Coconut
Code-Graph-Centred Parallelisation

Christopher K. Anand  Wolfram Kahl

McMaster University, Hamilton, Ontario, Canada

30 October 2008, CASCON CDP Workshop

1 Motivation
2 Code Graphs
3 Software Pipelining by Graph Calculation
4 Lifting ILP to Multi-Core
Hardware Parallelism on the Cell BE

- 1 PPE + 8 SPEs
- DMA independent of execution (double-buffering)

Hardware Parallelism on the Cell SPU

- SIMD instructions act in parallel on “polymorphic” registers:
  128bit = 2 double = 4 float = 4 int = 8 short = 16 byte
- 2 instruction units (arithmetic and data movement) with double issue
- Both units are pipelined:
  Up to 6 instructions “in-flight” simultaneously

Parallelism requires independence!
Dependencies in Real Code

res. lower bounds: [25,19]
fullLength: 93
CPU utilisation ratio: 26.9%

loop:  lqd  $34, 0($6)
hbr  jump, $8
rotdby $5, $6, 8
rotdbi $33, $6, 2
a $6, $6, $7
rotdbi $42, $8, 0
rotdby $7, $7, 0
fm $35, $34, $11
rotdby $8, $9, $33
cflts $35, $35, 14
a $33, $35, $12
lnop
rotdai $36, $33, -14
rotdbi $35, $33, 2
cflti $37, $36, 0
cgtbi $35, $35, -1
xor $33, $33, $35
shufb $35, $35, $35, $32
shufb $33, $33, $33, $20
fnms $34, $14, $37, $34
andbi $35, $35, 128
selb $36, $19, $33, $21
shufb $39, $22, $23, $36
shufb $40, $17, $18, $36
fnms $38, $13, $37, $34
shufb $41, $24, $25, $36
shufb $34, $30, $31, $36
shufb $33, $28, $29, $36
shufb $36, $26, $27, $36
xor $35, $34, $35
fnms $37, $10, $37, $38
fma $38, $37, $37, $15
fm $34, $37, $37
fma $35, $37, $33, $35
frest $33, $38
fma $39, $34, $40, $39
fi $33, $38, $33
fma $41, $34, $39, $41
fnms $39, $38, $33, $16
fma $34, $34, $41, $36
fma $33, $39, $33, $33
fm $33, $33, $37
fma $35, $33, $34, $35
stqd $35, 0($5)
nop $0

jump:  bi $42
Breaking Dependencies by Software Pipelining

<table>
<thead>
<tr>
<th></th>
<th><strong>Hardware</strong> Pipelining</th>
<th><strong>Software</strong> Pipelining</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unit:</strong></td>
<td>instruction</td>
<td>loop body</td>
</tr>
<tr>
<td><strong>Segments:</strong></td>
<td>pipeline stages</td>
<td>body stages</td>
</tr>
<tr>
<td><strong>Independence:</strong></td>
<td>different instructions</td>
<td>different iterations</td>
</tr>
</tbody>
</table>
res. lower bounds: [25,19]
fullLength: 25
CPU utilisation ratio: 100%

loop:
fma      $53, $40, $40, $13
shufb    $55, $24, $25, $45
cfils    $51, $32, 14
shufb    $52, $26, $27, $45
fnms     $32, $11, $34, $35
hbr      jump, $33
fma      $56, $46, $47, $48
rotqbyi  $35, $33, 0
lqd      $33, 0($65)
fm        $55, $40, $40
rotqbyi  $49, $65, 8
selb     $45, $17, $41, $19
frest    $43, $53
fma      $48, $39, $52, $38
rotqbii  $50, $65, 2
a        $52, $51, $10
shufb    $38, $20, $21, $45
fm        $46, $44, $39
rotqbyi  $39, $40, 0
rotmai   $51, $52, $14
shufb    $55, $15, $16, $45
fi       $54, $53, $43
rotqbyi  $44, $52, 0
fnms     $40, $8, $34, $32
shufb    $52, $22, $23, $45
fm        $32, $33, $9
shufb    $43, $28, $29, $45
csfll    $34, $51, 0
rotqbii  $51, $31, 0
fma      $55, $5, $55, $38
rotqbii  $41, $44, 2
andbi    $38, $42, 128
shufb    $42, $36, $36, $30
a        $65, $65, $63
shufb    $63, $63, $63, $7
fnms     $53, $53, $54, $14
rotqbyi  $31, $6, $50
cgtbi    $36, $41, $1
shufb    $41, $37, $37, $18
xor      $38, $43, $38
stqd     $56, 0($49)
fnms     $35, $12, $34, $35
fma      $43, $5, $55, $52
xor      $37, $44, $36
lnop     

jump:
bi       $51
Intermediate Representation: Code Graphs

Hypergraphs are a kind of typed term graphs with:

- node-labels (register or state \textit{types})
- edge-labels (functions, \textit{operation names}, state transformations)
- multiple edge arguments
- multiple edge results

Code graphs are hypergraphs with:

- designated \textbf{input} and \textbf{output} node sequences

Simple Transformation:

- \textbf{rewrite step} is DPO in hypergraph category
- \textbf{rule} is span in code graph category
Data-Flow Code Graphs

Code Graph Characteristics:

- No cycles, no joins
- Pure operations as edge labels
- Multiple arguments, multiple results
- Side effects translated into state arguments and results

Functorial semantics:

