (e.g. SSE, AVX, AVX2, then AVX512). In VLA SIMD, only one instruction is needed for an operation for various datapath widths. The datapath width (i.e. vector length) is reported by hardware at runtime. The semantics of each VLA SIMD instruction (number of elements processed) and the strip-mining process both follow the runtime-determined datapath width. Figure 1 shows a VLA example that performs a linear combination of two vectors x and y with coefficient a. In each iteration, it loads two vector strips from memory, performs a vectorized linear combination of the strips and stores the result back to memory. The program (or compiled binary) can run on hardware with different datapath widths (vector lengths), which is reflected by 'svcntb()'. On hardware with wider datapath, the vector strips are longer, more elements processed in one iteration, and fewer iterations.

However, as will be shown in Section III, VLA SIMDs still suffer from low datapath efficiency.

B. Application-specific Stream Computing Acceleration.

Recent work proposes application-specific acceleration for stream computing on small values. SparseCore [10] proposed stream ISA support that abstracts sparse data as streams of key-value pairs as well as computes on these streams. Bison-e [11] proposes mechanisms that embed multiple small values from two streams into two 32b/64b integer respectively, and carry out regular integer multiplication (a technique called binary segmentation) to get inner products or convolution of the two streams. VarVE, in constrast, is a general approach that provides full SIMD operation support for variable width values as a VLA SIMD extension. It applies to all kinds of workloads as will be shown in V.

III. Problem and Approach

A. Observations on Datapath Efficiency

As discussed in Section I, real-world data is diverse, and much does not need 64 bits to represent. The diversity usually comes from the biases, precisions and ranges embedded in the digitization process. Moreover, good programming practice considers the maximum value that could occur, even as a possible intermediate result, and type the variable accordingly. Execution of such programs on a conventional x64 processor leads to waste in memory traffic and bandwidth, cache capacity and bandwidth, and ALU, as in Figure 2.

For example, we consider a TPC-H filter [12]. To generate predicates on an enum column 'c_mktsegment' (the cardinality is five), all 64 bits of the datapath are involved in the compare instruction when only three bits are doing the essential work, producing a low datapath efficiency of 4.7%. Ideally, all of the datapath and register bits would be used for essential compute and storage. Here, 95% of them are wasted.

SIMD can partially mitigate the problem. MMX [13] was developed on this exact premise. By grouping a set of 8-bit values into a 64-bit register, one MMX instruction could apply the same operation to each value, both increasing performance, and wasting fewer ALU and register bits. Of course, MMX has evolved in to a long series of SIMD instruction set extensions for x86: SSE, AVX, AVX2, AVX-512 [2].

Modern SIMD instruction sets support various vector types, all with uniform bitwidths b ($b \in \{8, 16, 32, 64\}$) for each element. However, they still do not deliver high datapath efficiency. The primary reason is that programming must be conservative. When selecting an element type, a programmer must account for the largest outlier inputs, results, or even intermediate results, i.e. the worst case, as in Figure 3. This causes poor efficiency. In our motivating TPC-H filter example, the predicates need to be applied to the 'c_custkey' column, which is defined in the schema as 32-bit per value. As a result, all predicates from the 'c_mktsegment' column have to be calculated in 32-bit-per-value vector types in ARM SVE, achieving 9.4% datapath efficiency.

B. The Problem

The fundamental problem that gives rise to low datapath efficiency is the software-hardware interface, i.e. the instruction set, which have been defined around fixed-width datatypes for more than seven decades. If an instruction set can only address and describe operations on fixed-width datatypes, to allow for dynamic 'fluctuation' and outliers, an application (and compiler) has little leeway to select the datatype size conservatively, producing waste. The more variable the values in the computation, the greater the waste – in memory, data movement, and computation. Our objective is to redesign the software-hardware interface to address this waste.

IV. VarVE ISA Design

The VarVE ISA extension consists of two types of features: 1) efficient packing/unpacking to load/store variable-width

VarVE-pup	int FRStream(char* pdata, int width, int bits)
	int HRStream(char* pdata, char* pdict, int bits)
	int VRStream(char* pdata, char* pmeta, int bits)
	int pack(svbool t pred, svint<w>t v, int sid)
	svint<w>t unpack(int sid)

VarVE-full	int FRStream(char* pdata, int width, int bits)	
	int HRStream(char* pdata, char* pdict, int bits)	
	int VRStream(char* pdata, char* pmeta, int bits)	
	int vpack(vsbool t pred, vsint t v, int sid)	
	vsint t vunpack(vsbool t pred, int sid, int mb=64)	
	vadd, vsub, vmul, vdiv	**Arithmetic OP** Variants [15]
	vmax, vmin, vasl, vasr	Predicated Immediate
	vand, vorr, veor, vbic	Predicated Vectors
	vcle, vclt, vcne, vceq	Unpredicated Immediate
	vcpy, vdup, vindex, vabs	Unpredicated Vectors

TABLE I: Summary of VarVE instructions

data and 2) efficient computation on variable-width data. In both cases, these designs leverage and integrate gracefully with the modern vector-length agnostic (VLA) architectures [5]. Two ISA variants are proposed. VarVE-pup is a minimal extension that contains only the pack/unpack feature to allow conventional SIMD easily load variable-width data for compute. It reflects our best-effort capture of existing streaming ISA work [11], [14]. VarVE-full is a full variable-width value extension that not only inherits the vpack/vunpack support for efficient loading, but also performs SIMD computation on native variable-width vector type.

