Auto-vectorization for Image Processing DSLs

Oliver Reiche, Christof Kobylko, Frank Hannig, and Jürgen Teich
Hardware/Software Co-Design, Friedrich-Alexander University Erlangen-Nürnberg
LCTES, June 21, 2017, Barcelona
Motivation

(a) Bokeh effect

(b) Night filter

Domain of Image Processing

- many independent pixels
- mostly same operations per pixel

Performing the same instruction on a bunch of independent data elements?

⇒ use vector instructions (SIMD)
Outline

Whole-Function Vectorization

Auto-Vectorization for DSLs

Evaluation and Results

Conclusion
Whole-Function Vectorization
Whole-Function Vectorization [3]

Single Program, Multiple Data (SPMD)

• same program (kernel) for all threads
• consecutive threads process consecutive data

Flattening Control Flow

```
1 if (a > b) {
2   r = a + 1; }
3 else {
4   r = a - 1; }
```

Listing 1: Scalar control flow

```
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

Vectorizing Flattened Control Flow in SPMD

• transform groups of threads to vectors
• represent thread ID (tid) as vector of **consecutive values**
  e.g., (0, 1, 2, 3) or (4, 5, 6, 7).
Whole-Function Vectorization

Vectorization Markers

Table: Available markers in Whole-Function Vectorization.

<table>
<thead>
<tr>
<th>Marker</th>
<th>Property</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>same value</td>
<td>3</td>
</tr>
<tr>
<td>sa</td>
<td>same value, aligned</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>consecutive values</td>
<td>(3, 4, 5, 6)</td>
</tr>
<tr>
<td>ca</td>
<td>consecutive values, aligned</td>
<td>(4, 5, 6, 7)</td>
</tr>
<tr>
<td>⊤</td>
<td>unknown values</td>
<td>(2, 1, 3, 4)</td>
</tr>
</tbody>
</table>

Lattice of Whole-Function Vectorization: \(\mathbb{L}_{WFV} = \{s, sa, c, ca, ⊤\}\)

Example Rule

Add operator:

\[
[[v \leftarrow \oplus(x, y)]]^# a = a | v \rightarrow
\]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
a(x), a(y) & sa & s & ca & c & ⊤ \\
\hline
sa & sa & s & ca & c & T \\
s & s & s & c & c & T \\
ca & ca & c & T & T & T \\
c & c & c & T & T & T \\
⊤ & T & T & T & T & T \\
\hline
\end{array}
\]
Auto-Vectorization for DSLs
Auto-Vectorization for DSLs

Domain-Specific Languages (DSLs)
Compact, high-level representations of algorithms.
- restricted to certain domain
- target-specific code generation
- often employing source-to-source compilation

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
Vectorization Analysis: Simplification

DSL Restriction: Only Relative Indexing

Imagine threads #4, #5, #6, and #7 load neighboring pixels:

- `image[-4][0]` ⇒ load vector at `(0, 1, 2, 3)`
- `image[0][0]` ⇒ load vector at `(4, 5, 6, 7)`
- `image[4][0]` ⇒ load vector at `(8, 9, 10, 11)`

⇒ 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: \( \mathbb{L}_{DSL} = \{ s, v \} \)
Vectorization Analysis: Rules

tid parameter (instance ID):
\[
[[ v \leftarrow \text{tid}]^{\#} a = a \mid v \mapsto v]
\] (1)

Arithmetic and comparison operators:
\[
[[ v \leftarrow \text{op}(x, y)]^{\#} a = a \mid v \mapsto \begin{cases} s & \text{if } a(x) = s \land a(y) = s \\ v & \text{else} \end{cases}]
\] (2)

Memory loads:
\[
[[ v \leftarrow \text{load}_p(d)]^{\#} a = a \mid v \mapsto \begin{cases} v & \text{if } p \text{ is image} \\ a(d) & \text{else} \end{cases}]
\] (3)

Due to analyzing source code with scopes, assignment operators:
\[
[[ v \leftarrow \text{assign}_c(x)]^{\#} a = a \mid v \mapsto \begin{cases} v & \text{if } \exists c \in C : a(c) = v \\ a(x) & \text{else} \end{cases}]
\] (4)
Vectorization Analysis: 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 3: Example DSL code

Example

Variables $a,b,c,d,input,output,tid$

Dependencies $[a,(a)], [b,(a,input)], [c,(b,tid)], [d,(a,c)], [output,(d)]$

Vectors $input,output,tid,b,c,d$
Evaluation and Results
Experimental Setup

Compilers

- **Hipacc (DSL)**: domain-specific, SPMD
- **ISPC (Intel)**: special-purpose, SPMD
- **ICC (Intel)**: general-purpose C++
- **Clang (LLVM)**: general-purpose C++
- **GCC (Gnu)**: general-purpose C++
- **Baseline**: GCC with disabled auto-vectorization

Evaluation Environment

- **OS**: Ubuntu 14.04.5 LTS
- **CPU**: Intel Xeon E5-1620 v3
  - We disabled:
    - Hyper-Threading
    - SpeedStep
    - Turbo-Boost

Algorithm Classes

- **ISPC Suite**: simple, Mandelbrot, stencil
- **Preprocessing**: Gaussian, Laplace, Harris Corner, Optical Flow
- **Postprocessing**: bokeh effect, night filter, chromatic aberration

---

1 enabled auto-vectorization and aggressive optimization scheme (-O3) with pointer aliasing disabled
Speedup

Table: Geometric mean of speedups across all benchmarks.

<table>
<thead>
<tr>
<th></th>
<th>Hipacc</th>
<th>ISPC</th>
<th>ICC</th>
<th>Clang</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE4.2</td>
<td>2.32</td>
<td>1.45</td>
<td>1.97</td>
<td>1.10</td>
<td>1.14</td>
</tr>
<tr>
<td>AVX</td>
<td>2.53</td>
<td>1.40</td>
<td>2.27</td>
<td>1.32</td>
<td>1.13</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: Speedups on a single core, w.r.t. non-vectorized baseline.
Conclusion
Conclusion

Contributions
We presented how to:

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

Results
We showed:

- vectorization from unmodified existing DSL codes
- comparable performance to ISPC and ICC
- good results even without applying domain-specific optimizations

Applicable to other DSLs as well, such as Halide [2] and ExaSlang [1].
Thanks for listening.

Any questions?

Title  Auto-vectorization for Image Processing DSLs
Speaker  Oliver Reiche <oliver.reiche@fau.de>
References I


Backup Slides
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

```
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 5: Flattened with 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

⇒ \(\text{array\_size}(\text{float}) = 2\)

### Example: int16 and float

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

Listing 6: 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 7: 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 8: Insertion of virtual vectors