A Data Layout Transformation for Vectorizing Compilers

Arsène Pérard-Gayot
Computer Graphics Lab
Saarland University
Saarbrücken, Germany
perard@cg.uni-saarland.de

Richard Membarth
DFKI
Saarland University
Saarbrücken, Germany
richard.membarth@dfki.de

Philipp Slusallek
DFKI
Saarland University
Saarbrücken, Germany
slusallek@dfki.de

Simon Moll
Compiler Design Lab
Saarland University
Saarbrücken, Germany
moll@cs.uni-saarland.de

Roland Leißa
Compiler Design Lab
Saarland University
Saarbrücken, Germany
leissa@cs.uni-saarland.de

Sebastian Hack
Compiler Design Lab
Saarland University
Saarbrücken, Germany
hack@cs.uni-saarland.de

ABSTRACT

Modern processors are often equipped with vector instruction sets. Such instructions operate on multiple elements of data at once, and greatly improve performance for specific applications. A programmer has two options to take advantage of these instructions: writing manually vectorized code, or using an auto-vectorizing compiler. In the latter case, he only has to place annotations to instruct the auto-vectorizing compiler to vectorize a particular piece of code. Thanks to auto-vectorization, the source program remains portable, and the programmer can focus on the task at hand instead of the low-level details of intrinsics programming. However, the performance of the vectorized program strongly depends on the precision of the analyses performed by the vectorizing compiler. In this paper, we improve the precision of these analyses by selectively splitting stack-allocated variables of a structure or aggregate type. Without this optimization, automatic vectorization slows the execution down compared to the scalar, non-vectorized code. When this optimization is enabled, we show that the vectorized code can be as fast as hand-optimized, manually vectorized implementations.

CCS CONCEPTS

• **Computing methodologies → Vector / streaming algorithms**; • **Software and its engineering → Compilers**; *Software performance*;

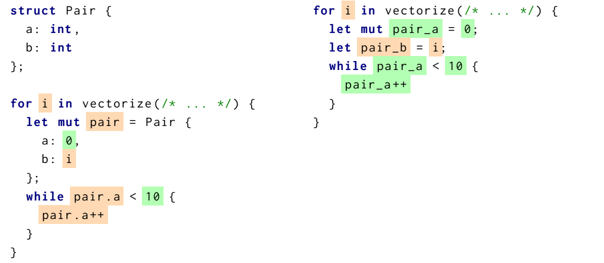
KEYWORDS

Vectorization, Compiler, Optimization

ACM Reference Format:
Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, Simon Moll, Roland Leißa, and Sebastian Hack. 2018. A Data Layout Transformation for Vectorizing Compilers. In *WPMVP'18 : Workshop on Programming Models for SIMD/Vector Processing, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3178433.3178440

1 INTRODUCTION

Vectorization is the process of transforming a program that operates on single elements of data into a program that operates on arrays of data. This transformation requires to convert the control flow of the input program to data flow, by inserting masking instructions and guards. As a result, the program can be executed on a machine that provides vector instructions, and execute faster. In order to perform this transformation, developers can either use compiler intrinsics and directly write vector code, or use a semi-automatic vectorizing compiler and write annotations in the parts of the program that should be vectorized. The advantages of the second approach are clear: The programmer can write cleaner, more portable code with less effort.



```
struct Pair {
  a: int,
  b: int
};

for i in vectorize(/* ... */) {
  let mut pair = Pair {
    a: 0,
    b: i
  };
  while pair.a < 10 {
    pair.a++
  }
}
```

(a) Code before transformation

```
for i in vectorize(/* ... */) {
  let mut pair_a = 0;
  let pair_b = i;
  while pair_a < 10 {
    pair_a++
  }
}
```

(b) Equivalent code after tranformation

Figure 1: Result of the vectorization analysis before and after our transformation. Uniform and varying expressions are colored in green and orange respectively.

Using state-of-the-art guided vectorization, the programmer only places a minimal amount of annotations, and the compiler analyzes the program to determine the *shape* of every variable or statement. For the sake of simplicity, such shapes can either be: *varying*, in which case the variable or statement will be vectorized; or *uniform*, in which case it will remain scalar. However, these analyses do not detect when only part of a structure or aggregate type is varying

WPMVP'18 , February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5646-6/18/02…$15.00
https://doi.org/10.1145/3178433.3178440

and the rest is uniform, and therefore will mark all their members as *varying* (see Figure 1a).

To solve this issue, we introduce a transformation that is executed in fixed point with the vectorization analysis. This transformation operates on the input scalar program and breaks aggregates or structures that are assigned a varying shape into smaller parts. The result (see Figure 1b) is a program in which the varying members of the aggregates are separated from the uniform ones. This transformation may impact the way control flow is vectorized. For instance, if the shape of a loop condition was changed from varying to uniform, masking is no longer necessary, because all lanes enter and exit the loop together. This, in turn, results in a spectacular performance improvement, as shown in Section 4.

