Coconut

COde CONstructing User Tool

Christopher Kumar Anand
Wolfram Kahl
+ McMaster students

Robert Enenkel
William O’Farrell
Award Winner

• 2017 - IBM CAS Faculty Fellow of the Year
• 2018 - IBM CAS Project of the Year
• why?
• highest ranking in internal IBM patent reviews
• dramatic acceleration of ML via instruction set/compiler co-optimization
Students

Stephen Adams
Konrad Anand
Tanya Bouman
Simon Broadhead
Kevin Browne
Shiqi Cao
Kriston Costa
Nathan Cumpson
Curtis d’Alves
Michal Dobrogost
Lucas Dutton
Saeed Jahed
Damith Karunaratne

Umme Salma Gadriwala
Clayton Goes
Gabriel Grant
William Hua
Yumna Irfan
Yusra Irfan
Fletcher Johnson
Wei Li
Stephanie Lin
Nick Mansfield
Mehrdad Mozafari

Adele Olejarz
Jessica Pavlin
Adam Schulz
Anuroop Sharma
Sanvesh Srivastava
Wolfgang Thaller
Gordon Uszkay
Christopher Venantius
Paul Vrbik
Sean Watson
James You
Fei Zhao
We can write safe software.
We can write fast software.

Sometimes we need both.
Performance = Parallelism

Cell BE

- 384-way \(||\)ism
- 4-way SIMD
- 8-way cores
- 6-times unrolling
- double buffering
Roadmap

- **SIMD Parallelism**
  - ✔ extensible DSL captures patterns
  - ½ verification via graph transformation
  - ✔ generated library shipping (Cell BE SDK 3.0)

- **Multi-Core Parallelism**
  - ✔ model on ILP
  - ✔ generation via graph transformation
  - ✔ linear-time verification
  - ✔ run time

- **Distant Parallelism**
  - ∞ verification via model checking
The Road to CoDesign

• Typical Math Function
• Lookups
• SIMD Lookup
• Accurate Table Method
• Exceptions
• New Instructions
• New New Instructions
• Sigmoid
SIMD
weird SIMD
Map Loop Overhead

- one arithmetic instruction
- in/out pointers + induction variable + hint
\[ \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \]

\[ \tanhSPU = \text{use16X2lookup tanhLookup tanhC tanhKeyResult} \]
\[ \text{tanhKeyResult coeffs } v = (\text{key, result}) \]

where

\[ \text{key} = \text{andc v signBit} \]
\[ \text{polyVal} = \text{hornerV coeffs key} \]
\[ \text{isBig} = \text{fcmgt key (unfloats4 tanhTreshold)} \]
\[ \text{resultOrOne} = \text{selb polyVal (unfloats4 1) isBig} \]
\[ \text{result} = \text{selb resultOrOne v signBit} \]
SIMD coefficient lookup

- Sign exponent: s
- Fraction: e
- 4 floats in vector reg

3 msb's of fraction determine 1 of 8 polynomials

Step 1: Rotate 3 msb's into low 5 bits.

Step 2: Shuffle to replicate into other bytes.
SIMD coefficient lookup

Step 3: Select bits, put 0's before, byte index after.

```
000ffe00 000ffe01 000ffe10 000ffe11
```

Step 4: For each k=0,1,...,degree of polynomial
Shuffle to get $a[k] = \text{coeff of } x^k$ for each of 4 polynomials in parallel.

<table>
<thead>
<tr>
<th>p0</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
</tr>
</thead>
<tbody>
<tr>
<td>p4</td>
<td>p5</td>
<td>p6</td>
<td>p7</td>
</tr>
</tbody>
</table>

fff determines which of p0, ..., p7 is selected.
Low Level DSL

- declarative assembly
- support functions
  - polynomial approximation
  - table lookup in registers
  - verify assertions @ compile time
  - compile time computation
- user extensible
6. Cube Root

The rest of this section is an unedited example of literate source code.

