Symbolic Crosschecking of Floating-Point and SIMD Code

Peter Collingbourne, Cristian Cadar, Paul H J Kelly

Department of Computing, Imperial College London

13 April, 2011
**SIMD**

- Single Instruction Multiple Data
- A popular means of improving the performance of programs by exploiting data level parallelism
- SIMD vectorised code operates over one-dimensional arrays of data called vectors

```c
__m128 c = _mm_mul_ps(a, b);
/* c = { a[0]*b[0], a[1]*b[1], a[2]*b[2], a[3]*b[3] } */
```

- SIMD code is typically translated manually based on a reference scalar implementation
- Manually translating scalar code into an equivalent SIMD version is a difficult and error-prone task
SIMD and Floating Point

- SIMD vectorised code frequently makes intensive use of floating point arithmetic
- Developers have to reason about subtle floating point semantics:
  - Associativity
  - Distributivity
  - Precision
  - Rounding
Spot the Difference

Scalar

\[
out[0] = x[0] \times y[0] \times z[0];
\]

SIMD

\[
outv = _mm\_mul\_ps(xv, _mm\_mul\_ps(yv, zv));
\]

Scalar

\[
out[0] = \text{std} :: \text{min}(x[0], y[0]);
\]

SIMD

\[
outv = _mm\_min\_ps(xv, yv);
\]
min and max are not commutative or associative in FP!

Scalar

$$\text{out}[0] = \text{std} :: \text{min}(x[0], y[0]);$$

SIMD

$$\text{out}_v = \_\text{mm}\_\text{min}\_ps(x_v, y_v);$$

- **SSE \_mm\_min\_ps:**
  
  $$\text{min}(X, Y) = \text{Select}(X <_{\text{ord}} Y, X, Y)$$

- **$X <_{\text{ord}} Y$ evaluates to false if either of $X$ or $Y$ is NaN**

  $$\text{min}(X, \text{NaN}) = \text{NaN}$$
  $$\text{min}(\text{NaN}, Y) = Y$$

  $$\text{min}(\text{min}(X, \text{NaN}), Y) = \text{min}(\text{NaN}, Y) = Y$$
  $$\text{min}(X, \text{min}(\text{NaN}, Y)) = \text{min}(X, Y)$$
min and max are not commutative or associative in FP!

Scalar

```cpp
out[0] = std::min(x[0], y[0]);
```

SIMD

```cpp
outv = __mm_min_ps(xv, yv);
```

- **SSE `__mm_min_ps`:**
  
  $$\min(X, Y) = \text{Select}(X <_{\text{ord}} Y, X, Y)$$

- **$X <_{\text{ord}} Y$ evaluates to false if either of $X$ or $Y$ is NaN**

  $$\begin{align*}
  \min(X, \text{NaN}) &= \text{NaN} \\
  \min(\text{NaN}, Y) &= Y
  \end{align*}$$

  $$\begin{align*}
  \min(\min(1, \text{NaN}), 200) &= \min(\text{NaN}, 200) = 200 \\
  \min(1, \min(\text{NaN}, 200)) &= \min(1, 200) = 1
  \end{align*}$$
min and max are not commutative or associative in FP!

Scalar

```cpp
out[0] = std::min(x[0], y[0]);
```

SIMD

```cpp
outv = _mm_min_ps(xv, yv);
```

- **SSE _mm_min_ps:**
  \[
  \min(X, Y) = \text{Select}(X <_{\text{ord}} Y, X, Y)
  \]

- **X <_{\text{ord}} Y** evaluates to false if either of X or Y is NaN

- **libstdc++ std::min**
  \[
  \text{stl}_{\text{min}}(X, Y) = \min(Y, X)
  \]

- **out[0] = \min(x[0], y[0])**
- **outv[0] = \min(yv[0], xv[0])**
Symbolic Execution for SIMD

- A novel automatic technique based on symbolic execution for verifying that the SIMD version of a piece of code is equivalent to its (original) scalar version

- Symbolic execution can automatically explore multiple paths through the program

- Determines the feasibility of a particular path by reasoning about all possible values using a constraint solver
Challenges

