Code CONstructing User Tool

Christopher Anand

Faculty of Engineering
Department of Computing and Software

McMaster University
Real-Time 3d MRI

• teraflops (on a budget)
• patient safety = provably correct
• novel optimization algorithms
Coconut

Application scientist

Computer scientist

DSP Guru

Model

Algorithms

code transformations

profiling

native code

no pipes ⇒ no leaks
One-Stop Shop

- no .s editing
- no untracked assumptions

- faster code
- better quality code
- faster development
Today's Menu

- Coconut
- HasSPUmbler
- MASS
  - atanh
  - asinh
- ExSSP
- multiloop
Why reinvent the wheel?
Declarative Assembly Language with Alternative Calculations

- in-lining special functions
- polynomial approximation in vector registers
- table lookup in vector registers
- user supplied tuned implementations
EDSL Example: Vector Exp()

\[
\begin{align*}
\text{exp2 v} &= \text{result} \\
\text{where} & \\
\text{result} &= \text{fm exponent evalPoly} \\
\text{vlog2} &= \text{fma v (unfloats $ replicate 4 (1/log 2))} \\
& \quad (\text{unfloats $ replicate 4 (127)}) \\
\text{expAsU32} &= \text{cfltu vlog2 (173 - 23)} \\
\text{exponent} &= \text{SPU.and expAsU32 $ unwrds4 0x7f800000} \\
\text{frac} &= \text{selb (unwrds4 0x3f800000) v (unwrds4 0x007fffff)} \\
\text{look1} &= \text{join [rotmi expAsU32 (-2), rotqbii expAsU32 (-2), roti expAsU32 (-2)]} \\
\text{look2} &= \text{shufb look1 look1} \\
& \quad (\text{unbytes [1,1,1,1, 5,5,5,5, 9,9,9,9, 13,13,13,13]}) \\
\text{look3} &= \text{selb (unwrds4 0x00010203) look2 (unwrds4 0x1c1c1c1c)} \\
\text{coeffs} &= \text{contigLookup table look3 -- 4 pipe1} \\
\text{evalPoly} &= \text{hornerV coeffs frac -- 3 pipe0} \\
\text{table} &= \text{contigTable expCoeffs24bitsFrom0}
\end{align*}
\]
EDSL Example: Vector Exp()

```
exp2 v = result
where
  result = fm exponent evalPoly
vlog2 = fma v (unfloats $ replicate 4 (1/log 2))
  (unfloats $ replicate 4 (127))
expAsU32 = cfltu vlog2 (173 - 23)
exponent = SPU.and expAsU32 $ unwrds4 0x7f800000

frac = selb (unwrds4 0x3f800000) v (unwrds4 0x007fffffff)
look1 = join [rotmi expAsU32 (-2), rotqbii expAsU32 (-2), roti expAsU32 (-2)]
look2 = shufb look1 look1
  $ unbytes [1,1,1,1, 5,5,5,5, 9,9,9,9, 13,13,13,13]
look3 = selb (unwrds4 0x00010203) look2 (unwrds4 0x1c1c1c1c1c)
coeffs = contigLookup table look3 -- 4 pipe1

evalPoly = hornerV coeffs frac -- 3 pipe0

table = contigTable expCoeffs24bitsFrom0
```
EDSL Example: Vector Exp()

```haskell
exp2 v = result
  where
    result = fm exponent evalPoly
    vlog2 = fma v (unfloats $ replicate 4 (1/log 2))
           (unfloats $ replicate 4 (127))
    expAsU32 = cfltu vlog2 (173 - 23)
    exponent = SPU.and expAsU32 $ unwrds4 0x7f800000

    frac = selb (unwrds4 0x3f800000) v (unwrds4 0x007fffff)
    look1 = join [rotmi expAsU32 (-2), rotqbii expAsU32 (-2), roti expAsU32 (-2)]
    look2 = shufb look1 look1
    $ unbytes [1,1,1,1, 5,5,5,5, 9,9,9,9, 13,13,13,13]
    look3 = selb (unwrds4 0x00010203) look2 (unwrds4 0x1c1c1c1c)
    coeffs = contigLookup table look3 -- 4 pipe1

    evalPoly = hornerV coeffs frac -- 3 pipe0

    table = contigTable expCoeffs24bitsFrom0
```

