Revec: Program Rejuvenation through Revectorization

Charith Mendis*
MIT CSAIL
USA
charithm@mit.edu

Ajay Jain*
MIT CSAIL
USA
ajayjain@mit.edu

Paras Jain
UC Berkeley
USA
paras_jain@berkeley.edu

Saman Amarasinghe
MIT CSAIL
USA
saman@csail.mit.edu

ABSTRACT

Modern microprocessors are equipped with Single Instruction Multiple Data (SIMD) or vector instructions which expose data level parallelism at a fine granularity. Programmers exploit this parallelism by using low-level vector intrinsics in their code. However, once programs are written using vector intrinsics of a specific instruction set, the code becomes non-portable. Modern compilers are unable to analyze and retarget the code to newer vector instruction sets. Hence, programmers have to manually rewrite the same code using vector intrinsics of a newer generation to exploit higher data widths and capabilities of new instruction sets. This process is tedious, error-prone and requires maintaining multiple code bases. We propose Revec, a compiler optimization pass which *revectorizes* already vectorized code, by retargeting it to use vector instructions of newer generations. The transformation is transparent, happening at the compiler intermediate representation level, and enables performance portability of hand-vectorized code.

Revec can achieve performance improvements in real-world performance critical kernels. In particular, Revec achieves geometric mean speedups of 1.160× and 1.430× on fast integer unpacking kernels, and speedups of 1.145× and 1.195× on hand-vectorized x265 media codec kernels when retargeting their SSE-series implementations to use AVX2 and AVX-512 vector instructions respectively. We also extensively test Revec's impact on 216 intrinsic-rich implementations of image processing and stencil kernels relative to hand-retargeting.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**; • **Software and its engineering → Compilers**; *Software performance.*

KEYWORDS

vectorization, program rejuvenation, Single Instruction Multiple Data (SIMD), optimizing compilation

ACM Reference Format:
Charith Mendis, Ajay Jain, Paras Jain, and Saman Amarasinghe. 2019. Revec: Program Rejuvenation through Revectorization. In *Proceedings of the 28th International Conference on Compiler Construction (CC '19), February 16–17, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3302516.3307357

1 INTRODUCTION

Modern microprocessors have introduced SIMD or vector instruction sets to accelerate various performance critical applications by performing computations on multiple data items in parallel. Moreover, processor vendors have introduced multiple generations of vector instruction sets, each either increasing vector width or introducing newer computational capabilities. For example, Intel introduced MMX with 64-bit operations in 1997, Streaming SIMD Extensions (SSE) and 128 bit registers in 1999, SSE2, SSE3, SSSE3 and SSE4 from 2000–2006, AVX, AVX2 and FMA with 256 bit registers in 2011, and AVX-512 and 512 bit registers in 2016. SIMD instruction sets from other processor vendors include AMD's 3DNow! [22], IBM's VMX/Altivec [8] and ARM's Neon [2]. In order to use these SIMD units, programmers must either hand-code directly using platform-specific intrinsics, or rely on existing compiler auto-vectorization techniques to discover opportunities in programs written in mid- or high-level languages.

Modern compilers employ two main auto-vectorization strategies, namely, loop vectorization [1] and Superword Level Parallelism (SLP) based vectorization [12]. Auto-vectorization allows programmers to write code in high-level languages, while still benefiting from SIMD code generation. However, both loop and SLP vectorization rely on programmers writing code in ways which expose existing data level parallelism. In certain cases, the programmer needs to know the underlying implementation of compiler vectorization passes to cater her code writing style. Even then, auto-vectorization may not vectorize all vectorizable code regions due to inaccuracies in cost models, inability to perform certain transformations etc [17].

*Both authors contributed equally to this research.

CC '19, February 16–17, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6277-1/19/02.
https://doi.org/10.1145/3302516.3307357

In contrast, writing hand-vectorized code allows the programmers to exploit fine-grained parallelism in the programs more precisely. Hand-vectorization allows programmers to explicitly embed domain optimizations such as performing intermediate operations without type promotions, which cannot be achieved by compiler auto-vectorization. However, when manual vectorizing, programmers give up on both code and performance portability. The code that is the fastest for an older vector instruction set may perform suboptimally on a processor which supports wider vector instructions. For example, manually vectorized SSE2 code will not utilize the full data width of a processor that supports AVX2 instructions. This issue is aggravated as modern compilers do not retarget code written in low-level intrinsics to use newer vector instructions. Hence, programmers frequently maintain several architecture-specific, version-specific implementations of each computationally intensive routine in a codebase to exploit newer instructions. This is tedious, error-prone, and is a maintenance burden.

In this work, we propose compiler *revectorization*, the retargeting of hand-vectorized code to use newer vector instructions of higher vector width. We developed Revec, a compiler optimization technique to algorithmically achieve revectorization, and implemented it in the LLVM compiler infrastructure [14] as an IR level pass. Revec rejuvenates performance of stale implementations of data-parallel portions of hand vectorized programs, automatically adding performance portability.

Revec finds opportunities to merge two or more similar vector instructions to form vector instructions of higher width. Revec has its foundations in SLP auto-vectorization [12], but rather than transforming scalar code, focuses only on revectorizing already vectorized code and hence has its own unique challenges. More specifically, Revec needs to find equivalences between vector intrinsics with complex semantics and decide how to merge and retarget vector shuffle instructions to newer instruction sets.

Revec automatically finds equivalences between vector intrinsics across different instruction generations by enumerating all combinations. Equivalences are established through randomized and corner case testing. Revec also introduces vector shuffle merge patterns to handle revectorizing shuffle instructions. During compiler transformation, Revec first does loop unrolling and reduction variable splitting, with heuristics that are catered towards revectorization. These preprocessing transformations expose more opportunities for revectorization. Finally, it uses the automatically found equivalences and shuffle merge rules to merge two or more vector instructions of a lower data width to form vector instructions of a higher data width.

1.1 Contributions

In this paper, we make the following contributions:

- A compiler optimization technique, Revec, which automatically converts the hand-vectorized code to vector instructions of higher vector width including computations that involve reductions.
- Automatically finding instances of two or more vector intrinsics of similar semantics which can be merged into a vector instruction of higher width by enumerating and refining candidates.
- Vector shuffle merge rules to enable revectorizing vector shuffle instructions.
- Implementation of Revec in LLVM compiler infrastructure as an LLVM IR level optimization pass to transparently perform revectorization.
- Extensive evaluation of Revec on real-world performance critical kernels ranging from media compression codecs, integer compression schemes, image processing kernels and stencil kernels. We show Revec automatically achieves geometric mean speedups of 1.145× and 1.195× on hand-vectorized x265 media codec kernels, and speedups of 1.160× and 1.430× on fast integer unpacking kernels when retargeting their SSE implementations to use AVX2 and AVX512 vector instructions respectively. Further, Revec achieves geometric mean speedups of 1.102× and 1.116× automatically on 216 intrinsic-rich implementations of image processing and stencil kernels written using SSE-series (SSE2+) vector instructions.

2 MOTIVATION

We use the `MeanFilter3x3` kernel from the Simd Image processing library [9] to motivate how Revec achieves performance portability of hand-vectorized code. Consider different `MeanFilter3x3` implementations shown in Figure 1 using C-like code. Pixel values of an output image are computed by averaging the corresponding pixel values in a 3 by 3 window of an input image. Figure 1(a) and (b) show the scalar version and the SSE2 hand-vectorized version of the mean filter respectively. Note that boundary condition handling and certain type conversions from the original implementations are omitted for the sake of clarity. The scalar version implements a naive algorithm which goes through each output pixel and individually computes the mean of the corresponding pixels in the input image. It is computationally inefficient since it recomputes column sums for each row repeatedly. However, LLVM is able to auto-vectorize this scalar kernel without much analysis, and hence its performance scales with increases in the width of vector instructions.

The SSE2 version is algorithmically different from the scalar version. It maintains an array to store column sums of 3 adjacent rows. The inner loop computes the column sums of the bottom-most row (of the 3 rows) 8 elements at a time using SSE2 vector instructions. The output image pixel values are computed, again 8 elements at a time by summing up the column sums for the 3 adjacent rows and finally by multiplying it with 1/9. The algorithm used in this version is computationally efficient and is explicitly vectorized to exploit data level parallelism.

Algorithms used in 1(a) and (b) are both data parallel. However, transforming (a) to (b) automatically is non-trivial. Further, we found that LLVM fails to auto-vectorize the scalarized, buffer version of 1(b). This shows that considerable human insight goes into developing work efficient hand-vectorized data parallel algorithms while compiler auto-vectorization is not always reliable.