Cube Root is defined to be the unique real cube root with the same sign as the input. We calculate it using

\[
(-1)^{\text{sign}} 2^e (1 + \text{frac}) \to (-1)^{\text{sign}} 2^{2r/3} f(1 + \text{frac})
\]

where \(q\) and \(r\) are integers such that

\[
e = 3q + r, \quad 0 \leq r < 3,
\]

and \(f(x)\) is a piecewise order-three polynomial minimax approximation of \((x)^{1/3}\) on the interval \([1, 2]\).

Warning: This function uses divShiftMA for fixed-point division. This is computation is inexact, but cbtAssert tests all the values which can occur as a result of extracting the exponent bits for the input float. If you modify the code you must modify the assertion.

\[
\text{cbtSPU} :: \forall v \cdot (\text{SPU} v, \text{HasJoin} v) \Rightarrow v \to v
\]
\[
\text{cbtSPU} v = \text{assert cbtAssert "cbtSPU" result}
\]

where

Since we process the input in components, we cannot rely on hardware to round denormals to zero, and must detect it ourselves by comparing the biased exponent with zero:

\[
\text{denormal} = \text{ceqi} \text{ exponent} 0
\]

and returning zero in that case

\[
\text{result} = \text{selb unsigned (unwrds4 0) denormal}
\]

We calculate the exponent and polynomial parts separately, and combine them using floating-point multiplication,

\[
\text{unsigned} = \text{fm signCbrtExp evalPoly}
\]

Insert the exponent divided by three into the sign and mantissa of the cube root of the remainder of the exponent division.

\[
\text{signCbrtExp} = \text{selb signMant}
\]
\[
(\\text{join} \, \land \, \text{map} (\lambda f \to f \\text{expDiv3shift16} 7)
\]
\[
[\text{shli, rotqbi}])
\]
\[
(\text{unwrds4} \, \land \, 2 \| 31 - 2 \| 23)
\]

Use the function extractExp to extract the exponent bits, dropping the sign bit, and placing the result into the third byte:

\[
\text{exponent} = \text{extractExp} 3 \, v
\]

\[
\text{coeffs} = \text{lookup8Word} (22, 20) \expCbrt24\text{bits} \, v
\]

Evaluate the polynomial on the fractional part.

\[
\text{evalPoly} = \text{hornerV} \, \text{coeffs frac}
\]
Compile-Time Assertions

cbtracestt :: Bool
cbtracestt = List.\textbf{and}
\[
\begin{array}{l}
  \text{divMod } i \equiv \text{extractDivMod } (\text{approxDiv3 } \& \text{ bias } i) \\
  \mid i \leftarrow [\text{expBias } - 255 \ldots \text{expBias}] \\
\end{array}
\]

where

\text{bias} :: \text{Integer } \rightarrow \text{ Val}
\text{bias } i = \text{unwrds4 } \& \text{ i + expBias}
\text{extractDivMod } w = \text{case bytes } w \text{ of}
\quad \_ : v1 : v2 : \_ \rightarrow (v1 - \text{expBias}, \text{div } v2 64)
\quad \_ \rightarrow \text{error "impossible"}

- simulate special instructions interactively
- verify assertions @ compile time
Multiple Instances

:: DSL

instance

SPUSim/ghci interactive development

codegraph

pretty printer

ExSSP

.visualization

.c

.s
# 25 cycles

loop: fma $55, $47, $47, $12$
shufb $37, $23, $24, $50$
cflts $54, $31, 14$
shufb $38, $25, $26, $50$
fms $31, $10, $41, $42$
hbr jump, $4$

