18-643 Lecture 6:
Good-for-HW Computation Models

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today: see the temporal and spatial patterns of compute and data access in classic good-for-HW compute models

• Notices
  – Handout #4: lab 1, due noon, 9/25
  – Project status report due each Friday

• Readings (see lecture schedule online)
  – Wikipedia is a good starting point
  – for a textbook treatment see Ch 5 (+ Ch 8, 9, 10) of Reconfigurable Computing by Hauck and Dehon
Structural RTL: Low Level/Full Detailed

• Designer in charge
  – arbitrary control and datapath schemes
  – precise control—when, what, where—at the bit and cycle granularity

  
  With great power comes great responsibility . . .

• RTL synthesis is quite literal
  – little room for timing and structural optimizations
  – faithful to both “necessary” and “artifacts”

  e.g., if a and b mutually exclusive

  how to simplify . . . .

```verilog
always@(posedge c)
  if (a)
    o<=1;
  else if (b)
    o<=2;
```
FSM-D “Design Pattern”

- datapath = “organized” combinational logic and registers to carry out computation (puppet)
- FSM = “stylized” combinational logic and registers for control and sequencing (puppeteer)
Cooperating FSM-Ds

• Partitioning large design into manageable chunks
  – natural decomposition by functionalities
  – inherent concurrency and replications

• Correct decomposition leads to simpler parts but coordination of the parts becomes the challenge
  – synchronization: having two FSM-Ds in the right state at the right time
  – communication: exchange information between FSM-D (requires synchronization)
Crux of RTL Design Difficulty

- We design concurrent FSM-Ds separately
  - liable to forget what one machine does when focused on another

- No language support for coordination
  - no explicit way to say how state transitions of two FSMs (i.e., control) must be related

- Coordination hardcoded into design implicitly
  - leave little room for automatic optimization
  - hard to localize design changes
  - (unless decoupled using request/reply-style handshakes)

Lacks standard interfacing of SoC IP composition
IP-Based Design

• Complexity wall
  – designer productivity grows slower than Moore’s Law on logic capacity
  – diminishing return on scaling design team size
  ⇒ must stop designing individual gates

• Decompose design as a connection of IPs
  – each IP fits in a manageable design complexity
  Bonus, IPs can be reused across projects
  ———— abstraction boundary ————
  – IP integration fits in a manageable design complexity
Systematic Interconnect

• More IPs, more elaborate IPs ⇒ intractable to design wires at bit- and cycle-granularity

• On-chip interconnect standards (e.g. AXI) with *address-mapped* abstraction
  – each *target* IPs assigned an *address* range
  – *initiator* IPs issue *read* (or *write*) transactions to pull (or push) data from (or to) addressed target IP
  – physical realization abstracted from IPs

• Plug-and-play integration of interface-compatible IPs

Network-on-chip ("route data not wires")
What is High-Level?

• Abstract away detail/control from designer
  – pro: need not spell out every detail
  – con: cannot spell out every detail

• Missing details must be filled by someone
  – implied in the abstraction, and/or
  – filled in by the synthesis tool

• To be meaningful
  – reduce work, and/or
  – improve outcome

In HW practice, low tolerance for degraded outcome regardless of ease
Good-for-HW Compute Model Examples

- Systolic Array
- Data Parallel
- Dataflow
- Stream Processing
- Commonalities
  - reduce design complexity/effort
  - supports scalable parallelism under simplified global coordination (by imposing a “structure”)
  - allows straightforward, efficient HW mapping
  - BUT, doesn’t work for all problems

These models are not tied-to HW or SW
Good compute models distilled from good design patterns

- Both temporal and spatial patterns in
  - computation
  - synchronization
  - data buffering
  - data movement

What is allowed? uniformity? complexity?

- What makes it good fit with hardware?
- What makes it good fit with application?
- What limits its generality?
Systolic Array

• An array of PEs (imagine each an FSM or FSM-D)
  – strictly, PEs are identical; cannot know the size of the array or position in the array
  – could generalize to other structured topologies

• Scope of design is a PE
  – do same thing in every position
  – localized neighbor-only interactions
    (no global signals or wires)

• Each PE in each round
  – exchange bounded data with direct neighbors
  – perform bounded compute on fixed local storage
E.g. Matrix-Matrix Multiplication

\[
\begin{align*}
&\text{a}=\text{nan}; \\
&\text{b}=\text{nan}; \\
&\text{accum}=0; \\
&\text{For each pulse} \{ \\
&\quad \text{send-}W(\text{a}); \text{send-}S(\text{b}); \\
&\quad \text{a}=\text{rcv-E}(); \text{b}=\text{rcv-N}(); \\
&\quad \text{if } (\text{a}!=\text{nan}) \\
&\quad \quad \text{accum}=\text{a}*\text{b}+\text{accum}; \\
&\} 
\end{align*}
\]