However, once code is written using vector intrinsics, compilers skip analyzing vector code to enable further vectorization. The assembly code generated for the SSE2 implementation when targeted to a Skylake processor with AVX-512 extensions still produces only SSE2 code (Figure 2). Even though the SSE2 version is fast,

```
for (i = 1; i < H - 1; ++i)
      for (j = 1; j < W - 1; ++j)
      {
          dst[i][j] = 1/9 * (in[i-1][j-1] + in[i-1][j] + in[i-1][j+1]
            + in[i][j-1] + in[i][j] + in[i][j+1]
            + in[i+1][j-1] + in[i+1][j] + in[i+1][j+1])
      }
```

(a) scalar code

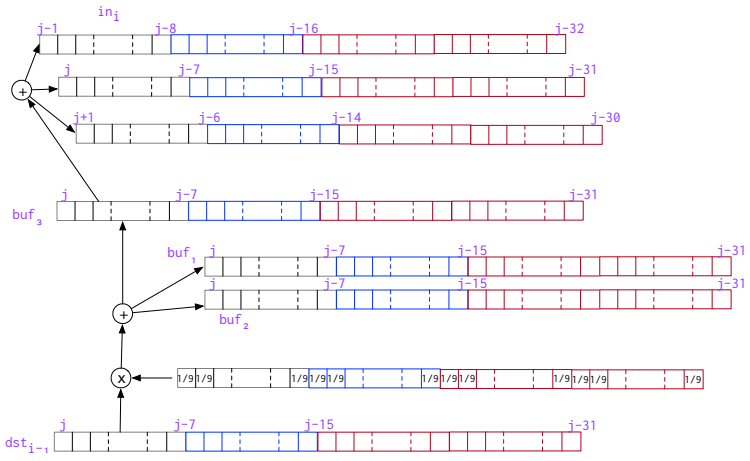
```
#define A 8
#define F (1 << 16)/9
__m128i div9 = _mm_set_epi16(F,F,F,F,F,F,F,F);

uint16_t colsum[3 * W];
__m128i * buf1 = &colsum[0 * W];
__m128i * buf2 = &colsum[1 * W];
__m128i * buf3 = &colsum[2 * W];

//code to compute column sums for first two rows in buf1, buf2

for (i = 2; i < H; ++i){
      for (j = 1; j < W - 1; j += A){
          a0 = _mm_loadu_si128(in[i][j-1]);
          a1 = _mm_loadu_si128(in[i][j]);
          a2 = _mm_loadu_si128(in[i][j+1]);
          buf3[j/A] =  _mm_add_epi16(a0, _mm_add_epi16(a1,a2)));
          dst[i - 1][j] =  _mm_mulhi_epu16(div9,
             _mm_add_epi16(buf1[j/A]
             _mm_add_epi16(buf2[j/A],buf3[j/A])));
      }
     //swap buffer colsums for next iteration
      __m128i * temp = buf1;
      buf1 = buf2;
      buf2  = buf3;
      buf3 = temp;
}
```

(b) hand-vectorized code (SSE2)



(c) revectorized (AVX2, AVX-512)

Figure 1: (a) scalar code and (b) simplified hand-vectorized SSE2 code for `Meanfilter3x3` from Simd image processing library. (c) Computation Graph for the revectorized versions targeting AVX2 (256 bits) and AVX-512 (512 bits) for the inner loop of SSE2 version. Note that handling of boundary conditions and type conversions are omitted for clarity of presentation.

it cannot fully utilize the wider vector instructions available in modern processors with AVX2, AVX-512 support. Essentially, hand-vectorized code loses performance portability, and programmers have to rewrite the same code using newer vector instructions to exploit wider vector instructions.

REVEC adds performance portability to hand-vectorized code by automatically retargeting it to vector instructions of higher vector width, whenever available. Figure 1(c) shows the computation graphs of widened instructions by REVEC for the inner loop of the SSE2 version. It uses vector intrinsic equivalences found across vector instruction sets and vector shuffle rules to widen the vector width of already vectorized code by merging two or more narrow vector width instructions akin to SLP vectorization [12]. For example, to form AVX2 and AVX-512 instructions for the `MeanFilter3x3` example, two and four SSE2 instructions are merged respectively. Given the SSE2 implementation of `MeanFilter3x3` found in the Simd library, REVEC achieves a 1.304× speedup when retargeting to AVX-512.

REVEC is developed as an LLVM IR level transformation pass and does revectorization transparently without human involvement. Figure 2 shows the LLVM IR and corresponding x86-64 assembly instructions when targeting AVX2 and AVX-512 instruction sets under REVEC. In comparison, stock LLVM compiler which does not revectorize vectorized code generates only SSE2 assembly instructions. We also compiled the SSE2 implementation under GCC5.4 and ICC and found that none of the compilers were able to revectorize the code, with similar limitations to LLVM.

3 REVEC OVERVIEW

Figure 3 shows the high level overview of REVEC compiler transformation pass. It first performs preprocessing transformations – loop unrolling and reduction variable splitting – which expose opportunities for revectorization (Section 4). REVEC then transforms the code to use vector instructions of higher width using a technique akin to bottom-up SLP vectorization [26], but specialized to handle vector instructions.

REVEC first finds adjacent vector stores and vector ϕ-node instructions [25] of same data type in a basic block. These act as the initial *revectorization packs*, which are bundles of vector instructions which can be merged into a single vector instruction of higher vector width. Starting from the initial packs which act as roots, REVEC next builds a *revectorization graph* following their use-def chains (Section 5). REVEC adds packs to the graph recursively until it terminates with vector instructions which are not mergeable. REVEC uses equivalences found among vector intrinsics (Section 7) and vector shuffle rules (Section 6.4) to determine whether two intrinsics can be merged.

Next, REVEC determines the profitability of the revectorization graph using a cost model. If revectorization is profitable, the graph is transformed to use vector instructions of higher width. REVEC continues to iteratively revectorize until all roots are exhausted for a given basic block.

Notation. We implement REVEC as a middle end compiler pass in the LLVM compiler infrastructure to transform LLVM IR instructions. In subsequent sections, we use the LLVM IR vector notation

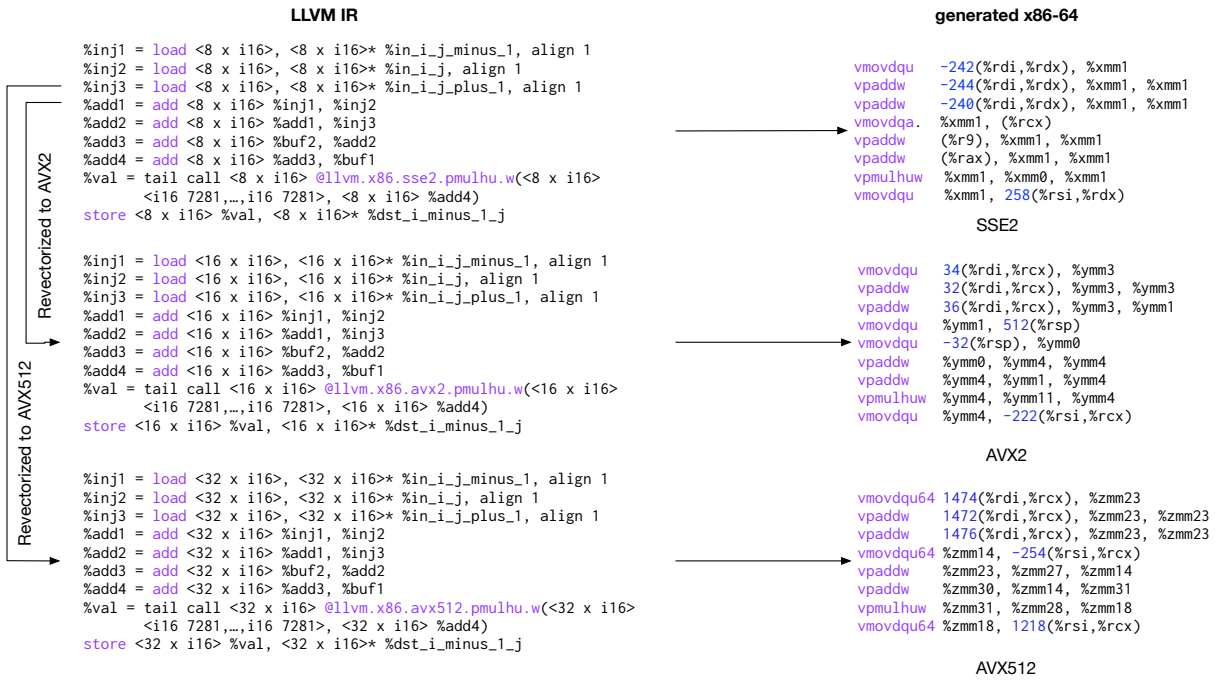


Figure 2: LLVM IR and x86-64 assembly code snippets for the inner loop of original SSE2 implementation of `MeanFilter3x3` and revectorized output targetting AVX2 and AVX-512 instruction sets.