A. *Efficiently Load Variable-width Values: Pack/Unpack*

Data are diverse (ints, strings, enums, codes, etc.). Incoming values can feature arbitrary bitwidths that are packed or coded for storage or transfers. Existing SIMD ISAs require lengthy and complex bit operations to capture a stream of variable-width values, placing them into a vector register. In Figure 4, we show an example of unpacking (and loading) bitpacked values. ARM SVE needs ∼10 instructions to first generate byte positions for each value, gather the values, and then do bit operations to mask out the extra bits gathered.

In VarVE-pup, we introduce a pair of universal pack & unpack instructions to solve this problem (see Table I). They convert between packed values in a stream and fixed-width elements in conventional SIMD vectors. An additional interface to describe input streams is also added, including fixed-width-value streams (FRStream), variable-width-value streams (VRStream) and Huffman streams (HRStream), as well as FWStream, VWStream, HWStream for outputs.

VarVE-pup instructions significantly reduce the instruction counts through ISA support as shown in Figure 4, to one instruction per iteration. And it enables performance improvement through microarchitecture optimizations like prefetch and preunpack [14]. Once a stream is specified with stream description instructions, unpacking could happen in the background and hide its latency among computation. Further, VarVE-pup instructions remove the roadblocks (performance penalty of loading) of using non-padded variable-width value format in storage and transfers, improving memory system efficiency compared to using padded fixed-width format.

```
void sve_unpack(int w, int bits, const void *data) {
  // read 2B around each value. truncate and align
  for (int i = 0; i < bits / w; i += svcntw()) {
    svbool_t pg = svwhilelt_b32(i, bits);
    svuint32_t vid = svdup_u32(0);
    for (int j = 0; j < w; j++)
      vid = svadd_u32_m(pg, vid, svindex_u32(0, 1));
    svuint32_t vbid = svlsr_n_u32_m(pg, vid, 3);
    svint32_t vx = svld1sh_gather_offset_s32(pg, data, vbid);
    vid = svand_n_u32_m(pg, vid, 7);
    vx = svasr_s32_m(pg, vx, vid);
    vx = svand_n_s32_m(pg, vx, (1 << w) - 1);
    // vx parsed and aligned in 32b-per-element vector
  }
}
void varve_unpack(int w, int bits, const void *data) {
  int sid = frstream(data, w, bits);
  for (int i = 0; i < bits / w; i += svcntb()) {
    svint8_t vx = unpack_s8(sid);
    // vx parsed and aligned for compute
  }
}
```

Fig. 4: SVE vs VarVE-pup: unpack and load the bitpacked

B. *Efficiently Compute on Variable-width Values: Native Vector Support*

In VarVE-full, we leverage the high-level view pioneered by VLA SIMD (SVE), that abstracts away datapath width[1] (see Section II-A). The key is that with the VLA approach, the same binary program works with different hardware datapath widths; the number of vector elements for each iteration is determined at runtime. The datapath width still determines performance, but it is no longer embedded in the programs.

A new vector type where the bitwidth of each element in a single vector can vary, as shown in Figure 5, is added. This allows denser packing of variable-width values into vector registers. In terms of architectural states, variable-width value vectors reuse the existing vector registers. New vector metadata registers are added, one metadata register per vector register, storing the bitwidth of each element for the associated variable-width values vector. Vector metadata registers are read/written when carrying out new vector operations defined below. No standalone instruction can read/write/change them.

Further, new operations on the new vector type are added, as summarized in the last five rows of Table I. These instructions still carry out element-wise operations between vectors. Though, the number of output vector elements produced by each instruction is dynamic, as shown in Equation 1. The rule reflects that output elements cannot be more than the input elements in either operand. And if there is backpressure when adapting corresponding input vectors to the datapath to compute the output vectors, the number of output elements may further decline. This dynamic property gracefully hides into the VLA scheme, as will show in Section IV-C.

$$E_{op(u,v)} <= min(E_u, E_v) \tag{1}$$

In VarVE-full, the pack & unpack instructions are upgraded as vpack & vunpack to target variable-width-value vectors. vunpack adds a scalar argument mb and a predicate argument

[1]This is a marked contrast to AVX or Neon where the datapath width is explicit in the ISA, and embedded in every program.

