Coconut:
**Code Constructing User Tool**

Christopher Kumar Anand
Wolfram Kahl

SQRL
Software Quality Research Laboratory
McMaster University

http://ocalgorithms.com
We can write safe software.

Sometimes we need both.

We can write fast software.
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
Layers of Domain Specific Languages

Haskell

- user code
- distribution patterns
- control flow patterns
- SIMD patterns
- SPU ISA

ExSSP
Higher Order Functions

- examples
  - map
  - zip
- control flow patterns
- matrix multiplication
- SIMD parallelization
- multi-core parallelization

Haskell

ExSSP
map

for (i=0; i<10; i++) {
    out[i] = fun(in[i]);
}

• apply a function to a list
• overhead
  • increment pointer
  • increment pointer
  • increment counter
  • compare counter
  • branch
• one arithmetic instruction
• in/out pointers + induction variable + hint
Low Level DSL

- declarative assembly
- support functions
- polynomial approximation
- table lookup in registers
- verify assertions @ compile time
- compile time computation
- user extensible
Compact Code

tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}

tanhSPU = use16X2lookup tanhLookup tanhC tanhKeyResult

tanhKeyResult coe s v = (key, result)

where
key = andc v signBit
polyVal = hornerV coe s key
isBig = fcmgt key (unfloats4 tanhTreshold)
resultOrOne = selb polyVal (unfloats4 1) isBig
result = selb resultOrOne v signBit
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

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

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

\[e = 3 \quad q + r, \quad 0 \leq r < 3, \quad (4)\]

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

**Warning:** This function uses \texttt{divShiftMA} for fixed-point division. This is computation is inexact, but \texttt{cbtAss} 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.

\begin{verbatim}
cbrtSU :: forall v (SPUType v, HasJoin v) v v
cbrtSU v = assert cbtAss \texttt{"cbrtSU\" 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:

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

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,

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

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

\[\text{signCbtExp} = \text{selb signMant}
   (\text{join \$ map ( f f expDivShift16 7)}
   [\text{shli, rotqbi}])
   (\text{unwrds4 \$ 2 31 - 2 23})\]

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

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

\[\text{coe s} = \text{lookup8Word (22, 20) expCoe s24bits v}\]

Evaluate the polynomial on the fractional part.

\[\text{evalPoly} = \text{hornerV coe s frac}\]
Multiple Instances

:: DSL

instance

SPUSim/ghci
interactive
development

instance
codegraph

pretty
printer

ExSSP

visualization

.c

.s
Figure 5. Scheduled assembly code graph for tanSPU.

Figure 6. tanSPU's # 25 cycles

loop: fma $55$, $47$, $47$, $12$
shufb $37$, $23$, $24$, $50$
cflts $54$, $31$, $14$
shufb $38$, $25$, $26$, $50$
fnms $31$, $10$, $41$, $42$
hbr jump, $4$
fma $53$, $3$, $51$, $52$
rotqbyi $42$, $32$, $0$
fma $51$, $5$, $40$, $37$
lqd $32$, $0$($33)$
fm $5$, $47$, $47$
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 $40$, $19$, $20$, $50$
fms $3$, $36$, $46$
shufb $54$, $14$, $15$, $50$
rotmai $48$, $45$, $-14$
rotqbyi $46$, $47$, $0$
fi $36$, $55$, $37$
rotqbyi $45$, $45$, $0$
fnms $47$, $7$, $41$, $31$
shufb $35$, $21$, $22$, $50$
fma $40$, $5$, $54$, $40$
shufb $39$, $27$, $28$, $50$
cf1t $41$, $48$, $0$
rotqby $54$, $4$, $0$
fms $31$, $32$, $8$
rotqby $48$, $45$, $2$
andbi $37$, $49$, $128$
rotqby $4$, $4$, $38$
a $33$, $33$, $30$
shufb $49$, $43$, $43$, $29$
fnms $38$, $55$, $36$, $13$
shufb $30$, $30$, $30$, $6$
cgtbi $43$, $48$, $-1$
shufb $48$, $44$, $44$, $17$
fma $40$, $5$, $40$, $35$
stdq $53$, $0$($34)$
fnms $42$, $11$, $41$, $42$
xor $44$, $45$, $43$
xor $45$, $39$, $37$
/inop fma $36$, $38$, $36$, $36$
jump: $54$
4X Faster than C

96 cycles

5 cycles
Fine Print on Comparison

- pink bars = C-callable vector SPU MASS
  - e.g., vsexp (in C-ABI library)
  - generated/scheduled by Coconut
  - distributed in SDK 3.0 and with xlC
  - single vector version slightly slower
    - distributed as (cryptic) C
    - e.g. expf4

- blue bars = SimdMath (circa SDK 3.0)
  - developed and distributed in readable C
  - scheduled by spuxlc
Ultimate Assembler

- access to machine instructions
- write patterns in Haskell
- unit test declarative assembly code
- where does performance come from?