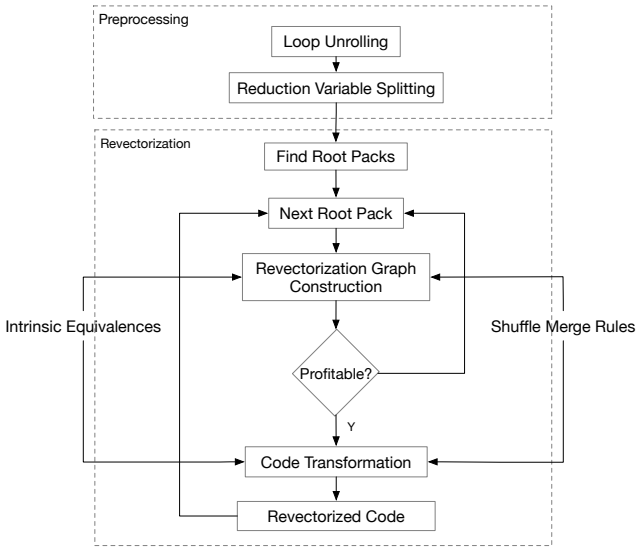


Figure 3: Overview of the Revec compiler transformation pass.

to explain the functionality of Revec. LLVM vector types are represented as `v = <m x ty>`, where vector type `v` contains `m` elements of scalar type `ty`. A generic LLVM IR vector instruction has the following form.

```
out = op <m x ty> opnd1, opnd2, ..., opndn
```

`op` is the opcode of the instruction, `opnd1` up to `opndn` are operands and `out` is the output value of the instruction. Note that output type is made explicit in the instruction, whereas operand types are omitted. Operand types are made explicit in instances when it is not clear from the context.

We call vector instructions which do not use the complete data width of the processor as *narrow vectors*, and these are widened through revectorization. We denote a revectorization pack consisting of $I_1, I_2, ..., I_n$ vector instructions by $\{I_1, I_2, ..., I_n\}$. Also we define *vectorization factor* to be the number of vectors packed together in a revectorization pack.

Running Example. We use the example shown in Figure 5 to illustrate Revec's functionality when targeting a processor with AVX2 instructions. The code shown in Figure 5(a) widens the data of the `in` array from 16 bits to 32 bits using vector shuffle instructions and then adds a constant to each element. Finally, the added values are downcasted back to 16 bits and are stored in the `out` array. All operations are written using SSE2 vector intrinsics.

4 PREPROCESSING TRANSFORMATIONS

Revec performs two preprocessing transformations to expose opportunities for revectorization: *Loop Unrolling* and *Reduction Variable Splitting*.

4.1 Loop Unrolling

Revec first unrolls inner loops containing vector instructions to expose opportunities for merging narrow vector instructions. First, Revec traverses each basic block within a given inner loop and

finds all vectorized stores. These stores are next separated into mulitple store chains, where each chain contains a set of vectorized stores which access adjacent memory locations. REVEC considers each store chain, beginning with the chain with the fewest number of stores, to update the unroll factor for the inner loop.

If the cumulative width of a given store chain (SCW = size of a store × number of stores) is less than the maximum vector width of the processor (VW), then there is an opportunity for revectorization. For such chains, REVEC checks if unrolling leads to a longer chain of consecutive stores by checking if the symbolic address of the tail of the store chain plus the size of a store is equal to the symbolic address of the head of the store chain in the next iteration. Symbolic address information is available through the Scalar Evolution pass of LLVM. If this condition is true, unrolling leads to a longer store chain. Then, the Unroll Factor (UF) is updated:

$$\text{UF} = \max\left(\frac{\texttt{lcm}(\text{VW}, \text{SCW})}{\text{SCW}}, \text{UF}\right) \quad (1)$$

REVEC initializes UF = 1 and iterates through all store chains in basic blocks of an inner loop to update its unroll factor.

Next, REVEC checks if there are any vector ϕ-node instructions that are part of a reduction chain. ϕ-nodes are used in Single Static Assignment [25] based IRs to select a value based on the predecessor of the current basic block. If any such ϕ-nodes are found, REVEC updates the unroll factor analogous to equation 1, replacing SCW with the width of the ϕ-node instruction.

Finally, REVEC unrolls the loop using this unroll factor. Figure 5(b) shows the unrolled LLVM IR for the code shown in Figure 5(a). Since there is only one 128 bit store within the loop, the unroll factor for an AVX2 processor is 2.

4.2 Reduction Variable Splitting

Reduction computations involve accumulation of a set of values into one reduction variable using a reduction operation. For example, consider the code snippet in Figure 4(a), where values of array `in` are added together into the reduction variable `R`. Reductions are not explicitly parallelizable and hence are not explicitly revectorizable. However, provided that the reduction operation is associative, we could use multiple accumulators in parallel to perform reduction of different parts of the data and finally accumulate the results into the original variable. To achieve this data partitioning, we introduce multiple independent reduction variables which accumulate results of different parts of the data making it amenable to revectorization.

Concretely, after loop unrolling, REVEC first identifies reduction variables with associative operations such as maximum, addition etc. whose values are only consumed outside the loop in which the reduction is computed. Next, for each update of the reduction, REVEC introduces a new reduction variable by *splitting* the original variable. For example, in Figure 4(c)Ⓑ, REVEC introduces two independent reduction variables, `R1` and `R2`, in place of the two occurences of `R` within the loop. Finally, REVEC emits cleanup prologue Ⓐ and epilogue Ⓒ code outside the loop, to first initialize the reduction variables and then to do the final extraction and accumulation of values of newly introduced reduction variables into the original reduction variable. Now, the reduction computation inside the loop is revectorizable. Figure 4 shows the final revectorized reduction code, where `R1` and `R2` values are computed in parallel.

5 REVECTORIZATION GRAPH CONSTRUCTION

REVEC first finds the initial revectorization packs which form the roots of the revectorization graph. Then, REVEC recursively builds a graph of packs by following the use-def chains of the instructions at the frontier of the graph.

5.1 Finding Root Packs

For each basic block, REVEC first finds chains of vector store instructions with adjacent memory accesses and chains of ϕ-node vector instructions of same data type. These two types of instructions are generally part of use-def chains with high computational workload and hence are suitable candidates to seed revectorization. Store instructions act as terminal instructions which write back values to memory after computation and ϕ-node instructions act as entry points of a basic block with reductions. REVEC forms initial revectorization packs from these chains.

For a target with maximum vector width VW, and a chain of vector seeds with operand width SW, the maximum number of vector instructions that can be packed together is $VF_{\max} = \frac{VW}{SW}$. REVEC takes chunks of $VF_{\max}$ instructions from each vector chain and forms the initial revectorization packs. These packs act as the roots of their respective revectorization graphs. For example, the two stores in the LLVM IR shown in Figure 5(b) are packed to form the root of the revectorization graph in Figure 5(c).

5.2 Building the Graph

REVEC grows the graph by following the use-def chains of packs at the frontier of the graph similar to bottom-up SLP [26]. REVEC iterates through all operands of a pack in order. If their definitions can be merged into vectors of higher width, REVEC adds the packs for the operands as new nodes of the graph with an edge connecting the current pack and the operand pack. The graph is built recursively traversing the operands of already formed packs. Expansion stops when all packs in the frontier have vector instructions which are not mergeable. Two or more instructions in a revectorization pack can be merged if the following conditions are met:

- Isomorphic: They perform the same operation on operands of the same data type and return vector values of the same data type, if any. REVEC handles both vector intrinsics which are lifted to LLVM vector IR form (e.g., Figure 5(c) Pack Ⓒ) and vector intrinsics which remain as opaque LLVM intrinsic functions (e.g., Figure 5(c) Pack Ⓓ).
- Independent: Their operands should be independent of the results they produce.
- Adjacent Memory Accesses: If instructions access memory, their accesses should be adjacent (e.g., Figure 5(c) Pack Ⓐ).
- Vector Shuffles: Special shuffle rules (Section 6.4) are applied to merge vector shuffles and to grow the graph beyond shuffles (e.g., Figure 5(c) Pack Ⓑ).

In general, a revectorization graph is a Directed Acyclic Graph (DAG), where a single pack can be used by multiple other packs. Figure 5(c) shows the complete revectorization graph for our running example. Note that the packs are shown in the upper half of

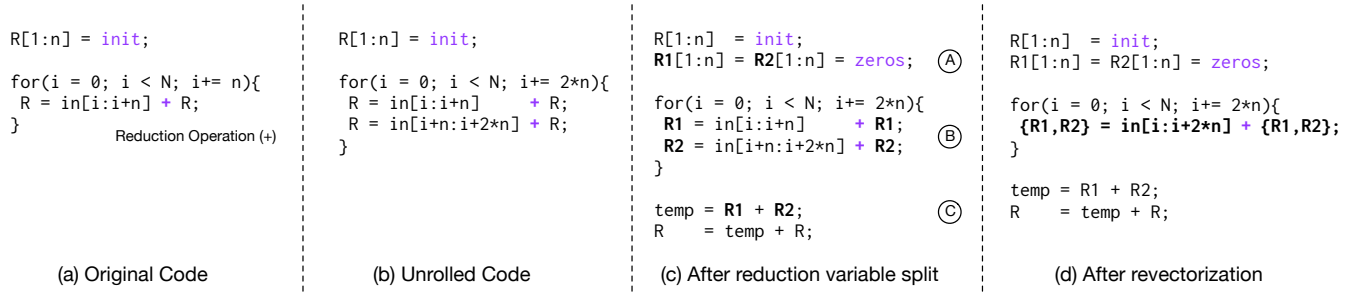


Figure 4: (a) Code example with a reduction (b) Code after the loop is unrolled twice (c) Code after reduction variable R is splitted into R1 and R2 (d) New independent reduction variables R1 and R2 are revectorized.

the nodes, whereas the bottom half shows the revectorized LLVM IR instruction after code transformation (Section 6).