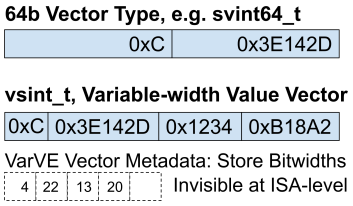

Fig. 5: Native support for variable-width values in VarVE-full

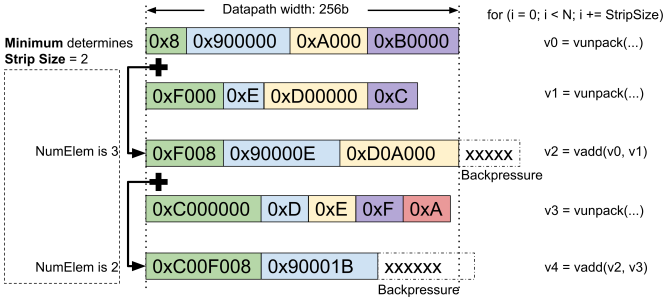

Fig. 6: Strip size is determined at the end of the iteration

```
SELECT c_custkey FROM Customer WHERE c_mktsegment == BUILDING
void q03c_sve(int N, void* custkey_i, void* mktsegment_i, void* custkey_o,
              int& elems) {
  for (int i = 0, n = svcntw(); i < N; i += n) {
    svbool_t pg = svwhilele_b32_s32(i, N);
    svint32_t v_custkey = svld1_s32(pg, (const int32_t*)custkey_i);
    svint32_t v_mktsegment = svld1ub_s32(pg, (const uint8_t*)mktsegment_i);
    pg = svcmpeq_n_s32(pg, v_mktsegment, BUILDING);
    unsigned d = svcntp_b32(pg, pg);
    elems += d;
    v_custkey = svcompact_s32(pg, v_custkey);
    svst1_s32(svwhilelt_b32_u32(0, d), v_custkey, custkey_o);
    custkey_i += n;
    mktsegment_i += n;
    custkey_o += d;
  }
}
void q03c_varve(int N, void* custkey_i, void* mktsegment_i, void* custkey_o,
                int& elems) {
  int c_custkey_s = frstream(custkey_i, 32, elems << 5);
  int c_mktsegment_s = frstream(mktsegment_i, 8 , elems << 3);
  int c_custkey_t = fwstream(custkey_o, 32);
  for (int i = 0, n = 0; i < N; i += n) {
    vsbool_t pg = vsptrue();
    vsint_t v_c_mktsegment = vunpack(pg, c_mktsegment_s);
    pg = vcmpeq(pg, v_c_mktsegment, BUILDING);
    vsint_t v_c_custkey = vunpack(pg, c_custkey_s);
    elems += vpack(pg, v_c_custkey, c_custkey_t);
    n = rcnume();
    crstream(c_custkey_s, n);
    crstream(c_mktsegment_s, n);
  }
}
```


Fig. 7: TPC-H filtering: SVE vs VarVE-full

pred. Argument *mb* constrains the maximum width vector elements can grow to (default is 64b). This maximum constraint is transitively applied through VarVE-full arithmetic operations. If the result of an operation is larger than the maximum, normal SIMD overflow behavior occurs. The *pred* argument is needed for an optimization (Section IV-D).

The benefits of VarVE-full vector type and instruction design are twofold. First, VarVE-full avoid VecReg wastes on padded values. Second, dynamic adapting of value vectors to the datapath allows increased data parallelism by putting more vector elements through the datapath each cycle.

C. VarVE-full Strip Mining: Lockstep Advancing

The management of strip mining pacing is the responsibility of any VLA architecture. In existing VLA SIMDs, the strip size, i.e. the number of elements processed for one stream each iteration, is abstracted from the program. It is instead a static property of implementation hardware, and a runtime constant in programming. This allows all VLA instructions in the strip mining loop to advance in *lockstep*, when processing multiple streams with element-wise correspondence.

However, in VarVE-full, the strip size is no longer a runtime constant. the feasible strip size is dependent on the dynamic size of the values. In principle, the strip size could be different for each iteration of a loop. Further, the appropriate strip size may not be determined until the last instruction in an iteration, as shown in Figure 6, where the effective strip size is successively constrained by afterwards instructions.

To efficiently determine the maximum feasible strip size, we introduce the 'NumE' misc register. This register is updated by each of the VarVE-full instructions, using the min function. After a sequence of VarVE-full instructions in a strip mine loop, it contains the lowest #elements produced by any operation, which is exactly the maximum feasible strip size to deliver lockstep advance, i.e. maintaining element-wise correspondence across streams. This 'NumE' register is similar to the ARM NZCV conditional flags [16].

For the overall scheme to work, the semantics of 'vunpack' are modified. It now just populates the vector with values from a stream, but no longer consumes the corresponding bits. At the end of the iteration, after the strip size/advance is determined, new 'crstream' instructions would be called to advance the stream, consuming bits representing 'NumE' values, as shown at the end of Figure 7.

