Compiler Technology for CGRAs

Bjorn De Sutter
Ghent University
Overview

• What makes compilation for CGRAs different?

• What CGRA code generation techniques exist?

• What is not automated?
CGRA vs VLIW
VLIW code scheduling

cycle 0
- ID: LD, LD
- EX: IS 2, IS 3
- MEM: ADD
- WB: IS 5, IS 6, IS 7

cycle 1
- ID: IS 0, IS 1
- EX: IS 2, IS 3
- MEM: CMP
- WB: IS 5, IS 6, IS 7

cycle 2
- ID: MUL, IS 1
- EX: IS 2, IS 3
- MEM: LD
- WB: IS 5, IS 6, IS 7

cycle 3
- ID: LD, LD
- EX: ADD, SUB
- MEM: MUL
- WB: IS 5, IS 6, IS 7

cycle 4
- ID: MUL, ST
- EX: IS 2, IS 3
- MEM: CMP
- WB: IS 5, IS 6, IS 7

cycle 5
- ID: CMP
- EX: IS 5, IS 6, IS 7
- MEM: CMP
- WB: IS 5, IS 6, IS 7

RF 0
- Time: cycle 0, cycle 1, cycle 2, cycle 3, cycle 4, cycle 5
- Operations: ID, EX, MEM, WB

RF 1
CGRA code scheduling

cycle 0

cycle 1

cycle 2

cycle 3

cycle 4

cycle 5

time

LD LD IS 2 IS 3

ADD IS 5 IS 6 IS 7

CMP IS 5 IS 6 IS 7

MUL IS 5 IS 6 IS 7

ST IS 0 IS 2 IS 3

RF 0

RF 1
Overview

• What makes compilation for CGRAs different

• What CGRA code generation techniques exist?

• What is not automated?
CGRA scheduling = graph mapping

data dependence graph models code

modulo routing resource graph models the whole architecture at some initiation interval
Functional Unit MRRG subgraph

cycle n

pred_in -> src1 -> T1 -> pred_dst1

pred_in -> src2 -> T2 -> pred_dst2

T1 -> T3

pred_dst2 -> dst

cycle n+1

pred_in -> src1 -> T1

pred_in -> src2 -> T2

T1 -> T3

pred_dst1 -> pred_dst2 -> dst

RF
Functional Unit MRRG subgraph

pred_in

src1

src2

dst

cycle n

cycle n+1
Functional Unit MRRG subgraph

- appropriate connections model latency
- internal nodes and edges model data passing and additional routing freedom
- more src inputs are possible
- all variations use same basics
- adapting the subgraph implements differences
- without requiring changes to mapping algorithm itself
- inherently retargetable
Register File MRRG subgraph

- model different delays:
  - delay 0 (forwarding)
Register File MRRG subgraph

- model different delays:
  - delay 1
- model rotation
- non-rotating
- model different delays:
  - delay 1
- model rotation
- rotating
Register File MRRG subgraph

- appropriate connections model latency
- edges model non-rotating vs. rotating RFs
- no explicit register allocation performed
- when a dependency is mapped onto a net through a RF, the value is allocated in the corresponding register
- if not, the value is not allocated to a RF
Muxes & pipelining register MRRG subgraphs

cycle n

cycle n+1

cycle n+2

cycle n

cycle n+1

cycle n+2

cycle n

cycle n+1

cycle n+2
Delayed muxes & bus MRRG subgraphs

cycle n

cycle n+1

cycle n+2

delay 1
CGRA scheduling = graph mapping

data dependence graph models code

to:

modulo routing resource graph models the whole architecture at some initiation interval

inherently retargetable easily model heterogeneous architectures
Data dependence graph

Sequential C code:

\[
\begin{align*}
  t_1 & \leftarrow x \\
  t_2 & \leftarrow A \times t_1 \\
  t_2 & \leftarrow t_2 + B \\
  t_2 & \leftarrow t_2 \times t_1 \\
  y & \leftarrow t_2 + C
\end{align*}
\]

Parallel data dependence graph:

- if-conversion
- predication
- predicate speculation
- hyperblock formation
- inspector-executor loops

But limited:
- single loop exit
- not too many predicates
CGRA scheduling = graph mapping

data dependence graph models code

modulo routing resource graph models the whole architecture at some initiation interval

minimal initiation interval depends on
- length of recurrence cycle: 2
- number of resources: 8/8 = 1

maximum utilization = 1/2
Generate ILP

- loop unrolling
- loop fusion
- loop interchange
CGRA scheduling = graph mapping

1. list scheduling
   - schedule nodes one by one
   - placement of nodes determined by router
   - resource cost models potential later need for resource
   - most constrained nodes scheduled first, almost no backtracking

2. simulated annealing
   - initial schedule overuses resources
   - try many small changes to schedule until overuse is gone
   - simpler cost function, but it includes overuse
Results multimedia processing