6 CODE TRANSFORMATION

Given a revectorization graph, Revec first decides whether it is profitable to transform the code to use wider vectors using a static cost model. Finally, Revec revectorizes graphs which are deemed profitable.

6.1 Profitability Analysis

With an additive cost model, Revec sums the benefit of revectorizing each pack in a graph to the find the cumulative benefit of the whole strategy. Replacing multiple narrow vector instructions with a single instruction is typically profitable, evaluated by querying the LLVM `TargetTransformInfo` interface. However, gathering nonrevectorizable packs or non-adjacent loads at the leaves of the revectorization graph has overhead. Further, if a pack contains narrow vectors that are used out-of-graph, Revec must emit a subvector extract instruction. For a particular graph, Revec only transforms the code if the total benefit of revectorizing packs is greater than the total gather and extract cost.

6.2 Revectorizing the Graph

Transformation happens recursively while traversing the graph starting from the roots. Revec invokes the RevectorizeGraph routine (Algorithm 1) with initial revectorization packs at the roots of the graph.

If the current pack is not mergeable to a wider vector instruction, Revec gathers its constituents using a tree of vector shuffle instructions (Section 6.5). If it is revectorizable, Revec handles revectorizing vector shuffle instructions as detailed in Section 6.4. For other vector instructions, Revec uses a generic widening strategy (Section 6.3) covering both vector instructions which are lifted to LLVM vector IR form and instructions which remain opaque LLVM vector intrinsics. The tree is traversed depth first to generate revectorized values of children (operands) first and these values are used to generate the revectorized values for the current pack. `IntrinsicMap` is an auto-generated map used for widening opaque LLVM vector intrinsics, which maps from narrow vector intrinsics to wider vector intrinsics at various vectorization factors. Generation of `IntrinsicMap` is detailed in Section 7.

Algorithm 1: RevectorizeGraph(Pack, IntrinsicMap)

```
if !IsMergeable(Pack) then
    return TreeGather(Pack) ;                    // Section 6.5
else
    if ContainsShuffles(Pack) then               // Section 6.4
        return RevectorizeShuffle(Pack)
    V = φ
    for ChPack ∈ Children(Pack) do
        V.add(RevectorizeGraph(ChPack, IntrinsicMap))
    if ContainsIntrinsics(Pack) then             // Section 6.3
        return RevectorizeIntrinsic(Pack, V, IntrinsicMap)
    else
        return RevectorizeVectorIR(Pack, V)
```

6.3 Generic Widening of Vectors

Revec widens both packs of vector intrinsics that have been lifted to high level IR and packs of vector intrinsics which remain as opaque calls. For instance, LLVM lifts the intrinsic `_mm_add_epi16` to the vector IR instruction `add <8 x i16>`, as the semantics of the intrinsic and IR instruction have a known correspondence. However, many intrinsics have complex semantics that do not map to simple compiler IR. For example, LLVM propagates the intrinsic `_mm_packus_epi32` to the backend with the LLVM IR intrinsic `@llvm.x86.sse41.packusdw`.

Widening Lifted IR. If the i-th instruction in a pack of p isomorphic vector IR instructions has the generic form,

$$\texttt{out}_i = \texttt{op <m x ty>}\ \ \texttt{opnd}_{i1}, \texttt{opnd}_{i2}, \ldots, \texttt{opnd}_{ik}$$

the widened instruction becomes,

$$\texttt{out} = \texttt{op <(p*m) x ty>}\ \ V_1, V_2, \ldots, V_p$$

where $V_1, V_2, ..., V_p$ are revectorized values of the operand packs (children of the original pack in the revectorization graph). Note that the opcode is reused. Operands in different lanes need not have the same type. However, as we build packs recursively from vector operands, only instructions that yield vectors are widened. Scalar packing is more appropriate for the scalar SLP autovectorizer.

Widening Intrinsic Calls. For a pack of isomorphic intrinsic calls, Revec queries `IntrinsicMap` to find a wider vector intrinsic. Revec then transforms the pack by first packing the operands as before, then emitting a single intrinsic call to the widened conversion found.

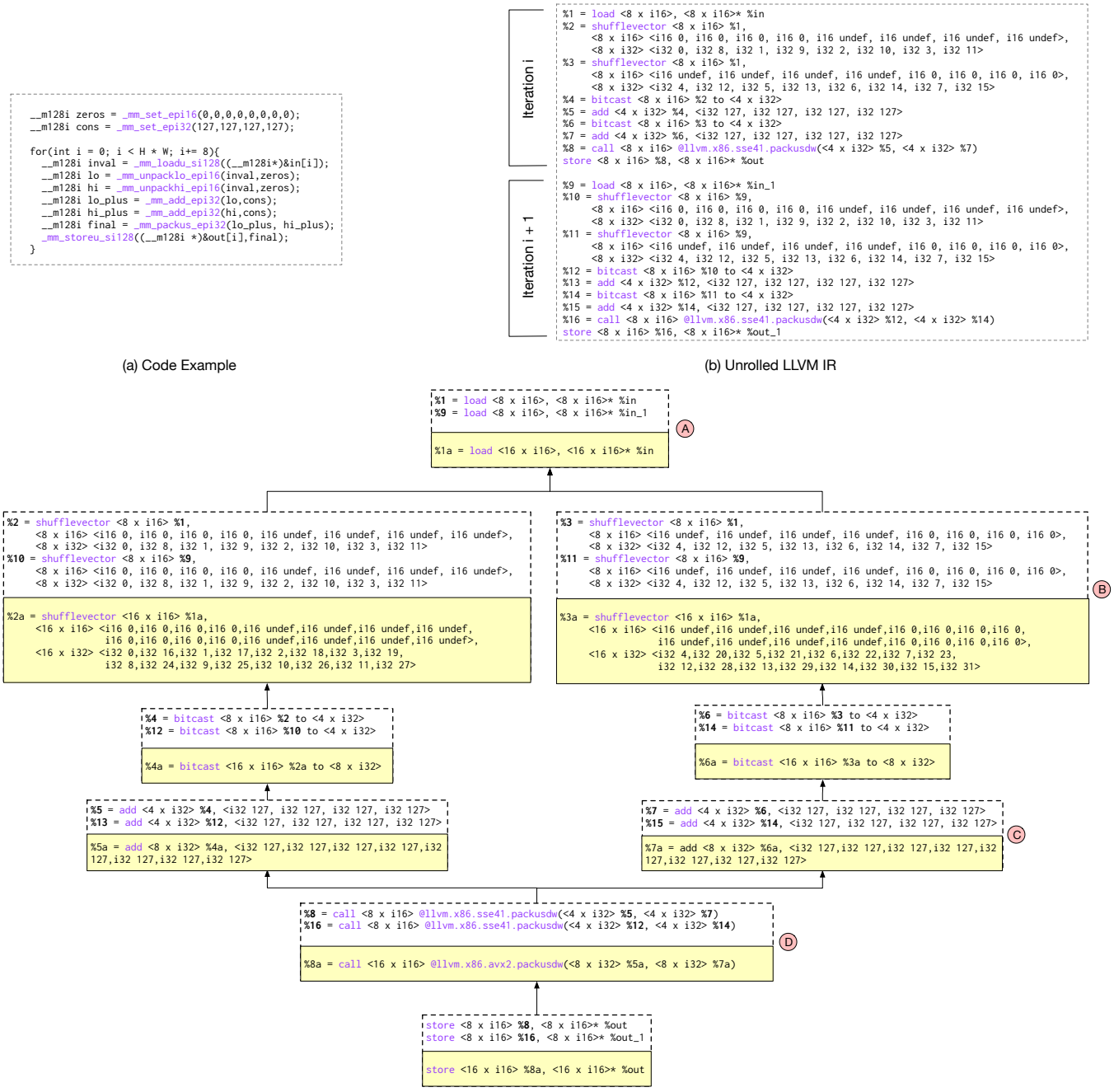


Figure 5: (a) Code example written using SSE2 vector intrinsics (b) LLVM IR after unrolling on a server with AVX2 instructions (c) Revectorization graph for the IR where the top half of the node shows the revectorization pack associated with that node and the bottom half shows the transformed IR after revectorization targeting AVX2.

Figure 5(c) pack Ⓓ illustrates opcode substitutions for the LLVM intrinsic `@llvm.x86.sse41.packusdw`.

