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.

<table>
<thead>
<tr>
<th>Marker Property</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>same value 3</td>
</tr>
<tr>
<td>sa</td>
<td>same value, aligned 4</td>
</tr>
<tr>
<td>c</td>
<td>consecutive values (3, 4, 5, 6)</td>
</tr>
<tr>
<td>ca</td>
<td>consecutive values, aligned (4, 5, 6, 7)</td>
</tr>
<tr>
<td>T</td>
<td>unknown values (2, 3, 4, 6)</td>
</tr>
</tbody>
</table>

Lattice of Whole-Function Vectorization: \( L_{WFV} = \{ s, sa, c, ca, T \} \)

Example Rule

Add operator:

\[
[v \leftarrow \circ(x, y)] a = a | v \rightarrow \text{if } a(x) = a \land a(y) = a \text{ (1)}
\]

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: \( L_{DSL} = \{ s, v \} \)

Rules

Arithmetic and comparison operators:

\[
[v \leftarrow \circ(x, y)] a = a | v \rightarrow \text{if } a(x) = a \land a(y) = a \text{ (1)}
\]

Memory loads:

\[
[v \leftarrow \text{load}(d)] a = a | v \rightarrow \text{if } \# \text{ is image} \text{ (2)}
\]

Due to analyzing source code with scopes, assignment operators:

\[
[v \leftarrow \text{assign}_C(x)] a = a | v \leftarrow \text{if } \#_{CC}: d(c) = v \text{ (3)}
\]

Algorithm

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

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

Example

\[
\text{Vars: } a, b, c, d, \text{input, output, t id}
\]

\[
\text{Dep: } [a, [a]], [b, [\text{input}]], [c, [b, \text{t id}]], [d, (a, c)], [\text{output}, (d)]
\]

Vectors: input, output, t id, b, c, d

Source-to-Source Vectorization

Flattened pure data flow might execute unnecessary branches.

Remedy

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

Example: If-Statement

\[
[a \leftarrow \text{branch_mask}] r = \begin{cases} 
\text{if (any(branch))} & 1 \text{ else if (any(branch))} \\
\text{if (any(branch))} & 2 \text{ else if (any(branch))} \\
\text{if (any(branch))} & 3 \text{ else if (any(branch))} \\
\text{if (any(branch))} & 4 \text{ else if (any(branch))} \\
\text{if (any(branch))} & 5 \text{ else if (any(branch))} \\
\text{if (any(branch))} & 6 \text{ else if (any(branch))} \\
\text{if (any(branch))} & 7 
\end{cases}
\]

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.

\[
\text{array_size(type)} = \frac{\text{sizeof(type)}}{\text{sizeof(type_min)}}
\]

Steps

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

Example: int16 and float

\[
\begin{align*}
\text{int16 } y &= x; \\
\text{float } z &= y \times 2; \\
\text{int16 } y &= \text{broadcast}(x); \\
\text{int16 } y &= \text{broadcast}(x) + \text{broadcast}(2); \\
\text{float } z &= \text{conv_i16_flt}(\text{top}); \\
\text{float } z &= \text{conv_i16_flt}(\text{top}); \\
\text{float } z &= \text{conv_i16_flt}(\text{top}); \\
\end{align*}
\]

Table 2: Geometric mean of speedups across all benchmarks.

<table>
<thead>
<tr>
<th>Compilers</th>
<th>SSE4.2</th>
<th>HIPACC</th>
<th>SPMD</th>
<th>ISPC</th>
<th>SPMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>2.33</td>
<td>1.45</td>
<td>1.97</td>
<td>1.10</td>
<td>1.14</td>
</tr>
<tr>
<td>AVX2</td>
<td>3.14</td>
<td>2.07</td>
<td>2.50</td>
<td>1.33</td>
<td>1.14</td>
</tr>
</tbody>
</table>

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