D. Masking Optimization

Figure 7 provides a concrete example of VarVE-full lockstep strip mining and its comparison with existing SVE strip mining on a filter kernel from the TPC-H benchmark [12]. It also exemplifies a new optimization opportunity in VarVE-full which we call 'Masking Optimization'.

One problem with VarVE-full is that one stream with wide values (in bitwidths) can slow an entire loop. That's because the minimum #elements of all vector operations will determine the rate of progress (lockstep strip size) for the entire loop. The datapath will be poorly utilized when processing other lock-stepped narrow-value streams.

We observe that if we have a mask for a slow input stream, we can accelerate it by pushing the mask into the stream. This idea is akin to predicate push-down in databases. VarVE-full allows the mask to be supplied to the vunpack instruction (as in Figure 7 for v_c_custkey) to transform unneeded values to zeroes, such that they consume neither vector register bits, nor datapath bits in later computation. This optimization would produce larger strip sizes and thus higher datapath efficiency.

This optimization is particularly useful when mask/filtering could be calculated from input streams with narrower (on average) values, and then the mask is applied to the unpacking of streams with wider values. With masking optimization, the reduction of strip size during instructions operating wider values could be smaller or even zero.

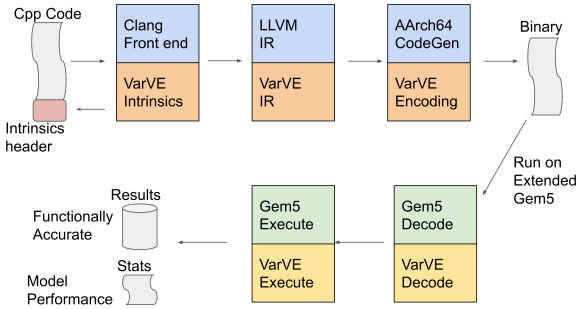

Fig. 8: Compilation and simulation infrastructure

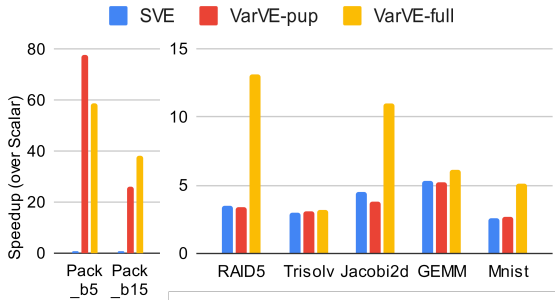

Fig. 9: Speedup for different kernels (vs. Scalar)

In fixed-width-element ISAs (SVE, VarVE-pup), masks cannot improve datapath efficiency the same way. The masked values still have to take up the same #bits to keep the element boundary unchanged for datapath computation. Only in variable-width vector element ISA like VarVE-full, where the element boundaries are relaxed to not to align to a fixed bitwidth, can we reduce the masked elements' bit consumption to zero and improve datapath efficiency.

V. Evaluation

A. Methodology

a) Modeling and Software Infrastructure: Gem5 [17] is extended with VarVE instructions as shown in Figure 8. Gem5 ARM-Ex5 core model [18] is employed as the performance model. The memory hierarchy is 8-way 32KB L1I, 8-way 32KB L1D, 16-way 256KB L2, all with 64B cache blocks and a DDR4-4800 16GB DRAM. Since most of our workloads feature streaming accesses [19], caches have little impact. Further, we added intrinsic support (for C/C++) and code generation for VarVE instructions to LLVM, occupying unused opcodes in the ARM SVE encoding space.

b) ISA Variants: Four ARM variants, AArch64 Scalar, SVE, VarVE-pup, VarVE-full are evaluated. The datapath widths (i.e. vector length) of three SIMD variants are all 256 bits. To recap, VarVE-pup is a partial extension that adds only the pack/unpack instructions that converts between packed streams and conventional fixed width vectors (as in Section IV-A), and serves as our best-effort capture of existing streaming ISA [11], [14]. VarVE-full adds the full-fledged vector support of variable-width values. VarVE-full's performance model includes the hardware characteristics as discussed in Section VI. There is an additional pipeline stage and alignment ceiling for the alignment process in EAU (Section VI-B).

c) Workloads: Various computation kernels (many from PolyBench [20]), database analytics filters from TPC-H [12], and neural network inference to show VarVE's generality.

d) Metrics:

- **Runtime:** Exec time in Gem5 cycle-accurate simulation
- **Speedup:** Ratio of scalar execution time to that of one SIMD implementation

B. Various Computation Kernels

In Figure 9, we show the speedups for each kernel of three SIMD variants, compared to the scalar implementation.

