Multi-dimensional Vectorization in LLVM

Simon Moll Shrey Sharma Matthias Kurtenacker Sebastian Hack
Saarland University
Saarland Informatics Campus
{moll,shrey.sharma,kurtenacker,hack}@cs.uni-saarland.de

Abstract

Loop vectorization is a classic technique to exploit SIMD instructions in a productive way. In multi-dimensional vectorization, multiple loops of a loop nest are vectorized at once. This exposes opportunities for data reuse, register tiling and more efficient memory accesses.

In this work, we present TensorRV, a multi-dimensional vectorization framework for LLVM IR. TensorRV is a generalization of the Region Vectorizer, a general purpose outer-loop and whole-function vectorizer, to the multi-dimensional setting.

We evaluate TensorRV on a set of stencil codes and matrix transpose. We find that stencil codes benefit from the reduction of load instructions with a speedup of $\times$1.45 on NEC SX-Aurora TSUBASA. Multi-loop vectorized matrix transpose leverages efficient SIMD shuffle instructions on AVX512, for which we report a speedup of $\times$3.27.

Keywords Tensor, Vectorization, SIMD, Compiler, Optimization

ACM Reference Format:
Simon Moll Shrey Sharma Matthias Kurtenacker Sebastian Hack. 2019. Multi-dimensional Vectorization in LLVM. In *Workshop on Programming Models for SIMD/Vector Processing (WPMVP'19), February 16, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3303117.3306172

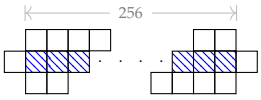
1 Introduction

Outer-loop vectorization is a standard technique in optimizing compilers to exploit SIMD hardware in a productive way. Basically every modern compiler includes some kind of loop vectorizer, let it be LLVM [11], the Region Vectorizer [14] or GCC. These vectorizers are one-dimensional in the sense that when applied to a

WPMVP'19 , February 16, 2019, Washington, DC, USA

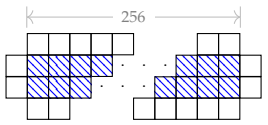
ACM ISBN 978-1-4503-6291-7/19/02…$15.00
https://doi.org/10.1145/3303117.3306172

```
1 for (j = 1; j < rows - 1; j += 1)
2 for (i = 1; i < cols - 1; i += 256) {
3   a = load_v256(A(j-1, i  ));
4   b = load_v256(A(j  , i-1));
5   c = load_v256(A(j  , i  ));
6   d = load_v256(A(j  , i+1));
7   e = load_v256(A(j+1, i  ));
8
9   store_v256(B(j,i), .2 * (a+b+c+d+e));  }
```



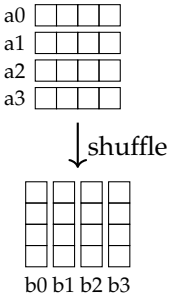
(a) 5-point Jacobi vectorized in the `i` loop (width 256).

```
1 for (j = 1; j < rows - 1; j += 2)
2 for (i = 1; i < cols - 1; i += 256) {
3   a0 = load_v256(A(j-1,i  ));
4   a1 = load_v256(A(j  ,i  ));
5   b0 = load_v256(A(j  ,i-1));
6   b1 = load_v256(A(j+1,i-1));
7   c0 = a1; // reuse
8   c1 = load_v256(A(j+1,i  ));
9   d0 = load_v256(A(j  , i+1));
10  d1 = load_v256(A(j+1, i+1));
11  e0 = c1; // reuse
12  e1 = load_v256(A(j+2, i ));
13
14  b0 = .2 * (a0+b0+c0+d0+e0);
15  b1 = .2 * (a1+b1+c1+d1+e1);
16  store_v256(B(j  ,i), b0);
17  store_v256(B(j+1,i), b1);  }
```



(b) Jacobi tensorized in both `j` (width 2) and `i` (width 256) resulting in 20% less vector loads.

```
1 for (j = 0; j < rows; j += 4)
2 for (i = 0; i < cols; i += 4) {
3   // fetch contiguous chunks of A
4   a0 = load_v4(A(j  , i);
5   ..
6   a3 = load_v4(A(j+3, i);
7
8   // in-register transpose
9   b0 = shuffle(a0[0], a1[0], a2[0], a3[0])
10  ..
11  b3 = shuffle(a0[3], a1[3], a2[3], a3[3])
12
13  // store contiguous chunks to B
14  store_v4(B(i, j),   b0)
15  ..
16  store_v4(B(i, j+3), b3)  }
```



(c) Matrix transpose tensorized in both loops by 4×4. Loads are contiguous in `i`. Stores are contiguous in `j`.

Figure 1. Optimizations enabled by loop nest tensorization.

loop nest, they will pick one of the loops and vectorize it, leaving the others un-vectorized.

