Auto-vectorization for Image Processing DSLs

Oliver Reiche Christof Kobylko Frank Hannig Jürgen Teich

Hardware/Software Co-Design, Department of Computer Science,
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Abstract

The parallelization of programs and distributing their workloads to multiple threads can be a challenging task. In addition to multi-threading, harnessing vector units in CPUs proves highly desirable. However, employing vector units to speed up programs can be quite tedious. Either a program developer solely relies on the auto-vectorization capabilities of the compiler or he manually applies vector intrinsics, which is extremely error-prone, difficult to maintain, and not portable at all.

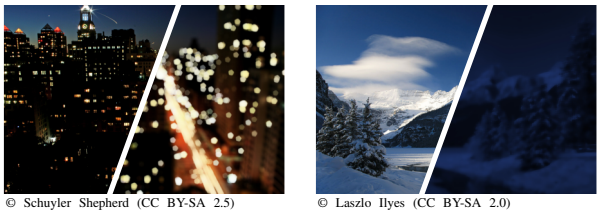
Based on whole-function vectorization, a method to replace control flow with data flow, we propose auto-vectorization techniques for image processing DSLs in the context of source-to-source compilation. The approach does not require the input to be available in SSA form. Moreover, we formulate constraints under which the vectorization analysis and code transformations may be greatly simplified in the context of image processing DSLs. As part of our methodology, we present control flow to data flow transformation as a source-to-source translation. Moreover, we propose a method to efficiently analyze algorithms with mixed bit-width data types to determine the optimal SIMD width, independently of the target instruction set. The techniques are integrated into an open source DSL framework. Subsequently, the vectorization capabilities are compared to a variety of existing state-of-the-art C/C++ compilers. A geometric mean speedup of up to 3.14 is observed for benchmarks taken from ISPC and image processing, compared to non-vectorized executions.

CCS Concepts • **Computing methodologies** → **Image processing**; • **Computer systems organization** → **Single instruction, multiple data**; • **Software and its engineering** → **Domain specific languages**

Keywords Domain-Specific Languages, Vectorization, Image Processing

1. Introduction

Central Processing Units (CPUs) offer a high degree of parallelism using multi-threading capabilities. Yet aside from multi-threading, many architectures possess Single Instruction Multiple Data (SIMD) units. Consequently, multiple instructions executed on different data lanes can be unified to a single vector instruction. This is particularly beneficial for highly data-parallel workloads, such as image processing algorithms, depicted in Figure 1. Modern embedded CPUs from ARM and Intel offer powerful vector instruction sets NEON and SSE, respectively. With the Goldmont architecture, Intel recently introduced SSE4.2 to its embedded Atom CPUs and the availability of the 256 bit-wide AVX2 instruction set is just a matter of time. Although Graphics Processing Units (GPUs) are primarily seen as the optimal target architecture for the domain of image processing, they might either be not available on an embedded architecture or introduce an intense overhead for transferring data to dedicated GPU memory. Therefore, CPUs may still be an attractive target, as they do not suffer from this drawback and are available in almost every embedded system.



© Schuyler Shepherd (CC BY-SA 2.5) © Laszlo Ilyes (CC BY-SA 2.0)

(a) Bokeh effect (b) Night filter

Figure 1. Typical postprocessing filters

To harness CPUs' vector units, vectorized code must either be manually written, or the developer must rely on the auto-vectorization features of the compiler. Explicitly writing vectorized code by hand is not only a very challenging task, it is also extremely error-prone, difficult to maintain, and not portable at all, as a commitment is made to a specific target instruction set. Consequently, relying on the compiler's auto-vectorization provides portability across multiple instruction sets and architectures. However, their vectorization analysis often fails in early stages, as they must guarantee functional correctness. A third option is to use languages explicitly tailored toward vectorization with certain keywords to guide the vectorization analysis, and thereby achieve better results.

Automatic vectorization is still an ongoing topic in research. An overview of the first compilers for this field of parallel computing is presented in [11]. Various other strategies have been developed relying on certain assumptions and making the amount of vectorizable code heavily dependent on the style of the source code.

One of the first techniques was the so-called inner-loop vectorization. It focused on the innermost loops in a control flow hierarchy and attempted to unroll them several times. Afterwards, the scalar variables inside the loop body were packed into vectors and the expressions were replaced by vector instructions. This method has been used in a number of experimental compilers [8] [22], as well as the GCC [13]. Major disadvantages of these methods are that entire programs cannot be vectorized and that loop-carried dependencies can quickly cause the vectorization to fail.

Another concept of vectorization is the Superword-Level Parallelization (SLP) [9]. This approach was inspired by Very Long Instruction Word (VLIW) processor architectures. It detects several *isomorphic expressions*[1] and replaces them with a series of vectorized instructions. One issue to deal with, is the possible aliasing of memory regions, which was addressed by the inter-procedural loop-oriented

LCTES'17, June 21–22, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-5030-3/17/06...$15.00
http://dx.doi.org/10.1145/3078633.3081039

[1] Isomorphic expressions are expressions, which execute the same operations in the same order on operands of the same type, e. g., $a = b + c \cdot d$ and $f = e + h \cdot g$ are isomorphic expressions.

pointer analysis in [24] to enable aggressive SLP optimizations. Another issue is the presence of diverging control flow, which can be handled for SLP by following the detailed description provided in [21]. There, the main problem is to find isomorphic expressions without data dependencies. Recently, *FlexVec* [3] was to proposed to handle cross-iteration dependencies by adding special extensions to the AVX-512 instruction set.

A more general approach is outer-loop vectorization [14]. It is based on vectorizing entire loop nests instead of only the inner-most loop. This increases the potential of the vectorization because more computational work needs to be done at higher control flow hierarchy levels, including nested loops. Unfortunately, this method cannot address diverging control flow inside the condition of the outer loops.

If-conversion [1] was one of the first transformations for translating control flow into data flow using predicated execution. Karrenberg and Hack addressed this problem in [6], specifically for control flow graphs in Single Static Assignment (SSA) form, with their proposed method *whole-function vectorization*. Their approach is based on the Single Program Multiple Data (SPMD) model, such as kernels in OpenCL or CUDA, which assumes that multiple instances of a program perform the same operations on a large amount of data elements. This assumption works very well for many domains, such as image processing or stencil computations. The same authors revisited their approach in [7], where they propose to retain control flow, however, not in the context of source-to-source compilation.

Exploiting the auto-vectorization features of compilers can be a quite tedious endeavor. It is often required to apply a specific coding style, such that there is still a strain put on the program developer. In order to reduce this requisite amount of work and to provide more performance transparency, special language extensions are being developed that provide a simplified and abstract way of addressing available SIMD units. Examples for these are the *Intel SPMD Program Compiler (ISPC)* [15], the *Sierra* language extensions for C++ [10], and special qualifiers for *OpenMP* proposed in [18]. Although grossly simplified, the vectorization process remains to be explicitly guided by the program developer.

Domain-Specific Languages (DSLs), such as *Halide* and *Spiral*, introduced in [16, 17], have removed this final stumbling block. With the use of such DSLs, the vectorization process can be guided by domain knowledge, instead of decisions explicitly made by the program developer. Thereby, algorithm designers, who are less familiar with low-level architectural details, can solely focus on algorithm design and let the compiler handle all code optimizations. However, existing techniques struggle to efficiently handle diverging control flow.