2 RELATED WORK

We first review standard vectorization techniques, and then present region vectorization. Finally, we describe support for vectorization in programming languages.

2.1 Traditional Approaches

2.1.1 Superword Level Parallelism (SLP). Vectorization on straight-line code exploits SLP: The compiler tries to merge several scalar operations into a vector operation. This can be done on a per-basic-block level [13] or in the presence of control flow [23]. SLP algorithms will usually give up if the exact number of needed instructions cannot be fed into the SIMD lanes. *Padded SLP* tries to overcome this limitation by injecting redundant instructions [22]. *Throttled SLP* uses a cost model in order to estimate whether SLP vectorization is actually worthwhile at all [21].

2.1.2 Loop Vectorization. Allen et al. [1, 2] present a technique to translate loop nests to array statements. An alternative is *outer loop vectorization* using a so-called *unroll-and-jam* technique [3, 16, 20]: A chosen outer loop is unrolled several times while the resulting loop bodies are re-fused ("jammed"). There is a good chance that the instructions stemming from the same instruction in the original version can be grouped into SIMD instructions.

Other work on loop vectorization also considers data alignment, reductions [19], and interleaved data accesses [18]. Furthermore, the polyhedral model [7]—a powerful mathematical loop analysis framework using Presburger sets—has also been instrumented for loop vectorization [17, 25].

2.2 The Region Vectorizer

The *region vectorizer (RV)*[1] is a state-of-the-art vectorization framework based on LLVM. RV is derived from the *whole-function vectorizer* (WFV) [10–12]. RV vectorizes regions, which are single-entry, multi-exit subgraphs of the Control-Flow Graph (CFG). If the region encapsulates a loop nest, RV performs outer-loop vectorization. If the region contains the whole CFG, RV will vectorize the whole function.

RV maps each SIMD lane to one instance of a region in a CFG. The region instances execute in SPMD-like fashion, meaning that RV assumes that there are no data races between SIMD lanes.

2.2.1 Analysis and Transformations. RV operates in three main phases. First, it performs a *divergence analysis* [5, 12, 14] that assigns each instruction and branch a *vector shape*. This vector shape determines which instructions and branches in the region will behave uniformly across the instances of the region.

RV uses a sophisticated lattice to keep track of each instruction's vector shape [8]. However, for the purpose of the paper it is sufficient to just differentiate between the vector shapes *uniform* and *varying* (see Section 1). Optimistically, RV assigns each instruction *uniform* (the bottom element in the lattice), and ascends in the lattice as required in a fixed-point iteration.

Second, because SIMD CPUs can not handle divergent branches in hardware—unlike GPUs, RV *linearizes* divergent control by if-conversion: RV emulates the original control flow by inserting bit operations and masking instructions. This linearization and the additional masking operations take their toll on the program's performance. Thus, it is a good idea to keep control flow *uniform* whenever possible.

Finally, the vector code generator emits vector instructions, thereby concluding the vectorization process.

2.2.2 Vectorization of stack objects. Due to the SPMD semantics, if the code makes use of stack-allocated objects (***alloca*** in LLVM) RV assumes that each SIMD lane sees its own copy of the object. RV vectorizes stack objects in the general case by replicating the structure in an array (array-of-struct) such that each SIMD lane receives its own instance. Vector accesses to these arrays are inefficient since if all SIMD threads access an element of their stack object in lock step, the accessed pointers will have a large stride. However, RV employs several optimizations to generate efficient data layouts for vectorized stack-allocated objects.

First, if all SIMD lanes access the same offsets and write only uniform values, the stack object remains scalar. Second, if all SIMD lanes access the same offsets of the stack object but write *varying* values, RV changes the layout of the object to struct-of-array instead of array-of-struct [24, 26]. For example, a scalar stack object with the type $\textbf{struct}\{\textbf{int}, \textbf{int}\}$ will be replicated as $\textbf{struct}\{[\textbf{int} \times N], [\textbf{int} \times N]\}$ where N refers to the vectorization factor.

2.3 Support in Programming Languages

Programming languages usually incorporate some mechanism to allows programmers to vectorize code. For instance, C/C++ compilers provide short vector data types that can be easily mapped to hardware vector units. Other languages, like APL [6], Vector Pascal [4], MatLab/Octave, or FORTRAN, and language extensions like ArBB [15] provide operations on arrays.

In order to instruct the compiler to vectorize a particular piece of code, languages can also provide a way to annotate a scalar program. In C/C++, Intel® Cilk™ Plus [9] and OpenMP 4.0 allow to place preprocessor directives in front of loops or functions. For the benchmarks used in the evaluation section of this paper, we integrated RV into the programming language Impala (see 4.1). The vectorization annotation is naturally added to the language through higher-order functions.