- Huge number of paths involved in typical SIMD vectorisations
- The current generation of symbolic execution tools lack symbolic support for floating point and SIMD
  - Due to lack of available constraint solvers
  - (Recent development: floating point support in CBMC)
Architecture

- **Execution Engine**
  - Scalar code
    \[ x[i] \times y[i] \times z[i] \]
  - SIMD code
    \[ _mm\_mul\_ps(xv, _mm\_mul\_ps(yv, zv)) \]
  - Test harness
    ```
    assert(scalar(...) == simd(...));
    ```
choose (scalar path, SIMD path)

scalar code
x[i] * y[i] * z[i]

SIMD code
_mm_mul_ps(xv, _mm_mul_ps(yv, zv))

test harness
assert(scalar(...) == simd(...));
Architecture

choose (scalar path, SIMD path)

paths equiv?

yes

no mismatch found!

scalar code
\[ x[i] \times y[i] \times z[i] \]

SIMD code
\[
_mm\_mul\_ps(xv, _mm\_mul\_ps(yv, zv))
\]

test harness
assert(scalar(...) == simd(...));
Architecture

execution engine

choose (scalar path, SIMD path)

paths equiv?

no more paths

all paths equivalent

mismatch found!

scalar code
\[ x[i] \times y[i] \times z[i] \]

SIMD code
\[ _mm\_mul\_ps(xv, _mm\_mul\_ps(yv, zv)) \]

test harness
\[ assert(scalar(...) == simd(...)); \]
Architecture

execution engine

choose (scalar path, SIMD path)

scalar code
\[ x[i] \times y[i] \times z[i] \]

SIMD code
\_mm\_mul\_ps(xv, \_mm\_mul\_ps(yv, zv))

test harness
assert(scalar(...) == simd(...));

paths equiv?

no more paths

all paths equivalent

no

mismatch found!

yes
Symbolic Execution – Operation

- Program runs on *symbolic input*, initially unconstrained
- Each variable may hold either a concrete or a symbolic value
- Symbolic value: an input dependent expression consisting of mathematical or boolean operations and symbols
  - For example, an integer variable \( i \) may hold a value such as \( x + 3 \)
- When program reaches a branch depending on symbolic input
  - Determine feasibility of each side of the branch
  - If both feasible, *fork* execution and follow each path separately, adding corresponding constraints on each side
Symbolic Execution – Example

int x;
msymbolic(x);

if (x > 0) {
    ...
} else {
    ...
}

if (x > 10) {
    ...
} else {
    ...
}
int x;
mksymbolic(x);

if (x > 0) {
    ...
} else {
    ...
}

if (x > 10) {
    ...
} else {
    ...
}
Symbolic Execution – Example

```plaintext
int x;
mksymbolic(x);

if (x > 0) {
    ...
} else {
    ...
}

if (x > 10) {
    ...
} else {
    ...
}
```
Huge number of paths involved in typical SIMD vectorisations

The current generation of symbolic execution tools lack symbolic support for floating point and SIMD
  - Due to lack of available constraint solvers
  - (Recent development: floating point support in CBMC)
Architecture

execute

 scalar code
  x[i] * y[i] * z[i]

SIMD code
  _mm_mul_ps(xv, _mm_mul_ps(yv, zv))

paths equiv?
  no
    mismatch found!
  yes
    all paths equivalent

choose
  (scalar path, SIMD path)

no more paths

40

Architecture

**Choose** (scalar path, SIMD path)

**Static path merging**

**Execution engine**

Scalar code:
\[ x[i] \times y[i] \times z[i] \]

SIMD code:
\[ _mm\_mul\_ps(xv, _mm\_mul\_ps(yv, zv)) \]

Test harness:
\[ \text{assert(scalar(...) == simd(...));} \]

Paths equiv?
- yes
- no

Paths equivalent:
- all paths equivalent
- mismatch found!