In this work, we will show that the automatic vectorization of diverging control flow can be efficiently handled by combining domain knowledge with whole-function vectorization. We will present speedups for codes generated with a DSL for image processing that are comparable to those of state-of-the-art compilers. Our contributions are as follows:

1. Adaptation of the work of Karrenberg and Hack [6] to DSL frameworks that (a) do not require input to be given in SSA form and (b) employ relative indexing for accessing memory. As a result, 5 vector markers can be reduced to just 2 (Section 3.2), which positively affects the vectorization analysis, and thereby may greatly simplify compiler development.
2. Source-to-source vectorization: In [7], Karrenberg and Hack proposed to retain some control flow when translating to data flow, which allows to skip the execution of entire branches. We show how to perform this transformation as a source-to-source compilation (Section 3.3). Aside from the possibility to take the fast path and skip the execution of entire branches, retaining control flow is particularly desirable for source-to-source compilation because of two reasons: First of all, it increases the readability of the generated source code for a human programmer. As DSLs are often used to generate only relevant code fractions of an algorithm, it might be necessary to manually adjust those codes before integrating them into a bigger program. Second, control flow statements guide several optimization stages of the target language compiler (e. g., loop-invariant code motion), which can only be performed if the control flow is not completely flattened.
3. Handling mixed bit-width data types: For algorithms that use such data types, our contribution is an analysis that automatically selects the optimal SIMD width for the specified target instruction set (Section 3.4) in order to (a) pack native vectors into virtual vectors and (b) apply on-demand type promotion. The latter, is a technique to avoid falling back to scalar execution for operations on data types that are not covered by the instruction set.
4. Implementation and integration of this approach in our own DSL framework Hipacc [12], an open source DSL and source-to-source compiler for image processing. We perform an evaluation for multiple pre- and postprocessing image filters and observe a geometric mean speedup of up to 3.14 compared to non-vectorized executions.

Note that these contributions are bound to certain constraints and are clearly not applicable to every DSL. Nevertheless, our approach is not limited to the domain of image processing alone. We will explicitly identify the exact conditions under which our techniques and contributions can be employed in Section 3. Subsequently, in Section 4, we present results that we were able to obtain with Hipacc, before concluding this work in Section 5.

2. Whole-Function Vectorization

The purpose of this section is to provide a rough understanding of the principles of this approach. The constraints of whole-function vectorization [6] are that (a) the input code is given in SSA form and (b) the SPMD data-parallel programming model is used, such as kernels in OpenCL or CUDA.

As CPU cores offer large scalar execution units and vector processing units, it is possible to utilize both. Harnessing scalar execution units has the advantage of reducing the SIMD register pressure and avoiding the unnecessary execution of vector instructions. The result of preceding scalar computations can always be broadcast to multiple instances of a vector for later use. Therefore, a data flow analysis is performed that provides information about which program parts may be vectorized and which may remain scalar. This is accomplished by assigning specific markers to all variables,[1] which indicate the properties that are relevant for vectorization. The available markers are outlined in Table 1.

Table 1. Available markers in whole-function vectorization.

Marker	Property
`s`	same value
`sa`	same value, aligned
`c`	consecutive values
`ca`	consecutive values, aligned
$\top$	unknown values

All variables marked as *same value* (`s` or `sa`) are known to produce the same value for all program instances and may remain scalar. The other markers indicate vector variables whose values may either be completely *unknown* ($\top$), or vector elements that store *consecutive values* (`c` or `ca`). This means that every vector element has an increment of one with respect to its direct predecessor, e. g.,

[1] The term *variable* shall be understood as a storage for data. Thus, named constant values are represented as variables, too.

```
1 Func blur_3x3(Func input) {
2   Func blur_x, blur_y;
3   Var x, y, xi, yi;
4
5   // The algorithm - no storage or order
6   blur_x(x,y) = (input(x-1,y) + input(x,y) + input(x+1,y))/3;
7   blur_y(x,y) = (blur_x(x,y-1) + blur_x(x,y) + blur_x(x,y+1))/3;
8
9   // The schedule - defines order, locality; implies storage
10  blur_y.tile(x,y,xi,yi,256,32).vectorize(xi,8).parallel(y);
11  blur_x.compute_at(blur_y,x).vectorize(x,8);
12
13  return blur_y;
14 }
```

Listing 1. Defining a 3×3 box filter as two 3×1 passes in Halide.

the vector $(3, 4, 5, 6)$ would be marked as `c` and the vector $(3, 5, 4, 6)$ would be marked as $\top$. A prime example for *consecutive values* in SPMD is the current instance ID, represented by the parameter `tid`. Another important attribute is the *alignment flag*, which can be added to the markers `s` or `c`. The impetus behind this is the fact that SIMD units can use faster memory transfers when a vector is loaded from or stored to an *aligned memory* address, meaning that the memory address is a multiple of the vector register size. A variable will be flagged as `sa` if it is same for all instances and a multiple of the vector width. For consecutive vectors, the same consideration is made for the first element of the vector; thus, the vector $(4, 5, 6, 7)$ would be flagged as `ca` for vector widths 2 and 4.

For memory transfers, the required operands are an index value and a pointer, which is always flagged as `s` or `sa` in the context of the SPMD programming model. If the index is flagged as `s` or `sa`, only a scalar value is transferred and may be broadcast into a vector at a later point in time. If the index is flagged as `c`, `ca`, or $\top$, a vector is loaded or stored. For consecutive indexes, a fast *vector load* or *store* instruction can be issued, which transfers an entire vector at once. Depending on the presence of the alignment flag for both the pointer *and* the consecutive index vector, an even faster *aligned vector load* or *store* instruction can be used. If an index is flagged as $\top$, an expensive *vector gather load* from or *scatter write* to multiple memory locations may be issued.

3. Vectorization for Image Processing DSLs

We propose techniques for adapting and applying the aforementioned whole-function vectorization approach in combination with a DSL for image processing and a source-to-source compiler. In the next subsection, we first narrow down the scope of DSLs we have in mind, before summarizing our specific contributions in the following three subsections.

3.1 Scope of Image Processing DSLs

We target image processing DSLs that employ the SPMD programming model to operate on two-dimensional data structures using relative indexes. Typical examples of such DSLs are Hipacc [12] and Halide [17], but also DSLs for other domains, such as ExaSlang [19] for stencil computations, are suitable. All of these have in common to restrict memory accesses to relative coordinates. For instance, take the example in Listing 1. Here, a 3×3 box filter is defined in Halide as a series of two 3×1 passes. The first pass (line 6) operates on the left and right neighboring pixels, while the second (line 7) incorporates the top and lower neighboring pixels. Direct accesses to horizontal and vertical neighbors are described using the relative expressions `x`$\pm$`1` and `y`$\pm$`1`, respectively. Many image processing DSLs introduce similar restrictions in order to ease code analysis and perform domain-specific optimizations.

3.2 Simplifying the Vectorization Analysis

Aside from the constraint to require an SPMD programming model, inherited from whole-function vectorization, our domain-specific vectorization analysis derives from the integration of two additional constraints:

1. Input DSL code is not given in SSA form, which is fairly reasonable as DSL codes should be compact, high-level representations of algorithms.
2. Image objects in the DSL, the two-dimensional data structure to access input and output memory, must always be accessed using relative indexes. Therefore, an instance accessing an image via the indexes `[-1][0]` and `[1][0]` would read the neighboring pixels from the left and right, respectively. However, constant arrays, such as mask coefficients or lookup tables, are allowed to be addressed with absolute indexes.

Based on the above constraints, two major simplifications can be introduced:

First, as an input DSL is given in a high-level representation, it is very hard to draw conclusions regarding memory alignment during that phase. Consequently, for the majority of memory accesses it cannot be guaranteed that they will be aligned. As such, the vectorization markers `sa` and `ca` are seen as obsolete within our approach. This does not affect functional correctness, as unaligned vector loads and stores can also be used for aligned addresses.

The second simplification concerns memory access patterns. Due to the relative indexing, accessing an image with a constant index will result in adjacent memory locations for adjacent instances. Thus, variables referring to DSL image objects can be marked as *vectorized*. Furthermore, the consecutiveness marker `c` is also obsolete, because consecutive memory transfers are already indicated by references to images. Therefore, we can reduce the necessary vectorization markers to whether a variable will be a vector (`v`) or remains scalar (`s`).