[1] see https://github.com/cdl-saarland/rv.

```
struct Parent {
  a: Child,
  b: int
};

struct Child {
  c: int,
  d: int
};

for i in vectorize(/* ... */) {
  let p = Parent {
    a: Child { c: 0, d: 1 },
    b: i
  };
}
```

(a) Before the first iteration

```
struct Child {
  c: int,
  d: int
};

for i in vectorize(/* ... */) {
  let p_a = Child { c: 0, d: 1 };
  let p_b = i;
}
```

(b) After the first iteration

Figure 2: Applying the transformation in a fix point with the vectorization analysis ensures it is only applied when necessary. Here, only one layer of a nested structure is split, because the vectorization analysis is more precise after one fix point iteration. Uniform and varying expressions are colored in green and orange respectively.

3 ANALYSIS AND TRANSFORMATION

Our analysis and transformation is integrated in RV (see subsection 2.2). Therefore, our transformation operates on the LLVM Intermediate Representation (IR), but the description we give here is valid for any other SSA-based IR.

Our transformation is applied in a fixed-point loop: We first run the vectorization analysis of RV. Then, we look for stack allocated structures or array of structures, and split those that have a varying shape. We repeat those two steps until the transformation has nothing left to split.

With such a design, the transformation is selective: In the case of a varying structure containing other structures, the first iteration of the transformation will only split the parent structure. In the next iteration, the vectorization analysis will be more precise, because the members of the parent structure will be split: They will now have their own shape. Therefore, the transformation will only split the members of the innermost structure if necessary (see Figure 2).

3.1 Notation

In order to present the transformation in a concise and formal manner, we introduce a simplified version of the LLVM IR in Figure 3.

This IR represents the relevant parts of the LLVM IR for our setting. In particular, it models LLVM instructions such as ***load***, ***store***, ***alloca***, or ***getelementptr***. A ***load*** takes only one pointer argument and loads the corresponding piece of memory into a register. A ***store*** takes a value to store and a pointer to store the value to. An ***alloca*** allocates stack memory to hold a value of the given type and returns a pointer to the new chunk of memory. Note that this instruction is not to be confused with the homonymous GNU C library function: Its purpose is to model stack allocated variables. The ***getelementptr*** instruction performs pointer arithmetic on the argument with the given indices. The first index is multiplied by the size of the pointed object and added to the pointer. The meaning of the next index depends on the type of the pointer. If the pointer points to a structure, then it represents the index to a structure member. If the pointer points to an array, then it represents the index into the array. The following indices are interpreted in the same way, creating a path inside the type of the memory location. For instance, if a pointer p has type $[\textbf{\textit{struct}}\{int, int\} \times 2]*$, then the instruction ***getelementptr*** p ***indices*** $0,\ 1,\ 0$ returns a pointer that points to the first member of the structure in the second element of the array pointed by p.

The *uses* of an instruction I are denoted with $U(I)$: it is a set containing all instructions that *use* I as an operand.

3.2 Analysis

The goal of the transformation is to split structures or arrays of structures allocated on the stack. Before doing so, we must ensure that the transformation is desirable (the structure is marked as varying) and valid (the address of the stack object is not used in incorrect ways). For these reasons, Algorithm 2 inspects every stack object and returns whether or not the transformation should be applied: For every varying ***alloca*** in the program, the algorithm checks if the allocated type is a structure or an array of structures, and if so, analyzes the uses of the ***alloca*** to ensure that they can be transformed (by calling the function AnalyzeUses in Algorithm 1). For instance, we must prevent the following instruction sequence from being transformed:

$$
\begin{aligned}
I_1 &= \textbf{\textit{alloca}}\ \textbf{\textit{struct}}\{\textbf{\textit{int}}, \textbf{\textit{int}}\}* \\
I_2 &= \textbf{\textit{alloca}}\ \textbf{\textit{struct}}\{\textbf{\textit{int}}, \textbf{\textit{int}}\} \\
I_3 &= \textbf{\textit{store}}\ I_2\ \textbf{\textit{to}}\ I_1
\end{aligned}
$$

In this example, the pointer I_2 is stored to the memory location pointed by I_1, which makes precise tracking of instructions writing to I_2 generally impossible.

We also make sure that the users of an alloca to transform can only be ***load***s, ***store***s, or ***getelementptr***s. In the case of a ***getelementptr***, we check the indices to verify that the transformation is possible. In particular, we reject the following instruction sequence:

$$
\begin{aligned}
I_1 &= \textbf{\textit{alloca}}\ \textbf{\textit{struct}}\{\textbf{\textit{int}}, \textbf{\textit{int}}\} \\
I_2 &= \textbf{\textit{getelementptr}}\ I_1\ \textbf{\textit{indices}}\ 5, 0
\end{aligned}
$$

In this case, I_2 points to memory outside of the region allocated by I_1: The ***getelementptr*** offsets the pointer in I_1 by 5 times the size of $\textbf{\textit{struct}}\{\textbf{\textit{int}}, \textbf{\textit{int}}\}$.

