Automatic SIMD vectorization for Haskell

Leaf Petersen, **Dominic Orchard**, Neal Glew
ICFP 2013 - Boston, MA, USA

Work done at Intel Labs
**SIMD**

- Trend towards parallel architectures (multi-core & instruction-level parallelism)
- SIMD: fine-grained data parallelism
  - Vector registers: e.g. 128-bit wide (4x integers)
  
  \[
  \begin{array}{cccc}
  A_0 & A_1 & A_2 & A_3 \\
  \end{array}
  \]
  
  - Vector instructions: e.g. `vadd r1, r2`

  \[
  \begin{align*}
  r1 &= \begin{array}{cccc}
  A_0 & A_1 & A_2 & A_3 \\
  \end{array} \\
  r2 &= \begin{array}{cccc}
  B_0 & B_1 & B_2 & B_3 \\
  \end{array} \\
  \text{vadd } r1, r2 & \implies r1 = \begin{array}{cccc}
  A_0 + B_0 & A_1 + B_1 & A_2 + B_2 & A_3 + B_3 \\
  \end{array}
  \end{align*}
  \]
Exploiting SIMD

- Need to preserve semantics
- Imperative: (undecidable) dependency & effects analysis
- Functional: much easier
Good things happen, e.g., type-checking, simplification, fusions, etc.
Take home messages

• FP optimisation not exhausted: still low-hanging fruit to be had

• Vectorization is low hanging and a big win:
  • up to 6.5x speedups for HRC + vectorization

• Many standard techniques + a few extras
MIL

• Aimed at compiling high-performance functional code
• Block-structured (CFG) (loops), SSA form
• Strict + explicit thunks
• Distinguishes mutable and immutable data
• Vector primitives (numerical and array ops)
Prior to vectorization

- Core → MIL: closure conversion, explicit thunks
- Optimisations (general simplifier, unboxing, representation opts.)
- Contification [1]
  - Converts (many) tail recursion uses to loops

Vectorization

• Targets inner-most loops that are:
  ➢ reductions over immutable arrays
  ➢ initialising writes
Initialising writes

- Allocate then initialize an (immutable) array
- Two invariants:
  - Reading an array element always follows the initializing write of the element
  - Each element may be initialized only once
- Modified GHC libraries to generate initializing writes rather than mutation
unstreamRMax :: (PrimMonad m, MVector v a) => MStream m a -> Int -> m (v (PrimState m) a)
unstreamRMax s n = do
  v <- INTERNAL_CHECK (checkLength) "unstreamRMax" n
  $ unsafeNew n
  let put i x = do
    let i' = i - 1
    INTERNAL_CHECK (checkIndex) "unstreamRMax" i' n
    $ unsafeWrite v i' x
    return i'
  i <- MStream.foldM' put n s
  return $ INTERNAL_CHECK (checkSlice) "unstreamRMax" i (n-i) n
  $ unsafeSlice i (n-i) v

(used for the immutable vector too)
Transformed to immutability
+ initializing writes

unstreamRPrimMmax :: (PrimMonad m, Vector v a) => Int -> MStream m a -> m (v a)
unstreamRPrimMmax n s = do
  v <- INTERNAL_CHECK (checkLength) "unstreamRPrimMmax" n $ unsafeCreate n
  let put i x = do
      let i' = i-1
      INTERNAL_CHECK (checkIndex) "unstreamMmax" i' n $ unsafeInitElem v i' x
      return i'
  i <- MStream.foldM' put n s
  v' <- basicUnsafeInited v
  return $ INTERNAL_CHECK (checkSlice) "unstreamRPrimMmax" i (n-i) n $ unsafeSlice i (n-i) v'
Start with...
Vectorize...

- Entry
- Vector
- Cleanup
- Serial
- Exit
Vectorization

• Transform each instruction
  ▸ ... depending on properties of instruction/arguments

• Let dead-code elimination handle clean up

\[
\begin{align*}
x & = \ldots \\
vectorize \\
x^s & = \ldots \\
x^v & = \ldots \\
x^l & = \ldots \\
( & x^b = \ldots \\
\text{first value (scalar)} \\
\text{vector version} \\
\text{last value (scalar)} \\
\text{basis vector} \\
\end{align*}
\]
Vectorization (simple)

- Pointwise operations, promote all constants

\[
y = + (x, 4) \quad \Rightarrow \quad y^v = \langle + \rangle (x^v, \langle 4, 4, 4, 4 \rangle)
\]

- Pointwise op
- Vector-promoted constant
- Projection

\[
y^1 = y^v ! 3
\]

- Last value of \( y \), needed if \( y \) is live-out
Vectorization (simple)

- Pointwise operations, promote all constants

\[ y = x[z] \quad \rightarrow \quad y^v = \langle x^v \rangle[\langle z^v \rangle] \]

general array load on vectors “gather”

equivalent to:
\[ y^v = \langle x^v_0[z^v_0], x^v_1[z^v_1], x^v_2[z^v_2], x^v_3[z^v_3] \rangle \]

\[ x[z] = y \quad \rightarrow \quad \langle x^v \rangle[\langle z^v \rangle] = y^v \]

general array store on vectors “scatter”
Why is this (sometimes) naive?

