Parallelism

Optimization in hardware design involves using parallelism to trade between cost and performance.

- Example: Student final grade calculation:
  \[
  \text{grade} = 0.2 \times m1 + 0.2 \times m2 + 0.2 \times m3 + 0.4 \times \text{project};
  \]
  \[
  \text{write grade};
  \]
- High performance hardware implementation:
  \[
  \begin{align*}
  &0.2 \times m1 + 0.2 \times m2 + 0.2 \times m3 + 0.4 \times \text{project}; \\
  &\text{write grade;}
  \end{align*}
  \]

As many operations as possible are done in parallel.

Parallelism

- Is there a lower cost hardware implementation? Different tree organization?
- Can factor out multiply by 0.2:
  \[
  \begin{align*}
  &0.2 \times m1 + 0.2 \times m2 + 0.4 \times \text{project}; \\
  &\text{grade}
  \end{align*}
  \]
- How about sharing operators (multipliers and adders)?

HW versus SW

- This time-multiplexed ALU approach is very similar to what a conventional software version would accomplish:
  \[
  \begin{align*}
  &\text{add } r2, r1, r3 \\
  &\text{add } r2, r2, r4 \\
  &\text{mult } r2, r4, r5
  \end{align*}
  \]
- CPUs time-multiplex function units (ALUs, etc.)
- This model matches our tendency to express computation sequentially - even though many naturally contain parallelism.
- Our programming languages also strengthen this tendency.
- In hardware we have the ability to exploit problem parallelism - gives us a “knob” on performance/cost.
- Maybe best to express computations as abstract computations graphs (rather than “programs”) - should lead to wider range of implementations.
- Note: modern processors spend much of their cost budget attempting to restore execution parallelism: “super-scalar execution”.

Power Consumption

- The critical factor is the total energy for a particular computation.
- As we trade cost for performance what happens to energy?
  \[
  \begin{align*}
  &4 \times E_{\text{elec}} + 3 \times E_{\text{fused}} \\
  &2 \times E_{\text{elec}} + 5 \times E_{\text{fused}} \\
  &2 \times E_{\text{elec}} + 3 \times E_{\text{fused}} + E_{\text{leak}} + E_{\text{leak}}
  \end{align*}
  \]
- The lowest energy consumer is the solution that minimizes cost without time multiplexing operations.
Optimizing Iterative Computations

- Hardware implementations of computations almost always involve looping. Why?
- Is this true with software?
- Are there programs without loops?
  - Maybe in "through away" code.
- We probably would not bother building such a thing into hardware, would we?
  - (FPGA may change this.)
- Fact is, our computations are closely tied to loops. Almost all our HW includes some looping mechanism.
- What do we use looping for?

Types of loops:
1) Looping over input data (streaming):
   - ex: MP3 player, video compressor, music synthesizer.
2) Looping over memory data
   - ex: vector inner product, matrix multiply, list processing
- These two are really very similar. 1) is often turned into 2) by buffering up input data, and processing "offline". Even for "online" processing, buffers are used to smooth out temporary rate mismatches.
3) CPUs are one big loop.
   - Instruction fetch ⇒ execute ⇒ Instruction fetch ⇒ execute ⇒ ...
   - but change their personality with each iteration.
4) Others?

Loops offer more opportunity for parallelism by executing more than one iteration at once, through parallel iteration execution &/or pipelining.

Pipelining

- With looping usually we are less interested in the latency of one iteration and more in the loop execution rate, or throughput.
- These can be different due to parallel iteration execution &/or pipelining.
- Pipelining review from CS61C:
  Analog to washing clothes:
  step 1: wash (20 minutes)  
  step 2: dry (20 minutes)  
  step 3: fold (60 minutes)
  60 minutes x 4 loads ⇒ 4 hours
  wash load1  load2  load3  load4
dry            load1  load2  load3  load4
fold          load1  load2  load3  load4
20 min overlapped ⇒ 2 hours

General principle:
- Cut the block into pieces (stages) and separate with registers:
- CL block produces a new result every 5ns instead of every 9ns.