<table>
<thead>
<tr>
<th>Application</th>
<th># loops</th>
<th>minimal II</th>
<th>simulated annealing</th>
<th>list scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>II</td>
<td>time (s)</td>
</tr>
<tr>
<td>3D rendering</td>
<td>80</td>
<td>290</td>
<td>305</td>
<td>44K</td>
</tr>
<tr>
<td>AAC decoder</td>
<td>35</td>
<td>123</td>
<td>141</td>
<td>35K</td>
</tr>
<tr>
<td>AMR-WB+ decoder</td>
<td>44</td>
<td>230</td>
<td>233</td>
<td>20K</td>
</tr>
<tr>
<td>eAAC+ decoder</td>
<td>47</td>
<td>204</td>
<td>217</td>
<td>25K</td>
</tr>
<tr>
<td>H.264 decoder</td>
<td>61</td>
<td>387</td>
<td>470</td>
<td>57K</td>
</tr>
<tr>
<td>mp3 decoder</td>
<td>8</td>
<td>44</td>
<td>45</td>
<td>17K</td>
</tr>
<tr>
<td>MPEG surround decoder</td>
<td>115</td>
<td>522</td>
<td>554</td>
<td>65K</td>
</tr>
<tr>
<td><strong>overall</strong></td>
<td><strong>390</strong></td>
<td><strong>1800</strong></td>
<td><strong>1965</strong></td>
<td><strong>264K</strong></td>
</tr>
</tbody>
</table>
Some more results WLAN

<table>
<thead>
<tr>
<th>Loop</th>
<th>#ops</th>
<th>ResMII</th>
<th>RecMII</th>
<th>II</th>
<th>IPC</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DemapQAM64</td>
<td>55</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>9.2</td>
<td>368.1</td>
</tr>
<tr>
<td>FFT (64point)</td>
<td>123</td>
<td>8</td>
<td>4</td>
<td>10</td>
<td>12.3</td>
<td>576.7</td>
</tr>
<tr>
<td>FFTRadix8</td>
<td>122</td>
<td>8</td>
<td>3</td>
<td>10</td>
<td>12.2</td>
<td>468.3</td>
</tr>
<tr>
<td>comp</td>
<td>54</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>10.8</td>
<td>270.6</td>
</tr>
<tr>
<td>DataShuffle</td>
<td>153</td>
<td>14</td>
<td>3</td>
<td>14</td>
<td>10.9</td>
<td>908.5</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11.1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pipelined</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>non-pipelined</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loop</th>
<th>#ops</th>
<th>RecMII</th>
<th>II</th>
<th>IPC</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DemapQAM64</td>
<td>1</td>
<td>6</td>
<td>9.2</td>
<td>110.8</td>
<td></td>
</tr>
<tr>
<td>FFT (64point)</td>
<td>2</td>
<td>12</td>
<td>10.3</td>
<td>256.0</td>
<td></td>
</tr>
<tr>
<td>FFTRadix8</td>
<td>1</td>
<td>12</td>
<td>10.2</td>
<td>195.6</td>
<td></td>
</tr>
<tr>
<td>comp</td>
<td>2</td>
<td>5</td>
<td>10.8</td>
<td>77.3</td>
<td></td>
</tr>
<tr>
<td>DataShuffle</td>
<td>1</td>
<td>16</td>
<td>9.6</td>
<td>878.0</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td>10.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Overview

• What makes compilation for CGRAs different

• What CGRA code generation techniques exist?

• What is not automated?
Data dependence graph

sequential C code

t1 ← x
t2 ← A × t1
t2 ← t2 + B
t2 ← t2 × t1
y ← t2 + C

parallel data dependence graph

• if-conversion
• predication
• predicate speculation
• hyperblock formation

• inspector-executor loops

• but limited
  - single loop exit
  - not too many predicates
Generate ILP

- loop unrolling
- loop fusion
- loop interchange
1. list scheduling
   - schedule nodes one by one
   - placement of nodes determined by router
   - resource cost models potential later need for resource
   - most constrained nodes scheduled first, almost no backtracking

2. simulated annealing
   - initial schedule overuses resources
   - try many small changes to schedule until overuse is gone
   - simpler cost function, but it includes overuse
(a) original 15-tap FIR filter

const short c[15] = {-32, ..., 1216};

for (i = 0; i < nr; i++) {
    for(value = 0, j = 0; j < 15; j++)
        value += x[i+j]*c[j];
    r[i] = value;
}

(b) filter after loop unrolling, with hard-coded constants

const short c00 = -32, ..., c14 = 1216;

for (i = 0; i < nr; i++)
    r[i] = x[i+0]*c00 + x[i+1]*c01 + ... + x[i+14]*c14;
Example: FIR filter

(c) filter after redundant memory accesses are eliminated

```c
int i, value, d0, ..., d14;
const short c00 = -32, ..., c14 = 1216;

for (i = 0; i < nr+15; i++) {
    d0 = d1;
    ...;
    d13 = d14;
    d14 = x[i];
    value = c00 * d0 + c01 * d1 + ... + c14 * d14;
    if (i >= 14) r[i-14] = value;
}
```

<table>
<thead>
<tr>
<th>loop</th>
<th>4x4 ADRES</th>
<th>TI C64+</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cycles</td>
<td>mem accesses</td>
</tr>
<tr>
<td>(a)</td>
<td>11828</td>
<td>6221</td>
</tr>
<tr>
<td>(b)</td>
<td>1247</td>
<td>3203</td>
</tr>
<tr>
<td>(c)</td>
<td>664</td>
<td>422</td>
</tr>
</tbody>
</table>
Further reading

- **hyperblock formation – predication**

- **list scheduling for CGRAs**

- **simulated-annealing based scheduling for CGRAs**
Further reading

• register allocation for CGRAs

• memory bank aware code generation for CGRAs

• Overview of CGRA techniques and issues

• Results on real-world applications on ADRES using simulated-annealing