- General vector array loads/stores not widely supported
- Specialised versions often faster
- Dependency between scalar/vector
- Does too much work (non-optimal)
  - HRC instead tracks “induction variables” and uses this information to optimise
Induction variables

• Base induction variables

• Loop-carried variable with constant step

• Derived induction variable

• Induction variable $+$ constant or $\times$ constant

$L_1(x)$: .... $x$ is a base induction variable (step 1)

$x' = + (x, 1)$ $x'$ is a derived induction variable

if ... goto $L_1(x')$ else goto $L_{end}(...)$
Vectorizing base I.V.s e.g. \( x \)

\[
L_1(x) : \ldots \nonumber
\]
\[
x' = + (x, 1) \nonumber
\]
\[
\text{if } \ldots \text{ goto } L_1(x') \text{ else goto } L_{\text{end}}(...)
\]

**HRC:**
\[
x^b = \langle 0, 1, 2, 3 \rangle
\]
\[
x^v = \langle + \rangle (x^b, \langle x^s, x^s, x^s, x^s \rangle)
\]
\[
x^l = x^s + 3
\]

**basis vector**

**last basis value**
Vectorizing derived I.V.s e.g. $x'$

$L_1(x)$: ....

\[ x' = + (x, 1) \]

if ... goto $L_1(x')$ else goto $L_{\text{end}}(...)$

**Simple:**

\[ x^b = \langle 0, 1, 2, 3 \rangle \]
\[ x^v = \langle + \rangle (x^b, \langle x^s, x^s, x^s, x^s \rangle) \]
\[ x'^v = \langle + \rangle (x^v, \langle 1, 1, 1, 1 \rangle) \]

**HRC:**

\[ x'^b = x^b \]
\[ x'^s = x^s + 1 \]
\[ x'^v = \langle + \rangle (x'^b, \langle x'^s, x'^s, x'^s, x'^s \rangle) \]
\[ x'^l = x^l + 1 \]

2 vector ops
+ 1 promotions

Removes dependence on $x^v$
1 vector op
+ 1 scalar op
+ 1 promotion
Vectorizing induction variables

\[ x'^v = \langle + \rangle(x^v, \langle 1, 1, 1, 1 \rangle) \]

\[ = \langle + \rangle(\langle + \rangle(x^b, \langle x^s, x^s, x^s, x^s \rangle), \langle 1, 1, 1, 1 \rangle) \]

\[ = \langle + \rangle(x^b, \langle + \rangle(\langle x^s, x^s, x^s, x^s \rangle, \langle 1, 1, 1, 1 \rangle)) \] (assoc)

“Naturality” of \textit{promotion}:

\[ \langle f \rangle \cdot (\text{promote} \times \text{promote}) \equiv \text{promote} \cdot f \]

\[ \langle f(a,b), f(a,b), f(a,b), f(a,b) \rangle \equiv \langle f(\langle a, a, a, a \rangle, \langle b, b, b, b \rangle) \rangle \]

\[ = \langle + \rangle(x^b, \langle x^s + 1, x^s + 1, x^s + 1, x^s + 1 \rangle) \] (naturality)

\[ = \langle + \rangle(x^b, \langle x's, x's, x's, x's \rangle) \] (simplify)
Vectorizing loads/stores

- Specialised gathers and scatters for unit strides provide higher performance

\[ y = x[z] \quad \rightarrow \quad y^v = x^s[\langle z^s:\rangle] \]

(when \( x \) is loop invariant, \( z \) is a unit stride induction variable)
Results

Test framework: Intel Xeon, 256-bit register AVX
Speedup over no SIMD
Higher is Better

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Add</td>
<td>1.05</td>
</tr>
<tr>
<td>Vector Sum</td>
<td>1.84</td>
</tr>
<tr>
<td>Dot Product</td>
<td>2.00</td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>1.34</td>
</tr>
<tr>
<td>Nbody</td>
<td>6.69</td>
</tr>
<tr>
<td>1D Convolution</td>
<td>3.52</td>
</tr>
<tr>
<td>2D Convolution</td>
<td>3.02</td>
</tr>
<tr>
<td>Blur</td>
<td></td>
</tr>
</tbody>
</table>

**SIMD Vectorisation Performance**

- **HRC SIMD**

**Repa**

**Repa lib**
Conclusions: what is important?

- Purity at the top level
- Fusion
- Understanding effects at the implementation level
- Use program properties (induction vars)
- Keep dependencies between scalars/vectors separate
Conclusions

• Future work
  • Masked instructions
  • Vectorised allocations
  • Alignment

• SIMD was straightforward to add to HRC, with very good results

We told ‘em we could do parallelism!
Backup Slides
Pillar

- C-like language
- Managed memory with garbage-collection
- Tail calls
- Continuations
- Compiles to C

<table>
<thead>
<tr>
<th></th>
<th>Vector Add</th>
<th>Vector Sum</th>
<th>Dot Product</th>
<th>N Body Repa</th>
<th>Matrix Multiply</th>
<th>1D Convolution</th>
<th>2D Convolution</th>
<th>Blur</th>
</tr>
</thead>
<tbody>
<tr>
<td>GHC</td>
<td>0.15</td>
<td>0.72</td>
<td>0.24</td>
<td>0.34</td>
<td>0.35</td>
<td>0.02</td>
<td>0.34</td>
<td>0.45</td>
</tr>
<tr>
<td>GHC LLVM</td>
<td>0.15</td>
<td>0.73</td>
<td>0.35</td>
<td>0.97</td>
<td>0.98</td>
<td>0.02</td>
<td>0.58</td>
<td>0.86</td>
</tr>
<tr>
<td>HRC SIMD</td>
<td>1.05</td>
<td>1.84</td>
<td>2.00</td>
<td>4.37</td>
<td>1.34</td>
<td>6.68</td>
<td>5.03</td>
<td>3.02</td>
</tr>
</tbody>
</table>