No more paths:
Static Path Merging

```c
for (unsigned i = 0; i < N; ++i) {
}
```

- $2^N$ paths!
Static Path Merging

\[
diff(x, y) = x > y \ ? \ x - y \ : \ y - x
\]

\[
\begin{align*}
& x > y \\
& \neg (x > y)
\end{align*}
\]

\[
\begin{array}{ll}
B & \ldots \\
& \%r1 = "x-y"
\end{array}
\begin{array}{ll}
C & \ldots \\
& \%r2 = "y-x"
\end{array}
\]

\[
D & \%r = \text{phi} [\%r1, \%B], [\%r2, \%C] \\
& \ldots
\]
Static Path Merging

\[ \text{diff}(x, y) = x > y \ ? \ x - y \ : \ y - x \]

\[ diff(x, y) = x > y \ ? \ x - y : y - x \]

\[ X \]

ABC

\[ %r1 = "x - y" \]

\[ %r2 = "y - x" \]

\[ \text{phi} [ %r1, %B ], [ %r2, %C ] \]

\[ \text{diff}(x, y) = x > y \ ? \ x - y : y - x \]

\[ \text{select} %p, %r1, %r2 \]

\[ A' \]

\[ %p = "x > y" \]

\[ B \]

\[ %r1 = "x - y" \]

\[ C \]

\[ %r2 = "y - x" \]

\[ D' \]

\[ %r = \text{select} %p, %r1, %r2 \]

\[ D' \]

\[ %r = \text{select} %p, %r1, %r2 \]
Static Path Merging

\[ \text{diff}(x, y) = x > y \ ? \ x - y : y - x \]

- A
  - \( x > y \)
  - \( \neg(x > y) \)
- B
  - \( \%r1 = "x - y" \)
- C
  - \( \%r2 = "y - x" \)
- D
  - \( \%r = \text{phi} \, [\%r1, \%B], \,[\%r2, \%C] \)
  - \( \ldots \)

\[ \begin{array}{c}
\text{morph benchmark, 16} \times 16 \text{ matrix:} \\
2^{256} \rightarrow 1
\end{array} \]
Challenges

- Huge number of paths involved in typical SIMD vectorisations
- The current generation of symbolic execution tools lack symbolic support for floating point and SIMD
  - Due to lack of available constraint solvers
  - (Recent development: floating point support in CBMC)
Architecture

- **Execution Engine**
  - Scalar code: \( x[i] * y[i] * z[i] \)
  - SIMD code: `_mm_mul_ps(xv, _mm_mul_ps(yv, zv))`
  - Test harness:
    ```
    assert(scalar(...) == simd(...));
    ```

- **Choose (scalar path, SIMD path)**
  - Paths equiv? (no more paths)
    - Yes: all paths equivalent
    - No: mismatch found!
Technique

- The requirements for equality of two floating point expressions are harder to satisfy than for integers
- Usually, the two expressions need to be built up in the same way to be sure of equality
- We can check expression equivalence via simple expression matching!
Architecture

static path merging

execution engine

choose (scalar path, SIMD path)

scalar code
\[ x[i] \times y[i] \times z[i] \]

SIMD code
\_mm\_mul\_ps(xv, \_mm\_mul\_ps(yv, zv))

test harness
assert(scalar(...) == simd(...));

paths equiv?
no
mismatch found!

yes
all paths equivalent

no more paths
Architecture

**Static Path Merging**

- Scalar code: \( x[i] \times y[i] \times z[i] \)
- SIMD code: 
  \[
  \_\_m\m\_mul\_ps(xv, \_\_m\m\_mul\_ps(yv, zv))
  \]

**Execution Engine**

- Test harness:
  \[
  \text{assert(scalar(...)} == \text{simd(...))}\
  \]

**Choose (Scalar Path, SIMD Path)**

- Canonicalisation

**Pathsequiv?**

- Paths equivalent
- Mismatch found!

**No More Paths**

- All paths equivalent
- No more paths
void zlimit(int simd, float *src, float *dst, size_t size) {
    if (simd) {
        __m128 zero4 = _mm_set1_ps(0.f);
        while (size >= 4) {
            __m128 srcv = _mm_loadu_ps(src);
            __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
            __m128 dstv = _mm_and_ps(cmpv, srcv);
            _mm_storeu_ps(dst, dstv);
            src += 4; dst += 4; size -= 4;
        }
    }
    while (size) {
        *dst = *src > 0.f ? *src : 0.f;
        src++; dst++; size--;
    }
}
void zlimit(int simd, float *src, float *dst, size_t size) {
    if (simd) {
        __m128 zero4 = _mm_set1_ps(0.f);
        while (size >= 4) {
            __m128 srcv = _mm_loadu_ps(src);
            __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
            __m128 dstv = _mm_and_ps(cmpv, srcv);
            _mm_storeu_ps(dst, dstv);
            src += 4; dst += 4; size -= 4;
        }
    }
    while (size) {
        *dst = *src > 0.f ? *src : 0.f;
        src++; dst++; size--;
    }
}
Scalar/SIMD Implementation

while (size) {
    *dst = *src > 0.f ? *src : 0.f;
    src++; dst++; size--;
}

Scalar dst[0]