fma $53, $3, $51, $52$
rotqbyi $42, $32, 0$

fma $51, $5, $40, $37$

lqd $32, 0($33)$
f $5, $47, $47$

rotqbyi $34, $33, 8$
selb $50, $16, $48, $18$

frest $37, $55$

fma $52, $46, $38, $45$
rotqbyi $38, $33, 2$
a $45, $54, $9$

shufb $40, $19, $20, $50$

fm $3, $36, $46$

shufb $54, $14, $15, $50$

rotmai $48, $45, -14$

rotqbyi $46, $47, 0$
fi $36, $55, $37$

rotqbyi $45, $45, 0$

fms $47, $7, $41, $31$

shufb $35, $21, $22, $50$

fma $40, $5, $54, $40$

shufb $39, $27, $28, $50$
csflt $41, $48, 0$

rotqbyi $54, $4, 0$

fm $31, $32, $8$

rotqbyi $48, $45, 2$
andbi $37, $49, 128$

rotqbyi $4, $4, $38$
a $33, $33, $30$

shufb $49, $43, $43, $29$
fms $38, $55, $56, $13$

shufb $30, $30, $30, $6$
cgtbi $43, $48, -1$

shufb $48, $44, $44, $17$

fma $40, $5, $40, $35$

stqd $53, 0($34)$
fms $42, $11, $41, $42$
xor $44, $45, $43$
xor $45, $39, $37$

lnop $fma $36, $38, $36, $36$
jump: bi $54
Instruction Scheduling

- Explicitly Staged Software Pipelining (ExSSP)
- Min-Cut to Chop into Stages
- Principled Graph Transformation
- supports control flow (MultiLoop)
Software Pipelining

- hide latency
- same length loop body
MultiLoop

hintable computed branch
Min-Cut Preparation

- cut into stages
- one by one
- minimize live registers
Bad Cut

- c produced in later stage
- c used in earlier stage

known above

known below

23
Transformation

known above

known below

collapse assigned

nodes and edges become nodes

weight 1 production edges

weight $\infty$ consumption edges

weight $\infty$ backwards edges
97% Optimal Schedules

- Cell SPU was a great machine
  - 128 registers
  - two pipelines
  - simple dispatch rules
  - in-order execution
  - complete, public documentation
4X Faster than C
Challenges = Opportunities

- out-of-order execution
- complex dispatch rules
- not enough registers

- developed two other approaches:
  - based on Karger’s min-cut
    - an “approximation algorithm”
  - based on continuous optimization
## Multi-Core = ILP Take 2

<table>
<thead>
<tr>
<th>Instruction Level Parallelism</th>
<th>Multi-Core Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Chip</td>
</tr>
<tr>
<td>Execution Unit</td>
<td>Core</td>
</tr>
<tr>
<td>Load/Store Instruction</td>
<td>DMA</td>
</tr>
<tr>
<td>Arithmetic Instruction</td>
<td>Computational Kernel</td>
</tr>
<tr>
<td>Register</td>
<td>Buffer / Signal</td>
</tr>
</tbody>
</table>
The Catch: Soundness

- on CPUs hardware maintains OOE
  - instructions execute out of order
  - hardware hides this from software
    - ensures order independence
- in our Multi-Core virtual CPU
  - compiler inserts synchronization
    - soundness up to software
    - uses asynchronous communication
Asynchronous

- no locks
- locking is a multi-way operation
- a lock is only local to one core
  - incurs long, unpredictable delays
- use asynchronous messages
- matches efficient hardware
Memory Bound

Comp Bound
Further computation
Async Signals

- SendSignal
- WaitData
- WaitSignal
- sendData
- WaitDMA

No writes to buffer until DMA completion is confirmed.

No reads or writes to buffer until past other operations.

Reorder Window

Hazard

Reorder Window

barrier WaitData
# Multi-Core Language

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computation</strong></td>
<td><em>operation</em> bufferList</td>
</tr>
<tr>
<td><strong>SendData</strong></td>
<td>localBuffer remoteBuffer tags</td>
</tr>
<tr>
<td></td>
<td>start DMA to send local data off core</td>
</tr>
<tr>
<td><strong>WaitData</strong></td>
<td>localBuffer tag</td>
</tr>
<tr>
<td></td>
<td>wait for arrival of DMAed data</td>
</tr>
<tr>
<td><strong>WaitDMA</strong></td>
<td>tag</td>
</tr>
<tr>
<td></td>
<td>wait for locally controlled DMA to complete</td>
</tr>
<tr>
<td><strong>SendSignal</strong></td>
<td>core signal</td>
</tr>
<tr>
<td></td>
<td>send a signal to distant core</td>
</tr>
<tr>
<td><strong>WaitSignal</strong></td>
<td>signal</td>
</tr>
<tr>
<td></td>
<td>wait for signal to arrive</td>
</tr>
</tbody>
</table>
**locally Sequential Program**

