Lecture 21: High-level Synthesis (2)

Deming Chen
Outline

- Binding for DFG
  - Leftedge algorithm
  - Network flow algorithm
- Binding to reduce interconnects
- Simultaneous scheduling and binding
- A case study: FCUDA
Left edge algorithm for binding

`LEFT_EDGE(I) {`
```
  Sort elements of I in a list L in ascending order of l_i; /* left edge*/
  c = 0;
  while (some variable/operation has not been bound) do {
    S = 0; r = 0;
    while (there is an element in L whose left edge coordinate is larger than r) do {
      s = first element in the list L with l_s > r;
      S = S \cup \{s\};
      r = r_s;
      delete s from L;
    }
    c = c + 1;
    bind S into one resource;
  }
}
```

Optimal for interval graphs - An interval graph is a graph whose vertices can be put in one-to-one correspondence with a set of intervals, so that two vertices are adjacent if and only if the corresponding intervals intersect
Example

a) Interval graph
b) Sorted intervals
c) Colored graph
d) Packed intervals

The coloring problem:

- Coloring the vertices of a graph such that no two adjacent vertices share the same color
- The problem of finding a minimum coloring of a graph is NP-hard for general graph
A DFG, \( G = (V, A) \)

\( V = \{v_1, v_2, \ldots, v_x\} \)

\( A = \{a_1, a_2, \ldots, a_y\} \)

\( a_i = \{v_m, v_n\} : \) data edge from \( v_m \) to \( v_n \)

\textit{life time} of a variable:
\begin{itemize}
  \item \textit{birth time} to \textit{death time},
  \item \([BT, DT]\)
\end{itemize}

Life time of variable \( b \) is \([2, 3]\)

Life time of a functional unit is similarly defined
Compatibility (Comparability) Graph for functional unit

- Given a DFG $G$, build a compatibility graph, $G_c = (V_c, A_c)$
- $V_c$: all the operations in $G$ (For FU binding)
- $A_c$: all the edges between compatible operations in $V_c$
  - $a_c = (v_i, v_j)$ iff $T_D(v_i) < T_B(v_j)$
- $W_{ij}$: weight of $a_c$, the cost of binding $v_i$ and $v_j$ into a single FU
  - switching activity

---

$G$ (additions)

<table>
<thead>
<tr>
<th>Op</th>
<th>$T_B$</th>
<th>$T_D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
Compatibility (Comparability) Graph for variables

A compatibility graph, $G_c = (V_c, A_c)$ for $G$

$V_c$: all the variables in $G$  (For register binding)

$A_c$: all the edges between compatible variables in $V_c$, e.g., $a_c = (v_i, v_j)$ iff $DT(v_i) < BT(v_j)$

$w_{ij}$: weight of $a_c$, the cost of binding $v_i$ and $v_j$ into a single register

```
<table>
<thead>
<tr>
<th>var</th>
<th>BT</th>
<th>DT</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>d</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>f</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
```
Cost function to calculate $W_{ij}$

- Use the number of multiplexers to reflect the connection requirement
  - the more MUXes, the more connections, and the larger cost
- A MUX occurs in two situations:
  - before a port $p$ of a functional unit
  - before a register $R$
Cost function to calculate $W_{ij}$

Cost of binding $v_i$ and $v_j$ together (Case 2):

$$W_{ij} = -\left( N_{mux} + \partial \cdot T_{rf} + \beta \cdot T_{fu} \right) - L$$

- $N_{mux}$: the number of MUXes saved (or MUXes wasted) by binding $v_i$ and $v_j$ into a single register (Case 2) than not binding them into a single register (Case 1)
- $T_{rf}$: the total number of connections between register $R$ and the successor functional units
- $T_{fu}$: the total number of successor functional units involved during this tempted binding of $v_i$ and $v_j$
- $L$: a large positive constant
Obtaining the register binding solution

- Solution obtaining procedure
  - Build a network flow graph (a decent algorithm course would cover this topic)
  - Find the minimum cost $k$-flow in the network
  - Find all the edges with unit flow
    - edge connecting the vertices of variables $v_i$ and $v_j$: $v_i$ and $v_j$ should be bound together
    - edge connecting the vertices of a single variable $v_i$: $v_i$ occupies a register just by itself

- Features of the solution
  - Guarantee of optimal number of registers
  - Minimization of the numbers of MUXes and MUX inputs
Port assignment

