Padded Non-SIMD Vectorization with VeGen and PSLP

Elizabeth Koning and Zheyuan Zhang

Department of Computer Science, Univeristy of Illinois, Urbana-Champaign

May 9, 2023

1 Introduction

Manufacturers are increasing the variety of instructions available for their processors. This includes adding vector instructions. In order to take advantage of the new instruction sets, compilers must match between the code and the machine instructions.

Past work has established techniques to match to non-SIMD instructions and other techniques find close matches that can be rewritten for vectorization. We combine these two approaches. Through adding padding to VeGen, we generate an optimization that can vectorize non-SIMD code with partially matched patterns. It uses the non-SIMD, but regular vectorizing techniques of VeGen and layers on top the padding techniques of PSLP. Together, they can vectorize more specialized patterns.

2 Background

Our work builds on VeGen and borrows the idea of padding from PSLP. Each project uses a different approach on vectorization.

2.1 PSLP

PSLP [3] uses padding with Superword-Level Parallelism for vectorization in cases where the data dependence graphs are similar, but not identical. It makes the graphs isomorphic, which allows for increased vectorization. It compares the cost of the original program and the vectorized version and selects the optimal version. As shown in their paper, a large part of this includes adding padding for constants in order to allow for vectorization between variables and constants.

2.2 VeGen

VeGen [1] vectorizes non-SIMD instructions based on the patterns generated from the architecture manual. It translates the architecture manual into instruction descriptions before generating a set of patterns. Then it is able to match the patterns with the input code and vectorize the code based on the available patterns, which may be non-SIMD. In their results, they show that many more vector instructions are included with VeGen's added pass than with LLVM alone.

3 Approach

Our approach begins with the implementation of VeGen and adds a layer of padding to expand the possible vector instructions. Through this, it incorporates the benefits of both VeGen and PSLP. After other compiler optimization steps, such as constant propagation, elements of patterns may have been optimized out, and so finding the partial patterns can improve the effectiveness of VeGen. Padding is useful with SIMD instructions to improve the coverage of the vectorization by increasing the isomorphic instructions, and so we predicted

that non-SIMD instructions could benefit from adjusting the instructions to be isomorphic with the available patterns in a similar way. This increases the vectorization coverage.

3.1 Algorithm

The algorithm begins with VeGen's algorithm and implementation, and adds paddings within the framework.

3.1.1 VeGen

VeGen begins with the offline stage and first identifies the vector instructions available for the current hardware. The semantics for the instructions come from Intel's Intrinsics Guide. The core of VeGen handles the listing of the target intrinsics. Based on these intrinsics, the second part of the offline stage requires pattern matching. The offline pattern matching canonicalizes the patterns from the intrinsics in order to identify the patterns as they occur in the IR. This offline stage remains consistent between the original version of VeGen and our updated edition.

3.1.2 Adding PSLP

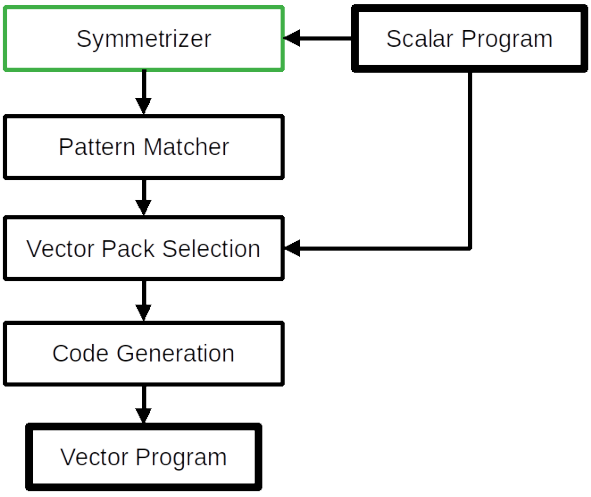

Figure 1: Diagram VeGen's workflow, adapted from [1]. Bolded boxes represent artifacts. Black boxes represent stages original to VeGen, and the green "symmetrizer" represents our added stage.

Our addition of the padding is added in the compile time stage. Once the scalar program has been interpreted as a DAG built from the vectorization seeds, the added step works to make it symmetric.