However, some codes benefit from considering multiple loops at once in the vectorization process. We refer to the vectorization in multiple loops as tensorization since a tensor is nothing more than the multi-dimensional generalization of a vector.

We provide two motivating examples to demonstrate the benefits of tensorization, shown in Figure 1, a 5-point Jacobi stencil and matrix transpose. The tensorized versions in Figure 1b and Figure 1c were automatically generated from scalar code with our tensorizer prototype.

The 5-point Jacobi example of Figure 1b shows how tensorization naturally results in vector register tiling and vector load coalescing. The loop-vectorized Jacobi kernel in Figure 1a performs five vector loads to compute one result. Contrast this with the tensorized version shown in Figure 1b where the stencil is tensorized in both the `i`-loop by 256 and the `j`-loop by 2 at once. We see two effects of tensorizing this kernel: First, since the native vector length of the target is 256, the tensor values of 2×256 elements are tiled into two vector registers. Second, our memory access chunking scheme implements tensor loads using fast contiguous vector loads. The memory loads of the tensorized code overlap, which is why `c0` and `e0` are equivalent to already loaded values. In effect, the tensorized kernel performs eight loads to compute two results on average where the loop vectorized requires ten loads to achieve the same.

However, there is more to tensorization than register tiling: By considering an entire loop nest, we can emit fast contiguous memory accesses even if the memory instructions of a kernel are contiguous in different iteration variables. We demonstrate this for matrix transposition, that is the transformation `B(i, j) = A(j, i)`. In the tensorized matrix transpose, shown in Figure 1c, four contiguous loads and four contiguous stores are used along with shuffles to efficiently transpose 16 matrix elements in one go. This leads to a speedup of $\times 3.27$ over the loop vectorized version on a AVX512 platform. When vectorized in only one loop, there is only one contiguous access and the other would be a slow scatter or gather instruction.

In this paper, we present a multi-dimensional vectorization framework for LLVM that performs these transformations automatically based on scalar LLVM IR. This paper is structured as follows: Section 2 places our approach in the context of related work. Section 3 presents our loop-nest analysis framework, including the multi-dimensional tensor shape analysis and our memory access analysis. Section 4 describes how we generate efficient SIMD code and memory accesses from the analysis results. Section 5 presents our evaluation results on AVX512 and NEC SX-Aurora TSUBASA. We finally conclude in Section 6.

1.1 Contributions

This paper makes the following contributions:

- We present the tensor shape analysis, a divergence analysis for multi-dimensional vectorization. Tensor shapes generalize the vector shapes used in outer-loop vectorization to multi-dimensional loop nests.
- We demonstrate how to efficiently generate vector code for tensorized loop nests through brush projections and memory chunking. Memory chunking partitions tensorized memory accesses into fast contiguous vector accesses.
- We implemented our approach in *TensorRV*, a fork of the Region Vectorizer [14] for LLVM IR. We evaluate *TensorRV* on a set of common stencil codes and a matrix transposition kernel. The target SIMD ISAs for our experiments are AVX512 (512bit vector length) and NEC SX-Aurora TSUBASA (16384bit vector length). We observe speedups of $\times 1.45$ on SX-Aurora and $\times 3.27$ on AVX512. The performance counters on SX-Aurora reveal that tensorization reduces the amount of loaded vector elements by up to 21.4% compared to loop vectorization.

2 Related Work

The idea of multi-dimensional loop vectorization can be traced back to Allen and Kennedy's work on the Parallel Fortran Compiler [1]. As a source to source translator, their framework would analyze loop nests for data dependencies and vectorize independent inner loops to generate multi-dimensional Fortran 8x code.

Since most modern systems have single dimensional memory, even contiguous accesses in higher dimensions lead to strided memory accesses that require gather and scatter instructions. This was addressed in Vector Folding [21] by performing data layout transformations as a preprocessing step before multi-dimensional vectorization. Data layout transformations have also been used [8] to reduce shuffle operations induced by unaligned memory accesses. In our approach, we use shuffles with contiguous loads to generate operand vectors [3, 6].

For loop nests with short trip counts, single dimensional vectorization leads to under utilization of available SIMD lanes. Rodrigues et al. [18] showed that multi-dimensional vectorization enables efficient vector register utilization in such cases. However, their compiler

requires a high level specification of the tensor operation algorithm and needs contiguous memory accesses without gaps. Our framework uses scalar code as input and can handle non constant strides. There is limited support for multi-dimensional vectorization in the ISPC programming language [15] through the `foreach_tiled` statement. However, lacking a multi-dimensional analysis, ISPC will use scatter/gather to vectorize every memory access that is not fully uniform in that mode.

Our approach is orthogonal to spatio-temporal tiling approaches [5, 13] in the sense that the generated tiles may still be processed by multi-dimensional vector code.