6.4 Transforming Packs of Vector Shuffles

A vector shuffle is a high level instruction that reorders scalars from lanes of two vector operands based on a constant indexing mask. A shuffle generalizes vector selects, permutations with arbitrary

masks, operand interleaving, and two-operand gathers. In the shuffle `s = shuffle a, b, m`, let `a` and `b` be source vector operands of the same type, `m` be a length l constant vector, and the shuffled value `s` be a length l vector created by indexing into the concatenation of `a` and `b` at each index in m. During code transformation, Revec matches shuffle packs to known operand patterns to create efficient revectorized shuffles.

In general, consider a pack of vector shuffles:

s_1 = shuffle a_1, b_1, m_1 ...

s_p = shuffle a_p, b_p, m_p

With shuffle merge rules (Figure 7), Revec transforms this pack into a single wide shuffle `S = shuffle A, B, M`, where `A` and `B` are some combination of all a_i and b_i, and `M` is a statically determined constant vector. In Figure 6, we show that four shuffle patterns match all packs encountered when revectorizing to AVX2, and 92.9% of packs when revectorizing to AVX-512. While a gather or a generic lane widening strategy, Pattern D, could be applied to all packs, Patterns A, B, and C allow Revec to emit fewer or less costly shuffles.

Pattern A: Sequential Subvector Extraction. When $a_1 = a_2 = ... = a_p$, $|a_i| = n$, $|m_i| = \frac{n}{p}$, and concat $m_1\ m_2\ ...\ m_p = \{0, 1, ..., n-1\}$, the narrow shuffles extract all adjacent $\frac{n}{p}$ length subvectors of source a (operands b_i are unused). Revec emits `S` = a_1 with no shuffling and erases the pack. Sequential subvector extraction is used by programmers to interleave subvectors from different source registers. By erasing the extractions, Revec can interleave the full source vectors elsewhere in the graph.

Pattern B: Permutations of Identical Operands. If $a_1 = a_2 = ... = a_p$, and $b_1 = b_2 = ... = b_p$, then Revec only widens the mask by concatenation. The source operands need not be packed as a shuffle mask can repeat indices.

Pattern C: Mergeable Constant Operands. Shuffles often include a constant vector as a source operand. For instance, the low and high halves of a source vector a can be separately interleaved with a constant vector. In LLVM, the unused halves of the constant vector are in practice set to be undefined. If constant operands are non-identical but can be elementwise merged as in Figure 7(C), and an operand is shared between narrow shuffles, Revec merges the constant and concatenates the masks.

Pattern D: Generic Lane Widening. Failing the above conditions, Revec attempts to form pack $\alpha = \{a_1, a_2, ..., a_p\}$ from left operands and pack $\beta = \{b_1, b_2, ..., b_p\}$ from right operands. The following local analyses indicate that a lane widening, vertical packing strategy is least costly:

- The left or right operand pack contains all constant vectors. Revec transforms the constant pack into a higher width constant vector.
- The left operands are identical or the right operands are identical. Revec can emit an efficient splat or broadcast instruction for this pack.
- All narrow masks are equal. The pack is isomorphic.

Revec generates a reindexed shuffle mask `M` by adding constant displacements to elements of each narrow mask m_i. Operands are displaced proportionally to their position in the pack. The constant displacement for indexes of narrow shuffle i that correspond to the left operand (i.e indexes $< n$) is $\sum_{j=1}^{i-1} |a_j| = n * (i-1)$. Similarly, the constant displacement for indexes that correspond to values of the right operand is $\sum_{j \neq i} |a_j| + \sum_{j=1}^{i-1} |b_j| = n(p-1) + n(i-1)$.

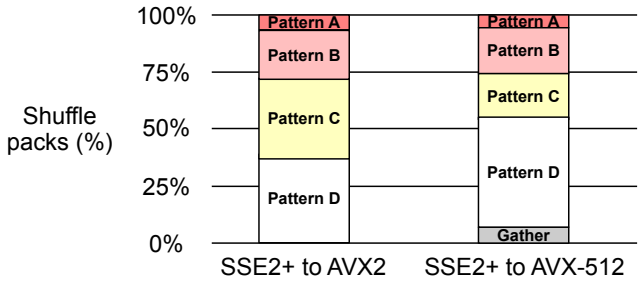


Figure 6: Revec's shuffle patterns match all shuffle packs encountered in benchmarks when revectorizing to AVX2, and 92.9% of shuffle packs encountered when revectorizing to AVX-512.

Gathering General Shuffles. For shuffles with arbitrary non-constant operands, 2^{p-1} packs can be formed. As local packing decisions lead to revectorization graphs of different profitabilities, Revec is unable to make a globally optimal decision without a search. In the general case, narrow shuffles are gathered as described in Section 6.5. Any gathered shuffle pack will be non-isomorphic, as isomorphic shuffles have the same mask and match patterns B, C or D. Revec gathered no shuffle packs when revectorizing SSE2+ benchmarks to AVX2, and 7.1% when revectorizing to AVX-512.

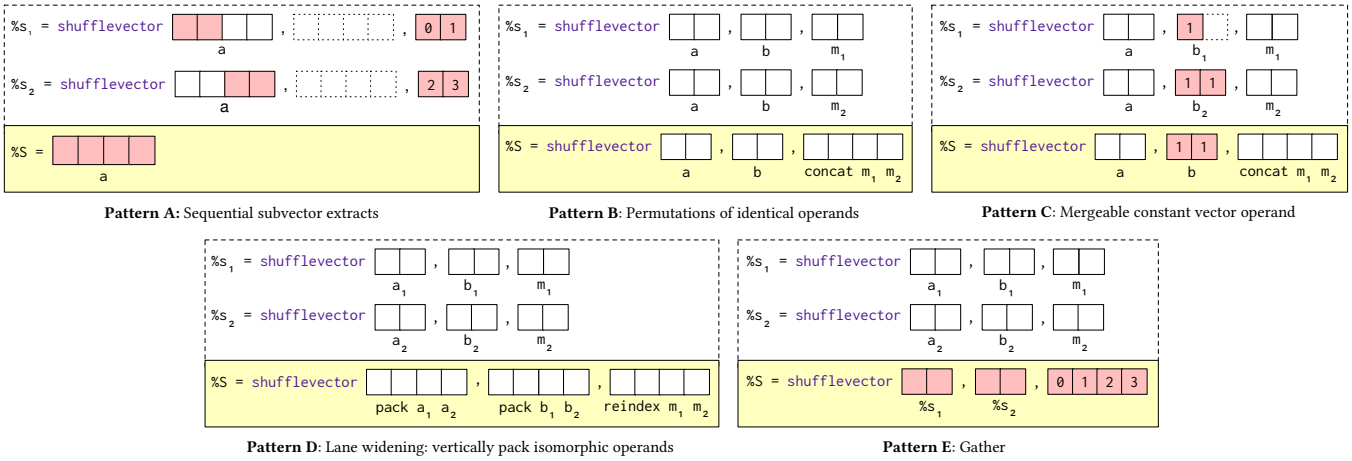
6.5 Gathering Non-mergeable Vectors

Revec gathers packs at the leaf nodes of the graph if they are not mergeable. This is realized by using a tree of vector shuffles, where at each level it concatenates two vectors of the same width to form a vector of twice the data width.

Assume we want to merge a pack $\{I_1, I_2, ..., I_{2n}\}$ of type `<m x ty>` instructions. The gathered wide vector of type `<(2n*m) x ty>` is formed using the tree of vector shuffles shown in Figure 8. Vectors formed at level j have $2^j m$ elements. The mask used for each shuffle at level j, M_j = <0, 1, 2, ..., $2^j m$>. The height of the tree is $k = \log 2n + 1$.

7 DISCOVERING INTRINSIC CONVERSIONS

In Section 6.3, we noted that Revec queries an `IntrinsicMap` data structure to transform packs of opaque vector intrinsic calls. Revec automatically generates intrinsic substitution candidates using explicit enumeration and pre-populates the `IntrinsicMap` structure used in Algorithm 1. When `IntrinsicMap(intrin, p) = wide_intrin`, an isomorphic pack of p calls to `intrin` has the same result as a single call to `wide_intrin` with vertically packed operands. In our implementation of Revec, a canonical representation of each narrow intrinsic is mapped to several translations of higher width vector instructions at different vectorization factors p. We limit vectorization factors to powers of two.



Pattern A: Sequential subvector extracts

Pattern B: Permutations of identical operands

Pattern C: Mergeable constant vector operand

Pattern D: Lane widening: vertically pack isomorphic operands

Pattern E: Gather

Figure 7: Shuffle pack transformation patterns. Revec avoids unnecessarily widening operands when applying patterns A, B and C. Pattern D, a lane widen, is applied when there is similarity between operands vertically in the pack, while narrow vectors are gathered only when an optimal packing decision cannot be made locally. Shaded operand elements are indexed by a shaded shuffle mask index.