Haskell functions to define constants
EDSL Example: Vector Exp()

given

exp2 v = result
where
result = fm exponent evalPoly
vlog2 = fma v (unfloats $ replicate 4 (1/log 2))
        (unfloats $ replicate 4 (127))
expAsU32 = cftu vlog2 (173 - 23)
exponent = SPU.and expAsU32 $ unwrds4 0x7f800000

frac  = selb (unwrds4 0x3f800000) v (unwrds4 0x007fffff)
look1 = join [rotmi expAsU32 (-2), rotqbii expAsU32 (-2), roti expAsU32 (-2)]
look2 = shufb look1 look1
        $ unbytes [1,1,1,1, 5,5,5,5, 9,9,9,9, 13,13,13,13]
look3 = selb (unwrds4 0x00010203) look2 (unwrds4 0x1c1c1c1c1c)
coeffs = contigLookup table look3 -- 4 pipe1

evalPoly = hornerV coeffs frac -- 3 pipe0

table = contigTable expCoeffs24bitsFrom0
EDSL Example: Vector Exp()

```
ex2 v = result
where
  result = fm exponent evalPoly
  vlog2 = fma v (unfloats $ replicate 4 (1/log 2))
           (unfloats $ replicate 4 (127))
  expAsU32 = cftu vlog2 (173 - 23)
  exponent = SPU.and expAsU32 $ unwrds4 0x7f800000

  frac   = selb (unwrds4 0x3f800000) v (unwrds4 0x007fffff)
  look1 = [rotmi expAsU32 (-2), rotqbii expAsU32 (-2), roti expAsU32 (-2)]
  look2 = shufb look1 look1
           $ unbytes [1,1,1,1, 5,5,5,5, 9,9,9,9, 13,13,13,13]
  look3 = selb (unwrds4 0x00010203) look2 (unwrds4 0x1c1c1c1c1c)
  coeffs = contigLookup table look3 -- 4 pipe1

  evalPoly = hornerV coeffs frac -- 3 pipe0

  table = contigTable expCoeffs24bitsFrom0
```
Scheduler

- everything else is about forming BIG code blocks
- implemented modulo scheduling
- framework for constraint programming
- modulo scheduling worked on small codegraphs
- targets SPU (was PPC+VMX)
- new semantics!
(6) if any $K$ overflows, need to branch code: shift bits right, to catch overflow, then left, by different amounts, then rotate together and .1. together (note: shifting can be done with integer multiplication to rebalance pipeline, but probably at the expense of a register)

$$\begin{align*}
ss1 &= \text{orbi newKwithOverflow } 32 \\
ss2 &= \text{clz } ss1 \\
ss3 &= \text{mpy } ss2 \times \text{unwrds } [0,2,6,18] \\
ss4 &= \text{shufb } ss3 \times \text{roti (fBAddr old) } 3 \\
&\hspace{.5cm} \times \text{unbytes } [x\text{Word*4+3,y\text{Word*4+3,z\text{Word*4+3,dWord*4+19]}+[20..31]} \\
\text{codeOffset'} &= \text{sumb ss4 ss4}
\end{align*}$$

(7) branch out of loop on negative density by setting branch index to -1

```haskell
\text{codeOffset} = \text{inject codeOffsetInjection} \\
\hspace{.5cm} \times \text{selb codeOffset'} \text{ selectForExit selectForExit} \\
\text{selectForExit} = \text{rotmai (if dw==0 then join [density,gxyzd]} \\
\hspace{1cm} \text{else density} \\
\hspace{1cm} (-32) \hspace{.5cm} \text{-- want sign bit as select mask}
```

(8) get address for branch and branch hint (this has to be done 15 cycles in advance)

```haskell
\text{branchAddress} = \text{mpyi codeOffset} \times \text{(mlBlockSize wolf)}
```

(9) separate unoverflowed parts, and use for spline lookups (this can also be done with integer arithmetic, keeping truncating the lowest bytes with a shift)

```haskell
\text{xKer} = (iInterp input) x\text{Word newKwithOverflow} \\
\text{yKer} = (iInterp input) y\text{Word newKwithOverflow} \\
\text{zKer} = (iInterp input) z\text{Word newKwithOverflow}
```
MultiLoop