In the bitpacking evaluation, two TPC-H columns 'p_brand' and 'o_orderdate' which are 8-bit and 32-bit unpacked integers respectively are bitpacked to fixwidth streams of 5-bit or 15-bit per value. Lack of byte-unaligned (bit-level) scatter support, bitpack implementation in SVE has to fall back to the Scalar implementation and performance. **VarVE-pup matches this workload well, and achieves 78x and 26x speedup** respectively through vectorization, ISA and microarchitecture support of latency-hiding write-path packing. The 3x difference in speedup for two columns results from data types: More elements can be processed in one iteration for narrower input type. **VarVE-full further improves the vectorization density for the 15-bit per value bitpacking, achieving 38x speedup.** For the 5-bit per value bitpacking, even with the same vectorization density as VarVE-pup, **VarVE-full was limited to 60x speedup by the additional auxiliary work (determine advance and consume input streams) in the lockstep strip mining.**

The other kernels all focus on computations on aligned elements, rather than data format transformations. VarVE-pup, with no unpacking and packing acceleration benefits to exploit, achieves roughly the same speedup with SVE across the board. Both their speedup comes from a wider datapath (256b vs. 32b), less the auxiliary work of conventional strip mining. VarVE-full benefits from denser usage of vector register and vector ALU. Especially on RAID5, where input values only average at 8 bits, but have to be conservatively handled in 32-bit type in normal SIMDs. VarVE-full can theoretically compute 4x more elements per instruction than SVE. For GEMM and Trisolv, the benefits of VarVE-full over SVE are less. Because the accumulative semantics in the workloads bring up the average value widths, and there is less datapath inefficiency to be reclaimed by VarVE-full. Last but not the least, in a 6-layer neural network for MNIST [21], by applying masking optimization on the 80% pruned weights, VarVE-full achieve 2x speedup over SVE. **On average, VarVE-full is 1.9x faster than SVE on these compute-oriented kernels.**

C. Database Analytics Workloads

Then we evaluate the performance of four ISAs on the filters in TPC-H data analytics. The workload is studied in the

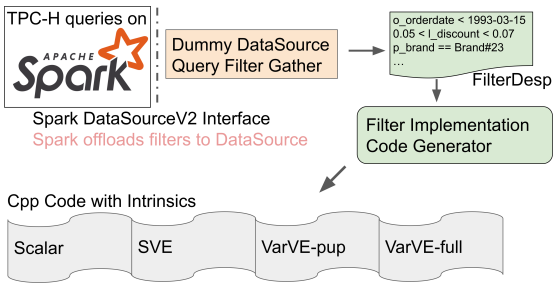

Fig. 10: TPC-H filters offload structure

majority of computational SSD work [22]–[26]. As shown in Figure 10, workloads are collected from the Spark DataSource interface where various filters [27] are pushed down to data sources. An ad hoc code generator with functionality similar to Spark Tungsten is implemented to generate four variants of filters in C++ with corresponding intrinsics. All input data are unpacked as types defined by TPC-H schema [12].

Figure 11 shows the speedup over Scalar. **VarVE-pup achieves 1.2x - 7.5x speedup over Scalar, which is 1.5x over SVE by GeoMean.** The performance benefits come from the elimination of pointer increments. **VarVE-full achieves 2.6x - 15x speedup which averaged (GeoMean) at 5.7x over Scalar, or 1.3x - 5.4x speedup which averaged (GeoMean) at 2.1x over SVE.** It results from efficient (denser) usage of the same 256-bit datapath, and also showcases the benefits of masking optimization as discussed in Section IV-D for multi-predicate filtering. The values for the rows filtered out by the first predicate are set to zero during predicated unpack of the second column, saving VecReg and datapath bits.

D. Summary

Evaluation with workloads of multiple domains including file system, data analytics and neural network inference suggest that VarVE delivers significant performance increase. These improvements arise from greater datapath efficiency We also performed experiments for VarVE-full that show both memory traffic reductions and almost-linear speedup scalability when varying the vector length, but those results are omitted here due to paper length restrictions.

All in all, evaluation results suggest VarVE-full delivers speedup via higher datapath efficiency for general computing workloads. Comparing VarVE-pup with VarVE-full, we find that even sophisticated pack/unpack is not sufficient to capture the speedup shown. Native support for variable-width values is key to the greatest performance benefits.

VI. Implementing VarVE-full

Our goal of this section is to show that VarVE-full could be implemented with same clock frequency and limited area/power increase, compared to conventional SIMD engines.

A. VarVE-full Microarchitecture

A representative SIMD microarchitecture based on previous work [4], [13], [28]–[30], is shown by orange components in Figure 12. Generally, the vector register provides operands for the the vector ALU (VecALU) and accepts the result vector.