We adopt a test case fuzzing approach for generating equivalences, with 18,000 randomly generated inputs and combinatorially generated corner-case inputs, inspired by test case generation in [6]. Test cases are byte strings of a specified length that initialize operands of a specified vector data type. To augment randomly generated test cases, we create corner-case byte string sequences such as 0000…, 0101…, 1010…, and 1111…. This helped Revec to significantly reduce erroneous equivalences, and is consistent with the methodology in [6].

For each vector intrinsic available on the target and each vectorization factor $1 \leq p \leq 8$, we generate and compile a testbed in LLVM IR that invokes the intrinsic p times. Input test cases are passed as arguments of these intrinsics, and outputs are saved after execution. Testbeds are evaluated on a server with AVX-512 support (Section 9.1). If a set of testbeds produce the same output when given the same input for all test cases, each pair of testbeds in the set corresponds to a conversion. After filtering redundant conversions, the remaining conversions are added to `InstrinsicMap`.

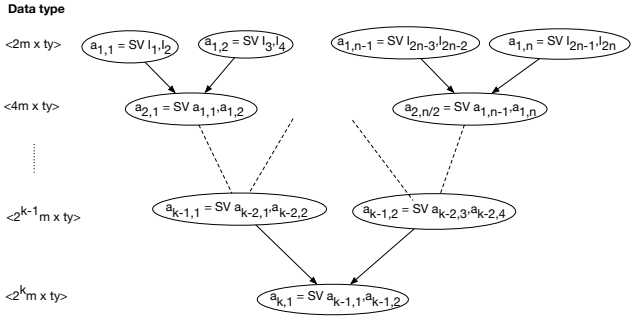


Figure 8: Tree of vector shuffles (SV) to gather packs which are not mergeable.

Automated enumeration discovered 53 SSE-series to AVX1/2 intrinsic conversions, 33 AVX1/2 to AVX-512 conversions, and 19 SSE-series to AVX-512 conversions. For instance, the SSE4.1 intrinsic `_mm_packus_epi32` found in pack Ⓓ of Figure 5 has a 2-to-1 conversion to `_mm256_packus_epi32` and a 4-to-1 conversion to `_mm512_packus_epi32`. Additionally, enumeration discovered several conversions that upgrade operand widths within the same instruction set.

8 IMPLEMENTATION

We implemented Revec as a LLVM IR-level compiler pass extending LLVM v7.0.0 (pre-release, commit hash 71650da2). In addition, we add a loop analysis pass to determine inner loop unroll counts prior to revectorization. We execute the passes through Clang (commit hash 3ed5b23) as a frontend to compile C++ benchmarks. Revec transforms each source function immediately after the built-in SLP scalar autovectorization pass.

9 EVALUATION

We evaluate the performance impact of Revec on real world performance critical kernels from video compression, integer packing, image processing, and stencil computation domains which contain heavily hand-vectorized routines. In Section 9.2, we examine Revec's impact on integer unpacking routines [30] in the FastPFor integer compression library. In Section 9.3, we evaluate Revec's imapact on important kernels from the x265 video encoding library [10]. Finally, in Section 9.4, we evaluate Revec's impact on popular image processing and stencil kernels from the Simd image processing library [9].

9.1 Experimental Setup

Benchmark kernels are run on a Google Cloud platform server with 4 vCPUs and 8GB of RAM. The benchmark server had Intel Xeon Skylake cores running at 2.00 GHz, and supported Intel's

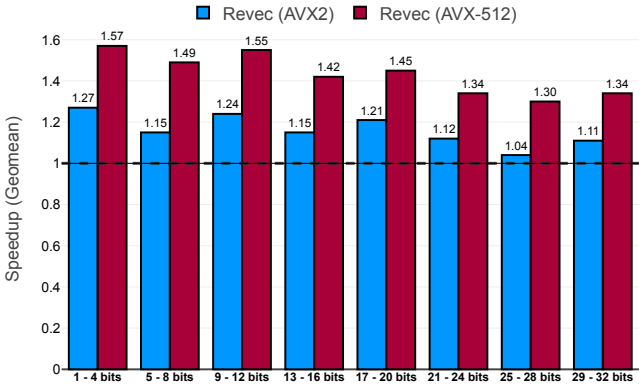


Figure 9: Revec speedups on 32 SSE4 horizontal bit unpacking kernels from SIMD-Scan and FastPFor when retargetted to AVX2 and AVX-512.

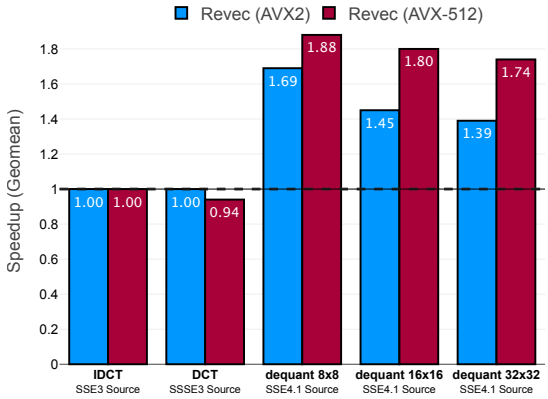


Figure 10: Revec speedups on hand-written SSE-series kernels used in the x265 video encoding library.

AVX-512 instructions (with the the F, CD, VL, DQ and BW instruction sets). We target AVX2 by disabling AVX-512 features with the compilation flags `-O3 -march=native -mno-avx512f -mno-avx512pf -mno-avx512er -mno-avx512cd`. We simply compile with `-O3 -march=native` to target AVX-512. Test code measures absolute runtimes, and speedups are computed with respect to the runtime under compilation by stock Clang v7.0.0 (commit 3ed5b23) with the same command line flags, with no preprocessing or revectorization.

9.2 Benchmark: SIMD-Scan Integer Unpacking

SIMD-Scan [29, 30] proposes an efficient horizontal codeword-to-integer unpacking algorithm accelerated by 128-bit SIMD processors. Integer array packing and unpacking are widely used in search engine and relational database software to store data as codewords, so extensive effort has been spent on optimizing these routines [15]. Decoding packed codewords into integers can be a bottleneck in lookup procedures, such as database column scans. The FastPFOR integer compression library [15] implements 32 separate SSE4 unpacking kernels from SIMD-Scan for different integer bitwidths. Specialized kernels are used for each integer bitwidth as codewords packed to different bitwidths span SIMD register lanes differently.

In Figure 9, we report the speedups that Revec achieves on these kernels when retargeting to the AVX2 and AVX-512 instruction sets. While the authors of [15] implement a separate function for each bitwidth, we aggregate speedups into groups of 4 for clarity.

Across the 32 SSE4 kernels, Revec nets a 1.160× geometric mean speedup over Clang when targeting AVX2. On the same experimental server, with AVX-512 instruction sets enabled, Revec nets a 1.430× geometric mean speedup.

Without Revec, Clang is unable to leverage newly available instruction sets given hand-vectorized code written to target the SSE4 instruction set. In fact, on this benchmark, we observe a slight slowdown from enabling AVX-512 instructions with stock Clang. However, Revec demonstrates the potential for automatic performance scaling of real-world code.

In fact, in later work, the SIMD-Scan authors propose a further optimized, AVX2-accelerated database column scan tool, AVX2-Scan, with horizontal unpacking being the key routine [29]. The authors report that AVX2-Scan is "around 30% faster than SIMD-Scan". Revec achieves considerable proportion of this proposed speedup automatically without manual intervention.

9.3 Benchmark: x265 Video Encoding Library

x265 [10] is the most widely used HEVC/H.264 video encoding library. The library includes intrinsic-rich C++ routines for computing an inverse discrete cosine transform (IDCT) and discrete cosine transform (DCT) over small image blocks, and for image dequantization with scaling, accelerated with SSE-series intrinsics.

In Figure 10, we note Revec's speedups on these routines. Overall, Revec achieves geometric mean speedups of 1.145× and 1.195× for AVX2 and AVX-512 targets, respectively.[1]

However, we observe a slight slowdown on the x265 DCT implementation for AVX-512 targets. Revec encounters packs of 4 SSSE3 phadd intrinsics. No 4-to-1 instrinsic equivalences are known for this instruction, as phadd is absent from AVX-512. While Revec currently gathers this pack with 3 shuffles (Section 6.5) such that an SSE-AVX switching penalty is incurred, `IntrinsicMap` contains a 2-to-1 equivalence to the AVX2 phadd intrinsic. The pack of 4 SSSE3 phadd intrinsics could be split and partially revectorized to use 2 AVX2 phadds, gathered with a single shuffle.

Further, the AVX2-retargetted DCT would benefit from horizontal packing of operands of an SSSE3 phadd pack followed by data reorganization, as opposed to lane widening (vertical packing). Lane widening packs operands generated by different intrinsics that must be gathered and terminates the tree; horziontal packing would prevent this. Revec's greedy algorithm misses this opportunity, however a search across packing strategies can be used uncover such uncommon opportunities.

[1] Geometric mean taken over 3 numbers: DCT speedup, IDCT speedup, and aggregated dequantized scaling (dequant) speedup.