Select

> src[0]

src[0] 0

src[0] 0
Scalar/SIMD Implementation

```c
void zlimit(int simd, float *src, float *dst, size_t size) {
    if (simd) {
        __m128 zero4 = _mm_set1_ps(0.f);
        while (size >= 4) {
            __m128 srcv = _mm_loadu_ps(src);
            __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
            __m128 dstv = _mm_and_ps(cmpv, srcv);
            _mm_storeu_ps(dst, dstv);
            src += 4; dst += 4; size -= 4;
        }
    }
    while (size) {
        *dst = *src > 0.f ? *src : 0.f;
        src++; dst++; size--;
    }
}
```
Scalar/SIMD Implementation

```c
__m128 zero4 = _mm_set1_ps(0.f);
while (size >= 4) {
    __m128 srcv = _mm_loadu_ps(src);
    __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
    __m128 dstv = _mm_and_ps(cmpv, srcv);
    _mm_storeu_ps(dst, dstv);
    src += 4; dst += 4; size -= 4;
}
```

<table>
<thead>
<tr>
<th>srcv</th>
<th>1.2432</th>
<th>-3.6546</th>
<th>2.7676</th>
<th>-9.5643</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpv</td>
<td>111...111</td>
<td>000...000</td>
<td>111...111</td>
<td>000...000</td>
</tr>
<tr>
<td>dstv</td>
<td>1.2432</td>
<td>0.0000</td>
<td>2.7676</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

| zero4 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |

| &     |        |        |        |        |

35
Scalar/SIMD Implementation

```c
__m128 zero4 = _mm_set1_ps(0.f);
while (size >= 4) {
    __m128 srcv = _mm_loadu_ps(src);
    __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
    __m128 dstv = _mm_and_ps(cmpv, srcv);
    _mm_storeu_ps(dst, dstv);
    src += 4; dst += 4; size -= 4;
}
```

**SIMD dst[0]**

```
&
  /
 SExt src[0]
    /
     >
      /
 src[0] 0
```
Scalar/SIMD Implementation

SIMD dst[0]

&

SExt src[0]

>

src[0] 0

Scalar dst[0]

Select

>

src[0] 0

SExt(P) & X → Select(P, X, 0)

One of our 18 canonicalisation rules
KLEE-FP

- Based on KLEE, a tool for symbolic testing of C and C++ code [Cadar, Dunbar, Engler, OSDI 2008]
- KLEE is based on the LLVM compiler [Lattner, Adve, CGO 2004]
- Supports integer constraints only; symbolic FP not allowed
- KLEE-FP: our modified version of KLEE, extended with support for:
  - Symbolic floating point
  - SIMD vector instructions
  - A substantial portion of Intel SSE instruction set
  - Static path merging
  - Extended expression canonicalisation and crosschecking

http://www.pcc.me.uk/~peter/klee-fp/ (or google klee-fp)
Evaluation

- The code base that we selected was OpenCV 2.1.0, a popular C++ open source computer vision library

Corner detection
Evaluation

- Out of the twenty OpenCV source code files containing SIMD code, we selected ten files upon which to build benchmarks.
- Crosschecked 58 SIMD/SSE implementations against scalar versions:
  - 41: verified up to a certain image size (bounded equivalence)
  - 10: found inconsistencies
  - 3: false positives
  - 4: could not run
Evaluation – Methodology

- Bounded verification
  - Started with smallest possible image size (4 × 1 in most cases)
  - Tried all possible sizes up to 16 × 16 (or 8 × 8 → 8 × 8 for benchmarks with different sized input and output images)
  - ∼ 200 or ∼ 1600 combinations per benchmark
- Verified 34 benchmarks up to these limits
- 7 on a smaller set of image sizes due to:
  - Constant sized input/output images
  - Path explosion (time/memory constraints)
  - Constraint solver blow-up
Evaluation – Limitations