The users of a ***getelementptr*** have to be analyzed as well if the pointer arithmetic does not descend into the ***alloca***. To illustrate this point, consider the program:

$$
\begin{aligned}
I_1 &= \textbf{\textit{alloca}}\ \textbf{\textit{struct}}\{\textbf{\textit{int}}, \textbf{\textit{int}}\} \\
I_2 &= \textbf{\textit{getelementptr}}\ I_1\ \textbf{\textit{indices}}\ 0 \\
I_3 &= \textbf{\textit{getelementptr}}\ I_2\ \textbf{\textit{indices}}\ 0, 1
\end{aligned}
$$

$$\begin{array}{lll} I ::= \textbf{\textit{load}}\ I & \textit{load a value from a memory location} \\ \quad \mid \textbf{\textit{store}}\ I\ \textbf{\textit{to}}\ I & \textit{store a value to a memory location} \\ \quad \mid \textbf{\textit{getelementptr}}\ I\ \textbf{\textit{indices}}\ \bar{I} & \textit{perform pointer arithmetic} \\ \quad \mid \textbf{\textit{alloca}}\ T & \textit{allocate an object on the stack} \\ \quad \mid \textbf{\textit{struct}}\ \{\bar{I}\} & \textit{a structure value} \\ \quad \mid N\,,\ N \in \mathbb{N} & \textit{an integer constant} \\ \quad \mid \textbf{\textit{inst}}\ \bar{I} & \textit{any other instruction} \end{array}$$

$$\begin{array}{ll} T ::= \textbf{\textit{int}} & \textit{integer type} \\ \quad \mid \textbf{\textit{struct}}\{\overline{T}\} & \textit{structure type} \\ \quad \mid [T \times N]\,,\ N \in \mathbb{N} & \textit{array type} \\ \quad \mid T* & \textit{pointer type} \end{array}$$

Figure 3: Program syntax and types for our transformation.

In this small sequence of instructions, I_2 is in fact pointing to the same memory location as I_1. We must then also analyze its single use I_3. Since I_3 points to the second member of the structure, its uses do not need to be analyzed.

The program in Algorithm 1 formalizes these constraints.

3.3 Transformation

Once the analysis determines that the transformation is valid, we split the ***alloca***. This process follows the same principle as the analysis: We start by creating one ***alloca*** per structure member, then replace the uses of the original ***alloca***. For example, we may perform the following transformation:

$$\begin{array}{l} I_1 = \textbf{\textit{alloca struct}}\{\textbf{\textit{int}}, \textbf{\textit{int}}\} \\ I_2 = \textbf{\textit{getelementptr}}\ I_1\ \textbf{\textit{indices}}\ 0, 1 \\ I_3 = \textbf{\textit{store}}\ 5\ \textbf{\textit{to}}\ I_2 \end{array} \Rightarrow \begin{array}{l} I_1 = \textbf{\textit{alloca int}} \\ I_2 = \textbf{\textit{alloca int}} \\ I_3 = \textbf{\textit{store}}\ 5\ \textbf{\textit{to}}\ I_2 \end{array}$$

If the original program contains a ***load*** or ***store*** to the entire alloca, we have to replace it by as many ***load***s or ***store***s as there are structure members, as in the following example:

$$\begin{array}{l} I_1 = \textbf{\textit{alloca struct}}\{\textbf{\textit{int}}, \textbf{\textit{int}}\} \\ I_2 = \textbf{\textit{load}}\ I_1 \end{array} \Rightarrow \begin{array}{l} I_1 = \textbf{\textit{alloca int}} \\ I_2 = \textbf{\textit{alloca int}} \\ I_3 = \textbf{\textit{load}}\ I_1 \\ I_4 = \textbf{\textit{load}}\ I_2 \\ I_5 = \textbf{\textit{struct}}\ \{I_3, I_4\} \end{array}$$

4 RESULTS

4.1 Benchmarks

In order to evaluate our approach, we implemented two benchmarking programs. The first one is a vectorized version of Bresenham's line drawing algorithm, and the second is a vectorized ray-tracing kernel. These two programs differ in complexity: While Bresenham's algorithm represents only a few lines of code, the ray-tracing program is more representative of real-world applications, as it amounts to approximately 1K lines of code.

Both of these programs are implemented using Impala, a dialect of Rust. In Impala, vectorization is triggered with the function `vectorize`. Impala then orders RV to vectorize the provided piece of code.

```
for i in vectorize(vec_width, default_alignment, 0, N) {
  /* ... */
}
```

In this example, the loop counter `i` is marked by Impala as being varying. Every variable captured from the inside of the `vectorize` block is assumed to be uniform across SIMD lanes, and marked uniform by Impala. RV then uses this initial information during the vectorization analysis.

4.1.1 Line Drawing.

Bresenham's line drawing algorithm is an algorithm to plot lines on a bitmap image. The core of its implementation is given below:

```
fn plot_line(line: &Line, plot: fn (i32, i32) -> ()) -> () {
    let dx = (line.x1 - line.x0) as f32;
    let dy = (line.y1 - line.y0) as f32;
    let de = fabsf(dy / dx);
    let ky = if line.y1 > line.y0 { 1 } else { 0 };

    let mut e = 0.0f;
    let mut y = line.y0;
    for x in range(line.x0, line.x1) {
        plot(x, y);
        e += de;
        if e > 0.5f {
            y += ky;
            e -= 1.0f;
        }
    }
}
```

We vectorize it by assigning each SIMD lane a line which differs from the others only by the y-coordinate of its endpoints.