In adding the padding, the symmetricizer identifies partial matches that can be modified into full matches. The algorithm works by adding the operands to a worklist and iterating over that list. While elements remain in the list, it examines the nodes and checks for their types and positions. It identifies pairs or lists of instructions that are in symmetric positions in the DAG, but which are not of the same type. The symmetricizer deals with two cases: cast instructions and constants. The cast instructions are important since they are a part of the pattern of many vector instructions. The symmetricizer will insert additional cast instructions if parts of the instructions have cast type in the worklist. For the constant type, the constant will be propagated until vector loads are found. The constant will be replaced by the load instructions to allow Vegen to generate vector load. The constants will be inserted back in the later stage. For binary operator, if the operator is commutative, the symmetricizer will check whether the symmetric pair of the operators will finally load from the same array and swap the operands to align the load if necessary.

Once the symmetricizer has modified the DAG to increase the symmetry, VeGen continues as before. In the Pattern Matching phase, the symmetry will allow it to match more patterns to the DAG than without the replacements. VeGen continues to use the same calculation to decide whether the vectorization is profitable, allowing it to discard the additional information if it finds that the padded vector instruction will likely not improve the performance. Then at the code generation stage, when generating the vector load, a vector shuffle will be inserted after the load to replace the values from arrays with constants on correct positions.

3.2 Usage

The framework for VeGen+PSLP begins with the implementation of VeGen [1]. Building and using VeGen does not change with the additions of PSLP.

4 Implementation

4.1 Required Transformations

VeGen automatically generates an LLVM pass based on the instruction semantics available, such as through the Intel Intrinsics Guide. As our implementation builds on VeGen, it is generating the LLVM pass in the same way and with the same input requirements. VeGen handles running LLVM, including the other optimizations. Our code doesn't require any extra analyses other than those required by VeGen. However, we need to update the analysis information after we modify the IR. The analyses updated include the DAG of accesses, GlobalDependenceAnalysis, and the ScalarToID map in VeGen. The scalar id is used to encode the dependence in a bit vector.

4.2 Code Components

The major code component is two functions. The first function is called when the VeGen is solving the vector pack for a given seed. The function try to insert some instructions to make the DAG built from the vector seed symmetric so that VeGen can emit the vector pack that it may not be able to emit. The second is called when emitting the actual vector instructions from the vector packs. The function inserts some shuffle instructions after inserting the vector load to preserve the semantics of the original code.

4.3 Testing

All of our testing was performed on UIUC's Campus Cluster. The cluster uses Intel E5-2670 v2 CPUs. The experimental results in the VeGen paper were from Intel E5-2680 v3 and Intel Xeon 8275CL processors. We focus on comparing the results on our Intel E5-2670 v2 to VeGen's reported vectorization on the Intel E5-2680 v3, as that is the more similar processor and the Xeon 8275CL processor has many instructions not available on the other two. However, the Intel E5-2680 v3 has supported intrinsics not available on our Intel E5-2670 v2. Our tests showed that we had 730 vector instructions available. We were not able to find documentation to show what processor type is available on EWS, but it includes the same vector instruction set. We don't know how many were available for the authors of VeGen, but they mention a number of vector instructions in the paper that were not supported on the CC.

4.3.1 Unit Tests

In writing unit tests, we targetted specific operational patterns that we expected to be vectorizable with or without our updates in VeGen. All examples are included in the benchmark git repository in the unit_tests directory.

4.3.2 Small Programs

We included all tests provided by VeGen, including both the examples in the paper and the repository. Initially, some but not all of the examples were vectorizable with the original version of VeGen. For example, the code in Figure 2 was not vectorized by either VeGen or llvm when we ran each compiler, though the results in the Figure showed vector instructions. These tests are included in the repository's performance_tests directory.

4.3.3 Large Programs

We selected three scientific applications from different fields. We began by confirming they compiled with the original VeGen, as other options we considered had compatibility issues.

1. **Plant Species Simulation**: The plant species simulation is a work in progress by Elizabeth for her research. She has been collaborating with a biology group on campus to improve the performance of their simulation. We chose to include this due to her familiarity with the system so that we could more precisely identify the ways VeGen and our improved VeGen were performing optimizations.