- Lemma [B. Pangrle, TCAD91]
  Given a functional unit $u$, the minimum-connectivity port assignment for $u$ can be obtained by minimizing the number of input-registers that are connected to both ports of $u$

- Two observations
  - An optimal solution can be obtained for $u$ if there are no input-registers that drive both ports of $u$
  - There are situations where an input-register has to drive both ports of $u$. It can still be optimal
Case study with Operand Swapping

Before Swapping

Case 1

\[ v_1 + x_1, v_2 + x_2 \]

\[ v_1 + x_1, v_2 + x_2 \]

Case 2

\[ a + b, v + c, v + a \]

\[ a + b, v + c, v + a \]

After Swapping

Case 1

Swap \( v_1 \) and \( x_1 \)

\[ v_1 + x_1, v_2 + x_2 \]

\[ v_1 + x_1, v_2 + x_2 \]

Case 2

Swap \( c \) itself

\[ a + b, v + c, v + a \]

\[ a + b, v + c, v + a \]
Find all the registers that are driving the two ports of the adder
For each $R_i$, analyze the variables in $R_i$
Swap operands that pair with the dual-port variables in $R_i$
Try each such dual-port variables
## Experimental results – benchmark data

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>aircraft</td>
<td>2283</td>
<td>4680</td>
<td>528</td>
</tr>
<tr>
<td>chem</td>
<td>347</td>
<td>731</td>
<td>112</td>
</tr>
<tr>
<td>dir</td>
<td>148</td>
<td>314</td>
<td>50</td>
</tr>
<tr>
<td>feig_dct</td>
<td>548</td>
<td>1899</td>
<td>282</td>
</tr>
<tr>
<td>honda</td>
<td>97</td>
<td>214</td>
<td>34</td>
</tr>
<tr>
<td>mcm</td>
<td>94</td>
<td>252</td>
<td>54</td>
</tr>
<tr>
<td>pr</td>
<td>42</td>
<td>134</td>
<td>31</td>
</tr>
<tr>
<td>steam_u4mul</td>
<td>220</td>
<td>472</td>
<td>55</td>
</tr>
<tr>
<td>u5ml_12</td>
<td>547</td>
<td>1144</td>
<td>163</td>
</tr>
<tr>
<td>wang</td>
<td>48</td>
<td>134</td>
<td>30</td>
</tr>
</tbody>
</table>
## Experimental results – comparison

### Large designs (with nodes > 200)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Aspdac’04 with pa</th>
<th>Aspdac’04 w/o pa</th>
<th>bipartite w/o pa</th>
<th>leftedge w/o pa</th>
</tr>
</thead>
<tbody>
<tr>
<td>aircraft</td>
<td>1</td>
<td>1.04</td>
<td>1.10</td>
<td>1.51</td>
</tr>
<tr>
<td>chem</td>
<td>1</td>
<td>1.03</td>
<td>1.16</td>
<td>1.54</td>
</tr>
<tr>
<td>feig_dct</td>
<td>1</td>
<td>1.06</td>
<td>1.09</td>
<td>1.43</td>
</tr>
<tr>
<td>steam_u4mul</td>
<td>1</td>
<td>1.02</td>
<td>1.07</td>
<td>1.66</td>
</tr>
<tr>
<td>u5ml_12</td>
<td>1</td>
<td>1.03</td>
<td>1.10</td>
<td>1.64</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>1</strong></td>
<td><strong>1.04</strong></td>
<td><strong>1.10</strong></td>
<td><strong>1.55</strong></td>
</tr>
</tbody>
</table>

### Small designs (with nodes <= 200)

<table>
<thead>
<tr>
<th></th>
<th>Aspdac’04 with pa</th>
<th>Aspdac’04 w/o pa</th>
<th>bipartite w/o pa</th>
<th>leftedge w/o pa</th>
</tr>
</thead>
<tbody>
<tr>
<td>dir</td>
<td>1</td>
<td>1.04</td>
<td>0.97</td>
<td>1.43</td>
</tr>
<tr>
<td>honda</td>
<td>1</td>
<td>1.03</td>
<td>1.13</td>
<td>1.47</td>
</tr>
<tr>
<td>mcm</td>
<td>1</td>
<td>1.04</td>
<td>1.09</td>
<td>1.40</td>
</tr>
<tr>
<td>pr</td>
<td>1</td>
<td>1.05</td>
<td>1.00</td>
<td>1.12</td>
</tr>
<tr>
<td>wang</td>
<td>1</td>
<td>1.03</td>
<td>0.99</td>
<td>1.22</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>1</strong></td>
<td><strong>1.04</strong></td>
<td><strong>1.04</strong></td>
<td><strong>1.33</strong></td>
</tr>
</tbody>
</table>