Instruction	**Width** (to denote as Max*)
Add/Sub	$Max(W_a, W_b) + 1$
Mul	$W_a + W_b$
Logical	$Max(W_a, W_b)$
Comparison	$Max(W_a, W_b)$

TABLE II: Alignment width, two inputs denoted as a and b

	Area (mm^2)	**#Gates**	**Power (mW)**	**CycleTime (ns)**
EAU	0.0134	43145	4.68	1.0

TABLE III: Area and power of EAU with SAED14nm

As discussed in Section IV-B, vector metadata registers are added in VarVE-full, one per vector register, to store the bitwidth of each vector element in the vector storing variable-width values. Vector metadata registers accept writes from VecALU and service reads for VecALU (through EAU, discussed later in Section VI-B) by VarVE-full instructions.

Since elements are of variable widths, corresponding elements between two vectors in a vec-vec operation are no longer aligned. VarVE-full adds an Element Alignment Unit (EAU) as a VecALU pipeline step as shown in Figure 12.

To provide operations on two elements aligned at an arbitrary bit position, the vector ALU has to be enhanced, especially for those operations that involves passing carry signals across element boundaries, as the element boundaries are now dynamic with variable-width values.

Further, we require a prefetcher and a packer for vunpack, and the complement for vpack. Previous work UVE [14] and Bison-E [11] demonstrated efficient designs for these.

We will now focus on missing pieces: EAU and VecALU.

B. Element Alignment Unit

The Element Alignment Unit (EAU) aligns corresponding elements from two variable-width-value vectors to compute in VecALU. The alignment process considers both the actual (leading-zero stripped) value widths from two vectors and the width of the result after applying the operation specified by each instruction. We summarize the rules of alignment width for different VarVE-full instructions in Table II.

Figure 13 depicts how the EAU is implemented in hardware. A vanilla Parallel Prefix Sum (PPS) unit is used to calculate element boundaries from the alignment widths determined according to Table II. And vector elements from each vector are aligned on these boundaries in an Aligner, which is mostly a hardware gather (i.e. per-position MUX) on the input vector.

To reduce circuit implementation cost and circuit latency, we opt to align vector elements only to rounded-up 8-bit boundaries. This reduces the circuit scale of the PPS unit and the MUXes in the Aligner by 8x and 64x respectively, compared to bit-level alignment. As shown in Table III, We closed timing of EAU implemented in SystemVerilog at 1GHz on SAED 14nm [31] via Design Compiler, fast enough as a pipeline step in a SIMD unit, e.g. in Ara [30]. Please notice that the round-up of alignment is invisible to the software programming. Because the strip mining process is already dynamic depending on the values processed, as discussed in Section IV-C. The round-up of alignment hides delicately as if each value is slightly larger. For the same reason, other

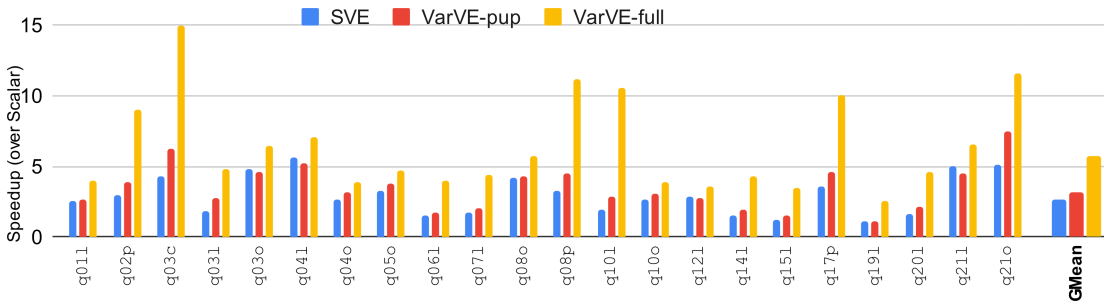

Fig. 11: Speedup for TPC-H filters. TaskName = QueryID + TableNameLeadingCharacter (vs. Scalar)

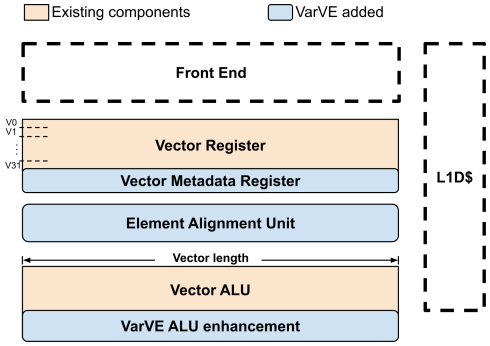

Fig. 12: Microarchitecture for SIMD and VarVE-full

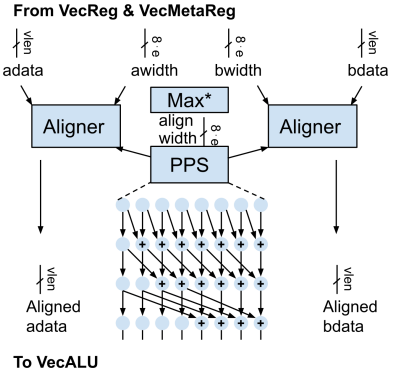

Fig. 13: Element alignment unit

and future implementations are free to choose finer alignment granularity without worries of breaking ISA promises. This is a similar mechanism as the vector-length agnosticism [5], [8] that allows implementation freedom while maintaining the compatibility of an already-compiled binary program.