Covered

Not Covered

- cvpyramids, cvstereobm, cvimgwarp, cvmorph: unrolled loops unreachable using bounded verification, constraint solver blow-up

- cvfilter: symbolic malloc

SIMD Instruction Count
## OpenCV – Mismatches found

<table>
<thead>
<tr>
<th>#</th>
<th>Benchmark/Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>eigenval (f32)</code></td>
<td>Precision</td>
</tr>
<tr>
<td>2</td>
<td><code>harris (f32)</code></td>
<td>Precision, associativity</td>
</tr>
<tr>
<td>3</td>
<td><code>morph (dilate, R, f32)</code></td>
<td>Order of min/max operations</td>
</tr>
<tr>
<td>4</td>
<td><code>morph (dilate, NR, f32)</code></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><code>morph (erode, R, f32)</code></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><code>thresh (TRUNC, f32)</code></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td><code>pyramid (f32)</code></td>
<td>Associativity, distributivity</td>
</tr>
<tr>
<td>8</td>
<td><code>resize (linear, u8)</code></td>
<td>Precision</td>
</tr>
<tr>
<td>9</td>
<td><code>transsf.43 (s16 f32)</code></td>
<td>Rounding issue</td>
</tr>
<tr>
<td>10</td>
<td><code>transcf.43 (u8 f32)</code></td>
<td>Integer/FP differences</td>
</tr>
</tbody>
</table>

- Reported to OpenCV developers
- 2 bugs (`eigenval`, `harris`) already confirmed
Conclusion and Future Work

- Automatic technique for checking correctness of SIMD vectorisations with support for floating point operations
- Applied to popular computer vision library, OpenCV
  - Proved the \textit{bounded} equivalence of 41 implementations
  - Found inconsistencies in 10
    - Precision, associativity, distributivity, rounding, ...
- Future work may involve:
  - Inequalities
  - Interval arithmetic
  - Affine arithmetic
  - Floating point counterexamples
  - OpenCL
- \url{http://www.pcc.me.uk/~peter/klee-fp/}
  (or google klee-fp)
OpenCL

- Race detection
- OpenCL runtime library
- Uses Clang as OpenCL compiler
- Used to cross-check the following benchmarks:
  - AMD SDK – TemplateC
  - Parboil – mri-q, mri-fhd, cp
  - Bullet Physics Library – softbody
- Found memory bugs, implementation differences
SSE Intrinsic Lowering

- Total of 37 intrinsics supported
- Implemented via a lowering pass that translates the intrinsics into standard LLVM instructions

Input code:

```assembly
%res = call <8 x i16> @llvm.x86.sse2.pslli.w(
    <8 x i16> %arg, i32 1)
```

Output code:

```assembly
%1 = extractelement <8 x i16> %arg, i32 0
%2 = shl i16 %1, 1
%3 = insertelement <8 x i16> undef, i16 %2, i32 0
%4 = extractelement <8 x i16> %arg, i32 1
%5 = shl i16 %4, 1
%6 = insertelement <8 x i16> %3, i16 %5, i32 1
...
%22 = extractelement <8 x i16> %arg, i32 7
%23 = shl i16 %22, 1
%res = insertelement <8 x i16> %21, i16 %23, i32 7
```
## OpenCV – Verified up to a certain size