Optimization of tensor operations on SIMD hardware is an active area of research and various manual [12] and automatic [18, 20] vectorization techniques have been developed to improve throughput. Such operations often involve memory accesses in multiple dimensions and can benefit from vector register tiling [2, 4, 17] enabled by multi-dimensional vectorization.

The tensor shape analysis is a multi-dimensional generalization of divergence analysis [10, 19]. The strides in the tensor shapes are comparable to affine constraints in some, one-dimensional, divergence analysis lattices [7, 19].

3 Tensorization of Loop Nests

Tensorization is the generalization of outer-loop vectorization to multiple dimensions. In one-dimensional vectorization, the vectorization factor specifies how many loop iterations are packaged into one vector. In tensorization, the *tensor brush* defines one vectorization factor per surrounding loop. If the loop nest has d loops then the tensor brush is formally defined as

$$\mathcal{B} = (m_0 \times \cdots \times m_{d-1})$$

Where $m_i \in \mathbb{N}$ is the size of the brush in the direction of the i-th dimension. Each dimension corresponds to a loop level of the loop nest. The outer-most loop is always at dimension 0 increasing with each nested loop.

Similar to how lanes identify the elements of a vector in loop vectorization, we will use *Coordinates* to refer to the multi-dimensional elements of a given brush. For example, the vector $(1, 2)$ is a valid brush coordinate of $\mathcal{B} = 4 \times 4$.

Given a loop nest and a tensor brush $\mathcal{B}$, our framework performs tensorization in three stages: First, we run the tensor shape analysis to determine the tensor shapes of all instructions in the loop nest. Second, we group the memory accesses that will occur in the vectorized loop body. Third, we use the tensor shapes and memory access groupings to generate vector code with optimized memory accesses.

```
1 for (i = 0; i < m; ++i) // dim 0
2   for (j = 0; j < n; ++j) // dim 1
3     for (k = 0; k < w; ++k) // dim 2
4       ..S..
```

(a) A 3D loop nest with placeholder statement `S`.

`..S..`	Tensor shape $\mathcal{T}_S$
`i`	$(1,0,0)$
`j`	$(0,1,0)$
`k`	$(0,0,1)$
`12 * i - 6 * j`	$(12,-6,0)$
`2*A(i) + 10 * j`	$(\top,10,0)$
`B(i,j) + 5 * k`	$(\top,\top,5)$
`C(i) * B(k)`	$(\top,0,\top)$
`C(B(j))`	$(0,\top,0)$

(b) Upper section: initial tensor shapes, Lower section: tensor shapes inferred by the tensor shape analysis.

Figure 2. Above: a 3D loop nest with placeholder statement `S`. Below: examples for `S` and their resulting tensor shapes.

3.1 Tensor Shape Analysis

The tensor shape analysis determines whether and how the instructions in the loop nest depend on the iteration variables of the loops that surround them. The analysis assigns to every instruction a *Tensor Shape* that captures the nature of this dependence.

The *tensor shape* describes per surrounding loop how the value of the instruction changes from one loop iteration to the next. This information is recorded separately for each dimension, similar to how the set of partial derivatives constitutes the complete derivative of a function. For each dimension of the loop nest separately, tensor shapes hold a vector shape defined as:

$$\mathcal{T}_{1\mathtt{D}} = \mathbb{Z} \cup \{\top\}$$

The set of d-dimensional tensor shapes, $\mathcal{T}^d$, is then the composition of d one-dimensional vector shapes:

$$\mathcal{T}^d = (\mathcal{T}_{1\mathtt{D}})^d$$

We will use the notation $(s_0, s_1, s_2) \in \mathcal{T}^3$ to denote to the elements of a $3D$ tensor shape. Each $s_i \in \mathcal{T}_{1\mathtt{D}}$ is the dimension shape of its dimension i.

If $s_i \in \mathcal{T}_{1\mathtt{D}}$ is an integer, then s_i is the constant loop increment for the annotated instruction for the loop at dimension i. In particular if $s_i = 0$ then the annotated instruction is invariant, that is *uniform*, in loop. If $s_i \in \mathcal{T}_{1\mathtt{D}}$ is $\top$, then the instruction is *varying* in dimension i but the nature of the variation is not reflected in the

tensor shape. For example, if i is used in an array load such as `A[..]` then the loaded value is always varying in i because the contents of the array are in general unknown.

Figure 2a shows a three-dimension loop nest and the tensor shapes of its induction variables are listed in the upper half of Figure 2b. The induction variables are invariant in all other dimensions than their own. For example, the induction variable `j` has a stride of 0, that is it is uniform, in all dimensions but dimension 1.

The tensor shape analysis propagates these initial shapes to all instructions in the loop nests. We show some examples for fill-ins for the placeholder `S` of Figure 2a in the lower half of Figure 2b.