**Overall**

<table>
<thead>
<tr>
<th></th>
<th>Aspdac’04 with pa</th>
<th>Aspdac’04 w/o pa</th>
<th>bipartite w/o pa</th>
<th>leftedge w/o pa</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Overall</strong></td>
<td><strong>1</strong></td>
<td><strong>1.04</strong></td>
<td><strong>1.07</strong></td>
<td><strong>1.44</strong></td>
</tr>
</tbody>
</table>
Simultaneous binding and scheduling

- Using FPGA as a case study [D. Chen, *ISLPED’03*]
- FPGA Characteristics
  - abundance of distributed registers
  - no efficient support for wide MUXes
  - smaller numbers of functional units and/or registers may not correspond to a smaller area or power
  - need to explore a large solution space considering FU binding, scheduling, register binding, and MUX generation simultaneously
- Simulated Annealing method is adopted in this work
Simulated Annealing

Local Search

Intro. VLSI System Design
Generic simulated annealing algorithm

1. Get an initial solution S

2. Get an initial temperature T\(>0\)

3. While not yet “frozen” do the following:
   3.1 For \(1 \leq i \leq L\), do the following:
      3.1.1 Pick a random neighbor S’ of S
      3.1.2 Let \(\Delta = \text{cost}(s’) - \text{cost}(s)\)
      3.1.3 If \(\Delta \leq 0\) (downhill move),
           Set S=S’
      3.1.4 If \(\Delta > 0\) (uphill move)
           set S=S’ with probability \(e^{-\Delta/T}\)
   3.2 Set T \(<= rT\) (reduce temperature)

4. Return S
Simultaneous binding and scheduling for low power

- Five types of moves of different binding gradually reduce the overall cost, i.e., estimated power consumption
  - *Reselect*: selects another FU of the same functionality but different implementation for a binding.
  - *Swap*: swaps two bindings of the same functionality but different implementations.
  - *Merge*: merges two bindings into one, i.e., the operations bound to the two FUs are combined into one FU.
  - *Split*: splits one binding into two. Reverse of *Merge*.
  - *Mix*: selects two bindings, *merge* them, sort the merged operations according to their *slack*, and then *split* the operations.

- After each move, a *list scheduling* is called to verify the total latency. Then, the *left edge* algorithm is used for register binding followed by MUX generation.
High-level power model

- Both *dynamic* and *static* power for various FPGA components are considered
- A mixed-level FPGA power model
  - use pre-characterization-based macro-modeling to capture the average switching power per access of the LUT and register
  - use switch level calculation for interconnects

\[
P_{\text{Dynamic}} = S(P_{LUT} + P_{REG} + P_{LW} + P_{GW})
\]

\[
P_{\text{Static}} = P_{\text{Idle\_LUT}} + P_{\text{Static\_LB}} + P_{\text{Static\_GB}}
\]

- \( S \): estimated switching activity
- \( P_{LUT}, P_{REG} \): macro-modeling-based power estimation
- \( P_{LW}, P_{GW} \): using rent’s rule to estimate routing wire usage

Intro. VLSI System Design
One example for switching activity estimation – operation scheduling

An Example *DFG.*

*Schedule 1*

Toggle Count:

\[5 + 4 = 9\]

*Schedule 2*

Toggle Count:

\[5 + 5 = 10\]
Example cont’ – two FUs

Scheduled $DFG$

Binding 1

Toggle Count: 5

Functional Unit

Toggle Count: 4

Binding 2
Synthesis Flow - LOPASS

1. Function Unit Binding
2. Scheduling
3. Register Binding
4. MUX Generation
5. high-level Power Estimation
6. Power
7. Simulated-Annealing Engine
8. Power Improvement?
   - Yes
   - No

- Intro. VLSI System Design
- Reselect
- Merge
- Swap
- Mix
- Split
- Intro. VLSI System Design
### Experimental results – Binding and Scheduling Results