	Time (seconds)
LLVM	4.6
VeGen	9.1
VeGen+PSLP	8.2

Table 1: Benchmarking results (in seconds) run on the CC for plant simulation with four compilers. gcc refers to gcc version 7.2.0. LLVM refers to clang version 12.0.1. VeGen is the original version of VeGen, before our modifications, and VeGen+PSLP is our updated version.

2. **SuperLU**: SuperLU [2] solves sparse linear systems using Gaussian elimination. We chose to benchmark based on larger sizes of their test cases, which are designed to cover much of its functionality.

4.4 Functionality

Our implementation successfully compiles our examples. For our unit tests, it vectorizes as expected. However, the pattern matching (as with the original VeGen) is brittle, and so many cases that ideally would be vectorized are not. In the result section, we expect this same brittleness also accounts for the neglible differences in production code examples between LLVM and VeGen. We also were limited by the testing in our hardware, so we weren't able to assess how it would work on a processor with a larger instruction set.

All tests were valgrind-clean.

4.5 Instructions

Our test suite is available in a git repository that includes VeGen and the larger tests programs as submodules.

Testing repository

Modified VeGen repository

Building VeGen requires llvm 12.0.1 (link to tarball). With llvm 12.0.1, it can be built using cmake and all default values.

The unit tests are included in the unit_tests directory within the test repo, which includes a Makefile. It requires building VeGen in a "build" directory in the submodule.

Plant Species Simulation: The plant simulation can be built using the included makefile by calling `make`.

SuperLU: SuperLU can be built by specifying the paths to VeGen's executables and running:

```
cmake -DCMAKE_C_COMPILER=/path/to/vegen/build/vegen-clang
-DCMAKE_CXX_COMPILER=/path/to/vegen/build/vegen-clang++ ..
```

5 Results

5.1 Small Programs

An example of a small program is `performance_tests/test12.c`. The core of the test is:

```
for (int i = 0; i < 400; ++i)
{
    C[i] = A[2 * i] * B[2 * i] + A[2 * i + 1] * 10;
}
```

VeGen+PSLP is able to vectorize the specific pattern of this test and outperform VeGen alone, as shown in Table 1.

5.2 Plant Species Simulation

In the plant species simulation, we found variation in the execution times between the four compilers, as shown in Table 2. When compiled with VeGen+PSLP, we obtained the best time, at 708 seconds. The original version of VeGen and LLVM alone were not far behind, at 739 seconds and 723 seconds. However, gcc performed much worse in this example, requiring 1211 seconds for the simulation.

	Time (seconds)
gcc	1211.28
LLVM	722.90
VeGen	738.72
VeGen+PSLP	707.55

Table 2: Benchmarking results (in seconds) run on the CC for plant simulation with four compilers. gcc refers to gcc version 7.2.0. LLVM refers to clang version 12.0.1. VeGen is the original version of VeGen, before our modifications, and VeGen+PSLP is our updated version.

Test	gcc time	LLVM time	VeGen time	VeGen+PSLP time
c_test	30.29	89.46	89.49	89.89
d_test	30.18	87.90	88.04	88.78
s_test	29.91	87.37	85.78	86.87
z_test	30.76	93.86	93.64	94.14

Table 3: Benchmarking results (in seconds) run on EWS for SuperLU with four compilers. gcc refers to gcc version 7.2.0. LLVM refers to clang version 12.0.1. VeGen is the original version of VeGen, before our modifications, and VeGen+PSLP is our updated version. The test was run once for each compiler.

5.3 SuperLU

The results with SuperLU were the unlike those of the plant species simulation. SuperLU results are listed in Table 3 While LLVM, VeGen, and VeGen+PSLP were again clustered, and this time even more closely, instead of being faster than gcc, they were about three times slower. LLVM, VeGen, and VeGen+PSLP required about 90 seconds per test, and gcc only required about 30.

5.4 Discussion

The discrepancies between gcc and clang between examples was a surprising aspect of the results. Because we were focusing on a single layer of optimization, we can't draw conclusions about why that difference occurred, but it seems like something that would be worth examination given more time. The differences may illuminate contributions to VeGen's success or lack of success.