If the instruction computes an affine combination of iteration variables, the tensor shape reflects these as strides. The tensor shape analysis assumes that there are no loop-carried dependencies between the instructions of the loop nest. Therefore, the tensor shapes of memory accesses only depends on the tensor shape of the address computation. The relation between values loaded from different addresses is, however, *varying*.

3.2 Memory access grouping

If memory accesses are naively widened, there is often an overlap in the accessed elements among different instructions. Memory access grouping is mapping out this overlap with the end of generating more efficient vector memory accesses in the code generation phase.

An access group is defined by the equivalence relation $\overset{M}{\sim}$ on memory instruction plus coordinate pairs. If $(\mathtt{L}, c) \overset{M}{\sim} (\mathtt{L}', c')$ then the value loaded by `L` at tensor coordinate c will be the same as the one loaded by `L'` at coordinate c'. For example, it holds that

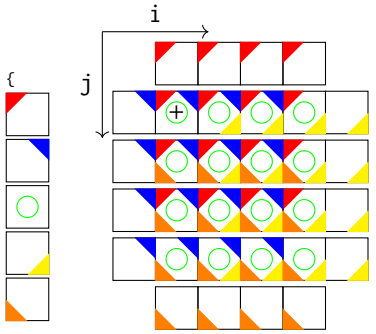
$\mathtt{A(i-1,\ j)}, (2,0) \overset{M}{\sim} \mathtt{A(i,j)}, (1,0)$.

Memory access grouping is concerned with constructing these access groups for all memory accesses in the loop nest. Since the tensor brush has finite extent and there are only finitely many memory instructions in the kernel, we can explicitly construct each access group by enumeration. The implementation builds on LLVM's *Scalar Evolution* analysis to compute the offsets between pointers [16].

Figure 3a shows the memory access grouping of a 5-points Jacobi stencil. Assume that the loop nest will be vectorized for a 4×4 tensor brush. Figure 3b shows the resulting access group for the loads in Line 3 to Line 7. We use a color/symbol coding scheme in the Figure to visualize each load instruction. To the right, in Figure 3b, we show the accessed elements of `A`. Each cell in Figure 3b is an array subscript to `A`. The symbols in the cells indicate which positions of the array are accessed by which load instructions.

```
1  for (j = 1..rows - 1)
2  for (i = 1..cols - 1) {
3    a = A(j-1, i  );
4    b = A(j  , i-1);
5    c = A(j  , i  );
6    d = A(j  , i+1);
7    e = A(j+1, i  );
8
9    B(j,i) =
10     .2 * (a+b+c+d+e);
11 }
```

(a) 5-point Jacobi kernel



(b) Load access group for `A` given a 4×4 brush. Elements in `i` direction are contiguous, the distance in `j` direction is m, indicated by a gap.

As an example consider the cell marked '+', which refers to the array element `A(j,i)`. The cell contains the instruction symbols of Line 3, Line 4 and Line 5. We can read off the access group that Coordinate $(1,0)$ of the load in Line 3 refers to the same pointer as Coordinate $(0,1)$ of the load in Line 4.

For our prototype, we assume that there are no conflicts between loads and stores, e.g. that the elements fetched by a load would become invalid due to an overwriting store. This is a typical pattern for stencil codes, which fetch elements from an input array and store the accumulated sum into an output array.

4 Vector Code Generation

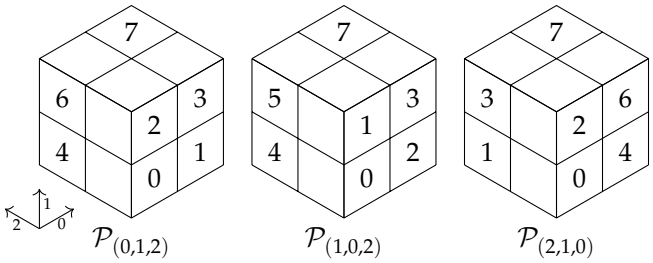
This section describes the generation of vector instruction for a tensorized loop nest starting from scalar code and the analysis data that has been gathered up to this stage. The analysis data are tensor shapes (Section 3.1) and memory access groups (Section 3.2).

The loop trip counts are trivially transformed by scaling them by the brush sizes. We will concentrate on the formation of vector instructions for two SIMD ISAs: AVX512 (x86) and VE (NEC SX-Aurora).

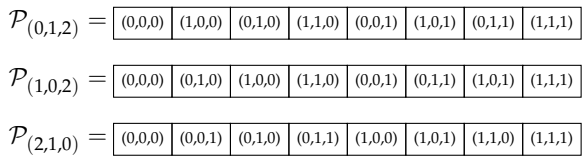
These two ISAs share the property that they do not actually have tensor registers but one-dimension vector registers. To this end, Section 4.1 describes how tensors can be represented unambiguously in vector registers. Section 4.2 explains the basic code generation algorithm and Section 4.3 covers how we efficiently vectorize tensor memory accesses.