<table>
<thead>
<tr>
<th>index</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>long computation</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SendSignal $s \rightarrow c2$</td>
<td>WaitSignal $s$ computation</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>WaitSignal $s$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>SendSignal $s \rightarrow c2$</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- total order for instructions
- easier to think in order
- send precedes wait(s)
NOT sequential

<table>
<thead>
<tr>
<th>index</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SendSignal $s \rightarrow c2$</td>
<td></td>
<td>SendSignal $s \rightarrow c2$</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>second signal overlaps the first, only one registered</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>long computation</td>
<td>WaitSignal $s$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>computation</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>no signal is sent, so the next WaitSignal blocks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>WaitSignal $s$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- can execute out of order
does NOT imply _order independent_

<table>
<thead>
<tr>
<th>index</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>long computation</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SendSignal $s \rightarrow c2$</td>
<td>WaitSignal $s$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>computation using wrong assumptions</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SendSignal $s \rightarrow c2$</td>
<td>WaitSignal $s$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>SendSignal $s \rightarrow c2$</td>
</tr>
</tbody>
</table>
Linear-Time Verification

• must show
  • results are independent of execution order
  • no deadlocks
• need to keep track of all possible states
• linear in time = one-pass verifier
  • constant space
    • i.e. possible states at each instruction
Proof State

- state of buffers (valid, waiting for DMA, ...)
- active signals
- \( \Phi \) follows map \( \Phi_n(c_1, c_2) \)
- records last instruction on core 1 known to complete before the last instruction on core 2 completing before instruction \( n \)
Algorithm

• maintain the state one instruction at a time
  • flag indeterminate states as errors

Proof

• show that any indeterminacy and/or deadlock would have been flagged
Impact

• no parallel debugging !!
• every optimization trick used for ILP can be adapted
• ready for algorithm “skeletons”
  • e.g. map, reduce
• enables optimization for power reduction:
  • replace caching with data in-flight
Memory Lookup

• good
• scales to higher precision
• uses other units

• bad
• doesn’t scale to wider SIMD
Accurate Table Method

- \([\ ] = \) round to floating point
- in each interval find

\[ c = [c] \]

\[ |1/c - [1/c]| \ll \text{ulp} \]

- loose very little precision on range reduction and restoration
Multiplicative Reduction Accurate Table

- unifies AT method
  - log, log1p, ...
  - exp, expm1, ...
- faster

![Bar chart showing performance comparison between SDK and new versions of functions.](image-url)
Better

<table>
<thead>
<tr>
<th>function</th>
<th>max error (ulps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>1.55</td>
</tr>
<tr>
<td>exp2</td>
<td>1.66</td>
</tr>
<tr>
<td>expm1</td>
<td>1.80</td>
</tr>
<tr>
<td>exp2m1</td>
<td>1.29</td>
</tr>
<tr>
<td>log</td>
<td>1.78</td>
</tr>
<tr>
<td>log1p</td>
<td>1.79</td>
</tr>
<tr>
<td>log21p</td>
<td>1.11</td>
</tr>
<tr>
<td>log2</td>
<td>1.00</td>
</tr>
<tr>
<td>acosh</td>
<td>2.01</td>
</tr>
<tr>
<td>asinh</td>
<td>2.20</td>
</tr>
<tr>
<td>atanh</td>
<td>1.46</td>
</tr>
</tbody>
</table>
Exceptions

- special case (e.g., log (-1) )
- extra computation
- branch
- predication
Problems

• hard to schedule

• exceptions slow and don’t scale
New Instructions

- 2 lookups  (LS / odd)
- lookupForReduce
- lookupForRestore
- \( fmax \)  (FPU / VPU)
- 12-bit exponent
- no subnormals
- non-standard exceptions
Exceptions