$\text{SCIMD} = \text{Single } \textit{Complex} \text{ Instruction } \textit{Merged} \text{ Data}$
Verification

• transform graphs
• break 128-bit register values up
• easy for “pure” SIMD

![Diagram showing verification process involving transform graphs, splitting and merging floating point values.](image)
Difficult for Creative Bit Shuffling

- easiest case: byte rotate by constant

- hardest case: rotate bits by register value
Status - SIMD

- code generation
  - rapid prototyping
  - peak performance
  - lots of work supporting other patterns
  - e.g. interpreting bit operations on floats
- verification
  - equivalent to symbolic execution
  - useful for debugging linear algebra
  - needs more transformation rules
## 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>Register</td>
<td>Buffer / Signal</td>
</tr>
<tr>
<td>Load/Store Instruction</td>
<td>DMA</td>
</tr>
<tr>
<td>Arithmetic Instruction</td>
<td>Computational Kernel</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
Async Signals

No writes to buffer until DMA completion is confirmed.

WaitSignal
SendData

SendSignal

No reads or writes to buffer until past barrier WaitData.

DMA
operations
other

WaitDMA

other
operations

Reorder Window

Reorder Window

Reorder Window

Hazard

WaitData
## Multi-Core Language

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computation</strong> <code>operation bufferList</code></td>
<td>do a computation with local data</td>
</tr>
<tr>
<td><strong>SendData</strong> <code>localBuffer remoteBuffer tags</code></td>
<td>start DMA to send local data off core</td>
</tr>
<tr>
<td><strong>WaitData</strong> <code>localBuffer tag</code></td>
<td>wait for arrival of DMAed data</td>
</tr>
<tr>
<td><strong>WaitDMA</strong> <code>tag</code></td>
<td>wait for locally controlled DMA to complete</td>
</tr>
<tr>
<td><strong>LoadMem</strong> <code>localBuffer remoteBuffer tag</code></td>
<td>start distant data load</td>
</tr>
<tr>
<td><strong>SendSignal</strong> <code>core signal</code></td>
<td>send a signal to distant core</td>
</tr>
<tr>
<td><strong>WaitSignal</strong> <code>signal</code></td>
<td>wait for signal to arrive</td>
</tr>
</tbody>
</table>
Concurrent Control-Flow

1. Scheduling
   • hide latency to eliminate stalls

2. WaitSignal / WaitData
   • stall when necessary, hardware won’t
   • timing less predictable
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</td>
<td>WaitSignal s c2</td>
<td>SendSignal s</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>WaitSignal s computation</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></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</td>
<td>c2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>second signal overlaps the first, only one registered</td>
<td>SendSignal s    c2</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>long computation</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>WaitSignal s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>computation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>no signal is sent, so the next WaitSignal blocks</td>
<td>WaitSignal s</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>SendSignal s c2</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SendSignal s c2</td>
<td>WaitSignal s computation using wrong assumptions</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SendSignal s c2</td>
<td>WaitSignal s</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></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
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
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
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
Transformation

known above

A

B

C

D

collapse assigned

nodes and edges become nodes

weight 1 production edges

weight \( \infty \) consumption edges

weight \( \infty \) backwards edges
Not Just Faster

• why a new algorithm?
  • higher assurance
    • principled graph transformation
  • not just scheduling instructions
• novel control flow
  • via nested control flow graphs
Example 1: MultiLoop
Coconut

• so far
  • functional-assembly programming
  • SIMD++
  • unbeaten scheduler
  • multi-core distribution
  • proof of soundness
• next
  • Multi-Core Patterns
Legal Notices

• Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc., in the United States, other countries, or both.
• IBM is a registered trademark of International Business Machines Corporation in the United States, other countries, or both.
• Other company, product, and service names may be trademarks or service marks of others.
Thanks

Stephen Adams
Kevin Browne
Shiqi Cao
Nathan Cumpson
Saeed Jahed
Damith Karunaratne
Clayton Goes

Gabriel Grant
William Hua
Fletcher Johnson
Wei Li
Nick Mansfield
Mehrdad Mozafari
Adam Schulz

Anuroop Sharma
Sanvesh Srivastava
Wolfgang Thaller
Gordon Uszkay
Christopher Venantius
Paul Vrbik
Fei Zhao

Robert Enenkel

IBM Centre for Advanced Studies, CFI, OIT, NSERC and Apple Canada for research support.