4.1 Brush projections

So far, we have used the abstract concept of tensors and coordinates to reason about the behavior of instructions in tensorized loop nests. However, the actual target ISAs have one-dimensional vector registers. In order to



(a)



(b)

Figure 4. Vector lane mapping according to brush projections.

generate code, we have to define a way to store the elements of the conceptual tensors in these concrete vector registers. There is ambiguity in this decision similar to how a matrix can be represented in column-major or row-major layout. To this end, we introduce the concept of brush projections. A *brush projection* ($\mathcal{P}_{\mathcal{B}}$) defines a unique mapping from the multi-dimensional coordinate space of the vector brush $\mathcal{B}$ to vector lanes.

A brush projection is defined by the projection vector, a vector of dimension numbers $(i_0, \ldots i_{d-1})$. The projection vector specifies a cardinality of the dimensions. The brush coordinates are then enumerated according to this order similar to numbers in positional notation. Equation (1) defines the projection from coordinates $(c_0, .., c_{d-1})$ to vector lanes, given a brush $\mathcal{B} = (m_0 \times .. \times m_{d-1})$ and a projection vector $(i_0, .., i_{d-1})$.

$$\mathcal{P}_{(i,i_1,..,i_{d'})}(c_0, .., c_{d-1}) = \left\{ \begin{array}{ll} c_i + m_i \mathcal{P}_{(i_1,..,i_{d'})}(c_0, .., c_{d-1}) & \text{if } d' > 1 \\ c_i & \text{otw} \end{array} \right\} \quad (1)$$

We show three brush projections of a $2 \times 2 \times 2$ brush in Figure 4.

4.2 Generic Vector Code Generation

The generic vector code generation loop visits every instruction in reverse post order, classifies it by the following cases and transforms it accordingly. We assume that all instructions except memory accesses are free of side effects. We deal with memory access separately (Section 4.3). If the instruction is side-effect free (e.g. arithmetic), there are two cases to be considered: all-strided and one-varying.

All-strided The first case applies when the tensor shape is strided (or uniform) in all dimensions with brush size greater than 1. In this case, we emit a scalar instruction that computes the result of the instruction at the zero coordinate $(0, .., 0)$. Since the operation is side-effect free, there is no need to replicate it for all coordinates, we only have to provide the computed result for all brush coordinates. Finally, we can compute the results at all other coordinates whenever needed simply by adding an appropriate multiple of the strides.

One-varying If the tensor shape of an operation is $\top$ in at least one dimension, the scalar data type will be widened to a vector data type. The length of the vector is the product of all dimension sizes of the brush $\mathcal{B}$. For example, if the brush is $4 \times 2 \times 4$ and the data type of the operations is `double`, the widened vector instruction will operate on the data type `<32 x double>`. If any of the operands of the operation fell in the *all-strided* case they will now be instantiated into the lanes of a full vector according to the brush projection, $\mathcal{P}$.

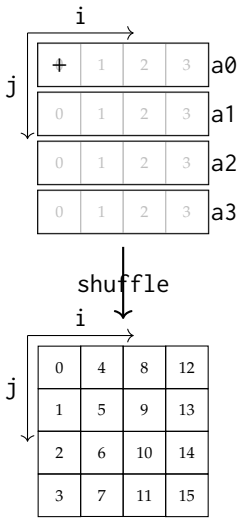
Vector register tiling We leverage LLVM's legalization phase to effectively implement vector register tiling. This occurs when the number of coordinates in the brush is greater than the vector register length. In that situation, LLVM will split the generated vector instruction into parts, each with the native vector length. For example, if $\mathcal{B} = 2 \times 256$ and the element type is `double`, our code generation phase will initially emit a 512-element vector instruction in LLVM IR. However, the vector register length for `double` on NEC SX-Aurora is 256 and so LLVM will replace the instruction by two vector instructions with 256 elements.

4.3 Memory Access Chunking

Memory chunking is the process of partitioning the accessed elements into contiguous parts, called *chunks*. Each chunk is then loaded or stored with a fast contiguous memory instruction.

On the top right of Figure 5, we show a single memory access in a loop nest of depth two. Consider that this loop nest is tensorized with a brush of 4×4. Using the generic one-varying strategy, the load would be vectorized with a slow gather instruction. However, scatter and gather instruction are the least effective means of accessing memory on AVX512 and SX-Aurora. It is advisable to use contiguous memory accesses whenever possible.