- Works for any \( n \)
- Only stores 3 vals per PE
- If \( N>n \), emulate at \( N^2/n^2 \) slowdown

\[
\begin{align*}
&A_0 & & & & & & & & \text{A}_1 \\
&\text{nan} & & & & & & & & \text{nan} \\
&\text{B}_0 & & & & & & & & \text{B}_1 \\
&\text{0,0} & & & & & & & & \text{col} \\
&\text{0,0} & & & & & & & & \text{0,0} \to \text{row}
\end{align*}
\]

\[
\begin{align*}
&\text{A}_0 & & & & & & & & \text{A}_1 \\
&\text{nan} & & & & & & & & \text{nan} \\
&\text{B}_0 & & & & & & & & \text{B}_1 \\
&\text{0,0} & & & & & & & & \text{col} \\
&\text{0,0} & & & & & & & & \text{0,0} \to \text{row}
\end{align*}
\]
Does the last slide come to mind when you see??

```c
float A[N][N], B[N][N], C[N][N];

for(int i=0; i<N; i++) {
    for(int j=0; j<N; j++) {
        for(int k=0; k<N; k++) {
            C[i][j]=C[i][j]+A[i][k]*B[k][j];
        }
    }
}
```
Why systolic array good for HW?

• Parallel and scalable in nature
  – can efficiently emulate key aspects of stream processing and data-parallel
  – easy to build corresponding HW on VLSI (especially 1D and 2D arrays)
• No global communication
• Scope of design/analysis/debug is 1 FSM-D
• Great when it works
  – linear algebra, sorting, FFTs
  – works more often than you think
  – but clearly not a good fit for every problem
Data Parallelism

- Same work on disjoint sets of data—abundant in linear algebra behind scientific/numerical apps
- Example: AXPY (from Level 1 Basic Linear Algebra Subroutine)

\[
\mathbf{Y} = a\mathbf{X} + \mathbf{Y} = \begin{cases} 
\text{for}(i=0; \ i<N; \ i++) \ { } \ { } \ { } \ { } \ { } \ { } \ { } \ { } \ { } \ { } \ { } \ { } \ { } \ { } \\
\quad \mathbf{Y}[i] = a\mathbf{X}[i] + \mathbf{Y}[i] 
\end{cases}
\]

- \(\mathbf{Y}\) and \(\mathbf{X}\) are vectors
- same operations repeated on each \(\mathbf{Y}[i]\) and \(\mathbf{X}[i]\)
- iteration \(i\) does not touch \(\mathbf{Y}[j]\) and \(\mathbf{X}[j], i \neq j\)

How to exploit data parallelism in HW?
Data Parallel Execution

```c
for(i=0; i<N; i++) {
    C[i]=foo(A[i], B[i])
}
```

- Instantiate \( k \) copies of the hardware unit `foo` to process \( k \) iterations of the loop in parallel
Pipelined Execution

```c
for(i=0; i<N; i++) {
    C[i]=foo(A[i], B[i])
}
```

- Build a deeply pipelined (high-frequency) version of `foo()`

Pipelining also works best when repeating identical and independent compute
E.g. SIMD Matrix-Vector Mult

// Each of the $P$ threads is responsible for
// $M/P$ rows of $A$; self is thread id
for ($i = self * M/P; i < ((self+1)*M/P); i++) {
    $y[i] = 0;
    for ($j = 0; j < N; j++) {
        $y[i] += A[i][j] * x[j];
    }
}

How to structure memory
and array layout?

seems wasteful to each thread to read each $x[]$
$M/P$ times
E.g. Vectorized Matrix-Vector Mult

Repeat for each row of $A$

```
LV V1, Rx ; load vector $x$
LV V2, Ra ; load i’th row of $A$
MULV V3,V2,V1 ; element-wise mult
“reduce” F0, V3 ; sum elements to scalar
S.D Ry, F0 ; store scalar result
```

No such instruction allowed (hint: is “reduce” data-parallel? what is II of MULV vs “reduce”?)

$$\mathbf{y} = \mathbf{A} \mathbf{x}$$
Aside: Vector Chaining

Visualize true (long) vectors “flowing” through the datapath as stream of elements, not as bulk objects.
E.g. Vectorized Matrix-Vector Mult

Repeat for each column of A

- LVWS V0,(Ra,Rs) ; load-strided i’th col of A
- L.D F0,Rx ; load i’th element of x
- MULVS.D V1,V0,F0 ; vector-scalar mult
- ADDV.D Vy,Vy,V1 ; element-wise add

Above is analogous (when/what/where) to the SIMD code
Why is data-parallel good-for-HW?