3.2.1 Vectorization Rules

Due to the above simplifications, the size of the vectorization analysis lattice $\mathbb{L} = \{\texttt{s}, \texttt{sa}, \texttt{c}, \texttt{ca}, \top\}$ is reduced to only two $\{\texttt{s}, \texttt{v}\}$. Thereby, the original vectorization analysis is severely affected. Analogous to [6], we use the transfer functions $[\![\cdot]\!]^\sharp : (\mathit{Vars} \to \mathbb{L}) \to (\mathit{Vars} \to \mathbb{L})$ to present our vectorization rules. Here, also the function a is used for reflecting the analysis information and mapping variables to lattice elements, with the same notation $a \mid v \mapsto l$, which represents

$$\lambda w. \begin{cases} l & \text{if } v = w \\ a(w) & \text{else.} \end{cases}$$

Our vectorization rules are as follows:

- Literals and constants are the same for all instances, and therefore remain scalar. This also applies to constant arrays.

$$[\![v \leftarrow x \in \{\texttt{int}, \texttt{float}\}]\!]^\sharp a = a \mid v \mapsto \texttt{s}$$

- In SPMD, the i-th kernel argument is also the same for all instances and needs to remain scalar.

$$[\![v \leftarrow \texttt{arg}(i)]\!]^\sharp a = a \mid v \mapsto \texttt{s}$$

- The `tid` parameter (instance ID) results in varying values for each instance and needs to be treated as vector.

$$[\![v \leftarrow \texttt{tid}]\!]^\sharp a = a \mid v \mapsto \texttt{v}$$

- Arithmetic and comparison operators (expressions) produce vectors if any operand, which might be a sub-expression, is a vector. This does not include assignment operators, which are handled differently.

$$[\![v \leftarrow \texttt{op}(x, y)]\!]^\sharp a = a \mid v \mapsto \begin{cases} \texttt{s} & \text{if } a(x) = \texttt{s} \wedge a(y) = \texttt{s} \\ \texttt{v} & \text{else} \end{cases} \quad (1)$$

- Memory loads will always produce vectors if a DSL image is accessed. If a memory load from a constant array is specified, it depends on the index d whether a vector is produced or not.

$$[\![v \leftarrow \texttt{load}_p(d)]\!]^\sharp a = a \mid v \mapsto \begin{cases} \texttt{v} & \text{if } p \text{ is image} \\ a(d) & \text{else} \end{cases} \quad (2)$$

Note that the last rule in Eq. (2) is a crucial difference of our method compared to whole-function vectorization as introduced in [6, 7]. Due to relative indexing of DSL images, loads from images will always result in vectors. However, one distinction can be made: if the index d is scalar, a simple *vector load* can be issued, otherwise a *vector gather load* is required.

Another novelty of our approach is that the input DSL code is not given in SSA form. Thereby, we cannot rely on ϕ-functions and need to deal with scopes. The execution of entire scopes can depend on a condition, such as in if-statements. In that case, the value of a variable could be changed for only a few instances due to diverging control flow. Therefore, we need to introduce another rule for handling assignments:

Assignment operators produce vectors if a variable is assigned in a scope that depends on a set of conditions C and at least one condition c is a vector.[2] Otherwise, the information of x is propagated.

$$[\![v \leftarrow \texttt{assign}_C(x)]\!]^\sharp a = a \mid v \mapsto \begin{cases} \texttt{v} & \text{if } \exists_{c \in C} : a(c) = \texttt{v} \\ a(x) & \text{else} \end{cases} \tag{3}$$

Creating a set of conditions is subject to additional rules. Due to space limitations, we express these verbally:

- If control flow statements are nested, such as a hierarchy of if-statements, all their conditions are added to the set.
- If a loop body contains `break` or `continue` statements, all conditions those statements depend on are added to the set.

The last rule assures that the programmer has the option to selectively let a few instances leave a loop in certain circumstances, even if the primary loop condition is not a vector.

3.2.2 Vectorization Algorithm

The propagation through our vectorization rules is monotonic, meaning a scalar variable might become a vector, but never the other way around. Therefore, we can initialize all variables as scalar, except DSL image objects and the `tid` parameter. Furthermore, we can derive simple *dependencies* from our rules. Those dependencies are the operands of arithmetic and comparison operators (Eq. (1)), array indexes (Eq. (2)), the *right hand side* of assignments and a set of conditions (Eq. (3)). Thereby, we can create a list of dependencies for every variable. According to our monotonic rules, *if any* of a variable's dependencies is a vector, we need to produce a vector for the variable as well. This variable might be part of another variable's dependency list. Therefore, we have to iteratively revisit the dependency lists of all other variables until a steady-state is reached.

As our analysis lattice $\mathbb{L}$ has a cardinality of 2, we can omit the markers `s` and `v` completely and work with simple Boolean values instead. For analysis purposes, we introduce the two sets V and D:

$$\text{Variables: } V$$
$$\text{Dependencies: } D \subseteq V \times \mathcal{P}(V)$$

Initially, the set V contains all variables present in the input program. Each element in D contains a variable followed by a list of dependencies that are used as a *right hand side* for assignments to the corresponding variable. Additionally, all relevant *conditions* in the hierarchy of control flow statements are also part of the dependency list if the particular variable is assigned inside of a control flow statement. If a variable is assigned inside a loop body, the loop's condition and all the conditions of nested if-statements containing `break` or `continue` statements are also part of the variable's dependency list.

The actual vectorization algorithm is summarized in Algorithm 1. In the first step, all DSL image objects and the parameter `tid` are added to the set of vector variables X. This is based on domain knowledge and represented by calling `InitialVectorMarkers()`. Afterwards, the dependency set for every variable is analyzed (7–9). According to the previously mentioned rules, a variable will be marked as vector *if any* of its dependencies is a vector. In this case, the variable is added to the set of vector variables (10). If any element in this set has changed, this may have cross-effects for other variables. Therefore, the last step checks whether the vectorization markers of all variables have reached a *steady-state* (15) and may start another analysis iteration. As soon as the steady-state has been reached, the vectorization analysis can stop and the vectorization markers for all variables in the input program will be known.

Algorithm 1: Vectorization analysis algorithm.

```
1  function VectorizationAnalysis(V, D)
2      X ← InitialVectorMarkers()   // Vector variables
3      Y ← {}                       // Dependencies of vectors
4      repeat
5          X' ← X
6          Y' ← Y
7          forall (v_i, d_k) ∈ D do
8              forall v ∈ d_k do
9                  if v ∈ X' then
10                     X ← X ∪ {v_i}
11                     Y ← Y ∪ {(v_i, d_k)}
12                 end
13             end
14         end
15     until |X| = |X'|
16     return X
17 end
```

```
1 a = a + 1;
2 b = a * input[a][0];
3 c = (b + tid) % 2;
4 if (c == 0) {
5   d = a - 1;
6 }
7 output[0][0] = d;
```

Listing 2. Arbitrary code with the DSL image objects `input` and `output` (initial vectors marked in bold red).

For demonstration purposes, assume the code snippet in Listing 2 is to be vectorized. The initialization and iterative propagation of vector markers through the variable set X and dependency set Y are shown in Figure 2. As soon as the dependency set has been established, domain knowledge is used to initially mark the images `input`, `output`, and the parameter `tid` as vectors. After the first iteration, two dependencies have been added to Y, leading to the two additional variables `b` and `c` in set X. The second and third iteration will subsequently add the dependencies [`d`, (`a`, `c`)] and [`output`, (`d`)], resulting in a *steady-state* and thus terminating the algorithm. Note that `d` is a vector, although only depending on scalar variable `a`. However, the assignment in line 5 is controlled by a condition depending on the vector `c`. Therefore, `c` is part of `d`'s dependency list, which needs to become a vector as well. In this example, the only variable that remains scalar is `a`.

3.3 Source-To-Source Control Flow Rebuilding

The technique we are presenting to retain control flow during the translation process is able to generate source code for any kind of nested control flow. The only constraints for applying our technique are (a) to generate source code and (b) that the target language supports control flow statements with scopes.