4.1.2 Ray-tracing.

Our ray-tracing benchmark uses a Bounding Volume Hierarchy (BVH) to compute the intersection between a ray and a 3D scene. The BVH is a tree containing a bounding box in each inner node, and list of triangles in the leaves. The children of a node are always contained in the bounding box of their parent.

The algorithm takes a ray as input and recursively descends into the tree, culling nodes whose bounding boxes are not intersected by the ray. In Impala, the ray-box intersection function is the following, and only consists in floating point arithmetic followed by min/max pairs:

Algorithm 1 Analysis for the uses of an instruction.

```
Inputs:
    $I$: Instruction to analyze
    $Arr$: $True$ iff $I$ is an alloca of array type
    $Off$: $True$ iff the first getelementptr index can be non-zero
Output:
    $True$ iff $I$ can be transformed

function AnalyzeUses($I$, $Arr$, $Off$)
    for $J \in U(I)$ do
        switch J
            case load $J_1$
                break

            case store $J_1$ to $J_2$
                if $J_1 = I$ then
                    return $False$                              ▷ Pointers cannot escape
                end if
                break

            case getelementptr $J_1$ indices $J_2, J_3, \ldots J_n$
                for $k \in [2, n]$ do                            ▷ Check indices
                    if $J_k \notin \mathbb{N}$ & $!(Arr \wedge k = 3)$ then
                        return $False$                          ▷ Non-constant indexing
                    end if
                    if $!Off$ & $k = 2$ & $J_k \neq 0$ then
                        return $False$                          ▷ First index must be zero
                    end if
                end for
                if $Arr$ then                                   ▷ Analyze uses of getelementptr
                    if $n < 4$ then
                        R ← AnalyzeUses(J, $n = 2$, $n = 3$)
                    end if
                else
                    if $n < 3$ then
                        R ← AnalyzeUses(J, $False$, $False$)
                    end if
                end if
                if !R then
                    return $False$                              ▷ Uses break constraints
                end if
                break

            default
                return $False$                                  ▷ Others instructions

        end switch
    end for
    return $True$
end function
```

```
fn @intersect_ray_box(math: Math, ray: Ray, bbox: BBox)
  -> (bool, float, float) {
  let t0 = vec3_add(vec3_mul(ray.inv_dir, bbox.min), ray.inv_org);
  let t1 = vec3_add(vec3_mul(ray.inv_dir, bbox.max), ray.inv_org);

  let (tentry, texit) =
    (math.fmaxmaxf(math.fminf(t0.x, t1.x), math.fminf(t0.y, t1.y),
       math.fminmaxf(t0.z, t1.z, ray.tmin)),
     math.fminminf(math.fmaxf(t0.x, t1.x), math.fmaxf(t0.y, t1.y),
       math.fmaxminf(t0.z, t1.z, ray.tmax)))

  (tentry <= texit, tentry, texit)
}
```

Algorithm 2 Analysis for a single instruction.

```
Inputs:
    $I$: Instruction to analyze
    $S$: Map from instruction to vector shape
Output:
    $True$ iff $I$ can be transformed

function AnalyzeInstruction($I$, $S$)
    if $S(I) \neq$ varying then
        return $False$                          ▷ $I$ must be varying
    end if
    if $I$ = alloca $T$ then
        if $T = [T_1 \times N]$ then            ▷ Analyze arrays of structs
            S ← $T_1$
            Arr ← $True$
        else
            S ← T
            Arr ← $False$
        end if
        if $S = struct\{S_1, \ldots S_n\}$ then
            return AnalyzeUses($I$, $Arr$, $False$)
        else
            return $False$                      ▷ Only applies to structs
        end if
    else
        return $False$
    end if
end function
```

Once the algorithm reaches a leaf of the tree, the triangles contained in it are intersected with the ray, and only the closest intersection is kept. A typical use case for this algorithm is a global illumination renderer, where many of such queries have to be performed to produce a single image.

The algorithm is vectorized by assigning every SIMD lane a different ray:

```
for i in vectorize(vec_width, default_alignment, 0, N) {
  let ray = load_ray(i);
  let hit = traverse_bvh(bvh, ray);
  store_hit(i, hit);
}
```

Inside the `traverse_bvh` function, nodes are pushed on the stack whenever *any* ray intersects their bounding boxes:

```
let (mask, tentry, texit) =
  intersect_ray_box(math, ray, node.bbox);
if any(mask) {
  if any(stack.top().tmin < tentry) {
    stack.push(child_id, tentry)
  } else {
    stack.push_after(child_id, tentry)
  }
}
```

Nodes are pushed in an approximate order on the stack, and we store their distance along the ray. This approximate sorting makes finding the closest intersection faster, and the distance can be used to cull nodes.