Unfortunately, though we expect that large scientific code bases must have patterns matching to VeGen and VeGen+PSLP, it appears that a small percentage of the code must fit that description because the optimizations they provide did not make a large dent in the execution time.

In some small examples, VeGen performed worse than LLVM. That is, looking at the specifics of the vector instructions it chose, it overcomplicated the instructions and added additional, unnecessary instructions, even for SIMD cases. This was a limitation of VeGen that we were unable to overcome with our modifications. It seems likely that in the larger cases, it was similarly including extraneous instructions.

In analyzing the performance in terms of time and considering unit tests, the overarching theme is that VeGen is limited in its vectorization success and our updated version shared the same limitations. Some of the limitations are inherent to VeGen, but they are exacerbated with the smaller instruction set available on the Campus Cluster hardware we used, compared to the hardware they used for analysis.

References

[1] Yishen Chen et al. "VeGen: A Vectorizer Generator for SIMD and Beyond". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021, pp. 902–914. ISBN: 9781450383172. DOI: 10.1145/3445814.3446692. URL: https://doi.org/10.1145/3445814.3446692.

[2] Xiaoye S. Li. "An Overview of SuperLU: Algorithms, Implementation, and User Interface". In: 31.3 (Sept. 2005), pp. 302–325.

[3] Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. "PSLP: Padded SLP automatic vectorization". In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2015, pp. 190–201. DOI: 10.1109/CGO.2015.7054199.

A File Layout

This report is in the `cs526projectBenchmarks` directory, the top level directory of the tar file. The tests are in `unit_tests` and `performance_tests` within the `cs526projectBenchmarks` directory as well. The directories for `superlu` and `plantSim` are in the same directory. The code for VeGen is in `cs526projectBenchmarks/vegen`, and the executables are in `cs526projectBenchmarks/vegen/build`, and they are named `clang-vegen` and `clang-vegen++`. The tests can be built using the relevant makefiles.

B Extended Example

In this extended example, we show a function performing addition, but with different combinations of inputs, benefits from VeGen+PSLP's pattern matching. While LLVM compiles the function to 27 lines, VeGen alone compiles it to 31, and VeGen+PSLP compiles it to 23 lines. All three are able to apply some vector instructions to the function, but our VeGen+PSLP does this to the furthest extent.

In comparing the VeGen and VeGen+PSLP versions, the key instructions are shared, but our version shows them as vector instructions. The very first load instructions are straightforward vector instructions for the 4 integers in the VeGen+PSLP version, but requires a masked gather for the VeGen version.

The VeGen version without PSLP has different vectorization than LLVM alone. Where LLVM alone loads the values and then extracts elements, the VeGen version instead uses the masked gather. This is a very short example, so it isn't obvious which is more efficient in terms of runtime, but VeGen would have estimated that this was a useful exchange.

The `shufflevector` instructions introduced in the VeGen+PSLP version demonstrate one of its key behaviors. Where VeGen uses only a masked gather and LLVM uses `extractelement`, VeGen+PSLP more simply incorporates the constants 9 and 11 into the flow of the function.

This example shows that all three versions have vectorizing capabilities, but they approach the problem in different ways. All three are successful at the straighforward vectorization of the adds and stores, once the values have been placed in the correct positions. The different lies in setting up the vectors. VeGen outsources it to `@llvm.masked.gather`. LLVM uses `insertelement`. VeGen+PSLP uses `shufflevector`.