3.3.1 Vector Mask Cascades

In general, a control flow statement is represented by a *vector mask* and a `select()` function. The elements of a vector mask represent

[2] A vector is only produced if the variable is visible outside of the control flow statement that is controlled by c, but this has been put aside for presentation.

INPUT

$V = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{input}, \texttt{output}, \texttt{tid}\}$

$D = \{[\texttt{a}, (\texttt{a})], [\texttt{b}, (\texttt{a}, \texttt{input})], [\texttt{c}, (\texttt{b}, \texttt{tid})], [\texttt{d}, (\texttt{a}, \texttt{c})], [\texttt{output}, (\texttt{d})]\}$

$X = \{\texttt{input}, \texttt{output}, \texttt{tid}\}$

$Y = \{\}$

ALGORITHM ITERATIONS

$X = \{\texttt{input}, \texttt{output}, \texttt{tid}, \texttt{b}, \texttt{c}\}$

$Y = \{[\texttt{b}, (\texttt{a}, \texttt{input})], [\texttt{c}, (\texttt{b}, \texttt{tid})]\}$

$X = \{\texttt{input}, \texttt{output}, \texttt{tid}, \texttt{b}, \texttt{c}, \texttt{d}\}$

$Y = \{[\texttt{b}, (\texttt{a}, \texttt{input})], [\texttt{c}, (\texttt{b}, \texttt{tid})], [\texttt{d}, (\texttt{a}, \texttt{c})]\}$

$X = \{\texttt{input}, \texttt{output}, \texttt{tid}, \texttt{b}, \texttt{c}, \texttt{d}\}$

$Y = \{[\texttt{b}, (\texttt{a}, \texttt{input})], [\texttt{c}, (\texttt{a}, \texttt{tid})], [\texttt{d}, (\texttt{a}, \texttt{c})], [\texttt{output}, (\texttt{d})]\}$

Figure 2. Propagation of the vectorization markers throughout the variable set X and dependency set Y during the vectorization analysis.

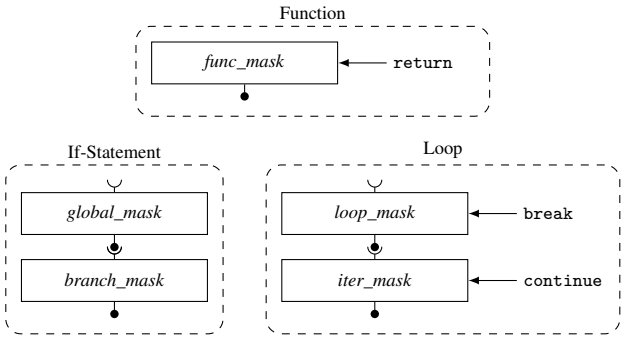


Figure 3. Vector masks for functions, if-statements, and loops with their affecting control flow statements and connectors for building up a vector mask hierarchy.

whether an instance is still active in the current execution or not. The `select()` function returns the elements from either one of two input vectors, controlled by the elements of a vector mask. Thereby, a control flow hierarchy can be matched with a hierarchy of such vector masks. Figure 3 shows the vector masks necessary for vectorized functions, if-statements, and loops. Depending on the given control flow of the program, these masks can be interconnected to build up a vector mask hierarchy. The vector mask of a nested control flow statement is always initialized with the *parent_mask*, which is the active vector mask of the parent control flow statement. For example, the *global_mask* of an if-statement that is nested within a loop is initialized with the loop's *iter_mask*. This is necessary to avoid the execution of inactive instances. Consequently, whenever a `return`, `break`, or `continue` statement appears inside a vectorized control flow, that statement is replaced by a *mask cascade update*. A *mask cascade update* removes all active instances from all vector masks in the hierarchy up to the mask that is affected by the statement. For loops, a `break` statement updates all masks up to the first *loop_mask*, whereas a `continue` statement only updates the masks up to the first *iter_mask*, leaving the *loop_mask* untouched.

3.3.2 Functions

The instances still active inside a function itself will only be tracked if a `return` statement inside a vectorized conditional branch appears. In this case, an additional *function mask* will be created for that program, which marks all instances that are still active in the function.

```
1 if (condition_1) {
2   // code for branch 1
3 } else if (condition_2) {
4   // code for branch 2
5 } else if (condition_3) {
6   // code for branch 3
7 } else {
8   // code for branch 4
9 }
```

(a) Original scalar code

```
1  if (condition_1) {
2    // code for branch 1
3  } else {
4    bool_vec global_mask = parent_mask;
5    bool_vec branch_mask;
6
7    // vectorized if-branch
8    branch_mask = select(global_mask, condition_2);
9    if (any(branch_mask)) {
10     // code for branch 2
11   }
12
13   // vectorized else-branch
14   branch_mask ^= global_mask;
15   if (any(branch_mask)) {
16     if (condition_3) {
17       // code for branch 3
18     } else {
19       // code for branch 4
20     }
21   }
22 }
```

(b) Vectorized pseudo code

Listing 3. Transformation of a multi-branch if-statement with `condition_2` being a vector.

3.3.3 If-Statements

In most DSLs, if-statements are not limited to only one `if`-branch and one `else`-branch. Multi-branch if-statements can be written by the programmer and must be handled. If a multi-branch if-statement contains one or more vectorized conditional branches, it is split into an equivalent hierarchy of simple (`if-else`) if-statements. For each of those if-statements it is separately evaluated whether it needs to be vectorized, depending on its condition being a vector or not.

In the example in Listing 3, the second condition of a multi-branch if-statement is a vector. This leads to the creation of an equivalent hierarchy of simple if-statements and forces the second if-statement to be vectorized. Note that any possible assignment in the first scalar `if`-branch (line 2) will also be subject to our vectorization rules, in particular Eq. (3), if the outer scope is vectorized. As the vectorized if-statement of `condition_2` is moved into the first scalar `else`-branch, it will not be evaluated if the first scalar `if`-branch is taken. Otherwise, the two masks *global mask* and *branch mask* are created for the vectorized if-statement and initialized with the *parent mask* in lines 4–5. The *parent mask* is either the most recent mask of a vectorized outer scope or an entirely new mask with all instances set to active. Depending on `condition_2`, all instances that are still globally active are selected and stored in the branch mask (line 8) immediately before it is evaluated as to whether the `if`-branch should be taken (line 9). Finally, all globally active instances in the *branch mask* are toggled (line 14) and the vectorized `else`-branch is processed if any active instance remains (line 15). Here, the *branch mask* represents the *parent mask* for any possibly nested control flow statement in the branches 2, 3, and 4.

3.3.4 Loops

The vectorization of loops can be heavily simplified, as each loop only has a single point of entry, namely its header. The only way to exit multiple loops at once is to exit the entire function with a `return` statement. Loops are only marked as vectorized if they have a vector condition or nested vectorized if-statements containing the loop

```
1 while (condition_1) {
2   // code 1
3   if (condition_2) continue;
4   // code 2
5   if (condition_3) break;
6   // code 3
7 }
```

(a) Original scalar code

```
1 bool_vec loop_mask = parent_mask;
2 while (true) {
3   // loop header
4   loop_mask = select(loop_mask, condition_1);
5   if (none(loop_mask)) break;
6
7   // loop body
8   bool_vec iter_mask = loop_mask;
9   // code 1
10  iter_mask ^= select(iter_mask, condition_2);
11  // code 2
12  loop_mask ^= select(iter_mask, condition_3);
13  iter_mask &= loop_mask;
14  // code 3
15 }
```

(b) Vectorized pseudo code

Listing 4. Transformation of a while-loop with all conditions being vectors.

```
1 float a = i;
2 int16 b = j;
3
4 float r = a / (b * 2);
```

(a) Original scalar code

```
1 float_vec a = broadcast(i);
2 int16_vec b = broadcast(j);
3
4 int16_vec tmp1 = b * broadcast(2);
5 float_vec tmp2 = conv_int16_to_float(tmp1);
6 float_vec r = a / tmp2;
```

(b) Splitting expressions

```
1 float_vec a[2];
2 a[0] = a[1] = broadcast(i);
3 int16_vec b = broadcast(j);
4
5 int16_vec tmp1 = b * broadcast(2);
6 float_vec tmp2[2], r[2];
7 tmp2[0] = conv_int16_to_float(tmp1, 0 /* low */);
8 tmp2[1] = conv_int16_to_float(tmp1, 1 /* high */);
9 r[0] = a[0] / tmp2[0];
10 r[1] = a[1] / tmp2[1];
```

(c) Insertion of virtual vectors

Listing 5. Creation of mono-type expressions by expression tree splitting and translation to virtual vectors with `i` and `j` being scalar.

control statements `break` or `continue`. The issue yet to be addressed is how to handle these loop control statements. If they are nested inside purely scalar if-statements, they can be left untouched because they have the same effect for all active instances in the loop. However, this proves different when only a few instances leave the loop early in a vectorized if-statement. Here, *mask cascade updates* as described above in Section 3.3.1 need to be inserted.