Scene	With opt.	Without opt.	Scalar
Sponza	5.79 (×17)	0.34	0.88
Crown	15.33 (×14)	1.08	3.27
San-Miguel	3.13 (×16)	0.19	0.53
Powerplant	6.69 (×15)	0.45	1.38

Table 1: Performance in Mray/s (higher is better) of the ray tracing algorithm with vectorization enabled (with and without our transformation) and disabled (the scalar variant), in different scenes. The speedups in parentheses are reported with respect to the version with vectorization enabled without our transformation.

With opt.	Without opt.	Scalar
32.31 (×2)	15.41	14.63

Table 2: Performance in Mline/s (higher is better) of the line drawing algorithm with vectorization enabled (with and without our transformation) and disabled (the scalar variant). The speedups in parentheses are reported with respect to the version with vectorization enabled without our transformation.

4.2 Performance

We compare the performance of our benchmarks in three versions: Vectorized with our optimization enabled, vectorized with our optimization disabled, and non-vectorized (scalar).

Overall, Table 1 shows that our transformation greatly improves performance, reaching sometimes up to 17× the speed of the non-optimized version. Without the transformation, vectorization is virtually useless, as the resulting code is slower than the non-vectorized version.

The reason for this huge gap is that our transformation makes the analyses in RV more precise. In our ray-tracing algorithm, the traversal stack is represented as an array of structures containing the distance to the node and its index into the array of nodes. Before the optimization, the stack is kept as a whole and therefore marked as varying by RV: The generated vector code then assumes that the control flow is not uniform, and generates masks and guards. After the optimization, the stack is split in two: One stack for the distance, and one for the index. RV marks the former as varying, but keeps the latter uniform, which in turn makes the traversal loop uniform. Therefore, no masks or guards are needed, and the generated code is similar to the hand written code in the Embree library, a ray tracing library by Intel (see Figure 4). Its performance is also similar, always within 10% of Embree for the tested scenes.

With the simpler line drawing algorithm (Table 2), the difference is only a factor of 2 over the vectorized variant. This shows that our transformation is more beneficial to complex algorithms. Indeed, smaller programs have simpler control flow, and hence fewer opportunities to remove redundant masking logic.

5 CONCLUSION

We have described a transformation that operates on an SSA-based IR. This transformation splits stack-allocated objects in order to improve existing vectorization analyses.

We proved that without our transformation, automatic vectorization fails to generate efficient code for a ray-tracing algorithm. For this algorithm, vectorization without our transformation is not a viable option, since the resulting code is slower than the scalar version. When our transformation is applied, the performance of the algorithm is within 10% of a hand-vectorized library by Intel.

Because this transformation is selective, it will not affect the whole program, but only the parts where the transformation is beneficial. Hence, the programmer can now use structures, arrays and aggregates without performance penalties.

ACKNOWLEDGMENTS

This work is supported by the Federal Ministry of Education and Research (BMBF) as part of the Metacca and ProThOS projects as well as by the Intel Visual Computing Institute (IVCI) and Cluster of Excellence on Multimodal Computing and Interaction (MMCI) at Saarland University.

REFERENCES

[1] John R. Allen, Ken Kennedy, Carrie Porterfield, and Joe D. Warren. 1983. Conversion of Control Dependence to Data Dependence. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983.* 177–189. https://doi.org/10.1145/567067.567085

[2] 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

[3] Randy Allen and Ken Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann.

[4] W. Paul Cockshott. 2002. Vector Pascal an array language for multimedia code. In *APL.* 83–91. https://doi.org/10.1145/602231.602242

[5] Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintão Pereira, and Wagner Meira Jr. 2011. Divergence Analysis and Optimizations. In *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011.* 320–329. https://doi.org/10.1109/PACT.2011.63

[6] Adin D. Falkoff and Kenneth E. Iverson. 1973. The Design of APL. *IBM Journal of Research and Development* 17, 5 (1973), 324–334. https://doi.org/10.1147/rd.174.0324

[7] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (1991), 23–53. https://doi.org/10.1007/BF01407931

[8] 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

[9] Intel Corporation. 2013. *Intel® Cilk™ Plus Language Extension Specification* (version 1.2 ed.).

[10] Ralf Karrenberg. 2015. *Automatic SIMD Vectorization of SSA-based Control Flow Graphs.* Springer. https://doi.org/10.1007/978-3-658-10113-8

[11] Ralf Karrenberg and Sebastian Hack. 2011. Whole-function vectorization. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011.* 141–150. https://doi.org/10.1109/CGO.2011.5764682