With the EAU, instructions without cross-bit carries can be implemented with conventional VecALU unchanged. This is a significant milestone on hardware feasibility of VarVE-full.

C. Dynamic Carry Look-ahead Adder

Adaptations are needed for instructions that have carries to account for the fact that element boundaries are now dynamic depending on the sizes of variable-width values in each vector. As an example, we dive deep into an adder hardware adaptations here, drawn in Figure 14.

In existing SIMD like AVX or SVE, the carry-lookahead unit for vector adding should already be runtime-configurable, because vector element type could be byte, half, word, double

	Size/ByteID	Carry Logic '|' denotes 'logical or'
AVX NEON SVE (MSB)	Byte	0
	Half	G_{-1}
	Word	$G_{-1} \mid P_{-1}G_{-2} \mid P_{-1}P_{-2}G_{-3}$
	Double	$\mid_{i=1}^{7} G_{-1}\Pi_{j=1}^{i-1}P_{-j}$
VarVE-full	0	0
	1	G_{-1}
	2	$G_{-1} \mid P_{-1}G_{-2}$
	3	$G_{-1} \mid P_{-1}G_{-2} \mid P_{-1}P_{-2}G_{-3}$
	4	$\mid_{i=1}^{4} G_{-1}\Pi_{j=1}^{i-1}P_{-j}$
	5	$\mid_{i=1}^{5} G_{-1}\Pi_{j=1}^{i-1}P_{-j}$
	6	$\mid_{i=1}^{6} G_{-1}\Pi_{j=1}^{i-1}P_{-j}$
	7	$\mid_{i=1}^{7} G_{-1}\Pi_{j=1}^{i-1}P_{-j}$

TABLE IV: Carry input logic for each 8-bit adder P_i and G_i are propagate and generate signals [32]

-sized. As a result, there are four different logics to choose from when calculating the input carry signal for each 8-bit adder, as summarized in Table IV with the logic for the most significant byte (MSB) shown. For VarVE-full, this list grows to eight different variants, with the selection signals being the in-element ByteIDs for each 8-bit adder position, which are byproducts of the Element Alignment Unit. Please notice that the carry at most comes from the 8-bit adder 7 bytes away, which is the same with conventional SIMD or VLA SIMD ISAs. As a result, the adaptations should have limited impacts on the circuit critical path, timing or silicon area/power.

A similar approach can be used for an integer multiplier in VarVE-full. The key insight is that multipliers are mostly composed of partial product compression, (i.e. multiple adders). We exemplify a Wallace-tree-based vector multiplier as in Figure 15, but a booth multiplier could be adapted similarly. This multiplier is dynamically partitioned based on the EAU's chosen alignment (Table II), keeping the computation for each pair of operands separate. Here a 5-bit product (2-bit, 3-bit) is cleanly separated from a 4-bit product (2-bit, 2-bit). The compression section at the bottom is just several dynamic carry look-ahead adders we just discussed.

VII. Summary and Future Work

To address the low datapath efficiency in SIMD architectures, we propose VarVE-full, a vector instruction set extension designed on top of the state-of-the-art vector-length agnostic SIMD instruction set: ARM SVE. VarVE-full provides native support to handle variable-width values.

Evaluations suggest that VarVE-full provides on average 2.1x speedup via datapath efficiency improvement over SVE for workloads across file system, data analytics and neural

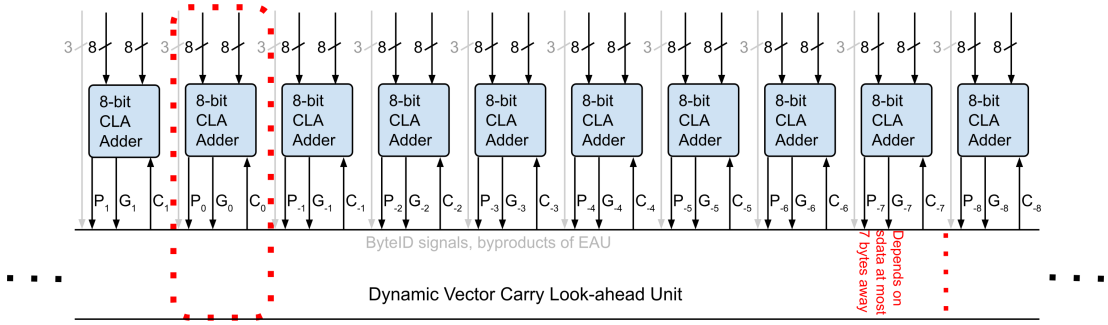

Fig. 14: Dynamic vector carry look-ahead adder (subscriptions follow the order of significance)