The vectorization of loops is described with the aid of a top-controlled while-loop, as shown in Listing 4. The loop is never left through its primary header, with the decision to exit the loop being moved to the pseudo-header (lines 4–5). The loop itself is not allowed to terminate completely as long as any instance is still active. The information about which instances are still active inside the loop is stored in the *loop mask*. In order to link the vectorized loop into the hierarchy of nested vectorized control flow statements, the *loop mask* is initialized with its *parent mask* (line 1). The *loop mask* is updated at the beginning of each loop iteration (line 4). The `none()` instruction (line 5) returns a scalar Boolean value indicating whether all instances in the *loop mask* are marked as inactive. In this case, the `break` statement terminates the loop entirely. The actual loop body starts with the initialization of the *iteration mask* (line 8), which stores all instances that are active in the current iteration. As the `continue` statement might cause some instances to leave the current iteration depending on `condition_2`, the *iteration mask* has to be updated accordingly (line 10). Similarly, the `break` statement affects instances in the *iteration* and the *loop mask*, causing a *mask cascade update* (lines 12–13).[3] Note that in contrast to whole-function vectorization, multiple vectorized `continue` statements can be handled by only one *iteration mask*, thus minimizing the overhead induced. The decrease in register pressure is even more notable for loops than for if-statements. This becomes even more dominant when more vectorized loop exits are present in the loop body.

3.4 Handling Mixed Bit-Width Data Types

Crucial to vectorization is the treatment of expressions with varying element types, which was not covered in [6]. The final stage of our approach is a instruction generation technique that is designed to support mixed bit-width data types. The only constraint required to apply our technique is that the target language supports the explicit use of specific vector instructions, in our case intrinsics. To achieve the support for mixed bit-width data types, we first preprocess all vectorized expressions and split them to operate on only a single data type. Part of that step is the insertion of so-called *broadcasts* for all scalar types that are direct operands of a vectorized expression. Afterwards, *virtual vectors* are inserted in order to map the vectorized expressions to the bit-width of a specific instruction set. The entire process is exemplarily depicted in Listing 5, with the preprocessing step in Listing 5b and the subsequent insertion of virtual vectors in Listing 5c.

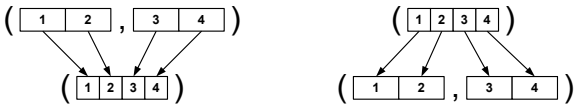
3.4.1 Expression Tree Splitting

Since vector instruction sets usually only support operations where all operands have the same element type, *vector conversions* need to be inserted wherever a vectorized expression must deal with varying element types. All vectorized expression trees containing any vector conversion are split into separate sub-expression trees, which operate with only one vector element type, as shown in Listing 5b. Temporary variables hold the results of the sub-expression trees as well as the conversions for promoting previously created temporary variables. Type promotion used for vectors in our approach aims to maintain the smallest possible element size for each operation, e. g., the addition of a 8-bit and 16-bit vector would merely lead to being promoted to a 16-bit vector. We want to emphasize that we do not rely on input language's promotion rules and apply conversions exactly as specified by the developer.

3.4.2 Virtual Vectors

All vectorized expressions are mapped to a specific instruction set (e. g., SSE4.2, AVX, or AVX2), selected by the user. Since this vectorization approach is based on the SPMD model, it is required that the vector width, i. e., the number of instances processed in parallel, is constant throughout the entire vectorized program. This requirement lies in conflict with the internal layout of the used SIMD units, as the number of instances that fit into a vector usually depends on the bit width of its element type and thus may differ for each expression. In order to meet the requirement of the SPMD model, a constant *virtual vector width* is automatically chosen for the entire program, which is sufficiently large enough such that one SIMD register is exactly filled with the smallest vector element type referenced in the input program, ensuring full occupancy of the SIMD units. Because the size of element types varies throughout the program, our vectorization approach works with *virtual vectors*, which are translated into arrays of SIMD vectors.

[3] The example has been slightly simplified for presentation, as the two nested if-statements would actually remain preserved and cause the creation of additional *branch masks*, used to perform the *mask cascade updates*.



(a) Downward conversion (b) Upward conversion

Figure 4. Conversion expressions for virtual vectors.

The usage of vector arrays requires that certain expression trees in the input program are replicated for every vector inside of the array described by a virtual vector. Since the preprocessing step ensures that all expression trees, except conversion expressions, use only one vector element type, this replication is straightforward. The pseudo code in Listing 5b illustrates using 16-bit integers as the smallest vector element type. Promoting a vector containing n 16-bit integers to 32-bit floating point numbers results in two vectors, each containing $n/2$ elements, with n depending on the vector width of the specific instruction set. Therefore, the abstract expression (line 6) must be replicated twice, as shown in Listing 5c (lines 9–10).

Conversion expressions are a special case, as they have to deal with differing vector element types in their input and output. Each of these expressions are replicated as often as their *output* element type requires, which is shown in Listing 5c (lines 7–8). Depending on the relationship between the input and the output element size, each of the generated instructions either packs multiple input vectors into one output vector, or extracts merely a portion of one input vector into a full output vector. This conversion is illustrated in Figure 4. The downward conversion shown in Figure 4a can be performed with a single instruction, which packs two vectors into one, whereas the upward conversion, depicted in Figure 4b, requires two instructions to be created, which each processing one half of the input vector.

3.4.3 On-Demand Type Promotion

The last issue requiring discussion is the deficiency of the vector instruction sets. Sometimes, a specific instruction is not supported for a certain vector element type, but only for a larger element type. For instance, the *SSE2* instruction set supports element-wise multiplications for 16-bit integer vectors, but not for 8-bit integer vectors. This caused certain ramifications for vectorization, and several strategies have already been proposed for this topic, one of which is *Hybrid Type Legalization* [2]. This work offers a heuristic approach that selectively promotes a few vector variables to a larger element type in order to avoid the most expensive deficiencies in the instruction set used. Although this method may lead to performance increases in some cases, it can raise severe performance issues due to cross-effects. The promotion of one vector variable to a larger type could force an equivalent promotion of its dependent variables and thus induce the demand for other operations that might not be covered by the instruction set. Furthermore, larger element sizes reduce the number of instances that can be processed in parallel. Thus, this coarse-grained type promotion can have a domino effect, while our approach addresses this issue with a more fine-grained strategy, which we call *on-demand type promotion*. Whenever a deficiency that can be ameliorated through vector element type promotion is encountered in the instruction set, it is handled by type promotion at the instruction level. To continue with the previous example, as soon as two 8-bit integer vectors are multiplied using the SSE2 instruction set, both operands are converted to only 16-bit vectors for this multiplication and the result is converted back to an 8-bit integer vector immediately thereafter.