[12] 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

[13] Samuel Larsen and Saman P. Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000.* 145–156. https://doi.org/10.1145/349299.349320

```
    vmovdqa      160(%rsp), %ymm12
    vmovdqa      128(%rsp), %ymm1
    vextracti128 $1, %ymm1, %xmm1
    vmovaps      480(%rsp), %ymm5
    vaddps       %ymm9, %ymm5, %ymm9
    vfmadd213ps  64(%rsp), %ymm4, %ymm6
    vmovaps      96(%rsp), %ymm2
    vfmadd132ps  32(%rsp), %ymm2, %ymm3
    vblendvps    %ymm11, %ymm14, %ymm0, %ymm10
    vfmadd132ps  192(%rsp), %ymm5, %ymm0
    testb        $1, %r15b
    je           .LBB12_1439

    vextracti128 $1, %ymm7, %xmm2
    vpextrq      $1, %xmm2, %rax
    vextractf128 $1, %ymm13, %xmm2
    vinsertps    $48, (%rax), %xmm2, %xmm2
    vinsertf128  $1, %xmm2, %ymm13, %ymm13
.LBB12_1439:
    vmovdqa      672(%rsp), %ymm7
    vfmadd213ps  64(%rsp), %ymm4, %ymm10
    vblendvps    %ymm11, %ymm13, %ymm0, %ymm2
    vmovaps      96(%rsp), %ymm4
    vfmadd132ps  32(%rsp), %ymm4, %ymm2
    vmovdqa      128(%rsp), %ymm4
    vpacksswb    %xmm1, %xmm4, %xmm11
    vpminsd      %ymm0, %ymm9, %ymm1
    vpmaxsd      %ymm9, %ymm0, %ymm0
    vpminsd      %ymm10, %ymm6, %ymm4
    vpmaxsd      %ymm6, %ymm10, %ymm5
    vpminsd      %ymm2, %ymm3, %ymm6
    vpmaxsd      %ymm3, %ymm2, %ymm2
    vpmaxsd      %ymm1, %ymm4, %ymm1
    vpminsd      %ymm5, %ymm0, %ymm0
    vpmaxsd      %ymm6, %ymm7, %ymm3
    vpminsd      %ymm15, %ymm2, %ymm2
    vpmaxsd      %ymm1, %ymm3, %ymm9
    vpminsd      %ymm2, %ymm0, %ymm0
    vpcmpgtd     %ymm0, %ymm9, %ymm0
    vpxor        .LCPI12_60, %ymm0, %ymm0
    vextracti128 $1, %ymm0, %xmm1
    vpacksswb    %xmm1, %xmm0, %xmm0
    vpand        %xmm8, %xmm0, %xmm0
    vpmovzxwd    %xmm0, %ymm0
    vpslld       $31, %ymm0, %ymm0
    vpsrad       $31, %ymm0, %ymm0
    vextracti128 $1, %ymm0, %xmm1
    vmovq        %xmm1, %rax
    vpextrq      $1, %xmm1, %rcx
    vmovq        %xmm0, %rdx
    vpextrq      $1, %xmm0, %rsi
    orq          %rcx, %rsi
    orq          %rax, %rdx
    xorl         %eax, %eax
    orq          %rsi, %rdx
    setne        %cl
    je           .LBB12_1456
```

(a) Assembly without our transformation

```
vbroadcastss    -192(%r14,%rdi,4), %ymm3
vfmsub213ps     %ymm0, %ymm6, %ymm3
vbroadcastss    -128(%r14,%rdi,4), %ymm4
vfmsub213ps     %ymm10, %ymm1, %ymm4
vbroadcastss    -64(%r14,%rdi,4), %ymm2
vfmsub213ps     %ymm8, %ymm9, %ymm2
vbroadcastss    -160(%r14,%rdi,4), %ymm11
vfmsub213ps     %ymm0, %ymm6, %ymm11
vbroadcastss    -96(%r14,%rdi,4), %ymm12
vfmsub213ps     %ymm10, %ymm1, %ymm12
vbroadcastss    -32(%r14,%rdi,4), %ymm14
vpminsd         %ymm11, %ymm3, %ymm7
vpmaxsd         %ymm3, %ymm11, %ymm3
vfmsub213ps     %ymm8, %ymm9, %ymm14
vpminsd         %ymm12, %ymm4, %ymm11
vpmaxsd         %ymm4, %ymm12, %ymm4
vpminsd         %ymm14, %ymm2, %ymm12
vpmaxsd         %ymm2, %ymm14, %ymm2
vpmaxsd         %ymm7, %ymm11, %ymm7
vpminsd         %ymm4, %ymm3, %ymm4
vpmaxsd         %ymm12, %ymm15, %ymm3
vpminsd         %ymm13, %ymm2, %ymm2
vpmaxsd         %ymm7, %ymm3, %ymm3
vpminsd         %ymm2, %ymm4, %ymm2
vpcmpgtd        %ymm2, %ymm3, %ymm2
vpxor           .LCPI12_6, %ymm2, %ymm2
vptest          %ymm2, %ymm2
je              .LBB12_40
```