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dir</td>
<td>152</td>
<td>9</td>
<td>16</td>
<td>32</td>
<td>75</td>
<td>5</td>
<td>7</td>
<td>32</td>
<td>55</td>
</tr>
<tr>
<td>honda</td>
<td>101</td>
<td>9</td>
<td>14</td>
<td>29</td>
<td>55</td>
<td>3</td>
<td>6</td>
<td>28</td>
<td>40</td>
</tr>
<tr>
<td>mcm</td>
<td>98</td>
<td>23</td>
<td>6</td>
<td>36</td>
<td>118</td>
<td>4</td>
<td>3</td>
<td>35</td>
<td>59</td>
</tr>
<tr>
<td>pr</td>
<td>46</td>
<td>13</td>
<td>8</td>
<td>24</td>
<td>33</td>
<td>2</td>
<td>2</td>
<td>23</td>
<td>34</td>
</tr>
<tr>
<td>wang</td>
<td>52</td>
<td>5</td>
<td>8</td>
<td>29</td>
<td>29</td>
<td>2</td>
<td>2</td>
<td>28</td>
<td>30</td>
</tr>
</tbody>
</table>
## Experimental results – LUT Number, Delay and Power Comparison

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Synopsys Behavioral Compiler</th>
<th>LOPASS</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>dir</td>
<td>LUT No.</td>
<td>Delay (ns)</td>
<td>Power (w)</td>
</tr>
<tr>
<td>dir</td>
<td>18658</td>
<td>54.7</td>
<td>2.55</td>
</tr>
<tr>
<td>honda</td>
<td>16426</td>
<td>43.4</td>
<td>1.85</td>
</tr>
<tr>
<td>mcm</td>
<td>15991</td>
<td>46.8</td>
<td>1.97</td>
</tr>
<tr>
<td>pr</td>
<td>7663</td>
<td>30.2</td>
<td>0.72</td>
</tr>
<tr>
<td>wang</td>
<td>9057</td>
<td>35.7</td>
<td>0.83</td>
</tr>
<tr>
<td>Ave.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FCUDA: CUDA-to-FPGA

- Use CUDA code in tandem with HLS to:
  - enable high abstraction FPGA programming
  - leverage different types of parallelism during hardware generation
  - tradeoff between cycles, frequency and concurrency

- CUDA: C-based parallel programming model for GPUs
  - Concise expression of parallelism
  - Large amount of existing applications
  - Good model for providing common programming interface for kernel acceleration on GPUs & FPGAs

- AutoPilot: Advanced HLS tool (from AutoESL, now Xilinx)
  - Automatic fine-grained parallelism extraction
  - Annotation-driven coarse-grained parallelism extraction
FCUDA-I Flow (SASP’09)

- **CUDA Code**
- **FCUDA Annotation**
- **Annotated CUDA**
- **FCUDA Translation**
- **AutoPilot C Code**
- **HLS & Logic Synthesis**
- **FPGA Bitfile**

**Basic**

**Programmer annotates CUDA code with pragmas to guide FCUDA translation**

**Translate CUDA coarse grained parallelism into parallel AutoPilot C tasks**

**Transform parallel C tasks into parallel RTL cores (AutoPilot) and synthesize RTL onto FPGA reconfigurable fabric (Xilinx toolset)**
Optimized FCUDA-II Flow (FCCM’11)

1. Characterize multilevel parallelism effect on cycles/frequency/resource
   - Data access parallelism
   - Loop Iteration parallelism
   - Task parallelism

2. Explore parallelism types space and identify best combination of different parallelism types for performance

3. Translate different types of parallelism to AutoPilot parallel source code

Intro. VLSI System Design
FCUDA Implementation Overview

- The FCUDA translation consists of two main stages:
  - FCUDA Front-End stage:
    - Convert logical threads into explicit thread-loops
    - Based on the MCUDA framework (John Stratton et al., “MCUDA: An efficient implementation of CUDA kernels on multi-core CPUs”)
  - FCUDA Back-End stage:
    - Extract coarse grained parallelism at the thread-block level

- Implemented with the Cetus compiler infrastructure
Task Synchronization

Pragma-driven source code transformation
- Sequential: temporally interleave compute & transfer
- Ping-Pong: temporally overlap compute & transfer
  • Higher BRAM cost