4. Evaluation and Results

In the following, we present the execution times we were able to obtain with our technique for a variety of algorithms and compare them to the execution times of other well-established compilers. For evaluation purposes, we implemented and integrated our vectorization approach into the DSL compiler of the Heterogeneous Image Processing Acceleration (Hipacc) framework [12]. Therefore in this section, we start with a brief introduction of Hipacc before presenting our evaluation environment and the results, achieved by our vectorization approach.

```
1  class Laplace : public Kernel<uchar> {
2   private:
3    Accessor<uchar> &input;
4    Mask<int> &mask;
5
6   public:
7    Laplace(IterationSpace<uchar> &output,
8            Accessor<uchar> &input, Mask<int> &mask)
9        : Kernel(output), input(input), mask(mask) {
10     add_accessor(&input);
11   }
12
13   void kernel() {
14     int sum = 0;
15     for (int yf = -1; yf <= 1; yf++)
16       for (int xf = -1; xf <= 1; xf++)
17         sum += input(xf, yf) * mask(xf, yf);
18     output() = (uchar)min(max(sum, 0), 255);
19   }
20 };
```

Listing 6. Defining the Laplacian filter in Hipacc.

4.1 The Hipacc Framework

The Hipacc framework was originally developed for GPUs. The framework is comprised of a DSL for 2D image processing embedded into C++, and a source-to-source compiler for generating target-specific source code [12]. The compiler is based on the Clang/LLVM infrastructure and can generate code for multiple target languages, such as Compute Unified Device Architecture (CUDA), Open Computing Language (OpenCL), Renderscript, a shader-like language developed for Android, and plain C++.

4.1.1 Domain-Specific Language

DSL source code is represented by C++ template classes. Using these classes is, in essence, similar to using the Application Programming Interface (API) of an image processing library. However, the difference is that API calls are replaced by appropriate constructs of the desired target language during source-to-source compilation. Additionally, the description of image filter kernels is severely transformed to run efficiently on the specified target architecture.

An example of defining a Laplacian filter for edge detection is given in Listing 6. First of all, it is necessary to derive a new class from Hipacc's `Kernel` class. Within this class, the `kernel()` method must be overridden in order to define custom C++ code for the kernel. The actual filter description can be found in lines 14–18. Here, a reduction over a predefined 3×3 local window is issued. For each position in this local window, the corresponding `mask` coefficient is multiplied with the corresponding pixel of the `input` image. Through using this image object, possible out-of-bounds accesses are automatically caught by the DSL compiler and replaced with user-defined border handling, introducing diverging control flow. The accumulated scalar value is stored in `sum`, before being clamped to 0–255 and written to the `output` image.

4.1.2 Vectorization Back End

We integrated our vectorization approach into the C++ back end of the Hipacc compiler. The back end supports the generation of vector instrinsics for the Intel instruction sets up to SSE4.2 and AVX2. For our benchmarks below, the generated source codes were compiled to executables using the Clang/LLVM compiler.

4.2 Evaluation Environment

All benchmarks were performed on an Ubuntu 14.04.5 LTS operating system. More detailed information about the hardware and compilers

we used for benchmarking are provided in this section. Furthermore, we also briefly introduce the algorithms we used to compare the resulting execution times of our vectorization approach.

4.2.1 Hardware

For our evaluation environment, we chose an Intel Xeon E5-1620 v3 CPU based on the Haswell architecture, clocked at 3.50 GHz. It is capable of handling the instruction sets up to SSE4.2 and AVX2. With the intent to measure the improvement through vectorization alone, we ran our benchmarks on only a single core. When running on multiple cores, performance is easily limited by the available memory bandwidth, which would diminish the achievable speedup through vectorization and make the results less comparable. As we employed Linux, we were faced with jitter and other overhead related to the operating system. To avoid exaggerated outliers from the start, we disabled hardware features such as HyperThreading, TurboBoost, and SpeedStep completely.

4.2.2 Compilers

To compare our vectorization method that we have integrated into the Hipacc framework [12], we chose four compilers with auto-vectorization support. For all compilers, we enabled auto-vectorization and the aggressive optimization scheme "-O3", which enables neglecting possible aliasing of input and output arrays. In the following, a brief introduction to the chosen compilers is provided.

Hipacc We call Hipacc to generate C++ code for CPUs with enabled vectorization for the specific target instruction set. The generated C++ sources contain explicit vector intrinsic calls, which leaves not potential for further optimization by the subsequent Clang compiler.

ISPC The Intel SPMD Program Compiler (ISPC) [15] (version 1.8.2) uses a shader-like language, very similar to C++, with specific keyword extensions to guide the vectorization process. The C-based language applies the Single Program Multiple Data (SPMD) programming model to compile code for the Intel instruction sets SSE2, SSE4, AVX, AVX2, and Xeon Phi.

ICC The Intel C++ Compiler (ICC) (version 15.0.1) is a general-purpose compiler from Intel with front ends for C and C++. It supports the Intel instruction sets up to SSE4 and AVX2. To prevent aliasing, we manually enabled the "-fno-alias" switch for ICC. Therefore, the auto-vectorization analysis should be more likely to succeed.

Clang/LLVM Clang (version 3.6.2) is a compiler front end for C, and C++. The compiler back end Low Level Virtual Machine (LLVM) supports auto-vectorization for Intel instruction sets SSE, AVX, and AVX2 by employing a loop vectorizer as well as an SLP vectorizer.

GCC The C++ compiler of the GNU Compiler Collection (GCC) (version 4.8.4) supports auto-vectorization for all well-established Intel instruction sets.

Baseline For a baseline comparison, we used the same GCC compiler with aggressive optimization scheme as well, but with auto-vectorization explicitly turned off. Thus, the bars in the speedup graphs in Figure 5 are normalized to the execution time of baseline.

We did not consider Halide [17] for comparison as their vectorization is not able to handle control flow. Although, previous work [7] implemented an OpenCL driver that is capable of vectorizing control flow, they merely implemented a "sufficiently complete fraction" of OpenCL to run their own benchmarks. Therefore, we decided not to use it for our set of algorithms.

4.2.3 Algorithms

Since we utilized a DSL for image processing in our experiments, the algorithms we chose for benchmarking are primarily based on image filters. For Hipacc and ISPC, both resemble a shader-like language, where mainly the innermost kernel computation needs to be implemented. However, for ICC, Clang, GCC, and baseline, we ported those benchmarks to C++ by adding outer loops to wrap inner kernel computations. Loops do not contain any loop-carried dependencies, and therefore should be embarrassingly easy to parallelize. A brief explanation regarding the specific implementation of all benchmarks is given in the following.

ISPC Suite The benchmarks *simple*, *Mandelbrot*, and *stencil* have been taken from ISPC's benchmark suite.[4] All have in common to mainly operate on single precision floating point data, whereas the former two heavily employ control flow in terms of if-statements and a loop with an early exit. One of the reasons for employing benchmarks from ISPC's suite is to avoid the impression that codes for comparison have not been well optimized.

Preprocessing The next algorithms *Laplace*, *Gaussian blur*, *Harris corner* [4], and *optical flow* [23] are typical filters for preprocessing and feature extraction. They all employ a local operator accessing neighboring pixels. This would inevitably lead to out-of-bounds accesses at the image border, and therefore the border handling mode *clamping* has been implemented. Clamping introduces additional control flow based on image coordinates, which causes a conditional adjustment of array indices for only some of the vector instances that leads to more gather loads. Input and output image data types are 8-bit integers. The Hipacc implementation of *Laplace* is shown in Listing 6.

Postprocessing Last, we benchmark the *bokeh effect*, *night filter*, and *chromatic aberration* representing widely-adopted postprocessing filters. In particular the *bokeh effect* and *chromatic aberration* are often used in computer graphics to simulate depth of field and the color shifts of real-world camera lenses, respectively. The *night filter* first performs a bilateral filtering through iteratively applying the *à trous* (with holes) algorithm [20] with different sizes (3×3, 5×5, 9×9, 17×17), before performing the actual tone mapping [5]. All filters use *clamping* and read or write RGBA image data packed into 32-bit integers, whereas most intermediate operations are performed on single precision floating point types. Images produced with our benchmarks *bokeh effect* and *night filter* are shown in Figure 1.