9.4 Benchmark: SIMD Library

SIMD [9] is a library of heavily hand-vectorized image processing and generic stencil kernels, with specialized implementations targetting different vector instruction sets. We group 216 SSE2+ kernels (168 SSE2, 47 SSE3/SSSE3, and 1 SSE4.1 kernels) based on their functionality in Table 1.

Each SSE-series kernel we extract and benchmark has an analogous AVX2 implementation. Hence, we can compare speedups of automatic revectorization with speedups that human experts have achieved. Other image processing libraries such as OpenCV provide a smaller variety of SIMD kernels. Further, the Intel IPP-ICV vector computer vision package does not provide source code.

9.4.1 Impact of REVEC on Runtime Performance. We report the speedup REVEC has over SSE2+ implementations of SIMD library kernels when retargeted to produce AVX2 or AVX-512 code in Figure 11. Note that for presentation purposes, we clustered the kernels into the classes shown in Table 1. In Figure 11, the speedup for each cluster is the geometric mean speedup of all kernels within a cluster. We benchmark with test code written by the SIMD authors that executes each kernel with large inputs (i.e. 1920 x 1080 pixels and other similar sizes).

Across 216 extracted, templatized variants of SSE2+ SIMD library kernels from 2016, REVEC achieves an overall geometric mean speedup of **1.1023**× when retargeting to AVX2, and **1.1163**× when retargeting to AVX-512. In comparison, handwritten AVX2 implementations from the same commit have a 1.2673× speedup over their SSE2+ counterparts. After we extracted kernels, the SIMD library authors added handwritten AVX-512 implementations. From a December 2018 commit of the SIMD library, handwritten AVX-512 kernels had a 1.3916× speedup over SSE2+ counterparts.

We exclude 14 kernels that achieve less than a 1.01 speedup when both hand-rewritten and revectorized to AVX2 and AVX-512 — these are memory-bound functions with limited opportunity for revectorization. Nonetheless, elided kernels also compile and pass tests with neutral performance.

Table 1: Classes of image processing and stencil kernels in the Simd library

Simd library benchmark functions		
Conversions	reducegray5x5	**Statistic**
bgratogray	shiftbilinear	conditional
bgratoyuv	stretchgray2x2	statistic
bgrtobgra		
bgrtoyuv	**Filters**	**Correlation**
deinterleave	absgradientsaturatedsum	absdifferencesum
graytobgra	gaussianblur3x3	squareddifferencesum
graytobgr	laplace	
int16togray	lbp	**Misc**
interleave	meanfilter3x3	addfeaturedifference
yuvtobgra	sobel	binarization
yuvtobgr		fill
yuvtohue	**Motion detection**	histogram
	background	hog
Resizing	edgebackground	neural
reducegray2x2	interference	operation
reducegray3x3	segmentation	reorder
reducegray4x4	texture	

In addition, on 6 SSSE3/SSE2 color conversion tests in the SIMD library that have no AVX2 or AVX-512 implementations to compare against, REVEC yields a 1.0618× and 1.0632× geometric mean speedup over stock Clang when retargeting to AVX2 and AVX-512 respectively.

9.4.2 Comparison with Hand-Vectorized Kernels. While hand-written AVX2 and AVX-512 SIMD kernels overall perform better than revectorized kernels, these required considerable manual effort and expertise to write. In particular, hand-vectorization achieves >3× speedup on one kernel variant in the `resizebilinear` cluster. This large speedup is due to the usage of `_mm256_maddubs_epi16` in the AVX2 implementation, which executes vertical multiplies and horizontal additions. The SSE2 version emulates this using multiple intrinsics involving vector shuffles. Such complex semantic transformations are not currently possible under REVEC. Similarly, REVEC can only generate the horizontal vector operation `_mm256_hadd_epi32` given SSE3-onward source; SSE2 sources emulate the function. Horizontal AVX2 operations are used in other hand-written AVX2 kernels such as `reducegray4x4`.

The AVX-512 implementation of the `hog` kernel uses fused multiply and add intrinsics while the SSE2 source, written to target processors without the FMA instruction set, separately multiplies and adds. The AVX-512 `hog` and `lbp` kernels also use masked selects or sets like `_mm512_mask_blend_epi32`, whereas REVEC widens casts, logical operations, or compares that emulate masked operations.

Many of these performance gaps could be narrowed by applying peephole-style conversion rules independently of the REVEC transformation pass.

REVEC still automatically delivers an overall speedup on SIMD. Notably, the `NeuralConvert` kernel enjoys a 1.757× speedup via revectorization to AVX2, but only 1.075× via hand vectorization. In the AVX2 kernel, SIMD programmers reduced the number of unpacks/shuffles executed by using an AVX2-only zero extension intrinsic, `_mm256_cvtepu8_epi32`, at the cost of using narrower loads (`load <8 x i8>`). The SSE2 implementation uses wider loads (`load <16 x i8>`), with more data rearrangement. By revectorizing, REVEC doubles the width of the high-latency loads to `<32 x i8>`, and uses our shuffle logic to merge unpacks. Revectorized AVX2 utilizes memory bandwidth better, even though it executes more instructions, yielding a higher speedup.

10 RELATED WORK

Revectorization is inspired by compiler auto-vectorization. Also, previous works perform limited revectorization, primarily outside of compilers.

Compiler Auto-vectorization. Compilers employ two main auto-vectorization techniques, namely loop vectorization and Superword Level Parallelism (SLP) based vectorization. Loop vectorization has been implemented in compilers since the era of vector machines [1] and subsequently many vectorization schemes have been proposed which use loop dependency analysis [27]. Other loop vectorization techniques explore vectorization under alignment constraints [3], outer loop transformations [21], handling data interleavings in loops [20] and exploiting mixed SIMD parallelism [13, 31].

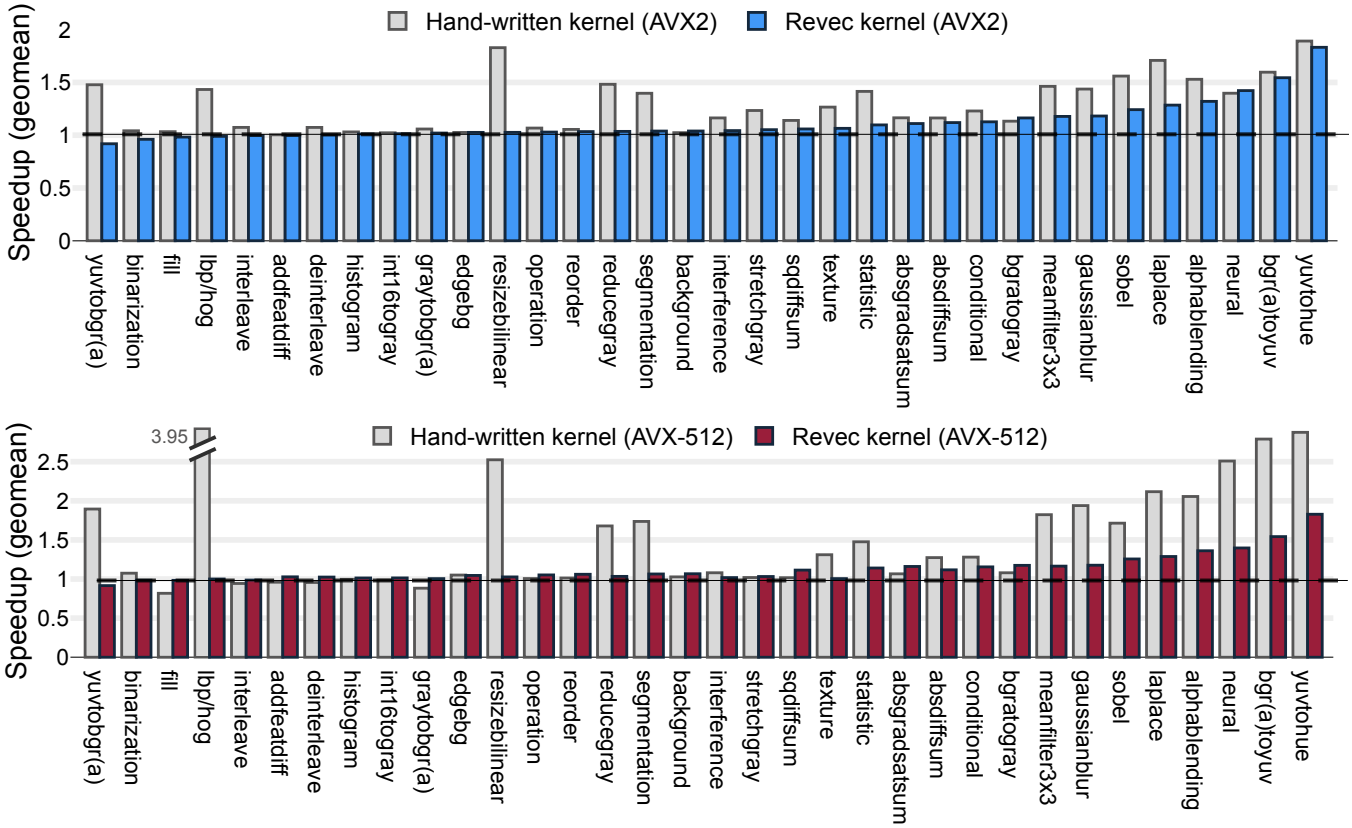


Figure 11: Revectorized and hand-written AVX2 kernel speedups (top) and AVX-512 speedups (bottom) over hand-written SSE2+ kernels for various Simd library benchmarks

Larsen [12] introduced SLP, which can capture vectorization opportunities that exist beyond loops at a much lower granularity. Revec has its roots in SLP vectorization which packs isomorphic independent statements starting from scalar loads and stores to form vectorized counterparts. In particular, Revec's transformation algorithm is inspired by a variant of SLP proposed in [26], but targets only vector instructions .