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int32_t A [32];
int32_t B [32];
int32_t C [16];

void vector_add()
{
    for(int count = 0; count < 16; count+=4) {
        C[count+0] = A[count+0] + B[count+0];
        C[count+1] = A[count+1] + B[count+1];
        C[count+2] = A[count+2] + 9;
        C[count+3] = A[count+3] + 11;
    }
}
```

```
; VeGen+PSLP extended example
define dso_local void @vector_add() local_unnamed_addr #0 {
  %1 = load <4 x i32>, <4 x i32>* bitcast ([3200 x i32]* @A to <4 x i32>*), align 16, !tbaa
    !2
  %2 = load <4 x i32>, <4 x i32>* bitcast ([3200 x i32]* @B to <4 x i32>*), align 16, !tbaa
    !2
  %3 = shufflevector <4 x i32> %2, <4 x i32> <i32 9, i32 11, i32 poison, i32 poison>, <4 x
    i32> <i32 0, i32 1, i32 4, i32 5>
```

```
  %4 = add <4 x i32> %3, %1
  store <4 x i32> %4, <4 x i32>* bitcast ([1600 x i32]* @C to <4 x i32>*), align 16, !tbaa !2
  %5 = load <4 x i32>, <4 x i32>* bitcast (i32* getelementptr inbounds ([3200 x i32], [3200 x i32]* @A, i64 0, i64 4) to <4 x i32>*), align 16, !tbaa !2
  %6 = load <4 x i32>, <4 x i32>* bitcast (i32* getelementptr inbounds ([3200 x i32], [3200 x i32]* @B, i64 0, i64 4) to <4 x i32>*), align 16, !tbaa !2
  %7 = shufflevector <4 x i32> %6, <4 x i32> <i32 9, i32 11, i32 poison, i32 poison>, <4 x i32> <i32 0, i32 1, i32 4, i32 5>
  %8 = add <4 x i32> %7, %5
  store <4 x i32> %8, <4 x i32>* bitcast (i32* getelementptr inbounds ([1600 x i32], [1600 x i32]* @C, i64 0, i64 4) to <4 x i32>*), align 16, !tbaa !2
  %9 = load <4 x i32>, <4 x i32>* bitcast (i32* getelementptr inbounds ([3200 x i32], [3200 x i32]* @A, i64 0, i64 8) to <4 x i32>*), align 16, !tbaa !2
  %10 = load <4 x i32>, <4 x i32>* bitcast (i32* getelementptr inbounds ([3200 x i32], [3200 x i32]* @B, i64 0, i64 8) to <4 x i32>*), align 16, !tbaa !2
  %11 = shufflevector <4 x i32> %10, <4 x i32> <i32 9, i32 11, i32 poison, i32 poison>, <4 x i32> <i32 0, i32 1, i32 4, i32 5>
  %12 = add <4 x i32> %11, %9
  store <4 x i32> %12, <4 x i32>* bitcast (i32* getelementptr inbounds ([1600 x i32], [1600 x i32]* @C, i64 0, i64 8) to <4 x i32>*), align 16, !tbaa !2
  %13 = load <4 x i32>, <4 x i32>* bitcast (i32* getelementptr inbounds ([3200 x i32], [3200 x i32]* @A, i64 0, i64 12) to <4 x i32>*), align 16, !tbaa !2
  %14 = load <4 x i32>, <4 x i32>* bitcast (i32* getelementptr inbounds ([3200 x i32], [3200 x i32]* @B, i64 0, i64 12) to <4 x i32>*), align 16, !tbaa !2
  %15 = shufflevector <4 x i32> %14, <4 x i32> <i32 9, i32 11, i32 poison, i32 poison>, <4 x i32> <i32 0, i32 1, i32 4, i32 5>
  %16 = add <4 x i32> %15, %13
  store <4 x i32> %16, <4 x i32>* bitcast (i32* getelementptr inbounds ([1600 x i32], [1600 x i32]* @C, i64 0, i64 12) to <4 x i32>*), align 16, !tbaa !2
  ret void
}
```