<table>
<thead>
<tr>
<th>#</th>
<th>Bench</th>
<th>Algo</th>
<th>K</th>
<th>Fmt</th>
<th>Max Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>morph</td>
<td>dilate</td>
<td>R</td>
<td>u8</td>
<td>5 × 5</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>s16</td>
<td>16 × 16</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>u16</td>
<td>16 × 16</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>NR</td>
<td>u8</td>
<td>8 × 3</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>s16</td>
<td>16 × 16</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>u16</td>
<td>16 × 16</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td>f32</td>
<td>15 × 15</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>erode</td>
<td>R</td>
<td>u8</td>
<td>4 × 4</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>s16</td>
<td>16 × 16</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>u16</td>
<td>16 × 16</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>NR</td>
<td>s16</td>
<td>16 × 16</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td>u16</td>
<td>16 × 16</td>
</tr>
<tr>
<td>13</td>
<td>pyramid</td>
<td></td>
<td></td>
<td>u8</td>
<td>8 × 2 → 4 × 1</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>nearest neighbor</td>
<td></td>
<td>u8</td>
<td>16 × 16</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td>s16</td>
<td>16 × 16</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td>u16</td>
<td>16 × 16</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td>f32</td>
<td>16 × 16</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>linear</td>
<td></td>
<td>u8</td>
<td>16 × 16</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td>s16</td>
<td>16 × 16</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td>u16</td>
<td>16 × 16</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td>f32</td>
<td>16 × 16</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>cubic</td>
<td></td>
<td>u8</td>
<td>16 × 16</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td>s16</td>
<td>16 × 16</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td>u16</td>
<td>16 × 16</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td>f32</td>
<td>16 × 16</td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>resize</td>
<td>linear</td>
<td>s16</td>
<td>8 × 8 → 8 × 8</td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td>cubic</td>
<td>f32</td>
<td>8 × 8 → 8 × 8</td>
</tr>
<tr>
<td>28</td>
<td></td>
<td>silhouette</td>
<td></td>
<td>s16</td>
<td>8 × 8 → 8 × 8</td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td>f32</td>
<td>8 × 8 → 8 × 8</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td>u8</td>
<td>16 × 16</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td>thresh</td>
<td>BINARY</td>
<td>u8</td>
<td>16 × 16</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td>BINARY_INV</td>
<td>f32</td>
<td>16 × 16</td>
</tr>
<tr>
<td>33</td>
<td></td>
<td></td>
<td>TRUNC</td>
<td>u8</td>
<td>16 × 16</td>
</tr>
<tr>
<td>34</td>
<td></td>
<td></td>
<td>TOZERO</td>
<td>f32</td>
<td>16 × 16</td>
</tr>
<tr>
<td>35</td>
<td></td>
<td></td>
<td>TOZERO_INV</td>
<td>u8</td>
<td>16 × 16</td>
</tr>
<tr>
<td>36</td>
<td></td>
<td></td>
<td></td>
<td>f32</td>
<td>16 × 16</td>
</tr>
<tr>
<td>37</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16 × 16</td>
</tr>
<tr>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16 × 16</td>
</tr>
<tr>
<td>39</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16 × 16</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>transff.43</td>
<td></td>
<td></td>
<td>16 × 16</td>
</tr>
<tr>
<td>41</td>
<td></td>
<td>transff.44</td>
<td></td>
<td></td>
<td>16 × 16</td>
</tr>
</tbody>
</table>
eigenval and harris

- Precision
- Associativity

- Scalar:

\[ k \cdot (a + c) \cdot (a + c) \]

- SIMD:

\[
\_\_m\_m\_m\_u\_l\_p\_s(\_\_m\_m\_m\_u\_l\_p\_s(t, t), k4)
\]

\[ k4 = (k, k, k, k) \]

\[ t = (a_0 + c_0, a_1 + c_1, a_2 + c_2, a_3 + c_3) \]

- To be fixed in OpenCV
eigenval and harris

- Precision
- Associativity

- Scalar:
  
  \[ ((\text{float})k) \ast (a + c) \ast (a + c) \]

- SIMD:

  \[
  \_\_m\_m\_u\_l\_p\_s(\_\_m\_m\_u\_l\_p\_s(t, t), k4) \\
  k4 = (k, k, k, k) \\
  t = (a_0 + c_0, a_1 + c_1, a_2 + c_2, a_3 + c_3)
  \]

- To be fixed in OpenCV
eigenval and harris

- Precision
- Associativity

- Scalar:

\[((\text{float})k) \times ((a + c) \times (a + c))\]

- SIMD:

\[
\text{\textunderscore mm\textunderscore mul\textunderscore ps}(\text{\textunderscore mm\textunderscore mul\textunderscore ps}(t, t), k4)\]

\[
k4 = (k, k, k, k)\]

\[
t = (a_0 + c_0, a_1 + c_1, a_2 + c_2, a_3 + c_3)\]

- To be fixed in OpenCV