Limits on Pipelining

- Without FF overhead, throughput improvement \( \alpha \) # of stages.
- After many stages are added. FF overhead begins to dominate:

Other limiters:
- clock skew contributes to clock overhead
- unequal stages
- FFs dominate cost
- clock distribution power consumption
- feedback (dependencies between loop iterations)
Example

- $F(x) = y_i + a x_i^2 + b x_i + c$

- Computation graph:

- $x$ and $y$ are assumed to be "streams".

- Divide into 3 (nearly) equal stages.

- Insert pipeline registers at dashed lines.

- Can we pipeline basic operators?

Pipelining Loops with Feedback

**Loop carry dependency**

- Example 1:
  
  $y_i = y_{i-1} + x_i + a$

  Can we "cut" the feedback?

  
  
  
  
  
  
  Add is associative and commutive.

  $y_i = a + x_i + y_{i-1}$

- Pipelining is limited to 2 stages.

- Shorten the feedback loop:

  
  
  
  
  
  
  Still need 2 cycles/iteration

Pipelining Loops with Feedback

**Example 2 Corrected Version**

- Example 2:
  
  $y_i = a y_{i-1} + x_i + b$

  The chart as drawn does not match the graph.

  This chart can’t work. Why?

  The chart implies a delay (flip-flop) after the $a y_{i-1}$ multiply, plus the original after the 2nd add.

  Therefore $y_i = f(y_{i-2})$ instead of $y_{i-1}$.

  
  
  
  
  
  
  Alternative is to move flip-flop to after multiply, but same critical path.
**“C-slow” Technique**

- We can try to fill in the “holes” in the chart with another (independent) computation:

  \[
  \begin{array}{c|c|c|c|c}
  \text{add} & x_i+b & x_{i+1}+b & x_{i+2}+b \\
  \text{mult} & ay_i & ay_i & ay_{i+1} \\
  \text{add} & y_i & y_{i+1} & y_{i+2} \\
  \end{array}
  \]

  If we have a second similar computation, can interleave it with the first:

  \[
  \begin{align*}
  x^1 & = y^1 = a^1 y^1_{i-1} + x^1_i + b^1_i \\
  x^2 & = y^2 = a^2 y^2_{i-1} + x^2_i + b^2_i 
  \end{align*}
  \]

- Here the feedback depth≥2 cycles (we say C=2).
- Each loop has throughput of F/C. But the aggregate throughput is F.
- With this technique we could pipeline even deeper, assuming we could supply C independent streams.

**Beyond Pipelining - SIMD Parallelism**

- An obvious way to exploit more parallelism from loops is to make multiple instances of the loop execution data-path and run them in parallel sharing the same controller.
- For P instances, throughput improves by a factor of P.
- example: \( y_i = f(x) \)

  \[
  \begin{array}{c|c|c|c|c}
  x_i & x_i+1 & x_i+2 & x_i+3 \\
  y_i & y_{i+1} & y_{i+2} & y_{i+3} \\
  \end{array}
  \]

  Usually called SIMD parallelism. Single Instruction Multiple Data

- Assumes the next 4 \( x \) values available at once. The validity of this assumption depends on the ratio of \( f \) repeat rate to input rate (or memory bandwidth).
- Cost \( \propto P \). Usually, much higher than for pipelining. However, potentially provides a high speedup. Often applied after pipelining.
- Limited, once again, by loop carry dependencies. Feedback translates to dependencies between parallel data-paths.

**SIMD Parallelism with Feedback**

- From earlier: \( y_i = a y_{i-1} + x_i + b \)

  \[
  \begin{array}{c|c|c|c|c|c|c|c}
  x_i & + & x_i & + & x_i & + & x_i & + \\
  y_i & + & y_i & + & y_i & + & y_i & + \\
  \end{array}
  \]

- End up with “carry ripple” situation.
- Could employ look-ahead / parallel-prefix optimization techniques to speed up propagation.