- all handled in-line
- special lookup values

<table>
<thead>
<tr>
<th>function</th>
<th>normal &gt; 0</th>
<th>subnormal &gt; 0</th>
<th>$+\infty$</th>
<th>$-\infty$</th>
<th>$\pm 0$</th>
<th>$&lt; 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>recip</td>
<td>$2^{-e}/c$, $2^{-e}$</td>
<td>$2^{-e+52}/c$, $(2^{-e}/c)_{\text{saturated}}$</td>
<td>0, 0</td>
<td>0, 0</td>
<td>0, $\pm\infty$</td>
<td>$-2^{-e}/c$, $-2^{-e}/c$</td>
</tr>
<tr>
<td>sqrt</td>
<td>$2^{-e}/c$, $2^{e/2}$</td>
<td>$2^{-e+52}/c$, $2^{e/2}$</td>
<td>0, $\infty$</td>
<td>0, NaN</td>
<td>0, 0</td>
<td>0, NaN</td>
</tr>
<tr>
<td>rsqrt</td>
<td>$2^{-e}/c$, $2^{-e/2}$</td>
<td>$2^{-e+52}/c$, $2^{-e/2}$</td>
<td>0, 0</td>
<td>0, NaN</td>
<td>0, $\infty$</td>
<td>0, NaN</td>
</tr>
<tr>
<td>log2</td>
<td>$2^{-e}/c$, $e + \log_2 c$</td>
<td>$2^{-e+52}/c$, $e-52 + \log_2 c$</td>
<td>0, $\infty$</td>
<td>0, NaN</td>
<td>0, $-\infty$</td>
<td>0, NaN</td>
</tr>
<tr>
<td>exp2</td>
<td>$c$, $2^l \cdot 2^c$</td>
<td>0, 1</td>
<td>0, $\infty$</td>
<td>NaN, 0</td>
<td>0, 1</td>
<td>$c$, $2^{-l} \cdot 2^c$</td>
</tr>
</tbody>
</table>
fmax (extended fma)

- override exceptions
- 1st argument extended
- 12-bit exponent
- no subnormals

<table>
<thead>
<tr>
<th>+_ext</th>
<th>finite</th>
<th>-∞</th>
<th>∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>finite</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>0</td>
</tr>
<tr>
<td>-∞</td>
<td>c</td>
<td>c</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>∞</td>
<td>c</td>
<td>0</td>
<td>c</td>
<td>0</td>
</tr>
<tr>
<td>NaN</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>*_ext</th>
<th>finite</th>
<th>-∞</th>
<th>∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>±0</td>
<td>±0_f</td>
<td>±0_f</td>
<td>±0_f</td>
<td>±0_f</td>
</tr>
<tr>
<td>finite≠0</td>
<td>c</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>-∞</td>
<td>-∞_f</td>
<td>-∞_f</td>
<td>-∞_f</td>
<td>-∞_f</td>
</tr>
<tr>
<td>∞</td>
<td>∞_f</td>
<td>∞_f</td>
<td>∞_f</td>
<td>∞_f</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN_f</td>
<td>NaN_f</td>
<td>NaN_f</td>
<td>NaN_f</td>
</tr>
</tbody>
</table>
Bitflow (log)

+ count leading zeros
+ 12 bit adds
Add Gather

- lots of processors have them
- use values in SIMD slots as indices
- reduces implementation cost
- reduces testing
recip Pre

height (gates in series)

width (gates in parallel)
The graph illustrates the relationship between the number of height (gates in series) and width (gates in parallel) for a recip Reduce scenario. The data shows a distribution where the width increases as the height increases, indicating a trade-off between the two parameters in this context.
recip Restore

height (gates in series)

width (gates in parallel)
Numbers of Gates

- Pre
- ForReduce
- ForRestore

![Bar chart showing numbers of gates for different functions (log, exp, recip, div, sqrt).]
Sigmoid

\[
\frac{1}{1 + e^{-x}}
\]

- used in ML (learning)
- uses \( \text{exp} + \text{recip} \)
- compiler-discoverable optimizations
  - merge \(-1\) and \(\log_2(e)\)
  - \(\text{fm} + \text{fa} \rightarrow \text{fma}\)
  - many times faster
Less than Sum of Parts

• for functions like sigmoid
• faster for all previous reasons
• code is inlinable (no func overhead)
Conclusions

• new instructions
  • much faster
  • not too many gates
  • let’s build it!

• context matters
  • software is still written by people
  • understanding their history helps