We employ the memory grouper analysis presented in Section 3.2 to divide the accessed memory locations into contiguous chunks. This results in the contiguous loads `a0` to `a3` shown below. This is an AVX2 example




```
1 // scalar loop nest
2 for (i=0; i<k; ++i) // dim 0
3 for (j=0; j<m; ++j) { // dim 1
4   x = A(j, i);
5   ...
6 }
```

```
1 // tensorized loop nest
2 for (i=0; i<k; i += 4) // dim 0
3 for (j=0; j<m; j += 4){// dim 1
4 // chunked load
5   a0 = vload_4(A(j,   i));
6   a1 = vload_4(A(j+1, i));
7   a2 = vload_4(A(j+2, i));
8   a3 = vload_4(A(j+3, i));
9
10 // shuffling
11   x = shufflevector [
12   a0[0], a1[0], a2[0], a3[0],
13   a0[1], a1[1], a2[1], a3[1],
14   a0[2], a1[2], a2[2], a3[2],
15   a0[3], a1[3], a2[3], a3[3],
16   ];
17   ...
18 }
```

Figure 5. Chunking a tensor load into contiguous memory accesses for $\mathcal{B} = 4 \times 4$. **Left:** the load `A(i, j)` is chunked into contiguous loads `a0` to `a1`. Below, the loaded chunks are shuffled according to the brush projection $\mathcal{P}_{\mathcal{B}} = (1, 0)$. Numbers in the cells refer to the lane number. **Right:** the scalar loop nest on top. Below, the tensorized version with chunked loads and shuffles.

(256*bit* SIMD registers) and thus each consecutive vector load can transfer four contiguous element from the `double` array.

Finally, the lanes need to be shuffled into the vector according to the brush projection. We do the inverse, shuffling followed by consecutive stores, to vectorize the tensor store.

5 Evaluation

We implemented our tensorizer prototype, coined *TensorRV*, as a fork of the Region Vectorizer [14] for LLVM [11]. We use the recent LLVM-VE backend [9] to generate code for NEC SX-Aurora.

Platforms We report single-thread results on the following platforms:

- **VE** A single NEC Aurora TSUBASA Vector Engine 10B model, with 1.4GHz clock frequency.
- **VH** Intel(R) Xeon(R) Gold 6126 CPU (Vector Host).

Compiler configuration

- **NCC** Version 1.6.0 of the official C/C++ compiler for SX-Aurora (VE only).
- **ND** LLVM+TensorRV with a multi-dimensional brush (VE and VH).
- **1D** LLVM+TensorRV with a single-dimensional brush. This is equivalent to vectorization with RV (VE and VH).

5.1 Benchmarks

We evaluate our technique on four stencil codes and matrix transpose. We report runtime measurements in Figure 6a on a collection of stencil codes and matrix transposition. The benchmarks are written in C and are compiled with Clang to LLVM IR.

Stencils Unless otherwise specified, the brushes are 2×256 on the VE and 4×8 on the VH. The brush sizes were chosen empirically.

- **jacobi** 5-point Jacobi stencil (cross shaped pattern).
- **jacobi9** 9-point Jacobi stencil (cross shaped pattern).
- **seidel** accesses all 9 elements within a distance of 1. We used a 4×8 brush on the VH.
- **sobel** Two 5-point stencil applications in a single kernel, one for each derivative.

For the tensorized versions (nd), the dimension of largest extent is chosen as the first element of the projection vector. In the single-dimensional versions (1d), the inner-most loop dimension is kept and all others are set to one.

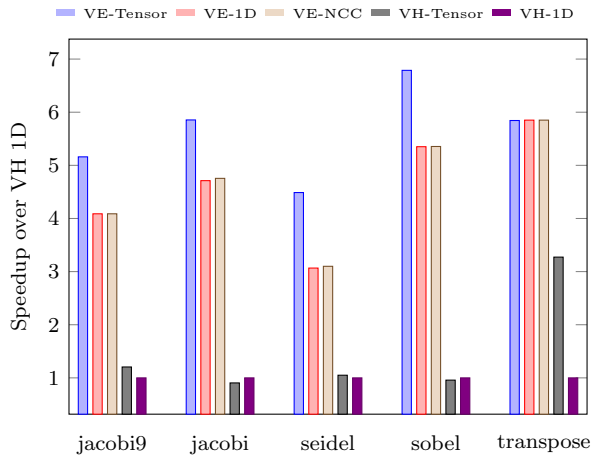
We apply the stencils to a 1024×1024 center part of an array of `double` elements. The arrays are linear in memory, that is `A(j,i)` translates to `A[j * m + i]`. We use the `restrict`-qualifier on all data arrays to indicate to NCC that the arrays do not overlap and the kernel can be safely vectorized.

Matrix Transpose Regarding memory re-use, the matrix transpose kernel is the opposite of the stencil codes. In the matrix transposition kernel, every array element is accessed exactly once.