```
; VeGen extended example
define dso_local void @vector_add() local_unnamed_addr #0 {
  %1 = load i32, i32* getelementptr inbounds ([32 x i32], [32 x i32]* @A, i64 0, i64 0), align 16, !tbaa !2
  %2 = call <4 x i32> @llvm.masked.gather.v4i32.v4p0i32(<4 x i32*> <i32* getelementptr inbounds ([32 x i32], [32 x i32]* @B, i64 0, i64 0), i32* getelementptr inbounds ([32 x i32], [32 x i32]* @B, i64 0, i64 1), i32* getelementptr inbounds ([32 x i32], [32 x i32]* @A, i64 0, i64 2), i32* getelementptr inbounds ([32 x i32], [32 x i32]* @A, i64 0, i64 3)>, i32 4, <4 x i1> <i1 true, i1 true, i1 true, i1 true>, <4 x i32> undef), !tbaa !2
  %3 = load i32, i32* getelementptr inbounds ([32 x i32], [32 x i32]* @A, i64 0, i64 1), align 4, !tbaa !2
  %4 = insertelement <4 x i32> <i32 poison, i32 poison, i32 9, i32 11>, i32 %3, i64 1
  %5 = insertelement <4 x i32> %4, i32 %1, i64 0
  %6 = add <4 x i32> %2, %5
  store <4 x i32> %6, <4 x i32>* bitcast ([16 x i32]* @C to <4 x i32>*), align 16, !tbaa !2
  %7 = load i32, i32* getelementptr inbounds ([32 x i32], [32 x i32]* @A, i64 0, i64 4), align 16, !tbaa !2
  %8 = call <4 x i32> @llvm.masked.gather.v4i32.v4p0i32(<4 x i32*> <i32* getelementptr inbounds ([32 x i32], [32 x i32]* @B, i64 0, i64 4), i32* getelementptr inbounds ([32 x i32], [32 x i32]* @B, i64 0, i64 5), i32* getelementptr inbounds ([32 x i32], [32 x i32]* @A, i64 0, i64 6), i32* getelementptr inbounds ([32 x i32], [32 x i32]* @A, i64 0, i64 7)>, i32 4, <4 x i1> <i1 true, i1 true, i1 true, i1 true>, <4 x i32> undef), !tbaa !2
  %9 = load i32, i32* getelementptr inbounds ([32 x i32], [32 x i32]* @A, i64 0, i64 5), align 4, !tbaa !2
  %10 = insertelement <4 x i32> <i32 poison, i32 poison, i32 9, i32 11>, i32 %9, i64 1
  %11 = insertelement <4 x i32> %10, i32 %7, i64 0
  %12 = add <4 x i32> %8, %11
  store <4 x i32> %12, <4 x i32>* bitcast (i32* getelementptr inbounds ([16 x i32], [16 x i32]* @C, i64 0, i64 4) to <4 x i32>*), align 16, !tbaa !2
  %13 = load i32, i32* getelementptr inbounds ([32 x i32], [32 x i32]* @A, i64 0, i64 8), align 16, !tbaa !2
  %14 = call <4 x i32> @llvm.masked.gather.v4i32.v4p0i32(<4 x i32*> <i32* getelementptr inbounds ([32 x i32], [32 x i32]* @B, i64 0, i64 8), i32* getelementptr inbounds ([32 x
```

```
    i32], [32 x i32]* @B, i64 0, i64 9), i32* getelementptr inbounds ([32 x i32], [32 x i32
    ]* @A, i64 0, i64 10), i32* getelementptr inbounds ([32 x i32], [32 x i32]* @A, i64 0,
    i64 11)>, i32 4, <4 x i1> <i1 true, i1 true, i1 true, i1 true>, <4 x i32> undef), !tbaa
    !2
  %15 = load i32, i32* getelementptr inbounds ([32 x i32], [32 x i32]* @A, i64 0, i64 9),
    align 4, !tbaa !2
  %16 = insertelement <4 x i32> <i32 poison, i32 poison, i32 9, i32 11>, i32 %15, i64 1
  %17 = insertelement <4 x i32> %16, i32 %13, i64 0
  %18 = add <4 x i32> %14, %17
  store <4 x i32> %18, <4 x i32>* bitcast (i32* getelementptr inbounds ([16 x i32], [16 x
    i32]* @C, i64 0, i64 8) to <4 x i32>*), align 16, !tbaa !2
  %19 = load i32, i32* getelementptr inbounds ([32 x i32], [32 x i32]* @A, i64 0, i64 12),
    align 16, !tbaa !2
  %20 = call <4 x i32> @llvm.masked.gather.v4i32.v4p0i32(<4 x i32*> <i32* getelementptr
    inbounds ([32 x i32], [32 x i32]* @B, i64 0, i64 12), i32* getelementptr inbounds ([32 x
     i32], [32 x i32]* @B, i64 0, i64 13), i32* getelementptr inbounds ([32 x i32], [32 x
    i32]* @A, i64 0, i64 14), i32* getelementptr inbounds ([32 x i32], [32 x i32]* @A, i64
    0, i64 15)>, i32 4, <4 x i1> <i1 true, i1 true, i1 true, i1 true>, <4 x i32> undef), !
    tbaa !2
  %21 = load i32, i32* getelementptr inbounds ([32 x i32], [32 x i32]* @A, i64 0, i64 13),
    align 4, !tbaa !2
  %22 = insertelement <4 x i32> <i32 poison, i32 poison, i32 9, i32 11>, i32 %21, i64 1
  %23 = insertelement <4 x i32> %22, i32 %19, i64 0
  %24 = add <4 x i32> %20, %23
  store <4 x i32> %24, <4 x i32>* bitcast (i32* getelementptr inbounds ([16 x i32], [16 x
    i32]* @C, i64 0, i64 12) to <4 x i32>*), align 16, !tbaa !2
  ret void
}
```