As input data for all pre- and postprocessing filters, we were using RGB images of roughly one megapixel size or slightly larger.

4.3 Results

We used the compilers mentioned in Section 4.2.2 to compile the benchmarks introduced in Section 4.2.3. We chose the target instruction sets SSE4.2, AVX, and AVX2. Each entry in Table 2 presents the median execution time of 100 benchmark runs. Figure 5 shows speedup graphs relative to the baseline execution. For every test case, the execution of baseline as well as the corresponding compiler was performed 100 times. Putting these numbers for both into relation results in 10000 speedup values per test case. Even though we turned off many hardware features, a specific amount of uncertainties remain. We visualize those uncertainties in the box plot overlay shown in the graph. The whiskers cover the entire range of results including the minimum and maximum speedup values. The box itself contains only the mean 50 % of all results and the line within the box represents the median. In many cases, the box is barely visible, due to very minor jitter, which results in many speedup values lying very close together. For a quick overview, Table 3 presents the geometric mean of all speedups per instruction set across all benchmarks. Sources of all our benchmarks as well as the input and output images are provided on the first author's personal homepage.

The first benchmark (*simple*) is a prime example for simple control flow. Here, the ICC is clearly a small step ahead of all other compilers

[4] https://github.com/ispc/ispc/tree/v1.8.2/examples

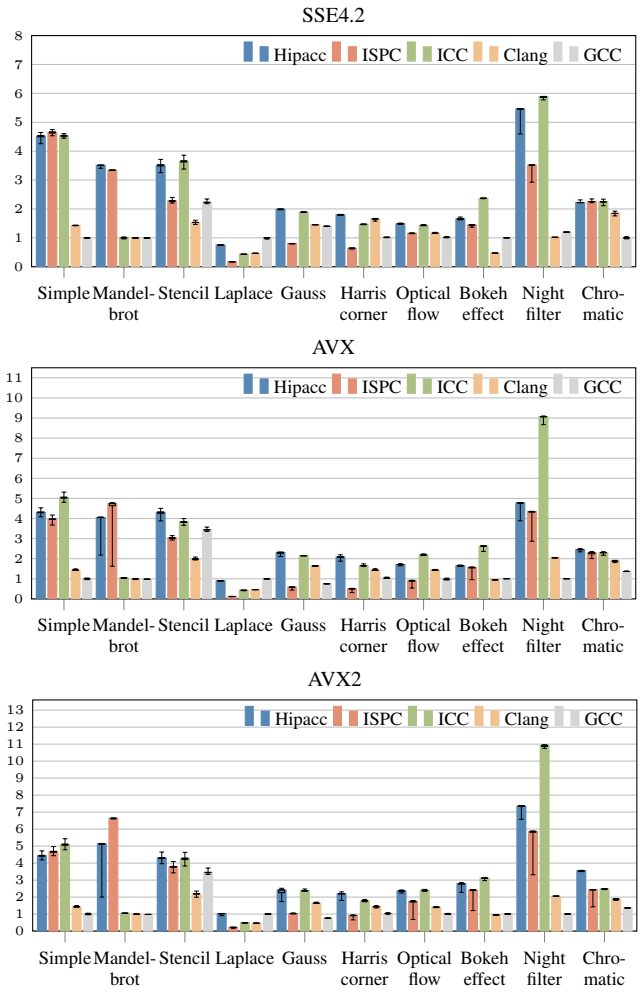


Figure 5. Speedups on a single core, w.r.t. non-vectorized baseline.

for the AVX and AVX2 instruction sets. Hipacc and ISPC achieve great results. Although Hipacc attained a little better speedup for AVX, ISPC remained slightly superior in the other cases. This benchmark contains only a simple (`if-else`) if-statement. Due to the SSA form of ISPC-generated low-level code, only a single blending operation is performed after executing both branches. Hipacc however, generates source code that contains a blending operation at the end of the `if`-branch and the `else`-branch, and therefore performs two blending operations. This additional overhead justifies the small but noticeable gap.

The ISPC compiler, in general, performs very well for the algorithms chosen from its benchmark suite (*simple*, *Mandelbrot*, *stencil*). In particular, *Mandelbrot* was carefully tuned by declaring specific variables as *uniform*, wrapping certain operations in an *unmasked* scope to omit blending, and deciding whether to use *coherent* control flow statements, which are used to retain a certain degree of control flow. Nevertheless, for the SSE4.2 instruction set, Hipacc achieves a slightly better speedup for *Mandelbrot* and even an improvement of 54 % for *stencil*. For the *Mandelbrot* benchmark, the vectorization analysis of all general-purpose compilers appeared to fail. Hipacc and ISPC attained an excellent result, as it is difficult to achieve results close to the theoretical speedup. This is illustrated by the fact that a loop with an early exit is processed, but inactive vector instances must remain within the loop iteration until all instances have met the exit condition.

Table 2. Median execution times in *ms* on a single core.

	Hipacc	ISPC	ICC	Clang	GCC	Baseline
			SSE4.2			
Simple	6.63	**6.44**	6.61	20.98	29.95	29.95
Mandelbrot	**27.75**	29.16	97.46	97.66	97.69	97.36
Stencil	7.97	12.31	**7.69**	18.28	12.59	28.00
Laplace	5.23	22.89	9.11	8.49	**3.93**	3.93
Gaussian	**25.39**	63.08	26.74	34.72	36.11	50.52
Harris corner	**37.60**	105.44	46.00	40.83	66.48	67.55
Optical flow	**283.29**	364.98	294.21	362.72	415.17	421.38
Bokeh effect	5079.96	5816.73	**3552.77**	17898.57	8396.78	8415.03
Night filter	247.25	383.91	**229.48**	1325.28	1128.26	1350.64
Chromatic	23.53	**23.18**	23.27	28.33	52.29	52.29
			AVX			
	Hipacc	ISPC	ICC	Clang	GCC	Baseline
Simple	6.95	7.55	**5.94**	20.88	29.95	29.95
Mandelbrot	24.05	**20.54**	93.63	98.17	99.42	97.32
Stencil	**6.50**	9.24	7.31	14.10	8.14	27.98
Laplace	4.35	32.32	9.16	8.49	**3.93**	3.93
Gaussian	**21.83**	86.57	23.50	30.80	67.00	50.52
Harris corner	**32.34**	134.57	40.72	47.13	65.33	67.52
Optical flow	246.32	468.37	**192.52**	293.40	432.91	421.20
Bokeh effect	5099.39	5383.07	**3204.57**	8855.21	8353.64	8415.11
Night filter	282.82	310.77	**148.99**	658.84	1341.78	1350.64
Chromatic	**21.30**	22.86	22.73	27.79	38.32	52.29
			AVX2			
	Hipacc	ISPC	ICC	Clang	GCC	Baseline
Simple	6.76	6.42	**5.89**	20.99	29.95	29.95
Mandelbrot	18.96	**14.68**	92.64	97.78	99.41	97.33
Stencil	**6.52**	7.41	6.56	12.79	8.13	27.98
Laplace	**3.84**	17.90	8.03	8.49	3.93	3.93
Gaussian	**20.91**	49.08	21.19	30.80	67.03	50.52
Harris corner	**30.64**	74.08	37.95	47.22	65.37	67.54
Optical flow	178.20	240.40	**175.67**	300.08	416.05	421.10
Bokeh effect	3019.57	3503.62	**2697.78**	8849.31	8327.95	8415.06
Night filter	183.50	230.48	**123.64**	657.52	1342.02	1350.63
Chromatic	**14.74**	21.49	21.09	27.79	38.28	52.29

Table 3. Geometric mean of speedups across all benchmarks.