• Simplest but highly restricted parallelism
• Open to mixed implementation interpretations
  – SIMD parallelism +
  – (deep) pipeline parallelism
• Great when it works
  – important form of parallelism for scientific and numerical computing
  – but clearly not a good fit for every problem
Dataflow Graphs

- Consider a von Neumann program
  - what is the significance of the program order?
  - what is the significance of the storage locations?

- Dataflow operation ordering and timing implied in data dependence
  - instruction specifies who receives the result
  - operation executes when all operands received
  - “source” vs “intermediate” representation

(There is a lot more to this, e.g., loops, fxns)
Token Passing Execution

- **fan-in**
- **fan-out**
- **switch** (conditional)
- **merge** (conditional)

“fire” output tokens when all required input present

consider multi-, variable-cycle ops and links
Synchronous Dataflow

- Operate on flows (sequence of data values)
  - i.e., \(X = \{ x_1, x_2, x_3, \ldots \}\), \(1 = \{1, 1, 1, 1, \ldots\}\)
- Flow operators, e.g., switch, merge, duplicate
- Temporal operators, e.g. \(\text{pre}(X) = \{\text{nil}, x_1, x_2, x_3, \ldots\}\)

Fig 1, Halbwachs, et al., The Synchronous Data Flow Programming Language LUSTRE

Function vs Execution vs Implementation
What do you make of this?

node ACCUM(init, incVal: int; reset: bool) returns (n: int);
let
    n = init -> if reset then init else pre(n) + incr
tel

pre({e₁,e₂,e₃, ....}) is {nil, e₁,e₂,e₃, ....}
{e₁,e₂,e₃, ....}->{f₁,f₂,f₃, ....} is {e₁,f₂,f₃,f₄ ....}
Try Simulink in Vitis Model Composer

[Figure 8.1: “Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation”]
Why is dataflow good-for-HW?

• Naturally express fine-grain, implicit parallelism
  Many variations, asynchronous, dynamic, . . .

• Loose coupling between operators
  – synchronize by order in flow, not cycle or time
  – no imposed operation ordering
  – no global synchronization/communications

• Declarative nature permits implementation flexibilities

• Great when it works
  – excellent match with signal processing
  – but clearly not a good fit for every problem
Stream Processing

• Related to dataflow
  – operate on data in sequence (no random access)
  – repeat same operation on data in a stream
• Emphasis on IPs and their composition
  – design in terms of composing valid stream-to-stream transformations
  – simple, elastic, plug-and-play “interface”
• More flexible rules
  – input and output flows need not be synchronized
  – operator can have a fixed amount of memory
    • buffer/compute over a window of values
    • carry dependencies over values in a stream
Regular and Data-Independent: E.g., Vision Processing Pipeline

Color-based object tracking (linear pipeline, 4 stages)

- Camera
- 1. Gaussian blur
- 2. Color threshold
- 3. Color threshold
- 4. Color threshold
- Display

Background subtraction (2-branch pipeline, 6 stages)

- Camera
- 1. Duplicate
- 2. Gaussian blur
- 3. Background subtraction
- 4. Synchronizer
- 5. Merge
- 6. Paint
- Display

Corner + edge detection (3-branch pipeline, 10 stages)

- Camera
- 1. Duplicate
- 2. Duplicate
- 3. Corner detection
- 4. Synchronizer
- 5. Edge detection
- 6. Synchronizer
- 7. Synchronizer
- 8. Merge
- 9. Merge
- 10. Paint
- Display
Irregular and Data-Dependent

E.g., Network Packet Processing

- ethrnt TCP flow reassembly
- "fast pattern" matching
- 2\textsuperscript{nd} filtering
- Offloading to CPU
- CPU full matching

https://github.com/cmu-snap/pigasus
Commonalities Revisited

• Parallelism under simplified global coordination
  – enforced regularity
  – asynchronous coupling
• Straightforward efficient mapping to hardware
  – low performance overhead
  – low resource overhead
  – high resource utilization
• Simplify design without interfering with quality
• But only works on specific problem patterns
Parting Thoughts:
Conflict between High-Level and Generality

insist on quality

high-level: tools know better than you

RTL synthesis: general-purpose but special handling of structures like FSM, arith, etc.

place-and-route: works the same no matter what design

18-643-F23-L06-S34, James C. Hoe, CMU/ECE/CALCM, ©2023