(b) Assembly with our transformation

```
vbroadcastss    64(%r14,%rsi,4), %ymm2
vmovaps         1152(%rsp), %ymm0
vmovaps         1184(%rsp), %ymm3
vmovaps         1216(%rsp), %ymm4
vfmsub213ps     %ymm12, %ymm0, %ymm2
vbroadcastss    128(%r14,%rsi,4), %ymm5
vfmsub213ps     %ymm11, %ymm3, %ymm5
vbroadcastss    192(%r14,%rsi,4), %ymm6
vfmsub213ps     %ymm13, %ymm4, %ymm6
vbroadcastss    96(%r14,%rsi,4), %ymm7
vfmsub213ps     %ymm12, %ymm0, %ymm7
vbroadcastss    160(%r14,%rsi,4), %ymm8
vfmsub213ps     %ymm11, %ymm3, %ymm8
vbroadcastss    224(%r14,%rsi,4), %ymm3
vfmsub213ps     %ymm13, %ymm4, %ymm3
vpminsd         %ymm7, %ymm2, %ymm0
vpminsd         %ymm8, %ymm5, %ymm4
vpminsd         %ymm3, %ymm6, %ymm9
vpmaxsd         %ymm4, %ymm0, %ymm0
vpmaxsd         %ymm9, %ymm0, %ymm0
vpmaxsd         %ymm7, %ymm2, %ymm2
vpmaxsd         %ymm8, %ymm5, %ymm4
vpmaxsd         %ymm3, %ymm6, %ymm3
vpminsd         %ymm4, %ymm2, %ymm2
vpminsd         %ymm3, %ymm2, %ymm2
vpmaxsd         1248(%rsp), %ymm0, %ymm3
vpminsd         768(%rsp), %ymm2, %ymm2
vcmple_oqps     %ymm2, %ymm3, %ymm2
vtestps         %ymm2, %ymm2
je              .LBB83_97
```

(c) Reference assembly in Embree

Figure 4: Comparison of the assembly generated by LLVM for the ray-box intersection routine of our ray-tracing algorithm. We show the x86 assembly after vectorization by RV with our optimization disabled (4a) and enabled (4b). We also provide the assembly generated by LLVM for the ray-box intersection routine of Embree (4c), a manually vectorized ray-tracing library by Intel. For brevety, we do not reproduce the listing in 4a in its entirety.

[14] Yunsup Lee, Ronny Krashinsky, Vinod Grover, Stephen W. Keckler, and Krste Asanovic. 2013. Convergence and scalarization for data-parallel architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*. 32:1–32:11. https://doi.org/10.1109/CGO.2013.6494995

[15] Chris J. Newburn, Byoungro So, Zhenying Liu, Michael D. McCool, Anwar M. Ghuloum, Stefanus Du Toit, Zhi-Gang Wang, Zhaohui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. 2011. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*. 224–235. https://doi.org/10.1109/CGO.2011.5764690

[16] Viet Nhu Ngo. 1995. *Parallel Loop Transformation Techniques for Vector-based Multiprocessor Systems*. Ph.D. Dissertation. Minneapolis, MN, USA. UMI Order No. GAX94-33091.

[17] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. 2011. Vapor SIMD: Auto-vectorize once, run everywhere. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*. 151–160. https://doi.org/10.1109/CGO.2011.5764683

[18] Dorit Nuzman and Richard Henderson. 2006. Multi-platform Auto-vectorization. In *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2006), 26-29 March 2006, New York, New York, USA*. 281–294. https://doi.org/10.1109/CGO.2006.25

[19] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*. 132–143. https://doi.org/10.1145/1133981.1133997

[20] Dorit Nuzman and Ayal Zaks. 2008. Outer-loop vectorization: revisited for short SIMD architectures. In *17th International Conference on Parallel Architecture and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008*.

2–11. https://doi.org/10.1145/1454115.1454119

[21] Vasileios Porpodas and Timothy M. Jones. 2015. Throttling Automatic Vectorization: When Less is More. In *2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*. 432–444. https://doi.org/10.1109/PACT.2015.32

[22] Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. 2015. PSLP: padded SLP automatic vectorization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*. 190–201. https://doi.org/10.1109/CGO.2015.7054199

[23] Jaewook Shin, Mary W. Hall, and Jacqueline Chame. 2005. Superword-Level Parallelism in the Presence of Control Flow. In *3nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005), 20-23 March 2005, San Jose, CA, USA*. 165–175. https://doi.org/10.1109/CGO.2005.33

[24] Artjoms Sinkarovs and Sven-Bodo Scholz. 2013. Semantics-preserving data layout transformations for improved vectorisation. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing, Boston, MA, USA, FHPC@ICFP 2013, September 25-27, 2013*. 59–70. https://doi.org/10.1145/2502323.2502332

[25] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. 2009. Polyhedral-Model Guided Loop-Nest Auto-Vectorization. In *PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 12-16 September 2009, Raleigh, North Carolina, USA*. 327–337. https://doi.org/10.1109/PACT.2009.18

[26] Shixiong Xu and David Gregg. 2014. Semi-automatic Composition of Data Layout Transformations for Loop Vectorization. In *Network and Parallel Computing - 11th IFIP WG 10.3 International Conference, NPC 2014, Ilan, Taiwan, September 18-20, 2014. Proceedings*. 485–496. https://doi.org/10.1007/978-3-662-44917-2_40