Oliver Reiche, **Christof Kobylko, Frank Hannig, and Jürgen Teich**
Hardware/Software Co-Design,
Friedrich-Alexander University Erlangen-Nürnberg

Auto-vectorization for Image Processing DSLs

Domain-Specific Auto-Vectorization

Inspired by Whole-Function Vectorization, contributions are:

- simplify the **vectorization analysis** for DSLs
- retain control flow for **source-to-source vectorization**
- automatically select **optimal SIMD width** per kernel

Whole-Function Vectorization

Vectorization Markers

Table 1: Available markers in Whole-Function Vectorization.

Marker	Property	Example
s	same value	3
sa	same value, aligned	4
c	consecutive values	(3, 4, 5, 6)
ca	consecutive values, aligned	(4, 5, 6, 7)
⊤	unknown values	(2, 1, 3, 4)

Lattice of Whole-Function Vectorization: $\mathbb{L}_{\text{WFV}} = \{\mathtt{s}, \mathtt{sa}, \mathtt{c}, \mathtt{ca}, \top\}$

Example Rule

Add operator:

$$[\![v \leftarrow \oplus(x, y)]\!]^\sharp a = a \mid v \mapsto$$

$a(x), a(y)$	sa	s	ca	c	⊤
sa	sa	s	ca	c	⊤
s	s	s	c	c	⊤
ca	ca	c	⊤	⊤	⊤
c	c	c	⊤	⊤	⊤
⊤	⊤	⊤	⊤	⊤	⊤

Vectorization Analysis

DSL restriction: only relative indexing
⇒ relative indexing implies consecutiveness!

Simplification

Due to **relative indexing** and analyzing **source code** we can drop

- consecutive marker (c)
- alignment markers (sa, ca)

Resulting reduced lattice: $\mathbb{L}_{\text{DSL}} = \{\mathtt{s}, \mathtt{v}\}$

Rules

Arithmetic and comparison operators:

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

Memory loads:

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

Due to analyzing source code with scopes, assignment operators:

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

Algorithm

As all rules are monotonic, only s might become v:

1. initialize all variables as s
2. mark images and tid as v
3. visit all variables:
   mark as v *if any* of its dependencies is v
4. if markers changed, go to 3

```
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 1: Example DSL code

Example

Vars a, b, c, d, input, output, tid
Deps [a, (a)], [b, (a, input)], [c, (b, tid)], [d, (a, c)], [output, (d)]
Vectors input, output, tid, b, c, d

Source-to-Source Vectorization

Flattened pure data flow might execute unnecessary branches.

```
1 bool mask = a > b;
2 float r1 = a + 1;
3 float r2 = a - 1;
4 r = select(mask, r1, r2);
```

Listing 2: Flattened pure data flow

Remedy

- retain control flow
- maintain a mask hierarchy for control flow statements

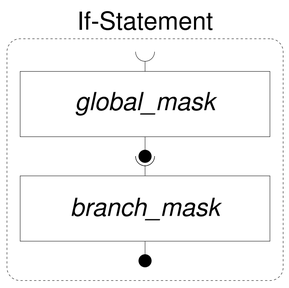

Figure 1: Masks for building up a hierarchy.

Example: If-Statement

```
1 bool_vec global = parent;
2 bool_vec branch = select(global, a > b);
3 if (any(branch)) {
4   r = select(branch, a + 1, r); }
5 branch ^= global;
6 if (any(branch)) {
7   r = select(branch, a - 1, r); }
```

Listing 3: Flattened conditional scopes

Selecting the Optimal SIMD Width

Mixed bit-width data types might cause some SIMD lanes to be unused.

Remedy

Introduce *virtual vectors* (array of vectors) to always match full SIMD width.

$$\text{array\_size}(\text{type}) = \frac{\text{sizeof}(\text{type})}{\text{sizeof}(\text{type}_{\text{min}})}$$

Steps

1. split expression tree into mono-type expressions
2. apply conversion functions
3. insert virtual vectors for larger types

⇒ array_size(float) = 2

Example: int16 and float

```
1 int16 y = x;
2 float z = y * 2;
```

Listing 4: Scalar code (x is s)

```
1 int16_vec y = broadcast(x);
2 int16_vec tmp = y * broadcast(2);
3 float_vec z = conv_i16_flt(tmp);
```

Listing 5: Splitting expressions

```
1 int16_vec y = broadcast(x);
2 int16_vec tmp = y * broadcast(2);
3 float_vec z[2];
4 z[0] = conv_i16_flt(tmp, 0);
5 z[1] = conv_i16_flt(tmp, 1);
```

Listing 6: Insertion of virtual vectors

Results

Compilers

Hipacc SPMD
ISPC SPMD
ICC C++
Clang C++
GCC C++

Table 2: 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

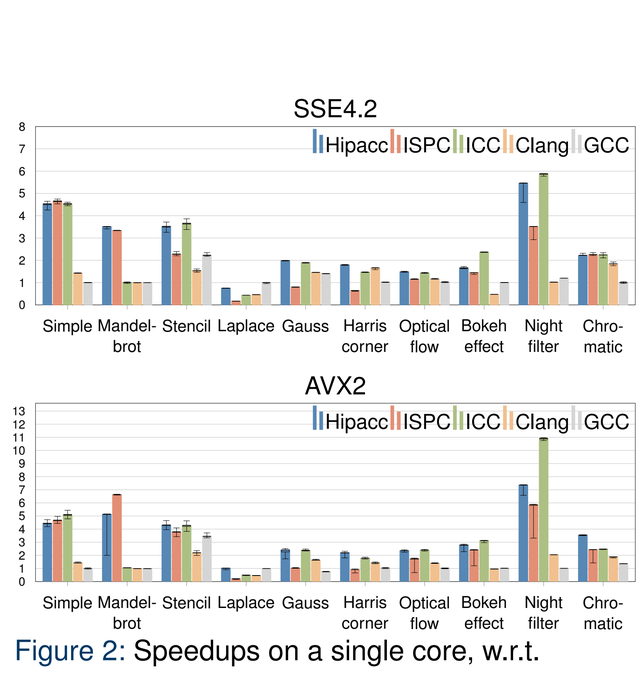

Figure 2: Speedups on a single core, w.r.t. non-vectorized baseline.

www12.cs.fau.de/downloads/reiche/autovec