- gs-monoidal category with code graphs as morphisms
- relations between input and output tuples
Scheduled Code Graph

fi $14, $12, $10
hbr jump, $20
fm $16, $17, $16
rotqbyi $11, $5, 8
ori $17, $12, 0
rotqbii $10, $5, 2
ori $13, $20, 0
rotqby $20, $6, $10
fm $10, $12, $14
stqd $16, 0($11)
fm $16, $14, $9
lqd $12, 0($5)
a $5, $5, $19
shufb $19, $19, $19, $7
fnms $11, $10, $14, $8
frsqest $10, $12
fma $16, $11, $16, $14
bi $13
Explicitly Staged Scheduling [Thaller 2006]

From sequential composition

\[ G = P; (G_1 \otimes \Pi_{I_2} \times \cdots \times I_k) \]
\[ ; (\Pi_{O_1} \otimes G_2 \otimes \Pi_{I_3} \times \cdots \times I_k) \]
\[ ; \cdots \]
\[ ; (\Pi_{O_1} \times \cdots \times O_{k-1} \otimes G_k) ; Q \]

to parallel composition

\[ G' := R; (G'_1 \otimes \cdots \otimes G'_k) ; S, \]
Control-Flow Rearrangement

- Original view of pipelining transformation: **Recomposition**
- More flexible: **Nested graphs**
- Data-flow graphs inside control-flow graph hyperedges
- Type interaction
- Semantically justified transformations
- Control-flow **rearrangement** for software pipelining

**Software Pipelining by Calculation**
Simple Control-Flow Code Graphs

State transformations as edge labels:
- Deterministic control flow:
  single argument, single result
- Cycles and joins allowed
- Different state types:
  live data sets
- **Nested** data-flow graphs

Standard semantics:
- Kleene categories
Concurrent Control-Flow Code Graphs

Generalised state transformations:
- Multiple results: fork
- Multiple arguments: join
- Best-known model: Petri nets

Standard semantics:
- Traced monoidal categories
- (Iteration theories, flownominals)
Sequentially Decompose Staged Body

Common composition of both code graph layers
Unroll Loop Twice

while \( b \) do \( S \) =
if \( b \) then \((S; \text{while } b \text{ do} S)\)
Distribute S3 into Branches

\[ Q: (R \sqcup S) = Q: R \sqcup Q: S \]
Compose S3 with Branches

Common composition of both code graph layers
Sequentially Decompose S3 after Branches

Functoriality of \( \otimes \); common composition
Un-Distribute $S_3$

\[
Q;R \sqcup Q;S = Q;(R \sqcup S)
\]
Move S2 over Branches Analogously
Move S3 Forward Again
Merge Consecutive Straight-Line Code

[diagram of straight-line code]

[ARGS] [RESULTS]

1

[diagram of straight-line code]

[ARGS] [RESULTS]

1

1

[diagram of straight-line code]

[ARGS] [RESULTS]

1

1
Complete Staging Transformation
Software Pipelining by Calculation

- Simple loops — similar to modulo scheduling
Variant: Staging with Branch Inside Loop Body
Software Pipelining by Calculation

- Simple loops — similar to modulo scheduling
- Multi-way switch inside loop body
Control-Flow Rearrangement: Matrix Mult.
Key to Minimalisation: Impossible Branche Edges
Software Pipelining by Calculation

- Simple loops — similar to modulo scheduling
- Multi-way switch inside loop body
- MatMult nested loop — involves “synthetic loop overhead”
- FFT nested loop — different structure
- General tool-box for loop pattern transformations with correctness proofs
- Automation?
## Using ILP Concepts for Multi-Core

<table>
<thead>
<tr>
<th></th>
<th>ILP</th>
<th>Multi-Core</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Context:</strong></td>
<td>core</td>
<td>chip</td>
</tr>
<tr>
<td><strong>Locality:</strong></td>
<td>execution unit</td>
<td>core</td>
</tr>
<tr>
<td><strong>Computation:</strong></td>
<td>arithmetic instruction</td>
<td>computational kernel</td>
</tr>
<tr>
<td><strong>Data Movement:</strong></td>
<td>load/store instruction</td>
<td>DMA</td>
</tr>
<tr>
<td><strong>Resources:</strong></td>
<td>registers</td>
<td>buffers, signals</td>
</tr>
<tr>
<td><strong>Synchronisation:</strong></td>
<td><strong>hardware</strong> stalling</td>
<td><strong>software</strong> stalling</td>
</tr>
<tr>
<td><strong>Scheduling:</strong></td>
<td><strong>avoids</strong> stalling</td>
<td><strong>avoids</strong> stalling</td>
</tr>
</tbody>
</table>
“∗” and “∗+” edges: block multiplication, resp. multiply-add:

- control-flow graph representing the loop structure
- software-pipelined submatrix multiplication program
- loop body and prologue edges labelled with pure data-flow graphs
Pure Data-Flow Graphs

- High-level semantic view of block computation
- Core assignment and schedule maximise data locality
- Similar to instruction/unit selection on RISC
edges and nodes assigned to cores

explicit mv edges — identity semantics

scheduling can use approximation of DMA latencies

dependencies still abstract
**Concurrent Data-Flow Graphs**

- Communication primitives and dependencies explicit
- **Real** data flow (DMA) hidden
- Can be scheduled as for pipelined RISC
Summary and Outlook

- **Code graphs**: uniform hypergraph syntax

- **Graph algebra and functorial semantics** essential for correctness and expressive power

- **Different combinations of control and data flow**

- **Nested graphs** to deal with complex control flow patterns at the outer level

- **Nesting again** for concurrency and distribution (PPU/SPU)

- Analogy to instruction-level parallelism carries over to multicore setting!