- restricted control flow (loop+switch)
- software pipelining with conditionals
if fromAligned
then
  if dX /= 0
    then let ([stencil1, stencil2], ls1)
        = nwayOps [moveYZ 0 (dY,dZ) dX
                   oldAddr
                   (loadAligned,storeAligned)
                   stencil0
                   ,alignedXMove (dY,dZ) dX (0,0)
                   oldAddr
                   stencil1
       ]
        ls0
      in (stencil2, ls1)
  else moveYZ 0 (dY,dZ) 0
        oldAddr
        (loadAligned,storeAligned)
        stencil0
        ls0
        {-(\(x,y,z)\)->error ("ax=0" ++ (show (map sRow x))))} $ -}
(6) if any K overflows, need to branch code: shift bits right, to catch overflow, then left, by different amounts, then rotate together and .1. together (note: shifting can be done with integer multiplication to rebalance pipeline, but probably at the expense of a register)

\[
\begin{align*}
ss1 &= orbi \text{ newKwith0Overflow} \ 32 \\
ss2 &= \text{ clz } ss1 \\
ss3 &= \text{ mpy } ss2 \ $ \ \text{ unwrds} \ [0,2,6,18] \\
ss4 &= \text{ shufb } ss3 \ (\text{ roti} \ (fBAaddr \ old) \ 3) \\
    & \ $ \ \text{ unbytes} \ $ \ [xWord*4+3, yWord*4+3, zWord*4+3, dWord*4+19] ++ [20..31] \\
codeOffset' &= \text{ sumb } ss4 \ ss4
\end{align*}
\]

(7) branch out of loop on negative density by setting branch index to -1

\[
\text{codeOffset = inject codeOffsetInjection} \\
\text{ $ selb codeOffset' selectForExit selectForExit} \\
\text{ selectForExit = rotmai (if dw==0 then join [density,gxyzd] else density} \\
\) \\
(-32) \ -- \ \text{want sign bit as select mask}
\]

(8) get address for branch and branch hint (this has to be done 15 cycles in advance)

\[
\text{branchAddress = mpyi codeOffset (m1BlockSize wolf)}
\]

(9) separate unoverflowed parts, and use for spline lookups (this can also be done with integer arithmetic, keeping truncating the lowest bytes with a shift)

\[
\begin{align*}
\text{xKer} &= (\text{iInterp input}) \ xWord \ \text{newKwith0Overflow} \\
\text{yKer} &= (\text{iInterp input}) \ yWord \ \text{newKwith0Overflow} \\
\text{zKer} &= (\text{iInterp input}) \ zWord \ \text{newKwith0Overflow}
\end{align*}
\]
Software Pipelining

Coconut Software Pipelining
Software Pipelining

secret sauce in dataflow
Coconut

MultiLoop

hintable computed branch
MASS @ McMaster

- two pipelines (arithmetic | permute, L/S)
- move computation to odd pipeline
  - use permute for lookup
- use quadword rotate/permute for key construction
<table>
<thead>
<tr>
<th>POWER5</th>
<th>Cycles/float</th>
<th>SPU</th>
<th>pipe0</th>
<th>pipe1</th>
<th>/float</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.3</td>
<td>vsacos</td>
<td>8.0</td>
<td>1.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45.6</td>
<td>vsacosh</td>
<td>5.75</td>
<td>1.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23.6</td>
<td>vsasin</td>
<td>8.0</td>
<td>1.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46.6</td>
<td>vsasinh</td>
<td>5.75</td>
<td>1.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31.3</td>
<td>vsatan2</td>
<td>6.5</td>
<td>1.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>53.2</td>
<td>vsatanh</td>
<td>4.5</td>
<td>1.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.7</td>
<td>vscbrt</td>
<td>5.5</td>
<td>1.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.2</td>
<td>vscos</td>
<td>7.0</td>
<td>0.75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.8</td>
<td>vscosh</td>
<td>4.75</td>
<td>1.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>vsdiv</td>
<td>1.25</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.3</td>
<td>vsexp</td>
<td>3.25</td>
<td>1.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.8</td>
<td>vsexpml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.6</td>
<td>vslog</td>
<td>3.5</td>
<td>1.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.4</td>
<td>vslog10</td>
<td>3.5</td>
<td>1.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.3</td>
<td>vslog1p</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22.6</td>
<td>vspow</td>
<td>6.5</td>
<td>2.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.7</td>
<td>vsqdrt</td>
<td>3.5</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.9</td>
<td>vsrcbt</td>
<td>6.5</td>
<td>1.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>vsrec</td>
<td>1.0</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.2</td>
<td>vsrqdrt</td>
<td>2.75</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.9</td>
<td>vsrsqrt</td>
<td>1.5</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17.9</td>
<td>vssin</td>
<td>7.0</td>
<td>0.75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25.4</td>
<td>vssincos</td>
<td>7.0</td>
<td>0.75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.2</td>
<td>vssinh</td>
<td>4.75</td>
<td>1.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.6</td>
<td>vssqrt</td>
<td>1.75</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22.0</td>
<td>vstan</td>
<td>8.25</td>
<td>0.75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.7</td>
<td>vstanh</td>
<td>5.25</td>
<td>1.25</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