Sequential scheme

Ping-Pong scheme
Basic FCUDA Translation Overview

- Extract coarse-grained parallelism at the thread-block (TB) level
- 1 TB ↔ 1 core
- TB threads are folded into thread-loops (i.e. 1 core = 1 thread)
- Each thread-block array is allocated one on-chip memory
- Replicate cores until FPGA’s compute or store logic capacity is met
Optimized FCUDA Translation Overview

- Consider multiple granularities of parallelism:
  - On-chip memory level parallelism
  - Thread level parallelism
  - Core level parallelism
  - Core-Cluster level parallelism
- Frequency vs. cycles vs. concurrency tradeoffs
- Efficient estimation and exploration techniques to look for best performing implementation
Design Space

Clusters/device
(1 cluster → 1 tile)

Cores/cluster

Threads/core

Partitions/array

Intro. VLSI System Design
ML-GPS Engine Overview

- Engines:
  - SST: Source-to-source Transformation Engine
  - DSE: Design Space Exploration Engine
  - HLS: High Level Synthesis Engine (AutoPilot)

- Cycle count
  - HLS estimate
- Thread count
  - Resource estimation model
- Frequency
  - Period estimation model
2D Binary Search Algorithm

- Observations:
  - For a fixed unroll degree, latency curve is convex w.r.t. array partitioning
  - Latency curve is convex w.r.t. unroll degree under best partitioning degree

- Use binary search to minimize HLS invocation count

- For each point $p$ in unroll-partition space
  - Estimate latency for $p$ and $p+1$
  - Decide which sub-space to focus into
  - Search complexity $\log|U| \times \log|P|$
DSE Engine Evaluation

- matrixmul space
- fwt2 space

Latency (ms) vs LUT number for both matrixmul and fwt2 space.
### CUDA Kernels

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Data Dimensions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiply (matmul)</td>
<td>4096x4096</td>
<td>Computes multiplication of two arrays (used in many applications)</td>
</tr>
<tr>
<td>Coulombic Potential (cp)</td>
<td>4000 atoms, 512x512 grid</td>
<td>Computation of electrostatic potential in a volume containing charged atoms</td>
</tr>
<tr>
<td>Fast Walsh Transform (fwt1)</td>
<td>32 Million element vector</td>
<td>Walsh-Hadamard transform is a generalized Fourier transformation used in various engineering applications</td>
</tr>
<tr>
<td>Fast Walsh Transform (fwt2)</td>
<td>120K points</td>
<td>1D DWT for Haar wavelets and signals</td>
</tr>
<tr>
<td>Discreet Wavelet Transform (dwt)</td>
<td>120K points</td>
<td>1D DWT for Haar wavelets and signals</td>
</tr>
</tbody>
</table>
Multi- vs. Single Granularity

- Up to 7X (for mm_16)
- No array partitioning applied to fwt1
  - Complex array access patterns prevent static partitioning
FPGA vs. GPU – Latency

- Nvidia G92 (65nm)
- Xilinx SX240T Virtex-5 (65nm)

Latency (normalized over GPU)

Latency

- GPU
- FPGA (8GB/s)
- FPGA (16GB/s)
- FPGA (64GB/s)
FPGA vs. GPU – Energy Consumption

- Nvidia G92: 170W
- Xilinx Power Estimator (XPE) tool

Energy (normalized over GPU)

- mm_32
- mm_16
- fwt2_32
- fwt2_16
- fwt1_32
- fwt1_16
- cp_32
- cp_16
- dwt_32
- dwt_16
CFUDA is open source now!

http://dchen.ece.illinois.edu/tools.html

FCUDA

A source-to-source transformation framework that can take CUDA code, generate functionally equivalent synthesizable C code, and map to an FPGA implementation using high-level synthesis for high performance and energy-efficient reconfigurable computation.

Download FCUDA

Conclusions

- DFG binding is relatively easy
  - Can be well formulated through network flow method
  - Coming up with the accurate cost estimation is the key
- CDFG binding can be carried out through binding DFG’s one by one and then sharing resources among these DFG bindings if possible
- Simultaneously carrying out high-level synthesis subtasks is a promising direction
  - Simulated annealing
  - Simulated evolution (genetic algorithm)
  - Ant colony algorithm (simulated ants)
- Case study: FCUDA

Next Lecture
- Logic synthesis (1)