	Hipacc	ISPC	ICC	Clang	GCC
SSE4.2	2.32	1.45	1.97	1.10	1.14
AVX	2.53	1.40	2.27	1.32	1.13
AVX2	3.14	2.07	2.50	1.33	1.14

The observed speedups for all preprocessing filters (*Laplace*, *Gaussian blur*, *Harris corner*, *optical flow*) are considerably less high. In particular, for the *Laplacian* operator on the AVX instruction set, the Hipacc speedup is even below the theoretical speedup compared to the baseline. AVX lacks instructions for handling vectors with integer elements. As our *Laplacian* benchmark mainly performs integer operations, the code generator must fall back to SSE4.2 instructions by extracting the upper and lower half of vectors, which causes transfers from AVX registers to SSE registers. This transfer leads to a distinct overhead introduced by state transitions, which each consume approximately 50 to 80 clock cycles. Therefore, this overhead severely diminishes the speedup gained from vectorization. In general, all other compilers are considerably affected by this issue, which might even lead to lower performance than the baseline, despite vectorization.

Furthermore, these AVX state transitions are an overall issue that also affects the remaining filters for postprocessing (*bokeh effect*, *night filter*, *chromatic aberration*), as their image data type is 8-bit integer packed into 32-bit integer. In particular, the ICC addresses this issue rather well, compared to all other compilers, but on the downside, demands compile times of several hours instead of minutes for *bokeh*. Nevertheless, the speedup attained remained far below the expected theoretical value for *bokeh* and *chromatic aberration*. Contrarily, the *night filter* achieves very good results, which we ascribe to approximating an exponential function with single precision floating point types that hold a high potential for gaining speed from vectorization.

4.4 Discussion

In this work, the Hipacc framework was used for evaluation purposes. The constraints in Sections 3.2 and 3.4 would supposably also apply to Halide [17] and other already existing DSLs. Therefore, those DSLs would also benefit from our proposed vectorization methodology. Our approach could be applied to different domains as well. In fact, DSLs for image processing are particularly suitable but so are DSLs for other domains, such as stencil computations in ExaSlang [19] by Schmitt et al., where data-locality is exploited on numerical grids instead of images.

Furthermore, we want to emphasize that we do not claim to be better than ISPC or ICC. Rather, we would like to demonstrate that we are able to achieve comparable results without rewriting already existing DSL codes. In addition, we want to highlight that DSL codes are usually more compact and easier to maintain for an algorithm designer who is not familiar with low-level architectural details and vectorization techniques. Despite the results presented in this work, Hipacc could apply further domain-specific optimizations to entirely circumvent gather loads for the innermost image region, as border handling is already covered by Hipacc's domain analysis. Thereby, we were able to observe even higher speedups of up to 9 for SSE and 13 for AVX, which we deliberately left aside in this work. However, those results justify to perform the vectorization, in combination with domain-specific optimizations, directly in the DSL compiler, instead of relying on the vectorization capabilities of a low-level compiler.

5. Conclusion

We presented a vectorization approach for image processing DSLs that could, within certain constraints, also be applied to other domains. Our novelty, over existing methods, is the capability to handle input code that is not given in SSA form and a simplified vectorization analysis tailored toward image processing DSLs. We furthermore presented translation techniques for generating vectorized source code while retaining control flow, which increases the readability of the generated source code, reduces the pressure on vector registers and maintains the advantage of skipping the execution of entire branches. Also, for each separate kernel, we can automatically derive the number of elements to processes in parallel, based on the smallest vector element type used throughout the entire kernel. Other element types, within the same kernel, are efficiently handled by automatic up- or downward conversion and packing into virtual vectors.

Evaluation results demonstrated the capabilities of our approach. With our implementation, integrated into the Hipacc framework, we were able to observe a geometric mean speedup of up to 3.14, compared to non-vectorized executions. Yet aside from the general benefits of DSLs, such as productivity and performance portability, combining vectorization with domain-specific optimizations still leaves enough potential to achieve even higher speedups.

Acknowledgments

This work is supported by the German Research Foundation (DFG), as part of the Research Training Group 1773 "Heterogeneous Image Systems", and as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89).

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th Symposium on Principles of Programming Languages (POPL)*, pages 177–189, Austin, Texas, 1983.

[2] Y. B. Asher and N. Rotem. Hybrid type legalization for a sparse SIMD instruction set. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(3):Article No. 11, September 2013.

[3] S. S. Baghsorkhi, N. Vasudevan, and Y. Wu. FlexVec: Auto-vectorization for irregular loops. In *Proceedings of the 37th International Conference on Programming Language Design and Implementation (PLDI)*, pages 697–710, Santa Barbara, CA, USA, 2016.

[4] C. Harris and M. Stephens. A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*, pages 147–151, 1988.

[5] H. W. Jensen, S. Premoze, P. Shirley, W. B. Thompson, J. A. Ferwerda, and M. M. Stark. Night rendering. Technical Report UUCS-00-016, Computer Science Department, University of Utah, Aug. 2000.

[6] R. Karrenberg and S. Hack. Whole-function vectorization. In *Proceedings of the 9th International Symposium on Code Generation and Optimization (CGO)*, pages 141–150, Chamonix, France, April 2011.

[7] R. Karrenberg and S. Hack. Improving performance of OpenCL on CPUs. In *Proceedings of the 21st International Conference on Compiler Construction (CC)*, pages 1–20, Tallinn, Estonia, 2012.

[8] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *Journal of Parallel Programming*, 28(4):347–361, August 2000.

[9] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 145–156, 2000.

[10] R. Leißa, I. Haffner, and S. Hack. Sierra: A SIMD extension for C++. In *Proceedings of the Workshop on Programming Models for SIMD/Vector Processing*, pages 17–24, Orlando, Florida, USA, February 2014.

[11] D. Levine, D. Callahan, and J. Dongarra. A comparative study of automatic vectorizing compilers. *Journal of Parallel Computing*, 17(10): 1223–1244, December 1991.

[12] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert. HIPAcc: A domain-specific language and compiler for image processing. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):210–224, January 2016.

[13] D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 281–294, New York, USA, March 2006.

[14] D. Nuzman and A. Zaks. Outer-loop vectorization - revisited for short SIMD architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 2–11, Toronto, Canada, October 2008.

[15] M. Pharr and W. R. Mark. ISPC: A SPMD compiler for high-performance CPU programming. In *Proceedings of the International Conference on Innovative Parallel Computing (InPar)*, pages 1–13, San Jose, USA, May 2012.

[16] M. Püschel, F. Franchetti, and Y. Voronenko. Spiral. In D. Padua, editor, *Encyclopedia of Parallel Computing*. 2011.

[17] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th Conference on Programming Language Design and Implementation (PLDI)*, pages 519–530, Seattle, USA, June 2013.

[18] H. Saito, S. Preis, N. Panchenko, and X. Tian. *Reducing the Functionality Gap Between Auto-Vectorization and Explicit Vectorization*, pages 173–186. Nara, Japan, Oct. 2016.

[19] C. Schmitt, S. Kuckuk, F. Hannig, H. Köstler, and J. Teich. ExaSlang: A domain-specific language for highly scalable multigrid solvers. In *Proceedings of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 42–51, New Orleans, LA, USA, 2014.

[20] M. J. Shensa. The discrete wavelet transform: Wedding the À Trous and Mallat algorithms. *IEEE Transactions on Signal Processing*, 40(10): 2464–2482, 1992.

[21] J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 165–175, San Jose, USA, March 2005.

[22] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *Journal of Parallel Programming*, 28(4):363–400, August 2000.

[23] F. Stein. Efficient computation of optical flow using the Census Transform. In C. Rasmussen, H. Bülthoff, B. Schölkopf, and M. Giese, editors, *Pattern Recognition*, volume 3175 of *Lecture Notes in Computer Science*, pages 79–86. 2004.

[24] Y. Sui, X. Fan, H. Zhou, and J. Xue. Loop-oriented array- and field-sensitive pointer analysis for automatic SIMD vectorization. In *Proceedings of the 17th International Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES)*, pages 41–51, Santa Barbara, CA, USA, 2016.