? 1.5 ulp?
\[
\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}
\]

\[
tanhLkup = \text{calcBreaks 2 2 3 8.6644}
\]

\[
ftanh(v1,v2) = (\text{selb result1OrOne v1 signBit}, \text{selb result2OrOne v2 signBit})
\]

where

- take the positive part
- \(v1Positive = \text{andc v1 signBit}\)
- \(v2Positive = \text{andc v2 signBit}\)

compare to \(\text{arctanh}(1 - 2^{-24})\) because this is the smallest number which rounds to 1, all higher numbers round to 1, and form a select mask

- \(\text{isBig1} = \text{fcmgt v1Positive (unfloats4 $ 8.6643397420981601947)}\)
- \(\text{isBig2} = \text{fcmgt v2Positive (unfloats4 $ 8.6643397420981601947)}\)

which is applied to the final result:

- \(\text{result1OrOne} = \text{selb result1 (unfloats4 1) isBig1}\)
- \(\text{result2OrOne} = \text{selb result2 (unfloats4 1) isBig2}\)

do parallel lookup for two vector inputs (8 floats) of polynomial coefficients, generated by Maple:

- \(\text{cs} = (\text{coeffs tanhLkup tanhC (v1Positive,v2Positive)})\)

evaluate polynomials using Horner's rule:

- \(\text{result1} = \text{hornerV (map fst cs) v1Positive}\)
- \(\text{result2} = \text{hornerV (map snd cs) v2Positive}\)
$$\sinh x = \frac{e^x - e^{-x}}{2}$$

<table>
<thead>
<tr>
<th>v</th>
<th>bitSwitch</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0ff000000</td>
</tr>
<tr>
<td>1</td>
<td>0x0ff400000</td>
</tr>
<tr>
<td>$\sqrt{2}$</td>
<td>0x0ff7fffff</td>
</tr>
<tr>
<td>$\sqrt{2} + ulp$</td>
<td>0x0ff800000</td>
</tr>
<tr>
<td>$\sqrt{3}$</td>
<td>0x0ffbfffff</td>
</tr>
<tr>
<td>$\sqrt{3} + ulp$</td>
<td>0x0ffc0000</td>
</tr>
<tr>
<td>$2 - ulp$</td>
<td>0x0fffffff</td>
</tr>
<tr>
<td>2</td>
<td>0x10000000</td>
</tr>
<tr>
<td>$2^{19} - 2^{-5}$</td>
<td>0x1fffffffe</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>0x20000000</td>
</tr>
</tbody>
</table>
Results: Speed vs SDK 1.1

acos asin atan cbrt cos exp log log10 log2 pow sin sinh tan tanh
Results: Cycles/Float

Power5/Linux

SPU/SDK 1.1+

SPU/McMaster

Figure 5: Left: measure cycle times per float when processing an array of floats for scalar code and our SPU functions. Right: insertion counts for every odd/even pipeline lines top/bottom open bars further broken into four possible pipeline lines.
The authors would like to thank Robert Einenkel and everyone at IBM who provided access to CELL hardware for testing and Wolfgang Thaller, William Hua, and Wolfram Kahl for their work on the...
ExSSP

Explicitly Staged Software Pipelining

• LoopSpec = codegraph with time-travel
• use min-cut to chop into stages
• glue and list schedule
Figure 6: Transforming parts of a code graph (left) to a minimum cut problem (right). Edges A and D are known to be above and below the stage boundary beforehand; using the minimum cut we decide to put B and C below the cut, because that requires the least number of registers to be alive at the top of the stage.

1. A minimum cut yields two stages with no illegal dependences, and
2. the size of the minimum cut equals the number of values that have to be kept in registers from one stage to the next.

Definition 5.4 Given a code graph $G$, we define the graph $G'$ cut as containing
- a "source" node $s$,
- a "sink" node $t$,
- for every operation in the code graph, an "operation" node,
- for every node in the code graph, a "value" node,
- an edge with infinite weight from each consumer of a value to the producer,
- an edge with weight 1.0 from each producer to the corresponding value node.
6. Numerical Results

We used code graphs defined using COCONUT declarative assembly language for the Cell...

ExSSP vs XLC

(Reported 11 4x unrolled)
Coconut

- a high-productivity environment for verifiable high-performance code generation
- automatic generation can beat hand-tuning