We chose a 2×256 brush on the VE and a 4×4 brush on the VH. The single-dimensional version vectorizes the inner loop by a width of eight.

5.1.1 VE: NEC SX-Aurora

The memory subsystem of NEC SX-Aurora consists of a 16*MB* memory-side Last Level Cache (LLC). Unlike for AVX512, there is no L1 cache on the Vector Processing Units. We therefore expect runtime improvements with tensorization whenever our memory load re-use scheme eliminates redundant loads. The runtime results in Figure 6a along with performance counter readings in Figure 6b confirm this hypothesis.



(a) Running time results.

name	VX	VECC	VLPC	VLEC
jacobi-1d	0.0	-0.1	0.0	0.0
jacobi-rv	-2.3	-4.0	-12.9	-10.0
jacobi9-1d	0.0	-0.0	0.0	0.0
jacobi9-rv	-3.9	-6.3	-17.0	-14.3
seidel-1d	0.0	-0.1	0.0	0.0
seidel-rv	-5.9	-9.4	-25.5	-21.4
sobel-1d	1.5	-0.4	0.0	0.0
sobel-rv	-2.2	-4.6	-16.4	-12.9
transpose-1d	36.4	-0.0	0.0	0.0
transpose-rv	26.4	0.1	0.0	0.0

(b) Performance counters compared to NCC version (%).

Figure 6b shows the change in performance counters of the tensorized codes relative to the NCC-compiled codes. We used inline assembly to read the performance counter registers and flushed the pipeline before and after each reading. When RV is used in loop vectorization mode, the performance counters and also the observed runtimes do not differ much from NCC. However, tensorization lead to a reduction in loaded vector elements (VLEC) and memory traffic (VLPC) for the stencil codes. This is due to the memory re-use incurred by our memory chunking scheme. Additionally, the counters for vector execution clocks (VECC) and executed vector instruction (VX) decrease by about 4%.

Since there is no re-use in the matrix transposition, we also see no improvement in the performance counters and in fact tensorization does not help here.

5.1.2 VH: AVX512

The host system features a multi-tier cache hierarchy. In particular, each core of the host CPU has a 32KB L1 data cache. The data arrays in our stencil experiments have a row length of 8208 Bytes. Therefore, when the loop-vectorized stencils compute a result for position `(j,i)`, the input data at position `(j-1,i)` is still L1-resident. Since L1 cache is fast, our contiguous load re-use scheme does not translate to speedups for the comparably small stencils we are considering. The 9-point Jacobi stencil which has the largest load re-use of the stencil collection sees a speedup of about 20.5%. This indicates that larger stencils than considered here might benefit more from the re-use optimization still.

Matrix transpose does not benefit from the vector load chunking. However, we see a $\times 3.27$ speed up over loop vectorization in this case with a 4×4 brush. The tensorized transpose uses four vector load and four vector stores to process 16 `(i,j)`-positions at once. The accesses operate on eight different data rows (four from each input and output array). The actual transposition is performed with fast shuffle instructions.

In contrast, the loop vectorized version has a single contiguous load and resorts to a slow scatter instruction to write back the results. The loop vectorized version only computes eight results but needs to touch eight array rows as well.

6 Conclusion

Classic loop vectorization provides a productive means to utilize SIMD instructions without resorting to libraries or intrinsic functions. In this work, we propose loop nest tensorization a technique that lifts outer-loop vectorization to the multi-dimensional setting of tensor operations.

We also present techniques to generate efficient SIMD code from tensorized loop nest. SIMD architectures benefit from loop nest tensorization due to less memory accesses and vector register tiling.

We implemented our techniques in TensorRV, a prototype tensorizer based on the Region Vectorizer for LLVM IR. We evaluate TensorRV on a set of stencil codes and a matrix transposition kernel on AVX512 and NEC SX-Aurora TSUBASA. When the overlap of accessed elements is high, as is the case in stencil codes, tensorization leads to less memory accesses. We report speedups of up to $\times 1.45$ on NEC SX-Aurora for tensorized stencils. Matrix transposition still benefits from loop nest tensorization due to loop tiling and fast shuffles on AVX512, for which we report a $\times 3.27$ speedup.

We present TensorRV as a research platform for loop nest tensorization and leave brush size heuristics and optimizations such as data layout transformations to future work.

Acknowledgments

This work is supported by the Federal Ministry of Education and Research (BMBF) as part of the Metacca project.

References

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

[2] Alejandro Berna, Marta Jiménez, and José María Llabería. 2012. Source code transformations for efficient SIMD code generation.