```
; clang 12.0.1 extended example
define dso_local void @vector_add() local_unnamed_addr #0 {
  %1 = load <2 x i32>, <2 x i32>* bitcast ([32 x i32]* @B to <2 x i32>*), align 16, !tbaa !2
  %2 = load i32, i32* getelementptr inbounds ([32 x i32], [32 x i32]* @B, i64 0, i64 4),
    align 16, !tbaa !2
  %3 = load i32, i32* getelementptr inbounds ([32 x i32], [32 x i32]* @B, i64 0, i64 5),
    align 4, !tbaa !2
  %4 = load <8 x i32>, <8 x i32>* bitcast ([32 x i32]* @A to <8 x i32>*), align 32, !tbaa !2
  %5 = extractelement <2 x i32> %1, i32 0
  %6 = extractelement <2 x i32> %1, i32 1
  %7 = insertelement <8 x i32> <i32 poison, i32 poison, i32 9, i32 11, i32 poison, i32
    poison, i32 9, i32 11>, i32 %5, i32 0
  %8 = insertelement <8 x i32> %7, i32 %6, i32 1
  %9 = insertelement <8 x i32> %8, i32 %2, i32 4
  %10 = insertelement <8 x i32> %9, i32 %3, i32 5
  %11 = add nsw <8 x i32> %10, %4
  store <8 x i32> %11, <8 x i32>* bitcast ([16 x i32]* @C to <8 x i32>*), align 32, !tbaa !2
  %12 = load <2 x i32>, <2 x i32>* bitcast (i32* getelementptr inbounds ([32 x i32], [32 x
    i32]* @B, i64 0, i64 8) to <2 x i32>*), align 16, !tbaa !2
  %13 = load i32, i32* getelementptr inbounds ([32 x i32], [32 x i32]* @B, i64 0, i64 12),
    align 16, !tbaa !2
  %14 = load i32, i32* getelementptr inbounds ([32 x i32], [32 x i32]* @B, i64 0, i64 13),
    align 4, !tbaa !2
  %15 = load <8 x i32>, <8 x i32>* bitcast (i32* getelementptr inbounds ([32 x i32], [32 x
    i32]* @A, i64 0, i64 8) to <8 x i32>*), align 32, !tbaa !2
  %16 = extractelement <2 x i32> %12, i32 0
  %17 = extractelement <2 x i32> %12, i32 1
  %18 = insertelement <8 x i32> <i32 poison, i32 poison, i32 9, i32 11, i32 poison, i32
    poison, i32 9, i32 11>, i32 %16, i32 0
  %19 = insertelement <8 x i32> %18, i32 %17, i32 1
  %20 = insertelement <8 x i32> %19, i32 %13, i32 4
  %21 = insertelement <8 x i32> %20, i32 %14, i32 5
  %22 = add nsw <8 x i32> %21, %15
  store <8 x i32> %22, <8 x i32>* bitcast (i32* getelementptr inbounds ([16 x i32], [16 x
    i32]* @C, i64 0, i64 8) to <8 x i32>*), align 32, !tbaa !2
  ret void
}
```