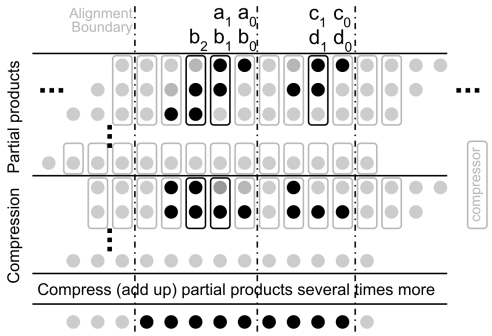

Fig. 15: A portion of a variable-width value vector multiplier (multiplication of two pairs of values shown)

network inference. Moreover, the memory system could also benefit from efficient variable-width value support.

Interesting future directions include different encoding mechanisms in vector metadata register, compilation for VarVE, and incorporation into out-of-order cores.

Acknowledgment

This work is supported by a grant from Samsung MSL, NSF Award CNS-1909364, and the UChicago CERES Center.

References

[1] "Intel's First Microprocessor," https://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html.

[2] "Intel Advanced Vector Extensions 512," https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html.

[3] K. McGrath *et al.*, "The amd x86-64 architecture: Extending the x86 to 64 bits," in *2002 IEEE hot chips 14 symposium (HCS)*. IEEE, 2002, pp. 1–16.

[4] S. K. Raman *et al.*, "Implementing streaming simd extensions on the pentium iii processor," *IEEE micro*, vol. 20, no. 4, pp. 47–57, 2000.

[5] N. Stephens *et al.*, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[6] T. Instruments, "Temperature sensing fundamentals," https://www.ti.com/lit/an/snoaa25/snoaa25.pdf.

[7] S. Wang *et al.*, "Bfloat16: The secret to high performance on cloud tpus," *Google Cloud Blog*, vol. 4, 2019.

[8] "riscv-v-spec," https://github.com/riscv/riscv-v-spec.

[9] Patterson, David, "SIMD Instructions Considered Harmful," https://www.sigarch.org/simd-instructions-considered-harmful.

[10] G. Rao *et al.*, "Sparsecore: stream isa and processor specialization for sparse computation," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 186–199.

[11] E. Reggiani *et al.*, "Bison-e: a lightweight and high-performance accelerator for narrow integer linear algebra computing on the edge," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 56–69.

[12] "TPC-H dataset," http://www.tpc.org/tpch/.

[13] A. Peleg *et al.*, "Mmx technology extension to the intel architecture," *IEEE micro*, vol. 16, no. 4, pp. 42–50, 1996.

[14] J. M. Domingos *et al.*, "Unlimited vector extension with data streaming support," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 209–222.

[15] "ARM SVE Encoding," https://developer.arm.com/documentation/ddi0602/2022-06/Index-by-Encoding/SVE-encodings?lang=en.

[16] "AArch64 NZCV Condition Flags," https://developer.arm.com/documentation/ddi0595/2021-06/AArch64-Registers/NZCV--Condition-Flags.

[17] N. Binkert *et al.*, "The gem5 simulator," vol. 39, no. 2, p. 1–7, aug 2011.

[18] "Gem5 Ex5 Core," https://gem5.googlesource.com/public/gem5/+/refs/heads/stable/configs/common/cores/arm/ex5_LITTLE.py.

[19] C. Zou *et al.*, "Assasin: Architecture support for stream computing to accelerate computational storage," in *2022 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022.

[20] "PolyBench/C, the Polyhedral Benchmark suite," https://web.cs.ucla.edu/~pouchet/software/polybench/.

[21] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE signal processing magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[22] R. Schmid *et al.*, "Accessible near-storage computing with fpgas," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–12.

[23] B. Gu *et al.*, "Biscuit: A framework for near-data processing of big data workloads," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, vol. 44, no. 3, 2016, pp. 153–165.

[24] Z. Ruan *et al.*, "Insider: Designing in-storage computing system for emerging high-performance drive," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 379–394.

[25] G. Koo *et al.*, "Summarizer: trading communication with computing near storage," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 219–231.

[26] I. Jo *et al.*, "Yoursql: a high-performance database system leveraging in-storage computing," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 924–935, 2016.

[27] "Spark Filters," https://github.com/apache/spark/blob/master/sql/catalyst/src/main/scala/org/apache/spark/sql/sources/filters.scala.

[28] K. Asanovic, *Vector microprocessors*. University of California, Berkeley, 1998.

[29] J. E. Smith *et al.*, "Vector instruction set support for conditional operations," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 260–269, 2000.

[30] M. Cavalcante *et al.*, "Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2019.

[31] "Synopsys teaching resources," https://www.synopsys.com/community/university-program/teaching-resources.html.

[32] "Lookahead," https://en.wikipedia.org/wiki/Lookahead_carry_unit.