[3] Diego Caballero, Sara Royuela, Roger Ferrer, Alejandro Duran, and Xavier Martorell. 2015. Optimizing Overlapped Memory Accesses in User-directed Vectorization. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*. 393–404. https://doi.org/10.1145/2751205.2751224

[4] Larry Carter, Jeanne Ferrante, and Susan Flynn Hummel. 1995. Hierarchical tiling for improved superscalar performance. In *Proceedings of IPPS '95, The 9th International Parallel Processing Symposium, April 25-28, 1995, Santa Barbara, California, USA*. 239–245. https://doi.org/10.1109/IPPS.1995.395939

[5] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine A. Yelick. 2009. Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors. *SIAM Rev.* 51, 1 (2009), 129–159. https://doi.org/10.1137/070693199

[6] 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 2004, Washington, DC, USA, June 9-11, 2004*. 82–93. https://doi.org/10.1145/996841.996853

[7] Michael Haidl, Simon Moll, Lars Klein, Huihui Sun, Sebastian Hack, and Sergei Gorlatch. 2017. PACXXv2 + RV: An LLVM-based Portable High-Performance Programming Model. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2017, Denver, CO, USA, November 13, 2017*. 7:1–7:12. https://doi.org/10.1145/3148173.3148185

[8] Thomas Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. 2011. Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures. In *Compiler Construction - 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbr"ucken, Germany, March 26-April 3, 2011. Proceedings*. 225–245. https://doi.org/10.1007/978-3-642-19861-8_13

[9] Kazuhisa Ishizaka, Kazushi Marukawa, Erich Focht, Simon Moll, Matthias Kurtenacker, and Sebastian Hack. 2018. NEC SX-Aurora - A Scalable Vector Architecture (poster). *US LLVM Developers' Meeting* (2018). http://compilers.cs.uni-saarland.de/papers/nec_poster_llvmdev18.pdf

[10] Ralf Karrenberg and Sebastian Hack. 2012. Improving Performance of OpenCL on CPUs. In *Compiler Construction - 21st International Conference, CC 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 1–20. https://doi.org/10.1007/978-3-642-28652-0_1

[11] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. 75–88. https://doi.org/10.1109/CGO.2004.1281665

[12] Florian Lemaitre, Benjamin Couturier, and Lionel Lacassagne. 2018. Small SIMD Matrices for CERN High Throughput Computing. In *Proceedings of the 4th Workshop on Programming Models for SIMD/Vector Processing, WPMVP@PPoPP 2018, Vienna, Austria, February 24, 2018*. 1:1–1:8. https://doi.org/10.1145/3178433.3178434

[13] Tareq M. Malas, Georg Hager, Hatem Ltaief, and David E. Keyes. 2018. Multidimensional Intratile Parallelization for Memory-Starved Stencil Computations. *TOPC* 4, 3 (2018), 12:1–12:32. https://doi.org/10.1145/3155290

[14] Simon Moll and Sebastian Hack. 2018. Partial control-flow linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 543–556. https://doi.org/10.1145/3192366.3192413

[15] M. Pharr and W. R. Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)*. 1–13. https://doi.org/10.1109/InPar.2012.6339601

[16] Sebastian Pop, Albert Cohen, and Georges-Andr'e Silber. 2005. Induction Variable Analysis with Delayed Abstractions. In *High Performance Embedded Architectures and Compilers, First International Conference, HiPEAC 2005, Barcelona, Spain, November 17-18, 2005, Proceedings*. 218–232. https://doi.org/10.1007/11587514_15

[17] Lakshminarayanan Renganarayanan, Uday Bondhugula, Salem Derisavi, Alexandre E. Eichenberger, and Kevin O'Brien. 2009. Compact multi-dimensional kernel extraction for register tiling. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*. https://doi.org/10.1145/1654059.1654105

[18] Christopher Rodrigues, Amarin Phaosawasdi, and Peng Wu. 2018. SIMDization of Small Tensor Multiplication Kernels for Wide SIMD Vector Processors. In *Proceedings of the 4th Workshop on Programming Models for SIMD/Vector Processing, WPMVP@PPoPP 2018, Vienna, Austria, February 24, 2018*. 3:1–3:8. https://doi.org/10.1145/3178433.3178436

[19] Diogo Sampaio, Rafael Martins de Souza, Sylvain Collange, and Fernando Magno Quintão Pereira. 2013. Divergence analysis. *ACM Trans. Program. Lang. Syst.* 35, 4 (2013), 13:1–13:36. https://doi.org/10.1145/2523815

[20] Kevin Stock, Thomas Henretty, Iyyappa Murugandi, P. Sadayappan, and Robert J. Harrison. 2011. Model-Driven SIMD Code Generation for a Multi-resolution Tensor Kernel. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*. 1058–1067. https://doi.org/10.1109/IPDPS.2011.101

[21] Charles Yount. 2015. Vector Folding: Improving Stencil Performance via Multi-dimensional SIMD-vector Representation. In *17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, August 24-26, 2015*. 865–870. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.27