Domain specific vectorization techniques [4, 19] have been proposed in systems like SPIRAL [24] where generic auto-vectorization fails. They use platform-specific vector intrinsics in their implementations and Revec can be used to rejuvenate their performance on newer instructions sets.

Dynamic Rewriting. Dynamic rewriting of SIMD instructions has been proposed in [5, 16] to find SIMD mappings between host and guest architectures in dynamic binary translation. The Dynamic Binary Translation system proposed in [7] details a technique to widen SIMD instructions during this mapping. It only targets loops and relies on recovery of LLVM IR from the binary, which is imprecise and can lead to spurious dependencies. Compared to [7], Revec is implemented as a compiler level transformation pass and inherently has access to loop structures without the need to recover them from the binary, making Revec more precise. Further, Revec applies to loop-free segments of code, making it more general than [7] – the Simd `NeuralConvert` benchmark that Revec greatly accelerates depends on this capability.

Static Rewriting. Manilov [18] proposes a source-to-source translation system that maps intrinsic code written in one platform to another. It uses intrinsic descriptions written separately in header files for two platforms to match dataflow graphs between intrinsics to do the translation. It can be used for widening vector intrinsics as well similar to Revec. However, it requires considerable manual engineering: semantically correct descriptions should be written for each vector intrinsic in plain C code. This is tedious and error prone. In comparison, Revec automatically finds these equivalences using enumeration. Further, Revec is implemented as a compiler pass and transformation happens transparently without the need for library modification: Revec can directly benefit end-users of software. Pokam [23] proposes a system which analyzes C-like code and finds common idioms which are vectorizable and translates those to SIMD intrinsics. Revec analyzes code already written using vector intrinsics.

Libraries like [11, 28] provide abstract ways of programming SIMD operations across platforms. These systems are suitable for writing new applications, but do not deal with the portability of high performance applications already written using platform-specific vector intrinsics.

11 CONCLUSION

Programmers write code using low-level platform-specific vector intrinsics to exploit data level parallelism as much as possible. However, hand vectorization results in non-portable code, and hence programmers resort to manually rewriting code using intrinsics of newer vector instruction sets to leverage the availability of higher vector width instructions. In this paper, we introduced REVEC, a compiler technique which automatically retargets hand vectorized code to use higher vector width instructions whenever available. We showed REVEC can be used to transparently rejuvenate performance of stale hand vectorized code, thereby achieving performance portability.

ACKNOWLEDGMENTS

We would like to thank Vladimir Kiriansky and all reviewers for insightful comments and suggestions. This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Numbers DE-SC0008923 and DE-SC0018121; the National Science Foundation under Grant No. CCF-1533753; and DARPA under Award Number HR0011-18-3-0007. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Randy Allen and Ken Kennedy. 1987. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Program. Lang. Syst.* 9, 4 (Oct. 1987), 491–542. https://doi.org/10.1145/29873.29875

[2] ARM. 2013. ARM Programmer Guide. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0018a/index.html

[3] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. 2004. Vectorization for SIMD Architectures with Alignment Constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, New York, NY, USA, 82–93. https://doi.org/10.1145/996841.996853

[4] Franz Franchetti, Stefan Kral, Juergen Lorenz, and Christoph W Ueberhuber. 2005. Efficient utilization of SIMD extensions. *Proc. IEEE* 93, 2 (2005), 409–425.

[5] S. Fu, D. Hong, J. Wu, P. Liu, and W. Hsu. 2015. SIMD Code Translation in an Enhanced HQEMU. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. 507–514. https://doi.org/10.1109/ICPADS.2015.70

[6] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 474–484.

[7] Ding-Yong Hong, Yu-Ping Liu, Sheng-Yu Fu, Jan-Jan Wu, and Wei-Chung Hsu. 2018. Improving SIMD Parallelism via Dynamic Binary Translation. *ACM Trans. Embed. Comput. Syst.* 17, 3, Article 61 (Feb. 2018), 27 pages. https://doi.org/10.1145/3173456

[8] IBM. 2006. PowerPC microprocessor family: Vector/SIMD multimedia extension technology programming environments manual. *IBM Systems and Technology Group* (2006).

[9] Yermalayeu Ihar, Antonenka Mikhail, Radchenko Andrey, Dmitry Fedorov, and Kirill Matsaberydze. 2016. Simd Library for Image Processing. http://ermig1979.github.io/Simd/index.html

[10] MulticoreWare Inc. 2018. x265 HEVC Encoder / H.265 Video Codec. http://x265.org

[11] Matthias Kretz and Volker Lindenstruth. 2012. Vc: A C++ library for explicit vectorization. *Software: Practice and Experience* 42, 11, 1409–1430.

[12] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 145–156. https://doi.org/10.1145/349299.349320

[13] Samuel Larsen, Emmett Witchel, and Saman P. Amarasinghe. 2002. Increasing and Detecting Memory Address Congruence. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT '02)*. IEEE Computer Society, Washington, DC, USA, 18–29. http://dl.acm.org/citation.cfm?id=645989.674329

[14] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[15] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Journal of Software Practice and Experience* (2015).

[16] Jianhui Li, Qi Zhang, Shu Xu, and Bo Huang. 2006. Optimizing dynamic binary translation for SIMD instructions. In *International Symposium on Code Generation and Optimization (CGO'06)*. 12 pp.–280. https://doi.org/10.1109/CGO.2006.27

[17] Saeed Maleki, Yaoqing Gao, Maria J Garzar, Tommy Wong, David A Padua, et al. 2011. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 372–382.

[18] Stanislav Manilov, Björn Franke, Anthony Magrath, and Cedric Andrieu. 2015. Free Rider: A Tool for Retargeting Platform-Specific Intrinsic Functions. *SIGPLAN Not.* 50, 5, Article 5 (June 2015), 10 pages. https://doi.org/10.1145/2808704.2754962

[19] Daniel S. McFarlin, Volodymyr Arbatov, Franz Franchetti, and Markus Püschel. 2011. Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. ACM, New York, NY, USA, 265–274. https://doi.org/10.1145/1995896.1995938

[20] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of Interleaved Data for SIMD. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 132–143. https://doi.org/10.1145/1133981.1133997

[21] Dorit Nuzman and Ayal Zaks. 2008. Outer-loop Vectorization: Revisited for Short SIMD Architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 2–11. https://doi.org/10.1145/1454115.1454119

[22] Stuart Oberman, Greg Favor, and Fred Weber. 1999. AMD 3DNow! Technology: Architecture and Implementations. *IEEE Micro* 19, 2 (March 1999), 37–48. https://doi.org/10.1109/40.755466

[23] Gilles Pokam, Stéphane Bihan, Julien Simonnet, and François Bodin. 2004. SWARP: a retargetable preprocessor for multimedia instructions. *Concurrency and Computation: Practice and Experience* 16, 2-3 (2004), 303–318.

[24] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (Feb 2005), 232–275. https://doi.org/10.1109/JPROC.2004.840306

[25] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. ACM, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

[26] Ira Rosen, Dorit Nuzman, and Ayal Zaks. 2007. Loop-Aware SLP in GCC. In *Proceedings of the GCC Developers' Summit*. 131–142.

[27] N. Sreraman and R. Govindarajan. 2000. A Vectorizing Compiler for Multimedia Extensions. *Int. J. Parallel Program.* 28, 4 (Aug. 2000), 363–400. https://doi.org/10.1023/A:1007559022013

[28] Haichuan Wang, Peng Wu, Ilie Gabriel Tanase, Mauricio J Serrano, and José E Moreira. 2014. Simple, portable and fast SIMD intrinsic programming: generic simd library. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*. ACM, 9–16.

[29] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. 2013. Vectorizing Database Column Scans with Complex Predicates.. In *Fourth International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS) at VLDB*.

[30] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment* 2, 1, 385–394.

[31] Hao Zhou and Jingling Xue. 2016. Exploiting Mixed SIMD Parallelism by Reducing Data Reorganization Overhead. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. ACM, New York, NY, USA, 59–69. https://doi.org/10.1145/2854038.2854054